GPUゼミ (川崎)

参考
- CUDA By Example, by Jason Sanderse + Edward Kandrot
  (翻訳) インプレスジャパン
- GPU tutorial By 牛川さん

準備 手順:
- ssh to gpu computer (budou, melon, einstein, landau.)
- Make "workespace" directory
- Do "git clone"

GPU ゼミ

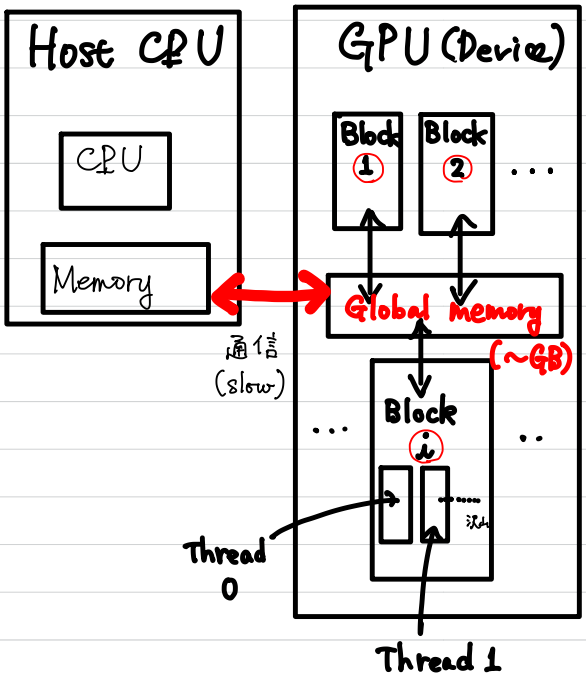
基本事項

CUDA : NVIDIA が開発した C / C++ ベースの GP GPU 計算用プログラミング言語

Host と Device

Host: CPU
Device: GPU

CUDA と GPU 構造



- Host と Device 間の通信は、時間がかかる
  計算は、なるべく、Device 内で主に行い、
  データの掃き出しなどに限り Host (CPU) で処理することがコツ。

Block と Thread は、3次元的に配置
上限: Block (x, y, z) =

$(2147483647,\ 65535,\ 65535)$
$O(10^9)$

Thread: (x, y, z) = (1024, 1024, 64)

☆ x方向のみで十分

(注) この Block数や Thread 数は、GPU の実際の構造を示しているわけではなく、CUDA が記述した数値

実際の構造
NVIDIA Tesla A100:
スレッド数 6812
(プロセッサ数)

x方向のみつかり
$O(10^9)$ まで ok 1024
↓ 単位 ↓
< block数 × thread数
[6812 スレッドごとに同時計算]

- GPU 並列による高速化

```
for(int i=0; i<N; i++)
  for(int j=0; j<N; j++){
      ....
  }
```

```
int i_global = blockId.x * blockDim.x
             + threadId.x ;
if(i_global < N){
   for(int j=0; j<N; j++){
      ...
   }}
```

i_global (Max) ~ O(N)
i が for文をすべてカバー

- Device code (GPU 計算)

▶ Kernel
デバイス上でプログラムを走らせる関数の内、ホスト側で呼び出される関数

```
__global__ void addVec(      ){
    .....
}

int main(){
    addVec<<<NB, NT>>>( 変数1, 変数2 )
                デバイス  デバイス
                block  thread
    ....
}
```

▶ Kernel 上で動く関数

- Kernel 内で Host 内の関数を呼び出すことは、出来ない。
- Device 上で呼び出して device 上で動く関数が必要

```
__device__ float square_dev(...){
    .....
}

__global__ void kernel(...){
    ....
    x = square_dev(  );
    ...
}
```

CUDA programing

▶ CUDAの実行

hello.cu

```
#include <iostream>
using name space std;
int main(){
    cout << "hello CUDA" << endl;
    return 0;
}
```

```
$ nvcc hello.cu
$ ./a.out

    hello CUDA
```

▶ kernelの導入

何もしないkernel

hello_kernel.cu

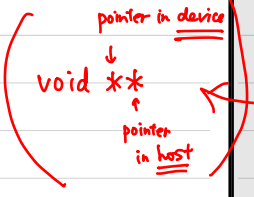
```
__global__ void myKernel(){
    // nothing here.
}

int main(){
    cout << "Hello CUDA" << endl;
    myKernel<<<1, 1>>>();
    return 0;
}
```

山括弧構文：
関数名 <<< block数, thread数 >>>( );

Book p22
cudaMallocの第1変数
(void **)

新たに割あてられたアドレスを保存するための ポインタのポインタ.



GPUを用いた加算

add.cu

```
#inculde <iostream>
#inculde <cuda.h>
using name space std;
__global__ void add(int *c, int *a,
                    int *b){
    *c = *a + *b;
}
int main(){
    int a, b, c;          // values for the host
    int *a_dev, *b_dev, *c_dev;
                          // values for the device
                          // *a_dev : ポインタの値
    // allocate memories on the device.
    cudaMalloc((void**)&a_dev, sizeof(int));
    cudaMalloc((void**)&b_dev, sizeof(int));
    cudaMalloc((void**)&c_dev, sizeof(int));

    a = 2;
    b = 7;

    cudaMemcpy(a_dev, &a, sizeof(int)
        , cudaMemcpyHostToDevice);

    cudaMemcpy(b_dev, &b, sizeof(int)
        , cudaMemcpyHostToDevice);

    add<<<1, 1>>>(c_dev, a_dev, b_dev);

    cudaMemcpy(&c, c_dev, sizeof(int),
        cudaMemcpyDeviceToHost);

    cout << "c: " << c << endl;

    cudaFree(a_dev);
    cudaFree(b_dev);
    cudaFree(c_dev);
    return 0;
}
```

a_dev は pointer
ポインタ ポインタ
pointer pointer pointer

```
$ nvcc  add.cu
$ ./a.out
    c: 9
```

■ ベクトル計算

- 逐次処理 (sequential processing)

timer.cuh

↳ mearsureTime( ) を2回呼ぶ間の時間をmsで返す.

