Introduction to CUDA Parallel Programming
CUDA平行計算導論

https://ceiba.ntu.edu.tw/1092Phys8061_CUDA

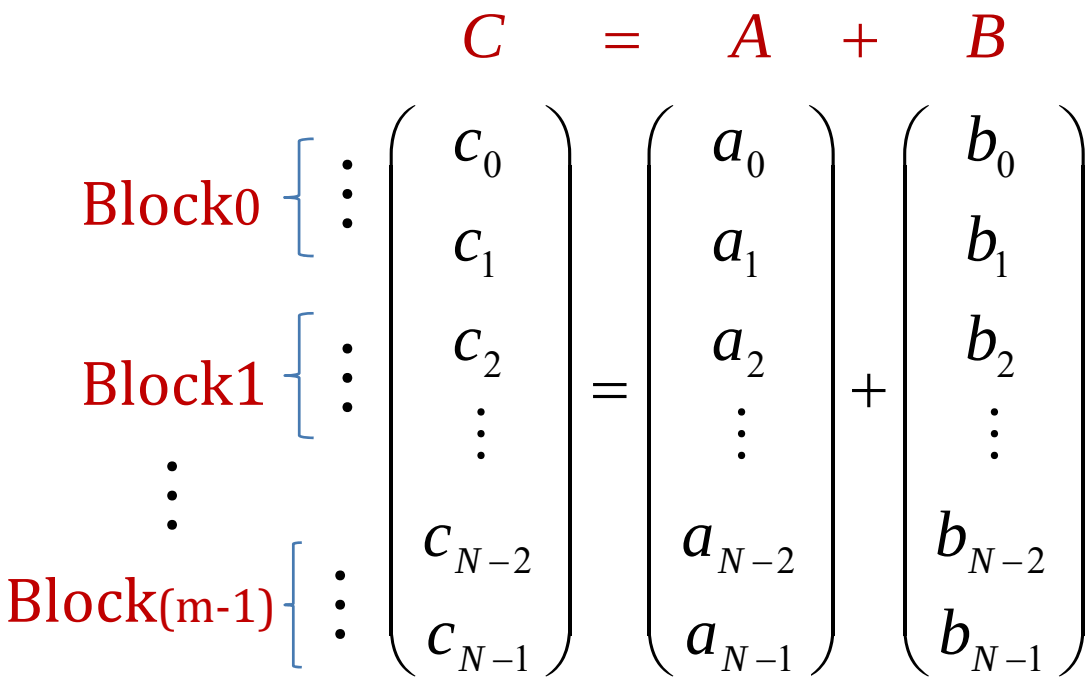
Professor Ting-Wai Chiu (趙挺偉)
Email: twchiu@phys.ntu.edu.tw
Physics Department
National Taiwan University

This lecture will cover:

- Vector Addition of Arbitrarily Long Vectors
- Dot Product of 2 Vectors using Parallel Reduction with Shared Memory

Vector Addition of Arbitrarily Long Vectors (1)

Recall that for $N <$ blockDim.x*gridDim.x



$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-2} \\ c_{N-1} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-2} \\ a_{N-1} \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{N-2} \\ b_{N-1} \end{pmatrix}$$

Use 1D grid and 1D blocks

vector index: i = blockDim.x * blockIdx.x + threadIdx.x

deviceQuery (GTX1060)

```
Device 0: "GeForce GTX 1060 6GB"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 6078 MBytes (6373179392 bytes)
  (10) Multiprocessors, (128) CUDA Cores/MP:     1280 CUDA Cores
  GPU Max Clock rate:                            1759 MHz (1.76 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536),
                                                 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  ...
  ...
```

Vector Addition of Arbitrarily Long Vectors (3)

The max. no. of threads per block is $2^{10}=1024$

The max. no. of blocks in a grid is $2^{31}-1=2147483647$

The max. no. of threads in a grid is $2^{10}(2^{31}-1)<2^{41}$

However, the optimal block size may be as small as 16.

In this case, the total number of threads is 2^{35} in the entire grid.

