ME759 Final Project Report
University of Wisconsin-Madison

GPU-Accelerated Particle Image Velocimetry

Alex Ames

December 21, 2017

Abstract

Experimental research in fluid mechanics relies heavily on particle image velocimetry, an image processing technique which correlates the positions of particles in two successive images to obtain a velocity field. Conventional approaches to the correlation process rely on matrix multiplication or fast Fourier transforms performed on the CPU, requiring processing times from minutes to hours for a single image pair. By adapting a gradient descent method from the computer vision community, calculation of velocity vectors can be accelerated by several orders of magnitude. Further acceleration is enabled by performing computations on the GPU, allowing for image processing times of less than a second.

Contents

1 Particle image velocimetry

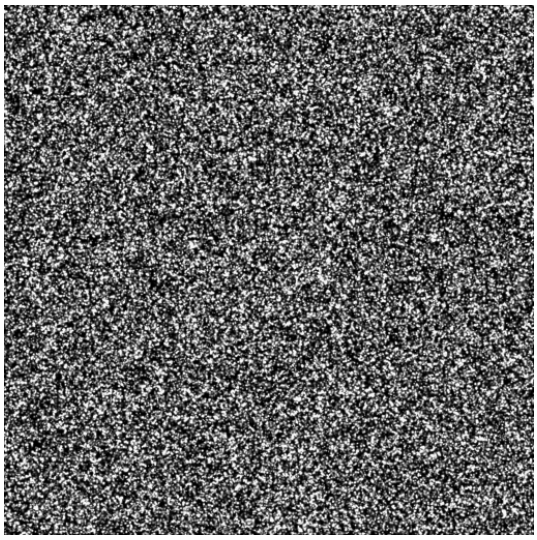


Figure 1: Particles tracking a strong vortex [1]

In recent decades, particle image velocimetry (PIV) has become an invaluable tool for studying the structure and properties of gas and liquid flows. By non-intrusively seeding a flow with a large number of small particles, then tracking the particles' displacement through multiple successive image frames, accurate velocity fields can be obtained. Although the technique was developed in the era of film cameras, with photographic negatives digitized and then processed, PIV has come into its own with the widespread availability of digital cameras.

Traditional PIV algorithms work by dividing the images into a grid of sub-images, then finding the inter-image displacements which maximize the correlation between each pair of sub-images. The simplest method of determining correlations is performing matrix multiplication of the sub-image pairs at every possible displacement (with appropriate boundary padding), but this approach can require several hours of processing time for a single image pair. More sophisticated approaches use fast Fourier transform (FFT) routines to determine correlations in a matter of minutes, but the resultant velocity fields exhibit a phenomenon known as peak-locking due to single-pixel particles, decreasing their accuracy [2]. To improve the resolution of the output velocity field, PIV algorithms typically overlap the search boxes and recursively decrease the searched area, using velocity fields from previous iterations to warp the second image

2 Optical flow

Computer vision often requires determination of displacement fields between successive video frames for motion detection, object segmentation, video compression, and as an aid to 3D reconstruction of the field of view. Although the objective of determining a displacement field is fundamentally the same as PIV, optical flow's computer-vision roots have led to less rigorous algorithms better suited to real-time processing. Among several other approaches, intensity gradient descent methods have become especially popular.

2.1 Lucas-Kanade method

One such method is Lucas-Kanade, which involves solving the following system for (U,V), where I is the intensity at each pixel q (with subscripts denoting differentiation):

$$\begin{bmatrix} I_x(q) & I_y(q) \end{bmatrix} \begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} -I_t(q) \end{bmatrix}$$

This requires the least-squares solution of a system of two variables at each pixel, with summation over a Gaussian-weighted interrogation window centered around the pixel:

$$\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_y I_x & \sum I_y I_y \end{bmatrix}^{-1} \begin{bmatrix} -\sum I_x I_t \\ -\sum I_y I_t \end{bmatrix}$$

Convergence can be stymied by local minima and large displacements, both of which are commonly encountered in particle-seeded flow fields.

