Technical report: How to implement your DdQq dynamics with only q variables per node (instead of 2q)

Jonas Latt
Tufts University
Medford, USA
jonas.latt@gmail.com

June 2007

1 Overview

Naive implementations of lattice Boltzmann (LB) code (or any other nearest-neighbor dynamics) often use unreasonably much memory. For each node of a *DdQq* lattice, they allocate q variables to store the particle distribution functions f_i. Then, they allocate q additional variables to store temporary data. That's because during the streaming step of the dynamics, data is shifted in space. If no temporary memory was available, some still required data would be overwritten during the process. On a lattice of size N^d, the total number of allocated variables is $2q * N^d$. Some work-arounds to this problem are obvious: instead of allocating temporary memory for all lattice sites, one can do it for one line/surface only, covering the extent of nearest-neighbors, and progressively re-use temporary storage space. Storage need is then reduced to $q * (N^d + N^{d-1})$, which is good. The code however becomes complex and unreadable, which is not good.

In this technical report, I suggest a different approach, in which no temporary memory is needed at all (the total number of variables is $q * N^d$). The basic idea is that data is never *copied* to a different memory location, but always *swapped* (*i.e.* exchanged) with another variable. This process is conservative: no information is ever destroyed, and thus, no temporary storage space is required.

The swap-instead-of-copy-approach offers a second advantage: the collision and the streaming step can be executed *synchronously.* This means that a full LB iteration is handled by iterating only once over the N^d lattice nodes. That contrasts again with a naive implementation, in which one would iterate once over all nodes to execute the local collision step, and a second time to implement non-local streaming. Given that memory-access is a bottle-neck on practically all modern computing platforms, this gains a lot of time.

The details of the algorithm are displayed below. It is interesting to point out that the complexity of your algorithm does not increase when you use this approach. Once you understand the philosophy of swap-based programming, you will implement q-variable LB's just as naturally as their $2q$-variable counterparts.

It is also interesting to know that the swap function is commonly used in software engineering for the implementation of conservative processes. A case in point is exception-safe programming in C++. The challenge here is to conserve the state of an object in presence of an exception (an object that changes state should either reach its complete new state, or recover the old state, but never end up in a messy inbetween-state). Modern approaches to this problem suggest to (1) keep the old state, (2) create a new object wich is in the new state and (3) swap the old and the new object. Nothing gets lost, the system is conservative and the program is exception safe (see for example [Scott Meyers: More Effective C++, Addison-Wesley] or [Herb Sutter: Exceptional C++, Addison-Wesley]).

Finally, there exists an open-source C++ code for LB dynamics which makes use of the techniques explained in the present report. You can download it from the internet at `http://www.openlb.org`.

2 Definitions

The discussion is kept general and applies to any lattice, and to all types of commonly used nearest-neigbor dynamics. The ideas are however illustrated on a $D2Q9$ lattice and with BGK dynamics. The essence of this model is explained in any textbook on LB, or in Section 1.3 of my thesis (`http://www.tufts.edu/~jlatt01/preprints/jlatt-thesis.pdf`). Let me simply remind that the dynamics can be decomposed into a collision step as follows:

$$f_i^{out} = (1-\omega) f_i^{in} + \omega f_i^{eq}(\rho, \vec{u}) \qquad \text{for} \quad i = 0 \cdots q. \tag{1}$$

This is followed by a streaming step, during which the value of f_i^{out} is shifted in space, along the lattice velocity $\vec{c}_i$, and copied back to f_i^{in}:

$$f_i^{in}(\vec{r} + \vec{c}_i, t+1) = f_i^{out}(\vec{r}, t). \tag{2}$$

The choice of the names f_i^{in} / f_i^{out} reflects the interpretation of these variables as particle distribution functions prior respectively posterior to the local collision between particles. As usual, the collision step is local, and the streaming step implements nearest-neighbor dynamics.

