CUDA PROGRAMMING

Yu-Hang Tang

June 23-26, 2015

CSRC, Beijing
WELCOME

- slides at /home/ytang/slides
  - posted after each day
- exercise at /home/ytang/exercise – make your own copy!
- solution at /home/ytang/solution
  - permission granted after each exercise

Online CUDA API documentation
http://docs.nvidia.com/cuda/index.html
DEMO - SELF ASSEMBLY

- 134,217,728 particles
- 12,400,000 steps
- 1024 GPUs
GPU = GRAPHICS PROCESSING UNIT

- Traditional GPUs
  - Fixed-function pipelines
  - Earliest consumer application of multi-core architecture

- General-purpose GPUs
  - Fully programmable
  - massively parallel
MOTIVATION
WHO USED MY POWER

• Frequency vs. Power

\[ P \propto V^2 f \]

• Higher voltages are necessary for higher frequency
• Rule of thumb: 2x frequency, 4x power consumption
WHO USED MY POWER

• ILP: Instruction Level Parallelism
  
  a = b + c;
d = c + a;
f = c + e;

• Hardware ILP extraction logic
  
  • Pipelining
  • Superscalar
  • Branch prediction / Speculative execution

ILP extraction is expensive!
WHO USED MY POWER

• Data transfer is expensive
  • Computation appears to be FREE
• Cache: a smaller but faster memory storing copies of data in frequently used main memory locations.

<table>
<thead>
<tr>
<th></th>
<th>64-bit DP FMA</th>
<th>256-bit On-chip SRAM</th>
<th>256-bit Off-chip DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Energy</strong></td>
<td>20 pJ</td>
<td>50 pJ</td>
<td>16 nJ</td>
</tr>
</tbody>
</table>

Cache is Cash
WHY GPU

- New Moore’s Law: parallelism increases exponentially
- Failure of Moore’s Law
  - Frequency wall
  - Power wall
  - ILP wall
- Green Computing
  - FLOPS/watt matters
  - Green500.org
    - Kepler GPUs dominate

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>GFLOPS/W</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L-CSC</td>
<td>5.3</td>
<td>ASUS ESC4000 FDR/G2S, Intel Xeon E5-2690v2 10C 3GHz, Infiniband FDR, AMD FirePro S9150</td>
</tr>
<tr>
<td>2</td>
<td>Suiren</td>
<td>4.9</td>
<td>ExaScaler 3U256SC Cluster, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, PEZY-SC</td>
</tr>
<tr>
<td>3</td>
<td>Tsubame-KFC</td>
<td>4.5</td>
<td>Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x</td>
</tr>
<tr>
<td>4</td>
<td>Storm1</td>
<td>4.0</td>
<td>Cray CS-Storm, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, Nvidia K40m</td>
</tr>
<tr>
<td>5</td>
<td>Wilkes</td>
<td>3.6</td>
<td>Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20</td>
</tr>
<tr>
<td>6</td>
<td>iDataPlex DX360M4</td>
<td>3.5</td>
<td>Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x</td>
</tr>
<tr>
<td>7</td>
<td>HA-PACS TCA</td>
<td>3.5</td>
<td>Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x</td>
</tr>
<tr>
<td>8</td>
<td>Cartesius Accelerator Island</td>
<td>3.5</td>
<td>Bullx B515 cluster, Intel Xeon E5-2450v2 8C 2.5GHz, InfiniBand 4x FDR, Nvidia K40m</td>
</tr>
<tr>
<td>9</td>
<td>Piz Daint</td>
<td>3.2</td>
<td>Xeon E5-2670 8C 2.600GHz, Aries interconnect, NVIDIA K20x</td>
</tr>
</tbody>
</table>
A QUICK FACT

- GPU vs. CPU performance comparison
MORE FACTS

- Programming GPU is less trivial
- "If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"

– Seymour Cray

Yu-Hang Tang & Karniadakis Group, Summer School on Parallel Computing, CSRC, Beijing
GPU PROS & CONS 知己知彼，百战不殆

- Data Parallel
- Intensive FP Arithmetic
- Fine-grained parallelism
- Task Parallel
- Thread Dependencies
- Serial work
- Coarse-grained parallelism
## HOW TO USE GPUs

