CPSC/ECE 4780/6780

General-Purpose Computation on Graphical Processing Units (GPGPU)

Lecture 9: Streams

Recap of Last Lecture

- What are race conditions?
- What is atomic operation?
- What kind of atomic operations do we have?
- What is atomic lock?
- How to implement lock function?

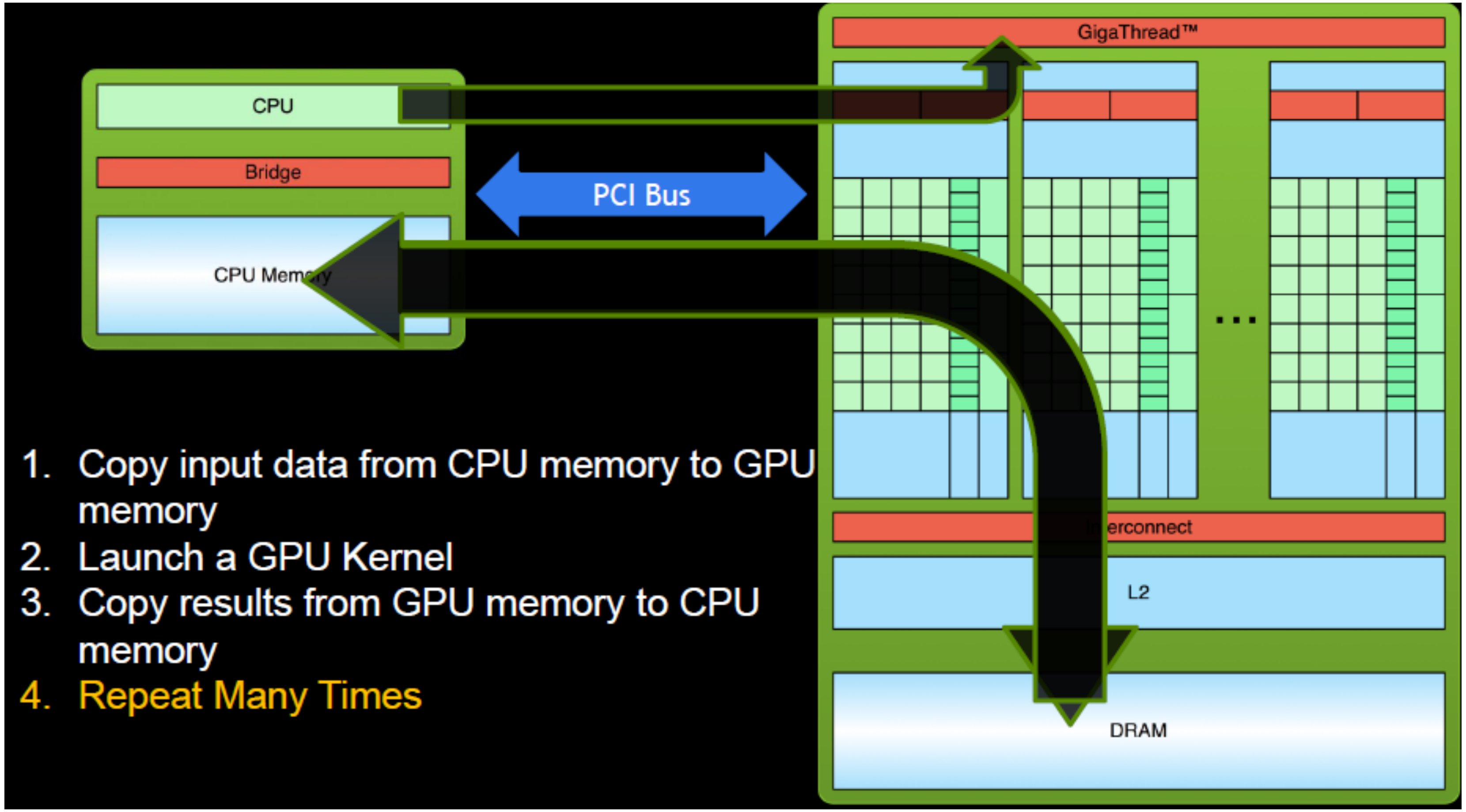
Concurrency

- The ability to perform multiple CUDA operations simultaneously
- Two levels of concurrency in CUDA C programming
  - Kernel level concurrency:
    - A single task, or kernel, is executed in parallel by many threads on the GPU
  - Grid level concurrency:
    - Multiple kernel launches are executed simultaneously on a single device

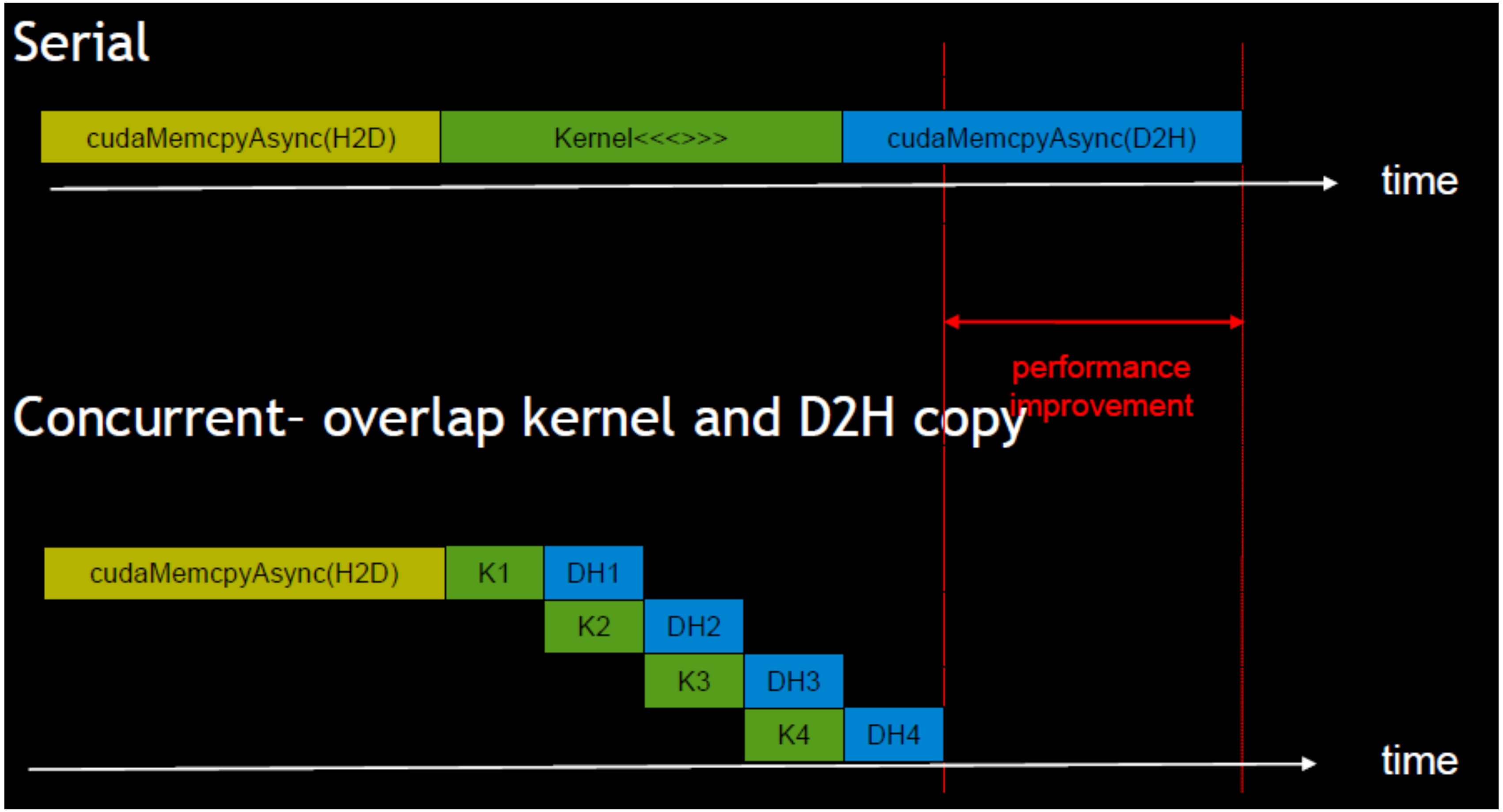
CUDA Streams

- 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 encapsulated in a stream
  - E.g., host-device data transfer, kernel launches, and etc
- Operations within the same stream are ordered (FIFO) and cannot overlap
- Operations in different streams are unordered and can overlap

