GPU Programming

Rupesh Nasre.
http://www.cse.iitm.ac.in/~rupesh

IIT Madras
July 2017

Asynchronous Processing

- Independent tasks can run simultaneously.
- C semantics are sequential.
  - Assume a barrier after every instruction.
- What can be concurrent?

1. Computation on the host
2. Computation on the device
3. CPU $\rightarrow$ GPU memory transfer
4. CPU $\leftarrow$ GPU memory transfer
5. GPU $\rightarrow$ GPU memory transfer on a device
6. GPU $\rightarrow$ GPU memory transfer across devices

1 Computation on the host

- Following device operations are asynchronous with the host:
  - Kernel launches
  - Memory copies within a device
  - $\leq$ 64 KB memory copy from CPU $\rightarrow$ GPU
  - Memory copies by *Async* functions
  - *memset* function calls
- For debugging:
  - Disable asynchronicity by setting environment variable CUDA_LAUNCH_BLOCKING = 1.

2 Computation on the device

- Kernels can execute concurrently.
  - Compute capability ≥ 2.x
  - Maximum number of resident grids: 32 … 128
  - Kernels using large amount of local memory is less likely to execute concurrently with other kernels.
  - By default, kernels execute sequentially.

```
K1<<<...>>>();
K2<<<...>>>();
K3<<<...>>>();
cudaDeviceSynchronize();
```

Need streams to make them concurrent.

Memory transfer

- Concurrent data transfer
  - Overlap copies to and from the device
  - Support kernel execution concurrently with data transfer
  - Compute capability ≥ 2.x

Concurrent copy and kernel execution: Yes with 2 copy engine(s)

```
__global__ void K1() {
	unsigned num = 0;
	for (unsigned ii = 0; ii < threadIdx.x; ++ii)
		num += ii;
	printf("K1: %d\n", threadIdx.x);
}
__global__ void K2() {
	printf("K2\n");
}
int main() {

	K1<<<1, 1024>>>();
	K2<<<1, 32>>>();
	cudaDeviceSynchronize();

	return 0;
}
```

Output

K1
K1
...
K2
K2

To specify stream, we need the **fourth parameter** of the kernel launch.

```
__global__ void K1() {
        unsigned num = 0;
        for (unsigned ii = 0; ii < threadIdx.x; ++ii)
                num += ii;
        printf("K1: %d\n", threadIdx.x);
}
__global__ void K2() {
        printf("K2\n");
}
int main() {
        cudaStream_t s1, s2;
        cudaStreamCreate(&s1);
        cudaStreamCreate(&s2);

        K1<<<1, 1024, 0, s1>>>();
        K2<<<1, 32, 0, s2>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

Possible Output

```
K2
K2
...
K1
K1
```

Streams

- A stream is a queue of device work
  - Host enqueues work and returns immediately.
  - Device schedules work from streams when resources are free
- CUDA operations are placed within a stream
  - kernel launches, memcpy, etc.
- Operations within a stream are ordered and cannot overlap.
- Operations across streams are unordered and can overlap.

Classwork

- Write a code to overlap two data transfers.
- Write a code to overlap data transfer with kernel execution.

```
for (unsigned ii = 0; ii < N; ++ii) {
    cudaMemcpyAsync(dinptr + ii * nbytesperstream, hptr + ii * nbytesperstream,
                    nbytesperstream, cudaMemcpyHostToDevice, stream[ii]);
    K<<<nbytesperstream / 512, 512, 0, stream[ii]>>>(
                    doutptr + ii * nbytesperstream,
                    dinptr + ii * nbytesperstream,
                    nbytesperstream);
    cudaMemcpyAsync(hptr + ii * nbytesperstream, doutptr + ii * nbytesperstream,
                    nbytesperstream, cudaMemcpyDeviceToHost, stream[ii]);
}
```

Homework

- Given a large array which does not fit into GPU memory, you want to bring in partitions of the array into memory, process on GPU and store results back (either in CPU memory or disk).
- Each partition has H2D, K and D2H processing.
- This can be implemented in a pipelined fashion, with H2D of i^{th} partition, K of $i+1^{th}$ partition and D2H of $i+2^{th}$ partition happening concurrently.
- Write CUDA code for the same.
- Discuss issues with this approach.

Across-Stream Synchronization

- Implement a global barrier across two kernels running in different streams.
  - Can there be a deadlock? Compare with resident blocks.
- Can there be a deadlock between two stream operations where:
  - one is a kernel and another is a memory transfer?
  - two memory transfers are involved?

Classwork

- You want to increment each element of an N-sized array.
- You launch a kernel with M threads.
- If only one kernel is launched, M threads increment all N elements.
- If another kernel is launched (in another stream), then 2*M threads divide the work equally.
- Generalize the above for arbitrary number of streams.

Classwork

- Devise a scheme such that a kernel dynamically shrinks or expands its work based on the availability or non-availability of other kernels.
  - Initially, each thread plans to work on X number of items.
  - If another kernel gets launched, each thread decides to work on X/2 items.
  - If yet another kernel gets launched, each thread decides to work on X/3 items, and so on.
  - If a kernel terminates, each thread decides to work on X/2 items again.

Find all possible outputs.

