



# Introduction to Accelerated Computing using CUDA

2021.08.04

Karthik Bala

KBVIS Software Pty. Ltd., Brisbane, Australia

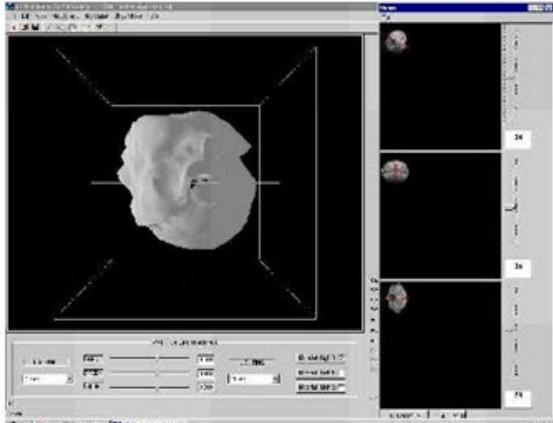
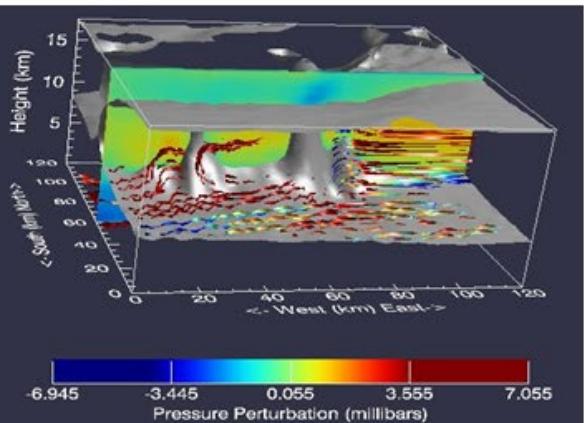
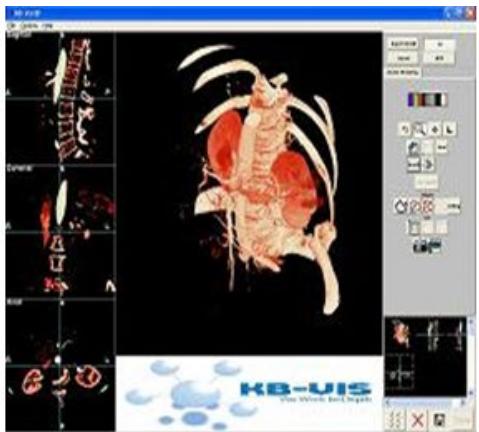
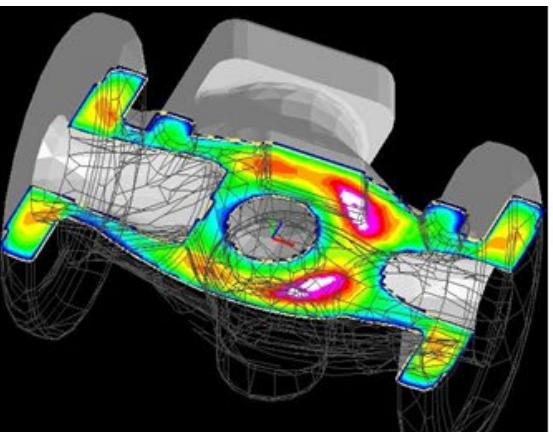
<http://kbvis.com>

[3d@kbvis.com](mailto:3d@kbvis.com)

For internal circulation only; contains copyrighted material.



# About KBVIS



## Clients



reflexion

Deswik

Rapiscan<sup>®</sup>  
systems

An OSI Systems Company



BARCO



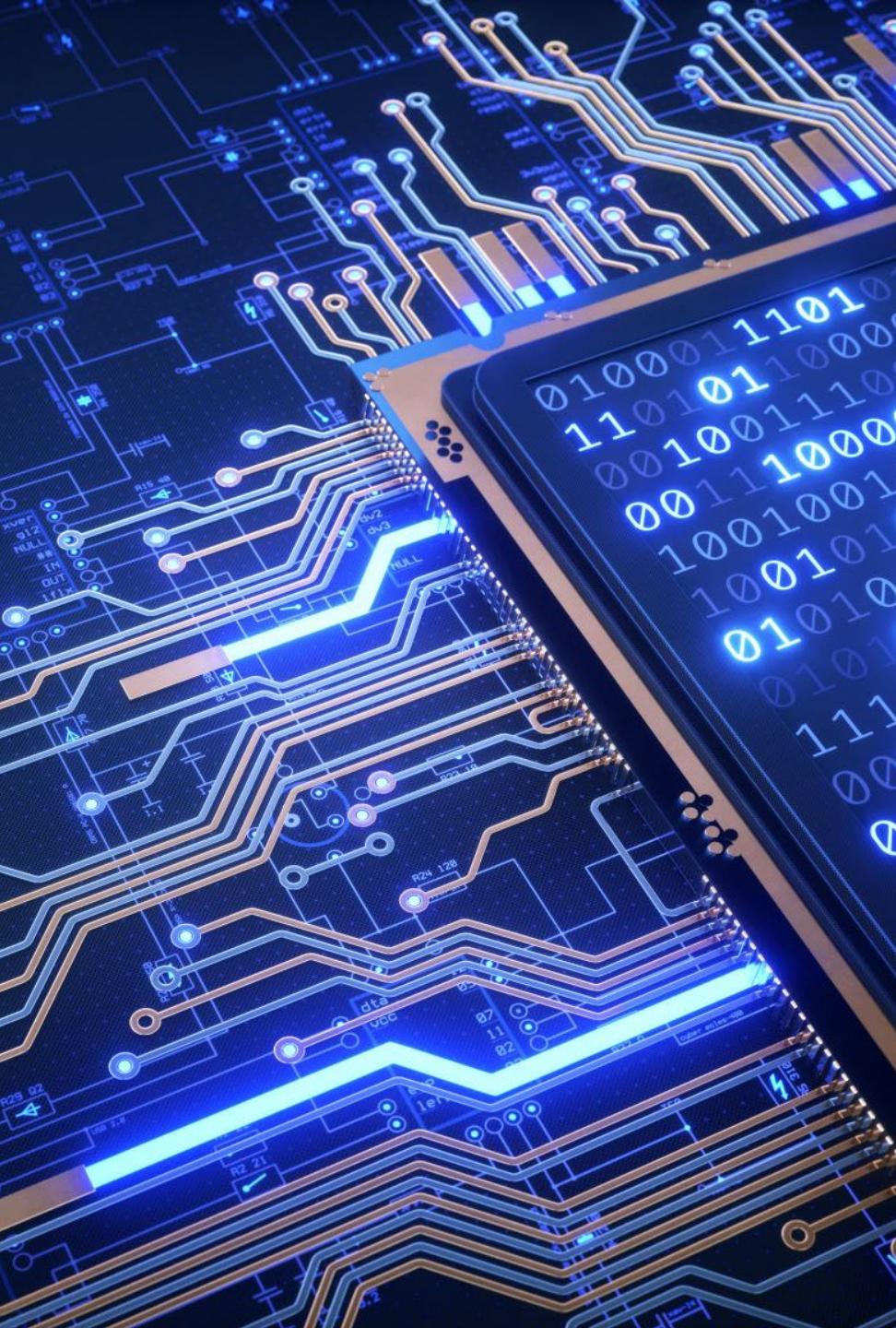
Telerad Tech<sup>®</sup>  
Smarter Healthcare On Demand

Tektronix<sup>®</sup>

i SPEC INDIA

Honeywell

DSP GROUP



# 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

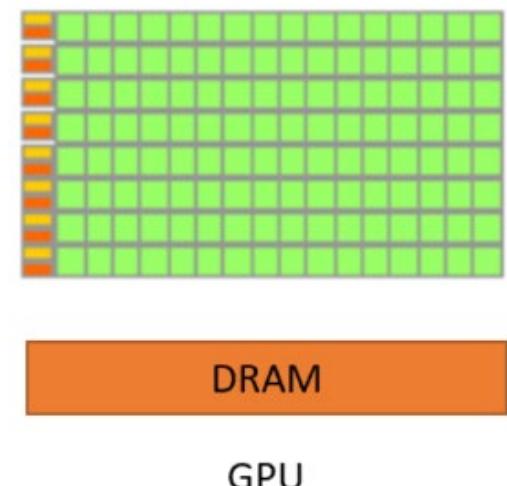
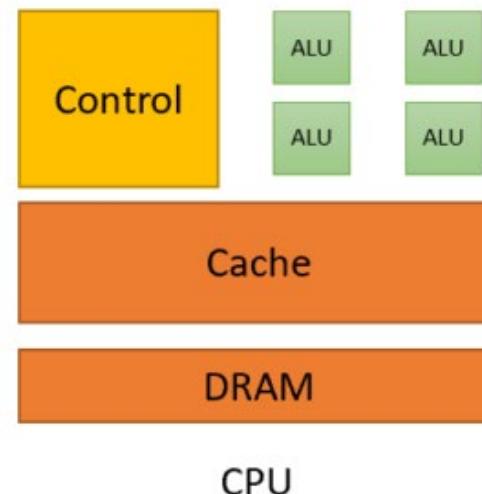
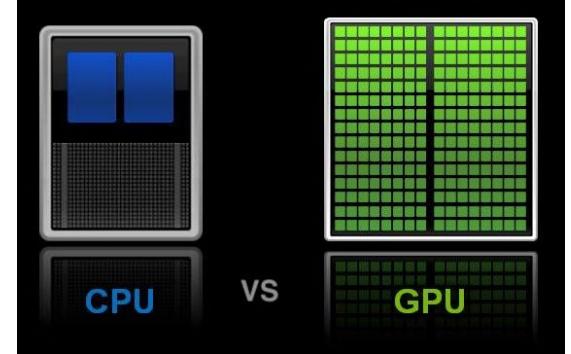
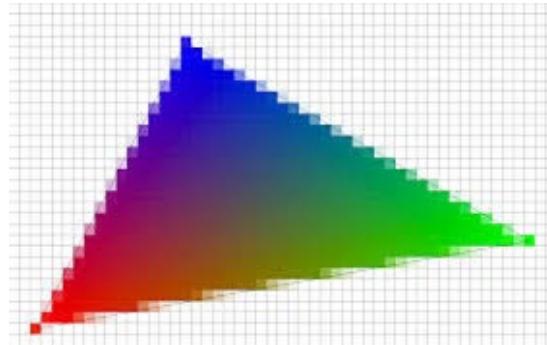
# Agenda (tentative)

- 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

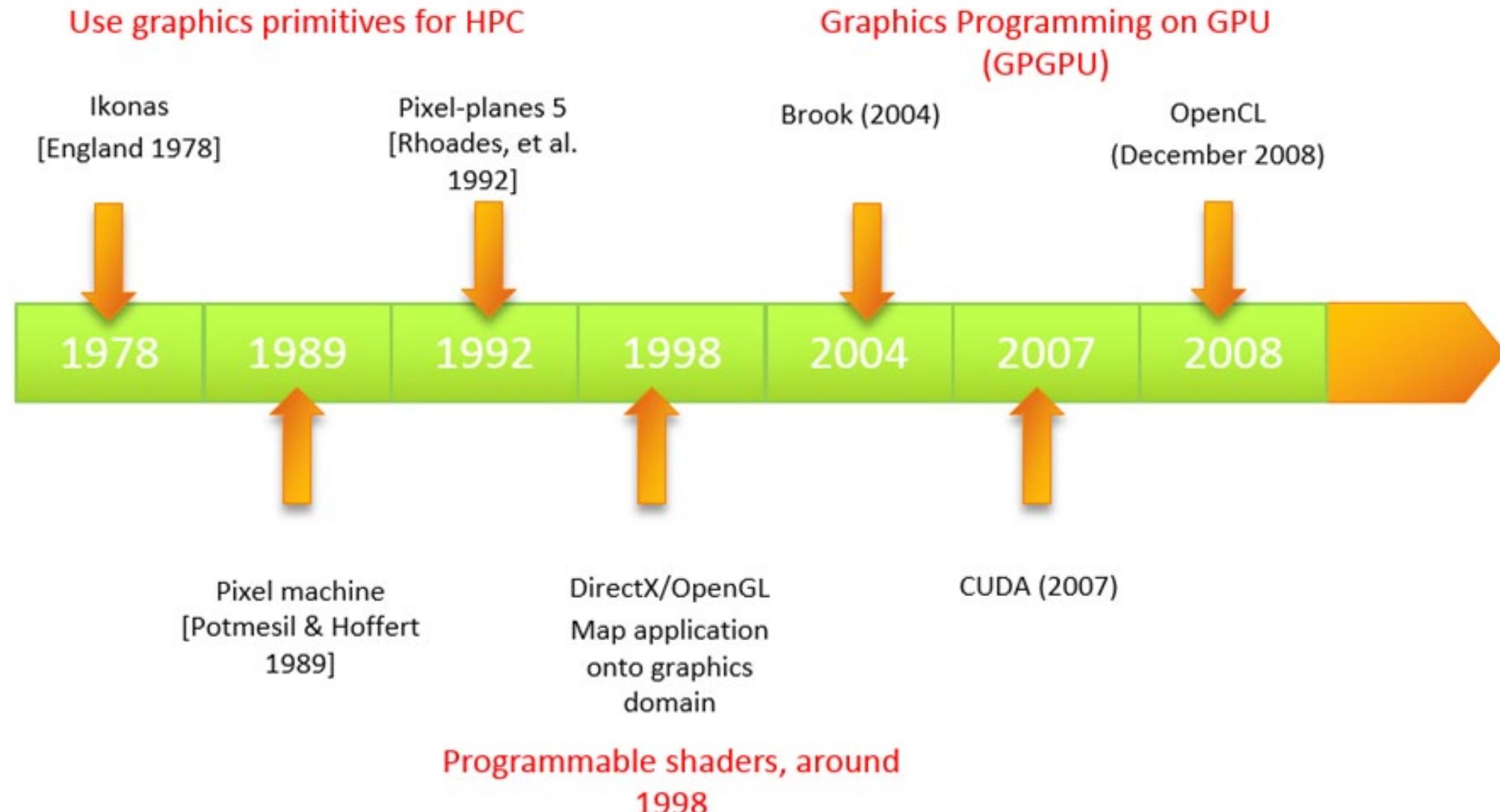
Feel free to suggest changes/additions!

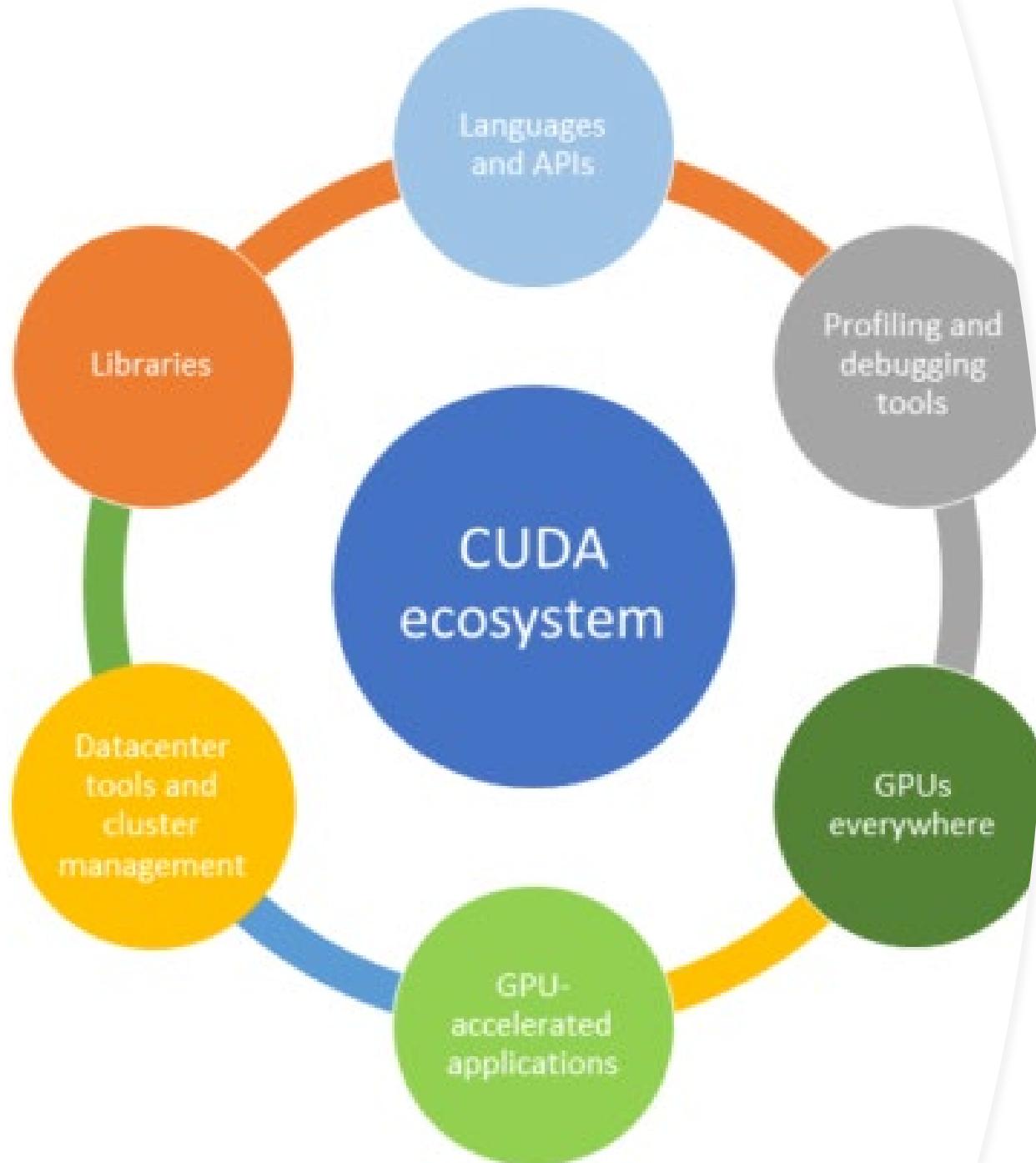
# Dawn of GPGPU

- 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



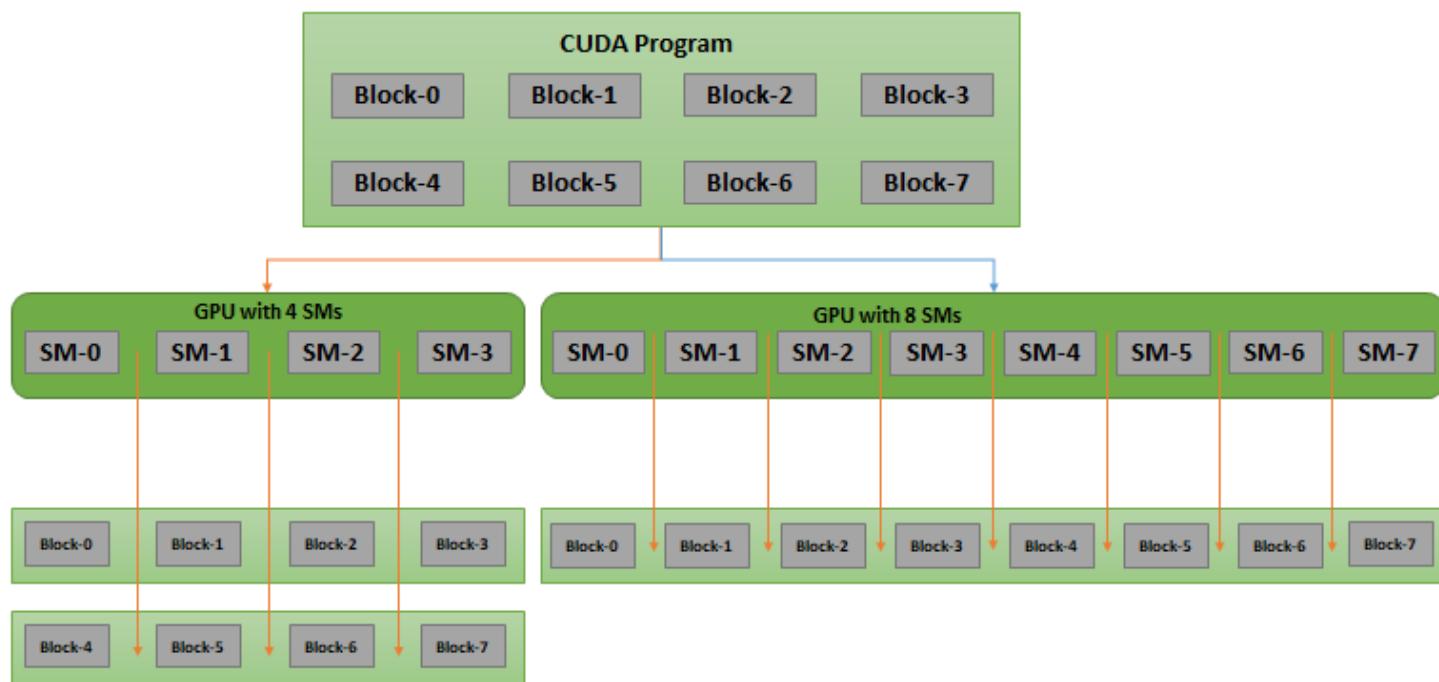


## CUDA Ecosystem

---

- 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

host code

```
#include <iostream>
#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 + blockDim.x * blockIdx.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 *x, int n) {
    fill(x, 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(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc(&d_in, size);
    cudaMalloc(&d_out, size);

    // 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;
}
```

parallel function

serial function

serial code

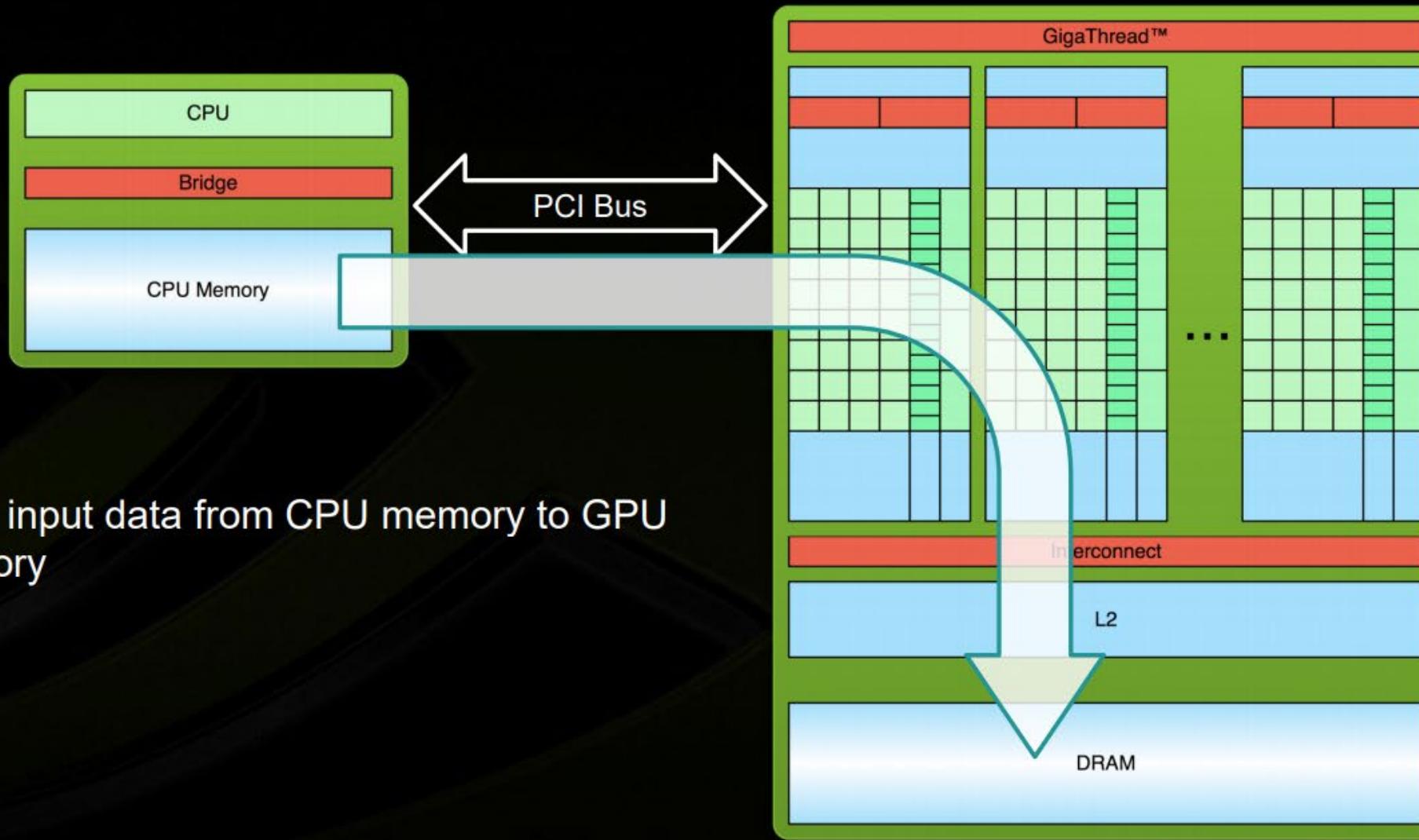
parallel code

serial code

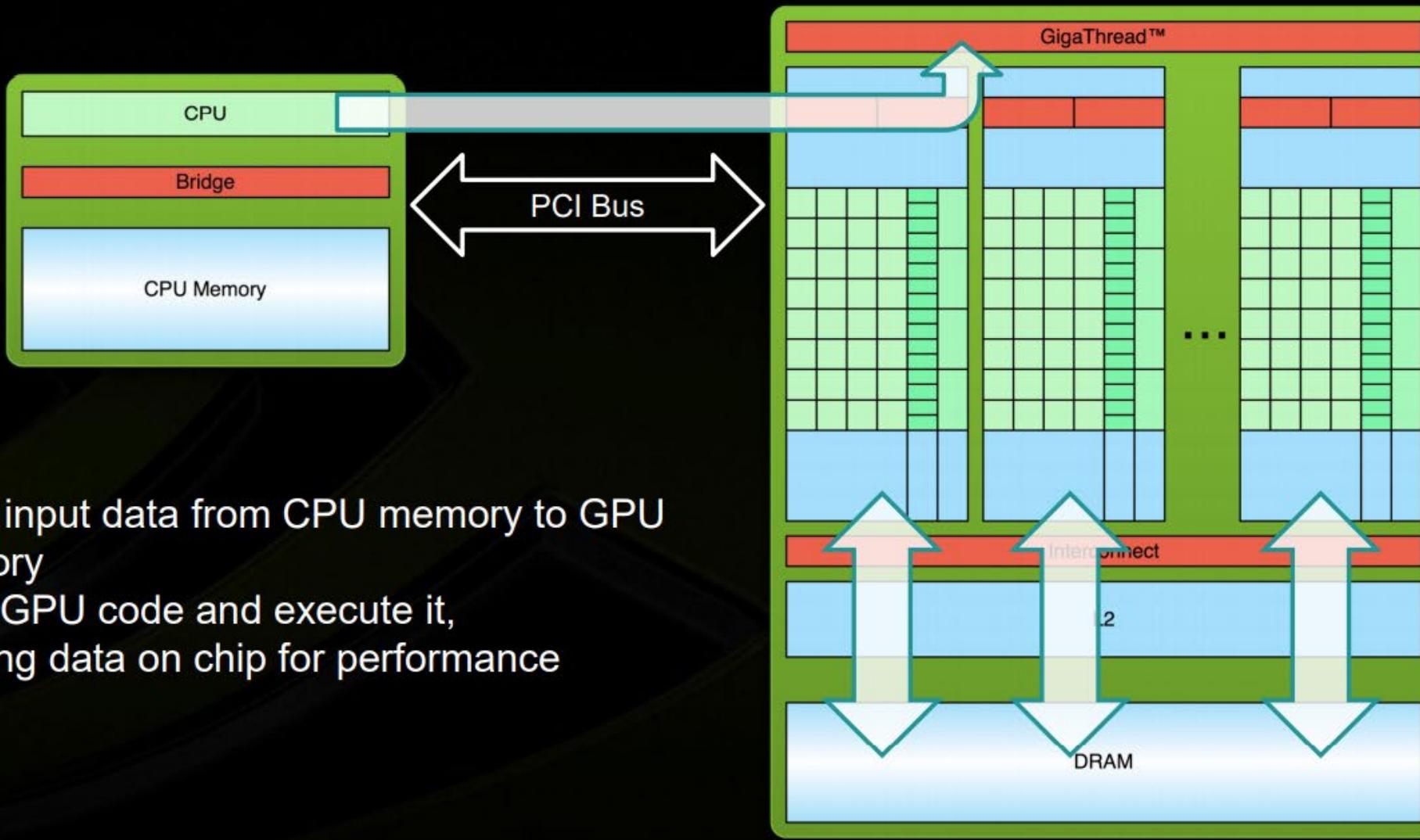


# Simple Processing Flow

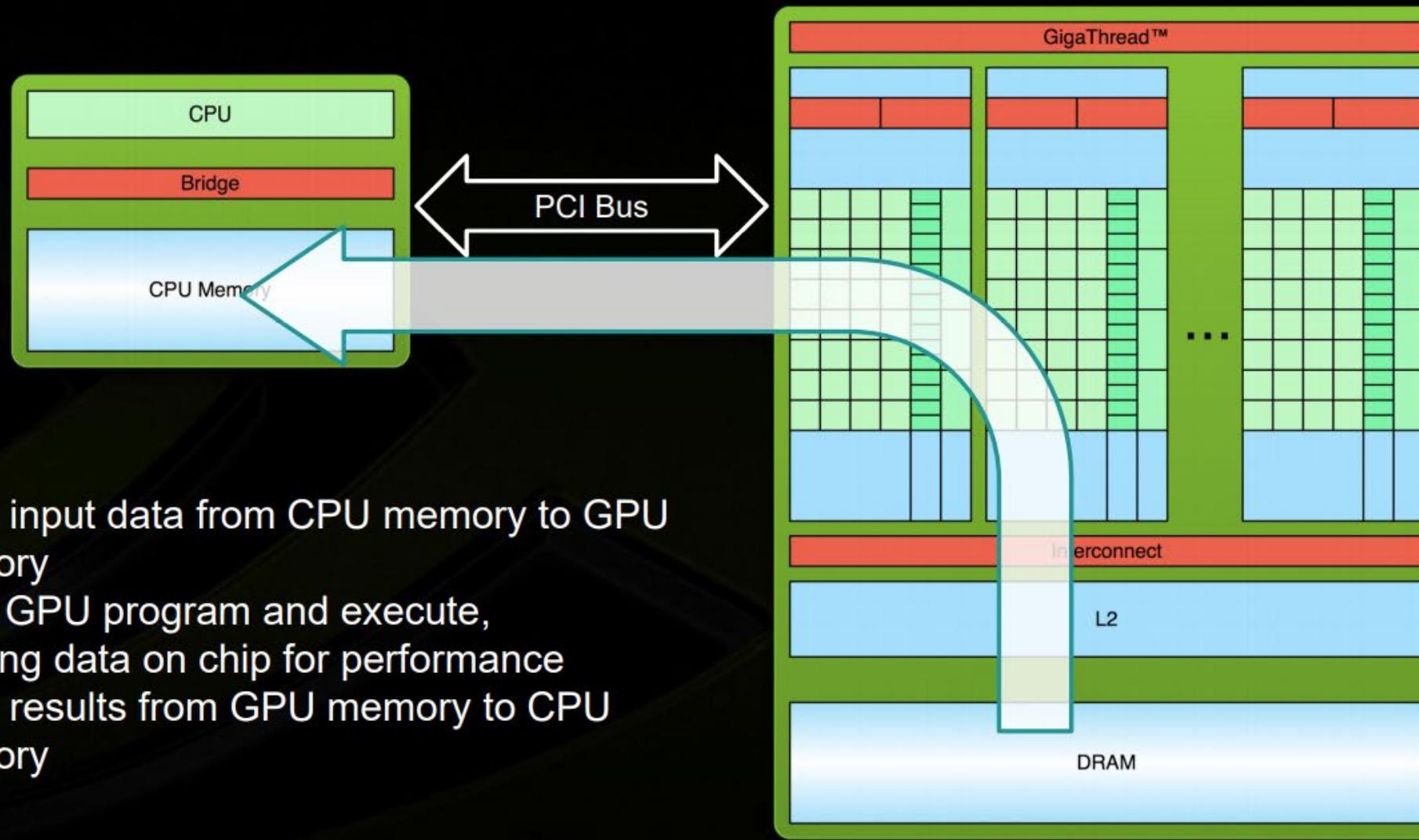
1. Copy input data from CPU memory to GPU memory



# 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:

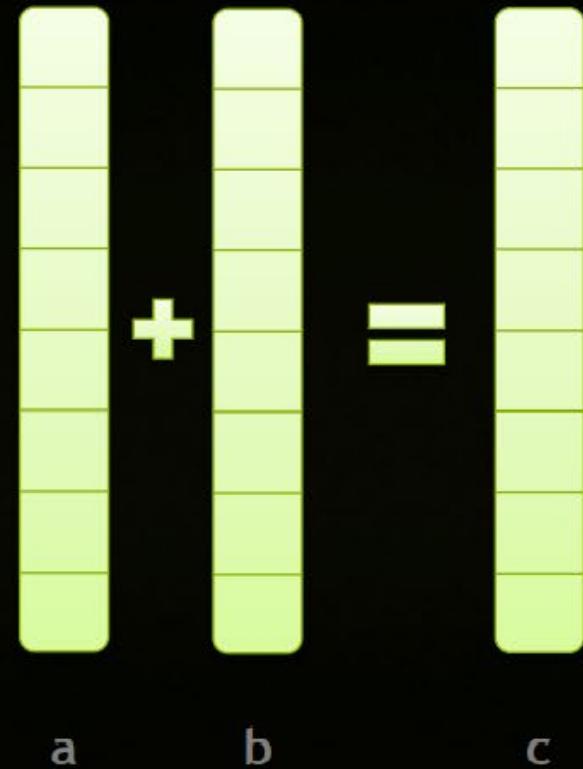
```
$ nvcc hello.cu  
$ a.out  
Hello World!  
$
```

- `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;
}
```

## CONCEPTS

# RUNNING IN PARALLEL

- 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



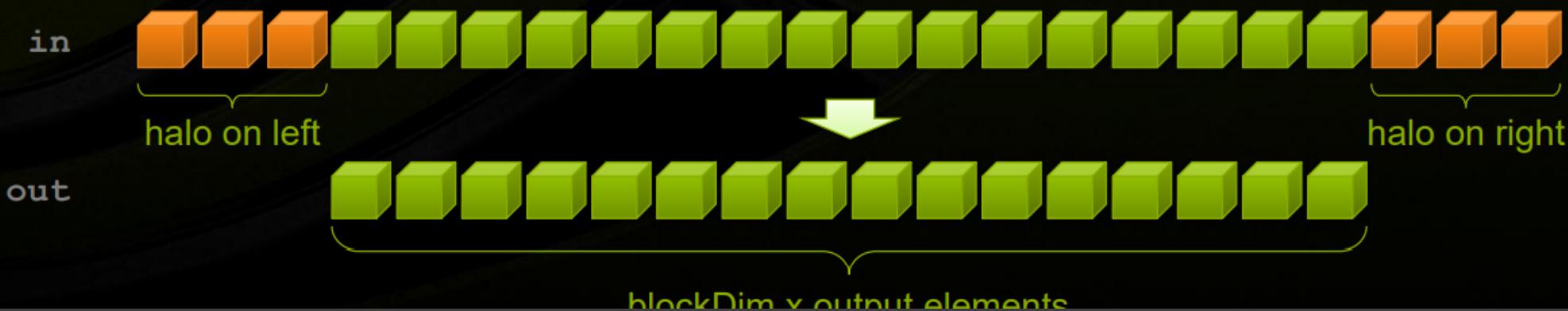


# Sharing Data Between Threads

- 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 `(blockDim.x + 2 * radius)` input elements from global memory to shared memory
  - Compute `blockDim.x` output elements
  - Write `blockDim.x` output elements to global memory
- Each block needs a halo of `radius` elements at each boundary

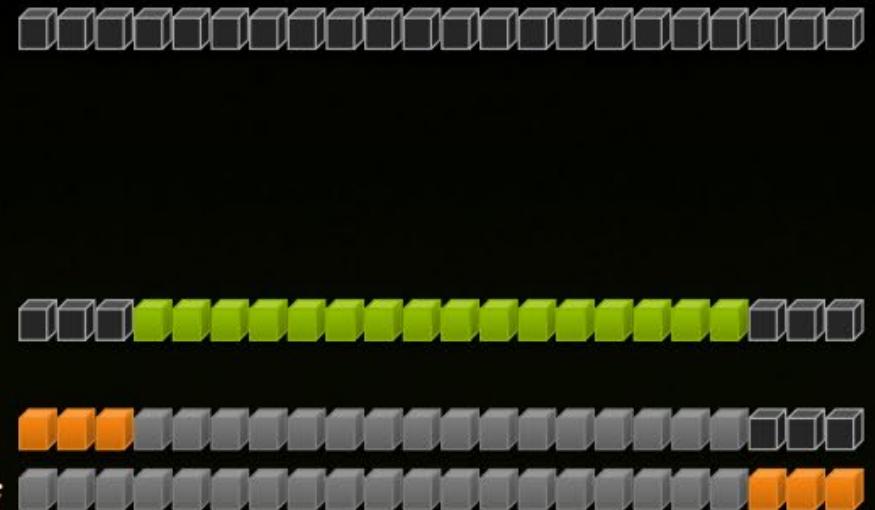


# Stencil Kernel



```
__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)

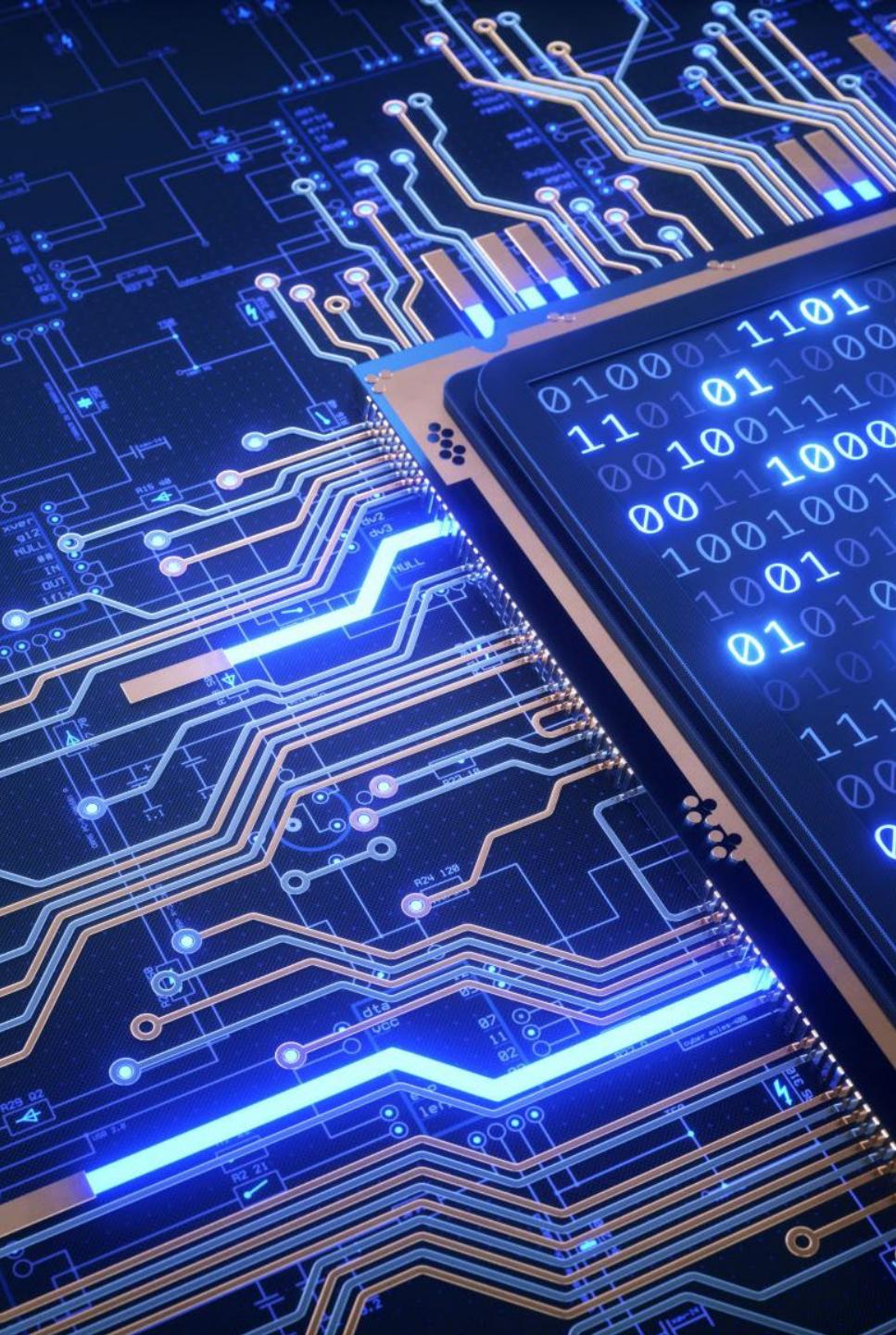
- Launching parallel threads
  - Launch  $N$  blocks with  $M$  threads per block with `kernel<<<N, M>>>(...);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
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

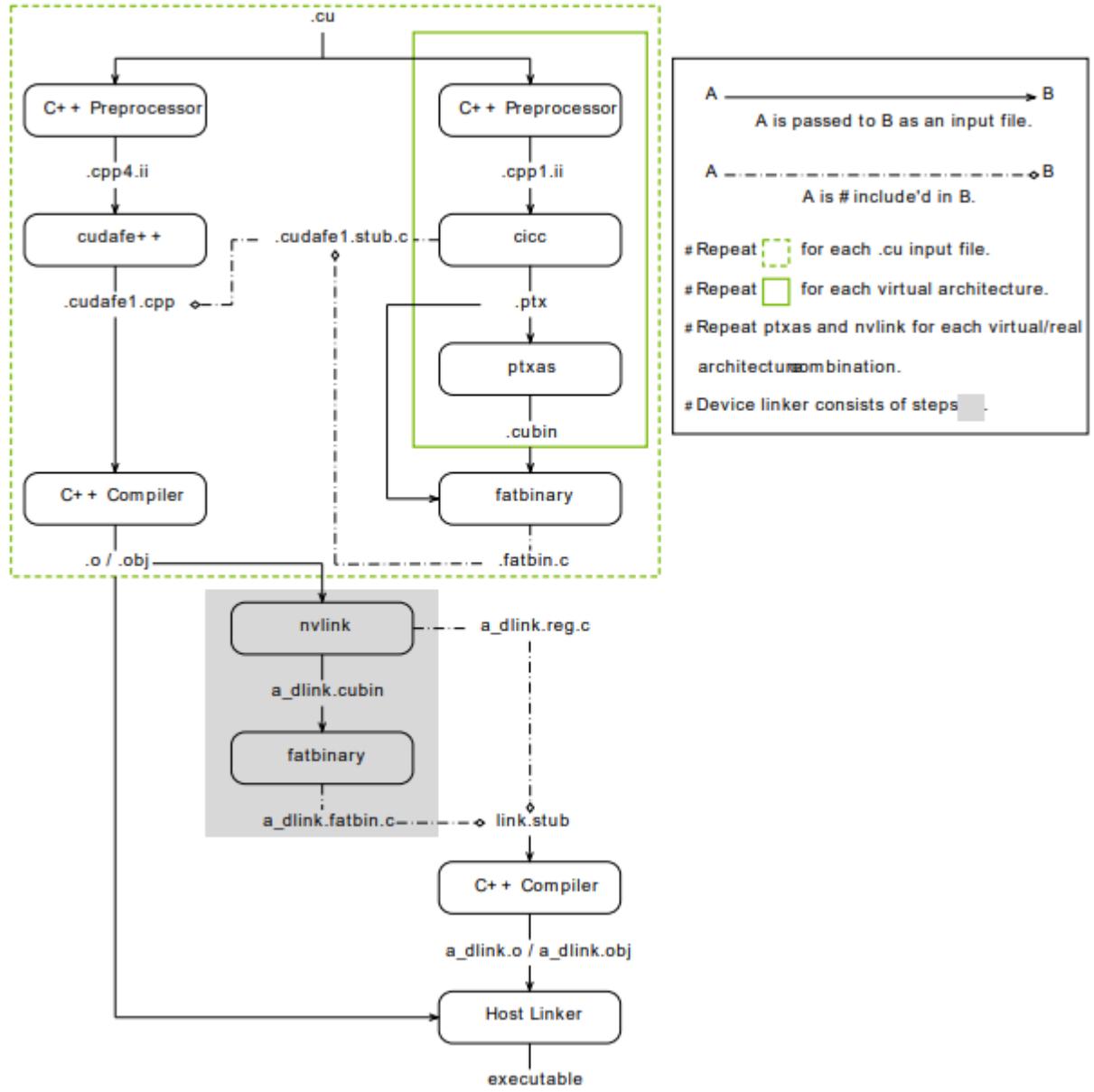


# DAY 2

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

# 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



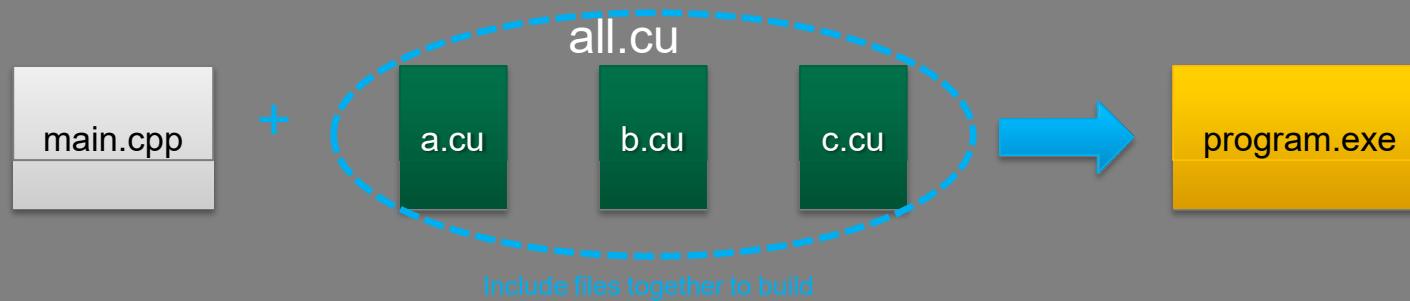
# Supported Input File Types

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

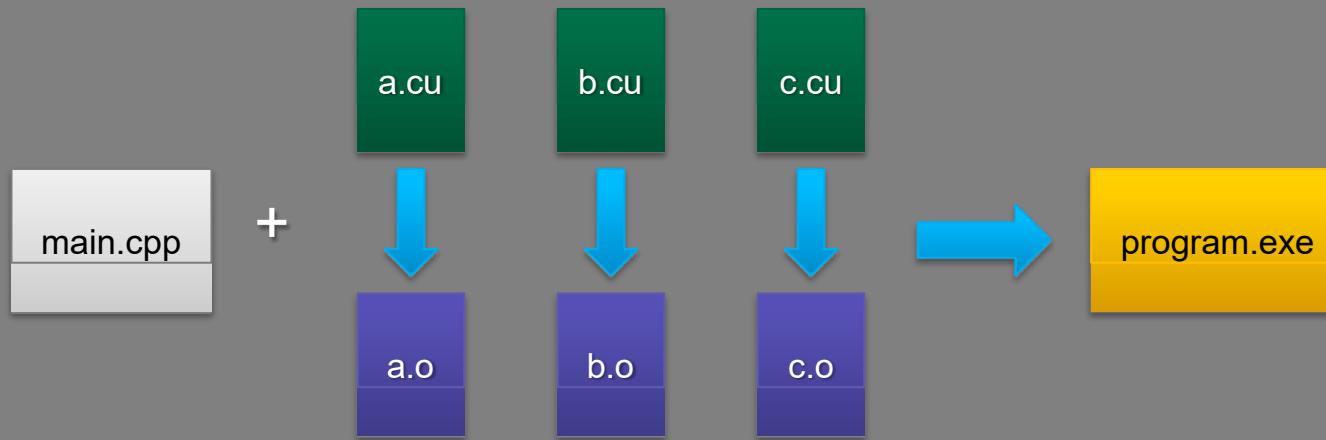
Phase	nvcc Option		Default Output File Name
	Long Name	Short Name	
CUDA compilation to C/C++ source file	<code>--cuda</code>	<code>-cuda</code>	.cpp.ii appended to source file name, as in <code>x.cu.cpp.ii</code> . This output file can be compiled by the host compiler that was used by <code>nvcc</code> to preprocess the <code>.cu</code> file.
C/C++ preprocessing	<code>--preprocess</code>	<code>-E</code>	<result on standard output>
C/C++ compilation to object file	<code>--compile</code>	<code>-c</code>	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	<code>--cubin</code>	<code>-cubin</code>	Source file name with suffix replaced by <code>cubin</code>
Cubin generation from PTX intermediate files.	<code>--cubin</code>	<code>-cubin</code>	Source file name with suffix replaced by <code>cubin</code>
PTX generation from CUDA source files	<code>--ptx</code>	<code>-ptx</code>	Source file name with suffix replaced by <code>ptx</code>
Fatbinary generation from source, PTX or cubin files	<code>--fatbin</code>	<code>-fatbin</code>	Source file name with suffix replaced by <code>fatbin</code>
Linking relocatable device code.	<code>--device-link</code>	<code>-dlink</code>	<code>a_dlink.obj</code> on Windows or <code>a_dlink.o</code> on other platforms
Cubin generation from linked relocatable device code.	<code>--device-link --cubin</code>	<code>-dlink -cubin</code>	<code>a_dlink.cubin</code>
Fatbinary generation from linked relocatable device code	<code>--device-link --fatbin</code>	<code>-dlink -fatbin</code>	<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	<code>--lib</code>	<code>-lib</code>	<code>a.lib</code> on Windows or <code>a.a</code> on other platforms
make dependency generation	<code>--generate-dependencies</code>	<code>-M</code>	<result on standard output>
make dependency generation without headers in system paths.	<code>--generate-nonsystem-dependencies</code>	<code>-MM</code>	<result on standard output>
Running an executable	<code>--run</code>	<code>-run</code>	

# 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



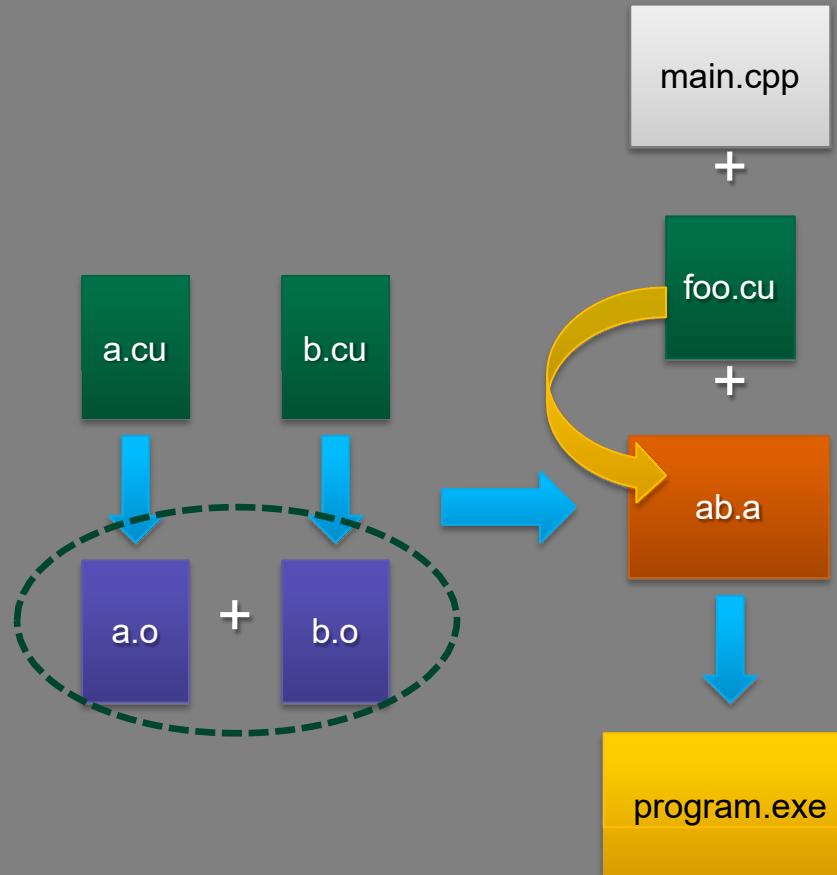
Separate compilation allows building independent object files

CUDA 5 can link multiple object files into one program

# Benefits of Separate Compilation

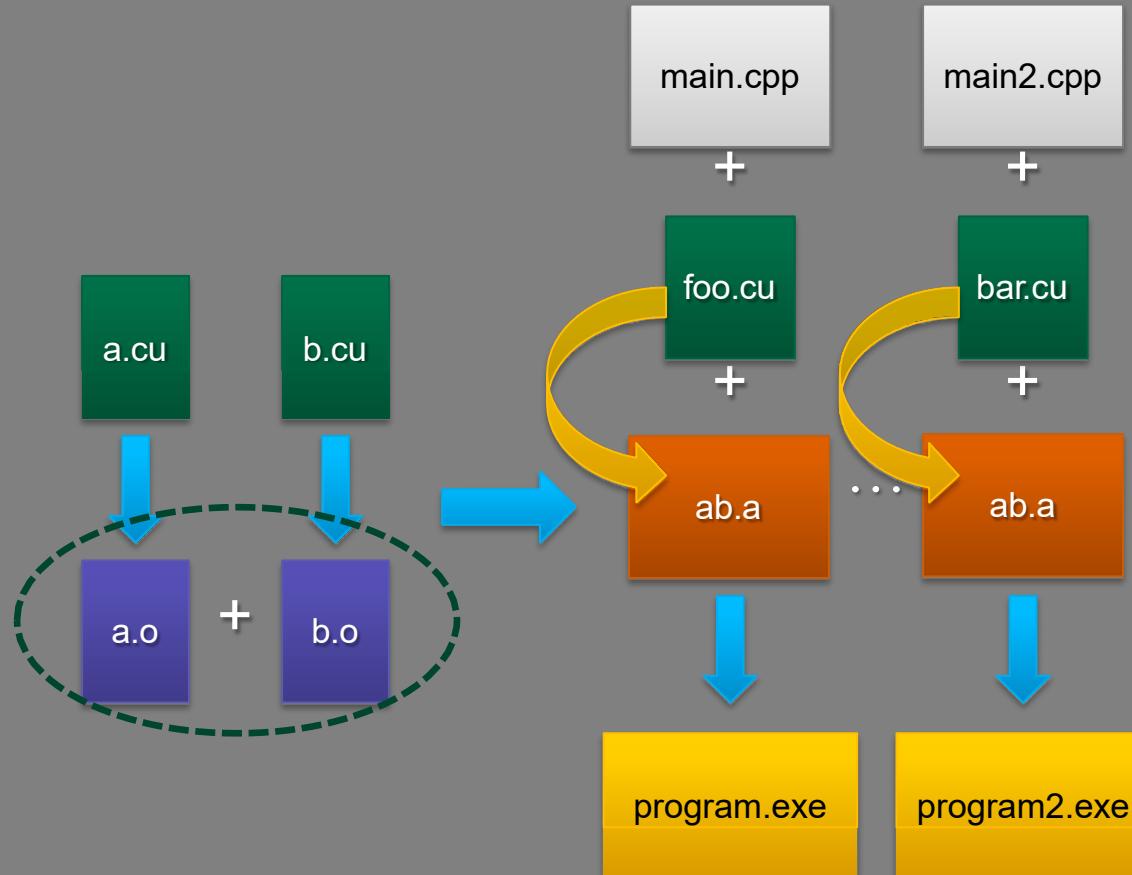
- Eases porting code
  - no longer have to include files together
  - “extern” attribute is respected
- Incremental compilation reduces build time
  - e.g. 47000 line app used to take 50 seconds to build, now when split into multiple files takes 4 seconds to build if only one file changed
- Can create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



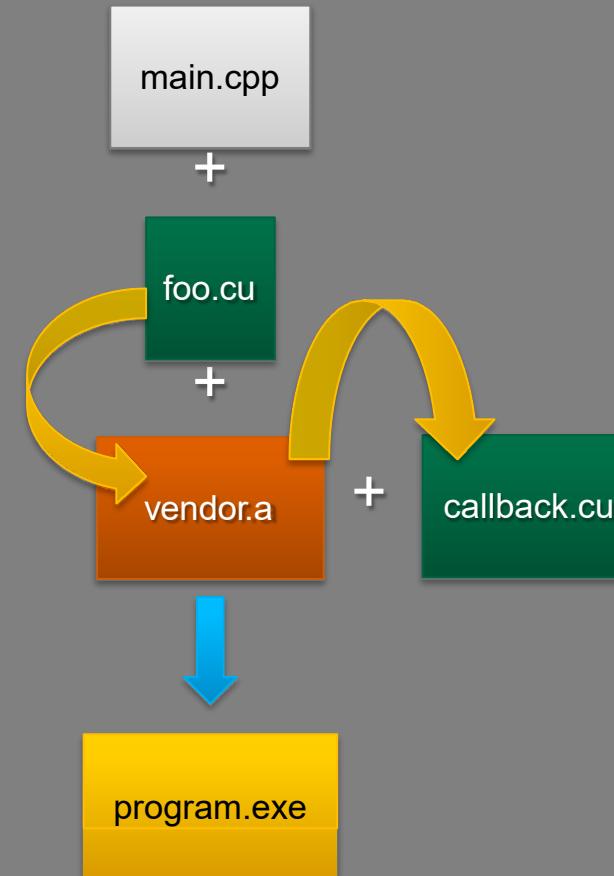
Can combine object files into static libraries

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



# Separate Compilation Features

- SM\_2x and above (Fermi & Kepler, no support for sm\_1x)
- All platforms (Linux, Windows, and MacOS)
- All CUDA features
- Optimized and Debug (-G) compilations
- Support both previous whole-program compilation and new separate compilation.
  - Default is whole-program compilation, have to opt in to separate compilation.

# 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/pdf/CUDA\\_Compiler\\_Driver\\_NVCC.pdf](https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf)

# 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

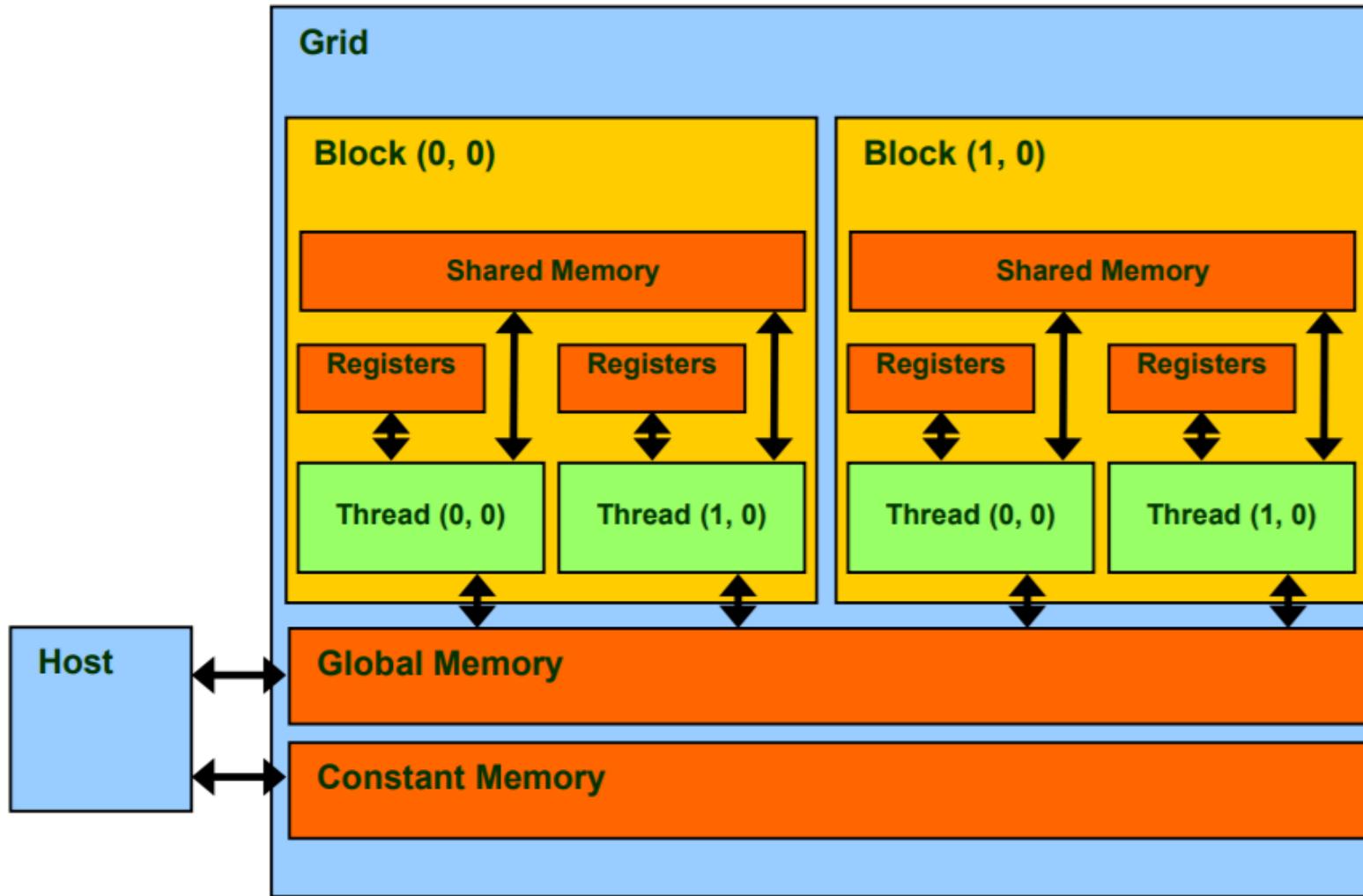
```
// ensure all reads are complete
__syncthreads();

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

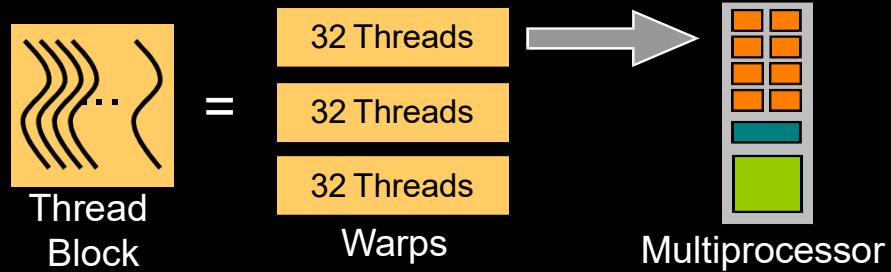


# Use Parallelism Efficiently

- 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

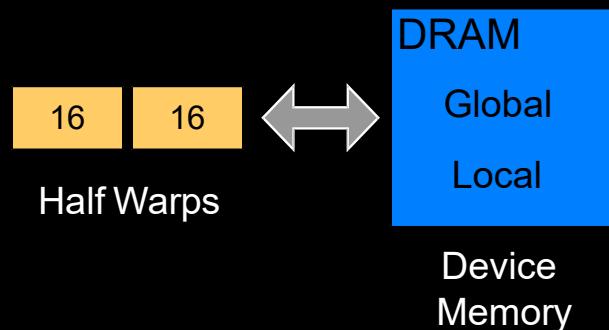


# Warps and Half Warps



A thread block consists of 32-thread warps

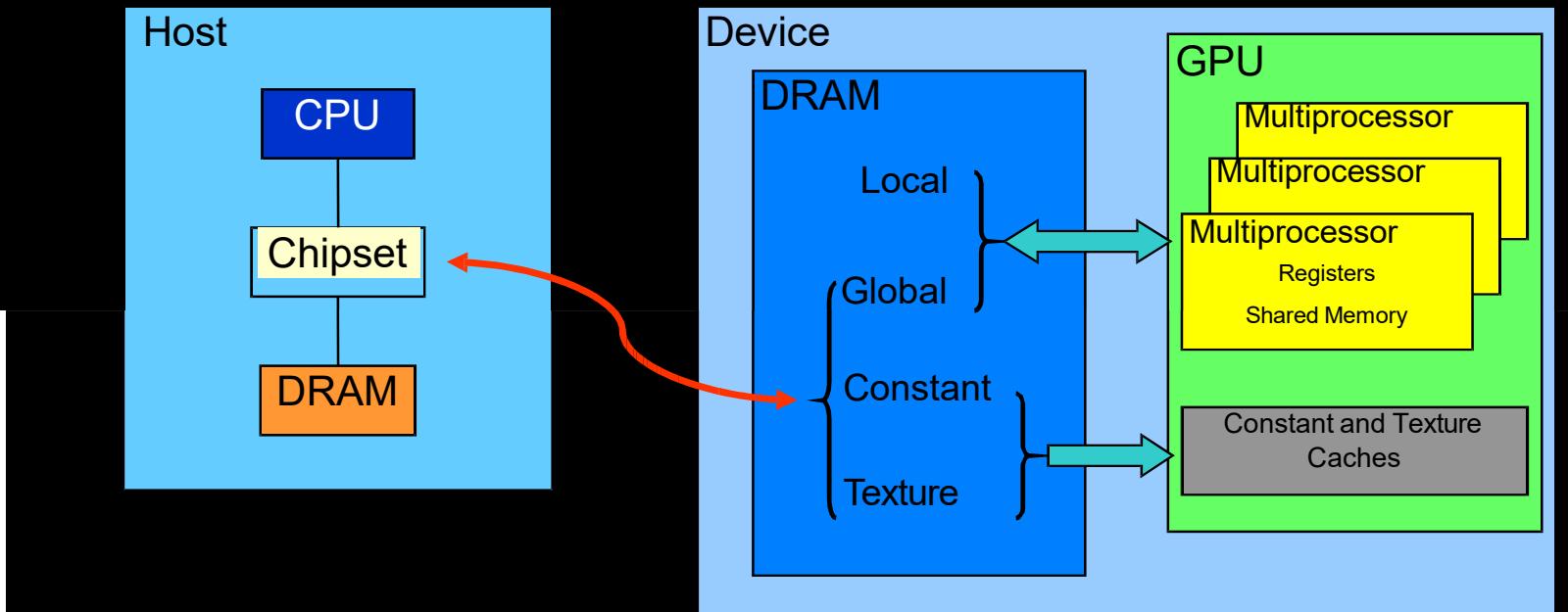
A warp is executed physically in parallel (SIMD) on a multiprocessor



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

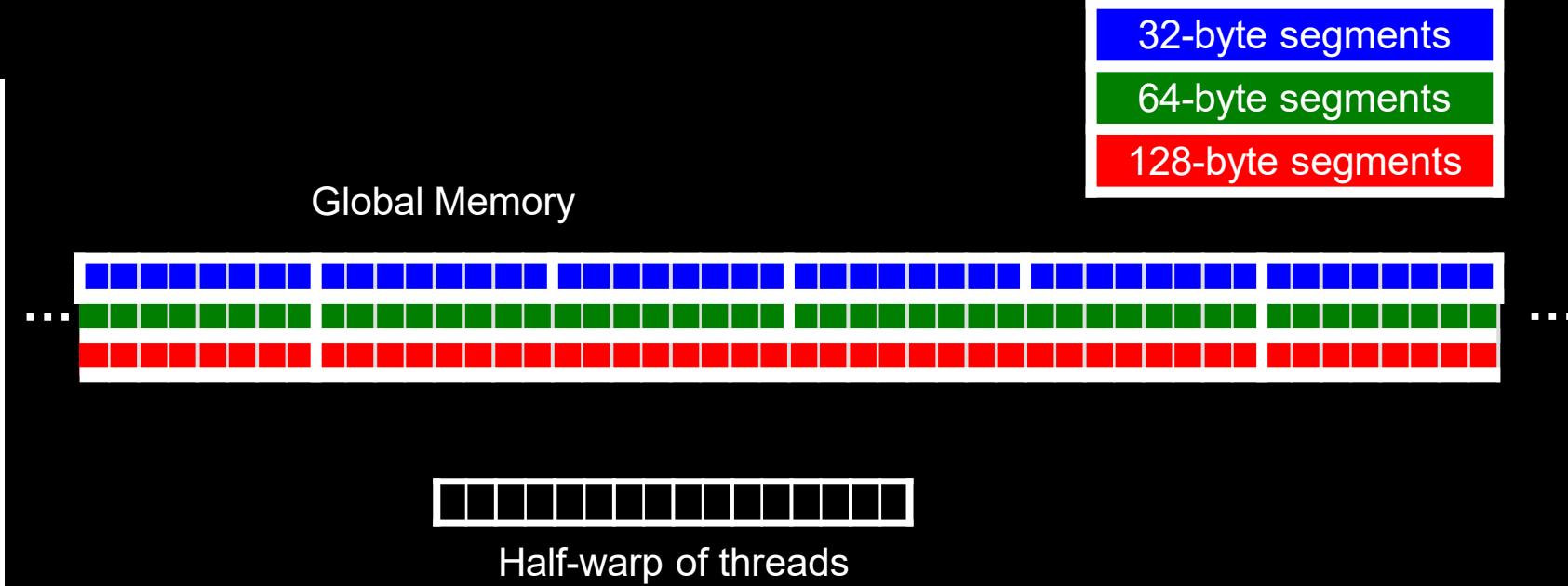


# Memory Architecture



# Coalescing

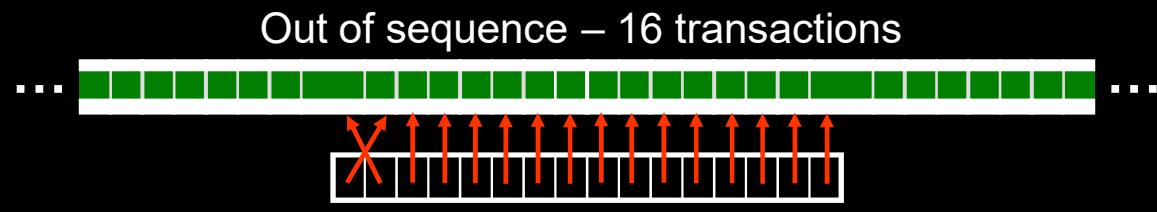
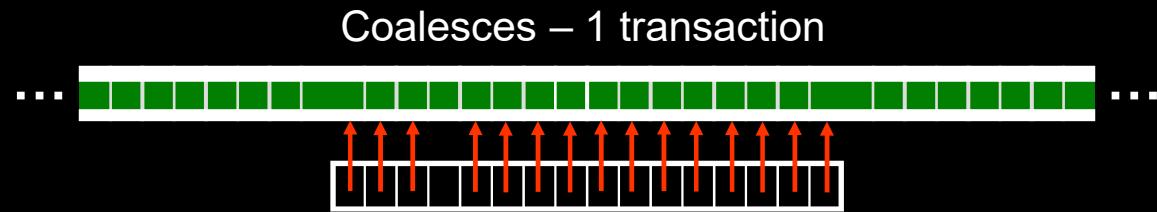
- Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met
- Depends on compute capability
  - 1.0 and 1.1 have stricter access requirements
- **Float (32-bit) data example:**



# 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

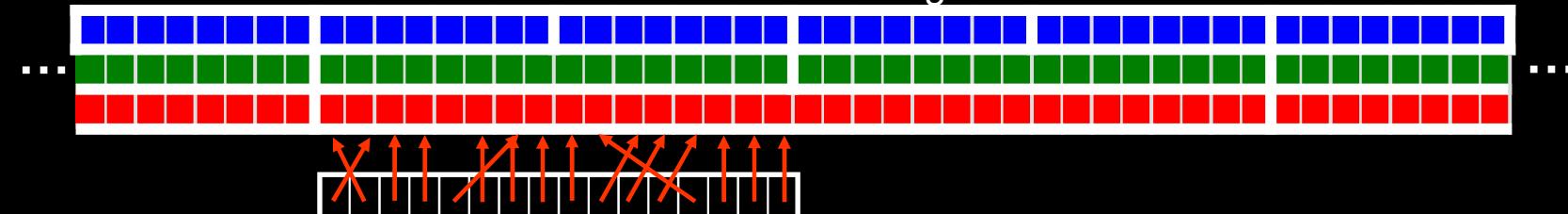


# Coalescing

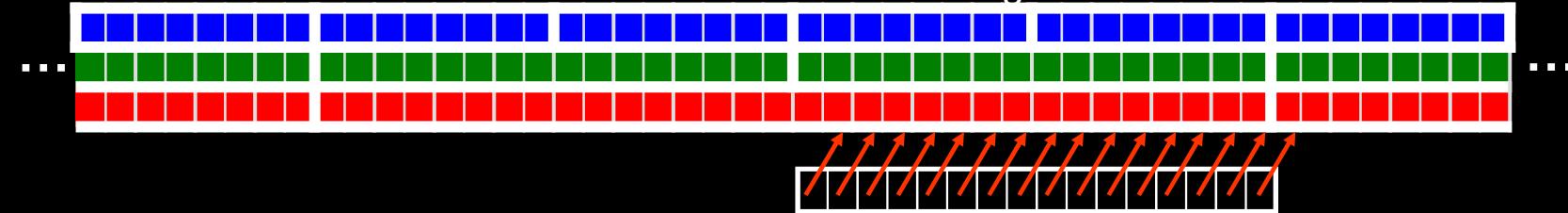
## Compute capability 1.2 and higher

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

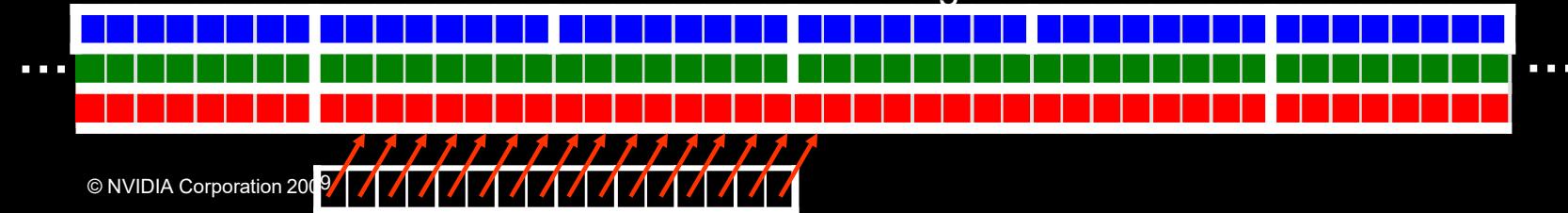
1 transaction - 64B segment



2 transactions - 64B and 32B segments



1 transaction - 128B segment



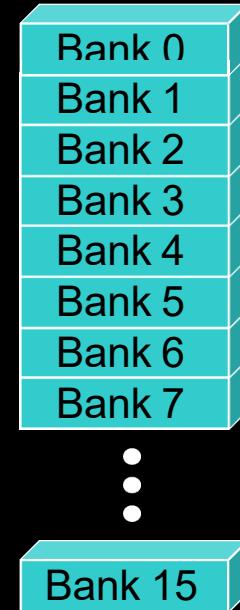


# Shared Memory

- ~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

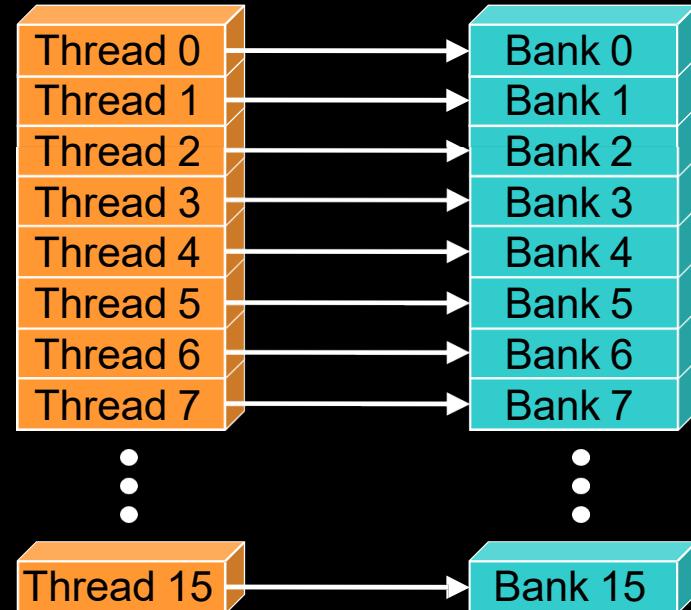
- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Successive 32-bit words assigned to successive banks
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in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



# Bank Addressing Examples

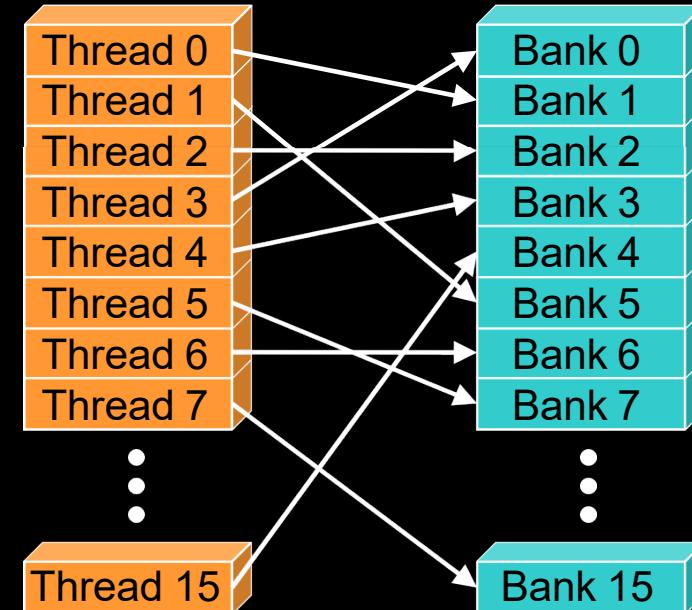
- **No Bank Conflicts**

- Linear addressing  
stride == 1



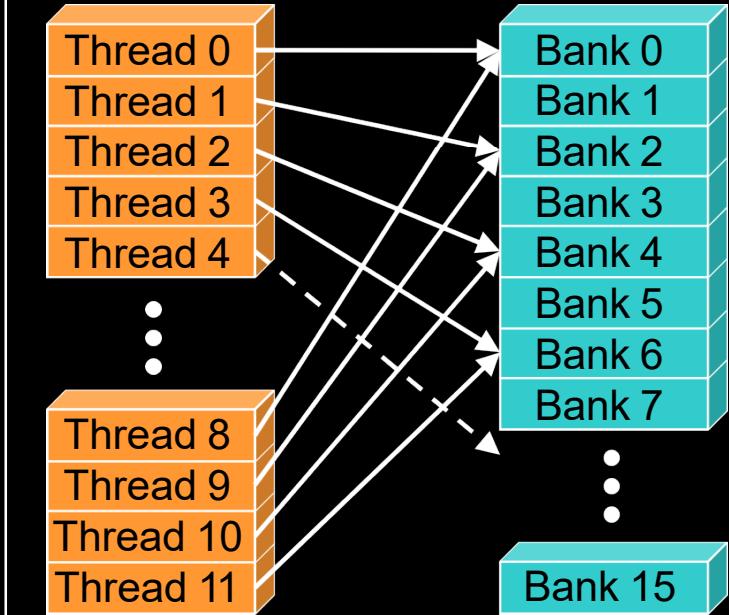
- **No Bank Conflicts**

- Random 1:1 Permutation

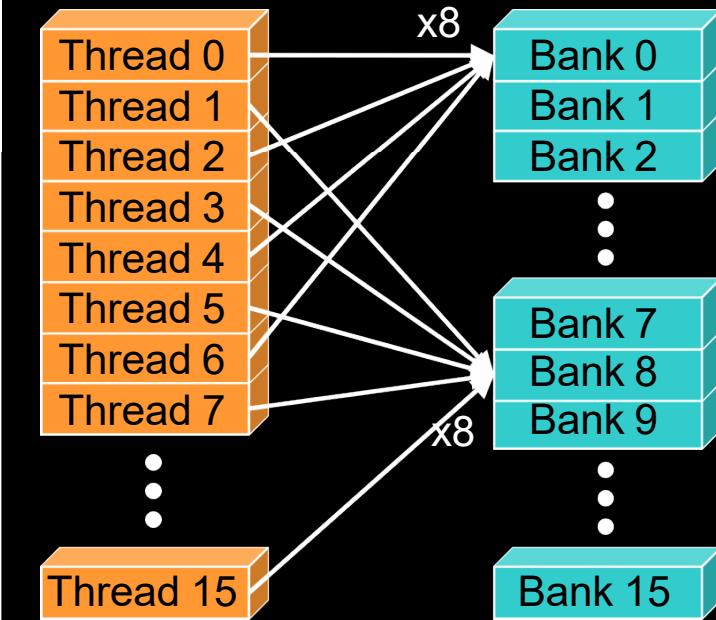


# Bank Addressing Examples

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



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





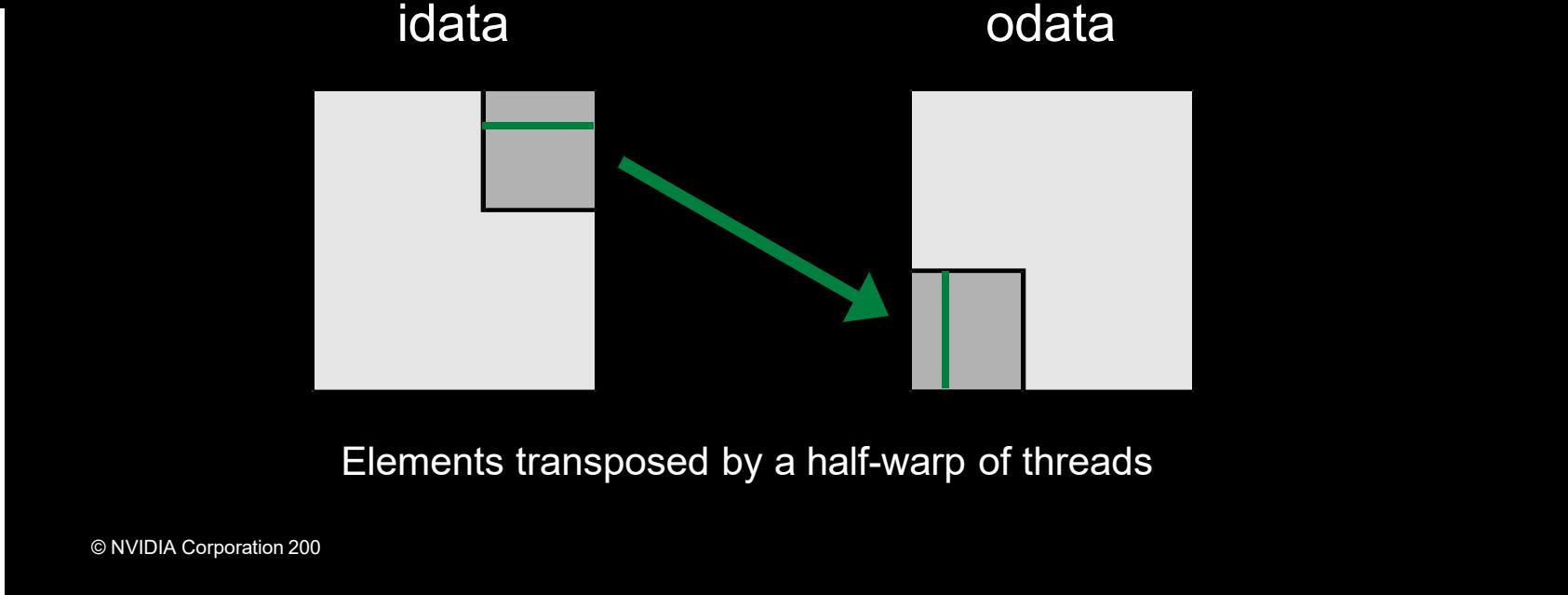
# Shared memory bank conflicts

- 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;

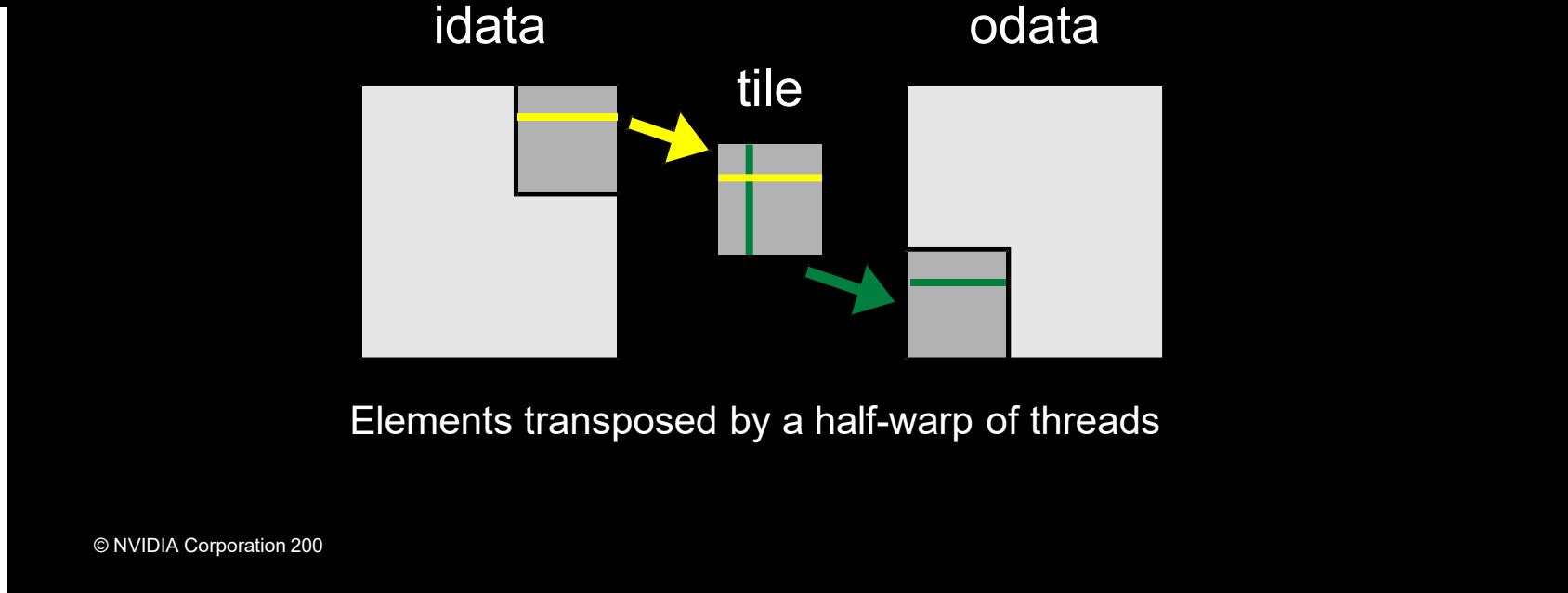
    odata[index_out] = idata[index_in];
}
```



