SuperCon 2018

GPUプログラミング・基礎編

この資料は，本選参加者に対する事前資料として配布するものです．できれば目を通しておいて下さい．ただし，わからなくても心配無用です．本選初日に，この資料を使って説明会をしますので．もちろん，CUDA プログラミングを事前に練習しておく必要もありません．

GPUの
説明の前に・・・

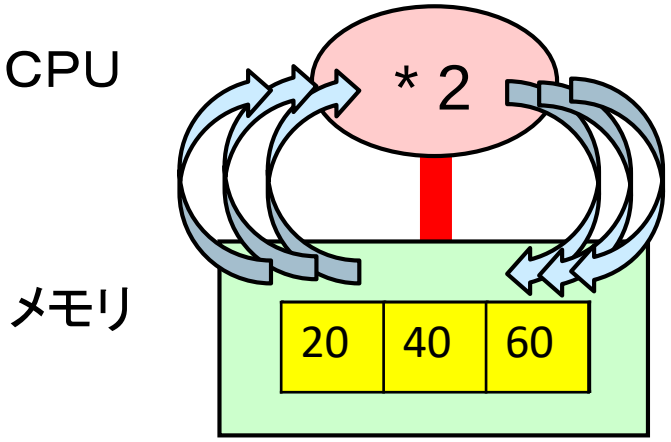
コンピュータ上での プログラムの動きを再確認

- コンピュータの重要な部品：CPUとメモリ
- メモリ：変数などのデータの置き場
- CPU: メモリからデータを読んで演算を行う

プログラムの例

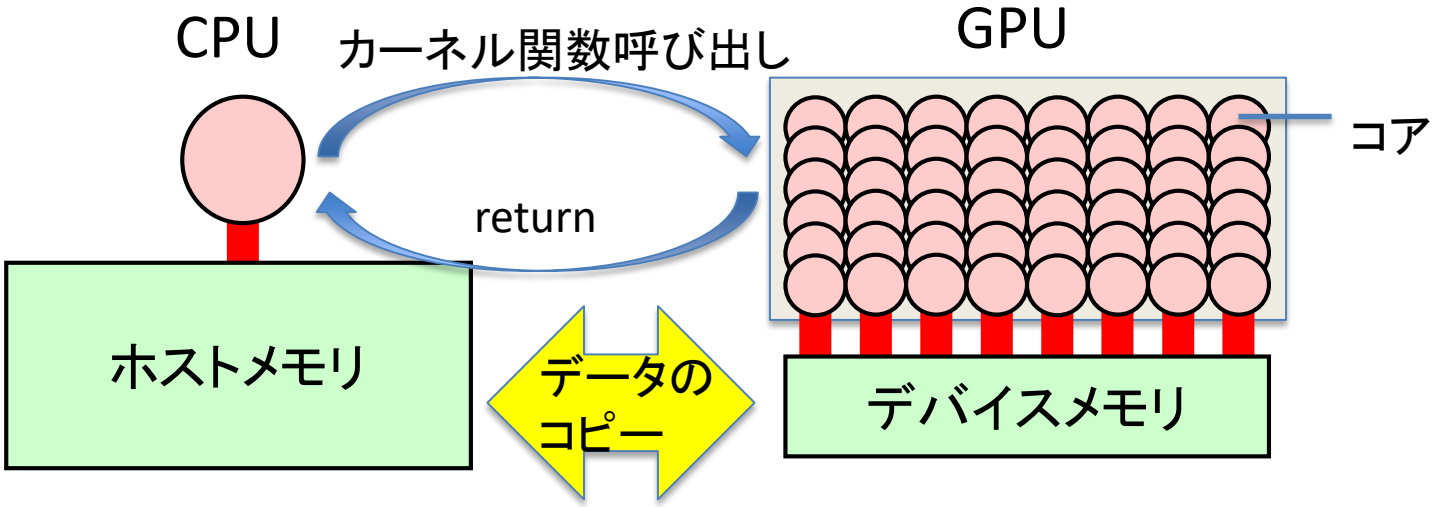
```
int main() {
  int a[3] = {10, 20, 30};
  int i;

  for (i = 0; i < 3; i++) {
     a[i] = a[i] *2;
  }
}
```

