

CSE 531

**Heterogeneous Computing and
GPU Programming (Part II)**

Spring 2023

Mahmut Taylan Kandemir

ACK: G. Barlas, X. Tang, L. Mao

Summary: launching a kernel

CPU program
(serial code)

```
==  
==  
==
```

```
cudaMemcpy ( ... )
```

```
==  
==
```

```
Function <<<nb,nt >>>
```

```
==  
==
```

```
cudaMemcpy ( ... )
```

```
==  
==
```

```
_global_ Function ( ... )
```

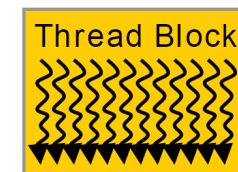
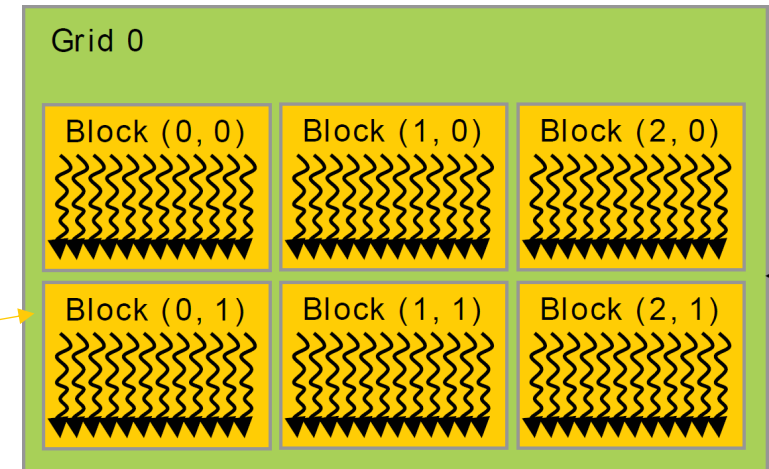
```
==  
==  
==  
==  
==
```

Copy data from CPU
memory to GPU memory

Launch a **kernel** with *nb*
blocks, each with *nt* threads

Copy results from GPU
memory to CPU memory

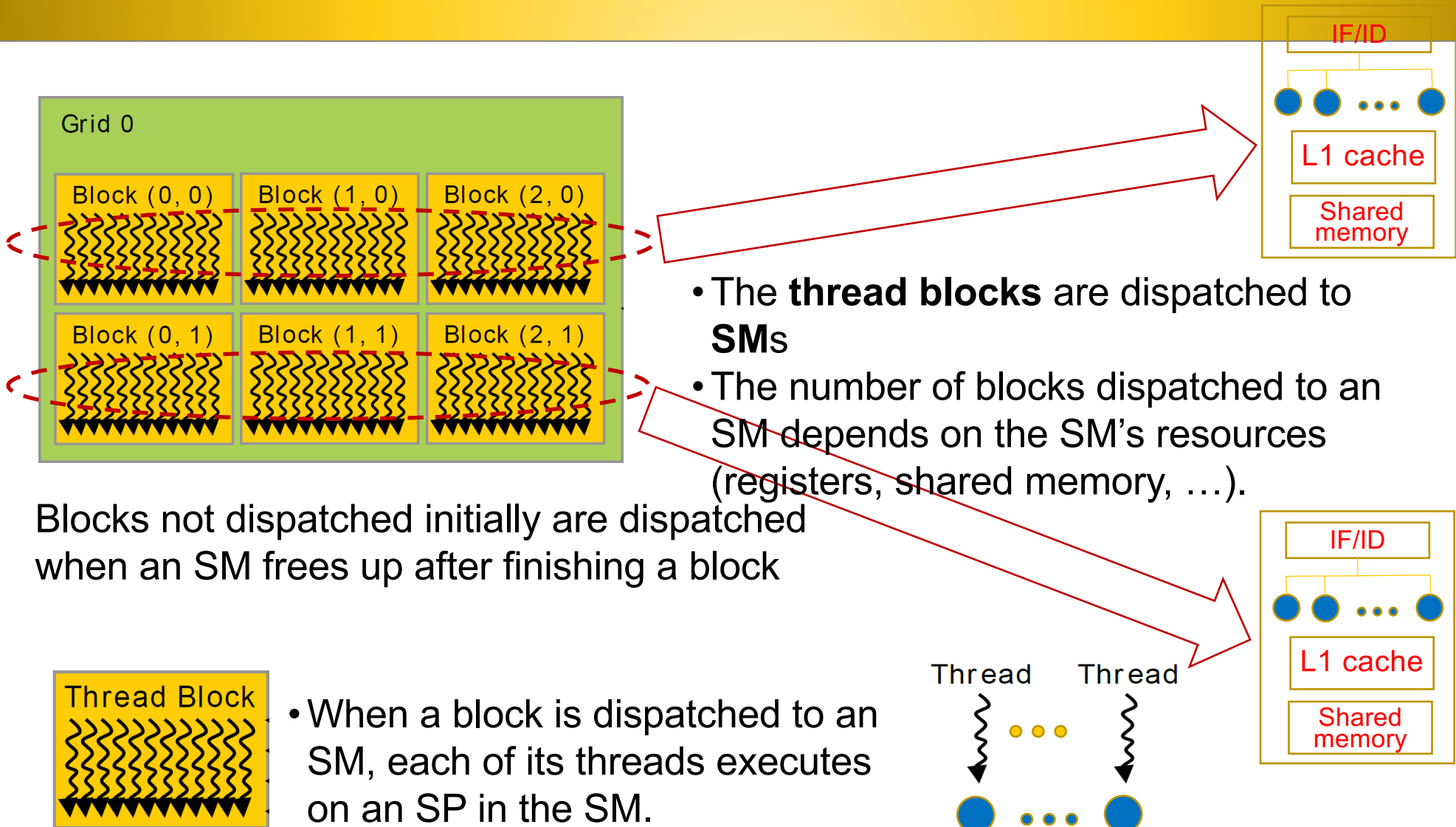
Implementation of kernel
(the function run by each **GPU thread**)



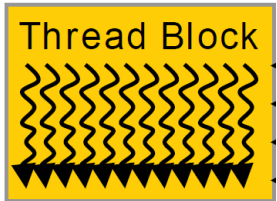
Thread



Summary: execution model



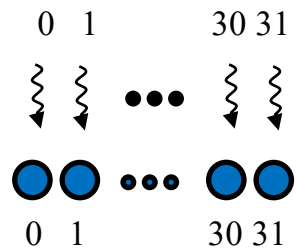
Summary: execution model



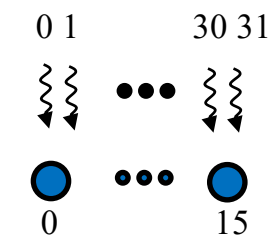
Each block (up to 1K threads) is divided into groups of 32 threads (called **warps**) – empty threads are used as fillers.

A warp executes as a SIMD **vector instruction** on the SM.

