

CSE 599 I

Accelerated Computing - Programming GPUS

Streams, Events, and Concurrency

Objective

- Learn about when and how to use CUDA streams and events
- Study strategies to maximize concurrency of host and device computation and data transfers

Beyond Kernel-Level Concurrency

- Host Computation / Device Computation
- Device Computation / Host-Device Transfer
- Host Computation / Host-Device Transfer
- Device / Device Computation

A Typical Scenario

1. Copy input data from the host to the device
2. Process the input data
3. Copy the results from the device back to the host

Host / Device Computation Concurrency

This is the easiest type of concurrency to achieve

A common pattern is:

1. A host thread launches a kernel
2. The host thread does some additional work as the kernel executes
3. The host synchronizes with the device and gets the results

SpMV / HYB Revisited

```
void hybridSpMV(const float * A, const int M, const int N const float * x, float * y) {  
  
    float * d_y;  
    float * d_x;  
    float * y_ELL;  
    SparseMatrixELL d_A_ELL;  
    SparseMatrixCOO A_COO;  
  
    // build sparse matrix representations, allocate / initialize host and device memory  
    ...  
  
    // launch ELL kernel  
    SpMV_ELL_kernel<<<(A.M + 127)/128,128>>>(d_A_ELL, d_x, d_y);  
  
    // perform host computation  
    SpMV_COO(A_COO, x, y);  
  
    // copy device result back to host  
    cudaMemcpy(y_ELL, d_y, A.N * sizeof(float) );  
  
    for (int i = 0; i < A.N; ++i) {  
        y[i] += y_ELL[i];  
    }  
  
}
```

These happen concurrently

The host blocks here until the kernel launch is complete

Introducing CUDA Streams

A CUDA stream is an ordered sequence of kernel launches and CUDA runtime API calls that are all executed sequentially, with no overlap (i.e. FIFO queue)

Work in different CUDA streams can (sometimes) be performed in parallel

Every kernel launch and CUDA runtime API call is in some stream

Explicitly vs. Implicitly Declared Streams

A kernel launch or runtime API call that does not explicitly declare a stream is issued in the default stream

Non-default streams must be explicitly created and managed

Achieving coarse-grained concurrency requires the use of explicitly declared streams

Synchronization in the Default Stream

All of these will issue in the default stream

```
cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);  
cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);
```

```
vectorAdditionKernel<<<grid, block>>>(d_A, d_B, d_C, N);
```

```
cudaMemcpy(h_C, d_C, N * sizeof(float), cudaMemcpyHostToDevice);
```

Synchronous:
These will begin when all prior jobs in the stream are complete and return when the copy is done

Asynchronous:
This will return immediately

Synchronous:
This will begin when the kernel is complete and return when the copy is done

Asynchronous Data Transfer

The host will return immediately from all four of these calls:

```
cudaMemcpyAsync(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);  
cudaMemcpyAsync(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);  
  
vectorAdditionKernel<<<grid, block>>>(d_A, d_B, d_C, N);  
  
cudaMemcpyAsync(h_C, d_C, N * sizeof(float), cudaMemcpyHostToDevice);
```

One caveat: `cudaMemcpyAsync()` requires pinned memory

Explicit Streams

`cudaError_t`

```
cudaMemcpyAsync(void * dst, const void * src, size_t count, cudaMemcpyKind kind,  
               cudaStream_t stream = 0);
```

This is a common parameter for asynchronous runtime API calls

Using the default value for `stream` (i.e. `0`) issues the call in the default stream

Creating Explicit Streams

```
cudaStream_t stream;
```

```
cudaStreamCreate(&stream);
```

```
// use stream as a parameter to asynchronous API calls
```

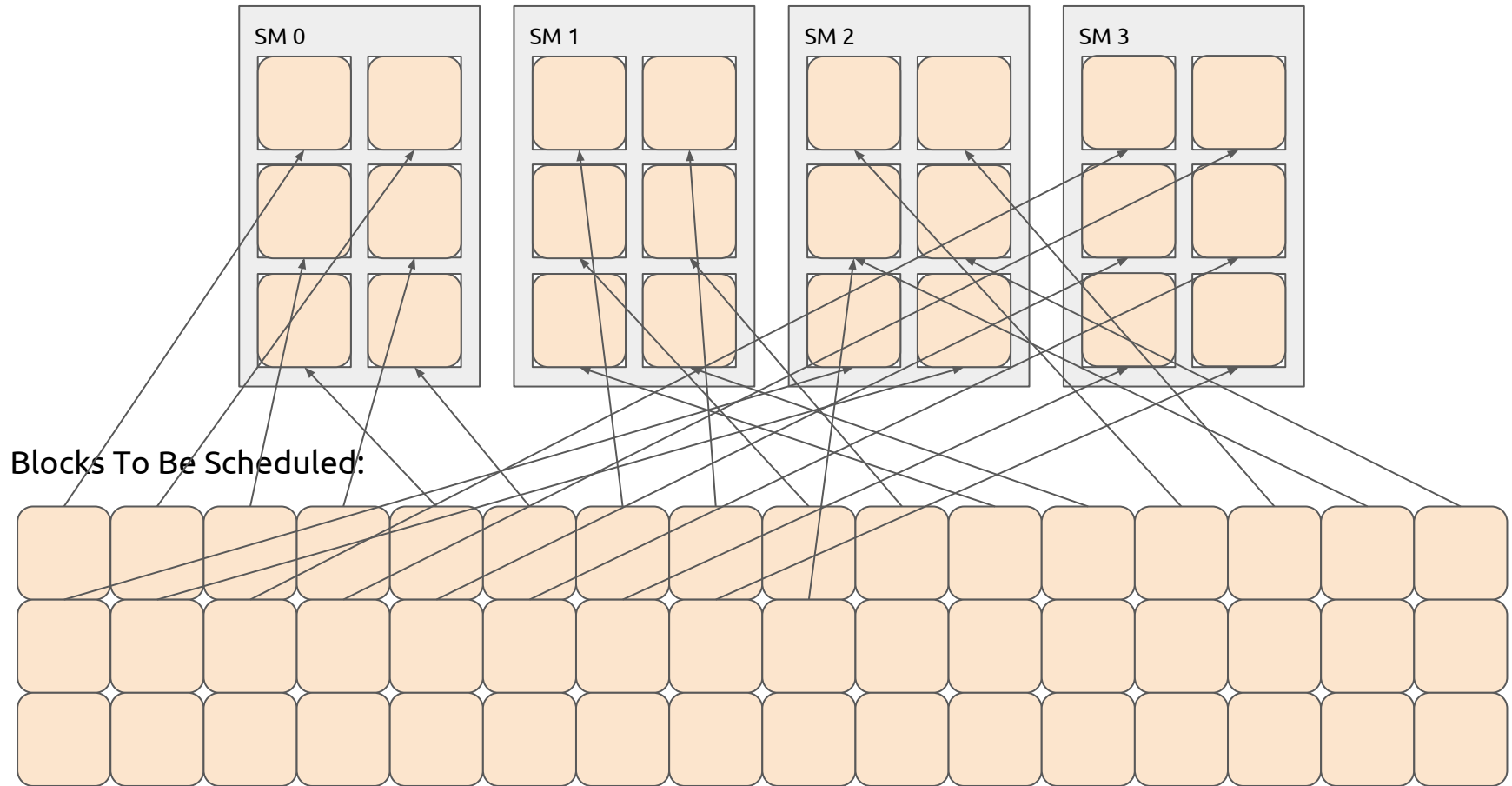
```
...
```

```
cudaMemcpyAsync(...,stream);
```

