CONCURRENCY WITH MULTITHREADING

Robert Searles, 7/16/2021
Pre-emptive scheduling

Processes share GPU through time-slicing
Scheduling managed by system

Concurrent scheduling

Processes run on GPU simultaneously
User creates & manages scheduling streams
CUDA STREAMS
STREAM SEMANTICS

1. Two operations issued into the same stream will *execute in issue-order*. Operation B issued after Operation A will not begin to execute until Operation A has completed.

2. Two operations issued into separate streams have *no ordering prescribed by CUDA*. Operation A issued into stream 1 may execute before, during, or after Operation B issued into stream 2.

   ▶ Operation: Usually, `cudaMemcpyAsync` or a kernel call. More generally, most CUDA API calls that take a stream parameter, as well as stream callbacks.
STREAM CREATION AND COPY/COMPUTE OVERLAP

▸ Requirements:

▸ D2H or H2D memcpy from pinned memory

▸ Kernel and memcpy in different, non-0 streams

▸ Code:

```c
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream2>>>(...);
}
```

```c
cudaStreamQuery(stream1); // test if stream is idle
cudaStreamSynchronize(stream2); // force CPU thread to wait
cudaStreamDestroy(stream2);
```
EXAMPLE STREAM BEHAVIOR FOR VECTOR MATH
(assumes algorithm decomposability)

non-streamed

cudaMemcpy(d_x, h_x, size_x, cudaMemcpyHostToDevice);
Kernel<<<b, t>>>(d_x, d_y, N);
cudaMemcpy(h_y, d_y, size_y, cudaMemcpyDeviceToHost);

streamed

for (int i = 0, i<c; i++){
    size_t offx = (size_x/c)*i;
    size_t offy = (size_y/c)*i;
    cudaMemcpyAsync(d_x+offx, h_x+offx, size_x/c, cudaMemcpyHostToDevice, stream[i%ns]);
    Kernel<<<b/c, t, 0, stream[i%ns]>>>(d_x+offx, d_y+offy, N/c);
    cudaMemcpyAsync(h_y+offy, d_y+offy, size_y/c, cudaMemcpyDeviceToHost, stream[i%ns]);}

Similar: video processing pipeline
Kernels or `cudaMemcpy`... that do not specify stream (or use 0 for stream) are using the default stream

Legacy default stream behavior: synchronizing (on the device):

- All device activity issued prior to the item in the default stream must complete before default stream item begins
- All device activity issued after the item in the default stream will wait for the default stream item to finish
- All host threads share the same default stream for legacy behavior
- Consider avoiding use of default stream during complex concurrency scenarios

Behavior can be modified to convert it to an “ordinary” stream

- `nvcc --default-stream per-thread`
- Each host thread will get its own “ordinary” default stream
OTHER CONCURRENCY SCENARIOS

- Host/Device execution concurrency:

```c
Kernel<<<b, t>>>(...);   // this kernel execution can overlap with
cpuFunction(...);        // this host code
```

- Concurrent kernels:

```c
Kernel<<<b, t, 0, streamA>>>(...);   // these kernels have the possibility
Kernel<<<b, t, 0, streamB>>>(...);   // to execute concurrently
```

- In practice, concurrent kernel execution on the same device is hard to witness
- Requires kernels with relatively low resource utilization and relatively long execution time
- There are hardware limits to the number of concurrent kernels per device
- Less efficient than saturating the device with a single kernel
MPI DECOMPOSITION

Very common in HPC

Many legacy codes use MPI + OpenMP

MPI handles inter-node communication

OpenMP provides better shared memory multithreading within each node

How can we add GPUs into the mix?
MULTITHREADING + CUDA STREAMS

Easier than rewriting entire legacy code

Individual OpenMP threads may still have a significant amount of work

Streams allow multiple threads to submit kernels for concurrent execution on a single GPU

Not possible pre-R465

Supported starting with CUDA 11.4/R470
SINGLE GPU EXAMPLE

- Multithreading + Concurrent kernels:

```c
cudaStream_t streams[num_streams];
for (int j=0; j<num_streams; j++)
    cudaStreamCreate(&streams[j]);

#pragma omp parallel for
double b, t;
for (int i=0; i<N; i++)
    Kernel<<<b/N, t, 0, streams[i % num_streams]>>>(...);
```

- Worth it if each thread has enough work to offset kernel launch overhead
- Requires less programmer overhead than rewriting entire codebase to submit single, large kernels to each GPU (remove OpenMP and replace with CUDA)
- Less efficient than saturating the device with streams from a single thread
- Less efficient than saturating the device with a single kernel
MULTI-GPU - STREAMS

- Streams (and cudaEvent) have implicit/automatic device association
- Each device also has its own unique default stream
- Kernel launches will fail if issued into a stream not associated with current device
- cudaStreamWaitEvent() can synchronize streams belonging to separate devices, cudaEventQuery() can test if an event is “complete”
- Simple device concurrency:

```
cudaSetDevice(0);
cudaStreamCreate(&stream0); //associated with device 0
cudaSetDevice(1);
cudaStreamCreate(&stream1); //associated with device 1
Kernel<<<b, t, 0, stream1>>>(...); // these kernels have the possibility
cudaSetDevice(0);
Kernel<<<b, t, 0, stream0>>>(...);  // to execute concurrently
```
MULTI-GPU EXAMPLE

Multithreading + Concurrent kernels:

```
cudaStream_t streams[num_streams];
#pragma omp parallel for
for (int i=0; i<N; i++){
    int j = i % num_streams;   // Stream number
    cudaSetDevice(j % num_gpus); // Round-robin across on-node GPUs
    cudaStreamCreate(&streams[j]); // Associated with device j % num_gpus
    Kernel<<<b/N, t, 0, streams[j]>>>(...);} // execute across threads/streams/GPUs
```

- Multiple threads submitting kernels across a number of streams distributed across available GPUs
- Example: 16 threads, 64 streams, 8 GPUs and N=1024
  - 8 streams per GPU, 16 kernels per stream
- Should have at least 1 stream per GPU
  - More will be optimal; Need as many streams on a GPU as it takes concurrent kernels to saturate that GPU
SINGLE THREAD + CUDA STREAMS
MULTITHREADING + CUDA STREAMS

- Runtimes
  - Single Thread + Default Stream = 0.01879s
  - Single Thread + 8 CUDA Streams = 0.00781s
  - 8 OpenMP Threads + 8 CUDA Streams (without profiling) = 0.00835s
  - 8 OpenMP Threads + 8 CUDA Streams (with profiling) = 0.01798s
- Issue with serialization when using the profiler
  - We’re working on that
MULTI-PROCESS SERVICE (MPS) OVERVIEW

Better solution in terms of performance

Designed to **concurrently** map multiple MPI ranks onto a single GPU

Used when each rank is **too small** to fill the GPU on its own

On Summit, use `-alloc_flags=gpumps` when submitting a job with `bsub`
FUTURE SESSIONS

- MPI/MPS
- CUDA Debugging
HOMEWORK

- Log into Summit (ssh `username@home.ccs.ornl.gov` -> ssh summit)

- Clone GitHub repository:
  - Git clone `git@github.com:olcf/cuda-training-series.git`

- Follow the instructions in the readme.md file:

- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming