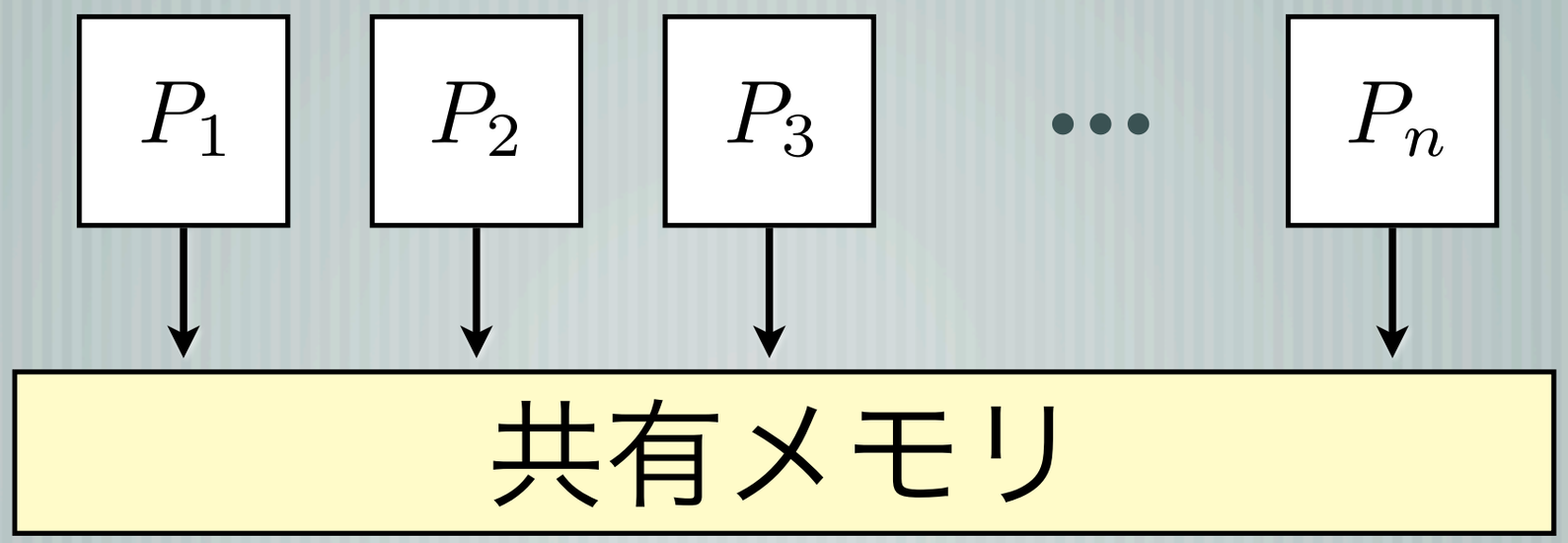
OpenMPによるマルチスレッド化とCUDA上のPointer Jumpingによる並列処理

山本修身

マシンモデル PRAM (1)

- PRAM (Parallel RAM) は初回に説明したRAM (random access machine) のモデルを並列化したものである.
- PRAMのモデルはn個のプロセッサ（普通のRAM）によって共有メモリを同時にアクセスするものである.



マシンモデル PRAM (2)

- それぞれのプロセッサが共有メモリからデータを読んだり，共有メモリにデータを書き込む場合の方法によって以下のように分類される
  - CRCW: 同じメモリを同時に読み出し，同時に書き込める.
  - EREW: 同じメモリを一つのプロセッサが読み出すことができ，書き込む場合もいずれか一つのプロセッサが書くことができる.

OpenMPについて

- 複数のコアを持つCPUアーキテクチャ上の並列化が簡単に実現できる方法としてOpenMPがある．多くのCコンパイラがOpenMPに対応している（gcc, Visual C++, Intel C compilerなど）．

簡単な例（１）

1からnまでの平方の逆数の和を計算する．すなわち，

$$S = \sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$$

を計算する．これをCで計算するには以下のようにする

```c
#include <stdio.h>
#include <stdlib.h>

double sumx(int n){
  int i;
  double s = 0.0;
  for (i = 1; i < n; i++){
    s += 1.0 / i / i;
  }
  return s;
} /* sumx */
```

```c
int main(){
  enum{N = 50000000};
  double s = sumx(N);
  printf("S = %f\n", s);
  return 0;
} /* main */
```

```
OMacBook:yama546> time ./sample0
S = 1.644934

real    0m0.958s
user    0m0.904s
sys     0m0.005s
```

簡単な例（2）

この計算をOpenMPで並列化してみる.

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

double sumx(int n1, int n2){
  int i;
  double s = 0.0;
  for (i = n1; i < n2; i++){
    s += 1.0 / i / i;
  }
  return s;
} /* sumx */

#define max(x, y) (((x) > (y))?x:y)
```

```c
int main(){
  enum{N = 50000000};
  double s = 0.0;
  int i;
#pragma omp parallel for
  for (i = 0; i < 2; i++){
    s = s + sumx(max(1, i * N / 2),
                 (i + 1) * N / 2);
  }
  printf("S = %f\n", s);
  return 0;
} /* main */
```

```
OMacBook:yama548> time ./sample2
S = 1.644934

real     0m0.500s
user     0m0.896s
sys      0m0.004s
```

簡単な例（3）

- 別の書き方もできる.

