GPGPU
Basic CUDA workshop

Prof. HONDA Kiyoshi
hondak@isc.chubu.ac.jp

Int’l Digital Earth Applied Science Research Center
Chubu University
Outlines 講習の概要

- Prerequisite
- Setting Up Environment
  - Your Machine or Login to GPGPU machine
    - 自分のPCのセットまたはGPGPUへのログイン
- Architecture、構造
- Getting Started、はじめましょう
- Programming CUDA
- Debugging
- Fine-Tuning
Outlines

- Setting Up Your Machine or Login to GPGPU machine
- Architecture
- Getting Started
- Programming CUDA
- Debugging
- Fine-Tuning
Prerequisite (知ってますね)

- I expected that everyone already knew C/C++, no need to be super, although.
  - Functions
    - `void myfunction(){
        printf("something\n");
    }`
  - Loops
    - `for(i=0;i<n;i++) printf("%d\n", i);`
  - Address and Pointer
    - `float *pt = &x;`
  - Arrays
    - `array_c[i] = array_a[i] + array_b[i];`
  - Dynamic Memory Allocation
    - `float *array;`  
    - `array = (float *) malloc(n*sizeof(float));`
  - Process and Thread
Setting Up Environment
login in to tesla

- use putty to login to tesla2. Use the account provided
  - Hostname: tesla2  isc  chubu  ac  jp  (connect by . !)
  - Just warm up:
  - Create **hello.c** either by...
    - An editor (**nano**, **vim**, **emacs**, etc.)
- Goto ex01/templates directory
- Type the following classic code
  - **gcc hello.c, then, ./a.out**
  - **gcc –o hello hello.c, then, ./hello**
- You should get something like
  
  ```
  #include <stdio.h>
  int main(){
      printf("Hello, world!!!\n");
      return 0;
  }
  ```
  
  ```
  [kaewgb@maeka ~]$ ./hello
  Hello, world!!!
  ```
Maeka Cluster

- Log in to a GPU machine using the account provided
  - Hostname: xxxx.xxxx.xxxx.xxxx
  - Change the password with command `passwd`
  - Goto `ex01/templates` directory (cd `ex01`, cd `templates`)
- Create `hello.c` either by...
  - An editor in the cluster (`nano`, `vi`, etc.) or edit on your PC then ftp
- Type the following classic code
- Compile and link by `gcc -o hello hello.c` or just `gcc hello.c`
- `./hello` or `a.out`
- You should get something like

```c
#include <stdio.h>

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

