Shallow Water Riemann Solver Optimization

Group 1: Ladin Gökalpay, Nandini Joshi, Ahmed Fouad
SCCS, Fakultät für Informatik, Technische Universität München, Munich, Germany

Abstract

This is the submission for the project on Shallow Water Equations (SWE). Our objective is to implement two tasks: Vectorisation of fWave Solver using intrinsics and prepare CUDA implementation. We also generated automated regression tests for the code. FWaveSolver with AVX2 intrinsics vectorization was executed on CoolMUC2 cluster and CUDA implementation was executed on the GPU cluster and our local computer. The results presented in this report were obtained on the given systems.

Parallelisation of Shallow Water Equations(SWE) Solver

SWEs can be utilized to simulate the generation of tsunami waves in the ocean. The tsunami waves behave similarly to the waves in a shallow water pond since they initiate a horizontal flow of the water and the vertical flow in a tsunami wave simulation can be neglected. In this project, Finite Volume Method(FVM) is used to discrete SWEs in two dimensions. Height represents the third dimension and it is given as the input to the solver. The code already has a working MPI implementation. Parallelization with SIMD instructions, parallelization with CUDA and including automated regression tests are aimed at this project.

1 Introduction

- `MPI-Runner.cpp` initiates the simulation. In the `main()` function, the following steps take place:
  - Determine the MPI rank and size, local block cordinates of each block, number of grid cells, size of a single cell,compute local number of cells for each block
  - create artifical scenario, get the origin from the scenario,
  - create waveblock instance, intialize the wave propagation block. Here, `WavePropagationBlock` initializes the `Block` with the intial heights, momentum and bathymetry values on the corners.
  - initialize scenario and get the final simulation time from the scenario. After this, the values to update depth and horizontal velocities on X- and Y-directions are calculated.
  - determine checkpoints
  - define connecting blocks at the boundaries, initially exchange ghost and copy layers, get the boundary size of the ghost layers, write to file.
  - **main computation** Loop over checkpoints (for-loop), within the for-loop, for each timestep until the next checkpoint:
    - exchange ghost and copy layers
    - reset CPU clock
    - set the values in the ghost layer
    - compute numerical flux on each edge using `computeNumericalFluxes()`
    - get the maximum allowed time step of all blocks

    * determine the smallest time step of all blocks
    * update cell values, CPU time in logger, simulation time with the time step width

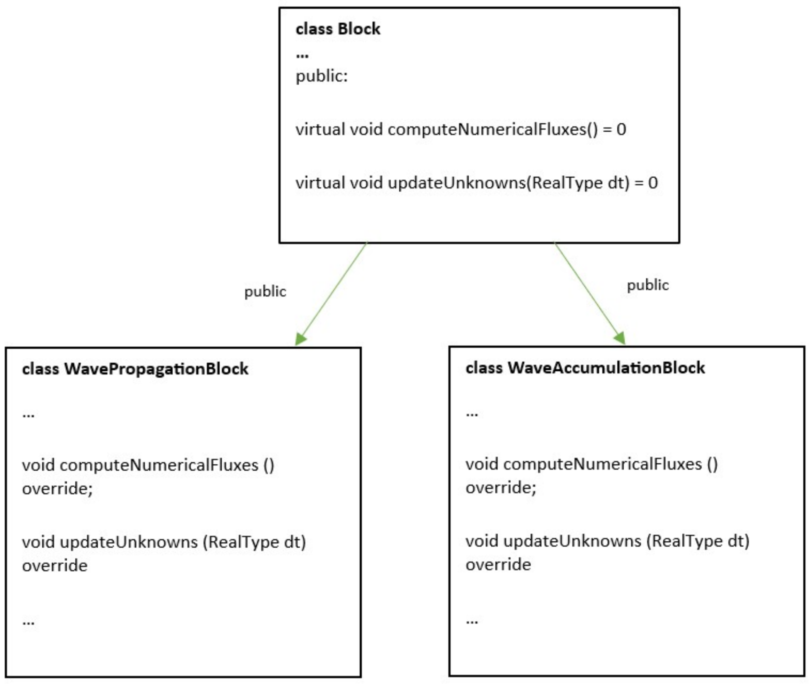

class Block
...
public:
virtual void computeNumericalFluxes() = 0
virtual void updateUnknowns(RealType dt) = 0
public
public
class WavePropagationBlock
...
void computeNumericalFluxes () override;
void updateUnknowns (RealType dt) override
...
class WaveAccumulationBlock
...
void computeNumericalFluxes () override;
void updateUnknowns (RealType dt) override
...

Figure 1: Class hierarchy for Block

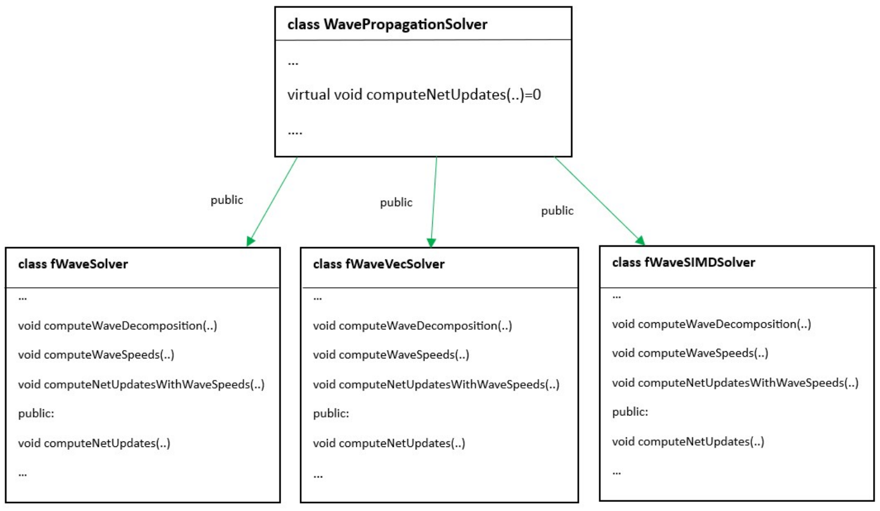

class WavePropagationSolver
...
virtual void computeNetUpdates(..)=0
....
public
public
public
class fWaveSolver
...
void computeWaveDecomposition(..)
void computeWaveSpeeds(..)
void computeNetUpdatesWithWaveSpeeds(..)
public:
void computeNetUpdates(..)
...
class fWaveVecSolver
...
void computeWaveDecomposition(..)
void computeWaveSpeeds(..)
void computeNetUpdatesWithWaveSpeeds(..)
public:
void computeNetUpdates(..)
...
class fWaveSIMDSolver
...
void computeWaveDecomposition(..)
void computeWaveSpeeds(..)
void computeNetUpdatesWithWaveSpeeds(..)
public:
void computeNetUpdates(..)
...

Figure 2: Class hierarchy for WavePropagationSolver

- `Block` class is an abstract class.1 `class WavePropagationBlock` and `class WaveAccumulationBlock` inherit as public from `class Block`. `computeNumericalFluxes()` and `updateUnknowns()` functions are defined as purely virtual functions inside the parent class and the implementation for it is defined inside each child class. In the `main()`, an instance of `WavePropagationBlock` is created. Thus, the methods defined under the `WavePropagationsSolver` class will be executed. `class WavePropagation Solver` is a parent class for all solvers defined for solving SWE.2 Some of these child classes include fWaveSolver (sequential), fWaveVecSolver (auto-vectorization), fWaveCUDASolver and fWaveSIMD solver (hold our implementations for this project)[1]. Few other solver such as augmented Reimann Solver, hybrid solver etc. are also available. Our focus for this project was on Optimizing the fWave Solver using intrinsics vectorization and CUDA.

- The fWave solver is implemented using the following main steps:[2]
  - **Treatment of 'Dry' cells:** `void determineWetDryState()` if the height of water in a cell is lesser than tolerance, it is marked as dry cell. If only one of the cells is `dry`, we assign the `h` and `hu` values according to the wall boundary condition. Otherwise, if both the cells are `dry`, it becomes a 0-velocity problem as without water, flow isn't possible.
  - **Computing the wave speeds:** `void computeWaveSpeeds(..)` Involves computation of the characteristic wave speeds and the Roe speeds using the given formulae.
  - **Solve the eigen system:** `void computeWaveDecomposition(..)` Solving the linear system of equations from the f-Wave formulation using the Einfeldt speeds.
  - **Compute net updates:** `void computeNetUpdates(..)` Compute net updates for the cell on the left/right side of the edge.

  The steps involving the treatment of dry cells and computing net-updates involves if-statements. Blending instructions can be used for vectorization here. However, the steps for computing wave speeds and solving the eigen system are purely arithmetic and can be easily vectorized. Thus, in the auto-vectorization implementation (`fWaveVecSolver.hpp`) auto vectorization was directly possible by the compiler.[2]

- The SWE simulation program contains different wave scenerios to select:
  - Bathymetry Dam Break Scenario
  - Radial Dam Break Scenario
  - Sea at Rest Scenario
  - Splashing Cone Scenario
  - Splashing Pool Scenario

  The desired scenario and the solver can be selected by the user before compiling the program by using the macro definitions `ccmake ...` By default, we calculate the results for Radial Dam Break scenario. But the other scenarios can also be tested.

- Depending on other macro settings for the run, type of solver can also be chosen. Respective solver class gets initialized and the following code is further executed.
- The outputs can be visualized using Paraview.
- As a baseline, for fWaveSolver, some optimized versions were already provided. This included:
  - **fWaveSolver:** with original sequential fWaveSolver
  - **fWaveVecSolver:** OMP SIMD based FWaveSolver using `#pragma omp simd`
  - **fWaveCUDASolver:** Same original solver with with CUDA keywords for device calls device

  Our tasks for this project were to implement solvers with intrinsics based SIMD Vectorization and CUDA.

2 Target platform for our tasks- CM2

- CoolMUC2 was used to execute the vectorization task. We used `cm2\_inter` partition of the cluster to run our code.

  Figure 3 shows the topology of cm2_inter. Detailed information about the node is presented as follows: using `-lscpu`

```
Architecture:         x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:           Little Endian
Address sizes:        46 bits physical, 48 bits virtual
CPU(s):               56
On-line CPU(s) list: 0-55
Thread(s) per core:  2
Core(s) per socket:  14
Socket(s):            2
NUMA node(s):         4
Vendor ID:            GenuineIntel
CPU family:           6
Model:                63
Model name:           Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
Stepping:             2
CPU MHz:              2599.920
CPU max MHz:          2600.0000
CPU min MHz:          1200.0000
BogoMIPS:             5199.84
Virtualization:       VT-x
L1d cache:            32K
L1i cache:            32K
L2 cache:             256K
L3 cache:             17920K
NUMA node0 CPU(s):    0-6,28-34
NUMA node1 CPU(s):    7-13,35-41
NUMA node2 CPU(s):    14-20,42-48
NUMA node3 CPU(s):    21-27,49-55

```

  The `cm2\_inter` has CPUs with 2.6 GHz clock speed. It has 14 physical CPUs (per socket) and each of them has 2 threads. Intel Xeon[3] CPUs are utilized.

- Likwid was used to generate a detailed topology of the CPU. This shows the placement of threads, details of the NUMA nodes, its distance, Cache topology with sizes and groups for each cache level. This file can be found under `Results\STREAM Benchmark\ cm2_topology_likwid`. Snippet:

```
        CPU name:       Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
CPU type:       Intel Xeon Haswell EN/EP/EX processor
CPU stepping:   2
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                2
Cores per socket:       14
Threads per core:       2
--------------------------------------------------------------------------------
HWThread        Thread        Core        Die        Socket        Available
0               0             0           0          0             *
1               0             1           0          0             *
2               0             2           0          0             *
3               0             3           0          0             *
```

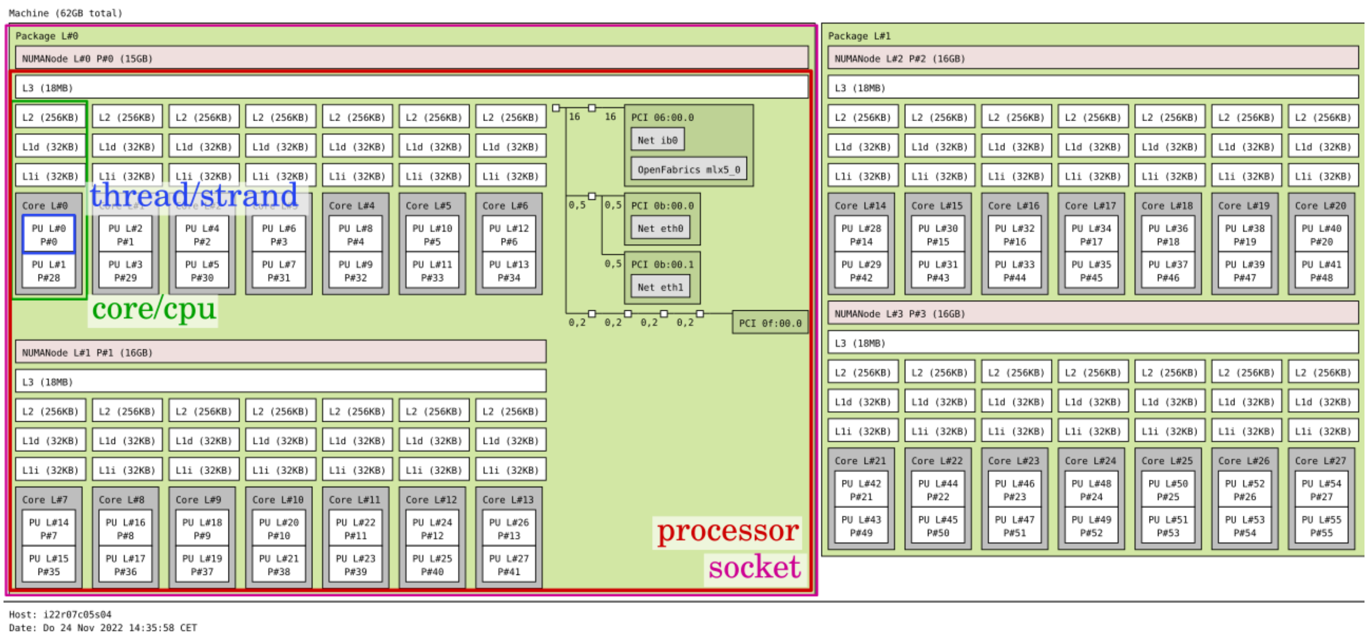


Figure 3: cm2_inter topology

```
     *
     *
55                1              27           0           1              *
--------------------------------------------------------------------------------
Socket 0:                ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 7 35 8 36 9 37 10 38
     11 39 12 40 13 41 )
Socket 1:                ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 21 49 22 50
    23 51 24 52 25 53 26 54 27 55 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                   1
Size:                    32 kB
Cache groups:            ( 0 28 ) ( 1 29 ) ( 2 30 ) ( 3 31 ) ( 4 32 ) ( 5 33 ) ( 6
     34 ) ( 7 35 ) ( 8 36 ) ( 9 37 ) ( 10 38 ) ( 11 39 ) ( 12 40 ) ( 13 41 ) ( 14
    42 ) ( 15 43 ) ( 16 44 ) ( 17 45 ) ( 18 46 ) ( 19 47 ) ( 20 48 ) ( 21 49 ) (
    22 50 ) ( 23 51 ) ( 24 52 ) ( 25 53 ) ( 26 54 ) ( 27 55 )
--------------------------------------------------------------------------------
Level:                   2
Size:                    256 kB
Cache groups:            ( 0 28 ) ( 1 29 ) ( 2 30 ) ( 3 31 ) ( 4 32 ) ( 5 33 ) ( 6
     34 ) ( 7 35 ) ( 8 36 ) ( 9 37 ) ( 10 38 ) ( 11 39 ) ( 12 40 ) ( 13 41 ) ( 14
    42 ) ( 15 43 ) ( 16 44 ) ( 17 45 ) ( 18 46 ) ( 19 47 ) ( 20 48 ) ( 21 49 ) (
    22 50 ) ( 23 51 ) ( 24 52 ) ( 25 53 ) ( 26 54 ) ( 27 55 )
--------------------------------------------------------------------------------
Level:                   3
Size:                    8.75 MB
Cache groups:            ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 ) ( 7 35 8 36 9 37
    10 38 11 39 12 40 13 41 ) ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 ) ( 21
    49 22 50 23 51 24 52 25 53 26 54 27 55 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:            4
```

```
--------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 28 1 29 2 30 3 31 4 32 5 33 6 34 )
Distances:              10 11 21 21
Free memory:            17438.7 MB
Total memory:           31362.7 MB
--------------------------------------------------------------------------------
Domain:                 1
Processors:             ( 7 35 8 36 9 37 10 38 11 39 12 40 13 41 )
Distances:              11 10 21 21
Free memory:            17365.5 MB
Total memory:           32223.8 MB
--------------------------------------------------------------------------------
Domain:                 2
Processors:             ( 14 42 15 43 16 44 17 45 18 46 19 47 20 48 )
Distances:              21 21 10 11
Free memory:            194.039 MB
Total memory:           32253.2 MB
--------------------------------------------------------------------------------
Domain:                 3
Processors:             ( 21 49 22 50 23 51 24 52 25 53 26 54 27 55 )
Distances:              21 21 11 10
Free memory:            466.586 MB
Total memory:           32252.3 MB

```

3 Optimization with SIMD

3.1 Explanation

The SIMD vectorization is only applied to the `FWaveSolver` code. The vectorized version of the FWave-Solver is saved as a new class named `FWaveSIMDSolver` in a separate header file and it is inherited from the base class `WavePropagationSolver` similar to the original serial version.

It is aimed that the function call `computeNetUpdates` inside `wavePropagationBlock.cpp` will update four values of `hNetUpdatesRight_`, `hNetUpdatesAbove_`, `huNetUpdatesRight_`, `hvNetUpdatesAbove_`, `hNetUpdatesLeft_`, `hNetUpdatesBelow_`, `huNetUpdatesLeft_`, `hvNetUpdatesBelow_` arrays (they will be named as `NetUpdate` arrays in the next parts) passed into the function `computeNetUpdate` in one for-loop iteration.

Thus the increment of column counter `j` should be changed to **four**. To make this change automatically for each calculation mode switch between SIMD-vectorized and serial, a macro variable is defined in `SIMD_defs.hpp` called **STRIDE**. It gets the value four, if the vectorization mode is enabled.

The function `computeNetUpdate` takes right and left height, bathymetry and momentum values on each edge. These values are assigned to the edges at the beginning of the program run and should not be changed in `computeNetUpdate` function call. The `NetUpdate` arrays are created outside of the FWaveSolver code and filled with the changes in the water height and momentum in the FWaveSolver.

To vectorize the code, the first approach was passing all the function inputs as pointers to the values. To be able to do so, virtual function `computeNetUpdates` under the `wavePropagationSolver` class is overloaded. Yet, this change was reversed since passing by reference and storing all four `NetUpdate` values in the reference addresses before leaving the `computeNetUpdate` function is a more convenient solution.

To vectorize the calculations in `FWaveSolver`, it is necessary to load all inputs of the function `computeNetUpdate` to SIMD vectors with `_mm256_loadu_pd()` command. The address of the input element is passed to the SIMD-load command, thus a vector holds the value in the reference address and the next three values coming after it in its original array. The constant initial values should be reachable by functions called inside the computeNetUpdate function and to simplify the calculations new values might be assigned to them inside the determineWetDryState function. But their original values should not be touched.

Therefore, a set of SIMD vectors is created in the WavePropagationSolver class and they are initialized with the constant array values. Additionally, SIMD vectors of gravity, dryness tolerance, zero tolerance and wet/dry condition are created. Since the wet/dry conditions of the neighbor cells are defined with an enum type these vectors are cast to doubles.

To prevent any conflict between the serial and vectorized code parts in the WavePropagationSolver, a macro, which is defined in the CMakeText file, is utilized to activate the vectorized variable initialization.

The computeNetUpdate function has three function calls (determineWetDryState, computeWaveSpeeds and computeNetUpdatesWithWaveSpeeds) in itself. determineWetDryState function checks the condition of the adjacent cells by comparing the height values with dryness tolerance. It writes the corresponding value to the variable wetDryState_. To vectorize this function, all single variables in this function are replaced with their SIMD vector versions which reside in the WavePropagationSolver. Height vectors are compared with the dryness tolerance and four wet/dry conditions are written to the wetDryState_v vector after controlling all possible results of the comparisons. To apply the if-condition to all four values with SIMD intrinsics _mm256_cmp_pd() command is called with _CMP_LT_OS operand to check if the first input is smaller than the second input. Then the result from the comparison is given to the command _mm256_and_pd() as the mask, to filter the vector elements which do not satisfy the if-condition. Additionally, not setting the values to zero if they do not satisfy the asked if-condition, the comparison command _mm256_andnot_pd() is used to reassign the old values to the updated vector.[4] Also, to prevent zero division in case of a vector element does not enter inside the if-case with a division operation in the serial version, the divisor is set to one. While updating the values, when the wet-dry and dry-wet cases are controlled, the result of the comparison operation will also be one for wet-wet and dry-dry cases, thus _mm256_xor_pd() command is used to have zero for wet-wet and dry-dry cases. Finally, the wetDryState_vec is updated.[5]

For the vectorized version of FWaveSolver, the arrays waveSpeeds and FWaves are transformed into arrays of four doubles and copied as many times as the number of elements in the serial version. As an example, the function computeWaveSpeeds takes two waveSpeeds arrays instead of an array of two and computes eigenvalues of the Jacobian matrices of four cell edges. This function consists of many mathematical operations and these operations are changed with SIMD intrinsics mathematical operations. In the final stage of the computeNetUpdate function, update values are set to zero, if the cell edge itself or its neighbors are defined as wall boundary. Then NetUpdate vectors are stored with the _mm256_storeu_pd() command in the original NetUPdate array addresses. The data is not aligned, therefore all load and store operations are unaligned.

For calculating the performance of the solver, we added a counter that counts the number of FLOP(s) in FWaveSolver. For example,

```
    #if defined(COUNT_FLOP)
            flop_counter += 33;
    #endif

```

Such lines were added around all floating point operations. Before executing the code, enter ccmake .., and enable COUNT_FLOP flag. Without this flag, the count won't be calculated and displayed. We ran the code for different values of nx/ny and obtained the counter values in 2 and 3 for sequential and SIMD implementations respectively. As counting of such operations can add to the runtime, we executed the code for all nx/ny values again with the COUNT_FLOP turned off. This gave us the values for runtime in 2 and 3.

Also as we move the STRIDE 4u macro to 4, we assume our grids will be in multiples of 4. Either that, or implementing a remainder handling routine. We avoided adding this extra loop in favour of a 1-to-3 additional deadl cells. implemented as a part by default of the class `Tools::Float2D<RealType>::data_`. You can check how provided an extra parameter that is the STRIDE definition, which will be appended to the original cofigured grid size, and additional remainder list. **NOTE** this list is not processed at all, it's only part of the initialization and part of the last element access to avoid accessing illegal memory or a crash.

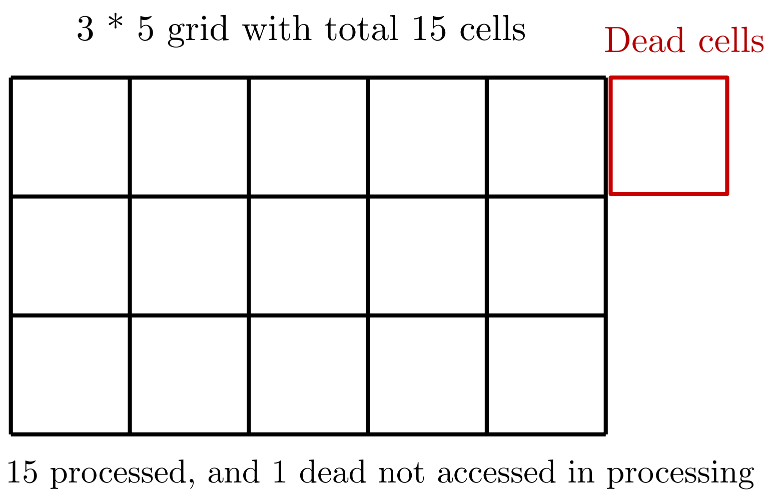

3 * 5 grid with total 15 cells
Dead cells
15 processed, and 1 dead not accessed in processing

Figure 4: Dead cells is 1 for 15 total grid size with YMM SIMDization as total allocated is 16

3.2 Progress/Achieved target

- SIMD intrinsic is the explicit way to allow the single core to handle multiple operations at the same time. In this project, AVX2 instructions are utilized for vectorization. All equations in the FWave solver are solved with doubles. AVX2 has 16 ymm registers and each register has a 64-bit line, which corresponds to 4 doubles.
- The first step of the SIMD implementation for the FWaveSolver was to replace all the operations with SIMD instructions. Taking into consideration that vectorized functions can process four doubles at the same time and the achieved speedup of approx. 4 in the article from Bader et. al on the vectorization of the shallow water equations[2] with AVX intrinsics, it is aimed to reach a speedup of around 4. However, the maximum speedup reached is around 2.8.
- With better management of registers by allocating more registers and data alignment, a higher speedup can be reached. We could not improve the FWaveSIMDsolver further due to the time constraints of the project.
- To be able to detect errors, we applied the vectorization function by function, thus if there is a problem caused by the last implemented part with AVX2 intrinsics, it can be detected easier.
- In order to make the serial code compatible with the vectorized code, the original serial code was changed in a way such that it can read four elements from the `height`, `momentum`, `velocity` and `NetUpdate` arrays when the function `computeNetUpdate` is called once. Here, the number of array elements read in a function call is determined by the value of STRIDE.
- During the implementation and testing of the SIMD vectorization task, we encountered two main problems. The first one was the vectorization of the case if some elements in the vector fulfill the dry-dry condition and their NetUpdate values should not be calculated at all

and execution of `computeNetUpdatesWithWaveSpeeds` function should be bypassed, otherwise, these values cause 0/0-error during the `NetUpdate` calculations. Since the vector elements cannot be separated and they cannot exit the function `computeNetUpdates` alone, these elements are carried out through all functions inside the `computeNetUpdate`, but all equations that are calculating their `NetUpdate` values are nullified. Although it increases the workload unnecessarily, it is the easiest and the least error-prone way. This computation load can be mitigated by checking if all vector values have dry-dry conditions (which might be the case on the boundaries) and if so, exiting the `computeNetUpdates` without any further computations.

- The second complication that slows down the process came up in the test phase. To pass the test, the new SIMD intrinsics implementation should produce the same height, momentum and bathymetry output as the original serial code. Until carrying out the automated tests, we did not realize the FWaveSIMDsolver calculates slightly different results than the FWaveSolver since the difference is around 10E-10. To solve this issue, the serial code is vectorized piece by piece and the results are checked after each change. Verifying the correctness of the `computeNetUpdatesWithWaveSpeeds` was the most cumbersome part because this function has a nested function `computeWaveDecomposition` inside. Both the outer and inner function are vectorized in turn, while the other is kept in its serial form. However, results from both cases had the same error margin. Then it has been observed that if the for-loop is calling the nested function inside `computeNetUpdatesWithWaveSpeeds` it does not cause the failure of the test; but if the for-loop is written inside the nested loop to serialize it, the test fails. The occurrence of this error only happens in the released mode. According to GCC's website [6], math flags -ffast-math, -fassociative-mat and -fno-math-errno can cause overflow, underflow of the results and in general incorrect outputs. -ffast-math flag enables aggressive math optimization, which might result in false output. -fno-math-errno disables error number calculation and depends on the IEEE exceptions for any error handling. -fassociative-mat changes the order of operands which might change the result. These flags are activated to optimize the code in the release mode in the shallow water equation simulation. Unfortunately, the confusion and the rechecking of the correctness of FWaveSIMDsolver stalled the progress of the project.

3.3 Theoretical Peak Performance

For theoretical peak performance calculation, the formula is given below:

$$P_{Peak} = ProcessingClock * FLOP * Multiplicity * Cores * Threads$$

With the `lscpu` bash command, it is printed that the CPU clock speed is equal to 2.6 GHz. Yet it is hard to predict the exact maximum counts of CPU instructions per cycle. The data to calculate speedup is taken at the end of the simulation runs with a single core and single thread. Thus we supported our calculations with the results of Intel Advisor's theoretical peak performance calculations. As can be seen in the roofline models the peak performance for double precision is 41.53 GFLOPS. 9

3.4 Executing the Code

Steps to be followed for executing our code:

```
    >>mkdir build  //if build doesn't exist
    >>cd build
    >>cmake ..
    >> ccmake .. // (optional) if needed to change the macros
    >> make -j 8
    >> sbatch -v job.sh //runs job file with run command. nx/ny can be changed here
```

First, the user should create a `build` folder to compile and run inside. The user has two options to create the Makefile with `cmake`: Running with `cmake ..` or `ccmake ..`. The first option will keep the default settings and run the serial version of the FWaveSolver code. If `ccmake` is called, the user can change the settings as shown below:

```
BUILD_SHARED_LIBS                OFF
CACHE_BINARY                     CACHE_BINARY-NOTFOUND
CACHE_OPTION                     ccache
CLANGFORMAT                      CLANGFORMAT-NOTFOUND
CLANGTIDY                        CLANGTIDY-NOTFOUND
CMAKE_BUILD_TYPE                 Debug
CMAKE_INSTALL_PREFIX             /usr/local
COUNT_FLOP                       OFF
CPPCHECK                         CPPCHECK-NOTFOUND
ENABLE_AUGMENTED_RIEMANN_EIGEN   OFF
ENABLE_CUDA                      OFF
ENABLE_DEVELOPER_MODE            ON
ENABLE_GTEST                     OFF
ENABLE_MPI                       ON
ENABLE_NETCDF                    OFF
ENABLE_OPENMP                    OFF
ENABLE_SINGLE_PRECISION          OFF
ENABLE_SWE_PROFILING             ON
ENABLE_VECTORIZATION             OFF
ENABLE_VECTORIZATION_WITH_SIMD   OFF
ENABLE_VISUALIZER                OFF
ENABLE_WRITERS                   OFF
FETCHCONTENT_BASE_DIR            /dss/dsshome1/lxc0D/t1221am/project/SWE_lad/build/
   _deps
FETCHCONTENT_FULLY_DISCONNECTE   OFF
FETCHCONTENT_QUIET               ON
FETCHCONTENT_SOURCE_DIR__PROJE
FETCHCONTENT_UPDATES_DISCONNEC   OFF
FETCHCONTENT_UPDATES_DISCONNEC   OFF
GTest_DIR                        GTest_DIR-NOTFOUND
OPT_DISABLE_EXCEPTIONS           OFF
OPT_DISABLE_RTTI                 OFF
OPT_ENABLE_BUILD_WITH_TIME_TRA   OFF
OPT_ENABLE_CACHE                 ON
OPT_ENABLE_CLANG_TIDY            ON
OPT_ENABLE_CONAN                 OFF
OPT_ENABLE_COVERAGE              OFF
OPT_ENABLE_CPPCHECK              ON
OPT_ENABLE_DOXYGEN               OFF
OPT_ENABLE_INCLUDE_WHAT_YOU_US   OFF
OPT_ENABLE_INTERPROCEDURAL_OPT   OFF
OPT_ENABLE_NATIVE_OPTIMIZATION   OFF
OPT_ENABLE_PCH                   OFF
OPT_ENABLE_SANITIZER_ADDRESS     ON
OPT_ENABLE_SANITIZER_LEAK        OFF
OPT_ENABLE_SANITIZER_MEMORY      OFF
OPT_ENABLE_SANITIZER_THREAD      OFF
OPT_ENABLE_SANITIZER_UNDEFINED   OFF
OPT_ENABLE_UNITY                 OFF
OPT_ENABLE_VS_ANALYSIS           OFF
OPT_WARNINGS_AS_ERRORS           OFF
USED_SCENARIO                    RadialDamBreakScenario
WITH_SOLVER                      FWave
```

Since this task is the vectorization of the FWaveSolver, the user should set the option `WITH_SOLVER` to `FWave`. The scenario selection is possible by toggling the `USED_SCENARIO` selections. Before the implementation of the SIMD-vectorised FWaveSolver, there was `ENABLE_VECTORIZATION` option in ccmakemenu available, but it was to activate the unfinished implementation of the `FWaveVecSolver`. To ease the selection of the SIMD-vectorised FWaveSolver, `ENABLE_VECTORIZATION_WITH_SIMD` is added as a sub-option of the `ENABLE_VECTORIZATION`. The option `ENABLE_VECTORIZATION` is not deactivated to not restrict possible future implementation of the FWaveVecSolver.

Thus, to run vectorized `FWaveSIMDsolver` code both modes should be switched on. `ENABLE_GTEST` can be switched on to check the correctness of the results of the result from the vectorized code version. If the program is run in the cluster, GTest library might not be supported. The options `ENABLE_NETCDF` and `ENABLE_WRITERS` are added to the list to shorten the run-time in case no visual output is desired. The user should enable either the `writers` or the `netCDF` to get a visual output. Calculations with single precision should not be enabled if the FWaveSIMDsolver code is to be used, because it is only able to run for AVX2 intrinsic with double precision. Both serial and vectorized version of the FWaveSolver count the FLOPs if `COUNT_FLOP` is activated. Turning on the option `ENABLE_SWE_PROFILING` measures the total wall clock time elapsed to run the whole simulation. After configuring with desired set up, Makefile is generated and the executable file is created in the `build` folder. To run the simulation program and save the output a SLURM script is written. The user can run the executable named `SWE-MPI-Runner` by calling the batch script. The size of the matrix can be adjusted by changing the cell number on the -X and -Y directions in the bash script. Then the simulation program will be run on the `cm2_inter`.

```bash

#!/bin/bash
# SLURM
#SBATCH -J swe_original_icc_i22r07c05s
#SBATCH -o output_icc_i22r07c05s.log
#SBATCH -e error_icc_i22r07c05s.log
#!SBATCH -D ./
#SBATCH --get-user-env
#SBATCH --time=00:03:00
#SBATCH --verbose
#SBATCH --mail-type=end
#SBATCH --mail-user=ge72yut@tum.de
#SBATCH --partition=cm2_inter
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=1
#SBATCH --ntasks-per-socket=1
#SBATCH --threads-per-core=2
#SBATCH --cpus-per-task=1
#SBATCH --sockets-per-node=2

export SLURM_CPU_BIND=verbose

scontrol -dd show job $SLURM_JOB_ID > scontrol.out

./SWE-MPI-Runner -x 500 -y 500
```

3.5 Verification

To validate the results of the vectorized FWaveSolver, height, momentum and bathymetry results are compared with the results from the original serial solution of the problem. As stated in the explanation of the vectorized FWaveSolver, due to the aggressive math optimization flags, the output of the simulation with the `Release` build mode contradicts the results of the not vectorized code. The error appears after the tenth digit. Since this error margin is caused by the optimization flags and not because of the incorrect vectorization of the FWaveSolver, a tolerance is introduced to overleap this difference in the results, so if the error margin is smaller than `1E-10` the test will pass successfully.

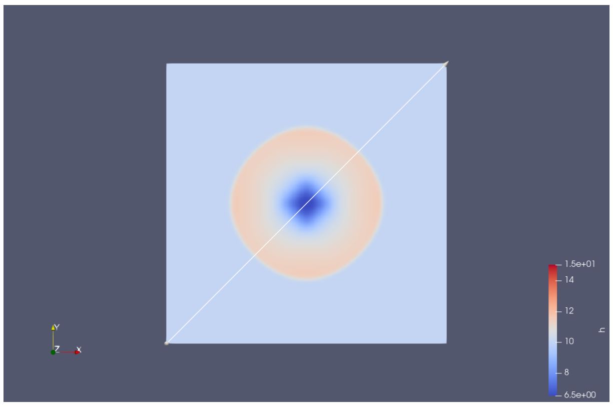
3.6 Validation

Test case for validation of the FWaveSIMD Solver is defined in `Tests/FWaveSIMDSolverTest.cpp`. Here, the tasks include determining the local block coordinates of each block, creating a simple artificial scenario, computing the size of a single cell, getting the origin from the scenario, initializing the wave propagation block and getting the final simulation time from this scenario. While looping over the checkpoints we check the results from both FWaveSolver and FWaveSIMDSolver.

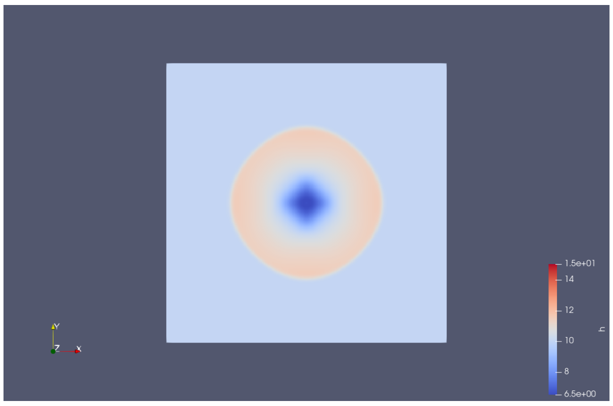
```
        // Loop over checkpoints
    // Do time steps until next checkpoint is reached
    while (simulationTime < endSimulationTime)
    {
        // Set values in ghost cells
        waveBlockFWaveSIMDSolver->setGhostLayer();
        // Compute numerical flux on each edge
        waveBlockFWaveSIMDSolver->computeNumericalFluxes();
        RealType maxTimeStepWidth = waveBlockFWaveSIMDSolver->getMaxTimeStep();
        // Update the cell values
        waveBlockFWaveSIMDSolver->updateUnknowns(maxTimeStepWidth);
        LOG_DBG("[FWaveSIMDSolver] Max TimeStep Width= " << maxTimeStepWidth);
        // Update simulation time with time step width
        simulationTime += maxTimeStepWidth;
    }
```

Similarly for the `waveBlockFWaveSolver` instance in place of `waveBlockFWaveSIMDSolver`. Then we compare the values for the parameters height, velocities in x- and y- directions, and bathymetry `h`, `hu`, `hv`, `b`. We check for all points the values of these parameters, if the difference is greater than the `kFaultTolerance`, test fails for that particular parameter measurement. We display the message accordingly.

Paraview[7] was used for visualizing the output for different scenarios. The following results were generated:



(a)



(b)

Figure 5: Radial Dam Break Scenario Results from Paraview

Figure **??** illustrates the heights at the end of the dam-break-simulation time. One can see the difference between the two results cannot be observed. However as mentioned earlier, aggressive math optimization causes ignorable differences between the serial and vectorized run. Also, one can see on the plot 6 below the results perfectly overlap.

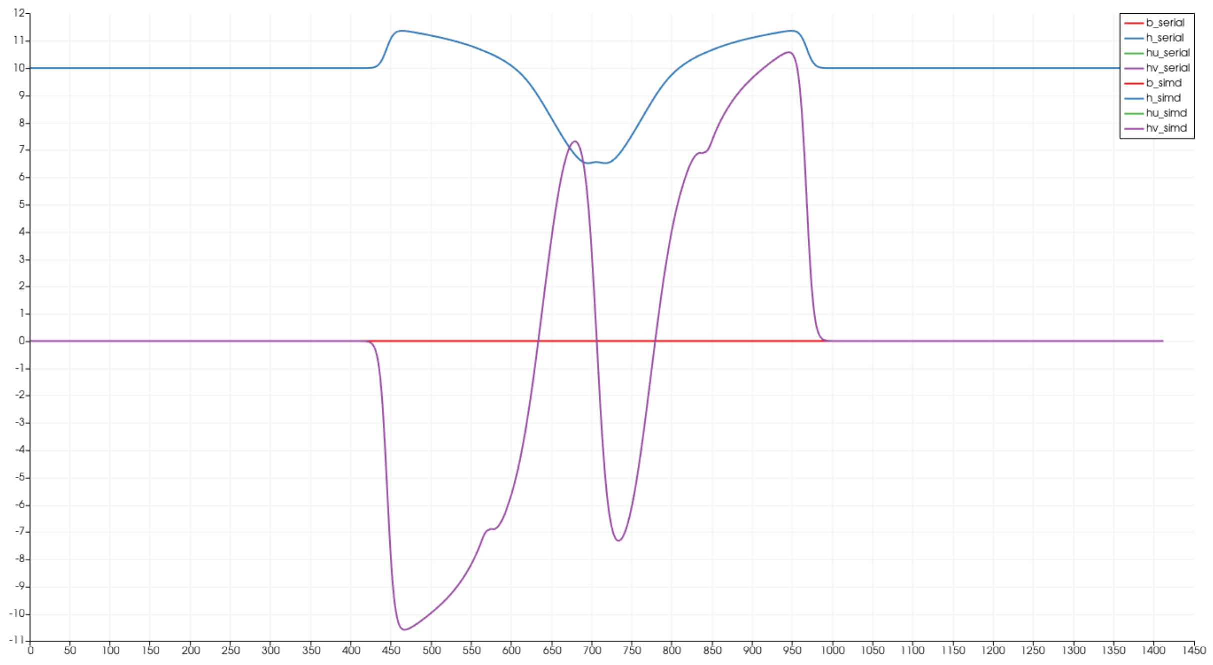


Figure 6: Results from Radial Dam Break Scenario

While testing a greater difference than 10E-10 is observed in momentum results from `splashingConeScenario` and `BathymetryDamBreakScenario`. However, the deviation is hard to remark on the plots. The plot 7 below visualizes the serial and vectorized `BathymetryDamBreakScenario` case results, and the difference is imperceptible.

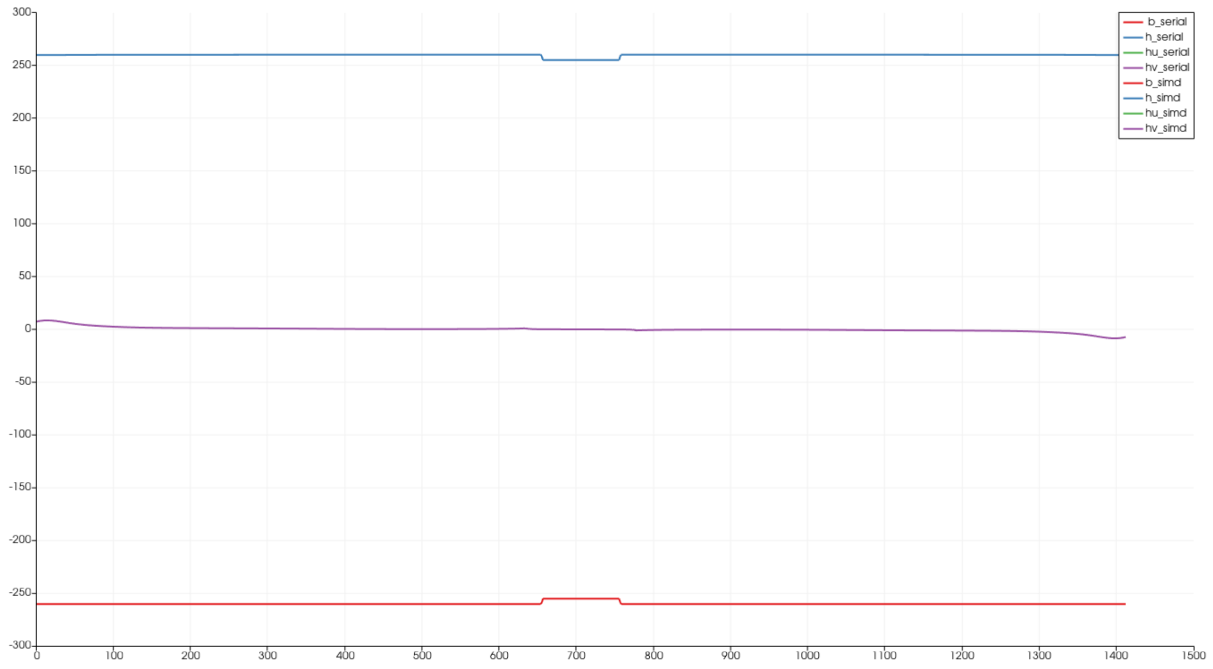
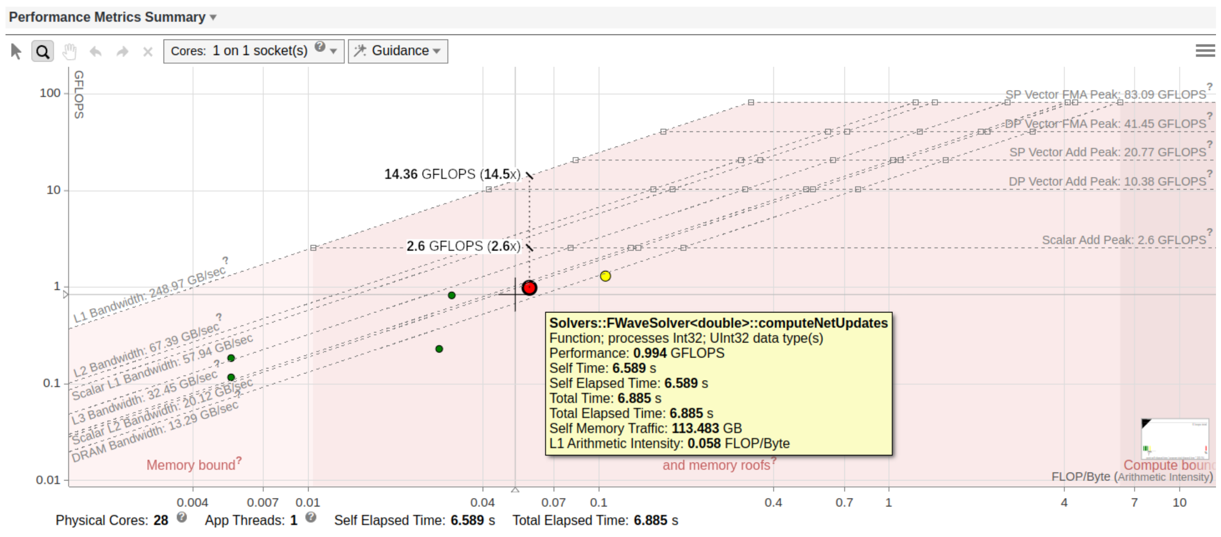


Figure 7: Results from Bathymetry Dam Break Scenario

3.7 Profiling Results

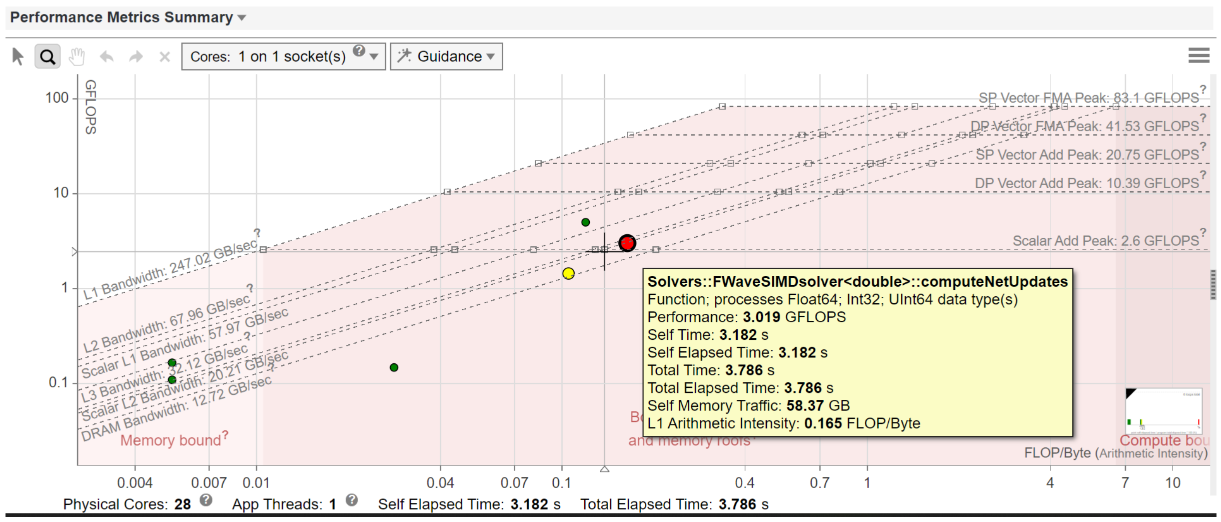
We used the Intel Advisor tool[8] to generate the roofline model for our code. On the cluster, we ran our jobscript `job.sh` to generate the files which were viewed on the Intel Advisor desktop app. The command used to generate the files for the roofline model from the jobscript was `advixe-cl --collect =roofline --project-dir=. --search-dir src:r=../ ./SWE-MPI-Runner -x 500 -y 500`. We used the value of 500 for elements in each dimension to generate the roofline model. For values smaller than this, the advisor gave an error as the provided data/runtime was not sufficient.



Solvers::FWaveSolver<double>::computeNetUpdates
Function; processes Int32; UInt32 data type(s)
Performance: **0.994** GFLOPS
Self Time: **6.589** s
Self Elapsed Time: **6.589** s
Total Time: **6.885** s
Total Elapsed Time: **6.885** s
Self Memory Traffic: **113.483** GB
L1 Arithmetic Intensity: **0.058** FLOP/Byte

Physical Cores: **28** App Threads: **1** Self Elapsed Time: **6.589** s Total Elapsed Time: **6.885** s

Figure 8: Roofline Model for sequential code



Solvers::FWaveSIMDsolver<double>::computeNetUpdates
Function; processes Float64; Int32; UInt64 data type(s)
Performance: **3.019** GFLOPS
Self Time: **3.182** s
Self Elapsed Time: **3.182** s
Total Time: **3.786** s
Total Elapsed Time: **3.786** s
Self Memory Traffic: **58.37** GB
L1 Arithmetic Intensity: **0.165** FLOP/Byte

Physical Cores: **28** App Threads: **1** Self Elapsed Time: **3.182** s Total Elapsed Time: **3.786** s

Figure 9: Roofline Model for SIMD vectorized code

Figures 8 and 9 present the roofline models for the sequential and SIMD vectorized codes for FWave Solver respectively. From these plots, we can understand the improvement in performance with vectorization. The theoretical peak for double precision vector FMA appears at 41.53 GFLOPS. When we changed the cores-on-socket setting to 14 cores on 1 socket, the peak performance for double precision with vectors increased to 581.3 GFLOPS. It can also be observed from 9 that the implementation for vectorization is compute-bound.

3.8 Results

Using different flags in ccmake, we were able to modify the mode of execution for the code.

Mode	count FLOP	OpenMP	Vect.	Vec. with SIMD	SWE Profiling
Sequential	ON	OFF	OFF	OFF	ON
with SIMD	ON	ON	ON	ON	ON

Table 1: ccmake edits to run different versions

As shown in table 1, we need to change the flags in ccmake to enable various features. These settings were used to generate the results presented in this section. The utilized scenario was RadialDamBreakScenario to create table 2 and 3. As the counting of floating point operations increases additional run-time, we executed the code again with ENABLE_COUNT_FLOP turned off in order to obtain an accurate run-time measurement.

elements per dim.	counter (GFLOP)	runtime (s)	Performance (GFLOPS)
100	0.0688416	0.0633243	1.08712769
200	0.5480064	0.490997	1.116109467
400	4.4186992	3.93251	1.123633303
500	8.608182	7.66179	1.123521005
600	14.9007132	13.3329	1.117589812
800	35.3965104	31.6867	1.11707784
1000	69.223154	62.0536	1.115538083
1200	119.720484	108.063	1.107876739

Table 2: Performance of the given sequential code (without vectorization)

elements per dim.	counter (GFLOP)	runtime (s)	Performance (GFLOPS)
100	0.097416	0.0342572	2.843664981
200	0.769824	0.247324	3.112613414
400	6.184332	1.97052	3.138426405
500	12.038895	3.85691	3.121383439
600	20.828907	6.66071	3.127130141
800	49.448124	15.9098	3.108029265
1000	96.667065	31.2398	3.094356078
1200	167.14269	53.7314	3.110707891

Table 3: Performance of the (SIMD) intrinsics vectorized code for FWave Solver

elements per dim.	**seq. perf. (GFLOPS)**	**SIMD perf. (GFLOPS)**	**SpeedUp**
100	1.08712769	2.843664981	2.615759866
200	1.116109467	3.112613414	2.788806569
400	1.123633303	3.138426405	2.793105542
500	1.123521005	3.121383439	2.778215472
600	1.117589812	3.127130141	2.798101869
800	1.11707784	3.108029265	2.78228531
1000	1.115538083	3.094356078	2.773868615
1200	1.107876739	3.110707891	2.807810457

