EasyChair Preprint

№ 2197

Accelerating Lattice Boltzmann Method by Fully Exposing Vectorizable Loops

Bin Qu, Song Liu, Hailong Huang, Jiajun Yuan, Qian Wang and Weiguo Wu

December 18, 2019

Accelerating Lattice Boltzmann Method by Fully Exposing Vectorizable Loops

Bin Qu, Song Liu(✉), Hailong Huang, Jiajun Yuan, Qian Wang, and Weiguo Wu(✉)

School of Computer Science & Technology,
Xi'an Jiaotong University,
Xi'an Shaanxi 710049, China
{qbqbqb,hhl15015970612,plusss,Rebeccamango}@stu.xjtu.edu.cn
(✉){liusong,wgwu}@mail.xjtu.edu.cn

Abstract. Lattice Boltzmann Method (LBM) plays an important role in CFD applications. Accelerating LBM computation indicates the decrease of simulation costs for many industries. However, the loop-carried dependencies in LBM kernels prevent the vectorization of loops and general compilers therefore have missed many opportunities of vectorization. This paper proposes a SIMD-aware loop transformation algorithm to fully expose vectorizable loops for LBM kernels. The proposed algorithm identifies most potential vectorizable loops according to a defined dependence table. Then, it performs appropriate loop transformations and array copying techniques to legalize loop-carried dependencies and makes the identified loops automatically vectorized by compiler. Experiments carried on an Intel Xeon Gold 6140 server show that the proposed algorithm significantly raises the ratio of number of vectorized loops to number of all loops in LBM kernels. And our algorithm also achieves a better performance than an Intel C++ compiler and a polyhedral optimizer, accelerating LBM computation by 147% and 120% on average lattice update speed, respectively.

Keywords: lattice boltzmann method, auto vectorization, performance, SIMD, loop transformation algorithm

1 Introduction

Lattice Boltzmann Method (LBM) [8] is a numerical simulation of physical phenomena. It is one of the most important CFD methods and is widely used for single-phase/multiphase flow, kinetics of surface and kinetics of crystallization and so on. LBM simulation plays an important role in aviation, water conservancy and thermal engineering and so on. Hence, accelerating the computation of LBM leads to the decrease of computing simulation costs for many industries. Previous studies on LBM computing are mainly about simulation algorithm. Qian et al proposed a Lattice Bhatnagar-Gross-Krook (LGBK) model [20] to

reduce computation by replacing collision matrix with single relaxation time coefficients. So far, LGBK is one of the fastest LBM models based on uniform grid scheme. As the complexity of simulation grows, the number of grids increases rapidly, which results in massive storage and temporal overhead. Therefore, later studies present many large-eddy simulation methods based on nonuniform grid scheme [16] to reduce spacial and temporal complexity.

Since the first generation of commercial vector processor — Pentium II is released in 1996 [21], vector computing becomes a necessary function of modern processors, especially at the age when AI applications are rapidly growing. Meanwhile, hardware vendors have also designed many SIMD instruction sets (also known as intrinsic instructions) for programmers to implement vector computing in their codes, such as MMX, SSE and AVX. More conveniently, major modern C/C++ and Fortran compilers provide auto vectorization function that automatically translates appropriate loops to SIMD instructions. Because of the natural data-level parallelizability, it is valuable to accelerate LBM by using vector computing. In terms of software running on general purpose processors, previous studies on optimization of vector computing are mainly about data organization and data layout schemes. Struct-of-Array (SoA) scheme [1,22] and data alignment [14] are the most common techniques that are applied for SIMD optimization. However, the loop-carried dependency (a statement in one iteration of a loop depends on a statement in a different iteration of the same loop) [9,11] roosted in LBM kernels is against the rule of parallel and vector computing, and thus it prevents auto-vectorization for the innermost loops of LBM kernels. Unfortunately, the data organization and data layout optimizations can not solve this problem.