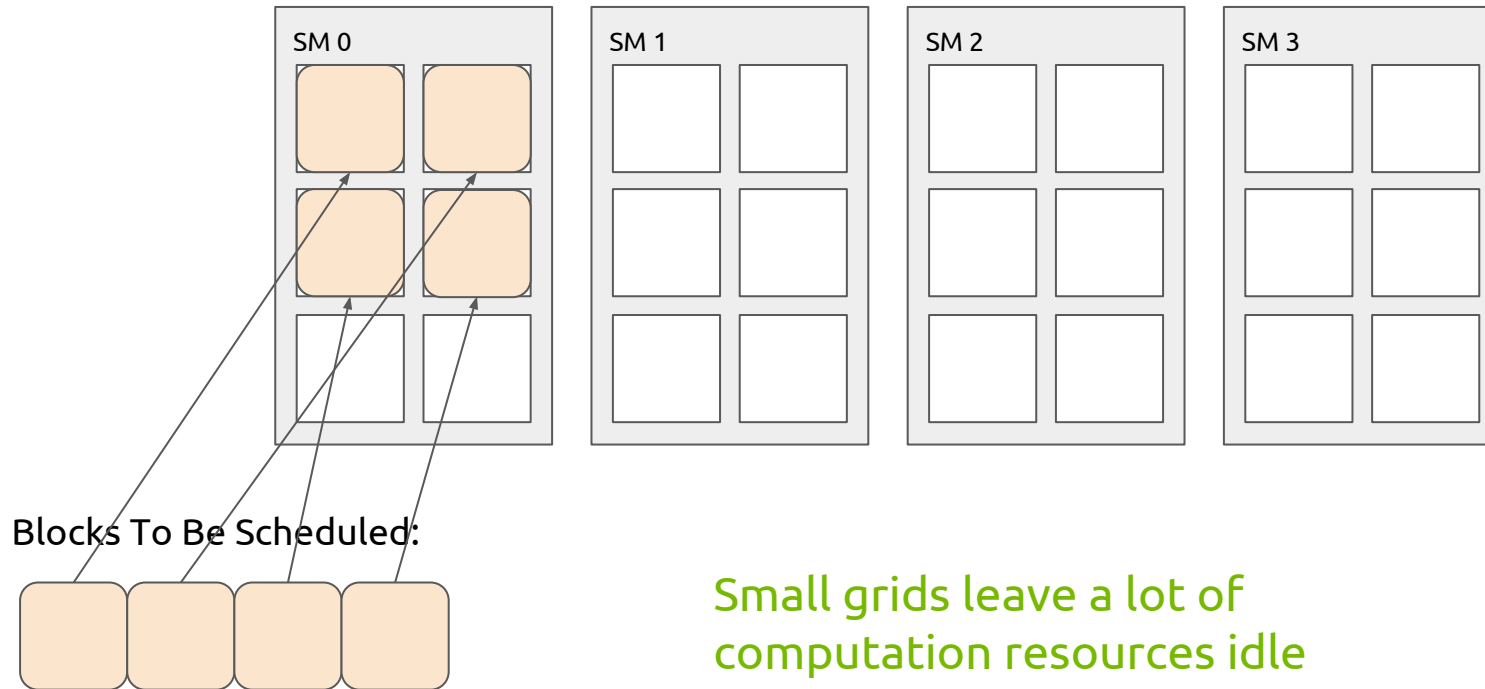
```
...
```

```
cudaStreamDestroy(stream);
```