```
__global__ void K1() {
        printf("K1\n”);
}
__global__ void K2() {
        printf("K2\n");
}
__global__ void K3() {
        printf("K3\n");
}
int main() {
        cudaStream_t s1, s2, s3;
        cudaStreamCreate(&s1);
        cudaStreamCreate(&s2);
        cudaStreamCreate(&s3);

        K1<<<1, 32, 0, s1>>>();
        K2<<<1, 32, 0, s2>>>();
        K3<<<1, 32, 0, s3>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

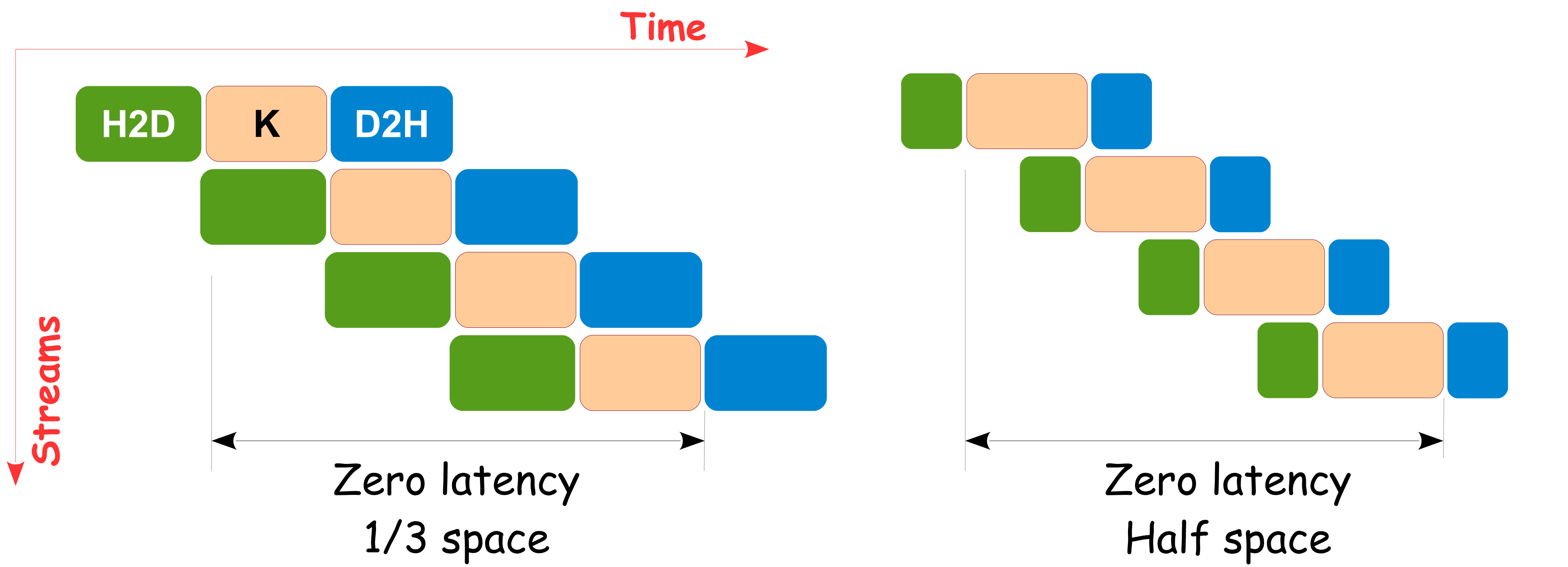
Find all possible outputs.

```
__global__ void K1() {
        printf("K1\n”);
}
__global__ void K2() {
        printf("K2\n");
}
__global__ void K3() {
        printf("K3\n");
}
int main() {
        int *ptr;
        cudaStream_t s1, s2, s3;
        cudaStreamCreate(&s1);
        cudaStreamCreate(&s2);
        cudaStreamCreate(&s3);

        K1<<<1, 32, 0, s1>>>();
        cudaHostAlloc(&ptr, 1, 0);
        K2<<<1, 32, 0, s2>>>();
        K3<<<1, 32, 0, s3>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

Stream Synchronization

- *CudaDeviceSynchronize()* waits for all commands from all streams.
- *CudaStreamSynchronize(s)* waits for all commands from a particular stream *s*.
  - For non-blocking check, use *cudaStreamQuery(s)*.
- Commands from different streams cannot run concurrently if they are separated by:
  - Page-locked host memory allocation
  - Device memory allocation
  - Device memset
  - Memcpy on the same device memory
  - CUDA commands to the default stream
  - Switch between L1/shared memory configurations

Overlapped Computation and Communication