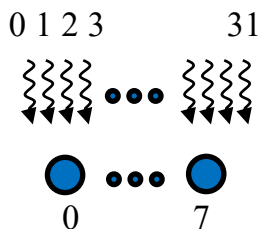
Depending on the number of SPs per SM:



If 32 SP per SM \rightarrow 1 thread of a warp executes on 1 SP
(32 lanes of execution, one thread per lane)



If 16 SP per SM \rightarrow 2 threads are time multiplexed on 1 SP
(16 lanes of execution, 2 threads per lane)

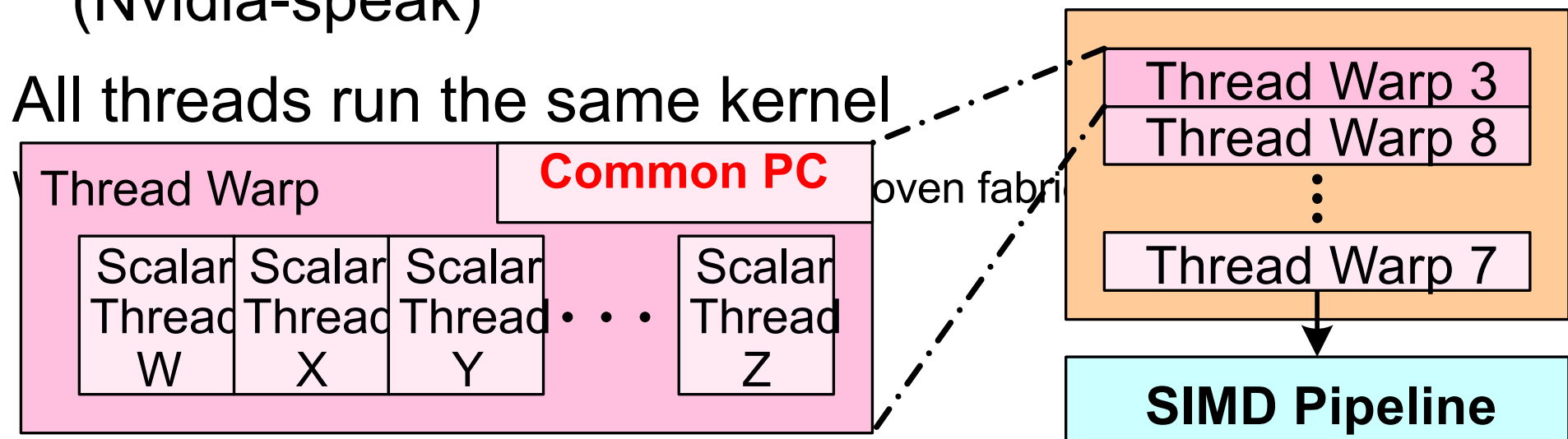


If 8 SP per SM \rightarrow 4 threads are time multiplexed on 1 SP
(8 lanes of execution, 4 threads per lane)

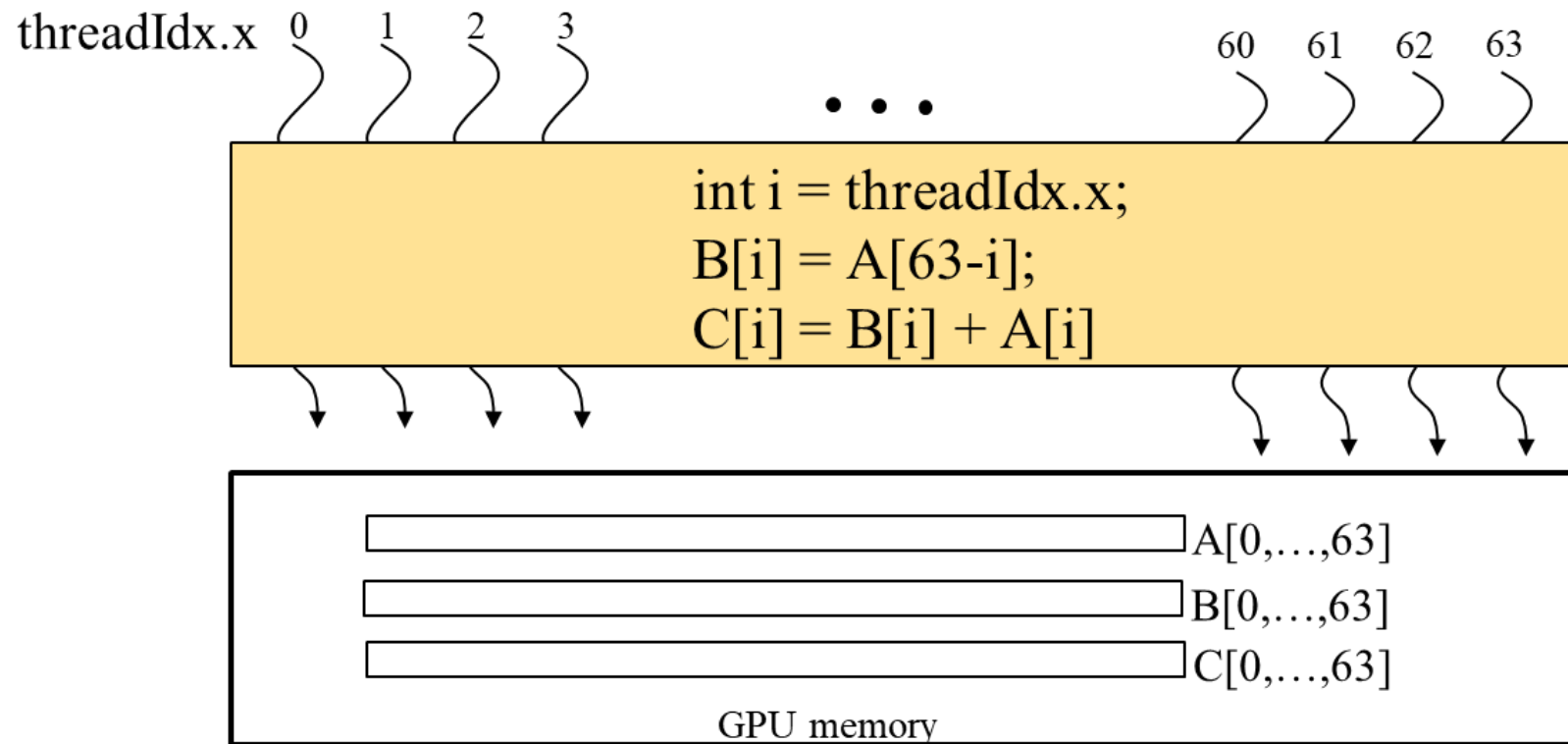
Summary: thread warps and SIMT

Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)

