

# NVIDIA CUDA



## Introduction to Accelerated Computing using CUDA

2021.08.04

Karthik Bala

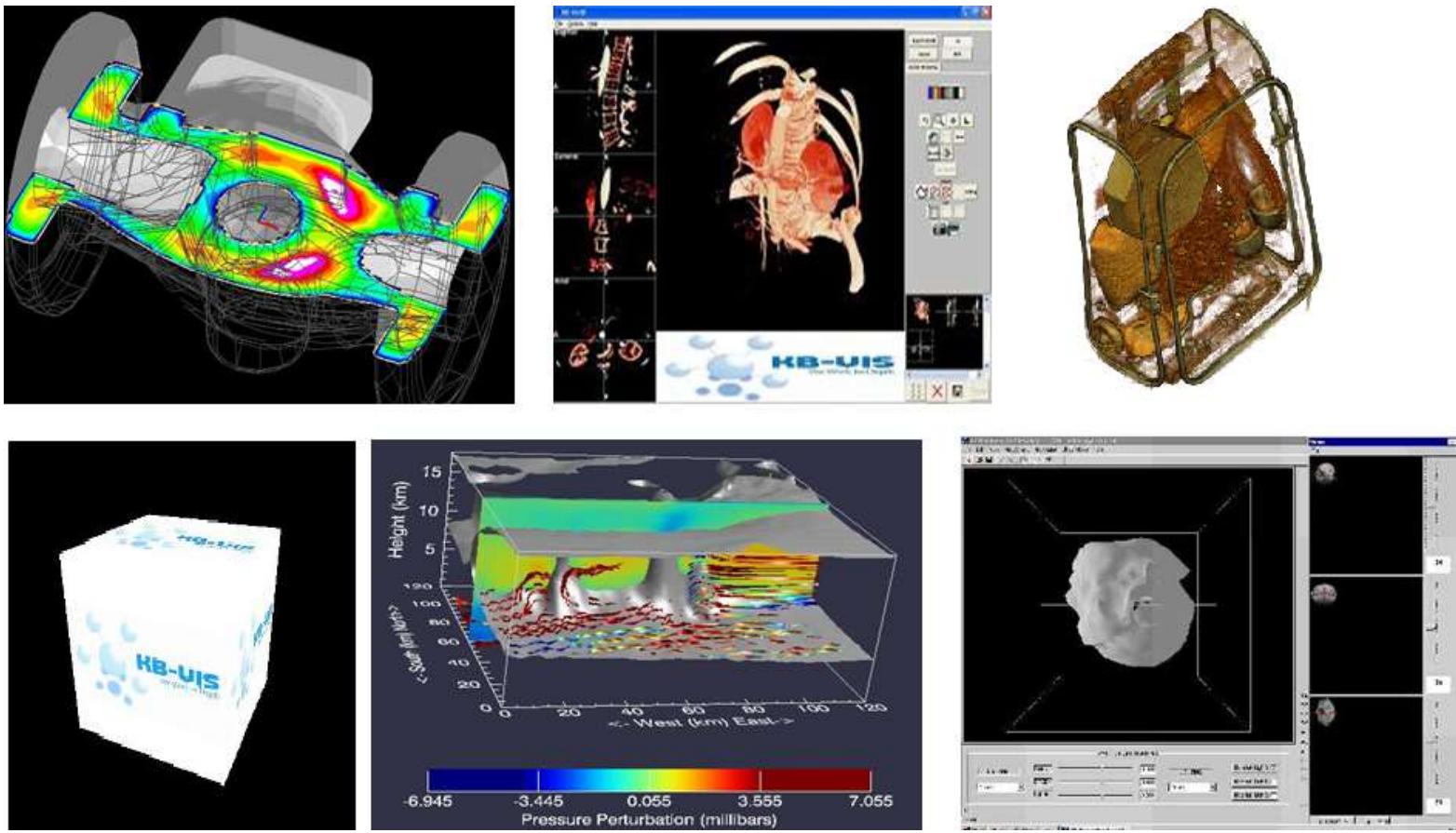
KBVIS Software Pty. Ltd., Brisbane, Australia

<http://kbvis.com>

3d@kbvis.com

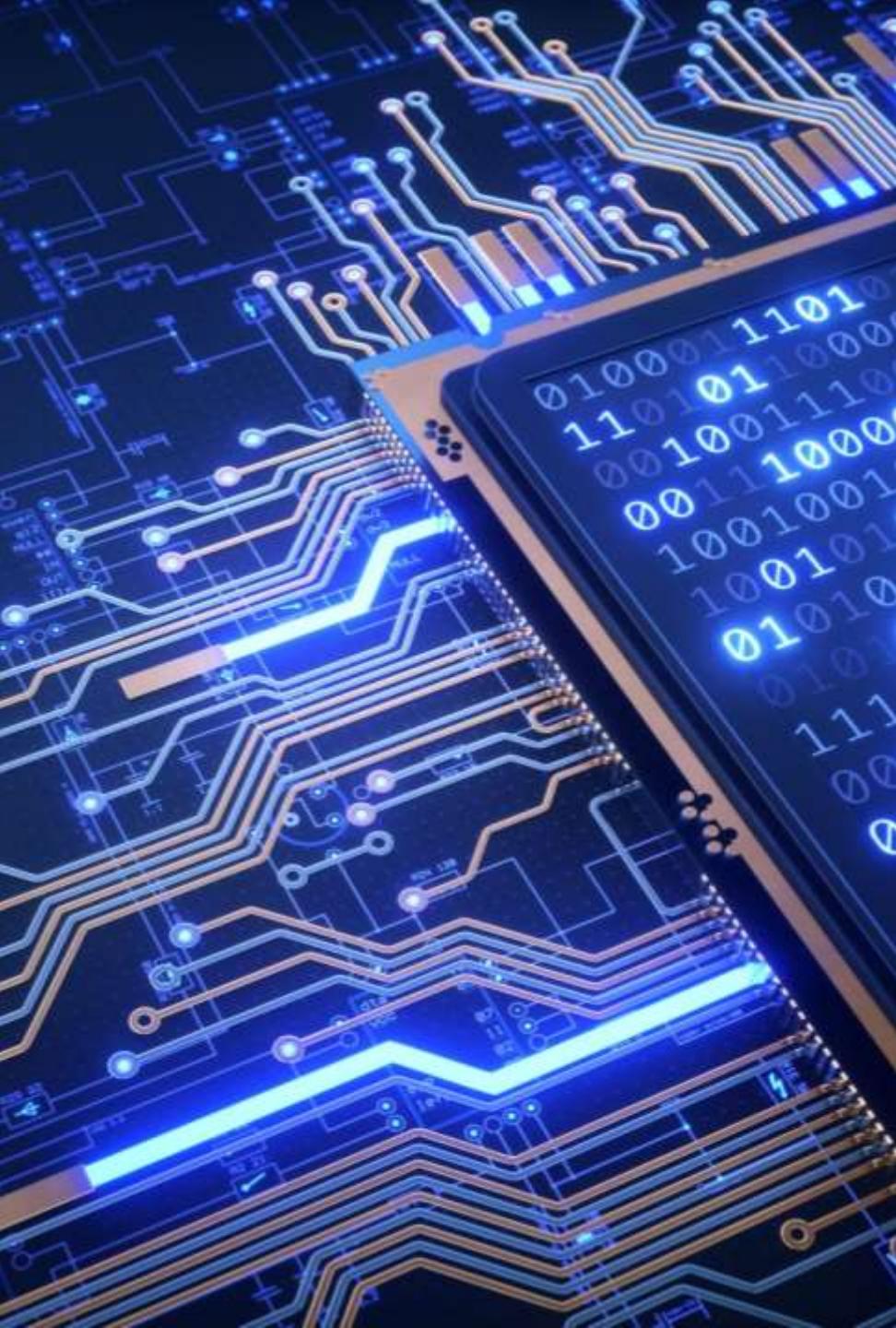


# About KBVIS



## Clients





# DAY 1

---

- CUDA - Background, History, Motivation
- Introduction to Massively Parallel Computing
- Introduction to GPU-based Programming
- CUDA Overview
- Getting Started with CUDA
  - CUDA SDK, Samples (Windows) - Overview
  - Nvidia Courseware, VM usage, Exercises - Overview
- Anatomy of a CUDA Program
  - Introduction to CUDA Syntax
  - “Hello World”
  - Vector Addition

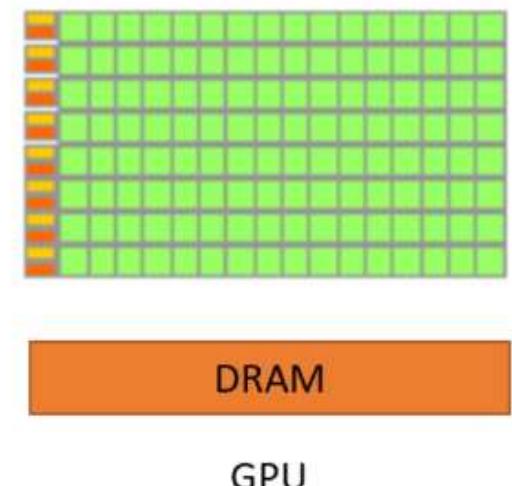
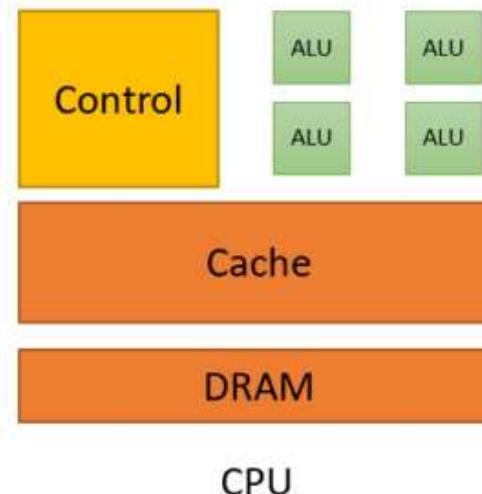
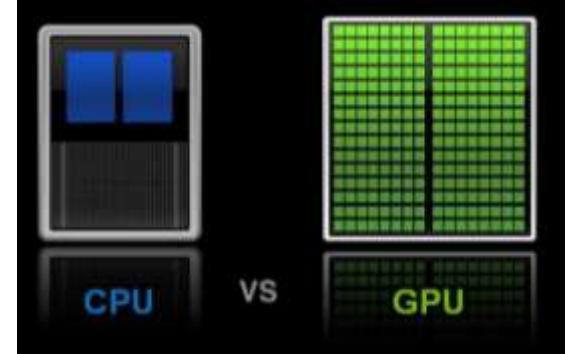
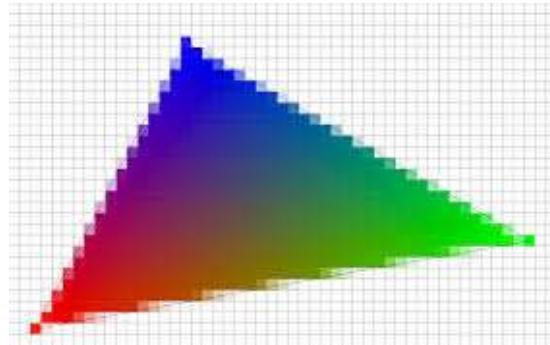
# Agenda (tentative)

- DAY 2
  - CUDA Memory Model
  - Unified Memory
  - Thread Hierarchy
  - Introduction to Shared Memory
- DAY 3
  - Shared Memory
  - Memory - Best Practices
- DAY 4
  - Asynchronicity
  - Streams
  - Dynamic Parallelism
  - CUDA SDK Samples
- DAY 5
  - Scan/Reduction
  - Atomic operations
  - CUDA SDK Samples
- DAY 6
  - Image Processing
  - Textures
  - Graphics Interop
  - CUDA SDK Samples
- DAY 7
  - Profiling and Debugging
  - Nsight
- DAY 8
  - Advanced Topics – Quick Peek
  - Tools/Libraries
  - Questions / Recap

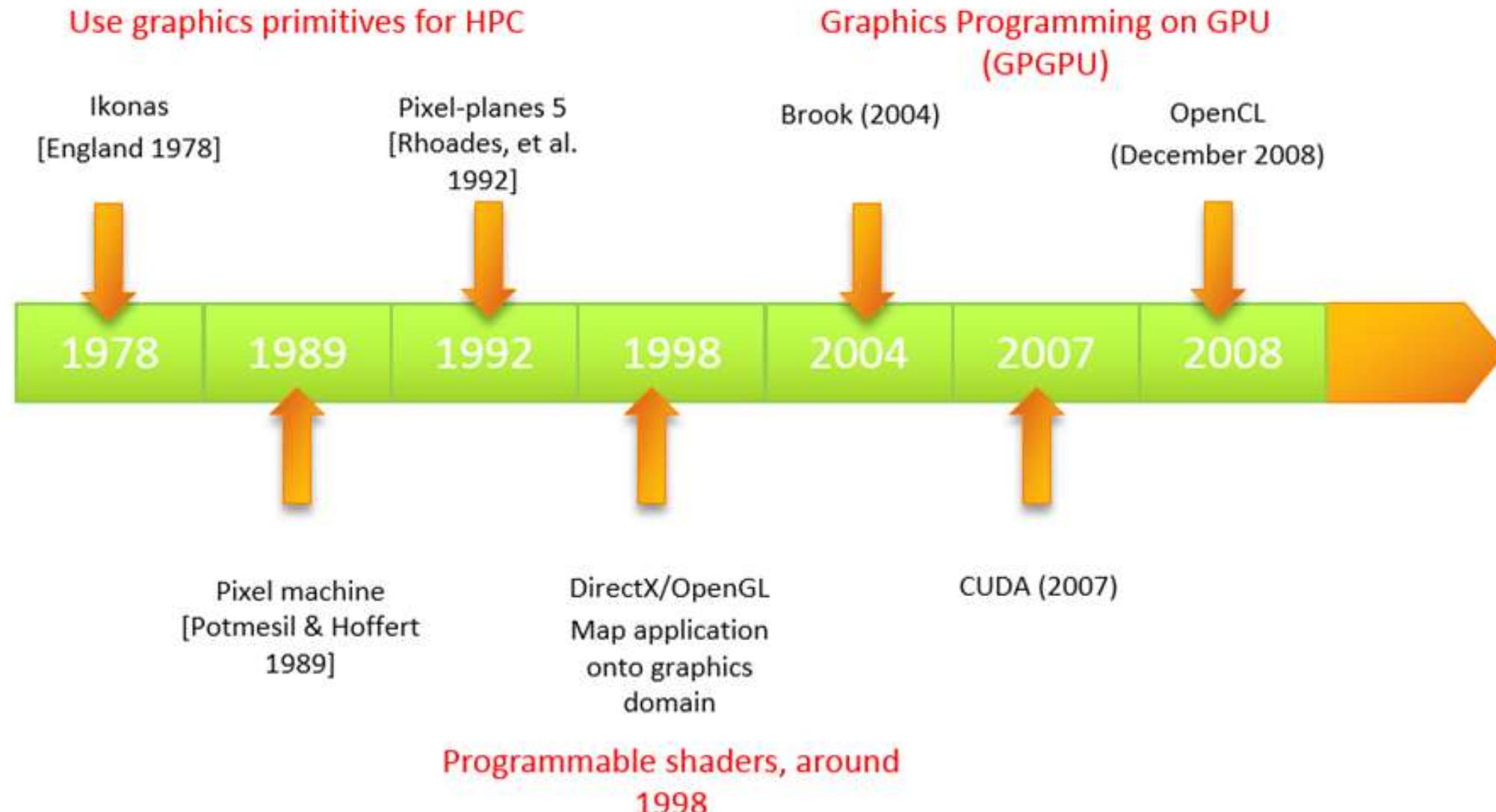
Feel free to suggest changes/additions!

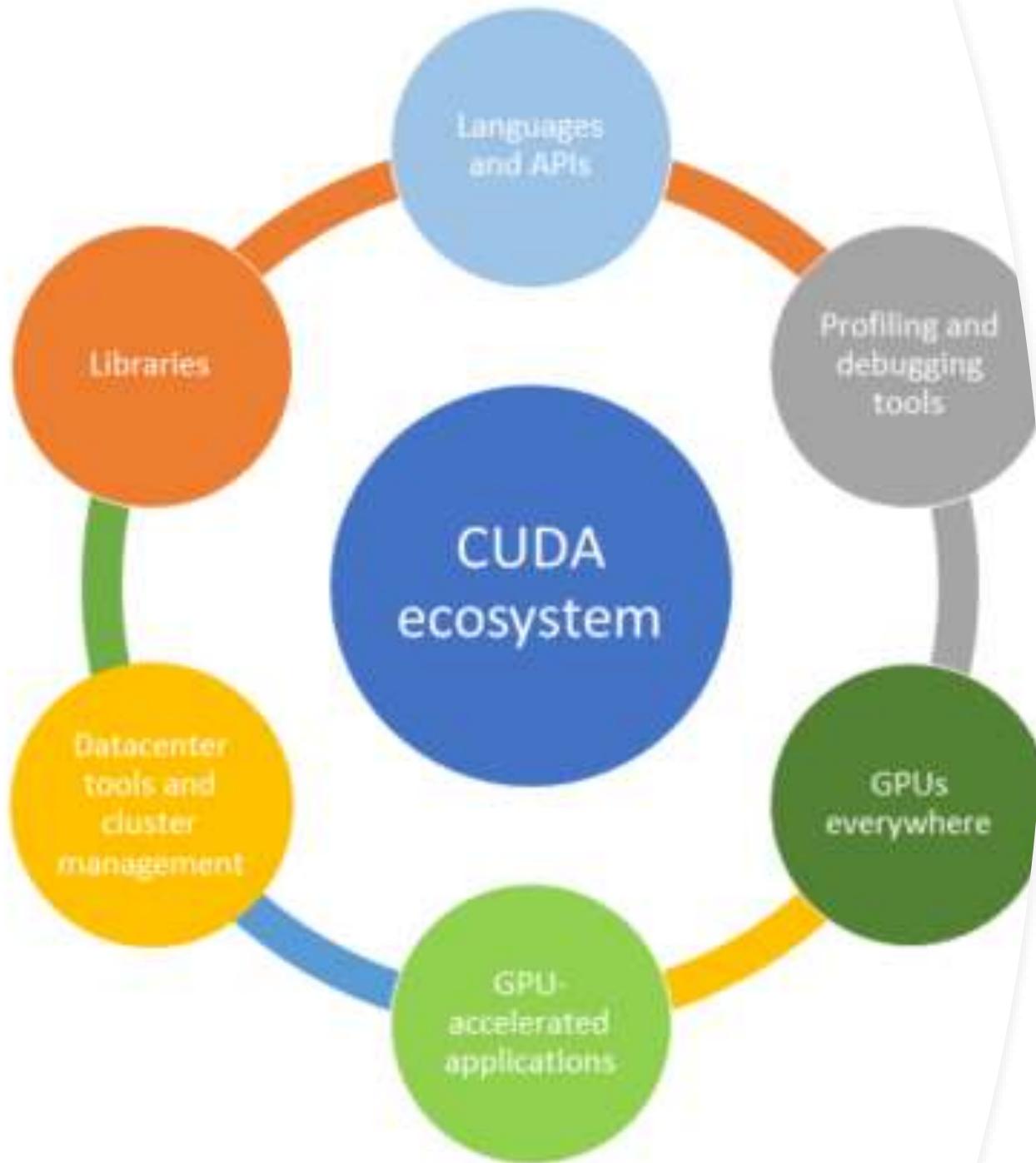
# Dawn of GPGPU

- Since 2005 – Shift to Parallelism rather than Transistor-density
- GPUs are designed for massively parallel computation:
  - Vertex and Fragment Shaders run in parallel – data locality (Cg, GLSL, HLSL)
  - Graphics/Shading essentially floating point computation
- GPGPU aimed to exploit graphics hardware for computation
- CPU – high complexity, low latency
- GPU – high throughput, data-centric



# GPGPU Evolution



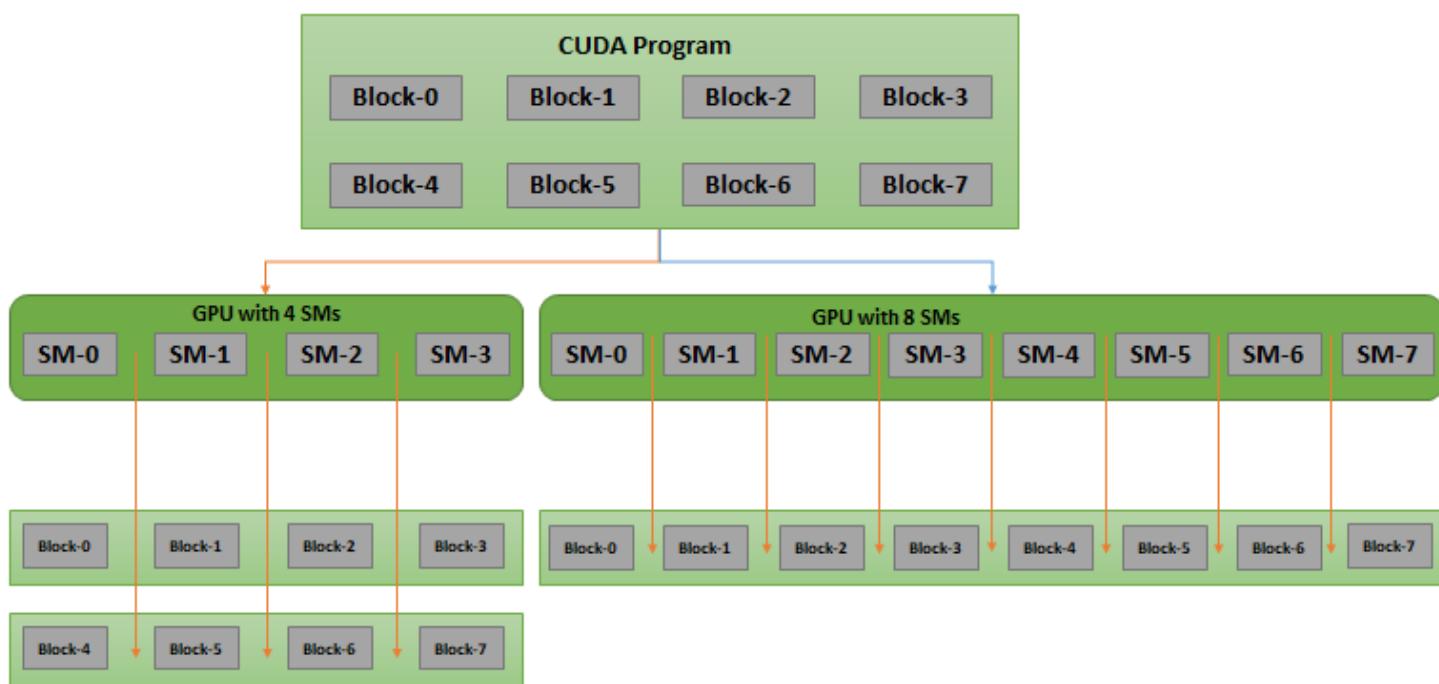


## CUDA Ecosystem

---

- HPC
- Medical Imaging
- Simulation
- Graphics & Visualization
- Computer Vision
- Data Science
- Machine Learning, AI
- Autonomous Vehicles

# Introduction to CUDA



- Launched by Nvidia in 2006
- C-API for Parallel Computation
- Familiar syntax
- Easily scalable across GPUs
- CUDA Toolkit – compiler, toolchain, libraries, runtime



# CUDA Resources

- Developer Zone: <https://developer.nvidia.com/cuda-toolkit>
- Download: <https://developer.nvidia.com/cuda-downloads>
- Documentation:
  - [CUDA Programming Guide \(PDF\)](#)
  - [CUDA Best Practices \(PDF\)](#)
- Training: <https://developer.nvidia.com/accelerated-computing-training>
- Nvidia Online Course Login:  
<https://courses.nvidia.com/courses/course-v1:DLI+C-AC-01+V1/about>

# CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

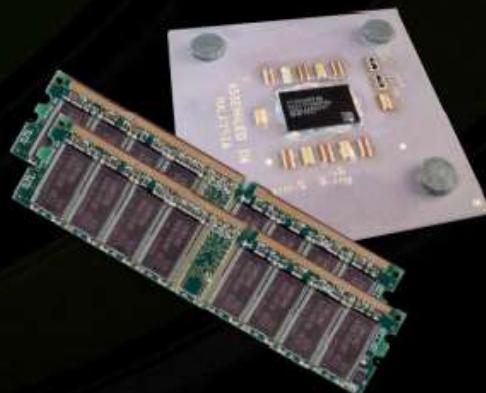
Handling errors

Managing devices

# Heterogeneous Computing



- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host



Device

# Heterogeneous Computing

device code

parallel function

host code

serial function

serial code

parallel code

serial code

```

#include <algorithm>
#include <algorithm>

using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int index = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[index] = in[index];
    if (threadIdx.x < RADIUS) {
        temp[index - RADIUS] = in[index - RADIUS];
        temp[index + BLOCK_SIZE] = in[index + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[index + offset];

    // Store the result
    out[index] = result;
}

void fill(int *in, int n) {
    fill_n(in, n, 1);
}

int main(void) {
    int *in, *out;           // host copies of a, b, c
    int *d_in, *d_out;       // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_n(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_n(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc(&d_in, in_size, cudaMemoryHostToDevice);
    cudaMalloc(&d_out, out_size, cudaMemoryHostToDevice);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<(N/BLOCK_SIZE,BLOCK_SIZE>>)(d_in + RADIUS, d_out + RADIUS);

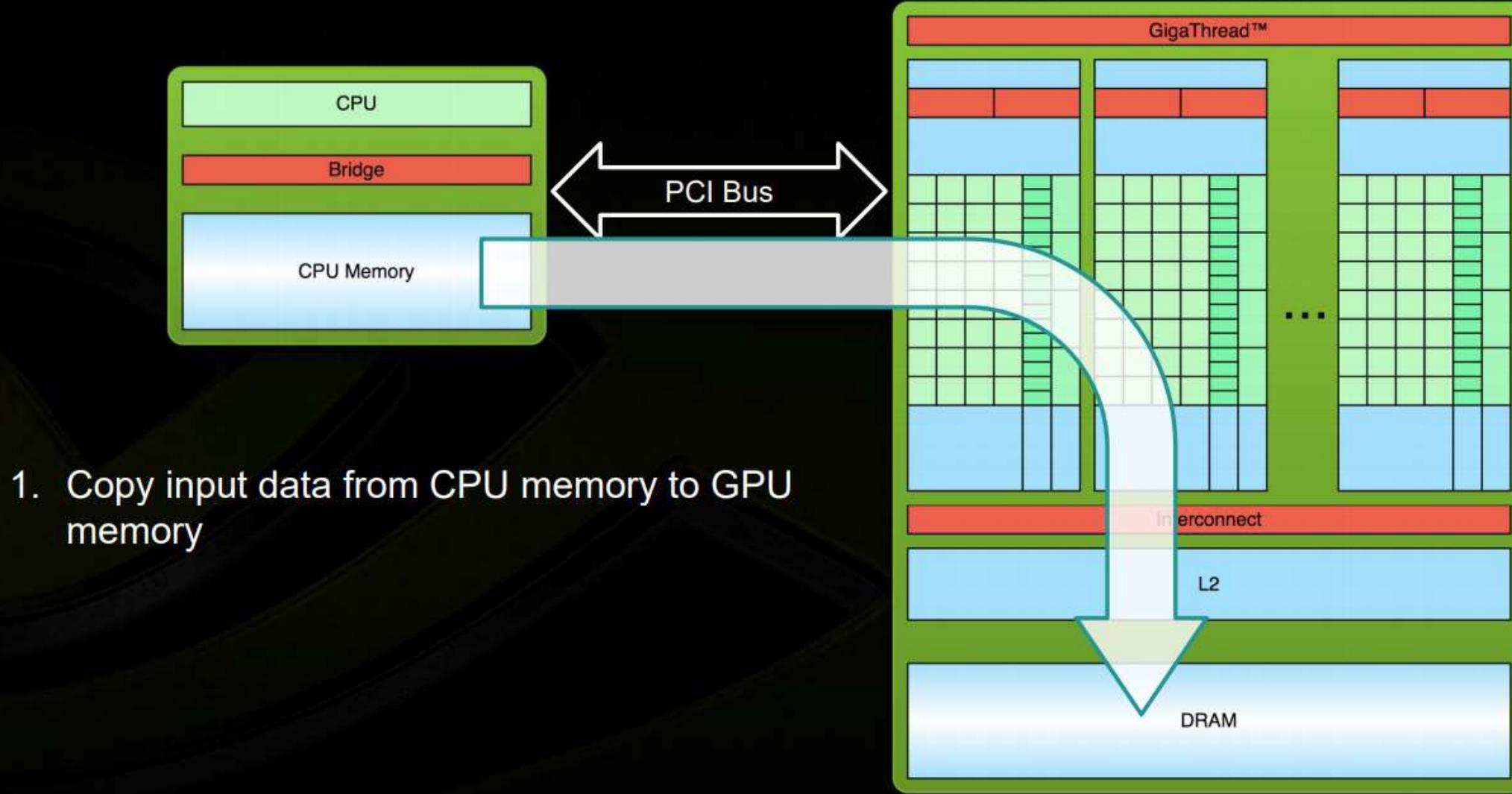
    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}

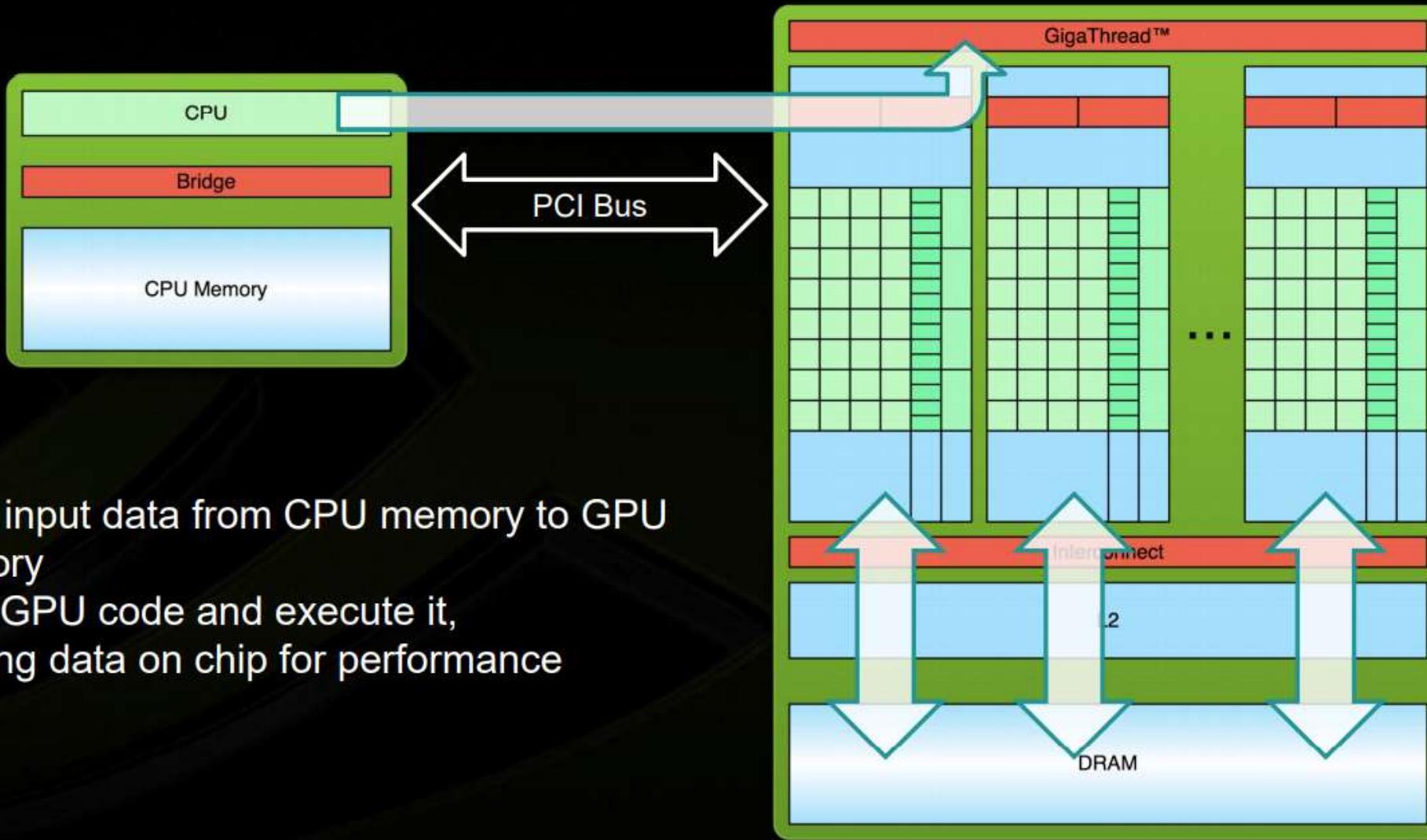
```



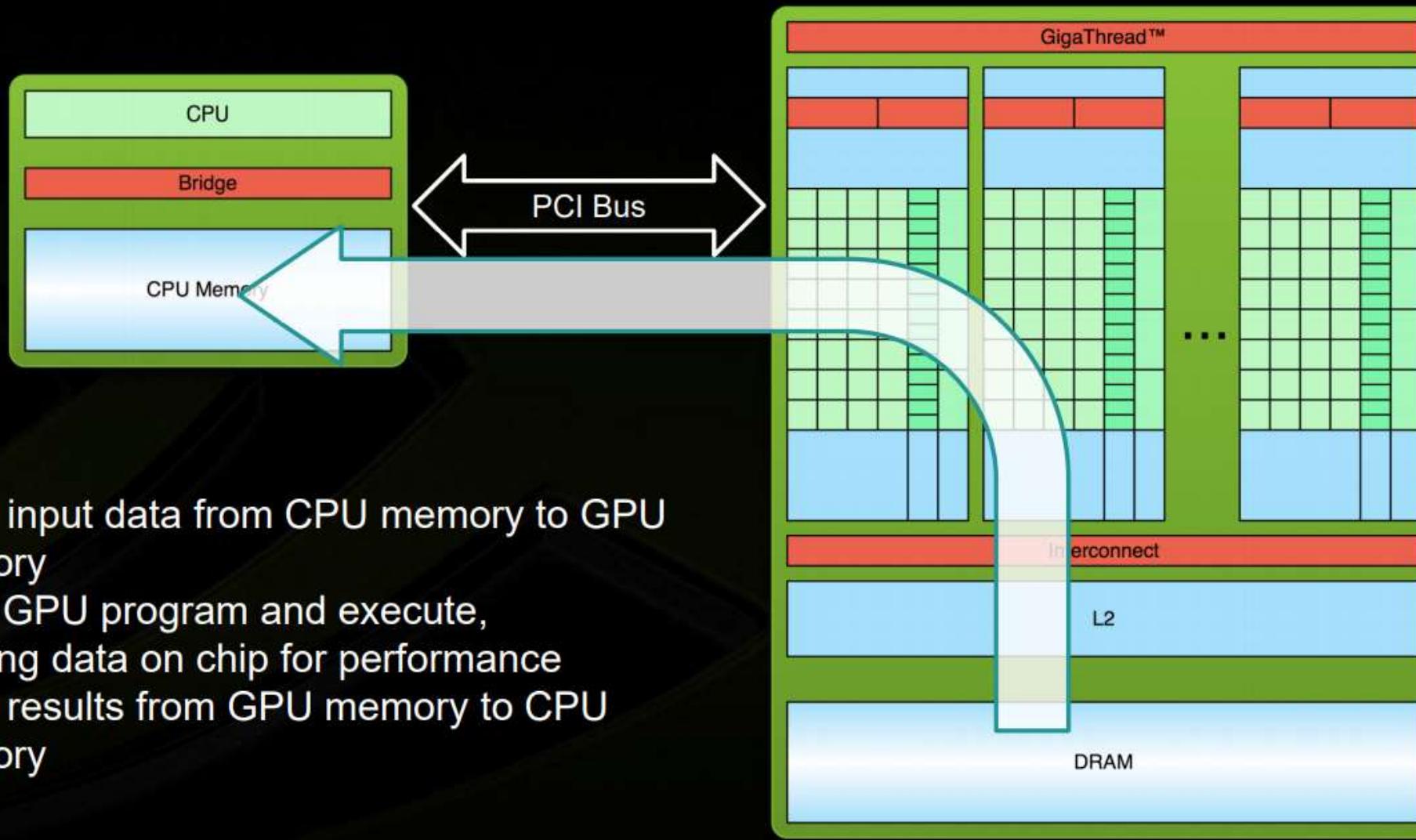
# Simple Processing Flow



# Simple Processing Flow



# Simple Processing Flow





# Hello World!

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc  
hello_world.cu  
$ a.out  
Hello World!  
$
```



# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Two new syntactic elements...

# Hello World! with Device Code



```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- nvcc separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - gcc, cl.exe

# Hello World! with Device Code



```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!



# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

Output:

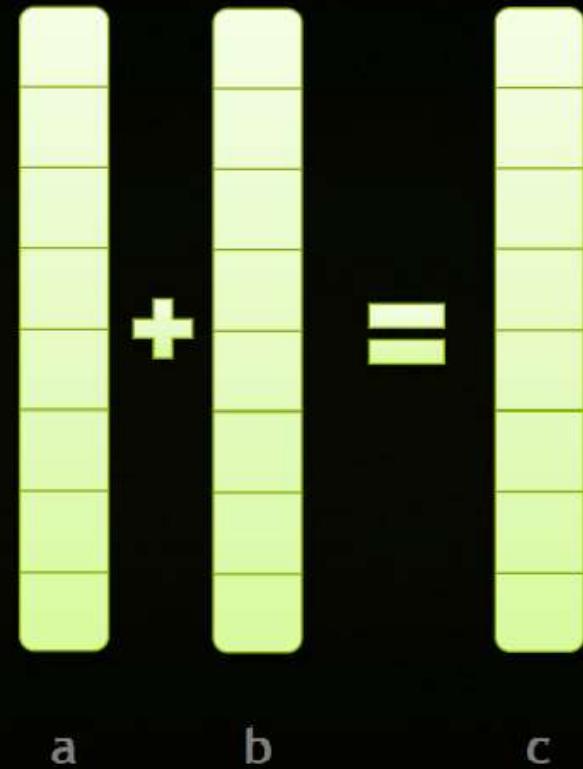
```
$ nvcc hello.cu  
$ a.out  
Hello World!  
$
```

- `mykernel()` does nothing, somewhat anticlimactic!

# Parallel Programming in CUDA C/C++



- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We'll start by adding two integers and build up to vector addition





# Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host



# Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU

# Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`





# Addition on the Device: add()

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Let's take a look at `main()`...



# Addition on the Device: main()

```
int main(void) {
    int a, b, c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                        // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Setup input values
    a = 2;
    b = 7;
```

# Addition on the Device: main()



```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# RUNNING IN PARALLEL

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices



# Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>> () ;  
          ^  
          |  
add<<< N, 1 >>> () ;
```

- Instead of executing add () once, execute N times in parallel

# Vector Addition on the Device



- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **block**
  - The set of blocks is referred to as a **grid**
  - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- By using `blockIdx.x` to index into the array, each block handles a different element of the array



# Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- On the device, each block can execute in parallel:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```



# Vector Addition on the Device: add()

- Returning to our parallelized add() kernel

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- Let's take a look at main()...

# Vector Addition on the Device: main()



```
#define N 512
int main(void) {
    int *a, *b, *c;                      // host copies of a, b, c
    int *d_a, *d_b, *d_c;                  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```



# Vector Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```



# Review (1 of 2)

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU
- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host
- Passing parameters from host code to a device function



# Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch  $N$  copies of `add()` with `add<<<N, 1>>>(...);`
  - Use `blockIdx.x` to access block index



# INTRODUCING THREADS

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

# CUDA Threads



- Terminology: a block can be split into parallel **threads**
- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

- We use `threadIdx.x` instead of `blockIdx.x`
- Need to make one change in `main()`...

# Vector Addition Using Threads: main()



```
#define N 512
int main(void) {
    int *a, *b, *c;                      // host copies of a, b, c
    int *d_a, *d_b, *d_c;                  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition Using Threads: main()



```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```

# COMBINING THREADS AND BLOCKS

## CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- `__syncthreads()`
- Asynchronous operation
- Handling errors
- Managing devices



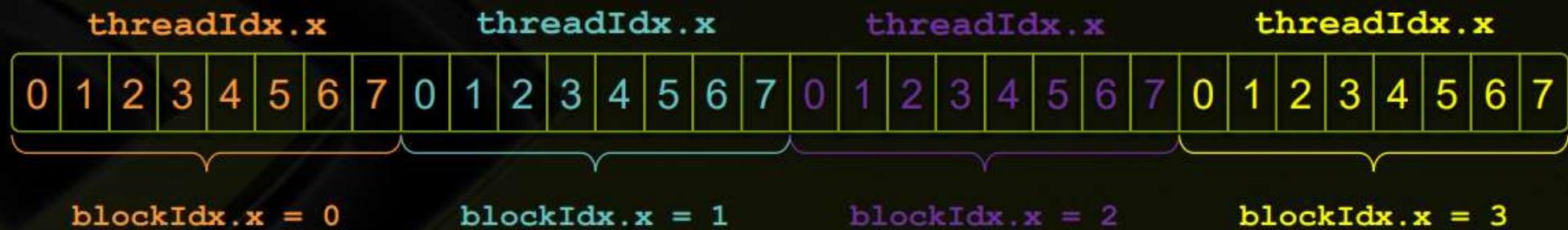
# Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Several blocks with one thread each
  - One block with several threads
- Let's adapt vector addition to use both *blocks* and *threads*
- Why? We'll come to that...
- First let's discuss data indexing...



# Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)



- With  $M$  threads per block, a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

# Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;  
=      5       +      2       * 8;  
= 21;
```

# Vector Addition with Blocks and Threads



- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    c[index] = a[index] + b[index];  
}
```

- What changes need to be made in `main()`?

# Addition with Blocks and Threads: main()



```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Addition with Blocks and Threads: main()



```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```



# Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        c[index] = a[index] + b[index];  
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```



# Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to efficiently:
  - Communicate
  - Synchronize
- To look closer, we need a new example...

# COOPERATING THREADS

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

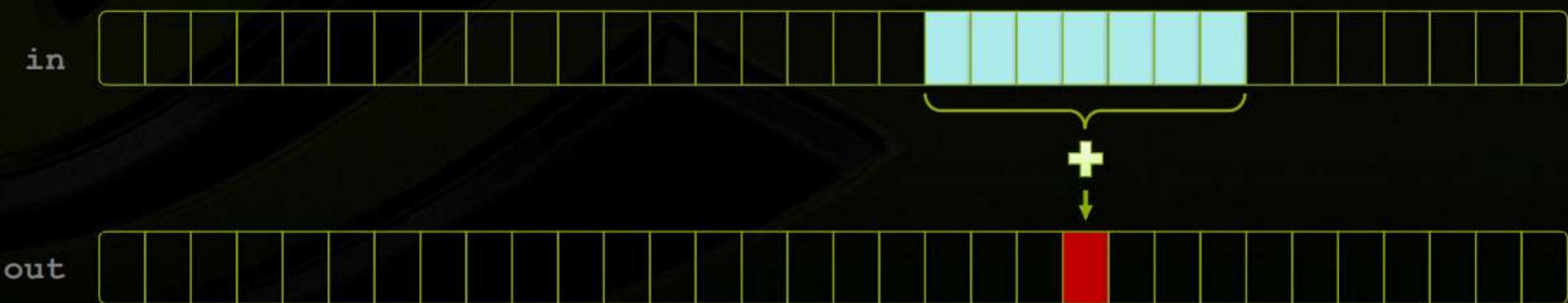
Asynchronous operation

Handling errors

Managing devices

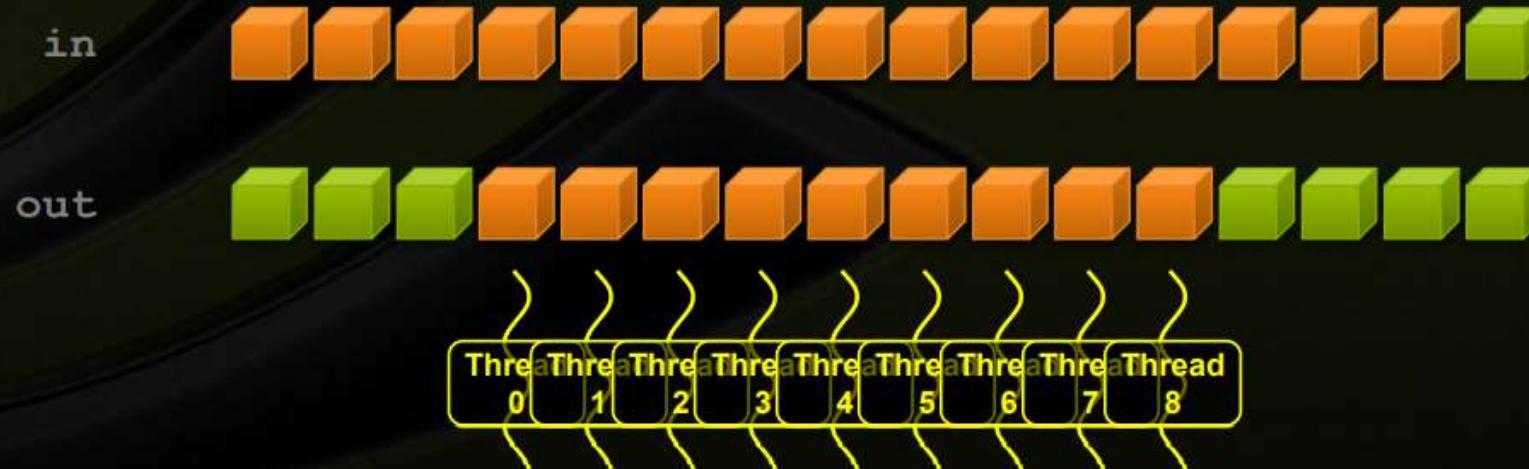
# 1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



# Implementing Within a Block

- Each thread processes one output element
  - `blockDim.x` elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times





# Sharing Data Between Threads

- Terminology: within a block, threads share data via **shared memory**
- Extremely fast on-chip memory
  - By opposition to device memory, referred to as **global memory**
  - Like a user-managed cache
- Declare using **\_\_shared\_\_**, allocated per block
- Data is not visible to threads in other blocks

# Implementing With Shared Memory

- Cache data in shared memory
  - Read  $(\text{blockDim.x} + 2 * \text{radius})$  input elements from global memory to shared memory
  - Compute  $\text{blockDim.x}$  output elements
  - Write  $\text{blockDim.x}$  output elements to global memory
- Each block needs a halo of  $\text{radius}$  elements at each boundary

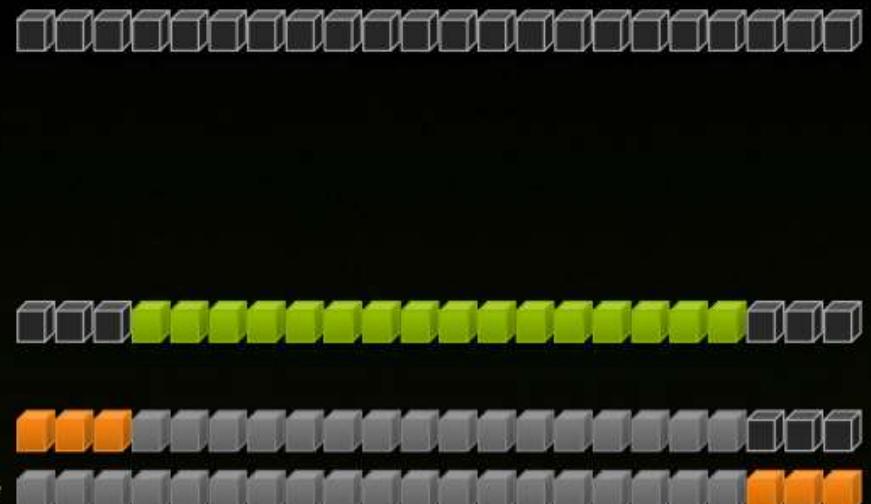


# Stencil Kernel



```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
}
```



# Stencil Kernel



```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```

# Data Race!

- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```
...
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
```

**Store at temp[18]**



**Skipped since threadIdx.x > RADIUS**





# \_\_syncthreads()

- `void __syncthreads();`
- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

# Stencil Kernel



```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
```

# Stencil Kernel



```
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}
```



# Review (1 of 2)

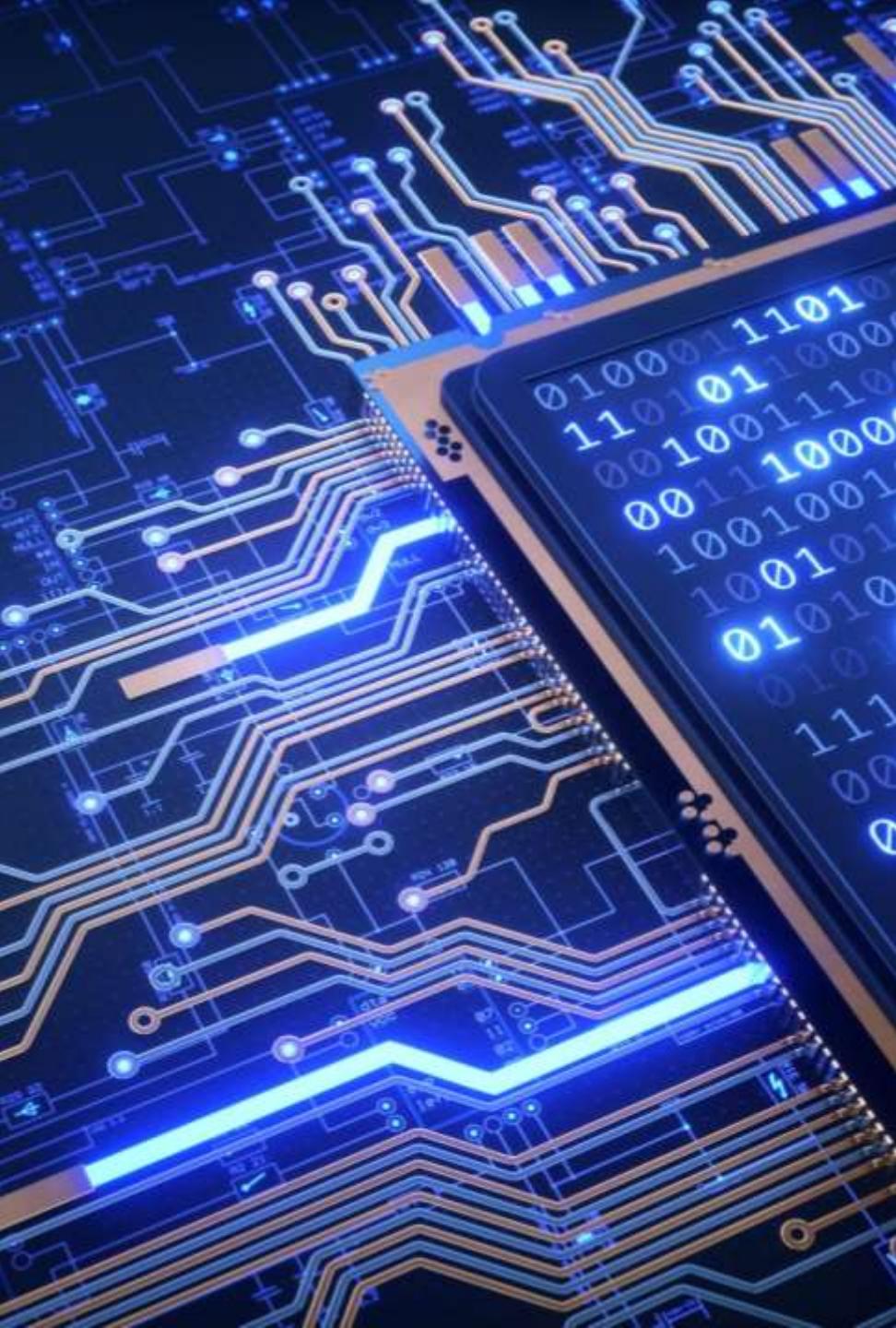
- Launching parallel threads
  - Launch  $N$  blocks with  $M$  threads per block with `kernel<<<N, M>>>(...);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```



## Review (2 of 2)

- Use `_shared_` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
  
- Use `_syncthreads()` as a barrier
  - Use to prevent data hazards



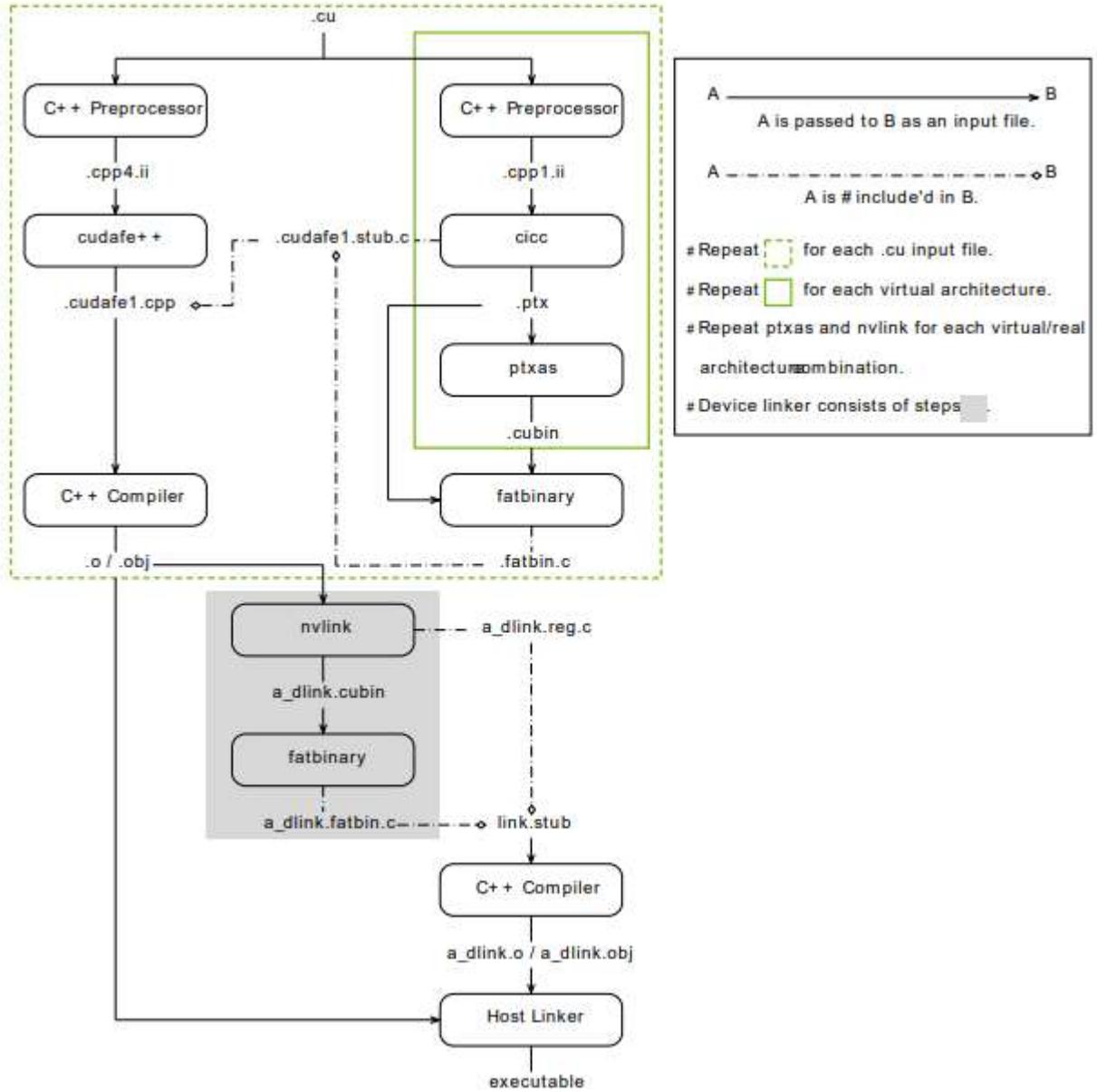
# DAY 2

- CUDA Compilation
- CUDA Memory Model
- Shared Memory
- Unified Memory
- Memory Management
- Example – Matrix Multiplication using Shared Memory
- Introduction to Streams
- Occupancy
- Atomics

Get the updated training content from:  
<https://github.com/kbvis3d/toshiba-cuda-2021>

# CUDA Compilation

- See NVCC Reference for full options:
- [https://docs.nvidia.com/cuda/pdf/CUDA Compiler Driver NVCC.pdf](https://docs.nvidia.com/cuda/pdf/CUDA%20Compiler%20Driver%20NVCC.pdf)
- Compilation Phases
- GPU Compilation – binary compatibility within GPU generation
- Separate Compilation (since CUDA 5.0)
- See nvcc options:
  - --compile
  - --relocatable-device-code
  - --device-c



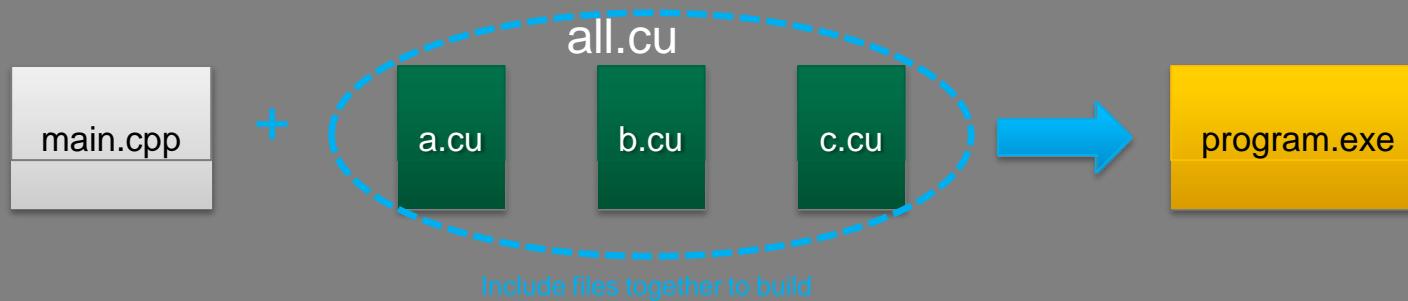
# Supported Input File Types

Input File Prefix	Description
.cu	CUDA source file, containing host code and device functions
.c	C source file
.cc, .cxx, .cpp	C++ source file
.ptx	PTX intermediate assembly file
.cubin	CUDA device code binary file (CUBIN) for a single GPU architecture
.fatbin	CUDA fat binary file that may contain multiple PTX and CUBIN files
.o, .obj	Object file
.a, .lib	Library file
.res	Resource file
.so	Shared object file

# Compilation Phases

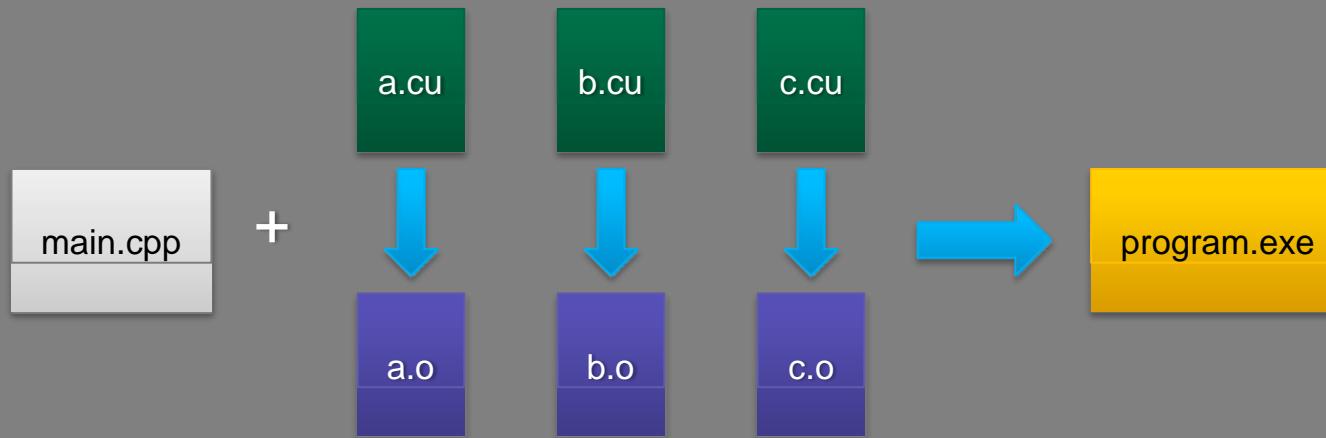
Phase	nvcc Option		Default Output File Name
	Long Name	Short Name	
CUDA compilation to C/C++ source file	<a href="#">--cuda</a>	<a href="#">-cuda</a>	.cpp.ii appended to source file name, as in <code>x.cu.cpp.ii</code> . This output file can be compiled by the host compiler that was used by <code>nvcc</code> to preprocess the <code>.cu</code> file.
C/C++ preprocessing	<a href="#">--preprocess</a>	<a href="#">-E</a>	<result on standard output>
C/C++ compilation to object file	<a href="#">--compile</a>	<a href="#">-c</a>	Source file name with suffix replaced by <code>o</code> on Linux and Mac OS X, or <code>obj</code> on Windows
Cubin generation from CUDA source files	<a href="#">--cubin</a>	<a href="#">-cubin</a>	Source file name with suffix replaced by <code>cubin</code>
Cubin generation from PTX intermediate files.	<a href="#">--cubin</a>	<a href="#">-cubin</a>	Source file name with suffix replaced by <code>cubin</code>
PTX generation from CUDA source files	<a href="#">--ptx</a>	<a href="#">-ptx</a>	Source file name with suffix replaced by <code>ptx</code>
Fatbinary generation from source, PTX or cubin files	<a href="#">--fatbin</a>	<a href="#">-fatbin</a>	Source file name with suffix replaced by <code>fatbin</code>
Linking relocatable device code.	<a href="#">--device-link</a>	<a href="#">-dlink</a>	<code>a_dlink.obj</code> on Windows or <code>a_dlink.o</code> on other platforms
Cubin generation from linked relocatable device code.	<a href="#">--device-link --cubin</a>	<a href="#">-dlink -cubin</a>	<code>a_dlink.cubin</code>
Fatbinary generation from linked relocatable device code	<a href="#">--device-link --fatbin</a>	<a href="#">-dlink -fatbin</a>	<code>a_dlink.fatbin</code>
Linking an executable	<no phase option>		<code>a.exe</code> on Windows or <code>a.out</code> on other platforms
Constructing an object file archive, or library	<a href="#">--lib</a>	<a href="#">-lib</a>	<code>a.lib</code> on Windows or <code>a.a</code> on other platforms
make dependency generation	<a href="#">--generate-dependencies</a>	<a href="#">-M</a>	<result on standard output>
make dependency generation without headers in system paths.	<a href="#">--generate-nonsystem-dependencies</a>	<a href="#">-MM</a>	<result on standard output>
Running an executable	<a href="#">--run</a>	<a href="#">-run</a>	

# No Separate Compilation in early releases



Earlier CUDA required single source file for a single kernel  
No linking external device code

# CUDA 5: Separate Compilation & Linking



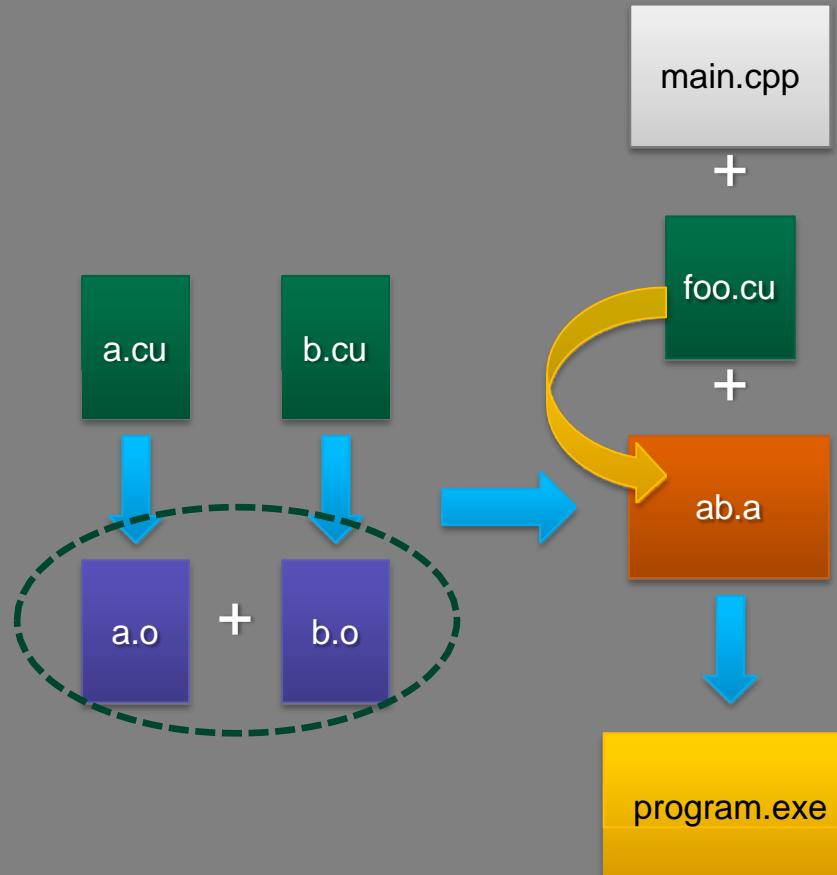
Separate compilation allows building independent object files

CUDA 5 can link multiple object files into one program

# Benefits of Separate Compilation

- Eases porting code
  - no longer have to include files together
  - “extern” attribute is respected
- Incremental compilation reduces build time
  - e.g. 47000 line app used to take 50 seconds to build, now when split into multiple files takes 4 seconds to build if only one file changed
- Can create and use 3<sup>rd</sup> party libraries

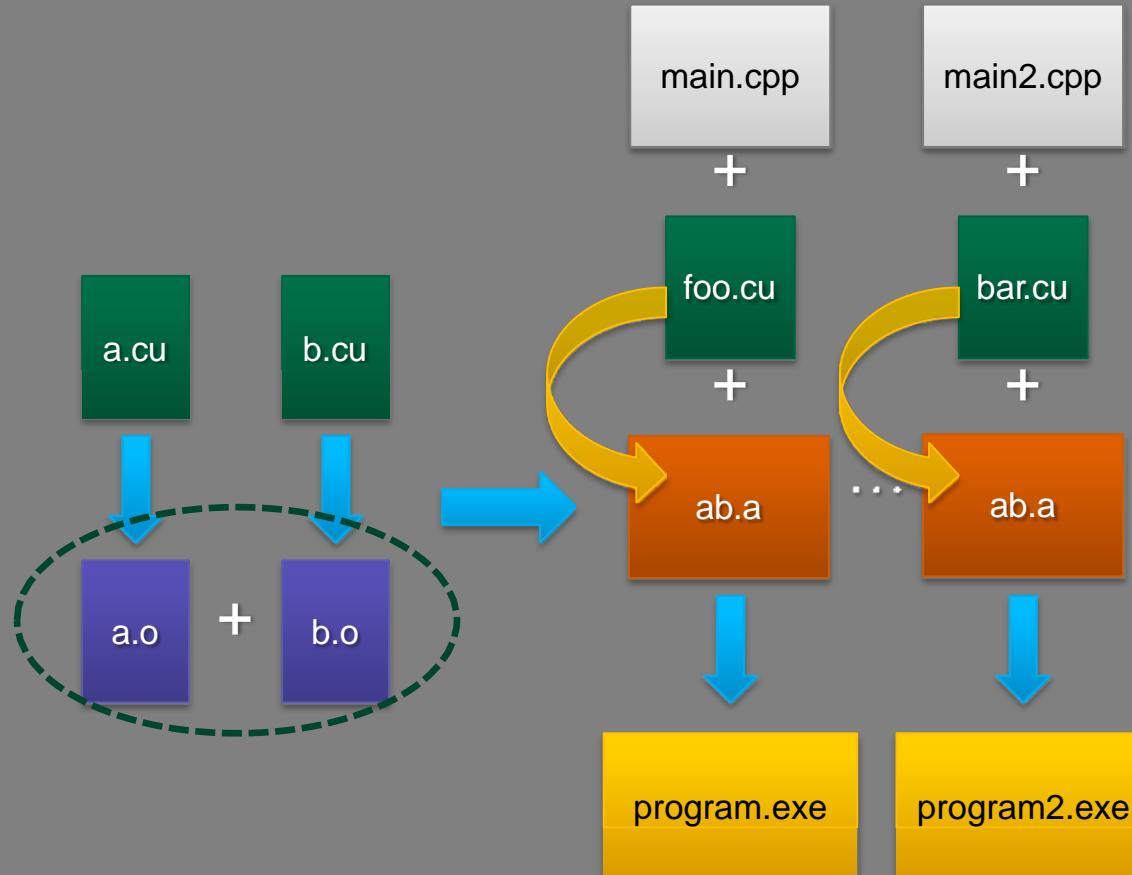
# CUDA 5: Library Support



Can combine object files into static libraries

Link and externally call *device* code

# CUDA 5: Library Support



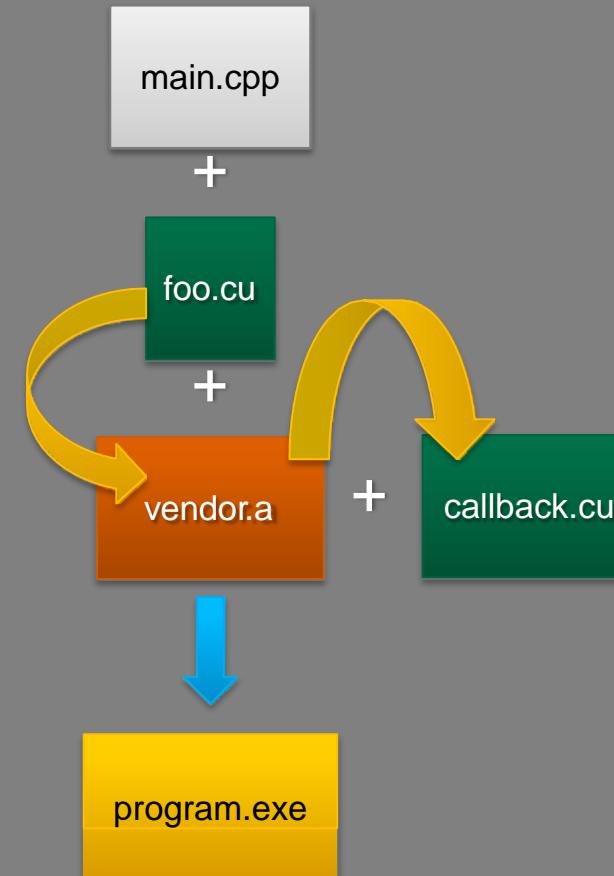
Can combine object files into static libraries

Link and externally call *device* code

Facilitates code reuse, reduces compile time

# CUDA 5: Callbacks

Enables closed-source device libraries to call user-defined device callback functions



# Summary

- Separate Compilation of device code is supported in CUDA 5.0
- Eases porting
- Incremental Recompilation
- Library Support
- For more info, see “Using Separate Compilation in CUDA” section at end of NVCC document.



## CUDA Compiler Driver NVCC

Reference Guide

TRM-06721-001\_v11.4 | August 2021

<https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html>  
(PDF copy in docs folder)

# Stencil 1D – Shared Memory

(samples\04-stencil-1d)

```
__global__ void stencil(const int N, int* in, int* out)
{
    __shared__ int temp[BLOCKSIZE + 2 * RADIUS];
    int gIndex = threadIdx.x + blockIdx.x * BLOCKSIZE;
    int lIndex = threadIdx.x + RADIUS;

    if (gIndex < N)
    {
        // read input data into shared memory
        temp[lIndex] = in[gIndex + RADIUS];
        if (threadIdx.x < RADIUS)
        {
            temp[lIndex - RADIUS] = in[gIndex];
            temp[lIndex + BLOCKSIZE] = in[gIndex + RADIUS + BLOCKSIZE];
        }
    }
}
```

# Stencil 1D – Shared Memory

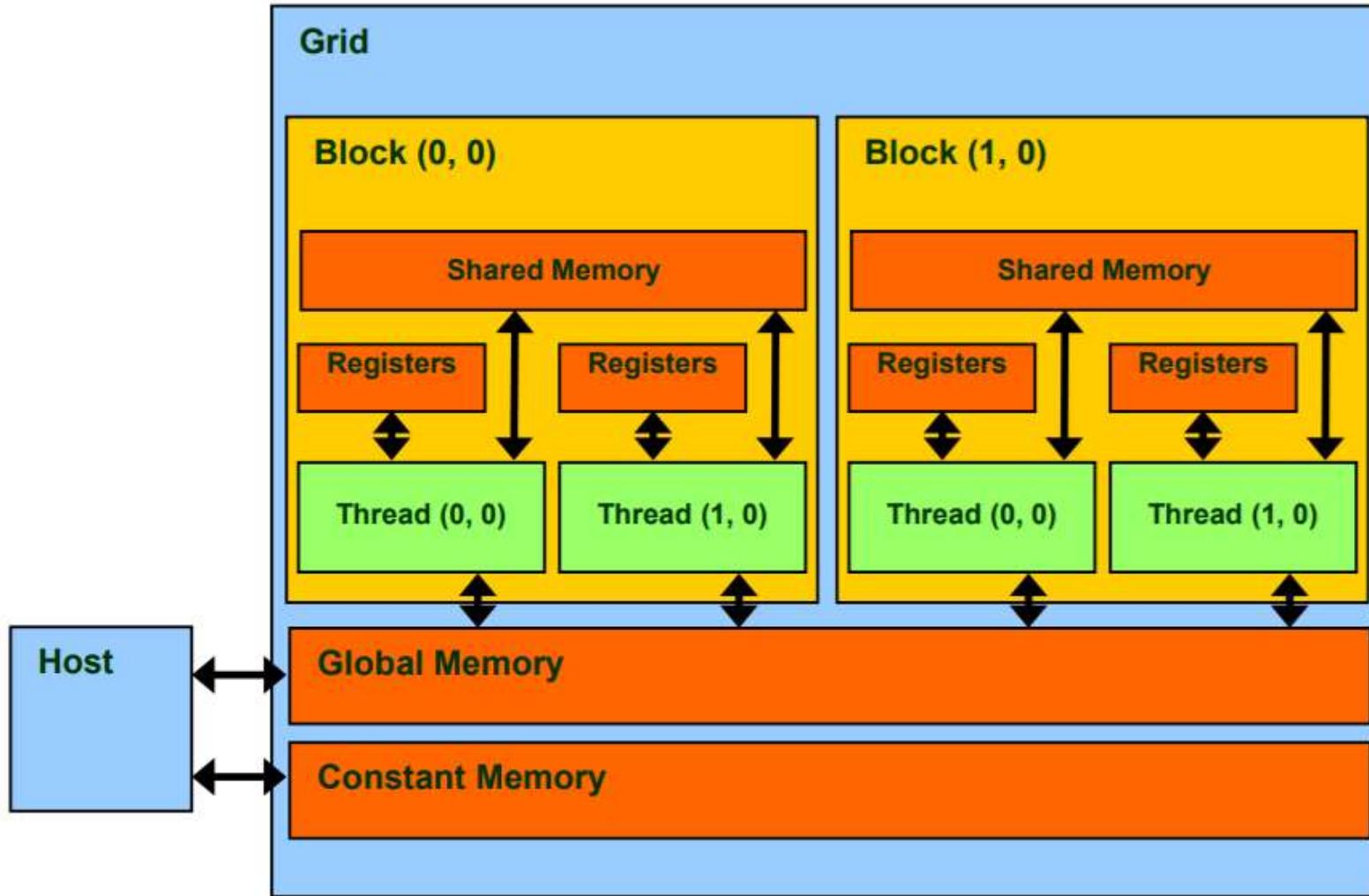
```
// ensure all reads are complete
__syncthreads();

if (gIndex < N - 2 * RADIUS)
{
    // apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; ++offset)
    {
        result += temp[lIndex + offset];
    }

    // output the filtered result
    out[gIndex + RADIUS] = result;
}
}
```

Exercise: Extend to 2D Filter  
(To be revisited in Image Processing session)

# CUDA Memory Model



# Declaring CUDA Variables

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

- `__device__` is optional when used with `__shared__`, or `__constant__`
- **Automatic variables** reside in a `register`
  - Except per-thread arrays that reside in global memory

# Optimize Algorithms for the GPU



- **Maximize independent parallelism**
- **Maximize arithmetic intensity (math/bandwidth)**
- **Sometimes it's better to recompute than to cache**
  - GPU spends its transistors on ALUs, not memory
- **Do more computation on the GPU to avoid costly data transfers**
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host



# Optimize Memory Access

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory
- Optimize for spatial locality in cached texture memory
- In shared memory, avoid high-degree bank conflicts



# Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory
- Use one / a few threads to load / compute data shared by all threads
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing

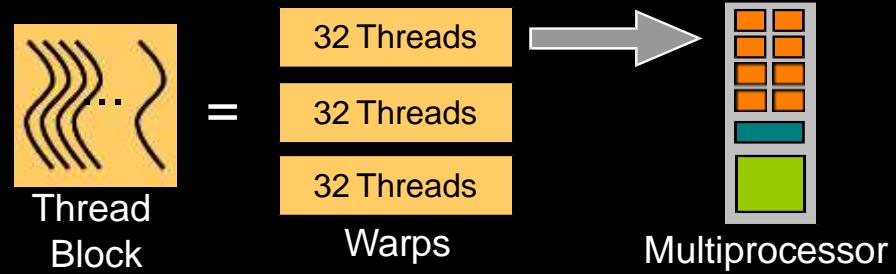


# Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

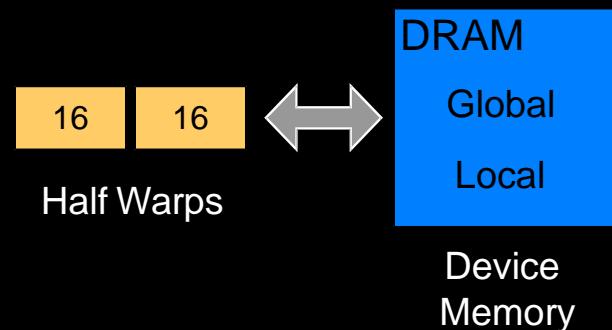


# Warps and Half Warps



A thread block consists of 32-thread warps

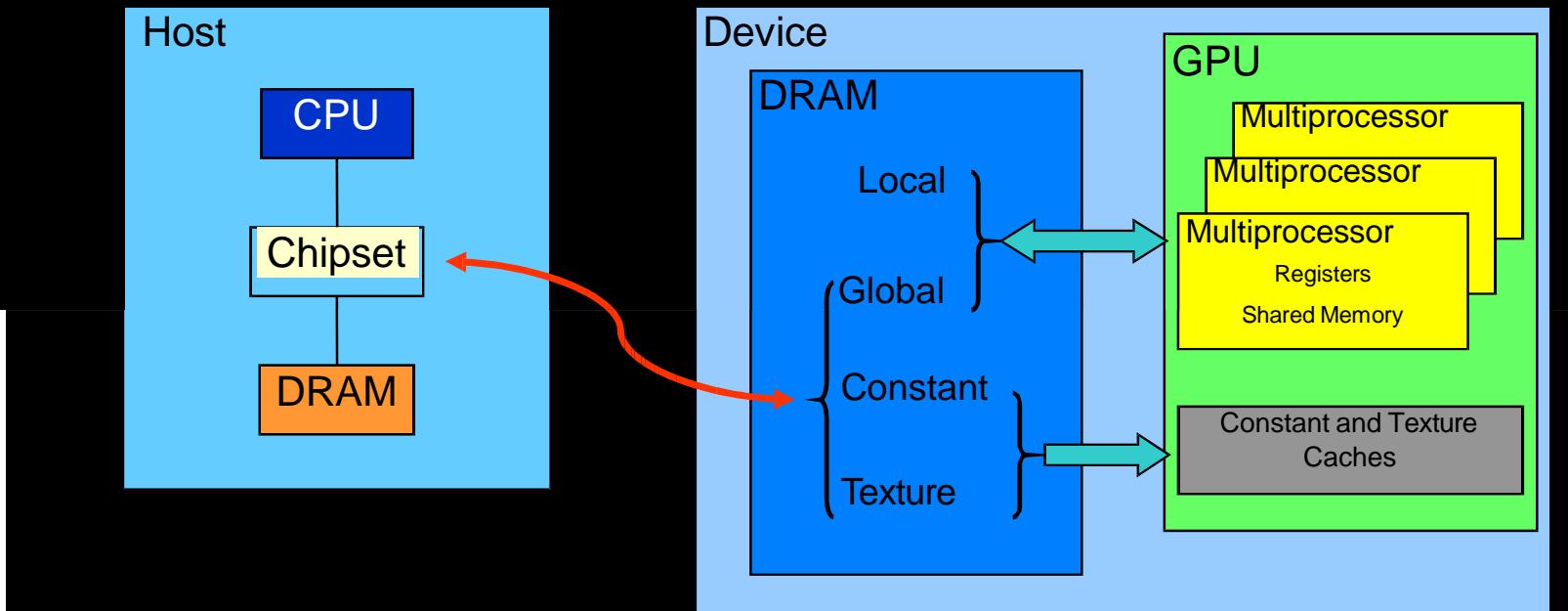
A warp is executed physically in parallel (SIMD) on a multiprocessor



A half-warp of 16 threads can coordinate global memory accesses into a single transaction

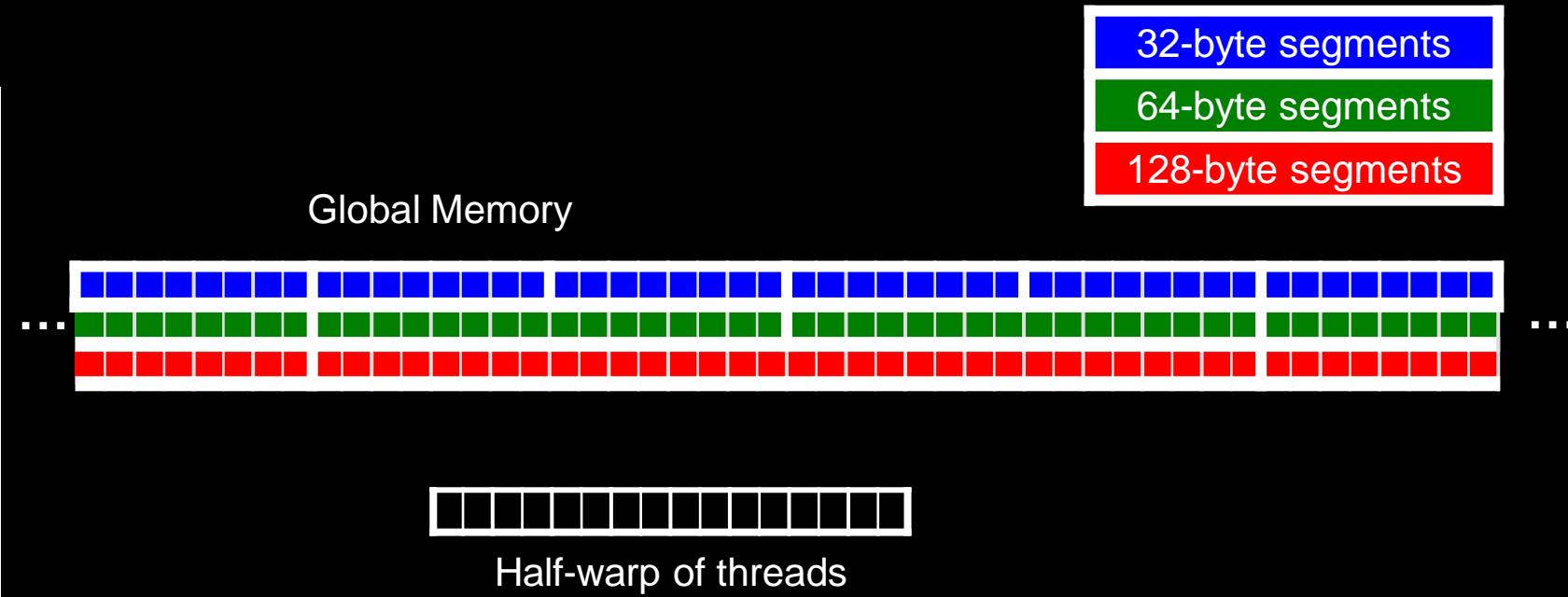


# Memory Architecture



# Coalescing

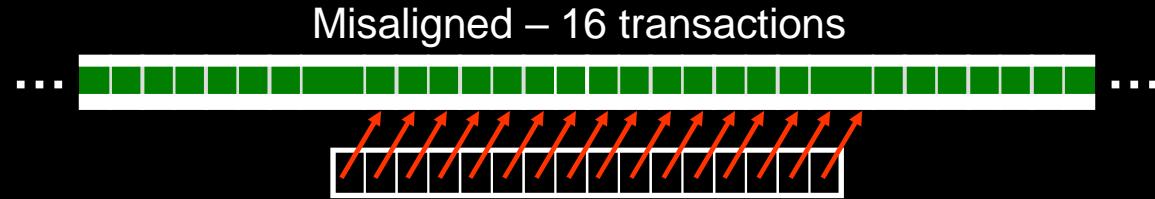
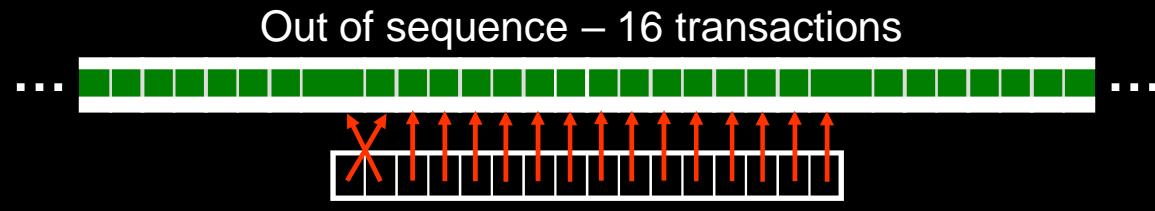
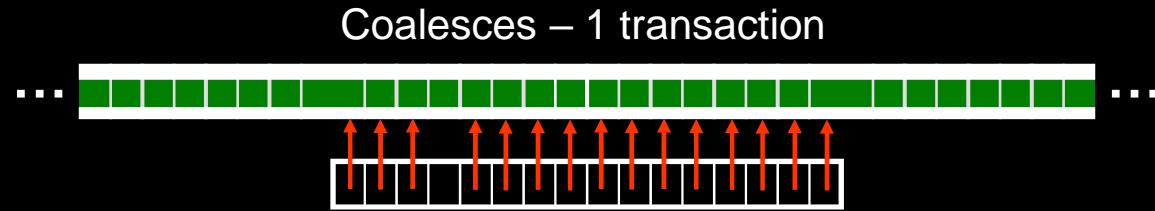
- Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met
- Depends on compute capability
  - Older versions have stricter access requirements
- **Float (32-bit) data example:**



# Coalescing

## Compute capability 1.0 and 1.1

- K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate

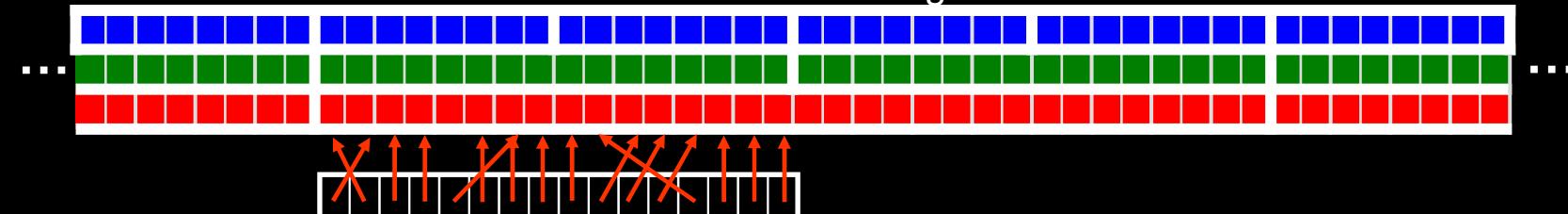


# Coalescing

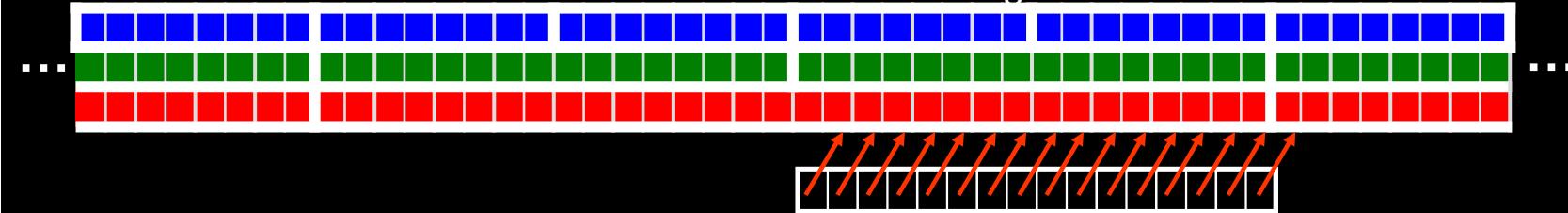
## Compute capability 1.2 and higher

- Issues transactions for segments of 32B, 64B, and 128B
- Smaller transactions used to avoid wasted bandwidth

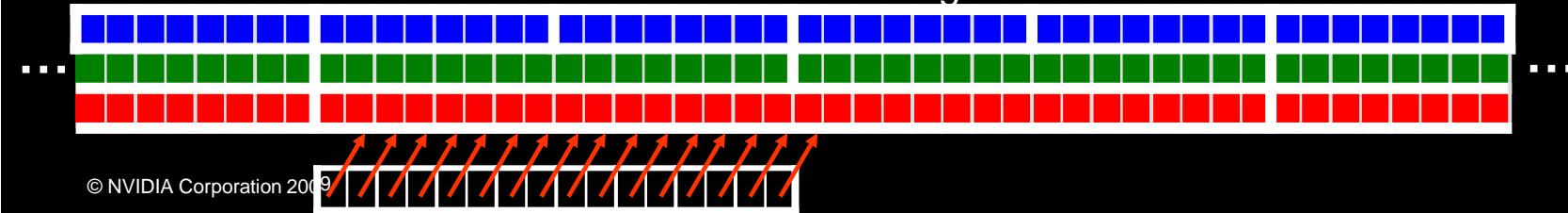
1 transaction - 64B segment



2 transactions - 64B and 32B segments



1 transaction - 128B segment





# Shared Memory

- ~Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing

# Shared Memory Architecture

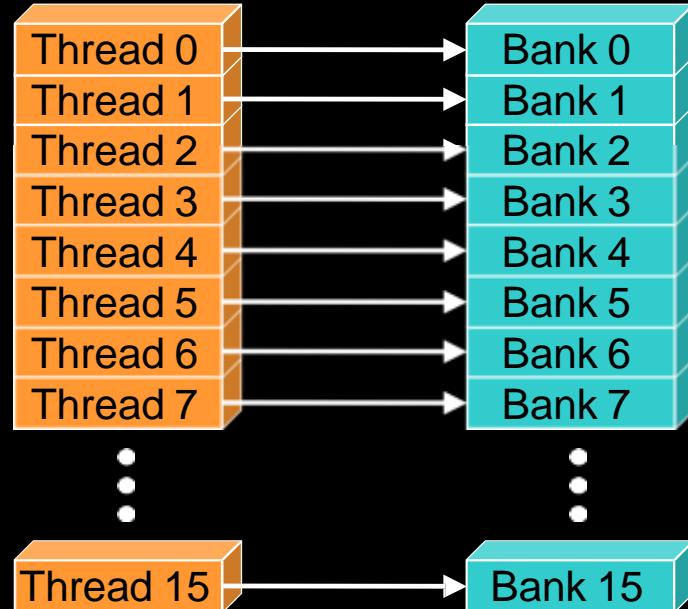
- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Successive 32-bit words assigned to successive banks
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



# Bank Addressing Examples

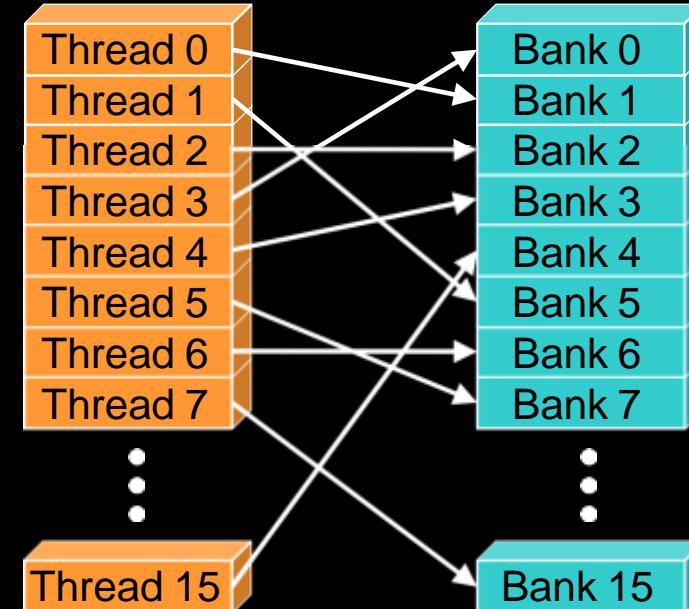
- **No Bank Conflicts**

- Linear addressing  
stride == 1



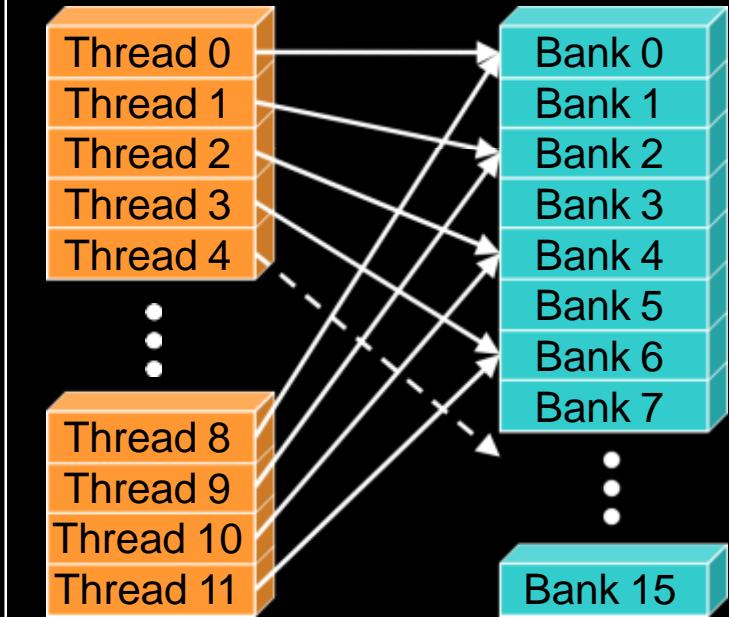
- **No Bank Conflicts**

- Random 1:1 Permutation

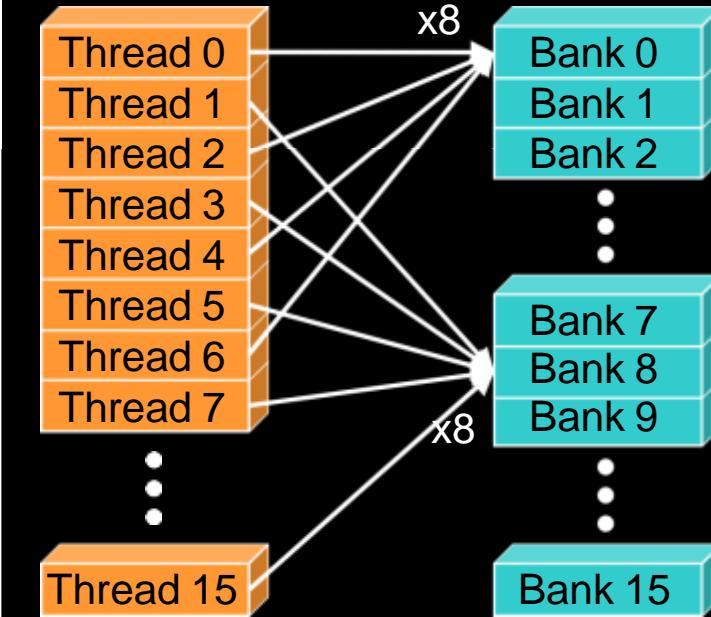


# Bank Addressing Examples

- **2-way Bank Conflicts**
  - Linear addressing stride == 2



- **8-way Bank Conflicts**
  - Linear addressing stride == 8





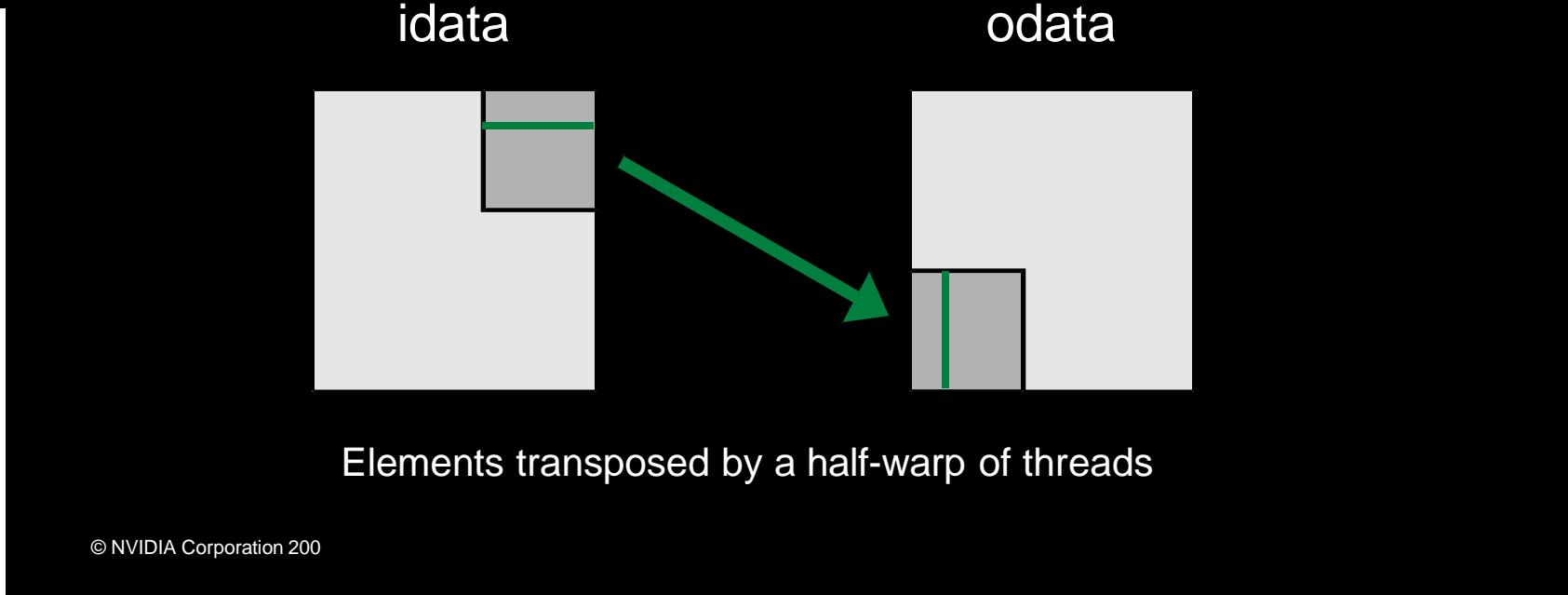
# Shared memory bank conflicts

- Shared memory is ~ as fast as registers if there are no bank conflicts
- warp\_serialize profiler signal reflects conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank



## Shared Memory Example: Transpose

- Each thread block works on a tile of the matrix
- Naïve implementation exhibits strided access to global memory



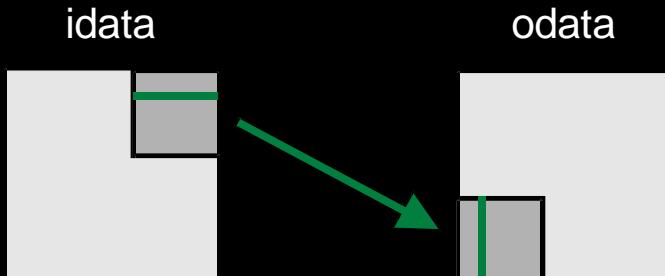
# Naïve Transpose

- Loads are coalesced, stores are not (strided by height)

```
__global__ void transposeNaive(float *odata, float *idata,
                               int width, int height)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index_in  = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;

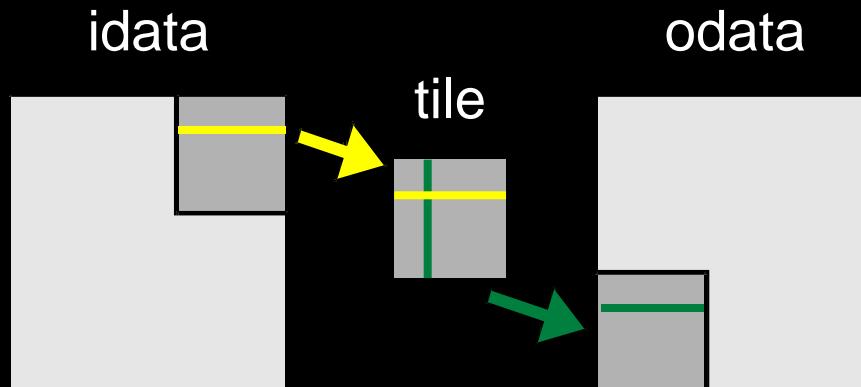
    odata[index_out] = idata[index_in];
}
```



# Coalescing through shared memory



- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires `__syncthreads()` since threads access data in shared memory stored by other threads



Elements transposed by a half-warp of threads

# Coalescing through shared memory



```
__global__ void transposeCoalesced(float *odata, float *idata,
                                   int width, int height)
{
    __shared__ float tile[TILE_DIM][TILE_DIM];

    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;

    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;

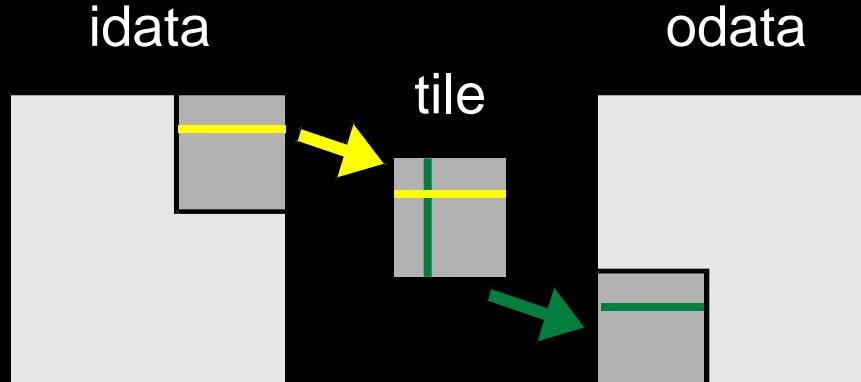
    tile[threadIdx.y][threadIdx.x] = idata[index_in];

    __syncthreads();

    odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

# Bank Conflicts in Transpose

- 16x16 shared memory tile of floats
  - Data in columns are in the same bank
  - 16-way bank conflict reading columns in tile
- Solution - pad shared memory array
  - `__shared__ float tile[TILE_DIM] [TILE_DIM+1];`
  - Data in anti-diagonals are in same bank



Elements transposed by a half-warp of threads



# Memory Padding

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4
0	1	2	3	4
0	1	2	3	4
0	1	2	3	4
0	1	2	3	4
0	1	2	3	4

Bank Conflict

Bank 0	Bank 1	Bank 2	Bank 3	Bank 4
0	1	2	3	4
4	0	1	2	3
3	4	0	1	2
2	3	4	0	1
1	2	3	4	0

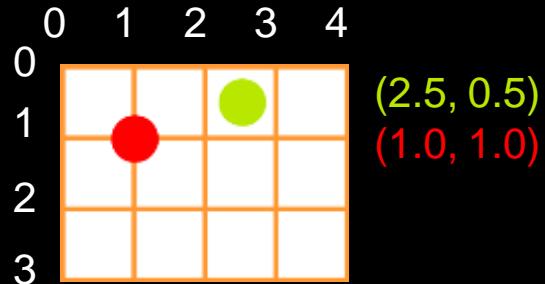
No Bank Conflict



# Textures in CUDA

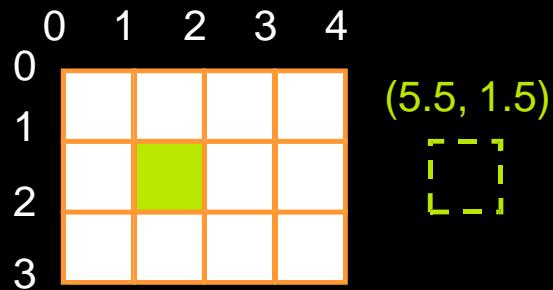
- **Texture is an object for reading data**
- **Benefits:**
  - Data is cached
    - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear interpolation
    - Dedicated hardware
  - Wrap modes (for “out-of-bounds” addresses)
    - Clamp to edge / repeat
  - Addressable in 1D, 2D, or 3D
    - Using integer or normalized coordinates

# Texture Addressing



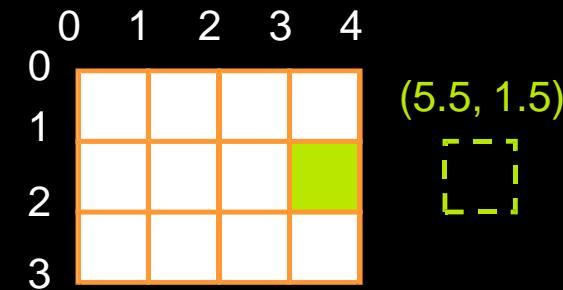
## Wrap

- Out-of-bounds coordinate is wrapped (modulo arithmetic)



## Clamp

- Out-of-bounds coordinate is replaced with the closest boundary





# CUDA Texture Types

- **Bound to linear memory**
  - Global memory address is bound to a texture
  - Only 1D
  - Integer addressing
  - No filtering, no addressing modes
- **Bound to CUDA arrays**
  - Block linear CUDA array is bound to a texture
  - 1D, 2D, or 3D
  - Float addressing (size-based or normalized)
  - Filtering
  - Addressing modes (clamping, repeat)
- **Bound to pitch linear (CUDA 2.2)**
  - Global memory address is bound to a texture
  - 2D
  - Float/integer addressing, filtering, and clamp/repeat addressing modes similar to CUDA arrays



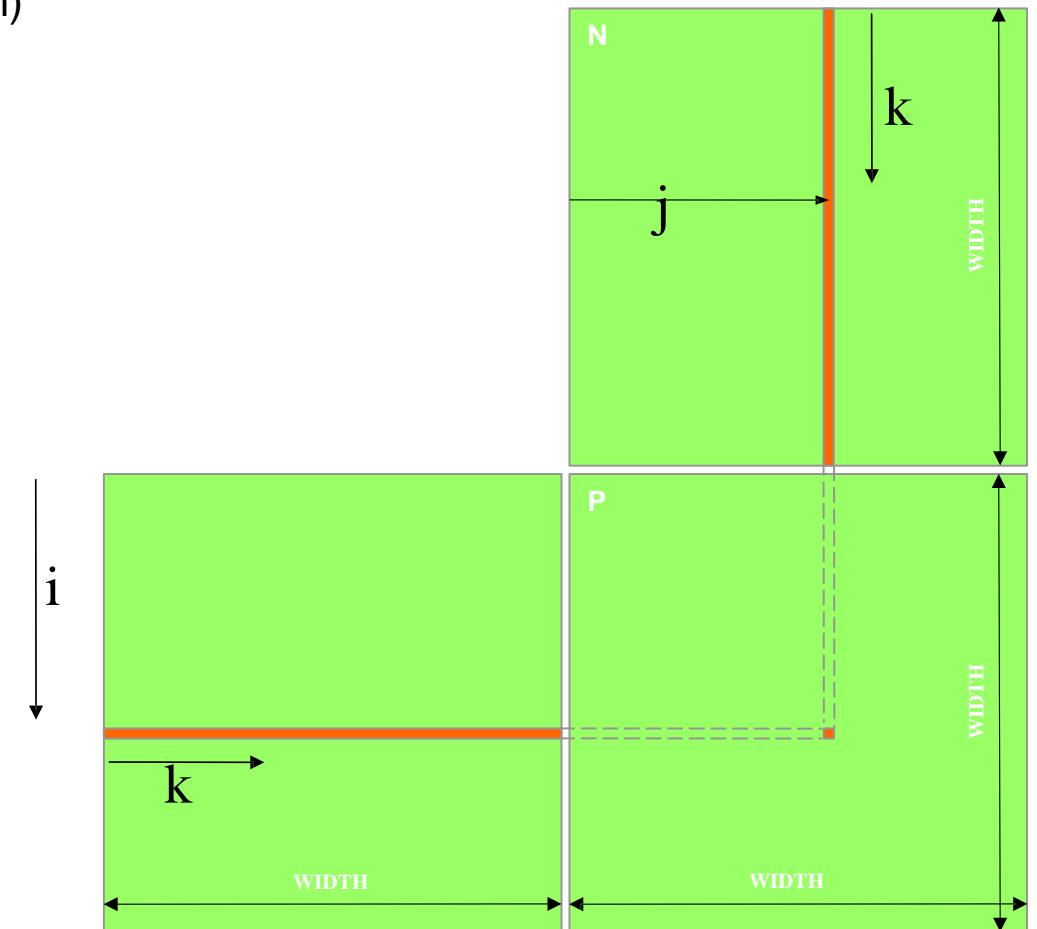
# Summary

- GPU hardware can achieve great performance on data-parallel computations if you follow a few simple guidelines:
  - Use parallelism efficiently
  - Coalesce memory accesses if possible
  - Take advantage of shared memory
  - Explore other memory spaces
    - Texture
    - Constant
  - Reduce bank conflicts

# Matrix Multiplication: Simple Host Version

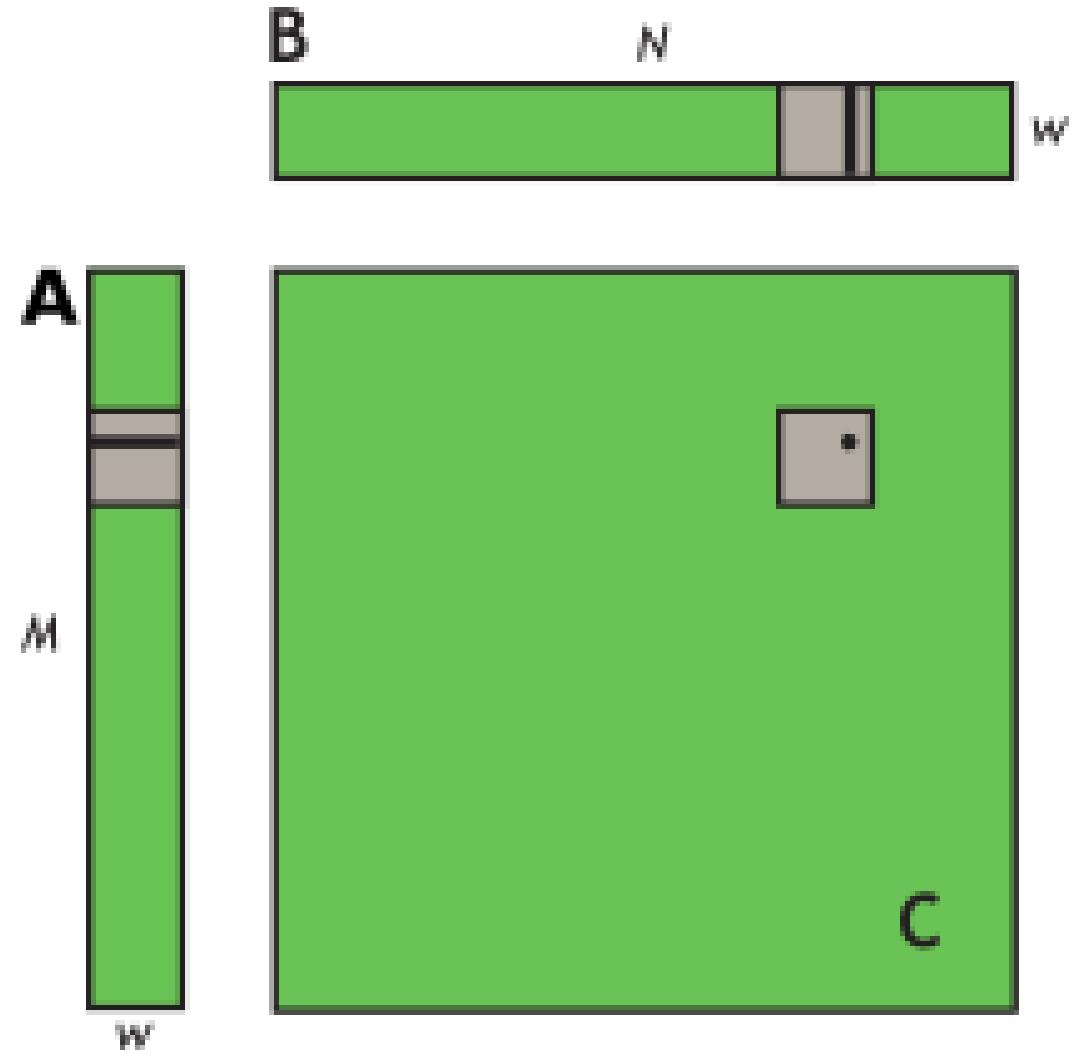
```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
```

```
{  
    for (int i = 0; i < Width; ++i)  
        for (int j = 0; j < Width; ++j) {  
            double sum = 0;  
            for (int k = 0; k < Width; ++k) {  
                double a = M[i * width + k];  
                double b = N[k * width + j];  
                sum += a * b;  
            }  
            P[i * Width + j] = sum;  
        }  
}
```



# Simple Kernel

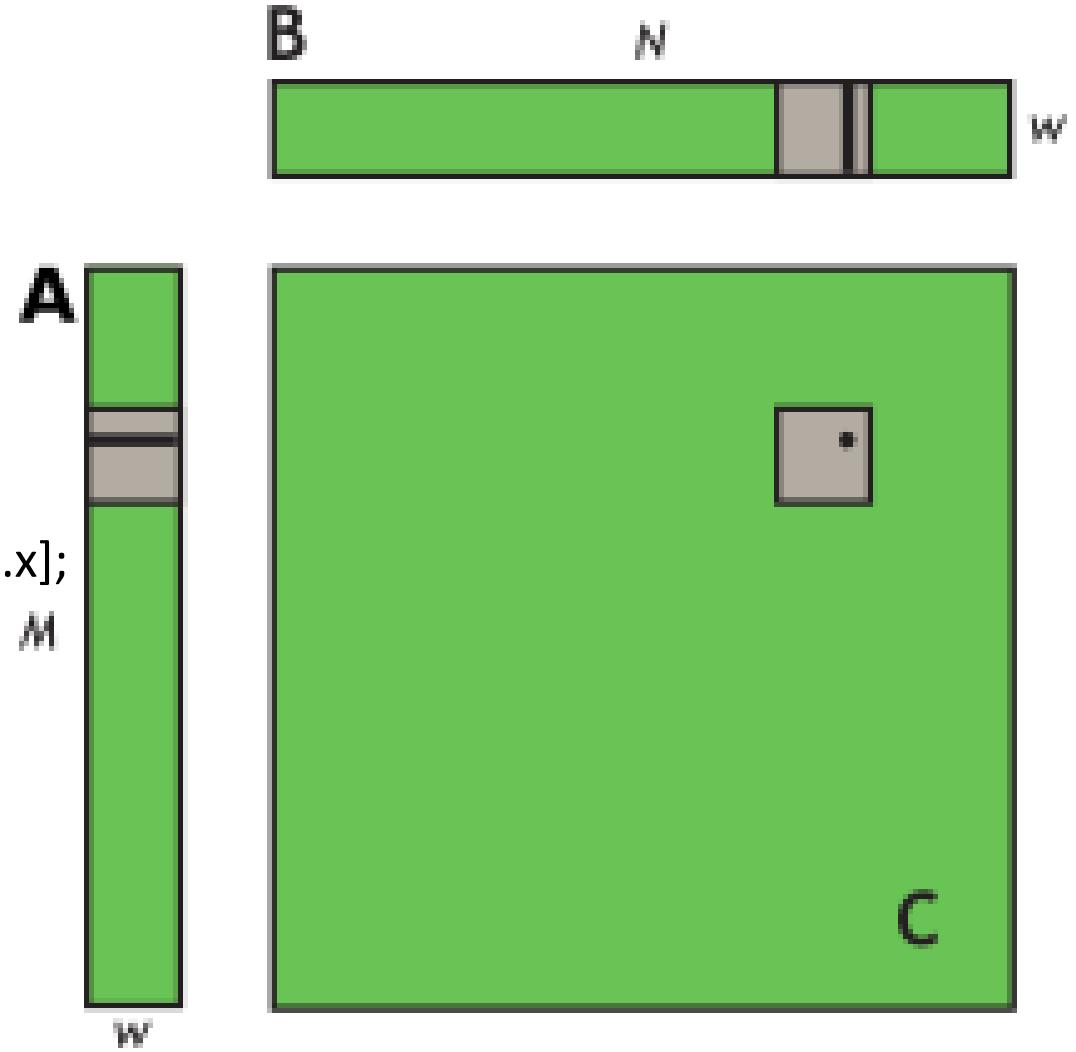
```
__global__ void simpleMultiply(float *a, float* b, float *c,  
                               int N)  
{  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    float sum = 0.0f;  
    for (int i = 0; i < w; i++) {  
        sum += a[row*w+i] * b[i*N+col];  
    }  
    c[row*N+col] = sum;  
}
```

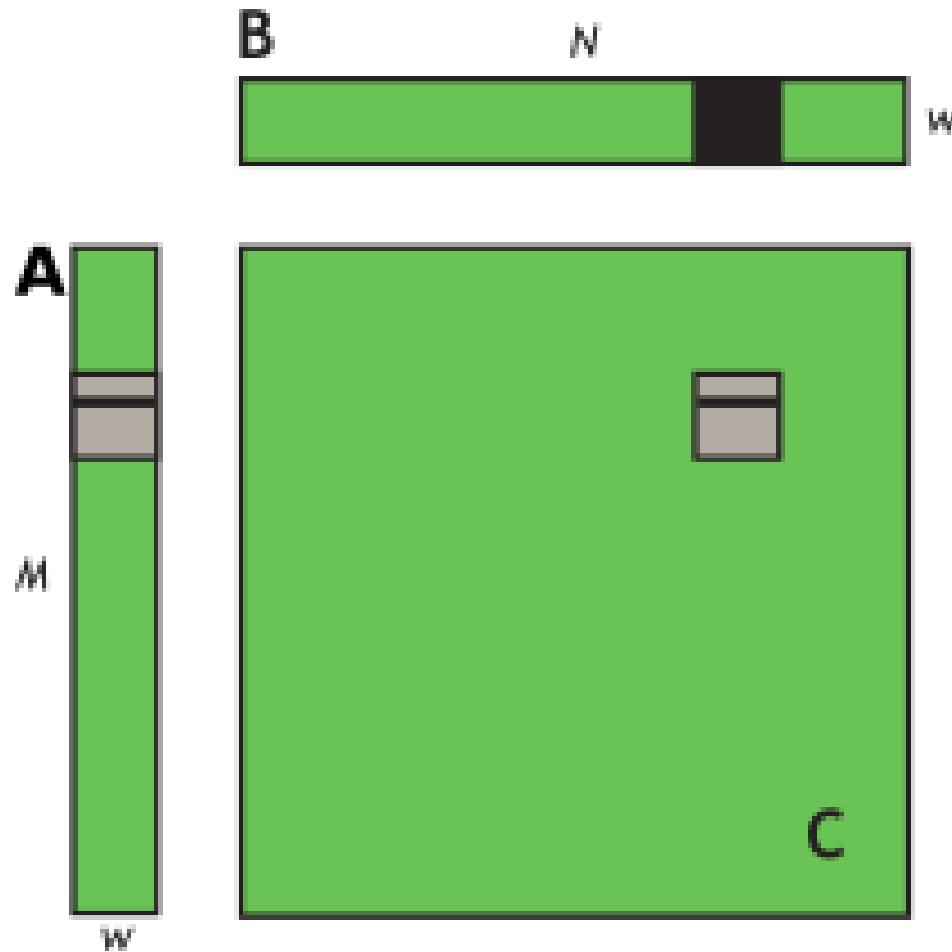


# Kernel with Shared Memory

(CUDA Samples\matrixMul)

```
__global__ void coalescedMultiply(float *a, float* b, float *c,  
                                  int N)  
{  
    __shared__ float aTile[TILE_DIM][TILE_DIM];  
  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    float sum = 0.0f;  
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];  
    __syncwarp();  
    for (int i = 0; i < TILE_DIM; i++) {  
        sum += aTile[threadIdx.y][i]* b[i*N+col];  
    }  
    c[row*N+col] = sum;  
}
```

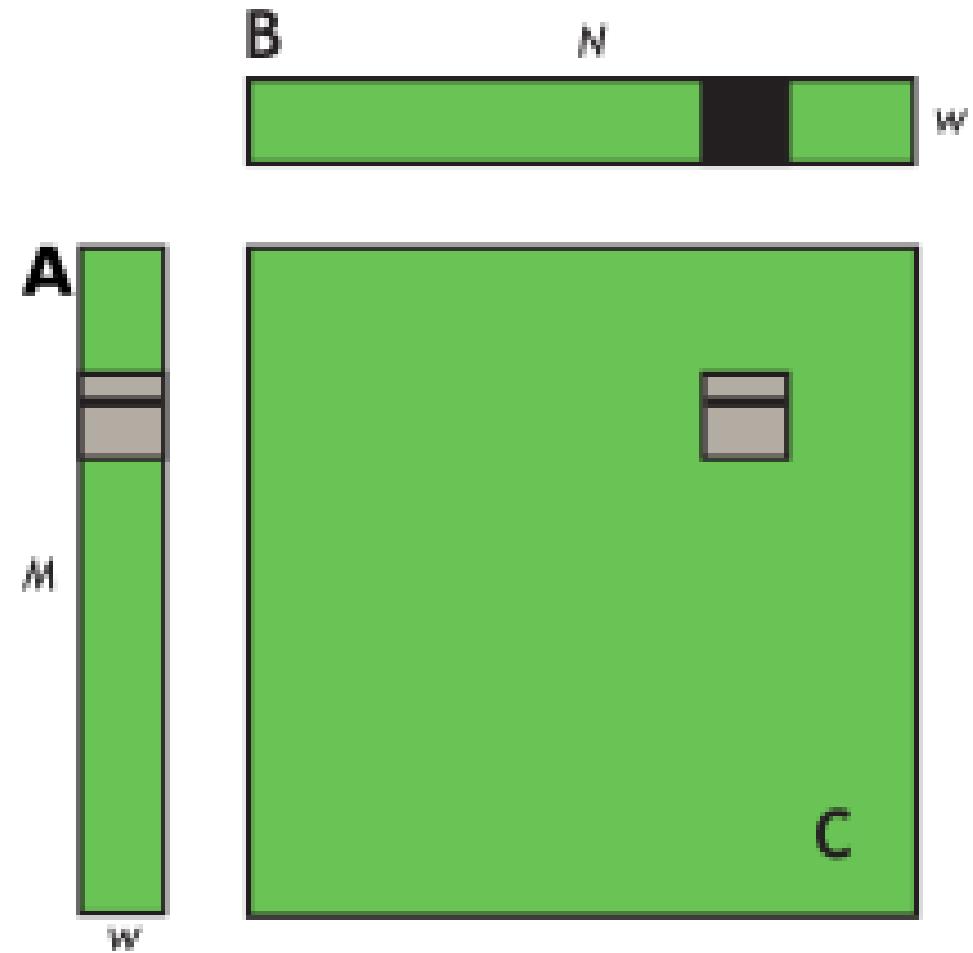




Computing a row of a tile in  $C$  using one row of  $A$  and an entire tile of  $B$

# Eliminate repeated reading of entire tile of B

```
__global__ void sharedABMultiply(float *a, float* b, float *c,
                                int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM],
                 bTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x]; M
    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col]; W
    __syncthreads();
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
    }
    c[row*N+col] = sum;
}
```



# Communication via Shared Mem.

- Little question:

```
__global__race_condition()
{
    __shared__int shared_var = threadIdx.x;
    // What is the value of shared_var here???
}
```

# Communication via Shared Mem.

- Answer:
  - Value of shared\_var is undefined
  - This is a race condition
    - Multiple threads writing to one variable w/o explicit synchronization
    - Variable will have arbitrary (i.e. undefined) value
  - Need for synchronization/barriers
    - \_\_syncthreads()
    - Atomic operations

# Communication via Shared Mem.

- `__syncthreads()`
  - Point of synchronization for all threads in a block
  - Not always necessary
    - Half-warps are lock-stepped
- Common usage: make sure data is ready

```
__global__ void kernel(float * d_src)
{
    __shared__ float a_sh[BLOCK_SIZE];
    a_sh[threadIdx.x] = d_src[threadIdx.x];
    __syncthreads();
    // a_sh is now correctly filled by all
    // threads in the block
}
```

# Communication via Shared Mem.

- Atomic operations
  - atomicAdd(), atomicSub(), atomicExch(), atomicMax(), ...
- Example

```
__global__ void sum(float * src, float * dst)
{
    atomicAdd(dst, src[threadIdx.x]);
}
```

# Communication via Shared Mem.

- But: atomic operations are not cheap
- Serialized write access to a memory cell
- Better solution:
  - Partial sums within thread block
    - atomicAdd() on a `__shared__` variable
  - Global sum
    - atomicAdd() on global memory

# Communication via Shared Mem.

- Better version of sum()

```
__global__ void sum(float * src, float * dst)
{
    int pos = blockDim.x*blockIdx.x + threadIdx.x;

    __shared__ float partial_sum;
    if (threadIdx.x == 0) partial_sum = 0.0f;
    __syncthreads();

    atomicAdd(&partial_sum, src[pos]);

    if (threadIdx.x == 0) atomicAdd(dst, partial_sum)
}
```

# Communication via Shared Mem.

- General guidelines:
  - Do not synchronize or serialize if not necessary
  - Use `__syncthreads()` to wait until `__shared__` data is filled
  - Data access pattern is regular or predictable  
→ `__syncthreads()`
  - Data access pattern is sparse or not predictable  
→ atomic operations
  - Atomic operations are much faster for shared variables than for global ones

# Atomic Memory Operations

(CUDA Samples / simpleAtomicIntrinsics)

# What Is an Atomic Memory Operation?

- Uninterruptable read-modify-write memory operation
  - Requested by threads
  - Updates a value at a specific address
- Serializes contentious updates from multiple threads
- Enables co-ordination among >1 threads
- Limited to specific functions & data sizes

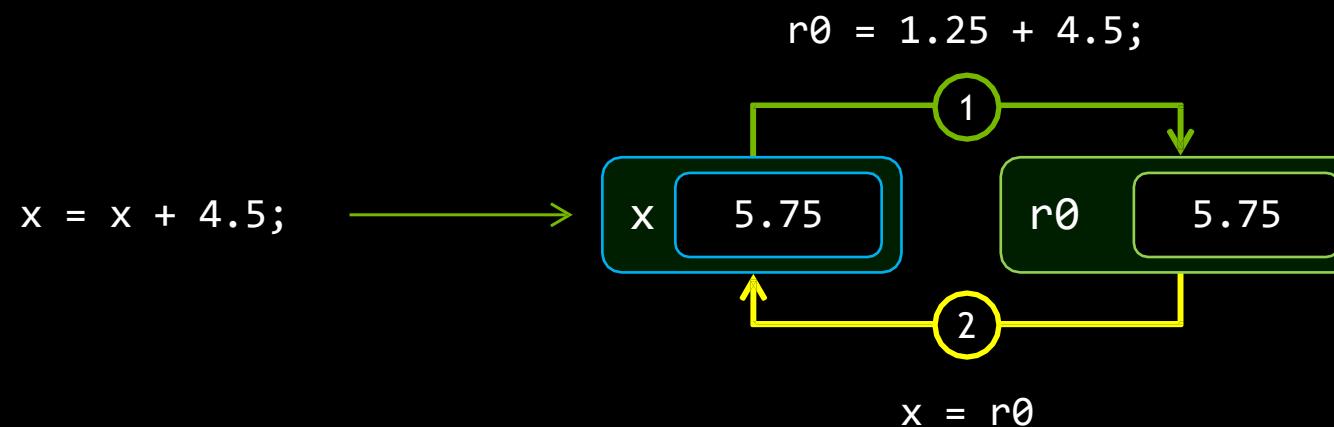
# Precise Meaning of atomicAdd()