```
OMacBook:yama553> time ./sample3
S = 1.644934

real    0m0.478s
user    0m0.895s
sys     0m0.005s
OMacBook:yama554>
```

```
int main(){
  enum{N = 50000000};
  double s1, s2;
  int i;
#pragma omp parallel
#pragma omp sections
  {
#pragma omp section
    {
      s1 = sumx(1, N / 2);
    }
#pragma omp section
    {
      s2 = sumx(N / 2, N);
    }
  }
  printf("S = %f\n", s1 + s2);
  return 0;
} /* main */
```

簡単な例（4）

- 粒度を調整する．荒すぎても細かすぎても良くない．

```
int main(){
  enum{N = 50000000, M = 20};
  double s = 0.0;
  int i;
#pragma omp parallel for
  for (i = 0; i < M; i++){
    s = s + sumx(max(1, i * N / M),
                 (i + 1) * N / M);
  }
  printf("S = %f\n", s);
  return 0;
} /* main */
```

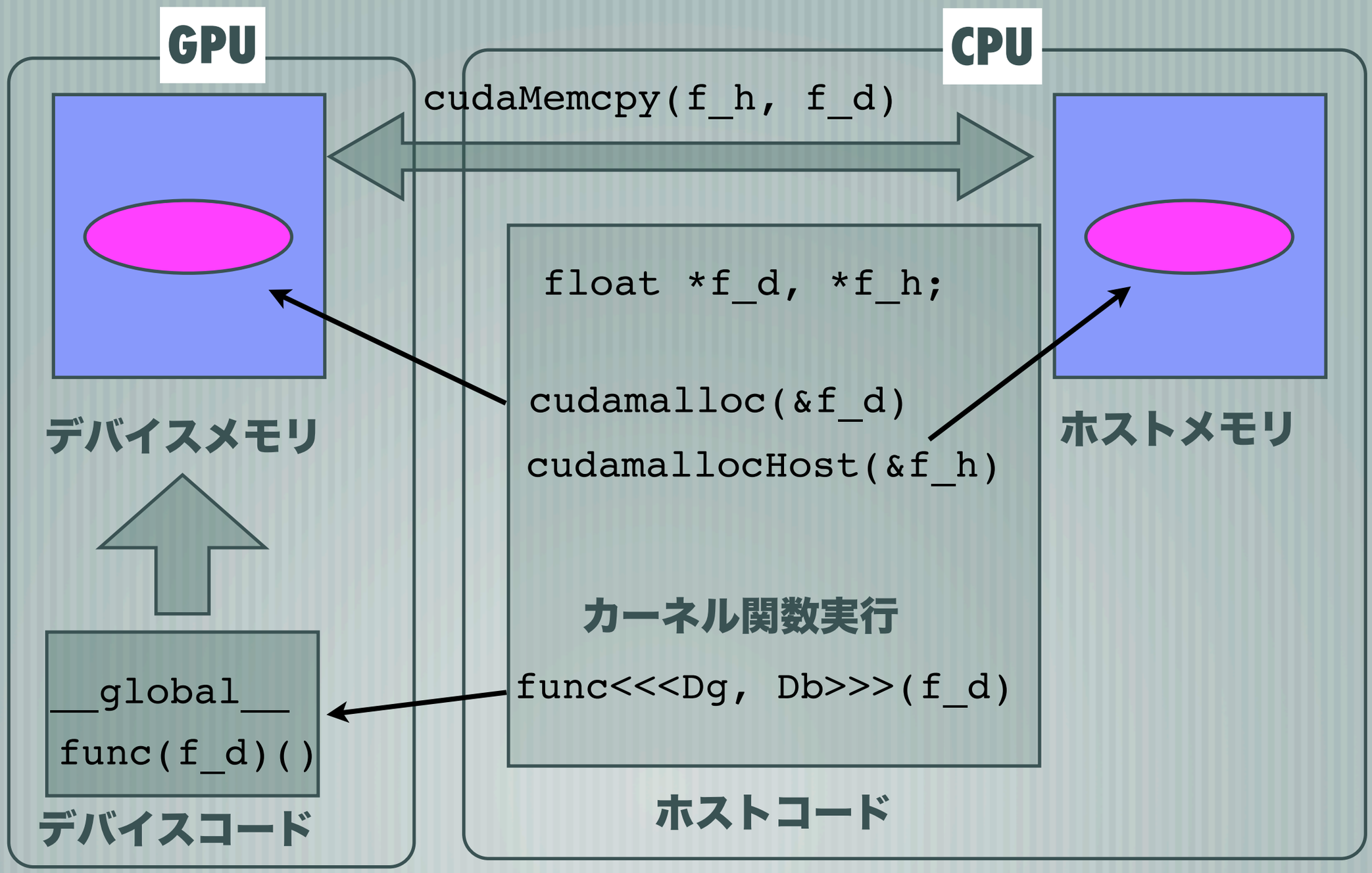
CUDAによる並列プログラミング (1)

- nVIDIA製のグラフィックスボードを超並列計算機として利用するための枠組み（開発環境）が**CUDA（Compute Unified Device Architecture)**である．CUDAを用いることにより，並列計算を実際にプログラムとして実現し実行することができる．（もちろんそれが高速であるかどうかはハードウエアの性能などによる．）

- ここで説明する並列アルゴリズムをCUDA上で実現してみる.

CUDAによる並列プログラミング (2)

- CUDAのアーキテクチャは以下のとおり.



CUDAによる並列プログラミング (3)

- CUDAのコードはほぼC言語であるが，カーネル関数の呼び出しなどにおいて特殊な文法を用いる.
- CUDAのコードはnVIDIAが提供するnvccによってコンパイルされる．nvccは既存のコンパイラ（VC++)などを利用しながら実行形式を生成する.

