CS 6023 - GPU Programming

Streams

26/10/2018

Setting and Agenda

- Understand GPU streams’ syntax
- How to apply streams to improve performance of data-parallel applications
- How to apply streams to extend types of parallelisation possible with GPU

Different kinds of streams

Streams - Why

- We have been looking at data-parallel execution on GPUs
  - Geometric decomposition
  - Mostly embarrassingly parallel
- What about more complex kinds of parallelism
  - What are the kinds we have seen earlier?

Streams - Why

- Streams can enable **task parallelism** on GPU
- What is task parallelism?
  - Instead of applying the same function (or kernel) on different pieces of data, we want to run two or more different functions (or tasks)
- The different tasks can be on the same or different data items
  - Examples?

Streams - Why

- Streams can enable **task parallelism** on GPU
- What is task parallelism?
  - Instead of applying the same function (or kernel) on different pieces of data, we want to run two or more different functions (or tasks)
- The different tasks can be on the same or different data items
  - Examples?

- Task parallelism is not the most *efficient* usage of GPU hardware, but for some applications will be required for *full* usage of GPU hardware

Streams - What

- For task parallelism, we need to have multiple kernels executing on the GPU concurrently
- The current programming model we have studied does not support this

- Define the notion of a stream and the corresponding **programming model**
  - A stream is a sequence of CUDA commands which run sequentially
- Multiple CUDA streams can be defined and executed on a GPU concurrently
- No ‘program order’ maintained between streams

So far

```
__global__ void kernel_a() {
    print("a");
}
__global__ void kernel_b() {
    print("b");
}
int main() {
    kernel_a<<<1, 16>>>();
    print(“ab”);
    kernel_b<<<1, 16>>>();
...
}
```

What is the expected output?

Streams example

```
__global__ void kernel_a() {
    print("a");
}
__global__ void kernel_b() {
    print("b");
}
int main() {
    cudaStream_t s1, s2;
    cudaStreamCreate(&s1);
    cudaStreamCreate(&s2);
    kernel_a<<<1, 16, size, s1>>>();
    print("ab");
    kernel_b<<<1, 16, size, s2>>>();
…
    cudaStreamDestroy(&s1);
    cudaStreamDestroy(&s2);
}
```

What is the expected output?

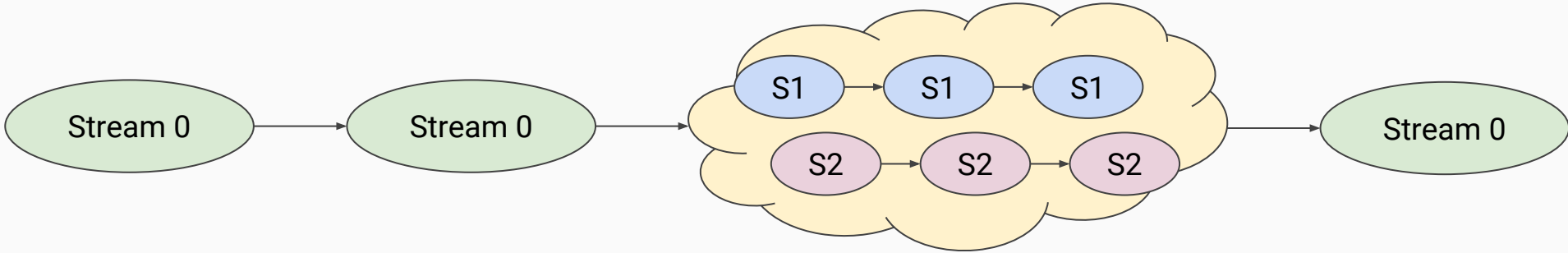
The two streams run concurrently,
so a's and b's can be non-deterministically interleaved

The default stream

- If we don’t specify the fourth parameter, the kernel is attached to the **default stream or the null stream or stream 0**
- There is no additional need to create or delete this null stream
- Default stream is special: It is implicitly synchronized wrt host and device