Maintaining Occupancy



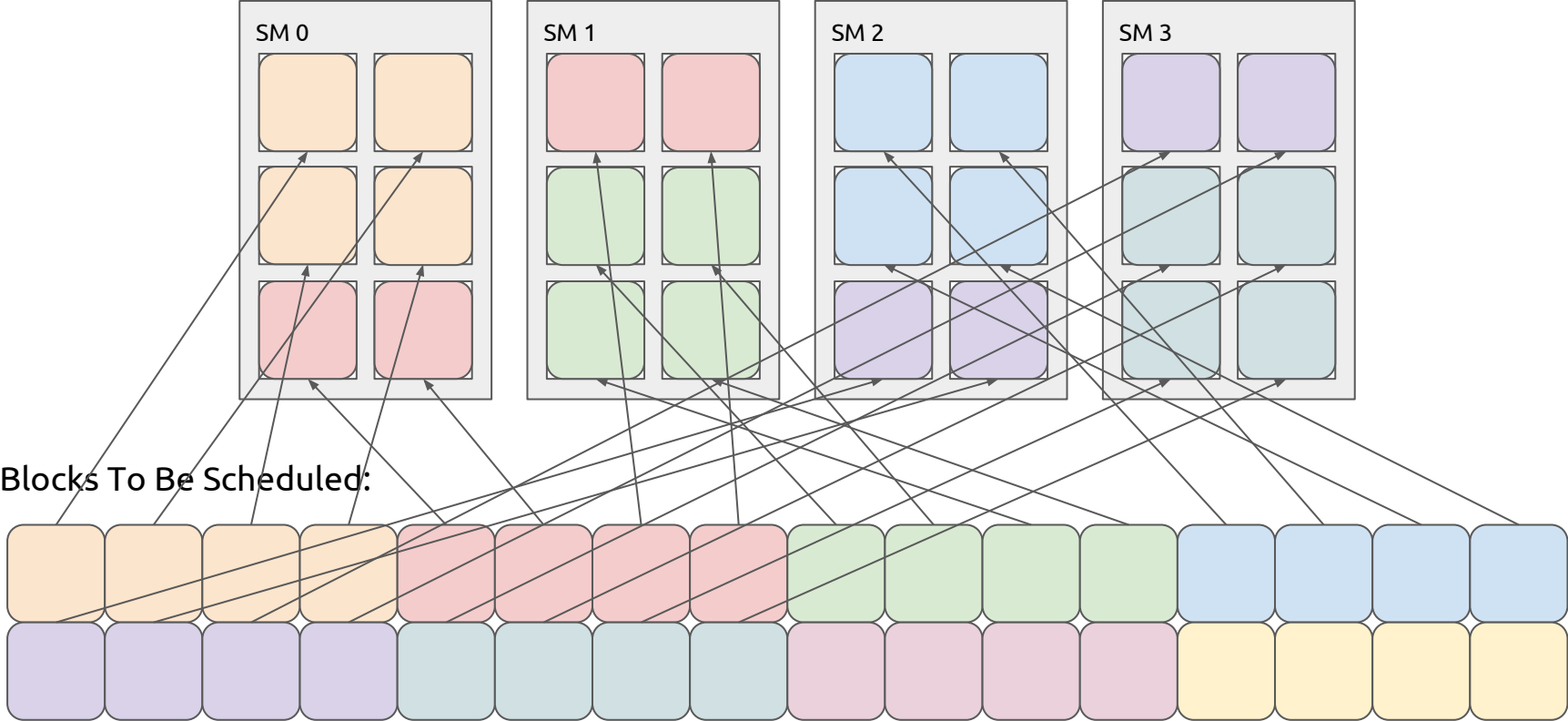
Maintaining Occupancy



Increased Concurrency with Streams

```
void batchVectorAdd(float * * As, float * * Bs, float * * Cs, int * Ns, int count) {  
  
    cudaStream_t * streams = new cudaStream_t[count];  
    float * * d_As = new float *[count];  
    float * * d_Bs = new float *[count];  
    float * * d-Cs = new float *[count];  
  
    // make sure host pointers are in pinned memory, allocate device memory  
    ...  
  
    for (int i = 0; i < count; ++i) {  
  
        // set up grid and block  
        ...  
  
        cudaStreamCreate(&streams[i]);  
        cudaMemcpyAsync(d_As[i], As[i], Ns[i] * sizeof(float), cudaMemcpyHostToDevice, streams[i]);  
        cudaMemcpyAsync(d_Bs[i], Bs[i], Ns[i] * sizeof(float), cudaMemcpyHostToDevice, streams[i]);  
        vectorAddKernel<<<grid,block,0,streams[i]>>>(d_As[i], d_Bs[i], d-Cs[i], Ns[i]);  
        cudaMemcpyAsync(Cs[i], d-Cs[i], Ns[i] * sizeof(float), cudaMemcpyDeviceToHost, streams[i]);  
  
    }  
  
    for (int i = 0; i < count; ++i) {  
        cudaStreamSynchronize(streams[i]);  
        cudaStreamDestroy(streams[i]);  
    }  
  
    delete [] streams; delete [] d_As; delete [] d_Bs; delete [] d-Cs;  
  
}
```

Device / Device Computation Concurrency



Device Computation / Host-Device Transfer Concurrency

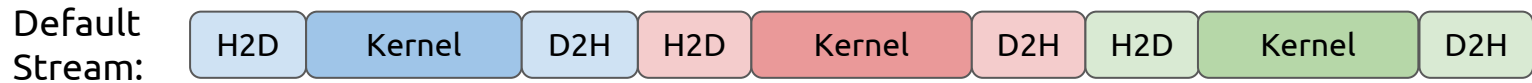
Modern CUDA devices have two copy engines

One host-to-device transfer and one device-to-host transfer can happen concurrently

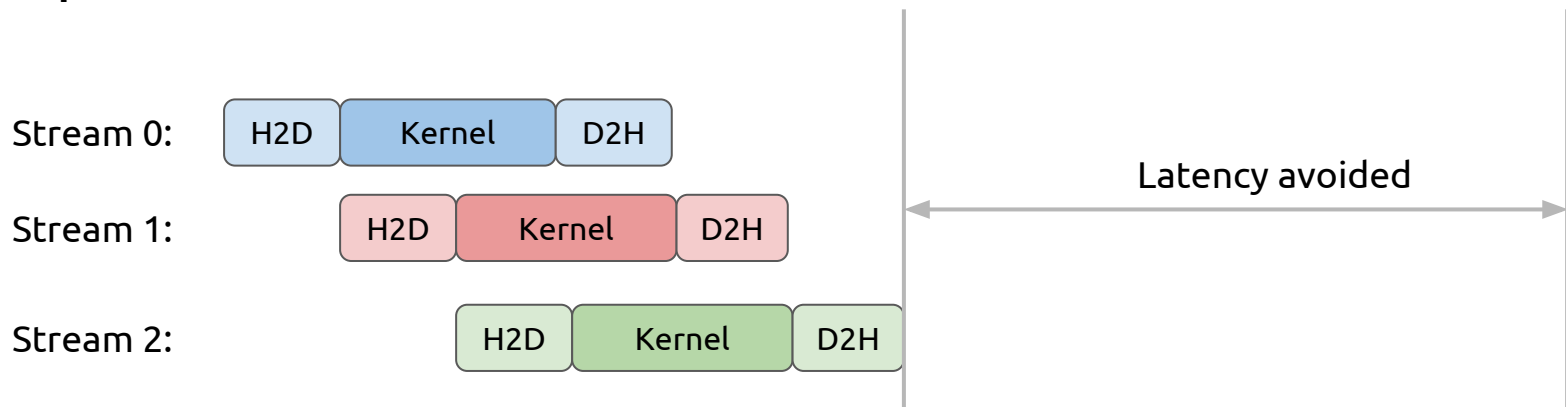
batchVectorAdd Device Execution Timeline

Scenario A: Kernel execution exceeds data transfer time, computation resources sufficient to execute kernels in parallel

