Introduction to GPU Programming

Hsi-Yu Schive (薛熙于)
Outline of GPU Lab Session

• Introduction to GPU

• GPU programming: CUDA
  • Working flow

• Simple examples
  • Array manipulation
  • Matrix multiplication
  • GPU shared memory

• Exercises: simple direct N-body code using CPU only
  → Rewrite to support GPU acceleration
Why GPU?

Performance: GPU vs. CPU
~ 7x

Bandwidth: GPU vs. CPU
~ 6x

Reference: CUDA Programming Guide
Why is GPU faster?

- In GPU, more transistors are devoted to data processing
- GPU is suitable for data-parallel computations with high arithmetic intensity
  → That’s what scientific computations usually need!
    (e.g., N-Body, fluid dynamics, matrix manipulation, …)

Reference: CUDA Programming Guide
Programming Language: CUDA (Compute Unified Device Architecture)

- GPU → **multithreaded coprocessor** to CPU
  - Execute thousands of threads in parallel
  - All threads execute the same function (called “kernel”)

![Diagram showing GPU as a multithreaded coprocessor with kernels and processors](image)
CUDA Working Flow

1. Declare and allocate host (CPU) and device (GPU) memories
2. Initialize the host arrays
3. Transfer data from host to device (CPU → GPU)
4. Do your calculations by GPU (invoke the GPU kernel)
5. Transfer data back from device to host (GPU → CPU)
CUDA Working Flow

1. Declare and allocate host (CPU) and device (GPU) memories
2. Initialize the host arrays
3. Transfer data from host to device (CPU → GPU)
4. Do your calculations by GPU (invoke the GPU kernel)
5. Transfer data back from device to host (GPU → CPU)
Manipulate GPU Memory

• CPU and GPU have different memory spaces
  • Both need to be declared and allocated explicitly

• CPU memory:

```c
float *Array_CPU = (float*)malloc( Array_Size );
free( Array_CPU );
```

• GPU memory:

```c
float *Array_GPU;
cudaMalloc( &Array_GPU, Array_Size );
cudaFree( Array_GPU );
```
Manipulate GPU Memory

• The GPU memory we just allocate is called the **global memory**
  • Need to be allocated and deallocate by the host (CPU) code
  • Need to be transferred in between CPU and GPU explicitly
  • Can be accessed by **ALL CUDA threads**
  • Largest memory space in GPU, usually a few GB
  • Bandwidth ~ a few hundred GB/s → high, but not high enough compared with the GPU computing power

• Another important memory in GPU is called the **shared memory**
  • Much faster, but also much smaller than the global memory
  • Will be discussed later
CUDA Working Flow

1. Declare and allocate host (CPU) and device (GPU) memories
2. Initialize the host arrays
3. Transfer data from host to device (CPU → GPU)
4. Do your calculations by GPU (invoke the GPU kernel)
5. Transfer data back from device to host (GPU → CPU)
Data Transfer: CPU ↔ GPU

• CPU → GPU:

```c
const int Array_Size = 1024;
float *GPU_Array, *CPU_Array;
cudaMalloc( &Array_GPU, Array_Size);

cudaMemcpy( Array_GPU, Array_CPU, Array_Size, cudaMemcpyHostToDevice);
```

• GPU → CPU:

```c
cudaMemcpy( Array_CPU, Array_GPU, Array_Size, cudaMemcpyDeviceToHost);
```

• Data transfer is expensive → minimize it as much as possible!
CUDA Working Flow

1. Declare and allocate host (CPU) and device (GPU) memories
2. Initialize the host arrays
3. Transfer data from host to device (CPU → GPU)
4. Do your calculations by GPU (invoke the GPU kernel)
5. Transfer data back from device to host (GPU → CPU)
CUDA Thread Hierarchy

- Two-level hierarchy
  - A grid contains GRID_SIZE thread blocks
  - A thread block contains BLOCK_SIZE threads
  - Total number of threads = GRID_SIZE * BLOCK_SIZE

- Block index: index of block within a grid
  - Can be 1D, 2D, or 3D

- Thread index: index of thread within a block
  - Can be 1D, 2D, or 3D

- Build-in variables
  - blockDim.x, blockIdx.x, threadIdx.x

Reference: CUDA Programming Guide
Invoke GPU Kernel

```c
int N = 10;
float *Array_GPU; // must be pre-allocated

GPU_Kernel <<< GRID_SIZE, BLOCK_SIZE >>> ( N, Array_GPU );
```

- **GPU_Kernel**: the name of your GPU function
- **<<< GRID_SIZE, BLOCK_SIZE >>>**: specify the thread hierarchy
- **N**: passed by value
  - Transferred *implicitly* into GPU
- **Array_GPU**: array pointing to the GPU global memory
  - Must be allocated in advance and transferred *explicitly*
  - For INPUT  ➞ transferred BEFORE the kernel invocation
  - For OUTPUT ➞ transferred AFTER the kernel invocation
Define GPU Kernel

• Example: vector addition (C=A+B)

