GPUs for Computing?

- GPUs traditionally handle rasterization on display
  - Evolve to manycore stream processors – over 100x cores
  - Now they are programmable – CUDA, OpenCL, ...

- To avoid the performance limitation from Amdahl’s law
  - Work on bigger problems (which improve the qualities of experience, simulation, ...)

- General computing on GPUs
  - Cannot help office productivity programs, but
  - Can accelerate data-parallel programs (supercomputers used to do this)
GPGPU: General Purpose Computing on GPUs

- GPUs as commodity massively-parallel manycore chips
  - NVIDIA Tesla C1060: 240 stream processors, 936 GFLOPS peak
  - GTX260: 192 stream processors, 777 GFLOPS peak
  - G80 (GeForce 8800 GTX): 128 stream processors, 518 GFLOPS peak
  - Single precision FP
    - Double precision FP per core on Fermi (540 cores)
  - Later architectures (Kepler, Maxwell, Pascal) all support DP FP

- CUDA: Compute Unified Device Architecture
  - General purpose SPMD programming model
  - User kicks off batches of threads on the GPU
  - GPU as a dedicated, super-threaded, massively data-parallel coprocessor

- Resource management critical to fully utilize GPUs
  - # of threads, registers, memory bandwidth, …
Microarchitecture (G80)

- **16 stream multiprocessors (SMs)**
  - Each SM is assigned several blocks of threads

- **8 stream processors (SPs) per SM**
  - Each SP executes single thread’s instructions in a SIMD fashion

- **Threads in a block → grouped into multiple warps of 32 parallel threads**
  - Scheduling unit used by SM
  - SM executes the same instruction for all threads in a warp (≈SIMD) – 4 cycles per warp
  - When a warp stalls, the SM quickly switches to a ready warp in the SM
    - Uses scoreboard to check if ready

![Diagram of G80 microarchitecture](image)

**Device**

- **SM 16**
- **SM 2**
- **SM 1**

- **Shared Memory (16KB)**
- **Register File (8192)**
- **Constant Cache (8KB)**
- **Texture Cache (16KB/2SM)**
- **Instruction Unit**
- **Global Memory (DRAM, 768MB)**

**Notes**

- **SP & SFU are fully pipelined, running @ 1.35GHz**
- **388.8GFLOPS = 16SM * 18FLOPS/SM * 1.35GHZ**
Fermi

- 3 billion transistors
- 512 CUDA cores
- ~ 2x the memory bandwidth
  - Dual DMA engines for CPU
- Unified address space
  - 40-bit AS for local, shared, global
- L1 and L2 caches
  - Between SPs and global memory
  - Configurable 64K L1$/PBSM
- 8x the FP64 performance
  - All SPs are capable of double operations
- CUDA extensions for C++
- Concurrent kernel execution
- Fast context switch
Number of Cores Increasing

- **Tesla 10 series architecture (C10xx)**
  - 240 thread processors (SPs)
  - 30 multiprocessors (SMs), each contains
    - 8 thread processors (SPs)
    - One double-precision unit
    - Shared memory enables thread cooperation
CUDA: Compute Unified Device Architecture

- CUDA program consists of multiple phases that are executed on either
  - CPU (host code) or
  - GPU (kernel code)

- Runs *many* threads in parallel (G80)
  - Interleaved scheduling
    - Invisible to programmers
  - Extremely lightweight threading
  - Needs 1,000s of threads for full utilization

- GPU: coprocessor to the CPU (or host)
  - Slow DRAM (device/global memory)
  - Fast shared memory per SM (16KB)
  - Latency to device memory should be overlapped with computation
Threading Model in CUDA

- **Kernel function**
  - Executed on the GPU, “Main function of a thread”
  - Single program multiple data (SPMD)

- **Threads are organized into three-level hierarchy**
  - *Grid, block, thread*
    - 1 grid, max $2^{32}$ blocks, max 512 threads/block

- **Communication between threads within a block**
  - Share data through *shared memory* (scratchpad)
  - Perform barrier synchronization
  - Otherwise independent
Extended C

- Declspecs
  - global, device, shared, local, constant

- Keywords
  - blockIdx, threadIdx
  - blockDim, gridDim

- Intrinsic
  - __syncthreads() : within a block

- Runtime API
  - Memory management, ...

- Function launch
  - fn<<< #blocks, #threads-per-block>>>
Thread Hierarchy

- Threads are partitioned into thread blocks
  - Grid – all thread blocks for a kernel launch
  - Thread block – a group of threads
    - Synchronize their execution
    - Communicate via shared memory

gridDim – up to 2D

gridDim

blockDim – up to 3D

Block (2, 1)

Thread (0,0)  Thread (1,0)  Thread (2,0)  Thread (3,0)
Thread (0,1)  Thread (1,1)  Thread (2,1)  Thread (3,1)
Thread (0,2)  Thread (1,2)  Thread (2,2)  Thread (3,2)
Thread (0,3)  Thread (1,3)  Thread (2,3)  Thread (3,3)
Memory Model in CUDA

- **block**
- **__shared__**
  - Per-block Shared Memory
  - on-chip (located within SM)

