The Abdus Salam International Centre for Theoretical Physics

Introduction to CUDA - 1

... curtsey of Dr. Massimo Bernaschi (CNR - http://www.iac.cnr.it/~massimo)

Ivan Girotto – igirotto@ictp.it

Information & Communication Technology Section (ICTS)

International Centre for Theoretical Physics (ICTP)

What is CUDA?

- CUDA = Compute Unified Device Architecture
  - Expose general-purpose GPU computing as first-class capability
  - Retain traditional DirectX/OpenGL graphics performance

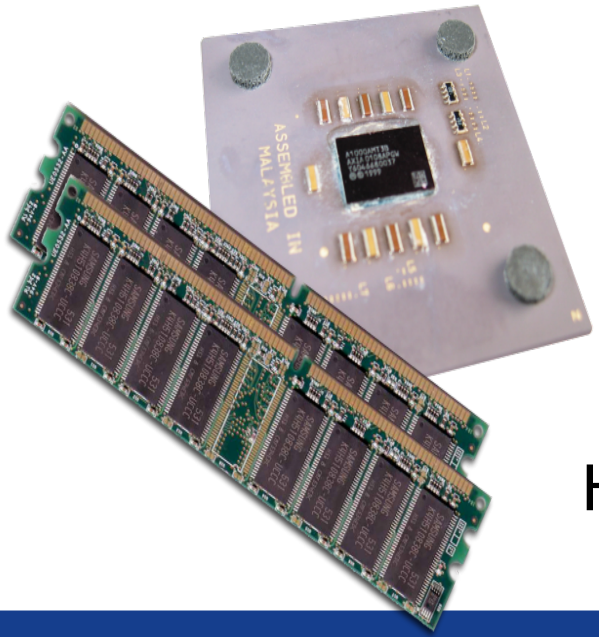
- CUDA C
  - Based on industry-standard C
  - A handful of language extensions to allow heterogeneous programs
  - Straightforward APIs to manage devices, memory, etc.

CUDA Programming Model

- The GPU is viewed as a compute device that:
  - has its own RAM (**device memory**)
  - runs data-parallel portions of an application as **kernels** by using many threads
- GPU *vs.* CPU threads
  - GPU threads are **extremely lightweight**
    - Very little creation overhead
  - GPU needs **1000s of threads for full efficiency**
    - A multi-core CPU needs only a few (basically one thread *per core)*

CUDA C Jargon: The Basics

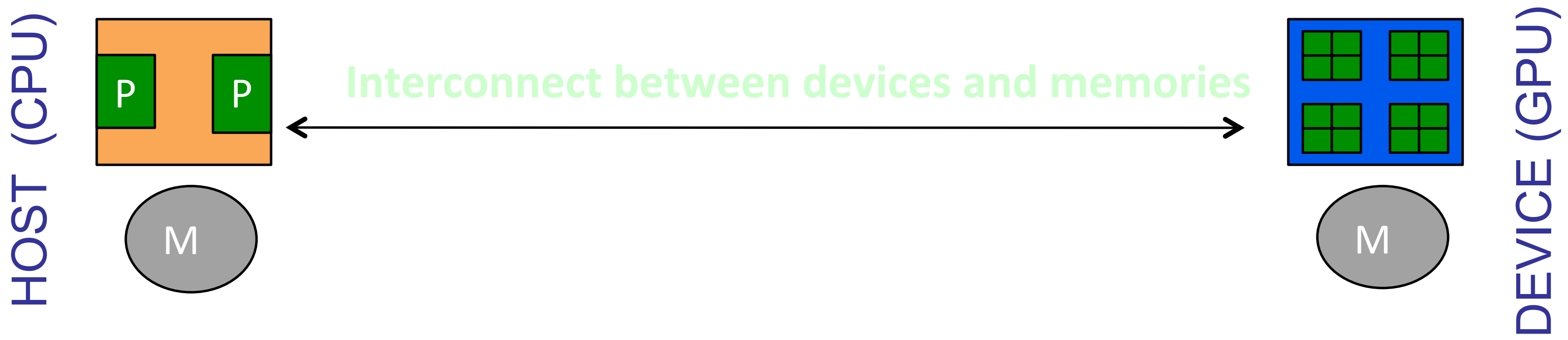
- The CPU and its memory (host memory)
  - The GPU and its memory (device memory)



Host

Device

What Programmer Expresses in CUDA



What Programmer Expresses in CUDA

- ✓ Computation partitioning (where does computation occur?)
  - ✓ Declarations on functions __host__, __global__, __device__
  - ✓ Mapping of thread programs to device: **compute <<<gs, bs>>>(<args>)**
- ✓ Data partitioning (where does data reside, who may access it and how?)
  - ✓ Declarations on data __shared__, __device__, __constant__, …
- ✓ Data management and orchestration
  - ✓ Copying to/from host: *e.g.,* cudaMemcpy(h_obj,d_obj, size, cudaMemcpyDevicetoHost)
- ✓ Concurrency management
  - ✓ *e.g.* __synchthreads()

Hello, World!

```
int main( void ) {
   printf( "Hello, World!\n" );
   return 0;
}
```

- To compile: **nvcc –o hello_world hello_world.cu**
- To execute: **./hello_world**
- This basic program is just standard C that runs on the *host*
- NVIDIA's compiler (**nvcc**) will not complain about CUDA programs with no *device* code
- At its simplest, CUDA C is just C!

