Optical Flow Estimation with CUDA

Mikhail Smirnov
msmirnov@nvidia.com

July 2012

Document Change History

Version	Date	Responsible	Reason for Change
		Mikhail Smirnov	Initial release

Abstract

Optical flow is the apparent motion of objects in image sequence. It has a number of applications ranging from medical imaging to visual effects. This report describes a CUDA implementation of a 2D optical flow method referred to as Hierarchical Horn and Schunck. This optical flow method is based on two assumptions: brightness constancy and spatial flow smoothness. These assumptions are combined in a single energy functional and solution is found as its minimum point. The numerical scheme behind the solver is based on a finite differences approximation of the corresponding Euler-Lagrange equation. Implementation incorporates coarse-to-fine approach with warping to deal with large displacements.

Motivation

When working with image sequences or video it's often useful to have information about objects movement. Optical flow describes apparent motion of objects in image sequence. It can be understood as a per-pixel displacement field.

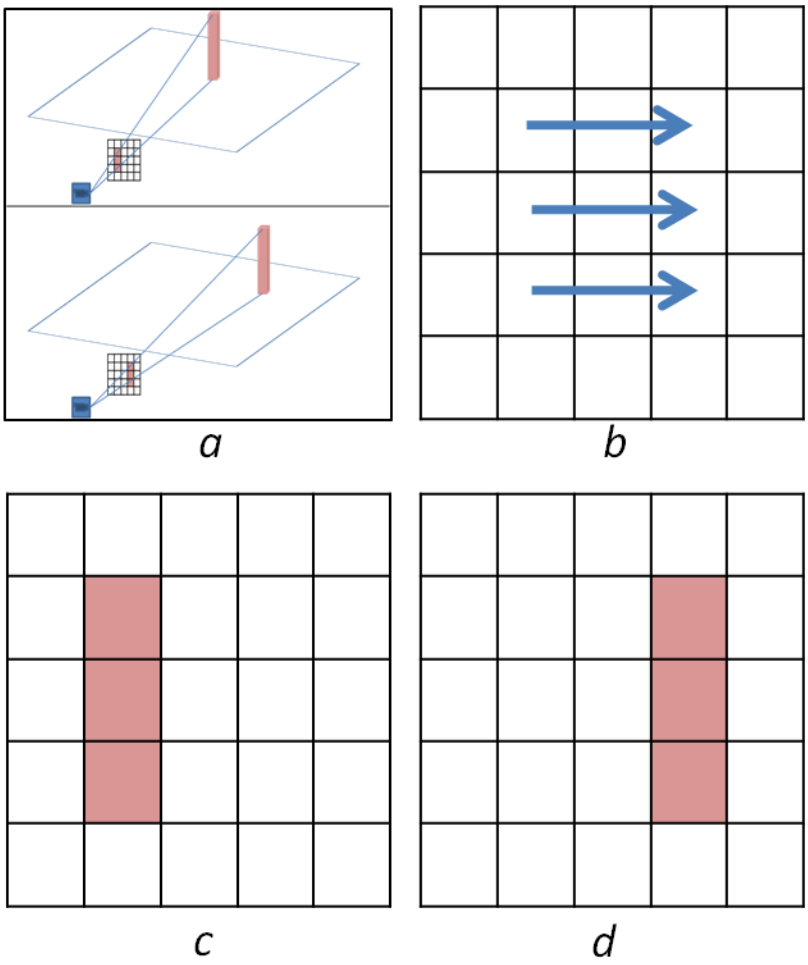

Figure 1. *a* – camera looking at a moving object; *b* – optical flow field on a pixel grid of the image (zero vectors are not shown); *c* – left view of object from a; *d* – right view of the same object

Figure 1 shows an example of brightness pattern movement. Relative motion of the object and the camera results in the movement of brightness patterns in the image. Optical flow describes per-pixel correspondence between old and new locations of an object within image.