```
OMacBook:yama531> nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Fri_Jan_13_01:52:47_PST_2012
Cuda compilation tools, release 4.1, V0.2.1221
OMacBook:yama532>
```

CUDAによる並列プログラミング (4)

デバイスメモリ（GPU上のメモリ）上でメモリ領域を確保するには`cudaMalloc`関数を用いる.

```
float *pt;

cudaMalloc(&pt, N * sizeof(float));
```

デバイスメモリ上のデータへのポインタ

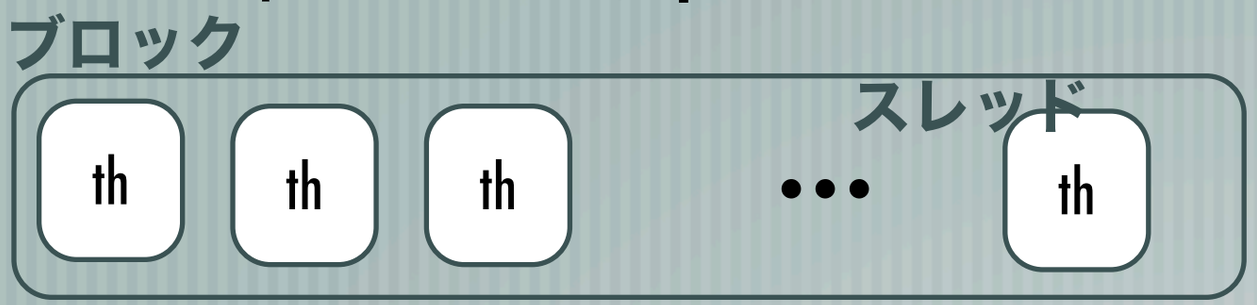
デバイスメモリ上のデータのサイズ

CUDAによる並列プログラミング (5)

- `cudaMalloc`関数で確保したメモリは`cudaFree`関数で解放することができる.
- `cudaMallocHost`関数によって，CPU上のメモリに領域を確保することができる. これは通常の `malloc`関数とほぼ同じ.
- `cudaMemcpy`関数によって，CPU⇔GPU, GPU⇔GPUのデータコピーが可能である.

CUDAによる並列プログラミング (6)

GPU上でカーネル関数の実行させるときのモデルは以下のとおり.



CPUでの実行

