Karlsruhe Institute of Technology

Seminar Report

Symbolic generation of GPU-based Lattice Boltzmann implementations

Adrian Kummerländer, B.Sc.

July 11, 2019

Supervisor: Dr. Mathias J. Krause

Department of Mathematics

Karlsruhe Institute of Technology

Contents

1 Lattice Boltzmann Method **3**
1.1 Algorithm 6
1.2 Constants of the discrete equilibrium distribution 6
1.3 Theoretical performance 7

2 Symbolic code generation **8**
2.1 Basics of symbolic computation 8
2.2 Symbolic collision step 10
2.2.1 Recovery of moments 10
2.2.2 Derivation of characteristic constants 11
2.2.3 Equilibrium distribution and collision 12
2.3 Common subexpression elimination 14

3 Evaluation **16**

References **19**

1 Lattice Boltzmann Method

Ludwig Eduard Boltzmann's contributions to statistical physics in the form of the Boltzmann equation deliver both the theoretical foundation and the eponym of the Lattice Boltzmann approach to computational fluid dynamics. His equation describes the behaviour of gasses on a mesoscopic scale as a distribution function of a mass of particles moving at a given speed inside a volume.

Definition 1.1 (The Boltzmann equation)**.** *Let $f(x, \xi, t$ be the distribution function of a particle mass at a location $x \in \mathbb{R}^d$ moving at time t with velocity $\xi \in \mathbb{R}^d$. We use ρ to describe the density and $F \in \mathbb{R}^d$ to model any body forces.*
The Boltzmann equation describes the temporal change of the distribution function using the total differential $\Omega(f)$:

$$\Omega(f) = \left(\frac{dt}{dt}\partial_t + \frac{dx}{dt}\partial_x + \frac{d\xi}{dt}\partial_\xi\right) f = \left(\partial_t + \xi\,\partial_x + \frac{F}{\rho}\,\partial_\xi\right) f.$$

This is an advection equation where the term $\partial_t f + \xi \cdot \partial_x f$ models the flux of the particle distribution with velocity ξ. Correspondingly $\frac{F}{\rho} \cdot \partial_\xi f$ represents forces acting on the fluid. The term $\Omega(f)$ models the local redistribution of f caused by collisions between the fluid particles. Thus Ω is commonly referred to as a collision operator*.*

Conservation of momentum and mass are essential requirements to be fulfilled by such an collision operator. The Lattice Boltzmann method we consider for the purposes of this report uses the conventional BGK approximation of the Boltzmann equation without external forces by Bhatnagar, Gross and Krook (see *The Lattice Boltzmann Method: Principles and Practice* [1, Kap. 3.5.3]).

The basic element of this approximation is the BGK operator:

$$\Omega(f) := -\frac{f - f^{\text{eq}}}{\tau}\Delta t$$

This operator relaxes the particle distribution towards a equilibrium distribution f^{eq} at rate τ. We set $\Delta t := 1$ without loss of generality and apply the BGK operator to the Boltzmann equation without external forces to obtain the BGK approximation:

Definition 1.2 (BGK approximation)**.** *Let $\tau \in \mathbb{R}_{\geq 0}$ be a relaxation time and f^{eq} the equilibrium distribution given by the Maxwell-Boltzmann distribution.*

$$(\partial_t + \xi \cdot \nabla_x) f = -\frac{1}{\tau}(f(x, \xi, t) - f^{eq}(x, \xi, t))$$

Up until this point the BGK approximation is defined for arbitrary velocities $\xi \in \mathbb{R}^d$. As we want to implement the LBM on a *finite* computer, this needs to be restricted to a finite set of discrete velocities.

Common sets of such discrete velocities are *D2Q9* in two dimensions and *D3Q19* or *D3Q27* in three dimensions. Note that *D2* encodes the number of dimensions and *Q9* describes the number of velocities.

Definition 1.3 (D2Q9 velocity set)**.**

$$\{\xi_i\}_{i=0}^{8} = \left\{ \begin{pmatrix} -1 \\ -1 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\} = \{-1, 0, 1\}^2$$

Definition 1.4 (D3Q27 velocity set)**.**

$$\{\xi_i\}_{i=0}^{2}6 = \left\{ \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix}, \begin{pmatrix} -1 \\ -1 \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\} = \{-1, 0, 1\}^3$$

We continue by using one of these sets to discretize the BGK approximation with respect to the velocity:

Definition 1.5 (BGK velocity discretization)**.** *Let ξ_i be a microscopic velocity vector as as given by e.g. D2Q9. Let $f_i(x,t) \equiv f(x, \xi_i, t)$ be a shorthand notation of referring to directed distribution functions. It follows that*

$$(\partial_t + \xi_i \cdot \nabla_x) f_i(x,t) = -\frac{1}{\tau}(f_i(x,t) - f_i^{eq}(x,t))$$

is a discretization of the BGK approximation with respect to the fluid velocity.