Vector Addition of Arbitrarily Long Vectors (3)

The max. no. of threads per block is $2^{10}=1024$

The max. no. of blocks in a grid is $2^{31}-1=2147483647$

The max. no. of threads in a grid is $2^{10}(2^{31}-1)<2^{41}$

However, the optimal block size may be as small as 16.

In this case, the total number of threads is 2^{35} in the entire grid.

For lattice QCD with domain-wall fermion on the $128^4\times16$ lattice, field vectors have the size

$128^4\times16\times3(\text{color})\times4(\text{Dirac})\times4(\text{complex and double})$

$\approx 2^{38}$ single-precision floating-point numbers.

Vector Addition of Arbitrarily Long Vectors (3)

The max. no. of threads per block is $2^{10}=1024$

The max. no. of blocks in a grid is $2^{31}-1=2147483647$

The max. no. of threads in a grid is $2^{10}(2^{31}-1)<2^{41}$

However, the optimal block size may be as small as 16.

In this case, the total number of threads is 2^{35} in the entire grid.

For lattice QCD with domain-wall fermion on the $128^4\times16$ lattice, field vectors have the size

$128^4\times16\times3$(color)$\times4$(Dirac)$\times4$(complex and double)

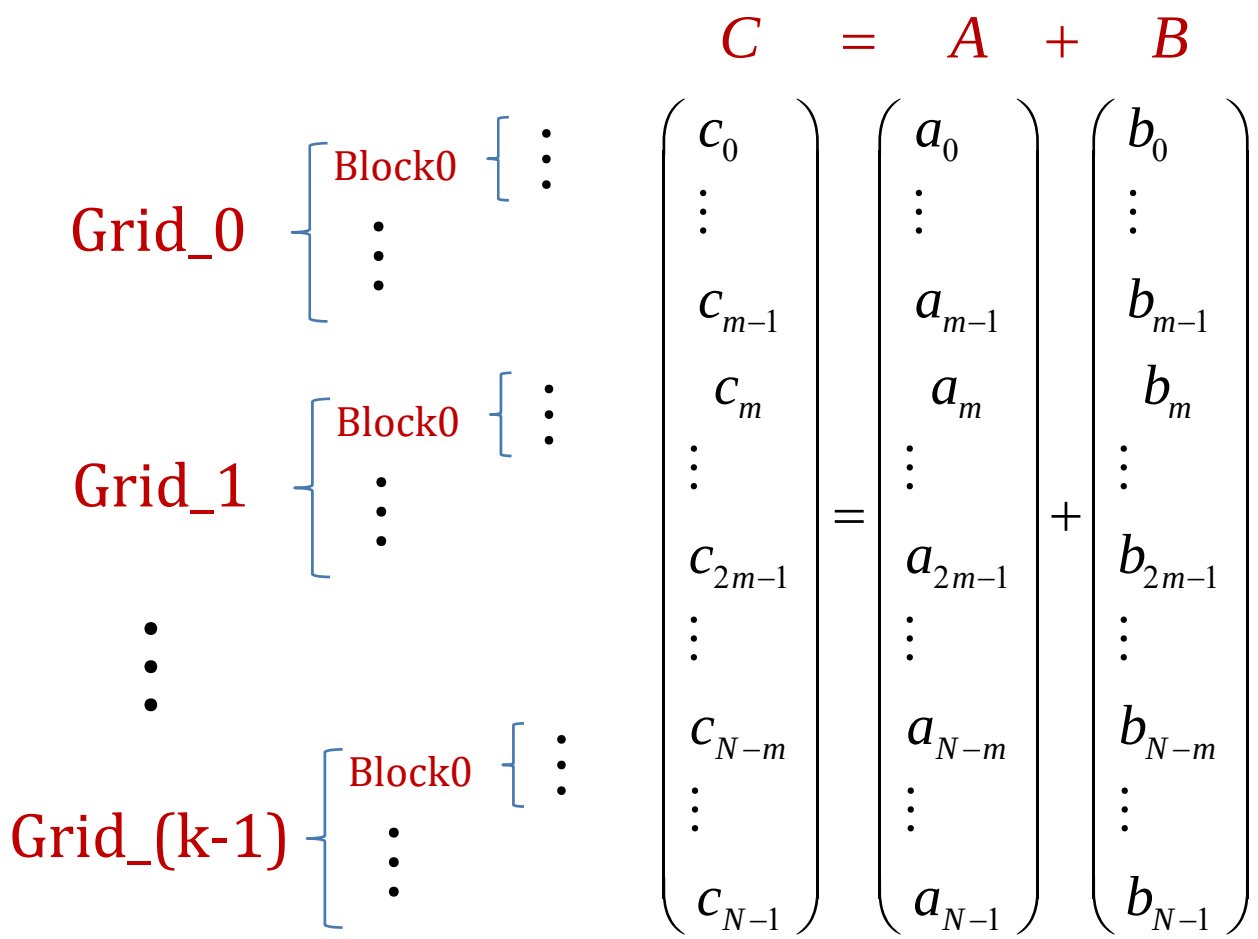
$\approx 2^{38}$ single-precision floating-point numbers.

