We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900
Open access books available

185,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



Chapter 9

Convolution Kernel for Fast CPU/GPU Computation of 2D/3D Isotropic Gradients on a Square/Cubic Lattice

Sébastien Leclaire, Maud El-Hachem and Marcelo Reggio

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/46455

1. Introduction

The design of discrete operators or filters for the calculation of gradients is a classical topic in scientific computing. Typical applications are gradient reconstruction in computational fluid dynamics, edge detection in computer graphics and biomedical imaging, and phase boundary definition in the modeling of multiphase flows.

Edge detection, which is widely performed in image analysis, is an operation that requires gradient calculation. Commonly used edge detection methods are *Canny*, *Prewitt*, *Roberts* and *Sobel*, which can be found in MATLAB's platform. In this field, edge detection techniques rely on the application of convolution masks to provide a filter or kernel to calculate gradients in two perpendicular directions. A threshold is then applied to obtain an edge shape.

For multiphase flows, an edge or contour corresponds to the interface between the fluids. In this respect, traditional gradient calculation methods based on 1D edge detection are not necessarily suited for the underlying physics, because there is no direction in which the gradients of the phase contours tend to evolve over time. As a result, definition of the geometric progress of the interface requires many gradient estimation computations, as is the case in moving and deforming bubbles or droplets, for example. Although it can still be a viable tool, it is clear that the 1D-based method is becoming less useful for simulating these phenomena, which are not, in general, biased toward any particular direction.

To address this issue, we present an efficient computational method for obtaining discrete isotropic gradients that was previously applied to simulate two-phase flows within the lattice Boltzman framework [1, 2]. This "omnidirectional" approach makes it possible to improve the limitations inherent in handling high density ratios between the phases and to significantly reduce spurious currents at the interface. The method is based on a filter which is generally not split along any direction, and there is no need to make the assumption of a continuous filter to reach isotropy, as done by [3]. We also believe that optimal or maximal isotropy can

INTECH
open science | open minds

©2012 Reggio et al., licensee InTech. This is an open access chapter distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

only be reached with a discrete filter when the error terms of Taylor's series expansion are isotropic, as explained in detail by [1, 2].

Below, we describe isotropic and anisotropic discretizations that will and will not conserve the isotropic property of the differentiated function respectively. This is followed by a description of how convolution can be used to reduce computer time in the gradient calculation. We then present details of the MATLAB implementation of these ideas, along with speedup comparisons of convolution performed on a single core of an Intel® Core i7-970 processor and on an Nvidia® GeForce GTX 580 GPU using the Jacket plugin for MATLAB developed by AccelerEyes®. The GPU with the Jacket plugin for MATLAB speeds up gradient computation by a factor of up to $138x$ in the more challenging case. Our study ends with an example of multiphase flow simulation conducted using the lattice Boltzmann method. Finally, we provide useful stencils and weights (or kernels) to yield isotropic gradients in two and three dimensions.

2. Gradient discretization

Let us define the real scalar function $F(x, y) \in \Re$. There are various methods for calculating the derivative of a function, and one way of doing so is to apply a finite difference approximation. If both the horizontal and vertical lattice spacings are h in length, a simple procedure for evaluating the gradient of this function is as follows:

$$\frac{\partial F}{\partial x} \approx \frac{1}{6h^2} \sum_i F(x + c_i^x, y + c_i^y) c_i^x \tag{1}$$

$$\frac{\partial F}{\partial y} \approx \frac{1}{6h^2} \sum_i F(x + c_i^x, y + c_i^y) c_i^y \tag{2}$$

with

$$c^x = [0, h, h, h, 0, -h, -h, -h] \tag{3}$$

$$c^y = [h, h, 0, -h, -h, -h, 0, h] \tag{4}$$

This finite difference discretization is very similar to Prewitt's operator/kernel [4], which is used in image processing. Note that $h = 1$ in this field, and that the application of Prewitt's operator results in a vector that points in the same direction as the finite difference gradient, but with a norm 6 times larger. The application of Prewitt's operator to an image can be computed very quickly using a convolution product. We address the topic of convolution product later in section (3), but first, let us analyze the isotropy property of the previous gradient discretization.

2.1. Anisotropic discretization

As in Ref. [2] and without loss of generality, the function F is expressed using a 2D Taylor series expansion around the zero vector:

$$\begin{aligned} F(x,y) =& F(0,0) + x\frac{\partial F}{\partial x}\bigg|_{(0,0)} + y\frac{\partial F}{\partial y}\bigg|_{(0,0)} + \frac{1}{2}x^2\frac{\partial^2 F}{\partial x^2}\bigg|_{(0,0)} + xy\frac{\partial^2 F}{\partial x \partial y}\bigg|_{(0,0)} \\ &+ \frac{1}{2}y^2\frac{\partial^2 F}{\partial y^2}\bigg|_{(0,0)} + \frac{1}{6}x^3\frac{\partial^3 F}{\partial x^3}\bigg|_{(0,0)} + \frac{1}{2}x^2 y\frac{\partial^3 F}{\partial x^2 \partial y}\bigg|_{(0,0)} \\ &+ \frac{1}{2}xy^2\frac{\partial^3 F}{\partial x \partial y^2}\bigg|_{(0,0)} + \frac{1}{6}y^3\frac{\partial^3 F}{\partial y^3}\bigg|_{(0,0)} + O(x^n y^m) \end{aligned}$$

with $n + m > 3$.

To calculate the gradient in the x direction, the following stencil values can be taken: $F(h,h)$, $F(h,0)$, $F(h,-h)$, $F(-h,-h)$, $F(-h,0)$, and $F(-h,h)$. When these stencil values are combined, as in the case of the gradient approximation in Eqs. (1) and (2), an exact expression for the gradient in the x direction at $(0,0)$ is obtained:

$$\begin{aligned} \frac{\partial F}{\partial x}\bigg|_{(0,0)} =& \frac{1}{6h}\Big(F(h,h) + F(h,0) + F(h,-h) - F(-h,-h) - F(-h,0) - F(-h,h)\Big) \\ &- \frac{h^2}{6}\frac{\partial^3 F}{\partial x^3}\bigg|_{(0,0)} - \frac{h^2}{3}\frac{\partial^3 F}{\partial x^2 \partial y}\bigg|_{(0,0)} + O(h^3) \end{aligned} \tag{5}$$

A similar expression is found for the y direction:

$$\begin{aligned} \frac{\partial F}{\partial y}\bigg|_{(0,0)} =& \frac{1}{6h}\Big(F(-h,h) + F(0,h) + F(h,h) - F(h,-h) - F(0,-h) - F(-h,-h)\Big) \\ &- \frac{h^2}{6}\frac{\partial^3 F}{\partial y^3}\bigg|_{(0,0)} - \frac{h^2}{3}\frac{\partial^3 F}{\partial x \partial y^2}\bigg|_{(0,0)} + O(h^3) \end{aligned} \tag{6}$$

In the gradients of Eqs. (5) and (6), the leading $O(h^2)$ differential operator of the error term can be written in vector form:

$$\vec{E}^{(2)} = -\frac{h^2}{6}\left[\frac{\partial^3}{\partial x^3} + 2\frac{\partial^3}{\partial x^2 \partial y}, \frac{\partial^3}{\partial y^3} + 2\frac{\partial^3}{\partial x \partial y^2}\right] \tag{7}$$

Using the following transformation from a Cartesian to a polar partial derivative operator:

$$\frac{\partial}{\partial x} = \cos(\theta)\frac{\partial}{\partial r} - \frac{1}{r}\sin(\theta)\frac{\partial}{\partial \theta} \tag{8}$$

$$\frac{\partial}{\partial y} = \sin(\theta)\frac{\partial}{\partial r} + \frac{1}{r}\cos(\theta)\frac{\partial}{\partial \theta} \tag{9}$$

And, by supposing that $F = F(r)$ is rotationally invariant, it is possible to show (after a lengthy algebraic manipulation) that the vector that results when the differential operator $\vec{E}^{(2)}$ is applied to $F(r)$ will have a Euclidean norm that is a function of θ and r, except if $F(r)$ is a constant. This result, that is, the norm $||\vec{E}^{(2)}F(r)|| \equiv f(r,\theta)$, is an equation that takes up almost a page, and so is not presented here. Let us note, however, that this expression can easily be obtained using symbolic mathematics software. For a vector function

to be rotationally invariant, it must have an Euclidean norm that is a function of the radius only. Therefore, this gradient approximation is not isotropic to the second order in space, but anisotropic. It is worth noting that the calculated derivatives have a leading error term that is not isotropic, even if the function F is isotropic, i.e. $F = F(r)$ around (0,0). This means that the discrete gradient will not conserve the isotropic property of the differentiated function when the gradient is approximated with this finite difference stencil.