All threads run the same kernel

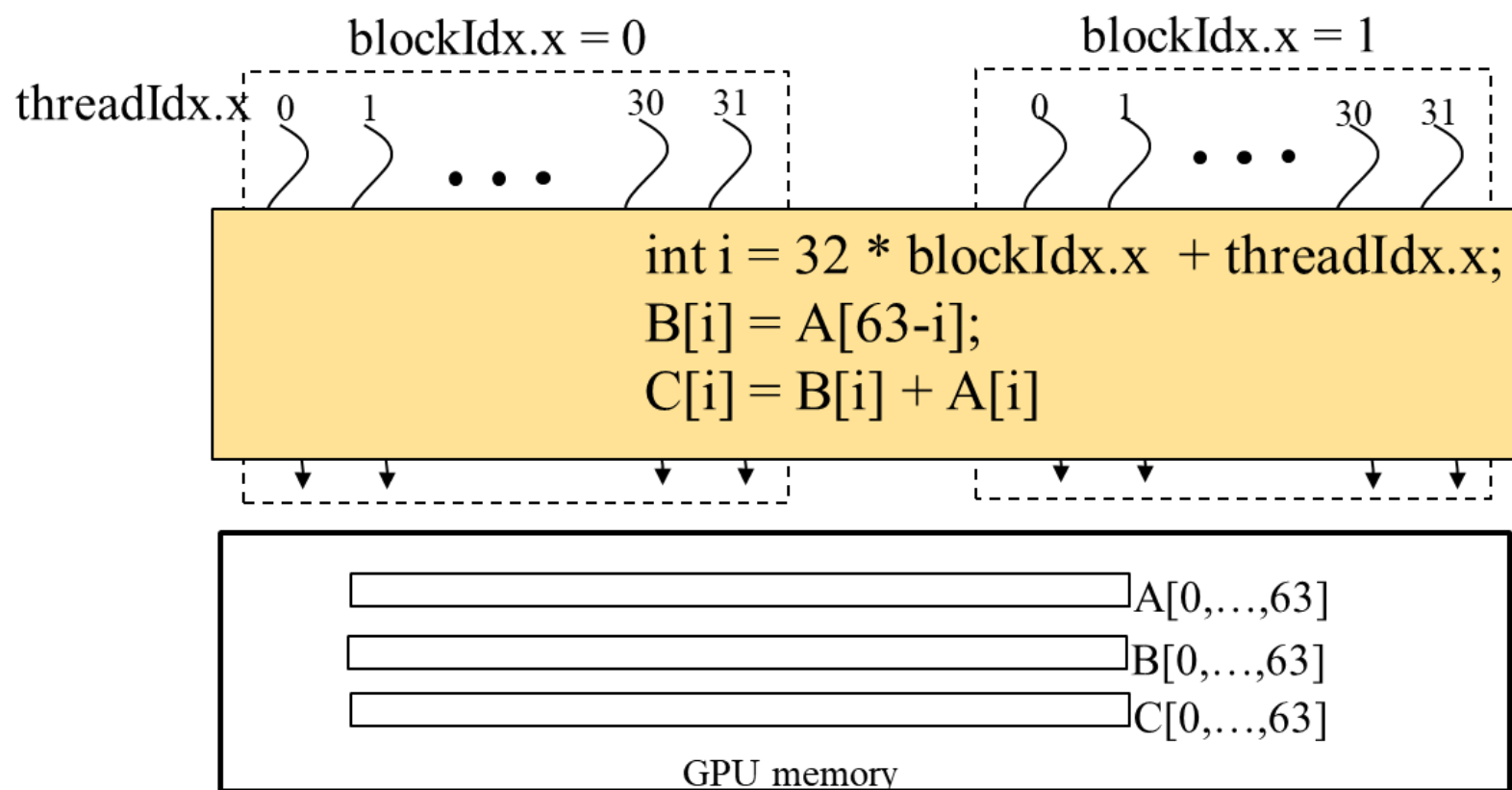


Summary: SIMT execution



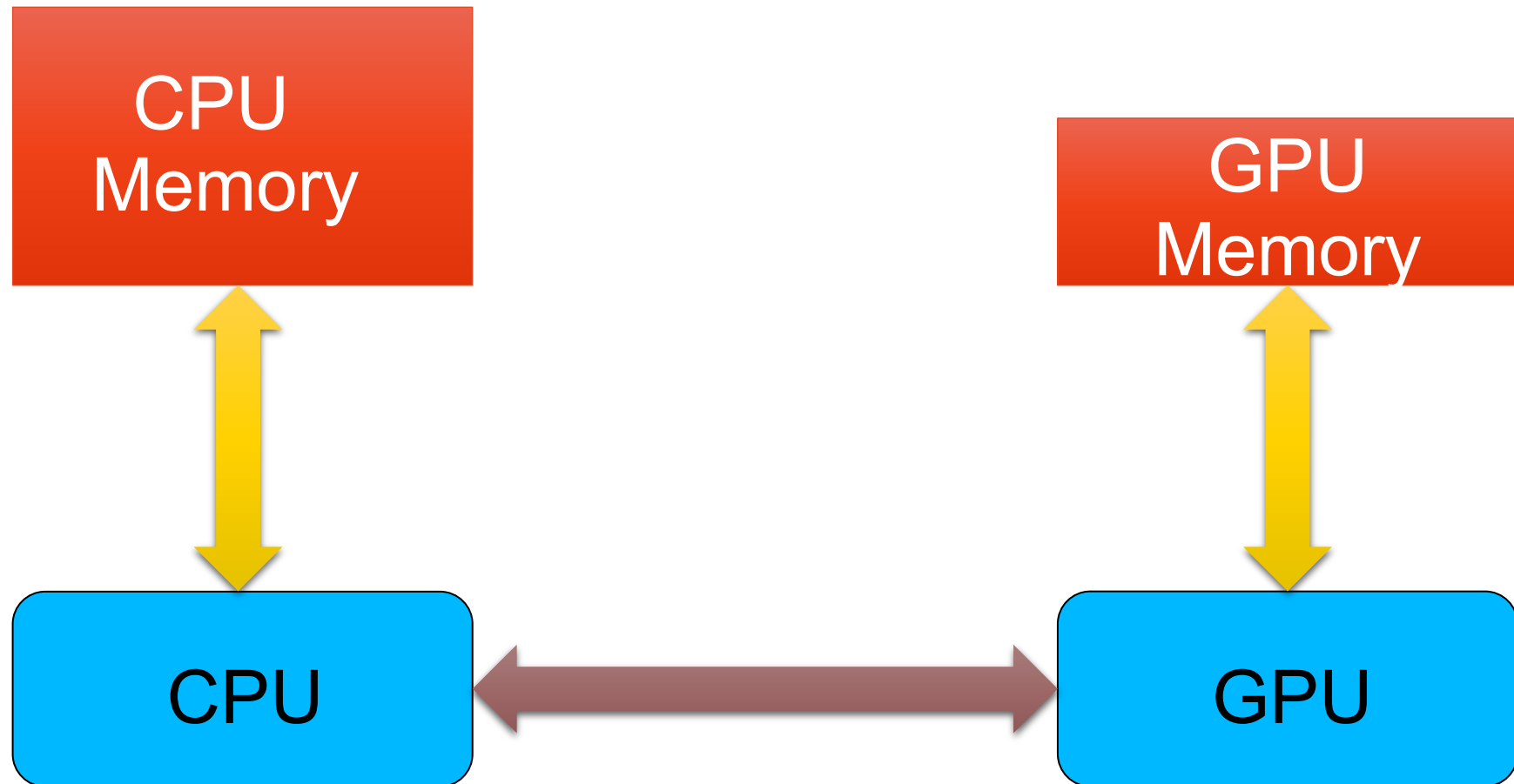
Launched using **Kernel <<<1, 64>>>** : 1 block with 64 threads

