GP-GPU
and
High Performances Computing

Lecture 5
Parallel Pattern

https://christophe.picard.pages.ensimag.fr/courses/course/gp-gpu/Lectures/cgpu-fall23-lecture5.pdf

Course logistic

Information

➔ Reset password: https://copass-client.grenet.fr/app.php/simsu/secure/modifypwd/modify_password

➔ Using a VPN: https://intranet.ensimag.grenoble-inp.fr/fr/informatique/vpn-ensimag
  ◆ On Linux: when nothing change on screen, the process is ready

➔ Accessing the GP-GPU: ssh -K login@vmgpu0xx.ensimag.fr

➔ Copying on GP-GPU: scp origin login@vmgpu0xx.ensimag.fr:target

➔ Compiling: you need to modify the PATH variable
  ◆ export PATH=$PATH:/usr/local/cuda/bin

Review of GP-GPU hierarchy

Block grid definition

Masking of GPU memory access times

➜ a GPU switches from one thread warp to another very quickly
➜ a GPU masks the latency of its memory accesses by multi-threading

Do not hesitate to create large numbers of small GPU threads

Ex.: to process an array of N elements, you can define:

➜ Threads dealing with ONE element each
➜ and a Grid of blocks of N threads in total

Or

➜ Threads dealing with n elements each
➜ and a Grid of blocks of N/n threads in total

Review processing hierarchy

➔ Thread block organization: a grid of blocks of threads

➔ Streaming multiprocessor (SM): set of cores, cache, schedulers
   ◆ A block is assigned to and executed on a single SM

➔ Warp: A group of up to 32 threads within a block
   ◆ Threads in a single warp can only run 1 set of instructions at once
   ◆ Performing different tasks can cause warp divergence and affect performance

➔ Need to overlap warp computation with data loads

Review memory hierarchy

➜   Global memory access should be coalesced

➜   Shared memory may lead to bank conflicts

➜   Local memory and registered are faster than all other memories

Core organization

Each thread in a warp share

➔   An instruction stream to decode
➔   An execution context for storage (64kB per thread)
➔   8 SIMD functional unit
➔   One control unit

➔   Each core can run a group of 32 threads, a warp.
➔   Warps can be interleaved to run simultaneously (up to 320)
➔   Up to 10240 threads context can be stored

Divergence

Executing « if…then…else »

➔ Divergences are sources of slow down on SIMD

```
if (x < 10) then {....} else {...}
```

➔ Execution within a warp:
  1. All threads test the condition (x < 10)
  2. All threads must execute the then statement do it in parallel
  3. All threads must execute the else statement do it in parallel

➔ Execution time:
  ◆ If all threads execute the **then** statement then: T(condition) + T(then)
  ◆ If all threads execute the **else** statement then: T(condition) + T(else)
  ◆ If at least one thread execute the **then** statement and one execute the **else** statement then:
    T(condition) + T(then) + T(else)

Dealing with divergences

Expensive divergence

Every warp will execute **then** followed by **else**

It will lead to slow execution for the block

```
if (threadIdx.x % 2 == 0)
{...}
else
{...}
```

Reduced cost divergence

In 1D, only a warp will execute **then** followed by **else.**

It will lead to slow execution for the warp only

```
if (threadIdx.x < s)
{...}
else
{...}
```

Introduction to pattern

Lecture 5 - Programming on GP-GPU

Patterns

➔ Think at a higher level than individual CUDA kernels

➔ Specify what to compute, not how to compute it

➔ Let programmer worry about algorithm

➔ Defer pattern implementation to someone else

Common Parallel Computing Scenarios

➔ Many parallel threads need to generate a single result

◆ Reduce

➔ Many parallel threads need to partition data

◆ Split

➔ Many parallel threads produce variable output / thread

◆ Compact / Expand

Pattern 0: embarrassing parallelism

Simple operation

➔ Simple copy (with arithmetic) operation

```
// assign device and host memory pointers, and allocate memory in host
int thread_index = threadIdx.x + blockIdx.x * blockDim.x;
while (thread_index < N) {
  A[thread_index] = sqrt(A[thread_index]);
}
```

➔ 2 access to global memory (1 read and 1 write).

