International Journal of Computational Intelligence Systems, Vol. 7, No. 2 (April 2014), 213-224

Curve-Fitting on Graphics Processors Using Particle Swarm Optimization

R. T. Kneusel,[1]

[1] *Professional Services Group, Exelis Visual Information Solutions,*
4990 Pearl East Circle,
Boulder, Colorado 80301 USA
E-mail: rkneusel@ittvis.com

Received 5 September 2011

Accepted 12 August 2012

Abstract

Curve fitting is a fundamental task in many research fields. In this paper we present results demonstrating the fitting of 2D images using CUDA (compute unified device architecture) on NVIDIA graphics processors via particle swarm optimization (PSO). Particle swarm optimization is particularly well-suited to implementation on graphics processors using CUDA as each CUDA thread can be made to model a single particle in a swarm with the swarm itself defined by thread blocks.
The motivation for this work was the reconstruction of interferometric photoactivated localization microscopy (iPALM) data sets. The reconstruction requires the fitting of 2D curves to potentially millions of detected photoactivation peaks. Additional motivation was to search for a solution that replaces a cluster with a single desktop machine using multiple CUDA graphics cards.
PSO curve fitting running on the GPU enabled a substantial performance increase over the CPU alone and scaled well with multiple CUDA cards. The performance gains increase with the number of images to be fit and the number of cards used. Two NVIDIA Tesla C1060 graphics cards achieved performance comparable to 30 nodes of the cluster.

Keywords: GPU, CUDA, particle swarm optimization, curve fitting

1. Introduction

Interferometric photoactivated localization microscopy (iPALM) is a novel light microscopy technique for imaging cellular ultrastructure. iPALM can generate 3D images of the distribution of properly tagged molecules [1]. A key step in the iPALM reconstruction process requires fitting images representing detected photoactivation peaks to a 2D Gaussian surface. This fitting must be done for each of the hundreds of thousands to several millions of peaks in the iPALM data.

The existing iPALM reconstruction and 3D visualization, implemented in IDL (*Interactive Data Language*, Exelis Visual Information Solutions, Boulder, CO, USA), makes use of about 100 nodes of the Janelia Farm cluster (Howard Hughes Medical Institute, Janelia Farm Research Campus, Ashburn, VA, USA). The cluster runs IDL sessions to do the curve fitting. It is desired, to make iPALM microscopes more accessible to other researchers, to replace the cluster with NVIDIA graphics cards using CUDA;

this work describes a step towards that goal.

The fitting of a 2D Gaussian to data represented as an image is certainly not new or novel. What is novel about this work is the particle swarm optimization curve-fitting algorithm, under CUDA, which we name *PSFIT*. The results section will show that this algorithm plays the performance gains possible with the GPU against the inherent slowness of the particle swarm approach to curve-fitting in order to arrive at a solution that meets the particular goal of liberating the iPALM reconstruction from the cluster. The *PSFIT* algorithm itself is general and applicable to any situation where the same function must be fit to many thousands or more data sets.

Particle swarm optimization (PSO) [2] is an algorithm that searches through a space, in this case the space of function parameters, in order to locate the global minimum or maximum. It is modeled after the swarm movements of flocks of birds or schools of fish. PSO has been successfully applied in a vast multitude of areas from neural networks [3] and medical image registration [4] to electrical power systems [5] and feature selection for classification [6]. In particular, PSO has been applied to curve fitting [7].

Graphics processors offer a unique environment for the implementation of highly parallel algorithms, especially through the use of the CUDA architecture. As with PSO, a large number of applications have been found for the GPU, including GPU-based PSO [8] [9]. The work in these last two references is particularly relevant and differences between these approaches to PSO on the GPU and that of the present paper will be discussed below.

Next we describe the actual implementation of the *PSFIT* algorithm. After this we present results demonstrating the performance gains achieved with the CUDA implementation. A discussion of the results, including limitations, and the general applicability of the *PSFIT* algorithm, follows. Finally, we conclude with some thoughts on possible future directions for this work.

2. Methods

For this work we use the simplest, or canonical, PSO algorithm. The canonical PSO algorithm is outlined in Figure 1. In the continuous parameter case, PSO populates the parameter space with a swarm of "particles" where the particles can be viewed as points in the n-dimensional parameter space. The goal of PSO is to move these particles through the parameter space in a way that searches for the global minimum (or maximum, here we only consider minima) without becoming trapped in local minima that might stymie other optimization techniques.

