Exercise 06 - Sum (reduction)

The objective of this lab is to familiarize you with a step often used in scientific computation, reduction. A reduction is combining all elements of an array into a single value using some associative operator. For example, the sum reduction

$$\text{sum} = \sum_{i=0}^{N} a_i$$

Data-parallel implementations take advantage of this associativity to compute many operations in parallel, computing the result in $\mathcal{O}(\log N)$ total steps without increasing the total number of operations performed. Three such $\mathcal{O}(\log N)$ reductions are shown below in the figures; pattern #1-#3. One of these reductions has benefits over other reduction patterns, in terms of minimizing thread divergence, coalescing memory accesses and avoiding shared memory bank conflict.

Concepts covered

The following concepts are covered in this exercise

- Data-parallel reduction algorithms
- Measuring execution time, branch divergence, and smem bank conflicts using the CUDA profiler.
- Experience how different different reduction schemes can affect branch divegence and bank conflicts.

Files needed

The following files will be needed for the exercise

- `sum.cu` (supposed to be correct and will not need changes)
- `sum_gold.c` (supposed to be correct and will not need changes)
- `sum_kernel.cu` (not correct and will need to be changed)

Work steps

You will need to modify the C code supplied in the files. You will implement four versions of the stencil code for the GPU. That is, one naive kernel and three kernels with different stride patterns. Then you will use the profiler to examine the overall performance to better understand the implications of the differences in terms of performance. To simplify things, N is allowed to be an integer multiple of `warpsize` and within each block of threads the number of threads should be of the form 2^p, p= $1, 2, ...$, in this exercise.

Device code

1. In the first implementation (`sum_kernel_naive`) you will perform a direct naive porting of the host code already implemented in `sum_gold`. In this version, threads read from and to global memory only. Use Pattern #1.
2. In the second implementation (`kernelStencil_v2`) shared memory will be used to do an in-place reduction per block. The partial sums from each block is then reduced by the host. Use Pattern #1.
3. The third implementation (`kernelStencil_v3`) will focus on using a different stride pattern. Use Pattern #2.
4. The fourth implementation (`kernelStencil_v4`) will focus on using a different stride pattern. Use Pattern #3.
5. In the last implementation (`kernelStencil_v5`) you should use the best of your previous implementations and optimize it as much as you can, e.g, by minimizing the instructions used.

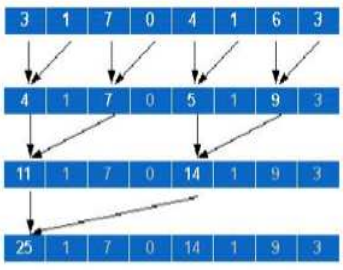
Compilation

Compile the final code using a `Makefile`. In case of compilation errors you will need to debug the code until it compiles successfully.

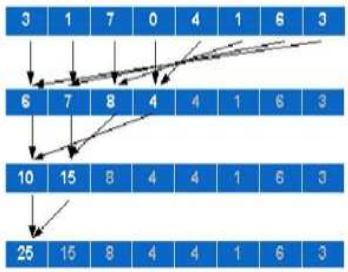
Execution

Execute your compiled program. If you program executes successfully the output should look like this

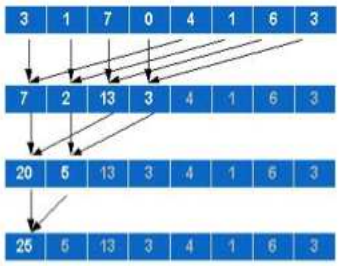
```
Parallel sum reduction.
  ./sum <N:default=64> <THREADS_PR_BLOCK:default=MaxOnDevice>

Device 0: Maximum number of threads per block is ?.
N: ?
Threads per block = ?.
Blocks allocated = ?

  CPU time           : ? (ms)

  GPU time (naive)   : ? (ms) , speedup ?x
  PASSED

  GPU time (v2)      : ? (ms) , speedup ?x
  PASSED

  GPU time (v3)      : ? (ms) , speedup ?x
  PASSED

  GPU time (v4)      : ? (ms) , speedup ?x
  PASSED
```



(a) Pattern #1



(b) Pattern #2



(c) Pattern #3