- CPUを用いたベクトル加算

add_vec_serial_cpu.cu

```
#include <iostream>
#include <cuda.h>
#include "timer.cuh"
                // current directryに置く.

using namespace std;

const int Nv = 1e+6;

void setup_vec(int *vec, int a){
   for(int i=0; i<Nv; i++)
      vec[i] = i*a;
}

void add_vec(int *c, int*a, int *b){
   for(i=0; i<Nv; i++)
      c[i] = a[i]+b[i];
}
```

```
int main( ){
   int *a, *b, *c;
a= (int*)malloc (Nv * sizeof(int));
b= (int*)malloc (Nv * sizeof(int));
c= (int*)malloc (Nv * sizeof(int));

 setup_vec(a, 1);
 setup_vec(b, 2);

 double ms;
 measureTime( );
 for(i=0; i<100000; i++)
     add_vec (c, a, b);
 ms = measureTime( );
 cout << "Time" << ms/1000
                << "ms" << endl;

 // Free
 free(a);
 free(b);
 free(c);
 return 0;
}
```

和の計算を100000回繰り返した際の計算時間を計測.

(3分くらいかかる)

add_vec_serial_gpu.cu

```
....

__global__ void add_vec(int *c, int *a, int *b){
    for(int i=0; i<Nv; i++)
      c[i] = a[i] + b[i];
}

int main(  ){
    int *a, *b, *c;
    int *a_dev, *b_dev, *c_dev;
    a= (int *)malloc(Nv * sizeof(int));
    b= (int *)malloc(Nv * sizeof(int));
    c= (int *)malloc(Nv * sizeof(int));
    cudaMalloc((void**)&a_dev, Nv*sizeof(int));
    cudaMalloc((void**)&b_dev, Nv*sizeof(int));
    cudaMalloc((void**)&c_dev, Nv*sizeof(int));
    setup_vec(a, 1);
    setup_vec(b, 1);
    cudaMemcpy(a_dev, a, Nv*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(b_dev, b, Nv*sizeof(int), cudaMemcpyHostToDevice);
    doube ms;
    measureTime( );                                  // ここから時間測る
    for(int i=0; i<~~1000~~ 100000; i++)
      add_vec<<<1, 1>>>(c_dev, a_dev, b_dev);        // gpu
    cudaMemcpy(c, c_dev, Nv*sizeof(int), cudaMemcpyDeviceToHost);
    ms = measureTime( );
    cout << "Time" << ms/1000 << "ms" << endl;

    free(a);
    free(b);
    free(c);
    cudaFree(a_dev);
    cudaFree(b_dev);
    cudaFree(c_dev);

    return 0;
}
```

- 並列計算
  - threadの活用

add_vec_prallel.cu

```
#include<iostream>
#include<cuda.h>
#include"timer.cuh"
using namespace std;
const int Nv = 1000000;
const int NT = 512;
const int NB = (Nv+NT-1)/NT;
.....
__global__ void add_vec(int *c, int *a, int *b){
  int i_global = blockIdx.x * blockDim.x + threadIdx.x;
                 (block ID)  (block数 x方向)  (thread ID = 1024)
  if(i_global < Nv)
      c[i_global] = a[i_global] + b[i_global];
}

int main(){
  .....
  double ms;
  measureTime();
  for(int i=0; i<~~1000~~ 100000; i++)
    add_vec<<<NB, NT>>>(c_dev, a_dev, b_dev);

  cudaMemcpy(c, c_dev, Nv*sizeof(int), cudaMemcpyDeviceToHost);
  ms = measureTime();
  cout<<"Time:"<<ms/1000<<"ms"<<endl;
  ....
  return 0;
}
```

Lecture2 総和計算

並列計算において、総和計算は注意が必要

悪い例

```
const int Narr = 10000000
const int NT = 512
const int NB = (Narr + NT - 1) / NT
__global__ gpu_sum(int *sum_dev, int *arr_dev){

    int i_global = blockIdx.x * blockDim.x + threadIdx.x;
     if(i_global < Narr)
          *sum_dev += arr_dev[i_global];
     }

   ....
int sum(int *arr){
   int sum;
   int *sum_dev, *arr_dev;
   cudaMalloc((void**)&sum_dev, sizeof(int));
   cudaMalloc((void**)&arr_dev, Narr * sizeof(int));
   cudaMemcpy(arr_dev, arr, Narr * sizeof(int), cudaMemcpyHostToDevice);
   gpu_sum<<<NB, NT>>>(sum_dev, arr_dev);
   cudaMemcpy(sum, sum_dev, sizeof(int), cudaMemcpyDeviceToHost);
   cudaFree(sum_dev);
   cudaFree(arr_dev);
   return sum;
}
以下 main
```

問題点：

- *sum_dev に 複数スレッドが同時に accessする。
- 計算結果は、正しくない結果になる。
- コンパイラは、この問題点を教えてくれない。

- Atomic 演算

sum_atomic.cu

```
....
__global__ gpu_sum(int *sum_dev, int *arr_dev){
    int i_global = blockIdx.x * blockDim.x + threadIdx.x;
    if(i_global < Narr)
        atomicAdd(sum_dev, arr_dev[i_global]);
}
....
```

atomicAdd を用いることで. access の衝突は. 起こらなくなる。
ただし、sum_dev にある thread が書き込んでいる間, 他の thread は. 待機
している為. 逐次処理と同等.