
Research Project 2023

Autumn

# Vectorization and Communication Based Optimizations for Walberla

*Research Project Final Report*

Shuxin
**Zheng**

Academic Year 2022/2023

# Contents

# Overview

## 0.1 Introduction to WalBerla

Walberla is a modern open-source software framework that supports complex multiphysics simulations, and that is specifically designed to address the performance challenge in CSE: exploiting the full power of the largest supercomputers for a wide class of scientific research questions. Walberla's main focus is computational fluid dynamics simulations with the lattice Boltzmann method (LBM). It, therefore, offers a wide range of state-of-the-art LBM models, together with a variety of utility and usability functionality.

## 0.2 Block Forest

The core data structure of the Walberla is the BlockForest[1]. It is responsible for domain partitioning and acts as a container for all data needed by the simulation. The starting point for the domain partitioning is a cuboidal simulation domain. This domain is partitioned regularly into equally sized subdomains. These intermediate subdomains, subsequently, act as root nodes for an octree, effectively forming a forest of octrees. These individual octrees can be refined independently of each other. The only restriction on the octree structure is a 2:1 size ratio, as illustrated in Figure1, which is maintained between neighboring subdomains at all times.

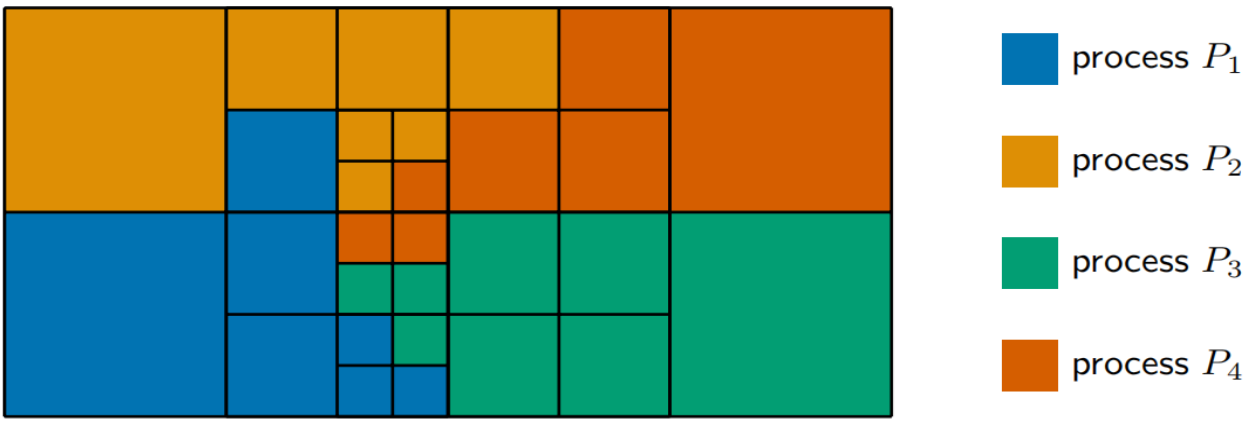


Figure 1: block forest

## 0.3 Code Generation

HPC software usually has to be modified extensively in order to make full use of new hardware architectures. WalBerla employs code generation techniques to generate time-critical numerical kernels from a high-level, domain-specific formulation. WalBerla used the pystencils metaprogramming project to generate highly efficient stencil codes for CPUs and GPUs based on a common high-level, symbolic description. As well as lbmpy, a code generation package that supports a wide variety of different lattice Boltzmann methods.[2] Backends then generate C/C++, CUDA, OepnCL code or LLVM IR from this representation. Kernel code generation has already been applied successfully to WalBerla-based LBM simulations and phase-field

simulations of alloy solidification on CPUs and GPUs.

# 1 Vectorization-Based GPU Optimizations

## 1.1 Test Case S2A

Walberla contains lots of different test cases. We use the test case S2A as our target. The S2A test case first reads the geometry of a car from a .obj file(like fig1.1). From this obj file, the size of the domain is determined. The size of the domain is then adapted to fit the number of cells per block. This means it is extended in all directions such that a natural number of blocks consisting of the given number of cells per block would fit the direction. In this way, the number of blocks for the block forest is calculated and distributed to the processes. Technically, there is no need to compute the block inside the domain of the car.

