KOÇ UNIVERSITY

Data Transfers, Unified Virtual Memory, CUDA Streams

Didem Unat

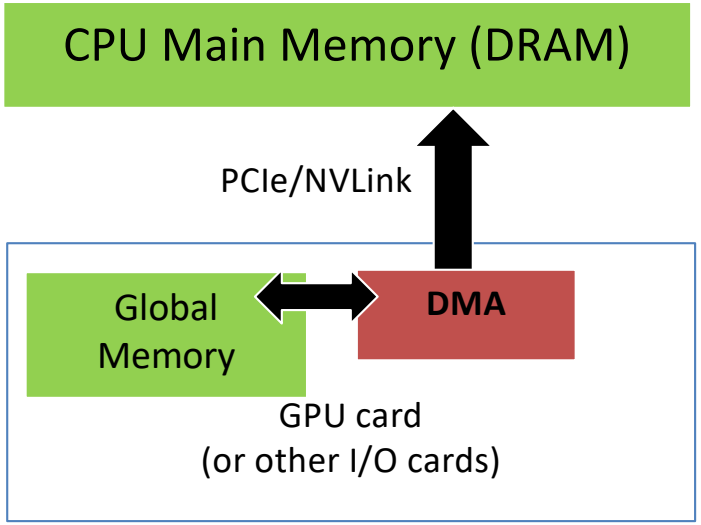
dunat@ku.edu.tr

http://parcorelab.ku.edu.tr

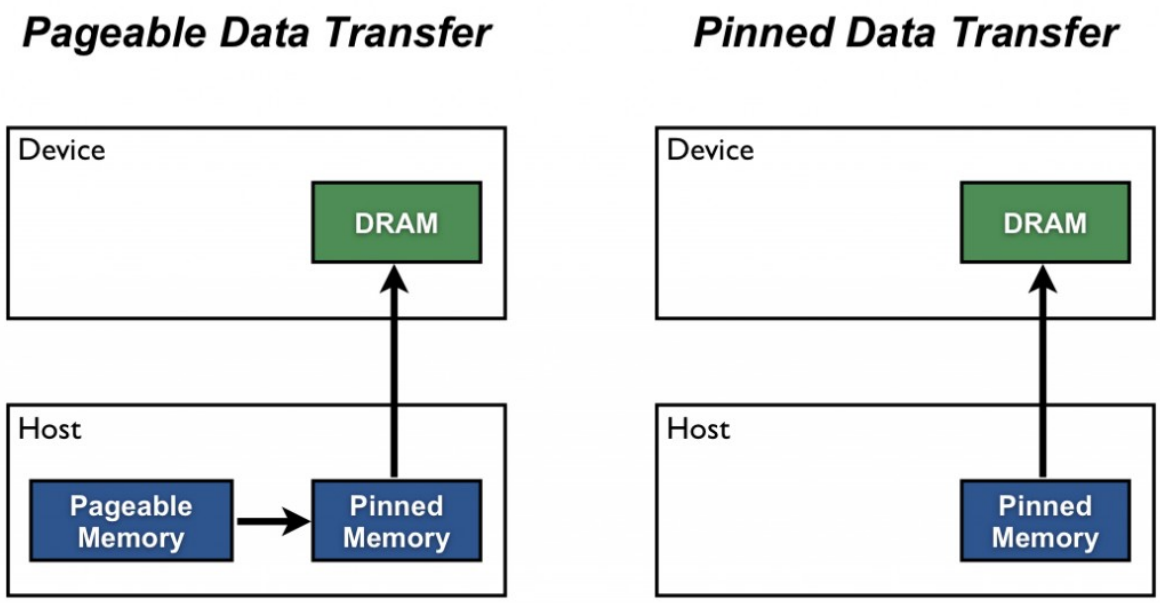
- Data Transfers
- CUDA Pageable Memory
- CUDA Pinned Memory
- Bandwidth Test
- Unified Virtual Memory
- Streams
- Overlap Transfers

CPU-GPU Data Transfer using DMA

- DMA (Direct Memory Access) hardware is used by `cudaMemcpy()` for better efficiency
  - Frees CPU for other tasks
  - Hardware unit specialized to transfer a number of bytes requested by OS
  - Between physical memory address space regions (some can be mapped I/O memory locations)
  - Uses system interconnect, typically PCIe or NVLink in today's systems



CUDA: Pageable Memory vs Pinned Memory



CUDA Pageable Memory

- Host data is in pageable memory by default.
- CUDA copies data to the pinned memory first.
- Then, transfers data to the device memory.

CUDA Pinned Memory

- Host data is allocated in pinned memory.
- CUDA directly transfers data to device memory.

