GPUコンピューティング No.10

Shared メモリの利用

東京工業大学　学術国際情報センター

青木　尊之

変数型の Qualifier

__shared__	shared memory 上に確保される。 block単位のthread実行中のみ確保される。 block内のthreadからしかread/write不可
__constant__	constant memory 上に確保される。 プログラムが終了するまで確保される。 全てのthreadからreadでき、CPU側からは APIを通じて write 可。constant cache が有効。
__device__	global memory の領域に確保される。 プログラムが終了するまで確保される。 全てのthreadからアクセスでき、CPU側 からはAPIを通じて read/write 可。

Shared メモリの確保(1/2)

KERNEL 関数の中で

静的に確保　　__shared__ double fs[1024];

実行時に確保　　extern __shared__ double fs[];

<<< Dg, Db, 1024, 0 >>> の第3引数でサイズを指定。

各Streaming Multiprocessor 当たり(物理的に存在)(on-chip memory)

16 kB　　// GF100 Core より前

16kB/48kB　　// GF100 Core 以降

これを超えると、静的な場合はコンパイルエラーが出る。
実行時確保は(勿論)正常に実行できない。

Shared メモリの確保(2/2)

GF100 以降で、

L1 cache 48kB + shared メモリ 16 kB にする API
cudaThreadSetCacheConfig(cudaFuncCachePreferL1);

- L1 cache 16kB + shared メモリ 48 kB にするには、
  cudaThreadSetCacheConfig(cudaFuncCachePreferShared);

nvcc で Compile する際、-Xptxas –dlcm=cg とすると L1 cache を使わなくなるが、shared メモリ のサイズは48 kB のまま変わらない。