The LBM kernels, as the hot spots of program, generally consist of multiple nested loops (space dimensional loops) within a time dimensional loop. Loop transformations are considered as effective techniques to optimize LBM codes. In recent decade, the polyhedron theories [5, 7, 12, 13, 17, 24] have been rapidly developed to boost the rise of many optimizer frameworks that guide performing efficient loop transformations. These transformations enable the parallel execution of loop codes with dependency preservation. In previous studies, the objectives of loop transformations are mainly about exploiting parallelism for outer-dimensional loops, minimizing synchronization and enhancing data locality, but vectorization for innermost loops (vectorizable loop in a nested loop) is rarely considered. Due to the phase-ordering problem, some affine schedule algorithms in optimizer frameworks even transform the innermost loop to a non-vectorizable loop, which seriously wastes the computing power of vector units and is adverse to computing performance. A few studies focus on SIMD optimization based on polyhedron model. Kong et al. proposed a polyhedral compiling framework [15] which aims for integrated data locality, multi-core parallelism and SIMD execution. In this framework, loop codes are blocked and plain codes in loop blocks are translated to SIMD codelets (code blocks). Experiments demonstrate that this framework achieves times of speedup than previous loop transformation meth-

ods. However, they do not present any clear algorithm to guide automatic SIMD codelet generation.

In this paper, we propose a novel SIMD-aware loop transformation (SLT) algorithm which identifies the plain loops that have potentiality to be vectorized and guides these loops to be transformed into vectorizable loops for LBM kernels. The identification is based on dependency analysis on a defined dependency table. This algorithm is easier to understand than the polyhedron-based loop transformation method since it does not require much knowledge of convex analysis and linear programming. The SLT algorithm has been validated for LBM kernels on an Intel Xeon Gold 6140 server which has processors with the Skylake microarchitecture and supports AVX-512 SIMD instruction set. Our algorithm can detect much more potentially vectorizable loops than Intel C++ compiler and a state-of-the-art polyhedral optimizer - PLuTo [4], and thus it can significantly accelerate the computation of LBM codes. Besides, since LBM is a kind of typical stencil computation, the proposed algorithm is also suitable for any other LBM-like numerical computations, such as FDTD, Gauss-Seidel iteration and convolutional neural network. In summary, this paper makes the following contributions.

- We propose a novel SIMD-aware algorithm which can identify all possible potential vectorizable loops that may be missed by compilers and polyhedral optimizers due to the loop-carried dependencies within LBM kernels. The identification is based on dependency analysis on a defined dependency table.
- The proposed algorithm can guides the identified loops to automatically perform loop transformations and array copying techniques to generate new LBM kernels with compiler-identifiable vectorizable loops. The algorithm has a polynomial-time solution.
- Experimental results demonstrate that the proposed algorithm can significantly raise the ratio of number of vectorized loops to number of all loops in the LBM kernels and achieves a better performance than a polyhedral optimizer PLuTo.

The rest of the paper is organized as follows. Section 2 presents related work about LGBK model and auto vectorization. Section 3 illustrates the details of our SIMD-aware loop transformation algorithm. Section 4 presents the experimental results and analysis. Finally, section 5 concludes this paper and points out the future work.

2 Related Work

2.1 LGBK Model

LGBK model is one of the most popular LBM models. Previous work [18] has revisited the basic principle of LGBK. LGBK model includes DnQb models where a particle is collided with b surrounding particles (including itself) in an n-dimensional space, such as D2Q9 model and D3Q19 model. Generally, in the

LGBK model, the procedure of a single time step of LBM is divided into two phases:

– Collision

$$f_i^*(\boldsymbol{x},t) = f_i(\boldsymbol{x},t) - \frac{1}{\tau}[f_i(\boldsymbol{x},t) - f_i^{(eq)}(\boldsymbol{x},t)], \tag{1}$$

– Streaming (or propagation)

$$f_i(\boldsymbol{x} + \boldsymbol{e_i}\Delta t, t + \Delta t) = f_i^*(\boldsymbol{x},t), \tag{2}$$

where i is the direction of particle velocity; $\boldsymbol{x}$ is the location of particle; t is current time step; Δt is a time slot; $\boldsymbol{e_i}$ is the particle velocity in direction i; τ is relaxation time; $f_i(\boldsymbol{x},t)$ is the distribution function of particle in direction i; and $f_i^*(\boldsymbol{x},t)$ is the equilibrium distribution function after collision. In D3Q19 model, $i \in [0,18]$, the directions of particle velocity $\boldsymbol{e_i}$ are from $\boldsymbol{e_0}$ to $\boldsymbol{e_{18}}$ (as shown in Figure 1). In equation (1), the equilibrium distribution function $f_i^{(eq)}(\boldsymbol{x},t)$ is defined as

$$f_i^{(eq)}(\boldsymbol{x},t) = w_i\rho[1 + \frac{3e_iu}{c^2} + \frac{9(e_iu)^2}{wc^4} - \frac{3u^2}{2c^2}], \tag{3}$$

where ρ denotes fluid density, u denotes fluid velocity, and c denotes lattice speed. And in D3Q19 model, the weighing factor w_i is defined as

$$w_i = \begin{cases} \frac{1}{3},\ i = 0 \\ \frac{1}{18},\ i = 2,4,6,8,9,14 \\ \frac{1}{36},\ i = 1,3,5,7,10,11,12,13,15,16,17,18. \end{cases} \tag{4}$$