# 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



# 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;

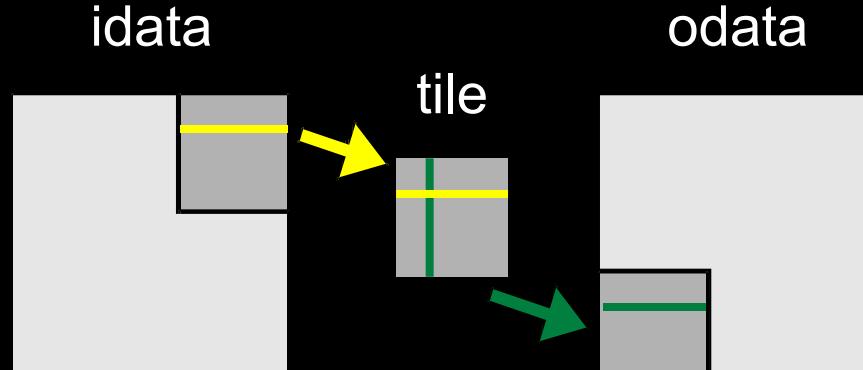
    tile[threadIdx.y][threadIdx.x] = idata[index_in];

    __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



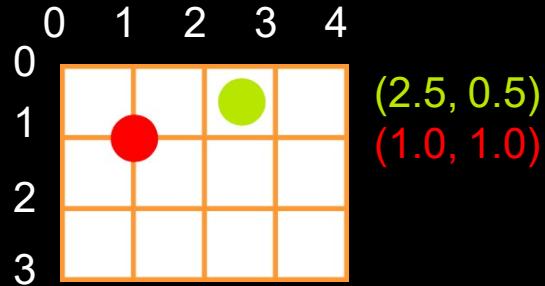
Elements transposed by a half-warp of threads



# Textures in CUDA

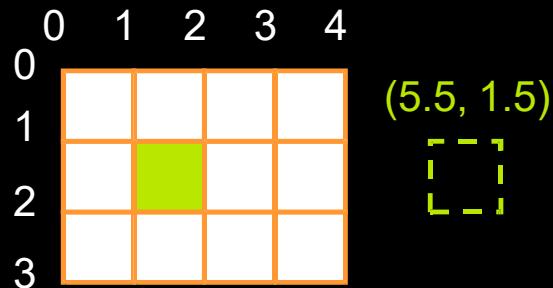
- **Texture is an object for reading data**
- **Benefits:**
  - Data is cached
    - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear interpolation
    - Dedicated hardware
  - Wrap modes (for “out-of-bounds” addresses)
    - Clamp to edge / repeat
  - Addressable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

# Texture Addressing



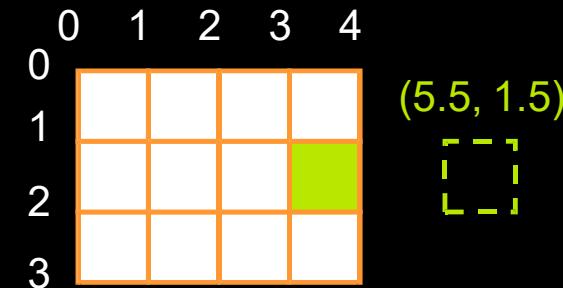
## Wrap

- 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

# 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)
}
```

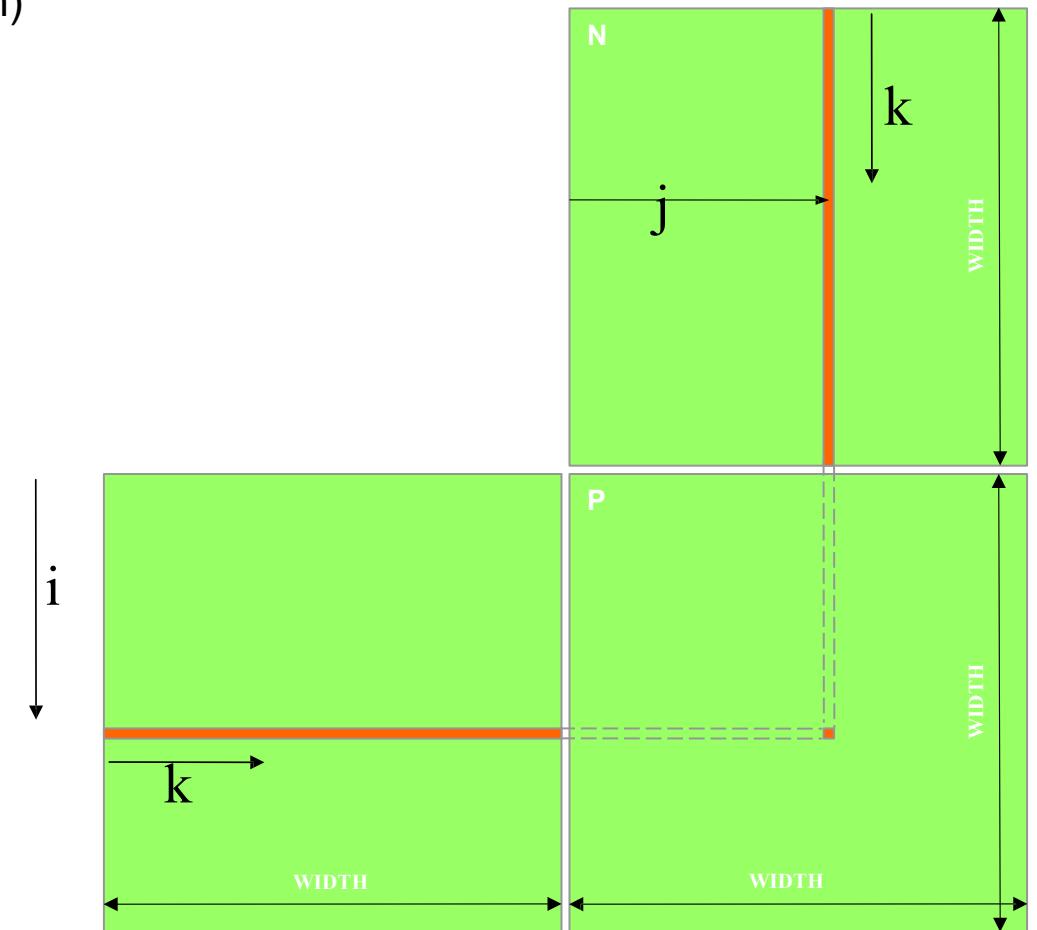
# Communication via Shared Mem.

- 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

# Matrix Multiplication: Simple Host Version

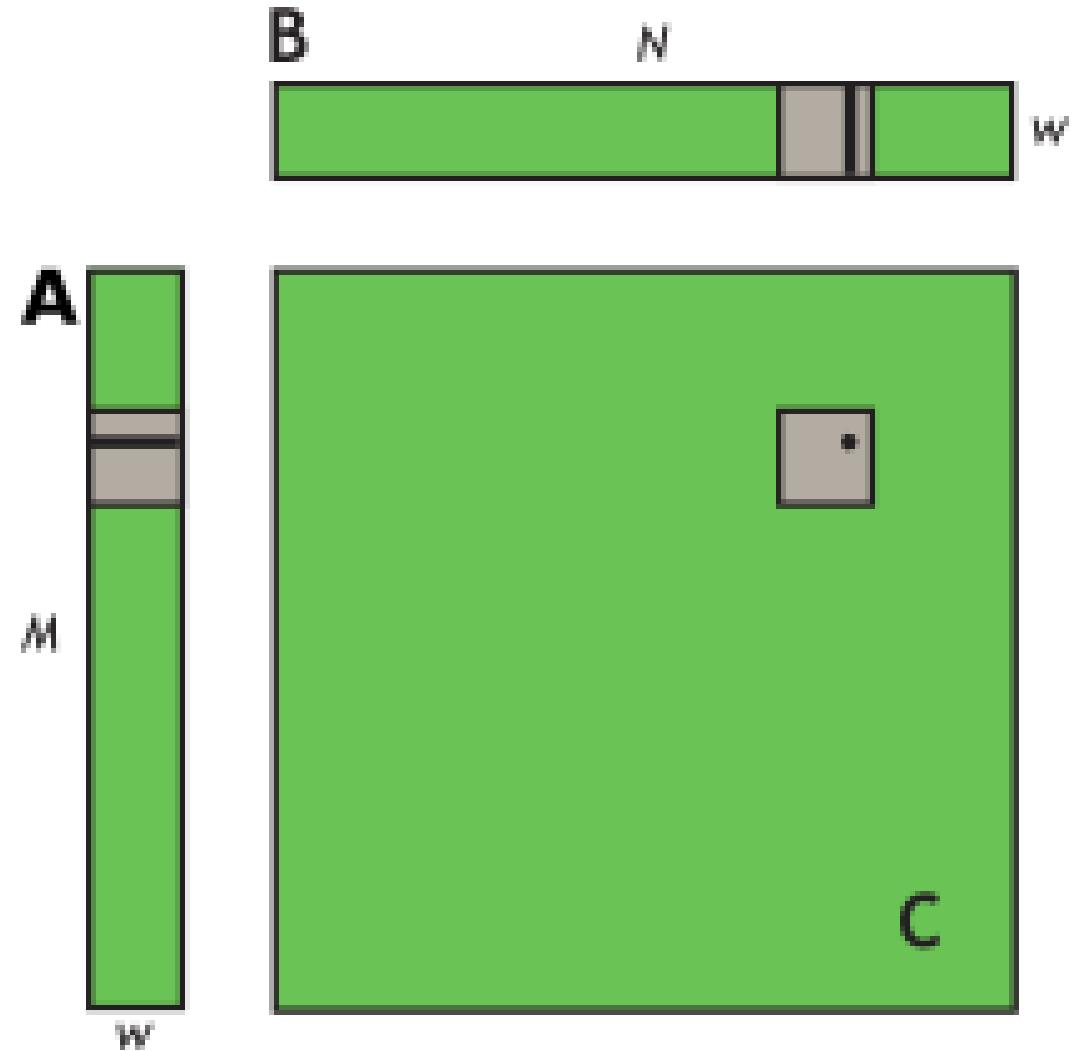
```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
```

```
{  
    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

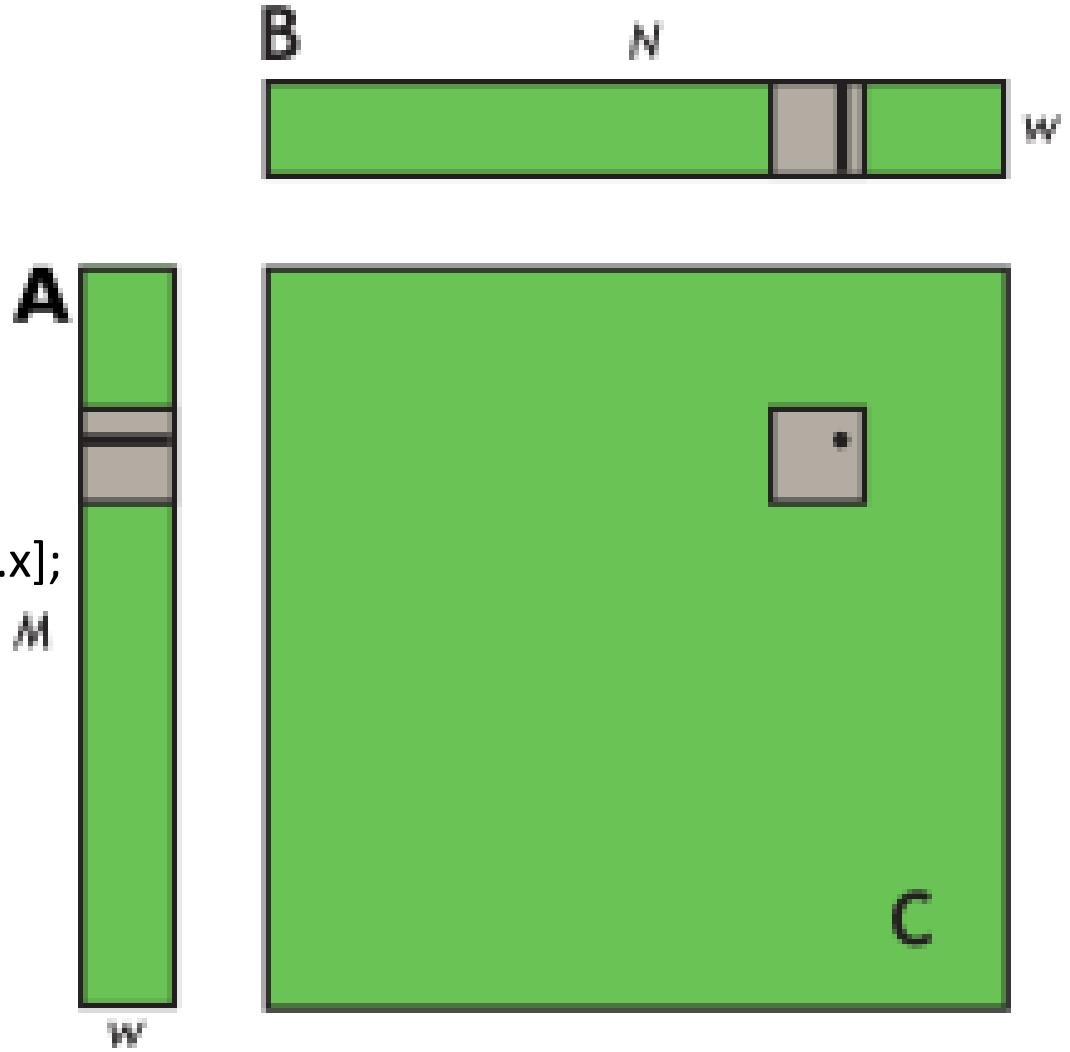
```
__global__ void simpleMultiply(float *a, float* b, float *c,  
                               int N)  
{  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    float sum = 0.0f;  
    for (int i = 0; i < w; i++) {  
        sum += a[row*w+i]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

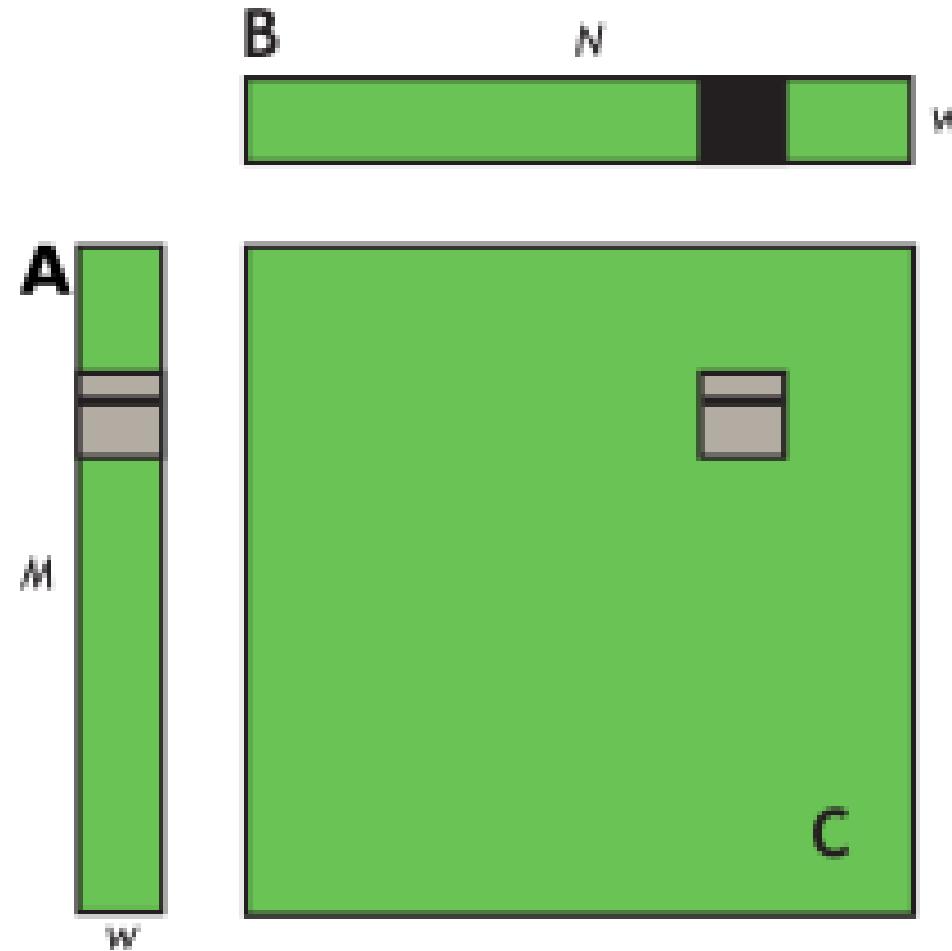


# Kernel with Shared Memory

(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;  
}
```

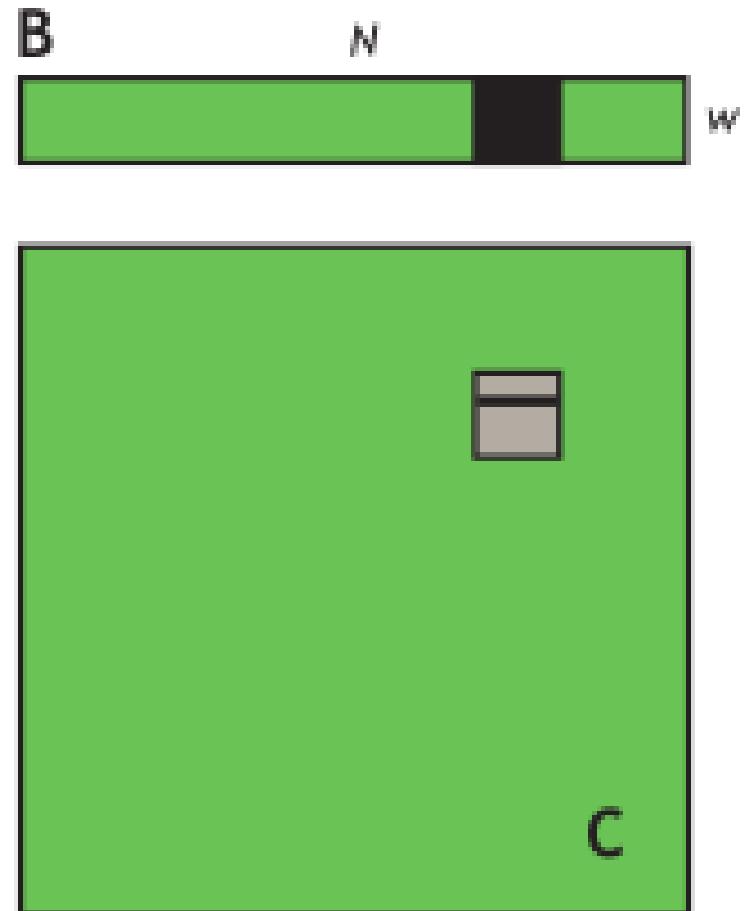




Computing a row of a tile in  $C$  using one row of  $A$  and an entire tile of  $B$

# 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;
}
```



# 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()

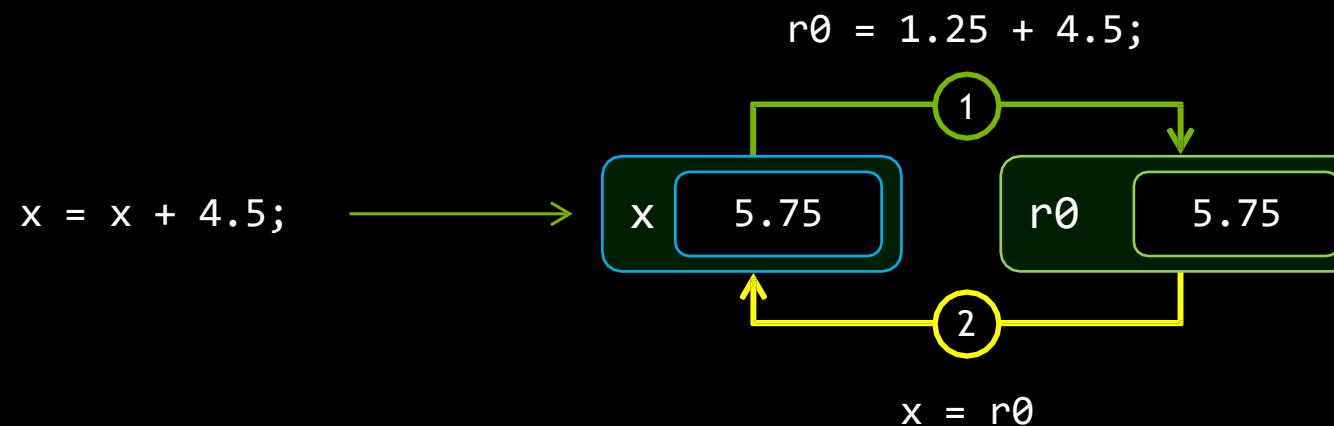
```
int atomicAdd(int *p, int v)
{
    int old;
    exclusive_single_thread
    {
        // atomically perform LD; ADD; ST ops
        old = *p; // Load from memory
        *p = old + v; // Store after adding v
    }
    return old;
}
```

# Simple Atomic Example

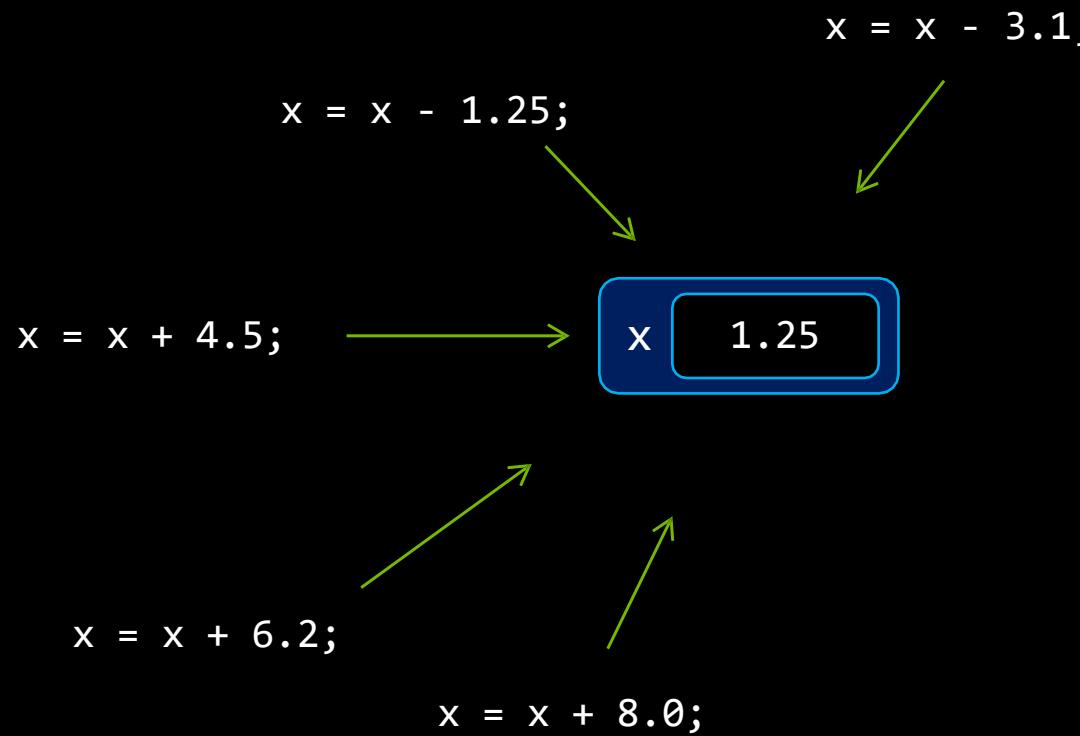
$x = x + 4.5;$



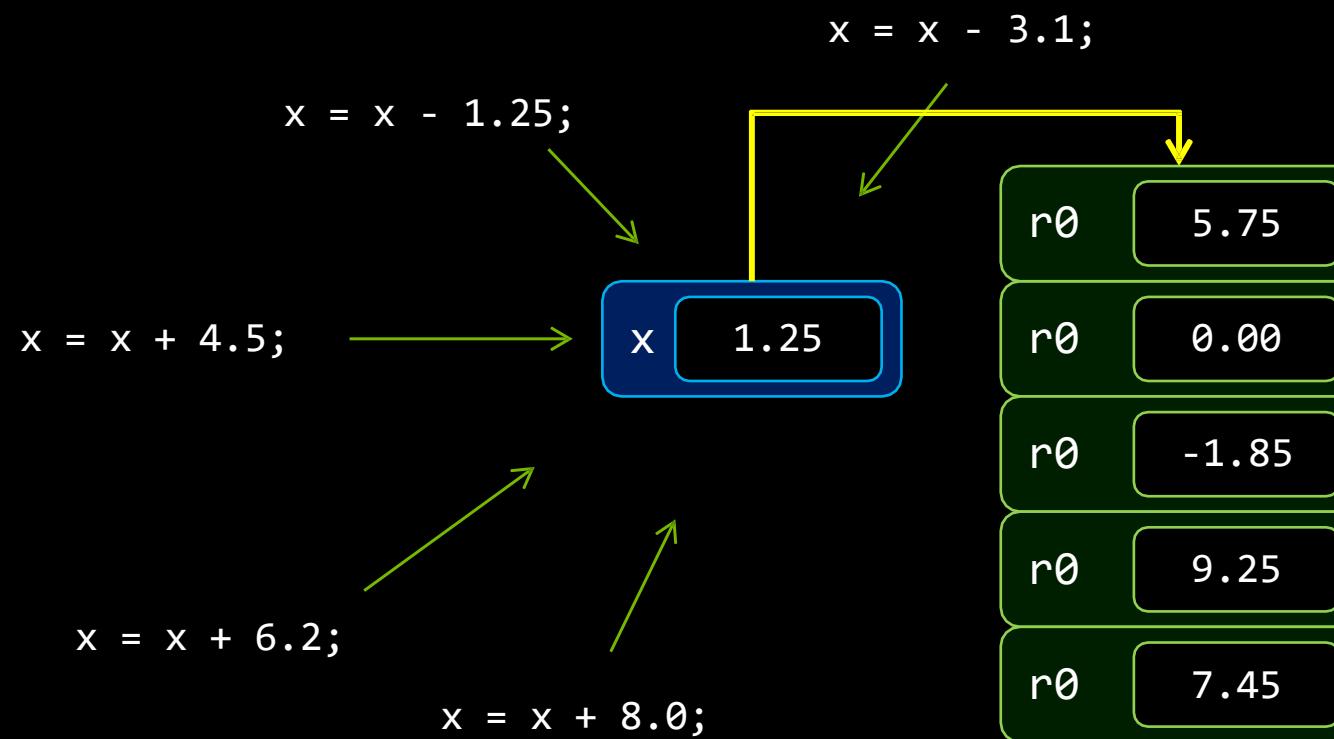
# Simple Atomic Example



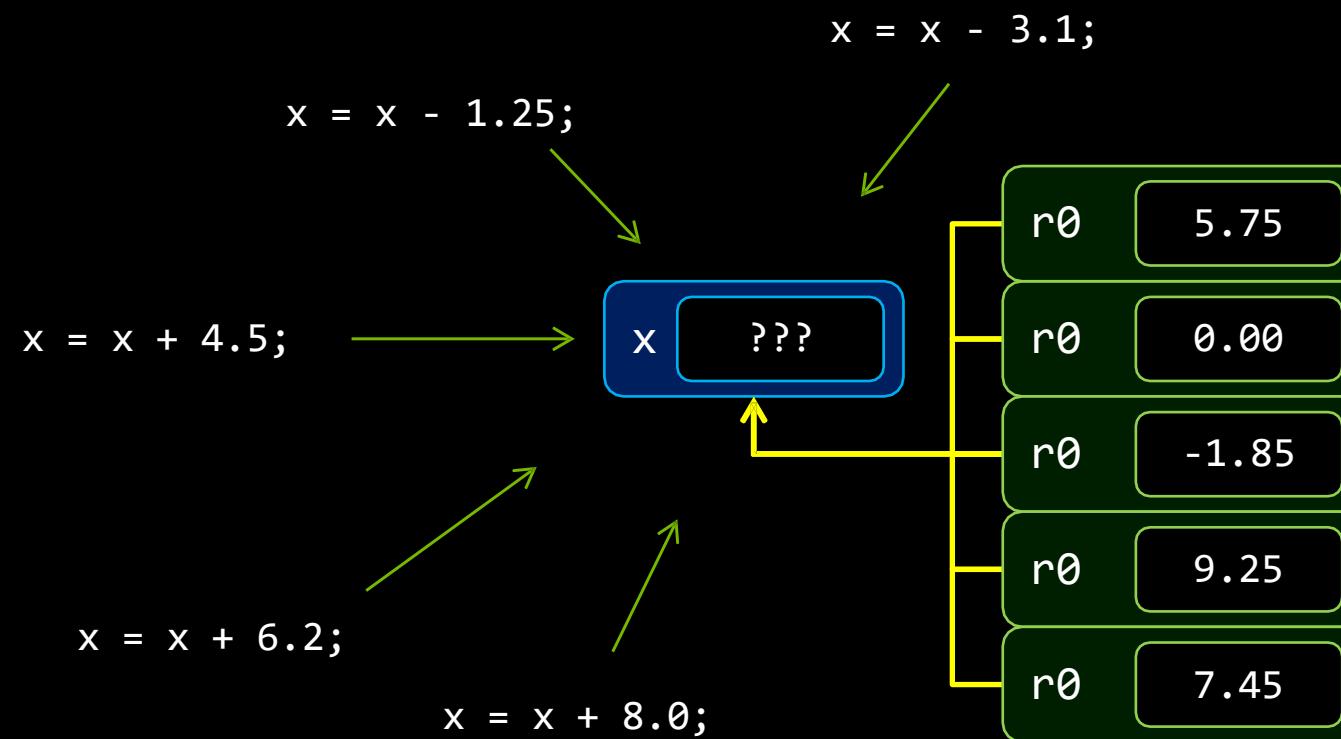
# Simple Atomic Example



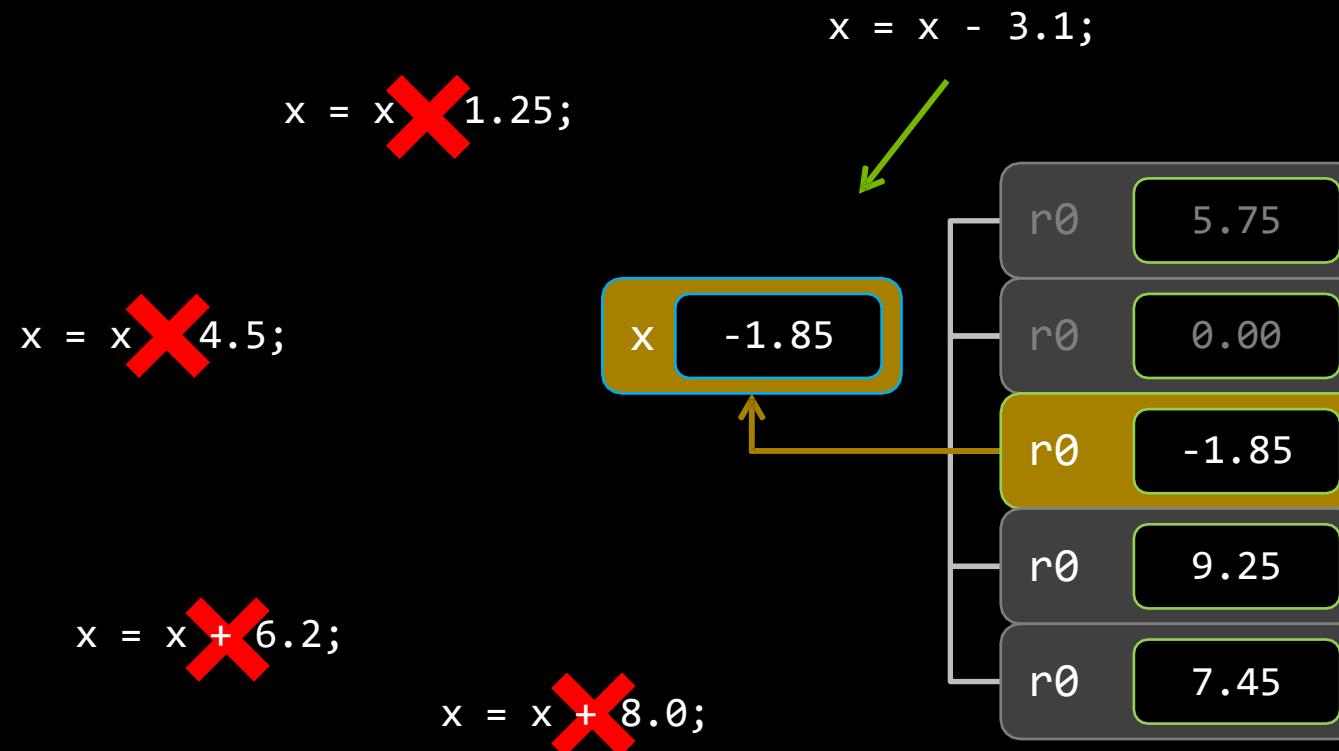
# Simple Atomic Example



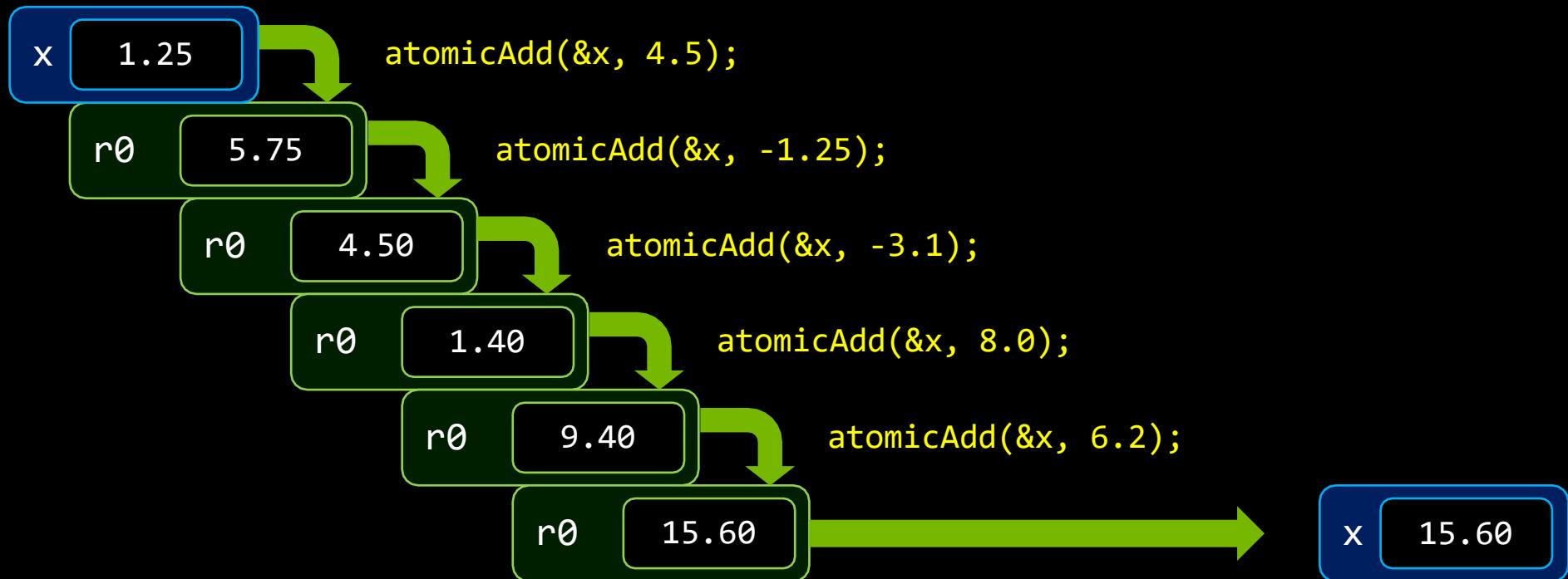
# Simple Atomic Example



# Simple Atomic Example



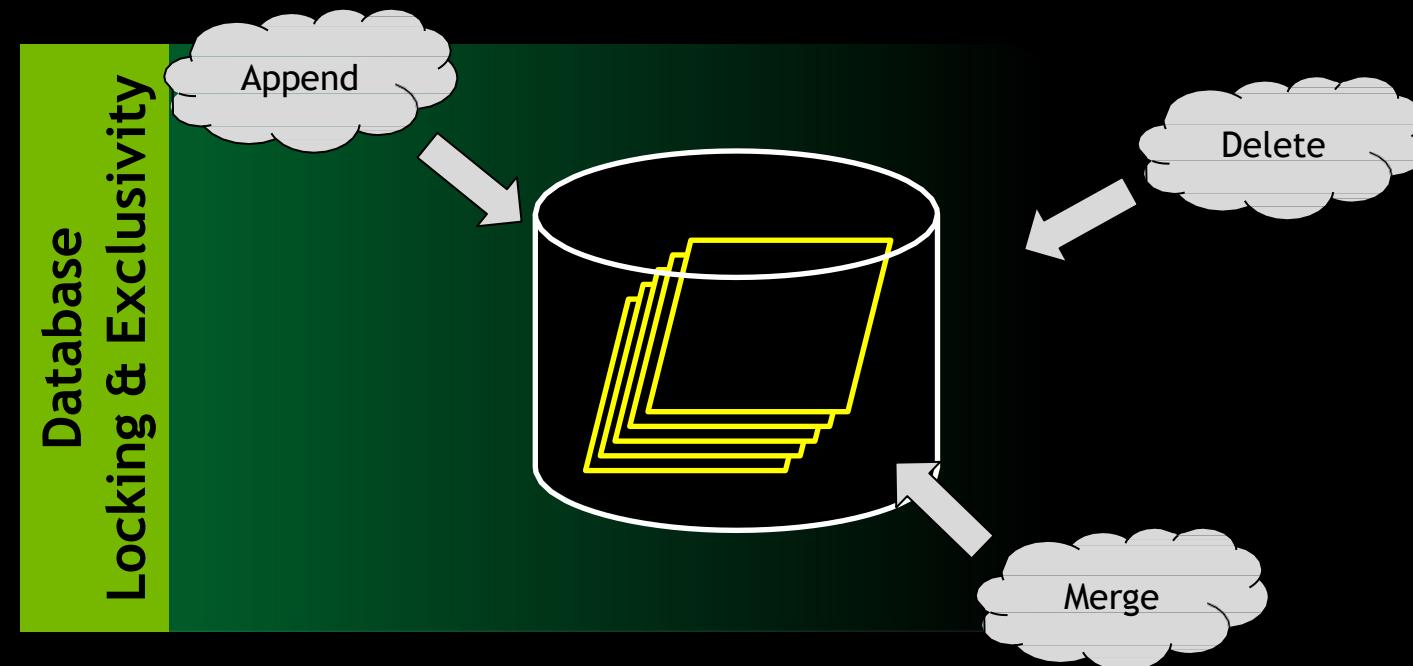
# Simple Atomic Example



# Why Use Atomics?

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

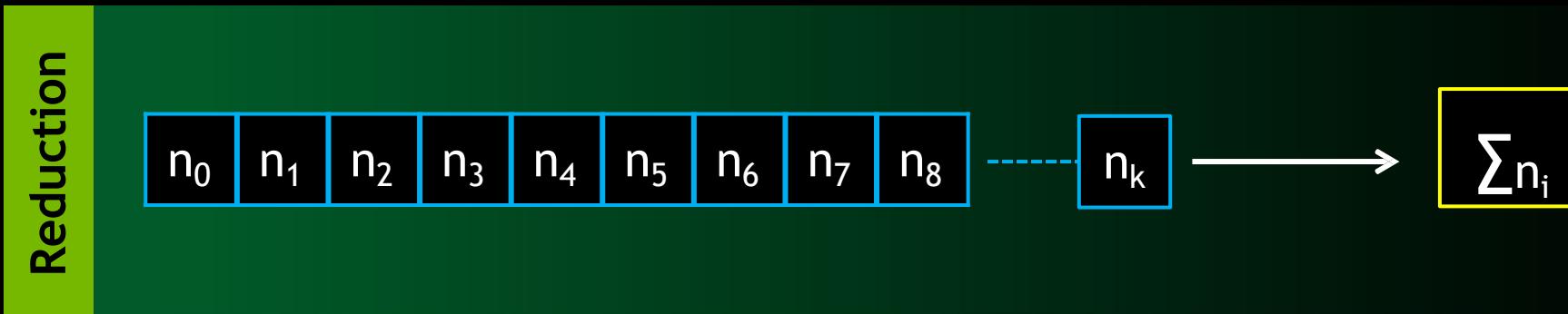
- Transactions & Data Access Control



# Why Use Atomics?

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

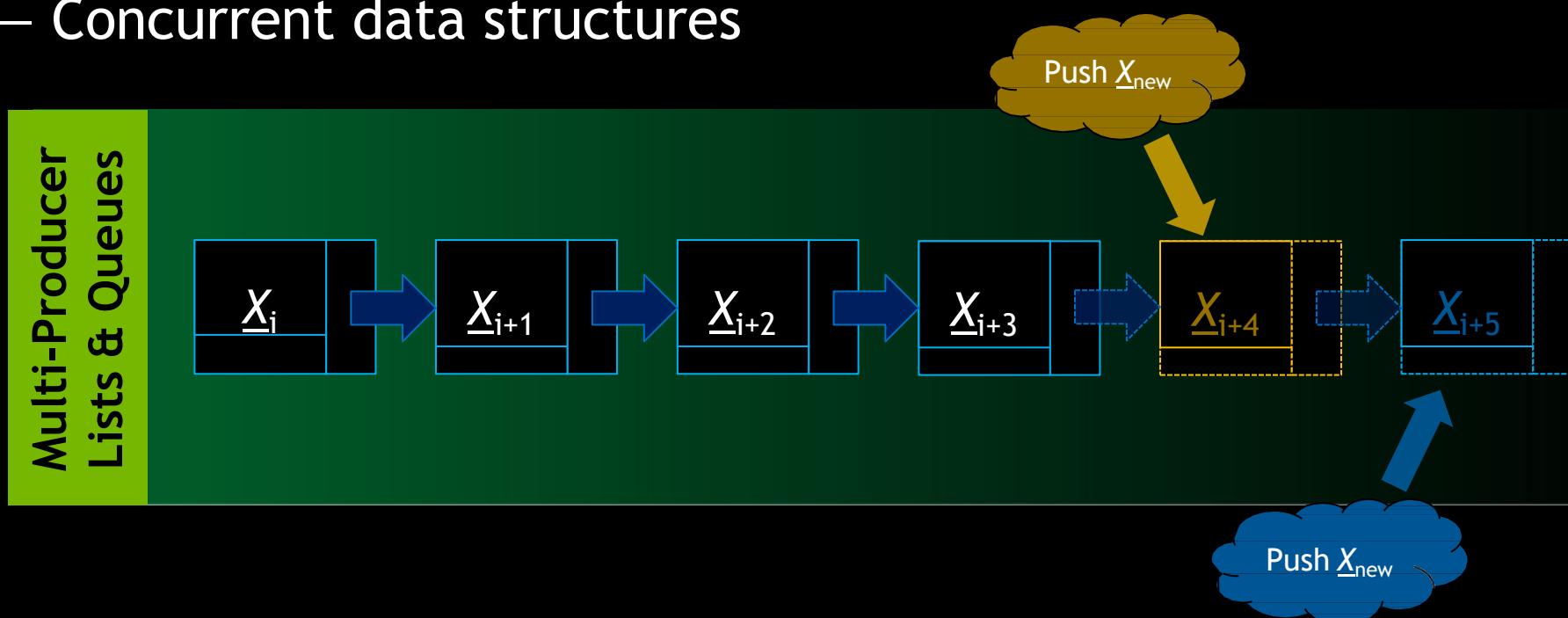
- Transactions & Data Access Control
- Data aggregation & enumeration



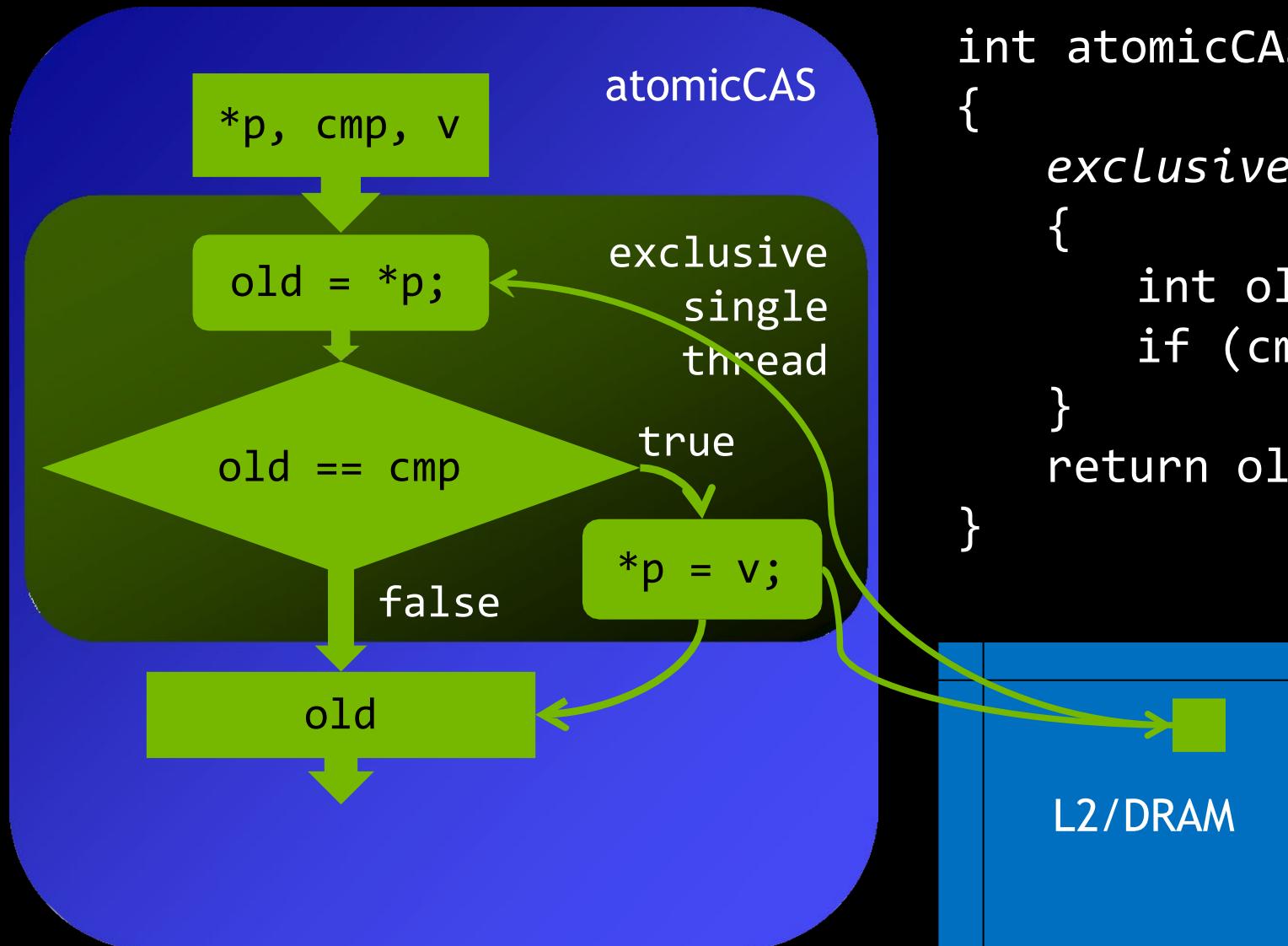
# Why Use Atomics?

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

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

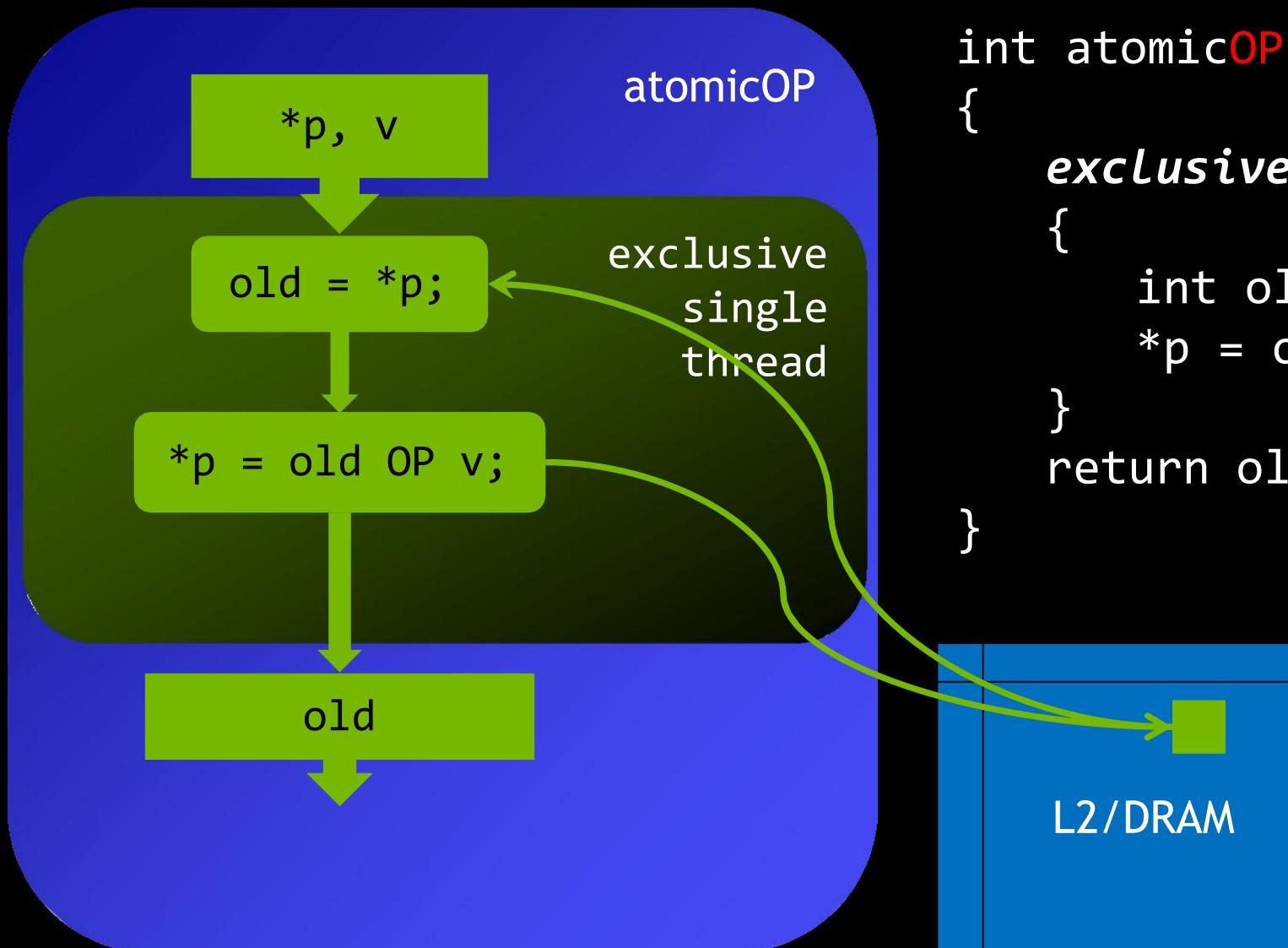


# Compare-and-Swap

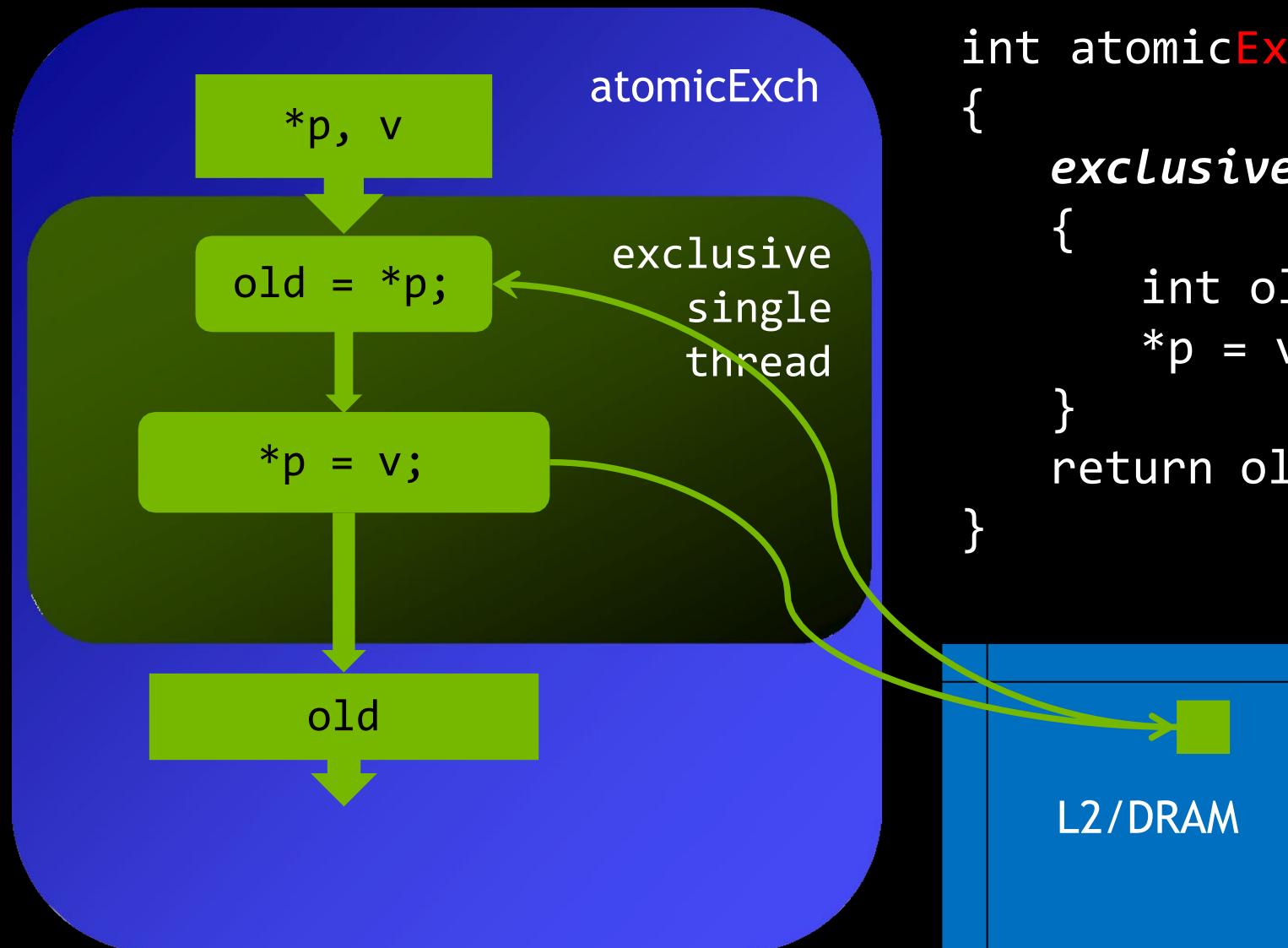


```
int atomicCAS(int *p, int cmp, int v
{
    exclusive_single_thread
    {
        int old = *p;
        if (cmp == old) *p = v;
    }
    return old;
}
```

# Arithmetic/Logical Atomic Operations



# Overwriting Atomic Operations



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

# Programming Styles using Coordination

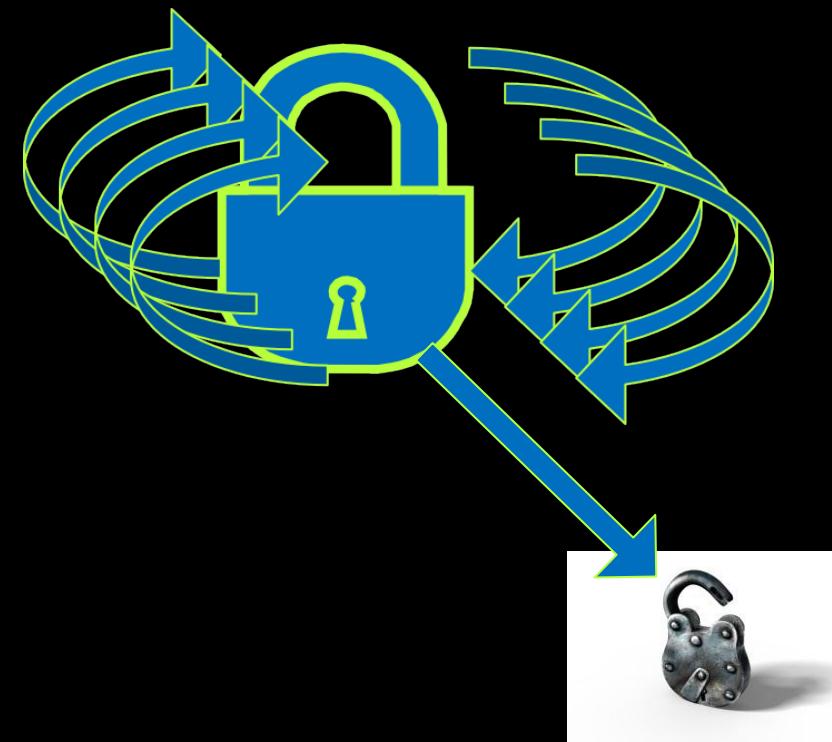
Locking

Lock-free

Wait-free

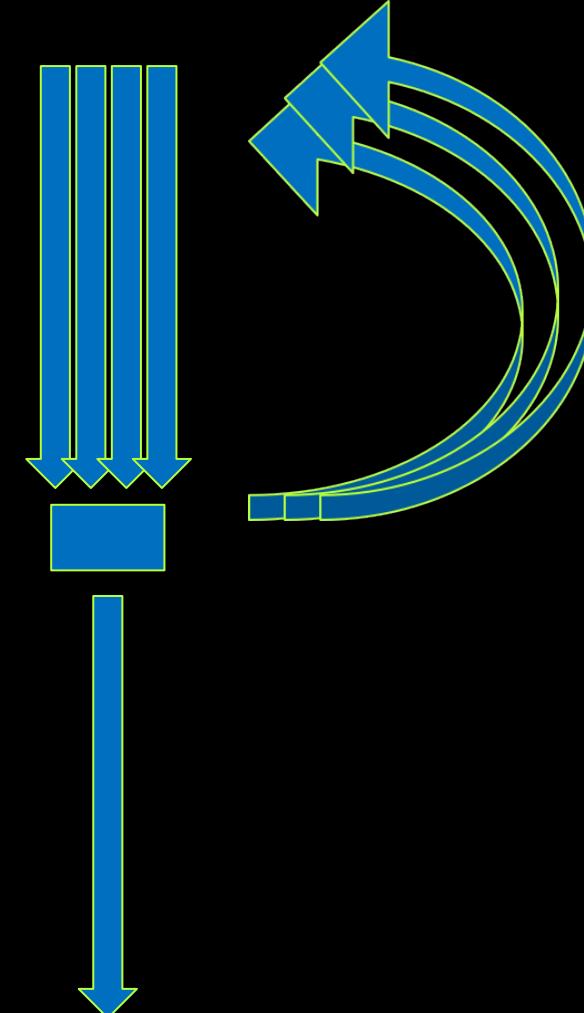
# Locking Style of Programming

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



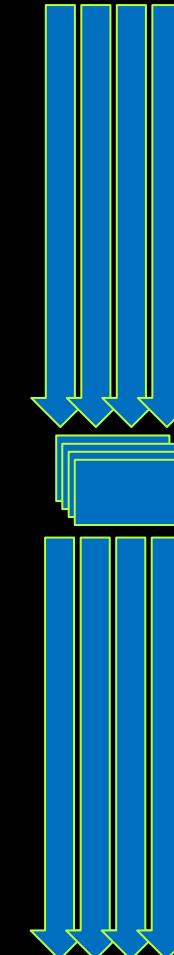
# Lock-Free Style of Programming

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

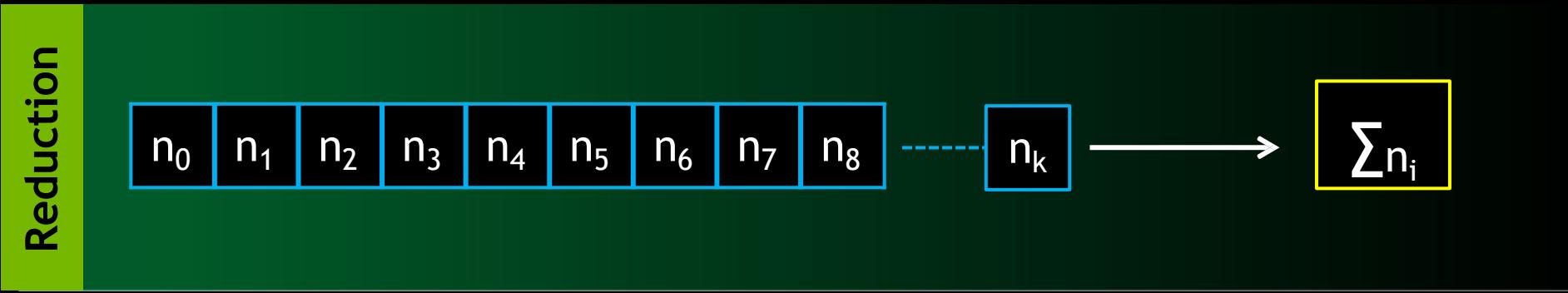


# Wait-free Style of Programming

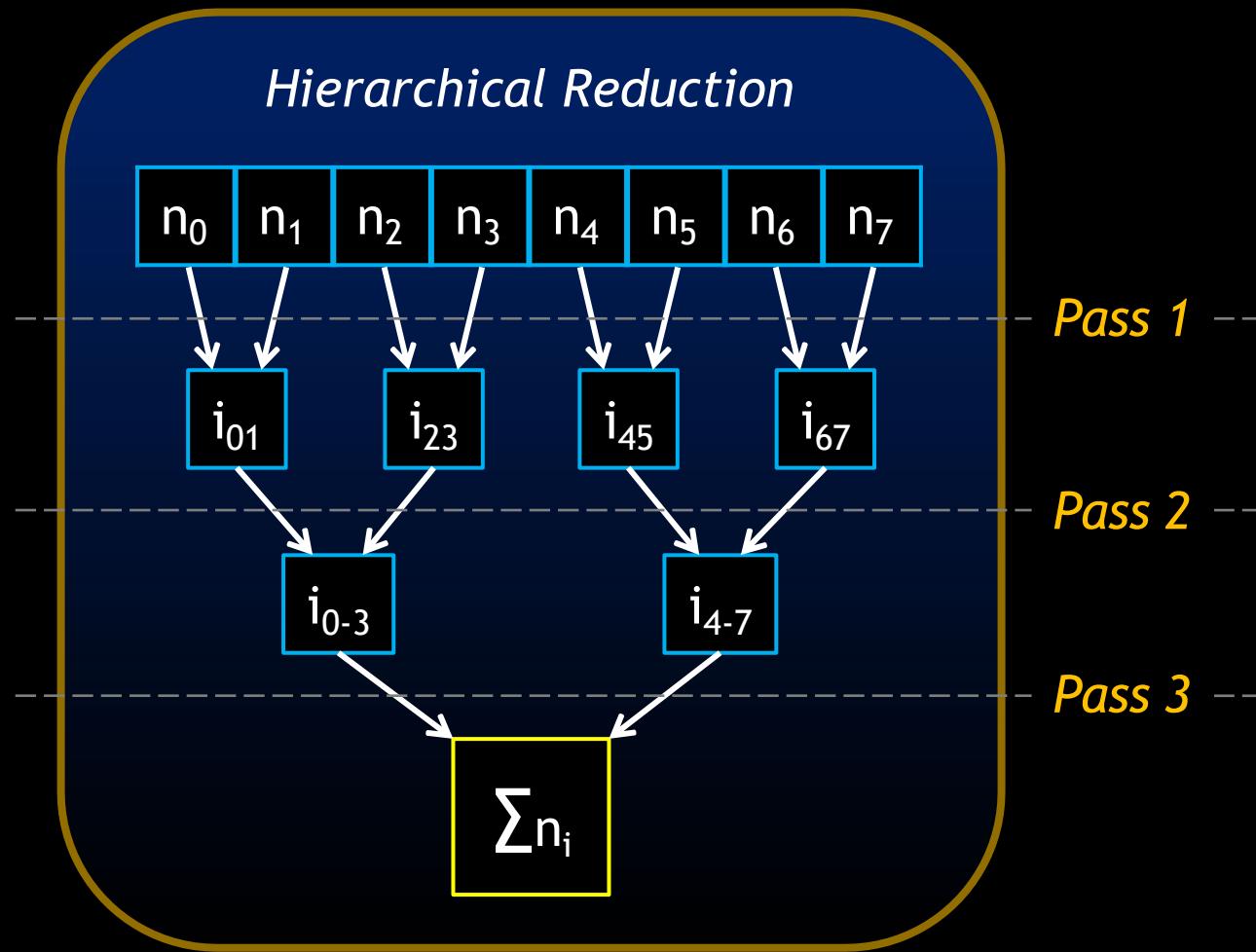
- All threads make progress
- Each updates memory atomically
- No thread blocked by other threads



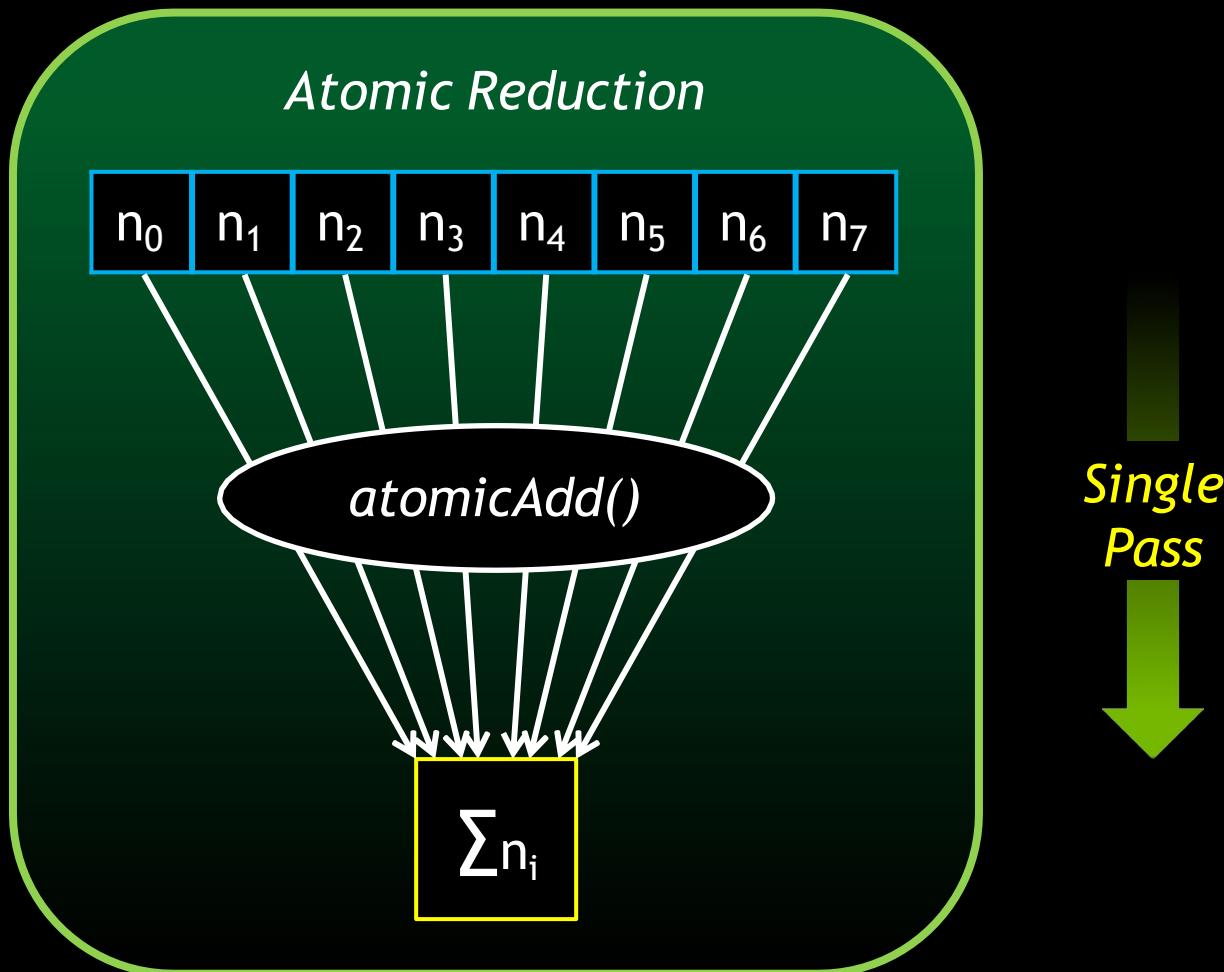
# Atomic Arithmetical Operations



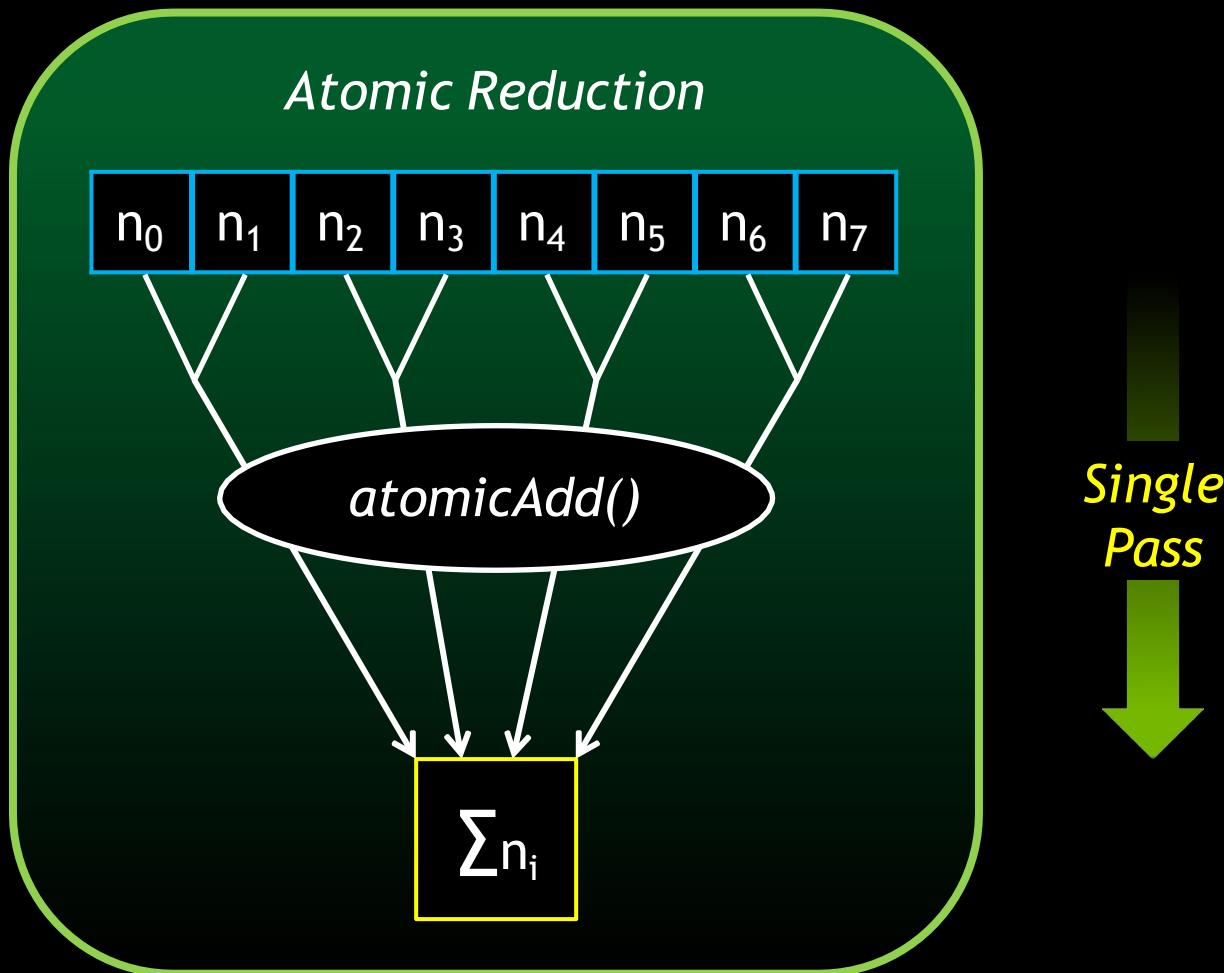
# Atomic Arithmetical Operations



# Atomic Arithmetical Operations

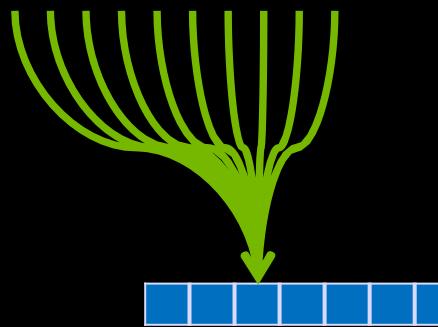


# Atomic Arithmetical Operations



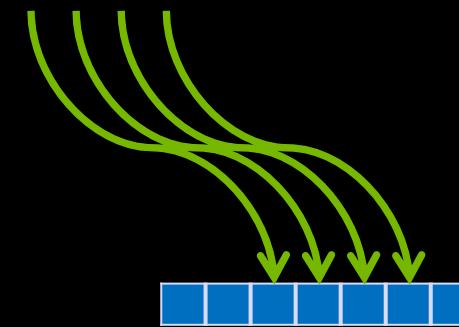
# Atomic Access Patterns

*Same Address*



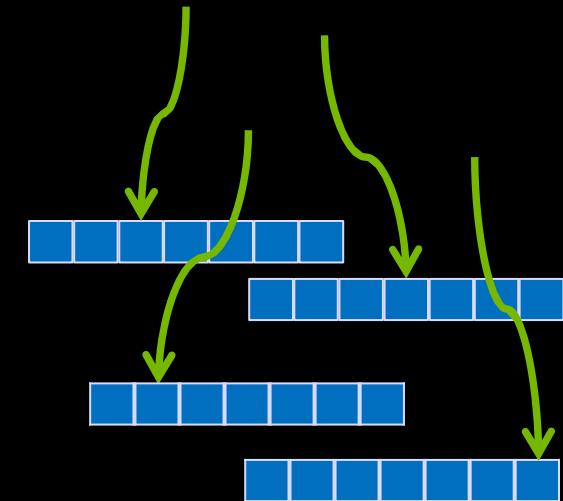
- 1 per clock

*Same Cache Line*



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

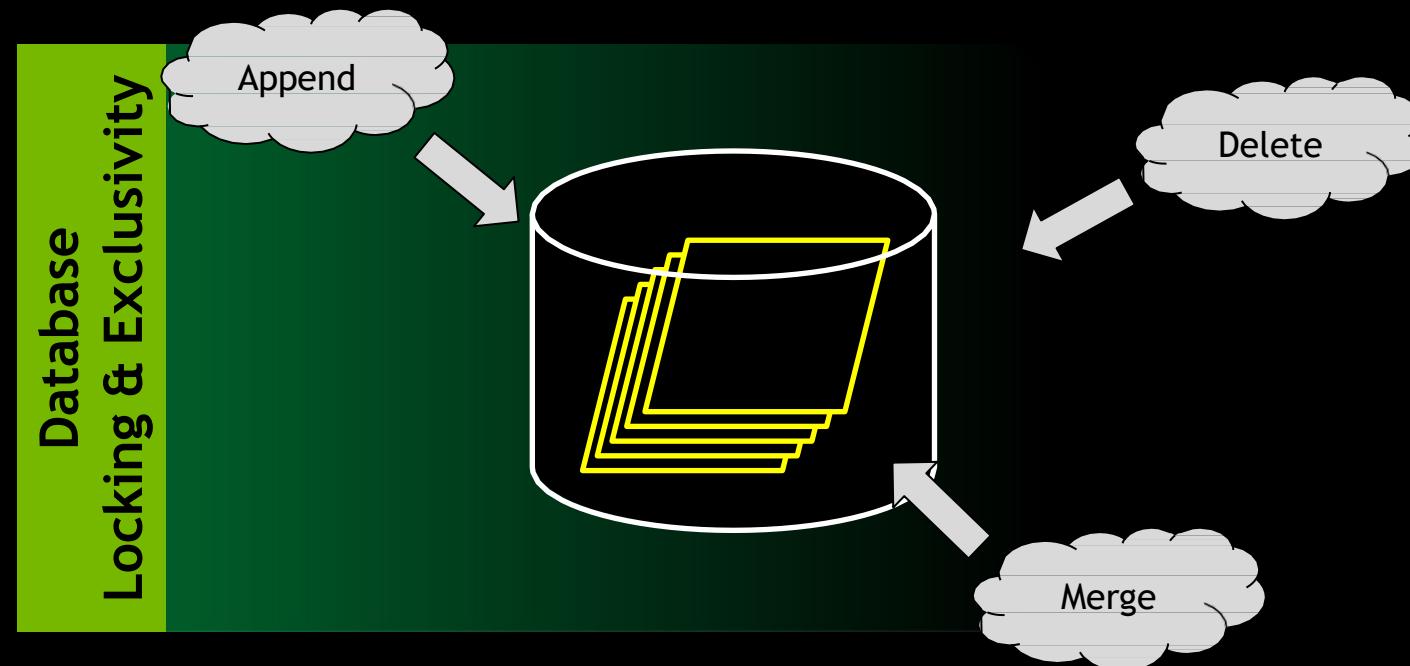
*Scattered*



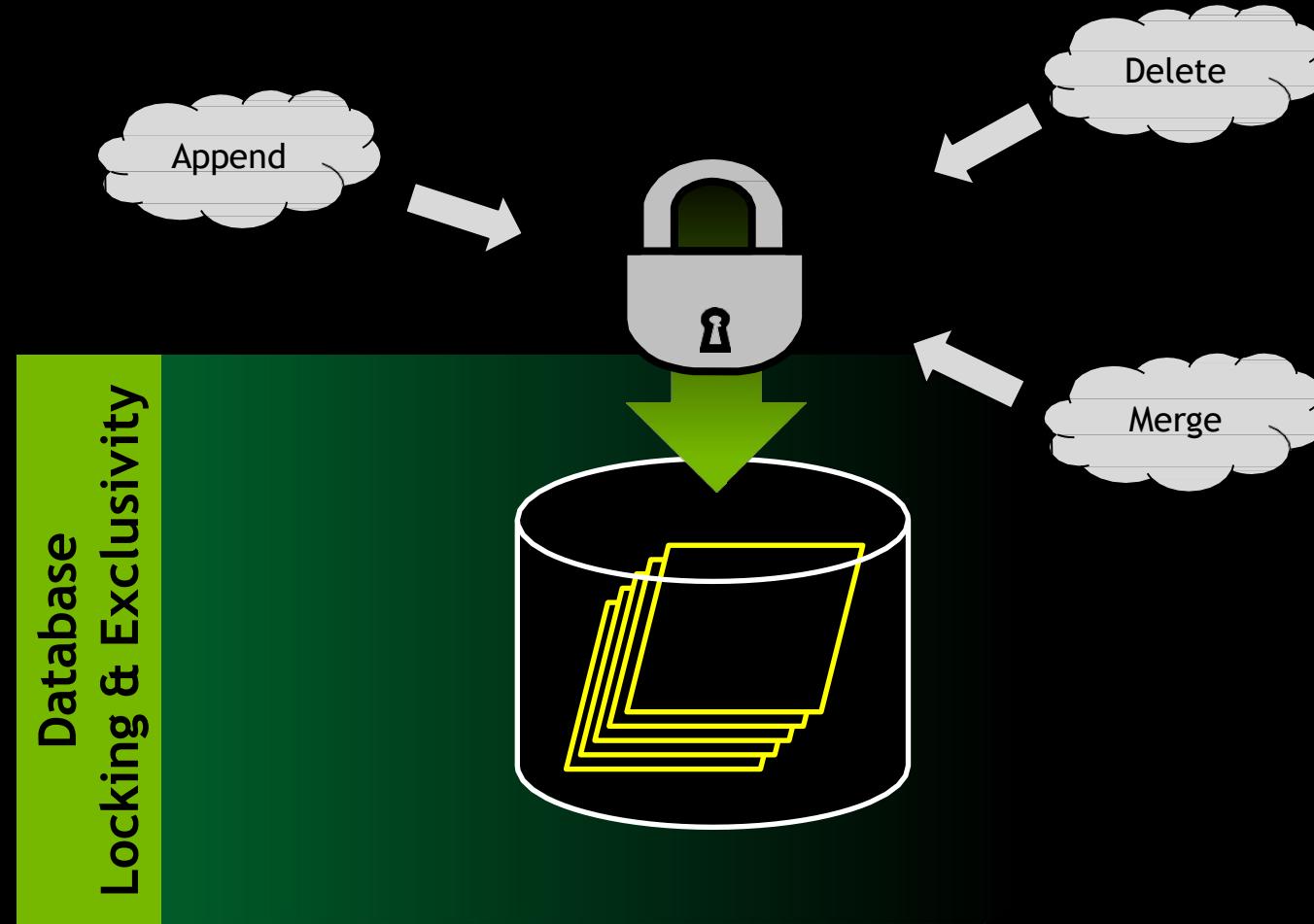
- Issued per cache-line
- 1 per SM per clock

# 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)*

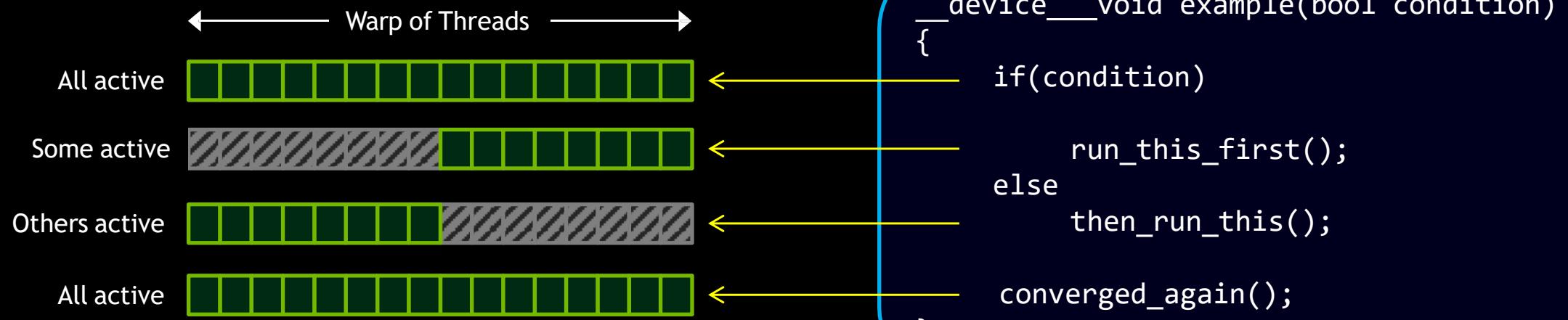
Atomically set (\*data = new), and return  
the previous value

```
// 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*:

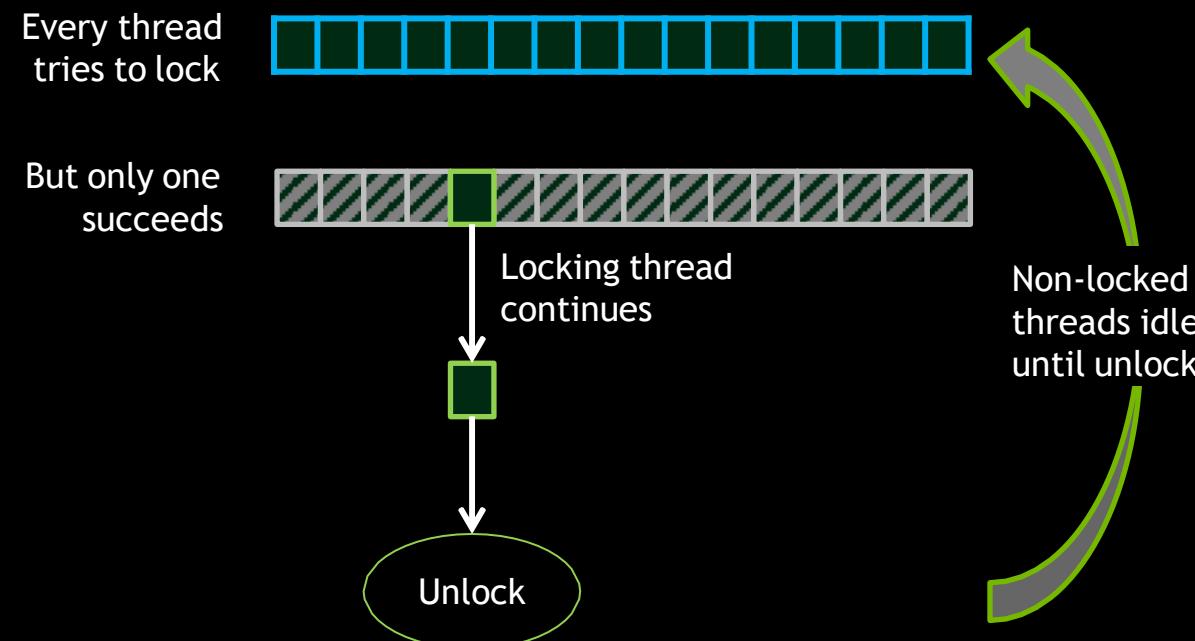
- A group of threads (32 on current GPUs) scheduled in lock-step
- All threads execute the same line of code
- Any thread not participating is idle



# Locks & Warp Divergence

What does this mean for locks?