Virtual Memory Management

- Modern computers use virtual memory management
  - Many virtual memory spaces mapped into a single physical memory
  - Virtual addresses are translated into physical addresses
- Not all variables and data structures are always in the physical memory
  - Each virtual address space is divided into pages that are mapped into and out of the physical memory
  - Virtual memory pages can be mapped out of the physical memory (page-out) to make room
  - Whether or not a variable is in the physical memory is checked at address translation time

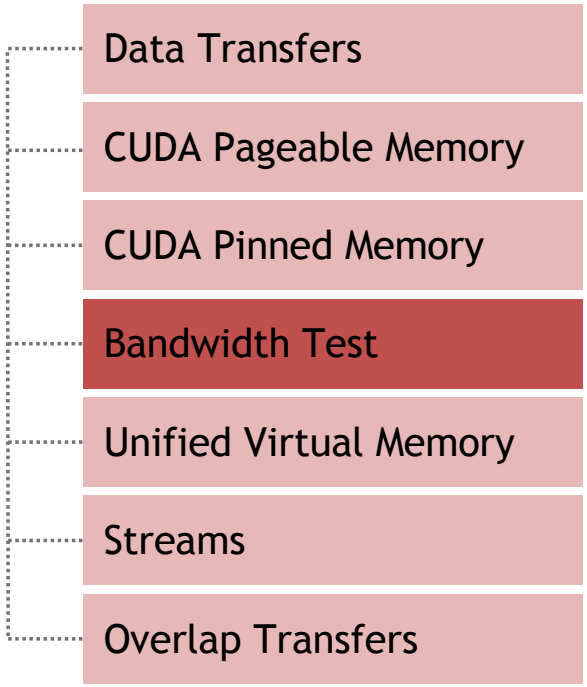
Data Transfers and Virtual Memory

- DMA uses physical addresses
  - When `cudaMemcpy()` copies an array, it is implemented as one or more DMA transfers
  - Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
  - No address translation for the rest of the same DMA transfer is performed so that high efficiency can be achieved

- During the copy, the OS could accidentally page-out the data that is being read or written by a DMA and page-in another virtual page into the same physical location

Data Transfers between Host & Device

- Minimize data transfer between the host and the device
- Pinned Memory
  - Use page-locked (pinned) memory which enables the GPU to request transfers to and from the host memory without the involvement of the CPU
    - `cudaHostAlloc()`
  - Prevents the **memory** from being swapped out and provides improved transfer speeds.
  - Should not be overused


Data Transfers
CUDA Pageable Memory
CUDA Pinned Memory
Bandwidth Test
Unified Virtual Memory
Streams
Overlap Transfers

Lab5-bandwidthTest

- Copy the lab from
  - /kuacc/users/dunat/COMP429/GPU/Hands-on-Labs/labs/5_bandwidthTest
- This is a simple test program to measure the memory copy bandwidth of the GPU and memory copy bandwidth across PCI-e bus
  - Measures host to device bandwidth
  - Measures device to host
  - Measure within device memory bandwidth (memory copy)
- Run bandwidth test with pinned memory

```
./bandwidth memory pinned
```

- Run bandwidth test with pageable memory

```
./bandwidth memory pageable
```

- Which transfer rate is higher?
- Compare PCI-e/Nvlink bandwidth versus memory bandwidth, which one is higher?

Bandwidth Test

- Nvidia provides a bandwidth test in its SDK

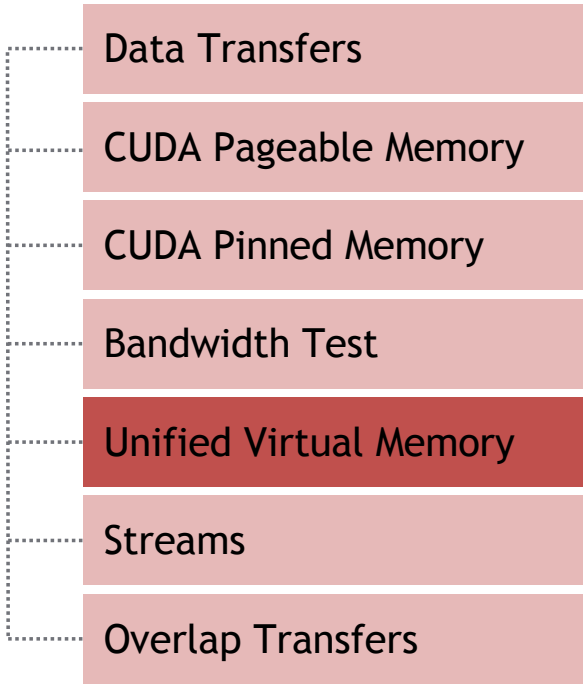
Pinned Memory

```
[dunat@it01 5_bandwidthTest]$ ./bandwidthTest
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla V100-PCIE-32GB
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     12021.9

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     12861.5

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     732061.2

Result = PASS
```

