Introduction to GPU Computing

Junjie Lai, NVIDIA Corporation
Outline

- Evolution of GPU Computing
  - Heterogeneous Computing
  - CUDA Execution Model & Walkthrough of Hello World
  - Walkthrough : 1D Stencil
Once upon a time . . .

- **GPU:** Graphical Processing Unit
  - Originated as specialized hardware for 3D games.

- **Why a different processor?**
  - Rendering is the most computationally intense part of a game.
  - CPU is not an ideal device for computer graphics rendering
  - Freed CPU allows more complex AI, dynamic world generation, realistic dynamics.
GPU Computing Era: G80 and Fermi

- **G80 (2006)**
  - Unified Shader Architecture
  - Double Precision
  - CUDA Introduced

- **Fermi (GF100) (2010)**
  - ECC
  - Enhanced Double precision
  - Memory hierarchy and expanded caching
12 years later

- NVIDIA Kepler
  - 1.31 tflop double precision
  - 3.95 tflop single precision
  - 250 gb/sec memory bandwidth
  - 2,688 Functional Units (cores)
- ~= #1 on Top500 in 1997
Science Uses GPUs

Medical Imaging
U of Utah

Molecular Dynamics
U of IL, Urbana

Video Transcoding
Elemental Tech

Matlab Computing
AccelerEyes

Astrophysics
RIKEN

Financial Simulation
Oxford

Linear Algebra
Universidad Jaime

3D Ultrasound
Techniscan

Quantum Chemistry
U of Illinois, Urbana

Gene Sequencing
U of Maryland
3 Ways to Accelerate Applications

Libraries

Directives

Programming Languages

Easiest Approach for 2x to 10x Acceleration

Maximum Performance
Takeaways

- GPUs accelerate science
  - Programmable, and become easier to program
  - Increase throughput on parallel computations
  - GPUs have proven their worth in accelerating real parallel applications.
Outline

- Evolution of GPU Computing
- Heterogeneous Computing
- CUDA Execution Model & Walkthrough of Hello World
- Walkthrough : 1D Stencil
Heterogeneous Computing

- Use latency optimized processor for serial part
- Use throughput optimized processor for parallel part
```
#include <iostream>
#include <algorithm>
using namespace std;
#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__
void stencil_1d(
    int *in,
    int *out) {

    __shared__
    int temp[BLOCK_SIZE + 2 * RADIUS];

    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(
    int *x,
    int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;
    // host copies of a, b, c
    int *d_in, *d_out;
    // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int*) malloc(size);
    fill_ints(in, N + 2*RADIUS);
    out = (int*) malloc(size);
    fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void**)&d_in, size);
    cudaMalloc((void**)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
                                         d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in);
    cudaFree(d_out);
    return 0;
}
```
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it
1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it
3. Copy results from GPU memory to CPU memory
Device Management

- APIs to query and select GPUs

  ```c
  cudaGetDeviceCount(int *count)
  cudaSetDevice(int device)
  cudaGetDevice(int *device)
  cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
  ```

- Keywords to distinguish between host and device code

  ```c
  __global__, __device__ (), __host__
  ```
Memory Management

- CPU and GPU have separate memory spaces
  Data is moved across PCIe bus

- APIs to allocate/free memory on device, copy data between host & device

  cudaMalloc ( void** devPtr, size_t size )
  cudaFree ( void* devPtr )
  cudaMemcpy ( void* devPtr, int value, size_t count )
  cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind )
Outline

- Evolution of GPU Computing
- Heterogeneous Computing
- CUDA Execution Model & Walkthrough of Hello World
- Walkthrough : 1D Stencil
Some Terminology

- **Kernel**: threads running in parallel
  - Declaration: `__global__` void kernel (param0, param1, ...);
  - Launch: `kernel <<< GRID, BLK>>> (param0, param1, ...);

- **Global Memory**: GPU’s on-board DRAM

- **Shared Memory**: On-chip fast memory local to a thread block
CUDA Execution Model

