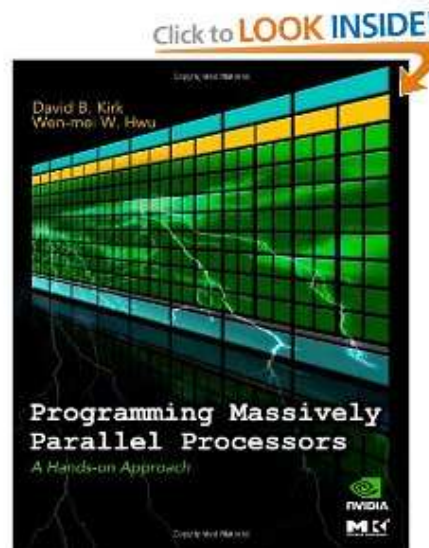


Lecture 15: Introduction to GPU programming

Overview

- GPUs & computing
- Principles of CUDA programming
- A good reference:
David B. Kirk and Wen-mei W. Hwu, **Programming Massively Parallel Processors**, Morgan Kaufmann Publishers, 2010.



New trends of microprocessors

- Since 2003, there has been two main trajectories for microprocessor design
- *Multicore* – a relatively small number of cores per chip, each core is a full-flesh processor in the “traditional sense”
- *Many-core* – a large number of much smaller and simpler cores
 - NVIDIA GeForce GTX 280 GPU (graphics processing unit) has 240 cores, each is heavily multi-threaded, in-order, single-instruction issue processor. Eight cores share control and instruction cache.
 - As of 2009 peak performance of many-core GPUs is at around 10 fold the peak performance of multicore CPUs
 - GPUs have larger memory bandwidth (simpler memory models and fewer legacy requirements)

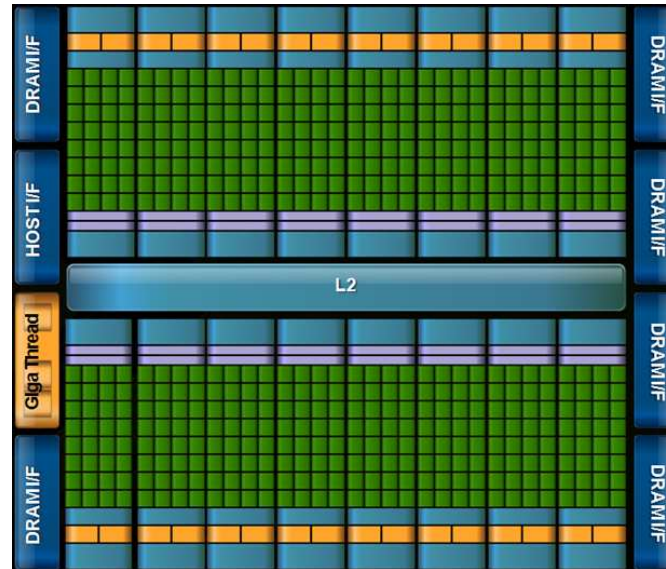
Early use of GPU for computing

- Design of GPU is shaped by video game industry, ability to perform a massive number of floating-point (single precision) calculations per video frame
- Full 3D pipeline: transform, lighting, rasterization, texturing, depth testing and display
- Computing on the earlier GPU architectures had to cast computing as graphics operations
 - GeForce 3 in 2001 – programmable pixel shading
 - Later GeForce products – separate programmable engines for vertex and geometry shading

GPGPU

- General-purpose GPU – capable of performing non-graphics processing
 - running shader code against data presented as vertex or texture information
 - computing results retrieved at later stage in the pipeline
 - still “awkward programming” compared with CPU
- “Unified shader architecture” – each shader core can be assigned with any shader task, no need for stage-by-stage balancing
 - GeForce 8800 in 2006 (128 processing elements distributed among 8 shader cores)
 - Tesla product line – graphics cards without display outputs and drivers optimized for GPU computing instead of 3D rendering

NVIDIA's Fermi architecture



- Designed for GPU computing (graphics-specific bits largely omitted)
- 16 streaming multiprocessors (SMs)
- 32 CUDA cores (streaming processors) in each SM (512 cores in total)
- Each streaming processor is massively threaded
- 6 DRAM 64-bit memory interfaces
- GigaThread scheduler
- Peak double-precision floating-point rate: 768 GFLOPs

Massive (but simple) parallelism

- One streaming processor is the most fundamental execution resource
- Simpler than a CPU core
- But capable of executing a large number of threads simultaneously, where the threads carry out same instruction to different data elements — single-instruction-multiple-data (SIMD)
- A number of streaming processors constitute one streaming multiprocessor (SM)
- On Nvidia's GT200, 1024 threads per SM, up to about 30,000 threads simultaneously

CUDA

- CUDA – Compute Unified Device Architecture
- C-based programming model for GPUs
- Introduced together with GeForce 8800
- Joint CPU/GPU execution (*host/device*)
- A CUDA program consists of one or more phases that are executed on either host or device
- User needs to manage data transfer between CPU and GPU
- A CUDA program is a unified source code encompassing both host and device code