2.2. Isotropic discretization

Taking into consideration the previous anisotropy problem, it is possible to change the weights of the grid points when computing the gradients to make them isotropic, up to the second order in space, by defining gradients in the x and y directions, as follows:

$$\begin{aligned}\left.\frac{\partial F}{\partial x}\right|_{(0,0)} =& \frac{1}{12h}\Big(F(h,h)+4F(h,0)+F(h,-h)-F(-h,-h)-4F(-h,0)-F(-h,h)\Big)\\ &-\frac{h^2}{6}\frac{\partial}{\partial x}\left(\frac{\partial^2 F}{\partial x^2}+\frac{\partial^2 F}{\partial y^2}\right)\Big|_{(0,0)}-\frac{h^4}{72}\frac{\partial}{\partial x}\left(\frac{\partial^4 F}{\partial x^4}+2\frac{\partial^4 F}{\partial x^2\partial y^2}+\frac{\partial^4 F}{\partial y^4}\right)\Big|_{(0,0)}\\ &-\frac{h^4}{180}\frac{\partial^5 F}{\partial x^5}\Big|_{(0,0)}+O(h^5)\end{aligned} \tag{10}$$

$$\begin{aligned}\left.\frac{\partial F}{\partial y}\right|_{(0,0)} =& \frac{1}{12h}\Big(F(-h,h)+4F(0,h)+F(h,h)-F(h,-h)-4F(0,-h)-F(-h,-h)\Big)\\ &-\frac{h^2}{6}\frac{\partial}{\partial y}\left(\frac{\partial^2 F}{\partial x^2}+\frac{\partial^2 F}{\partial y^2}\right)\Big|_{(0,0)}-\frac{h^4}{72}\frac{\partial}{\partial y}\left(\frac{\partial^4 F}{\partial y^4}+2\frac{\partial^4 F}{\partial x^2\partial y^2}+\frac{\partial^4 F}{\partial x^4}\right)\Big|_{(0,0)}\\ &-\frac{h^4}{180}\frac{\partial^5 F}{\partial y^5}\Big|_{(0,0)}+O(h^5)\end{aligned} \tag{11}$$

With this new discretization, the dominant differential operator of the second order error term takes the form:

$$\vec{E}^{(2)} = \left[E_x^{(2)}, E_y^{(2)}\right] = -\frac{h^2}{6}\left[\frac{\partial}{\partial x}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right), \frac{\partial}{\partial y}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)\right] \tag{12}$$

If a gradient has a small dependence on direction, this would imply that the dominant error term has only an axial dependence when the function being derived also only depends on the radius. That is, the operator in Eq. (12) applied on $F(r)$ would lead to a function that depends only on the radius:

$$\vec{E}^{(2)}F(r) \equiv \vec{e}_r f(r) \tag{13}$$

with $\vec{e}_r = [\cos(\theta), \sin(\theta)]$ being the unit radial vector. Using the partial derivative operator transformation of Eqs. (8) and (9), and supposing that $F = F(r)$ around (0,0), the

components $E_x^{(2)}F(r)$ and $E_y^{(2)}F(r)$ are:

$$E_x^{(2)}F(r) = \frac{h^2\cos(\theta)}{6r^2}\left(-\frac{\partial F(r)}{\partial r} + r\frac{\partial^2 F(r)}{\partial r^2} + r^2\frac{\partial^3 F(r)}{\partial r^3}\right) \tag{14}$$

$$E_y^{(2)}F(r) = \frac{h^2\sin(\theta)}{6r^2}\left(-\frac{\partial F(r)}{\partial r} + r\frac{\partial^2 F(r)}{\partial r^2} + r^2\frac{\partial^3 F(r)}{\partial r^3}\right) \tag{15}$$

which can be rewritten in the same form as Eq. (13). Similarly, the first differential operator of the fourth order error term in Eqs. (10) and (11) takes the form:

$$\vec{E}_{iso}^{(4)} = \left[E_{x,iso}^{(4)}, E_{y,iso}^{(4)}\right] = -\frac{h^4}{72}\left[\frac{\partial}{\partial x}\left(\frac{\partial^4}{\partial x^4} + 2\frac{\partial^4}{\partial x^2\partial y^2} + \frac{\partial^4}{\partial y^4}\right), \frac{\partial}{\partial y}\left(\frac{\partial^4}{\partial y^4} + 2\frac{\partial^4}{\partial x^2\partial y^2} + \frac{\partial^4}{\partial x^4}\right)\right] \tag{16}$$

and the associated components in polar coordinates, when applied to a rotationally invariant function $F(r)$ around (0,0), are given by:

$$E_{x,iso}^{(4)}F(r) = \frac{h^4\cos(\theta)}{72}\left(3\frac{\partial F(r)}{\partial r} - 3r\frac{\partial^2 F(r)}{\partial r^2} + 3r^2\frac{\partial^3 F(r)}{\partial r^3} - 2r^3\frac{\partial^4 F(r)}{\partial r^4} - r^4\frac{\partial^5 F(r)}{\partial r^5}\right) \tag{17}$$

$$E_{y,iso}^{(4)}F(r) = \frac{h^4\sin(\theta)}{72}\left(3\frac{\partial F(r)}{\partial r} - 3r\frac{\partial^2 F(r)}{\partial r^2} + 3r^2\frac{\partial^3 F(r)}{\partial r^3} - 2r^3\frac{\partial^4 F(r)}{\partial r^4} - r^4\frac{\partial^5 F(r)}{\partial r^5}\right) \tag{18}$$

which again meets the rotational invariance requirement, and can be rewritten in the same form as given in Eq. (13). The last differential error operator of the fourth order error term in Eqs. (10) and (11) is:

$$\vec{E}_{ani}^{(4)} = -\frac{h^4}{180}\left[\frac{\partial^5}{\partial x^5}, \frac{\partial^5}{\partial y^5}\right] \tag{19}$$

and can be shown to be anisotropic (i.e. f in Eq. 13 would also be a function of θ). Therefore, the error associated with the anisotropy is lower by two orders when compared to the main second order leading term. It is important to point out that both gradient approximations presented so far are second order approximations in space, but only the latter is a second order approximation in space for the isotropy.

In this work, we only consider the gradient approximation of scalar functions over a square or cubic lattice of unit spacing, so we need to take $h = 1$ from now on. In a more general way, the stencil points with the corresponding weights needed to obtain 2D and 3D isotropic gradients was generalized by [1].

3. Convolution

Here, we present the mathematical and computational aspects of convolution. As finite difference discretization and edge detection kernels are very similar, let us return to the mathematical foundations of these techniques. We often give examples involving 2D images, but we could give the same examples and talk about 2D discrete functions. We don't know the exact coding in the MATLAB and Jacket libraries, as they are under license, but we do have a general idea about function algorithms, which is given in the next section.

3.1. Frequential and spatial filtering

The convolution product of two functions $f(x)$ and $g(x)$, defined by Eq. (20), calculates an average function by sliding g all over f. In common language, we say that the function $g(x)$ acts as a filter of the function $f(x)$.

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x)g(x-s)ds \tag{20}$$

If Eq. (20) defines the convolution of two functions in the space domain, it is also equivalent to the point wise product of these functions in the frequency domain. In mathematics, the convolution theorem for Fourier transforms [5, chap. 4] is formalized by the following equation, where $\mathcal{F}$ corresponds to the Fourier transform applied on a function:

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\} \tag{21}$$

The Fourier transform of a function generates the frequency spectrum of a function. This representation can easily be extended to two dimensions, which is more suitable to image or spatial analysis if the image is a function of two spatial variables. In our case, the output of the Fourier transform will generate a 2D function in the frequency domain from the 2D spatial domain. High frequencies will correspond to information varying rapidly in the original function, while low frequencies correspond to slow variations.

By applying the inverse Fourier Transform to both sides of Eq. (21), we obtain a method for calculating the convolution of two functions:

$$f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\} \tag{22}$$

Therefore, there are two possible ways of convoluting two functions: the Fourier transform pipeline in three steps (Fourier transform, frequency spectrum filtering, and Fourier inverse transform), or the application of a stencil in the spatial domain. Indeed, edge detection kernels acts as high pass filters, accentuating high frequency contributions to the image or to the 2D function in the form of edges and details.