Optical flow has a number of applications in visual effects and robotics. Examples of such applications are retiming, 3D reconstruction, tracking, and autonomous navigation.

How Does It Work?

The rest of the paper applies to the case of grayscale images, but this approach can be extended to the case of multichannel images. Example of such extension is (Mileva, Bruhn, & Weickert, 2007).

Model

Horn & Schunck optical flow method is based on two assumptions (Horn & Schunck, 1981).

First assumption is called brightness constancy assumption. It states that the apparent brightness of objects in scene remains constant.

Let Ω be an image and by $I(x, y, t)$ denote brightness of a pixel (x, y) at a moment t, which can be considered as a frame number. Let $\big(u(x, y), v(x, y)\big)$ be a displacement field defined on Ω, then the brightness constancy assumption can be written as follows:

$$I(x + u, y + v, t + 1) = I(x, y, t) \tag{1}$$

Second assumption states that neighboring pixels move similarly. In other words flow field is smooth. One of the ways to describe degree of smoothness at a particular point is to compute the square of the magnitude of the velocity gradient:

$$\|\nabla u\|^2 = \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 \text{ and } \|\nabla v\|^2 = \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2$$

Deviations from brightness constancy assumption and flow smoothness assumption can be penalized with the following energy functionals:

$$E_b(u, v) = \int_\Omega (I(x + u, y + v, t + 1) - I(x, y, t))^2 dxdy \tag{2}$$

$$E_s(u, v) = \int_\Omega \|\nabla u\|^2 + \|\nabla v\|^2 dxdy \tag{3}$$

Weighted sum of these functionals gives us idea of how good particular displacement field is. The problem is then to find a global minimum point of the following energy functional:

$$E(u, v) = E_b(u, v) + \alpha E_s(u, v), \qquad \alpha > 0 \tag{4}$$

Original Horn & Schunck approach uses linearized version of (1):

$$I(x + u, y + v, t + 1) \approx I(x, y, t) + I_x(x, y, t) \cdot u + I_y(x, y, t) \cdot v + I_t(x, y, t) \cdot 1$$

$$I(x + u, y + v, t + 1) - I(x, y, t) \approx I_x u + I_y v + I_t$$

And (2) becomes

$$E_b(u, v) = \int_\Omega \big(I_x u + I_y v + I_t\big)^2 dxdy \tag{5}$$

According to the calculus of variations u and v must satisfy the Euler-Lagrange equations

$$\begin{aligned}(I_x u + I_y v + I_t)I_x - \alpha(u_{xx} + u_{yy}) = 0 \\ (I_x u + I_y v + I_t)I_y - \alpha(v_{xx} + v_{yy}) = 0\end{aligned} \quad (6)$$

with reflecting (Neumann) boundary conditions. The displacement field can be found as a solution of (6).

Note: $\Delta u \stackrel{\text{def}}{=} u_{xx} + u_{yy}$

The problem with this approach is that it assumes small pixel displacements (due to the use of 1st degree Taylor polynomials). The problem can be addressed by the following consideration. To the moment we already have estimation of (u, v) up to small unknown increment (du, dv). Using conventional Horn & Schunck approach we can find (du, dv) as a displacement field between $I(x, y, t)$ and $I(x + u, y + v, t + 1)$:

$$\tilde{I}(x, y, t + 1) \stackrel{\text{def}}{=} I(x + u, y + v, t + 1)$$

$$\tilde{I}(x + du, y + dv, t + 1) \approx \tilde{I}_x(x, y, t + 1)du + \tilde{I}_y(x, y, t + 1)dv + \tilde{I}(x, y, t + 1)$$

$$E_b(du, dv) = \int_\Omega \left(\tilde{I}_x(\mathbf{x}, t + 1)du + \tilde{I}_y(\mathbf{x}, t + 1)dv + \tilde{I}(\mathbf{x}, t + 1) - I(\mathbf{x}, t)\right)^2 d\mathbf{x}$$

