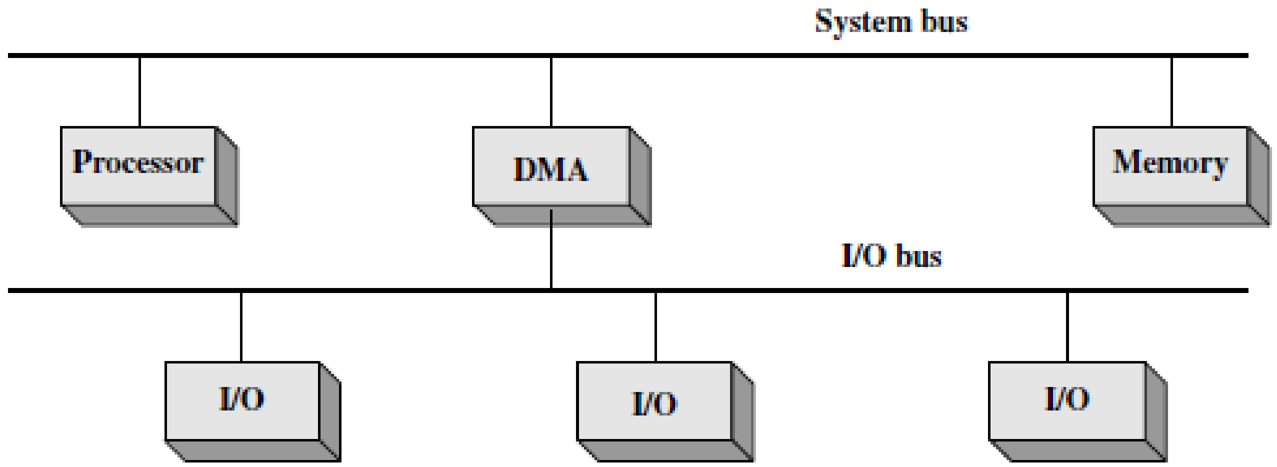
Lecture No.8: Host-Device Data Transfer and Streams

Muhammad Osama Mahmoud, TA

Computer and Systems Eng. Dept. - Faculty of Engineering – University of Minia

CPU-GPU Data Transfer

- Data transfers between CPU and GPU is performed using DMA (Direct Memory Access)
- DMA is a direct connection between the system memory (DRAM) and other I/O devices including GPU (through PCIe)



Virtual Memory Management

- Virtual memory addresses (pointers) are translated into physical addresses and vise versa while locating existing data or storing new data
- Each virtual address space is sectioned into pages when mapped into physical memory
- DMA uses physical addresses not virtual addresses

Data Transfer and Virtual Memory

- To transfer data to/from GPU and CPU, *cudaMemcpu*() is used
- The copy process may be done in one or more DMA transfers
- Address is translated and page presence checked at the beginning of each DMA transfer
- No address translation for the rest of the same DMA transfer so that high efficiency can be achieved

Data Transfer with Pinned Memory

- Pinned memory are virtual memory pages that are specially marked so that they cannot be paged out

- Allocated with a special system API function *cudaMallocHost*()