3.2. Discrete circular convolution

Knowing that the properties of the Fourier transform also work for a sampled signal, we present the definition of the 2D discrete Fourier transform (2D DFT), where M and N represent the number of samples in each dimension, x and y are the discrete spatial variables, and u and v are the transform or frequency variables:

$$\mathcal{F}\{u,v\} = \frac{1}{MN} \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x,y) e^{-2\pi j(\frac{ux}{N} + \frac{vy}{M})} \tag{23}$$

In addition to having the same properties as the continuous Fourier transform, which are outside the scope of this presentation and can be found in [6], there are two more in 2D DFT that are important: periodicity and separability. Since the discrete signal is sampled from the finite length sequences, its frequency spectrum will be periodic. In the spatial domain, this property allows us to slide the filter from one frontier to the opposite one, and then start again at the frontier where we began. Moreover, its Fourier transform is separable. The 2D DFT

can be processed in two steps: applying 1D DFT on the lines, and then applying 1D DFT on the resulting columns. In the spatial domain, a 2D filter represented by K_1-by-K_2 matrix is separable if it can be obtained by the outer product of two vectors of size K_1 and K_2, knowing that DFT separability does not imply that the matrix kernel is always separable.

As an example, let us take kernelX, the 2D separable kernel in section (4.2):

$$\frac{1}{6}\begin{bmatrix}3\\0\\-3\end{bmatrix}\otimes\frac{1}{6}\begin{bmatrix}1\\4\\1\end{bmatrix}=\frac{1}{36}\begin{bmatrix}3\\0\\-3\end{bmatrix}\begin{bmatrix}1&4&1\end{bmatrix}=\begin{bmatrix}\frac{1}{12}&\frac{1}{3}&\frac{1}{12}\\0&0&0\\-\frac{1}{12}&-\frac{1}{3}&-\frac{1}{12}\end{bmatrix} \tag{24}$$

We can easily deduce the main functionality of the 2D kernel, the gradient component in x that approximates the derivative in x, by calculating the difference between the first and third rows in the neighborhood of a point, instead of the first and third columns, because of the rotation of our axis system. The separated 1D filters give us further information about the kernel function: a smoothing mask is applied on the differentiating mask, in order to reduce noise that could be exaggerated by the first filter.

Now, to explain how to apply a circular convolution mask in the spatial domain, we go back to the definition of convolution for discrete functions, presented in 2D and 3D in Eqs. (25) and (26) respectively, where A is the image and B is the mask. Processing a convolution filter consists of computing the scalar product of the filter weights with the input values within a window of the filter dimensions surrounding each of the output values, after flipping the mask in each dimension, as described in [7, chap. 4].

$$(A*B)(x,y)=\sum_i\sum_j A(i,j)B(x-i,y-j) \tag{25}$$

$$(A*B)(x,y,z)=\sum_i\sum_j\sum_k A(i,j,k)B(x-i,y-j,z-k) \tag{26}$$

In circular convolution, the equation is slightly modified to model periodic shifting. Here we consider a square image N by N:

$$(A\circledast B)(x,y)=\sum_i\sum_j A(i,j)B[(x-i)\mod N,(y-j)\mod N] \tag{27}$$

The values on the image we see in the window of the K-by-K filter are distributed on a periodic surface. The 2D filter is rolled up in the opposite direction and will turn in a clockwise direction. For each output value, the stencil is lined up on the input values, the scalar product is applied, and then the stencil rotates to shift to the next position to compute. Therefore, in the spatial domain, an N-by-N image convoluted with a K-by-K filter requires N^2K^2 multiplications and nearly N^2K^2 additions. The K^2 multiplications or additions can be reduced to $2K$ if the filter is separable, resulting in a total complexity of $O(N^2K)$ for spatial convolution with a separable filter.

In the frequency domain, the 2D discrete Fourier transform and its inverse are computed using the divide-and-conquer algorithm of the Fast Fourier Transform (FFT and IFFT), which reduces the complexity from N^3 multiplications and N^3 additions to $2N^2\log_2(N)$ operations. The same order of complexity is required to compute the FFT of the K-by-K filter: $2K^2\log_2(K)$.

To those computational complexities, the product cost of two complex numbers, which is six operations, has to be added N^2 times. In the frequency domain, the resulting computational complexity of convoluting an N-by-N image by a K-by-K filter would equal $2N^2 \log_2(N) + K^2 \log_2(K) + 6N^2$, or $O(N^2 \log_2(N))$ if $N \ggg K$. Furthermore, the separability property could be used to implement a 2D FFT by the application of 1D FFT along each direction, in turn, with a global transpose between each of the 1D transform, which computationnal complexity is $O(N \log_2(N))$. To decide whether to use the FFT algorithm or spatial convolution, the two complexity functions should be compared to verify that convolution in the spatial domain performs better than FFT for small kernels only.

4. MATLAB

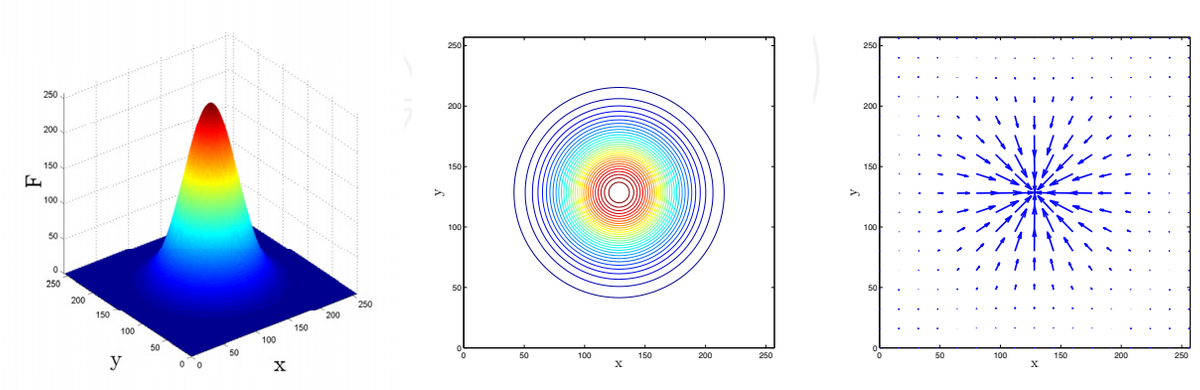
In this section, we present a systematic method that can be applied in MATLAB to calculate the gradient of scalar functions on a square lattice, or in 2D, which is simply the gradient of the images. The 3D case is a straightforward extension of the 2D case. First, let us define a function with which it is possible to test the gradient discretization previously defined in section (2.2):

$$F(x,y) = Ne^{-30\left((\frac{x}{N}-\frac{1}{2})^2+(\frac{y}{N}-\frac{1}{2})^2\right)} \tag{28}$$

For the purpose of this test and for simplicity, we consider only a square image N in width. The exponential function is multiplied by N to make sure that the scaling ratio of the function F is always the same, whatever the value of N. In MATLAB m-code, this function can be defined as:

```
N=256+1;Nx=N;Ny=N;
[X,Y]=ind2sub([Nx Ny],1:Nx*Ny);
X=reshape(X,Nx,Ny)-1;Y=reshape(Y,Nx,Ny)-1;
F=N*exp(-30*(((X)/N-0.5).^2+((Y)/N-0.5).^2));
```

Figures 1(a) and 1(b) show the surface shape and contour of this function.



(a) Surface shape. (b) Contour. (c) Gradient.

Figure 1. Characteristics of the function $F(x,y)$.