➔ 1 floating point operation.

The computational intensity is 0.5

C[i] = A[i] + B[i]

➔ CPU code:

```c
float *C = malloc(N * sizeof(float));
for (int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
```

➔ GPU code:

```c
// assign device and host memory pointers, and allocate memory in host
int thread_index = threadIdx.x + blockIdx.x * blockDim.x;
if (thread_index < N) {
  C[thread_index] = A[thread_index] + B[thread_index];
```

➔ 3 access to global memory (2 read and 1 write).
➔ 1 floating point operation.

The computational intensity is 1/3

Pattern 1 : Blocking

Blocking

→ Partition data to operate in well-sized blocks

  ◆ Small enough to be staged in shared memory
  ◆ Assign each data partition to a thread block
  ◆ No different from cache blocking!

→ Provides several performance benefits

  ◆ Have enough blocks to keep processors busy
  ◆ Working in shared memory cuts memory latency dramatically
  ◆ Likely to have coherent access patterns on load/store to shared memory

Blocking scheme: splitting

➔ Each thread block handle some different data

Lecture 5 - Programming on GP-GPU
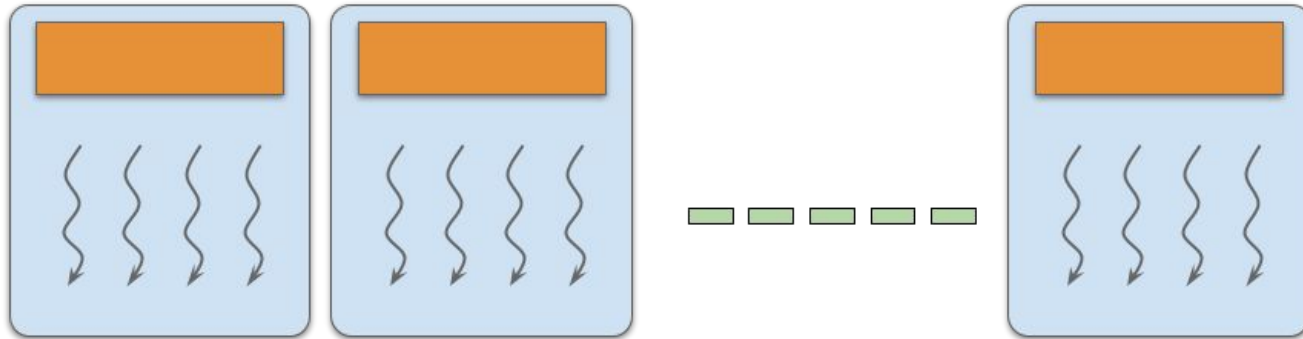
Blocking scheme: loading

➔ Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
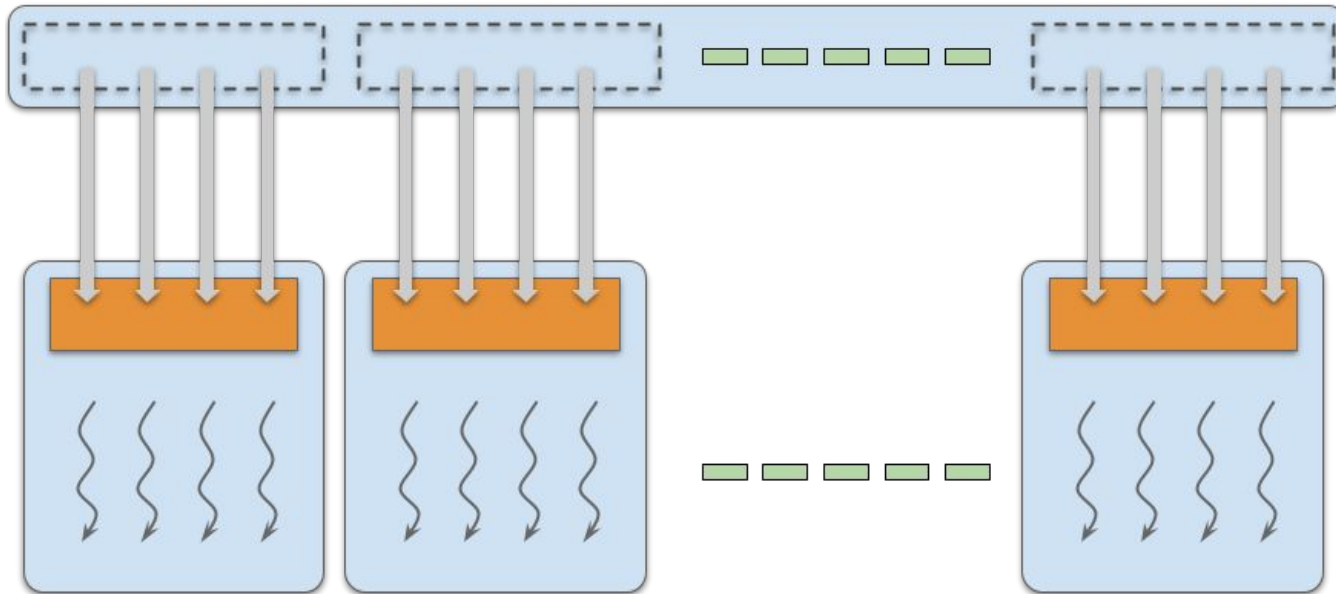
Blocking scheme: executing