Obviously, the computation of LGBK model is suitable for parallel execution. However, in LGBK model, a lattice references its neighboring lattices, which indicates loop-carried dependencies that prevent auto vectorization for loop codes. In this paper, we use the D3Q19 LGBK model to illustrate and validate our algorithm for LBM codes.

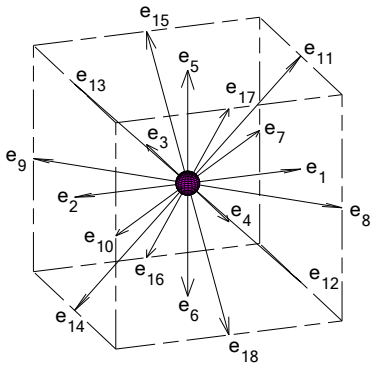


Fig. 1. Particle velocities of D3Q19 model.

2.2 Auto Vectorization

Auto vectorization is a commonly-used technique embedded in most modern compilers. When this technique is activated in compilation, it scans each loop in LBM kernels and tries to translate vectorizable loops into SIMD instructions. Since not all kinds of loops are vectorizable, a compiler that supports auto vectorization must identify vectorizable loops based on certain features. If a loop can be automatically vectorized, it must satisfy the following conditions [2,15,25]

1. **Countable**: the loop trip count must remain constant in the duration of the loop.
2. **Single entry & exit**: the loop must be a perfect loop with only one entry and one exit.
3. **Containing straight-line code**: branches are not permitted, but the *if*{...} statement is permitted as long as it can be implemented as masked assignments.
4. **Innermost loop of a nest**: the loop must be an innermost loop of a nest or a one-dimensional loop.
5. **Without function calls**: no function calls in loop body.

In addition, if a loop contains any loop-carried dependencies, parallel execution of multiple iterations may lead to error results. Hence, vectorizable loop does not allow the existence of loop-carried dependency. Most previous loop transformation methods are effective to both preserve dependencies and implement parallel execution by exploiting pipeline/wavefront parallelism. However, the transformed innermost loops are usually uncountable and are with multiple entries & exits and branches. Therefore, they can not be vectorized by compilers.

The expected performance of vectorization should also be considered, and it helps to decide whether vectorization is profitable. Inappropriate data organization, such as Array-of-Struct (AoS) and unaligned data, may lead to unsatisfied performance of vectorization. Trifunovic et al. have proposed a vectorization cost model to estimate speedup [23]. The cost model computes the expected speedup by comparing the total execution time of vectorized loops with the execution time of scalar loops. The cost model is also applied in modern compilers. If the expected speedup of a vectorizable loop is not greater than one, the loop will not be vectorized by compiler.

3 SIMD-aware Loop Transformation (SLT) Algorithm

For simplicity, the loop referred is related to the innermost loop of a nested loop and the dependency referred is related to the loop-carried dependency of innermost loop without specification. All pseudo codes are written in C-language-like format.

3.1 Representation of Loop Domain

We use a triple (l, i, s) to represent the domain of each loop of innermost loops, where l denotes the sequence number of one loop of innermost loops, i denotes the iteration variable of current loop and s denotes the sequence number of a statement in innermost loops. For instance, Figure 2 shows an example of innermost loops. There are 2 innermost loops and 3 statements in the example. Given that $i = 2$, then triple $(1, 2, 1)$ denotes statement $S1$ in innermost loop $L1$ and triple $(2, 2, 3)$ denotes statement $S3$ in innermost loop $L2$.

```
for ...
    for ...
    {
        L1: for(i=1;i<=4;i++)
            S1: B[i]=A[i+1];
        L2: for(i=1;i<=4;i++)
        {
            S2: A[i]=A[i+1];
            S3: C[i]=A[i];
        }
    }
```

Fig. 2. Example of innermost loops.

