Streams

Streams

- Task parallelism (found in multithread CPU apps)
- Not as general as they are on CPUs
- Page-locked Host memory
- Single stream
- Multiple streams
- GPU work scheduling

Page-locked Host Memory

- Memory allocation:
  - In C: `malloc()`
  - In the GPU: `cudaMalloc()`
  - In the Host using CUDA: `cudaHostAlloc()`
- If we want to keep the page in the physical memory (not on the disk): use `cudaHostAlloc()`
- The speed is bounded by the slowest PCIE
- Warning: we run out of memory faster!

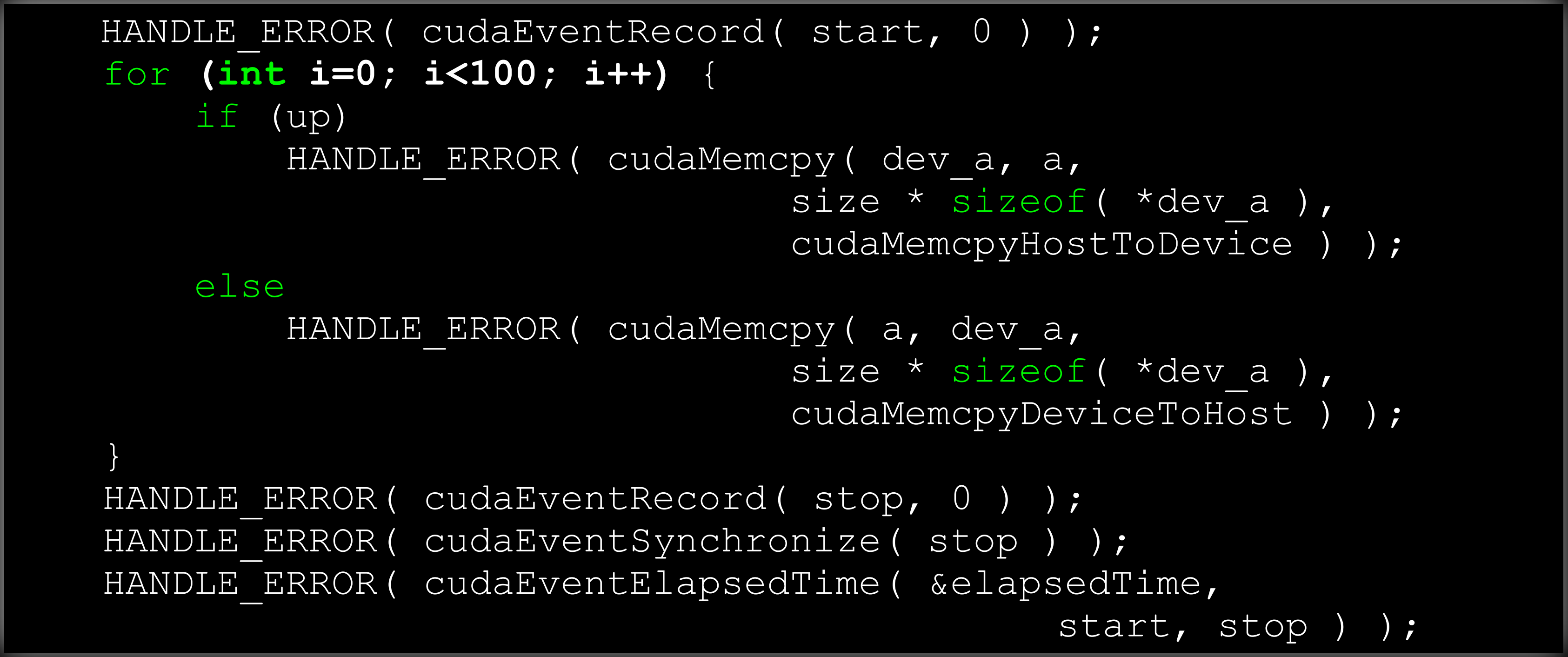
Page-locked Host Memory

Allocation of host and GPU buffers

```
float cuda_malloc_test( int size, bool up ) {
    cudaEvent_t     start, stop;
    int             *a, *dev_a;
    float           elapsedTime;

    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    a = (int*)malloc( size * sizeof( *a ) );
    HANDLE_NULL( a );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                              size * sizeof( *dev_a ) ) );
```

Page-locked Host Memory

- 100 copies specified by “up”



```
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
for (int i=0; i<100; i++) {
    if (up)
        HANDLE_ERROR( cudaMemcpy( dev_a, a,
                              size * sizeof( *dev_a ),
                              cudaMemcpyHostToDevice ) );
    else
        HANDLE_ERROR( cudaMemcpy( a, dev_a,
                              size * sizeof( *dev_a ),
                              cudaMemcpyDeviceToHost ) );
}
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                          start, stop ) );
```

Page-locked Host Memory

- After the 100 copies, we clean up the host and GPU buffers + destroy timing events

```
free( a );
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );

    return elapsedTime;
}
```

