The CUDA Language and API: Device Queries and Threading

ECE 6397
Electrical and Computer Engineering
University of Houston
Dr. David Mayerich
CUDA platform

- Application Programming Interface (API)
  - C/C++ function calls designed to implement GPU features
  - Allocating space in device memory, copying data, getting device information
  - Setting up special types of memory on the GPU

- CUDA language
  - programming language for implementing code that executes on CUDA devices
  - traditional ANSI C – all C code is acceptable in CUDA
  - includes several special CUDA keywords:
    - specify regions of the code that are highly data parallel
    - describe how these parallel regions are executed
    - describe how resources are allocated to optimize efficiency
CUDA kernels

• *kernel* – a data parallel function that is executed on the GPU
CUDA compiler

• *nvcc* – CUDA compiler, strips CUDA code from C/C++
  • CUDA code is converted to device code
    • *parallel thread execution* (PTX) code is compiled just-in-time specifically for the available CUDA hardware device
    • runtime compiler converts PTX into device binary code that can be executed directly on the CUDA device (GPU)
  • C/C++ code is passed to the native compiler – gcc, g++, Visual Studio, etc.
    • C/C++ is compiled and linked as normal
    • code is linked to CUDA libraries that handle links to device functions

• This operation is generally seamless and transparent to the programmer

• There may be reasons to look at PTX code for profiling and optimization
  • generally results in the programmer making changes to CUDA code and re-compiling
CUDA compiler

standard C/C++ AND CUDA code

nvcc compiler

standard C/C++

gcc, g++, Visual Studio

CPU binary

device code

(device code (PTX))

device compiler

GPU binary

heterogeneous system
CUDA program structure

1) use the CUDA API to query for compatible devices
   
   ```c
   cudaMemcpy_t cudaGetDeviceCount(int* num)
   cudaError_t cudaGetDeviceProperties(cudaDevProp* prop, int dev)
   ```

2) allocate memory on the CUDA device
   ```c
   cudaMemcpy_t cudaMalloc(void** devPtr, size_t size)
   ```

3) copy data from main memory to the CUDA device
   ```c
   cudaMemcpy_t cudaMemcpy(void* dest,
                              const void* source,
                              size_t size,
                              cudaMemcpyKind type)
   ```

4) launch the kernel (discussed later)

5) copy results from the CUDA device to main memory
   ```c
   cudaMemcpy_t cudaMemcpy(⋯)
   ```
Device Properties

• get the number of devices (GPUs) available
  • code can be run on multiple devices
    cudaError_t cudaGetDeviceCount(int* n)  //get # of CUDA compatible devices

• the programmer can control what devices execute code
  cudaError_t cudaGetDevice(int* d)  //get the current device #
  cudaError_t cudaSetDevice(int d)  //send future commands to this device

• check for compatibility
  • test for compute capability – determine what features the device has
  • test for available memory – specifies how much data can be copied to the device
    cudaError_t cudaGetDeviceProperties(cudaDevProp* prop, int dev)
Device Properties

• *global memory* – defines how large of a problem you can solve with a single kernel

• *compute capability* – specifies the set of features that the GPU is capable of
  • most newer devices will do everything you will need for now
  • examples:
    - 1.1 atomic operations were introduced in (2006)
    - 1.3 64-bit floating point operations were introduced in (2008 – 2009)
    - 6.2 current capability (2016)

```
struct cudaDeviceProp{
    char name[256];
    size_t totalGlobalMem;
    ...
    //includes several important GPU features
    int major;
    int minor;
    ...
    //more GPU features
}
```
Error Checking

• Almost all CUDA functions return a cudaError_t value
  • Check This Every Time!
    cudaError_t =
    cudaSuccess //operation successful
    cudaErrorMemoryAllocation //not enough memory
    cudaErrorInvalidValue //a parameter is wrong
    cudaErrorInvalidDevicePointer //didn’t allocate memory?