All illustrative codes are written in an easy, C-stylish C++ notation. The array which holds the distribution functions is written as follows (remember that all illustrations are for $D2Q9$ and for BGK, but the generalization is obvious):

```
double f[lx][ly][q];       // lx * ly * q double-precision matrix (2D case)
```

With the choice of indexes used in this code snippet, the q distributions functions at each lattice site are aligned contiguously in memory. Some people prefer to iterate over the distribution functions with the left-most index, and there are some considerations about efficiency and code architecture involved in this. That's however not an issue here. This report is all about fiddling around with indexes, and we don't care whether they are on the left, on the right or in the middle.

3 Basic code

In this report, a full $D2Q9$-BGK algorithm is developed step-by-step. The present section lists the basic code (computation of density, velocity and local equilbrium) that's used in the following sections:

Computation of ρ

```
void computeRho(double f[q], double& rho) {
    rho = 0.;
    for (int iF=0; iF<q; ++iF) {
        rho += f[iF];
    }
}
```

Computation of $\vec{u}$ and u^2

```
void computeU(double f[q], double& rho, double u[d], double& uSqr) {
    uSqr = 0.;
    for(int iD=0; iD<d; ++iD) {
        u[iD] = 0.;
        for (int iF=0; iF<q; ++iF) {
            u[iD] += f[iF] * c[iF][iD];
        }
        u[iD] /= rho;
        uSqr += u[iD]*u[iD];
    }
}
```

Computation of local equilbrium

```
double fEq(int iF, double rho, double u[d], double uSqr) {
    double c_u = 0.; // scalar product between c_{iF} and u
    for (int iD=0; iD<d; ++iD) {
        c_u += c[iPop][iD] * u[iD];
    }
    return rho*t[iPop]*(1. + 3.*c_u + 4.5*c_u*c_u - 1.5*uSqr);
}
```

4 A naive implementation

The naive LB code, the one that uses $2q$ variables per lattice nodes, is straightforward. Two sets of distribution functions are needed, say `fIn` and `fOut` (other people prefer a `f` *vs.* `fTmp` approach, but that is not fundamentally different):

```
double fIn[lx][ly][q];
double fOut[lx][ly][q];
```

The collision step maps the incoming distribution functions on the outgoing ones (see Section 2):

Collision step

```
void naiveCollision2D(double fIn[lx][ly][q], double fOut[lx][ly][q]) {
    for (int iX=0; iX<lx; ++iX) {
        for (int iY=0; iY<ly; ++iY) {
            double rho, u[d], uSqr;
            computeRho(f, rho);
            computeU(f, rho, u, uSqr);
            for (int iF=0; iF<q; ++iF) {
                fOut[iF] = (1.-omega)*fIn[iF] + omega*fEq(f, iF, rho, u, uSqr);
            }
        }
    }
}
```

The streaming step shifts the `fOut` and copies them back to `fIn`:

```
void naiveStream2D(double fIn[lx][ly][q], double fOut[lx][ly][q]) {
    for (int iX=0; iX<lx; ++iX) {
        for (int iY=0; iY<ly; ++iY) {
            for (int iF=0; iF<q; ++iF) {
                int nextX = iX + c[iF][0];
                int nextY = iY + c[iF][1];
                if (nextX>=0 && nextY>=0 && nextX<lx && nextY<ly) {
                    fIn[nextX][nextY][iF] = fOut[iX][iY][iF];
                }
            }
        }
    }
}
```

5 The swap-trick

In the naive code of the previous section, the variables `fIn` and `fOut` cannot be identical, because during the loop traversal, values which are still needed further ahead get overwritten. A new algorithm is introduced in the present section, in which `fIn` and `fOut` share the same data space, but use different indexes in the space of population functions. For this, let's define the opposite *opposite*(i) of an index $i \in [0 \cdots q-1]$ to respect the following relation:

$$\vec{c}_{opposite(i)} = -\vec{c}_i. \tag{3}$$

The next step is to use a single set of distribution functions `f`. The incoming functions are stored at the location `fIn[i] = f[i]`, and the outgoing functions at `fOut[i] = f[opposite(i)]`.