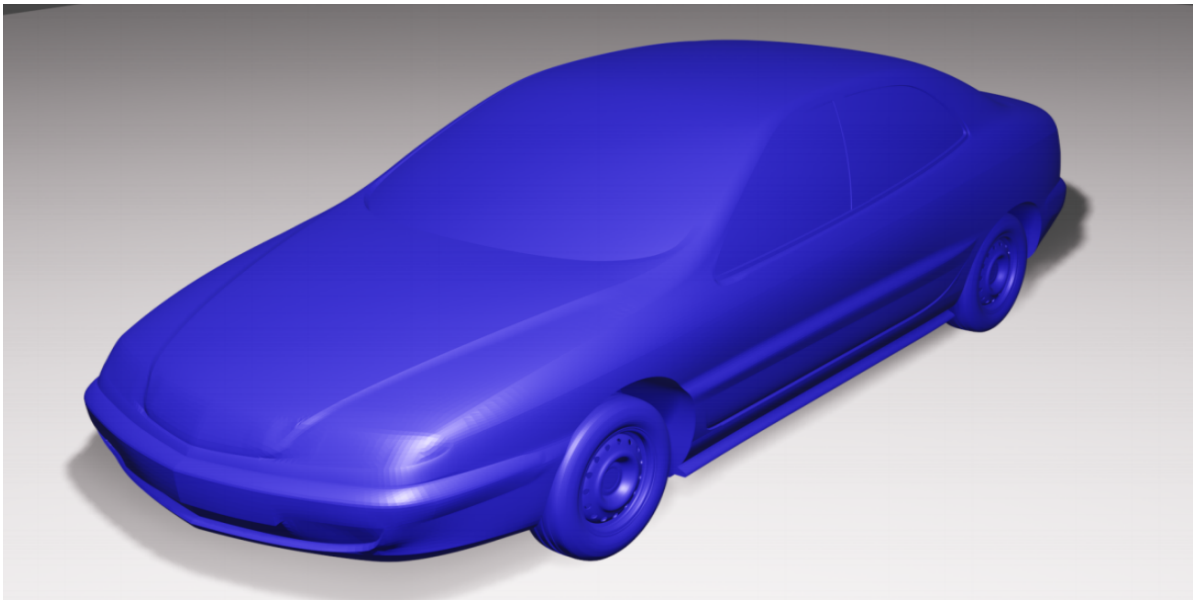

Figure 1.1: car for test case S2A

## 1.2 FlagField

### Basic Idea

It's not very easy, especially in a large software like Walberla, to ignore the block inside the domain of the car due to the way the mesh is implemented. So we need to perform all computations both inside and outside the domain of the car even though the inside part is not needed.
So we come up with the idea called 'FlagField'. In this idea, we hope to reduce the cost of computing kernels by not computing the domain inside the car. To achieve this goal, we added an 'if' condition in order to check whether the target block is inside the boundary. If yes, just do nothing with it, or else we finish the origin calculation process. By doing this we reduced the computation requirement.

### Result & Analysis

We check the performance of this modification using NVIDIA Nsight Compute to check whether the modification has achieved the desired effect. Something to mention here is that the data we use is the throughput which reports the achieved percentage of utilization with respect to the theoretical maximum. The result is partly shown below in Table1.1. As we can obviously find, memory usage is decreased by around 9.1%.

Besides the GPU throughput, we can also check the detailed memory chart. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem

| Test Case | Master branch | FlagField branch |
|---|---|---|
| Compute (SM) Throughput [%] | 4.33 | 4.68 |
| Memory Throughput [%] | 43.37 | 39.41 |
| L1/TEX Cache Throughput [%] | 48.39 | 45.32 |
| L2 Cache Throughput [%] | 54.25 | 49.57 |
| DRAM Throughput [%] | 9.73 | 5.71 |
| Duration [usecond] | 19.49 | 13.54 |
| Elapsed Cycles [cycle] | 17,788 | 11,783 |
| SM Active Cycles [cycle] | 10,765.11 | 6,717.06 |
| SM Frequency [cycle/usecond] | 909.65 | 868.95 |
| DRAM Frequency [cycle/nsecond] | 1.01 | 966.43 |

Table 1.1: GPU throughput for Master and FlagField branch

Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). The Memory Chart shows the utilization of the involved hardware units and the communication between units. The result is partly shown in Table1.2, Table1.3, Table1.4, and Table 1.5. In this analysis, we can find that the memory transfer between the local and the L1 Cache is reduced by almost half. Also, the utilization of communication from local to the kernel and from L1 to L2 Cache is all hugely decreased, also the amount of data transfer inside the L2 Cache is decreased. So the optimization here using the 'if' condition is effective. We then need to check the final achievement.

| Test Case | Wavefronts(Master) | Peak(Master) | Wavefronts(FlagField) | Peak (FlagField) |
|---|---|---|---|---|
| Total | 1,024 | 0.05 | 1,024 | 0.08 |

Table 1.2: Shared Memory for Master and FlagField branch

| Test Case | Reqs (Master) | Hit Rate (Master) | Reqs (FlagField) | Hit Rate (FlagField) |
|---|---|---|---|---|
| Local Load | 7,154 | 91.53 | 4,088 | 98.95 |
| Global Load | 5,110 | 66.51 | 5,110 | 66.66 |
| Global Store | 1,022 | 21.98 | 1,022 | 21.34 |
| Local Store | 136,192 | - | 77,824 | - |
| Loads | 12,264 | 73.45 | 9,198 | 72.48 |
| Stores | 137,214 | 0.39 | 78,846 | 0.67 |
| Total | 149,478 | 11.89 | 88,044 | 16.52 |

Table 1.3: L1/TEX Cache for Master and FlagField branch

By checking the MLUPS table, we get the final result for this step. The comparison table is shown in Table1.6. We do the test both with and without the NVIDIA analysis tool and the FlagField performance is around 10% better than the master one in both cases.

**Conclusion**

By ignoring the blocks inside the boundary, we achieved a 10% improvement in performance. Specifically, the optimization is for the memory transfer part since in this case, the part of the message in these ignored blocks has no need to be transferred around. However, we can easily obtain that this process is still memory bound.

| Test Case | Reqs (Master) | Hit Rate (Master) | Reqs (FlagField) | Hit Rate (FlagField) |
|---|---|---|---|---|
| L1/TEX Load | 8,322 | 10.09 | 7,542 | 3.05 |
| L1/TEX Store | 139,803 | 100 | 81,606 | 100 |
| L1/TEX Total | 148,110 | 95.49 | 89,143 | 92.55 |
| L2 Fabric Total | 154,199 | 97.82 | 95,579 | 96.34 |
| GPU Total | 311,655 | 96.69 | 193,215 | 94.54 |

Table 1.4: L2 Cache for Master and FlagField branch

| Test Case | Throughput for Master | Throughput for FlagField |
|---|---|---|
| Load | 43,178,981,937.60 | 61,144,208,037.83 |
| Store | 83,106,732,348.11 | 9,513,002,364.07 |
| Total | 126,285,714,285.71 | 70,657,210,401.89 |