The discrete equilibrium distribution f_i^{eq} is defined as follows:

Definition 1.6 (Discrete equilibrium distribution)**.** *Let $\rho \in \mathbb{R}_{\geq 0}$ be the density, $u \in \mathbb{R}^d$ the total velocity, ξ_i the ith discrete velocity component, ω_i the weight of this component with respect to the lattice and c_s the lattice speed of sound.*

$$f_i^{eq} = \omega_i \rho \left(1 + \frac{u \cdot \xi_i}{c_s^2} + \frac{(u \cdot \xi_i)^2}{2c_s^4} - \frac{u \cdot u}{2c_s^2} \right)$$

The values of $u = u(x,t)$ and $\rho = \rho(x,t)$ at location x and time t are recovered from the *moments* of the distribution function f_i:

Definition 1.7 (Moments of the distribution function)**.**

$$\rho(x,t) = \sum_{i=0}^{q-1} f_i(x,t) \text{ and } \rho u(x,t) = \sum_{i=0}^{q-1} \xi_i f_i(x,t)$$

To continue we require actual values for the weights ω_i and the lattice speed of sound c_s used by our discrete equilibrium distribution. [1, Eq. 3.60 resp. Tab. 3.3] provides these values for the D2Q9 velocity set:

$$\omega_4 = \frac{4}{9},\ \omega_{1,3,5,7} = \frac{1}{9},\ \omega_{0,2,6,8} = \frac{1}{36},\ c_s = \sqrt{\frac{1}{3}}$$

We are going to investigate a way to recover these constants using a given velocity set in section 1.2. But first, to develop an *implementable* explicit BGK equation we continue by integrating the velocity discretization 1.5:

$$f_i(x+\xi_i, t+1) - f_i(x,t) = \int_0^1 \Omega_i(x+\xi_i s, t+s) ds.$$

Note that $\Omega_i(x,t)$ represents the discrete formulation of the BGK collision operator:

$$\Omega_i(x,t) := -\frac{1}{\tau}(f_i(x,t) - f_i^{\text{eq}}(x,t))$$

It turns out that an exact solution of the integral's right hand side is non-trivial which is why it is only approximated in practice. While there is a wide variety of approaches, we can get away with an application of the basic trapezoidial rule:

$$\begin{aligned} f_i(x+\xi_i, t+1) - f_i(x,t) &= \frac{1}{2}\left(\Omega_i(x,t) + \Omega_i(x+\xi_i, t+1)\right) \\ &= -\frac{1}{2\tau}\left(f_i(x+\xi_i,t+1) + f_i(x,t) - f_i^{\text{eq}}(x+\xi_i,t+1) - f_i^{\text{eq}}(x,t)\right) \end{aligned}$$

The only thing still missing for a solution of this implicit equation is a suitable shift of f_i and τ:

Definition 1.8 (Discrete LBM BGK equation)**.** *Let $\overline{f_i}$ and $\overline{\tau}$ be defined as:*

$$\begin{aligned} \overline{f_i} &:= f_i + \frac{1}{2\tau}(f_i - f_i^{eq}) \\ \overline{\tau} &:= \tau + \frac{1}{2} \end{aligned}$$

Inserting these shifted variables into the result of the trapezoidial rule yields [1, Ch. A.5 with $\Delta t = 1$] the fully discretized LBM BGK equation:

$$\overline{f_i}(x+\xi_i, t+1) = \overline{f_i}(x,t) - \frac{1}{\overline{\tau}}(\overline{f_i}(x,t) - f_i^{eq}(x,t))$$

Interestingly the moments $\overline{f_i}$ of the shifted distribution may be calculated analogously to 1.7:

$$\begin{aligned} \sum_{i=0}^{q-1} \overline{f_i} &= \sum_{i=0}^{q-1} f_i + \frac{1}{2\tau}\sum_{i=0}^{q-1}(f_i - f_i^{\text{eq}}) = \rho \\ \sum_{i=0}^{q-1} \xi_i \overline{f_i} &= \sum_{i=0}^{q-1} \xi_i f_i + \frac{1}{2\tau}\sum_{i=0}^{q-1}(f_i - f_i^{\text{eq}}) = \rho u \end{aligned}$$

It is also worth noting that the only places where we had to explicitly care about the desired spatial dimension $d \in \{2,3\}$ was during the choice of discrete velocities and their associated constants. This will become useful for developing generic symbolic code generators.

1.1 Algorithm

It is common to consider the actual implementation of the discrete LBM BGK equation as separated into a collision and a streaming step.

Definition 1.9 (Collision step)**.** *Relaxation of the distribution function to the locally determined equilibrium distribution according to the BGK collision operator.*

$$f_i^{out}(x,t) = f_i(x,t) - \frac{1}{\tau}(f_i(x,t) - f_i^{eq}(x,t))$$

Definition 1.10 (Streaming step)**.** *Moving the new distributions to the neighbording cells as defined by the corresponding discrete velocity set.*

$$f_i(x+\xi_i, t+1) = f_i^{out}(x,t)$$

An essential observation in this context is, that such a collision step only uses information local to a cell which provides a ideal foundation for parallel processing.

1.2 Constants of the discrete equilibrium distribution

Requirements for determining the lattice specific weight and speed of sound constants are preservation of momentum and mass as well as the demand for *rotational isotropy*.

In general one can use Gauss-Hermite quadrature to obtain the weights for a given discrete velocity set [2]. In fact it is possible to construct the weights for D2Q9 and D3Q27 using only their velocities. Determining the weights for e.g. D3Q19 in this way is also possible but requires additional restrictions.