Serial Processing Flow without Streams



Concurrent Processing Flow with Streams



Streams and Concurrency

- All CUDA operations (both kernels and data transfers) either explicitly or implicitly run in a stream
  - Implicitly declared stream (**NULL stream**): default stream
  - Explicitly declared stream (**non-NULL stream**)
- **Asynchronous, stream-based** kernel launches and data transfers enable four types of concurrency:
  - Overlapped host computation and device computation
  - Overlapped host computation and host-device data transfer
  - Overlapped host-device data transfer and device computation
  - Concurrent device computation

NULL Stream

- Consider the following code using NULL stream:

```
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
kernel<<<grid, block>>>(d_a);
cudaMemcpy(h_a, d_a, size, cudaMemcpyDeviceToHost);
```

- From the device perspective
  - All three operations are executed in order on the stream
  - No awareness of any other host operations being performed
- Form the host perspective
  - Each data transfer is synchronous
  - Kernel launch is asynchronous => overlap device and host computation

```
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
Kernel<<<grid, block>>>(d_a);
anyCPUfunction();
cudaMemcpy(h_a, d_a, size, cudaMemcpyDeviceToHost);
```

Non-NULL Stream

- Non-NULL streams in CUDA are declared, created, and destroyed in host code as follows:
  - **cudaStream_t stream;** // Declare a stream handle
  - **cudaStreamCreate(&stream);** // Allocate a stream
  - **cudaStreamDestroy(stream);** // Deallocate a stream
- To issue data transfer to non-NULL stream
  - **cudaMemcpyAsync(d_a, h_a, size, cudaMemoryHostToDevice, stream)**
  - **cudaMemcpyAsync(h_a, d_a, size, cudaMemoryDeviceToHost, stream)**
- To launch a kernel to non-NULL stream
  - **Kernel<<<grid, block, sharedMemSize, stream>>>(d_a);**

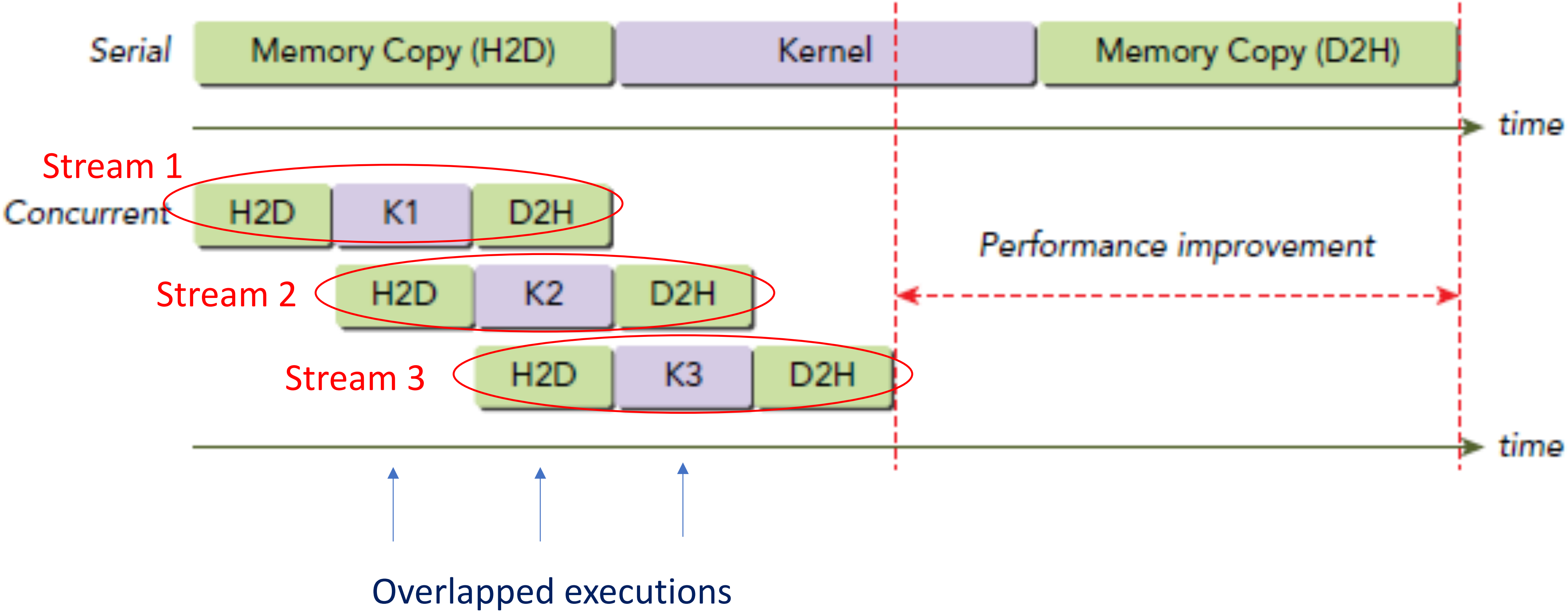
Synchronize and Query in Non-NULL Stream

- All operations in non-NULL streams are non-blocking with respect to the host
- Sometimes you need to synchronize the operations with the host
  - **cudaStreamSynchronize(stream);** => blocks host
- Sometimes you want to check if all operations in a stream have completed, but does not want to block the host if they have not completed
  - **cudaStreamQuery(stream);** => does not block host
    - Returns cudaSuccess if all operations are complete
    - Returns cudaErrorNotReady otherwise

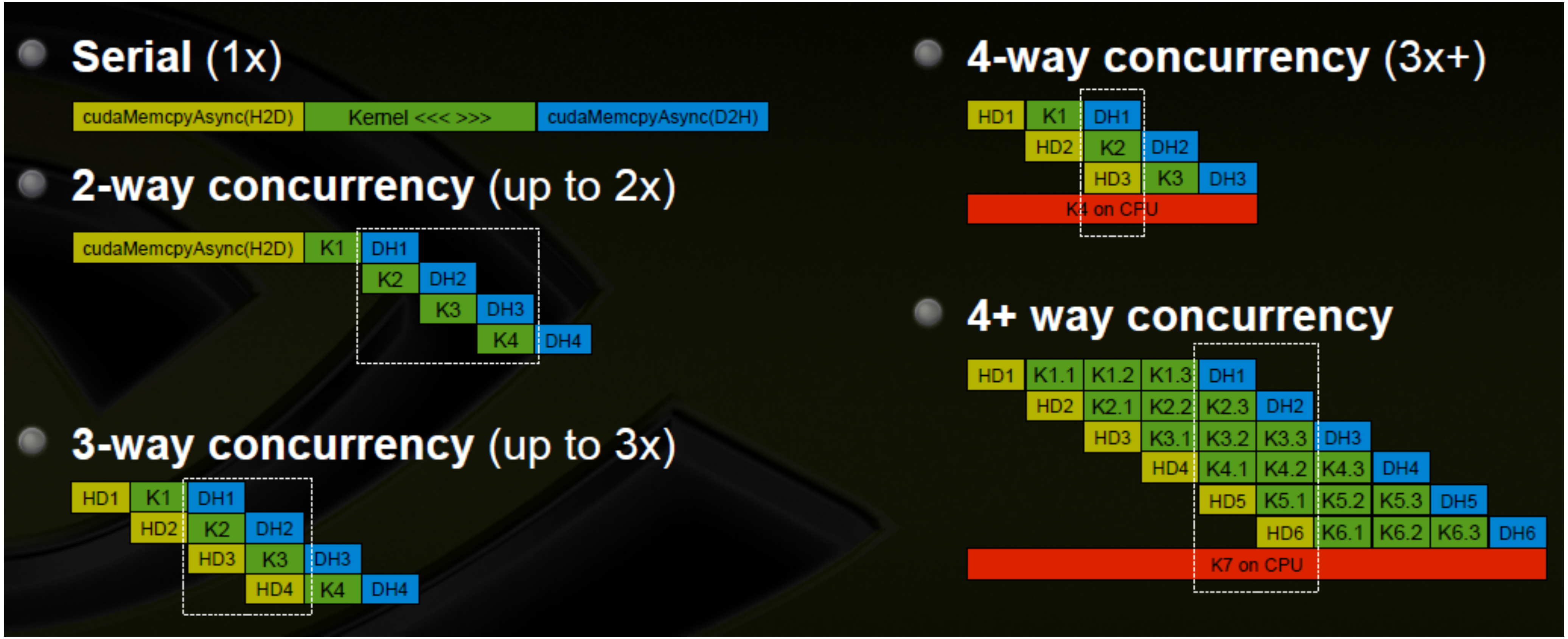
A Common Pattern for Dispatching CUDA Operations to Multiple Streams

```
for (int i = 0; i < nStreams; i++) {
  int offset = i * bytesPerStream;
  cudaMemcpyAsync(&d_a[offset], &h_a[offset], bytePerStream, cudaMemcpyHostToDevice, streams[i]);
  kernel<<<grid, block, 0, streams[i]>>>(&d_a[offset]);
  cudaMemcpyAsync(&h_a[offset], &d_a[offset], bytePerStream, cudaMemcpyDeviceToHost, streams[i]);
}

for (int i = 0; i < nStreams; i++) {
  cudaStreamsSynchronize(streams[i]);
}
```