コンピュータ



GPUを用いた演算とは

- CPUのほかにGPUも計算することができる
- GPUの特徴:
  - 単体では動くことはできない。CPUから何らかの指示(カーネル関数呼び出し)をされて動く
  - GPUが使えるメモリ(デバイスメモリ)は、CPU側のメモリ(区別のためホストメモリと呼ぶ)と別
    - GPUで用いたいデータは、デバイスメモリにコピーしておく必要
  - GPU内のたくさんの「コア」が並列に動作する。一つ一つのコアは非力で、単純な動作向きだが、力を合わせると高速な処理が可能
    - 今回使うTesla P100 GPUは3584 コア



GPUの計算速度の威力

int配列Aの全要素を2倍する計算時間をTSUBAME3.0上で測ってみた。配列長さは2^{30}

- CPU版 (C言語で書かれたarray.c)
  → 0.62秒かかった
- GPU版 (「CUDA」で書かれたarray.cu)
  → 0.016秒。CPU版より約39倍も速い！！

なぜ？ GPU中のたくさんのコアに分業させたから

ただし・・・

- 3584倍は無理。1つ1つはCPUよりかなり劣る
- 上の時間には、データコピー時間は含んでいない。
  含むと0.85秒となり、CPU版より遅い

プログラミング言語CUDA

- NVIDIA GPU向けのプログラミング言語
- CやC++言語＋GPU用拡張機能
- nvccコマンドを用いてコンパイル
  - ソースコードの拡張子は.cu

  例: array.cu をコンパイルして、実行ファイルarrayを作るコマンド

```
nvcc –o array –O2 –arch sm_60 array.cu
```

最近のGPU機能を使うための
おまじない(通常つけたほうがよい)

C言語のプログラムをCUDAにするには

だいたい以下のような考え方

- プログラムのうち、GPU上で動かしたい場所を決めて、別関数にする
  - GPUカーネル関数と呼ぶ
  - もともと時間のかかるループだったところがねらい目？
- GPUカーネル関数を、多数のスレッドで動くようにする
  - 「並列化」と呼ぶ。最大の難関！
- GPUの中で使いたいデータを、正しい位置へコピー
  - GPUカーネル関数実行の前に、メモリ確保 (cudaMalloc)や、データコピー (cudaMemcpy)をしておく
  - 計算後のデータがCPU側で必要なら、GPUからCPUへcudaMemcpy
- 動かしてみて、正しく動くか、速く動くか確認 ⇒ さらなる工夫へ！

サンプルプログラム(C版)

array.c: int配列の内容を2倍するプログラム

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

#define N ((long)1024*1024*1024) /* 配列の長さ、2の30乗 */

int main(int argc, char *argv[])
{
  long i;
  int *A;

  A = (int *)malloc(sizeof(int)*N); /* 配列Aの領域確保 */
  for (i = 0; i < N; i++) {  A[i] = i;  } /* Aの内容を初期化 */
  for (i = 0; i < N; i++) {  A[i] *= 2; } /* Aの内容を2倍 */