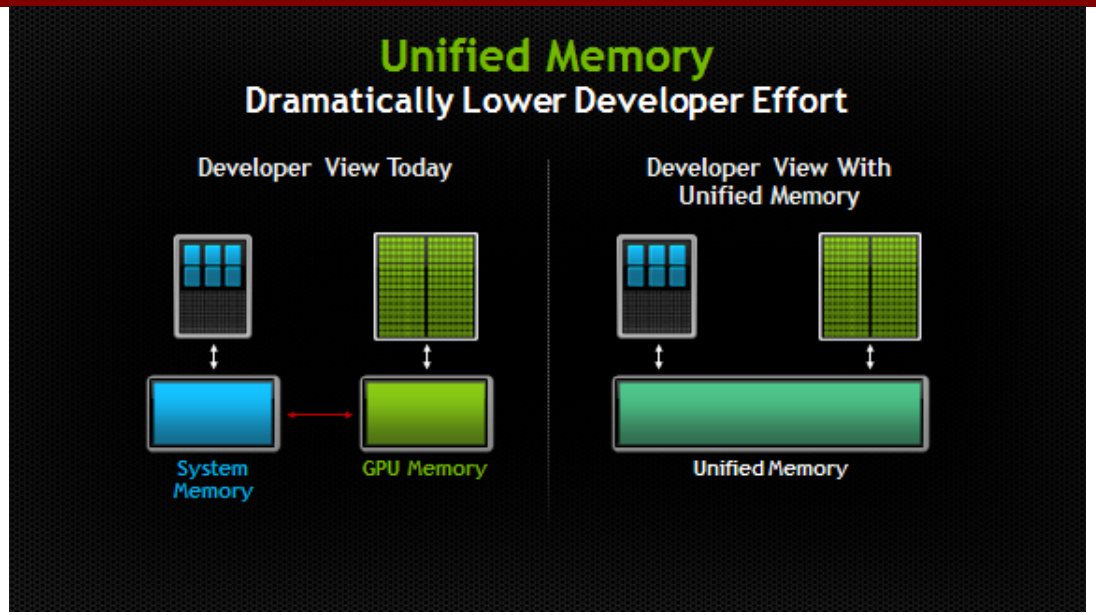
Pageable Memory

```
[dunat@it01 5_bandwidthTest]$ ./bandwidthTest memory pageable
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla V100-PCIE-32GB
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PAGEABLE Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     5014.1

 Device to Host Bandwidth, 1 Device(s)
 PAGEABLE Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     4755.7

 Device to Device Bandwidth, 1 Device(s)
 PAGEABLE Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     733146.2

Result = PASS
```

- Pinned memory provides higher bandwidth (x2) of pageable memory
  - Device bandwidth is higher than the PCI-bus bandwidth


Data Transfers
CUDA Pageable Memory
CUDA Pinned Memory
Bandwidth Test
Unified Virtual Memory
Streams
Overlap Transfers

CUDA: Unified Virtual Memory (UVM)



source: NVIDIA

- Handles memory management internally.
  - Programmer does not need to transfer memory.
- Simplifies GPU programming in terms of memory management
  - But results in lower performance
  - NVIDIA works on improvements in unified memory for future devices.

UVM

CUDA

- In addition to host variables create device variables
- Allocate device variables
- Transfer data from host to device
- Launch kernel
  - Specify block and thread geometry
- Transfer data from device to host
- Deallocate device variables

CUDA Unified Memory

- Allocate variables — cudaMallocManaged
- → nothing to be done
- → Same as before
- → nothing to be done
- Deallocate variables

Lab6-vecAdd_UVM

- Copy the lab from
  - **/kuacc/users/dunat/COMP429/GPU/Hands-on-Labs/labs/6_vectorAdd_UVM**
- Let's modify our vector addition example so that it uses UVM.
- Step 1: Define a common variable for each array that should be used in place of `device_array_a` and `host_array_a` such as `array_a`
  - `float *array_a = NULL;`
  - `float *array_b = NULL;`
  - `float *array_c = NULL;`

  Comment out rest of the array declarations
- Step 2: Remove/Comment out the host allocation code
  - `//host_array_a = (float*)malloc(num_bytes);`
- Step 3: Convert the device allocations so that it uses UVM
  - `cudaMallocManaged(&array_a, num_bytes);  //example`
- Step 4: Replace allocation check and initialization arrays with the newly created arrays
  - `array_a[i] = (float)i;  //example`

Lab6-vecAdd_UVM (cont.)