4.1. Naive computation of the gradient

When the gradient of a function needs to be computed, it may be tempting to apply Eqs. (10) and (11) directly. This is quite a simple approach, and in MATLAB can be applied as follows:

```
% Stencils and weights

w1=1/3;w2=1/12;
xi=[w1*ones(1,4),w2*ones(1,4)];
d=unique([perms([1 0]);perms([-1 0])],'rows');
d=[d;unique([perms([1 1]);perms([-1 1]);perms([-1 -1])],'rows')];
dxX=(d(d(:,1)~=0,1));dyX=(d(d(:,1)~=0,2));
dxY=(d(d(:,2)~=0,1));dyY=(d(d(:,2)~=0,2));
xiX=(xi(d(:,1)~=0));xiY=(xi(d(:,2)~=0));

% Periodic padding

nPad=max(dxX);
Fpad=padarray(F,[nPad nPad],'circular');

% Naive computation of the gradient

gradFX=0;
for i=1:numel(dxX)
    gradFX=gradFX+xiX(i)*dxX(i).*Fpad(nPad+(1:Nx)+dxX(i),nPad+(1:Ny)+dyX(i));
end

gradFY=0;
for i=1:numel(dyY)
    gradFY=gradFY+xiY(i)*dyY(i).*Fpad(nPad+(1:Nx)+dxY(i),nPad+(1:Ny)+dyY(i));
end
```

However, this is rather a naive and slow implementation for MATLAB. It is far more efficient to evaluate the gradient by using a convolution product, as presented in section (4.2).

4.2. Techniques for calculating the gradient with convolution

Instead of calculating the gradient as previously shown, the use of a convolution product between the functions to differentiate them, and a kernel representing the weights and stencils of the finite difference approximation is more beneficial. For the isotropic discretization of Eqs. (10) and (11), this computation can be performed in MATLAB as follows:

```
% 2D kernel

w1=1/3;w2=1/12;
c01=1*[w2;w1;w2];
kernelX=[-c01,zeros(3,1),c01];
kernelX=-kernelX';
kernelY=kernelX';

% Periodic padding

nPad=(numel(kernelX(:,1))-1)/2;
Fpad=padarray(F,[nPad nPad],'circular');
```

```
% Fast convolution for computing the gradient

gradFX=conv2(Fpad,kernelX,'valid');
gradFY=conv2(Fpad,kernelY,'valid');
```

A useful list of 2D and 3D kernels for calculating isotropic gradients is available in sections (6.1) and (6.2).

4.3. Separable kernel

Sometimes the kernel is separable, which means that, instead of applying an nD convolution, we can apply an 1D convolution n times to obtain the same result with much less computational effort. The previous kernel for computing a 2D isotropic gradient with only the nearest neighbors is, fortunately, separable, and in MATLAB the evaluation takes place as follows:

```
% 1D kernels

kernelXsep10=[3 0 -3]'/6;kernelXsep01=[1 4 1]/6;
kernelYsep10=[1 4 1]'/6;kernelYsep01=[3 0 -3]/6;

% Faster separable convolution for computing the gradient

gradFXconvSep=conv2(kernelXsep10,kernelXsep01,Fpad,'valid');
gradFYconvSep=conv2(kernelYsep10,kernelYsep01,Fpad,'valid');
```

Note that in the work of [1], one of the most important 3D isotropic kernels, which is the one that uses the nearest neighbors only, was not presented. In this work, two variants of this kernel were obtained, one using only 10 nearest neighbors and the other using only 18 nearest neighbors, and their MATLAB form are given in section (6.2). Moreover, the 3D kernel using 18 nearest neighbors has the advantage of being separable, which means that we should expect to be able to rapidly compute a 3D gradient of low isotropic order using the knowledge presented in this chapter. Also note that the higher isotropic order (>2nd) kernels given in sections (6.1) and (6.2) are not separable.

4.4. Accuracy

In this work, the order accuracy is defined by n, the value in h^n in Taylor's series expansion for which the space or the isotropic order is achieved. Therefore, the space and isotropic order may be different. Although this may not have been explicitly stated by [1, 2], all isotropic gradients found in sections (6.1) and (6.2) are of second order accuracy in space, i.e. if the lattice size doubles, the error on the gradient is divided approximately by four. This can be demonstrated numerically, or analytically by means of the expression $E_x^{(2)}F(r)$ and $E_y^{(2)}F(r)$ in polar coordinates, as in Eqs. (14) and (15). These expressions are non zero for every isotropic gradient presented in this work.

The main point about the kernel that we make in this study is that, as its size increases, it becomes possible to set isotropic the higher order error term in the Taylor series expansion at fixed space second order accuracy. It is therefore important to be careful not to confuse

space order accuracy and isotropic order accuracy. We believe that, based on this result, future research could provide other kernels (perhaps of similar size) for which the leading space order accuracy would be the same as the leading isotropic order accuracy. For some applications, finding such kernels could be a significant step forward. In fact, achieving leading higher space order accuracy with equal leading isotropic order accuracy might have greater impact than achieving leading low space order with very high isotropic order accuracy, as is currently the case. However, higher space order gradient discretizations may suffer from another non physical numerical artifacts, known as spurious oscillations.

4.5. Performance

As previously indicated, computation of the gradient by applying convolution is faster than using a simpler, more straightforward method, but a naive one. We present some numerical results in this section that will show that this is indeed true. We also show that using a GPU instead of a CPU significantly reduces computation time.

First, all performance testing consists in evaluating gradients of random 2D and 3D double precision images. Note that we suppose a periodic padding for these images. All these image gradients are computed using MATLAB with the `-singleCompThread` startup option. This is done for benchmarking purposes, because the reference case should be computed using a sequential algorithm, that is, with a single core only. The CPU used in this work is an Intel® Core i7-970 processor, while the GPU is an Nvidia® GeForce GTX 580. All computations on the GPU are performed in MATLAB via the Jacket plugin developed by AccelerEyes®. The version of MATLAB is R2010b, and the Jacket version is 2.0 (build a15607c).

The timing method on the CPU is the usual MATLAB `tic; m-code; toc;` procedure. However, this method is not suited for timing m-code on the GPU. For this, we refer the reader to the method proposed by [8]. In Figures 2-5, the time taken for one simulation "dot" or result "dot" is the average of a hundred simulations.

To test performance, five different algorithms are considered for computing the gradient:

1. MATLAB singlethread naive (section 4.1)
2. MATLAB singlethread convolution (section 4.2) [REFERENCE CASE]
3. MATLAB Jacket naive
4. MATLAB Jacket convolution
5. MATLAB Jacket GFOR + convolution

Note that all results differ with respect to machine accuracy in double precision, and that the padding of the images is computed on the fly to save computer memory. This is because padding is very cheap in terms of computing cost, when compared to the cost of evaluating the gradient.

Case (4), MATLAB Jacket convolution, is the GPU equivalent of reference case (2) with a CPU. Case (3) is the GPU equivalent of case (1) with a CPU. The last case (5), MATLAB Jacket GFOR + convolution, is a special case that is not available on the CPU. To explain this, let us suppose that the user wishes to evaluate the gradient of N different images simultaneously. Using MATLAB, this is not possible without `parfor` , which is only available with the Parallel

Computing Toolbox. Also note that `parfor` is usually used for coarse-grained parallelism, while `gfor` can be used for fine-grained parallelism. Without `parfor`, an `for` loop is required to evaluate the gradients of the images sequentially, and to store the results in one large matrix by subassignment. With Jacket, it is possible to perform these N convolutions in parallel on the GPU using an `gfor` loop. Usually, this significantly reduces computation time, compared to the sequential `for` loop. More details on the functionality and limitations of the `gfor` loop can be found in [8]. In order to be able to compare the `gfor` loop case to the other cases, all performance tests have been conducted with $N = 3$, unless otherwise stated, i.e. it is supposed the user needs to evaluate three image gradients simultaneously at a given time.

In all the figures showing performance behavior, the y-axis is in log scale. Figures 2(a) and 2(b) show the performance speedup with the 2D 2nd and 14th order isotropic gradients as a function of image size. For large images, speedups of $31x$ to $39x$ can be achieved.

Figures 3(a) and 3(b) show the same situation, but with the 3D 2nd and 8th order isotropic gradients. For large images, a speedup of $34x$ to $124x$ can be achieved. These results indicate an important speedup and show the usefulness of Jacket for MATLAB.

Figures 4(a) and 4(b) show the speedup with the 2D and 3D isotropic gradient as function of the isotropy order at a fixed image size. In 2D, images are 992x992, and in 3D they are 110x110x110. For high isotropy, speedups of $49x$ to $124x$ can be achieved.

