Graphics Processing Units -- GPUs (Chapt 5, Matloff)

GPU - outcome of the video game market
  □ CPU processing too slow
  □ Added processing power on graphics cards
  □ Many graphics cards now allow non-graphics use of GPU
  □ GPU not general purpose hardware

NVIDIA -- One manufacturer, CUDA -- NVIDIA’s GPU programming system
OpenCl -- Another system that covers GPU programming
  □ Wants to cover all "parallel processing systems"
  □ May look more at this if time permits

Look at using CUDA for this class (S17)
GPU Architecture  (Very NVIDIA specific?)

GPU consists of:
- a set (large) of streaming multiprocessors (SMs)

Each SM consists of:
- a set (large) of streaming processors (SPs)

http://hothardware.com/articleimages/Item1447/gf100-diagram.png
NVIDIA fermi SP (From http://benchmarkreviews.com/.../NVIDIA_Fermi/...)

---

GPU Architecture (page 2)
Cores run threads
Threads in a single SM run in lock step
Threads in different SMs can not synchronize (eg barrier)
Word size is 32 bits. (newer devices can use double)

SIMT architecture -- Single Instruction, multiple threads (aka SIMD)
  group of threads is called a "Block"
  hardware divides a block into warps (32 threads/warp)
  all threads in a warp run the code in lockstep
  A thread either executes or does nothing
  if/then/else causes waiting of all threads

GPU manage warps in hardware
  warps are run in a timesharing fashion
  fixed length time slices given to warps
  access to global memory by a warp => warp not scheduled
  hardware context switches very fast

Typical app: Many threads, little work per thread (fine grain)
GPU Memory Structure

GPUs have several types of memory ... all separate from CPU’s memory

- **Shared Memory**
  - Each SM has a small shared memory
  - SP’s have access to shared memory
  - Memory is very fast to access
  - Host has no access to shared memory
  - Used for communication between threads in the same warp
  - Managed by programmer for speed

- **Global Memory**
  - Shared by all threads in the entire GPU
  - Persistent across "code runs (called kernels)"
  - Host has access to global memory
    - copy in/ copy out operations common
    - transfer issues need special attention
  - Access from a thread is "slow" (latency)
    - Bandwidth can be high by "coalescing"
  - Access to memory (global and shared) is done in "half-warps"
Other Memory in a GPU

- Register: "used by the compiler"
  - non-constant indexed arrays not allocated to registers
- Local memory: Part of Global, but for a specific thread
  - register spills go here
- Constant memory: readonly from device (64K)
  - host has read/write access
- Texture: similar to constant, caching is 2D

Threads Hierarchy

- grid: all threads for an application
- grid consists of one or more "blocks"
  - each block has an address: (x,y)
  - each thread in a block has an address (z)
  - all threads in a block run on the same SM
  - a SM may have more than one block
CUDA Code

Code that runs looks like C with extra attributes and "special syntax"
Code is comprised of the following
- Host code ... regular C or C++ programs (Looks like C++ always used)
- Device code, functions to run on the device (kernel)
- Host code to control the device, set up grids and blocks, copy memory ...

__global__ void functionname (args)
- Must be void functions
- "global" code called from host code
- No access to C library
- special versions of math functions, e.g. __sin()
- No recursion
- No stack ... e.g. "inline calls"
- No pointers to functions
- Local variables in "local memory"
- Can declare "shared memory variables" __shared__ int ....;
- Addresses of code ... blockIdx and threadIdx (more later)

__device__ type name (args) // device called from device
- May specify a return type, may not be recursive
Device memory management from host

- `cudaMalloc(void **mem, msize)` // size in bytes
- `cudaMemcpy(dest, src, msize, cudaMemcpyHostToDevice)`
  - or `cudaMemcpyDeviceToHost` or `cudaMemcpyDeviceToDevice`

Kernel call .... starting a `__global__` function (in a big group)

- `functionname<<<dimGrid, dimBlock>>>(args)`
  - args may be pointers to device global memory or values
  - total threads is `dimGrid * dimBlock` -- all started
    - `dim3()` object (C++), x,y,z element
    - Both grid (of blocks) and blocks (of threads) are grids
    - using integers => `dim3(n,1,1)`
  - NOTE: kernel code needs to check bounds!
    - `gridDim.x/y/z, blockIdx.x/y/z` -- where in the grid
    - `blockDim.x/y/z, threadIdx.x/y/z` -- where in the block
- no blocking is done at this point
  - `cudaThreadSynchronize();` // in the host, blocks
  - `err = cudaGetLastError();` // also appears to block
Synchronization, within and between blocks
- __syncthreads() works in a single block.
- Threads across blocks can not synchronize with each other ....
- But ... there are atomic operations that execute without pre-emption
  - atomicAdd(pointertovar,inc)
  - atomicExch(), atomicCAS(f,s,t) == (f==s? f=t : 0)
  - atomicMin(), atomicMax(), atomicAnd(), AtomicOr(), ...
- Could build interblock sync ... better to use cudaThreadSynchronize()

blocks and threads
- blocks are assigned to the same SM, run in warps
- blocks typically want more than 32 threads per block
  - Usually some multiple of 32 ...eg 128 or 256
- There is a limit to threads/block (512?) and threads per SM (786?)
- want as many blocks as there are SMs to make use of the machine
  - may have many more thus mapping multiple blocks to each SM
- more blocks per SM allow for "latency hiding"
- Total threads are determined at kernel call time.
Vendor supplied: /usr/local/cuda/samples

- Should be able to copy them and make them. (can’t make in place)
- Need to copy ALL since some directories include from other places
- Start looking at 0_Simple/vectorAdd
- Compiler: /usr/local/cuda/bin/nvcc
  - nvcc -g -G file.cu
    - May have other compiler flags
    - -g, -G debug options for host and device
    - There is a cuda-gdb program in the cuda/bin directory

Book’s vector example (modified by Phil)
- rowSum.cu, seqSum.c
  - I used ints for dimGrid and dimBlock (one dimensional)
  - I ran the kernel multiple times to try to get a time delay

Getting information ( info.cu )

    cudaDeviceProp Props;
    cudaMemcpyProperties(&Props,0);
__global__ void procpairs (int *m, int *tot, int n)
{
    int totth = gridDim.x * blockDim.x;
    int me = blockIdx.x * blockDim.x + threadIdx.x;
    int i, j, k, sum;
    for (i = me; i < n; i += totth)
        for (j = j_1; j < n; j++)
            for (k = 0; k < n; k++)
                sum += m[n*i+k] * m[n*j+k];
    atomicAdd(tot, sum);
}

□ Good use of atomicAdd ....
□ *tot must be device global memory

Book has several other examples .... including a form of parallel prefix

Errors: cudaGetLastError(), cudaGetErrorString(err)
Other Topics

Loop Unrolling

```c
for (i = 0; i < 2; i++) {
    sum += x[i];
    sum2 += x[i]*x[i];
}
sum += x[1];
sum2 += x[1]*x[1];
sum += x[2];
sum2 += x[2]*x[2];
```

- Unrolling allows storage of sum and sum2 in registers
- `#pragma unroll k` // k-fold unrolling
  - `k = 1` => no unrolling

Short Vectors -- up to 4 elements, names typeN (e.g. uint3) ...
- Can be treated as a single word in terms of memory access

Library code ....

- CUBLAS -- Cuda Basic Linear Algebra Subroutines
  - callable from C
  - Highly tuned, uses kernel calls
- CUFFT -- Cuda FFT ....