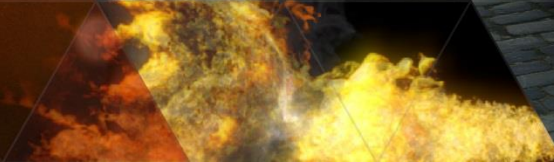
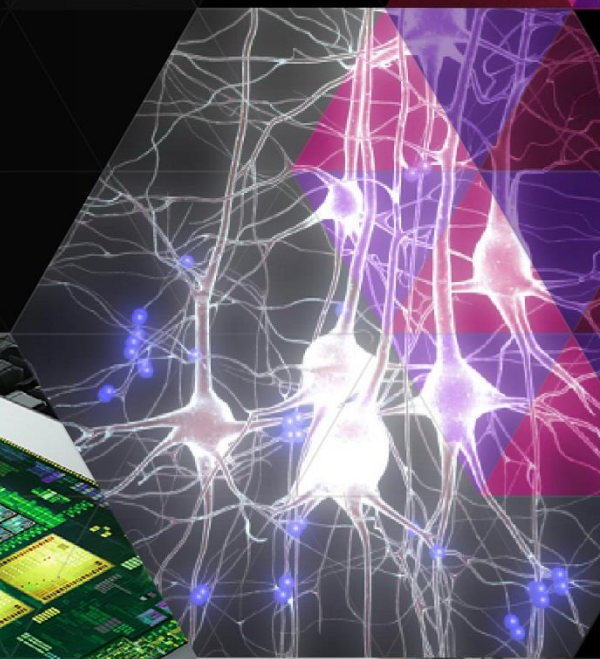
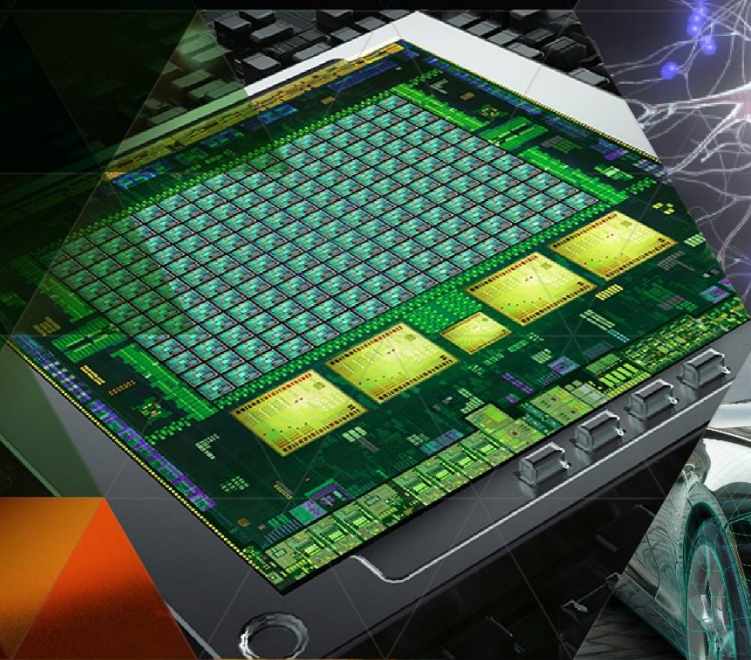




CUDA C/C++



What is CUDA?



- **CUDA Architecture**
 - Expose GPU parallelism for general-purpose computing
 - Retain performance
- **CUDA C/C++**
 - Based on industry-standard C/C++
 - Small set of extensions to enable heterogeneous programming
 - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C/C++

Introduction to CUDA C/C++



- **What will you learn in this session?**
 - Start from “Hello World!”
 - Write and launch CUDA C/C++ kernels
 - Manage GPU memory
 - Manage communication and synchronization

Prerequisites



- You (probably) need experience with C or C++
- You don't need GPU experience
- You don't need parallel programming experience
- You don't need graphics experience

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices



CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

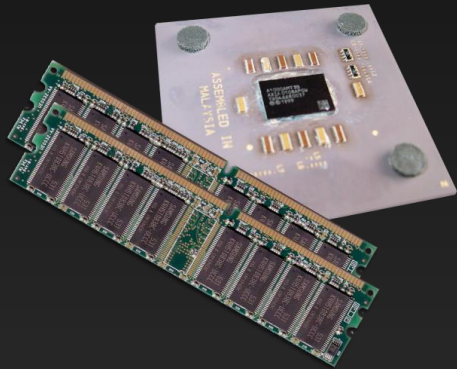
Managing devices

HELLO WORLD!

Heterogeneous Computing



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



Host



Device

Heterogeneous Computing



```
#include <ostream>
#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 gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int index = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[gindex] = in[gindex];
    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[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(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_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&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<<<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 fn

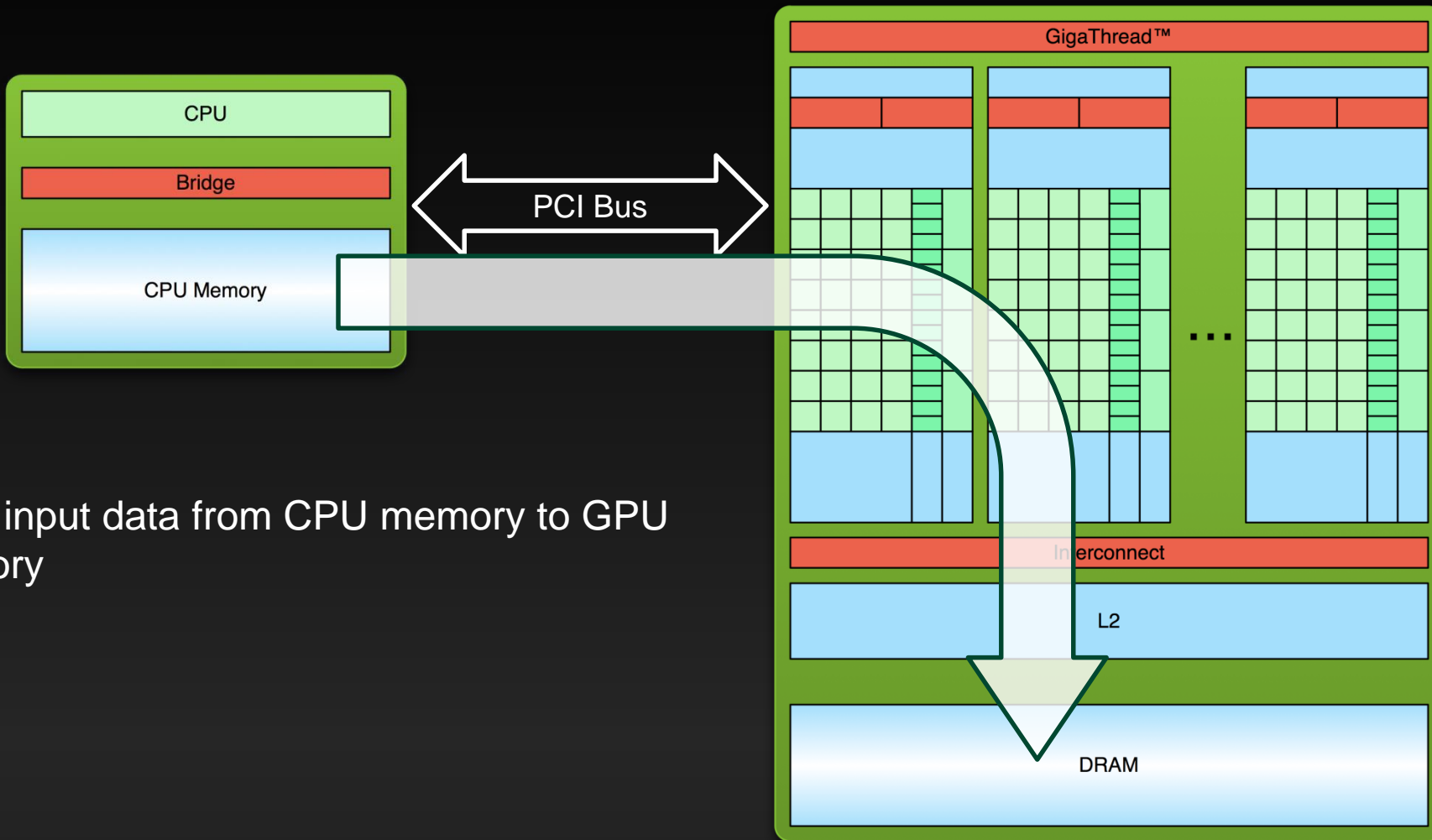
serial code

parallel code

serial code

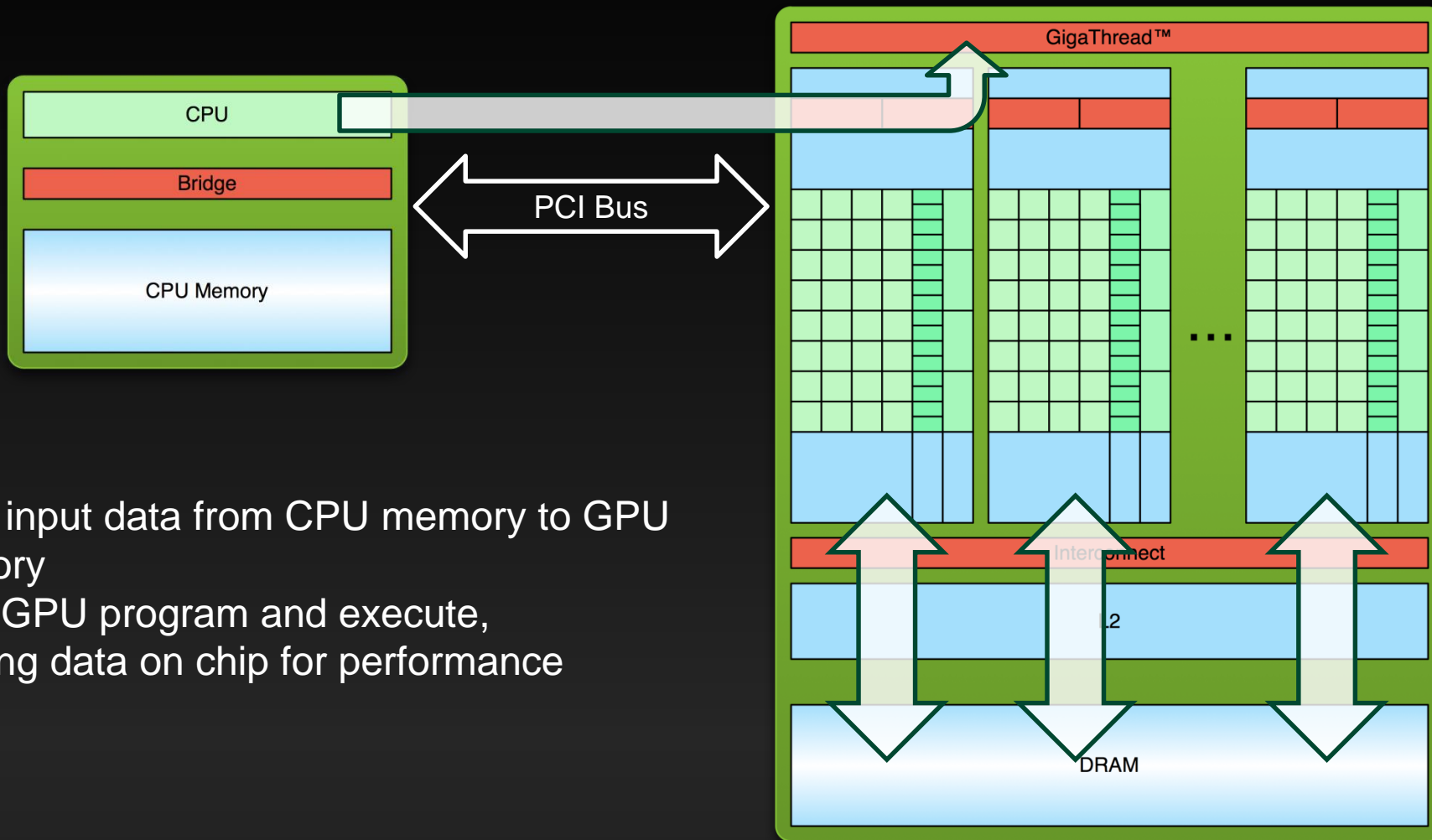


Simple Processing Flow



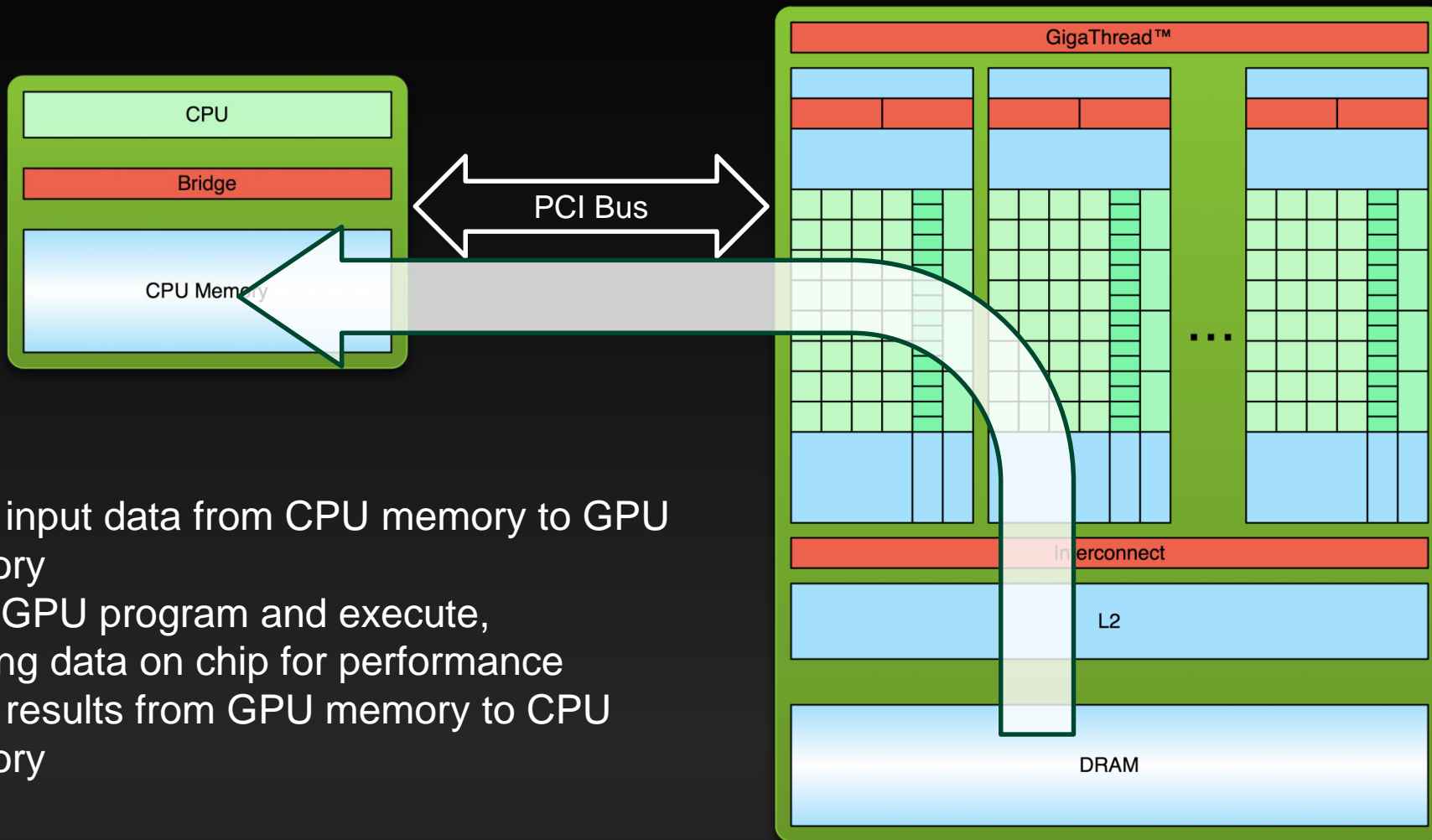
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

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

- Two new syntactic elements...

Hello World! with Device Code