To elaborate: Following the approach by He and Luo in [2, Sect. B] the derivation of weights ω_i for D2Q9 and D3Q27 uses a third order Gauss-Hermite quadrature scheme. The weight coefficients of the underlying Hermite polynomial on three abscissas are:

$$\eta_{-1} = \frac{\sqrt{\pi}}{6},\ \eta_0 = \frac{2\sqrt{\pi}}{3},\ \eta_1 = \frac{\sqrt{\pi}}{6}$$

This leads to the following formulation of the discrete equilibrium distribution's weights:

$$\omega_i = \frac{1}{\sqrt{\pi^d}} \prod_{j=0}^{d-1} \eta_{(\xi_i)_j} \text{ where } (\xi_i)_j \text{ is the } j\text{-th component of the } i\text{-th velocity.} \tag{1.1}$$

The lattice speed of sound c_s then follows by evaluating the condition [1, Eq. 3.60]:

$$\sum_{i=1}^{q-1} \omega_i (\xi_i)_a (\xi_i)_b = c_s^2 \delta_{a,b}. \tag{1.2}$$

This produces a value of $c_s = \sqrt{1/3}$ for both D2Q9 and D3Q27.

1.3 Theoretical performance

Meaningful evaluation of the performance of a given LBM implementation depends on knowledge about the maximum performance that is theoretically supported by the underlying hardware. A common performance measurement in the context of Lattice Boltzmann Methods is the number of cell updates per second. This number is most often encoded in millions of updates per second:

Definition 1.11 (MLUPS)**.** *Let n_{cells} be the number of simulated cells, $n_{updates}$ the number of processed timesteps and Δt the total time taken for updating n_{cells} cells $n_{updates}$ times.*

$$\text{MLUPS} := \frac{n_{\text{cells}}\ n_{\text{updates}}}{10^6\ \Delta t} \qquad \text{(Mega Lattice Updates Per Second)}$$

It is frequently observed [1, Sec. 13.2.3] that the performance of LBM based simulations is not restricted by the speed at which a computer can *compute* – i.e. perform floating point operations – but rather the speed at which memory can be transferred between the processing units and their shared memory. This supports the introduction of MLUPS as an LBM specific performance measure compared to more common measures such as the possible number of floating point operations FLOPS.

Definition 1.12 (Upper bound of the theoretical performance)**.** *Let B be the maximum memory bandwidth of a given processing unit measured in bytes per second. Let B_{update} be the number of bytes read and written per cell during the collide and stream steps. The maximum theoretical performance in MLUPS may then be estimated using:*

$$\text{MLUPS}_{\text{max}} := \frac{B}{B_{\text{update}} 10^6}.$$

As an example we consider an imaginary processing unit with a memory bandwidth of $B = 10\text{GiB}/s$. The number of bytes accessed per cell depends on the used discrete velocity set as well as the floating point precision. e.g. a D2Q9 collide and stream step on 64-bit floating point values accesses nine eight-byte values two times each. This adds up to 144 bytes which yields a theoretical performance of:

$$\text{MLUPS}_{\text{max}} = \frac{10 \cdot 1024^3}{144 \cdot 10^6} \simeq 74.565 \text{ MLUPS}.$$

i.e. if we implemented a LBM code on such a system and optimized as far as theoretically possible we would expect a maximal performance of about 75 MLUPS. The smaller the difference between actual and theoretical performance the less improvement we can expect from our optimization efforts.

2 Symbolic code generation

While LBM packages such as OpenLB [3] rely primarily on manual implementations of their underlying numerical methods, computer algebra systems promise the possibility of moving parts of the actual derivation of these methods into the program scope. i.e. CAS libraries make it possible to programmatically work with symbolic variables to calculate the symbolic forms of derivatives and integrals as well as to apply arbitrary simplifications and transformations.

This report aims to investigate the possible benefits provided by symbolically formulating a LBM collision step. In this context the main focus will be the impact of common subexpression elimination on the performance of a GPU-based LBM code.

2.1 Basics of symbolic computation

While the development of a productively usable computer algebra system such as SymPy [4] is a distinctly non-trivial undertaking, one can write down a basic starting point in just a few lines of code. This is in fact what we do in listing 1 using the Prolog programming language.

```
% derivative of x is 1
diff(X, X, 1) :- !. % cut to prevent further rule applications

% derivative of a constant is 0
diff(C, X, 0) :- number(C).

% sum rule
diff(U+V, X, D) :- diff(U, X, DU), diff(V, X, DV), D = DU + DV.
diff(U-V, X, D) :- diff(U, X, DU), diff(V, X, DV), D = DU - DV.

% product rule
diff(U*V, X, U*DV+V*DU) :- diff(U, X, DU), diff(V, X, DV).

% quotient rule
diff(U/V, X, (DU*V-DV*U)/(V*V)) :- diff(U, X, DU), diff(V, X, DV).

% some special derivatives
diff(sin(U), X,   DU*cos(U))  :- diff(U, X, DU).
diff(cos(U), X, -(DU*sin(U))) :- diff(U, X, DU).
diff(log(U), X, DU/U) :- diff(U, X, DU).
```

Listing 1: Basic symbolic differentiator in Prolog

As we can see Prolog's integrated tree unification algorithm [5] makes it language that is well suited to implementing automatic manipulations of symbolic expressions – in this case a program that allows us to calculate the symbolic derivative of a given term. As our basic implementation lacks any advanced simplification steps the results can quickly

become quite verbose. This is already noticeable for the example invokations given in listing 2. Ignoring that, this approach of representing the symbolic forms as trees on which the common derivation rules are logical statements illustrates the very concept that is also employed by more developed CAS libraries.

```
?- diff(x*x,x,D).
D = x*1+x*1.                                  % ∂_x x^2 = 2x

?- diff(3*sin(x*x),x,D).
D = 3*((x*1+x*1)*cos(x*x))+sin(x*x)*0.        % ∂_x 3sin(x^2) = 6xcos(x^2)

?- diff(x*x*x+2*x+log(x),x,D).
D = x*x*1+x*(x*1+x*1)+(2*1+x*0)+1/x.          % ∂_x x^3 + 2x + log x = 3x^2 + 2 + 1/x
```

