# MOHID in CUDA
# Test report


| | |
|---|---|
| *Author* | Jonathan van der Wielen |
| *Date* | 2011-09-20 |
| *Last modified* | 2011-11-04 |
| *Version* | 1.0 |

## Version Control

| *Version* | *Editor* | *Comments* |
|---|---|---|
| 0.1 | Jonathan | First draft. |
| 0.2 | Jonathan | Add structure for tests.  Describe test cases. |
| 1.0 | Jonathan | Added results for all tests. |
| | | |

# Contents

# Introduction

This document analyzes the performance and correctness of a number of MOHID parts that have been implemented in CUDA. The purpose of these tests is to see if it is feasible to run parts of MOHID in CUDA rather than in FORTRAN.

Chapter 1 performs a correctness benchmark on the Thomas algorithm, a tri-diagonal matrix solver. The test is performed to determine the differences in floating point precision between the FORTRAN implementation and the CUDA implementation. The purpose of this test is also to make sure the algorithm has been implemented correctly.

After confirming the correctness of the CUDA implementation, chapter 2 researches the performance of the Thomas algorithm. Several CUDA implementations are tested, and the best result is compared to the original algorithm in FORTRAN in chapter 3.

# 1 Correctness benchmark Thomas algorithm

The Thomas algorithm is a tri-diagonal matrix solver that is widely used in hydro-dynamics. A detailed description of the algorithm can be found in chapter 1 of the MOHID – CUDA documentation[1].

The Thomas algorithm is a performance bottleneck in MOHID, it consumes up to 10% of the total calculation time of a model, depending on the configuration.

The purpose of this correctness benchmark is to make sure that the Thomas algorithm is executed the same in CUDA as in FORTRAN.

## 1.1 Test cases

Thomas is generally executed for a number of properties. This benchmark does not test all properties; it tests only one property for each dimension since the Thomas algorithm does not have branches that may diverge differently depending on the given property.

The Thomas algorithm is tested for all dimensions: X, Y and Z, in release mode with optimizations enabled. Each of these tests is run for 24 hours with time steps of 20 seconds. The results are printed every 25$^{th}$ time step that the algorithm is executed (every 8m20s for Z and every 16m40s for X and Y). The tests are executed for the velocity for the Z dimension and for the cohesive sediment transport in the X and Y dimension.

The following test cases are performed:

1. X dimension in FORTRAN
2. X dimension in CUDA
3. Y dimension in FORTRAN
4. Y dimension in CUDA
5. Z dimension in FORTRAN
6. Z dimension in CUDA

## 1.2 Metrics

Results of the Thomas calculation are printed at the last time step of every simulation hour. Each time step is printed into a different file, so the deviation at each hour can be seen. If the Thomas algorithm is implemented correctly, the deviations should be very minor. However the deviation will probably grow with each time step. Three deviations are calculated:

- The average deviation aDev. Formula:

  ```
  aDev = totalDev / cellCount
  ```

- The relative average deviation in %. Formula:

  ```
  aDevRel = aDev / (absSum / cellCount)
  ```

  where absSum is the absolute sum of all values in the grid. Both positive and negative values can occur, but calculating a reliable average deviation requires a positive sum and of course totalDev is also an absolute value.

[1] Wielen, J.P. van der (2011) *Hydro-dynamics in CUDA - Performance optimizations in MOHID*

- The relative maximum deviation in %. Formula:

```
maxDevRel = (maxDev / (maxVal - minVal))
```

  where maxVal and minVal are the minimum and maximum values of all values in both grids and maxDev is the maximum absolute deviation.

These three metrics give a good view of the differences in floating point precision between the FORTRAN and the CUDA implementation.

The average deviation shows how the absolute deviation grows or diminishes throughout time steps. The relative maximum deviation shows any peaks in deviation. If suddenly a large peak appears, it is probable that the algorithm contains an error. The relative average deviation compares the average deviation to the average value. It is expected that values will grow or diminish throughout time steps, and the relative average deviation shows if the deviations grow and diminish with the average value. The relative average deviation is expected to grow.

Some cells in MOHID get a value of -9900000000000000 because no calculations should be performed on those cells. If calculations are performed on these cells, they are not taken into account in the correctness benchmark. All values <= -9900000000000000 and >= 9900000000000000 are stored as NaN, which makes the difference always 0, since abs(NaN – NaN) = NaN, treated as 0.

## 1.3 Test results and analysis

After running the benchmark, it appeared that the FORTRAN implementation became instable after approximately 14 hours, while the CUDA implementation ran stable for the complete 24 hours. The used model is very sensitive to small changes in the configuration, and apparently the configuration was not appropriate to run the model for a long time. The results shown are results for the first 14 hours.

The numerical test results are not shown here, because a large number of time steps are analyzed. Instead charts are used to give a visualization of the test results.

Solving Thomas for the X and Y dimension is only done at the even time steps. Z is solved for every time step, so there are twice as much test results for Z. It should be noted that Z solves Thomas for the velocity and X and Y solve Thomas for cohesive sediment.

### 1.3.1 Average deviation

FIGURE 1-1 – Average deviation of velocity and cohesive sediment, FORTRAN vs. CUDA shows the average deviation for all dimensions. The deviation for all dimensions grows less than linearly for the first nine hours. The deviation seems very small, but it should be noted that this deviation is the average deviation of all cells in the 122x147x52 grid. Local errors might be larger.

The deviation for the Z dimension seems a lot more stable than the deviations for X and Y. X and Y grow instable between nine and eleven hours.

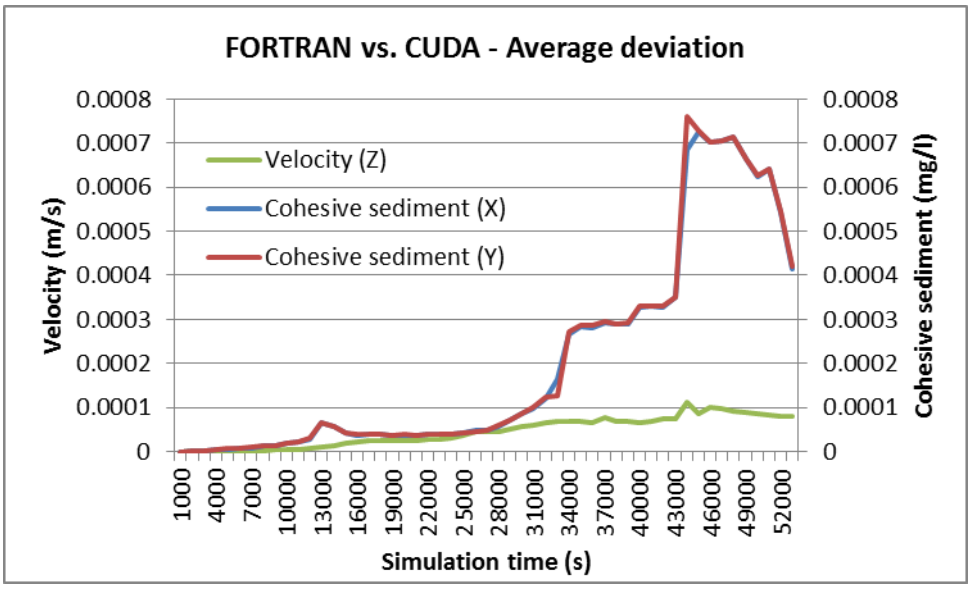


Figure 1-1 – Average deviation of velocity and cohesive sediment, FORTRAN vs. CUDA

### 1.3.2 Relative average devation

The relative average deviation (FIGURE 1-2) is very different from the average deviation; the Z dimension is less stable. This difference can be explained by the fact that the velocity values are a lot lower, they vary from -0.3 to 0.3 m/s, while the cohesive sediment values vary from -3 to 100 mg/l. The 100 value is when the model grows instable.