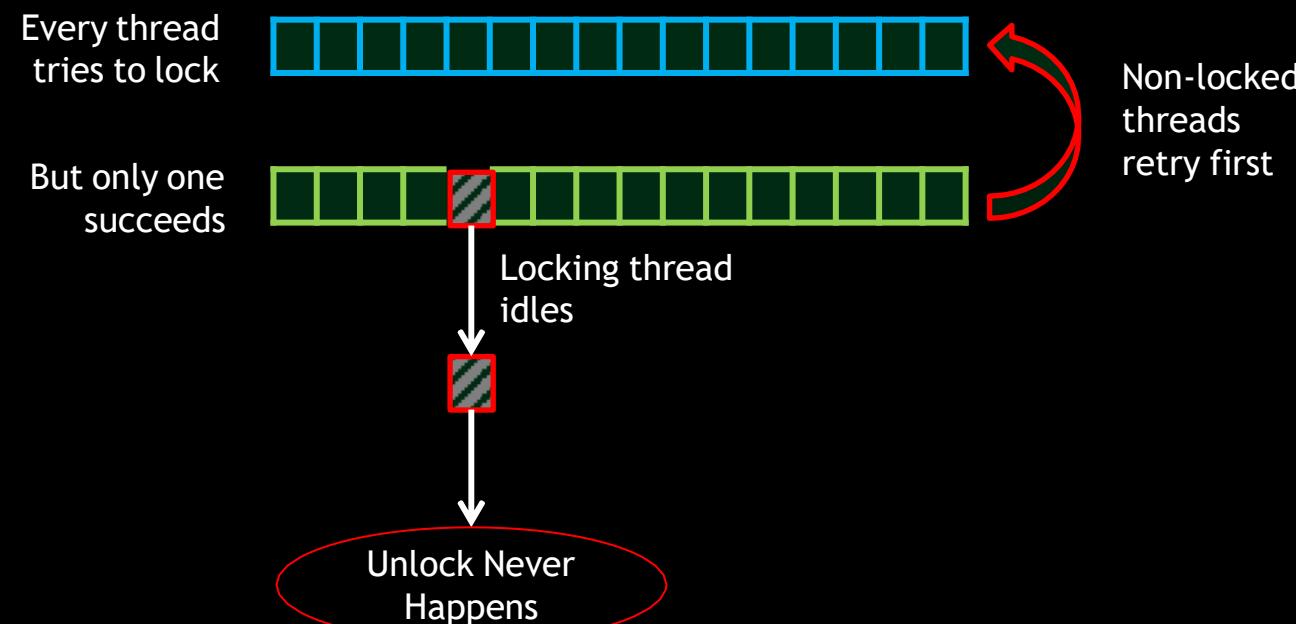
- Only one thread in the warp will lock
- We're okay so long as that's the thread which continues



# 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...

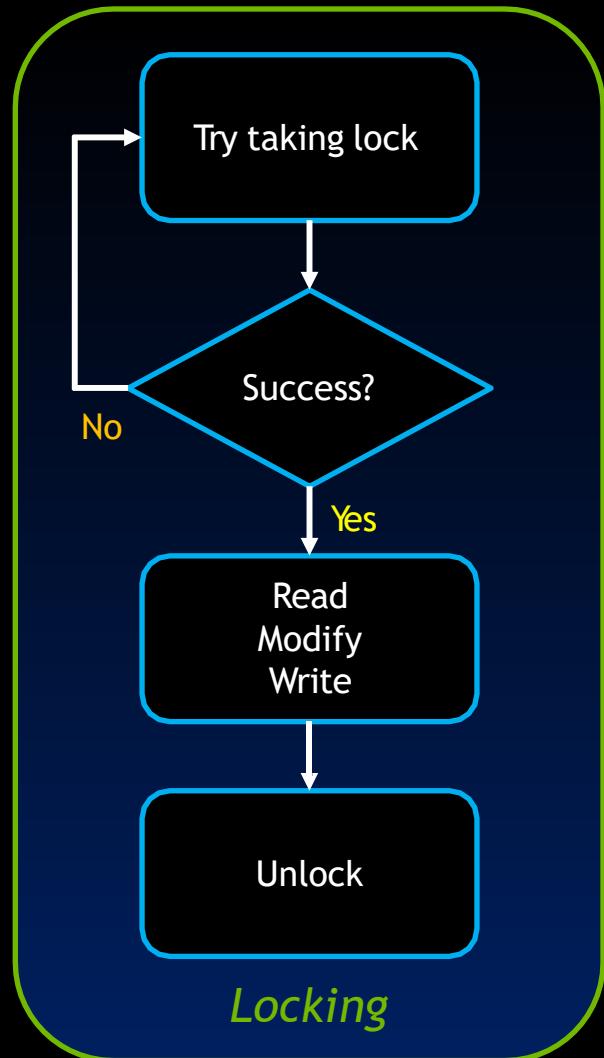
# Lock Free Algorithms: Better Than Locks

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

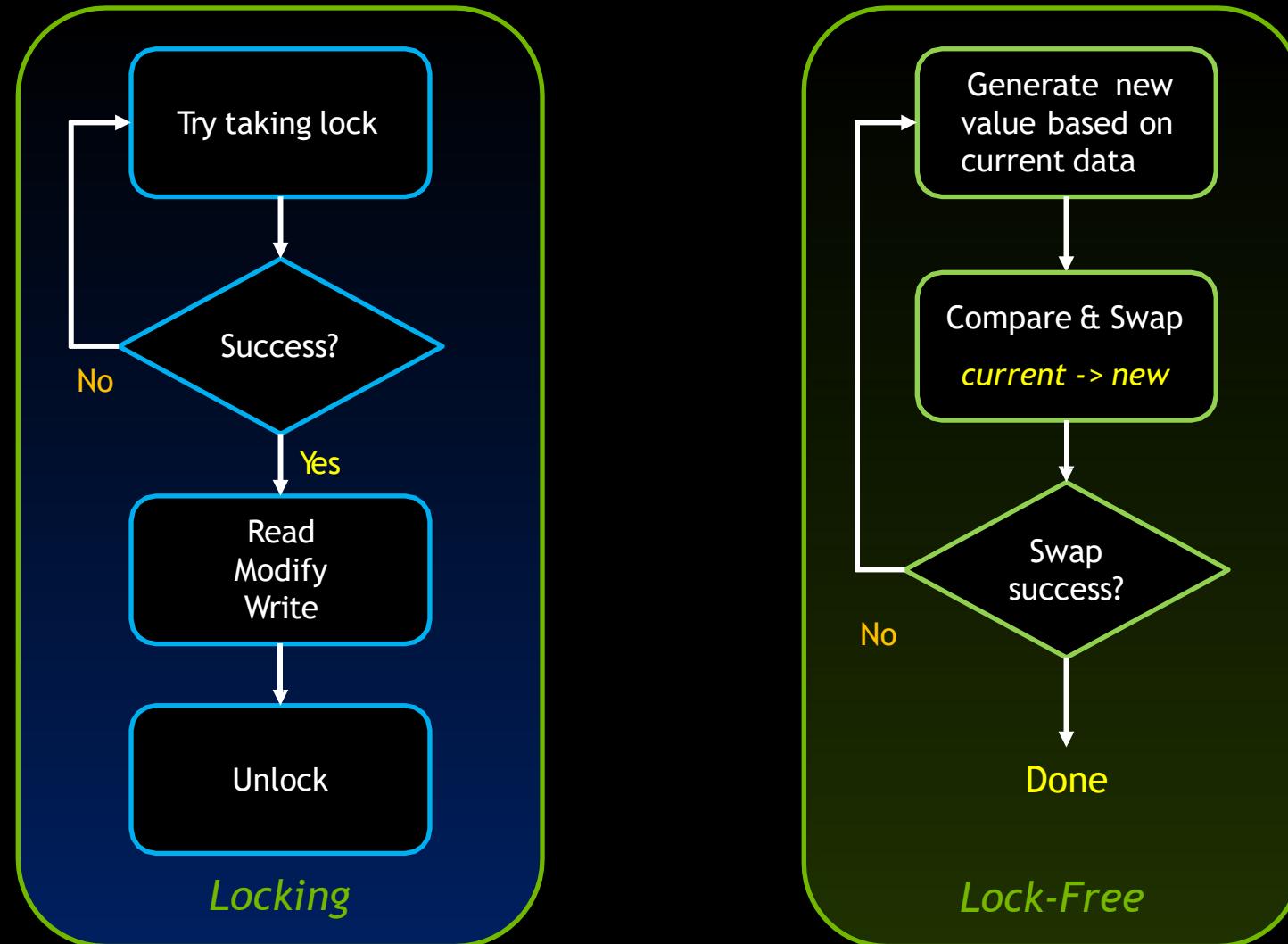
```
uint64 atomicCAS(uint64 *data, uint64 oldval, uint64 newval);  
If “*data” is equal to “oldval”, replace it with “newval”  
Always returns original value of “*data”
```

# Lock-Free Data Updates

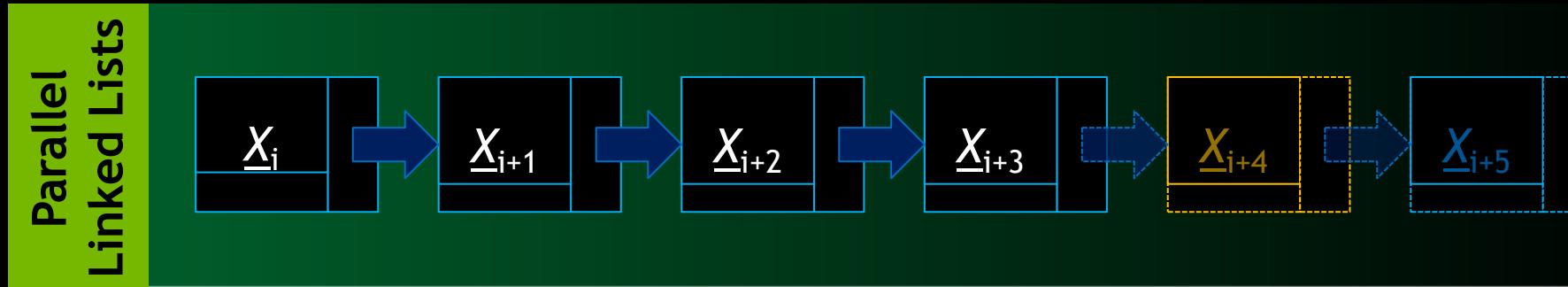


```
// Add "val" to "*data". Return old value.  
double atomicAdd(double *data, double val)  
{  
    while(atomicExch(&locked, 1) != 0)  
        ; // Retry lock  
  
    double old = *data;  
    *data = old + val;  
    locked = 0;  
  
    return old;  
}
```

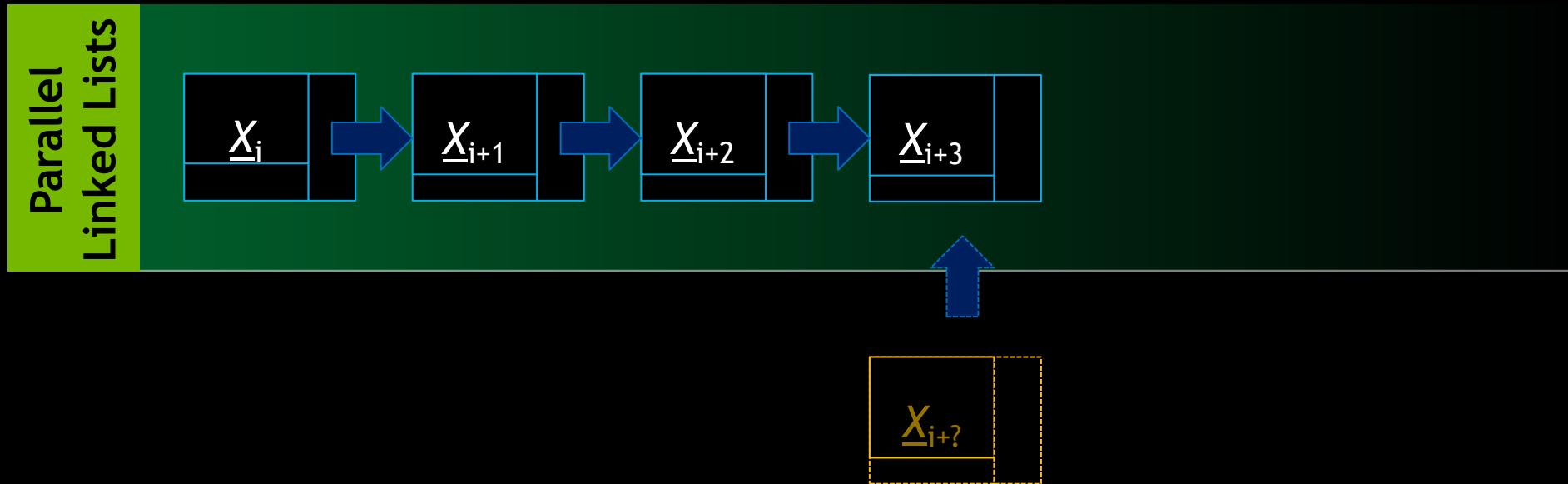
# Lock-Free Data Updates



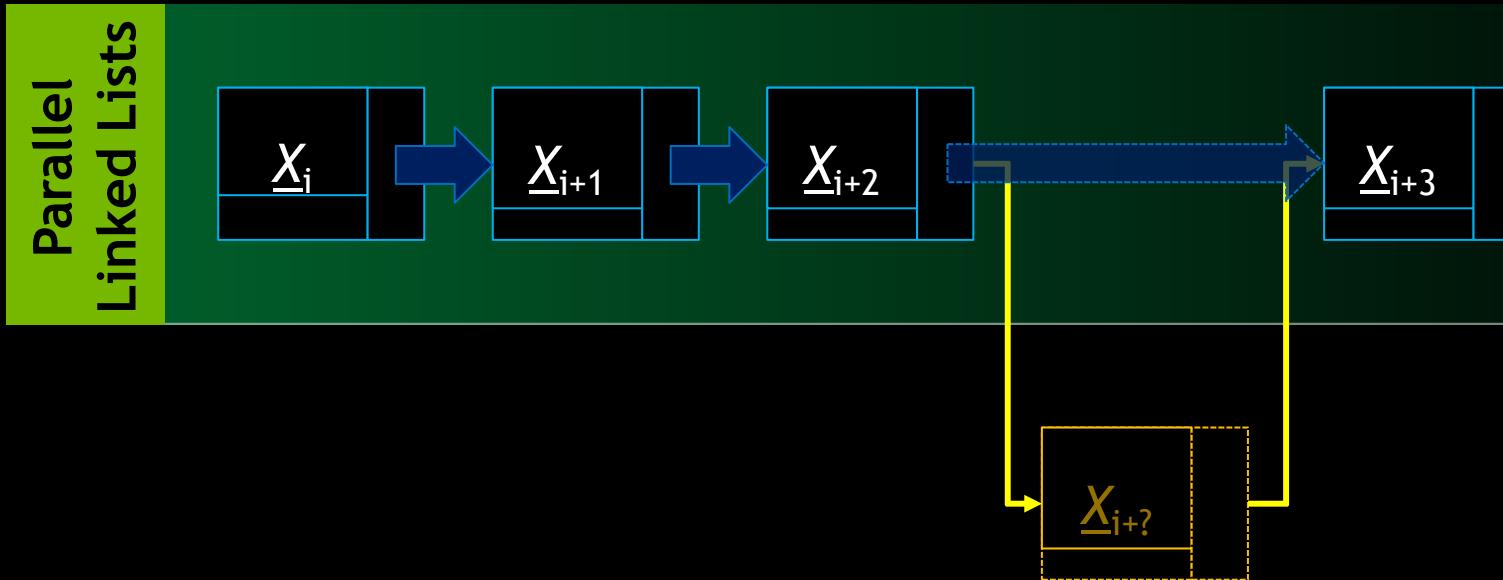
# Lock-Free Parallel Data Structures



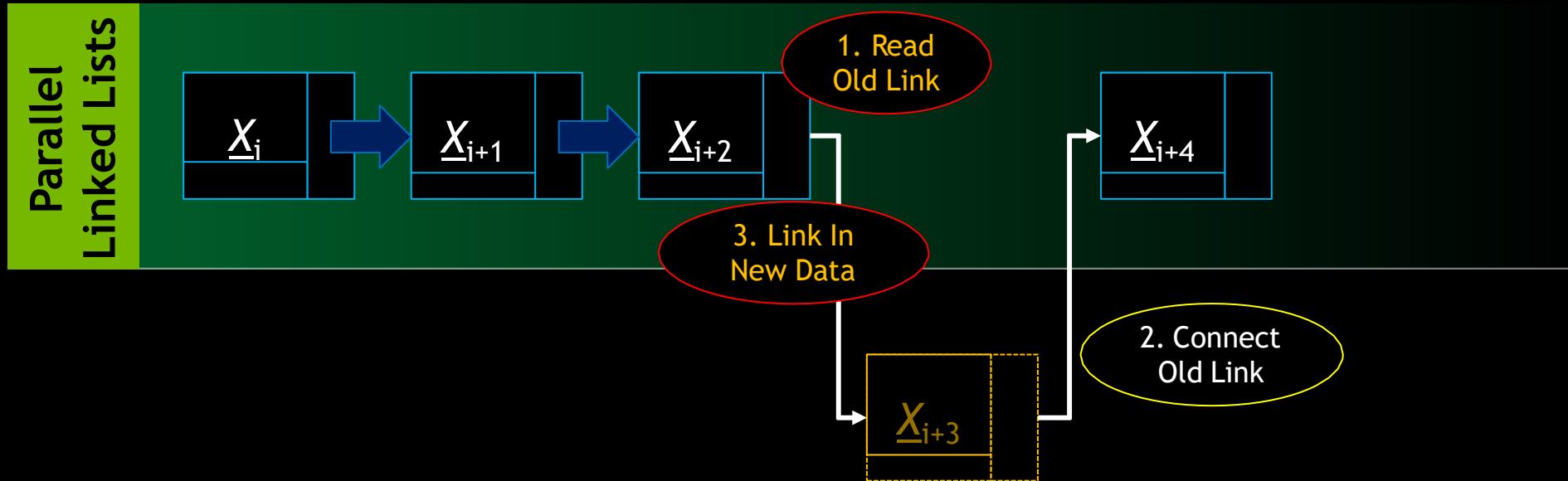
# Lock-Free Parallel Data Structures



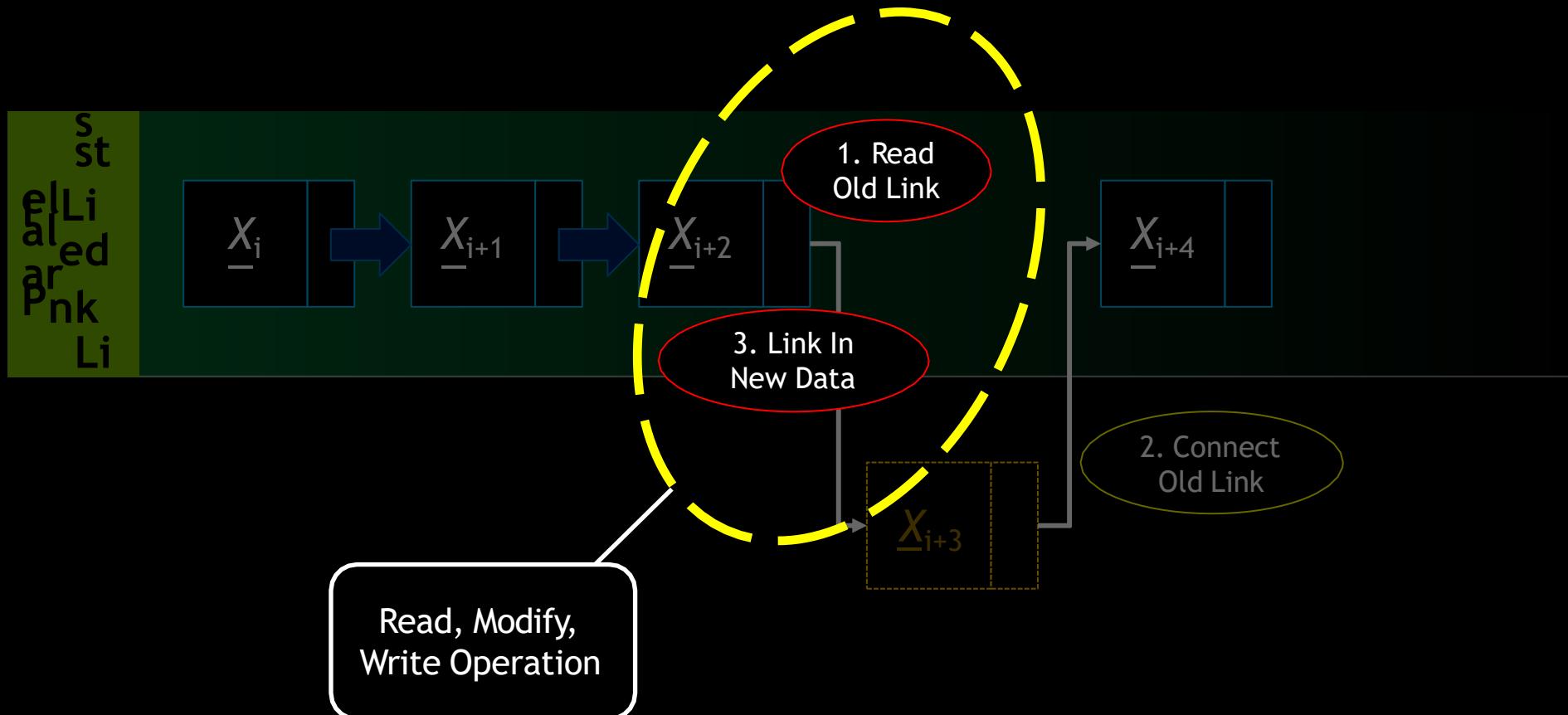
# Lock-Free Parallel Data Structures



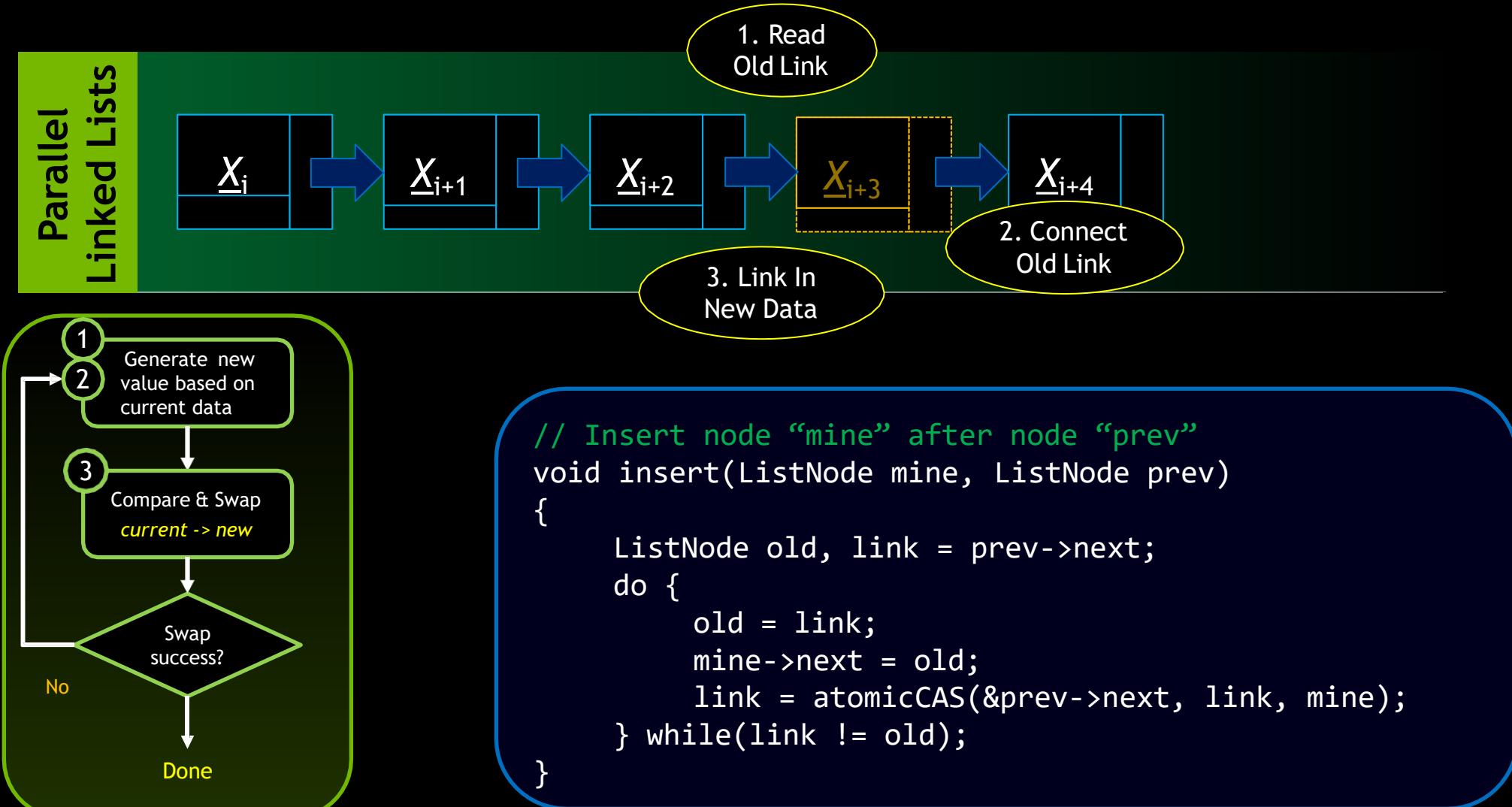
# Lock-Free Parallel Data Structures



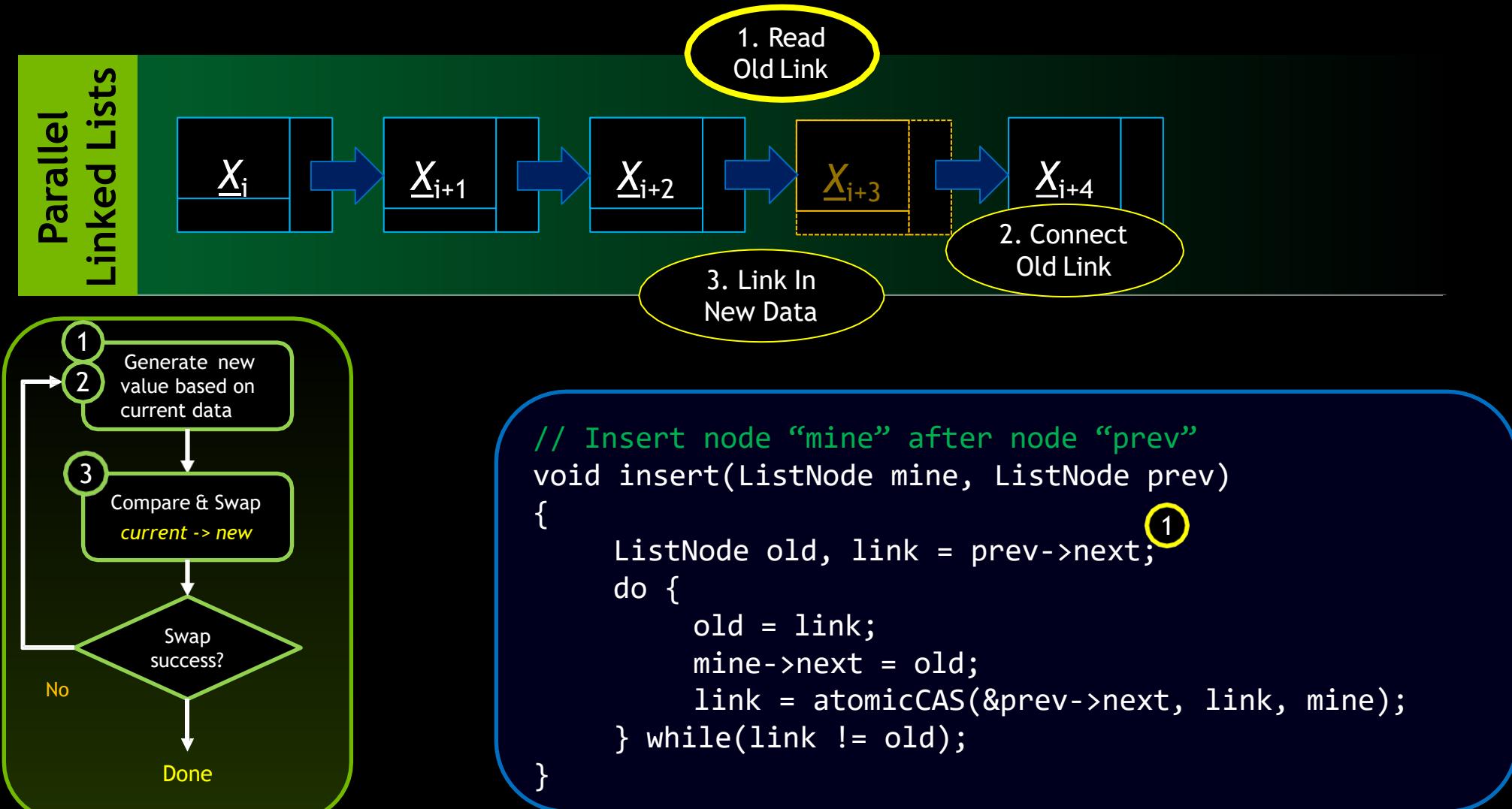
# Lock-Free Parallel Data Structures



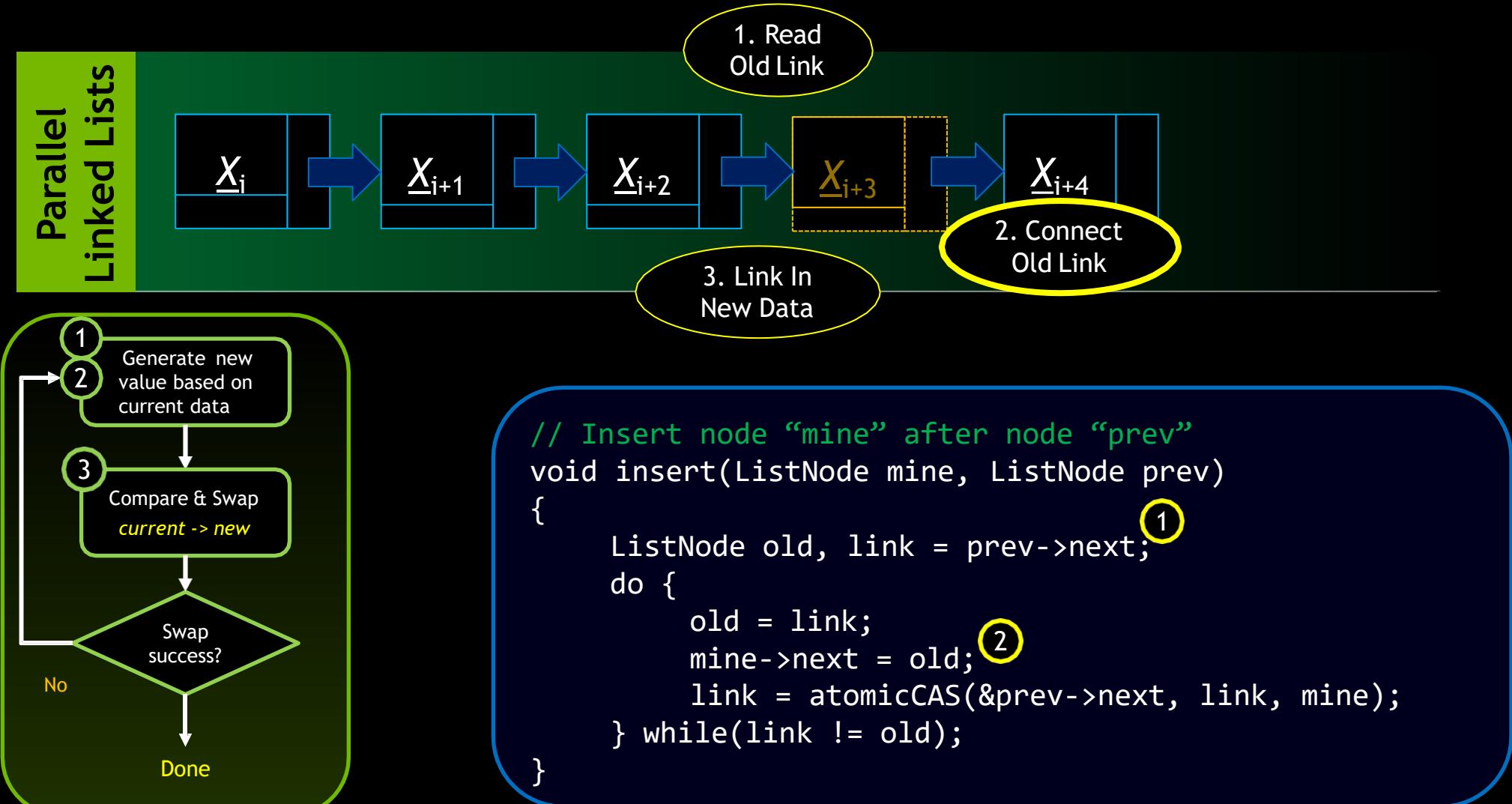
# Lock-Free Parallel Data Structures



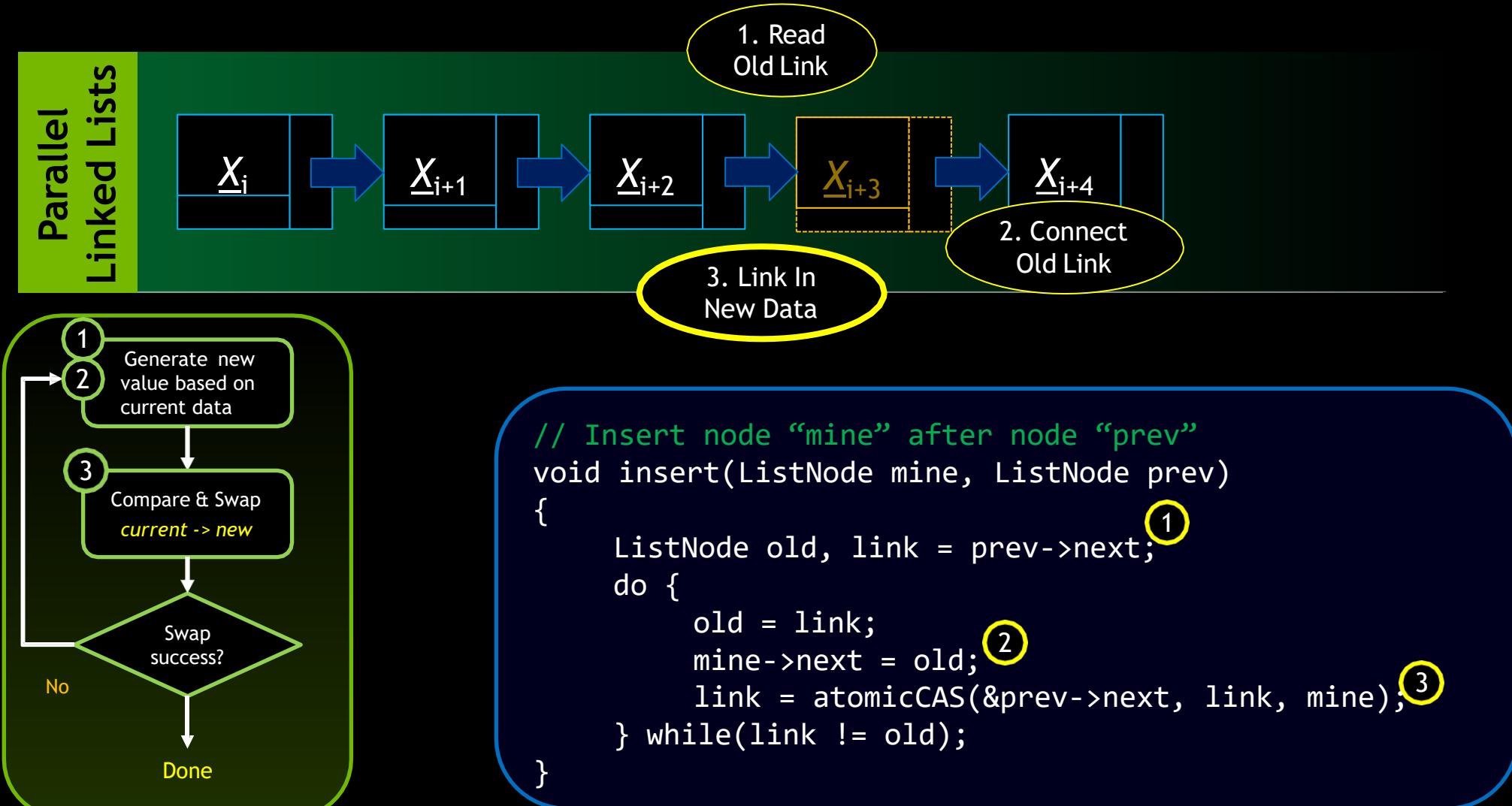
# Lock-Free Parallel Data Structures



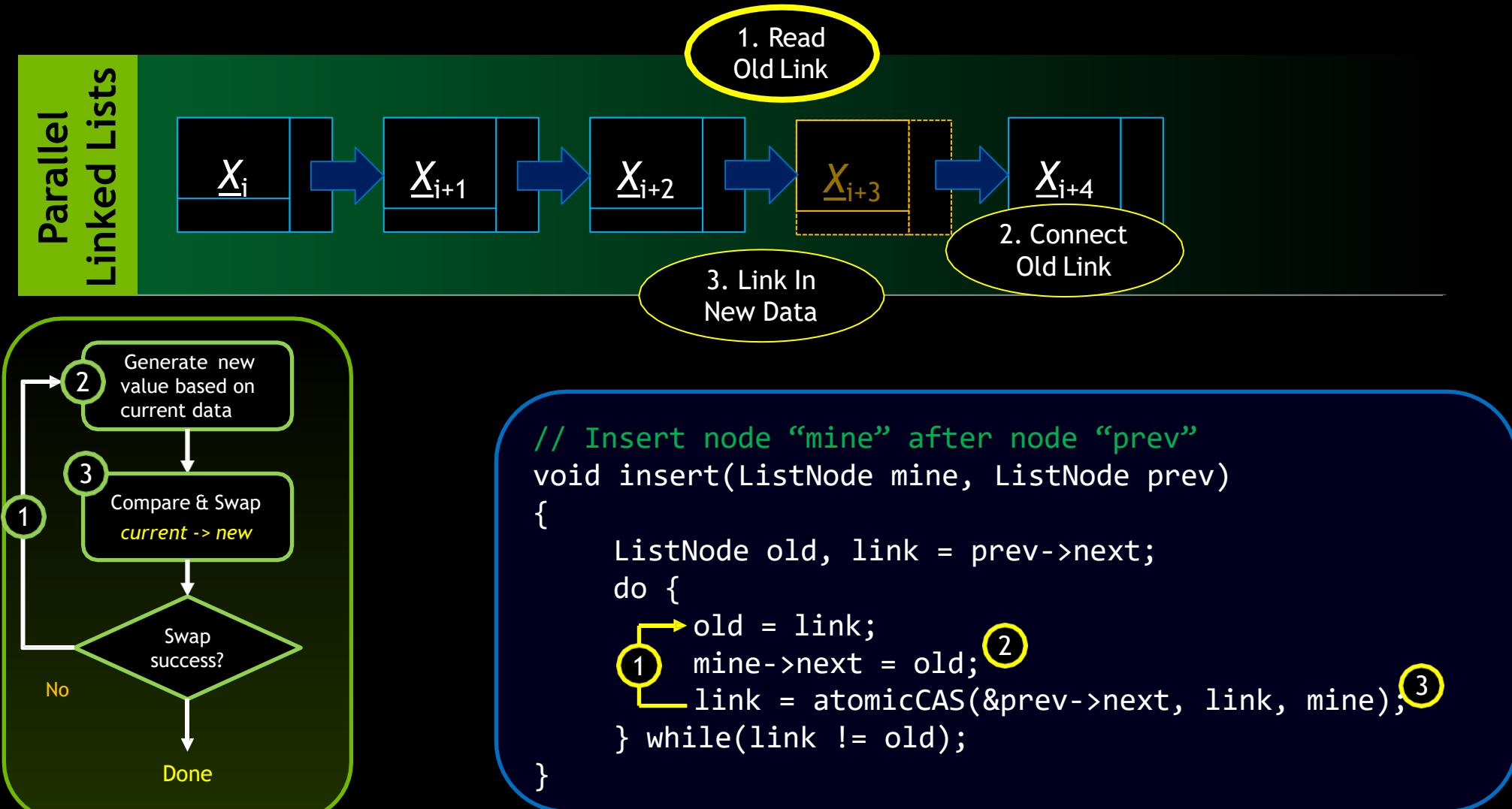
# Lock-Free Parallel Data Structures



# Lock-Free Parallel Data Structures



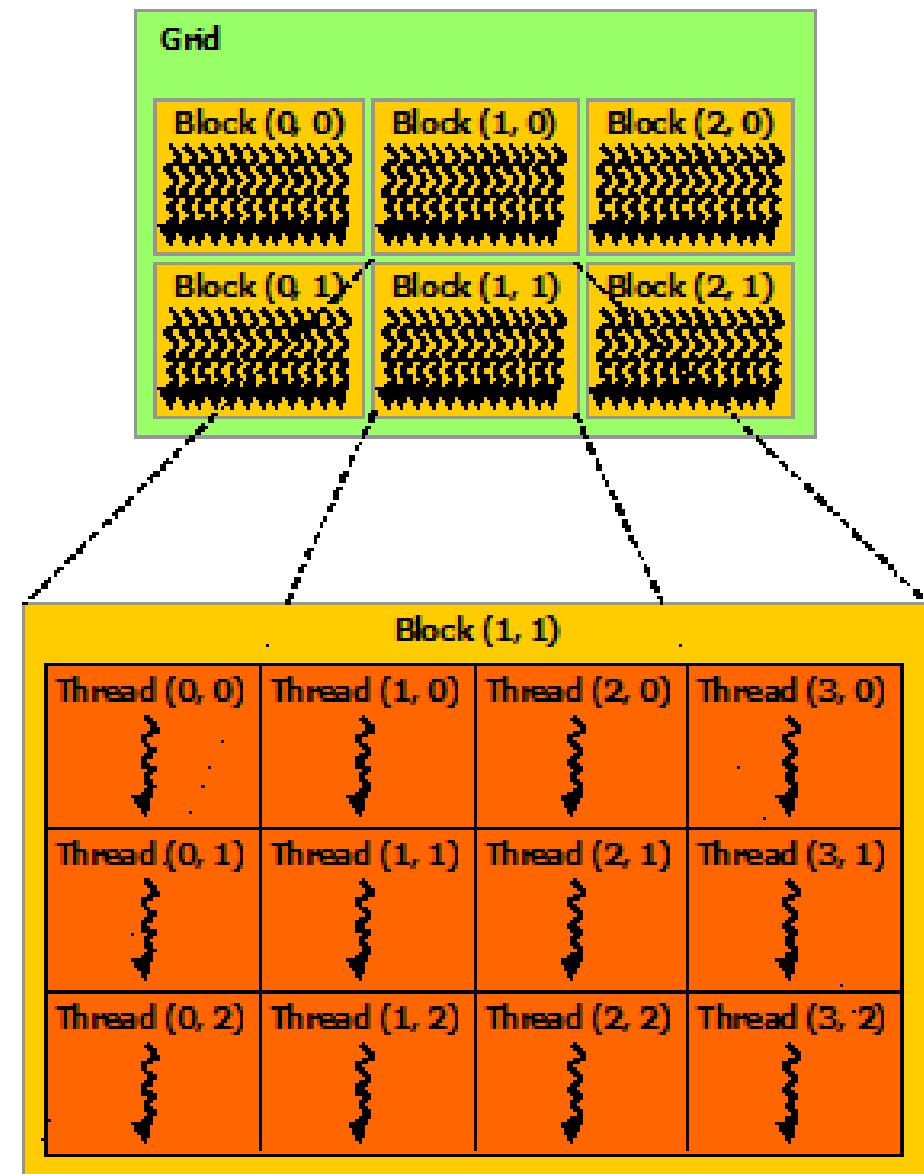
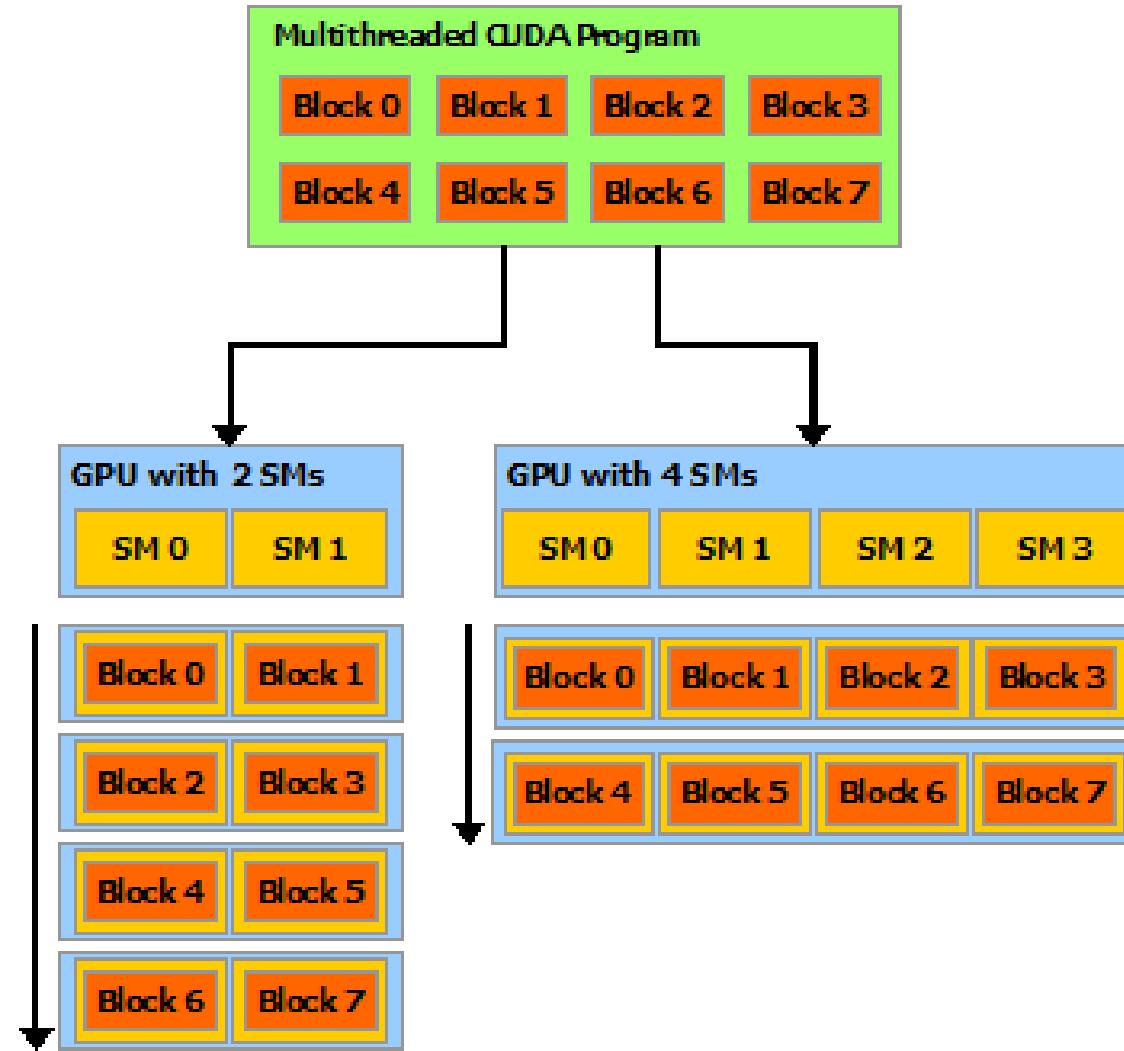
# Lock-Free Parallel Data Structures



# 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

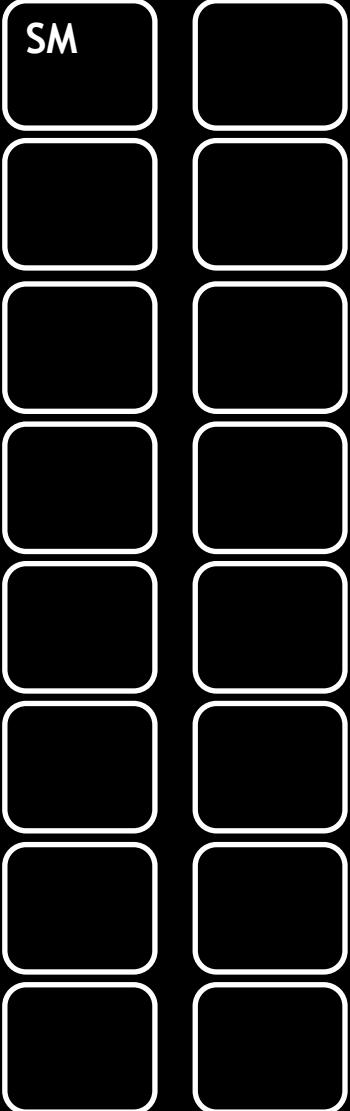
# Streaming Multiprocessors



# GPU

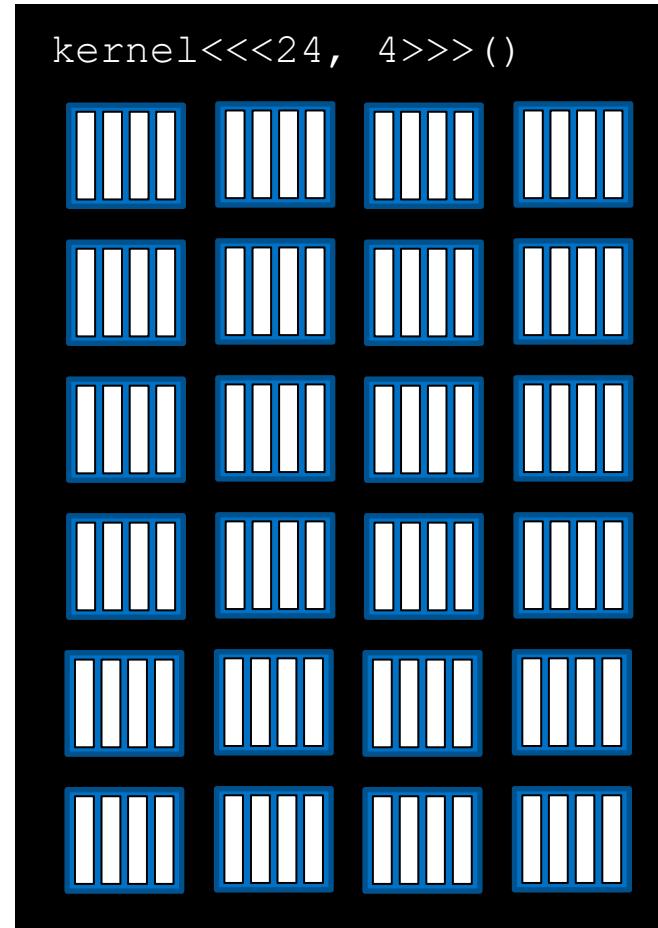
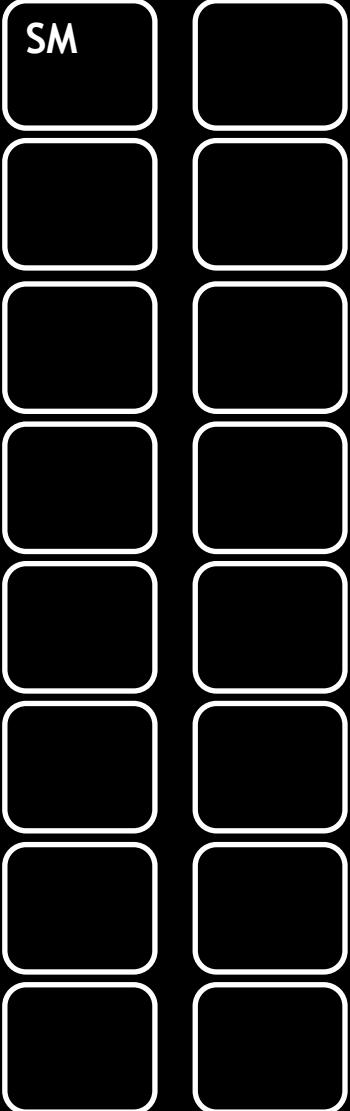
NVIDIA GPUs contain functional units called **Streaming Multiprocessors**, or **SMs**

# GPU



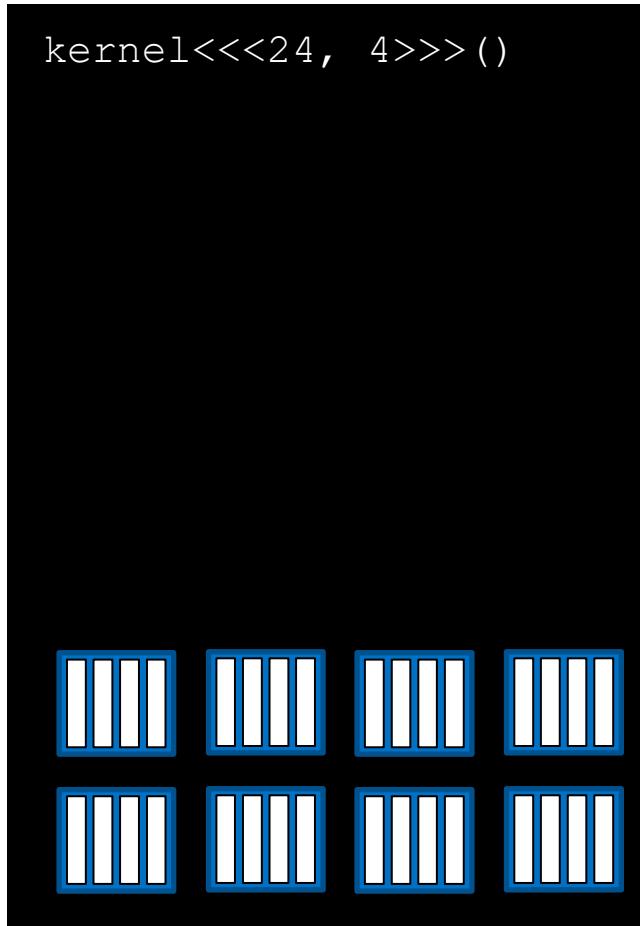
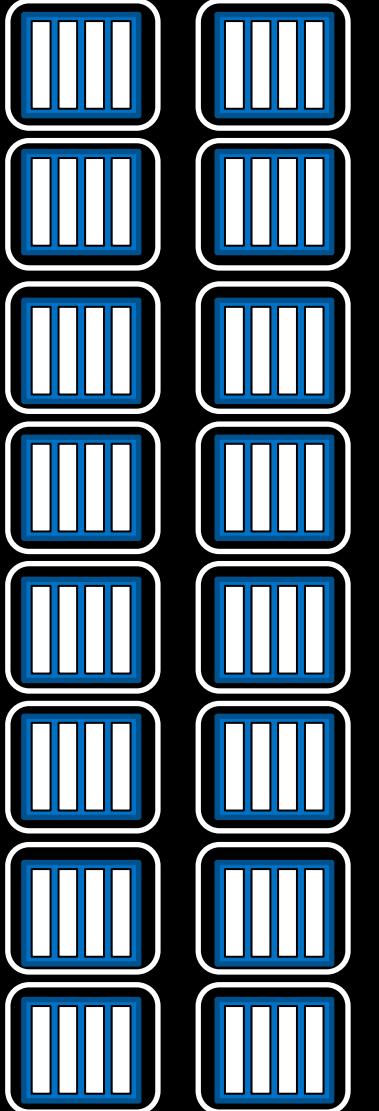
NVIDIA GPUs contain functional units called **Streaming Multiprocessors**, or **SMs**

# GPU



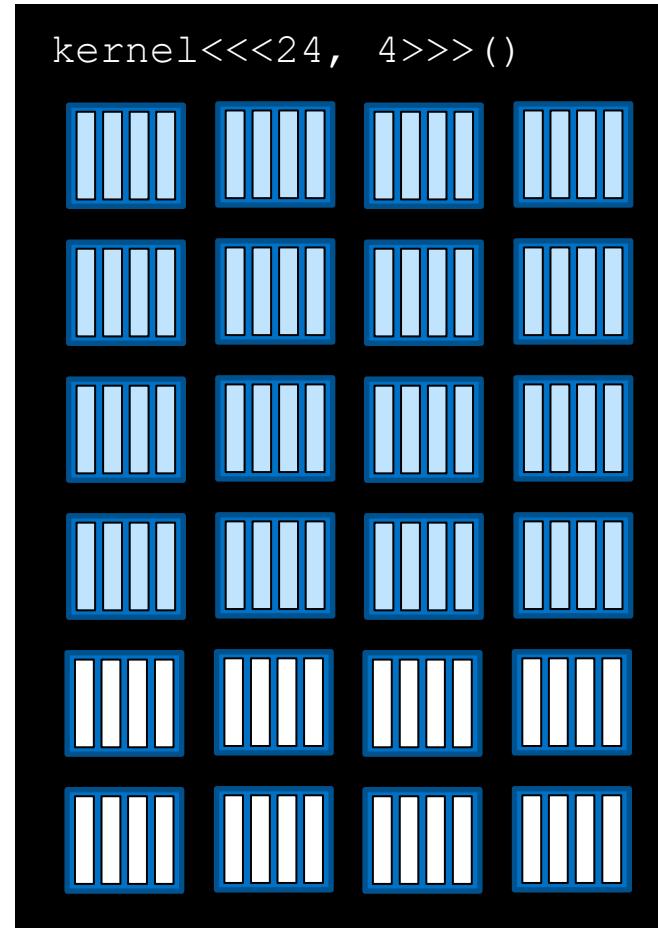
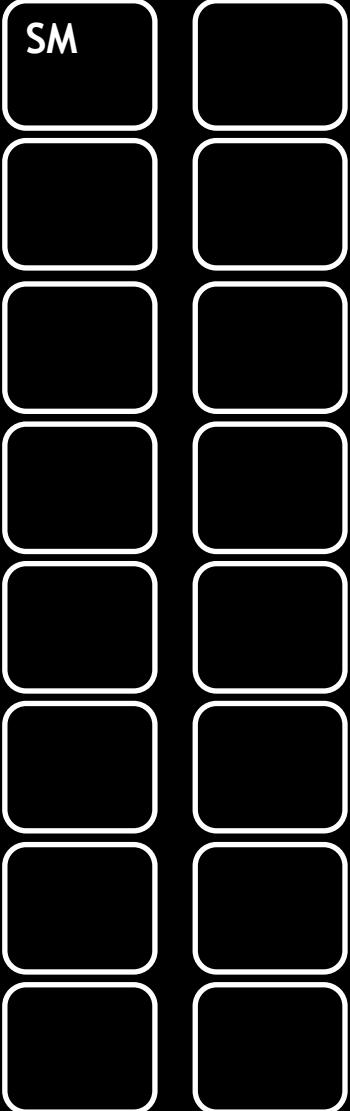
Blocks of threads are scheduled to run on SMs

# GPU



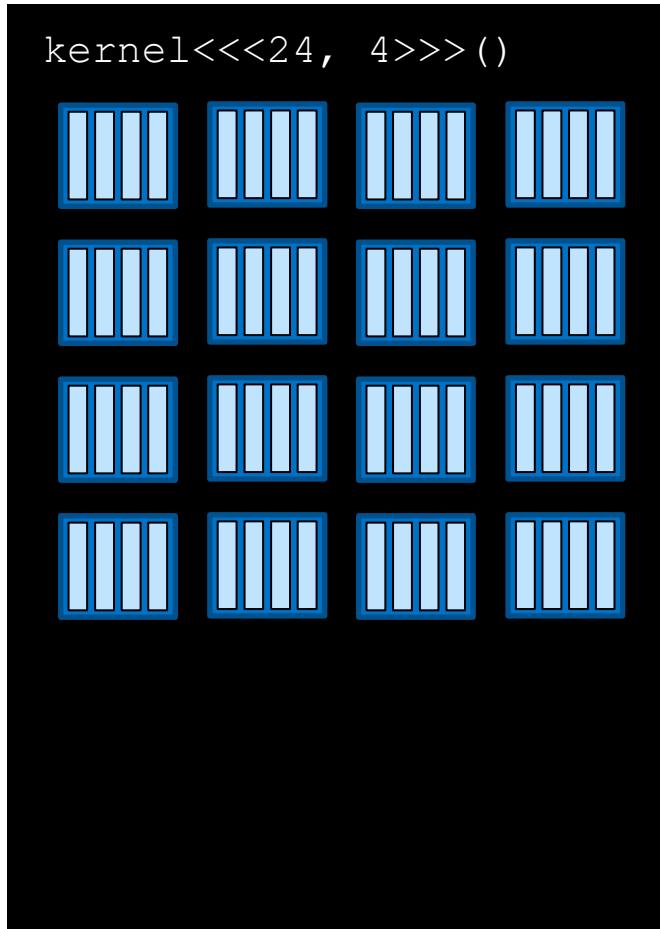
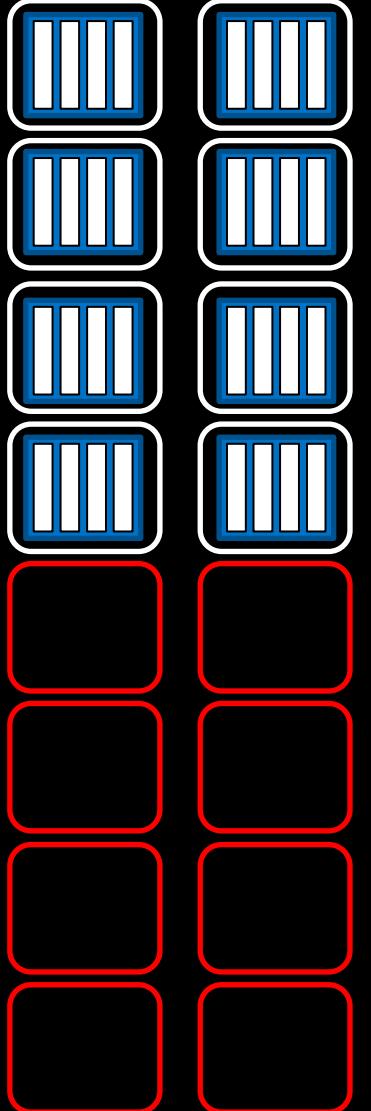
Depending on the number of SMs on a GPU, and the requirements of a block, more than one block can be scheduled on an SM

# GPU



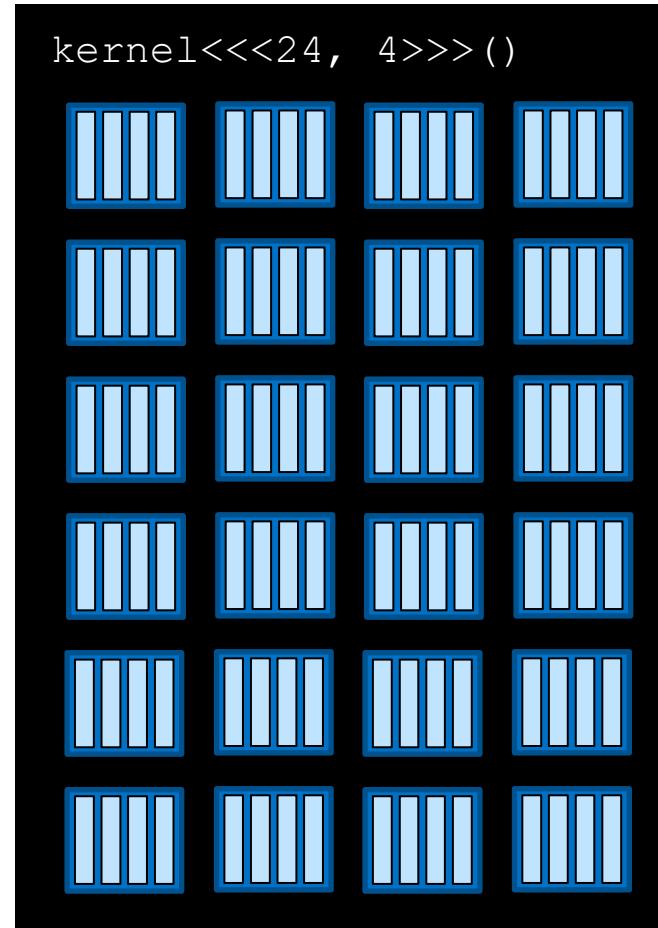
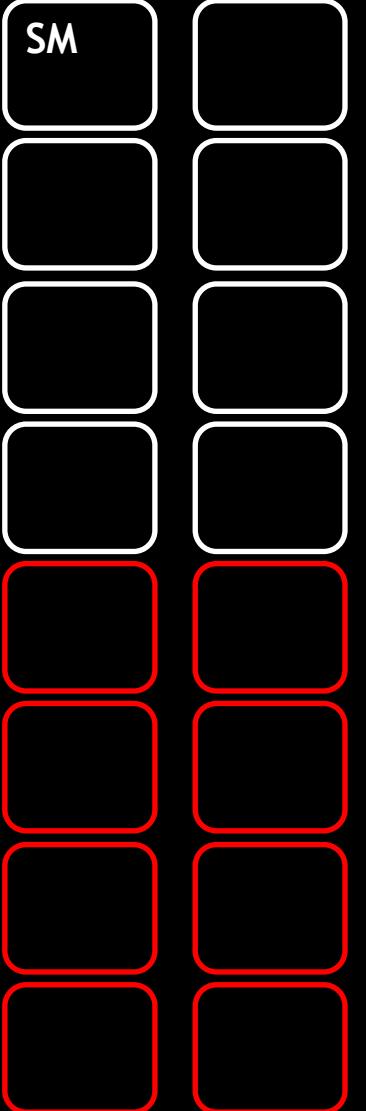
Depending on the number of SMs on a GPU, and the requirements of a block, more than one block can be scheduled on an SM

# GPU



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

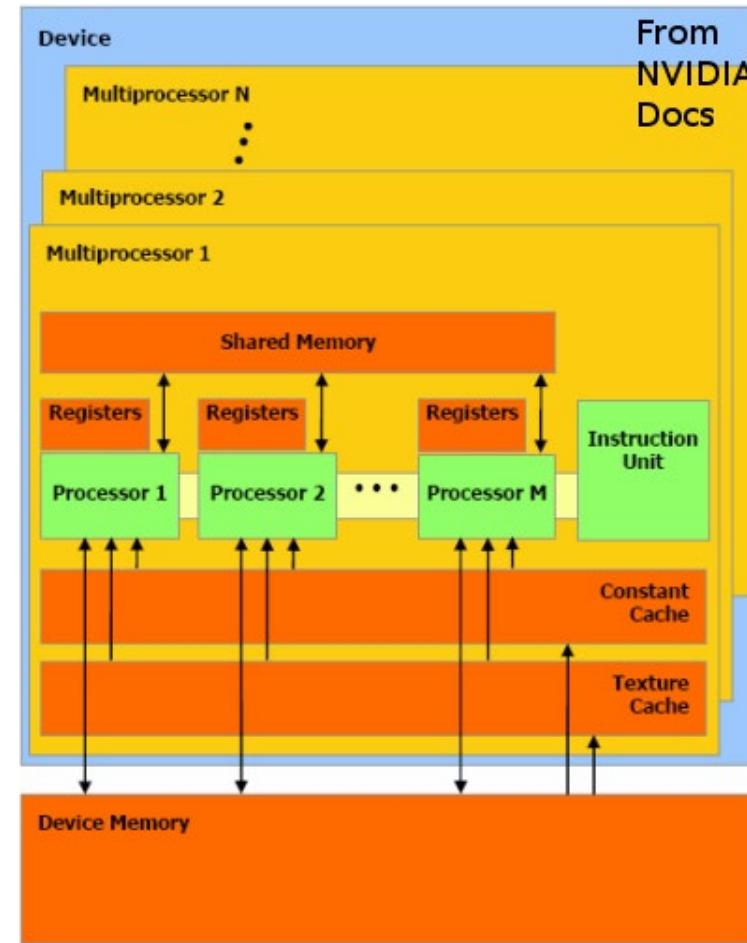
# GPU



Here there are fallow SMs

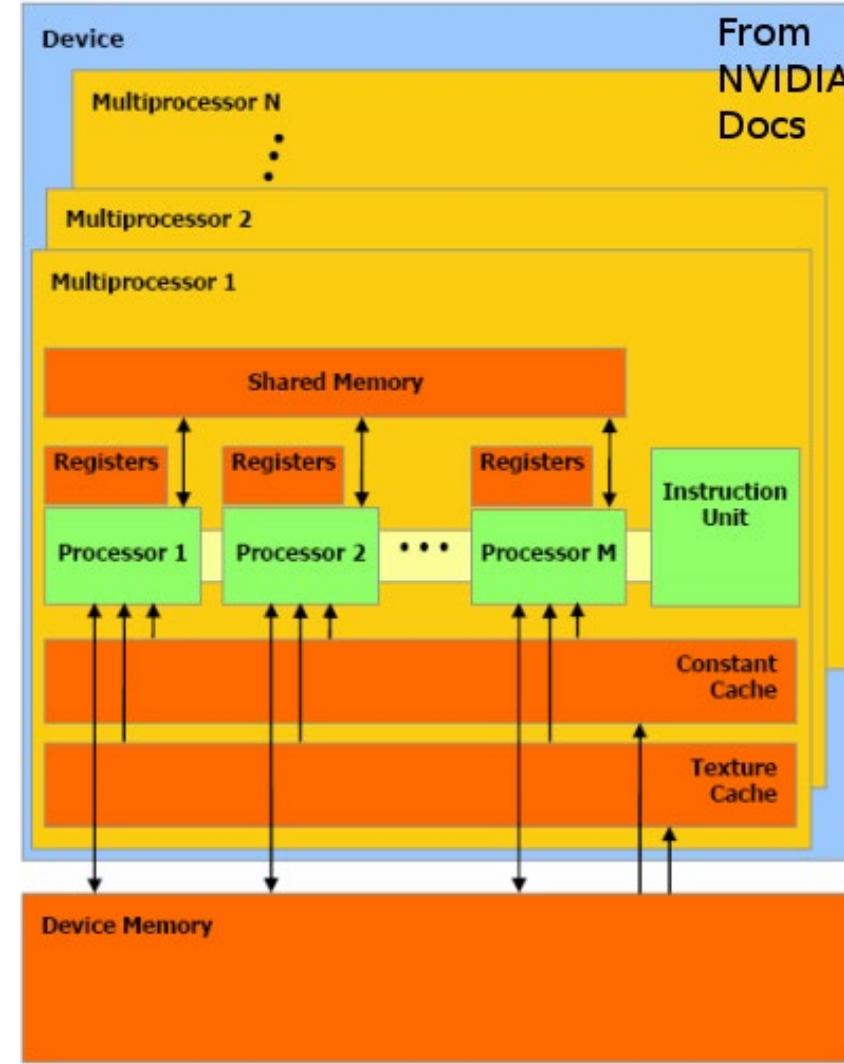
# Multiprocessor Occupancy

- 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



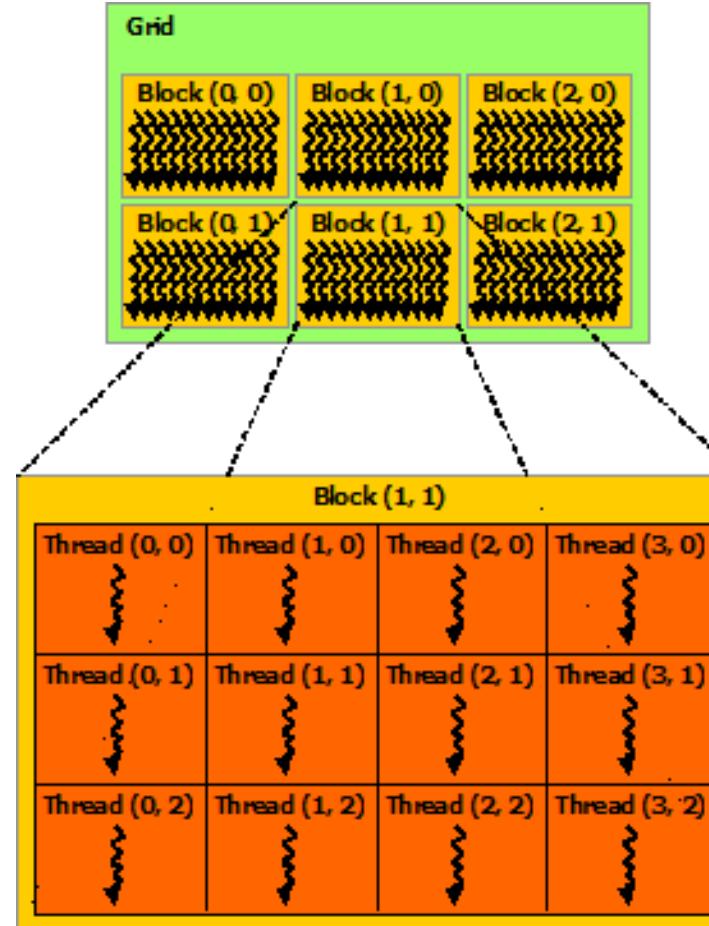
# Multiprocessor Occupancy

- 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	<a href="#">(Help)</a>
Registers Per Thread	16	<a href="#">(Help)</a>
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

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

Active Threads per Multiprocessor	1024	<a href="#">(Help)</a>
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	100%	