Page-locked Host Memory

Here is the pinned memory version:

```
float cuda_host_alloc_test( int size, bool up ) {
    cudaEvent_t     start, stop;
    int             *a, *dev_a;
    float           elapsedTime;

    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    HANDLE_ERROR( cudaHostAlloc( (void**)&a,
                                 size * sizeof( *a ),
                                 cudaHostAllocDefault ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                              size * sizeof( *dev_a ) ) );
```

Page-locked Host Memory

```
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );
    for (int i=0; i<100; i++) {
        if (up)
            HANDLE_ERROR( cudaMemcpy( dev_a, a,
                                  size * sizeof( *a ),
                                  cudaMemcpyHostToDevice ) );
        else
             HANDLE_ERROR( cudaMemcpy( a, dev_a,
                                  size * sizeof( *a ),
                                  cudaMemcpyDeviceToHost ) );
    }
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                        start, stop ) );

    HANDLE_ERROR( cudaFreeHost( a ) );
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );

    return elapsedTime;
}
```

Page-locked Host Memory

Body of main:

```
#include "../common/book.h"
#define SIZE    (64*1024*1024)

int main( void ) {
    float           elapsedTime;
    float           MB = (float)100*SIZE*sizeof(int)/1024/1024;


    // try it with cudaMalloc
    elapsedTime = cuda_malloc_test( SIZE, true );
    printf( "Time using cudaMalloc:  %3.1f ms\n",
            elapsedTime );
    printf( "\tMB/s during copy up:  %3.1f\n",
            MB/(elapsedTime/1000) );
```

Page-locked Host Memory

Test of the performance

```
    elapsedTime = cuda_malloc_test( SIZE, false );
    printf( "Time using cudaMalloc:  %3.1f ms\n",
            elapsedTime );
    printf( "\tMB/s during copy down:  %3.1f\n",
            MB/(elapsedTime/1000) );
```

Page-locked Host Memory

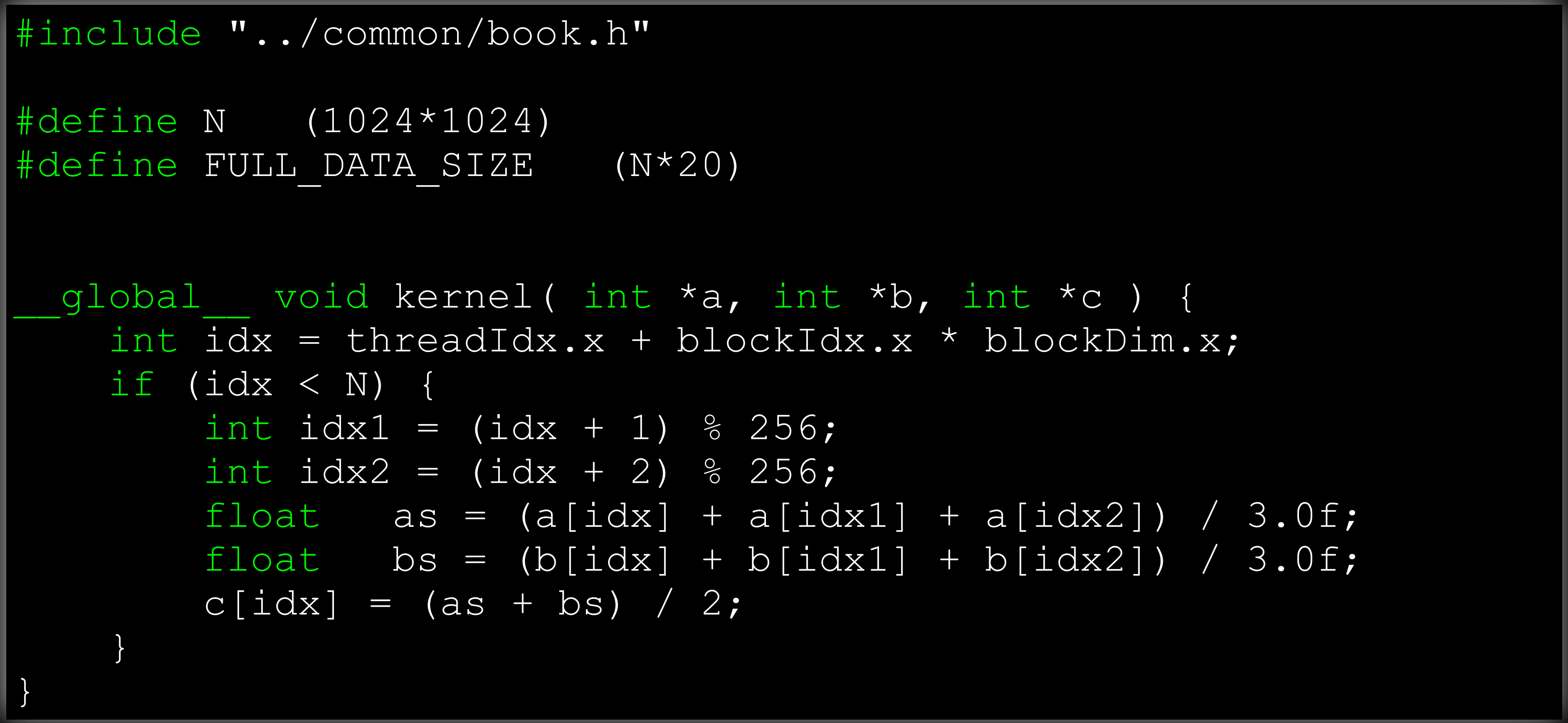
Same set of steps for the performance of `cudaHostAlloc()`

```
    elapsedTime = cuda_host_alloc_test( SIZE, true );
    printf( "Time using cudaHostAlloc:  %3.1f ms\n",
            elapsedTime );
    printf( "\tMB/s during copy up:  %3.1f\n",
            MB/(elapsedTime/1000) );

    elapsedTime = cuda_host_alloc_test( SIZE, false );
    printf( "Time using cudaHostAlloc:  %3.1f ms\n",
            elapsedTime );
    printf( "\tMB/s during copy down:  %3.1f\n",
            MB/(elapsedTime/1000) );
```