```
[kaewgb@maeka ~]$ ./hello
Hello, world!!!
```
Basic Linux Command

- See manual
  - `man command`, `man --section command` (section: 1, 2, 3, 4, 5,..)

- List directory contents
  - `ls`, `ls -l`, `ls -la`

- Change directory
  - `cd`
  - One up, `cd ..`
  - Just “cd” to go back your home directory
  - Show current directory `pwd`
  - “pushd directory” to go, “popd” to come back, “dirs” to see directory stack
  - Home directory `~`

- copy files; `cp`, `-r` recursive, `-p` preserve time

- rename or move files: `mv oldFileName newFileName[target directory]`

- Remove file `rm`
Useful Materials 便利な資料

- Goto
  - [http://tesla2.isc.chubu.ac.jp/tutorial](http://tesla2.isc.chubu.ac.jp/tutorial)
  - Explore materials and documents、探検してください。
- Go to [http://tesla2.isc.chubu.ac.jp/tutorial/links.html](http://tesla2.isc.chubu.ac.jp/tutorial/links.html)
  - Goto CUDA Zone
    - Explore CUDA Zone, Toolkit
Installation  
– if you have your own NVIDIA GPU

- The latest documents are on tesla ( links.html) ,  
  but better to refer to CUDA Zone
- Go to official CUDA Toolkit Download page  
  ( Cuda Zone -> Toolkit ->Download)  
  http://developer.nvidia.com/  
  Follow the instruction: CUDA_C_Getting_Started_Linux[Windows].pdf
- Download and install CUDA driver  
  only if your machine has NVIDIA GPU
- Download and install CUDA toolkit (SDK)  
  Linux (.run): chmod +x, then run by ./
- Download and install GPU Computing SDK Code Samples  
  cd ~/NVIDIA_GPU_Computing_SDK/C  
  make  
  Sample programs are in ~/NVIDIA_GPU_Computing_SDK/C/bin/linux/ release
  This directory is there already under your directory
IDE Syntax Highlighting

- Visual Studio (Windows)
  - Refer to C:\NVIDIA GPU Computing SDK\C\doc\syntax_highlighting\visual_studio_8\readme.txt
  - May use Visual Assist X for Auto-Completion

- Code::Blocks (Windows/Linux)
  - Open C:\Program Files\CodeBlocks\share\CodeBlocks\lexers\lexer_cpp.xml
  - Add .cu (and probably .cuh if you use) to filemasks attribute in Lexer tag
    - filemasks="*.c,*.cpp,*.cc,*.cxx,*.h,*.hpp,*.hh,*.hxx,*.inl,*.cu,*.cuh"
  - My Lexer
    - http://kaewgb.com/cuda/lexer_cpp.xml
Hardware Architecture

...in general
NVIDIA GPU Architecture

- Highly parallel, multithreaded, many-core processor

14x32 = 448 (SP)
NVIDIA GPU Architecture

- Highly parallel, multithreaded, many-core processor

![Diagram of NVIDIA GPU Architecture](image-url)

- 14x32 = 448 (SP)

- Host Memory (Allocate by malloc)
- PC Main Memory
- Off-Chip
- 4GB Video Memory (Global Memory)
- On-Chip
- A Streaming Multiprocessor (SM)
- 32 Streaming Processors (SP)
- 16KB Shared Memory
- 16k Register Memory

NVIDIA Tesla M2050
### Obtaining your GPU spec by `deviceQuery`

#### Device 0: "Tesla C2070"  
**CUDA Driver Version / Runtime Version**: 4.0 / 4.0  
**CUDA Capability Major/Minor version number**: 2.0  
**Total amount of global memory**: 5375 MBytes (5636292608 bytes)  

<table>
<thead>
<tr>
<th>Multiprocessors x CUDA Cores/MP</th>
<th>448 CUDA Cores</th>
</tr>
</thead>
</table>

**GPU Clock Speed**: 1.15 GHz  
**Memory Clock rate**: 1494.00 Mhz  
**Memory Bus Width**: 384-bit  
**L2 Cache Size**: 786432 bytes  
**Max Texture Dimension Size (x,y,z)**:  
1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)  
**Max Layered Texture Size (dim) x layers**:  
1D=(16384) x 2048, 2D=(16384,16384) x 2048  
**Total amount of constant memory**: 65536 bytes  
**Total amount of shared memory per block**: 49152 bytes  
**Total number of registers available per block**: 32768  
**Warp size**: 32  
**Maximum number of threads per block**: 1024  
**Maximum sizes of each dimension of a block**: 1024 x 1024 x 64  
**Maximum sizes of each dimension of a grid**: 65535 x 65535 x 65535  
**Maximum memory pitch**: 2147483647 bytes  
**Texture alignment**: 512 bytes  
**Concurrent copy and execution**: Yes with 2 copies  
**Run time limit on kernels**: Yes  
**Integrated GPU sharing Host Memory**: No  
**Support host page-locked memory mapping**: Yes  
**Concurrent kernel execution**: Yes  
**Alignment requirement for Surfaces**: Yes  
**Device has ECC support enabled**: Yes  
**Device is using TCC driver mode**: No  
**Device supports Unified Addressing (UVA)**: Yes  
**Device PCI Bus ID / PCI location ID**: 3 / 0  
**Compute Mode**: Default (multiple host threads can use :: `cudaSetDevice()` with device simultaneously).  

To obtain info from your program, use `cudaGetDeviceProperties()`.
Let’s begin!

How to get started.
Drill #0: Code Samples

- Try running some test program from *GPU Computing SDK Code Samples* in your directory `~/NVIDIA_GPU_Computing_SDK/C/bin/linux/release/`
- *Or Goto*
- `/usr/local/cuda/NVIDIA_GPU_Computing_SDK/C/bin/linux/release`
- Many programs use X Windows so if you’re not on Linux console, you may not be able to try. Just skip them.
  - ここではXwindow用のプログラムがたくさんあるので、そのときはスキップしてください。
- This directory is a symbolic link – run from your home directory
  - The actual folder is not under your directory
  - Some test programs try to write files at the running location
    - Might have permission problems
  - このディレクトリはシンボリックリンクなので、自分のホームディレクトリから実行してください。
  - パスは .profile で通してあります。
Drill #0: Code Samples

- Try running some test program from **GPU Computing SDK Code Samples** in your directory `~/NVIDIA_GPU_Computing_SDK/C/bin/linux/release/`

  - Many programs use X Windows so if you’re not on Linux console, you may not be able to try. Just skip them.
  - Path is already set for you to this directory – see `.profile`
  - otherwise
  - Run by prefix the whole path like this