```
int atomicAdd(int *p, int v)
{
    int old;
exclusive_single_thread
    {
        // atomically perform LD; ADD; ST ops
        old = *p; // Load from memory
        *p = old + v; // Store after adding v
    }
    return old;
}
```

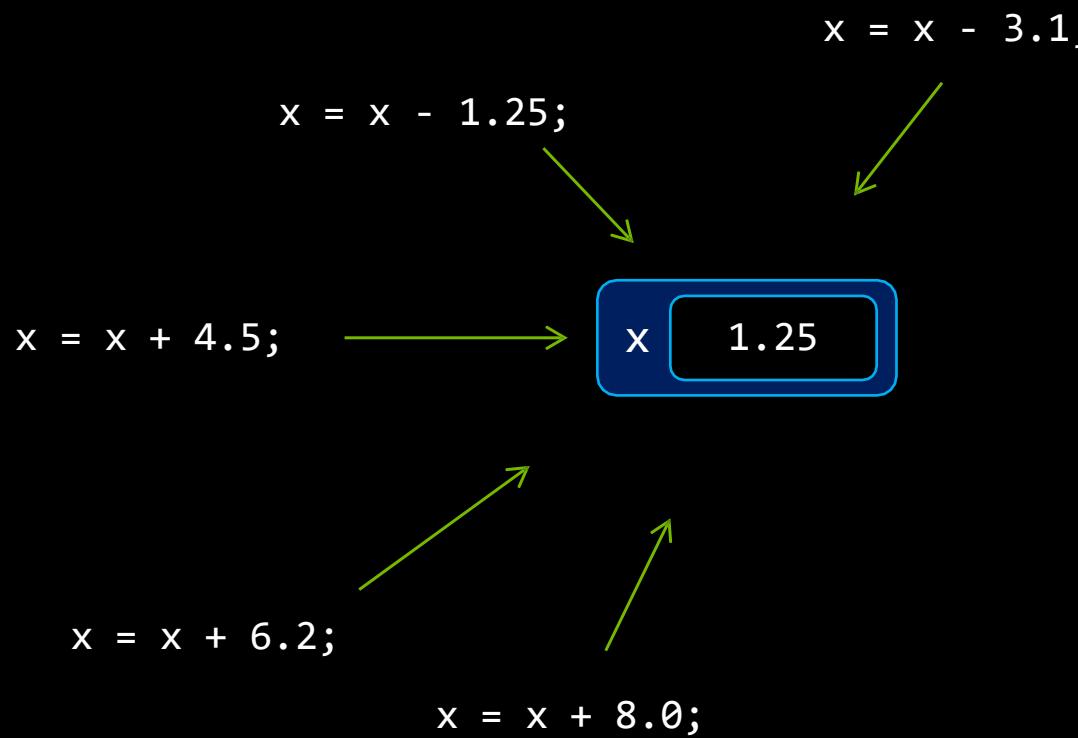
# Simple Atomic Example

$x = x + 4.5;$        x 1.25

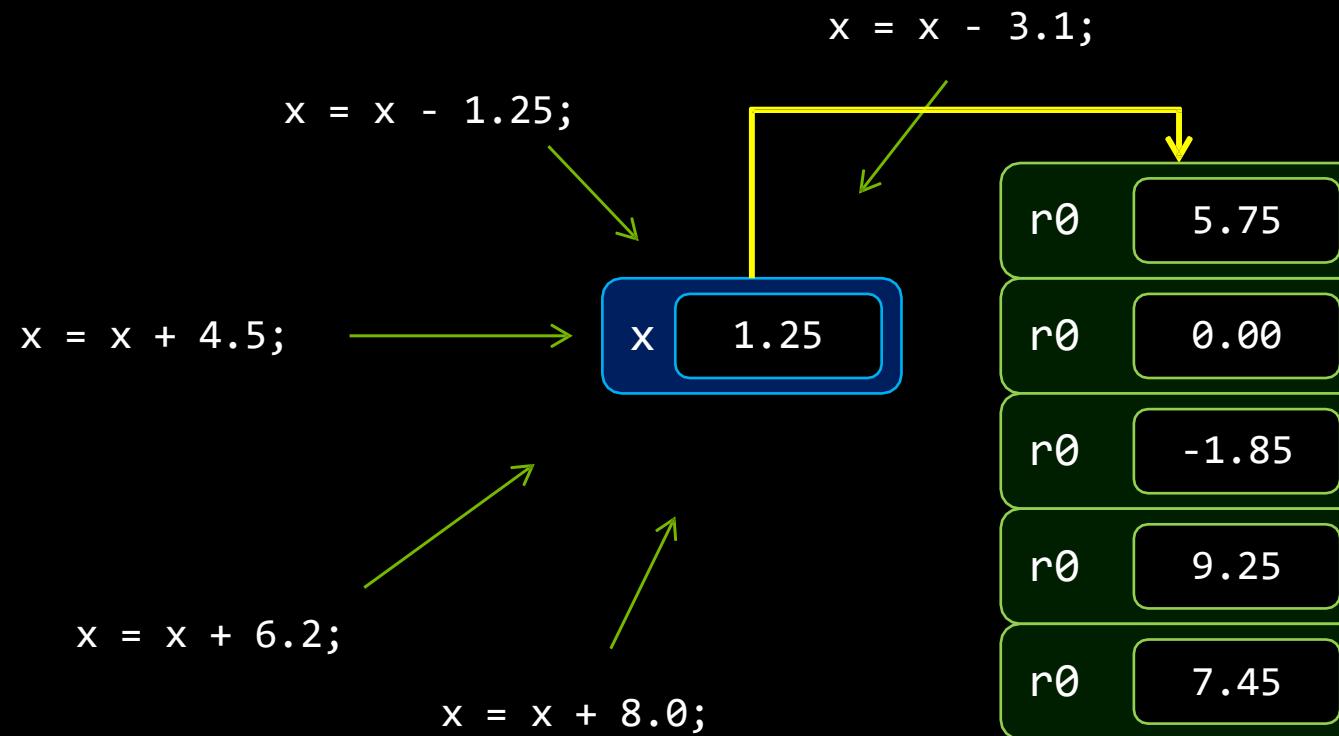
# Simple Atomic Example



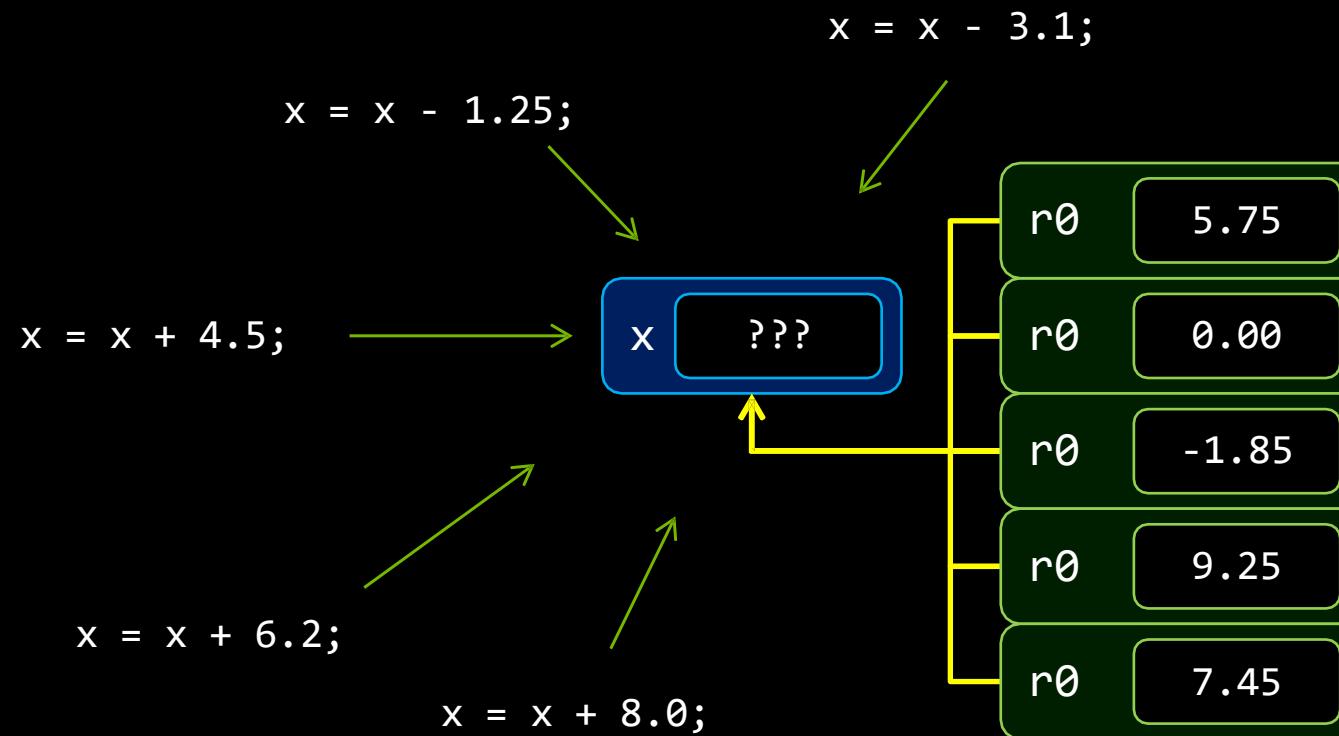
# Simple Atomic Example



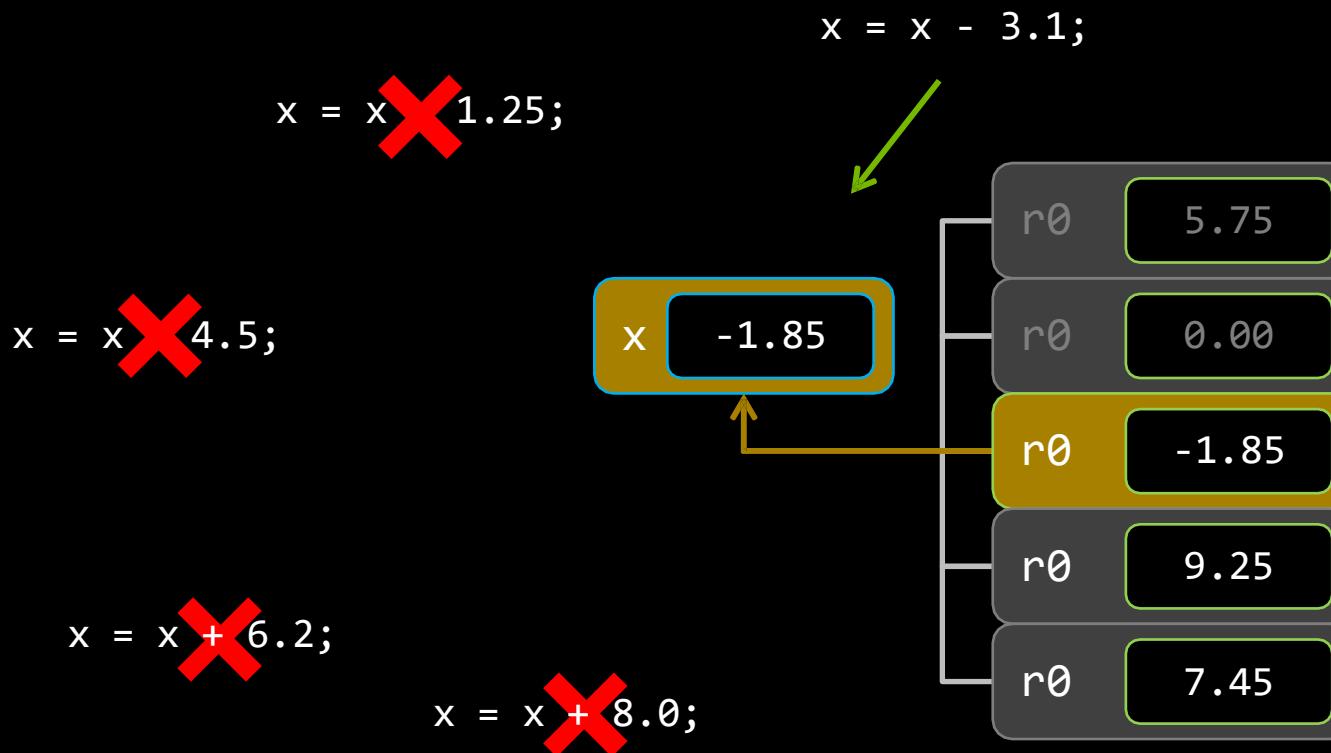
# Simple Atomic Example



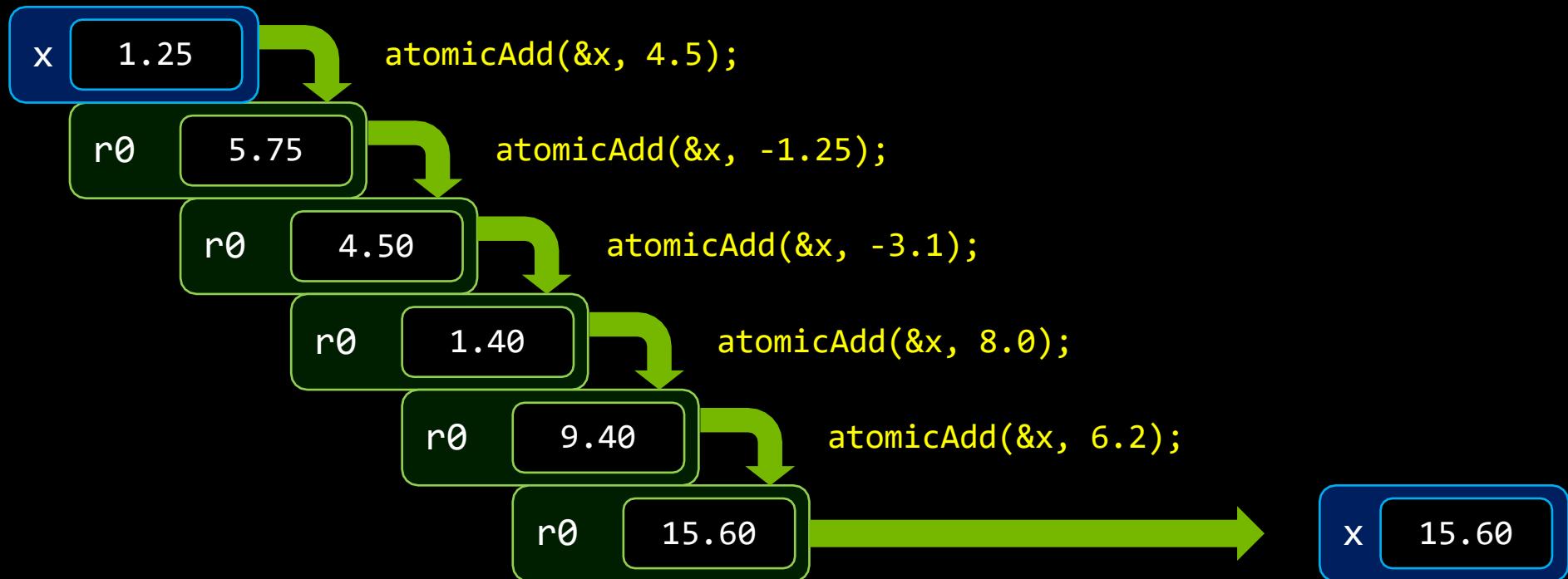
# Simple Atomic Example



# Simple Atomic Example



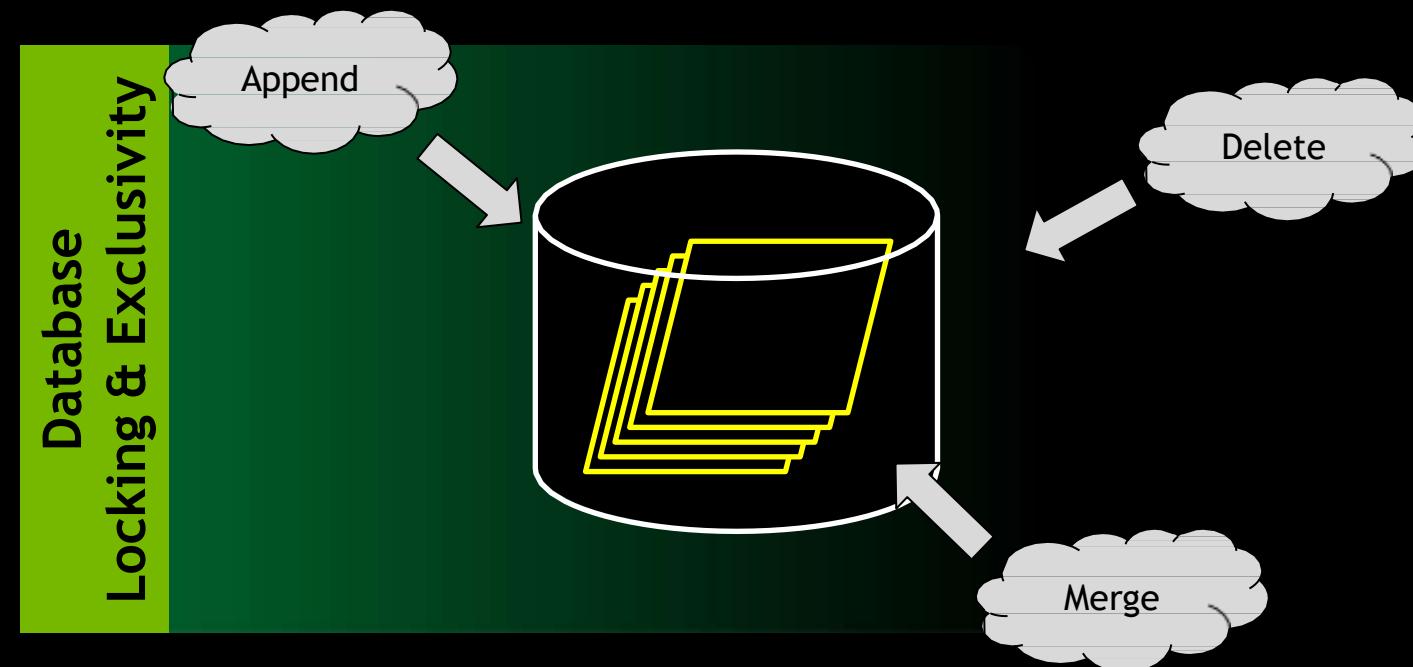
# Simple Atomic Example



# Why Use Atomics?

Common problem: races on read-modify-write of shared data

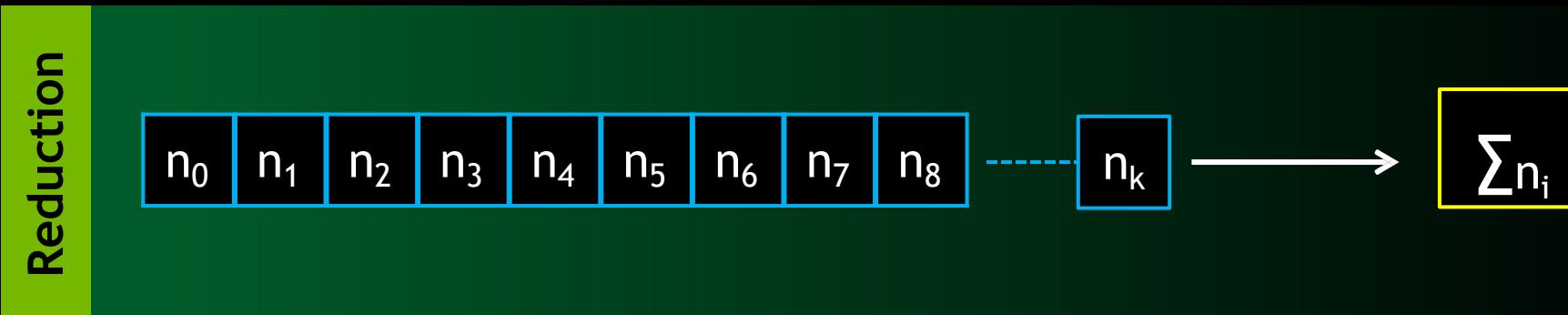
- Transactions & Data Access Control



# Why Use Atomics?

Common problem: races on read-modify-write of shared data

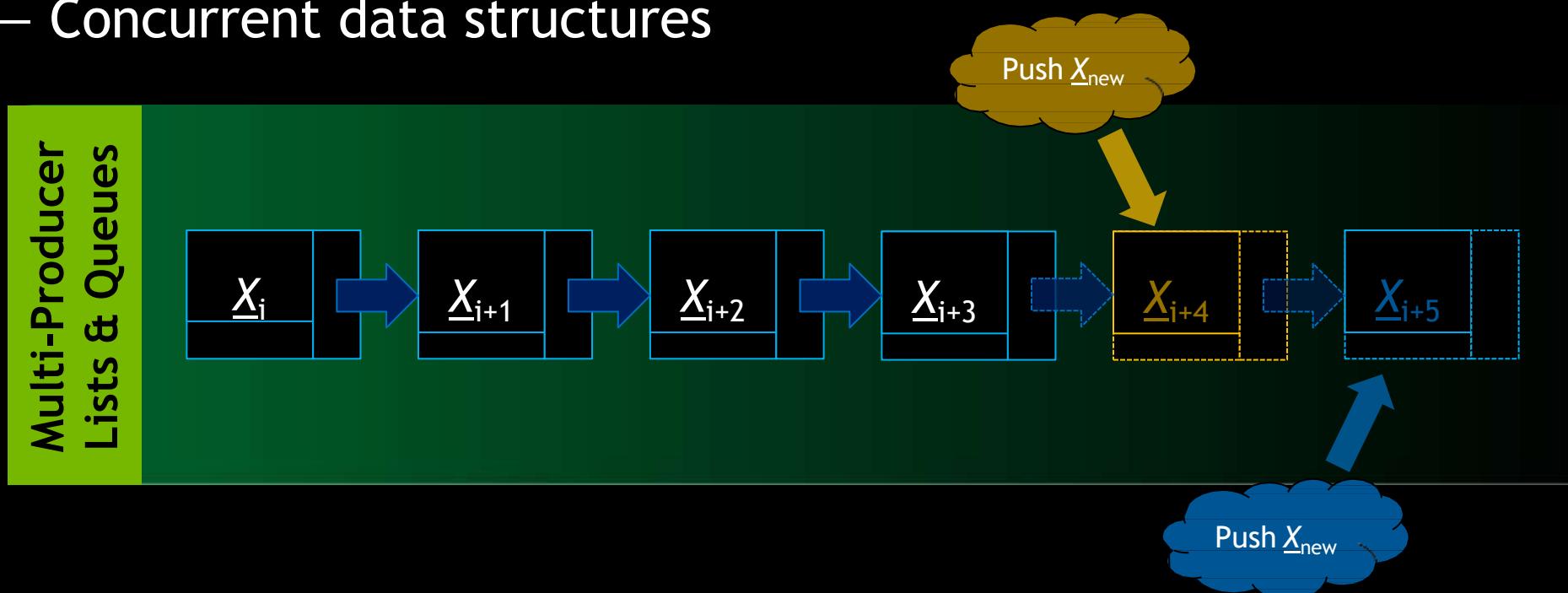
- Transactions & Data Access Control
- Data aggregation & enumeration



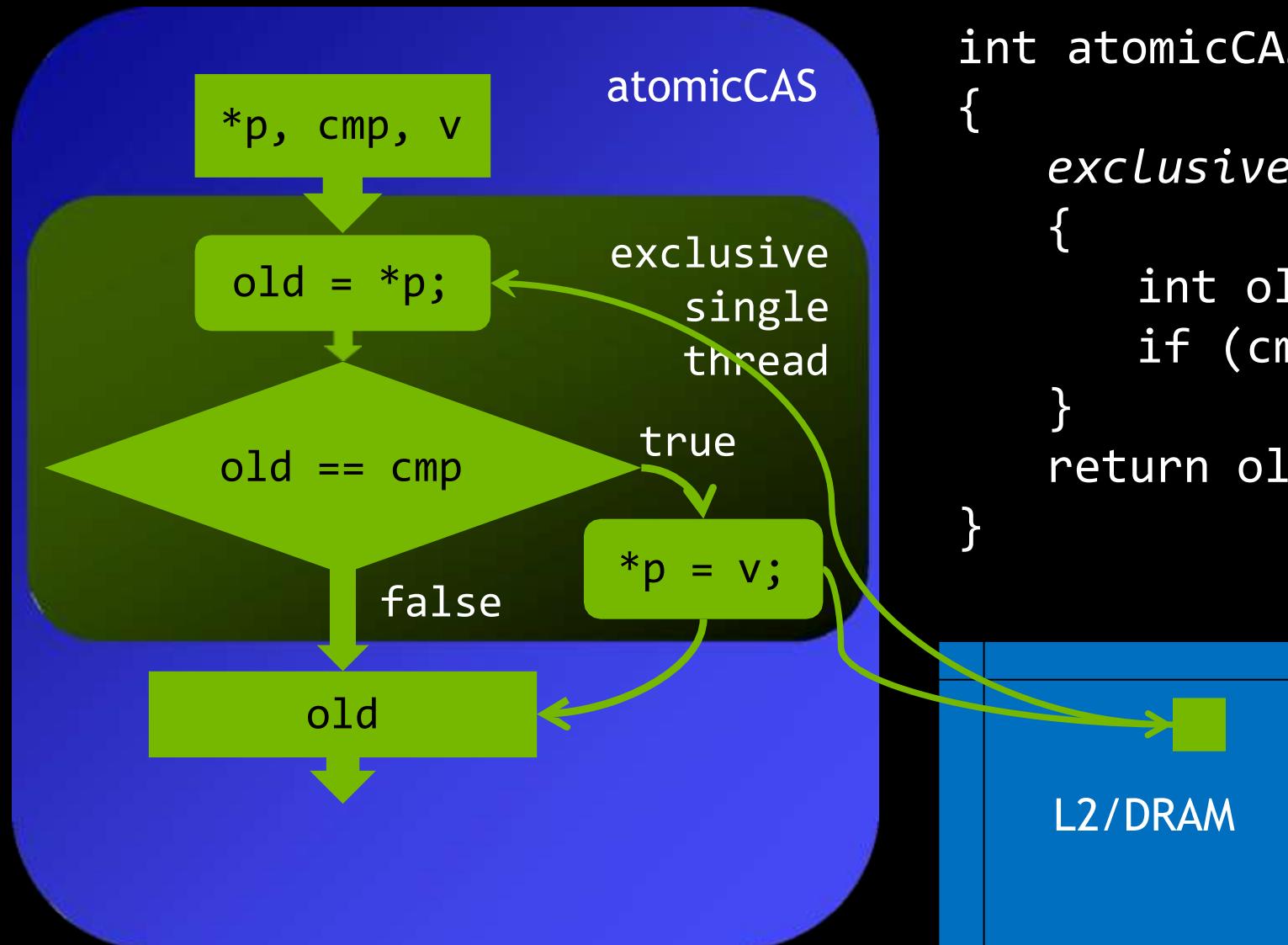
# Why Use Atomics?

Common problem: races on read-modify-write of shared data

- Transactions & Data Access Control
- Data aggregation & enumeration
- Concurrent data structures

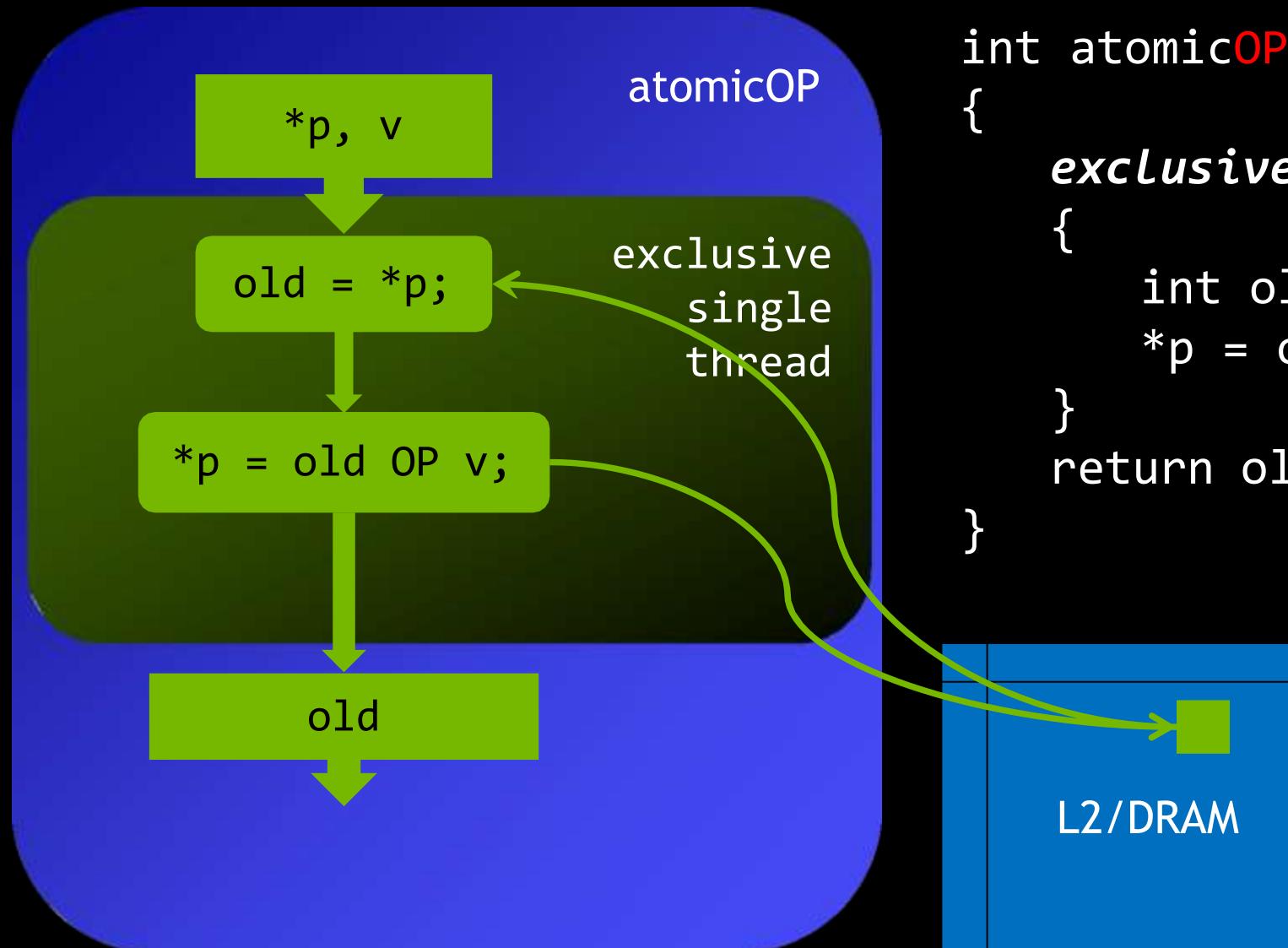


# Compare-and-Swap



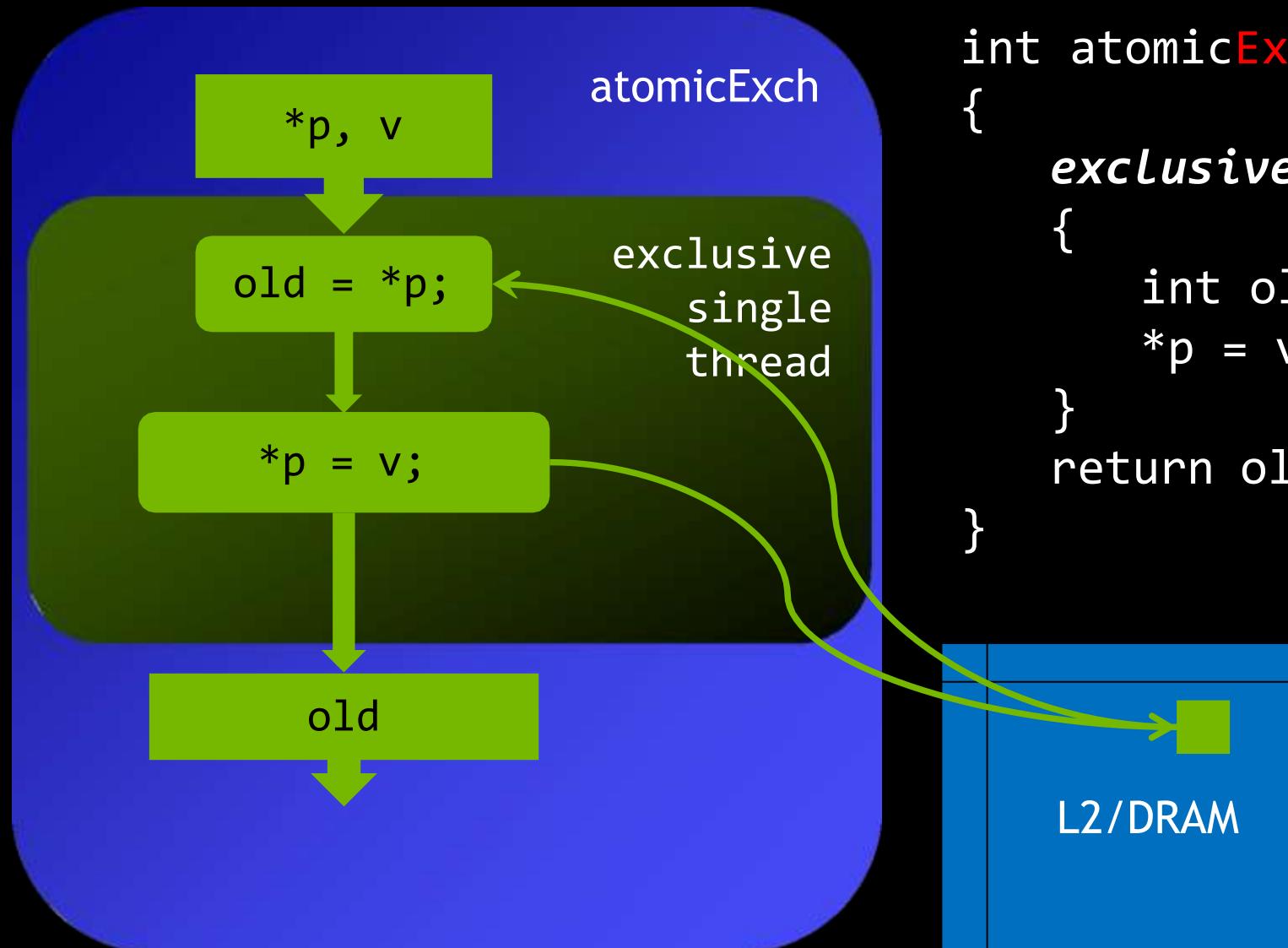
```
int atomicCAS(int *p, int cmp, int v
{
    exclusive_single_thread
    {
        int old = *p;
        if (cmp == old) *p = v;
    }
    return old;
}
```

# Arithmetic/Logical Atomic Operations



Binary Ops:  
**Add, Min, Max  
And, Or, Xor**

# Overwriting Atomic Operations



```
int atomicExch(int *p, int v)
{
    exclusive_single_thread
    {
        int old = *p;
        *p = v;
    }
    return old;
}
```

# Programming Styles using Coordination

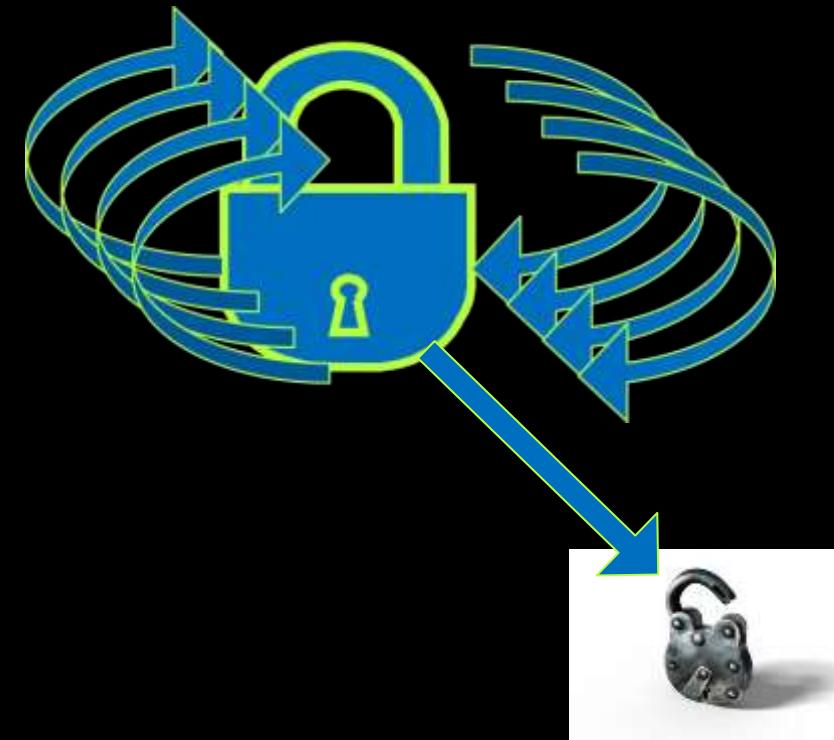
Locking

Lock-free

Wait-free

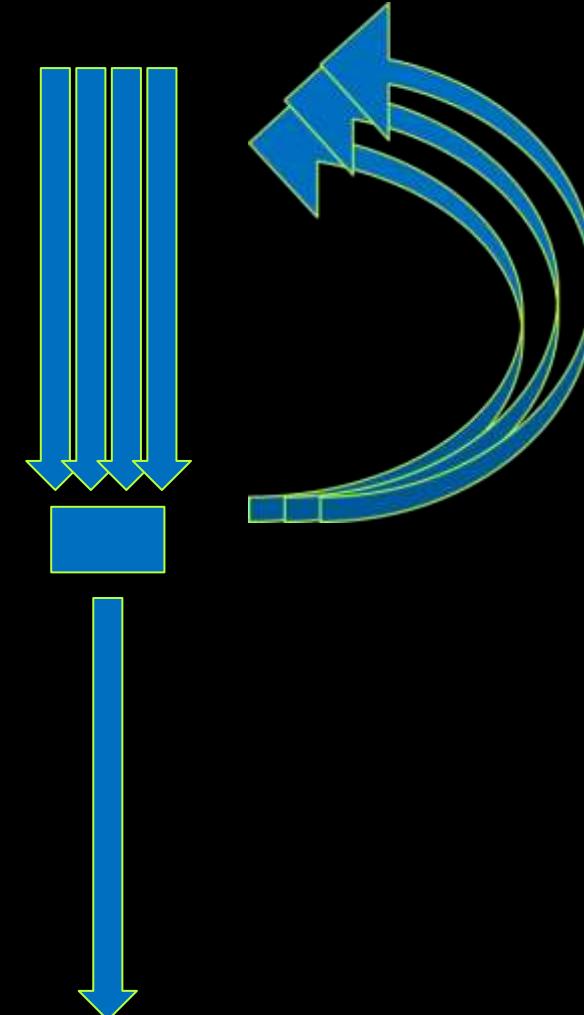
# Locking Style of Programming

- All threads try to get the lock
- One does
  - Does its work
  - Releases the lock



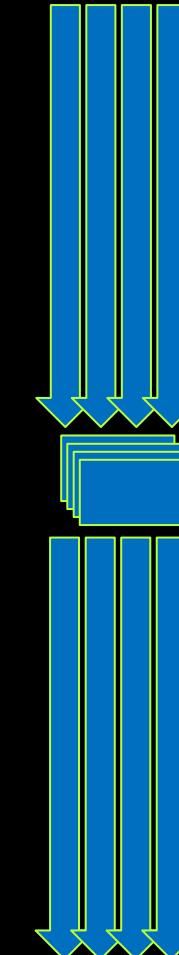
# Lock-Free Style of Programming

- At least one thread always makes progress
- Try to write their result
  - On failure, repeat
- Usually atomicCAS
  - atomicExch, atomicAdd also used

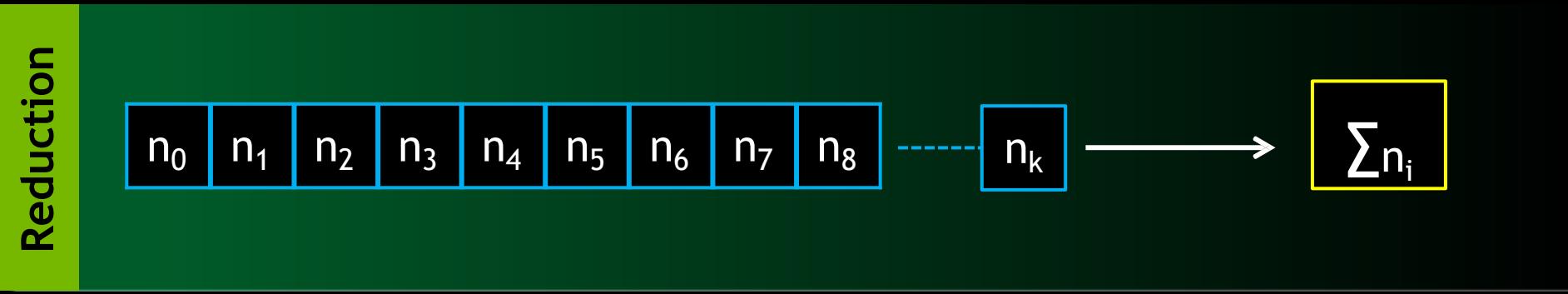


# Wait-free Style of Programming

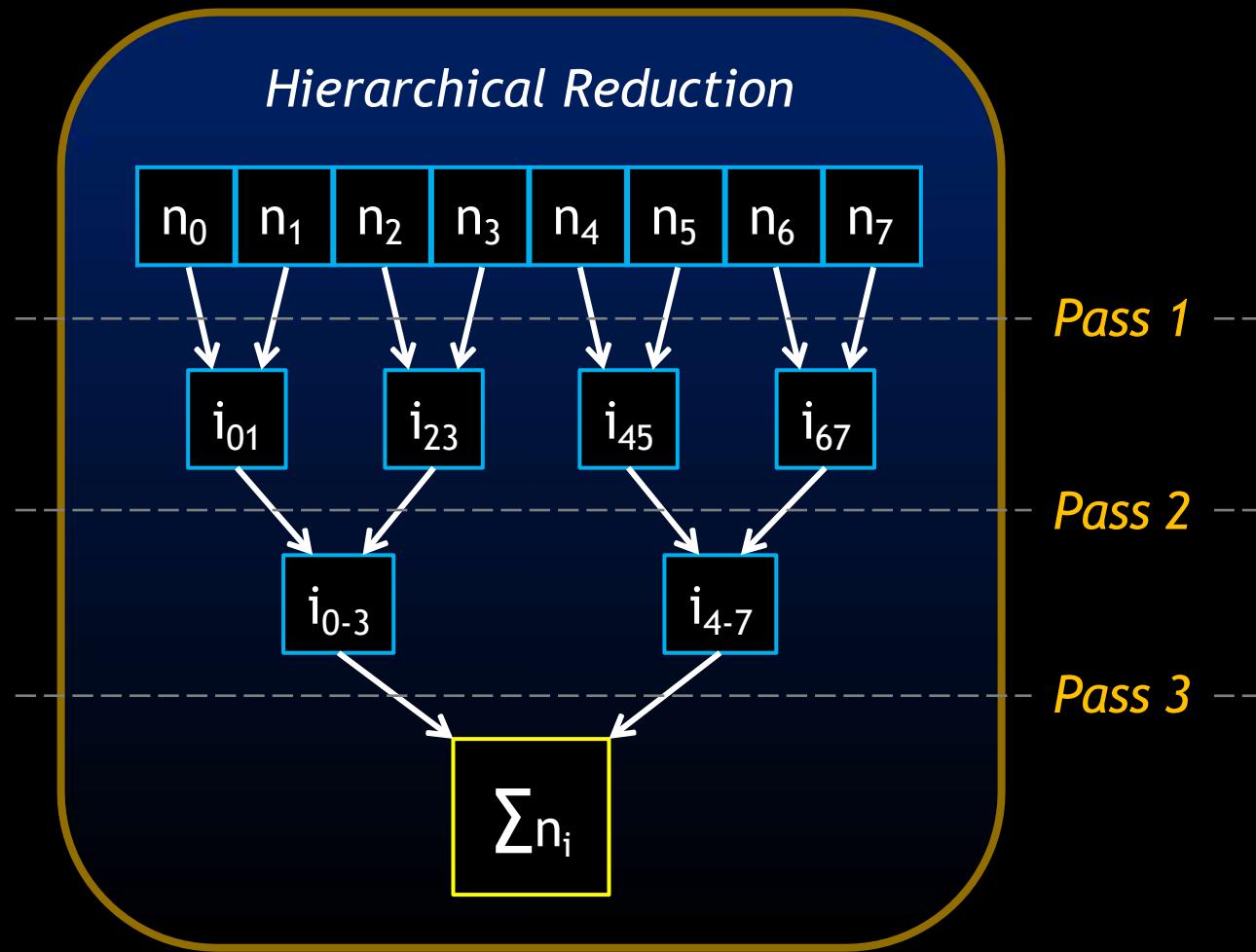
- All threads make progress
- Each updates memory atomically
- No thread blocked by other threads



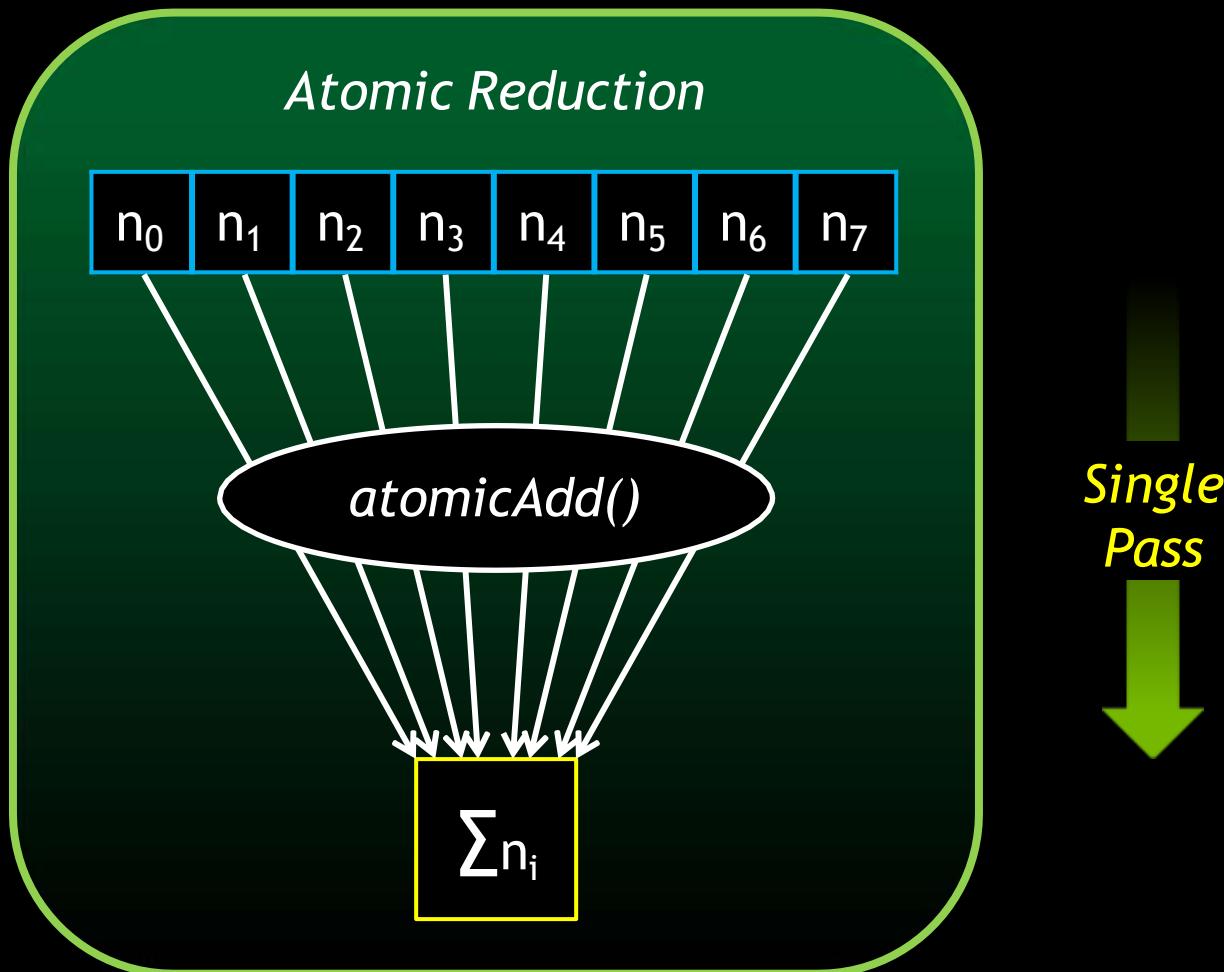
# Atomic Arithmetical Operations



# Atomic Arithmetical Operations



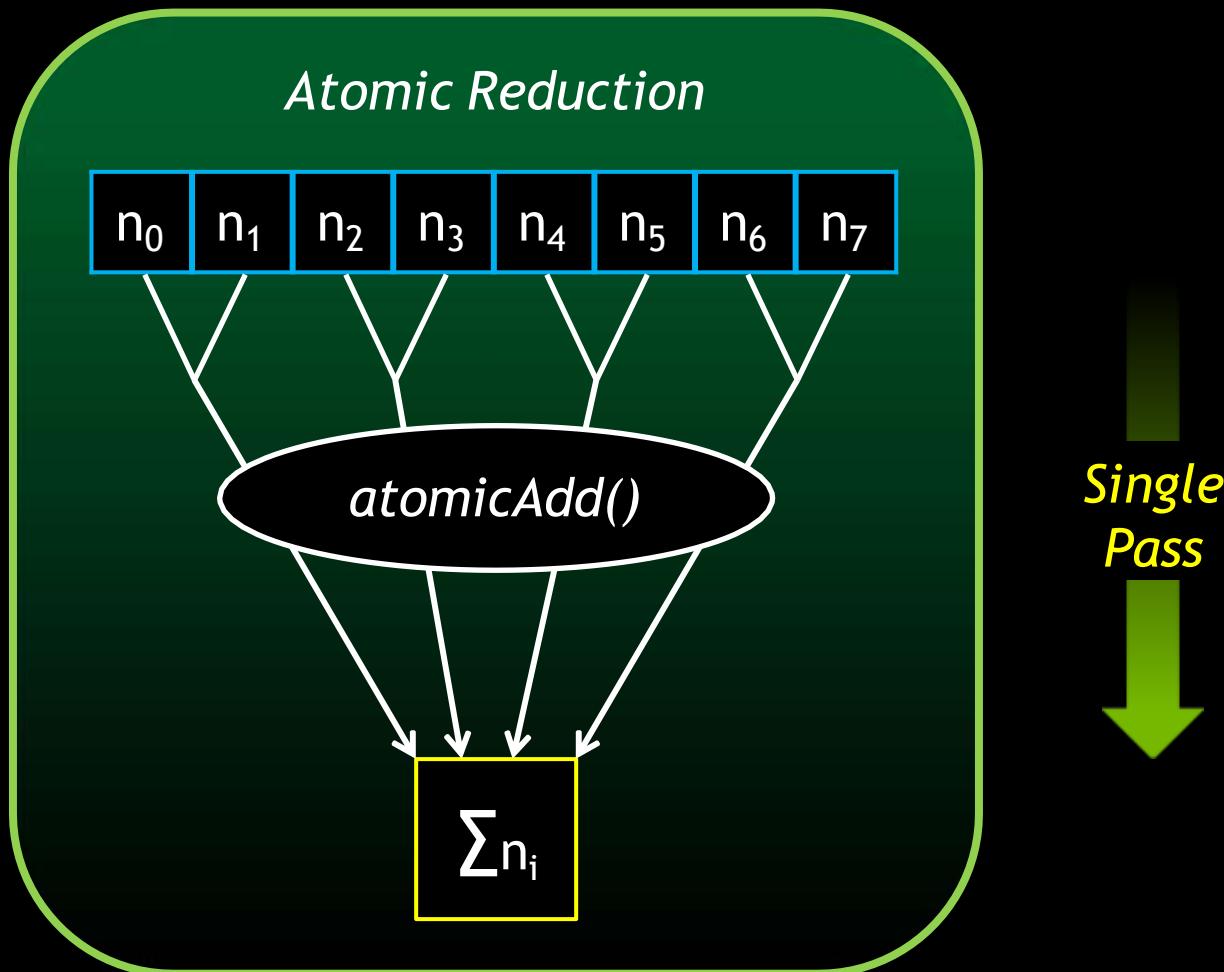
# Atomic Arithmetical Operations



*Single Pass*

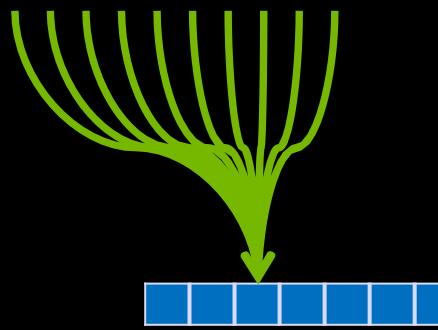
A vertical bar with a downward-pointing arrow, indicating a single pass or iteration through the data.

# Atomic Arithmetical Operations



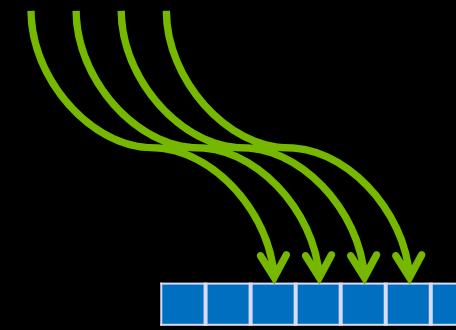
# Atomic Access Patterns

*Same Address*



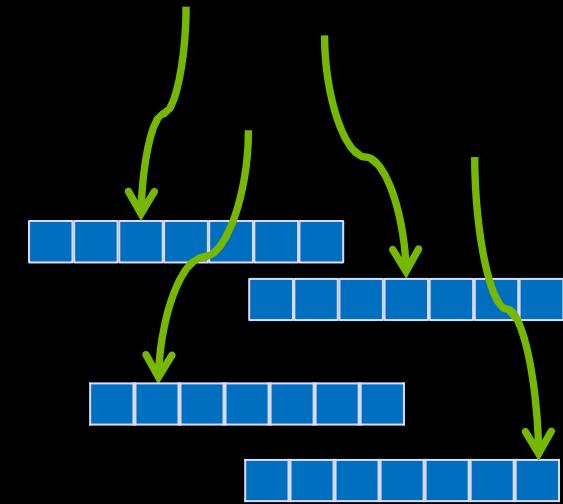
- 1 per clock

*Same Cache Line*



- Adjacent addresses
- Same issuing warp
- 8 per SM per clock

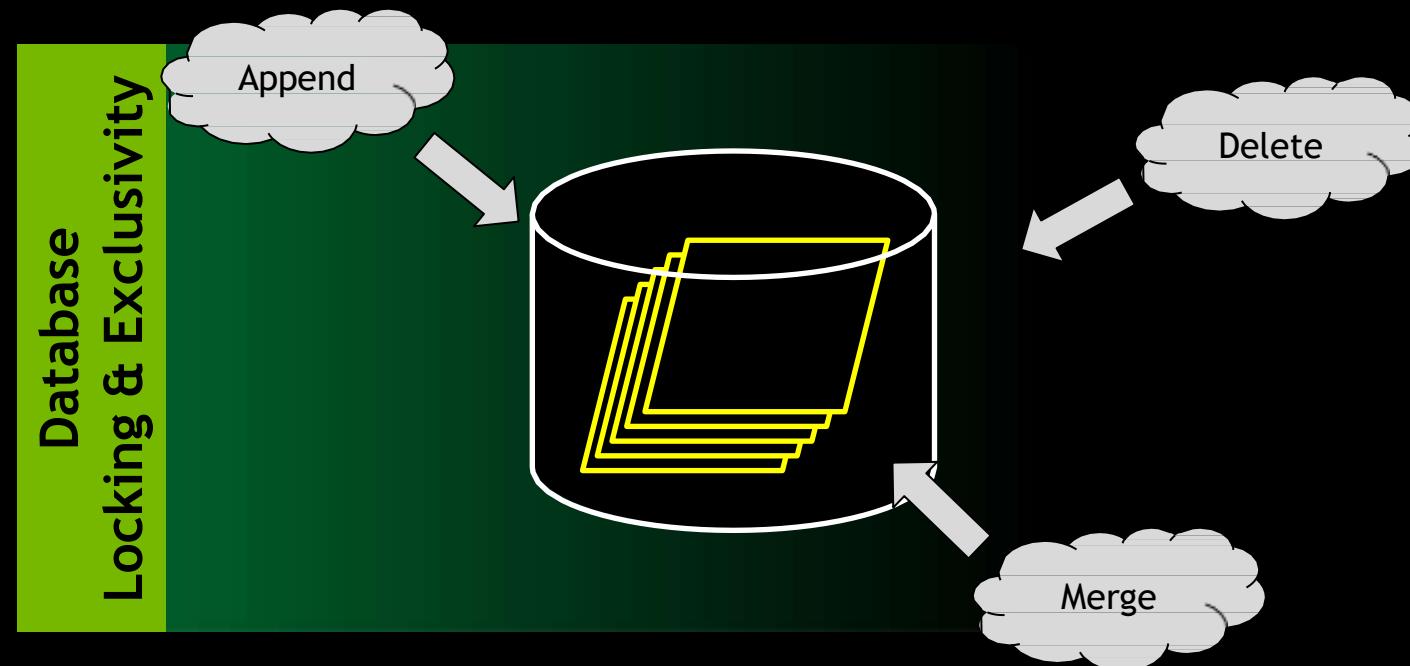
*Scattered*



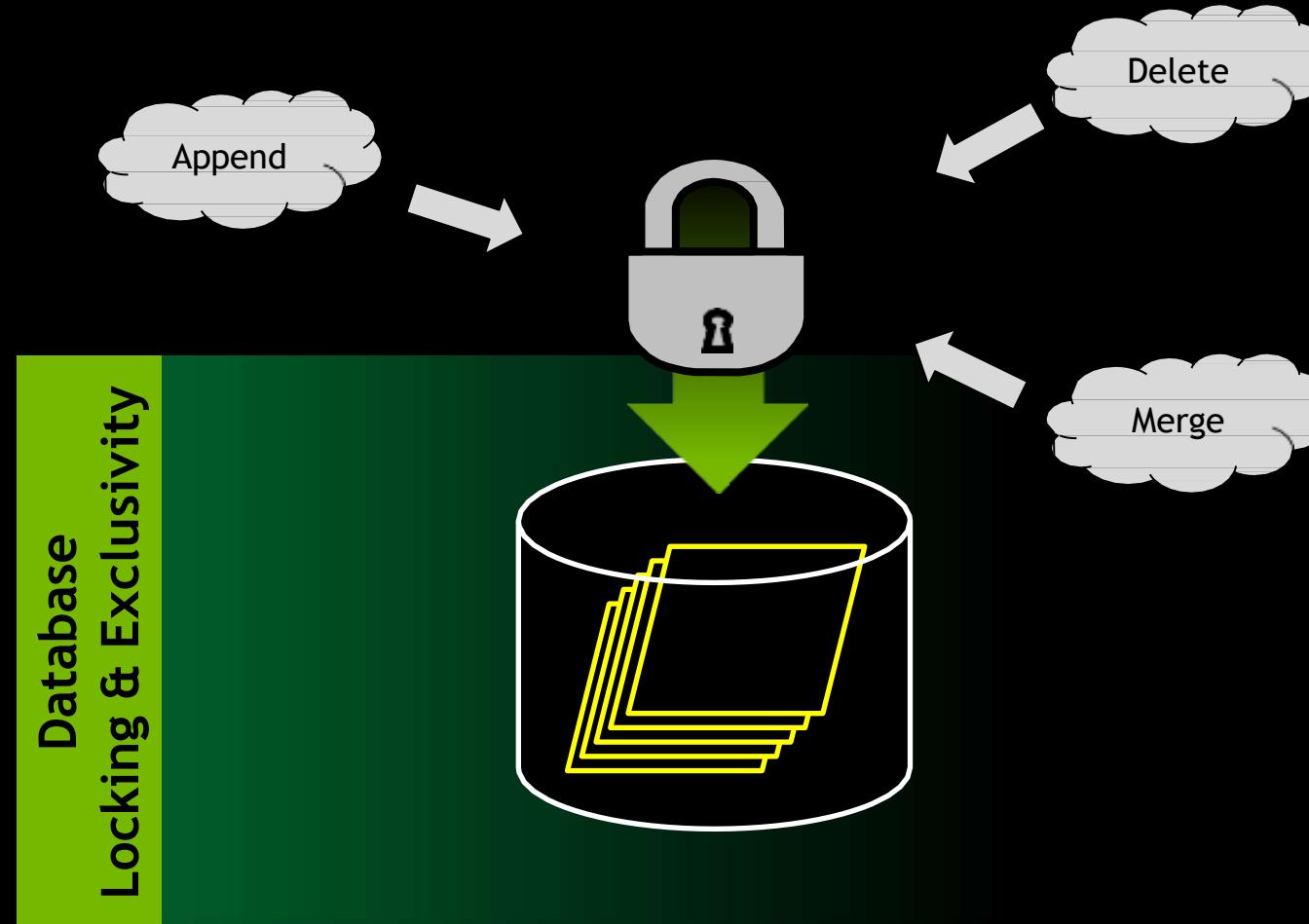
- Issued per cache-line
- 1 per SM per clock

# Locks & Access Control

Locking guarantees exclusive access to data



# Locks & Access Control



# Locks & Access Control

## Multi-threaded arithmetic

- Double precision addition
- Simple code is unsafe

```
// Add "val" to "*data". Return old value.  
double atomicAdd(double *data, double val)  
{  
  
    double old = *data;  
    *data = old + val;  
  
    return old;  
}
```

# Locks & Access Control

## Multi-threaded arithmetic

- Double precision addition
- Simple code is unsafe
- Add locks to protect critical section

```
// Add "val" to "*data". Return old value.  
double atomicAdd(double *data, double val)  
{  
    while(try_lock() == false)  
        ; // Retry lock  
  
    double old = *data;  
    *data = old + val;  
    unlock();  
  
    return old;  
}
```

# Locks & Access Control

```
int locked = 0;
bool try_lock()
{
    if(locked == 0) {
        locked = 1;
        return true;
    }
    return false;
}
```

```
// Add “val” to “*data”. Return old value.
double atomicAdd(double *data, double val)
{
    while(try_lock() == false)
        ;      // Retry lock

    double old = *data;
    *data = old + val;
    unlock();

    return old;
}
```

# Locks & Access Control

```
int locked = 0;
bool try_lock()
{
    int prev = atomicExch(&locked, 1);
    if(prev == 0)
        return true;

    return false;
}
```

*int atomicExch(int \*data, int new)*

Atomically set (\*data = new), and return  
the previous value

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(try_lock() == false)
        ;      // Retry lock

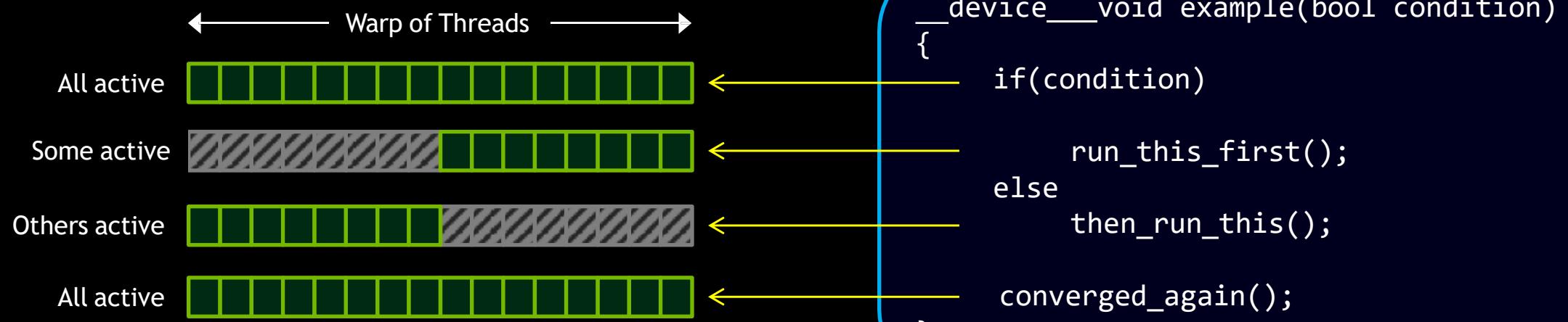
    double old = *data;
    *data = old + val;
    unlock();

    return old;
}
```

# Locks & Warp Divergence

A CUDA *warp*:

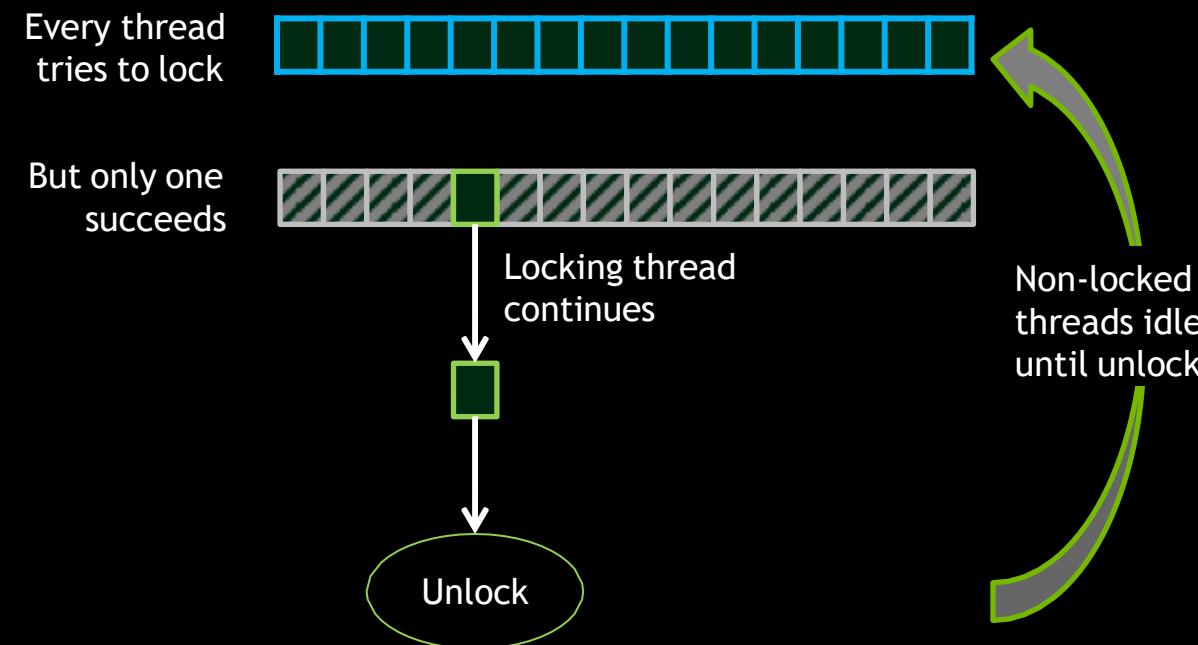
- A group of threads (32 on current GPUs) scheduled in lock-step
- All threads execute the same line of code
- Any thread not participating is idle



# Locks & Warp Divergence

What does this mean for locks?

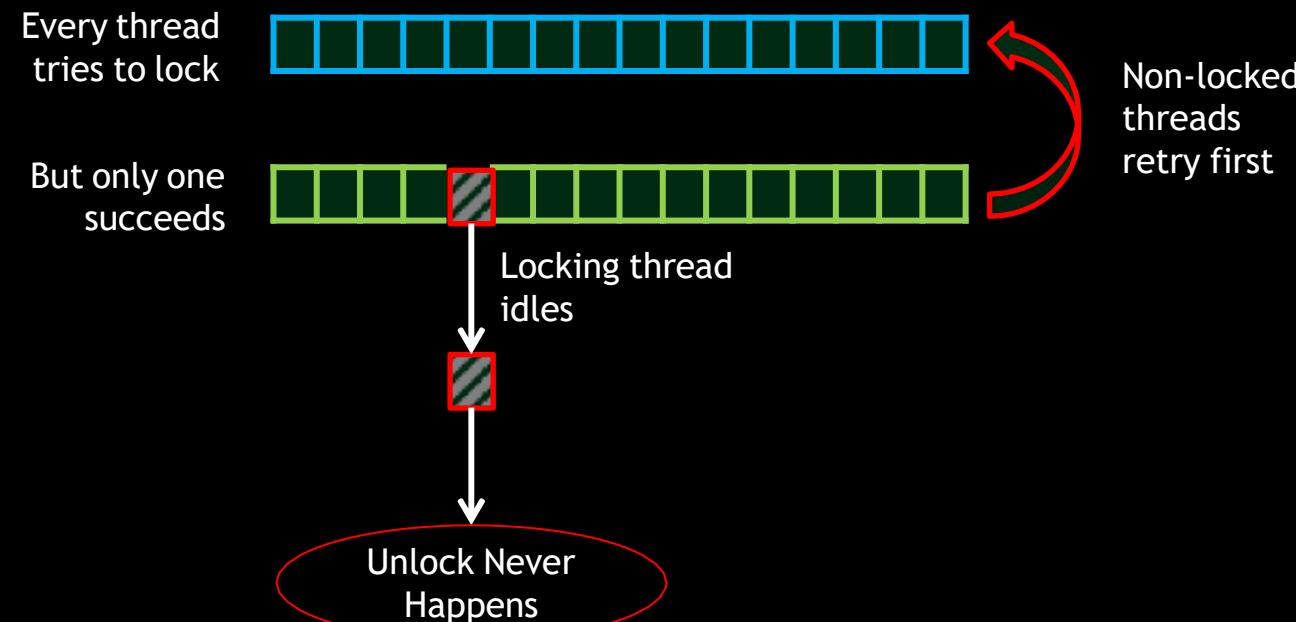
- Only one thread in the warp will lock
- We're okay so long as that's the thread which continues



# Locks & Warp Divergence

What does this mean for locks?

- BUT: If the wrong thread idles, we deadlock
- No way to predict which threads idle



# Locks & Warp Divergence

Working around divergence deadlock

1. Don't use locks between threads in a warp
2. Elect one thread to take the lock, then iterate
3. Use a lock-free algorithm...

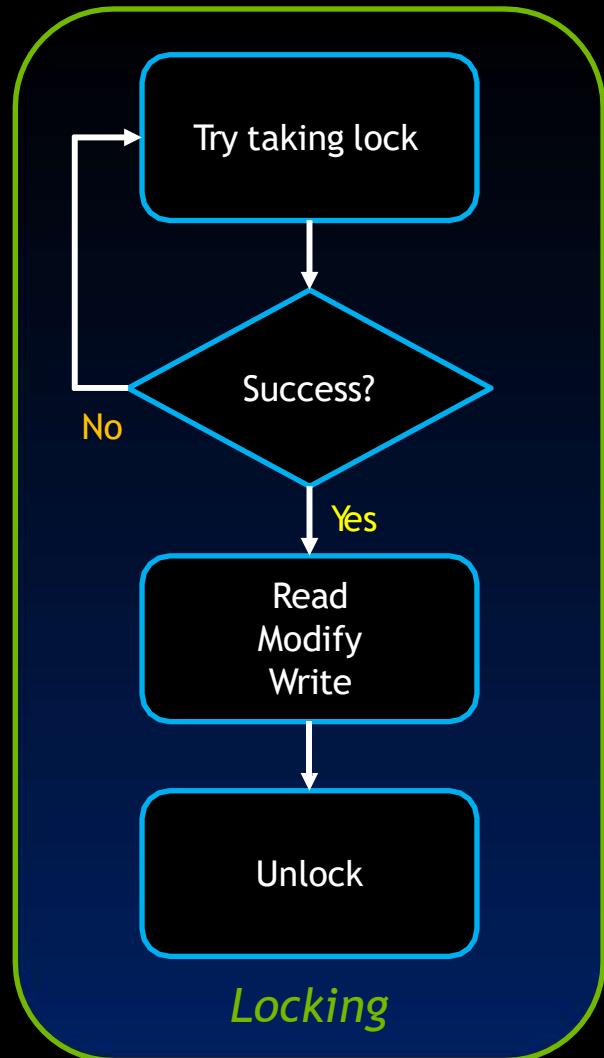
# Lock Free Algorithms: Better Than Locks

Use atomic compare-and-swap to combine read, modify, write

- Under contention, exactly one thread is guaranteed to succeed
- High throughput - less work in critical section
- Only applies if transaction is a single operation

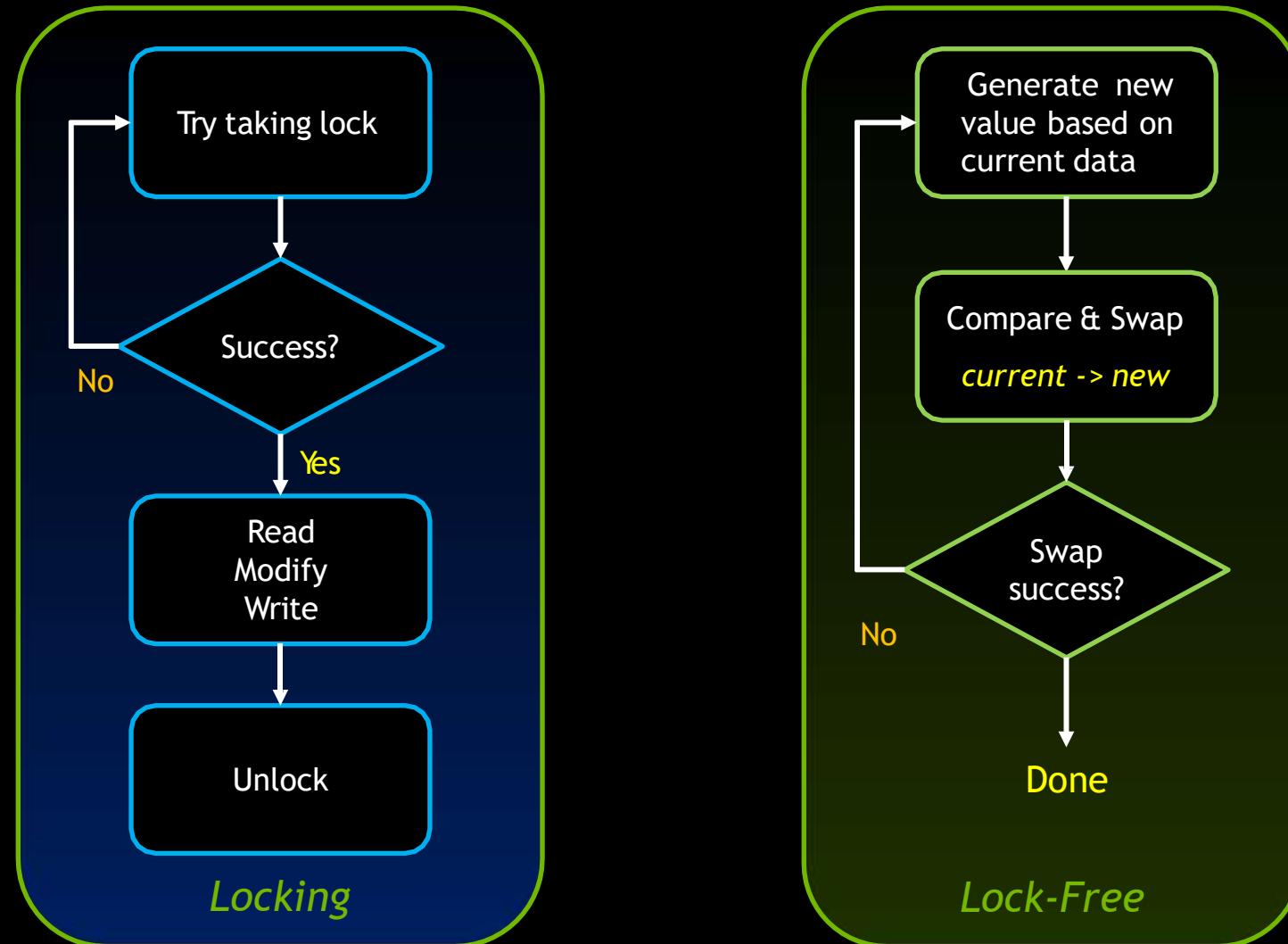
```
uint64 atomicCAS(uint64 *data, uint64 oldval, uint64 newval);  
If “*data” is equal to “oldval”, replace it with “newval”  
Always returns original value of “*data”
```

# Lock-Free Data Updates

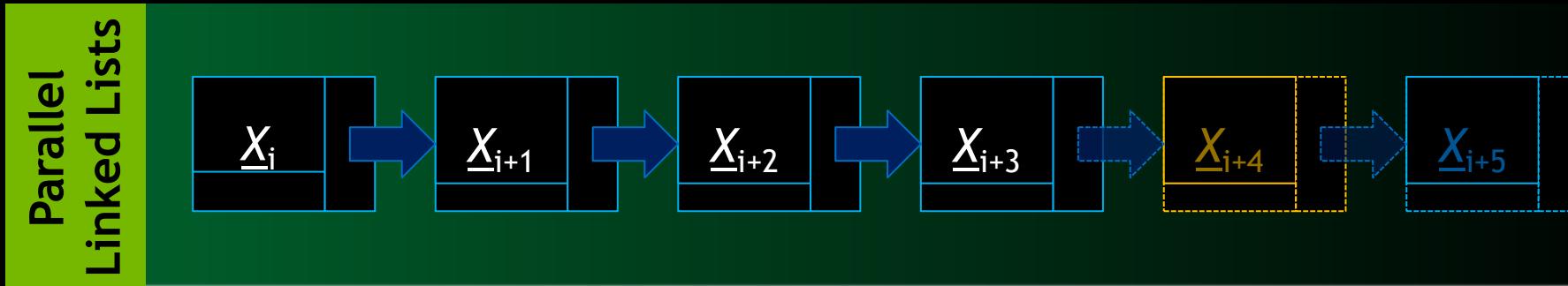


```
// Add "val" to "*data". Return old value.  
double atomicAdd(double *data, double val)  
{  
    while(atomicExch(&locked, 1) != 0)  
        ; // Retry lock  
  
    double old = *data;  
    *data = old + val;  
    locked = 0;  
  
    return old;  
}
```

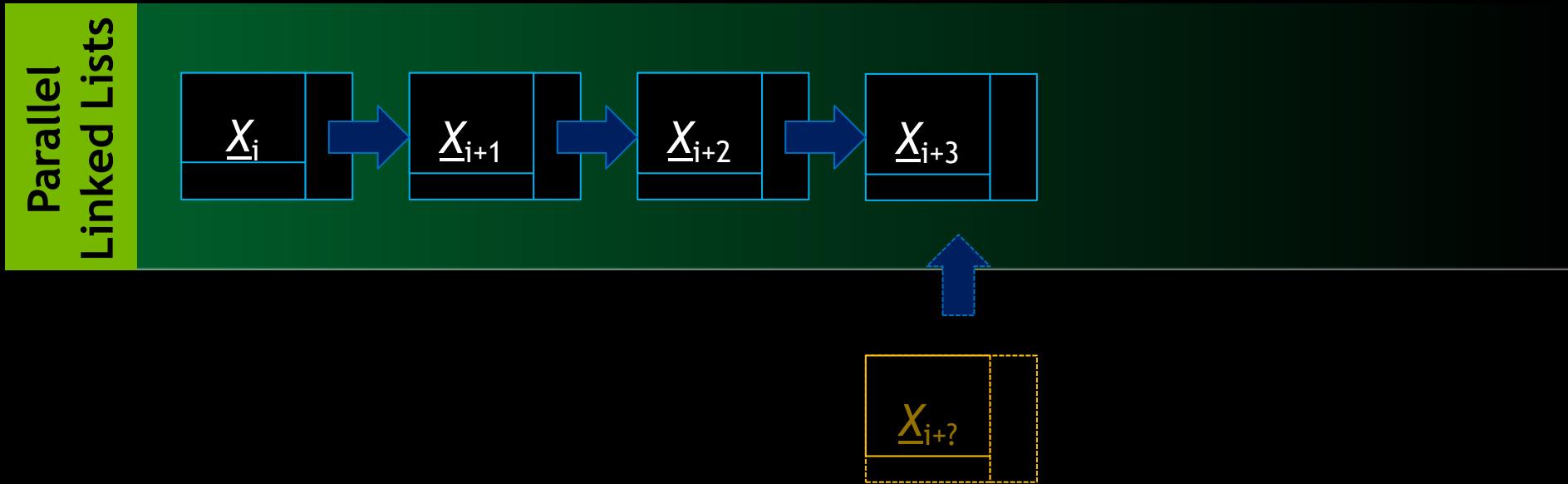
# Lock-Free Data Updates



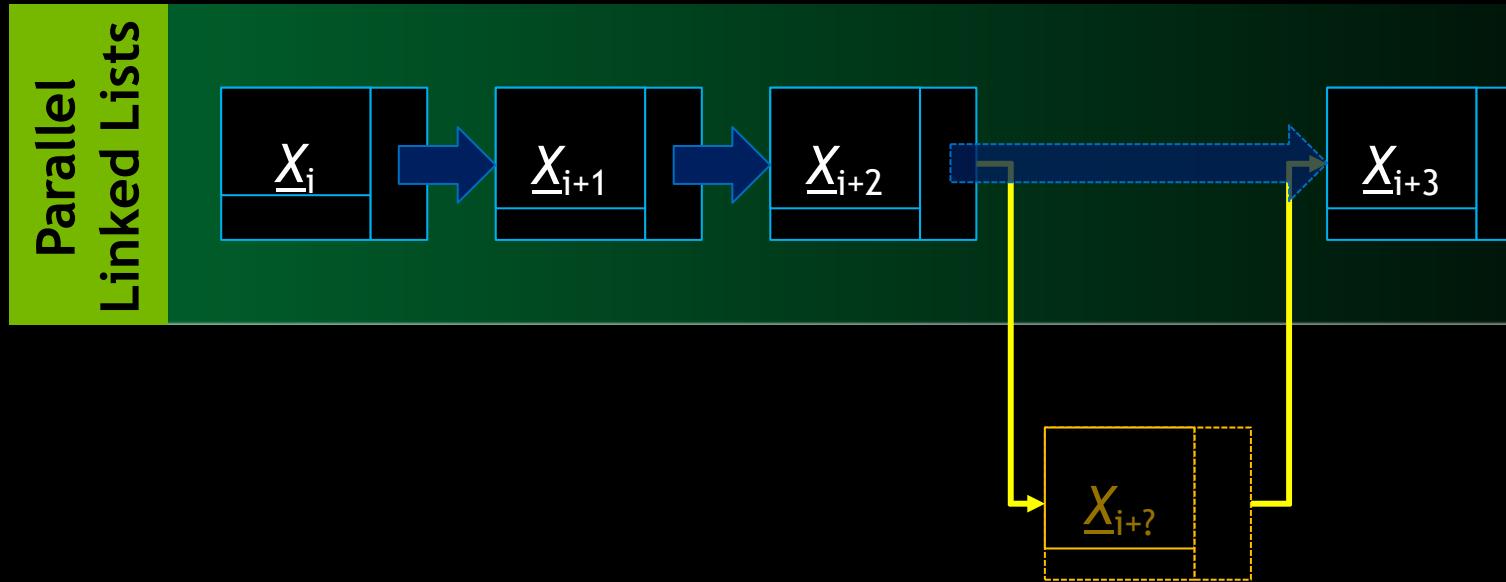
# Lock-Free Parallel Data Structures



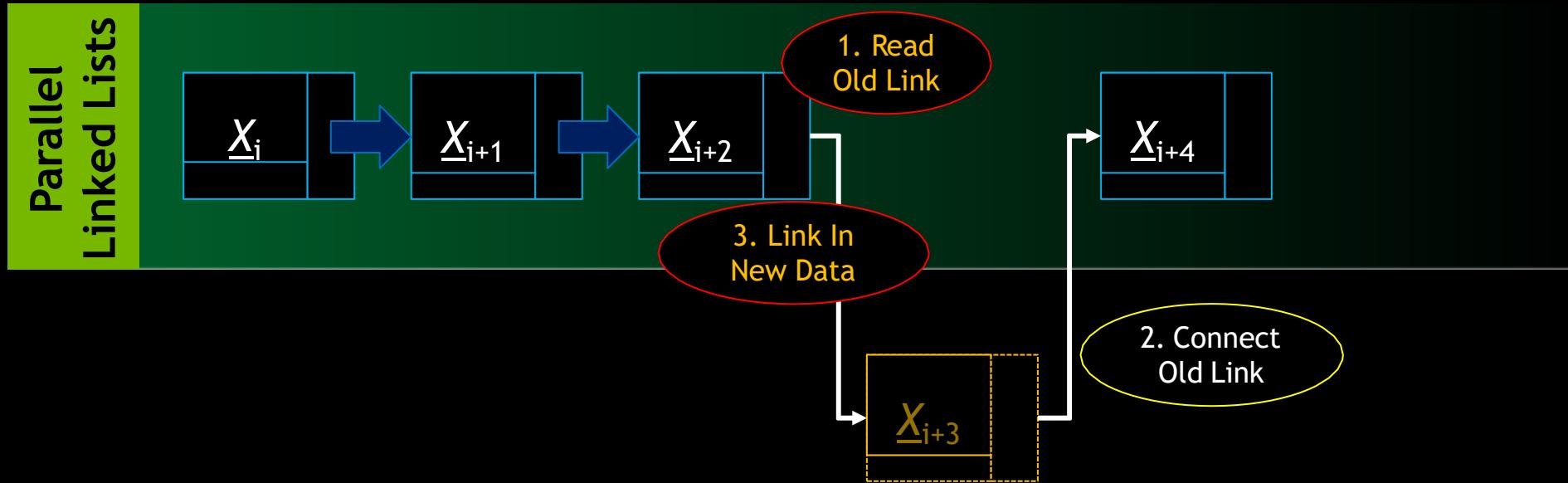
# Lock-Free Parallel Data Structures



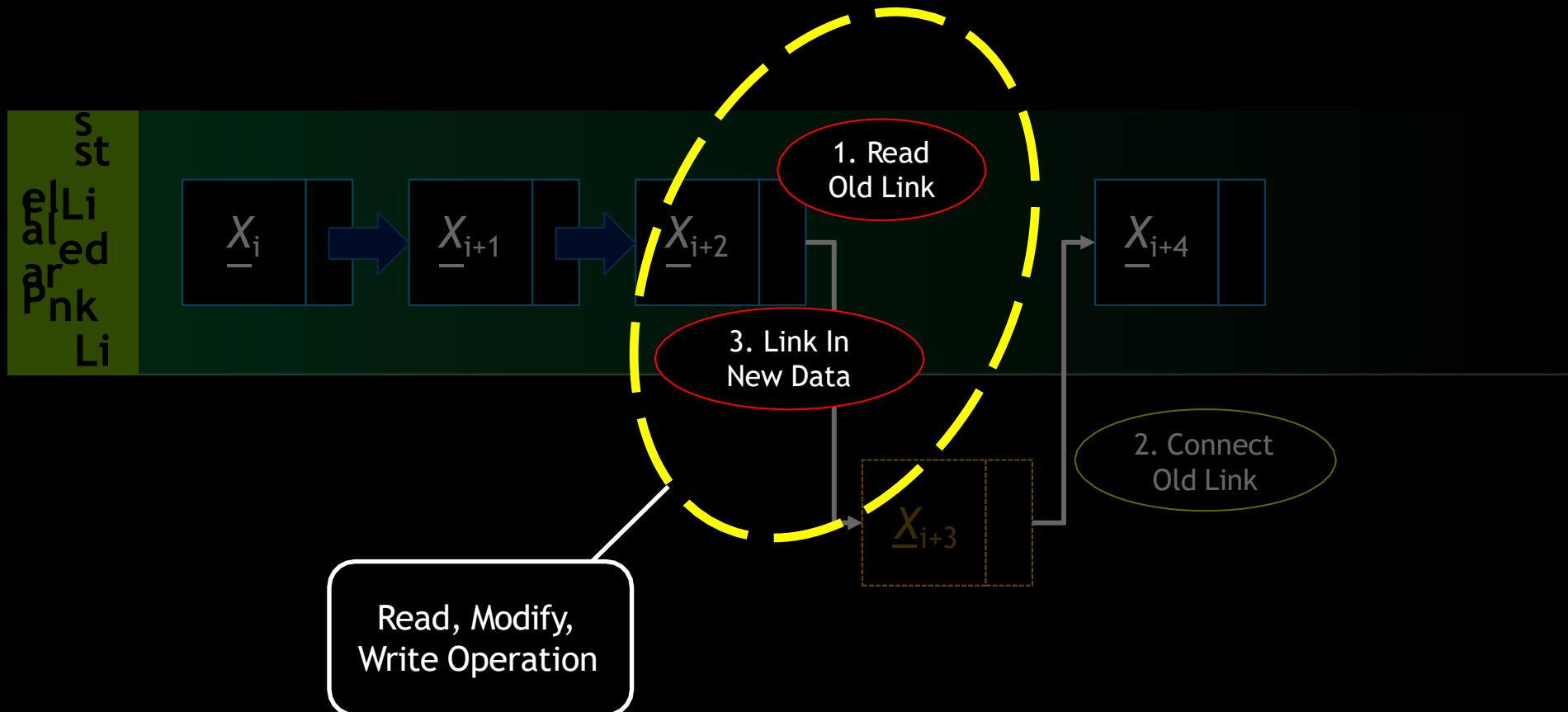
# Lock-Free Parallel Data Structures



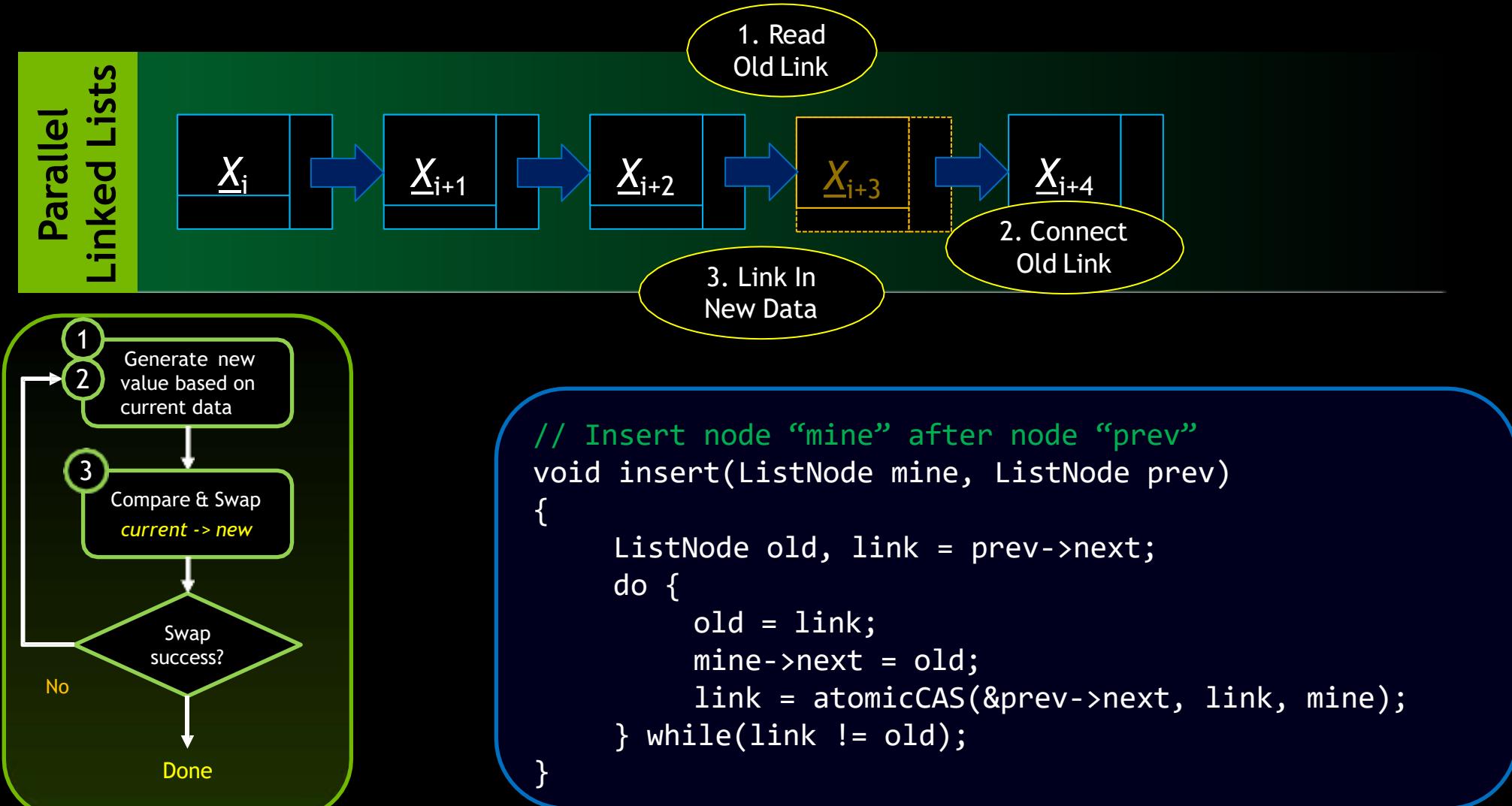
# Lock-Free Parallel Data Structures



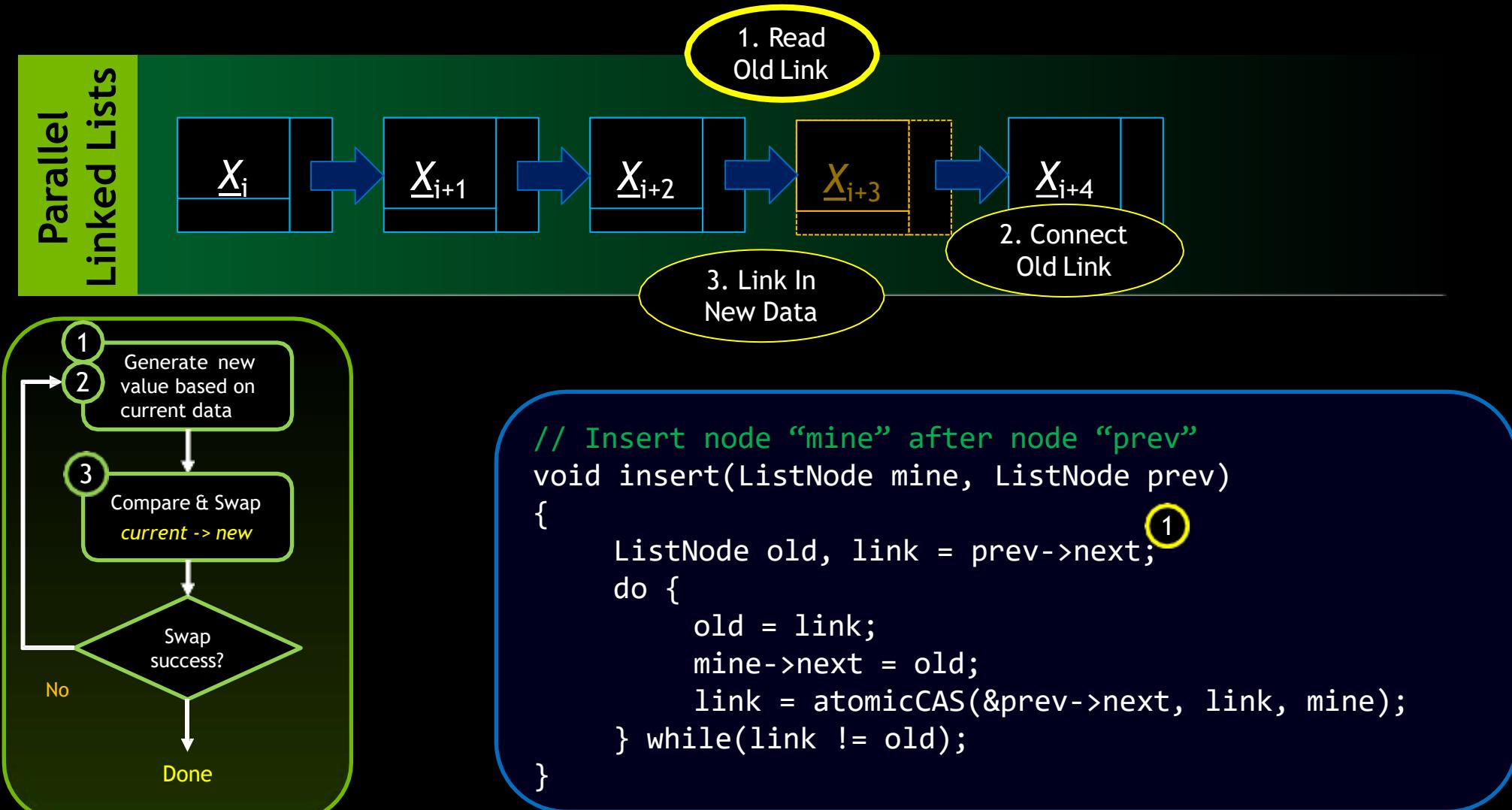
# Lock-Free Parallel Data Structures



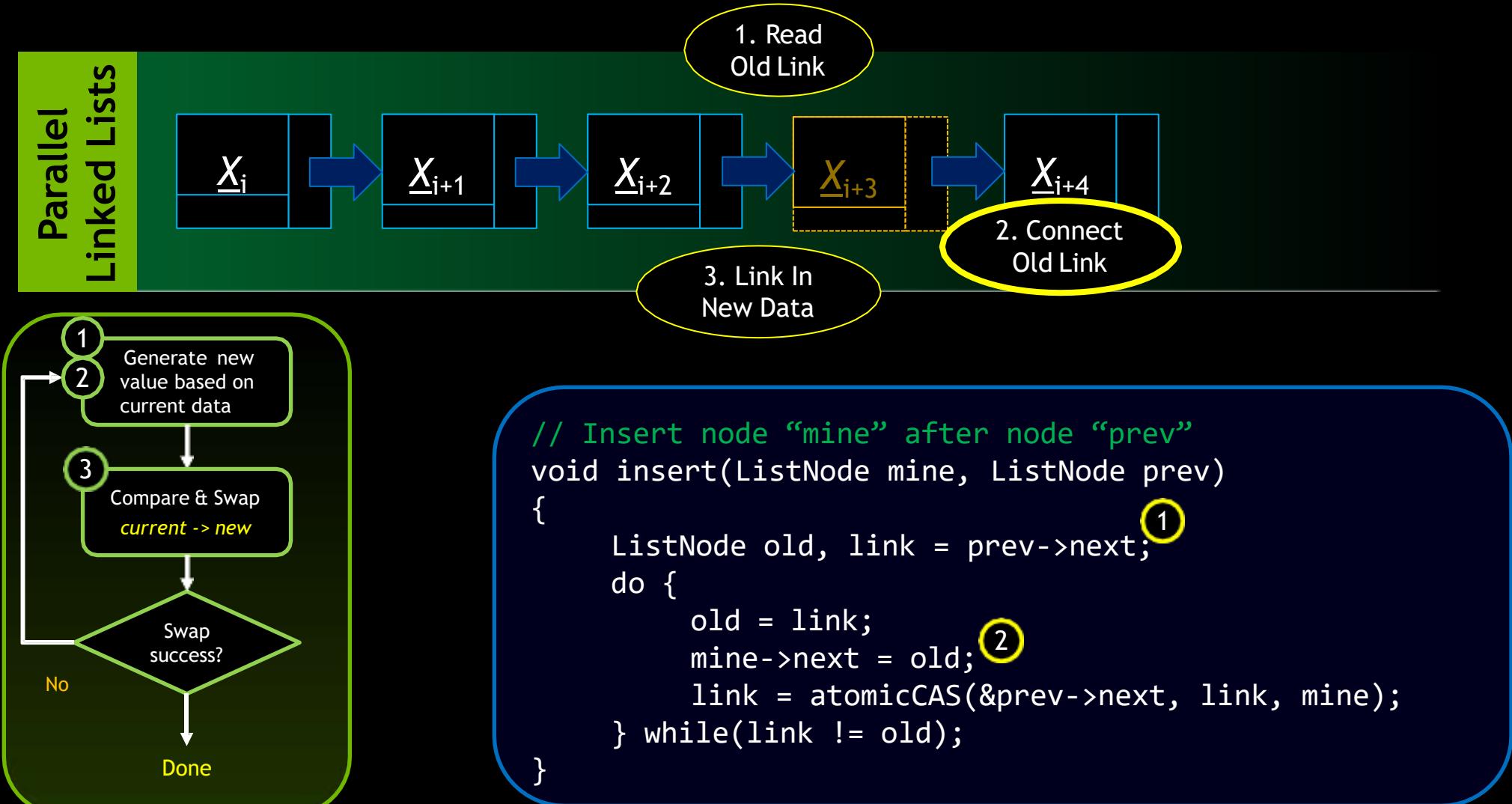
# Lock-Free Parallel Data Structures



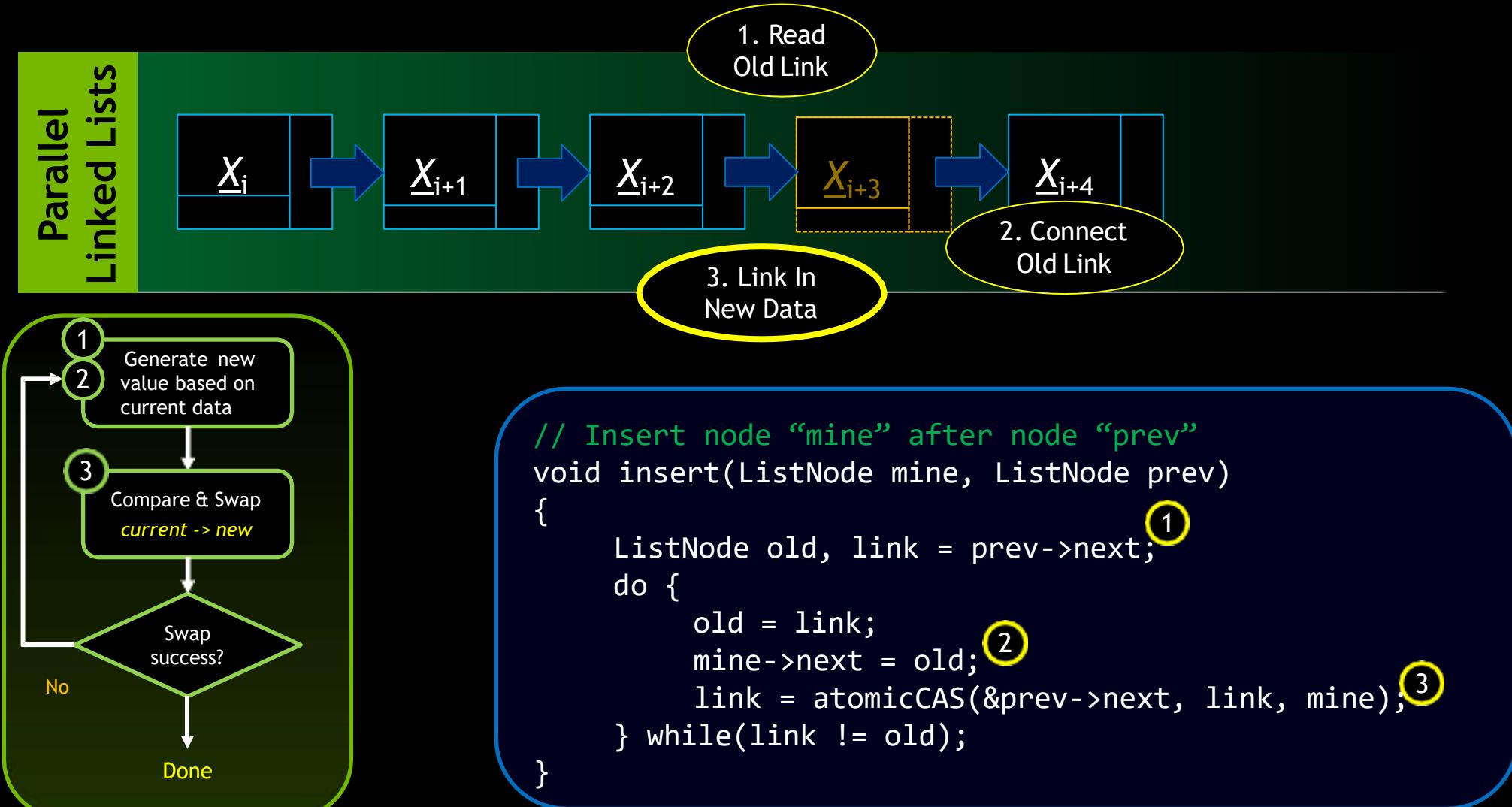
# Lock-Free Parallel Data Structures



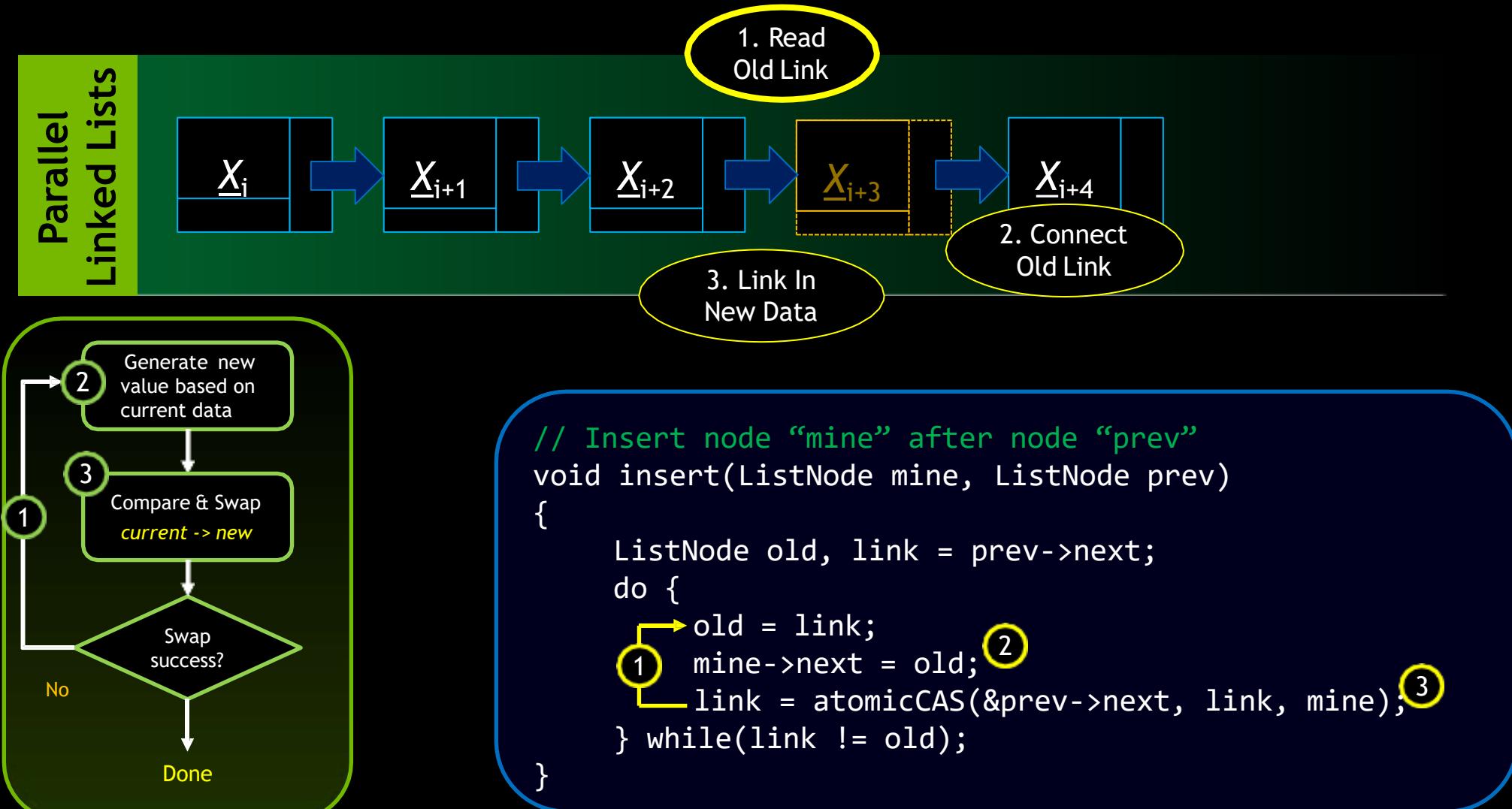
# Lock-Free Parallel Data Structures



# Lock-Free Parallel Data Structures



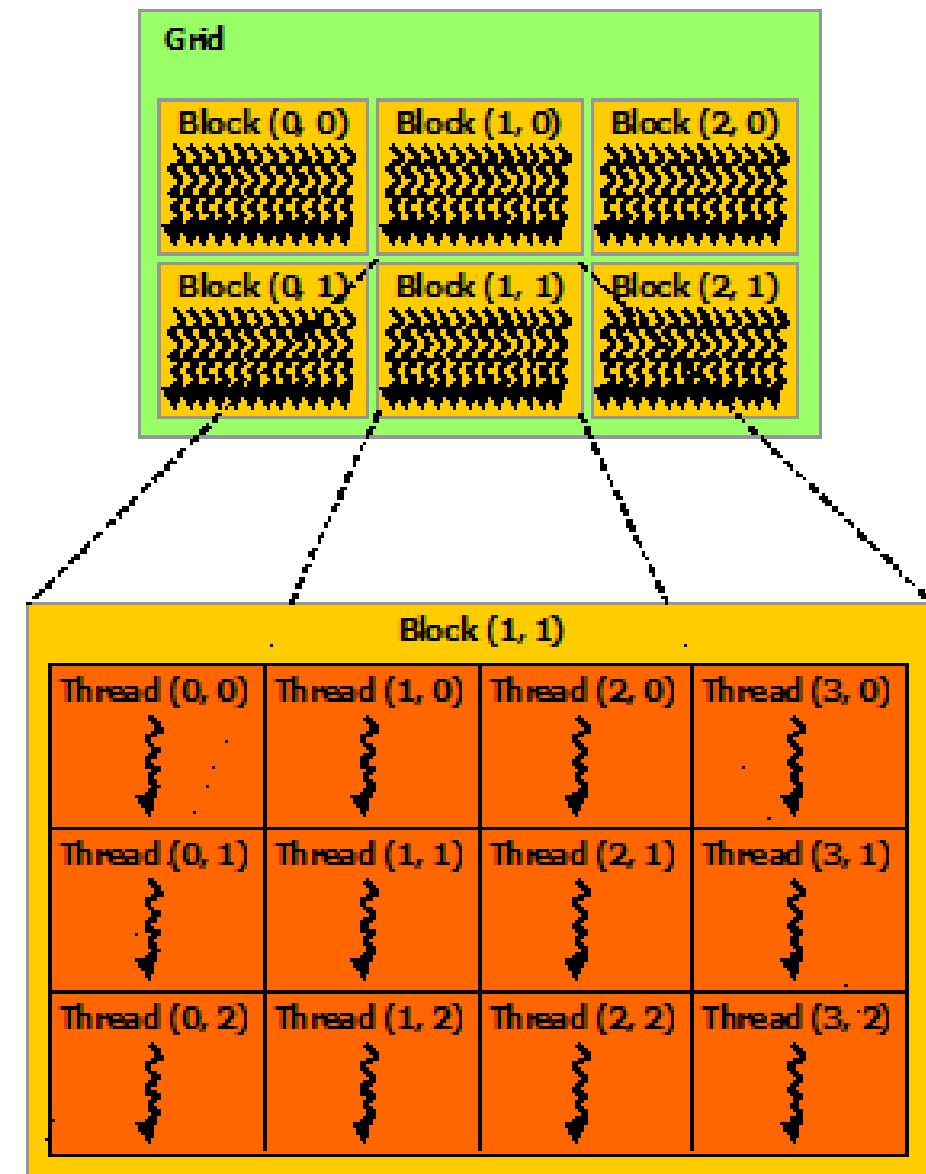
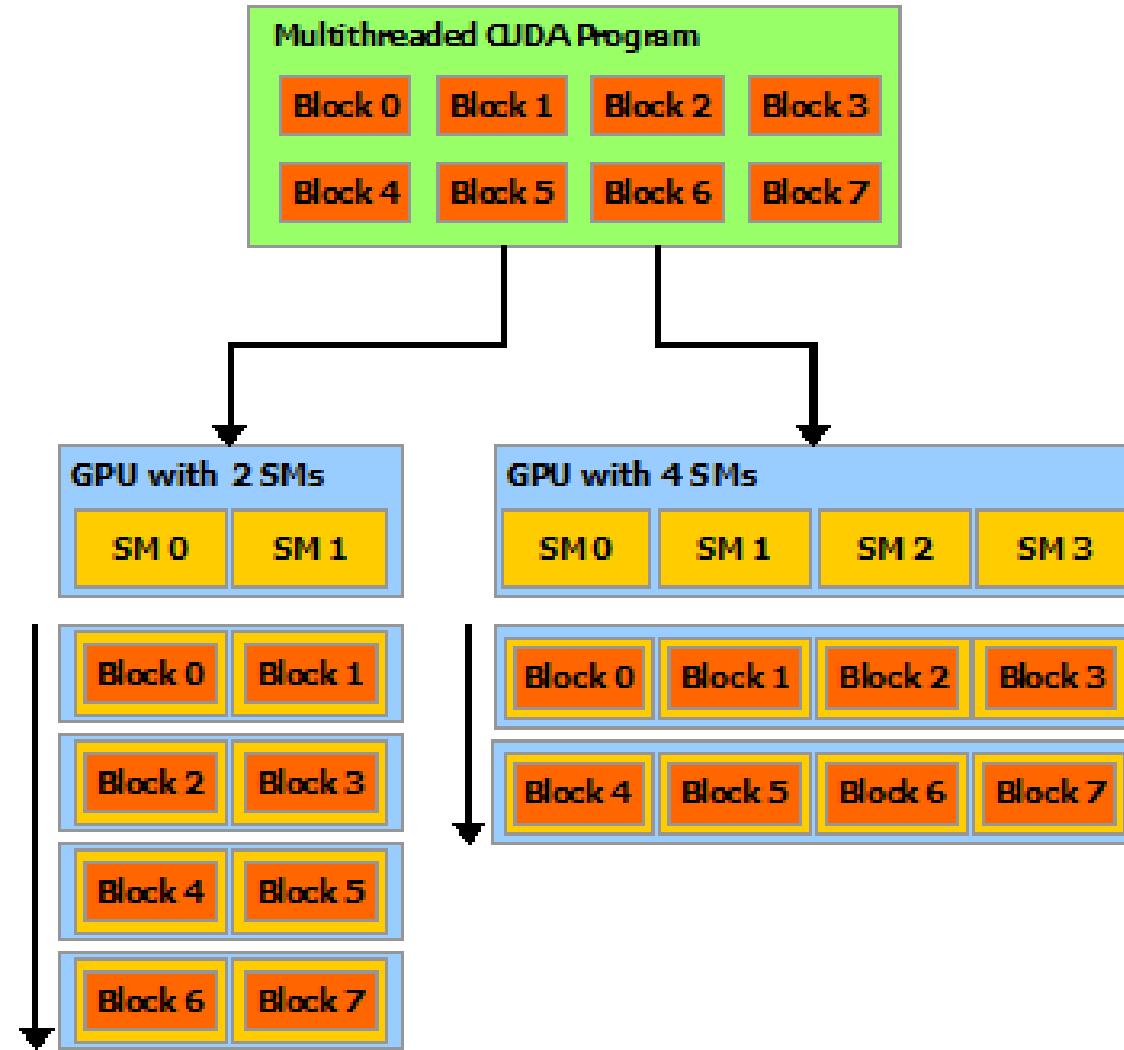
# Lock-Free Parallel Data Structures



# Conclusions

- Atomics allow the creation of much more sophisticated algorithms that have higher performance
- GPU has parallel hardware to execute atomics
- AtomicCAS can be used to mimic any coordination primitive
- Atomics force serialization
  - don't ask for serialization when you don't need it
  - or, perform concurrent reductions when possible

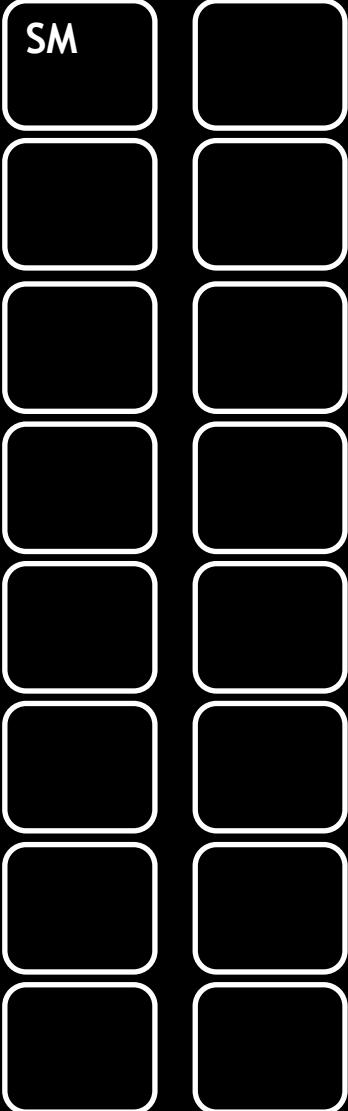
# Streaming Multiprocessors



# GPU

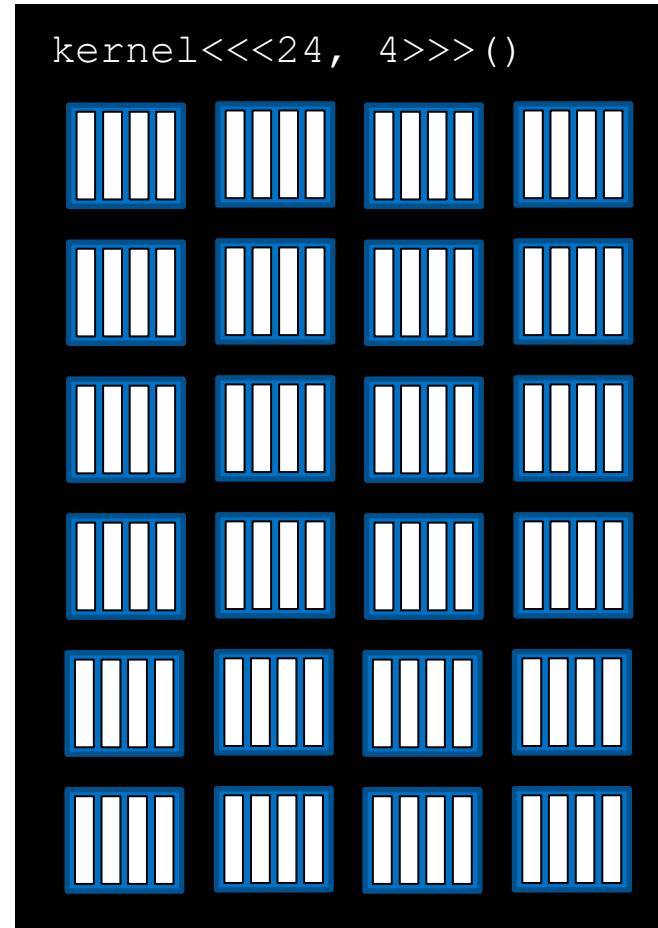
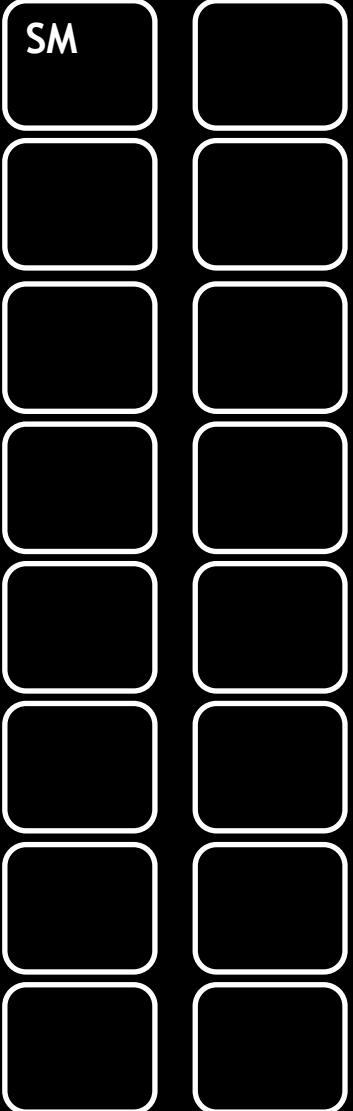
NVIDIA GPUs contain functional units called **Streaming Multiprocessors**, or **SMs**

# GPU



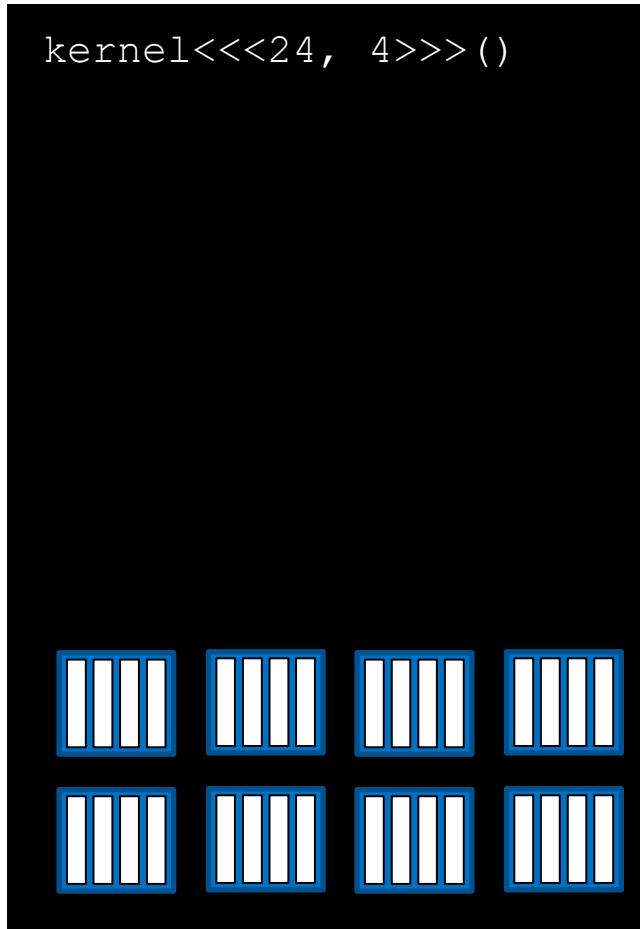
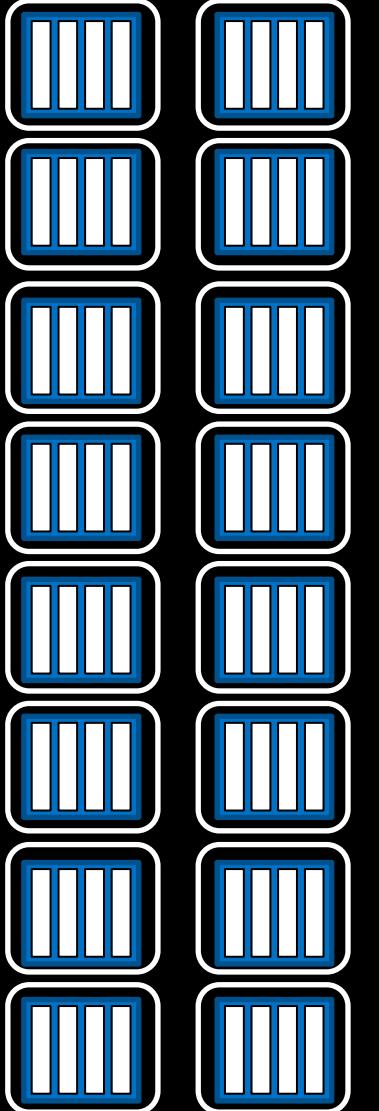
NVIDIA GPUs contain functional units called **Streaming Multiprocessors**, or **SMs**

# GPU



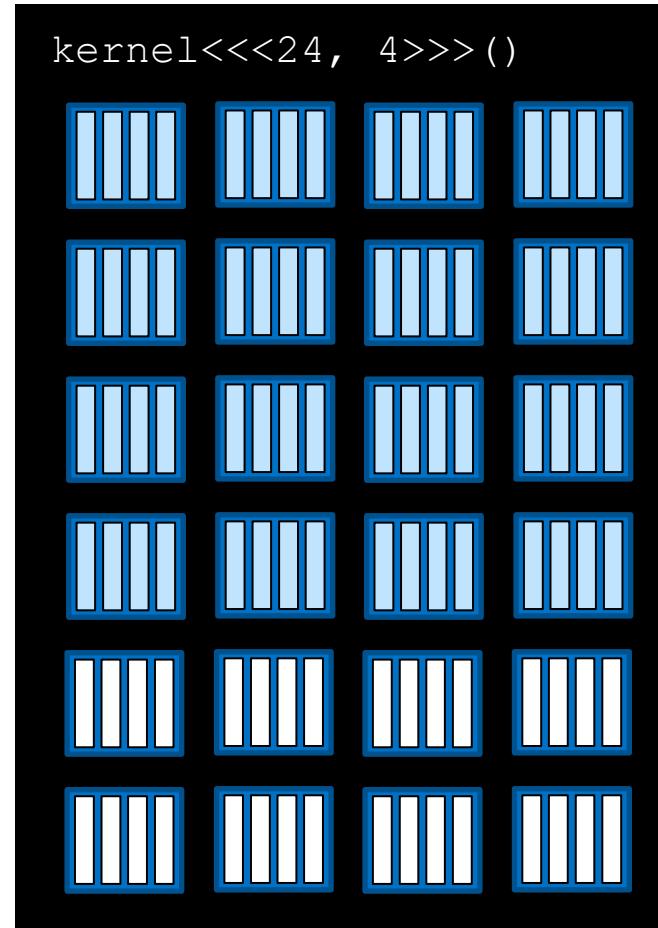
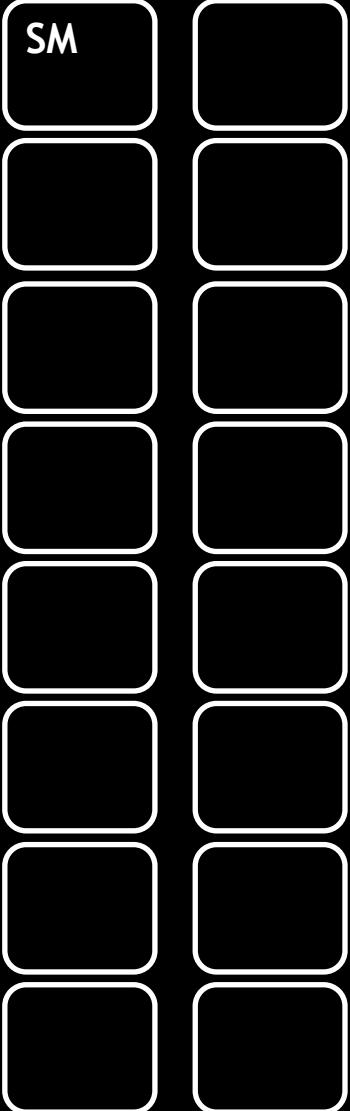
Blocks of threads are scheduled to run on SMs

# GPU



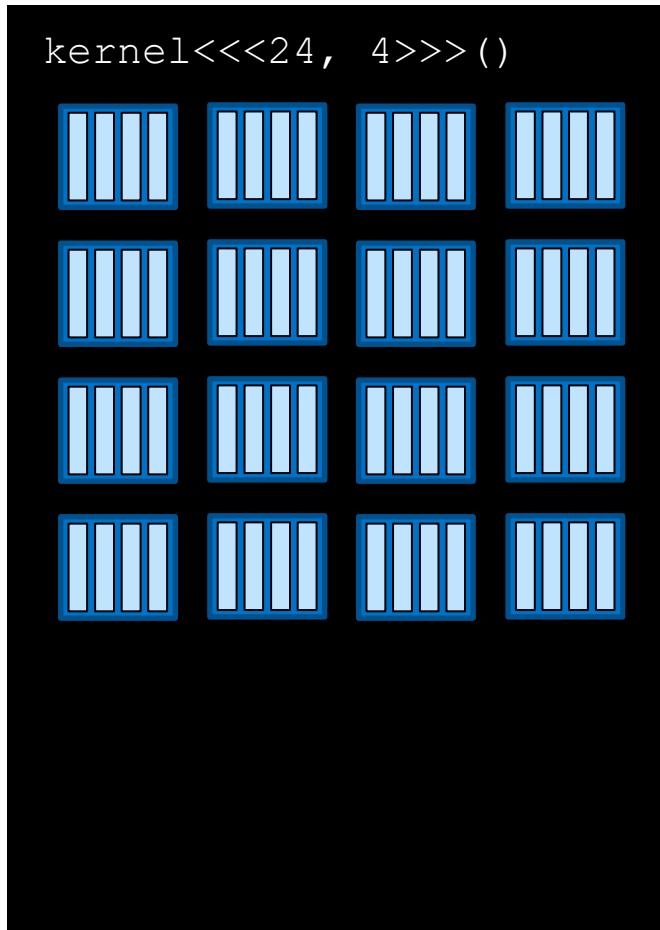
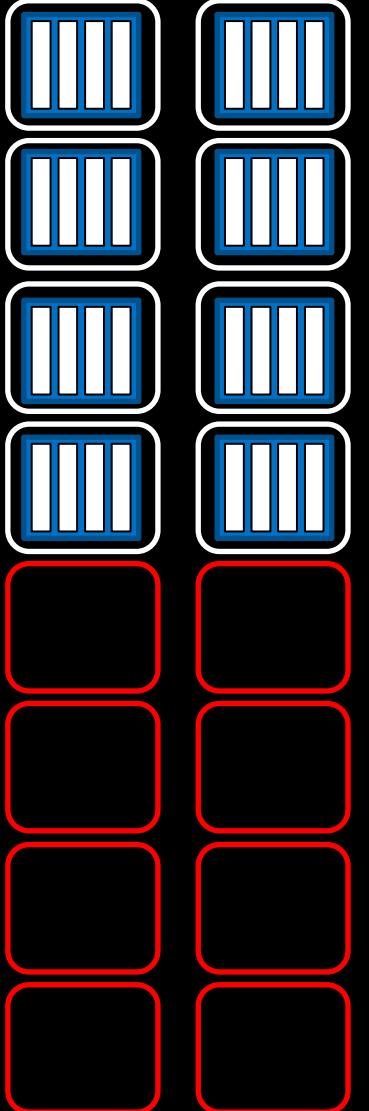
Depending on the number of SMs on a GPU, and the requirements of a block, more than one block can be scheduled on an SM

# GPU



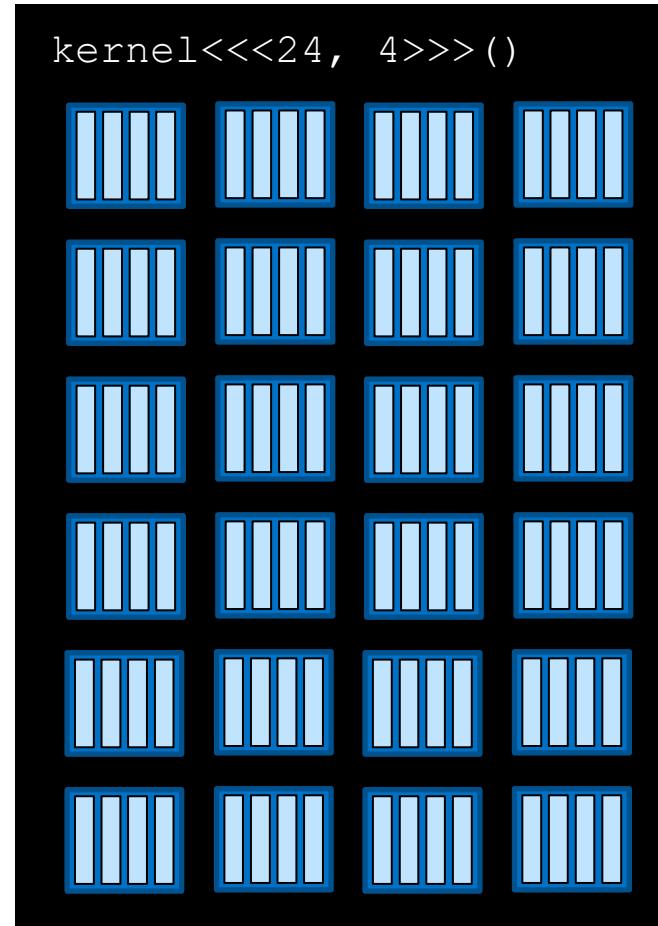
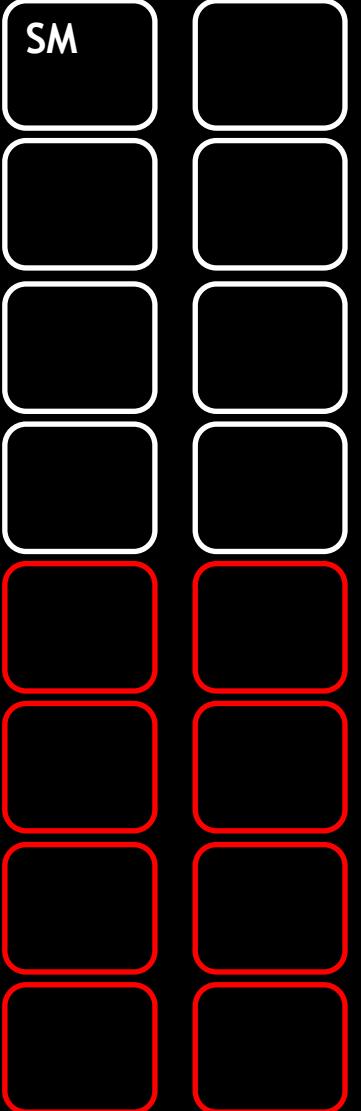
Depending on the number of SMs on a GPU, and the requirements of a block, more than one block can be scheduled on an SM

# GPU



Grid dimensions divisible by the number of SMs on a GPU can promote full SM utilization

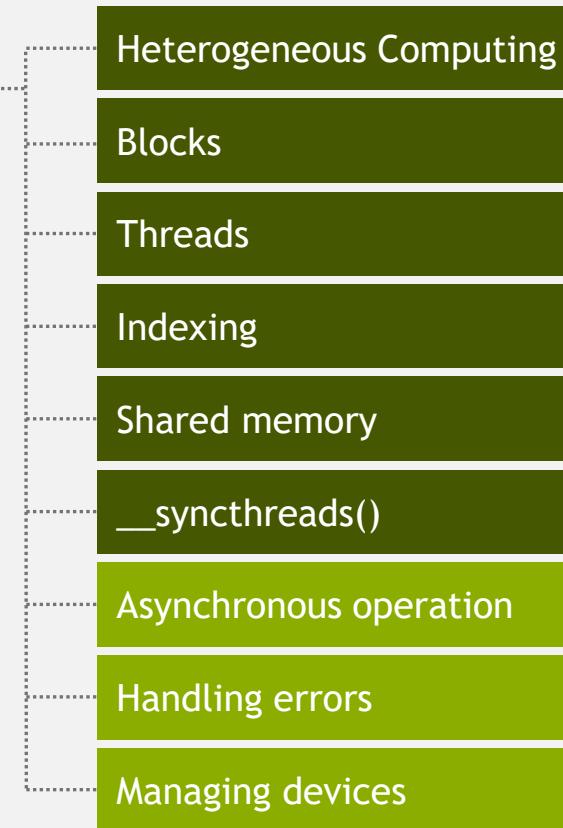
# GPU



Here there are fallow SMs

# MANAGING THE DEVICE

## CONCEPTS



# Coordinating Host & Device

- Kernel launches are **asynchronous**
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

**cudaMemcpy()** Blocks the CPU until the copy is complete

Copy begins when all preceding CUDA calls have completed

**cudaMemcpyAsync()** Asynchronous, does not block the CPU

**cudaDeviceSynchronize()** Blocks the CPU until all preceding CUDA calls have completed

# Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error:  
`cudaError_t cudaGetLastError(void)`
- Get a string to describe the error:  
`char *cudaGetString(cudaError_t)`  
  
`printf("%s\n", cudaGetString(cudaGetLastError()));`

# Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)  
cudaSetDevice(int device)  
cudaGetDevice(int *device)  
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices

```
cudaSetDevice(i) to select current device  
cudaMemcpy(...) for peer-to-peer copies†
```

<sup>†</sup> requires OS and device support



## CUDA Runtime API

API Reference Manual

vRelease Version | July 2019

<https://docs.nvidia.com/cuda/cuda-runtime-api/index.html>  
PDF copy included in docs folder

# Managing Accelerated Application Memory with CUDA C/C++ Unified Memory (Nvidia courseware)

## Streaming Multiprocessors and Warps

The GPUs that CUDA applications run on have processing units called **streaming multiprocessors**, or **SMs**. During kernel execution, blocks of threads are given to SMs to execute. In order to support the GPU's ability to perform as many parallel operations as possible, performance gains can often be had by *choosing a grid size that has a number of blocks that is a multiple of the number of SMs on a given GPU.*

Additionally, SMs create, manage, schedule, and execute groupings of 32 threads from within a block called **warps**. A more [in depth coverage of SMs and warps](#) is beyond the scope of this course, however, it is important to know that performance gains can also be had by *choosing a block size that has a number of threads that is a multiple of 32.*

## Programmatically Querying GPU Device Properties

In order to support portability, since the number of SMs on a GPU can differ depending on the specific GPU being used, the number of SMs should not be hard-coded into a codebase. Rather, this information should be acquired programmatically.

The following shows how, in CUDA C/C++, to obtain a C struct which contains many properties about the currently active GPU device, including its number of SMs:

```
int deviceId;
cudaGetDevice(&deviceId); // `deviceId` now points to the id of the currently
// active GPU.

cudaDeviceProp props;
cudaGetDeviceProperties(&props, deviceId); // `props` now has many useful properties about
// the active GPU device.
```

# Managing Accelerated Application Memory with CUDA

## C/C++ Unified Memory

### (Nvidia courseware)

```
int main()
{
    /*
     * Device ID is required first to query the device.
     */

    int deviceId;
    cudaGetDevice(&deviceId);

    cudaDeviceProp props;
    cudaGetDeviceProperties(&props, deviceId);

    /*
     * `props` now contains several properties about the current device.
     */

    int computeCapabilityMajor = props.major;
    int computeCapabilityMinor = props.minor;
    int multiProcessorCount = props.multiProcessorCount;
    int warpSize = props.warpSize;

    printf("Device ID: %d\nNumber of SMs: %d\nCompute Capability Major: %d\nCompute Capability Minor: %d\nWarp Size: %d\n", deviceId, multiProcessorCount, computeCapabilityMajor, computeCapabilityMinor, warpSize);
}
```

# Managing Accelerated Application Memory with CUDA

## C/C++ Unified Memory

### (Nvidia courseware)

```
In [1]: !nvcc -o get-device-properties 04-device-properties/01-get-device-properties.cu -run
```

```
Device ID: 0
Number of SMs: 40
Compute Capability Major: 7
Compute Capability Minor: 5
Warp Size: 32
```

## Re-visiting vector-add

```
In [2]: !nvcc -o sm-optimized-vector-add 01-vector-add/01-vector-add.cu -run
```

Success! All values calculated correctly.

```
In [ ]: !nsys profile --stats=true ./sm-optimized-vector-add
```

# Nsight on Windows

## Note:

- For Pascal hardware, install Nsight 2019.5 or older
- Nsight 2020/2021 has dropped Pascal support.