➜ Perform the computation on the subset from shared memory
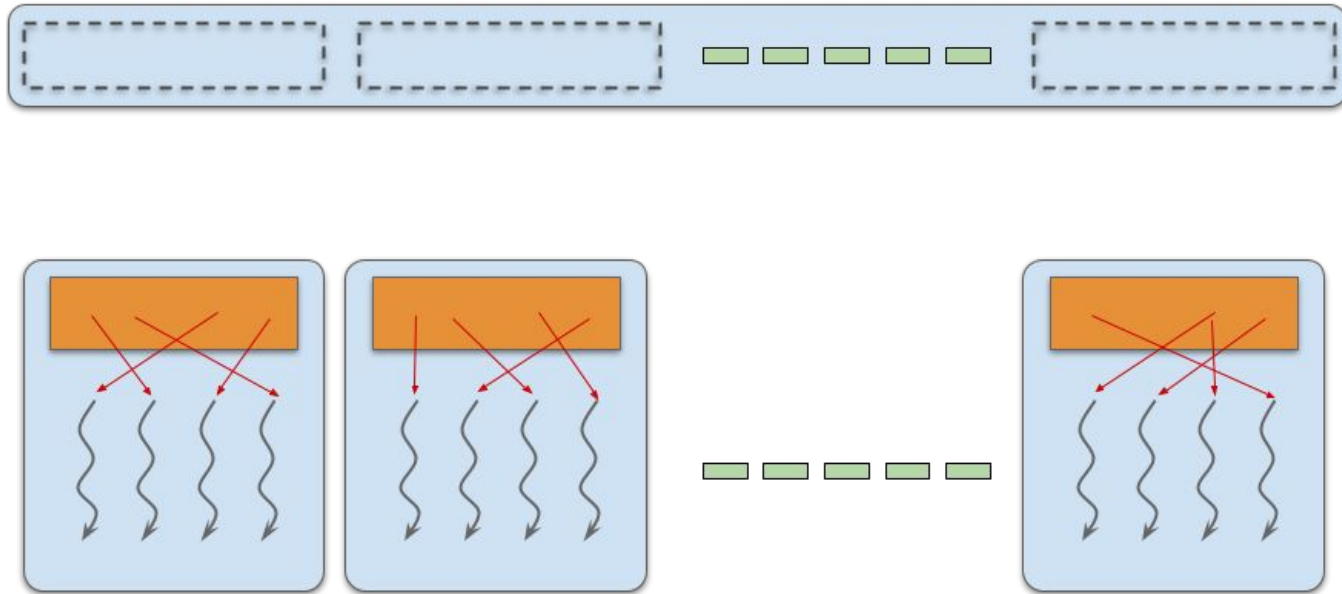
Blocking scheme: writing

