Lecture 12

Beyond OpenMP & MPI: GPU parallelization

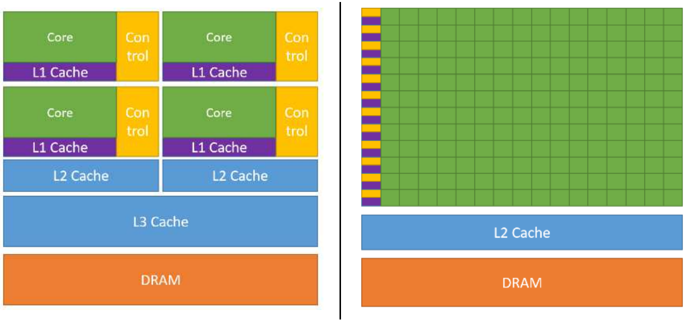
Introduction, Architecture, Programming

IKC-MH.57 *Introduction to High Performance and Parallel Computing* at January 05, 2024

Dr. Cem Özdoğan
Engineering Sciences Department
İzmir Kâtip Çelebi University

Contents

Exploring the GPU Architecture I



- CPUs are **latency oriented** (minimize execution of serial code).
- If the CPU has n cores, each core processes 1/n elements.
- Launching, scheduling threads adds overhead.

- GPUs are **throughput oriented** (maximize number of floating point operations).
- GPUs process one element per thread.
- Scheduled by GPU hardware, not by OS.

Exploring the GPU Architecture II

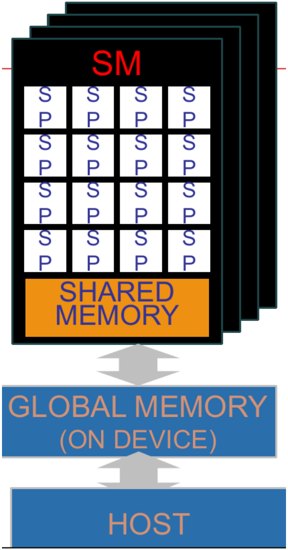
- A Graphics Processor Unit (GPU) is mostly known for the hardware device used when running applications that weigh heavy on graphics.
- Highly parallel, highly multithreaded multiprocessor optimized for graphic computing and other applications.

1. GPU Programming API: **CUDA (Compute Unified Device Architecture)** : parallel GPU programming API created by NVIDA
    - NVIDIA GPUs can be programmed by CUDA, extension of C language
    - API libaries with C/C++/Fortran language
    - CUDA C is compiled with nvcc
    - Numerical libraries: cuBLAS, cuFFT, Magma, ...
2. GPU Programming API: **OpenGL** - an open standard for GPU programming.
3. GPU Programming API: **DirectX** - a series of Microsoft multimedia programming interfaces.

- https://developer.nvidia.com/ Download: CUDA Toolkit, NVIDIA HPC SDK (Software Development Kit)

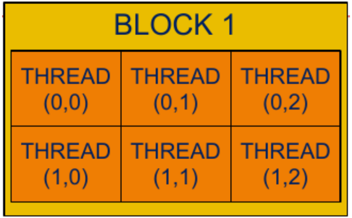
Exploring the GPU Architecture II

- **SP**: Scalar Processor 'CUDA core'. **Executes one thread.**
- **SM**: Streaming Multiprocessor 32xSP (or 16, 48 or more).
- Fast local 'shared memory' (shared between SPs) 16 KiB (or 64 KiB)
- For example: NVIDIA Maxwell GeForce GTX 750 Ti.
  - 32 SP, 20 SM : 640 CUDA Cores
- **Parallelization**: Decomposition to threads.
- **Memory**: Shared memory, global memory.
- **Thread communication**: Synchronization

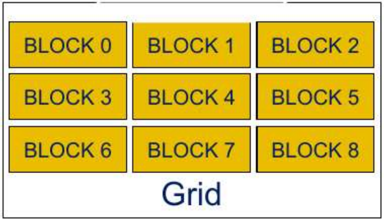


Exploring the GPU Architecture III

- Threads grouped in thread blocks: 128, 192 or 256 threads in a block
- One thread block executes on one SM.
  - All threads sharing the 'shared memory'.
  - Each thread block is divided in scheduled units known as a *warp*.

