

Introduction

- ▶ In theory, GPUs are very fast, $\sim 1e10^9$ FLOPS
- ▶ Not all problems are suited to these devices
- ▶ The trick is knowing when it's worth it

Architecture

- ▶ One GPU typically consists of hundreds of cores
- ▶ Each core executes one **thread**, cores are dynamically clustered into **blocks**
- ▶ User writes a function (called a **kernel**) that executes concurrently on many cores
- ▶ Device usually has $\sim 1 - 2$ GB of global memory
- ▶ Blocks have fast shared memory banks, cores have registers

GPU Diagram

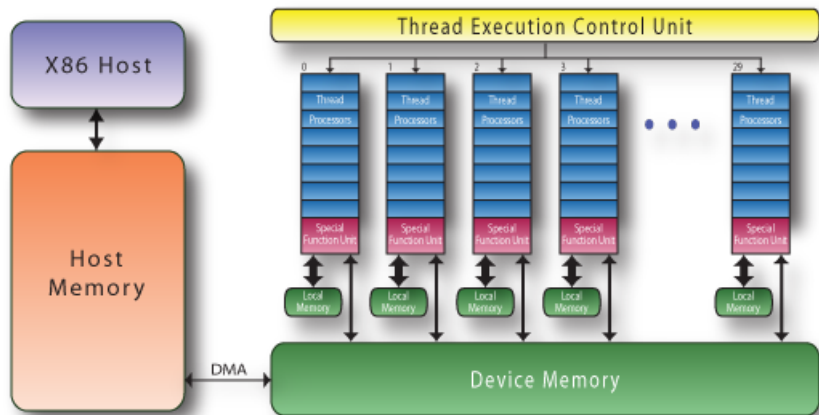


Image source: The Portland Group

Memory Access

- ▶ Threads can read/write shared and global memory, but not CPU memory
- ▶ Writes to global memory synced at end of kernel call
- ▶ Writes to shared memory can be synced within block on command
- ▶ Writes within **warp** (sets of 32 threads) are automatically synced
- ▶ Memory architecture limits amount of interdependence in problems

Interfaces

- ▶ NVIDIA's native interface is a C++-like language called CUDA
- ▶ OpenCL supports both NVIDIA and ATI - has bindings for C and Fortran
- ▶ There are interfaces with
 - ▶ MATLAB/Mathematica
 - ▶ Python/Perl
 - ▶ Fortran (PGI = \$\$\$)

CUDA

- ▶ Compiled language based on C++
- ▶ Defines a new type of function called a kernel, to be run on GPU
- ▶ Requires basic knowledge of C and pointers
- ▶ Allows for memory allocation on GPU and calling of kernels
- ▶ Compiled using NVIDIA's `nvcc`

Simple Example I

- ▶ First we define our kernel

```
__global__ void vecAdd(double* x, double* y, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    if (i < N) {
        y[i] = y[i] + x[i];
    }
}
```

- ▶ The variables `blockIdx.x`, `blockDim.x`, and `threadIdx.x` are given to each instance of the kernel

Simple Example II

- ▶ Now we allocate memory on the device

```
double* d_x, d_y;  
int vsize = sizeof(double)*N;  
cudaMalloc((void**)&d_x, vsize);  
cudaMalloc((void**)&d_y, vsize);
```

- ▶ Then we copy our data over to this memory

```
cudaMemcpy(d_x, h_x, vsize, cudaMemcpyHostToDevice);
```

- ▶ Finally we execute the kernel

```
int threadsPerBlock = 16;  
int numBlocks = N/threadsPerBlock; // assume divisible  
vecAdd<<<numBlocks, threadsPerBlock>>>(d_x, d_y, N);
```

- ▶ Copy back to CPU using cudaMemcpy

```
cudaMemcpy(h_y, d_y, vsize, cudaMemcpyDeviceToHost);
```


Things to Note

- ▶ Global memory is not updated across threads - can't do

```
vout[i] = vin[i] + 1;  
vout[i] = vout[i+1];
```

- ▶ Consecutive memory reads are coalesced within half-warps (16 threads)
- ▶ To construct 2D kernels, `threadsPerBlock` and `numBlocks` are 2D (x and y)
- ▶ Volume of blocks must not exceed 512

Multiple Functions

- ▶ Can define multiple kernels that call one another
- ▶ `__global__` keyword allows calling from CPU
- ▶ `__device__` functions can only be called from GPU

```
__device__ double utility(double c, double sigma)
{
    return (powf(c,1.0-sigma)-1.0)/(1.0-sigma);
}
```

- ▶ Device functions are automatically inlined by compiler

Shared Memory

- ▶ Shared memory - use memory across threads within block

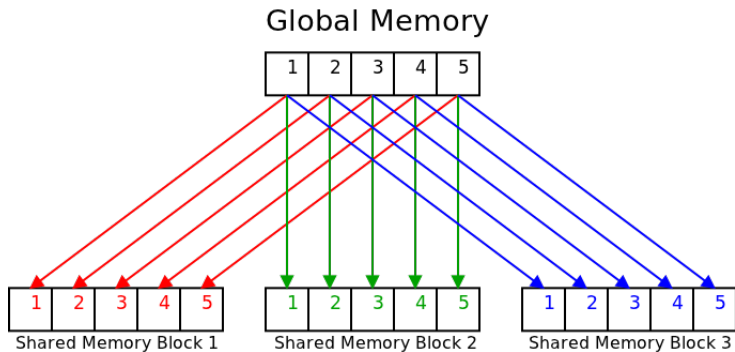
```
__global__ void vec_ma(double* vin, double* vout, double* mult)
{
    __shared__ double s_mult[M];

    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + tid;
    if (i < N) {
        if (tid < M) s_mult[tid] = mult[tid];
        __syncthreads();

        double v = 0.0;
        for (int j = 0; j < M; j++) {
            if (i-j >= 0) v = v + vin[i-j]*s_mult[j];
        }

        vout[i] = v;
    }
}
```

