An Example in C

Add vector A and vector B to vector C.

```c
void add_vector (float *A, float *B, float *C, in N){
    for ( index = 0; index < N; index++ )
...
}

void main ( ) { ...
    add_vector (A1,B1,C1,N1);
...
}
```
Review: Programming with CUDA

An Example in CUDA

Add vector A and vector B to vector C.

```c
_global_ void add_vector (float *A, float *B, float *C, in N){
    ...
}

void main ( ) {
    add_vector <<<dimGrid, dimBlock>>> (A1,B1,C1,N1);
    ...
}
```

Kernel function to run on GPU

Main function to run on CPU
**Review: CUDA Thread View**

```
add_vector <<<dimGrid, dimBlock>>> (A1,B1,C1,N1);
```

- A kernel is executed as a grid of thread blocks. All threads share off-chip memory space.
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - Efficiently sharing data through a low latency shared memory
- Threads in a block is organized into warps, typically 32 threads/warp.
- A (half) warp is a SIMD thread group.
Review: SIMD (Single Instruction Multiple Data)


thread 0: \( C[0] = A[0] + B[0] \);
....

Execute at the same time.
Overview

- **Parallel Sum (also called as reduction)**
  - four optimization levels: divergence eliminations, shared memory bank conflicts, code specialization

- **Parallel Prefix Sum**
  - work efficiency and shared memory bank conflicts

- **Parallel Sort**
  - bitonic sort, merge sort, radix sort
Parallel Sum

Sequential code

sum = 0;
for ( i = 0; i < N; i++ )
    sum = sum + A[i];
Parallel Sum

Tree based implementation

An initial implementation

```cpp
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```
Parallel Sum

Any possible improvement on the initial implementation?

thread divergence due to control branch
Parallel Sum

- Second version -- reduced thread divergence

**initial implementation**

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

**after thread divergence removed**

```c
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```
Parallel Sum

Any possible improvement on the second version?

shared memory bank conflicts
Parallel Sum

Third version -- sequential addressing

Any possible further improvement?
Parallel Sum

Fourth version

reduce number of conditional checks and thread sync for warps

for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}
if (tid < 32) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}

Fifth version

halve the threads at the first iteration

unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
Parallel Sum

Sixth version -- code specialization with template

```c
if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
}
if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
    if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
}
if (tid < 32) {
    if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```

code in red evaluated in compile-time
Parallel Prefix Sum

Sequential implementation

```c
int PrefixSum[N];
PrefixSum[0] = 0;
for ( i = 1; i < N; i++ )
    PrefixSum[i] = PrefixSum[i-1] + A[i-1];
```

Complexity: $O(n)$
Parallel Prefix Sum

Initial parallel implementation

Complexity: $O(n \log(n))$

$n$ is the total number of elements in this array

[ Hillis and Steele 1986 ]
Parallel Prefix Sum

- Work efficient implementation

(A). The Up-Sweep (Reduce) Phase

Complexity: $O(n)$

[ Blelloch 1990 ]
Parallel Prefix Sum

- Work efficient implementation

(B). The Down-Sweep Phase

Complexity: $O(n)$
Parallel Prefix Sum

- Shared memory bank conflicts

```
int ai = offset*(2*thid+1)-1;
int bi = offset*(2*thid+2)-1;
ai += ai / NUM_BANKS;
bi += bi / NUM_BANKS;
temp[bi] += temp[ai]
```

Padding addresses every 16 elements removes bank conflicts.
Parallel Sort

Bitonic sort

A bitonic sequence is defined as a list with no more than one local maximum and no more than one local minimum.

Binary split: Divide the bitonic list into two equal halves. Compare-exchange each item on the first half with the corresponding item in the second half.

Result: Two bitonic sequences where the numbers in one sequence are all less than the numbers in the other sequence.
Parallel Sort

- Bitonic sort: many steps of bitonic split

Complexity: $O(n \log(n)^2)$
Parallel steps: $O(\log(n)^2)$
Parallel Sort

- Bitonic sort code in CUDA

```c
__global__ static void bitonicSort(int * values) {
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;
    // Copy input to shared mem.
    shared[tid] = values[tid];
    __syncthreads();
    // Parallel bitonic sort.
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        // Bitonic merge:
        for (unsigned int j = k / 2; j>0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if ((tid & k) == 0)  {
                    if (shared[tid] > shared[ixj])
                        swap(shared[tid], shared[ixj]);
                } else {
                    if (shared[tid] < shared[ixj])
                        swap(shared[tid], shared[ixj]);
                }
            }
        }
        __syncthreads();
    }
    // Write result.
    values[tid] = shared[tid];
}
```
Parallel Sort

- Merge Sort (can also utilize bitonic split)

Every node may correspond to one processor/core
Parallel Sort

- Radix Sort
  1. Sort for every digit from least significant to most significant bit or the other way around. No order change for the data elements in previous sorted groups during the shuffling phase.
  2. Parallel prefix sum fits nicely in this framework. Every block of threads count the frequency of different digit values and their corresponding location in the same digit value group.

Designing efficient sorting algorithms for manycore GPUs (IPDPS’09)
Reference

(1). Designing efficient sorting algorithms for manycore GPUs (IPDPS’09)
   Radix sort and merge sort. Used prefix sum extensively.

(2). Fast parallel GPU-sorting using a hybrid algorithm (JPDC’08).
   Combination of bucket sort and vector merge sort

(3). GPU-ABiSort: optimal parallel sorting on stream architectures (IPDPS’06)
   Complexity: $\sigma((n \log(n)/p)$ on with up to $p=n/\log(n)$ streaming processors