Overall the relative average deviation seems acceptably low for the cohesive sediment, it stays below 0.05%. The Z dimension grows above 0.3%, due to the instability in the FORTRAN implementation. Before the instability it can be said that the relative average deviation grows approximately linear for the Z dimension.

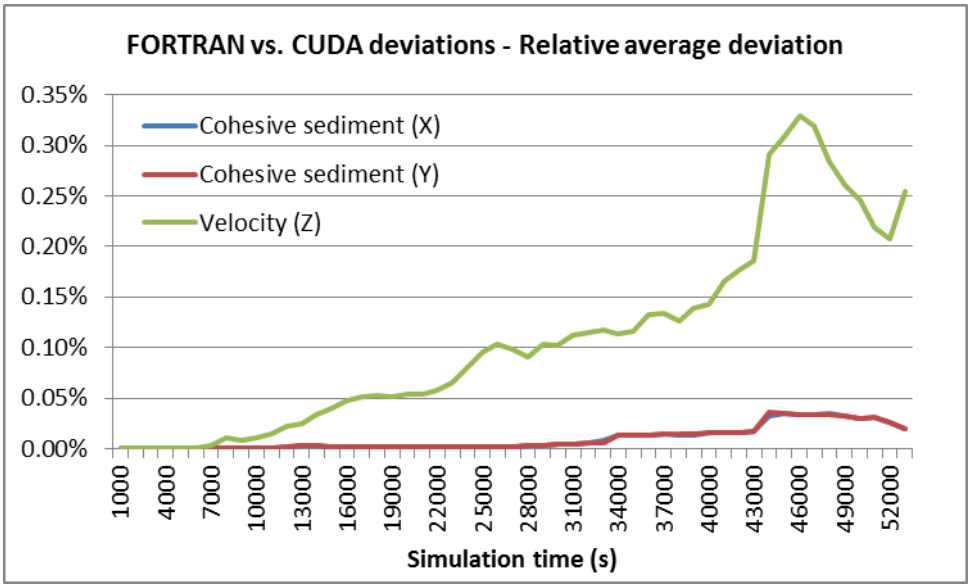


Figure 1-2 – Relative average deviation of velocity and cohesive sediment, FORTRAN vs. CUDA

### 1.3.3 Relative maximum deviation

The relative maximum deviation shows the largest local deviation. FIGURE 1-3 shows that the absolute maximum deviation generally remains below 1%, with a peak at 3h40m and eventually the instability which results in a deviation of 90%.

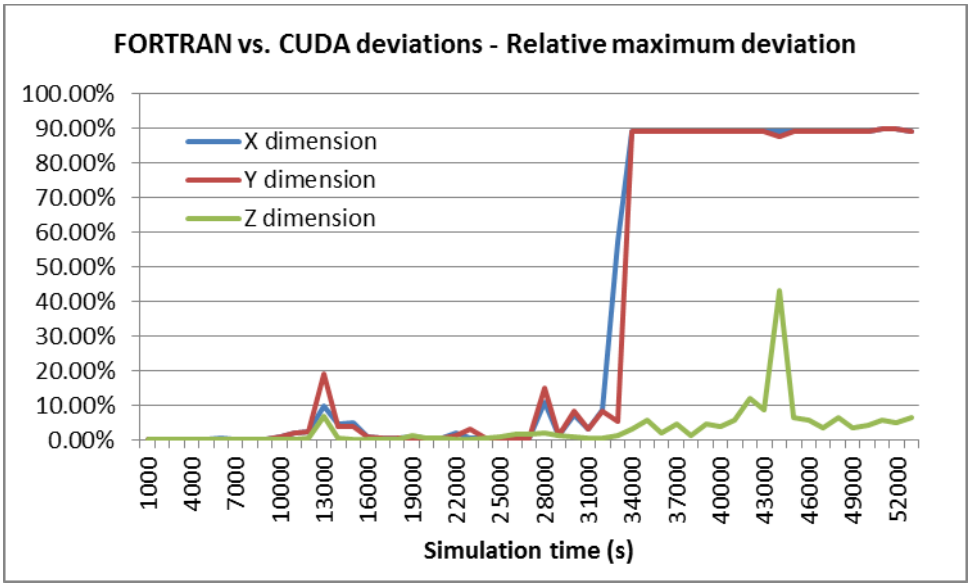


Figure 1-3 – Relative maximum deviation of velocity and cohesive sediment, FORTRAN vs. CUDA

## 1.4 Disabling FORTRAN optimizations

The fact that the FORTRAN implementation becomes unstable while the CUDA implementation remains stable is peculiar. This makes it difficult to make reliable statements about the correctness of the CUDA implementation. The implementation seems to be correct, since even the smallest error would probably have caused exponential error propagation in the very beginning of the model.

To get a better perspective on the impact of the differences, the same case has been run in MOHID without CUDA and without optimizations. If the differences between optimized and un-optimized code are larger than the differences between optimized FORTRAN code and CUDA, the impact will probably be low.

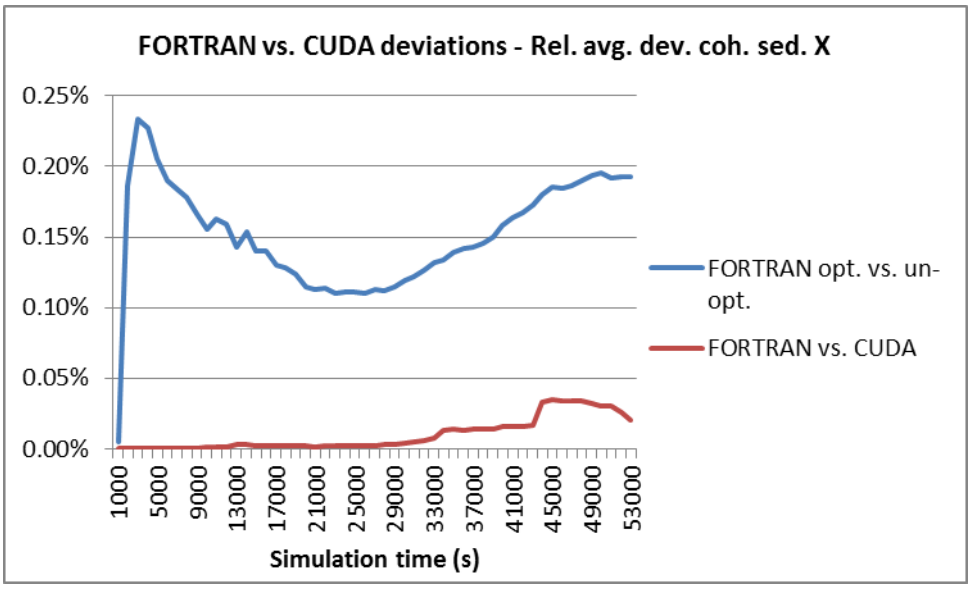


Figure 1-4 – Relative average deviation of cohesive sediment for X, FORTRAN optimized vs. un-optimized vs. CUDA

FIGURE 1-4 shows the relative average deviation for the optimized vs. un-optimized deviation FORTRAN implementations in blue and the FORTRAN optimized vs. CUDA implementation in red for the cohesive sediment in the X dimension. The results for the Y dimension are very similar so they are not shown.

