22-24 CUDA

Lecture 22 CUDA - I

CUDA

Compute Unified Device Architecture

Integrated host+device app C program

- Serial or modestly parallel parts in host C code (host — CPU)
- Highly parallel parts in device SPMD kernel C code (device — GPU)

Parallel Threads

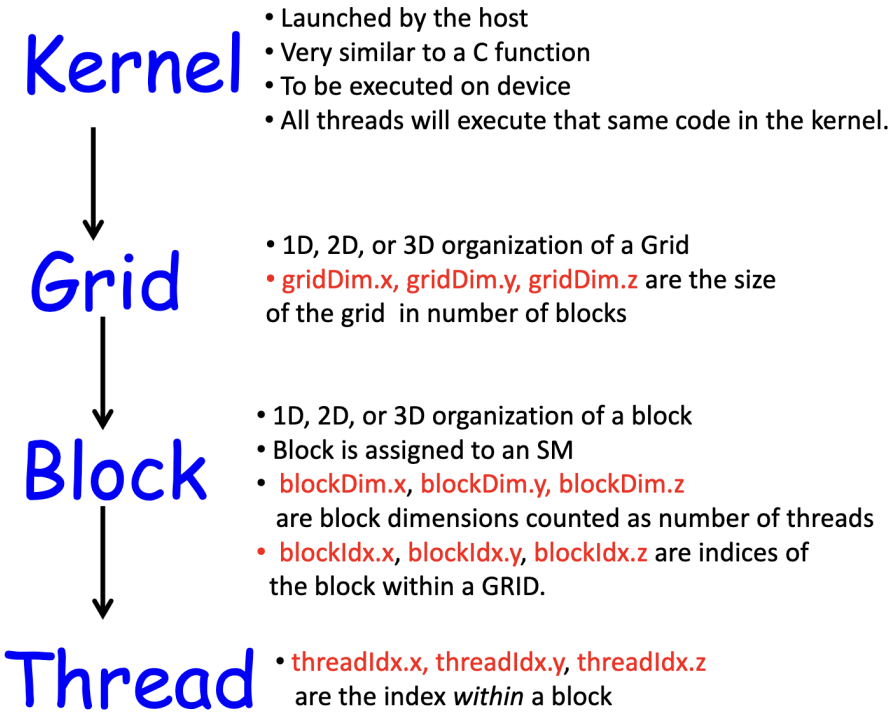
A CUDA kernel is executed by an array of threads

- All threads run the same code (the SP in SPMD)
- Each thread has an ID that it uses to compute memory addresses and make control decisions

Blocks

Divide monolithic thread array into multiple blocks

- Threads within a block cooperate via shared memory, atomic operations and barrier synchronization, ...
- Threads in different blocks cannot cooperate



Decisions you have to make as a GPU programmer:

1. Which part(s) of the program will be executed on the GPU?
2. How many total threads will you spawn?
3. How many blocks? That is: how many threads per block?
4. What will be the geometry of the block (1D, 2D, or 3D)?
5. What will be the geometry of the grid (1D, 2D, or 3D)?

CUDA Memory Model

Global memory

- Main means of communicating R/W Data between host and device
- Contents visible to all threads
- Long latency access

Shared memory:

- Per SM
- Shared by all threads in a block

In CUDA, host and devices have separate memory spaces. But in recent GPUs we have Unified Memory Access

If GPU and CPU are on the same chip, then they share memory space → fusion

CUDA Device Memory Allocation

cudaMalloc()

- Allocates object in the device Global Memory
- Requires two parameters
  - Address of a pointer to the allocated object
  - Size of allocated object

cudaFree()

- Frees object from device Global Memory
  - Pointer to freed object

cudaMemcpy()

- memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied

- Type of transfer (Host to Host, Host to Device, Device to Host, Device to Device)

Examples:

WIDTH = 64;
float* Md
int size = WIDTH * WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);

cudaFree(Md);

cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

Note About Error Handling

Almost all API calls return success or failure.

The type of the outcome is: cudaError_t

Success → cudaSuccess

Translate the error code to an error message: char * cudaGetErrorString (cudaError_t error)

A Simple Example: Vector Addition

```
void vecAdd(float* A, float* B, float* C, int n)
{
  int size = n * sizeof(float);
   float* A_d, *  B_d, * C_d;

1. // Transfer A and B to device memory
   cudaMalloc((void **) &A_d, size);
   cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &B_d, size);
   cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    // Allocate device memory for
    cudaMalloc((void **) &C_d, size);

2.  // Kernel invocation code – to be shown later
   …
3.   // Transfer C from device to host
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
     // Free device memory for A, B, C
    cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

How to launch a kernel?

```
int vecAdd(float* A, float* B, float* C, int n)
{
 // A_d, B_d, C_d allocations and copies omitted
 // Run ceil(n/256) blocks of 256 threads each
  vecAddKernel<<<ceil(n/256),256>>>(A_d, B_d, C_d, n);
}
```

#blocks #threads/blks

```
// Each thread performs one pair-wise addition
__global__
void vecAddkernel(float* A_d, float* B_d, float* C_d, int n)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

Unique ID

1D grid of 1D blocks: blockIdx.x *blockDim.x + threadIdx.x

1D grid of 2D blocks: blockIdx.x * blockDim.x * blockDim.y + threadIdx.y * blockDim.x + threadIdx.x

Kernels

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

__global__ defines a kernel function. Must return void

__device__ and __host__ can be used together

For functions executed on the device:

- No static variable declarations inside the function
- No indirect function calls through pointers