```bash
kaewgb@maeka ~]$ NVIDIA_GPU_Computing_SDK/C/bin/linux/release/bandwidthTest
```

- Some test programs try to write files at the running location
  - Might have permission problems
- If not
  - `ln -s /usr/local/cuda/NVIDIA_GPU_Computing_SDK/C/bin/linux/release .`
Useful Sample Programs

- deviceQuery
  - Driver version
  - SDK (toolkit) version
  - Compute Capability
  - Hardware/Software Specifications
- bandwidthTest
  - Peak Bandwidth (Practical/Experimental)
- etc.
  - Many more in newer SDKs
Compute Capabilities

- Tells how much a GPU is capable of
  - 1.0, 1.1, 1.2, 1.3, 2.0
  - 1.3 – supports double-precision floating point
  - 2.0 – Fermi
    - Global Memory cache
    - Native printf() : cuPrintf() : see simplePrintf example
- Learn more in CUDA C Programming Guide
  - Appendix A: CUDA-Enabled GPUs
  - Appendix G: Compute Capabilities
A Closer Look at the Specifications

... 

Multiprocessors x Cores/MP = Cores: 14 (MP) x 32 (Cores/MP) = 448 (Cores)

Total amount of constant memory: 65536 bytes

Total amount of shared memory per block: 49152 bytes

Total number of registers available per block: 32768

Warp size: 32

Maximum number of threads per block: 1024

Maximum sizes of each dimension of a block: 1024 x 1024 x 64

Maximum sizes of each dimension of a grid: 65535 x 65535 x 1

...What is this all about!?!?!?
Programming CUDA

Terminology, Memory Hierarchy, APIs, etc.
Kernels  カーネル

- A Function that is executed on GPU.
- Many threads executed the same instructions together.
  - GPU上で多数のスレッドとして同時実行される関数です。
- SPMD (Single Program Multiple Data) or
- SIMD (Single Instruction Multiple Data)
Multithreaded

- Massive number of threads
  - GPU’s threads are of much lighter weight than CPU’s.
    - Take very few cycle to generate and schedule
  - GPU上のスレッドはとても軽く数百万以上のスレッドが走ります。

- Compute each data element simultaneously
  - 一つのスレッドが一つのデータを計算します

<table>
<thead>
<tr>
<th>Thread#0</th>
<th>Thread#1</th>
<th>Thread#2</th>
<th>Thread#3</th>
<th>Thread#4</th>
<th>Thread#5</th>
<th>Thread#6</th>
<th>Thread#7</th>
<th>...</th>
<th>Thread#n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₀</td>
<td>a₁</td>
<td>a₂</td>
<td>a₃</td>
<td>a₄</td>
<td>a₅</td>
<td>a₆</td>
<td>a₇</td>
<td>...</td>
<td>aₙ</td>
</tr>
<tr>
<td>b₀</td>
<td>b₁</td>
<td>b₂</td>
<td>b₃</td>
<td>b₄</td>
<td>b₅</td>
<td>b₆</td>
<td>b₇</td>
<td>...</td>
<td>bₙ</td>
</tr>
</tbody>
</table>
Multithreaded

- Massive number of threads
  - GPU’s threads are of much lighter weight than CPU’s.
    - Take very few cycle to generate and schedule
    - GPU上のスレッドはとても軽く数百万以上のスレッドが走ります。

- Compute each data element simultaneously
  - 一つのスレッドが一つのデータを計算します

<table>
<thead>
<tr>
<th>Thread#0</th>
<th>Thread#1</th>
<th>Thread#2</th>
<th>Thread#3</th>
<th>Thread#4</th>
<th>Thread#5</th>
<th>Thread#6</th>
<th>Thread#7</th>
<th>...</th>
<th>Thread#n</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₀</td>
<td>a₁</td>
<td>a₂</td>
<td>a₃</td>
<td>a₄</td>
<td>a₅</td>
<td>a₆</td>
<td>a₇</td>
<td>...</td>
<td>aₙ</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>b₀</td>
<td>b₁</td>
<td>b₂</td>
<td>b₃</td>
<td>b₄</td>
<td>b₅</td>
<td>b₆</td>
<td>b₇</td>
<td>...</td>
<td>bₙ</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c₀</td>
<td>c₁</td>
<td>c₂</td>
<td>c₃</td>
<td>c₄</td>
<td>c₅</td>
<td>c₆</td>
