CUDA Stream and Concurrency

Some Basics

Qi SUN

July 22, 2021

Department of Computer Science and Engineering
The Chinese University of Hong Kong

Introduction

Introduction

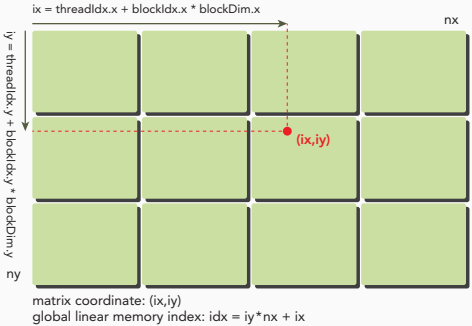


Figure 1: Block and Thread.

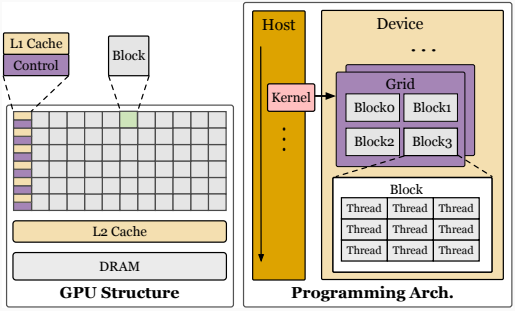


Figure 2: CUDA Programming Architecture.

Introduction

Synchronous APIs

Functions with synchronous behavior block the host thread until they complete.

- Implicit Synchronization
- Explicit Synchronization

Asynchronous APIs

Functions with asynchronous behavior return control to the host immediately after being called.

Stream

Stream

What's Stream?

A CUDA stream refers to a sequence of asynchronous CUDA operations that execute on a device in the order issued by the host code.

Typical operations:

- Host-device data transfer.
  - host-to-device, device-to-host.
  - Usually blocks the host thread.
- Kernel Launches.

NULL Stream

- Default and implicitly declared stream.

```
1  cudaError_t cudaMemcpy (dst, src, bytes,
   ↪  cudaMemcpyHostToDevice);
2  __global__ void kernel<<<grid,block>>>();
```

- Each data transfer is synchronous and forces idle host time.
- Kernel launch is asynchronous. The host application almost immediately resumes execution afterwards.

Non-NULL Stream

- Explicitly declared stream
- To overlap different CUDA operations, you must use non-null streams.

```
1  cudaStream_t stream;
2  cudaError_t cudaStreamCreate(&stream);
3  cudaError_t cudaMemcpyAsync(dst, src, bytes,
   ↪  cudaMemcpyHostToDevice, stream);
4  __global__ void kernel<<<grid, block, shareMemSize,
   ↪  stream>>>();
5  cudaError_t cudaStreamDestroy(cudaStream_t stream);
```

Non-NULL Stream

- Check if operations have completed.

```
1  // force the host to block until all operations in the
   ↪  provided have completed.
2  cudaError_t cudaStreamSynchronize(cudaStream_t
   ↪  stream);
3
4  // check the operations, but does not block the host
   ↪  if not completed.
5  cudaError_t cudaStreamQuery(cudaStream_t stream);
```

Stream

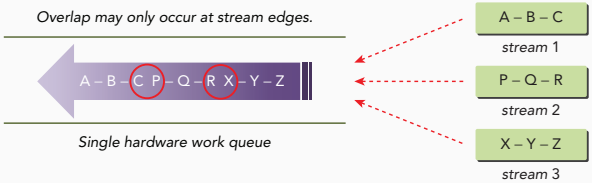
- Non-NULL Stream
  - A Non-NULL stream is an asynchronous stream with respect to the host.
  - All operations applied to it do not block host execution.
- NULL Stream
  - A synchronous stream with respect to the host.
  - Most operations added to the NULL-stream cause the host to block on all preceding operations.
  - Kernel launches are asynchronous. Do the synchronization by ourselves.