Exercice: What performance do you get on your computer? Compare...

CUDA Streams

- We compute three values in a and b:



```
#include "../common/book.h"

#define N   (1024*1024)
#define FULL_DATA_SIZE   (N*20)


__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float   as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float   bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2;
    }
}
```

CUDA Streams

- Body of `main()`:

```
int main( void ) {
    cudaDeviceProp  prop;
    int whichDevice;
    HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
    HANDLE_ERROR( cudaGetDeviceProperties( &prop,
                                     whichDevice ) );
    if (!prop.deviceOverlap) {
          printf( "Device will not handle overlaps,
          so no speed up from streams\n" );
        return 0;
    }
```

CUDA Streams

We start the timers:

```
cudaEvent_t     start, stop;
float           elapsedTime;

// start the timers
HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );
HANDLE_ERROR( cudaEventRecord( start,0 ) );
```

Creation of the stream:

```
cudaStream_t    stream;

// initialize the stream
HANDLE_ERROR( cudaStreamCreate( &stream ) );
```

Data allocation

```
int *host_a, *host_b, *host_c;
int *dev_a, *dev_b, *dev_c;

// allocate the memory on the GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c,N * sizeof(int) ) );

// allocate host locked memory, used to stream
HANDLE_ERROR( cudaHostAlloc( (void**)&host_a,
                              FULL_DATA_SIZE * sizeof(int),
                              cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_b,
                              FULL_DATA_SIZE * sizeof(int),
                              cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_c,
                              FULL_DATA_SIZE * sizeof(int),
                              cudaHostAllocDefault ) );

for (int i=0; i<FULL_DATA_SIZE; i++) {
     host_a[i] = rand();
     host_b[i] = rand();
}
```

Chuckification

```
// now loop over full data, in bite-sized chunks

for (int i=0; i<FULL_DATA_SIZE; i+= N) {

// copy the locked memory to the device, async

HANDLE_ERROR( cudaMemcpyAsync( dev_a, host_a+i,
              N * sizeof(int),cudaMemcpyHostToDevice,stream ) );
HANDLE_ERROR( cudaMemcpyAsync( dev_b, host_b+i,
              N * sizeof(int),cudaMemcpyHostToDevice,stream ) );

kernel<<<N/256,256,0,stream>>>( dev_a, dev_b, dev_c );

// copy the data from device to locked memory

HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c,
              N * sizeof(int),cudaMemcpyDeviceToHost,stream ) );
}
```

```
   // copy result chunk from locked to full buffer
    HANDLE_ERROR( cudaStreamSynchronize( stream ) );
```