Name	Date modified	Type	Size
nvtx	7/24/2021 1:46 PM	File folder	
python	7/24/2021 1:46 PM	File folder	
reports	7/24/2021 1:46 PM	File folder	
rules	7/24/2021 1:46 PM	File folder	
vulkan-layers	7/24/2021 1:46 PM	File folder	
bifrost_loader.dll	5/25/2021 12:46 AM	Application extension	282 KB
bifrost_plugin.dll	5/25/2021 12:46 AM	Application extension	119 KB
config.ini	5/25/2021 12:46 AM	Configuration settings	0 KB
cupti64_100.dll	5/25/2021 12:46 AM	Application extension	3,571 KB
cupti64_101.dll	5/25/2021 12:46 AM	Application extension	3,578 KB
cupti64_102.dll	5/25/2021 12:46 AM	Application extension	3,447 KB
cupti64_110.dll	5/25/2021 12:46 AM	Application extension	3,652 KB
cupti64_111.dll	5/25/2021 12:46 AM	Application extension	3,659 KB
cupti64_112.dll	5/25/2021 12:46 AM	Application extension	3,694 KB
cupti64_113.dll	5/25/2021 12:46 AM	Application extension	3,758 KB
cupti64_114.dll	5/25/2021 12:46 AM	Application extension	3,970 KB
etw_providers_template.json	5/25/2021 12:55 AM	JSON Source File	3 KB
GpuMetrics.config	5/25/2021 12:55 AM	Configuration Source File	56 KB
nsight-sys-service.exe	5/25/2021 12:53 AM	Application	129 KB
nsys.exe	5/25/2021 12:55 AM	Application	15,098 KB
NsysVslIntegration.xml	5/25/2021 12:47 AM	XML Document	2 KB
nvperf_host.dll	5/25/2021 12:49 AM	Application extension	13,298 KB
sqlite3.exe	5/25/2021 12:46 AM	Application	1,064 KB
targetsettings.xml	5/25/2021 12:46 AM	XML Document	4 KB
ToolsInjection64.dll	5/25/2021 12:52 AM	Application extension	11,433 KB

# Profiling vector-add

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
90.5	2339293277	1	2339293277.0	2339293277	2339293277	cudaDeviceSynchronize
8.8	226837438	3	75612479.3	18059	226773774	cudaMallocManaged
0.7	18227575	3	6075858.3	5479985	7131312	cudaFree
0.0	53218	1	53218.0	53218	53218	cudaLaunchKernel

CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	2339281682	1	2339281682.0	2339281682	2339281682	addVectorsInto(float*, float*, float*, int)

CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
76.6	68387966	2304	29682.3	1855	181951	[CUDA Unified Memory memcpy HtoD]
23.4	20840704	768	27136.3	1119	159807	[CUDA Unified Memory memcpy DtoH]

CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
393216.000	2304	170.667	4.000	1020.000	[CUDA Unified Memory memcpy HtoD]
131072.000	768	170.667	4.000	1020.000	[CUDA Unified Memory memcpy DtoH]

## Optimizing vector-add: Use SM count in kernel configuration

```
size_t numberOfBlocks = ceil(ceil(N/1024.0)/40.0)*40;
size_t threadsPerBlock = ceil(N/float(numberOfBlocks));

cudaError_t addVectorsErr;
cudaError_t asyncErr;

addVectorsInto<<<numberOfBlocks, threadsPerBlock>>>(c, a, b, N);
```

### CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	133485799	1	133485799.0	133485799	133485799	addVectorsInto(float*, float*, float*, int)

## Unified Memory and Page-faulting: CPU Only

```
> toshiba-cuda-2021 > docs > nvidia-courseware > page faults > 01-page-faults-solution-cpu-only.cu
__global__
void deviceKernel(int *a, int N)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = idx; i < N; i += stride)
    {
        a[i] = 1;
    }
}

void hostFunction(int *a, int N)
{
    for (int i = 0; i < N; ++i)
    {
        a[i] = 1;
    }
}

int main()
{
    int N = 2<<24;
    size_t size = N * sizeof(int);
    int *a;
    cudaMallocManaged(&a, size);
    hostFunction(a, N);
    cudaFree(a);
}
```

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
98.3	229814053	1	229814053.0	229814053	229814053	cudaMallocManaged
1.7	3914137	1	3914137.0	3914137	3914137	cudaFree

# Unified Memory and Page-faulting: GPU Only

CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
91.1	225642941	1	225642941.0	225642941	225642941	cudaMallocManaged
6.4	15883701	1	15883701.0	15883701	15883701	cudaDeviceSynchronize
2.5	6087696	1	6087696.0	6087696	6087696	cudaFree
0.0	34077	1	34077.0	34077	34077	cudaLaunchKernel

CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	15878955	1	15878955.0	15878955	15878955	deviceKernel(int*, int)

solution-gpu-only.cu

```
int main()
{
    int N = 2<<24;
    size_t size = N * sizeof(int);
    int *a;
    cudaMallocManaged(&a, size);

    deviceKernel<<<256, 256>>>(a, N);
    cudaDeviceSynchronize();

    cudaFree(a);
}
```

# Unified Memory and Page-faulting: CPU then GPU

## CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
77.5	231322845	1	231322845.0	231322845	231322845	cudaMallocManaged
20.3	60595299	1	60595299.0	60595299	60595299	cudaDeviceSynchronize
2.1	6345710	1	6345710.0	6345710	6345710	cudaFree
0.0	57660	1	57660.0	57660	57660	cudaLaunchKernel

## CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	60583520	1	60583520.0	60583520	60583520	deviceKernel(int*, int)

## CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
100.0	29952964	4812	6224.6	2111	158112	[CUDA Unified Memory memcpy HtoD]

## CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
131072.000	4812	27.239	4.000	948.000	[CUDA Unified Memory memcpy HtoD]

```
int main()
{
    int N = 2<<24;
    size_t size = N * sizeof(int);
    int *a;
    cudaMallocManaged(&a, size);

    hostFunction(a, N);
    deviceKernel<<<256, 256>>>(a, N);
    cudaDeviceSynchronize();

    cudaFree(a);
}
```

# Unified Memory and Page-faulting: GPU then CPU

## CUDA API Statistics:

Time(%)	Total Time (ns)	Num Calls	Average	Minimum	Maximum	Name
88.9	227260385	1	227260385.0	227260385	227260385	cudaMallocManaged
7.9	20221120	1	20221120.0	20221120	20221120	cudaDeviceSynchronize
3.2	8079876	1	8079876.0	8079876	8079876	cudaFree
0.0	46971	1	46971.0	46971	46971	cudaLaunchKernel

## CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	20207169	1	20207169.0	20207169	20207169	deviceKernel(int*, int)

## CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
100.0	21122068	768	27502.7	1631	159743	[CUDA Unified Memory memcpy DtoH]

## CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
131072.000	768	170.667	4.000	1020.000	[CUDA Unified Memory memcpy DtoH]

```
int main()
{
    int N = 2<<24;
    size_t size = N * sizeof(int);
    int *a;
    cudaMallocManaged(&a, size);
    deviceKernel<<<256, 256>>>(a, N);
    cudaDeviceSynchronize();
    hostFunction(a, N);
    cudaFree(a);
}
```

# Unified Memory and Page-faulting: Initialize Vector in Kernel

```
t > toshiba-cuda-2021 > docs > nvidia-courseware > page faults > 01-vector-add-init-in-kernel-solution.cu
```

```
* Refactor host function to run as CUDA kernel
*/
```

```
__global__
void initwith(float num, float *a, int N)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for(int i = index; i < N; i += stride)
    {
        | a[i] = num;
    }
}
```

```
__global__
void addArraysInto(float *result, float *a, float *b, int N)
{
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;

    for(int i = index; i < N; i += stride)
    {
        | result[i] = a[i] + b[i];
    }
}
```

## CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
95.7	38433915	3	12811305.0	12502560	13266204	initWith(float, float*, int)
4.3	1706359	1	1706359.0	1706359	1706359	addArraysInto(float*, float*, float*, int)

## CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
100.0	21119182	768	27498.9	1630	159839	[CUDA Unified Memory memcpy DtoH]

## CUDA Memory Operation Statistics (by size in KiB):

Total	Operations	Average	Minimum	Maximum	Operation
131072.000	768	170.667	4.000	1020.000	[CUDA Unified Memory memcpy DtoH]

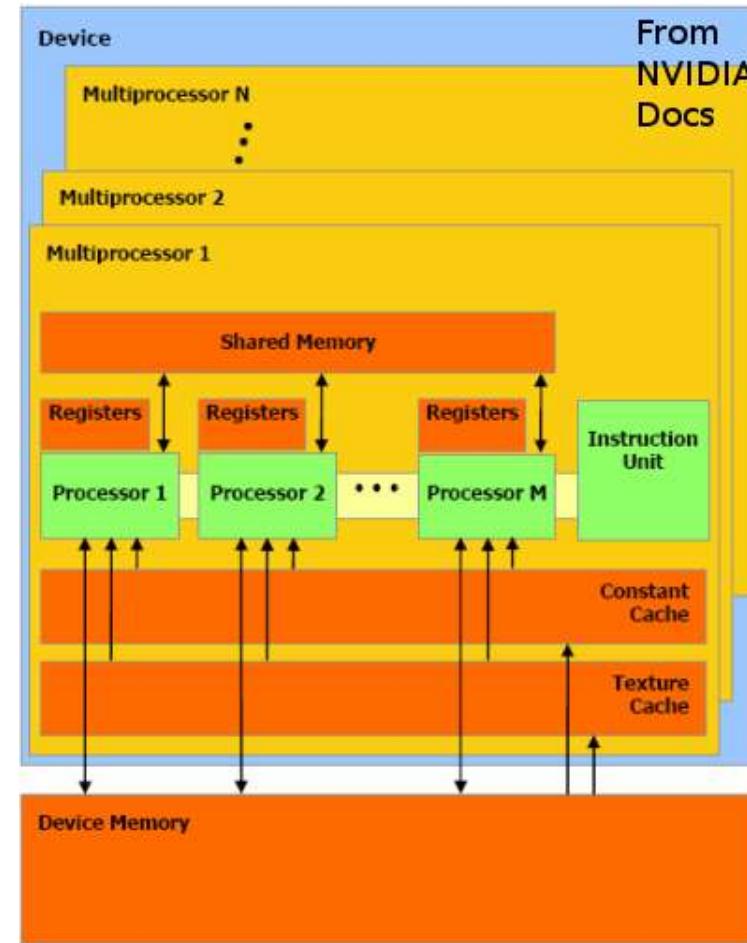
s PC > Storage (E:) > git > toshiba-cuda-2021 > docs > nvidia-courseware >



Name	Date modified	Type	Size
page faults	8/6/2021 1:19 PM	File folder	
01-get-device-properties-solution.cu	8/6/2021 12:31 PM	CU File	1 KB
01-heat-conduction-solution.cu	8/4/2021 8:06 PM	CU File	4 KB
01-vector-add-block-count-sm-multiple.cu	8/5/2021 2:22 PM	CU File	2 KB
AC_CUDA_C-whole.pptx	8/4/2021 12:22 AM	Microsoft PowerPo...	2,359 KB
NVPROF_UM_all.pptx	8/4/2021 8:08 PM	Microsoft PowerPo...	1,842 KB
NVVP-Streams-all.pptx	8/4/2021 11:13 PM	Microsoft PowerPo...	1,893 KB

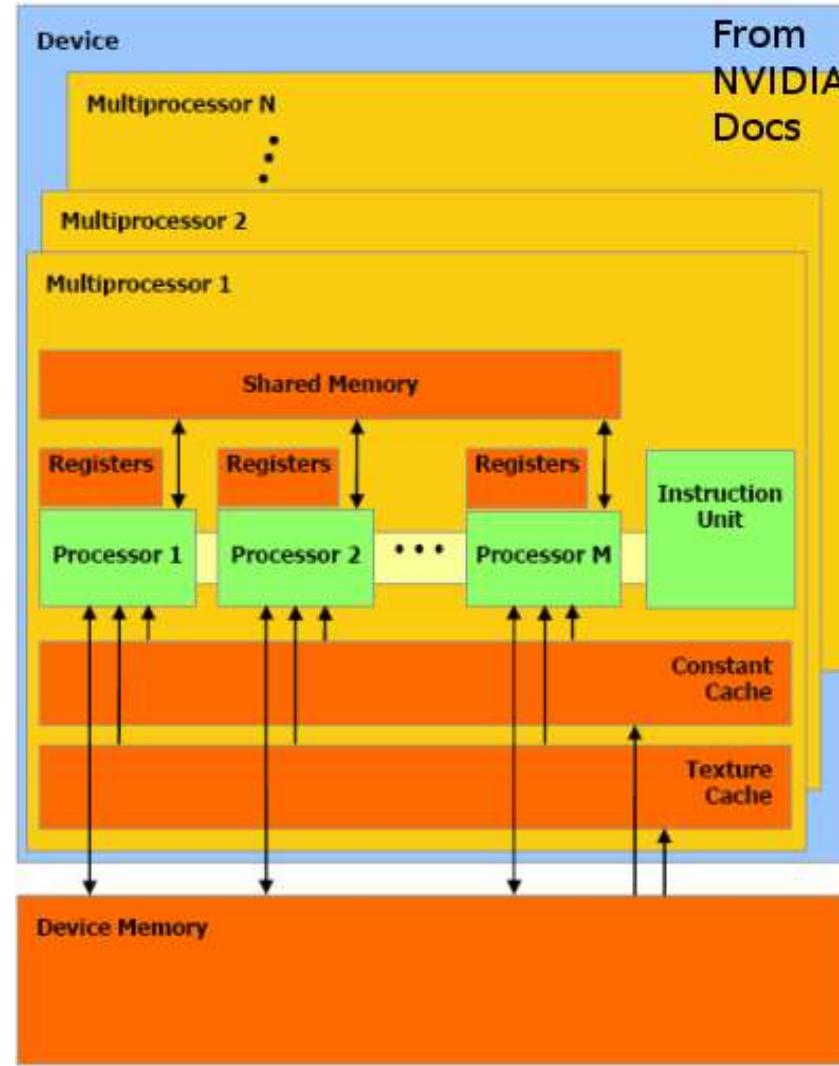
# Multiprocessor Occupancy

- Higher occupancy is often a goal in GPU optimization
- Greater occupancy can hide instruction latency
  - Read-after-write register latency
  - Latency in reading data from global memory
  - While threads in one warp waiting for data, another warp can run on multiprocessor



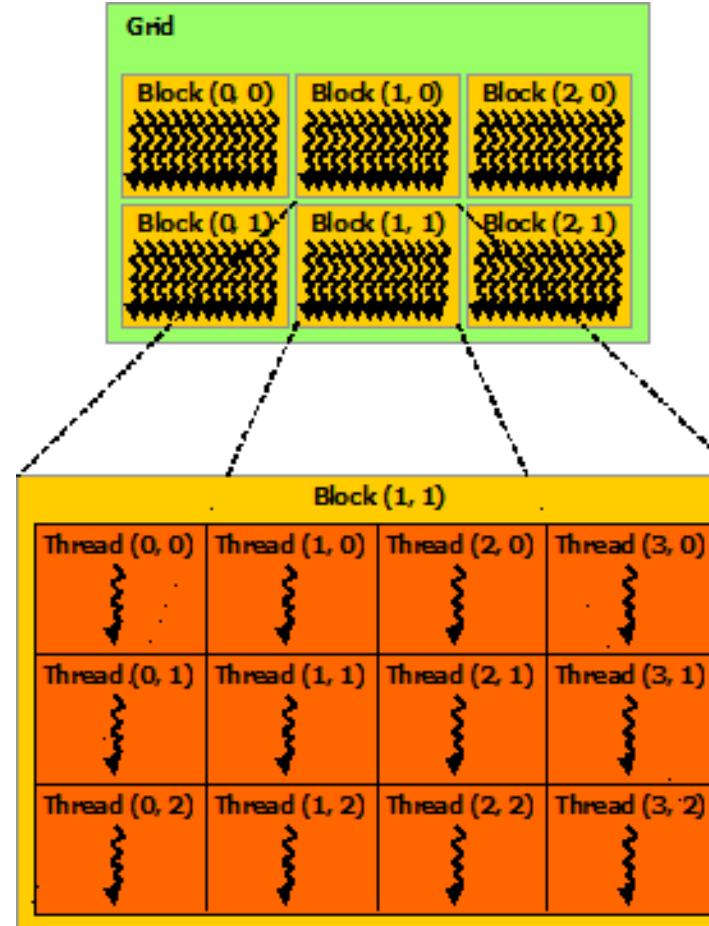
# Multiprocessor Occupancy

- However, maximizing occupancy does not always result in best performance
- Increased multiprocessor occupancy can be at expense of faster register/shared memory accesses



# Multiprocessor Occupancy Factors

- Thread block dimensions
- Register usage per thread
- Shared memory usage per thread block
- Target GPU architecture



# CUDA Occupancy Calculator

- Provided by NVIDIA to compute occupancy of CUDA kernel
- User enters GPU compute capability and resource usage
- Occupancy calculator computes occupancy
- Shows impact of...
  - Adjusting thread block size
  - Adjusting register usage
  - Adjusting shared memory usage
- Can be used as a tool to tweak CUDA program to improve multiprocessor occupancy

# CUDA Occupancy Calculator

Available at [http://developer.download.nvidia.com/compute/cuda/CUDA\\_Occupancy\\_calculator.xls](http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls)

## CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): 1.3 [\(Help\)](#)

2.) Enter your resource usage:

Threads Per Block 256 [\(Help\)](#)

Registers Per Thread 16 [\(Help\)](#)

Shared Memory Per Block (bytes) 4096 [\(Help\)](#)

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:

Active Threads per Multiprocessor 1024 [\(Help\)](#)

Active Warps per Multiprocessor 32 [\(Help\)](#)

Active Thread Blocks per Multiprocessor 4 [\(Help\)](#)

Occupancy of each Multiprocessor 100% [\(Help\)](#)

Physical Limits for GPU Compute Capability: 1.3

Threads per Warp 32

Warps per Multiprocessor 32

Threads per Multiprocessor 1024

Thread Blocks per Multiprocessor 8

Total # of 32-bit registers per Multiprocessor 16384

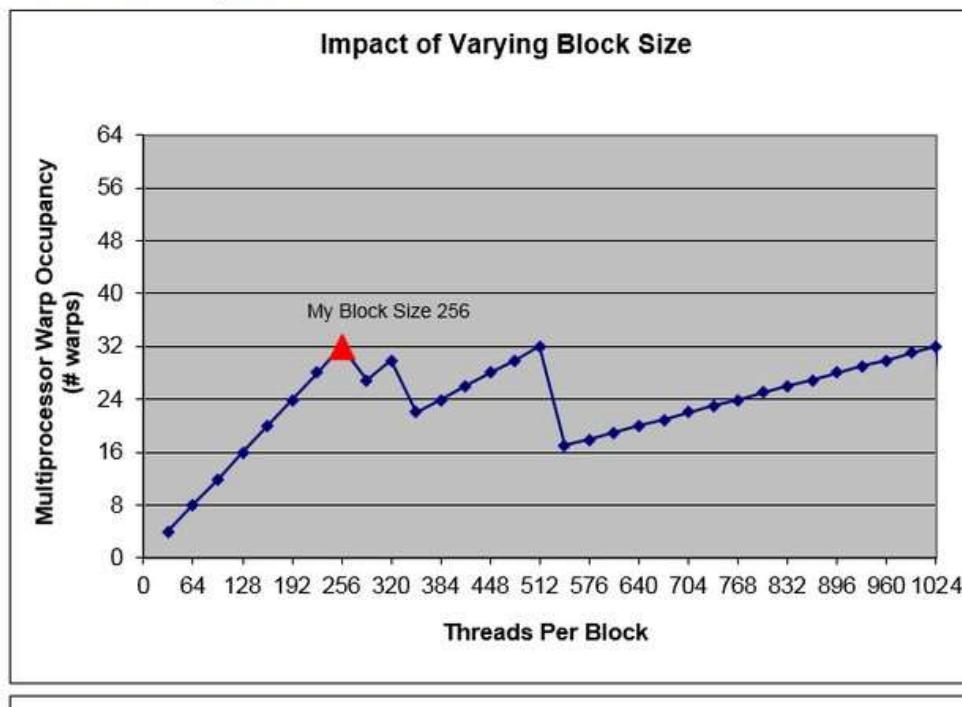
# CUDA Occupancy Calculator

Available at [http://developer.download.nvidia.com/compute/cuda/CUDA\\_Occupancy\\_calculator.xls](http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls)

[Click Here for detailed instructions on how to use this occupancy calculator.](#)

[For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda](#)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



# CUDA Occupancy Considerations

- All things equal, same program with higher occupancy should run faster
- However, may be necessary to sacrifice register / shared memory usage to increase occupancy
  - May increase necessary memory transfers from global memory
  - May slow program more than reduced occupancy

# Other Optimization Considerations

- Thread block dimensions
- Global memory load/store pattern
- Register usage
- Local memory usage
- Branches within kernel
- Shared memory
- Constant memory
- Texture memory

# Thread Block Dimensions

- CUDA threads grouped together in thread block structure
  - Run on same multiprocessor
  - Have access to common pool of fast shared memory
  - Can synchronize between threads in same thread block
- On Pascal, maximum of 64 concurrent warps, and 32 active thread blocks / SM
- Max thread block size on Pascal is 1024

# Optimizing Thread Block Dimensions

- Use multiple of warp size (32)
  - Otherwise will be partially full warp --> wasted resources
- Different thread block dimensions work best for different programs
- Experiment and see what works best
  - Common thread block sizes: 128, 192, 256, 384, 512
  - If 2D thread block, common dimensions are 32X4, 32X6, 32X8, 32X12, 32X18



## Tuning CUDA Applications for Pascal

Application Note

DA-08134-001\_v11.4 | August 2021

<https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html>  
PDF copy in docs folder



## Tuning CUDA Applications for Volta

Application Note

DA-08647-001\_v11.4 | August 2021

<https://docs.nvidia.com/cuda/volta-tuning-guide/index.html>  
PDF copy in docs folder



## Tuning CUDA Applications for Turing

Application Note

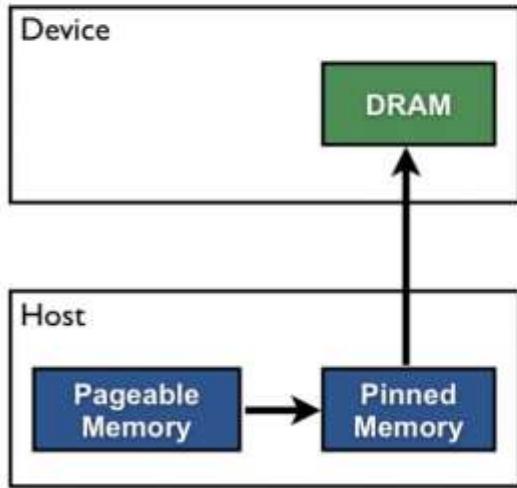
DA-09073-001\_v11.4 | August 2021

<https://docs.nvidia.com/cuda/turing-tuning-guide/index.html>  
PDF copy in docs folder

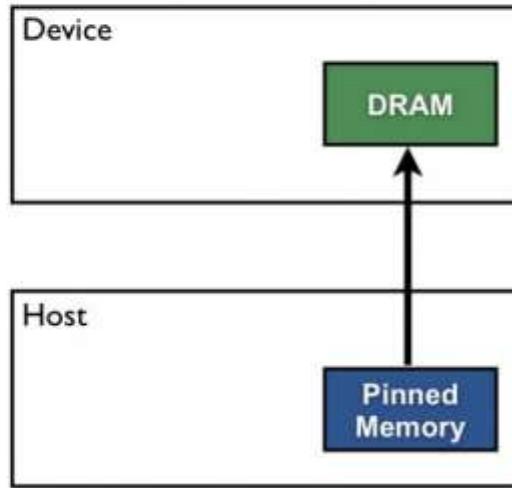
# CUDA Samples / simpleOccupancy

# Pinned Memory

*Pageable Data Transfer*



*Pinned Data Transfer*



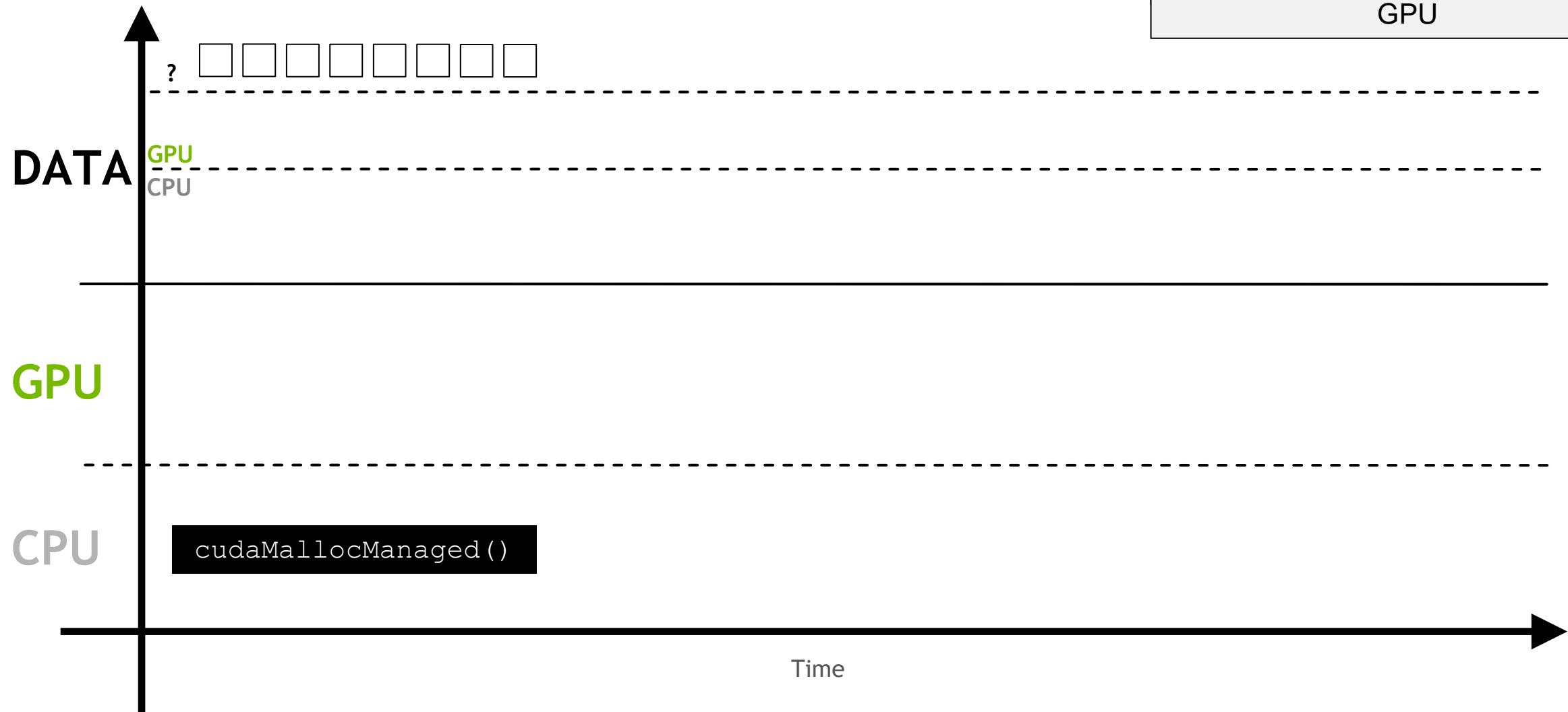
- Using pinned memory speeds up `cudaMemcpy()` calls.
- Pinned memory is a limited resource.
- Pinning memory “removes” it from the host paged virtual memory system.
- Less useful since Over-subscription introduced for Unified Memory in CUDA 8.

# **samples / zeroCopy**

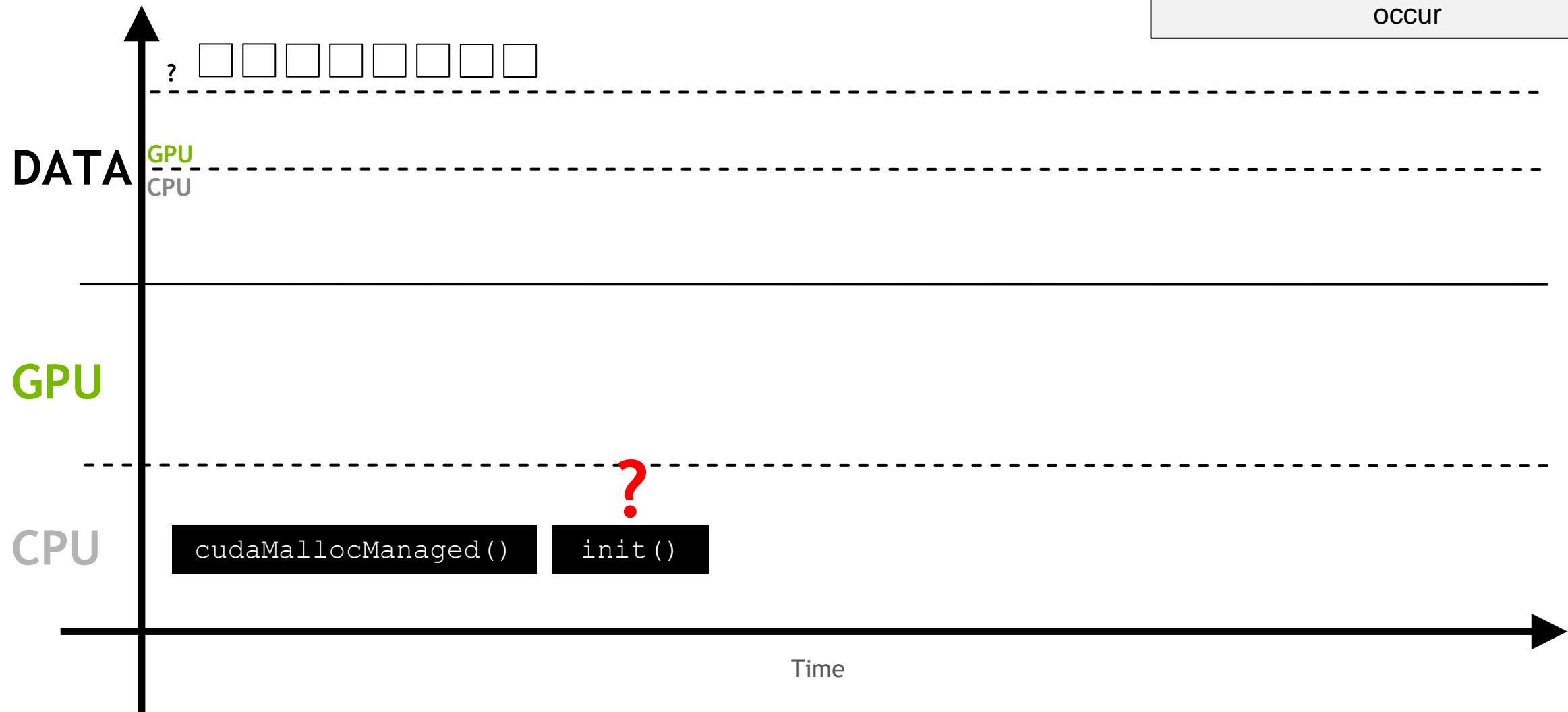
- **cudaHostAlloc** – Allocates page-locked memory on host.
- **cudaHostRegister** – Registers (page-locks) an existing host memory range for use by CUDA.
- Just use Unified Memory with **cudaMallocManaged()** and async prefetch/copy (to be discussed) unless there is a clear case for explicit memory management.

# Unified Memory Behavior

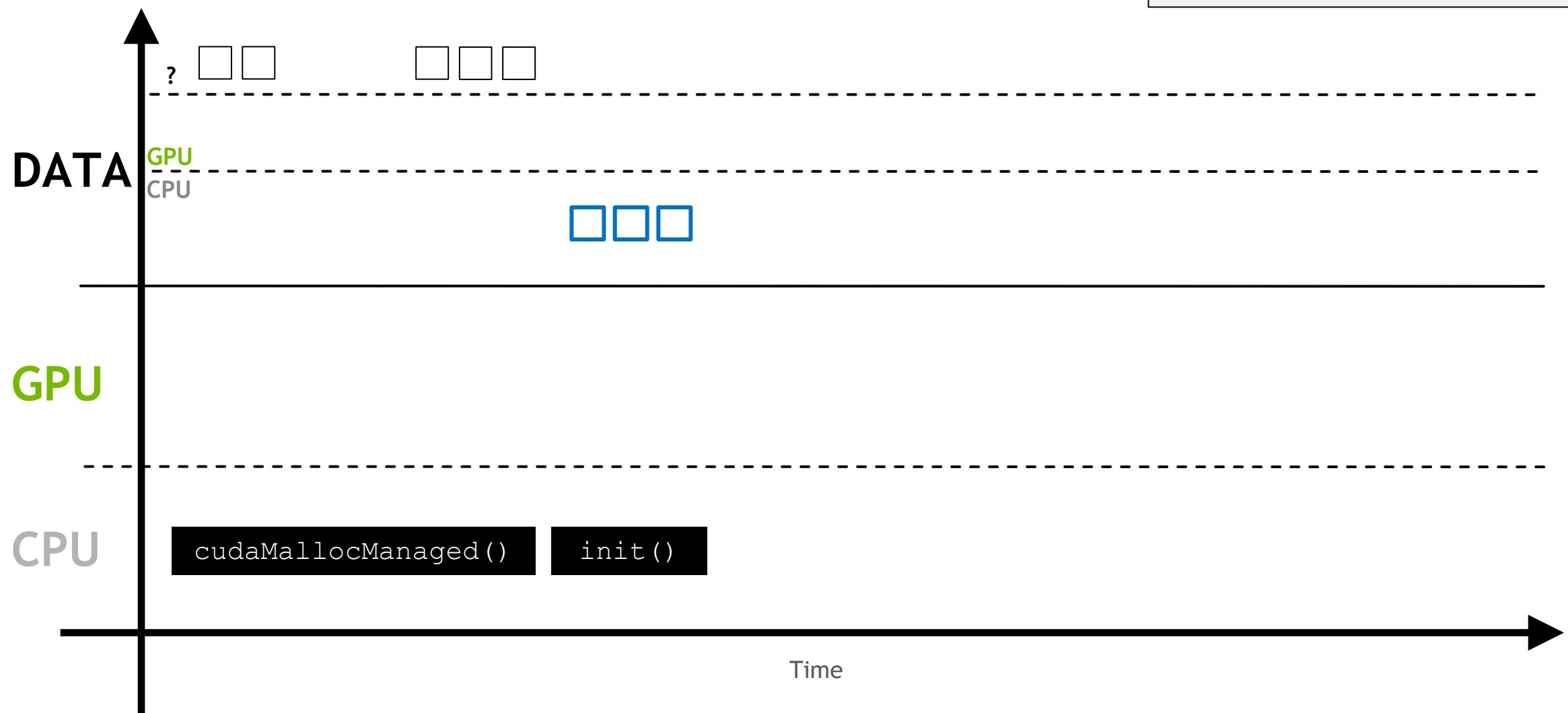
When **UM** is allocated, it may not be resident initially on the CPU or the GPU



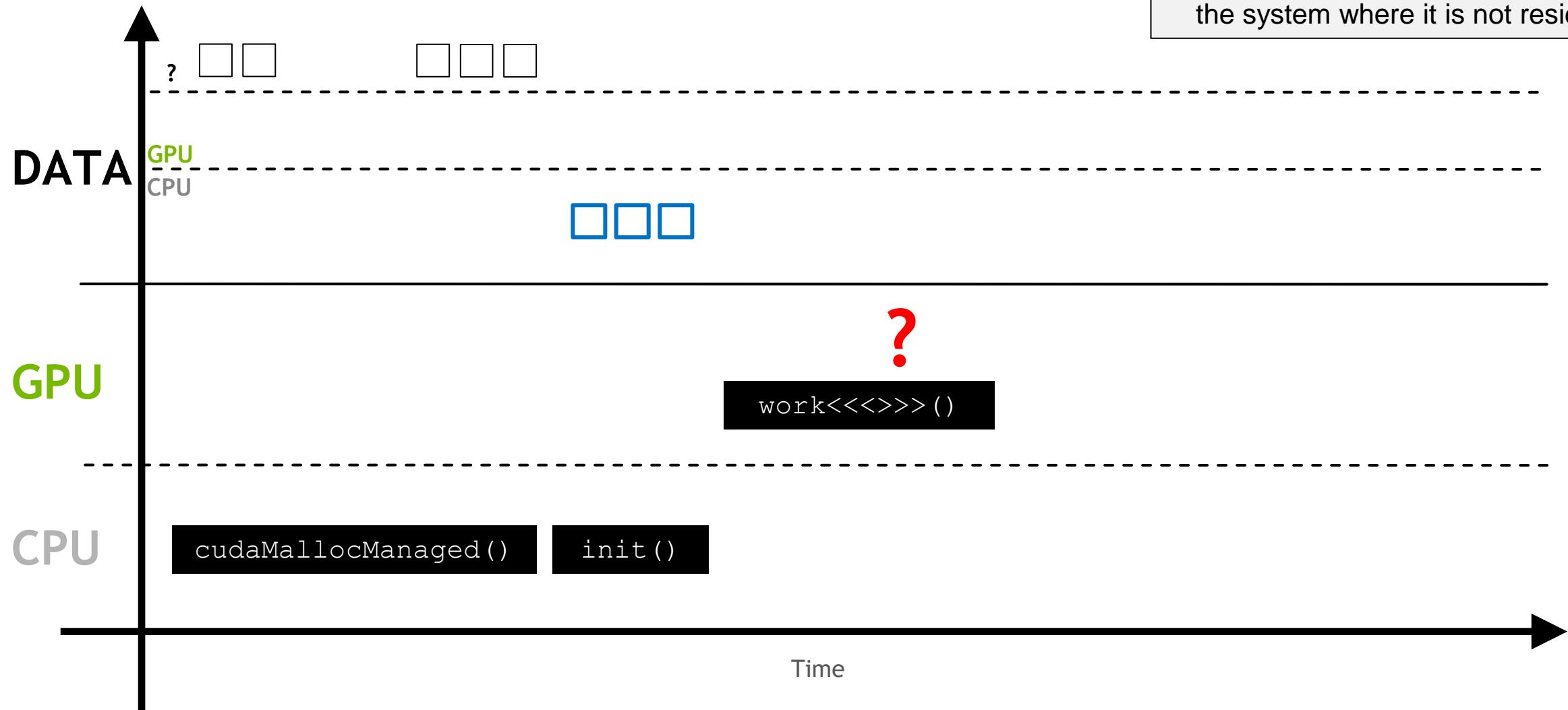
When some work asks for the memory  
for the first time, a **page fault** will  
occur



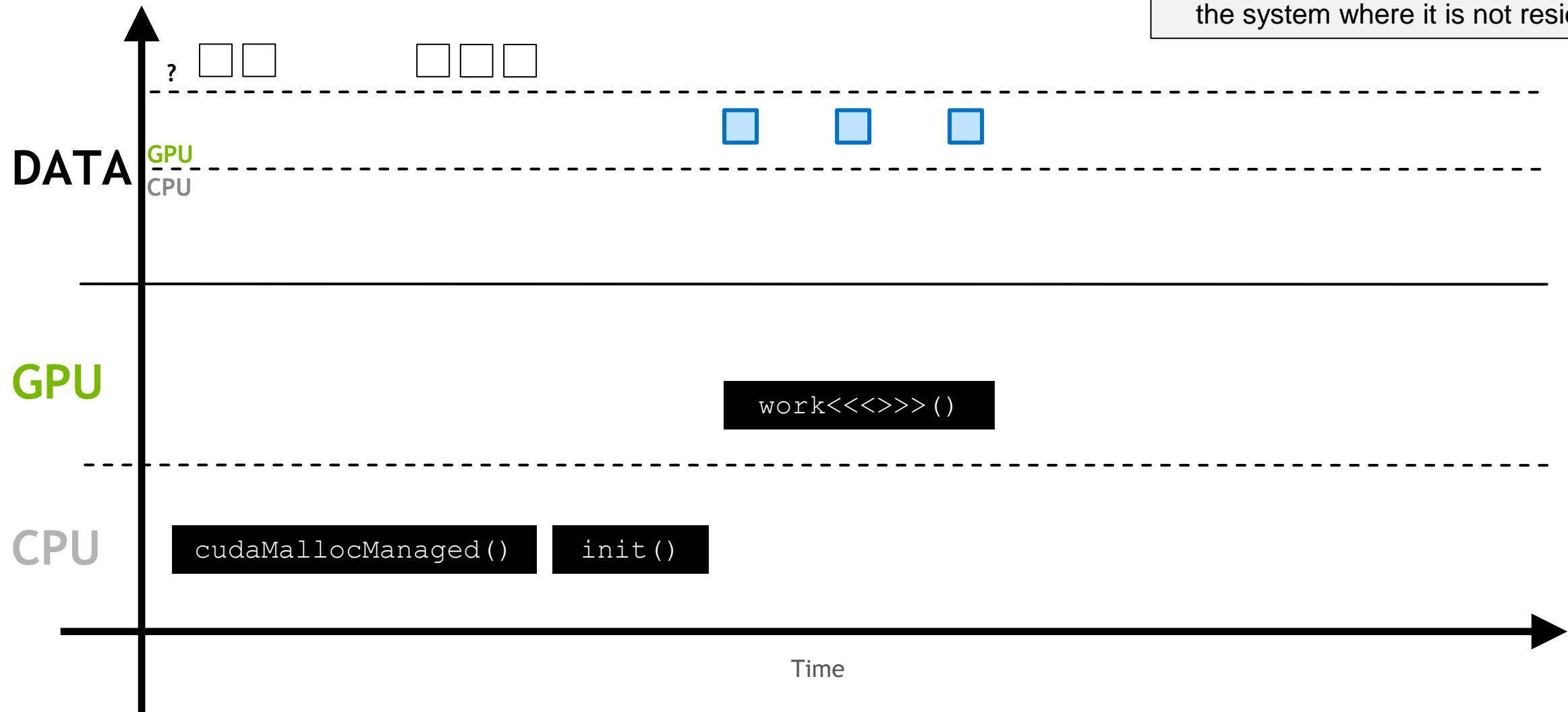
The page fault will trigger the migration  
of the demanded memory



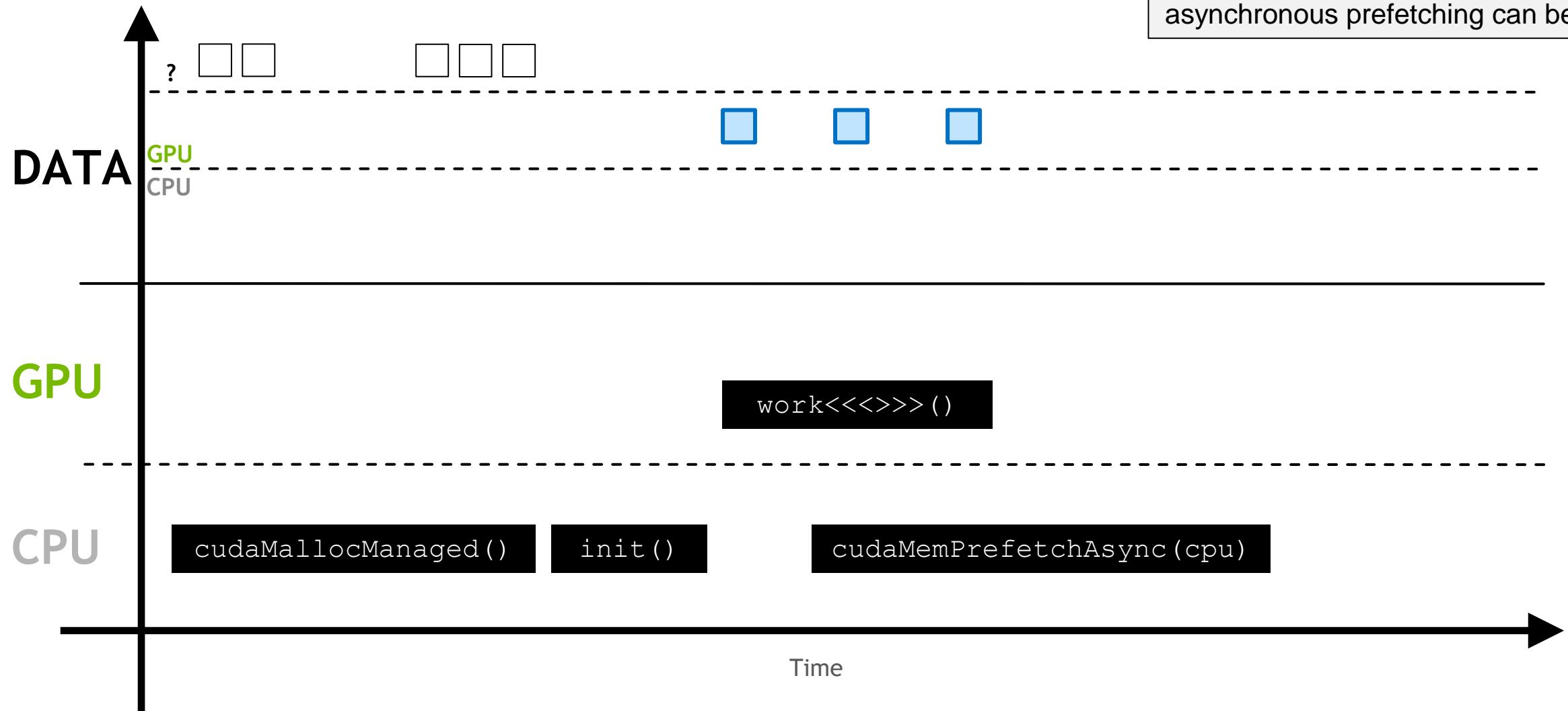
This process repeats anytime the memory is requested somewhere in the system where it is not resident



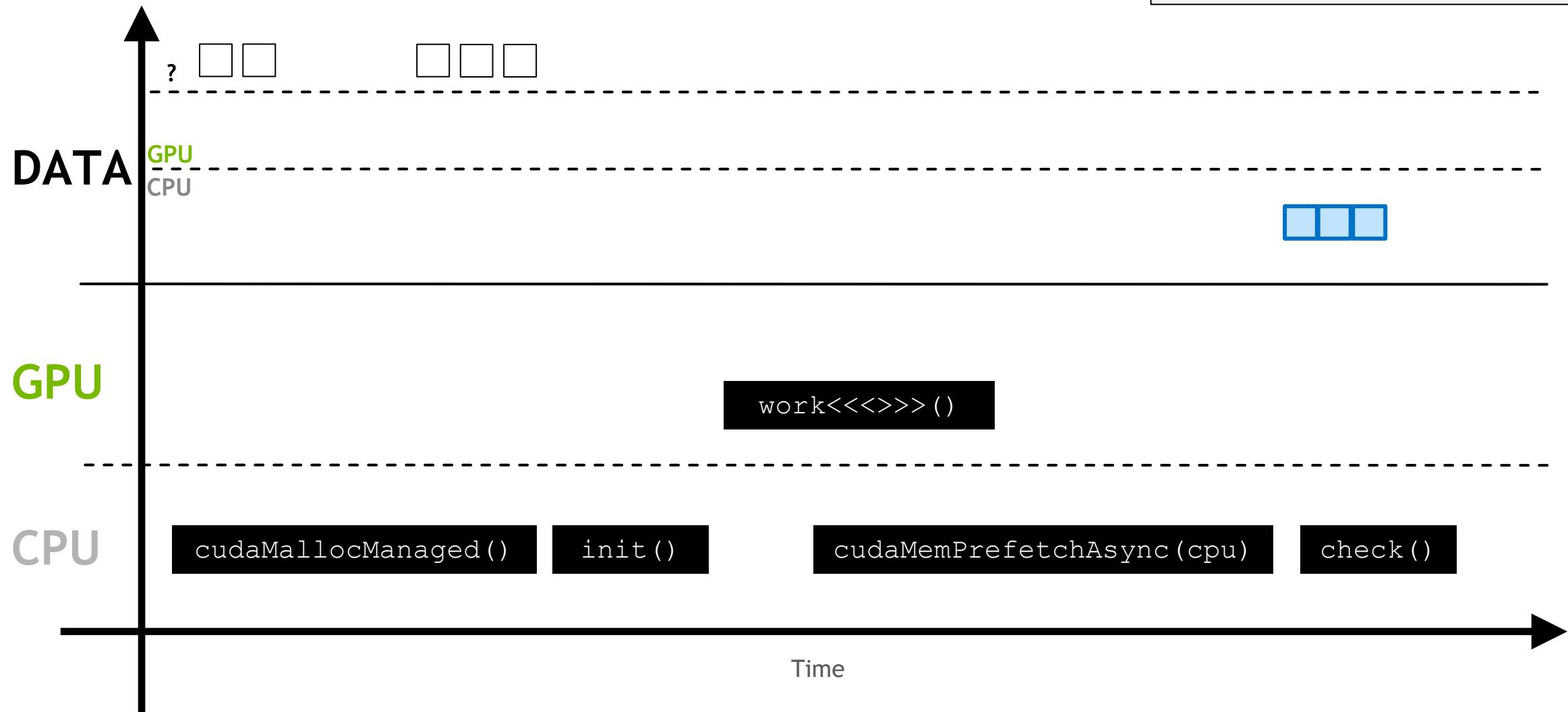
This process repeats anytime the memory is requested somewhere in the system where it is not resident

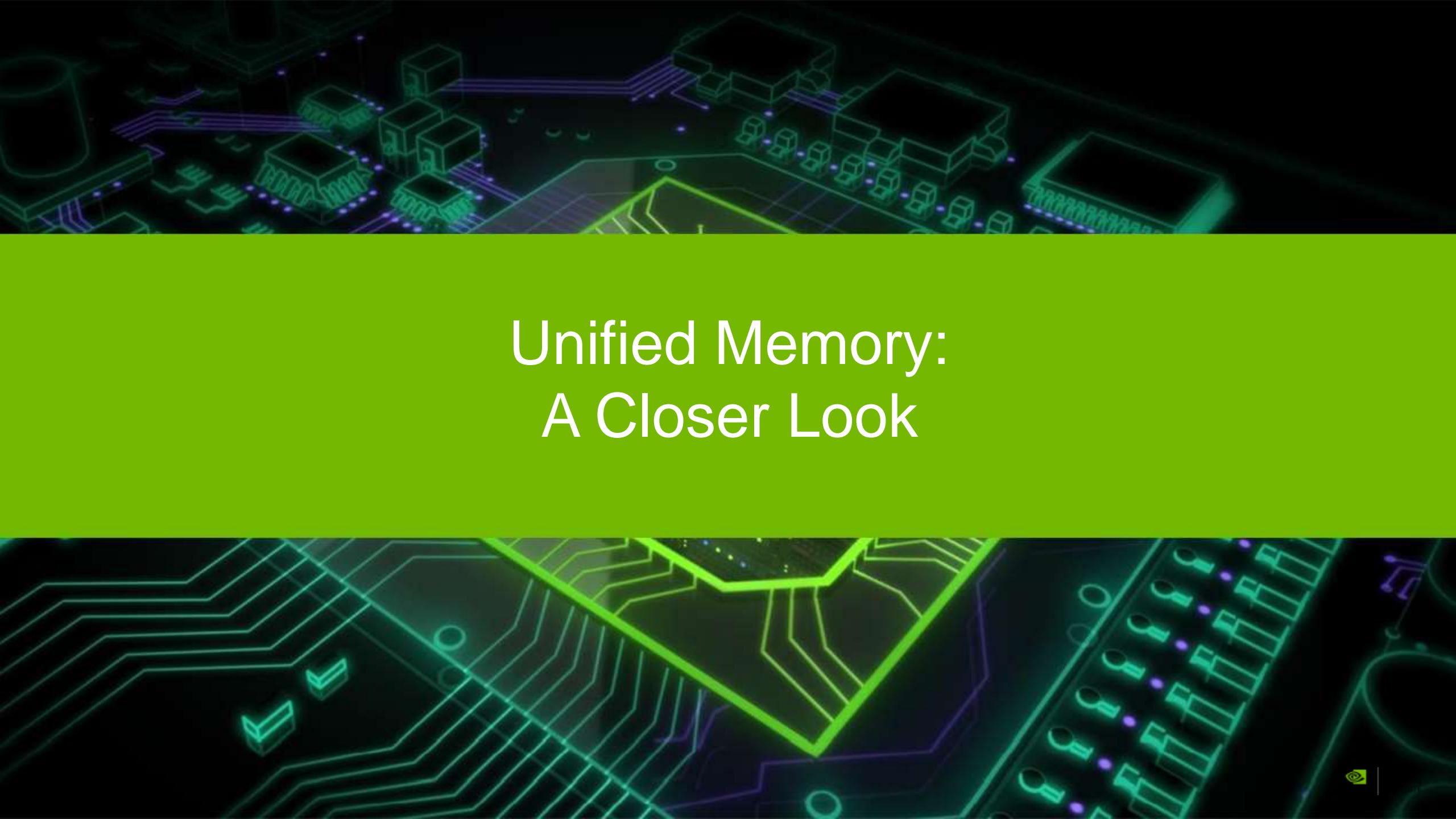


If it is known that the memory **will be** accessed somewhere it is not resident, asynchronous prefetching can be used



This moves the memory in larger batches, and prevents page faulting





# Unified Memory: A Closer Look

# SINGLE POINTER

## CPU vs GPU

CPU code

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
cpu_func2(data, N);  
  
cpu_func3(data, N);  
  
free(data);
```

GPU code w/ Unified Memory

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
gpu_func2<<<...>>>(data, N);  
cudaDeviceSynchronize();  
  
cpu_func3(data, N);  
  
free(data);
```

# SINGLE POINTER

## Explicit vs Unified Memory

### Explicit Memory Management

```
void *data, *d_data;  
data = malloc(N);  
cudaMalloc(&d_data, N);  
cpu_func1(data, N);  
cudaMemcpy(d_data, data, N, ...)  
gpu_func2<<<...>>>(d_data, N);  
cudaMemcpy(data, d_data, N, ...)  
cudaFree(d_data);  
cpu_func3(data, N);  
  
free(data);
```

### GPU code w/ Unified Memory

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
gpu_func2<<<...>>>(data, N);  
cudaDeviceSynchronize();  
  
cpu_func3(data, N);  
  
free(data);
```

# SINGLE POINTER

## Full Control with Prefetching

### Explicit Memory Management

```
void *data, *d_data;  
data = malloc(N);  
cudaMalloc(&d_data, N);  
cpu_func1(data, N);  
cudaMemcpy(d_data, data, N, ...)  
gpu_func2<<<...>>>(d_data, N);  
cudaMemcpy(data, d_data, N, ...)  
cudaFree(d_data);  
cpu_func3(data, N);  
  
free(data);
```

### Unified Memory + Prefetching

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
cudaMemPrefetchAsync(data, N, GPU)  
gpu_func2<<<...>>>(data, N);  
cudaMemPrefetchAsync(data, N, CPU)  
cudaDeviceSynchronize();  
cpu_func3(data, N);  
  
free(data);
```

# SINGLE POINTER

## Deep Copy

### Explicit Memory Management

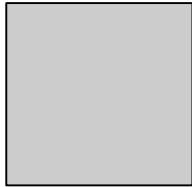
```
char **data;  
// allocate and initialize data on the CPU  
  
char **d_data;  
char **h_data = (char**)malloc(N*sizeof(char*));  
for (int i = 0; i < N; i++) {  
    cudaMalloc(&h_data[i], N);  
    cudaMemcpy(h_data[i], data[i], N, ...);  
}  
cudaMalloc(&d_data, N*sizeof(char*));  
cudaMemcpy(d_data, h_data, N*sizeof(char*), ...);  
  
gpu_func<<<...>>>(d_data, N);
```

### GPU code w/ Unified Memory

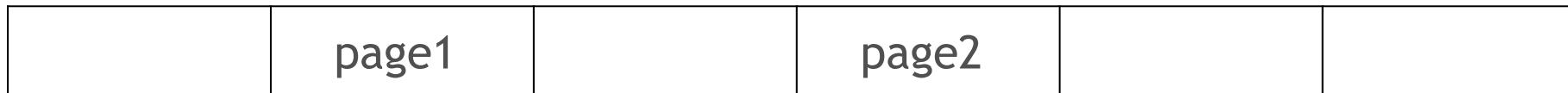
```
char **data;  
// allocate and initialize data on the CPU  
  
gpu_func<<<...>>>(data, N);
```

# UNIFIED MEMORY BASICS

GPU A

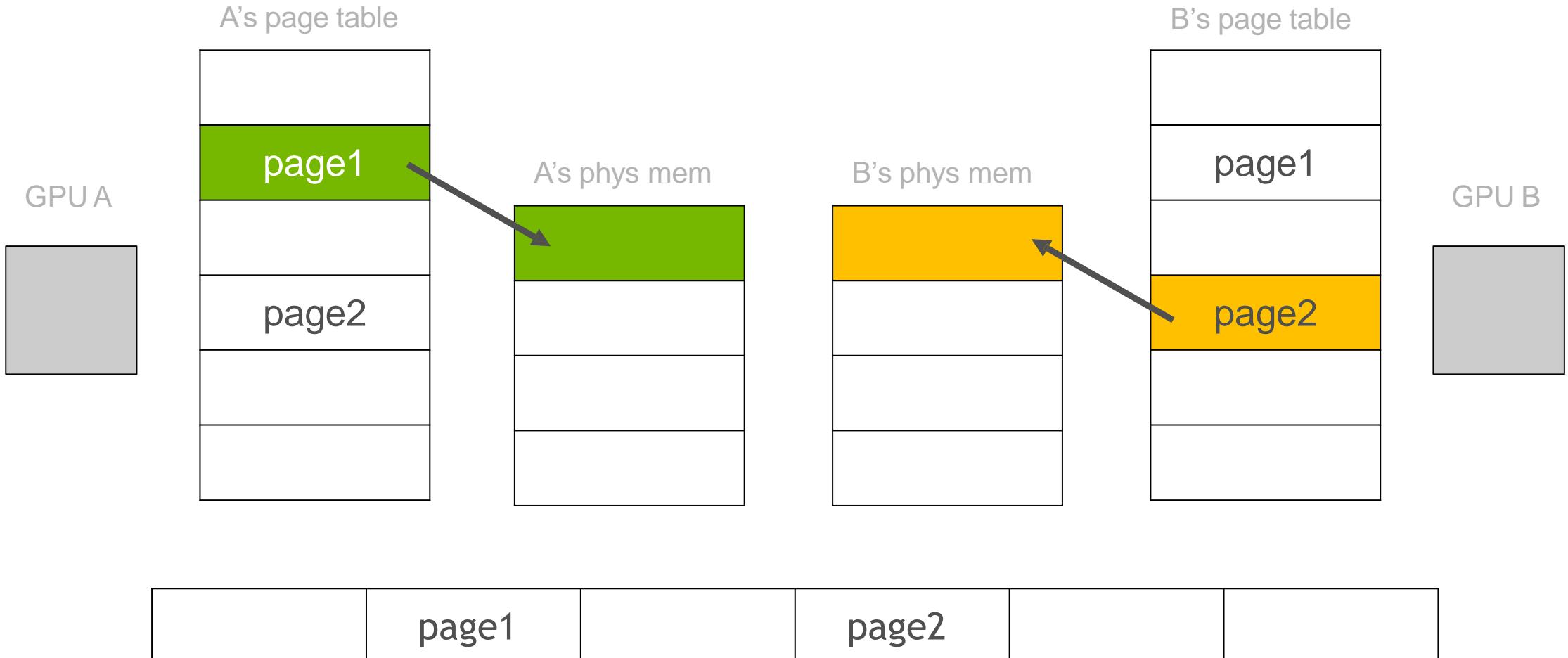


GPU B



Single virtual memory shared between processors

# UNIFIED MEMORY BASICS

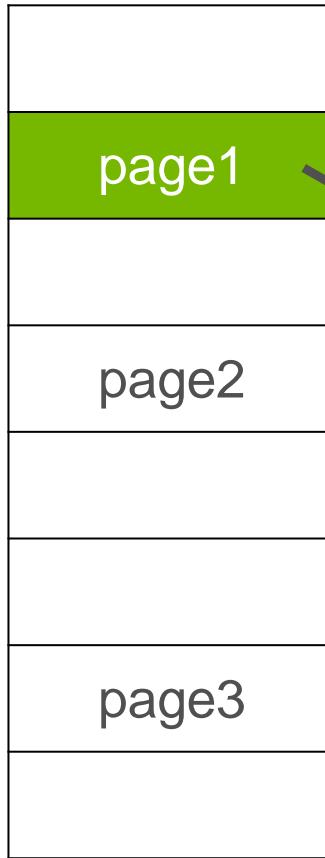


# UNIFIED MEMORY BASICS



# UNIFIED MEMORY BASICS

A's page table

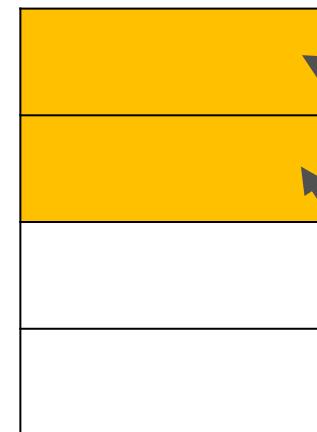


A's phys mem



B's page table

B's phys mem



page1

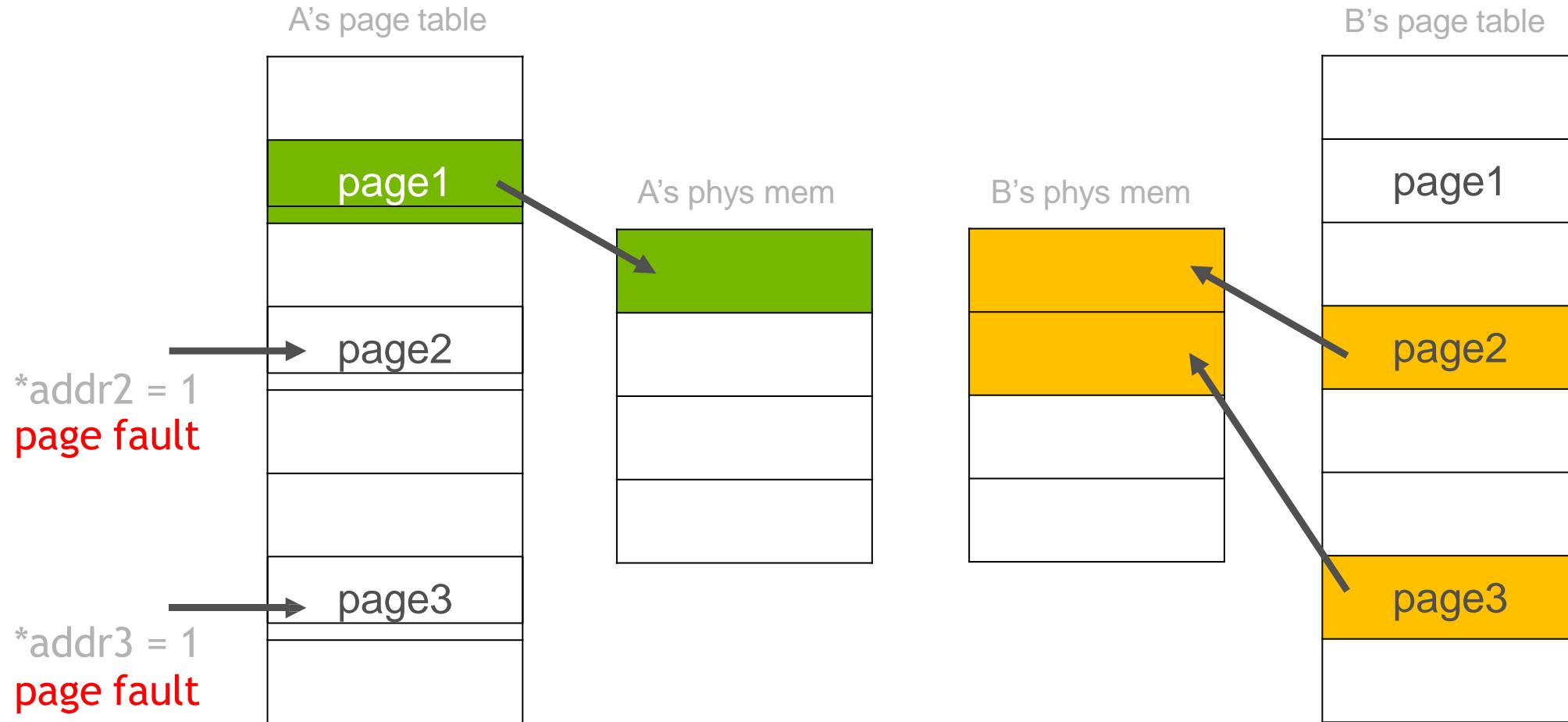
page2

page3

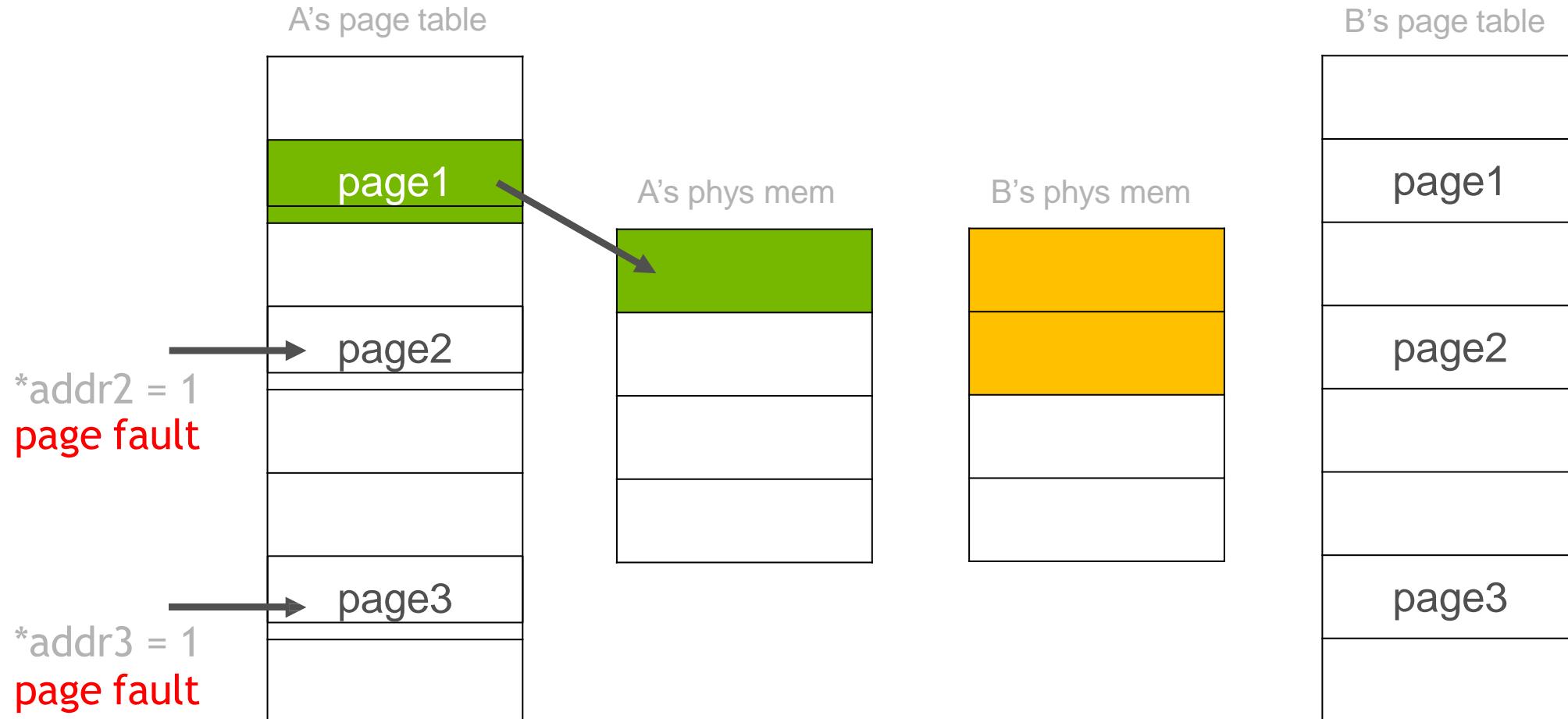
\*addr3 = 1  
access replay

page3 populated and mapped into B's memory

# UNIFIED MEMORY BASICS

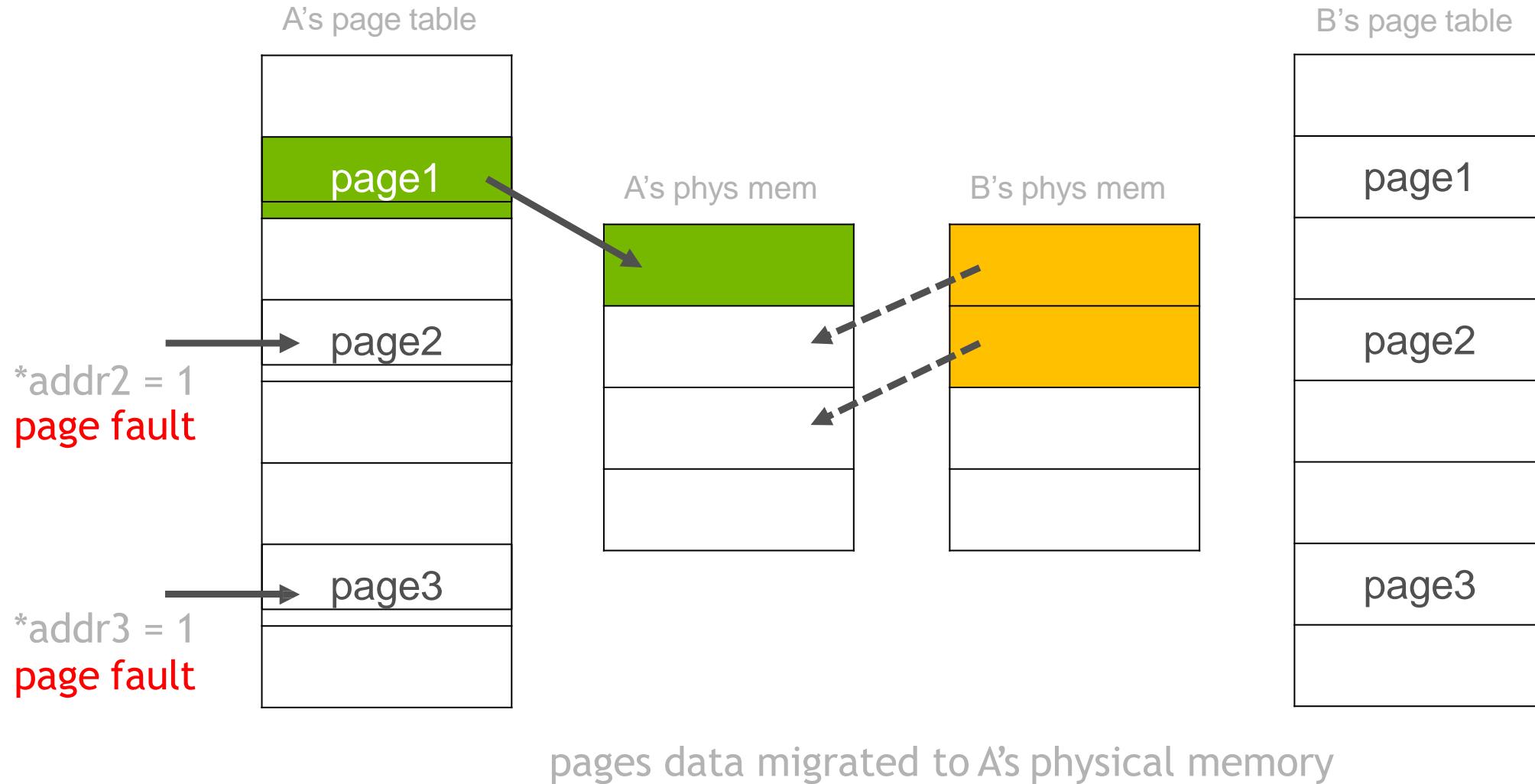


# UNIFIED MEMORY BASICS

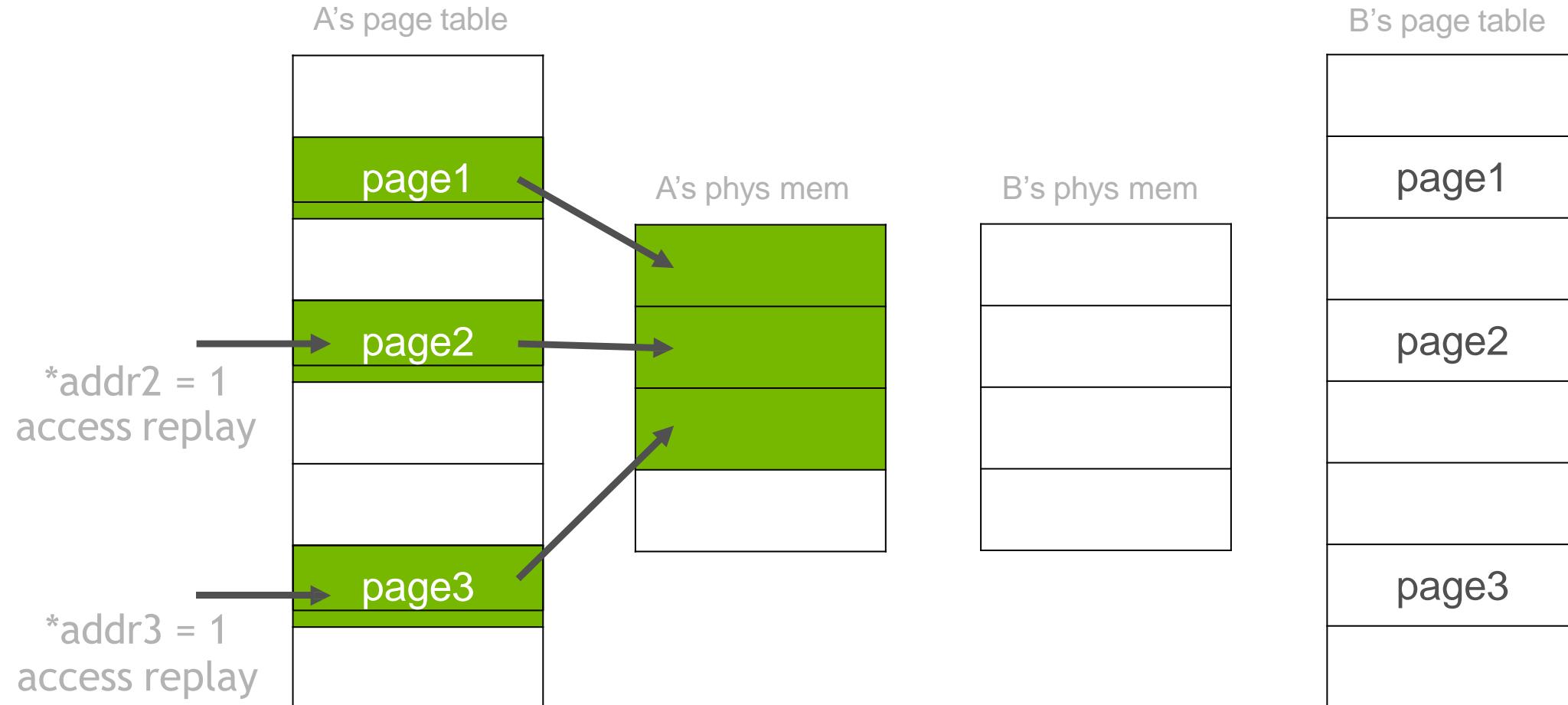


page2 and page3 unmapped from B's memory

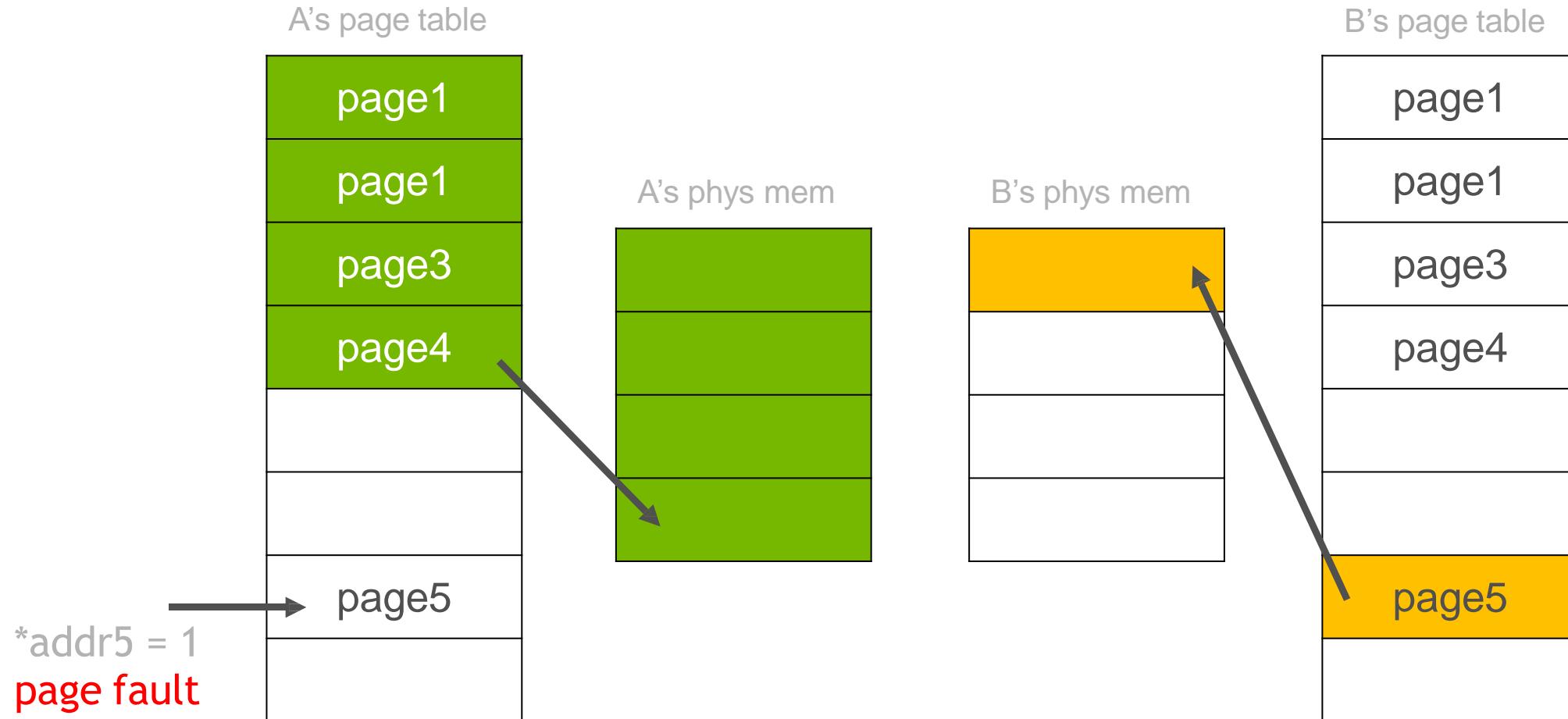
# UNIFIED MEMORY BASICS



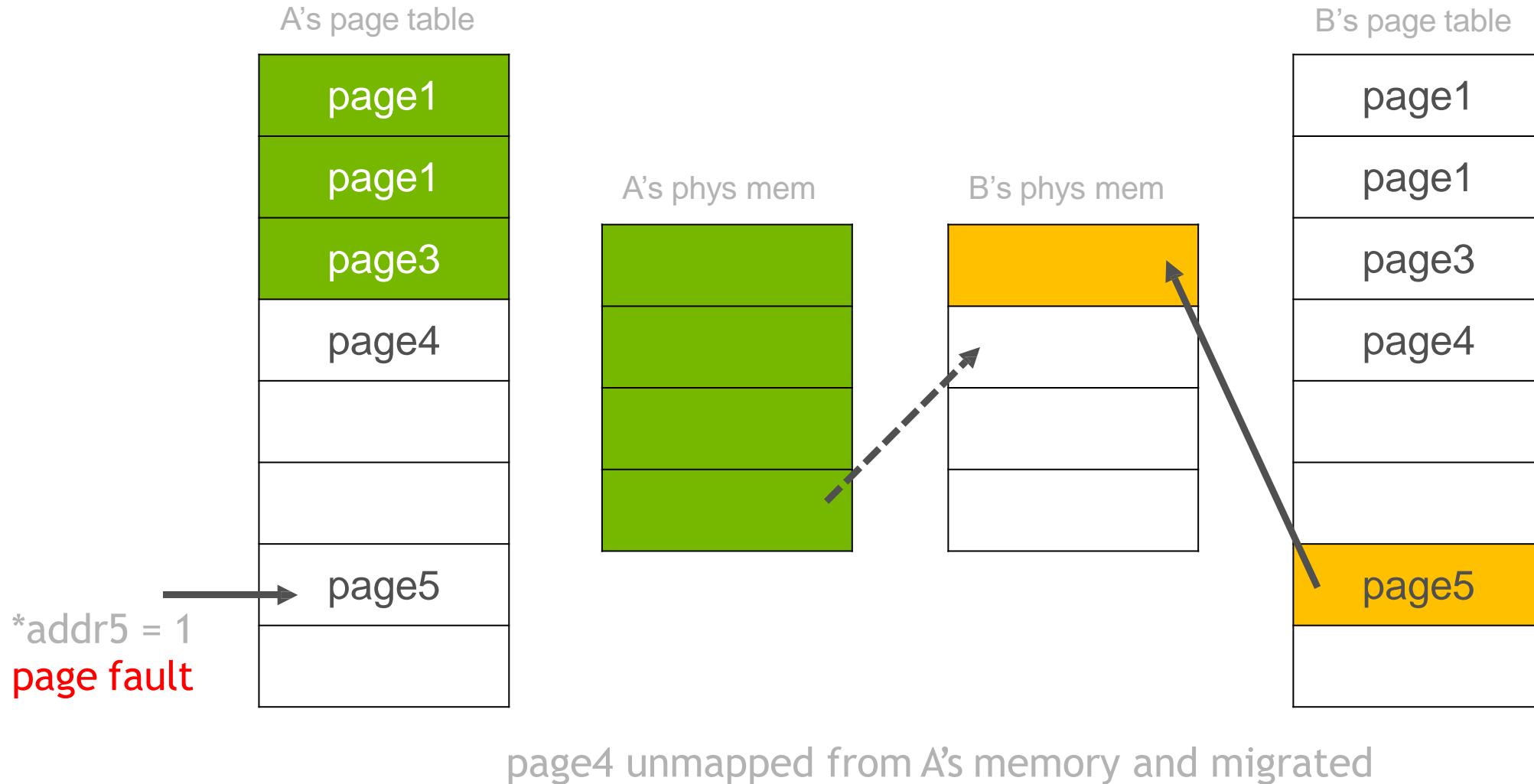
# UNIFIED MEMORY BASICS



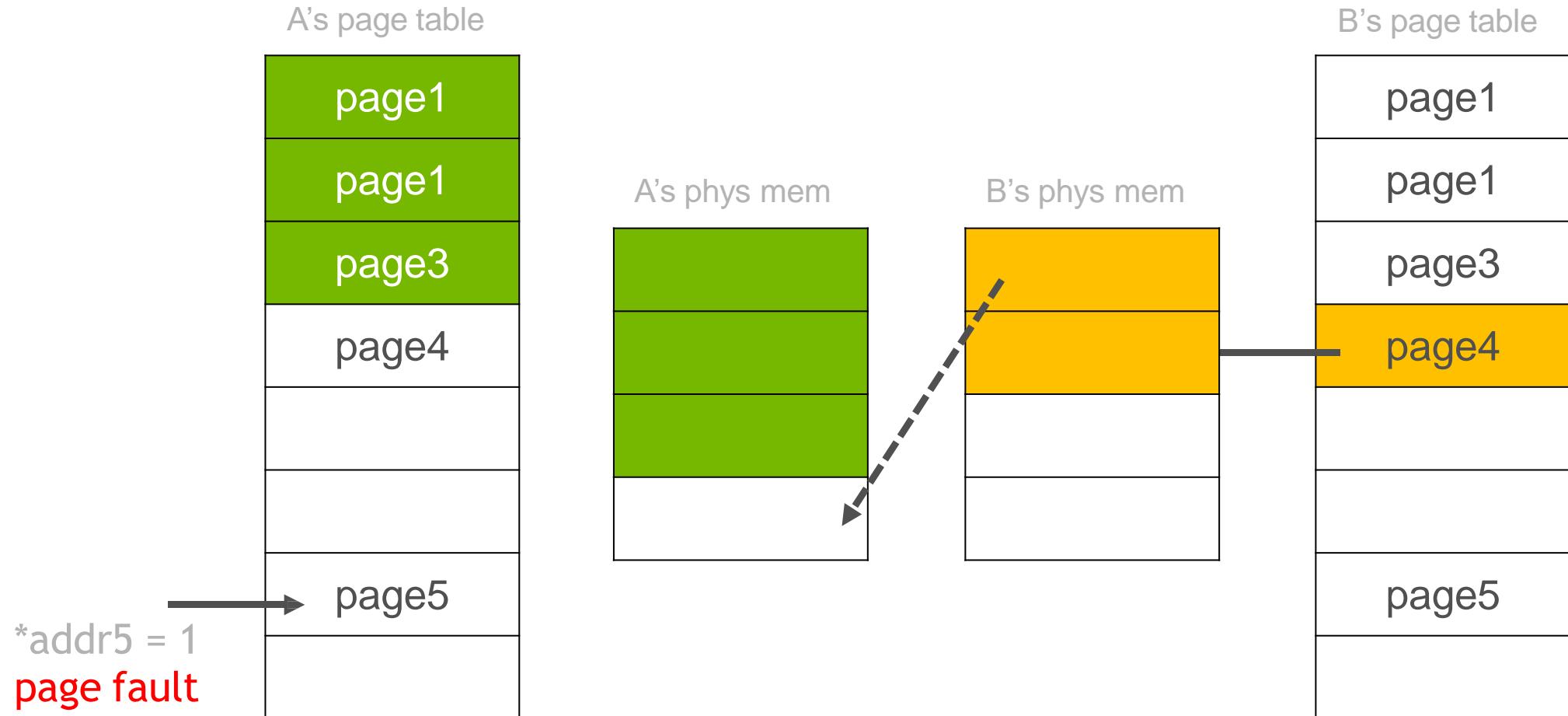
# MEMORY OVERSUBSCRIPTION



# MEMORY OVERSUBSCRIPTION

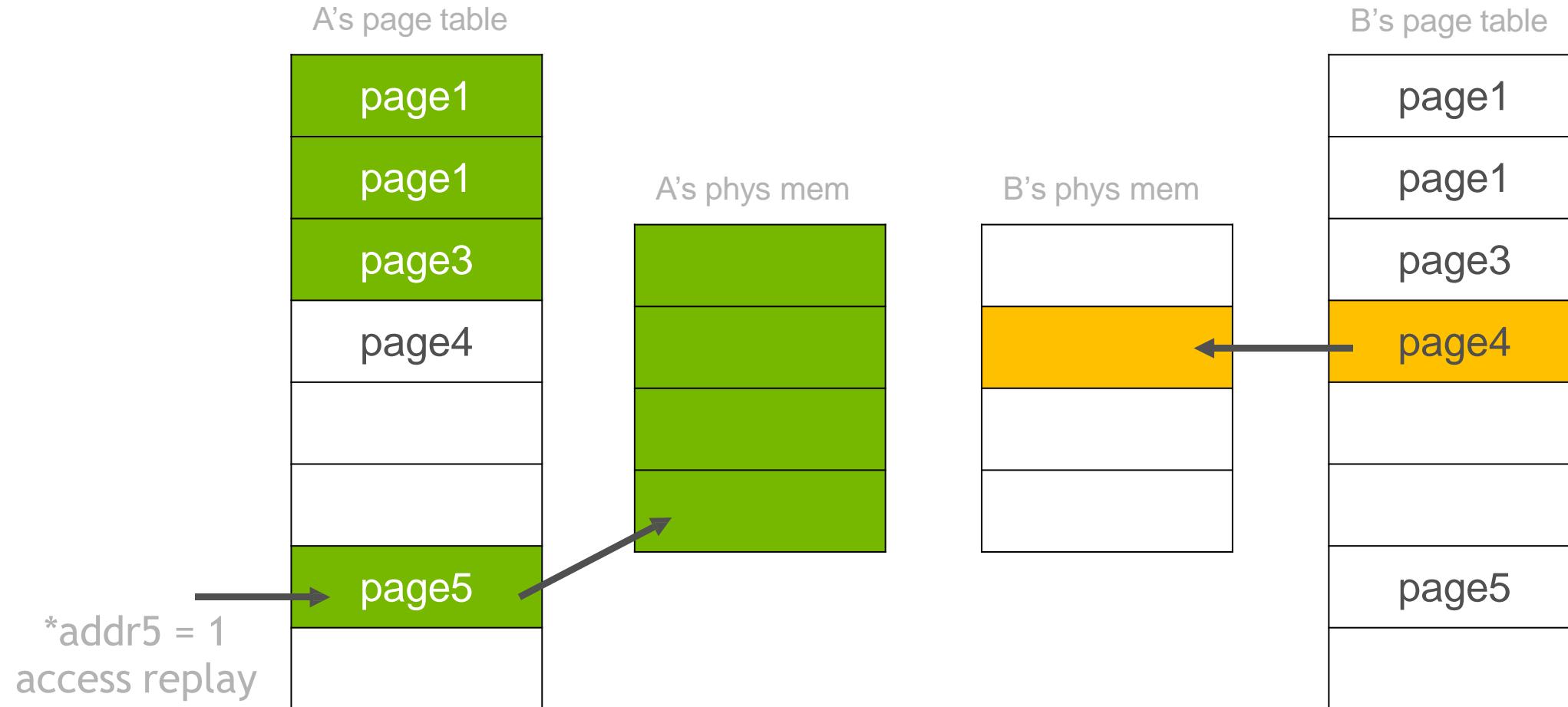


# MEMORY OVERSUBSCRIPTION



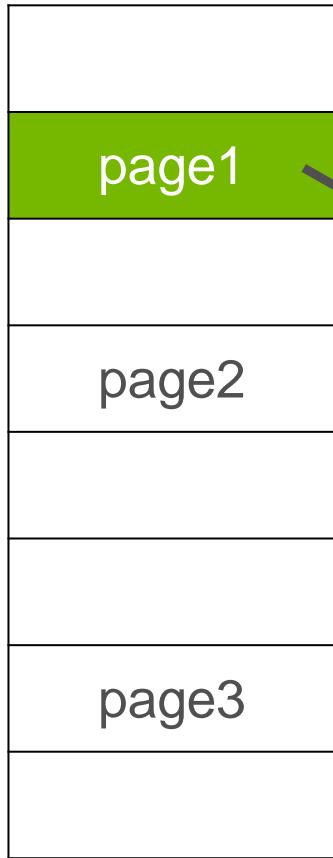
page4 mapped in B's memory, page5 unmapped and migrated to A

# MEMORY OVERSUBSCRIPTION



# CONCURRENT ACCESS

A's page table

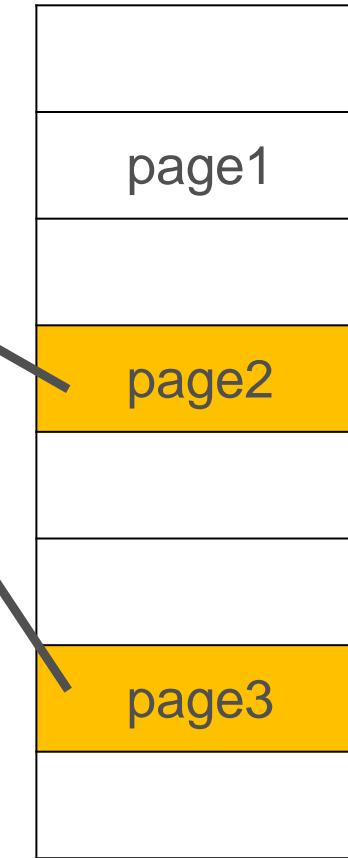
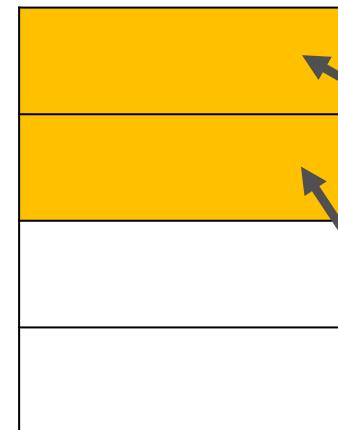


A's phys mem



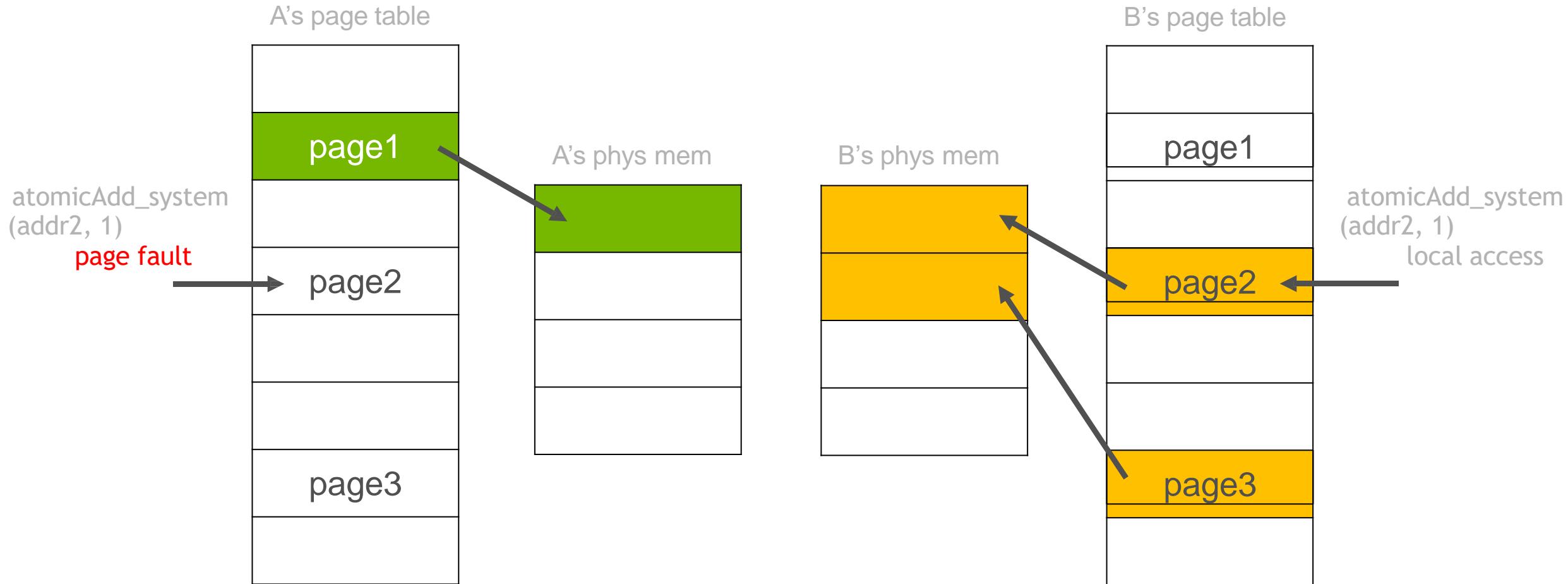
B's page table

B's phys mem



# CONCURRENT ACCESS

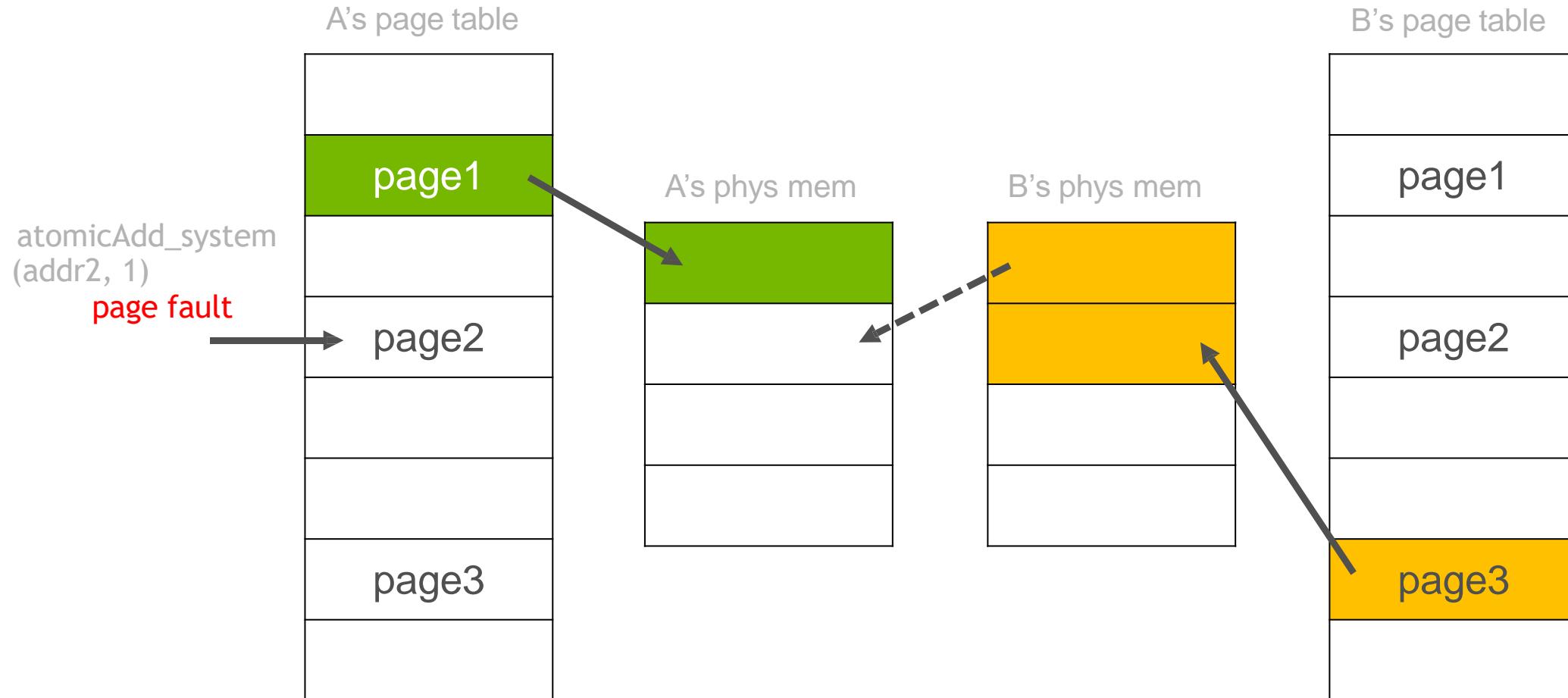
## Exclusive Access\*



\*this is a possible implementation and to guarantee this behavior you need to use `cudaMemAdvise` policies

# CONCURRENT ACCESS

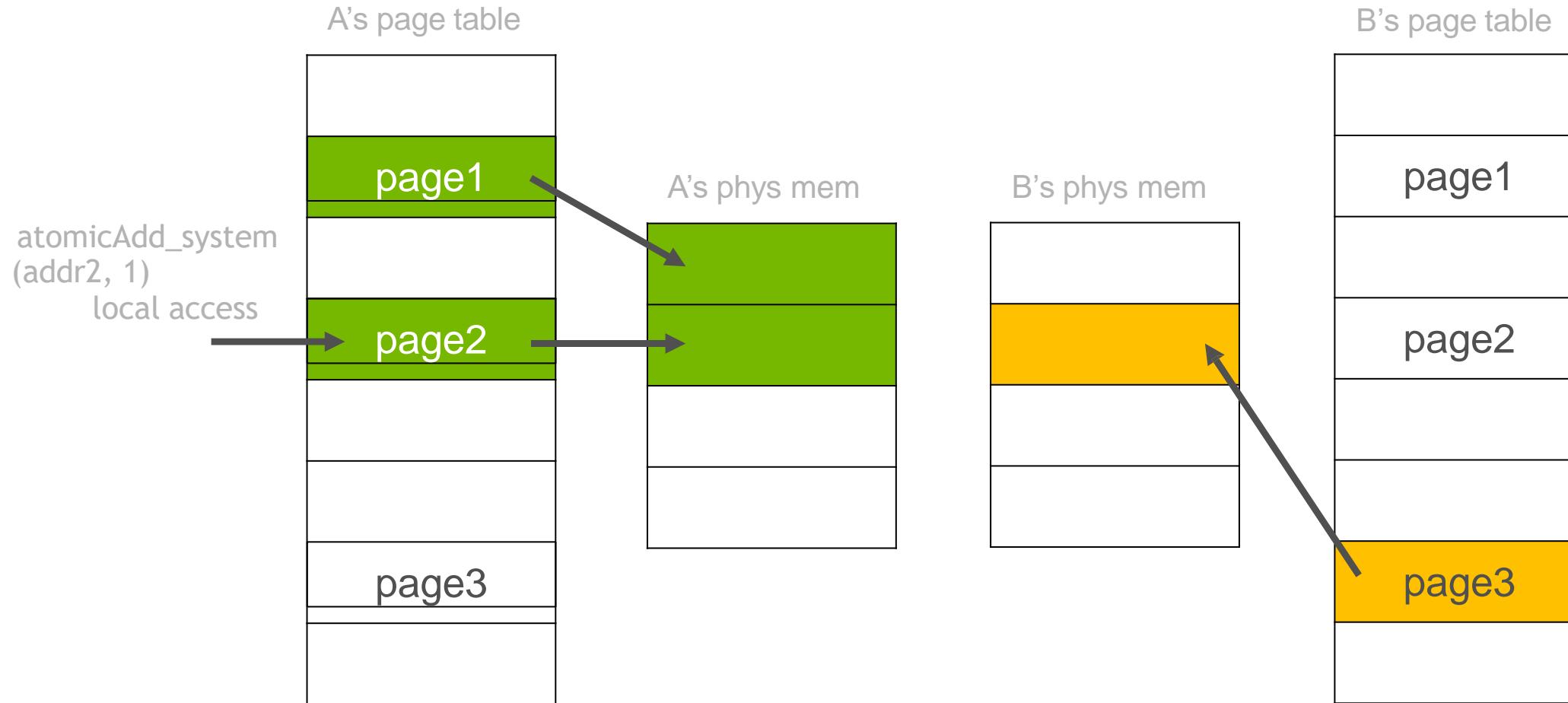
## Exclusive Access



page2 unmapped in B's memory and migrated to A

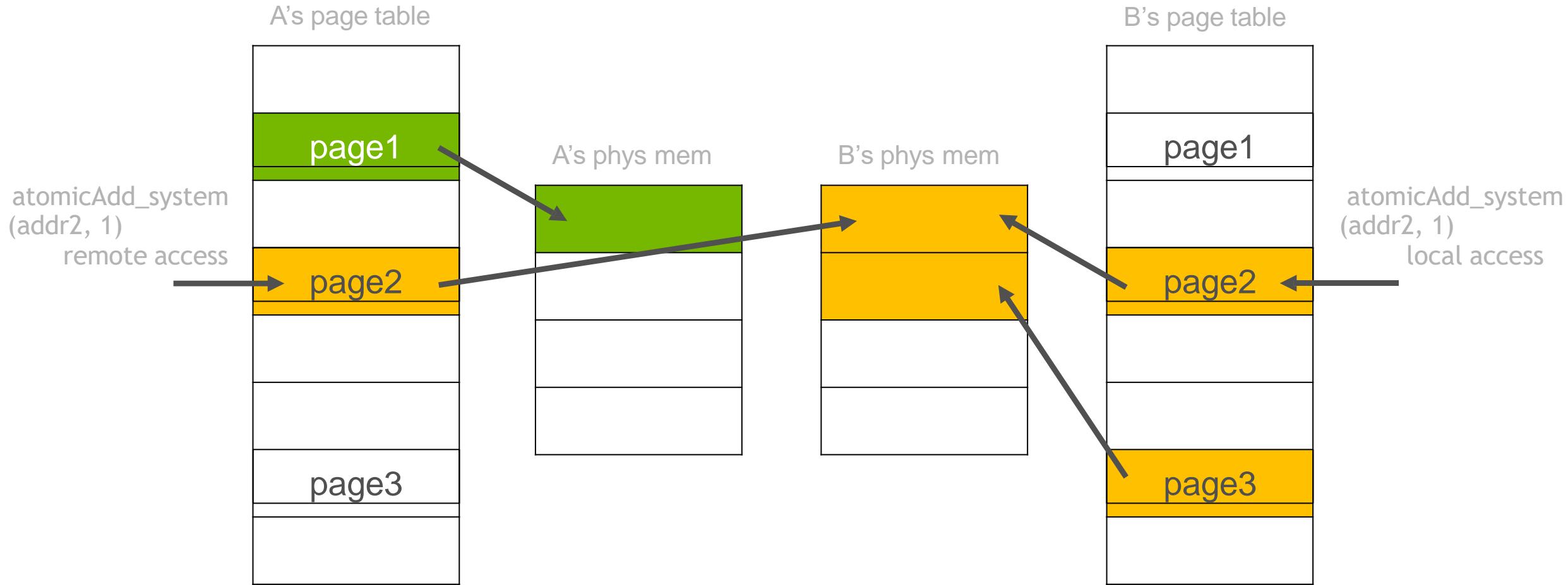
# CONCURRENT ACCESS

## Exclusive Access



# CONCURRENT ACCESS

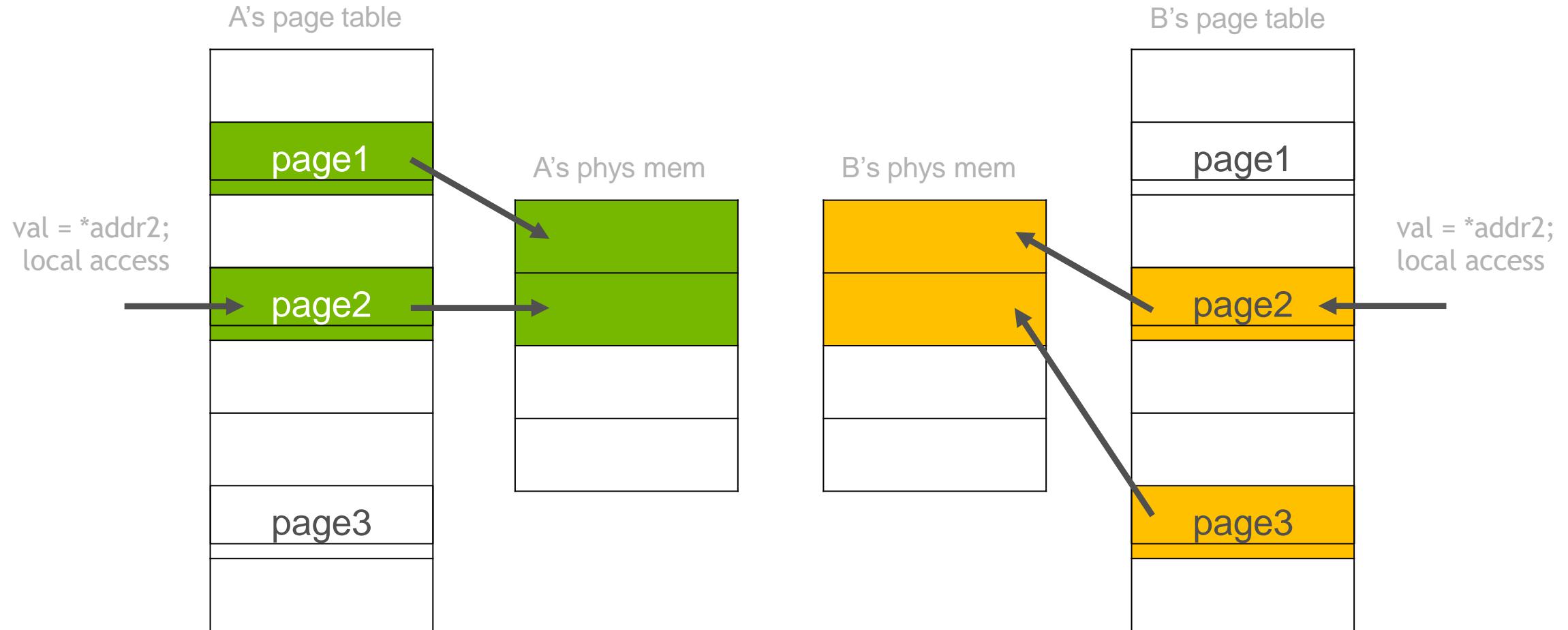
## Atomsics over NVLINK\*



\*both processors need to support atomic operations

# CONCURRENT ACCESS

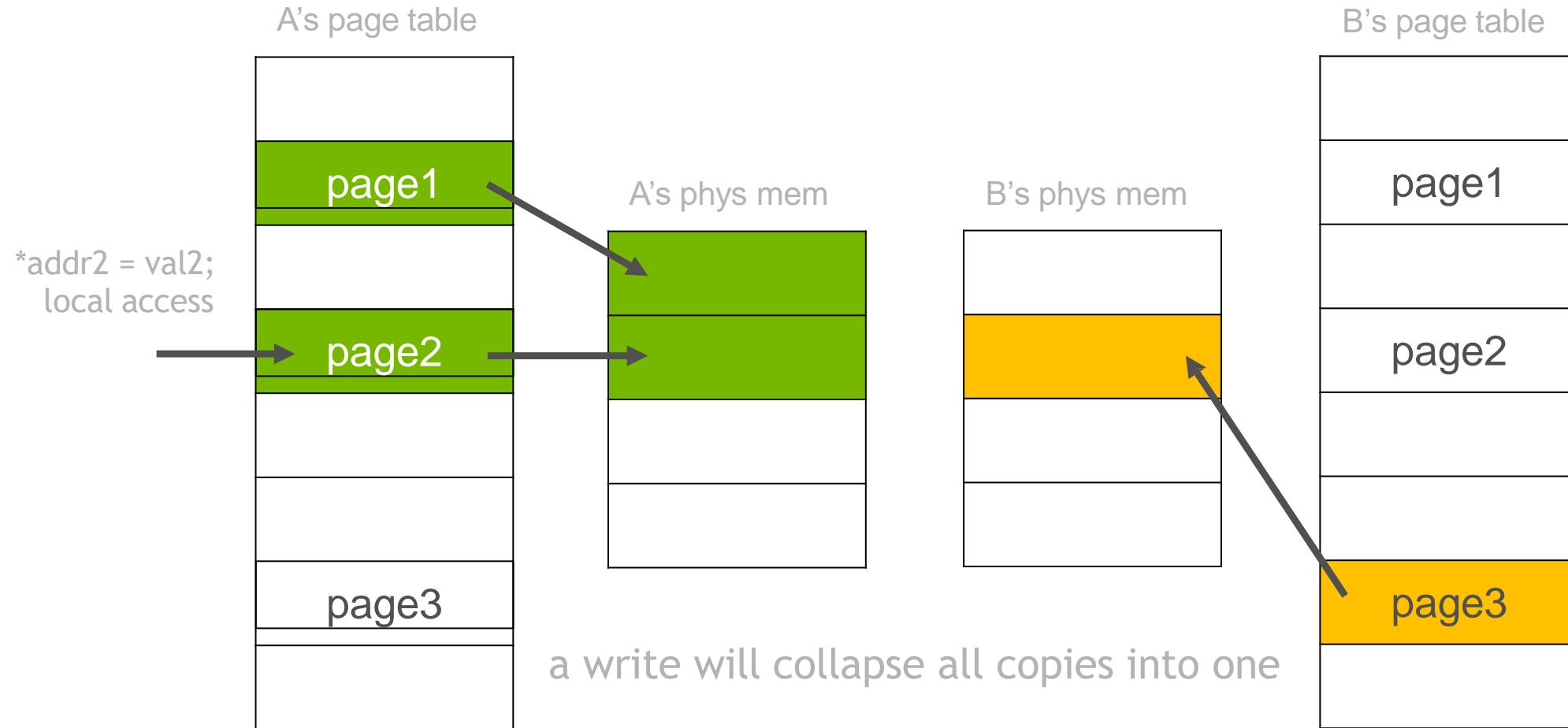
## Read duplication\*



\*each processor must maintain its own page table

# CONCURRENT ACCESS

Read duplication: write



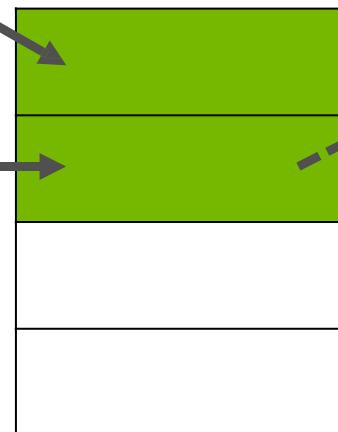
# CONCURRENT ACCESS

Read duplication: read after write

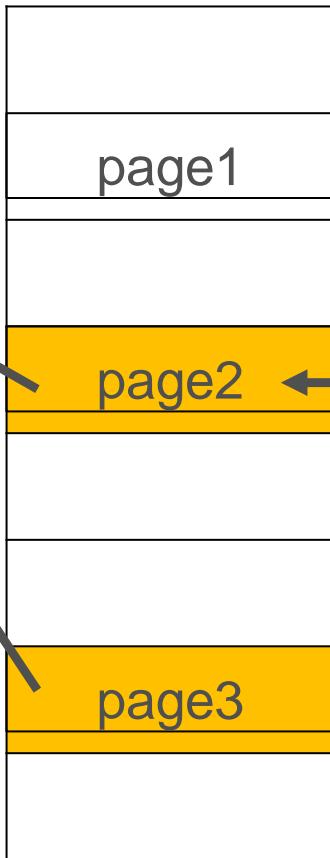
A's page table



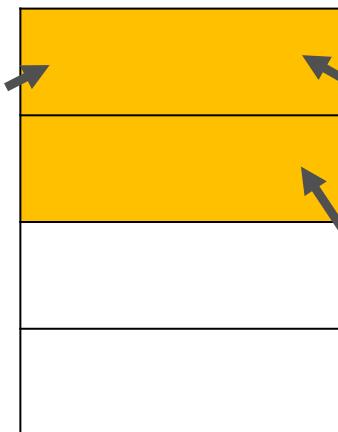
A's phys mem



B's page table



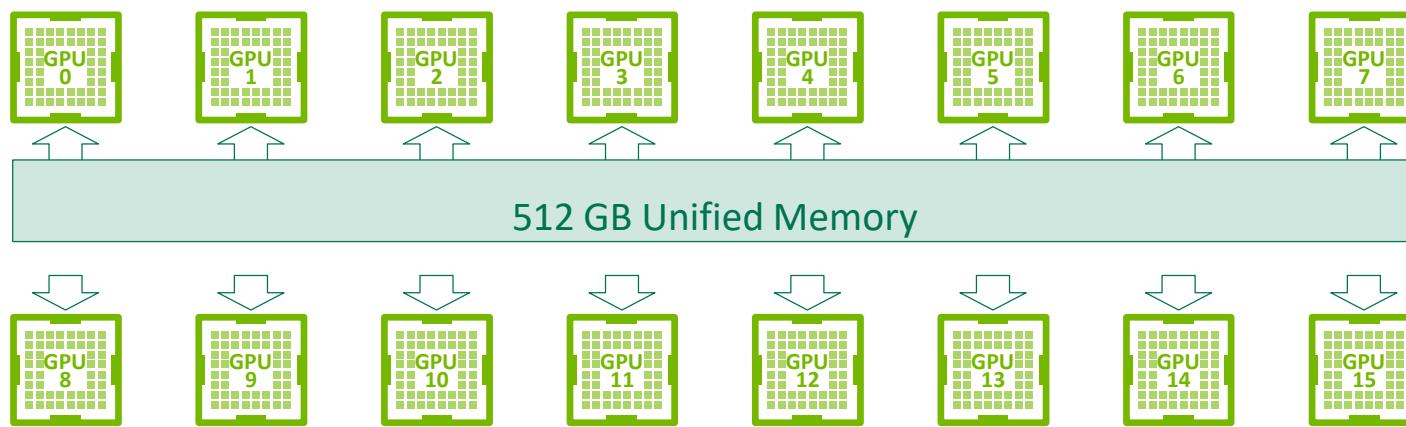
B's phys mem



val = \*addr2;  
local access

pages are duplicated again on faults

# UNIFIED MEMORY + DGX-2



## UNIFIED MEMORY PROVIDES

Single memory view shared by all GPUs

Automatic migration of data between GPUs

User control of data locality

# ENABLING MULTI-GPU

## Single-GPU

```
__global__ void kernel(int *data) {  
    int idx = threadIdx.x + blockDim.x * blockIdx.x;  
  
    doSomeStuff(idx, data, ...);  
}  
  
cudaMallocManaged(&data, N * sizeof(int));  
// initialize data on the CPU  
  
kernel<<<grid, block>>>(data);
```

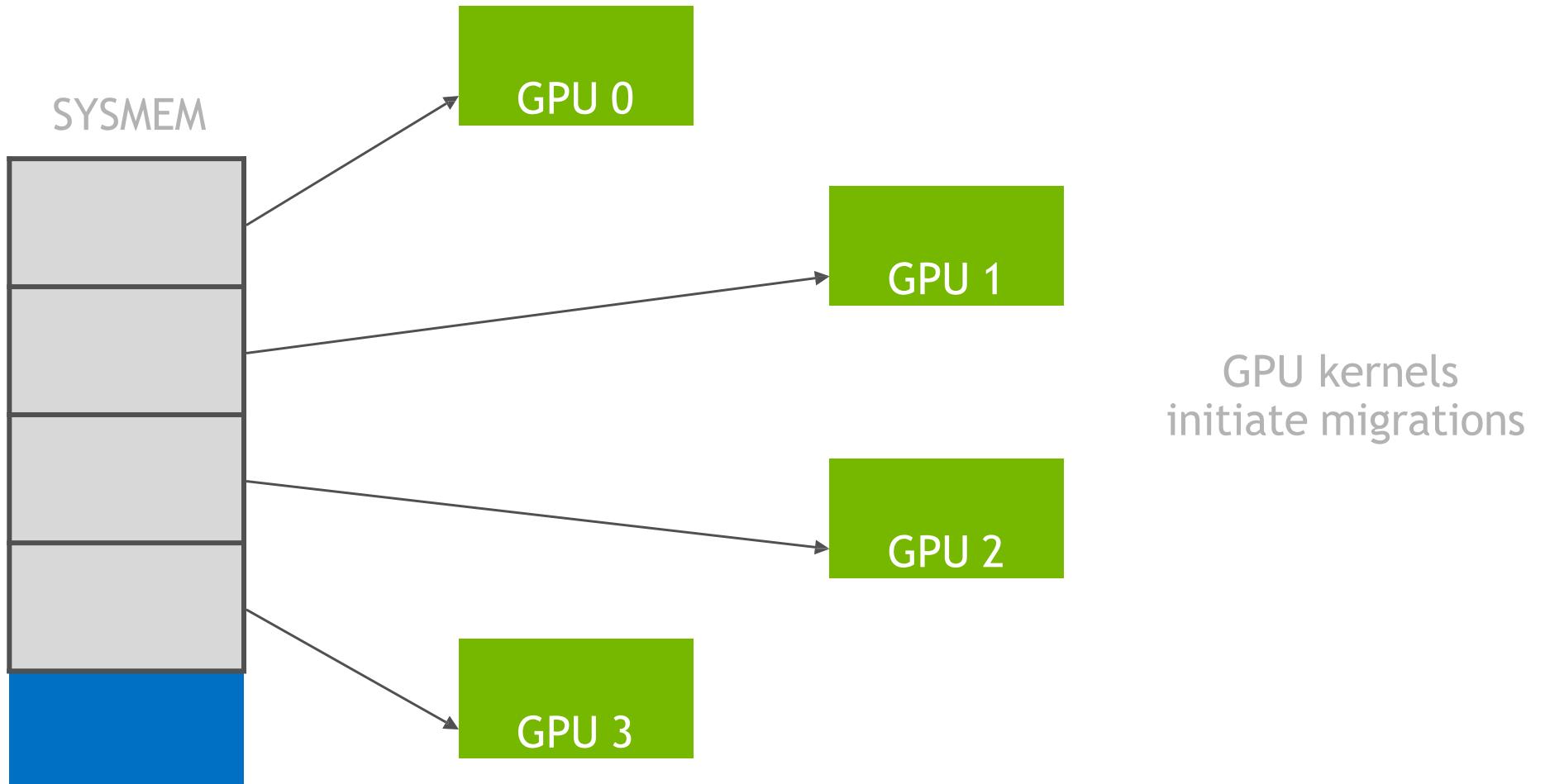
## Multi-GPU

```
__global__ void kernel(int *data, int gpuId) {  
    int idx = threadIdx.x + blockDim.x * (blockIdx.x  
        + gpuId * gridDim.x);  
  
    doSomeStuff(idx, data, ...);  
}  
  
cudaMallocManaged(&data, N * sizeof(int));  
// initialize data on the CPU  
for (int i = 0; i < numGPUs; i++) {  
    cudaSetDevice(i);  
    kernel<<<grid/numGPUs, block>>>(data, i);  
}
```

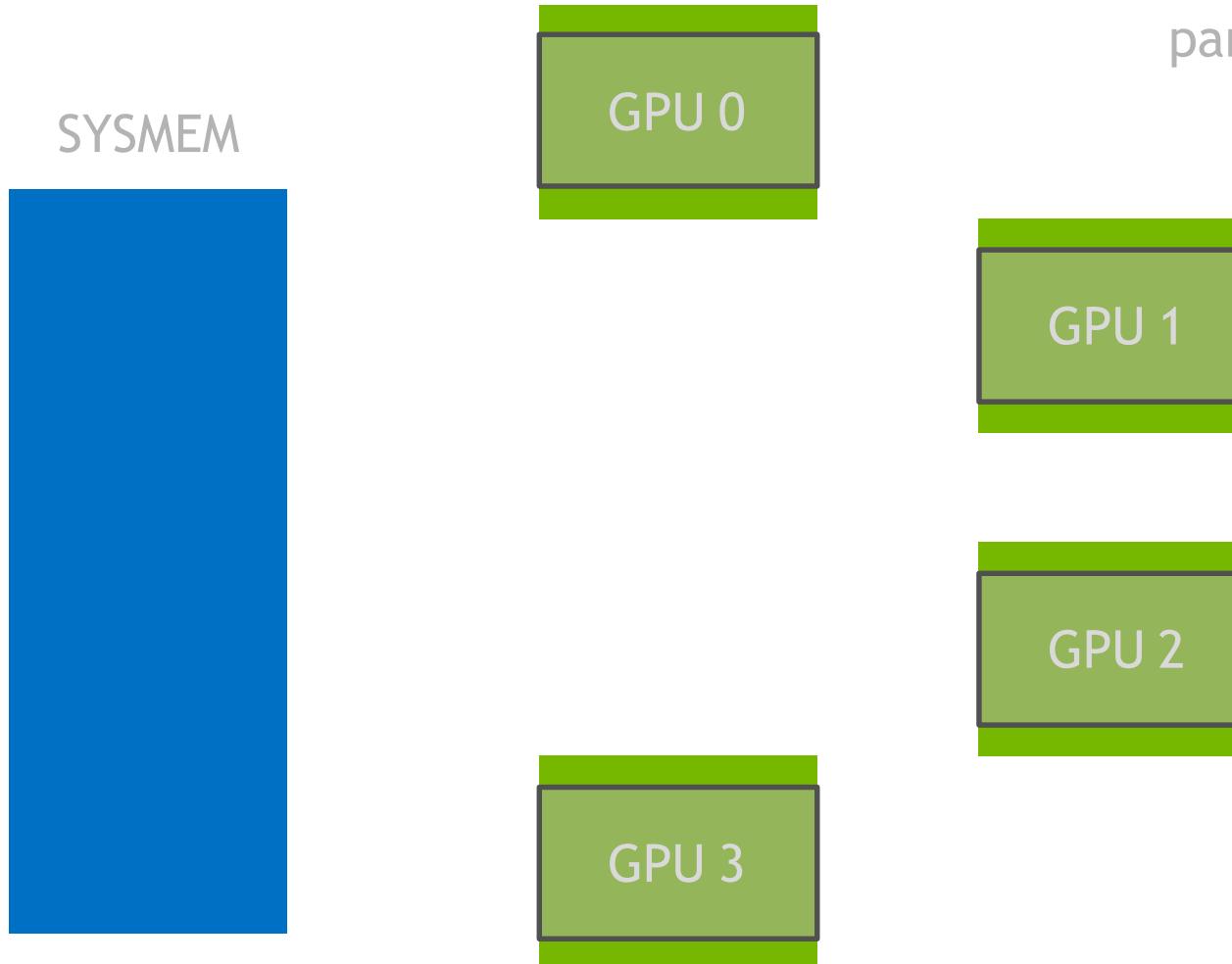
update blockIdx.x

update launch config

# MULTI-GPU WITH UNIFIED MEMORY

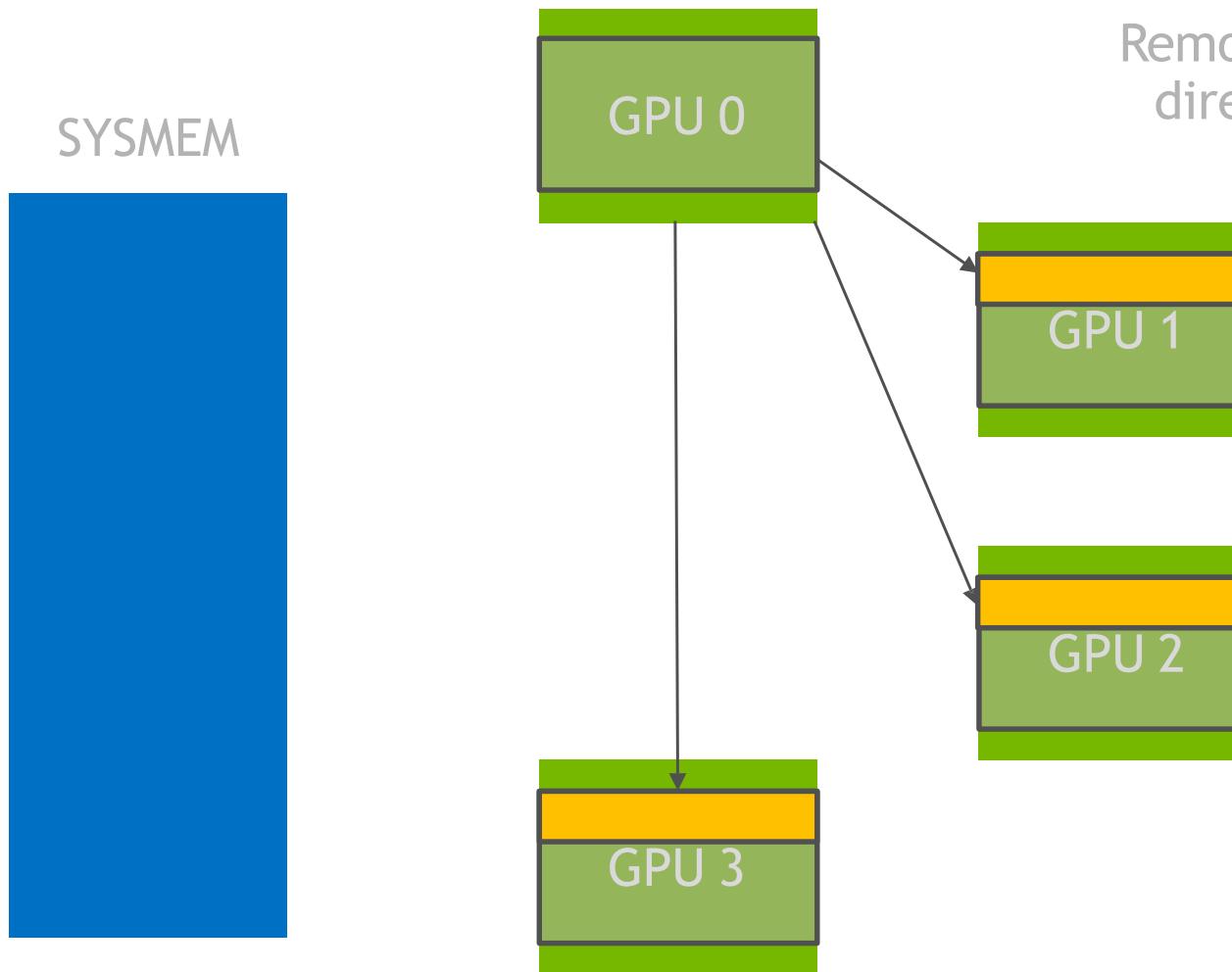


# MULTI-GPU WITH UNIFIED MEMORY



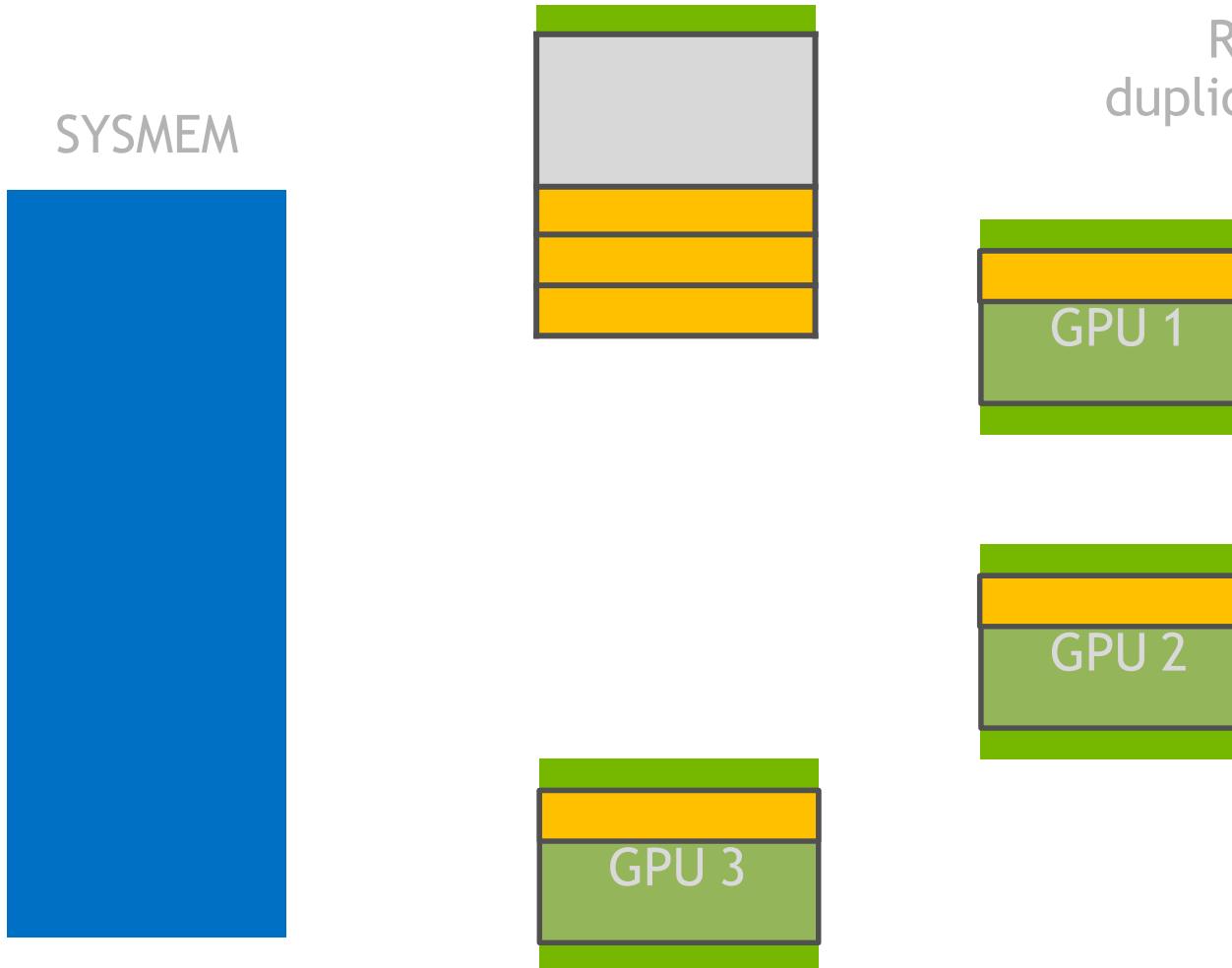
*Data automatically  
partitioned between GPUs  
on first-touch*

# MULTI-GPU WITH UNIFIED MEMORY



**With policies:**  
Remote data can be accessed  
directly without migrations

# MULTI-GPU WITH UNIFIED MEMORY



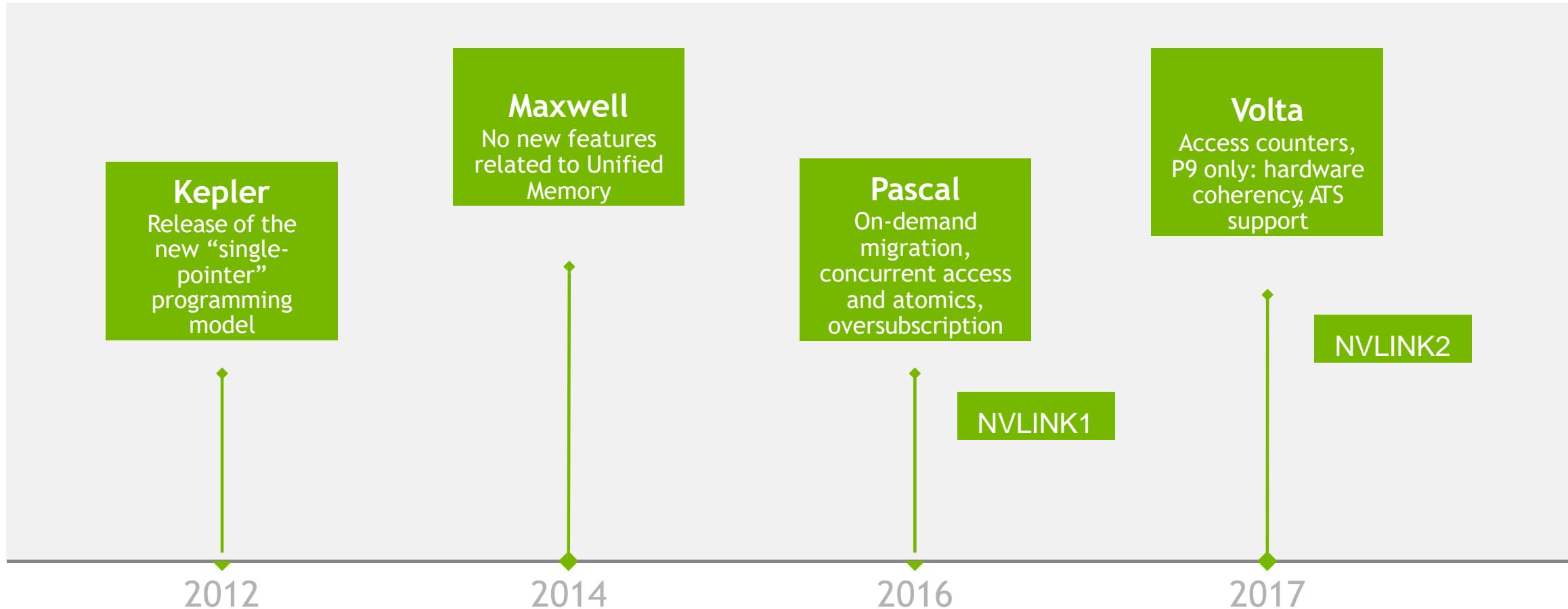
With policies:

Read-only data can be  
duplicated and accessed locally

# GPU ARCHITECTURE AND SOFTWARE EVOLUTION

# UNIFIED MEMORY

## Evolution of GPU architectures



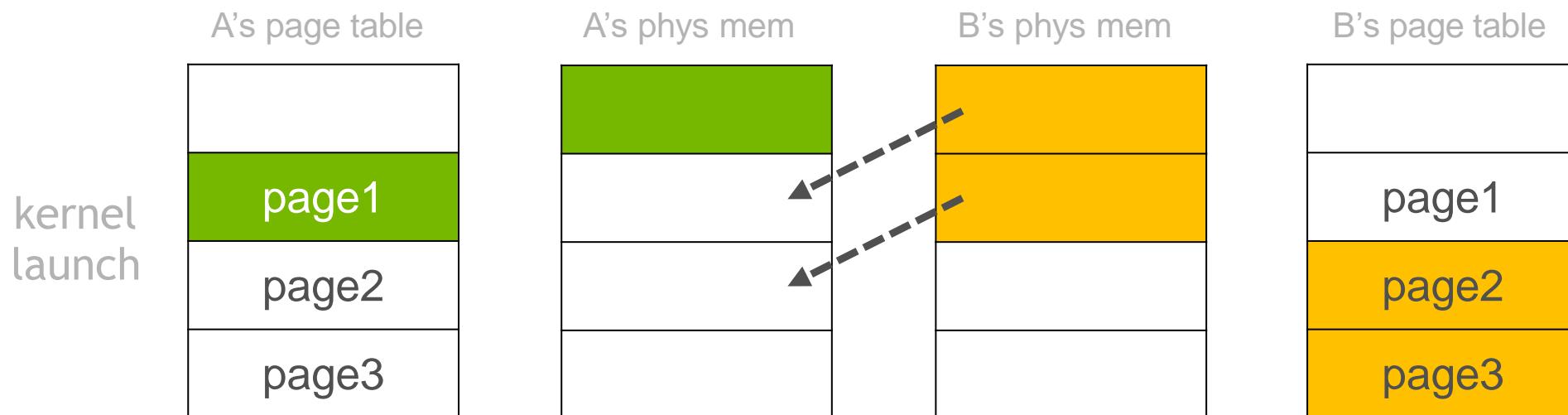
\*Not all features are available on all platforms

# UNIFIED MEMORY: BEFORE PASCAL

Available since CUDA 6

No GPU page fault support: move all dirty pages on kernel launch

**No concurrent access, no GPU memory oversubscription, no system-wide atomics**

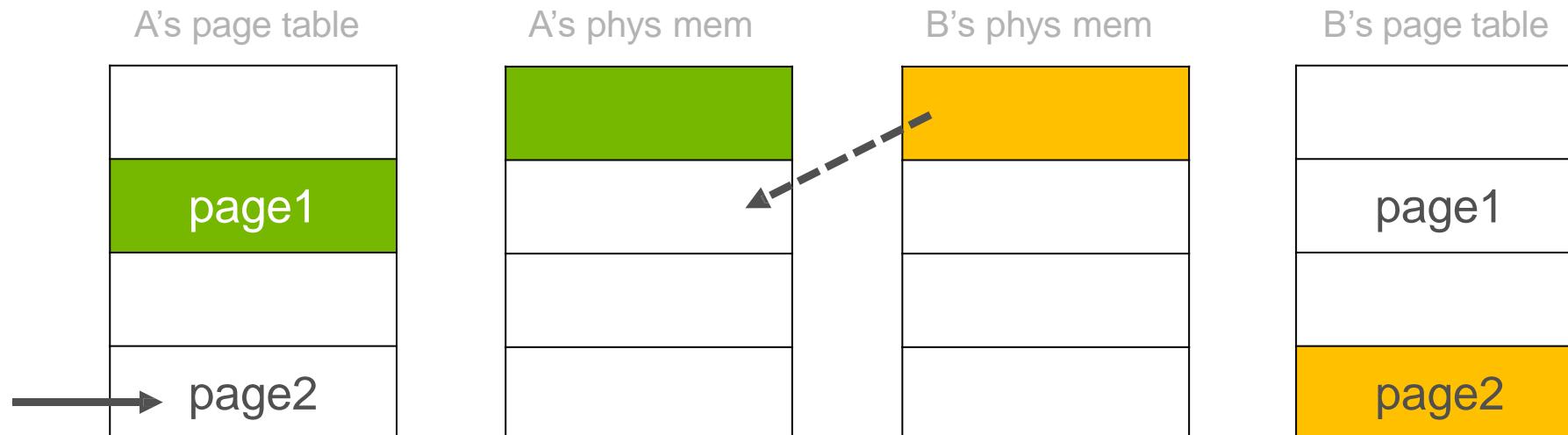


# UNIFIED MEMORY: PASCAL AND VOLTA

Available since CUDA 8

GPU page fault support, concurrent access, extended VA space (48-bit)

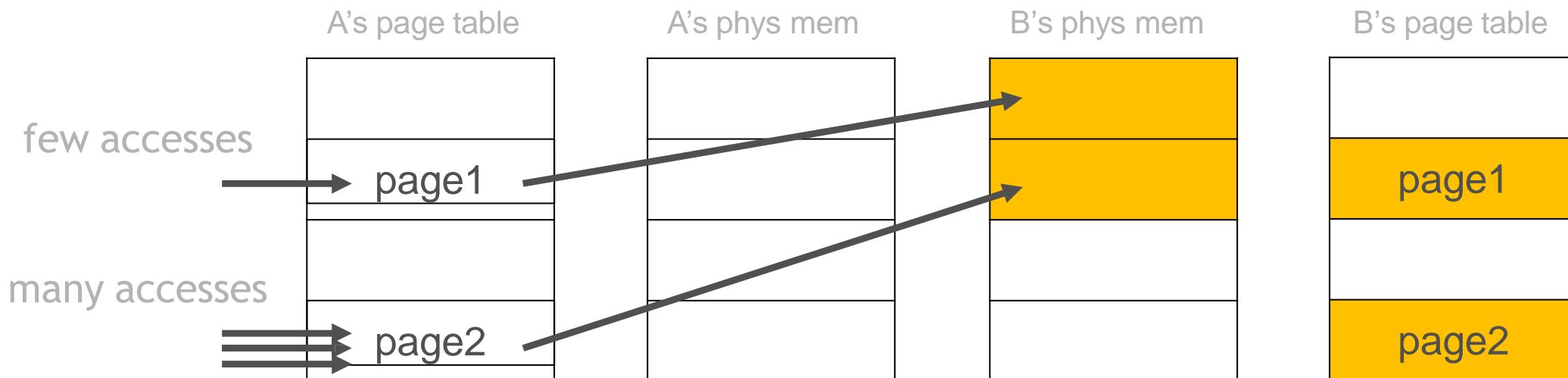
On-demand migration to accessing processor **on first touch**



# UNIFIED MEMORY ON VOLTA+P9

## New Feature: Access Counters

If memory is mapped to the GPU, migration can be triggered by access counters

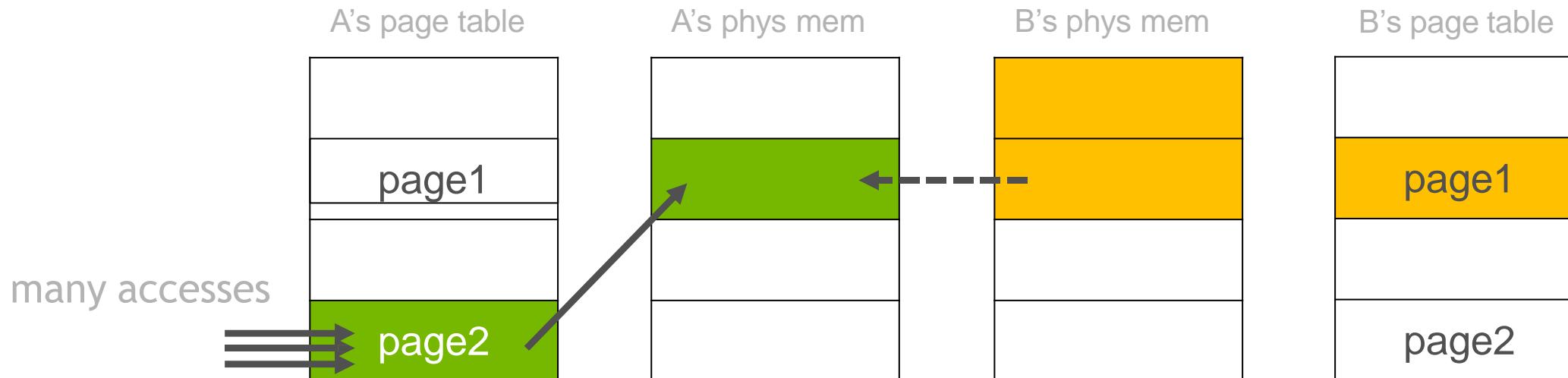


# UNIFIED MEMORY ON VOLTA+P9

## New Feature: Access Counters

With access counters **only hot pages** will be moved to the GPU

Migrations are *delayed* compared to the fault-based method



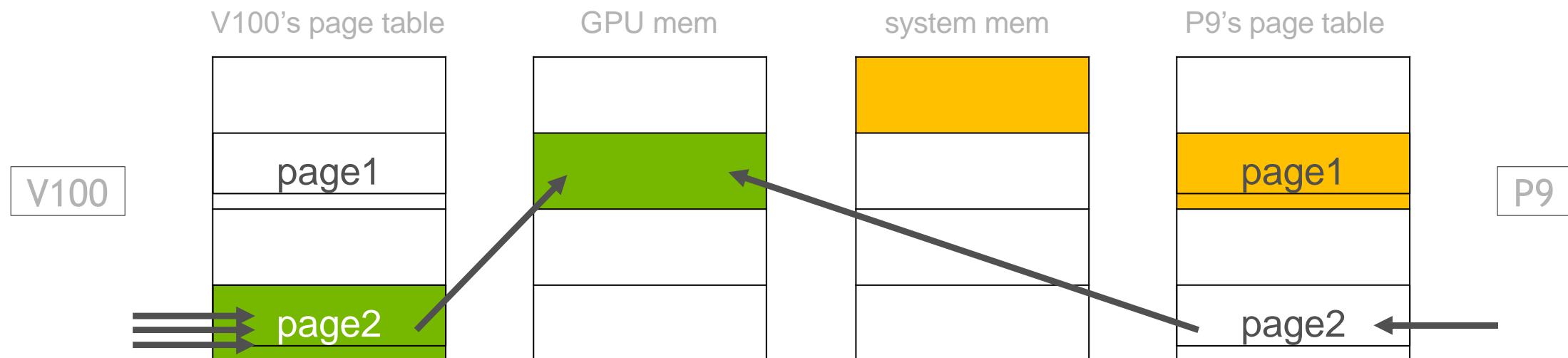
\*When implemented this feature can be enabled with `cudaMemAdvise` policies

# UNIFIED MEMORY ON VOLTA+P9

New Feature: Hardware Coherency with NVLINK2

CPU can directly access and *cache* GPU memory

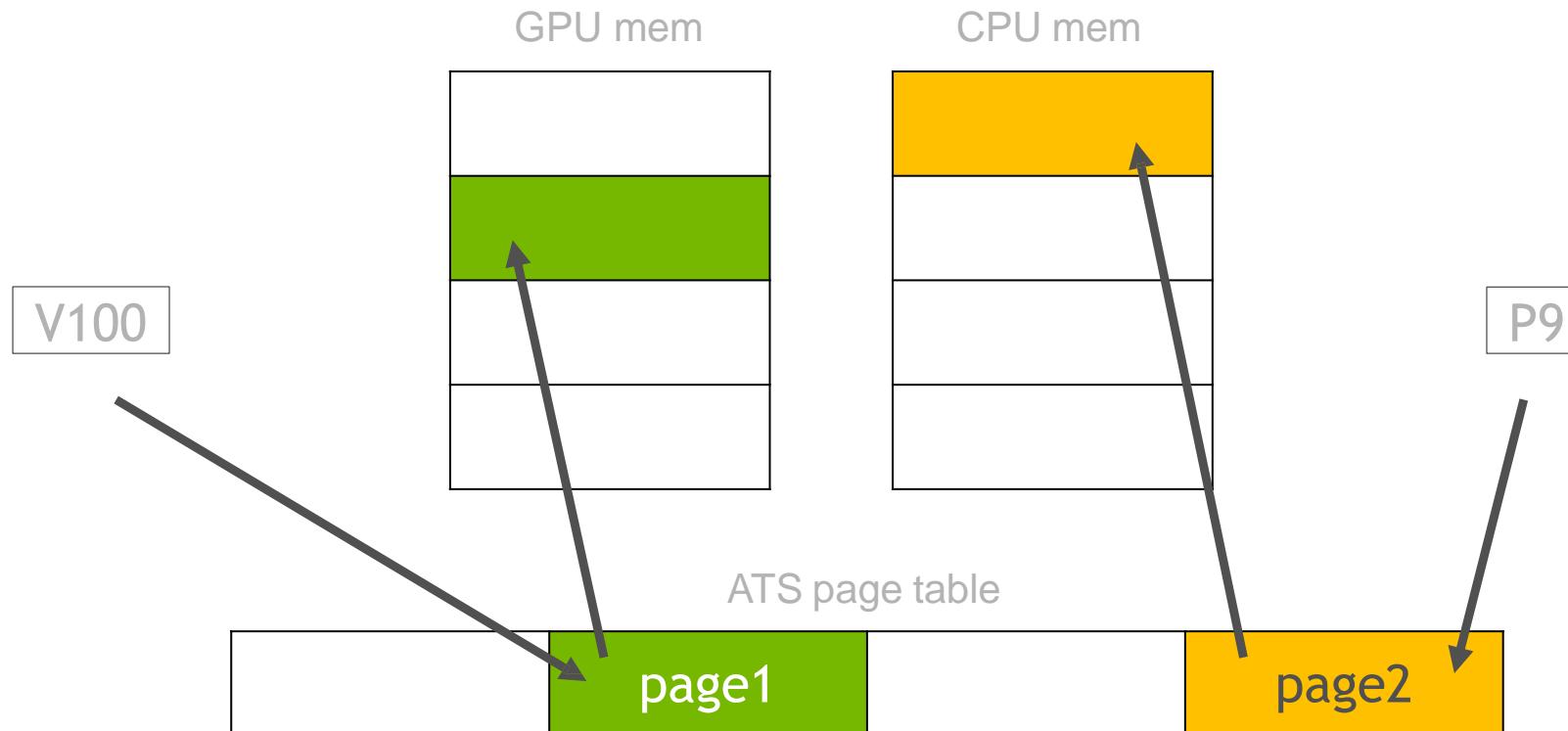
*Native* atomics support for all accessible memory



# UNIFIED MEMORY ON VOLTA+P9

New Feature: ATS support

ATS: address translation service; CPU and GPU can share a *single* page table



# UNIFIED MEMORY WITH SYSTEM ALLOCATOR

System allocator support allows GPU to access all system memory  
malloc, stack, global, file system

## P9: Address Translation Service (ATS)

Support enabled in CUDA 9.2

## x86: Heterogeneous Memory Management (HMM)

Initial version of the patchset is integrated into 4.14 kernel

NVIDIA will be supporting upcoming versions of HMM

# WHAT YOU CAN DO WITH UNIFIED MEMORY

See it in action at the end of the talk!