```
__global__ void mykernel(void) {  
    printf("Hello world from device!\n");  
}
```

- 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>>>();
```

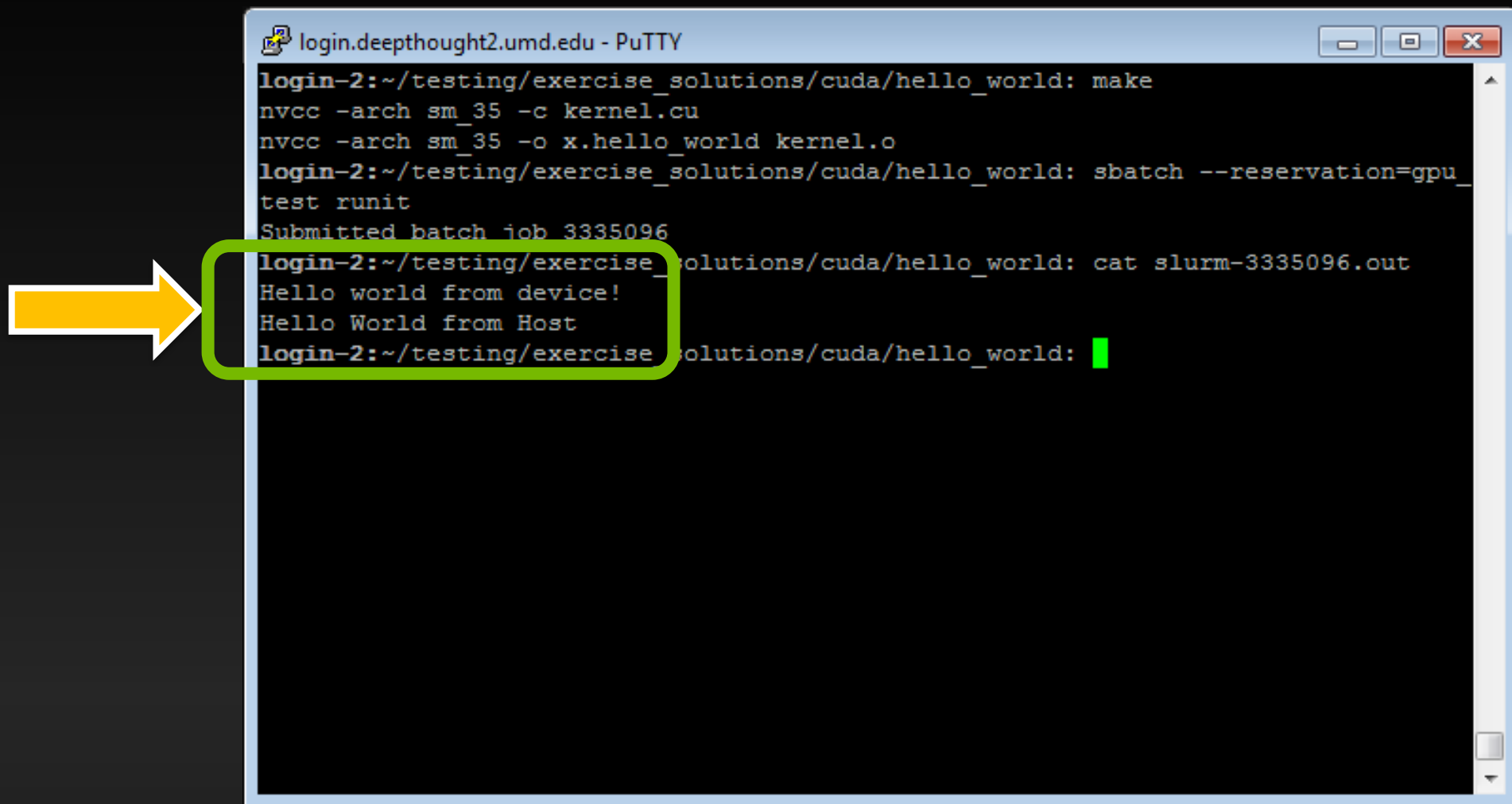
- Triple angle brackets mark a call from *host* code to *device* code
 - Also called a “kernel launch”
 - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

Hello world



- Login to your workstation
- Each coding project in a separate folder in the following dir
 - `maryland/exercises/`
- `cd maryland/exercises/cuda/hello_world`
- `module unload pgi`
- `module load cuda/6.5.14`
- All dirs have Makefiles for you
- Try building/running the code
 - `make`
 - `sbatch runit`

Screenshot



```
login.deepthought2.umd.edu - PuTTY
login-2:~/testing/exercise_solutions/cuda/hello_world: make
nvcc -arch sm_35 -c kernel.cu
nvcc -arch sm_35 -o x.hello_world kernel.o
login-2:~/testing/exercise_solutions/cuda/hello_world: sbatch --reservation=gpu_
test runit
Submitted batch job 3335096
login-2:~/testing/exercise_solutions/cuda/hello_world: cat slurm-3335096.out
Hello world from device!
Hello World from Host
login-2:~/testing/exercise_solutions/cuda/hello_world: █
```

Hello World! with Device Code



```
__global__ void mykernel(void) {  
    printf("Hello from device!\n");  
}
```

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

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

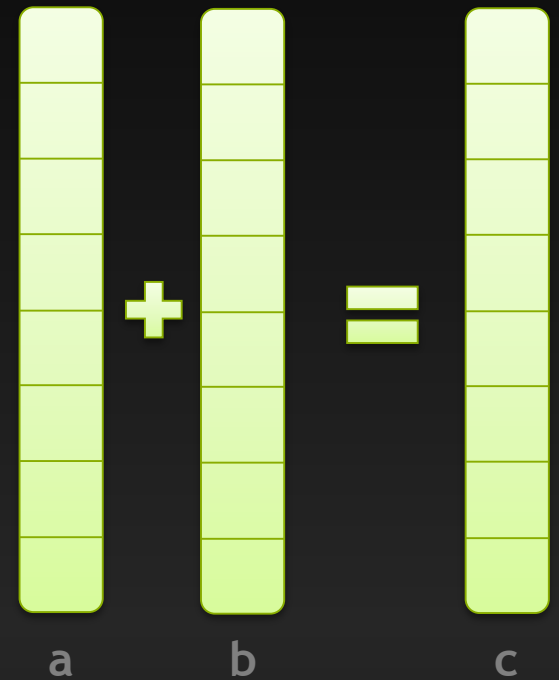
Output:

```
$ nvcc hello.cu  
$ a.out  
Hello from  
device!  
Hello from  
Host!  
$
```

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()`



Error checking



- **Kernels**
 - `kernel<<<>>> (...)`
 - `CUDA_CHECK ()`
 - `CUDA_CALL (cudaDeviceSynchronize)`
- **All other CUDA calls**
 - `CUDA_CALL (cuda... ())`
- **Add `-DDEBUG` to your compile line in Makefile**

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()...
- Open `exercises/cuda/simple_add/kernel.cu`
- Fill-in missing code as indicated.
 - Need to replace "FIXME" with code. Comments should help.
 - If something isn't clear, PLEASE ASK! 😊
 - PASS/FAIL will be printed to the screen

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

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

RUNNING IN PARALLEL

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 index

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()`...
- Open `exercises/cuda/simple_add_blocks/kernel.cu`
- Fill-in missing code as indicated.
 - Should be clear from comments where you need to add some code
 - Need to replace "FIXME" with the proper piece of code.

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



CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

INTRODUCING THREADS

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()`...
- Open `exercises/cuda/simple_add_threads/kernel.cu`