➜ Copy the result from shared memory back to global memory
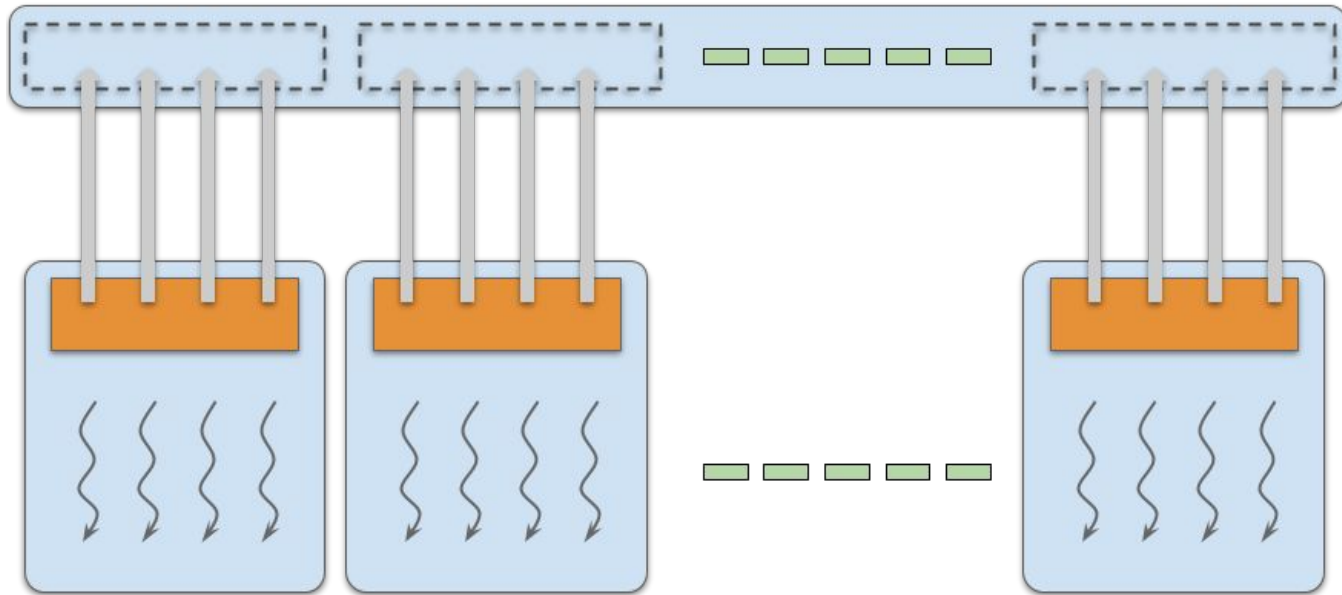
Blocking (2)

➔ All CUDA kernels are built this way

◆ Blocking may not matter for a particular problem, but you're still forced to think about it

◆ Not all kernels require `__shared__` memory

◆ All kernels do require registers

All the parallel patterns in this class will make use of blocking

Pattern 2 : Reduction

Reduction in sequential

➡ Reduce vector to a single value via an associative operator (+, *, min/max, AND/OR, ...)

```
// reduction via serial iteration
float sum(float *data, int n) {
    float result = 0;
    for(int i = 0; i < n; ++i) {
        result += data[i];
    }
    return result;
}
```

Reduction in parallel: strategy 0

```
reduce0(int *g_idata, int *g_odata, int n)
{
  unsigned int tidx = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int i0 = tidx * n;
  int sdata = 0;
  g_odata[blockIdx.x] = 0;

  // do reduction
  for (unsigned int s = i0; s < i0+n; s++) {
      sdata += g_idata[s];
  }
  g_odata[blockIdx.x] += sdata;
}
```

Reduction in parallel: strategy 0 bis

```
reduce0(int *g_idata, int *g_odata, int n)
{
  unsigned int tidx = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned int i0 = tidx * n;
  int sdata = 0;

  // do reduction
  for (unsigned int s = i0; s < i0+n; s++) {
      sdata += g_idata[s];
  }
  atomicAdd(g_odata[blockIdx.x], sdata);
}
```

Reduction in parallel: strategy 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2