  return 0;
}
```

サンプルプログラム(CUDA版)

array.cu

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

#define N ((long)1024*1024*1024)
#define BS 1024

/* GPUカーネル関数の定義 */
__global__ void mul2(int *DA)
{
   long i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i >= N) return;
   DA[i] *= 2;   /* GPU上配列の中身を2倍 */
   return;
}

int main(int argc, char *argv[])
{
   long i;
   int *A;     /* ホストメモリ用のポインタ */
   int *DA;   /* デバイスメモリ用のポインタ */

   A = (int *)malloc(sizeof(int)*N); /* 配列Aの領域確保 */
   for (i = 0; i < N; i++) {  A[i] = i;  } /* Aの内容を初期化 */

   /* 配列A (GPU) の領域確保 */
   cudaMalloc((void**)&DA, sizeof(int)*N);

   /* ホストメモリからデバイスメモリへコピー */
   cudaMemcpy(DA, A, sizeof(int)*N, cudaMemcpyDefault);

   /* GPUカーネル関数呼び出し */
   mul2<<<(N+BS-1)/BS, BS>>>(DA);

   /* デバイスメモリからホストメモリへコピー */
   cudaMemcpy(A, DA, sizeof(int)*N, cudaMemcpyDefault);

   return 0;
}
```

GPUで動作する部分

長いプログラムになりましたが、
あせらず理解しましょう

array.cuのCUDA化の方針

- プログラムのうち、GPU上で動かしたい場所を決めて、別関数にする

➔今回は、配列Aを2倍にするところをGPUで行うことにする
　配列初期化はCPUのままとする

- GPUカーネル関数を、多数のスレッドで動くようにする

➔行うべき「仕事」の個数は$N=2^{30}$個ある
　N個の(超多数)スレッドを用い、1スレッドは1個の仕事を担当することにする

- GPUの中で使いたいデータを、正しい位置へコピー

➔GPUから使う必要のあるデータは、配列A
　また、計算が終わったら、CPUに戻すことにする

array.cu の流れ (CPU側その1)

```
int main(int argc, char *argv[])
{
   long i;
   int *A;     /* ホストメモリ用のポインタ */
   int *DA;   /* デバイスメモリ用のポインタ */


   A = (int *)malloc(sizeof(int)*N); /* 配列Aの領域確保 */


   for (i = 0; i < N; i++) {  A[i] = i;  } /* Aの内容を初期化 */
```

実行はmain関数から、
CPU上で始まる

このDAは後で使う

配列Aのためのメモリ領域を
mallocで準備
→ ホストメモリ上に確保される

このサンプルでは、Aの初期化
はCPUで行う

array.cu の流れ(CPU側その2)

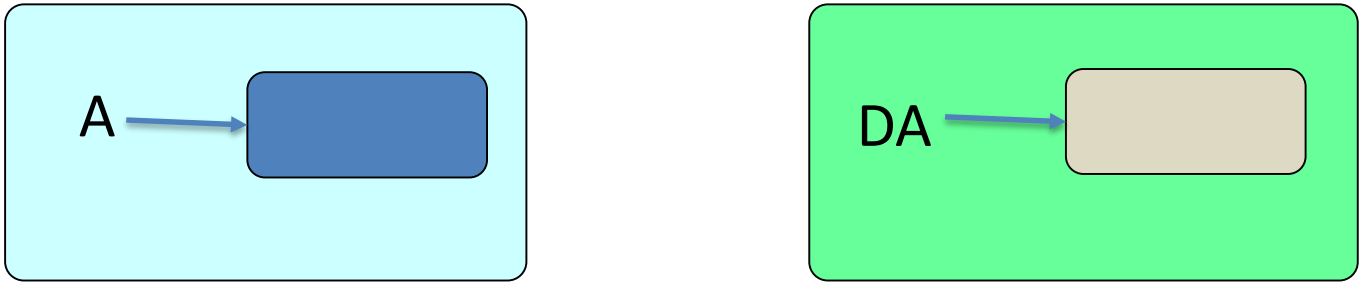
GPU上で計算を行いたいが、その前にデバイスメモリ上のデータ(ここでは配列A)の準備が必要！
まずは領域確保

```
/* 配列A (GPU) の領域確保 */
cudaMalloc((void**)&DA, sizeof(int)*N);
```