Let $\boldsymbol{x}_t(l_t, i_t, s_t)$ and $\boldsymbol{x}_s(l_s, i_s, s_s)$ respectively denote the target and source iteration instances of a loop-carried dependency among innermost loops, the dependence vector $\boldsymbol{d}$ of the loop-carried dependency is defined as

$$\boldsymbol{d} \equiv \boldsymbol{x}_t - \boldsymbol{x}_s = (\Delta l, \Delta i, \Delta s). \tag{5}$$

The loop-carried dependencies are detected based on Bernstein Conditions [6].

Theorem 1 (Bernstein Conditions). *Given two references, there exists a dependency between them if the three following conditions hold*

1. *they reference the same array (cell);*
2. *one of this access is a write;*
3. *the two associated statements are executed.*

According to Bernstein Conditions, we can find out all dependence vectors of loop-carried dependencies and construct a dependence table with these vectors. The dependence table is used for identification of vectorizable loops and guidance of loop transformations. For instance, Table 1 shows the dependence table D of the example of Figure 2.

3.2 Identification of Vectorizable Loops

L.N Pouchet has pointed out that no dependence between points of a hyperplane indicates parallelism on the hyperplane [19]. In Figure 2, loop $L1$ can not be

Table 1. Dependence table D of example loops.

Dependency	Type	$\boldsymbol{x}_s(l_s, i_s, s_s)$	$\boldsymbol{x}_t(l_t, i_t, s_t)$	$\boldsymbol{d}(\Delta l, \Delta i, \Delta s)$
$\boldsymbol{d}_1$: S1(RHS)→S2(LHS)	anti	$(1, i, 1)$	$(1, i+1, 2)$	$(0, 1, 1)$
$\boldsymbol{d}_2$: S2(RHS)→S2(LHS)	anti	$(1, i, 2)$	$(1, i+1, 2)$	$(0, 1, 0)$
$\boldsymbol{d}_3$: S2(LHS)→S3(RHS)	flow	$(1, i, 2)$	$(2, i, 3)$	$(1, 0, 1)$

vectorized, while loop $L2$ is vectorizable. Iterations of loop $L1$ can not be parallel executed due to the dependencies $\boldsymbol{d}_1$ and $\boldsymbol{d}_2$ (as shown in Table 1) between points of hyperplane $[1, 0, 0] \cdot [l, i, s]^T = 1$. Based on Pouchet's work, the constrains of vectorizable loop can be summed up as Lemma 1.

Lemma 1 (Constrains of Vectorizable Loop). *The l^{th} innermost loop is vectorizable if $\forall \boldsymbol{d}(0, \Delta i, \Delta s) \in D(l_s = l, l_t = l)$:*

$$\Delta i = 0.$$

A similar goal is to find dependency that prevents vectorization (illegal dependency) or not (legal dependency). The features of these two kinds of dependencies can be summed up as Lemma 2 and Lemma 3.

Lemma 2 (Illegal Dependency). *A dependency $\forall \boldsymbol{d}(\Delta l, \Delta i, \Delta s) \in D$ prevents vectorization if*

$$\Delta l = 0 \quad \textit{and} \quad \Delta i \neq 0.$$

Lemma 3 (Legal Dependency). *A dependency $\forall \boldsymbol{d}(\Delta l, \Delta i, \Delta s) \in D$ does not prevent vectorization if*

$$\Delta l \neq 0 \quad \textit{or} \quad \Delta i = 0.$$

Based on Lemma 1, we can identify vectorizable loops. For potentially vectorizable loops, we can find out the illegal dependencies that prevent vectorization. Then we try to transform illegal dependencies to legal dependencies by modifying Δl.

3.3 Loop Transformations for Vectorization

Statement Rearrangement Statement rearrangement is a considerable way to transform a potentially vectorizable loop to a vectorizable one. It is able to modify Δl of illegal dependency and does not incur uncountable loop trip, multiple entries & exits and branches which dose not satisfy the requirements of auto vectorization. For instance, to legalize dependency $\boldsymbol{d}_1$ in Table 1, we can perform statement rearrangement on loop $L1$ and move statement $S2$ to $L2$. Hence, $\boldsymbol{d}_1$ is legalized to $(1, 1, 1)$ and loop $L1$ becomes vectorizable. The example codes after the rearrangement are shown in Figure 3. If statements in the last loop need to be rearranged, we create a new loop and move statements to the new loop.

```
for ...
    for ...
    {
        L1: for(i=1;i<=4;i++)
            S1: B[i]=A[i+1];
        L2: for(i=1;i<=4;i++)
        {
            S2: A[i]=A[i+1];
            S3: C[i]=A[i];
        }
    }
```

Fig. 3. Example loops after rearrangement.