Example: A Simple Timeline of CUDA Operations Using Three Streams



Number of Concurrency



Conditions to Be Satisfied When Using Streams to Overlap Device Execution with Data Transfer

- First, the device must support a feature known as **device overlap**
  - A GPU supporting device overlap possesses the capacity to simultaneously execute a CUDA C kernel while performing a copy between device and host memory

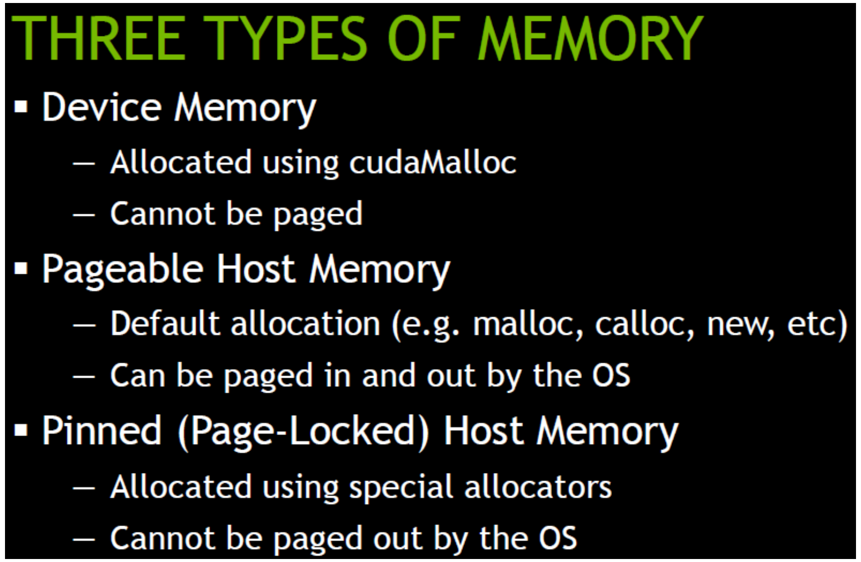
```
cudaDeviceProp  prop;
int whichDevice;
HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
if (!prop.deviceOverlap) {
    printf( "Device will not handle overlaps, so no speed up from streams\n" );
    return 0;
}
```

Conditions to Be Satisfied When Using Streams to Overlap Device Execution with Data Transfer

- Second, The kernel execution and the data transfer to be overlapped must both occur in different, non- NULL streams

Conditions to Be Satisfied When Using Streams to Overlap Device Execution with Data Transfer

- Third, the host memory involved in data transfer must be **pinned (page-locked, non-pageable) memory:**
  - Cannot be swapped (paged) out by the OS
  - Transferred using the host CPU
  - Transferred using the direct memory access , can reach higher bandwidths for large transfers
  - Has higher overhead for allocation



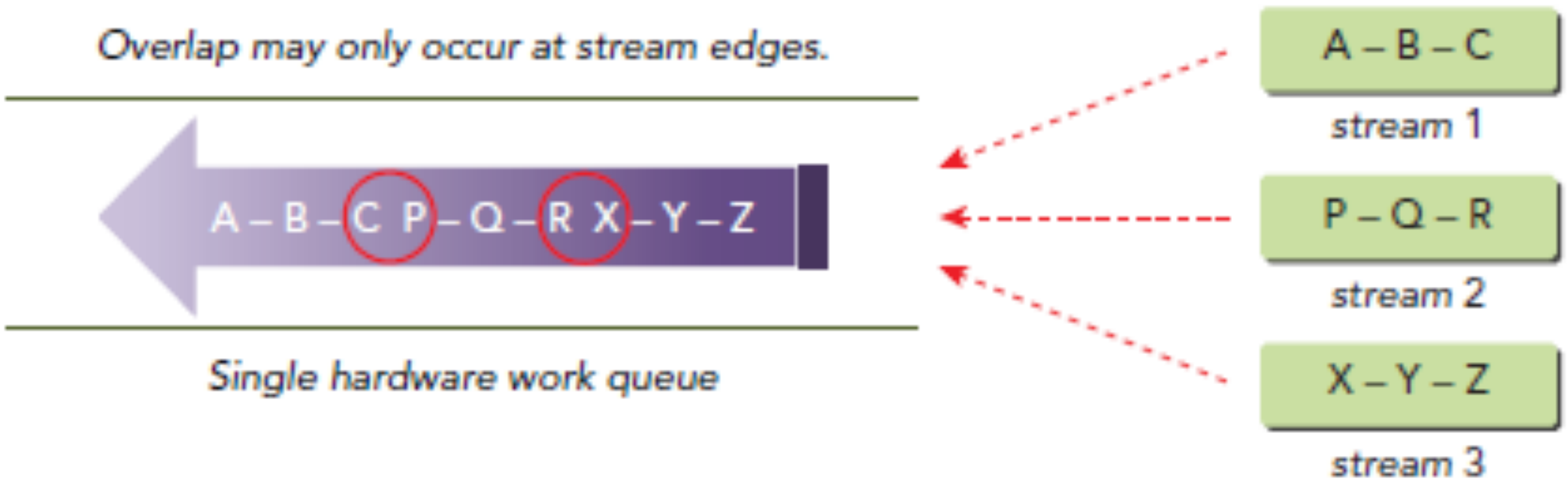
- Allocating pinned memory
  - cudaMallocHost()
  - cudaHostAlloc()
- Free allocated memory
  - cudaFreeHost()

Stream Scheduling

- Kernel and copy engine (possibly x2) have different queues
  - Fermi hardware has 3 queues
    - 1 Compute Engine queue
    - 2 Copy Engine queues – one for H2D and one for D2H
- CUDA operations are dispatched to hardware in sequence they were issued
  - CUDA operations are placed in the relevant queue
  - Stream dependencies between engine queues are maintained
  - Stream dependencies within an engine queue are lost
- A CUDA operation is dispatched from the engine queue if:
  - Preceding calls in the same stream have completed
  - Preceding calls in the same queue have been dispatched, and
  - Resources are available
- CUDA kernels may be executed concurrently if they are in different streams