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;
}
```



CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

COMBINING THREADS AND BLOCKS

Combining Blocks and Threads

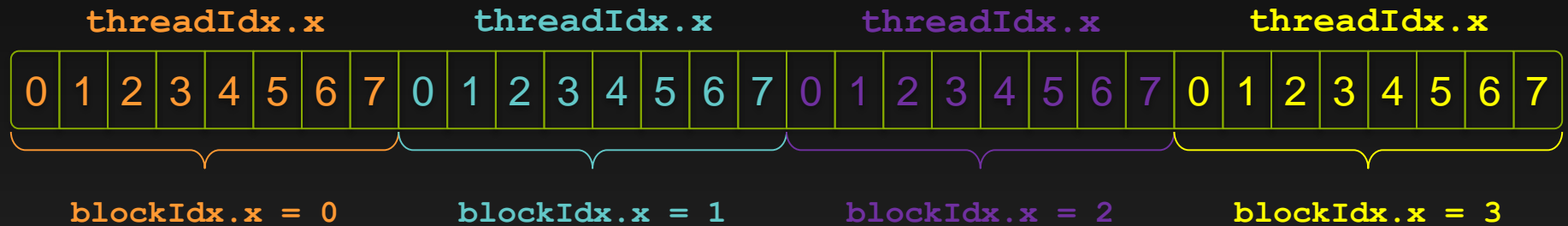


- We've seen parallel vector addition using:
 - Many blocks with one thread each
 - One block with many threads
- Let's adapt vector addition to use both *blocks* and *threads*
- Why? We'll come to that...
- First let's discuss data indexing...

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)

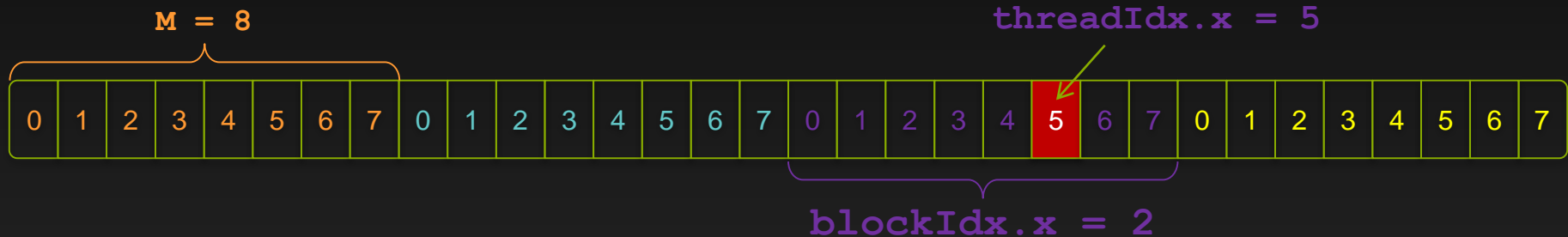
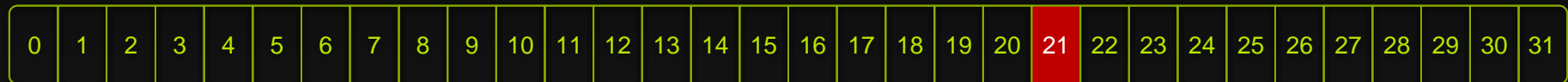


- With `M` threads/block a unique index for each thread is given by:
`int index = threadIdx.x + blockIdx.x * M;`

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()`?
- Open [exercises/cuda/simple_add_blocks_threads/kernel.cu](#)

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>>>(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:**
 - Communicate
 - Synchronize
- **To look closer, we need a new example...**

Review



- Launching parallel kernels
 - Launch n copies of `add()` with `add<<<N/M,M>>> (...)` ;
 - Use `blockIdx.x` to access block index
 - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

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

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

COOPERATING THREADS

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



dim3 datatype

- 3-dimensional vector type
- Used for setting threadblock and grid dimensions
- `dim3 threads(THREADS_PER_BLOCK, 1, 1);`
- `dim3 blocks(N / threads.x, 1, 1);`
- `kernel_call<<< blocks, threads >>>(args...);`
- Equivalent to:
- `kernel_call<<< N / THREADS_PER_BLOCK,
THREADS_PER_BLOCK >>> (args...);`

Simple Stencil in 1d



- Open [exercises/cuda/simple_stencil/kernel.cu](#)
- Finish the kernel and the kernel launch
 - Each thread calculates one stencil value
 - Reads $2 \cdot \text{RADIUS} + 1$ values
 - `dim3` type: CUDA 3 dimensional struct used for grid/block sizes
- Inserted GPU timers into code to time the execution of the kernel
- Try various sizes of N, RADIUS, BLOCK
- Time a large (2048*2048) value of N with a RADIUS of 7

Can we do better?



- **Input elements are read multiple times**
 - With $RADIUS=3$, each input element is read seven times!
 - Neighbouring threads read most of the same elements.
 - Thread 7 reads elements 4 through 10
 - Thread 8 reads elements 5 through 11
- **Can we avoid redundant reading of data?**

Sharing Data Between Threads

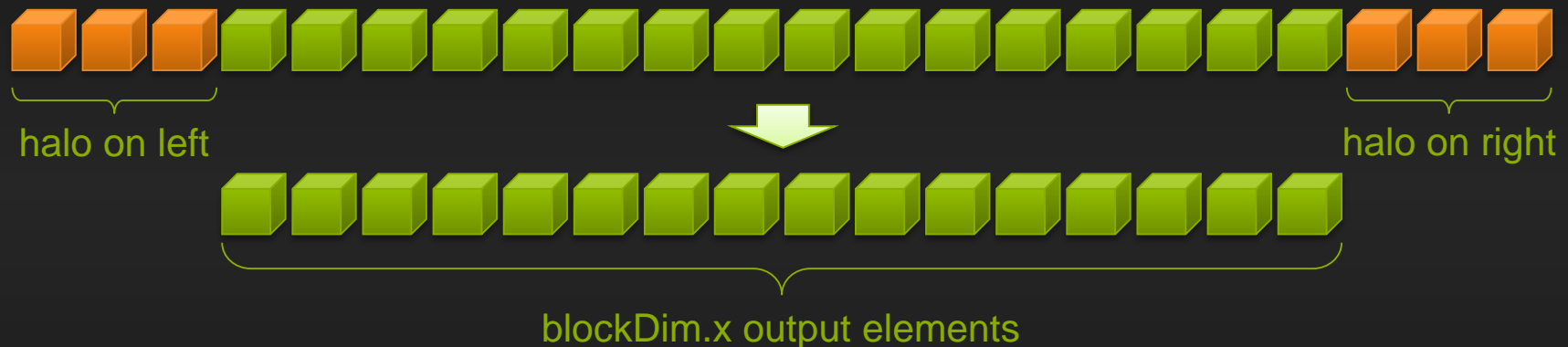


- Terminology: within a block, threads share data via **shared memory**
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory



- Cache data in shared memory (user managed scratch-pad)
 - Read $(\text{blockDim.x} + 2 * \text{radius})$ 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;
}
```

Simple Stencil 1d with shared memory





- `cd exercises/cuda/simple_stencil_smem/`
- Run the code. It should build/run without modification.
 - If Errors occur, first offending element will be printed to the screen
 - FAIL will be printed to the screen
- What is the result with $N=(1024*1024)$ and `THREADS_PER_BLOCK=32`?
- What is the result with $N=(1024*1024)$ and `THREADS_PER_BLOCK=64`?
 - Why?

Data Race!



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

```
temp[lindex] = in[gindex];      Store at temp[18]        
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];      Skipped, threadIdx > RADIUS  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
int result = 0;  
result += temp[lindex + 1];    Load from temp[19]    
```

__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
- Insert `__syncthreads ()` into the kernel in the proper location
- Compare timing of previous simple stencil with the current shared memory implementation for same (large N) and BLOCK=512

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
 - Using large shared mem size impacts number of blocks that can be scheduled on an SM (48K total smem size)
- Use `__syncthreads()` as a barrier
 - Use to prevent data hazards

Global Memory Throughput

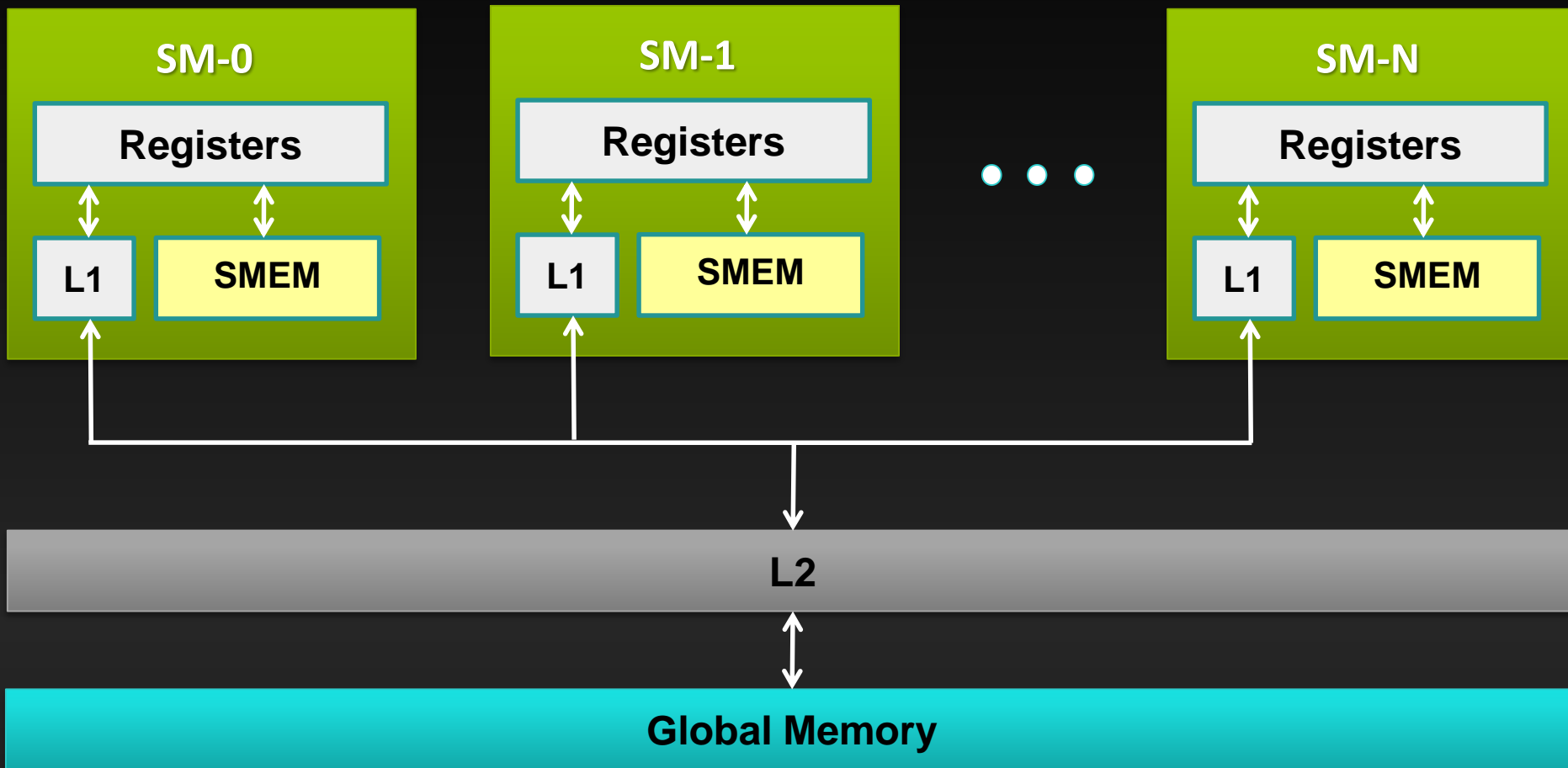


Memory Hierarchy Review



- Local storage
 - Each thread has own local storage
 - Mostly registers (managed by the compiler)
- Shared memory / L1
 - Program configurable: **16KB shared / 48 KB L1** OR **48KB shared / 16KB L1**
 - Shared memory is accessible by the threads in the same threadblock
 - Very low latency
 - Very high throughput: **1+ TB/s** aggregate
- L2
 - All accesses to global memory go through L2, including copies to/from CPU host
- Global memory
 - Accessible by all threads as well as host (CPU)
 - High latency (**400-800 cycles**)
 - Throughput: up to **177 GB/s**

GPU Memory Hierarchy Review



Load Operation

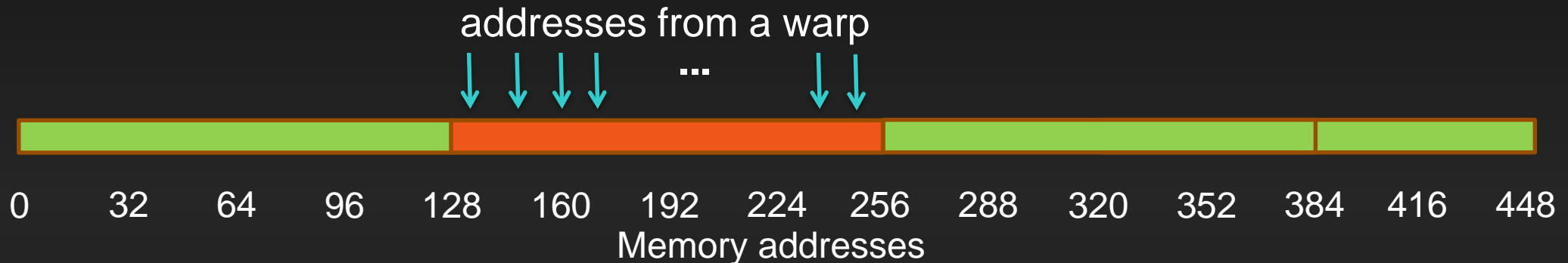


- Memory operations are issued **per warp (32 threads)**
 - Just like all other instructions
- Operation:
 - Threads in a warp provide memory addresses
 - Determine which lines/segments are needed
 - Request the needed lines/segments

Caching Load--coalesced



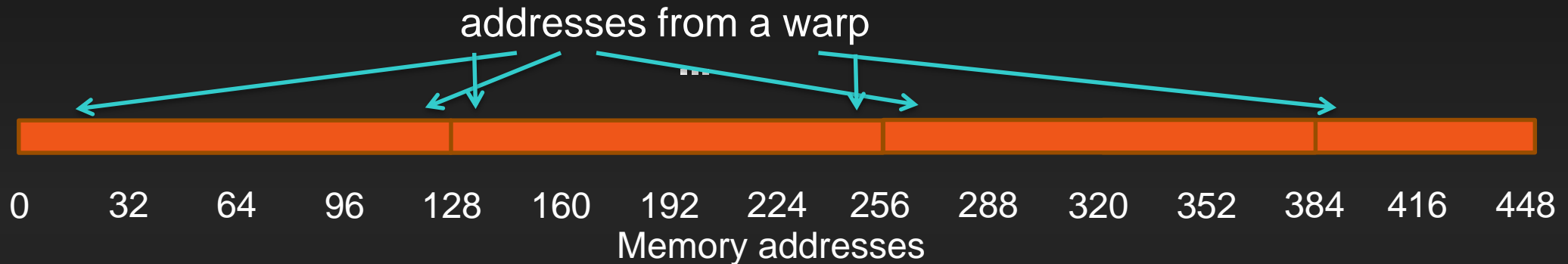
- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
 - Warp needs 128 bytes
 - 128 bytes move across the bus on a miss
 - Bus utilization: **100%**



Caching Load--uncoalesced



- Warp requests 32 scattered 4-byte words
- Addresses fall within N cache-lines
 - Warp needs 128 bytes
 - $N*128$ bytes move across the bus on a miss
 - Bus utilization: $128 / (N*128)$



GMEM OPTIMIZATION GUIDELINES

- ▶ Strive for perfect coalescing!!!
- ▶ Have enough concurrent accesses to saturate the bus
 - ▶ Process several elements per thread
 - ▶ Multiple loads get pipelined
 - ▶ Indexing calculations can often be reused
 - ▶ Launch enough threads to maximize throughput
 - ▶ Latency is hidden by switching threads (warps)

Shared Memory

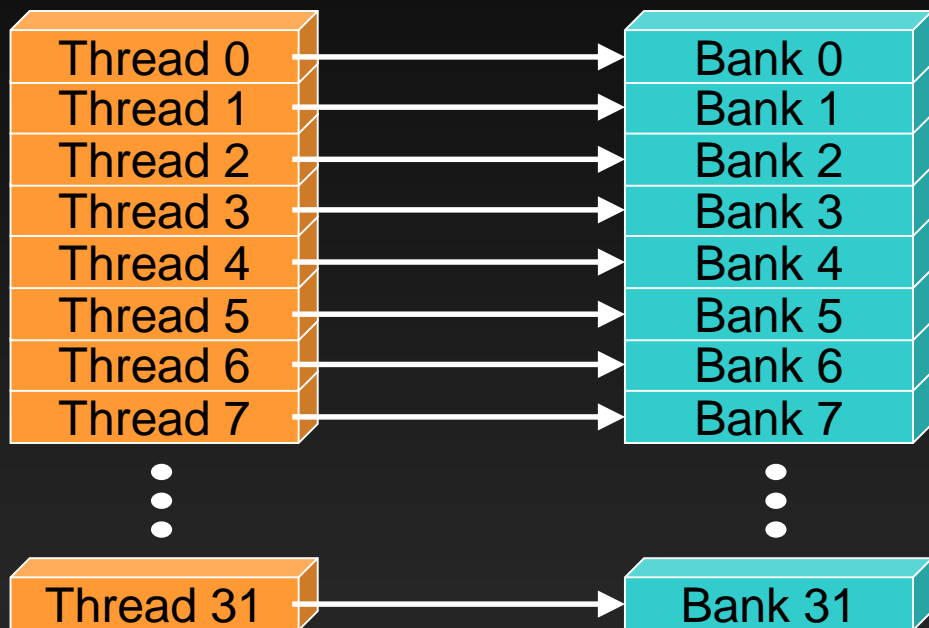
SHARED MEMORY

- ▶ Uses:
 - ▶ Inter-thread communication within a block
 - ▶ Cache data to reduce redundant global memory accesses
 - ▶ Use it to improve global memory access patterns
- ▶ Organization:
 - ▶ 32 banks, 4-byte wide banks
 - ▶ Successive 4-byte words belong to different banks
- ▶ If you use shared memory in a kernel, you should almost always use `__syncthreads()` to avoid race conditions!!!