Without Explicit Streams:



With Explicit Streams:



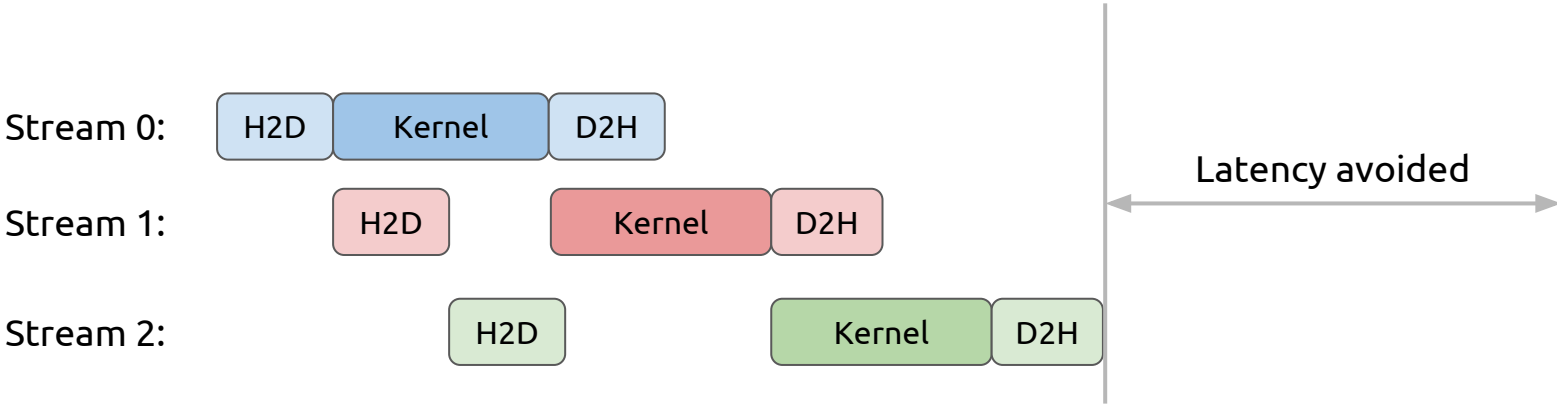
batchVectorAdd Device Execution Timeline

Scenario B: Kernel execution exceeds data transfer time, computation resources insufficient to execute kernels in parallel

Without Explicit Streams:



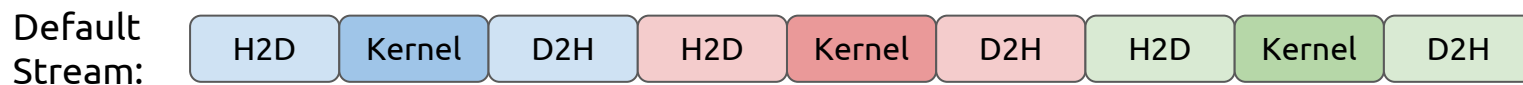
With Explicit Streams:



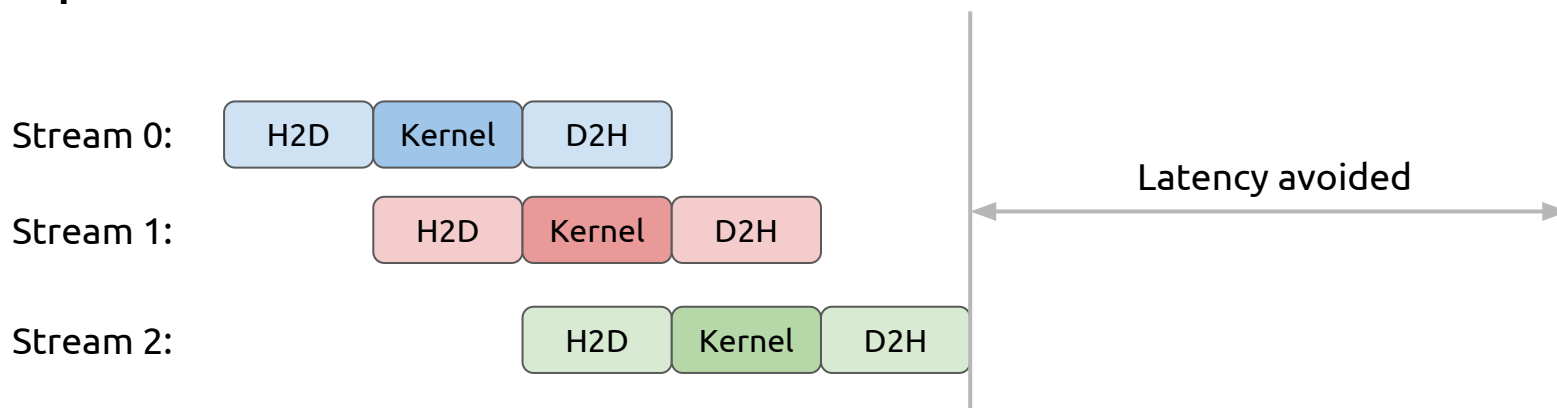
batchVectorAdd Device Execution Timeline

Scenario C: Kernel execution and data transfer time are roughly equivalent

Without Explicit Streams:



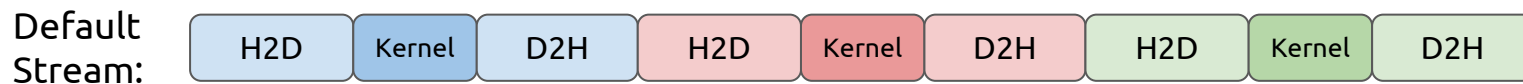
With Explicit Streams:



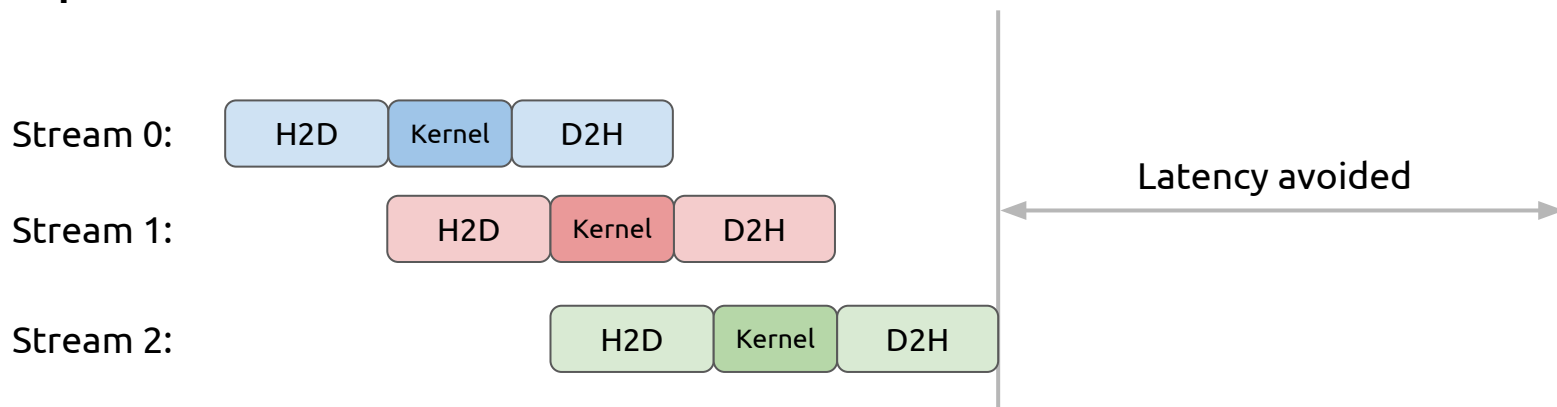
batchVectorAdd Device Execution Timeline

Scenario D: Data transfer time exceeds kernel execution time

Without Explicit Streams:



With Explicit Streams:



What If There Is Only One Massive Vector?

For large, data-parallel kernels, streaming can be used to hide data transfer latency

Simply divide the input and output into chunks and do batch processing

Increased Concurrency with Streams

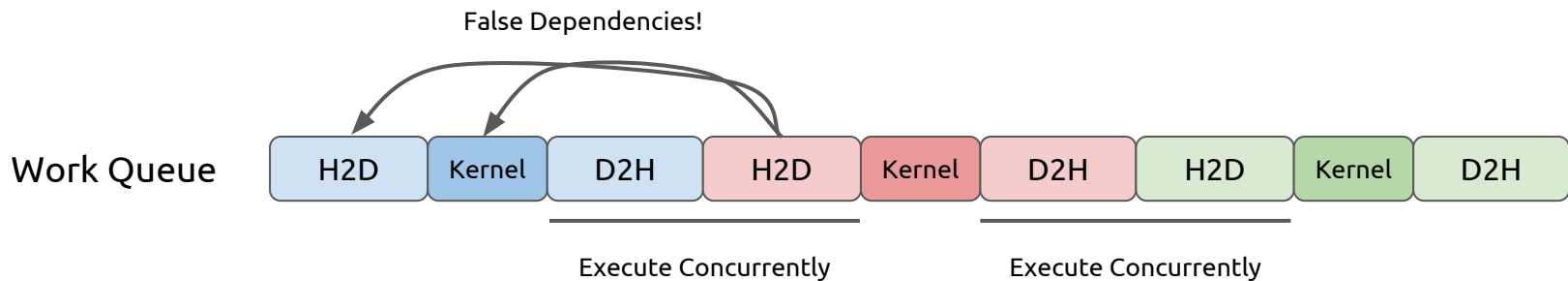
```
void massiveVectorAdd(float * A, float * B, float * C, int N) {  
  
    const int count = (N - 1) / CHUNK_SIZE + 1;  
  
    cudaStream_t * streams = new cudaStream_t[count];  
    float * d_A, * d_B, * d_C;  
  
    // make sure host pointers are in pinned memory, allocate device memory  
    ...  
  
    for (int i = 0; i < count; ++i) {  
  
        const int start = i * CHUNK_SIZE;  
        const int end = min(N, (i+1) * CHUNK_SIZE);  
  
        // set up grid and block  
        ...  
  
        cudaStreamCreate(&streams[i]);  
        cudaMemcpyAsync(d_A + start, A + start, (end-start) * sizeof(float), cudaMemcpyHostToDevice, streams[i]);  
        cudaMemcpyAsync(d_B + start, B + start, (end-start) * sizeof(float), cudaMemcpyHostToDevice, streams[i]);  
        vectorAddKernel<<<grid,block,0,streams[i]>>>(d_A + start, d_B + start, d_C + start, end-start);  
        cudaMemcpyAsync(C + start, d_C + start, (end-start) * sizeof(float), cudaMemcpyDeviceToHost, streams[i]);  
  
    }  
  
    for (int i = 0; i < count; ++i) {  
        cudaStreamSynchronize(streams[i]);  
        cudaStreamDestroy(streams[i]);  
    }  
  
    delete [] streams;  
  
}
```

Stream Scheduling in Fermi

All work was placed in a single queue

The CUDA runtime launched the next element in the queue when all its dependencies were complete

This introduced *false dependencies*



Breadth-First Ordering

```
void massiveVectorAdd(float * A, float * B, float * C, int N) {

    const int count = (N - 1) / CHUNK_SIZE + 1;

    cudaStream_t * streams = new cudaStream_t[count];
    float * d_A, * d_B, * d_C;

    // make sure host pointers are in pinned memory, allocate device memory
    ...

    for (int i = 0; i < count; ++i)    cudaStreamCreate(&streams[i]);

#define start(i) i * CHUNK_SIZE
#define size(i) min(N, (i+1) * CHUNK_SIZE) - start(i)

    for (int i = 0; i < count; ++i)
        cudaMemcpyAsync(d_A + start(i), A + start(i), size(i) * sizeof(float), cudaMemcpyHostToDevice, streams[i]);
    for (int i = 0; i < count; ++i)
        cudaMemcpyAsync(d_B + start(i), B + start(i), size(i) * sizeof(float), cudaMemcpyHostToDevice, streams[i]);
    for (int i = 0; i < count; ++i) {
        // set up grid and block
        ...
        vectorAddKernel<<<grid,block,0,streams[i]>>>(d_A + start(i), d_B + start(i), d_C + start(i), size(i));
    }
    for (int i = 0; i < count; ++i)
        cudaMemcpyAsync(C + start(i), d_C + start(i), size(i) * sizeof(float), cudaMemcpyDeviceToHost, streams[i]);

    for (int i = 0; i < count; ++i) {
        cudaStreamSynchronize(streams[i]);
        cudaStreamDestroy(streams[i]);
    }

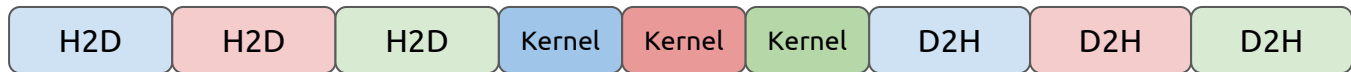
    delete [] streams;

}
```