Queue

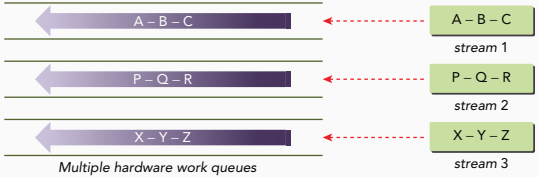
- Hardware work queue.
  - Environment variable: `CUDA_DEVICE_MAX_CONNECTIONS` (1 to 32, default 8)
  - Set the number of concurrent connections from host to device.
- Two copy engine queues
  - One to device, one from device.
  - You can at most overlap two data transfers.

Stream Scheduling

False Dependency



Hyper-Q



CUDA Events and Synchronization

Event

A marker in a CUDA stream associated with a certain point in the flow of operations in the stream.

- Synchronize stream execution.
- Check progress.

```
1  cudaEvent_t event;
2  cudaError_t cudaEventCreate(cudaEvent_t * event);
3  cudaError_t cudaEventRecord(cudaError_t event,
   ↪  cudaStream_t stream=0);
4  cudaError_t cudaEventSynchronize(cudaEvent_t event);
5  cudaError_t cudaEventQuery(cudaEvent_t event);
```

Blocking and Non-Blocking Stream

Non-NULL streams can be further classified into two types:

Blocking Stream

- The streams created using `cudaStreamCreate`.
- The executions of operations in those streams can be blocked by operations in the NULL stream.
- You may need it sometimes.

```
1  kernel_1<<<grid, block, 0, stream>>>();
2  kernel_2<<<grid, block>>>();
3  kernel_3<<<grid, block, 0, stream>>>();
```

Blocking and Non-Blocking Stream

Non-Blocking Streams

- The operations in it cannot be blocked by operations in the NULL stream.
- Specifying `cudaStreamNonBlocking` disables the blocking behavior of Non-NULL streams.

```
1  // cudaStreamDefault: default flag, blocking
2  // cudaStreamNonBlocking: non-blocking
3  cudaError_t cudaStreamCreateWithFlags(cudaStream_t
   ↪  *pStream, unsigned int flags);
```

Implicit Synchronization

- A page-locked host memory allocation
- A device memory allocation
- A device memset
- A memory copy between two addresses on the same device
- A modification to the L1/shared memory configuration

Explicit Synchronization

```
1  // Wait for all computation and communication to finish
2  cudaError_t cudaDeviceSynchronize(void);
3  // Stream and Event
4  cudaError_t cudaStreamSynchronize(cudaStream_t stream);
5  cudaError_t cudaStreamQuery(cudaStream_t stream);
6  cudaError_t cudaEventSynchronize(cudaEvent_t event);
7  cudaError_t cudaEventQuery(cudaEvent_t event);
8  // cross-stream synchronization, control the stream with
   ↪ event of other stream
9  cudaError_t cudaStreamWaitEvent(cudaStream_t stream,
   ↪ cudaEvent_t event);
```

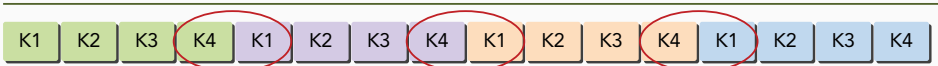
Concurrency

Execution Order

Depth-first

```
1  for (int i = 0; i < n_streams; i++) {
2    kernel_1<<<grid, block, 0, streams[i]>>>();
3    kernel_2<<<grid, block, 0, streams[i]>>>();
4    kernel_3<<<grid, block, 0, streams[i]>>>();
5    kernel_4<<<grid, block, 0, streams[i]>>>();
6  }
```

Issue order from host: depth-first way



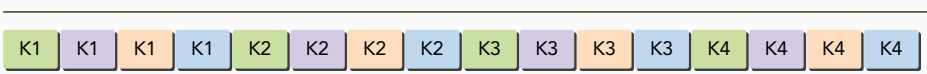
Only the three stream edges are independent.