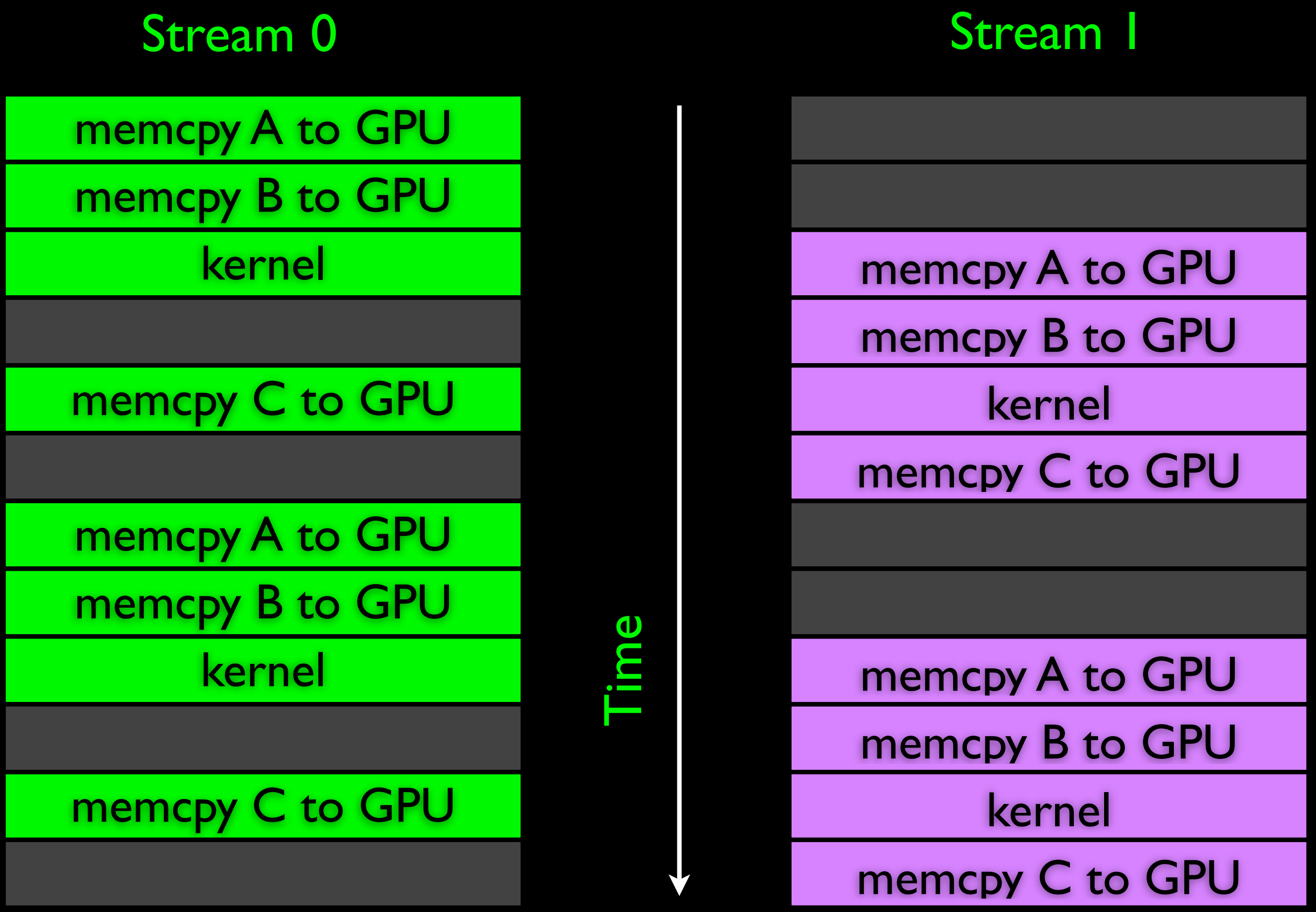
Streams

After the synchronization of streams with the host we can stop the timer:

```
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );

HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Time taken:  %3.1f ms\n", elapsedTime );

// cleanup the streams and memory
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );
HANDLE_ERROR( cudaStreamDestroy( stream ) );

return 0;
}
```

Execution of two streams

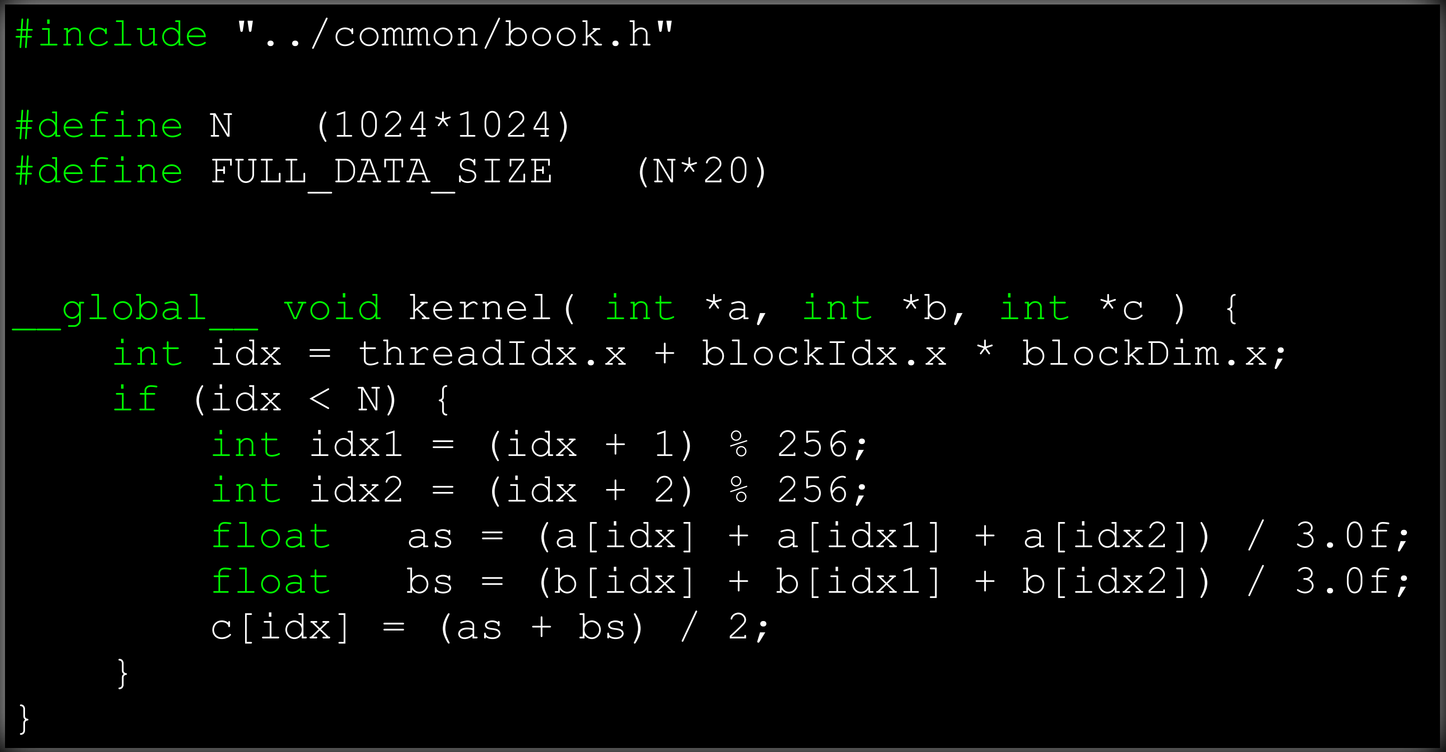
Timeline execution:



Stream 0	Stream 1
memcpy A to GPU	
memcpy B to GPU	
kernel	memcpy A to GPU
	memcpy B to GPU
memcpy C to GPU	kernel
	memcpy C to GPU
memcpy A to GPU	
memcpy B to GPU	
kernel	memcpy A to GPU
	memcpy B to GPU
memcpy C to GPU	kernel
	memcpy C to GPU

Execution of two streams

Despite of acceleration plans, the kernel remains unchanged:



```
#include "../common/book.h"

#define N   (1024*1024)
#define FULL_DATA_SIZE   (N*20)


__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float   as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float   bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2;
    }
}
```

Execution of two streams

```
int main( void ) {
    cudaDeviceProp  prop;
    int whichDevice;
    HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
    if (!prop.deviceOverlap) {
        printf( "Device will not handle overlaps, so no speed up
                from streams\n" );
        return 0;
    }

    cudaEvent_t     start, stop;
    float           elapsedTime;

// start the timers
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start,0 ) );
```

Execution of two streams

```
// initialize the streams

cudaStream_t    stream0, stream1;
HANDLE_ERROR( cudaStreamCreate( &stream0 ) );
HANDLE_ERROR( cudaStreamCreate( &stream1 ) );
```

Execution of two streams

```
int *host_a, *host_b, *host_c;
int *dev_a0, *dev_b0, *dev_c0;
int *dev_a1, *dev_b1, *dev_c1;

// allocate the memory on the GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a0,N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b0,N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c0,N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_a1,N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b1,N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c1,N * sizeof(int) ) );

// allocate host locked memory, used to stream
HANDLE_ERROR( cudaHostAlloc( (void**)&host_a,FULL_DATA_SIZE *
            sizeof(int),cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_b,FULL_DATA_SIZE *
            sizeof(int),cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_c,FULL_DATA_SIZE *
            sizeof(int),cudaHostAllocDefault ) );

for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}
```

Execution of two streams

Stream 0: Queueing of copies of a and b

```
// now loop over full data, in bite-sized chunks

for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {

// copy the locked memory to the device, async

HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i,N * sizeof(int),
              cudaMemcpyHostToDevice,stream0 ) );

HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i,N * sizeof(int),
              cudaMemcpyHostToDevice,stream0 ) );

kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );

// copy the data from device to locked memory

HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0,N * sizeof(int),
              cudaMemcpyDeviceToHost,stream0 ) );
```