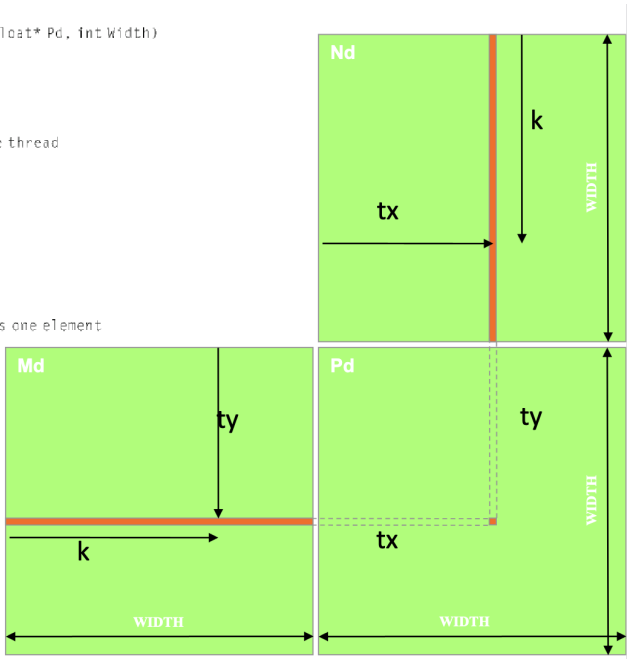
Matrix Multiplication

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;

  for (int k = 0; k < Width; ++k)
  {
     float Mdelement = Md[ty * Width + k];
     float Ndelement = Nd[k * Width + tx];
     Pvalue += Mdelement * Ndelement;
  }

  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
```



The Kernel Function

More On Specifying Dimensions

// Setup the execution configuration

dim3 dimGrid(x, y, z);

dim3 dimBlock(x, y, z);

// Launch the device computation threads!

MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);

Important:

dimGrid and dimBlock are user defined

gridDim and blockDim are built-in predefined variable accessible in kernel functions

Compiler: NVCC

nvcc -o prog prog.cu

Conclusions

Data parallelism is the main source of scalability for parallel programs。

Each CUDA source file can have a mixture of both host and device code.

What we learned today about CUDA:

- KernelA<<< nBlk, nTid >>>(args)
- cudaMalloc()
- cudaFree()
- cudaMemcpy()

- gridDim and blockDim
- threadIdx and blockIdx
- dim3

Lecture 23 CUDA - II

Software ↔ Hardware

From a programmer's perspective:

- Kernel
- Grid
- Blocks
- Threads

Hardware Implementation:

- SMs
- SPs (per SM)
- Warps

Some Restrictions First

All threads in a grid execute the same kernel code.

A grid is organized as a 1D, 2D, or 3D array of blocks (gridDim.x, gridDim.y, and gridDim.z)

Each block is organized as 1D, 2D, or3D array of threads (blockDim.x, blockDim.y, and blockDim.z)

Once a kernel is launched, its dimensions cannot change.

All blocks in a grid have the same dimension.

The total size of a block, in terms of number of threads, has an upper bound.

Once assigned to an SM, the block must execute in its entirety by the SM.

Compute Capability

It is a number in the form of x.y

A standard way to expose hardware resources to applications.

CUDA compute capability starts with 1.0 and latest one is 9.x (as of today)

API: cudaGetDeviceProperties()

Revisiting Matrix Multiplication

Break-up Pd into tiles

Each block calculates one tile

- Each thread calculates one element

- Block size equals tile size

```
// Setup the execution configuration
  dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
  dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);


__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // Calculate the row index of the Pd element and M
 int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
  // Calculate the column idenx of Pd and N
 int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

 float Pvalue = 0;
 // each thread computes one element of the block sub-matrix
 for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

 Pd[Row*Width+Col] = Pvalue;
}
```

Synchronization

__syncthreads()

called by a kernel function

The thread that makes the call will be held at the calling location until every thread in the block reaches the location

Beware of if-then-else

Threads in different blocks cannot synchronize → CUDA runtime system can execute blocks in any order

transparent scalability: the ability to execute the same application code on hardware with different number of execution resources

Scheduling of Blocks

To be assigned to an SM, a block needs to have all its resources (registers, shared memory, number of threads, ...) assigned beforehand.

CUDA runtime automatically reduces number of blocks assigned to each SM until resource usage is under limit.

What Is a Resource?

Characteristics of a resource:

- Must be inside the SM

- Must be determined before kernel launch

Example of SM resources

- number of threads that can be simultaneously tracked and scheduled.
- Registers
- Shared memory

Warps

Once a block is assigned to an SM, it is divided into units called warps.

- Thread IDs within a warp are consecutive and increasing

Warp is unit of thread scheduling in SMs

Partitioning is always the same.

We cannot determine which warp finishes first.

Each warp is executed in a SIMD fashion (i.e. all threads within a warp must execute the same instruction at any given time).

- Problem: branch divergence, occurs when threads inside warps branches to different execution paths.

Latency Tolerance

When an instruction executed by the threads in a warp must wait for the result of a previously initiated long-latency operation, the warp is not selected for execution → latency hiding

Scheduling does not introduce idle time → zero-overhead thread scheduling

Scheduling is used for tolerating long-latency operations.

Conclusion

We must be aware of the restrictions imposed by hardware:

- threads/SM
- blocks/SM
- threads/blocks
- threads/warps

The only safe way to synchronize threads in different blocks is to terminate the kernel and start a new kernel for the activities after the synchronization point.