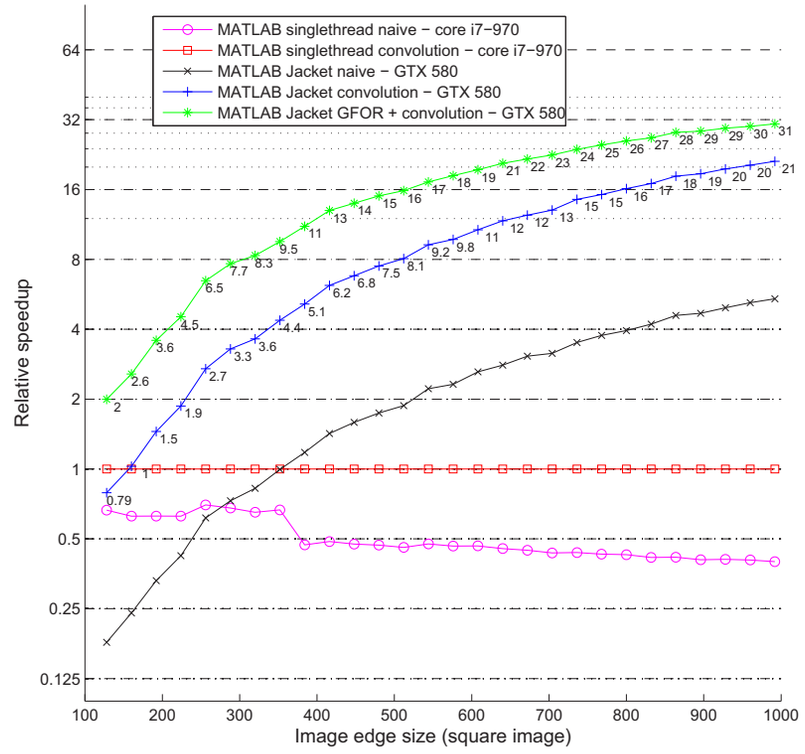
Figure 5 shows the speedup with the 3D 8th order isotropic gradient as a function of N, the number of images to be evaluated simultaneously at a fixed image size of 110x110x110. Speedups of $86x$ to $138x$ can be achieved, depending on the value of N.

For both 2D and 3D cases, as the isotropy order or the number of images to evaluate simultaneously increases, the speedup that can be achieved using the GPU also increases. This is to be expected, since the computational complexity increases. Nevertheless, the chances of obtaining a speedup of $86x$ to $138x$ for the more challenging case were much better than we had anticipated. In fact, the Jacket plugin allowed us to tackle problems that would not have been possible to deal with without using a low level programming language.

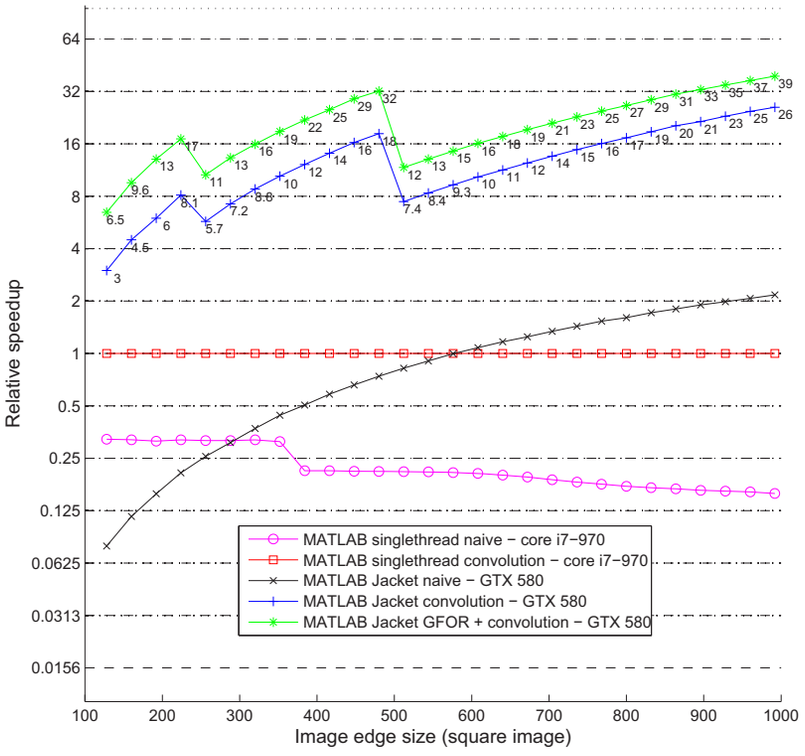
We must remember that the speedups were computed with the reference case, which uses convolution. It is important to note that a speedup of $280x$ may be achieved for the most computationally expensive test (Fig. 5) when comparing the two following cases: Jacket GFOR + convolution, and MATLAB singlethread naive. Thus, depending on how the gradient evaluations are implemented, a huge difference in computational time may be reached. Note that Jacket code does not currently take advantage of the zero entry in the kernel, and that the convolutions are performed in the spatial domain only for 9x9 kernels in 2D and 3x3x3 kernels in 3D. The naive implementation always takes advantage of the zero entry in the kernel, which means that the zero entry could yield an additional speedup, because some kernels are sparse. For example, 2D kernels have 26% to 68% zeros, while in 3D, this number varies from 33% to 84%. The 3x3 kernel contains 33% zeros while the 3x3x3 kernels contain 67% or 33% zeros.

5. Scientific application: lattice Boltzmann method

The lattice Boltzmann method is a computational approach that is mainly used for fluid flow simulation with its roots in the field of cellular automata [9]. This method is particularly useful for solving complex flow systems, such as multiphase flows, in porous media where classical

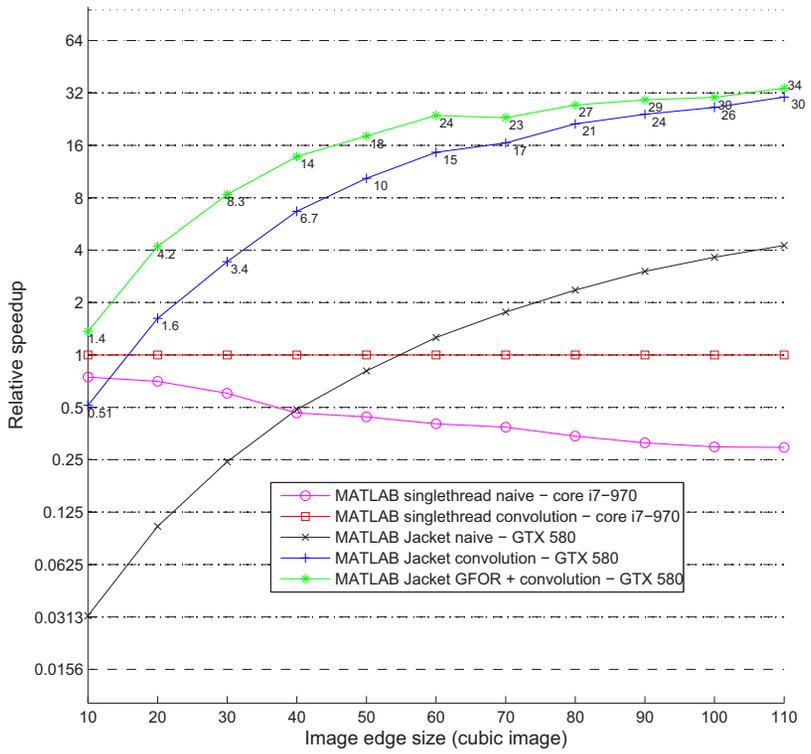


(a) 2nd order isotropic gradient.

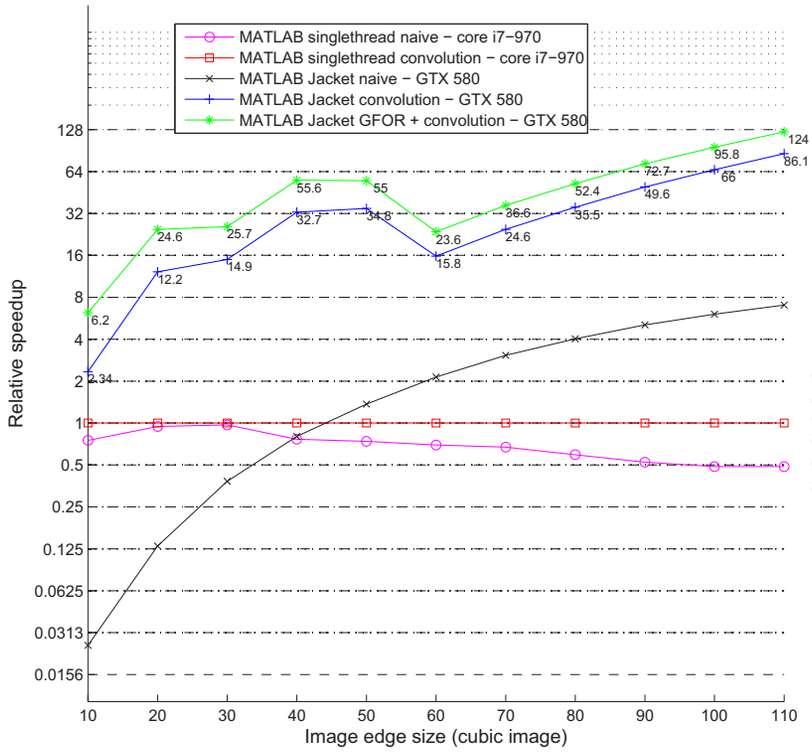


(b) 14th order isotropic gradient.