Summary: SIMT execution

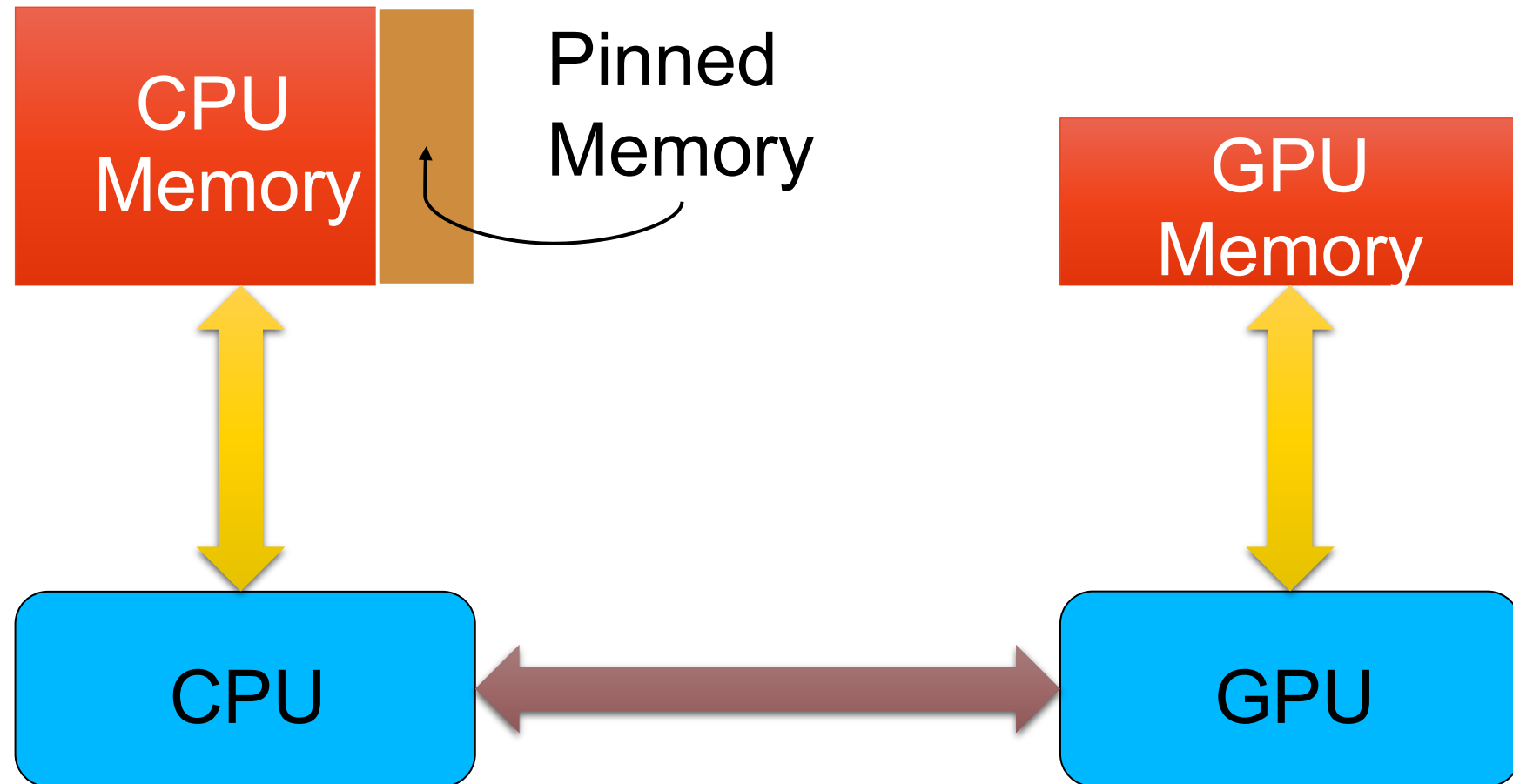


Launched using **Kernel <<<2, 32>>>** : 2 blocks each with 32 threads

Pageable and pinned memory transfer



Pageable and pinned memory transfer



Page-locked memory

- **Page-locked** or **pinned** memory, refers to host memory that cannot be swapped-out as part of the regular virtual memory operations.
- Pinned memory is used to hold critical code and data that cannot be moved out of the main memory, e.g., the OS kernel
- It is also used by DMA to transfer data across the PCIe bus. The CUDA driver has to copy host data to pinned memory before a transfer can occur.
- Placing program data in pinned memory saves extra data transfers but can be detrimental for the efficiency of the host's virtual memory (especially, when it is heavily used)
- Pinned memory can be allocated with:
 - `malloc()` followed by a call to `mlock()`. Deallocation is done in the reverse order, i.e., `munlock()` then `free()`.
 - Or, by calling the `cudaMallocHost()` function. Memory allocated in this fashion has to be deallocated with a call to `cudaFreeHost()`.

Page-locked memory (cont.)

```
cudaError_t cudaMallocHost(  
    void ** ptr,    // Addr. of pointer to pinned  
                    // memory (IN/OUT)  
    size_t size);  // Size in bytes of request (IN)  
  
cudaError_t cudaFreeHost (void * ptr);
```

- CUDA Driver checks, if the memory range is locked or not and then it will use a different code-path. Locked memory is stored in the physical memory (RAM), so device can fetch it w/o help from CPU. Not-locked memory can generate a page fault on access, and it is stored not only in memory (e.g., it can be swapped out); so, driver need to access every page of non-locked memory, copy it into pinned buffer and pass it to DMA.
- The performance gain obtained via pinned memory depends on the size of the data to be transferred.
- The gain can range from a modest 10% to a massive 2.5x.

Zero-copy memory

- **Zero-copy memory** (aka **mapped memory**) is just a term used to convey that no **explicit** memory transfer between the host and the device needs to be initiated.
- Using zero-copy memory, a transfer across the PCIe bus will be initiated by the CUDA run-time upon the first attempt to access a region of memory that is designated as mapped memory, stalling the active kernel while it is taking place.
- In short, zero-copy memory is page-locked memory that can be mapped to the address space of the device.
- This makes program logic simpler. We now have a memory region with **two addresses/pointers**:
 - One for the *host* and one for the *device*.
- NOTE: Zero-copy memory is page-locked memory. For this reason, it is freed with a call to `cudaFreeHost()`.