### Language Extensions
- C
- C++
- Fortran
- ...

### Directives
- OpenACC
- OpenMP
- ...

### Libraries
- cuBLAS
- cuSPARSE
- cuFFT
- cuRAND
- ...

### Scripting
- PyCUDA
- MATLAB
- ...

Yu-Hang Tang & Karniadakis Group, Summer School on Parallel Computing, CSRC, Beijing  
June 23-26, 2015
MATLAB GPGPUARRAY

• gpuArray: Create arrays on GPU

```matlab
m = magic(64); % m is on CPU
M = gpuArray(m); % M is on GPU now
```

• GPU-enabled functions

```matlab
n = fft2(m); % FFT on CPU
N = fft2(M); % FFT on GPU
```

• Collect result

```matlab
L = gather(N); % transfer N back to CPU
find(abs(L - n) > 1e-9);
```

• Reference: http://www.mathworks.com/discovery/matlab-gpu.html
# CUDA C/C++ Overview

- Subset of C++ with CUDA-specific extensions

<table>
<thead>
<tr>
<th>Feature</th>
<th>Availability</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Built-in data types: char, int, float, etc.</td>
<td>Y</td>
<td>vector types: int2, float4...</td>
</tr>
<tr>
<td>Built-in operators</td>
<td>Y</td>
<td>including new/delete</td>
</tr>
<tr>
<td>Overloading</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Object-oriented programming</td>
<td>Y</td>
<td>Inheritance virtual methods</td>
</tr>
<tr>
<td>Templates</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>C standard library</td>
<td>Partial</td>
<td>printf, malloc, free supported</td>
</tr>
<tr>
<td>C++ standard library</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>C++11 extensions</td>
<td>Y</td>
<td>variadic template, lambda</td>
</tr>
</tbody>
</table>
EXAMPLE #0: HELLO WORLD

• Compilation command

    nvcc -arch=sm_35 hello.cu -o hello.x

• -arch: specifies target compute capability

    1.0 → 1.1 → 1.2 → 1.3 → 2.0 → 2.1
    → 3.0 → 3.5* → 5.0 → ...

#include <cstdio>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void hello_gpu() {
    printf( "Hello, world!", says the GPU.\n" );
}

void hello_cpu() {
    printf( "Hello, world!", says the CPU.\n" );
}

int main( int argc, char **argv )
{
    hello_cpu();
    hello_gpu<<< 1, 1>>>() ;
    cudaDeviceSynchronize();
    return 0;
#include <cstdio>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void hello_gpu()
{
    printf( "Hello, world!", says the GPU."
    );
}

void hello_cpu()
{
    printf( "Hello, world!", says the CPU."
    );
}

// host code entrance
int main( int argc, char **argv )
{
    hello_cpu();
    hello_gpu<<< 1, 1>>>();
    cudaDeviceSynchronize();
}

• NVCC compiler
  • a wrapper around a host compiler, e.g.
    • GCC
    • Intel Compiler
    • MSC
  • -arch: specify compute capability
  • -ccbin: specify host compiler (default to g++ on Linux)

• CUDA runtime
  • Automatic initialization
  • Asynchronous Execution
PROGRAM FLOW

- Hardware
  - CPU
  - GPU
  - RAM
  - GRAM

- Software
  - init
  - parallel work 1
  - parallel work 2
  - serial work 1
  - serial work 2
  - finalize
THREAD HIERARCHY

- To manage thousands of threads
  \textit{divide et impera}
- Block: a group of at most 1024 threads
  - Why: hardware limitation
  - Threads arranged in 1D/2D/3D
  - threads share resources within a block:
    - register, shared memory, etc.
    - synchronization available
- Grid: group of blocks
  - 1D/2D/3D arrangement
  - Practically unlimited number of blocks
  - No synchronization between blocks
Kernels

- Executed in parallel by many CUDA threads
  - begin with the __global__ qualifier

```c
// each thread will print once
__global__ void hello()
{
    printf( "Hello, world!\n", says the GPU.\n" );
}
```

- Need to configure parallelism when launching

```c
kernel<<<numBlocks,threadsPerBlock>>>(args);
```