- *cudaMemcpy*() is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

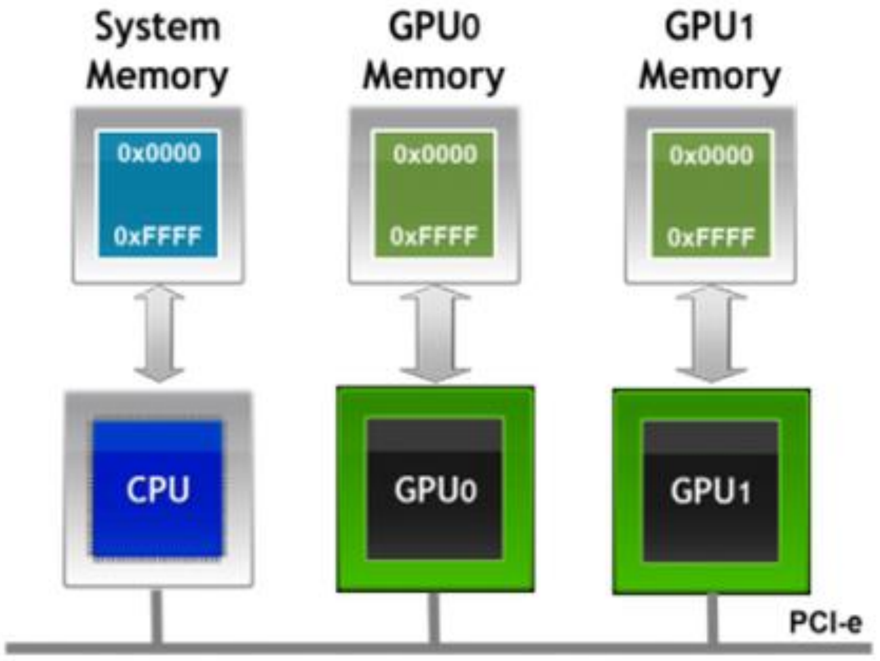
- Pinned memory is a limited resource

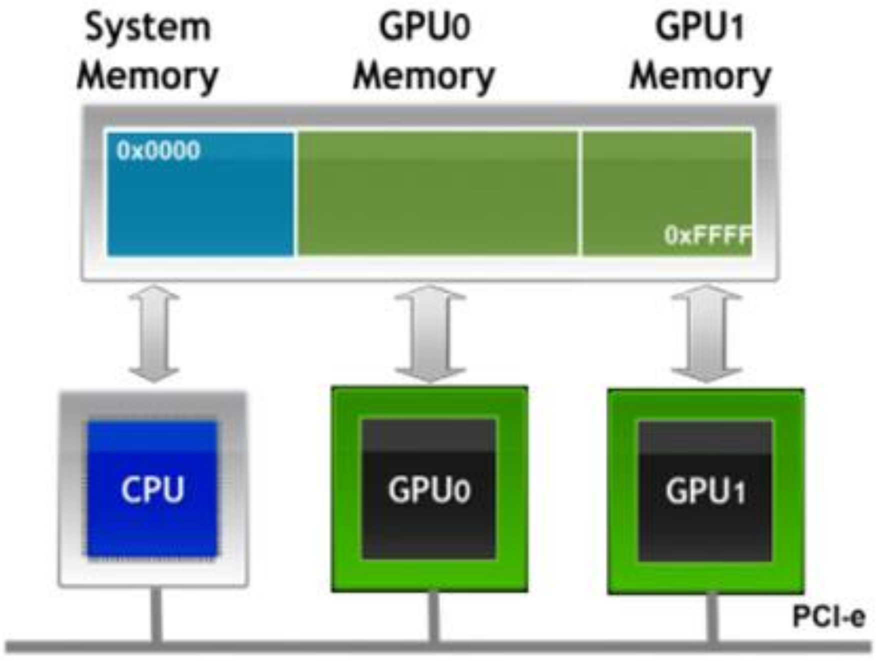
CUDA Unified Memory

- Simplifies programming by enabling applications to access CPU and GPU memory without the need to manually copy data from one to the other
- makes it easier to add support for GPU acceleration in a wide range of programming languages.
- Actually it’s not a unified memory by literally speaking, it just simulates the behavior of the unified memory between CPU and GPU
- Allocated with a special system API function *cudaMallocManaged*()
- Separate pointers allocated for CPU and GPU are not required any more

CUDA Unified Memory (Cont.)

- *cudaMallocManaged*() intelligently manages the memory copies between CPU and GPU memory