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

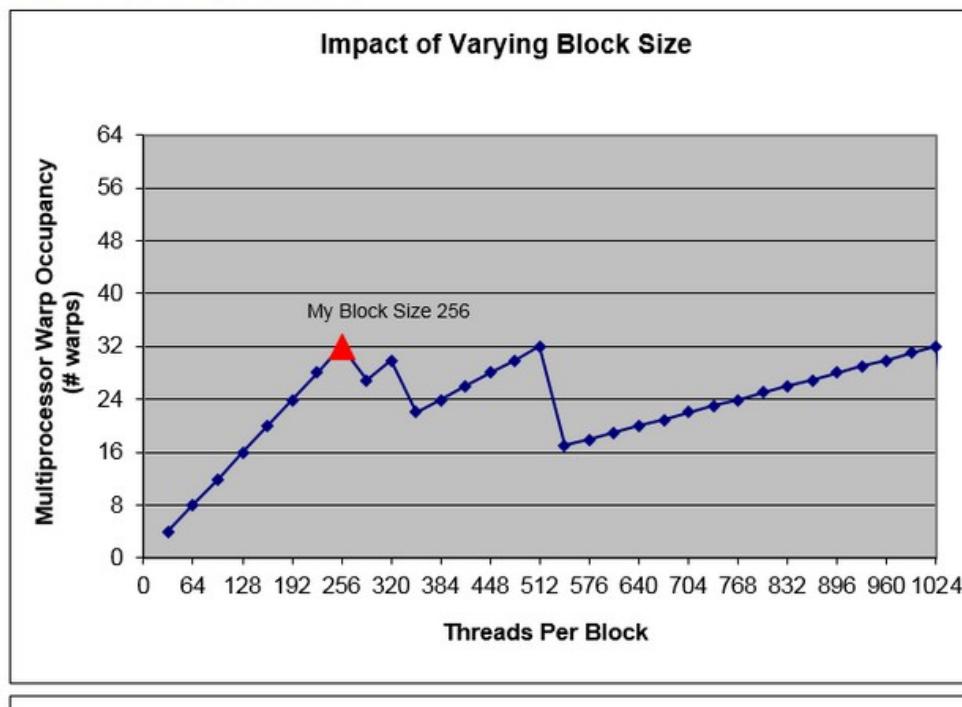
# CUDA Occupancy Calculator

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



## Tuning CUDA Applications for Volta

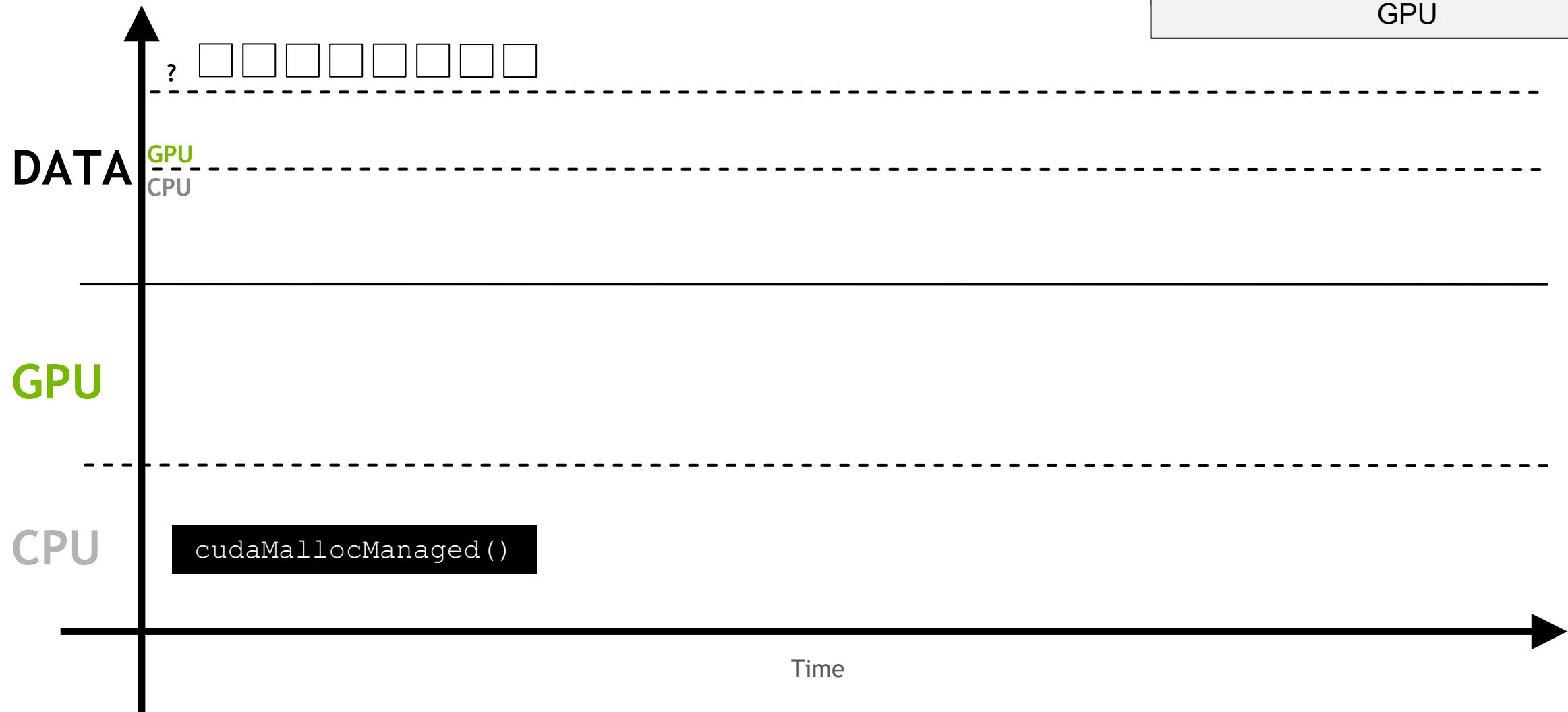
Application Note

# CUDA Samples / simpleOccupancy

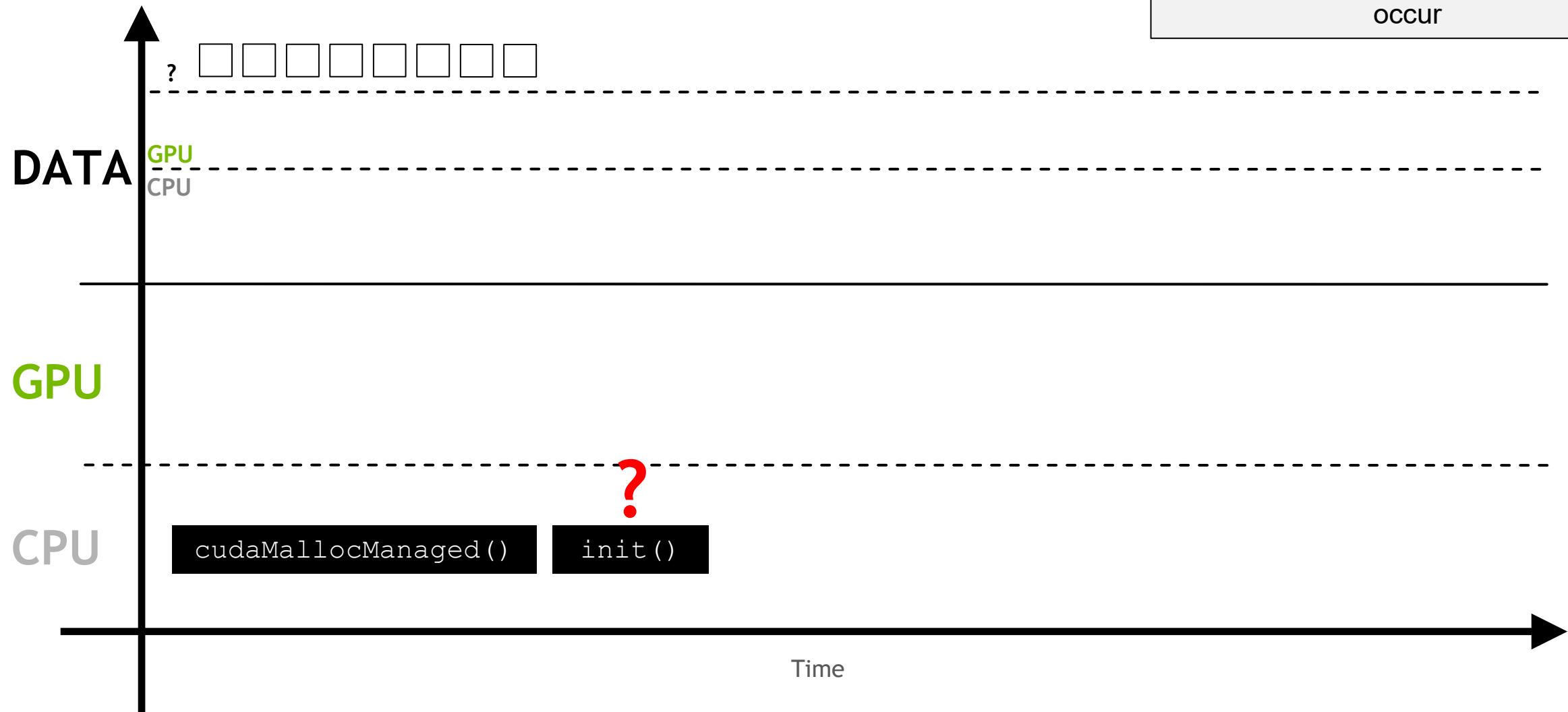
# CUDA Samples / zeroCopy

# Unified Memory Behavior

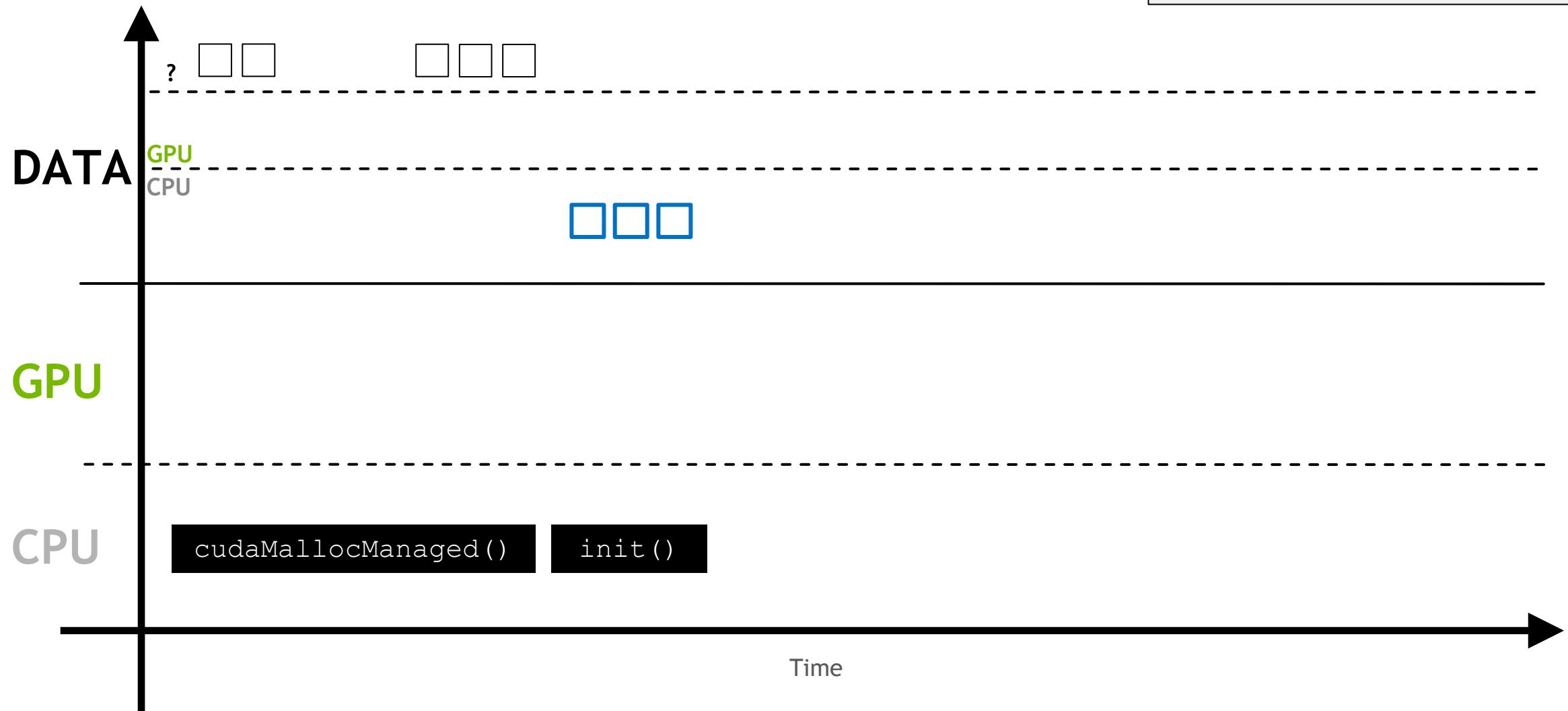
When **UM** is allocated, it may not be resident initially on the CPU or the GPU



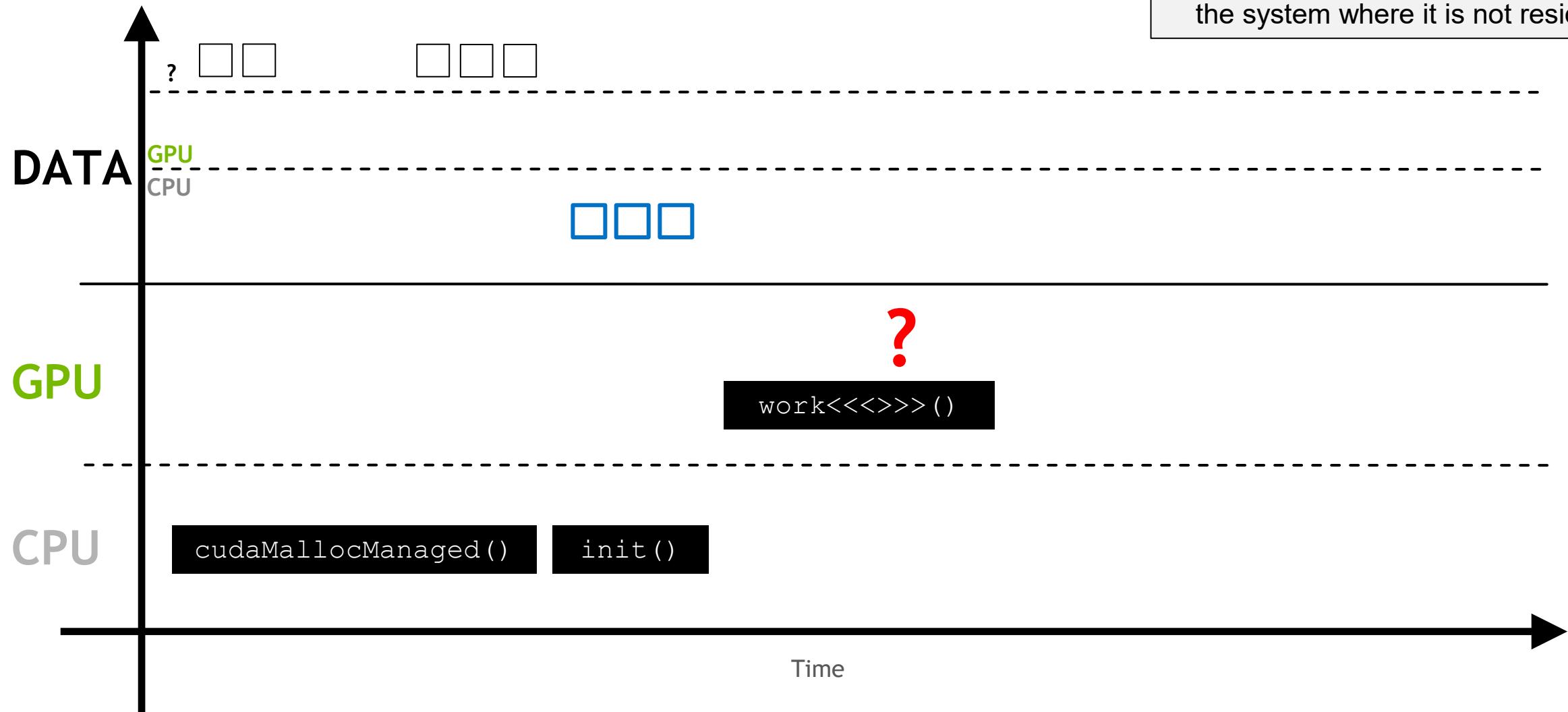
When some work asks for the memory  
for the first time, a **page fault** will  
occur



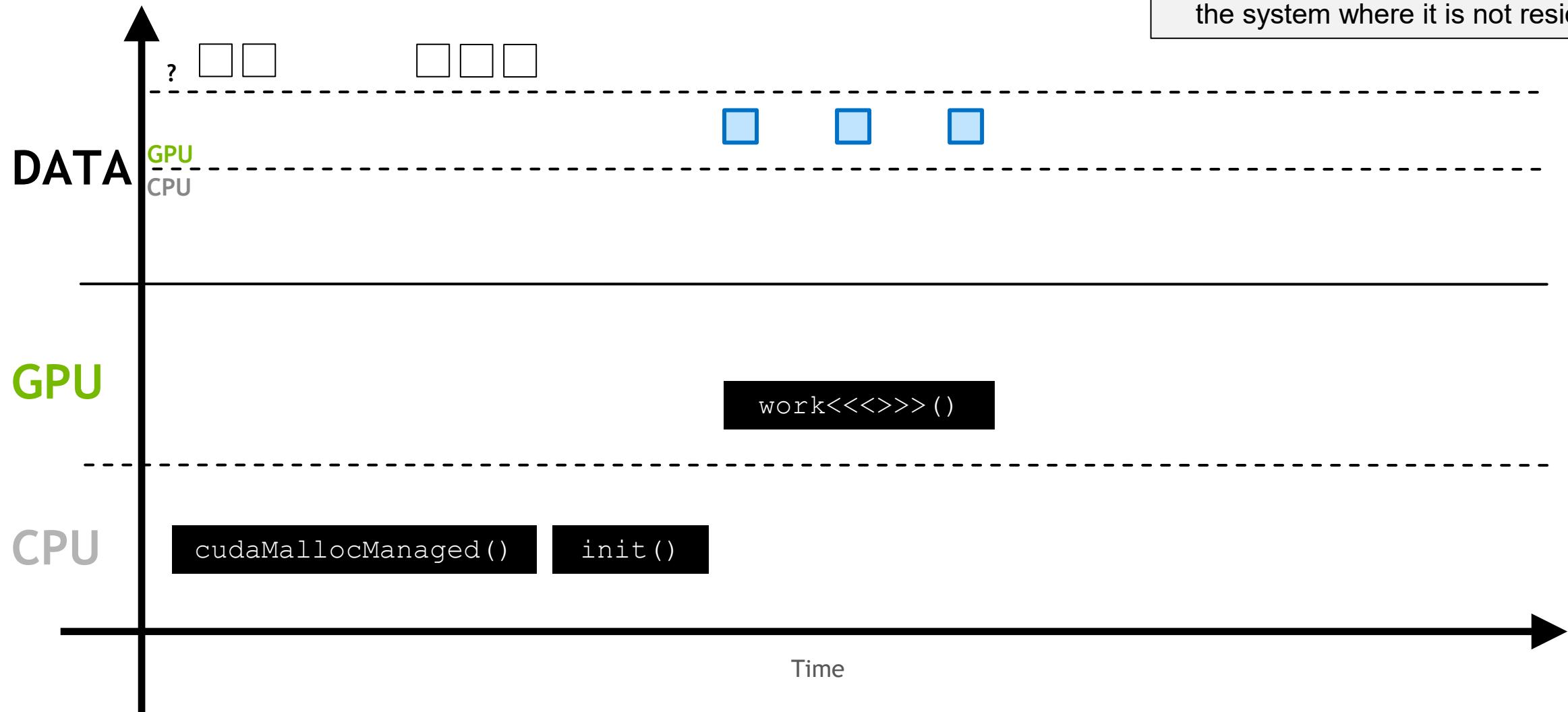
The page fault will trigger the migration  
of the demanded memory



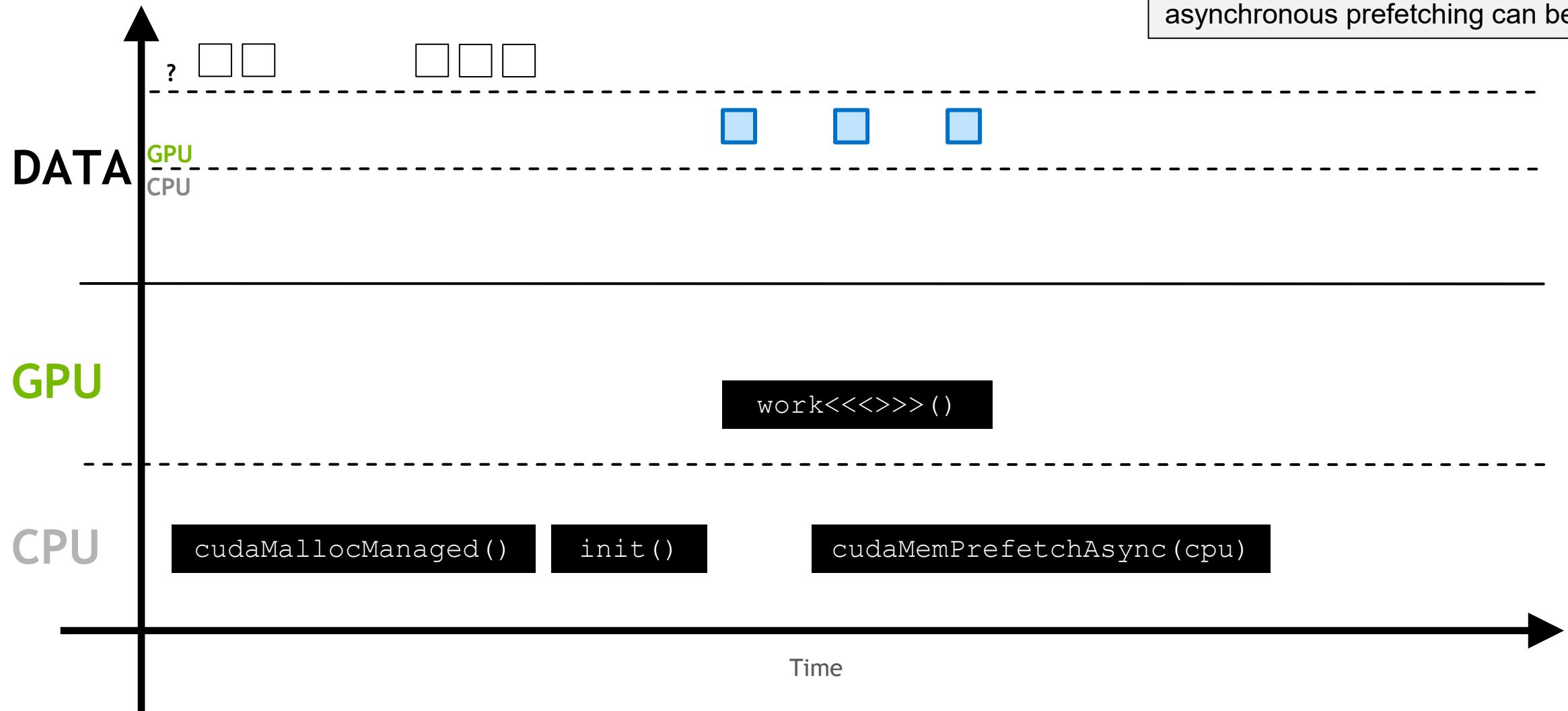
This process repeats anytime the memory is requested somewhere in the system where it is not resident



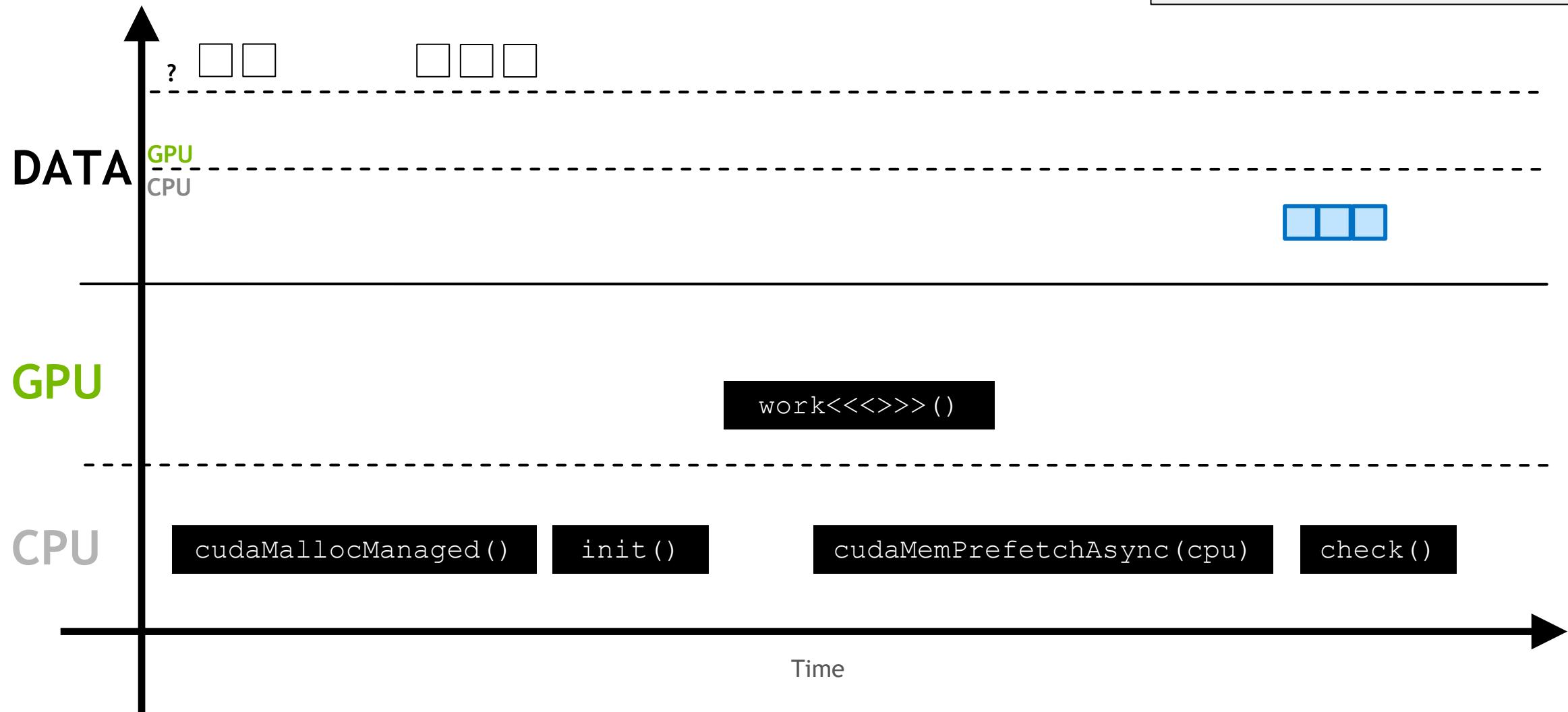
This process repeats anytime the memory is requested somewhere in the system where it is not resident

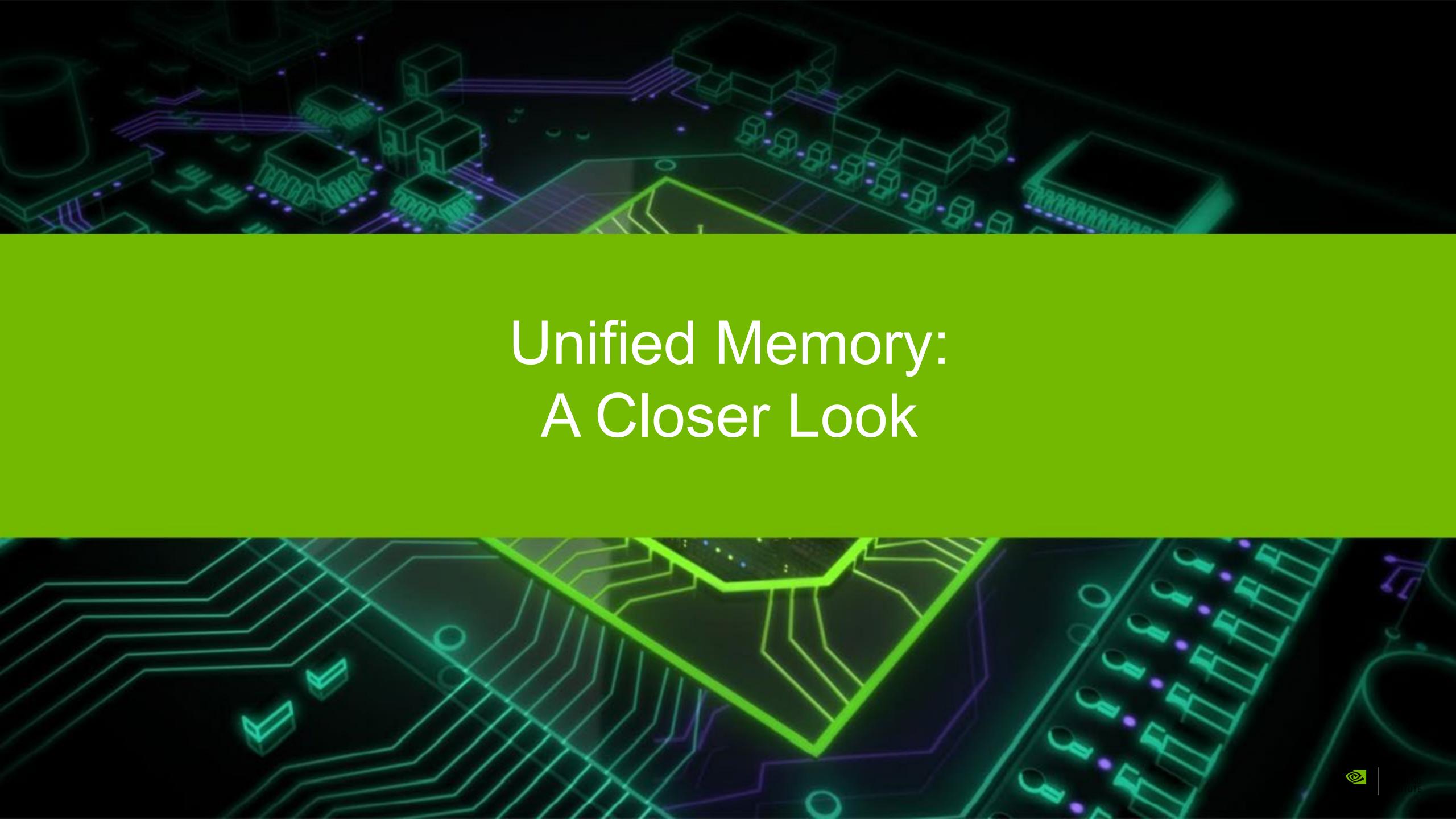


If it is known that the memory **will be** accessed somewhere it is not resident, asynchronous prefetching can be used



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

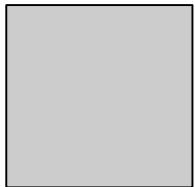
```
char **data;  
// allocate and initialize data on the CPU  
  
char **d_data;  
char **h_data = (char**)malloc(N*sizeof(char*));  
for (int i = 0; i < N; i++) {  
    cudaMalloc(&h_data[i], N);  
    cudaMemcpy(h_data[i], data[i], N, ...);  
}  
cudaMalloc(&d_data, N*sizeof(char*));  
cudaMemcpy(d_data,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

### GPU code w/ Unified Memory

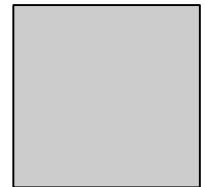
```
char **data;  
// allocate and initialize data on the CPU  
  
gpu_func<<<...>>>(data, N);
```

# UNIFIED MEMORY BASICS

GPU A

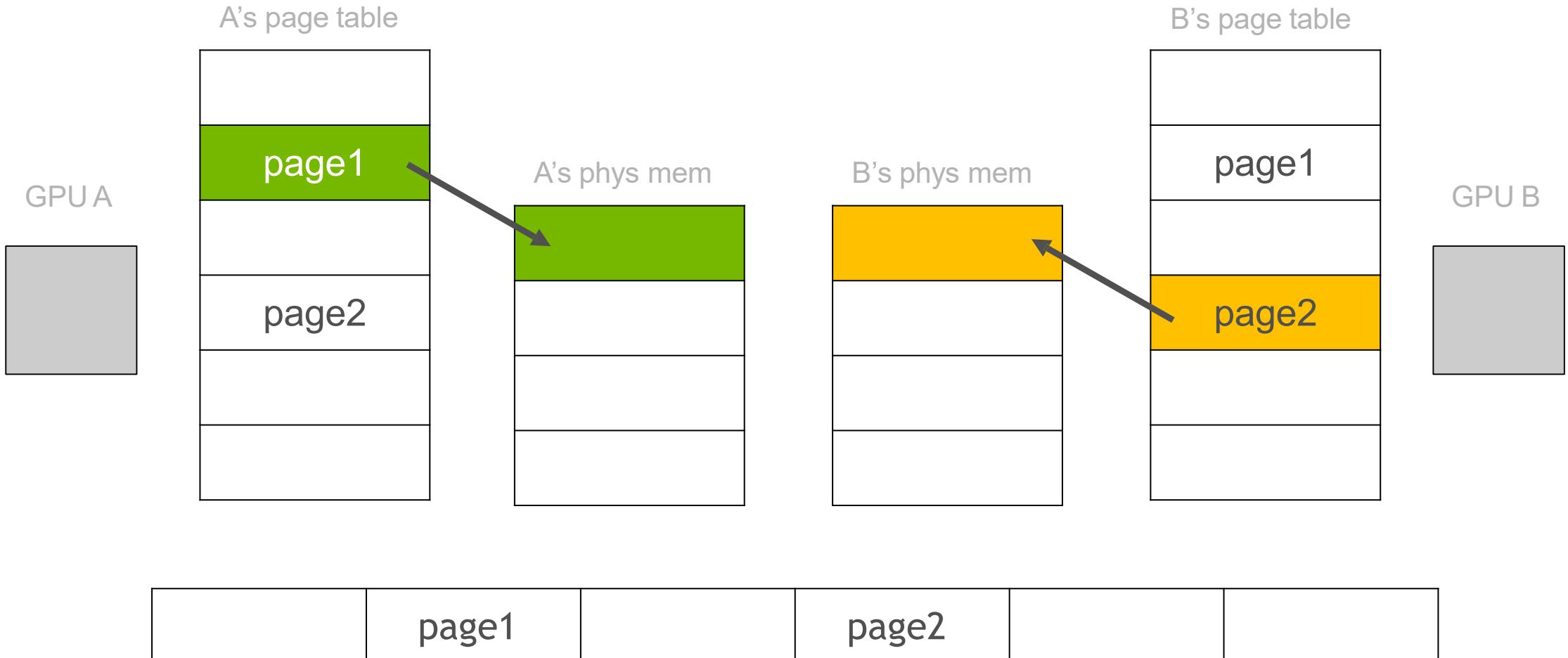


GPU B



Single virtual memory shared between processors

# UNIFIED MEMORY BASICS



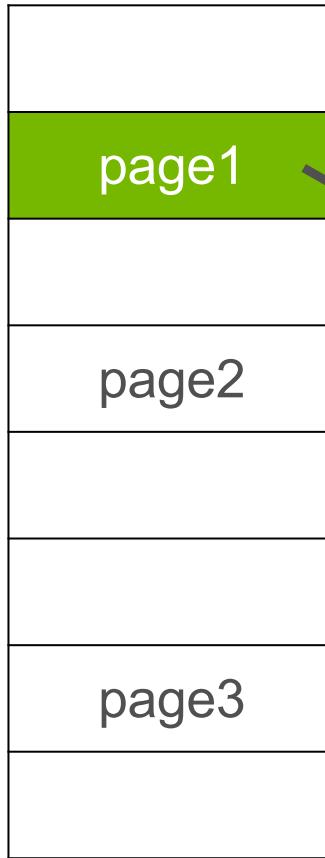
Single virtual memory shared between processors

# UNIFIED MEMORY BASICS



# UNIFIED MEMORY BASICS

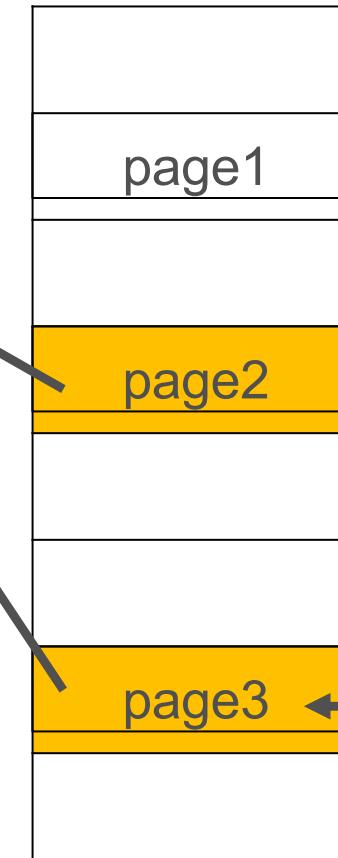
A's page table



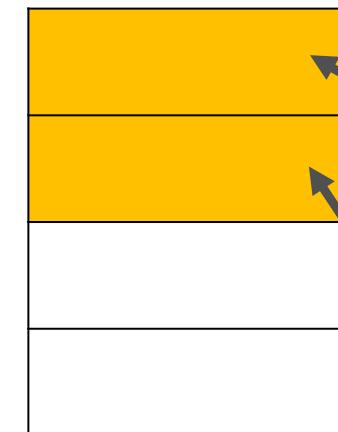
A's phys mem



B's page table



B's phys mem



\*addr3 = 1  
access replay

page3 populated and mapped into B's memory

# UNIFIED MEMORY BASICS

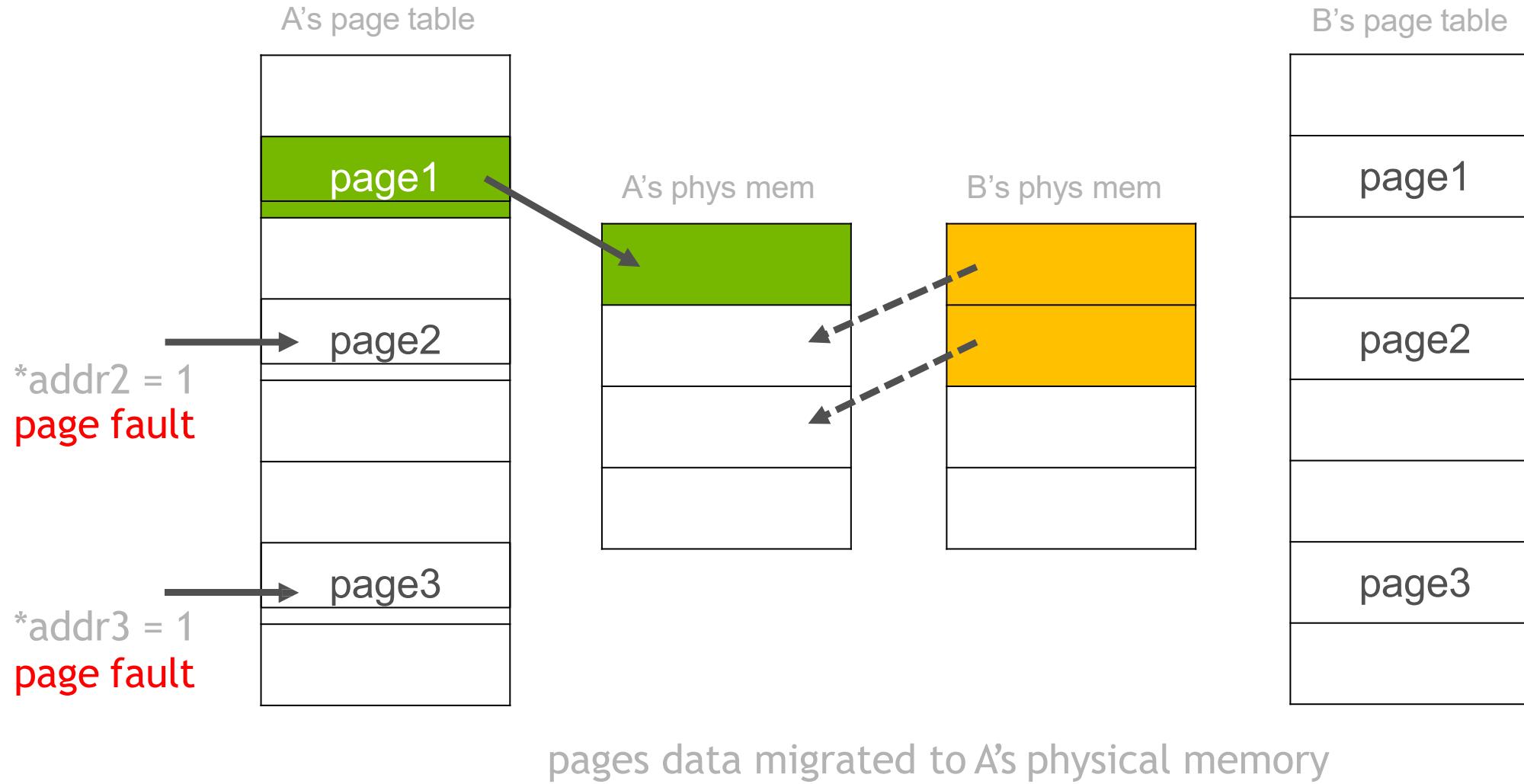


# UNIFIED MEMORY BASICS

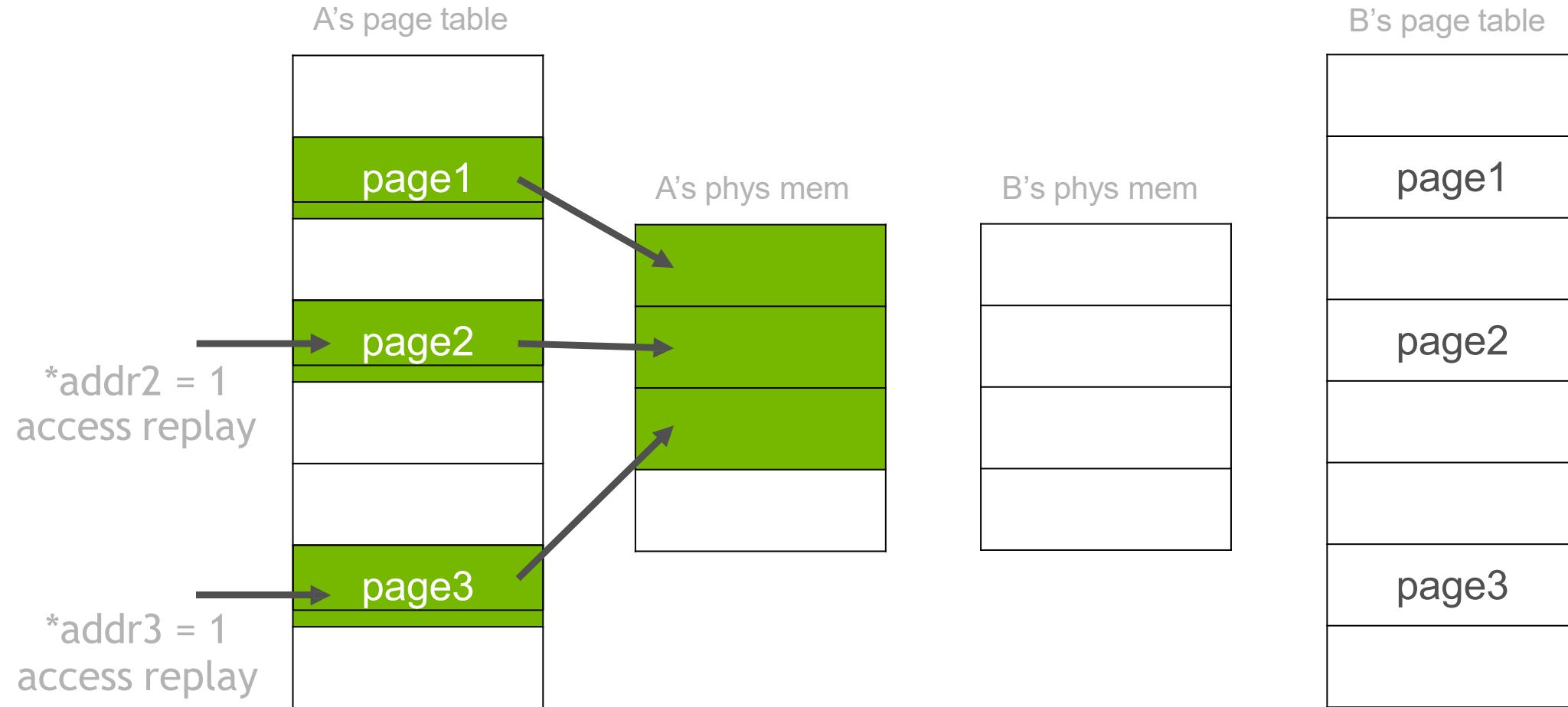


page2 and page3 unmapped from B's memory

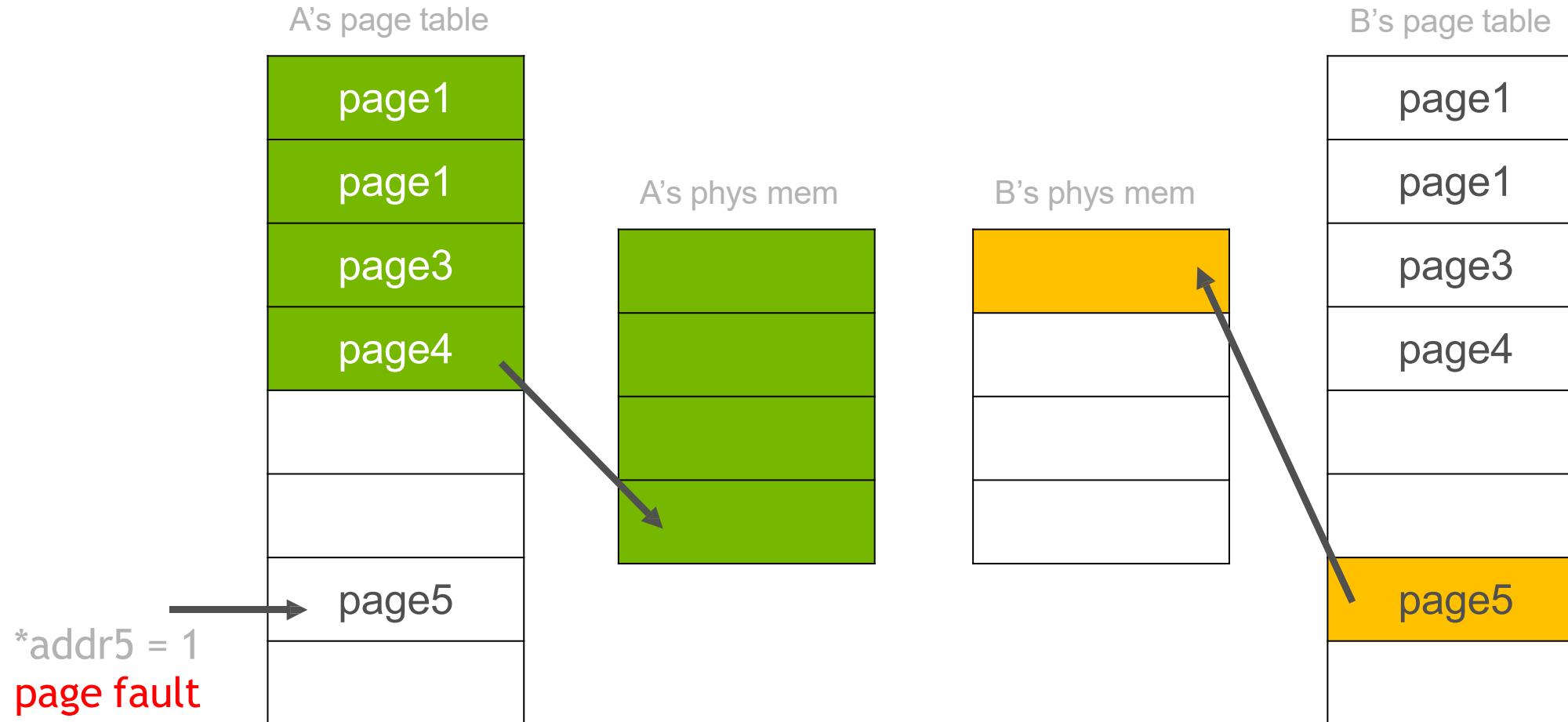
# UNIFIED MEMORY BASICS



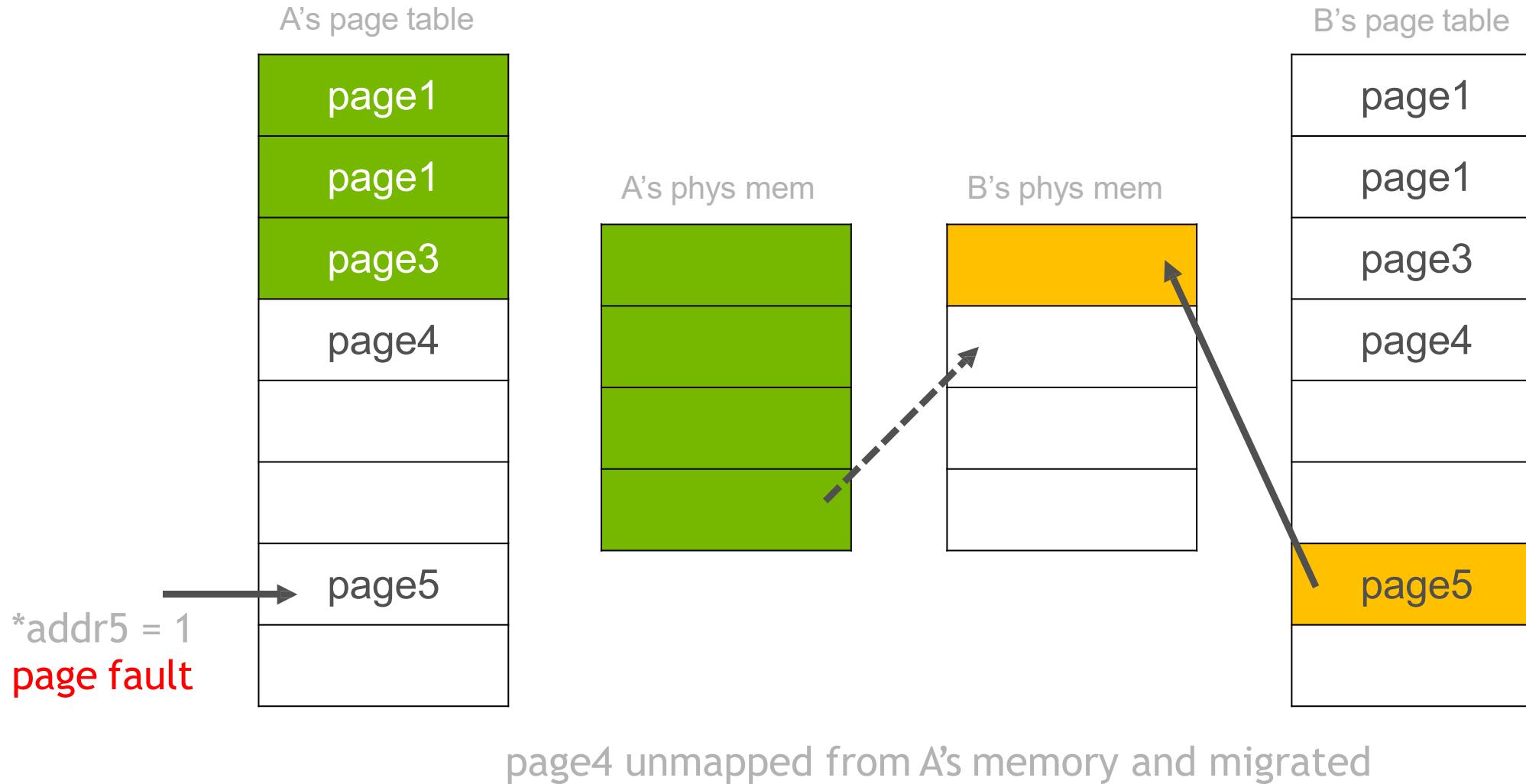
# UNIFIED MEMORY BASICS



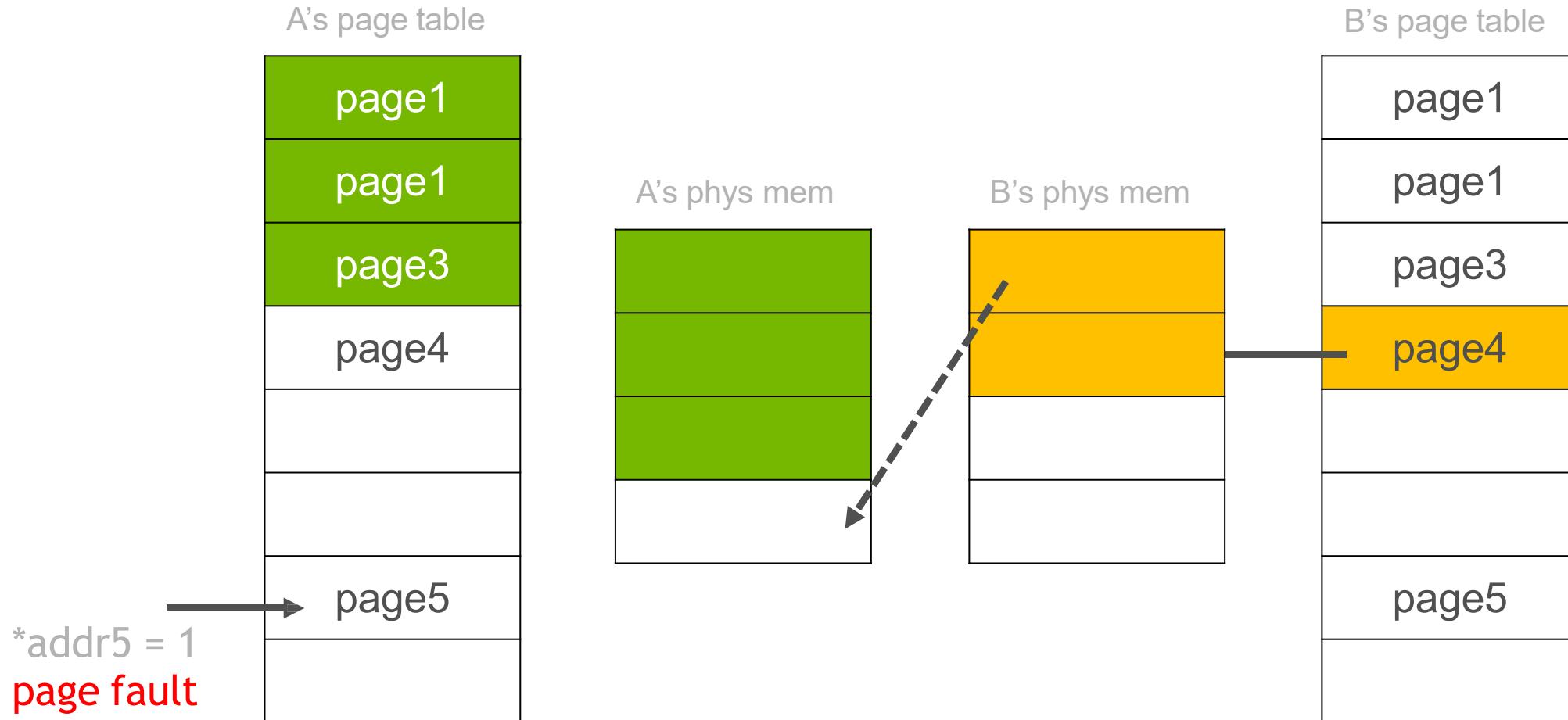
# MEMORY OVERSUBSCRIPTION



# MEMORY OVERSUBSCRIPTION



# MEMORY OVERSUBSCRIPTION



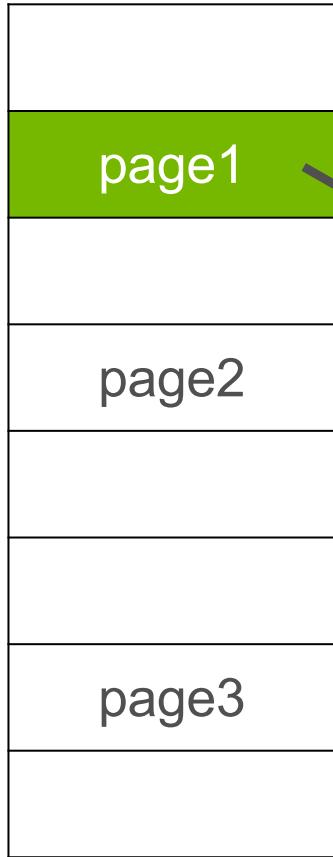
page4 mapped in B's memory, page5 unmapped and migrated to A

# MEMORY OVERSUBSCRIPTION



# CONCURRENT ACCESS

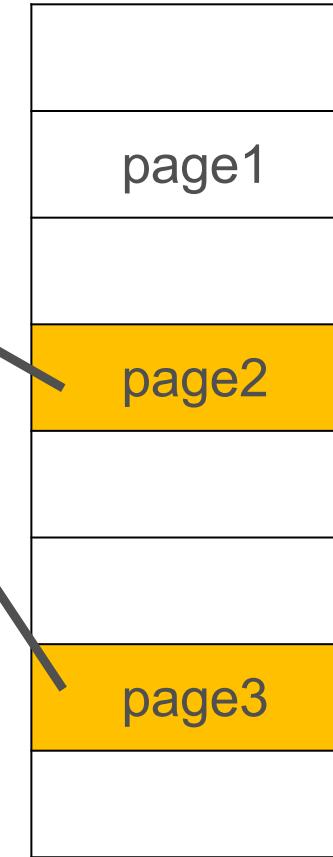
A's page table



A's phys mem



B's page table

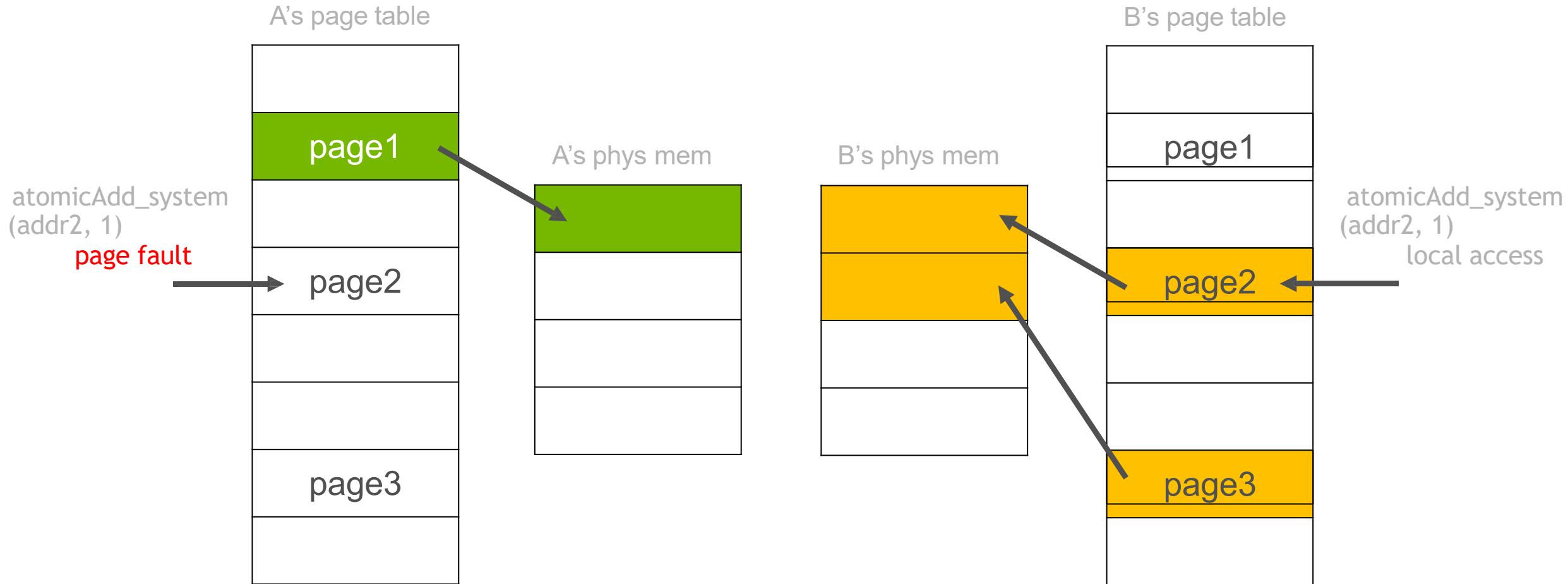


B's phys mem



# CONCURRENT ACCESS

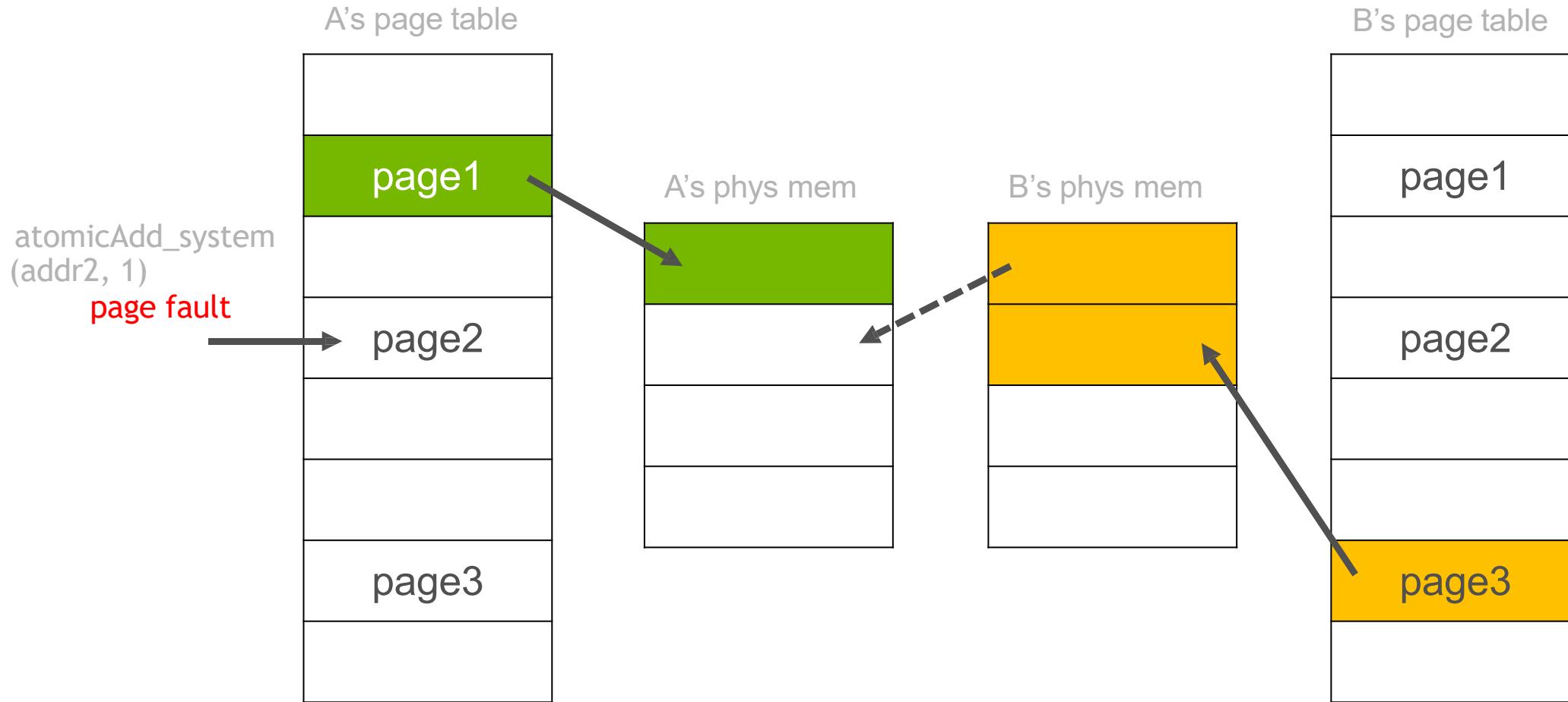
## Exclusive Access\*



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

# CONCURRENT ACCESS

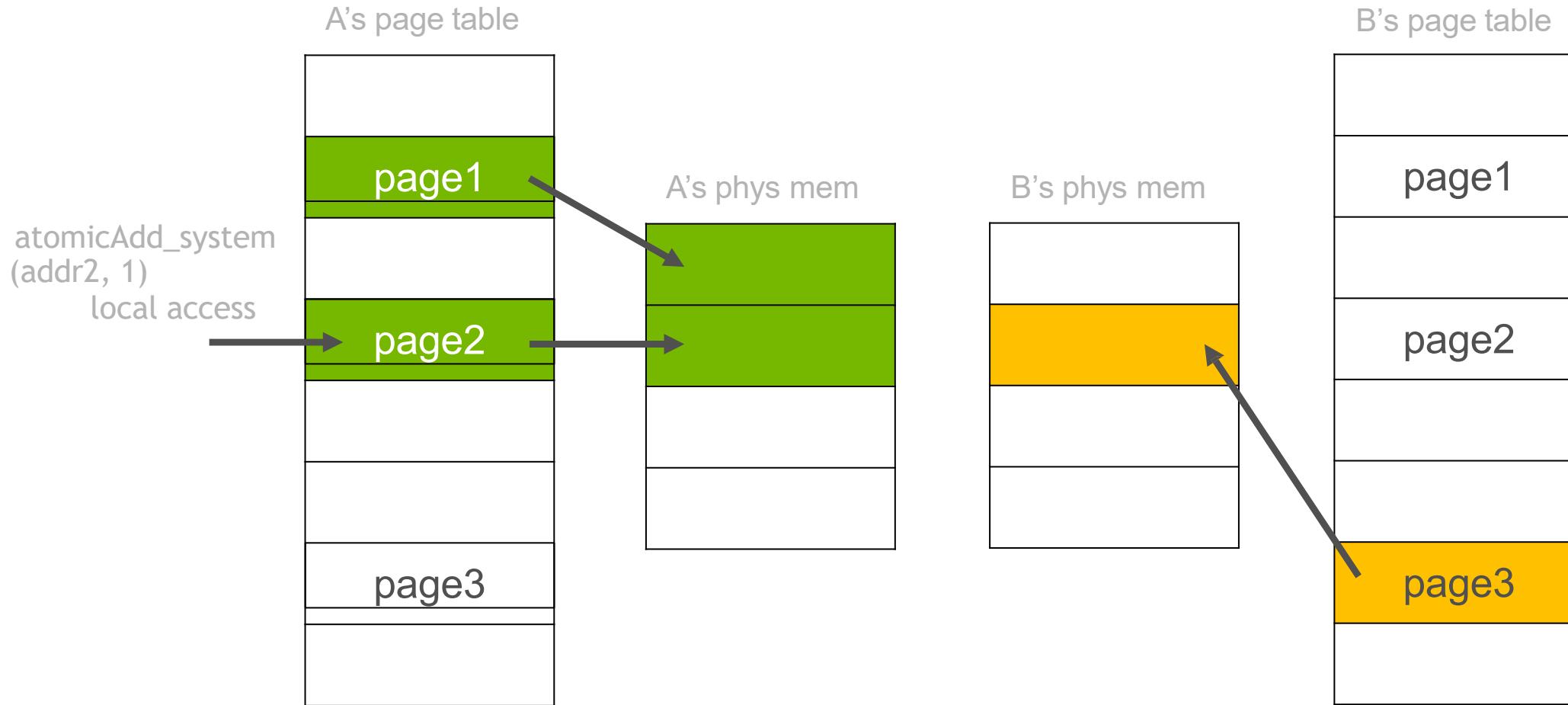
## Exclusive Access



page2 unmapped in B's memory and migrated to A

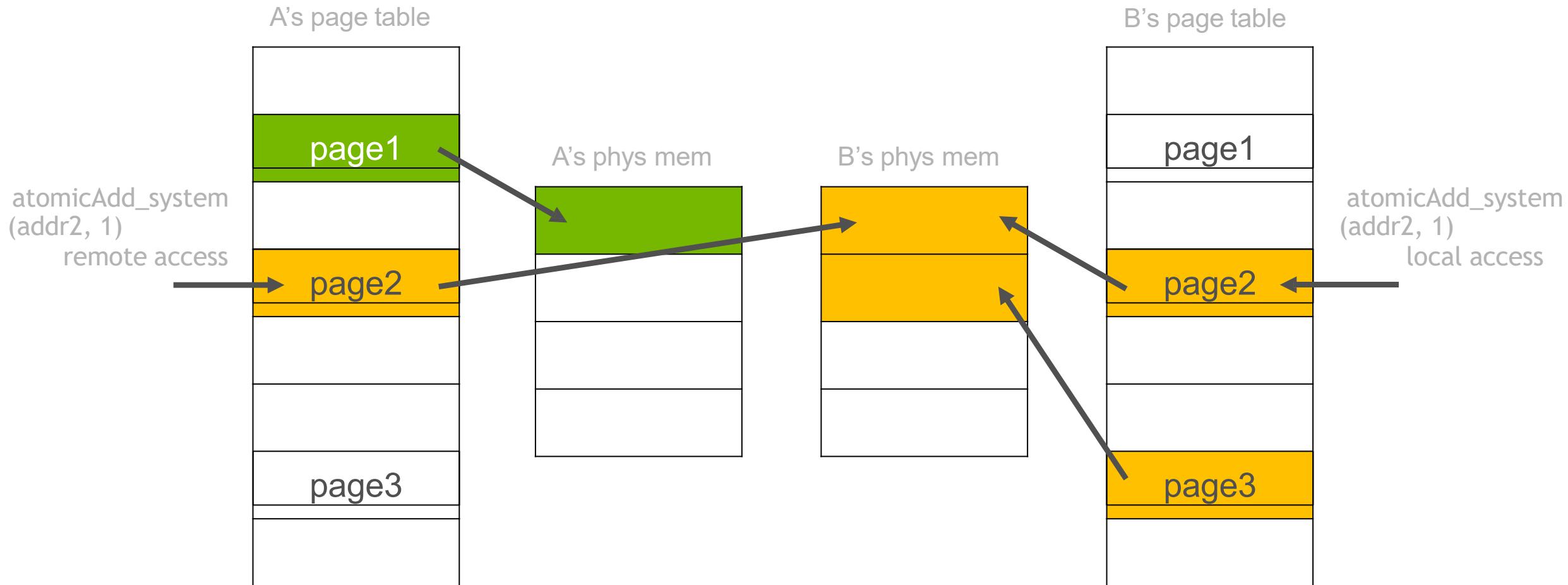
# CONCURRENT ACCESS

## Exclusive Access



# CONCURRENT ACCESS

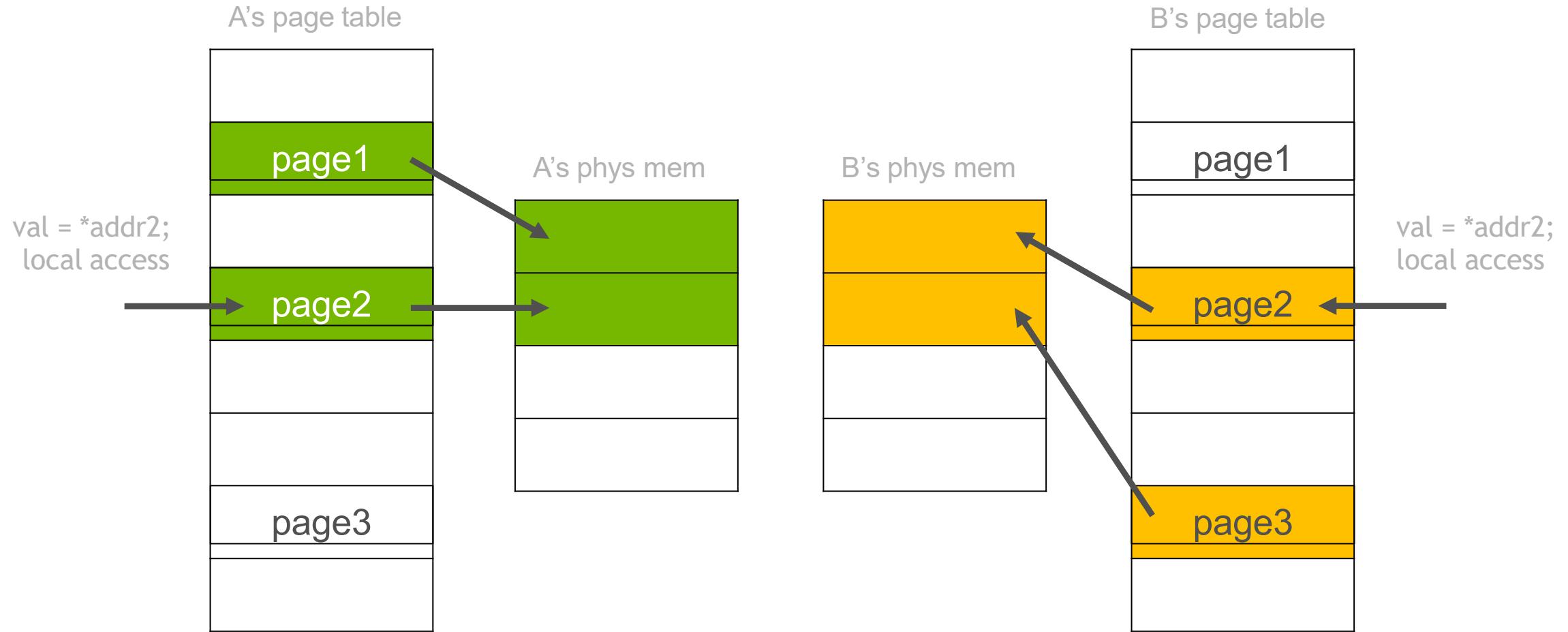
## Atomsics over NVLINK\*



\*both processors need to support atomic operations

# CONCURRENT ACCESS

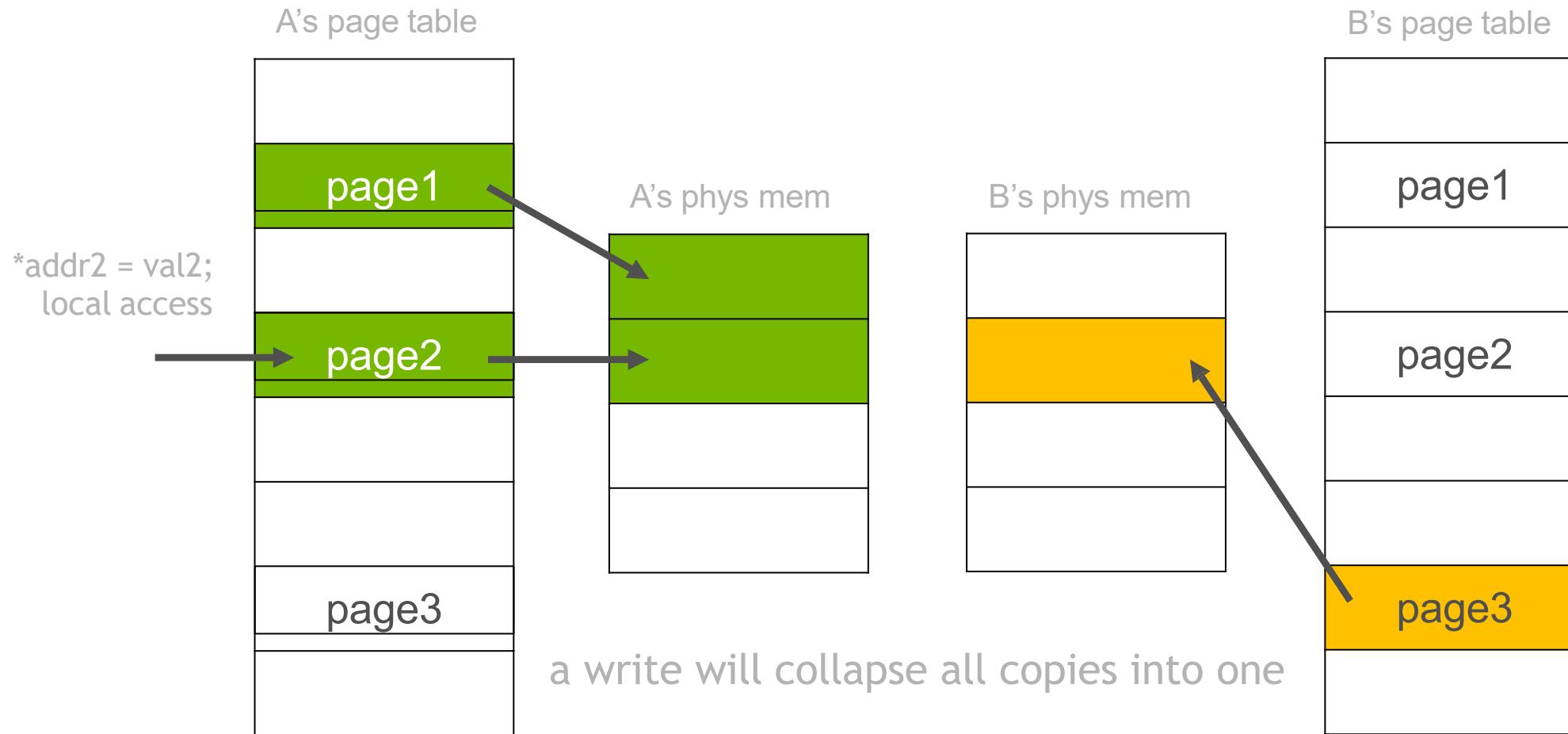
## Read duplication\*



\*each processor must maintain its own page table

# CONCURRENT ACCESS

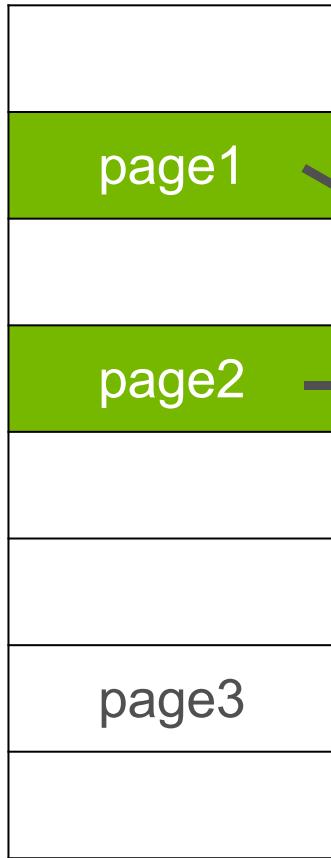
## Read duplication: write



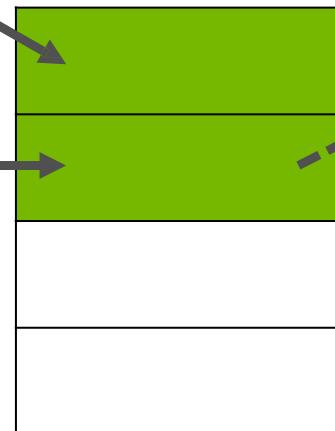
# CONCURRENT ACCESS

Read duplication: read after write

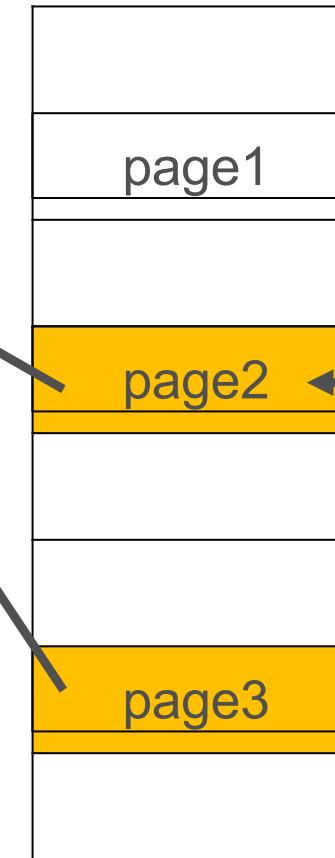
A's page table



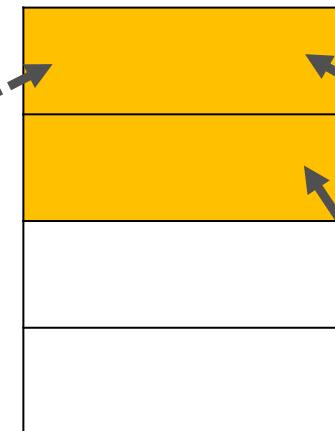
A's phys mem



B's page table



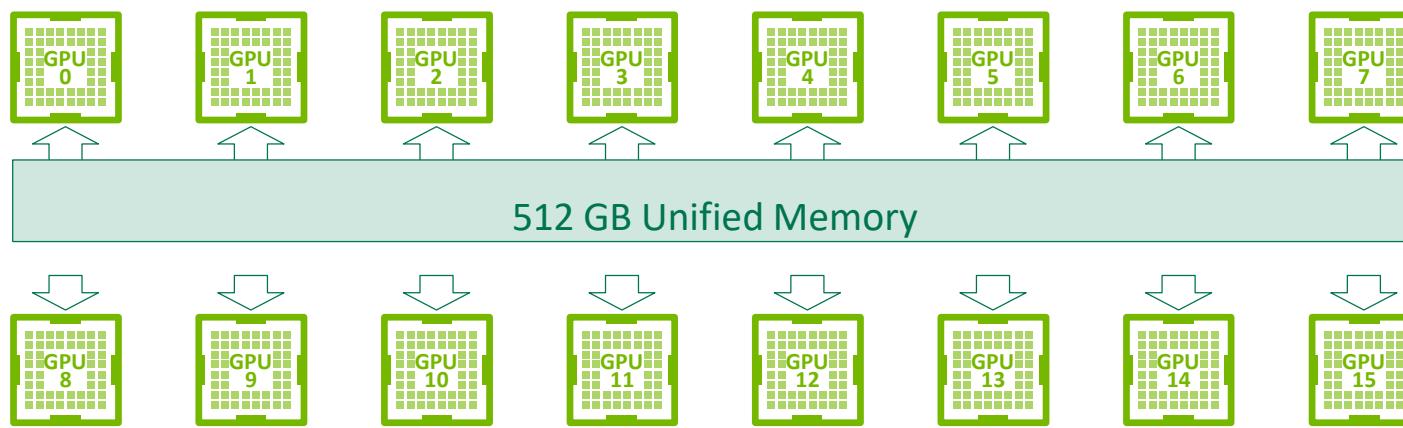
B's phys mem



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);
```

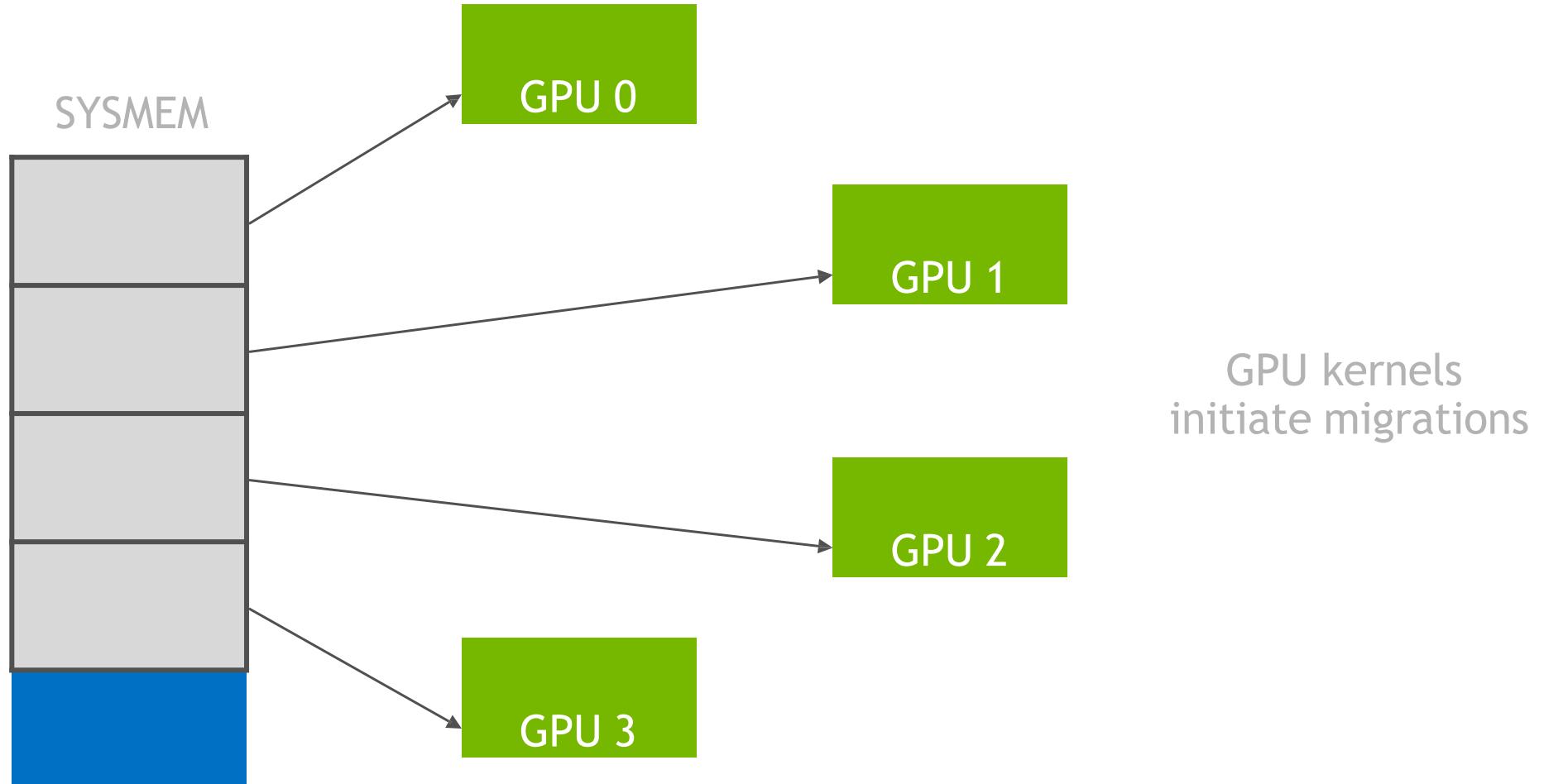
## Multi-GPU

```
__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);  
}
```

