Vectorization-Based GPU Optimizations for Walberla

Shuxin Zheng
Department of Computer Science
University of Trento
Italy
shuxin.zheng@studenti.unitn.it

Abstract *Walberla is a multiphysics framework based on LBM. During this report, I will present two modifications to the code. The first one is to reduce memory usage by adding an 'if' condition, while the second one is to eliminate the affection due to the use of 'if' in GPU. Some specially designed toy test cases will be shown, and the result of integrating these modifications into the Walberla will be discussed here.*

1 Introduction

1.1 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.

The LBM compute kernels make use of various node-level optimization techniques. By coupling the LBM with the RPD (high efficient rigid particle dynamics), multiphysics simulations can be realized on a large scale and without compromising on performance or scalability. In a word, it's important for parallelizability.

Another important part of walberla is the code generation function. As is always the case that the HPC software usually has to be modified extensively 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.[1] Backends then generate C/C++, CUDA, OpenCL 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.

The core data structure of the Walberla is the BlockForest[2]. 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 Fig 1.1, which is maintained between neighboring subdomains at all times.

1.2 Introduction to S2A test case

The S2A test case consists of a modern sedan car placed inside a wind tunnel. The geometry of the car is displayed in Fig 1.2. The test case first reads the geometry of the car from a .obj

[Draft] • Branch: main@35e0e82
Author: Shuxin Zheng, 2022-10-13 14:08:21 +0200

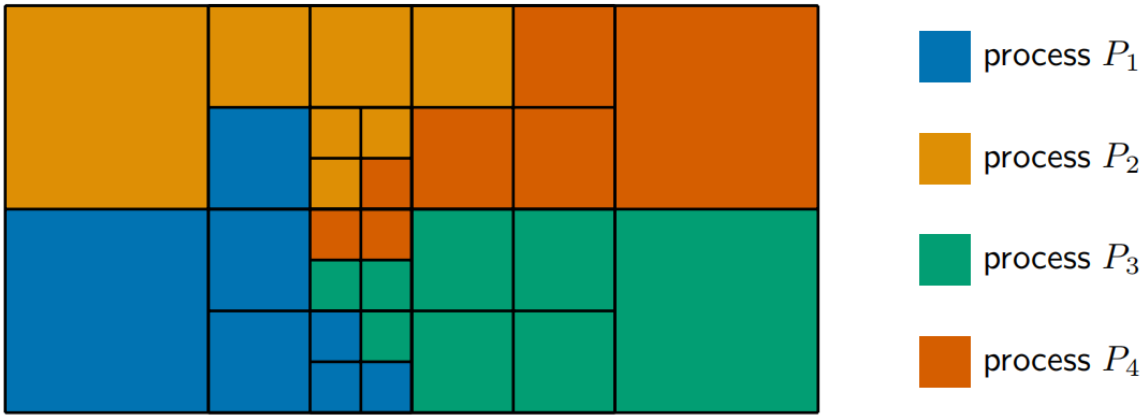

Figure 1.1: Figure: block forest

file. 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 for 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.