It is obvious that the difference between the FORTRAN optimized and un-optimized implementation is a lot more than the difference between CUDA and FORTRAN. The FORTRAN deviation increases a lot in the first hour, but also decreases after some time. The CUDA deviation is a lot more stable, until the instability in the model appears.

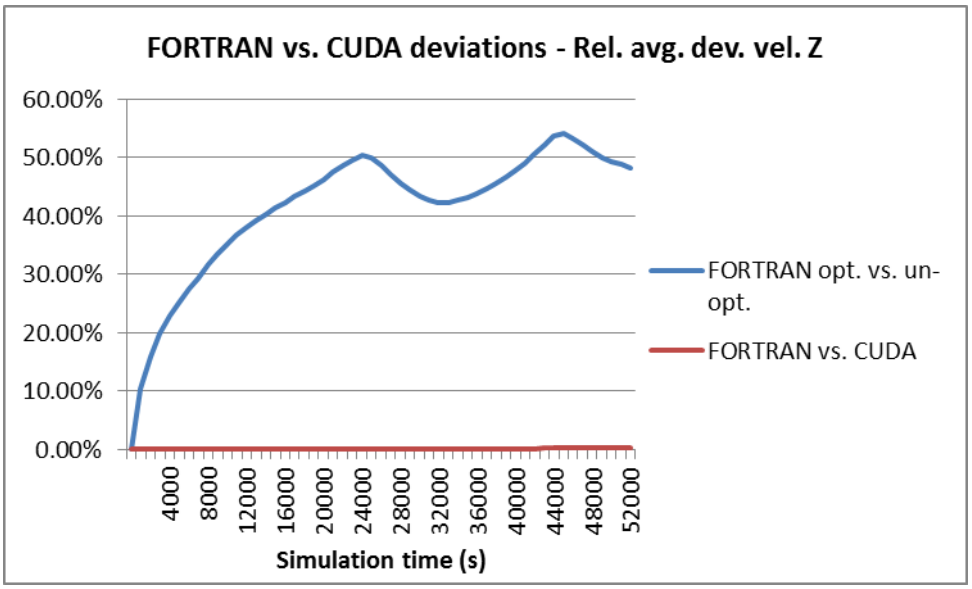


Figure 1-5– Relative average deviation of velocity for Z, FORTRAN optimized vs. un-optimized vs. CUDA

The relative differences between optimized and un-optimized code for the velocity in FIGURE 1-5 are a lot more than for the cohesive sediment. The FORTRAN vs. CUDA deviation stays below 3.5% at all times, but the optimized – un-optimized deviation grows above 50%.

In general the impact of the CUDA implementation on the accuracy is a lot lower than the impact of optimizing the FORTRAN code.

## 1.5 Visual impact of differences

The most important aspect in computational differences is what the end user sees. If the differences are graphically visible, the results of at least one implementation are unreliable, or the model is very sensitive. This paragraph compares the visual results of the three different implementations over a run of one hour. Two points in the [X,Y] grid have been chosen to show the water level.

Point [25, 31] (FIGURE 1-6) shows a difference between optimized and un-optimized code of approximately 9% (11cm) at maximum. This point is the closest point to one of the areas where the deviation in velocity was significant between optimized and un-optimized code. The CUDA line is not visible, since it is entirely covered by the optimized FORTRAN implementation. The deviation in water level is negligible, around 0.002mm at most, or 0.017%.

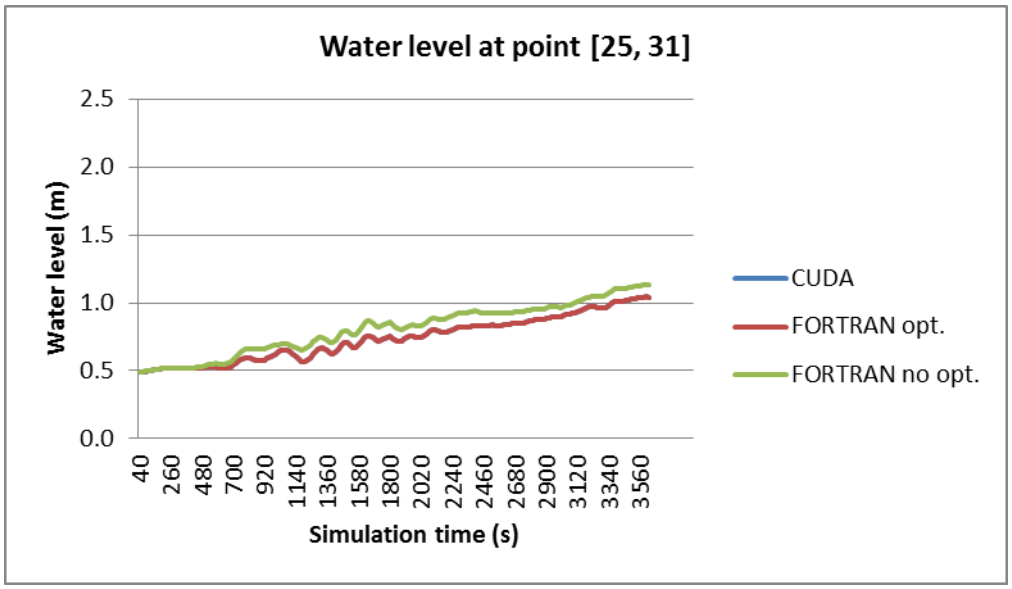


Figure 1-6 – Water level at [X=25, Y=31] throughout one hour

Point [88, 130] (FIGURE 1-7) shows a smaller deviation for the un-optimized code, at most 3cm. The CUDA deviation is at most 0.004mm, comparable to the deviation at point [25, 31].