The default stream

- If we don’t specify the fourth parameter, the kernel is attached to the **default stream or the null stream or stream 0**
- There is no additional need to create or delete this null stream
- Default stream is special: It is implicitly synchronized wrt host and device



- If we want a specific stream to be non-blocked by the default stream, then we can create it with a specific flag `cudaStreamNonBlocking`

Asynchronous calls

- To support the semantics of concurrent streams, we need to define asynchronous calls to the GPU
  - Ie, the control at the host does not wait for the GPU to finish

Asynchronous calls

- To support the semantics of concurrent streams, we need to define asynchronous calls to the GPU
  - Ie, the control at the host does not wait for the GPU to finish
- Already the following are asynchronous:
  - Kernel calls
  - `cudaMemcpy()` within the device - D2D
- But to support streams meaningfully, we need an asynchronous version of some other call. Which one?

AsyncMemcpy()

```
cudaStream_t stream1, stream2;
cudaStreamCreate ( &stream1) ;cudaStreamCreate ( &stream2) ;
...
cudaMalloc ( &dev1, size ) ;
cudaHostAlloc ( &host1, size ) ; // pinned memory required on host
cudaMalloc ( &dev2, size ) ;
cudaHostAlloc ( &host2, size ) ; // pinned memory required on host
...
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;
kernel1 <<< grid, block, 0, stream1 >>> ( …, dev1, … ) ;
cudaMemcpyAsync ( dev2, host2, size, H2D, stream2 ) ;
kernel1 <<< grid, block, 0, stream1 >>> ( …, dev2, … ) ;
some_CPU_method ();
...
cudaFreeHost(&host1); cudaFreeHost(&host2);
```

Page locked host memory

- So far we have only been allocating memory on the device with `cudaMalloc`
- For allocating memory on host, we use the standard C `malloc`

- For async memory copies, we need to allocate memory on the host too
- Further, this memory must not be *paged out* so as to enable async load
    - Standard malloc creates pageable host memory

Page locked host memory

- So far we have only been allocating memory on the device with `cudaMalloc`
- For allocating memory on host, we use the standard C `malloc`

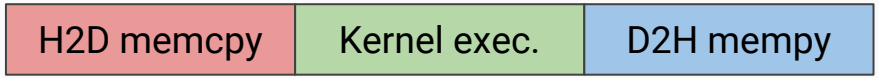
- For async memory copies, we need to allocate memory on the host too
- Further, this memory must not be *paged out* so as to enable async load
  - Standard malloc creates pageable host memory
- `cudaHostAlloc()` allocates **page-locked** or **pinned** memory
  - They are guaranteed to remain in the physical memory by the OS
  - Further, can use real addresses rather than virtual address
  - But pinning memory can starve CPU

Stream - use cases

- Would you want to use streams when you have a single kernel?

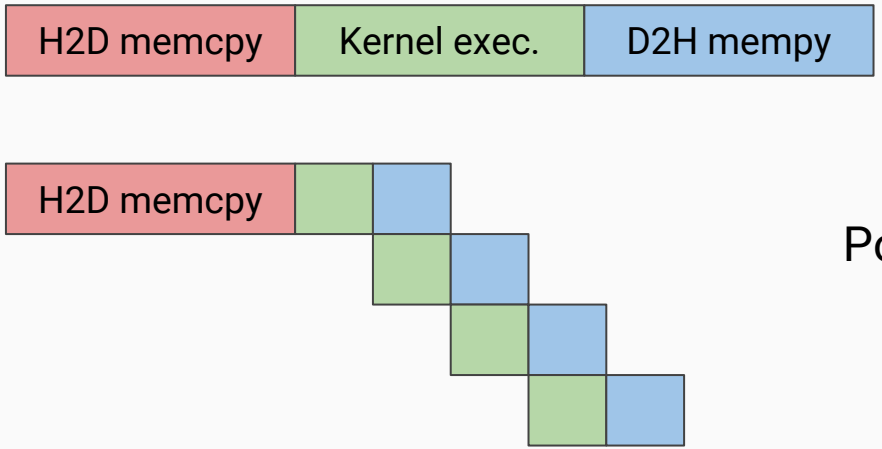
Stream - use cases

- Would you want to use streams when you have a single kernel?
- Yes to take advantage of async memory copies