Zero-copy memory (cont.)

- To allocated zero-copy memory we use:

```
cudaError_t cudaHostAlloc(  
    void ** pHost,          // Addr. of pointer to mapped  
                            // memory (IN/OUT)  
    size_t size,           // Size in bytes of request (IN)  
    unsigned int flags);   // Options for function (IN)
```

where flags should be set to cudaHostAllocMapped.

- To get the **device** pointer for the mapped region we use:

```
cudaError_t cudaHostGetDevicePointer(  
    void ** pDevice,        // Address where the returned dev  
                            // pointer is stored (IN/OUT)  
    void * pHost,          // Address of host pointer (IN)  
    unsigned int flags)    // Currently should be set to 0
```

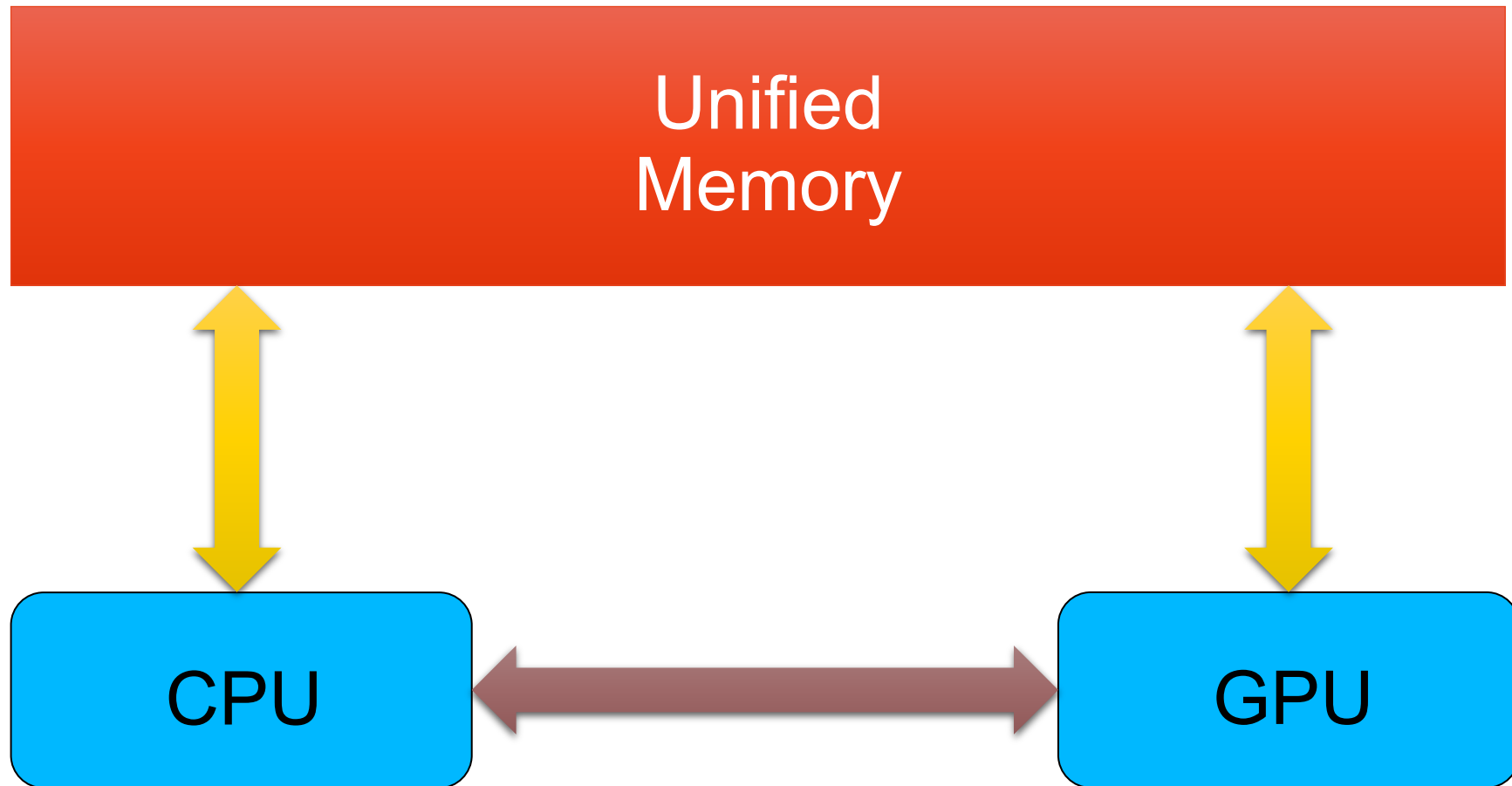
Unified virtual address (UVA)

- **Unified Virtual Address** space is a facility available to devices running on 64-bit hosts.
- UVA *unifies* the address space of host and device which means pointers are unique and the CUDA run-time can determine from the value of a pointer where the memory pointed-to resides.
- With UVA, using mapped memory is as simple as:

```
int *h_data;  
cudaHostAlloc((void **)&h_data, sizeof(int)* DATASIZE, ←  
             cudaHostAllocMapped);  
doSmt<<< gridDim, blkDim >>>(h_data);
```

i.e., we do not need to use `cudaHostGetDevicePointer`.

Unified memory



NOTE: this is only a ***logical*** view (physically, memory are still separate)

Zero-copy and UVA

- Retrieving the result of a GPU computation is still needed when UVA is enabled. Example:

```
int *h_in;  
int *h_out, *d_out;  
.  
.  
.  
cudaHostAlloc((void **)&h_in, sizeof(int)* DATAINSIZE, ←  
    cudaHostAllocMapped); // Allocate mapped memory  
cudaMalloc((void **)&d_out, sizeof(int) * DATAOUTSIZE); ←  
    // Allocate device memory  
doSmt<<< gridDim, blkDim >>>(h_in, d_out);  
cudaMemcpy(h_out, d_out, sizeof(int) * DATAOUTSIZE, ←  
    cudaMemcpyDeviceToHost); // Device-to-host transfer
```

- The last line can be simplified to:

```
cudaMemcpy(h_out, d_out, sizeof(int)* DATAOUTSIZE, cudaMemcpyDefault); ←  
    // "Implied" device-to-host transfer
```


Unified memory

- Unified Memory advances the zero-copy memory mechanism by introducing **managed memory**, i.e., memory allocated on *both* host and device that is maintained **coherent** under the control of the device driver.
- The differences between unified memory and zero-copy memory are:
 - In zero-copy memory, the data transfer is initiated by data access. In unified memory, the transfer from host to device is triggered before a kernel is launched.
 - In unified memory, the results of the computation are transferred from the device to the host after termination of a kernel. In zero-copy memory, an explicit memory copy operation is required.
- A **single-pointer** to the data is maintained. However, host access to managed memory is restricted (causes an exception) when the device is accessing managed memory (e.g., during a kernel execution).