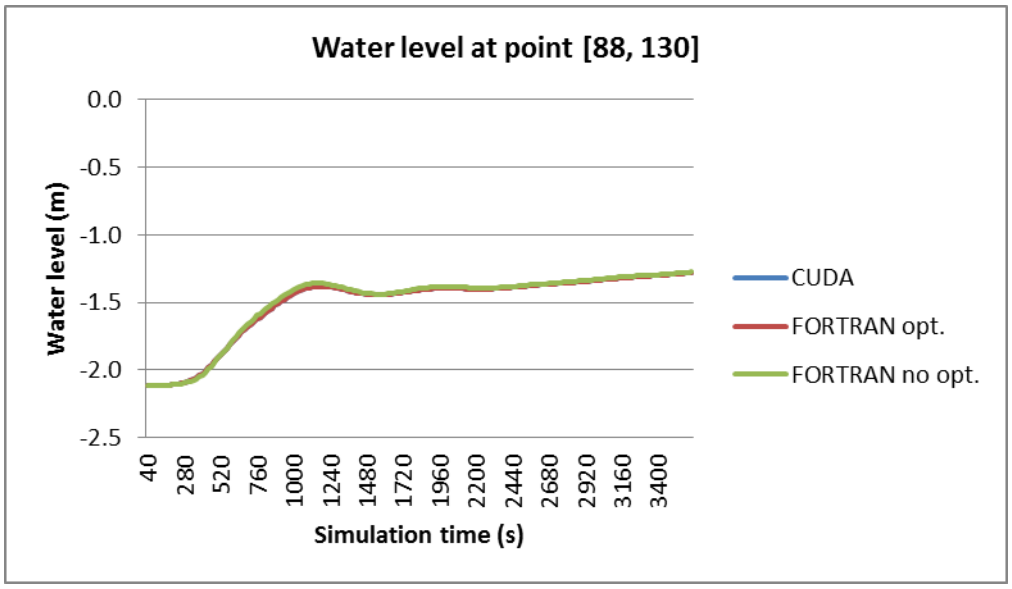


Figure 1-7 - Water level at [X=88, Y=130] throughout one hour

## 1.6 Conclusion

It is hard to estimate the correctness of the model, since the rounding errors propagate very fast. The differences between the optimized and un-optimized FORTRAN implementations are a lot larger than the differences between the FORTRAN and CUDA implementation, especially for velocity. This leads to the assumption that the algorithm is implemented correctly in CUDA and the deviations are purely caused by rounding errors which occur due to a different way of rounding and by changes in the execution order of the calculations.

Analyzing the water level showed that the used model is very sensitive to changes, for example the difference between using optimized and un-optimized code. The difference between the CUDA and FORTRAN implementation is invisible in the water level; the difference is maximal 0.017% after one hour. Performing a longer run would probably show a larger difference, since the velocity deviation is very low after a simulation time of one hour.

If another correctness benchmark would be executed, it would be wise to use a less sensitive model. Determining the difference between optimized vs. un-optimized and FORTRAN vs. CUDA would still be a good method to put the results into perspective.

# 2 Performance benchmark Thomas algorithm: CUDA

A performance benchmark is executed to compare the performance of several implementations in CUDA to each other. The best result of this performance benchmark is used in the next chapter to compare to the FORTRAN implementation.

## 2.1 Test cases

Several optimization considerations have been made to get the best implementation:

- Using page-locked memory instead of page-able memory
- Using registers for temporary variables to decrease global memory pressure
- Limiting register count per thread to increase device occupancy
- Matrix transposing to enable coalesced reading for X dimension
- Concurrent copy and execution when transposing for X dimension (column Async)

This led to the following test cases:

| # | Precision | Max registers | Dimension | Coalesced | Registers / Global | Async |
|---|---|---|---|---|---|---|
| 1. | Double | 32 | X | No | Global | No |
| 2. | Double | 32 | X | No | Registers | No |
| 3. | Double | 32 | X | Yes | Global | No |
| 4. | Double | 32 | X | Yes | Registers | No |
| 5. | Double | 20 | X | Yes | Registers | No |
| 6. | Double | Unlimited | X | Yes | Registers | No |
| 7. | Double | 32 | X | Yes | Registers | Yes |
| 8. | Double | 32 | Y | Yes | Registers | No |
| 9. | Double | 32 | Z | Yes | Global | No |
| 10. | Double | 32 | Z | Yes | Registers | No |
| 11. | Single | Unlimited | X | Yes | Registers | No |
| 12. | Single | 16 | X | Yes | Registers | No |

Table 2-1 – Performance test cases Thomas in CUDA

A number of unofficial tests made clear that the 32 registers per thread limitation had the best performance for double precision, so most tests have been done with this limitation. Also most tests have been done with double precision since MOHID cannot execute the Thomas algorithm with single precision.

The use of page-locked vs. page-able memory has not been tested because previous general tests have proved that page-locked memory is faster than page-able memory for large matrices (> 1MB)[2].

The most important optimizations only apply to the X dimension, since the X dimension has non-coalesced reads and writes by default (if no matrix transposing is applied).

The wrapper that initializes the matrices and calls the Thomas methods is written in C++. It is independent from MOHID, to avoid unnecessary complexity.

[2] Van der Wielen, J.P. (2011) *Hydro-dynamics in CUDA – Test report*, ch. 2

## 2.2 Metrics

The NVIDIA CUDA Visual Profiler has been used to measure the execution times of the test. The tests have been performed on a Tesla C1060 with 30 streaming multiprocessors with each 8 cores.

Each test takes a 128x128x128 model and executes Thomas 180x for a given dimension[3]. All dimensions are tested. For most tests the CUDA Visual Profiler has been used to determine the runtime per GPU element (kernels, copy operations).

The Visual Profiler has difficulties in profiling applications with concurrent copy and execution, so test (7.) has been done with a CPU timer only. The consequence of this is that there is no detailed information about kernel and copy operation execution times. To be able to compare this timing, all tests have not only been executed with the Visual Profiler but also with a CPU timer. The total execution time including CPU time gives a good indication whether using concurrent copy and execution is useful.

The test results for GPU times are averages of seven runs; the CPU times are averages of five runs for each test. Both GPU and CPU times do not include initialization overhead since this is a once-per-run overhead that is insignificant when running a model for a long time.

## 2.3 Test results

TABLE 2-2 shows the test results per kernel or copy operation. The copy times are approximately the same for every test, they are shown to put the kernel execution times into perspective. The execution times of host-to-device and device-to-host copies have been merged. Execution times are in milliseconds.

| # | Thomas | Transpose | Copy | Total GPU | Total CPU |
|---|---|---|---|---|---|
| 1. | 29733.7 | 0 | 2835.848 | 32569.55 | 33079.06 |
| 2. | 17056 | 0 | 2836.305 | 19892.31 | 20445.82 |
| 3. | 741.082 | 491.144 | 2835.163 | 4067.389 | 4625.694 |
| 4. | 634.972 | 491.05 | 2835.737 | 3961.759 | 4513.912 |
| 5. | 1274.41 | 491.295 | 2836.25 | 4601.955 | 5061.646 |
| 6. | 678.099 | 490.993 | 2835.889 | 4004.981 | 4553.888 |
| 7. | ? | ? | ? | ? | 4449.174 |
| 8. | 672.676 | 0 | 2838.33 | 3511.006 | 4051.58 |
| 9. | 779.492 | 0 | 2835.293 | 3614.785 | 4163.662 |
| 10. | 658.065 | 0 | 2835.299 | 3493.364 | 4045.394 |
| 11. | 238.972 | 431.078 | 1415.063 | 2085.113 | 2369.084 |
| 12. | 199.117 | 431.443 | 1408.123 | 2038.683 | 2342.408 |

**Table 2-2 – Performance test results Thomas in CUDA, numerical**

[3] *Note: the executed tests do not involve copying Res to the device, since the correctness benchmark had not been executed when this test was performed. The correctness benchmark showed that Res should be copied to the device to preserve the boundary values. The implication is that the execution time for the memory transfers increases with approximately 20%, since six instead of five transfers are now performed.*

## 2.4 Analysis

### 2.4.1 X dimension

The X dimension has the most notable variations. Tests (1.) and (2.) are very slow compared to tests (3.) through (7.), see FIGURE 2-1. This is as expected, coalesced accesses are a lot faster than non-coalesced accesses.

