Parallel Visual Computing 2012/13
Assignment 6

Mandatory Tasks

Multi-Scale Horn & Schunck for large displacements (12 points)

Implement a multi-scale version of the Horn & Schunck algorithm. The relevant literature are the original Horn & Schunck paper [HS80] for understanding the basic principle (if you are interested), an ICCV course on variational optical flow [BB09] (also as background reading), and an article on a multi-scale version of the Horn & Schunck method [MLP12] that comes with an extensive evaluation website and CPU C-code.

Start with "oflow_v3.cu" which is the same algorithm as "oflow.cu" discussed in the lecture, but has a visualization component in OpenGL (you need to press "ESC" in the visualization window to start the computation, pressing it again stops the computation). The visualization uses a GLSL shader "DebugFragment.glsl". The visualization can be turned off completely by commenting the "#define SHOW" statement. The code (both CUDA and OpenGL parts) has been verified to run on a "GeForce 8600 GT" with 4 multi-processors à 8 cores, 256 MB RAM (compute capability 1.1). I suspect it to work with 1.0 devices as well, but I do not have any of those lying around.

A pseudo code of what needs to be done is included in the lecture slides. The paper [MLP12] (and associated code) may give you some inspiration, however, there are slight differences between the provided implementation (and pseudo-code on the slides). In particular there are no new equations that have to be implemented. The classical Horn & Schunck method can be used as is.

In order to make this run well, the parameters must be chosen adequately. The website is not much of a help in that respect since our images are internally represented at a different scale. I have made good experiences with a pre-blur of $\sigma_{max} = 7.0$ pixels, the sigma for every outer iteration being linearly reduced from σ_{max} to $\sigma_{min} = 0.01$. Start with the largest blur level and work you way towards finer scales.

The largest blur determines the amount of motion that can be estimated correctly (at this scale, the linear Taylor approximation should hold for the largest displacement). The number of outer loop steps (i.e. the number of scales) has an effect on the run-time and the accuracy of the estimate. 7 steps have worked well for me. Remember to adjust the kernel half-size when pre-blurring.

The regularization parameter α controls the smoothness of the solution. The larger it is, the smoother the solution. However, propagating this smoothness towards everywhere in the image is a diffusion process (and therefore slow with the current numerical scheme). For large values (> 1.5) you might need more than 2000 inner iterations to converge. Good values for α are between 0.1 and 1.5 (test with the provided code). For the multi-scale version I obtained the best results with large α (≈ 1.5 for $\sigma = 7.0$) on coarser scales, decreasing its value towards finer scales (to ≈ 0.1 for $\sigma = 0.01$).

Note: There is some bug related to grid and block sizes that do not fully divide the image size. Running the algorithm with BLOCK_SIZE_X = 4, BLOCK_SIZE_Y = 4 should work on the provided images. Whoever fixes this gets 5 points extra.

Convergence-based stopping criterion for inner iterations (overall 8 points)

In the provided code, there is a fixed number of 2000 inner iterations. It might not be necessary to have that many iterations (because nothing changes anymore), or your solution might not be fully converged (in which case you might want to run it longer. The standard way of dealing with this is to introduce a stopping criterion. Colloquially this would be stated as "If nothing changes significantly anymore, then stop the iterations". The condition is that the flow vectors do not change much between iterations. It can be checked by computing the difference between two successive (inner loop) iterations. You compute $(u_x^{j+1} - u_x^j)^2 + (u_y^{j+1} - u_y^j)^2$ at every pixel (where j is the inner iteration counter) and sum all of these together. For normalization, you divide this sum by the number of pixels (you do not want to choose a different stopping threshold when computing on differently sized images). The slides have some information on this too. Continuation of your main loop should depend on the stopping threshold being reached or a maximum number of user-set iterations (to avoid infinite computations).

Implement 3 options

- Read the flow vectors back to the CPU and compute the stopping criterion there. This solution will serve as ground truth for GPU implementations. (3/8 points).
- Implement a GPU kernel that avoids reading back the flow vectors. A simple solution computes a partial sum for each block and stores it into a global memory array. The array of partial sums is read back to the CPU, and the final sum is computed there. Alternatively, use atomic operations (see CUDA manual [NVI12]). You can try both integer (think about scaling your intermediate results appropriately) and floating point atomicAdd (available hardware permitting, requires $>$ GT400). (3/8 points).

- compute the sum in parallel, some hints for this can be found in [NBGS08]. (2/8 points)

Optional Tasks

* Multi-Scale by Sub-Sampling (10 points)

Implement the multi-scale scheme in a sub-sampling fashion (do not forget to pre-blur to avoid aliasing). This measure will save many computations and will provide tremendous speed-ups (I hope). The reasons for this is that diffusion distances are propagated much faster on low resolution images than on high-resolution ones. The limiter for convergence in our application is the diffusion component that carries flow estimates into the texture-less regions.

* Extended to 3-channel colored flow (5 points)

The current equations are for gray-valued images. Extend the estimate to work with 3 color channels. Only a single set of flow vectors should be estimated.

* Investigate bottlenecks with the NVIDIA Visual Profile (5 points)

Time your algorithm with nvvp. For this task, use the no SHOW variant of the algorithm. Which synchronization points are really necessary ? What is the bottleneck ? Can the algorithm be improved - how ? Document the benefits.

* Compute $\bar{u}_x$ and $\bar{u}_y$ in shared memory (5 points)

The slowest part of the algorithm (74% in my implementation) is the computation of the $\bar{u}_x$ and $\bar{u}_y$ averaged values. Can we benefit if the data that is necessary per block is first transferred to shared memory ? By how much ? Document your findings.

References

[BB09] Andrés Bruhn and Thomas Brox. Variational Optical Flow Estimation. ICCV tutorial, 2009.

[HS80] Berthold K. P. Horn and Brian G. Schunck. Determining Optical Flow. MIT A.I. Memo No. 572, 1980.

[MLP12] Enric Meinhardt-Llopis and Javier Sánchez Pérez. Horn-Schunck Optical Flow with a Multi-Scale Strategy. IPOL preprint (http://www.ipol.im/pub/pre/20/), 2012.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40 – 54, March/April 2008.

[NVI12] NVIDIA. CUDA 5.0 C Programming Guide. technical document, 2012.