• Seriously Check It. Trust me.

cudaError_t error;
error = cudaMalloc(gpuPtr, N * sizeof(float));
if( error != cudaSuccess){
    std::cout<<“Error allocating memory”<<std::endl;
    exit(1);
}
Error Checking – a useful macro

```cpp
static void HandleError( cudaError_t err, const char *file, int line ) {
    if (err != cudaSuccess) {
        std::cout << cudaGetErrorString( err ) << " in " << file << " at line " << line;
    }
}
#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))
```

• Example:

```
HANDLE_ERROR( cudaMalloc(&gpuPtr, N * sizeof(float)) );
```

• this will display a string describing the error, along with the file and line number where the error occurred
Copy Data to a Device

```c
void main(int argc, char* argv[]){
    int N;               //length of an array
    ...
    //get array length
    float* x;            //declare a pointer to main memory
    x = (float*) malloc(N * sizeof(float));  //allocate an array in main memory
    ...
    //load array values
    float* gpu_x;        //declare another pointer (will point to device memory)
    HANDLE_ERROR(        //error checking
        cudaMalloc(&gpu_x, N * sizeof(float))        //allocate memory on device
    );                    //will return an error or cudaSuccess
    HANDLE_ERROR(        //do whatever you want to the array gpu_x on the device
        cudaMemcpy(gpu_x, x, N * sizeof(float), cudaMemcpyHostToDevice)
    );
    ...
    //copy the array back to main memory
    HANDLE_ERROR(        //error checking
        cudaMemcpy(x, gpu_x, N * sizeof(float), cudaMemcpyDeviceToHost)
    );
}
```
Execution Threads

- **thread** – a sequential set of instructions executed alongside other threads in a processor
  - *multithreading* – the ability for a processor to execute several processes simultaneously
  - used for both data and task parallel processes
  - threads do not have to be executed in parallel
    - one processor can execute multiple threads
    - *temporal multithreading* – threads execute by taking
    - *simultaneous multithreading* – threads can share the same pipeline

- Rely on the operating system to allocate resources based on local optimization rules

- Never assume which order threads will be executed, or when they will stop or start
CUDA Thread Resources

• Threads are executed in parallel on the GPU
• A Tesla P100 has 56 SMs, each with 64 cores = 3584 total cores
  • At least this many threads can be executed *simultaneously*
  • Additional threads can be accommodated with pipelining
• Some resources are shared:
  • *Global memory* is shared across all threads
  • *Shared memory* is shared across threads running on a single SM
  • *Registers* are only available to a single thread
  • Proximity $\approx$ speed
    • Global memory (200 – 400 clock cycle latency)
    • Shared memory (20 – 40 clock cycle latency)
    • Registers are fastest ($\approx$11 clock cycles for a calculation involving registers)
CUDA Function Declarations

• CUDA contains three additional keywords for function declarations
• Declarations specify how the code is to be compiled by `nvcc`
  ```
  __host__ int abs(int v){
    if(v < 0) return -v
    return v;
  }
  ```
  ```
  __device__
  int abs(int v){
    if(v < 0) return -v
    return v;
  }
  ```
  ____global__
  called and executed on the host
  called and executed on a device
  called on the host, executed on a device

• These keywords start the function declaration:
__host__ functions

• Mostly ignored by nvcc
  • kernels can be launched from host functions
    • kernel launches are stripped from the C/C++ and replaced with links to CUDA launch instructions

• host functions can only be called by other host functions
• functions without CUDA declarations are assumed to be __host__ functions
• A standard C function:

```c
int abs(int v){
    if(v < 0) return -v;
    return v;
}
```

is identical to

```c
__host__ int abs(int v){
    if(v < 0) return -v;
    return v;
}
```
__device__ functions

• Compiled completely by nvcc – not passed to the host compiler
• Only executable on a device
• Only __device__ functions can be called by __device__ functions
• Once you’re executing on the device, the only way off is to return

• Functions can be declared as both __host__ and __device__

```c
__host__ __device__
int abs(int v){
    if(v < 0) return -v;
    return v;
}
```

• Two versions of the function are created: one in PTX (by nvcc) and the other by the native compiler
__global__ functions

- Used to specify kernels (functions that execute on the GPU)
- Kernels are called by host functions
  - Special parameters are specified to define how the kernel is launched
  - Once launched, the kernel runs on the device
  - Execution continues in the host function that launched the kernel
- __global__ functions are the only device functions that can be called from a host function
- __global__ functions must have a void return type
CUDA Thread Organization

- The programmer arranges threads into a **grid**
- A grid is composed of **blocks**
- A block is composed of threads
- The processor executes threads in **warps**
Launching Threads

• User specifies parameters of the grid layout

• Example:
  • a grid dimension of 100 blocks
  • a block dimension of 256 threads
  • GPU launches $100 \times 256 = 25600$ threads

• All threads execute the same kernel

• Launch a kernel from a host function:
  kernelFunc <<< gridDim, blockDim >>>(⋯);
  kernelFunc <<< 100, 256 >>>(⋯); //launch 25600 threads