Works everywhere today

```
int *data;  
cudaMallocManaged(&data, sizeof(int) *  
n); kernel<<<grid, block>>>(data);
```

Works on Power9 + ATS in CUDA 9.2  
Will work in the future on x86 + HMM

```
int *data = (int*)malloc(sizeof(int) * n);  
kernel<<<grid, block>>>(data);
```

```
int data[1024];  
kernel<<<grid, block>>>(data);
```

```
int *data = (int*)alloca(sizeof(int) * n);  
kernel<<<grid, block>>>(data);
```

```
extern int *data;  
kernel<<<grid, block>>>(data);
```

# UNIFIED MEMORY LANGUAGES

CUDA C/C++: `cudaMallocManaged`

CUDA Fortran: `managed` attribute (per allocation)

Python: `pycuda.driver.managed_empty` (allocate `numpy.ndarray`)

OpenACC: `-ta=managed` compiler option (all dynamic allocations)

# UNIFIED MEMORY + OPENACC

## Effortless way to run your code on GPUs

Literally adding a single line will get your code running on the GPU

```
#pragma acc kernels
{
    for (i = 0; i < n; ++i) {
        c[i] = a[i] + b[i];
        ...
    }
}
...
```

←  
Initiate parallel execution

Easy to optimize later: add loop and data directives

# PERFORMANCE DEEP DIVE

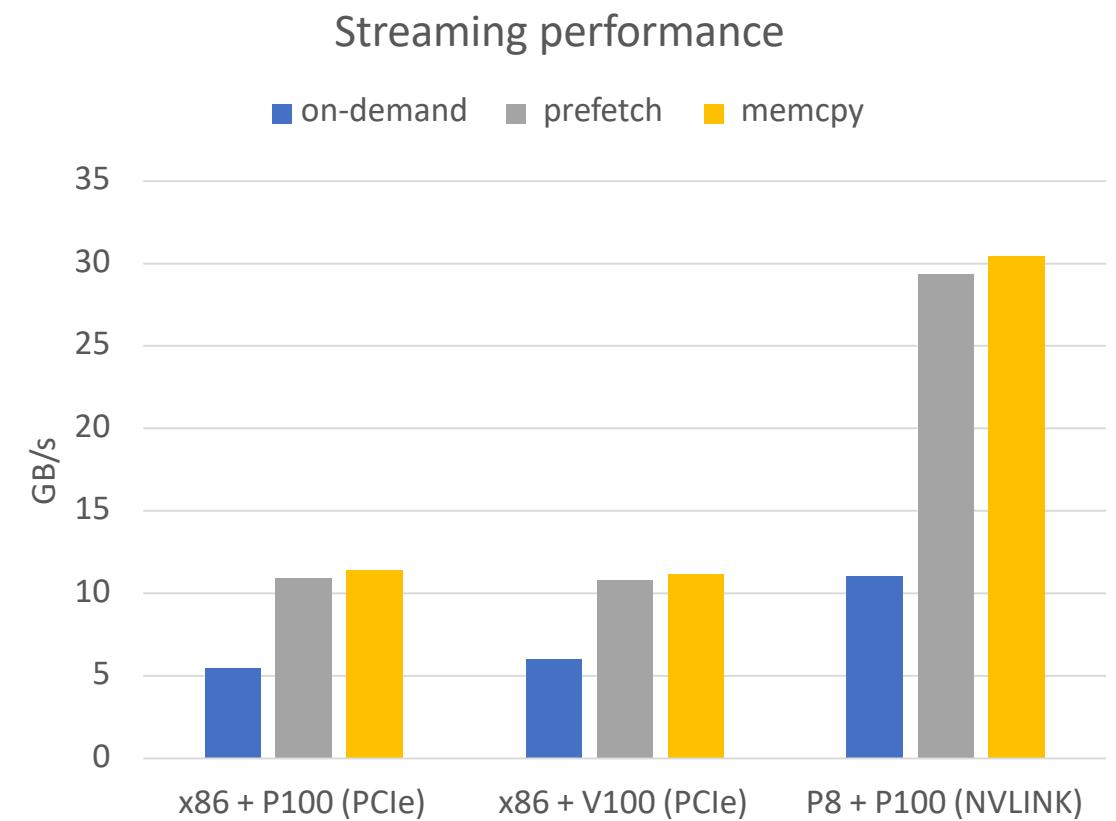
# STREAMING BENCHMARK

## How fast is on-demand migration?

```
__global__ void kernel(int *host, int *device) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    device[i] = host[i];
}

// allocate and initialize memory
cudaMallocManaged(&host, size);
cudaMalloc(&device, size);
memset(host, 0, size);

// benchmark CPU->GPU migration
if (prefetch)
    cudaMemPrefetchAsync(host, size, gpuId);
kernel<<<grid, block>>>(host, device);
```



# UNDERSTANDING PROFILER OUTPUT

```
==14487== Profiling result:
```

Type	Time (%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	100.00%	23.270ms	1	23.270ms	23.270ms	23.270ms	void kernel(int*, int*)
API calls:	79.56%	23.272ms	1	23.272ms	23.272ms	23.272ms	cudaDeviceSynchronize
	20.42%	5.9732ms	1	5.9732ms	5.9732ms	5.9732ms	cudaLaunch
	0.01%	2.0490us	1	2.0490us	2.0490us	2.0490us	cudaConfigureCall
	0.01%	1.8360us	4	459ns	138ns	833ns	cudaSetupArgument

```
==14487== Unified Memory profiling result:
```

```
Device "Tesla V100-PCIE-16GB (0)"
```

Count	Avg	Size	Min Size	Max Size	Total Size	Total Time	Name
3012	21.758KB	4.0000KB	952.00KB	64.00000MB	13.49043ms	Host To Device	
81	-	-	-	-	-	23.23181ms	Gpu page fault groups

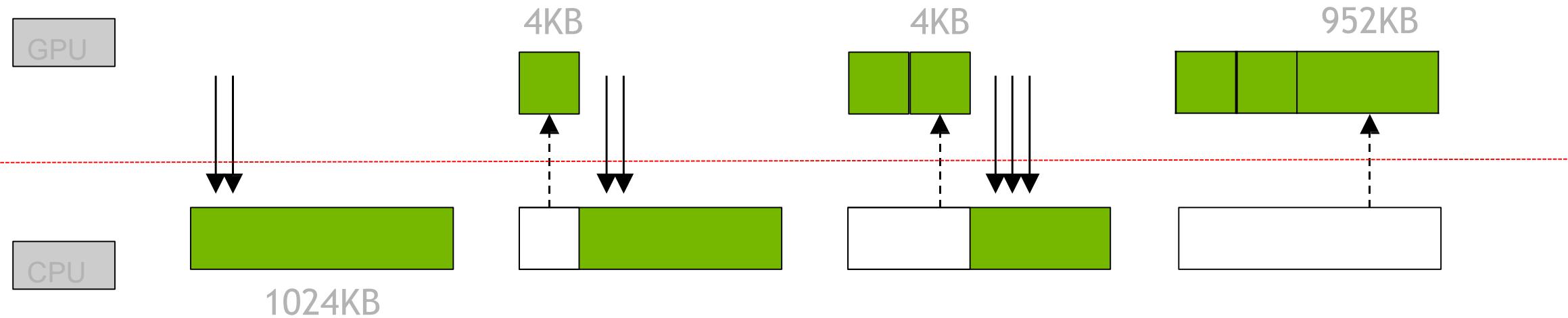
# HEURISTIC PREFETCHING

Do Not Confuse with API-prefetching

GPU architecture supports different page sizes

Contiguous pages up to a large page size are promoted to the larger size

Driver prefetches whole regions if pages are accessed *densely*



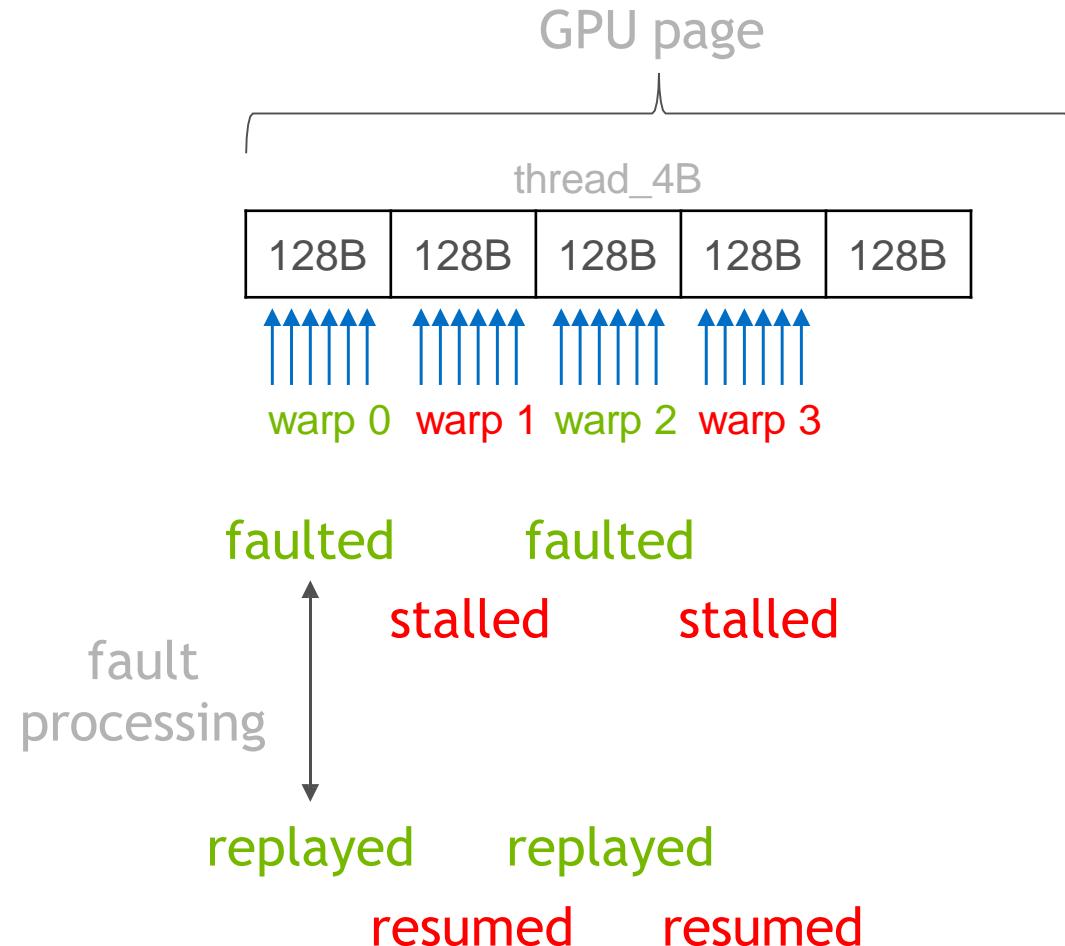
# WHAT IS PAGE FAULT GROUPS?

```
==14487== Unified Memory profiling result:  
Device "Tesla V100-PCIE-16GB (0)"  
Count Avg Size Min Size Max Size Total Size Total Time Name  
3012 21.758KB 4.0000KB 952.00KB 64.00000MB 13.49043ms Host To Device  
81 - - - - 23.23181ms Gpu page fault groups
```

nvprof --print-gpu-trace ...

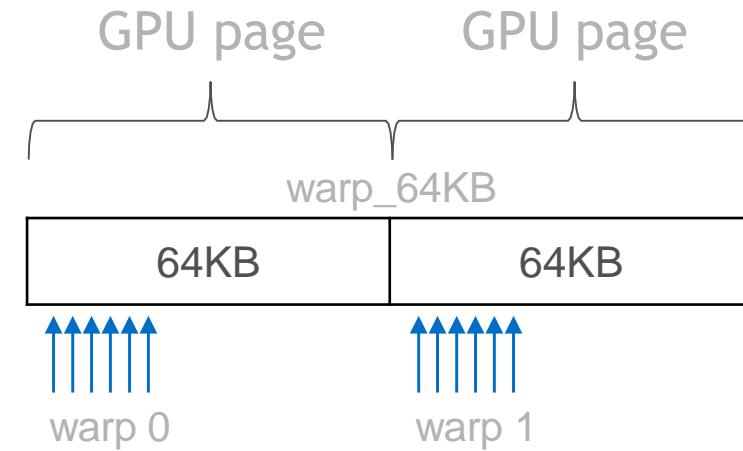
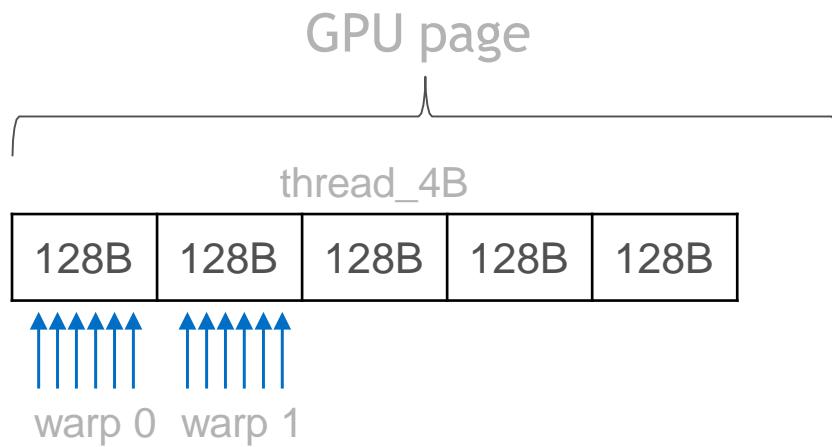
Unified Memory	Virtual Address	Name
8	0x3900010000	[Unified Memory GPU page faults]
9	0x3900040000	[Unified Memory GPU page faults]
5	0x3900108000	[Unified Memory GPU page faults]
5	0x3900200000	[Unified Memory GPU page faults]

# PAGE FAULTS HANDLING



# OPTIMIZING ON-DEMAND MIGRATION

Increase fault concurrency to reduce page fault stalls



# OPTIMIZING ON-DEMAND MIGRATION

Thread/4B

Count	Avg Size	Min Size	Max Size	Total Size	Total Time	Name
3012	21.758KB	4.0000KB	952.00KB	64.00000MB	13.49043ms	Host To Device
81	-	-	-	-	-	Gpu page fault groups
Unified Memory			Virtual Address	Name		
8			0x3900010000	[Unified Memory GPU page faults]		
9			0x3900040000	[Unified Memory GPU page faults]		
5			0x3900108000	[Unified Memory GPU page faults]		

Warp/64KB

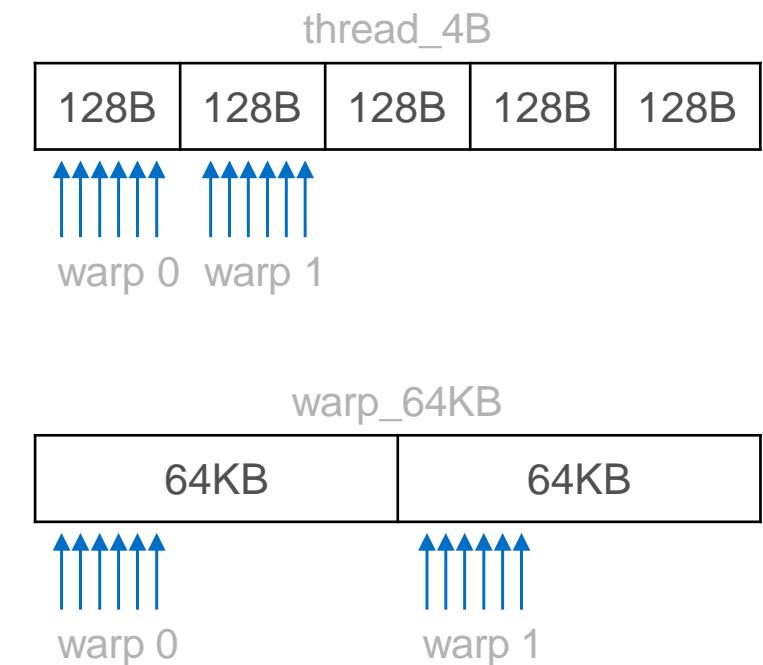
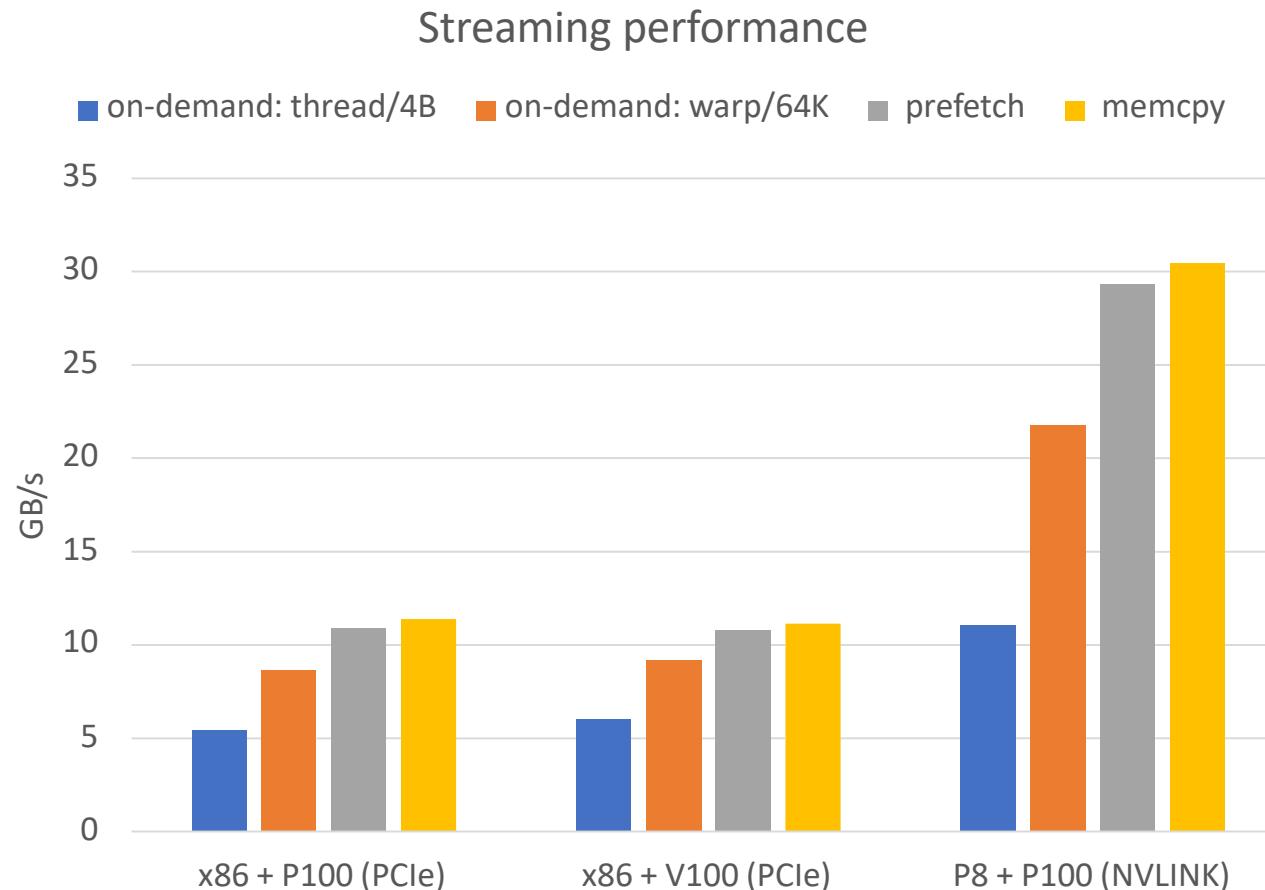
Count	Avg Size	Min Size	Max Size	Total Size	Total Time	Name
957	68.481KB	4.0000KB	576.00KB	64.00000MB	8.242080ms	Host To Device
6	-	-	-	-	-	Gpu page fault groups
Unified Memory			Virtual Address	Name		
1			0x39000d0000	[Unified Memory GPU page faults]		
1			0x39000c0000	[Unified Memory GPU page faults]		
1			0x3900080000	[Unified Memory GPU page faults]		

more efficient prefetching

fewer stalls

# STREAMING BENCHMARK

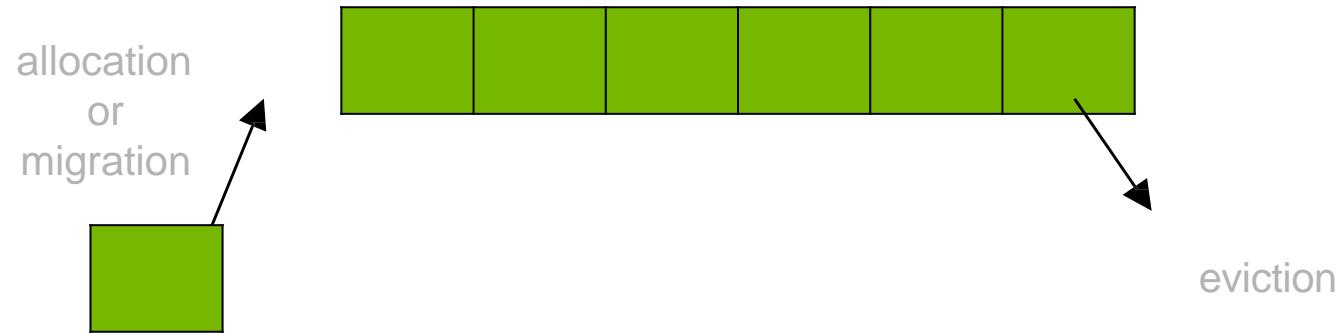
## How fast is on-demand migration?



Also check the Parallel Forall blog: <https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/>

# EVICTION ALGORITHM

## What Pages Are Moving Out of the GPU



Driver keeps a list of physical chunks of GPU memory

Chunks from the front of the list are evicted first (LRU)

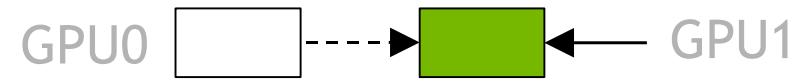
A chunk is considered “in use” when it is fully-populated or migrated

Prefetching and policies may impact eviction heuristic in the future

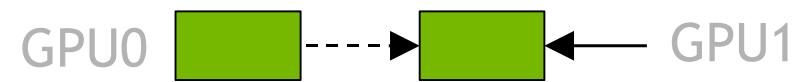
# LOCALITY AND ACCESS CONTROL

## cudaMemAdvise

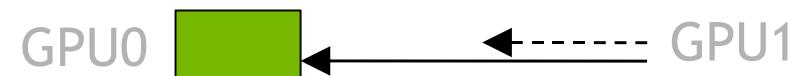
**Default:** data *migrates* on first touch



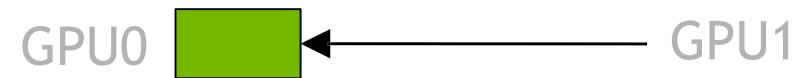
**ReadMostly:** data *duplicated* on first touch



**PreferredLocation:** *resist* migrating away from the preferred location



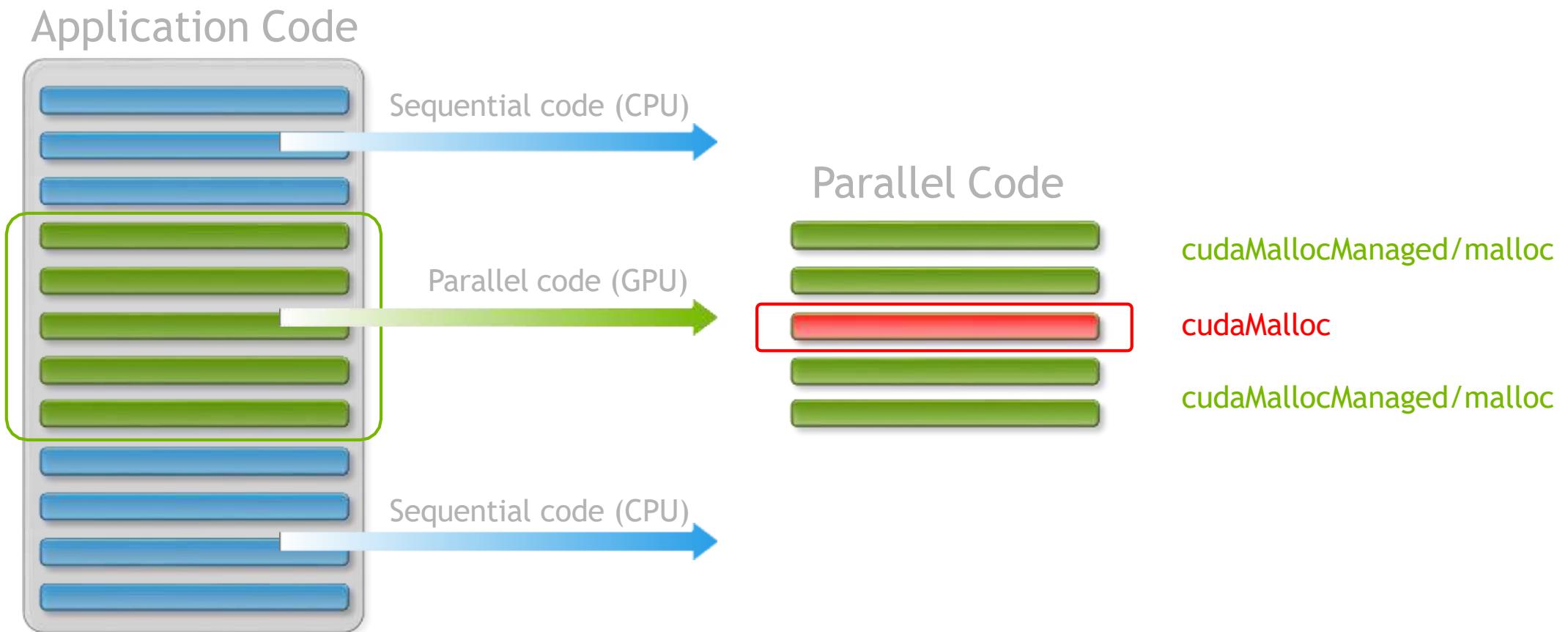
**AccessedBy:** establish *direct mapping* and avoid faults



# WHEN TO USE UNIFIED MEMORY

	<code>cudaMalloc</code>	<code>cudaMallocManaged</code>
Pinned allocation	<code>cudaMalloc</code>	<code>cudaMallocManaged</code> <code>PreferredLocation(GPU)</code> <code>SetAccessedBy(peer GPUs)</code> <code>cudaMemPrefetchAsync(GPU)</code>
<code>cudaMemcpy: ptrA -&gt; ptrB</code>	Staging for non-pinned allocations or between non-P2P GPUs	Staging or a copy kernel required in all cases
Memory migration	Not possible	<code>cudaMemPrefetchAsync</code>
Debugging	Difficult	Easy
Oversubscription	No	Yes
IPC support	Yes	No

# WHEN TO USE UNIFIED MEMORY



# UNIFIED MEMORY PLATFORMS

	KEPLER	PASCAL	VOLTA
Linux + x86	No GPU fault support No concurrent access	On-demand migration	On-demand migration
Linux + Power		On-demand migration 80GB/s CPU-GPU BW*	On-demand migration 150GB/s CPU-GPU BW** Access counters HW coherency ATS support
Windows		No GPU fault support No concurrent access	
MacOS		No GPU fault support No concurrent access	
Tegra		Cached on CPU and iGPU No concurrent access	

\*IBM Minsky: 4xP100 + 2xP8, 2xNVLINK1 links between P100 and P8, bi-directional aggregate BW

\*\*IBM Newell: 4xV100 + 2xP9, 3xNVLINK2 links between V100 and P9, bi-directional aggregate BW

# READ DUPLICATION

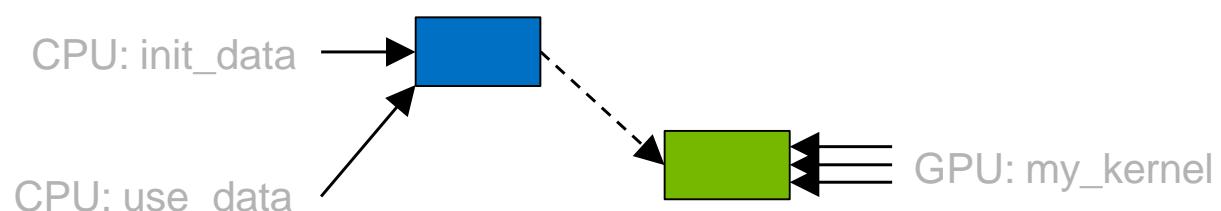
## Usage example

```
char *data;  
cudaMallocManaged(&data, N);  
  
init_data(data, N);  
  
cudaMemAdvise(data, N, ..SetReadMostly, myGpuId);  
cudaMemPrefetchAsync(data, N, myGpuId, s);  
mykernel<<<..., s>>>(data, N);  
  
use_data(data, N);  
cudaFree(data);
```

The prefetch creates a copy instead of moving data

Both processors can read data simultaneously without faults

Writes will collapse all copies into one, subsequent reads will fault and duplicate



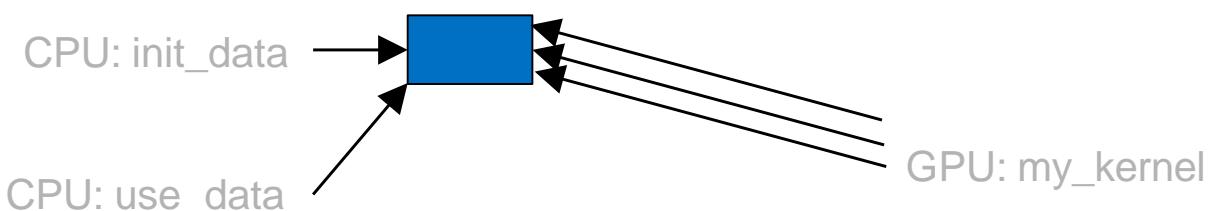
# PREFERRED LOCATION

## Resisting migrations

```
char *data;  
cudaMallocManaged(&data, N);  
  
init_data(data, N);  
  
cudaMemAdvise(data, N, ..PreferredLocation, cudaCpuDeviceId);  
  
mykernel<<<..., s>>>(data, N);  
  
  
use_data(data, N);  
  
cudaFree(data);
```

The kernel will *page fault* and generate direct mapping to data on the CPU

The driver will “resist” migrating data away from the preferred location



# PREFERRED LOCATION

## Page population on first-touch

```
char *data;  
cudaMallocManaged(&data, N);
```

The kernel will *page fault*,  
populate pages on the CPU  
and generate direct mapping to  
data on the CPU

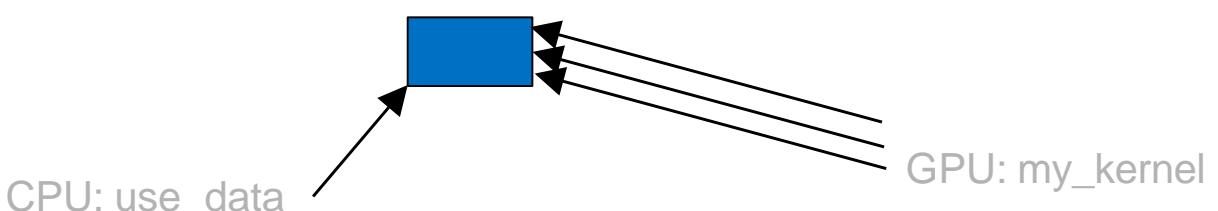
```
cudaMemAdvise(data, N, ..PreferredLocation, cudaCpuDeviceId);
```

Pages are populated on the  
preferred location if the  
faulting processor can access it

```
mykernel<<<..., s>>>(data, N);
```

```
use_data(data, N);
```

```
cudaFree(data);
```



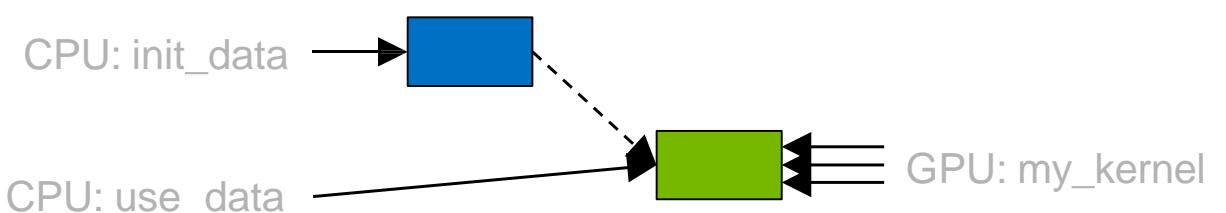
# PREFERRED LOCATION ON P9+V100

CPU can directly access GPU memory

```
char *data;  
cudaMallocManaged(&data, N);  
  
init_data(data, N);  
  
cudaMemAdvise(data, N, ..PreferredLocation, gpuId);  
  
mykernel<<<..., s>>>(data, N);  
  
use_data(data, N);  
cudaFree(data);
```

The kernel will *page fault* and migrate data to the GPU

CPU will fault and access data directly instead of migrating



on non P9+V100 systems the driver will migrate back to the CPU

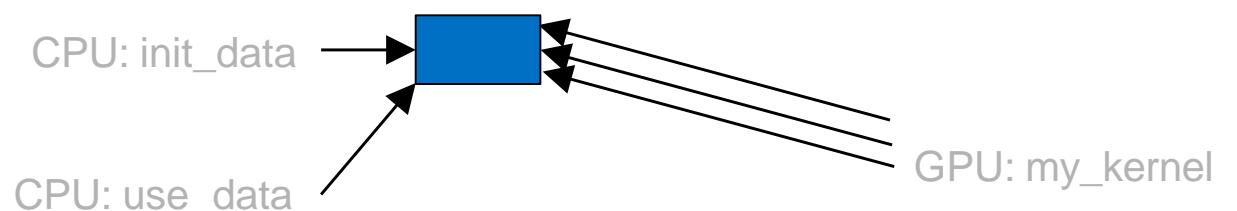
# ACCESSED BY

## Usage example

```
char *data;  
cudaMallocManaged(&data, N);  
  
init_data(data, N);  
  
cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);  
  
mykernel<<<..., s>>>(data, N);  
  
  
use_data(data, N);  
  
cudaFree(data);
```

GPU will establish direct mapping of data in CPU memory, no page faults will be generated

Memory can move freely to other processors and mapping will carry over



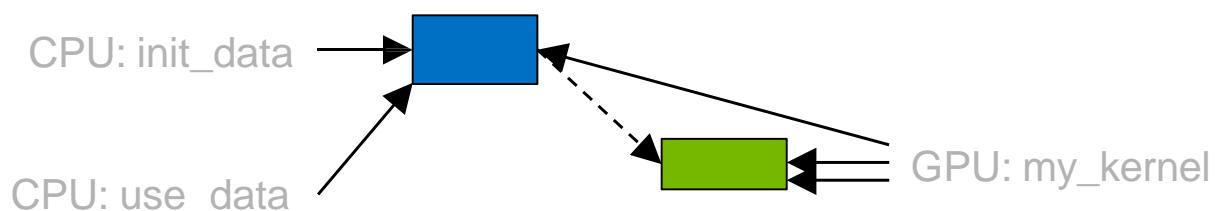
# ACCESSED BY

## Using access counters on Volta

```
char *data;  
cudaMallocManaged(&data, N);  
  
init_data(data, N);  
  
cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);  
  
mykernel<<<..., s>>>(data, N);  
  
  
use_data(data, N);  
  
cudaFree(data);
```

GPU will establish direct mapping of data in CPU memory, no page faults will be generated

Access counters may eventually trigger migration of **frequently accessed pages** to the GPU



# MANAGED VS MALLOC ON VOLTA+P9

## First touch allocation policy

```
ptr = cudaMallocManaged(size);  
doStuffOnGpu<<<...>>>(ptr, size);
```

↑  
GPU page faults

Unified Memory driver allocates on GPU  
GPU accesses GPU memory

```
ptr = malloc(size);  
doStuffOnGpu<<<...>>>(ptr, size);
```

↑  
GPU uses ATS, faults  
OS allocates on CPU (by default)  
GPU uses ATS to access CPU memory

\*You may alter this behavior by using `cudaMemAdvise` policies

# MANAGED VS MALLOC ON P9

cudaMallocManaged: same behavior as x86

```
ptr = cudaMallocManaged(size);  
  
fillData(ptr, size);  
  
doStuffOnGpu<<<...>>>(ptr, size); ← GPU page faults  
ptr migrated to GPU  
  
cudaDeviceSynchronize();  
  
doStuffOnCpu(ptr, size); ← CPU page faults  
ptr migrated to CPU
```

# MANAGED VS MALLOC ON P9

malloc: no on-demand migrations\*

```
ptr = malloc(size);  
  
fillData(ptr, size);  
  
doStuffOnGpu<<<...>>>(ptr, size); ←  
  
cudaDeviceSynchronize();  
  
doStuffOnCpu(ptr, size); ←
```

GPU uses ATS to  
access CPU memory  
**(no on-demand migration  
except cudaMemPrefetchAsync\*)**

CPU accesses  
CPU memory

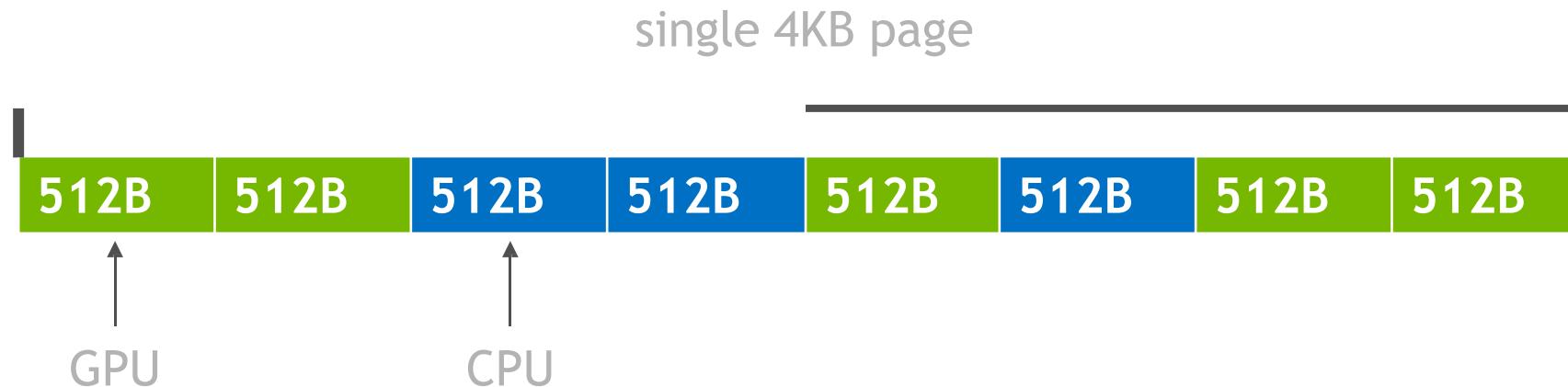
\*In the future Volta access counters will be used to migrate malloc memory

# HYPRE-INSPIRED USE CASE

Algebraic Multi-Grid library: <https://github.com/LLNL/hypre>

Lots of small allocations: multiple variables may end up on the same page

If used by different processors this will result in **false-sharing**



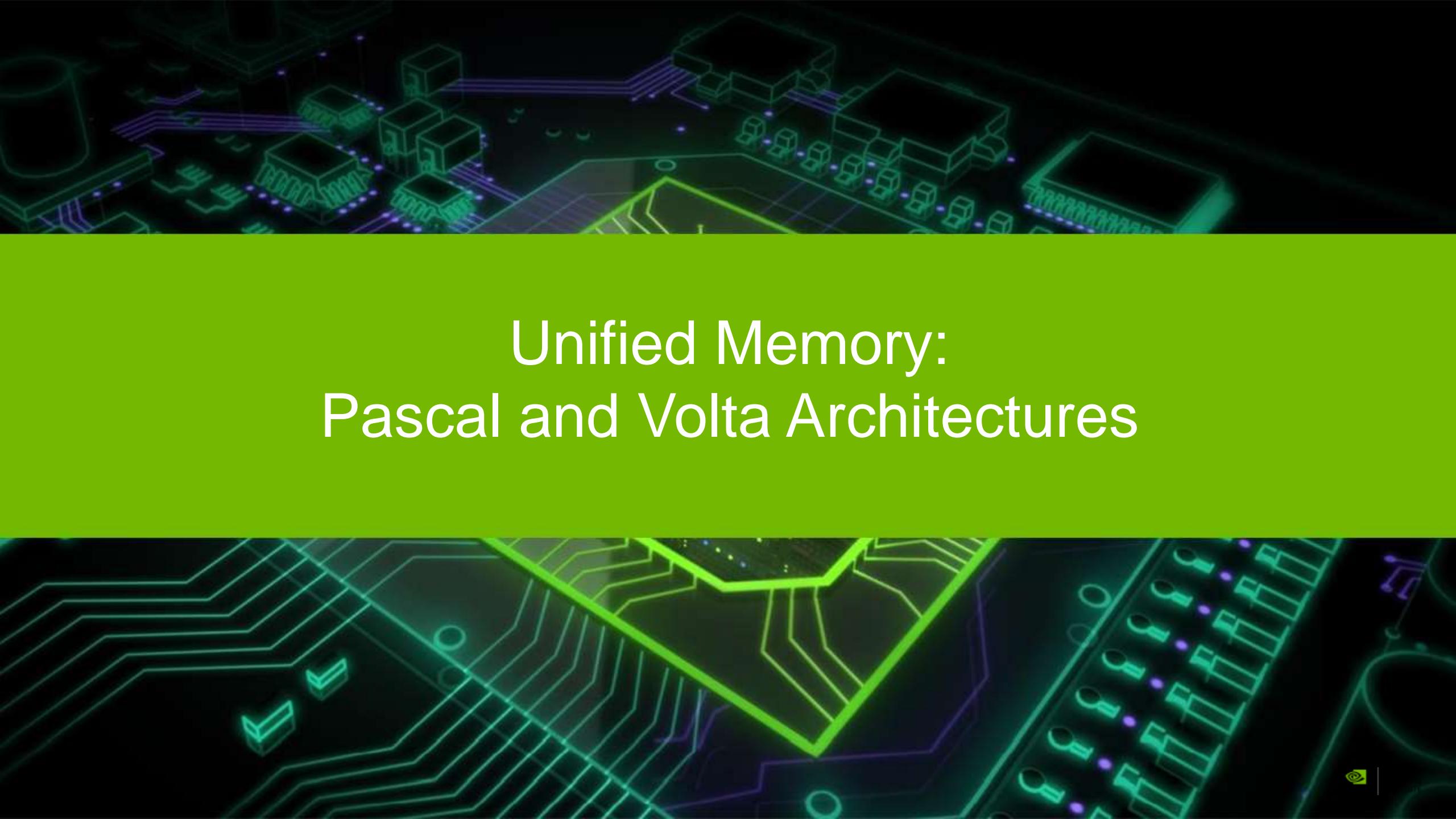
# FALSE-SHARING

**Issues with false-sharing:**

- Spurious migrations, thrashing mitigation does not solve it
- Performance hints are applied on page boundaries, due to suballocation data may inherit the wrong policies

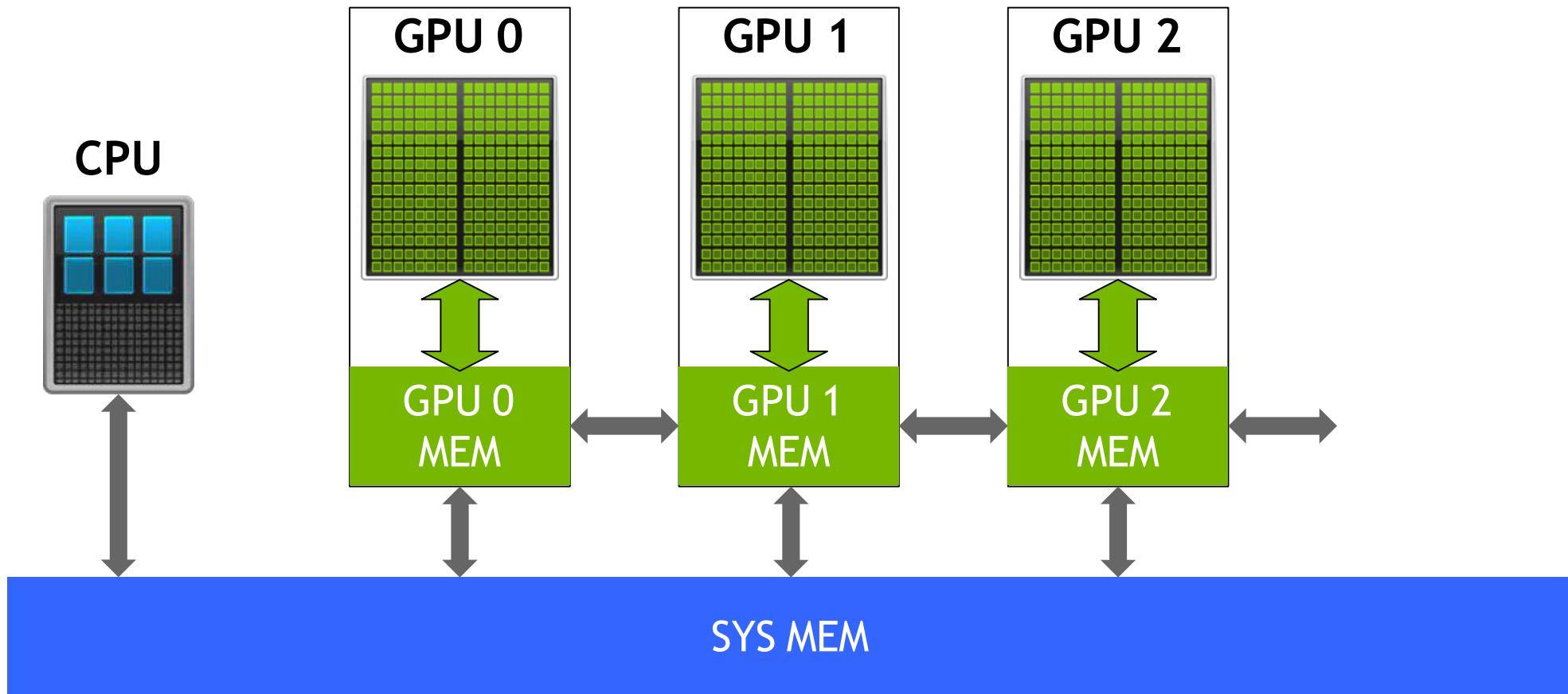
**How to mitigate this:**

- Use separate allocators or memory pools for CPU and GPU



# Unified Memory: Pascal and Volta Architectures

# HETEROGENEOUS ARCHITECTURES



# UNIFIED MEMORY FUNDAMENTALS

## Single Pointer

CPU code

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
cpu_func2(data, N);  
  
cpu_func3(data, N);  
  
free(data);
```

GPU code

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
gpu_func2<<<...>>>(data, N);  
cudaDeviceSynchronize();  
  
cpu_func3(data, N);  
  
free(data);
```

# UNIFIED MEMORY FUNDAMENTALS

## Single Pointer

### Explicit Memory Management

```
void *h_data, *d_data;  
h_data = malloc(N);  
cudaMalloc(&d_data, N);  
cpu_func1(h_data, N);  
cudaMemcpy(d_data, h_data, N, ...)  
gpu_func2<<<...>>>(data, N);  
  
cudaMemcpy(h_data, d_data, N, ...)  
cpu_func3(h_data, N);  
  
free(h_data);  
cudaFree(d_data);
```

### Unified Memory

```
void *data;  
data = malloc(N);  
  
cpu_func1(data, N);  
  
gpu_func2<<<...>>>(data, N);  
cudaDeviceSynchronize();  
  
cpu_func3(data, N);  
  
free(data);
```

# UNIFIED MEMORY FUNDAMENTALS

## Deep Copy Nightmare

### Explicit Memory Management

```
char **data;
data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++)
    data[i] = (char*)malloc(N);

char **d_data;
char **h_data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++) {
    cudaMalloc(&h_data2[i], N);
    cudaMemcpy(h_data2[i], h_data[i], N, ...);
}
cudaMalloc(&d_data, N*sizeof(char*));
cudaMemcpy(d_data, h_data2, N*sizeof(char*), ...);

gpu_func<<<...>>>(data, N);
```

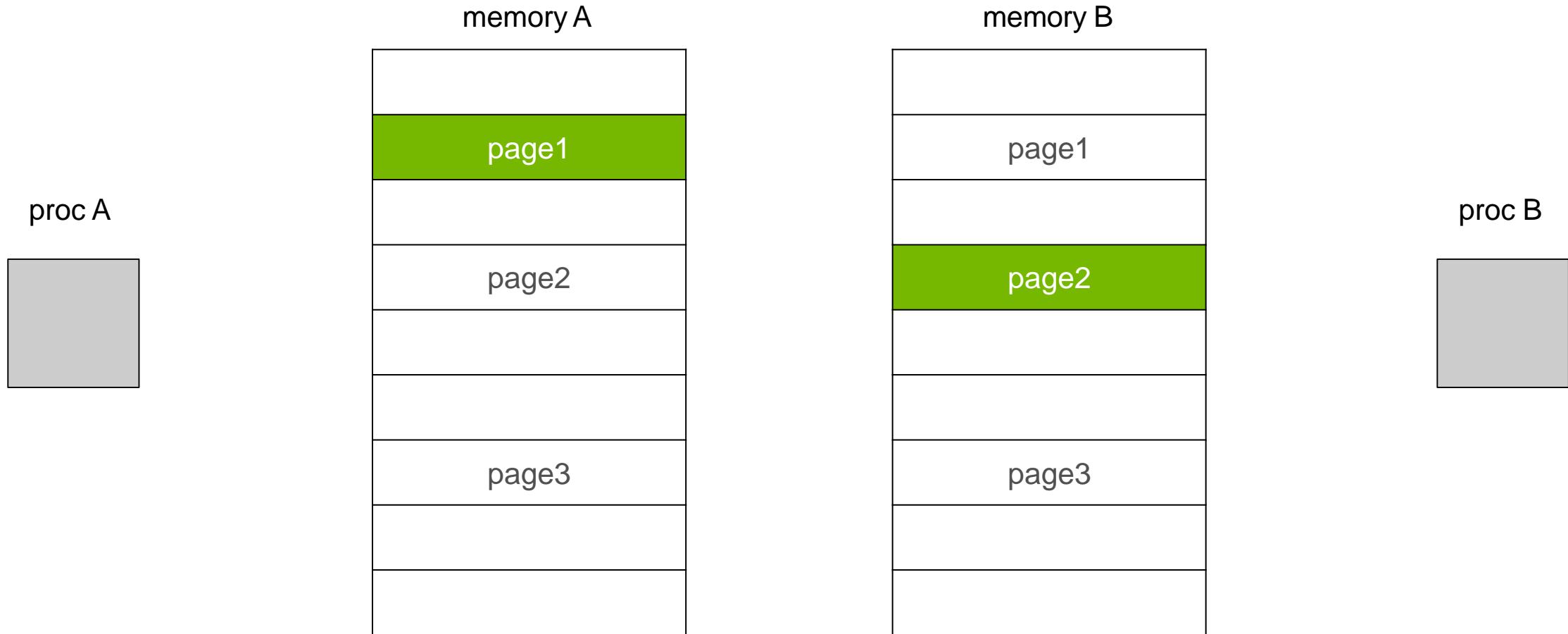
### Unified Memory

```
char **data;
data = (char**)malloc(N*sizeof(char*));
for (int i = 0; i < N; i++)
    data[i] = (char*)malloc(N);

gpu_func<<<...>>>(data, N);
```

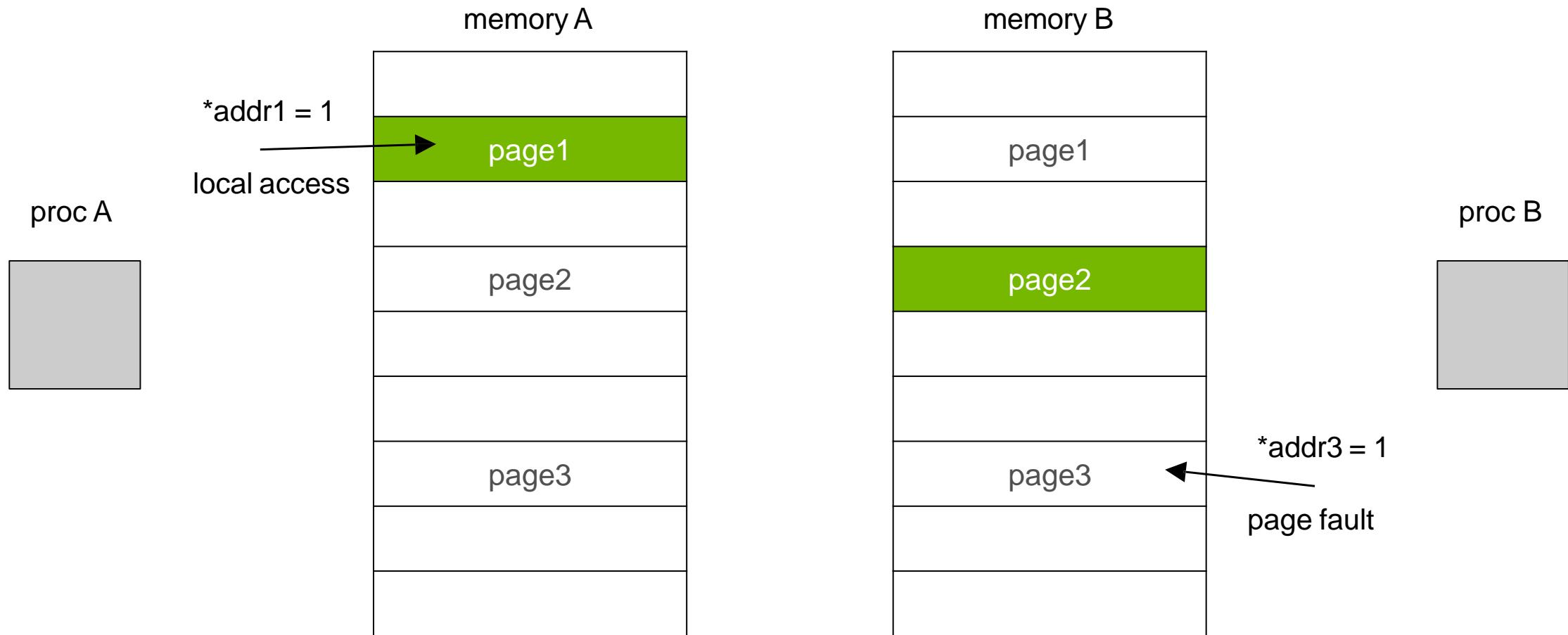
# UNIFIED MEMORY FUNDAMENTALS

## On-Demand Migration



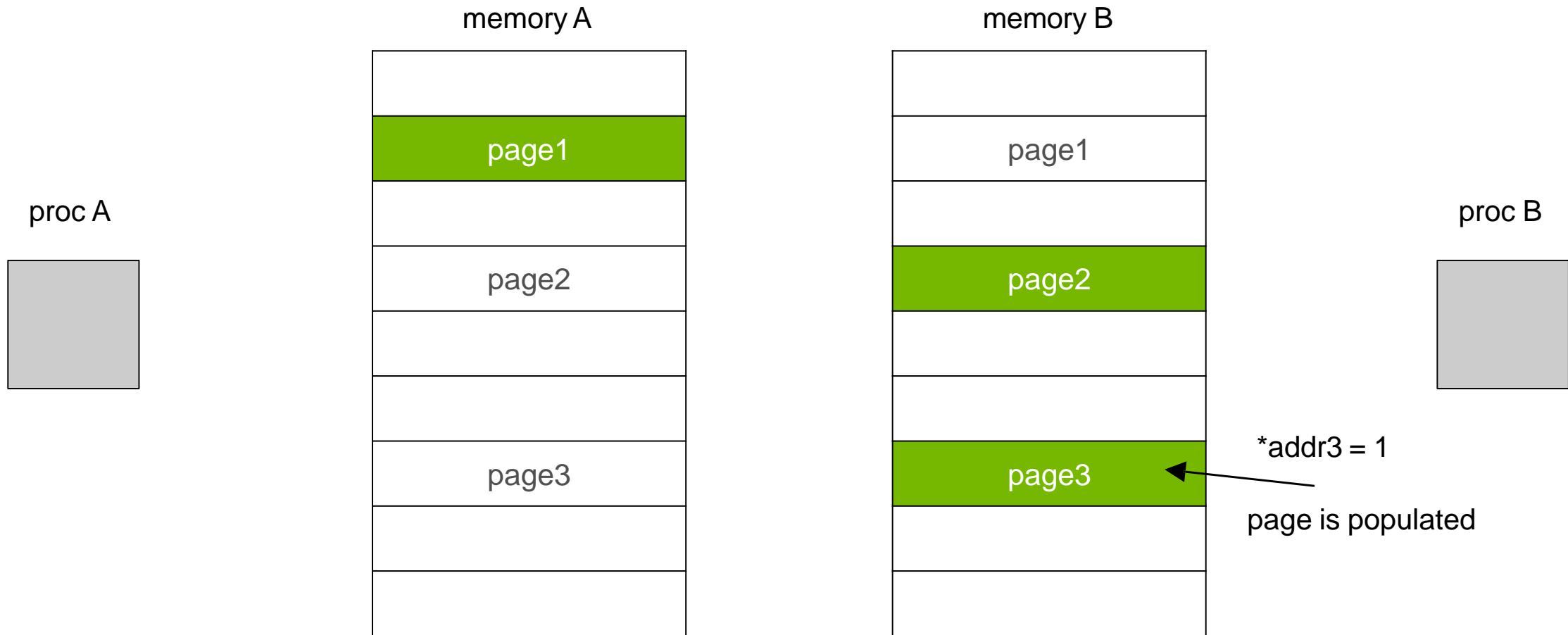
# UNIFIED MEMORY FUNDAMENTALS

## On-Demand Migration



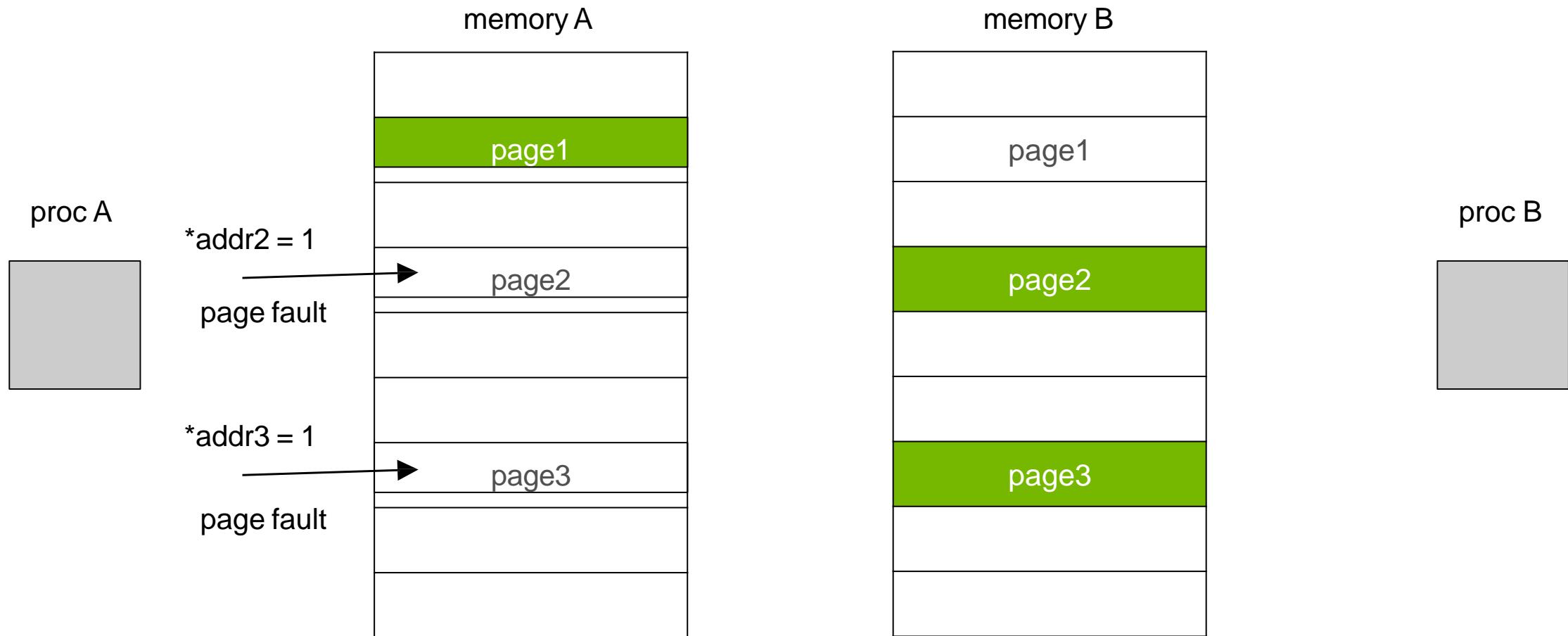
# UNIFIED MEMORY FUNDAMENTALS

## On-Demand Migration



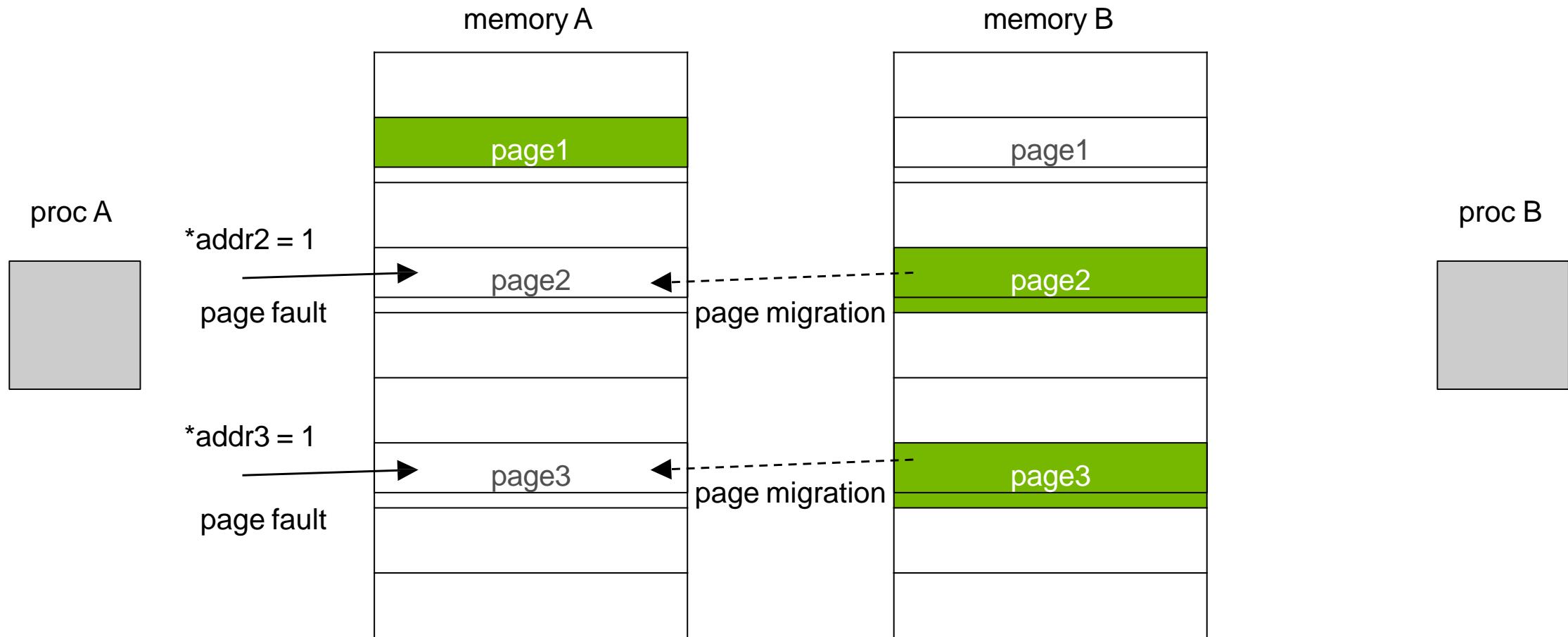
# UNIFIED MEMORY FUNDAMENTALS

## On-Demand Migration



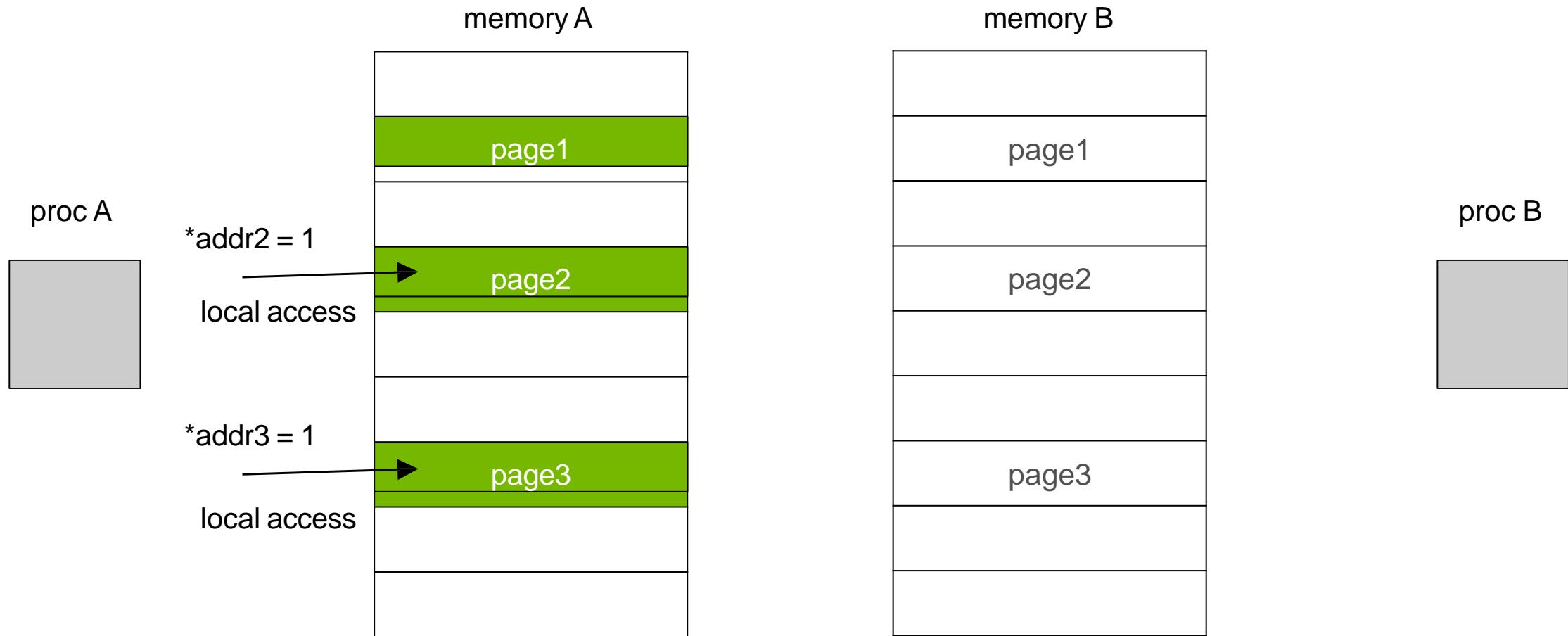
# UNIFIED MEMORY FUNDAMENTALS

## On-Demand Migration



# UNIFIED MEMORY FUNDAMENTALS

## On-Demand Migration



# UNIFIED MEMORY FUNDAMENTALS

## When Is This Helpful?

When it doesn't matter *how* data moves to a processor

- 1) Quick and dirty algorithm prototyping
- 2) Iterative process with lots of data reuse, migration cost can be amortized
- 3) Simplify application debugging

When it's difficult to isolate the working set

- 1) Irregular or *dynamic* data structures, unpredictable access
- 2) Data partitioning between multiple processors

# UNIFIED MEMORY FUNDAMENTALS

## Memory Oversubscription Benefits

When you have **large** dataset and not enough physical memory

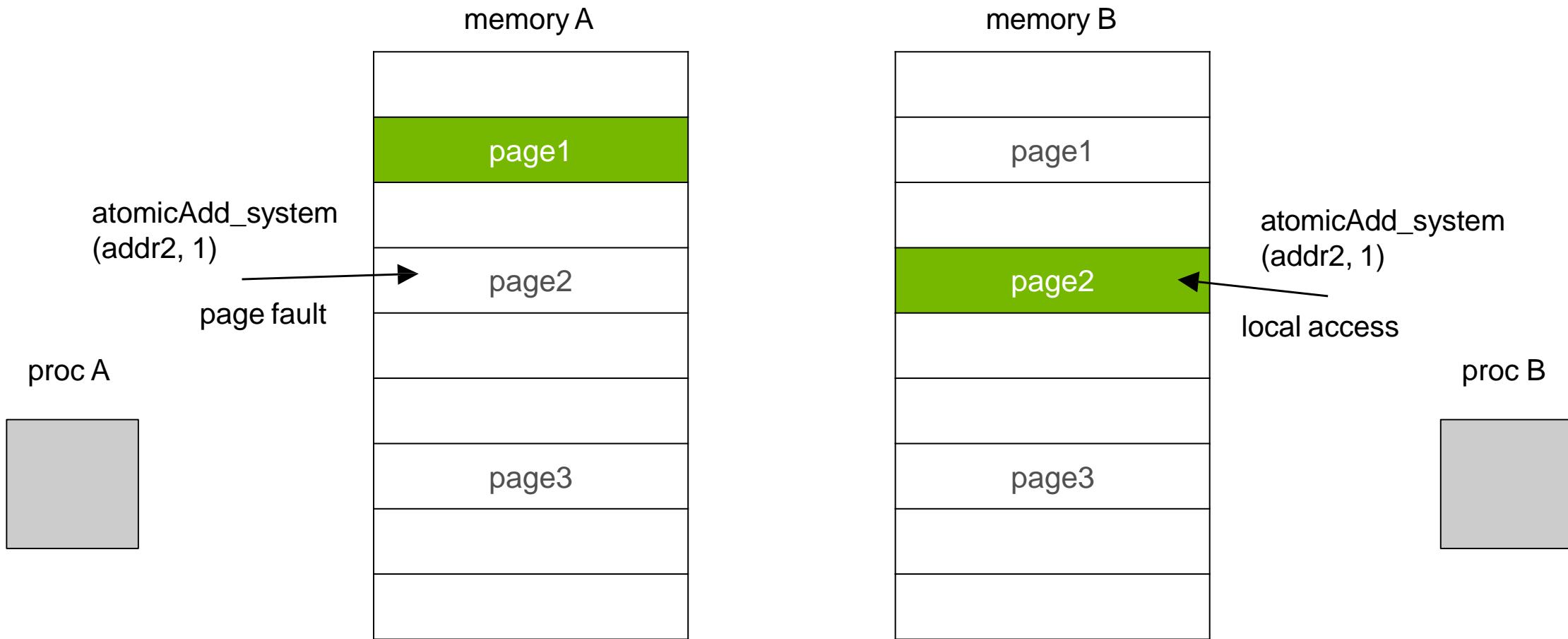
Moving pieces by hand is error-prone and requires tuning for memory size

Better to run slowly than get fail with out-of-memory error

You can actually get **high performance** with Unified Memory!

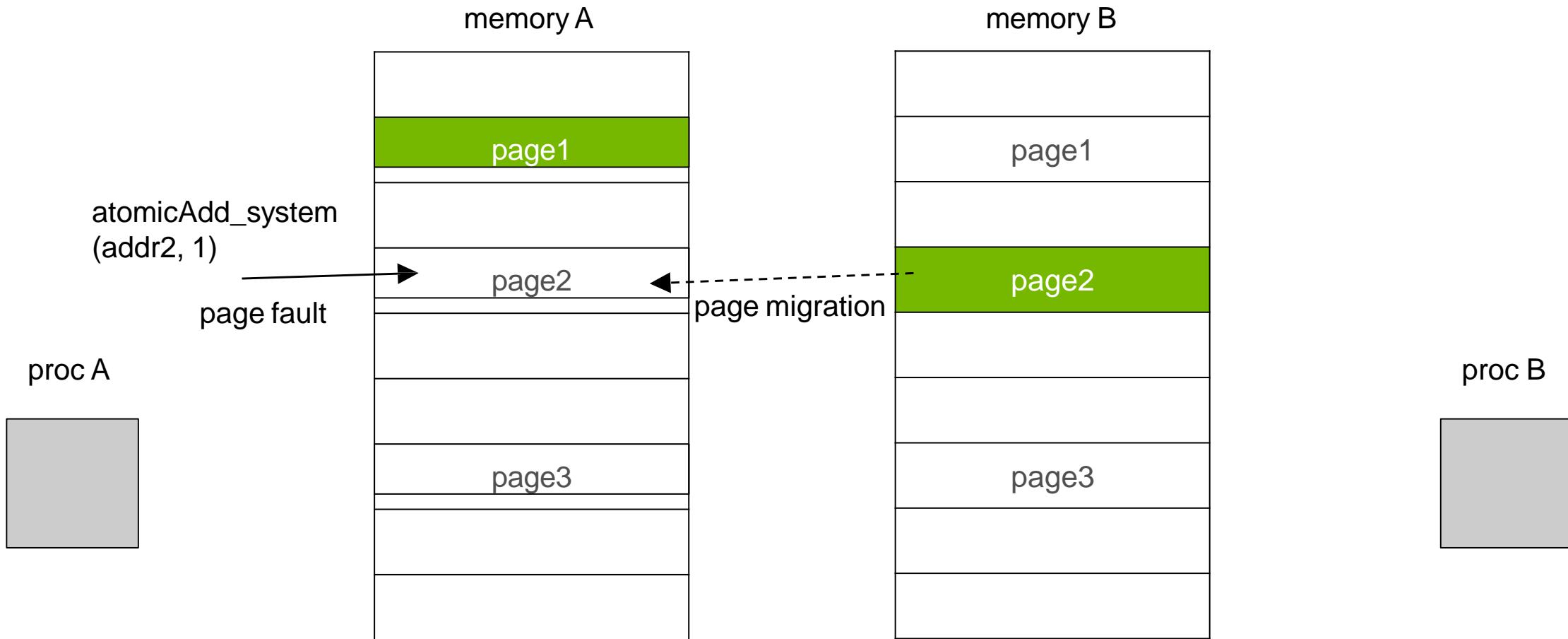
# UNIFIED MEMORY FUNDAMENTALS

## System-Wide Atomics with Exclusive Access



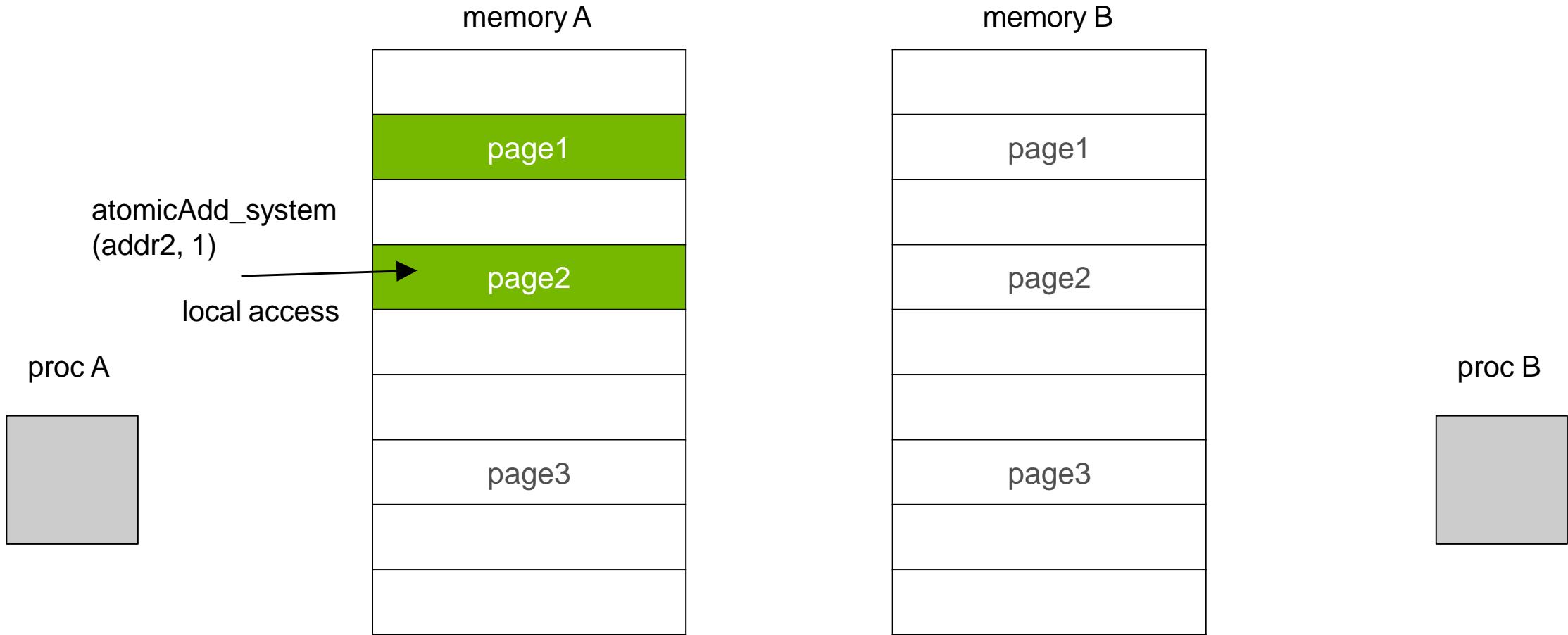
# UNIFIED MEMORY FUNDAMENTALS

## System-Wide Atomics with Exclusive Access



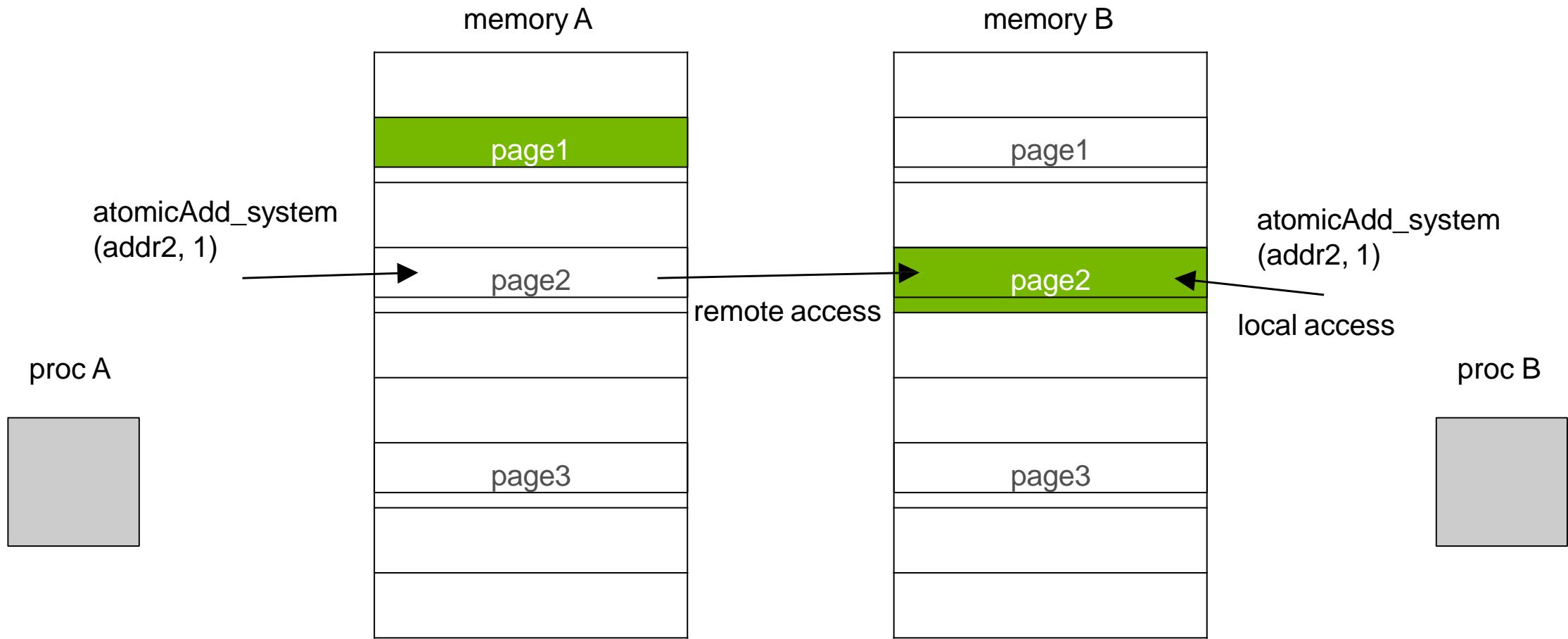
# UNIFIED MEMORY FUNDAMENTALS

## System-Wide Atomics with Exclusive Access



# UNIFIED MEMORY FUNDAMENTALS

## System-Wide Atomics over NVLINK\*



\*both processors need to support atomic operations

# UNIFIED MEMORY FUNDAMENTALS

## System-Wide Atomics

GPUs are very good at handling atomics from *thousands of threads*

Makes sense to utilize atomics between GPUs or between CPU and GPU

We will see this in action on a realistic example later on

# AGENDA

Unified Memory Fundamentals  
Under the Hood Details  
Performance Analysis and Optimizations  
Applications Deep Dive

# UNIFIED MEMORY ALLOCATOR

## Available Options

CUDA C: **cudaMallocManaged** is your most reliable way to opt in today

CUDA Fortran: **managed** attribute (per allocation)

OpenACC: **-ta=managed** compiler option (all dynamic allocations)

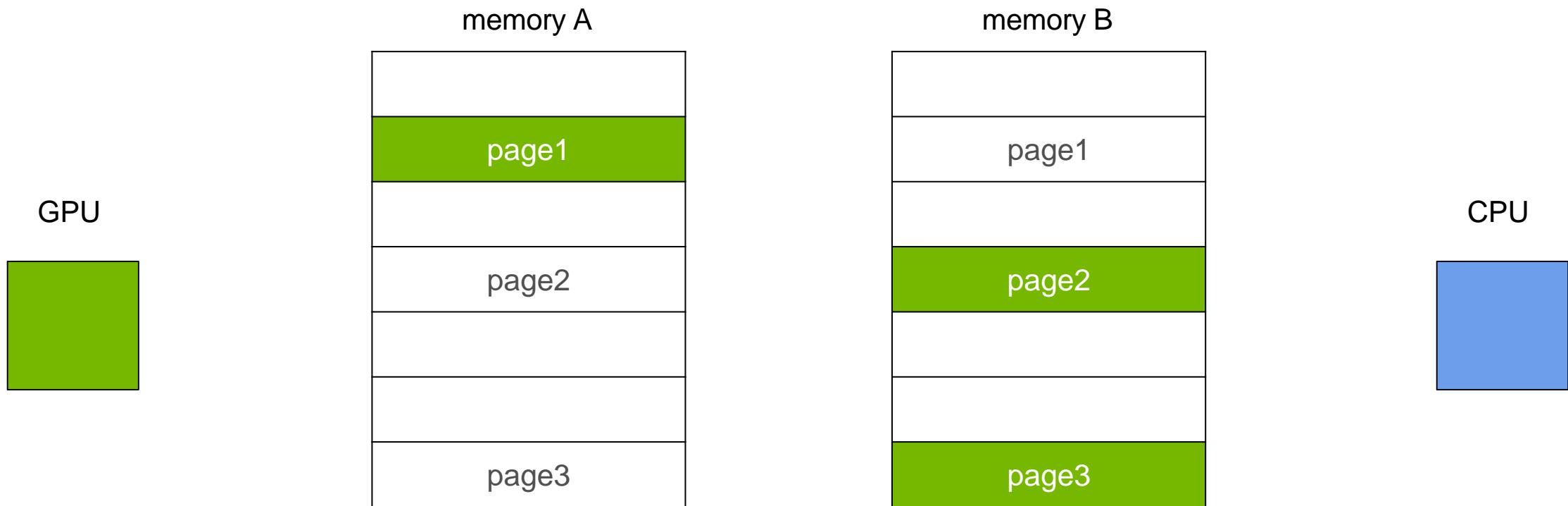
**malloc** support is coming on Pascal+ architectures (Linux only)

Note: you can write your own malloc hook to use **cudaMallocManaged**

# UNIFIED MEMORY ON KEPLER

Available since CUDA 6

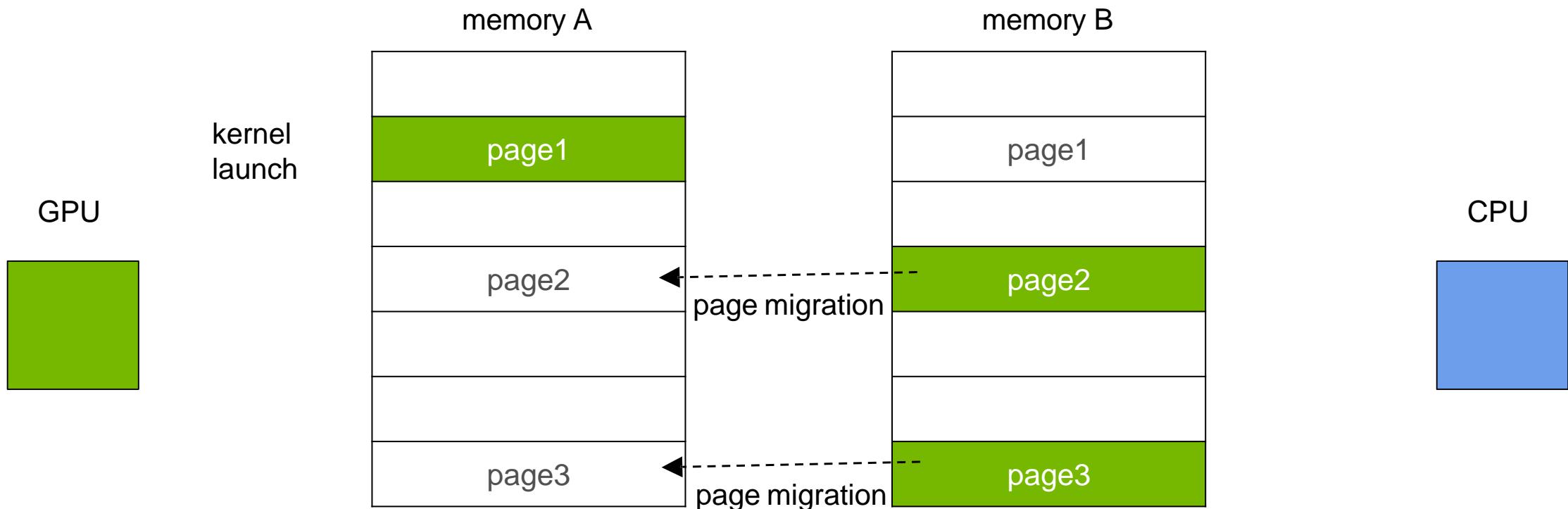
Kepler GPU: no page fault support, limited virtual space



# UNIFIED MEMORY ON KEPLER

Available since CUDA 6

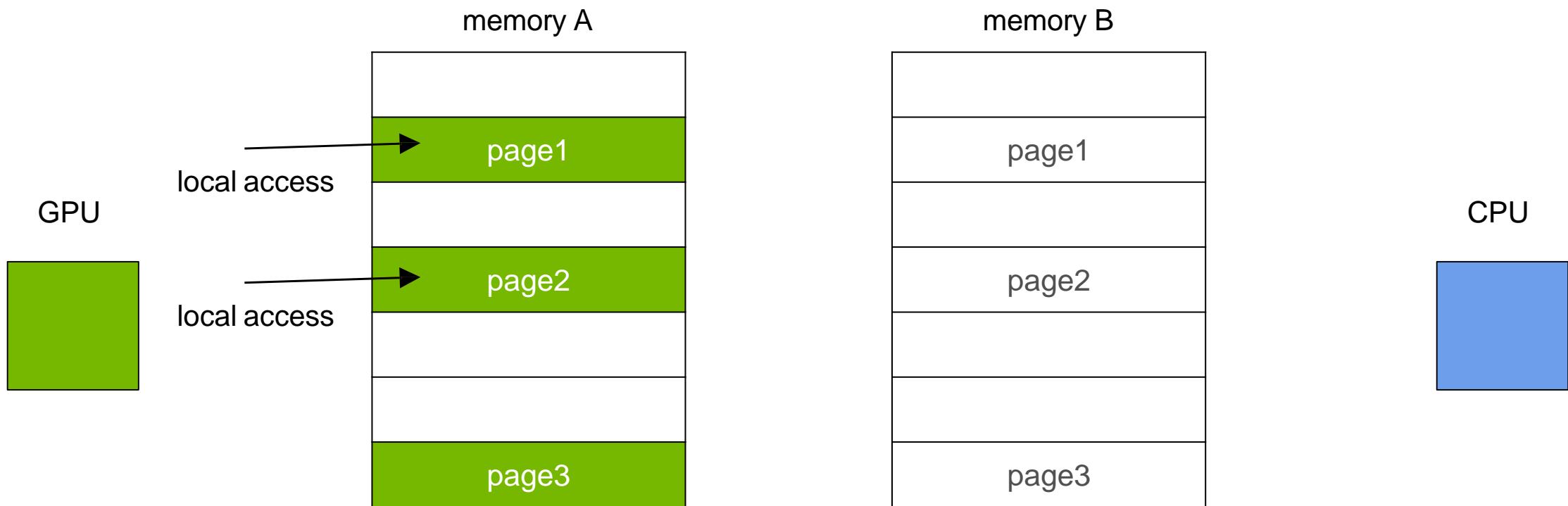
Bulk migration of **all pages** attached to current stream on kernel launch



# UNIFIED MEMORY ON KEPLER

Available since CUDA 6

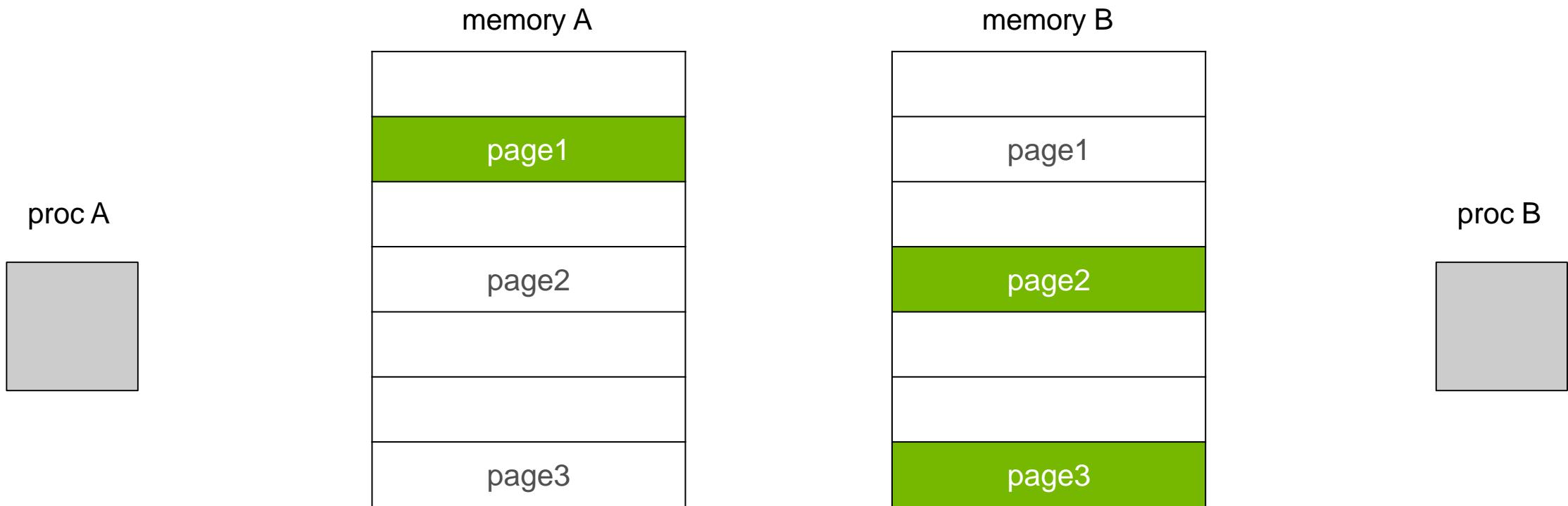
No on-demand migration for the GPU, no oversubscription, no system-wide atomics



# UNIFIED MEMORY ON PASCAL

Available since CUDA 8

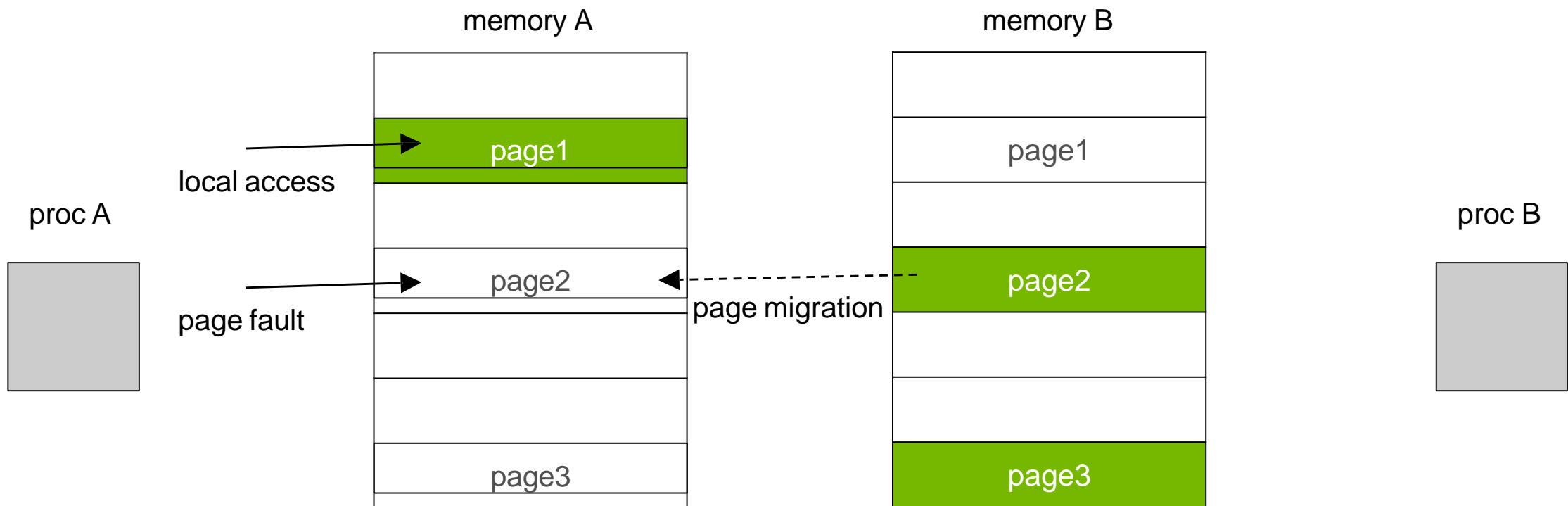
Pascal GPU: page fault support, extended virtual address space (48-bit)



# UNIFIED MEMORY ON PASCAL

Available since CUDA 8

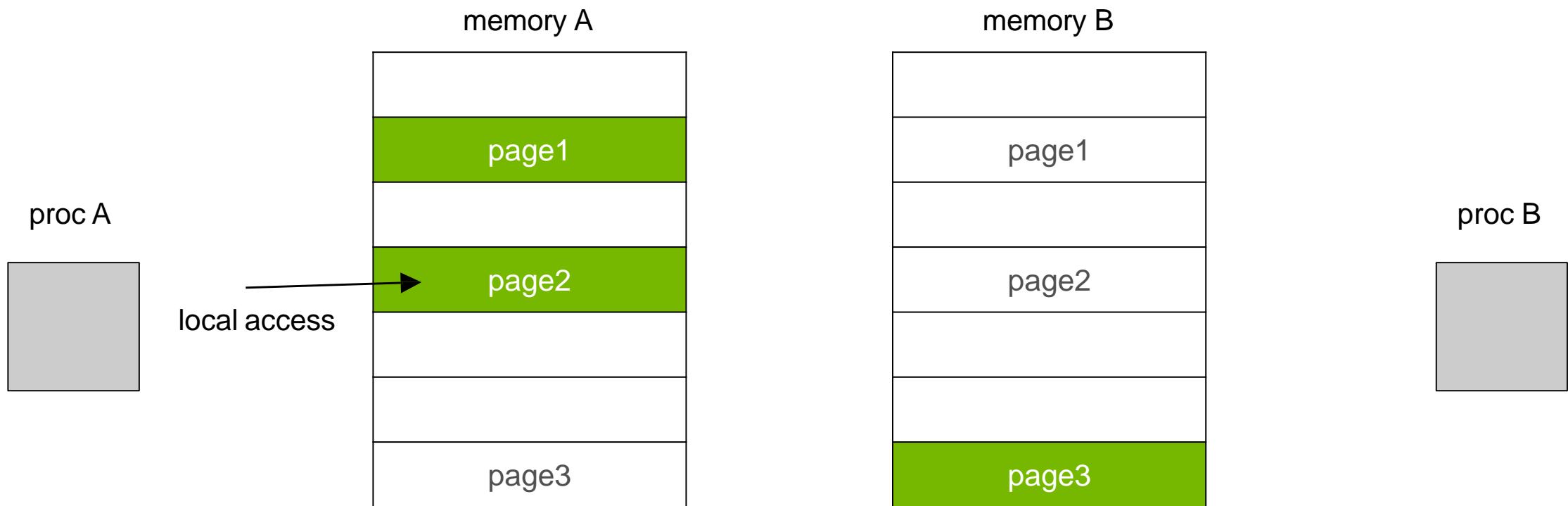
On-demand migration to accessing processor on first touch



# UNIFIED MEMORY ON PASCAL

Available since CUDA 8

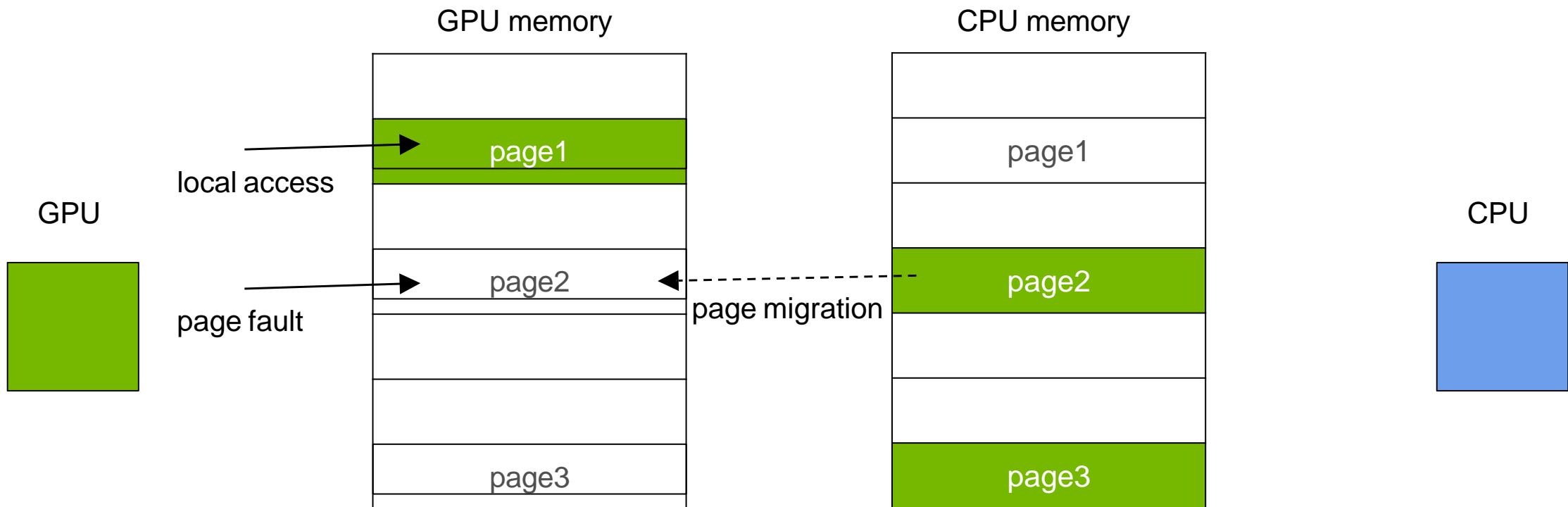
All features: on-demand migration, oversubscription, system-wide atomics



# UNIFIED MEMORY ON VOLTA

## Default model

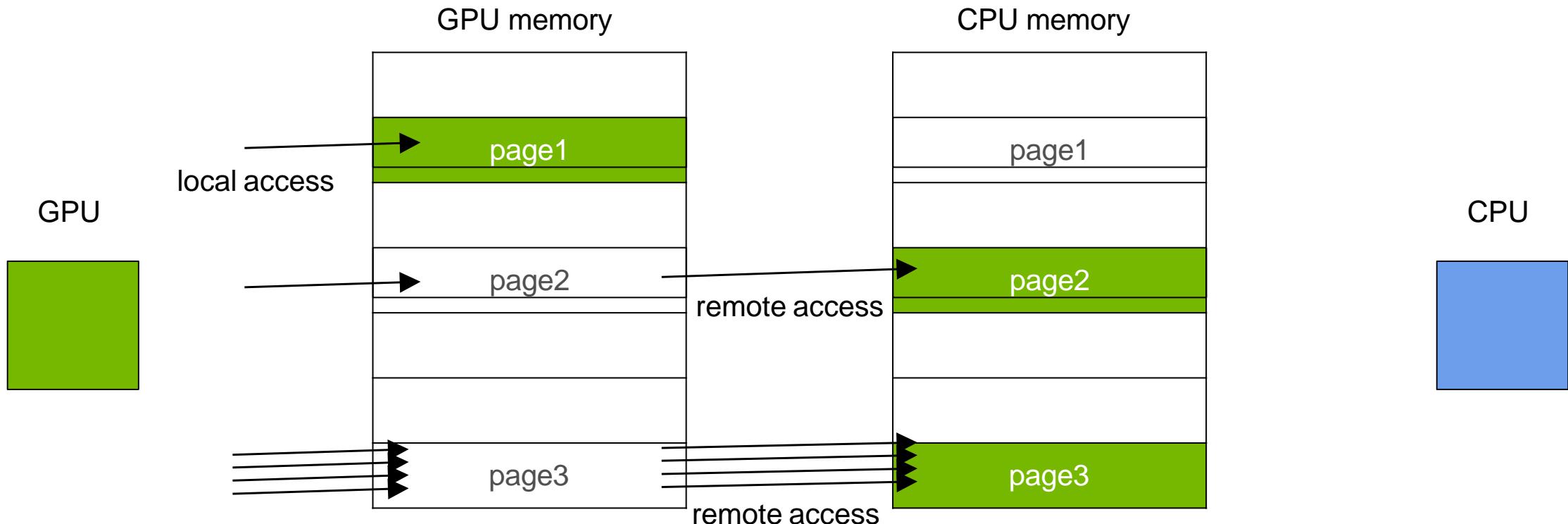
Volta GPU: uses fault on first touch for migration, same as Pascal



# UNIFIED MEMORY ON VOLTA

New Feature: Access Counters

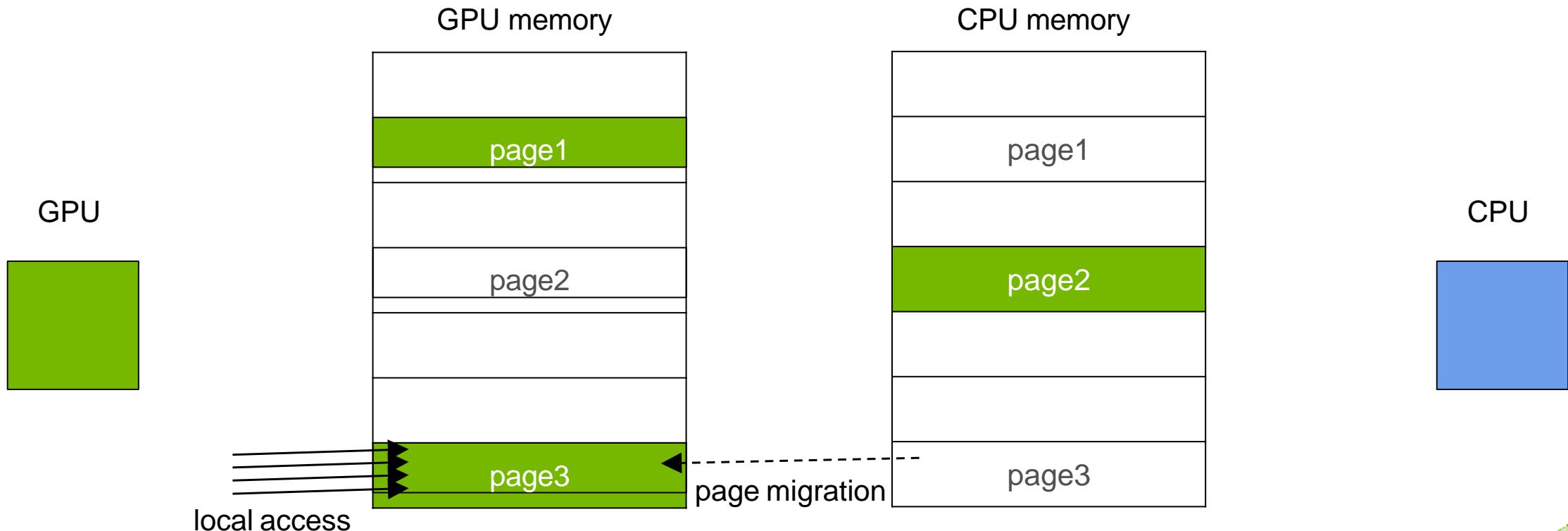
If memory is mapped to the GPU, migration can be triggered by access counters



# UNIFIED MEMORY ON VOLTA

New Feature: Access Counters

With access counters migration **only hot pages** will be moved to the GPU



# USER HINTS

## Why, When, and How to Use Them

If you know your application well you can optimize with hints

These are also useful to override some of the driver heuristics

**cudaMemPrefetchAsync(ptr, size, processor, stream)**

Similar to `move_pages()` in Linux

**cudaMemAdvise(ptr, size, advice, processor)**

Similar to `madvise()` in Linux

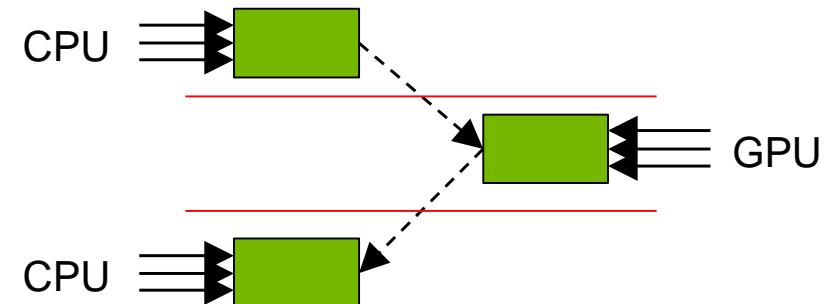
# USER HINTS

## Prefetching

```
char *data;  
cudaMallocManaged(&data, N);  
  
init_data(data, N);  
  
cudaMemPrefetchAsync(data, N, myGpuId, s);  
mykernel<<<..., s>>>(data, N);  
cudaMemPrefetchAsync(data, N, cudaCpuDeviceId, s);  
cudaStreamSynchronize(s);  
  
use_data(data, N);  
  
cudaFree(data);
```

Page faults can be expensive  
and they stall SM execution

Avoid faults by prefetching data  
to the accessing processor



# USER HINTS

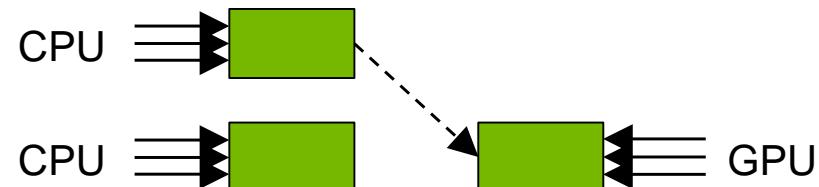
## Read Mostly

```
char *data;  
cudaMallocManaged(&data, N);  
  
init_data(data, N);  
  
cudaMemAdvise(data, N, ..SetReadMostly, myGpuId);  
cudaMemPrefetchAsync(data, N, myGpuId, s);  
mykernel<<<..., s>>>(data, N);  
  
use_data(data, N);  
  
cudaFree(data);
```

In this case prefetch creates a copy instead of moving data

Both processors can read data **simultaneously** without faults

Writes are allowed but they are expensive



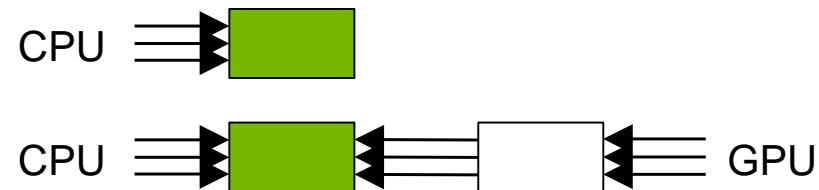
# USER HINTS

## Preferred Location

```
char *data;  
cudaMallocManaged(&data, N);  
  
init_data(data, N);  
  
cudaMemAdvise(data, N, ..PreferredLocation, cudaCpuDeviceId);  
  
mykernel<<<..., s>>>(data, N);  
  
use_data(data, N);  
  
cudaFree(data);
```

Here the kernel will *page fault* and generate direct mapping to data on the CPU

The driver will “resist” migrating data away from the preferred location



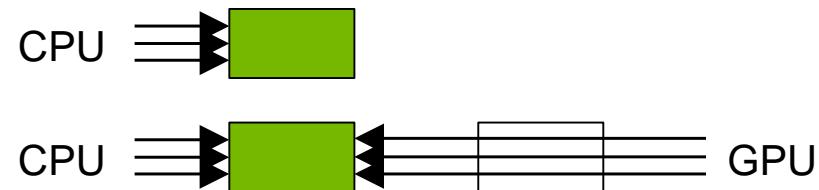
# USER HINTS

## Accessed By

```
char *data;  
cudaMallocManaged(&data, N);  
  
init_data(data, N);  
  
cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);  
  
mykernel<<<..., s>>>(data, N);  
  
  
use_data(data, N);  
  
cudaFree(data);
```

GPU will establish direct mapping of data in CPU memory, **no page faults** will be generated

Memory can move freely to other processors and mapping will carry over



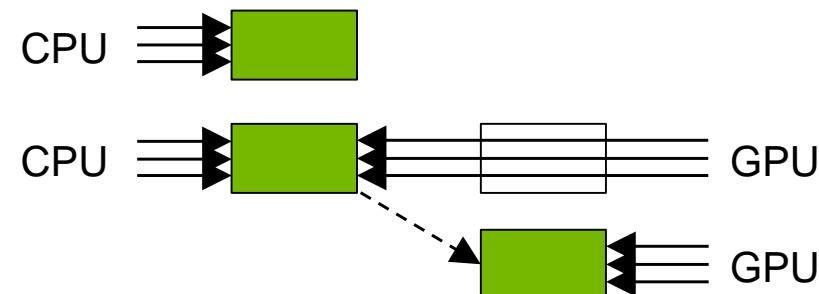
# USER HINTS

## Accessed By on Volta

```
char *data;  
cudaMallocManaged(&data, N);  
  
init_data(data, N);  
  
cudaMemAdvise(data, N, ..SetAccessedBy, myGpuId);  
  
mykernel<<<..., s>>>(data, N);  
  
  
use_data(data, N);  
  
cudaFree(data);
```

GPU will establish direct mapping of data in CPU memory, **no page faults** will be generated

Access counters may eventually trigger migration of this memory to the GPU



# CONCLUSIONS AND OUTLOOK

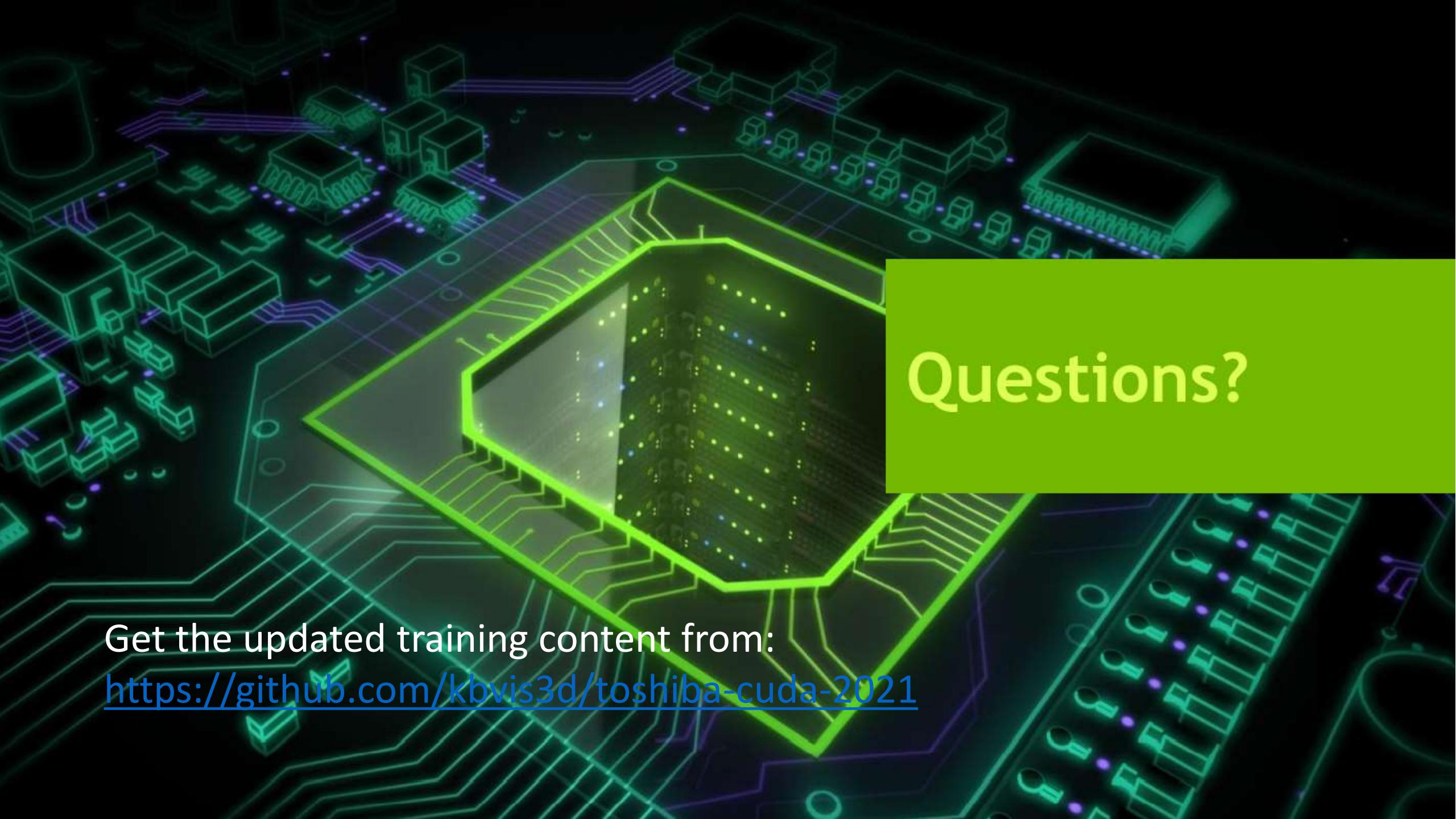
Consider using Unified Memory for any new application development

Get your code *running* on the GPU much sooner!