Stream - use cases

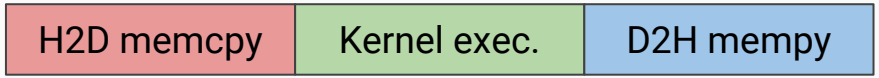
- Would you want to use streams when you have a single kernel?
- Yes to take advantage of async memory copies

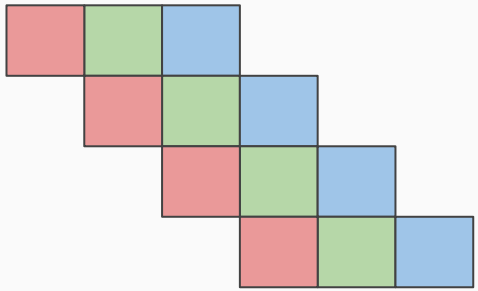


Potentially, 2x speed-up

Stream - use cases

- Would you want to use streams when you have a single kernel?
- Yes to take advantage of async memory copies



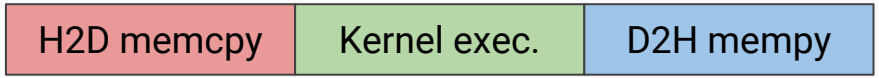


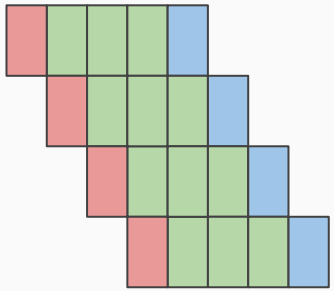
Potentially, 3x speed-up

Note we are to now use async memcpy

Stream - use cases

- Would you want to use streams when you have a single kernel?
- Yes to take advantage of async memory copies





Can achieve even higher speedup with pipelining

When is this the right approach?

Impact of streams on data parallel applications

- For standard applications (GEMM, Blas), streams can almost hide all latency of accessing host memory over PCI-Express
- Also helps nullify the effect of small device memories

- Application in deep learning where learning happens on batches of inputs
  - Batch size to be decided both from learning and GPU perspectives

Interleaving memcpy and kernel

```
cudaStream_t stream;
cudaStreamCreate (&stream) ;
...
cudaMalloc (&devA, ... ); cudaMalloc (&devB, ... );
cudaMalloc (&devC, ... );
cudaHostAlloc (&a, ...); cudaHostAlloc (&b, ...);
cudaHostAlloc (&c, ...);
...
for(int i=0;i<SIZE;i+=N) {
    cudaMemcpyAsync(dev_a,a+i,N*sizeof(int),H2D,stream);
    cudaMemcpyAsync(dev_b,b+i,N*sizeof(int),H2D,stream);
    vecAddKernel<<<N, 1, 0, stream>>>(dev_a,dev_b,dev_c);
    cudaMemcpyAsync(c+1,dev_c,N*sizeof(int),D2H,stream);
}
...
cudaStreamSynchronize(stream);
cudaStreamDestroy(stream);
```