Finally we have the following partial differential equations for du, dv

$$\begin{aligned}(\tilde{I}_x du + \tilde{I}_y dv + \tilde{I} - I)\tilde{I}_x - \alpha(du_{xx} + du_{yy}) = 0 \\ (\tilde{I}_x du + \tilde{I}_y dv + \tilde{I} - I)\tilde{I}_y - \alpha(dv_{xx} + dv_{yy}) = 0\end{aligned} \quad (7)$$

Note: $\tilde{I}(x, y, t + 1)$ is $I(x, y, t + 1)$ **warped** with vector field (u, v), i.e. every pixel (x, y) in $\tilde{I}$ is the pixel $(x + u, y + v)$ from I.

Once we have found the unknown increment, u and v can be updated. We can keep refining the displacement field until convergence. Such incremental approach can be combined with the coarse-to-fine strategy. We start with solving problem for simplified downsampled images, where large displacements become order of pixel, and use the solution as the starting point for larger resolution images. In such way we can solve the original problem step by step.

Putting all the ideas together, we come up with the following method:

1. Prepare downsampled versions of original images, i.e. generate image pyramid.
2. Initialize u, v with zero
3. Select the lowest resolution
4. Warp image $I(x, y, t + 1)$ with current u, v
5. Compute $\tilde{I}_x, \tilde{I}_y$ and $\tilde{I} - I$
6. Solve (7) for du, dv
7. Update $u \leftarrow u + du, \ v \leftarrow v + dv$
8. If du, dv are not small enough, go to step 4

9. If current resolution is less than original, proceed to the finer scale and go to step 4

Steps 4-7 are called warping iteration.

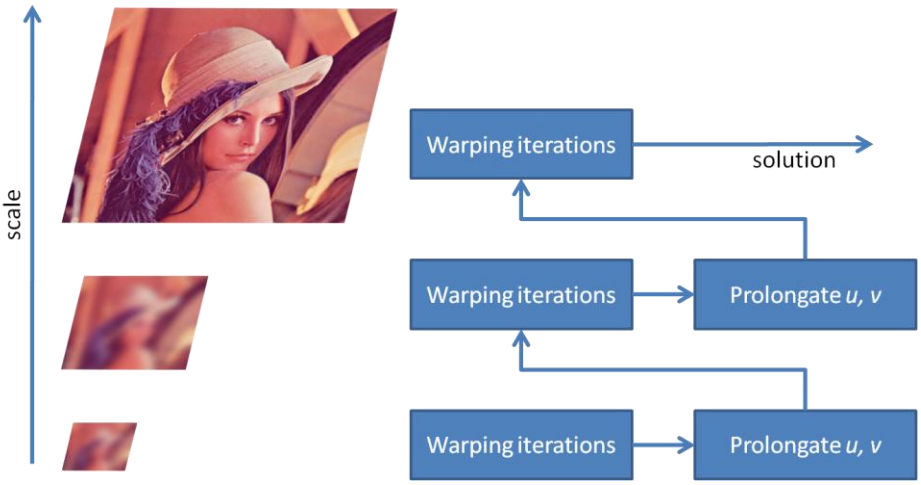

Figure 2. Method outline

Solving PDE

The most complex problem of the algorithm above is to solve (7). It can be done with finite differences method.

