ASSIGNMENT 2:

Has two main parts:

1. The first part is aimed at writing non-trivial CUDA programs that ***use*** the parallel blocks of parallel programming: *reduce* and (*segmented*) *scan*.
2. The second part is aimed at verifying basic understanding of simple architectures such as vector machine and 5-stage pipeline with forwarding.

I. CUDA Programming with Reduce and Segmented Scan

Your started point is the CUDA implementation of *inclusive scan* and *inclusive segmented scan*, attached in Absalon under "Lecture and Lab Slides, And Sample Code" module in archive GenericInclusiveSgmScan.zip.
The archive contains three files:

- **ScanMain.cu** contains the main function that performs a segmented scan with operator addition on single-precision floating point numbers.
- **ScanHost.cu.h** provides the host wrappers for calling (segmented) scan: they implement a recursion in which the next GPU call scans the array resulted from the previous step, whose size is reduced with a factor of block size.
  Note that both of them are generically defined in terms of type parameters (templates):
  For example, to perform a segmented scan with addition on floats you will write `sgmScanInc< Add<float>,float >`, and similarly for scan.
- **ScanKernels.cu.h** provides the (generic) kernels that implement (segmented) scan. It should also implement the abstraction of the operator, as a templated C++ class that exports :
  - `identity` which returns the neutral element, and
  - `apply` which applies the associative operator. Such an example is Class `Add` at the beginning of the file.

The CUDA programming part consists of three tasks, where the goal is to quickly solve each of them in a manner that preserves the algorithmic work and depth complexity, but we are not necessarily looking yet to get the most tuned GPU implementation. ***However, we ask that for all tasks you provide (i) makefiles via which your programs can be easily compiled and run, and (ii) validation of the CUDA version, for example, by comparing with the results obtained from the sequential implementation and testing that the difference is epsilon (small). Please also write a short document explaining the key idea of how you implemented and validated your CUDA implementation. Ideally, you would also measure performance with respect to a sequential C(++) version and with Futhark.***

Task I.1.) Implement (**quickly**) exclusive scan and exclusive segmented scan. Please keep the same structure with a generic operator and type.
Hint: you can for example call the provided inclusive scan and then shift-right the result by one element, see already implemented kernel "shiftRightByOne" in ScanKernels.cu.h. For segmented exclusive scan, you can probably call the provided inclusive segmented scan and then call kernel "sgmShiftRightByOne", which you need to implement yourselves in file ScanKernels.cu.h. Don't forget to write the CPU wrappers and validate your implementation on suitably large dataset.

Task I.2.) Implement the maximum-segment sum problem in CUDA. It is a map-reduce problem, where the reduce operator is available in the lecture notes (L2-LH-Flattening).
Hint: the last element of the inclusive scan is the reduction. Implement by filling in the blanks in file ScanKernels.cu.h (i) the kernel (function) "msspTrivialMap", which corresponds to the MSSP's map part, and (ii) the function "MsspOp::apply", which corresponds to the MSSP reduce operator. Don't forget to write the CPU wrappers and validate your implementation on suitably large dataset.

Task I.3.) Implement sparse-matrix vector multiplication in CUDA.
A functional implementation with nested parallelism would represent the sparse matrix as a list of lists, in which each sublist corresponds to a row of the matrix containing index-value pairs corresponding to the elements that are not zeros. For example a row (sublist) such as [(1,5.0), (9,10.0)] means that only the elements on column 1 and 9 (on that row) are non-zero and have values 5.0 and 10.0 respectively. A more detailed explanation is found in archive EgHaskell.tar.gz in file PrimesQuicksort.hs attached as helper for the first assignment. A Futhark template can be found in archive EgFuthark.tar.gz in folder Flattening/SparseMatVct. The task has 2 subtasks:

- (1) Write a Haskell nested-parallel implementation, i.e., fill in function `nestSparseMatVctMult` from `PrimesQuicksort.hs`, and flatten the previous implementation by filling in function `flatSparseMatVctMult` from `PrimesQuicksort.hs OR`
- (1) Write directly a Futhark flat-parallel implementation, i.e., fill in function `spMatVctMult` from file `Flattening/SparseMatVct/spMVmult-flat.fut` (since Futhark does not support nested parallelism – the sequential version can be found already implemented in file `Flattening/SparseMatVct/spMVmult-seq.fut`).
- (2) Now translate the flat Haskel/Futhark implementation to CUDA! (The other steps are helpers for this one.) Note that are two kernel placeholder in file `ScanKernels.cu.h` that you can fill: kernel `spMatVctMult_pairs` is supposed to pairwise multiply the elements of rows with their corresponding vector values, and `write_lastSgmElem` is supposed to select the last element of the segment (after you have performed the segmented scan operation.) Don't forget to write the CPU wrappers and validate your implementation on a suitably large dataset.

II. Hardware Track Exercises

Task II.1.) With the classic 5-stage pipeline design presented in the fourth lecture (L3-InOrderPipe), consider the following program, which searches an area of memory and counts the number of times a memory word is equal to a key word:

```
SEARCH:     LW R5, 0(R3)         /I1 Load item
            SUB R6, R5, R2       /I2 compare with key
            BNEZ R6, NOMATCH     /I3 check for match
            ADDI R1, R1, #1      /I4 count matches
NOMATCH:    ADDI R3, R3, #4      /I5 next item
            BNE R4, R3, SEARCH   /I6 continue until all items
```

Branches are predicted untaken always and are taken in EX if needed. Hardware support for branches is included in all cases. Consider several possible pipeline interlock designs for data hazards and answer the following questions for the common loop iteration (i.e., all but the last one):

- (a) Assume first that the pipeline has no forwarding unit and no hazard detection unit. Values are not even forwarded inside the register file. Re-write the code by inserting NOOPs wherever needed so that the code will execute correctly.
- (b) Next, assume no forwarding at all, but a hazard detection unit that stalls instructions in ID to avoid hazards. How many clocks does it take to execute one iteration of the loop (1) on a match and (2) on no match? (Show the number of cycle associated to each instruction and explain briefly.)
- (c) Next, assume internal register forwarding and a hazard detection unit that stalls instructions in ID to avoid hazards. How many clocks does it take to execute one iteration of the loop (1) on a match and (2) on no match? (Show the number of cycle associated to each instruction and explain briefly.)
- (d) Next, assume full forwarding and a hazard detection unit that stalls instructions in ID to avoid hazards. How many clocks does it take to execute one iteration of the loop (1) on a match and (2) on no match? (Show the number of cycle associated to each instruction and explain briefly.)
- (e) Can the compiler improve performance by unrolling the loop? How? Would delayed branches help?

Task II.2.) Refers to vector processors. One very common operation is the dot product of two vectors, which results in a scalar. The C-pseudocode for computing the dot-product of vectors X and Y of size 1024 is given below:

```
for(k=0; k<1024; k++) p += x[k]*y[k];
```

Assuming that the size of the hardware vector instruction is `64`, the code can be vectorized by strip-mining the loop:

```
for(k=0; k<1024; k+=64) {
    for(j=0; j<64; j++)       // vectorize this inner loop
        p += x[k+j] * y[k+j];
} //end of loop
```

followed by vectorizing the inner loop, i.e., `p` becomes a vector (register) of size `64`, the values of `x[k:k+63]` and `y[k:k+63]` are loaded into vector registers, etc., and a final stage is added after the loop that sums up the elements of the resulted vector `p` to yield a scalar value. Assume the instructions to load (L.V), multiply (MUL.V) and add (ADD.V) vectors are available. Assume also that the vector operations are chained, and a vector architecture with eight vector registers of 64 components each and a very large number of memory banks (say 1024 so that memory bank conflicts never appear). Assume also that the bank access time is 30 cycles, the latency (start-up cost) of the multiply pipeline is 10 and of the add pipeline is 5.

- (a) Give the machine code (with vector instructions) for the processing of each 64-component slice using vector loads and arithmetic instructions.
- (b) Compute the time taken by a dot product where the vector sizes are 1024 each. (Neglect the final scalar phase that accumulates the values of the resulted `p` vector)
- (c) How many clocks does it take to compute the multiplication of two 1024 x 1024 (dense) matrices using the same algorithm (neglect the final scalar phase).