Execution of two streams

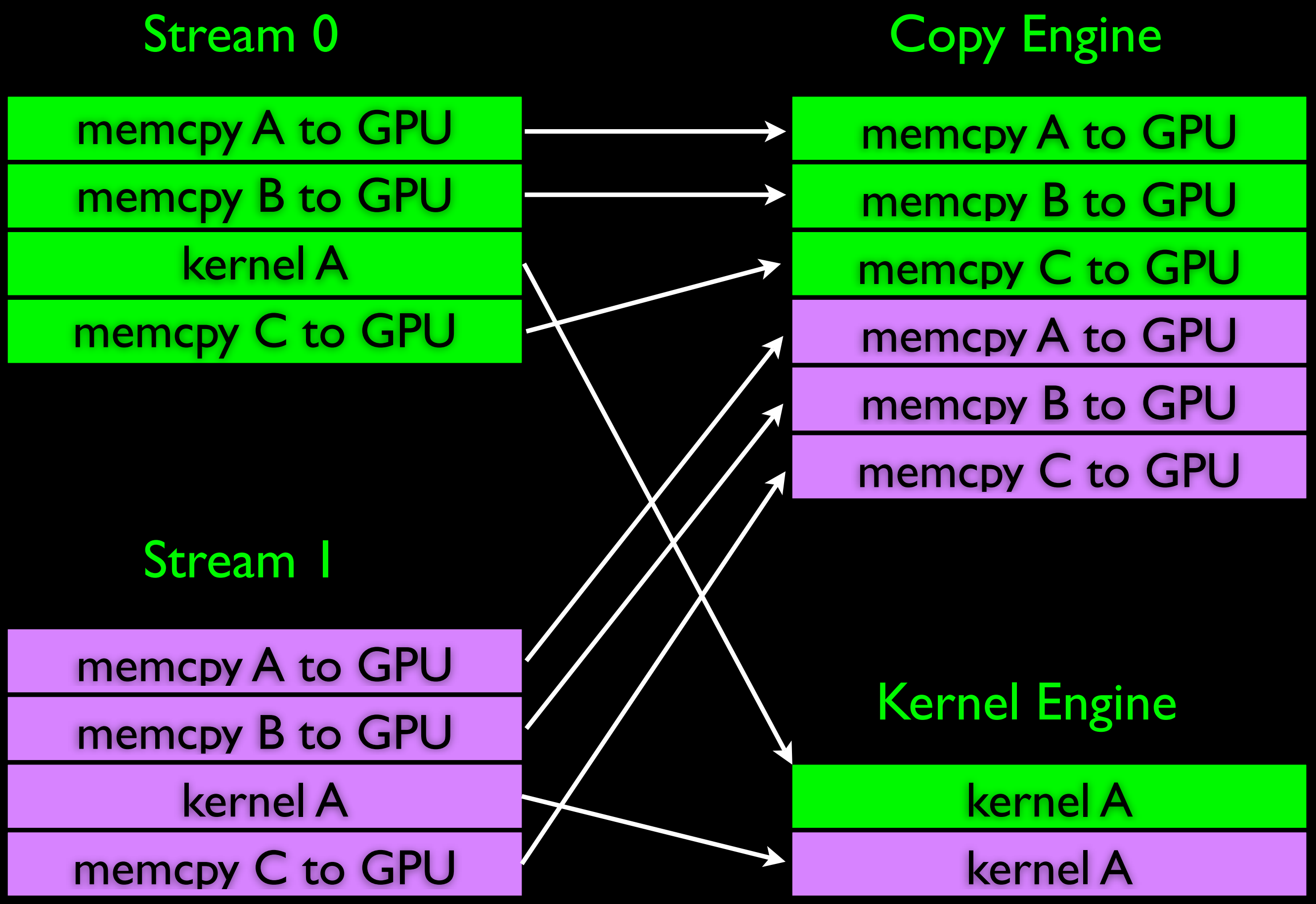
Stream 1: identical queueing operations

```
// copy the locked memory to the device, async
HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N,N * sizeof(int),
              cudaMemcpyHostToDevice,stream1 ) );
HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N,N * sizeof(int),
              cudaMemcpyHostToDevice,stream1 ) );

kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );

// copy the data from device to locked memory
HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1,N * sizeof(int),
              cudaMemcpyDeviceToHost,stream1 ) );
}
```

Execution of two streams

```
HANDLE_ERROR( cudaStreamSynchronize( stream0 ) );
HANDLE_ERROR( cudaStreamSynchronize( stream1 ) );

HANDLE_ERROR( cudaEventRecord( stop, 0 ) );

HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Time taken:  %3.1f ms\n", elapsedTime );

// cleanup the streams and memory
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a0 ) );
HANDLE_ERROR( cudaFree( dev_b0 ) );
HANDLE_ERROR( cudaFree( dev_c0 ) );
HANDLE_ERROR( cudaFree( dev_a1 ) );
HANDLE_ERROR( cudaFree( dev_b1 ) );
HANDLE_ERROR( cudaFree( dev_c1 ) );
HANDLE_ERROR( cudaStreamDestroy( stream0 ) );
HANDLE_ERROR( cudaStreamDestroy( stream1 ) );

return 0;
}
```