Array Copying Statement rearrangement is unable to legalize the dependency whose source statement and target statement are the same ($\Delta s = 0$). To solve this problem, we need to separate write operation and read operations on the same array into different statements. Therefore, we should perform array copying [10] before performing statement rearrangement to ensure more dependencies can be legalized.

We use a buffer array to replace the left side of a statement and create a following statement that reads data from the buffer array and writes it to original array. For instance, the example loops after performing array copying on statement $S2$ are shown in Figure 4. The original statement $S2$ is divided into statements $S2$ and $S3$. Dependency $\boldsymbol{d}_2$ becomes $(0, 1, 1)$ and it can be legalized by further performing statement rearrangement.

```
for ...
    for ...
    {
        L1: for(i=1;i<=4;i++)
        {
            S1: B[i]=A[i+1];
            S2: buf[i]=A[i+1];
            S3: A[i]=buf[i];
        }
        L2: for(i=1;i<=4;i++)
            S4: C[i]=A[i];
    }
```

Fig. 4. Example loops after array copying.

However, to preserve dependency, array copying should not be performed on the statement whose read operation depends on its written value in previous iteration. The dependence vector of the corresponding dependency is like

$$\boldsymbol{d}(0, \Delta i, 0), \Delta i < 0.$$

The expected speedup of array copying should also be considered, since array copying doubles the complexity of computation to gain profit from vectorization.

The expected speedup of array copying is roughly calculated by

$$speedup_{array\_copying} = \frac{speedup_{vectorization}}{2} \tag{6}$$

where the $speedup_{vectorization}$ is estimated by compiler. If $speedup_{array\_copying} \leq 1$, array copying will be not launched.

Algorithm Guiding Loop Transformations The procedure of performing loop transformations can be divided into two phases — array copying and statement rearrangement. The detail of the procedure is shown in Algorithm 1.

Algorithm 1: SIMD-aware Loop Transformation

Input:
Plain LBM kernel codes
Output:
Transformed codes

```
Scan input codes and generate dependence table D
while ∃d(0, Δi, 0) ∈ D and Δi > 0 do
    Try to find the first dependency like d(0, Δi, 0) ∈ D, Δi > 0 and its
    source/target statement S
    if such S is found then
        if speedup_array_copying(S) > 1 then
            /* Perform array copying */
            Use a buffer array to replace the written array of S and create
            a new statement where the buffer array is read next to S
            Update D
        end
    end
end
while ∃d(0, Δi, Δs) ∈ D and Δi ≠ 0 and Δs ≠ 0 do
    Try to find the first dependency like d(0, Δi, Δs) ∈ D, Δi ≠ 0, Δs ≠ 0
    and its target statement S_t
    if such S_t is found then
        if speedup_vectorization(S_t) > 1 then
            /* Perform statement rearrangement */
            Move S_t to the next loop (if the next loop does not exist,
            create a new one)
            Update D
        end
    end
end
```

The input of the algorithm is plain LBM kernel codes and the output is trans-

formed codes. The algorithm is able to legalize most dependencies except dependencies with $\boldsymbol{d}(0, \Delta i, 0)$ where $\Delta i < 0$. Since many loops with illegal dependencies will not be analyzed and further transformed for vectorization by most general compilers, a variety of vectorization opportunities have been missed. Whereas, our algorithm can effectively avoid this case and generate more vectorizable loops. Therefore, we achieve the acceleration of LBM computing by taking full use of vector units on processors. In addition, Algorithm 1 is not only suitable for optimizing LBM, but also suitable for other LBM-like computation.

Algorithm 1 is a polynomia-time solution. Given that there are n statements in a plain code, the temporal complexity of generating a dependence table D, scanning each $\boldsymbol{d}$ in D, updating D, performing array copying and performing statement rearrangement are $T_1 = O(n)$, $T_2 = O(n)$, $T_3 = O(1)$, $T_4 = O(n)$ and $T_5 = O(n)$, respectively. Hence, the total temporal complexity of Algorithm 1 is

$$T = max(T_1, T_2, T_3 \times T_4, T_3 \times T_5) = O(n)$$

which indicates that the algorithm is a polynomial-time solution for SIMD-aware loop transformation problem.