<td>c₇</td>
<td>...</td>
<td>cₙ</td>
</tr>
</tbody>
</table>

eg. n=1 billion
Threads can be structured as 2Ds
スレッドは2次元に構成することができます
Dimensional Structure

- **Threads** are grouped into **Blocks**
- スレッドはブロックにグループ化され、複数のSMPにそれぞれ送り込まれます.

<table>
<thead>
<tr>
<th></th>
<th>t₀₀</th>
<th>t₀₁</th>
<th>t₀₂</th>
<th>t₀₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁₀</td>
<td>t₁₁</td>
<td>t₁₂</td>
<td>t₁₃</td>
<td></td>
</tr>
<tr>
<td>t₂₀</td>
<td>t₂₁</td>
<td>t₂₂</td>
<td>t₂₃</td>
<td></td>
</tr>
<tr>
<td>t₃₀</td>
<td>t₃₁</td>
<td>t₃₂</td>
<td>t₃₃</td>
<td></td>
</tr>
</tbody>
</table>

threads
Dimensional Structure

- **Threads** are grouped into **Blocks**
- A group of **Blocks** is executed simultaneously on each **Core (Streaming Multiprocessor)**

_threads_

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>t₀,₀</td>
<td>t₀,₁</td>
<td>t₀,₀</td>
<td>t₀,₁</td>
<td></td>
</tr>
<tr>
<td>t₁,₀</td>
<td>t₁,₁</td>
<td>t₁,₀</td>
<td>t₁,₁</td>
<td></td>
</tr>
<tr>
<td>t₀,₀</td>
<td>t₀,₁</td>
<td>t₀,₀</td>
<td>t₀,₁</td>
<td></td>
</tr>
<tr>
<td>t₁,₀</td>
<td>t₁,₁</td>
<td>t₁,₀</td>
<td>t₁,₁</td>
<td></td>
</tr>
</tbody>
</table>
Dimensional Structure

- **Threads** are grouped into **Blocks** for management reasons.
- Each **Block** is executed simultaneously on each **Core** (Streaming Multiprocessor)

![Diagram of dimensional structure with threads and blocks]
Dimensional Structure (cont.)

Host

Device

Kernel 1

Grid 1

Block (0, 0)

Block (1, 0)

Block (0, 1)

Block (1, 1)

Grid 2

Block (1, 1)

Block (0, 1)

Block (1, 0)

Block (2, 0)

Block (3, 0)

Block (0, 0)

Block (1, 0)

Block (2, 0)

Block (3, 0)

Block (0, 1)

Block (1, 1)

Block (2, 1)

Block (3, 1)

Thread (0, 0, 0)

Thread (1, 0, 0)

Thread (2, 0, 0)

Thread (3, 0, 0)

Thread (0, 1, 0)

Thread (1, 1, 0)

Thread (2, 1, 0)

Thread (3, 1, 0)

Courtesy: NVIDIA
Automatic Scalability
Heterogeneous Computing

Asynchronous launch

Each kernel has independent - #blocks, #threads
Normal Program Flow

Input Data

Allocated for output

33
Normal Program Flow

- Allocate device memory (CPU = host, GPU=device)
Normal Program Flow

- Allocate device memory (CPU = host, GPU=device)
- Transfer input into device memory
Normal Program Flow

- Allocate device memory (CPU = host, GPU=device)
- Transfer input into device memory
- Compute (Call *Kernel* Function)
Normal Program Flow

- Allocate device memory (CPU = host, GPU=device)
- Transfer input into device memory
- Compute (Call *Kernel* Function)
Normal Program Flow

- Allocate device memory (CPU = host, GPU=device)
- Transfer input into device memory
- Compute (Call *Kernel* Function)
- Transfer output back to host memory
Normal Program Flow

- Allocate device memory (CPU = host, GPU = device)
- Transfer input into device memory
- Compute (Call *Kernel* Function)
- Transfer output back to host memory
- Free unused memory
Checkpoint

- Thread
- Block
- Grid
- Kernel
Drill #0.50: HelloCPU

- // hello.c: hello world program on CPU to show hello 4 times.
- #include <stdio.h>
- void hello()
  
  { 
    printf("Hello, I am hello on cpu\n");
  } 
- int main()
  
  { 
    int i;
    for( i=0; i<4; i++ )
      hello();
    return 0;
  }
- Go to ~/ex01/templates directory
- Modify your previous hello.c
- Compile and run gcc hello.c and ./a.out
Drill #0.51: HelloGPU

```c
#include<stdio.h>

__global__ void hellogpu()
{
    printf("Hello, I am hello thread on GPU\n");
}

int main()
{
    hellogpu <<< 1, 4 >>> ();
    cudaThreadExit();
    return 0;
}

cp hello.c hello.cu, edit, compile by
nvcc --arch=sm_20 hello.cu, then ./a.out
Wow, we do not need “for loop”!
```
Drill #0.52: HelloGPU – Who am I?

- `#include <stdio.h>`

- `__global__ void hellogpu()`
  ```c
  {
  printf("Hello I am a thread (%3d,%3d)n on GPU\n", threadIdx.x, threadIdx.y);
  }
  ```

- Change hellogpu kernel to see thread index.
  ```c
  hellpgpu カーネルをスレッドインデックスが見えるように変更
  ```
  ```c
  Humm, each thread can know “Who am I?”
  ```
Drill #0.53: HelloGPU What is <<<1,4>>> ?

- #include <stdio.h>
- __global__ void hellogpu()
  { 
    printf("Hello, I have blockIdx(%d %d), threadIdx (%3d,%3d)\n", blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y);
  }
- int main()
  {
    dim3 blockDim(4,1,1);
    dim3 blockDim(4,1,1);
    hellogpu<<<gridDim, blockDim >>>();
  }
- ..... 
- blockDim (4,1,1) -> block has 1-d structure, it contains 4 threads, threadIdx.x from 0 to 3
  - blockDim (4,1,1) は1次元で、threadIdx.x は0から3に変化します。
  - Change gridDim to (2,1,1): now you have two blocks ( 4 threads each) to distribute blocks to two multi streaming processors.
    - gridDim(2,1,1)にしてください。これで2ブロックになり（1ブロックは4スレッド）、各ブロックはMSPにそれぞれ送込まれます。
  - Change blockDim(2,2,1), now block has 2-d structure threadIdx.x and y is from 0 to 1, still 4 threads –
    - blockDim(2,2,1)に変更、これでスレッドのインデックスが2次元になりました。
- Recall grid and block structure
Drill #0.6: vecset.cu

To set an integer series 0,1,2,3,4,... N-1（整数の数列を生成）

- #define N 32
- int main()
- {
  // Allocate memory for data on host
  int *h_series;
  h_series = (int *) malloc( N * sizeof(int) );

- #if defined(CPU)
  for( int i=0; i<N; i++)
    h_series[i] = i;
- #else
  45
Drill #0.61: vecset.cu : set 1d vector

- Allocate memory on device (GPU)
  
  int *d_series;

- cudaMalloc( &d_series, N*sizeof(int));

- // Set grid and block dimension
  
  dim3 gridDim(2);

- dim3 blockDim(N/2,1,1);

- // launch kernel
  
  vecset<<<gridDim,blockDim>>>( d_series );

- // Copy result from Device (GPU) -> Host (CPU)

  cudaMemcpy( h_series, d_series, N*sizeof(int), cudaMemcpyDeviceToHost );

- #endif

- // List the result

  for(int i=0; i<N;i++)
  
    printf("%d: %5d\n", i, h_series[i]);
Drill #0.62: vecset.cu – kernel function

- `#include<stdio.h>`
- `__global__ void vecset(int *d_series)`

```c
{
    // calculate index of data to operate
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    // set a value to the data this thread is responsible for
    d_series[i] = i;
}
```

- `nvcc vecadd.cu, then ./a.out`
  - コンパイルして実行
- `Explore the source code`
  - ソースコードを探検
- `Change code to set odd numbers` 奇数列にするよう変更

47
Vector Addition: Host

```c
float a[MAX], b[MAX], c[MAX];

int main(){
...

  float *d_a, *d_b, *d_c;
  cudaMalloc((void **) &d_a, MAX*sizeof(float));
  cudaMalloc((void **) &d_b, MAX*sizeof(float));
  cudaMalloc((void **) &d_c, MAX*sizeof(float));

  cudaMemcpy(d_a, a, MAX*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, MAX*sizeof(float), cudaMemcpyHostToDevice);

  dim3 dimBlock(32);
  dim3 dimGrid(MAX/32);

  vecadd<<<dimGrid, dimBlock>>>(d_a, d_b, d_c);

  cudaMemcpy(c, d_c, MAX*sizeof(float), cudaMemcpyDeviceToHost);

  return 0;
}
```

Device Memory Allocation

Copy input to Device Memory

Launch Kernel

Copy output back
API List (Part I)

- `cudaError_t cudaMalloc (void **devPtr, size_t size)
  - Allocate memory on device

- `cudaError_t cudaMemcpy (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  - kind: specify the host->device or device->host

- `cudaError_t cudaFree (void *devPtr)

- `cudaError_t cudaGetSymbolAddress (void **devPtr, const char *symbol)

- cudaError_t tells execution result

- Checking errors
  - if(cudaMalloc((void **) &d_a, 4096)!=cudaSuccess)
    - ...

- Refer to **Cuda Reference Manual** for more details

- Find these API from reference linked from links.html

49
dim3

- An integer vector type based on `uint3`
- Is used to specify dimension (block/grid)
- Any component left unspecified is initialized to 1

**Syntax**
- `dim3 <name>(x[, y, z])`
  - `[]` = optional

**Examples**
- `dim3 abc(1, 2, 3);`
- `dim3 haha(13);`
  - `= dim3 haha(13,1,1);`

**Check dim3.cu, dim3.cu を見るとよくわかります。**
- `~/ex01/templates/dim3.cu`
Kernel Call Syntax

- **KernelName** <<<#blocks per grid, #threads per block>>> (parameter list)
- #blocks per grid & #threads per block can be..
  - dim3 variables
    - dim3 gridDim(2,2);
    - dim3 blockDim(8,8,8);
    - vecadd<<<gridDim, blockDim>>>(d_a, d_b, d_c);
  - A natural number (if only 1 dimension)
    - vecadd<<<4, 512>>>(d_a, d_b, d_c);
  - Of course you can mix them
    - vecadd<<<4, blockDim>>>(d_a, d_b, d_c);
Kernel Code

- Standard C
- Can use standard Math library (math.h)
- Can’t use other
  - stdio.h
    - This means it can’t print anything out
    - Except we’re on Fermi.. and we are!!!
  - stdlib.h
- Syntax
  - __global__ void <name>(<parameter list>)
## Function Type Qualifiers

<table>
<thead>
<tr>
<th>Qualifiers</th>
<th>Executed on</th>
<th>Callable from</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong></td>
<td>Device</td>
<td>Device</td>
</tr>
<tr>
<td><strong>global</strong></td>
<td>Device</td>
<td>Host</td>
</tr>
<tr>
<td><strong>host</strong></td>
<td>Host</td>
<td>Host</td>
</tr>
<tr>
<td>Both <strong>host</strong> and <strong>device</strong></td>
<td>Host/Device</td>
<td>Host/Device</td>
</tr>
</tbody>
</table>

53
Built-in Variables

- **Hardware Registers**
  - threadIdx (thread index)
    - .x, .y, .z
  - blockDim (block dimension)
    - Number of threads in each dimension
      - .x, .y, .z
  - blockIdx (block index)
    - .x, .y
  - gridDim (grid dimension)
    - Number of blocks in each dimension
      - .x, .y
- **warpSize** (ウオープサイズ: SIMDとして一緒に実行されるスレッド数)
  - Number of threads per warp
vector addition: device

```c
__global__ void vecadd (float *d_a, float *d_b, float *d_c)
{
    int i;
    i = blockIdx.x * blockDim.x + threadIdx.x;

    d_c[i] = d_a[i] + d_b[i];
}
```

<table>
<thead>
<tr>
<th>thread</th>
<th>blockIdx.x:0</th>
<th>blockIdx.x:1</th>
</tr>
</thead>
</table>

blockDim.x = 2
Compiling

- nvcc
  - Just like gcc
- Kernels must be in .cu files
- `nvcc -o vecadd vecadd.cu`
- Can link with object files compiled from gcc
  - `gcc -c main.c`
    - `nvcc -o vecadd main.o vecadd.cu`
    - Or
      - `nvcc -c vecadd.cu`
      - `nvcc -o vecadd main.o vecadd.o`
- Specify GPU Architecture
  - `nvcc -arch=sm_20 -o vecadd vecadd.cu` -> for Fermi
    - If want to use double precision at least; `-arch=sm_13`
    - If want to use printf in kernels, `-arch=sm_20`
Drill #1: Vector Addition

- Goto ex01/templates
- Write a 1-d vector addition Program
  - Fixed size, 2048 elements
  - `int`
  - Find `C = A+B`
  - Try compiling and running by yourself.
- Use a template “templates/vecadd.cu”
- Templates source code may include code for CPU
  - `#if defined CPU`
  - Compile with –DCPU option
- `timer_init()`, and `timer_diff(“comment”)`, are to know the speed
  - Note: kernel launch is asynchronous, thus you may have to call cudaThreadSynchronize() to wait for all threads to be completed. (cudaMemCpy automatically synchronizes)
  - 注意: 時間計測用です。timer_diff をところどころに入れられます。
  - 注意: kernel は asynchronous です。cudaThreadSynchronize() でカーネルの終了を待ってから timer_diff()。（cudaMemCpyはその前に自動的にsyncします。）
- All solutions for all tasks are in “solutions/”, but try by yourself first...
APIs so far...

- Refer to CUDA C Reference Manual for the usage

Memory APIs
- cudaMalloc
- cudaFree
- cudaMemcpy
- cudaGetSymbolAddress

Synchronization
- syncthreads()
  - Block level
- cudaThreadSynchronize()
  - Grid level
Multidimensional Arrays

- are actually stored in 1 dimension
- Array [4][4]
- Array[16]
Drill #2: Matrix Addition

- Write a matrix addition Program
  - Fixed size
    - 2048x2048
  - int
  - Find C = A+B
- Use my template “templates/matadd.cu”
- Change data type to float and double
  - typedef double FLOAT, typedef float FLOAT, typedef int FLOAT
  - FLOAT *a
  - sizeof(FLOAT)
  - If you want to use double should be arch=sm_13 or sm_20
  - See vecaddf.cu in solutions directory.
APIs

- `cudaError_t cudaMallocPitch (void **devPtr, size_t *pitch, size_t width, size_t height)`
  - allocate memory by padding from line to line so that the access to global memory can be performed by coalescing.
  - グローバルメモリに高速でアクセス（coalescing）できるようにパディングしてアロケートします。

- `cudaError_t cudaMemcpy2D (void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)`
  - Copy memory block which includes padding.

- See example ~/ex01/solutions/matadd2.cu
Drill #3: Matrix Multiplication

- Write a Matrix Multiplication Program
  - Fixed size
    - 2048x2048
  - Find C = AB
- Use my template “templates/matmul.cu”

- Compare the calculation Speed between CPU and GPU
  - Probably 2048*2048 is too big for CPU, then reduce size to 1024*1024
  - Calculate and compare GFLOPS by CPU and GPU
Fine-Tuning
Overall Optimization Strategies

- Maximize parallel execution
  - Exposing data parallelism in algorithms
  - Overlap memory access with computation
  - Keep the hardware busy
    - Reduce Divergence

- Maximize memory bandwidth
  - Access data efficiently
  - Use on-chip memory as a buffer

- Maximize instruction throughput
  - Loop unrolling
Memory Space

- **Off-chip (Slow)**
  - Global
  - Local
  - Texture (cached)

- **On-chip (Fast)**
  - Registers
  - Shared Memory
  - Constant Memory

http://i cmpnet.com/ddj/images/article/2008/0806/080604cuda4_f1.gif
## Scopes

<table>
<thead>
<tr>
<th>Variable Declaration</th>
<th>Memory</th>
<th>Space</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic variables other than arrays</td>
<td>Register</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td>Automatic array variable</td>
<td>Global</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>Shared</td>
<td>Block</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>Global</td>
<td>Grid</td>
<td>Application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstVar;</td>
<td>Constant</td>
<td>Grid</td>
<td>Application</td>
</tr>
</tbody>
</table>
Example

int b;
__constant__ char c;
__device__ float d;

__global__ void myKernel(){
    int a, array[10];
    __shared__ float temp[20];
}

__device__ float square(float num){
    return num*num;
}

67
Warps

- 32-thread unit
- Only 1 warp will be actually executed by the hardware at any point in time
- <more on the whiteboard>
- Zero-overhead scheduling
- Long latency operations
Performance Evaluation

- Speedup
- Comparing actual/max. theoretical value
  - Memory Bandwidth
  - Instruction Bandwidth
  - Peak TFLOPS

69
Shared Memory Example

- `__shared__ int temp[16][16];`
- `int j = blockIdx.x * blockDim.x + threadIdx.x`
- `int i = blockIdx.y * blockDim.y + threadIdx.y`

- `temp[i][j] = array_global[i][j];`
- `__syncthreads();`
- `...`

Each thread can read value of `temp[i][j]` independently

E.g. 1x3 filter: Each thread copy only 1 data from global memory to shared memory first. Then, for example thread #((i,j-1) can read `temp[i][j-2],temp[i][j-1], temp[i][j]` much faster than accessing to global memory independently.
Drill #3.9: Tune up Matrix Multiplication (using register mem)

- Edit your matrix multiplication program
  - $c[\text{row}*N+\text{col}] = 0;$
  - for( $k=0; k<N; k++$ )
    - $c[\text{row}*N+\text{col}] += a[\text{row}*N+k] \times b[ k*N + \text{col }];$
  - To reduce access to global memory using register memory.
    - int val = 0;  // val is allocated in on-chip register memory
