

CUDA

System heterogeneity

- From Pollack's rule: Heterogeneous system offers potential for good performance on range of workloads
- Same idea for other specialized hardware: wastes silicon sometimes, but may be worthwhile if it's sufficiently useful often enough

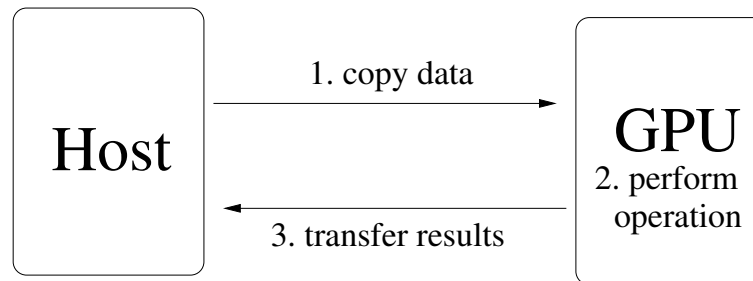
GPGPU programming

(General-Purpose Graphics Processing Unit)

- Graphics processing requires many similar operations in “graphics pipeline”
 - Triangles going through rotation and scaling, shading, and texturing
- Graphics Processing Units (GPUs) develop to meet this need and then get converted for general purpose programs
- CUDA (Compute Unified Device Architecture) is a GPU design and extension of C (et al) to support GPGPU programming developed by Nvidia
 - Market share leader; leading open alternative is OpenCL

Programming model: Memory

- Program mainly runs on “Host” (= CPU), but can call functions on “Device” (= GPU)
- Host and Device have separate address spaces (at least historically)
 - Memory must be explicitly transferred



Programming model: Processing

- GPU can run many threads simultaneously, but not independently
 - Device threads connected in groups called warps
 - All members of a warp perform the same operation
 - SIMD = Single Instruction, Multiple Data
- Programmer writes function to run on device (kernel)
- Invokes it with a number of blocks and threads (per block)
- All these threads run the function
 - Use implicit arguments blockIdx and threadIdx to identify itself

Logistics: Compiling and running CUDA programs

- Compiling:

```
nvcc -o hello hello.cu
```

Name of the source code. CUDA files use the extension .cu

Compiler; really a wrapper for gcc or another C compiler

-o flag: Used to specify the executable to create

- Running:

```
./hello
```

“Hello World” for CUDA

```
#include <stdio.h>
```

```
__global__ void hello() {  
    int id = threadIdx.x + blockIdx.x * blockDim.x;  
    printf("Hello from thread %d (%d of block %d)\n", id, threadIdx.x, blockIdx.x);  
}
```

```
int main() {  
    hello<<<3,4>>>();           //launch 3 blocks of 4 threads each  
  
    cudaDeviceSynchronize();   //make sure kernel completes  
}
```

“Hello World” for CUDA

```
#include <stdio.h>
```

```
__global__ void hello() {  
    int id = threadIdx.x + blockIdx.x * blockDim.x;  
    printf("Hello from thread %d (%d of block %d)\n", id, threadIdx.x, blockIdx.x);  
}
```

Device code (runs on GPU)

```
int main() {  
    hello<<<3,4>>>();           //launch 3 blocks of 4 threads each  
  
    cudaDeviceSynchronize();   //make sure kernel completes  
}
```

Possible output

Hello from thread 0 (0 of block 0)

Hello from thread 1 (1 of block 0)

Hello from thread 2 (2 of block 0)

Hello from thread 3 (3 of block 0)

Hello from thread 8 (0 of block 2)

Hello from thread 9 (1 of block 2)

Hello from thread 10 (2 of block 2)

Hello from thread 11 (3 of block 2)

Hello from thread 4 (0 of block 1)

Hello from thread 5 (1 of block 1)

Hello from thread 6 (2 of block 1)

Hello from thread 7 (3 of block 1)



Out of order; blocks are executed separately and in any order

Overview of a CUDA program

- In host code:
 - Allocate memory on device
 - Copy data to device
 - Kernel call
 - Copy results to host
 - Free device memory
- In device code:
 - `__global__`
 - determine thread ID
 - bounds check

Adding vectors using CUDA

(Not actually fast...)

- Take 2 input arrays and add index-wise to produce output array

$$\begin{array}{|c|} \hline 2 \\ \hline 5 \\ \hline 3 \\ \hline 1 \\ \hline 0 \\ \hline 6 \\ \hline \end{array} + \begin{array}{|c|} \hline 5 \\ \hline 3 \\ \hline 2 \\ \hline 0 \\ \hline 4 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|} \hline 7 \\ \hline 8 \\ \hline 5 \\ \hline 1 \\ \hline 4 \\ \hline 7 \\ \hline \end{array}$$

Adding vectors using CUDA

(Not actually fast...)

```
int main() {  
    int* a;          //first input array (on host)  
    int* a_dev;     //first input array (on device)  
  
    a = (int*) malloc(N*sizeof(int));  
    cudaMalloc((void**) &a_dev, N*sizeof(int));  
  
    ...             //same for b and res  
  
    free(a);  
    cudaFree(a_dev);  
}
```

- In host code:
 - **Allocate memory on device**
 - Copy data to device
 - Kernel call
 - Copy results to host
 - **Free device memory**
- In device code:
 - `__global__`
 - determine thread ID
 - bounds check

Adding vectors using CUDA

(Not actually fast...)

```
int main() {
```

```
...
```

```
cudaMemcpy(a_dev, a, N*sizeof(int),  
           cudaMemcpyHostToDevice));
```

```
...
```

```
cudaMemcpy(res, res_dev, N*sizeof(int),  
           cudaMemcpyDeviceToHost)
```

```
...
```

```
}
```

- In host code:
 - Allocate memory on device
 - **Copy data to device**
 - Kernel call
 - **Copy results to host**
 - Free device memory
- In device code:
 - `__global__`
 - determine thread ID
 - bounds check