4 Experiments

4.1 Experimental Setup

We carried out experiments on an Intel(R) Xeon(R) Gold 6140 server. The test server has 36 cores with Skylake microarchitecture and it supports the most advanced SIMD instruction set AVX-512. The LBM benchmark is openLBMflow [3] based on LGBK D3Q19 model. We divided the benchmark into 4 different versions: baseline-no-vec (baseline benchmark without auto-vectorization compiler option), baseline-vec (baseline benchmark with auto-vectorization compiler option), Pluto (codes generated by PLuTo and with auto-vectorization compiler option) and SLT (codes transformed by our SLT algorithm with auto-vectorization compiler option). The compiler is Intel C++ Compiler version 19.1.

4.2 Comparison of Ratio of Vectorized Loops

The vectorization reports generated by compiler provide detailed information about the number of loops and the number of auto-vectorized loops by compiler for four tested versions of LBM kernel codes. Based on the information, we calculated the ratio of number of vectorized loops to number of all loops (RVL). Table 2 shows the vectorization information of each version of benchmark.

One-quarter of innermost loops in baseline version of codes is vectorized. That is, even with no optimization, the RVL of baseline-vec reaches 25%. The PLuTo optimizer performs pipeline/wavefront parallelism to legalize loop-carried dependencies, and the RVL of Pluto reaches 50%. Since the polyhedron-based transformations incur uncountable, multiple entries & exits and branched loops, half innermost loops of Pluto can not be vectorized by compiler. As our SLT

Table 2. Vectorization information

Benchmark	# of loops	# of vectorized loops	RVL
baseline-no-vec	4	0	0%
baseline-vec	4	1	25%
Pluto	4	2	50%
SLT	53	52	98%

algorithm created new loops to legalize loop-carried dependencies, the number of loops has been increased. However, the majority of transformed loops are vectorized due to the advantages of our algorithm which maintains the new loops satisfying the auto-vectorization conditions. After all, 98% of loops in the version of SLT codes are automatically vectorized by compiler.

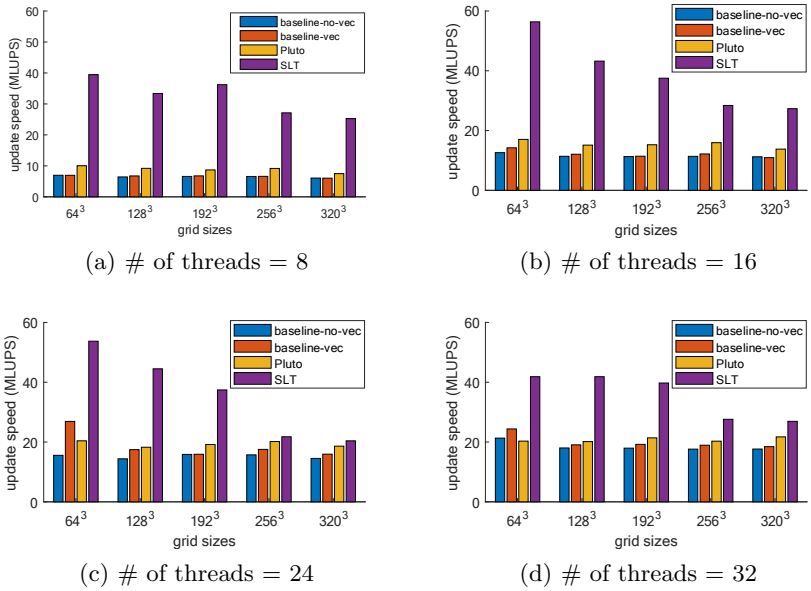
Theoretically, the update speed of LBM kernels is positively correlated with the value of RVL. Therefore, the descending order of four versions of benchmarks sorted by update speed should be: SLT > Pluto > baseline-vec > baseline-no-vec.

4.3 Performance Comparison

We tested the update speed of million lattice updates per second (MLUPS) and the speedup of four versions of LBM codes to evaluate the performance of the proposed SLT algorithm. Different grid sizes and numbers of threads are used in our tests. The grid sizes are $64 \times 64 \times 64$, $128 \times 128 \times 128$, $192 \times 192 \times 192$, $256 \times 256 \times 256$ and $320 \times 320 \times 320$, respectively. And the numbers of threads are 8, 16, 24 and 32, respectively. The vectorization of loops was automatically realized by compiler with AVX-512.