Listing 2: Some symbolic derivatives calculated via Listing 1

This is also illustrated by the interface-level similarity between our small playground implementation and listing 3.

```
>>> from sympy import symbols, diff, sin, log
>>> diff(x*x,x)
2*x                                           # ∂_x x^2 = 2x
>>> diff(3*sin(x*x),x)
6*x*cos(x**2)                                 # ∂_x 3sin(x^2) = 6xcos(x^2)
>>> diff(x**3+2*x+log(x),x)
3*x**2 + 2 + 1/x                              # ∂_x x^3 + 2x + log x = 3x^2 + 2 + 1/x
```

Listing 3: Symbolic derivatives in SymPy

2.2 Symbolic collision step

The LBM code developed during the course of this report[1] follows the overall approach taken by the Sailfish LBM code developed at the University of Silesia [6]. i.e. the Python programming language is used to symbolically generate OpenCL kernels in SymPy [4] to be executed with the help of PyOpenCL [7].

The symbolic formulation of a BGK collision kernel can be separated into three distinct parts: Recovery of moments, calculation of the intended equilibrium distribution and the actual relaxation towards the equilibrium.

2.2.1 Recovery of moments

As we saw in the introductory section on the basics of LBM the density and momentum density (resp. velocity) moments are easily recovered using the discrete populations The symbolic translation of definition 1.7 requires a set of placeholder variables representing the discrete population values.

```
rho = symbols('rho')
u   = Matrix(symarray('u', d))

f_next = symarray('f_next', q)
f_curr = symarray('f_curr', q)
```

Listing 4: Framework for moments recovery and streaming

By convention we are going to calculate the moments based on the `f_curr` variables in Listing 4. The results of the collision step can then be assigned to the `f_next` variables. Streaming is easily implemented by changing the memory locations assigned to these variables in the generated code.

```
moments = [ Eq(rho, sum(f_curr)) ]
for i, u_i in enumerate(u):
    moments.append(
        Eq(u_i,
           sum([ (c_j*f_curr[j])[i] for j, c_j in enumerate(c) ]) / sum(f_curr)))
```

$$
\begin{aligned}
\rho &:= f_{curr0} + f_{curr1} + f_{curr2} + f_{curr3} + f_{curr4} + f_{curr5} + f_{curr6} + f_{curr7} + f_{curr8} \\
u_0 &:= \frac{-f_{curr0} - f_{curr1} - f_{curr2} + f_{curr6} + f_{curr7} + f_{curr8}}{f_{curr0} + f_{curr1} + f_{curr2} + f_{curr3} + f_{curr4} + f_{curr5} + f_{curr6} + f_{curr7} + f_{curr8}} \\
u_1 &:= \frac{-f_{curr0} + f_{curr2} - f_{curr3} + f_{curr5} - f_{curr6} + f_{curr8}}{f_{curr0} + f_{curr1} + f_{curr2} + f_{curr3} + f_{curr4} + f_{curr5} + f_{curr6} + f_{curr7} + f_{curr8}}
\end{aligned}
$$

Listing 5: Recovery of moments using the current populations

[1] Source available at `https://code.kummerlaender.eu/symlbm_playground`

2.2.2 Derivation of characteristic constants

While both OpenLB and Sailfish rely on explicit definitions of discrete velocities, weights and lattice speed of sound as the foundation for their respective equilibrium implementations, the observations in section 1.2 allow us to reduce the definition of e.g. a D3Q27 lattice to more or less a single line of code in listing 6.

```
d = 3
q = 27
c = [ Matrix(x) for x in product([-1,0,1], repeat=d) ]
```

Listing 6: Definition of the D3Q27 model

Listing 7 contains the SymPy translation of (1.1) and (1.2) that allow recovery of weights and speed of sound using only a given discrete velocity set.

```
# determine weights of a d-dimensional LBM model on velocity set c
def weights(d, c):
    _, omegas = gauss_hermite(3)
    return list(map(lambda c_i: Mul(*[ omegas[1+c_i[iDim]] for iDim in range(0,d) ])
    ↪  / pi**(d/2), c))

# determine lattice speed of sound using directions and their weights
def c_s(d, c, w):
    speeds = set([ sqrt(sum([ w[i] * c_i[j]**2 for i, c_i in enumerate(c) ])) for j
    ↪  in range(0,d) ])
    assert len(speeds) == 1 # verify isotropy
    return speeds.pop()
```