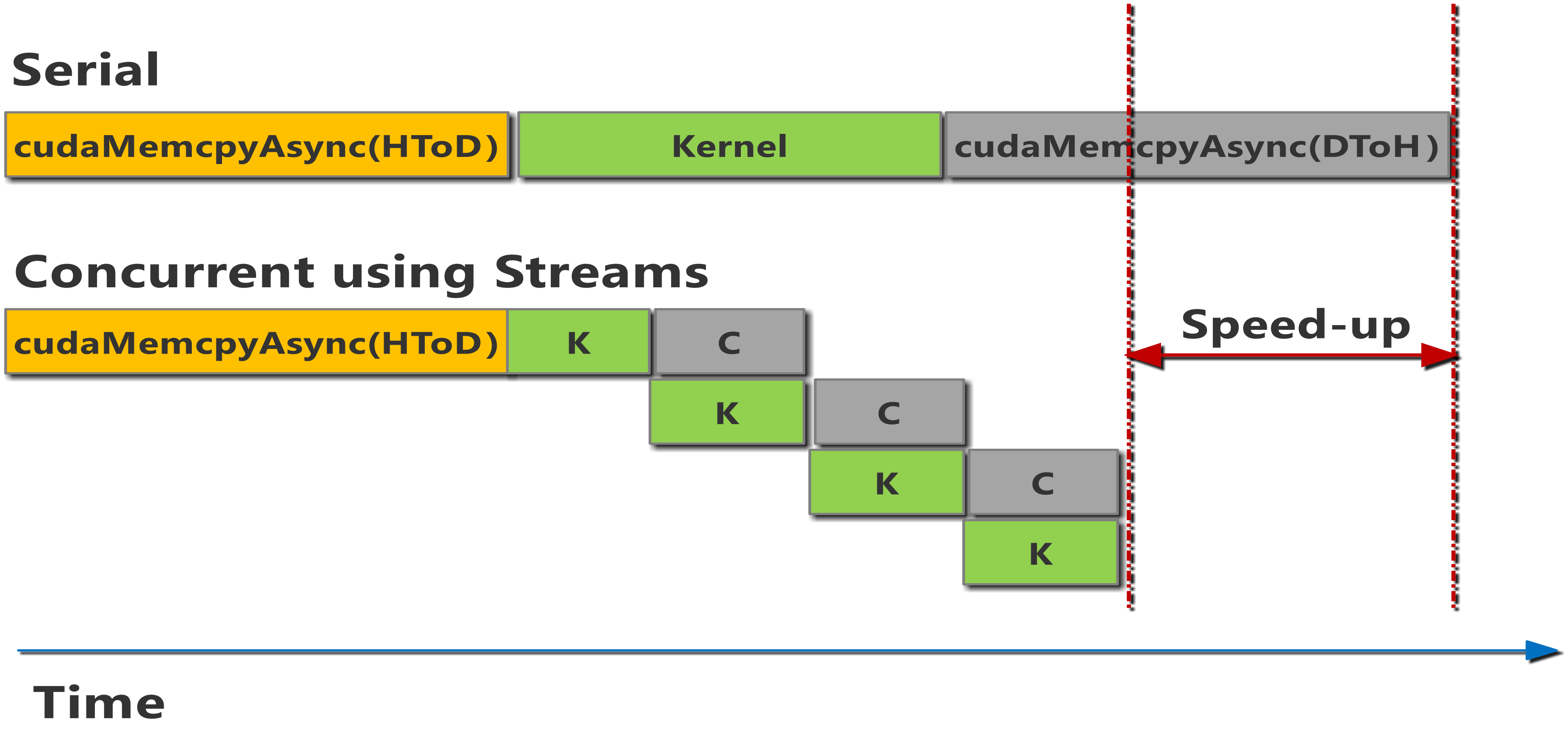




CUDA Streams

- A sequence of operations that execute in issue-order on the GPU
- CUDA operations in different streams may run concurrently
- CUDA operations from different streams may be pipelined or overlapped
- Overlapping kernel execution with memory transfer is done using *cudaMemcpyAsync*()