Figure 5 shows the results of update speed comparison. It is clear that the SLT achieves the fastest update speed for all kinds of test codes in Figure 5. The maximum update speed is about 56 MLUPS with grid size of $64 \times 64 \times 64$ and 16 threads in our tests. And the average update speed of SLT is about 36 MLUPS for all test codes, which is 174%, 147% and 120% faster than the average update speed of baseline-no-vec, baseline-vec and Pluto, respectively. According to the RVL values in Section 4.2, the results also indicate that the update speed of LBM codes is positively correlated with the RVL values. We can also observe that the update speed decreases along with the growth of grid sizes in Figure 5. This is because we do not perform any other optimizations on tested codes except automatic vectorization of compiler. And the limitation of memory bandwidth, i.e. the "memory wall" issue, has arisen when grid sizes are growing. However, the solution to the memory optimization is beyond the scope of this paper.

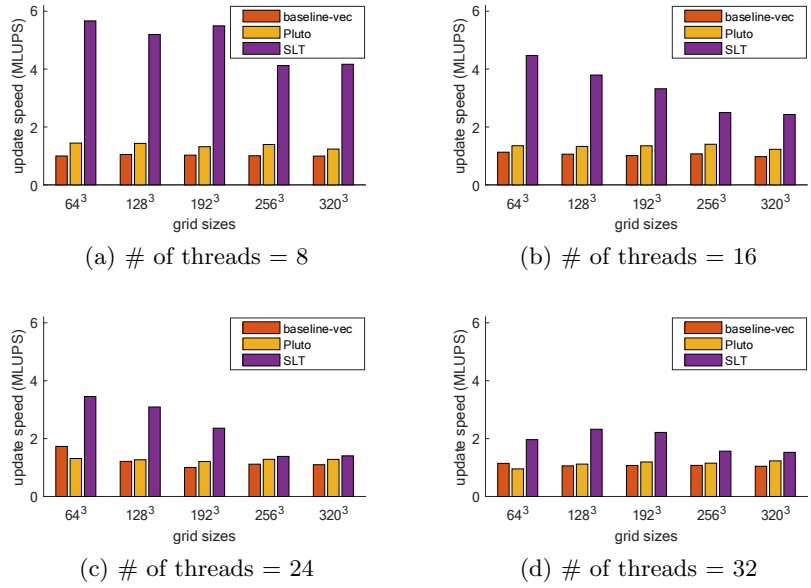
The speedups are calculated by respectively comparing the execution time of baseline-vec, Pluto and SLT with the execution time of baseline-no-vec. Figure 6 shows the results of speedup comparison. The results are similar to the update speed.



(a) # of threads = 8 (b) # of threads = 16

(c) # of threads = 24 (d) # of threads = 32

Fig. 5. Update speeds comparison.



(a) # of threads = 8 (b) # of threads = 16

(c) # of threads = 24 (d) # of threads = 32

Fig. 6. Speedups comparison.

The speedups of SLT are much higher than the speedups of baseline-vec and Pluto for all tested codes. SLT achieves an average speedup of 3.1 in our tests, which is 186% and 145% higher than the average speedups of baseline-vec and Pluto, respectively. The downtrend of speedups is also observed in Figure 6 when grid sizes and the number of threads increase. As explained before, the memory bandwidth becomes the major performance bottleneck and the performance profit from vectorization decreases. Besides, when the number of threads grows, the contention of memory bandwidth exacerbates the problem and leads to a further decline of speedup. However, despite of these factors, the experimental results can demonstrate that our SLT algorithm is effective to accelerate LBM computation by fully exposing vectorizable loops.

5 CONCLUSION

In this paper, we propose a SIMD-aware loop transformation algorithm to accelerate the computation of LBM codes by making much of vectorization. The proposed SLT algorithm is able to identify most potential vectorizable loops that are ignored by general compilers based on a defined dependence table. And it also provides a solution to transform these loops into automatically identifiable vectorizable loops by compilers, which performs array copying and statement rearrangement for LBM kernels. Compared with polyhedron-based loop transformation techniques, SLT algorithm maintains the conditions of vectorization for loops. Experimental results show that SLT algorithm can significantly raise the ratio of number of vectorized loops to number of all loops for LBM kernels. And our algorithm gets better performance than a polyhedral optimizer and the Intel C++ compiler in vectorization. It also indicates that the proposed algorithm should be effective in the acceleration of other LBM-like computations.