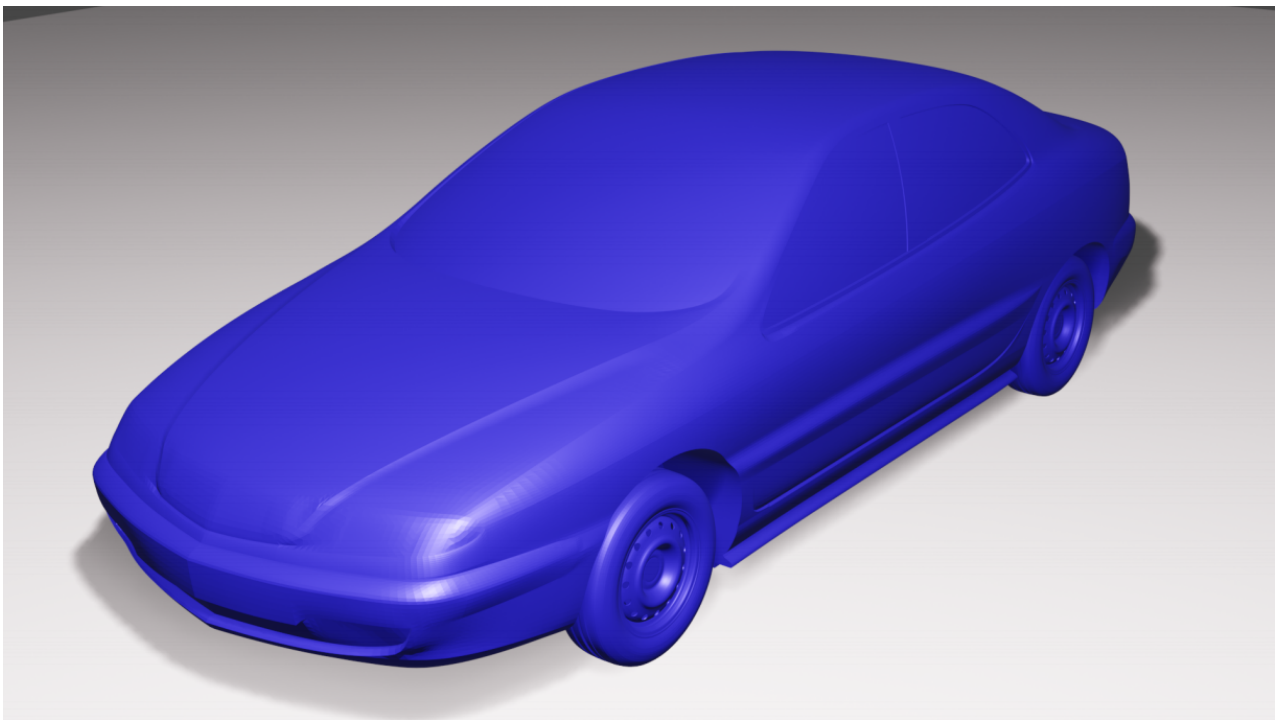

Figure 1.2: Figure: car for test case S2A

2 Performance Analysis Tools

As to check the final result of the optimization due to our modification, we employed several tools. The first kind of data comes from the Walberla it self, after running the test,

it will give out a MLUPS table which reports the lattice cell and fluid lattice cell updates per second. We use these data to measure the final optimization.

Besides, to analyze the performance bottleneck, we also need help from NVIDIA Nsight Systems(nsys) and NVIDIA Nsight Compute(ncu). These tools helped check the GPU running time, also the detailed compute/memory usage percentage. For the time limitation, we test the program using only the weak scaling otherwise the analysis tools will run extremely long.

Another thing I want to mention here is the JUBE script, by writing a JUBE script we can compile the code automatically from the very beginning so no more trouble to solve the compiling problem after modifications every time.

3 IDEA: FLAGFIELD

It's not very easy, especially in Walberla, to ignore the block inside the domain of the car due to the way the mesh is implemented. Before the development of the FlagField, we need to perform all computations both inside and outside the domain of car. So by implement the FlagField, we hope to reduce the cost of computing kernels by not computing the domain inside the car. To achieve this goal, we added a '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.

3.1 RESULT & ANALYSIS

We check the performance of this modification using NVIDIA Nsight Compute to check whether the modification have 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 as Table 3.1. As we can obviously find, the memory usage is decreased by around 9.1%.

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 3.1: GPU throughput for Master and FlagField branch

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 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 Table 3.2, Table 3.3, Table 3.4 and Table 3.5. In these analysis, we can find that the memory transfer between the local and the L1 Cache is reduced by almost a half. Also, the utilization of communication from local to 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 3.2: 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 3.3: L1/TEX Cache for Master and FlagField branch

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 3.4: L2 Cache for Master and FlagField branch

By checking the MLUPS table, we get the final result for this step. The comparison table is shown as table 3.6. We do the test both with and without the NVIDIA analysis tool and the FlagField performance around 10% better than the master one in both cases.

3.2 CONCLUSION

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

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 3.5: Device Memory for Master and FlagField branch

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

Table 3.6: MLUPS table for Master and FlagField branch

4 IDEA: SIMD INSTRUCTIONS

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.

4.1 TOY TEST 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.

```
//origin if condition in FlagField
if (((((int64_t)
(_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[3] here, especially $4 * x * (x - 1)$, where x is the random filled unsigned int value). The 'if' command compares the two randomly filled array.