Table 4: Speed Up with Vectorization

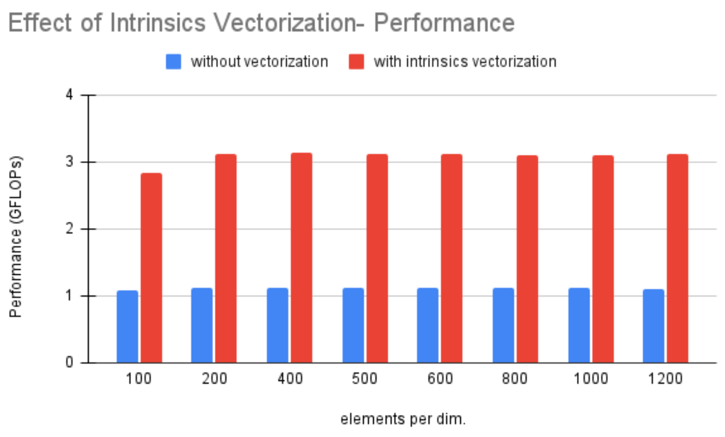


Figure 10: Effect of Intrinsics Vectorization- Performance

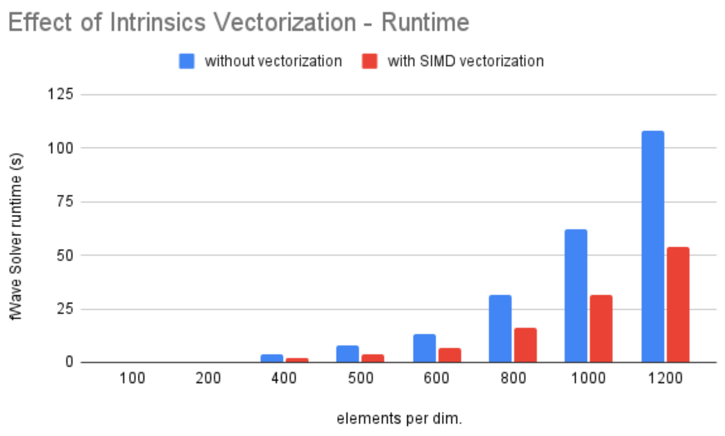


Figure 11: Effect of Intrinsics Vectorization- Runtime

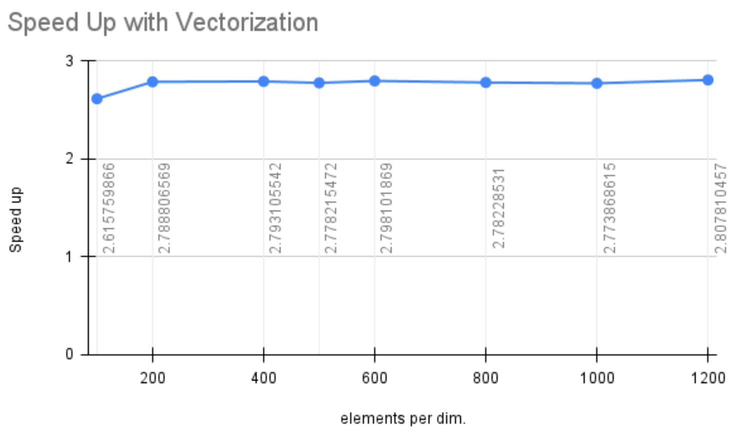


Figure 12: Speed Up with Vectorization

Tables 2 and 3 represent the results obtained for different values of elements per dimension (nx,ny). Counter takes into account all floating point operations taking place in FWaveSolver implementation and runtime is the time spent in FWave Solver functionality [computeNumericalFluxes]. Performance was calculated as counter divided by runtime in GFLOPS. We linearly increased the number of elements per dimension, to obtain further results but for increased sizes beyond 1200 we could not get complete output as it started to give timeout error. Table 4 depicts the calculated speedup with SIMD intrinsics vectorization. We obtained around 2.8 speedup with intrinsics vectorization. For comparison, the auto-vectorization as presented in the paper [2], where a speedup of 4.2 is obtained on a single core.

Figure 10 shows the effect of vectorization on the performance of the Solver for different number of elements per dimension. For each case, performance is better for vectorized output. In contrast to this, Figure 11 shows the time spent in FWave Solver implementation for different values on X axis. Unlike the performance, runtime is lesser for vectorized output as performance is inversely proportional to the runtime. Figure 12 plots the speedup calculated for each of these cases between the sequential and SIMD vectorized programs. We can interpret from the result that speedup is almost constant around 2.7x.

All the above results were generated for default scenario RadialDamBreakScenario. To study the performance of the solver under different scenarios, we analyzed the results of sequential and SIMD vectorized code for each of these scenarios (nx/ny=500). The results were as follows:

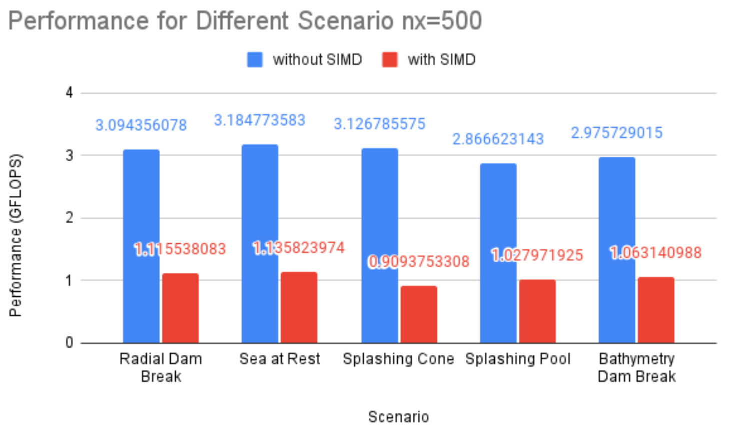


Figure 13: Performance for Different Scenario nx=500

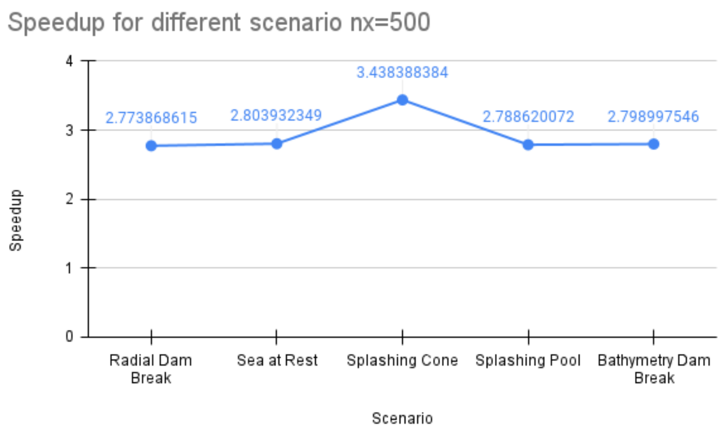


Figure 14: Speedup for different scenario nx=500

Scenario	Counter(GFLOP)	Runtime(s)	Performance (GFLOPS)
Radial Dam Break	69.223154	62.0536	1.115538083
Sea at Rest	44.072469	38.8022	1.135823974
Splashing Cone	86.57444118	95.2021	0.9093753308
Splashing Pool	33.43674	32.5269	1.027971925
Bathymetry Dam Break	33.863592	31.8524	1.063140988

Table 5: Performance under different scenarios for sequential code nx=500

Scenario	Counter(GFLOP)	Runtime(s)	Performance (GFLOPS)
Radial Dam Break	96.667065	31.2398	3.094356078
Sea at Rest	61.6371525	19.3537	3.184773583
Splashing Cone	195.8559075	62.6381	3.126785575
Splashing Pool	46.76265	16.3128	2.866623143
Bathymetry Dam Break	47.35962	15.9153	2.975729015

Table 6: Performance under different scenarios for sequential code nx=500

Scenario	Speedup
Radial Dam Break	2.773868615
Sea at Rest	2.803932349
Splashing Cone	3.438388384
Splashing Pool	2.788620072
Bathymetry Dam Break	2.798997546

Table 7: Speedup for different scenario nx=500

All of the output log files can be found under Results folder.

3.9 Conclusion and Further Scope of Improvement

We were able to run a vectorized SIMD code with AVX2 intrinsics for FWaveSolver and get a speed up of 2.8. We tested our code for different test cases, matrix sizes and scenario. The paper [9] presented the component analysis of the execution times showed that the benefits of vectorization clearly compensate for the overhead introduced to make the simulation data structures suitable for SIMD instructions.

Further improvement in the speedup can be obtained by optimizing the cache misses, and moreover using slicing and blocking accordingly. Substantial improvements could be gained by fusing the net-update computation with the final update of unknowns, which leads to a stencil-type scheme, but requires a modified time-step-size control.[2]

4 Optimization with CUDA

4.1 Introduction

GPU acceleration is an industry must-do task when it comes to scalable parallelization solutions. In this study we analyze the impact of NVIDIA's GPU accelerator's offloading using the CUDA API on the SWE code.

4.2 The GPU Offloading Environment

As CUDA is target dependent, we restrict our runs on the same compatibility platform. Specifically, 8.6. This is A100 Ampere Architecture [10].

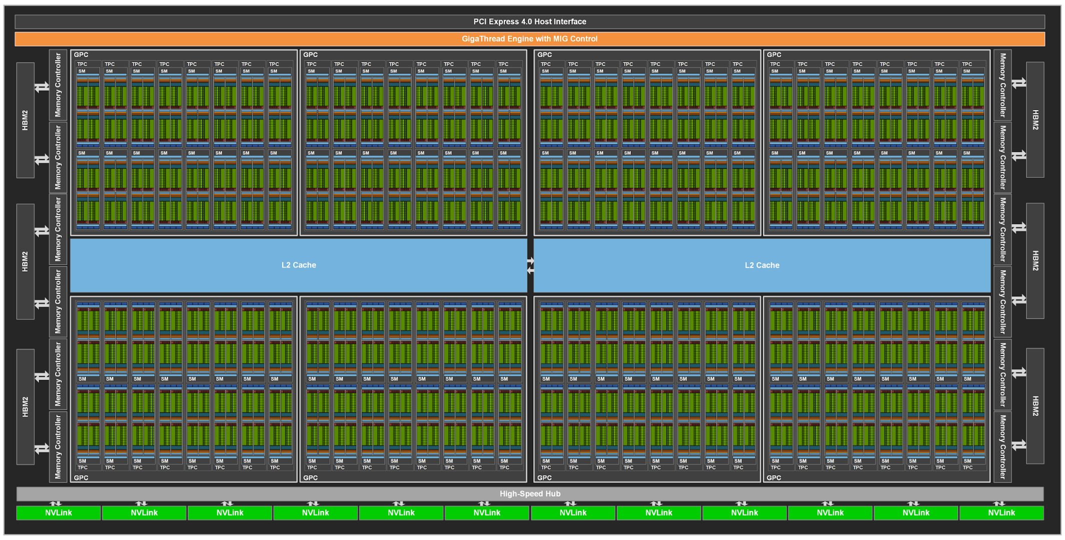

PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

Figure 15: Amere's Architecture

Key highlights that impacts our focus on the parallelization of that architecture[11]:

- It features 192 KB L1 caches per a streaming multiprocessor
- 40 MB Level 2 (L2) cache
- 40 GB of high-speed HBM2 for memory
- 555 GB/sec of memory bandwidth
- GPU-CPU NVLink yielding 600 GB/sec total bandwidth via 12 links
- 32 threads / warp
- 64 warps

- 2048 Max threads / SM
- 255 Max registers / thread
- 1024 Max thread block size

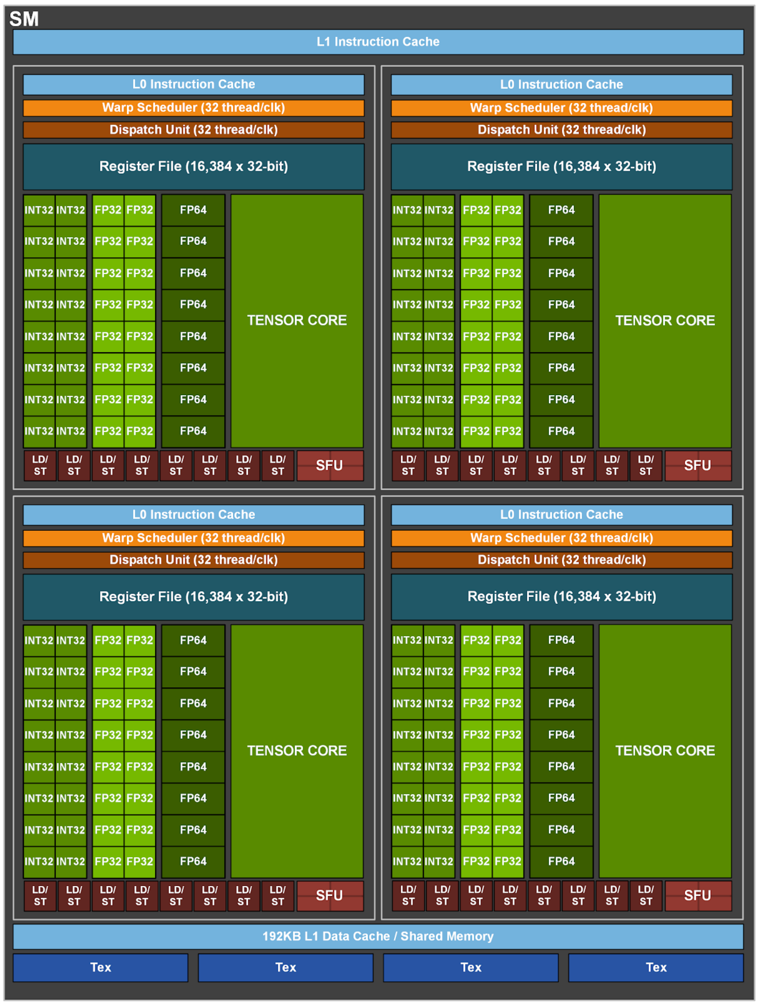

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Figure 16: Amere's SM Architecture

4.3 Task description

In our task, we should improve the current CUDA implementation to support faster data processing by using Unified Memory Architecture.

We received the SWE code without CUDA code injection. Indeed by even going back in history we were not successful in finding a building environment of CUDA.

That being said, searching for it, we found some useful snippets that helped us in the current code model.

So we listed our tasks as following:

- Setup a building CUDA H2D(Host-to-Device) code with correct outputs
- Implement UMA and compare
- Profile the code and tune the performance for the cluster
- Explore different approaches for optimization e.g. CUDA async Streams and Pinned Memory

4.4 Development Environment

As mentioned above we are developing our CUDA code on the SCCS GPU Cluster. This cluster features 4 3080 RTX GPUS(we used only a single GPU in our developlments and tests). They are all based on the A100 architecture explained above

The problem with SWE is that it has multiple dependencies and doesn't run out of the box on the cluster. Due to spack restrictions, we now pull the development environment from a handmade Docker Image that was uploaded to a member of the team. This is the same docker image that was on the main github repo of the SWE, but stripped of all the unnecessary dependencies to compress it's size.

Nevertheless, this image should build the code out of the box once downloaded. To run this image locally feel free to download and follow the instructions provided for docker setup on the SWE Readme.md.

To build and run the code on the cluster. You have to load the singularity that converts the docker image with all deps inside into running virtual image, then run the singularity command to pull the aforementioned docker image:

```
$ module load singularity-3.8.5
$ module load cuda-11.4.4
$ module load module load openmpi-4.0.2
$ module load cmake-3.23.0
$ module load googletest-1.11.0
$ singularity run --nv docker://asam11/swe:latest
```

Once Singularity started, user can build the code as explained in the README.md file:

```
$ mkdir build && cd build
$ cmake ..
$ ccmake ..
```

And once `ccmake` windows opens onscreen, we have to enable CUDA feature by setting the following form the list of options:

```
CMAKE_BUILD_TYPE Debug (to see cuda calls as they are wrapped for profiling)
CMAKE_CUDA_ARCHITECTURES 86
ENABLE_CUDA ON
ENABLE_CUDA_ALLOCATOR UMA/Pagelocked-PinnedMem/Device-Host
```

This should be enough to compile via `make -j` and run as already explained in the SIMD section.

4.5 Explanation of the code

We structured all of our CUDA code under folder Block. The files have `.cu`, `.cuh` extensions for sources and headers respectively. The kernels were separated into different files. We followed the advised approach by having coalesced memory for the GPU data transfer to have optimiuma alignment into different threads.

Then we separated our cuda code into different namespace `cuda::`. We stiched the `cuda::Block` class to the rest of the code base as in the SIMD via a global getter `getCudaBlockInstance`. From which it's treated as a normal Block class with same interface.

We managed to implement three different CUDA implementations, all wrapped inside dynamic macros switching located in HelperFunctions.cuh. According to the previous ccmake settings, the relative MACRO will be set, replacing the sub-code snippets to fit the respective scheme:

```
ENABLE_CUDA_UMA_ALLOCATOR: enabled UMA impolementation
ENABLE_CUDA_PINNEDMEM_ALLOCATOR: enabled the pinned memory implementation
NONE: if none of them defined,we use device-host normal memory copy
```

Example wrapper is the cuda malloc:

```
#ifdef ENABLE_CUDA_UMA_ALLOCATOR
#define CUDA_MALLOC(ptr, size) \
    cudaMallocManaged(ptr, size); \
    CUDA_CHECK_ERROR("Allocating UMA memory ", #ptr);
#elif defined(ENABLE_CUDA_PINNEDMEM_ALLOCATOR)
#define CUDA_MALLOC(ptr, size) \
    cudaMallocHost(ptr, size); \
    CUDA_CHECK_ERROR("Allocating Pinned memory ", #ptr);
#else
#define CUDA_MALLOC(ptr, size) \
    cudaMalloc(ptr, size); \
    CUDA_CHECK_ERROR("Allocating Device memory ", #ptr);
#endif
```

4.6 Tests

Following the Automated Integration and Unit tests task, we implemented our tests for CUDA to run on our local server explained in the Automated Tests CI section. The test file was integrated also via cmake and using gtest as a library instead of catch2 that came with the SWE. This file could be found here FWaveCudaSolverTest.cu.

The test body follows the same main loop inside MPI-Runner.cpp.

As a part of our verification process, we loaded both of CUDA scenarios and original scenarios to Paraview and compared the results. Here it is for the dam break scenario.

We also loaded the Google Test library to SWE's cmake files. And added the key `ENABLE_GTEST` ccmake option to be set in order to compile and run the tests for both cuda and simd if they were enabled via cmake.

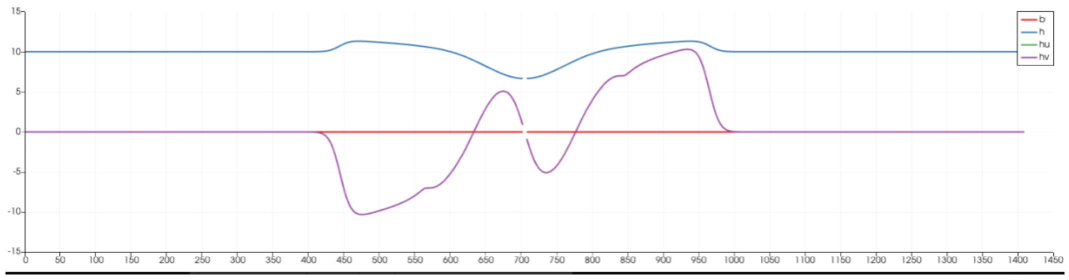


Figure 17: Original serial profile of h, b, hu, hv

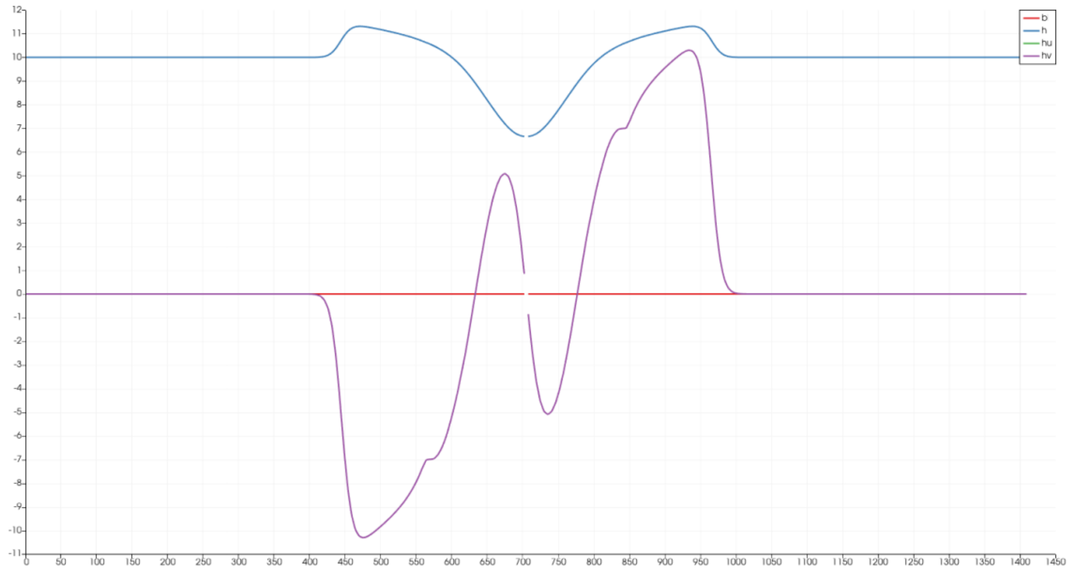


Figure 18: CUDA based profile of h, b, hu, hv

4.7 Results

Matrix size	sequential	UMA	device host	memory pinned
1024	36.36810548	2.238082781	5.506366408	49.49713703
824	19.19105382	1.142100154	3.310554291	23.23632318

Table 8: Results of CUDA implementation- runtime for Radial Dam Break scenario for different matrix sizes

Matrix size	UMA	device host	memory pinned
1024	16.24966949	6.60473764	0.7347516979
824	16.80330202	5.796930706	0.8259075103

Table 9: Speedup for different matrix sizes for Radial Dam Break Scenario

Scenario	seq.	UMA	device host	memory pinned
RadialDamBreak	181.181182688049	2.238082781	5.506366408	49.49713703
SeaatRest	181.1811827	7.70190362	21.21200556	247.2474672
SplashingCone	45.15861918	10.10783816	57.57297882	800.8005405
SplashingPool	30.05817508	4.495807456	15.15640679	15.15661698
BathymetryDamBreak	138.1381206	5.526320381	16.16117288	192.1924604

Table 10: Runtime for different scenarios for 1024x1024 case

Scenario	UMA	device host	memory pinned
RadialDamBreak	16.24966949	6.60473764	0.7347516979
SeaatRest	23.52420799	8.541445182	0.7327928765
SplashingCone	45.15861918	7.928302889	0.5699996331
SplashingPool	30.05817508	8.916082122	8.915958476
BathymetryDamBreak	24.99640106	8.547530652	0.7187489058