In applying PSO to curve fitting, we combine the usual least-squares metric with PSO searching. In this case, each particle in the swarm represents a candidate curve with the objective function measuring the deviation between the candidate curve and the supplied data in a least-squares sense. As in normal least-squares, this deviation is to be minimized. With the swarm, this minimization is found in the tension between the best position found by each particle and that of the entire swarm (the global best, in Figure 1). In this work, the photoactivation peak images, each localized to an 11x11 pixel region, were fit to,

$$z(x,y) = p_0 + p_1 exp(-\frac{1}{2}((\frac{x-p_4}{p_2})^2 + (\frac{y-p_5}{p_3})^2))$$

in order to determine the peak location, (p_4, p_5) and spread around the peak, p_2^2 and p_3^2.

CUDA requires the same operations to be performed by each thread of the GPU. The threads are grouped into blocks of up to 512 threads. Each thread in a block has access to up to 16k bytes of shared memory but it is not possible for threads to communicate with threads outside their own block. Blocks are conceptually arranged in a grid, according to the design of the problem.

To implement PSO under CUDA we assigned each swarm particle to one thread and defined each swarm as one block. The number of particles was fixed at 256 to make use of that many threads per block. This number worked well as it maximized the GPU usage according to NVIDIA's occupancy calculator. Therefore, each block represented a swarm

1. *Given*
   (a) $\vec{x}_i$ - a particle position in the search space, $i = 1 \ldots n$
   (b) $\vec{v}_i$ - a velocity associated with each $\vec{x}_i$
2. *Initialization for each* $\vec{x}_i$
   (a) $\vec{x}_i = (\vec{b}_{max} - \vec{b}_{min})\vec{U} + \vec{b}_{min}$
   (b) $\vec{v}_i = \vec{0}$
   (c) $\widehat{x}_i = \vec{x}_i$ - the particle's best position
   (d) $\vec{g} = \vec{x}_i$, if $f(\vec{x}_i) < f(\vec{g})$ - swarm best position
3. *Iteration for each* $\vec{x}_i$
   (a) $\vec{v}_i \leftarrow \omega\vec{v}_i + c_1\vec{r}_1(\widehat{x}_i - \vec{x}_i) + c_2\vec{r}_2(\vec{g} - \vec{x}_i)$
   (b) $\vec{x}_i \leftarrow \vec{x}_i + \vec{v}_i$
   (c) $\widehat{x}_i = \vec{x}_i$, if $f(\vec{x}_i) < f(\widehat{x}_i)$
   (d) $\vec{g} = \vec{x}_i$, if $f(\vec{x}_i) < f(\vec{g})$
4. *Repeat iteration step until convergence*

Figure 1: The canonical particle swarm optimization algorithm. The algorithm conducts a search of the space representing the function to be minimized (or maximized) by moving a swarm of particles through the space and testing the function value at each particle. The tension between the best location found by the swarm and the best location found by each individual particle directs the search. Here $\vec{b}_{min}$ and $\vec{b}_{max}$ define the bounds of the search space, minimum and maximum, for each dimension, while ω, c_1, and c_2 are constants with typical values of 0.9, 2.0, and 2.0 respectively. Finally, $\vec{U}$, $\vec{r}_1$ and $\vec{r}_2$ are random vectors with components drawn uniformly from the range $[0,1)$ and $f()$ is the fitness function to be minimized (or maximized). The PSFIT algorithm uses the swarm to minimize the mean-squared error between the given function values (the image intensity values) and those of a 2D Gaussian with the components of the particle position acting as the parameters of the fit function.

dedicated to fitting a single photoactivation peak image. The blocks were laid out as a grid of n rows by 128 columns. The number of rows was a function of the number of input images and was set to a multiple of 128 in order to efficiently access GPU global memory. This arrangement scaled readily to other GPU configurations and enabled the straightforward addition of multiple GPU cards.

The heart of the *PSFIT* algorithm is the *k_particle* CUDA kernel which is outlined in Figure 2.

Initialization

The canonical PSO algorithm sets the initial particle velocities to zero and the initial particle positions (in parameter space) to random values. In this case, the particle positions are restricted to a specified domain which was determined empirically from Gaussian fits to test images. This restriction prevents particles from flying off wildly and also limits the search space to better improve the probability of swarm convergence. The allowed particle velocities are also restricted to prevent excessively rapid particle motion. The reduced χ^2 for this initial position is used as the function value for the initial best position of the particle. Since all particles are implemented as threads, these operations in code become simple updates to individual components of the velocity or position vectors for a single particle; "looping" over particles is handled by the thread scheduler inside the graphics card hardware.