Listing 7: Determination of LBM characteristics

Lattice Boltzmann models that do not rely on the full neighborhood of a cell such as D3Q19 may be easily maintained alongside this automatic generation. Note that at least the speed of sound recovery works for all models including D3Q19.

```
d = 3
q = 19
c = [ Matrix(x) for x in [
    ( 0, 1, 1), (-1, 0, 1), ( 0, 0, 1), ( 1, 0, 1), ( 0, -1, 1), (-1, 1, 0), ( 0, 1,
    ↪  0), ( 1, 1, 0), (-1, 0, 0), ( 0, 0, 0), ( 1, 0, 0), (-1,-1, 0), ( 0, -1, 0),
    ↪  ( 1, -1, 0), ( 0, 1,-1), (-1, 0,-1), ( 0, 0,-1), ( 1, 0,-1), ( 0, -1,-1)
]]
w = [Rational(*x) for x in [
    (1,36), (1,36), (1,18), (1,36), (1,36), (1,36), (1,18), (1,36), (1,18), (1,3),
    ↪  (1,18), (1,36), (1,18), (1,36), (1,36), (1,36), (1,18), (1,36), (1,36)
]]
```

Listing 8: Definition of the D3Q19 model

2.2.3 Equilibrium distribution and collision

The previous two sections provided a symbolic formulation of moments and characteristic constants. This allows us to continue by implementing the discrete equilibrium distribution in listing 9.

```
def equilibrium(descriptor):
    rho = symbols('rho')
    u   = Matrix(symarray('u', descriptor.d))

    f_eq = []

    for i, c_i in enumerate(descriptor.c):
        f_eq_i = descriptor.w[i] * rho * ( 1
                                         + c_i.dot(u)    /    descriptor.c_s**2
                                         + c_i.dot(u)**2 / (2*descriptor.c_s**4)
                                         - u.dot(u)      / (2*descriptor.c_s**2) )
        f_eq.append(f_eq_i)

    return f_eq
```

Listing 9: Discrete equilibrium distribution in SymPy

For comparision let us consider non-symbolic definitions written in the C++ language: Listing 10 showcases the corresponding definition in the latest version of OpenLB [3]. Ignoring obvious syntax-level differences these two implementations of the discrete equilibrium distribution look reasonably similar.

```
static T equilibrium(int iPop, T rho, const T u[DESCRIPTOR::d], const T uSqr)
{
  T c_u = T();
  for (int iD=0; iD < DESCRIPTOR::d; ++iD) {
    c_u += descriptors::c<DESCRIPTOR>(iPop,iD)*u[iD];
  }
  return rho
         * descriptors::t<T,DESCRIPTOR>(iPop)
         * ( T{1}
             + descriptors::invCs2<T,DESCRIPTOR>() * c_u
             + descriptors::invCs2<T,DESCRIPTOR>() *
             ↪  descriptors::invCs2<T,DESCRIPTOR>() * T{0.5} * c_u *c_u
             - descriptors::invCs2<T,DESCRIPTOR>() * T{0.5} * uSqr )
         - descriptors::t<T,DESCRIPTOR>(iPop);
}
```

Listing 10: OpenLB's generic discrete equilibrium distribution

However there is one stark difference – while the C++ version calculates and returns the equilibrium by directly executing its compiled code, the SymPy version returns not

a value but the symbolic formulation of how to calculate the equilibrium. This symbolic formulation can then be passed into e.g. a BGK relaxation generator which in turn doesn't just receive an explicit equilibrium value but rather the symbolic instructions for calculating it. If we follow this further we end up with a symbolic formulation of a complete LBM collision step such as the one in figure 1.

This formulation of a BGK collision step is amenable to the application of advanced optimizations such as common subexpression elimination – higher level optimizations of this kind are in general not performed to a comparable degree during C++ compilation.

$$
\begin{aligned}
f_{next0} &:= f_{curr0} + \frac{-f_{curr0} + \frac{\rho\left(-\frac{3u_0^2}{2}-\frac{3u_1^2}{2}+3u_1-\frac{3u_2^2}{2}+3u_2+\frac{9(u_1+u_2)^2}{2}+1\right)}{36}}{\tau},\\
f_{next1} &:= f_{curr1} + \frac{-f_{curr1} + \frac{\rho\left(-\frac{3u_0^2}{2}-3u_0-\frac{3u_1^2}{2}-\frac{3u_2^2}{2}+3u_2+\frac{9(-u_0+u_2)^2}{2}+1\right)}{36}}{\tau},\\
f_{next2} &:= f_{curr2} + \frac{-f_{curr2} + \frac{\rho\left(-\frac{3u_0^2}{2}-\frac{3u_1^2}{2}+3u_2^2+3u_2+1\right)}{18}}{\tau},\\
&\dots\\
f_{next16} &:= f_{curr16} + \frac{-f_{curr16} + \frac{\rho\left(-\frac{3u_0^2}{2}-\frac{3u_1^2}{2}+3u_2^2-3u_2+1\right)}{18}}{\tau},\\
f_{next17} &:= f_{curr17} + \frac{-f_{curr17} + \frac{\rho\left(-\frac{3u_0^2}{2}+3u_0-\frac{3u_1^2}{2}-\frac{3u_2^2}{2}-3u_2+\frac{9(u_0-u_2)^2}{2}+1\right)}{36}}{\tau},\\
f_{next18} &:= f_{curr18} + \frac{-f_{curr18} + \frac{\rho\left(-\frac{3u_0^2}{2}-\frac{3u_1^2}{2}-3u_1-\frac{3u_2^2}{2}-3u_2+\frac{9(-u_1-u_2)^2}{2}+1\right)}{36}}{\tau}
\end{aligned}
$$

Figure 1: Symbolic BGK collision step generated and printed as LaTeX by SymPy

Note that plain symbolic simplifications such as the removal of terms that cancel each other out is mostly performed transparently in the background. The last remaining step is to generate executable OpenCL kernel code from our symbolic collision formulation. While SymPy provides a specialized code generation subsystem, the discussed code utilizes the straight forward way of embedding individually converted[2] expressions into a handwritten kernel template.