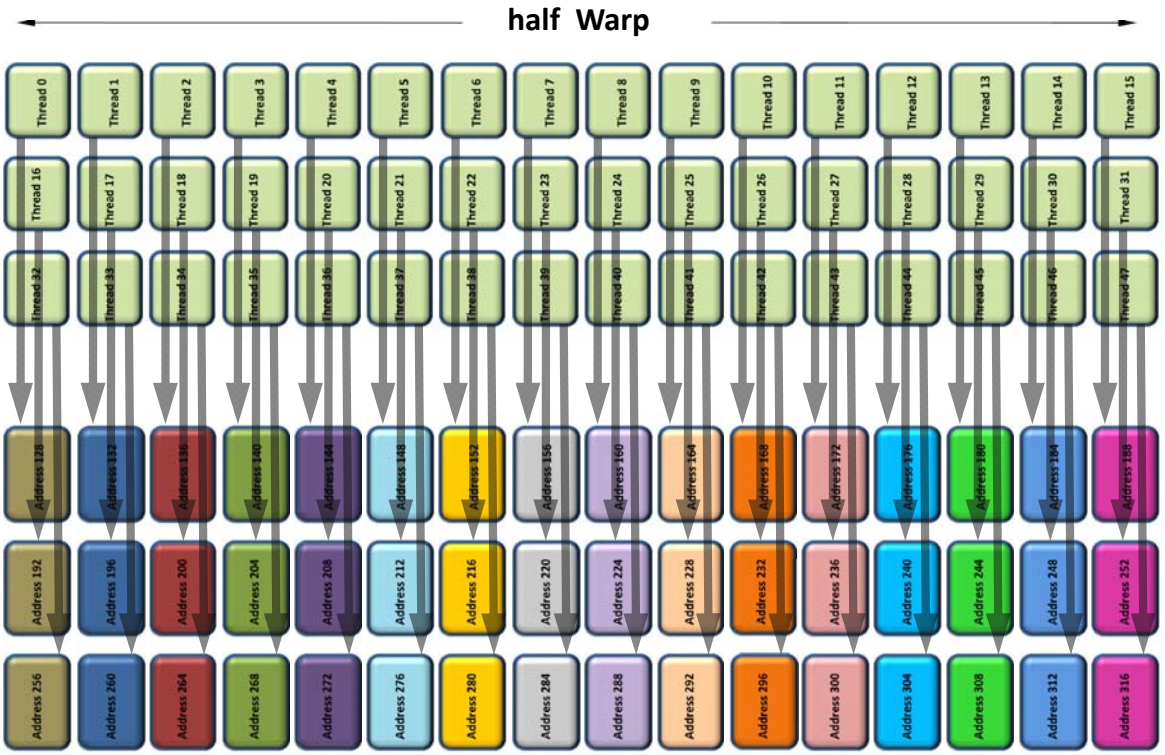
Shared メモリの性質

- 非常に高速なアクセスが可能
- block 内のthread でデータの交換が可能。
- L1 cache がない場合は、Software Managed Cache として利用価値が高い。