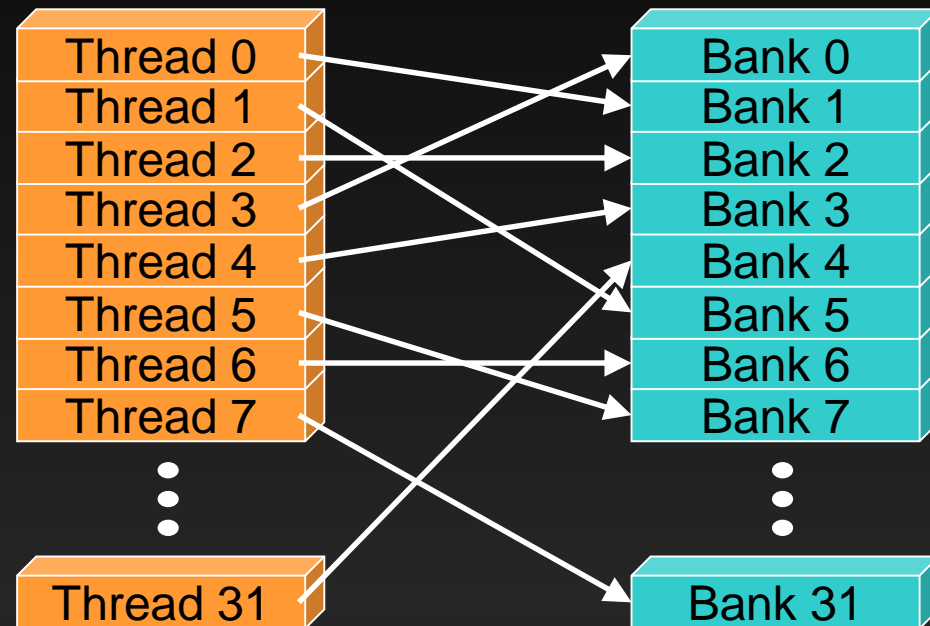
Bank Addressing Examples



● No Bank Conflicts



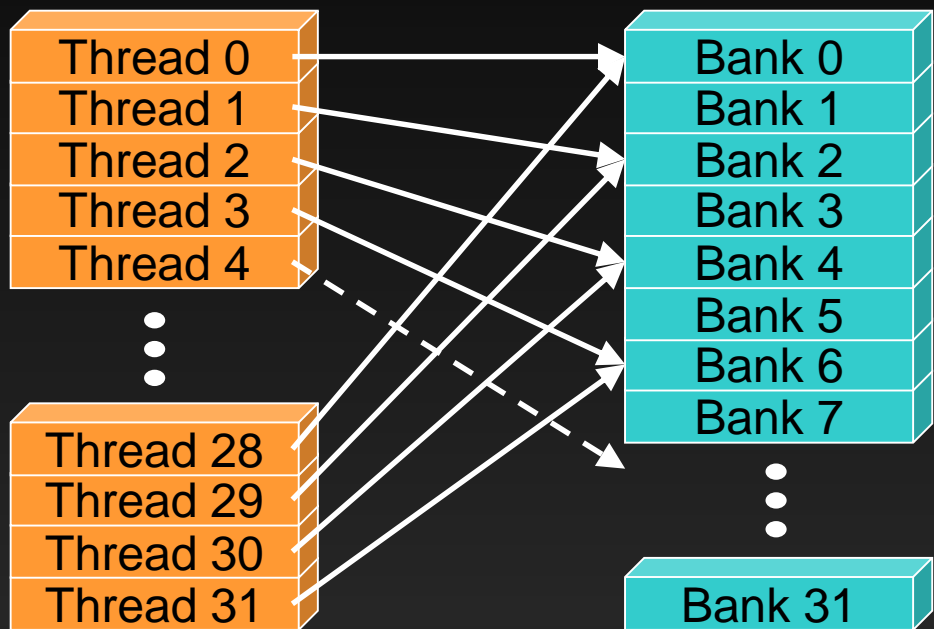
● No Bank Conflicts



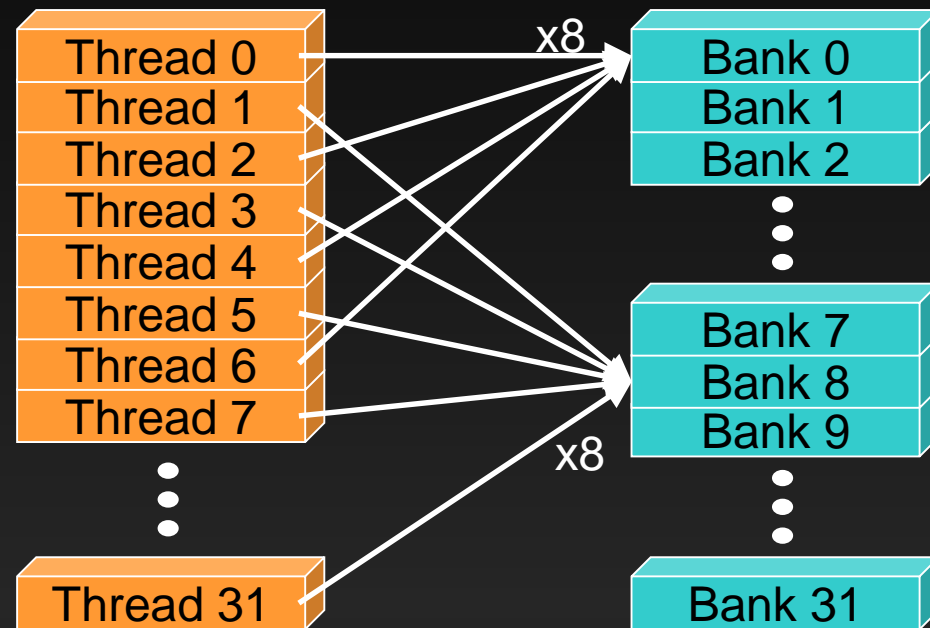
Bank Addressing Examples



2-way Bank Conflicts



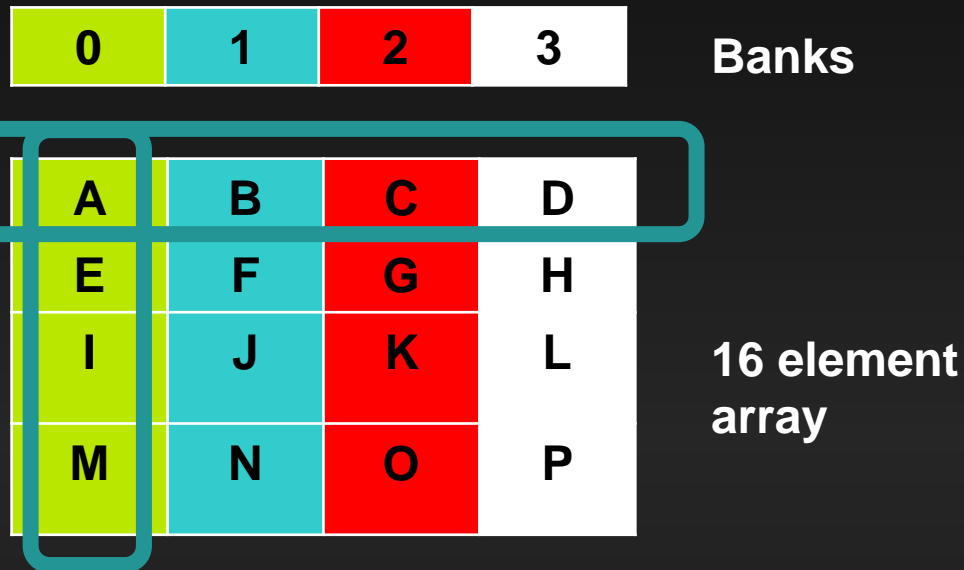
8-way Bank Conflicts



Shared memory: Banks and Conflicts



```
__shared__ float temp[4][4]
```

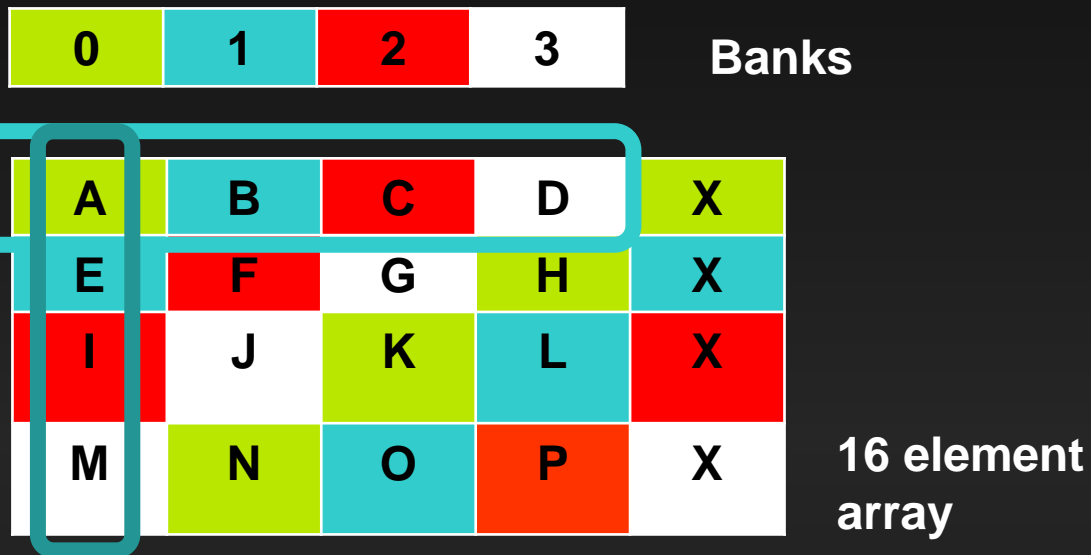


- Consider 4x4 SMEM array
- Consecutive threads access consecutive elements in a row
 - PERFECT ACCESS!
- Consecutive threads access consecutive elements in a column
 - 4-way bank conflict because 4 threads try to access same bank.

Shared memory: Avoiding bank conflicts



`__shared__ float temp[4][5]`



- Consider 4x5 SMEM array
 - Padded the column dimension
- Consecutive threads access consecutive elements in a row
 - PERFECT ACCESS!
- Consecutive threads access consecutive elements in a column
 - PERFECT ACCESS!



Matrix Transpose

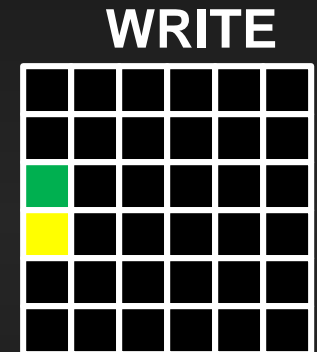
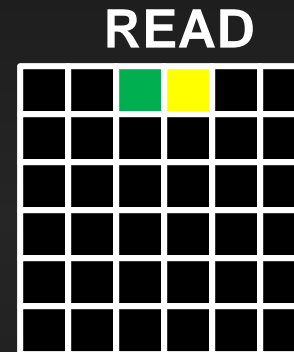


Matrix Transpose



- Open `exercises/cuda/naive_transpose/kernel.cu`
 - Defined `INDX(row,col,ld)` to translate 2d coordinates to 1d index
- Column-major double precision elems
 - Out-of-place transpose
- Naïve implementation:
 - Square threadblocks
 - Each thread:
 - Computes its global x and y coordinates
 - Reads from (x,y) , writes to (y,x)

 thread X
 thread (X+1)



naive_transpose/kernel.cu



- Use `INDX(row,col,ld)` to translate 2d coordinates to 1d index
 - Defined as `MACRO` at top of source code
- Finish the kernel and kernel launch parameters
 - Each thread transposes one element from A to C, out-of-place
- Kernel is compared to a CPU-side transpose
 - Both performance and answers are compared
- Once you get correct answers in your kernel:
 - Measure performance of kernel with `N=1024` and `N=4096`

NVIDIA Visual Profiler



- `nvvp` &
- Choose `File->New Session`
- Choose `exercises/cuda/naive_transpose/x.transpose`
- Select proper “naive_transpose” for working directory
- Then click “Next”
- Choose timeout = 60 seconds.
- Uncheck “Run Analysis” then click “Finish”
- Click “Analyze All”
 - Profiler will execute the code multiple times to record all performance counters
- In the timeline window, click on the kernel of interest

Profiler results?