- **Thread**: Sequential execution unit
  - All threads execute same sequential program
  - Threads execute in parallel

- **Threads Block**: a group of threads
  - Executes on a single Streaming Multiprocessor (SM)
  - Threads within a block can cooperate
    - Light-weight synchronization
    - Data exchange

- **Grid**: a collection of thread blocks
  - Thread blocks of a grid execute across multiple SMs
  - Thread blocks do not synchronize with each other
  - Communication between blocks is expensive
IDs and Dimensions

- A kernel is launched as a grid of blocks of threads

- Built-in variables:
  - threadIdx
  - blockIdx
  - blockDim
  - gridDim
```c
#include <iostream>
#include <algorithm>
using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__
void stencil_1d(int* in, int* out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int* x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int* in, *out;

    // host copies of a, b, c
    int* d_in, *d_out;

    // device copies of a, b, c

    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int*)malloc(size);
    fill_ints(in, N + 2*RADIUS);
    out = (int*)malloc(size);
    fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void**)&d_in, size);
    cudaMalloc((void**)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    free(in); free(out);
    cudaFree(d_in);
    cudaFree(d_out);
    return 0;
}
```

CUDA Code Example

void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nbblocks, 256>>>(n, 2.0, x, y);
Hello World!

threadIdx == 21
Consider indexing an array with one element per thread (8 threads/block)

With M threads per block, a unique index for each thread is given by:

```c
int index = blockIdx.x * M + threadIdx.x;
```
Allocate memory for 2 vectors on host and device
Initialize 1 source vector
Copy 1 source vector to device (GPU)
Vector operation on GPU
Copy result vector to host (CPU)
int main () {
    float *vecA;    float *vecB;
    float *d_vecA;    float *d_vecB;
    int SIZ = (int)10240000;
    vecA = (float*)malloc(SIZ*sizeof(float));
    vecB = (float*)malloc(SIZ*sizeof(float));
    cudaMalloc((void**)&d_vecA, SIZ*sizeof(float));
    cudaMalloc((void**)&d_vecB, SIZ*sizeof(float));
Code Walkthrough 2

```c
int main () {
    float *vecA;   float *vecB;
    float *d_vecA; float *d_vecB;
    int SIZ = (int)10240000;
    vecA = (float*)malloc(SIZ*sizeof(float));
    vecB = (float*)malloc(SIZ*sizeof(float));
    cudaMalloc((void**)&d_vecA, SIZ*sizeof(float));
    cudaMalloc((void**)&d_vecB, SIZ*sizeof(float));
    for(int i=0; i<SIZ; i++)
        vecA[i] = (float)(rand())/(float)(RAND_MAX);
```
int main () {
    float *vecA;    float *vecB;
    float *d_vecA;    float *d_vecB;
    int SIZ = (int)10240000;
    vecA = (float*)malloc(SIZ*sizeof(float));
    vecB = (float*)malloc(SIZ*sizeof(float));
    cudaMemcpy((void**)&d_vecA,vecA,SIZ*sizeof(float));
    cudaMemcpy((void**)&d_vecB,vecB,SIZ*sizeof(float));
    for(int i=0; i<SIZ; i++)
        vecA[i] = (float)(rand())/(float)(RAND_MAX);
    cudaMemcpy(d_vecA, vecA, SIZ*sizeof(float), cudaMemcpyHostToDevice)
}
int main () {
    float *vecA;    float *vecB;
    float *d_vecA;    float *d_vecB;
    int SIZ = (int)10240000;
    vecA = (float*)malloc(SIZ*sizeof(float));
    vecB = (float*)malloc(SIZ*sizeof(float));
    cudaMalloc((void**)&d_vecA, SIZ*sizeof(float));
    cudaMalloc((void**)&d_vecB, SIZ*sizeof(float));
    for(int i=0; i<SIZ; i++)
        vecA[i] = (float)(rand())/(float)(RAND_MAX);
    cudaMemcpy(d_vecA, vecA, SIZ*sizeof(float), cudaMemcpyHostToDevice);
    vecOpGPU<<<(int)(SIZ/512), 512>>>(SIZ, d_vecA, d_vecB);
int main () {
float *vecA;    float *vecB;
float *d_vecA;    float *d_vecB;
int SIZ = (int)10240000;
vecA = (float*)malloc(SIZ*sizeof(float));
vecB = (float*)malloc(SIZ*sizeof(float));
cudaMalloc((void**)&d_vecA, SIZ*sizeof(float));
cudaMalloc((void**)&d_vecB, SIZ*sizeof(float));
for(int i=0; i<SIZ; i++)
    vecA[i] = (float)(rand())/(float)(RAND_MAX);
cudaMemcpy(d_vecA, vecA, SIZ*sizeof(float), cudaMemcpyHostToDevice);
vecOpGPU<<<(int)(SIZ/512), 512>>>(SIZ, d_vecA, d_vecB);
cudaMemcpy(vecB, d_vecB, SIZ*sizeof(float), cudaMemcpyDeviceToHost);
}
void vecOpCPU(int N, float * vecIn, float * vecOut) {
    for(int i=0; i<N; i++)
        vecOut[i]=sin(vecIn[i])-cos(vecIn[i]);
    return;
}