In future work, we will combine loop tiling techniques and SLT algorithm to further enhance data locality and reduce the adverse impact of memory bandwidth. We will also apply SLT algorithm to other LBM-like applications to mine the potential of vectorization power on modern processors.

Acknowledgement This work was supported in part by the National Key Research and Development Program of China under Grant No. 2016YFB0201800, the National Natural Science Foundation of China under Grant No. 91630206 and 61672423.

References

1. Aos and soa, accessed 1 April 2019. https://en.wikipedia.org/wiki/AOS_and_SOA
2. Intel c++ compiler 19.0 developer guide and reference, accessed 6 June 2019. https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-vectorization-and-loops
3. openlbmflow, accessed June 15, 2019. https://sourceforge.net/projects/lbmflow

4. Pluto - an automatic parallelizer and locality optimizer for affine loop nests, accessed 7 June 2019. pluto-compiler.sourceforge.net
5. Acharya, A., Bondhugula, U.: PLUTO+: near-complete modeling of affine transformations for parallelism and locality. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015. pp. 54–64 (2015)
6. Bernstein, A.J.: Analysis of programs for parallel processing. IEEE Transactions on Electronic Computers (5), 757–763 (1966)
7. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral program optimization system. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (Jun 2008)
8. Chen, S., Doolen, G.D.: Lattice boltzmann method for fluid flows. Annual review of fluid mechanics **30**(1), 329–364 (1998)
9. Devan, P.S., Kamat, R.: A review–loop dependence analysis for parallelizing compiler. International Journal of Computer Science and Information Technologies **5**(3), 4038–4046 (2014)
10. Di, P., Ye, D., Su, Y., Sui, Y., Xue, J.: Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus. In: 2012 41st International Conference on Parallel Processing. pp. 350–359. IEEE (2012)
11. Du, X., Kuang, D., Ye, Y., Li, X., Chen, M., Du, Y., Wu, W.: Comparative study of distributed deep learning tools on supercomputers. In: International Conference on Algorithms and Architectures for Parallel Processing. pp. 122–137. Springer (2018)
12. Feautrier, P.: Some efficient solutions to the affine scheduling problem. i. one-dimensional time. International journal of parallel programming **21**(5), 313–347 (1992)
13. Feautrier, P.: Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. International journal of parallel programming **21**(6), 389–420 (1992)
14. Feng, Y., Tang, J., Wang, C., Xie, J.: Cuapss: A hybrid cuda solution for allpairs similarity search. In: International Conference on Algorithms and Architectures for Parallel Processing. pp. 421–436. Springer (2018)
15. Kong, M., Veras, R., Stock, K., Franchetti, F., Pouchet, L., Sadayappan, P.: When polyhedral transformations meet SIMD code generation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 127–138 (2013)
16. Krafczyk, M., Tölke, J., Luo, L.S.: Large-eddy simulations with a multiple-relaxation-time lbe model. International Journal of Modern Physics B **17**(01n02), 33–39 (2003)
17. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 201–214. ACM (1997)
18. Liu, S., Zou, N., Cui, Y., Wu, W.: Accelerating the parallelization of lattice boltzmann method by exploiting the temporal locality. In: 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC). pp. 1186–1193. IEEE (2017)
19. Pouchet, L.N.: Interative Optimization in the Polyhedral Model. Ph.D. thesis, University of Paris-Sud 11, Orsay, France (Jan 2010)
20. Qian, Y., d'Humières, D., Lallemand, P.: Lattice bgk models for navier-stokes equation. EPL (Europhysics Letters) **17**(6), 479 (1992)

21. Shanley, T.: Pentium Pro and Pentium II system architecture. Addison-Wesley Professional (1998)
22. Tran, N.P., Lee, M., Choi, D.H.: Memory-efficient parallelization of 3d lattice boltzmann flow solver on a gpu. In: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC). pp. 315–324. IEEE (2015)
23. Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., Rosen, I.: Polyhedral-model guided loop-nest auto-vectorization. In: 2009 18th International Conference on Parallel Architectures and Compilation Techniques. pp. 327–337. IEEE (2009)
24. Xue, J.: Loop tiling for parallelism, vol. 575. Springer Science & Business Media (2012)
25. Zhang, W., Zhang, L., Chen, Y.: Asynchronous parallel dijkstras algorithm on intel xeon phi processor. In: International Conference on Algorithms and Architectures for Parallel Processing. pp. 337–357. Springer (2018)