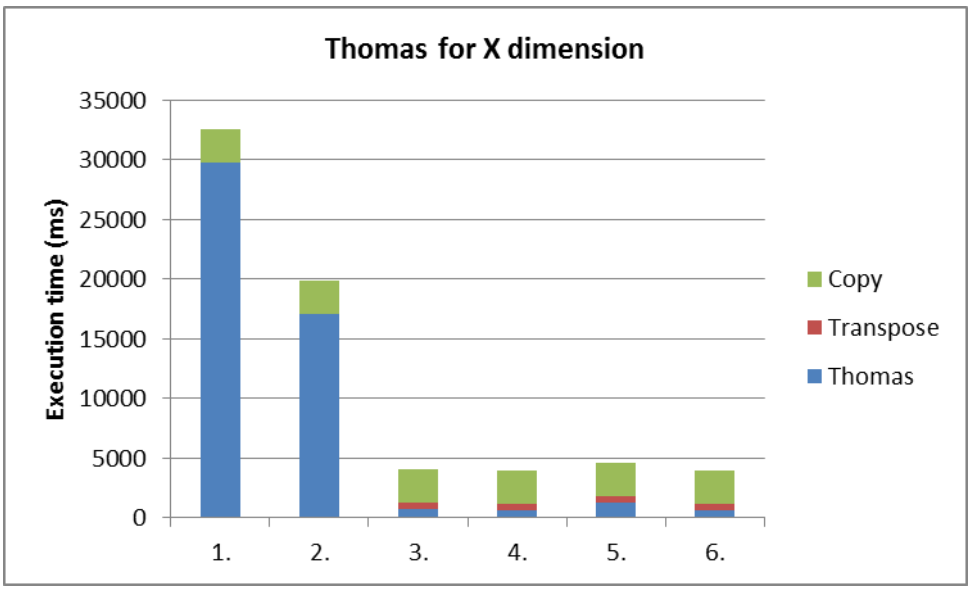


Figure 2-1 – Performance test results Thomas in CUDA, X dimension

Tests (3.) through (7.) are almost the same, with test (7.) being the fastest. (7.) is not shown in the chart since the GPU times are unknown. The CPU time of (7.) is 1.014x faster than (4.). This is hardly any speedup, which would indicate that the concurrent copy and execution hardly brings any performance gain in this case.

There is hardly any difference between restricting the register count per thread to 32 (test (4.)) and having an unlimited number of registers. The CUDA Visual Profiler shows that the number of blocks with unlimited registers is 1 per multiprocessor, and 2 per multiprocessor when limiting to 32 threads. However the doubled occupancy of the latter hardly gives any benefit, probably because the global memory has to be accessed more often. Limiting the register count to 20 in test (5.) decreases the performance. Now 3 blocks per multiprocessor are used, but the global memory becomes the bottleneck.

Using more registers in (4.) instead of global memory in (3.) makes the algorithm 1.03x faster. That the speedup is so marginal can be explained by the fact that the compiler tries to minimize the global memory accesses by using registers. The profiler shows that 29 registers are used, while only 6 registers are used directly.

### 2.4.2 Z dimension

The test results of tests (9.) with global memory and test (10.) with registers is comparable with the results of (3.) and (4.): the performance gain when using registers is 3%, which is very minimal. Both the X and Z dimension use the same method, so this does not come as a surprise.

### 2.4.3 All dimensions

The test for Y (8.) and the coalesced register test (10.) for Z are the fastest of all tests. This is as expected, the most optimizations have been applied here and there is no need for transposing. They are shown together with test (4.) for the X dimension in FIGURE 2-2. (4.) is slightly slower due to the overhead of transposing. What stands out the most is the copy overhead. Optimizing the Thomas algorithm itself any further would hardly be beneficial at this point since the copy overhead is around 80%. Any further optimizations should involve having a device with multiple copy engines and doing asynchronous copies.

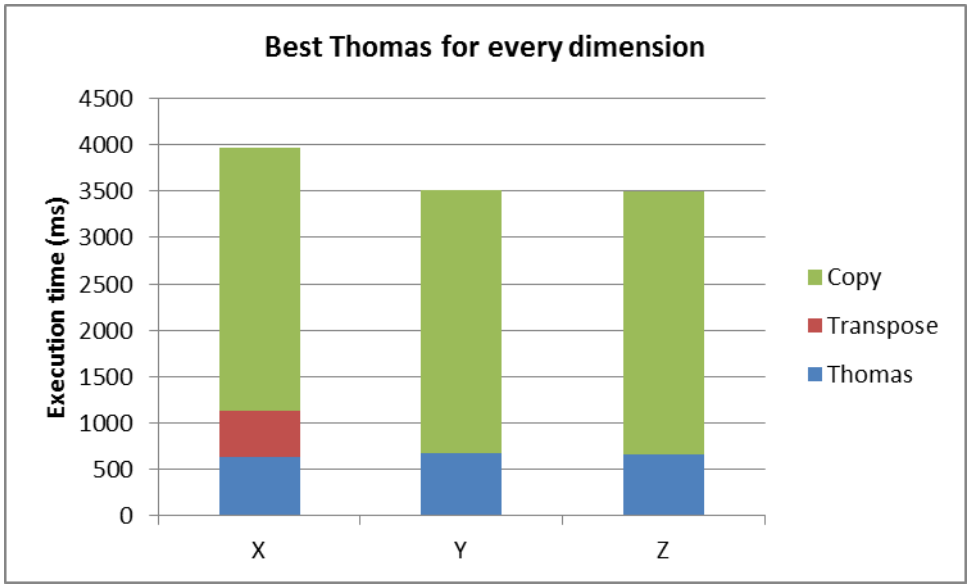


Figure 2-2 – Performance test results Thomas in CUDA, best of every dimension

## 2.5 Thomas in C++

To make developing easier, a C++ implementation of the algorithm has been made to compare output. A small test has been performed to see the performance difference between CUDA and C++. The results are shown in the FIGURE 2-3.

The matrices are stored in an X-minor manner, which means that having the X dimension as inner loop is the fastest. This is reflected in the results: there is a significant difference in execution time between the dimensions. The Z dimension is 6.9x slower than the X dimension and 3.8x slower than the Y dimension.

The Thomas algorithm is 4x, 8x and 31.3x times faster for respectively the X, Y and Z dimension.