結果のポインタはここに入る

確保したいサイズをバイト単位で

➔ これを実行すると、DAは
「sizeof(int)*N」バイトの大きさの、デバイスメモリ上の領域を指す



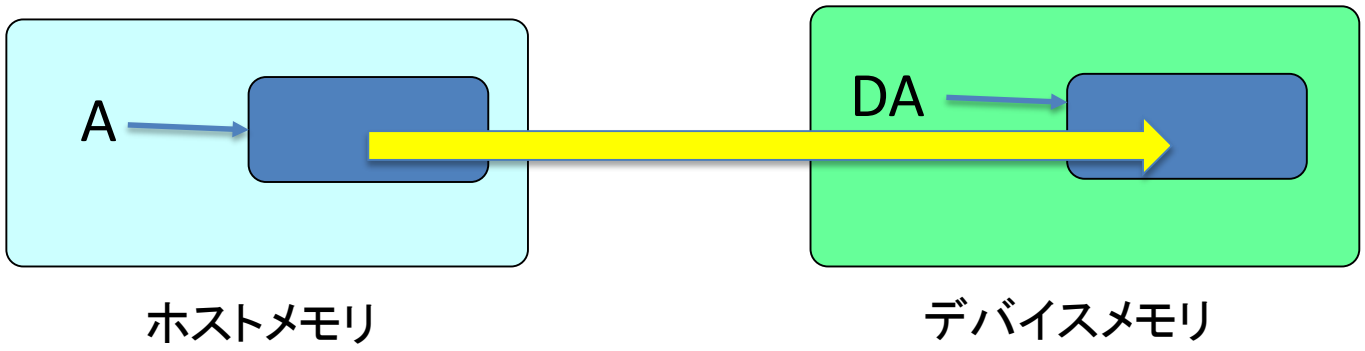
ホストメモリ　　デバイスメモリ

注意1 まだ、DAの中身は無効なデータ
注意2 CPU側からは、DAの中身には触れない。
DA[i] = … などしてはダメ！

array.cu の流れ (CPU側その3)

次に、CPU上で初期化しておいたAのデータをDAへコピー

```
/* ホストメモリからデバイスメモリへコピー */
cudaMemcpy(DA, A, sizeof(int)*N, cudaMemcpyDefault);
```



cudaMemcpyの引数は4つ

1. コピー先
2. コピー元
3. 何バイトコピーしたいか
4. とりあえず、cudaMemcpyDefault でok

ここまでで、
やっとGPUが動ける
準備ができた！

array.cu の流れ (CPU側その4)

いよいよGPU上で動く関数を呼び出そう !!

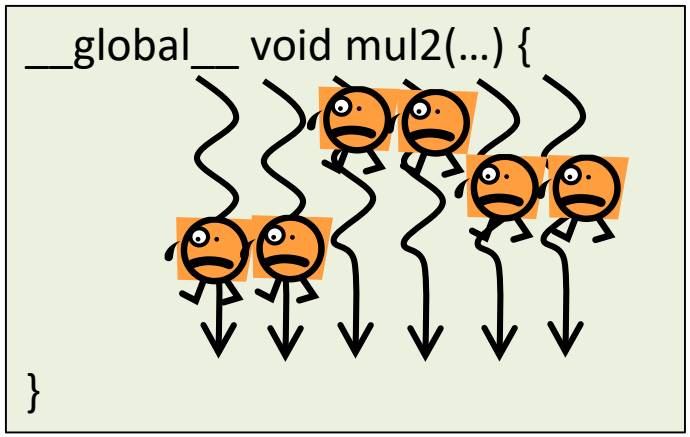
```
/* GPUカーネル関数呼び出し */
/* なお、このプログラムではBSは1024 */
mul2 <<<(N+BS-1)/BS, BS)>>> (DA);
```