- What does the profiler output show us?
- Why are load and store performance so divergent?
- Ideas for potential improvement?

Cause and Remedy



- **Cause:**
 - Due to nature of operation, one of the accesses (read or write) will be at large strides, i.e., uncoalesced!
 - 32 doubles (256 bytes) in this case (on a warp basis)
 - Thus, bandwidth will be wasted as only a portion of a transaction is used by the application
- **Remedy**
 - Stage accesses through shared memory
 - A threadblock:
 - Reads a tile from GMEM to SMEM
 - Transposes the tile in SMEM
 - Write a tile, in a coalesced way, from SMEM to GMEM

smem_transpose/kernel.cu



- Finish the kernel code
 - Particularly the index calculations for the transpose.
- One thread block cooperatively operates on a block of the matrix
 - Index calculations need to know which block I am in the full matrix
- **HINT: You are using shared memory. What should you include when using smem?**
- Once you get correct answers in your kernel:
 - Measure performance of kernel with $N=1024$ and $N=4096$

Visual Profiler



- Run the profiler
- What are the results?
- What is happening?
- How to fix it?
 - Hint: requires adding only 2 characters to the kernel source!

SMEM bank conflicts



- Recall that smem has 32 banks of 4 bytes each
- When multiple threads IN THE SAME WARP access the same bank, a conflict occurs and performance is affected negatively
- Consider `__shared__ double s[16][16]`
 - Read/write `s[tidx][tidy]` (each read/write requires two 4byte banks)
 - `tidx` are consecutive threads
 - They are accessing the `s[][]` array with stride of 16 doubles
 - 16 doubles == 128 bytes == 32 banks * 4 bytes.
 - `s[0][tidy]` accesses banks 0,1 to grab its 8 byte double
 - `s[1][tidy]` accesses banks 0,1 to grab its 8 byte double
 - **16 threads are all accessing banks 0,1 in the same transaction!!!**

SMEM bank conflicts cont'd

- tidx from 0 to 16 all access banks 0,1
 - 16 way bank conflict! VERY BAD for performance
- How to remedy?
 - Pad shared memory.
- `__shared__ double s[16][17]`
 - Now the stride between success tidx is 17 doubles,
 - i.e., $17 * 8\text{bytes} = 136\text{ bytes}$
 - More importantly, 136 byte stride will be $32 + 2$ banks
 - `s[0][tidy]` accesses banks 0,1 to grab its 8 byte double
 - `s[1][tidy]` accesses banks 2,3 to grab its 8 byte double
 - **16 threads are all accessing different banks!!!**

Still in smem_transpose/kernel.cu



- Remedy the smem bank conflicts
- Once you get correct answers in your kernel:
 - Measure performance of kernel with $N=1024$ and $N=4096$
- Run with the profiler again.
 - Verify the bank conflicts have gone away

Review



- **SMEM often used to alleviate poor GMEM accesses**
 - **Uncoalesced loads/stores were solved using SMEM**
- **SMEM almost always requires `__syncthreads()`**
- **SMEM often requires an analysis to minimize bank conflicts.**
- **Use NVIDIA Visual Profiler to identify performance bottlenecks**

Matrix multiply



Matrix Multiply



- The foundation of lots of linear algebra
- High compute/communication ratio
 - Access $O(N^2)$ data and execute $O(N^3)$ operations
- Relatively simple algorithm
 - Great teaching algorithm
- Well-written code shows off the power of CPU and/or GPU

Matrix Multiply cont'd

- Matrix A with M rows and K cols
- Matrix B with K rows and N cols
- $A * B = C$
 - C has M rows and N cols

- $$C_{i,j} = \sum_k^K A_{i,k} * B_{k,j}$$

- The dot product of the *i*th row of A and *j*th col of B yields the *i,j* element of C

matmul_CPU



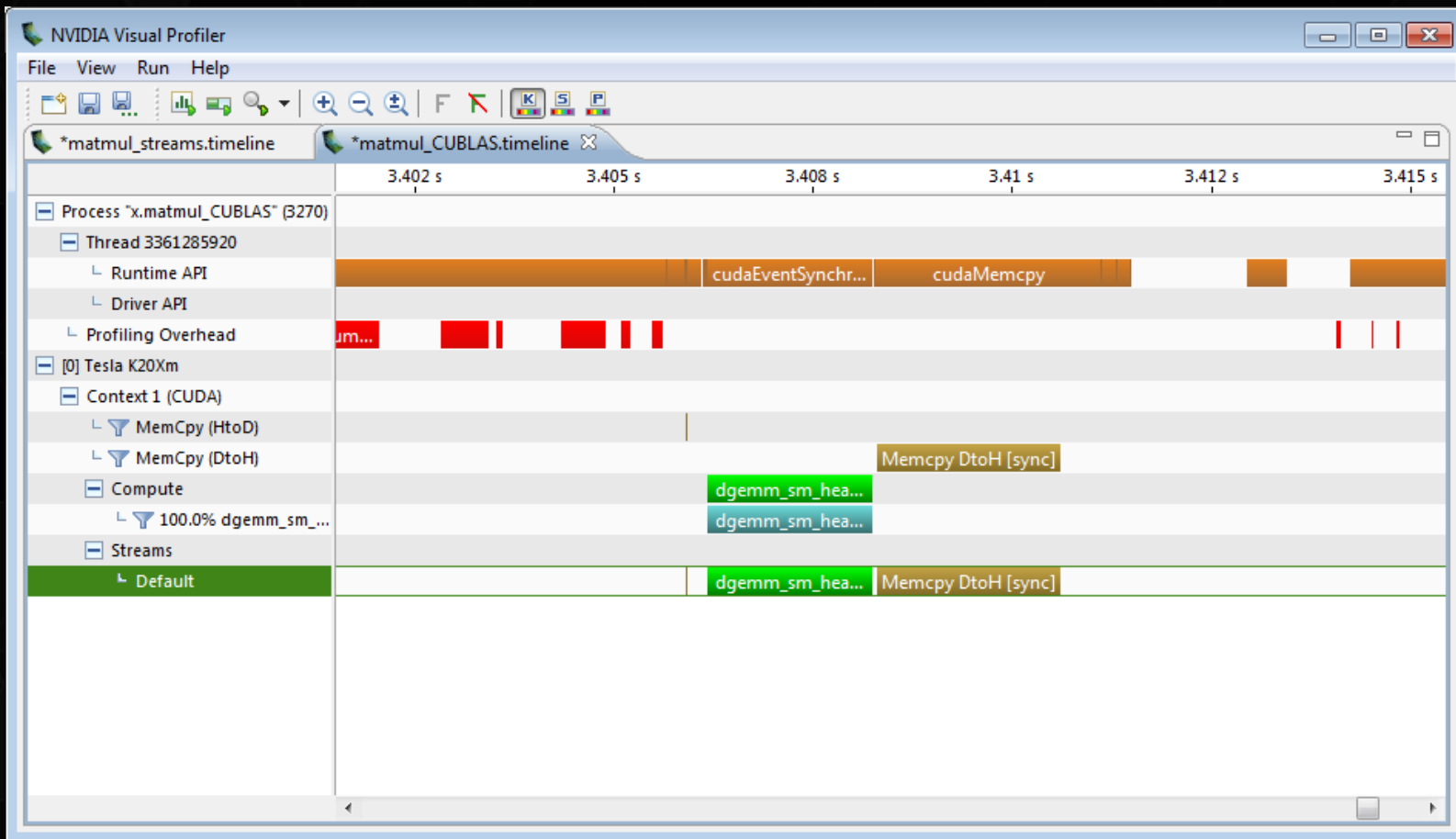
- Open `exercises/cuda/matmul_CPU/kernel.cu`
- CPU-only example
- Complete “host_dgemm” function
 - Finish the index calculations for the arithmetic in inner loop.
- What is the performance of your naïve CPU matrix multiply?
- DGEMM is often measured in terms of percentage of peak.
 - Intel [E5-2665@2.4](#) GHz has peak of $2.4\text{GHz} * 8 \text{ DP flops/clock} = 19.2\text{GF}$
 - What is your code’s percent of peak?

Matmul on GPU



- In reality we'd never write a matmul for GPU
 - Call NVIDIA's CUBLAS library
- `cd exercixes/cuda/matmul_CUBLAS`
 - Code should build/run without modification.
- Run the code with $N=1024$ and record performance of CUBLAS `dgemm`
 - How does it compare to your naïve CPU code?

NVVP OF CUBLAS



Matmul on GPU

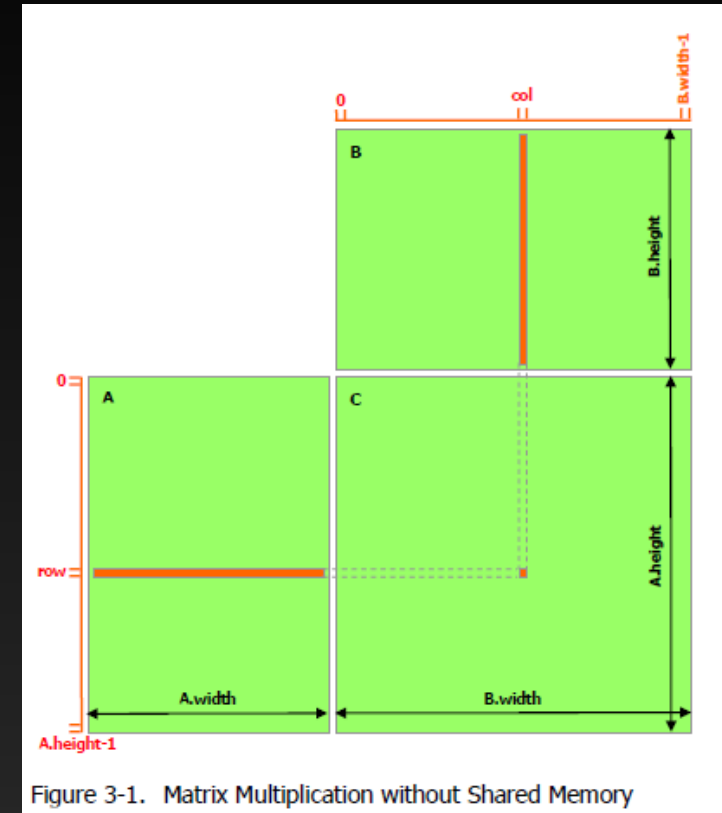


- We will write matrix multiply on GPU
- Use square matrices for simplicity
- Use powers of 2 so we can avoid writing the extra code for end cases
- Write three versions utilizing successively advanced optimization ideas, based in part on profiling results we obtain

matmul_GPU_naive



- Open `exercises/cuda/matmul_GPU_naive/kernel.cu`
- Finish the kernel
 - Add the appropriate index calculations to the kernel
 - Answers are compared against CUBLAS
 - You should see an error message printed to the console if your results are suspect!
- Record performance of $N=1024$
 - How does it compare to CUBLAS?



Visual profiler



- Profile the code with Visual Profiler
- What are some performance considerations?
- Consider two successive threads
 - How is global memory accessed for matrices A, B, C?
- What is a choice we have if we wish to remedy uncoalesced GMEM accesses?

Strategies to improve



- **Use shared memory to achieve better coalescing**
 - Allows us to share data among threads
 - Reduces number of times data must be read because we reuse from smem rather than fetch from GMEM each time.
- **Similar to the matrix transpose example:**
 - Load block of A into SMEM
 - Load block of B into SMEM
 - Compute block of C from A and B in SMEM
- **Open `exercises/cuda/matmul_GPU_shmem/kernel.cu`**
 - `#pragma unroll` to unroll loops of predefined trip count

matmul_GPU_shmem



- For each block in K direction
 - Load block of A into SMEM
 - Load block of B into SMEM
 - Use these blocks to contribute to block of C
- When using SMEM, what function should you include???
- Record performance of N=1024
 - How does it compare to naïve kernel?
 - Why???

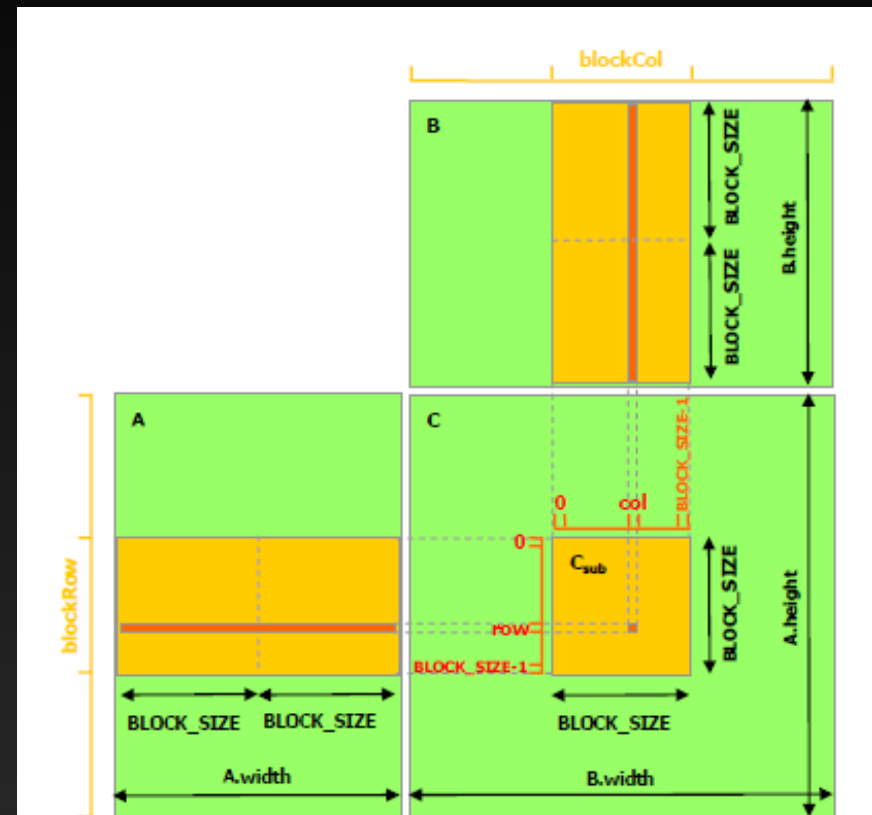


Figure 3-2. Matrix Multiplication with Shared Memory

Visual profiler



- We solved the uncoalesced memory issue.
- What does profiler show us?
- What is the fix for this?
- What is the performance with this change?