The swarm best, the best position found by any particle (thread), is computed at this point. This is an operation over threads, therefore, only one thread is used for this calculation. Since the positions and velocities for each swarm are stored in shared memory, a single thread can access all the particle best positions and determine which of them has the lowest χ^2 value. All other threads in the block must of necessity sit idle while this updated swarm best is found. Afterwards, the threads are synchronized and execution begins again in unison. Thread specific operations are used in several other key places in the kernel.

Iteration

For any one particle in the swarm, iteration is the

1. *Initialization*
    (a) Set particle velocity components to zero.
    (b) Set particle position components to a random value within the given constraints (this becomes the initial particle best position).
    (c) Calculate the χ^2 for this position.
    (d) Set the swarm best as the best of all the threads in the block.
2. *Iteration*
    (a) Update each velocity component restricting to $\pm V_{max}$.
    (b) Update each position component restricting to the given domain.
    (c) Calculate the χ^2 for the new position, updating the particle best, if necessary.
    (d) Update the swarm best position and list of last n swarm best positions, if necessary.
3. *Output*
    (a) Copy swarm best position to output memory.
    (b) Update the standard deviation of the last n swarm best positions for each fit parameter.

Figure 2: Outline of the *k_particle* CUDA kernel. This kernel represents an individual particle as a thread of a block where each block represents a swarm. Each swarm fits a single input image to a 2D Gaussian returning the fit parameters and estimates of their uncertainties.

maximum number of times that the swarm will be allowed to update itself in search of the global minimum. This idea is key for the CUDA implementation, namely, that each particle is restricted to a pre-specified number of steps in searching the parameter space. This imposes uniform behavior on each particle, and by extension each swarm, in its search for the best fit parameters and allows for efficient parallel fitting of many thousands to hundreds of thousands of images simultaneously. The number of swarm iterations necessary to converge is dependent upon the function being fit. In the case of 2D Gaussians, the effect of the number of iterations, I_{max}, was investigated and a value of 25 to 30 was deemed proper.

For an individual particle, the velocity component was updated at each iteration according to the canonical algorithm given in Figure 1. Each particle requires several pseudo-random values at this point, as in the initial placement of the particles in the search space. Each thread made use of the Park and Miller *MINSTD* pseudo-random number generator [10]. Once the new velocity of the particle is calculated, it is simply added to the existing position components to arrive at the new position. The terms "velocity" and "position" are, of course, used loosely in this context.

Once a new position is calculated, each component is checked to ensure that it is within the allowed domain. If it is not, the offending component is set to the boundary value. After this, the χ^2 is calculated using the particle position as the parameter values of the fit. Next, each iteration concludes by updating the particle best position if the newly calculated χ^2 is less than any previously discovered by the particle. Finally, an update of the swarm best position is done, if warranted. As in the initialization step, only thread zero of the swarm is used when checking all particles for a new swarm best.

As part of the search for a new swarm best position, the list of n previous swarm best positions is updated by pushing the new best at the end and dropping the oldest best off the front. These values are tracked for the swarm in order to calculate un-

certainties for the final parameter values.

Output

The result of the parameter search consists of the six parameter values, associated uncertainties and one reduced χ^2. The parameter values and reduced χ^2 are copied to output memory by the first seven threads of each swarm, each one copying a single value. The parameter uncertainties are estimated by the standard deviation of the last n swarm best positions and copied to output memory as well by thread zero.

Random number generation

The *PSFIT* algorithm needs to have access to many pseudo-random values, here simply referred to as "random values" with the understanding that they are generated via some algorithm. Random values are necessary for the initialization of the swarm and the determination of initial parameter values (particle positions). Each iteration of the swarm requires two additional random values for each particle (thread). The highly parallel nature of the GPU makes serial determination of random values difficult in practice. For *PSFIT*, the two areas that need random values can be cleanly separated. The problem becomes one of selecting pseudo-random number generation algorithms and corresponding seed values. For swarm initialization, 32-bit integer seed values were determined rapidly on the CPU, prior to starting the kernel as the number of swarms was known (the number of images to be fit). These were calculated from a single hybrid-Tausworthe generator [11] using the current Unix system time value as a seed. On the GPU, each thread managed its own Park and Miller *MINSTD* generator by storing the seed value in a register variable, the initial seed selected from those passed to the GPU from the CPU.

The hybrid Tausworthe generator used on the CPU is given in Figure 3. The seed value was the integer returned by the Unix `time` function, though it could easily be fixed to produce a completely deterministic execution environment.