Table 1.5: Device Memory for Master and FlagField branch

| **Master without profile** | **FlagField without profile** |
|---|---|
| 739.293 MLUPS | 890.257 MLUPS |
| (millian lattice cell updates per second) | (million lattice cell updates per second) |
| 739.293 MLUPS / process | 890.257 MLUPS / process |
| 734.668 MFLUPS | 884.688 MFLUPS |
| (millian fluid lattice cell updates per second) | (million fluid lattice cell updates per second) |
| 734.668 MFLUPS | 884.688 MFLUPS |
| 11280.7 time steps / second | 13584.2 time steps /second |
| **Master with profile** | **FlagField with profile** |
| 633.677 MLUPS | 707.113 MLUPS |
| (millian lattice cell updates per second) | (million lattice cell updates per second) |
| 633.677 MLUPS / process | 707.133 MLUPS /process |
| 629.713 MFLUPS | 702.689 MFLUPS |
| (millian fluid lattice cell updates per second) | (million fluid lattice cell updates per second) |
| 629.713 MFLUPS / process | 702.689 MFLUPS / process |
| 9669.15 time steps / second | 10789.7 time steps / second |

Table 1.6: MLUPS table for Master and FlagField branch

## 1.3 SIMD instructions

### 1.3.1 Basic Idea

By developing the 'FlagField', we reduced the cost of computing kernels and the memory transfer by ignoring the block inside the car. However, putting an 'if' condition in a GPU-based 'for' loop is not always a good thing due to the parallel nature of GPU threads. The motivation for using SIMD instructions here is to reduce the penalty of the 'if' condition, and also potentially increase the throughput. Since the performance of the SIMD instructions is not as clear, we first made a toy test case and finally instigate it into Walberla.

### 1.3.2 Toy Case

To test if the SIMD does work for our cases, we first have a careful observation of the origin code. The 'if' condition is inside a kernel function which will be called several times inside a lot of calculation functions.

```
1 //origin if condition in FlagField
2 if (((((int64_t)
3 (_data_flag_field_10_20[_stride_flag_field_0*ctr_0])))) == (8))
```

By developing the SIMD, we are aiming to modify the origin by comparing 'if' to use the function '__vsetne4()' in CUDA MATH API. This function performs a per-byte (un)signed comparison. It splits 4 bytes of each argument into 4 parts, each consisting of 1 byte, and returns 1 if a != b, else return 0.
To verify the performance, I developed two versions of the testing case:
The first one fills three arrays with random unsigned int value, then call a kernel function, which does some calculation (for enlarging the GPU time) under the 'if' condition (using the Logistic Map[5] here, especially $4*x*(x-1)$, where x is the random filled unsigned int value). The 'if' command compares the two randomly filled arrays.

```
1  __global__ void compare(const unsigned int* a, const unsigned int* b,
2  double *rd, unsigned int *c)
3  {
4      // this thread handles the data at its thread id
5      int tid = threadIdx.x;
6      double x = rd[tid];
7      // Non-SIMD Version
8      // if(a[tid] != b[tid])
9      // SIMD Version
10     if(__vsetne4(a[tid], b[tid])) {
11         for(std::size_t j =0; j < 1e5 ; ++j)
12             x = 4 * x * (x-1);
13     } else {
14         for(std::size_t j =0; j < 1e5 ; ++j)
15             x = 3.8 * x * (x-1);
16     }
17     rd[tid] = x;
18 }
```

The second one also fills three arrays with random unsigned int value, then call a kernel function, which has the 'for' loop outside the 'if' condition (still using the Logistic Map). The 'if' command compares the two randomly filled arrays.

```
1  __global__ void compare(const unsigned int* a, const unsigned int* b,
2  double *rd, unsigned int *c)
3  {
4      // this thread handles the data at its thread id
5      int tid = threadIdx.x;
6      double x = rd[tid];
7      for(std::size_t i =0; i < 1e6 ; ++i) {
```

```
8           // Non-SIMD Version
9           // if(a[tid] != b[tid])
10          // SIMD Version
11          if(__vsetne4(a[tid], b[tid])) {
12              x = 4 * x * (x-1);
13          } else {
14              x = 3.8 * x * (x-1);
15          }
16      }
17      rd[tid] = x;
18  }
```

### 1.3.3 Toy case result & analysis

By observing using Nvidia Nsight System, we check the detailed GPU kernel running time of the two test cases. In the first case, hardly any optimization can be observed even when increasing the iteration number to 1e6. The result is partly shown in Table 1.7.

| Test Case | Begin | End | Duration |
|---|---|---|---|
| outside for loop with SIMD instruction | 1.25114s | 21.9697s | 20.719s |
| outside for loop without SIMD instruction | 0.586796s | 21.2936s | 20.707s |
| speedup | | | not observed |

Table 1.7: running time comparing for outside 'for' loop cases

In the second, we obtained a 1.5x times speed up by put the data array with size 1024*1024, and the for loop with iteration time 1e5. The result is shown in Table 1.8.

| Test Case | Begin | End | Duration |
|---|---|---|---|
| inside for loop with SIMD instruction | 0.556659s | 2.66065s | 2.104s |
| inside for loop without SIMD instruction | 0.603715s | 4.18042s | 3.577s |
| speedup | | | 41.18% |

Table 1.8: running time comparing for inside 'for' loop cases

By this result, we assume that this SIMD function can give optimizations but only if the 'if' condition is called enough many times. To make sure the optimization, we make a more specific look inside the running case using NVIDIA Nsight Compute (only for the second case). The GPU throughput of this comparing case is shown in Table 1.9.

| Test Case | None SIMD | SIMD |
|---|---|---|
| Compute (SM) Throughput [%] | 58.53 | 99.96 |
| Memory Throughput [%] | 58.77 | 0.00 |
| L1/TEX Cache Throughput [%] | 58.76 | 0.00 |
| L2 Cache Throughput [%] | 19.19 | 0.00 |
| DRAM Throughput [%] | 0.00 | 0.00 |
| Duration [usecond] | 4.54 | 2.66 |
| Elapsed Cycles [cycle] | 4,976,815,358 | 2,913,894,674 |
| SM Active Cycles [cycle] | 4,978,100,584.19 | 2,913,591,974.92 |
| SM Frequency [cycle/usecond] | 1.10 | 1.09 |
| DRAM Frequency [cycle/nsecond] | 1.22 | 1.21 |