Unified memory (cont.)

- Unified Memory can be allocated:
 - **Dynamically**, via a call to the `cudaMallocManaged()` function, which is just a variant of `cudaMalloc()`:

```
template < class T > cudaError_t cudaMallocManaged (
    T **devPtr,           // Address for storing the memory pointer
                          // (IN/OUT)
    size_t size,          // Size in bytes of the required memory (IN)
    unsigned flags)       // Creation flag, defaults to
                          // cudaMemAttachGlobal (IN)
```

- **Statically**, by declaring a global variable as being `__managed__`.
- The flags in `cudaMallocManaged` can be either:
 - `cudaMemAttachGlobal` : memory is accessible by all kernels
 - `cudaMemAttachHost` : memory is accessible only to kernels launched by the thread that allocates this block

Unified memory (cont.)

- Unified Memory is more of a *productivity tool* than a *performance-enhancing mechanism*:
 - Memory is still transferred across the PCIe bus : no magic there!
 - Source code can be slimmed-down as far as memory management is concerned.
- In the example that follows we perform histogram calculation with:
 - Image data dynamically allocated (via `cudaMallocManaged`).
 - Histogram data statically allocated (via `__managed__`).

Unified memory histogram example

```
__device__ __managed__ int hist[BINS];  
//*****  
int main (int argc, char **argv)  
{  
    PGImage inImg (argv[1]);  
  
    int *in;  
    int *cpu_hist;  
    int i, N, bins;  
  
    N = ceil ((inImg.x_dim * inImg.y_dim) / (sizeof (int) * 1.0));  
    cudaMallocManaged ((void **) &in, sizeof (int) * N);  
  
    memcpy (in, inImg.pixels, sizeof (int) * N);  
    memset (hist, 0, bins * sizeof (int));  
  
    GPU_histogram_atomic <<< 16, 256 >>> (in, N, hist);  
    cudaDeviceSynchronize ();    // Wait for the GPU launched work to complete  
  
    for (i = 0; i < BINS; i++)  
        printf ("%i %i\n", i, hist[i]);  
  
    cudaFree ((void *) in);  
    cudaDeviceReset ();
```

Static allocation

Host functions affect device memory!

__device__ memory is accessible from the host

CUDA streams

CUDA kernels are powerful because they allow us solve problem ***asynchronously*** by taking advantage of the large collections of CUDA cores on GPU. Here, we use the concept of **threads** for executing the kernels asynchronously.

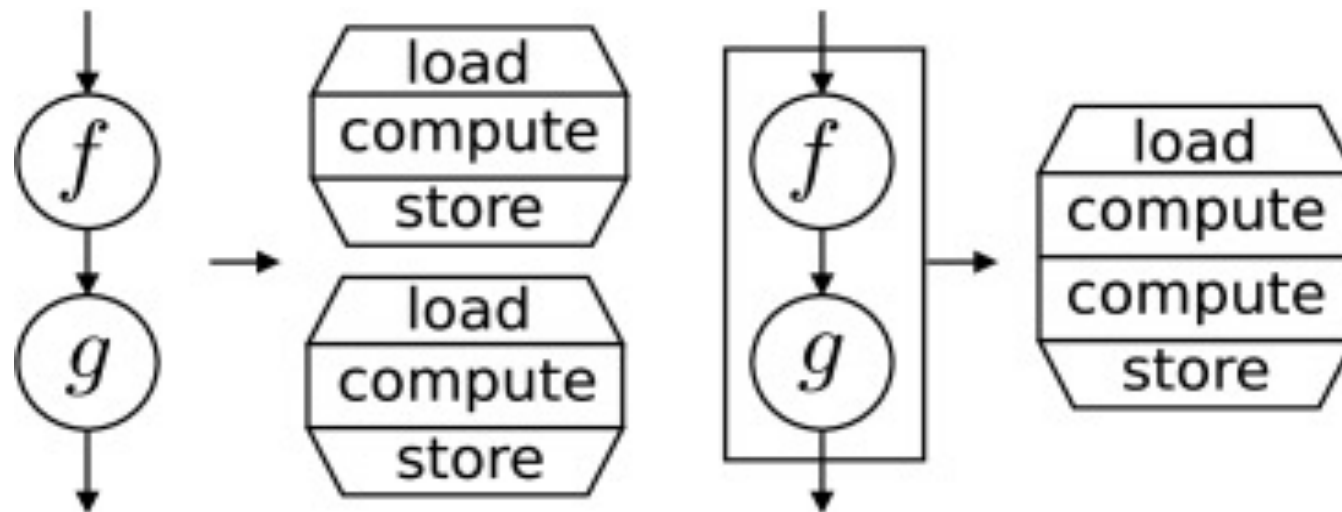
In practice, we have two additional steps, in addition to kernel executions, to solve the entire problem. These two steps are **memory copy from host to device** and **memory copy from device to host**. Intuitively, we would copy the input memory from host to device first, then execute the kernel to compute the output, and finally copy the output memory from device back to host.

However, this serial approach might not be ***optimal*** because we may further improve the performance by doing memory copy from host to device, kernel executions, and memory copy from device to host, **concurrently**. To do this, we would need to understand the concept of **streams**.

CUDA streams

- Contemporary GPUs are equipped with gigabytes of global memory, that can be used to preserve intermediate data between the successive invocations of different kernels.
- **Kernel fusion** is the term used to refer to the approach of replacing multiple kernel invocations by a single one, in an attempt to minimize data transfers.
- Alternatively, one could **hide** the cost of memory transfers, by overlapping them with kernel execution.
- A **stream** is a sequence of commands (including device memory copies, memory setting and kernel invocations) that are executed in order.
- All CUDA functions are associated with a **default stream**.

Kernel fusion



Loop fusion

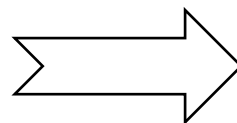
Merges two ***adjacent*** countable loops into a single loop

Reduces the cost of test and branch code

Fusing loops that refer to the same data enhances temporal locality, thus reducing the volume of memory-to-memory transfers

One potential drawback is that larger loop body may reduce instruction locality when the instruction cache is very small

```
for i = 1, N
  A(i) = B(i) + C(i)
for i = 1, N
  D(i) = A(i) + B(i)
```



```
for i = 1, N
  A(i) = B(i) + C(i)
  D(i) = A(i) + B(i)
```


Loop fussion (con't)

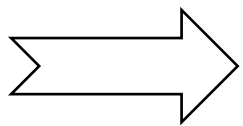
Be careful about loop-carried backward dependences

Sometimes, we may want to do the opposite transformation (called **loop fission**). Why?

Going back to streams ...

