Vectorization-Based GPU Optimizations for Walberla

September 21, 2022 | Shuxin Zheng | Jülich Supercomputing Centre

Part I: Introduction

What is Walberla?

Welberla

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. WelBerla's main focus are 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.

Why is LBM important?

Parallelizability

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

How does WalBerla achieve peak performance?

Code generation

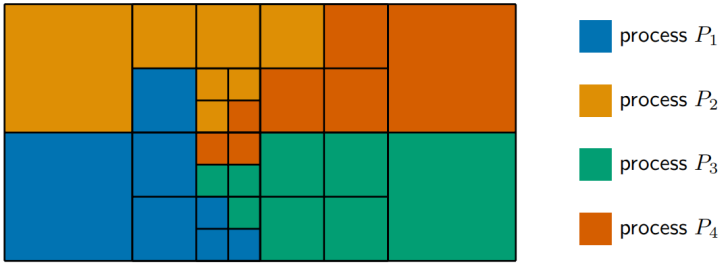
HPC software usually has to be modified extensively in order to make full use of new hardware architectures. WalBerla employs **code generation techniques** to generate time-critical numerical kernels from a high-level, domain-specific formulation. WalBerla used the *pystencils* metaprogramming project 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. 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.

Part II: The S2A Test Case

Pre-knowledge

Block Forest

It is responsible for the domain partitioning and acts as a container for all data needed by the simulation. It manages the domain partitioning into blocks, independent of data structures required for the actual simulation. A figure to illustrate is shown below:



JÜLICH Forschungszentrum

Test Case

S2A Test Case

The S2A test case first reads the geomerey of a car from a .obj 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 extened 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.

JÜLICH Forschungszentrum

A general idea

FlagField

However, it is not very easy, especially in Walberla, to ingore the block inside the domain of the car, due to the way the mesh is implemented. Before developing the FlagFied, we need to perform all computations both inside and outside the domain of car. By implement the FlagField, we hope to reduce the cose of computing kernels by not computing the domain inside the car.

the MLUPS table

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

SIMD instructions

CUDA MATH API

By developing the FlagField, we reduced the cost of computing kernels by not computing the domain inside the car. However, putting an ’if’ condition in a GPU-based for loop is not optimal dur to parallel nature of GPU threads. The motivation for using SIMD instructions here is to reduce the penality of that ’if’ condition.

The toy test

By developing the SIMD, we modify the origin comparing in 'if' to use the function '__vsetne4()' in CUDA MATH api. The function performs per-byte (un)signed comparison. It splits 4 bytes of each argument into 4 parts, each consisting of 1 byte, and retuens 1 if a != b, else return 0.
So to be familiar with this function, and also verify the performance, I developed two version of the testing case: First fill three arrays with random unsigned int value, then call a kernel function, which do some calculation(for enlarge the GPU time) under the 'if' condition(using the *Logistc Map* here, especially $4 * x * (x - 1)$, where x is the random filled unsigned int value). The 'if' command comparing the two random filled array. By develop two version of tests, we get the results:

JÜLICH Forschungszentrum

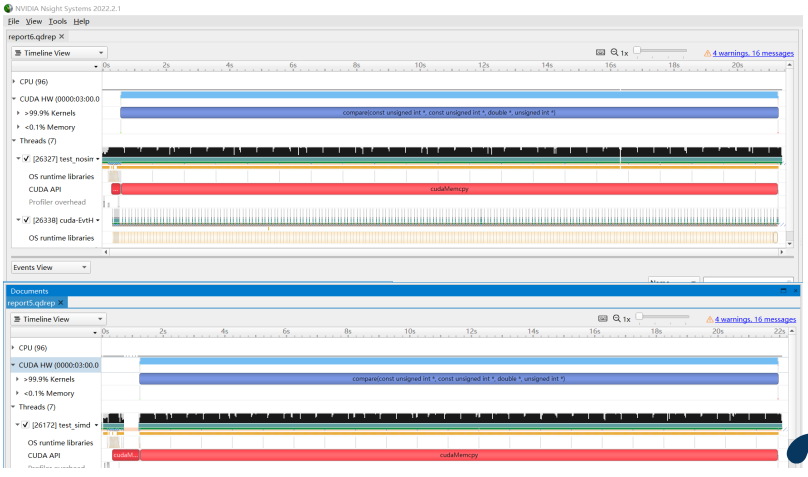
Results1

By put the data array with size 1024*1024, a for loop with iteration time 1e5, I got about 1.5x speed up when put the 'if' condition inside the loop.

JÜLICH Forschungszentrum

Results2

By put the data array with size 1024*1024, a for loop with iteration time 1e6(for ensure and emulate the performance fluctuations), no obvious difference shown in case I put the 'if' condition outside the loop



JÜLICH Forschungszentrum

Current test

With the assumption that the kernel function in WalBerla is called large more than once, I adapting this SIMD modification into the origin branch. The preliminary tested performance shows a 1.11x speed up:

Master with profile	**SIMD with profile**
633.677 MLUPS	704.314 MLUPS
(millian lattice cell updates per second)	(million lattice cell updates per second)
633.677 MLUPS / process	704.314 MLUPS / process
629.713 MFLUPS	699.908 MFLUPS
(millian fluid lattice cell updates per second)	(million fluid lattice cell updates per second)
629.713 MFLUPS	699.908 MFLUPS
9669.15 time steps / second	10747 time steps /second

Future Work

work need to be finished

- Improve the test for including SIMD command on basic code
- Deploy the SIMD into the FlagField version
- Test the performance for SIMD inside the FlagField

Vectorization-Based GPU Optimizations for Walberla

September 21, 2022 | Shuxin Zheng | Jülich Supercomputing Centre