```
for all i {
      H2D(i)
      K(i)
      D2H(i)
}
```

```
for all i: H2D(i)
for all i: K(i)
for all i: D2H(i)
```

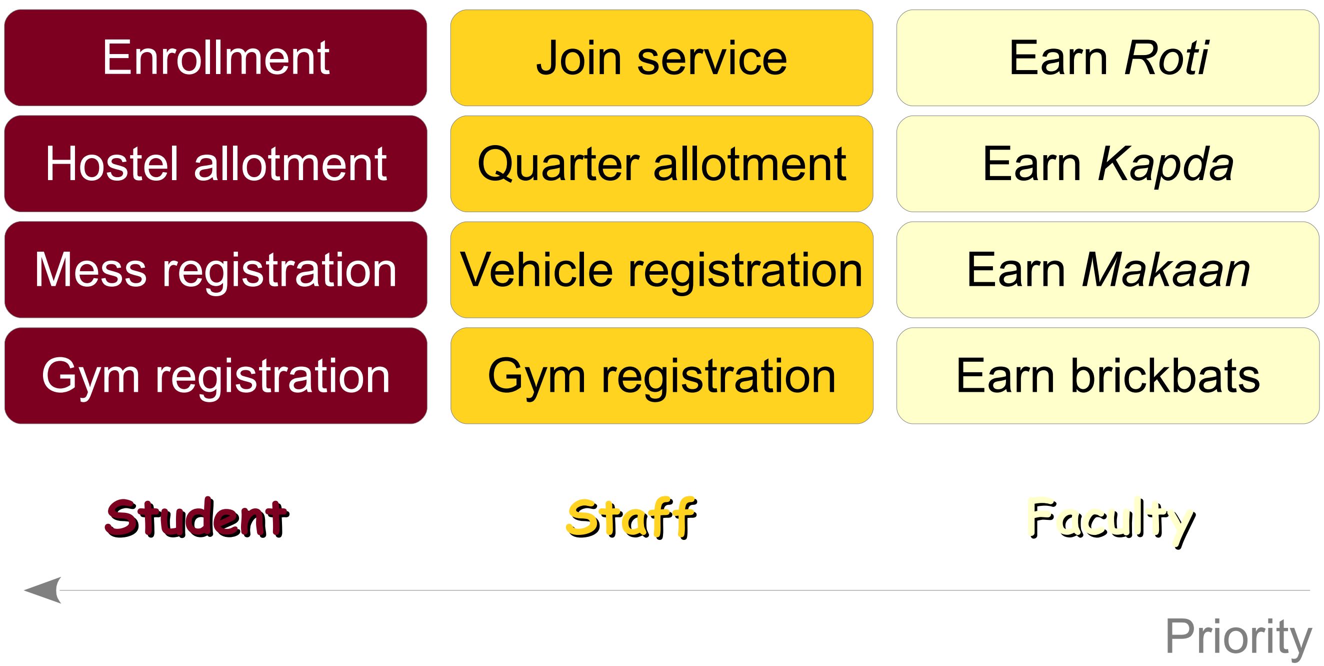
```
for all i {
      H2D(i)
      K(i)
}
for all i: D2H(i)
```

Classwork

- Sorted output 1..10: Host prints 0 mod 3, K1 prints 1 mod 3 and K2 prints 2 mod 3.
- Double it up:
  - Host launches K1, K2, K3 concurrently.
  - K1, K2, K3 print a random number of elements each.
  - K1 returns to the host and host knows how many elements were printed by K1, say N1. It informs this to already running K2.
  - K2 makes sure it prints at least 2*N1 elements.
  - K2 returns now and host knows its number of elements, say N2.
  - K3 makes sure it prints at least 2*N2 elements in total.

Streams with Priorities

- By default, all streams have the same priority.
- Streams can be created with varying priorities.