11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2

18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2

24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2
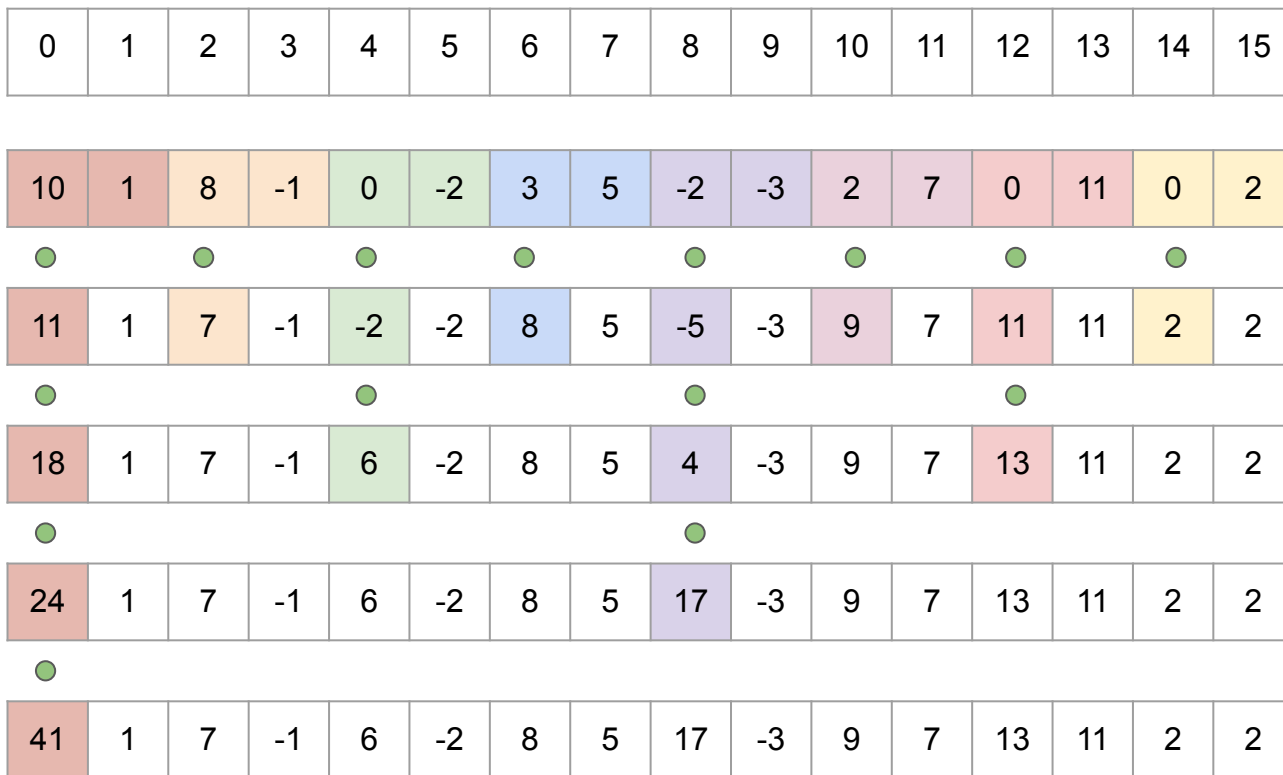
Reduction in parallel: strategy 1

➔ Strong divergence

➔ Reduction of more dispersed data in memory
➔ Memory accessed are not coalesced
➔ Active threads are more dispersed
➔ Activated warps with low number of active threads
➔ Bank conflicts