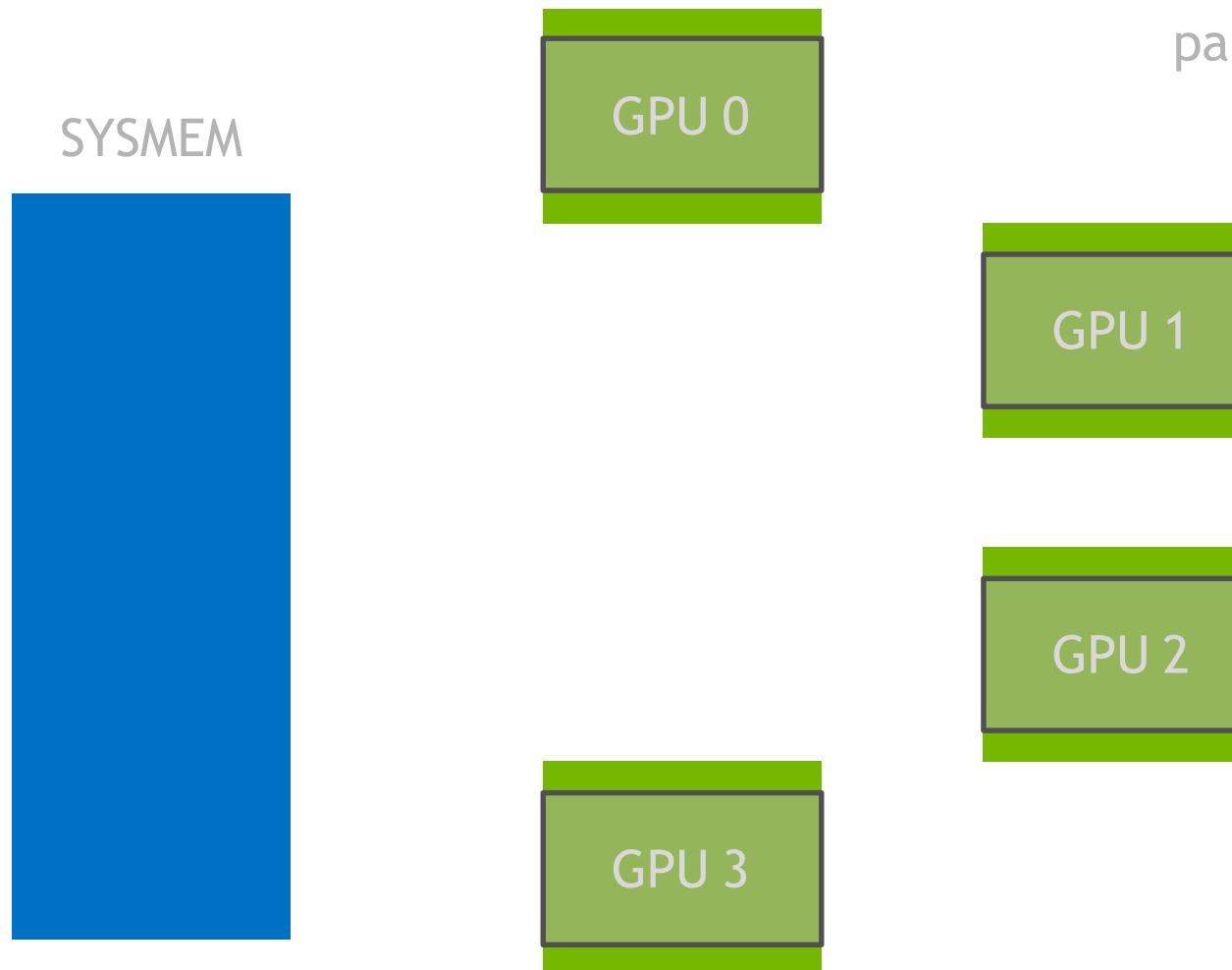
update blockIdx.x

update launch config

# MULTI-GPU WITH UNIFIED MEMORY

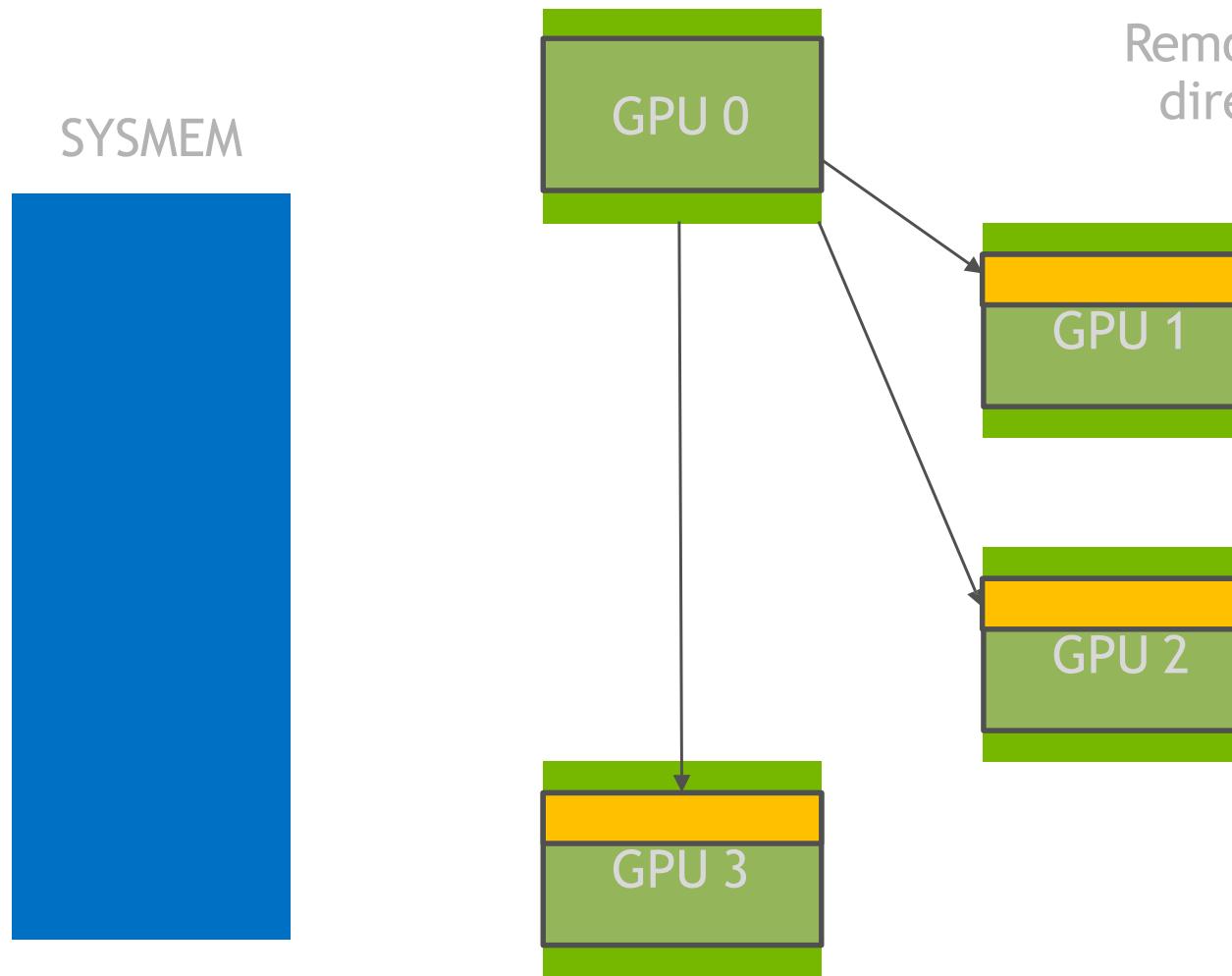


# MULTI-GPU WITH UNIFIED MEMORY



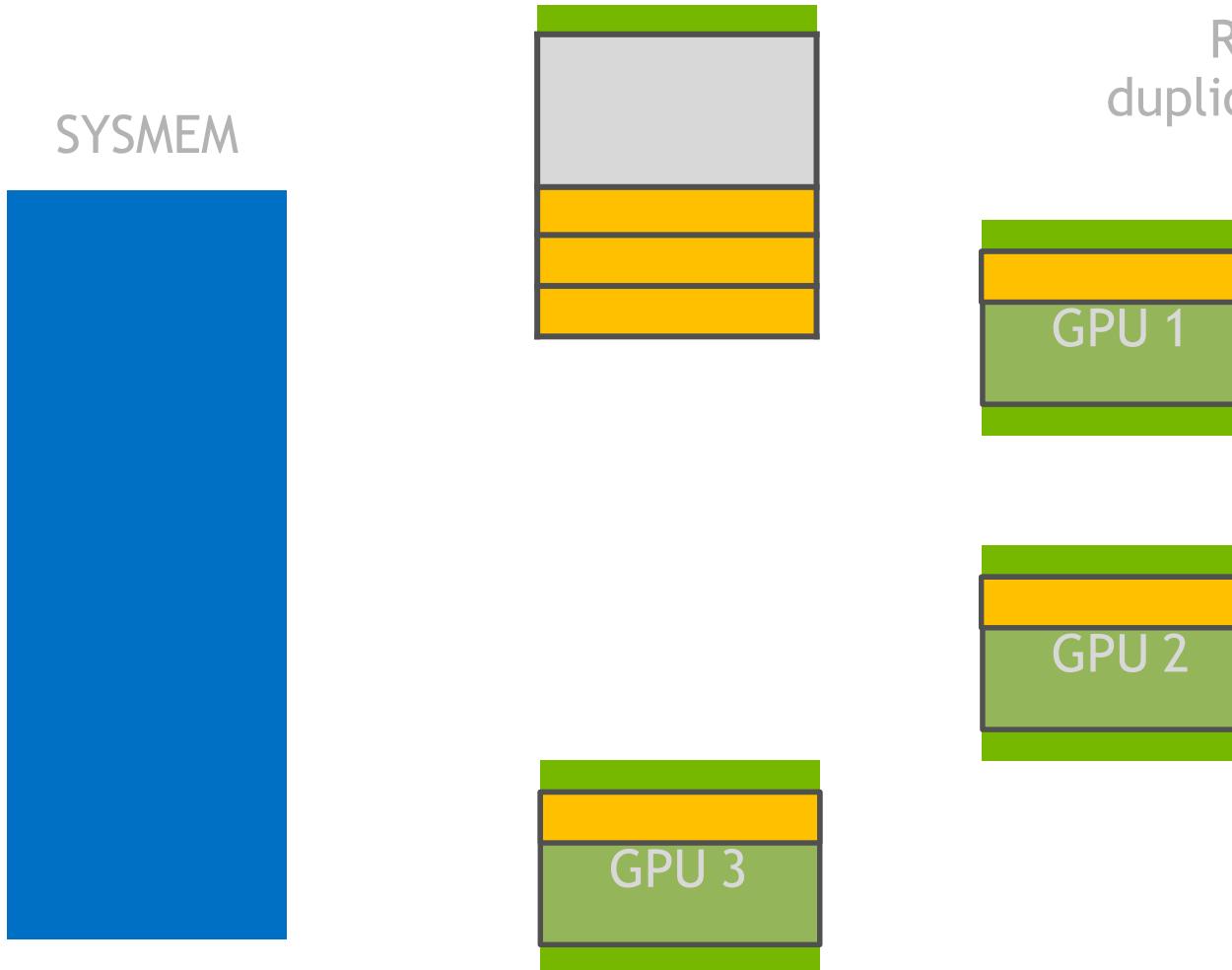
Data *automatically*  
partitioned between GPUs  
on first-touch

# MULTI-GPU WITH UNIFIED MEMORY



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

# MULTI-GPU WITH UNIFIED MEMORY

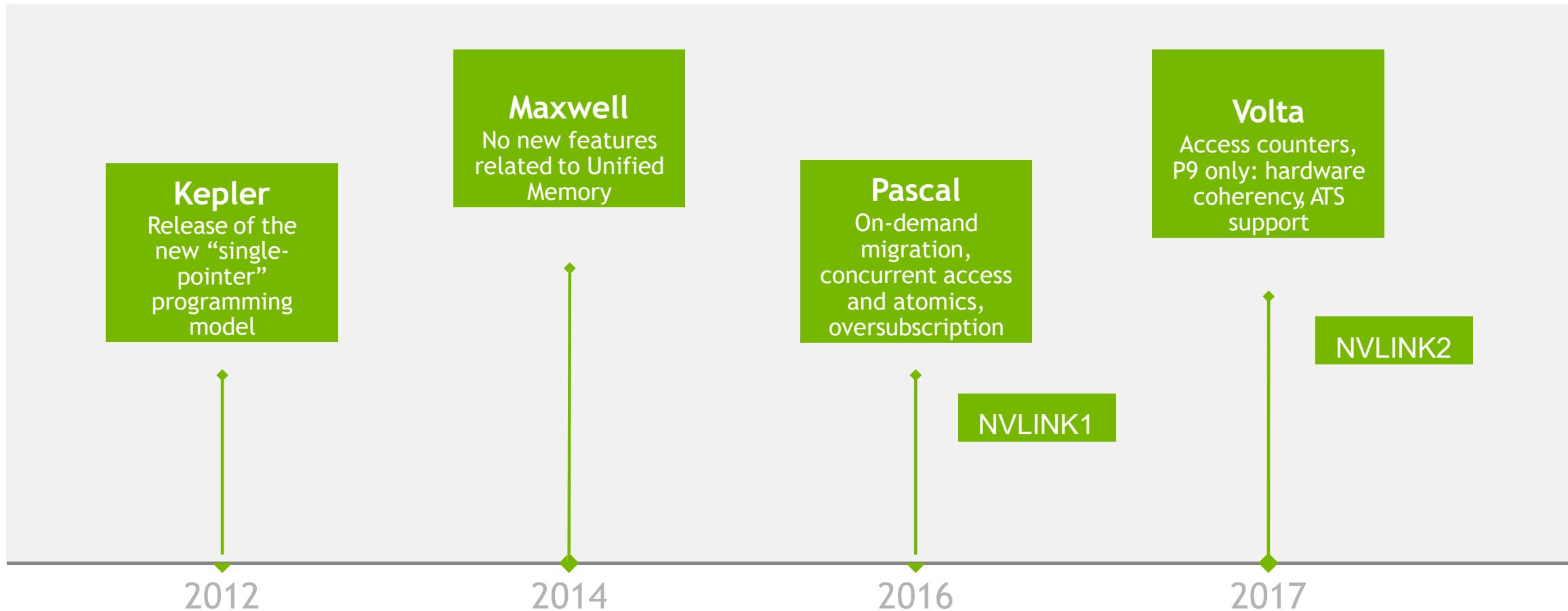


**With policies:**  
Read-only data can be  
duplicated and accessed locally

# GPU ARCHITECTURE AND SOFTWARE EVOLUTION

# UNIFIED MEMORY

## Evolution of GPU architectures



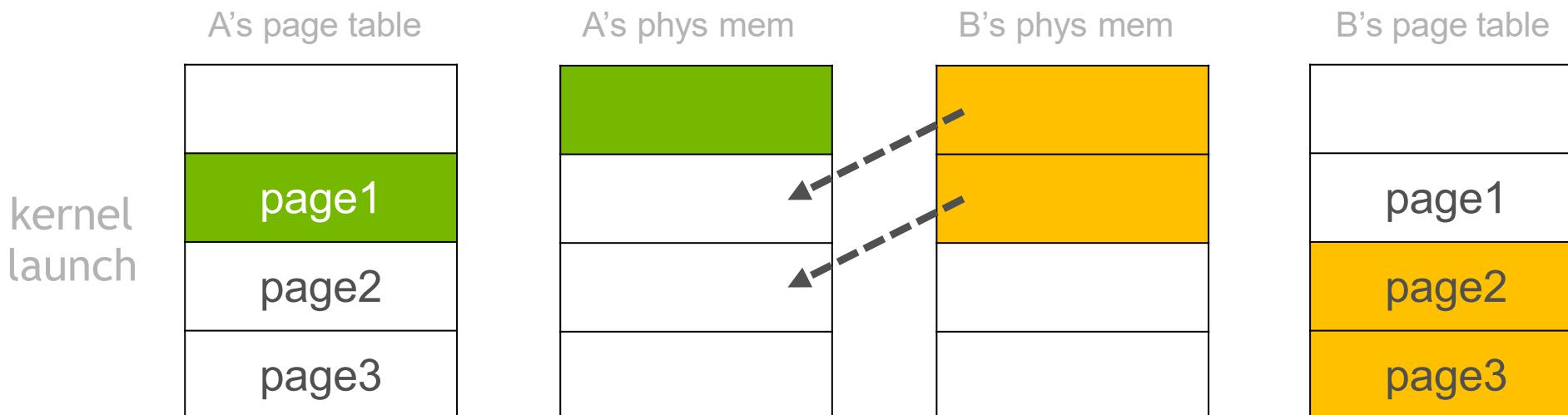
\*Not all features are available on all platforms

# 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**

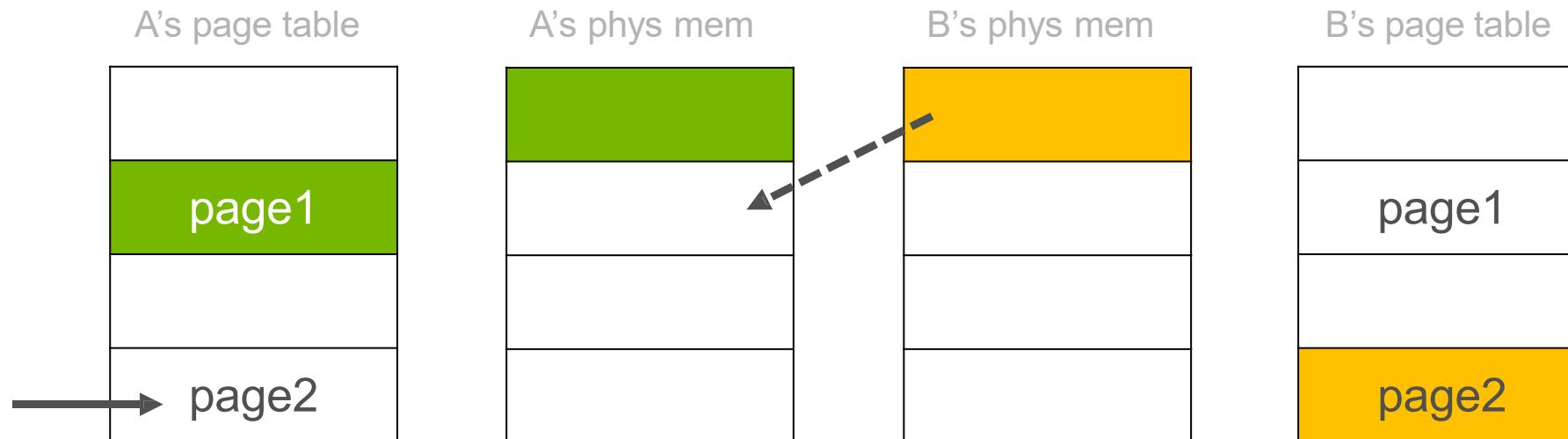


# UNIFIED MEMORY: PASCAL AND VOLTA

Available since CUDA 8

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

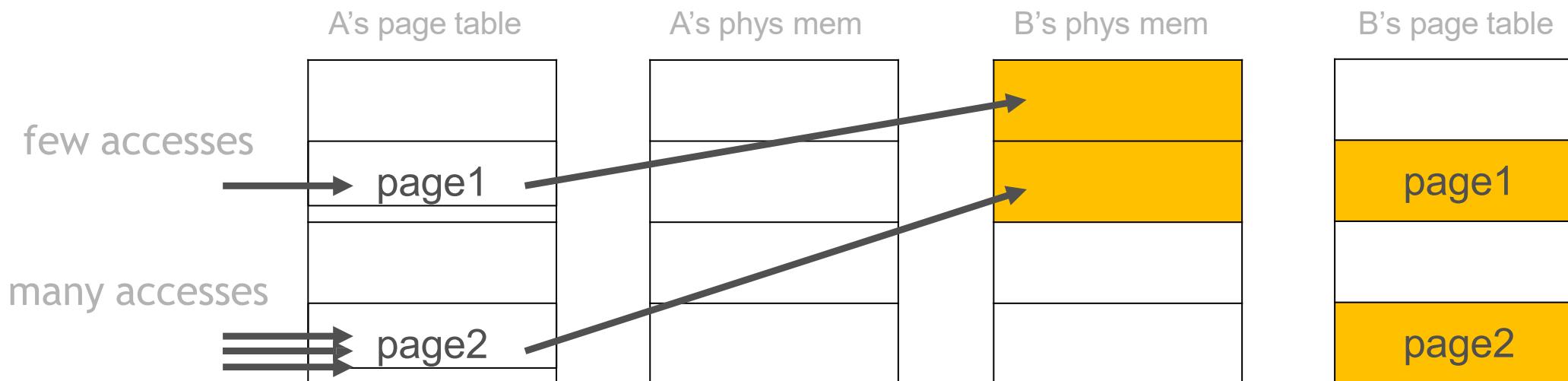
On-demand migration to accessing processor **on first touch**



# UNIFIED MEMORY ON VOLTA+P9

## New Feature: Access Counters

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

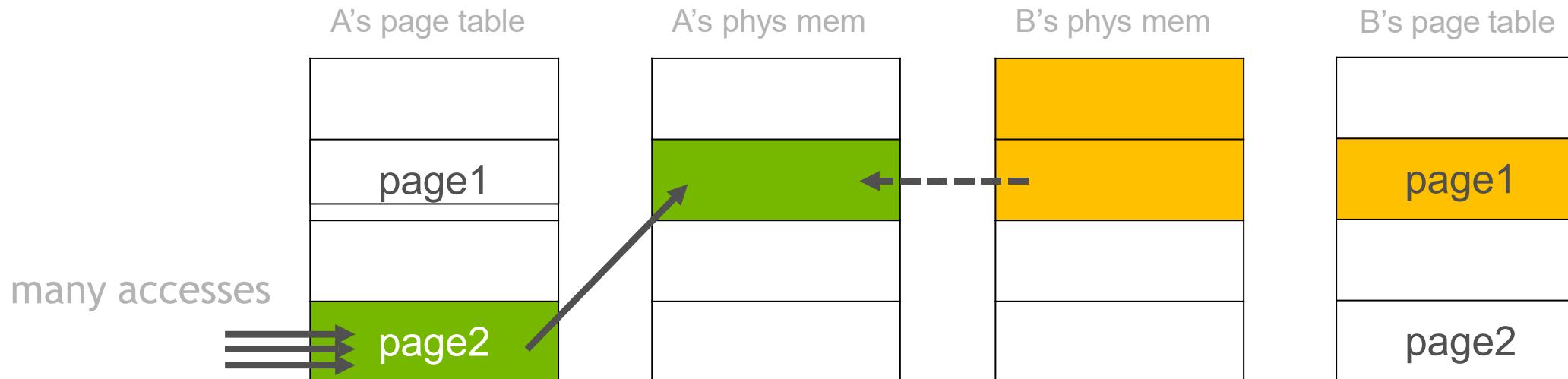


# UNIFIED MEMORY ON VOLTA+P9

## 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



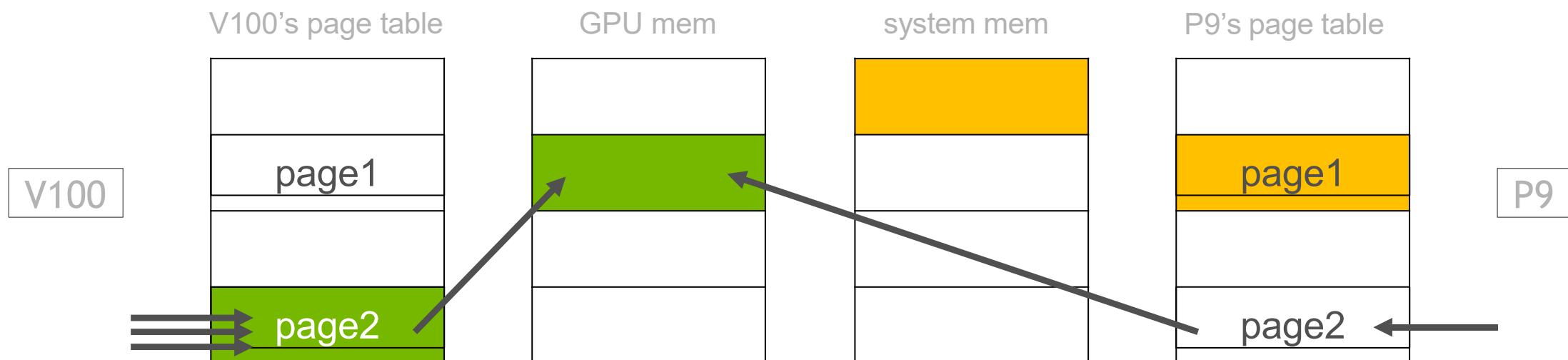
\*When implemented this feature can be enabled with `cudaMemAdvise` policies

# UNIFIED MEMORY ON VOLTA+P9

New Feature: Hardware Coherency with NVLINK2

CPU can directly access and *cache* GPU memory

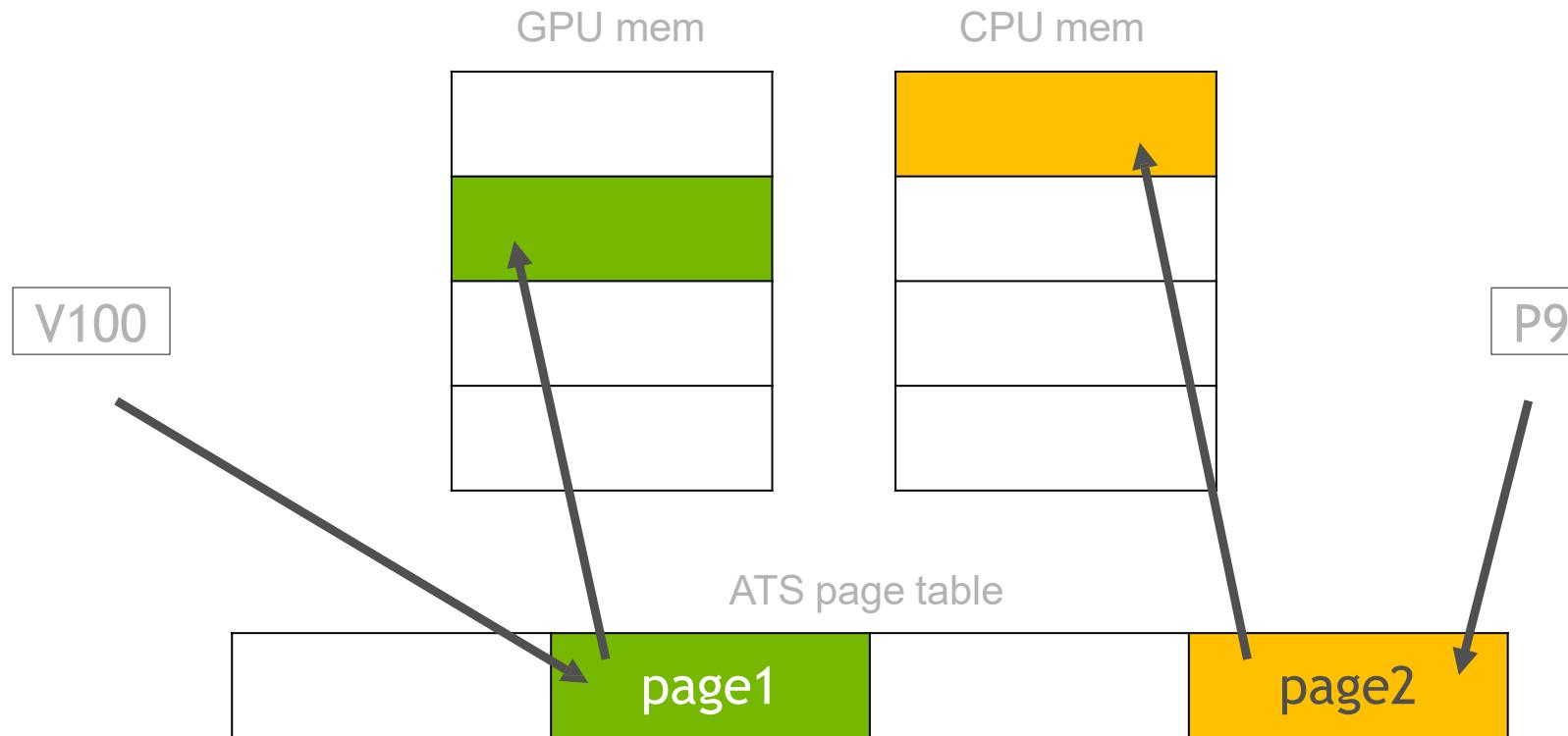
*Native* atomics support for all accessible memory



# 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

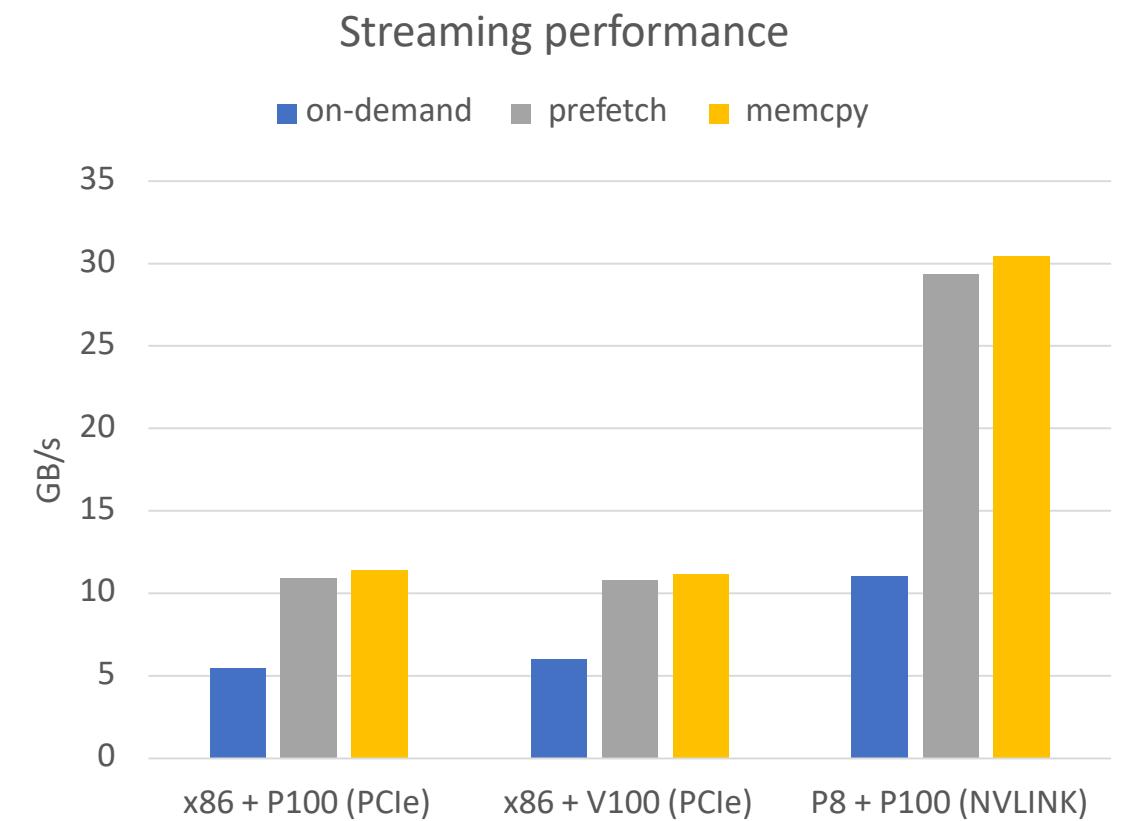
# STREAMING BENCHMARK

## 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

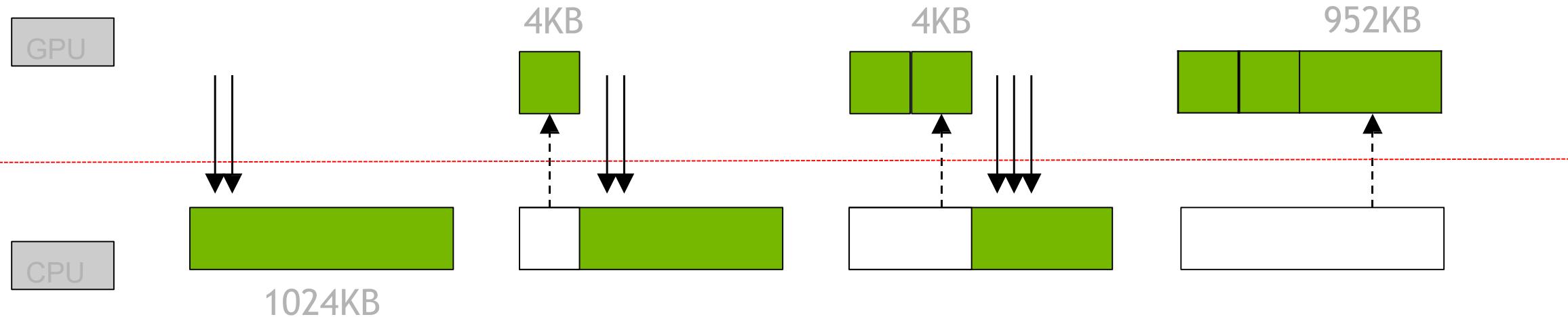
# HEURISTIC PREFETCHING

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*



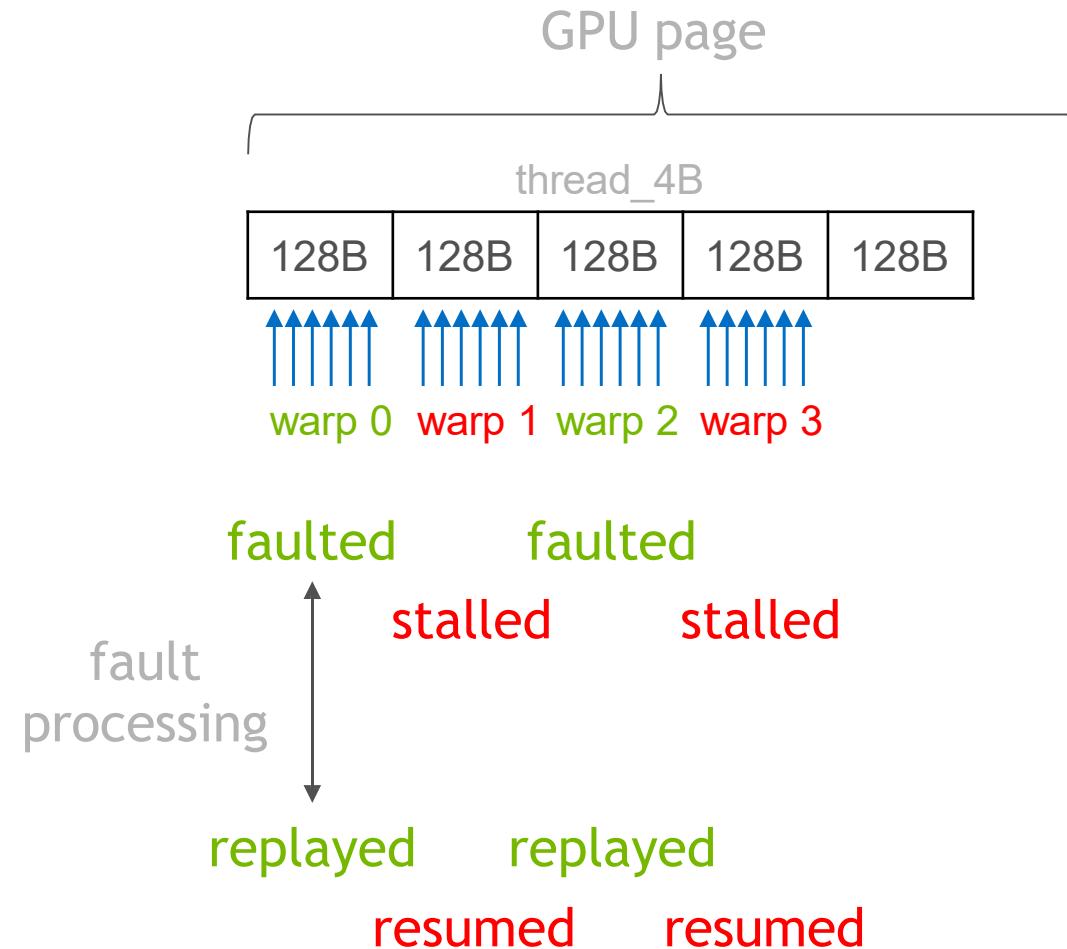
# WHAT IS PAGE FAULT GROUPS?

```
==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 ...

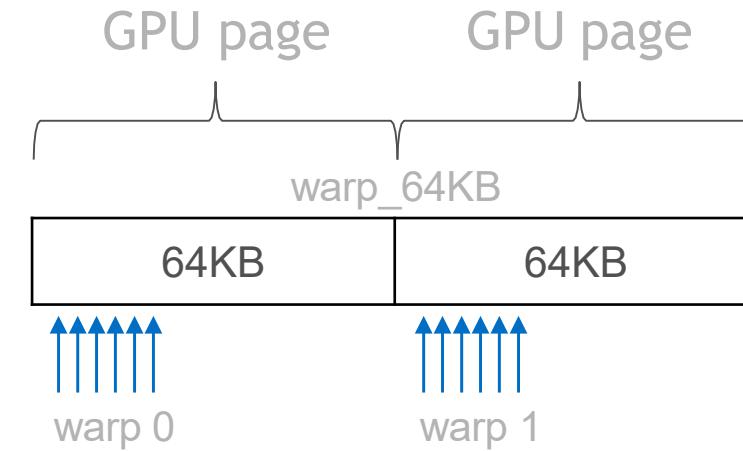
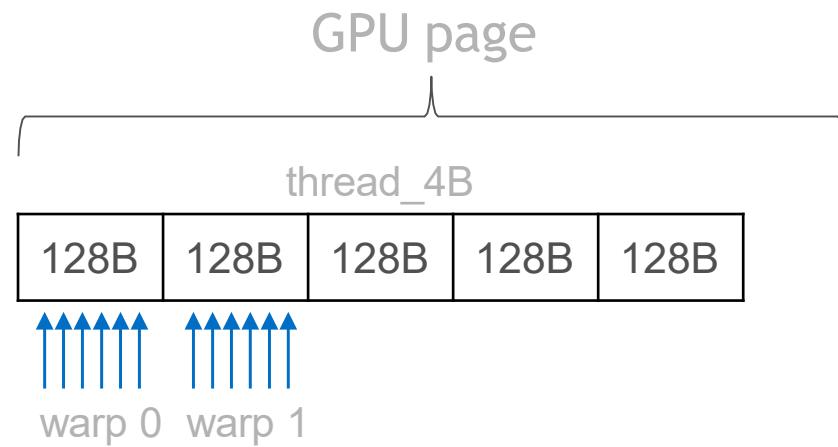
Unified Memory	Virtual Address	Name
8	0x3900010000	[Unified Memory GPU page faults]
9	0x3900040000	[Unified Memory GPU page faults]
5	0x3900108000	[Unified Memory GPU page faults]
5	0x3900200000	[Unified Memory GPU page faults]

# 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]

more efficient prefetching

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

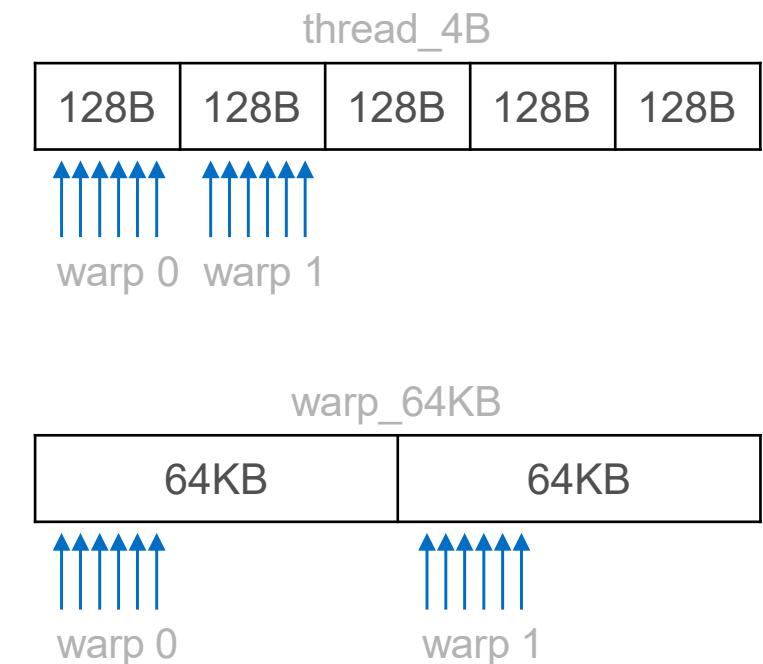
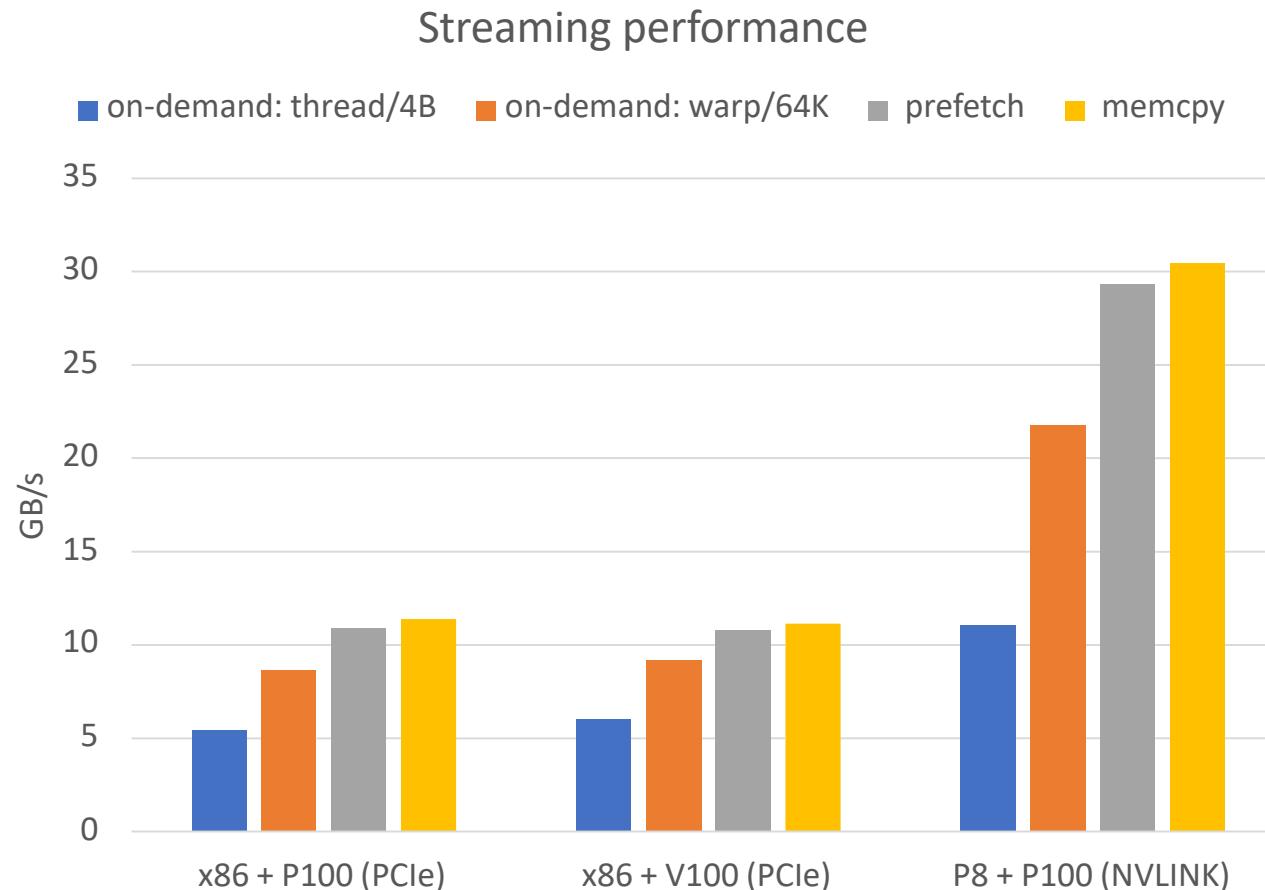
  

Unified Memory	Virtual Address	Name
1	0x39000d0000	[Unified Memory GPU page faults]
1	0x39000c0000	[Unified Memory GPU page faults]
1	0x3900080000	[Unified Memory GPU page faults]

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)

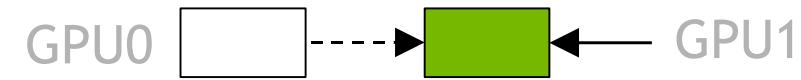
A chunk is considered “in use” when it is fully-populated or migrated

Prefetching and policies may impact eviction heuristic in the future

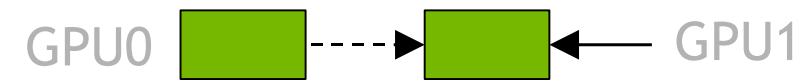
# LOCALITY AND ACCESS CONTROL

## cudaMemAdvise

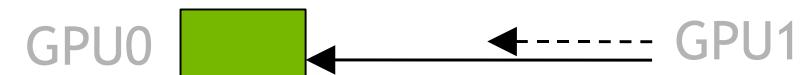
**Default:** data *migrates* on first touch



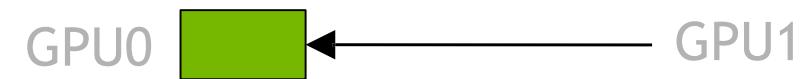
**ReadMostly:** data *duplicated* on first touch



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



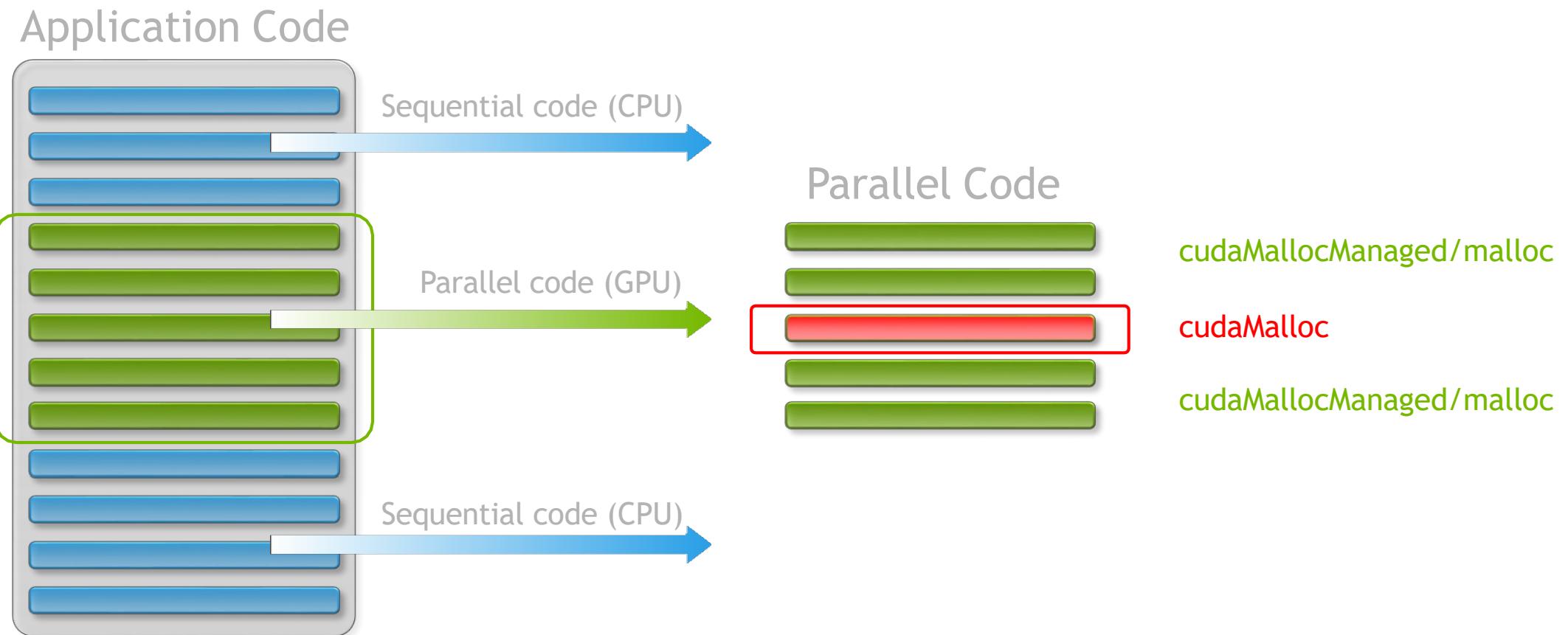
**AccessedBy:** establish *direct mapping* and avoid faults



# 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

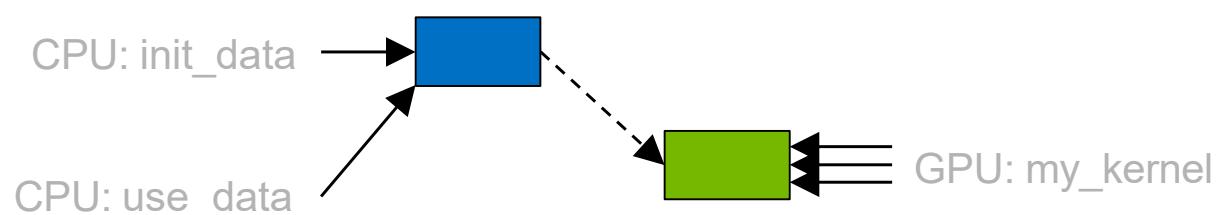
## Usage example

```
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



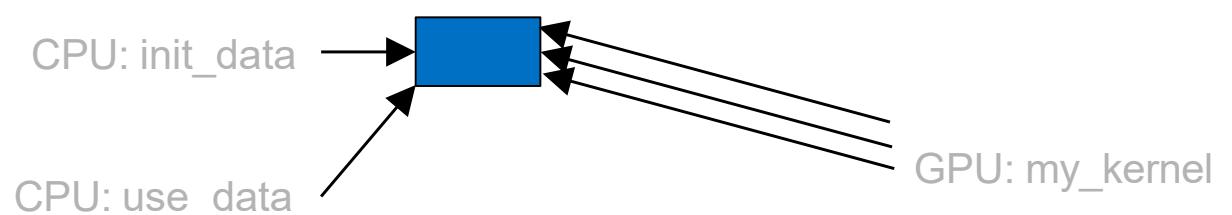
# PREFERRED LOCATION

## 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



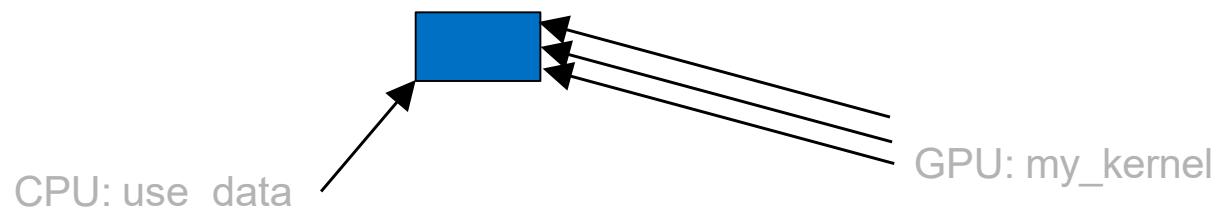
# PREFERRED LOCATION

## Page population on first-touch

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

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

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



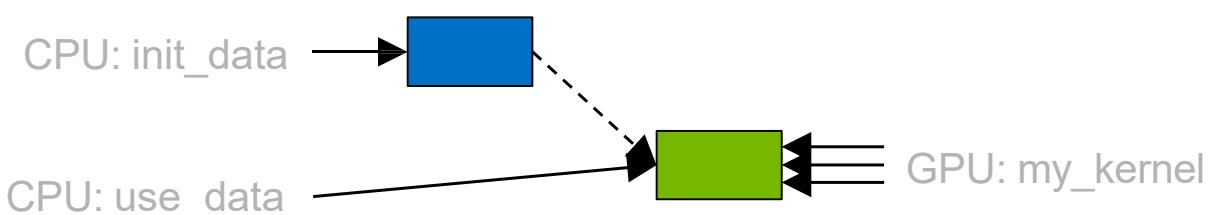
# PREFERRED LOCATION ON P9+V100

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

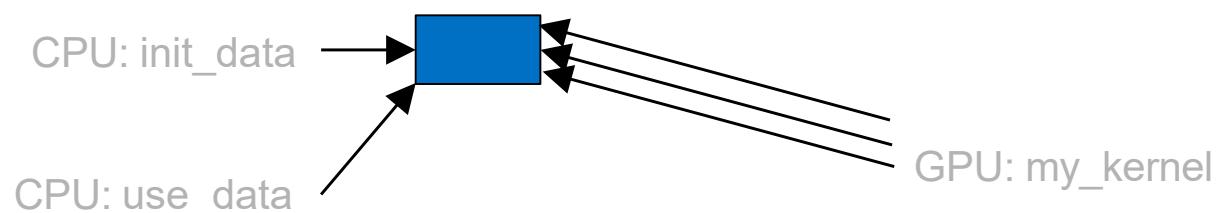
# ACCESSED BY

## 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



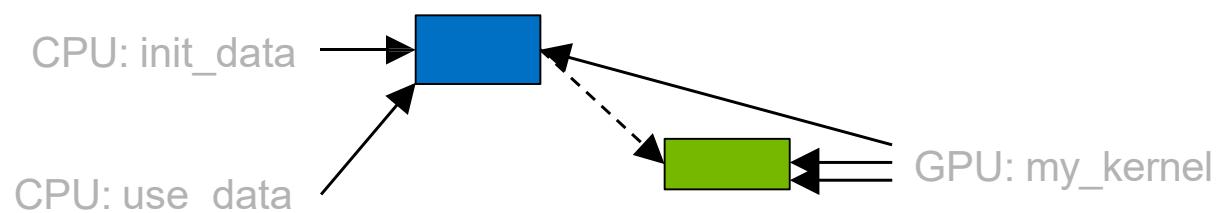
# ACCESSED BY

## 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

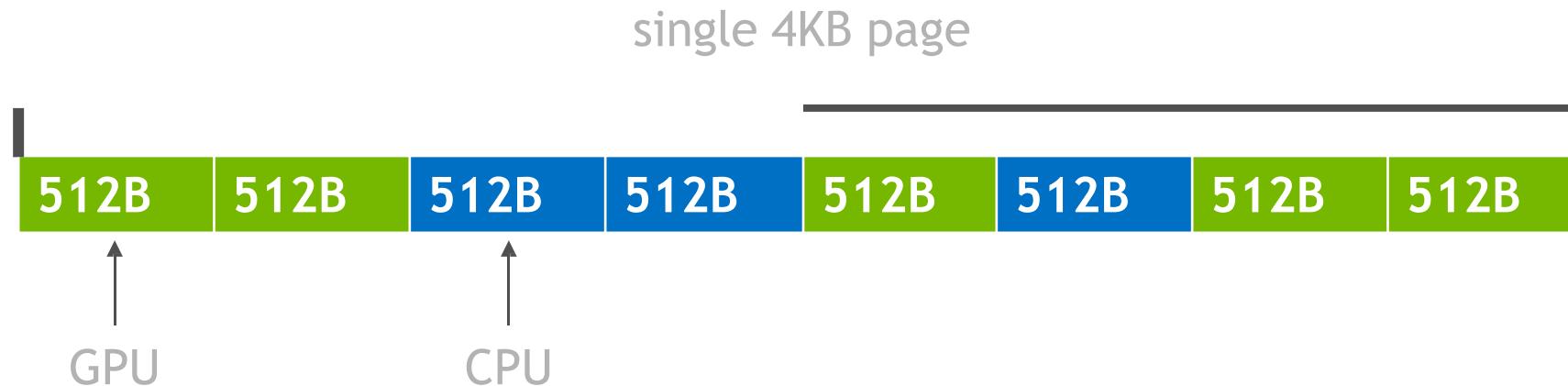
\*In the future Volta access counters will be used to migrate malloc memory

# 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**



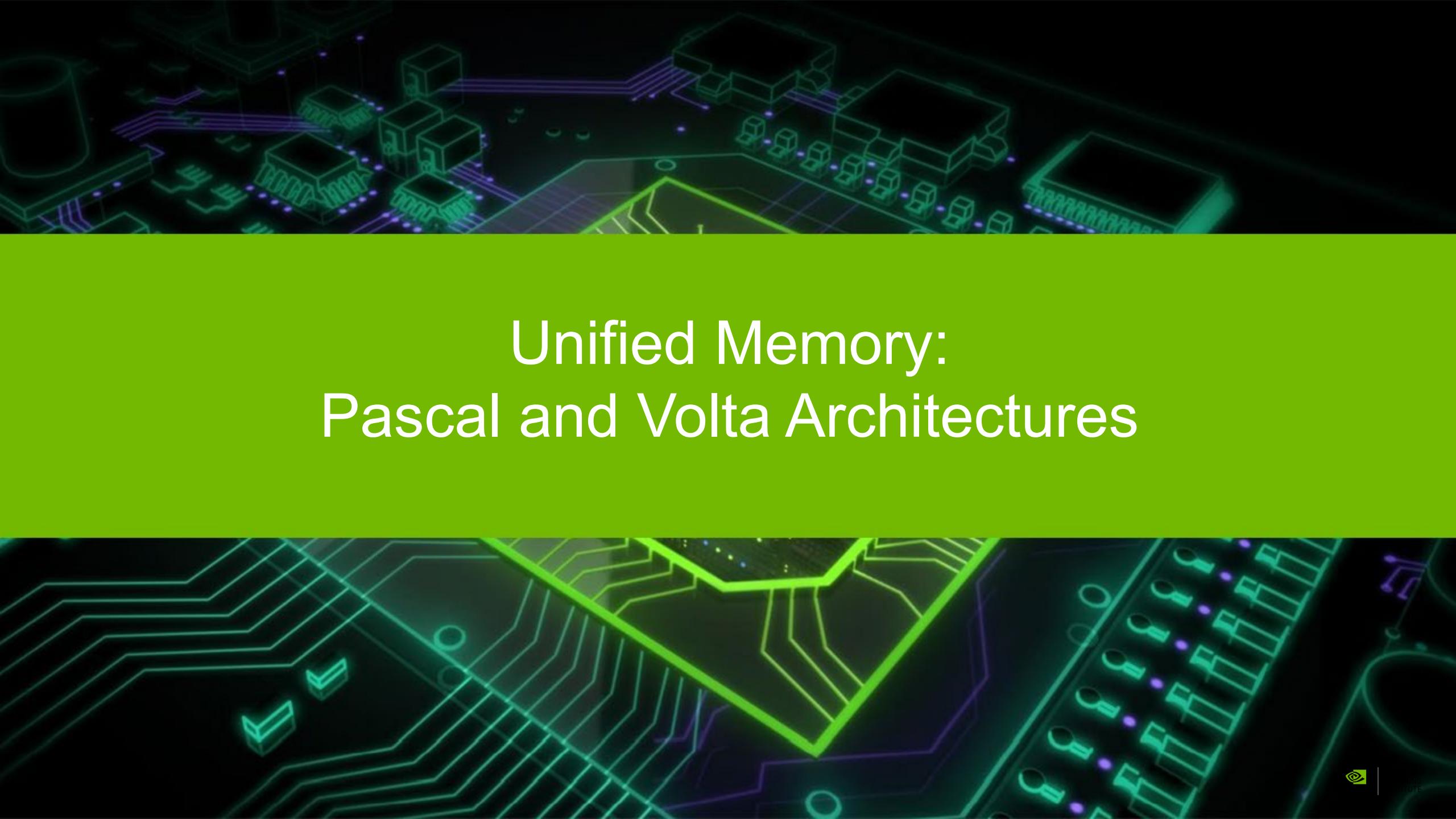
# FALSE-SHARING

## 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

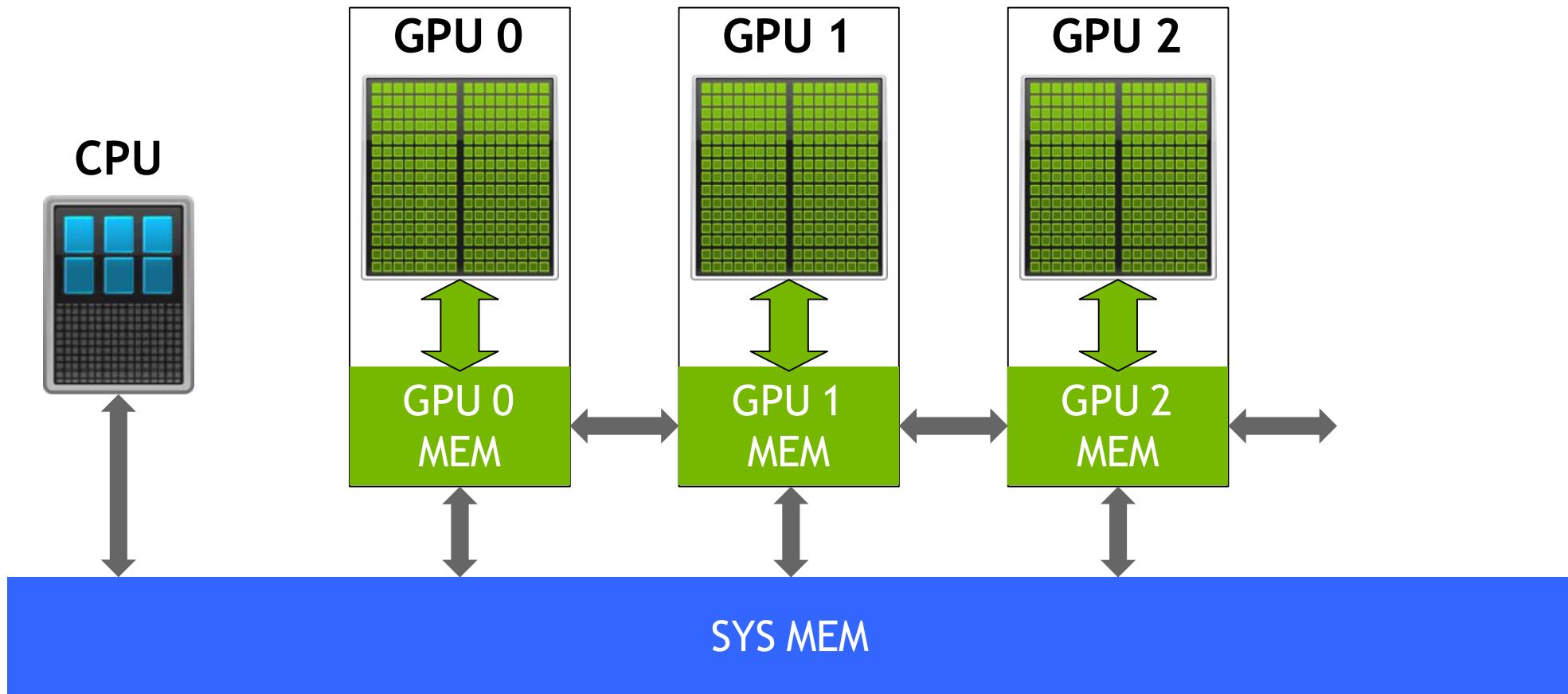
## How to mitigate this:

- 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);
```

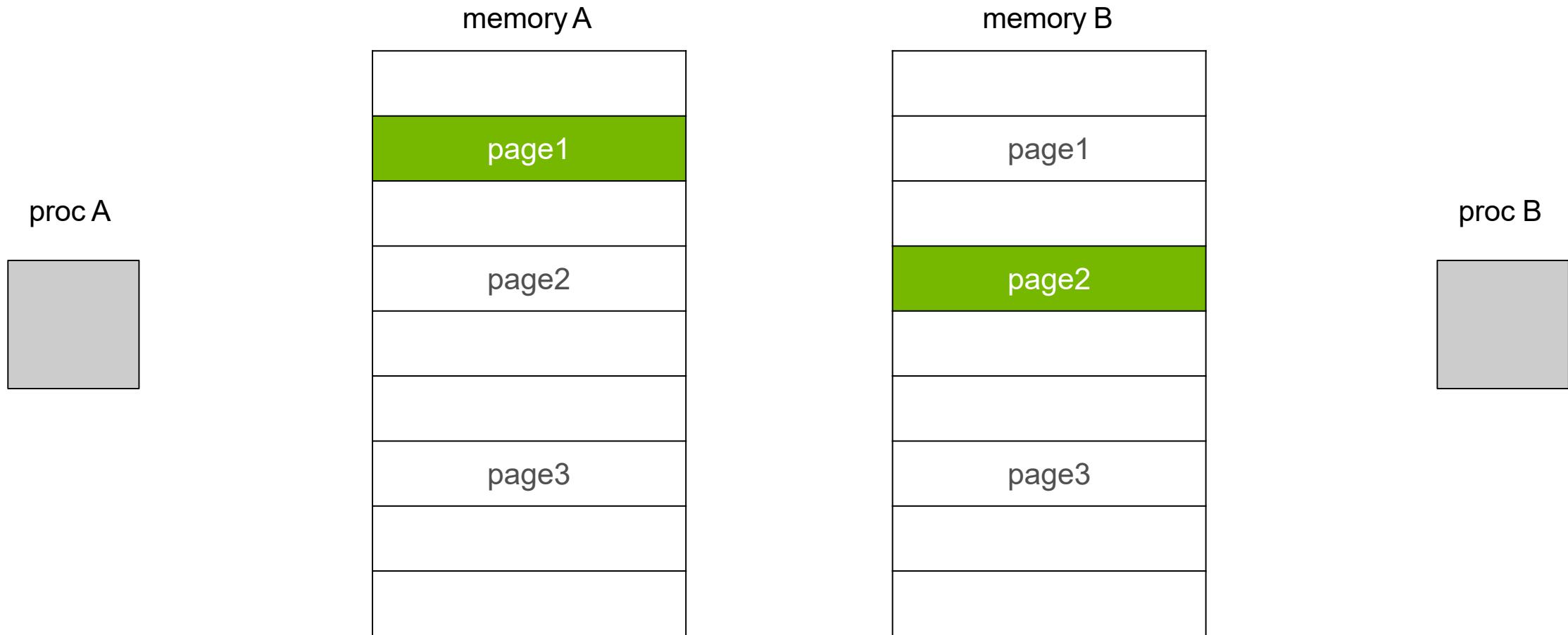
### Unified Memory

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

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

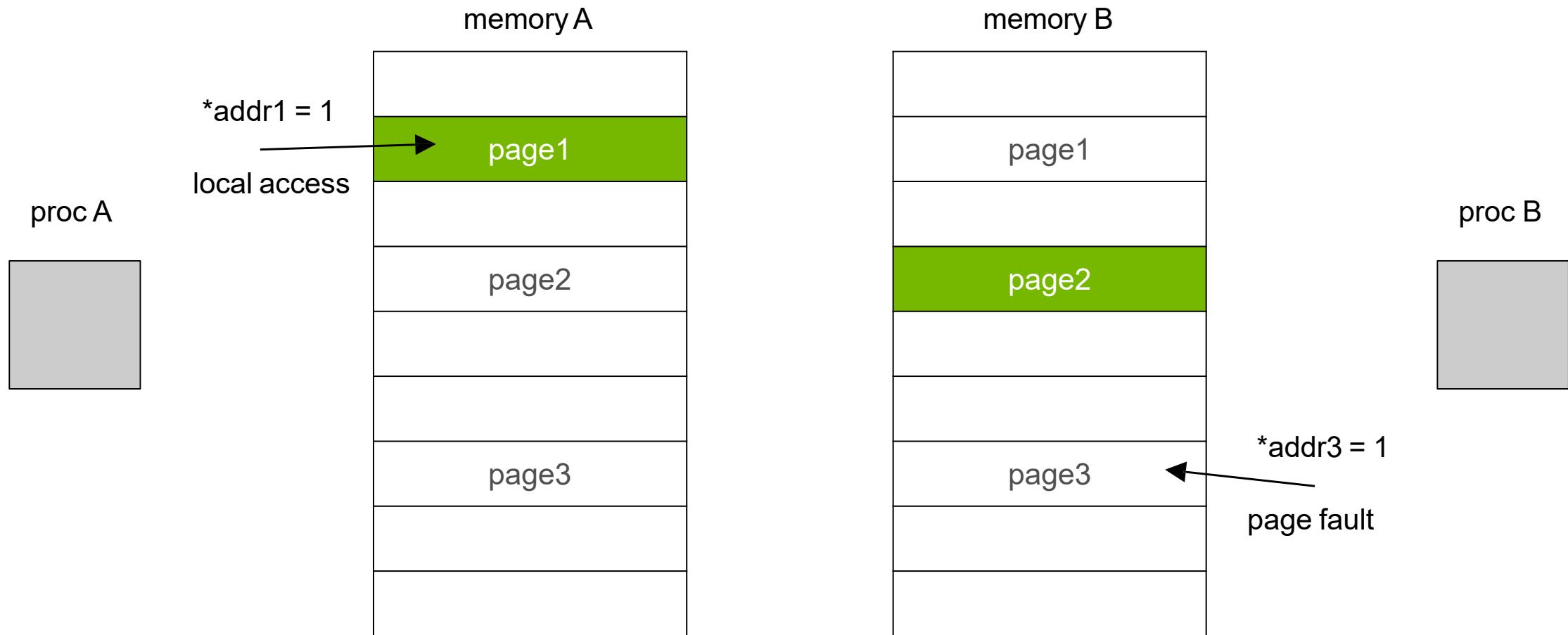
# UNIFIED MEMORY FUNDAMENTALS

## On-Demand Migration



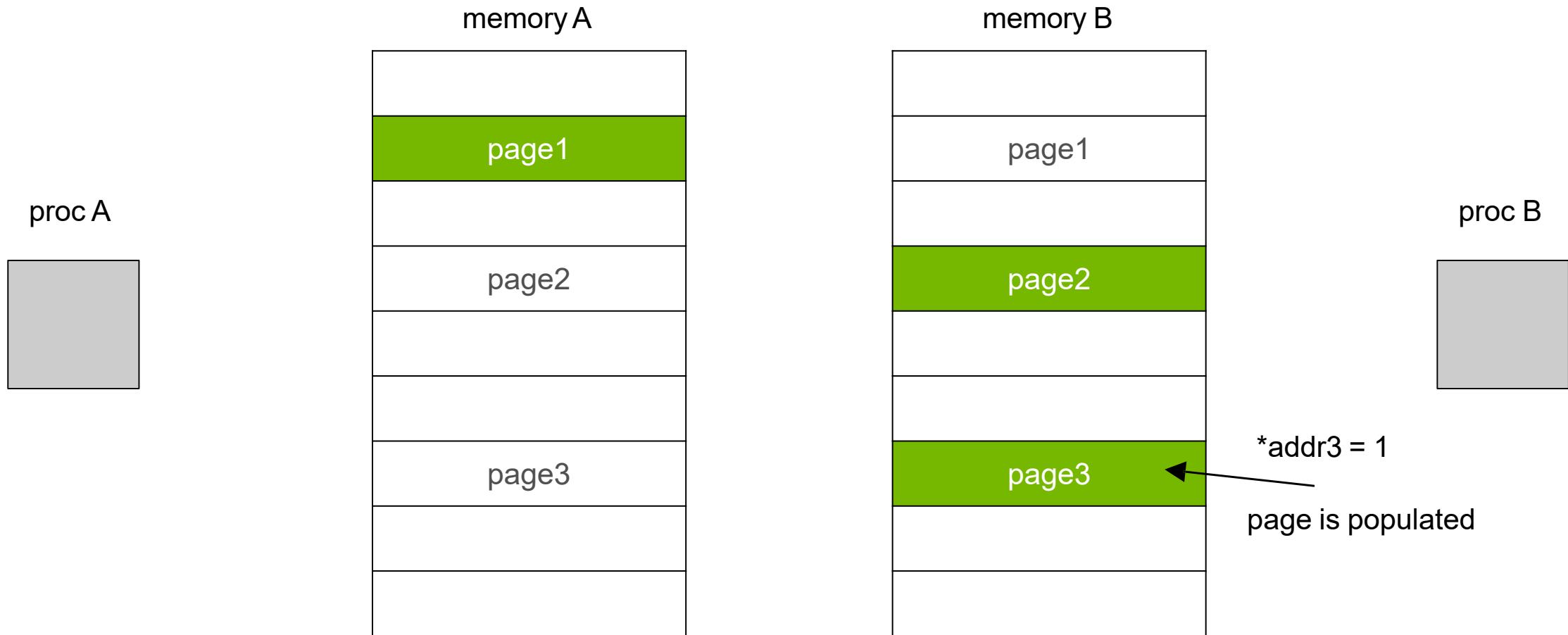
# UNIFIED MEMORY FUNDAMENTALS

## On-Demand Migration



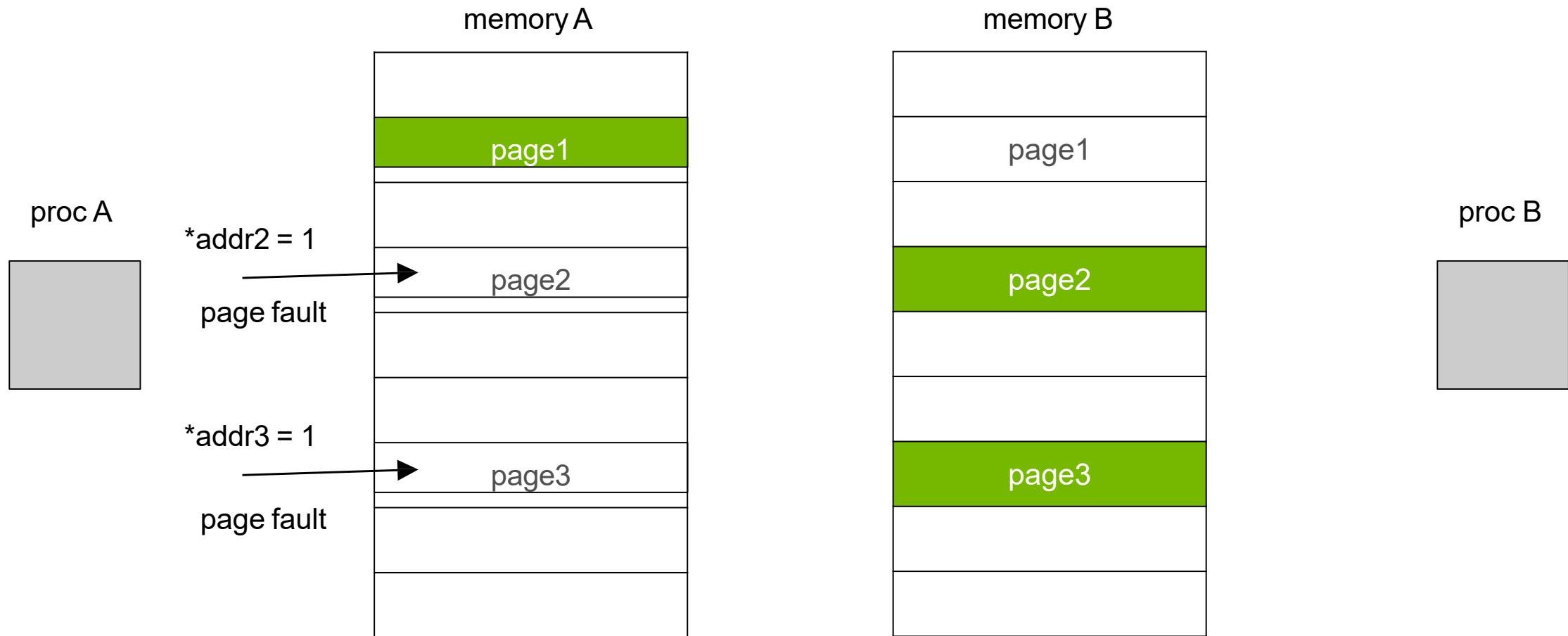
# UNIFIED MEMORY FUNDAMENTALS

## On-Demand Migration



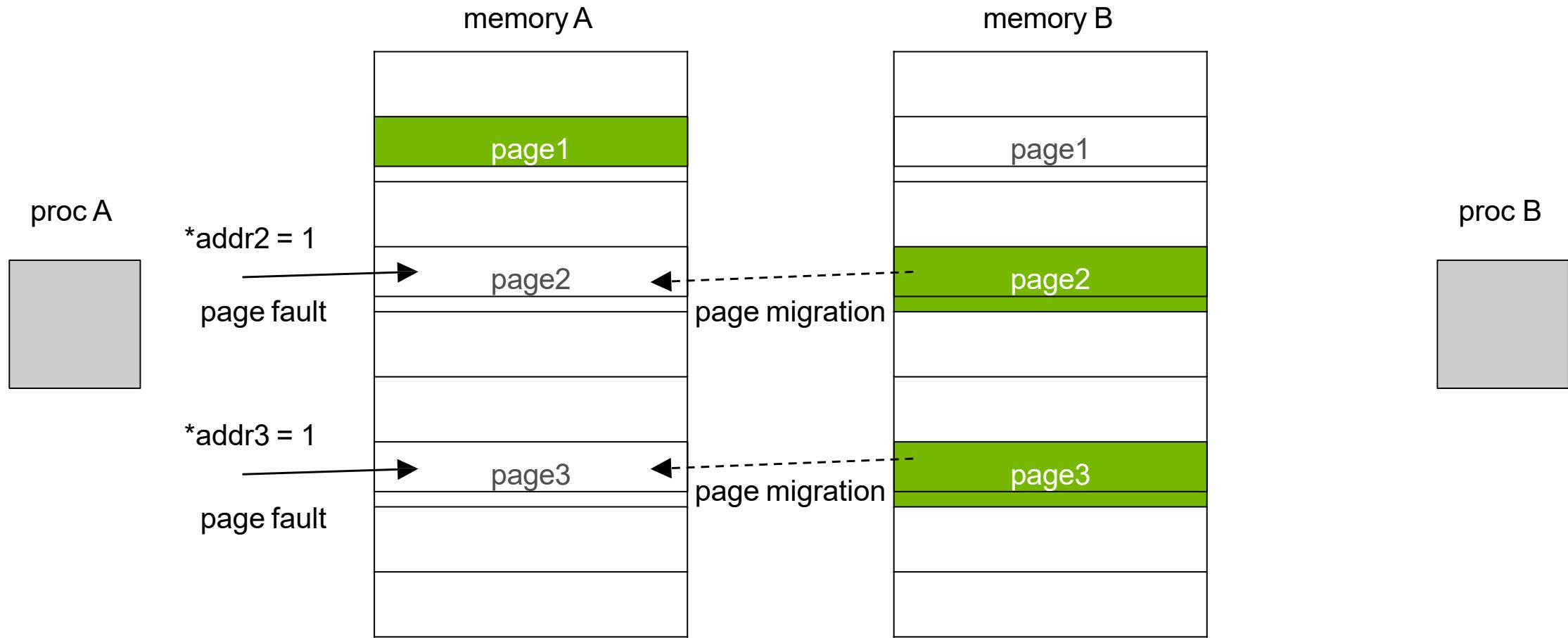
# UNIFIED MEMORY FUNDAMENTALS

## On-Demand Migration



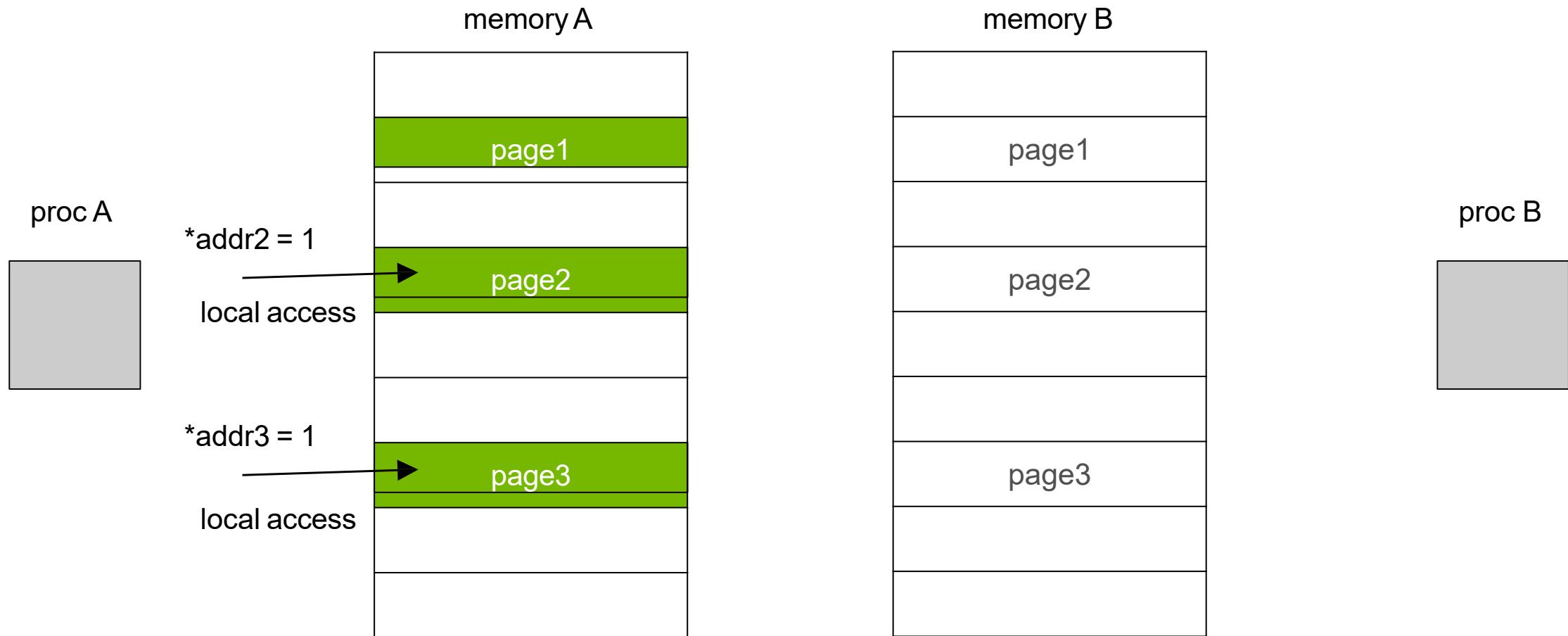
# UNIFIED MEMORY FUNDAMENTALS

## 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

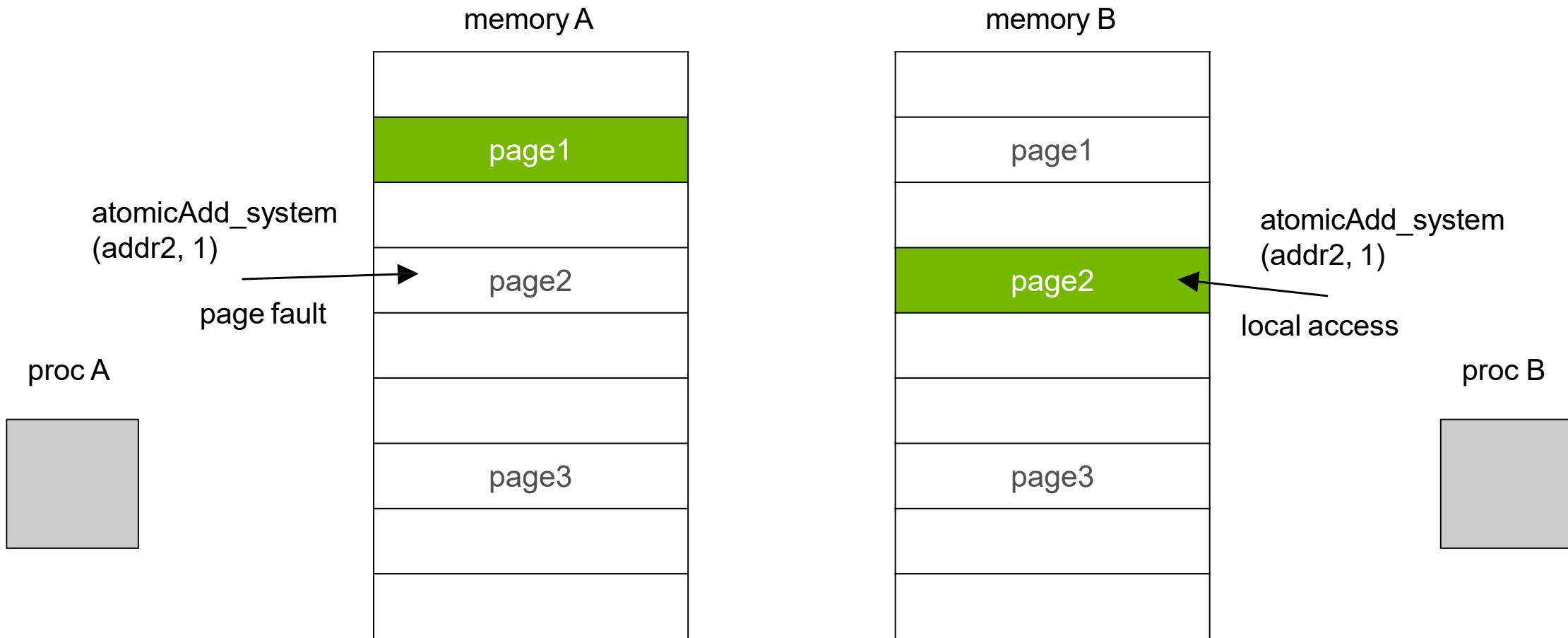
Moving pieces by hand is error-prone and requires tuning for memory size

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

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

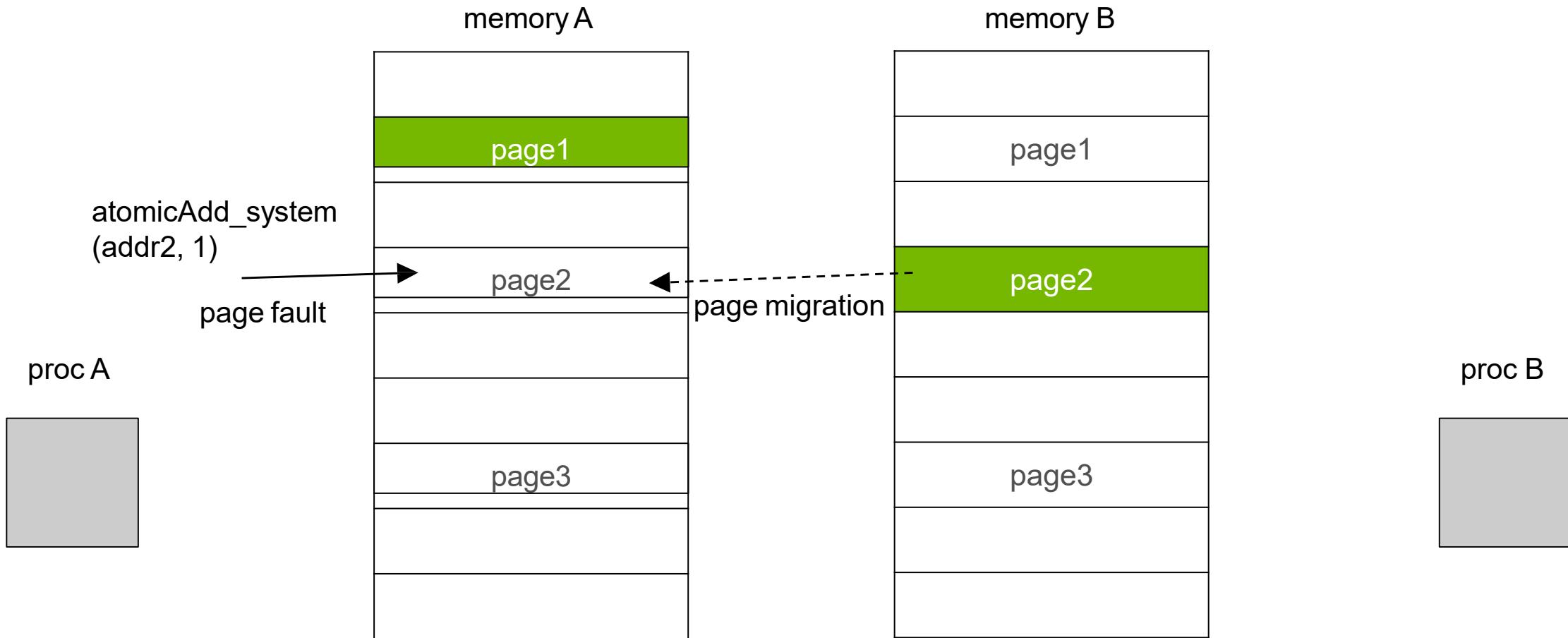
# UNIFIED MEMORY FUNDAMENTALS

## System-Wide Atomics with Exclusive Access



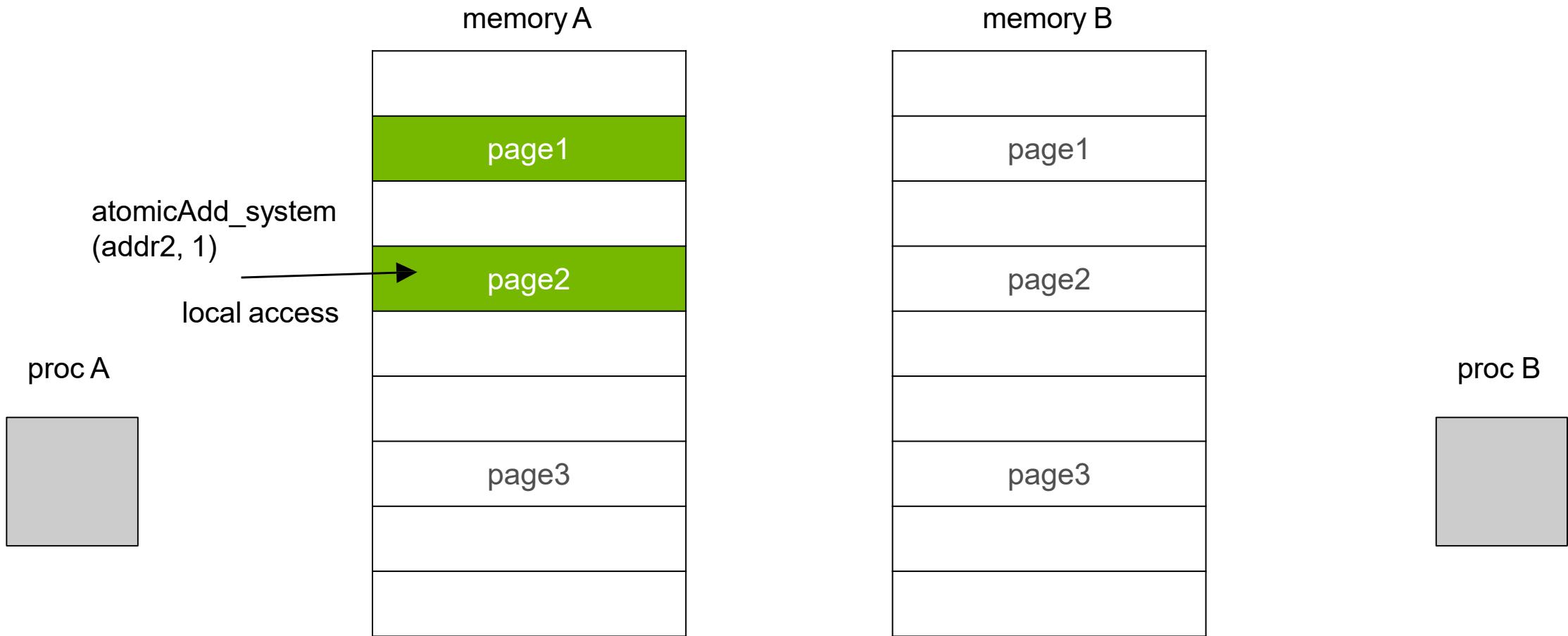
# UNIFIED MEMORY FUNDAMENTALS

## System-Wide Atomics with Exclusive Access



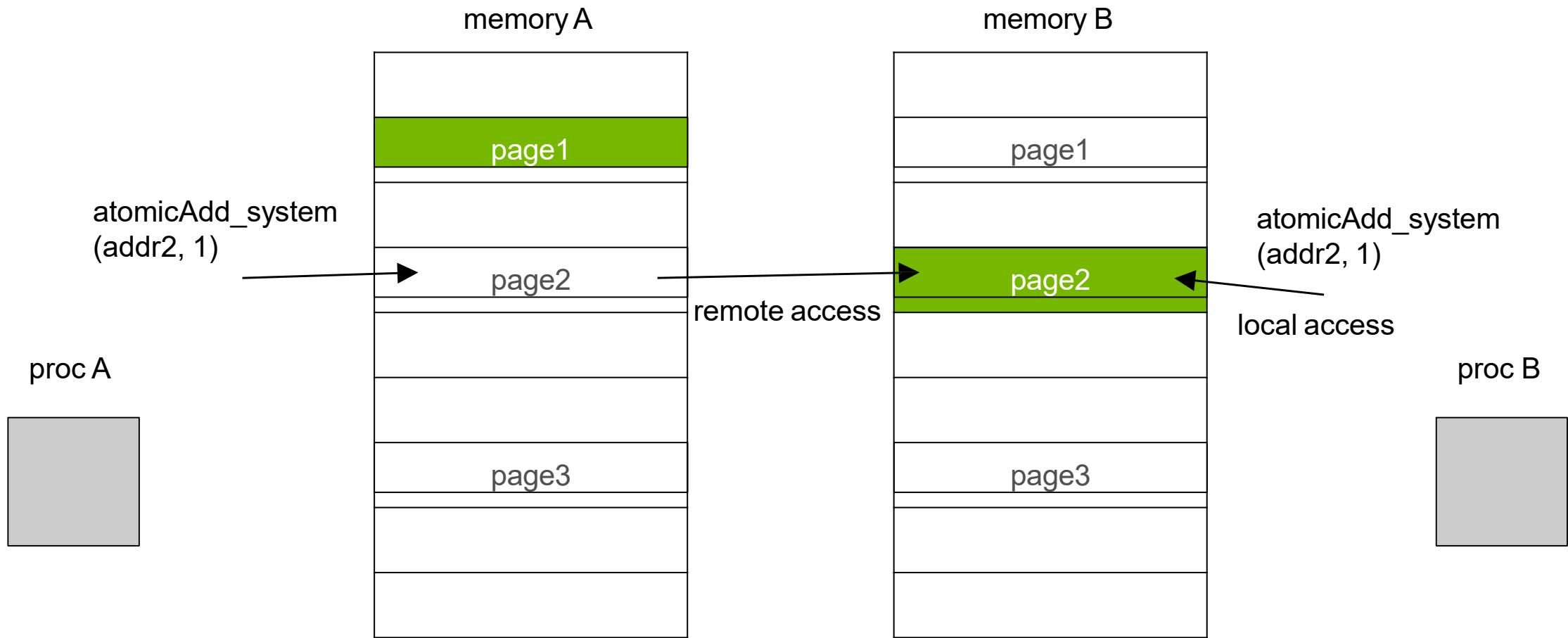
# UNIFIED MEMORY FUNDAMENTALS

## 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)

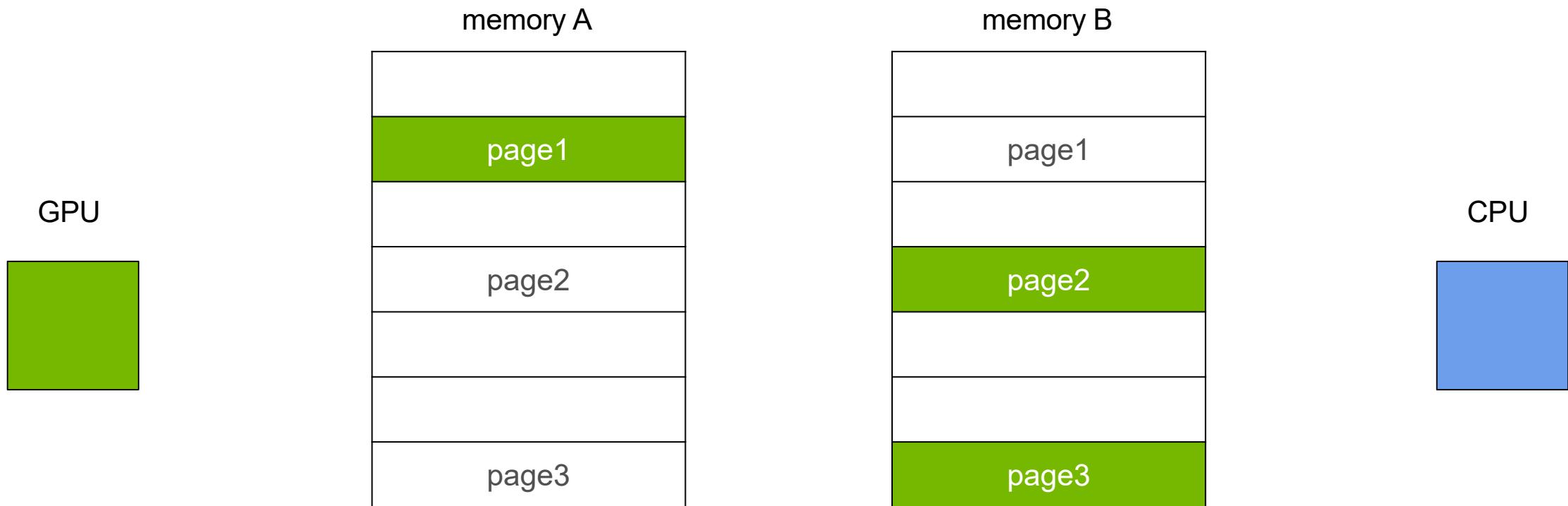
**malloc** support is coming on Pascal+ architectures (Linux only)

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

# UNIFIED MEMORY ON KEPLER

Available since CUDA 6

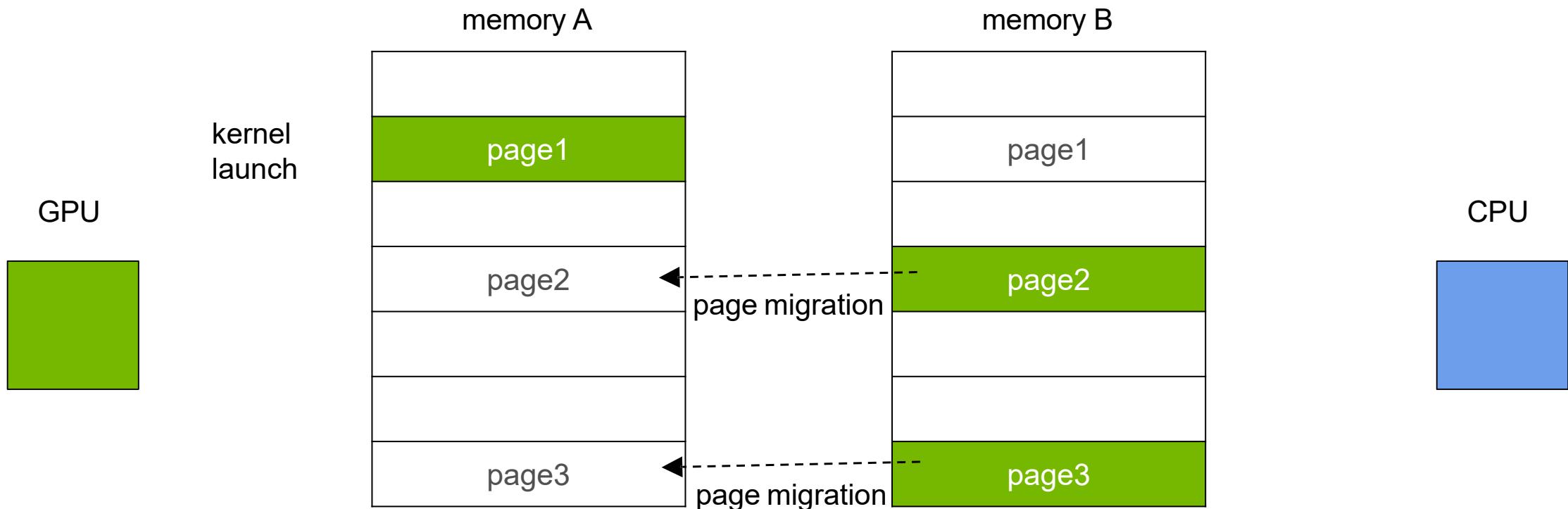
Kepler GPU: no page fault support, limited virtual space



# UNIFIED MEMORY ON KEPLER

Available since CUDA 6

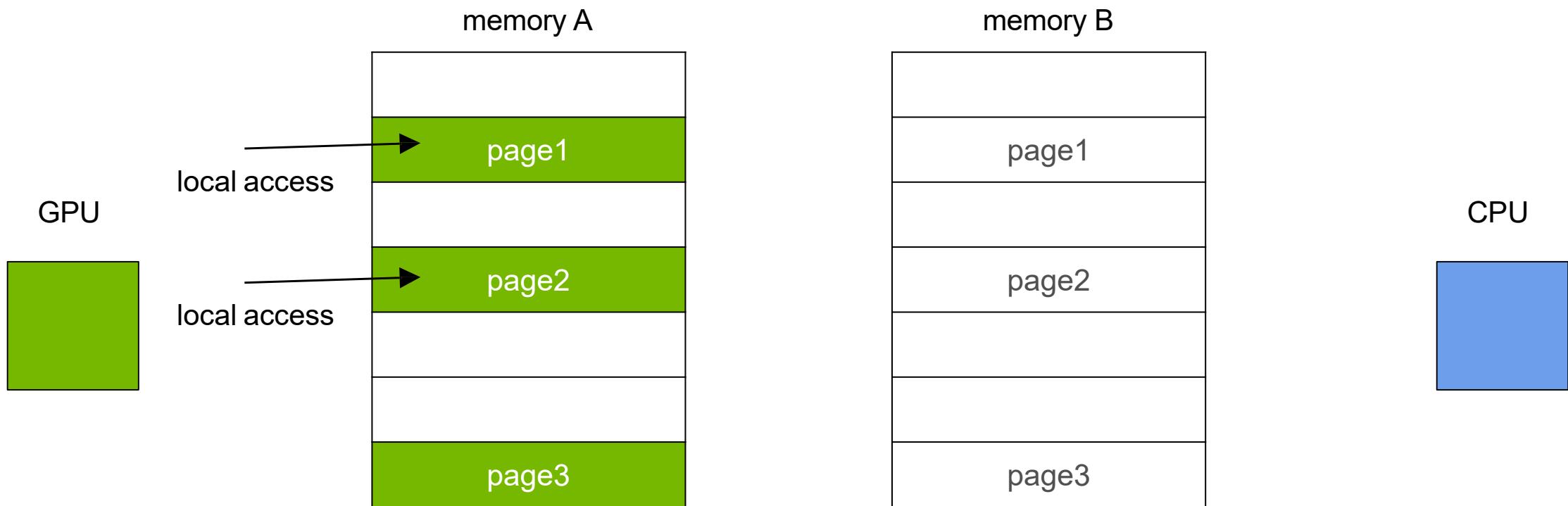
Bulk migration of **all pages** attached to current stream on kernel launch



# UNIFIED MEMORY ON KEPLER

Available since CUDA 6

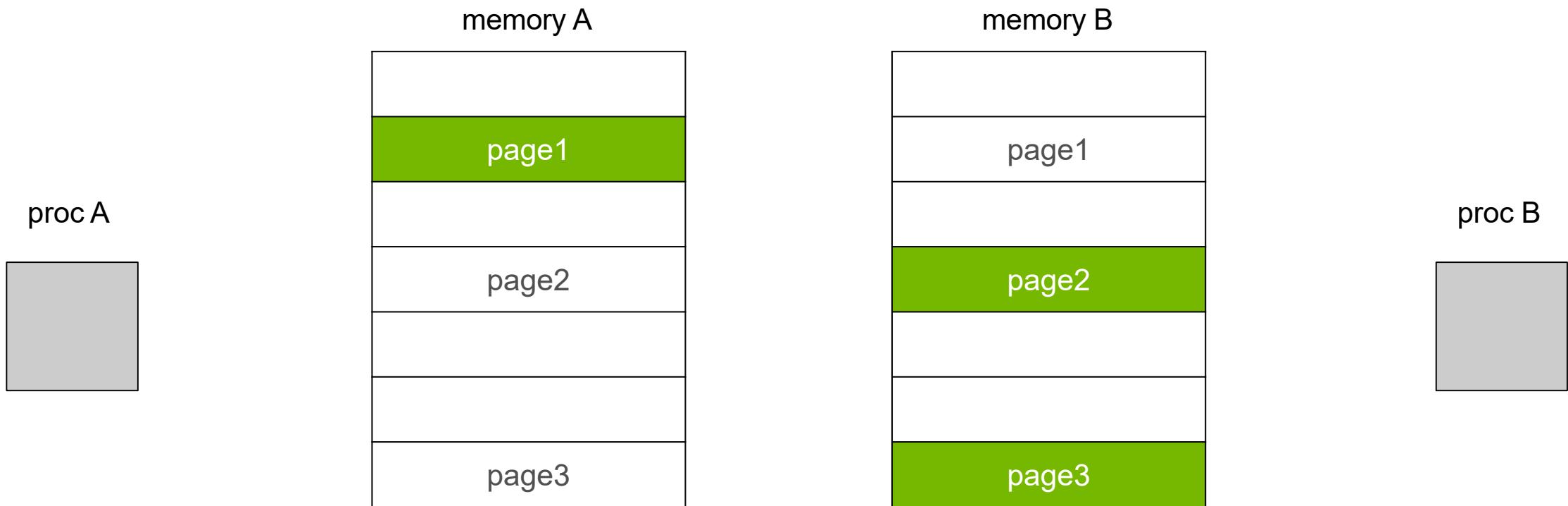
No on-demand migration for the GPU, no oversubscription, no system-wide atomics



# UNIFIED MEMORY ON PASCAL

Available since CUDA 8

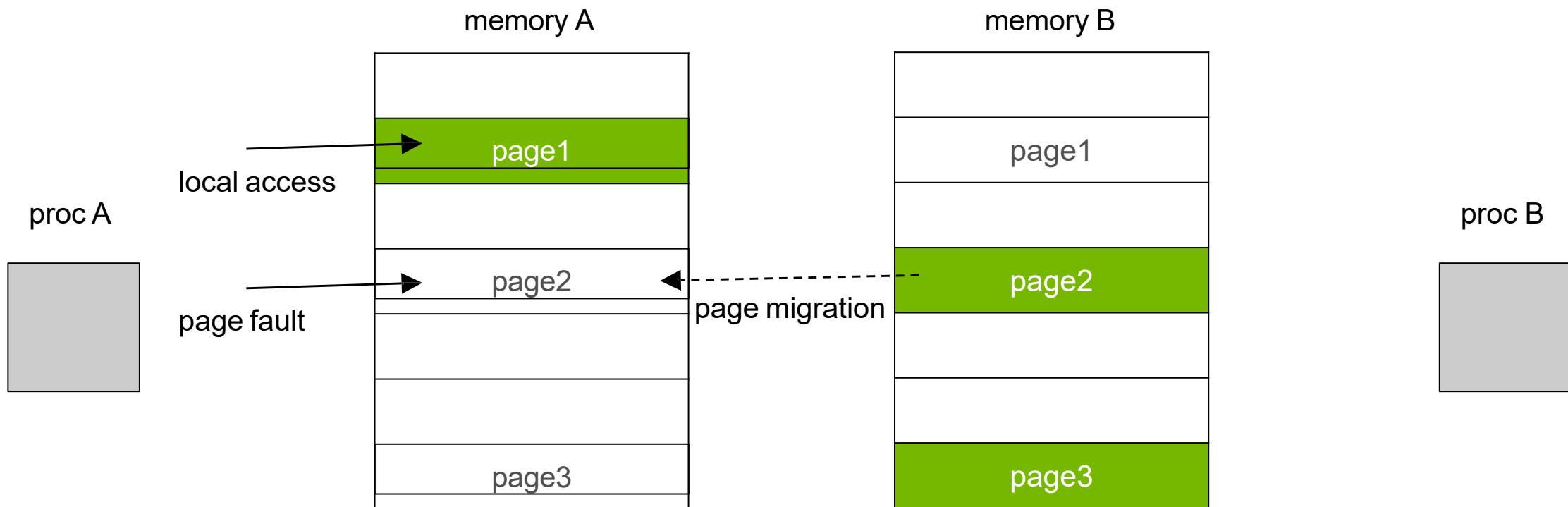
Pascal GPU: page fault support, extended virtual address space (48-bit)



# UNIFIED MEMORY ON PASCAL

Available since CUDA 8

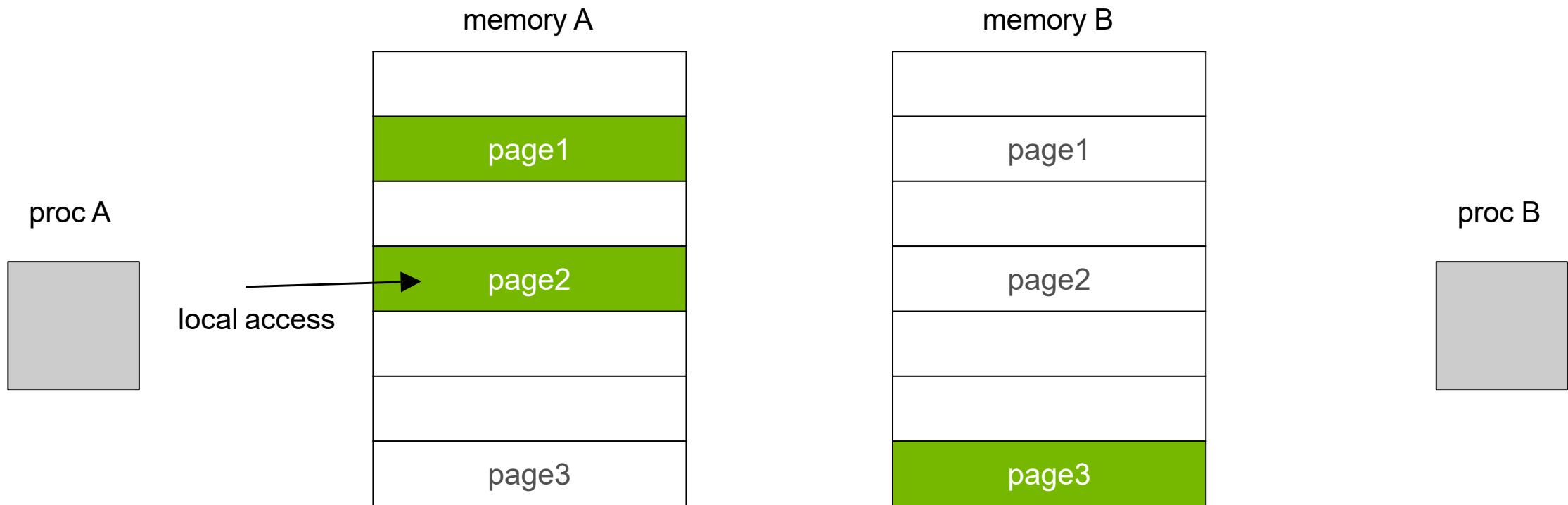
On-demand migration to accessing processor **on first touch**



# UNIFIED MEMORY ON PASCAL

Available since CUDA 8

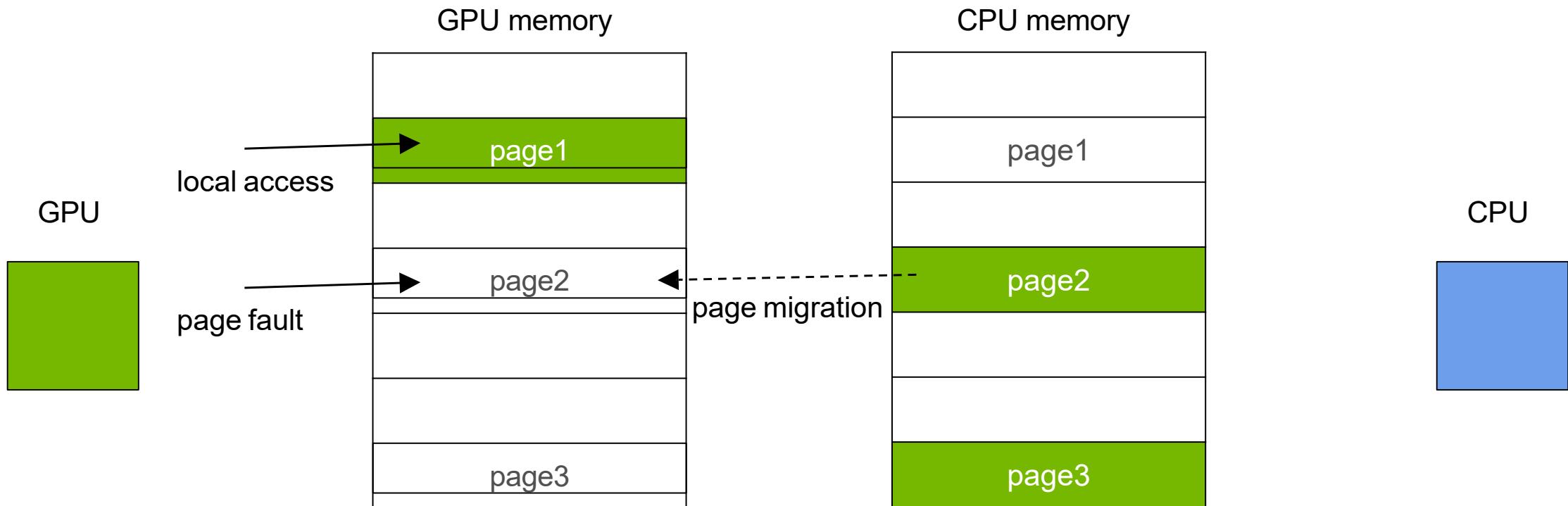
All features: on-demand migration, oversubscription, system-wide atomics



# UNIFIED MEMORY ON VOLTA

## Default model

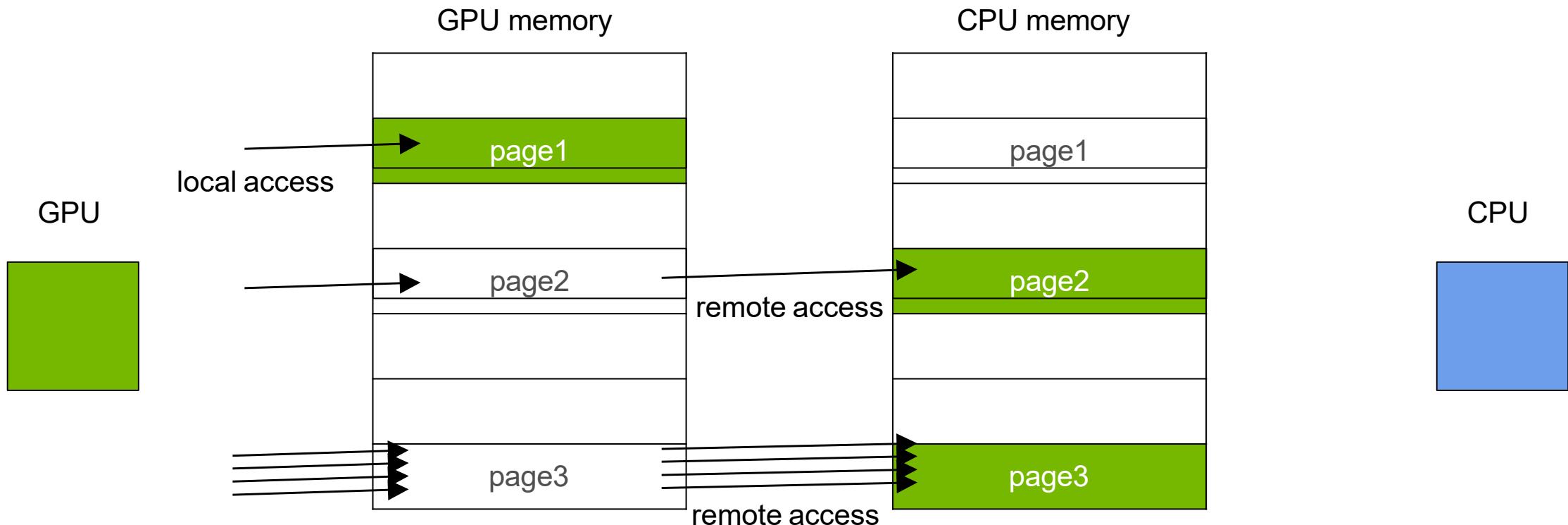
Volta GPU: uses fault on first touch for migration, same as Pascal



# UNIFIED MEMORY ON VOLTA

## New Feature: Access Counters

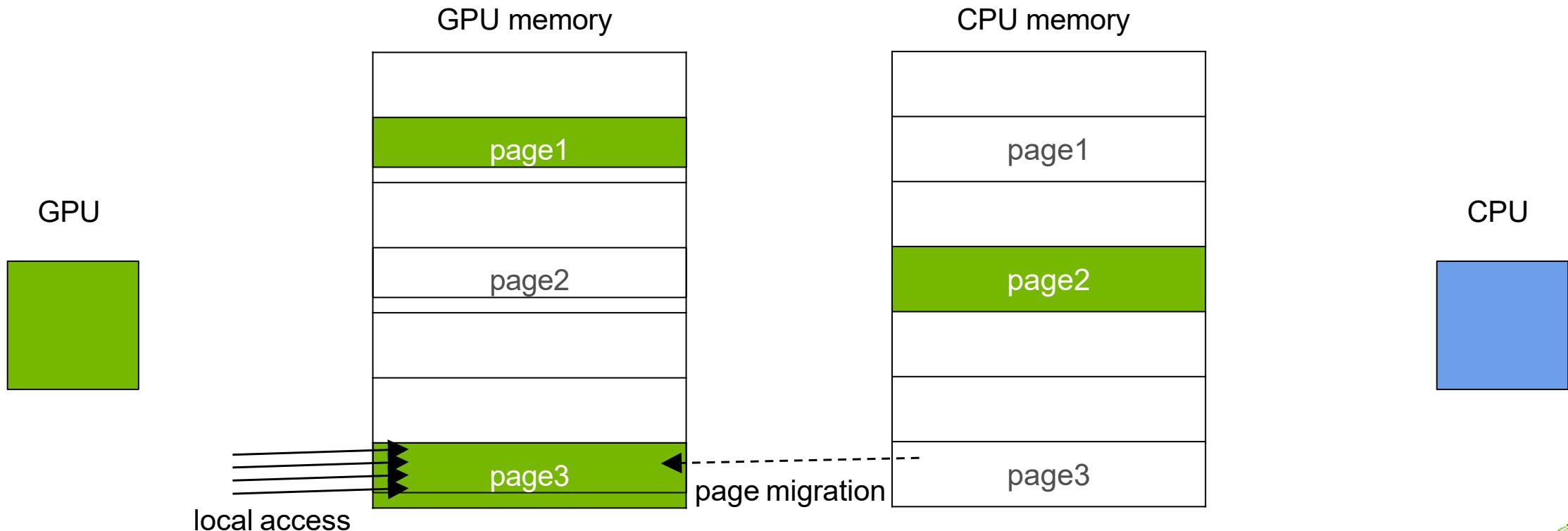
If memory is mapped to the GPU, migration can be triggered by access counters



# 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

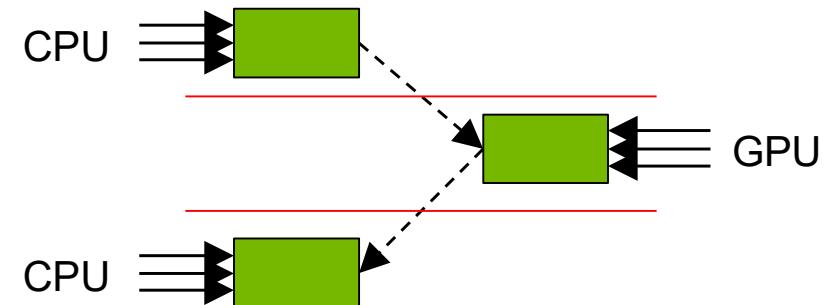
# USER HINTS

## 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

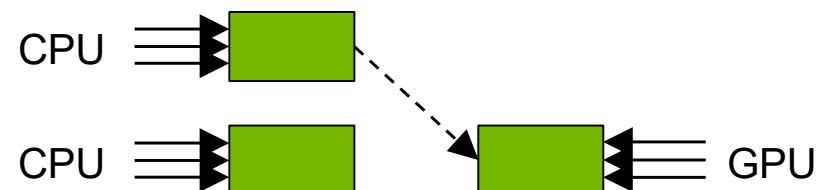
## Read Mostly

```
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



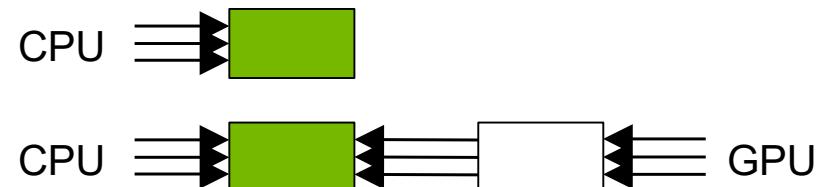
# USER HINTS

## 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



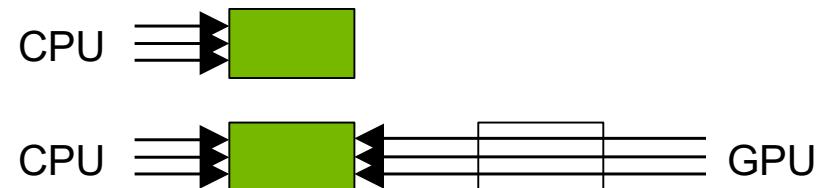
# USER HINTS

## 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



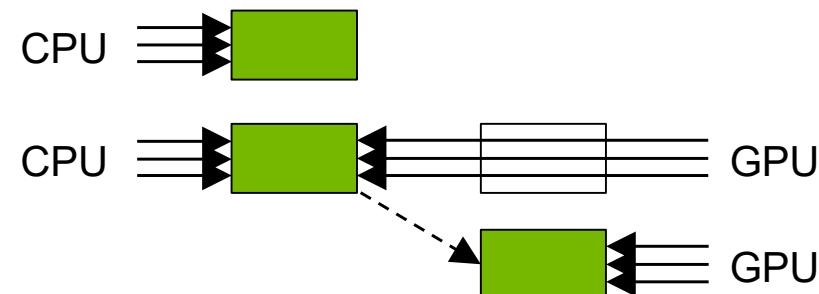
# USER HINTS

## 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>

# Concurrent CUDA Streams

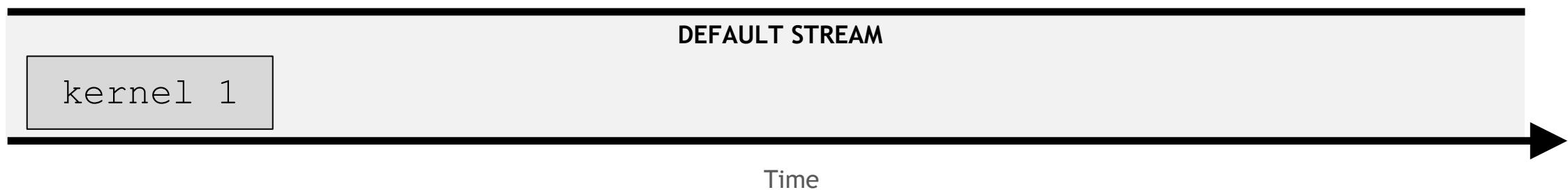
A **stream** is a series of instructions,  
and CUDA has a **default stream**

# DEFAULT STREAM

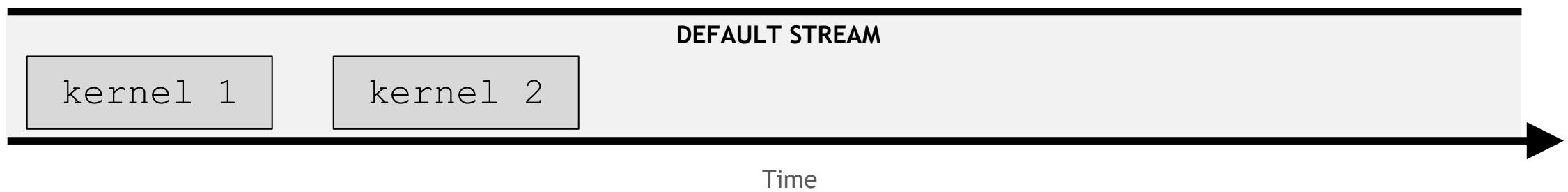
Time



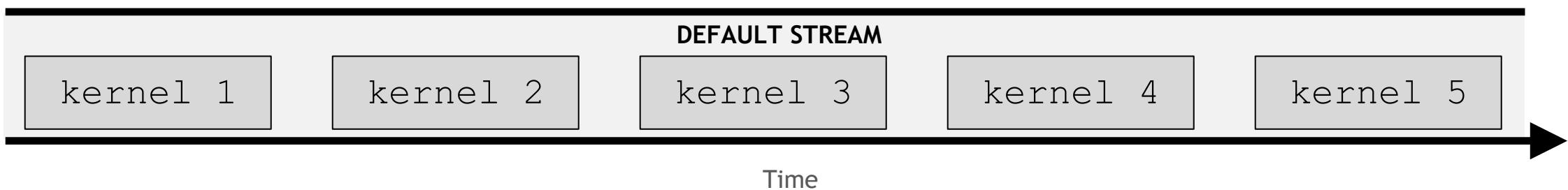
By default, CUDA kernels run in the  
**default stream**



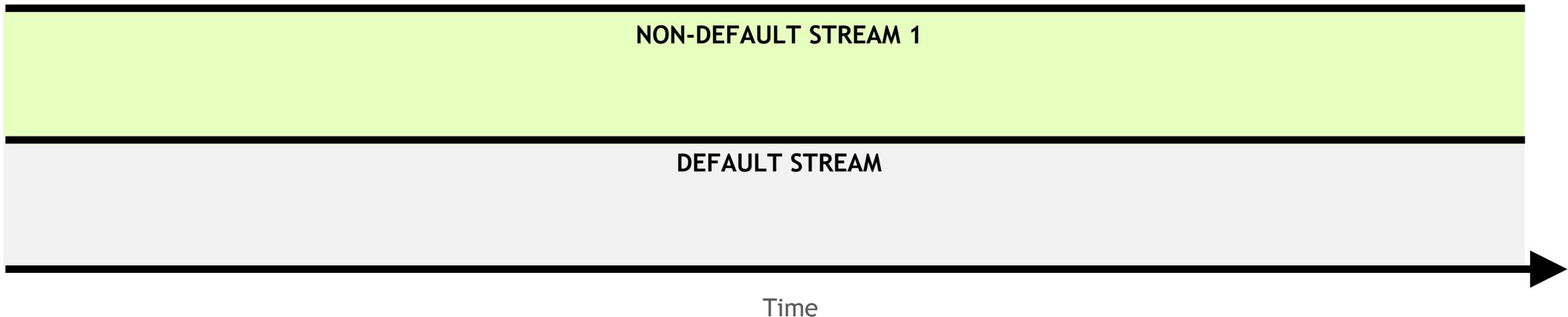
In any stream, including the default, an instruction in it (here a kernel launch) must complete before the next can begin



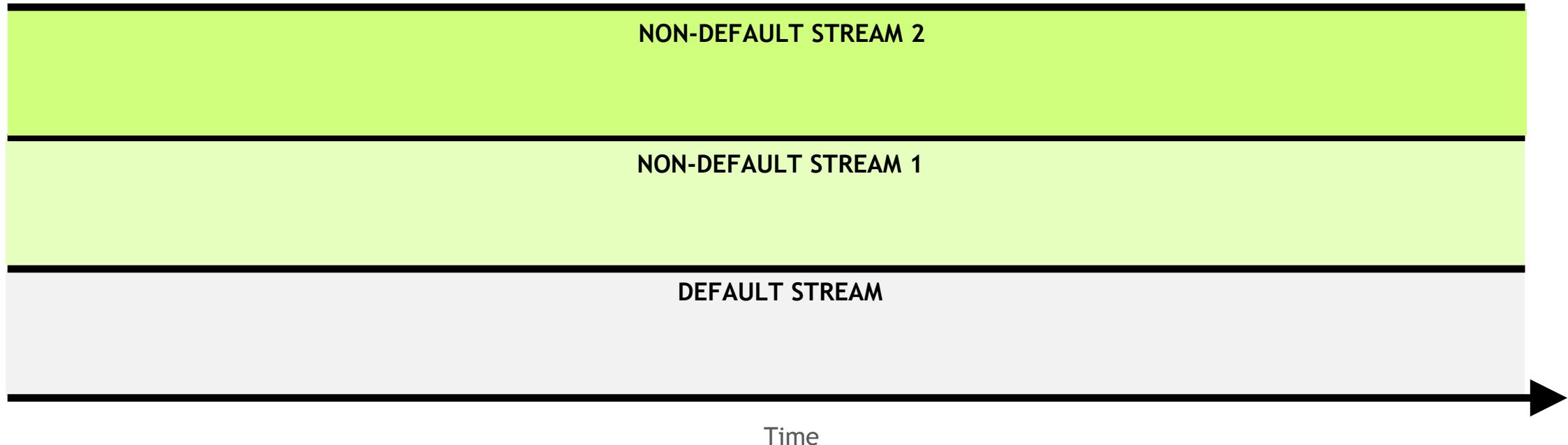
In any stream, including the default, an instruction in it (here a kernel launch) must complete before the next can begin



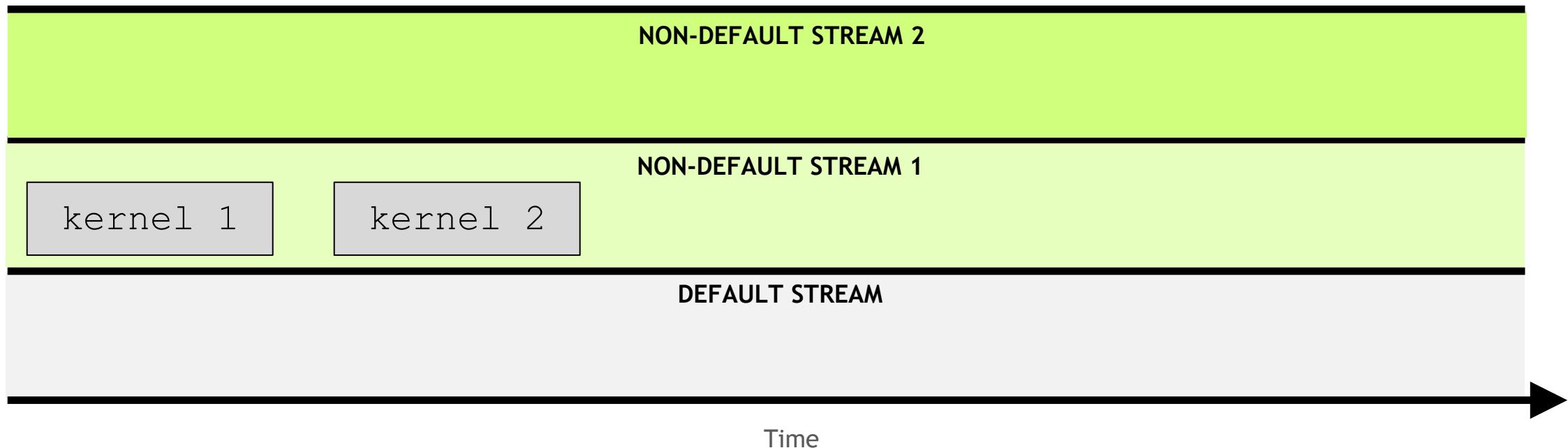
**Non-default streams** can also be created for kernel execution



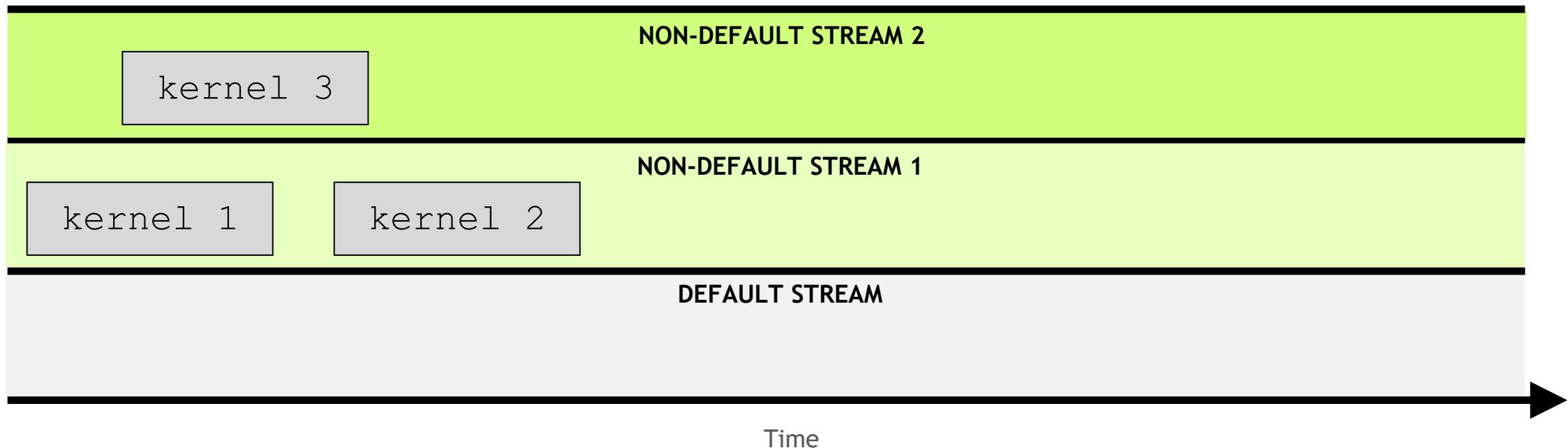
**Non-default streams** can also be created for kernel execution



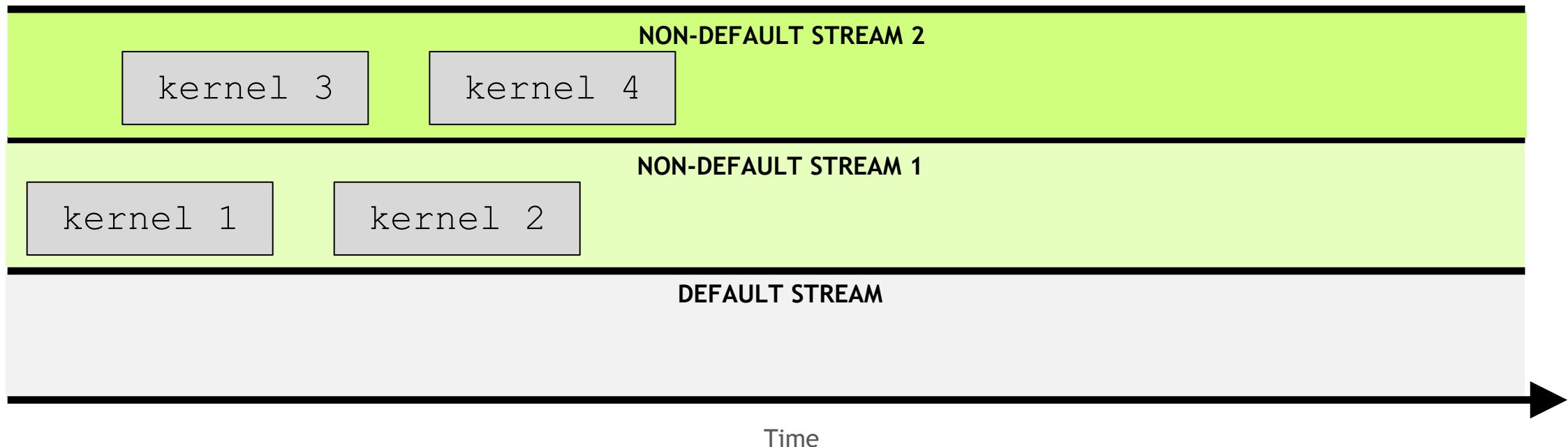
Kernels within any single stream must execute in order



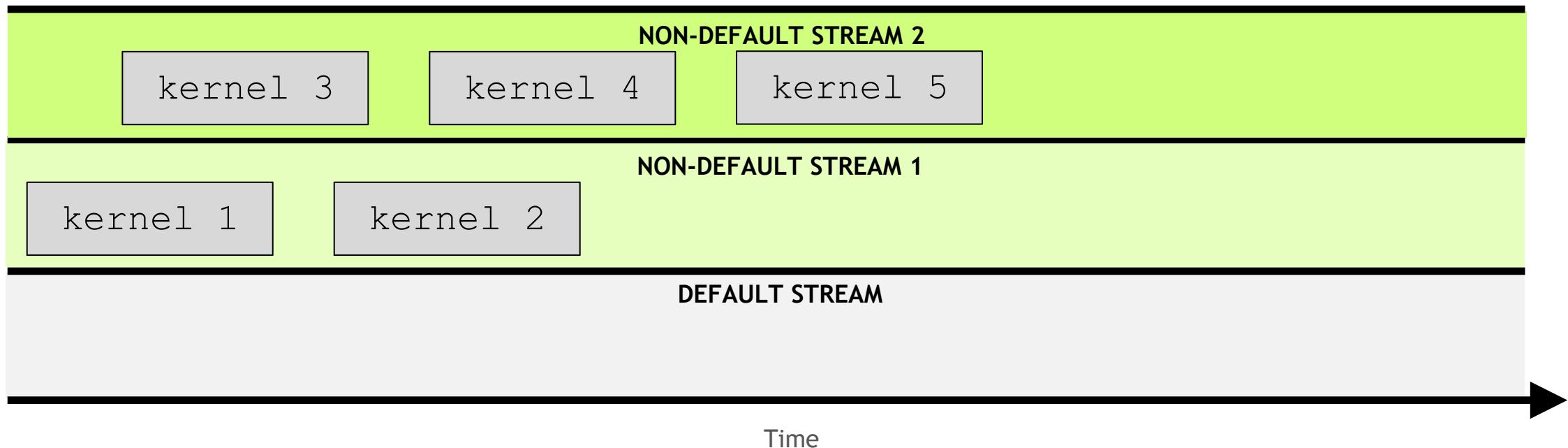
However, kernels in **different, non-default streams**, can interact concurrently



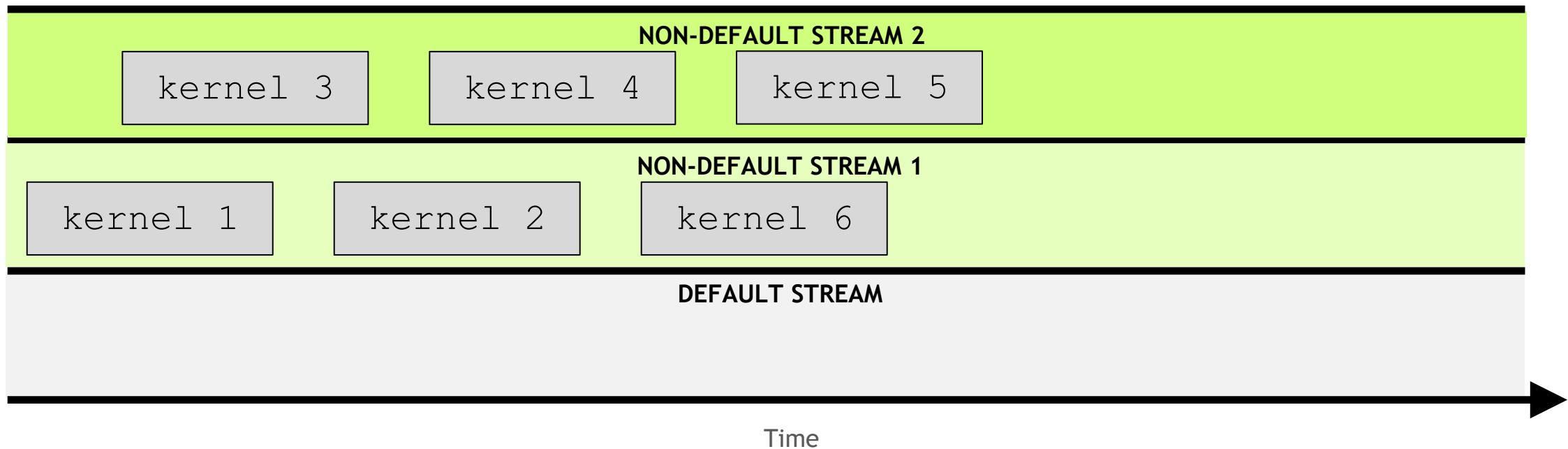
However, kernels in **different, non-default streams**, can interact concurrently



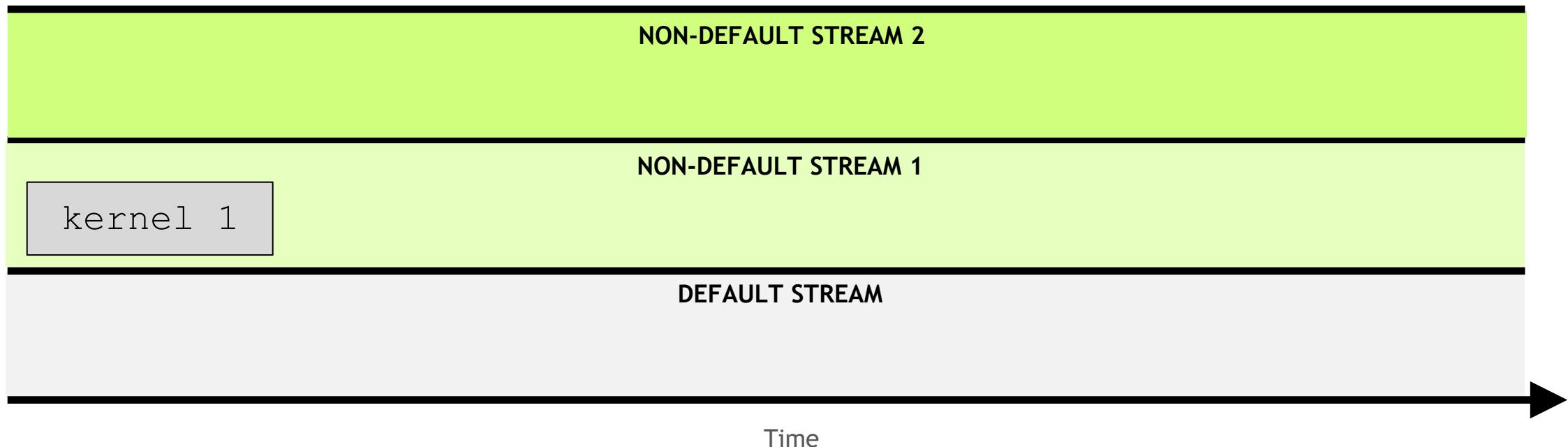
However, kernels in **different, non-default streams**, can interact concurrently



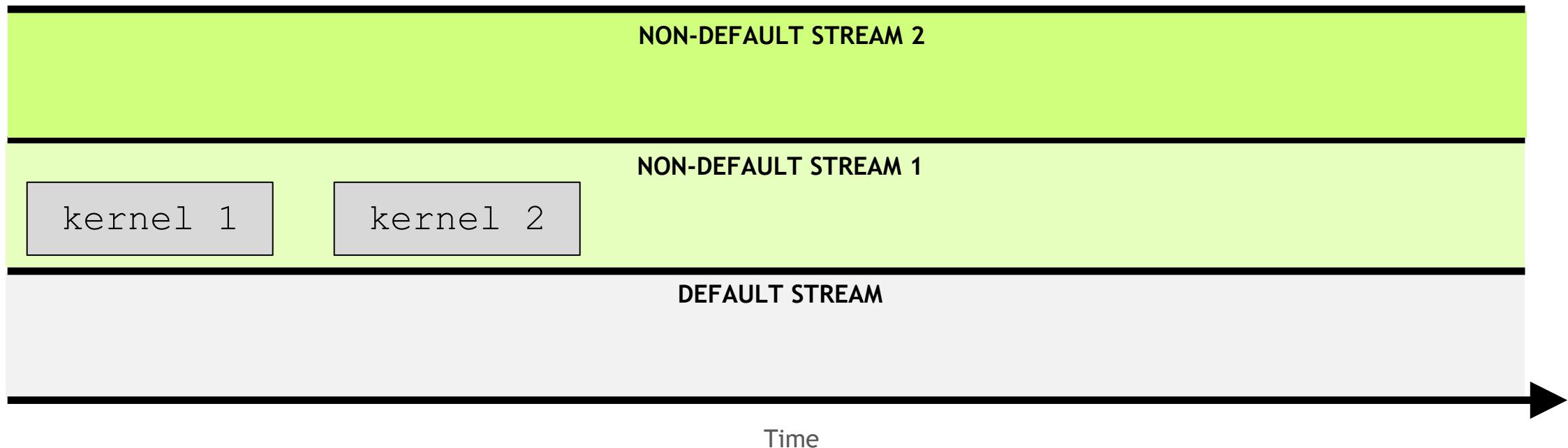
However, kernels in **different, non-default streams**, can interact concurrently



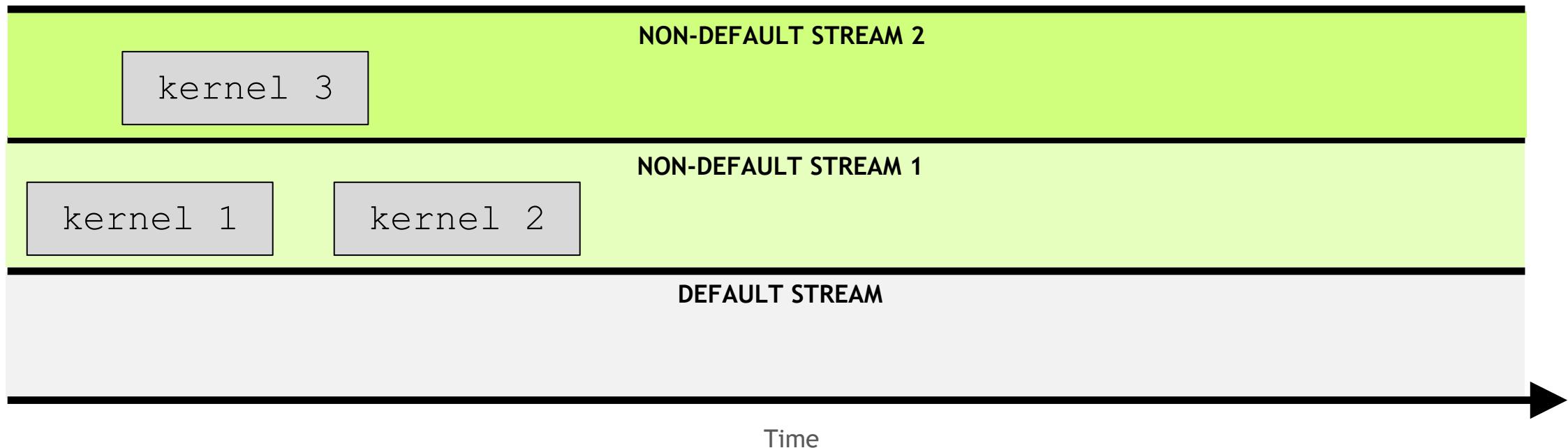
The default stream is special: it  
blocks all kernels in all other  
streams



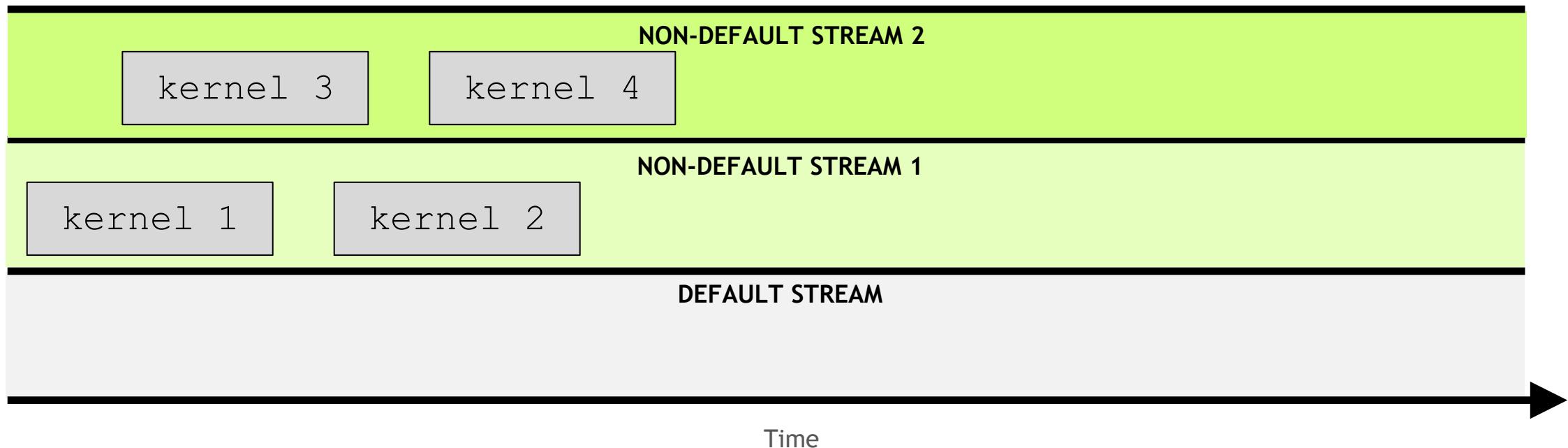
The default stream is special: it  
blocks all kernels in all other  
streams



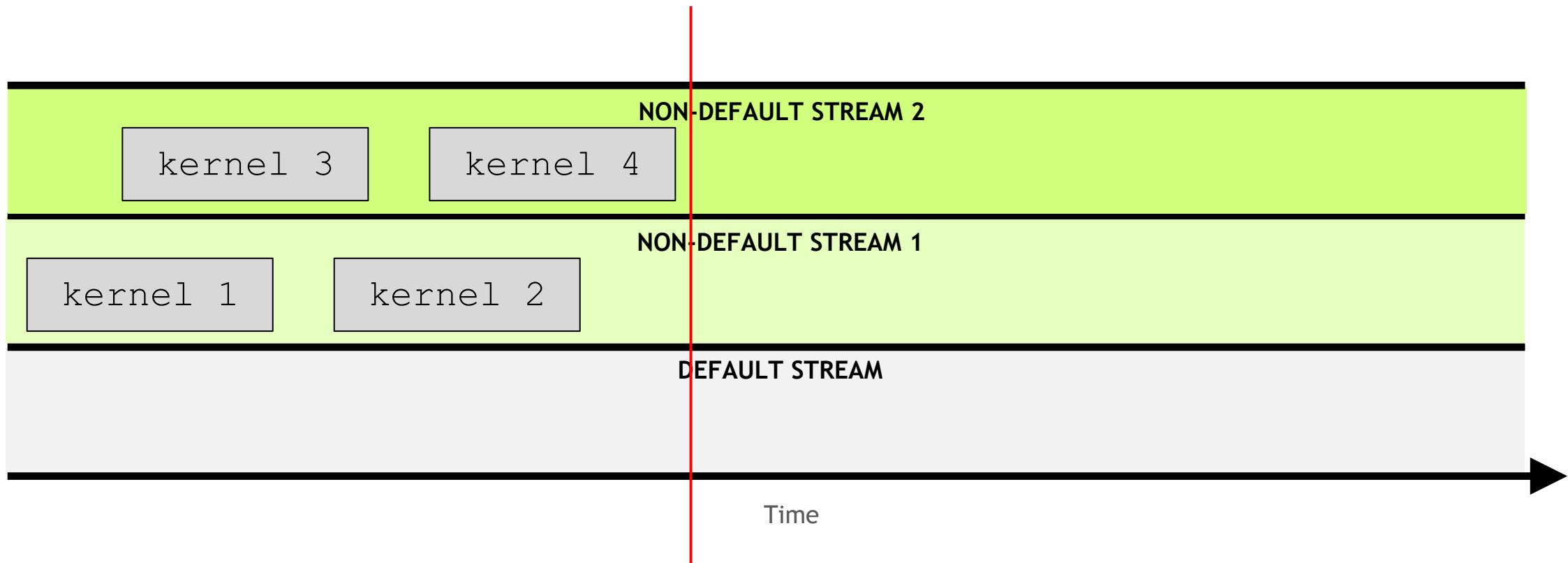
The default stream is special: it  
blocks all kernels in all other  
streams



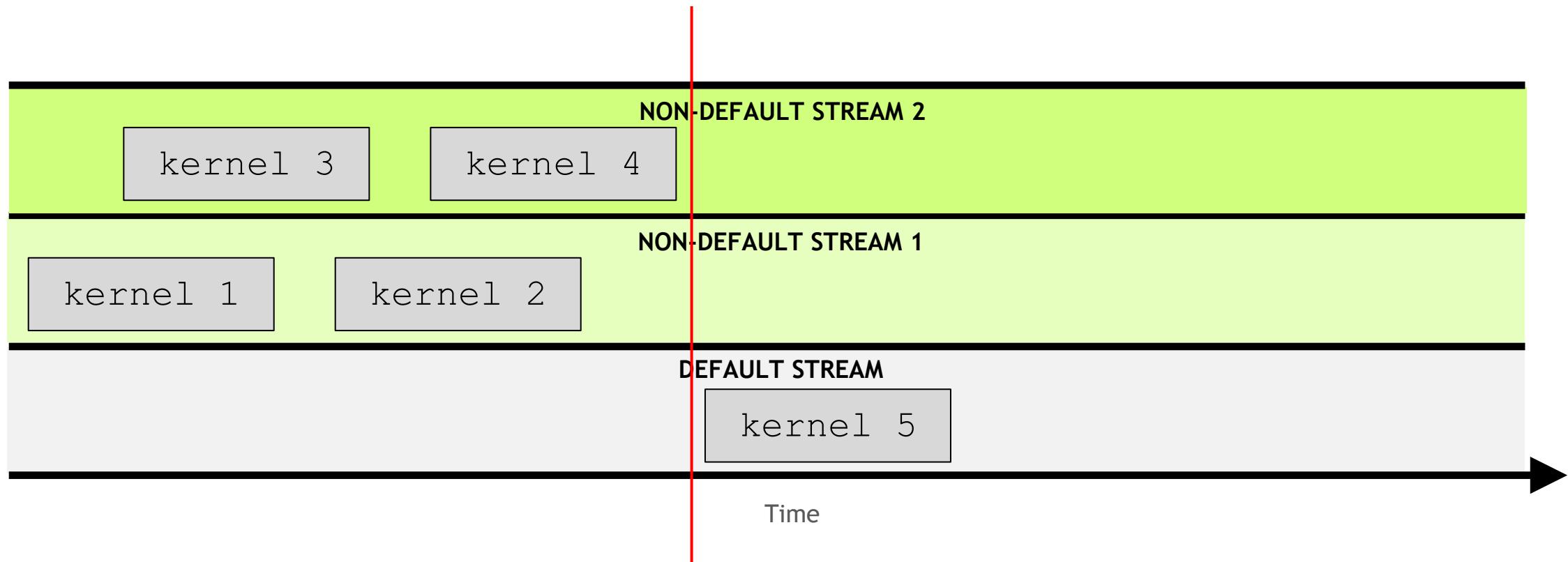
The default stream is special: it  
blocks all kernels in all other  
streams



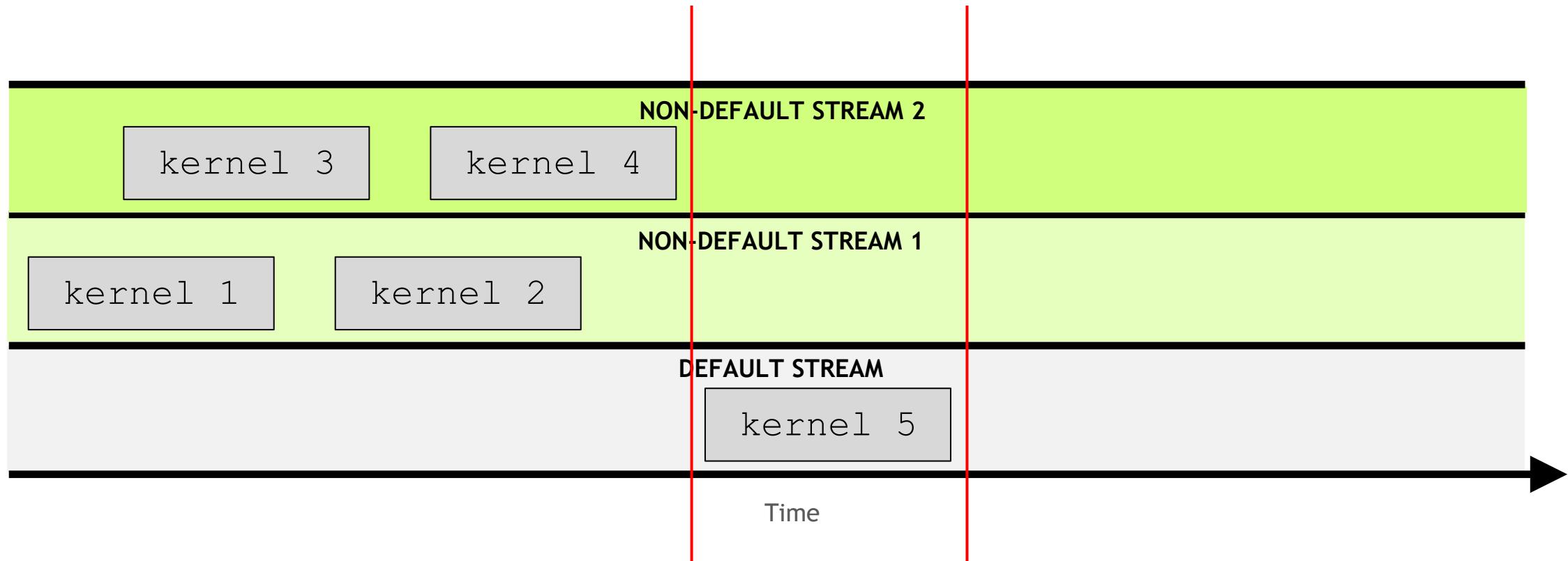
The default stream is special: it  
blocks all kernels in all other  
streams



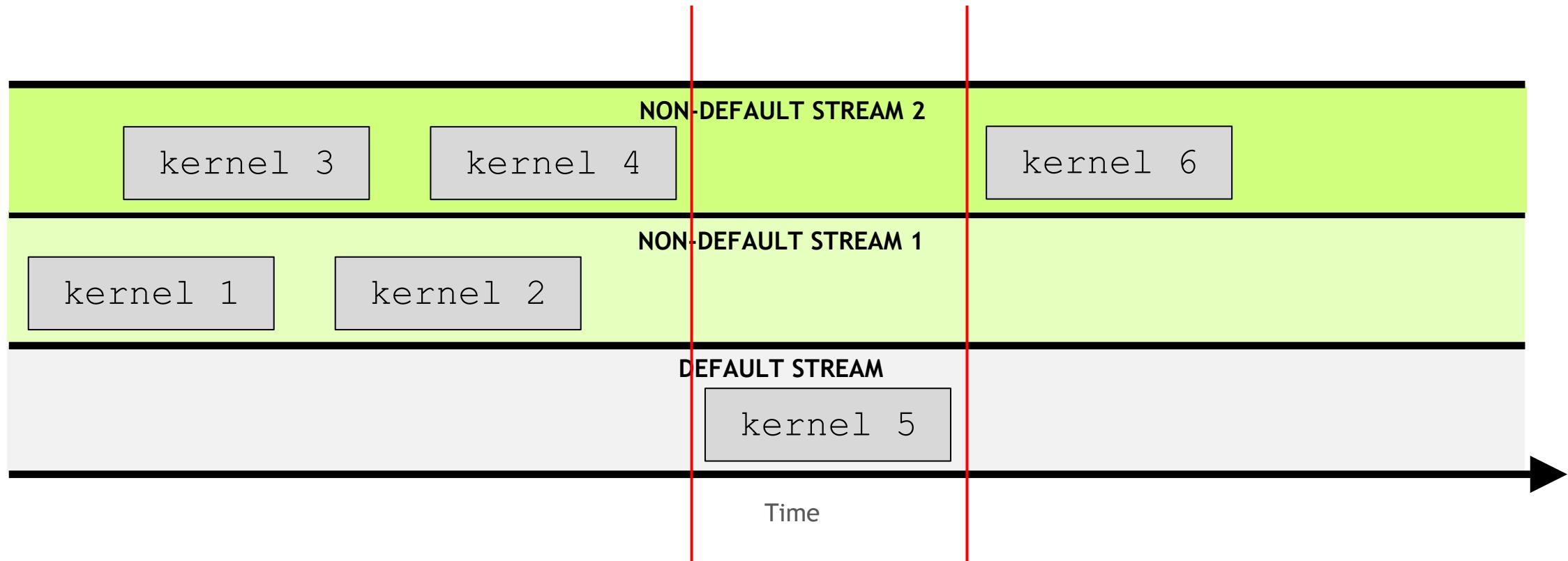
The default stream is special: it  
blocks all kernels in all other  
streams



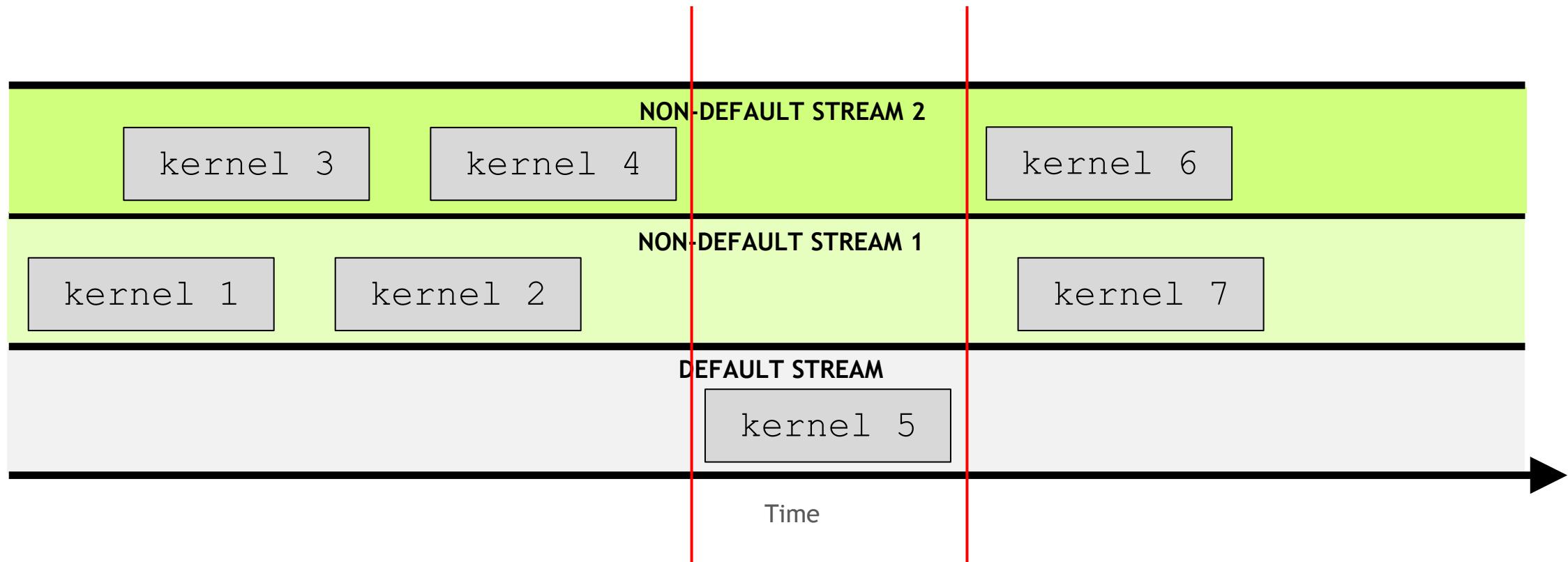
The default stream is special: it  
blocks all kernels in all other  
streams



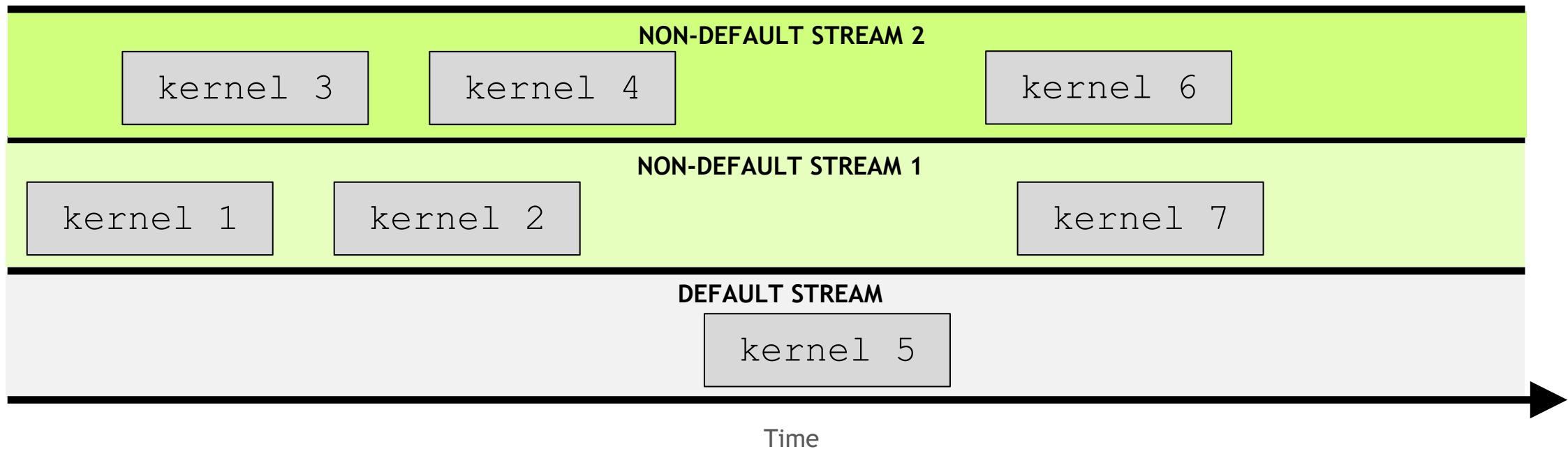
The default stream is special: it  
**blocks all kernels in all other streams**