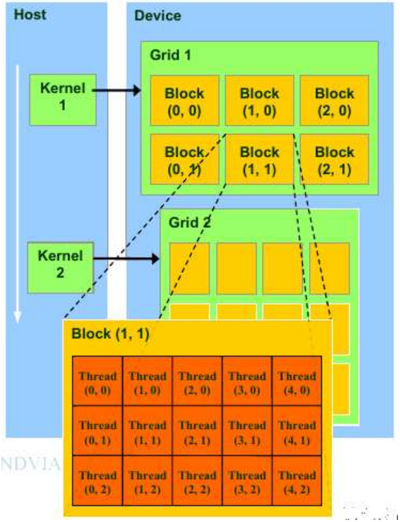


- Blocks form a GRID.
- **Thread ID**: unique within block.
- **Block ID**: unique within grid.



Exploring the GPU Architecture IV

- A kernel is executed as a grid of thread blocks. All threads share data memory space.
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution.
  - Efficiently sharing data through a low latency shared memory.
- Two threads from two different blocks cannot cooperate.



Execution and Programming Models I

- Computation partitioning (where to run)
  - Declarations on functions

```
__host__, __global__, __device__
```

```
__global__ void cuda_hello(){
}
```

  - Mapping of thread programs to device:

```
compute <<<gs,bs>>>(<args>)
```

```
cuda_hello <<<blocks_per_grid,threads_per_block>>> ();
```

- Data partitioning (where does data reside, who may access it and how?)
  - Declarations on data

```
__shared__, __device__, __constant__, ...
```

```
__device__ const char *STR = "HELLO WORLD!";
```

- Data management and orchestration
  - Copying to/from host: e.g.,

```
cudaMemcpy(h_obj,d_obj, cudaMemcpyDevicetoHost)
```

```
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
```

Execution and Programming Models II

- Concurrency management. e.g..

  `__synchthreads()`

  ```
  cudaDeviceSynchronize();
  ```

Kernel

- a simple C function
- executes on GPU in parallel as many times as there are threads
- The keyword

  `__global__`

  tells the compiler nvcc to make a function a kernel (and compile/run it for the GPU, instead of the CPU)
- It's the functions that you may call from the host side using CUDA kernel call semantics (<<< ... >>>).

Execution and Programming Models III

Setup and data transfer

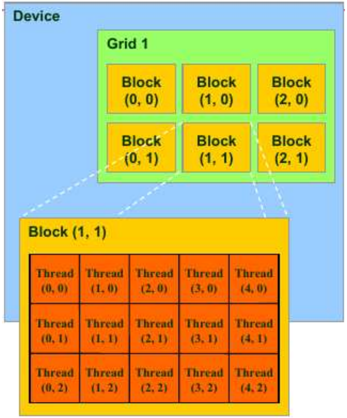
- *cudaMemcpy* : Transfer data to and from GPU (global memory)
- *cudaMalloc* : Allocate memory on GPU (global memory)

```
double *h_a; // Host input vectors
double *d_a; // Device input vectors
h_a = (double*)malloc(bytes); // Allocate memory for each vector on host
cudaMalloc(&d_a, bytes); // Allocate memory for each vector on GPU
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice); // Copy data from host array h_a to device arrays d_a
add_vectors<<<blk_in_grid, thr_per_blk>>>(d_a, d_b, d_c); // Execute the kernel
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost ); // Copy data from device array d_c to host array h_c
```

- GPU is the 'device', CPU is the 'host'. They do not share memory!
- The HOST launches a kernel that execute on the DEVICE.