Also there are always some reasons why one needs to use the block size and the grid size less than their upper bounds.

Vector Addition of Arbitrarily Long Vectors (5)

```
// Device code fragment

__global__ void VecAdd(const float* A, const float* B,  float* C,  int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

//     if (i < N)                    // only for N < blockDim.x*gridDim.x
//       C[i] = A[i] + B[i];

    while (i < N) {
       C[i] = A[i] + B[i];
        i += blockDim.x * gridDim.x;     // go to the next grid
    }
    __syncthreads();
}

// The complete host+device code at twqcd80:/home/cuda_lecture/vecAdd_1GPU_anyN
```

Vector Addition of Arbitrarily Long Vectors (6)



Note that only one Grid is executed at any time !

Vector Dot Product (1)

For vectors with real number elements

$$A^T \cdot B = a_0 b_0 + a_1 b_1 + \cdots + a_{N-1} b_{N-1}$$

In general, for vectors with complex number elements

$$A^\dagger \cdot B = a_0^* b_0 + a_1^* b_1 + \cdots + a_{N-1}^* b_{N-1}$$

Vector Dot Product (1)

For vectors with real number elements

$$A^T \cdot B = a_0 b_0 + a_1 b_1 + \cdots + a_{N-1} b_{N-1}$$

In general, for vectors with complex number elements

$$A^\dagger \cdot B = a_0^* b_0 + a_1^* b_1 + \cdots + a_{N-1}^* b_{N-1}$$

Use 1D grid and 1D block

vector index: i = blockDim.x * blockIdx.x + threadIdx.x

Vector Dot Product (1)

For vectors with real number elements

$$A^T \cdot B = a_0 b_0 + a_1 b_1 + \cdots + a_{N-1} b_{N-1}$$

In general, for vectors with complex number elements

$$A^\dagger \cdot B = a_0^* b_0 + a_1^* b_1 + \cdots + a_{N-1}^* b_{N-1}$$

Use 1D grid and 1D block

vector index: i = blockDim.x * blockIdx.x + threadIdx.x

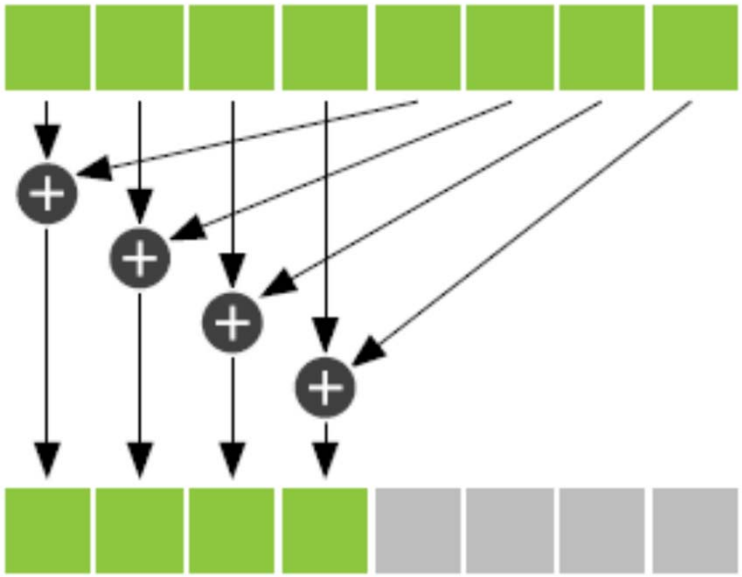
It is trivial to obtain $a_0 b_0,\ a_1 b_1,\ a_2 b_2, \cdots$