Table 11: Speedup for different scenario

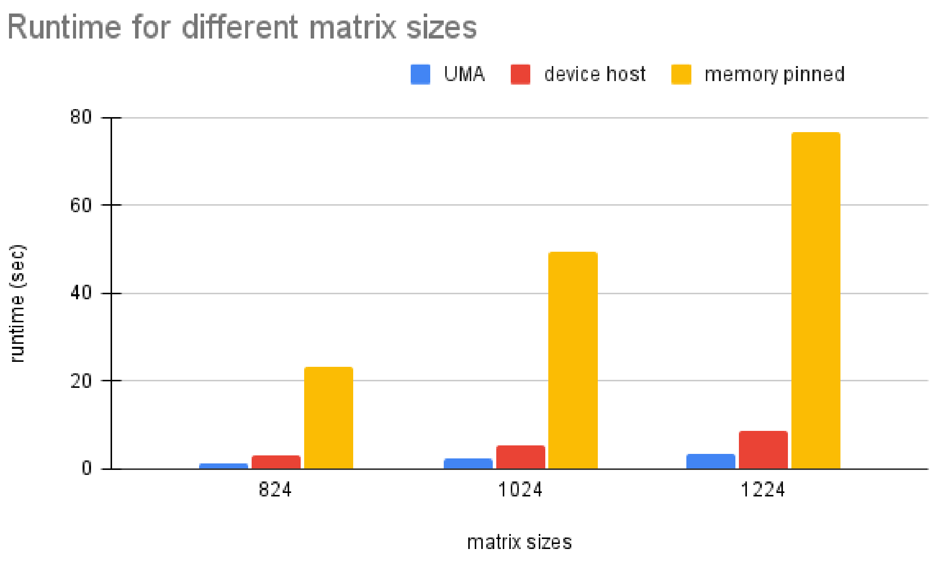


Figure 19: Runtime for different matrix sizes

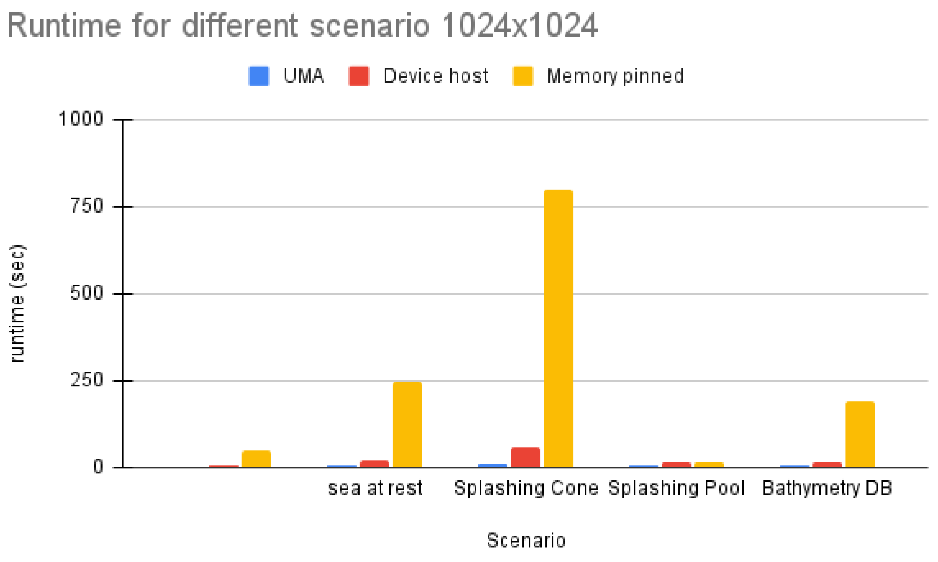


Figure 20: Runtime for different scenario 1024x1024

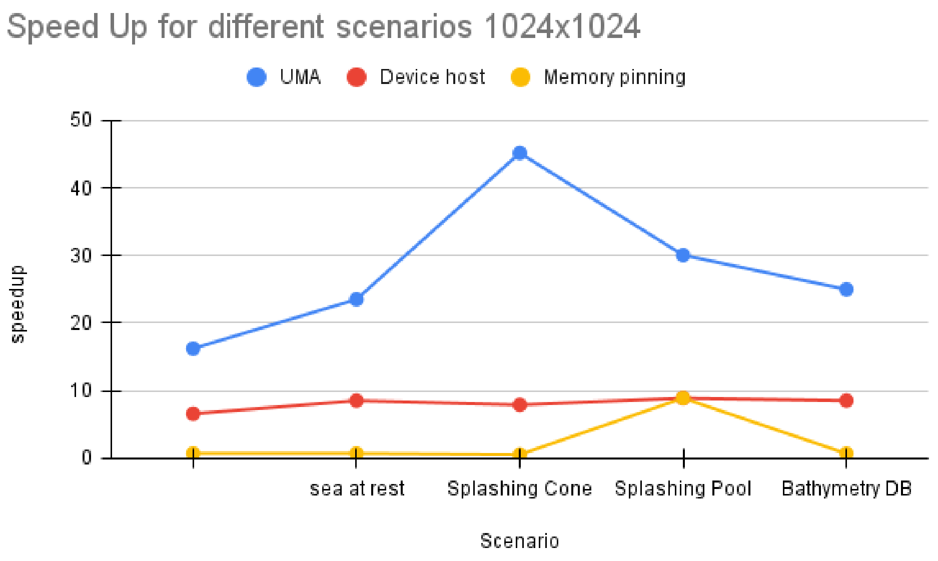
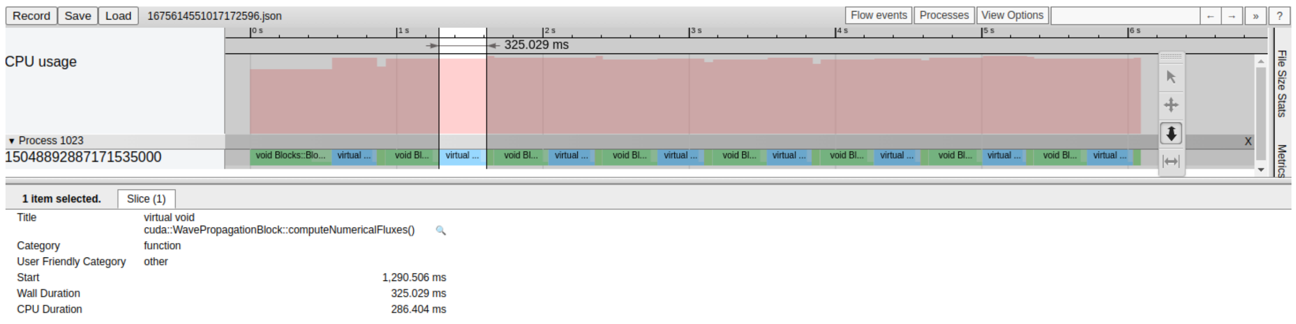


Figure 21: Speed Up for different scenarios 1024x1024

4.8 Profiling Performance Analysis

As we wanted to analyze on finer level in contrast to overall pipeline profiling, we wrote our own custom profiler using the chrome tracing tool. This allows us to control what functions to profile and what not in quick manner. This can be set via ccmake option `ENABLE_TRACING_PROFILER`.

Our profiler full implementation can be found in TraceViewerProfiler.hpp.



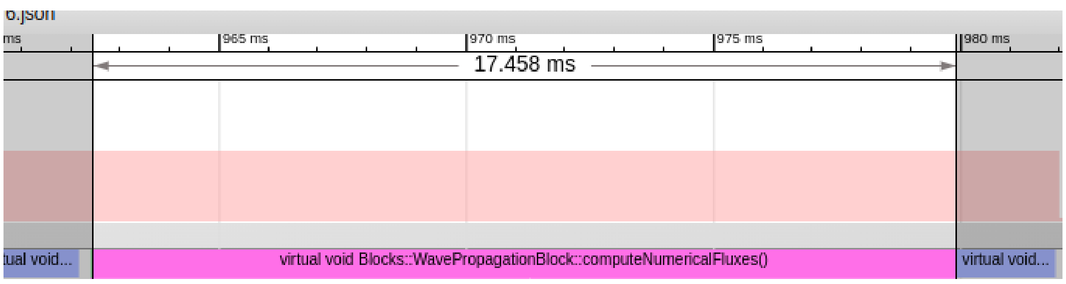
1 item selected. Slice (1)

Title	virtual void cuda::WavePropagationBlock::computeNumericalFluxes()
Category	function
User Friendly Category	other
Start	1,290.506 ms
Wall Duration	325.029 ms
CPU Duration	286.404 ms

Figure 22: Sample of oour Tracing profiler

We instantiate the profiling instance inside the desired to profile function, upon destruction of the object, we know how long it stayed alife. Then we use the google chrome browser to visually further investigate it. To collect such logs, build the program in Debug mode via `CMAKE_BUILD_TYPE : DEBUG`. Then you will see a `*.json` e.g. `build/1675361205013145258.json` created in the build folder. load it into any chromium based browser such as Brave or Chrome, etc by going to `brave://tracing` `chrome://tracing`.

Once loaded you can see the whole pipeline for each function where we instantiated the profiler instance `PROFILER_INSTANCE(0);`



6.json
ms
965 ms
970 ms
975 ms
980 ms
17.458 ms
ual void...
virtual void Blocks::WavePropagationBlock::computeNumericalFluxes()
virtual void...

Figure 23: Original serial computeNumericalFluxes duration

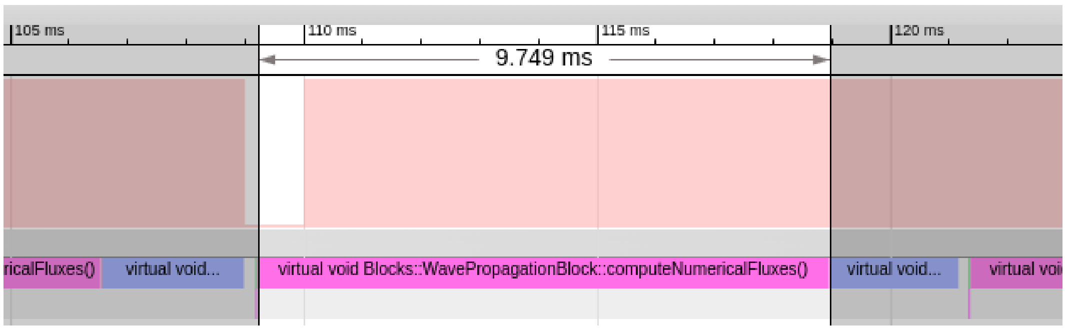
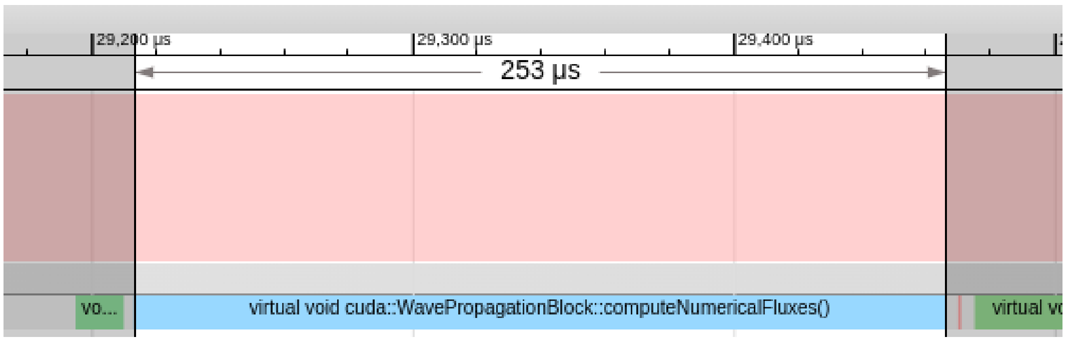
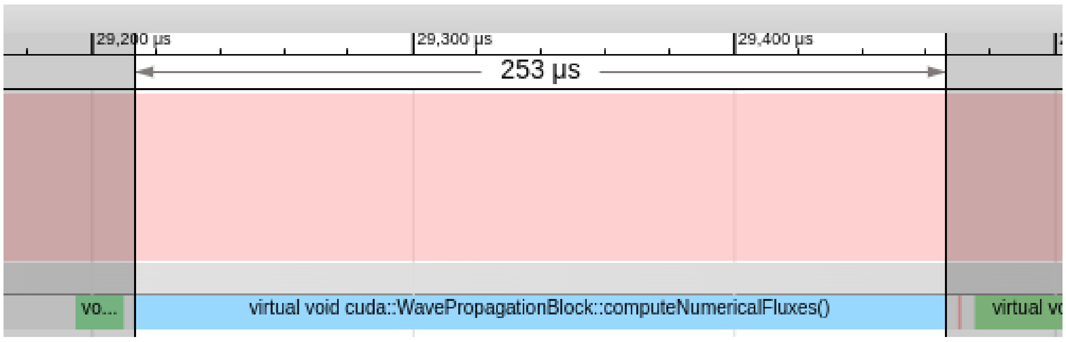


Figure 24: CUDA H2D computeNumericalFluxes duration



29,200 µs
29,300 µs
29,400 µs
253 µs
vo...
virtual void cuda::WavePropagationBlock::computeNumericalFluxes()
virtual v

Figure 25: CUDA UMA computeNumericalFluxes duration



29,200 µs
29,300 µs
29,400 µs
253 µs
vo...
virtual void cuda::WavePropagationBlock::computeNumericalFluxes()
virtual v

Figure 26: CUDA UMA computeNumericalFluxes duration

As we can see UMA has the best results, we actually expected that Pinned Memory implementation will be better according to the analysis and the results of the cuda-samples performance utility that reported the following:

```
GPU Device 0: "Ampere" with compute capability 8.6

Running .......................................................

Overall Time For matrixMultiplyPerf

"UMhint",   // Managed Memory With Hints
"UMhntAs",  // Managed Memory With_Hints Async
"UMeasy",   // Managed_Memory with No Hints
"0Copy",    // Zero Copy
"MemCopy",  // USE HOST PAGEABLE AND DEVICE_MEMORY
"CpAsync",  // USE HOST PAGEABLE AND DEVICE_MEMORY ASYNC
"CpHpglk",  // USE HOST PAGELOCKED AND DEVICE MEMORY
"CpPglAs"   // USE HOST PAGELOCKED AND DEVICE MEMORY ASYNC

Printing Average of 20 measurements in (ms)
Size_KB  UMhint UMhntAs  UMeasy   0Copy MemCopy CpAsync CpHpglk CpPglAs
4          0.213   0.222   0.333   0.016   0.032   0.026   0.032   0.026
16         0.234   0.253   0.467   0.028   0.043   0.046   0.054   0.044
64         0.323   0.357   0.834   0.113   0.119   0.093   0.088   0.078
256        0.566   0.592   1.163   0.469   0.282   0.267   0.248   0.235
1024       2.516   1.967   2.540   2.595   1.134   1.096   0.976   0.961
4096       6.533   5.672   8.640  13.487   4.846   4.806   4.298   4.283
16384     29.209  26.390  38.098  93.926  22.238  22.168  21.201  21.265
```

But we noticed it's much slower. We have not managed to analyze why this was happening.

We profiled the code using Nsight Compute for UMA, since it gave us best performance. Can be found in cuda folder on the repo.

Reviewing it, we found our bad occupancy grid. But as our code is using single thread/block by default in the first iteration, we couldnt rerun in different threads decomposition to improve the warp density. ATM, we only run single thread in the entire block which although gave almsot 16x speedup from the original code, we are yet still can improve it.

```
    Section: Occupancy
    ------------------------------- ----------- ------------
    Metric Name                     Metric Unit Metric Value
    ------------------------------- ----------- ------------
    Block Limit SM                        block           16
    Block Limit Registers                 block           84
    Block Limit Shared Mem                block           16
    Block Limit Warps                     block           48
    Theoretical Active Warps per SM        warp           16
    Theoretical Occupancy                     %        33.33
    Achieved Occupancy                        %         2.10
    Achieved Active Warps Per SM           warp         1.01
    ------------------------------- ----------- ------------

    WRN   This kernel's theoretical occupancy (33.3%) is limited by the number of
   blocks that can fit on the SM This
          kernel's theoretical occupancy (33.3%) is limited by the required amount of
   shared memory The difference
          between calculated theoretical (33.3%) and measured achieved occupancy
   (2.1%) can be the result of warp
          scheduling overheads or workload imbalances during the kernel execution.
   Load imbalances can occur between
          warps within a block as well as across blocks of the same kernel.
```

As can be seen, due to low occupancy, we didn't reach the theoreticial occupancy peak at 33%. This occupancy changes from a kernel to another, we have a list of all kernels in the same cuda results folder.