More about CUDA

- To a CUDA programmer, the computing system consists of a host (CPU) and one or more devices (GPUs)
- Data must be explicitly copied from host to device (and back)
- On device, there is so-called *global memory*
- Device global memory tends to have long access latencies and finite bandwidth
- On-chip memories: registers and shared memory (per SM block) have limited capacity, but are much faster
- Registers are private for individual threads
- All threads with a thread block can access variables in the shared memory
- Shared memory is an efficient means for threads to cooperate, by sharing input data and intermediate results

Programming GPUs for computing (1)

- *Kernel functions* – the computational tasks on GPU
 - An application or library function may consist of one or more kernels
 - Kernels can be written in C, extended with additional keywords to express parallelism
- Once compiled, kernels consist of many threads that **execute the same program in parallel**
- Multiple threads are grouped into thread blocks
 - All threads in a thread block run on a single SM
 - Within a thread block, threads cooperate and share memory
 - A thread block is divided into warps of 32 threads
 - Warp is the fundamental unit of dispatch within an SM
 - Threads blocks may execute in any order

Programming GPUs for computing (2)

- When a kernel is invoked on host, a *grid* of parallel threads are generated on device
- Threads in a grid are organized in a two-level hierarchy
 - each grid consists of one or more thread blocks
 - all blocks in a grid have the same number of threads
 - each block has a unique 2D coordinates `blockIdx.x` and `blockIdx.y`
 - each thread block is organized as a 3D array of (up to 512) threads `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
- The grid and thread block dimensions are set when a kernel is invoked
- A centralized scheduler

Programming GPUs for computing (3)

- Syntax for invoking a kernel

```
dim3 dimGrid(64,32,1)
dim3 dimBlock(4,2,2);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

- `gridDim` and `blockDim` contain the dimension info
- All threads in a block share the same `blockIdx`
- Each thread has its unique `threadIdx` within a block
- One common usage of `blockIdx` and `threadIdx` is to determine which data element(s) that a thread is to work on
- Possible to let one thread to compute one data element (fine-grain parallelism)

Thread execution

- Launching a CUDA kernel will generate a 1D or 2D array of thread blocks, each having a 1D or 2D or 3D array of threads
- The thread blocks can execute in any order relative to each other
- CUDA runtime system bundles several threads for simultaneous execution, by partitioning each thread block into *warps* (32 threads)
- Thread indices determine warp partitioning
- Scheduling of warps is taken care by CUDA runtime system
- The hardware executes same instruction for all threads in the same warp
- If-tests can cause thread divergence, which will require multiple passes of divergent paths (involving all threads of a warp)

Simple example of CUDA program

The device is used to square each element of an array

```
// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
    // 1D thread blocks and 1D thread array inside each block
    // N is the length of the data array a, stored in device memory

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) a[idx] = a[idx] * a[idx];
}
```

Simple example of CUDA program (cont'd)

```
// main routine that executes on the host
int main(void)
{
    float *a_h, *a_d; // Pointer to host & device arrays
    const int N = 10; // Number of elements in arrays
    size_t size = N * sizeof(float);
    a_h = (float *)malloc(size); // Allocate array on host
    cudaMalloc((void **) &a_d, size); // Allocate array on device
    // Initialize host array and copy it to CUDA device
    for (int i=0; i<N; i++) a_h[i] = (float)i;
    cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
    // Do calculation on device:
    int block_size = 4;
    int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
    square_array <<< n_blocks, block_size >>> (a_d, N);
    // Retrieve result from device and store it in host array
    cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    // Print results
    for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
    // Cleanup
    free(a_h); cudaFree(a_d);
}
```

Matrix multiplication, example 1

- We want to compute $Q = M \times N$, assuming Q, M, N are all square matrices of same size `Width×Width`
- Each matrix has a 1D contiguous data storage
- Naive kernel implementation

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Qd,
                                int Width)
{
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;
    float Qvalue = 0;
    for (int k=0; k<Width; ++k)
        Qvalue += Md[Row*Width+k] * Nd[k*Width+Col];
    Qd[Row*Width+Col] = Qvalue;
}
```


Matrix multiplication, example 2

- The previous implementation is not memory efficient
- Each thread reads $2 \times \text{width}$ values from global memory
- A better approach is to let a patch of $\text{TILE_WIDTH} \times \text{TILE_WIDTH}$ threads share the $2 \times \text{TILE_WIDTH} \times \text{width}$ data reads
- That is, each thread reads $2 \times \text{width} / \text{TILE_WIDTH}$ values from global memory
- Shared memory is important to use!
- Also beware that size of shared memory is limited

Matrix multiplication, example 2 (cont'd)

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Qd,
                                int Width)
{
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Qvalue = 0;
    for (int m = 0; m<Width/TILE_WIDTH; ++m) {
        Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
        __syncthreads();

        for (int k=0; k<TILE_WIDTH; ++k)
            Qvalue += Mds[ty][k] * Nds[k][tx];
        __syncthreads();
    }

    Qd[Row*Width+Col] = Qvalue;
}
```

CPU computing vs. GPU computing

- CPUs are good for applications where most of the work is done by a small number of threads, where the threads have high data locality, a mixture of different operations and conditional branches
- GPU aims at the “other end of the spectrum”
 - data parallelism – many arithmetic operations performed on data structures in a simultaneous manner
 - applications with multiple threads that are dominated by longer sequences of computational instructions
 - computationally intensive (not control-flow intensive)
- GPU computing will not replace CPU computing