To keep the discussion simple, indexes are now reorganized in such a way that the rest-particle function has index 0, and that the opposite of an index from the lower half is found by adding $(q-1)/2$ to it: $opposite(i) = i + (q-1)/2$ for $i = 1 \cdots (q-1)/2$.

The above-mentioned storage of incoming and outgoing distribution functions is simply achieved by identifying `fIn` with `fOut` and swapping the `f`'s after collision:

```
void collideAndSwap(double f[lx][ly][q]) {
    const int half = (q-1)/2;
    for (int iX=0; iX<lx; ++iX) {
        for (int iY=0; iY<ly; ++iY) {
            double rho, u[d], uSqr;
            computeRho(f, rho);
            computeU(f, rho, u, uSqr);
            for (int iF=0; iF<q; ++iF) {
                f[iF] *= (1.-omega);
                f[iF] += omega * fEq(f, iF, rho, u, uSqr);
            }
            for (iF=1; iF<=half; ++iF) {
                swap(f[iF], f[iF+half]);
            }
        }
    }
}
```

The swap function is predefined in most programming languages, or can be defined as follows:

```
inline void swap(double& val1, double& val2) {
  double tmp = val1;
  val1 = val2;
  val2 = tmp;
}
```

The full magic of the swap-trick starts with the implementation of the streaming function. With the present definitions, streaming boils down to the following formula:

$$f_i(\vec{r}+\vec{c}_i, t+1) \leftarrow f_{opposite(i)}(\vec{r}, t), \tag{4}$$

which is symmetric. Indeed, it can be reverted by using the properties $i = opposite(opposite(i))$ and Eq. (3):

$$f_{opposite(i)}(\vec{r}, t+1) \leftarrow f_i(\vec{r}+\vec{c}_i, t). \tag{5}$$

Equations (4) and (5) are nothing else than a swap-relation between $f_i(\vec{r}+\vec{c}_i, t+1)$ and $f_{opposite(i)}(\vec{r}, t)$. The streaming function thus becomes:

```
void streamBySwapping2D(double f[lx][ly][q]) {
    const int half = (q-1)/2;
    for (int iX=0; iX<lx; ++iX) {
        for (int iY=0; iY<ly; ++iY) {
            for (int iF=1; iF<=half; ++iF) {
                nextX = iX + c[iF][0];
                nextY = iY + c[iF][1];
                if (nextX>=0 && nextY>=0 && nextX<lx && nextY<ly) {
                    swap(f[iX][iY][iF+half], f[nextX][nextY][iF]);
                }
            }
        }
    }
}
```

That's all you need to implement the full LB dynamics without any temporary data space. Two additional nice properties are included in this algorithm:

1. You don't lose any data on the boundary. Those distribution functions f_i that would "stream out of the domain" stay where they are, which is, in the location $f_{opposite(i)}$. They are allowed to do so, because this location is unused (it corresponds to non-existent distribution functions which would "stream in from outside the domain"). If you are for example implementing a parallel code and need to communicate the boundary values from one lattice to another, you can do so without having to care for extra boundary space: *all the data is always preserved.*

2. You can step through memory in any random order you wish. If you are one of those who (try to) optimize cache-access by stepping through the data block-wise, slice-wise, or even modulo-a-prime-number-wise, feel free to do so. All swaps need to be done at some point, but *order does not matter.*

To complete the sample codes, here is a suggested order of indexes which respects the conditions above:

```
static const int c[9][2] = {
    {0,0},
    {-1,1}, {-1,0}, {-1,-1}, {0,-1},
    {1,-1}, {1,0},  {1,1},   {0,1}
};
```

The corresponding lattice weights are

```
static const double t[9] = { 4./9., 1./36., 1./9., 1./36., 1./9.,
                             1./36., 1./9., 1./36., 1./9. };
```

6 Step through memory only once

When the swap-trick is used, things happen so naturally that you might wonder if you couldn't do the swap-streams directly after collision on each lattice node. You would need only one loop over space for both collision and streaming, instead of two.