Algorithmic improvements



- Currently one thread block calculates one block of C
 - By extension one thread only calculates one element of C
 - A natural extension is to try having one thread block calculate multiple portions of C and thus one thread calculating multiple values of C
 - This would reduce the number of times A and B are fetched from $GMEM$ and increase the computational intensity of the thread block
 - A priori not obvious how many blocks C should calculate
- Write a new kernel that does this idea in a general fashion
 - Experiment with different block sizes etc.
- MAGMA from UTK <http://icl.cs.utk.edu/magma/> has done extensive work on dense linear algebra

matmul_GPU_shmem1



- Open `exercises/cuda/matmul_gpu_shmem1/kernel.cu`
- Lots of `#define` at the top of the source code
 - Pay special attention to the defined constants and how they interact with each other!
 - Keep $TX=TY=BK=16$
 - $TBX=TBY=16$ and $NX=NY=1$ is equivalent to the previous kernel.
- Complete the kernel with some important reminders
 - What function should you automatically use with SMEM???
 - How can you avoid those nasty bank conflicts in SMEM???
 - Refer to diagram in cheatsheet for pictorial view of algorithm
 - Lots of code to fill in. This one is challenging!!!

<http://www.netlib.org/lapack/lawnspdf/lawn227.pdf>

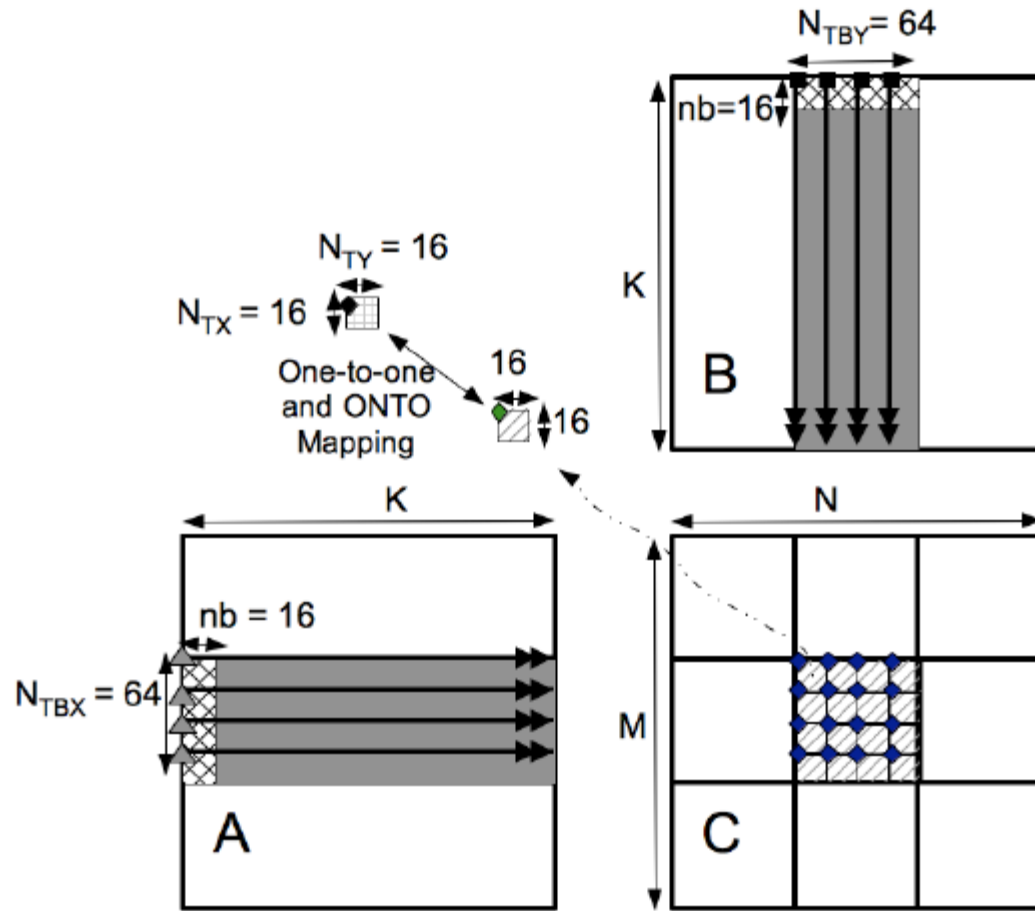


Fig. 2. The GPU GEMM ($C := \alpha AB + \beta C$) of a single TB for Fermi.

matmul_GPU_shmem1



- **Keep N=1024**
 - Try different values of TBX, TBY, NX, NY
 - What is the best combination?
 - Are there multiple “best” values?
- **With the best combination you found, record performance for N=1024**
 - How does it differ with padded or unpadded SMEM?
 - How does it compare to the CUBLAS result we saw earlier?

Review



- Compute intensity (arithmetic ops/memory reference) on a per-thread basis impacts performance
- Avoiding SMEM bank conflicts is critical
- Visual Profiler is helpful to identify performance issues

Final Wrap-up



- **Optimizations we looked at**
 - Coalesced global memory accesses
 - Shared memory usage
 - `__syncthreads()`
 - Padding to avoid bank conflicts
 - Appropriate arithmetic intensity
- **Things we didn't examine**
 - Optimizations of host->device data transfer
 - Overlap of communication/computation
 - Texture memory usage
 - Multi-GPU programming

Questions?



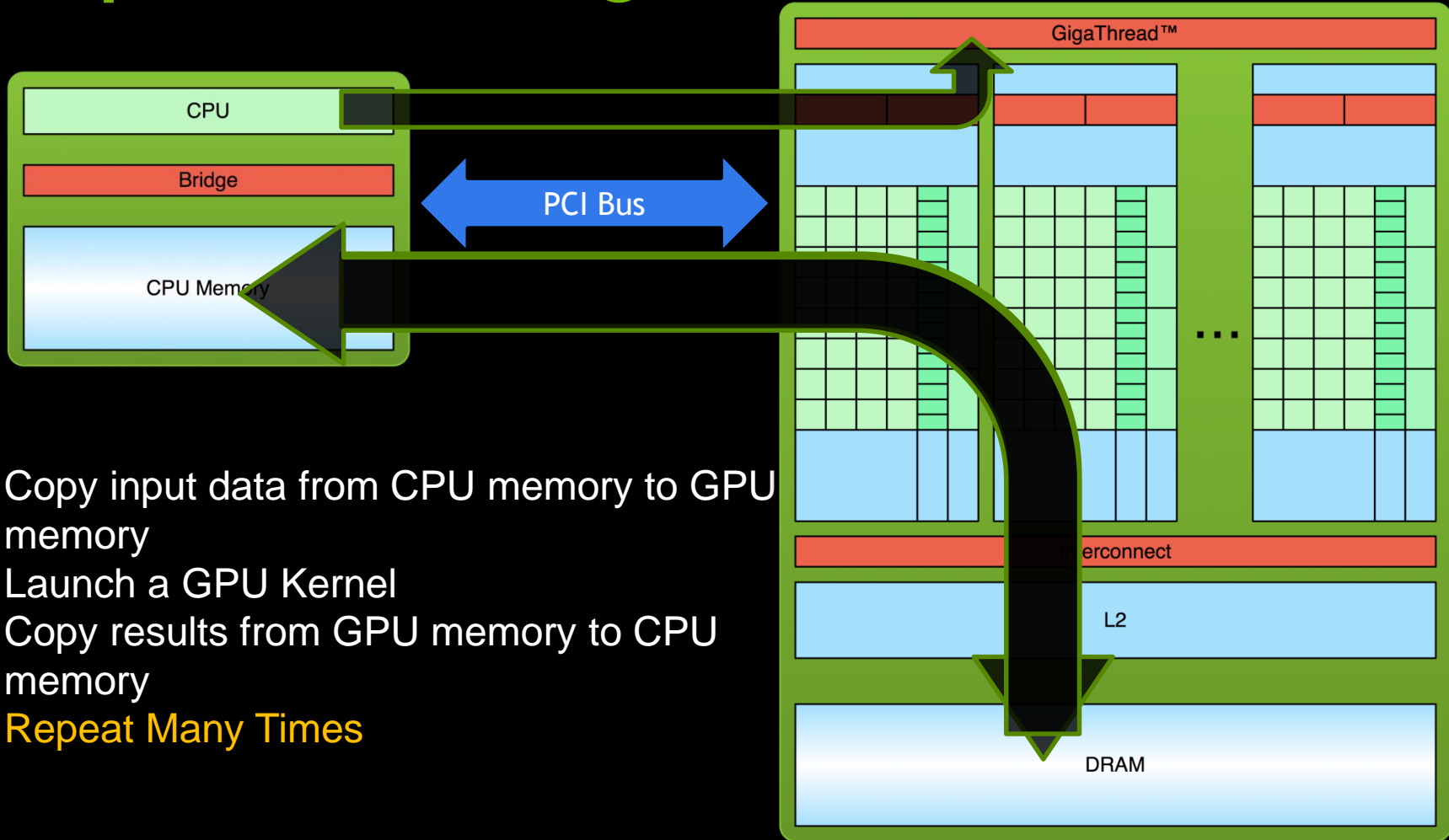
- Let me know if you questions in future
 - jbentz@nvidia.com

LOGIN

- ▶ `login.deepthought2.umd.edu`
- ▶ `module load cuda/6.5.14`

*CUDA STREAMS (slides adapted from
Justin Luitjens, NVIDIA)*

Simple Processing Flow

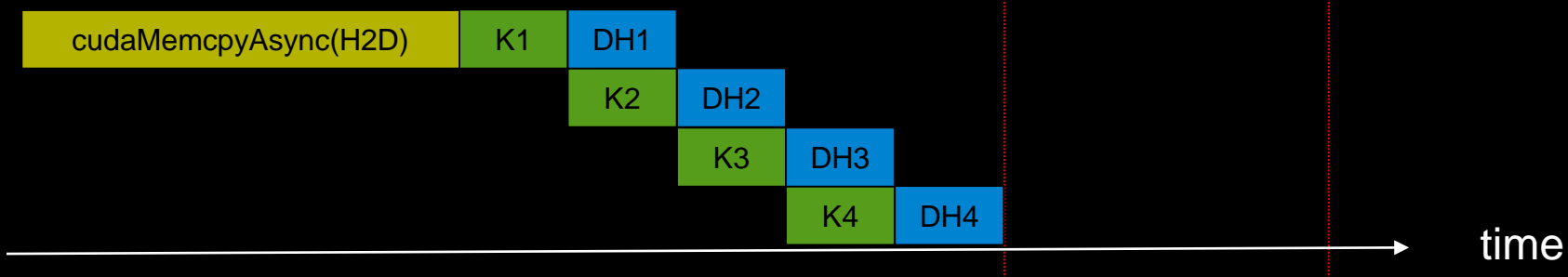


CONCURRENCY THROUGH PIPELINING

- Serial



- Concurrent- overlap kernel and D2H copy

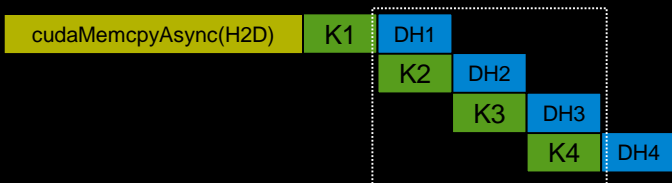


CONCURRENCY THROUGH PIPELINING

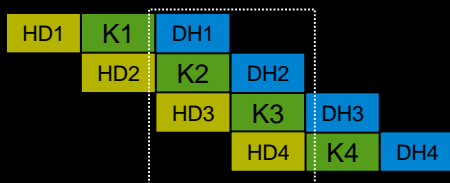
- Serial (1x)



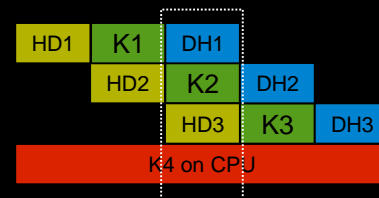
- 2-way concurrency (up to 2x)