Laplace operator is approximated with the standard 5-point stencil:

$$(\Delta u)_{i,j} \approx \left(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\right) + \left(u_{ij-1} - 2u_{i,j} + u_{i,j+1}\right) \tag{8}$$

Note: In image processing applications grid spacing is usually taken equal to one

Using (8) we can rewrite (7) as a system of linear equations

$$\begin{aligned} \left(\tilde{I}_x du_{ij} + \tilde{I}_y dv_{ij} + \tilde{I} - I\right)\tilde{I}_x - \alpha(\Delta du)_{ij} = 0 \\ \left(\tilde{I}_x du_{ij} + \tilde{I}_y dv_{ij} + \tilde{I} - I\right)\tilde{I}_y - \alpha(\Delta dv)_{ij} = 0 \end{aligned} \tag{9}$$

These equations can be solved by applying Jacobi method (Horn & Schunck, 1981):

$$\begin{aligned} du_{ij}^{n+1} = \overline{du}_{ij}^{n} - \frac{\tilde{I}_{x,ij}\left(\tilde{I}_{x,ij}\overline{du}_{ij}^{n} + \tilde{I}_{y,ij}\overline{dv}_{ij}^{n} + \tilde{I}_{ij} - I_{ij}\right)}{\alpha + I_{x,ij}^2 + I_{y,ij}^2} \\ dv_{ij}^{n+1} = \overline{dv}_{ij}^{n} - \frac{\tilde{I}_{y,ij}\left(\tilde{I}_{x,ij}\overline{du}_{ij}^{n} + \tilde{I}_{y,ij}\overline{dv}_{ij}^{n} + \tilde{I}_{ij} - I_{ij}\right)}{\alpha + I_{x,ij}^2 + I_{y,ij}^2} \end{aligned} \tag{10}$$

Where

$$\overline{du}_{ij}^{n} \stackrel{\text{def}}{=} \frac{1}{4}\left(du_{i-1j}^{n} + du_{i+1j}^{n} + du_{ij-1}^{n} + du_{ij+1}^{n}\right),$$

$$\overline{dv}_{ij}^{n} \stackrel{\text{def}}{=} \frac{1}{4}\left(dv_{i-1j}^{n} + dv_{i+1j}^{n} + dv_{ij-1}^{n} + dv_{ij+1}^{n}\right)$$

Border conditions are approximated as follows:

$$du_{ij} = \begin{cases} du(i,j), & 0 \le i < W,\ 0 \le j < H \\ du(0,j), & i = -1,\ 0 \le j < H \\ du(W-1,j), & i = W,\ 0 \le j < H \\ du(i,0), & 0 \le i < W,\ j = -1 \\ du(i,H-1), & 0 \le i < W,\ j = H \end{cases}$$

The same holds for dv.

Computing image derivatives

Image derivatives are approximated with 5-point derivative filter $\frac{1}{12}(-1,\ 8,\ 0, -8,\ 1)$ and then averaged temporally as suggested in (Black & Sun, 2010). For example, $\tilde{I}_x(i,j)$ is replaced with $\frac{1}{2}(\tilde{I}_x(i,j,t+1) + I_x(i,j,t))$. Points outside of the image domain are reflected across borders.

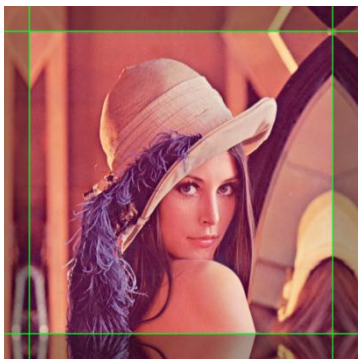

Figure 3. Handling points outside of the image. Image borders are marked with green.

Restriction and Prolongation

Restriction and prolongation are two of the most important operations in the coarse-to-fine approach. These terms originate from the theory of multigrid methods. Restriction is responsible for injecting a function into the coarse grid from the fine grid. Prolongation is the opposite operation: it restores function on the fine grid from its coarse grid representation. In our case restriction is downsampling.

It's shown in (Black & Sun, 2010) that downscaling factor of 0.5 should provide good enough solution. We need to blur source image before downsampling to avoid aliasing. Simple 2x2 averaging provides good enough quality.