But the question is how to sum all of them.

Use Parallel Reduction with Shared Memory

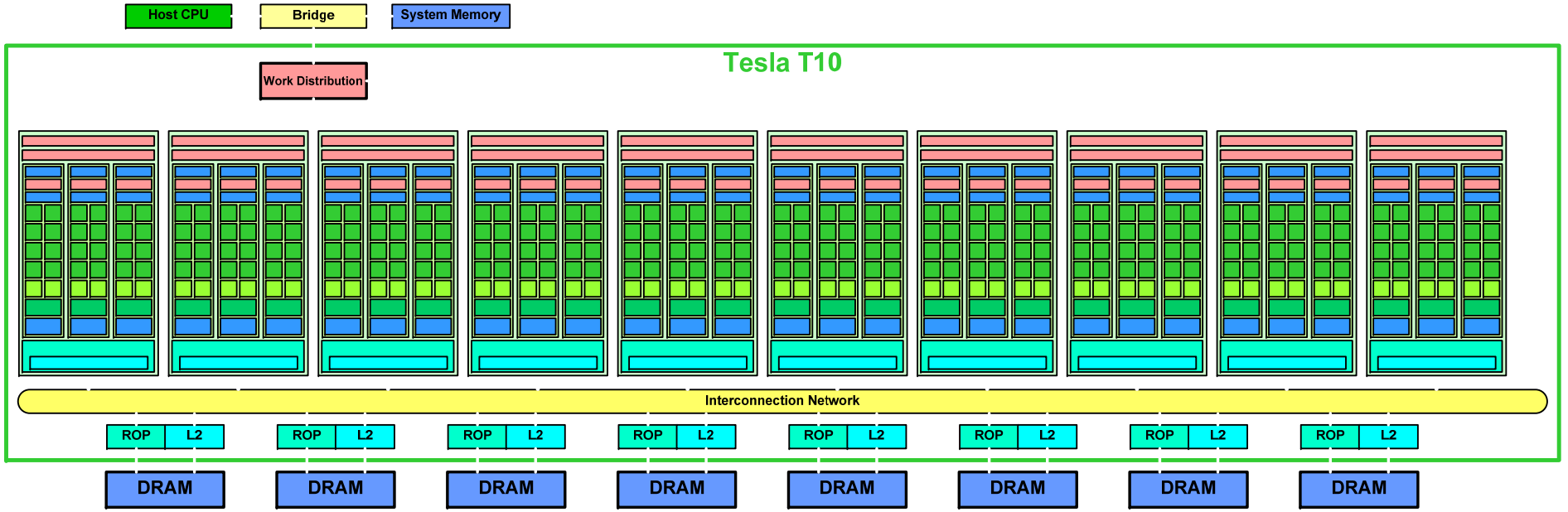
If the partial sums of a_0b_0, a_1b_1, a_2b_2, $\cdots$ are saved in the shared memory of each block, then they can be accessed by each thread in the block, thus can be summed by Parallel Reduction.