Stream Scheduling in Fermi

Breadth-first ordering avoided false dependencies

Work Queue

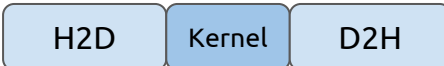


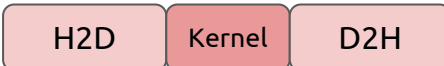
Stream Scheduling in Kepler and Beyond

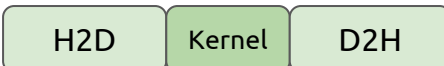
Kepler introduced “Hyper-Q”, which just means that there are 32 separate work queues

Each stream (up to 32) is mapped to a separate queue, eliminating false dependencies

The 33rd stream will have to share a work queue (may be false dependencies)

Work Queue 0 

Work Queue 1 

Work Queue 2 

Work Queue 3

...

Stream Priority

Devices with compute capability ≥ 3.5 can assign priorities to kernel launches (but not to memory transfers or other API calls)

```
cudaStreamCreateWithPriority(&stream, cudaStreamDefault, priority);
```

Grids with a lower priority value are scheduled sooner

Minimum and maximum priority values can be determined using:

```
int leastPriority, greatestPriority;
```

```
cudaDeviceGetStreamPriority(&leastPriority, &greatestPriority);
```

Blocking Streams

Non-default streams are **blocking** streams by default

When an operation is issued to the default stream:

1. The CUDA context waits for all operations already issued to blocking streams to finish
2. The operation begins
3. Other operations issued in blocking streams in the meantime will wait until the operation in the default stream has finished

Blocking Streams

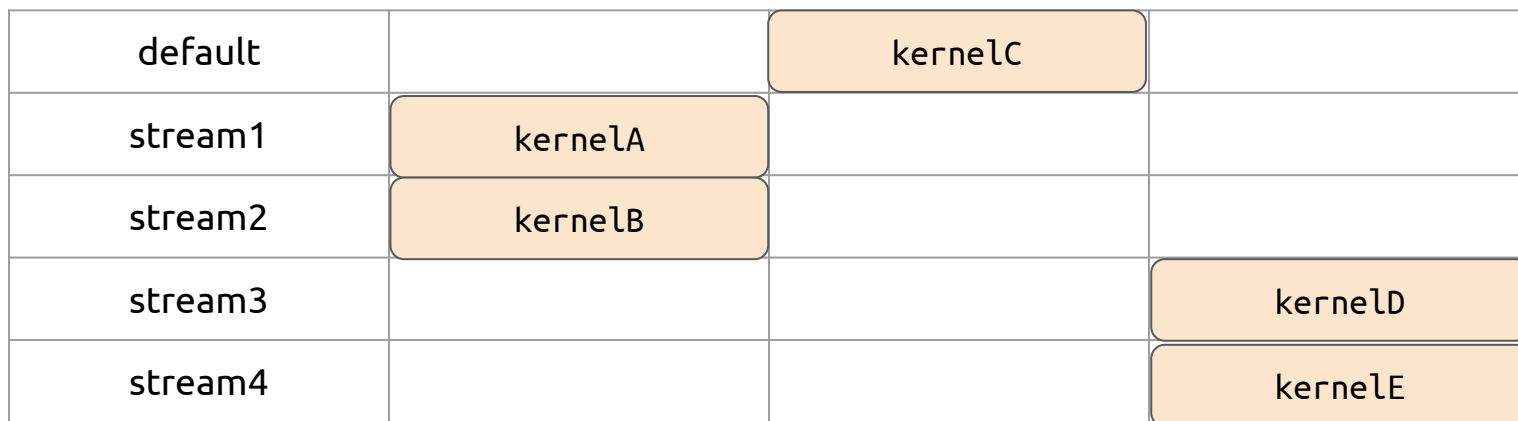
```
kernelA<<<gridA, blockA, 0, stream1>>>();
```

```
kernelB<<<gridB, blockB, 0, stream2>>>();
```

```
kernelC<<<gridC, blockC>>>();
```

```
kernelD<<<gridD, blockD, 0, stream3>>>();
```

```
kernelE<<<gridE, blockE, 0, stream4>>>();
```



time

Non-blocking Streams

Non-default streams can also be **non-blocking** if such behaviour is explicitly requested as follows:

```
cudaStream_t nonBlockingStream;
```

```
cudaStreamCreateWithFlags(&nonBlockingStream, cudaStreamNonBlocking);
```

This can be very useful when using a library that issues operations in the default stream

Events

Events are markers that can be inserted into CUDA streams and used to determine if and when a particular checkpoint is reached

Events are created and destroyed much like streams:

```
cudaEvent_t event;
```

```
cudaEventCreate(&event);
```

```
// issue operations in stream
```

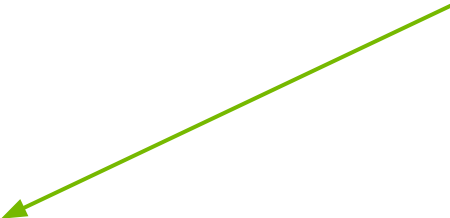
```
...
```

```
cudaEventRecord(event, stream);
```

```
...
```

```
cudaEventDestroy(event);
```

The host returns from this call immediately. However, the event is not marked as completed until the stream reaches the operation



Timing with Events

```
// create events
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

// record start
cudaEventRecord(start);

kernel<<<grid,block>>>(...);

// record stop
cudaEventRecord(stop);

// wait for stop event to be recorded
cudaEventSynchronize(stop);

// compute elapsed time
float time;
cudaEventElapsedTime(&time, start, stop);

// destroy events
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Implicit Host-Device Synchronization

Implicit synchronizations are functions with a purpose other than synchronization, but which produce synchronization as a side effect


- Page-locked host memory allocations
- Device memory allocations
- Device memset
- Memory copies between addresses on the same device

One must be careful not to accidentally serialize otherwise parallelizable tasks

Increased Concurrency with Streams