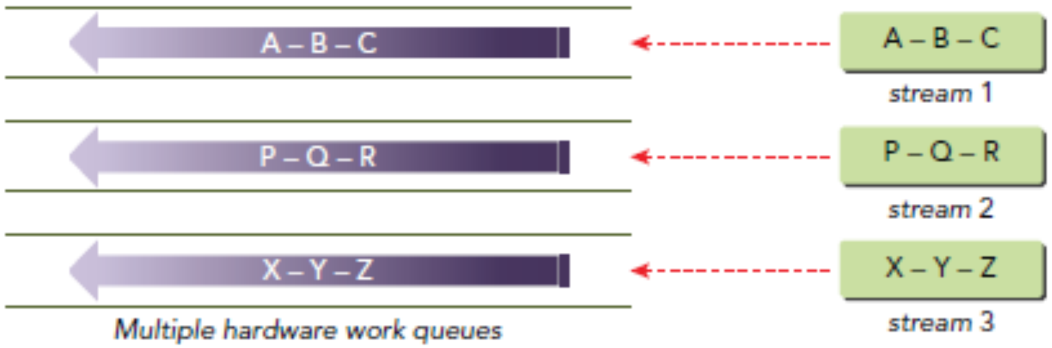
False Dependencies

- All streams are multiplexed into a single hardware work queue. The single pipeline may result in a false dependency for the preceding streams to block successive streams
- **Note a blocked operation blocks all other operations in the queue, even in other streams**



Hyper-Q

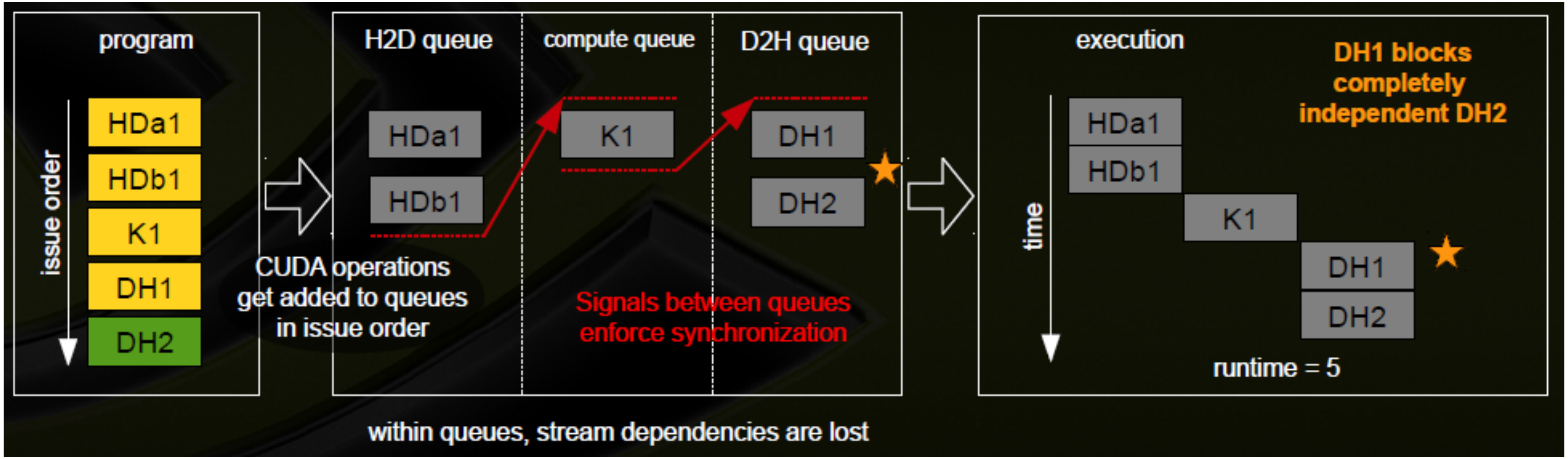
- False dependencies are reduced in the Kepler family of GPUs using multiple hardware work queues, a technology called **Hyper-Q**
- Hyper-Q allows multiple CPU threads or processes to launch work on a single GPU simultaneously by maintaining multiple hardware-managed connections between the host and the device
- deviceProp.**concurrentKernels**




```
int dev = 0;
cudaDeviceProp deviceProp;
CHECK(cudaGetDeviceProperties(&deviceProp, dev));
printf("> Using Device %d: %s\n", dev, deviceProp.name);
CHECK(cudaSetDevice(dev));

// check if device support hyper-q
if (deviceProp.major < 3 || (deviceProp.major == 3 && deviceProp.minor < 5))
{
    if (deviceProp.concurrentKernels == 0)
    {
        printf("> GPU does not support concurrent kernel execution (SM 3.5 "
                "or higher required)\n");
        printf("> CUDA kernel runs will be serialized\n");
    }
    else
    {
        printf("> GPU does not support HyperQ\n");
        printf("> CUDA kernel runs will have limited concurrency\n");
    }
}
```

Example – Blocked Queue

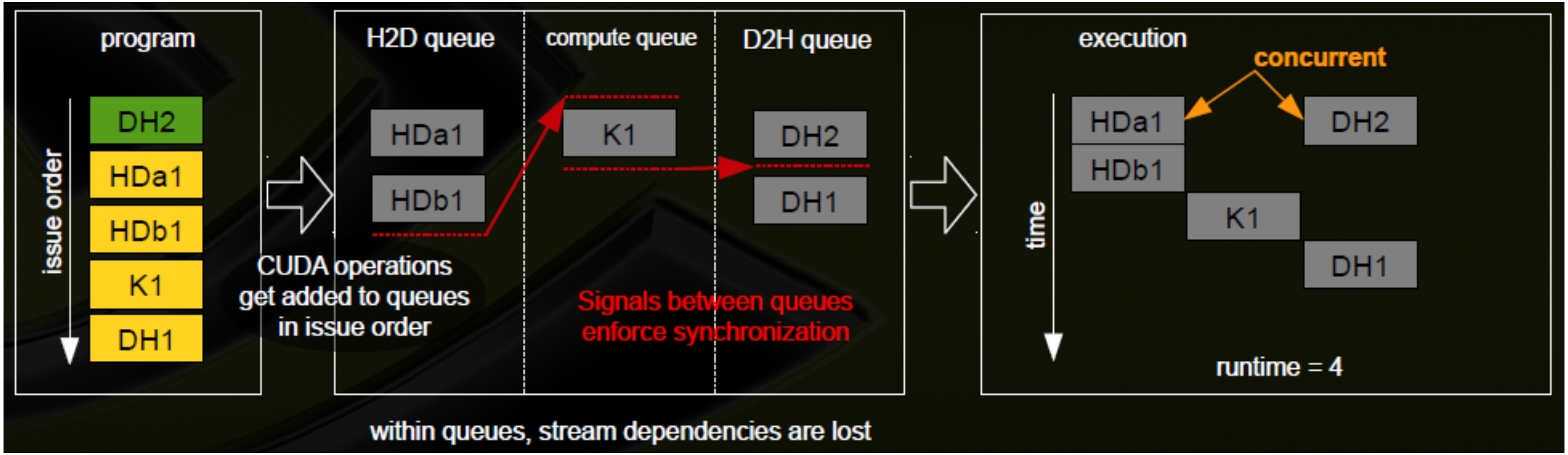
- Two streams, stream 1 is issued first
  - Stream 1: HDa1, HDb1, K1, DH1
  - Stream 2: DH2 (completely independent of stream 1)



Example – Blocked Queue

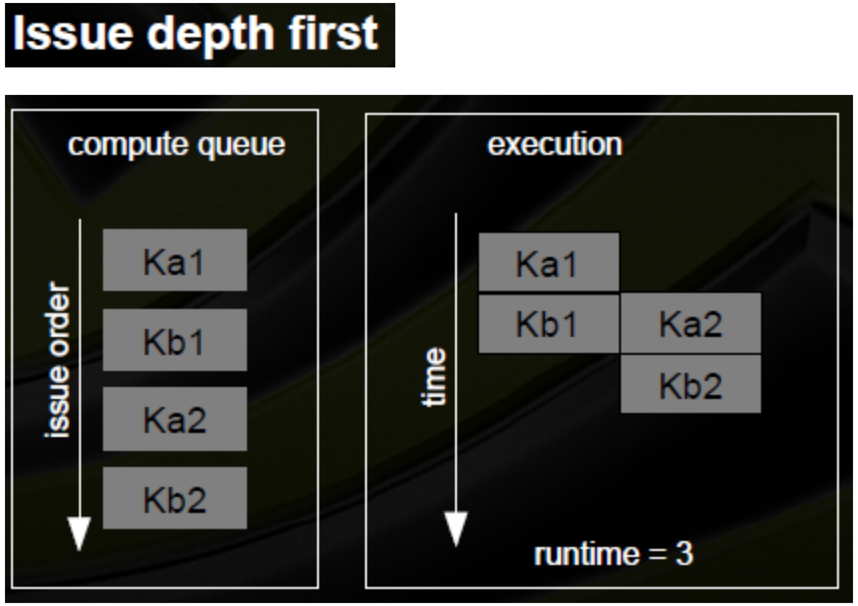
- Two streams, stream 2 is issued first
  - Stream 1: HDa1, HDb1, K1, DH1
  - Stream 2: DH2