Execution and Programming Models IV

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - **Block ID**: 1D or 2D (blockIdx.x, blockIdx.y)
  - **Thread ID**: 1D, 2D, or 3D (threadIdx.x,y,z)
- Simplifies memory addressing when processing multi-dimensional data.



Courtesy: NDVIA

- Compiler nvcc takes as input a .cu program and produces
  - C Code for host processor (CPU), compiled by native C compiler
  - Code for device processor (GPU), compiled by nvcc compiler

Execution and Programming Models V - Hello World I

Cuda Code:

```
#include <stdio.h>
#include <unistd.h>
__device__ const char *STR = "HELLO WORLD!";
const int STR_LENGTH = 12;
__global__ void cuda_hello(){
// blockIdx.x: Block index within the grid in x-direction
// threadIdx.x: Thread index within the block
// blockDim.x: # of threads in a block
    printf("Hello World from GPU! (%d ,%d) : %c ThreadID %d \n", blockIdx.x, threadIdx.x, STR[threadIdx.x % STR_LENGTH], (threadIdx.x +blockIdx.x*blockDim.x));
}
int main() {
    printf("Hello World from CPU!\n");
    sleep(2);
    int threads_per_block=12;
    int blocks_per_grid=2;
    cuda_hello <<<blocks_per_grid,threads_per_block>>> ();
    cudaDeviceSynchronize(); /* Halt host thread execution on CPU until the device has finished processing all previously requested tasks */
    return 0;
}
```

Execution and Programming Models VI - Hello World I

```
Hello World from CPU!
Hello World from GPU! (1 ,0) : H ThreadID 12
Hello World from GPU! (1 ,1) : E ThreadID 13
Hello World from GPU! (1 ,2) : L ThreadID 14
Hello World from GPU! (1 ,3) : L ThreadID 15
Hello World from GPU! (1 ,4) : O ThreadID 16
Hello World from GPU! (1 ,5) :   ThreadID 17
Hello World from GPU! (1 ,6) : W ThreadID 18
Hello World from GPU! (1 ,7) : O ThreadID 19
Hello World from GPU! (1 ,8) : R ThreadID 20
Hello World from GPU! (1 ,9) : L ThreadID 21
Hello World from GPU! (1 ,10) : D ThreadID 22
Hello World from GPU! (1 ,11) : ! ThreadID 23
Hello World from GPU! (0 ,0) : H ThreadID 0
Hello World from GPU! (0 ,1) : E ThreadID 1
Hello World from GPU! (0 ,2) : L ThreadID 2
Hello World from GPU! (0 ,3) : L ThreadID 3
Hello World from GPU! (0 ,4) : O ThreadID 4
Hello World from GPU! (0 ,5) :   ThreadID 5
Hello World from GPU! (0 ,6) : W ThreadID 6
Hello World from GPU! (0 ,7) : O ThreadID 7
Hello World from GPU! (0 ,8) : R ThreadID 8
Hello World from GPU! (0 ,9) : L ThreadID 9
Hello World from GPU! (0 ,10) : D ThreadID 10
Hello World from GPU! (0 ,11) : ! ThreadID 11
```

Execution and Programming Models VII - Hello World II

Cuda Code:

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#define N 720 // number of computations
#define GRID_D1 20 // constants for grid and block sizes
#define GRID_D2 3  // constants for grid and block sizes
#define BLOCK_D1 12 // constants for grid and block sizes
#define BLOCK_D2 1 // constants for grid and block sizes
#define BLOCK_D3 1 // constants for grid and block sizes