• CPU code: loop over all N elements sequentially

```c
void VecAdd_CPU(int N, float *A, float *B, float *C) {
    for (int t=0; t<N; t++)
        C[t] = A[t] + B[t];
}
```

• GPU code: invoke N threads to calculate all N elements in parallel

```c
__global__
void VecAdd_GPU(int N, float *A, float *B, float *C) {
    int t = blockDim.x*blockIdx.x + threadIdx.x;
    C[t] = A[t] + B[t];
}
```
Define GPU Kernel

- Example: vector addition ($C = A + B$)
- CPU code: loop over all $N$ elements sequentially
- GPU code: invoke $N$ threads to calculate all $N$ elements in parallel

```c
void VecAdd_CPU( int N, float *A, float *B, float *C ) {
    for (int t=0; t<N; t++)
        C[t] = A[t] + B[t];
}

__global__
void VecAdd_GPU( int N, float *A, float *B, float *C ) {
    int t = blockDim.x*blockIdx.x + threadIdx.x;
    C[t] = A[t] + B[t];
}
```

Specify it’s a GPU kernel

Unique index to distinguish all threads
Define GPU Kernel

- Example: vector addition \((C = A + B)\)
- CPU code: loop over all \(N\) elements sequentially
- GPU code: invoke \(N\) threads to calculate all \(N\) elements in parallel

```c
void VecAdd_CPU(int N, float *A, float *B, float *C) {
    for (int t = 0; t < N; t++)
        C[t] = A[t] + B[t];
}

__global__ void VecAdd_GPU(int N, float *A, float *B, float *C) {
    int t = blockDim.x * blockIdx.x + threadIdx.x;
    C[t] = A[t] + B[t];
}
```

Each thread only calculates one element of \(C\)
Example 1 : SAXPY

- Single-precision $A^*[X] + [Y]$
  - $A$: scalar
  - $[X]$ and $[Y]$: arrays with size $N$

- Let’s see how to convert a CPU SAXPY code into GPU step-by-step
  - Example1_SAXPY/SAXPY_CPU.cpp
    1. Declare and allocate host (CPU) and device (GPU) memories
    2. Initialize the host arrays
    3. Transfer data from host to device (CPU $\rightarrow$ GPU)
    4. Do your calculations by GPU (invoke the GPU kernel)
    5. Transfer data back from device to host (GPU $\rightarrow$ CPU)

- Compilation: `nvcc Your_GPU_Code.cu`
Exercise 1 : Evolving Particles

• Simple 1\textsuperscript{st}–order Euler integration:

\[
\vec{r}_i(t + dt) = \vec{r}_i(t) + \vec{v}_i(t) \times dt \\
\vec{v}_i(t + dt) = \vec{v}_i(t) + \vec{a}_i(t) \times dt
\]

• For the \(i\textsuperscript{th}\) particle
• \(dt\): evolution time-step
• \(\vec{a}\): gravitational acceleration (assuming it has been calculated)
• For more advanced time integration schemes, see “HH Wang--Orbit Integrator” (tomorrow afternoon)

• Exercise\_SimpleNBody/SimpleNBody\_CPU.cu
  • Declare your GPU arrays for Pos, Vel, Acc
  • Transfer GPU arrays in between CPU and GPU
  • Rewrite the CPU function “EvolveParticle\_CPU” \(\rightarrow\) “EvolveParticle\_GPU”
  • Define proper BLOCK\_SIZE and GRID\_SIZE
  • Invoke the GPU kernel
Example 2: Matrix Multiplication

\[ C_{nm} = A_{nk} \times B_{km} \]

- Example2_MatMul/MatMul_CPU_GPU.cu
- Each thread calculates one element of C
- Two-dimensional threads and thread blocks

```c
row = blockIdx.y*blockDim.y + threadIdx.y;
col = blockIdx.x*blockDim.x + threadIdx.x;
```

- Using global memory only
  - A will be loaded B.width times
  - B will be loaded A.height times
  - Redundant global memory access
  - Please refer to “CUDA Programming Guide” for a much faster version taking advantage of the shared memory

Reference: CUDA Programming Guide
Exercise 2: Calculating Direct N-Body Force with Global Memory

\[ a_i = \sum_j \frac{m_j \hat{r}_{ij}}{(r_{ij}^2 + \varepsilon^2)^{3/2}}, \quad \hat{r}_{ij} = \hat{r}_j - \hat{r}_i \]

- Assuming \( G=m=1 \)
- Use thread \( i \) to calculate the acceleration on \( i^{th} \) particle exerted from all \( j \) particles
GPU Memory Hierarchy

- Global memory access is expensive
  - Reduce it as much as possible
- Shared memory and per-thread registers are much faster

Reference: CUDA Programming Guide
Exercise 2: Calculating Direct N-Body Force with **Global** Memory

- Performance optimization?
- Each thread will use the same i-particle position for N times
  - Store in the per-thread registers
- All threads will access the same j-particle position at the same time
  - Store in the per-block shared memory