[2]SymPy provides a `ccode` function for rendering individual symbolic expressions into OpenCL-compatible instructions of the C programming language

2.3 Common subexpression elimination

One benefit of formulating numerical methods in a symbolic fashion as an intermediate step is that repeated arithmetic expressions are easy to identify. It can make sense to extract and reuse such *common subexpressions* in order to improve a program's overall performance by reducing the number of expensive floating point operations.

```
//
double x1 = a*b + c;
double x2 = a*b + d;
```

No CSE

```
double s1 = a*b;
double x1 = s1 + c;
double x2 = s1 + d;
```

CSE

Listing 11: Simple example for common subexpression elimination

Modern optimizing C++ compilers such as GCC and LLVM / Clang already try to perform such eliminations [8, Sec. 6.1]. However safely performing transformations of this nature on a C++ expression tree is very difficult due to the language's high complexity and the possibility of side effects. In comparison a symbolic expression tree as it is maintained by computer algebra systems is much simpler as there are fewer special cases and no side effects. In addition to that mathematical expressions offer the powerful concept of equivalence transformations whereas equivalence is problematic for both C++ code and the resulting machine code.

Note that the cost of extracting common subexpressions into helper variables lies in increased memory usage – especially the fastest processor-local registers. As is the case for most optimizations there is a tradeoff to be made between various conflicting options [8, Sec. 6.2]. Extraction of too many subexpressions can lead to excessive register pressure which can force spilling into slower shared memory.

```
>>> from sympy import cse, count_ops
>>> count_ops(collide) # collide is the unoptimized list of assignments in figure 1
459  # 55*ADD + 101*DIV + 19*EQ + 118*MUL + 3*NEG + 69*POW + 94*SUB
>>> collide_opt = cse(collide, optimizations='basic')
>>> count_ops(collide_opt)
215  # 57*ADD + 4*DIV + 19*EQ + 75*MUL + 6*NEG + 12*POW + 42*SUB
```

Listing 12: Common subexpression elimination in SymPy

For the problems considered in the context of this report CSE significantly improved performance when applied to the merged collision and equilibrium expressions while keeping the calculation of moments separate. Table 1 summarizes the impact of CSE on the number of operations in such a real-world kernel. Figure 2 shows an overview of the result of SymPy's cse function applied on a D3Q19 collision expression in listing 12.

Manual common subexpression elimination is common practice [1, Sec. 13.2.1] when developing LBM codes. For instance OpenLB provides hand-optimized collision implementations for various common LB models that provide a performance benefit of about

$$
\begin{aligned}
x_0 &:= 9\left(u_1 + u_2\right)^2 & x_1 &:= 6u_2 & x_2 &:= u_1^2 \\
x_3 &:= 3x_2 & x_4 &:= -x_3 & x_5 &:= u_0^2 \\
x_6 &:= 3x_5 & x_7 &:= -x_6 + 2 & x_8 &:= x_4 + x_7 \\
x_9 &:= x_1 + x_8 & x_{10} &:= u_2^2 & x_{11} &:= 3x_{10} \\
x_{12} &:= -x_{11} & x_{13} &:= 6u_1 & x_{14} &:= x_{12} + x_{13} \\
x_{15} &:= \frac{1}{\tau} & x_{16} &:= \frac{x_{15}}{72} & x_{17} &:= 6u_0 \\
x_{18} &:= -u_0 & x_{19} &:= x_{11} + x_3 - 2 & x_{20} &:= x_{19} + x_6 \\
x_{21} &:= -x_1 + x_{20} & x_{22} &:= 6x_{10} & x_{23} &:= \frac{x_{15}}{36} \\
x_{24} &:= 9\left(u_0 + u_2\right)^2 & x_{25} &:= x_{12} + x_{17} & x_{26} &:= -u_1 \\
x_{27} &:= -x_{13} & x_{28} &:= x_{17} + x_{20} & x_{29} &:= 6x_2 \\
x_{30} &:= 9\left(u_0 + u_1\right)^2 & x_{31} &:= 6x_5 & x_{32} &:= x_6 - 2 \\
x_{33} &:= -x_{17} + x_{20} & x_{34} &:= -u_2 & x_{35} &:= x_1 + x_{20}
\end{aligned}
$$

$$
\begin{aligned}
f_{next0} &= f_{curr0} - x_{16}\left(72 f_{curr0} - \rho\left(x_0 + x_{14} + x_9\right)\right), \\
f_{next1} &= f_{curr1} - x_{16}\left(72 f_{curr1} + \rho\left(x_{17} + x_{21} - 9\left(u_2 + x_{18}\right)^2\right)\right), \\
f_{next2} &= f_{curr2} - x_{23}\left(36 f_{curr2} - \rho\left(x_{22} + x_9\right)\right), \\
&\dots \\
f_{next16} &= f_{curr16} - x_{23}\left(36 f_{curr16} + \rho\left(x_1 - x_{22} + x_3 + x_{32}\right)\right), \\
f_{next17} &= f_{curr17} - x_{16}\left(72 f_{curr17} + \rho\left(x_1 + x_{33} - 9\left(u_0 + x_{34}\right)^2\right)\right), \\
f_{next18} &= f_{curr18} - x_{16}\left(72 f_{curr18} + \rho\left(-x_0 + x_{13} + x_{35}\right)\right)
\end{aligned}
$$

Figure 2: D3Q9 BGK subexpressions as extracted by SymPy

10 percent compared to a generic implementation. As such the ability to automatically perform CSE for any given LB model promises benefits independent of the specifically targeted hardware platform.

