Task Parallelism in C and CUDA

ECE 6397
Electrical and Computer Engineering
University of Houston
Dr. David Mayerich
Multiple Devices – Device Enumeration

- `cudaGetDeviceCount()` returns the number of devices $D$ that have compute capability $> 1.0$
- `cudaGetDeviceProperties()` fills a structure with information about an enumerated device

```c
int D; //stores the number of devices
cudaGetDeviceCount(&D); //get # CUDA devices
for (int d = 0; d < D; d++) {
    //for each device
    cudaDeviceProp prop; //create a property structure
    cudaGetDeviceProperties(&prop, d); //get properties
    //output the device ID, compute capability, and name
    std::cout<<"("<<d<<"): "<<prop.major<<"."<<prop.minor<<"     "<<prop.name<<std::endl;
}
```
Multiple Devices – Device Selection

- `cudaSetDevice()` specifies the device that will be sent commands
  - Default is device 0 (zero)

```c
size_t N;
float* f;
...
cudaSetDevice(0);  // set the current device to 0
float* gpu_f0;    // create a pointer
cudaMalloc(&gpu_f0, N * sizeof(float));  // allocate memory on device 0
cudaMemcpy(gpu_f0, f, N * sizeof(float), cudaMemcpyHostToDevice);  // copy to device 0
cudaSetDevice(1);  // set the current device to 0
float* gpu_f1;    // create a pointer
cudaMalloc(&gpu_f1, N * sizeof(float));  // allocate memory on device 0
cudaMemset(gpu_f1, 0, N * sizeof(float));  // set all values to zero
```
Concurrency

- CUDA specifies that the following actions can be done concurrently:
  - Host computation
  - Device computation (kernels)
  - Memory copies from host to device
  - Memory copies from device to host
  - Memory copies on a device
  - Memory copies between devices
- The extent of concurrency available can be queried using \texttt{cudaGetDeviceProperties()}
- Concurrency with the host is the default behavior for kernels:
  - CUDA kernel calls return execution to the host thread
  - Control is handled asynchronously through the CUDA library
  - Multiple kernels sent to the GPU will run concurrently with the host thread that sends them
    - This is why precise timing is harder and requires inserting events
CUDA Streams

- A *stream* is a series of CUDA commands that are executed in serial
- Concurrent operations are placed in different streams
- CUDA streams are executed concurrently
- The order of execution *between* streams is not defined

- kernel 0A will execute before 0B
- 0A may *or may not* execute before 1C (which *will* execute before 1D)
Memory Copies Concurrent with the Host

- **asyncEngineCount** property specifies the number of memory copies that can be performed concurrently *while* computing
  - **asyncEngineCount == 0** can’t copy while executing a kernel
  - **asyncEngineCount == 1** can copy to/from memory while executing
  - **asyncEngineCount == 2** can copy to and from memory while executing

- Concurrent copies with the host
  - Memory copies to/from the host do not run concurrently with the host thread by default
  - `cudaMemcpy()` includes a synchronization barrier – host code doesn’t continue until the copy is resolved
  - The host program can change memory
  - The host memory can be paged (may not actually be in memory)
Concurrent Memory Copies

• The operating system optimizes performance using *paging*, which occasionally copies unused data from main memory to secondary storage (usually a hard drive)

• Concurrent copies to/from the host are possible
  • The host memory used in the copy must be *page locked*
  • The user should not change values in memory while the copy occurs

• Page locked memory can be allocated through the CUDA API

```c
size_t N = 1000;
float* hostPtr;
cudaMallocHost(&hostPtr, N * sizeof(float));
```
Concurrent Memory Copies

```c
size_t N = 1000;
const size_t S = 2;
float* ptr[S];
float* gpu_ptr[S];
cudaStream_t stream[S];
for(int s = 0; s < S; s++){
    cudaStreamCreate(&streams[s]);
    cudaMallocHost(&ptr[s], N * sizeof(float));
    cudaMalloc(&gpu_ptr[s], N * sizeof(float));
}
...
for(int i = 0; i < S; i++){
    cudaMemcpyAsync(gpu_ptr[i], ptr[i], N * sizeof(float), cudaMemcpyHostToDevice, stream[i]);
    kernel<<<blocks, threads, 0, stream[i]>>>(gpu_ptr[i], N);
    cudaMemcpyAsync(ptr[i], gpu_ptr[i], N * sizeof(float), cudaMemcpyDeviceToHost, stream[i]);
}
```
Concurrent Memory Copies - Examples

\[ \text{asyncEngineCount} = \emptyset \]

\[ \text{asyncEngineCount} = 1 \]
Destroy Streams and Synchronize

• Use cudaStreamDestroy() to destroy the CUDA streams

```c
for(int s = 0; s < S; s++)
    cudaStreamDestroy(stream[s]);
```

• Applies a synchronization barrier – cudaStreamDestroy() doesn’t return until all streams have finished