Figure 2. Speedup as a function of image size (2D isotropic gradient).

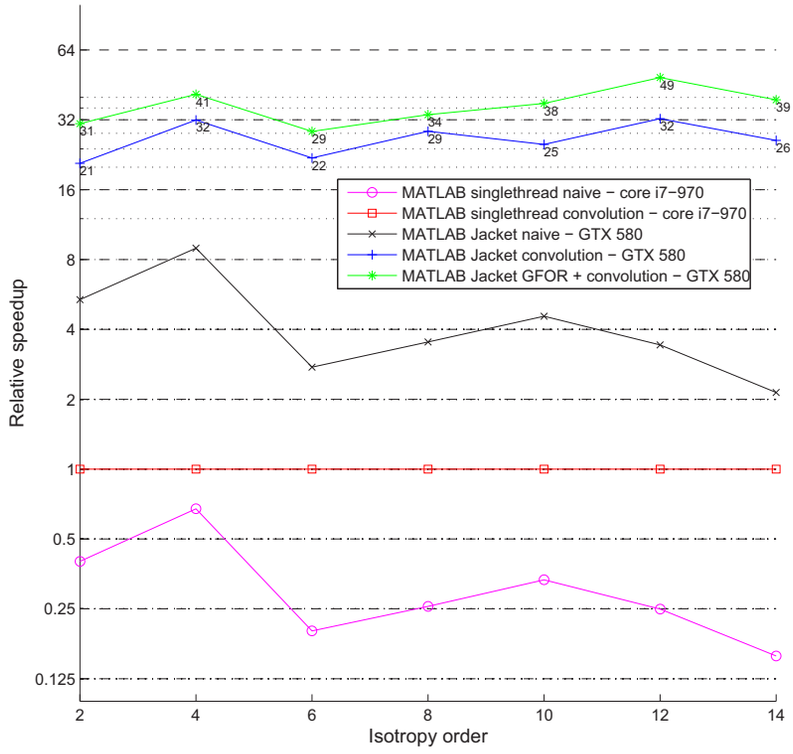


(a) 2nd order isotropic gradient.

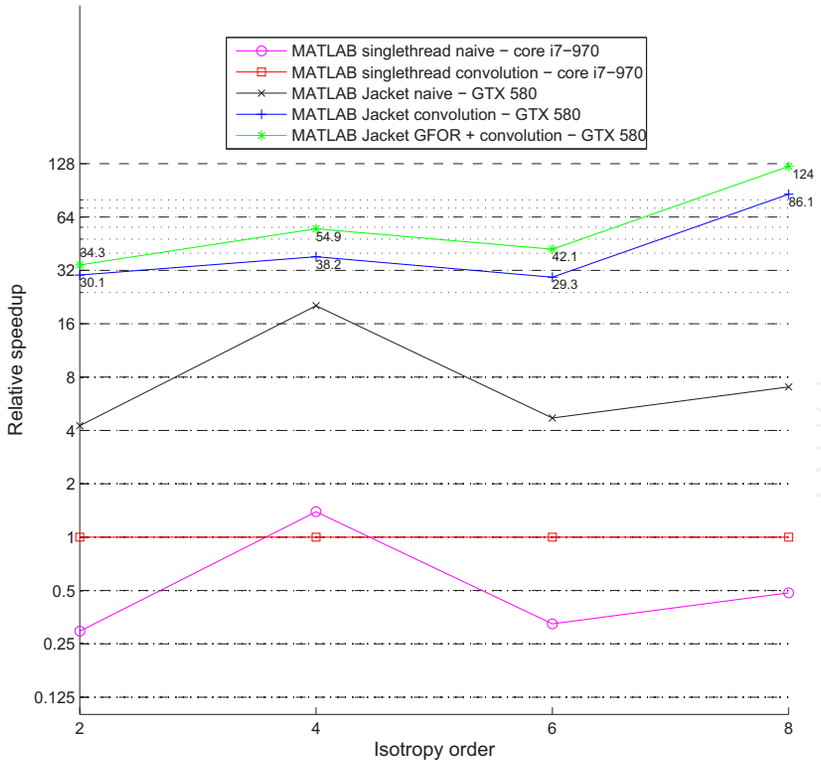


(b) 8th order isotropic gradient.

Figure 3. Speedup as a function of image size (3D isotropic gradient).



(a) Square image (992x992).



(b) Cubic image (110x110x110).

Figure 4. Speedup as a function of the gradient isotropy order.

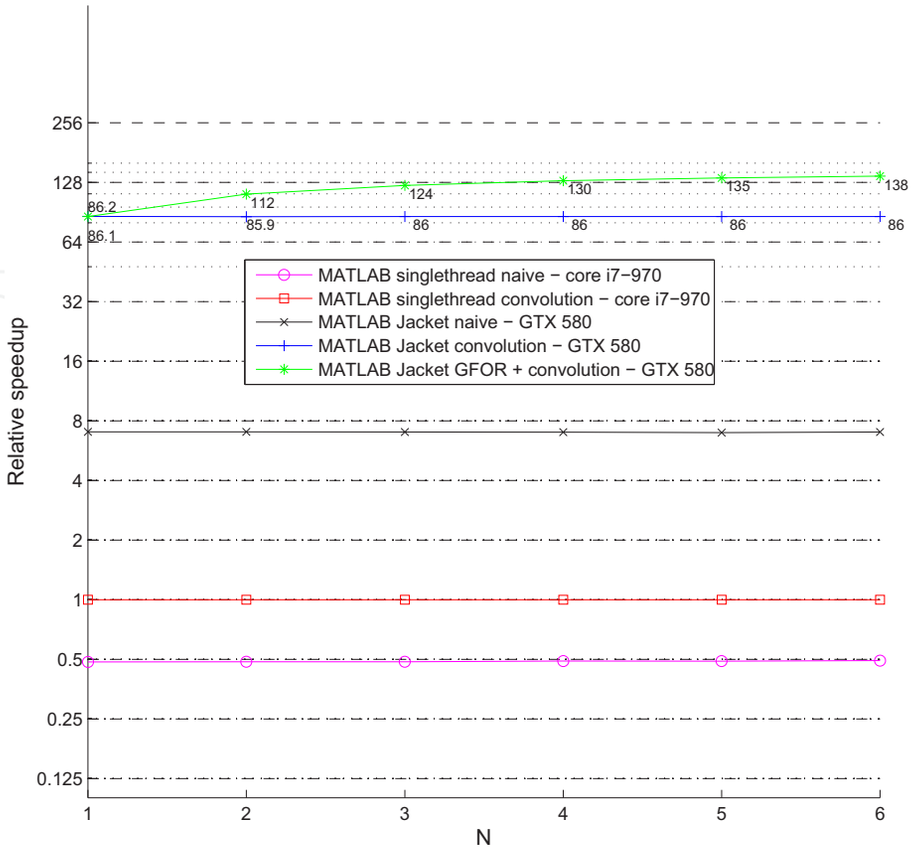


Figure 5. Speedup as a function of the number of cubic images (110x110x110) to be evaluated simultaneously using the 3D 8th order isotropic gradient.

approaches based on the Navier-Stokes equations, like finite volumes or finite elements, encounter some difficulties. The method we use is based on the original Ph.D. thesis of [10]. Since then, there have been several improvements. However, these enhancements are outside the scope of this chapter, and will not be described.