• configuration parameters – CUDA code in a host function that specifies how a kernel is launched
Declaring and launching CUDA kernels

```c
//define a kernel
__global__
void kernelAbs(float* in, float* out, int N){
    ...
    ...
}

//launch the kernel in a host function
void main(int argc, char* argv[]){
    ...
    //initialize function parameters
    gridDim = ...;
    //initialize configuration parameters
    blockDim = ...;

    //launch kernel
    kernelAbs<<<gridDim, blockDim>>>(gpu_in, gpu_out, N);
}
```
Data Parallelism in Kernels

• If all threads execute the same kernel, how do you operate on different data?

• Each thread has an independent set of parameters available assigned by CUDA at launch
  - `gridDim` specifies the number of blocks in the grid
  - `blockDim` specifies the number of threads in a block
  - `blockIdx` gives a unique identifier for the current block
  - `threadIdx` gives a unique identifier for the current thread

• Each parameter is a *triple*: \((x, y, z)\)
  - grids and blocks can be specified in three dimensions
    - `gridDim.x` gives the number of blocks along \(x\)
    - `threadIdx.y` gives the identifier along the \(y\) dimension within this block

• For now, we will stick to 1D grids (only the \(x\) dimension is used)
**Data Parallelism in Kernels**

```c
__global__ void kernel(...){
    int i;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    ...
}
```

- What is the value of \( i \)?
  \[ i = (m - 1) \cdot n + 31 \]

---

<table>
<thead>
<tr>
<th></th>
<th>block 0</th>
<th>block 1</th>
<th>...</th>
<th>block ( m - 2 )</th>
<th>block ( m - 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_0 )</td>
<td>...</td>
<td>( t_{31} )</td>
<td>( t_{32} )</td>
<td>...</td>
<td>( t_{n-1} )</td>
</tr>
<tr>
<td>( t_0 )</td>
<td>...</td>
<td>( t_{31} )</td>
<td>( t_{32} )</td>
<td>...</td>
<td>( t_{n-1} )</td>
</tr>
<tr>
<td>( t_0 )</td>
<td>...</td>
<td>( t_{31} )</td>
<td>( t_{32} )</td>
<td>...</td>
<td>( t_{n-1} )</td>
</tr>
<tr>
<td>( t_0 )</td>
<td>...</td>
<td>( t_{31} )</td>
<td>( t_{32} )</td>
<td>...</td>
<td>( t_{n-1} )</td>
</tr>
</tbody>
</table>

---

| \( A[i] \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|-------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| \( t_0 \) | \( t_1 \) | \( t_2 \) | \( t_3 \) | \( t_4 \) | \( t_5 \) | \( t_0 \) | \( t_1 \) | \( t_2 \) | \( t_3 \) | \( t_4 \) | \( t_5 \) | \( t_0 \) | \( t_1 \) | \( t_2 \) | \( t_3 \) | \( t_4 \) | \( t_5 \) | \( t_0 \) | \( t_1 \) | \( t_2 \) | \( t_3 \) | \( t_4 \) | \( t_5 \) |

---

<table>
<thead>
<tr>
<th>( B_0 )</th>
<th>( B_1 )</th>
<th>( B_2 )</th>
<th>( B_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_0 )</td>
<td>( t_1 )</td>
<td>( t_2 )</td>
<td>( t_3 )</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>( t_5 )</td>
<td>( t_0 )</td>
<td>( t_1 )</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>( t_3 )</td>
<td>( t_4 )</td>
<td>( t_5 )</td>
</tr>
<tr>
<td>( t_6 )</td>
<td>( t_7 )</td>
<td>( t_8 )</td>
<td>( t_9 )</td>
</tr>
</tbody>
</table>
Absolute Value of an Array – CPU

calculate $B_i = |A_i|$ where $i \in [0, N - 1]$

//declare and define the abs( ) function
float abs(float x){
    if(x < 0) return -x;
    return x;
}

int main(int argc, char* argv[]){
    //N is the number of values in the array
    size_t bytes = sizeof(float) * N; //calculate the number of bytes in the array
    float* A = malloc(bytes); //allocate space for A
    //load the array from a file
    float* B = malloc(bytes); //allocate space for B