```
foobar<<<20, 30>>>(a)
```

ブロック数　スレッド数

GPUでの並列実行

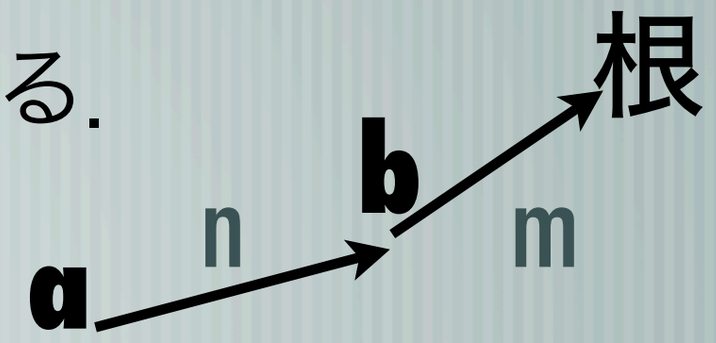
```
foobar(a) [0, 0]
foobar(a) [0, 1]
foobar(a) [0, 2]
    ⋮
foobar(a) [19, 29]
```

もちろんGPU上のプロセッサーの個数には限りがあるので，すべてが同時に実行されるわけではない．どの関数呼び出しの順番は保証されていないので順番を仮定してはいけない

Pointer Jumpingアルゴリズム (1)

- Pointer Jumping アルゴリズムは木のそれぞれのノードからルートまでの経路についてある量 f(n) を計算するためのアルゴリズムである.

- ただし，計算すべき量は木の上の経路について準同形であり，この量に関する演算は結合則を満たすとする．すなわち，f(n △ m) = f(n) ○ f(m) であり，○は結合則を満たす演算となっている.



Pointer Jumpingアルゴリズム (2)

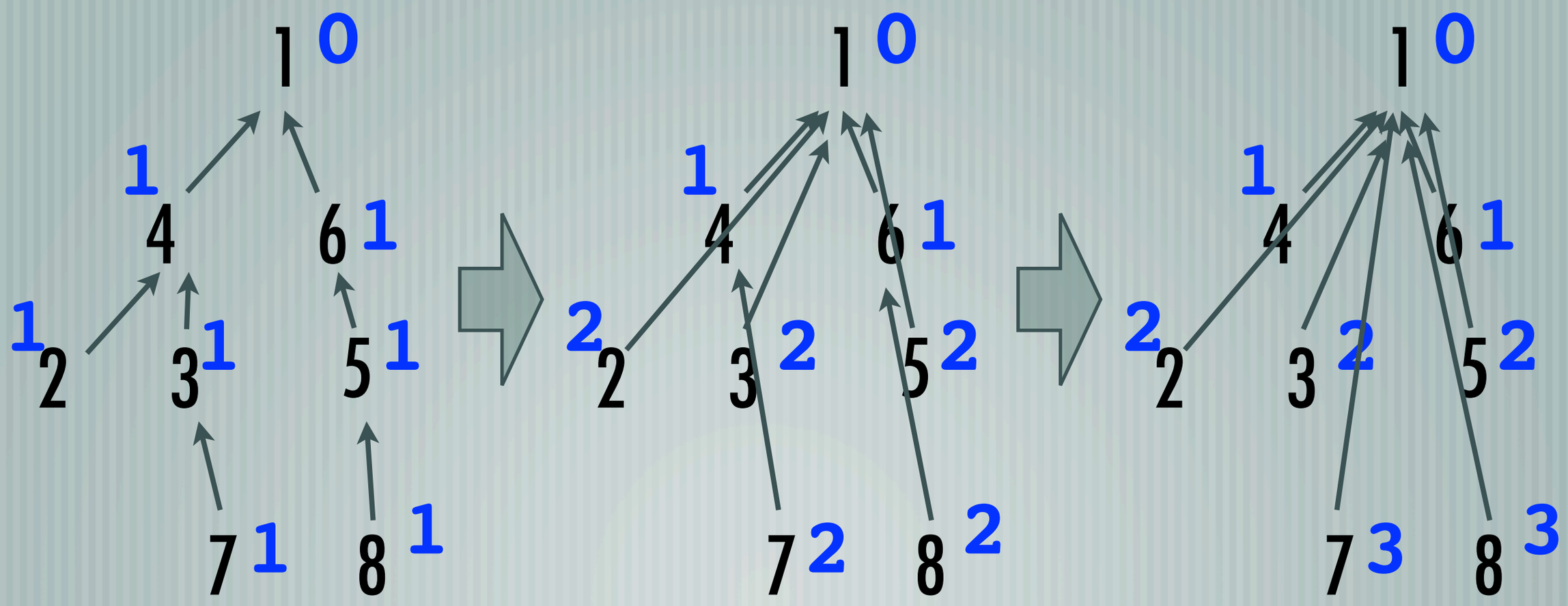
- アルゴリズムは以下のとおり：

  ノードiのデータがx[i]に入っているとする．また，next[i]にノードiの親ノードが入っているとする．総和をとる演算は○である．