Using multiple streams

```
cudaStream_t stream1, stream2;
cudaStreamCreate (&stream); cudaStreamCreate (&stream2);
...
cudaMalloc (&devA1, ... ); cudaMalloc (&devB1, ... ); cudaMalloc (&devC1, ... );
cudaMalloc (&devA2, ... ); cudaMalloc (&devB2, ... ); cudaMalloc (&devC2, ... );
cudaHostAlloc (&a, ...); cudaHostAlloc (&b, ...); cudaHostAlloc (&c, ...);
...
for(int i=0;i<SIZE;i+=2*N) {
    cudaMemcpyAsync(dev_a1,a+i,N*sizeof(int),H2D,stream1);
    cudaMemcpyAsync(dev_b1,b+i,N*sizeof(int),H2D,stream1);
    vecAddKernel<<<N, 1, 0, stream1>>>(dev_a1,dev_b1,dev_c1);
    cudaMemcpyAsync(c+1,dev_c1,N*sizeof(int),D2H,stream1);
    cudaMemcpyAsync(dev_a2,a+N+i,N*sizeof(int),H2D,stream2);
    cudaMemcpyAsync(dev_b2,b+N+i,N*sizeof(int),H2D,stream2);
    vecAddKernel<<<N, 1, 0, stream2>>>(dev_a2,dev_b2,dev_c2);
    cudaMemcpyAsync(c+N+i,dev_c2,N*sizeof(int),D2H,stream2);
}
...
cudaStreamSynchronize(stream1); cudaStreamSynchronize(stream2);
```

What is the interleaving pattern?

```
cudaMemcpyAsync(dev_a1,a+i,N*sizeof(int),H2D,stream1);
cudaMemcpyAsync(dev_b1,b+i,N*sizeof(int),H2D,stream1);
vecAddKernel<<<N, 1, stream1>>>(dev_a1,dev_b1,dev_c1);
cudaMemcpyAsync(c+1,dev_c1,N*sizeof(int),D2H,stream1);
cudaMemcpyAsync(dev_a2,a+N+i,N*sizeof(int),H2D,stream2);
cudaMemcpyAsync(dev_b2,b+N+i,N*sizeof(int),H2D,stream2);
vecAddKernel<<<N, 1, stream2>>>(dev_a2,dev_b2,dev_c2);
cudaMemcpyAsync(c+N+i,dev_c2,N*sizeof(int),D2H,stream2);
```

What is the interleaving pattern?

```
cudaMemcpyAsync(dev_a1,a+i,N*sizeof(int),H2D,stream1);
cudaMemcpyAsync(dev_b1,b+i,N*sizeof(int),H2D,stream1);
vecAddKernel<<<N, 1, stream1>>>(dev_a1,dev_b1,dev_c1);
cudaMemcpyAsync(c+1,dev_c1,N*sizeof(int),D2H,stream1);
cudaMemcpyAsync(dev_a2,a+N+i,N*sizeof(int),H2D,stream2);
cudaMemcpyAsync(dev_b2,b+N+i,N*sizeof(int),H2D,stream2);
vecAddKernel<<<N, 1, stream2>>>(dev_a2,dev_b2,dev_c2);
cudaMemcpyAsync(c+N+i,dev_c2,N*sizeof(int),D2H,stream2);
```

Here we are assuming that each of the operations (3 memcopies and 1 kernel) take the same amount of time

Stream 1	Stream 2
A H2D	
B H2D	
kernel	A H2D
	B H2D
C D2H	kernel
	C D2H
A H2D	
B H2D	
kernel	A H2D
	B H2D