The impact of the occupancy per block can be tuned according to Nsight Compute calculator:

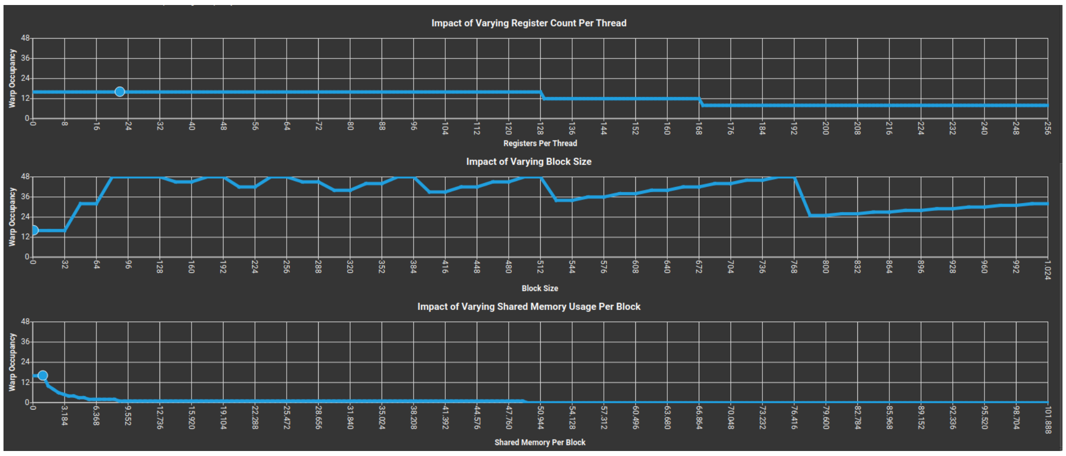


Figure 27: CUDA UMA computeNumericalFluxes duration

So according to the Nsight Compute occupancy calculator, we could utilize the best HW if we go in block size up to 96 for this example kernel. This is fine tuning task and due to time constraints we havn't managed to do it.

We implemented a non-working version of cuda streams, but as it was not passing our tests nor we noticed a significant improvement over UMA values, we could not include our analysis in it.

NOTE As we mentioned in the target imformation of Ampere Architecture, theoretically it's up to 600 GB/s. But the gpu is driven by the cpu, the actual top BW is defined by the PCI bus the CPU is installed on.

4.9 Conclusion

As we can see, CUDA has a significant impact on the performance, specially with coalesed memory setup. We also noticed that UMA has significant improvment IF done right, that includes eliminating any calls(a mistake from our side had a single call undetected make it worse that Host-Device memory approach). Scalability of that with MPI is expected, but as it doesn't change much weakly or strongly, we didn't include it in our analysis.

5 Automated Tests with Gitlab Continuous Integration (CI)

Automated tests are important to control if all commits from all the developers are compatible with each other and work without frequent conflicts or development bugs interleaving. Such issues can be minimized with Continuous Integration. In addition to that, the master branch became more stable often for any release(i.e. submission). Without last-minute conflict resolution which usually backfires with a buggy release.

To be able to test each change on the working tree, automated tests were added as part our SWE project. This was possible using Gitlab Runner.

In order to run the Gitlab Runner CI, we needed to setup an environment where `sudo` rights were granted to us as we can't install it in the user space. We tried to look up a spack module on the `CM2 cluster` but we couldn't find it. So to save time, we picked a laptop of a member of our team which has all development dependencies installed in order to test both the original code base, SIMD and CUDA unit tests. This means it needs at least a single GPU device with CUDA compatibility> 7.0.

The CI is currently active and has a nightly build at 12:00 pm daily on the master. The registered runners can be found here on the CI/CD.

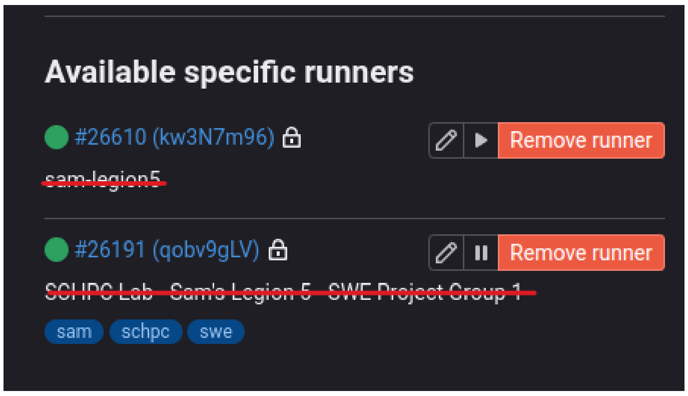


Figure 28: Currently registered local server Runners

5.1 CI Configuration

The current configuration is setup as follows:

- Once a branch that is connected to CI/CD via a Merge request pipeline has a new commit, GitLab's CI automatically builds the latest commit on any branch that is a part of a current open merge request. And runs our Unit/Integration tests.
- The current master status is added as a part of the homepage of our repo.
- Artifacts are kept for 2 days after which they expire and get deleted due to limited space

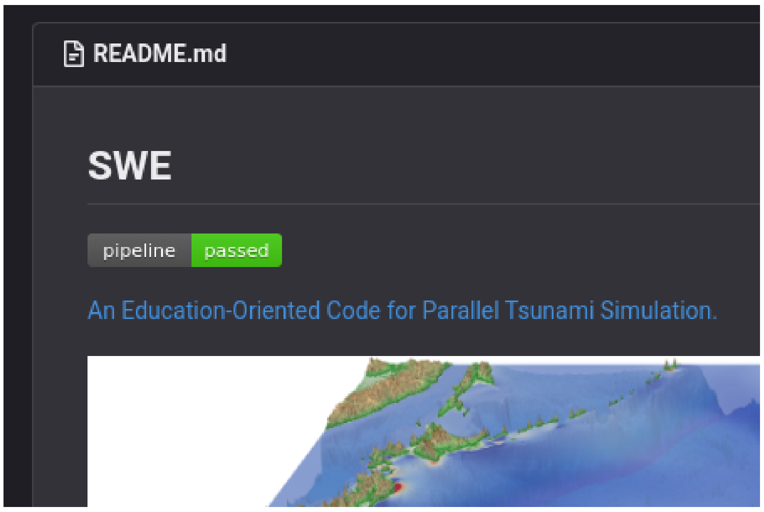
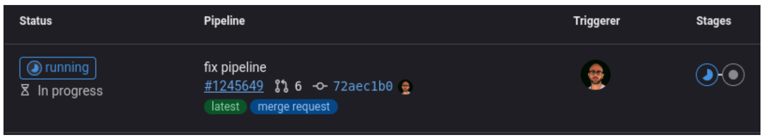


Figure 29: Current Status of the `master` branch



Status	Pipeline	Triggerer	Stages
running In progress	fix pipeline #1245649 6 72aec1b0 latest merge request		

Figure 30: A new commit is under build + test stages without manual trigger on the CI

Our CI consists of 2 interdependent stages `build` and `test`. The Test job will only trigger once the build has reported `PASSED` for the respective code base. As our code base mainly developed SIMD and CUDA. We have 2 main tests that verify the code[1]

Example of a CUDA build job can be found here and test job here.

Users also can trigger the build manually via the CI/CD page:

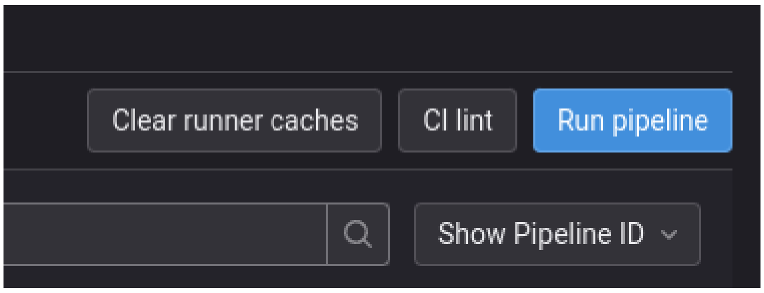


Figure 31: Run pipeline will trigger the jobs on chosen branch

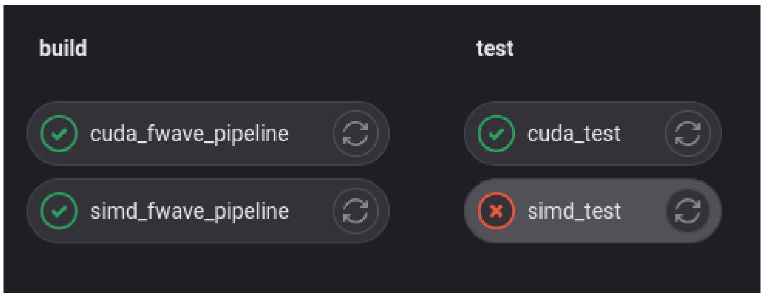
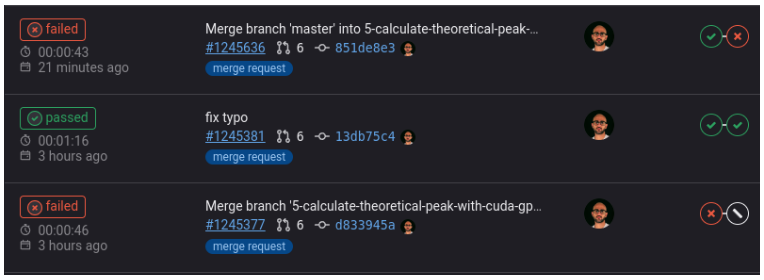


Figure 32: Our CI consists of 2 interdependent-stages `build` and `test`



failed 00:00:43 21 minutes ago	Merge branch 'master' into 5-calculate-theoretical-peak-... #1245636 6 851de8e3 merge request		✓ ✗
passed 00:01:16 3 hours ago	fix typo #1245381 6 13db75c4 merge request		✓ ✓
failed 00:00:46 3 hours ago	Merge branch '5-calculate-theoretical-peak-with-cuda-gp... #1245377 6 d833945a merge request		✗ ⊘

Figure 33: CI Pipeline Example of Failed test, Passed build + test and Failed build detection in CI

Configuration example can be found in the current master `SWE/.gitlab-ci.yml`.

Different Scenario can be selected using eg. `DUSED_SCENARIO=BathymetryDamBreakScenario`.

[1]While we wanted to focus on unit tests, we also wanted to test the final result. That's why we have an overall assertions.It's considered more of an integration. That's not practical but provides us with the information we need!

```yaml
 workflow:
    rules:
        - if: '$CI_PIPELINE_SOURCE == "merge_request_event"'
        - if: '$CI_PIPELINE_SOURCE == "web"'
        - if: '$CI_PIPELINE_SOURCE == "schedule"'
image:
    pull_policy: never
    name: swe
stages:
    - simd_fwave_pipeline
    - simd_test
    # - cuda_fwave_pipeline
    # - cuda_test
simd_fwave_pipeline:
    stage: simd_fwave_pipeline
    # instead of calling g++ directly you can also use some build toolkit like make
    # install the necessary build tools when needed
    script:
        # - echo "Setting up Intel OneAPI Toolkit Variables and NVCC.."
        # - source /opt/intel/oneapi/setvars.sh
        #-DUSED_SCENARIO=SplashingConeScenario crashing scenario
        - mkdir ${CI_PIPELINE_ID}_SIMD
        - cd ${CI_PIPELINE_ID}_SIMD
        - echo -e "=========== Start of job [${CI_PIPELINE_ID}_SIMD] - $(date) =========="
        - echo ------------------------- VECTORIZATION TEST BUILD -------------------------
        - cmake .. -DENABLE_VECTORIZATION=1 -DENABLE_VECTORIZATION_WITH_SIMD=1 -DUSED_SCENARIO=BathymetryDamBreakScenario

       # - cmake .. -DENABLE_VECTORIZATION=1 -DENABLE_VECTORIZATION_WITH_SIMD=1
        - make -j8
        - echo -e "=========== End of job [${CI_PIPELINE_ID}_SIMD] - $(date) =========="
    # NOTE fix artifacts collection for paraview
    # TODO deploy and avoid artifacts retention unnecessarily
    artifacts:
        paths:
            - ${CI_PIPELINE_ID}_SIMD
        expire_in: "2 days"
        name: ${CI_PIPELINE_ID}_SIMD

# run simd_tests using the binary built before
simd_test:
    needs:
        - simd_fwave_pipeline
    stage: simd_test
    dependencies:
        - "simd_fwave_pipeline"
    script:
        - cd ${CI_PIPELINE_ID}_SIMD
        - ./FWaveSIMDSolverTest
        - ctest
```

References

[1] "Cuda swe." https://www5.in.tum.de/lehre/vorlesungen/hpc/WS12/uebung/cuda_swe.pdf. (Accessed on 11/13/2022).

[2] M. Bader, A. Breuer, W. Hölzl, and S. Rettenberger, “Vectorization of an augmented riemann solver for the shallow water equations,” in *2014 International Conference on High Performance Computing Simulation (HPCS)*, pp. 193–201, 2014.

[3] “Intel® xeon® processor e5-2697 v3.” https://ark.intel.com/content/www/us/en/ark/products/81059/intel-xeon-processor-e52697-v3-35m-cache-2-60-ghz.html//. (Accessed on 11/26/2022).

[4] “Intrinsics guide.” https://en.algorithmica.org/hpc/simd/intrinsics/. (Accessed on 11/26/2022).

[5] “Intel intrinsics guide.” https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. (Accessed on 10/30/2022).

[6] “Gcc optimize-options.” https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html---. (Accessed on 25/01/2022).

[7] “Paraview guide.” https://docs.paraview.org/en/latest/UsersGuide/index.html. (Accessed on 12/01/2023).

[8] “Intel advisor guide.” https://www.intel.com/content/dam/develop/external/us/en/documents/advisor-user-guide.pdf. (Accessed on 11/13/2022).

[9] “Vectorization of riemann solvers for the single- and multi-layer shallow water equations.” https://mediatum.ub.tum.de/doc/1506348/1506348.pdf. (Accessed on 18/12/2022).

[10] “Wikipedia: A100 gpu hardware architecture.” https://en.wikipedia.org/wiki/Ampere_(microarchitecture). (Accessed on 11/13/2022).

[11] “A100 gpu hardware architecture.” https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth. (Accessed on 11/13/2022).