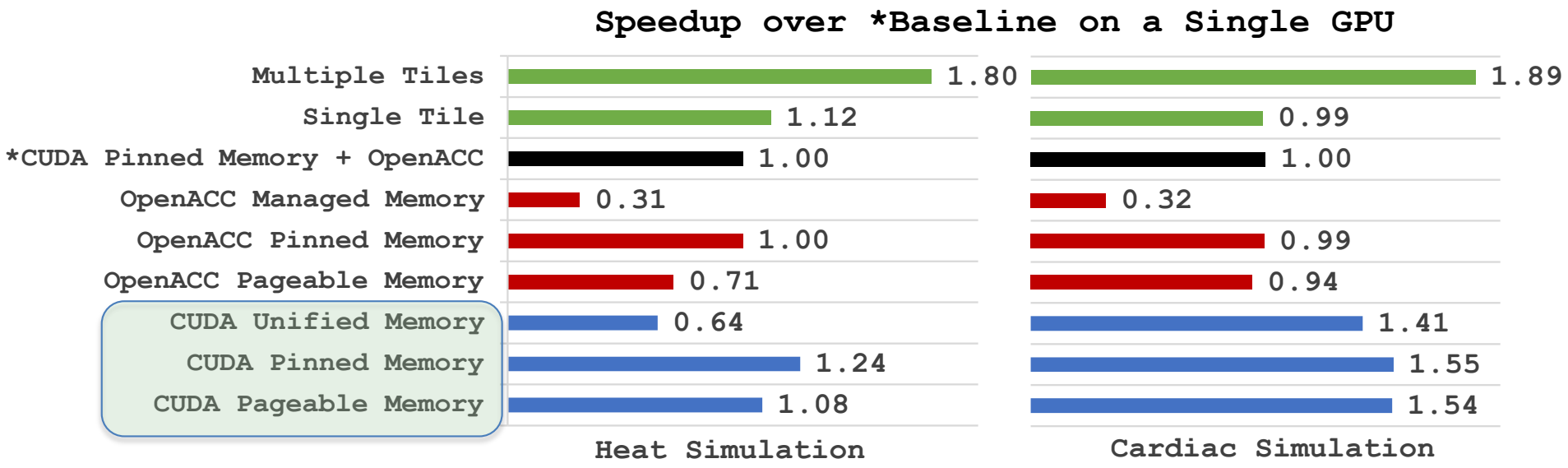
- Step 5: Comment out the cudaMemcpy codes – no need to transfer to device
  - `//cudaMemcpy (. . .);`
- Step 6: Kernel code does not change, only change the parameters so that it uses the UVM arrays
  - `vector_add<<<nBlocks, nThreads>>>(array_a, array_b, array_c, num_elements);`
- Step 7: Comment out the cudaMemcpy codes – no need to transfer back to host
  - `//cudaMemcpy (. . .);`
- Step 9: Print should use the UVM arrays as well
- Step 10: Remove deallocate the host arrays, rename the arrays used in cudaFree
  - `//free (. . .);`
  - `cudaFree(array_a);`

Lab6-vecAdd_UVM (cont.)

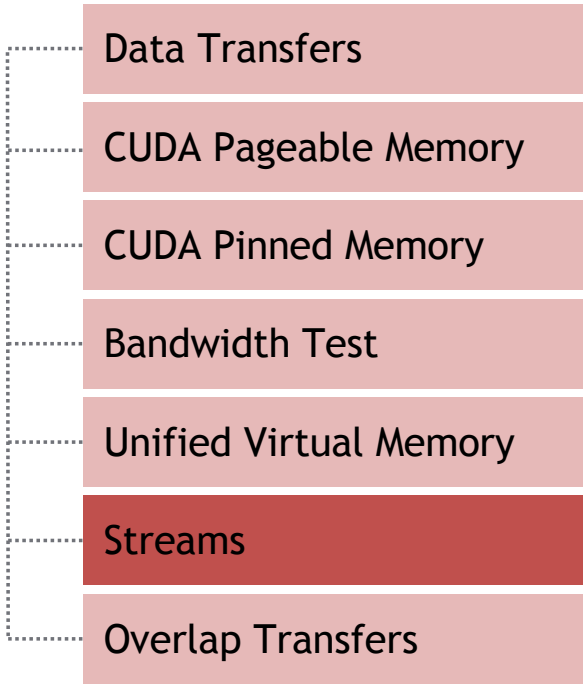
- Compile and run!
- Did it work? Do you get the correct results?
  - What is missing?

- Step 8: Need to wait for GPU to finish before accessing (printing values) on host because GPU and CPU are concurrently running
  - Add the following line after kernel launch to make sure kernel is completed
  - `cudaDeviceSynchronize();`