• Explicit synchronization
  • cudaDeviceSynchronize() – waits for all streams on the device to finish
  • cudaStreamSynchronize(cudaStream_t s) – waits for a stream to finish
  • cudaStreamWaitEvent(cudaStream_t s, cudaEvent_t e, 0) – waits for an event
Synchronization Across Streams

• Calculate  \( C[i] = e^{A[i]} + \sqrt{B[i]} \) where \( i = [0 \ N) \)

• How do we guarantee that kernel 2 runs after kernel 0 is completed?

```c
// On host A
cudaEventCreate(&e0);
cudaEventRecord(e0);
streamWaitEvent(s1, e0, 0);
```
Peer-to-Peer Memory Access

• Using a pointer to memory on one device to access data from another device
  • `cudaDeviceCanAccessPeer(int* canAccessPeer, int dev, int peer)`
  • `canAccessPeer` is 1 if device `peer` can access memory on device `dev`

• Enable peer-to-peer memory access
  • `cudaDeviceEnablePeerAccess(int device, unsigned int flags)`—flags should be 0
  • creates a unified address space for both devices
Peer-to-Peer Memory Access – Example

cudaSetDevice(0); //current device is 0
float f0; //create a pointer
size_t N = 1024; //size of an array
cudaMalloc(&f0, N * sizeof(float)); //allocate memory on device 0
kernel<<<blocks, threads>>>(f0); //launch a kernel on 0

//launch a kernel on the second device if possible
int p2p;
cudaDeviceCanAccessPeer(&p2p, 1, 0); //can device 1 access memory on 0?
if(p2p){
    cudaSetDevice(1); //current device is 1
    cudaDeviceEnablePeerAccess(0, 0); //enable peer access
    kernel<<<blocks, threads>>>(f0); //thread on 1 can access memory on 0
}
Peer-to-Peer Memory Copy API Example

• Copy memory from one device to another using the CUDA API

cudaSetDevice(0);
float* p0; //set current device to 0
float* p1; //create a pointer
size_t N = 1024; //array size
cudaMalloc(&p0, N * sizeof(float)); //allocate memory on device 0
cudaSetDevice(1); //switch to device 1
float* p1;
cudaMalloc(&p1, N * sizeof(float)); //allocate memory on device 1
kernel<<<blocks, threads>>>(p1); //launch a kernel on device 1
cudaSetDevice(0); //go back to device 0
cudaMemcpyPeer(p0, 0, p1, 1, N * sizeof(float)); //copy from 1 to 0
kernel<<<blocks, threads>>>(p0); //launch a kernel on device 0
Unified Virtual Address Spaces

• Allows a single address space to be used for all host and device pointers
• Virtual pointers can be acquired for any memory allocated using the API
• Device information is available through the device pointer
  • cudaPointerGetAttributes( struct cudaPointerAttributes* att, void* ptr)
  • fills an attribute structure describing pointer ptr

```c
struct cudaPointerAttributes {
    int device; //device where memory is allocated
    void* devicePointer; //pointer used to access memory on this device
    void* hostPointer; //pointer to use if accessing memory on a host
    cudaMemoryType memoryType; //cudaMemoryTypeHost or cudaMemoryTypeDevice
}
```
Unified Virtual Address Space Example

cudaSetDevice(0);
float* p0;
size_t N = 1024;
cudaMalloc(&p0, N * sizeof(float));
float* vp;
cudaPointerAttributes attrib;
cudaPointerGetAttributes(&attrib, p0);
vp = attrib.devicePointer;
kernel<<<blocks, threads>>>(vp);
cudaSetDevice(1);
kernel<<<blocks, threads>>>(vp);

//set current device to 0
//create a pointer
//array size
//allocate memory on device 0
//create a pointer attribute structure
//get the attributes of p0
//store the virtual device pointer
//launch a kernel on device 0
//go back to device 1
//launch a kernel on device 1
C++ Standard Library Multi-threading

• C++ (C++11) provides support for task parallelism
  • thread library <thread>
  • specify a function pointer to execute at the same time

```cpp
#include <thread>

void thread_function(){
  //this function is executed in a thread
}

int main(){
  std::thread t(thread_function); //create a thread and launch it
  //execution continues in the host
  t.join(); //wait for the thread to finish
}
```
C++ Multi-threading Example


```cpp
#include <thread>

void add(float* x, float* y, size_t n){
    //add all values, save result in x
    for(size_t i = 0; i < n; i++)
        x[i] += y[i];
}

void mul(float* x, float* y, size_t n){
    //multiply all values, save in x
    for(size_t i = 0; i < n; i++)
        x[i] *= y[i];
}

int main(){
    size_t N;              //array size
    float* B;              //declare arrays
    float* C;
    float* D;
    float* E;
    ...
    //allocate and load

    std::thread t0(add, B, C);
    std::thread t1(multiply, D, E);
    t0.join();              //sync barrier (t0)
    t1.join();              //sync barrier (t0)
    add(B, D);              //add final result
}
```

where $i \in [1, N)$