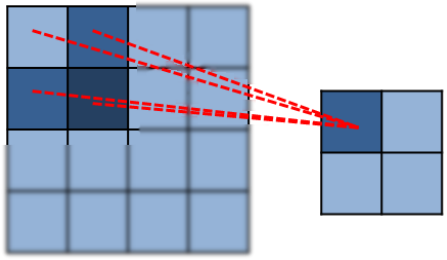

Figure 4. Restriction

We perform prolongation by simple bilinear interpolation followed by appropriate scaling. Since pyramid downscaling factor is of 0.5, we need to scale interpolated vector by a factor of $\frac{1}{0.5} = 2$.

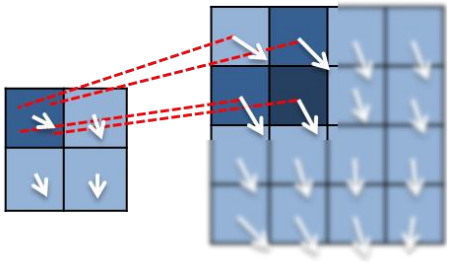

Figure 5. Prolongation

Implementation Details

This section focuses on CUDA implementation. The reference CPU implementation is quite similar.

CUDA Memory Organization

2D data are represented as linear arrays. The memory layout is depicted in Figure 6. Indices are explicitly calculated by the kernels. For example, element (i, j) has index

$$index = i * stride + j$$

Note: The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called **warps** (CUDA Programming Guide, Chapter Hardware Implementation).

Note: When a warp executes an instruction that accesses global memory, it **coalesces** the memory accesses of the threads within the warp into one or more of these memory transactions (CUDA Programming Guide, Chapter Performance Guidelines, section Global Memory

Arrays are padded in such way that $stride$ is always a multiple of warp size. This ensures coalesced loads.

$$stride = width + padding$$

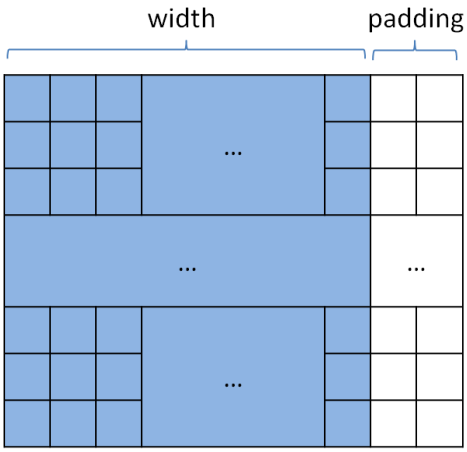

Figure 6. Memory layout

Texture unit has several features that are especially useful in our case:

- Mirror addressing mode
- Bilinear interpolation
- Cache

Mirror addressing mode allows automatically handle out-of-range coordinates. It eases computing derivatives and warping whenever we need to reflect out-of-range coordinates across borders.

Bilinear interpolation is used in restriction, prolongation and warping. Texture unit performs interpolation in fixed point arithmetic but its precision is enough for our case.

Texture cache is organized to handle 2D locality and our application can benefit from it due to the following reasons:

- Derivative filters are local by their nature
- Neighboring pixels undergo similar motion which leads to 2D locality during image warping

Shared memory can be efficiently utilized in Jacobi iterations. Each thread block processes its own image tile. Data for this tile can be loaded into shared memory to allow data reuse.

CUDA Kernels

Recall the method outline:

1. Generate image pyramid (one for source image and one for target)
2. Initialize u, v
3. Select the top level of the pyramid
4. Warp target image with current flow
5. Compute derivatives

6. Solve for du, dv
7. Update u, v
8. Go to step 4 if required (i.e. if solution has not converged)
9. If current level isn't the lowest pyramid level
    a. Prolong u, v to a finer grid
    b. Go to step 4

Let's go through this algorithm step by step. The first step is the image pyramid generation.