Enjoy clean code and *\*virtually\** no memory limits

Increase productivity, explore and prototype new algorithms

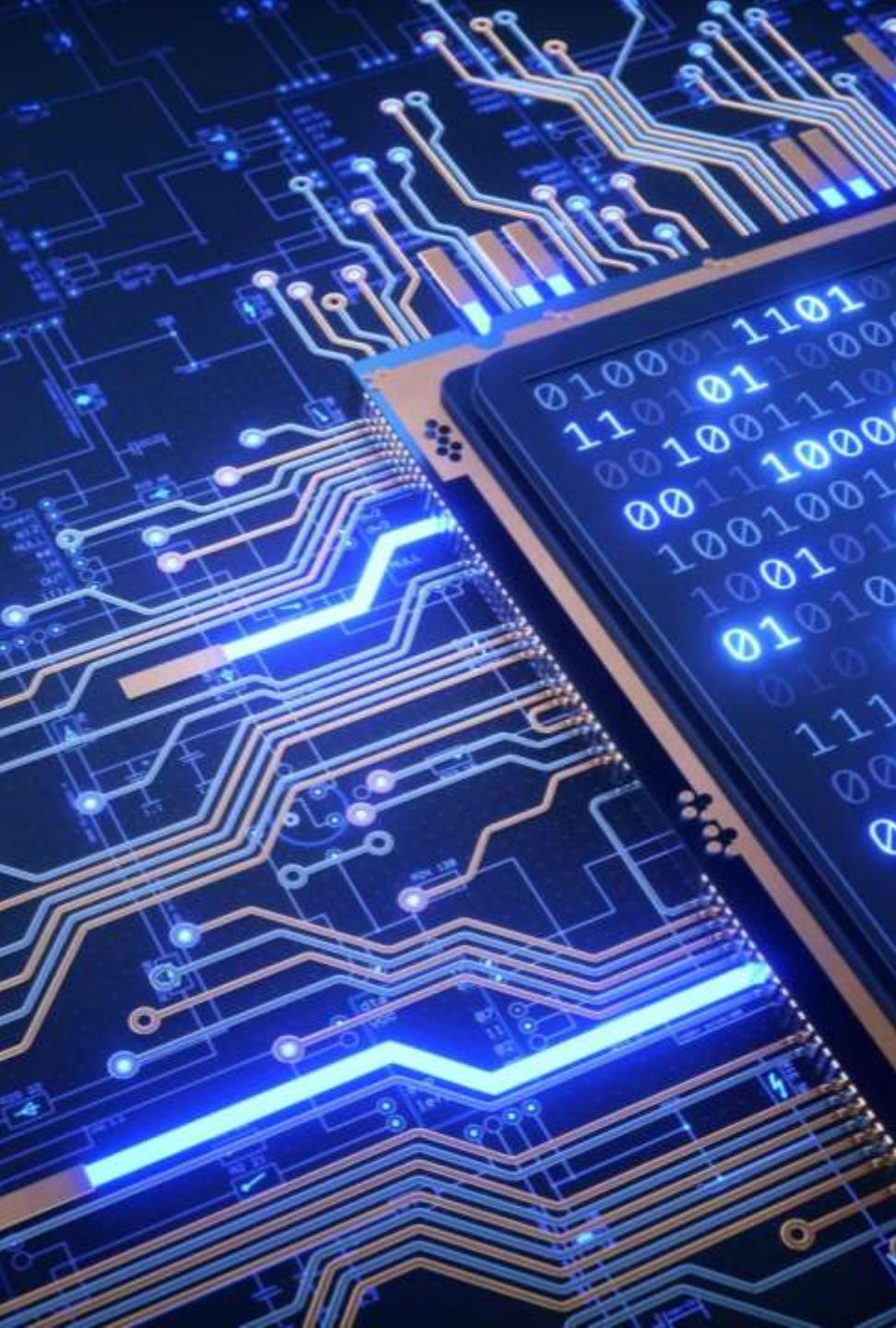
Use the explicit data management only *where you need it*



# Questions?

Get the updated training content from:

<https://github.com/kbvis3d/toshiba-cuda-2021>



# DAY 3

- CUDA Profiling with Nsight Systems
  - vector-add with pre-fetch
- Non-Unified Memory
  - cudaHostAlloc/cudaMallocHost
  - cudaMemcpyAsync
- CUDA Streams
  - vector-add with overlapped data init
  - vector-add with overlapped data transfer
  - streams-cipher sample
  - cudaStreamSynchronize

Get the updated training content from:

<https://github.com/kbvis3d/toshiba-cuda-2021>

# System- and Kernel-Level Profiling

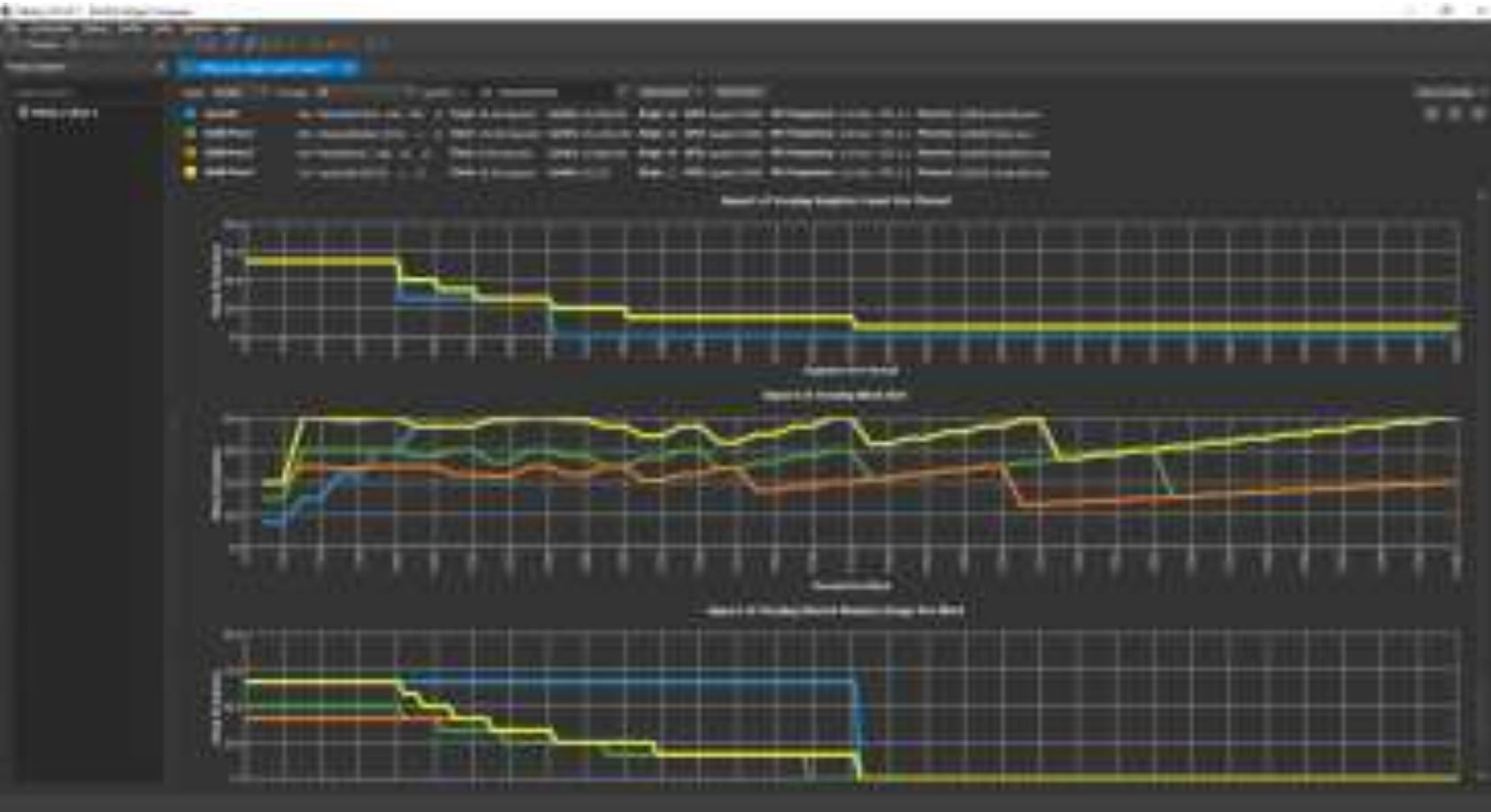
- **Nsight Systems:** System-level Profiling:
  - How effectively is my system delivering work to the GPU?
  - What is my system doing while the GPU is working?
  - How fast is data moving to/from the GPU? o How much time does the CPU take to control the GPU?
  - When do asynchronous operations occur?
- **Nsight Compute:** Kernel-Level Profiling
  - How fast does the GPU execute my kernel?

# Nvidia Nsight Systems



<https://developer.nvidia.com/nsight-systems>

# Nvidia Compute



<https://developer.nvidia.com/nsight-compute>

(install v2019.5.1 for compatibility with Pascal GPUs)

# Nsight Visual Studio Extension

The screenshot shows the Visual Studio IDE interface with the Nsight Visual Studio Extension installed. The menu bar is visible at the top, and the main code editor window displays CUDA C++ code for a vector addition kernel. The Solution Explorer and Properties windows are also visible on the right side of the interface.

The Nsight Compute 2019.5.1 and Nsight Systems 2021.2.4 options in the Extensions menu are highlighted with red boxes. The Trace option in the Nsight Systems submenu is also highlighted with a red box.

```
vectorAdd.cu // File contents
65
66    cudaMemPrefetchAsync(a, size, deviceId);
67    addVectorsErr = cudaGetLastError();
68    cudaMemPrefetchAsync(b, size, deviceId);
69    addVectorsErr = cudaGetLastError();
70    cudaMemPrefetchAsync(c, size, deviceId);
71    addVectorsErr = cudaGetLastError();
72
73    size_t threadsPerBlock;
74    size_t numberOfBlocks;
75
76    threadsPerBlock = 256;
```

Output window content:

```
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\cryptnet.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\crypt32.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\cryptbase.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\dwimapi.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\combase.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\uxtheme.dll'.
'vectorAdd.exe' (Win32): Unloaded 'C:\Windows\System32\dwimapi.dll'
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\nvapif64.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\setupapi.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\cfgmgr32.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\bcrypt.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\shell32.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\shlwapi.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\ole32.dll'.
'vectorAdd.exe' (Win32): Loaded 'C:\Windows\System32\kernel.appcore.dll'.
The program '[4028] vectorAdd.exe' has exited with code 0 (0x0).
```

System tray icons and status bar information are visible at the bottom of the screen.

# Nsight Visual Studio Extension

NVIDIA Nsight Systems 2021.2.4

File View Tools Help

Project 1 >

GRAPHICS-LAPTOP Target is ready [More info...](#)

Target application

Command line with arguments: [Edit arguments](#)  
"C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.4\bin\win64\Debug\vectorAdd.exe"

Working directory: C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.4\0\_Simple\09b-prefetch-to-gpu

Directory of executable is different from the working directory

Start

Start profiling manually  
 Start profiling after 10.0 seconds  
 Limit profiling to 10.0 seconds  
 Hotkey Start/Stop F12 (not available in console apps)

Sample target process

Sampling rate: 1 kHz

Collect call stacks of executing threads

Collect CPU context switch trace

Collect call stacks of blocked threads

Collect CUDA trace

Collect GPU metrics

Collect NVTX trace

Collect DX11 trace

Collect DX12 trace

Collect OpenGL trace

Collect Vulkan trace

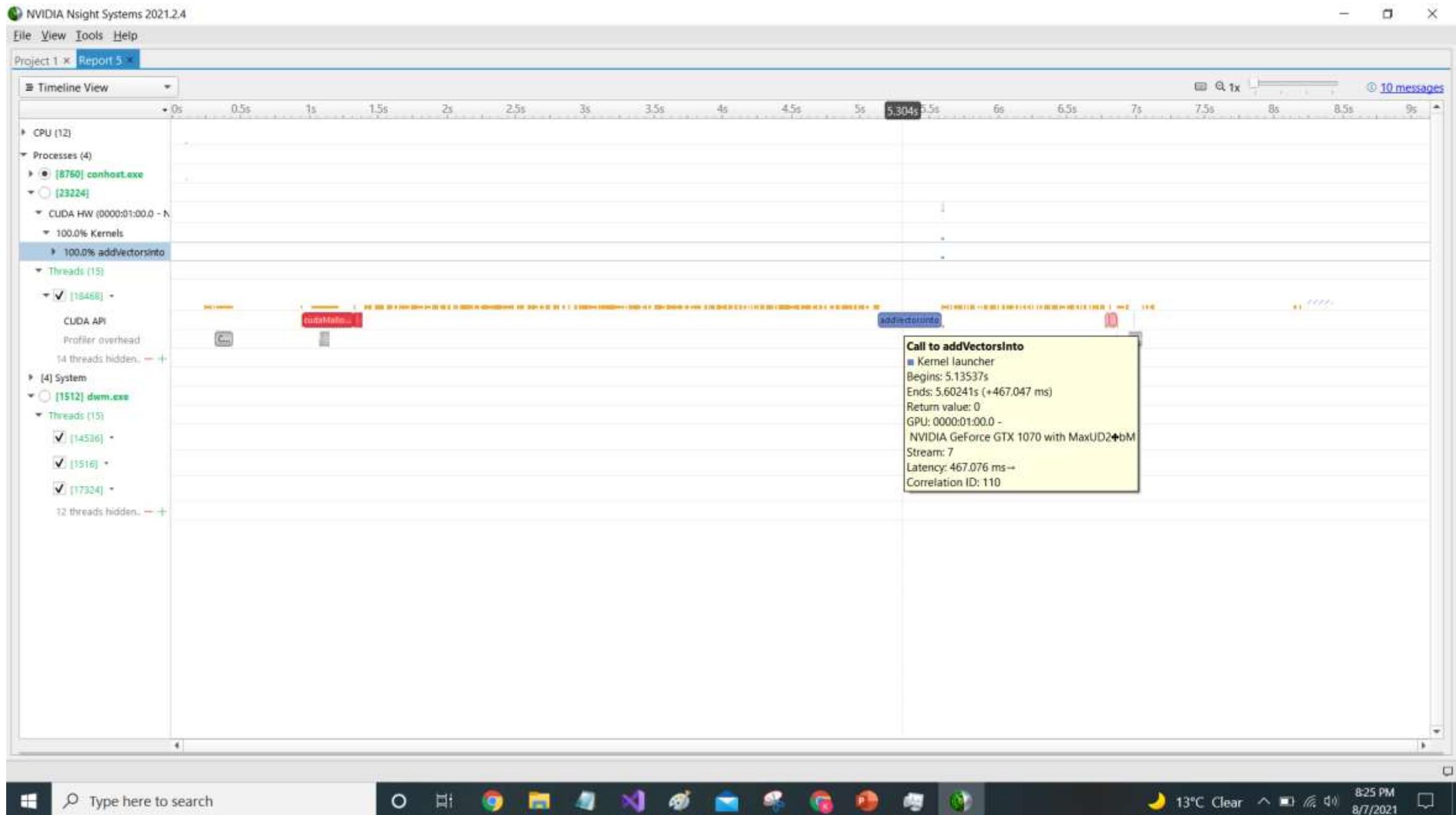
Custom ETW trace

Collect WDDM trace

Type here to search

13°C Clear 8:18 PM 8/7/2021

# Nsight Visual Studio Extension



**Call to addVectorsInto**

Kernel launcher  
Begins: 5.13537s  
Ends: 5.60241s (+467.047 ms)  
Return value: 0  
GPU: 0000:01:00.0 - NVIDIA GeForce GTX 1070 with MaxUD2+bm  
Stream: 7  
Latency: 467.076 ms--  
Correlation ID: 110



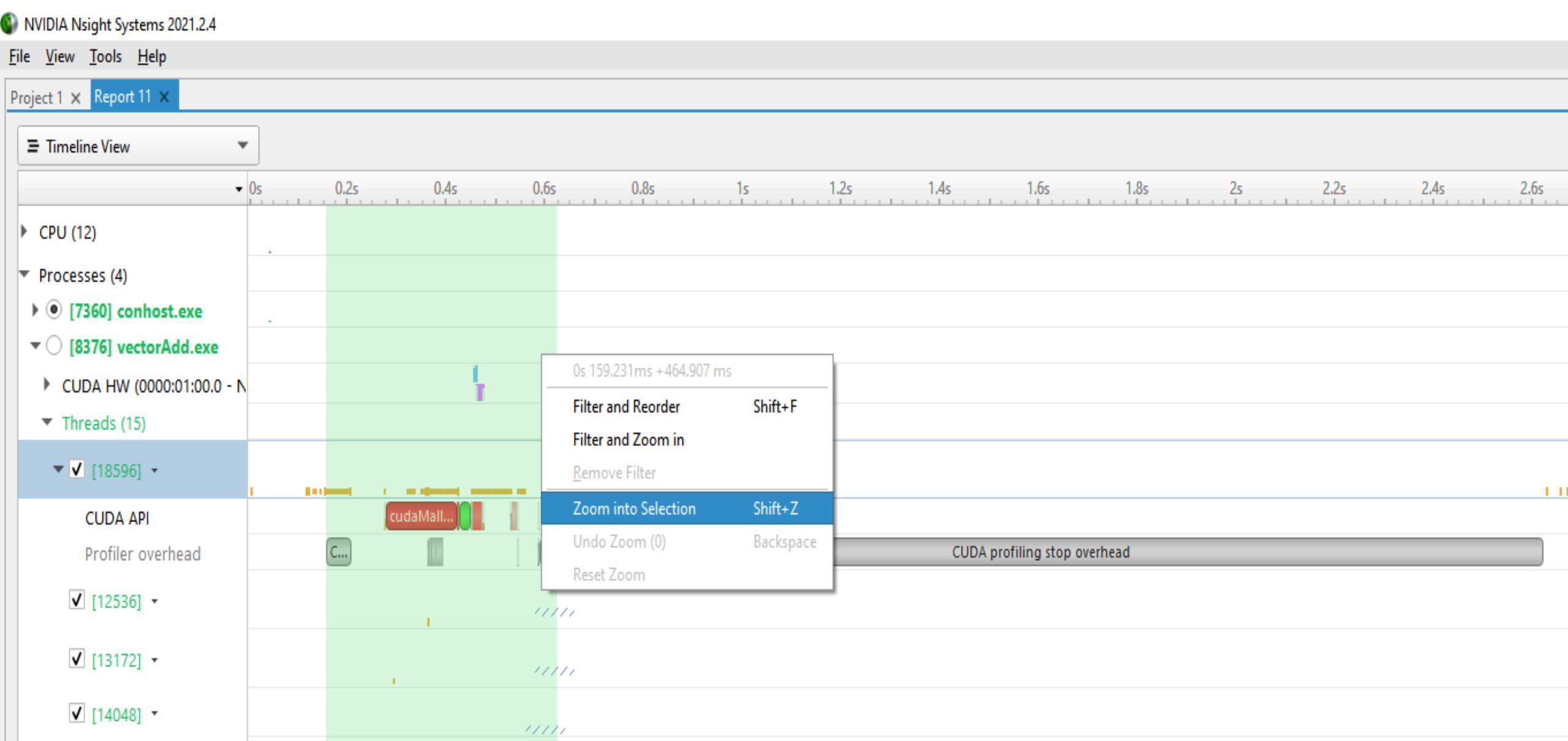
Type here to search



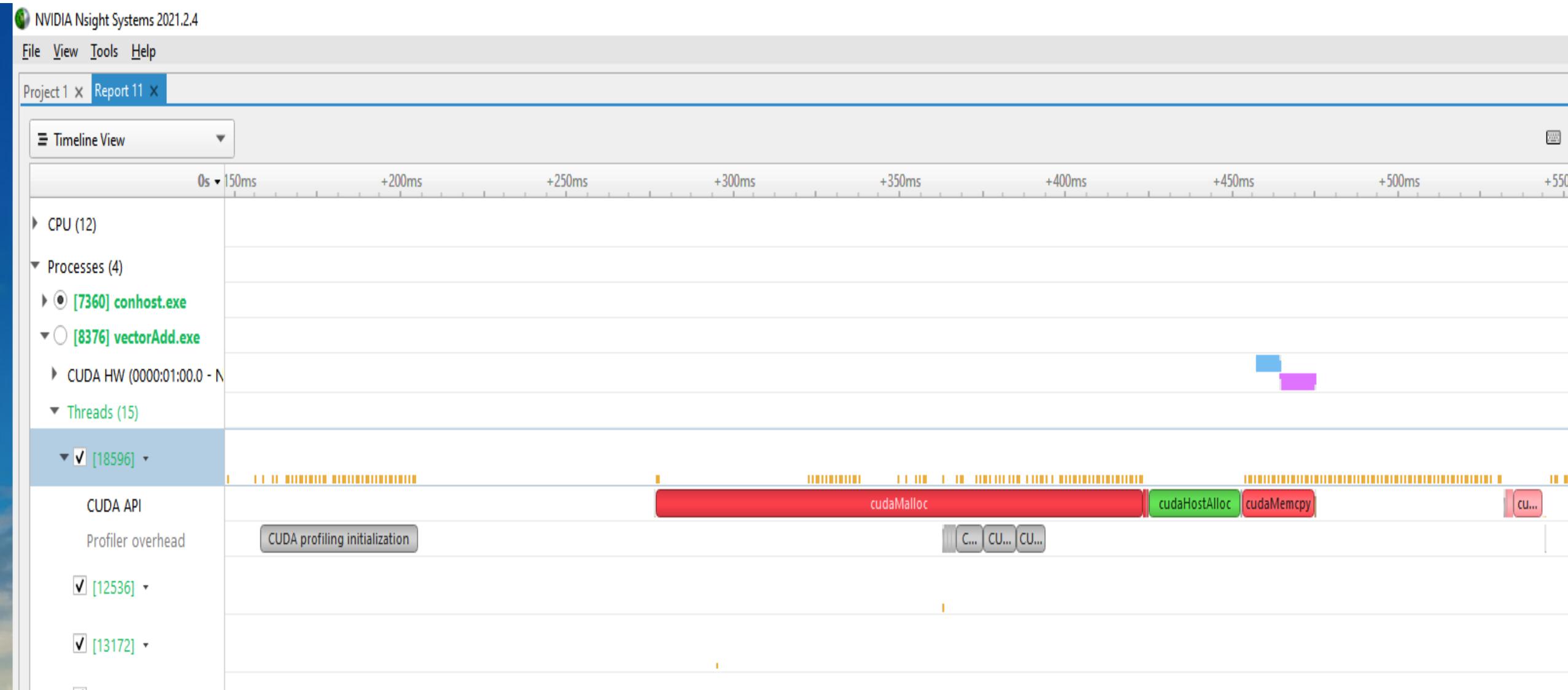
13°C Clear

825 PM  
8/7/2021

# Nsight Visual Studio Extension



# Nsight Visual Studio Extension



# No Pre-fetch

NVIDIA Nsight Systems 2020.3.4

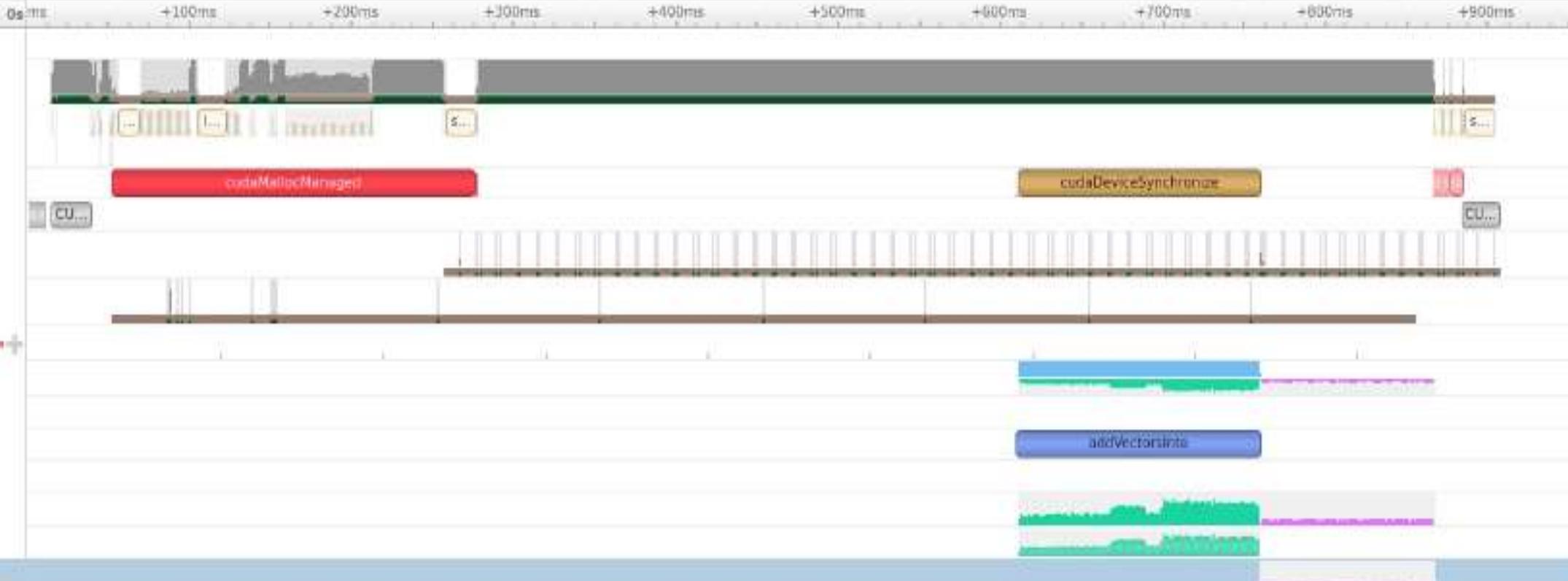
File View Tools Help

Project 1 X vector-add-no-prefetch-report.qdrep

vector-add-prefetch-report.qdrep X init-kernel-report.qdrep X prefetch-to-host-report.qdrep X

1 error, 4 warnings, 11 messages

Timeline View



CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
78.7	77932864	7526	10355.2	1824	133920	[CUDA Unified Memory memcpy HtoD]
21.3	21067424	768	27431.5	1376	159840	[CUDA Unified Memory memcpy DtoH]

# Pre-fetch to GPU



CUDA Memory Operation Statistics (nanoseconds)

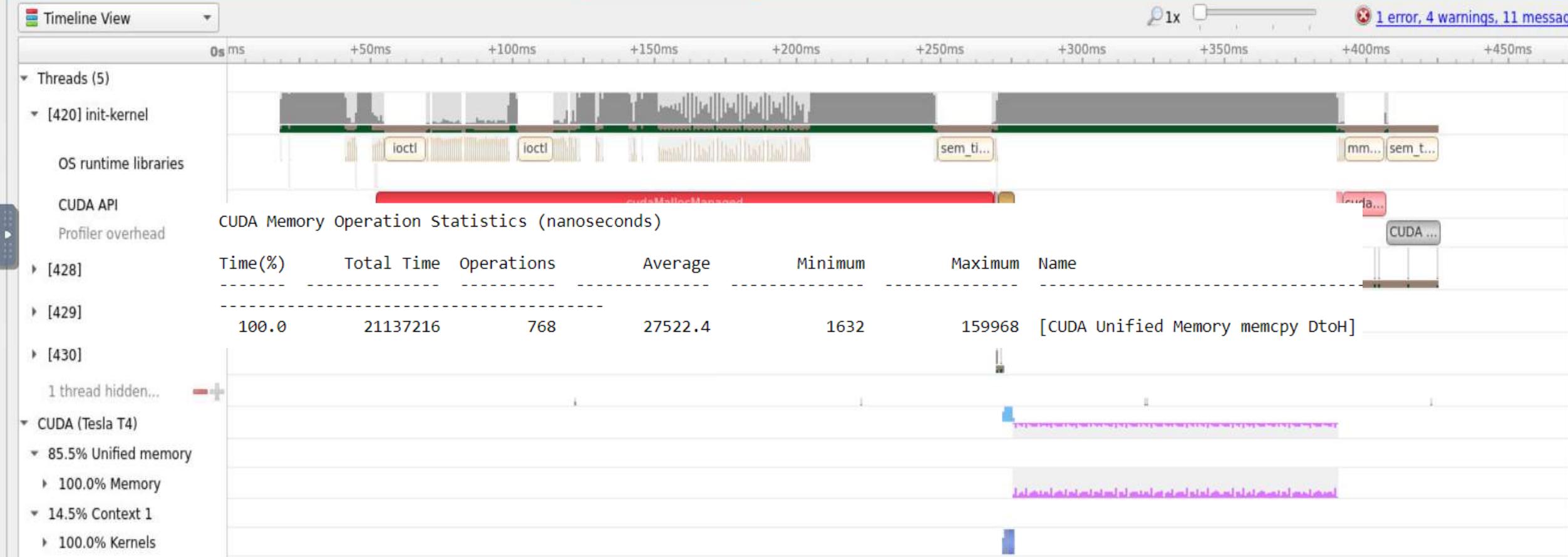
Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
75.7	65827744	192	342852.8	339808	353760	[CUDA Unified Memory memcpy HtoD]
24.3	21150752	768	27540.0	1632	159872	[CUDA Unified Memory memcpy DtoH]

# Pre-fetch with Init in Kernel

NVIDIA Nsight Systems 2020.3.4

File View Tools Help

Project 1 × vector-add-no-prefetch-report.qdrep × vector-add-prefetch-report.qdrep × init-kernel-report.qdrep × prefetch-to-host-report.qdrep ×



CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
100.0	21137216	768	27522.4	1632	159968	[CUDA Unified Memory memcpy DtoH]

# Pre-fetch back to Host

NVIDIA Nsight Systems 2020.3.4

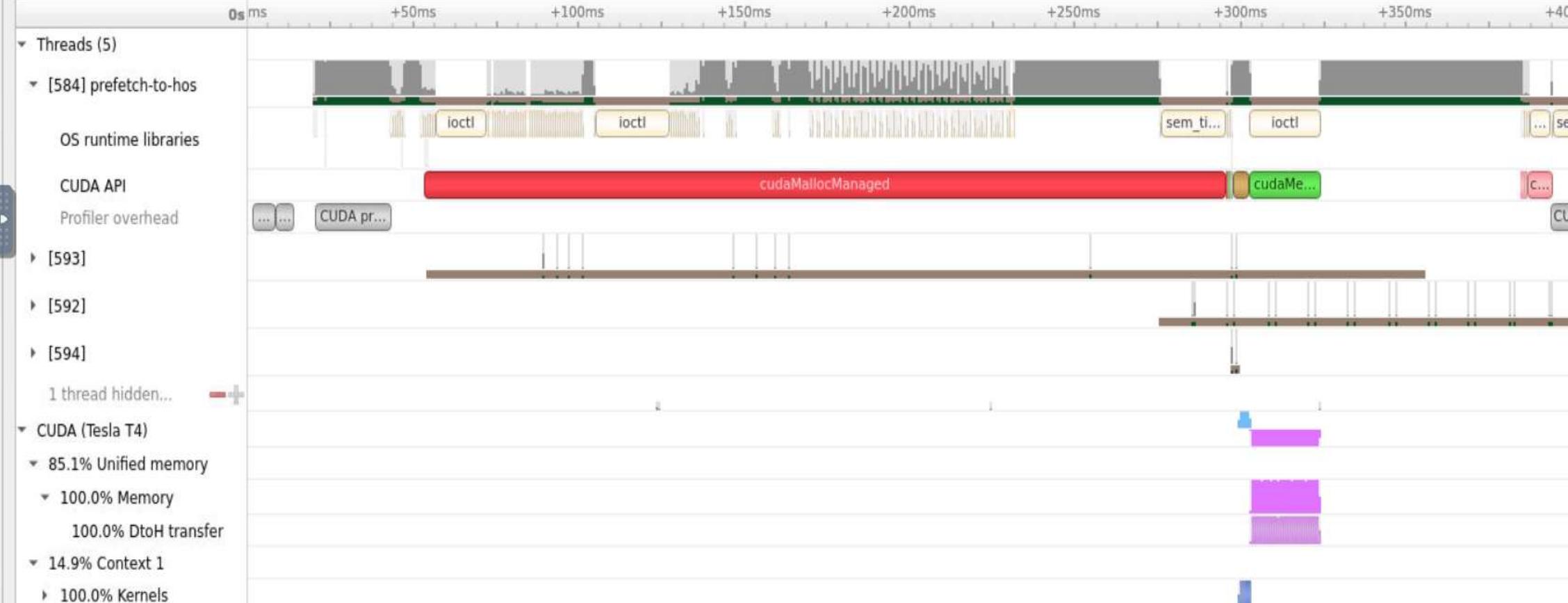
File View Tools Help

Project 1 X vector-add-no-prefetch-report.qdrep X vector-add-prefetch-report.qdrep X init-kernel-report.qdrep X prefetch-to-host-report.qdrep X

1 error, 4 warnings, 11 messages

Timeline View

1x

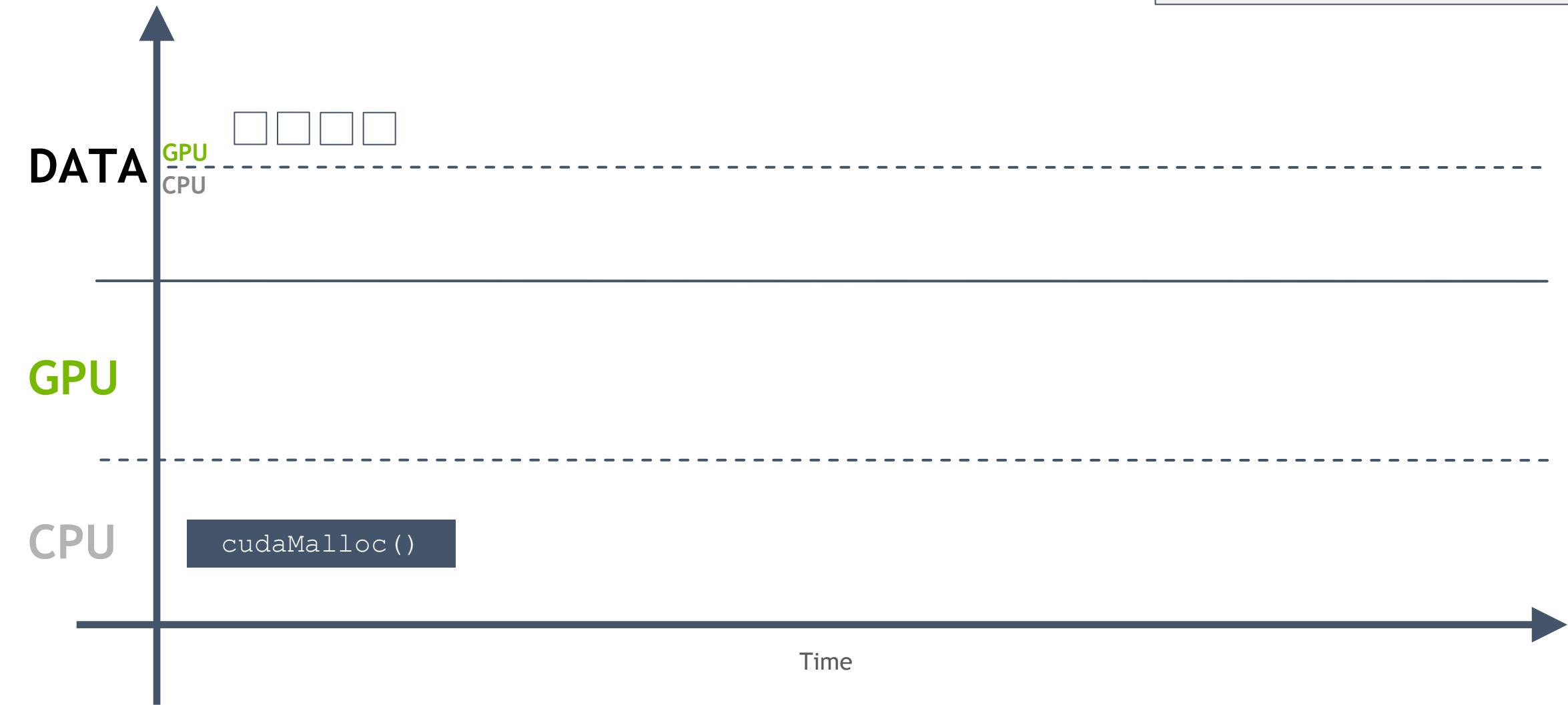


CUDA Memory Operation Statistics (nanoseconds)

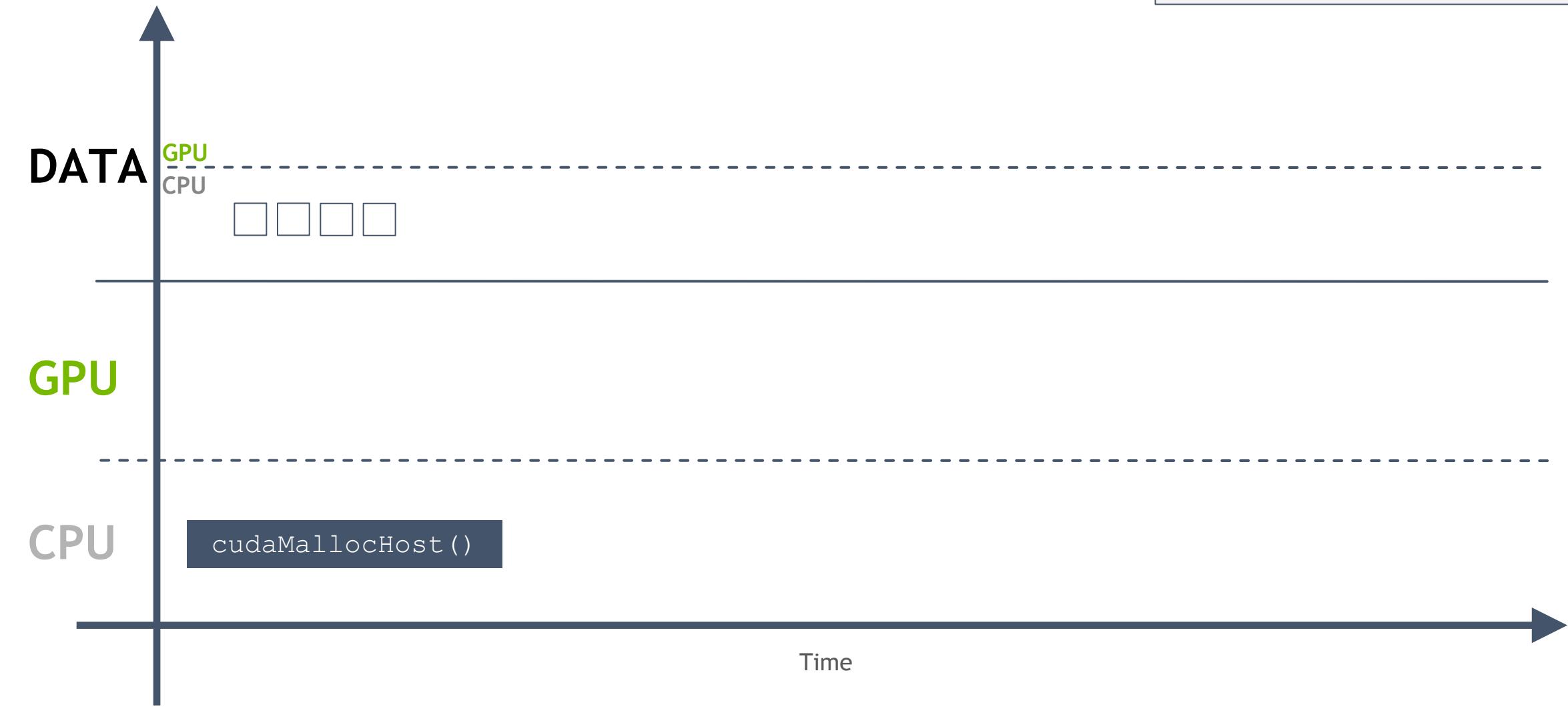
Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
100.0	20335360	64	317740.0	311040	319680	[CUDA Unified Memory memcpy DtoH]

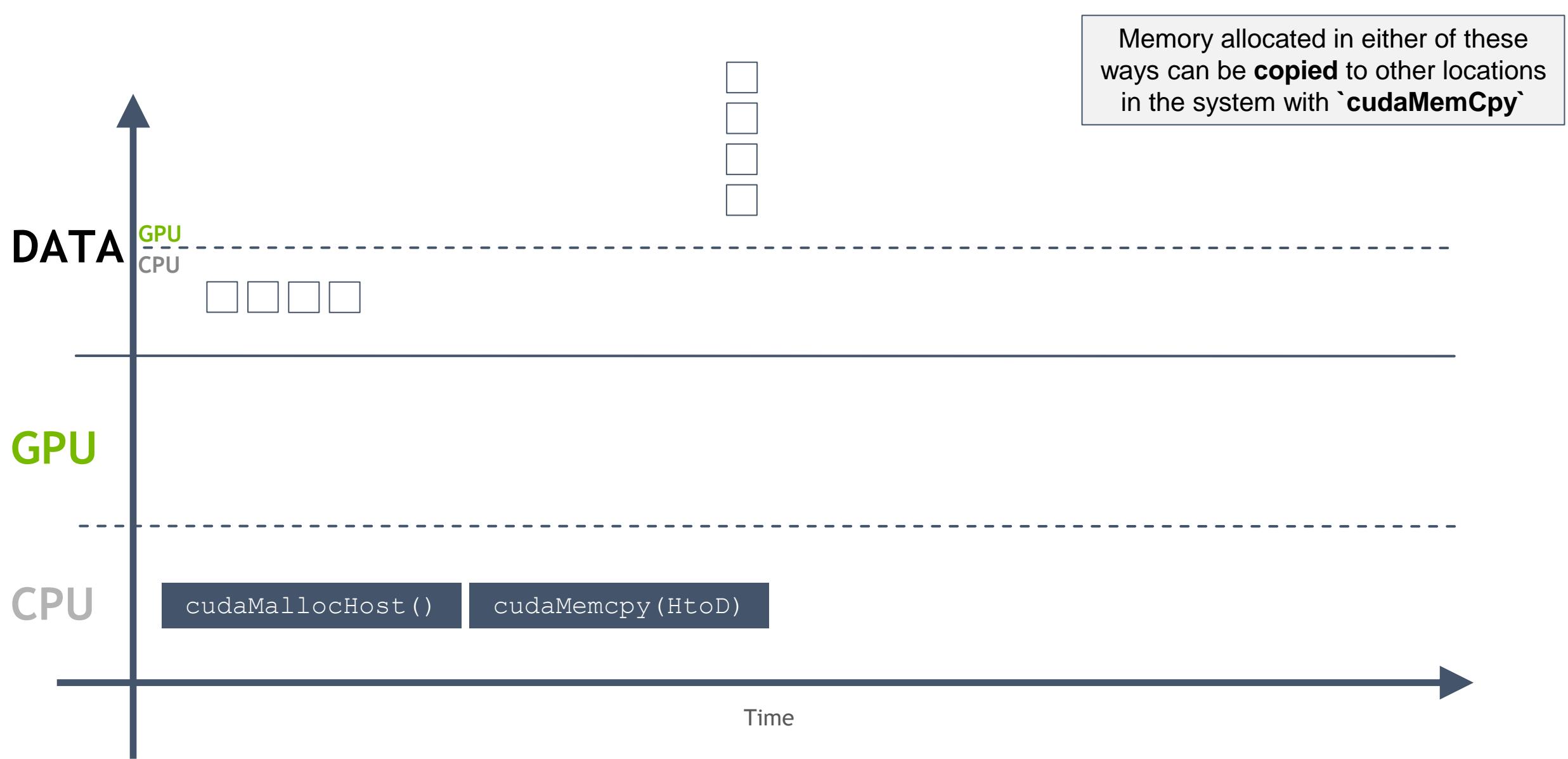
# Non-Unified Memory

Memory can be allocated directly to  
the GPU with ``cudaMalloc``



Memory can be allocated directly to  
the host with `'cudaMallocHost'`





Memory allocated in either of these ways can be **copied** to other locations in the system with `'cudaMemcpy'`

Copying leaves 2 copies in of in the system

**DATA**

GPU

CPU



**GPU**

**CPU**

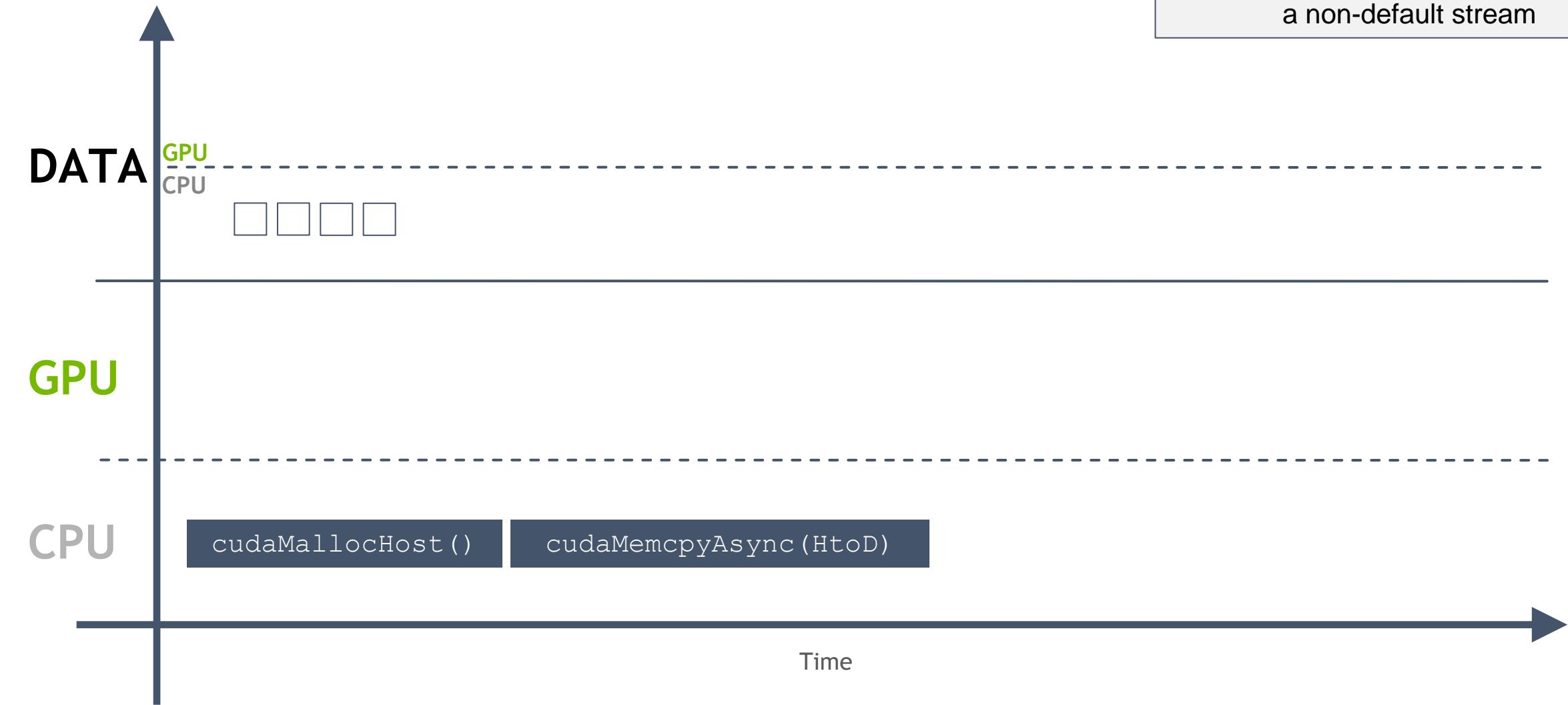
cudaMallocHost()

cudaMemcpy (HtoD)

Time

# cudaMemcpyAsync

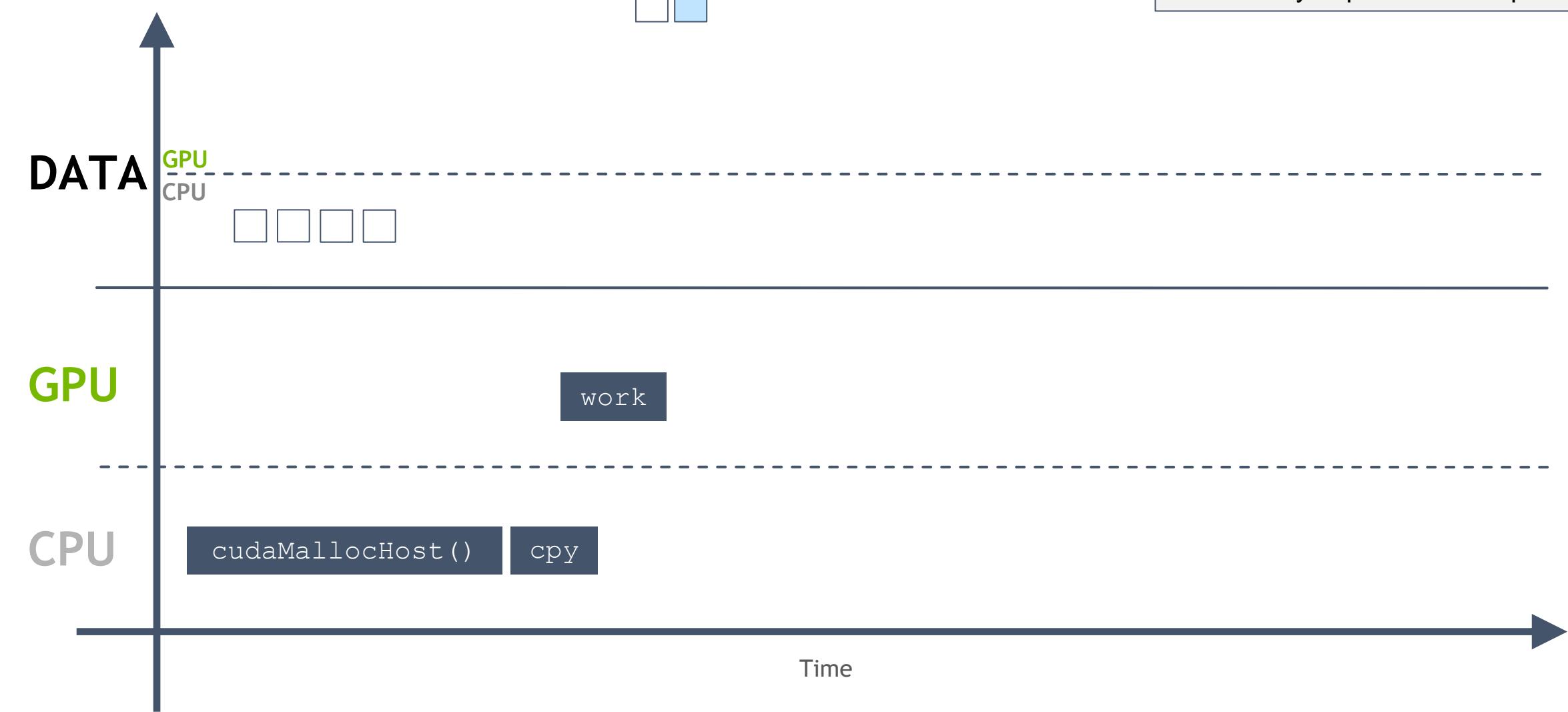
``cudaMemcpyAsync`` can  
asynchronously transfer memory over  
a non-default stream



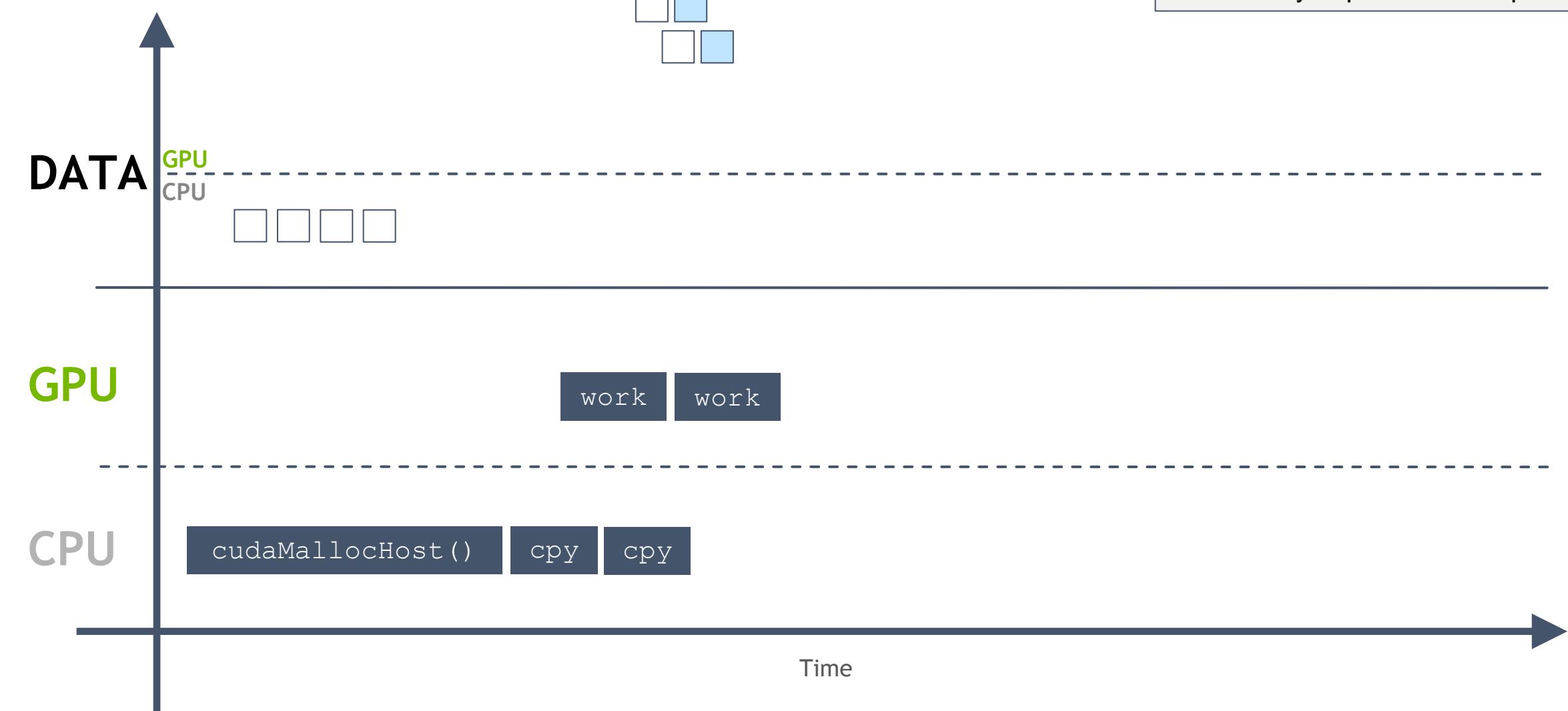
This can allow the **overlapping** memory copies and computation



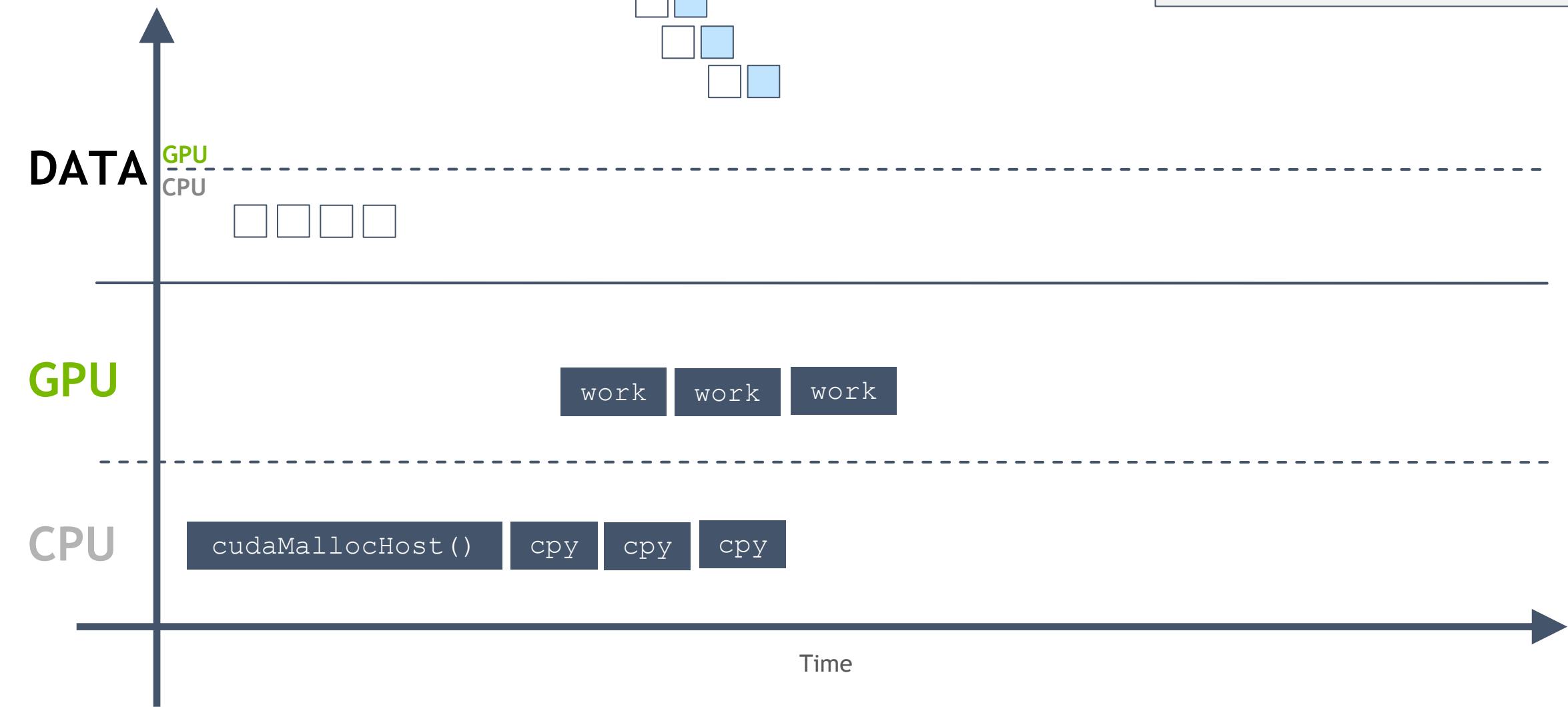
This can allow the **overlapping** memory copies and computation



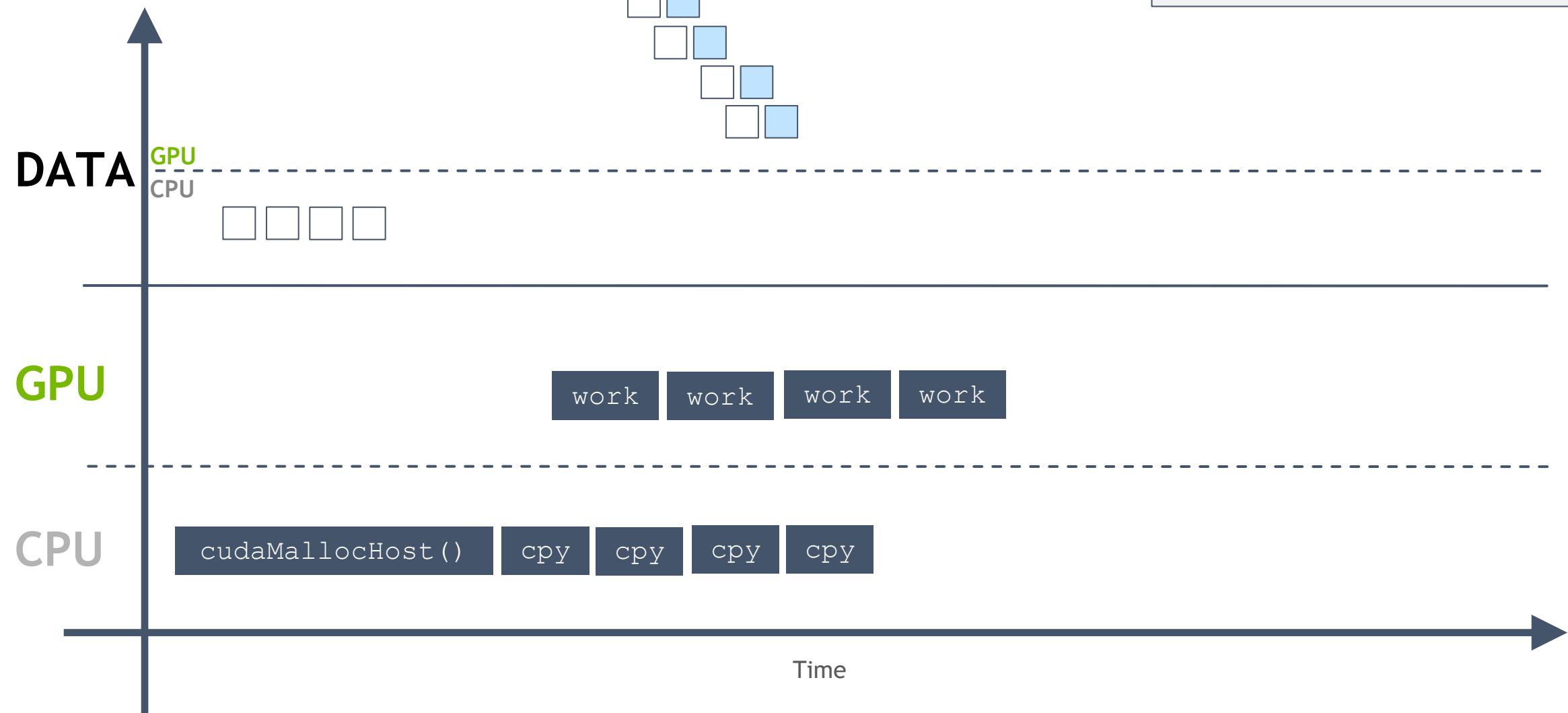
This can allow the **overlapping** memory copies and computation



This can allow the **overlapping** memory copies and computation

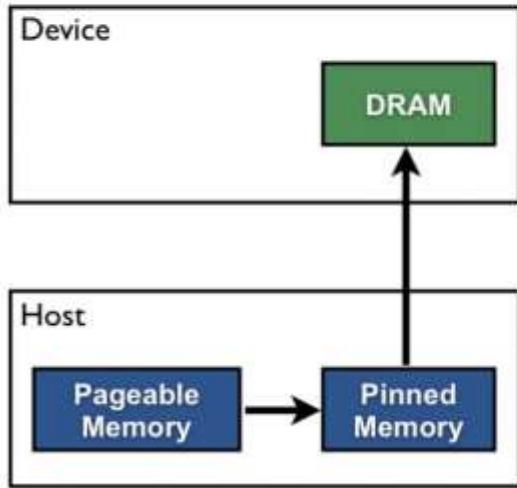


This can allow the **overlapping** memory copies and computation

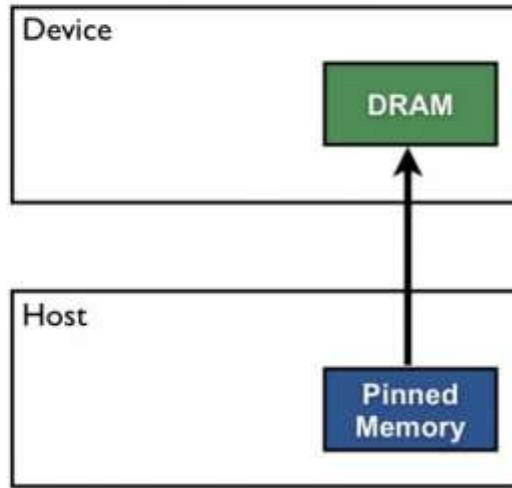


# Pinned Memory

*Pageable Data Transfer*



*Pinned Data Transfer*



- Using pinned memory speeds up `cudaMemcpy()` calls.
- Pinned memory is a limited resource.
- Pinning memory “removes” it from the host paged virtual memory system.
- Less useful since Over-subscription introduced for Unified Memory in CUDA 8.

# `samples / simpleZeroCopy`

- **cudaHostAlloc** – Allocates page-locked memory on host.
- **cudaHostRegister** – Registers (page-locks) an existing host memory range for use by CUDA.
- **cudaHostGetDevicePointer** – Gets device pointer corresponding to the mapped, pinned host buffer.
- Just use Unified Memory with **cudaMallocManaged()** and async prefetch/copy unless there is a clear case for explicit memory management.

# cudaHostAlloc()

```
flags = cudaHostAllocMapped;
checkCudaErrors(cudaHostAlloc((void **)a, bytes, flags));
checkCudaErrors(cudaHostAlloc((void **)b, bytes, flags));
checkCudaErrors(cudaHostAlloc((void **)c, bytes, flags));
```

```
checkCudaErrors(cudaHostGetDevicePointer((void **)&d_a, (void *)a, 0));
checkCudaErrors(cudaHostGetDevicePointer((void **)&d_b, (void *)b, 0));
checkCudaErrors(cudaHostGetDevicePointer((void **)&d_c, (void *)c, 0));
```

# cudaHostRegister()

```
a_UA = (float *) malloc(bytes + MEMORY_ALIGNMENT);
b_UA = (float *) malloc(bytes + MEMORY_ALIGNMENT);
c_UA = (float *) malloc(bytes + MEMORY_ALIGNMENT);

// We need to ensure memory is aligned to 4K (so we will need to pad memory accordingly)
a = (float *) ALIGN_UP(a_UA, MEMORY_ALIGNMENT);
b = (float *) ALIGN_UP(b_UA, MEMORY_ALIGNMENT);
c = (float *) ALIGN_UP(c_UA, MEMORY_ALIGNMENT);

checkCudaErrors(cudaHostRegister(a, bytes, cudaHostRegisterMapped));
checkCudaErrors(cudaHostRegister(b, bytes, cudaHostRegisterMapped));
checkCudaErrors(cudaHostRegister(c, bytes, cudaHostRegisterMapped));
```

```
checkCudaErrors(cudaHostGetDevicePointer((void **)&d_a, (void *)a, 0));
checkCudaErrors(cudaHostGetDevicePointer((void **)&d_b, (void *)b, 0));
checkCudaErrors(cudaHostGetDevicePointer((void **)&d_c, (void *)c, 0));
```

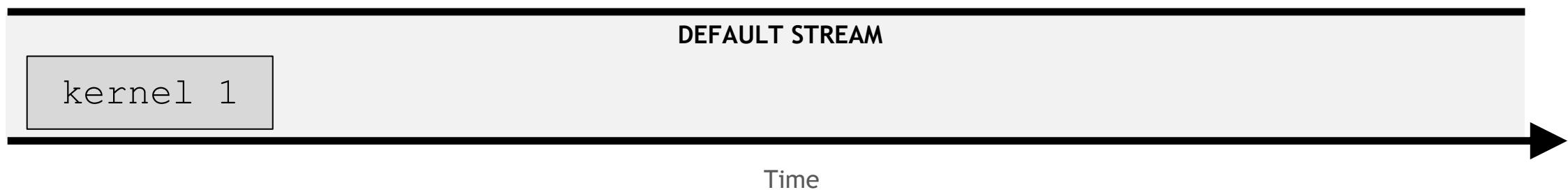
# Concurrent CUDA Streams

A **stream** is a series of instructions,  
and CUDA has a **default stream**

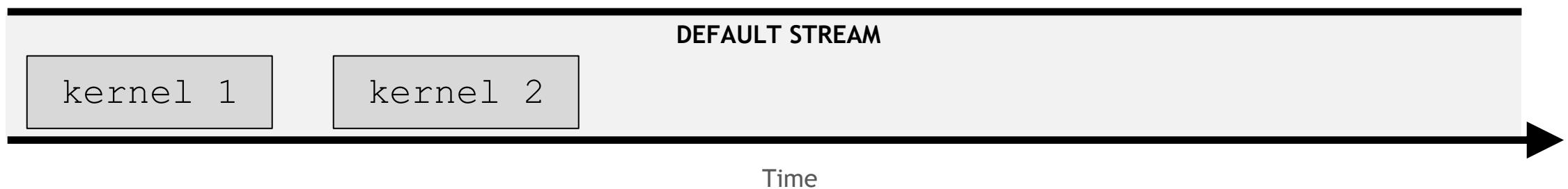
# DEFAULT STREAM

Time

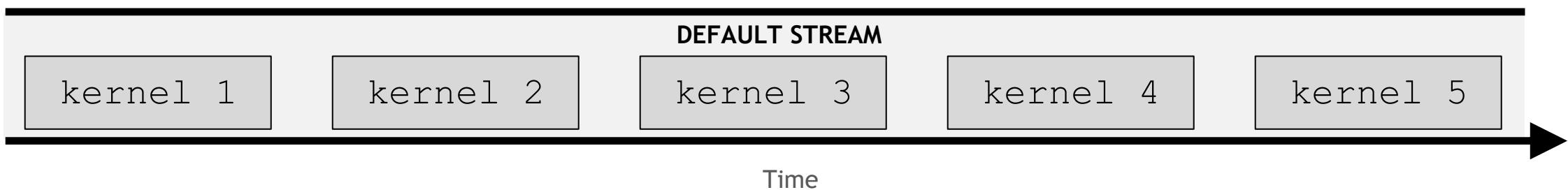
By default, CUDA kernels run in the  
**default stream**



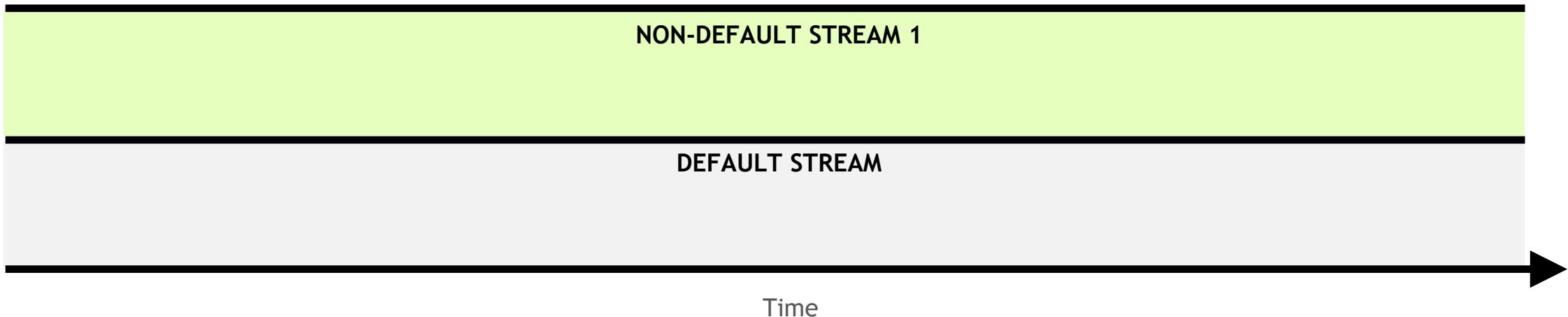
In any stream, including the default, an instruction in it (here a kernel launch) must complete before the next can begin



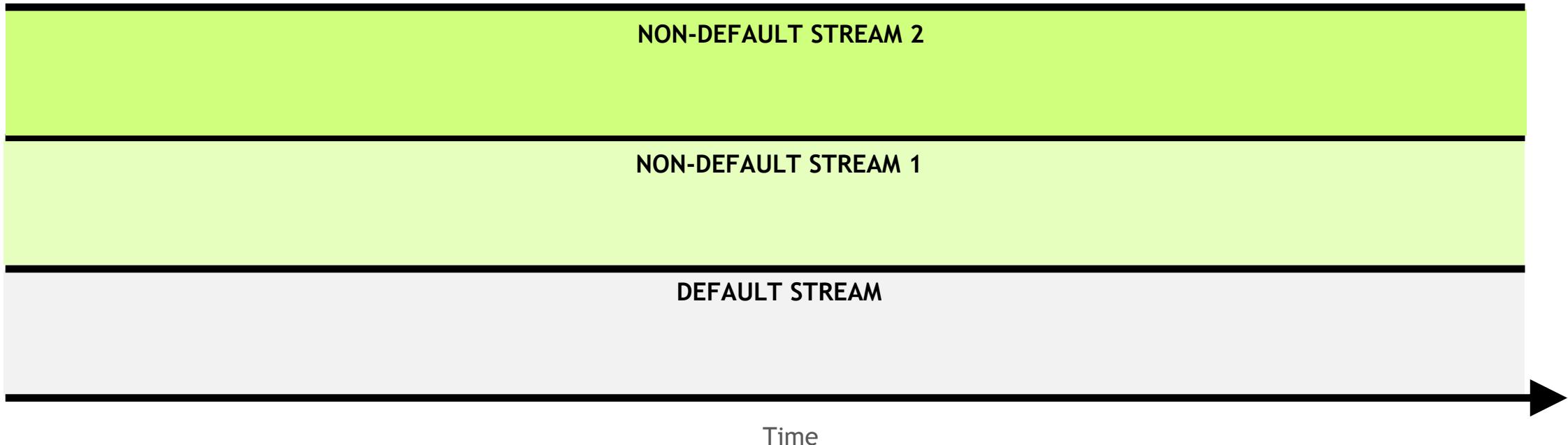
In any stream, including the default, an instruction in it (here a kernel launch) must complete before the next can begin



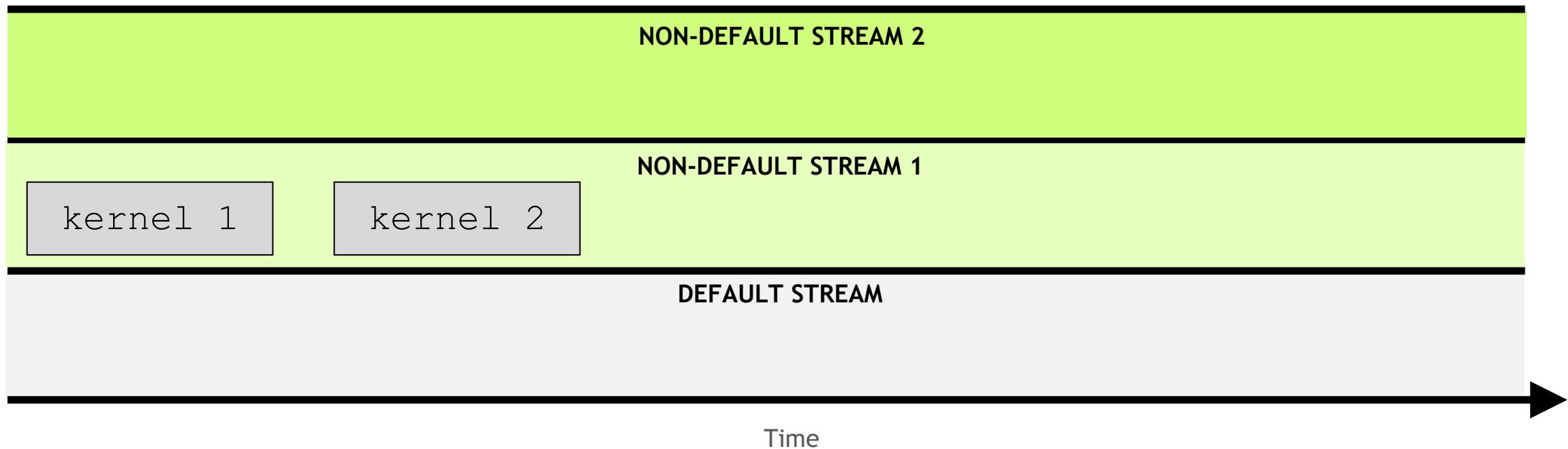
**Non-default streams** can also be created for kernel execution



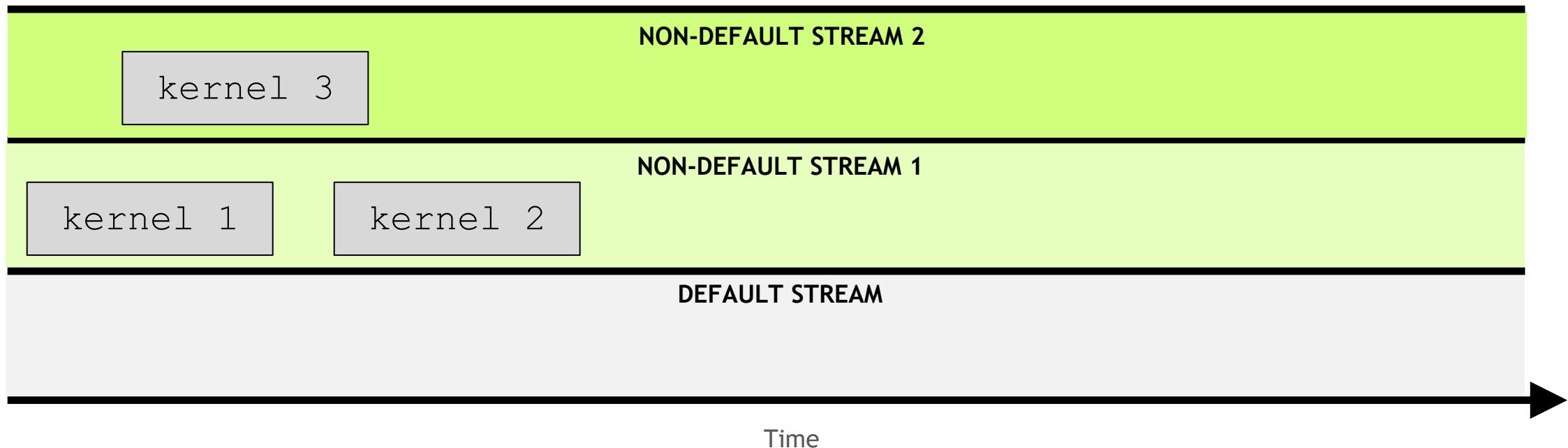
**Non-default streams** can also be created for kernel execution



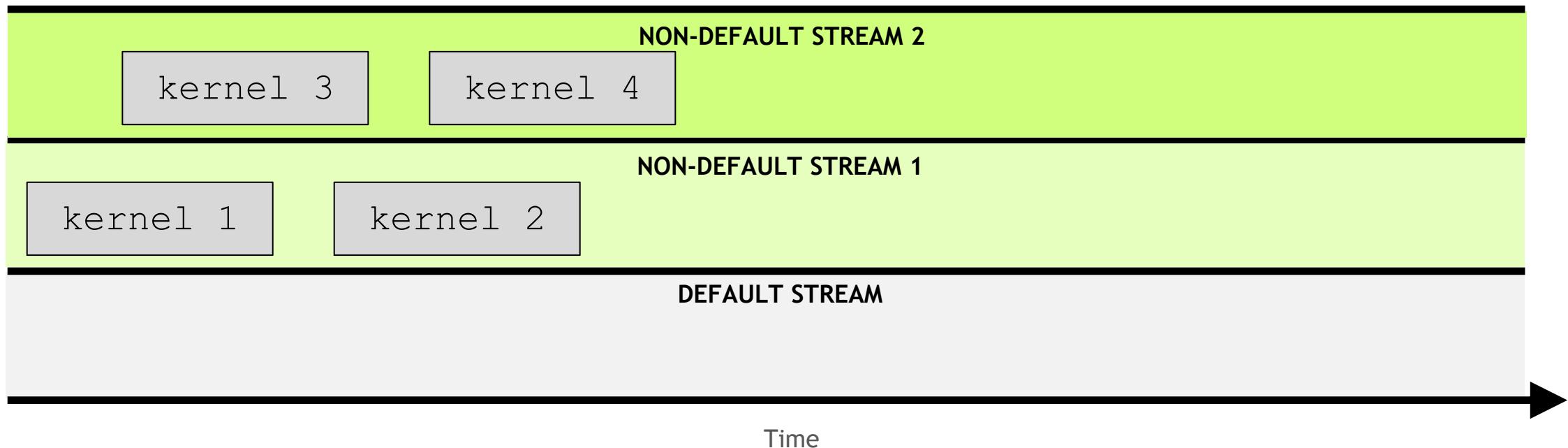
Kernels within any single stream must execute in order



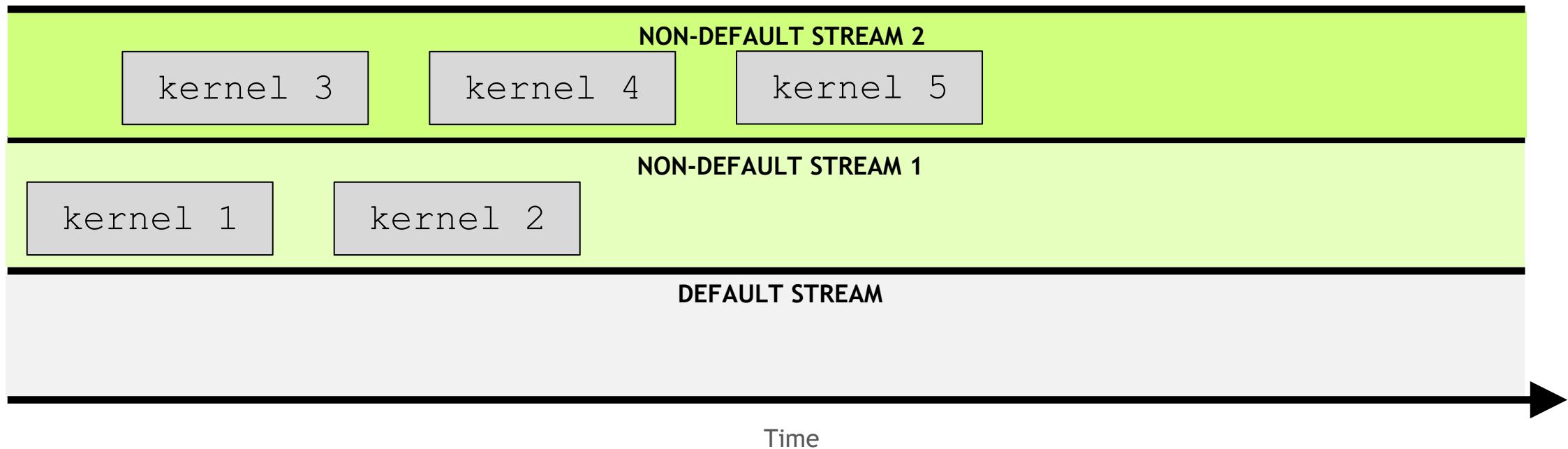
However, kernels in **different, non-default streams**, can interact concurrently



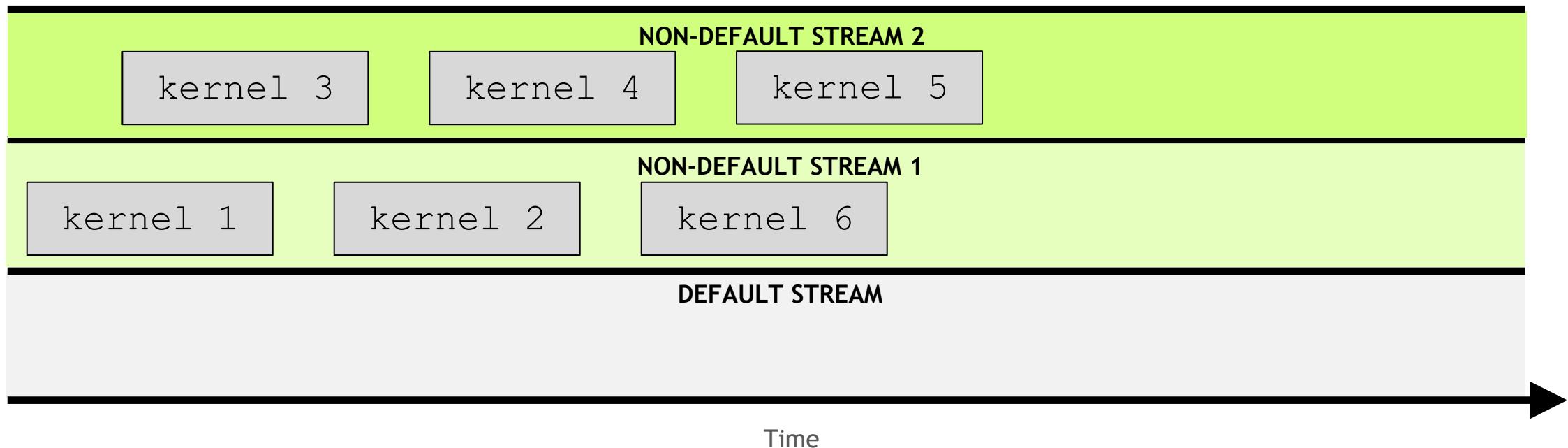
However, kernels in **different, non-default streams**, can interact concurrently



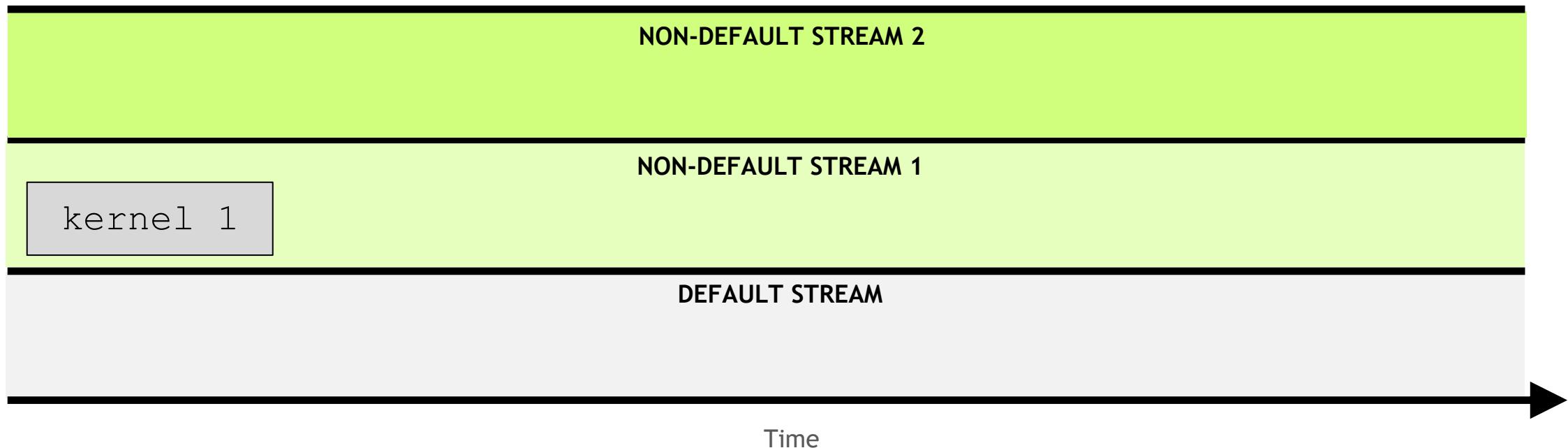
However, kernels in **different, non-default streams**, can interact concurrently



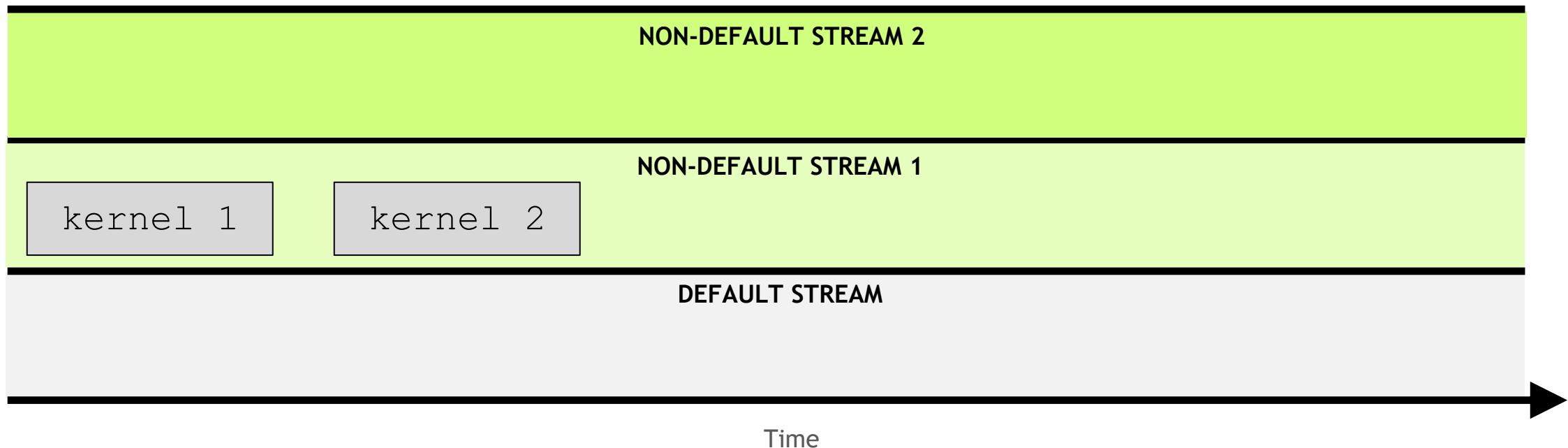
However, kernels in **different, non-default streams**, can interact concurrently



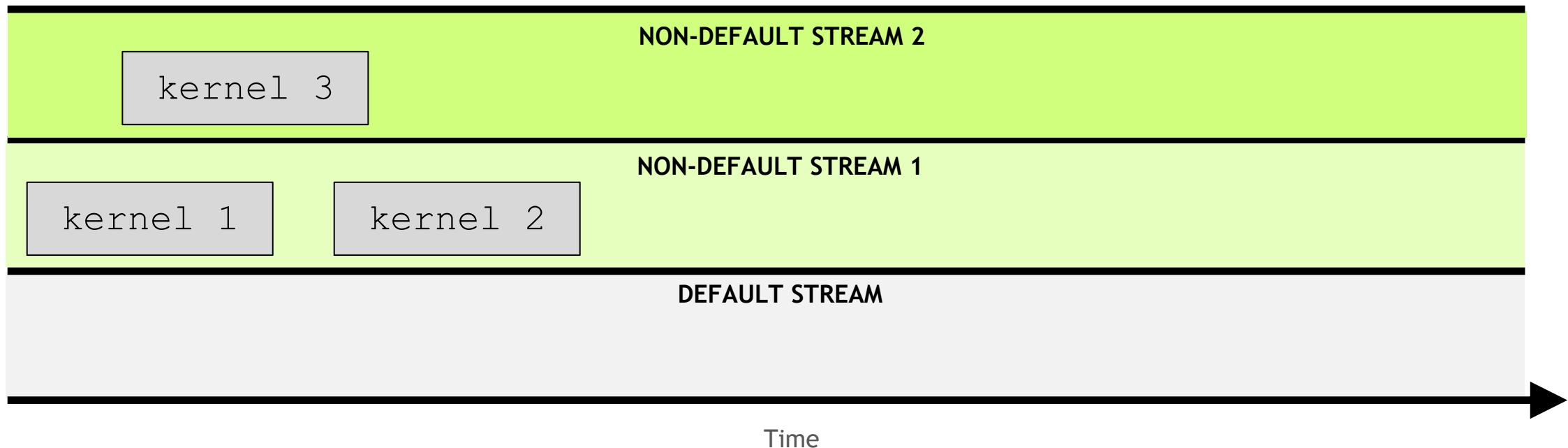
The default stream is special: it  
blocks all kernels in all other  
streams



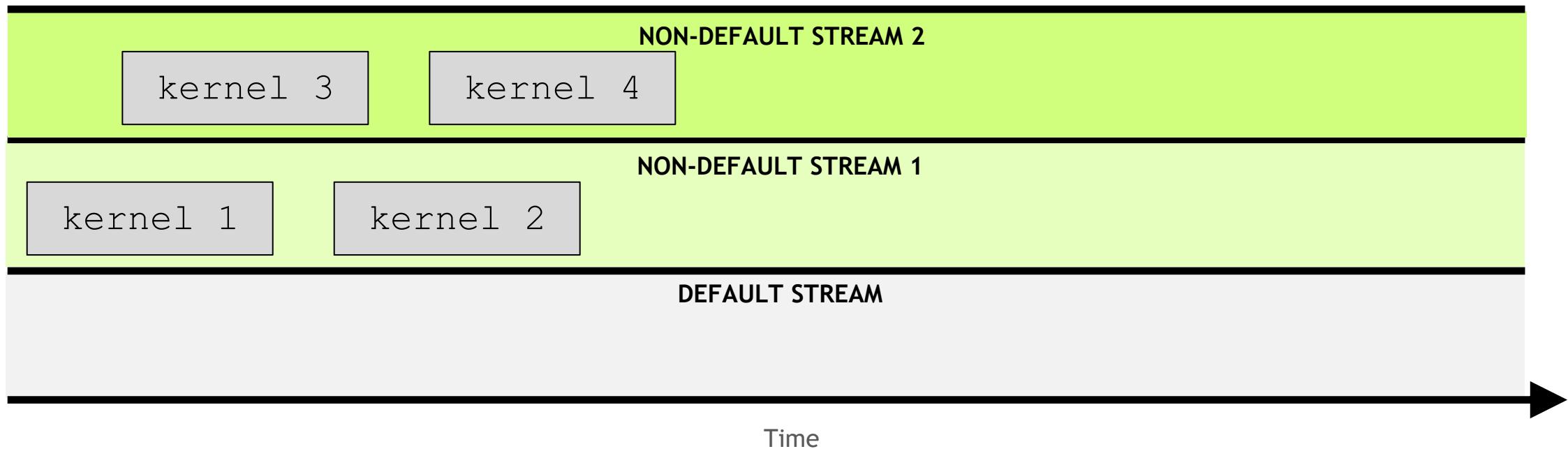
The default stream is special: it  
blocks all kernels in all other  
streams



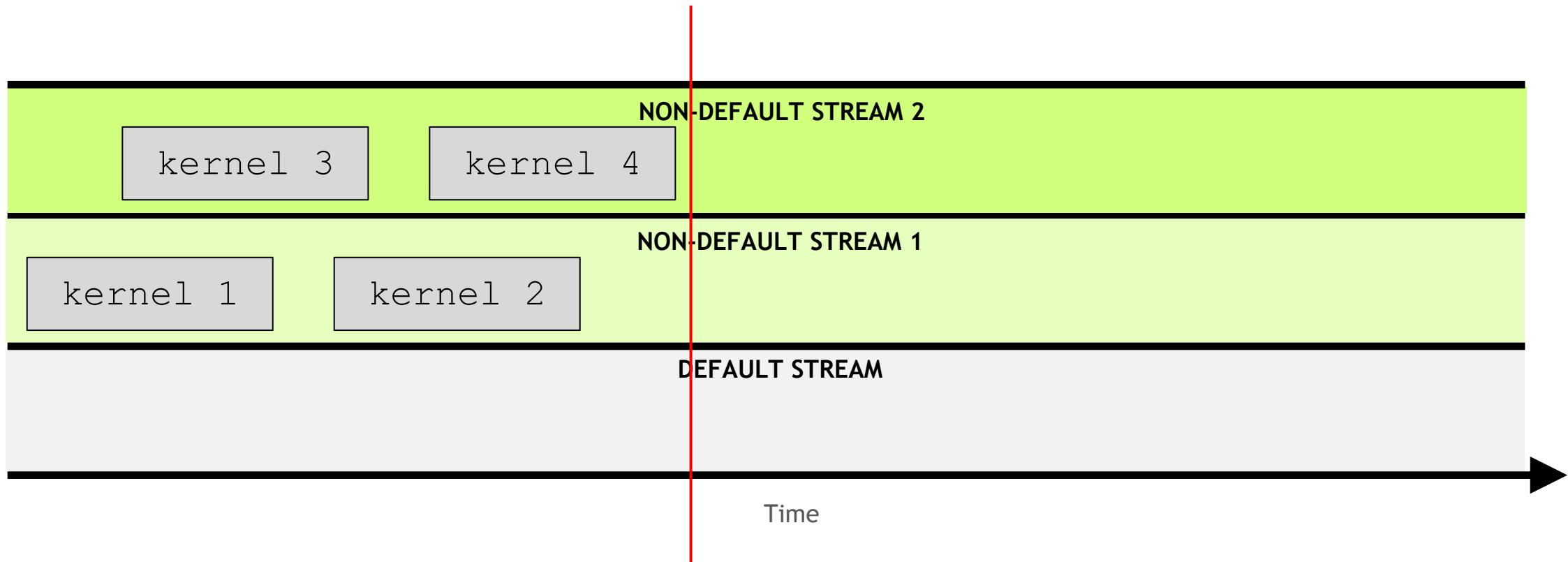
The default stream is special: it  
blocks all kernels in all other  
streams



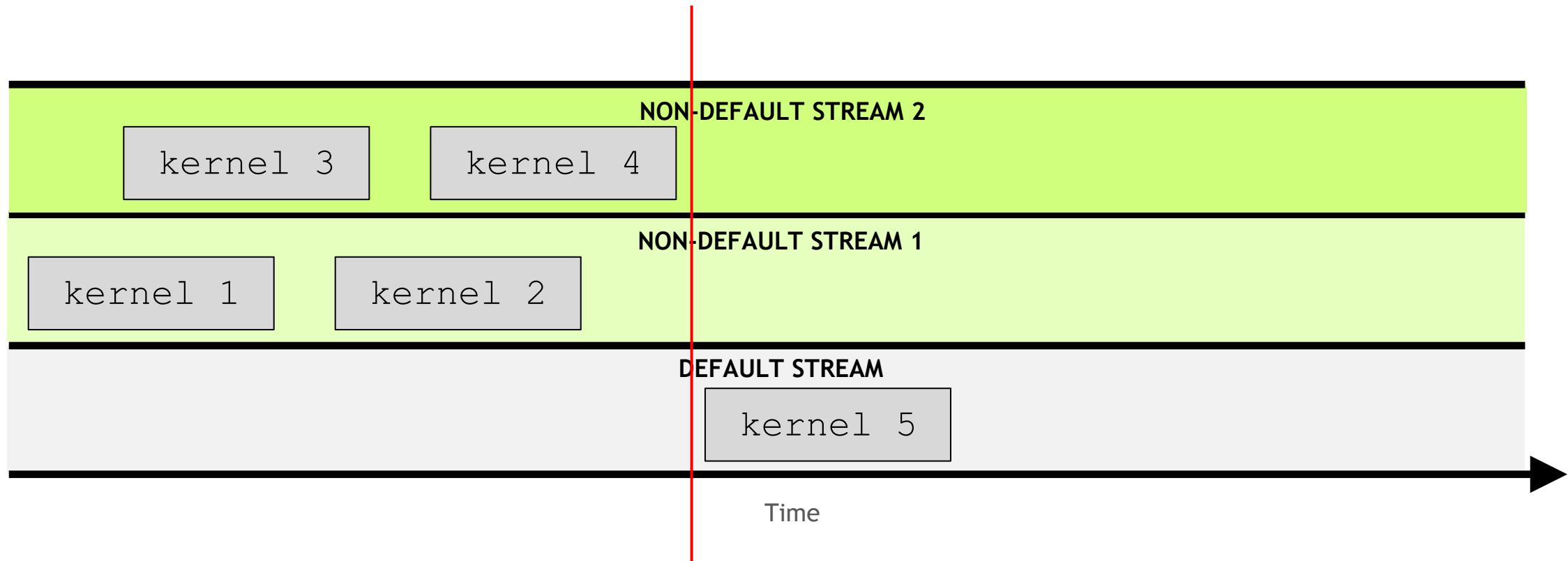
The default stream is special: it  
blocks all kernels in all other  
streams



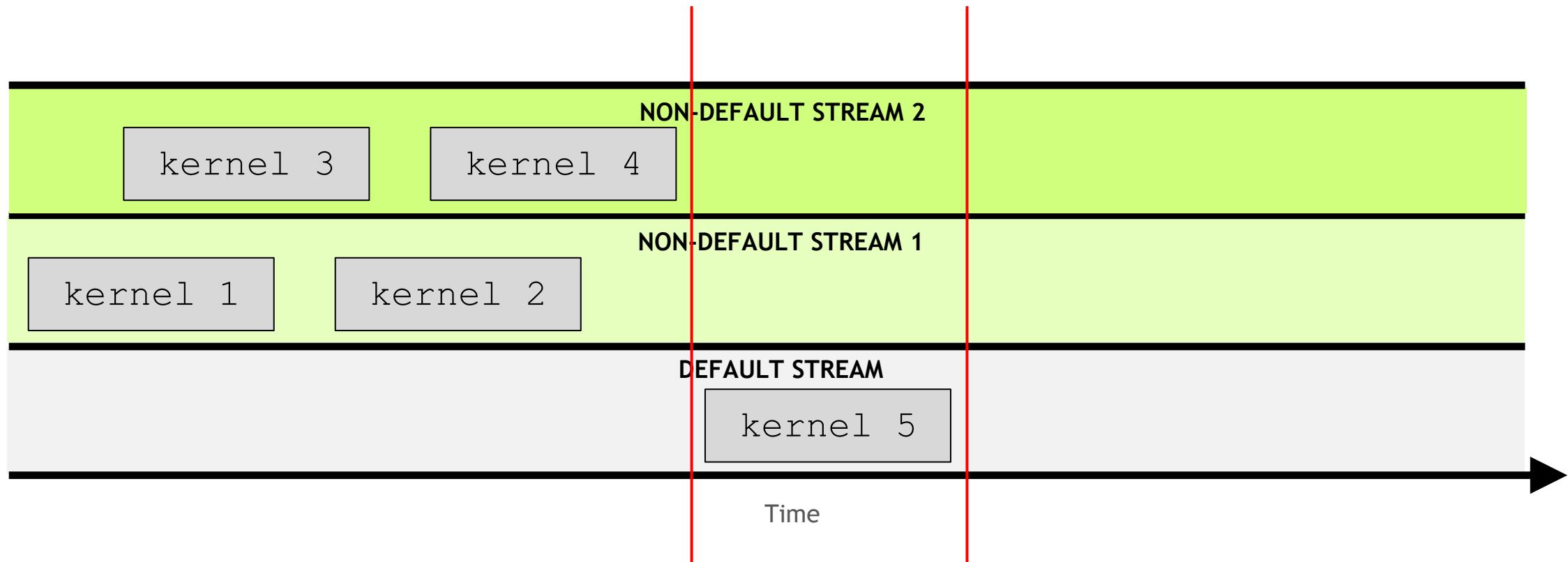
The default stream is special: it  
**blocks all kernels in all other streams**



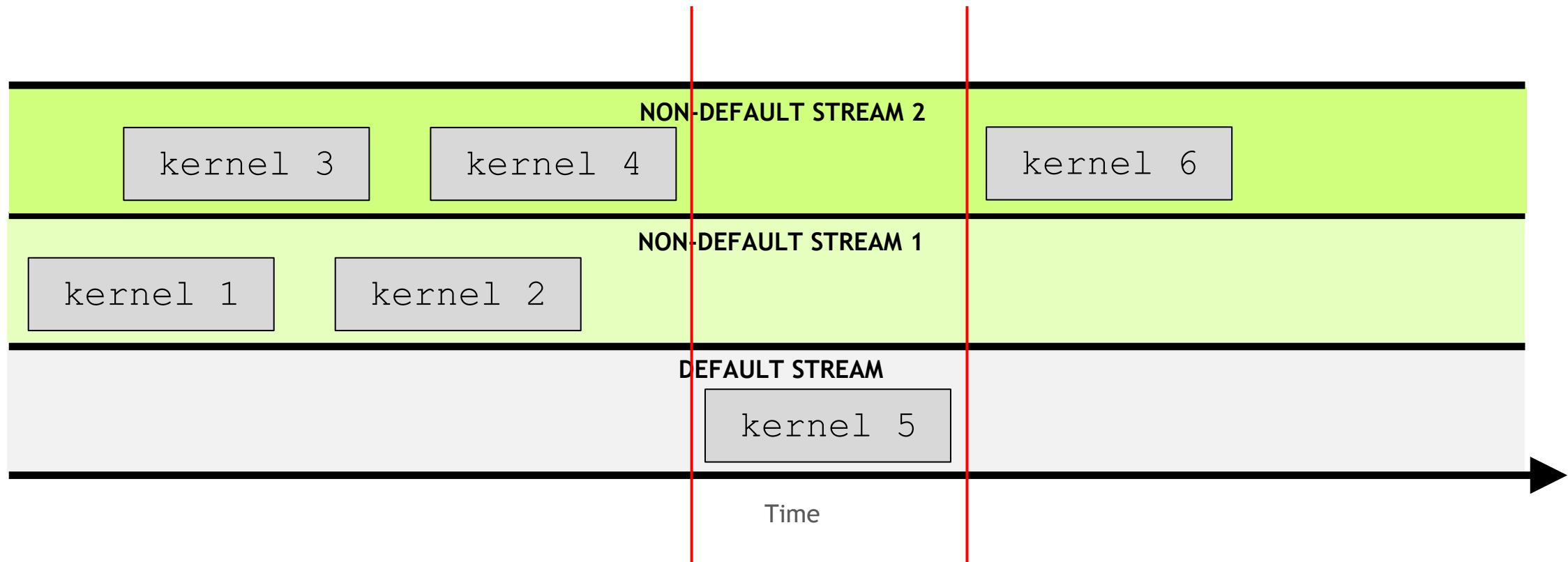
The default stream is special: it  
blocks all kernels in all other  
streams



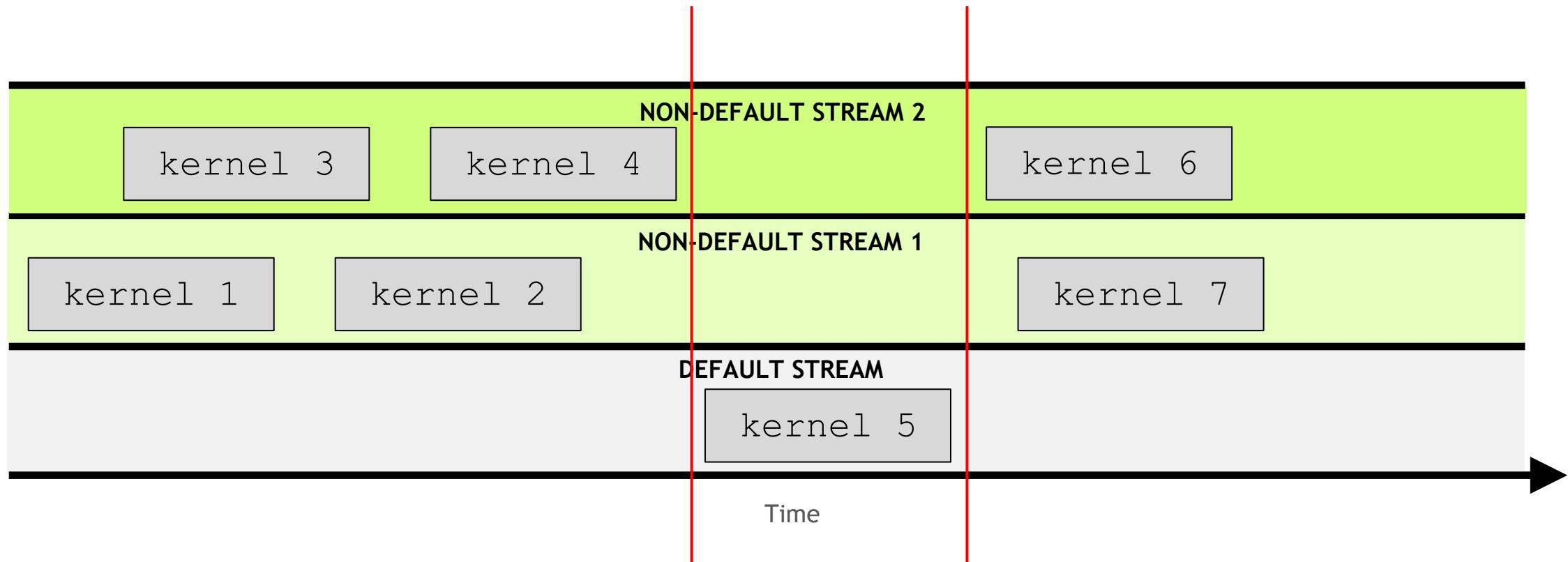
The default stream is special: it  
**blocks all kernels in all other streams**



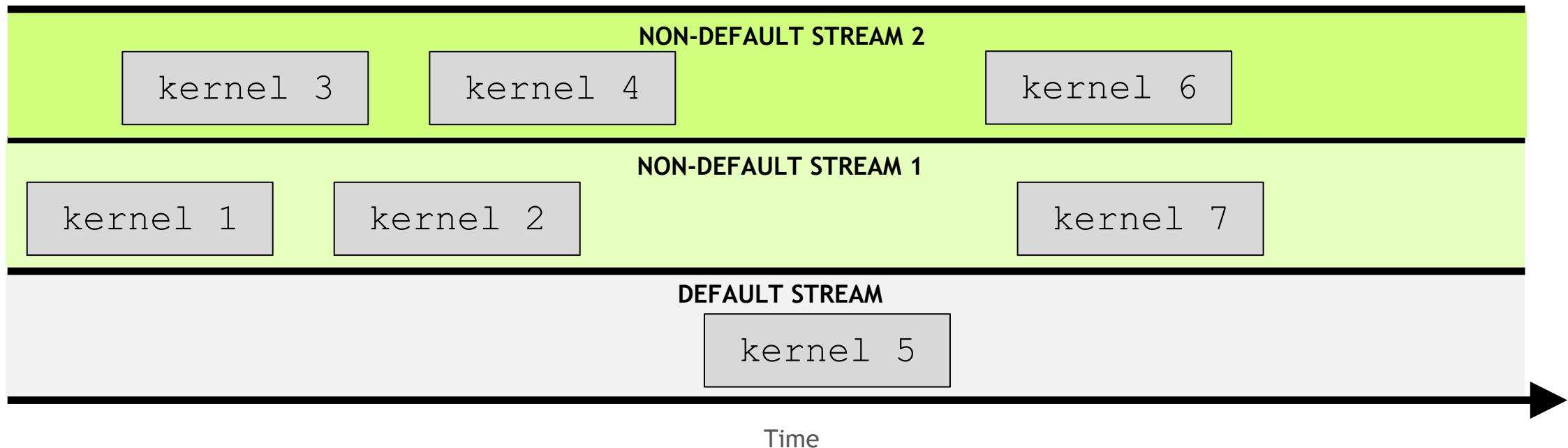
The default stream is special: it  
**blocks all kernels in all other streams**



The default stream is special: it  
**blocks all kernels in all other streams**



The default stream is special: it  
**blocks all kernels in all other streams**



# samples \ simple-streams

- Default Stream

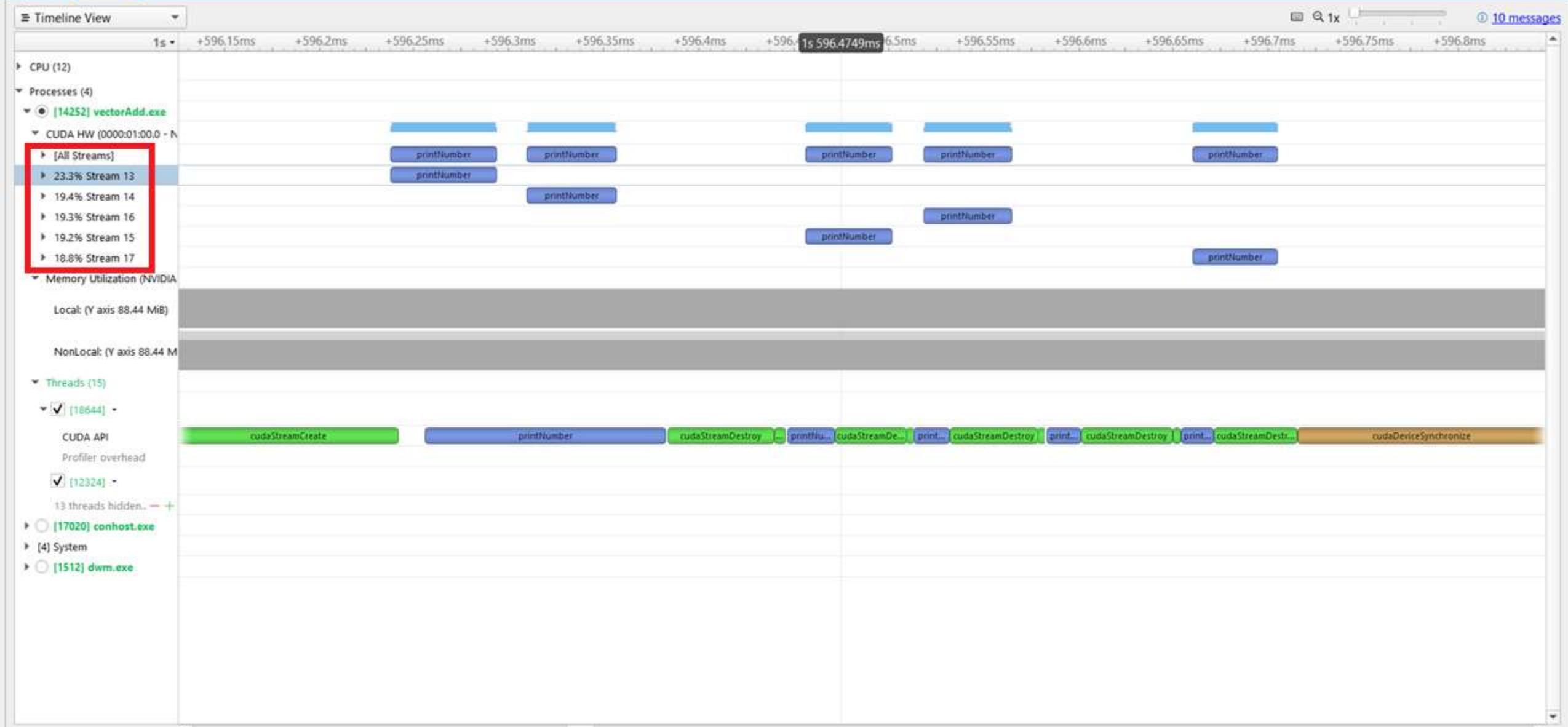
```
__global__ void printNumber(int number)
{
    printf("%d\n", number);
}
```

```
int main()
{
    for (int i = 0; i < 5; ++i)
    {
        printNumber<<<1, 1>>>(i);
    }
    cudaDeviceSynchronize();
}
```

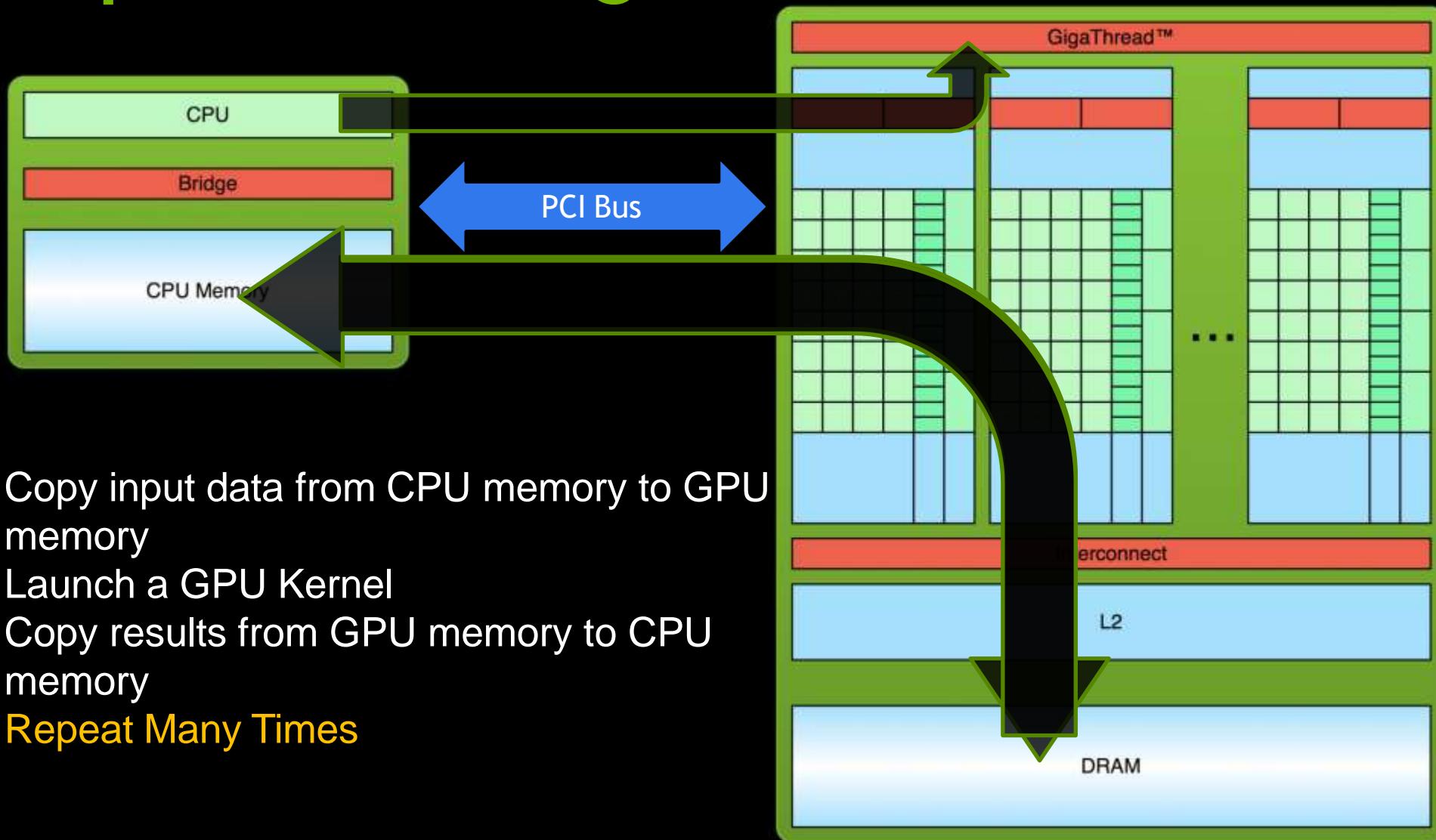
- Concurrent Streams

```
__global__ void printNumber(int number)
{
    printf("%d\n", number);
}
```

```
int main()
{
    for (int i = 0; i < 5; ++i)
    {
        cudaStream_t stream;
        cudaStreamCreate(&stream);
        printNumber<<<1, 1, 0, stream>>>(i);
        cudaStreamDestroy(stream);
    }
    cudaDeviceSynchronize();
}
```



# Simple Processing Flow

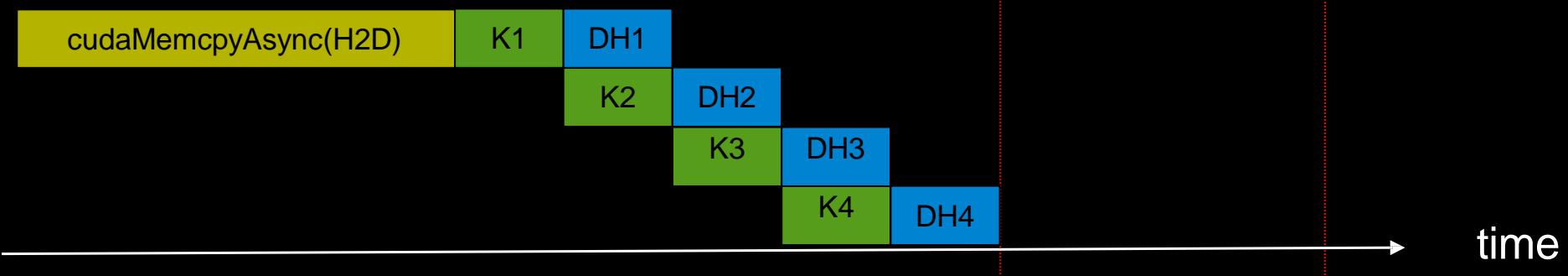


# CONCURRENCY THROUGH PIPELINING

- Serial



- Concurrent- overlap kernel and D2H copy

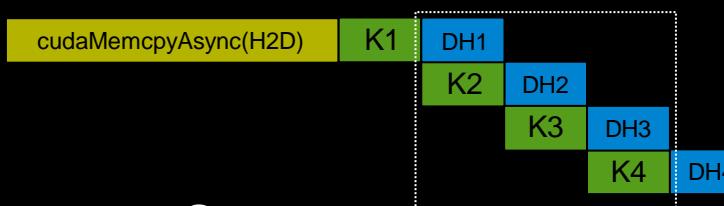


# CONCURRENCY THROUGH PIPELINING

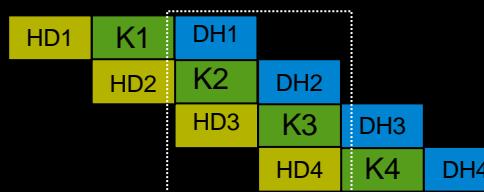
- Serial (1x)



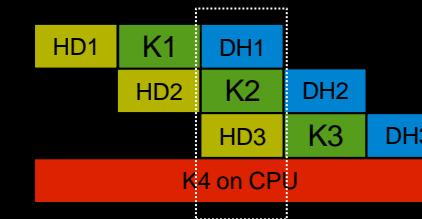
- 2-way concurrency (up to 2x)



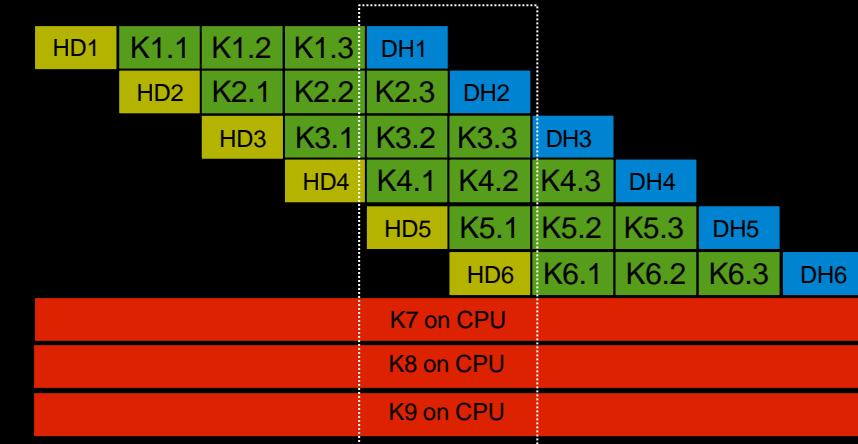
- 3-way concurrency (up to 3x)



- 4-way concurrency (3x+)



- 4+ way concurrency

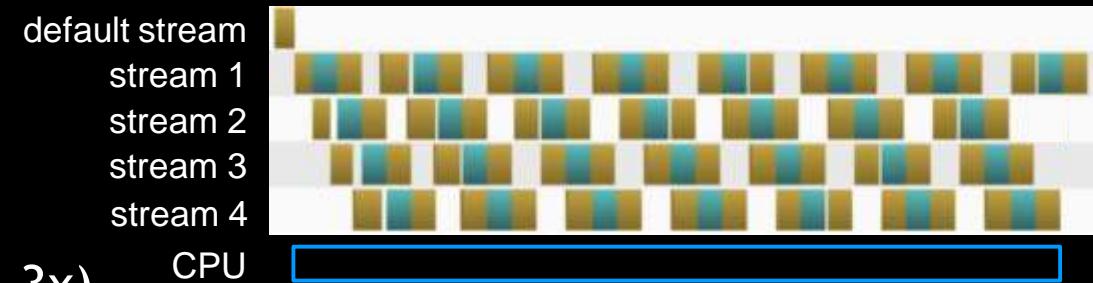


# EXAMPLE - TILED DGEMM

- CPU (dual 6 core SandyBridge E5-2667 @2.9 Ghz, MKL)
  - 222 Gflop/s
- GPU (K20X)
  - Serial: 519 Gflop/s (2.3x)
  - 2-way: 663 Gflop/s (3x)
  - 3-way: 990 Gflop/s (4x)
- GPU + CPU
  - 4-way con.: 1180 Gflop/s (5.3x)
- Obtain maximum performance by leveraging concurrency
- All PCI-E traffic is hidden
  - Effectively removes device memory size limitations!

DGEMM: m=n=16384, k=1408

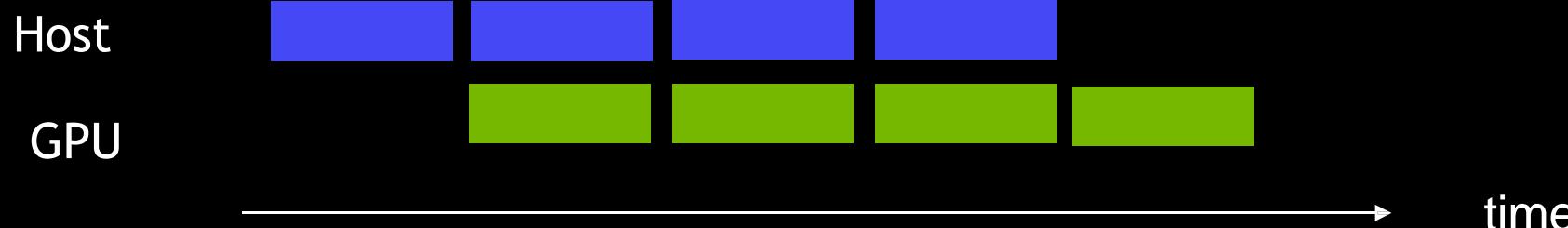
Nvidia Visual Profiler (nvvp)



# ENABLING CONCURRENCY WITH STREAMS

# SYNCHRONICITY IN CUDA

- All CUDA calls are either synchronous or asynchronous w.r.t the host
  - Synchronous: enqueue work and wait for completion
  - Asynchronous: enqueue work and return immediately
- Kernel Launches are asynchronous Automatic overlap with host



# CUDA STREAMS

- A **stream** is a queue of device work
  - The host places work in the queue and continues on immediately
  - Device schedules work from streams when resources are free
- CUDA operations are placed within a stream
  - e.g. Kernel launches, memory copies
- Operations within the **same stream** are **ordered** (FIFO) and cannot overlap
- Operations in **different streams** are **unordered** and can overlap

# MANAGING STREAMS

- **cudaStream\_t stream;**
  - Declares a stream handle
- **cudaStreamCreate (&stream) ;**
  - Allocates a stream
- **cudaStreamDestroy (stream) ;**
  - Deallocates a stream
  - Synchronizes host until work in stream has completed

# PLACING WORK INTO A STREAM

- Stream is the 4<sup>th</sup> launch parameter
  - kernel<<< blocks , threads, smem, **stream**>>>();
- Stream is passed into some API calls
  - cudaMemcpyAsync( dst, src, size, dir, **stream**);

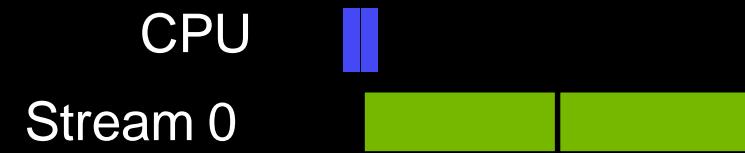
# DEFAULT STREAM

- Unless otherwise specified all calls are placed into a default stream
  - Often referred to as “Stream 0”
- Stream 0 has special synchronization rules
  - Synchronous with all streams
    - Operations in stream 0 cannot overlap other streams
- Exception: Streams with non-blocking flag set
  - `cudaStreamCreateWithFlags (&stream, cudaStreamNonBlocking)`
  - Use to get concurrency with libraries out of your control (e.g. MPI)

# KERNEL CONCURRENCY

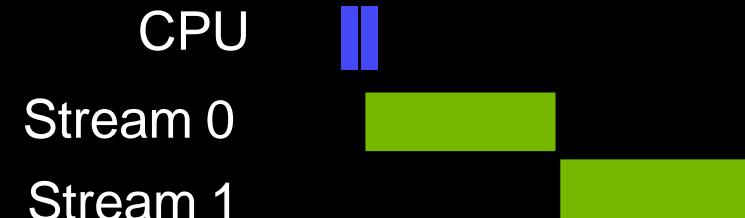
- Assume foo only utilizes 50% of the GPU
- Default stream

```
foo<<<blocks, threads>>>();  
foo<<<blocks, threads>>>();
```



- Default & user streams

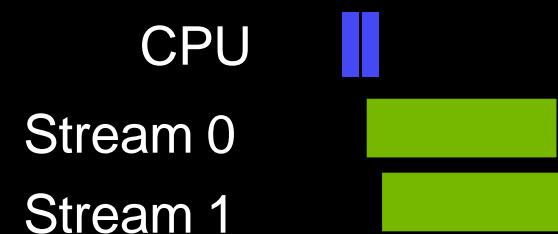
```
cudaStream_t stream1;  
cudaStreamCreate(&stream1);  
foo<<<blocks, threads>>>();  
foo<<<blocks, threads, 0, stream1>>>();  
cudaStreamDestroy(stream1);
```



# KERNEL CONCURRENCY

- Assume foo only utilizes 50% of the GPU
- Default & user streams

```
cudaStream_t stream1;  
  
cudaStreamCreateWithFlags (&stream1, cudaStreamNonBlocking);  
  
foo<<<blocks, threads>>>();  
  
foo<<<blocks, threads, 0, stream1>>>();  
  
cudaStreamDestroy (stream1);
```



# KERNEL CONCURRENCY

- Assume foo only utilizes 50% of the GPU User streams

```
cudaStream_t stream1, stream2;
```

```
cudaStreamCreate(&stream1);
```

```
cudaStreamCreate(&stream2);
```

```
foo<<<blocks, threads, 0, stream1>>>(); Stream 1
```

```
foo<<<blocks, threads, 0, stream2>>>(); Stream 2
```

```
cudaStreamDestroy(stream1); cudaStreamDestroy(stream2);
```

CPU ||



# REVIEW

- The host is automatically asynchronous with kernel launches
- Use streams to control asynchronous behavior
  - Ordered within a stream (FIFO)
  - Unordered with other streams
  - Default stream is synchronous with all streams.

# Concurrent Memory Copies

# CONCURRENT MEMORY COPIES

- First we must review CUDA memory

# THREE TYPES OF MEMORY

- Device Memory
  - Allocated using `cudaMalloc`
  - Cannot be paged
- Pageable Host Memory
  - Default allocation (e.g. `malloc`, `calloc`, `new`, etc)
  - Can be paged in and out by the OS
- Pinned (Page-Locked) Host Memory
  - Allocated using special allocators
  - Cannot be paged out by the OS

# ALLOCATING PINNED MEMORY

- **cudaMallocHost(...)** / **cudaHostAlloc(...)**
  - Allocate/Free pinned memory on the host
  - Replaces malloc/free/new
- **cudaFreeHost(...)**
  - Frees memory allocated by cudaMallocHost or cudaHostAlloc
- **cudaHostRegister(...)** / **cudaHostUnregister(...)**
  - Pins/Unpins pageable memory (making it pinned memory)
  - Slow so don't do often
- Why pin memory?
  - Pagable memory is transferred using the host CPU
  - Pinned memory is transferred using the DMA engines
    - Frees the CPU for asynchronous execution
    - Achieves a higher percent of peak bandwidth

# CONCURRENT MEMORY COPIES

- **cudaMemcpy( . . . )**
  - Places transfer into default stream
  - Synchronous: Must complete prior to returning
- **cudaMemcpyAsync( . . . , &stream)**
  - Places transfer into stream and returns immediately
- To achieve concurrency
  - Transfers must be in a non-default stream
  - Must use async copies
  - 1 transfer per direction at a time
  - Memory on the host must be pinned

# PAGED MEMORY EXAMPLE

```
int *h_ptr, *d_ptr;  
  
h_ptr=malloc(bytes);  
cudaMalloc(&d_ptr,bytes);  
  
cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);  
  
free(h_ptr);  
cudaFree(d_ptr);
```

# PINNED MEMORY: EXAMPLE 1

```
int *h_ptr, *d_ptr;  
  
cudaMallocHost(&h_ptr,bytes);  
cudaMalloc(&d_ptr,bytes);  
  
cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);  
  
cudaFreeHost(h_ptr);  
cudaFree(d_ptr);
```

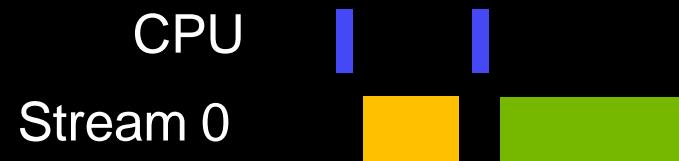
## PINNED MEMORY: EXAMPLE 2

```
int *h_ptr, *d_ptr;  
  
h_ptr=malloc(bytes) ;  
cudaHostRegister(h_ptr,bytes,0) ;  
cudaMalloc(&d_ptr,bytes) ;  
  
cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice) ;  
  
cudaHostUnregister(h_ptr) ;  
free(h_ptr) ; cudaFree(d_ptr) ;
```

# CONCURRENCY EXAMPLES

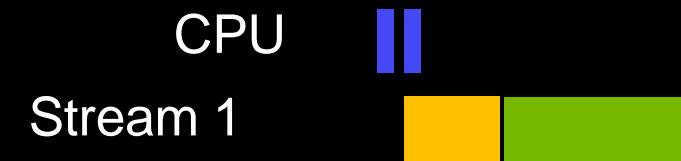
## Synchronous

```
cudaMemcpy(...);  
foo<<<...>>>();
```



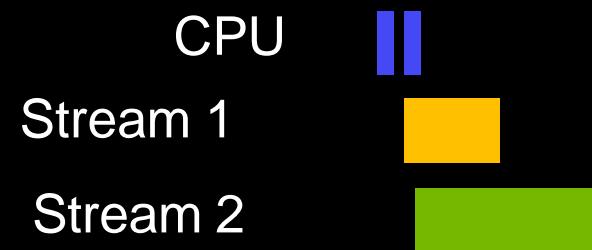
## Asynchronous Same Stream

```
cudaMemcpyAsync(..., stream1);  
foo<<<..., stream1>>>();
```



## Asynchronous Different Streams

```
cudaMemcpyAsync(..., stream1);  
foo<<<..., stream2>>>();
```



# REVIEW

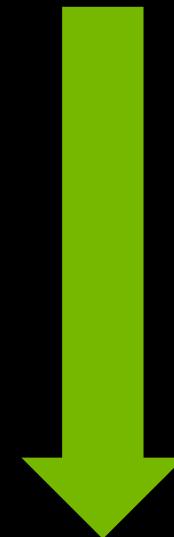
- Memory copies can execute concurrently if (and only if)
  - The memory copy is in a different non-default stream
  - The copy uses pinned memory on the host
  - The asynchronous API is called
  - There isn't another memory copy occurring in the same direction at the same time.

# Synchronization

# SYNCHRONIZATION APIs

- Synchronize everything
  - `cudaDeviceSynchronize()`
    - Blocks host until all issued CUDA calls are complete
- Synchronize host w.r.t. a specific stream
  - `cudaStreamSynchronize( stream)`
    - Blocks host until all issued CUDA calls in stream are complete
- Synchronize host or devices using events

More  
Synchronization



Less  
Synchronization

# CUDA EVENTS

- Provide a mechanism to signal when operations have occurred in a stream
  - Useful for profiling and synchronization
- Events have a boolean state:
  - Occurred
  - Not Occurred
  - **Important: Default state = occurred**

# MANAGING EVENTS

- **cudaEventCreate (&event)**
  - Creates an event
- **cudaEventDestroy (&event)**
  - Destroys an event
- **cudaEventCreateWithFlags (&ev, cudaEventDisableTiming)**
  - Disables timing to increase performance and avoid synchronization issues
- **cudaEventRecord (&event, stream)**
  - Set the event state to not occurred
  - Enqueue the event into a stream
  - Event state is set to occurred when it reaches the front of the stream

# SYNCHRONIZATION USING EVENTS

- Synchronize using events
  - `cudaEventQuery` ( event )
    - Returns CUDA\_SUCCESS if an event has occurred
  - `cudaEventSynchronize` ( event )
    - Blocks host until stream completes all outstanding calls
  - `cudaStreamWaitEvent` ( stream, event )
    - Blocks stream until event occurs
    - Only blocks launches after this call
    - Does not block the host!
- Common multi-threading mistake:
  - Calling `cudaEventSynchronize` before `cudaEventRecord`

# CUDA\_LAUNCH\_BLOCKING

- Environment variable which forces synchronization
  - `export CUDA_LAUNCH_BLOCKING=1`
  - All CUDA operations are synchronous w.r.t the host
- Useful for debugging race conditions
  - If it runs successfully with `CUDA_LAUNCH_BLOCKING` set but doesn't without you have a race condition.

