FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Programming Massively Parallel Hardware – Optimising Tridag

Group Project
Department of Computer Science

Written by:

Morten Espensen & Niklas Høj & Mathias Svennson
dzr440 - 19/06/1991 & nwv762 - 24/07/1991 & tpx783 - 20/07/1990
October 31, 2014

Supervised by:

Cosmin E. Oancea

Contents

Chapter 1

Versions of our code

We have included 6 versions of our code in the attached tarball:

- The directory `1_OriginalCPU` contains the original code, only modified slightly to make e.g. whitespace more consistent with the result of our code.
- The directory `2_OpenMP` contains our OpenMP code, which parallelizes the outer loop in `run_OrigCPU`.
- The directory `3_NaiveCuda` contains our initial CUDA version. It is coded without any major code transformations: We have simply taking the loops that were naïvely parallelizable and implemented kernels for them. To achieve this we inlined the `tridag` function, so parts of it could be made more parallel.
- The directory `4_OuterParallelCuda` gets a large speedup by expanding the arrays and moving the outer loop into `rollback`, thus making all the kernels run on a larger number of blocks.
- The directory `5_ReducedCudaDimensions` gets a further speedup by reducing the outer-dimension of the some of the arrays, as their value did not depend on that dimension.
- The directory `6_CudaFinal` is our final version. It is for all intents and purposes the same as the previous version, except almost every memory access have been made coalesced.

Chapter 2

Summary of different loops

There are only two true sources of loop-dependencies in the code:

- The loop in `value` have dependencies upon previous versions of the same data.
- The loops in `tridag` are bascially two scans, though after the expansions provided in `3_NaiveCuda`, we altered it to three scans and a few maps, implemented as three kernels.

Every other loop is completely parallelizable.

Chapter 3

Transformations

3.1 Transformation 1 → 2: OpenMP privatization

The transformations done in our OpenMP version are very small: We moved the calculation of `PrivGlobs` and `strike` into the `value`-function. This caused the outer loop to be parallelizable, so we put a pragma on it for OpenMP parallelism.

This caused a 19 times speedup (from 194.1 seconds to 10.3 seconds). We did manage to squeeze slightly more performance out of the code, e.g. by using arrays instead of vectors. However the performance gains were quite small and the changes quite large (and uninteresting for this version), so we decided not to include the further optimized version.

3.2 Transformation 1 → 3: Naive CUDA

3.3 Transformation 3 → 4: Parallelizing the outer loop

3.4 Transformation 4 → 5: Reducing dimensions

The transformation between version 4 and 5 mainly consisted of reducing the dimensions of the arrays in `PrivGlobs` such that we do not recalculate identical instances of the same arrays in each outer dimension as.

Thus the array definitions seen in Figure 3.1 has been changed such that they are only calculated once and then reused the result thoughtout the execution. Accordingly all accesses has been changed to not respect the outer dimension.

4_OuterParallelCuda/ProjHelperFun.h

```
checkCudaError(cudaMalloc(&this->a,          numO * numY * numX * sizeof(REAL)));
checkCudaError(cudaMalloc(&this->b,          numO * numY * numX * sizeof(REAL)));
checkCudaError(cudaMalloc(&this->c,          numO * numY * numX * sizeof(REAL)));
checkCudaError(cudaMalloc(&this->yy,         numO * numY * numX * sizeof(REAL)));
```

5_ReducedCudaDimensions/ProjHelperFun.h

```
checkCudaError(cudaMalloc(&this->a,                 numY * numX * sizeof(REAL)));
checkCudaError(cudaMalloc(&this->b,                 numY * numX * sizeof(REAL)));
checkCudaError(cudaMalloc(&this->c,                 numY * numX * sizeof(REAL)));
checkCudaError(cudaMalloc(&this->yy,                numY * numX * sizeof(REAL)));
```

Figure 3.1: Overview of data structure changes between version 4 and 5

3.5 Transformation 4 → 5: Coalesced access

Chapter 4

Does it validate?

Yes.

Chapter 5

Results

Data set size / Implementation	Small	Medium	Large
Sequential	2041425μs	4996503μs	187699474μs
OpenMP	183016μs	241972μs	9680948μs
Naive CUDA	2142530μs	3344009μs	40035689μs
CUDA with rollback loop propagated	480526μs	503774μs	7794961μs
CUDA with dimmension reductions	452969μs	371853μs	3727729μs
Optimised CUDA	314699μs	324522μs	791390μs