呼びたい関数名

いくつのスレッドで
関数を実行するか

渡したい引数

GPU上で、
たくさんのスレッドが
mul2関数を実行



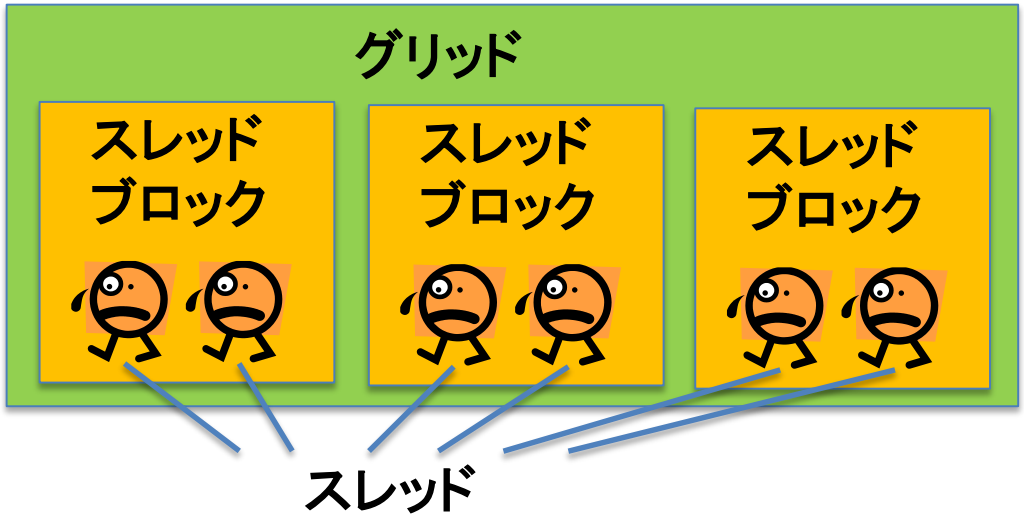
上の例はちょっとややこしいので、次ページではより簡単な例を考えましょう

スレッド数の指定について

GPUカーネル関数を実行する全部のスレッドたちをグリッドと呼ぶ

- グリッド = 複数のスレッドブロック
- スレッドブロック = 複数のスレッド

たとえば、学年のなかに
クラスがあって、
クラスの中に生徒がいる




```
func<<<3, 2>>>(a);
```

スレッドブロック
の個数

スレッドブロックあたりの
スレッドの数

この例では3x2=6スレッド
がGPU上で動作する！
全員がfunc関数を実行

スレッド数をどう決めればよい？(1)

GPUを動かすとき、スレッドブロック数と(ブロックあたり)スレッド数の両方を決める必要がある・・・

[ステップ1] 全スレッド数を決めよう

- できるだけ、独立に計算可能なものは別々のスレッドに担当させる
- 全スレッド数は、とても大きくても(100万、1億とか)よい!
  - GPU内のコア数は3584だが、それよりずっと大きくてもok

array.cuでは、$N=2^{30}$個のスレッドを作ってしまう。
1スレッドが1要素を計算する

スレッド数をどう決めればよい？(2)

[ステップ2] ブロックあたりスレッド数を決めよう

- たいてい、1024でよい
  - CUDAのルールで1以上1024以下と決められている
  - 他の数でも動くが、小さすぎると動作が遅い

➔ array.cu でもBS=1024にした

[ステップ3] スレッドブロック数を決めよう

- ステップ1の数を、ステップ2の数で割ればよい
  - ただし、割り切れないときのために、切り上げしておくのがおすすめ

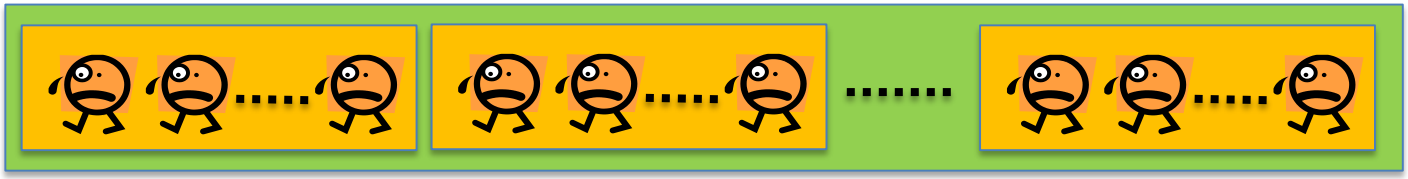
C言語の整数割り算 N/BS は、切り捨てになる
(N+BS-1)/BS で切り上げになる

array.cuではN=2^{30}, BS=1024で、もともと
割り切れるので、この2つの式は同じですが・・・

結局、array.cuでは `mul2 <<<(N+BS-1)/BS, BS)>>> (DA);`

array.cu の流れ (GPUカーネル関数その1)

