CS 470
Spring 2023

Mike Lam, Professor

GPU Programming
(mainly w/ CUDA)
A brief digression into gaming

- **1970s**: arcades began using specialized graphics chips
- **1980s**: increasingly sophisticated capabilities
  - E.g., sprites, blitters, and scrolling
- Early-mid **1990s**: first 3D consoles and 3D accelerators for PCs
  - E.g., Nintendo 64 and Voodoo graphics cards
- Late **1990s**: “classic” graphics wars begin
  - Nvidia vs. ATI and DirectX vs. OpenGL
- Early **2000s**: new "shaders" enable easier non-graphical use of accelerators
Bringing it back

- Late **2000s** and early **2010s**: rise of General-Purpose GPU (GPGPU) frameworks
  - 2007: Compute Unified Device Architecture (CUDA) released (newer library: Thrust)
  - 2009: OpenCL standard released
  - 2011: OpenACC standard released
  - 2013: OpenMP 4.0 standard added target directive
    - Enhanced w/ 4.5 standard in 2015

- Heterogeneous computing
  - Manycore CPUs and GPUs in the same system (hybrid clusters)
  - Field-Programmable Gate Arrays (FPGAs) for general/reconfigurable applications
  - Digital Signal Processors (DSPs) for specialized purposes
"Kernels" or "shaders" run on many logical threads grouped into blocks
- Blocks are assigned to a streaming multiprocessor (SM) w/ many individual cores
- Threads are run in warps w/ access to shared memory within the block
- Limited, low-power instruction set that operates primarily on vector data
- Must copy data back and forth between host and device memory
Terminology note

- **Single Instruction, Multiple Thread (SIMT)**
  - Differs from SIMD in that threads may not always execute simultaneously on the SM
  - Some threads may block for I/O while others execute
CUDA

- **CUDA**: NVIDIA’s GPU computation API for C++
  - Compile .cu source files with NVIDIA compiler (nvcc)
  - CUDA Programming Guide provided online

- Many-way parallelism
  - Write a **kernel** routine to be run on each thread (like Pthreads)
    - __global__ routines are called on host and executed on the device
  - Must manually split up work among threads (arranged in a **grid of blocks**)
    - Common approach: grid-stride loop
  - Call kernel: `kernel_func<<<numBlocks, blockSize>>>(())`
  - Kernels are **asynchronous** by default
    - Permits simultaneous computation on CPU and GPU
    - Call `cudaDeviceSynchronize()` to wait for a kernel to finish
Hello world in CUDA

```c
__global__
void hello()
{
    printf("Hello from thread %d in block %d\n", threadIdx.x, blockIdx.x);
}

int main(int argc, char* argv[])
{
    // parse command-line parameters
    int nblocks = strtol(argv[1], NULL, 10);
    int nthreads = strtol(argv[2], NULL, 10);

    // launch kernel on GPU
    hello<<<nblocks, nthreads/nblocks>>>();

    // wait for GPU to finish
    cudaDeviceSynchronize();

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

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

__global__ void saxpy_parallel(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 SAXPY kernel with 256 threads/block
int nbblocks = (n + 255) / 256;
saxpy_parallel<<<nbblocks, 256>>>(n, 2.0, x, y);
__global__ void square_array(float *a, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx<N) a[idx] = a[idx] * a[idx];
}

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 and cleanup
    for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
    free(a_h); cudaFree(a_d);
}
CUDA

- Device runs many threads in blocks
  - Each block is scheduled to a streaming multiprocessor (SM)
    - An SM might be responsible for multiple blocks
  - Block size should be a multiple of the warp size
    - (probably the maximum allowed)
  - Number of blocks should be related to number of SMs
    - Could also be a function of the total data size divided by the block size

Quadro P1000 (in EnGeo 2204 lab machines)

- Maximum global memory: 4G
- Maximum shared memory per block: 48K
- Maximum block size: 1024
- Warp size: 32
- Number of cores: 640 (5 SMs, 128 CUDA cores/SM)
CUDA

- **Grid-stride access** in kernel loops generalizes to any data size
  - Threads skip `numBlocks * blockSize` each iteration
  - Essentially performs a cyclic data “distribution”
CUDA example (serial version)

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

int main(void) {
    int N = 1<<20;

    float *x, *y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // run add routine
    add(N, x, y);

    // check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++) {
        maxError = fmax(maxError, fabs(y[i]-3.0f));
    }
    printf("Max error: \%f\n", maxError);

    // free memory
    free(x);
    free(y);
    return 0;
}
```

From https://devblogs.nvidia.com/even-easier-introduction-cuda/
__global__
void add(int n, float *x, float *y) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride) {
        y[i] = x[i] + y[i];
    }
}

int main(void) {
    int N = 1<<20;

    // unified memory – accessible from CPU or GPU
    float *x, *y;
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // run kernel on the GPU
    int blockSize = 256;
    int blockCount = (N+blockSize-1) / blockSize;
    add<<<blockCount, blockSize>>>(N, x, y);

    // wait for GPU to finish
    cudaDeviceSynchronize();

    // check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++) {
        maxError = fmax(maxError, fabs(y[i]-3.0f));
    }
    printf("Max error: %f\n", maxError);

    // free memory
    cudaFree(x);
    cudaFree(y);

    return 0;
}

From https://devblogs.nvidia.com/even-easier-introduction-cuda/
• Traditional (manual) model: host vs. device memory
  – Local variables marked with annotations
    • __device__ variables in GPU global memory, accessible by all threads
    • __shared__ variables in GPU shared memory, accessible by threads in the same block
  – cudaMalloc to allocate large regions of device memory
    • cudaMemcpy to copy memory to or from the device
      – “kind” parameter: cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost
    • cudaFree to deallocate device memory

• Newer (automatic) model: unified memory
  – Movement handled by CUDA
  – Call cudaMemcpyManaged() to allocate unified memory
  – __managed__ variables accessible on both host and device
A warp is a set of CUDA threads with consecutive ranks
- Fixed size (32 at the moment)
  - Index of a thread inside a warp is called its lane
- In general, warps behave in a SIMD fashion
- If the control paths diverge, performance will suffer
  - (E.g., threads take different branches of an if/else)

CUDA provides some atomic operations
- E.g., atomicAdd() or atomicMax()
- Full list in CUDA programming guide

“Fast barrier” in CUDA: __syncthreads()
- Causes all threads in a block to sync up
#pragma acc data copy(A) create(Anew)
while (error > tol && iter < iter_max) {
    error = 0.0;

    #pragma acc kernels
    {
        #pragma acc loop
        for (int j = 1; j < n-1; j++) {
            for (int i = 1; i < m-1; i++) {
                error = fmax(error, fabs(Anew[j][i] - A[j][i]));
            }
        }

        #pragma acc loop
        for (int j = 1; j < n-1; j++) {
            for (int = i; i < m-1; i++ ) {
                A[j][i] = Anew[j][i];
            }
        }
    }

    if (iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
    iter++;
}