※ Bank Conflict を起こすと、性能が著しく低下。
※ kernel 関数の引数も、shared メモリ上に展開。

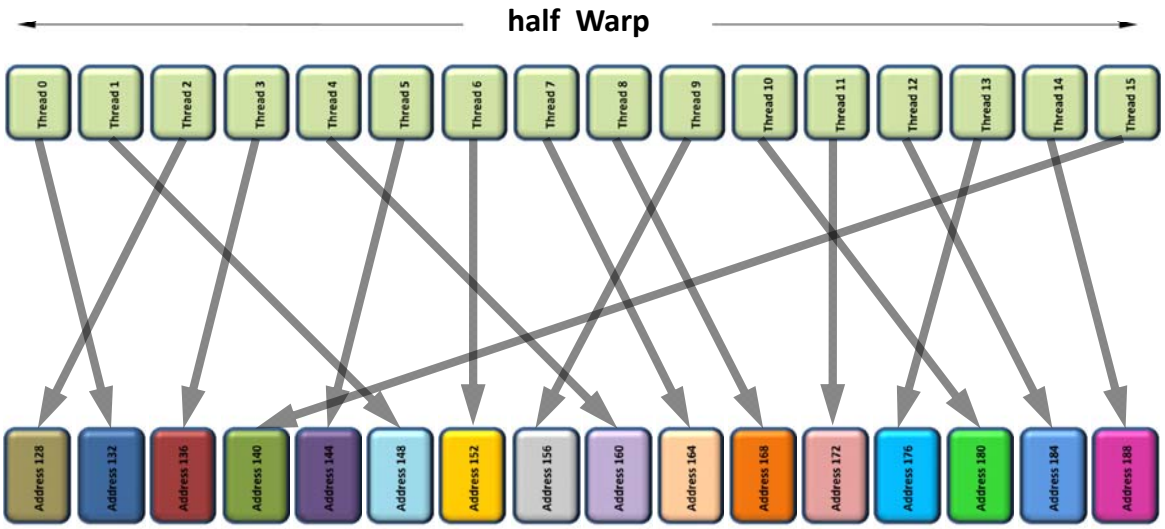
Shared Memory

Bank Conflict Free Pattern



Shared Memory

Bank Conflict Free Pattern

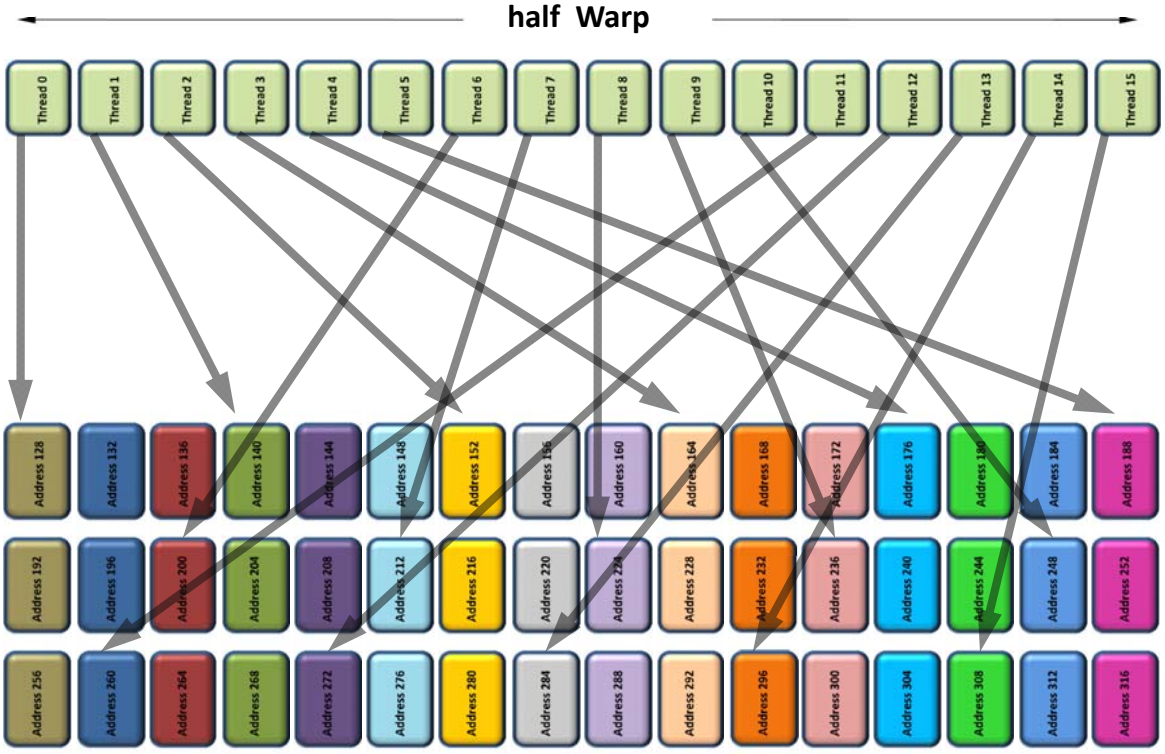


Random Access without Bank Conflict

Banc_conflict_2

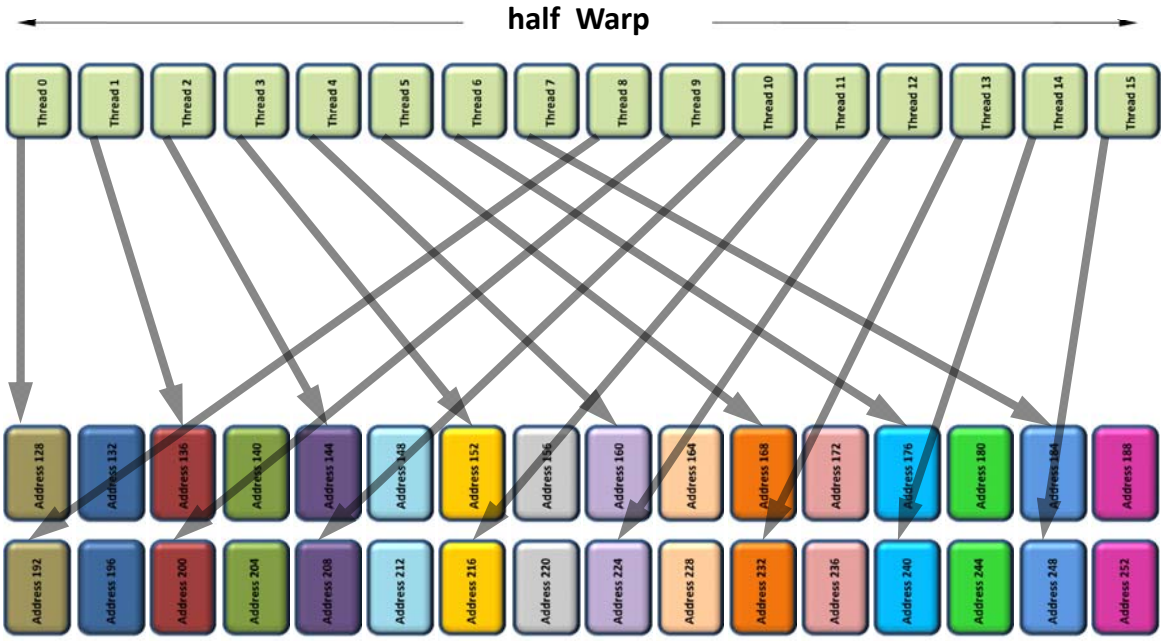
Shared Memory

Bank Conflict Free Pattern



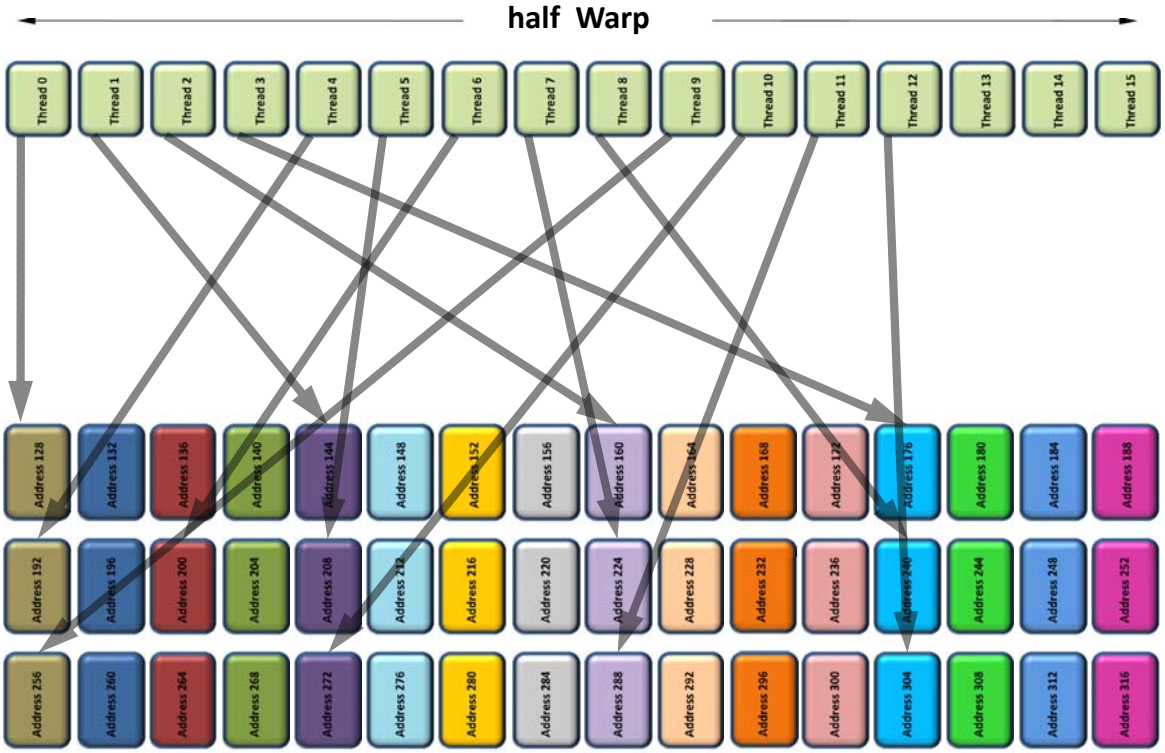
Shared Memory

Bank Conflict Pattern



Shared Memory

Bank Conflict Pattern



Sharedメモリの利用（Reduction）

Global memory の配列の中の、最大値、最小値、総和等を求めたり、Sort などを行う必要がある。

CPU では容易であり、様々なアルゴリズムが開発されているが、GPU では data parallel に基づいた thread 並列であるため、

Parallel Reduction

を行う必要がある。

配列要素の総和

Global memory の配列の中の、最大値、最小値、総和等を求める方法は基本的に同じであるため、ここでは総和を求める。

Reduction は配列要素間の比較を行う必要があるが、thread 間で情報交換を行うには block 内では shared memory を使い、grid 全体では global memory を使う。

global memory の頻繁なアクセスを行うと、GPU本来の性能が引き出せないので、可能な限り shared memory を利用する。

総和計算の問題設定

n（～1024x1024x16（double で約134.2MB））個の要素を持つ配列 A[n] に対して、総和（平均値）を求める。

初期に a_h[n] に 0～0.99999999 の疑似乱数を設定し、CPU での総和計算と GPU での総和計算の速度と結果を比較する。

初期設定：

```
srand(12131);
for(i = 0; i < n; i++) a_h[i] = (double) rand()/RAND_MAX;
```