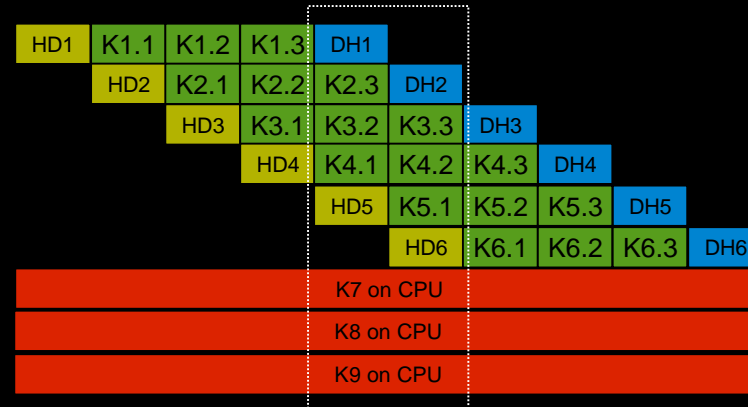
- 3-way concurrency (up to 3x)



- 4-way concurrency (3x+)



- 4+ way concurrency

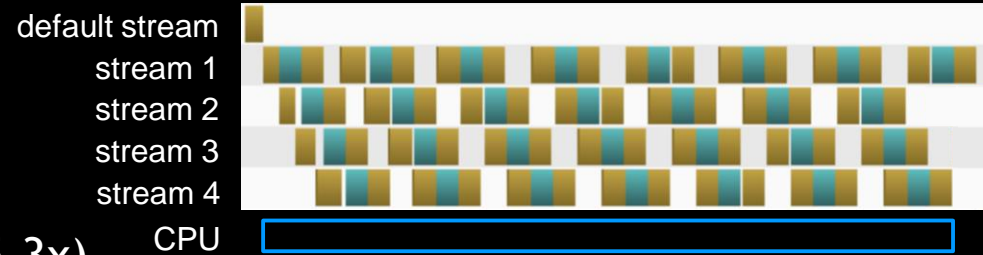


EXAMPLE - TILED DGEMM

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

DGEMM: m=n=16384, k=1408

Nvidia Visual Profiler (nvvp)

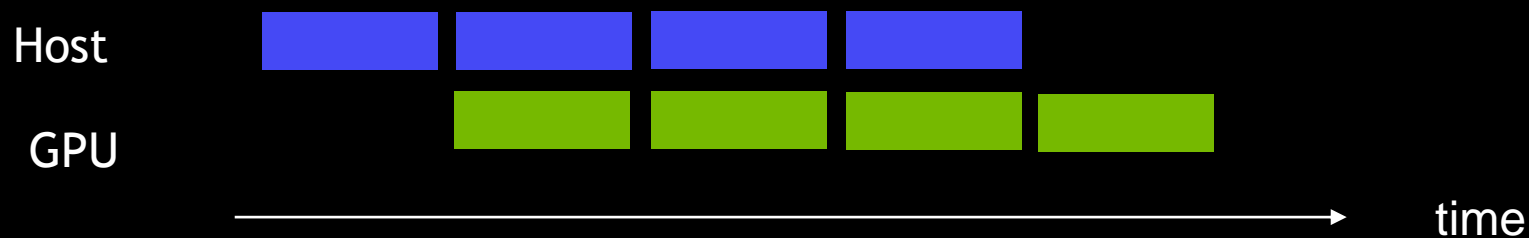




Enabling Concurrency with Streams

SYNCHRONICITY IN CUDA

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



CUDA STREAMS

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

MANAGING STREAMS

- `cudaStream_t stream;`
 - Declares a stream handle
- `cudaStreamCreate(&stream);`
 - Allocates a stream
- `cudaStreamDestroy(stream);`
 - Deallocates a stream
 - Synchronizes host until work in stream has completed

PLACING WORK INTO A STREAM

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

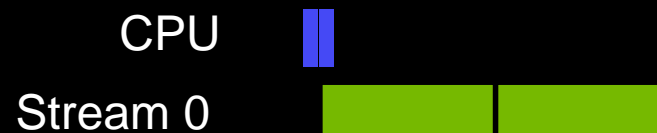
DEFAULT STREAM

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

KERNEL CONCURRENCY

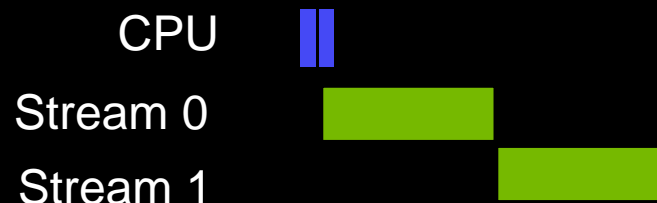
- Assume foo only utilizes 50% of the GPU
- Default stream

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



- Default & user streams

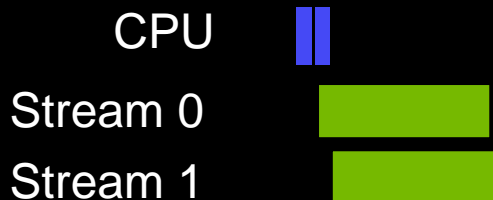
```
cudaStream_t stream1;  
cudaStreamCreate (&stream1);  
foo<<<blocks, threads>>> ();  
foo<<<blocks, threads, 0, stream1>>> ();  
cudaStreamDestroy (stream1);
```



KERNEL CONCURRENCY

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

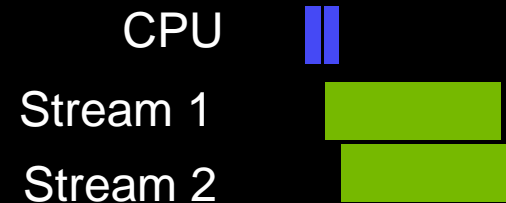
```
cudaStream_t stream1;  
cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking);  
foo<<<blocks, threads>>>();  
foo<<<blocks, threads, 0, stream1>>>();  
cudaStreamDestroy(stream1);
```



KERNEL CONCURRENCY

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

```
cudaStream_t stream1, stream2;  
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
foo<<<blocks, threads, 0, stream1>>>();  
foo<<<blocks, threads, 0, stream2>>>();  
cudaStreamDestroy(stream1);  
cudaStreamDestroy(stream2);
```



REVIEW

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



Concurrent Memory Copies

CONCURRENT MEMORY COPIES

- First we must review CUDA memory

THREE TYPES OF MEMORY

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

ALLOCATING PINNED MEMORY

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

CONCURRENT MEMORY COPIES

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

PAGED MEMORY EXAMPLE

```
int *h_ptr, *d_ptr;
```

```
h_ptr=malloc(bytes);
```

```
cudaMalloc(&d_ptr,bytes);
```

```
cudaMemcpy(d_ptr,h_ptr,bytes,cudaMemcpyHostToDevice);
```

```
free(h_ptr);
```

```
cudaFree(d_ptr);
```

PINNED MEMORY: EXAMPLE 1

```
int *h_ptr, *d_ptr;
```

```
cudaMallocHost (&h_ptr, bytes) ;
```

```
cudaMalloc (&d_ptr, bytes) ;
```

```
cudaMemcpy (d_ptr, h_ptr, bytes, cudaMemcpyHostToDevice) ;
```

```
cudaFreeHost (h_ptr) ;
```

```
cudaFree (d_ptr) ;
```

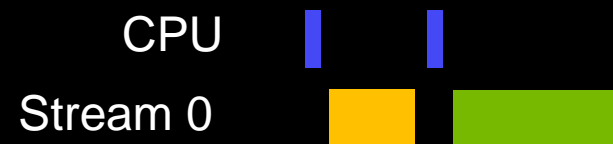
PINNED MEMORY: EXAMPLE 2

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

CONCURRENCY EXAMPLES

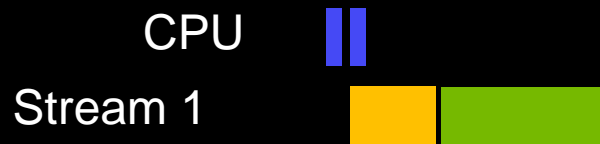
Synchronous

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



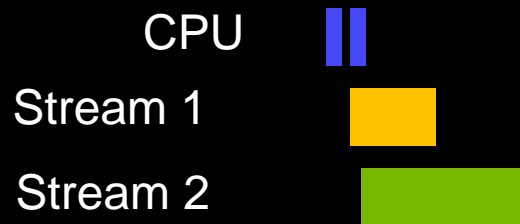
Asynchronous Same Stream

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



Asynchronous Different Streams

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



REVIEW

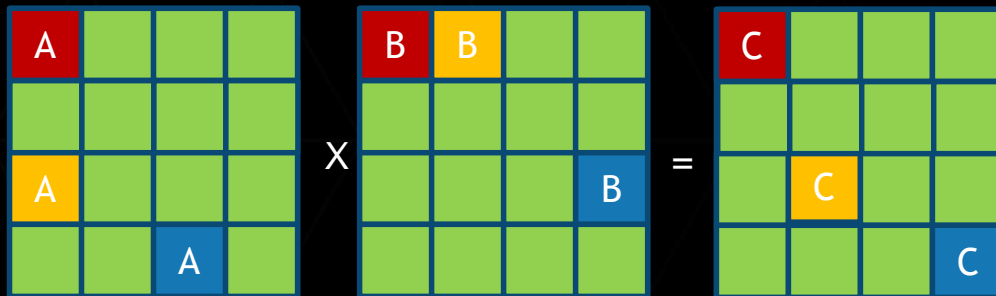
- Memory copies can execute concurrently if (and only if)
 - The memory copy is in a different non-default stream
 - The copy uses pinned memory on the host
 - The asynchronous API is called
 - There isn't another memory copy occurring in the same direction at the same time.



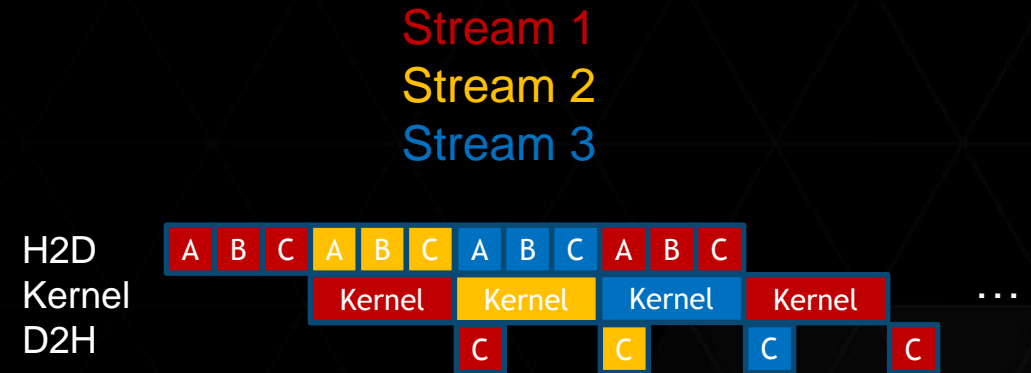
Tiled Matrix Multiply with Streams

TILED MATRIX MULTIPLY

- ▶ Handle arbitrarily large data
 - ▶ Full matrix never on the device



- ▶ GPU running at full capacity
 - ▶ Communication cost is hidden due to overlap



ALGORITHM OUTLINE

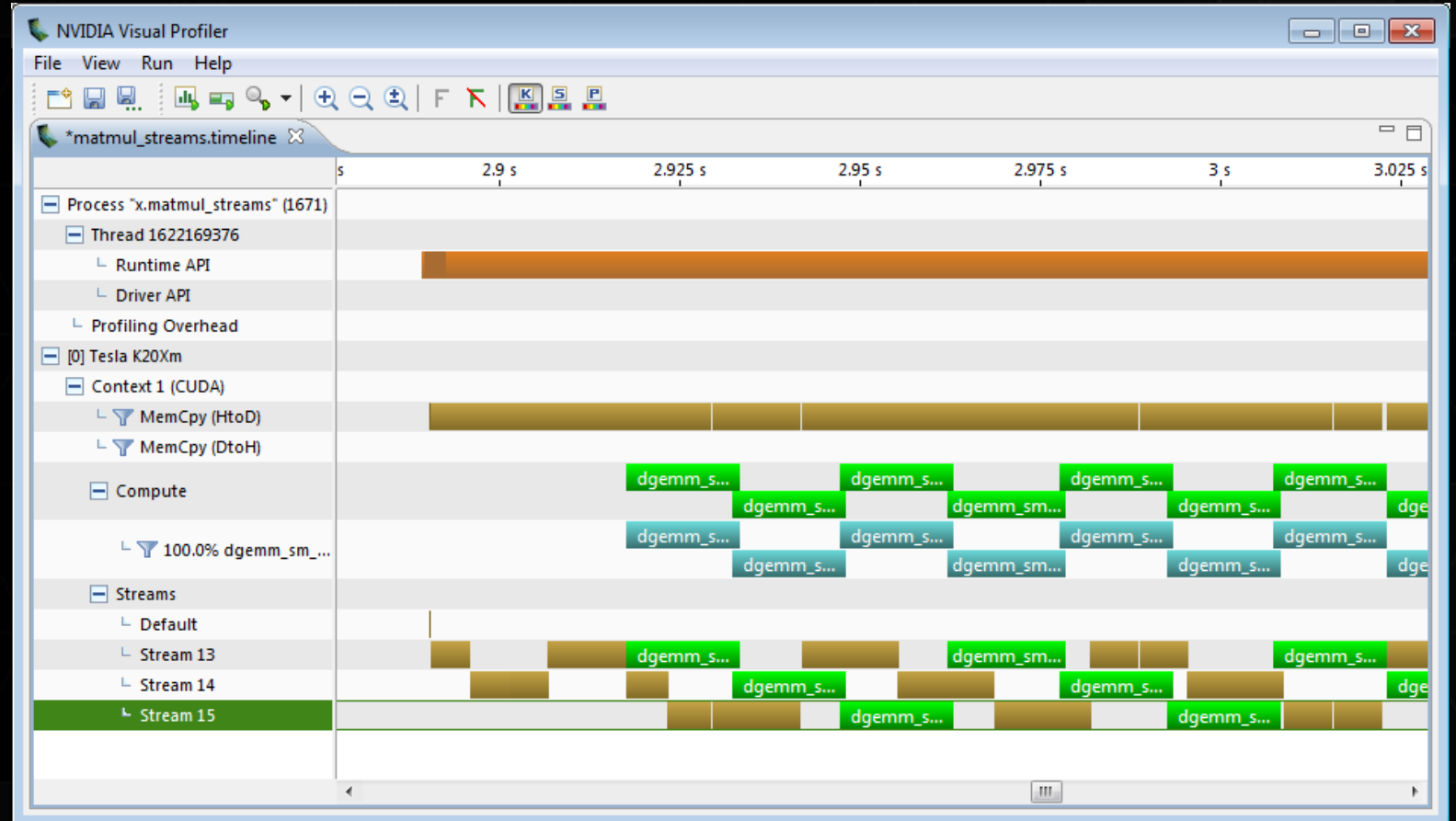
- ▶ Copy tiles of A/B/C into pinned memory buffers
- ▶ Loop over tiles of C
 - ▶ Copy tile of C to device
 - ▶ loop over K
 - ▶ Copy tiles of A and B to device
 - ▶ Call cublasDgemm and accumulate to tile of C
 - ▶ Copy C back to host

matmul_streams/kernel.cu

- ▶ Finish the kernel by inserting appropriate streams/cublas API calls
 - ▶ Use cheatsheet and/or docs.nvidia.com for syntax
- ▶ Answers (and perf) compared against single call of cublas
- ▶ Once you get correct answers try to find best combo of TILESIZE, SIZE and NUM_STREAMS
 - ▶ How much better perf is the streams code?
- ▶ If time permits, profile the app to ensure overlap
- ▶ `CUDA_LAUNCH_BLOCKING=1` to enforce synchronous behavior