# REVIEW

- Synchronization with the host can be accomplished via
  - `cudaDeviceSynchronize()`
  - `cudaStreamSynchronize(stream)`
  - `cudaEventSynchronize(event)`
- Synchronization between streams can be accomplished with
  - `cudaStreamWaitEvent(stream,event)`
- Use `CUDA_LAUNCH_BLOCKING` to identify race conditions

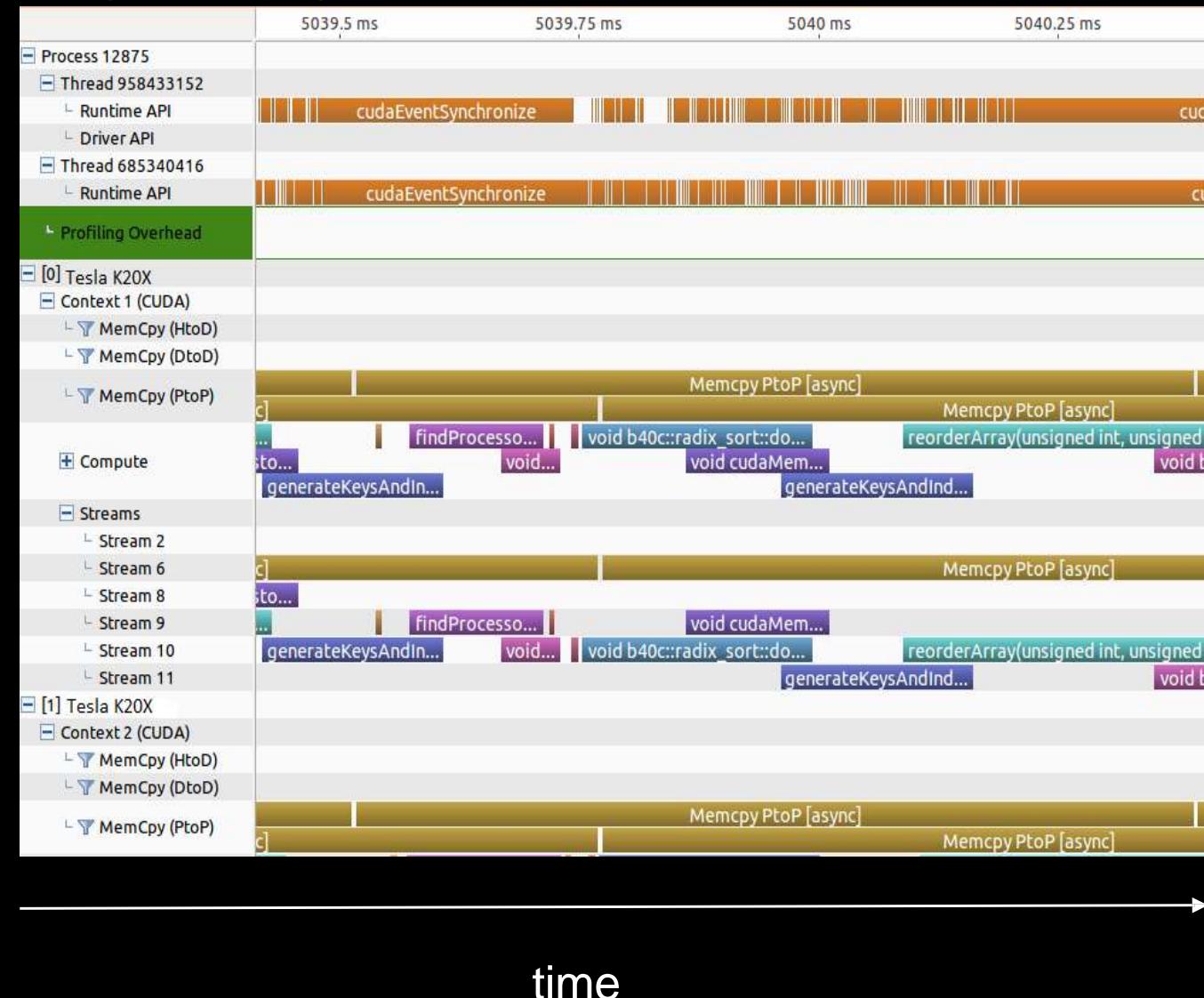
# Streaming Performance

# PROFILING TOOLS

- Windows
  - Nsight Visual Studio Edition
  - NVIDIA Visual Profiler
- Linux, Mac
  - Nsight Eclipse Edition
  - NVIDIA Visual Profiler
  - nvprof

# NVVP PROFILER TIMELINE

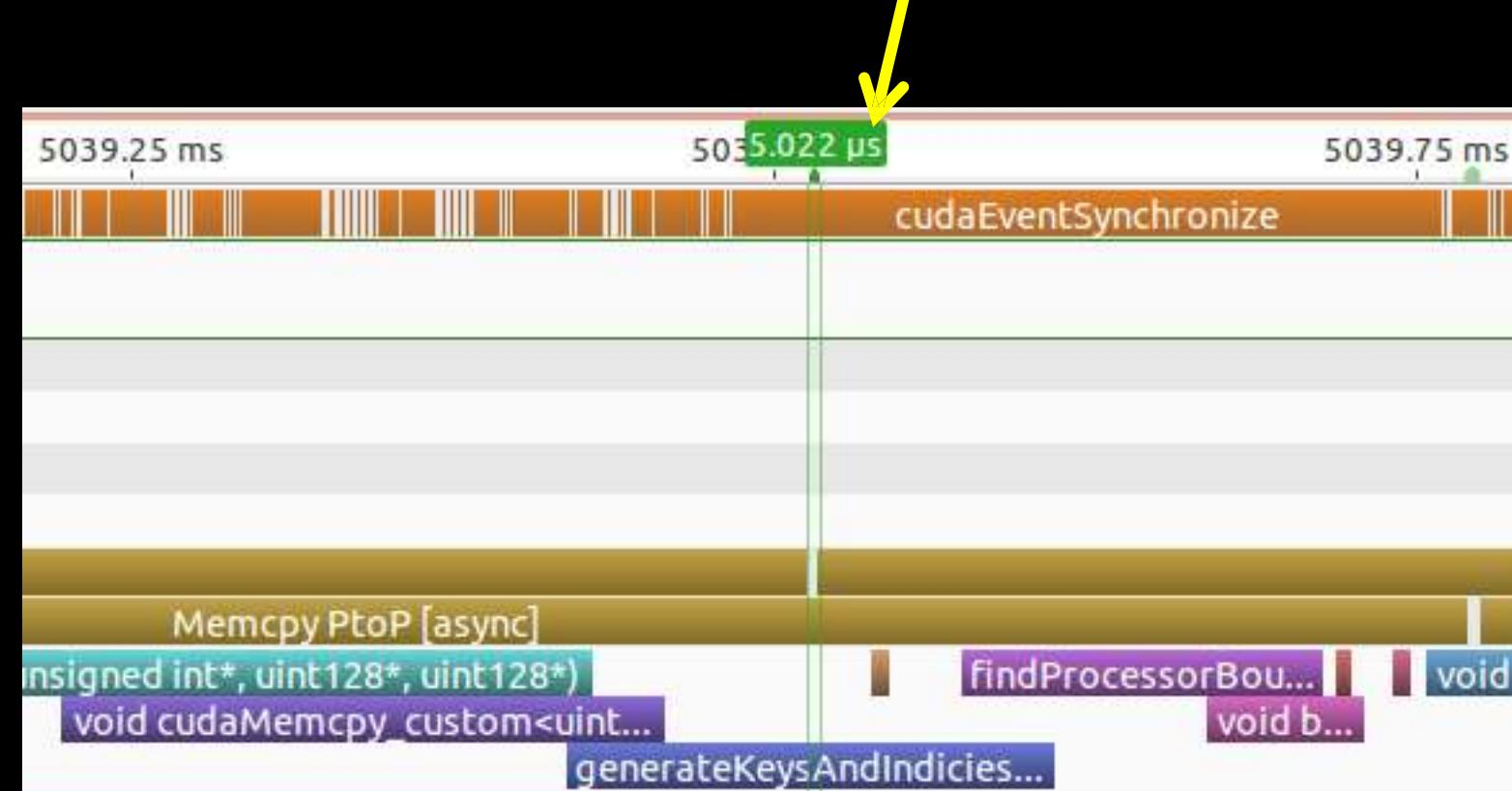
- Host API Calls
- Multi-threaded
- Multi-GPU
- Multi-process
- Kernels
- Memory copies
- Streams



# OPTIMAL TIMELINE

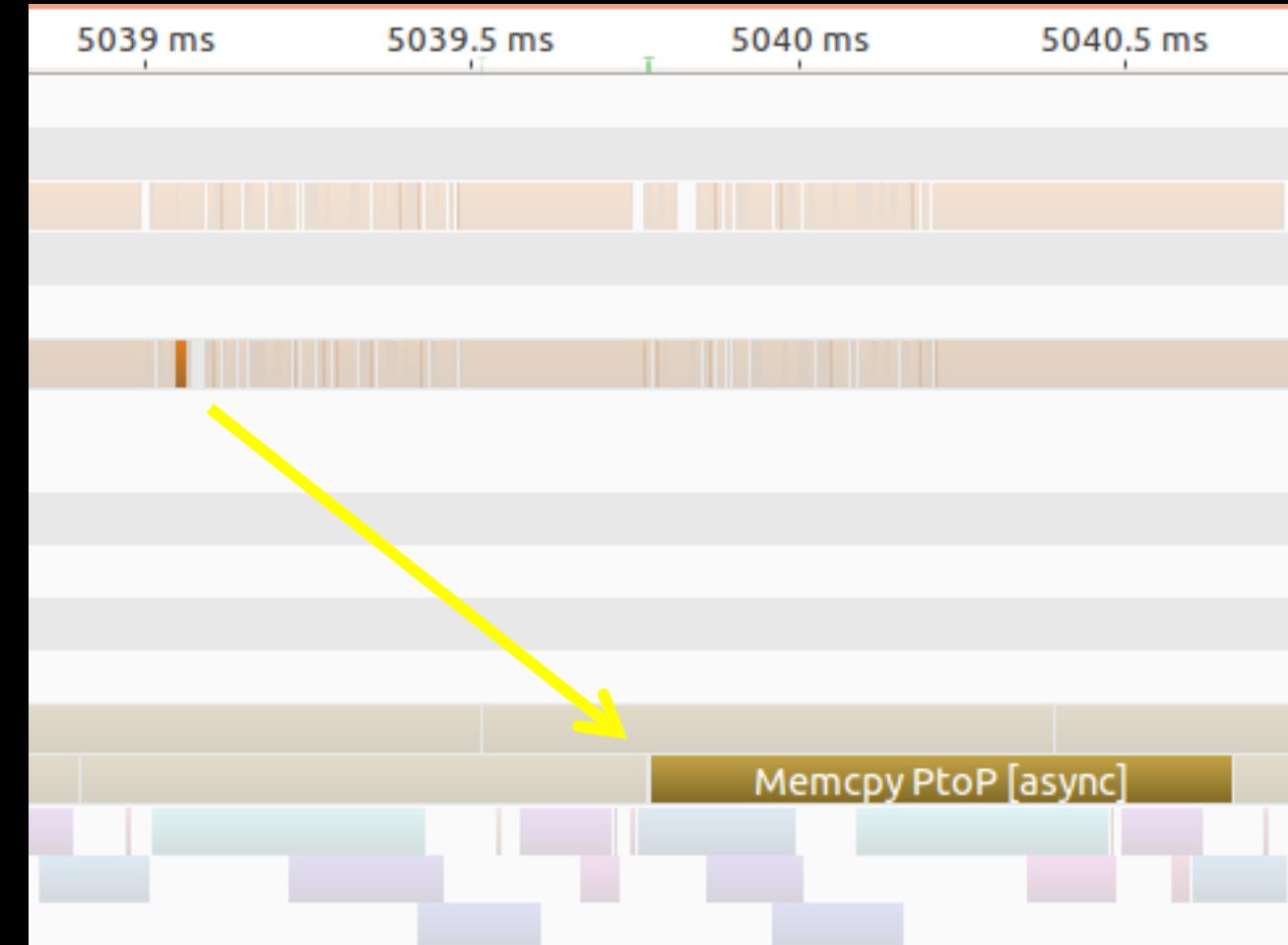
Less than 10 us idle time between successive operations

Concurrent  
Operations



# OPTIMAL TIMELINE

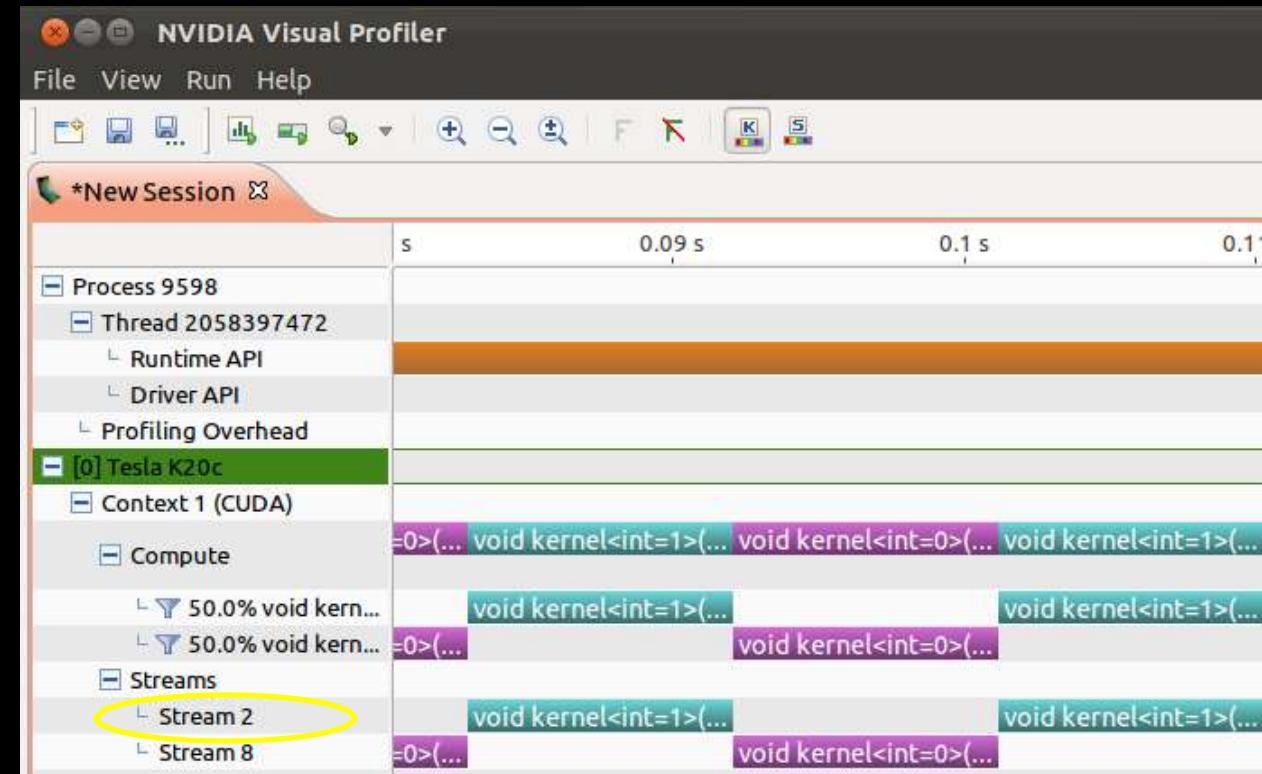
Host is running ahead of  
the device >30 us



# COMMON STREAMING PROBLEMS

# CASE STUDY 1-A

```
for(int i=0;i<repeat;i++)  
{  
    kernel<<<1,1,0,stream1>>>();  
    kernel<<<1,1>>>();  
}
```

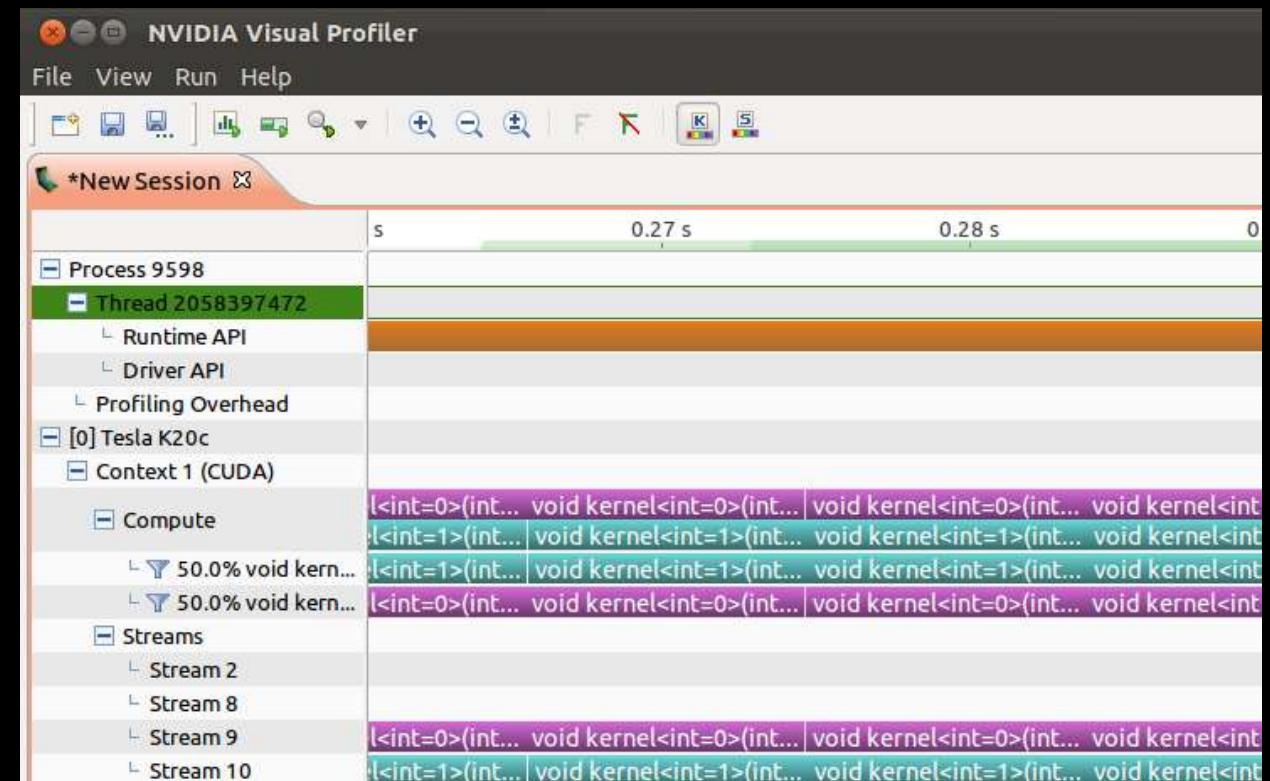


**Problem:**  
One kernel is in the default stream

Stream 2 is the  
default stream

# CASE STUDY 1-A

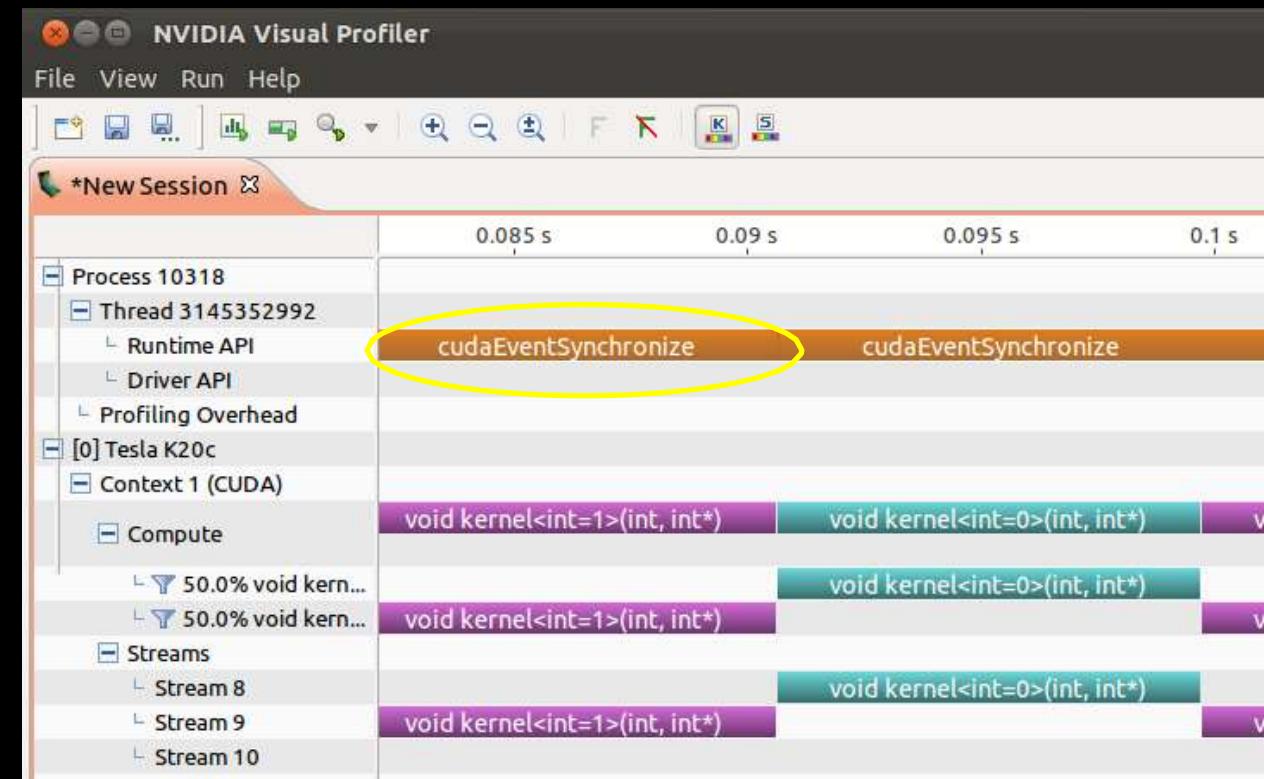
```
for(int i=0;i<repeat;i++) {  
    kernel<<<1,1,0,stream1>>>();  
    kernel<<<1,1,0,stream2>>>();  
}
```



**Solution:**  
Place each kernel in its own stream

# CASE STUDY 1-B

```
for(int i=0;i<repeat;i++) {  
    kernel<<<1,1,0,stream1>>>();  
    cudaEventRecord(event1);  
    kernel<<<1,1,0,stream2>>>();  
    cudaEventRecord(event2);  
  
    cudaEventSynchronize(event1);  
    cudaEventSynchronize(event2);  
}
```



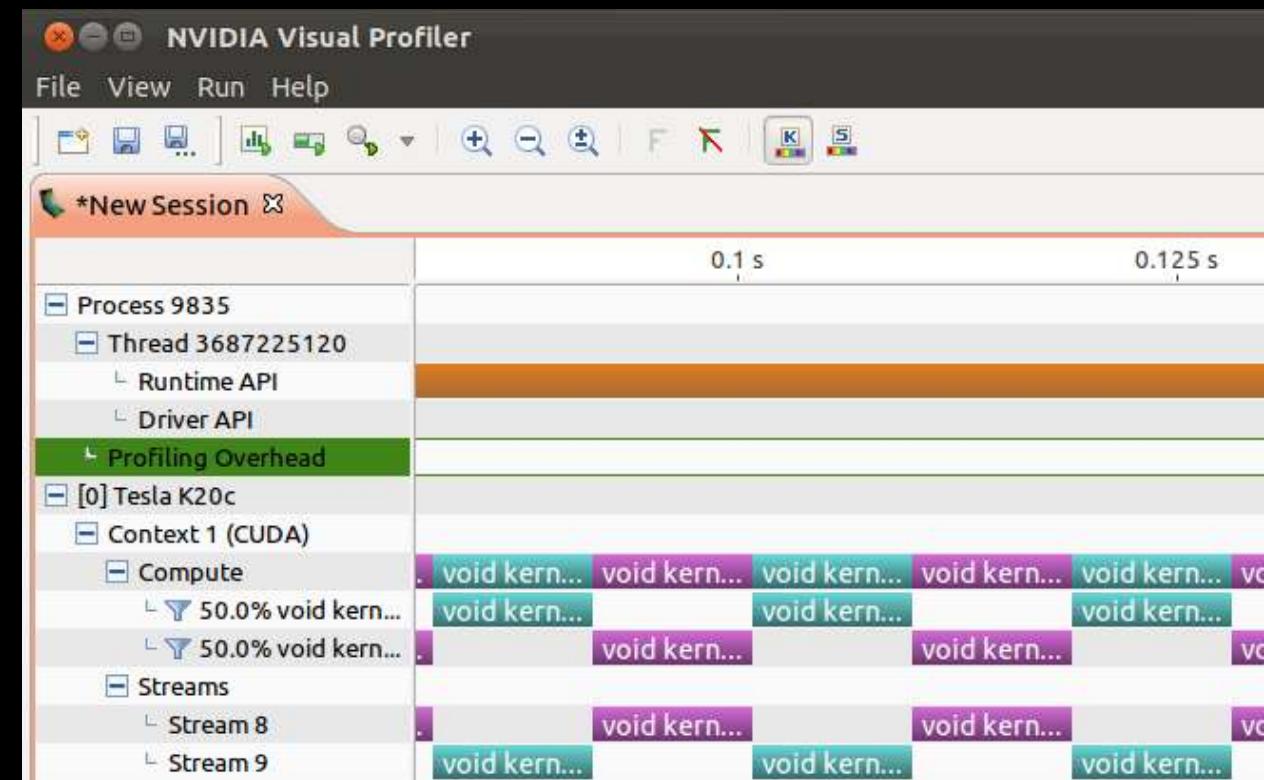
Are events causing the problem?

# CASE STUDY 1-B

```
for(int i=0;i<repeat;i++) {  
    kernel<<<1,1,0,stream1>>>();  
    cudaEventRecord(event1);  
    kernel<<<1,1,0,stream2>>>();  
    cudaEventRecord(event2);  
  
    cudaEventSynchronize(event1);  
    cudaEventSynchronize(event2);  
}
```

Problem:

cudaEventRecord by without a stream goes into the default stream

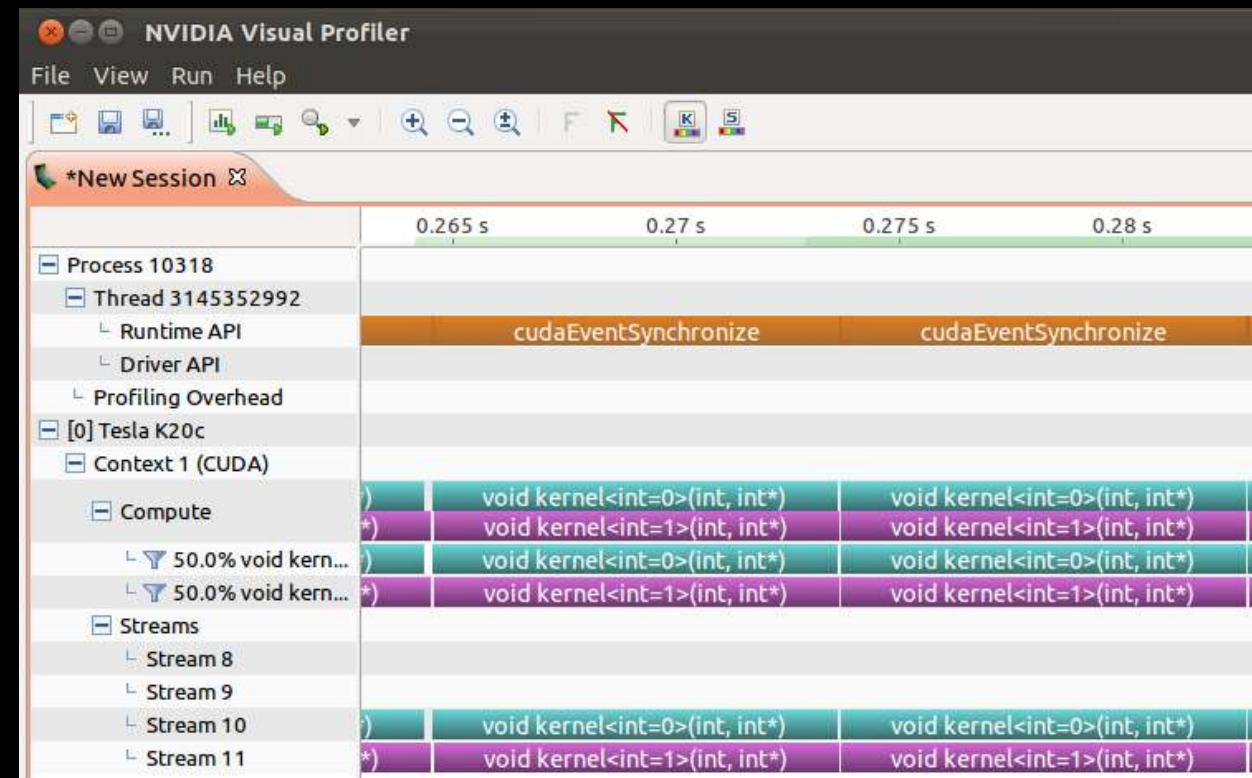


# CASE STUDY 1-B

```
for(int i=0;i<repeat;i++) {  
    kernel<<<1,1,0,stream1>>>();  
    cudaEventRecord(event1,stream1);  
    kernel<<<1,1,0,stream2>>>();  
    cudaEventRecord(event2,stream2);  
  
    cudaEventSynchronize(event1);  
    cudaEventSynchronize(event2);  
}
```

Solution:

Record events into non-default streams



# PROBLEM 1: USING THE DEFAULT STREAM

## ■ Symptoms

- One stream will not overlap other streams
  - In Cuda 5.0 stream 2 = default stream
- Search for `cudaEventRecord(event)` , `cudaMemcpyAsync()`, etc.
  - If stream is not specified it is placed into the default stream
- Search for kernel launches in the default stream
  - `<<<a,b>>>`

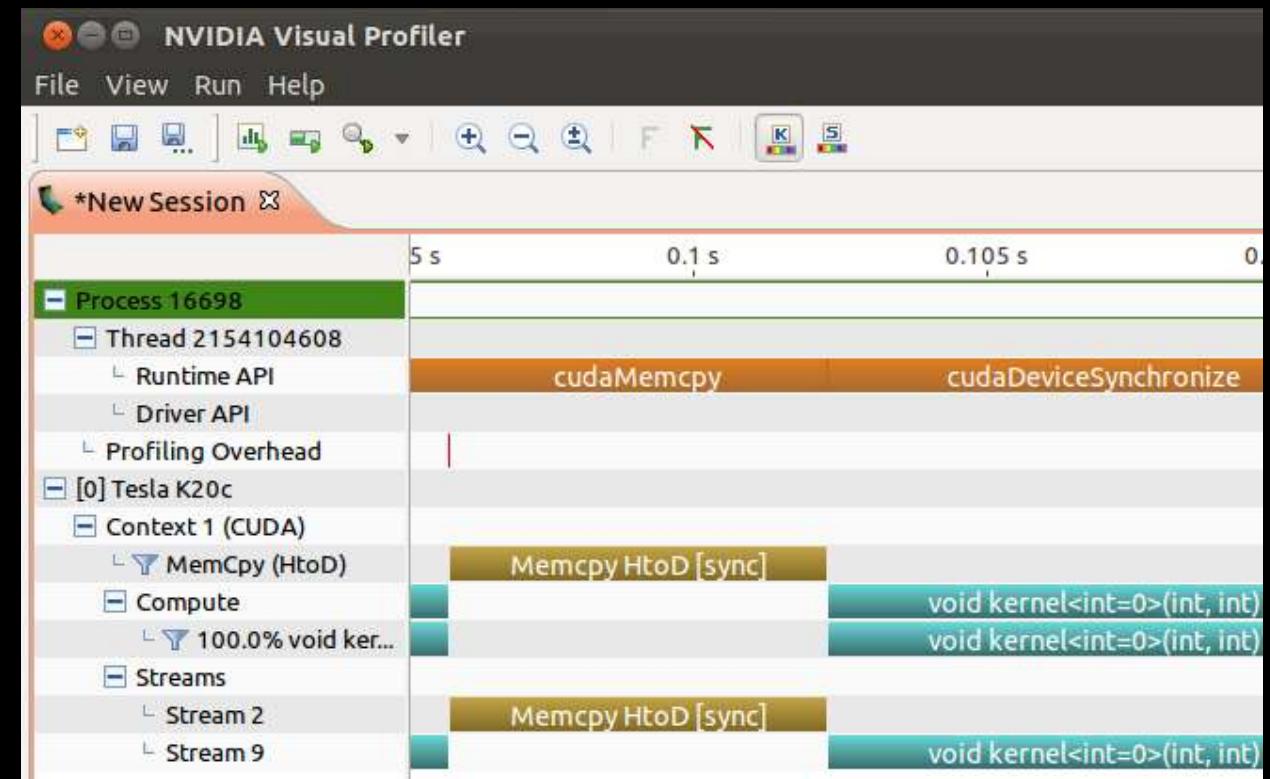
## ■ Solutions

- Move work into a non-default stream
- `cudaEventRecord(event,stream)`, `cudaMemcpyAsync(...,stream)`
- Alternative: Allocate other streams as non-blocking streams

# CASE STUDY 2-A

```
for(int i=0;i<repeat;i++) {  
    cudaMemcpy(d_ptr,h_ptr,bytes, cudaMemcpyHostToDevice);  
    kernel<<<1,1,0,stream2>>>();  
    cudaDeviceSynchronize();  
}
```

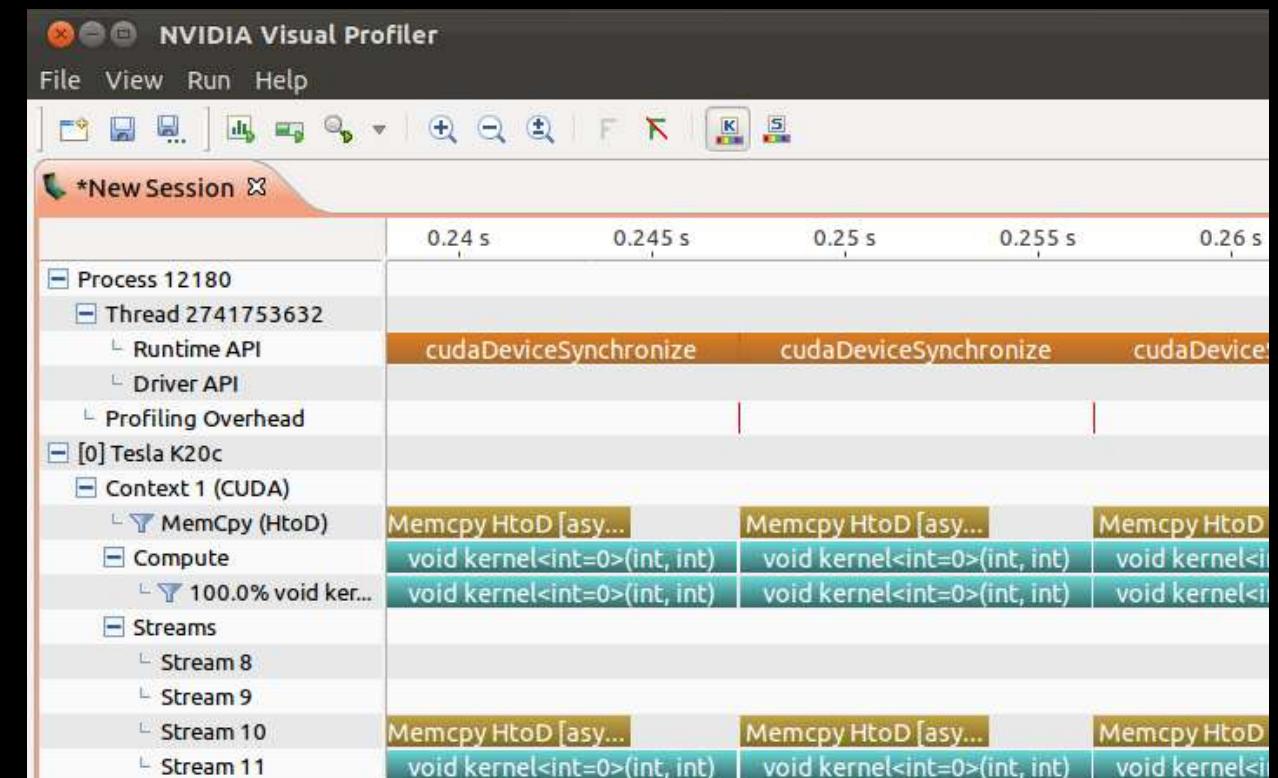
**Problem:**  
**Memory copy is synchronous**



# CASE STUDY 2-A

```
for(int i=0;i<repeat;i++) {  
    cudaMemcpyAsync(d_ptr,h_ptr,bytes, cudaMemcpyHostToDevice, stream1);  
    kernel<<<1,1,0,stream2>>>();  
    cudaDeviceSynchronize();  
}
```

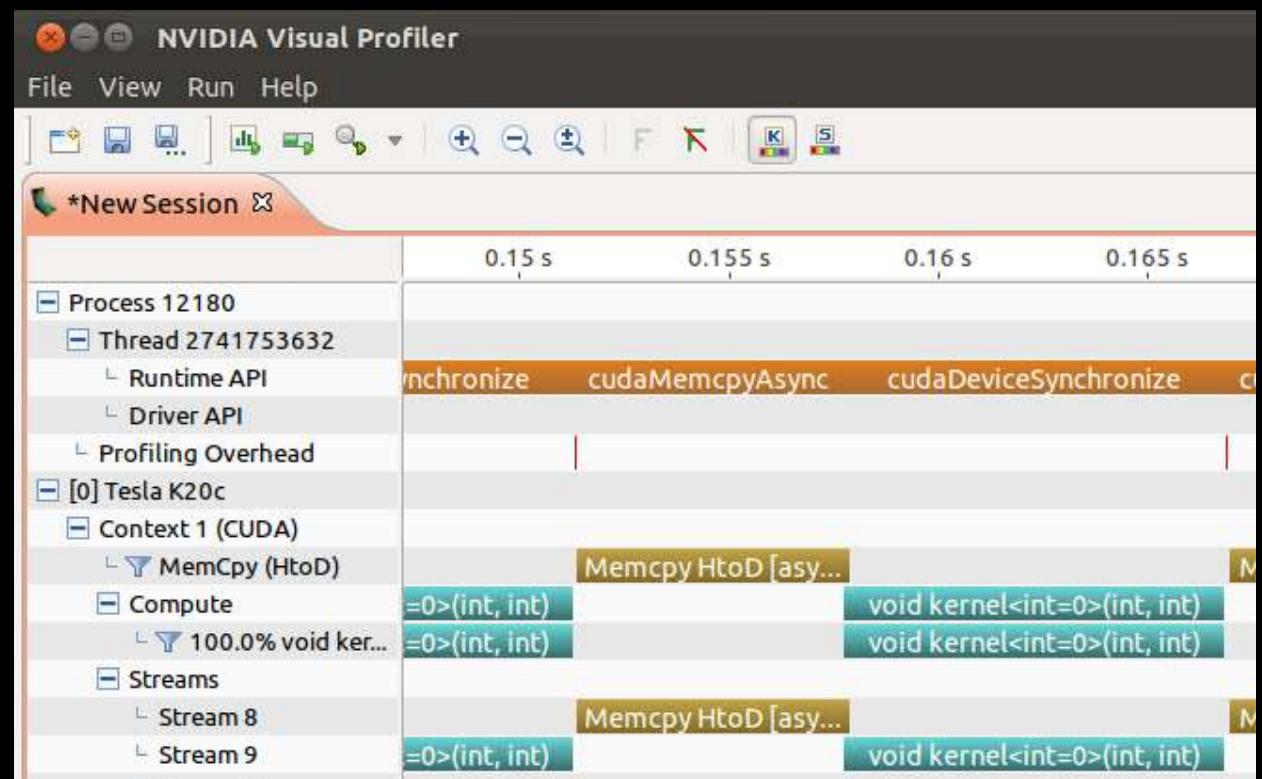
**Solution:**  
**Use asynchronous API**



# CASE STUDY 2-B

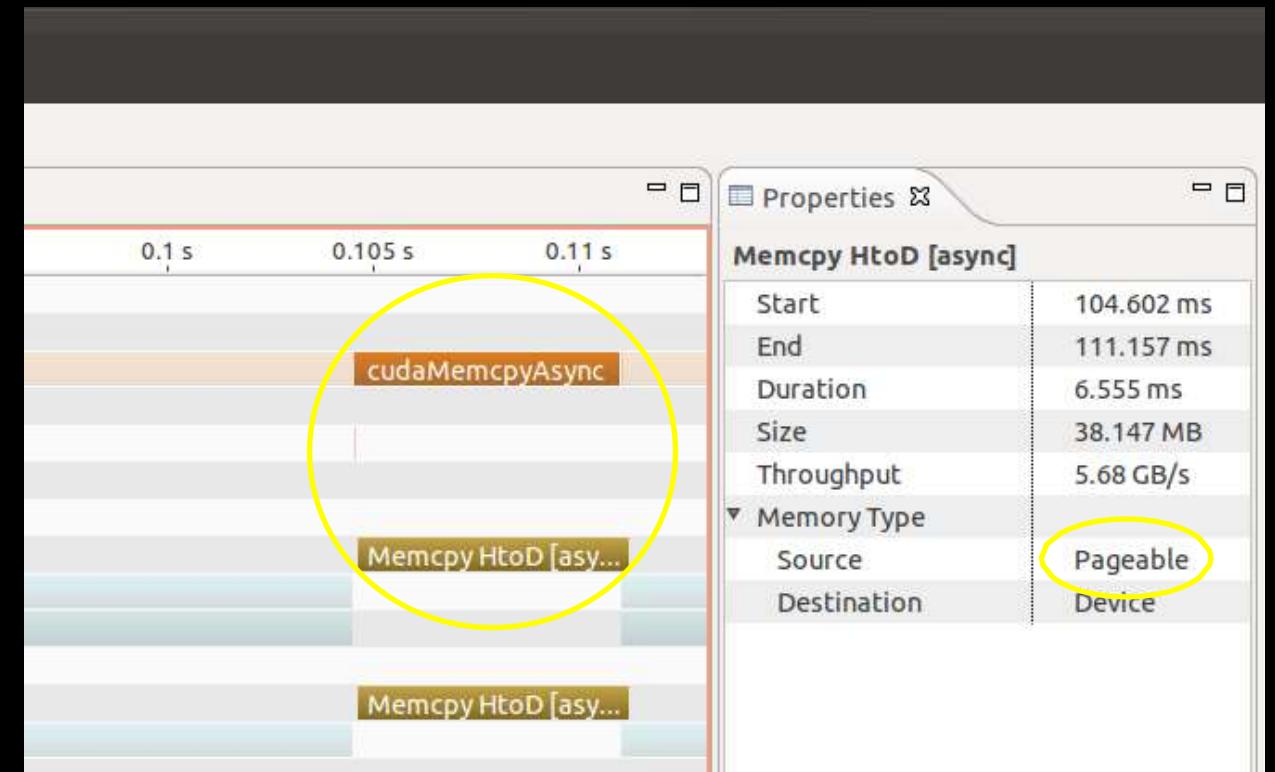
```
for(int i=0;i<repeat;i++) {  
    cudaMemcpyAsync(d_ptr,h_ptr,bytes, cudaMemcpyHostToDevice, stream1);  
    kernel<<<1,1,0,stream2>>>();  
    cudaDeviceSynchronize();  
}
```

Problem: ??



# CASE STUDY 2-B

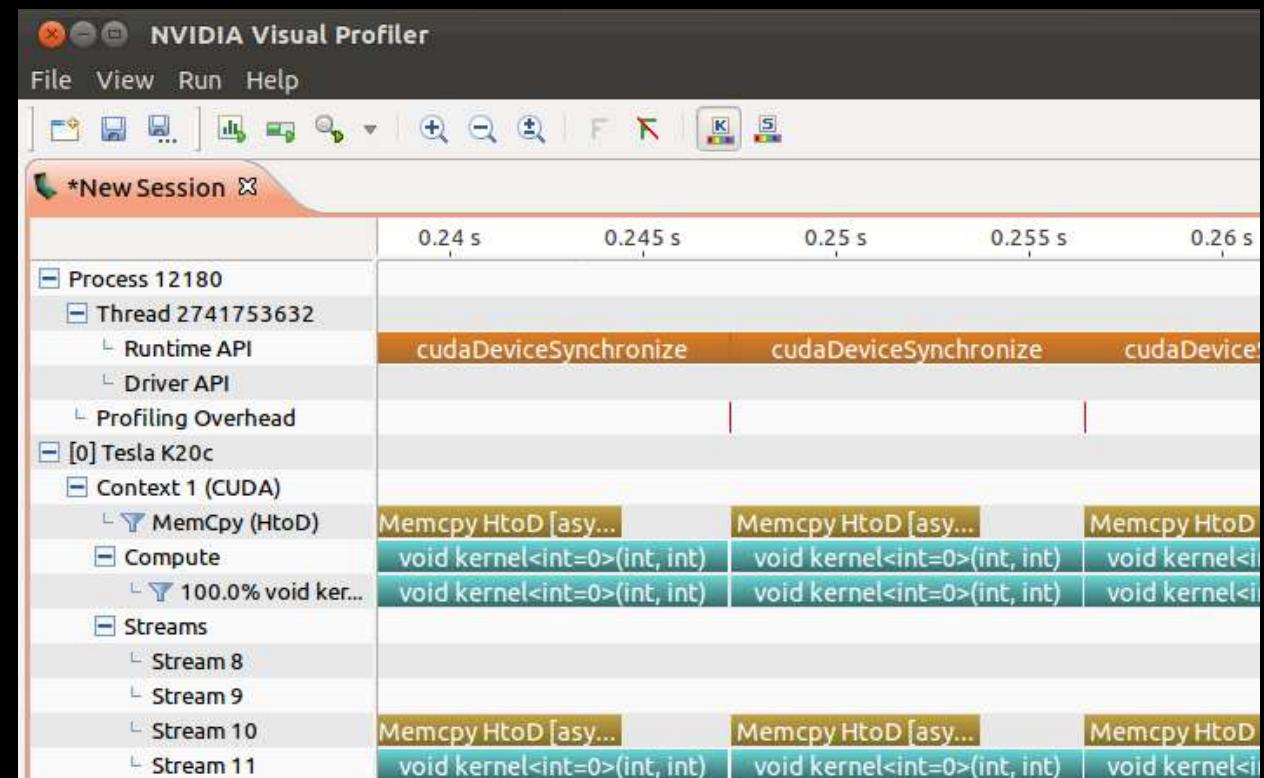
```
for(int i=0;i<repeat;i++) {  
    cudaMemcpyAsync(d_ptr,h_ptr,bytes, cudaMemcpyHostToDevice, stream1);  
    kernel<<<1,1,0,stream2>>>();  
    cudaDeviceSynchronize();  
}
```



Host doesn't get ahead  
Cuda 5.5 reports “Pageable” type

# CASE STUDY 2-B

```
cudaHostRegister(h_ptr,bytes,0);
for(int i=0;i<repeat;i++) {
    cudaMemcpyAsync(d_ptr,h_ptr,bytes, cudaMemcpyHostToDevice, stream1);
    kernel<<<1,1,0,stream2>>>();
    cudaDeviceSynchronize();
}
cudaHostUnregister(h_ptr);
```



Solution:

Pin host memory using `cudaHostRegister` or `cudaMallocHost`

## PROBLEM 2: MEMORY TRANSFERS ISSUES

- Symptoms

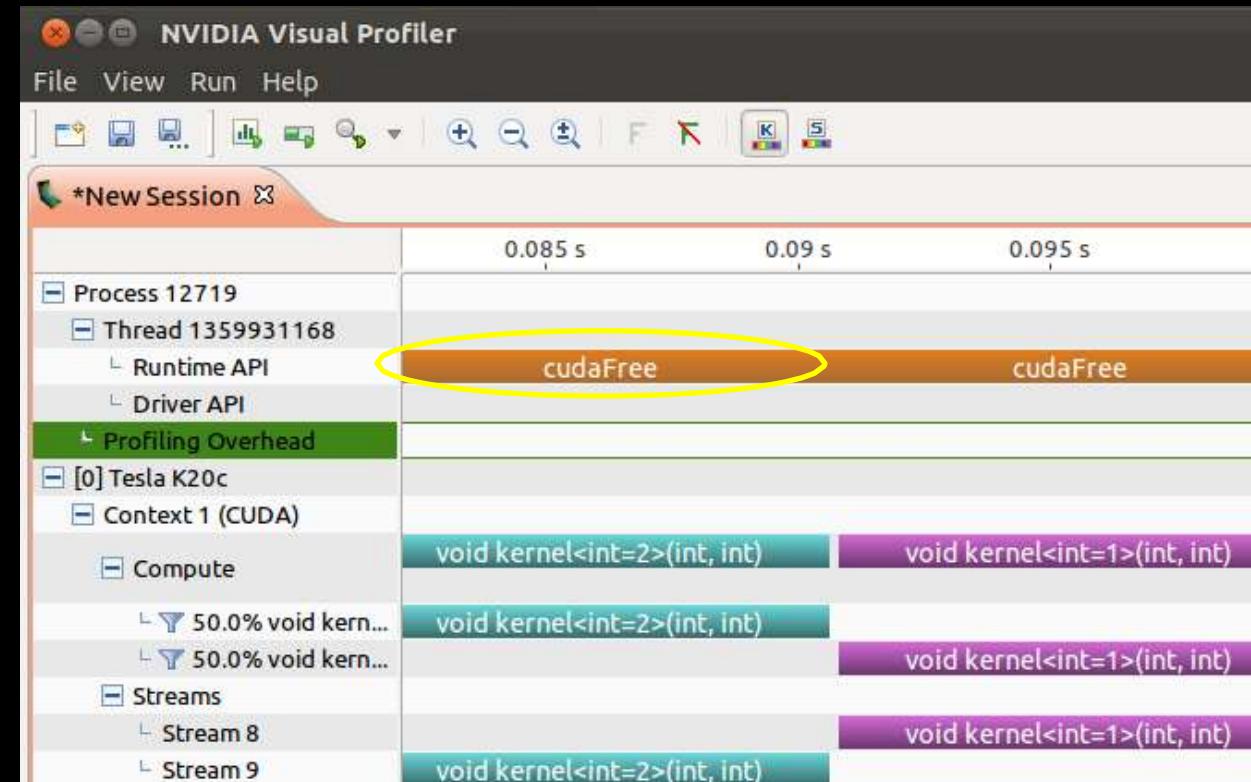
- Memory copies do not overlap
- Host spends excessive time in memory copy API
- Cuda reports “Pageable” memory (Cuda 5.5+)

- Solutions

- Use asynchronous memory copies
- Use pinned memory for host memory
  - `cudaMallocHost` or `cudaHostRegister`

# CASE STUDY 3

```
void launchwork(cudaStream_t stream) {  
    int *mem;  
    cudaMalloc(&mem,bytes);  
    kernel<<<1,1,0,stream>>>(mem);  
    cudaFree(mem);  
}  
...  
  
for(int i=0;i<repeat;i++) {  
    launchwork(stream1);  
    launchwork(stream2);  
}
```



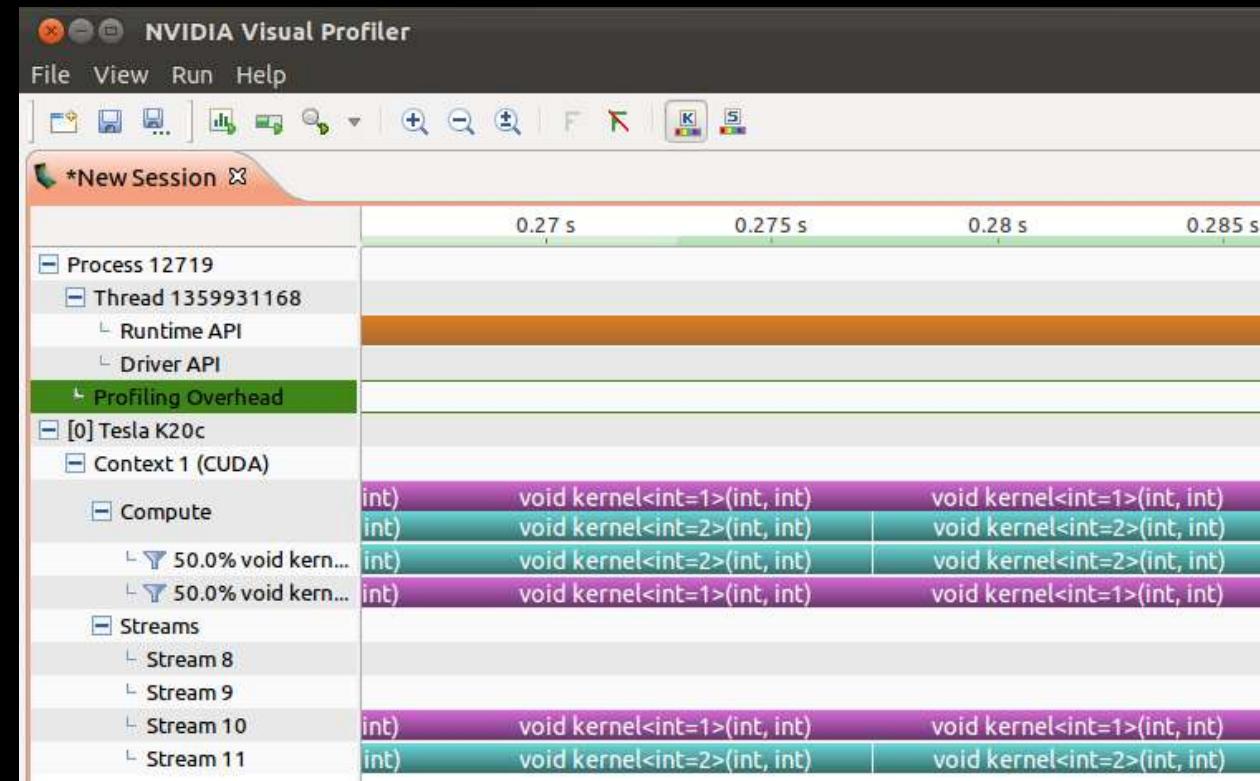
**Problem:**

Allocation & deallocation synchronize the device

Host blocked in allocation/free

# CASE STUDY 3

```
void launchwork(cudaStream_t stream, int *mem) {  
    kernel<<<1,1,0,stream>>>(mem);  
}  
...  
  
for(int i=0;i<repeat;i++) {  
    launchwork<1>(stream1,mem1);  
    launchwork<2>(stream2,mem2);  
}
```



**Solution:**

Reuse cuda memory and objects including streams and events

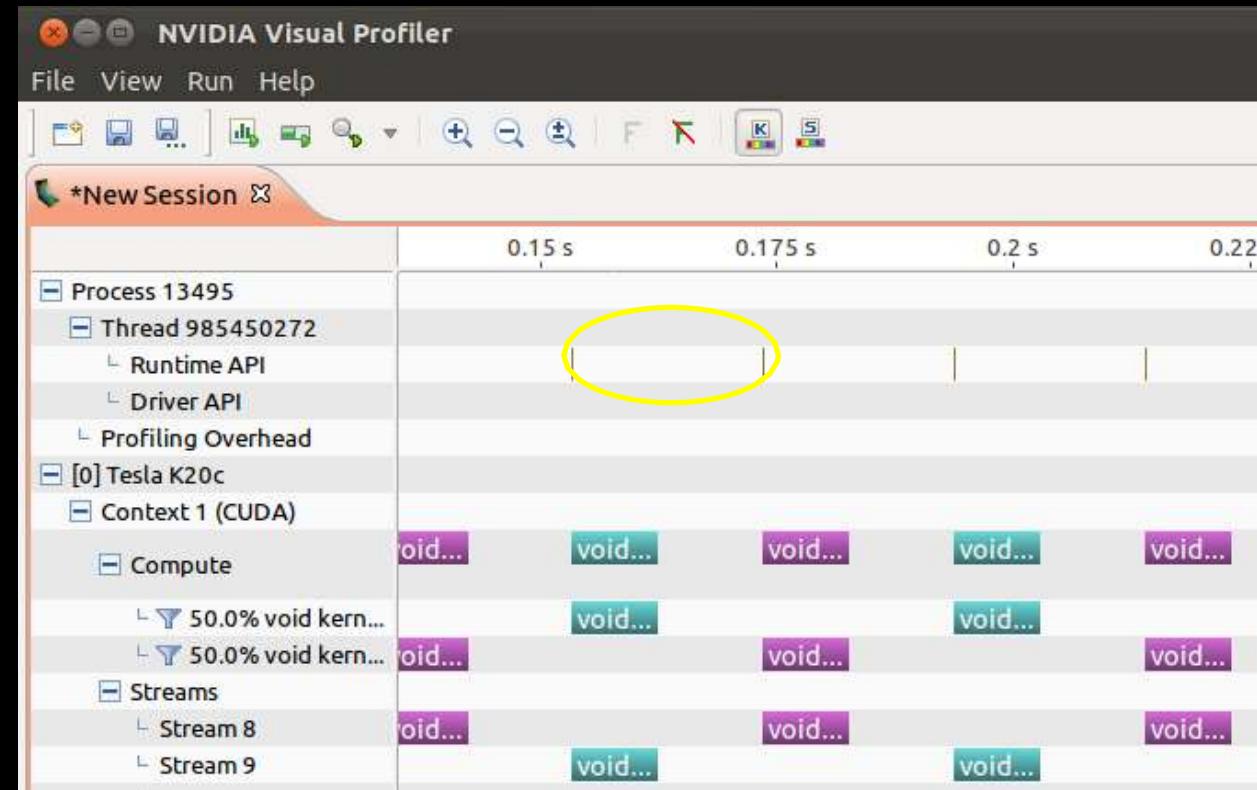
# PROBLEM 3: IMPLICIT SYNCHRONIZATION

- Symptoms
  - Host does not get ahead
  - Host shows excessive time in certain API calls
    - `cudaMalloc`, `cudaFree`, `cudaEventCreate`, `cudaEventDestroy`, `cudaStreamCreate`,  
`cudaStreamCreate`, `cudaHostRegister`, `cudaHostUnregister`,  
`cudaFuncSetCacheConfig`
- Solution:
  - Reuse memory and data structures

# CASE STUDY 4

```
for(int i=0;i<repeat;i++)  
{  
    hostwork();  
    kernel<<<1,1,0,stream1>>>();  
    hostwork();  
    kernel<<<1,1,0,stream2>>>();  
}
```

**Problem:**  
**Host is limiting performance**



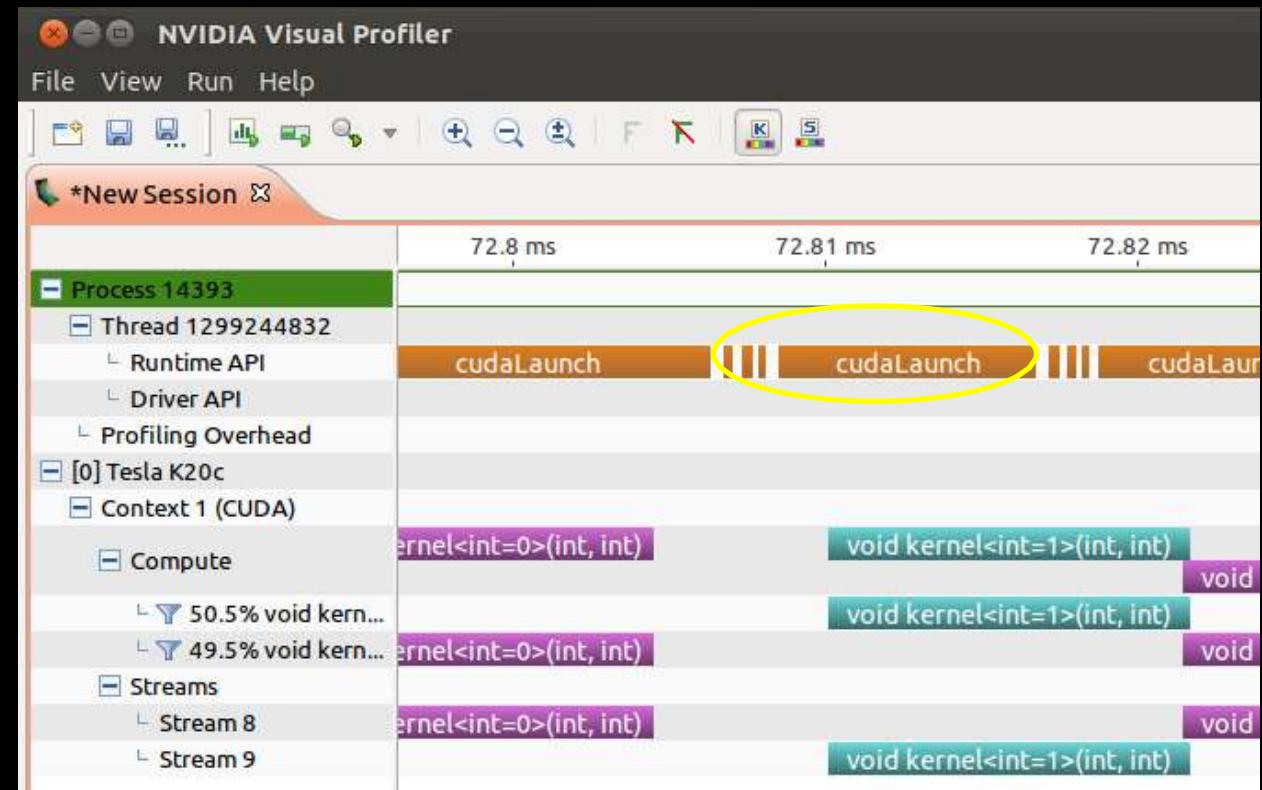
Host is outside of API calls

## PROBLEM 4: LIMITED BY HOST

- Symptoms
  - Host is outside of cuda APIs
  - Large gaps in timeline where the host and device are empty
- Solution
  - Move more work to the GPU
  - Multi-thread host code

# CASE STUDY 5

```
for(int i=0;i<repeat;i++)
{
    kernel<<<1,1,0,stream1>>>();
    kernel<<<1,1,0,stream2>>>();
}
```

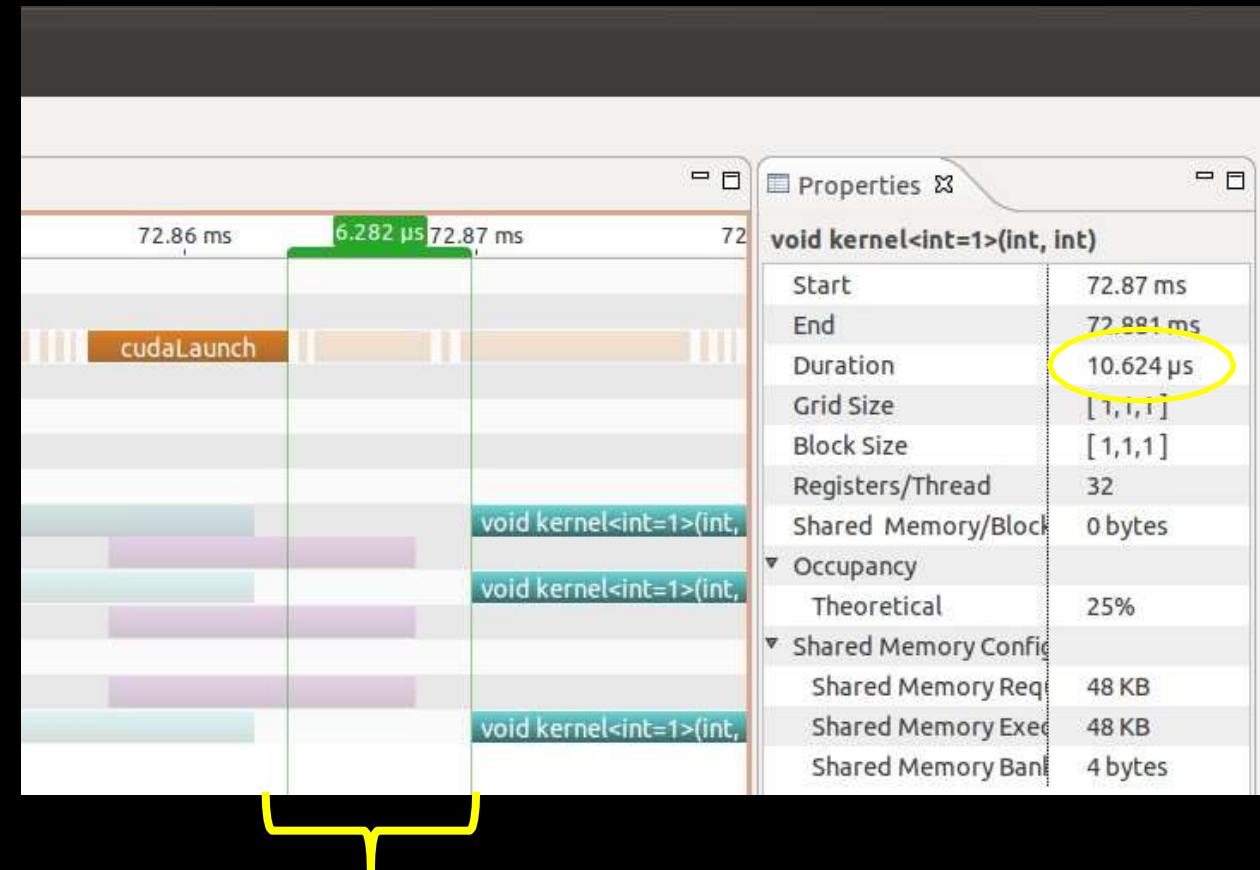


Host is in cudaLaunch or other APIs

# CASE STUDY 5

```
for(int i=0;i<repeat;i++)
{
    kernel<<<1,1,0,stream1>>>();
    kernel<<<1,1,0,stream2>>>();
}
```

**Problem:**  
**Not enough work to cover launch overhead**



Host is not far ahead  
Kernel runtime is short (<30us)

# PROBLEM 5: LIMITED BY LAUNCH OVERHEAD

- Symptoms

- Host does not get ahead
  - Kernels are short <30 us
  - Time between successive kernels is >10 us

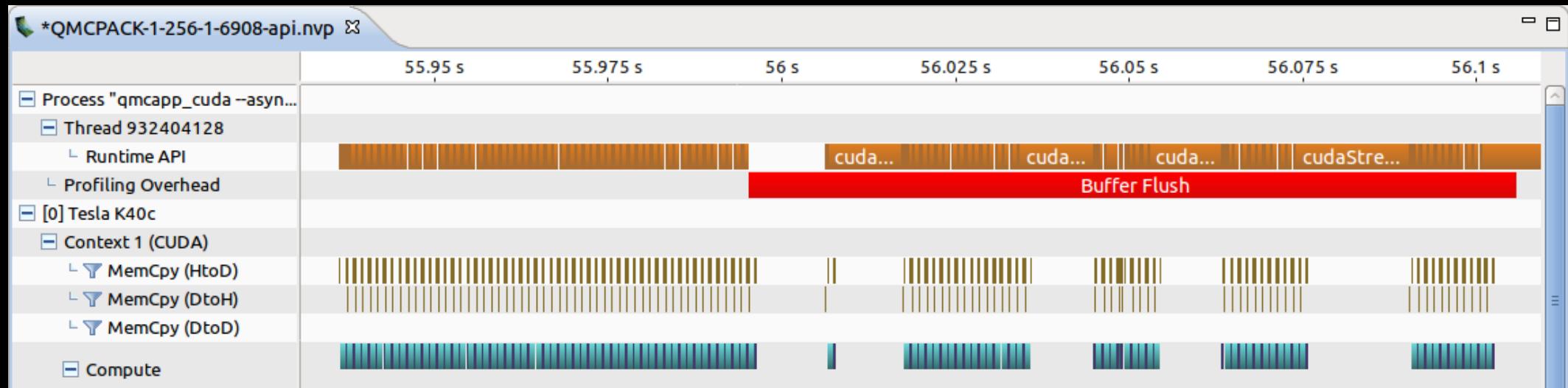
- Solutions

- Make longer running kernels
    - Fuse nearby kernels together
    - Batch work within a single kernel
    - Solve larger problems

## PROBLEM 6: EXCESSIVE SYNCHRONIZATION

- Symptoms
  - Host does not get ahead
  - Large gaps of idle time in timeline
  - Host shows synchronization API calls
- Solutions
  - Use events to limit the amount of synchronization
  - Use `cudaStreamWaitEvent` to prevent host synchronization
  - Use `cudaEventSynchronize`

# PROBLEM 7: PROFILER OVERHEAD

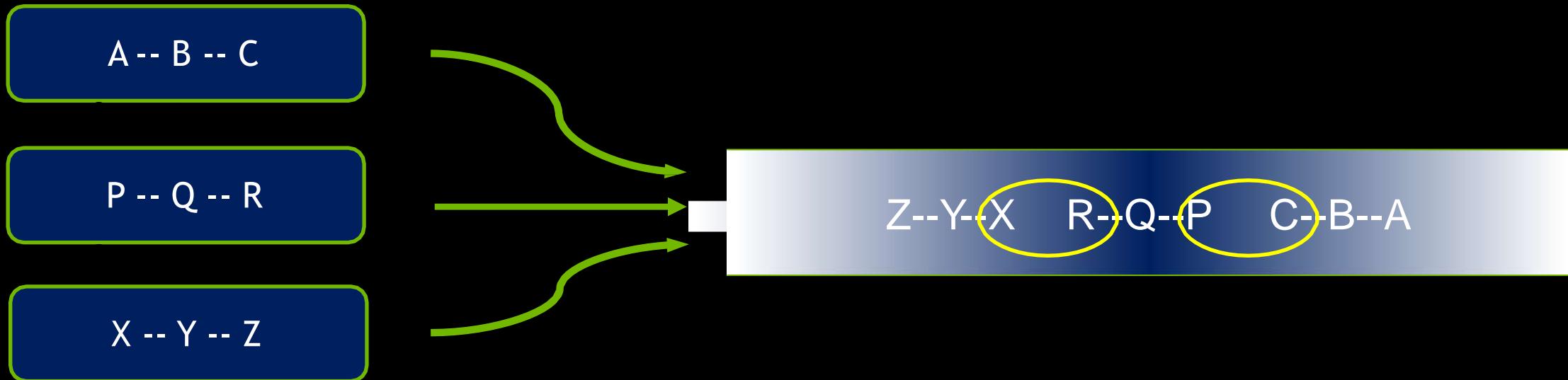


- Symptoms: Large gaps in timeline, Timeline shows profiler overhead
- Real code likely does not have the same problem
- Solution: Avoid `cudaDeviceSynchronize()` & `cudaStreamSynchronize()`

```
cudaEventRecord(event, stream);
```

```
cudaEventSynchronize(event);
```

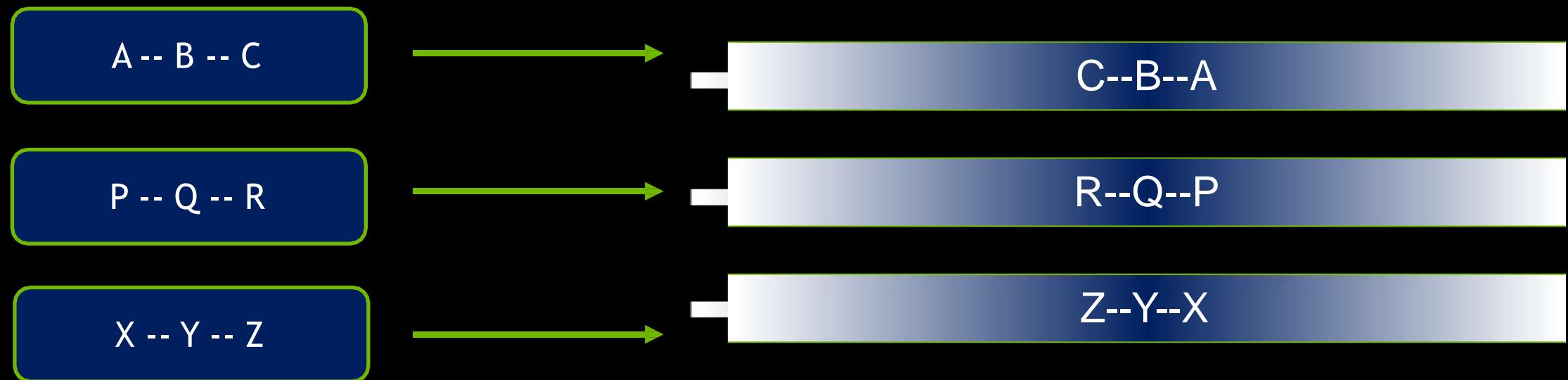
# FERMI CONCURRENCY



Fermi allows 16-way concurrency

- But CUDA streams multiplex into a single queue
- Issue order matters for concurrency
- For more info see the streams webinar
  - <https://developer.nvidia.com/gpu-computing-webinars>

# KEPLER IMPROVED CONCURRENCY



**Kepler allows 32-way concurrency**

- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies

# REVIEW

- Common Streaming Problems
  - 1. Using the default stream
  - 2. Memory transfer issues
  - 3. Implicit synchronization
  - 4. Limited by host throughput
  - 5. Limited by launch overhead
  - 6. Excessive synchronization
  - 7. Profiler overhead
  - 8. False serialization on Fermi

# ADVANCED STREAMING TOPICS

# STREAM CALLBACKS

- Cuda 5.0 now allows you to add stream callbacks (K20 or newer)
  - Useful for launching work on the host when something has completed

```
void CUDART_CB MyCallback(void *data){  
    ...  
}  
...  
MyKernel<<<100, 512, 0, stream>>>();  
cudaStreamAddCallback(stream, MyCallback, (void*)i, 0);
```

- Callbacks are processed by a driver thread
  - The same thread processes all callbacks
  - You can use this thread to signal other threads

# PRIORITY STREAMS

- You can give streams priority
  - High priority streams will preempt lower priority streams.
    - Currently executing blocks will complete but new blocks will only be scheduled after higher priority work has been scheduled.
- Query available priorities:
  - `cudaDeviceGetStreamPriorityRange(&low, &high)`
  - Kepler: low: -1, high: 0
  - Lower number is higher priority
- Create using special API:
  - `cudaStreamCreateWithPriority(&stream, flags, priority)`
- Cuda 5.5+

# REVIEW

- Enabling concurrency is vital to achieving peak performance
- Use MPS+MPI to get concurrency automatically
- Or use streams to add concurrency
  - Watch out for common mistakes
    - Using stream 0
    - Synchronous memory copies
    - Not using pinned memory
    - Overuse of synchronization primitives

## samples \ data-init-in-streams

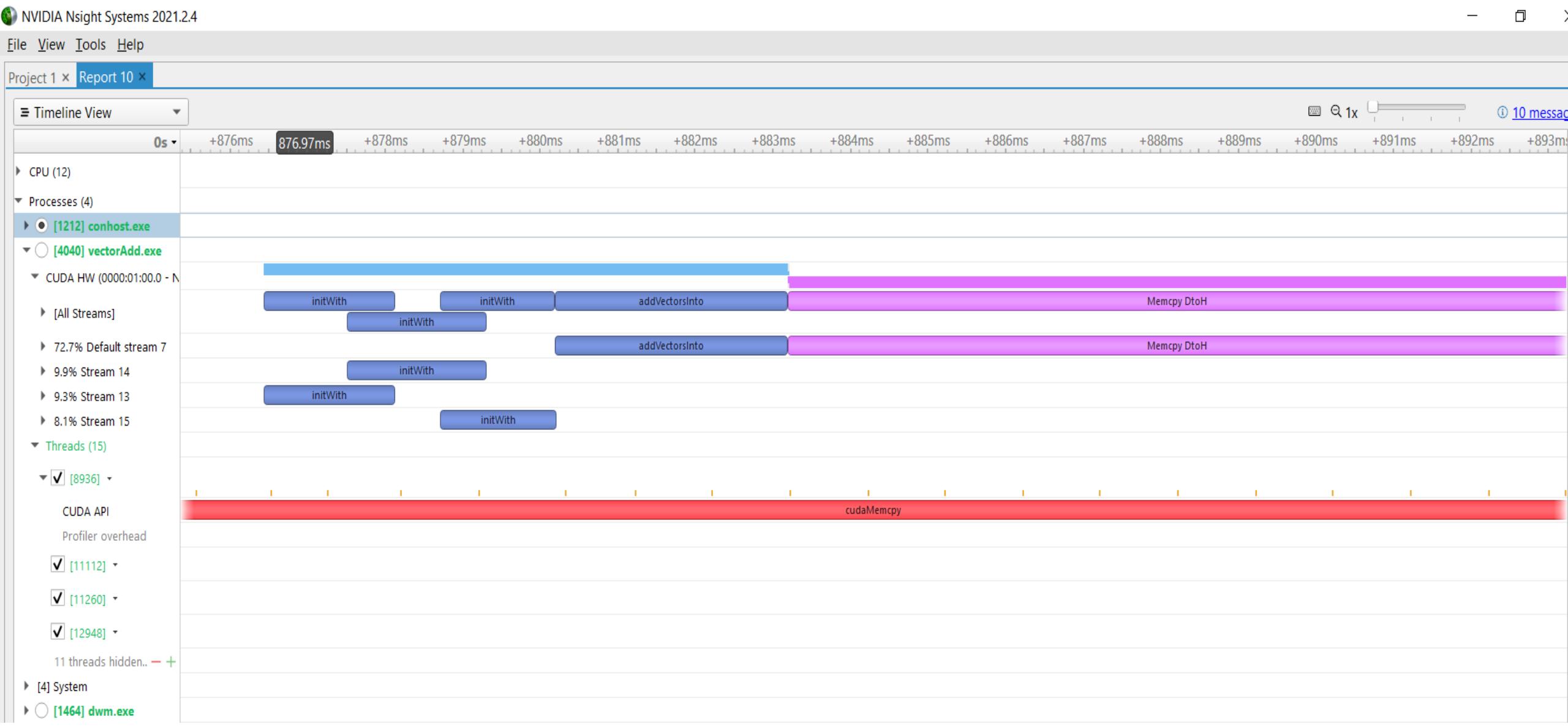
```
/*
 * Create 3 streams to run initialize the 3 data vectors in parallel.
 */

cudaStream_t stream1, stream2, stream3;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaStreamCreate(&stream3);

/*
 * Give each `initWith` launch its own non-standard stream.
 */

initWith<<<numberOfBlocks, threadsPerBlock, 0, stream1>>>(3, a, N);
initWith<<<numberOfBlocks, threadsPerBlock, 0, stream2>>>(4, b, N);
initWith<<<numberOfBlocks, threadsPerBlock, 0, stream3>>>(0, c, N);
```

# samples \ data-init-in-streams



## samples \ streams-overlapped-transfer

```
/*
 * Synchronize - we are using different non-standard streams for initialisation
 * and calcs, so we need to explicitly synchronize to ensure all data has been
 * initialised before starting the calculations
 */
asyncErrInit = cudaDeviceSynchronize();
if(asyncErrInit != cudaSuccess) printf("Error init: %s\n", cudaGetErrorString(asyncErrInit));

for (int i = 0; i < 4; ++i)
{
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    addVectorsInto<<<numberOfBlocks/4, threadsPerBlock, 0, stream>>>(&c[i*N/4], &a[i*N/4], &b[i*N/4], N/4);
    cudaMemcpyAsync(&h_c[i*N/4], &c[i*N/4], size/4, cudaMemcpyDeviceToHost, stream);
    cudaStreamDestroy(stream);
}
```

# samples \ streams-overlapped-transfer

NVIDIA Nsight Systems 2021.2.4

File View Tools Help

Project 1 x Report 8 x

Timeline View

Q 1x

+836ms +838ms +840ms +842ms +844ms +846ms +848ms +850ms +852ms +854ms

▶ CPU (12)

▼ Processes (4)

▼ [11064] vectorAdd.exe

▼ CUDA HW (0000:01:00.0 - N

▶ [All Streams]

▶ 19.1% Stream 16

▶ 18.6% Stream 19

▶ 18.3% Stream 17

▶ 18.3% Stream 18

▶ 9.4% Stream 14

▶ 8.5% Stream 13

▶ 7.9% Stream 15

initWith

initWith

addVecto...

Memcpy DtoH

Memcpy DtoH

Memcpy DtoH

Memcpy DtoH

initWith

addVecto...

Memcpy DtoH

addVecto...

addVecto...

Memcpy DtoH

addVecto...

Memcpy DtoH

addVecto...

Memcpy DtoH

initWith

initWith

initWith

## samples\streams-cipher (single stream)

```
int main (int argc, char * argv[]) {
    const char * encrypted_file = "/dli/task/encrypted";

    Timer timer;

    const uint64_t num_entries = 1UL << 26;
    const uint64_t num_iters = 1UL << 10;
    const bool openmp = true;

    uint64_t * data_cpu, * data_gpu;
    cudaMallocHost(&data_cpu, sizeof(uint64_t)*num_entries);
    cudaMalloc    (&data_gpu, sizeof(uint64_t)*num_entries);

    read_encrypted_from_file(encrypted_file, data_cpu, sizeof(uint64_t)*num_entries);

    cudaMemcpy(data_gpu, data_cpu, sizeof(uint64_t)*num_entries, cudaMemcpyHostToDevice);

    // Create non-default stream.
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    // Launch kernel in non-default stream.
    decrypt_gpu<<<80*32, 64, 0, stream>>>(data_gpu, num_entries, num_iters);

    // Destroy non-default stream.
    cudaStreamDestroy(stream);

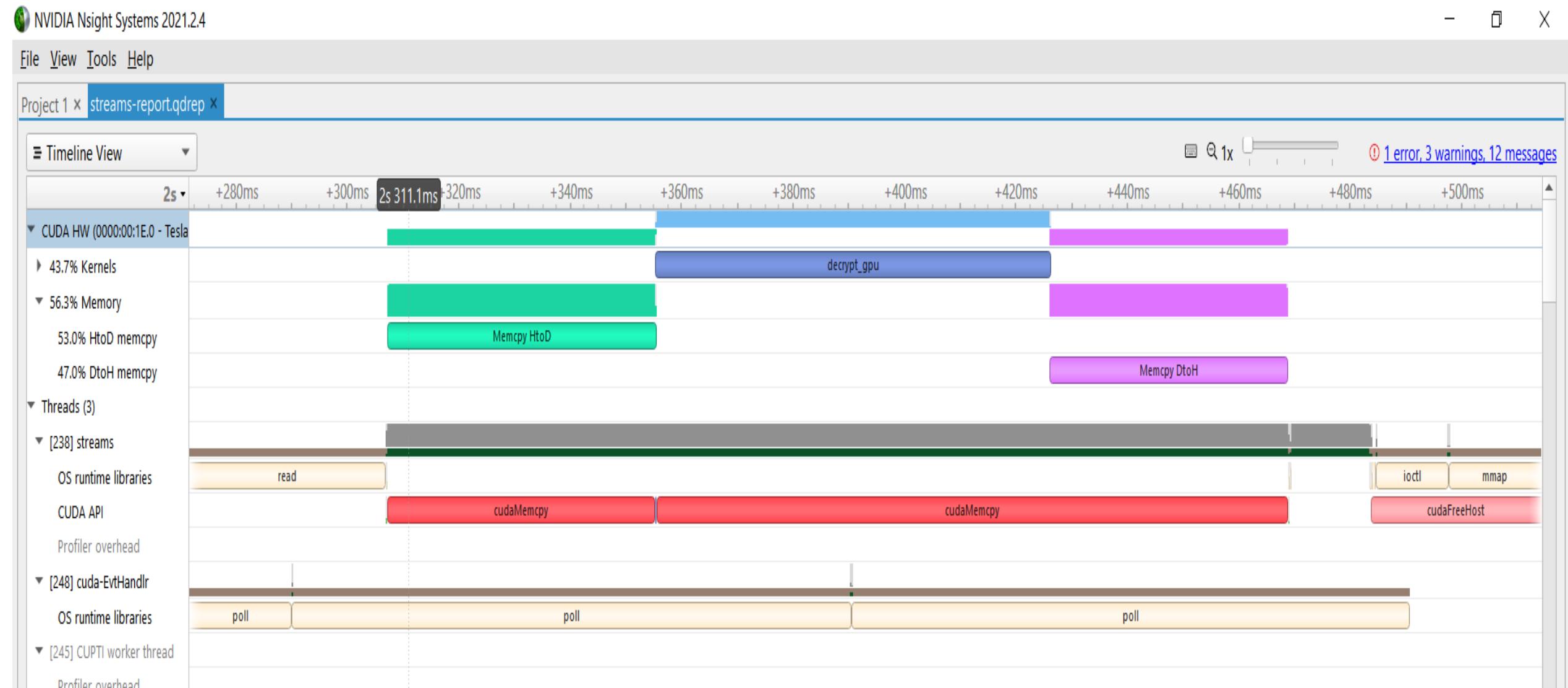
    cudaMemcpy(data_cpu, data_gpu, sizeof(uint64_t)*num_entries, cudaMemcpyDeviceToHost);

    const bool success = check_result_cpu(data_cpu, num_entries, openmp);

    cudaFreeHost(data_cpu);
    cudaFree    (data_gpu);
}
```

samples\13-streams\_cipher\baseline\_solution.cu

# samples\streams-cipher (single stream)



samples\13-streams\_cipher\baseline\_solution-report.qdrep

## samples\streams-cipher (concurrent streams)

```
int main (int argc, char * argv[]) {  
  
    const char * encrypted_file = "/dli/task/encrypted";  
    const uint64_t num_entries = 1UL << 26;  
    const uint64_t num_iters = 1UL << 10;  
    const bool openmp = true;  
  
    // Define the number of streams.  
    const uint64_t num_streams = 32;  
  
    // Use round-up division to calculate chunk size.  
    const uint64_t chunk_size = sdiv(num_entries, num_streams);  
  
    uint64_t * data_cpu, * data_gpu;  
    cudaMallocHost(&data_cpu, sizeof(uint64_t)*num_entries);  
    cudaMalloc(&data_gpu, sizeof(uint64_t)*num_entries);  
    check_last_error();  
  
    read_encrypted_from_file(encrypted_file, data_cpu, sizeof(uint64_t)*num_entries);  
  
    // Create array for storing streams.  
    cudaStream_t streams[num_streams];  
  
    // Create number of streams and store in array.  
    for (uint64_t stream = 0; stream < num_streams; stream++)  
        cudaStreamCreate(&streams[stream]);
```

## samples\streams-cipher (concurrent streams)

```
// For each stream...
for (uint64_t stream = 0; stream < num_streams; stream++) {

    // ...calculate index into global data (`lower`) and size of data for it to process (`width`).
    const uint64_t lower = chunk_size*stream;
    const uint64_t upper = min(lower+chunk_size, num_entries);
    const uint64_t width = upper-lower;

    // ...copy stream's chunk to device.
    cudaMemcpyAsync(data_gpu+lower, data_cpu+lower,
                   sizeof(uint64_t)*width, cudaMemcpyHostToDevice,
                   streams[stream]);

    // ...compute stream's chunk.
    decrypt_gpu<<<80*32, 64, 0, streams[stream]>>>
        (data_gpu+lower, width, num_iters);

    // ...copy stream's chunk to host.
    cudaMemcpyAsync(data_cpu+lower, data_gpu+lower,
                   sizeof(uint64_t)*width, cudaMemcpyDeviceToHost,
                   streams[stream]);
}

for (uint64_t stream = 0; stream < num_streams; stream++)
    // Synchronize streams before checking results on host.
    cudaStreamSynchronize(streams[stream]);

const bool success = check_result_cpu(data_cpu, num_entries, openmp);
```

# samples\streams-cipher (concurrent streams)

NVIDIA Nsight Systems 2021.2.4

File View Tools Help

Project 1 x streams-solution-report.qdrep x

Timeline View ▾

Q 1x

0s 0.1s 0.2s 0.3s 0.4s 0.5s 0.6s 0.7s 0.8s 0.9s 1s

▶ CUDA HW (0000:00:1E.0 - Tesla)

▼ Threads (3)

▼ [335] streams\_solutio

OS runtime libraries

CUDA API

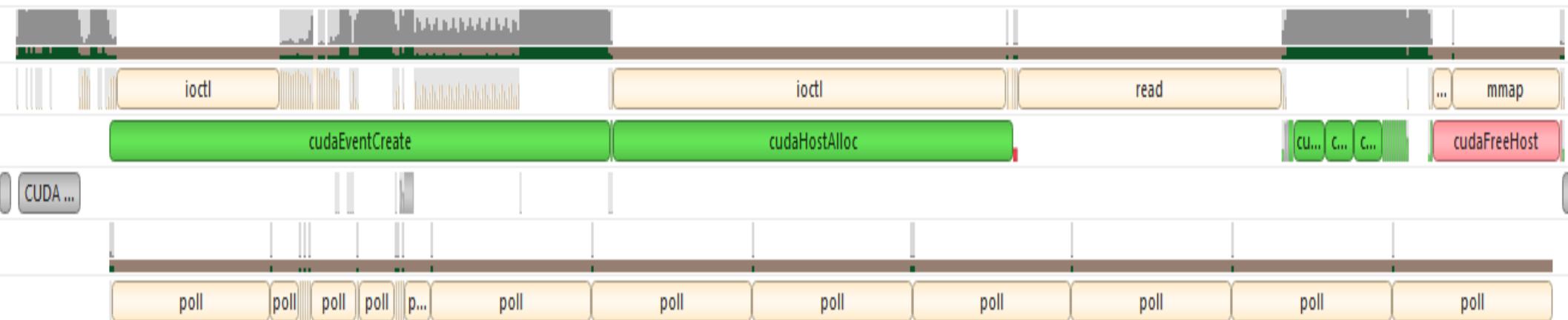
Profiler overhead

▼ [345] cuda-EvtHandlr

OS runtime libraries

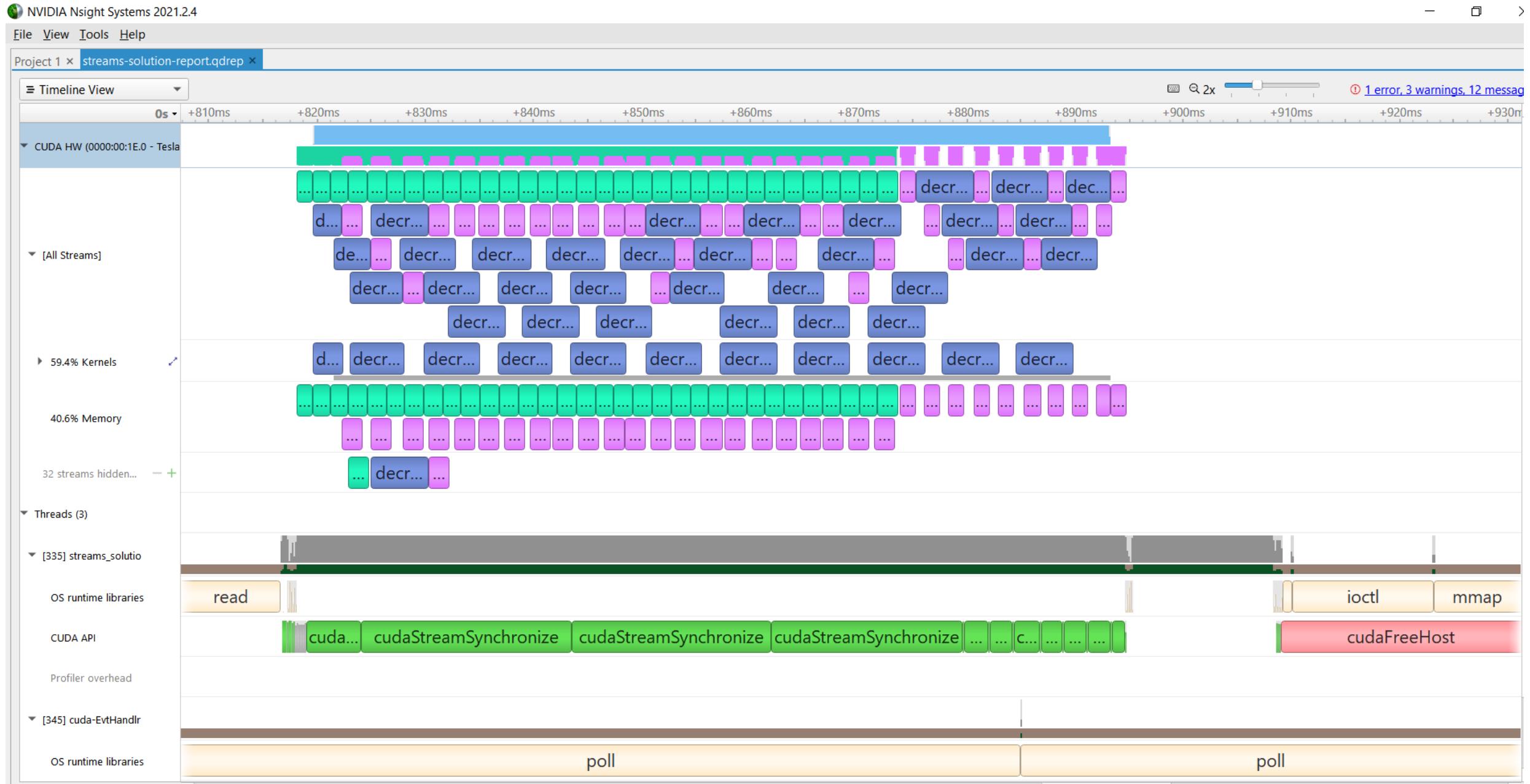
▼ [342] CUPTI worker thread

Profiler overhead



docs\nvidia-courseware\streams\streams\_cipher\streams-solution-report.qdrep

# samples\streams-cipher (concurrent streams) – zoomed timeline



# UPDATE: PINNED (NON-PAGEABLE) MEMORY

- ▶ Pinned memory enables:
  - ▶ faster Host<->Device copies memcopies asynchronous with CPU memcopies
  - ▶ asynchronous with GPU

## Usage

- ▶ `cudaHostAlloc` / `cudaFreeHost`
- ▶
  - ▶ instead of `malloc` / `free` or `new` / `delete`
- ▶ `cudaHostRegister` / `cudaHostUnregister`
  - ▶
    - ▶ pin regular memory (e.g. allocated with `malloc`) after allocation

## Implication:

pinned memory is essentially removed from host virtual (pageable) memory

# UPDATE: CUDALAUNCHHOSTFUNC()

- ▶ Allows definition of a host-code function that will be issued into a CUDA stream
- ▶ Follows stream semantics: function will not be called until stream execution reaches that point
- ▶ Uses a thread spawned by the GPU driver to perform the work
- ▶ Has limitations: do not use any CUDA runtime API calls (or kernel launches) in the function
- ▶ Useful for deferring CPU work until GPU results are ready
- ▶ `cudaLaunchHostFunc()` replaces legacy `cudaStreamAddCallback()`

# UPDATE: COPY-COMPUTE OVERLAP WITH MANAGED MEMORY

In particular, with demand-paging

- ▶ Follow same pattern, except use `cudaMemPrefetchAsync()` instead of `cudaMemcpyAsync()`
- ▶ Stream semantics will guarantee that any needed migrations are performed in proper order
- ▶ However, `cudaMemPrefetchAsync()` has more work to do than `cudaMemcpyAsync()` (updating of page tables in CPU and GPU)
- ▶ This means the call can take substantially more time to return than an “ordinary” async call - can introduce unexpected gaps in timeline
- ▶ Behavior varies for “busy” streams vs. idle streams. Counterintuitively, “busy” streams may result in better throughput
- ▶ Read about it:
  - ▶ <https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/>

# CUDAEVENT

- ▶ cudaEvent is an entity that can be placed as a “marker” in a stream
- ▶ A cudaEvent is said to be “recorded” when it is issued
- ▶ A cudaEvent is said to be “completed” when stream execution reaches the point where it was recorded
- ▶ Most common use: timing

```
cudaEvent_t start, stop;          // cudaEvent has its own type
cudaEventCreate(&start);          // cudaEvent must be created
cudaEventCreate(&stop);           // before use
cudaEventRecord(start);          // “recorded” (issued) into default stream
Kernel<<<b, t>>>(...);        // could be any set of CUDA device activity
cudaEventRecord(stop);
cudaEventSynchronize(stop);       // wait for stream execution to reach “stop”
cudaEventElapsedTime(&float_var, start, stop); // measure Kernel duration
```

- ▶ Also useful for arranging complex concurrency scenarios
- ▶ Event-based timing may give unexpected results for host activity or complex concurrency scenarios

# Multi-GPU

# MULTI-GPU - DEVICE MANAGEMENT

- ▶ Not a replacement for OpenMP, MPI, etc.
- ▶ Application can query and select GPUs

```
cudaGetDeviceCount (int *count)

cudaSetDevice (int device)

cudaGetDevice (int *device)

cudaGetDeviceProperties (cudaDeviceProp *prop, int device)
```

- ▶ Multiple host threads can share a device
- ▶ A single host thread can manage multiple devices

```
cudaSetDevice (i) to select current device

cudaMemcpyPeerAsync (...) for peer-to-peer copies
```

# MULTI-GPU - STREAMS

- ▶ Streams (and cudaEvent) have implicit/automatic *device association*
- ▶ Each device also has its own unique default stream
- ▶ Kernel launches will fail if issued into a stream not associated with current device
- ▶ `cudaStreamWaitEvent()` can synchronize streams belonging to separate devices, `cudaEventQuery()` can test if an event is “complete”
- ▶ Simple device concurrency:

```
cudaSetDevice(0);
cudaStreamCreate(&stream0);          //associated with device 0
cudaSetDevice(1);
cudaStreamCreate(&stream1);          //associated with device 1
Kernel<<<b, t, 0, stream1>>>(...); // these kernels have the possibility
cudaSetDevice(0);
Kernel<<<b, t, 0, stream0>>>(...); // to execute concurrently
```

# MULTI-GPU - DEVICE-TO-DEVICE DATA COPYING

- ▶ If system topology supports it, data can be copied directly from one device to another over a fabric (PCIE, or NVLink)
- ▶ Device must first be explicitly placed into a peer relationship (“clique”)
- ▶ Must enable “peering” for both directions of transfer (if needed)
- ▶ Thereafter, memory copies between those two devices will not “stage” through a system memory buffer (GPUDirect P2P transfer)

```
cudaSetDevice(0);
cudaDeviceCanAccessPeer(&canPeer, 0, 1); // test for 0, 1 peerable
,
cudaDeviceEnablePeerAccess(1,           // device 0 sees device 1 as a
0);                                "peer"
cudaSetDevice(1);
cudaDeviceEnablePeerAccess(dst_ptr, 0, // device 1 sees device 0 as a dev 0 copy
0); // dev 0 is no longer a peer of dev 1
cudaDeviceDisablePeerAccess(0);
```

- ▶ Limit to the number of peers in your “clique”

# OTHER CONCURRENCY SCENARIOS

- ▶ Host/Device execution concurrency:

```
Kernel<<<b, t>>>(...); // this kernel execution can overlap  
                           with  
cpuFunction(...);        // this host code
```

- ▶ Concurrent kernels:

```
Kernel<<<b, t, 0, streamA>>>(...); // these kernels have the possibility  
Kernel<<<b, t, 0, streamB>>>(...); // to execute concurrently
```

- ▶ In practice, concurrent kernel execution on the same device is hard to witness
- ▶ Requires kernels with relatively low resource utilization and relatively long execution time
- ▶ There are hardware limits to the number of concurrent kernels per device
- ▶ Less efficient than saturating the device with a single kernel

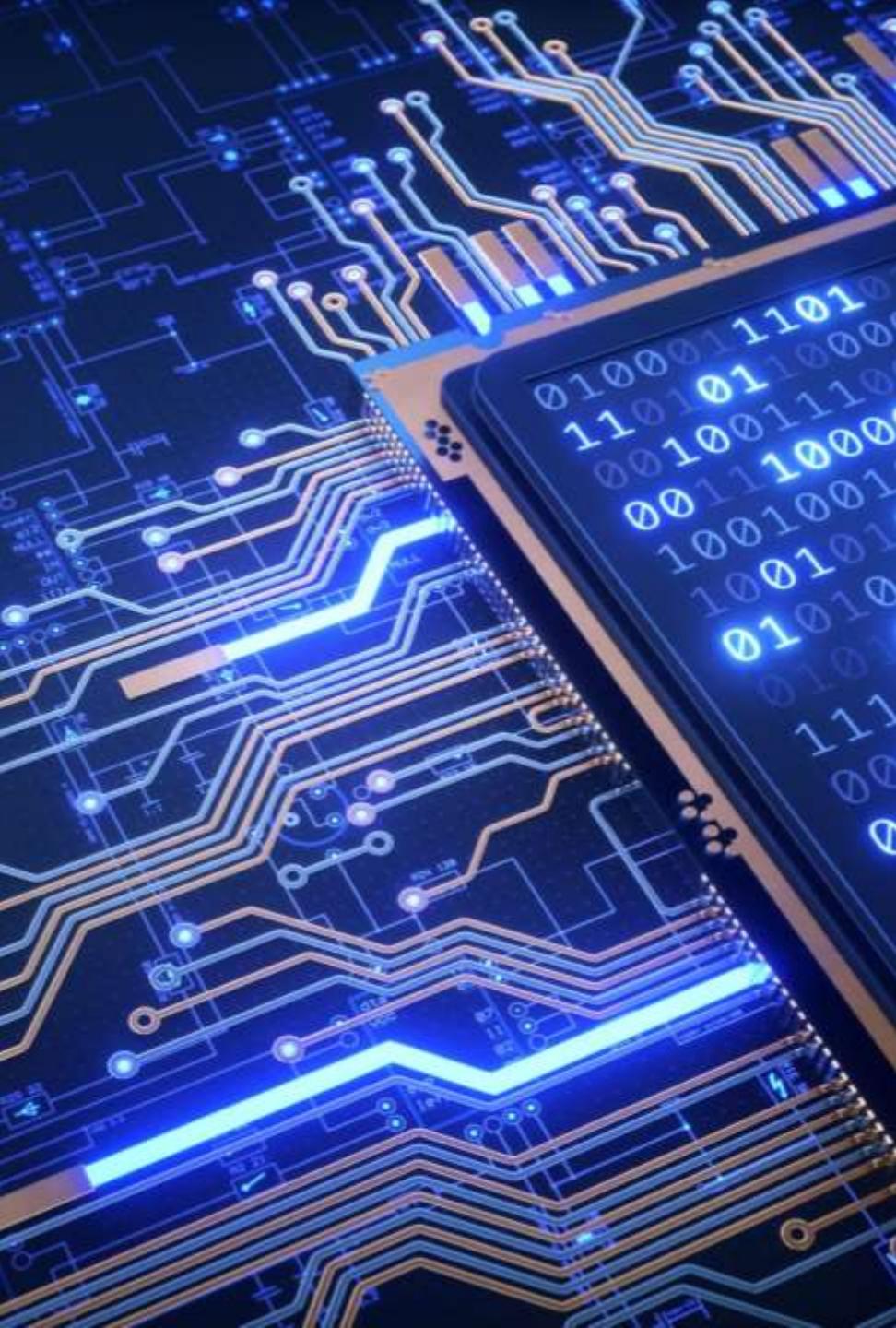
# STREAM PRIORITY

- ▶ CUDA streams allow an optional definition of a *priority*
- ▶ This affects execution of concurrent kernels (only).
- ▶ The GPU block scheduler will attempt to schedule blocks from high priority (stream) kernels before blocks from low priority (stream) kernels
- ▶ Current implementation only has 2 priorities
- ▶ Current implementation does not cause preemption of blocks

```
// get the range of stream priorities for this device
int priority_high, priority_low;
cudaDeviceGetStreamPriorityRange(&priority_low, &priority_high);
// create streams with highest and lowest available priorities
cudaStream_t st_high, st_low;
cudaStreamCreateWithPriority(&st_high, cudaStreamNonBlocking, priority_high);
cudaStreamCreateWithPriority(&st_low, cudaStreamNonBlocking, priority_low);
```

# CUDA GRAPHS (OVERVIEW)

- ▶ New feature in CUDA 10
- ▶ Allows for the definition of a sequence of stream(s) work (kernels, memory copy operations, callbacks, host functions, graphs)
- ▶ Each work item is a *node* in the graph
- ▶ Allows for the definition of *dependencies* (e.g. these 3 nodes must finish before this one can begin)
- ▶ Dependencies are effectively graph edges
- ▶ Once defined, a graph may be executed by launching it into a stream
- ▶ Once defined, a graph may be re-used
- ▶ Has both a manual definition method and a “capture” method



# DAY 4

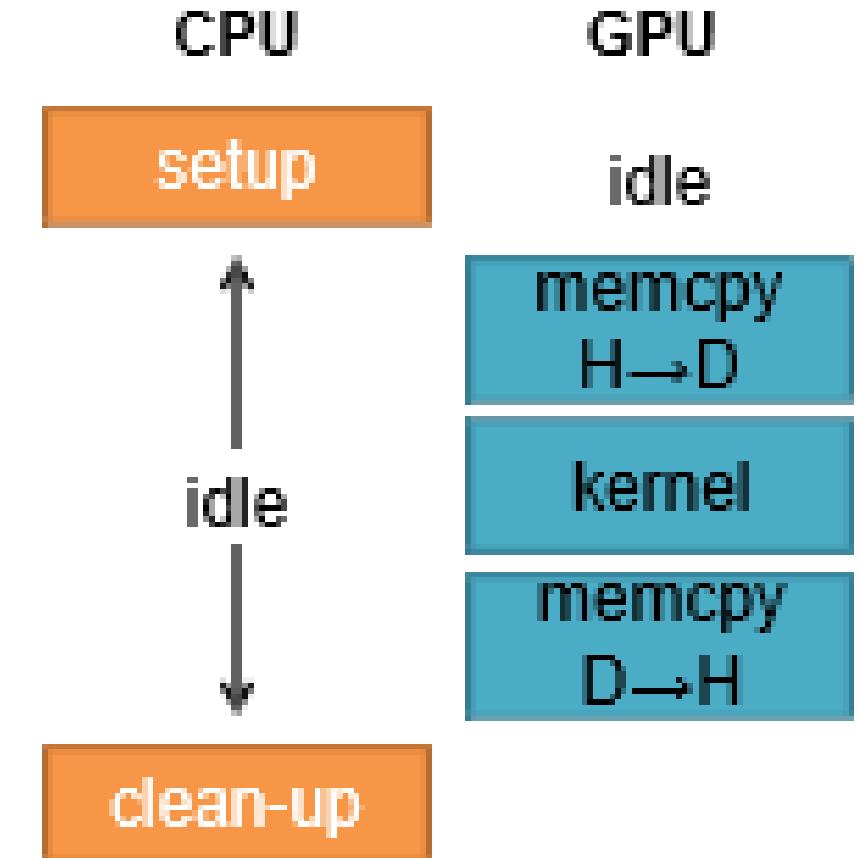
- Dynamic Parallelism
  - Quicksort
  - Mandelbrot
- Parallel Reduction
- Cooperative Thread Groups
- Image Processing
  - Textures
  - Sobel Filter
  - Box Filter
  - Histogram
  - Transpose

Get the updated training content from:  
<https://github.com/kbvis3d/toshiba-cuda-2021>

# Streams and Dynamic Parallelism

# Synchronicity in CUDA

- All CUDA calls are either synchronous or asynchronous relative to the host
  - Synchronous – enqueue work and wait for completion
  - Asynchronous – enqueue work and return immediately
- Use asynchronous operations to introduce another level of parallelism
  - Better utilization of the system
  - Use all the available resources on your system concurrently
    - CPU cores
    - GPU(s)
- What is the CPU doing during GPU kernels?
- Waiting for the GPU, typically...
- Use it for an independent task in parallel!
- Eg. Re-ordering data, writing to a file



# Concurrent CPU-GPU Computation

- Kernel invocations are asynchronous
- Need to synchronize CPU and GPU
  - Explicit:  
  cudaDeviceSynchronize()
  - Implicit: cudaMemcpy()
- Order matters: launch kernels then call CPU functions

```
void foo()
{
    cpuCode1();

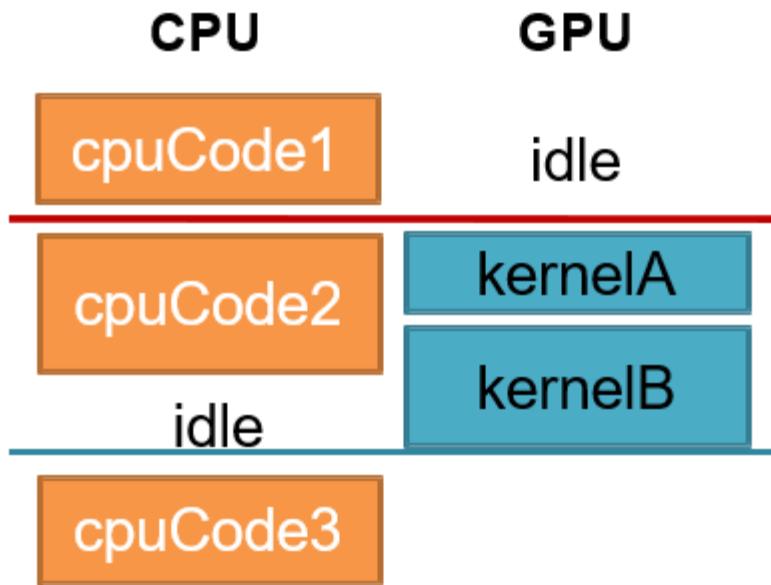
    KernelA<<<grid,block>>>(...);
    KernelB<<<grid,block>>>(...);

    cpuCode2();

    cudaDeviceSynchronize();

    // or cudaMemcpy for
    // implicit synchronization
    // cudaMemcpy(...,
    // cudaMemcpyDeviceToHost)
}
```

# Task Timeline



```
void foo()
{
    cpuCode1();

    KernelA<<<grid,block>>>(...);
    KernelB<<<grid,block>>>(...);

    cpuCode2();

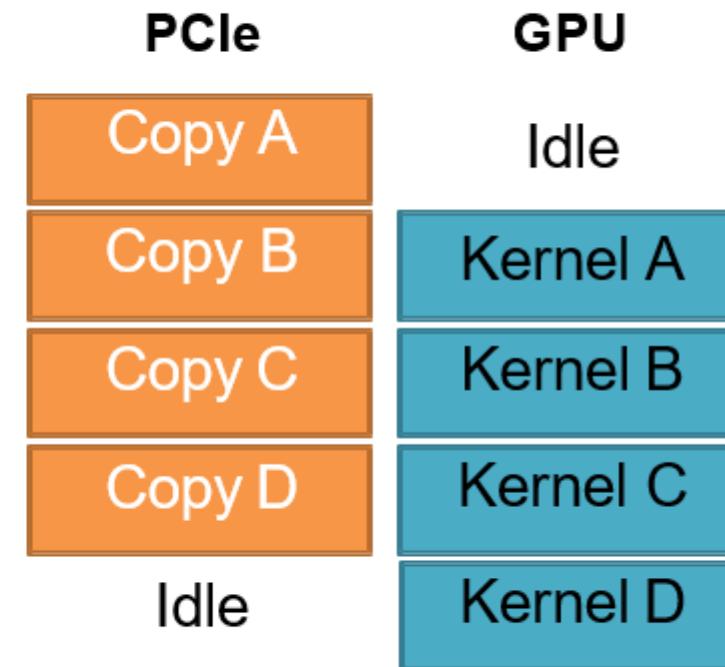
    // Explicit synchronization
    cudaDeviceSynchronize();

    // Or implicit synchronization
    // cudaMemcpy(...)

    cpuCode3();
}
```

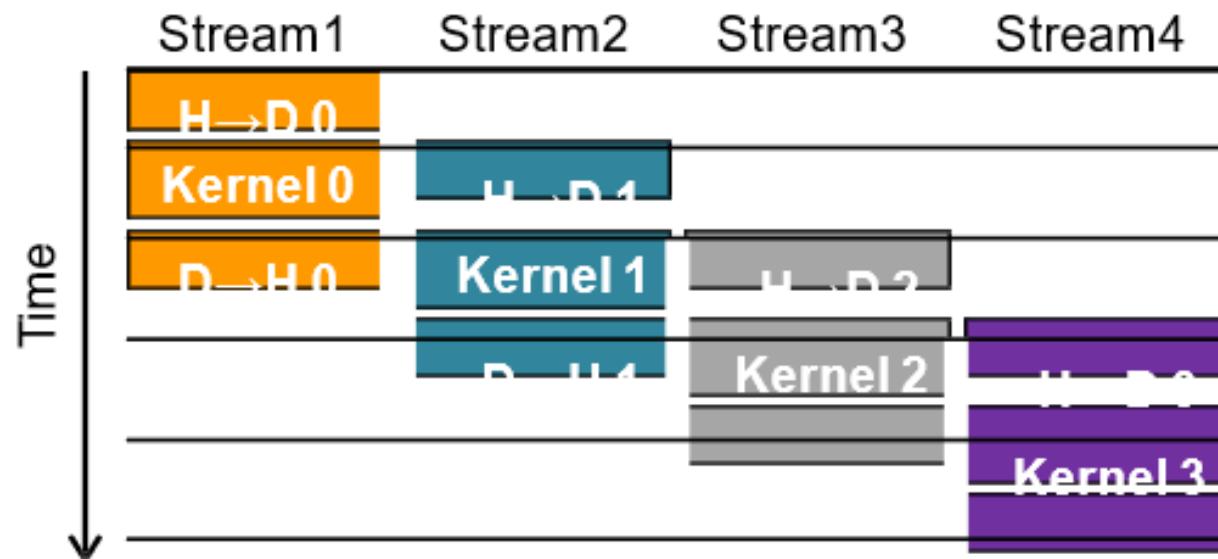
# Concurrent Transfers

- Copy data over bus while kernels are executing on GPUs
- Check **asyncEngineCount** property in **cudaGetDeviceProperties()**
  - 0: No overlap is supported
  - 1: One data transfer can overlap with kernel execution
  - 2: Bidirectional data transfers can overlap with kernel execution
    - These devices have 2 DMA Engines!



# Asynchronous Communications

- Implemented via asynchronous variations of `cudaMemcpy*`()
- Stream = sequence (queue) of operations that execute *in order* on GPU
  - No ordering constraints on operations between different streams
- Operations can execute asynchronously if they:
  - 1) Are from different streams
  - 2) Use different hardware resources
  - 3) The necessary hardware resources are available



# Page-Locked Host Memory (1)

- Page-locked (“pinned”) host memory required for asynchronous transfers
- `cudaMemcpyAsync()` does not require page-locked host memory
  - HOWEVER then transfers are synchronous
- `cudaMallocHost()` or `cudaHostAlloc()` and `cudaFreeHost()`
  - Allocate/free page-locked memory
- `cudaHostRegister()/cudaHostUnregister()`
  - Make existing memory page-locked

# Page-Locked Host Memory (2)

- Page-locked memory is typically faster with regular `cudaMemcpy()` too!
  - 6.2 vs 10.9 GB/s on K80, PCIe x16 Gen3 (Device to Host)
  - 3.0 vs 13.1 GB/s on P100, PCIe x16 Gen 3 (Device to Host)
- Use with caution!
  - Allocating too much page-locked memory can reduce system performance
  - Calls to `cudaMallocHost()` will fail long before allocations by `malloc()`

# Syntax – Asynchronous Communications

- Asynchronous Data Transfers
  - `cudaMemcpyAsync(void* dst, void* src, size_t nbytes,  
enum cudaMemcpyKind direction,  
cudaStream_t stream = 0)`
- Handles created and destroyed using API
  - `cudaError_t cudaStreamCreate(cudaStream_t* stream)`
  - `cudaError_t cudaStreamDestroy(cudaStream_t stream)`
- Launch kernels to different streams using the 4<sup>th</sup> argument in the chevron syntax <<<>>>
  - `myKernel<<<gSize, bSize, smBytes, stream>>>(...)`

# Default Stream

- Unless otherwise specified all operations are enqueued in the default stream or “Stream 0”
- Default stream has special synchronization properties
  - Synchronous with all streams
    - Operations in default stream cannot overlap with operations from any other stream
- Exception: Streams with non-blocking flag set
  - `cudaStreamCreateWithFlags(&stream,  
cudaStreamNonBlocking)`
  - Allows operations in current stream to run concurrently with default stream
  - Use to get concurrency with operations out of your control (eg. libraries)

# Syntax Example

```
void foo(...)  
{  
    cudaStream_t stream1, stream2;  
  
    cudaStreamCreate(&stream1);  
    cudaStreamCreate(&stream2);  
  
    cudaMemcpyAsync(d_bar, h_bar, sizeof(bar),  
                   cudaMemcpyHostToDevice, stream1);  
  
    kernelA<<<grid, block, 0, stream2>>>(...);  
    kernelB<<<grid, block, 0, stream1>>>(...);  
  
    ...  
  
    cudaStreamDestroy(stream1);  
    cudaStreamDestroy(stream2);  
}
```

These can overlap because they:

- 1) are on different streams
- 2) use different resources

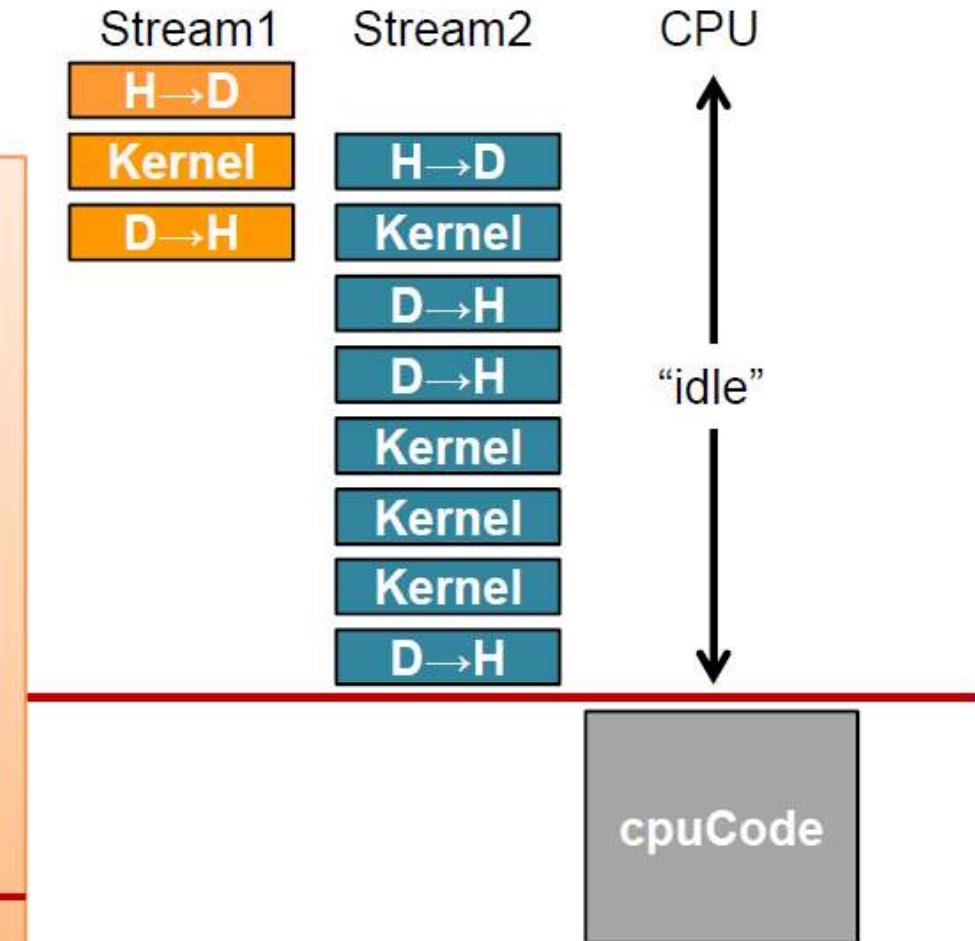
- These are asynchronous with CPU execution
- Synchronization is required!

# Synchronization

## 1. cudaDeviceSynchronize()

CPU ↔ all operations on current device

```
cudaMemcpyAsync(...HostToDevice, stream1);  
  
cudaMemcpyAsync(...HostToDevice, stream2);  
Kernel<<<..., stream1>>>(...);  
  
cudaMemcpyAsync(...DeviceToHost, stream1);  
Kernel<<<..., stream2>>>(...);  
  
cudaMemcpyAsync(...DeviceToHost, stream2);  
cudaMemcpyAsync(...DeviceToHost, stream2);  
Kernel<<<..., stream2>>>(...);  
Kernel<<<..., stream2>>>(...);  
Kernel<<<..., stream2>>>(...);  
cudaMemcpyAsync(...DeviceToHost, stream2);  
  
cudaDeviceSynchronize();  
cpuCode();
```

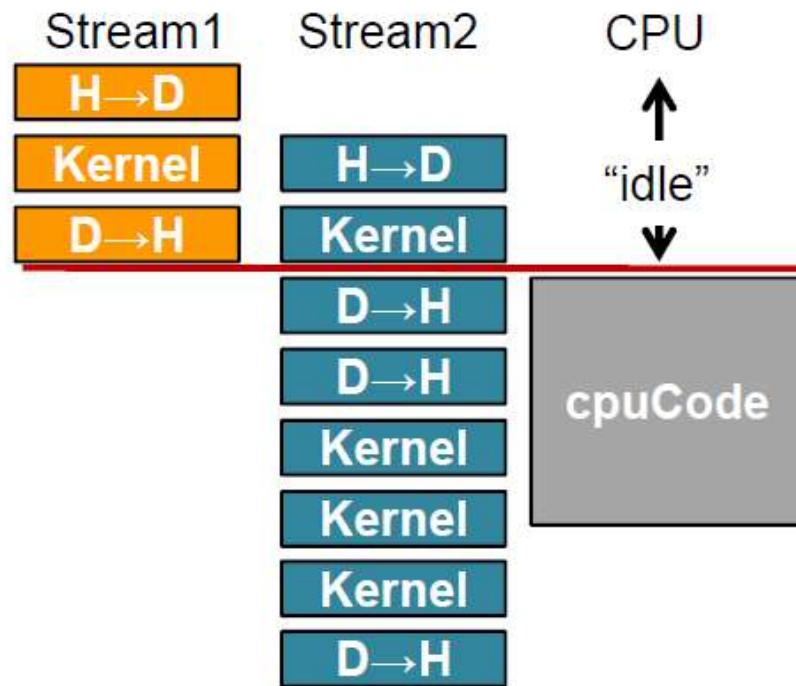


# Synchronization

## 2. cudaStreamSynchronize()

- CPU ↔ all operations in a stream

```
cudaMemcpyAsync(...HostToDevice, stream1);  
  
cudaMemcpyAsync(...HostToDevice, stream2);  
Kernel<<<..., stream1>>>(...);  
  
cudaMemcpyAsync(...DeviceToHost, stream1);  
Kernel<<<..., stream2>>>(...);  
  
cudaMemcpyAsync(...DeviceToHost, stream2);  
cudaMemcpyAsync(...DeviceToHost, stream2);  
Kernel<<<..., stream2>>>(...);  
Kernel<<<..., stream2>>>(...);  
Kernel<<<..., stream2>>>(...);  
cudaMemcpyAsync(...DeviceToHost, stream2);  
  
cudaStreamSynchronize(stream1);  
  
cpuCode();
```



# Events (1)

- Events provide a mechanism to determine when operations in a stream are complete
  - ‘Dummy’ operations that do no work BUT
  - Useful for profiling and synchronization
- Events have a boolean state
  - Occurred (cudaSuccess)
  - Not Occurred
  - Default state = Occurred

# Events (2)

- Create and destroy events
  - `cudaEventCreate(cudaEvent_t* e)`
  - `cudaEventDestroy(cudaEvent_t e)`
- Record an event
  - `cudaEventRecord(cudaEvent_t e, cudaStream_t s = 0)`
    - Sets the event state to not occurred
    - Enqueues the event into a stream
    - Event state is set to occurred when all previous operations in the stream are complete
      - If stream is default stream, all prior operations in ALL other streams must be complete too!