2.2 FOLKI (Iterative Lucas-Kanade Optical Flow)

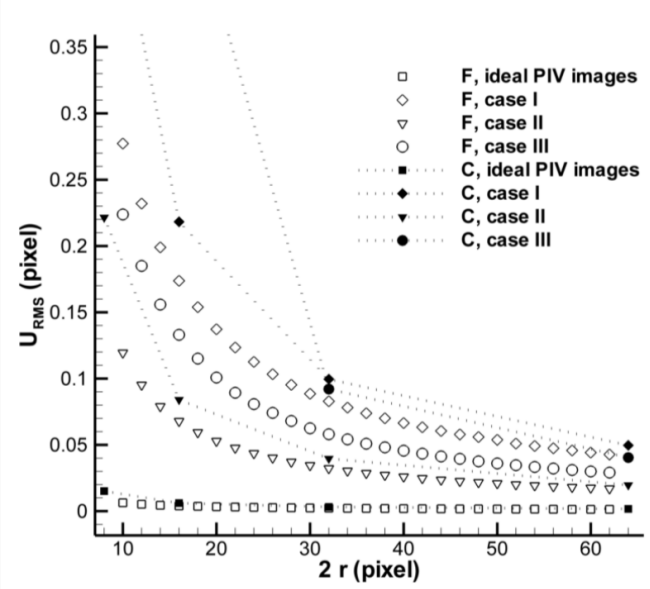


Figure 2: Comparison of root mean square displacement error U_{RMS} versus interrogation window size r for FOLKI (F) and conventional FFT-based PIV (C) for several representative test cases [3]

To circumvent the aforementioned issues, a group associated with ONERA, the French aerospace research directorate, developed FOLKI. At its core, FOLKI involves downscaling the input images by a factor of 2^n, determining displacement fields at that scale, and mapping the downscaled velocity fields as initial guesses for the next-highest scale. This process is repeated until the desired field resolution is attained. Additionally, the displacement field at each scale is computed iteratively, with the time derivative of image intensity calculated using an advectively-resampled second image.

Several additional adaptations were made to increase the algorithm's accuracy and suitability for PIV images, including computing the sum of squared differences in a time-symmetric fashion, and increasing the robustness to varying illumination by zero-normalizing the mean and standard deviation of each interrogation window's intensity. Due to the difficulty of accurately computing gradients and Gaussian averaging around borders, velocity vectors are subjected to global outlier rejection. These adaptations increase the computation time by a factor of 10 or more compared to the unmodified Lucas-Kanade method, but they increase the accuracy of the resulting vector fields to a degree surpassing conventional PIV algorithms [3] while retaining a speedup of several orders of magnitude.

3 Julia image processing & GPU capabilities

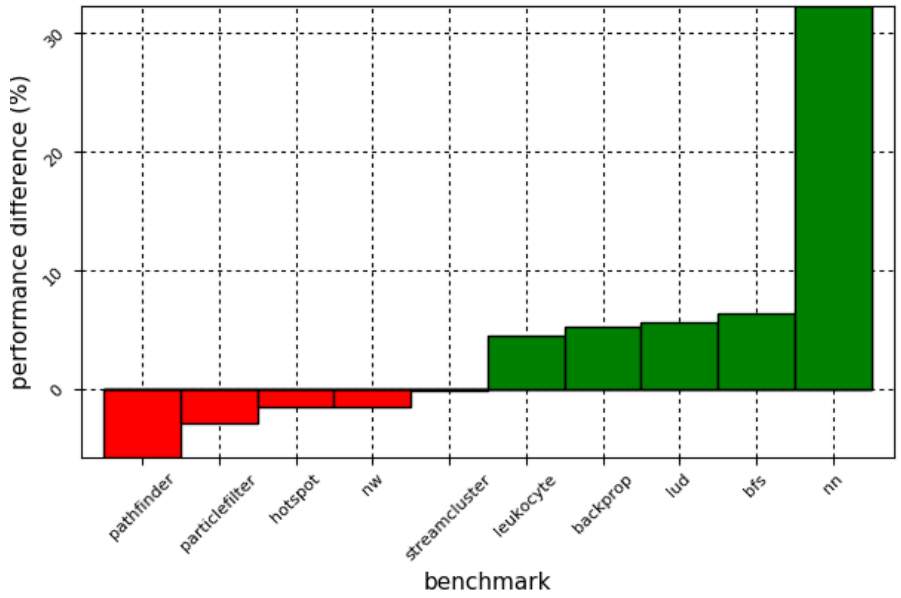


Figure 3: Performance difference between Julia and C++ kernels on the Rodina benchmark suite using a GTX 1080 and CUDA 8.0.61 [4]

Julia is a recently-released dynamic language with a variety of attractive features for scientific computing. The syntax is similar to both Matlab and Python, with a thriving user-driven package ecosystem. Its sophisticated type system allows optimization even for user-defined functions and types; one doesn't need to stick to vectorization or arrays of doubles for speed as with Matlab or Numpy. Underneath the hood, function call chains are transpiled to LLVM/Clang, which produces efficient bytecode for virtually all modern platforms.