GPU上で、超多数のスレッドがmul2関数を実行開始する



GPUカーネル関数の印

```
__global__ void mul2(int *DA)
{
    long i = blockIdx.x * blockDim.x + threadIdx.x;
                :
```

デバイスメモリ上のポインタが来ているはず

自分の通し番号を計算

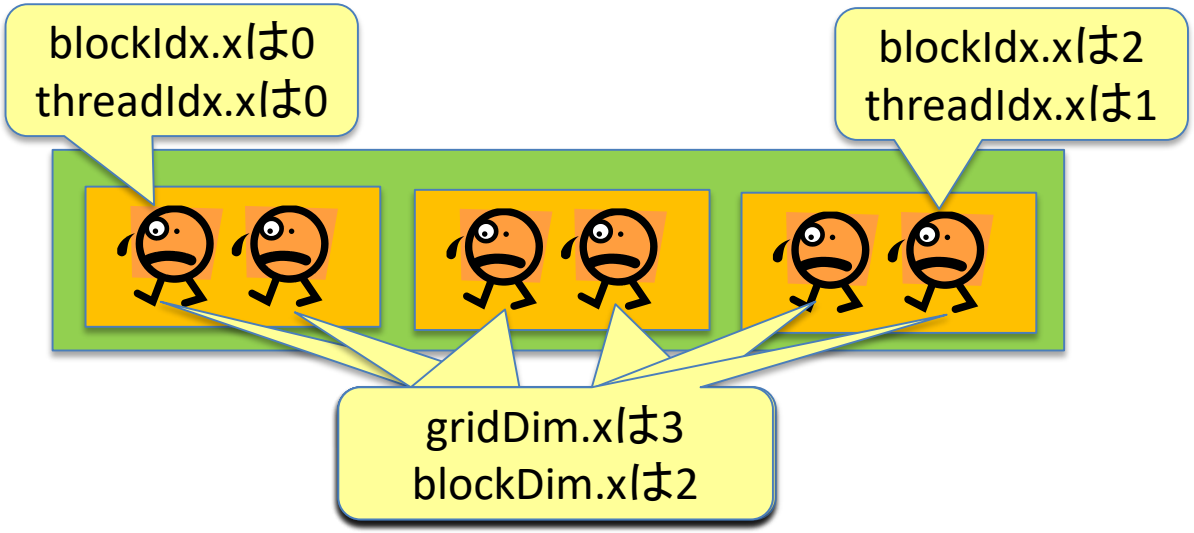
- みんなが同じコードを実行 ➔ 自分が「誰か」わかっていないと、そもそも分業できない

array.cuでは、通し番号がiのスレッドに、
A[i]の計算をさせることにする

スレッドの番号を知る

自分の番号は、GPUカーネル関数内で、特別な変数を読むと分かる

- gridDim.x: グリッドにいくつスレッドブロックがあるか
- blockIdx.x: 自分が何番のスレッドブロックにいるか
- blockDim.x: スレッドブロックにいくつスレッドがあるか
- threadIdx.x: 自分がスレッドブロック内で何番のスレッドか



