Introduction to GPGPU

- Powerful and Inexpensive
  - Semiconductor capability, driven by advances in fabrication technology, market
- Flexible and programmable
- Limitations and difficulties

Design philosophies are different.
- The GPU is specialized for compute-intensive, massively data parallel computation (exactly what graphics rendering is about)
  - So, more transistors can be devoted to data processing rather than data caching and flow control

GPGPU Movement
- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

Application

- Physically Based Simulation
- Signal and Image Processing
  - Image segmentation
  - Computer Vision
  - Image Processing
- Geometric Computing

GPGPU Movement

- Significant application-level speedup over uni-processor execution
  - No more “killer micros”
- Easy entrance
  - An initial, naïve code typically get at least 2-3X speedup
- Wide availability to end users
  - available on laptops, desktops, clusters, super-computers
GPGPU Constraints

- Dealing with graphics API
  - Working with the corner cases of the graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
- Instruction sets
  - Lack of integer & bit ops

These have all changed with CUDA!

CUDA — C with no shader limitations!

- Integrated host+device app C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code

Parallel Kernel (device)
KernelA<<<nBlk, nTid>>>(args);

Serial Code (host)
KernelB<<<nBlk, nTid>>>(args);

CUDA Extends C

- Declspecs
  - global, device, shared, local, constant
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads()
- Runtime API
  - Memory, symbol, execution management
- Function launch

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

A Simple Running Example

Matrix Multiplication

- $P = M \times N$ of size $WIDTH \times WIDTH$
- Without tiling:
  - One thread calculates one element of $P$
  - $M$ and $N$ are loaded $WIDTH$ times from global memory
Step 1: Matrix Multiplication
A Simple Host Version in C

// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width) {
    for (int i = 0; i < Width; ++i) {
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * WIDTH + k];
                double b = N[k * WIDTH + j];
                sum += a * b;
            }
            P[i * WIDTH + j] = sum;
        }
    }
}

Step 2: Input Matrix Data Transfer
(Host-side Code)

void MatrixMulOnDevice(float* M, float* N, float* P, int Width) {
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;
    // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
    // Allocate P on the device
    cudaMalloc(&Pd, size);
}

Step 3: Output Matrix Data Transfer
(Host-side Code)

2. // Kernel invocation code – to be shown later
   ...
3. // Read P from the device
   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
   // Free device matrices
cudaFree(Md); cudaFree(Nd); cudaFree(Pd);

Step 4: Kernel Function

// Matrix multiplication kernel – per thread code
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
        float Melement = Md[ty * Width + k];
        float Nelement = Nd[k * Width + tx];
        Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
    // each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}

Step 5: Kernel Invocation
(Host-side Code)

// Setup the execution configuration
dim3 dimBlock(Width, Width);
dim3 dimGrid(1, 1);
// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);
Only One Thread Block Used

• One Block of threads compute matrix P_d
  – Each thread computes one element of P_d
• Each thread
  – Loads a row of matrix M_d
  – Loads a column of matrix N_d
  – Perform one multiply and addition for each pair of M_d and N_d elements
  – Compute to off-chip memory access ratio close to 1:1 (not very high)
• Size of matrix limited by the number of threads allowed in a thread block

© David Kirk/NVIDIA and Wen-mei W. Hwu
Taiwan, June 30-July 2, 2008