Reduction in parallel: strategy 1

```
reduce1(int *g_idata, int *g_odata)
{
  extern __shared__ int sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  // do reduction in shared mem
  for (unsigned int s = 1; s < blockDim.x / 2; s *= 2) {
    __syncthreads();
    int index = 2 * s * tid;
    if (index < blockDim.x) {
      sdata[tid] += sdata[tid + s];
    }
  // Thread 0 writes result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Reduction in parallel: strategy 2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2

11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2

18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2

24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

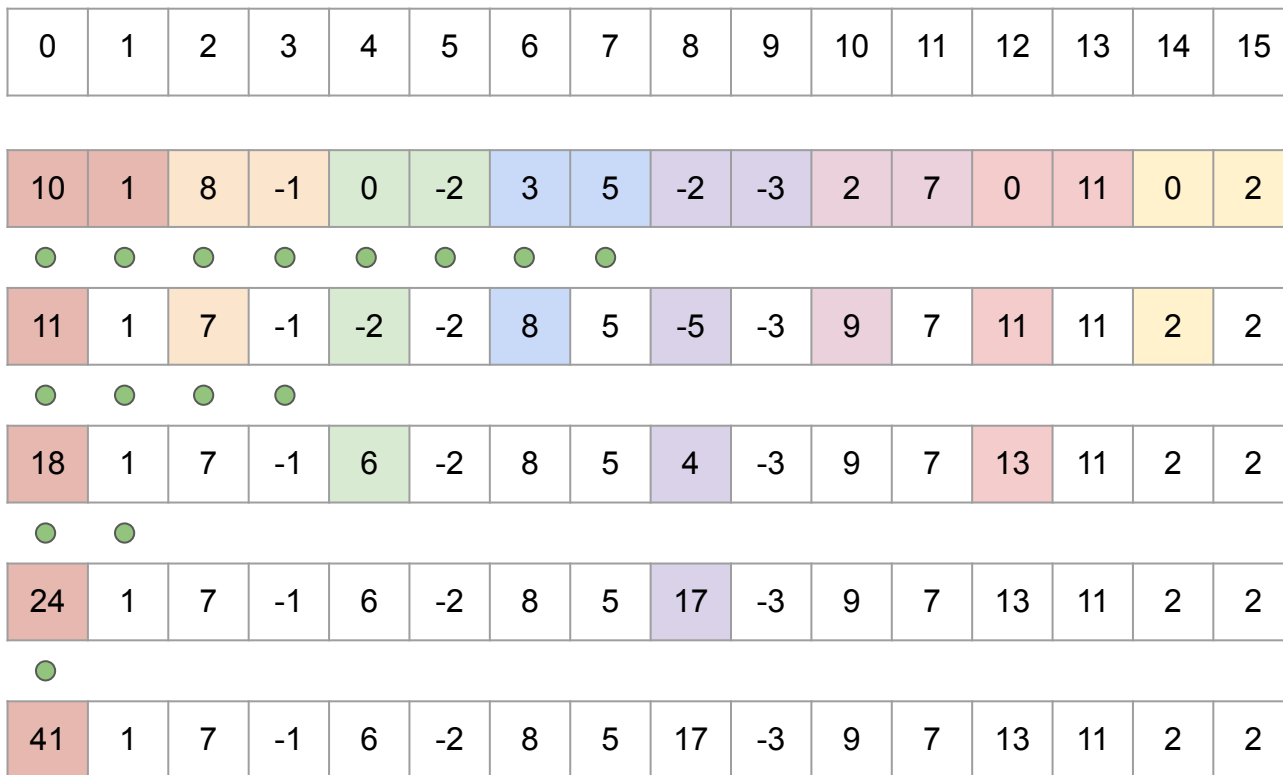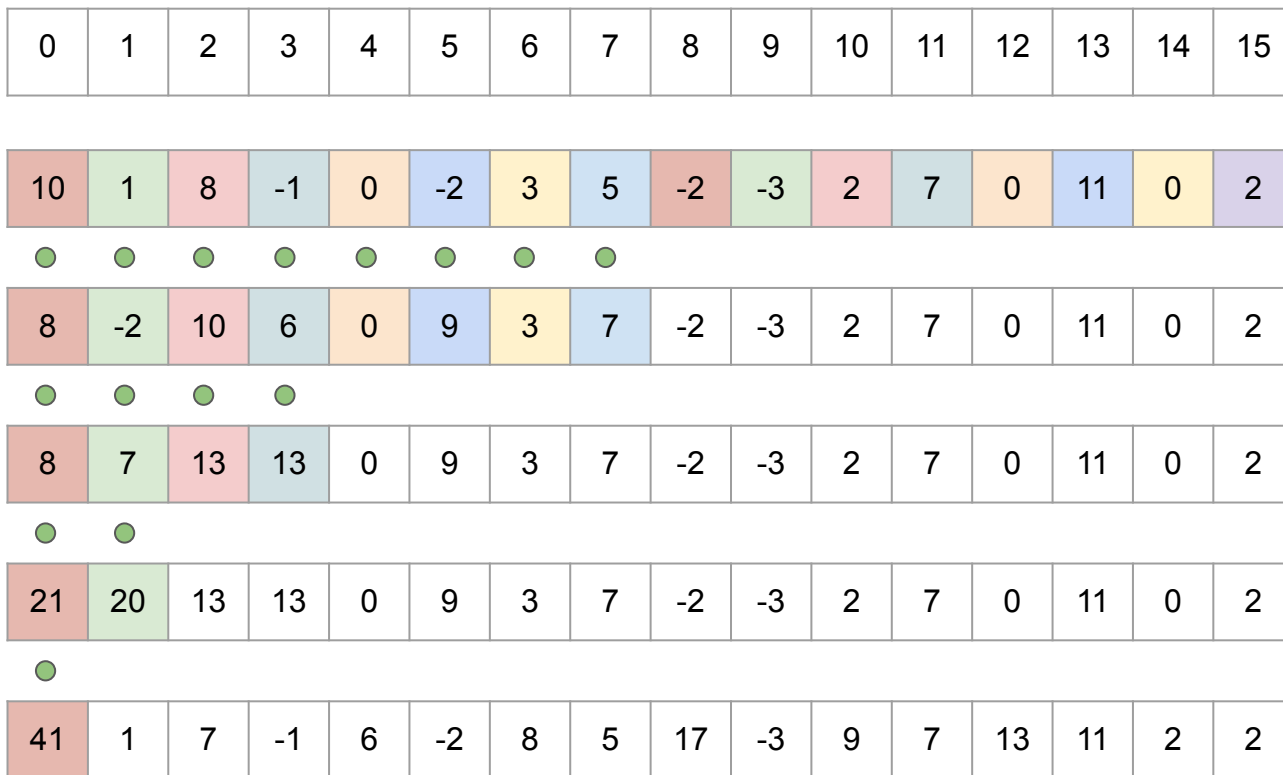| 41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

Reduction in parallel: strategy 2

➔ Limited divergence

➔ Reduction of more dispersed data in memory

➔ Memory accessed are not coalesced

➔ Subset of active threads coalesced from thread 0

➔ Activated warps with low number of active threads

Reduction in parallel: strategy 2