Thanks to recent efforts by Google developers to allow transpilation from LLVM to CUDA PTX, it is also possible to write and interactively dispatch CUDA kernels directly in Julia using native syntax. This includes code for specialized types such as dual numbers, which allows for efficient computation of function derivatives through forward-mode automatic differentiation.

Additionally, Julia has built-in support for calling C/C++/Fortran libraries with virtually no overhead, enabling bindings to the C API for both CUDA and OpenCL.

4 Implementing FOLKI in Julia

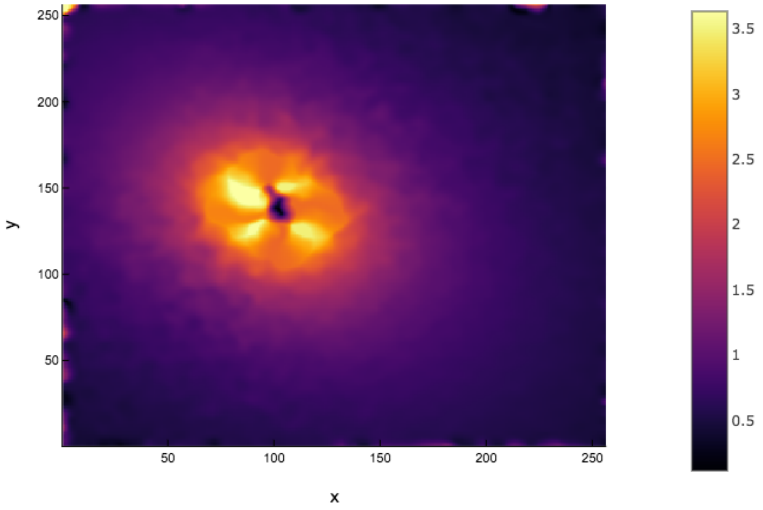
The FOLKI algorithm relies heavily on image processing techniques, including convolution, interpolation, and matrix multiplication, and is therefore well-suited to GPU acceleration; Champagnat *et al.* were able to achieve processing rates of up to five image pairs per second with their CUDA implementation. Because of OpenCL's ability to use whatever resources are available (whether CPU, integrated GPU, or discrete GPU), and the heterogeneous nature of the computing platforms for which this implementation is written (personal laptops and desktops as well as lab workstations), OpenCL was chosen as the intended accelerator backend for the present work.

Julia's image processing facilities have undergone heavy development over the past several years, and feature parity has been achieved with Matlab and SciPy for most common operations. Despite this, little attention has been paid to GPU acceleration of these operations until quite recently. The intended course of action for FOLKI development in Julia was thus:

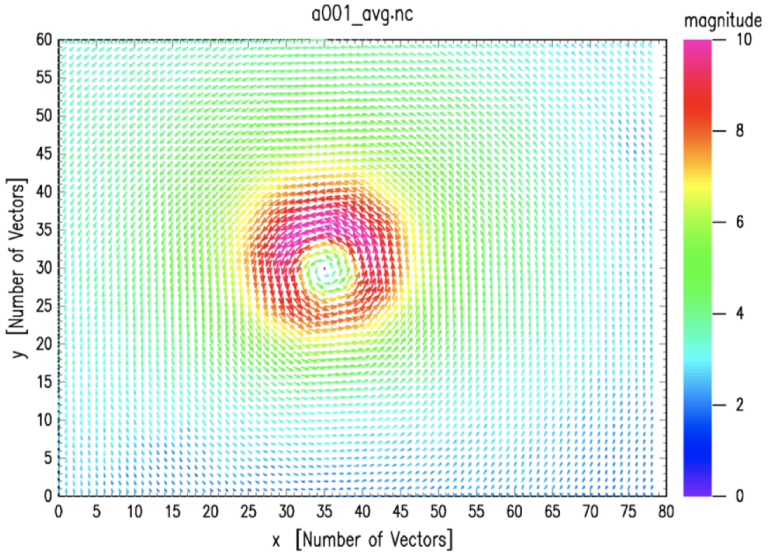
1. Implement a performant, feature-complete version of the FOLKI algorithm using native image processing libraries
2. Identify expensive portions of the computation and accelerate using OpenCL kernels
3. Implement the full algorithm using OpenCL kernels written in Julia, mirroring the CUDA effort of Champagnat *et al.*

However, thanks to Champagnat's description of the algorithm (opaque and short on necessary detail) and various challenges resulting from idiosyncrasies of Julia's image processing library, virtually all development time was devoted to implementing FOLKI in a merely correct fashion, leaving no time for GPU acceleration.