Thread resizing

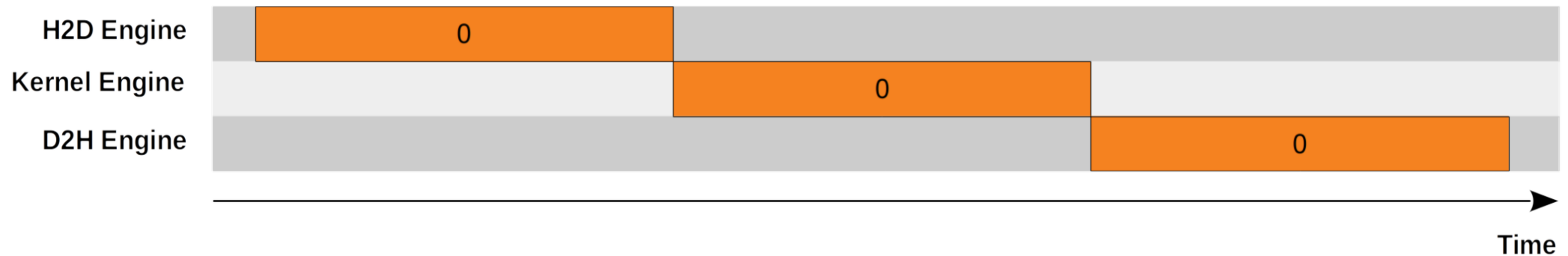
```
for (i=1; i < 1024; i++)  
    D(i) = B(i-1) + A(i) + B(i+1)  
    C(i) = A(i-1) + B(i) + B(i+2)
```



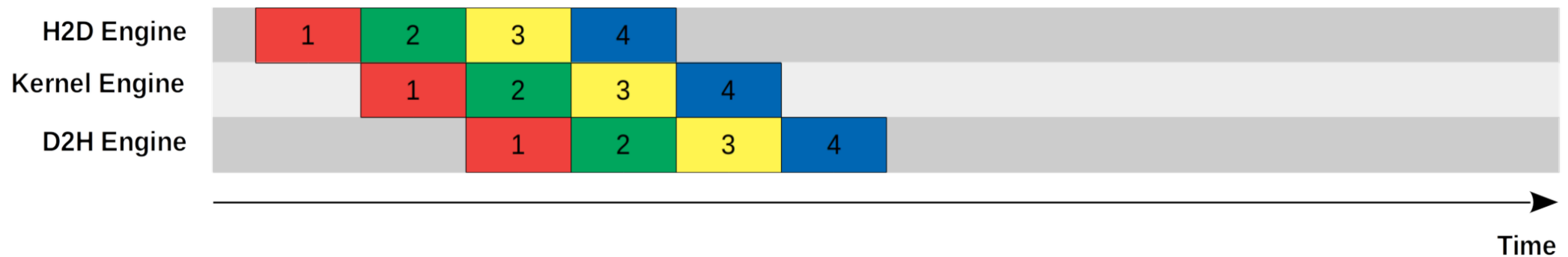
```
for (i=1; i < 1024; i=i+2)  
    D(i) = B(i-1) + A(i) + B(i+1)  
    C(i) = A(i-1) + B(i) + B(i+2)  
    D(i+1) = B(i) + A(i+1) + B(i+2)  
    C(i+1) = A(i) + B(i+1) + B(i+3)
```

Serial model vs concurrent model

Serial Model



Concurrent Model



sort of *pipelining* ...

Serial model vs concurrent model

Serial Model

We first copy the input memory from host to device first, then execute the kernel to compute the output, and finally copy the output memory from device back to host.

Concurrent Model

We make memory copy from host to device, kernel executions, and memory copy from device to host, asynchronous. We divide the memory into N chunks. After finishing copying the first chunk from host to device, we launch the smaller kernel execution to process the first chunk. In the meantime, the host to device (H2D) engine becomes available and it proceeds to copy the second chunk from host to device. Once the first chunk has been processed by the kernel, the output memory would be copied from device to host using the device to host engine (D2H) engine. In the meantime, the host to device (H2D) engine and the kernel engine becomes available and they proceed to copy the third chunk from host to device and process for the second chunk, respectively.

The question then becomes how do we write a CUDA program such that the commands for each of the chunks are executed in order, and different chunks could be executed ***concurrently***. The answer is to use **CUDA stream**.

CUDA streams

In CUDA terminology, a **stream** is a sequence of commands (possibly issued by different host threads) that execute in order.

Different streams may execute their commands *out of order* with respect to one another or *concurrently*.

This is exactly what we want to implement the concurrent model for CUDA programs.

CUDA streams

```
cudaStream_t stream[nStreams];
```

```
for (int i = 0; i < nStreams; i++)  
{  
    checkCuda(cudaStreamCreate(&stream[i]));  
}
```

```
for (int i = 0; i < nStreams; i++)  
{  
    checkCuda(cudaStreamDestroy(stream[i]));  
}
```

CUDA streams

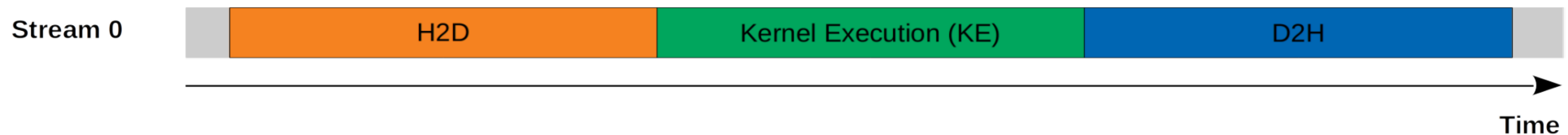
To implement the concurrent model, instead of calling `cudaMemcpy`, we call `cudaMemcpyAsync` and launch kernel with the stream specified so that they will return to the host thread immediately after call.

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

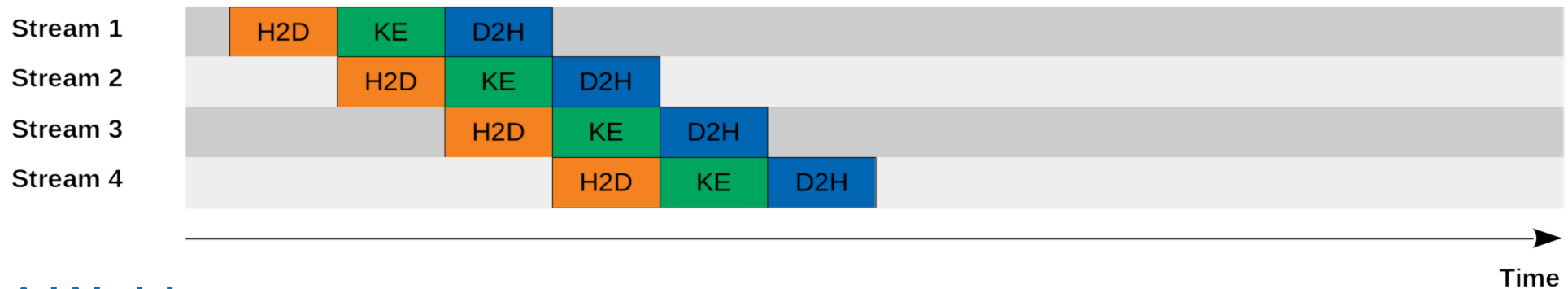
Serial vs concurrent model

We could also view the serial model and the concurrent model from the *stream's* perspective:

Serial Model



Concurrent Model



Serial Model

We only have one (default) CUDA stream. All the commands were executed in order.