- Each thread has an id defined in built-in threadIdx variable.
- Must be launched from host
FUNCTION QUALIFIER

• Kernels: __global__
• Device functions: __device__
  • callable from kernels
• Host functions: __host__
  • functions without a qualifier are default to be host functions
• A function can be both __device__ and __host__
  • No qualifier can be used together with __global__

```c
__inline__ __host__ __device__ double force( double x ) {
    return -0.5 * K * ( x - x0 );
}
```
PARALLELISM CONFIGURATION

- **struct dim3 { uint x,y,z; };**
  - CUDA built-in type for describing the thread configuration
- Built-in variables for each thread:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>threadIdx</td>
<td>thread index within the current block</td>
</tr>
<tr>
<td>blockIdx</td>
<td>block index within the current grid</td>
</tr>
<tr>
<td>blockDim</td>
<td>block size</td>
</tr>
<tr>
<td>gridDim</td>
<td>grid size, i.e. number of blocks in each dimension</td>
</tr>
</tbody>
</table>

- Launch configuration `<<<numBlocks, threadsPerBlock>>>`
  - `<<<uint,uint>>>`
  - `<<<dim3,dim3>>>`
GPU GLOBAL MEMORY

• GPU has its own on-board memory
  • Accessing CPU RAM is another story
• Allocatable from host/device
  • cudaError_t cudaMalloc ( void** devPtr, size_t size );
  • cudaError_t cudaFree ( void* devPtr );
  • device-side malloc/new/free/delete
• Accessible from device
  

```c
ptr[ index ] = value;
```

• Copiable from host
  • cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind );
  • cudaError_t cudaMemcpy ( void* devPtr, int value, size_t count );
\[ f(x) = \sin x \cdot \cos 7x \cdot e^x, x \in [0,1] \]
EXAMPLE 1B: AXPY

- AXPY = a \times x + y
  - a: scalar
  - x, y: N-by-1 vectors
EXAMPLE 2: HELLO WORLD 2D

```c
#include <cstdio>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void hello_gpu()
{
    printf( "Hello, world!\n", says GPU block (%d,%d) thread (%d,%d).\n", 
            blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y );
}

void hello_cpu()
{
    printf( "Hello, world!\n", says the CPU.\n" );
}

// host code entrance
int main( int argc, char **argv )
{
    hello_cpu();
    printf( "launching 2x2 blocks each containing 4 threads\n" );
    hello_gpu <<< dim3( 2, 2, 1 ), dim3( 4, 1, 1 ) >>>();
    cudaDeviceSynchronize();
    printf( "launching 2x2 blocks each containing 2x2 threads\n" );
    hello_gpu <<< dim3( 2, 2, 1 ), dim3( 2, 2, 1 ) >>>();
    cudaDeviceSynchronize();
    cudaDeviceSynchronize();
}
```
EXAMPLE 3A: 2D FUNCTION

- \( f(x, y) = \sin 5x \cdot \cos 16y \cdot e^x, \ x \in [0,1], y \in [0,1] \)
EXAMPLE 3B: IMAGE FILTERING

- 2D convolution

![Diagram showing 2D convolution process with examples of convolution matrices and resulting images.](image-url)
ATOMICs

• Race condition

__shared__ int sum;
int b = ...;
sum += b;

• Atomicity: a guarantee that the operation will be performed without interference from other threads.

• Performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory
  • modify = add, sub, exchange, etc...
  • Only atomicExch() and atomicAdd() for float values

__shared__ int sum;
int b = ...;
register r = sum;
r += b;
sum = r;

__shared__ int sum;
int b0 = ...;
register r0 = sum;
r0 += b0;

int b1 = ...;
register r1 = sum;
sum = r0;
r1 += b1;
sum = r1;
EXAMPLE 4: PARALLEL REDUCTION

- Reduction: a summary of data
  - summary = summation, mean, max, min, etc.
- Parallel summation: \( S_n = \sum_{i=0}^{n-1} a_i \)
  - The serial way: for(int i = 0 ; i < n ; i++) sum += a[i];
  - How to reduce in parallel?
Thank you for coming to this workshop!

ACKNOWLEDGEMENT

Y.H.T. appreciate invitation from Professor Wei Cai and support from CSRC.