NVVP PROFILE

- ▶ K20X/CUDA6.5
 - ▶ N=8192
 - ▶ TILE=2048
 - ▶ Streams=3
 - ▶ 1061 GF
- ▶ Single dgemm
 - ▶ 548 GF



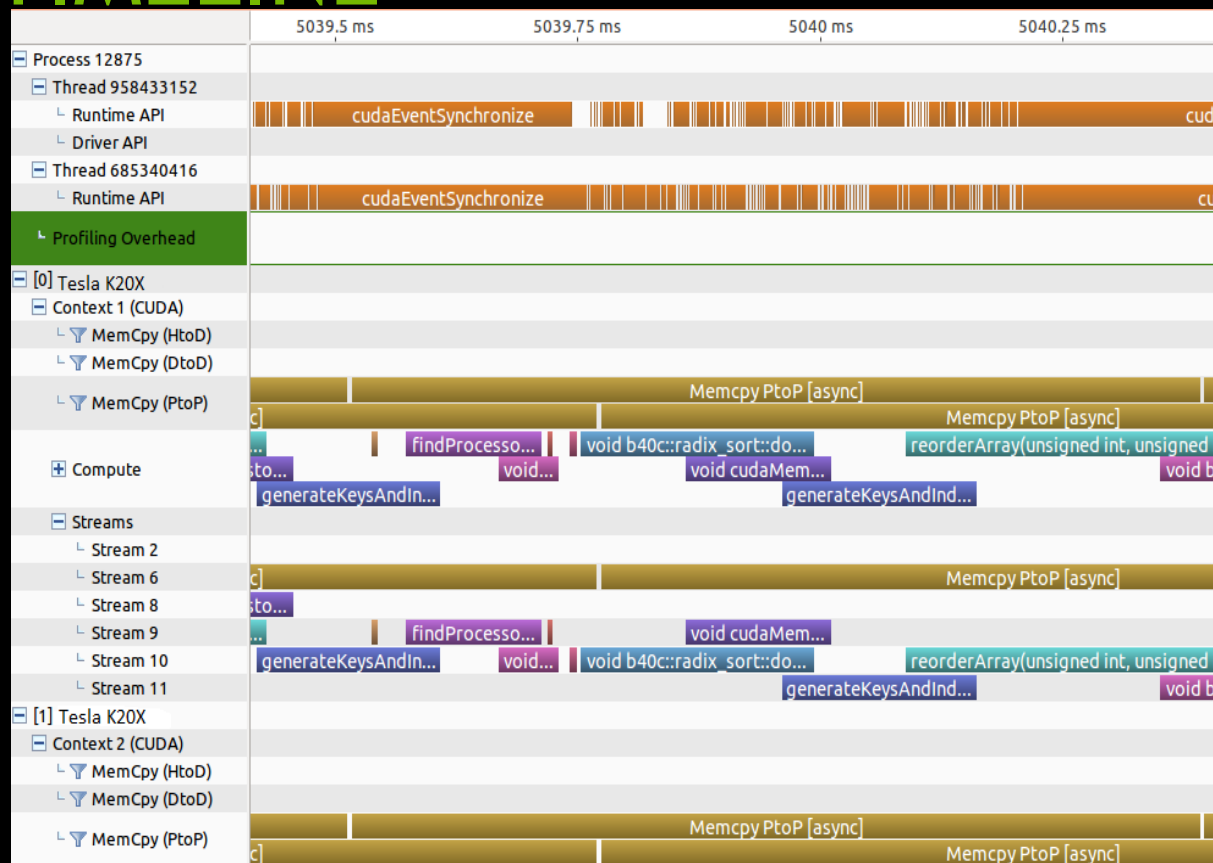
Streaming Performance

PROFILING TOOLS

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

NVVP PROFILER TIMELINE

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

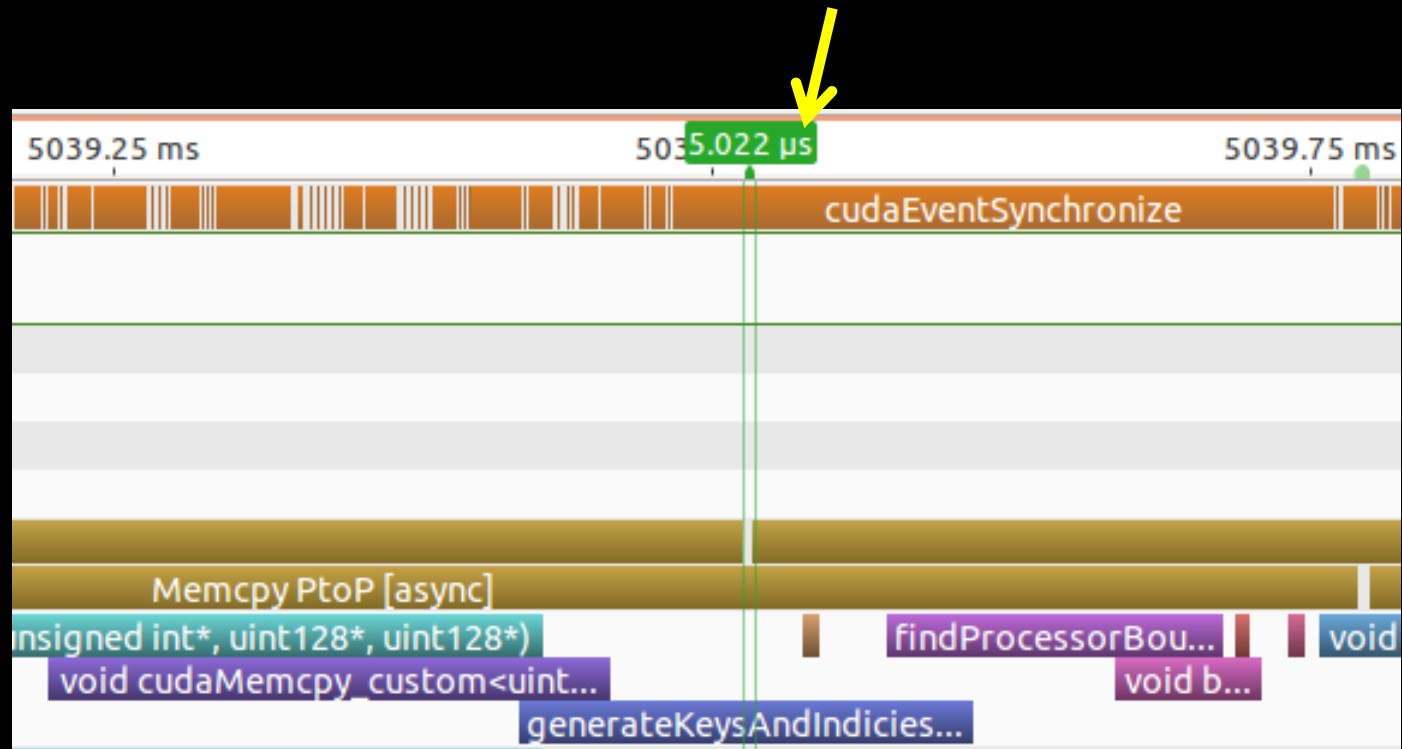


time

OPTIMAL TIMELINE

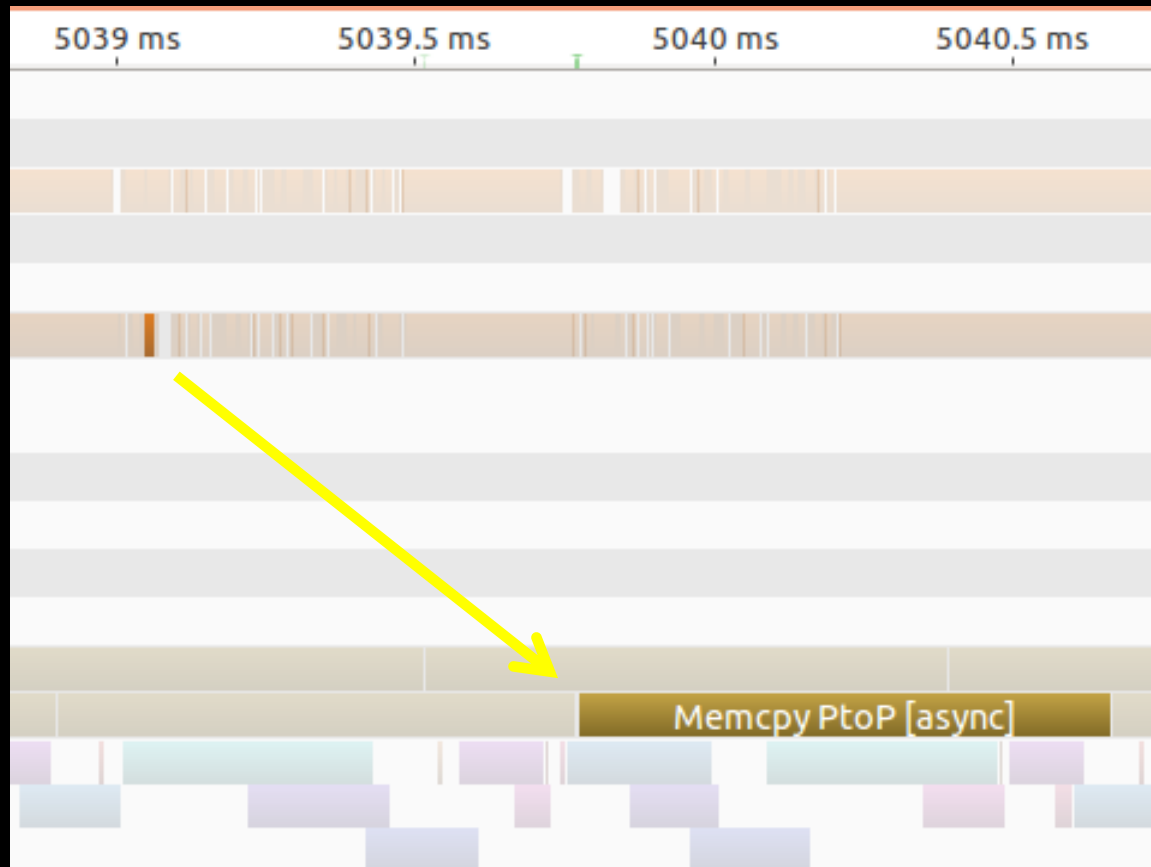
Less than 10 us idle time between successive operations

Concurrent Operations



OPTIMAL TIMELINE

Host is running ahead of the device >30 us



GPU TECHNOLOGY CONFERENCE

March 17-20, 2015 | Silicon Valley
www.gputechconf.com #GTC15



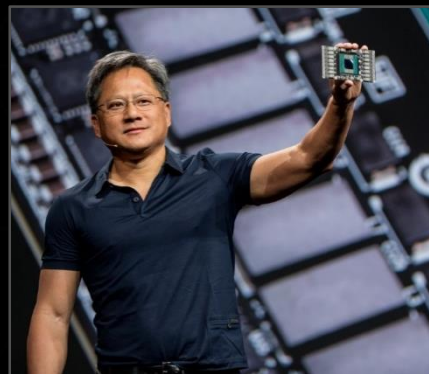
CONNECT

Connect with experts from NVIDIA and other organizations across a wide range of fields



LEARN

Get key learnings and hands-on training in the 400+ sessions and 150+ research posters



DISCOVER

Discover the latest technologies shaping the GPU ecosystem



INNOVATE

Hear about disruptive innovations as early-stage start-ups present their work

4 Days | 3400+ Attendees | 400+ Sessions | 150+ Research Posters
40+ Countries | 180+ Press & Analytics | 100+ Exhibitors