Third and Fourth Weekly Assignment for the PMPH Course (21 pts total)

This is the text of the third and fourth weekly assignment for the DIKU course "Programming Massively Parallel Hardware", 2023-2024. The assignments consists of two pen-and-paper exercises and four coding exercises in Cuda.

- Assignment 3 consists of tasks 1, 2 and 3: total 10 pts (tasks 1 and 2 are pen and paper, task 3 is a programming task)
- Assignment 4 consists of tasks 4, 5 and 6: total 11pts (all are programming tasks)
- The due date is the same for both assignments; you are encouraged to submit once.

Hand in your solution in the form of a short report in text or PDF format, along with the whole codebase that has been improved with your solutions. We hand-in incomplete code in the archive `w3-code-handin.tar.gz`. You are supposed to fill in the missing code such that all the tests are valid and to report performance results. Please send back a zipped archive containing the same files under the same structure that was handed in---implement the missing parts directly in the provided files. There are comments in the source files that are supposed to guide you (together with the text of this assignment).

Unziping the handed in archive `w3-code-handin.tar.gz` will create the `w3-code-handin` folder, which contains:

- A `helper.h` file that contains helper functionality used by all exercises
- A `REDME.md` that provides a short rationale about the Cuda-coding exercises and presents the code structure; **make sure to read it**.
- Folder `histo-L3-thrashing` provides the code base for the first coding exercise referring to optimizing last-level cache (LLC) threshing in the context of a histogram-like computation.
- Folder `gpu-coalescing` provides the code base for the second coding exercise referring to optimizing GPU spatial locality (coalescing) by means of transposition.
- Folder `mmm` provides the code base for the third coding exercise referring to optimizing temporal locality for matrix-matrix multiplication.
- Folder `batch-mmm` provide the code base for the fourth coding exercise referring to optimizing temporal locality for a batch instance of matrix multiplication, in which the same matrices are multiplied but under a different mask.

Write a neat and short report containing the solutions to the first two theoretical questions, and also the missing code and short explanations for the four coding exercises in Cuda. Also provide comments regarding the performance behavior of your programs:

- what is performance (in GB/sec or Gflops) and what is the speedup generated by your improvement with respect to the best performing GPU version provided?
- short and human-understandable rationale for justifying the speedup, for example:
  - for `histo-L3-thrashing` and `gpu-coalescing`: why does the optimized GPU program is several times faster than the GPU baseline in spite of performing a factor of 4x and 3x more accesses to global memory than the baseline, respectively ?
  - for matrix multiplication and batch matrix multiplication: how does the optimization

improves the temporal reuse ?

Task 1: Pen and Paper Exercise Aimed at Applying Dependency-Analysis Transformations (3 pts)

Consider the C-like pseudocode below:

```
float A[2*M];

for (int i = 0; i < N; i++) {
    A[0] = N;

    for (int k = 1; k < 2*M; k++) {
        A[k] = sqrt(A[k-1] * i * k);
    }

    for (int j = 0; j < M; j++) {
        B[i+1, j+1] = B[i, j] * A[2*j  ];
        C[i,   j+1] = C[i, j] * A[2*j+1];
    }
}
```

Your task is to apply privatization, array expansion, loop distribution and loop interchange in order to parallelize as many loops as possible. Answer the following in your report:

- Explain why in the (original) code above **neither** the outer loop (of index `i`) **nor** the inner loops (of indices `k` and `j`) **are parallel**;
- Explain why is it safe to privatize array `A`;
- Once privatized, explain why is it safe to distribute the outermost loop across the `A[0] = N;` statement and across the other two inner loops. Perform (safely!) the loop distribution, while remembering to perform array expansion for `A`.
- Now you can use direction vectors to determine which loops in the resulted three loop nests are parallel. Please explain and annotate each loop with the comment `// parallel` or `// sequential`.
- Can loop interchange be applied so that each loop nest contains one parallel loop? Please explain why (or why not) and show the code exhibiting maximum parallelism --- please annotate each loop with the comment `// parallel` or `// sequential`.

Task 2: Pen and Paper Exercise Aimed at Recognizing Parallel Operators (3 pts)

Assume that both A and B are matrices with N rows and 64 columns. Consider the pseudocode

below:

```
float A[N,64];
float B[N,64];
float accum, tmpA;
for (int i = 0; i < N; i++) { // outer loop
    accum = 0;
    for (int j = 0; j < 64; j++) { // inner loop
        tmpA = A[i, j];
        accum = sqrt(accum) + tmpA*tmpA; // (**)
        B[i,j] = accum;
    }
}
```

Reason about the loop-level parallelism of the code above and answer the following in your report:

- Why is the outer loop **not** parallel?
- What technique can be used to make it parallel and why is it safe to apply it? Re-write the code such that the outer loop is parallel, i.e., the outer loop does not carry any dependencies.
- Explain why the inner loop is **not** parallel.
- Assume the line marked with `(**)` is re-written as `accum = accum + tmpA*tmpA`. Now it is possible to rewrite both the inner and the outer loop as a nested composition of parallel operators! Please write in your report the semantically-equivalent Futhark program.