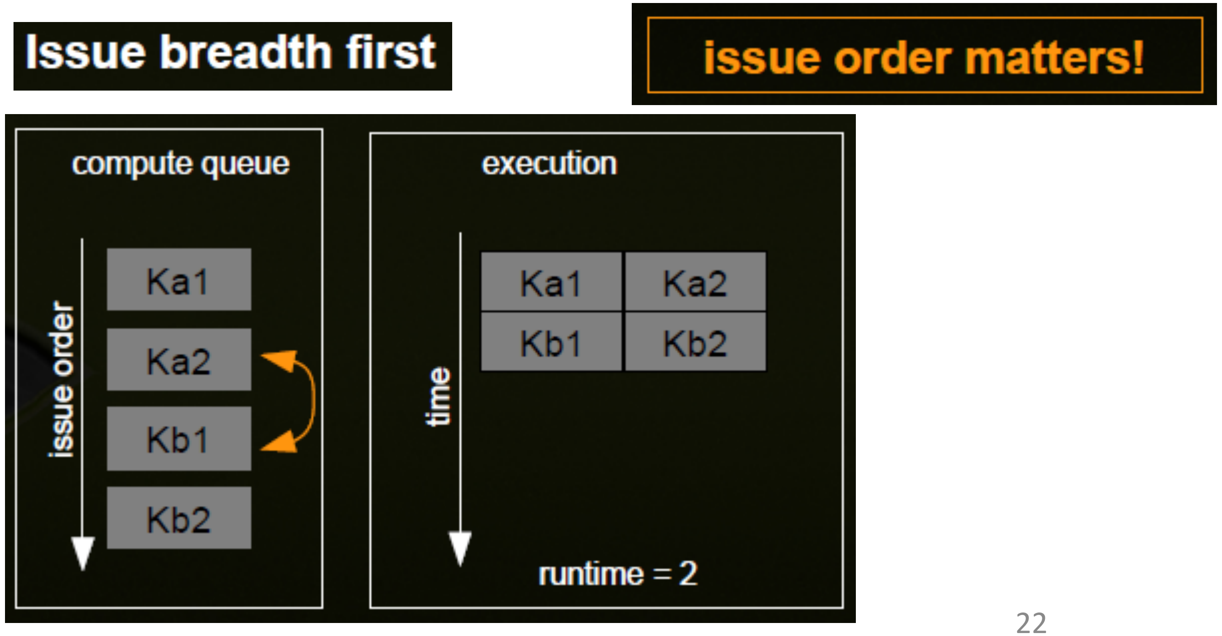
issue order matters!



Example – Blocked Kernel

- Two streams – just issuing CUDA kernels
  - Stream 1: Ka1, Kb1
  - Stream 2: Ka2, Kb2
  - Kernels are **similar size**, fill ½ of the SM resources



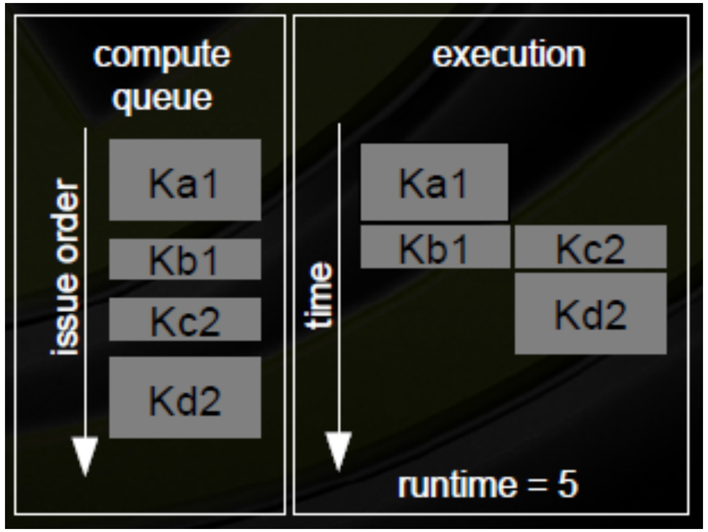


Example – Optimal Concurrency Can Depend on Kernel Execution Time

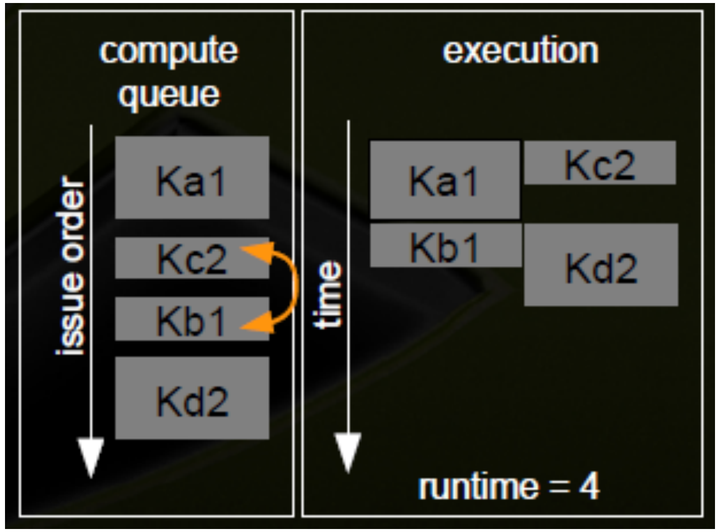
- Two streams – just issuing CUDA kernels – but kernels are **different “sizes”**
  - Stream 1: Ka1 {2}, Kb1 {1}
  - Stream 2: Kc2 {1}, Kd2 {2}
  - Kernels fill ½ of the SM resources

issue order matters!
execution time matters!