```
for 各ノードi（並列に）:
    y[i] = x[i]
while next[i] != nil であるようなノードiが存在するかぎり:
    for 各ノードi（並列に）:
        if next[i] != nil:
            y[i] = y[i] ○ y[next[i]]
            next[i] = next[next[i]]
```

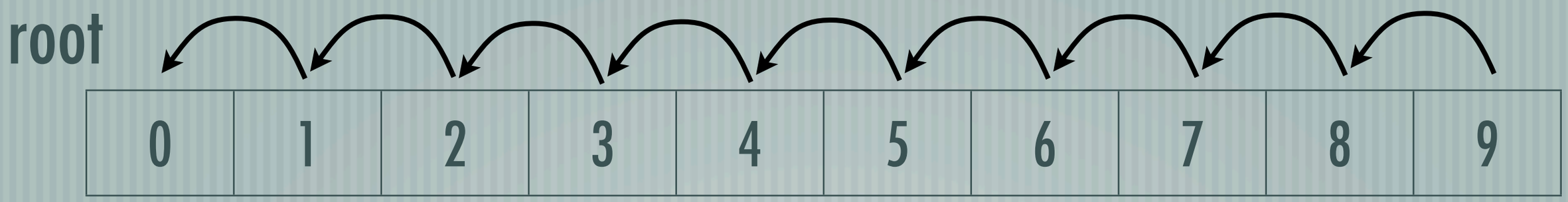
Pointer Jumpingアルゴリズム (3)

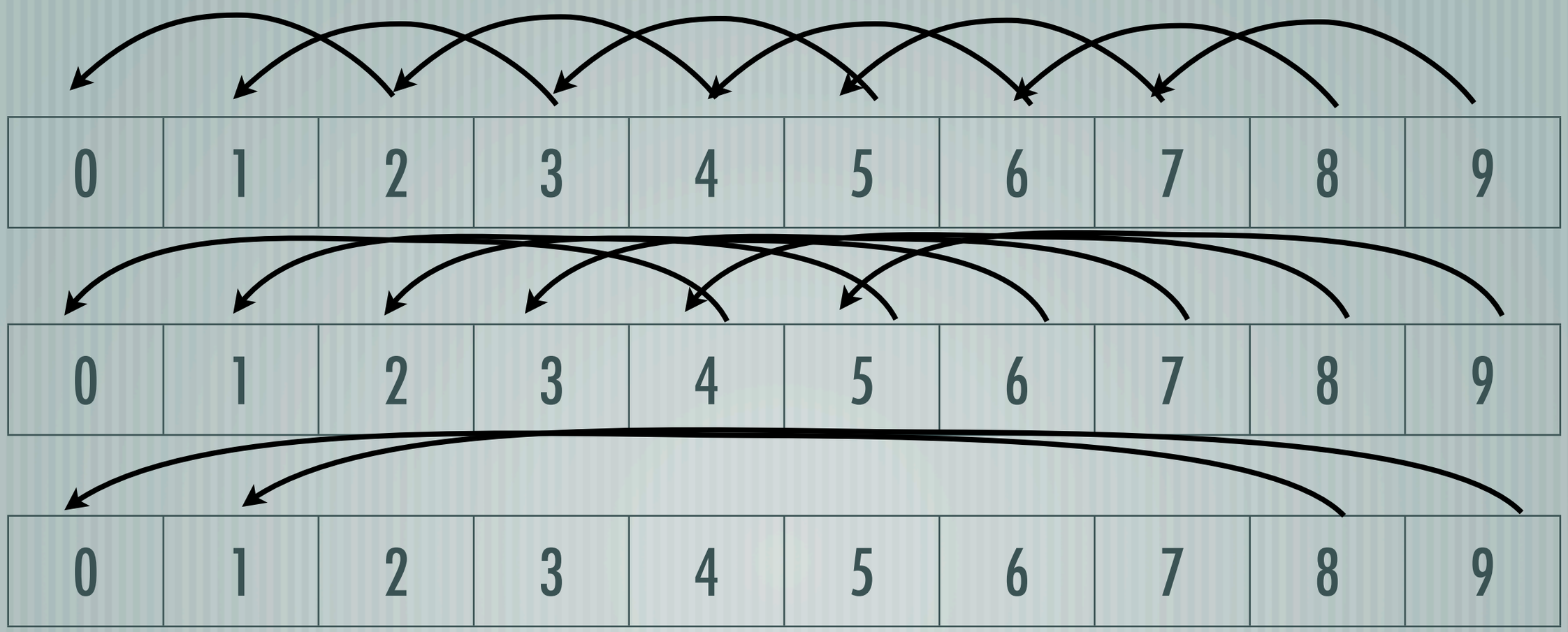
- Pointer Jumping アルゴリズムを用いて木の深さをそれぞれのノードについて計算する．木の深さのlogくらいの時間ですべての深さが確定する．



Pointer Jumpingアルゴリズム (4)

- 線形リストのpointer jumping アルゴリズム





Pointer Jumpingアルゴリズム (5)

- pointer jumpingでリストの内容の和を計算してみる

CUDAのプログラム例１（1）

右の級数を計算してみる

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

```
int main(){
  float *pt, *pt1;
  int m = 1, count = 0;
  float ans;

  cudaMalloc(&pt, N * sizeof(float));
  cudaMalloc(&pt1, N * sizeof(float));
  init <<< Na, Nb >>>(pt);
  while (m < N){
    add1 <<< Na, Nb >>>(pt, pt1, m);
    add2 <<< Na, Nb >>>(pt, pt1, m);
    m = m * 2;
    count += 1;
  }
  printf("count = %d\n", count);
  cudaMemcpy(&ans, &pt[N - 1], sizeof(float), cudaMemcpyDeviceToHost);
  printf("%10.8f: %10.8f\n", ans, sqrt(ans * 6));

  cudaFree(pt);
  cudaFree(pt1);
  return 0;
}
```

デバイス上にメモリを確保する．pt1はワーキングエリア

それぞれの$1/n^2$をメモリに入れる

pointer jumpingを行う

結果の取り出し

```
#include <stdio.h>
#include <math.h>

const int Na = 10000;
const int Nb = 50;
const int N = Na * Nb;
```

CUDAのプログラム例１ (2)

それぞれのカーネル関数の定義

```
__global__
void init(float *pt){
  int i = blockIdx.x;
  int j = threadIdx.x;
  int k = i * Nb + j;
  float m = 1 + k;
  pt[k] = 1.0f / m / m;
}
```