# Event Synchronization

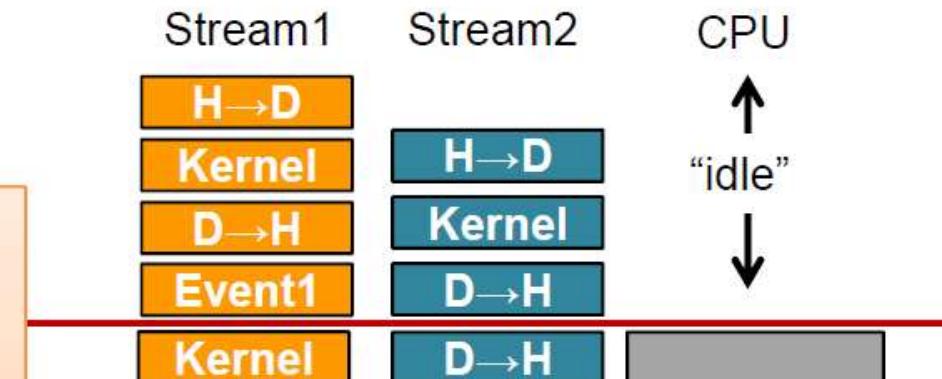
## 3. cudaEventSynchronize()

- CPU ↔ specific point in a stream's queue of operations

```
cudaMemcpyAsync(...HostToDevice, stream1);
Kernel<<<..., stream1>>>(...);
cudaMemcpyAsync(...DeviceToHost, stream1);
cudaEventRecord(event1,stream1);
Kernel<<<..., stream1>>>(...);

cudaMemcpyAsync(...HostToDevice, stream2);
Kernel<<<..., stream2>>>(...);
cudaMemcpyAsync(...DeviceToHost, stream2);
cudaMemcpyAsync(...DeviceToHost, stream2);
Kernel<<<..., stream2>>>(...);
Kernel<<<..., stream2>>>(...);
Kernel<<<..., stream2>>>(...);
cudaMemcpyAsync(...DeviceToHost, stream2);

cudaEventSynchronize(event1);
cpuCode();
```



cpuCode

# Event Synchronization

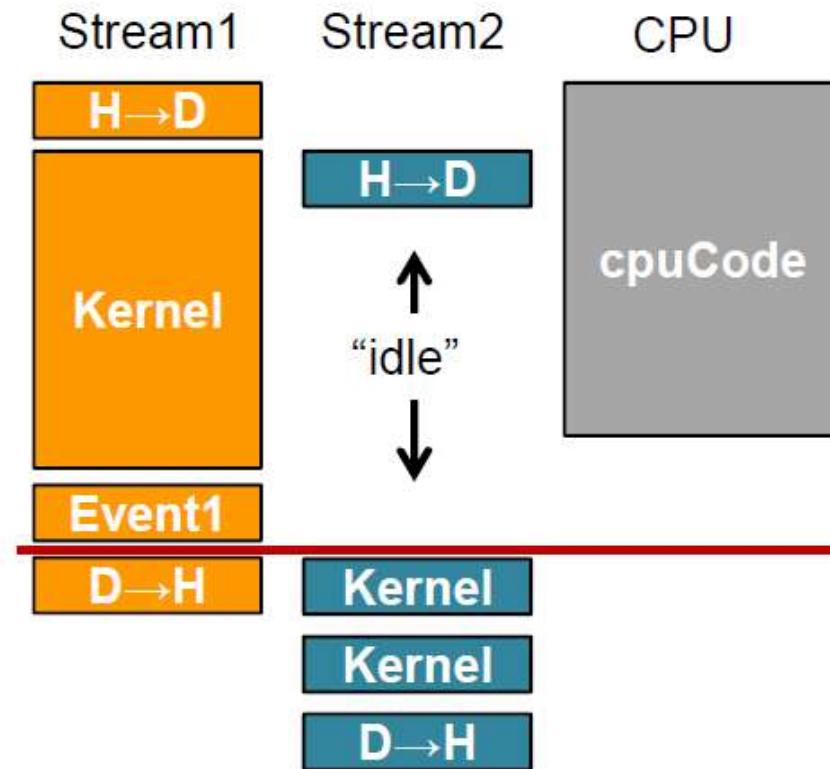
## 4. cudaStreamWaitEvent()

- One stream synchronizes on an event

```
cudaMemcpyAsync(...HostToDevice, stream1);
cudaMemcpyAsync(...HostToDevice, stream2);
Kernel<<<..., stream1>>>(...);
cudaEventRecord(event1, stream1);
cudaMemcpyAsync(...DeviceToHost, stream1);

cudaStreamWaitEvent(stream2, event1, 0);
Kernel<<<..., stream2>>>(...);
Kernel<<<..., stream2>>>(...);
cudaMemcpyAsync(...DeviceToHost, stream2);

cpuCode();
```



# samples / asyncAPI

```
// create cuda event handles
cudaEvent_t start, stop;
checkCudaErrors(cudaEventCreate(&start));
checkCudaErrors(cudaEventCreate(&stop));

StopWatchInterface *timer = NULL;
sdkCreateTimer(&timer);
sdkResetTimer(&timer);

checkCudaErrors(cudaDeviceSynchronize());
float gpu_time = 0.0f;

// asynchronously issue work to the GPU (all to stream 0)
sdkStartTimer(&timer);
cudaEventRecord(start, 0);
cudaMemcpyAsync(d_a, a, nbytes, cudaMemcpyHostToDevice, 0);
increment_kernel<<<blocks, threads, 0, 0>>>(d_a, value);
cudaMemcpyAsync(a, d_a, nbytes, cudaMemcpyDeviceToHost, 0);
cudaEventRecord(stop, 0);
sdkStopTimer(&timer);
```

# samples / asyncAPI

```
    unsigned long int counter = 0;  
#ifdef WAIT_FOR_EVENT  
    // have CPU do some work while waiting for stage 1 to finish  
    while (cudaEventQuery(stop) == cudaErrorNotReady)  
    {  
        counter++;  
    }  
#else  
    cudaEventSynchronize(stop);  
#endif  
  
checkCudaErrors(cudaEventElapsedTime(&gpu_time, start, stop));  
  
// print the cpu and gpu times  
printf("time spent executing by the GPU: %.2f\n", gpu_time);  
printf("time spent by CPU in CUDA calls: %.2f\n", sdkGetTimerValue(&timer));  
printf("CPU executed %lu iterations while waiting for GPU to finish\n", counter);
```

# Concurrent Kernel Execution

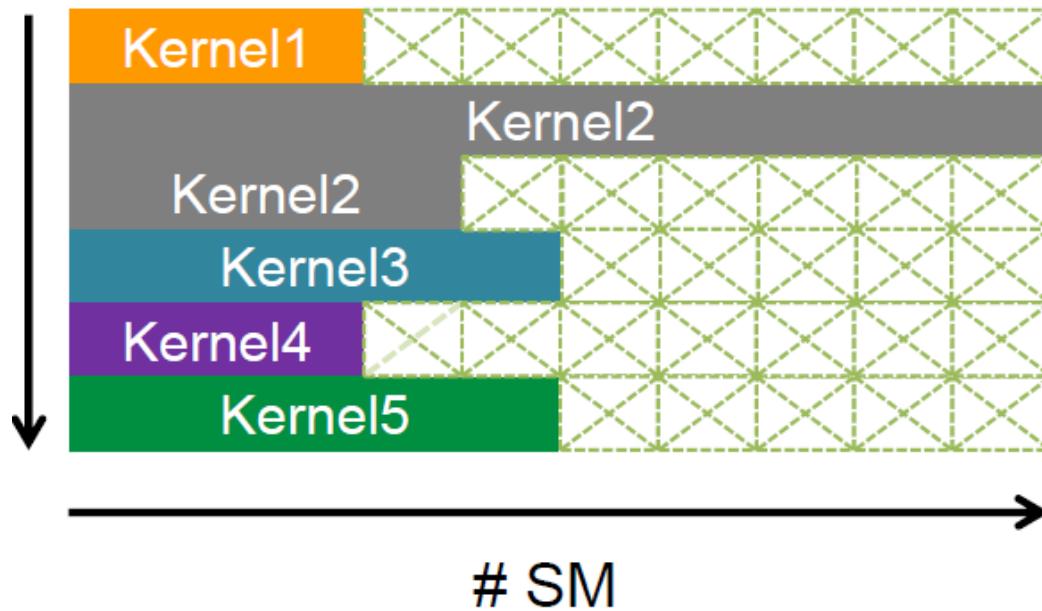
- Most (all?) modern GPUs can execute multiple kernels concurrently
  - Check **concurrentKernels** property in **cudaGetDeviceProperties**
  - Maximum number of concurrent kernels

CC3.0	CC3.2	CC3.5-5.2	CC5.3	CC6.0	CC6.1	CC6.2	CC7.0
16	4	32	16	128	32	16	128

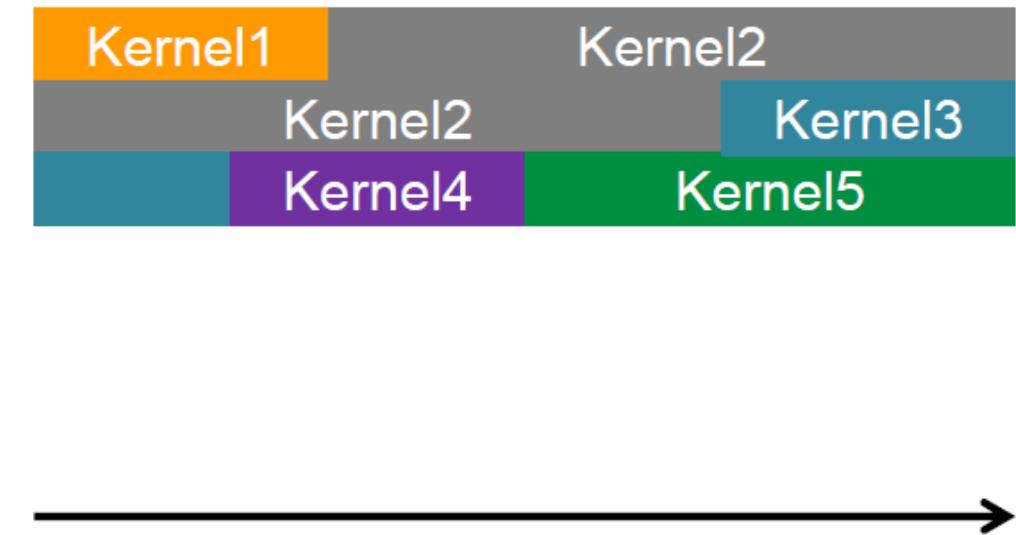
- Kernels must be from the same process
- Kernels must be from different, **non-default**, streams

# Concurrent Kernel Execution

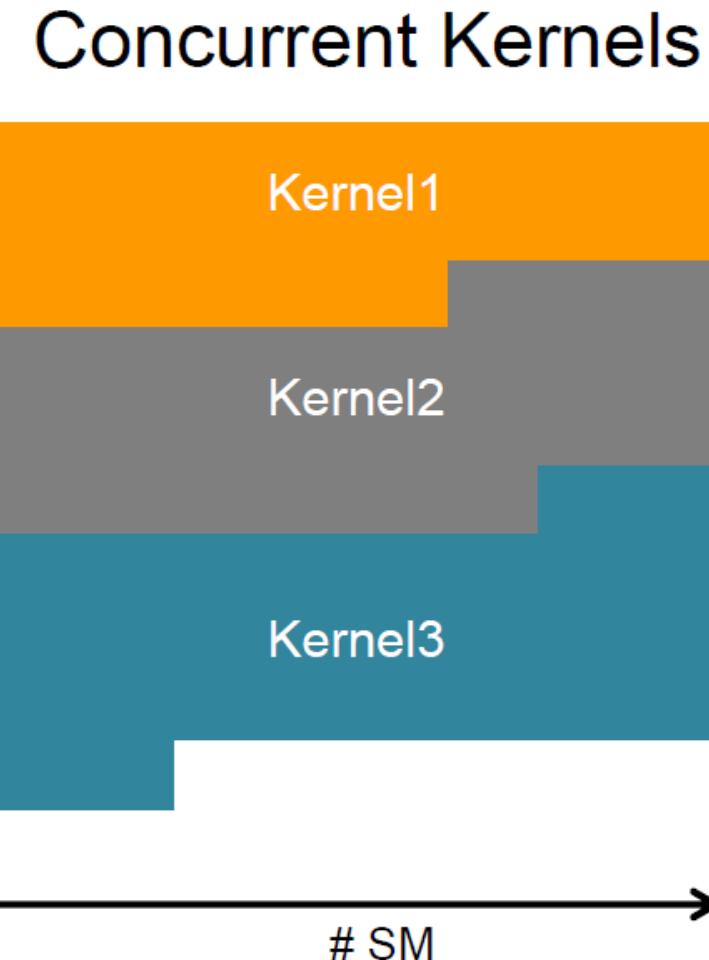
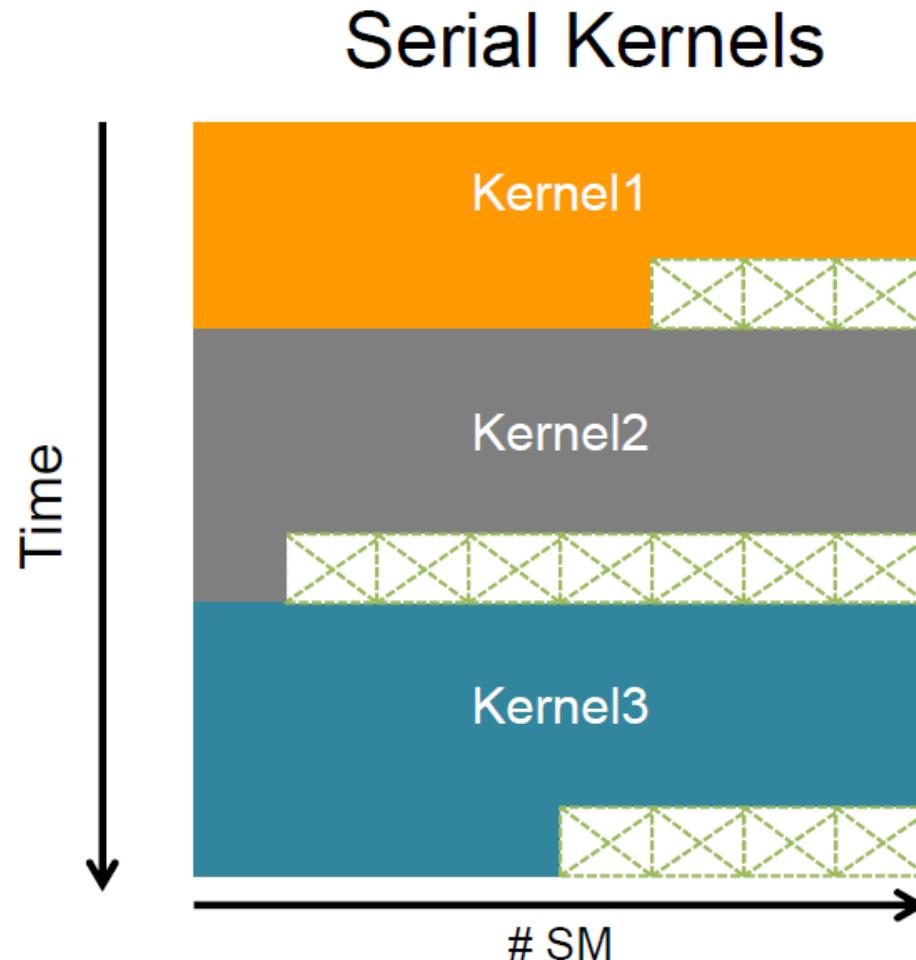
Serial Kernels



Concurrent Kernels



# Concurrent Kernel Execution

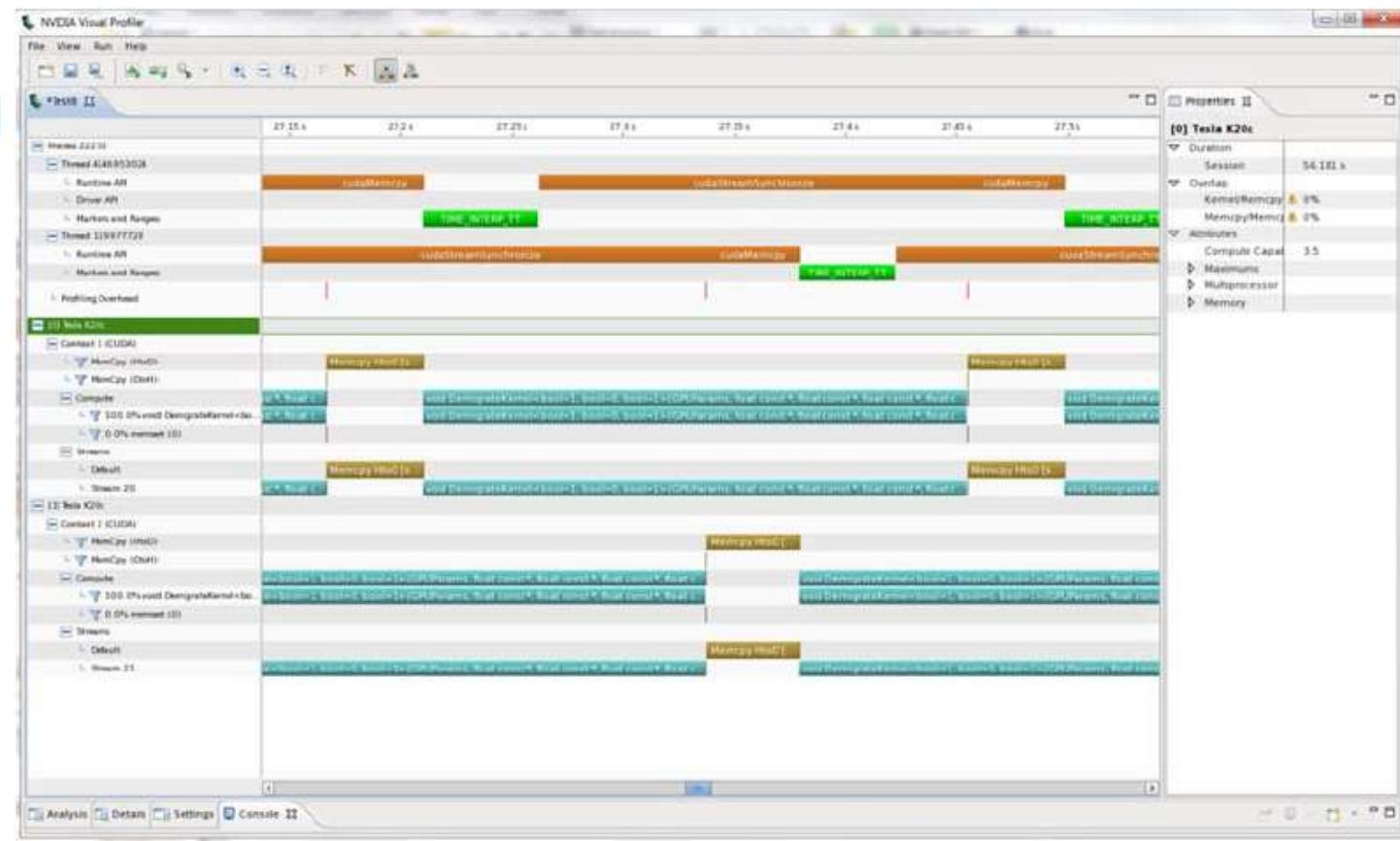


# Pipelining in CUDA

- Pipelining is common in CUDA as you have multiple resources, with each resource specialized for different tasks:
  - CPU – serial algorithms, I/O
  - GPU SMs – parallel algorithms
  - GPU DMA engines – CPU→GPU transfers

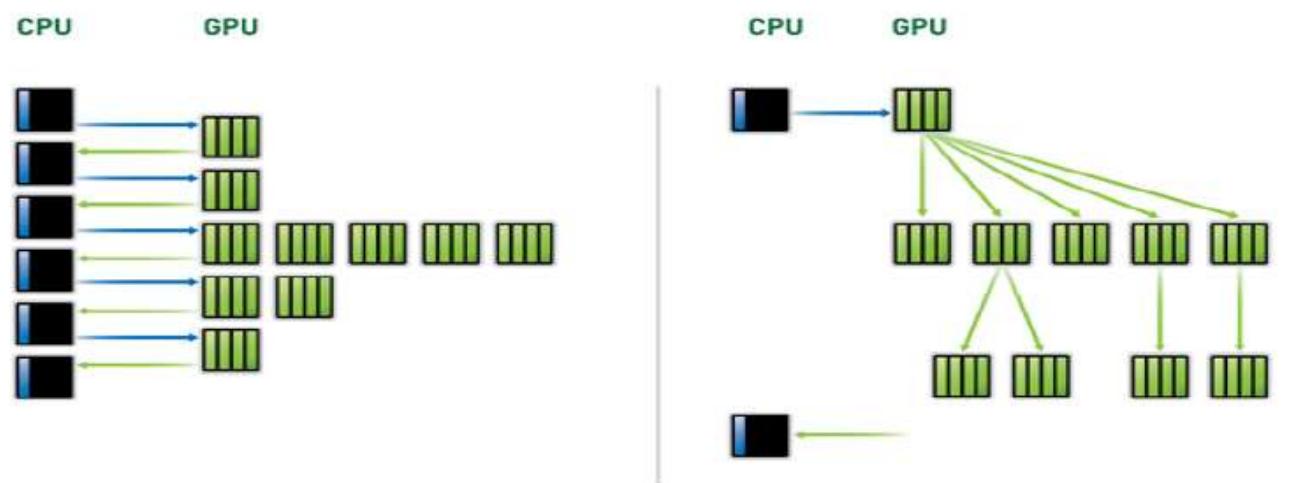
# Profilers and Asynchronous Operations

Timeline views  
useful for visualizing  
actual execution  
order of  
asynchronous  
operations



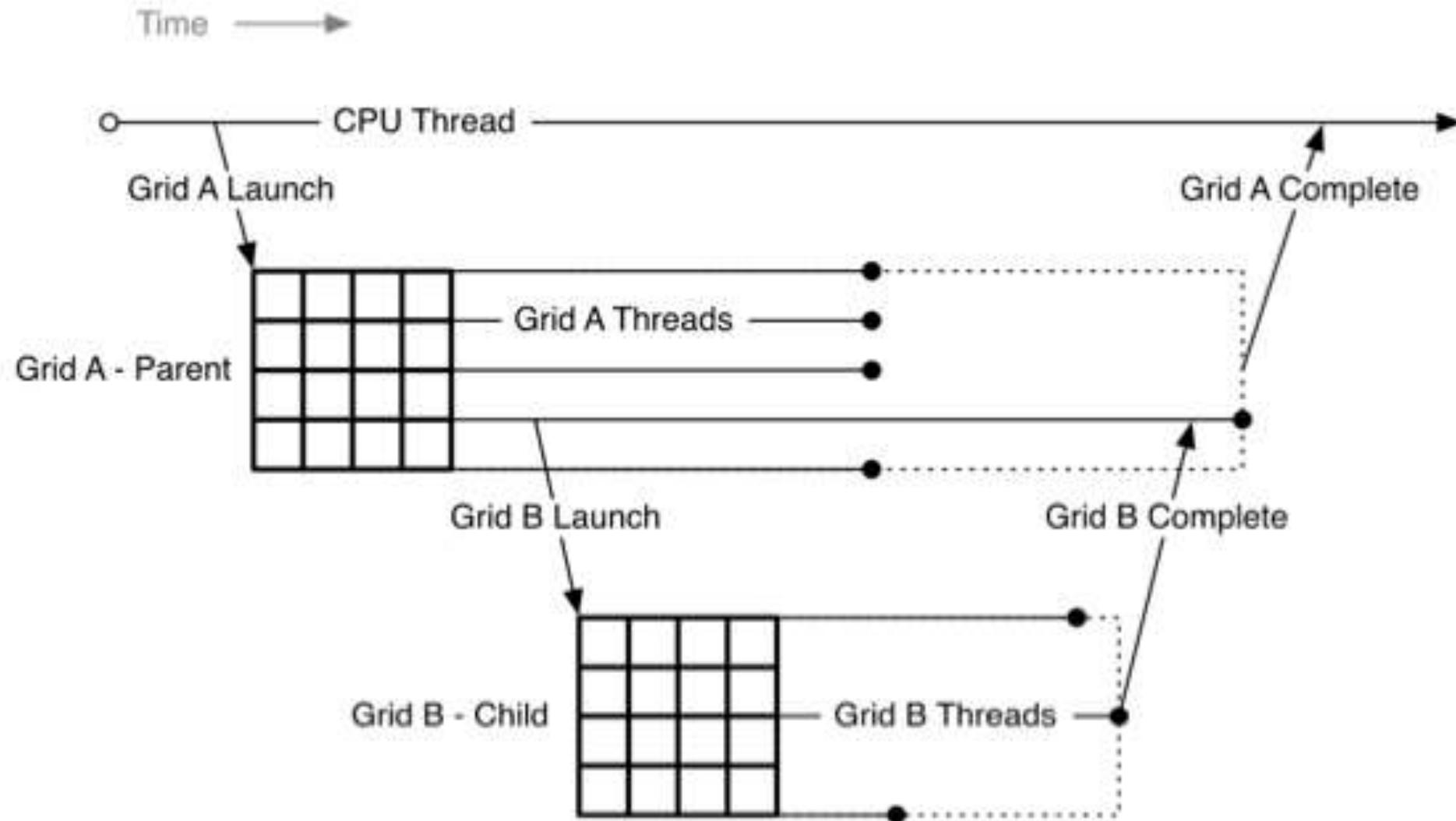
# Dynamic Parallelism

- Allows threads in GPU kernels to launch other GPU kernels
  - GPUs can operate more autonomously from host CPU



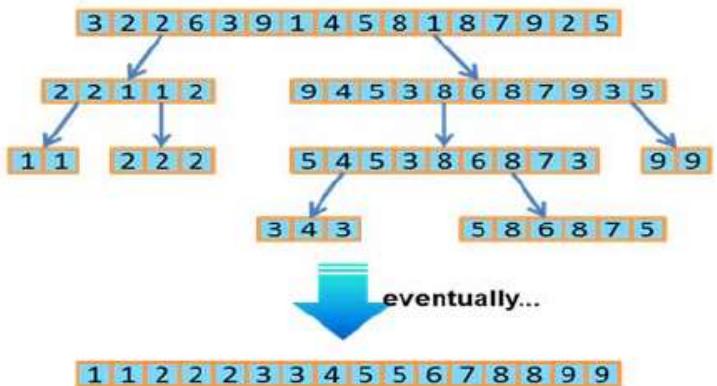
CPU Kernel Launches vs. Dynamic Parallelism

# Grids are fully nested

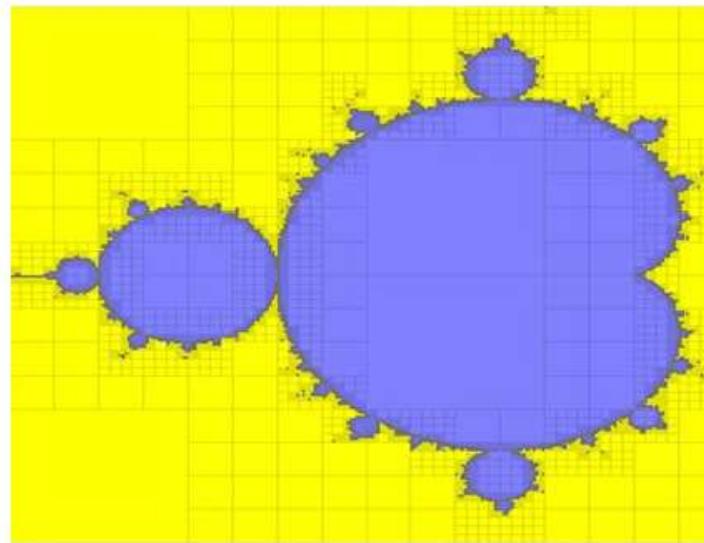


# Dynamic Parallelism

- Dynamic Parallelism is well-suited to algorithms that use recursive subdivision



Quicksort Algorithm



Mariani-Silver Mandelbrot  
Set Algorithm

# Dynamic Parallelism (Continued)

- Potential advantages of Dynamic Parallelism:
  - Simpler code for recursive problems
  - Eliminates device to host data transfers required for host to launch subsequent kernels
  - More efficient
    - Launch kernels from GPU as needed without any need for host synchronization that might introduce false data dependencies
      - eg. Non-Dynamic QuickSort implementation you need to wait for all kernels at one level of the tree are complete before launching any kernels at the next level!

# Dynamic Parallelism

- Use <<<>>> syntax from within a kernel
- Synchronization
  - `cudaDeviceSynchronize()` synchronizes kernels launched from current thread block
  - All work launched by a thread block is implicitly synchronized when the block exits
    - Parent doesn't finish until children have finished

```
_global__ void childKernel()
{
    printf("Hello ");
}

_global__ void parentKernel()
{
    cudaError_t e;
    childKernel<<<1,1>>>();
    cudaDeviceSynchronize();
    e = cudaGetLastError();
    if(e) return;

    printf("World!\n");
}

void host_code()
{
    parentKernel<<<1,1>>>();
    ...
}
```

# Dynamic Parallelism & Memory Model

- Parent and child grids share the same global and constant memory storage
  - You can pass pointers to global/constant memory to child kernels
- Thread-private variables still have thread scope
  - Can't pass pointers to thread-private variables to child kernels
- Shared memory still has thread block scope
  - Can't pass pointers to shared memory to child kernels

```
__global__ void Child(int* );
__global__ void Child2(int );

__constant__ int constantX[10];

__global__ void ParentKernel(int *globalX)
{
    __shared__ int sharedX[10];
    int localX = 5;

    Child<<<1,1>>>(sharedX); ✗
    Child<<<1,1>>>(&localX); ✗
    Child2<<<1,1>>>(sharedX[0]); ✓
    Child2<<<1,1>>>(localX); ✓

    Child<<<1,1>>>(globalX); ✓
    Child<<<1,1>>>(constantX); ✓
}
```

# Dynamic Parallelism

- `cudaDeviceSynchronize()`
  - Parent blocks may pause execution and backup state (program counters, registers/shared memory contents) to host memory and yield to allow child kernels to make forward progress
  - Block may resume executing on a different SM

# Dynamic Parallelism

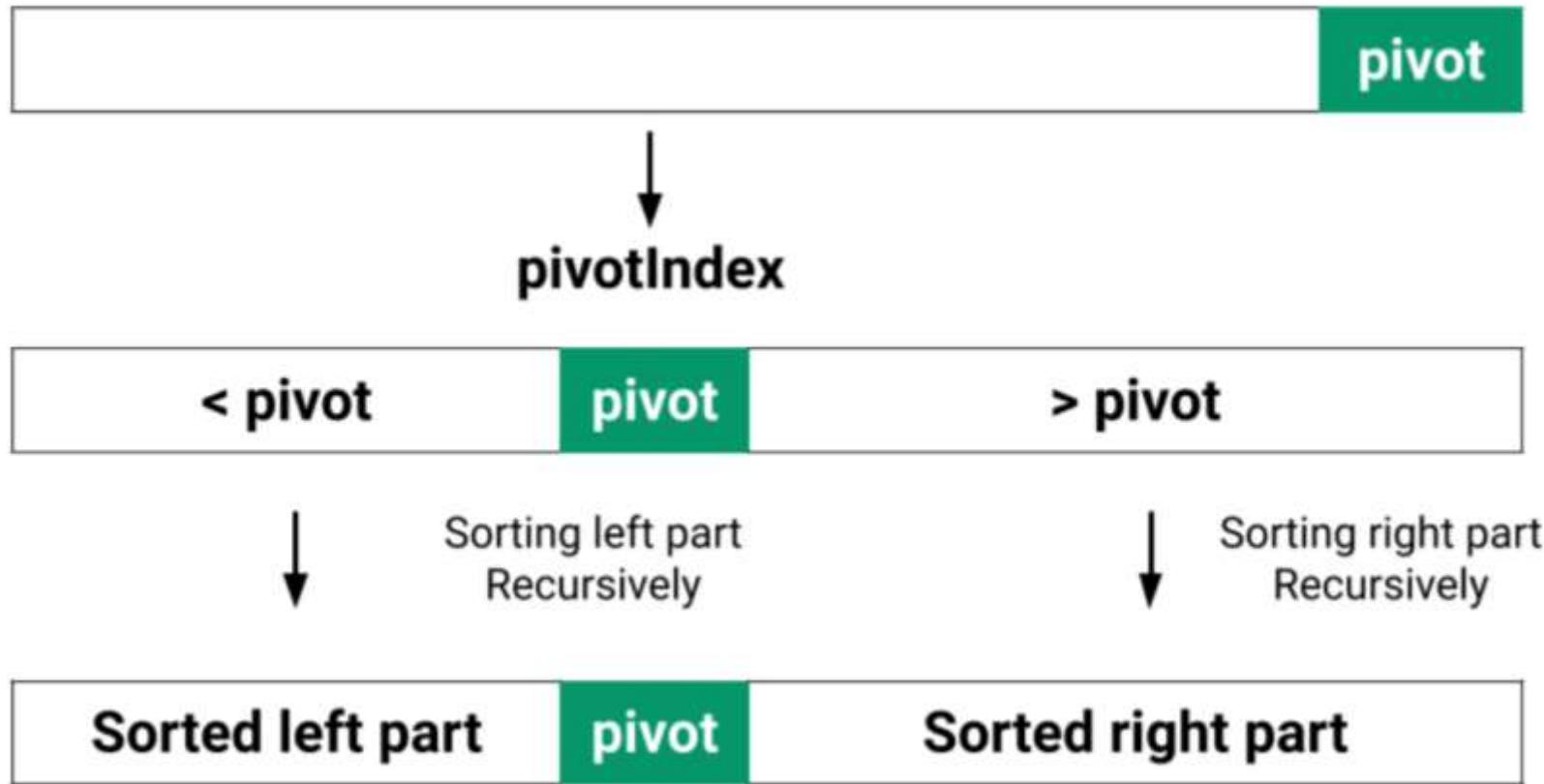
- Kernels launched within a thread-block are executed in order by default
  - Use streams/events to allow concurrent execution of kernels
- Streams and Events are supported
  - A stream and event created by a thread has thread block scope
  - Using streams and events created on the host has undefined behavior
- Create Streams and Events using flags:
  - `cudaStreamCreateWithFlags()` with `cudaStreamNonBlocking` flag
  - `cudaEventCreateWithFlags()` with `cudaEventDisableTiming` flag

# Dynamic Parallelism & Streams

```
__global__ void parentKernel()
{
    cudaStream_t stream1, stream2;
    cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking);
    cudaStreamCreateWithFlags(&stream2, cudaStreamNonBlocking);

    // launch child
    if(threadIdx.x == 0)
    {
        foo<<<1,1,0,stream1>>>();
    }
    else if(threadIdx.x == 32)
    {
        bar<<<1,1,0,stream2>>>();
    }
    ...
}
```

# samples / cdpSimpleQuicksort



# samples / cdpSimpleQuicksort

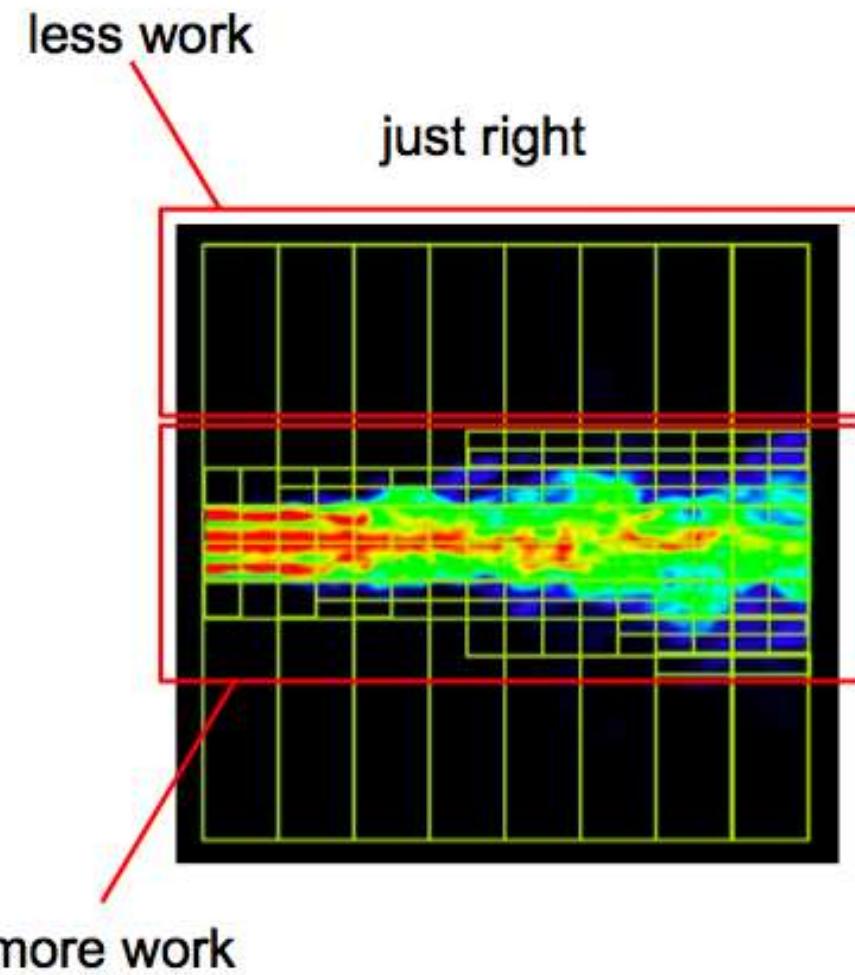
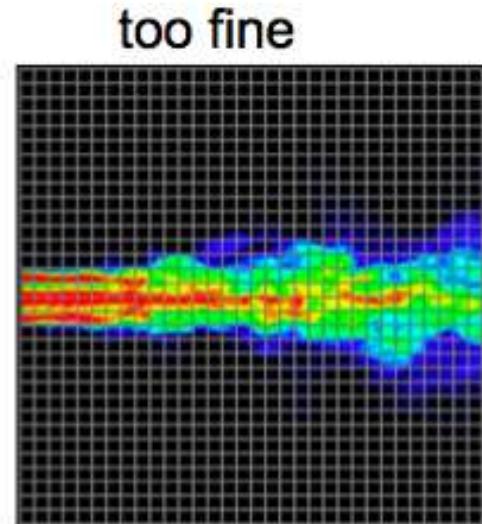
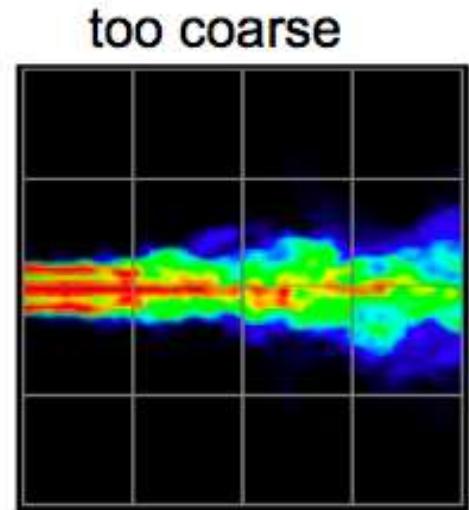
- Basic Quicksort – Left / Right partition of elements about a pivot
- Swap elements on either side of pivot if out of order
- Recursively partition left and right subranges:
  - Launch `cdp_simple_quicksort` recursively (dynamic parallel kernel launch) on left and right partitions
  - If depth is too great (partitions too small), use simple selection sort on partition elements

```
__global__ void cdp_simple_quicksort(unsigned int *data, int left, int right, int depth)
{
    // If we're too deep or there are few elements left, we use an insertion sort...
    if (depth >= MAX_DEPTH || right-left <= INSERTION_SORT)
    {
        selection_sort(data, left, right);
        return;
    }
    ...
}
```

# samples / cdpSimpleQuicksort

```
// Launch a new block to sort the left part.  
if (left < (rptr-data))  
{  
    cudaStream_t s;  
    cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);  
    cdp_simple_quicksort<<< 1, 1, 0, s >>>(data, left, nright, depth+1);  
    cudaStreamDestroy(s);  
}  
  
// Launch a new block to sort the right part.  
if ((lptr-data) < right)  
{  
    cudaStream_t s1;  
    cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);  
    cdp_simple_quicksort<<< 1, 1, 0, s1 >>>(data, nleft, right, depth+1);  
    cudaStreamDestroy(s1);  
}
```

# Adaptive Workload Sizing



# Passing Pointers to Child Grids

Can be passed

- global memory (incl. `__device__` variables and malloc'ed memory)
- zero-copy host memory
- constant memory (inherited and not writeable)

Cannot be passed

- shared memory (`__shared__` variables)
- local memory (incl. stack variables)

The results of dereferencing a pointer in a child grid that cannot be legally passed to it are undefined. The following code on the left is illegal, while the code on the right is OK.

```
// common __global__ void child_k(void *p) { // ... *p = res; // ... }
```

```
__global__ void parent_k(void) {
    // ...
    int v = 0;
    child_k <<< 1, 256 >>> (&v);
    // ...
}
```

```
__device__ int v;
__global__ void parent_k(void) {
    // ...
    child_k <<< 1, 256 >>> (&v);
    // ...
}
```



# samples / mandelbrot-dynamic

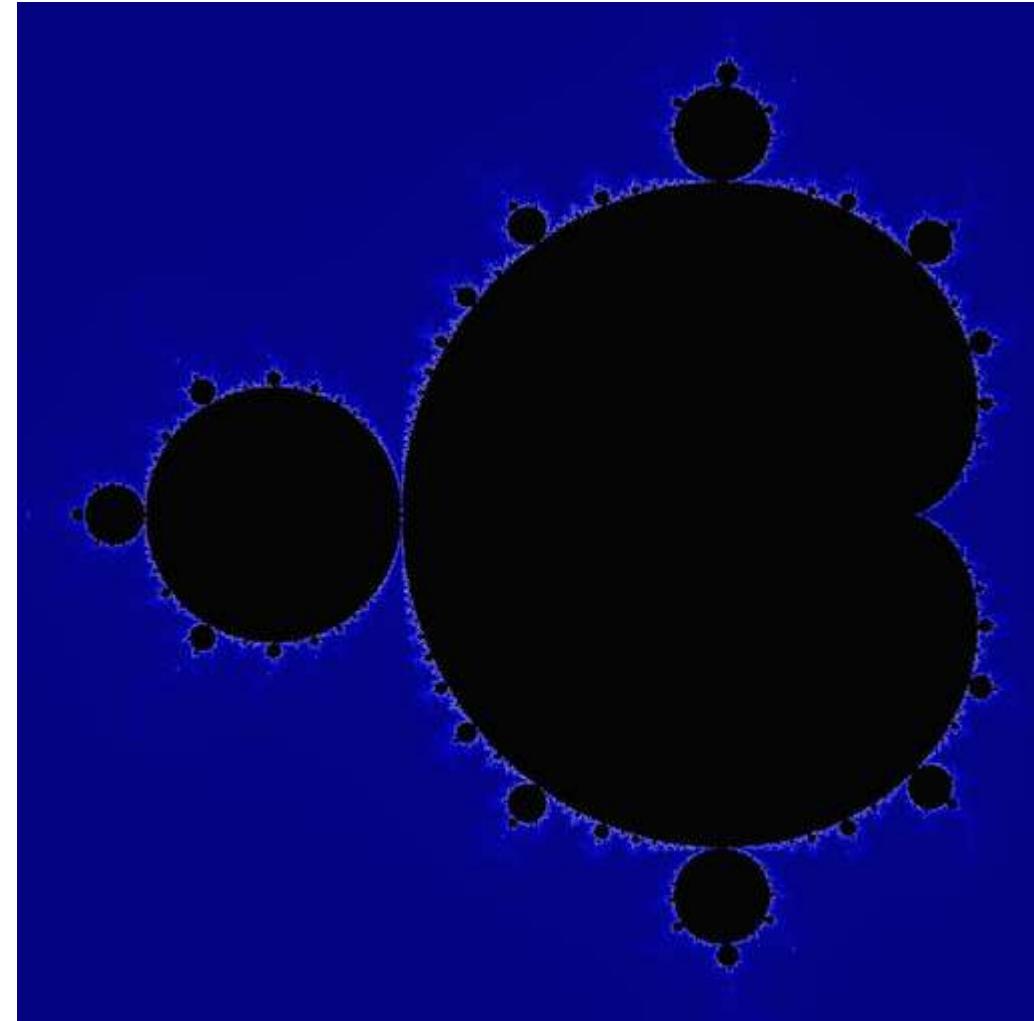
The **Mandelbrot set** is perhaps the best known fractal. It is defined as

$$z_0 = c$$

$$z_{n+1} = z_n^2 + c$$

$$M = \{c \in \mathbb{C} : \exists R \forall n : |z_n| < R\}.$$

The interior of Figure 2 (the black part), is the Mandelbrot Set.



# Simple Per-Pixel Kernel

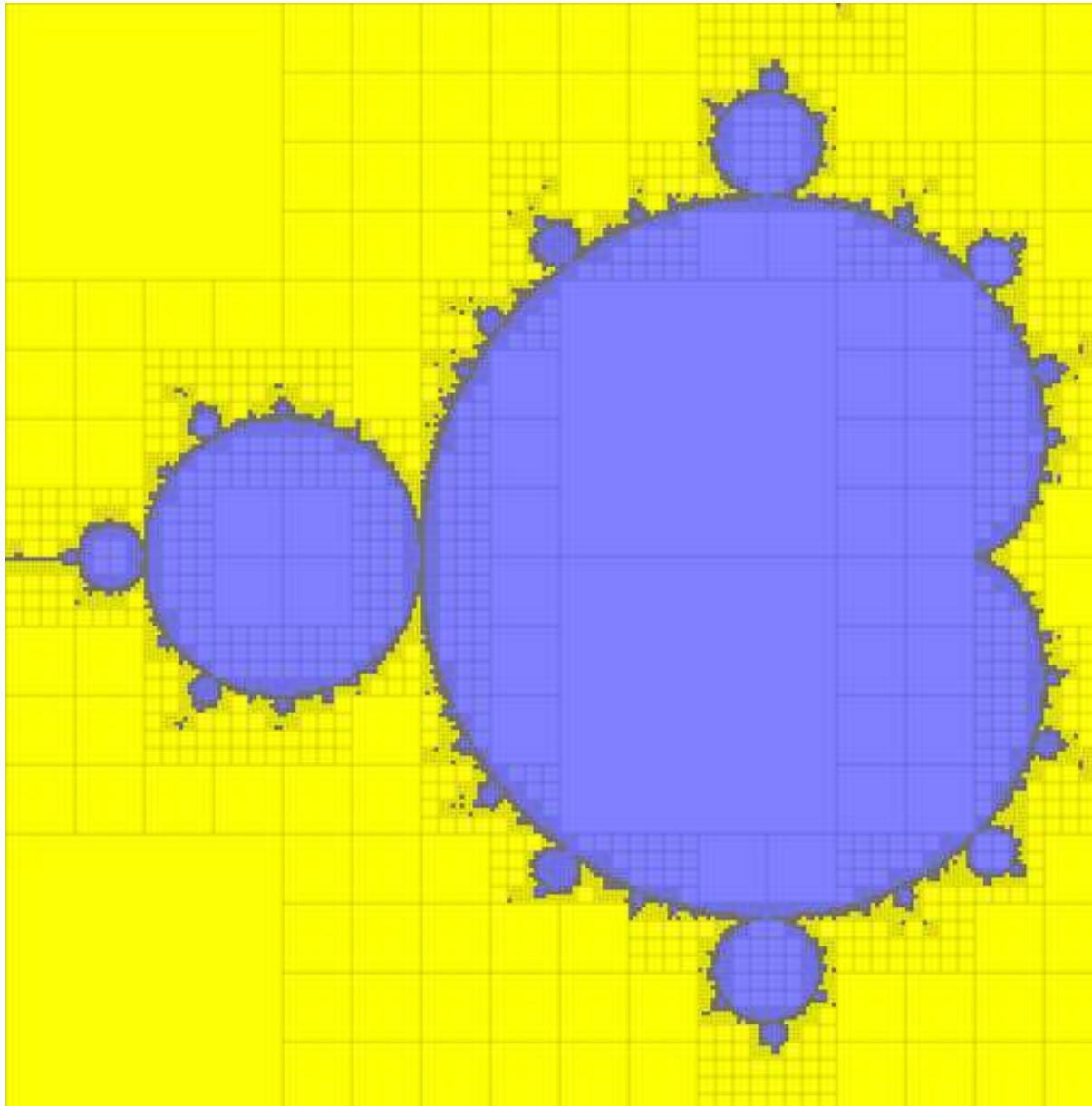
```
// kernel
__global__ void mandelbrot_k(int *dwells,
                             int w, int h,
                             complex cmin, complex cmax) {
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    int y = threadIdx.y + blockDim.y * blockIdx.y;
    if(x < w && y < h)
        dwells[y * w + x] = pixel_dwell(w, h, cmin, cmax, x, y);
}

int main(void) {
    // ... details omitted ...

    // kernel launch
    int w = 4096, h = 4096;
    dim3 bs(64, 4), grid(divup(w, bs.x), divup(h, bs.y));
    mandelbrot_k <<<grid, bs>>>(d_dwells, w, h,
                                      complex(-1.5, 1), complex(0.5, 1));

    // ...
}
```

# Mariani-Silver Hierarchical Algorithm

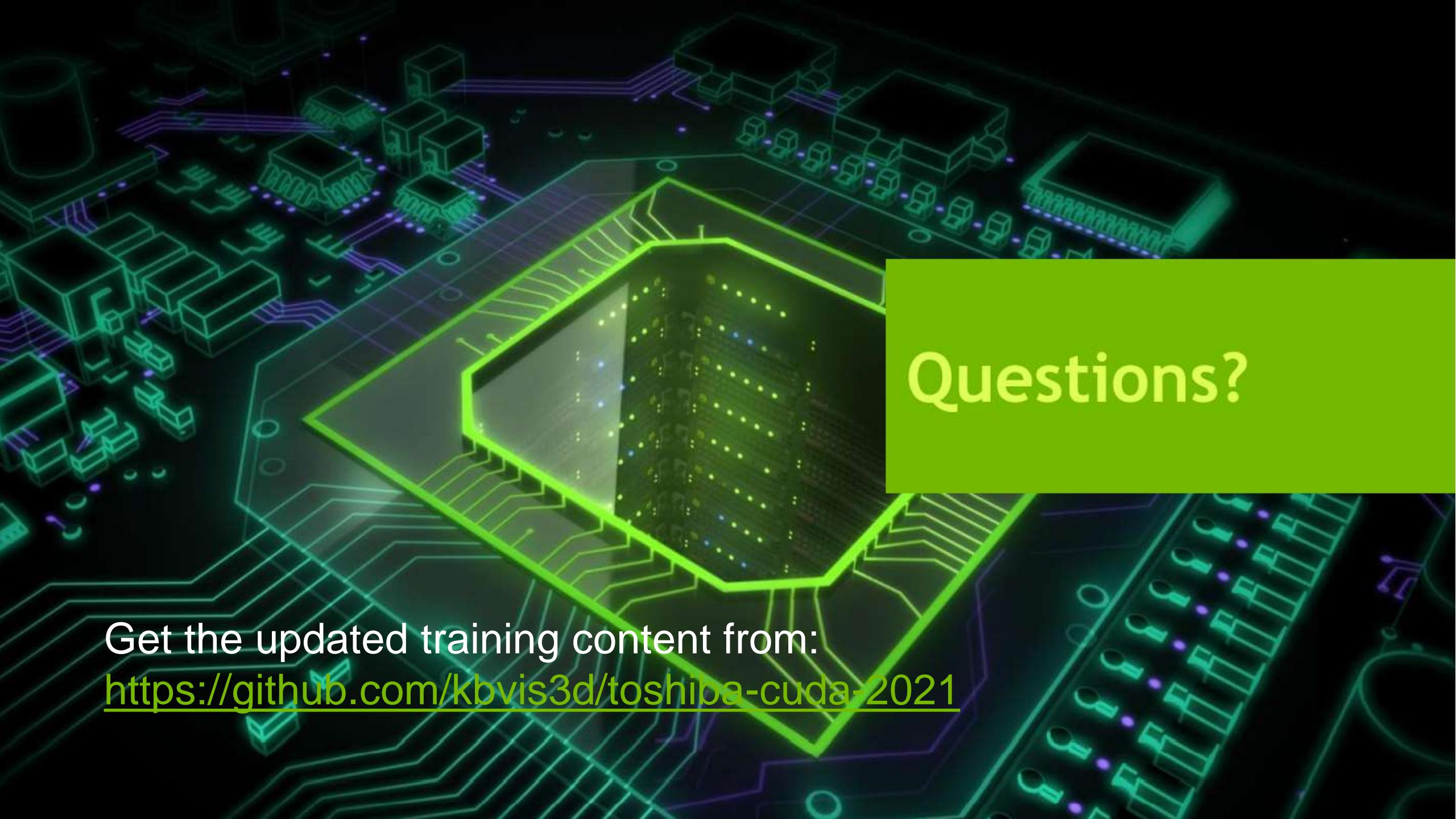


# Adaptive Kernel with Dynamic Parallelism

<https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/>

```
__global__ void mandelbrot_block_k(int *dwells,
                                    int w, int h,
                                    complex cmin, complex cmax,
                                    int x0, int y0,
                                    int d, int depth) {

    x0 += d * blockIdx.x, y0 += d * blockIdx.y;
    int common_dwell = border_dwell(w, h, cmin, cmax, x0, y0, d);
    if (threadIdx.x == 0 && threadIdx.y == 0) {
        if (common_dwell != DIFF_DWELL) {
            // uniform dwell, just fill
            dim3 bs(BSX, BSY), grid(divup(d, BSX), divup(d, BSY));
            dwell_fill<<<grid, bs>>>(dwells, w, x0, y0, d, comm_dwell);
        } else if (depth + 1 < MAX_DEPTH && d / SUBDIV > MIN_SIZE) {
            // subdivide recursively
            dim3 bs(blockDim.x, blockDim.y), grid(SUBDIV, SUBDIV);
            mandelbrot_block_k<<<grid, bs>>>
                (dwells, w, h, cmin, cmax, x0, y0, d / SUBDIV, depth+1);
        } else {
            // leaf, per-pixel kernel
            dim3 bs(BSX, BSY), grid(divup(d, BSX), divup(d, BSY));
            mandelbrot_pixel_k<<<grid, bs>>>
                (dwells, w, h, cmin, cmax, x0, y0, d);
        }
        cucheck_dev(cudaGetLastError());
    }
} // mandelbrot_block_k
```



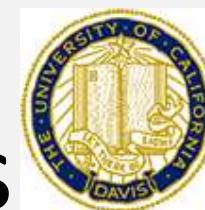
# Questions?

Get the updated training content from:  
<https://github.com/kbvis3d/toshiba-cuda-2021>

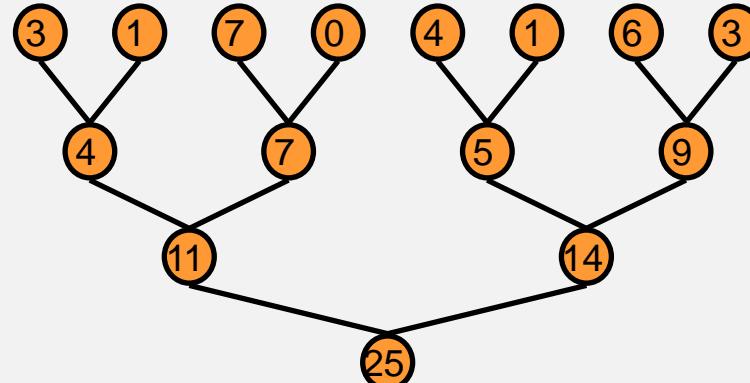


# Advanced Topics: Reduce, Scan, Sort

# Reduction

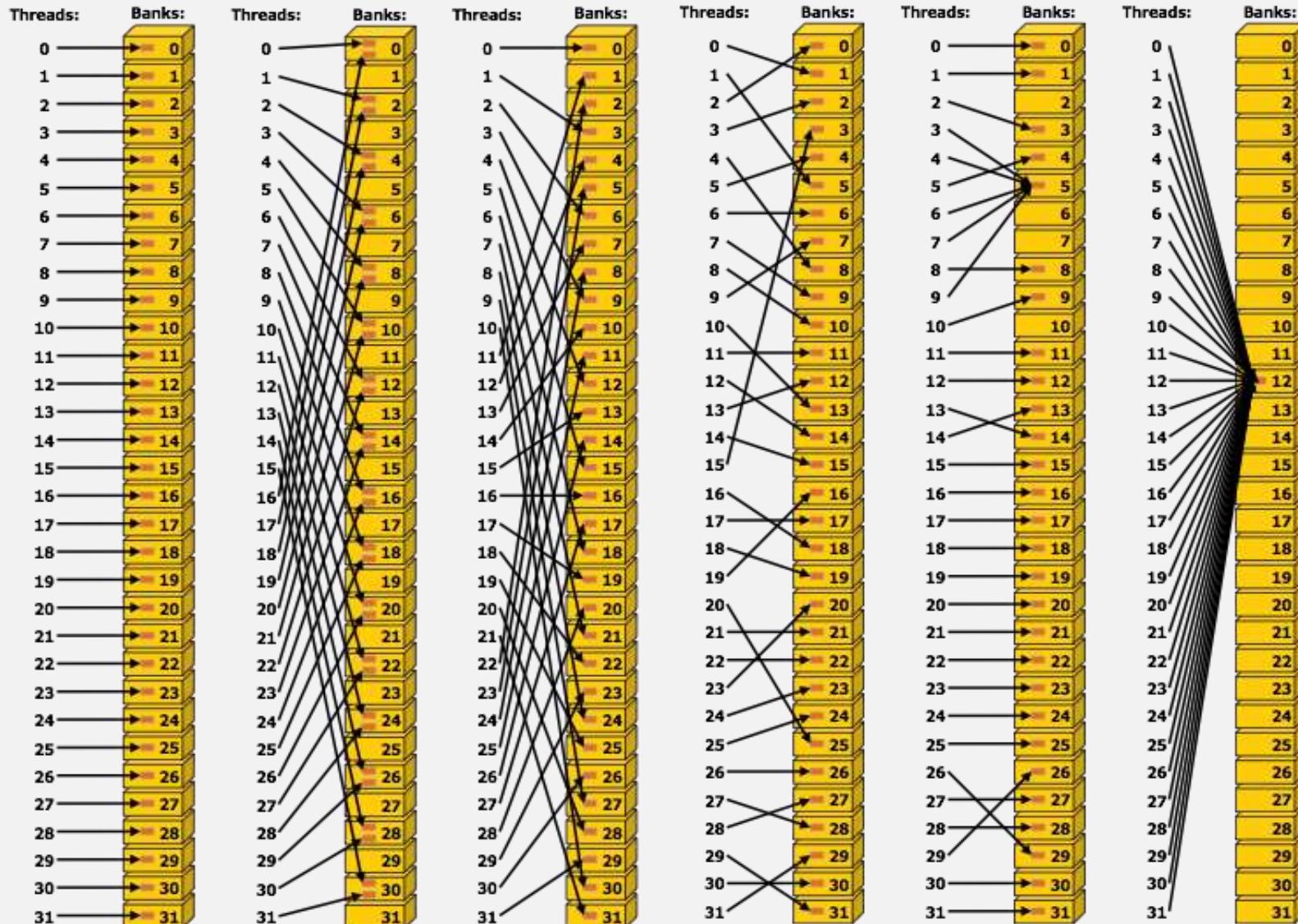


# Tree-Based Parallel Reductions



- Commonly done in traditional GPGPU
  - Ping-pong between render targets, reduce by  $1/2$  at a time
  - Completely bandwidth bound using graphics API
  - Memory writes and reads are off-chip, no reuse of intermediate sums
- CUDA solves this by exposing on-chip shared memory
  - Reduce blocks of data in shared memory to save bandwidth

# CUDA Bank Conflicts



Left: Linear addressing with a stride of one 32-bit word (no bank conflict).

Middle: Linear addressing with a stride of two 32-bit words (2-way bank conflicts).

Right: Linear addressing with a stride of three 32-bit words (no bank conflict).

Left: Conflict-free access via random permutation.

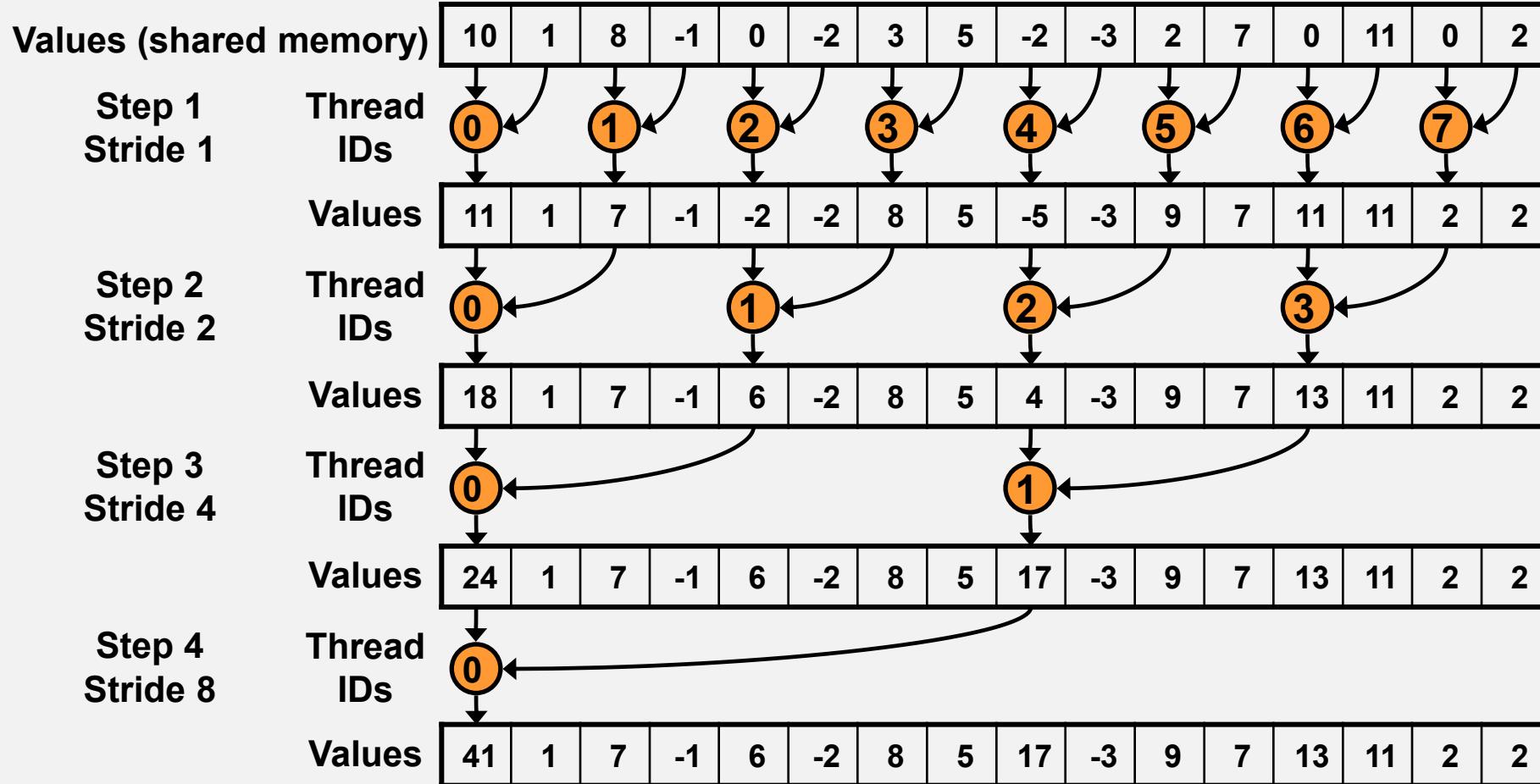
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.

Right: Conflict-free broadcast access (all threads access the same word).



# Parallel Reduction: Interleaved Addressing

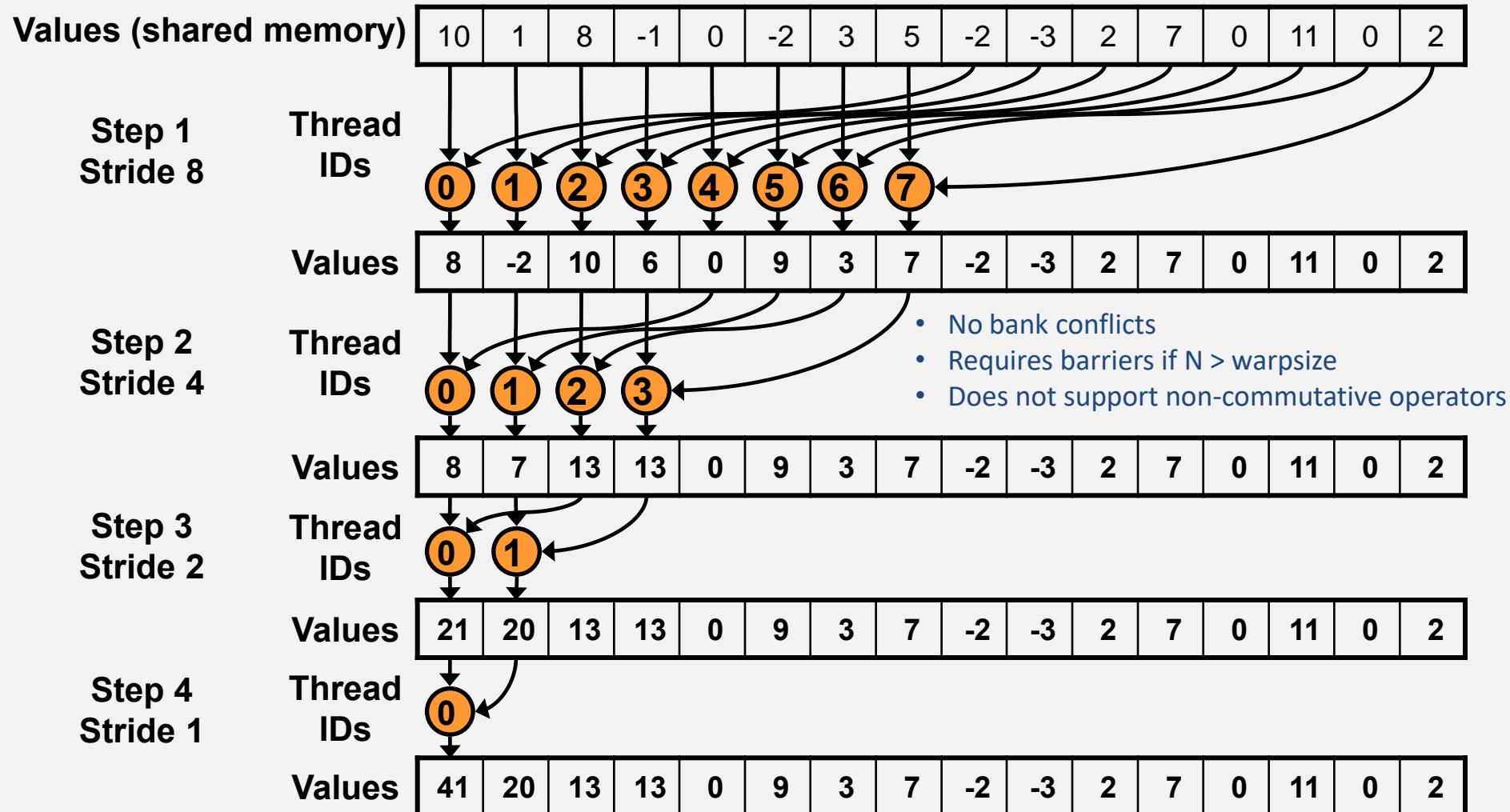
- Arbitrarily bad bank conflicts
- Requires barriers if  $N >$  warpsize
- Supports non-commutative operators



Interleaved addressing results in bank conflicts

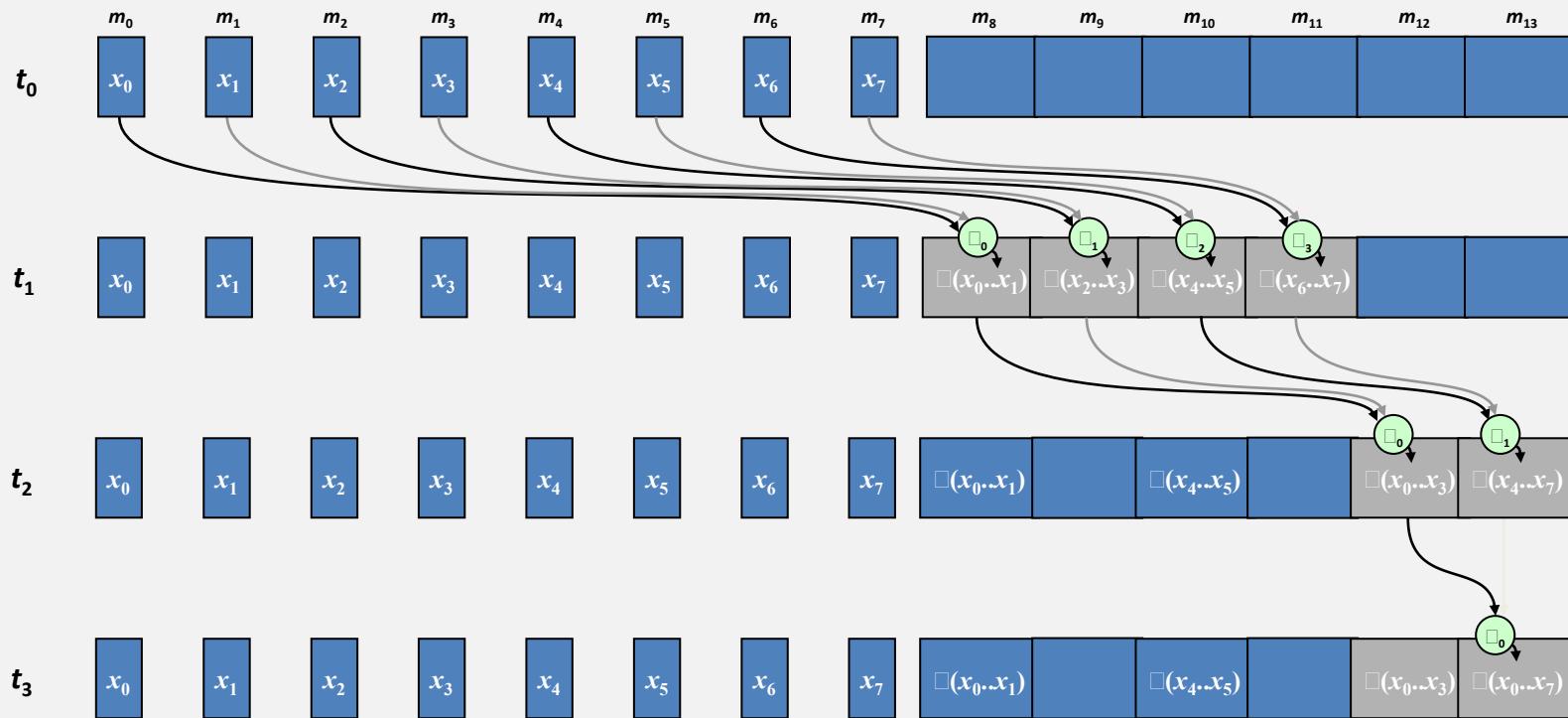


# Parallel Reduction: Sequential Addressing



# Reduction

- Only two-way bank conflicts
- Requires barriers if  $N >$  warp size
- Requires  $O(2N-2)$  storage
- Supports non-commutative operators



# Reduction memory traffic

- Ideal:  $n$  reads, 1 write.
- Block size 256 threads. Thus:
  - Read  $n$  items, write back  $n/256$  items.
  - Read  $n/256$  items, write back 1 item.
  - Total:  $n + n/128 + 1$ . Not bad!

# Reduction optimization

- Ideal:  $n$  reads, 1 write.
- Block size 256 threads. Thus:
  - Read  $n$  items, write back  $n/256$  items.
  - Read  $n/256$  items, write back 1 item.
  - Total:  $n + n/128 + 1$ . Not bad!
- What if we had more than one item (say, 4) per thread?
  - This is an optimization for all the algorithms I talk about today.
  - Tradeoff: Storage for efficiency

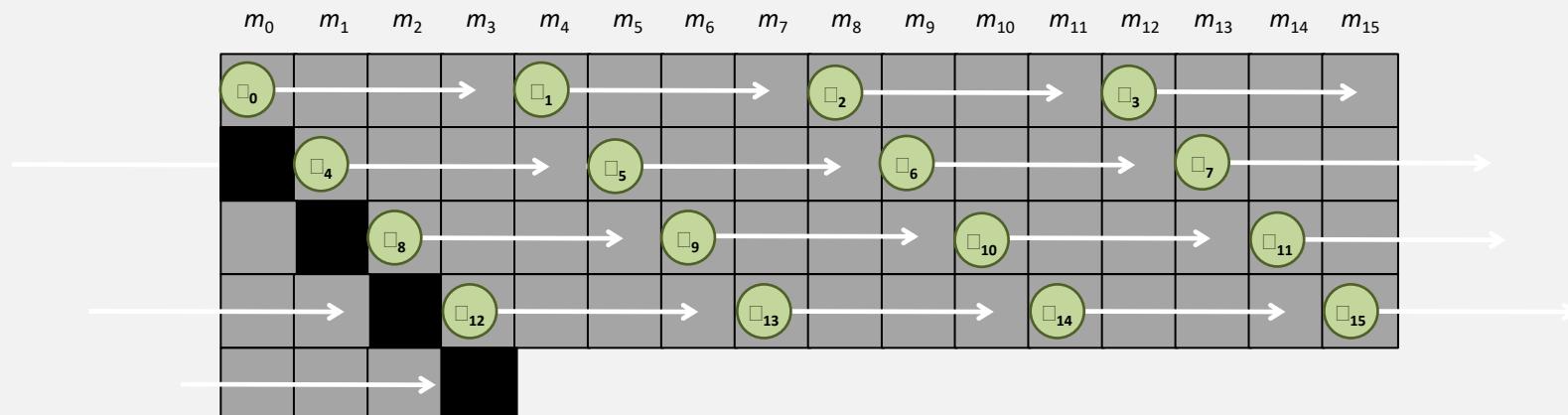
# Persistent Threads

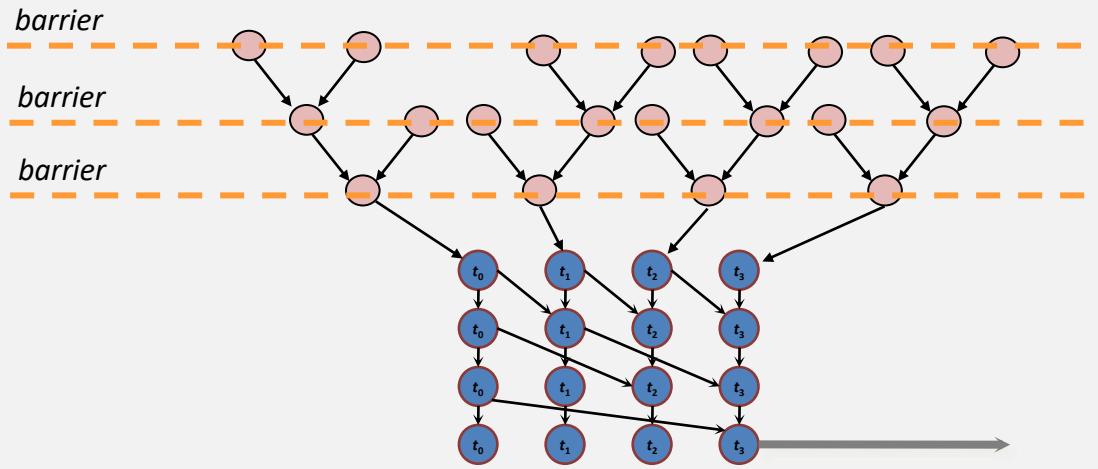
- GPU programming model suggests one thread per item
- What if you filled the machine with just enough threads to keep all processors busy, then asked each thread to stay alive until the input was complete?
- Minus: More overhead per thread (register pressure)
- Minus: Violent anger of vendors

# (Serial) Raking Reduction Phase

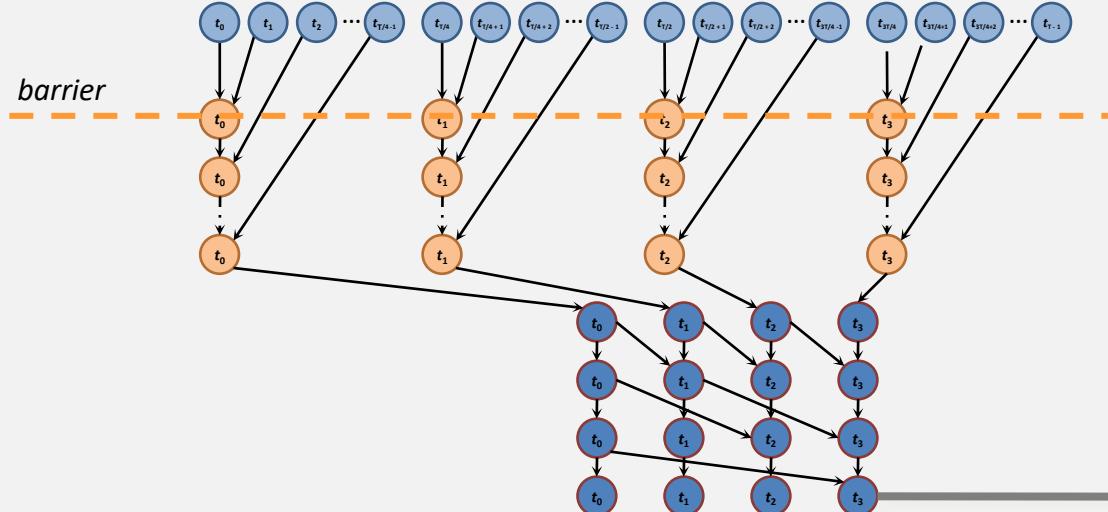
- No bank conflicts, only one barrier to after insertion into smem
- Supports non-commutative operators
- Requires subsequent warp scan to reduce accumulated partials

- Less memory bandwidth overall
- Exploits locality between items within a thread's registers

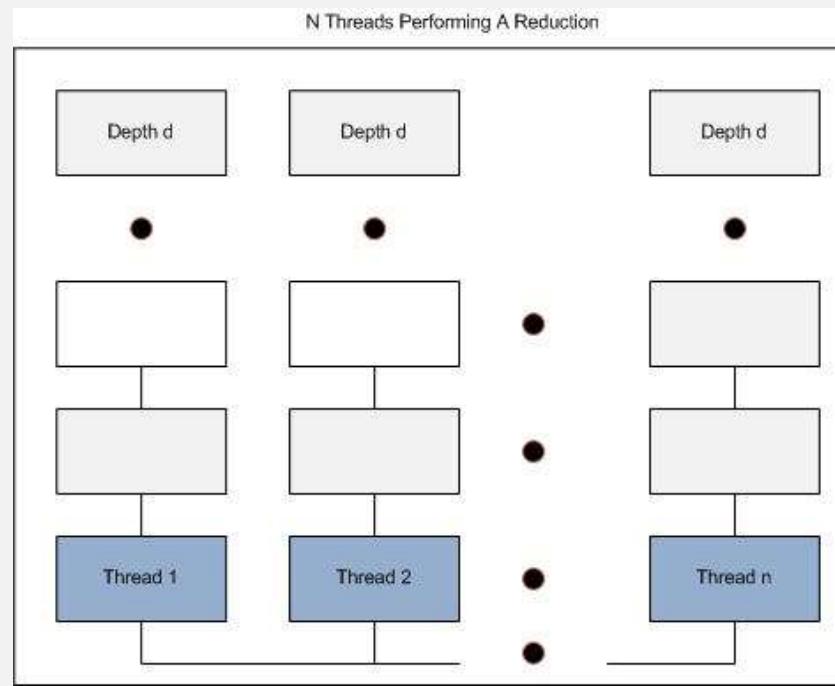




VS.



# Reduction



- Many-To-One
  - Parameter to Tune => **Thread Width** (total number of threads)

# Reduction papers

- Mark Harris, Mapping Computational Concepts to GPUs, GPU Gems 2, Chapter 31, pp. 495–508, March 2005.
- Andrew Davidson and John D. Owens. Toward Techniques for Auto-Tuning GPU Algorithms. In Para 2010: State of the Art in Scientific and Parallel Computing, June 2010.
- NVIDIA SDK (reduction example)

# samples / simpleCooperativeGroups

To use Cooperative Groups, include its header file.

```
#include <cooperative_groups.h>
```

Cooperative Groups types and interfaces are defined in the `cooperative_groups` C++ namespace, so you can either prefix all names and functions with `cooperative_groups::`, or load the namespace or its types with `using` directives.

```
using namespace cooperative_groups; // or...
using cooperative_groups::thread_group; // etc.
```

It's not uncommon to alias it to something shorter. Assume the following namespace alias exists in the examples in this post.

```
namespace cg = cooperative_groups;
```

Code containing any intra-block Cooperative Groups functionality can be compiled in the normal way using `nvcc` (note that many of the examples in this post use `C++11` features so you need to add the `--std=c++11` option to the compilation command line).

<https://developer.nvidia.com/blog/cooperative-groups/>

# samples / simpleCooperativeGroups

## Thread Groups

The fundamental type in Cooperative Groups is `thread_group`, which is a handle to a group of threads. The handle is only accessible to members of the group it represents. Thread groups expose a simple interface. You can get the size (total number of threads) of a group with the `size()` method:

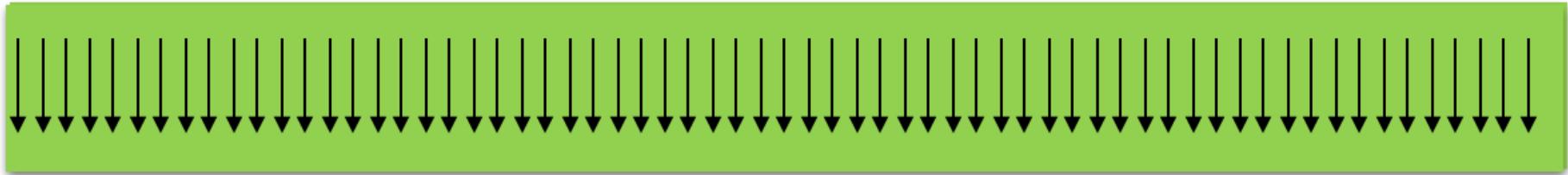
```
unsigned size();
```

To find the index of the calling thread (between `0` and `size()-1`) within the group, use the `thread_rank()` method:

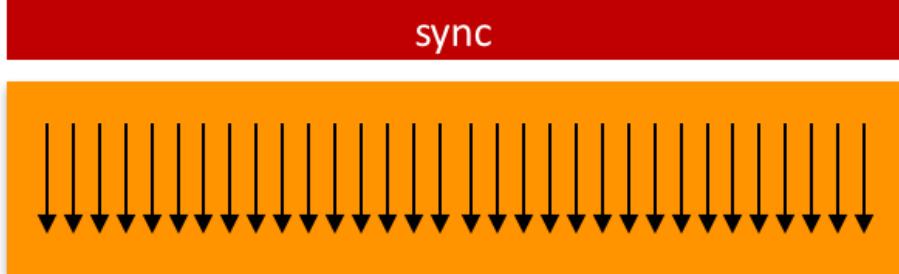
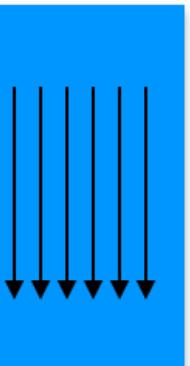
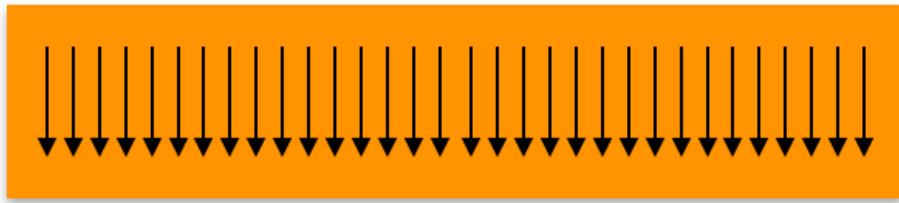
```
unsigned thread_rank();
```

Finally, you can check the validity of a group using the `is_valid()` method.

```
bool is_valid();
```

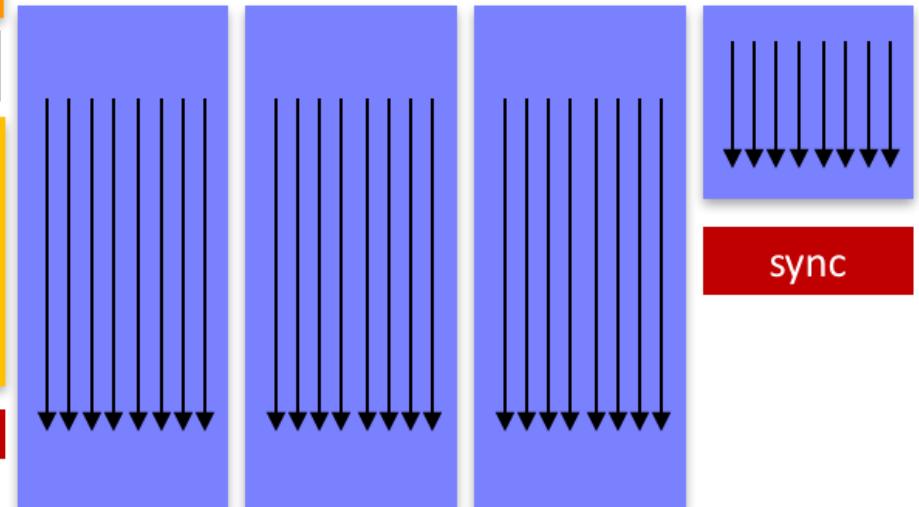
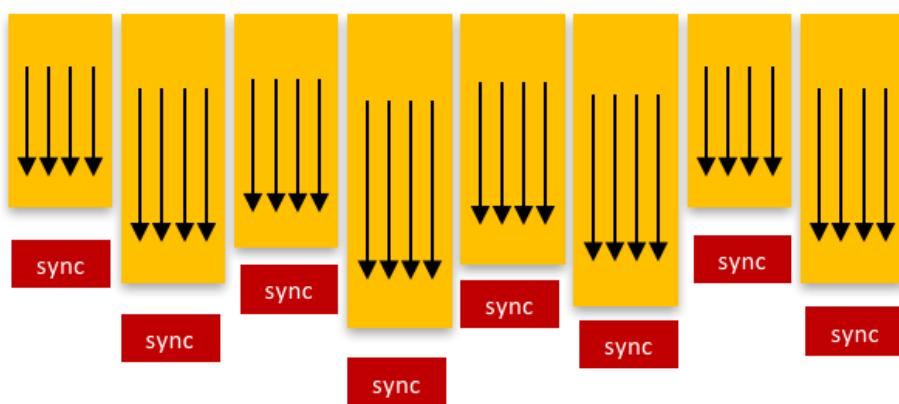


tiled\_partition



tiled\_partition

tiled\_partition



sync

## samples / simpleCooperativeGroups

```
using namespace cooperative_groups;
__device__ int reduce_sum(thread_group g, int *temp, int val)
{
    int lane = g.thread_rank();

    // Each iteration halves the number of active threads
    // Each thread adds its partial sum[i] to sum[lane+i]
    for (int i = g.size() / 2; i > 0; i /= 2)
    {
        temp[lane] = val;
        g.sync(); // wait for all threads to store
        if(lane<i) val += temp[lane + i];
        g.sync(); // wait for all threads to load
    }
    return val; // note: only thread 0 will return full sum
}
```

```
__device__ int thread_sum(int *input, int n)
{
    int sum = 0;

    for(int i = blockIdx.x * blockDim.x + threadIdx.x;
        i < n / 4;
        i += blockDim.x * gridDim.x)
    {
        int4 in = ((int4*)input)[i];
        sum += in.x + in.y + in.z + in.w;
    }
    return sum;
}

__global__ void sum_kernel_block(int *sum, int *input, int n)
{
    int my_sum = thread_sum(input, n);

    extern __shared__ int temp[];
    auto g = this_thread_block();
    int block_sum = reduce_sum(g, temp, my_sum);

    if (g.thread_rank() == 0) atomicAdd(sum, block_sum);
}
```

# samples / simpleCooperativeGroups

```
int n = 1<<24;
int blockSize = 256;
int nBlocks = (n + blockSize - 1) / blockSize;
int sharedBytes = blockSize * sizeof(int);

int *sum, *data;
cudaMallocManaged(&sum, sizeof(int));
cudaMallocManaged(&data, n * sizeof(int));
std::fill_n(data, n, 1); // initialize data
cudaMemset(sum, 0, sizeof(int));

sum_kernel_block<<<nBlocks, blockSize, sharedBytes>>>(sum, data, n);
```

## samples / simpleCooperativeGroups

```
_global_ void sum_kernel_32(int *sum, int *input, int n)
{
    int my_sum = thread_sum(input, n);

    extern __shared__ int temp[];

    auto g = this_thread_block();
    auto tileIdx = g.thread_rank() / 32;
    int* t = &temp[32 * tileIdx];

    auto tile32 = tiled_partition(g, 32);
    int tile_sum = reduce_sum(tile32, t, my_sum);

    if (tile32.thread_rank() == 0) atomicAdd(sum, tile_sum);
}
```

# Warp-level Collectives: shfl\_down()

```
template <int tile_sz>
__device__ int reduce_sum_tile_shfl(thread_block_tile<tile_sz> g, int val)
{
    // Each iteration halves the number of active threads
    // Each thread adds its partial sum[i] to sum[lane+i]
    for (int i = g.size() / 2; i > 0; i /= 2) {
        val += g.shfl_down(val, i);
    }

    return val; // note: only thread 0 will return full sum
}

template<int tile_sz>
__global__ void sum_kernel_tile_shfl(int *sum, int *input, int n)
{
    int my_sum = thread_sum(input, n);

    auto tile = tiled_partition<tile_sz>(this_thread_block());
    int tile_sum = reduce_sum_tile_shfl<tile_sz>(tile, my_sum);

    if (tile.thread_rank() == 0) atomicAdd(sum, tile_sum);
}
```

NOTE: Use  
\_\_shfl\_down\_sync()  
instead for Compute  
Capability 7.x and above.

# Warp-level Collectives

```
T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize);
T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int width=warpSize);
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=warpSize);
```

`__shfl_sync()`

Direct copy from indexed lane

`__shfl_up_sync()`

Copy from a lane with lower ID relative to caller

`__shfl_down_sync()`

Copy from a lane with higher ID relative to caller

`__shfl_xor_sync()`

Copy from a lane based on bitwise XOR of own lane ID

The exchange occurs simultaneously for all active threads within the warp (and named in *mask*)

See: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-shuffle-functions>

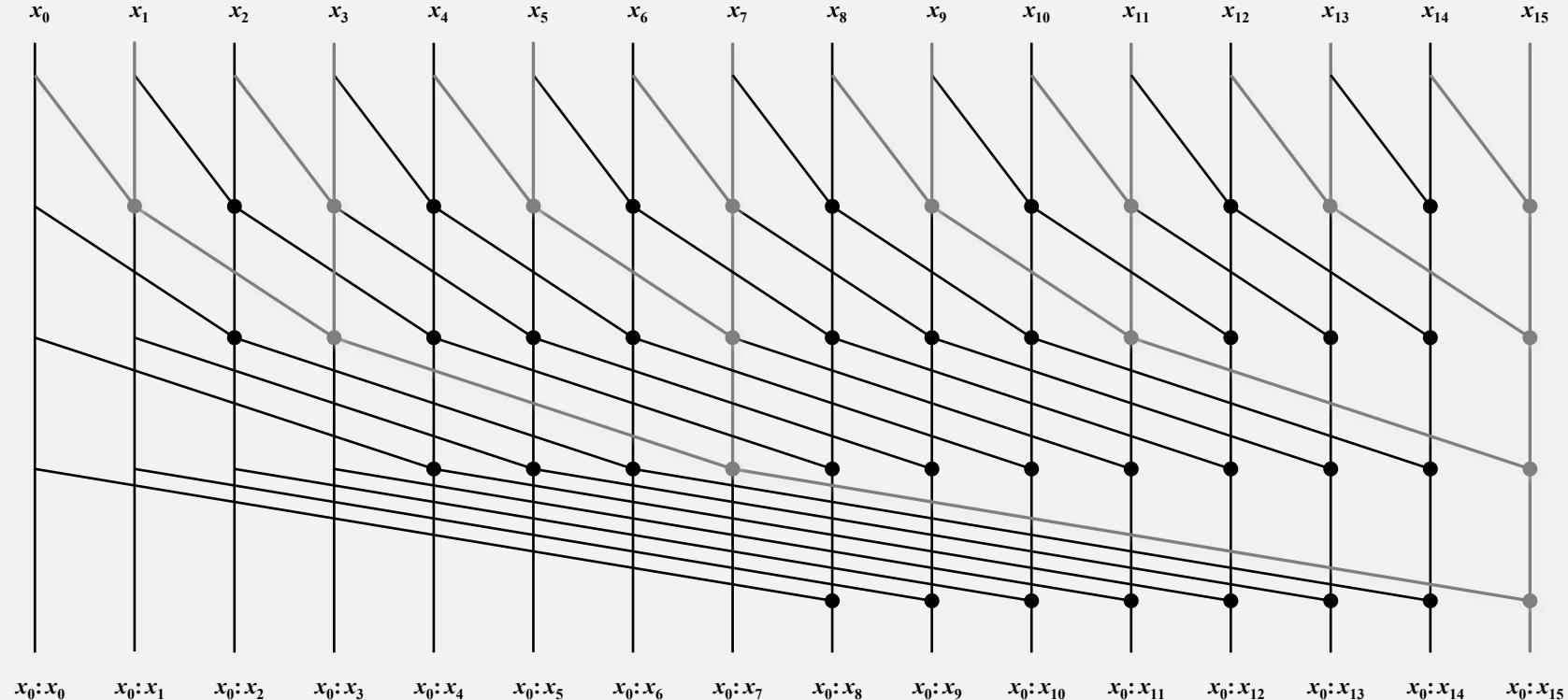
# Scan (within a block)

# Parallel Prefix Sum (Scan)

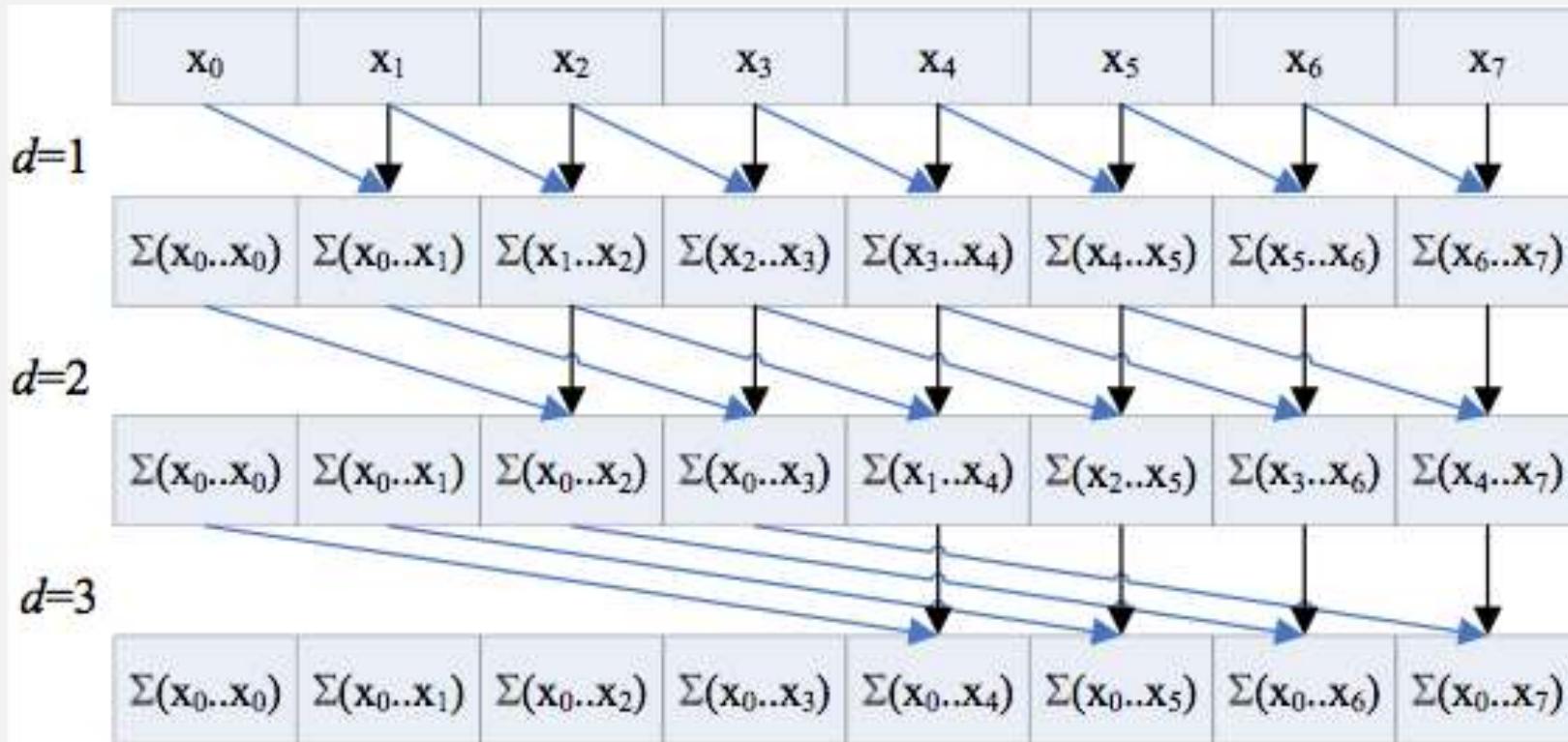
- Given an array  $A = [a_0, a_1, \dots, a_{n-1}]$  and a binary associative operator  $\oplus$  with identity  $I$ ,
- $\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})]$
- Example: if  $\oplus$  is addition, then scan on the set
  - $[3 1 7 0 4 1 6 3]$
- returns the set
  - $[0 3 4 11 11 15 16 22]$

# Kogge-Stone Scan

Circuit family



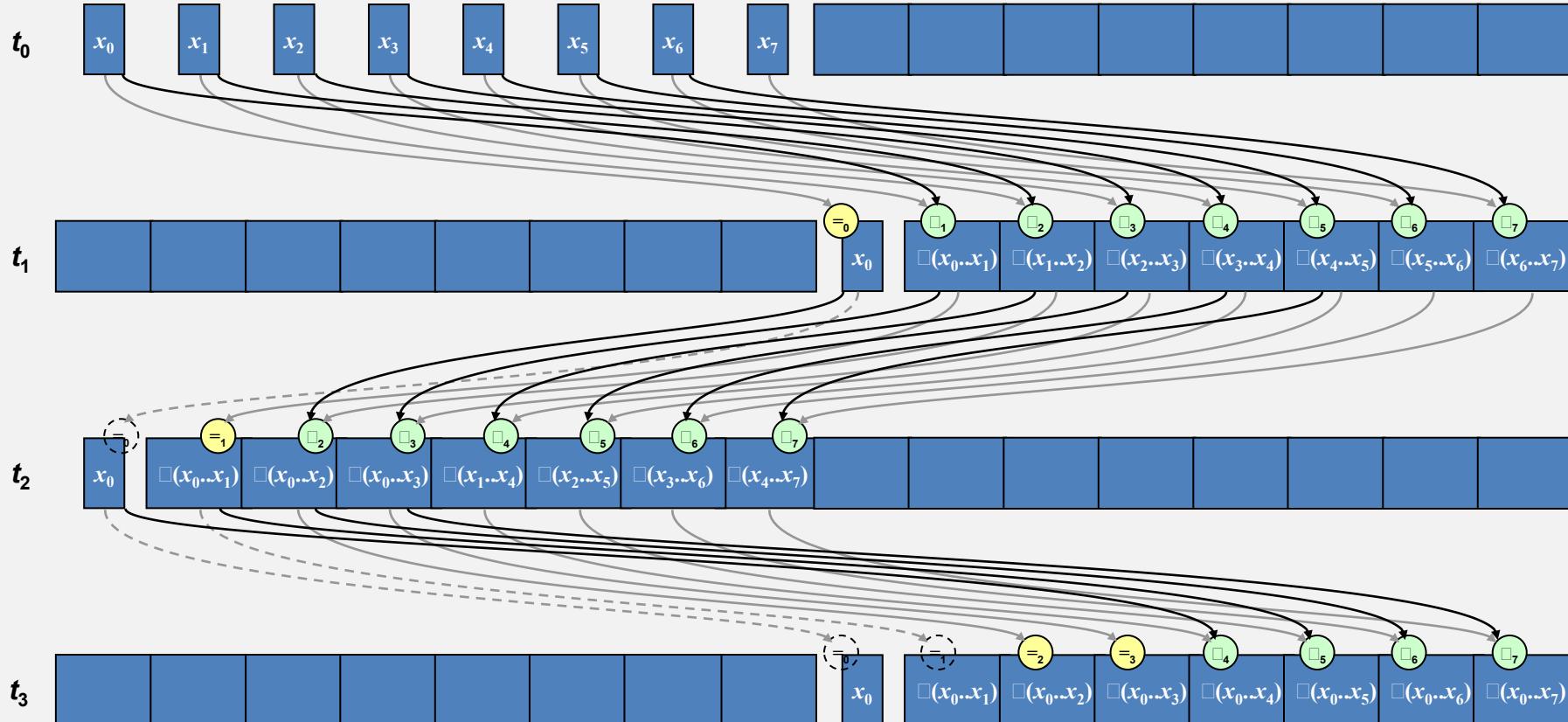
# $O(n \log n)$ Scan



- Step efficient ( $\log n$  steps)
- Not work efficient ( $n \log n$  work)
- Requires barriers at each step (WAR dependencies)

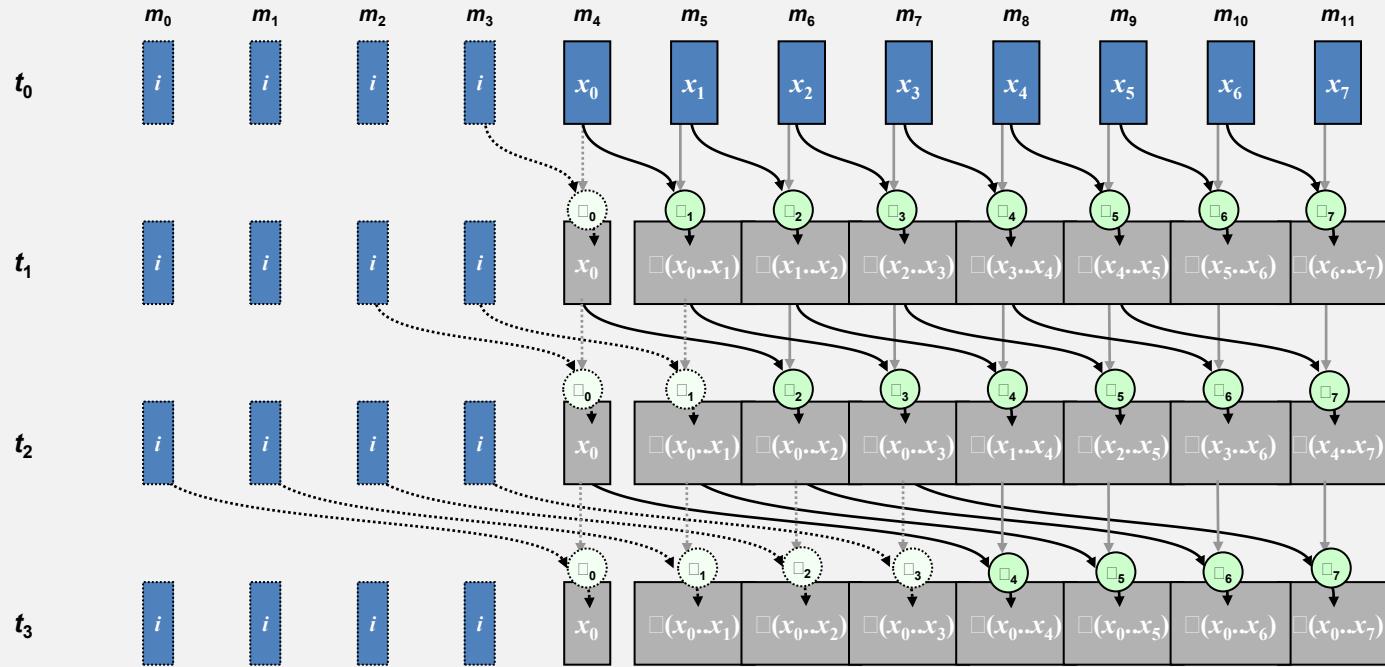
# Alt. Hillis-Steele Scan Implementation

No WAR conflicts,  $O(2N)$  storage



# Alt. Hillis-Steele Scan

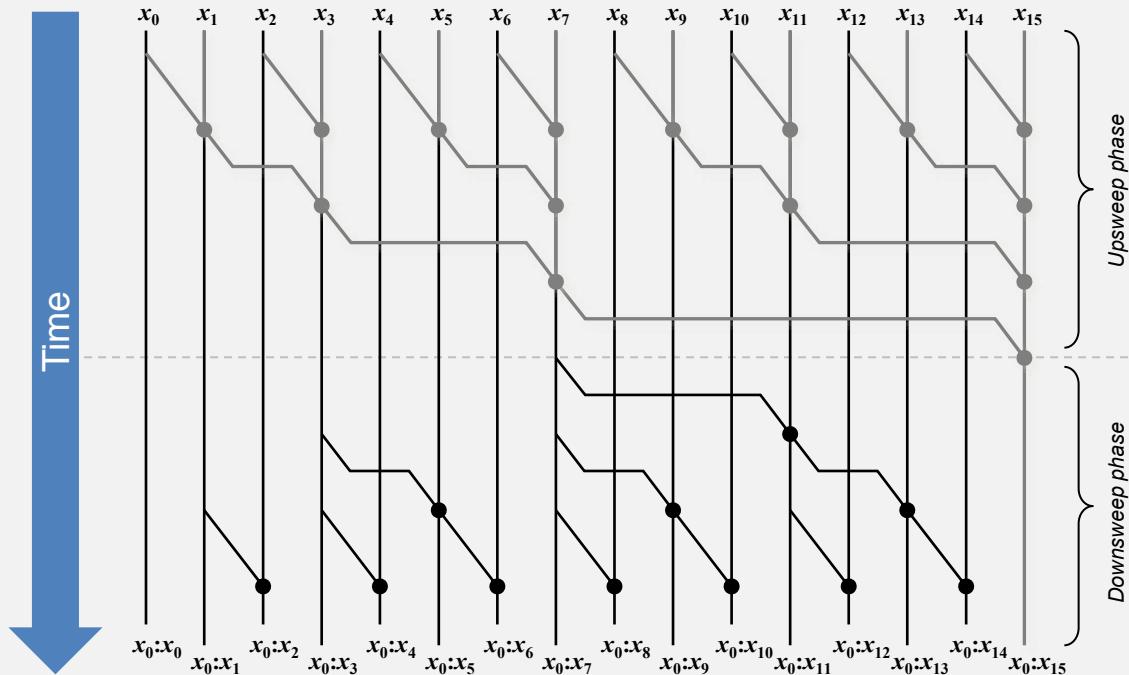
Warp-synchronous: SIMD without divergence or barriers



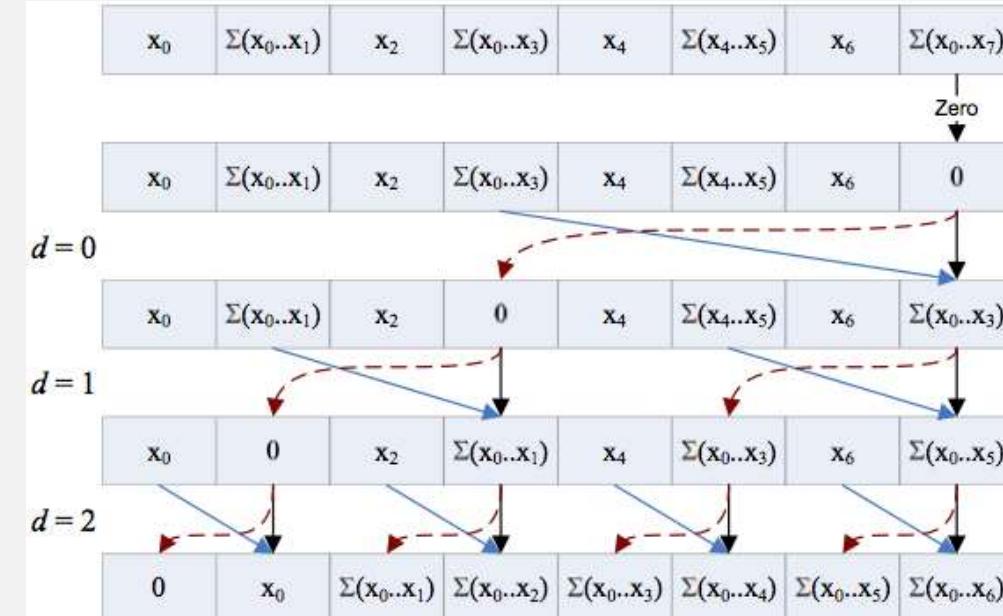
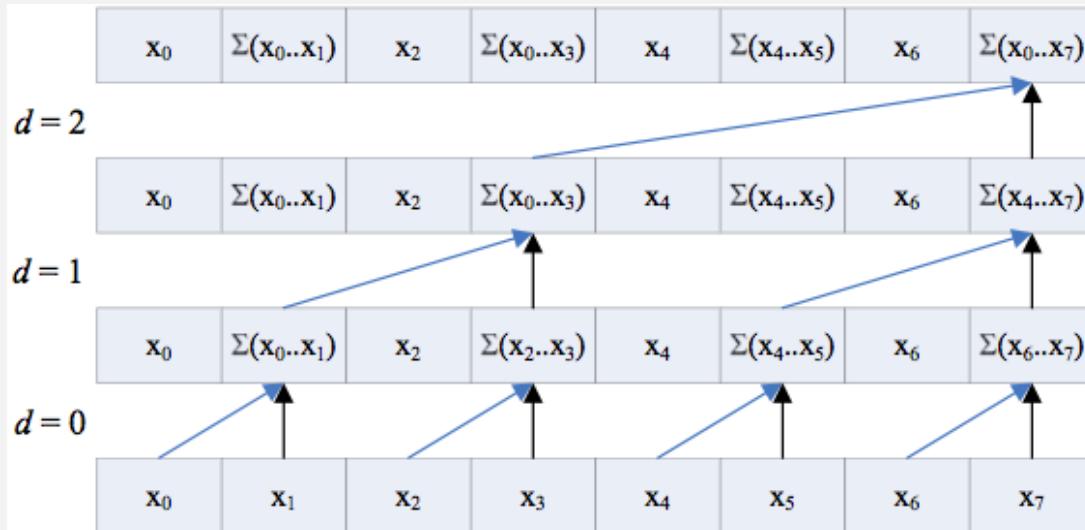
- What if we truly had a SIMD machine?
- Recall CUDA warps (32 threads) are strictly SIMD
- “Warp-synchronous”

# Brent Kung Scan

Circuit family

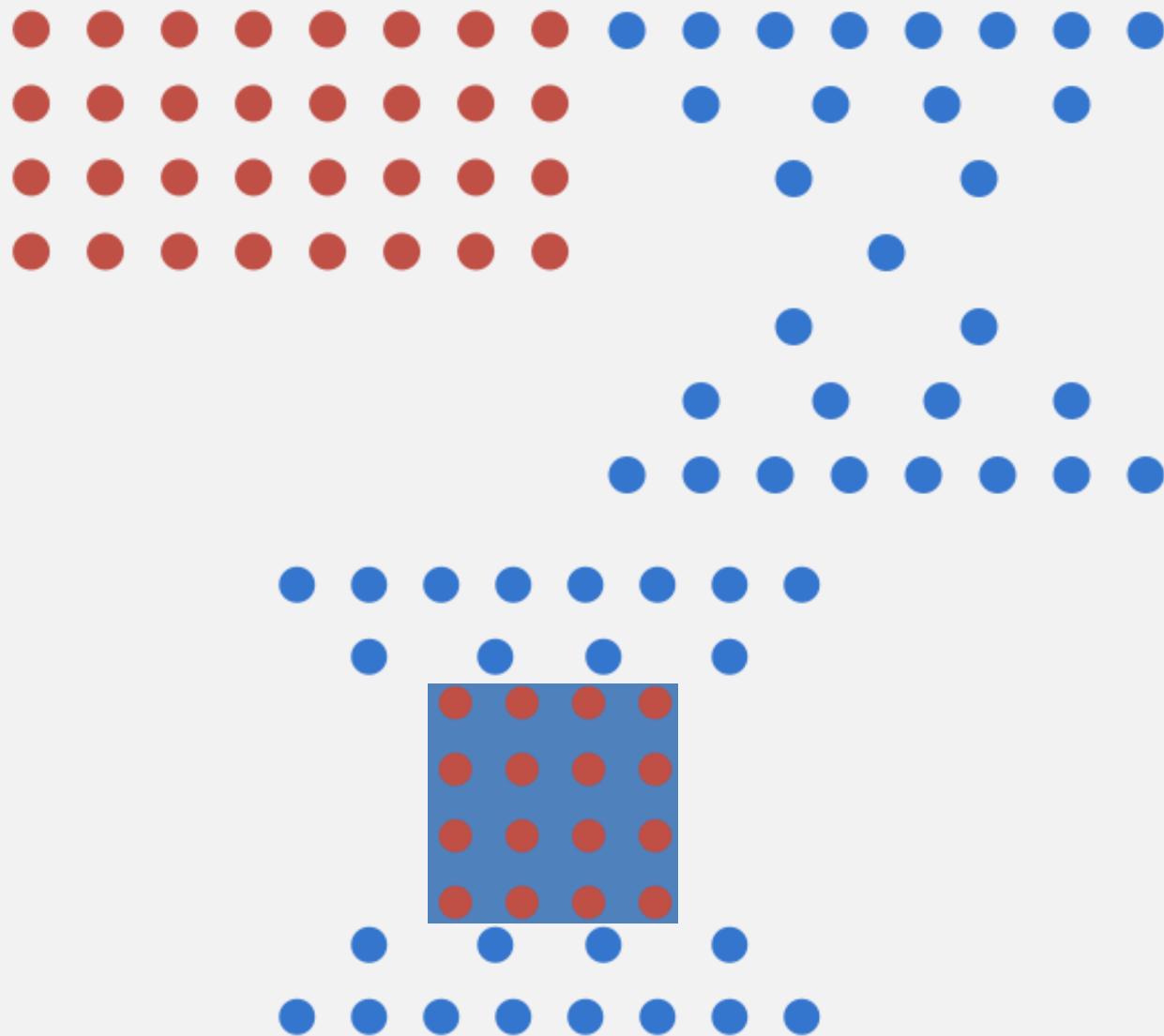


# $O(n)$ Scan [Blelloch]



- Not step efficient ( $2 \log n$  steps)
- Work efficient ( $O(n)$  work)
- Bank conflicts, and lots of 'em

# Hybrid methods

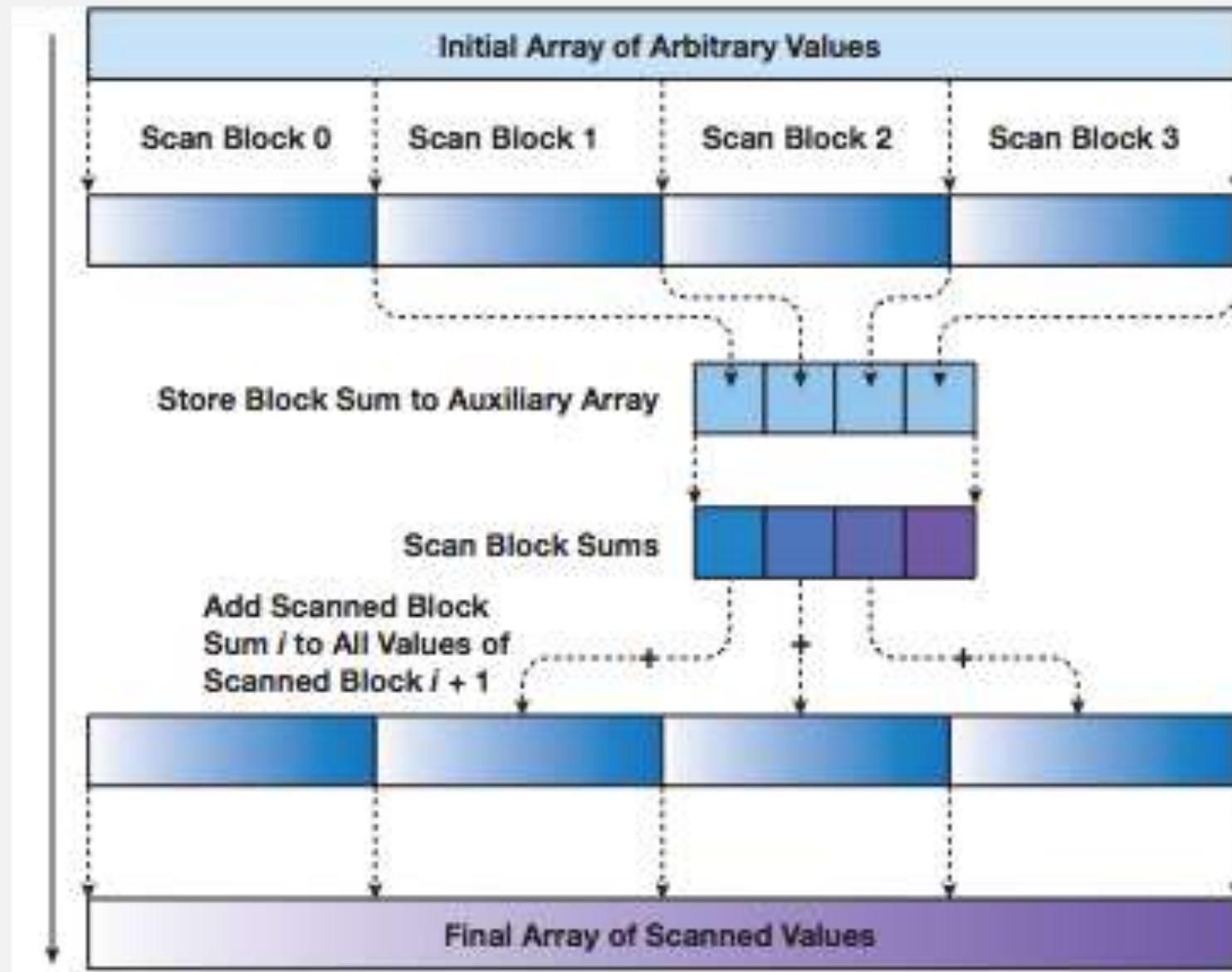


# Scan papers

- Daniel Horn, Stream Reduction Operations for GPGPU Applications, GPU Gems 2, Chapter 36, pp. 573–589, March 2005.
- Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A Work-Efficient Step-Efficient Prefix Sum Algorithm. In Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures, pages D–26–27, May 2006
- Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007.
- Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In Graphics Hardware 2007, pages 97–106, August 2007.
- Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli, “Fast scan algorithms on graphics processors,” in ICS ’08: Proceedings of the 22nd Annual International Conference on Supercomputing, 2008, pp. 205–213.
- Shubhabrata Sengupta, Mark Harris, Michael Garland, and John D. Owens. Efficient Parallel Scan Algorithms for many-core GPUs. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, Scientific Computing with Multicore and Accelerators, Chapman & Hall/CRC Computational Science, chapter 19, pages 413–442. Taylor & Francis, January 2011.
- D. Merrill and A. Grimshaw, Parallel Scan for Stream Architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, 2009, 54pp.

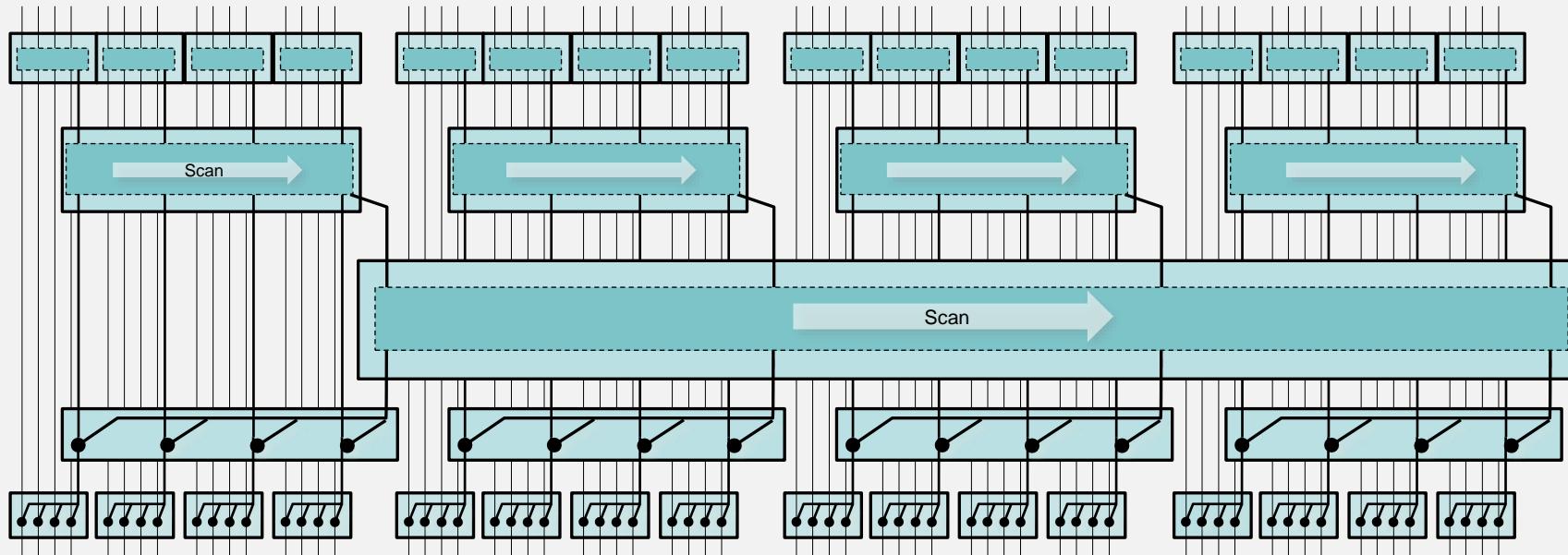
Scan (across blocks)

# Scan-then-propagate (4x)



# Scan-then-propagate (4x)

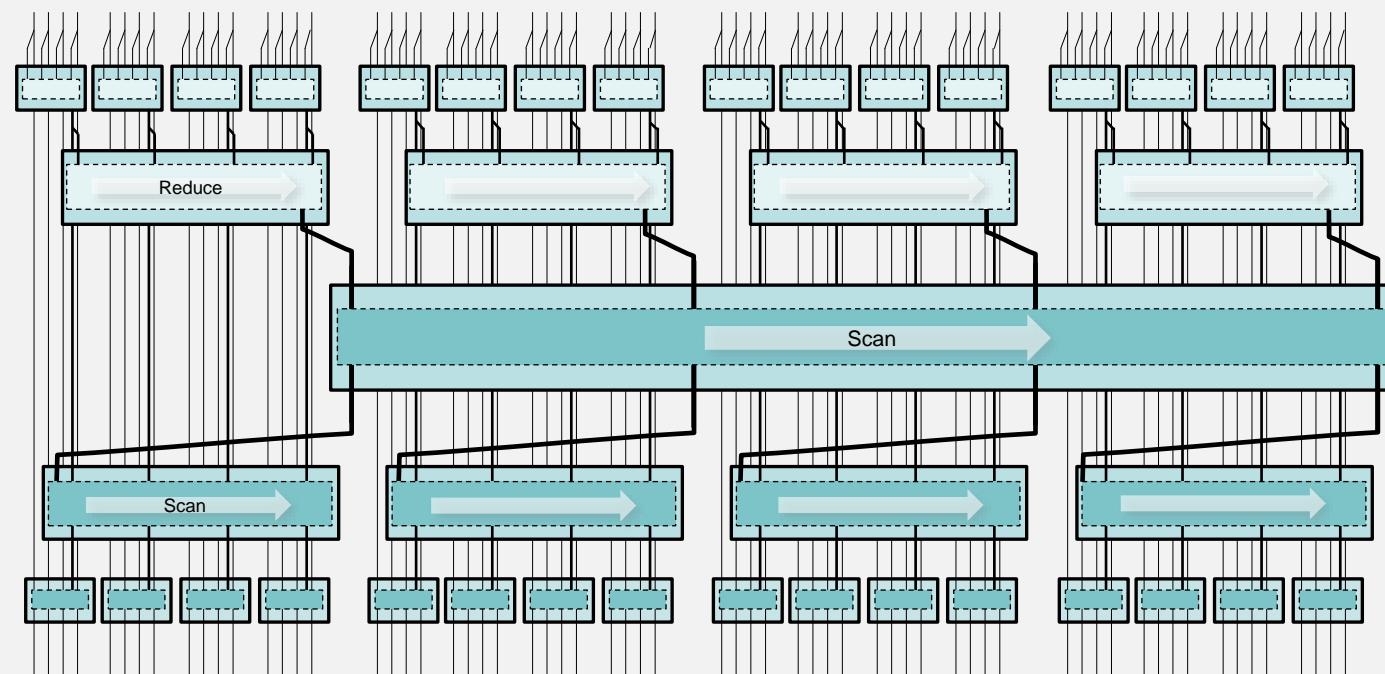
$\log_b(N)$  –level upsweep/downsweep  
(scan-and-add strategy: CudPP)



# Reduce-then-scan (3x)

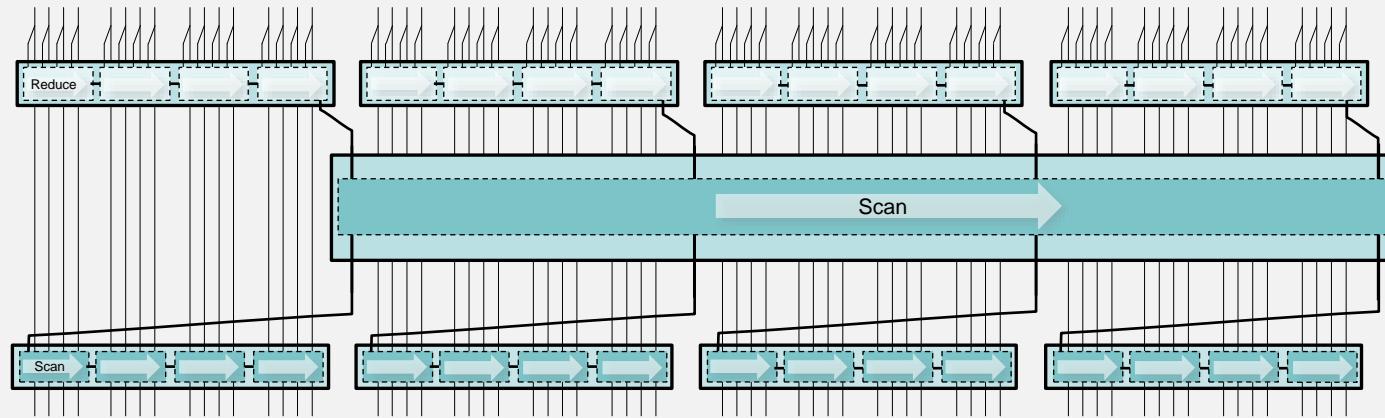
$\log_b(N)$  –level upsweep/downsweep

(reduce-then-scan strategy: Matrix-scan)



# Merrill's 2-level upsweep/downsweep

(reduce-then-scan)



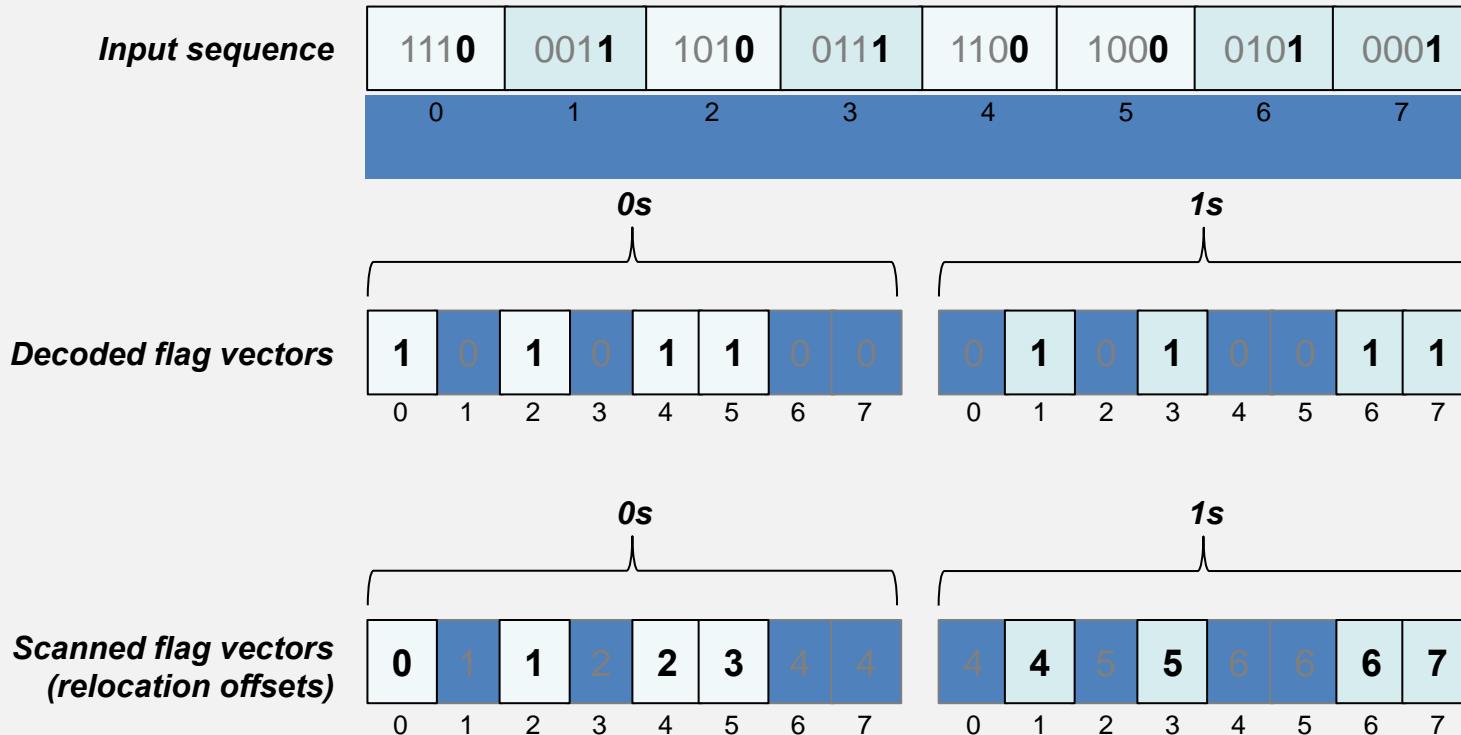
- Persistent threads
- Requires only 3 kernel launches vs.  $\log n$
- Fewer global memory reads in intermediate step  
(constant vs.  $O(n)$ )

# Sort papers

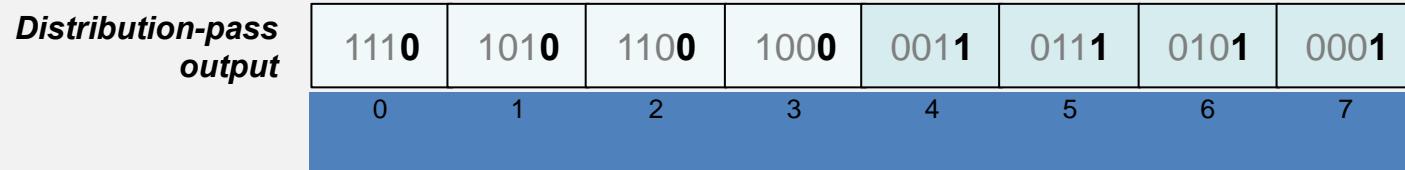
- Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007.
- N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore GPUs,” IPDPS 2009: IEEE International Symposium on Parallel & Distributed Processing, May 2009.
- D. Merrill and A. Grimshaw, Revisiting Sorting for GPGPU Stream Architectures. Technical Report CS2010-03, Department of Computer Science, University of Virginia, 2010, 17pp.

# Radix Sort

# Radix Sort Fundamentals



scatter—not efficient!



Goals: (1) minimize number of scatters; (2)  
maximize coherence of scatters

# Radix Sort Memory Cost

Step	Kernel	Purpose	Read Workload	Write Workload
1	<i>binning</i>	Create flags	$n$ keys	$nr$ flags
2	<i>bottom-level reduce</i>	Compact flags (scan primitive)	$nr$ flags	<i>(insignificant constant)</i>
3	<i>top-level scan</i>		<i>(insignificant constant)</i>	<i>(insignificant constant)</i>
4	<i>bottom-level scan</i>		$nr$ flags + <i>(insignificant constant)</i>	$nr$ offsets
5	<i>scatter</i>	Distribute keys	$n$ offsets + $n$ keys (+ $n$ values)	$n$ keys (+ $n$ values)

Total Memory Workload:  $(k/d)(n)(r + 4)$  keys only  
 $(k/d)(n)(r + 6)$  with values

- d-bit radix digits
- radix  $r = 2^d$
- n-element input sequence of k-bit keys

# Radix Sort

- Apply counting sort to successive digits of keys
- Performs  $d$  scatter steps for  $d$ -digit keys
- Scattering in memory is fundamentally costly

# Parallel Radix Sort

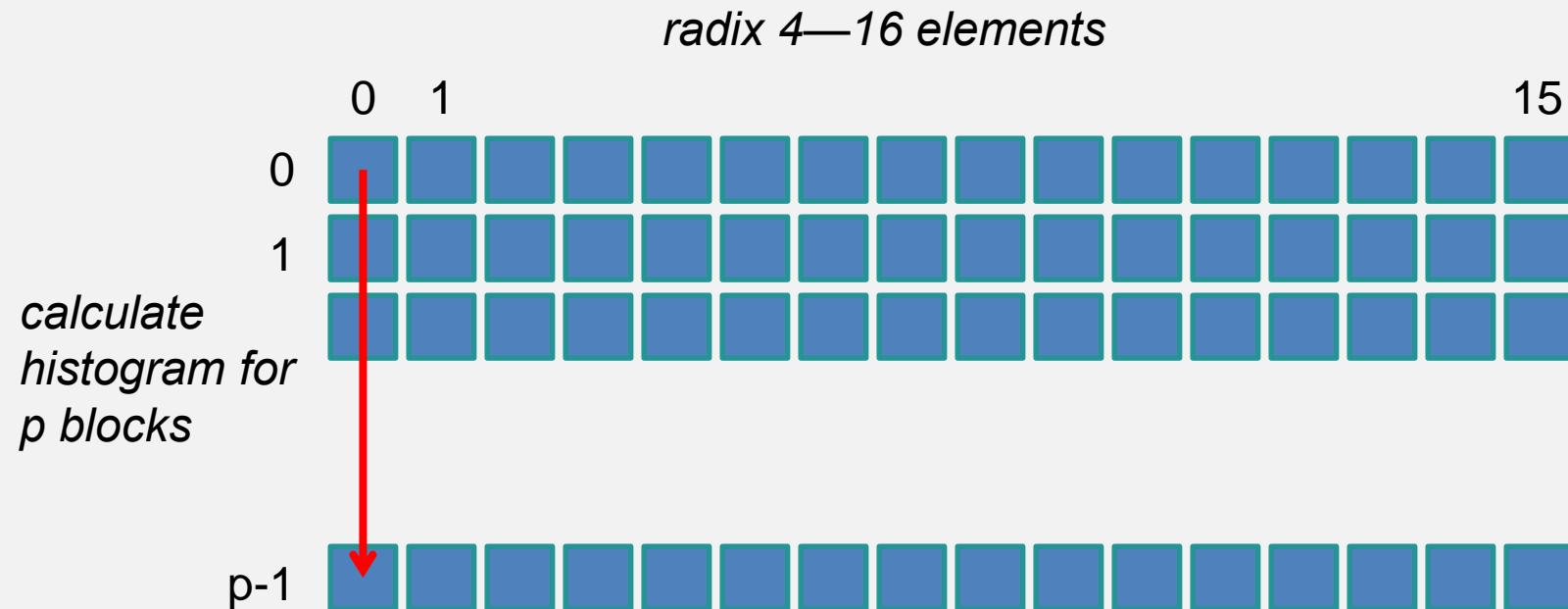
- Assign tile of data to each block (1024 elements)  
*Satisf uses 256-thread blocks and 4 elements per thread*
- Build per-block histograms of current digit (4 bit)  
*this is a reduction*
- Combine per-block histograms (P x 16)  
*this is a scan*
- Scatter

# Per-Block Histograms

- Perform  $b$  parallel splits for  $b$ -bit digit
- Each split is just a prefix sum of bits
  - each thread counts 1 bits to its left
- Write bucket counts & partially sorted tile
  - sorting tile improves scatter coherence later

# Combining Histograms

- Write per-block counts in column major order & scan



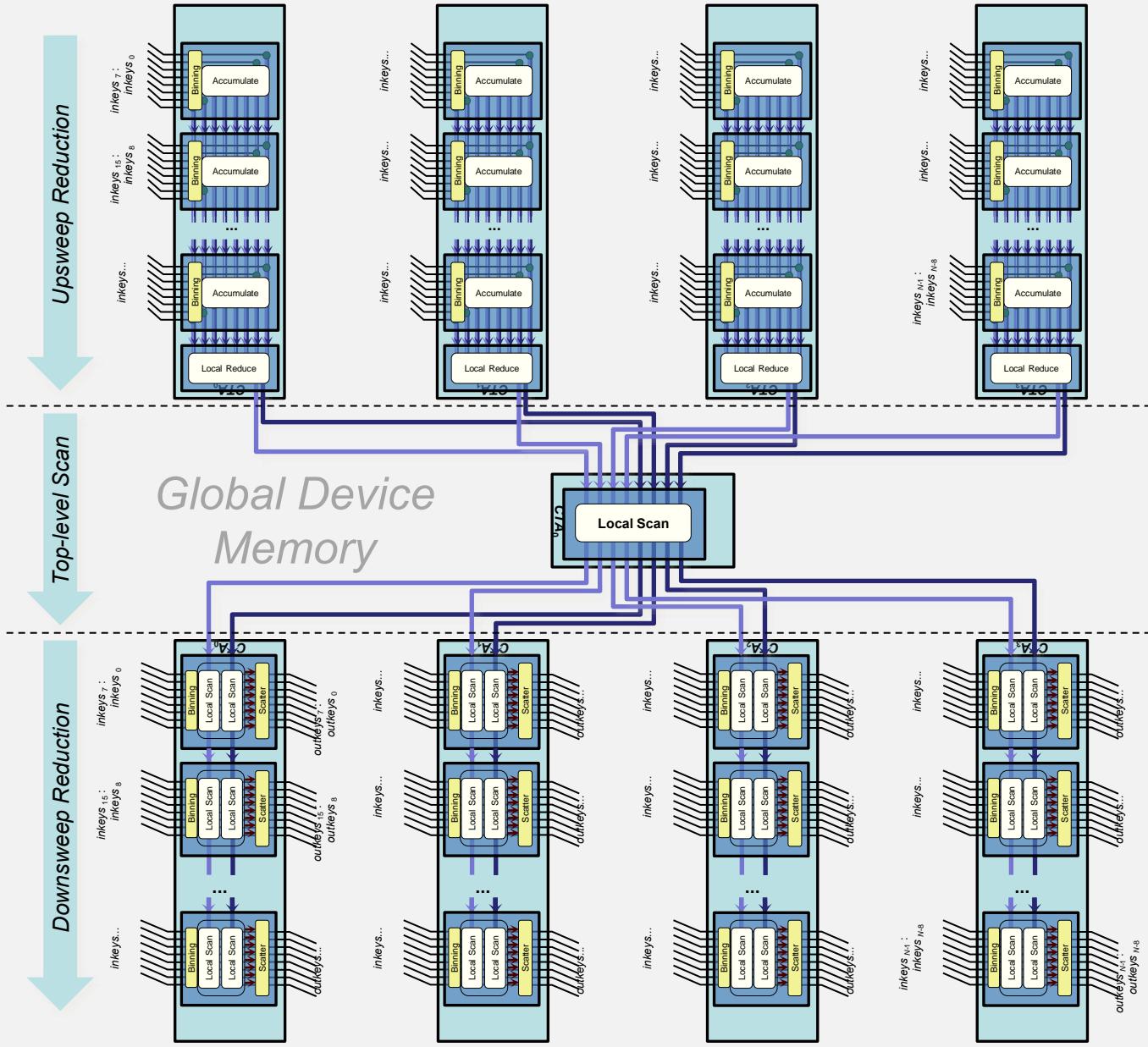
# Satish's Radix Sort Memory Cost

Step	Kernel	Purpose	Read Workload	Write Workload
1	<i>local digit-sort</i>	Maximize coherence	$n$ keys (+ $n$ values)	$n$ keys (+ $n$ values)
2	<i>histogram</i>	Create histograms	$n$ keys	$nr/b$ counts
3	<i>bottom-level reduce</i>	Scan histograms (scan primitive)	$nr/b$ counts	<i>(insignificant constant)</i>
4	<i>top-level scan</i>		<i>(insignificant constant)</i>	<i>(insignificant constant)</i>
5	<i>bottom-level scan</i>		$nr/b$ counts + <i>(insignificant constant)</i>	$nr/b$ offsets
6	<i>scatter</i>	Distribute keys	$nr/b$ offsets + $n$ keys (+ $n$ values)	$n$ keys (+ $n$ values)

**Total Memory Workload:**  $(k/d)(n)(5r/b + 7)$  keys only  
 $(k/d)(n)(5r/b + 9)$  with values

- d-bit radix digits
- radix  $r = 2^d$
- n-element input sequence of k-bit keys
- b bits per step

# Merrill's 3-step sort



# Merrill's sort, costs

Step	Kernel	Purpose	Read Workload	Write Workload
1	<i>bottom-level reduce</i>	Create flags, compact flags, scatter keys	$n$ keys <i>(insignificant constant)</i>	<i>(insignificant constant)</i>
2	<i>top-level scan</i>		<i>(insignificant constant)</i>	<i>(insignificant constant)</i>
3	<i>bottom-level scan</i>		$n$ keys (+ $n$ values) + <i>(insignificant constant)</i>	$n$ keys (+ $n$ values)

Total Memory Workload:  $(k/d)(3n)$  keys only  
 $(k/d)(5n)$  with values

- d-bit radix digits
- radix  $r = 2^d$
- n-element input sequence of k-bit keys
- Current GPUs use  $d=4$  (higher values exhaust local storage)

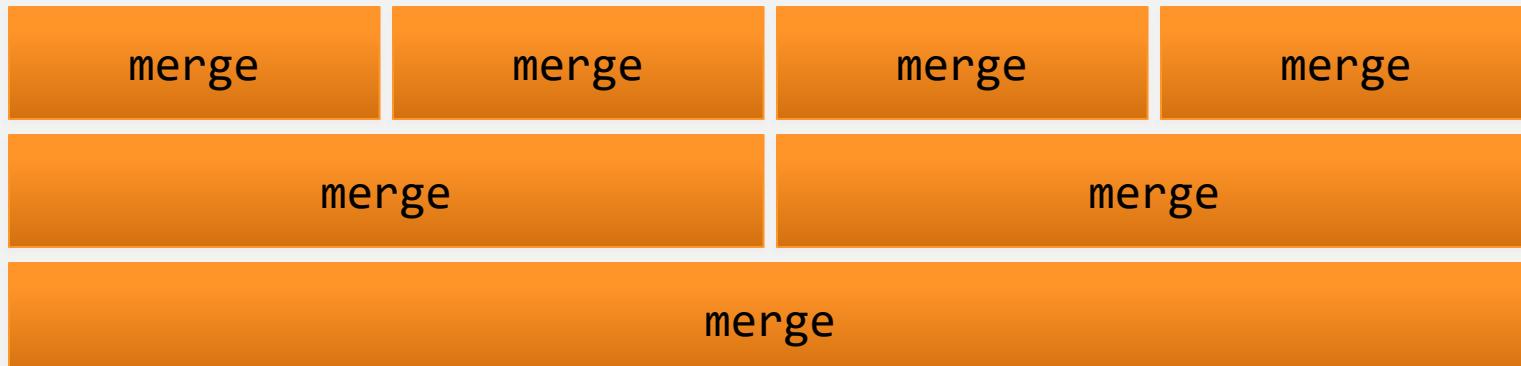
# Merge Sort

# Merge Sort

- Divide input array into 256-element tiles
- Sort each tile independently



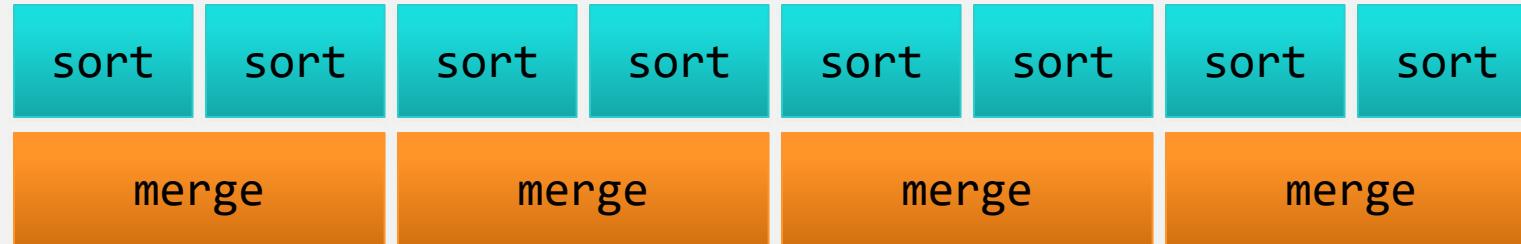
- Produce sorted output with tree of merges



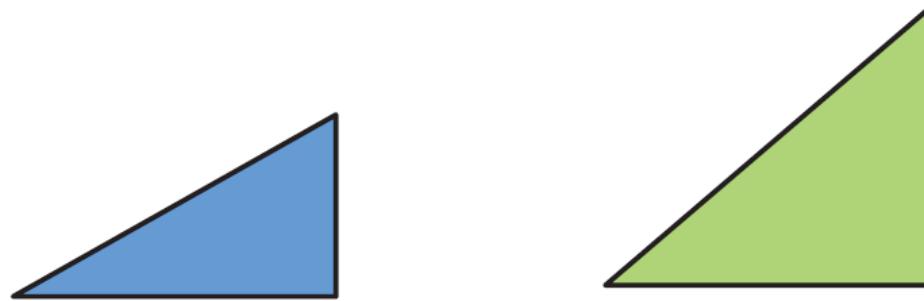
# Sorting a Tile

- Tiles are sized so that:
  - a single thread block can sort them efficiently
  - they fit comfortably in on-chip memory
- Sorting networks are most efficient in this regime
  - we use **odd-even merge sort**
  - about 5-10% faster than comparable bitonic sort
- Caveat: sorting networks may reorder equal keys

# Merging Pairs of Sorted Tiles



- Launch 1 thread block to process each pair of tiles
- Load tiles into on-chip memory
- Perform **counting merge**
- Store merged result to global memory

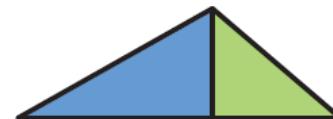


Sorted  
Input A

Sorted  
Input B



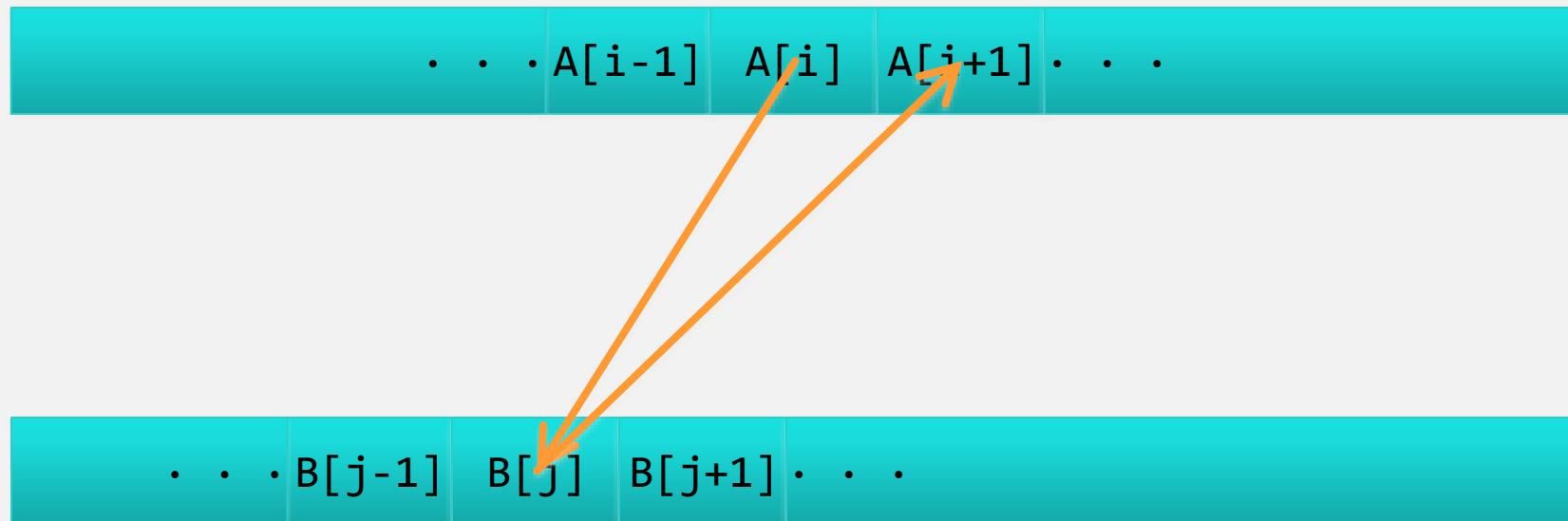
Flip B, pairwise compare to A



Smallest element in each comparison  
yields smallest p elements overall in a  
bitonic sequence

# Counting Merge

`upper_bound(A[i], B) = count( j where A[i] ≤ B[j] )`



`lower_bound(B[j], A) = count( i where B[j] < A[i] )`

**Use binary search since A & B are sorted**

# Counting Merge

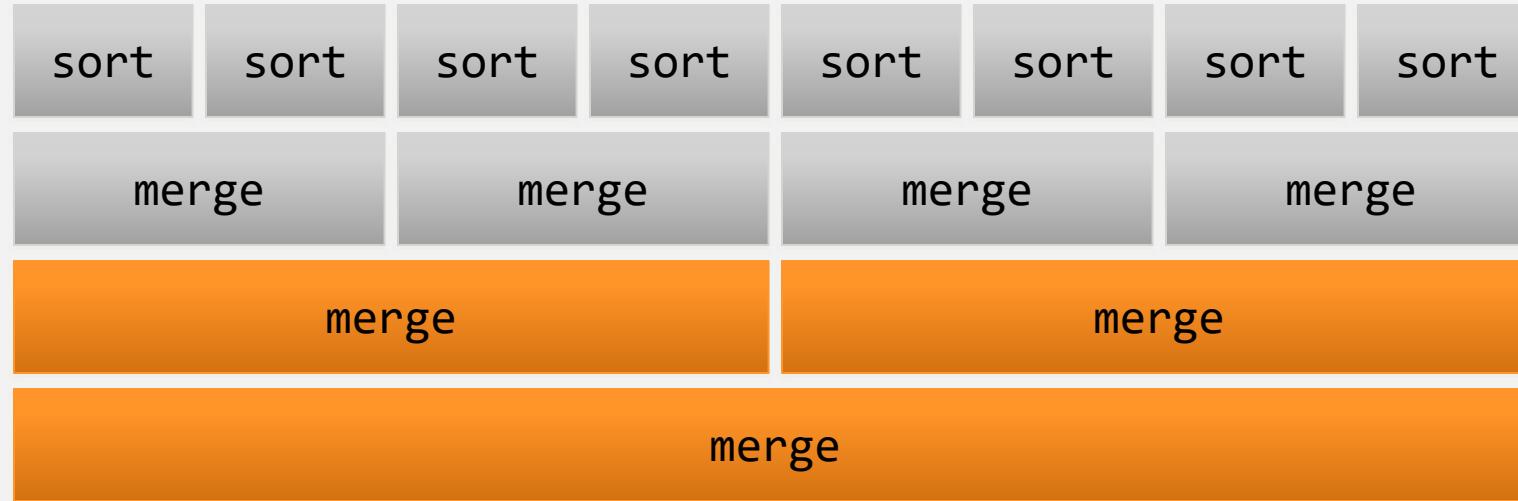
`upper_bound(A[i], B) = count( j where A[i] ≤ B[j] )`



`lower_bound(B[j], A) = count( i where B[j] < A[i] )`

```
scatter( A[i] -> C[i + upper_bound(A[i], B)] )
scatter( B[j] -> C[lower_bound(B[j], A) + j] )
```

# Merging Larger Subsequences



- Partition larger sequences into collections of tiles
- Apply counting merge to each pair of tiles

# Two-way Partitioning Merge

- Pick a splitting element from either A or B

... A[i] ...