```
__global__ void compare(const unsigned int* a, const unsigned int* b,
double *rd, unsigned int *c)
{
    // this thread handles the data at its thread id
    int tid = blockIdx.x;
    double x = rd[tid];
    // Non-SIMD Version
    // if(a[tid] != b[tid])
    // SIMD Version
    if(__vsetne4(a[tid], b[tid])) {
        for(std::size_t j =0; j < 1e5 ; ++j)
            x = 4 * x * (x-1);
    } else {
        for(std::size_t j =0; j < 1e5 ; ++j)
            x = 3.8 * x * (x-1);
    }
    rd[tid] = x;
}
```

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.

```
__global__ void compare(const unsigned int* a, const unsigned int* b,
double *rd, unsigned int *c)
{
    // this thread handles the data at its thread id
    int tid = blockIdx.x;
    double x = rd[tid];
    for(std::size_t i =0; i < 1e6 ; ++i) {
        // Non-SIMD Version
        // if(a[tid] != b[tid])
        // SIMD Version
        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  }
```

4.2 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 4.1.

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 4.1: 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 4.2.

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 4.2: 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 as Table 4.3.

It's easy to find that the SIMD command give advantages in reduce 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 4.4, Table 4.5 and Table 4.6.

In these tables, its 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, at the current problem sizes, the SIMD commands needs to load the data into the core only once and store it back only when the computation is finished. While in the case of non-SIMD code, the data is loaded and stored for every single computation. So there should be huge improvement to adapting the SIMD command if the data in code have this reuse feature.

4.3 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,

[Draft] • Branch: main@35e0e82
Author: Shuxin Zheng, 2022-10-13 14:08:21 +0200

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 4.3: GPU throughput for inside 'for' loop cases

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 4.4: 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 4.5: 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 4.6: Device Memory for inside 'for' loop cases

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 4.7 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 4.7: MLUPS table for FlagField with/without SIMD

4.4 CONCLUSION

Out of our toy case testing, the SIMD instruction helps enlarge the throughput by enable the data reuse and so reduce the memory transfer, and it will work in special case (e.g called enough large time, and the same core will reuse the same data). Unfortunately, our S2A case do not benefit from this instruction. But this work is still valuable since there be more other cases which may fit this special case.

5 SUMMARY

During this project, we verified the optimization of the FlagField and obtained a 10% performance upgrade. To further make the optimization thorough, we tried to use SIMD instruction to solve the bad effect taken in by the 'if' condition which is used in FlagField. To do this, we first check the performance of the SIMD instruction by designing two special toy test cases and found out there be specific cases in which the SIMD instructions will give benefits. Unfortunately, the S2A case we used during this project does not fit the specific situation, so it does not get any improvement from the SIMD instruction out of the final test. However, the toy test is still meaningful since there be many more different running cases so we can know there is a way to optimize it once meet the kind of case which will call the 'if' condition that compares two numbers for huge time and the computation is limited due to memory transfer. Optimization is never easy work, we still have other directions to test and verify. Since we have performed ncu for GPU throughput analysis, optimizing the memory transfer should be surely the next step as shown during our test.

6 Acknowledgments

A research project is never the work of one person alone. First of all, I would like to thank my supervisor Jayesh Badwaik who direct me to get familiar with the software and give me several suggestions on design and test during this project. The discussions with him benefit me much for understanding varies effect taken from different instructions.

Also, one person that cannot be left unmentioned is Ivo Kabadshow who organized the Guest Student Program. He provided invaluable information about the usage of the cluster and Jenkin.

Last but not least, I wanna thanks my fellow guest students for making this programme a special experience that I will never forget.

References

[1] M. Bauer, H. Köstler, and U. Rüde. **lbmpy: Automatic code generation for efficient parallel lattice boltzmann methods**. Journal of Computational Science, 49:101269, 2021. doi:https://doi.org/10.1016/j.jocs.2020:101269.

[2] M. Bauer, sebastian Eibl, C. Godenschwager, N. Kohl, M. Kuron, C. Rettinger, F. Schornbaum, C. Schwarzmeier, D. Thönnes, H. Köstler, and U. Rüde. **walberla: A block-structired high-performance framework for multiphysics simulations**. CoRR, abs/1909.13772, 2049. http://arxiv.org/abs/1909.13772.

[3] J. Socolar. **Chaos**. In R. A. Meyers, editor, Encyclopedia of Physical Science and Technology (Third Edition), pages 637–665. Academic Press, New York, third edition edition, 2003. https://www.sciencedirect.com/science/article/pii/B0122274105000946, doi:https://doi.org/10.1016/B0-12-227410-5/00094-6.