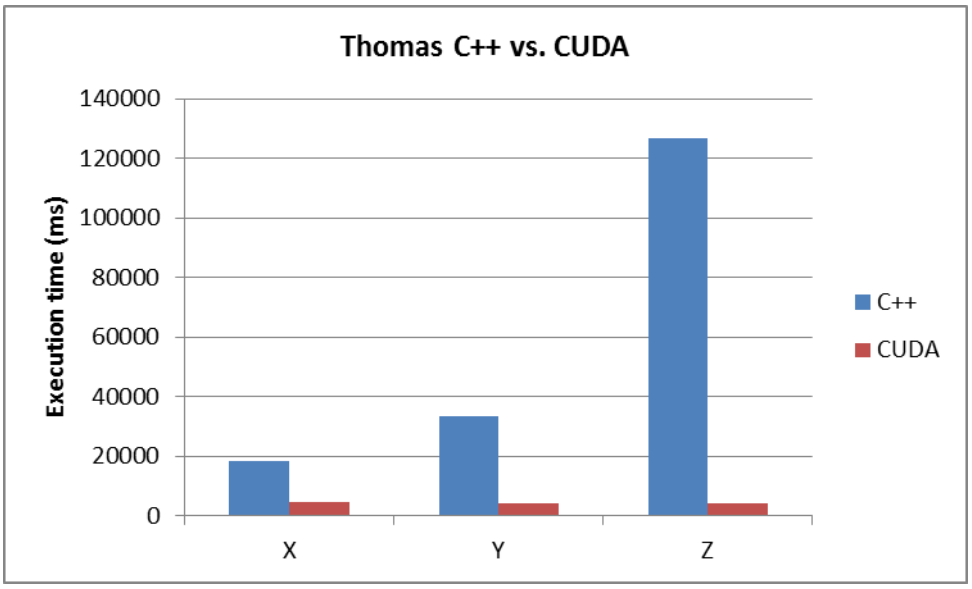


Figure 2-3 – Performance test results Thomas in CUDA, compared to C++

## 2.6 Conclusion

The most important optimization is using coalesced memory. The other optimizations show hardly any performance gain.

It is better not to restrict the maximum register count. In general the less registers are used per thread, the more the execution time will increase. This case showed that restricting the maximum register count to 32 gives a small performance gain, but this might not be the case for other applications.

Concurrent copy and execution gives a very small speed up. The speed up is small because the transposing is very fast already. However implementing concurrent copy and execution is not a lot of work if page-locked memory is already used, so it is good practice to use it anyway. The test report for CUDA has shown that using page-locked memory is always faster for large matrices, so it should be used[4].

[4] Van der Wielen, J.P. (2011) *Hydro-dynamics in CUDA – Test report*, ch. 2

# 3 Performance benchmark Thomas algorithm: CUDA vs. FORTRAN

After determining the best CUDA implementation in the previous chapter, this implementation is compared to the original FORTRAN implementation in MOHID.

## 3.1 Test cases

The CUDA vs. FORTRAN test has three test cases:

1. Thomas in FORTRAN with an explicit scheme for horizontal advection (no Thomas)
2. Thomas in FORTRAN with an implicit scheme for horizontal advection (Thomas)
3. Thomas in CUDA with page-locked memory
4. Thomas in CUDA with page-able memory

The tests involve calculating velocity, and advection and diffusion for a number of properties.

If an explicit scheme is used for horizontal advection, the Thomas algorithm is not used. In this case the Thomas algorithm is used to calculate velocity and vertical diffusion (Z dimension). The explicit vs. implicit runs in FORTRAN are performed to see the difference in performance between using these schemes.

The CUDA test with page-locked memory is performed to see the maximum speedup that can be gained with the CUDA implementation.

The test with page-able memory is executed to see the relative speed up of using page-locked memory in CUDA. If page-able memory is used, the transfers will be slower and concurrent copy and execution is disabled. Padding matrices is also disabled on both the host and the device, which will cause the matrices to be unaligned in memory. Using page-locked memory requires extra effort in programming, since all used matrices should be allocated using the CUDA wrapper. Therefore, if using page-locked memory has hardly any speed up, it is better to use page-able memory to save development time. However it is expected that using page-locked memory will improve the performance significantly.

All tests are executed in MOHID. MOHID has a Stopwatch module that can measure execution times per called method. This module is used to measure the performance of every called method. The CUDA Visual Profiler is not used for any of the tests.

The test is done for a grid with dimensions [X=122, Y=147, Z=52] and run for one hour. The test run could be longer to get a more reliable result but previous tests have shown that the execution time of CUDA kernels is very stable.

Advection and diffusion is calculated for the following sixteen properties:

- Salinity
- Temperature
- Cohesive sediment
- Oxygen
- Nitrate
- Nitrite
- Ammonia

- Dissolved non-refractory organic nitrogen
- Dissolved refractory organic nitrogen
- Particulate organic nitrogen
- Inorganic phosphorus
- Dissolved refractory organic phosphorus
- Dissolved non-refractory organic phosphorus
- Particulate organic phosphorus
- Phytoplankton
- Zooplankton

In the implicit test horizontal advection is only calculated explicitly for salinity and temperature. Advection of the other properties is calculated implicitly. Vertical diffusion is always calculated implicitly.

## 3.2 Metrics

MOHID Outwatch is used to measure the execution time of all tests.

The CUDA tests have been performed on a Tesla C1060 with 30 streaming multiprocessors with each 8 cores. The CPU tests have been performed on a system with 16GB RAM and an AMD Phenom II X6 1100T 3.3GHz processor, using one core.

## 3.3 Test results

The test results are shown in TABLE 3-1. All times are in seconds. The total time is shown; the time used for Thomas Z and Thomas X, Y is shown and the calculation time for horizontal advection is shown (Hor. adv.).

| Test | Total time (s) | Thomas Z (s) | Hor. adv. (s) | Thomas X / Y (s) |
|---|---|---|---|---|
| 1. FORTRAN, explicit | 1665.104 | 331.259 | 296.755 | 0 |
| 2. FORTRAN, implicit | 1865.722 | 334.922 | 495.821 | 167.277 |
| 3. CUDA, page-locked | 1316.384 | 42.629 | 323.134 | 31.640 |
| 4. CUDA, page-able | 1469.814 | 76.831 | 377.132 | 52.604 |

Table 3-1 – Performance test cases Thomas in CUDA and FORTRAN

## 3.4 Analysis

### 3.4.1 Implicit Thomas Z dimension

FIGURE 3-1 shows the performance gain of using CUDA for the Z dimension. The execution time for FORTRAN explicit is not shown, since the Z dimension for velocity and vertical diffusion is always solved implicitly.

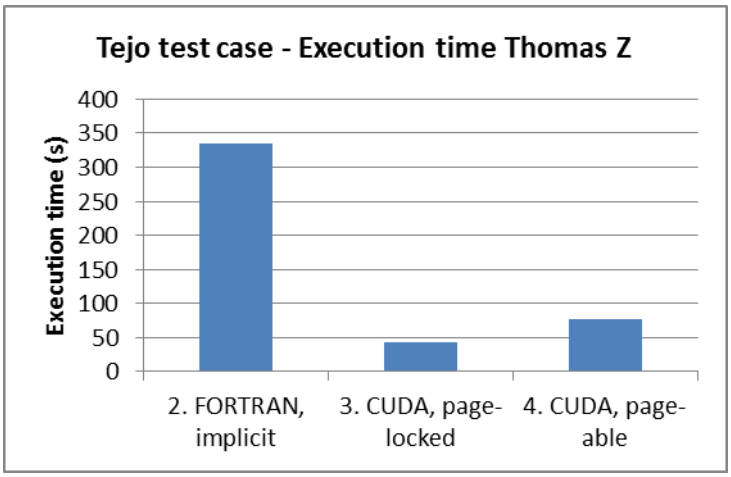


Figure 3-1 – Performance test results Thomas in CUDA and FORTRAN, Z dimension

The execution time is reduced with 87.3% when using CUDA with page-locked memory, a speed up of 7.8x. When using page-able memory the speed up is 4.4x. This can be explained by the limitations that using page-able memory imposes:

- No concurrent copy and execution
- No padded matrices, meaning decreased memory coalescing
- Memory transfers of page-able memory require and extra copy to a page-locked area in the host RAM.