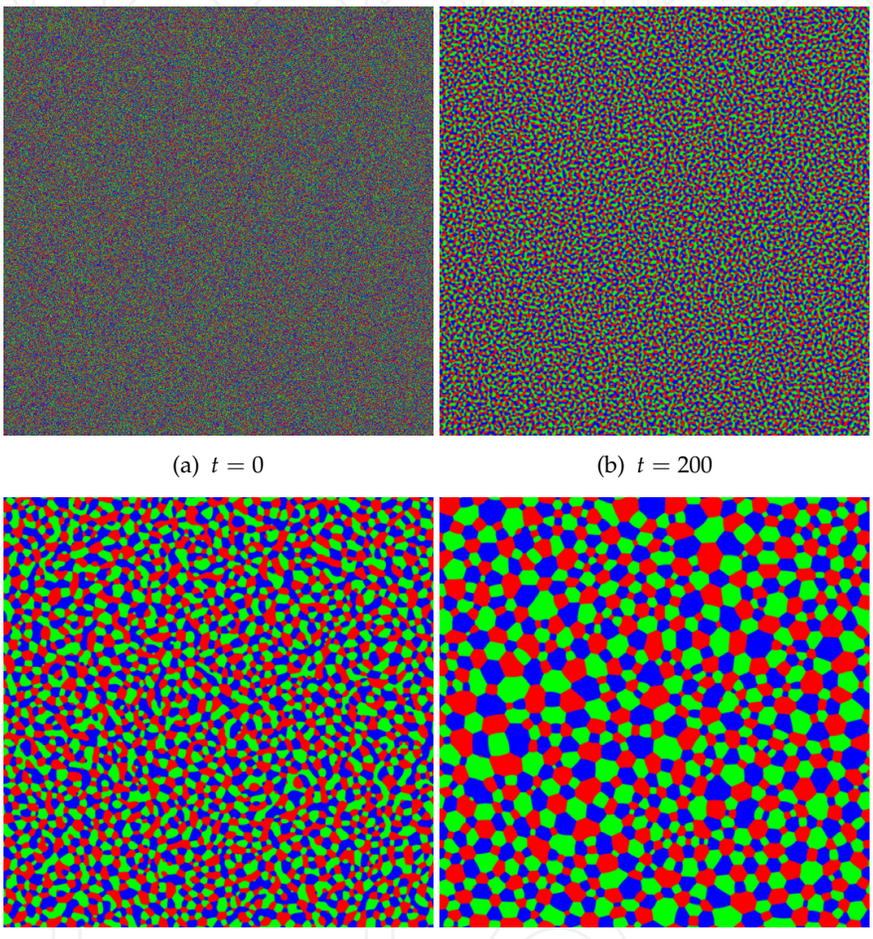
The isotropic gradients we present here are useful for simulating immiscible multiphase flows, where the orientation of the various fluid interfaces has to be computed very frequently. The gradients of the density of each fluid color (phase) define the normal orientation of the interface, and special operators are used to keep the different interfaces between the fluids defined. Moreover, the norm of these gradients serves as a means to introduce a certain amount of surface tension between the fluids.

Suppose we wish to simulate the behavior of three fluids. At a certain point in the algorithm, three image gradients need to be computed, corresponding to the interface normal of each fluid density. Here is where the methods presented in this chapter become useful, a situation described in section (4.5) with $N = 3$. It is important to point out that the isotropy of the gradient helps to reduce some of the computational artifacts that appear at the interface [1, 2]. These are called spurious currents in the lattice Boltzmann community, and are non-physical.

In multiphase flow simulation, calculation of a high order isotropic gradient is the most expensive part of the method, and the use of Jacket, has enabled us to reduce the

computational cost by an astonishing amount. This type of calculation would not have been possible, in a reasonable time, by applying plain MATLAB.

We end this section with a simulation example that shows the spinodal decomposition of a three phase flow. This flow consists of an initial random mixture, where each phase self-agglomerates until a steady state is achieved. Figure 6 shows the spinodal decomposition at various times (in lattice Boltzmann units). Note that this simulation is given for illustration purposes only, and that spinodal decomposition has been quantitatively studied by [11].



(a) $t = 0$ (b) $t = 200$

(c) $t = 2000$ (d) $t = 100000$

Figure 6. Random spinodal decomposition of a three phase flow.

6. Convolution kernel for discrete gradient approximation

In this section, we give several convolution kernels in the form of MATLAB m-code, which is very useful for calculating 2D/3D isotropic gradients on a square or cubic lattice. Most of the weights were taken from Ref. [1].

6.1. Convolution kernel for a 2D isotropic gradient

```
% 2D isotropy up to 2nd order
% This approximation use nearest neighbors only
```

```
w1=1/3;w2=1/12;
c01=1*[w2;w1;w2];
kernel=[-c01,zeros(3,1),c01];

% 2D isotropy up to 2nd order
% This approximation use nearest neighbors only (separable version)

kernelXsep10=[-3 0 3]'/6;kernelXsep01=[-1 -4 -1]/6;
kernelYsep10=[-1 -4 -1]'/6;kernelYsep01=[-3 0 3]/6;

% 2D isotropy up to 4th order

w1=4/15;w2=1/10;w4=1/120;
c01=1*[0;w2;w1;w2;0];c02=2*[0;0;w4;0;0];
kernel=[-c02,-c01,zeros(5,1),c01,c02];

% 2D isotropy up to 6th order

w1=4/21;w2=4/45;w4=1/60;w5=2/315;w8=1/5040;
c01=1*[w5;w2;w1;w2;w5];c02=2*[w8;w5;w4;w5;w8];
kernel=[-c02,-c01,zeros(5,1),c01,c02];

% 2D isotropy up to 8th order

w1=262/1785;w2=93/1190;w4=7/340;w5=6/595;
w8=9/9520;w9=2/5355;w10=1/7140;
c01=1*[w10;w5;w2;w1;w2;w5;w10];
c02=2*[0;w8;w5;w4;w5;w8;0];
c03=3*[0;0;w10;w9;w10;0;0];
kernel=[-c03,-c02,-c01,zeros(7,1),c01,c02,c03];

% 2D isotropy up to 10th order

w1=68/585;w2=68/1001;w4=1/45;w5=62/5005;w8=1/520;
w9=4/4095;w10=2/4095;w13=2/45045;w16=1/480480;
c01=1*[0;w10;w5;w2;w1;w2;w5;w10;0];
c02=2*[0;w13;w8;w5;w4;w5;w8;w13;0];
c03=3*[0;0;w13;w10;w9;w10;w13;0;0];
c04=4*[0;0;0;0;w16;0;0;0;0];
kernel=[-c04,-c03,-c02,-c01,zeros(9,1),c01,c02,c03,c04];

% 2D isotropy up to 12th order

w1=19414/228375;w2=549797/10048500;w4=175729/7917000;w5=50728/3628625;
w8=3029/913500;w9=15181/7536375;w10=221/182700;w13=68/279125;
w16=1139/26796000;w17=68/2968875;w18=17/1425060;w20=17/5742000;
w25_50=1/32657625;w25_34=1/32657625;
c01=1*[0;w17;w10;w5;w2;w1;w2;w5;w10;w17;0];
c02=2*[0;w20;w13;w8;w5;w4;w5;w8;w13;w20;0];
c03=3*[0;w25_34;w18;w13;w10;w9;w10;w13;w18;w25_34;0];
c04=4*[0;0;w25_34;w20;w17;w16;w17;w20;w25_34;0;0];
c05=5*[0;0;0;0;0;w25_50;0;0;0;0;0];
kernel=[-c05,-c04,-c03,-c02,-c01,zeros(11,1),c01,c02,c03,c04,c05];

% 2D isotropy up to 14th order

w1=285860656/3979934595;w2=2113732952/43779280545;w4=940787801/43779280545;
w5=124525000/8755856109;w8=15841927/3979934595;w9=2046152/795986919;
w10=14436304/8755856109;w13=18185828/43779280545;w16=13537939/140093697744;
w17=231568/3979934595;w18=1516472/43779280545;w20=18769/1591973838;
```

```
w25_50=184/315867825;w25_34=464/795986919;w26=1448/4864364505;
w29=148/4864364505;w32=629/400267707840;
c01=1*[w26;w17;w10;w5;w2;w1;w2;w5;w10;w17;w26];
c02=2*[w29;w20;w13;w8;w5;w4;w5;w8;w13;w20;w29];
c03=3*[0;w25_34;w18;w13;w10;w9;w10;w13;w18;w25_34;0];
c04=4*[0;w32;w25_34;w20;w17;w16;w17;w20;w25_34;w32;0];
c05=5*[0;0;0;w29;w26;w25_50;w26;w29;0;0;0];
kernel=[-c05,-c04,-c03,-c02,-c01,zeros(11,1),c01,c02,c03,c04,c05];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Final 2D kernels for gradient approximation %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

kernelX=-kernel';
kernelY=permute(kernelX,[2 1]);
```