Table 1.9: GPU throughput for inside 'for' loop cases

It's easy to find that the SIMD command gives advantages in reducing the memory bottleneck so as to make the GPU able to compute in full power. This can also be found in the memory chart as shown in Table 1.10, Table 1.11, and Table 1.12.

| Test Case | Reqs (No-SIMD) | Hit Rate (No-SIMD) | Reqs (SIMD) | Hit Rate (SIMD) |
|---|---|---|---|---|
| Local Load | 0 | 0 | 0 | 0 |
| Global Load | 209,716,248,576 | 100 | 3,145,728 | 13.17 |
| Global Store | 104,857,600,000 | 100 | 1,048,576 | 99.66 |
| Local Store | 0 | 0 | 0 | 0 |
| Loads | 209,716,248,576 | 100 | 3,145,728 | 13.17 |
| Stores | 104,857,600,000 | 100 | 1,048,576 | 99.66 |
| Total | 314,573,848,576 | 100 | 4,194,304 | 34.79 |

Table 1.10: L1/TEX Cache for inside 'for' loop cases

| Test Case | Reqs (No-SIMD) | Hit Rate (No-SIMD) | Reqs (SIMD) | Hit Rate (SIMD) |
|---|---|---|---|---|
| L1/TEX Load | 1,503,110 | 51.77 | 2,759,811 | 68.08 |
| L1/TEX Store | 29,174,180,876 | 100 | 1,045,696 | 100 |
| L1/TEX Total | 29,183,425,893 | 100 | 3,876,945 | 76.07 |
| L2 Fabric Total | 19,042,690,503 | 100 | 1,537,094 | 99.42 |
| GPU Total | 776,332,448 | 100 | 5,845,188 | 82.91 |

Table 1.11: L2 Cache for inside 'for' loop cases

| Test Case | Throughput for No-SIMD | Throughput for SIMD |
|---|---|---|
| Load | 43,178,981,937.60 | 566,648 |
| Store | 83,106,732,348.11 | 0 |
| Total | 126,285,714,285.71 | 566,648 |

Table 1.12: Device Memory for inside 'for' loop cases

In these tables, it's easy to find that the utilization of communication between kernel and global, global and L1/TEX Cache, L1, and L2 Cache is all hugely decreased, also the amount of data transfer inside the L2 Cache is decreased benefited by SIMD command. This is because the SIMD command load all the data into the core and stores back once since our data is reusable, while the normal instructions will load and store any time data is needed. So there should be a huge improvement in adapting the SIMD command if the data in the code have this reuse feature.

### 1.3.4 S2A case result & analysis

With the toy case result, we assume that this SIMD instruction helped solve the memory bottleneck in a specific situation, e.g the SIMD allows the same core to reuse the same data, which then makes it so that the data doesn't have to be transferred out of the core registers, or in a word, data reuse. However, this characteristic doesn't exist in the data movement of S2A, so it is not clear if the same optimizations will work for the S2A test case. In this case, we still make a test. By modifying the origin 'if' condition to the following one, we integrate the SIMD instruction. Since the code is generated by the backend every time run the CMake automatically, we need to first modify the CMakeList and make sure the executable file is compiled with our modified code but not the generated one.

```
1 //SIMD if condition
2 if (!__vsetne4(_data_flag_field_10_20[_stride_flag_field_0*ctr_0],
3                 (unsigned int)8))
```

Unfortunately, our S2A did not benefit from this SIMD instruction, as shown both in the MLUPS table and nuc. The table 1.13 is shown below. The difference is within the fluctuation range of the machine. Even in ncu, no noticeable difference can be found.

| **FlagField without SIMD** | **FlagField with SIMD** |
|---|---|
| 707.113 MLUPS | 704.314 MLUPS |
| (millian lattice cell updates per second) | (million lattice cell updates per second) |
| 707.113 MLUPS / process | 704.314 MLUPS / process |
| 702.689 MFLUPS | 699.908 MFLUPS |
| (millian fluid lattice cell updates per second) | (million fluid lattice cell updates per second) |
| 702.689 MFLUPS | 699.908 MFLUPS |
| 10789.7 time steps / second | 10747 time steps /second |

Table 1.13: MLUPS table for FlagField with/without SIMD

### 1.3.5 Conclusion

Out of our toy case testing, the SIMD instruction helps enlarge the throughput by enabling the data reuse and so reducing the memory transfer, and it will work in special cases (e.g called enough large time, and the same core will reuse the same data). Unfortunately, our S2A case does not benefit from this instruction, this is most probably because this case is also limited by other bottlenecks but not the memory. But this work is still valuable since there be more other cases that may fit this special case.

# 2 Communication-Bases Optimizations for Walberla

## 2.1 Optimization by overlap

The overlap mode is to hide the communication cost by using non-block communication while doing the computation when waiting. The working flow is shown in Figure2.2.

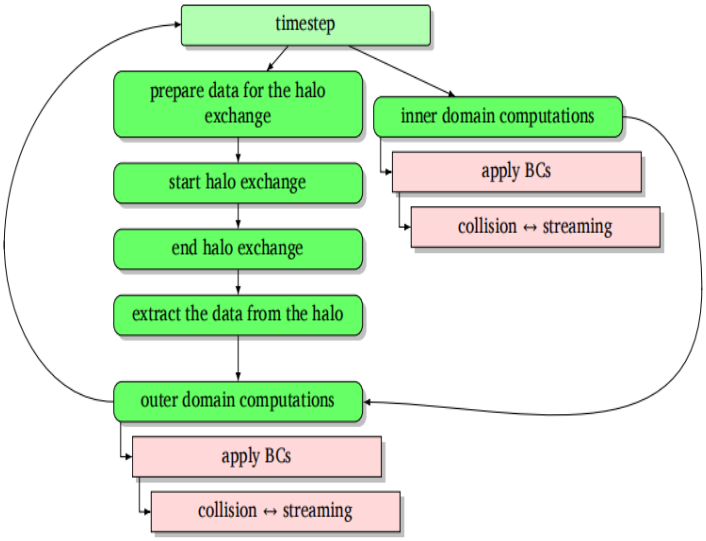


Figure 2.1: Algorithmic steps of waLBerla's LBM

Here I list the performance the target benchmark achieves within Table2.1. The highlight ones are the performance varying over 5%, after the discussion and once more verification, we consider these performance

floating is due to networking and seems reasonable. (The performance evaluated here uses the unit MLUPS which refers to 'in mega lattice updates per second)