      for( $k=0; k<N; k++$ )
        - val += $a[\text{row}*N+k] \times b[ k*N + \text{col }];$
        - $c[\text{row}*N+\text{col}] = \text{val};$

- See how much you could accelerate
Drill #4: Tune up
Matrix Multiplication (using shared memory)

- Write a matrix multiplication program
  - Use template “/templates/matmul3.cu”
  - Use shared memory of size 16x16
  - Divide matrix into tiles of size 16x16
  - Do multiple rounds
  - Each round
    - Load a tile of A into shared mem
    - Load a tile of B into shared mem
    - Accumulate the results in C
Drill #4: Matrix Multiplication (shared)

- Write a matrix multiplication program
  - Use template "/templates/matmul3.cu"
  - Use shared memory of size 16x16
  - Divide matrix into tiles of size 16x16
  - Do multiple rounds
  - Each round
    - Load a tile of A into shared mem
    - Load a tile of B into shared mem
    - Accumulate the results in C
Drill #4: Matrix Multiplication (shared)

- Write a matrix multiplication program
  - Use template “/templates/matmul3.cu”
  - Use shared memory of size 16x16
  - Divide matrix into tiles of size 16x16
  - Do multiple rounds
  - Each round
    - Load a tile of A into shared mem
    - Load a tile of B into shared mem
    - Accumulate the results in C

A tile of size 16x16
Drill #4: Matrix Multiplication (shared)

- Write a matrix multiplication program
  - Use template “/templates/matmul3.cu”
  - Use shared memory of size 16x16
  - Divide matrix into tiles of size 16x16
  - Do multiple rounds
  - Each round
    - Load a tile of A into shared mem
    - Load a tile of B into shared mem
    - Accumulate the results in C
Drill #4: Matrix Multiplication (shared)

- **Wrong results?** (highlight to see)
  - This kernel needs 2 __syncthreads()
    - Why?
    - Ask me next time if you can’t figure out 😊
Useful Compiler Options

- `--ptxas-options=-v`
  - Report usage of:
    - Registers
    - Shared memory
    - Constant memory

- `-maxrregcount n`
  - Limits maximum number of register to $n$
  - For example,
    - `-maxrregcount 32`
**Drill #5: Range Matching (1/2)**

- Calculate \( C = \) the element in \( B \) that has nearest value to each element in \( A \) in each index range

<table>
<thead>
<tr>
<th>Index-</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>87</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>37</td>
<td>65</td>
<td>43</td>
</tr>
<tr>
<td>2</td>
<td>61</td>
<td>3</td>
<td>65</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>43</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>21</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>43</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

**Range = 4**

*If there are more than one nearest match, choose what comes first (lowest index)*
Drill #5: RangeMatching (2/2)

- Calculate Range Matching
  - Fixed size = 1024 (if works, try to make it work for 1000000)
  - Use template “templates/rangematch.cu”
  - Use range = 512
  - Use shared memory

- Wrong results?
  - Check if it loads data to shared memory correctly
  - Any miscalculated index?
  - Did you understand the task correctly?

- Still wrong? Maybe you forgot to... (highlight to see)
  - Use __syncthreads

- Wrong again?
  - Ask me 😊
Drill #6: Sum Reduction

- Find the sum of all elements in an array.
- Don’t really have to implement yet.
  - 3 drills are too much for one week
- Just some riddle for fun.
- How to assign jobs to each thread?
  - No 1<->1 corresponding between data<-> thread like previous drills.
- Hints (highlight to see)
Drill #6: Sum Reduction (cont)

- In case you have already done thinking and want to try
- Find summation of all elements in an array
  - Fixed size: 1024
  - Use the given template “templates/sumreduce.cu”

Solution

- I only gave one simple solution “solutions/sumreduce.cu”
- Will explain in the next workshop (if we have time)

Wrong results? Maybe you forgot to... (highlight to see)

- Use __syncthreads
Debugging
Debugging

- cuda-gdb
  - Extended from GNU gdb
  - Doesn’t work well when kernel code diverges (branches)

- printf
  - Now you can use printf in a kernel (compute capability >2, Fermi), if GTX higher than GTX465
  - Compile with arch=sm_20 or sm_21

  - Maybe the best and most convenient way
  - cudaMemcpyFromSymbol(<symbol name>)
Visual Studio Integration

- ...to be continued...
Tutorials

- Chapter 1-7 of online textbook of a course at Illinois
  http://courses.engr.illinois.edu/ece498/al/Syllabus.html

- Official CUDA page
  http://developer.nvidia.com/category/zone/cuda-zone
  - CUDA C Programming Guide
  - CUDA C Best Practices Guide
  - CUDA Reference Manual

- NVIDIA Developer Zone
Thank you.

Any Questions?

hondak@isc.chubu.ac.jp
## NVIDIA GPU

<table>
<thead>
<tr>
<th></th>
<th>Intel Core i7 Extreme</th>
<th>GeForce GTX 285</th>
<th>GeForce GTX 480</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peak Performance [GFlops]</td>
<td>102.4</td>
<td>708*, 1063</td>
<td>1344.96*</td>
</tr>
<tr>
<td>Number of Processor</td>
<td>4</td>
<td>240</td>
<td>480</td>
</tr>
<tr>
<td>Core Clock [MHz]</td>
<td>3200</td>
<td>1476</td>
<td>1401</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bandwidth [GB/s]</td>
<td>32</td>
<td>159</td>
<td>177.4</td>
</tr>
<tr>
<td>Memory Interface [bit]</td>
<td>64</td>
<td>512</td>
<td>384</td>
</tr>
<tr>
<td>Memory Data Clock [MHz]</td>
<td>1333 (DDR3)</td>
<td>2484 (GDDR3)</td>
<td>3696 (GDDR5)</td>
</tr>
<tr>
<td>Capacity [GB]</td>
<td>-----</td>
<td>2046</td>
<td>1536</td>
</tr>
</tbody>
</table>

*Courtesy of Tokyo Institute of Technology*