These results show that using page-locked memory is worth the extra effort of programming.

### 3.4.2 Implicit Thomas X / Y dimension

FIGURE 3-2 shows the test results for the Thomas X and Y dimension. In MOHID these dimension are called in one method, Thomas_3D. Therefore no separate X and Y test results are available.

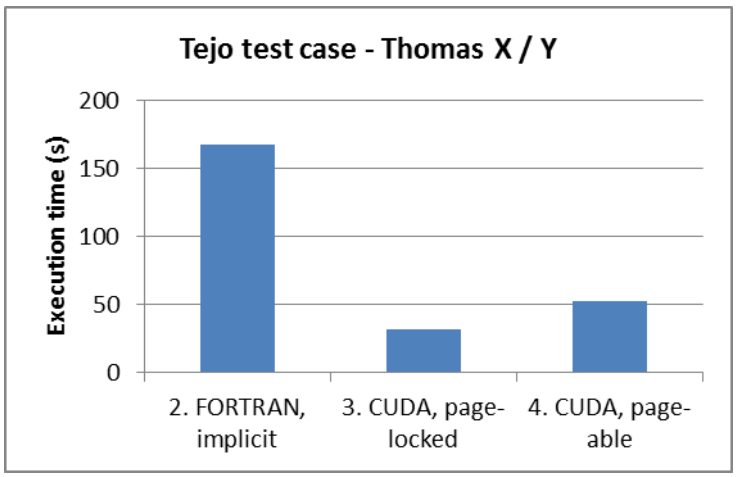


Figure 3-2 - Performance test results Thomas in CUDA and FORTRAN, X and Y dimensions

Again it is proved that using page-locked memory is a lot faster than page-able memory.

The execution time is reduced with 81.1% when using CUDA, a speed up of 5.29x. This speed up is less than the speed up of the Thomas Z dimension, firstly because probably in FORTRAN the

algorithm executes slower for the Z than for the X and Y dimension (see §2.5). When solving the algorithm for the Z dimension, no contiguous memory accesses can be performed.

Secondly the Thomas X dimension in CUDA requires a matrix transpose, as shown in the previous chapter.

Thirdly the horizontal advection for two of the sixteen properties is still executed in FORTRAN.

### 3.4.3 Explicit vs. implicit: horizontal advection

The previous paragraph showed a speed up for the X and Y dimension compared to the implicit scheme in FORTRAN. FIGURE 3-3 shows that the CUDA implementation for the whole horizontal advection process is slower than the explicit FORTRAN implementation. This can be explained by the fact that executing the Thomas algorithm is only 34% of the implicit horizontal advection in FORTRAN. The other 66% is spent calculating coefficients that are using in the Thomas algorithm. If calculating these coefficients would also be executed in CUDA, an estimated speed up of at least 3x could be achieved compared to the explicit model[5].

Note that the chart does not show the page-able implementation, since it has no added value here.

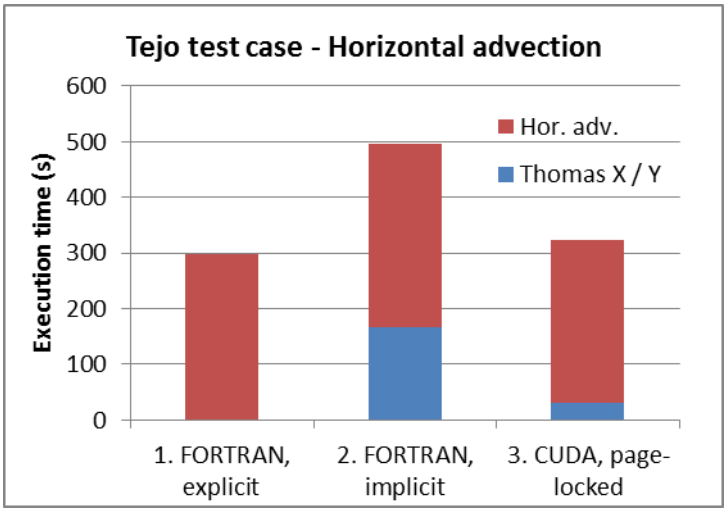


Figure 3-3 - Performance test results Thomas in CUDA and FORTRAN, X and Y dimension, horizontal advection

### 3.4.4 Overall speed up

The impact of the CUDA implementation on the performance is shown in FIGURE 3-4. The execution time is reduced with 29.4% compared to the implicit FORTRAN run, a speed up of 1.41x.

The execution time is reduced with 20.9% compared to the explicit FORTRAN run, a speed up of 1.27x.

[5] Calculated by extrapolating the speed up for the Thomas part to the coefficients part. Probably more than three times, because some transfer overhead does not have to be extrapolated.

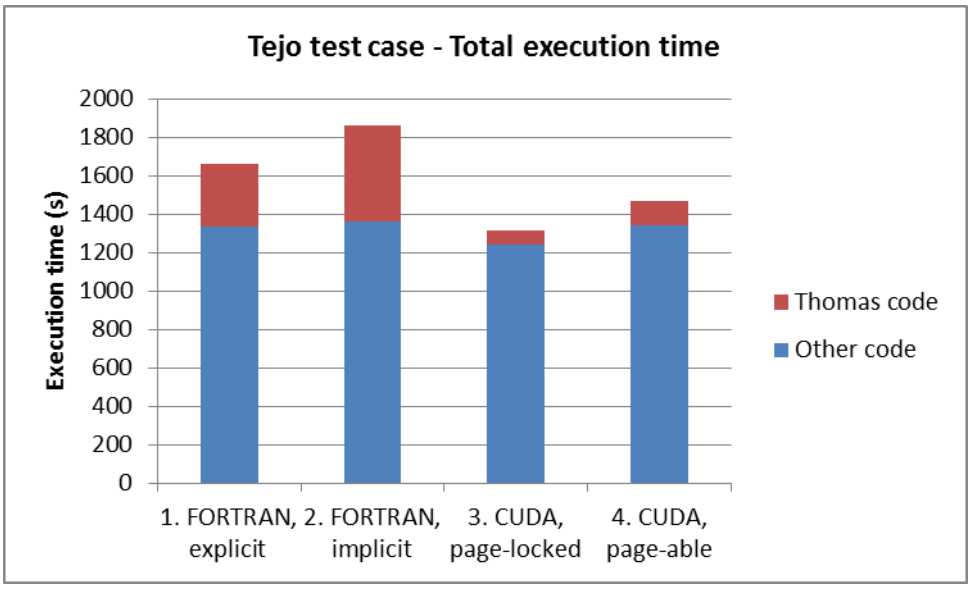


Figure 3-4 - Performance test results Thomas in CUDA and FORTRAN, total execution time

The column for test 3 shows clearly that not only the Thomas code runs faster, but the execution time of the other code also decreases by almost 9%. The column for test 4 shows that this gain is not achieved when using page-able memory, which indicates that using page-locked memory and padded arrays is not only faster for the CUDA code but for the CPU code as well.