```
__global__
void add2(float *pt, float *pt1, int m){
  int i = blockIdx.x;
  int j = threadIdx.x;
  int k = i * Nb + j;

  if (k + m < N)
    pt[k + m] = pt1[k + m];
}
```

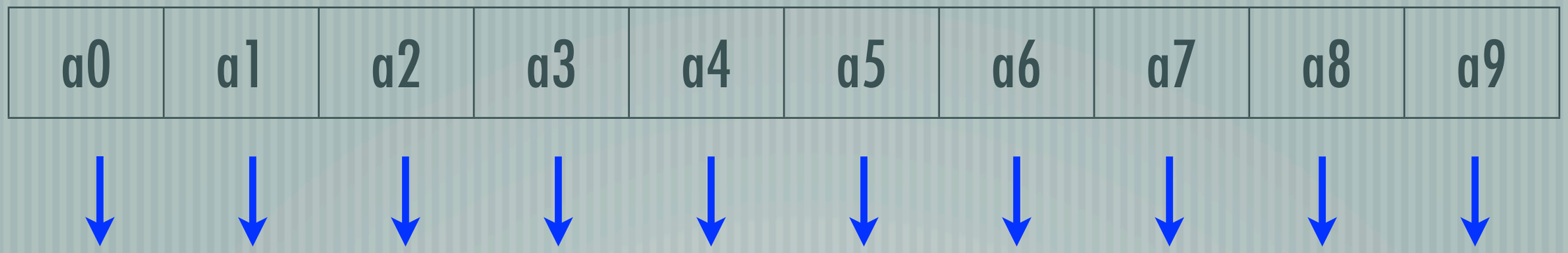
```
__global__
void add1(float *pt, float *pt1, int m){
  int i = blockIdx.x;
  int j = threadIdx.x;
  int k = i * Nb + j;

  if (k + m < N)
    pt1[k + m] = pt[k] + pt[k + m];
}
```

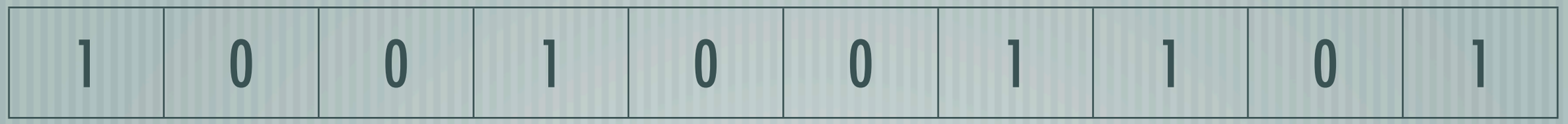
```
OMacBook:yama539> time ./sample1
count = 19
1.64493191: 3.14159060

real    0m0.083s
user    0m0.056s
sys     0m0.023s
OMacBook:yama540>
```

GeForce 9400M 256M

配列上のデータをフィルタリングしてみる (1)

- 配列上の条件に合うデータだけ取り出す



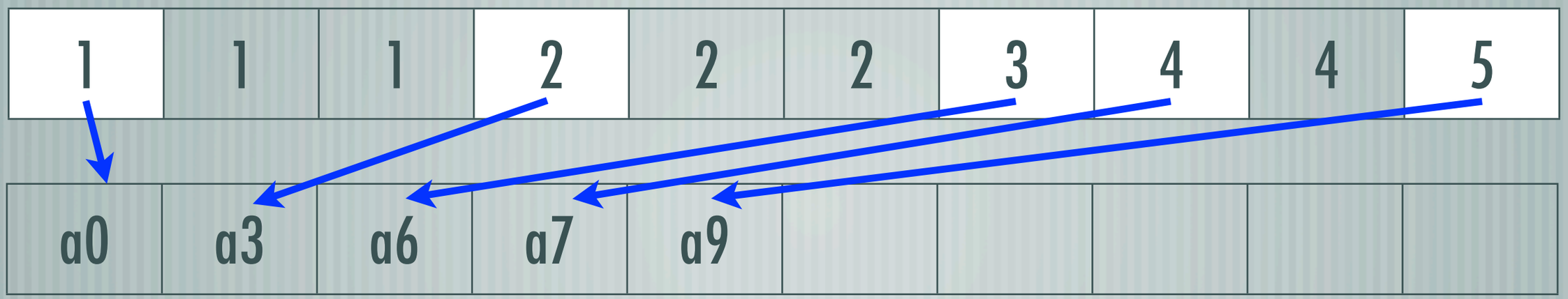


pointer jumpingを用いてこれを生成する

↘

1: 条件に合う

0: 条件に合わない



配列上のデータをフィルタリングしてみる (2)

[0, 1]の一様分布に従う確率変数X[1], ... X[10]の和は中心極限定義より，正規分布に漸近する．各確率変数について

$$E(X) = \int_0^1 x dx = \left[\frac{x^2}{2}\right]_0^1 = \frac{1}{2}$$

$$V(X) = E(X^2) - E(X)^2 = \int_0^1 x^2 dx - \left(\frac{1}{2}\right)^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}$$

なので10個確率変数の和の平均と分散は，5と5/6である．
したがって，以下の確率変数Yはほぼ標準正規分布に従う．

$$Y = \frac{\sum_{i=1}^{10} X_i - 5}{\sqrt{5/6}}$$

配列上のデータをフィルタリングしてみる (3)

Yに対応する50万個データを作成して，その分布を調べてみる.