6.2. Convolution kernel for a 3D isotropic gradient

```
% 3D isotropy up to 2nd order
% This approximation use only 10 nearest neighbors
% Note that this kernel IS NOT separable

w1=1/6;w2=1/12;w3=0;
c01=1*[w2;w1;w2];c11=1*[w3;w2;w3];
kernel(:,:,1)=[-c11,zeros(3,1),c11];
kernel(:,:,2)=[-c01,zeros(3,1),c01];
kernel(:,:,3)=kernel(:,:,1);

% 3D isotropy up to 2nd order
% This approximation use only 18 nearest neighbors
% Note that this kernel IS separable

w1=2/9;w2=1/18;w3=1/72;
c01=1*[w2;w1;w2];c11=1*[w3;w2;w3];
kernel(:,:,1)=[-c11,zeros(3,1),c11];
kernel(:,:,2)=[-c01,zeros(3,1),c01];
kernel(:,:,3)=kernel(:,:,1);

% 3D isotropy up to 4th order

w1=2/15;w2=1/15;w3=1/60;w4=1/120;
c01=1*[0;w2;w1;w2;0];c02=2*[0;0;w4;0;0];c11=1*[0;w3;w2;w3;0];
c12=2*[0;0;0;0;0];c21=1*[0;0;0;0;0];c22=2*[0;0;0;0;0];
kernel(:,:,1)=[-c22,-c21,zeros(5,1),c21,c22];
kernel(:,:,2)=[-c12,-c11,zeros(5,1),c11,c12];
kernel(:,:,3)=[-c02,-c01,zeros(5,1),c01,c02];
kernel(:,:,4:5)=kernel(:,:,2:-1:1);

% 3D isotropy up to 6th order

w1=4/45;w2=1/21;w3=2/105;w4=5/504;w5=1/315;w6=1/630;w8=1/5040;
c01=1*[w5;w2;w1;w2;w5];c02=2*[w8;w5;w4;w5;w8];c11=1*[w6;w3;w2;w3;w6];
c12=2*[0;w6;w5;w6;0];c21=1*[0;w6;w5;w6;0];c22=2*[0;0;w8;0;0];
kernel(:,:,1)=[-c22,-c21,zeros(5,1),c21,c22];
kernel(:,:,2)=[-c12,-c11,zeros(5,1),c11,c12];
kernel(:,:,3)=[-c02,-c01,zeros(5,1),c01,c02];
kernel(:,:,4:5)=kernel(:,:,2:-1:1);
```

```
% 3D isotropy up to 8th order

w1=352/5355;w2=38/1071;w3=271/14280;w4=139/14280;w5=53/10710;w6=5/2142;
w8=41/85680;w9_221=1/4284;w9_300=1/5355;w10=1/10710;w11=1/42840;
c01=1*[w10;w5;w2;w1;w2;w5;w10];c02=2*[0;w8;w5;w4;w5;w8;0];
c03=3*[0;0;w10;w9_300;w10;0;0];c11=1*[w11;w6;w3;w2;w3;w6;w11];
c12=2*[0;w9_221;w6;w5;w6;w9_221;0];c13=3*[0;0;w11;w10;w11;0;0];
c21=1*[0;w9_221;w6;w5;w6;w9_221;0];c22=2*[0;0;w9_221;w8;w9_221;0;0];
c23=3*[0;0;0;0;0;0;0];c31=1*[0;0;w11;w10;w11;0;0];
c32=2*[0;0;0;0;0;0;0];c33=3*[0;0;0;0;0;0;0];
kernel(:,:,1)=[-c33,-c32,-c31,zeros(7,1),c31,c32,c33];
kernel(:,:,2)=[-c23,-c22,-c21,zeros(7,1),c21,c22,c23];
kernel(:,:,3)=[-c13,-c12,-c11,zeros(7,1),c11,c12,c13];
kernel(:,:,4)=[-c03,-c02,-c01,zeros(7,1),c01,c02,c03];
kernel(:,:,5:7)=kernel(:,:,3:-1:1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Final 3D kernels for gradient approximation %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

kernelX=-permute(kernel,[2 1 3]);
kernelY=permute(kernelX,[2 1 3]);
kernelZ=permute(kernelX,[3 2 1]);
```

6.3. Convolution kernel for 2D anisotropic edge detection

```
% 2D Prewitt kernel

w1=1;w2=1;
c01=1*[w2;w1;w2];
kernel=[-c01,zeros(3,1),c01];

% 2D Sobel kernel

w1=2;w2=1;
c01=1*[w2;w1;w2];
kernel=[-c01,zeros(3,1),c01];

% 2D Scharr kernel

w1=3;w2=10;
c01=1*[w2;w1;w2];
kernel=[-c01,zeros(3,1),c01];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Final 2D kernels for gradient approximation %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

kernelX=-kernel';
kernelY=permute(kernelX,[2 1]);
```

7. Conclusion

In this work, a detailed description of isotropic gradient discretizations and convolution products has been presented. These isotropic gradients are useful, and superior to anisotropic

discretizations. This is especially true in the field of flow simulation, when the lattice Boltzmann method is used. However, high order isotropic gradients are computationally expensive. To address this issue, we combined the convolution product with the Jacket plugin in MATLAB and GPU hardware, which enabled us to achieve high computational speedups (up to 138x), compared to plain MATLAB computations using a CPU. We end this chapter with one final note. While we have provided a useful list of MATLAB m-code defining the kernels needed for evaluating 2D and 3D isotropic gradients, these kernels only lead to second order space accuracy with high isotropic order. The development of kernels that would lead to high order space accuracy combined with high isotropic order could generate significant benefits, particularly for the lattice Boltzmann community addressing multiphase flows. However, the benefits may not be straightforward, because the higher space order gradient discretization may lead to another unwanted numerical artifacts, such as spurious oscillations in regions separating two phases.

Acknowledgments

We extend our special thanks to Pavan Yalamanchili from AccelerEyes for his quick response to our queries and his generous support. We applied the sequence-determines-credit (SDC) approach to our listing of authors [12]. This work was supported by a grant from the NSERC (Natural Sciences and Engineering Research Council of Canada).

Author details

Sébastien Leclaire, Maud El-Hachem and Marcelo Reggio
Department of Mechanical Engineering, École Polytechnique de Montréal, Canada

8. References

[1] M. Sbragaglia, R. Benzi, L. Biferale, S. Succi, K. Sugiyama, and F. Toschi. Generalized lattice boltzmann method with multirange pseudopotential. *Physical Review E*, 75(2):026702, 2007.

[2] Sébastien Leclaire, Marcelo Reggio, and Jean-Yves Trépanier. Isotropic color gradient for simulating very high-density ratios with a two-phase flow lattice boltzmann model. *Computers & Fluids*, 48(1):98–112, 2011.

[3] Sayed Kamaledin Ghiasi Shirazi and Reza Safabakhsh. Omnidirectional edge detection. *Computer Vision and Image Understanding*, 113(4):556 – 564, 2009.

[4] M.K. Ray, D. Mitra, and S. Saha. Simplified novel method for edge detection in digital images. In *Signal Processing, Communication, Computing and Networking Technologies (ICSCCN), 2011 International Conference on*, pages 197–202, july 2011.

[5] T.L. Chow. *Mathematical methods for physicists: a concise introduction*. Cambridge University Press, 2000.

[6] E. Oran Brigham. *The fast Fourier transform and its applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[7] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[8] Accelereyes website, 2012.

[9] Hasslacher B. Frish U and Y. Pomeau. *Phys. Rev. Lett.*, 56(14):1505, 1986.

[10] Andrew K. Gunstensen. *Lattice-Boltzmann Studies of Multiphase Flow Through Porous Media*. PhD thesis, 1992.

[11] F. J. Alexander, S. Chen, and D. W. Grunau. Hydrodynamic spinodal decomposition: Growth kinetics and scaling functions. *Physical Review B*, 48(1):634, 1993.

[12] Teja Tscharntke, Michael E. Hochberg, Tatyana A. Rand, Vincent H. Resh, and Jochen Krauss. Author sequence and credit for contributions in multiauthored publications. *PLoS Biol*, 5(1):e18, 2007.

IntechOpen

IntechOpen