Our Recent Work

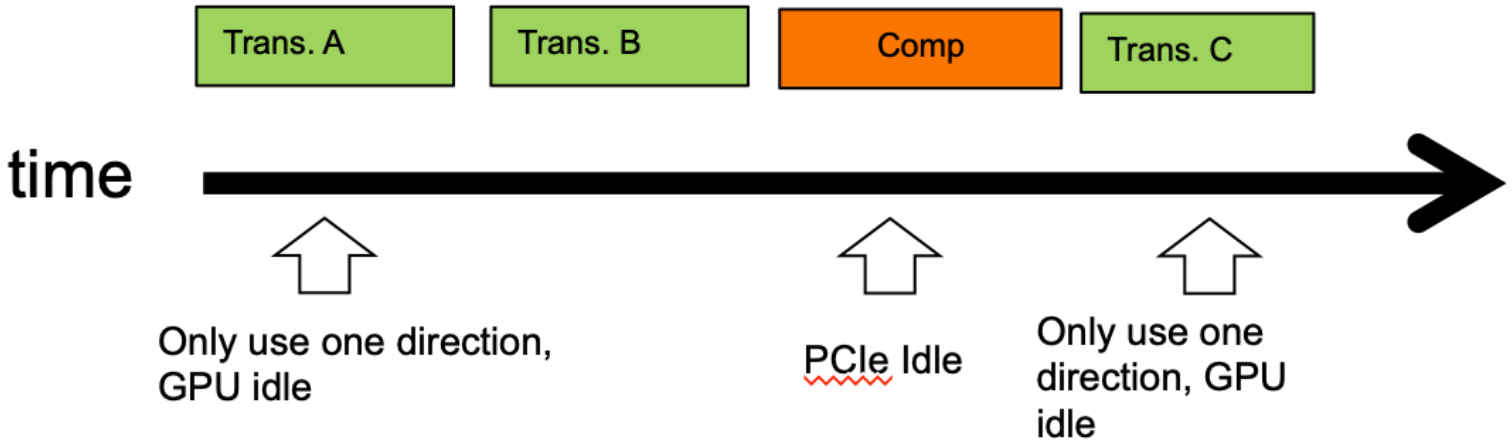


- It is convenient to develop programs with UVM, however it gives suboptimal performance
- The best performance is achieved with pinned memory.


Data Transfers
CUDA Pageable Memory
CUDA Pinned Memory
Bandwidth Test
Unified Virtual Memory
Streams
Overlap Transfers

Serialized Data Transfer and Computation

- So far, the way we use `cudaMemcpy` serializes data transfer and GPU computation



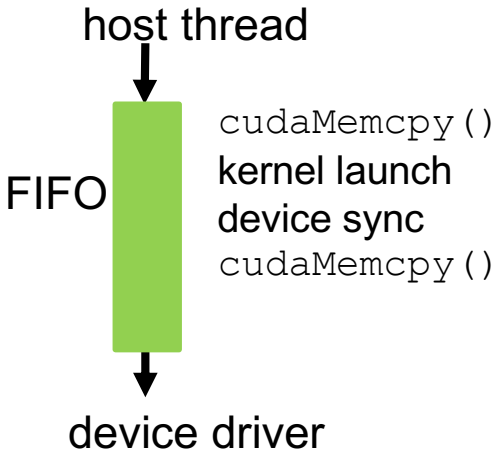
Streams

- CUDA operations in different streams may run concurrently
- Operations in the same stream run in order
- Operations from different streams may be interleaved

- Default Steam
  - Even if you are not using streams, there is one default stream
  - All device operations (kernels and data transfers) in CUDA run in the default stream

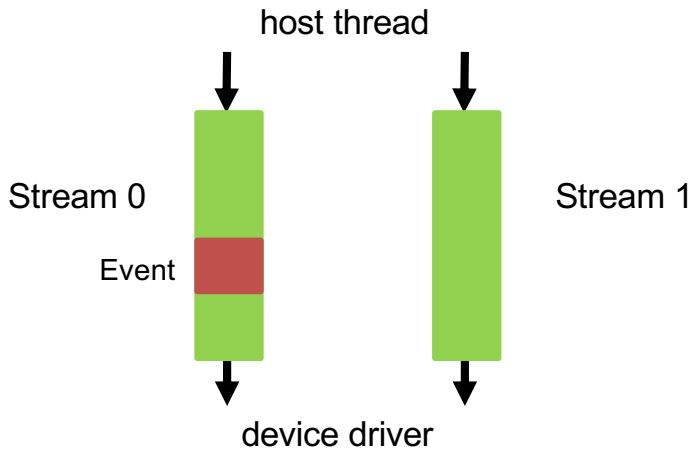
FIFO Order

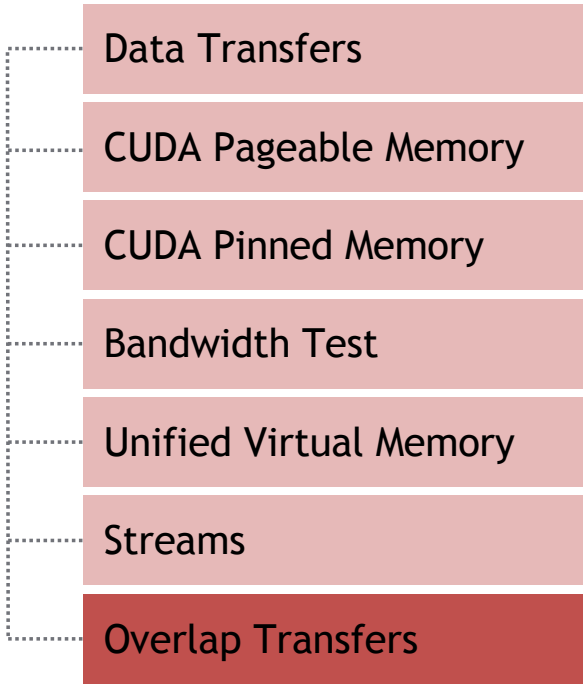
- Requests made from the host code are put into First-In-First-Out queues
  - Queues are read and processed asynchronously by the driver and device
  - Driver ensures that commands in a queue are processed in sequence. E.g., Memory copies end before kernel launch, etc.



Streams cont.