4.1 Results



(a) Calculated velocity field magnitude



(b) Average of four velocity fields produced by conventional PIV correlators [1]

Figure 4: Julia-FOLKI results for synthetic test images

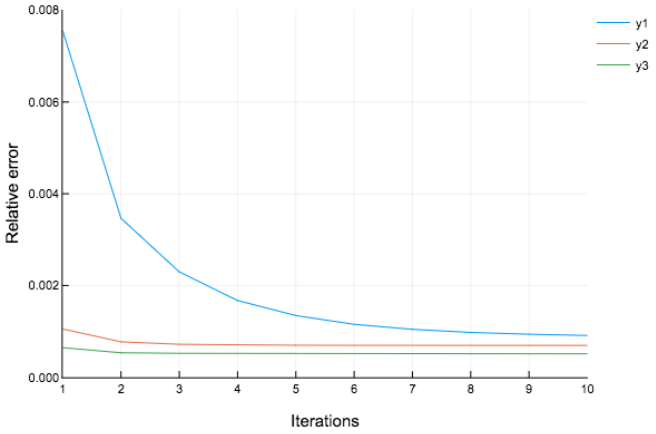


Figure 5: Relative error convergence for $N_{\text{levels}} = 3$, $N_{\text{iterations}} = 10$

Although the Julia implementation fell short on GPU acceleration, the naïve, single-threaded approach proved both performant and robust, delivering displacement fields in a matter of seconds rather the minutes or hours seen with conventional FFT-based algorithms. Further attention must be paid to boundary-value handling due to the impossibility of tracking particles which pass into or out of the field of view between the first and second images. This is a minor difficulty, though, since experimental images frame the region of interest in the center of the field of view, allowing spurious velocity vectors near the borders to be discarded.

5 Conclusions

Due to the intended use of Julia-FOLKI as a research tool and the consequent importance of obtaining accurate results, improvements in robustness and accuracy were prioritized over GPU acceleration. To paraphrase a quote attributed to Kent Beck, "Make it work, make it right, make it fast". Unfortunately, I only got as far as "make it right", and this project is intended to reveal competence (or lack thereof) in the concepts learned in ME759; I ran out of time to accelerate the computation using OpenCL as originally intended.

However, my doctoral research relies on the results of particle image velocimetry, and development of Julia-FOLKI will continue. Immediately, work is still needed to arrive at reliable choices of parameters for filter sizing, required iterations, and vector validation, but once the algorithm is sufficiently robust to satisfy research needs, further speed-up should be easily within reach. Per profiling of the naïve algorithm, at least 80% of processing time is spent calculating matrix multiplications (both for filter convolution and for the solution of linear systems) and interpolations (for image resampling). Matrix multiplication is readily accelerated using GPU kernels, and many GPUs are capable of hardware-accelerated texture interpolation. Julia's OpenCL binding isn't yet capable of leveraging texture interpretation (I asked), but development is ongoing, and Julia-FOLKI should be fully-accelerated within several months.

References

[1] M. Stanislas, K. Okamoto, and C. Kähler, "Main results of the first international piv challenge," *Measurement Science and Technology*, vol. 14, no. 10, p. R63, 2003.

[2] O. Pust, "Piv: Direct cross-correlation compared with fft-based cross-correlation," in *Proceedings of the 10th International Symposium on Applications of Laser Techniques to Fluid Mechanics, Lisbon, Portugal*, vol. 27, 2000, p. 114.

[3] F. Champagnat, A. Plyer, G. Le Besnerais, B. Leclaire, S. Davoust, and Y. Le Sant, "Fast and accurate piv computation using highly parallel iterative correlation maximization," *Experiments in fluids*, vol. 50, no. 4, p. 1169, 2011.

[4] T. Besard. (2017, October) High-performance gpu computing in the julia programming language. [Online]. Available: https://devblogs.nvidia.com/parallelforall/gpu-computing-julia-programming-language/