```
cudaStreamCreateWithPriority(
        &student, …, 2);
cudaStreamCreateWithPriority(
        &staff, …, 1);
cudaStreamCreateWithPriority(
        &faculty, …, 0);
```

Streams with Priorities

- Supported with CUDA >= 7.0

```
#include <stdio.h>
#include <cuda.h>

int main() {
	int plow, phigh;
	cudaDeviceGetStreamPriorityRange(&plow, &phigh);
	printf("%d -- %d\n", plow, phigh);
}
```

Classwork

- Given the following tasks, find the execution sequence with a single execution unit under
  - Round-robin scheduling (quantum = 1)
  - Prioritized scheduling (s1 < s2 < s3)

Task	Burst time	Arrival time	Stream
C1	1	0	s1
K1	4	0	s1
C2	2	2	s1
K2	3	2	s2
C3	1	4	s2

Asynchronous Execution

- is good, but
  - Increases number of concurrent behaviors; makes code understanding difficult
  - Satisfying dependencies becomes difficult
  - With streams, multiple tasks may execute *at their own pace*, making interception difficult.
- needs callbacks to satisfy dependencies
  - *Let me know!*
  - Compare with sleep and wakeup (I am talking about OS threads)

Stream Callbacks

- We can insert a callback into a stream.
- The callback function gets executed on the host after all previous commands have completed.
- Callbacks in the default stream are executed after all the preceding commands issued in all streams have completed.
- Callbacks are blocking.

```
#include <stdio.h>
#include <cuda.h>
#define N 2

void mycallback(cudaStream_t stream, cudaError_t status, void *data) {
	printf("inside callback %d\n", (long)data);
}
__global__ void K() {
	printf("in kernel K\n");
}
int main() {
	cudaStream_t stream[N];
	for (long ii = 0; ii < N; ++ii) {
		cudaStreamCreate(&stream[ii]);
		K<<<1, 1, 0, stream[ii]>>>();
		cudaStreamAddCallback(stream[ii], mycallback, (void *)ii, 0);
		cudaDeviceSynchronize();
	}
}
```

```
in kernel K
inside callback 0
in kernel K
inside callback 1
```

```
#include <stdio.h>
#include <cuda.h>
#define N 2

void mycallback(cudaStream_t stream, cudaError_t status, void *data) {
    printf(“inside callback %d\n”, (long)data);
}
__global__ void K() {
    printf(“in kernel K\n”);
}
int main() {
    cudaStream_t stream[N];
    for (long ii = 0; ii < N; ++ii) {
        cudaStreamCreate(&stream[ii]);
        K<<<1, 1, 0, stream[ii]>>>();
        cudaStreamAddCallback(stream[ii], mycallback, (void *)ii, 0);
        K<<<1, 1, 0, stream[ii]>>>();
        cudaDeviceSynchronize();
    }
}
```

in kernel K
inside callback 0
in kernel K
in kernel K
in kernel K
inside callback 1

Impossible

```
#include <stdio.h>
#include <cuda.h>
#define N 2

void mycallback(cudaStream_t stream, cudaError_t status, void *data) {
    printf("inside callback %d\n", (long)data);
}
__global__ void K() {
    printf("in kernel K\n");
}
int main() {
    cudaStream_t stream[N];
    for (long ii = 0; ii < N; ++ii) {
        cudaStreamCreate(&stream[ii]);
        K<<<1, 1, 0, stream[ii]>>>();
        cudaStreamAddCallback(stream[ii], mycallback, (void *)ii, 0);
        K<<<1, 1, 0, stream[ii]>>>();
        cudaDeviceSynchronize();
    }
}
```

```
in kernel K
inside callback 0
in kernel K
in kernel K
inside callback 1
in kernel K
```