Shared Memory



Shared Memory

- ▶ Declare/allocate the memory, M must be constant

```
__shared__ double s_mult[M];
```

- ▶ Have each thread copy one value, need $M < BlockSize$

```
if (tid < M) s_mult[tid] = mult[tid];
```

- ▶ Sync data across thread block

```
__syncthreads();
```

- ▶ Data within warp is automatically synced

Value Function Iteration

- ▶ Each thread handles one state point
- ▶ Performs optimization over choice variables, updates value function
- ▶ Or you could calculate value for each state/choice pair and find max in 2D

Tricks

- ▶ Texture memory for lookup tables
- ▶ Do max absolute difference on GPU
- ▶ Utilize shared memory (transition probabilities, shock values)
- ▶ Debugging can be a pain, but there are ways to output text
- ▶ Use powers of 2 for grid size: modulus is slow, memory likes to be aligned

```
int ik = i & (nK-1);  
int iz = (i-ik)/nK;
```

Reduction

- ▶ Operations like `sum` and `max` must be done differently
- ▶ Here's how to do it when `blockSize = 512`

```
int tid = threadIdx.x;
int i = blockIdx.x*blockDim.x + tid;

if (tid < N) s_data[tid] = vin[i];
__syncthreads();

if (tid < 256) s_data[tid] = fmax(s_data[tid],s_data[tid+256]); // sync
if (tid < 128) s_data[tid] = fmax(s_data[tid],s_data[tid+128]); // sync
if (tid < 64) s_data[tid] = fmax(s_data[tid],s_data[tid+ 64]); // sync
if (tid < 32) {
    s_data[tid] = fmax(s_data[tid],s_data[tid+ 32]);
    s_data[tid] = fmax(s_data[tid],s_data[tid+ 16]);
    s_data[tid] = fmax(s_data[tid],s_data[tid+ 8]);
    s_data[tid] = fmax(s_data[tid],s_data[tid+ 4]);
    s_data[tid] = fmax(s_data[tid],s_data[tid+ 2]);
    s_data[tid] = fmax(s_data[tid],s_data[tid+ 1]);
}

if (tid == 0) vout[blockIdx.x] = s_data[0];
```


Texture Memory

- ▶ Texture memory can be good for read-only memory like lookup tables
- ▶ Fast caching, hardware interpolation
- ▶ Declare texture reference

```
texture<double> texRef;
```

- ▶ Bind it to some memory

```
cudaBindTexture(0, texRef, d_tex, tex_size);
```

- ▶ Access in kernel

```
double x = 0.457;  
double v = tex1D(texRef, x);
```

When To Use

- ▶ Ideal for high computation, low memory access problems
 - ▶ Many CPU problems are already memory bound
 - ▶ Not a huge speedup ($< 2x$) on matrix multiplication
- ▶ Local and sequential memory access patterns
 - ▶ Coalesced reads are faster
 - ▶ Global memory utilizes caching
- ▶ Avoid branching execution from conditional statements
 - ▶ Warp divergence - executes each branch serially
- ▶ You always pay the fixed cost of initial and final data transfer
 - ▶ Increasing returns

Use Cases

- ▶ Well suited to GPU
 - ▶ Value function iteration - in 1D or 2D
 - ▶ Simulation - Monte Carlo, simulated method of moments, random numbers
 - ▶ Steady states - transition matrix iteration, discrete
- ▶ Not so much
 - ▶ Sorting (e.g., bubble sort)
 - ▶ Histograms
 - ▶ VFI accelerator
- ▶ Even for the latter cases, GPU optimized algorithms are getting better

Other Interfaces

- ▶ BLAS routines come for free with CUDA runtime
- ▶ CULA Tools gives advanced linear algebra routines
- ▶ PyCUDA provides a very easy interface in Python

```
mod = SourceModule("""
    __global__ void vecAdd(double* x, double* y, int N)
    {
        int i = threadIdx.x*blockSize.x + threadIdx.x;
        if (i < N) y[i] = y[i] + x[i];
    }
    """)
func = mod.get_function("vecAdd")
func(d_x,d_y,N,block=(BS,1,1))
```

MATLAB Interface (NVIDIA only)

- ▶ Compile kernel from CUDA into PTX using `nvcc`

```
nvcc -ptx -O2 --ptxas-options=-v vecAdd.cu
```

- ▶ Load kernel into MATLAB

```
kern = parallel.gpu.CUDAKernel('vecAdd.ptx', 'vecAdd.cu');  
kern.ThreadBlockSize = 32;  
kern.GridSize = ceil(N/kern.ThreadBlockSize(1));
```

- ▶ Transfer input and allocate output

```
d_in = gpuArray(h_in);  
d_out = gpuArray(N);
```

- ▶ Execute kernel and copy result back to CPU side

```
d_out = feval(kern,d_out,d_in,N);  
h_out = gather(d_out);
```

- ▶ Can also write MEX file in CUDA

Hardware

- ▶ You may need to buy a new graphics card
- ▶ Can run from \$100-\$300+ depending on what you want
- ▶ Rapidly changing landscape - convergence of GPUs and CPUs?

References

- ▶ NVIDIA
`developer.nvidia.com/category/zone/cuda-zone`
- ▶ CUDA Toolkit (nvcc)
`developer.nvidia.com/cuda-toolkit-40`
- ▶ Jesus's paper on applications to VFI and associated code
- ▶ PyCUDA `mathema.tician.de/software/pycuda`