```
float * init(){
  int i, j;
  float *pt;
  float *ptx;

  cudaMallocHost(&pt, sizeof(float) * N);
  cudaMalloc(&ptx, sizeof(float) * N);
  float r = sqrt(6.0f / 5.0f);
  for (i = 0; i < N; i++){
    float s = 0.0f;
    for (j = 0; j < 10; j++)
      s += (float)random() / max_rand;
    pt[i] = (s - 5) * r ;
  }
  cudaMemcpy(ptx, pt, sizeof(float) * N, cudaMemcpyHostToDevice);
  cudaFree(pt);
  return ptx;
}
```

デバイスメモリに書き込む

デバイスメモリ上のアドレスを返す

配列上のデータをフィルタリングしてみる (4)

データからある限界以下のデータの個数を数えるプログラムをGPUで計算させる．カーネル関数markによってlistにカウントのための0/1のイメージの配列を構成する.

```
__global__
void mark(int *list, float *pt, float r){
  int i = blockIdx.x;
  int j = threadIdx.x;
  int k = i * Nb + j;

  if (pt[k] < r) list[k] = 1;
  else list[k] = 0;
}
```

配列上のデータをフィルタリングしてみる (5)

```
__global__
void add1(int *list, int *list1, int m){
  int i = blockIdx.x;
  int j = threadIdx.x;
  int k = i * Nb + j;

  if (k + m < N)
    list1[k + m] = list[k] + list[k + m];
}

__global__
void add2(int *list, int *list1, int m){
  int i = blockIdx.x;
  int j = threadIdx.x;
  int k = i * Nb + j;

  if (k + m < N)
    list[k + m] = list1[k + m];
}
```

配列 list の中の1の個数を数えるためのpointer jumping のプログラムを作る

配列上のデータをフィルタリングしてみる (6)

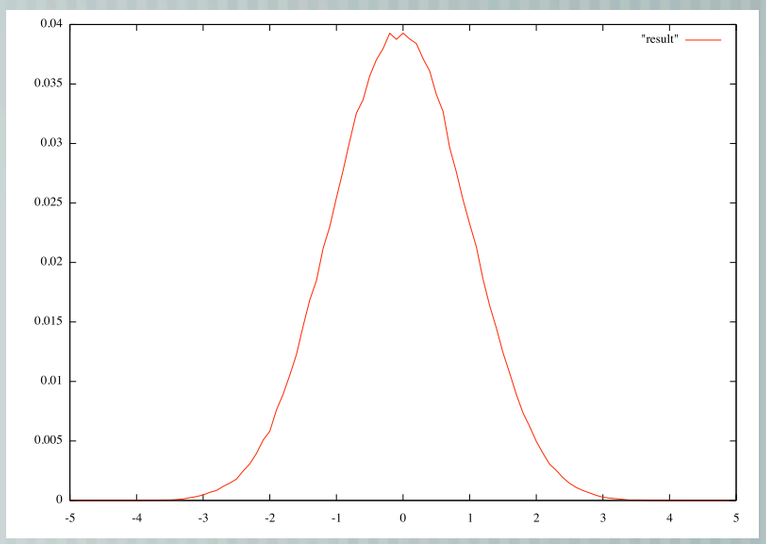
- しきい値rを設定して，r以下の値の個数をカウントするプログラム

```
int count_it(float r, int *list, int *list1, float *pt){
  int m = 1;
  int ans;

  mark <<< Na, Nb >>>(list, pt, r);

  while (m < N){
    add1 <<< Na, Nb >>>(list, list1, m);
    add2 <<< Na, Nb >>>(list, list1, m);
    m = m * 2;
  }
  cudaMemcpy(&ans, &list[N - 1], sizeof(int), cudaMemcpyDeviceToHost);
  return ans;
}
```

配列上のデータをフィルタリングしてみる (7)

```
int main(){
  int *list, *list1;

