Group Project

The handed out (sequential) code implements a simplified version of volatility calibration, which uses Crank-Nicolson finite difference method. The code is implemented in a simple C++ style, which is morally C, but where we have used C++ vectors to make the multi-dimensional indexing clean and clear. The code is tested on three datasets: ***small***, ***medium*** and ***large***, where each dataset exposes a different amount of available parallelism. Parsing the datasets and validation are built in the code.

Your task is to parallelize the code on both CPU via ***OpenMP*** (trivial) and on GPU via ***CUDA***. For ***CUDA***, the aim is to tune your implementation to the characteristics of the dataset, such as to best utilize one GPGPU. (Note that you are not required to understand the algorithm, just to parallelize it.)

I. A Cursory Glance at the Handed-Out Code

Folder **`include`** contains parsing and validation functionality, together with some CUDA stubs for inclusive (segmented) scan and matrix transposition, i.e., file **`CudaUtilProj.cu.h`**. (A basic CUDA test program for it can be found in **TestCudaUtil.cu**) .

Folder **`OrigImpl`** contains the implementation: File **`ProjectMain.cpp`** implements the main function with built-in parsing and validation. File **`ProjCoreOrig.cpp`** provides the core implementation that needs to be parallelized, while file **`ProjHelperFun.cpp`** provides data initialization helpers, which can also be parallelized.

When getting familiar with the code, a good entry point is the loop in function **`run_OrigCPU`** in file **`ProjCoreOrig.cpp`**. The sizes of various multi-dimensional arrays can be found in the declaration of class **`PrivGlobs`** in file **`ProjHelperFun.h`**.

Folder **`Data`** contains the three input sets and reference output arrays for each (used for validation). The comments corresponding to each number name the (constant) program variable that will hold that value. For example, **`NUM_T`** is the count of a time series loop, which is sequential and cannot be efficiently parallelized. The rest of the input **`OUTER`**, **`NUM_X`**, and **`NUM_Y`** correspond to the loop nests that can be parallelized.

In essence, the important loop nests of the application exhibits two outer parallel loops which are morally maps, i.e., trivially parallel, and an innermost one, corresponding to the **`tridag`** function which requires (several) scans, and, as such, is less efficient to parallelize.

Folder **`ParTridagCuda`** contains code that demonstrates how to parallelize **`tridag`** function.

II. Parallelization Strategy:

The parallelization strategy should be thus sensitive to the input dataset:

1. This project targets first-and-foremost efficient parallelization of the **`large`** dataset. This refers to parallelizing only the outer two dimensions of (map-like, hence trivial) parallelism, i.e., **`tridag`** is sequentially executed inside the cuda kernels. This parallelization strategy can be tested on the ***`large`*** dataset, which has large-enough parameters to fully occupy one GPU, i.e., **`OUTER=128, NUM_X = 256, NUM_Y = 256`**, results in degree of parallelism **`OUTER*MAX(NUM_X,NUM_Y)=32768`**.
2. If time permits, you can also try to efficiently parallelize the **`small`** dataset. This dataset corresponds to values **`OUTER=16, NUM_X = 32, NUM_Y = 256`**, and requires that all parallelism is exploited in order to efficiently utilize the GPGPU, i.e., the loops in the **`tridag`** function must also be parallelized. The latter requires the computation of several segmented scans (interleaved with maps) in the innermost dimension. However the segment size is either **`32`** or **`256`**, which means that the scans can be performed at CUDA-block level, hence efficient. (Meaning, if the block size is chosen 256 then the elements of a segment will never cross two blocks, and in particular there is a multiple of segments that would fit exactly the size of the block.) You will receive ample help on how to rewrite **`tridag`** into segmented scans and how to parallelize the scans. For example, code that demonstrates **`tridag`** parallelization is located in folder **`ParTridagCuda.`**
3. ***Bonus:*** The medium dataset uses parameters **`OUTER=32, NUM_X = 47, NUM_Y = 181,`** which, as in the first case, requires all parallelism to be exploited. This however, do not fit exactly any block size, because the block size should be a multiple of 32. This case can be treated similar to the first case, except that some of the threads of the block would be considered idle: For example, for the segment of size 47 and 181, a suitable cuda-block size would be 256 and 192 respectively. The idle threads (i.e., the last 256-47*5 and 192-181 threads) do not participate in map operations (via **`if`**s), but they do participate in **`scan`**s (since all threads need to reach the barrier, hence they use the neutral element for padding). Also the global indexing needs to be adjusted to fit the iddle threads, because for example only 181 array elements are processed in a cuda block, rather than 192, which is the block size.

Note that, for good performance it is rather important to have coalesced access to global memory everywhere!

III. Suggested Steps in Approaching the Project:

Summarize accesses at function and loop level, i.e., which indices are written first or read only at each level. Then decide: which loops are (already) parallel, which loops can be made parallel by privatization, which loops are inherently sequential, and which loops can be made parallel via changing the algorithm, e.g., **`tridag`** can be re-written in terms of scans with 2x2 matrix multiplication and linear function composition, as we shall discuss in class.

Write the **`OpenMP`** version that parallelizes the outermost parallel loop (of count **`OUTER`**). After privatization has been applied, this corresponds to placing an OpenMP pragma directive to the outermost parallel loop, such as:

```
#pragma omp parallel for default(shared) schedule(static) if(outer>8)
```

For each of the CUDA versions you implement, I advice that you do the main transformations, e.g., loop distribution, interchange, matrix transposition, etc., within the C(++) program and that you move to CUDA at the very end. Meaning, transform the CPU program to resemble the CUDA program that you wish to obtain, so that you can debug it on the CPU rather on the GPU. This would allow easier debugging, and working with the C++ vector will enable cleaner, multi-dim index expressions.

IV. Report Should Contain:

A *CLEAR*, *CONCISE*, and *COMPREHENSIVE* explanation of your work. (Note that marks will be added/subtracted based on the quality of the report.)

1. The sequence of transformations that you have applied to obtain each version of the CUDA code: Show first the sequence of transformations on a (very) simplified version of the code, and explain it at a high level. Then explain in more detail:
    - What were each of these transformations aiming to optimize, e.g., degree of parallelism, memory coalescing, etc. Explain the relation with the GPU hardware.
    - Why were they safe to apply (correctness argument)
2. The discussion should be informative and concise: Use an illustrative example for each such transformation, i.e., the simplest one that still captures the essence of the transformation. Then, for example, say that this was applied to arrays named ... in loops ...

3. Mention whether the CUDA versions you have implemented validate.
4. Provide an experimental evaluation of your CUDA code. This could measure:
    - application total (wall-clock) runtime ,
    - speedup in comparison with the parallel CPU execution,
    - speedup in comparison (perhaps) with a ``naive'' CUDA version to show the impact of each optimization.
    - anything else that you might think is of interest.