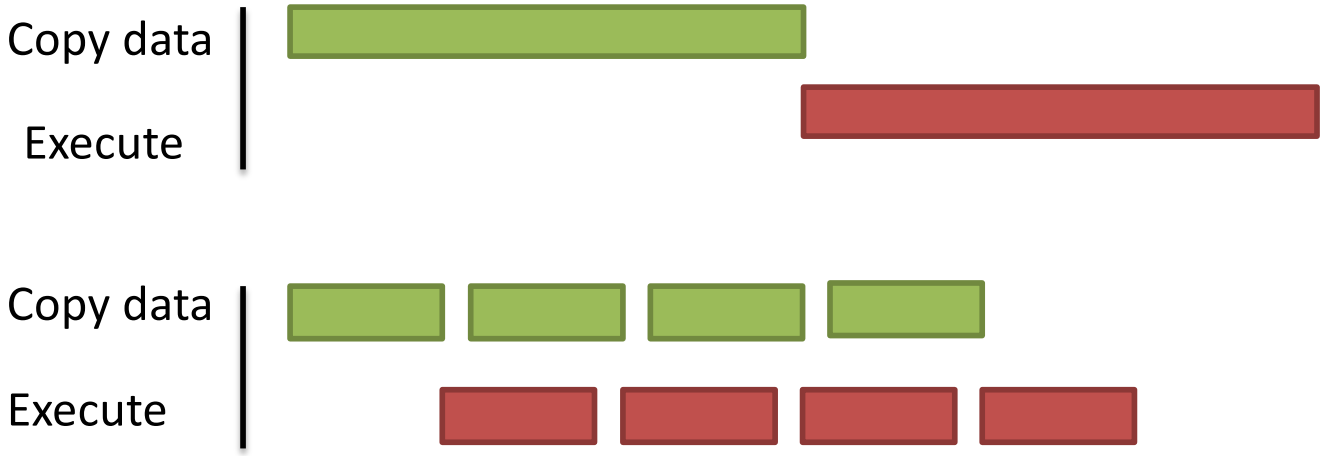
- To allow concurrent copying and kernel execution, use multiple queues, called “streams”
  - CUDA “events” allow the host thread to query and synchronize with individual queues (i.e. streams).




Data Transfers
CUDA Pageable Memory
CUDA Pinned Memory
Bandwidth Test
Unified Virtual Memory
Streams
Overlap Transfers

Data Transfers between Host & Device

- Asynchronous Data Transfers
  - Overlapping computation and data transfers
  - **`cudaMemcpyAsync()`**

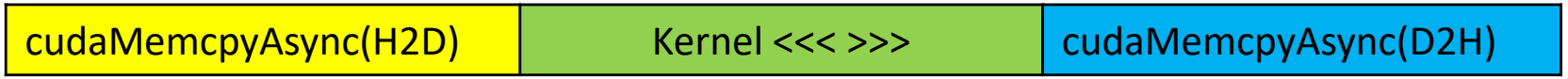


More concurrency

- Independent tasks can operate concurrently with one another
  - Computation on the host
  - Computation on the device
  - Memory transfers from the host to the device
  - Memory transfers from the device to the host
  - Memory transfers within the memory of a given device
  - Memory transfers among devices
- Need to use streams
  - A sequence of operations that execute in issue-order on the GPU
- Level of support depends on the compute capability of the device

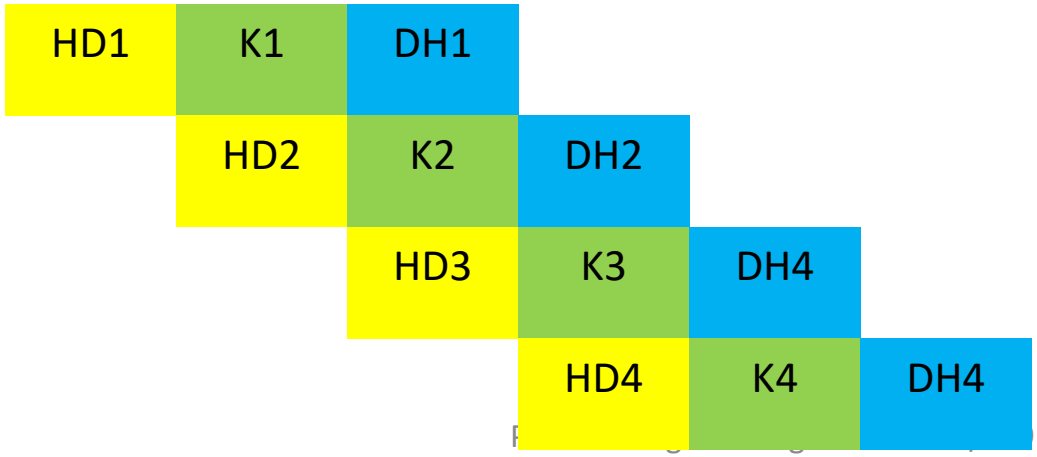
Stream Example

Serial (1x) Single Stream



2-way concurrency (up to 2x)