Execution Order

Breadth-first

```
1  for (int i = 0; i < n_streams; i++)
2    kernel_1<<<grid, block, 0, streams[i]>>>();
3  for (int i = 0; i < n_streams; i++)
4    kernel_2<<<grid, block, 0, streams[i]>>>();
5  for (int i = 0; i < n_streams; i++)
6    kernel_3<<<grid, block, 0, streams[i]>>>();
7  for (int i = 0; i < n_streams; i++)
8    kernel_4<<<grid, block, 0, streams[i]>>>();
```

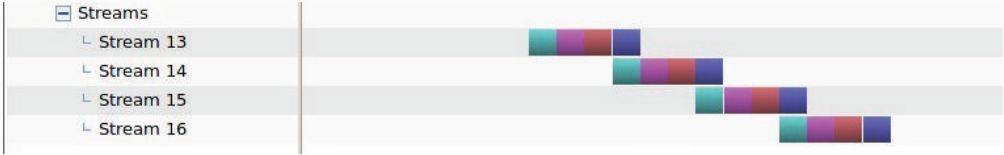
Issue order from host: breadth-first order



There is no dependence between any adjacent kernels.

Execution Order

Depth-first



Breadth-first

Blocking Behavior of the Default Stream

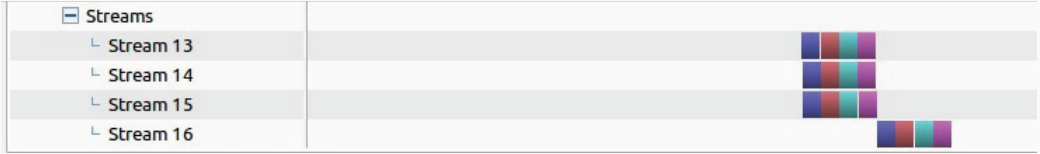
- Any later operations on non-null streams will be blocked until the operations in the default stream complete.

```
1  for (int i = 0; i < n_streams; i++) {
2    kernel_1<<<grid, block, 0, streams[i]>>>();
3    kernel_2<<<grid, block, 0, streams[i]>>>();
4    kernel_3<<<grid, block>>>();
5    kernel_4<<<grid, block, 0, streams[i]>>>();
6  }
```

- Sequentially launch the kernels.

Create Inter-Stream Dependencies

```
1  for (int i = 0; i < n_streams; i++) {
2    kernel_1<<<grid, block, 0, streams[i]>>>();
3    kernel_2<<<grid, block, 0, streams[i]>>>();
4    kernel_3<<<grid, block, 0, streams[i]>>>();
5    kernel_4<<<grid, block, 0, streams[i]>>>();
6    cudaEventRecord(kernelEvent[i], streams[i]);
7    cudaStreamWaitEvent(streams[n_streams-1],
      ↪  kernelEvent[i], 0);  }
```



Overlap Kernel Execution and Data Transfer

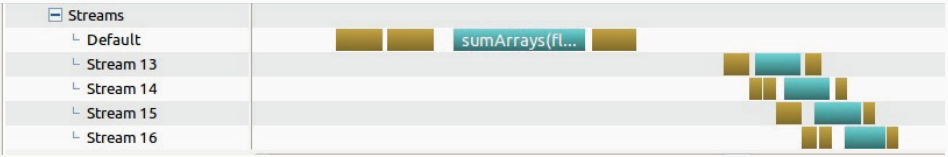
Two copy engine queues

- You can overlap two data transfers, but only if their directionalities differ and they are dispatched to different streams.
- When a dependency exists between the kernel and the transfer:
  - A kernel blocked by preceding data transfers in the same stream.
  - Partition the input and output data into subsets, solve the sub-problems.

```
1  __global__ void sumArrays(float *A, float *B, float *C,
   ↪  const int N) {
2    int idx = blockIdx.x * blockDim.x * threadIdx.x;
3    if (idx < N) C[i] = A[i] + B[i];
4  }
```

Overlap Kernel Execution and Data Transfer

```
1 int iElem = nElem / NSTREAM;
2 for (int i = 0; i < NSTREAM; ++i) {
3   int ioffset = i * iElem;
4   cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], ...);
5   cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], ...);
6   sumArrays<<<grid, block, 0, stream[i]>>>(&d_A[ioffset],
      ↪  &d_B[ioffset], &d_C[ioffset], iElem);
7   cudaMemcpyAsync(&gpuRef[ioffset],&d_C[ioffset], ...);
8 }
```



Thank you!