```
reduce2(int *g_idata, int *g_odata)
{
  extern __shared__ int sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();
  // do reduction in shared mem
  for (int s = 1; s < blockDim.x; s *= 2) {
    __syncthreads();
    if (threadIdx.x % (2 * s) == 0)
      sdata[tid] += sdata[threadIdx.x + s];
  }
  __syncthreads();
  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Reduction in parallel: strategy 3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2

8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2

8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

Reduction in parallel: strategy 3

➔   Limited divergence

➔   Memory accessed are coalesced

➔   Subset of active threads coalesced from thread 0

Reduction in parallel: strategy 3

```
reduce3(int *g_idata, int *g_odata)
{
  extern __shared__ int sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();
  // do reduction in shared mem
  for (unsigned int s = blockDim.x/2; s > 0; s >>= 1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
  }
  __syncthreads();
  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Complexity

➔ Takes **log(N)** parallel steps (step complexity) and each step S performs $\dfrac{N}{2^s}$ independent operations

➔ For $N = 2^D$ performs $\displaystyle\sum_{S=1}^{D} 2^{D-S} = N - 1$ operations

➔ It is work-efficient (i.e. does not perform more operations than a sequential reduction)

➔ With **P** threads physically in parallel (P processors), time complexity is O(N/P + log N)

➔ Compare to O(N) for sequential reduction

Conclusion

Lecture 5 - Programming on GP-GPU

Conclusions

Memory patterns

➔   Parallel programming make use of patterns to access memory efficiently.
➔   Patterns should be tuned to specific architectures.

Themes of this class

➔   Patterns
➔   Avoiding memory conflicts