Apart from the performance verification, this test also shows the speedup by using overlap. Scenario 1 is the 'noOverlap' case while the others are tests that have varying "innerOuterSplit" which defines the split rate. The speedup rate for using non-block overlap communication is also listed together with the value of the "innerOuterSplit" Scenario in Tabele2.2. By observing, we found that for a larger number of nodes, a reasonably larger block will gain better performance while a small block may lead to a communication bottleneck. The best speedup achieved shows the potential that a larger number of nodes prefers a larger inner block with an outer block which will not lead to a communication block.

| #node | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|
| Scenario 1 | 1622.65 | 1640.08 | 1584.63 | 1523.11 | 1538.47 |
| Scenario 2 | 2055.47 | 1942.84 | 1751.00 | 1976.95 | 1423.74 |
| Scenario 3 | 2076.59 | 2023.41 | 1753.80 | 1940.94 | 1511.33 |
| Scenario 4 | 2093.22 | 2057.04 | 1963.84 | 1999.01 | 1557.90 |
| Scenario 5 | 2215.06 | 2209.13 | 2054.30 | 1864.07 | 1812.53 |
| Scenario 6 | 2081.36 | 2084.79 | 2049.62 | 1969.29 | 1537.09 |
| Scenario 7 | 2098.47 | 2020.72 | 2100.50 | 1954.67 | 1595.74 |
| Scenario 8 | 2136.46 | 2103.8 | 2107.41 | 1923.53 | 1528.38 |
| Scenario 9 | 2134.75 | 2153.91 | 2130.23 | 2112.86 | 2149.21 |
| Scenario 10 | 1900.3 | 1917.86 | 1915.79 | 1896.77 | 1900.68 |
| Scenario 11 | 2093.3 | 2034.34 | 1976.95 | 1961.23 | 1612.45 |
| Scenario 12 | 2117.57 | 2127.64 | 2144.97 | 2063.41 | 1873.09 |
| Scenario 13 | 2022.03 | 2041.15 | 2049.11 | 2019.97 | 2032.47 |
| Scenario 14 | 1821.74 | 1840.86 | 1797.26 | 1721.82 | 1699.35 |

Table 2.1: MLUPS for origin data structure

| #node | 8 | 16 | 32 | 64 | 128 | innerOuterSplit |
|---|---|---|---|---|---|---|
| Scenario 1 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | (1, 1, 1) |
| Scenario 2 | 1.27 | 1.18 | 1.10 | 1.30 | 0.93 | (1, 1, 1) |
| Scenario 3 | 1.28 | 1.23 | 1.11 | 1.27 | 0.98 | (4, 1, 1) |
| Scenario 4 | 1.29 | 1.25 | 1.24 | 1.31 | 1.01 | (8, 1, 1) |
| Scenario 5 | **1.37** | **1.35** | 1.30 | 1.22 | 1.18 | (16, 1, 1) |
| Scenario 6 | 1.28 | 1.27 | 1.29 | 1.29 | 1.0 | (32, 1, 1) |
| Scenario 7 | 1.29 | 1.23 | 1.33 | 1.28 | 1.04 | (4, 4, 1) |
| Scenario 8 | 1.32 | 1.28 | 1.33 | 1.26 | 0.99 | (8, 8, 1) |
| Scenario 9 | 1.32 | 1.31 | 1.34 | 1.33 | **1.40** | (16, 16, 1) |
| Scenario 10 | 1.17 | 1.17 | 1.21 | 1.25 | 1.24 | (32, 32, 1) |
| Scenario 11 | 1.29 | 1.24 | 1.25 | 1.29 | 1.05 | (4, 4, 4) |
| Scenario 12 | 1.31 | 1.30 | **1.35** | **1.35** | 1.22 | (8, 8, 8) |
| Scenario 13 | 1.25 | 1.24 | 1.29 | 1.33 | 1.32 | (16, 16, 16) |
| Scenario 14 | 1.22 | 1.12 | 1.13 | 1.13 | 1.10 | (32, 32, 32) |

Table 2.2: Speedup for overlap communication & split ratio

## 2.2 Idea about data compression

Research has found that significant speedup is achieved by decoupling arithmetic precision and memory precision[3], e.g, storing with the origin data precise while translating on lower precise(for example, the data transported in 16-bit while stored as FP32). With this result, we come up with the idea to compress the data before communication and decompress it after to achieve the purpose to reduce the communication cost. To again make this idea convincing, we start with a tiny test: By building a sender and a receiver using MPI,

we randomly generate some data in a double format and send/receive them by 'Isend' and 'Irecv'(keep the same as our project) both using the format 'MPI_FLOAT' and 'MPI_DOUBLE', then counting the waiting time. The result is really obvious in the table2.3. However, it's not safe to just convert the data type during communication in Walberla since we can not guarantee the effect in the result accuracy. So we turn for some help from other research and finally decide to use the ZFPlib which helps to compress the data from double to float size with a specially designed format to reduce the load in communication, and then decompress it back to double after communication.

| Data Length | wait time(FLOAT) | wait time(DOUBLE) |
|---|---|---|
| 100000 | 0.000135 | 0.000234 |
| 1000000 | 0.001133 | 0.001861 |
| 10000000 | 0.010961 | 0.019268 |
| 100000000 | 0.109485 | 0.189590 |