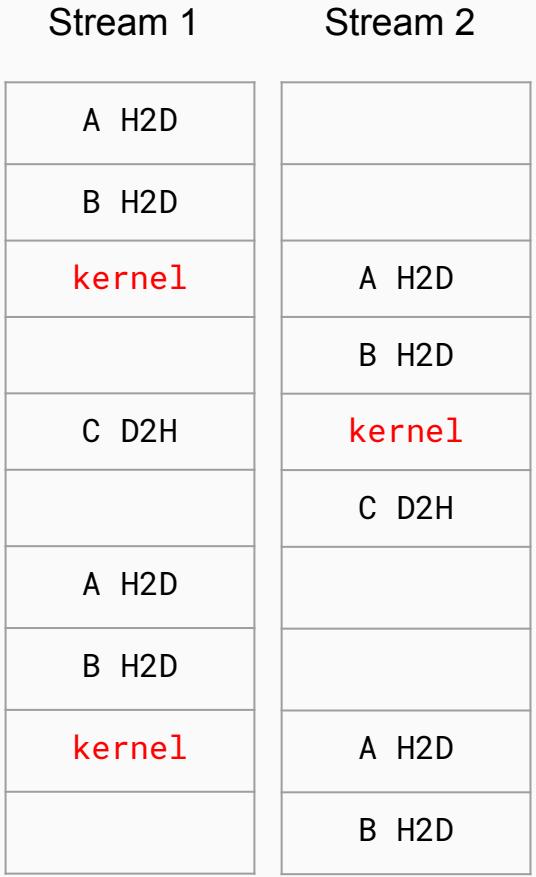
A different interleaving pattern

```
cudaMemcpyAsync(dev_a1,a+i,N*sizeof(int),H2D,stream1);
cudaMemcpyAsync(dev_b1,b+i,N*sizeof(int),H2D,stream1);
cudaMemcpyAsync(dev_a2,a+N+i,N*sizeof(int),H2D,stream2);
cudaMemcpyAsync(dev_b2,b+N+i,N*sizeof(int),H2D,stream2);
vecAddKernel<<<N, 1, stream1>>>(dev_a1,dev_b1,dev_c1);
vecAddKernel<<<N, 1, stream2>>>(dev_a2,dev_b2,dev_c2);
cudaMemcpyAsync(c+1,dev_c1,N*sizeof(int),D2H,stream1);
cudaMemcpyAsync(c+N+i,dev_c2,N*sizeof(int),D2H,stream2);
```

A different interleaving pattern

```
cudaMemcpyAsync(dev_a1,a+i,N*sizeof(int),H2D,stream1);
cudaMemcpyAsync(dev_b1,b+i,N*sizeof(int),H2D,stream1);
cudaMemcpyAsync(dev_a2,a+N+i,N*sizeof(int),H2D,stream2);
cudaMemcpyAsync(dev_b2,b+N+i,N*sizeof(int),H2D,stream2);
vecAddKernel<<<N, 1, stream1>>>(dev_a1,dev_b1,dev_c1);
vecAddKernel<<<N, 1, stream2>>>(dev_a2,dev_b2,dev_c2);
cudaMemcpyAsync(c+1,dev_c1,N*sizeof(int),D2H,stream1);
cudaMemcpyAsync(c+N+i,dev_c2,N*sizeof(int),D2H,stream2);
```

Here we are assuming that each of the operations (3 memcopies and 1 kernel) take the same amount of time



Synchronization

- `cudaDeviceSynchronize()`
  - Synchronize device and host across all streams
  - Block host (CPU) until all device commands are completed

Synchronization

- `cudaDeviceSynchronize()`
  - Synchronize device and host across all streams
  - Block host (CPU) until all device commands are completed

- `cudaStreamSynchronize( streamId )`
  - Synchronize host and device on a particular steam
  - Block host until all commands to a particular stream are completed
  - But this creates a blocking wait, we need something else. What?

Status

- `cudaStreamQuery(streamId)`
  - Returns `cudaSuccess` if all operations in stream have completed, or `cudaErrorNotReady` if not
  - What will you use this for?

Once again - Cuda Events!