```
void batchVectorAdd(float * * As, float * * Bs, float * * Cs, int * Ns, int count) {  
  
    cudaStream_t * streams = new cudaStream_t[count];  
    float * * d_As = new float *[count];  
    float * * d_Bs = new float *[count];  
    float * * d-Cs = new float *[count];  
  
    // make sure host pointers are in pinned memory  
    ...  
  
    for (int i = 0; i < count; ++i) {  
  
        cudaMalloc(&d_As[i], Ns[i] * sizeof(float));  
        cudaMalloc(&d_Bs[i], Ns[i] * sizeof(float));  
        cudaMalloc(&d-Cs[i], Ns[i] * sizeof(float));  
  
        // set up grid and block  
        ...  
  
        cudaStreamCreate(&streams[i]);  
        cudaMemcpyAsync(d_As[i], As[i], Ns[i] * sizeof(float), cudaMemcpyHostToDevice, streams[i]);  
        cudaMemcpyAsync(d_Bs[i], Bs[i], Ns[i] * sizeof(float), cudaMemcpyHostToDevice, streams[i]);  
        vectorAddKernel<<<grid,block,0,streams[i]>>>(d_As[i], d_Bs[i], d-Cs[i], Ns[i]);  
        cudaMemcpyAsync(Cs[i], d-Cs[i], Ns[i] * sizeof(float), cudaMemcpyDeviceToHost, streams[i]);  
  
    }  
  
    for (int i = 0; i < count; ++i) {  
        cudaStreamSynchronize(streams[i]);  
        cudaStreamDestroy(streams[i]);  
    }  
  
    delete [] streams; delete [] d_As; delete [] d_Bs; delete [] d-Cs;  
  
}
```

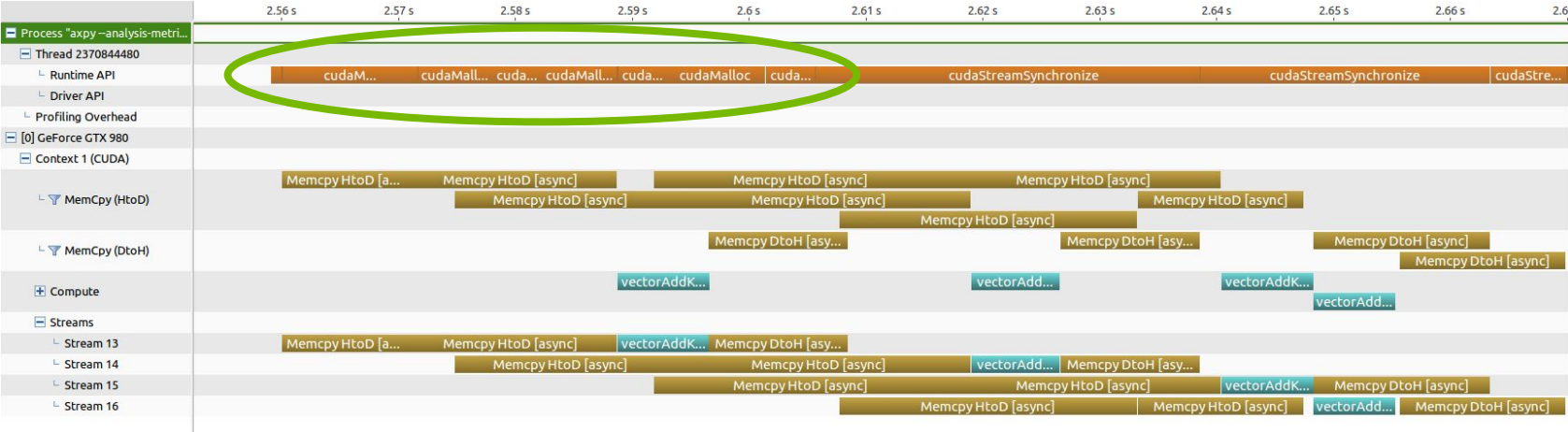
Oops! These belong in a separate loop



NVVP Revisited

cudaMalloc in the loop:

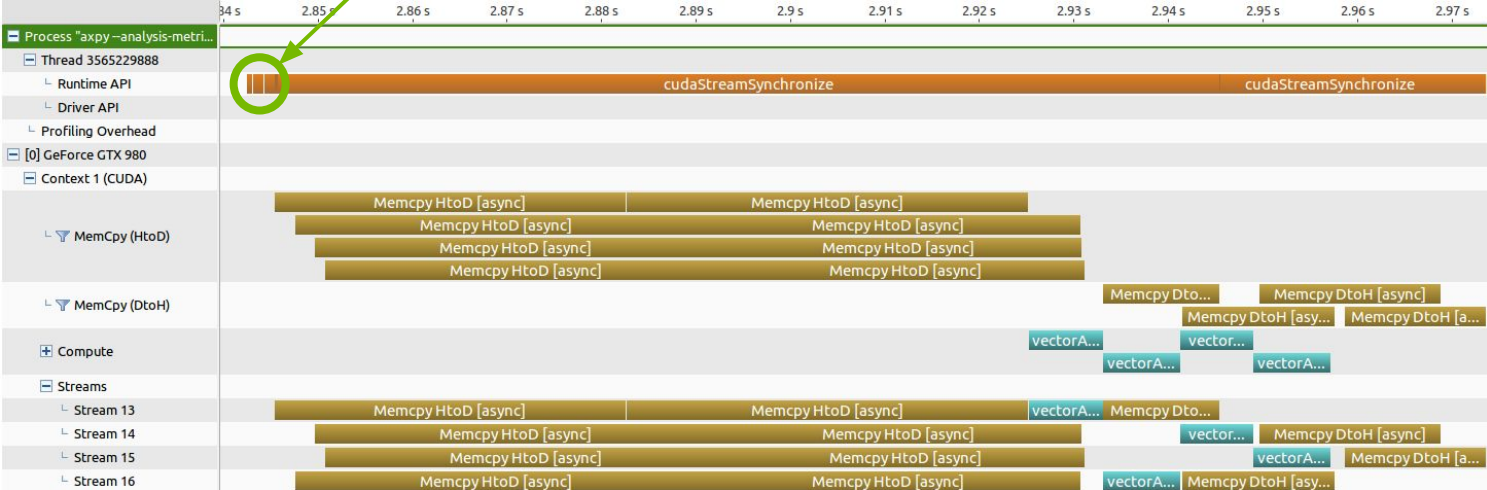
Why is the host spending so much time in cudaMalloc??



NVVP Revisited

cudaMalloc in its own loop:

This is all of the cudaMalloc calls!



Explicit Host-Device Synchronization

Block a host thread until all device operations have completed:

```
cudaDeviceSynchronize();
```

Block a host thread until all operations in a stream have completed:

```
cudaStreamSynchronize(stream);
```

Check if there are pending operations in a stream:

```
bool operationsPending = cudaStreamQuery(stream) == cudaErrorNotReady;
```

Block a host thread until an event is recorded:

```
cudaEventSynchronize(event);
```

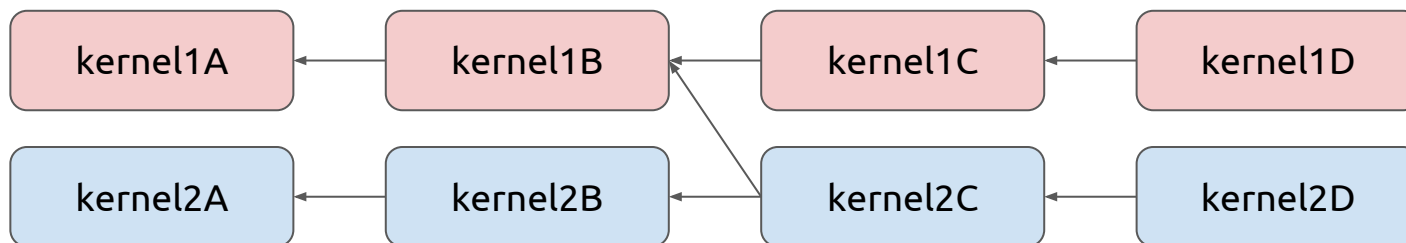
Check if an event has been recorded:

```
bool eventRecorded = cudaEventQuery(event) != cudaErrorNotReady;
```

Block a stream until an event is recorded:

```
cudaStreamWaitEvent(stream, event);
```

An Example Stream Dependency Graph



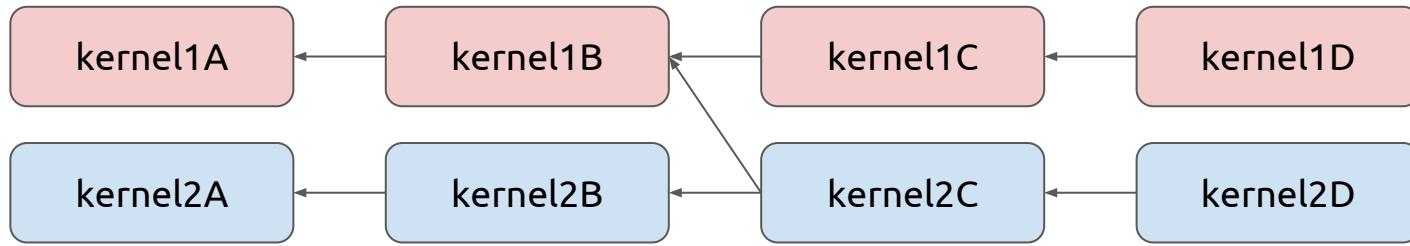
Without events:

```
kernel1A<<<..., stream1>>>(...);  
kernel2A<<<..., stream2>>>(...);  
kernel1B<<<..., stream1>>>(...);  
kernel2B<<<..., stream2>>>(...);
```

```
// host must wait here to make sure kernel1B is done  
cudaStreamSynchronize(stream1);
```

```
kernel1C<<<..., stream1>>>(...);  
kernel1D<<<..., stream1>>>(...);  
kernel2C<<<..., stream2>>>(...);  
kernel2D<<<..., stream2>>>(...);
```

An Example Stream Dependency Graph



```
cudaEvent_t kernel1BFinished;  
cudaEventCreate(&kernel1BFinished);  
  
kernel1A<<<..., stream1>>>(...);  
kernel2A<<<..., stream2>>>(...);  
kernel1B<<<..., stream1>>>(...);  
kernel2B<<<..., stream2>>>(...);  
cudaEventRecord(kernel1BFinished, stream1);  
cudaStreamWaitEvent(stream2, kernel1BFinished);  
kernel1C<<<..., stream1>>>(...);  
kernel2C<<<..., stream2>>>(...);  
kernel1D<<<..., stream1>>>(...);  
kernel2D<<<..., stream2>>>(...);
```

stream1	stream2
kernel1A	kernel2A
kernel1B	kernel2B
cudaEventRecord	cudaStreamWaitEvent
kernel1C	kernel2C
kernel1D	kernel2D

```
// the host is free to do other work after issuing all operations
```

```
...
```


Stream Callbacks

A stream callback is a host function call which can be queued in a CUDA stream

Restrictions:

1. You cannot make CUDA API calls from a callback function
2. You cannot perform any synchronization in a callback function

```
void CUDART_CB printStreamNumber(cudaStream_t stream, cudaError_t status, void * data) {  
    printf("stream %d complete\n", *((int *)data));  
}
```

```
int streamIds[nStreams];  
for (int i = 0; i < nStreams; ++i) {  
    streamIds[i] = i;  
    kernelA<<<..., streams[i]>>>(...);  
    kernelB<<<..., streams[i]>>>(...);  
    cudaStreamAddCallback(streams[i], printStreamNumber, (void *)&streamIds[i], 0);  
}
```

Conclusion / Takeaways

- Kernel launches and CUDA runtime API calls can be processed concurrently using explicitly managed streams
- Operations issued to the default stream are blocking with respect to all blocking streams
- Events allow the host to determine if and when a stream reaches a checkpoint, and can be used to synchronize multiple streams
- A number of CUDA runtime API calls implicitly synchronize device and host and should be placed where such synchronization will not block otherwise concurrent execution
- nVidia Visual Profiler is a very valuable development tool for applications with many kernels and host / device memory transfers

Sources

<https://www.wikipedia.org/>

Cheng, John, Max Grossman, and Ty McKercher. Professional Cuda C Programming. John Wiley & Sons, 2014.

Hwu, Wen-mei, and David Kirk. "Programming massively parallel processors." Special Edition 92 (2009).

Wilt, Nicholas. The cuda handbook: A comprehensive guide to gpu programming. Pearson Education, 2013.