※ 番号の値は0からはじまる

※ ～Dimはだれが読んでも同じ値

結局、一番役に立つのは全体での「通し番号」
→ blockIdx.x * blockDim.x + threadIdx.x　で求めることができる

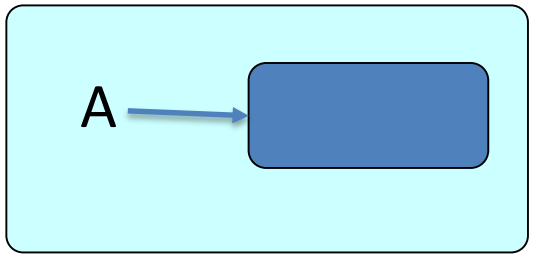
array.cu の流れ (GPUカーネル関数その2)

配列サイズを超える番号だったら何もしない (Nが1024で割り切れないときの対策)

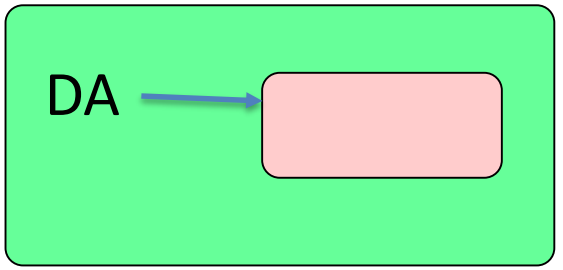
```
    if (i >= N) return;

    DA[i] *= 2;   /* GPU上配列の中身を2倍 */

    return;
}
```

実質的な計算はこれだけ

array.cuでは、1スレッドは1要素だけ計算 → forループがいらなくなった



ホストメモリ



デバイスメモリ

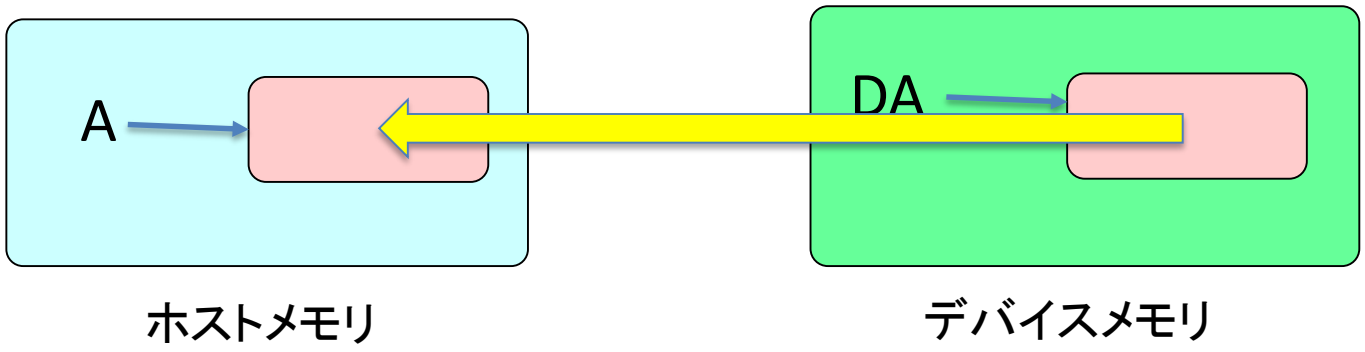
ここまでで、デバイスメモリ上の配列DA全体が2倍された

array.cu の流れ(CPU側その5、これで終了！)

GPUカーネル関数からmain関数へ戻ってきた
array.cuでは、計算結果をホストメモリのAにコピーする

```
  /* デバイスメモリからホストメモリへコピー */
  cudaMemcpy(A, DA, sizeof(int)*N, cudaMemcpyDefault);
  /* 必要があれば、Aを出力などに利用(略) */
}
```

前のcudaMemcpyと比べて
コピー先とコピー元が反対



次のステップへ

ここまでで、CUDAプログラミングの基本が終了
応用編では:

基本編の例では、各要素の計算が完全に独立だった
→ 配列の最大値などは、かなり難しい(サンプルプログラム有)

GPU上で、隣のスレッドどうしは、同時に配列の近い場所にアクセスすると速い (コアレスドアクセス)

GPU上でif文で分岐すると非効率になることがある

不要なcudaMemcpyは削ったほうがよい