Table 2.3: Isend waiting time(s) for sending different data types

### 2.2.1 Mini Application

To further confirm whether this optimization will give benefits to our software, we also build a mini-application that simulates the kernel function of Welberla which not only contains the communication but also the calculation for physics formulation. The mini-kernel is to simulate the wave propagation. To make it easier to understand, I put figures to show the initial and final status. By controlling the 'ncell'(which defined the resolution, which means the whole domain is divided into ncell parts) and the 'nhalo'(which defined the sampling rate or say, data length for communication. nhalo = ncell/8 means one sampling every 8 cells and we will need to send a data with the length nhalo), we tested the effect take by compression the data with the size from double to float. The result is shown in the table2.4. From this table, we can clearly find that compressing the data into float size will still give benefit when combined with the calculation kernel. However, the effect will be smaller when the accuracy(necll) growth which means the amount of calculation growth. Also, the smaller 'nhalo' means fewer data transfers, which means less time consumption. Also, to simulate the real software more realistic, we also considered including the OpenMP for another test, the result is similar to the last one(see table2.5 and 2.6). From this mini-application, we can find that the compression will still benefit our kernel though have a limited effect. Though compared to calculation, the effect brought by communication is really small, it's not hard to imagine that once the data size becomes larger(which is really the thing in the real application). Anyhow, this modification can definitely lower the bandwidth requirement for communication. Now we can now start to modify the source code.

| ncell | nhalo | float_time(s) | double_time(s) |
|---|---|---|---|
| 100000 | ncell/2 | 261.982 | 264.587 |
| 100000 | ncell/4 | 252.67 | 257.162 |
| 100000 | ncell/8 | 249.412 | 249.817 |
| 100000 | ncell/16 | 247.153 | 247.339 |
| 10000 | ncell/2 | 2.54108 | 2.74232 |
| 10000 | ncell/4 | 2.41549 | 2.80097 |
| 10000 | ncell/8 | 2.37467 | 2.55179 |
| 10000 | ncell/16 | 2.34761 | 2.47443 |
| 1000 | ncell/2 | 0.0299569 | 0.0375347 |
| 1000 | ncell/4 | 0.0289351 | 0.0337446 |
| 1000 | ncell/8 | 0.028591 | 0.0305071 |
| 1000 | ncell/16 | 0.0278682 | 0.0312777 |

Table 2.4: Communication time for different ncell and nhalo config. The test is done with two nodes and each node has one task.

| ncell | nhalo | float_time(s) | double_time(s) |
|---|---|---|---|
| 100000 | ncell/2 | 114.888 | 116.297 |
| 100000 | ncell/4 | 108.691 | 111.183 |
| 100000 | ncell/8 | 107.47 | 109.169 |
| 100000 | ncell/16 | 105.283 | 106.481 |
| 10000 | ncell/2 | 1.18867 | 1.15762 |
| 10000 | ncell/4 | 1.08312 | 1.11132 |
| 10000 | ncell/8 | 1.05376 | 1.0832 |
| 10000 | ncell/16 | 1.04873 | 1.06448 |
| 1000 | ncell/2 | 0.0177266 | 0.00175192 |
| 1000 | ncell/4 | 0.0172013 | 0.0171973 |
| 1000 | ncell/8 | 0.0165005 | 0.0163586 |
| 1000 | ncell/16 | 0.0156856 | 0.0162979 |

Table 2.5: Communication time for different ncell and nhalo config with OMP_THREAD=4. The test is done with two nodes and each node has one task.

| ncell | nhalo | float_time(s) | double_time(s) |
|---|---|---|---|
| 100000 | ncell/2 | 16.2163 | 16.4481 |
| 100000 | ncell/4 | 14.3357 | 14.8465 |
| 100000 | ncell/8 | 13.7941 | 14.5594 |
| 100000 | ncell/16 | 13.481 | 13.837 |
| 10000 | ncell/2 | 0.504733 | 0.454218 |
| 10000 | ncell/4 | 0.465676 | 0.461985 |
| 10000 | ncell/8 | 0.45871 | 0.511093 |
| 10000 | ncell/16 | 0.487557 | 0.577424 |
| 1000 | ncell/2 | 0.0392923 | 0.0391511 |
| 1000 | ncell/4 | 0.0389243 | 0.0394692 |
| 1000 | ncell/8 | 0.0384033 | 0.0385646 |
| 1000 | ncell/16 | 0.0388547 | 0.0399244 |

Table 2.6: Communication time for different ncell and nhalo config with OMP_THREAD=64. The test is done with two nodes and each node has one task.