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

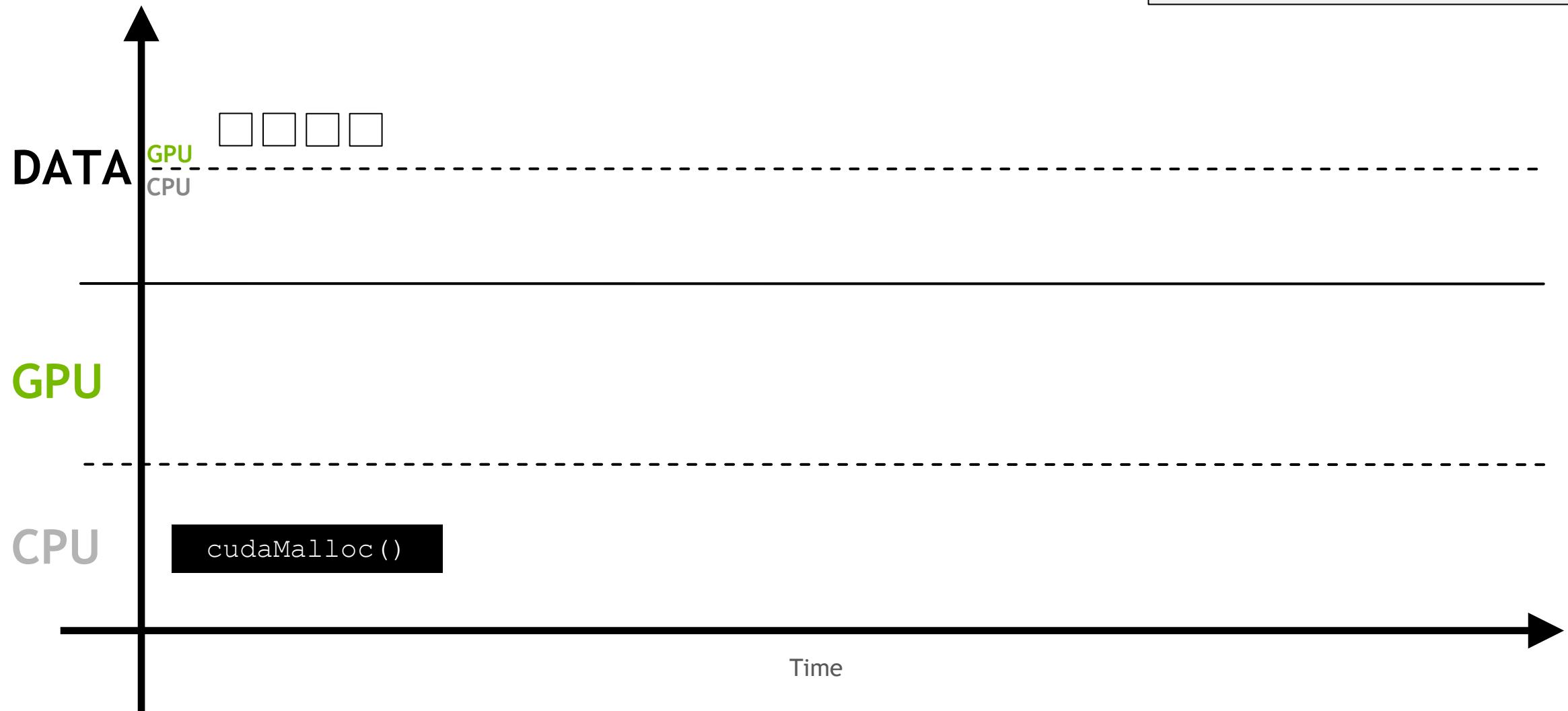


The default stream is special: it  
**blocks all kernels in all other streams**

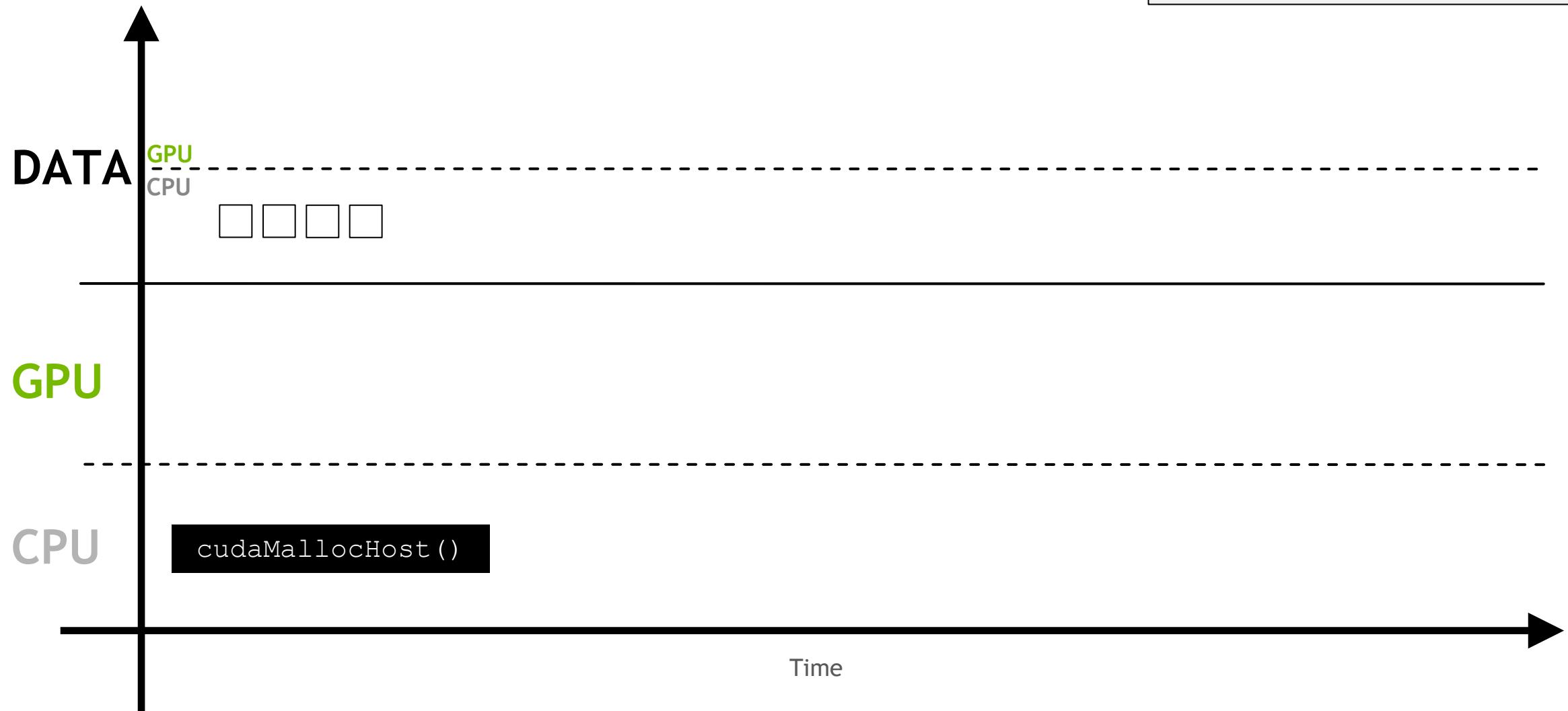


# Non-Unified Memory

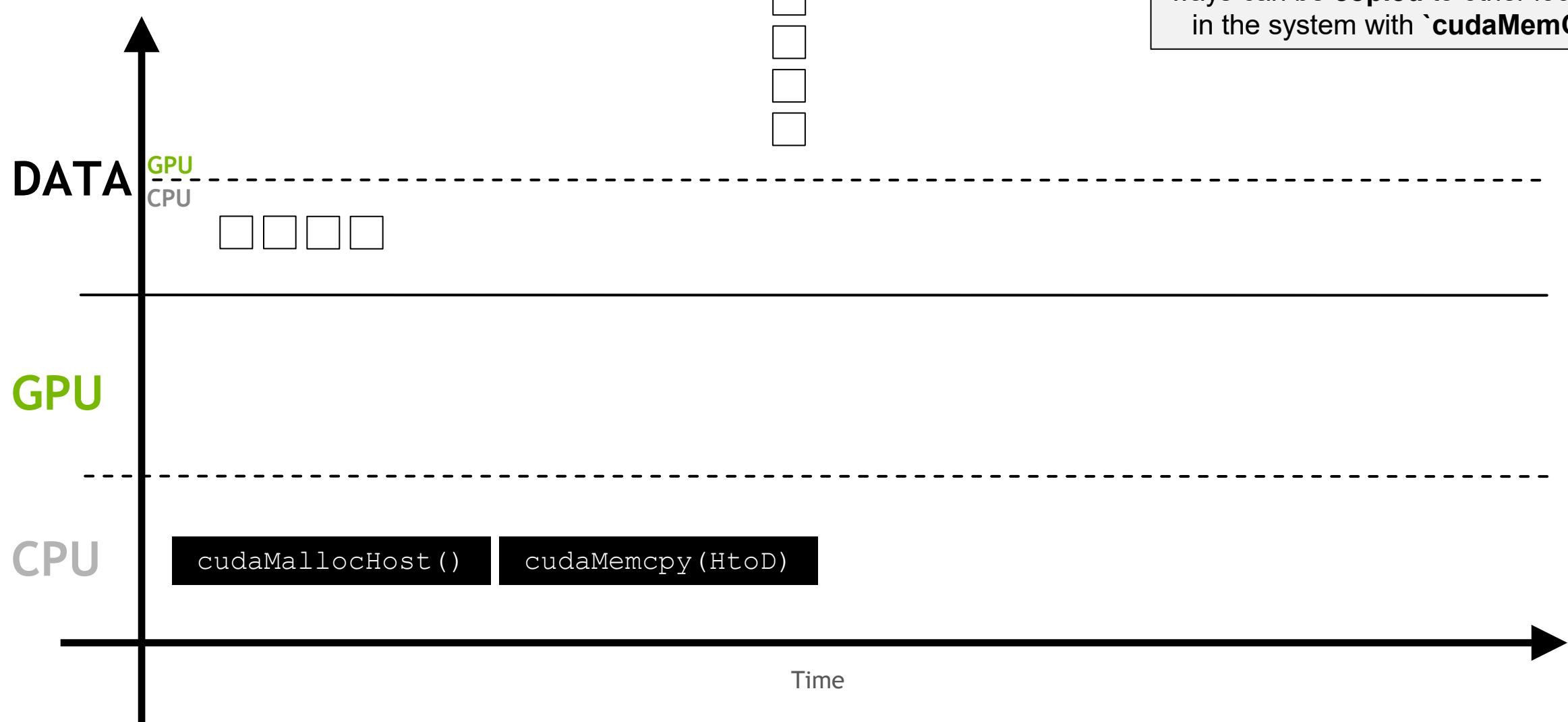
Memory can be allocated directly to  
the GPU with `cudaMalloc`



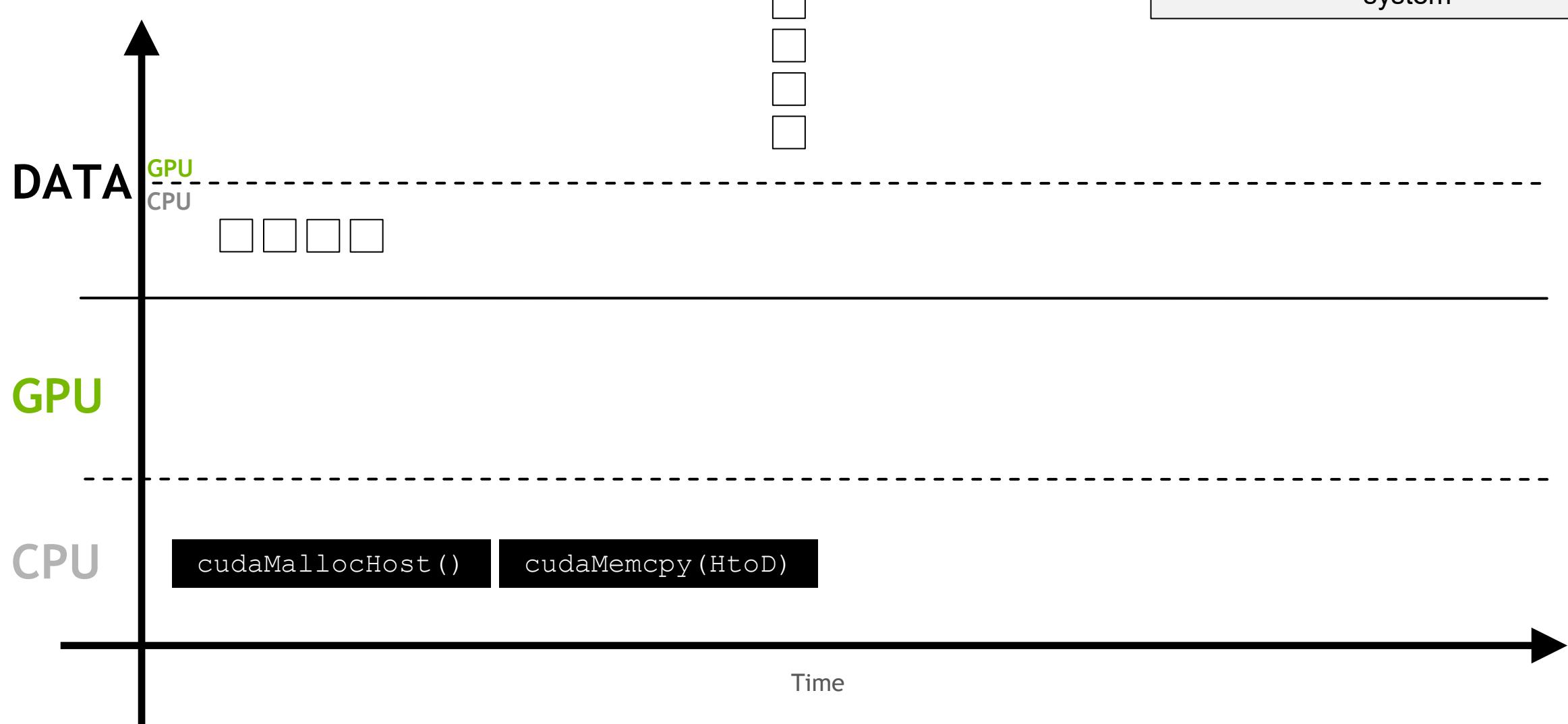
Memory can be allocated directly to  
the host with `'cudaMallocHost'`



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

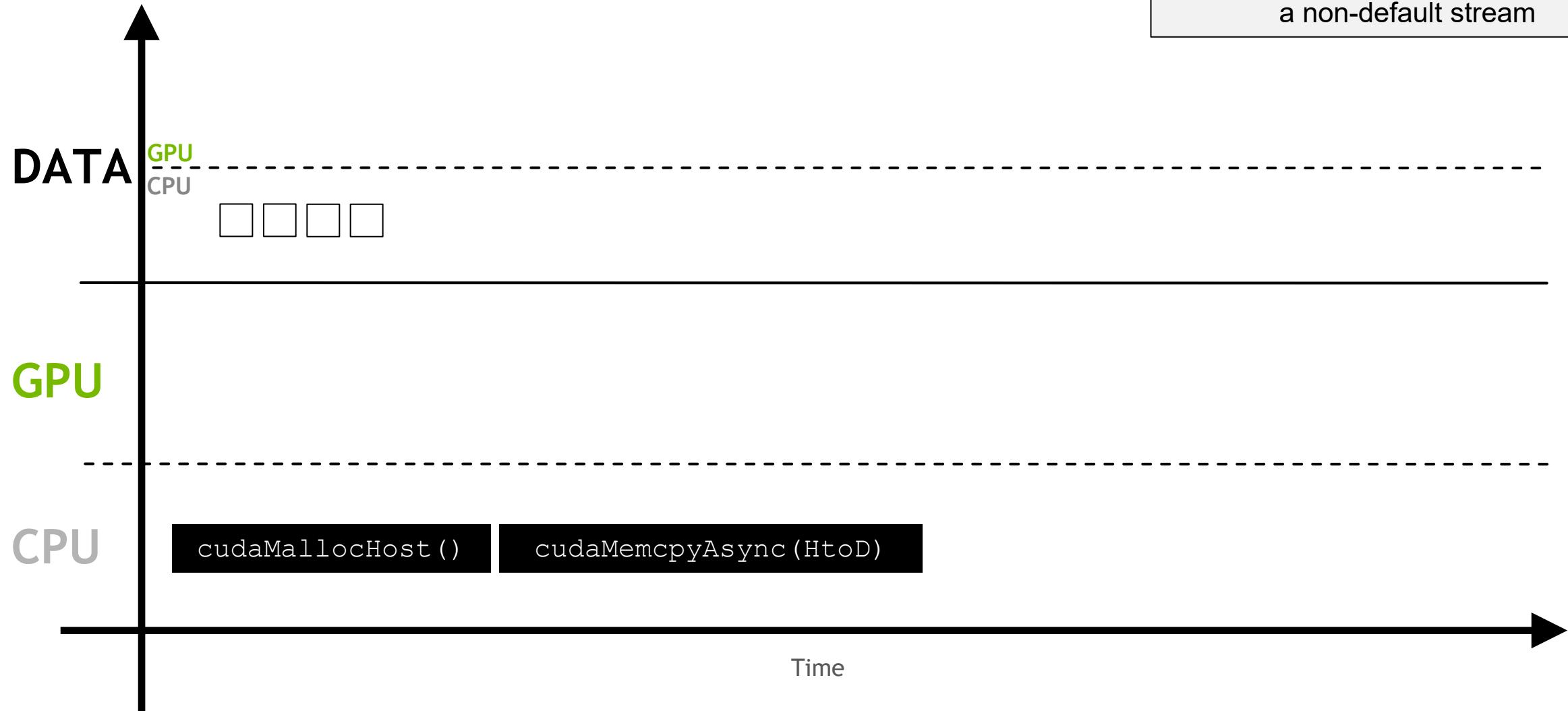


Copying leaves 2 copies in of in the system



# cudaMemcpyAsync

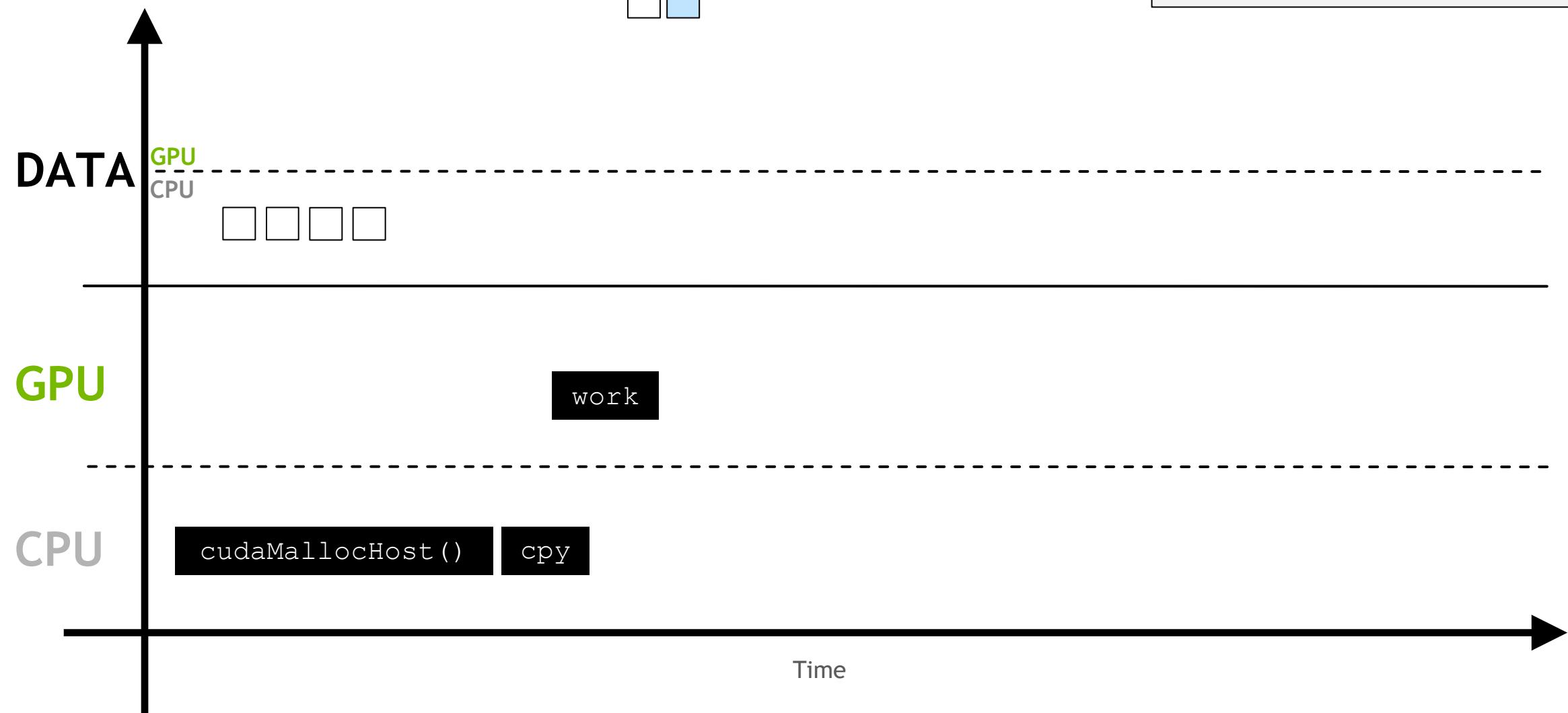
``cudaMemcpyAsync`` can asynchronously transfer memory over a non-default stream



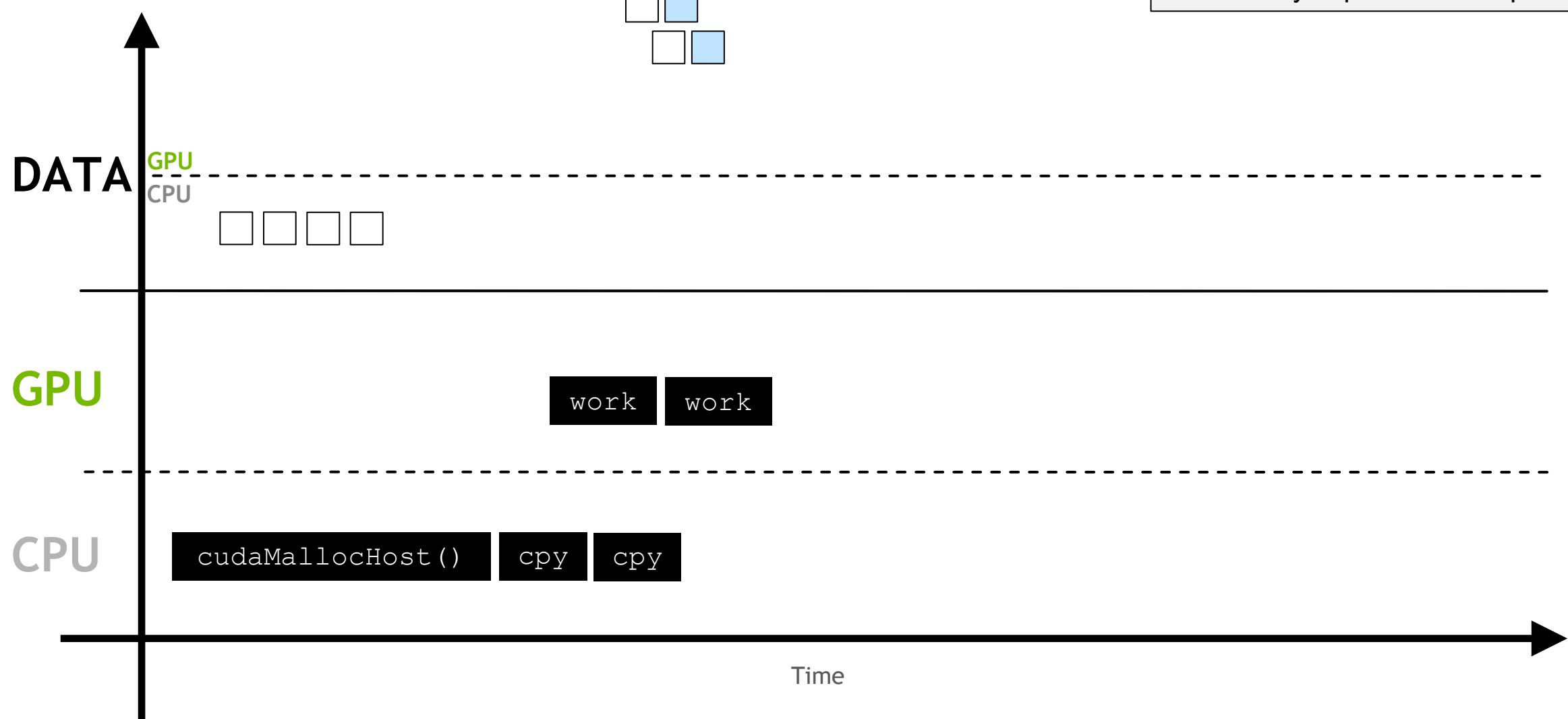
This can allow the **overlapping** memory copies and computation



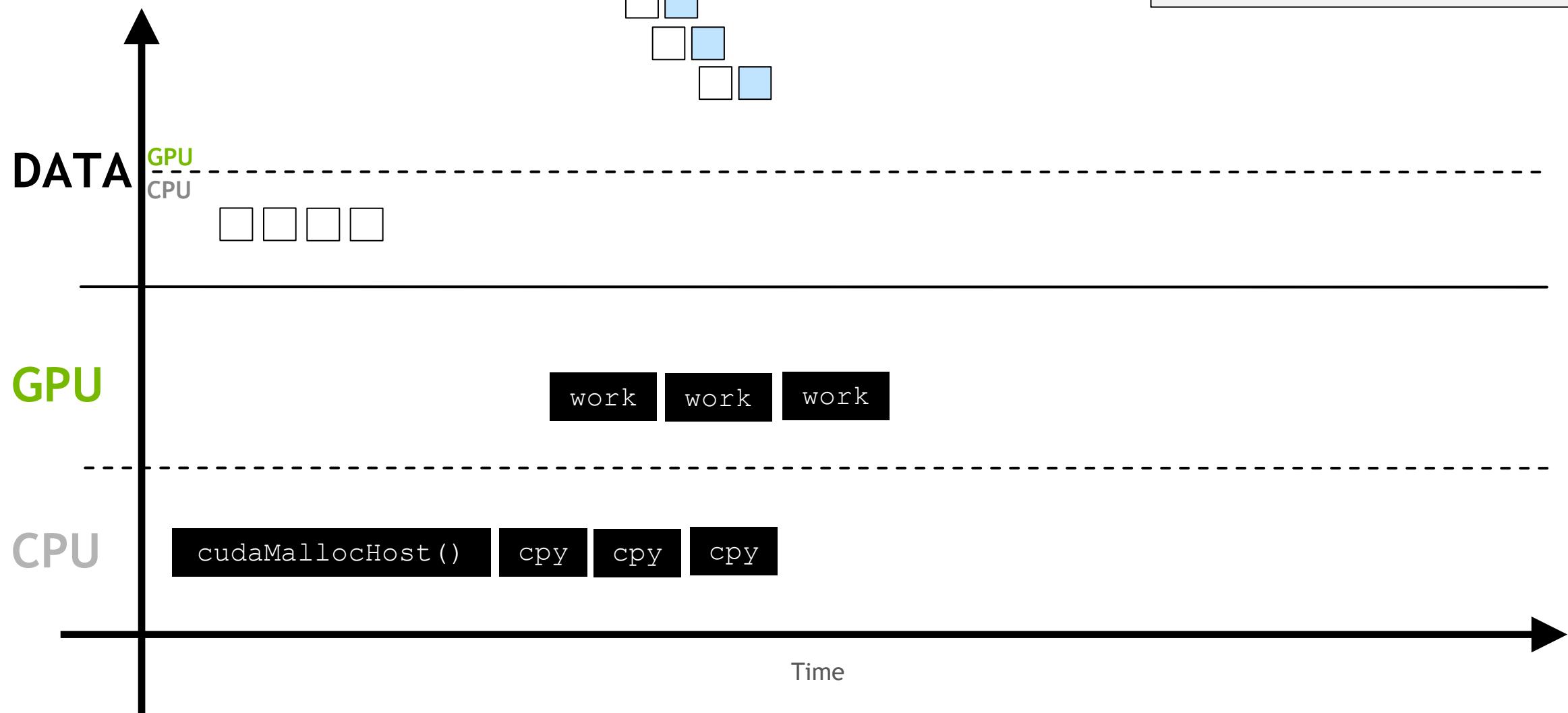
This can allow the **overlapping** memory copies and computation



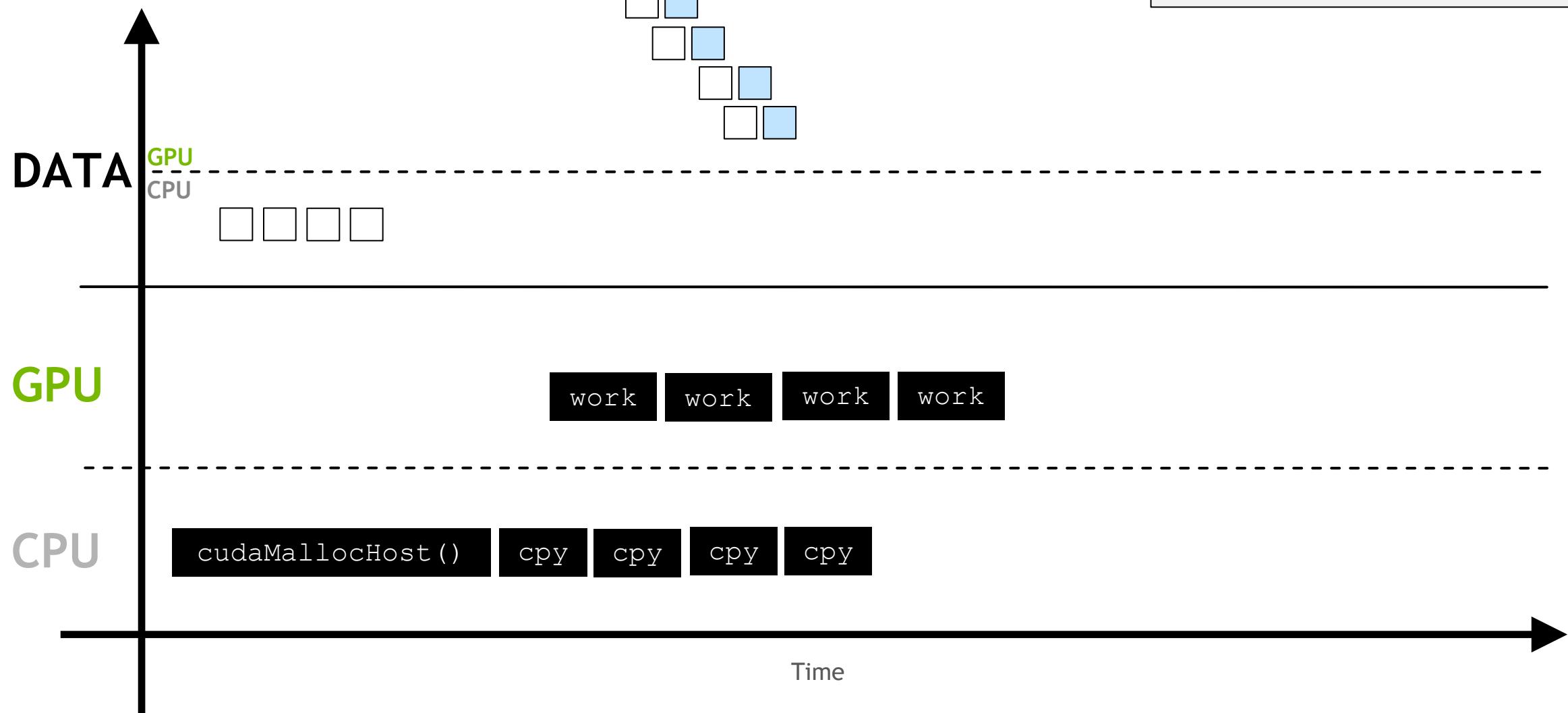
This can allow the **overlapping** memory copies and computation



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



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



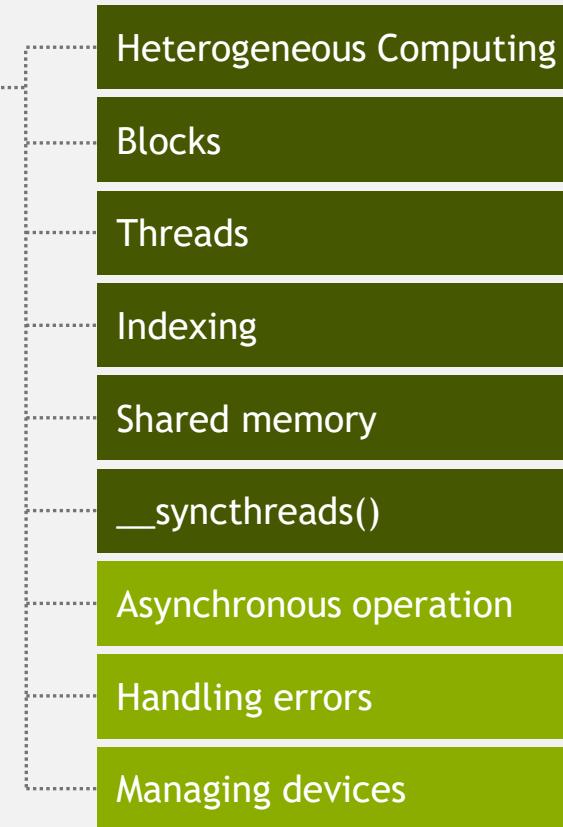


# Questions?

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

# 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

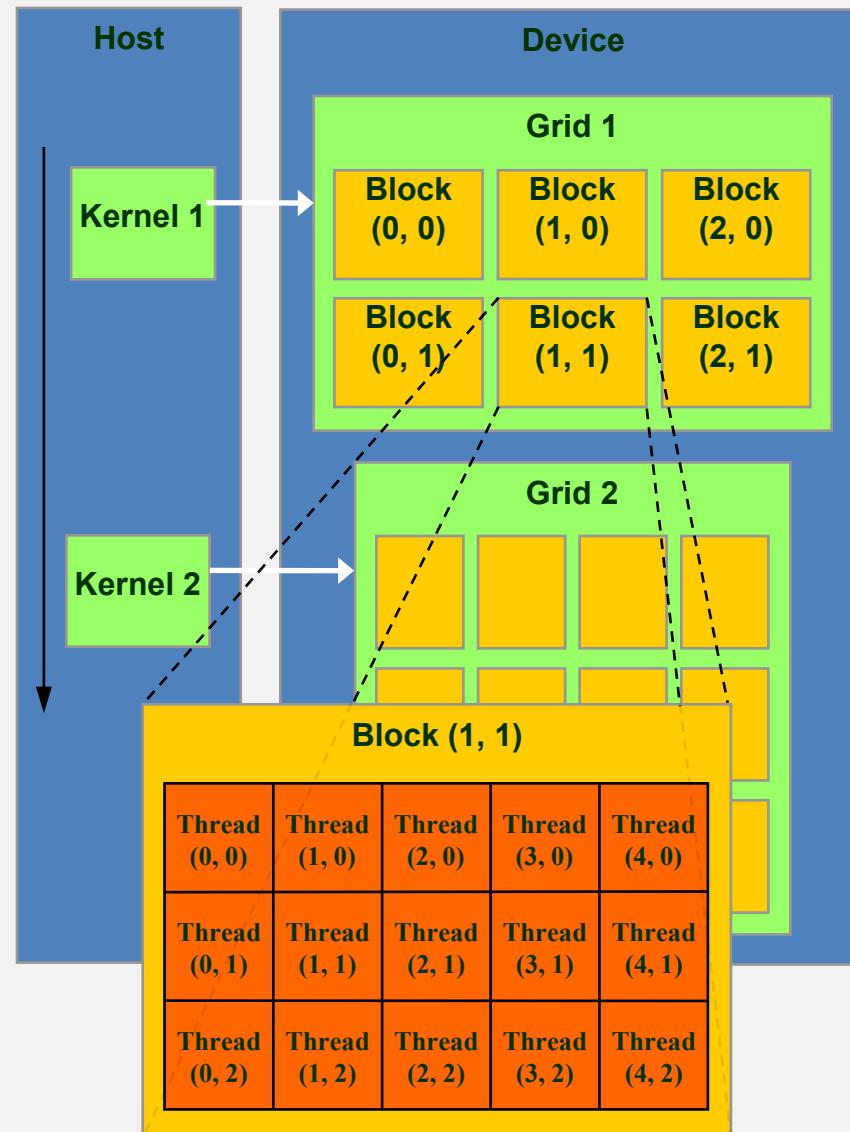
vRelease Version | July 2019



# Advanced Topics: Reduce, Scan, Sort

# Programming Model (SPMD + SIMD): Thread Batching

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Efficiently sharing data through shared memory
  - Synchronizing their execution
    - For hazard-free shared memory accesses
- Two threads from two different blocks cannot cooperate
  - Blocks are independent

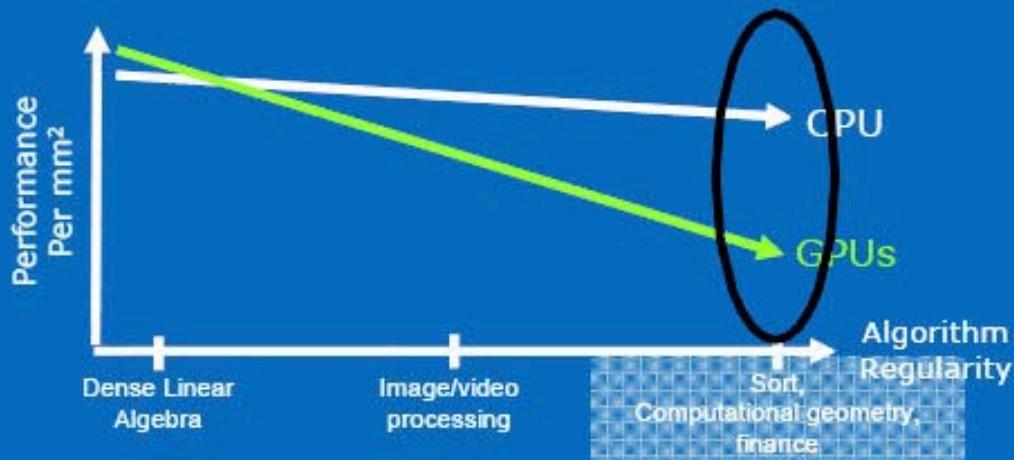


# Basic Efficiency Rules

- Develop algorithms with a data parallel mindset
- Minimize divergence of execution within blocks
- Maximize locality of global memory accesses
- Exploit per-block shared memory as scratchpad  
(registers > shared memory > global memory)
- Expose enough parallelism

# Expanding Manycore Territory

## Algorithm Examples



- Sort, computational geometry, finance
  - Modest control flow
  - Sparse/Irregular data structures
  - Irregular communication between elements
- CPU Territory
  - General purpose features vital for software efficiency
  - Latency sensitive applications



# Today's Big Picture

Complexity =  $k(O(f(n)))$

```
graph TD; Today[Today] -- blue arrow --> Complexity["Complexity = k(O(f(n)))"]; Rest[Rest of class] -- red arrow --> Complexity;
```

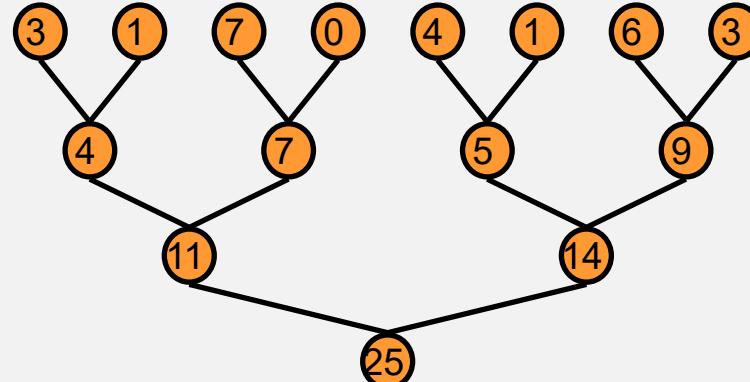
Today

Rest of class

# Reduction

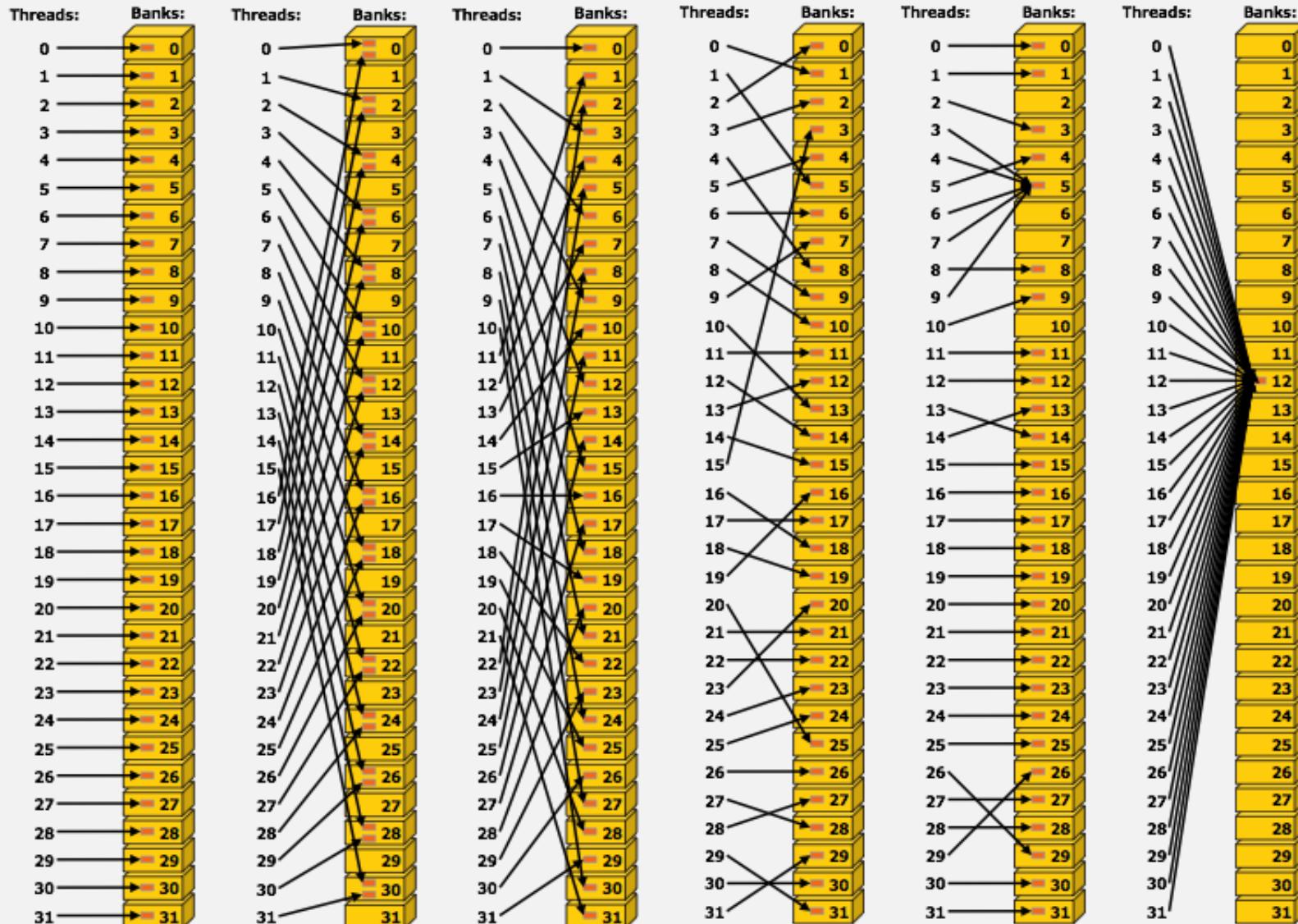


# Tree-Based Parallel Reductions



- 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.

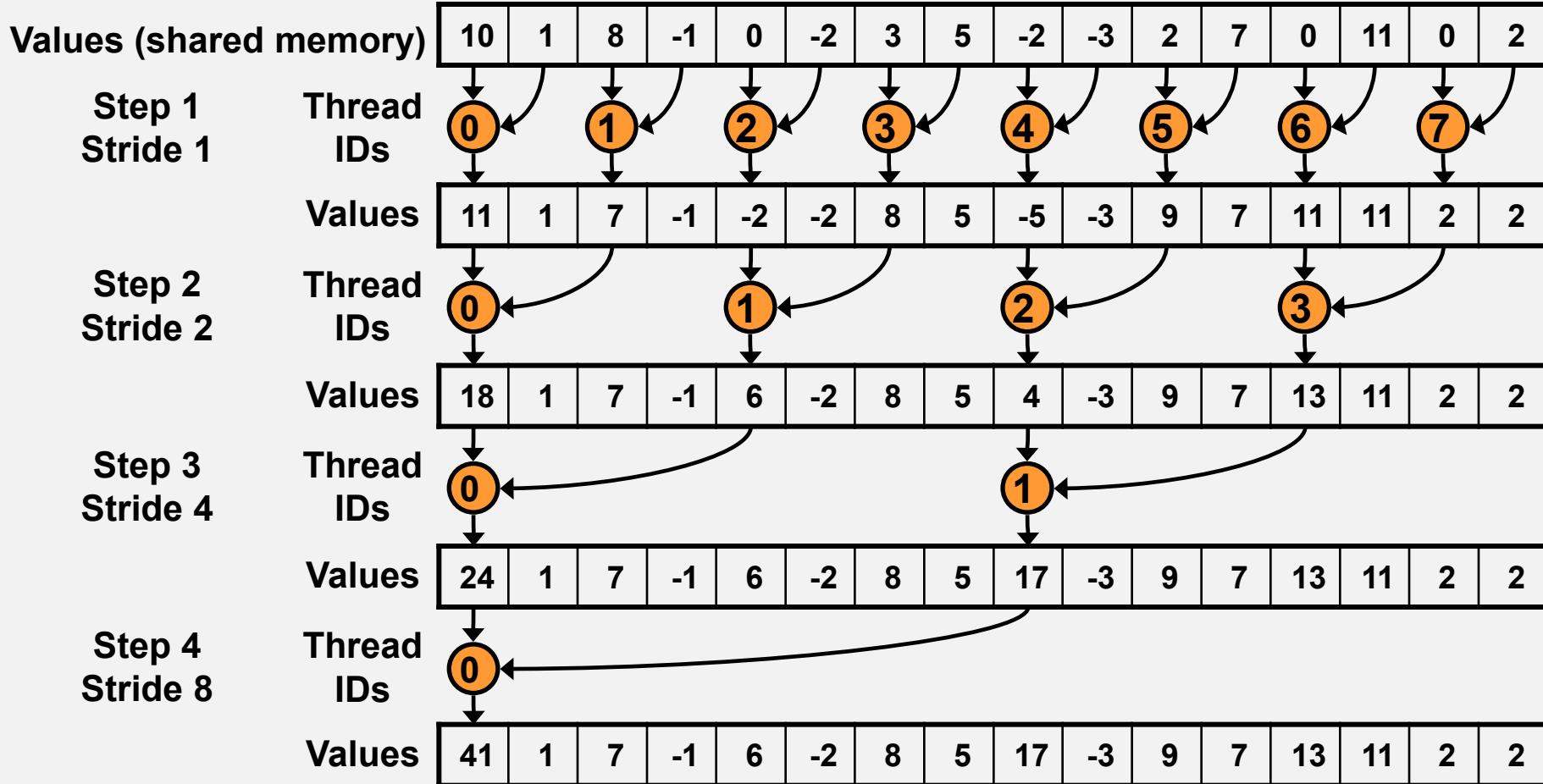
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.

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



# Parallel Reduction: Interleaved Addressing

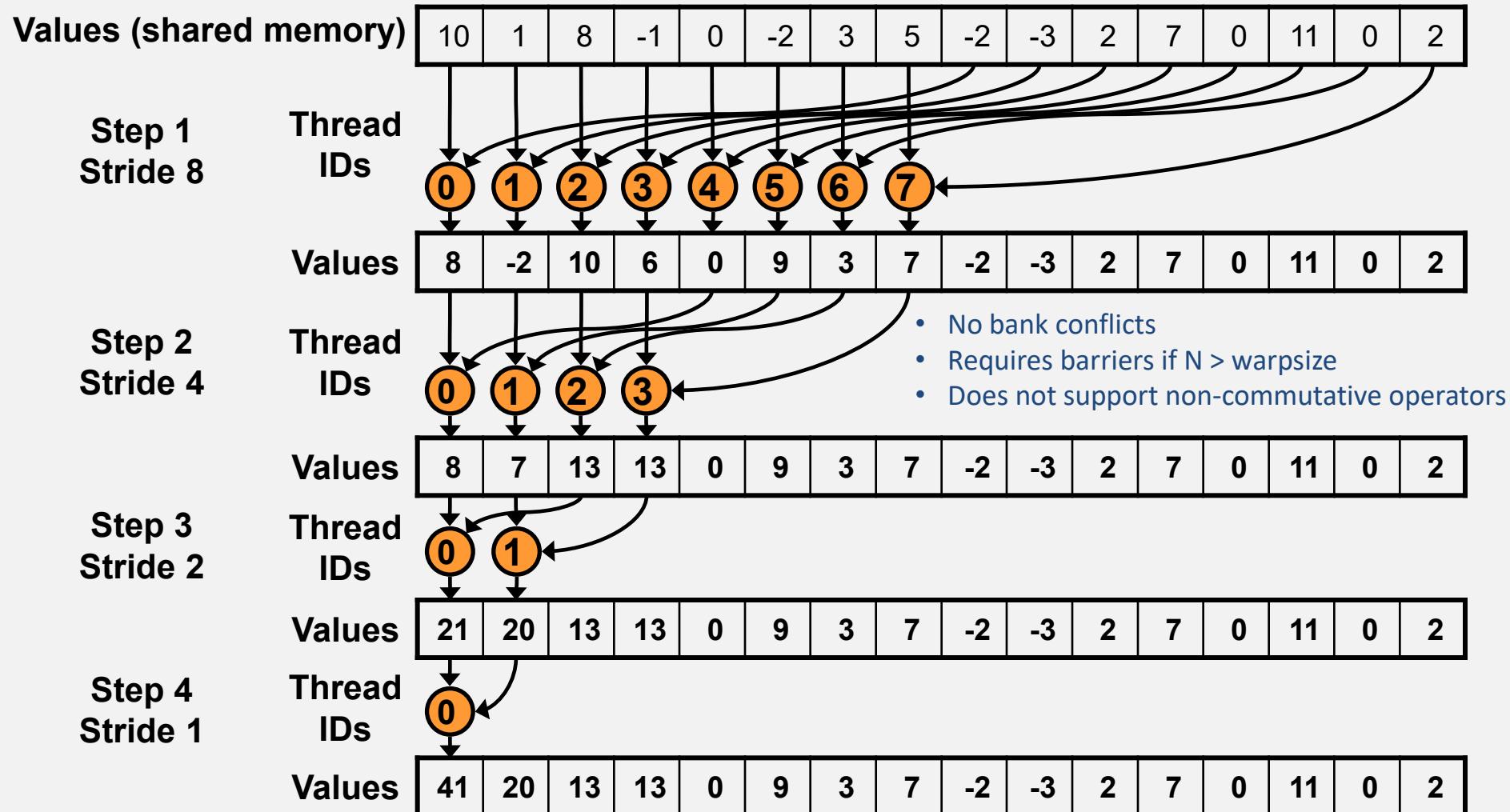
- Arbitrarily bad bank conflicts
- Requires barriers if  $N >$  warpsize
- Supports non-commutative operators



Interleaved addressing results in bank conflicts



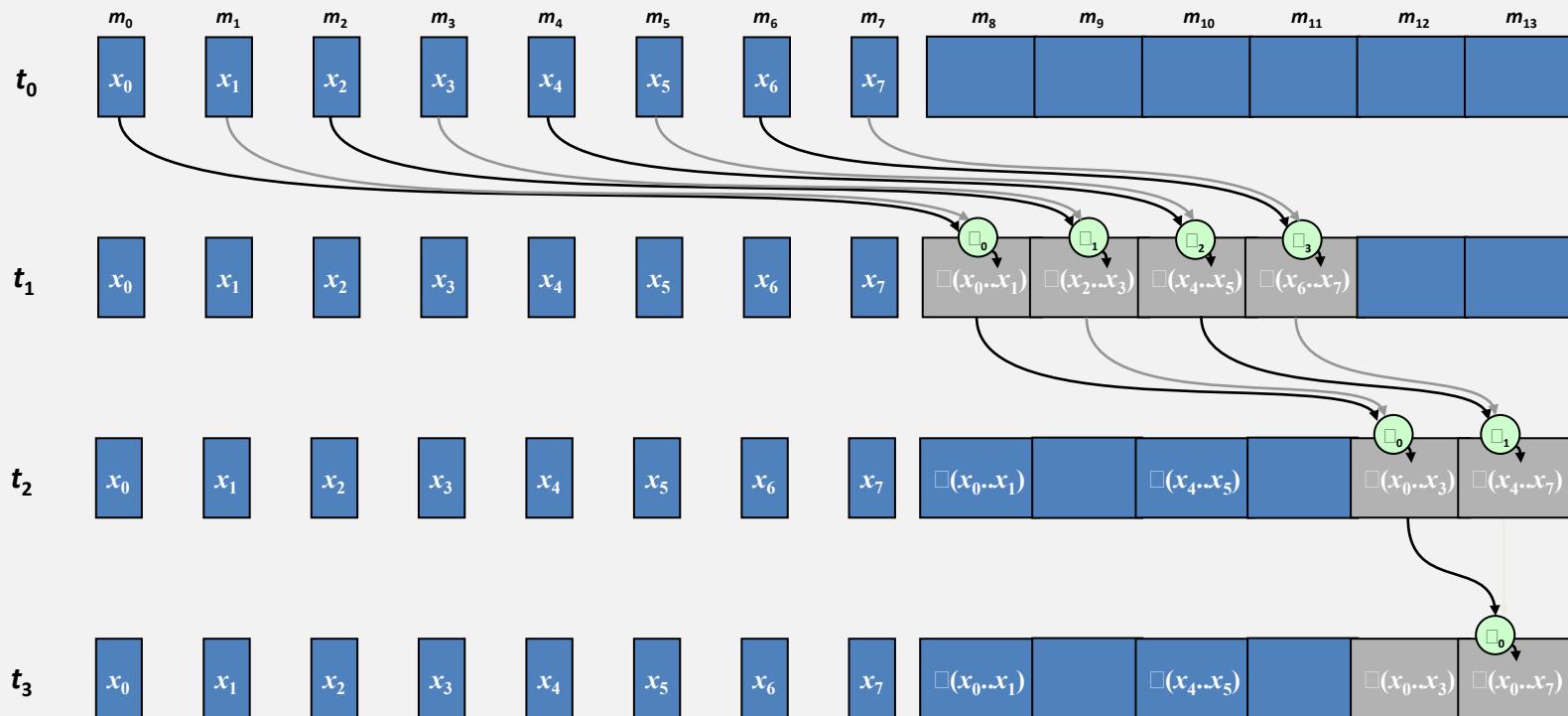
# Parallel Reduction: Sequential Addressing



**Sequential addressing is conflict free**

# 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

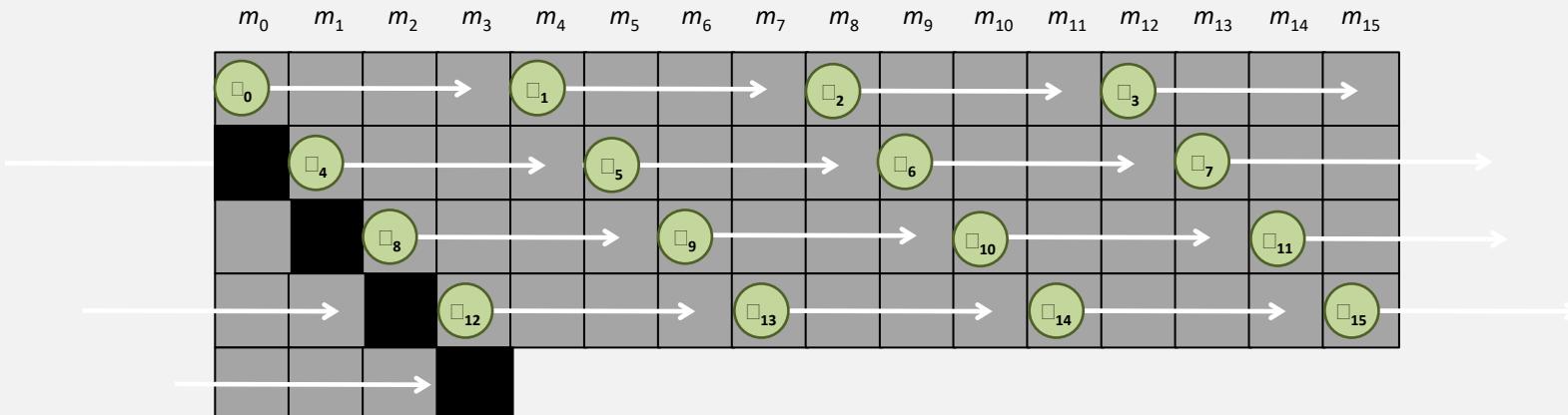
# Persistent Threads

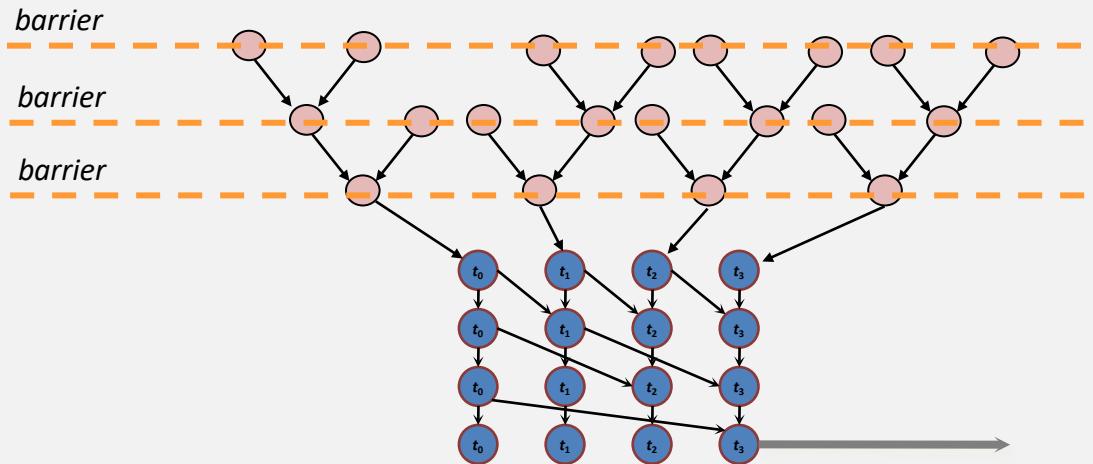
- GPU programming model suggests one thread per item
- What if you filled the machine with just enough threads to keep all processors busy, then asked each thread to stay alive until the input was complete?
- Minus: More overhead per thread (register pressure)
- Minus: Violent anger of vendors

# (Serial) Raking Reduction Phase

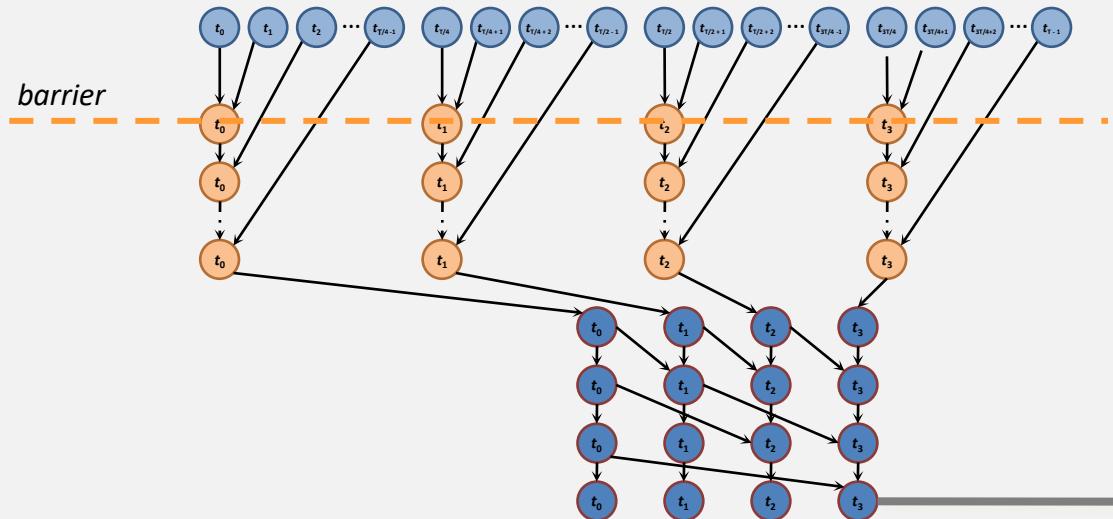
- No bank conflicts, only one barrier to after insertion into smem
- Supports non-commutative operators
- Requires subsequent warp scan to reduce accumulated partials

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

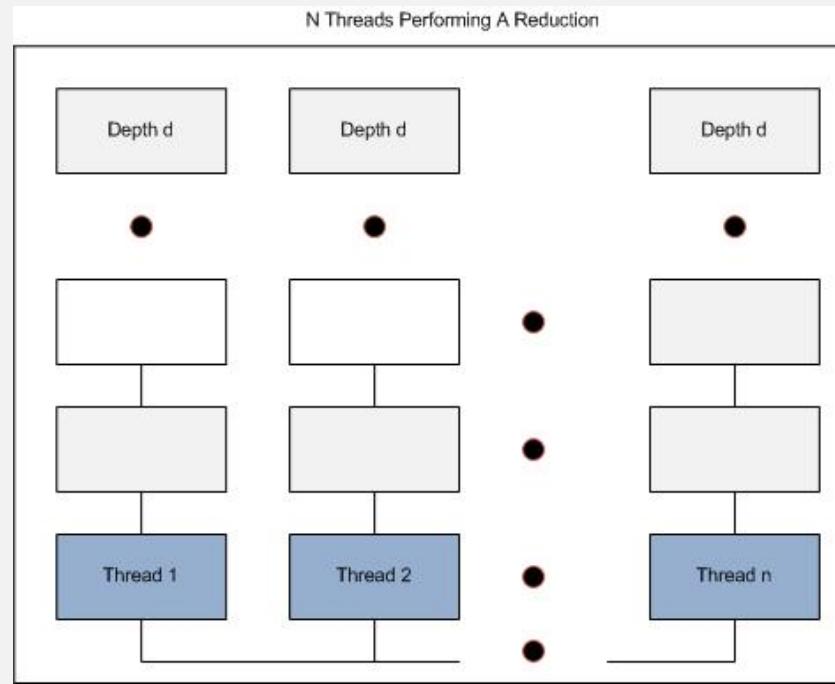




VS.



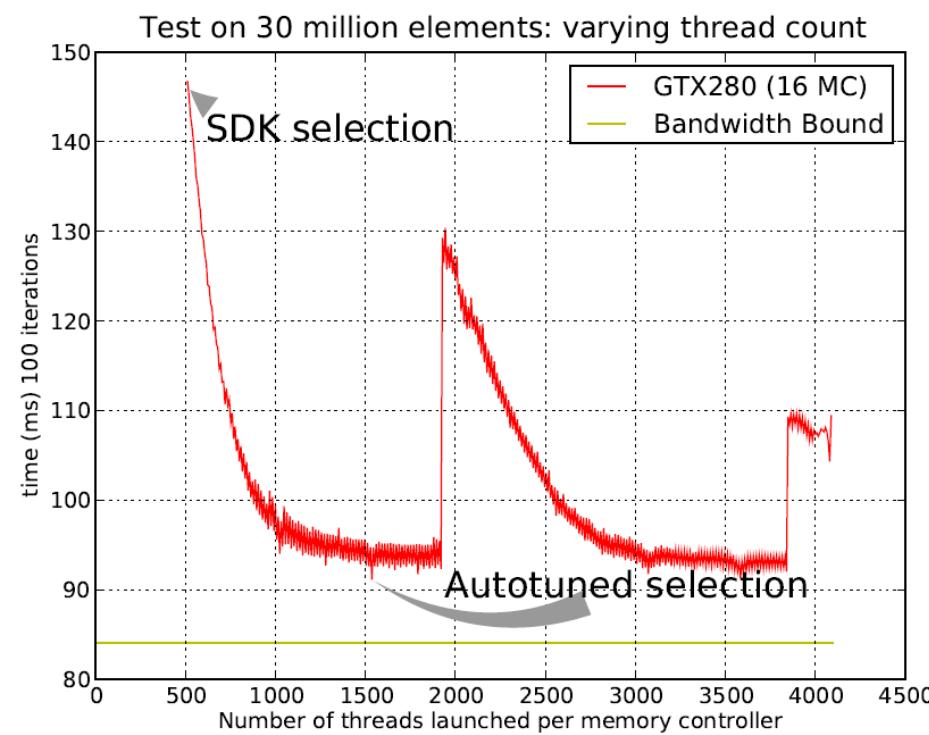
# Reduction



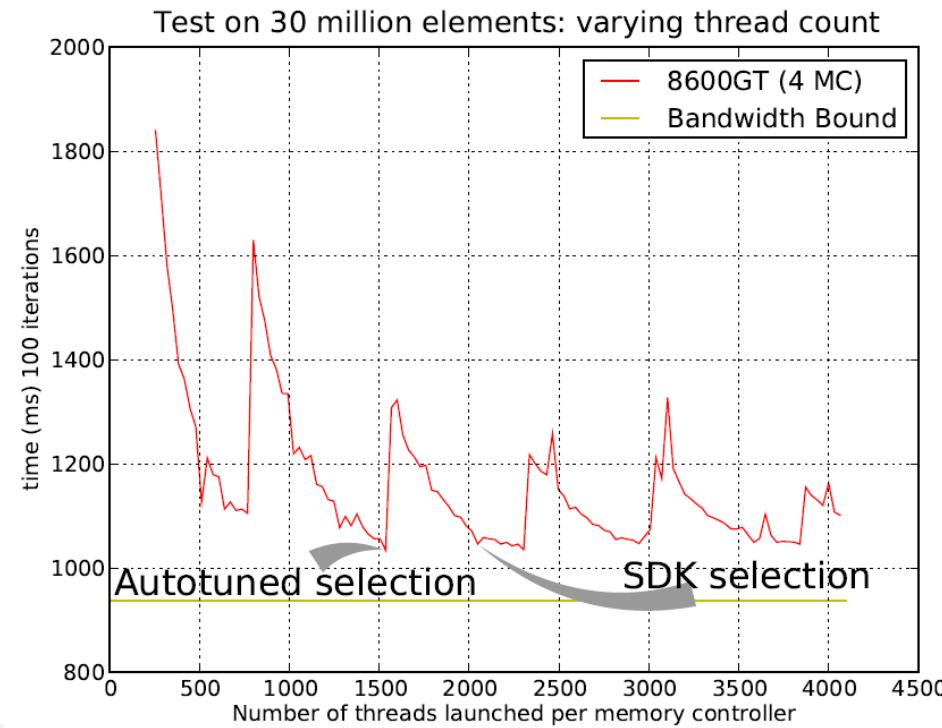
- Many-To-One
  - Parameter to Tune => **Thread Width** (total number of threads)

# Parameter Selection Comparison

GTX280

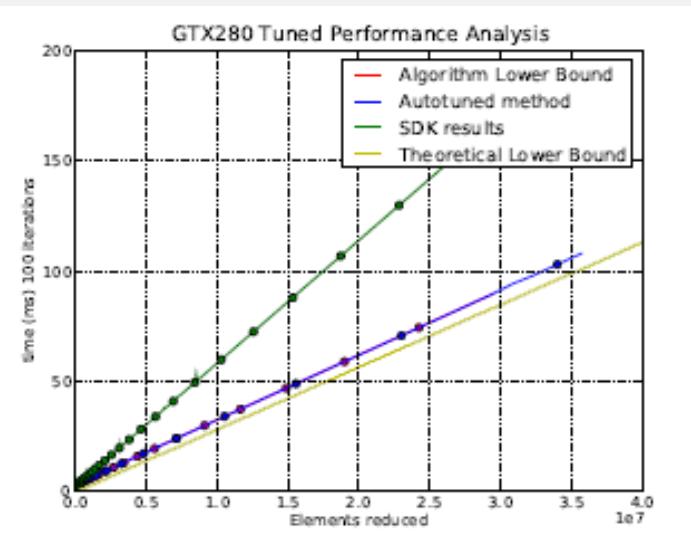
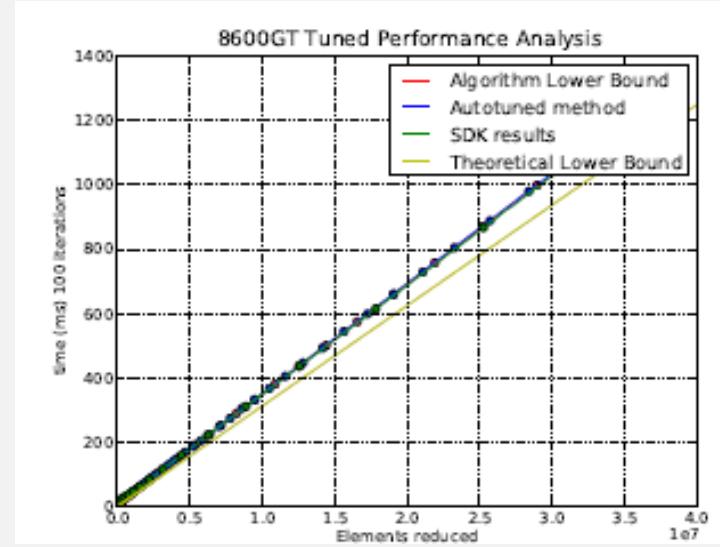
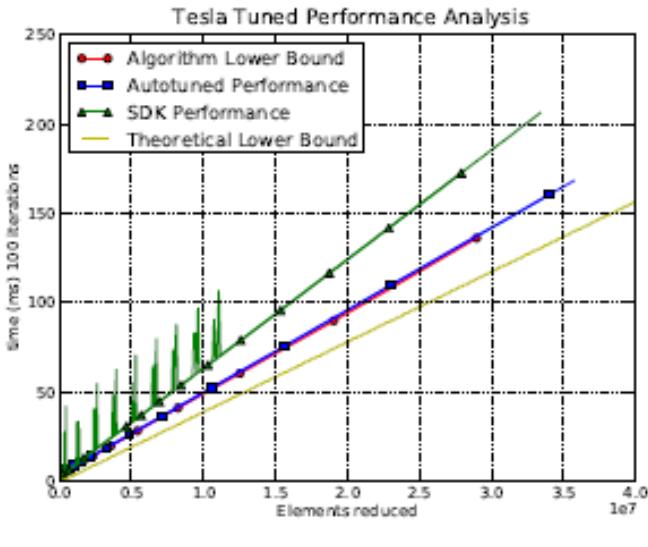


8600GT



Parameter selection comparison between the static SDK and our tuned (thread cap) algorithm

We see some of the problems with having static thread parameters, for different machines.



- Auto-tuned performance always exceeded SDK performance
  - Up to a 70% performance gain for certain cards and workloads

# 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)

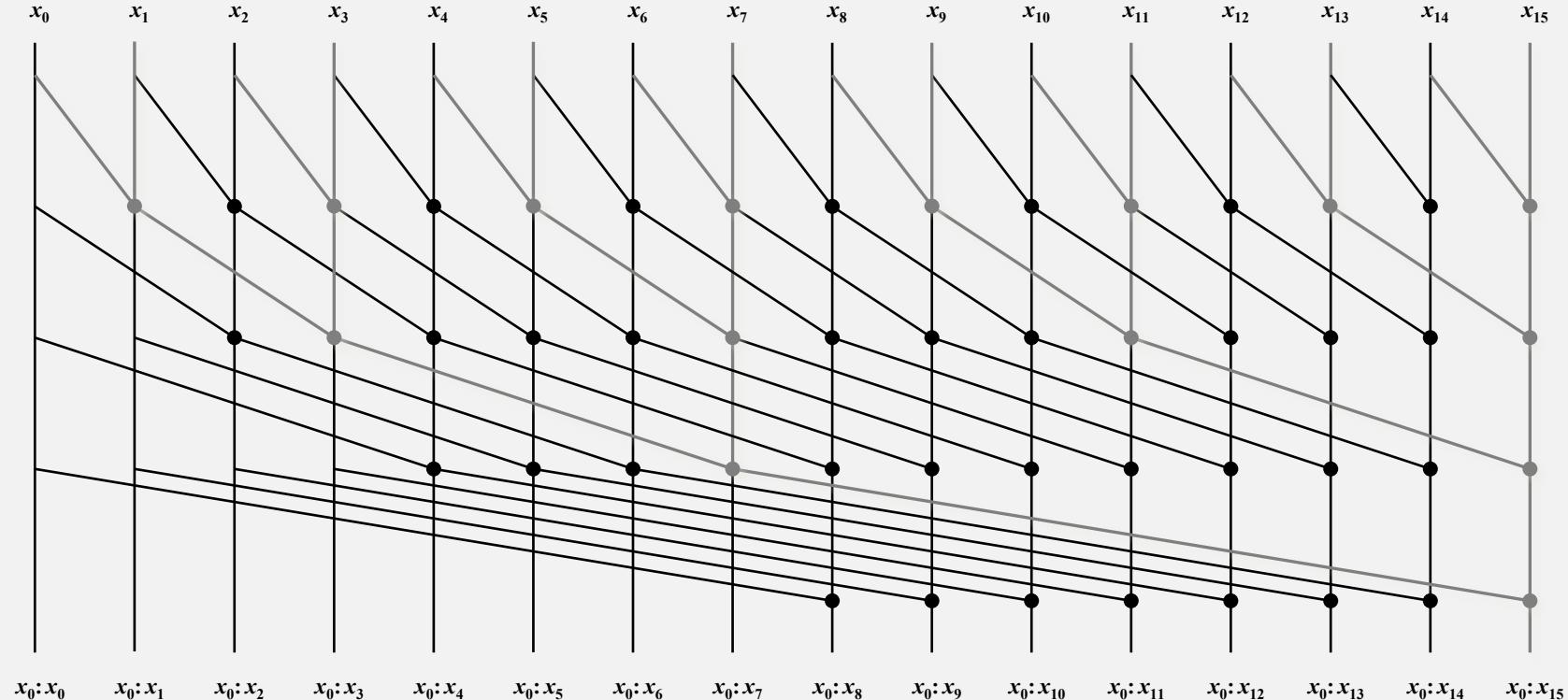
# Scan (within a block)

# Parallel Prefix Sum (Scan)

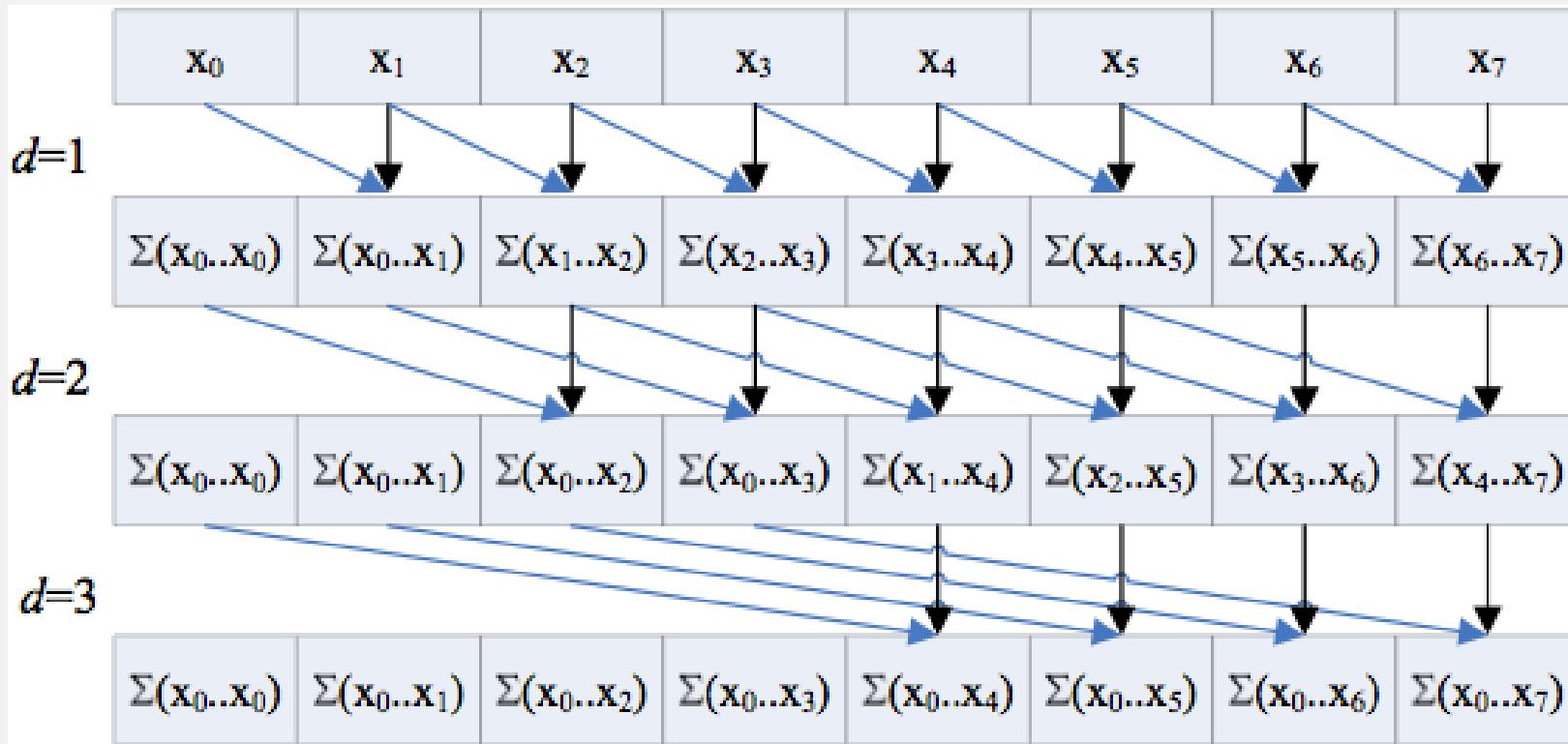
- Given an array  $A = [a_0, a_1, \dots, a_{n-1}]$  and a binary associative operator  $\oplus$  with identity  $I$ ,
- $\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})]$
- Example: if  $\oplus$  is addition, then scan on the set
  - $[3 1 7 0 4 1 6 3]$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

# Kogge-Stone Scan

Circuit family



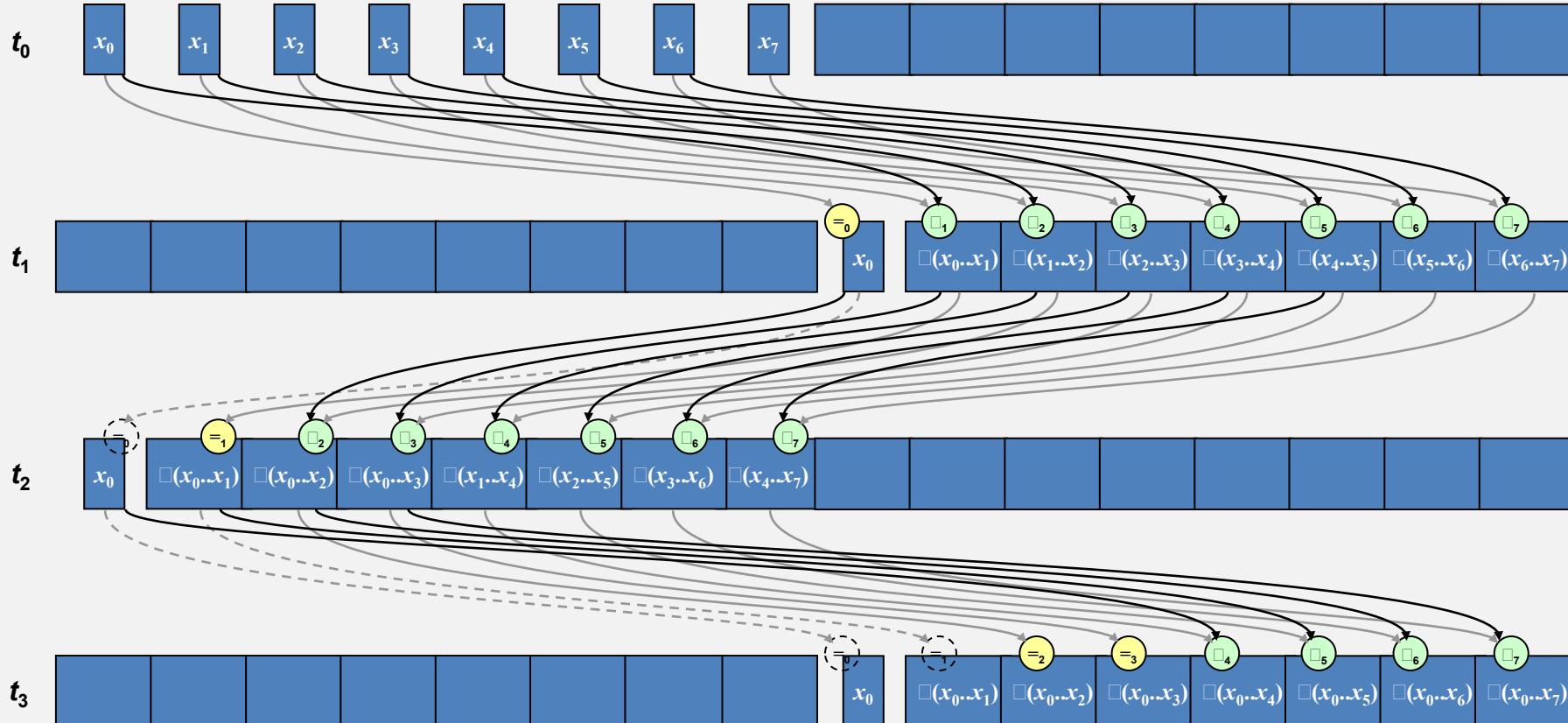
# $O(n \log n)$ Scan



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

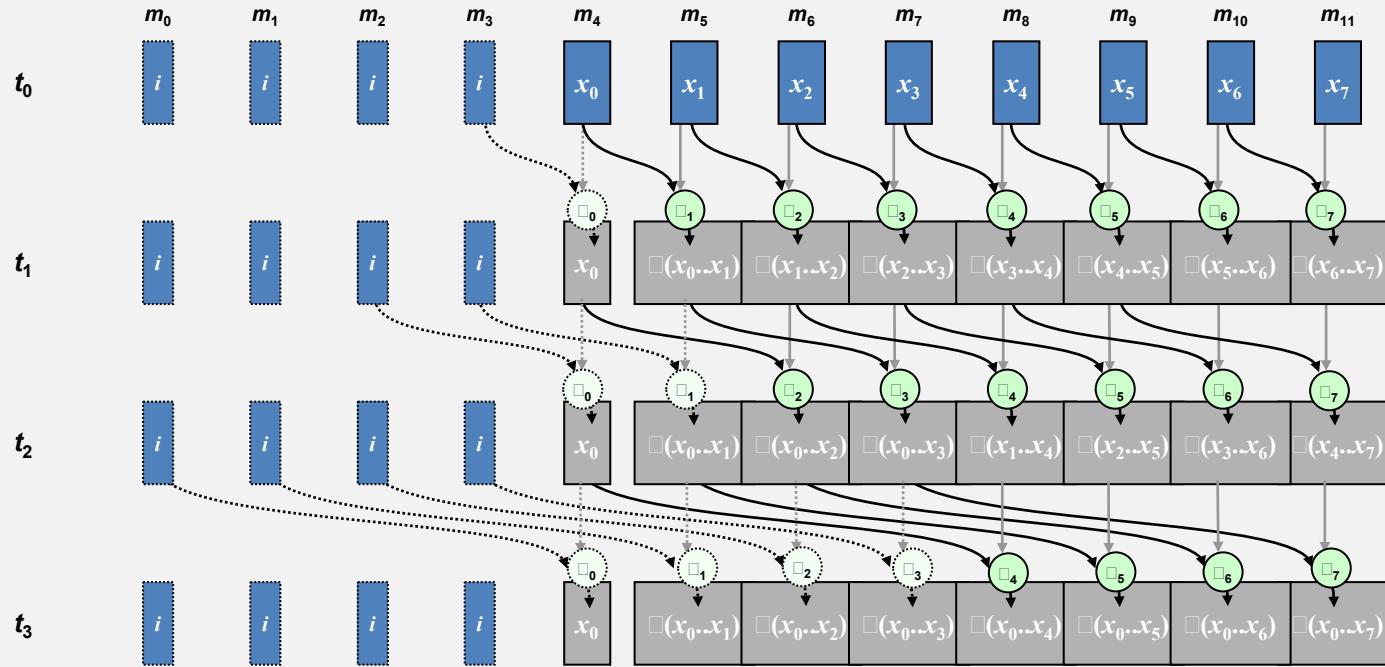
# Alt. Hillis-Steele Scan Implementation

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



# Alt. Hillis-Steele Scan

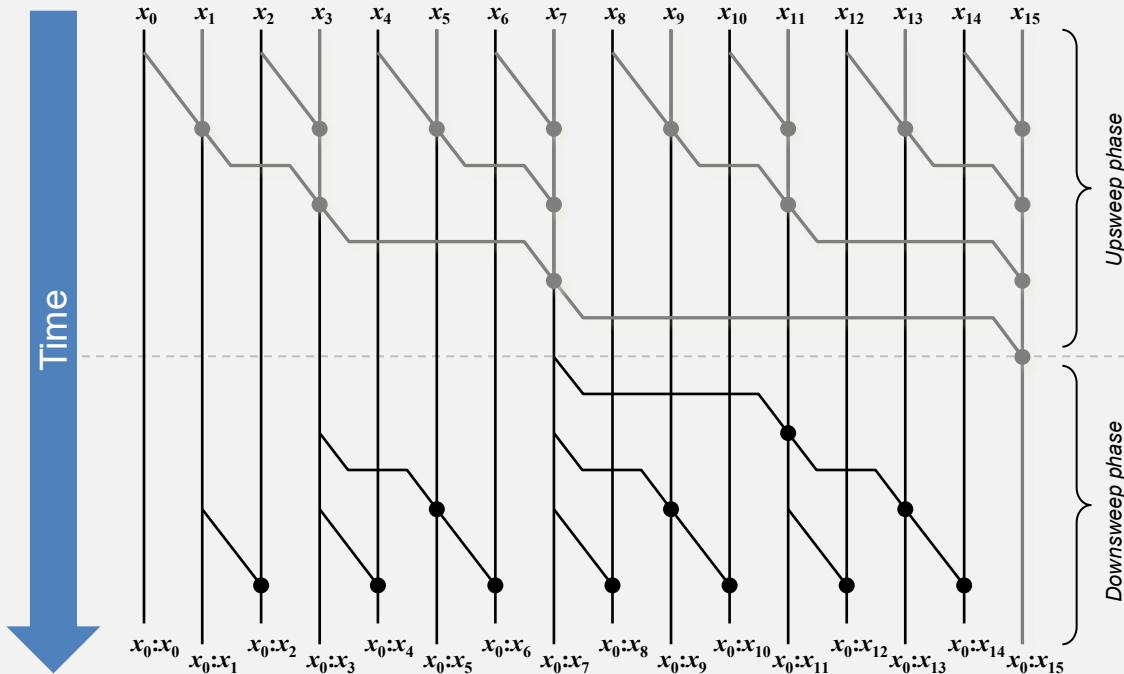
Warp-synchronous: SIMD without divergence or barriers



