CUDA
A Parallel Computing Architecture for NVIDIA GPUs

Supports standard languages and APIs
- C
- OpenCL
- Fortran (PGI)
- DX Compute

Supported on common operating systems:
- Windows
- Mac OS
- Linux
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

```c
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...```

threadID

0 1 2 3 4 5 6 7
Example: Increment Array Elements

**CPU program**

```c
void increment_cpu(float *a, float b, int N) {
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}

void main()
{
    ....
    increment_cpu(a, b, N);
}
```

**CUDA program**

```c
__global__ void increment_gpu(float *a, float b, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}

void main()
{
    ....
    dim3 dimBlock(blocksize);
    dim3 dimGrid(ceil(N / (float)blocksize));
    increment_gpu<<<<<dimGrid, dimBlock>>>(a, b, N);
}
```
Outline of CUDA Basics

- Basics Memory Management
- Basic Kernels and Execution on GPU
- Coordinating CPU and GPU Execution
- Development Resources

See the Programming Guide for the full API
**Memory Spaces**

- **CPU and GPU have separate memory spaces**
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
    - Very similar to corresponding C functions

- **Pointers are just addresses**
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa
Host (CPU) manages device (GPU) memory:
- `cudaMalloc (void ** pointer, size_t nbytes)`
- `cudaMemset (void * pointer, int value, size_t count)`
- `cudaFree (void* pointer)`

```c
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```
Data Copies

cudaMemcpy (void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);

- returns after the copy is complete
- blocks CPU thread until all bytes have been copied
- doesn’t start copying until previous CUDA calls complete

enum cudaMemcpyKind
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Non-blocking memcopies are provided
Allocate CPU memory for $n$ integers
Allocate GPU memory for $n$ integers
Initialize GPU memory to 0s
Copy from GPU to CPU
Print the values
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
Code Walkthrough 1

```c
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMemcpy( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemcpy( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
Code Walkthrough 1

#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
   cudaMalloc( (void**)&d_a, num_bytes);

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i]);
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
Basic Kernels and Execution on GPU
CUDA Programming Model

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables
Thread Hierarchy

Threads launched for a parallel section are partitioned into thread blocks
- Grid = all blocks for a given launch

Thread block is a group of threads that can:
- Synchronize their execution
- Communicate via shared memory
IDs and Dimensions

- **Threads:**
  - 3D IDs, unique within a block

- **Blocks:**
  - 2D IDs, unique within a grid

- **Dimensions set at launch time**
  - Can be unique for each grid

- **Built-in variables:**
  - threadIdx, blockIdx
  - blockDim, gridDim
C function with some restrictions:
- Can only access GPU memory
- No variable number of arguments
- No static variables
- No recursion

Must be declared with a qualifier:
- __global__: launched by CPU, cannot be called from GPU must return void
- __device__: called from other GPU functions, cannot be launched by the CPU
- __host__: can be executed by CPU
- __host__ and __device__ qualifiers can be combined
  sample use: overloading operators
Code Walkthrough 2

- Build on Walkthrough 1
- Write a kernel to initialize integers
- Copy the result back to CPU
- Print the values
```c
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```
Launching kernels on GPU

Launch parameters:
- grid dimensions (up to 2D), \texttt{dim3} type
- thread-block dimensions (up to 3D), \texttt{dim3} type
- shared memory: number of bytes per block
  - for extern smem variables declared without size
  - Optional, 0 by default
- stream ID
  - Optional, 0 by default

\texttt{dim3 grid(16, 16);}  
\texttt{dim3 block(16, 16);}  
\texttt{kernel<<<grid, block, 0, 0>>>(...);}  
\texttt{kernel<<<32, 512>>>(...);}
#include <stdio.h>

__global__ void kernel( int *a )
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  a[idx] = 7;
}

int main()
{
  int dimx = 16;
  int num_bytes = dimx*sizeof(int);

  int *d_a=0, *h_a=0; // device and host pointers
  h_a = (int*)malloc(num_bytes);
  cudaMemcpy( (void**)&d_a, num_bytes );
  cudaMalloc( (void**)&d_a, num_bytes);

  if( 0==h_a || 0==d_a )
  {
    printf("couldn't allocate memory\n");
    return 1;
  }

  cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

  dim3 grid, block;
  block.x = 4;
  grid.x = dimx / block.x;

  kernel<<<grid, block>>>( d_a );

  cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

  for(int i=0; i<dimx; i++)
    printf("%d ", h_a[i] );
  printf("\n");