- **Kernel 0**
- **Kernel 1**

- **__device__**
  - Per-device Global Memory
  - off-chip (located outside SM)
Memory Model in CUDA (cont’d)

- **Thread**
  - **__local__**
    - Per-thread Local Memory
    - Off-chip (located outside SM)
  - **__constant__**
    - Device Memory
    - On-chip (located within SM)
    - Off-chip (located outside SM)

- **Texture Cache**
  - On-chip (located within thread processor)
  - BUT not available for CUDA
Memory Model in CUDA (cont’d)

CPU
- Host Memory (main memory)
  - cudaMemcpy()
  - cudaMemcpyAsync()
  - cudaMallocHost()

GPUs
- Device 0 Memory
- Device 1 Memory
  - cudaMalloc()
  - cudaFree()
  - cudaMemcpy()
  - cudaMemcpyAsync()
CUDA Example

Serial C code

```c
void serial_fn(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = a*x[i] + y[i];
}

// invoke serial function
serial_fn(n, 2.0, x, y);
```

CUDA Parallel C code

```c
void parallel_fn(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}

// invoke parallel function with 256 threads/block
int nblock = (n + 255) / 256;
parallel_fn<<<nblocks, 256>>>(n, 2.0, x, y);
```
Resources & Limitations

- **Registers (8192 regs)**
  - → partitioned among the threads running on a SM
  - E.g. 10 regs/thread * 256 threads/block * 3 blocks → 7,680 regs

- **Shared memory**
  - → partitioned among the blocks running on a SM

- **Max 768 threads, 8 blocks per SM**
- **# of blocks active in a SM**
  - → Limited by above factors

- **Implications**
  - Some optimizations may have negative effects in some cases
  - Easy to be “trapped” in a local maximum when hand-optimizing
CUDA Restrictions

- Threads are created only by launching a kernel
  - Task parallelism is restricted
  - Dynamic/irregular thread creation is restricted

- Thread blocks run to completion – no yield
  - Thread blocks can run in arbitrary order
  - No recursion

- No writable on-chip persistent state between thread blocks
  - Global communication is restricted
  - No static variables within kernels

- Separate address space between CPU and GPUs
  - Kernels can only access GPU memory

- Host has to manage OS services and multiple GPUs
  - No OS services (such as malloc) within kernels
  - Explicit management for multi-GPUs

- Compute and 3D-rendering can’t run concurrently
Optimization Principles

- Manage the partitioned resources
  - Registers – partitioned by the threads
  - Shared memory – partitioned by the blocks (*also avoid bank conflicts)

- Manage global memory latency
  - Create enough threads to keep SPs occupied
  - Resource limitations may constrain this*

- Manage the global memory BW
  - Use registers & shared memory to reuse data
  - Can limit the number of simultaneous threads

- Manage host memory latency & BW
  - Overlap computation with host memory accesses
Asynchronous Data Transfers

- Asynchronous host-device memory copy returns control immediately to CPU
  - cudaMemcpyAsync(dst, src, size, dir, stream);
  - Requires *pinned* host memory (allocated with cudaMallocHost)
- Overlap CPU computation with data transfer
  - Use default stream = 0

```c
cudaMemcpyAsync(dst, src, size, cudaMemcpyHostToDevice, 0);
cpuFunction();
cudaThreadSynchronize();
kernel<<<grid, block>>>(dst);
```
GPU/CPU Synchronization

- **Context based synch.**
  - `cudaThreadSynchronize()`
  - Blocks until all previously issued CUDA calls from a CPU thread complete

- **Stream based synch.**
  - `cudaStreamSynchronize(stream)`
  - Blocks until all CUDA calls issued to given stream complete
  - `cudaStreamQuery(stream)`
  - Indicate whether stream is idle (cudaSuccess, cudaErrorNotReady, …)
  - Does not block CPU thread
Overlapping Kernel with Data Transfer

- Concurrent copy and execute
  - `cudaDeviceProp.deviceOverlap` should be "1"
- Kernel and transfer use different, non-zero streams
  - CUDA calls to stream 0 blocks until all previous calls complete

```c
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, direction stream1);
kernel<<<grid, block, 0, stream2>>>(…);
cudaStreamSynchronize(stream1);
```
Shared Memory Architecture

- Shared memory is divided into banks
  - Allow many threads access the shared memory
  - Successive 32-bit words are assigned to successive banks
  - Each bank can serve one address per cycle
    - Serialize multiple requests
  - Multiple simultaneous accesses to a bank cause bank conflict
- Shared memory is almost as fast as registers, if no bank conflict
  - All threads of a half-warp access different banks – no bank conflict
  - All threads of a half-warp read the identical address – no bank conflict with broadcast
Bank Conflicts in Shared Memory

- e.g., 16 threads (half-warp) access linear addresses
  - stride = 1
  - no bank conflict

- stride = 2
  - 2-way bank conflict
Summary

- **GPGPU**
  - Manycore for general data parallel computations

- **CUDA / OpenCL**
  - Programming APIs for GPUs

- **Optimizations**
  - Warp (32 threads)
  - Shared memory vs. global memory
  - Overlapping computation with memory accesses
    - double buffering, many threads