GPU の場合は、その後 cudaMemcpy で GPU にa_h[n] を転送。

CPU での総和計算


```
double sum = 0.0;
for(i = 0; i < n; i++)  sum += a_h[i];
```

Intel Xeon X5670 1 coreで、n=1024x1024x16 のとき、

Data size = 134.2 [MB]
Elapsed time = 25.1 [msec]
Memory Bandwidth = 5.4 [GB/sec]
Sum = 0.4998859488

GPU での総和計算

block 分割（n/2048 個のblock）

kernel 関数 gpu_summation
の中に、shared memory の確保

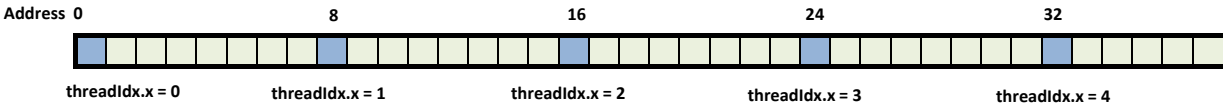
```
__shared__  double  fs[2048];
```

shared memory へのデータの読み込み

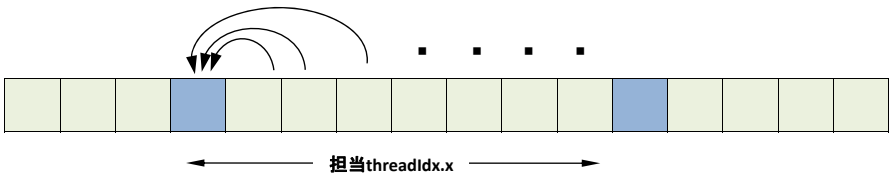
```
#pragma unroll
for(i = 0; i < 2048/256; i++) {
    js = 2048/256*threadIdx.x + i;   j = 2048*blockIdx.x + js;
    fs[js] = A[j];
}
```


GPU での総和計算

shared memory : fs[2048]



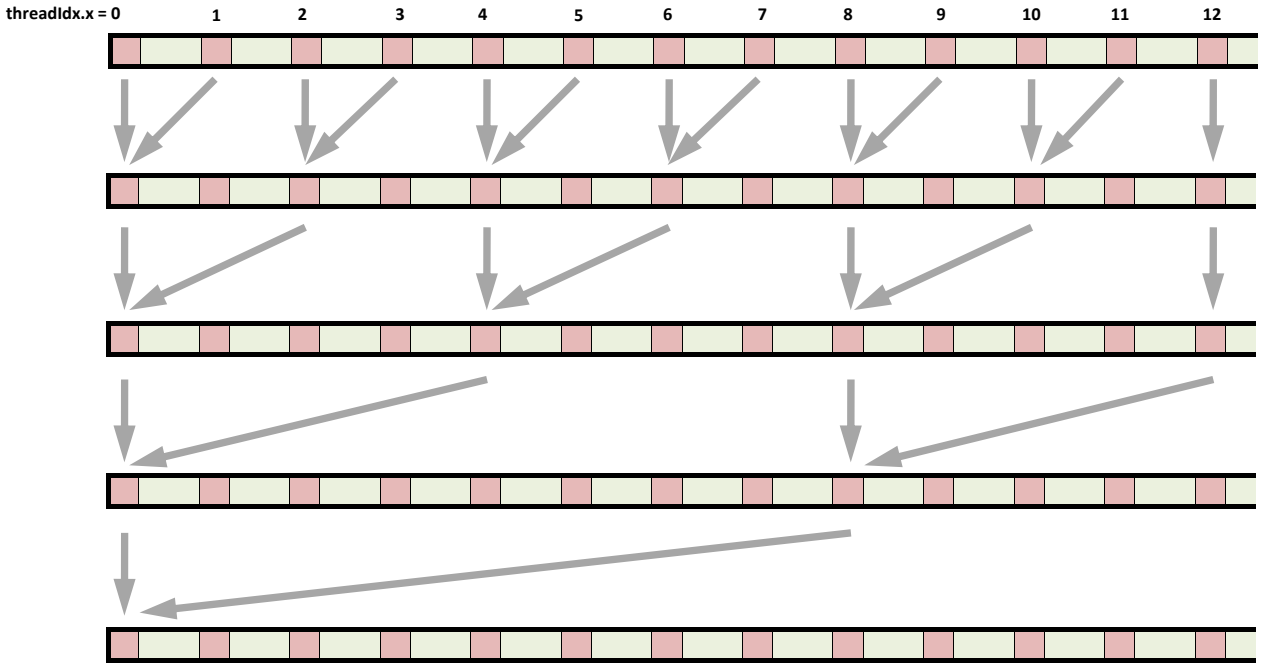
各thread が担当する8要素の総和を計算し、最初の要素の shared memory に格納する。




```
for(i = 1; i < 8; i++)   fs[8*threadIdx.x] += fs[8*threadIdx.x + i];
```