- Cuda events signal when events have occured in streams
- They are used to
  - Profile applications
  - Synchronize between streams

- Events have a boolean status: Occurred or Not occurred

The Events API

- `cudaEventCreate(&event)`
  - Create an event with the default state as occurred
- `cudaEventDestroy(event)`
  - Destroy event
- `cudaEventRecord(event, stream)`
  - The event's status is set to not-occured
  - Enqueue the event into the stream and report when set to occured
- `cudaEventElapsedTime( &time, start, end )`
  - Time between two events

Using events (beyond timing)

- `cudaEventQuery ( event )`
  - Returns `CUDA_SUCCESS` if an event has occurred
- `cudaEventSynchronize ( event )`
  - Blocks host until stream completes all outstanding calls
- `cudaStreamWaitEvent ( stream, event )`
  - Blocks stream until event occurs
  - Only blocks launches after this call
  - Does not block the host!

Using events (beyond timing)

- Provides mechanisms for fine-grained synchronization
  - Within a stream
  - And across streams

- We know how to synchronize
  - CPU and GPU (all streams) -> `cudaDeviceSynchornize()`
  - CPU and one stream -> `cudaStreamSynchornize()`

Using events (beyond timing)

- Provides mechanisms for fine-grained synchronization
  - Within a stream
  - And across streams

- We know how to synchronize
  - CPU and GPU (all streams) -> `cudaDeviceSynchornize()`
  - CPU and one stream -> `cudaStreamSynchornize()`
- How do we synchronize two streams, i.e. a barrier sync for two streams?

Using events (beyond timing)

- Provides mechanisms for fine-grained synchronization
  - Within a stream
  - And across streams

- We know how to synchronize
  - CPU and GPU (all streams) -> `cudaDeviceSynchornize()`
  - CPU and one stream -> `cudaStreamSynchornize()`
- How do we synchronize two streams, i.e. a barrier sync for two streams?
- N streams?

Stream callback functions

- With concurrent behavior of multiple streams, it becomes challenging for the CPU to react to the behavior of streams
- For instance, CPU may be interested to do something after a (out of many) stream finishes

More functions

- With concurrent behavior of multiple streams, it becomes challenging for the CPU to react to the behavior of streams
- For instance, CPU may be interested to do something after a (out of many) stream finishes
- Usual programming pattern in software engineering: **callbacks**
- We can add callback function to a stream which gets executed after all previous commands on the stream have completed

Callback example

```
__global__ void myKernel() {
    ...
}
void myCallBack(cudaStream_t s, cudaError_t st, void *d) {
    printf("The stream id is %d\n", (int)d);
}
int main() {
    cudaStream_t s[8];
    for (int i = 0; i < 8; i++) {
        cudaStreamCreate(&s[i]);
        myKernel<<<1, 16, 0, s[i]>>>();
        cudaStreamAddCallback(s[i], myCallBack, (void *)i);
        ...
    }
}
```

Stream callback functions

- Callback function runs on the host
- It is blocking for the stream
  - Until the callback function is completed, the stream does not proceed
- Bummer: No CUDA api calls in callback function

- For synchronizing between streams - use events
- For synchronizing within a stream and run CPU post-processing - use callback

Stream priorities

- By default all streams have the same priority
- Can create streams with different priorities
  - `cudaStreamCreateWithPriority(&streamId)`
- Can create a custom range of priority levels with
  - `cudaDeviceGetStreamPriorityRange()`

Stream priorities

- By default all streams have the same priority
- Can create streams with different priorities
  - `cudaStreamCreateWithPriority(&streamId)`
- Can create a custom range of priority levels with
  - `cudaDeviceGetStreamPriorityRange()`

- Semantics: If there are higher priority kernels running, then no lower priority kernels are executed
- Minefield of deadlock bugs!

Sample end-sem question

- We have a program with s+1 CUDA streams
- One of the streams pre-processes and post-processes the data
- All other streams compute on the data between these two steps
- How would you implement this with events?

Sample end-sem question

- We have a series of words which have to be encrypted
- Each word is assigned to a different stream
- But the results are to be printed by the CPU in the same order as in the original text
- How would you do this?

- What if the input set of words is very large? For streams that are freed up, we want to continue with the next word.