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

1 Versions of our code **2**

2 Summary of different loops **3**

3 Transformations **4**

3.1 Transformation 1 → 2: OpenMP privatization 4

3.2 Transformation 1 → 3: Naïve CUDA 4

3.3 Transformation 3 → 4: Parallelizing the outer loop 5

3.4 Transformation 4 → 5: Reducing dimensions 5

3.5 Transformation 5 → 6: Coalesced access 5

4 Does it validate? **7**

5 Results **8**

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_ReducedCudaDimmensions` 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: Naïve CUDA

The purpose of this transformation was to take the original code and make a parallel version of it without doing much work in terms of code transformations. We simply took all loops that were trivially parallizable and created kernels for them.

To make this work without being utterly slow, to also inlined tridag and unpacked it into a few maps and three scans. The three scans in each tridag was implemented as 2 kernels.

3.3 Transformation 3 → 4: Parallelizing the outer loop

3.4 Transformation 4 → 5: Reducing dimensions

3.5 Transformation 5 → 6: Coalesced access

With a working and well-structured naive Cuda implementation, the next step was to optimise it to use coalesced access whenever possbile. To this end, we have written and read from arrays at consecutive indices by consecutive Cuda thread ids whenever possible. In doing this, we have effectively transposed (in-place during assignment) most (if not all) of the arrays from their initial versions.

When an array must be arranged one way in one place and transposed in another, we have transposed the array locally in a kernel into shared memory. We did this for kernels 0 and 5, but ended up reversing the change for kernel 5 (more below).

To take more advantage of the coalesced access, we tried using various block sizes for the kernels. The block size of kernel 0 remained the same in the tests, namely 32x32, since we use shared memory in this and cannot spawn more threads per core. The table below shows the timings of individual kernels at one-dimensional block sizes 32, 64, 128, 256 and 512.

```
Table of performance in micro-seconds of kernels:
(Note: Kernel 0 is always 32x32 = 1024 for shared memory.)

Block size:        32       64      128      256      512

Kernel   0:    149963
Kernel   1:      2927     2517     2516     2497     2455
Kernel   2:     18333    18504    18582    18622    18644
Kernel   3:     93541    53443    36580    37026    37070
Kernel   4:    132734    96662    85957    91119    92739
Kernel   5:    143290    86995    70856    71047    71148
Kernel   6:      2951     2478     2408     2412     2351
Kernel   7:     18178    18291    18385    18446    18471
Kernel   8:    166507   154633   153778   154123   154369
Kernel   9:    132842    98206    87702    91106    91275
Kernel  10:      7143     6474     6470     6523     6537

Total time:   2031194  1860382  1793703  1808618  1818025
```

Looking at the table, we found that all kernels (besides 0) performed best (allowing room for variation in measurements) at block size 128.

Kernel 5 remains partially non-coalesced, and we did try to fix that by using shared memory and a block size of 32x32. However, the resulting kernel took 110000 micro-seconds to execute, which was an improvement to the 143290 micro-seconds from the one-dimensional block size of 32. But the one-dimensional block size of 128 without shared memory only took 70856 microseconds, which is clearly better.

We suspect that a similar improvement might be made in kernel 0, again by sacrificing shared memory for a one-dimensional block size of 128 (or others), but we will not pursue this optimisation this time around.

Looking at the kernel timings as above, we continued to tweak on coalesced accesses, benefiting in one kernel at the cost of another, searching for the optimal configuration.

The final kernel timings are listed below, and the final running time is 1695149 micro-seconds (1.7 seconds).

Kernel	micro-seconds
6	2214
1	2278
10	5961
7	18728
2	18760
3	36609
8	37111
9	85748
4	85761
5	96671
0	156493

Chapter 4

Does it validate?

Yes.

Chapter 5

Results

Data set size / Implementation	Small	Medium	Large
Sequential with flatten arrays	2020815μs	4325245μs	191298141μs
OpenMP	183016μs	241972μs	9680948μs
Naive CUDA	3956322μs	3639421μs	34980508μs
CUDA with rollback loop propagated	462572μs	486986μs	19281810μs
CUDA with dimmension reductions	391842μs	371853μs	3727729μs
Optimised CUDA	183016μs	241972μs	9680948μs