The answer is actually yes. You can do that, but only if you step through memory in a specific way. The trick is to only swap with f_i's which have already been computed (*i.e.* which are at time $t+1$). That is, you should only swap backward with respect to your traversal of space indexes, and never forward. That's always posssible, it is just a matter of reordering your indexes properly. Here's the technical requirement the indexes must verify, additionally to those mentioned in the previous section. It is now assumed that the outer loop iterates over space index 0, the next loop over space index 1, followed by (in 3D) space index 2, and that the inner loop iterates over the distribution functions. The condition is that all velocities c_i, for $i = 1..(q-1)/2$ verify the following (you will need to sit down, close your eyes and think through the whole thing to convince yourself of this formula):

In 2D

$$(c[i][0] < 0) \text{ OR } (c[i][0] = 0 \text{ AND } c[i][1] < 0) \tag{6}$$

In 3D

$$\begin{aligned}(c[i][0] < 0) \quad &\text{OR} \quad (c[i][0] = 0 \text{ AND } c[i][1] < 0) \\ &\text{OR} \quad (c[i][0] = 0 \text{ AND } c[i][1] = 0 \text{ AND } c[i][2] < 0).\end{aligned} \tag{7}$$

The $D2Q9$ lattice displayed at the end of the previous section was chosen so as to verify this condition. For your convenience, 3D lattices with appropriate indexing are listed in the appendix.

Once your lattice indexes are properly reordered, the one-loop algorithm is straightforward (the two swaps which were used in the previous section, the one after collision and the one during streaming, can be integrated into a tremendous three-way-swap):

```
void collideAndStream(double f[lx][ly][q]) {
    const int half = (q-1)/2;
    for (int iX=0; iX<lx; ++iX) {
        for (int iY=0; iY<ly; ++iY) {
            double rho, u[d], uSqr;
            computeRho(f, rho);
            computeU(f, rho, u, uSqr);
            for (int iF=0; iF<q; ++iF) {
                f[iF] *= (1.-omega);
                f[iF] += omega * fEq(f, iF, rho, u, uSqr);
            }
            for (int iF=1; iF<=half; ++iF) {
                nextX = iX + c[iF][0];
                nextY = iY + c[iF][1];
                if (nextX>=0 && nextY>=0 && nextX<lx && nextY<ly) {
                    T fTmp              = f[iX][iY][iF];
                    f[iX][iY][iF]       = f[iX][iY][iF+half];
                    f[iX][iY][iF+half]  = f[nextX][nextY][iF];
                    f[nextX][nextY][iF] = fTmp;
                }
            }
        }
    }
}
```

The algorithm you get by colliding and streaming synchronously is likely to be substantially faster than the previous one. That's because computers are not only limited by the speed of computations, but also (and very much so) by the speed of memory accesses. However, the code is a bit more complicated to read and to understand. And you lose the ability to step through space in the order you want. And it simply looks somewhat less elegant. Those things are probably a matter of taste, and the choice between the two algorithms, the one in this section and the one in the previous section, is up to you.

A Lattices with appropriate index ordering

If you define your lattice constants as in the following tables, both algorithms explained in this report will work. Each lattice structure has three types of lattice weights, the weight t_0 corresponding to the zero velocity $\vec{c}_0 = 0$, the weight t_s corresponding to the short velocities and the weight t_l corresponding to the long velocities.

A.1 D2Q9 lattice

$$\begin{array}{llll}
c_s^2 = \frac{1}{3} & & & \\
t_0 = \frac{4}{9} \quad t_s = \frac{1}{9} \quad t_l = \frac{1}{36} & & & \\
\vec{c}_0 = (0,0) & & & \\
\vec{c}_1 = (-1,1) & \vec{c}_2 = (-1,0) & \vec{c}_3 = (-1,-1) & \vec{c}_4 = (0,-1) \\
\vec{c}_5 = (1,-1) & \vec{c}_6 = (1,0) & \vec{c}_7 = (1,1) & \vec{c}_8 = (0,1)
\end{array}$$