    for(int i = 0; i < N; i++) //for each value in the array
        B[i] = abs(A[i]); //calculate the absolute value
}
int main(int argc, char* argv[]){
    //N is the number of values in the array
    size_t bytes = sizeof(float) * N; //calculate the number of bytes in the array
    float* A = malloc(bytes); //allocate space for A
    //load the array from a file
    float* B = malloc(bytes); //allocate space for B
    float* gpuA; //create a pointer (device memory)
    //allocate memory on the CUDA device and copy A
    HANDLE_ERROR(cudaMalloc(&gpuA, bytes)); //allocate device memory
    HANDLE_ERROR(cudaMemcpy(gpuA, A, bytes, cudaMemcpyHostToDevice)); //copy A to gpuA (on device)
    float* gpuB; //create a pointer (device memory) to store the result
    HANDLE_ERROR(cudaMalloc(&gpuB, bytes)); //allocate device memory

    HANDLE_ERROR(kernelAbs<<<N, 1>>>(gpuB, gpuA, N); //launch the kernel

    HANDLE_ERROR(cudaMemcpy(B, gpuB, bytes, cudaMemcpyDeviceToHost)); //copy result to main memory
    //the result is in B
}

Absolute Value of an Array: __global__

\[
\begin{array}{cccccccccccc}
\hline
B_0 & B_1 & B_2 & B_3 & B_4 & B_5 & B_6 & B_7 & B_8 & B_9 & B_{10} & B_{11} & \cdots & B_{N-3} & B_{N-2} & B_{N-1} \\
\end{array}
\]

__global__ void kernelAbs(float* out, float* in, int N){
    int i = blockIdx.x;
    if(in[i] < 0) out[i] = -in[i];
    out[i] = in[i];
}

• What are the configuration parameters for the following grid layout?

\[
\begin{array}{cccccccccccc}
\hline
t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & \cdots & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 \\
\hline
B_0 & \cdots & \text{...} & t_0 & t_1 & B_M-1 \\
\end{array}
\]

kernelAbs<<< M, 6 >>>(gpuB, gpuA, N);

• How about the kernel index \( i \)?

\[ i = blockIdx.x \times blockDim.x + threadIdx.x; \]
Grid Constraints

- There are a limited number of threads per block
  - currently usually around 1024
- The grid and block dimensions are all limited
  - max size of any block dimension is usually the max number of threads (≈1024)
  - max size of any grid dimension is usually ≈65535
- All parameters can be queried with
  \[
  \text{cudaError_t cudaGetDeviceProperties(cudaDevProp* prop, int dev)}
  \]

```c
struct cudaDeviceProp{
    char name[256];
    size_t totalGlobalMem;
    int maxThreadsPerBlock;  // maximum total threads per block
    int maxThreadsDim[3];    // maximum threads in every dimension
    int maxGridSize[3];      // maximum blocks in every dimension
    ...                      // includes several important GPU features
    int major;
    int minor;
    ...                      // more GPU features
}
```
Warps

- GPU block architecture is broadly a SPMD:
  - single program multiple data (SPMD) – programming architecture where a single program is executed on data
  - threads generally don’t have to execute the same instruction at the same time
- The GPU organizes threads within a block into warps
- warp – a series of threads in a block that execute the same instruction in a SIMD manner
- The warp size is generally ≈32 threads
- Help mitigate the problem of thread divergence
  - divergent threads (loops, conditionals) only stall the warp
- Scattered divergent threads can still cause damage in the form of stalls!
How CUDA Grids Map to Architecture

• All threads in a block will run on the same streaming multiprocessor (SM)
  • an SM can execute multiple threads
  • not all threads will start or finish at the same time
  • all threads in a block have access to the same shared memory

• Maximizing throughput \(\approx\) maximizing occupancy
  • maximize the number of threads running simultaneously on an SM
  • limiting factors include:
    • number of registers used (variables in the kernel)
    • shared memory used (discussed later)
    • stalls (thread divergence)
Techniques for Designing Grid Layouts

• If you expect low thread divergence, make blocks a multiple of the warp size (usually 32)
• Minimizing the number of registers used in the kernel will allow more blocks to be executed
• Map the grid onto your data
  • 1D arrays should use a 1D grid (threadIdx.x, blockDim.x, etc.)
  • 2D problems often use a 2D grid (x, y)
    • image processing: map one thread to each output pixel
    • matrix multiplication: map one thread to each value $M_{ij}$ in the output matrix

```c
struct cudaDeviceProp{
    char name[256];
    size_t totalGlobalMem;
    int regsPerBlock;
    int warpSize;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    ...
    int major;
    int minor;
    ...
}
```