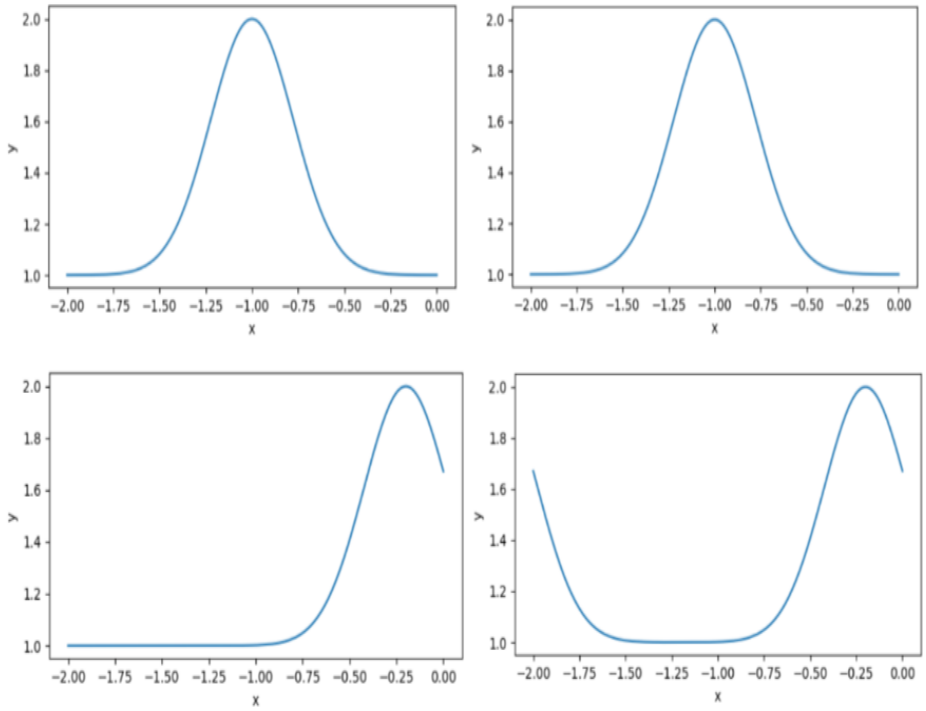


Figure 2.2: Mini-kernel: the up two figures show the origin wave shape, and the bottom two are the final shape. The left figures are the data in node 0 while the right two are in node 1.

### 2.2.2 Verify the compression ZFPlib

The compressing algorithm used in ZFPlib is based on the paper: "Fixed-Rate Compressed Floating-Point Arrays"[4] As found in the documentation, the ZFP lib has the CUDA support but only with the fixed rate compress. By designing a toy test to compress and decompress the random-generated data on the device(GPU), then comparing the result, we can practice and verify the correctness of the lib. The rate used in the fix-rate mode will affect the accuracy while also used as the limitation that limits the size of the data after compression. By referring to the lib doc, the $rate = maxibits/4^d$ while $maxbits = 4^d * CHAR\_BIT * sizeof(Type)$, we finally set it to 32 to achieve the compression rate at 2(the compressed data have the size half smaller than the origin one) while keeping the data error smaller than $10^{-6}$.

## 2.3 Modification in Walberla

### 2.3.1 Data structure duplication

Since Walberla is a huge software, we must do things carefully. So the idea is first to duplicate the origin data structure and test if it works well, then add the data compression. The first test is to copy the origin data structure and test the execution with the most basic case - fully blocked communication. The modification run correctly and performance well.(src here)

### 2.3.2 Modify the MPI functions

After verifying, we try to combine the compression into the MPI function. Now we check the resource code referring to the data communication in the Welberla. First of all, our data is all located on the GPU in the test case, so what we can develop is to compress the data before transfer and decompress after transfer on GPU, just using the lib which we have verified before.

#### Analysis of the MPI functions

A communication step is divided into several phases:

1) First, for all expected messages, a non-blocking receive (MPI_IRecv) is scheduled (here). This is done even before any messages are sent or packed such that the MPI system can already prepare for the

expected messages, for example, by allocating buffers.

2) The next step is to fill and send the outgoing buffers (here), again using non-blocking MPI functions. Note that the possibly time-consuming operation of serialization can be done after the receives have been scheduled.

3) The last wait step finally receives and unpacks incoming messages (here). Internally, this phase is realized by calling a MPI_Waitany function, that waits for any of the previously scheduled sends or receives to finish. If this function returns with a finished receive operation, the message is given to the user while further messages can be received.

### Idea and preparing

The idea for the compression is clear:

1) In the first step, the schedule is done by dispatching a "header" message. So what we need to do is to modify the size of the buffer to half(or another ratio that is defined by the compression rate).

2) On the second step, compress the data before it is packed into the *SendBuffer*.

3) Finally, decompress the data after it is received but before any further usage.

### Current State

The work is not yet finished due to the complicity of the huge software. The visible result yet is that the modification is valid for the benchmark overlap scenarios, while the performance is limited by the compression issue. However, the integration needs more time and work since we surely reduced the MPI cost(Figure2.4 shows the size of the data need to be transferred before and after the compression) while bringing the compression issue as the bottleneck(The performance data shown in here).