Concurrent Model

We have N (non-default) CUDA streams. All the commands in the same stream were executed in order. Different streams can (and, hopefully, will) overlap in execution.

CUDA streams

Kernel executions on different CUDA streams could run simultaneously, provided that the computation resources are sufficient.

Exactly how this parallelism achieved is not revealed

CUDA operations in different streams may run
concurrently

CUDA operations from different streams may run in an
interleaved fashion

CUDA streams

Default stream

- Stream used when no stream is specified

Completely synchronous w.r.t. host and device

- As if `cudaDeviceSynchronize()` inserted before and after every CUDA operation

Requirements for concurrency

- CUDA operations must be in different streams
- `cudaMemcpyAsync` with host from 'pinned' memory

Page-locked memory

Allocated using `cudaMallocHost()` or `cudaHostAlloc()`

- Sufficient resources must be available
`cudaMemcpyAsync`s in different directions Device resources (SMEM, registers, blocks, etc.)

Using CUDA streams

- **Creation** : via a call to `cudaStreamCreate()`.

```
cudaError_t cudaStreamCreate(cudaStream_t * pStream);
```

- **Use** :

- Kernel launch:

```
kernelFunction <<< gridDim, blockDim, sharedMem, stream >>>  
( list_of_parameters );
```

- Memory operations:

- `cudaMemcpyAsync` : **same as** `cudaMemcpy` with the addition of a stream parameter
- `cudaMemsetAsync` : **same as** `cudaMemSet`, with the addition of the stream parameter

- **Synchronization** : via a call to `cudaStreamSynchronize()`.

Synchronization is blocking. Upon return, all the stream-queued operations are complete.

- **Destruction** : via a call to `cudaStreamDestroy()`. It's non-blocking.

Stream example

- An example sporting two streams.
- Using pinned host memory speeds up data transfer to the device.

```
cudaStream_t str[2];
int *h_data[2], *d_data[2];
int i;

// Allocate memory first in both host and device
for(i=0;i<2;i++)
{
    cudaMallocHost ((void **) &(h_data[i]), sizeof(int) * DATASIZE);
    cudaMalloc((void **) &(d_data[i]), sizeof(int) * DATASIZE);
}

// initialize h_data[i]....
```

Stream example (cont.)

```
// Now start populating the streams
for (i=0; i<2; i++)
{
    cudaStreamCreate(&(str[i]));

    cudaMemcpyAsync(d_data[i], h_data[i], sizeof(int) * DATASIZE, ←
        cudaMemcpyHostToDevice, str[i]);

    doSmt <<< 10, 256, 0, str[i] >>> (d_data[i]);

    cudaMemcpyAsync(h_data[i], d_data[i], sizeof(int) * DATASIZE, ←
        cudaMemcpyDeviceToHost, str[i]);
}

// Synchronization and clean-up
cudaStreamSynchronize(str[0]);
cudaStreamSynchronize(str[1]);
cudaStreamDestroy(str[0]);
cudaStreamDestroy(str[1]);
```

Loop finishes pretty much immediately...

Blocking calls ensure termination of queued/pipelined operations.

Summary: Synchronous

```
cudaMalloc ( &dev1, size ) ;
```

```
double* host1 = (double*) malloc ( &host1, size ) ;
```

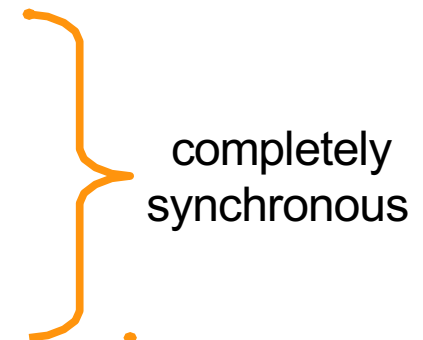
```
...
```

```
cudaMemcpy ( dev1, host1, size, H2D ) ;
```

```
kernel2 <<< grid, block, 0 >>> ( ..., dev2, ... ) ;
```

```
kernel3 <<< grid, block, 0 >>> ( ..., dev3, ... ) ;
```

```
cudaMemcpy ( host4, dev4, size, D2H ) ;
```



completely
synchronous

All CUDA operations in the default stream are synchronous!

Summary: Simple Example: Asynchronous, No Streams

```
cudaMalloc ( &dev1, size ) ;  
double* host1 = (double*) malloc ( &host1, size ) ;  
...
```

```
cudaMemcpy ( dev1, host1, size, H2D ) ;
```

```
kernel2 <<< grid, block >>> ( ..., dev2, ... ) ;  
some_CPU_method () ;
```

```
kernel3 <<< grid, block >>> ( ..., dev3, ... ) ;  
cudaMemcpy ( host4, dev4, size, D2H ) ;  
...
```



potentially
overlapped

GPU kernels are asynchronous with host by default

Summary: Simple Example: Asynchronous with Streams

```
cudaStream_t stream1, stream2, stream3, stream4 ;  
cudaStreamCreate ( &stream1) ;  
...  
cudaMalloc ( &dev1, size ) ; cudaMallocHost ( &host1, size ) ; // pinned memory required on host  
...  
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;  
kernel2 <<< grid, block, 0, stream2 >>> ( ..., dev2, ... ) ;  
kernel3 <<< grid, block, 0, stream3 >>> ( ..., dev3, ... ) ;  
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 ) ;  
some_CPU_method ( ) ;  
...
```

potentially
overlapped

Fully asynchronous / concurrent

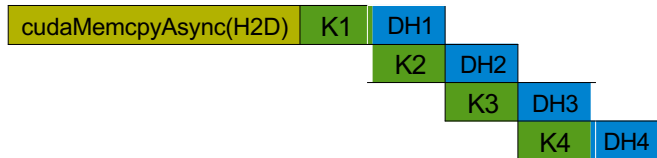
Data used by concurrent operations should be independent

Amount of Concurrency

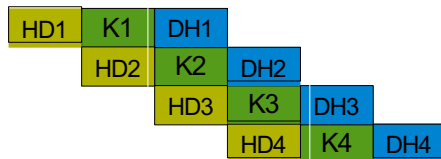
Serial (1x)



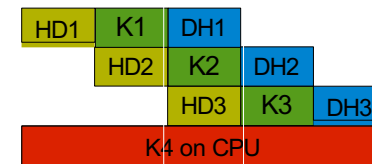
2-way concurrency (up to 2x)



3-way concurrency (up to 3x)



4-way concurrency (3x+)



4+ way concurrency