```
#include <stdio.h>
#include <cuda.h>
#define N 2
__global__ void K() {
    printf("in kernel K\n");
}
void mycallback(cudaStream_t stream, cudaError_t status, void *data) {
    printf("inside callback %d\n", (long)data);
    K<<<1, 1, 0, stream>>>();
    cudaDeviceSynchronize();
}
int main() {
    cudaStream_t stream[N];
    for (long ii = 0; ii < N; ++ii) {
        cudaStreamCreate(&stream[ii]);
        K<<<1, 1, 0, stream[ii]>>>();
        cudaStreamAddCallback(stream[ii], mycallback, (void *)ii, 0);
        K<<<1, 1, 0, stream[ii]>>>();
        cudaDeviceSynchronize();
    }
}
```

```
in kernel K
inside callback 0
in kernel K
in kernel K
inside callback 1
in kernel K
```

```
#include <stdio.h>
#include <cuda.h>
#define N 2
__global__ void K() {
    printf("in kernel K\n");
}
void mycallback(cudaStream_t stream, cudaError_t status, void *data) {
    printf("inside callback %d\n", (long)data);
    K<<<1, 1, 0, stream>>>();
    CudaDeviceSynchronize();
    cudaError_t err = cudaGetLastError();
    printf("%d, %s, %s\n", err, cudaGetErrorName(err), cudaGetErrorString(err));
}
int main() {
    cudaStream_t stream
    for (long ii = 0; ii < N;
        cudaStreamCreat
        K<<<1, 1, 0, stream
        cudaStreamAddC
        K<<<1, 1, 0, stream[ii]>>>();
        cudaDeviceSynchronize();
    }
}
```

```
in kernel K
inside callback 0
error=70, cudaErrorNotPermitted, operation not permitted
in kernel K
in kernel K
inside callback 1
error=70, cudaErrorNotPermitted, operation not permitted
in kernel K
```

Events

- Events can be created.

```
#include <stdio.h>
#include <cuda.h>

int main() {
      cudaEvent_t start, stop;
      cudaEventCreate(&start);
      cudaEventCreate(&stop);
      ...
      cudaEventDestroy(start);
      cudaEventDestroy(stop);
}
```

Events

- Events can help us time.

```
#include <stdio.h>
#include <cuda.h>

int main() {
	cudaEvent_t start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);
	cudaEventRecord(start, 0);
	K<<<...>>>(); cudaDeviceSynchronize();
	cudaEventRecord(stop, 0);
	float elapsedtime;
	cudaEventElapsedTime(&elapsedtime, start, stop);
	printf("time = %f ms\n", elapsedtime);
}
```

Synchronization across Streams

- Callbacks allow us to communicate / synchronize *within* a stream.
- Events allow us to communicate / synchronize *across* streams.

```
cudaStream_t streamA, streamB;
cudaEvent_t event;

cudaStreamCreate(&streamA);
cudaStreamCreate(&streamB);
cudaEventCreate(&event);

K1<<<1,1,0, streamA>>>(arg_1);
cudaEventRecord(event, streamA);
K2<<<1,1,0,streamA>>>(arg_2);

cudaStreamWaitEvent(streamB, event, 0);
K3<<<1,1, 0, streamB>>>(arg_3);
```

K3 won't execute until **K1** is over.
K2 and **K3** may execute concurrently.
Note:K1 is in stream A and K3 is in B.

Source: http://cedric-augonnet.com/

Classwork

- Create a parent stream and N child streams.
- The parent stream enqueus kernels preamble and postamble. Each child stream enqueues a kernel work.
- Make sure all work kernels start and complete in-between preamble and postamble.

```
int N = 16;
cudaStream_t parent_stream, child_streams[N];
cudaEvent_t parent_event, child_events[N];

cudaStreamCreate(&parent_stream);
preamble<<<1,1,0, parent_stream>>>(arg1);

/* Create a Synchronization point in the parent stream */
cudaEventCreate(&parent_event);
cudaEventRecord(parent_event, parent_stream);

/* Create N concurrent streams */
for (i = 0; i < N; i++)
{
   cudaStreamCreate(&child_streams[i]);
   cudaStreamWaitEvent(child_streams[i], parent_event, 0);
   work<<<1,1,0,child_streams[i]>>>(arg2);

   /* Create a Synchronization point in the child stream */
   cudaEventCreate(&child_events[i]);
   cudaEventRecord(child_events[i], child_streams[i]);
   cudaStreamWaitEvent(parent_stream, child_events[i], 0);
}

/* Enqueue work in the parent stream again, this work will not be
 * executed until all work is done in the children. */
postamble<<<1,1,0, parent_stream>>>(arg3);
```

Source: http://cedric-augonnet.com/