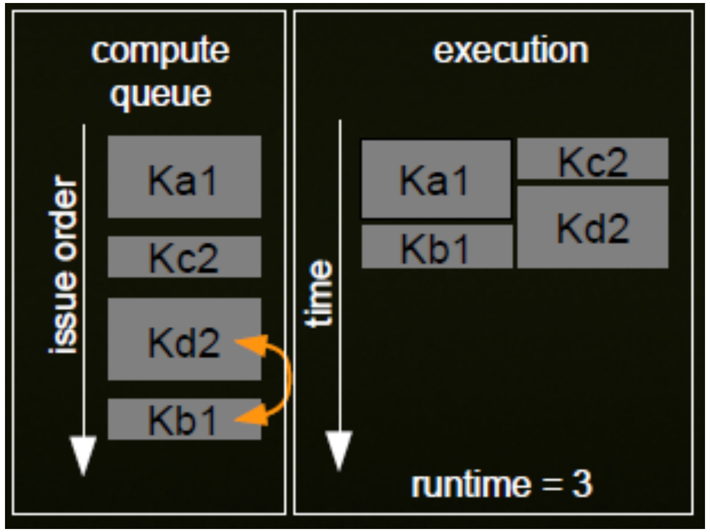
Issue depth first



Issue breadth first



Custom



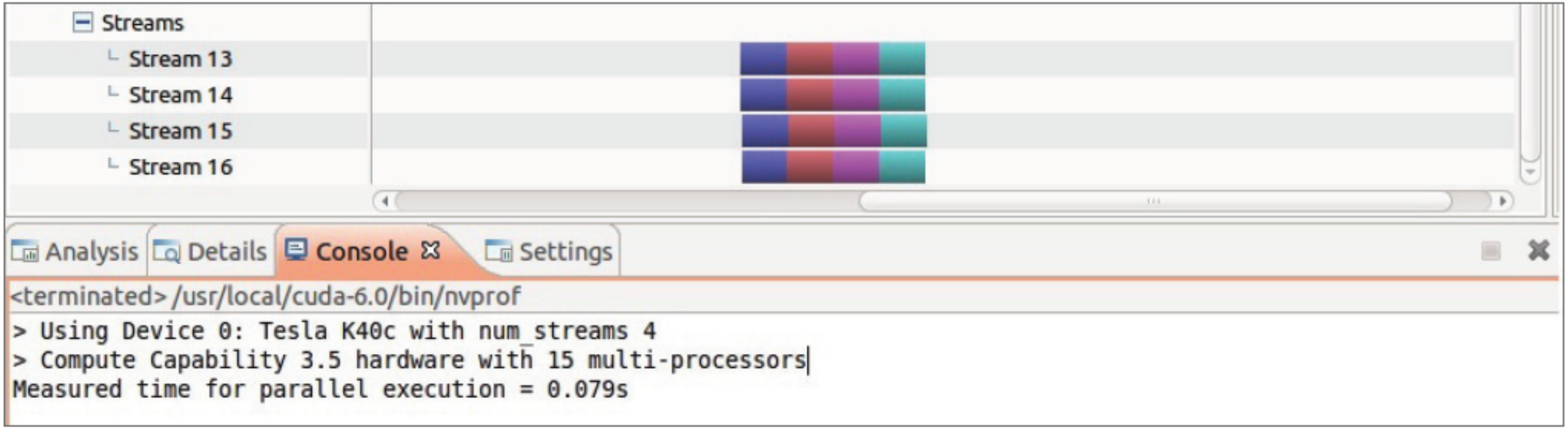
NVIDIA Visual Profiler (nvvp)

- Allows you to visualize and optimize the performance of your application
- Displays a timeline of your application's activity on both the CPU and GPU so that you can identify opportunities for performance improvement
- Analyzes your application to detect potential performance bottlenecks and direct you on how to take action to eliminate or reduce those bottlenecks
- **$nvvp ./a.out**
- A timeline will contain a **Stream** row for each stream used by the application (including both the default stream and any application created streams). Each interval in a **Stream** row represents the duration of a memcpy or kernel execution performed on that stream

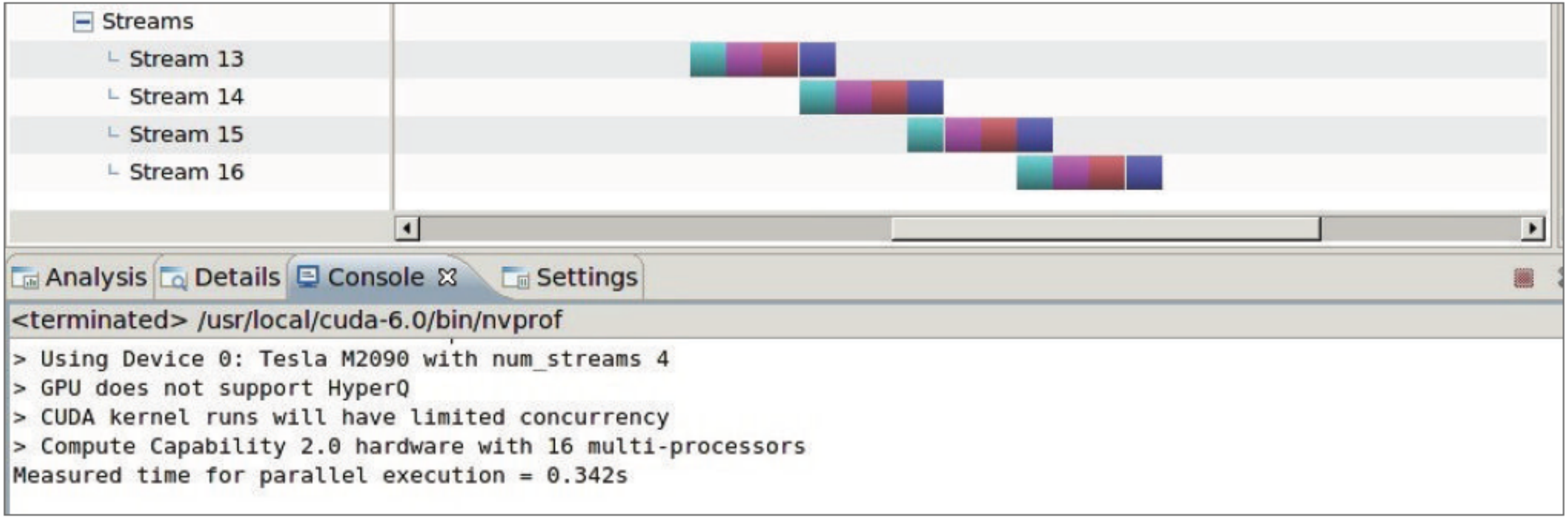
Visualizing Concurrent Kernel Executions on Tesla K40

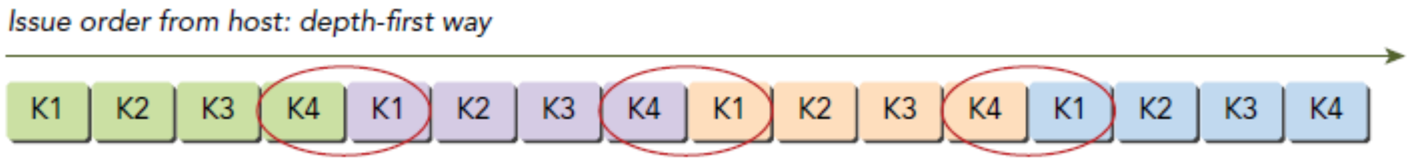
```
for (int i = 0; i < n_streams; i++) {
  kernel_1<<<grid, block, 0, streams[i]>>>();
  kernel_2<<<grid, block, 0, streams[i]>>>();
  kernel_3<<<grid, block, 0, streams[i]>>>();
  kernel_4<<<grid, block, 0, streams[i]>>>();
}
```

```
__global__ void kernel_1() {
  double sum = 0.0;
  for (int i = 0; i < N; i++) {
    sum = sum + tan(0.1) * tan(0.1);
  }
}
```