__shared__ float cache[threadsPerBlock]; // declare shared memory

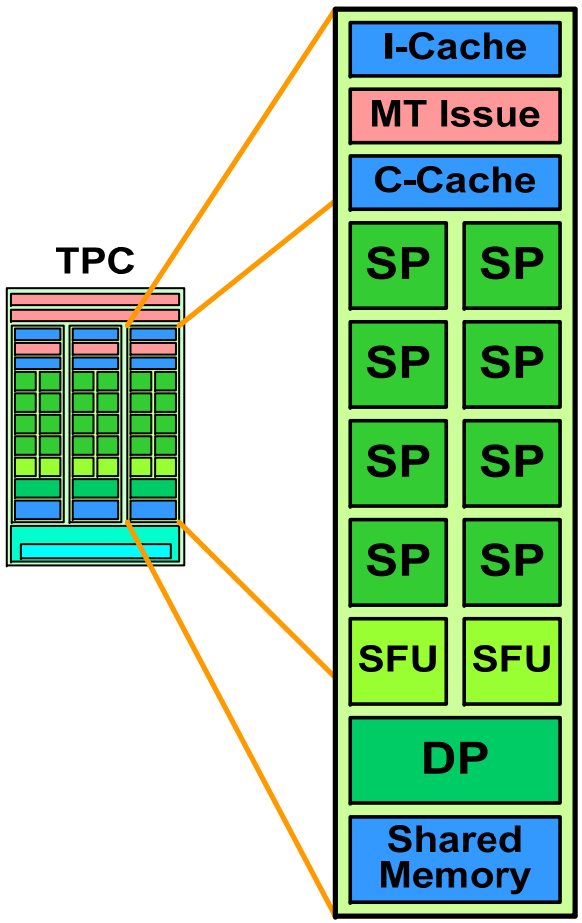


Tesla 1070 Architecture (2008)

- Massively parallel general computing architecture
- 30 Streaming multiprocessors @ 1.45 GHz with 1.0/4.0 GB of RAM
  - 1 TFLOPS single precision (IEEE 754 floating point)
  - 87 GFLOPS double precision



Streaming Multiprocessor of Tesla T10



- 8 SP Thread Processors
  - IEEE 754 32-bit floating point
  - 32-bit float and 64-bit integer
  - 16K 32-bit registers
- 2 SFU Special Function Units
- 1 Double Precision Unit (DP)
  - IEEE 754 64-bit floating point
  - Fused multiply-add
- Scalar register-based ISA
- Multithreaded Instruction Unit
  - 1024 threads, hardware multithreaded
  - Independent thread execution
  - Hardware thread scheduling
- 16KB Shared Memory
  - Concurrent threads share data
  - Low latency load/store

deviceQuery (GTX1060)

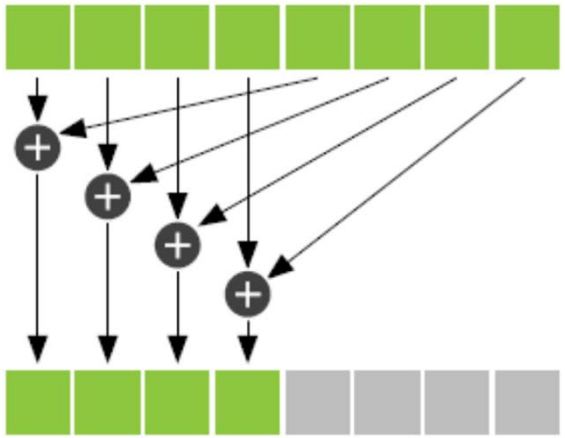
```
Device 0: "GeForce GTX 1060 6GB"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 6078 MBytes (6373179392 bytes)
  (10) Multiprocessors, (128) CUDA Cores/MP:     1280 CUDA Cores
  GPU Max Clock rate:                            1759 MHz (1.76 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536),
                                                 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  ...
  ...
```

Vector Dot Product (2)

```
__global__ void VecDot(const float* A, const float* B, float* C, int N)
{
   extern __shared__ float cache[];    //  its size is allocated  at runtime call
   int  i = blockDim.x * blockIdx.x + threadIdx.x;
   int  cacheIndex = threadIdx.x;
   float  temp = 0.0;
   while (i < N) {
      temp += A[i]*B[i];
      i += blockDim.x*gridDim.x;    // go to the next grid
   }
    cache[cacheIndex] = temp;       // set the cache value
     __syncthreads();
    int ib = blockDim.x/2;            // perform parallel reduction
    while (ib != 0) {
      if(cacheIndex < ib)  cache[cacheIndex] += cache[cacheIndex + ib];
      __syncthreads();
      ib /=2;
    }
   if(cacheIndex == 0)   C[blockIdx.x] = cache[0];
}
```