Model	No-CSE	CSE	Reduction
D2Q9	109	72	34%
D3Q19	287	152	47%
D3Q27	435	214	50%

Table 1: Impact of CSE on number of operations for the benchmarked collide step

3 Evaluation

We now want to use a basic lid driven cavity setup with velocity boundary conditions to investigate the performance of our symbolically generated OpenCL LBM kernel. The specific example was chosen due to its reasonably simple but still interesting setup. Furthermore boundary conditions are only applied on the outer edges of the simulation domain such that we mostly measure the bulk performance of our GPU code. All simulations were performed on a recent Nvidia Tesla P100 GPU with 3584 cores and 16 GiB of shared memory.

To isolate the performance impact of the implemented CSE optimization a variety of different sizes and thread block layouts was checked. Comparing the resulting performance across various configuration while turning CSE either on or off thus allows us to detect CSE-specific performance benefits by comparing the respective maximum MLUPS values.

GPU	Bandwidth	D2Q9		D3Q19		D3Q27	
		single	double	single	double	single	double
P100	512.6 GiB/s	7242	3719	3528	1787	2502	1262

Table 2: Upper performance limit in MLUPS

Further classification of the measured performance requires knowledge about the upper performance limit as discussed in section 1.3. Note that the bandwidth values used as the foundation for these upper bounds are not the theoretical manufacturer-provided values but throughput measurements performed on real hardware.

CSE	D2Q9		D3Q19		D3Q27	
	single	double	single	double	single	double
No	6957.4	2814.4	2581.8	998.8	1576.4	647.4
Yes	6922.4	3585.0	3420.2	1763.8	2374.6	1259.6

Table 3: Highest measured performance per LBM model in MLUPS

CSE	D2Q9		D3Q19		D3Q27	
	single	double	single	double	single	double
No	96.1%	75.7%	73.2%	55.9%	63.0%	51.3%
Yes	95.6%	96.4%	96.9%	98.7%	94.9%	99.8%

Table 4: Measured performance per LBM model relative to theoretical maximum

As we can see in table 6 the maximum performance of the CSE-optimized LBM code is very good across all tested models. In fact the results leave little room for improvement and reach to up to 99.8% of the theoretic maximum in the case of a CSE-optimized double precision D3Q27 model.

The only configuration where CSE optimizations did not significantly improve the maximal performance but actually slightly decrease it was the single precision D2Q9 model. In general common subexpression elimination seems to yield better results the higher the precision and the larger the number of discrete velocities.

CSE	D2Q9		D3Q19		D3Q27	
	single	double	single	double	single	double
No	3003.0	1384.2	1305.7	549.5	838.1	356.1
Yes	3328.7	2105.8	2040.8	1230.2	1473.9	907.4

Table 5: Average measured performance per LBM model in MLUPS

CSE	D2Q9		D3Q19		D3Q27	
	single	double	single	double	single	double
No	41.5%	37.2%	37.0%	30.8%	33.5%	28.2%
Yes	46.0%	56.6%	57.8%	68.8%	58.9%	71.9%

Table 6: Average performance per LBM model relative to theoretical maximum

To make sure that the maximum performance values are not outliers but fit in with a larger trend we check the average values across all tested sizes and layouts. These values are expected to be significantly lower than the previous values as the thread layout and size are major factors in the overall performance. As we can see in table 6 this comparison confirms the positive performance impact of CSE over all measured models.