Demonstrating False Dependencies with Depth-First Assignment on Fermi GPUs

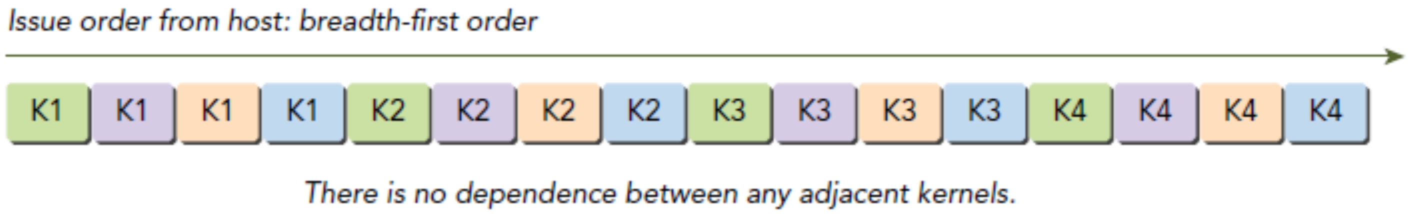


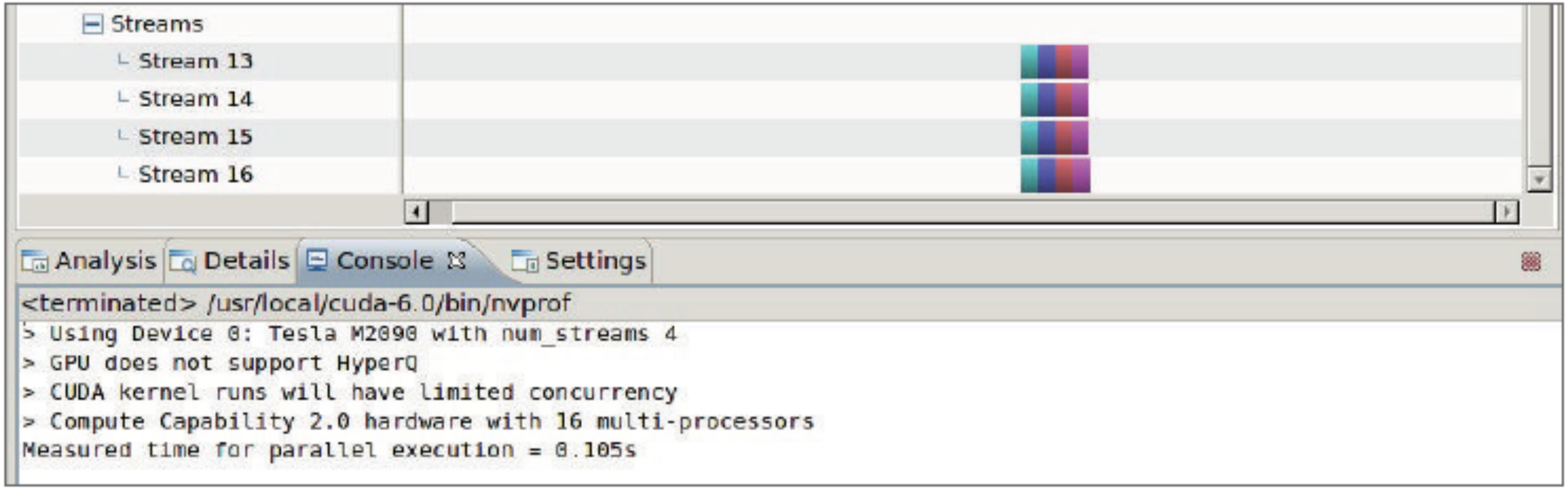


Only the three stream edges are independent.

Avoid False Dependencies on Fermi GPUs with Breadth-First Assignment

```
// dispatch job with breadth first way
for (int i = 0; i < n_streams; i++)
  kernel_1<<<grid, block, 0, streams[i]>>>();
for (int i = 0; i < n_streams; i++)
  kernel_2<<<grid, block, 0, streams[i]>>>();
for (int i = 0; i < n_streams; i++)
  kernel_3<<<grid, block, 0, streams[i]>>>();
for (int i = 0; i < n_streams; i++)
  kernel_4<<<grid, block, 0, streams[i]>>>();
```





Coding Examples

- Pinned-memory
- Vector addition
  - Single stream version
  - Double stream with depth-first assignment version
  - Double stream with breadth-first assignment version

Example: Pinned Memory

- cuda_malloc_test() does the memory copy for pageable memory
  - Allocate host memory by malloc()
- cuda_host_alloc_test() does the memory copy for pinned memory
  - Allocate host memory by cudaHostAlloc()

```
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 ) ) );
```

```
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 ) ) );
```

Example: Pinned Memory

- The loops of copy, and stop events for timer are the same inside both cuda_malloc_test() and cuda_host_alloc_test()

```
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 ) );
```

Example: Pinned Memory

- cuda_malloc_test() frees memory by free()
- cuda_host_alloc_test() frees memory by cudaFreeHost()

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

    return elapsedTime;
}
```

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

    return elapsedTime;
}
```

Example: Pinned Memory

```
#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) );

    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) );

    // now try it with 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) );
}
```

```
[jin6@node1733 11_StreamsBasics]$ ./a.out
Time using cudaMalloc:  6688.8 ms
        MB/s during copy up:  3827.3
Time using cudaMalloc:  7377.6 ms
        MB/s during copy down:  3470.0
Time using cudaHostAlloc:  4281.8 ms
        MB/s during copy up:  5978.7
Time using cudaHostAlloc:  4050.1 ms
        MB/s during copy down:  6320.8
```

Example: Vector Addition

- Use a CUDA kernel to take two input buffers of data, a and b
- The kernel compute an average of three values in a and b to produce an output buffer c

```
#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;
    }
}
```

Example: Vector Addition – Single Stream

- Device check and declarations

```
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;

    cudaStream_t    stream;
    int *host_a, *host_b, *host_c;
    int *dev_a, *dev_b, *dev_c;

    // start the timers
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

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

Example: Vector Addition – Single Stream

- Memory allocations and initializations

```
// 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();
}
```

Example: Vector Addition – Single Stream

- Split data, perform kernel operations, and copy result back

```
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
// 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 ) );
```

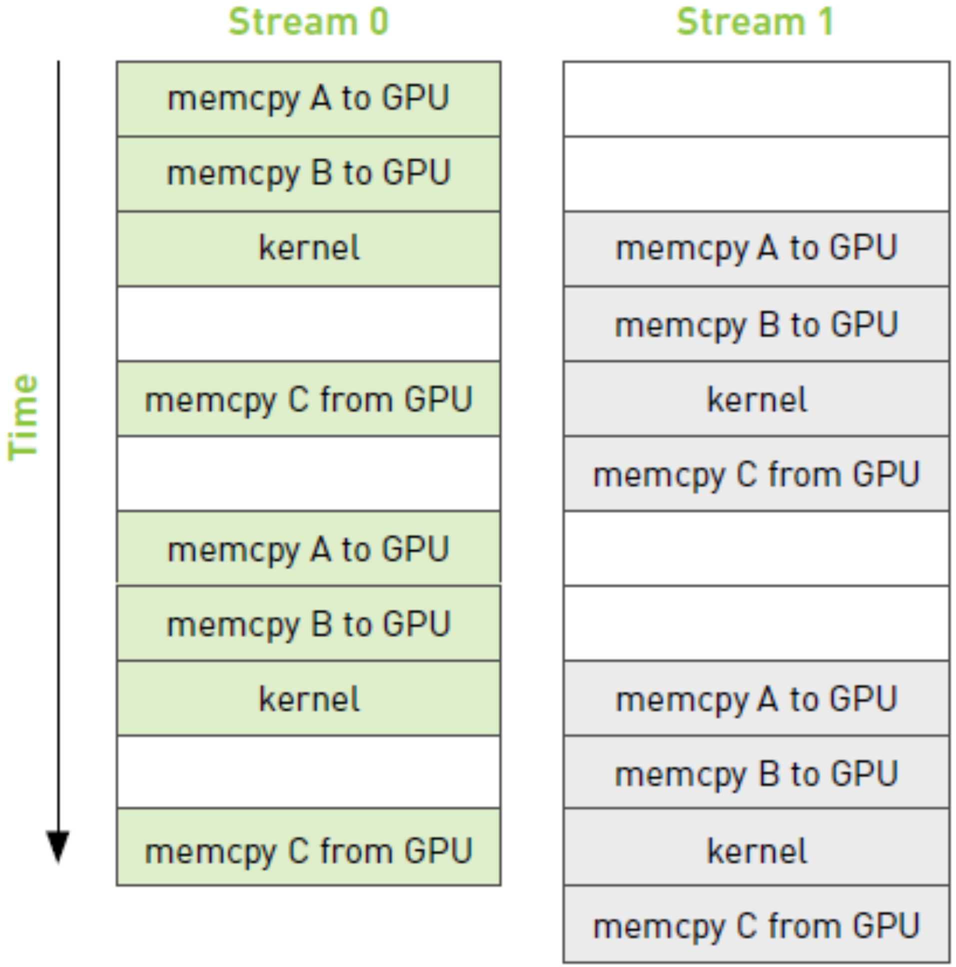
Example: Vector Addition – Single Stream

- Stop timer, collect performance data, free buffers, and destroy stream

```
    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;
}
```

Example: Vector Addition – Double Streams

- The idea underlying this version relies on two things:
  - The “chunked” computation, and
  - The overlap of memory copies with kernel execution.
- Enqueue operations across streams
  - Depth-first
  - Breadth-first



Timeline of intended application execution using two independent streams. (Calls to cudaMemcpyAsync() are abbreviated to “memcpy”.)

Example: Vector Addition – Double Streams (Depth-first)