Task 3: Histogram-like Computation — Cuda Exercise 1 (4 pts)

See section "LL$ threshing: Histogram-like computation" in companion lecture slides `L6-locality.pdf`.

The programming task refers to implementing the missing code in files `main-gpu.cu` and `kernels.cu.h`---search for keyword "Exercise" in those files and follow the instructions.

Program arguments are, e.g., see Makefile:

- The first argument of the program is the size `N` of the array of indices/values.
- The second argument of the program is the size of the last-level cache (LL$) in bytes. Please make sure to adjust it to the hardware you are running on (both CPU and GPU), otherwise you will not observe much. The sizes used in the makefile are particularized to the `futharkhpa01fl` and `futharkhpa03fl` machines.
- The size of the histogram is computed internally such as four passes over the input are always performed.

Briefly comment in your report on:

- the code implementing your solution, i.e., present the code and comment on its correctness and on how it optimizes locality. For example, why do you expect speedup when the improved implementation performs a factor of 3-4x more access to global memory (since it traverses the input four times).
- specify whether your implementation validates
- report the GB/sec achieved by your implementations and of the GPU baseline and also report the speedup in comparison with the GPU baseline (i.e., the other provided implementation)

Task 4: Optimizing Spatial Locality by Transposition — CUDA exercise 2 (4 pts)

See section "Optimizing Spatial Locality by Transposition" in companion lecture slides `L6-locality.pdf`.

The programming task refers to implementing the code of Cuda kernel `transKernel` in file `kernels.cu.h`, which works on the transposed versions of A and B, named `A_tr` and `B_tr`, respectively. Please search for keyword "Exercise" in file `kernels.cu.h` to find the implementation place.

Briefly comment in your report on:

- the code implementing your solution, i.e., present the code and comment on its correctness and on how it optimizes spatial locality (i.e., coalesced access to global memory). For example, why do you expect speedup when **your** implementation performs a factor of 3x more access to global memory than the baseline.
- specify whether your implementation validates.
- report the GB/sec achieved by **your** GPU implementation and of the GPU **baseline** , and also report the speedup w.r.t. the baseline.
- briefly explain why the CPU implementation that uses GPU-like coalescing has abysmal performance (i.e., much slower than the baseline).
- **BONUS** briefly explain at a very high level, why/how "the Optimal-GPU Program" is about 2x faster than your implementation. ("the Optimal-GPU Program" is the last GPU program run by the Makefile)

Task 5: Matrix-Matrix Multiplication (MMM) — Cuda Exercise 3 (3 pts)

See section "L1$ and Register: Matrix-Matrix Multiplication" in companion lecture slides `L6-locality.pdf`.

The programming task refers to implementing some of the code of Cuda kernel `mmmSymBlkRegInnSeqKer` in file `kernels.cu.h`. Please search for keyword "Exercise" in file `kernels.cu.h` to find the implementation place, and follow the instructions there. Also look around to see how it

is called from the CPU (host) code.

Please be aware that Section 6.4 of lecture notes presents a different tiling strategy for matrix-matrix multiplication; i.e., it is related but it is **not** what you have to do.

Briefly comment in your report on:

- the code implementing your solution,
- specify whether your implementation validates,
- report the performance in Gflops achieved by **your** GPU implementation and by the GPU **baseline** , and also report the speedup w.r.t. the baseline.
- Finally, explain in your report the high-level reasons for obtaining this speedup, i.e., how did your implementation improved the temporal locality (e.g., by what factor has decreased the number of accesses to global memory).

Task 6: Batched Matrix Multiplication Under a Mask — Cuda Exercise 4 (4 pts)

See section "L1$ and Register: Batch Matrix Multiplication under a Mask" in companion lecture slides `L6-locality.pdf`.

The programming task refers to implementing the code of the Cuda kernel `bmmmTiledKer` in file `kernels.cu.h`. Please search for keyword "Exercise" in file `kernels.cu.h` to find the implementation place, and follow the instructions there. Remember to flatten the indices to all arrays hold in global memory. Also look around to see how it is called from the CPU (host) code.

Briefly comment in your report on:

- the code implementing your solution,
- specify whether your implementation validates,
- report the performance in Gflops achieved by **your** GPU implementation and by the GPU **baseline** , and also report the speedup w.r.t. the baseline.
- Finally, explain in your report the high-level reasons for obtaining this speedup, i.e., how did your implementation improved the temporal locality (e.g., by what factor has decreased the number of accesses to global memory).