- Divide A and B into elements below/above splitter

A[j] ≤ A[i] A[i] A[j] > A[i]

B[j] ≤ A[i] B[j] > A[i]



found by binary search

merge : A[j] ≤ A[i] A[i]  
B[j] ≤ A[i]

merge : A[j] > A[i]  
B[j] > A[i]

# Multi-way Partitioning Merge

- Pick every 256<sup>th</sup> element of A & B as splitter

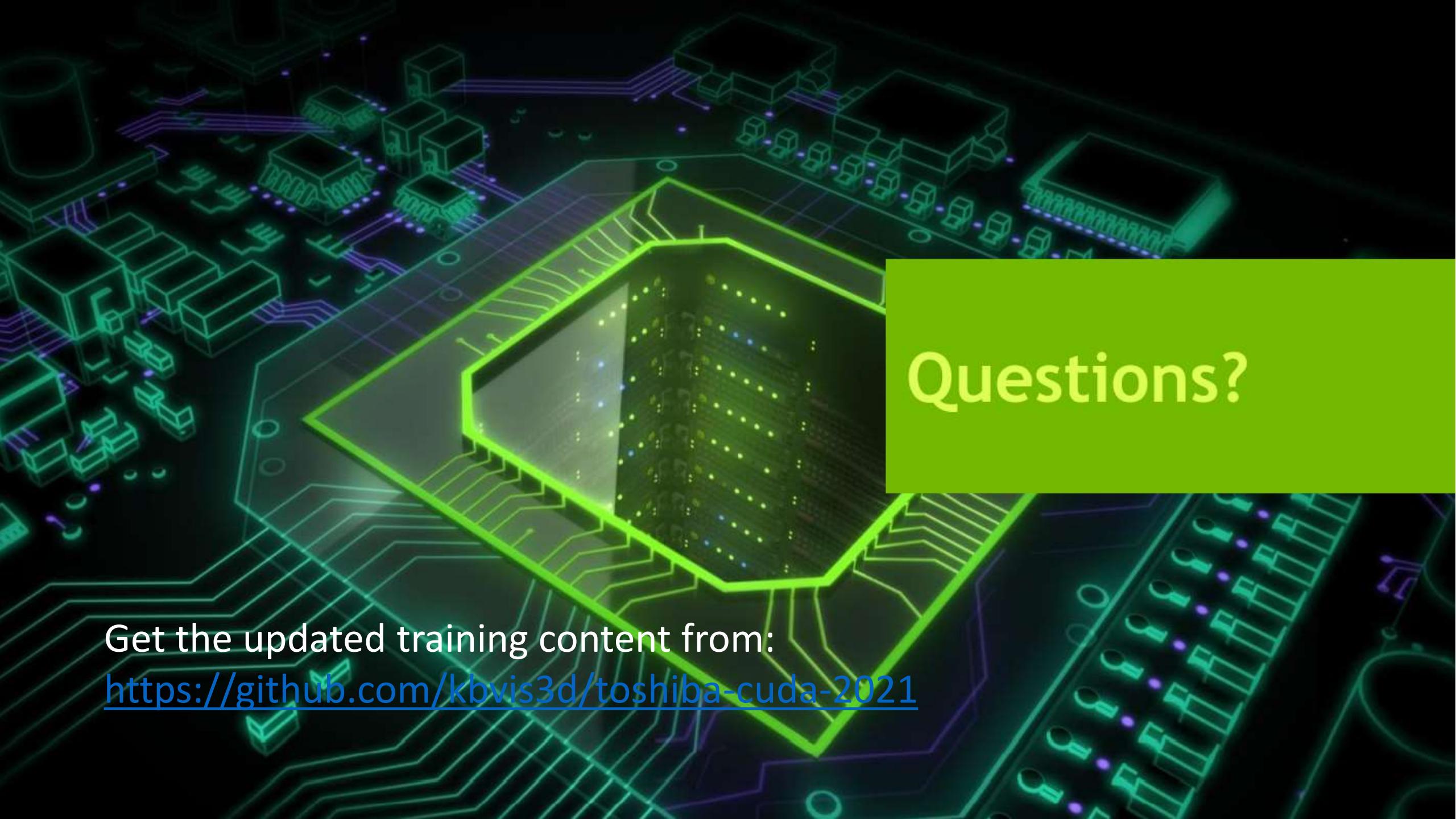


- Apply merge recursively to merge splitter sets
  - recursively apply merge procedure

- Split A & B with merged splitters



- Merge resulting pairs of tiles (at most 256 elements)



# Questions?

Get the updated training content from:

<https://github.com/kbvis3d/toshiba-cuda-2021>

# Image Processing Kernels



# introduction

- Image processing is a natural fit for data parallel processing
  - Pixels can be mapped directly to threads
  - Lots of data is shared between pixels
- Advantages of CUDA vs. pixel shader-based image processing
- CUDA supports sharing image data with OpenGL and Direct3D applications



# advantages of CUDA

- Shared memory (high speed on-chip cache)
- More flexible programming model
  - C with extensions vs HLSL/GLSL
- Arbitrary scatter writes
- Each thread can write more than one pixel
- Thread Synchronization



# applications

- Convolutions
- Median filter
- FFT
- Image & Video compression
- DCT
- Wavelet
- Motion Estimation
- Histograms
- Noise reduction
- Image correlation
- Demosaic of CCD images (RAW conversion)



# shared memory

- Shared memory is fast
  - Same speed as registers
  - Like a user managed data cache
- Limitations
  - 64KB per multiprocessor
  - Can store 128 x 128 pixels with 4 bytes per pixel
- Typical operation for each thread block:
  - Load image tile from global memory to shared
  - Synchronize threads
  - Threads operate on pixels in shared memory in parallel
  - Write tile back from shared to global memory
- Global memory vs Shared
  - Big potential for significant speed up depending on how many times data in shared memory can be reused

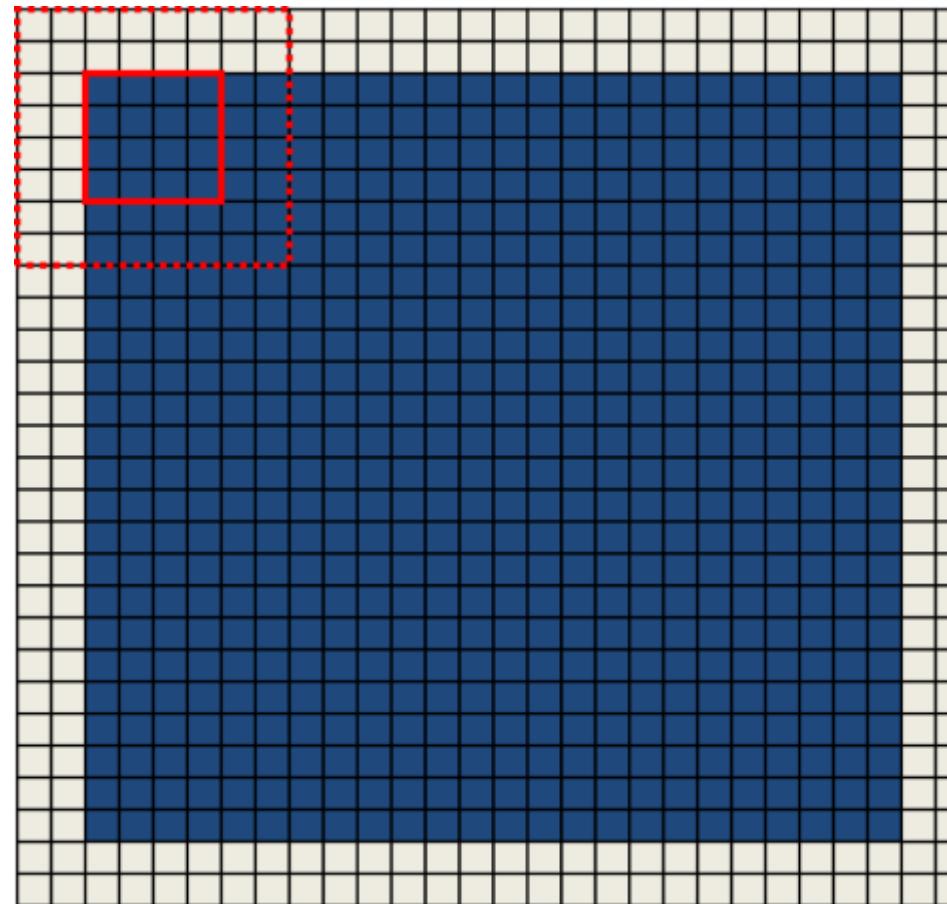
© 2008 NVIDIA Corporation.



# separable convolutions

- Filter coefficients can be stored in constant memory
- Image tile can be cached to shared memory
- Each output pixel must have access to neighboring pixels within certain radius  $R$
- This means tiles in shared memory must be expanded with an apron that contains neighboring pixels
- Only pixels within the apron write results
  - The remaining threads do nothing

# tile apron



- Image
- Image Apron
- Tile
- Tile with Apron

© 2008 NVIDIA Corporation.



# image processing with CUDA

- How does image processing map to the GPU?
  - Image Tiles      ↔      Grid/Thread Blocks
  - Large Data      ↔      Lots of Memory BW
  - 2D Region      ↔      Shared Memory (cached)

# define tile sizes

```
#define TILE_W    16
#define TILE_H    16
#define R          2      // filter radius
#define D          (R*2+1) // filter diameter
#define S          (D*D)  // filter size
#define BLOCK_W   (TILE_W+(2*R))
#define BLOCK_H   (TILE_H+(2*R))
```

# simple filter example

```
__global__ void d_filter(int *g_idata, int *g_odata,
                        unsigned int width, unsigned int height)
{
    __shared__ int smem[BLOCK_W*BLOCK_H];
    int x = blockIdx.x*TILE_W + threadIdx.x - R;
    int y = blockIdx.y*TILE_H + threadIdx.y - R;

    // clamp to edge of image
    x = max(0, x);
    x = min(x, width-1);
    y = max(y, 0);
    y = min(y, height-1);

    unsigned int index = y*width + x;
    unsigned int bindex = threadIdx.y*blockDim.y+threadIdx.x;

    // each thread copies its pixel of the block to shared memory
    smem[bindex] = g_idata[index];
    __syncthreads();
}
```

© 2008 NVIDIA Corporation.

# simple filter example (cont.)

```
// only threads inside the apron will write results
if ((threadIdx.x >= R) && (threadIdx.x < (BLOCK_W-R)) &&
    (threadIdx.y >= R) && (threadIdx.y < (BLOCK_H-R)))
{
    float sum = 0;
    for(int dy=-R; dy<=R; dy++) {
        for(int dx=-R; dx<=R; dx++) {
            float i = smem[bindex + (dy*blockDim.x) + dx];
            sum += i;
        }
    }
    g_odata[index] = sum / s;
}
```

# sobel edge detect filter

- Two filters to detect horizontal and vertical change in the image
- Computes the magnitude and direction of edges
- We can calculate both directions with one single CUDA kernel



© 2008 NVIDIA Corporation.

$$C_{horizontal} = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

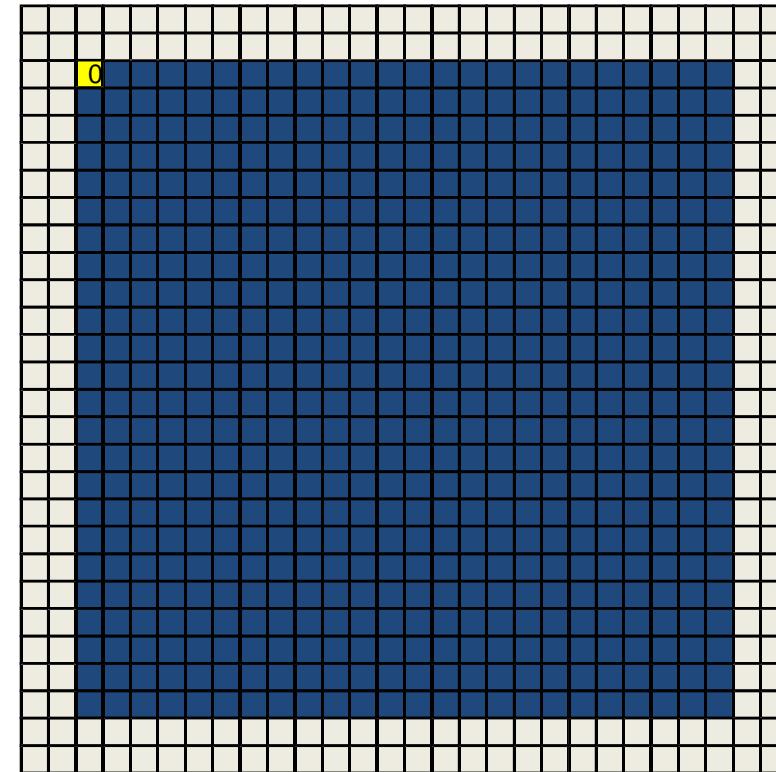
$$C_{vertical} = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

$$Magnitude_{Sobel} = norm \cdot \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

$$Direction_{Sobel} = \arctan\left(\frac{G_{vertical}}{G_{horizontal}}\right)$$

# sobel edge detect filter

- 3x3 window of pixels for each thread



$$\bullet \quad C_{vertical} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

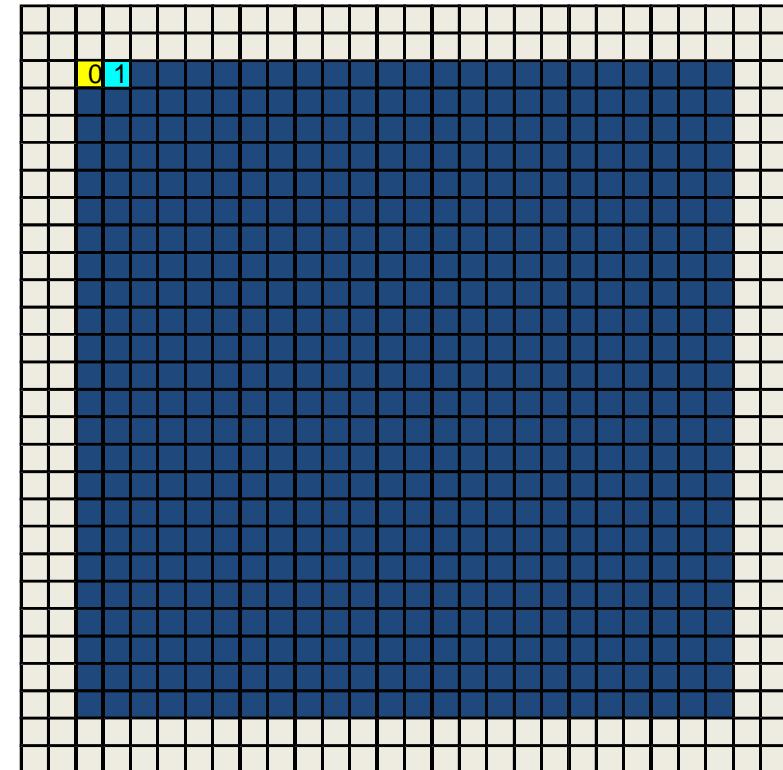
$$\bullet \quad C_{horizontal} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizontal}$$

$$Magnitude_{Sobel} = norm \cdot \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

© 2008 NVIDIA Corporation.

# sobel edge detect filter

- 3x3 window of pixels for each thread



$$\bullet \quad C_{vertical} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

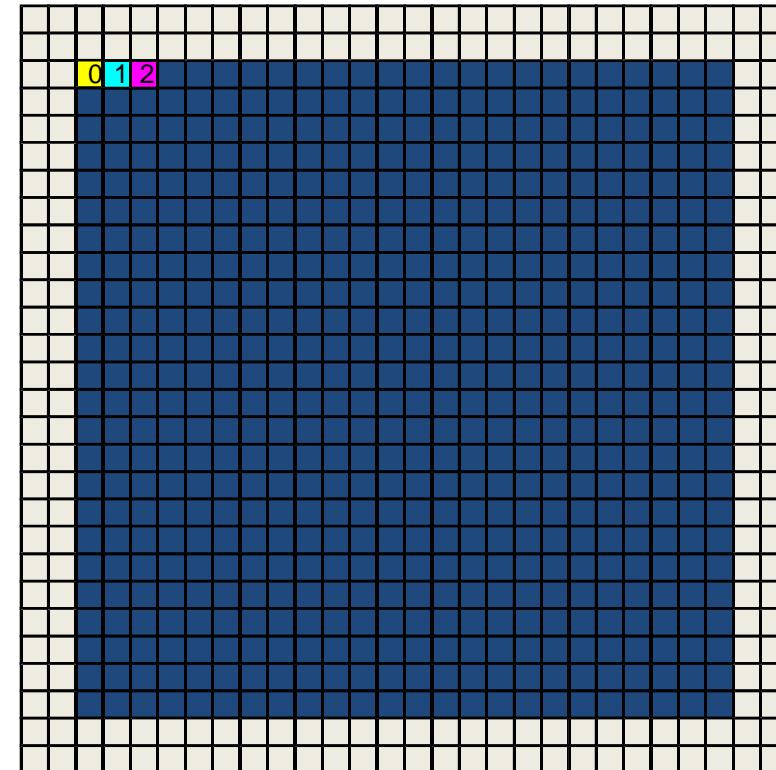
$$\bullet \quad C_{horizontal} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizontal}$$

$$Magnitude_{Sobel} = norm \cdot \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

© 2008 NVIDIA Corporation.

# sobel edge detect filter

- 3x3 window of pixels for each thread



$$\bullet \quad C_{vertical} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

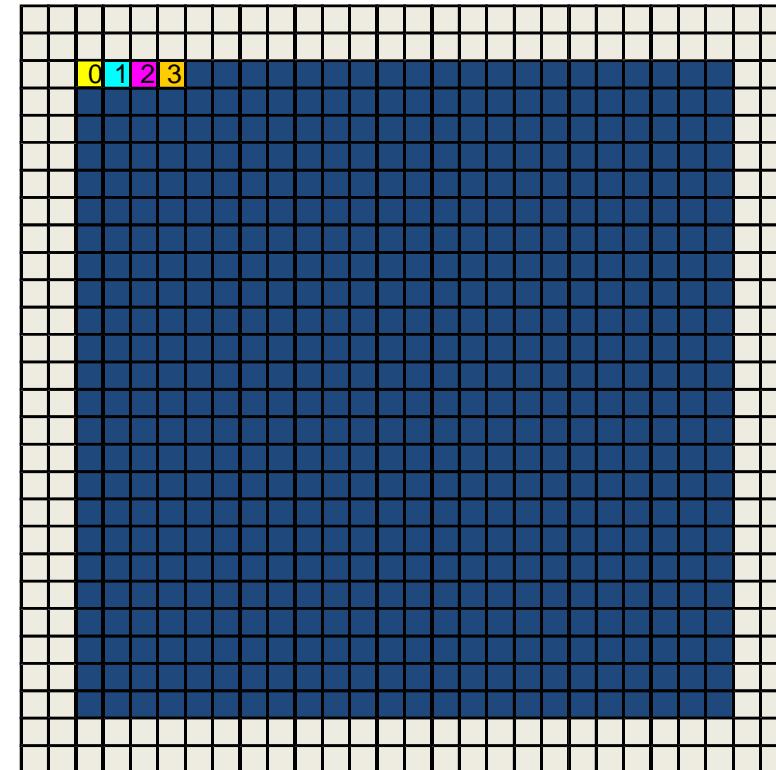
$$\bullet \quad C_{horizontal} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizontal}$$

$$Magnitude_{Sobel} = norm \cdot \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

© 2008 NVIDIA Corporation.

# sobel edge detect filter

- 3x3 window of pixels for each thread



$$\bullet \quad C_{vertical} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

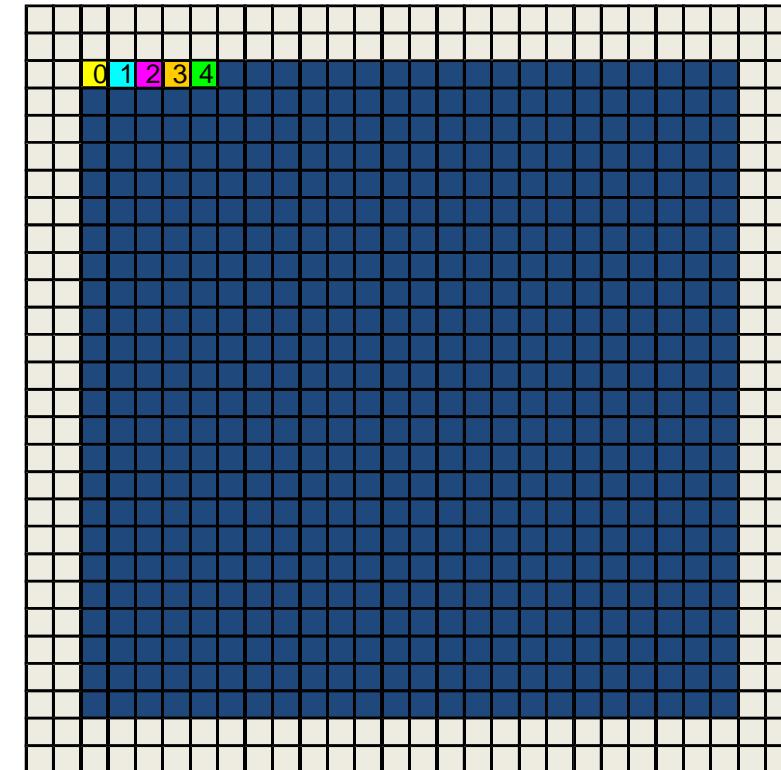
$$\bullet \quad C_{horizontal} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizontal}$$

$$Magnitude_{Sobel} = norm \cdot \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

© 2008 NVIDIA Corporation.

# sobel edge detect filter

- 3x3 window of pixels for each thread



$$\bullet \quad C_{vertical} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

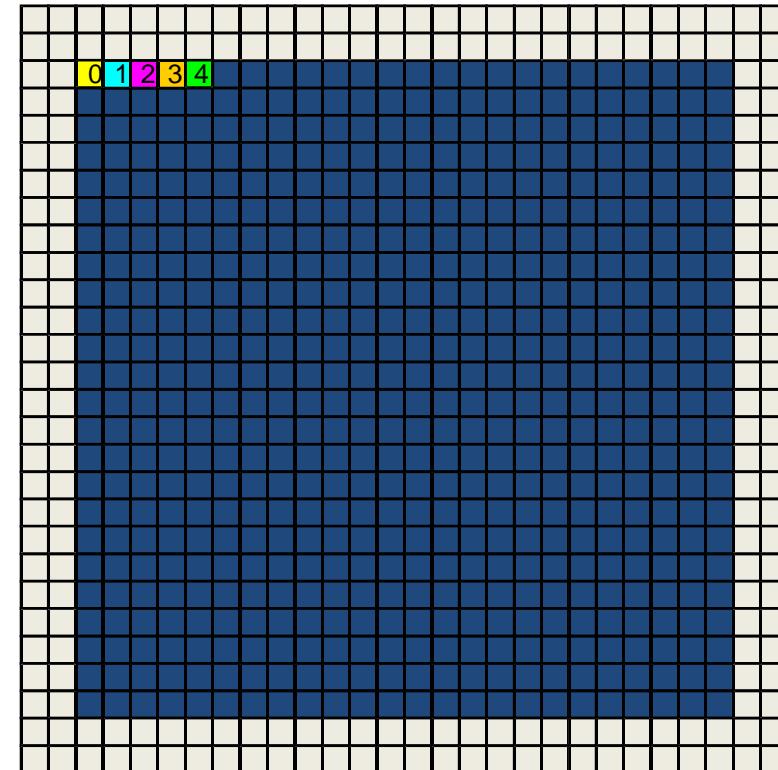
$$\bullet \quad C_{horizontal} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizontal}$$

$$Magnitude_{Sobel} = norm \cdot \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

© 2008 NVIDIA Corporation.

# sobel edge detect filter

- 3x3 window of pixels for each thread



$$\begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$$

$$\bullet \quad C_{vertical} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

$$\begin{bmatrix} 4 & 4 & 4 \\ 4 & 4 & 4 \\ 4 & 4 & 4 \end{bmatrix}$$

$$\bullet \quad C_{horizontal} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizontal}$$

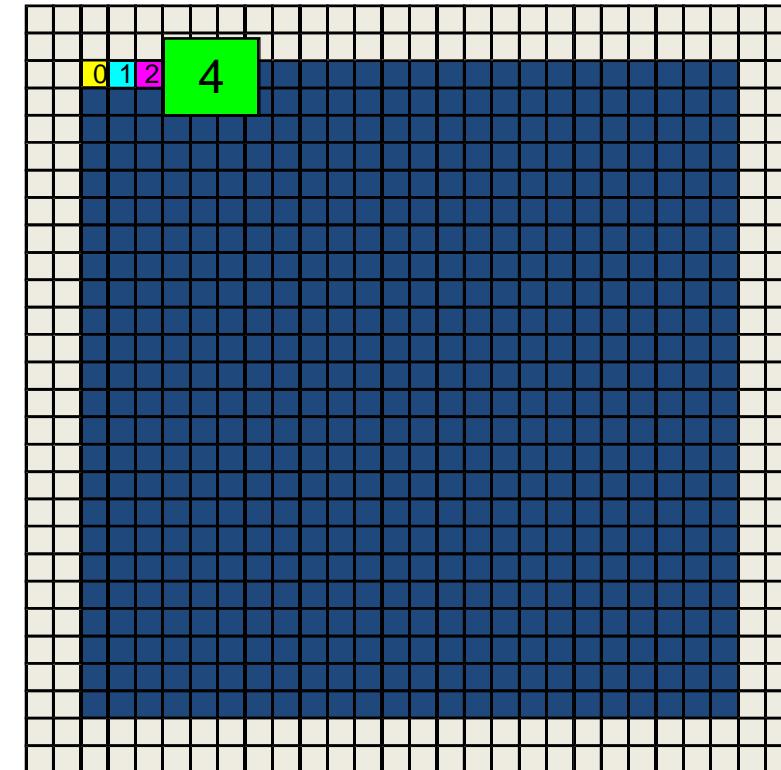
$$Magnitude_{Sobel} = norm \bullet \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$



© 2008 NVIDIA Corporation.

# sobel edge detect filter

- 3x3 window of pixels for each thread



$$\bullet \quad C_{vertical} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

$$\bullet \quad C_{horizontal} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizontal}$$

$$Magnitude_{Sobel} = norm \cdot \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

© 2008 NVIDIA Corporation.

# sobel edge detect filter

- 3x3 window of pixels for each thread



$$\bullet \quad C_{vertical} \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = G_{vertical}$$

$$\bullet \quad C_{horizontal} \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = G_{horizontal}$$

$$Magnitude_{Sobel} = norm \cdot \sqrt{G_{horizontal}^2 + G_{vertical}^2}$$

© 2008 NVIDIA Corporation.

# samples / SobelFilter

```
_device_ unsigned char
ComputeSobel(unsigned char ul, // upper left
            unsigned char um, // upper middle
            unsigned char ur, // upper right
            unsigned char ml, // middle left
            unsigned char mm, // middle (unused)
            unsigned char mr, // middle right
            unsigned char ll, // lower left
            unsigned char lm, // lower middle
            unsigned char lr, // lower right
            float fScale)
{
    short Horz = ur + 2*mr + lr - ul - 2*ml - ll;
    short Vert = ul + 2*um + ur - ll - 2*lm - lr;
    short Sum = (short)(fScale*(abs((int)Horz)+abs((int)Vert)));

    if (Sum < 0)
    {
        return 0;
    }
    else if (Sum > 0xff)
    {
        return 0xff;
    }

    return (unsigned char) Sum;
}
```

# samples / SobelFilter

```
_global_ void
SobelTex(Pixel *pSobelOriginal, unsigned int Pitch,
          int w, int h, float fScale, cudaTextureObject_t tex)
{
    unsigned char *pSobel =
        (unsigned char *)(((char *) pSobelOriginal)+blockIdx.x*Pitch);

    for (int i = threadIdx.x; i < w; i += blockDim.x)
    {
        unsigned char pix00 = tex2D<unsigned char>(tex, (float) i-1, (float) blockIdx.x-1);
        unsigned char pix01 = tex2D<unsigned char>(tex, (float) i+0, (float) blockIdx.x-1);
        unsigned char pix02 = tex2D<unsigned char>(tex, (float) i+1, (float) blockIdx.x-1);
        unsigned char pix10 = tex2D<unsigned char>(tex, (float) i-1, (float) blockIdx.x+0);
        unsigned char pix11 = tex2D<unsigned char>(tex, (float) i+0, (float) blockIdx.x+0);
        unsigned char pix12 = tex2D<unsigned char>(tex, (float) i+1, (float) blockIdx.x+0);
        unsigned char pix20 = tex2D<unsigned char>(tex, (float) i-1, (float) blockIdx.x+1);
        unsigned char pix21 = tex2D<unsigned char>(tex, (float) i+0, (float) blockIdx.x+1);
        unsigned char pix22 = tex2D<unsigned char>(tex, (float) i+1, (float) blockIdx.x+1);
        pSobel[i] = ComputeSobel(pix00, pix01, pix02,
                                pix10, pix11, pix12,
                                pix20, pix21, pix22, fScale);
    }
}
```

# Textures and Surfaces



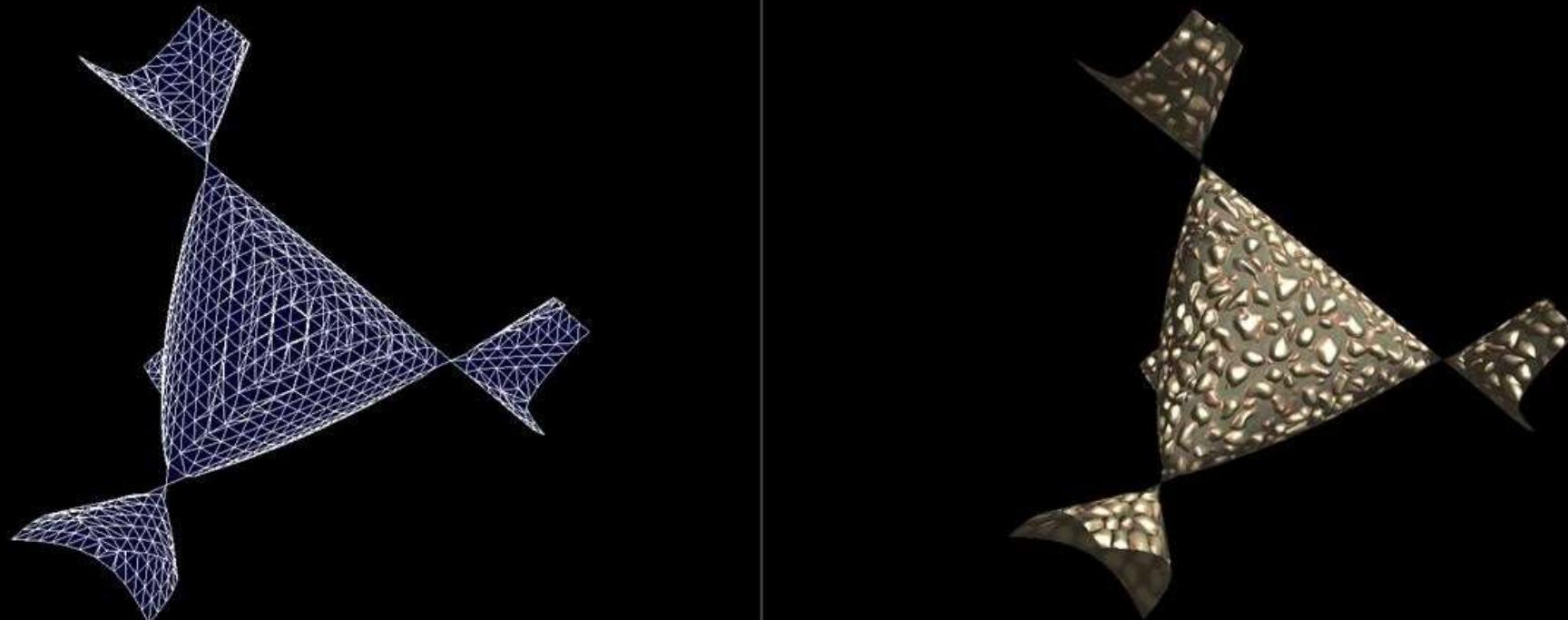
- **Intro to Texturing and Texture Unit**
- **CUDA Array Storage**
- **Textures in CUDA C (Setup, Binding Modes, Coordinates)**
- **Texture Data Processing**
- **Texture Interpolation**
- **Surfaces**
- **Layered Textures (CUDA 4.0 Features)**
- **Usage Advice**
- **Misc: 16 bit-floating point textures, OpenGL/DirectX Exchange**
- **Summary, Further Reading and Questions**

# Texturing



- **Original purpose:**

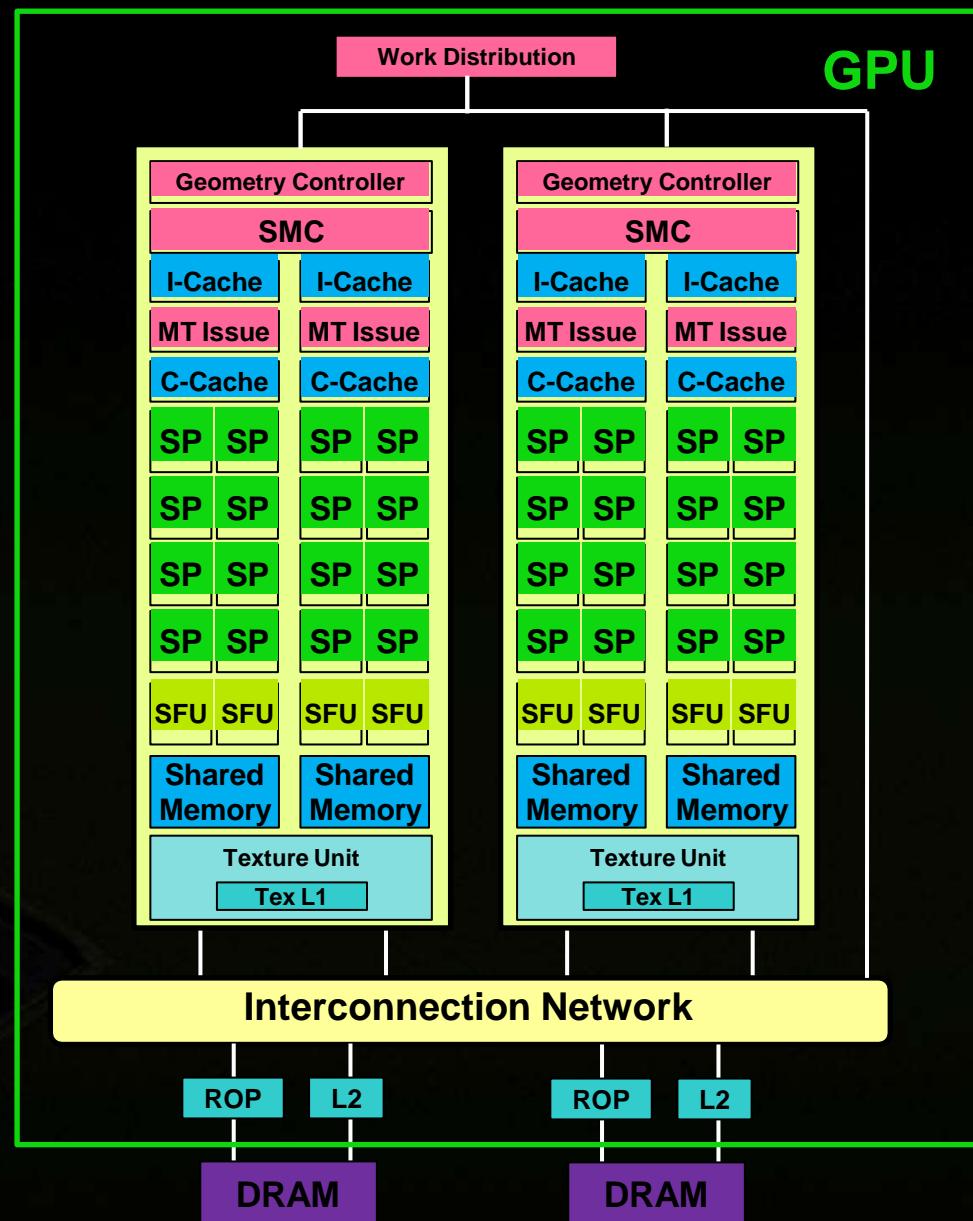
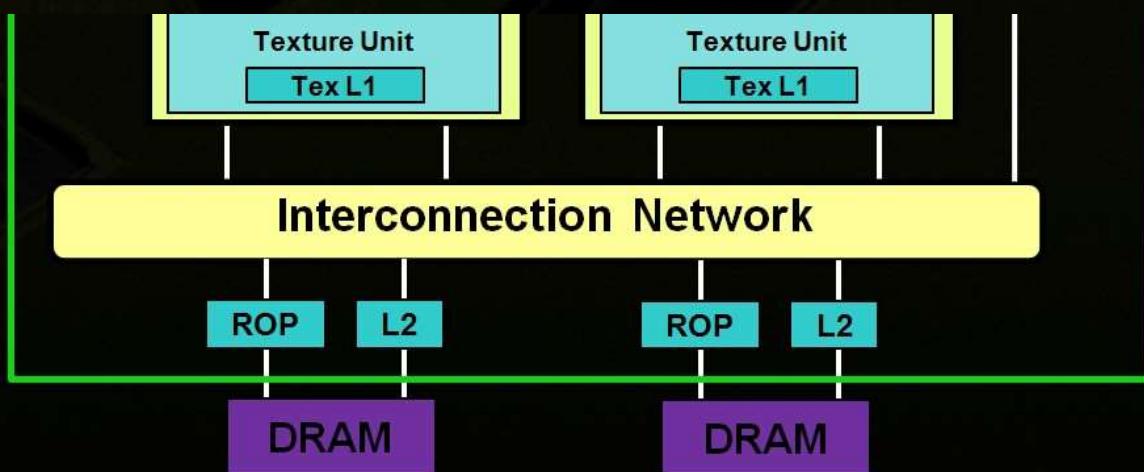
- Provide surface coloring for 3D meshes (a "wrapping")
- 3D mesh has "texture coordinates", hardware looks up 2D color array



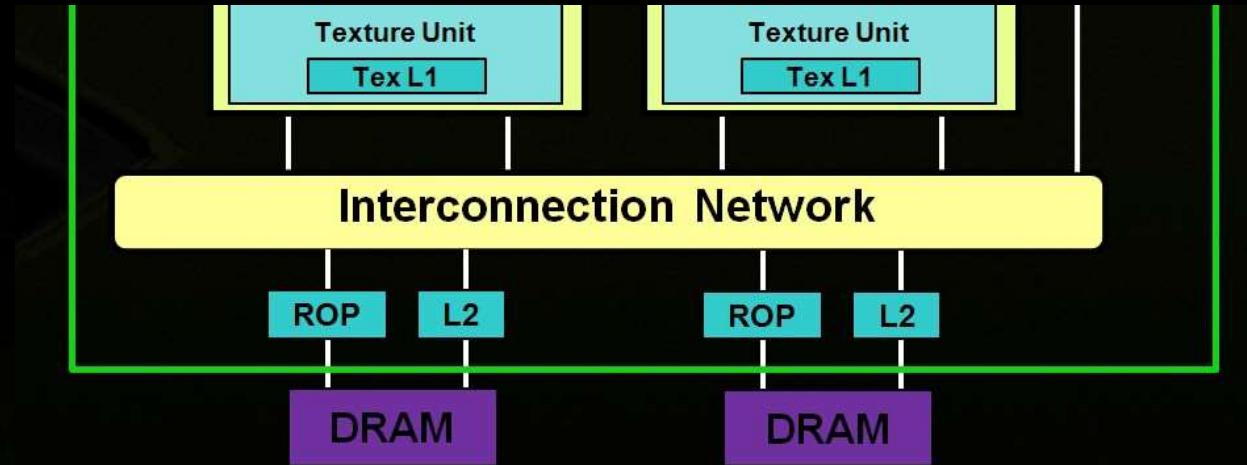
# Texture Unit



- **Texture Read:**  
**Global memory read via texture hardware path**
- **Data reads are cached**
  - Texture Cache (separate from L1)
  - Specialized for 2D/3D spatial locality



# Texture Unit



- **Data conversion (integer to float, 16 bit float to 32 bit float)**
- **Data Interpolation (aka Filtering)**
  - Linear / bilinear / trilinear data interpolation in hardware
- **Boundary modes (for “out-of-bounds” addresses)**
  - Addressable in 1D, 2D, or 3D.
  - Coordinate normalization mode (access becomes resolution-independent)
  - Clamp to edge / Clamp to Border color / Repeat / Mirror
- **Works best with CUDA Array as Data Storage**

# CUDA Array



- **Opaque object for 1D/2D/3D data storage in global memory**
- **Purpose**
  - Optimal caching for 2D/3D spatial locality  
(for 2D/3D threadblocks accessing in "cloud" pattern)
  - Standard exchange format for OpenGL/DirectX texture exchange
- **Data resides in Global Memory**
  - Host access through special cudaMemcpy operations
  - Device access through texture reads or surface read/write (explained later)

# Setup of Textures



## ▪ Host Code

- Create Channel Description
  - Used for allocation of CUDA arrays and texture binding
  - Defines number of channels, type and bitness of data stored
  - E.g. 1 x float32, 4 x uchar
- Declare a texture reference (must currently be at file-scope)
- Allocate texture data storage (global memory as linear/pitch linear, or CUDA array)
- Bind texture to its data storage (device pointer / CUDA array)

## ▪ Device Code

- Fetch data using texture reference
  - Textures bound to linear memory: `tex1Dfetch(tex, int coord)`
  - Textures bound to pitch linear memory: `tex2D(tex, float2 coord)`
  - Textures bound to CUDA arrays: `tex1D()` `tex2D()` `tex3D()`
  - Layered textures bound to CUDA arrays: `tex1DLayered()` `tex2DLayered()`



# Texture binding modes

- **Texture references are bound to device pointer or CUDA Array**
  - Sets the data source for all reads from this texture reference
- **Bind to linear memory (device pointer)**
  - Texture is bound directly to global memory address
  - Large 1D extents ( $2^{27}$  elements), but integer indexing only
  - Simple, but: No data interpolation, no clamp/repeat addressing modes
- **Bind to pitch linear (device pointer)**
  - Texture is bound directly to global memory address of pitchlinear data
  - 2D indexing (but cache locality still sees pitchlinear mem)
  - Provides data interpolation and clamp/repeat addressing modes
  - SDK: "simplePitchLinearTexture"
- **Bind to CUDA array (handle)**
  - Texture is bound to CUDA array (1D, 2D, or 3D)
  - Float addressing (within array bounds, or normalized bounds)
  - Provides data interpolation and clamp/repeat addressing modes
  - Addressing modes (clamping, repeat)
  - SDK: "simpleTexture", "simpleTexture3D", "simpleTextureDrv"



# Linear memory example (1D texture, simple caching access)

- Host Code

```
// global reference (visible for host and device code)
texture<float, cudaTextureType1D, cudaReadModeElementType> linmemTexture;
...
// host code: bind texture reference to linear memory
// (use implicitly created channel description)
cudaBindTexture(NULL, linmemTexture, d_linmemory_ptr,
                cudaCreateChannelDesc<float>(), linmemory_size);
// start kernel that uses texture reference!
```

- Device Code

```
float A = tex1Dfetch(linmemTexture, position);
```

# CUDA Array example (2D texture interpolation)



## Host Code

```
// global declaration of 2D float texture (visible for host and device code)
texture<float, cudaTextureType2D, cudaReadModeElementType> tex;

...
// Create explicit channel description (could use an implicit as well)
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);

// Allocate CUDA array in device memory
cudaArray* cuArray;
cudaMallocArray(&cuArray, &channelDesc, width, height);

// Copy some data located at address h_data in host memory into CUDA array
cudaMemcpyToArray(cuArray, 0, 0, h_data, size, cudaMemcpyHostToDevice);

// Set the texture parameters (more sophisticated than a simple linear memory texture)
// boundary handling in x and y-direction!
tex.addressMode[0] = cudaAddressModeWrap; tex.addressMode[1] = cudaAddressModeWrap;
tex.filterMode = cudaFilterModeLinear; // linear interpolation
tex.normalized = true; // normalized coordinate bounds [0.0 .. 1.0]

// Bind the array to the texture reference
cudaBindTextureToArray(tex, cuArray, channelDesc);
```

## Device Code

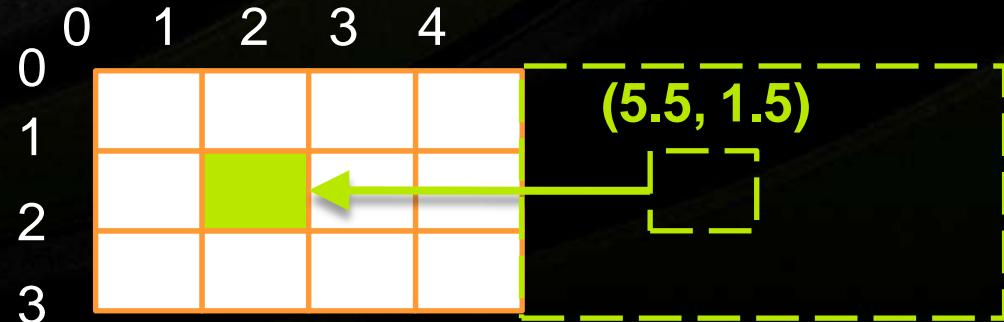
```
float value = tex2D(tex, xpos, ypos);
```

# Texture Coordinates

- Texture fetch in device code takes floating point **texture coordinates**
- **Lookup mode** and coordinates determine data element fetch from global memory:  
"Nearest neighbour" mode uses less data than "linear interpolation" mode
- Coordinate bounds can reflect input data dimensions, or be **normalized** (0.0 .. 1.0)
- Boundary handling in different ways:

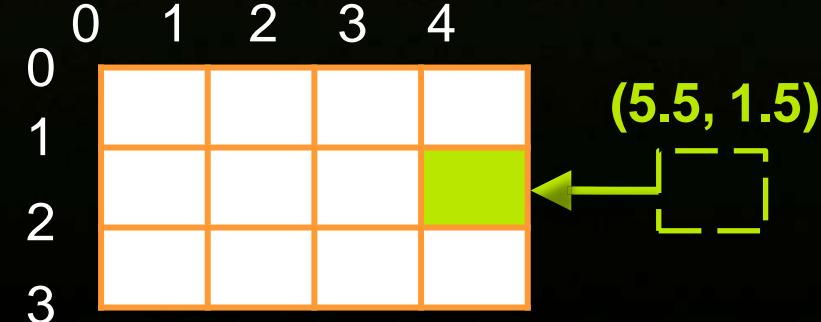
## Wrap

- Out-of-bounds coordinate is wrapped (modulo arithmetic)



## Clamp

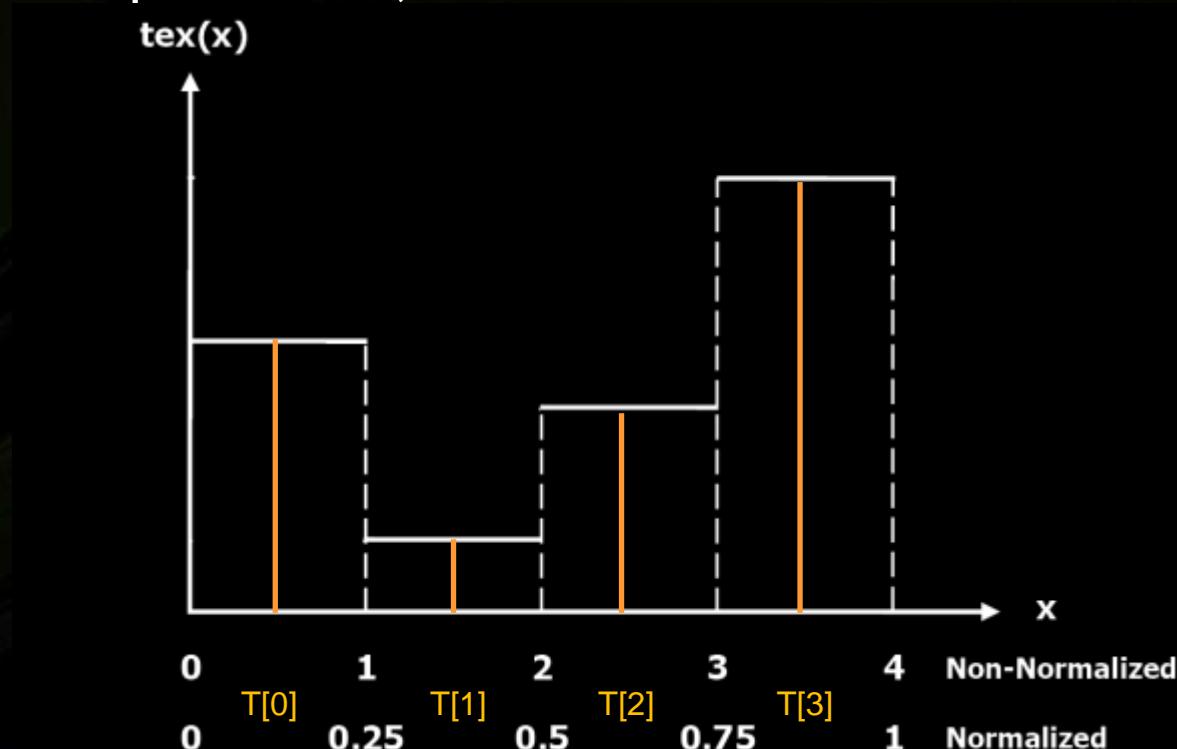
- Out-of-bounds coordinate is clamped to closest boundary



# Texture Data Processing



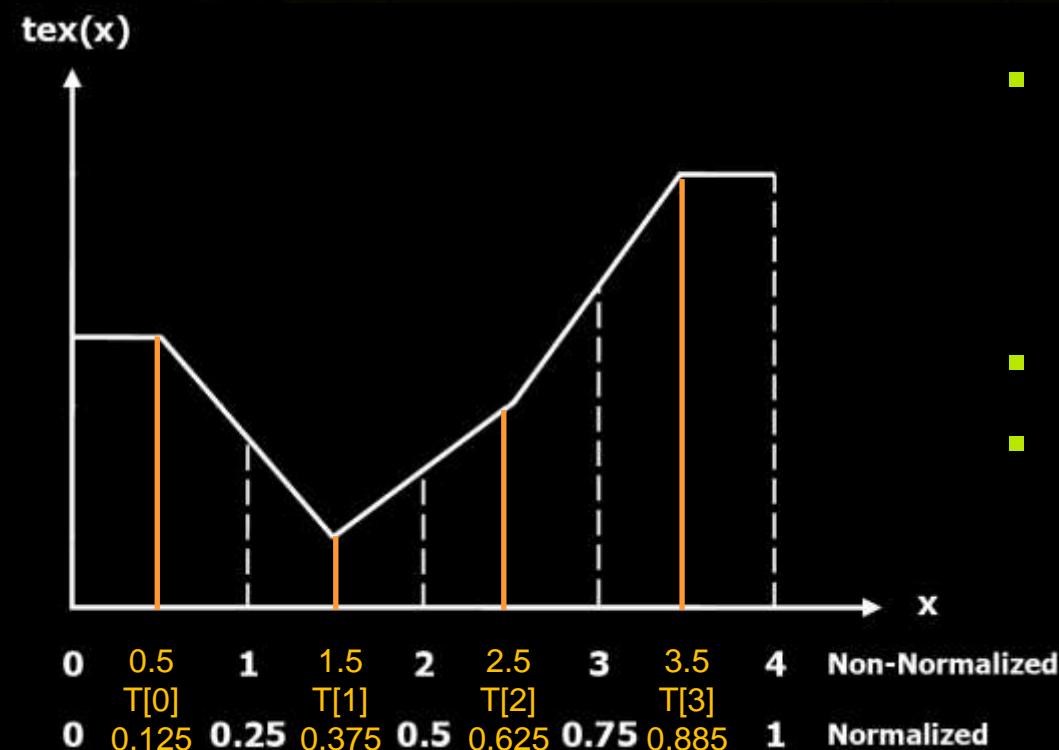
- **Texture unit can convert integer input to floating point output**
  - E.g. 8bit input: uchar4(255, 128, 0, 0) becomes float4(1.0, 0.5, 0.0, 0.0)
- **Coordinate to Data mapping for "Nearest neighbour" mode:**
  - Example: Input data T, four values:



- All input data elements cover output ranges
- Details in Programming Guide Appendix E

# Texture Interpolation

- **Texture unit can interpolate between adjacent data elements**
  - Fractional part of texture coordinate becomes interpolation weight  
(Note: Interpolation weight is 8 bit quantized!)
  - Only in float conversion mode, bind to CUDA array or pitchlinear memory



- Warning:  
Input's data values  
can NOT be read at  
integer offsets!
- But: Additional GFlops!
- Details in  
Programming Guide,  
Appendix E

# Surfaces

- Device code can read and write CUDA arrays via Surfaces  
(Programming Guide, Appendix B.9 and SDK "simpleSurfaceWrite" )
- Requires Compute Capability 2.0 or higher
- Currently available for 1D and 2D CUDA arrays
  - Use flag `cudaArraySurfaceLoadStore` during CUDA array creation
- Can also bind surface and texture to same CUDA array handle (write-to-texture)
- Surface operations have
  - no interpolation or data conversion
  - but some boundary handling
- Texture cache is not notified of CUDA array modifications!
  - Start new kernel to pick up modifications
- Note: Surface writes take x coordinate in byte size!



# Layered Textures

- **Requires Compute Capability 2.0 or higher and CUDA 4.0**
- **3D coordinate, but z dimension is only integer (only xy-interpolation)**
- **Ideal for processing multiple textures with same size/format**
  - Reduced CPU overhead: single binding for entire texture array
  - Large sizes supported on Fermi GPUs with CC  $\geq 2.0$  (up to 16k x 16k x 2k)
  - e.g. Medical Imaging, Terrain Rendering (flight simulators), etc.
- **Faster Performance**
  - Faster than 3D Textures: better texture cache performance, since Linear/Bilinear interpolation only within a layer, not across layers
  - Fast interop with OpenGL / Direct3D for each layer
  - No need to create/manage a texture atlas
- **Can be bound to specially created CUDA Arrays**
  - Use `cudaMalloc3DArray()` with `cudaArrayLayered` flag
- **Details: Programming Guide 4.0, 3.2.10.1.5 Layered Textures**

# Usage Advice

- **Texture bound to linear memory (device pointer)**
  - No interpolation!
  - Integer addressing, large extents ( $2^{27}$  elements)
  - Use if texture cache shall assist L1 cache
- **Texture bound to CUDA arrays (handle)**
  - Use if texture content changes rarely  
(Can still modify content via surface writes or `cudaMemcpy`)
- **Texture bound to pitch linear memory (device pointer)**
  - Has float/integer addressing, filtering, and clamp/repeat addressing modes
  - Use if conversion to CUDA arrays too tedious (performance / code)
  - Performance caveat: 2D Threadblocks/Warps should only access rows!



# 16-bit floating point textures

- GPU supports 16bit floating point format (aka *half*)
  - Used e.g. for High Definition Color Range in OpenEXR format
  - Specified in IEEE standard 754-2008 as binary2
  - Not native for CPU, but C++ datatype routines are easy to find online
- Compact representation of floating point data arrays
  - CUDA arrays can hold 16bit float, use `cudaCreateChannelDescHalf*`()
  - Device code (e.g. for GPU manipulation of pitchlinear memory):  
`__float2half(float)` and `__half2float(unsigned short)`
- Texture unit hides 16 bit float handling
  - Texture lookups convert 16bit half to 32 bit float, can also interpolate!
  - Lookup result is always 32 bit float



# Texture exchange with OpenGL/DirectX

- Interoperability API can bind OpenGL / DirectX context to CUDA C context
- Textures/Surfaces from graphics APIs are exported as CUDA Arrays
  - Currently available for 2D textures only
  - Direction flags tell which way data exchange goes from graphics API towards CUDA C (read-only, write-discard, read/write)
  - Host code can then modify textures with cudaMemcpy
  - Device code can modify textures with surface read/write:  
E.g. while registering an OpenGL texture, use cudaGraphicsGLRegisterImage()  
with flag cudaGraphicsRegisterFlagsSurfaceLoadStore
- See Programming Guide 4.0, 3.2.11 Graphics Interoperability
- See Reference Manual 4.0, 14.1 Graphics Interoperability
- SDK: "postProcessGL", "simpleD3D11Texture" and similar

# Profiler hints

- **Visual Profiler has profiling signals for texture requests and texture cache**
  - Compute Capability < 2.0: `texture_cache_hit`, `texture_cache_miss`
  - Compute Capability >= 2.0: `tex_cache_requests`, `tex_cache_misses`
  - Derived signals:
    - Texture cache memory throughput (GB/s), Texture cache hit rate (%)
  - Use these to determine texture cache assistance
- **Visual Profiler can also derive L2 cache requests caused by texture unit**
  - L2 cache texture memory read throughput (GB/s)
  - Compare to global memory throughput to determine how L2 cache assists all texture units' caches
- **See Visual Profiler user guide, "Derived Statistic"**

# Summary



- **Texturing provides additional performance**
  - Extra cache capacity
  - Linear interpolation of adjacent data in hardware
  - Array boundary handling
  - Integer-to-float conversion, data unpacking
- **Algorithmic design considerations**
  - Texture binding modes (linear memory, pitchlinear memory, CUDAArray)
  - Texture coordinate offsets for correct linear interpolation
  - 8bit weight quantization during linear interpolation
  - Can't flush texture cache during kernel execution
  - 3D: xy-interpolation (layered textures) vs. Trilinear xyz-interpolation (3D textures)



# fast box filter

- Allows box filter of any width with a constant cost
  - Rolling box filter
- Uses a sliding window
  - Two adds and a multiply per output pixel
  - Adds new pixel entering window, subtracts pixel leaving
- Iterative Box Filter  $\approx$  Gaussian blur
- Using pixel shaders, it is impossible to implement a rolling box filter
  - Each thread requires writing more than one pixel
- CUDA allows executing rows/columns in parallel
  - Uses tex2D to improve read performance and simplify addressing

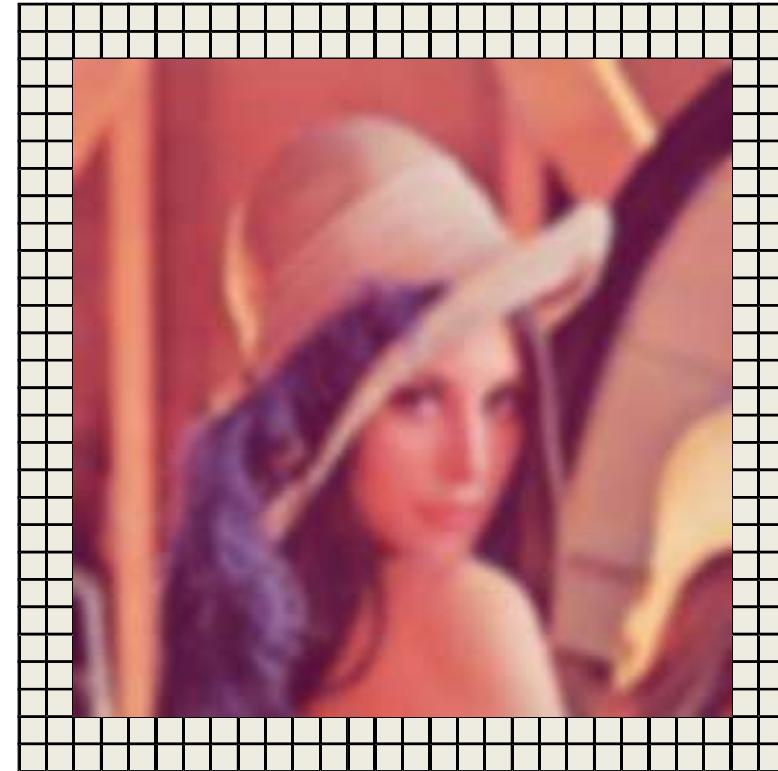
# fast box filter

- Separable, two pass filter. First row pass, then column pass

**Source Image (input)**



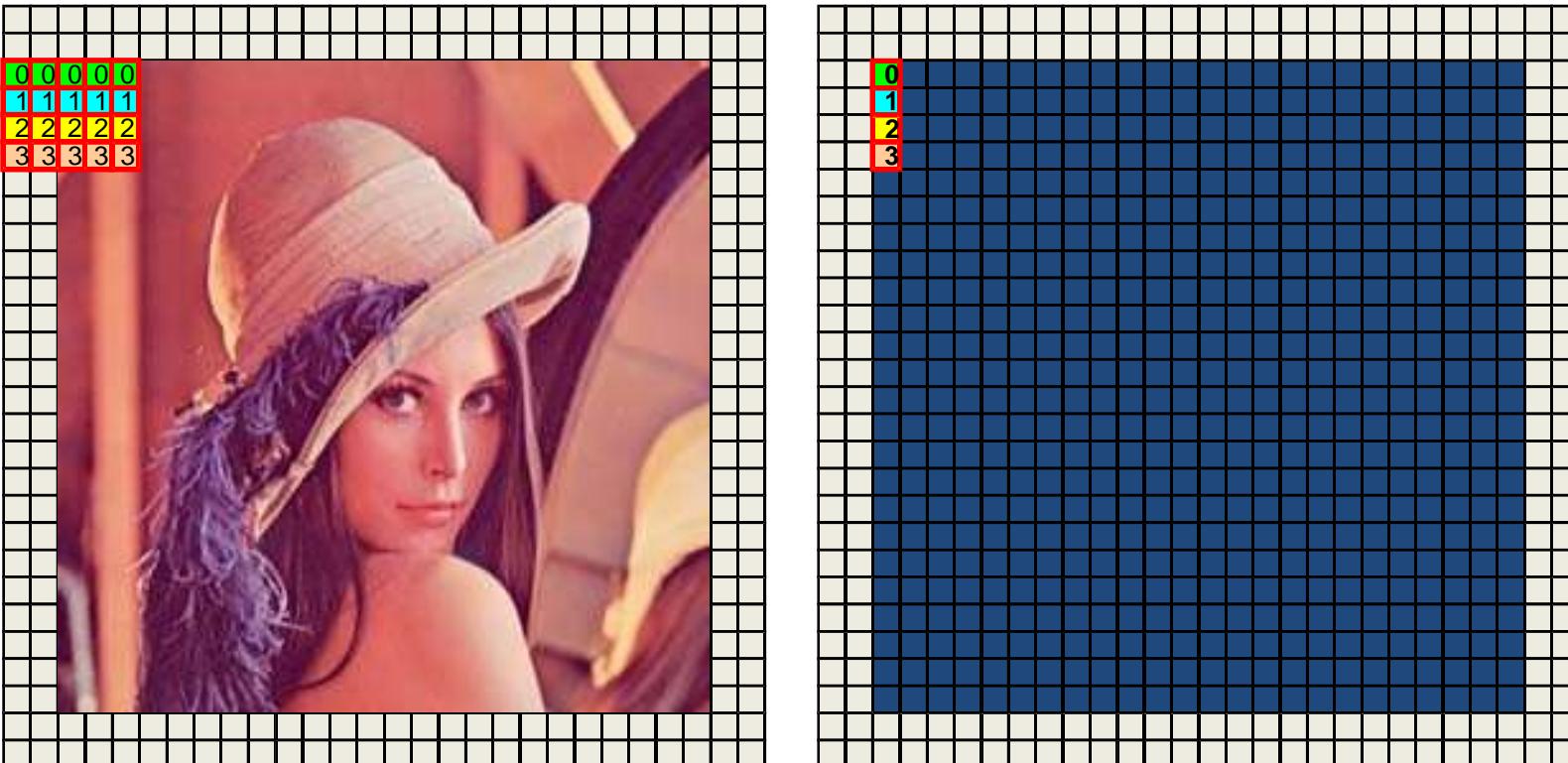
**Output Result**



© 2008 NVIDIA Corporation.

# fast box filter (row pass pixel 0)

- Assume  $r = 2$ , each thread works pixels along the row and sums  $(2r+1)$  pixels
- Then average  $(2r+1)$  pixels and writes to destination  $(i, j)$

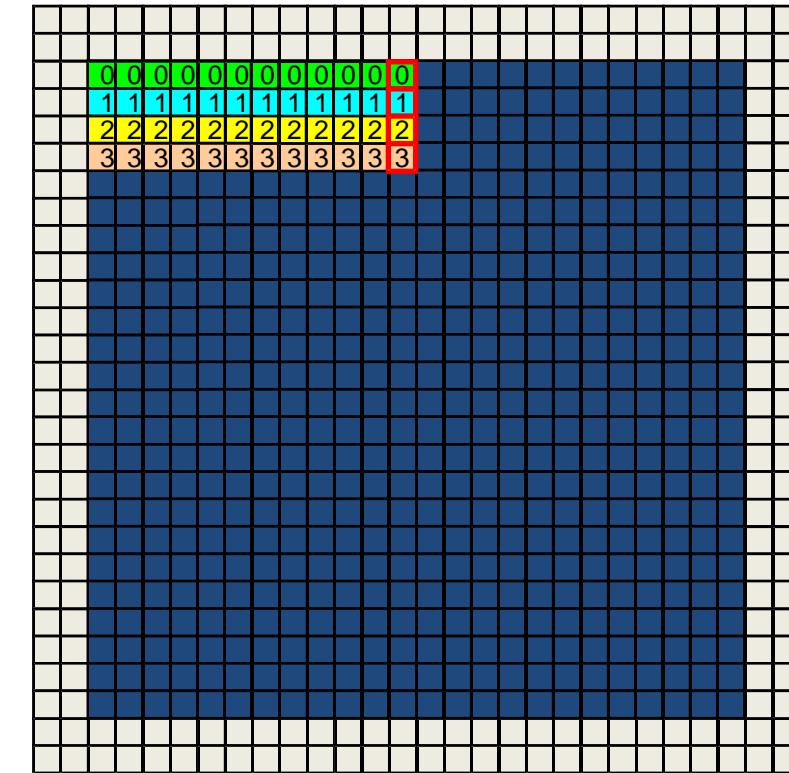


$$\frac{3 + 3 + 3 + 3 + 3}{(2r + 1)} = 3$$

© 2008 NVIDIA Corporation.

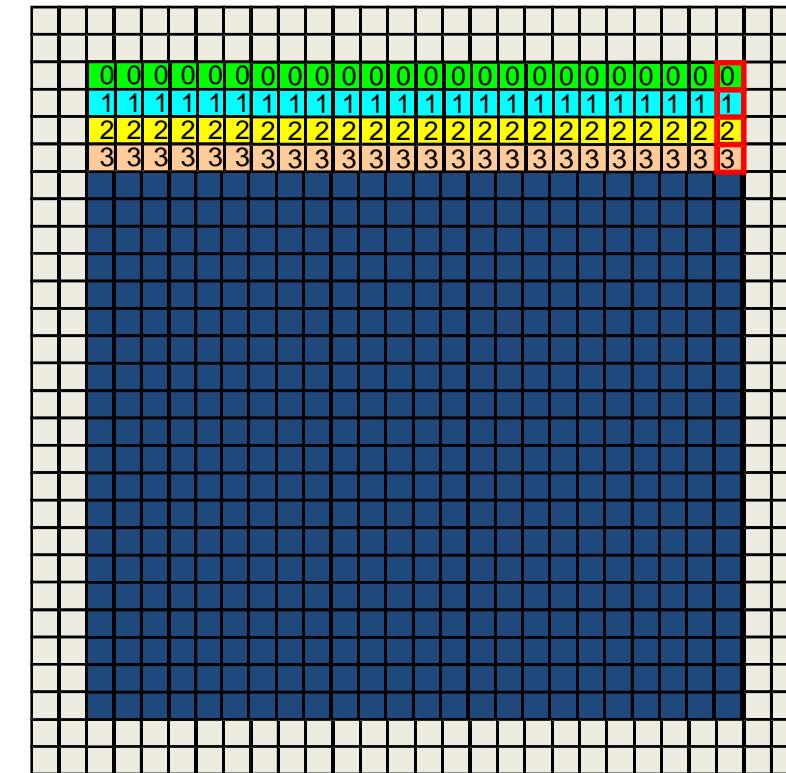
# fast box filter (row pass pixel 11)

- Take previous sum from pixel 10, -1 pixel ( $i - (r+1), j$ ), +1 pixel ( $i + (r+1), j$ )
- Average  $(2r+1)$  pixels and Output to  $(i, j)$



# fast box filter (finish row pass)

- Each thread continues to iterate until the entire row of pixels is done
- Average then Write to  $(i, j)$  in destination image
- A single thread writes the entire row of pixels

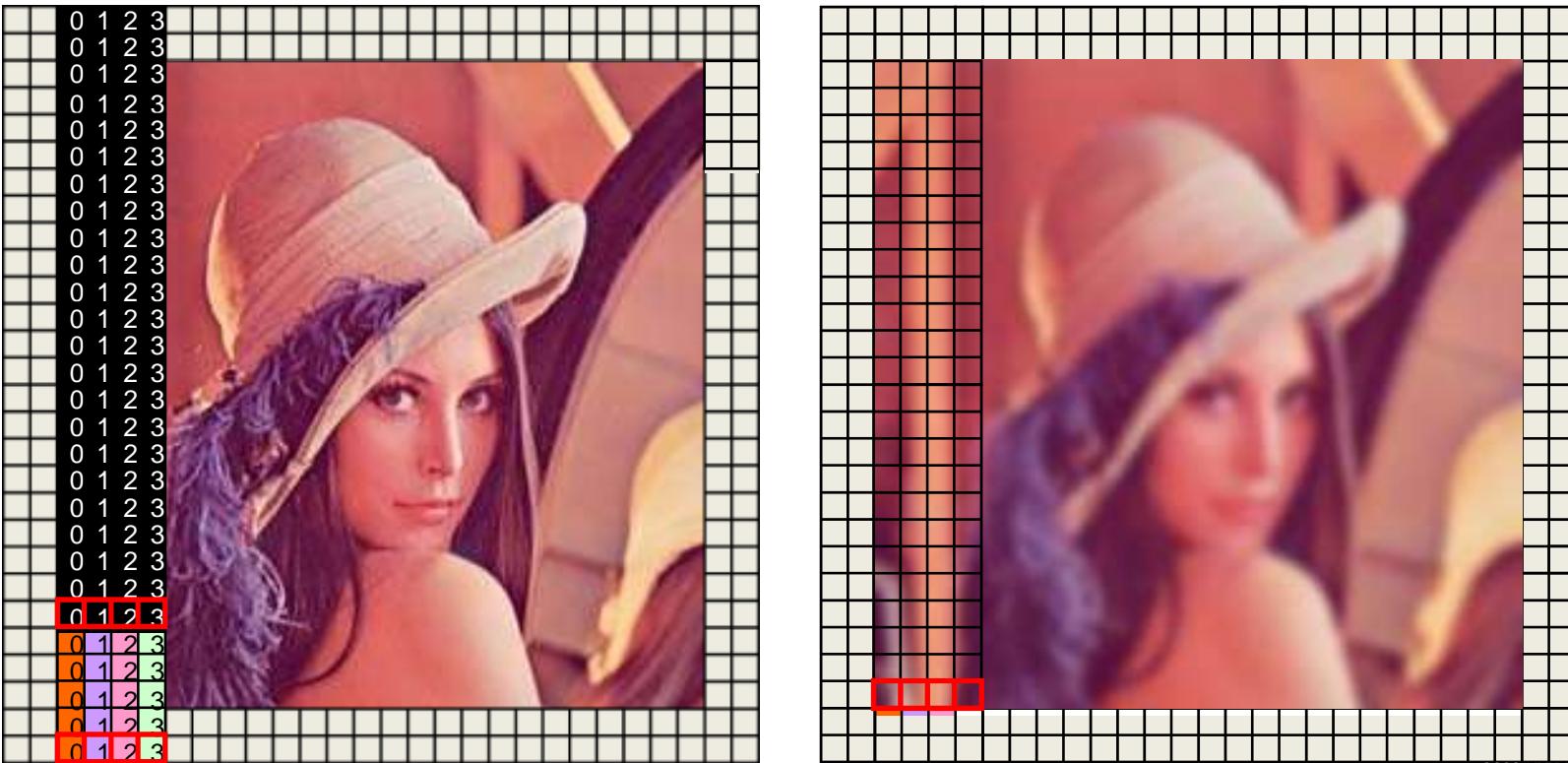


- Note: Writes are not coalesced
- Solution: Use shared memory to cache results per warp,  
`call __syncthreads()`, then copy to global mem to achieve Coalescing

© 2008 NVIDIA Corporation.

# Column Filter Pass (final)

- Threads  $(i, j)$  read from global memory and sum along the column from row pass image, we get Coalesced Reads
- Compute pixel sums from previous pixel, -1 pixel, +1 pixel
- Average result and Output to  $(i, j)$ . We get Coalesced Writes



© 2008 NVIDIA Corporation.

# samples / boxFilter

```
// process row
__device__ void
d_boxfilter_x(float *id, float *od, int w, int h, int r)    // main loop
{
    float scale = 1.0f / (float)((r << 1) + 1);

    float t;
    // do left edge
    t = id[0] * r;

    for (int x = 0; x < (r + 1); x++)
    {
        t += id[x];
    }

    od[0] = t * scale;

    for (int x = 1; x < (r + 1); x++)                         }
    {
        t += id[x + r];
        t -= id[0];
        od[x] = t * scale;
    }

    // do right edge
    for (int x = w - r; x < w; x++)
    {
        t += id[w - 1];
        t -= id[x - r - 1];
        od[x] = t * scale;
    }
}
```

```
// process column
__device__ void
d_boxfilter_y(float *id, float *od, int w, int      // main loop
h, int r)
{
    float scale = 1.0f / (float)((r << 1) + 1);

    float t;
    // do left edge
    t = id[0] * r;

    for (int y = 0; y < (r + 1); y++)
    {
        t += id[y * w];
    }
    od[0] = t * scale;

    for (int y = 1; y < (r + 1); y++)
    {
        t += id[(y + r) * w];
        t -= id[0];
        od[y * w] = t * scale;
    }

    // main loop
    for (int y = (r + 1); y < (h - r); y++)
    {
        t += id[(y + r) * w];
        t -= id[((y - r) * w) - w];
        od[y * w] = t * scale;
    }

    // do right edge
    for (int y = h - r; y < h; y++)
    {
        t += id[(h-1) * w];
        t -= id[((y - r) * w) - w];
        od[y * w] = t * scale;
    }
}
```

```
__global__ void
d_boxfilter_x_global(float *id, float *od, int w, int h, int r)
{
    unsigned int y = blockIdx.x*blockDim.x + threadIdx.x;
    d_boxfilter_x(&id[y * w], &od[y * w], w, h, r);
}
```

```
__global__ void
d_boxfilter_y_global(float *id, float *od, int w, int h, int r)
{
    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
    d_boxfilter_y(&id[x], &od[x], w, h, r);
}
```

# Histogram

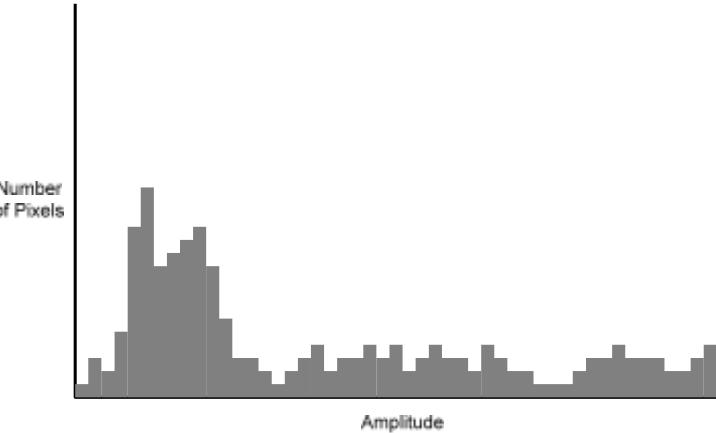


- Extremely Important Algorithm
  - Histogram Data used in large number of “compound” algorithms:
    - Color and contrast improvements
    - Tone Mapping
    - Color re-quantization/posterize
    - Device Calibration

© 2008 NVIDIA Corporation.

# Histogram Algorithm

- Distribution of intensities/colors in an image
- Standard algorithm:



```
for all i in [0, max_luminance]:  
    h[i] = 0;  
for all pixel in image:  
    ++h[luminance(pixel)]
```

- How to parallelize?



# Histogram Parallelization

- Subdivide “for-all-pixel” loop
  - Thread works on block of pixels (in extreme, one thread per pixel)
  - Need `++h[luminance(pixel)]` to be atomic (global atomics  $\geq$  compute1\_1)
- Breaking up Image I into sub-images  
 $I = \text{UNION}(A, B)$ :
  - $H(\text{UNION}(A, B)) = H(A) + H(B)$
  - Histogram of concatenation is sum of histograms

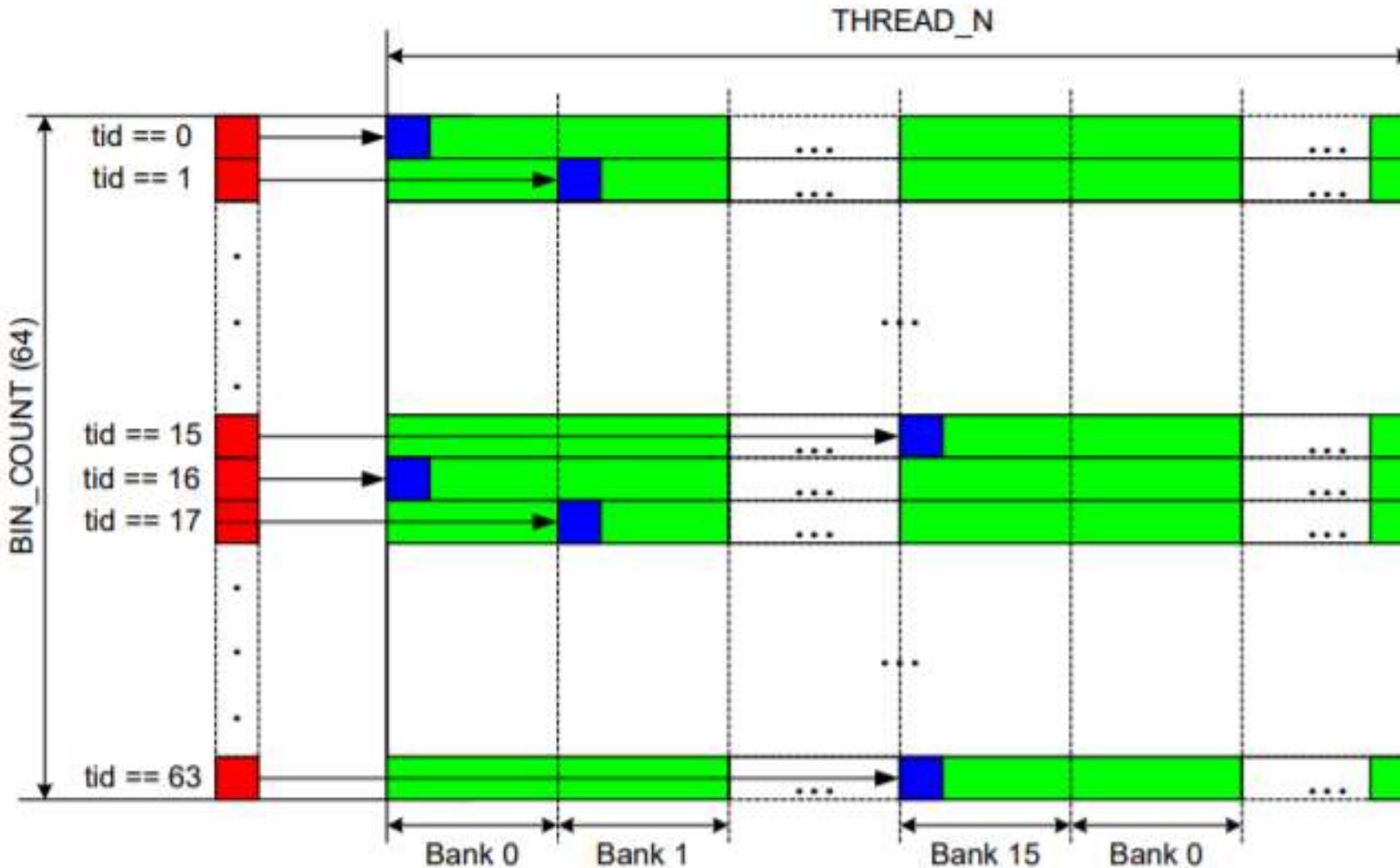
© 2008 NVIDIA Corporation.



# Better Parallel Histogram

- Have one histogram per thread
  - Memory consumption!
  - Consolidate sub-histograms in parallel (parallel-reduction).
- CUDA:
  - Histograms in shared memory
  - $64 \text{ bins} * 256 \text{ threads} = 16 \text{kByte}$  (8bit bins)
  - $256 \text{ bins} * 256 \text{ threads} = 64 \text{kByte}$  (8bit bins)

# Better Parallel Histogram



orporation.

# Image Transpose

- Naïve implementation:

```
kernel(char * pi, int si, char * po, int so,
       int w, int h)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (y < w && x < h)
        OUTPUT_PIXEL(y, x) = INPUT_PIXEL(x, y);
}
```



# Problems with Naïve Code

- Memory reads AND writes not coalesced because reading/writing bytes, not words.
- Idea:
  - Need to read (at least) 16 words in subsequent threads.



# Improved Transpose Idea

- Subdivide image into “micro-blocks” of 4x4 pixels (16Byte).
- Thread blocks of 16x16 threads.
- Each thread operates on a micro-block.
- Shared memory for micro-blocks:  
 $16 \times 16 \times 4 \times 4 = 4\text{kByte}$ .



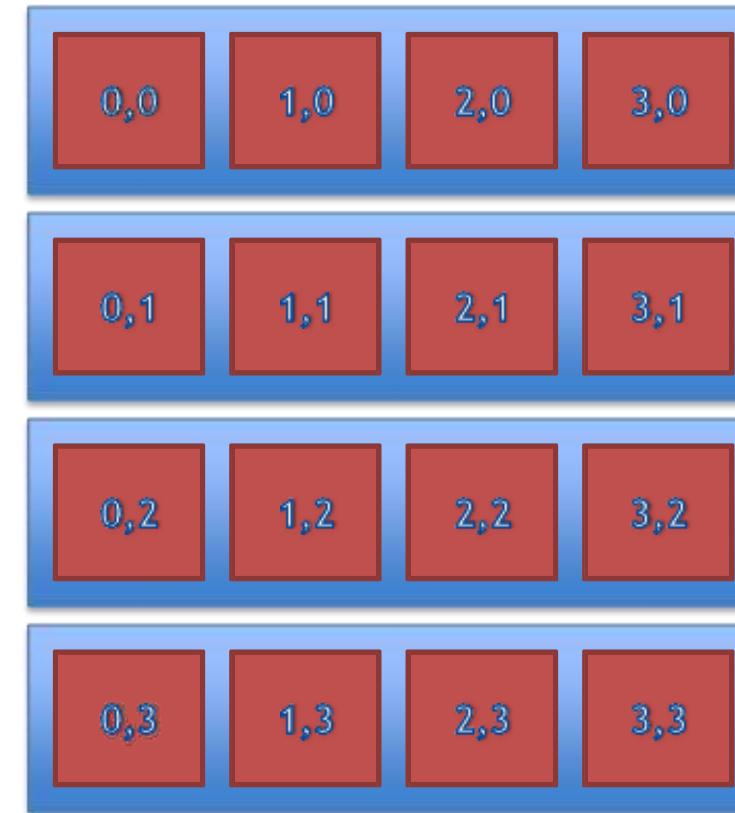
# Basic Algorithm

- Each thread reads its micro-block into shared memory.
- Each thread transposes its micro-block.
- Each thread writes its micro-block back into global memory.



# Reading and Writing MicroBlocks

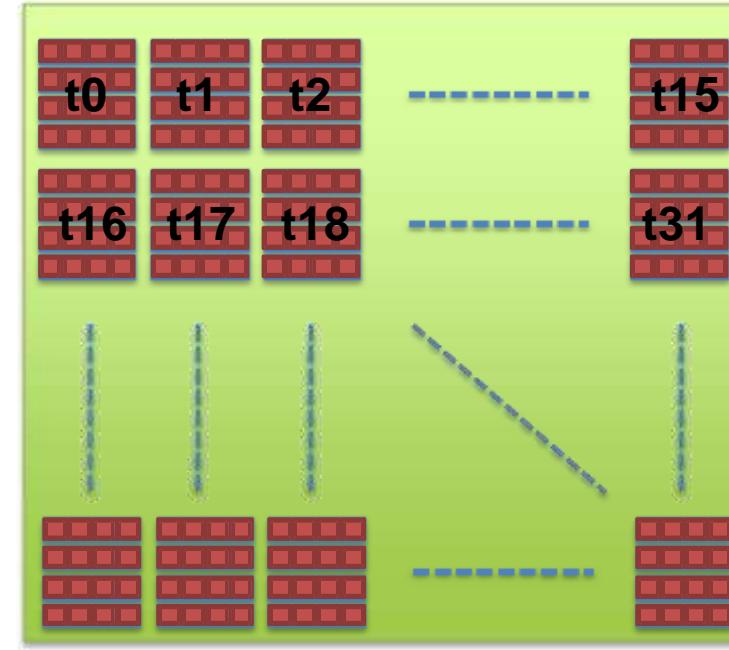
- Reading one row of MicroBlock via **uchar4** rather than 4x **unsigned char**



© 2008 NVIDIA Corporation.

# 16x16 Thread Blocks

- One (16-thread) warp reads one row of MicroBlocks.
- One 16x16 block of threads deals with a 64x64 pixel region (8-bit luminance pixels).



# Pseudo Code

- Assume single 64x64 image.

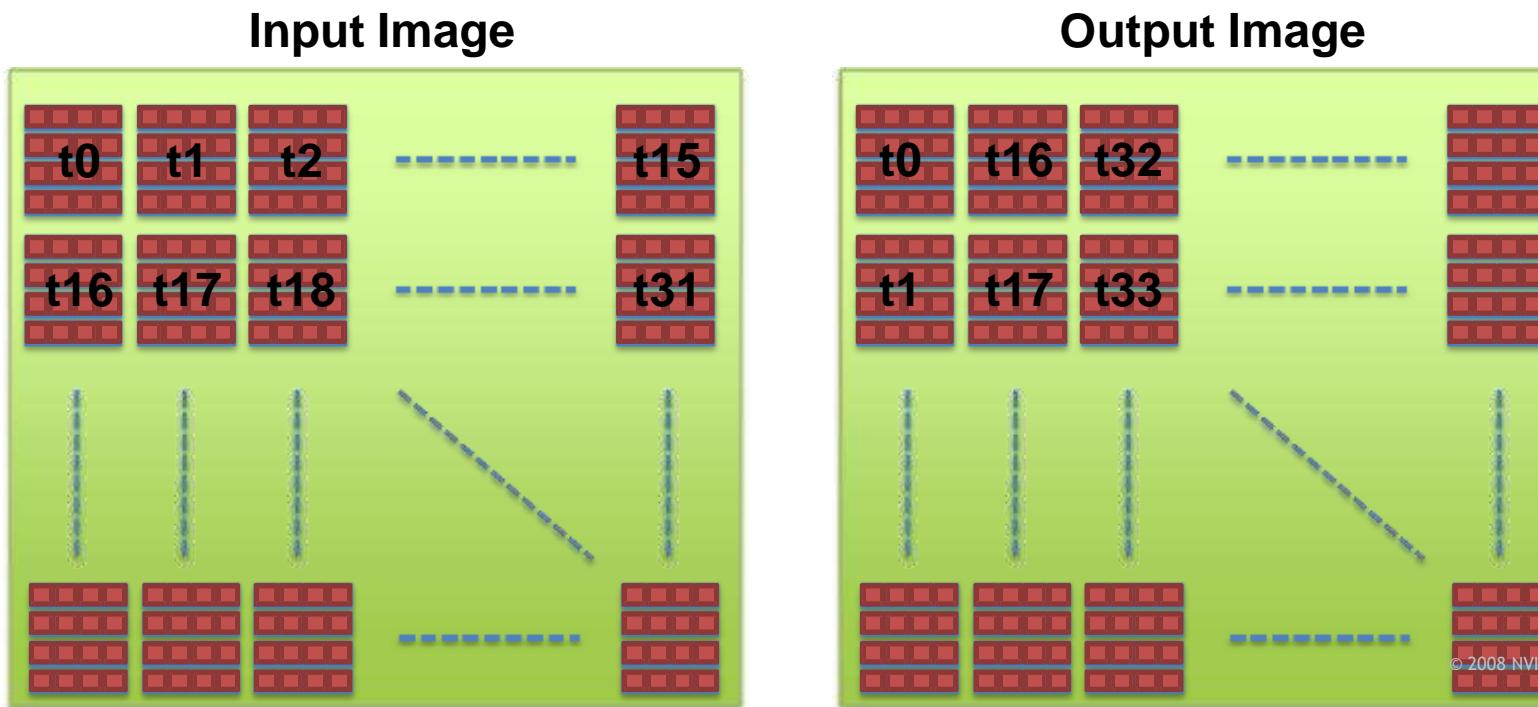
```
kernel(...)  
{  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
  
    readMicroBlock(image, i, j, shared, i, j);  
    transposeMicroBlock(shared, i, j);  
    writeMicroBlock(shared, i, j, image, j, i);  
}
```

- Problem: Non-coalesced writes!



# Write Coalescing for Transpose

- `readMicroBlock(image, i, j, shared, i, j);`
- `writeMicroBlock(shared, i, j, image, j, i);`



© 2008 NVIDIA Corporation.

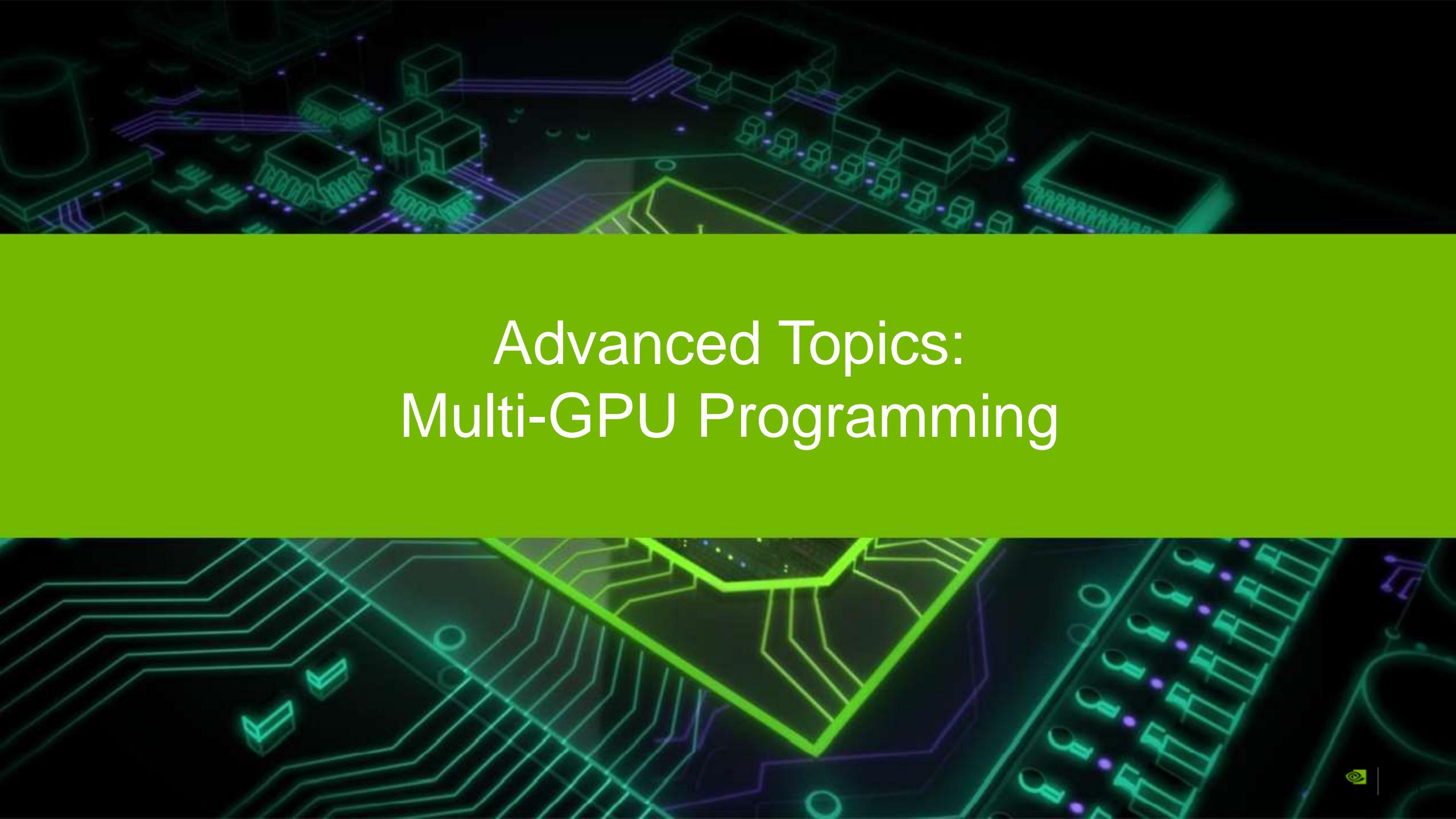
# Coalesced Writes

- Simple fix:

```
kernel(...)  
{  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
  
    readMicroBlock(image, i, j, shared, i, j);  
    transposeMicroBlock(shared, i, j);  
    __syncthreads();  
    writeMicroBlock(shared, j, i, image, i, j);  
}
```

- Must `__syncthreads()` because  $T_{i,j}$  now writes data produced by  $T_{j,i}$ .

© 2008 NVIDIA Corporation.



# Advanced Topics: Multi-GPU Programming

# MOTIVATION

## Why use multiple GPUs?

Need to compute larger, e.g. bigger networks, car models, ...

Need to compute faster, e.g. weather prediction

Better energy efficiency with dense nodes with multiple GPUs

# DGX-1

Two fully connected quads,  
connected at corners

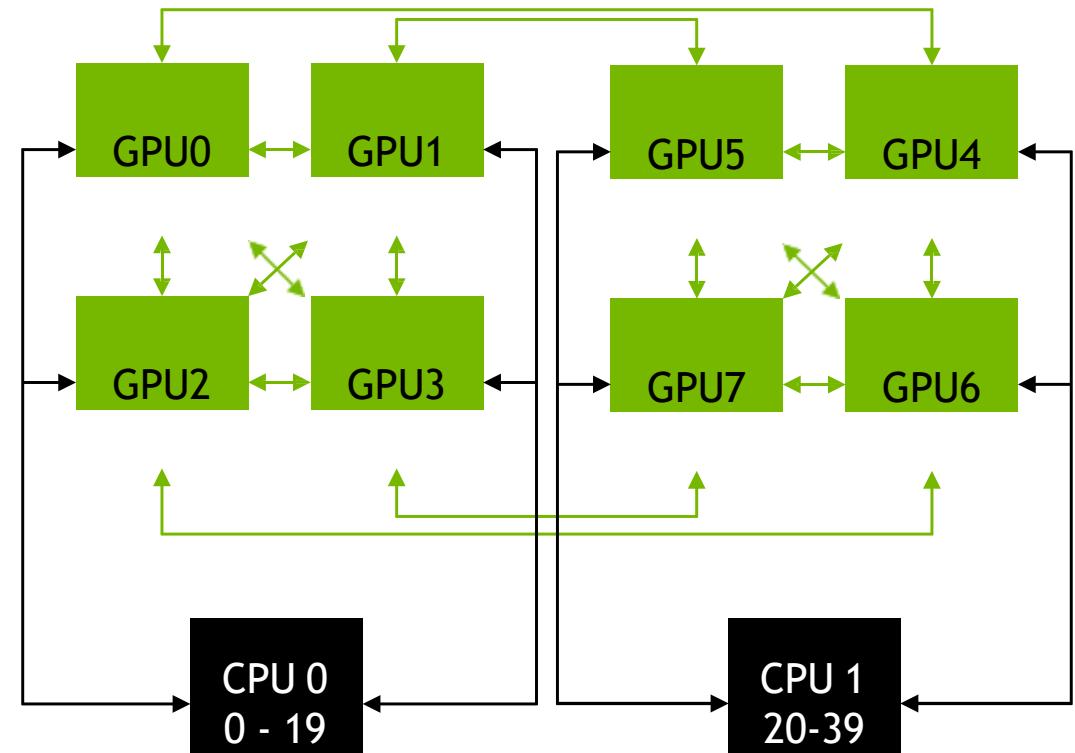
160GB/s per GPU bidirectional to  
Peers

Load/store access to Peer Memory

Full atomics to Peer GPUs

High speed copy engines for bulk  
data copy

PCIe to/from CPU



# EXAMPLE: JACOBI SOLVER

Solves the 2D-Laplace Equation on a rectangle

$$\Delta u(x, y) = \mathbf{0} \quad \forall (x, y) \in \Omega \setminus \delta\Omega$$

Dirichlet boundary conditions (constant values on boundaries) on left and right boundary

Periodic boundary conditions on top and bottom boundary

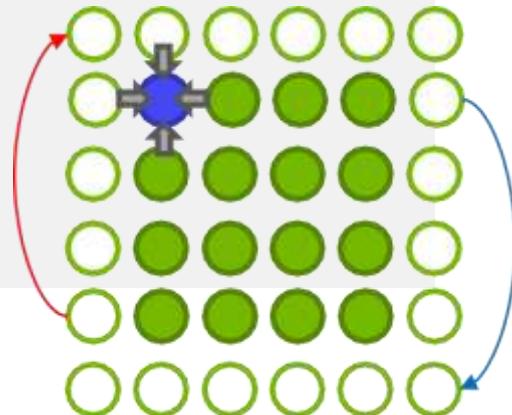
# EXAMPLE: JACOBI SOLVER

## Single GPU

While not converged

Do Jacobi step:

```
for( int iy = 1; < ny-1; iy++ )  
    iy  
    for( int ix = 1; < ny-1; ix++ )  
        ix  
        a_new[iy*nx+ix] = -0.25 * *nx+ix-1]  
            -( a[ iy *nx+(ix+1)] + a[ (iy+1)*nx+ix ] );  
            iy
```



Apply periodic boundary conditions

Swap `a_new` and `a`

Next iteration

# DOMAIN DECOMPOSITION

Different Ways to split the work between processes:

Minimize number of neighbors:

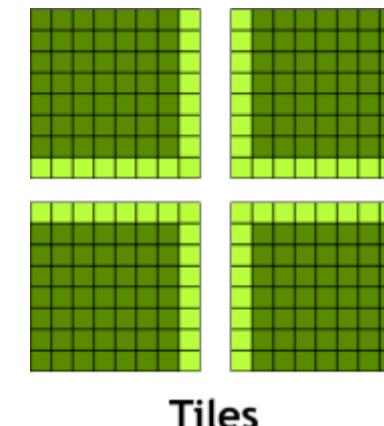
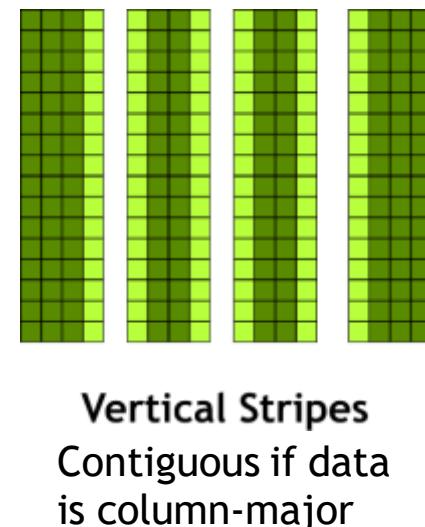
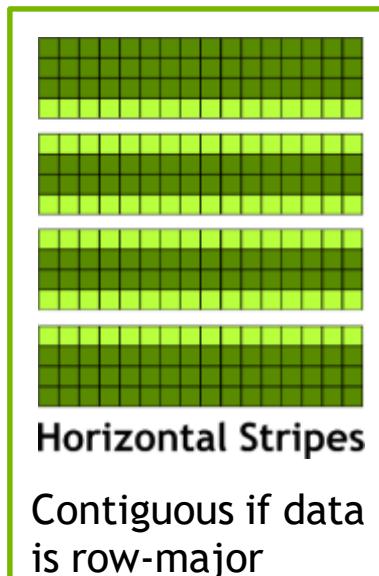
Communicate to less neighbors

Optimal for latency bound communication

Minimize surface area/volume ratio:

Communicate less data

Optimal for bandwidth bound communication



# EXAMPLE: JACOBI SOLVER

## Multi GPU

While not converged

Do Jacobi step:

```
for (int iy = iy_start; iy < iy_end; iy++ )  
for( int ix = 1; ix < ny-1; ix++ )  
    a_new[iy*nx+ix] = -0.25 *  
        -( a[ iy    *nx+(ix+1) ] + a[ iy    *nx+ix-1]  
        + a[(iy-1)*nx+ ix    ] + a[(iy+1)*nx+ix    ] );
```

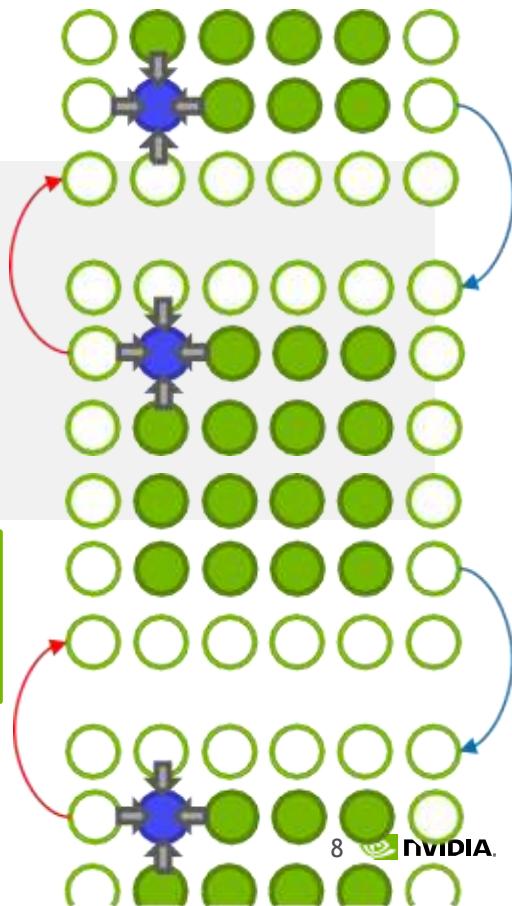
Apply periodic boundary conditions

Exchange halo with 2 neighbors

Swap `a_new` and `a`

Next iteration

One-step with  
ring exchange



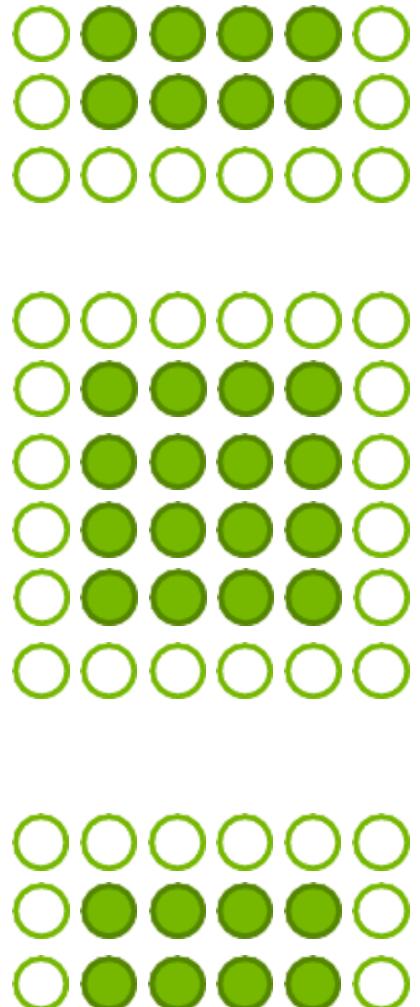
# SINGLE THREADED MULTI GPU PROGRAMMING

```
while ( l2_norm > tol && iter < iter_max ) {
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
        const int top = dev_id > 0 ? dev_id - 1 : (num_devices-1); const int bottom = (dev_id+1)%num_devices;
        cudaSetDevice( dev_id );
        cudaMemsetAsync( l2_norm_d[dev_id], 0 , sizeof(real) );
        jacobi_kernel<<<dim_grid,dim_block>>>( a_new[dev_id], a[dev_id], l2_norm_d[dev_id],
                                                    iy_start[dev_id], iy_end[dev_id], nx );
        cudaMemcpyAsync( l2_norm_h[dev_id], l2_norm_d[dev_id], sizeof(real), cudaMemcpyDeviceToHost );
        cudaMemcpyAsync( a_new[top]+(iy_end[top]*nx), a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ... );
        cudaMemcpyAsync( a_new[bottom], a_new[dev_id]+(iy_end[dev_id]-1)*nx, nx*sizeof(real), ... );
    }
    l2_norm = 0.0;
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) {
        cudaSetDevice( dev_id ); cudaDeviceSynchronize();
        l2_norm += *(l2_norm_h[dev_id]);
    }
    l2_norm = std::sqrt( l2_norm );
    for ( int dev_id = 0; dev_id < num_devices; ++dev_id ) std::swap(a_new[dev_id],a[dev_id]);
    iter++;
}
```

# EXAMPLE JACOBI

## Top/Bottom Halo

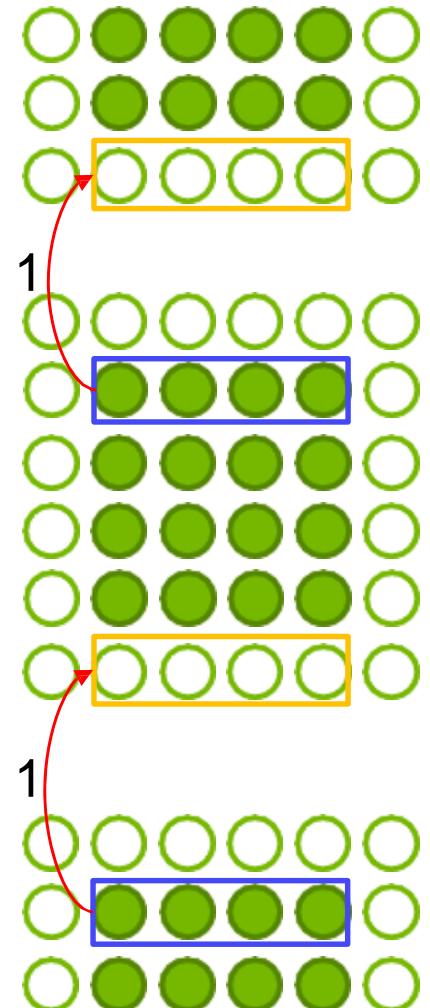
```
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



# EXAMPLE JACOBI

## Top/Bottom Halo

```
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx),  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



# MULTI THREADED MULTI GPU PROGRAMMING

## Using OpenMP

```
int num_devices = 0;

cudaGetDeviceCount( &num_devices );

#pragma omp parallel num_threads( num_devices )

{
    int dev_id = omp_get_thread_num();

    cudaSetDevice( dev_id );
}
```

# MULTI THREADED MULTI GPU PROGRAMMING

## Using OpenMP and P2P Mappings

```
while ( l2_norm > tol && iter < iter_max ) {  
    cudaMemsetAsync(l2_norm_d, 0 , sizeof(real), compute_stream );  
    #pragma omp barrier  
    cudaStreamWaitEvent( compute_stream, compute_done[iter%2][top], 0 );  
    cudaStreamWaitEvent( compute_stream, compute_done[iter%2][bottom], 0  
); jacobi_kernel<<<dim_grid,dim_block,0,compute_stream>>>(  
    a_new[dev_id], a, l2_norm_d, iy_start, iy_end[dev_id], nx,  
    a_new[top], iy_end[top], a_new[bottom], 0 );  
    cudaEventRecord( compute_done[(iter+1)%2][dev_id], compute_stream );  
    cudaMemcpyAsync(l2_norm,l2_norm_d,sizeof(real),cudaMemcpyDeviceToHost,compute_stream);  
    // l2_norm reduction btw threads skipped ...  
    #pragma omp barrier  
    std::swap(a_new[dev_id],a); iter++;  
}
```

# MULTI THREADED MULTI GPU PROGRAMMING

## Using OpenMP and P2P Mappings

```
__global__ void jacobi_kernel( ... ) {
    for (int iy = bIdx.y*bDim.y+tIdx.y + iy_start; iy < iy_end; iy += bDim.y*gDim.y) {
        for (int ix = bIdx.x*bDim.x+tIdx.x + 1; ix < (nx-1); ix += bDim.x*gDim.x)
            const real new_val = 0.25 * ( a[ iy * nx + ix + 1 ] + a[ iy * nx + ix - 1 ]
                + a[ (iy+1)*nx + ix ] + a[ (iy-1)*nx + ix ] );
        if ( iy_start == iy ) { a_new_top[ top_iy * nx + ix ] = new_val; }
        if ( (iy_end - 1) == iy ) { a_new_bottom[ bottom_iy*nx + ix ] = new_val; }
        atomicAdd( &residue_l2_norm, residue * residue );
    }
}
```

# MESSAGE PASSING INTERFACE - MPI

Standard to exchange data between processes via messages

Defines API to exchanges messages

Point to Point: e.g. `MPI_Send`, `MPI_Recv`

Collectives: e.g. `MPI_Reduce`

Multiple implementations (open source and commercial)

Bindings for C/C++, Fortran, Python, ...

E.g. MPICH, OpenMPI, MVAPICH, IBM Platform MPI, Cray MPT, ...

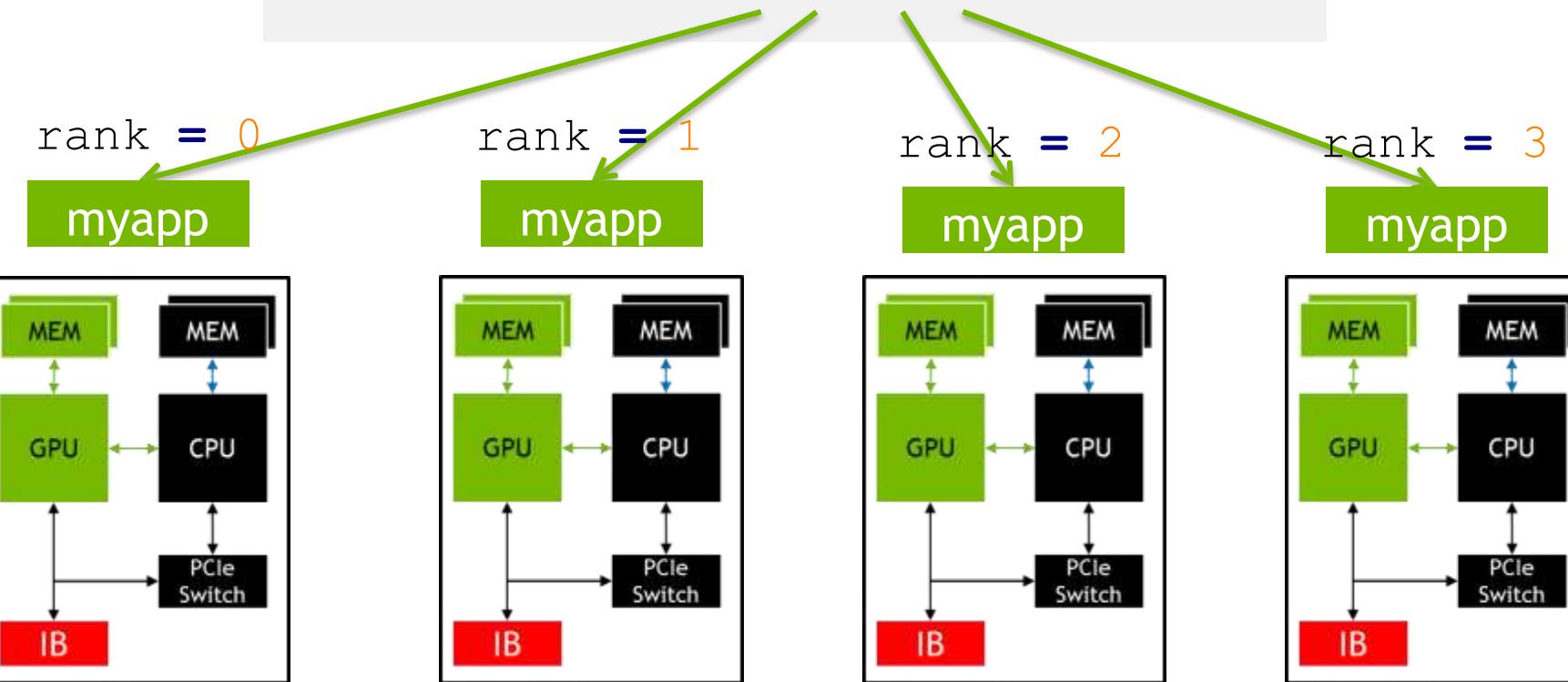
# MPI - SKELETON

```
#include <mpi.h>
int main(int argc, char *argv[]) {
    int rank, size;
    /* Initialize the MPI library */
    MPI_Init(&argc, &argv);
    /* Determine the calling process rank and total number of ranks */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    /* Call MPI routines like MPI_Send, MPI_Recv, ... */
    ...
    /* Shutdown MPI library */
    MPI_Finalize();
    return 0;
}
```

# MP

## Compiling and Launching

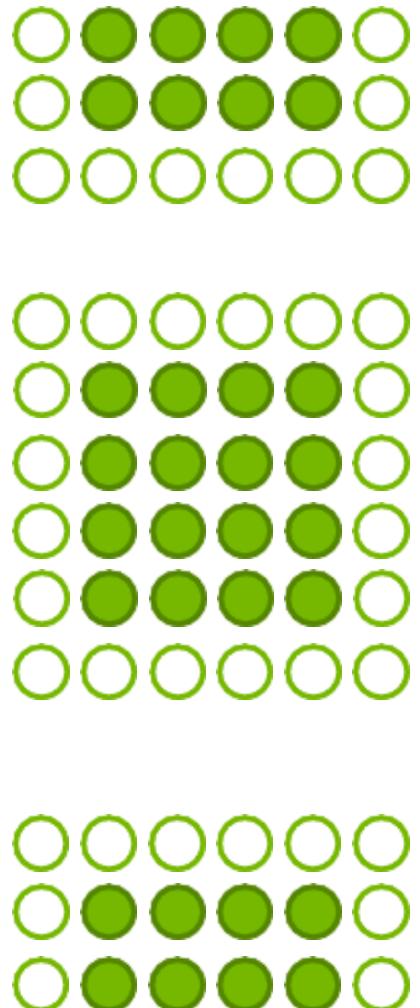
```
$ mpicc -o myapp myapp.c  
$ mpirun -np 4 ./myapp <args>
```



# EXAMPLE JACOBI

## Top/Bottom Halo

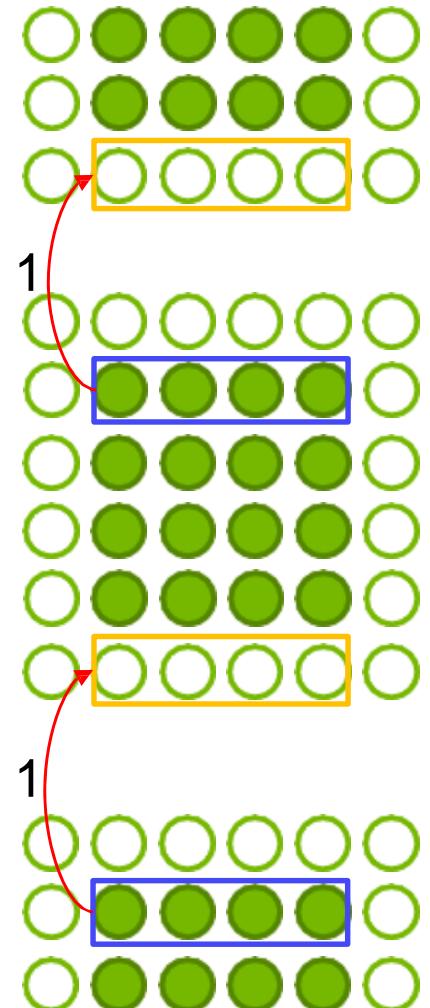
```
MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top , 0,  
             a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



# EXAMPLE JACOBI

## Top/Bottom Halo

```
MPI_Sendrecv(a new+iy start*nx, nx, MPI_FLOAT, top , 0,  
             a new+(iy end*nx), nx, MPI_FLOAT, bottom, 0,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

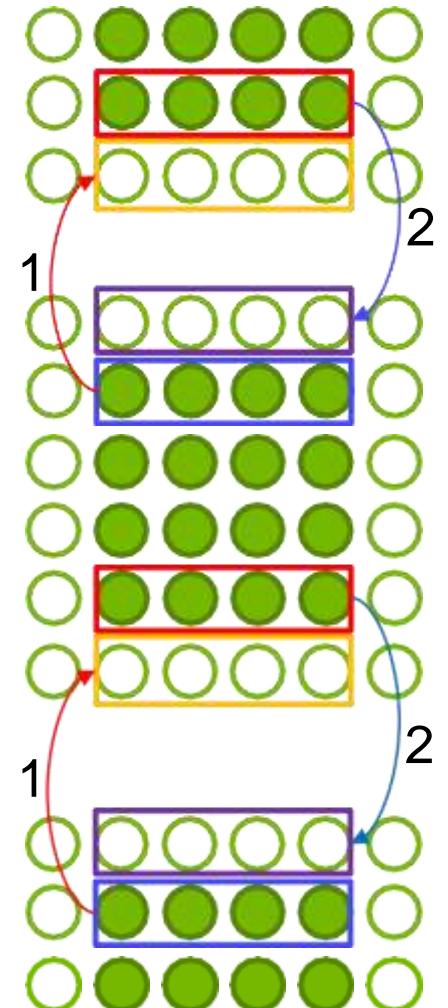


# EXAMPLE JACOBI

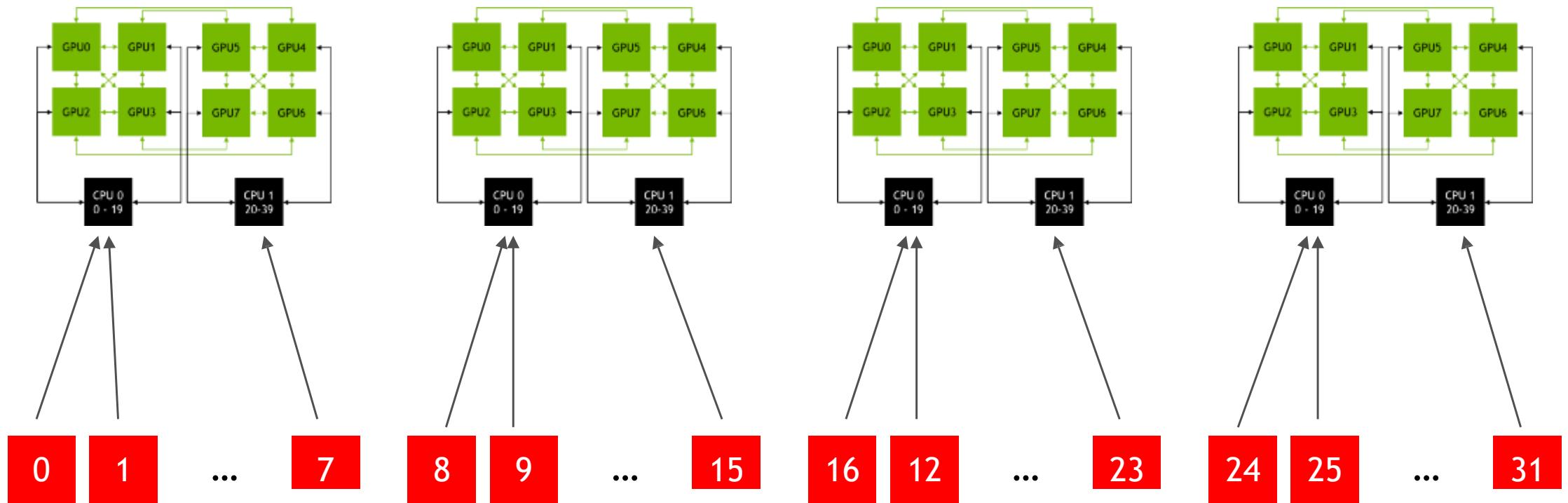
## Top/Bottom Halo

```
MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top , 0,
              a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(a_new+(iy_end-1)*nx, nx, MPI_FLOAT, bottom, 0,
              a_new, nx, MPI_FLOAT, top, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
```



# HANDLING MULTIPLE MULTI GPU NODES

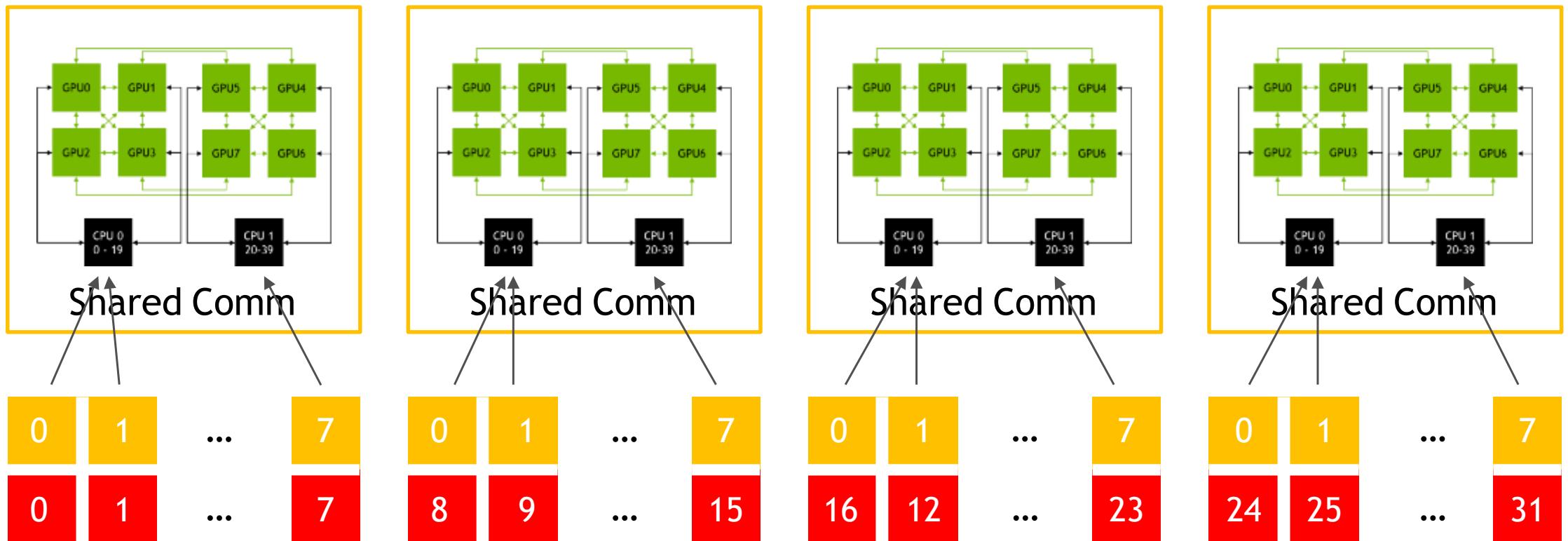


# HANDLING MULTIPLE MULTI GPU NODES

## How to determine the local rank? - MPI-3

```
MPI_Comm local_comm;  
  
MPI_Info info;  
  
MPI_Info_create(&info);  
  
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, rank, info, &local_comm);  
int local_rank = -1;  
  
MPI_Comm_rank(local_comm, &local_rank);  
  
MPI_Comm_free(&local_comm);  
  
MPI_Info_free(&info);
```

# HANDLING MULTIPLE MULTI GPU NODES



# HANDLING MULTIPLE MULTI GPU NODES

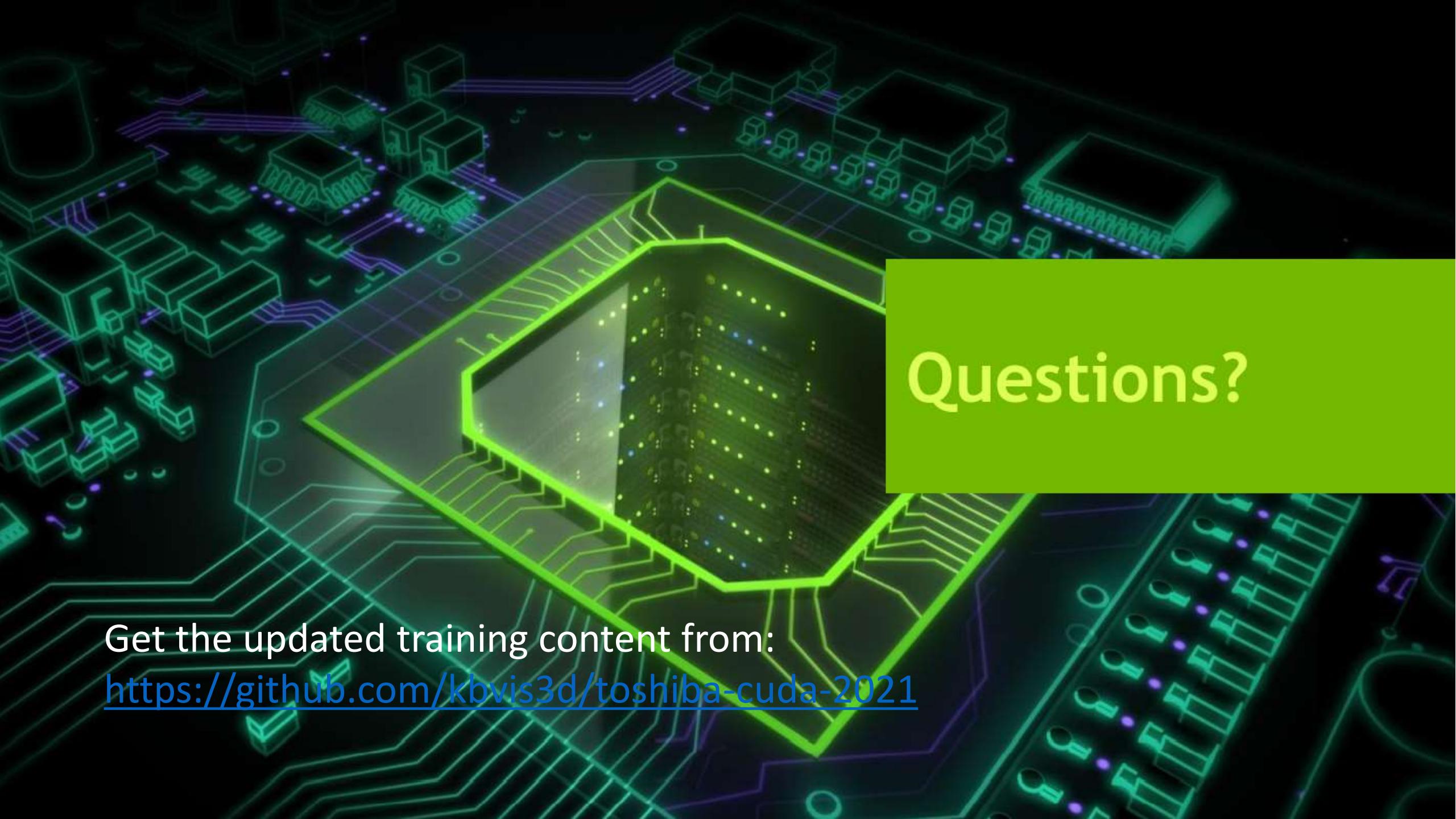
## GPU-affinity

Use local rank:

```
int local_rank = -1;  
MPI_Comm_rank(local_comm,&local_rank);  
  
int num_devices = 0;  
cudaGetDeviceCount (&num_devices);  
  
cudaSetDevice (local_rank % num_devices);
```

# COMMUNICATION + COMPUTATION OVERLAP

```
launch_jacobi_kernel( a_new, a, l2_norm_d, iy_start, (iy_start+1), nx, push_top_stream );
launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_end-1), iy_end, nx, push_bottom_stream );
launch_jacobi_kernel( a_new, a, l2_norm_d, (iy_start+1), (iy_end-1), nx, compute_stream
const int top = rank > 0 ? rank - 1 : (size+1);
const int bottom = (rank+1)%size;
cudaStreamSynchronize( push_top_stream );
MPI_Sendrecv( a_new+iy_start*nx, nx, MPI_REAL_TYPE, top, 0,
              a_new+(iy_end*nx), nx, MPI_REAL_TYPE, bottom, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE );
cudaStreamSynchronize( push_bottom_stream );
MPI_Sendrecv( a_new+(iy_end-1)*nx, nx, MPI_REAL_TYPE, bottom, 0,
              a_new, nx, MPI_REAL_TYPE, top, 0, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE );
```



# Questions?

Get the updated training content from:

<https://github.com/kbvis3d/toshiba-cuda-2021>