GPU での総和計算

shared memory : fs[2048]



block 内の Reduction


```
unsigned int t = 8*threadIdx.x;

for (unsigned int stride = 1;  stride < blockDim.x;  stride *= 2) {
      __syncthreads();

     if (threadIdx.x % (2*stride) == 0)   fs[t] += fs[t + 8*stride];
}

If(threadIdx.x == 0)  B[blockIdx.x] = fs[0];
```

（block内の総和の保存）

GPU での総和計算

block 分割（n/2048 個のblock）

a_d[N]

Blockの総和計算後 Bd に Copy

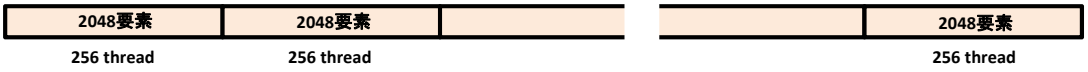
b_d[N/2048]

CPU側にCopyして同じ処理

```
sum = 0.0;  for(i = 0; i < n/2048; i++) sum += B[i];
```


繰り返し計算

b_d[N/2048]　block 分割（n/2048）/2048 個のblock）



Blockの総和計算後 c_d に Copy

c_d[N/(2048*2048)]

CPU側にCopyして同じ処理

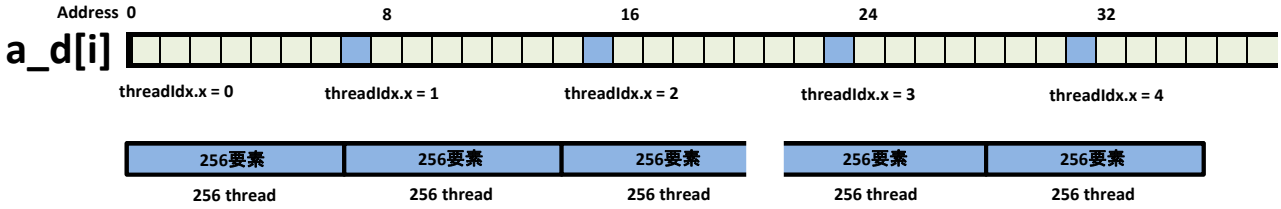
```
sum = 0.0;  for(i = 0; i < n/(2048*2048); i++) sum += c_h[i];
```

```
Elapsed time = 5.4[msec]
Memory Bandwidth =  24.9[GB/sec]

Sum = 0.4998859488
```

Source Code #13

幾つかの改善

Global memory からのデータ読込みを Coalesced Access にする。

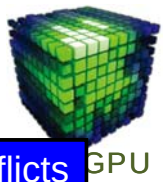



```
__shared__ double fs[2048];
```

shared memory へのデータの読み込み

```
#pragma unroll
for(i = 0; i < 2048/256; i++) {
   js = 2048/256*threadIdx.x + i;        j = 2048*blockIdx.x + js;
    js = 256*i + threadIdx.x;
   fs[js] = A[j];
}
```