```
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;

    cudaStream_t    stream0, stream1;
    int *host_a, *host_b, *host_c;
    int *dev_a0, *dev_b0, *dev_c0;
    int *dev_a1, *dev_b1, *dev_c1;

    // start the timers
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    // initialize the streams
    HANDLE_ERROR( cudaStreamCreate( &stream0 ) );
    HANDLE_ERROR( cudaStreamCreate( &stream1 ) );
```

Example: Vector Addition – Double Streams (Depth-first)

```
    // 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();
    }
```

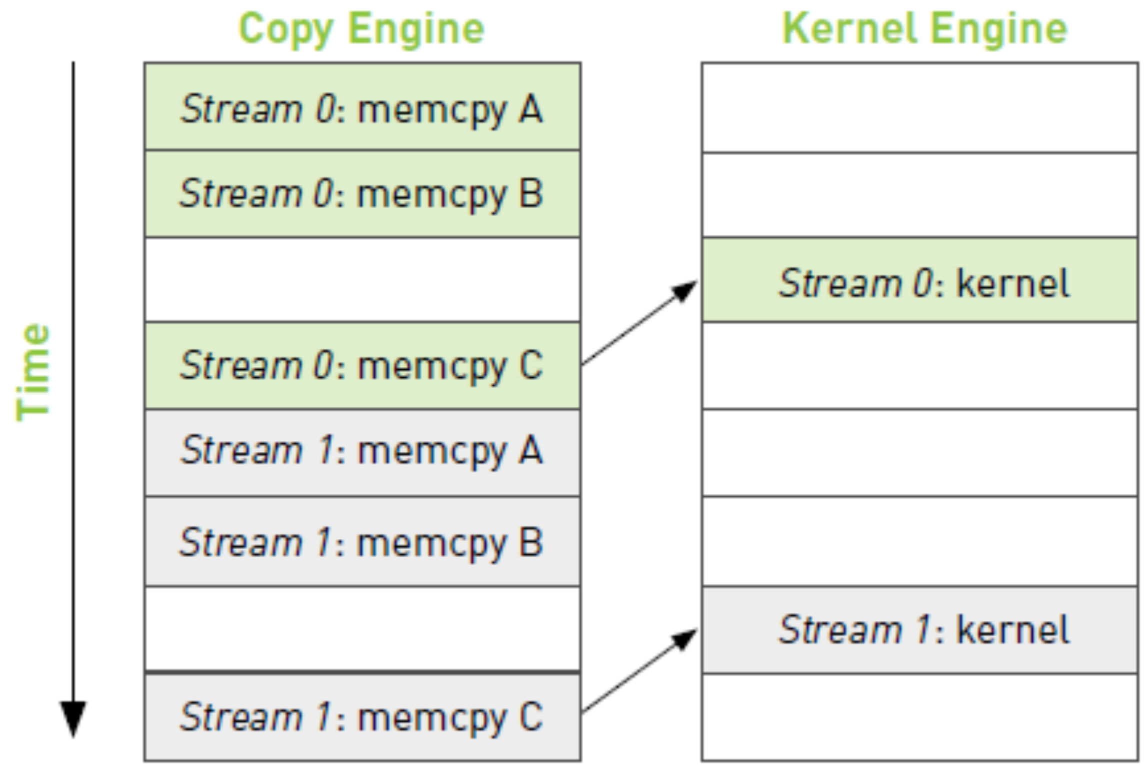
Example: Vector Addition – Double Streams (Depth-first)

```
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
// 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 ) );

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

Example: Vector Addition – Double Streams (Depth-first)

```
    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;
}
```

Example: Vector Addition – Double Streams (Depth-first)

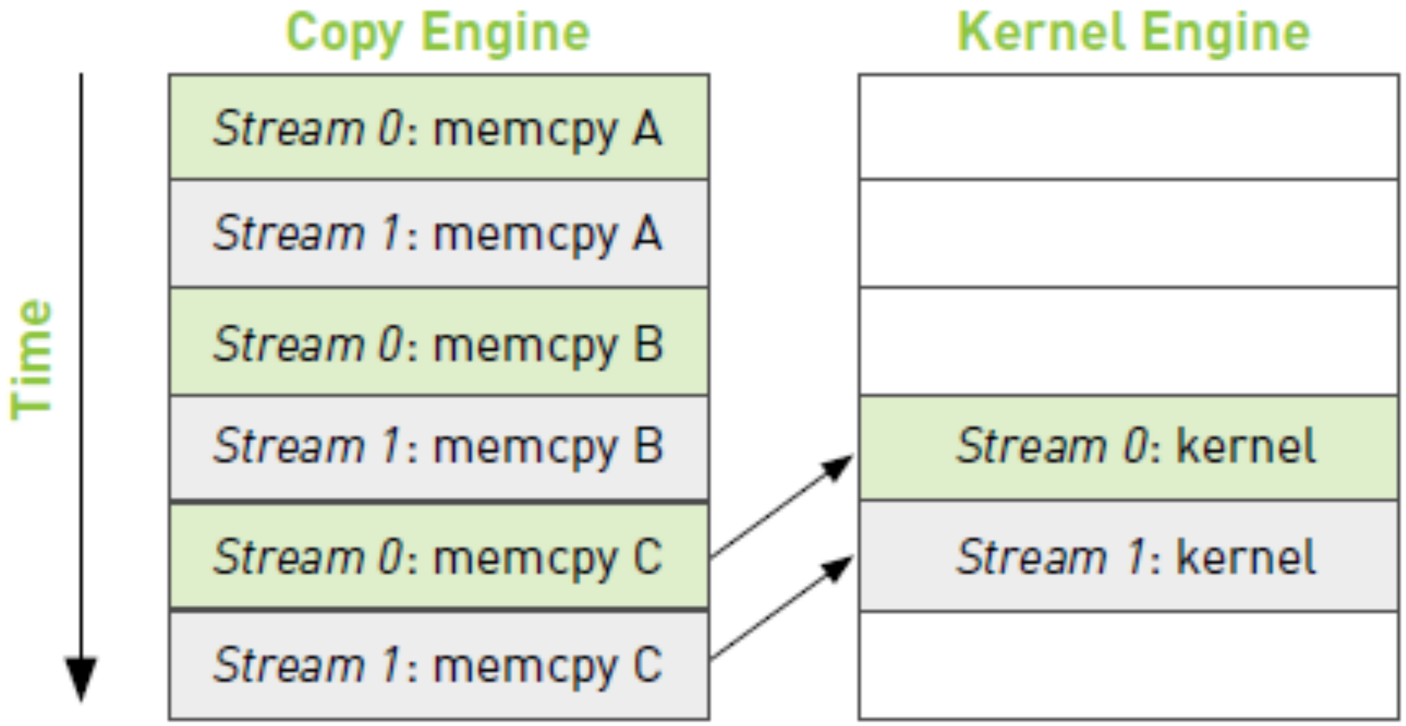


Stream 0's copy of c back to the host depends on its kernel execution completing
Stream 1's completely independent copies of a and b to the GPU get blocked because the GPU's engines execute work in the order it's provided
(Arrow's depicting the dependency of cudaMemcpyAsync() calls on kernel executions)

Example: Vector Addition – Double Streams (Breadth-first)

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

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

Example: Vector Addition – Double Streams (Breadth-first)



Example: Vector Addition – Results

```
[jin6@node1733 11_Streams]$ ./basic_single_stream
Time taken:  48.8 ms
[jin6@node1733 11_Streams]$ ./basic_double_stream_depth
Time taken:  31.0 ms
[jin6@node1733 11_Streams]$ ./basic_double_stream_breadth
Time taken:  31.1 ms
```

Conclusions

- Using two or more CUDA steams enables simultaneous execution of kernel operations
- Asynchronous functions need to be performed with pinned memory allocated by cudaHostAlloc()
- The order in which we add operations to streams affect the capacity to achieve overlapping of memory copies and kernel executions
  - A general guideline is to use breadth-first assignment
- nvvp allows to you connect and view profiling data visually