__global__ void hello(void) // this is the kernel function called for each thread
{
// CUDA variables {threadIdx, blockIdx, blockDim, gridDim} to determine a unique thread ID
    int myblock = blockIdx.x + blockIdx.y * gridDim.x; // id of the block
    int blocksize = blockDim.x * blockDim.y * blockDim.z; // size of each block
    int subthread = threadIdx.z*(blockDim.x * blockDim.y) + threadIdx.y*blockDim.x + threadIdx.x; // id of thread in a given block
    int idx = myblock * blocksize + subthread; // assign overall id/index of the thread
    int nthreads=blocksize*gridDim.x*gridDim.y; // Total # of threads
    int chunk=20; // Vary this value to see the changes at the output
    if(idx < chunk || idx > nthreads-chunk) { // only print first and last chunks of threads
        if (idx < N){
            printf("Hello world! My block index is (%d,%d) [Grid dims=(%d,%d)], 3D-thread index within block=(%d,%d,%d) => thread index=%d \n", blockIdx.x, blockIdx.y, gridDim.x, gridDim.y, threadIdx.x, threadIdx.y, threadIdx.z, idx);
        }
        else
        {
            printf("Hello world! My block index is (%d,%d) [Grid dims=(%d,%d)], 3D-thread index within block=(%d,%d,%d) => thread index=%d [### this thread would not be used for N=%d ###]\n", blockIdx.x, blockIdx.y, gridDim.x, gridDim.y, threadIdx.x, threadIdx.y, threadIdx.z, idx, N);
        }
    }
}
```

Execution and Programming Models VIII - Hello World II

```
int main(int argc,char **argv)
{
    // objects containing the block and grid info
    const dim3 blockSize(BLOCK_D1, BLOCK_D2, BLOCK_D3);
    const dim3 gridSize(GRID_D1, GRID_D2, 1);
    int nthreads = BLOCK_D1*BLOCK_D2*BLOCK_D3*GRID_D1*GRID_D2; // Total # of threads
    if (nthreads < N){
        printf("\n============ NOT ENOUGH THREADS TO COVER N=%d ===============\n\n",N);
    }
    else
    {
        printf("Launching %d threads (N=%d)\n",nthreads,N);
    }
    hello<<<gridSize, blockSize>>>(); // launch the kernel on the specified grid of thread blocks
    cudaError_t cudaerr = cudaDeviceSynchronize(); // Need to flush prints, otherwise none of the prints from within the kernel will show up as program exit does not flush the print buffer
    if (cudaerr){
        printf("kernel launch failed with error \"%s\".\n",
                cudaGetErrorString(cudaerr));
    }
    else
    {
        printf("kernel launch success!\n");
    }
    printf("That's all!\n");
    return 0;
}
```

Execution and Programming Models IX - Hello World II

```
Launching 720 threads (N=720)
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(0,0,0) => thread index=12
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(1,0,0) => thread index=13
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(2,0,0) => thread index=14
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(3,0,0) => thread index=15
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(4,0,0) => thread index=16
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(5,0,0) => thread index=17
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(6,0,0) => thread index=18
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(7,0,0) => thread index=19
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(5,0,0) => thread index=701
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(6,0,0) => thread index=702
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(7,0,0) => thread index=703
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(8,0,0) => thread index=704
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(9,0,0) => thread index=705
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(10,0,0) => thread index=706
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(11,0,0) => thread index=707
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(0,0,0) => thread index=708
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(1,0,0) => thread index=709
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(2,0,0) => thread index=710
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(3,0,0) => thread index=711
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(4,0,0) => thread index=712
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(5,0,0) => thread index=713
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(6,0,0) => thread index=714
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(7,0,0) => thread index=715
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(8,0,0) => thread index=716
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(9,0,0) => thread index=717
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(10,0,0) => thread index=718
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(11,0,0) => thread index=719
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(0,0,0) => thread index=0
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(1,0,0) => thread index=1
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(2,0,0) => thread index=2
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(3,0,0) => thread index=3
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(4,0,0) => thread index=4
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(5,0,0) => thread index=5
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(6,0,0) => thread index=6
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(7,0,0) => thread index=7
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(8,0,0) => thread index=8
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(9,0,0) => thread index=9
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(10,0,0) => thread index=10
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(11,0,0) => thread index=11
kernel launch success!
That's all!
```