Reduction計算の問題点



BAD: bank conflicts

BAD: Branch diverge

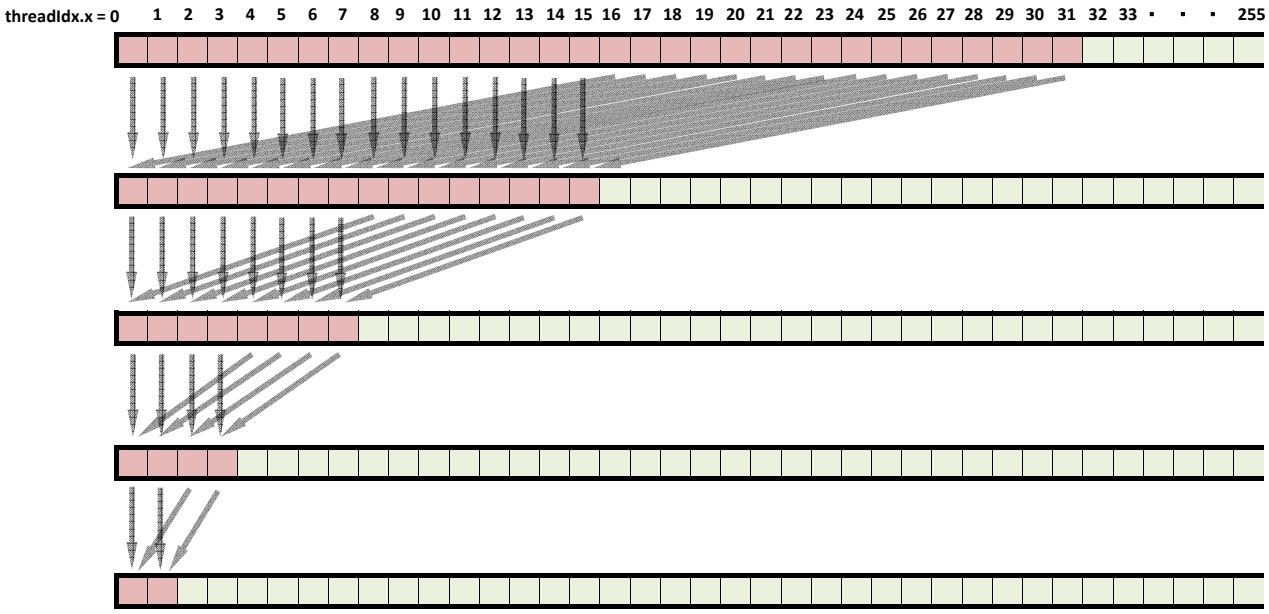
BAD: bank conflicts

```
int t = 8*threadIdx.x;

for(i = 1; i < 8; i++)  fs[t] += fs[t + i];

for (unsigned int stride = 1;
          stride < blockDim.x;  stride *= 2)
{
 __syncthreads();

 if (threadIdx.x % (2*stride) == 0)
          fs[t] += fs[t + 8*stride];
}

If(threadIdx.x == 0)  B[blockIdx.x] = fs[0];
```

（block内の総和の保存）

収集場所の変更

shared memory : fs[2048]



Reduction計算の改善


```
int t = threadIdx.x;

for(i = 1; i < 8; i++)  fs[t] += fs[t + 256*i];

for (int  stride = blockDim.x; stride > 1;  stride /= 2) {
  __syncthreads();

   if (t < stride/2) fs[t] += fs[t + stride];
}

if(t == 0)  B[blockIdx.x] = fs[0] + fs[1];
```

stride ≧ 16 までは、diverge せず、bank conflict free

レポート課題4

『総和計算の高速化』

期限：2013年 6月27日（木） 17:00
場所：学術国際情報センター・国際棟 1F のI7-3メールボックスに提出。または Subject:「総和計算の高速化」とし、メールの宛先
gpu_report2013@sim.gsic.titech.ac.jp
に上記の内容を pdf ファイルとして提出すること。

ソースコード Source Code #13 のbank conflict や branch diverge 等を改善することで高速化を試みよ。レポートには何をどのように変更して、どのくらい高速化できたかを記述すること。