Hello, World! with Device Code

```
__global__ void kernel( void ) {
}

int main( void ) {

    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

To compile: **nvcc –o simple_kernel simple_kernel.cu**

To execute: **./simple_kernel**

Hello, World! with Device Code

```
__global__ void kernel( void ) {
}
```

- CUDA C keyword **`__global__`** indicates that a function
  - Runs on the device
  - Called from host code

- **`nvcc`** splits source file into host and device components
  - NVIDIA's compiler handles device functions like **`kernel`**()
  - Standard host compiler handles host functions like **`main`**()
    - **`gcc, icc,`** …
    - **Microsoft Visual C**

Hello, World! with Device Code

```
int main( void ) {
    kernel<<< 1, 1 >>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- Triple angle brackets mark a call from *host* code to *device* code
  - A “kernel launch” in CUDA jargon
  - We’ll discuss the parameters inside the angle brackets later
- This is all that’s required to execute a function on the GPU!

A More Complex Example

- A kernel to add two integers:

```
__global__ void add( int *a, int *b, int *c ) {
        *c = *a + *b;
}
```

- As before, `__global__` is a CUDA C keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

A More Complex Example

- Notice that now we use ***pointers*** for all our variables:

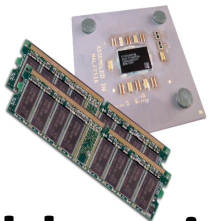
```
__global__ void add( int *a, int *b, int *c ) {
        *c = *a + *b;
}
```

- `add()` runs on the device…so **a**, **b**, and **c** must point to device memory
- How do we allocate memory on the GPU?

Memory Management

- Up to CUDA 4.0 host and device memory were distinct entities from the programmers' viewpoint
  - Device pointers point to GPU memory
    - May be passed to and from host code
    - (In general) May not be dereferenced from host code

  - Host pointers point to CPU memory
    - May be passed to and from device code
    - (In general) May not be dereferenced from device code



Starting on CUDA 4.0 there is a **Unified Virtual Addressing** feature.

Memory Management

- Basic CUDA API for dealing with device memory
  - `cudaMalloc(&p, size)`, `cudaFree(p)`, `cudaMemcpy(t, s, size, direction)`
  - Similar to their C equivalents: `malloc()`, `free()`, `memcpy()`

pointer to pointer

A More Complex Example: `main()`

```
int main( void ) {
    int a, b, c;                 // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
    int size = sizeof( int );    // we need space for an integer
    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );
    a = 2;
    b = 7;
// copy inputs to device
    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );
    // launch add() kernel on GPU, passing parameters
    add<<< 1, 1 >>>( dev_a, dev_b, dev_c );
    // copy device result back to host copy of c
    cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );
    cudaFree( dev_a ); cudaFree( dev_b ); cudaFree( dev_c )
    return 0;
}
```

```
#include "cuPrintf.cu"

__global__ void testKernel(int param){
    cuPrintf("Param value: %d\n", param);
}

int main(void){
    // initialize cuPrintf
    cudaPrintfInit();
    int a = 456;
    testKernel<<<4,1>>>(a);
    // display the device's greeting
    cudaPrintfDisplay();
    // clean up after cuPrintf
    cudaPrintfEnd();
}  // compile with nvcc -o test.x test.cu -I$CUDADIR/samples/0_Simple/simplePrintf
```

Also simple printf() works!!!

CUDA Error Checking

- CUDA host function calls usually return a value of type **cudaError_t**

  cudaError_t cudaMalloc (void **devPtr, size_t size)

- Example: to check if device allocation was successful

```
cudaError_t error;
[...]
error = cudaMalloc(&d_a, memSize);
if (error != cudaSuccess)
{
        printf("Error in device allocation: %s\n",! ! ! ! cudaGetErrorString(error));
}
```

CUDA Error Checking

- Kernels can't have a return value, so cudaGetLastError() is used

```
cudaError_t error;
[...]
myKernel<<<1, 1>>>(a_d);
error = cudaGetLastError();
if (error != cudaSuccess)
{
      printf("Error in Kernel  execution: %s\n", cudaGetErrorString(error) );
}
```