Vector Dot Product (3)

```
__global__ void VecDot(const float* A, const float* B, float* C, int N)
{
   extern __shared__ float cache[];
   int  i = blockDim.x * blockIdx.x + threadIdx.x;
   int  cacheIndex = threadIdx.x;
   float  temp = 0.0;
   while (i < N) {
      temp += A[i]*B[i];
      i += blockDim.x*gridDim.x;
   }
    cache[cacheIndex] = temp;
     __syncthreads();
    int ib = blockDim.x/2;
    while (ib != 0) {
      if(cacheIndex < ib)  cache[cacheIndex] += cache[cacheIndex + ib];
      __syncthreads();
      ib /=2;
    }
   if(cacheIndex == 0)   C[blockIdx.x] = cache[0];
}
```



Vector Dot Product (4)

```
int main(void)      // host (CPU) code fragment
{
   …
   int size = N * sizeof(float);
   int sb = blocksPerGrid * sizeof(float);
   …
   h_A = (float*)malloc(size);
   h_B = (float*)malloc(size);
   h_C = (float*)malloc(sb);   // result of  dot-product in each block
   …
   int sm = threadsPerBlock*sizeof(float);    // size of shared memory
   VecDot <<< blocksPerGrid, threadsPerBlock, sm >>>(d_A, d_B, d_C, N);
   cudaMemcpy(h_C, d_C, sb, cudaMemcpyDeviceToHost);

   float h_G=0.0;              //  add the partial sums of all blocks
   for(int i = 0; i < blocksPerGrid; i++)
      h_G += h_C[i];
}
// The complete host+device code at twqcd80:/home/cuda_lecture/vecDotProduct
```

Parallel Reduction is a Common Building Block for many Parallel Algorithms

It is used in

- Finding the trace of a matrix
- Finding the maximum/minimum of an array of numbers
- Matrix-vector multiplication
- ...

For a faster parallel reduction using the wrap shuffling, see
https://devblogs.nvidia.com/faster-parallel-reductions-kepler/

Remarks

- Arrays (int, float, double) with fixed sizes can be allocated in the shared memory. However, only the size of one array in the shared memory can be dynamically allocated at the runtime.

- The code fragment for parallel reduction:

```
blockDim.x = 1024;
int ib = blockDim.x/2;
while (ib != 0) {
   if(cacheIndex < ib)  cache[cacheIndex] += cache[cacheIndex + ib];
   __syncthreads();
   ib /=2;
 }
```

can be replaced by the following:

```
if(cacheIndex < 512) cache[cacheIndex] += cache[cacheIndex + 512]; __syncthreads();
if(cacheIndex < 256) cache[cacheIndex] += cache[cacheIndex + 256]; __syncthreads();
if(cacheIndex < 128) cache[cacheIndex] += cache[cacheIndex + 128]; __syncthreads();
if(cacheIndex < 64) cache[cacheIndex] += cache[cacheIndex + 64]; __syncthreads();
if(cacheIndex < 32) cache[cacheIndex] += cache[cacheIndex + 32]; __syncthreads();
if(cacheIndex < 16) cache[cacheIndex] += cache[cacheIndex + 16]; __syncthreads();
if(cacheIndex < 8) cache[cacheIndex] += cache[cacheIndex + 8]; __syncthreads();
if(cacheIndex < 4) cache[cacheIndex] += cache[cacheIndex + 4]; __syncthreads();
if(cacheIndex < 2) cache[cacheIndex] += cache[cacheIndex + 2]; __syncthreads();
if(cacheIndex < 1) cache[cacheIndex] += cache[cacheIndex + 1]; __syncthreads();
```