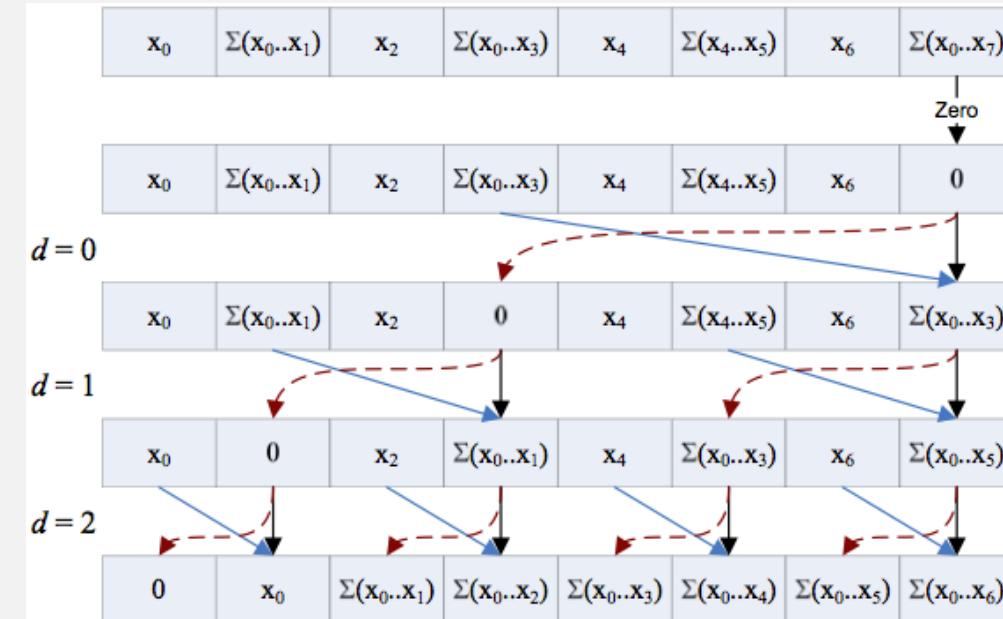
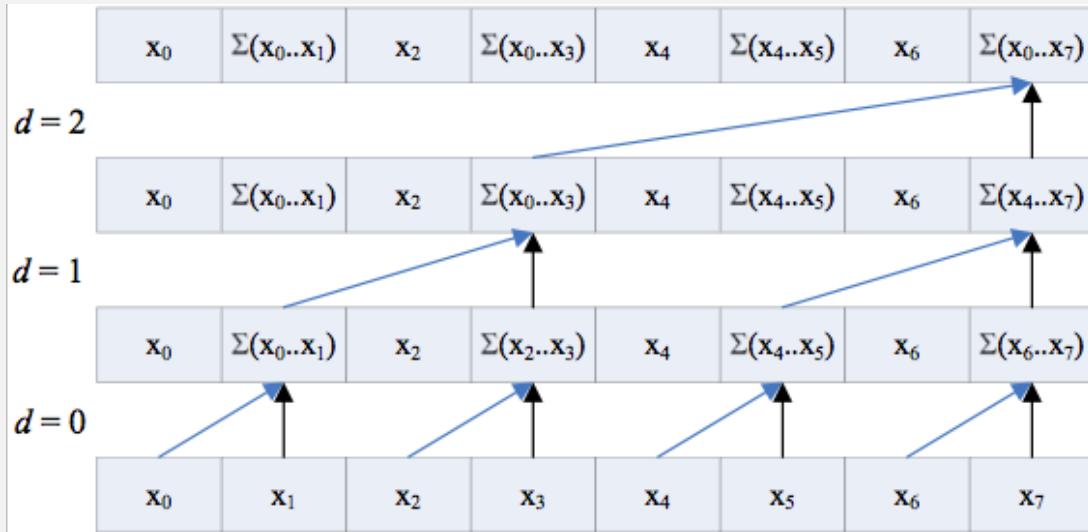
- What if we truly had a SIMD machine?
- Recall CUDA warps (32 threads) are strictly SIMD
- “Warp-synchronous”

# Brent Kung Scan

Circuit family

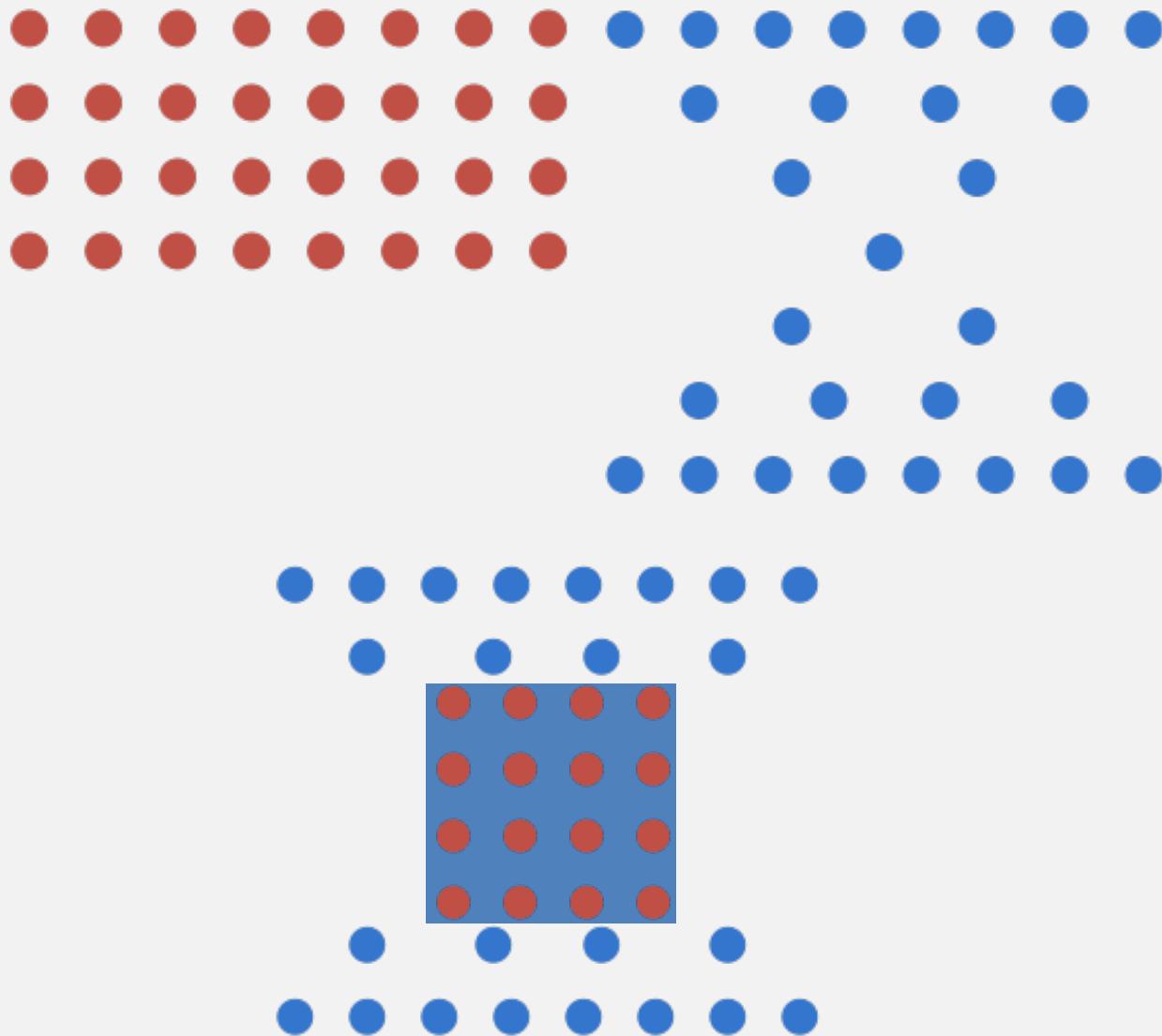


# $O(n)$ Scan [Blelloch]



- Not step efficient ( $2 \log n$  steps)
- Work efficient ( $O(n)$  work)
- Bank conflicts, and lots of 'em

# Hybrid methods

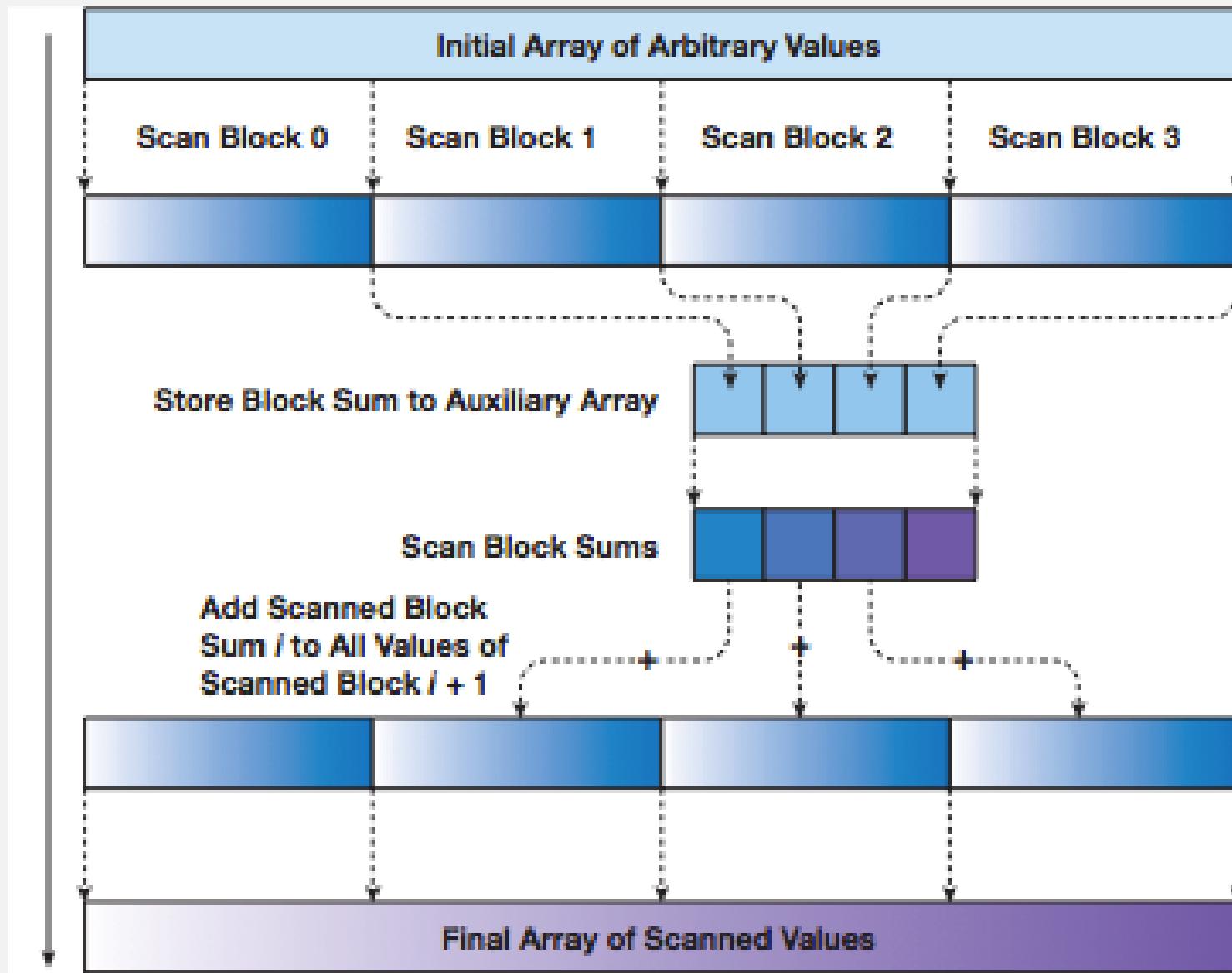


# Scan papers

- 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.

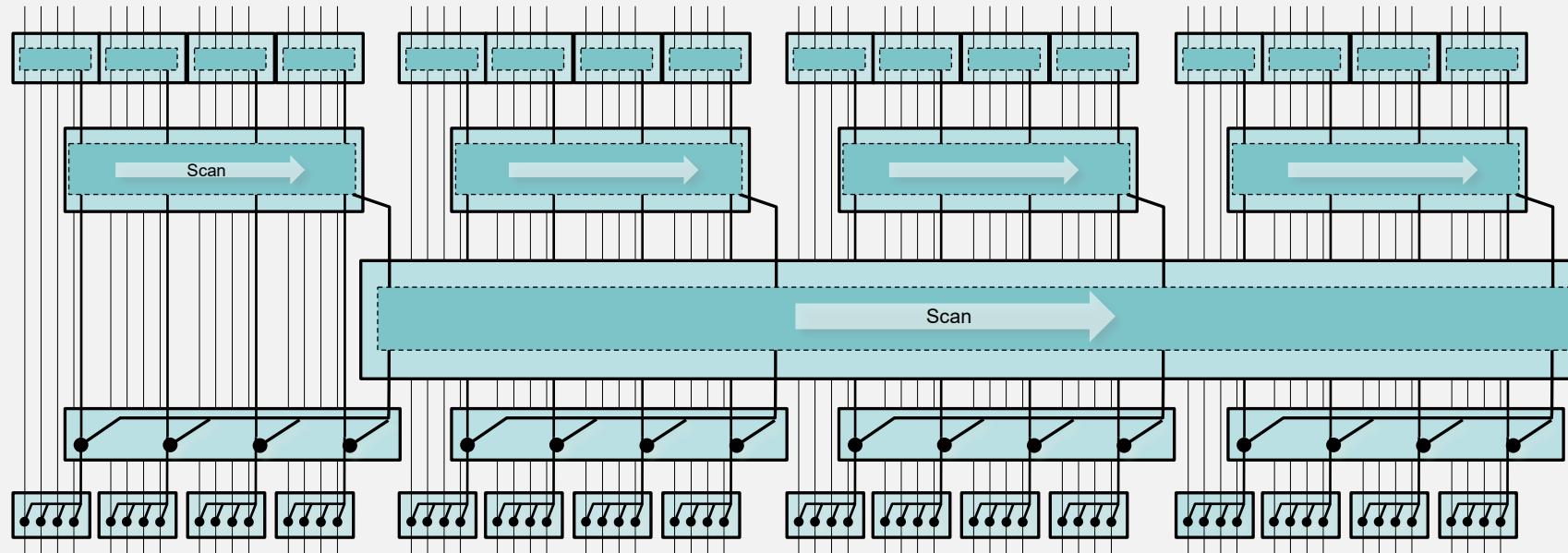
# Scan (across blocks)

# Scan-then-propagate (4x)



# Scan-then-propagate (4x)

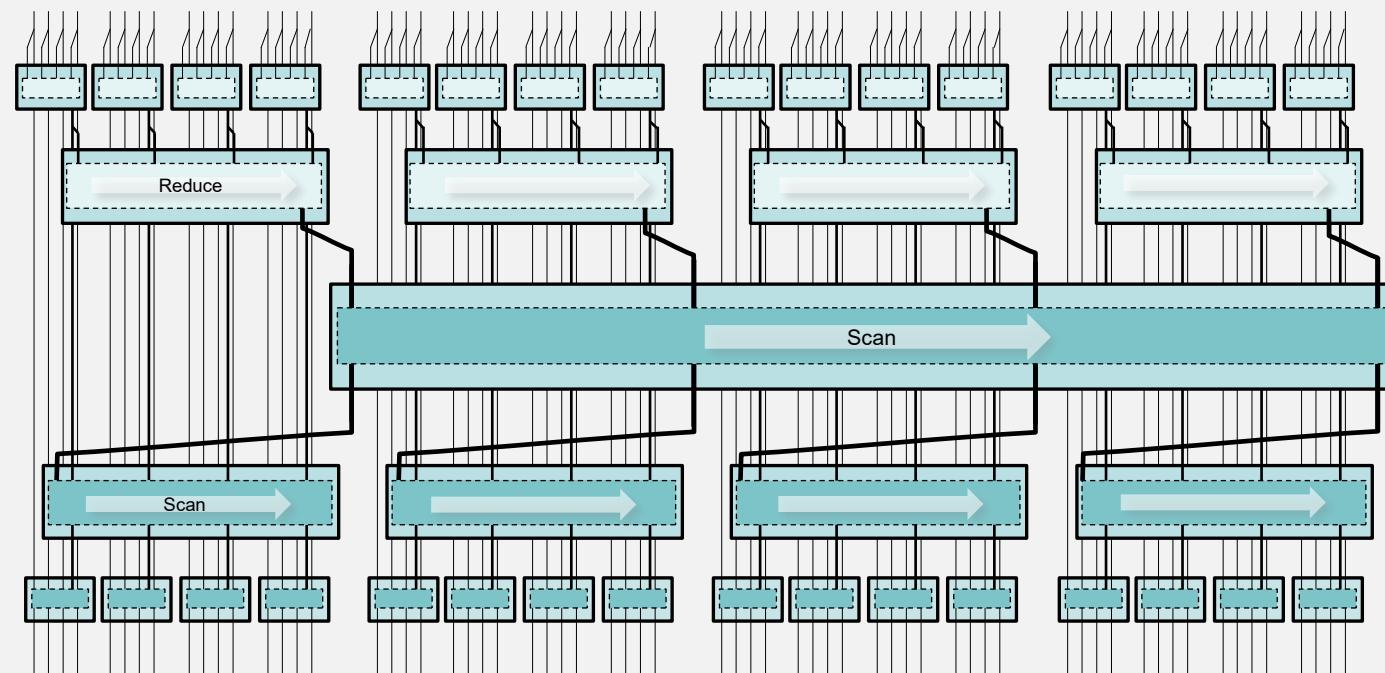
$\log_b(N)$  –level upsweep/downsweep  
(scan-and-add strategy: CudPP)



# Reduce-then-scan (3x)

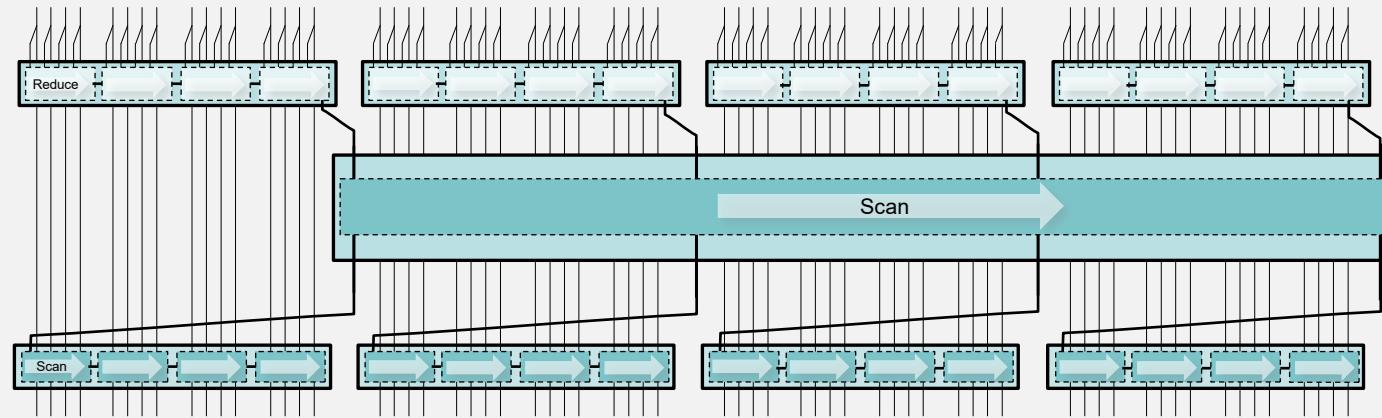
$\log_b(N)$  –level upsweep/downsweep

(reduce-then-scan strategy: Matrix-scan)



# Merrill's 2-level upsweep/downsweep

(reduce-then-scan)



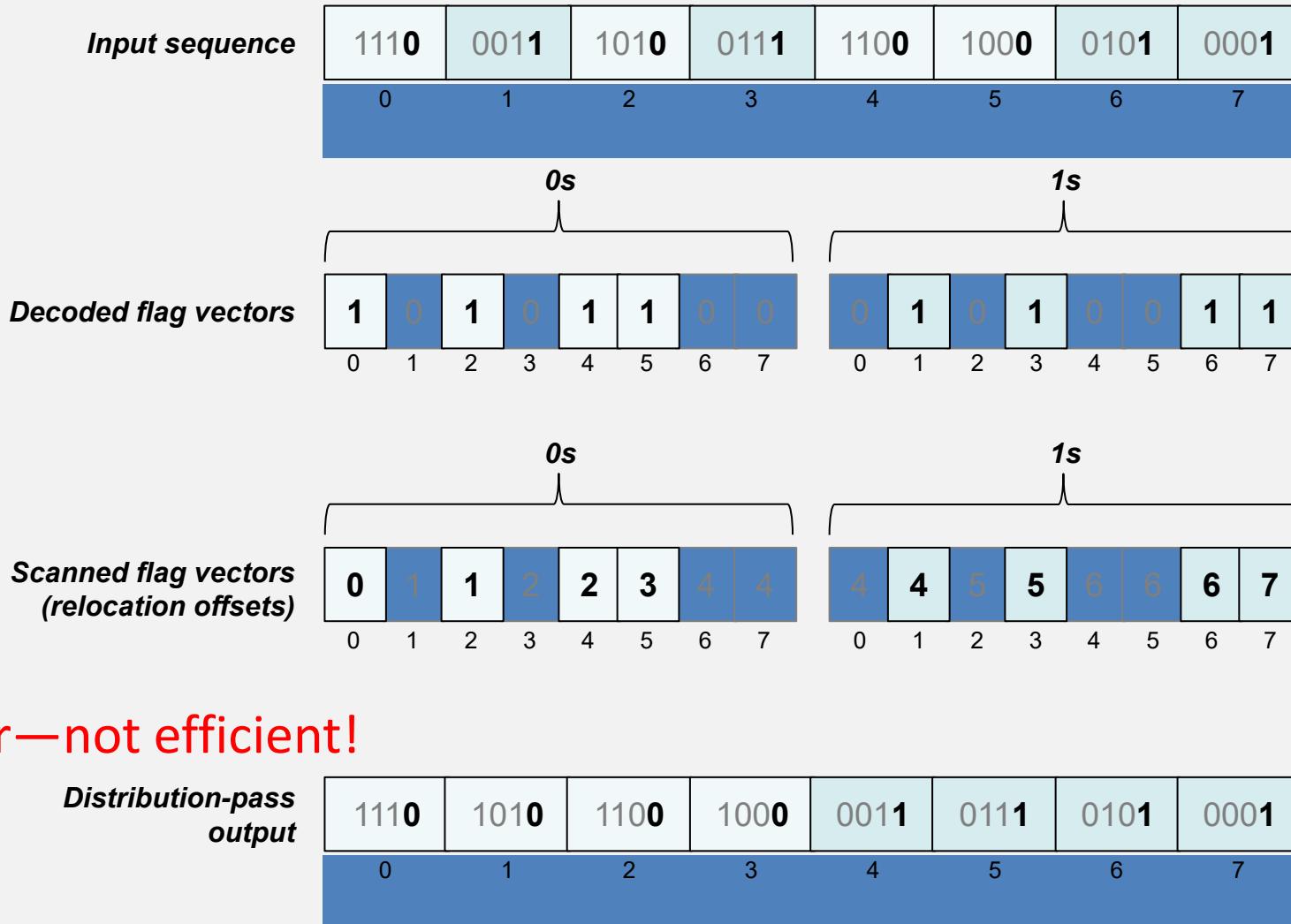
- Persistent threads
- Requires only 3 kernel launches vs.  $\log n$
- Fewer global memory reads in intermediate step  
(constant vs.  $O(n)$ )

# Sort papers

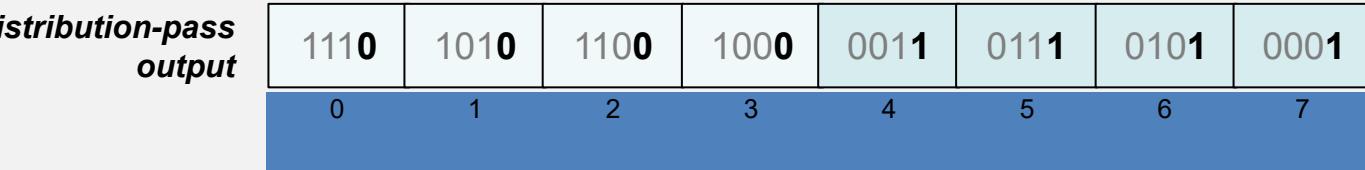
- Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007.
- N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore GPUs,” IPDPS 2009: IEEE 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

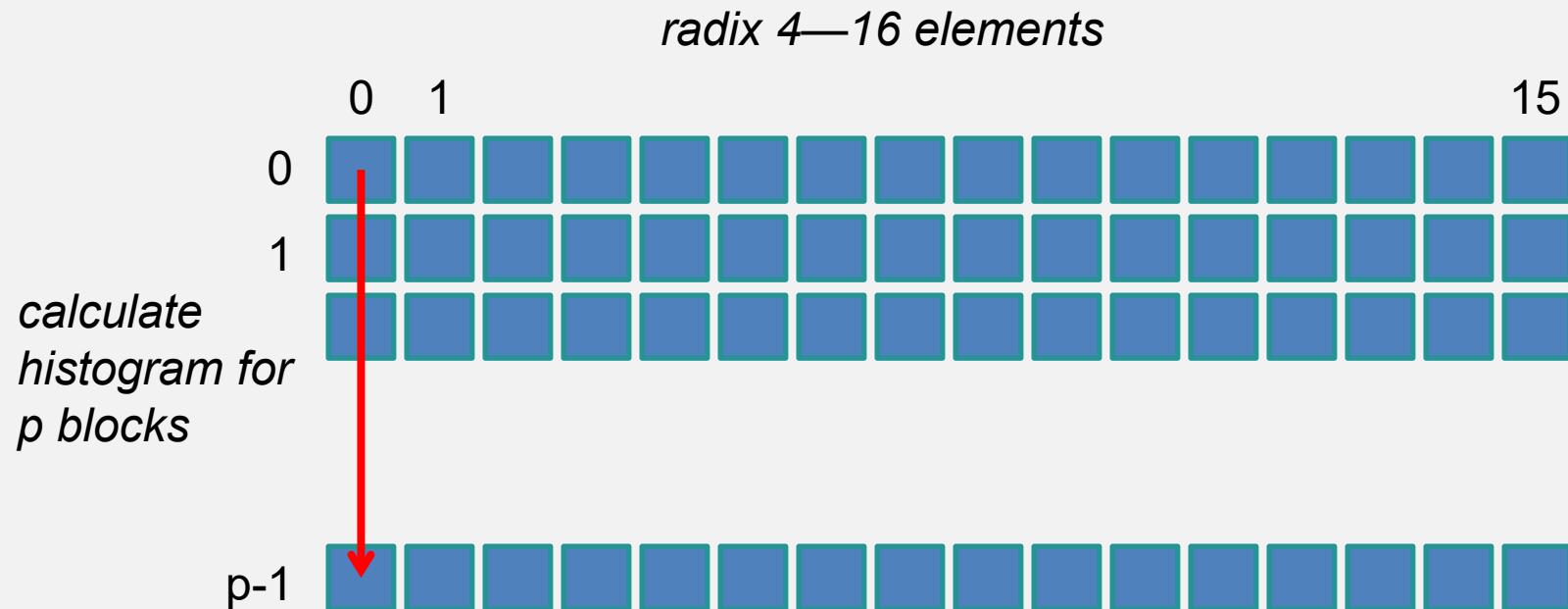
- Assign tile of data to each block (1024 elements)  
*Satish uses 256-thread blocks and 4 elements per thread*
- Build per-block histograms of current digit (4 bit)  
*this is a reduction*
- Combine per-block histograms (P x 16)  
*this is a scan*
- Scatter

# 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



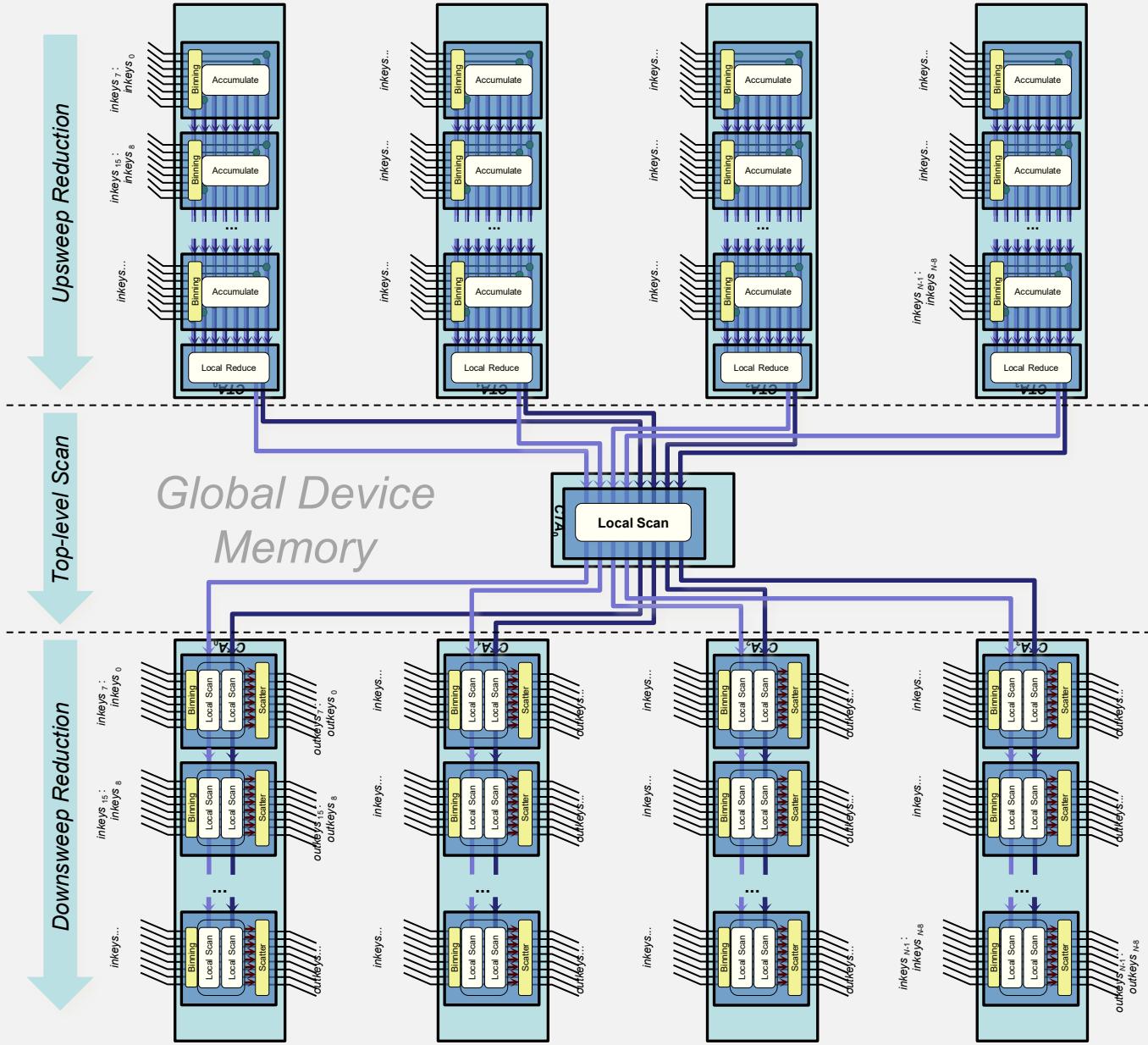
# Satish's Radix Sort Memory Cost

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



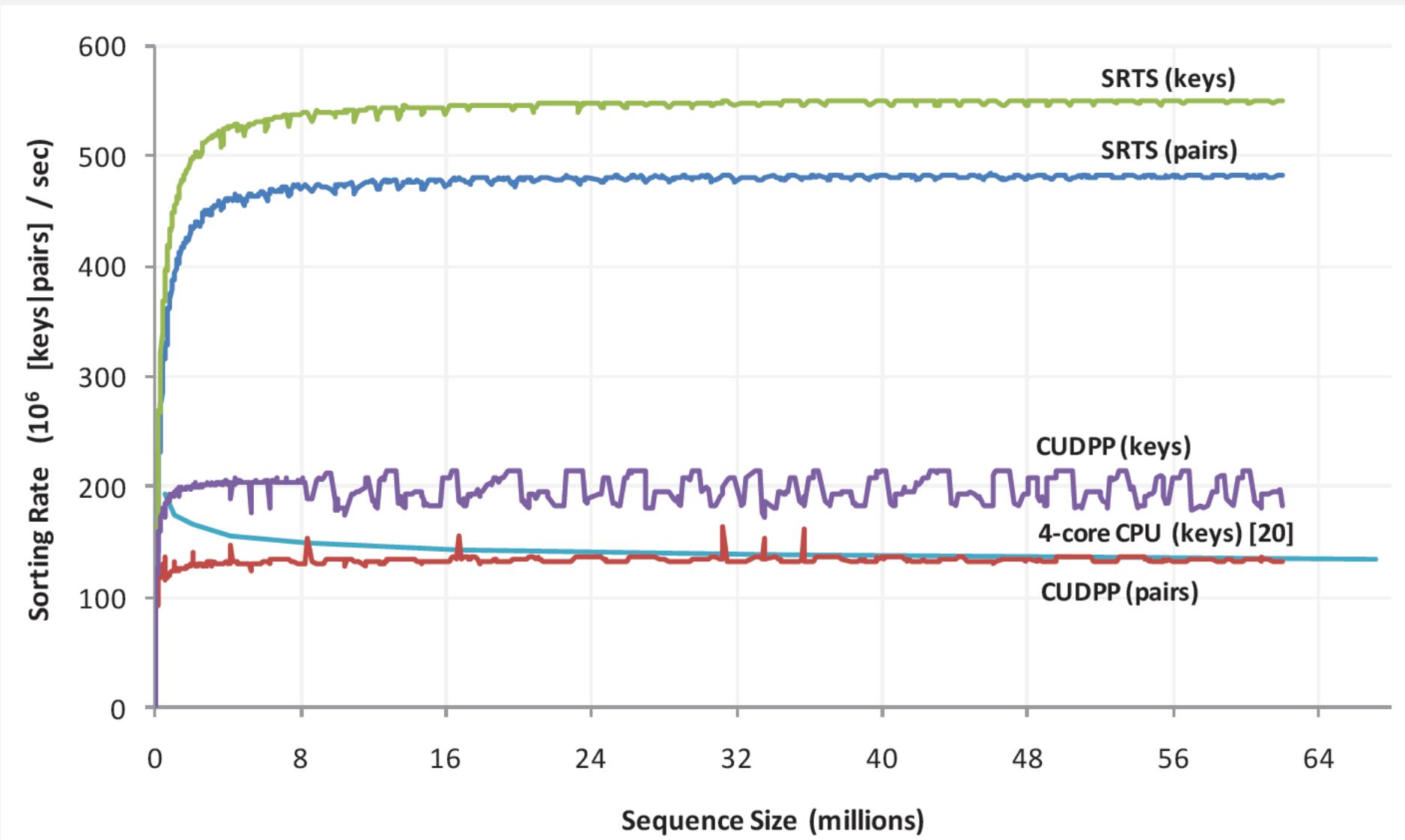
# Merrill's sort, costs

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)

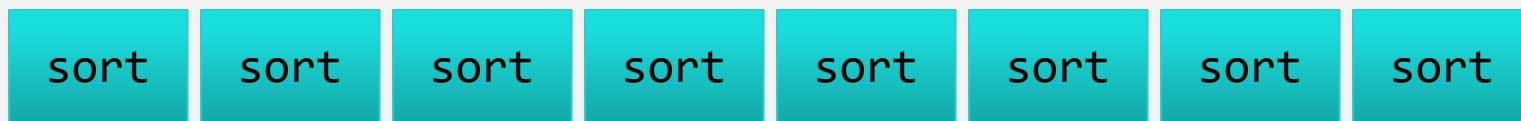
# Results (NVIDIA GTX 285)



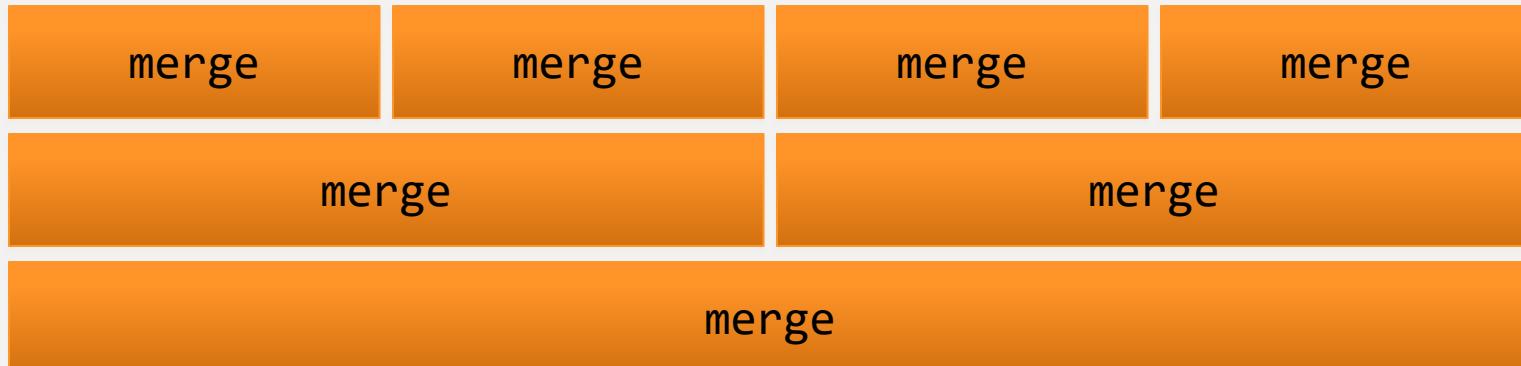
# Merge Sort

# Merge Sort

- Divide input array into 256-element tiles
- Sort each tile independently



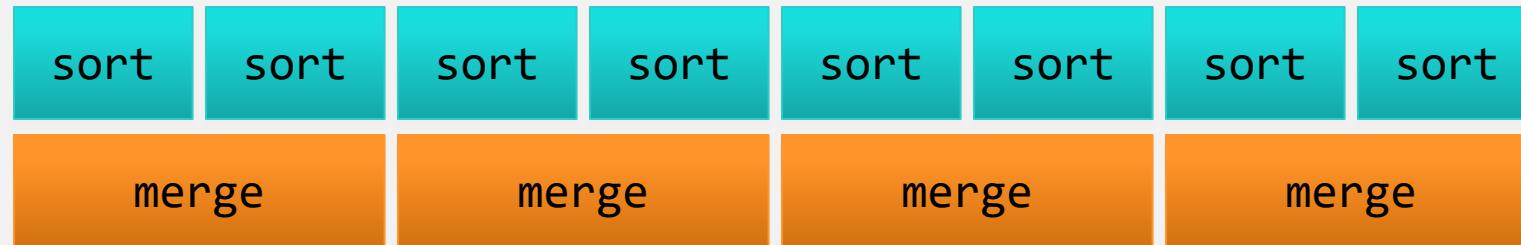
- Produce sorted output with tree of merges



# Sorting a Tile

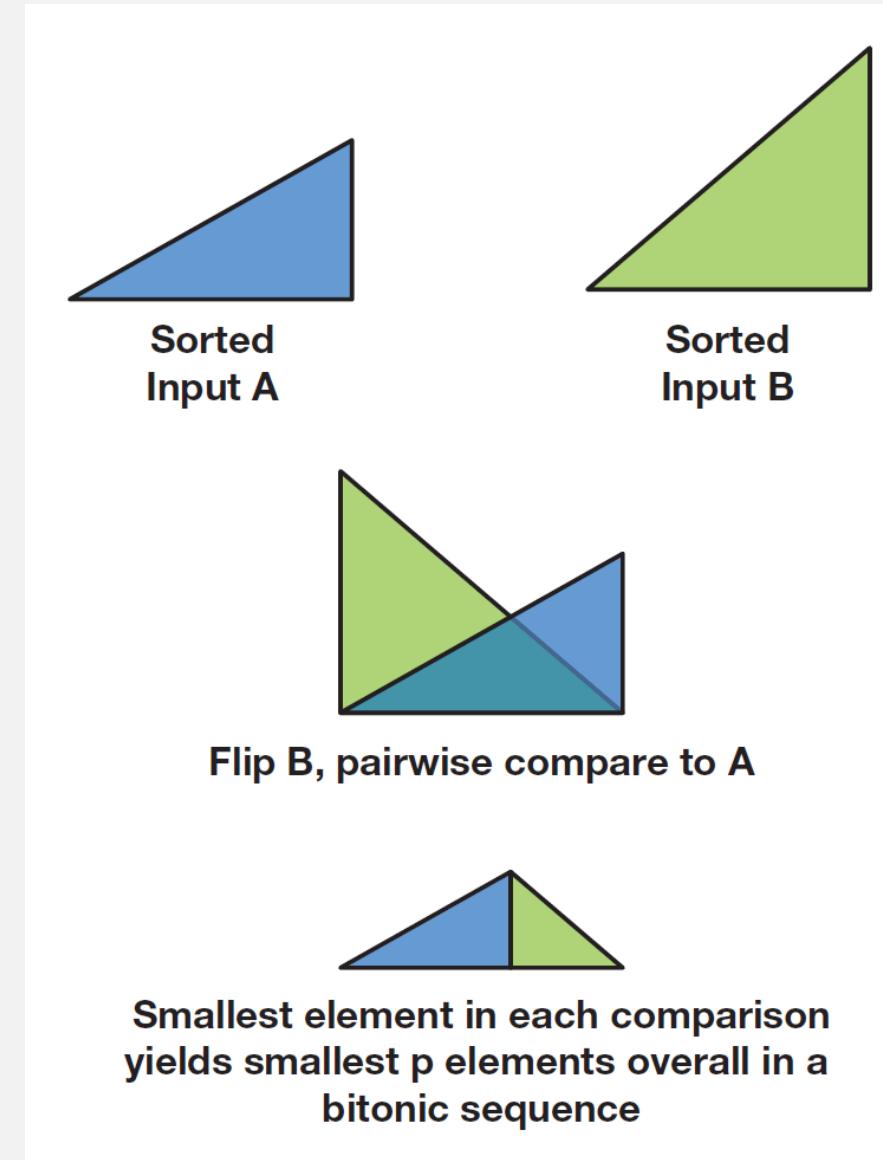
- Tiles are sized so that:
  - a single thread block can sort them efficiently
  - they fit comfortably in on-chip memory
- Sorting networks are most efficient in this regime
  - we use **odd-even merge sort**
  - about 5-10% faster than comparable bitonic sort
- Caveat: sorting networks may reorder equal keys

# Merging Pairs of Sorted Tiles



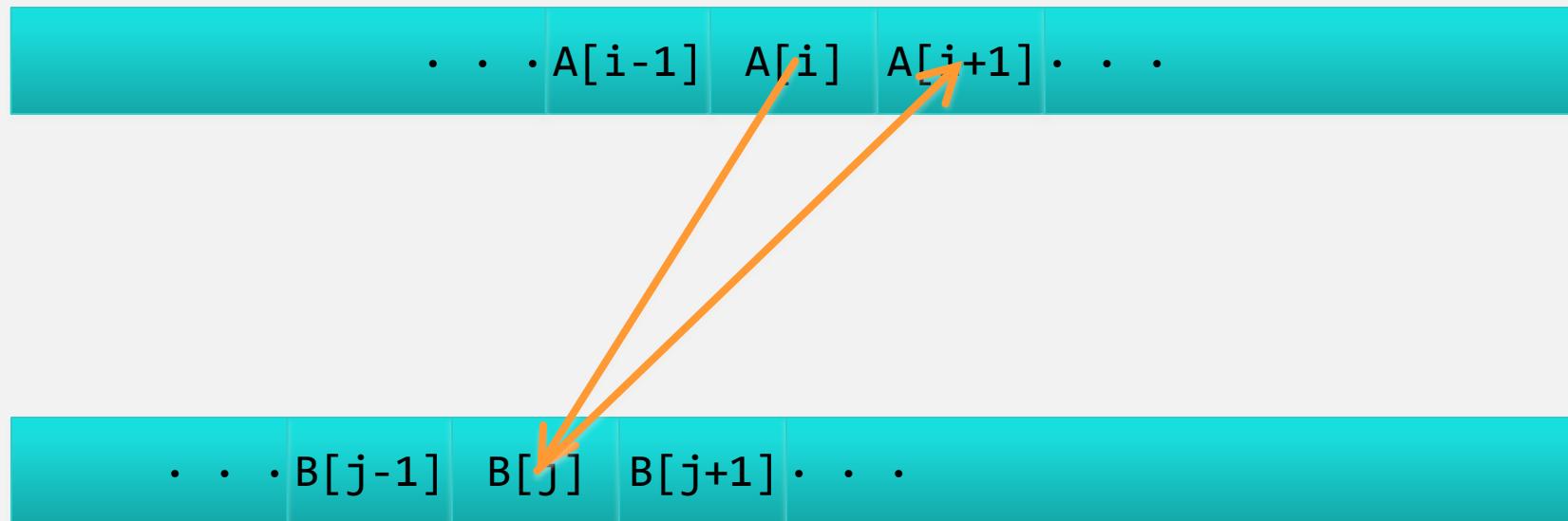
- Launch 1 thread block to process each pair of tiles
- Load tiles into on-chip memory
- Perform **counting merge**
- Store merged result to global memory

# My grad-student-days merge



# 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

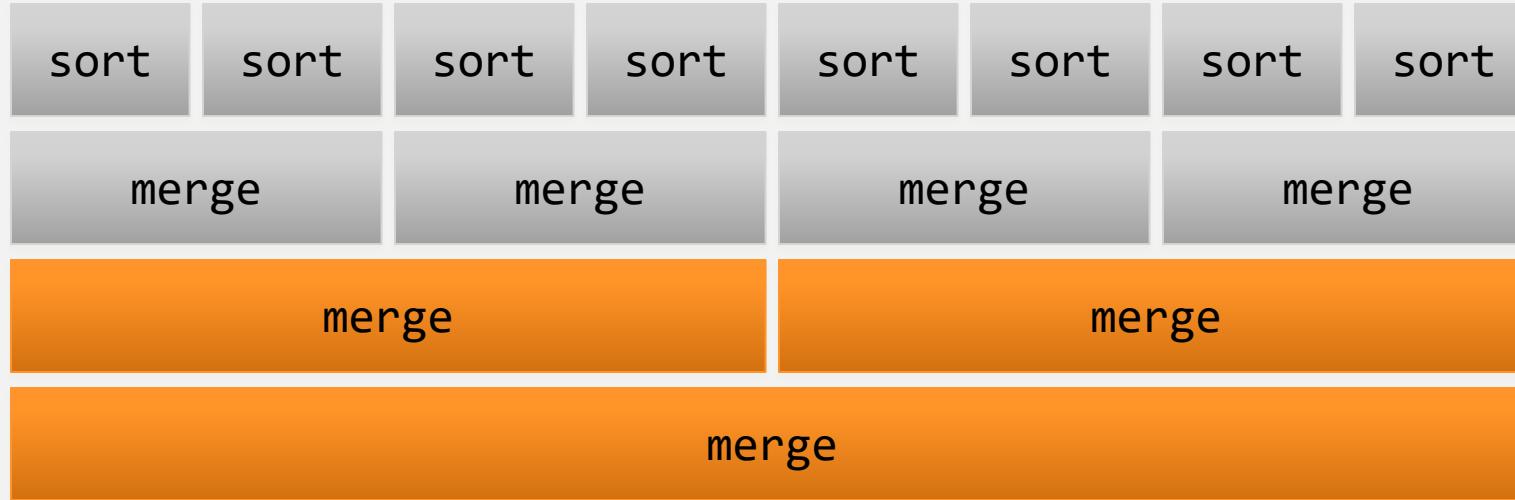
`upper_bound(A[i], B) = count( j where A[i] ≤ B[j] )`



`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

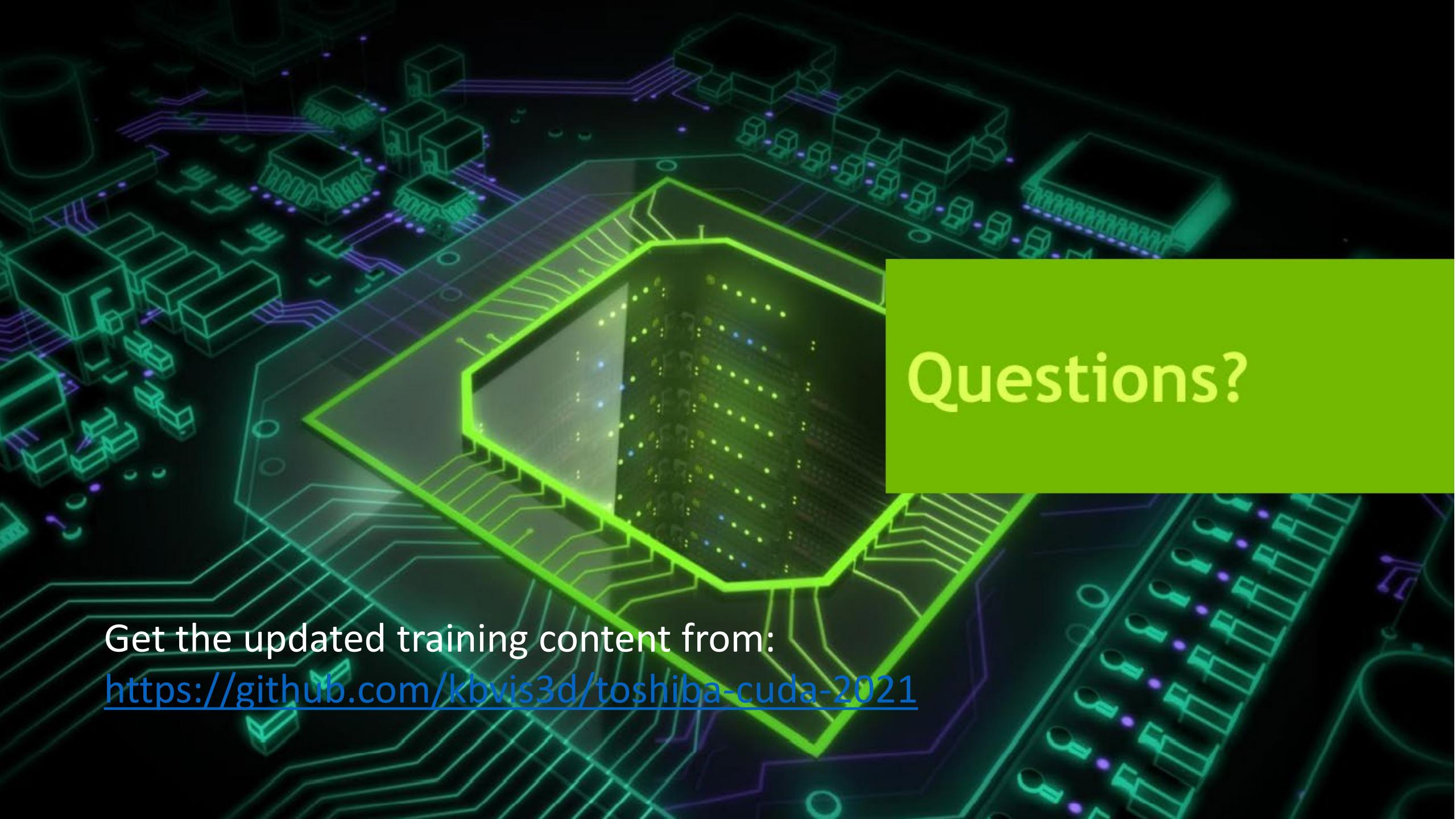


- Apply merge recursively to merge splitter sets
  - recursively apply merge procedure

- Split A & B with merged splitters



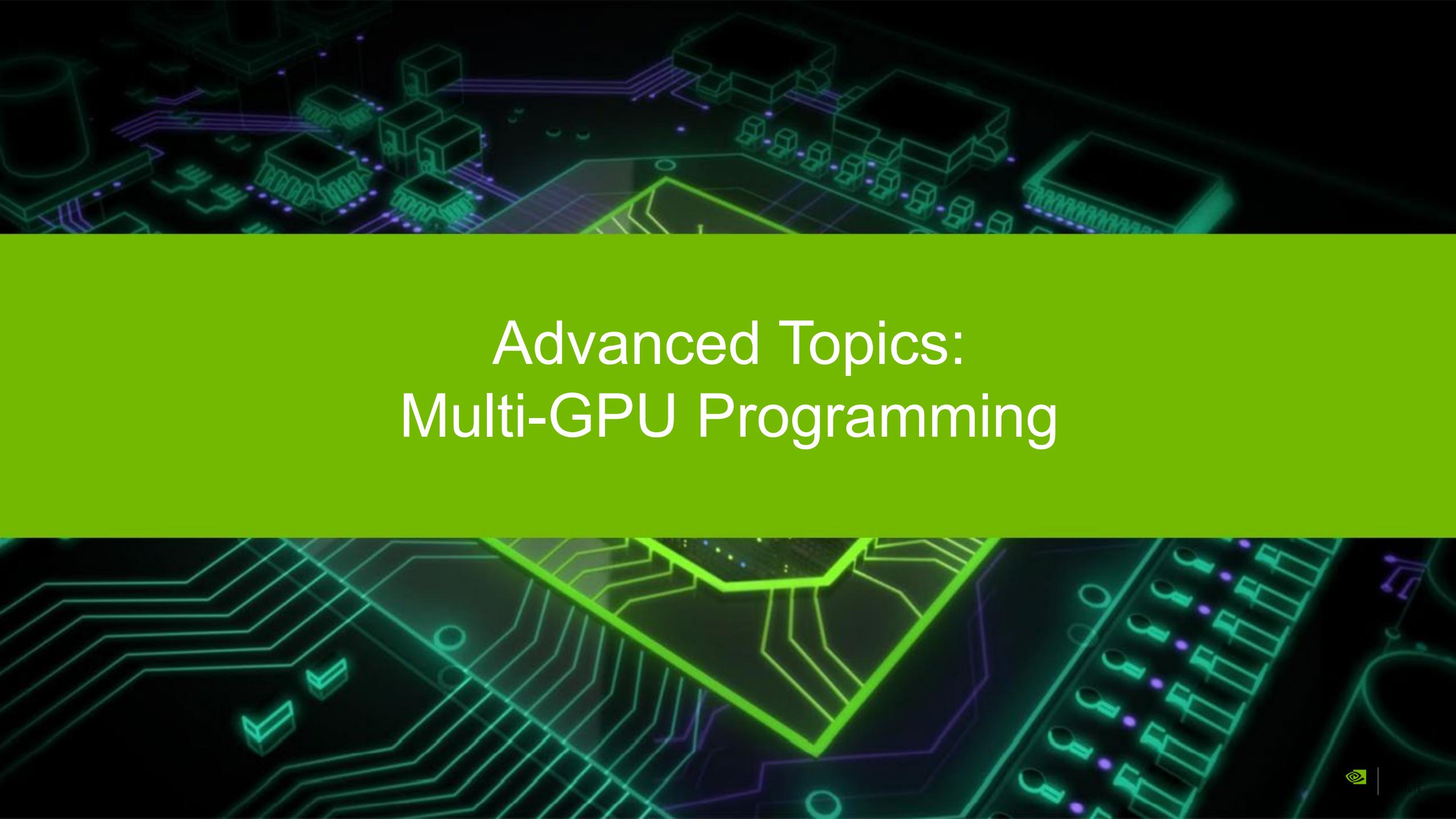
- Merge resulting pairs of tiles (at most 256 elements)



# Questions?

Get the updated training content from:

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



# 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

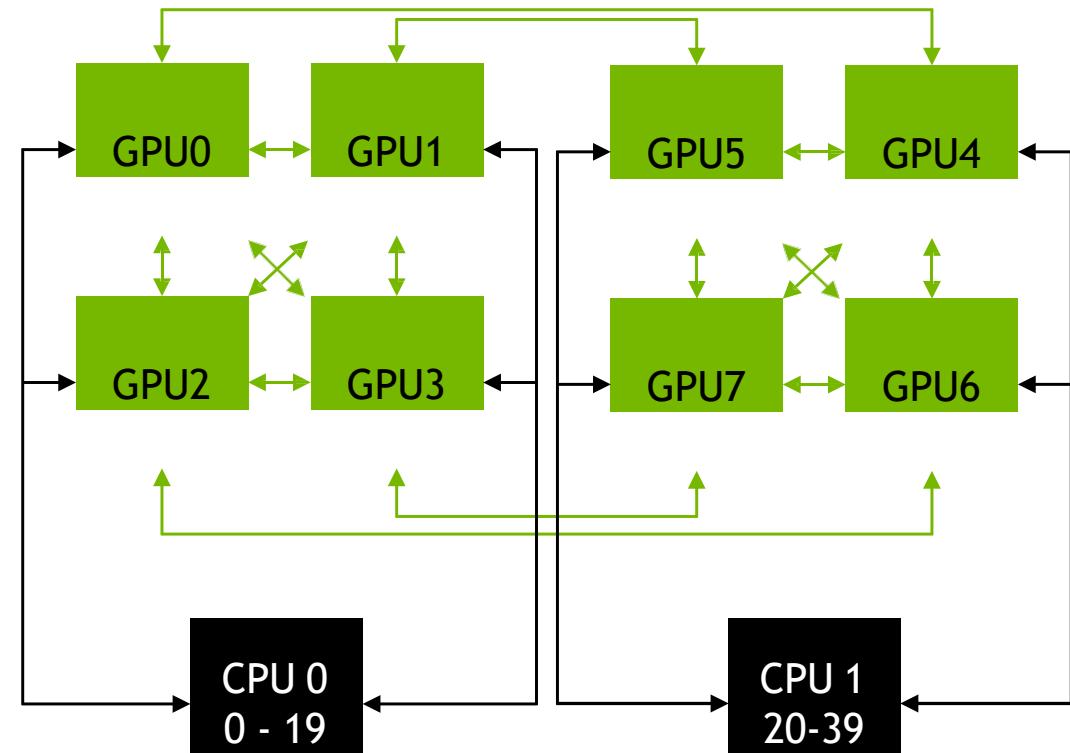
160GB/s per GPU bidirectional to  
Peers

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

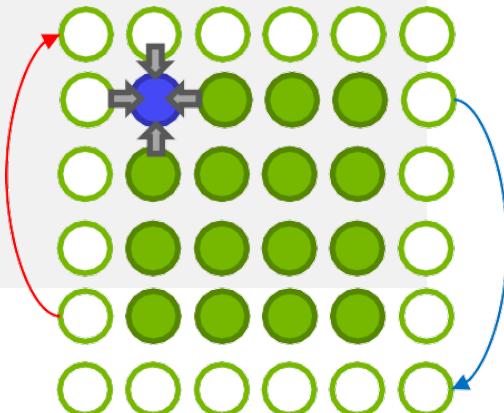
# EXAMPLE: JACOBI SOLVER

## 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 *  
            -t a[(iy-1)*nx+(ix+1)] + a[(iy+1)*nx+ix ] ;  
            *nx+ix-1]  
            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:

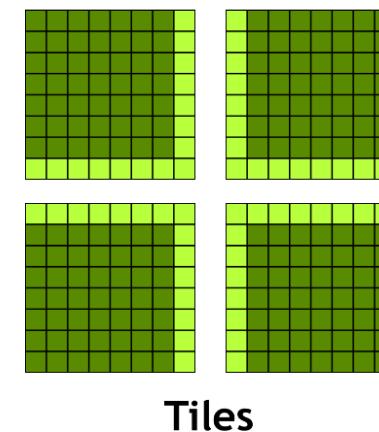
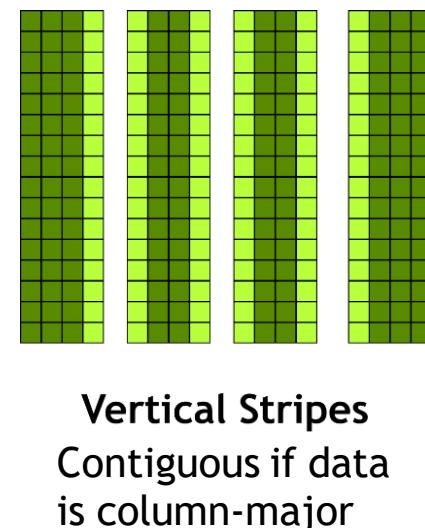
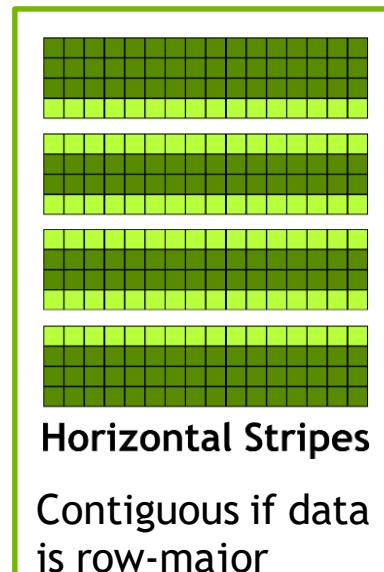
Communicate to less neighbors

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    ] );
```

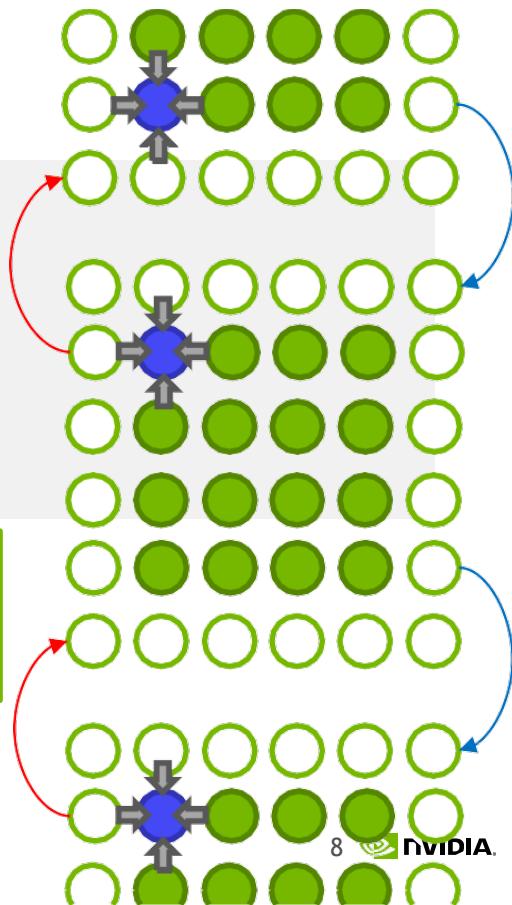
Apply periodic boundary conditions

Exchange halo with 2 neighbors

Swap `a_new` and `a`

Next iteration

One-step with  
ring exchange



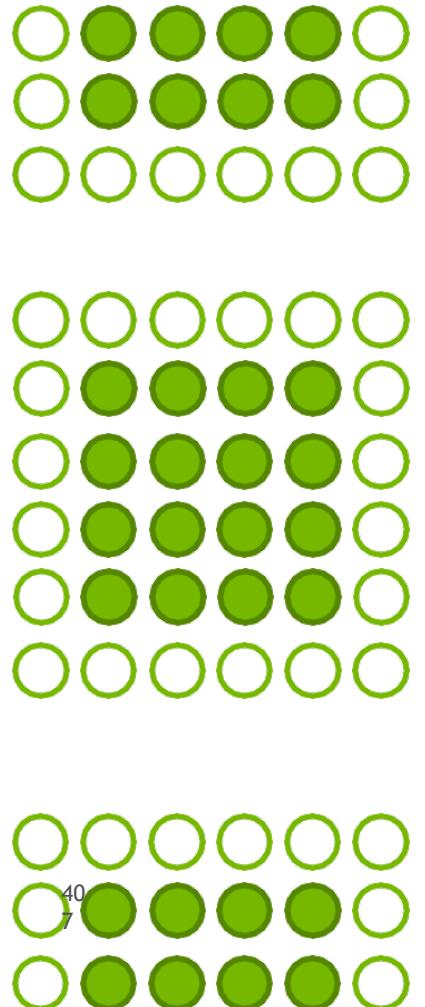
# SINGLE THREADED MULTI GPU PROGRAMMING

```
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

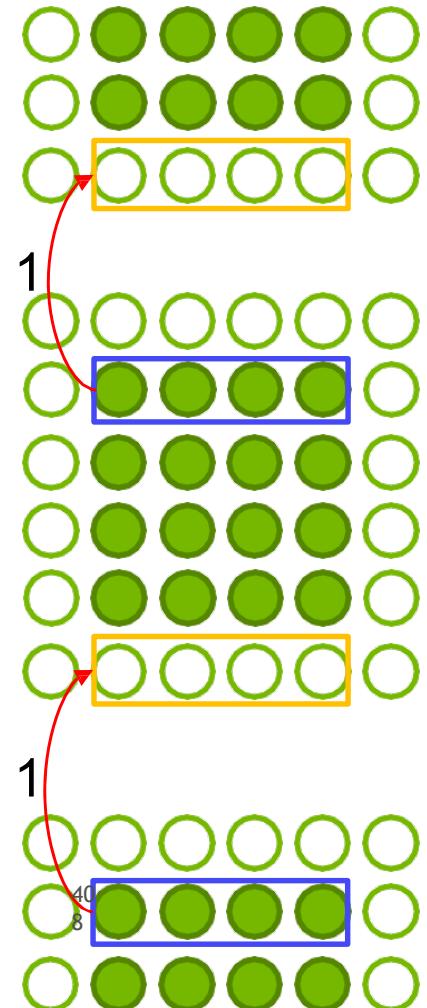
```
cudaMemcpyAsync(  
    a_new[top]+(iy_end[top]*nx) ,  
    a_new[dev_id]+iy_start[dev_id]*nx, nx*sizeof(real), ...);
```



# 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( 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, ...

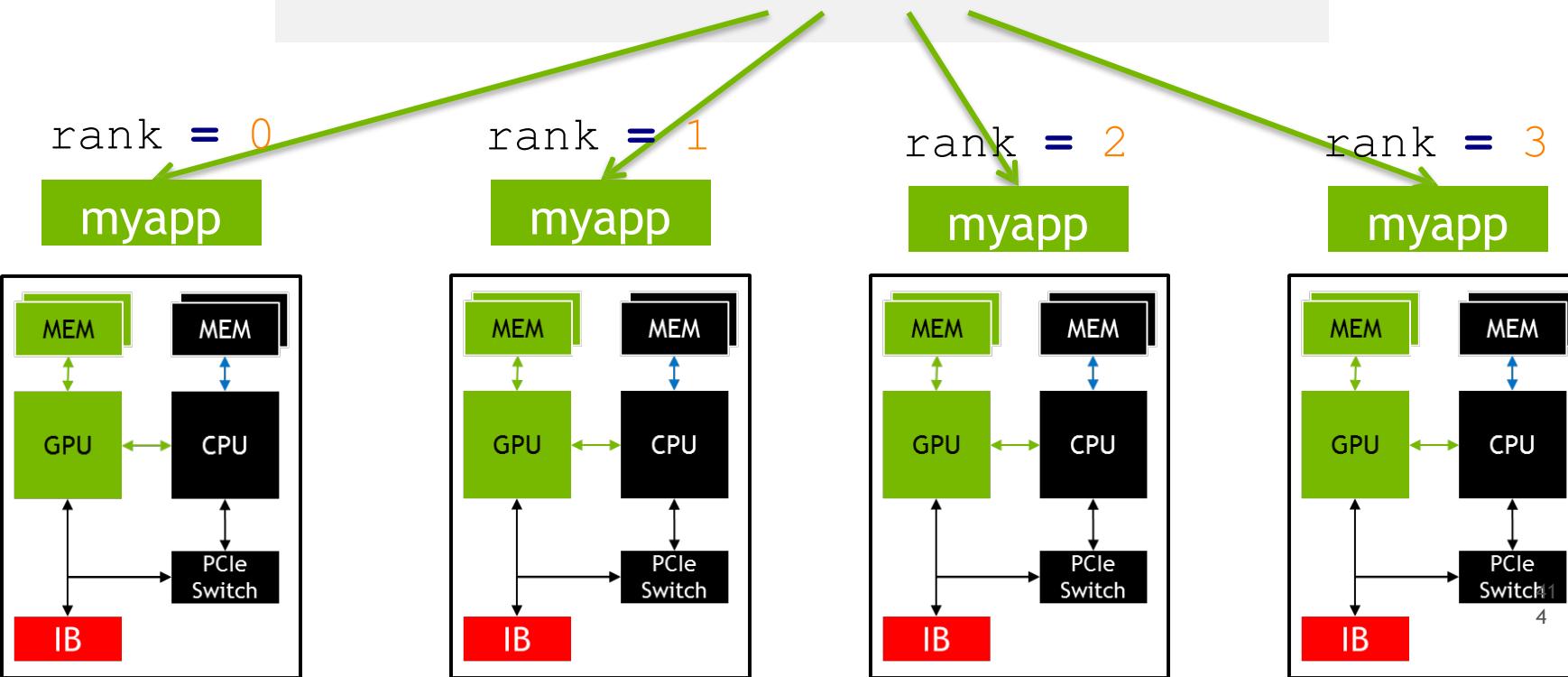
# MPI - SKELETON

```
#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

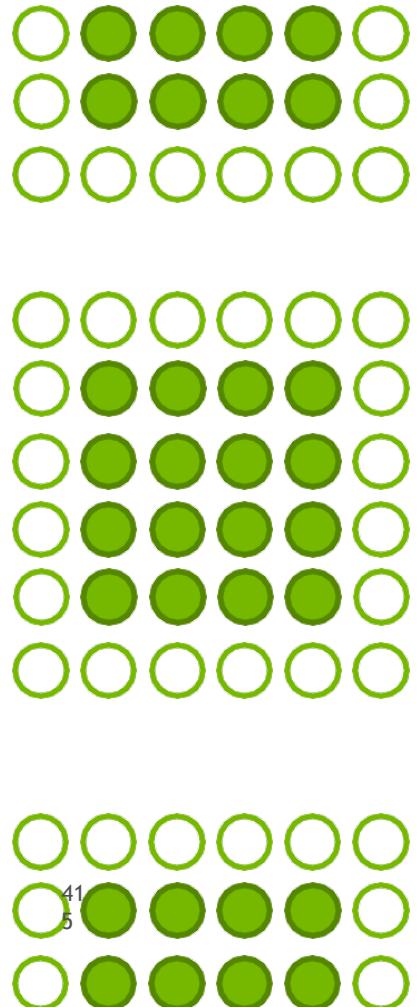
```
$ mpicc -o myapp myapp.c  
$ mpirun -np 4 ./myapp <args>
```



# EXAMPLE JACOBI

## Top/Bottom Halo

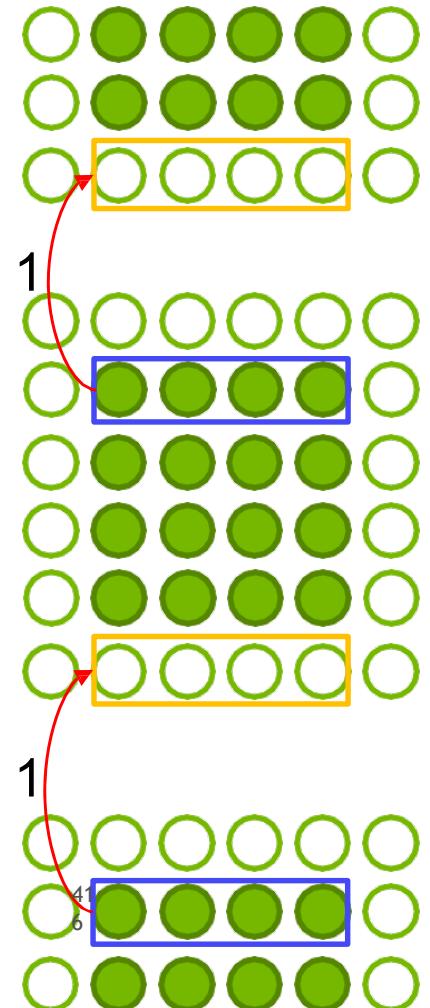
```
MPI_Sendrecv(a_new+iy_start*nx, nx, MPI_FLOAT, top , 0,
              a_new+(iy_end*nx), nx, MPI_FLOAT, bottom, 0,
              MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



# 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);
```

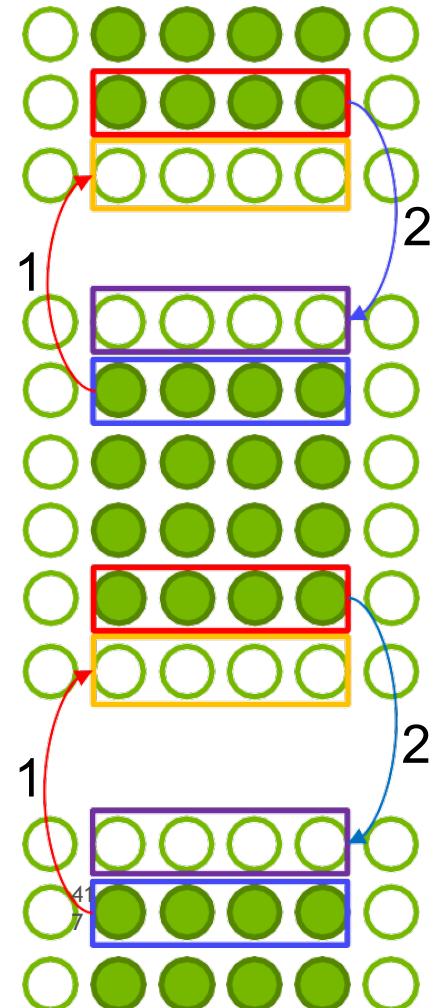


# EXAMPLE JACOBI

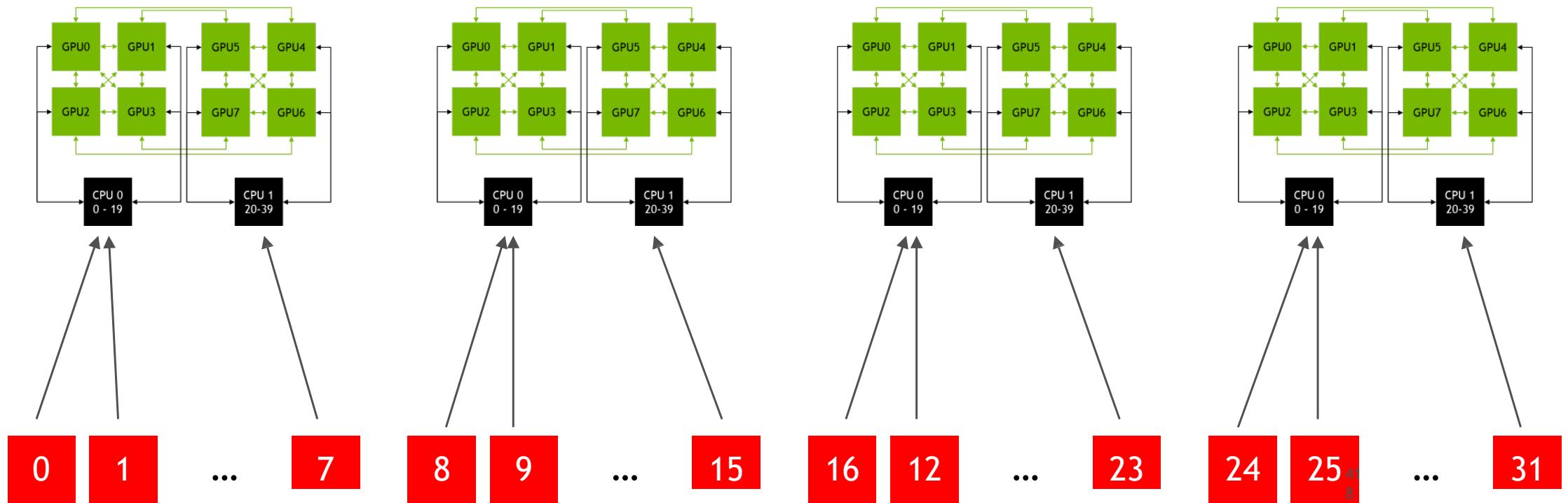
## Top/Bottom Halo

```
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

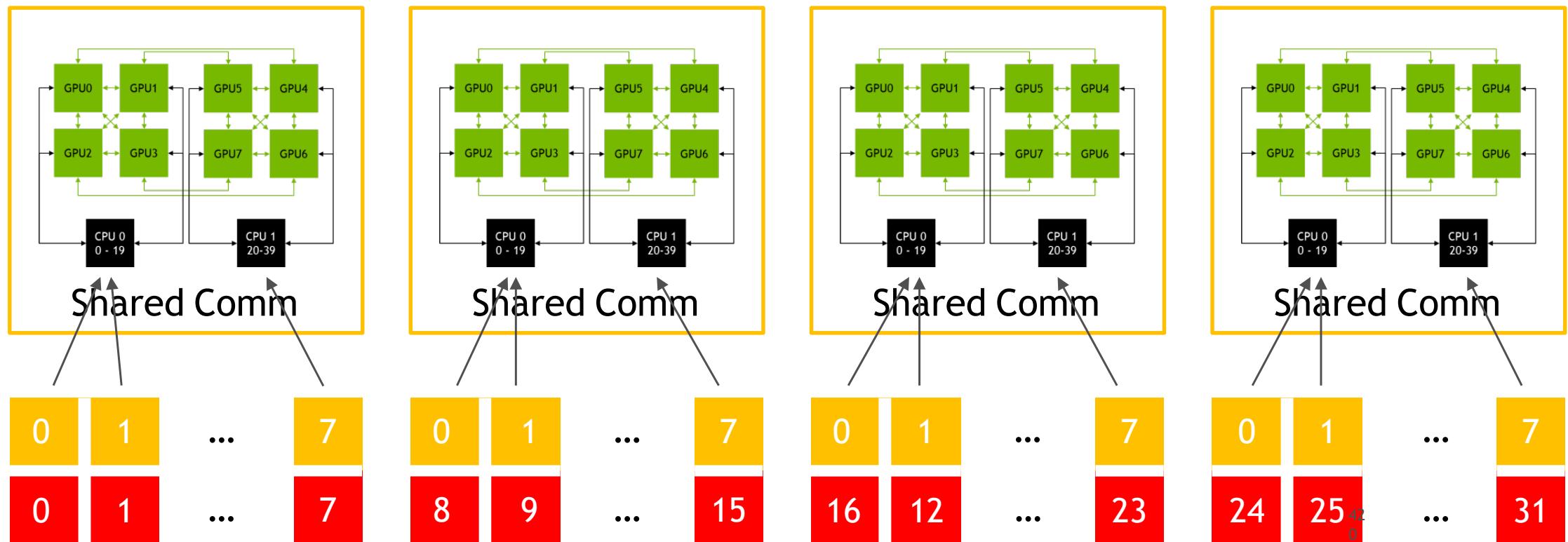


# HANDLING MULTIPLE MULTI GPU NODES

## 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

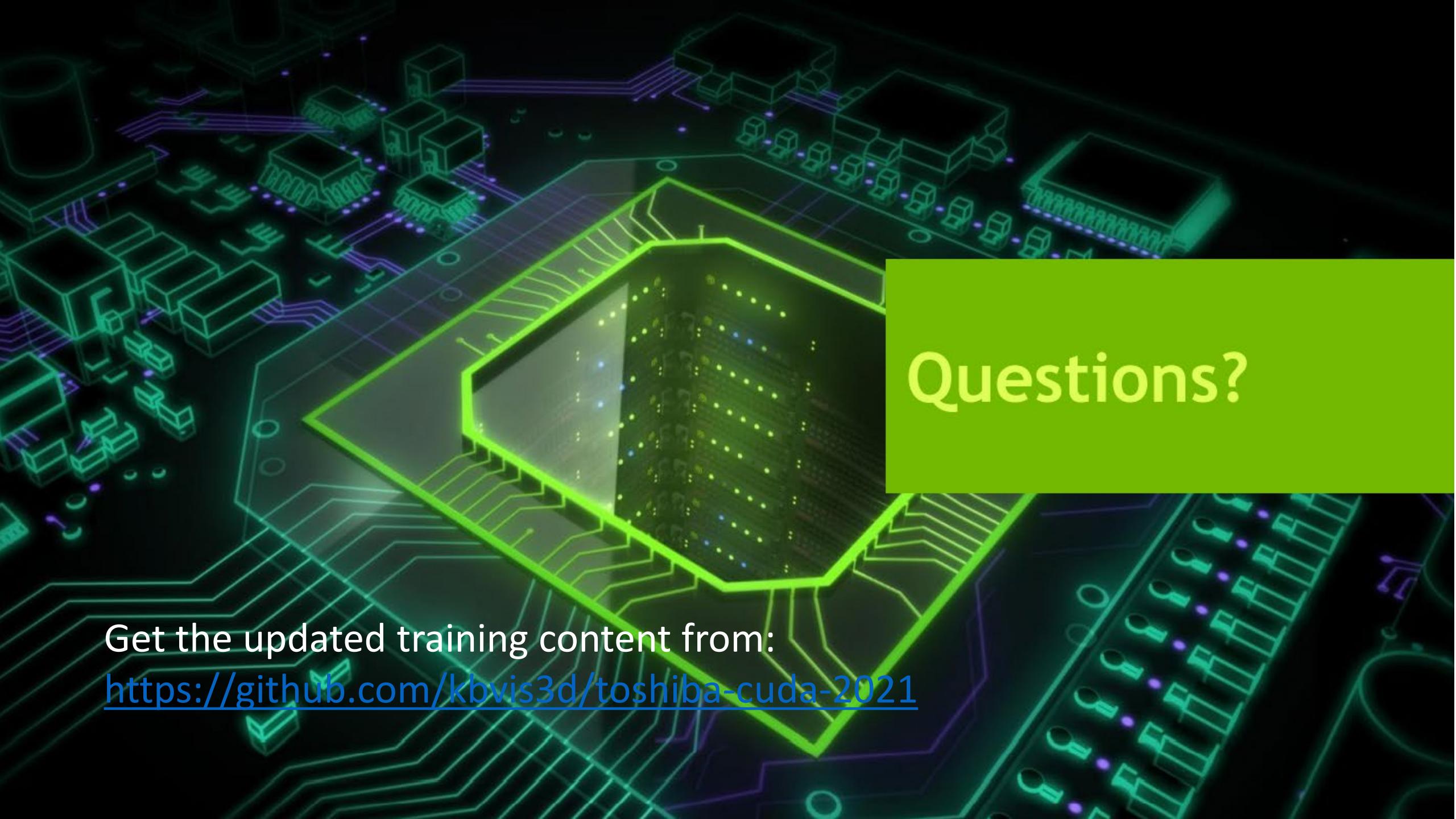
## GPU-affinity

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>