We can recursively obtain pyramid levels one by one. This approach is faster than generation of levels "on the fly" but at the expense of additional memory consumption. An overhead isn't as large as it may seem: asymptotically, for a pyramid with scale factor of two we'll need only twice as much memory space as required for an original image. All output pixels can be computed independently, therefore we create thread for each output location (x, y) and obtain output value by averaging over four source locations:

$$\left(\frac{1}{w}(x-0.5), \frac{1}{h}(y-0.5)\right), \left(\frac{1}{w}(x+0.5), \frac{1}{h}(y-0.5)\right),$$
$$\left(\frac{1}{w}(x-0.5), \frac{1}{h}(y+0.5)\right), \text{ and } \left(\frac{1}{w}(x+0.5), \frac{1}{h}(y+0.5)\right)$$

Image warping is quite straightforward: we create thread for each output pixel (x, y) and fetch from a source texture at location $(x + u, y + v)$. Texture addressing mode is set to `cudaAddressModeMirror`, texture coordinates are normalized.

Once we have the warped image, derivatives become simple to compute. For each pixel we fetch required stencil points from texture and convolve them with filter kernel. In terms of CUDA we create a thread for each pixel. This thread fetches required data and computes derivative.

Step 6 employs several Jacobi iterations with respect to formulas (10). Border conditions are explicitly handled within the kernel. Each thread updates one component of du and one component of dv. The number of iterations is held fixed during computations. This eliminates the need for checking error on every iteration. The required number of iterations can be determined experimentally.

In order to perform one iteration of Jacobi method in a particular point we need to known results of previous iteration for its four neighbors. If we simply load these values from global memory each value will be loaded four times. We store these values in shared memory. This approach significantly reduces number of global memory accesses, provides better coalescing, and improves overall performance.

Updating of u, v is a simple element-wise vector addition. It is done once per warping iteration.

Prolongation is performed with bilinear interpolation followed by scaling. u and v are handled independently. For each output pixel there is a thread which fetches output value from the texture and scales it.

Code Organization

The SDK Project contains both CPU and GPU implementations. For every CUDA kernel there is a .cu file containing kernel and its wrapper.

The source code is divided into the following files:

- `addKernel.cu`: Vector addition.
- `derivativesKernel.cu`: Image derivatives: spatial and temporal
- `downscaleKernel.cu`: Image downscaling
- `flowCUDA.cu`: A high-level structure of GPU implementation, also handles host-device transfers.
- `flowGold.cpp`: CPU implementation.
- `main.cpp`: image I/O related code, calls of CPU and GPU functions and results comparison.
- `solverKernel.cu`: Jacobi iteration
- `upscaleKernel.cu`: Prolongation
- `warpingKernel.cu`: Image warping

Performance

This sample is not intended to provide the best possible performance but to show one of the possible ways to implement modern optical flow methods with CUDA.

There are several ways to improve the performance. One of them is to replace Jacobi iterations with faster solver, PCG for example. Another way is to optimize execution speed of Jacobi iterations: one may notice that denominators in formulas (10) are the same. Using this observation, it is possible to compute reciprocal of the denominator only once during computation of derivatives. In such way the number of required arithmetic operations will be reduced along with the number of loads from DRAM.

Conclusion

In this paper we described approach to optical flow estimation with CUDA. This implementation can be a good starting point for more complex modern variational methods because it is already incorporates coarse-to-fine approach with warping and high-order image derivatives approximation.

Bibliography

Black, M. J., & Sun, D. (2010). Secrets of Optical Flow and Their Principles. *CVPR*.

Horn, B. K., & Schunck, B. G. (1981). Determining Optical Flow. *Artificial Intelligence , 17*, pp. 185-203.

Mileva, Y., Bruhn, A., & Weickert, J. (2007). Illumination-robust variational optical flow with photometric invariants. *Pattern Recognition. Lecture Notes in Computer Science , 4713* , 152-162. (F. A. Hamprecht, C. Schnorr, & B. Jahne, Eds.) Berlin: Springer.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com