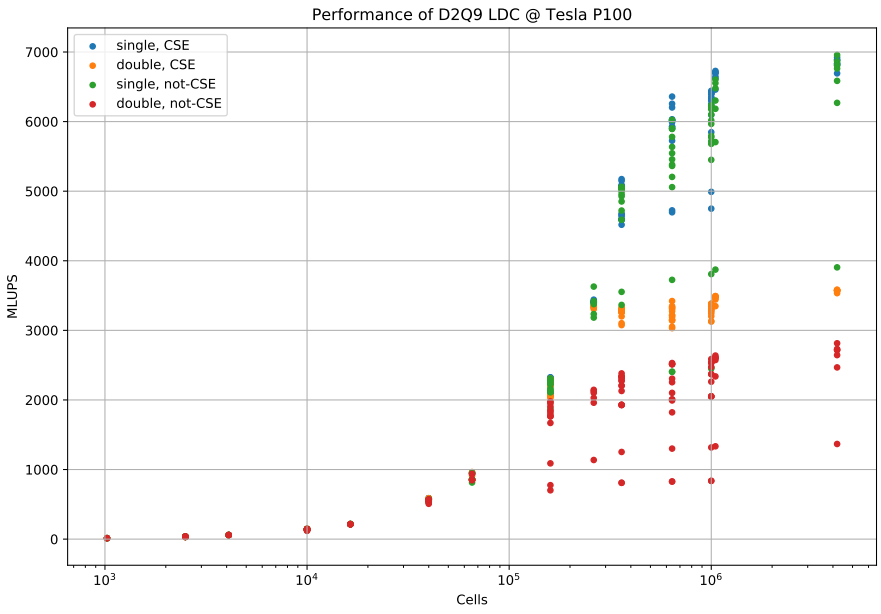

Figure 3: Overview of performance measurements for D2Q9

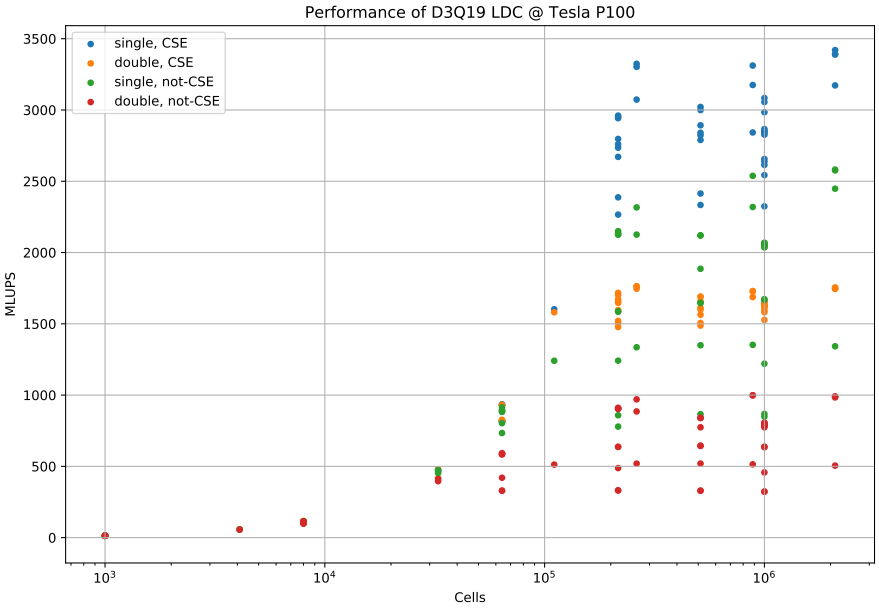

Figure 4: Overview of performance measurements for D3Q19

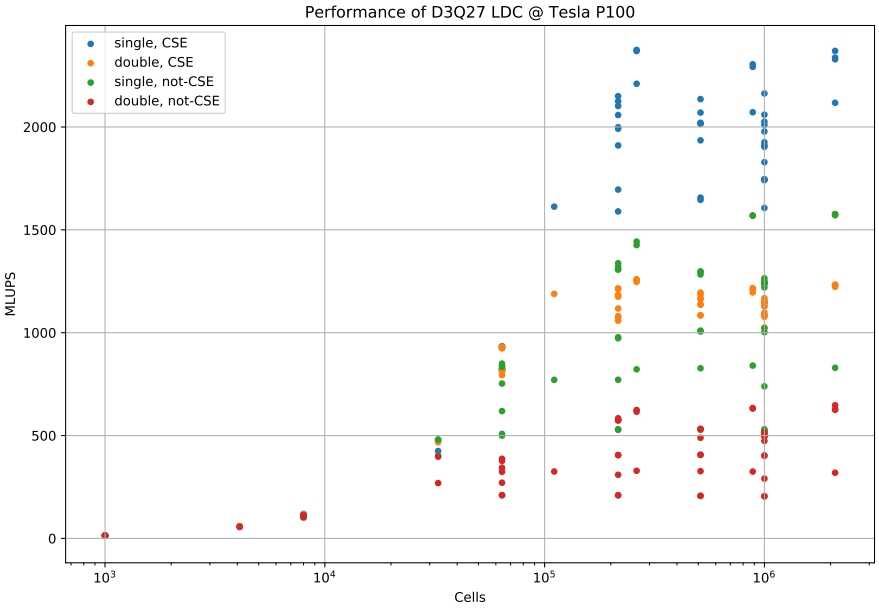

Figure 5: Overview of performance measurements for D3Q27

References

[1] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, and E. M. Viggen. *The Lattice Boltzmann Method: Principles and Practice*. Graduate Texts in Physics. Springer International Publishing, 2017. ISBN: 978-3-319-44647-9.

[2] Xiaoyi He and Li-Shi Luo. "Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation". In: *PHYSICAL REVIEW E* 56 (Dec. 1997), pp. 6811–6817. DOI: 10.1103/PhysRevE.56.6811.

[3] M.J. Krause, S. Avis, D. Dapalo, M. Gaedtke, N. Hafen, M. Haußmann, F. Klemens, A. Kummerländer, M.-L. Maier, A. Mink, J. Ross-Jones, S. Simonis, and R. Trunk. *OpenLB Release 1.3: Open Source Lattice Boltzmann Code*. 2019. URL: http://www.openlb.net/download.

[4] Aaron Meurer et al. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103.

[5] Leon Sterling and Ehud Shapiro. *The Art of Prolog: advanced programming techniques*. 2nd ed. Logic programming. MIT Press, 1994. ISBN: 978-0-262-19338-2.

[6] Michal Januszewski and Marcin Kostur. "Sailfish: a flexible multi-GPU implementation of the lattice Boltzmann method". In: (2013). DOI: 10.1016/j.cpc.2014.04.018. eprint: arXiv:1311.2404.

[7] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation". In: *Parallel Computing* 38.3 (2012), pp. 157–174. ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001.

[8] B.J. Gough and R.M. Stallman. *An Introduction to GCC: For the GNU Compilers GCC and G++*. Network Theory Ltd., 2004. ISBN: 978-0-954-16179-8.