  float *pt = init();
  cudaMalloc(&list, N * sizeof(int));
  cudaMalloc(&list1, N * sizeof(int));
  float r1 = (float)(0 - 50) / 10;
  int c1 = count_it(r1, list, list1, pt);
  for (int i = 0; i < 100; i++){
    float r2 = (float)(i + 1 - 50) / 10;
    int c2 = count_it(r2, list, list1, pt);
    printf("%f  %f\n", r1, (float)(c2 - c1) / N);
    c1 = c2;
    r1 = r2;
  }
  cudaFree(pt);
  cudaFree(list);
  cudaFree(list1);
  return 0;
} /* main */
```

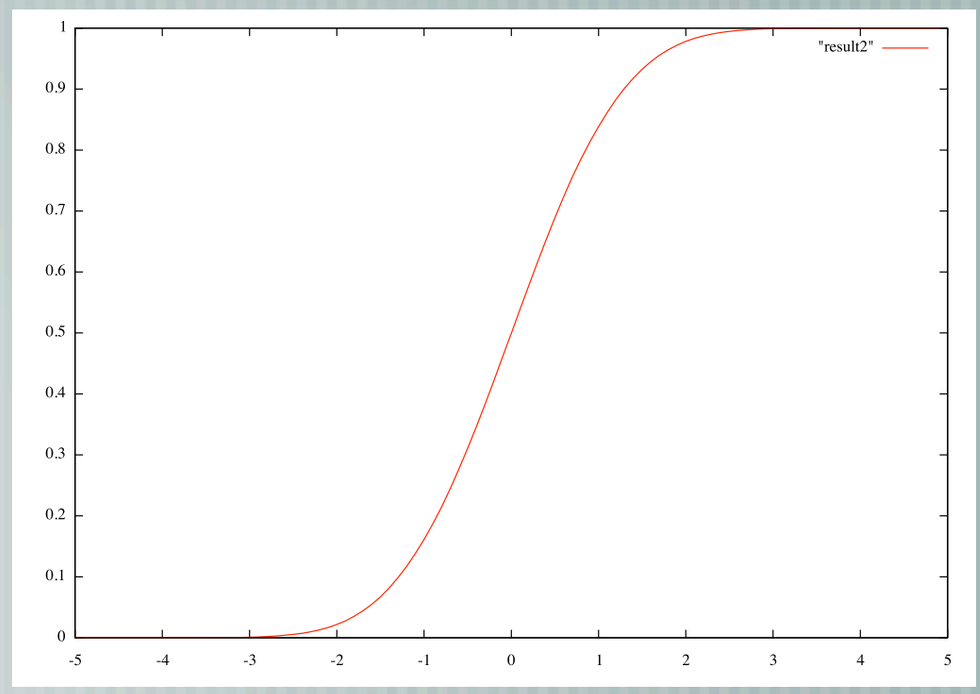
この関数を用いて標準正規分布の密度関数を描く



配列上のデータをフィルタリングしてみる (8)

標準正規分布の分布関数を描くためには以下のようにする.

```
int main(){
  int *list, *list1;
  float *pt = init();
  cudaMalloc(&list, N * sizeof(int));
  cudaMalloc(&list1, N * sizeof(int));
  for (int i = 0; i < 100; i++){
    float r = (float)(i - 50) / 10;
    int c = count_it(r, list, list1, pt);
    printf("%f  %f\n", r, (float)c / N);
  }
  cudaFree(pt);
  cudaFree(list);
  cudaFree(list1);
  return 0;
} /* main */
```



配列上のデータをフィルタリングしてみる (9)

実際に条件に合う要素だけを取り出すには以下のようなカーネル関数を用いる.

```
__global__
void filter(int *list, float *pt, float *pta){
  int i = blockIdx.x;
  int j = threadIdx.x;
  int k = i * Nb + j;

  if (k > 0){
    if (list[k] - 1 == list[k - 1])
      pta[list[k] - 1] = pt[k];
  } else if (k == 0){
    if (list[k] == 1){
      pta[0]  = pt[0];
    }
  }
}
```

取り出された要素は配列 `pta` にコピーされる

配列上のデータをフィルタリングしてみる (10)

- カーネル関数filterを用いてフィルタリングしてみる.

```
int filter_it(float r, int *list, int *list1, float *pt, float *pta){
  int m = 1;
  int ans;

  mark <<< Na, Nb >>>(list, pt, r);
  while (m < N){
    add1 <<< Na, Nb >>>(list, list1, m);
    add2 <<< Na, Nb >>>(list, list1, m);
    m = m * 2;
  }
  filter <<< Na, Nb >>>(list, pt, pta);
  cudaMemcpy(&ans, &list[N - 1], sizeof(int), cudaMemcpyDeviceToHost);
  return ans;
}
```

配列上のデータをフィルタリングしてみる (11)

```
int main(){
  int *list, *list1;
  float *ptx, *pty;

  float *pt = init();
  cudaMalloc(&list, N * sizeof(int));
  cudaMalloc(&list1, N * sizeof(int));
  cudaMalloc(&ptx, N * sizeof(float));
  cudaMallocHost(&pty, N * sizeof(float));
  int c = filter_it(-4.0, list, list1, pt, ptx);
  printf("%d\n", c);
  cudaMemcpy(pty, ptx, sizeof(float) * N, cudaMemcpyDeviceToHost);
  for (int i = 0; i < c; i++){
    printf("%f\n", pty[i]);
  }
  cudaFree(pt);
  cudaFree(list);
  cudaFree(list1);
  cudaFree(ptx);
  return 0;
} /* main */
```

```
OMacBook:yama519> ./sample2
err = 0(no error)
err = 0(no error)
3
-4.370888
-4.030611
-4.542349
OMacBook:yama520>
```

実際に-4.0以下になる値になる

まとめ

- CUDAを用いた並列計算のプログラミングについて解説した
- 基本的なアルゴリズムとしてpointer jumpingアルゴリズムを