  free( h_a );
  cudaFree( d_a );
  return 0;
}
Kernel Variations and Output

```c
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```c
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```c
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
Build on Walkthrough 2
Write a kernel to increment $n \times m$ integers
Copy the result back to CPU
Print the values
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx] = a[idx]+1;
}

 Kernel with 2D Indexing
int main()
{
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMemcpy( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int row=0; row<dimy; row++)
    {
        for(int col=0; col<dimx; col++)
        {
            printf("%d ", h_a[row*dimx+col] );
            printf("\n");
        }
    }

    free( h_a );
    cudaFree( d_a );

    return 0;
}

__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx  = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially

- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD ... can easily deadlock

- Independence requirement gives scalability
Blocks must be independent

- Thread blocks can run in any order
  - Concurrently or sequentially
  - Facilitates scaling of the same code across many devices

Scalability
Coordinating CPU and GPU Execution
Synchronizing GPU and CPU

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel starts executing once all previous CUDA calls have completed

- Memcopies are synchronous
  - control returns to CPU once the copy is complete
  - copy starts once all previous CUDA calls have completed

- `cudaThreadSynchronize()`
  - blocks until all previous CUDA calls complete

- Asynchronous CUDA calls provide:
  - non-blocking memcopies
  - ability to overlap memcopies and kernel execution
CUDA Error Reporting to CPU

- All CUDA calls return error code:
  - except kernel launches
  - cudaError_t type

- cudaError_t cudaGetLastError(void)
  - returns the code for the last error ("no error" has a code)

- char* cudaGetErrorString(cudaError_t code)
  - returns a null-terminated character string describing the error

printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
CUDA Event API

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed
- asyncAPI sample in CUDA SDK

```c
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernelforward <<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start); cudaEventDestroy(stop);
```
Device Management

CPU can query and select GPU devices

- `cudaGetDeviceCount( int* count )`
- `cudaSetDevice( int device )`
- `cudaGetDevice( int *current_device )`
- `cudaGetDeviceProperties( cudaDeviceProp* prop, int device )`
- `cudaChooseDevice( int *device, cudaDeviceProp* prop )`

Multi-GPU setup:

- device 0 is used by default
- one CPU thread can control one GPU
  - multiple CPU threads can control the same GPU
    - calls are serialized by the driver
Shared Memory

- **On-chip memory**
  - 2 orders of magnitude lower latency than global memory
  - Order of magnitude higher bandwidth than gmem
  - 16KB per multiprocessor
  - NVIDIA GPUs contain up to 30 multiprocessors

- **Allocated per threadblock**
- **Accessible by any thread in the threadblock**
  - Not accessible to other threadblocks

- **Several uses:**
  - Sharing data among threads in a threadblock
  - User-managed cache (reducing gmem accesses)
Using shared memory

Size known at compile time

```c
__global__ void kernel(...) {
  ...
  __shared__ float sData[256];
  ...
}

int main(void) {
  ...
  kernel<<<nBlocks, blockSize>>>(...);
  ...
}
```

Size known at kernel launch

```c
__global__ void kernel(...) {
  ...
  extern __shared__ float sData[];
  ...
}

int main(void) {
  ...
  smBytes = blockSize * sizeof(float);
  kernel<<<nBlocks, blockSize, smBytes>>>(...);
  ...
}
```
Example of Using Shared Memory

- Applying a 1D stencil:
  - 1D data
  - For each output element, sum all elements within a radius
  - For example, radius = 3
    - Add 7 input elements
Implementation with Shared Memory

- 1D threadblocks (partition the output)
- Each threadblock outputs `BLOCK_DIMX` elements
  - Read input from gmem to smem
    - Needs `BLOCK_DIMX + 2*RADIUS` input elements
  - Compute
  - Write output to gmem

Input elements corresponding to output as many as there are threads in a threadblock

© NVIDIA Corporation 2009
__global__ void stencil( int *output, int *input, int dimx, int dimy )
{
    __shared__ int s_a[BLOCK_DIMX+2*RADIUS];

    int global_ix = blockIdx.x*blockDim.x + threadIdx.x;
    int local_ix   = threadIdx.x + RADIUS;

    s_a[local_ix] = input[global_ix];

    if ( threadIdx.x < RADIUS )
    {
        s_a[local_ix - RADIUS] = input[global_ix - RADIUS];
        s_a[local_ix + BLOCK_DIMX + RADIUS] =
            input[global_ix + BLOCK_DIMX + RADIUS];
    }

    __syncthreads();

    int value = 0;
    for( offset = -RADIUS; offset<=RADIUS; offset++ )
        value += s_a[ local_ix + offset ];

    output[global_ix] = value;
}
Thread Synchronization Function

```c
void __syncthreads();
```

Synchronizes all threads in a **thread-block**
- Since threads are scheduled at run-time
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Should be used in conditional code only if the conditional is uniform across the entire thread block
Memory Model Review

- **Local storage**
  - Each thread has own local storage
  - Mostly registers (managed by the compiler)
  - Data lifetime = thread lifetime

- **Shared memory**
  - Each thread block has own shared memory
  - Accessible only by threads within that block
  - Data lifetime = block lifetime

- **Global (device) memory**
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation
Memory Model Review

Thread

Per-thread Local Storage

Block

Per-block Shared Memory
Memory Model Review

Kernel 0

Kernel 1

Per-device Global Memory

Sequential Kernels
Memory Model Review

Host memory

`cudaMemcpy()`

Device 0 memory

Device 1 memory