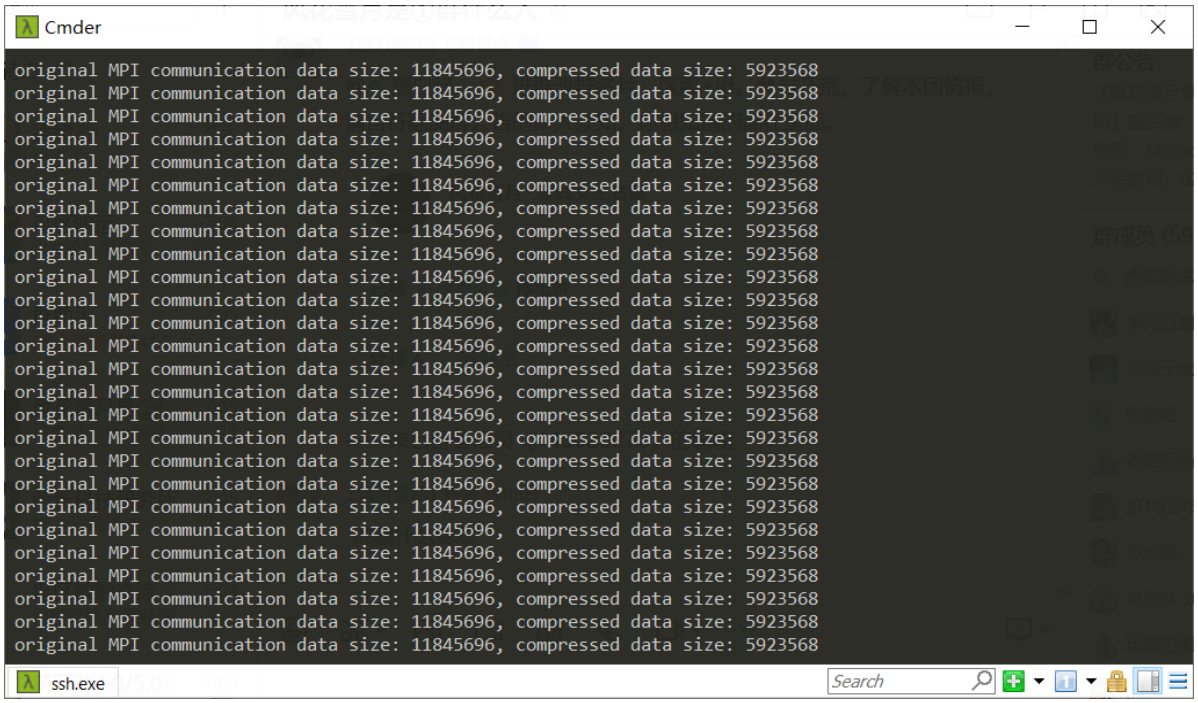


Figure 2.3: communication data size before/after compress: The reason why the size is not a half is that when meeting with the small size of data(e.g. one double), the compressed data may need more size than the origin(in the one double data case, the compressed data need 16 bits while the origin data need only 8 bits). So the data is not strictly halved after compression

To verify the compression will give advantages, we analyze the modified program using (NVIDIA Nsight Systems). By checking the cost of the compress and the origin pack, we found the cost is comparable(shown in Figure2.4). In fact, the time consumed in the pack is about 5.21$\mu$s while the unpack runs 5.06$\mu$s. So

by this result, we can assume that by further modifying the bottom-level functions, we can get significant speed up.

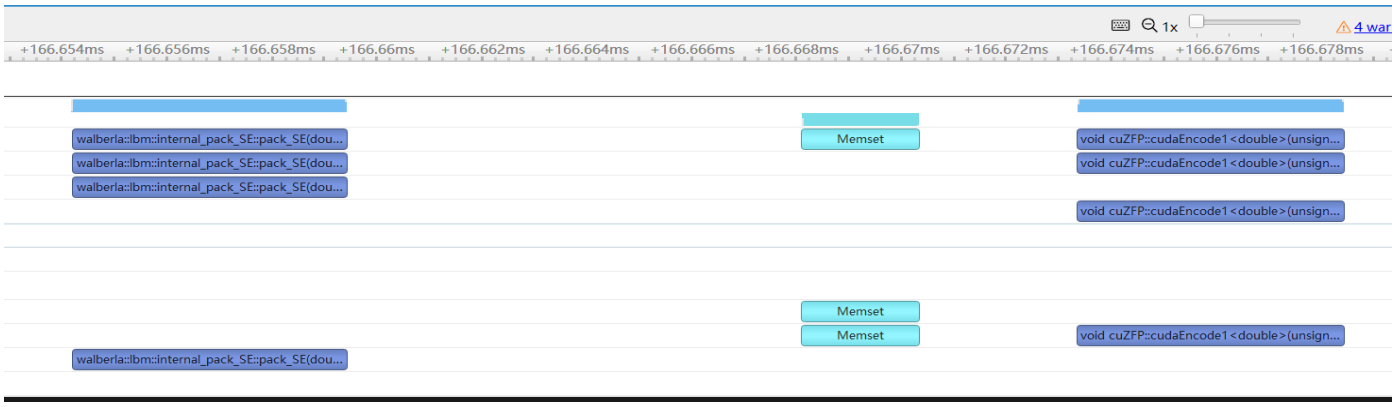


Figure 2.4: Profiling for MPI cost

# 3 Conclusion

## 3.1 Concludion

During this project, we test and verified in two directions to optimize the huge software Walberla. We gain some results in both the communication and vectorization parts. In the vectorization part, we improve the MLUPS by about 10% by decreasing the non-essential calculation data. Though the idea of 'SIMD' do not gain obvious results, this idea surely decreased the memory load. For the communication part, overlap gives significant improvement. Then to further improve the communication idea, we also try to compress the data during communication. We build the mini-application for testing while getting the result shows the effect is limited. We still combined the compress with the origin code, finding the compression cost is not large. So we can conclude that though this idea was not effective now, when the communication data size comes huge, it will give benefits.

## 3.2 Summary for this project

The main difficulties while working on the project were related to the huge and complex code of WalBerla. Additionally, many programming and data dependencies should be taken into account in the working process. The code has many levels of abstraction, e.g. finding a particular function's call chain might be a time-consuming and difficult process. It is important to realize that neither can a developer fix everything at once nor is it always possible to foresee what might go wrong when introducing new features. Tackling such an amount of code was very challenging, e. g. one single change in the code led to a bug we could not have recovered from for a week. Optimizing such a large software is not easy work, so we always do as build up some small test cases and then gradually port the modification into the software. Though this work can not yet give a direct effect on the whole software scope, the toy tests can verify that all the works can truly somehow benefit the execution process.

# References


[1] Martin Bauer, Sebastian Eibl, Christian Godenschwager, Nils Kohl, Michael Kuron, Christoph Rettinger, Florian Schornbaum, Christoph Schwarzmeier, Dominik Thönnes, Harald Köstler, et al. walberla: A block-structured high-performance framework for multiphysics simulations. *Computers & Mathematics with Applications*, 81:478–501, 2021.

[2] Martin Bauer, Harald Köstler, and Ulrich Rüde. lbmpy: Automatic code generation for efficient parallel lattice boltzmann methods. *Journal of Computational Science*, 49:101269, 2021.

[3] Moritz Lehmann, Mathias J. Krause, Giorgio Amati, Marcello Sega, Jens Harting, and Stephan Gekle. Accuracy and performance of the lattice boltzmann method with 64-bit, 32-bit, and customized 16-bit number formats. *Phys. Rev. E*, 106:015308, Jul 2022.

[4] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20, 08 2014.

[5] Joshua Socolar. Chaos. In Robert A. Meyers, editor, *Encyclopedia of Physical Science and Technology (Third Edition)*, pages 637–665. Academic Press, New York, third edition edition, 2003.