__global__ void vecOpCPU(int N, float * vecIn, float * vecOut) {
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    if(i<N)
        vecOut[i]=sin(vecIn[i])-cos(vecIn[i]);
    return;
}
Code Walkthrough 2

[gpu-app@ln1%tianhe.vecOp]$ yhrun -N 1 -n 1 -p gpu_test ./vecOp
yhrun: job 1240529 queued and waiting for resources
yhrun: job 1240529 has been allocated resources

Start Benchmarking.. Vector Operation (10240000). Iteration: 10 times

CPU, 1 thread
Elapsed time : 4631628.6 us

CPU, OpenMP 12 threads
Elapsed time : 398803.5 us  Speedup (over 1 thread) : 11.6X

CudaGetErrorString: no error
GPU Elapsed time : 19810.2 us  Speedup (over 1 thread) : 233.8X and (over openmp) : 20.1X

Error number: 0
[gpu-app@ln1%tianhe.vecOp]$
Takeaways

- Threads are grouped into **blocks**
- **Blocks** are grouped into a grid
- **A kernel** is executed as a grid of **blocks of threads**
- **Each block** is distributed onto one SM(X) at execution time
Outline

- Evolution of GPU Computing
- Heterogeneous Computing
- CUDA Execution Model & Walkthrough of Hello World
- Walkthrough : 1D Stencil
Revisit Vector Operation

threadIdx == 21
Stencil Code (Radius=3)

threadIdx == 21
Stencil Code Continue

- Each thread processes one output element
- Input elements are read several times
  - With radius 3, each input element is read seven times
Implementing With Shared Memory

- Cache data in shared memory
  - Read $(\text{blockDim}.x + 2 \times \text{radius})$ input elements from global memory to shared memory
  - Compute $\text{blockDim}.x$ output elements
  - Write $\text{blockDim}.x$ output elements to global memory

- Each block needs a halo of $\text{radius}$ elements at each boundary
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
}
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

out[gindex] = result;
Data Race!

- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```c
... temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
for (int offset = -RADIUS; offset <= RADIUS; offset++)
    result += temp[lindex + offset];
...```

Store at temp[18]  
Skipped since threadIdx.x > RADIUS  
Load from temp[19]
__syncthreads()
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + radius;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }

  // Synchronize (ensure all the data is available)
  __syncthreads();
}
// Apply the stencil
int result = 0;
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
Takeaways

- **Shared memory** can be used to reduce global memory pressure

- In some cases, proper *synchronization* is needed to eliminate data hazards
CUDA

CONCEPTS

- Heterogeneous Computing
- Device Management
- Memory Management
- Threads & Indexing
- Blocks
- Shared Memory
- Synchronization
Summary

- APIs & architecture details may change, the concepts shall last
Questions?