A.2 D3Q15 lattice

$$\begin{array}{llll}
c_s^2 = \frac{1}{3} & & & \\
t_0 = \frac{2}{9} \quad t_s = \frac{1}{9} \quad t_l = \frac{1}{72} & & & \\
\vec{c}_0 = (0,0,0) & & & \\
\vec{c}_1 = (-1,0,0) & \vec{c}_2 = (0,-1,0) & \vec{c}_3 = (0,0,-1) & \\
\vec{c}_4 = (-1,-1,-1) & \vec{c}_5 = (-1,-1,1) & \vec{c}_6 = (-1,1,-1) & \vec{c}_7 = (-1,1,1) \\
\vec{c}_8 = (1,0,0) & \vec{c}_9 = (0,1,0) & \vec{c}_{10} = (0,0,1) & \\
\vec{c}_{11} = (1,1,1) & \vec{c}_{12} = (1,1,-1) & \vec{c}_{13} = (1,-1,1) & \vec{c}_{14} = (1,-1,-1)
\end{array}$$

A.3 D3Q19 lattice

$$
\begin{array}{lll}
c_s^2 = \frac{1}{3} & & \\
t_0 = \frac{1}{3} \quad t_s = \frac{1}{18} \quad t_l = \frac{1}{36} & & \\
\vec{c}_0 = (0,0,0) & & \\
\vec{c}_1 = (-1,0,0) & \vec{c}_2 = (0,-1,0) & \vec{c}_3 = (0,0,-1) \\
\vec{c}_4 = (-1,-1,0) & \vec{c}_5 = (-1,1,0) & \vec{c}_6 = (-1,0,-1) \\
\vec{c}_7 = (-1,0,1) & \vec{c}_8 = (0,-1,-1) & \vec{c}_9 = (0,-1,1) \\
\vec{c}_{10} = (1,0,0) & \vec{c}_{11} = (0,1,0) & \vec{c}_{12} = (0,0,1) \\
\vec{c}_{13} = (1,1,0) & \vec{c}_{14} = (1,-1,0) & \vec{c}_{15} = (1,0,1) \\
\vec{c}_{16} = (1,0,-1) & \vec{c}_{17} = (0,1,1) & \vec{c}_{18} = (0,1,-1)
\end{array}
$$

A.4 D3Q27 lattice

The D3Q27 lattice uses lattice vecors of three different lengths: the short ones, such as $(1,0,0)$ with weight t_s, the medium ones like $(1,1,0)$ with weight t_m and the long ones like $(1,1,1)$ with weight t_l.

$$
\begin{array}{llll}
c_s^2 = \frac{1}{3} & & & \\
t_0 = \frac{8}{27} \quad t_s = \frac{2}{27} \quad t_m = \frac{1}{54} \quad t_l = \frac{1}{216} & & & \\
\vec{c}_0 = (0,0,0) & & & \\
\vec{c}_1 = (-1,0,0) & \vec{c}_2 = (0,-1,0) & \vec{c}_3 = (0,0,-1) & \\
\vec{c}_4 = (-1,-1,0) & \vec{c}_5 = (-1,1,0) & \vec{c}_6 = (-1,0,-1) & \\
\vec{c}_7 = (-1,0,1) & \vec{c}_8 = (0,-1,-1) & \vec{c}_9 = (0,-1,1) & \\
\vec{c}_{10} = (-1,-1,-1) & \vec{c}_{11} = (-1,-1,1) & \vec{c}_{12} = (-1,1,-1) & \vec{c}_{13} = (-1,1,1) \\
\vec{c}_{14} = (1,0,0) & \vec{c}_{15} = (0,1,0) & \vec{c}_{16} = (0,0,1) & \\
\vec{c}_{17} = (1,1,0) & \vec{c}_{18} = (1,-1,0) & \vec{c}_{19} = (1,0,1) & \\
\vec{c}_{20} = (1,0,-1) & \vec{c}_{21} = (0,1,1) & \vec{c}_{22} = (0,1,-1) & \\
\vec{c}_{23} = (1,1,1) & \vec{c}_{24} = (1,1,-1) & \vec{c}_{25} = (1,-1,1) & \vec{c}_{26} = (1,-1,-1)
\end{array}
$$