Mapping of CUDA streams onto GPU engines



Stream 0

memcpy A to GPU

memcpy B to GPU

kernel A

memcpy C to GPU

Stream 1

memcpy A to GPU

memcpy B to GPU

kernel A

memcpy C to GPU

Copy Engine

memcpy A to GPU

memcpy B to GPU

memcpy C to GPU

memcpy A to GPU

memcpy B to GPU

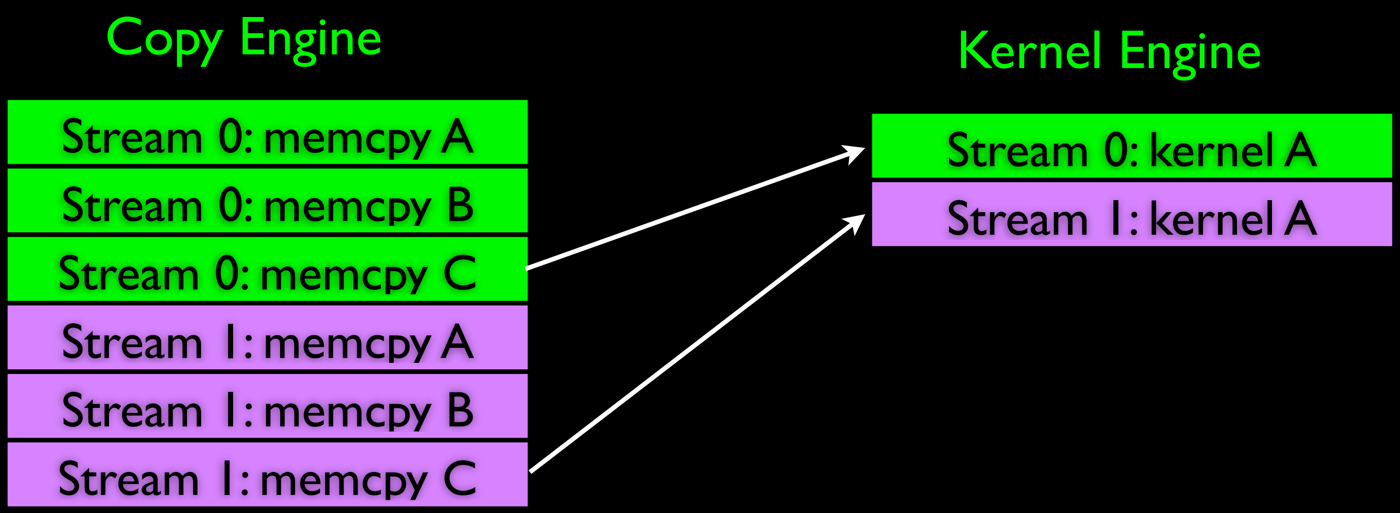
memcpy C to GPU

Kernel Engine

kernel A

kernel A

Mapping of CUDA streams onto GPU engines



Copy Engine

Stream 0: memcpy A

Stream 0: memcpy B

Stream 0: memcpy C

Stream 1: memcpy A

Stream 1: memcpy B

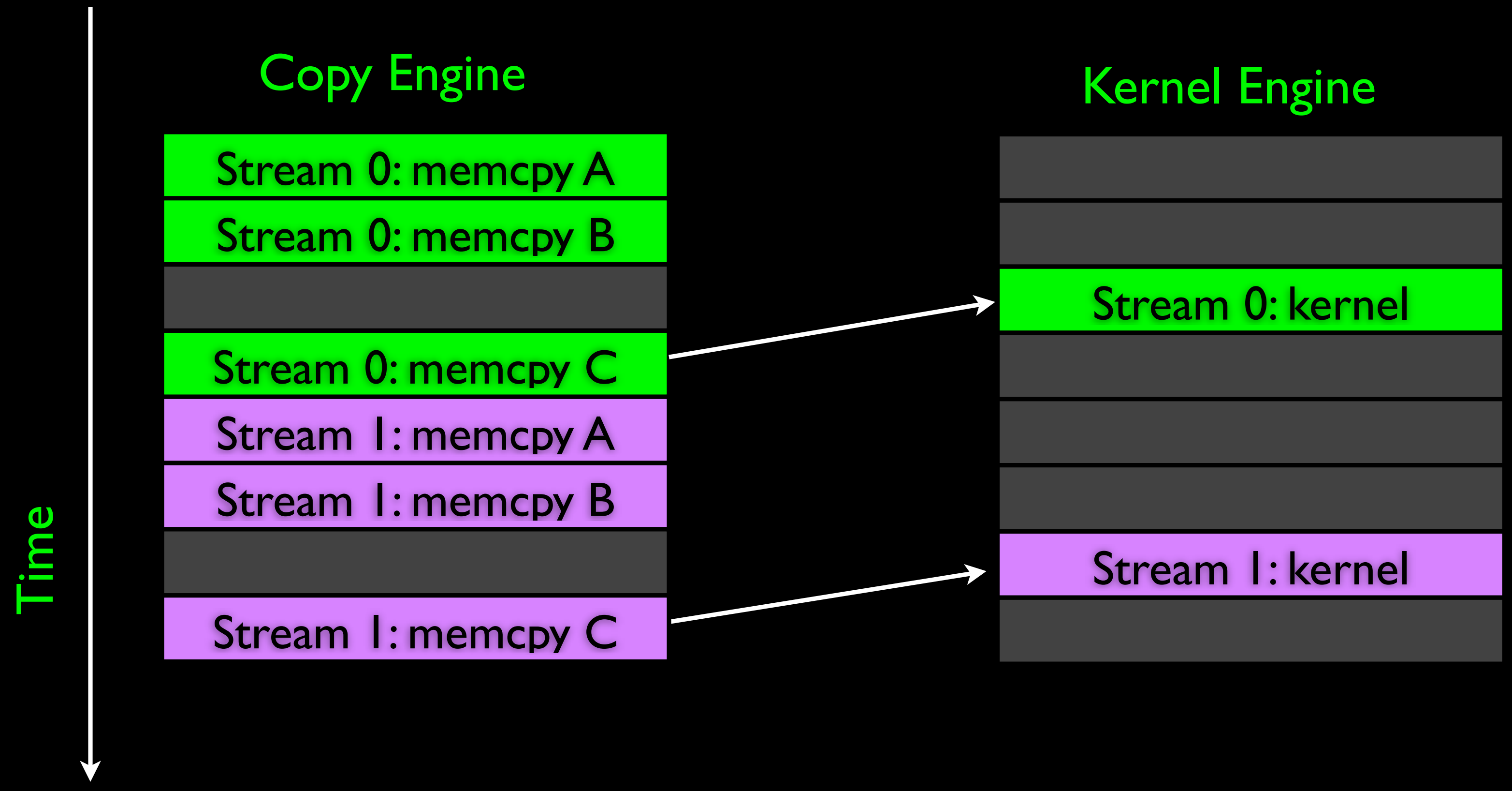
Stream 1: memcpy C

Kernel Engine

Stream 0: kernel A

Stream 1: kernel A

Exucution timeline



Time

Copy Engine

Stream 0: memcpy A

Stream 0: memcpy B

Stream 0: memcpy C

Stream 1: memcpy A

Stream 1: memcpy B

Stream 1: memcpy C

Kernel Engine

Stream 0: kernel

Stream 1: kernel

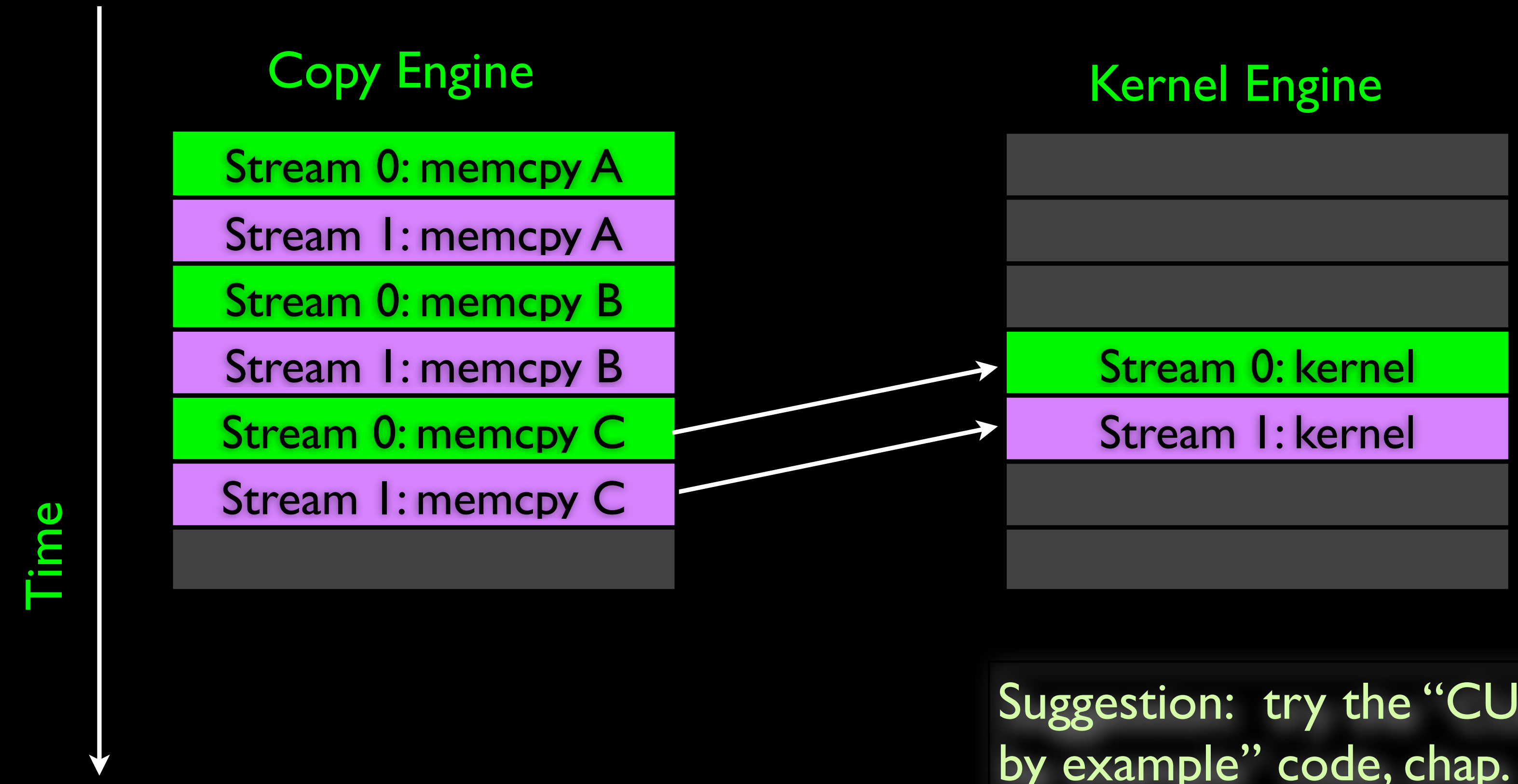
Using Multiple CUDA Streams Effectively

```
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
// enqueue copies of a in stream0 and stream1
     HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i,N * sizeof(int),
                                    cudaMemcpyHostToDevice,
                                    stream0 ) );
     HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N,N * sizeof(int),
                                    cudaMemcpyHostToDevice,
                                    stream1 ) );
// enqueue copies of b in stream0 and stream1
     HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i,N * sizeof(int),
                                    cudaMemcpyHostToDevice,
                                    stream0 ) );
     HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N,N * sizeof(int),
                                    cudaMemcpyHostToDevice,
                                    stream1 ) );
```

Using Multiple CUDA Streams Effectively

```
// enqueue kernels in stream0 and stream1

     kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
     kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );

// enqueue copies of c from device to locked memory

     HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0,
                                    N * sizeof(int),
                                    cudaMemcpyDeviceToHost,
                                    stream0 ) );

     HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1,
                                    N * sizeof(int),
                                    cudaMemcpyDeviceToHost,
                                    stream1 ) );
}
```

Execution timeline of the improved example



Suggestion: try the "CUDA by example" code, chap. 10