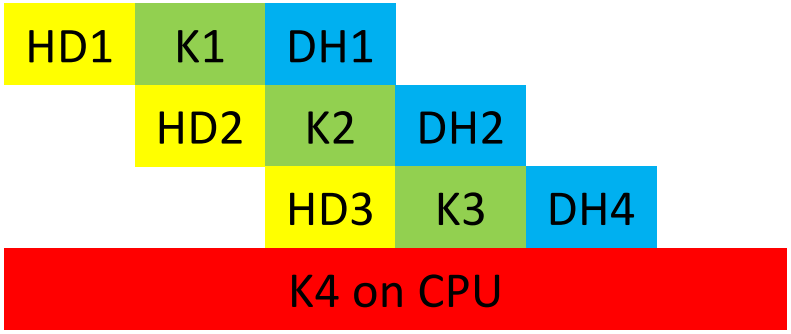
3-way concurrency (up to 3x)



Another example

- Can create concurrency only with new kernels
- Only two transfers can be active at a time
  - HD: Host to Device
  - DH: Device to Host

4-way concurrency (3x+)



Stream Usage

- Non-default streams in CUDA C/C++ are declared, created, and destroyed in host code as follows.

```
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1)
result = cudaStreamDestroy(stream1)
```

- Non-default stream we can use the cudaMemcpyAsync(), which performs non-blocking data transfers

```
cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
```

- To issue a kernel to a non-default stream, add a stream identifier (4^{th} argument)

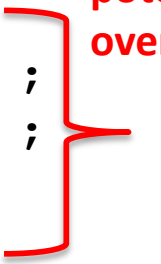
```
increment<<<1,N,0,stream1>>>(d_a);
```

Synchronization

- Synchronize everything
  - cudaDeviceSynchronize ()
  - Blocks host until all issued CUDA calls are complete
- Synchronize w.r.t. a specific stream
  - cudaStreamSynchronize (streamid)
  - Blocks host until all CUDA calls in streamid are complete
- Query a stream
  - cudaStreamQuery(stream)
  - whether all operations issued to the specified stream have completed, without blocking host execution.
- Synchronize operations within a single stream on a specific event
  - cudaStreamWaitEvent(event)

Simple Example

```
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 ();
```



- Fully asynchronous / concurrent
- Data used by concurrent operations should be independent

Simple Multi-Stream Host Code

```
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

// cudaMalloc() calls for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here
```

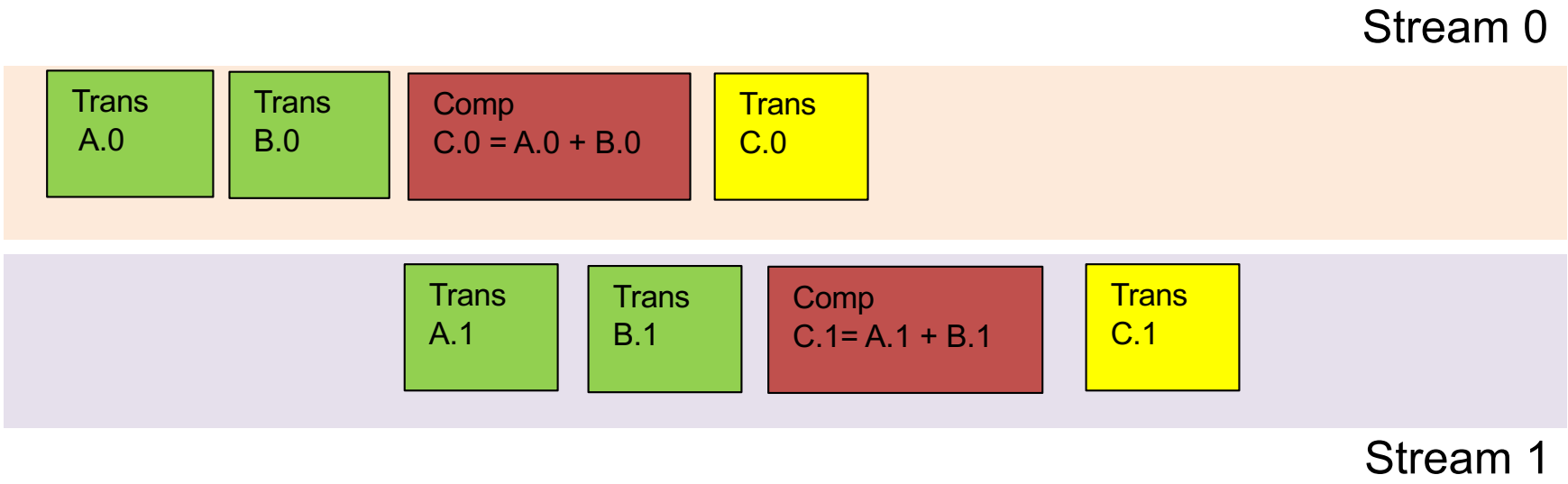
- Vector Add example with streams
- Create two streams, each of which computes half of the array