CUDA Streams



A simple Host Program

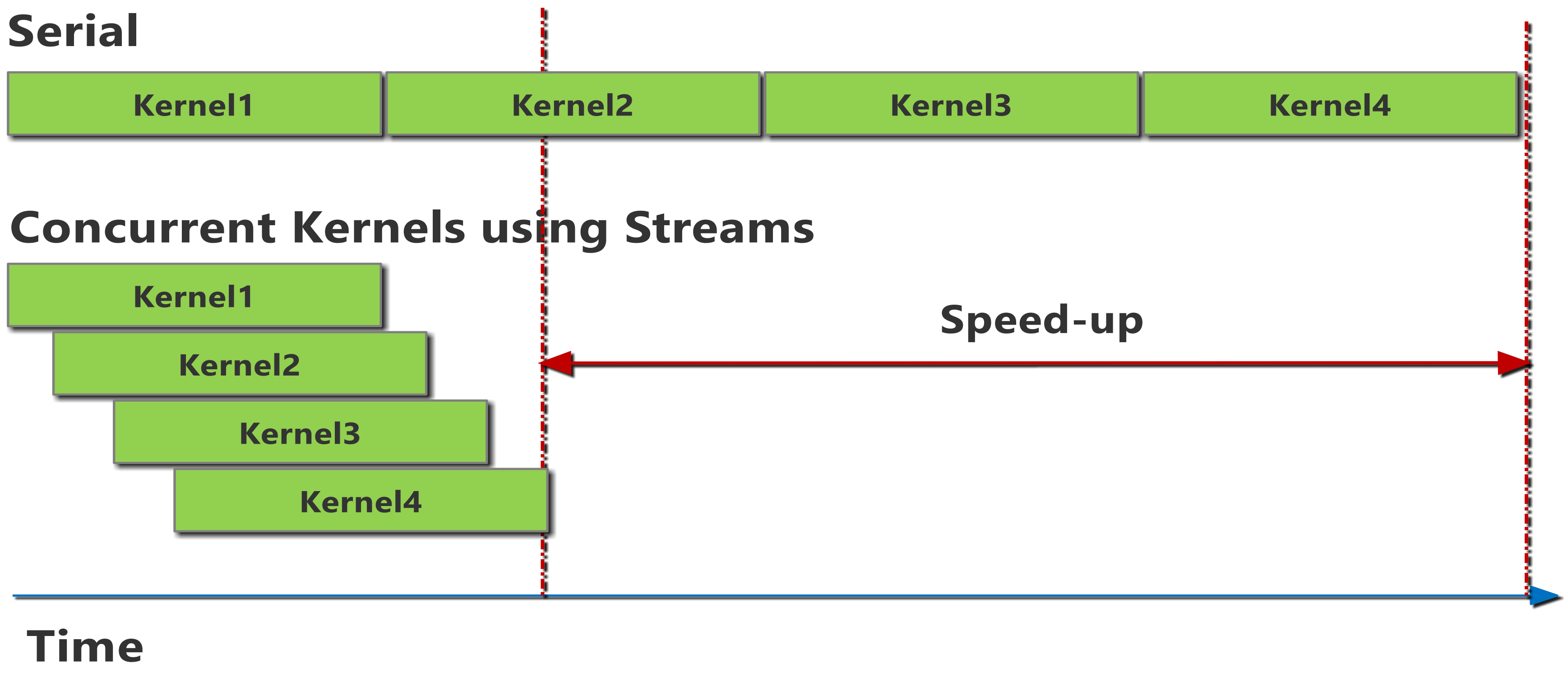
```
cudaStream_t stream0, stream1;
cudaStreamCreate(&stream0);
cudaStreamCreate(&stream1);

float *d_A0, *d_B0, *d_C0;// device memory for stream 0
float *d_A1, *d_B1, *d_C1;// device memory for stream 1

// Allocation functions
…
```

A simple Host Program (Cont.)

```
for (int i = 0; i < N; i += CHUNK*2)
{
        cudaMemcpyAsync(d_A0, h_A + i, CHUNK*2*sizeof(float) , …, stream0);
        cudaMemcpyAsync(d_B0, h_B + i, CHUNK*2*sizeof(float) , …, stream0);
        // first kernel launch with stream0
        vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0,…);
        cudaMemcpyAsync(h_C + i, d_C0, CHUNK*sizeof(float),…, stream0);
        cudaMemcpyAsync(d_A1, h_A + i + CHUNK, CHUNK*sizeof(float) , …, stream1);
        cudaMemcpyAsync(d_B1, h_B + i + CHUNK, CHUNK*sizeof(float) , …, Stream1);
        // first kernel launch with stream1
        vecAdd<<<CHUNK/256, 256, 0, stream1>>>(d_A1, d_B1, …);
}
```

Concurrent Kernel Execution

- Concurrent kernels must support operations on different data arrays
- Each kernel must assigned a different stream other than 0 (default stream)
- Kepler architecture supports up to 32 concurrent kernels



References

[1] Wen-mei W. Hwu, “Heterogeneous Parallel Programming”. Online course, 2014. Available: https://class.coursera.org/hetero-002

[2] S. Rennich, “CUDA C/C++ Streams and Concurrency”, Jan. 2012.