![Diagram showing i-particle and j-particle arrays]
GPU Shared Memory

- Threads within the same block can share data through the shared memory
- Declared by the `__shared__` qualifier in the kernel
- Usually filled by copying data from the global memory
- Much faster than the global memory
- Very small compared to the global memory
  - Maximum per thread block ~ 48 KB
- **Must be synchronized by calling `__syncthreads()`**
  - Because some threads may run faster than the others
  - Add `__syncthreads()` to force all threads in the same block to wait until all preceding work is done
  - Otherwise the shared memory may be used BEFORE it’s ready
GPU Shared Memory

• Imagine you want to share pictures with your friends by uploading them onto Google drive

1. Apply Google drive
   ↔ declared shared memory array

2. Upload pictures
   ↔ global memory → shared memory

3. Make sure that the uploading is done and send a notice to your friends
   ↔ __syncthreads()

4. Friends can download your pictures
   ↔ all threads can start to access data just copied to the shared memory
#define BLOCK_SIZE 100

__global__ void Kernel( float *In ) {
    __shared__ float SharedMemoryArray[BLOCK_SIZE];
    int GlobalIdx = blockDim.x*blockIdx.x + threadIdx.x;

    SharedMemoryArray[threadIdx.x] = In[GlobalIdx];

    __syncthreads();

    ... (start to do your calculations);
}
Example 3 : Data Smoothing using Shared Memory

• $\text{Out}[i] = 0.25*\text{In}[i-1] + 0.5*\text{In}[i] + 0.25*\text{In}[i+1]$

• Example3_SharedMemory/Smoothing.cu

• Let each thread work on one element of $\text{Out}[]$

• Each element of $\text{In}[]$ will be loaded 3 times by nearby threads ...

→ Use shared memory!

• Be careful when loading the ghost-cell data
  • Assuming periodicity: $\text{In}[-1] = \text{In}[N-1]$; $\text{In}[N] = \text{In}[0]$;
  • For an adopted BLOCK_SIZE (number of threads per block), declare

    ```c
    __shared__ s_In[BLOCK_SIZE+2];
    ```

  • “+2” because we need one ghost cell on each side
  • We can use thread 0 to load these additional data
Exercise 3: Calculating N-Body Force with Shared Memory

- N: total number of particles
- BS=BLOCK_SIZE, GS=GRID_SIZE
- Taking advantage of shared memory
  - Each thread block loads BS j-particle data into the shared memory at a time
  - Calculate BS*BS pairwise accelerations
  - Move to the next set of BS j particles, accumulate the accelerations
  - Store accelerations of the current i particles back to the global memory
  - Move to the next set of i particles if necessary
Exercise 3: Calculating N-Body Force with Shared Memory

- Inside one thread block:
- Each thread loads one j-particle data into the per-block shared memory
- Synchronize
- Use thread k to accumulate the acceleration of k\textsuperscript{th} i particle exerted from all j particles in the shared memory
- Synchronize again before reloading the next set of j-particle data into the shared memory
Advanced Topics

• Multi-GPU acceleration (MPI + GPUs)
  • See “YH Tseng--Parallel Programming” in tomorrow’s lab session

• Heterogeneous Open/MPI/GPUs acceleration
  • Fully exploit the multi-CPU + multi-GPU horsepower

• Asynchronous data transfer between CPU and GPU
  • Overlap communication with CPU/GPU computations

• Concurrent execution between CPUs and GPUs
  • Overlap CPU computations with GPU computations
  • Double the performance if CPU and GPU consume about the same time

• Load balancing between multiple GPUs

• GPU libraries: cuFFT, cuBLAS, cuRand, cuSparse, …
GAMER: GPU-accelerated AMR
Multi-GPU Acceleration Summary

• Multi-GPU acceleration in GAMER relies on three parallelization levels

1. Different sub-domains ↔ Different GPUs
   ◆ Rectangular domain decomposition or Hilbert curve
   ◆ Data exchange: MPI

2. Different patches ↔ Different multiprocessors
   ◆ The boundary condition of each patch is prepared by CPU in advance

3. Different cells within the same patch ↔ Different processors within the same multiprocessor
   ◆ Store common and frequently reused data (e.g., fluid flux and potential) in the shared memory
GAMER Performance

**NERSC Dirac GPU Cluster**

- **GPU:** 1-32 NVIDIA Tesla C2050
- **CPU:** 1-32 Intel Xeon E5530

With self-gravity (80x speed-up in GPU) and individual time-step

- Async : CPU/GPU overlap
- OMP(4) : 4 OpenMP threads

32 GPU vs. 32 CPU cores: 71x
32 GPU vs. 128 CPU cores: 18x
⇒ Equivalent to 2,304 CPU cores

MPI ~ 11% of $T_{total}$
References

• Materials used in this course:
  • NVIDIA tutorial: https://developer.nvidia.com/how-to-cuda-c-cpp

• Useful information provided by Rainer Spurzem:
  • N-body workshop: http://kiaa.pku.edu.cn/~kouwenhoven/nbody.html
  • NBODY6++GPU paper: http://ads.bao.ac.cn/abs/2015MNRAS.450.4070W