### 3.4.5 Theoretical speed up for a whole module

The above paragraphs gave an interesting insight in how well the Thomas algorithm performs on CUDA. Speed ups of 5 to 8 times can be expected compared to a single core FORTRAN implementation. An interesting question is: what is the expected speed up when running a whole MOHID module in CUDA, for example ModuleHydroDynamic?

The most important thing when estimating this is taking the memory transfers into account. Currently the Thomas algorithm copies five matrices from host to device and one matrix from device to host per time step. This accounts for approximately 84% of the execution time for the Z dimension. The previous chapter showed that the Thomas algorithm in CUDA only uses 16% for the actual algorithm in the Z dimension, which is used in ModuleHydroDynamic.

Imagine that the whole module is run on GPU; this would eliminate the need for copying matrices between the host and the device. The only matrices that would need to be copied to the host now and then are the U and V matrices. The hydro-dynamical module is independent on the rest of MOHID; some modules require the U and V matrices as input.

If the transfers are eliminated, the Thomas algorithm is suddenly 49x faster in CUDA than in FORTRAN:

$$s_{zbase} = fortranTimeZ \;/\; cudaTimeZ = 334.92 \;/\; (42.63 * 0.16) \approx 49$$

where $S_{zbase}$ is the estimated speed up based on the Z dimension. Of course the transfer time cannot be eliminated completely, since U and V still have to be calculated. It is known that the Thomas Z takes approximately 3.5% of the total execution time of the hydro-dynamical model in FORTRAN, which means it currently takes 3.5 / 7.8 ≈ 0.45% when running in CUDA. Copying two matrices is

approximately 28% of the total execution time, which would make the total transfer time 0.45 * 0.28 = 0.13% for the complete hydro-dynamical model. For the whole module the transfer time is negligible.

§3.3.2 indicated that looping over the Z dimension is slower than looping over the X or Y dimension in FORTRAN. When extrapolating the performance for the X and Y dimension in the both manner as for the Z dimension, the results are as follows:

$$s_{xybase} = fortranTimeXY / cudaTimeXY = 167.23 / (31.64 * 0.21) \approx 25$$

where $S_{xybase}$ is the estimated speed up based on the X and Y dimensions. The 21% is the average between the 25.7% that the X dimension uses and 16.6% that the Y dimension uses for the actual algorithm in CUDA compared to the total execution time including transfers.

Some overhead should be taken into account for operations that do not benefit from execution on the GPU, operations that cannot be parallelized. A thorough study on ModuleHydroDynamic should be performed to indicate which parts cannot be parallelized. It is likely that these parts will run slightly slower on the GPU, since a single GPU thread is generally slower than a single CPU thread. The actual speed up of the CUDA implementation depends highly on the amount over overhead. If an overhead of 5% is estimated, the estimated speed up based on the results for Thomas Z is 14.4x, FIGURE 3-5. If an overhead of 2% is estimated, the speed up is 25.4x.

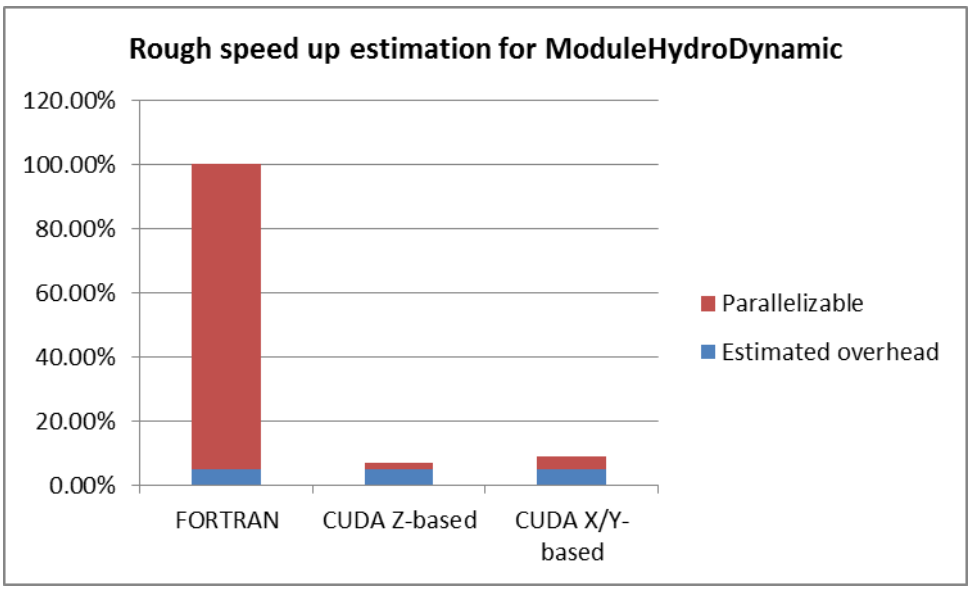


Figure 3-5 – Rough estimation of performance gain for ModuleHydroDynamic

Considering above statements it is safe to say that an estimated **speed up between 15x and 50x** can be reached compared to the single core CPU implementation of ModuleHydroDynamic in FORTRAN, a decrease in execution time between 93% and 97.8%.

## 3.5 Conclusion

The tests performed in this chapter showed that a speed up of 5.29x to 7.8x has been reached for the Thomas algorithm in MOHID compared to a single core CPU implementation, a decrease in execution time between 81.1% and 87.3%.

Besides that, using page-locked memory and padded matrices showed a decrease of 9% in execution time for the MOHID part that has not been implemented in CUDA.

An extrapolation of the CUDA results led to the assumption that a speed up of 15x to 50x can be achieved for ModuleHydroDynamic if this module is completely implemented in CUDA. This speed up is more than the currently achieved speed up since ModuleHydroDynamic is highly independent on input from other models and does not have a lot of output, eliminating the need for memory transfers between host and device.

# 4 Conclusion

The purpose of this document was to determine both the correctness and the performance gain of the implementation of the Thomas algorithm in CUDA and integrated into MOHID.

Chapter 1 showed that it is difficult to establish a reliable test case, especially if the model is sensitive to small changes. The image that is created depends highly on which data is analyzed. The velocity deviation between optimized and un-optimized FORTRAN code was 20% after a one hour run, but the water level showed a difference of 9%. The CUDA implementation appears to be correct, since the deviations are significantly lower and probably only caused by rounding errors and a different calculation ordering.

Chapter 2 helped to choose the best CUDA implementation and to determine the impact of optimizations. It appeared that assuring coalesced memory access is the most important optimization which in this case can be established by matrix transposing and smart programming. Other optimizations, like a maximum register count or concurrent copy and execution proved to be less important in this specific case.

Chapter 3 showed that a speed up **between 5.29x and 7.8x** is achieved by running the Thomas algorithm in CUDA. An extrapolation of these results showed ModuleHydroDynamic might run **15x to 50x** faster on a Tesla C1060 than on a single core of a high-end CPU if implemented completely in CUDA. This is a rough estimation that needs further research to be proven valid. The actual achievable performance gain might either be lower or higher.

Overall this document proves that running hydro-dynamical models in CUDA is feasible, it can lead to significant speed ups. An advice report will be written to weigh these gains against the financial cost and training effort of implementing GPGPU programming.