Simple Multi-Stream Host Code

```
cudaMemcpyAsync(d_A0, h_A, SegSize*sizeof(float),…, stream0);
cudaMemcpyAsync(d_B0, h_B, SegSize*sizeof(float),…, stream0);
vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, …);
cudaMemcpyAsync(h_C, d_C0, SegSize*sizeof(float),…, stream0);

cudaMemcpyAsync(d_A1, h_A+SegSize, SegSize*sizeof(float),…, stream1);
cudaMemcpyAsync(d_B1, h_B+SegSize, SegSize*sizeof(float),…, stream1);
vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);
cudaMemcpyAsync(h_C+SegSize, d_C1, SegSize*sizeof(float),…, stream1);
```

- Two streams: stream 0 and stream 1

Pipelined Operations



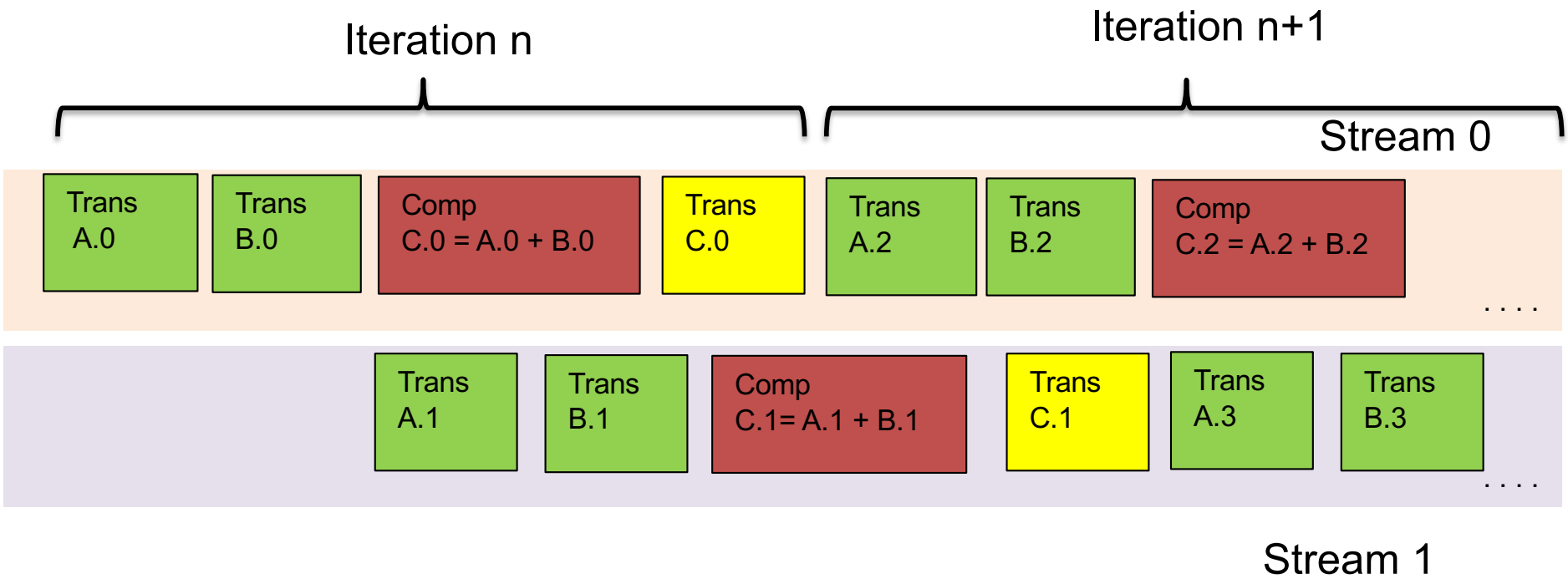
- Computation is overlapped with the transfer of A1. and B1.
- Transfer back is overlapped with H2D device transfer and computation

Simple Multi-Stream Host Code

```
cudaMemcpyAsync(d_A0, h_A, SegSize*sizeof(float),…, stream0);
cudaMemcpyAsync(d_B0, h_B, SegSize*sizeof(float),…, stream0);
cudaMemcpyAsync(d_A1, h_A+SegSize, SegSize*sizeof(float),…, stream1);
cudaMemcpyAsync(d_B1, h_B+SegSize, SegSize*sizeof(float),…, stream1);

vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, …);
vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);

cudaMemcpyAsync(h_C, d_C0, SegSize*sizeof(float),…, stream0);
cudaMemcpyAsync(h_C+SegSize, d_C1, SegSize*sizeof(float),…, stream1);
```

- Issuing the memory transfers first results in the same effect

Pipelined Operations



- More iterations allow more overlap
- The code gets complicated though

Acknowledgments

- These slides are inspired and partly adapted from
  - Programming Massively Parallel Processors: A Hands On Approach, available from *http: David Kirk and Wen-mei Hwu, February 2010, Morgan Kaufmann Publishers, ISBN 0123814723.*
  - *Streaming Blog*
    - https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/
  - CUDA Streams and Concurrency
    - By Steve Rennich NVIDIA
    - https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.