This algorithm was selected for seed generation

```
unsigned int z1 = 0xff32422;
unsigned int z2 = 0xee03202;
unsigned int z3 = 0xcc23423;
unsigned int z4 = 0x1235;

unsigned int TausStep(unsigned *z, int S1,
                      int S2, int S3,
                      unsigned int M) {
  unsigned int b = ((*z << S1) ^ *z) << S2;
  *z = (((*z & M) << S3) ^ b);
  return *z;
}

unsigned int LCGStep(unsigned int *z,
                     unsigned int A,
                     unsigned int C) {
  *z = (A*(*z)+C);
  return *z;
}

unsigned int HybridTaus() {
  return TausStep(&z1, 13, 19, 12, 4294967294UL) ^
         TausStep(&z2, 2, 25, 4, 4294967288UL) ^
         TausStep(&z3, 3, 11, 17, 4294967280UL) ^
         LCGStep(&z4, 1664525, 1013904223UL);
}
```

Figure 3: The hybrid Tausworthe generator used to create seed values for the GPU. This code was run on the CPU and very quickly generated a vector of 32-bit integer seed values, one for each particle of each swarm. The variable `z4` is the seed for the linear congruential step and was used as the seed for the entire generator.

because of its favorable characteristics. Since it is running on the CPU, it must be fast, which it is; there are no divisions (direct or via modulo arithmetic) and no floating point operations. Even the linear congruential step is implemented so as to avoid divisions. Also, the generator has a high period, on the order of 2^{121}, and it is of a different class of algorithms when compared to the *MINSTD* for which it is generating seeds.

The *MINSTD* algorithm has been in use for a long time. It is, as its name suggests, a minimum standard and uses the linear congruential approach to generating random values. It has known issues [12], but in this application it was selected for two

```
__device__ float rnd(unsigned int *seed) {
    *seed = 16807*(*seed) % 2147483647;
    return 4.6566128730773926e-10 * (float)(*seed);
}
```

Figure 4: *MINSTD* as implemented in CUDA. The seed value is passed on each call and updated. Multiplication by the constant saves a floating point division by the maximum value of 2147483648. The `__device__` label extends standard C to tell the NVIDIA compiler that this function is to be compiled for the GPU and not the CPU.

reasons. First, its implementation is straightforward, thereby reducing the size of the code, and second, for any given swarm, the number of random values necessary, for any particular thread, is on the order of a few hundred, not millions or tens of millions. The C implementation, as used in *PSFIT*, is given in Figure 4.

Parameter uncertainties

PSO searches a space attempting to minimize or maximize some objective function. The output of the search is a point in the search space that, according to some criteria, mets this goal of minimizing or maximizing. There is no concept of how well this goal has been met, no concept of the uncertainty of the point selected. To apply PSO to curve fitting, where the point located represents the parameter values for the fit function, one would like to have an uncertainty associated with each value, therefore, the *PSFIT* algorithm estimates the uncertainty of each value by tracking the last n swarm best positions and, when iteration has ceased, uses the standard deviation of these values as conservative estimates of the uncertainty in the position selected. This empirical approach tracks well with uncertainties calculated for parameters located using traditional curve fitting algorithms, at least for the 2D Gaussian functions used here.

Integration with reconstruction software

The iPALM reconstruction software is written in the IDL language. In order to make use of the CUDA portion of the reconstruction, it was necessary to extend the IDL language via a module written in C. This module acts as a bridge between IDL and CUDA, converting IDL arrays to values in GPU memory and *vice versa* with the output. Additionally, the C module implements the hybrid Tausworthe pseudo-random number generator described above to calculate seed values for the particles in all the swarms. Lastly, the C module coordinates the execution of kernels on multiple GPU cards, if present in the system.

3. Results

Random Seed Generation

The many thousands of *MINSTD* generators used by the GPU threads each need a 32-bit seed value. The seed value, as described above, was supplied by a vector of seed values pre-computed for each run on the CPU using the hybrid Tausworthe generator. If this approach of using one pseudo-random generator to initialize a second one is to be successful, it should be expected that the output of the parallel random number generators ought to pass statistical tests associated with parallel generators. One of these tests is the block test [13].

In the block test, a set of samples is collected. The samples are drawn from a number of streams, each of which is a separate *MINSTD* generator seeded with a value drawn from a single hybrid Tausworthe generator. For each stream, a group of individual random values is collected. The resulting set of values, then, should be normally distributed if the streams are statistically independent. This process is shown graphically in Figure 5. In our tests, the number of samples in a set was 20,000. There were 500 streams each with a group size of three. The number of sets tested was 100 with a set being considered normally distributed if its Jarque-Bera test statistic [14] was below a cutoff of 5.99 ($p > 0.05$ to accept the null hypothesis of normality). With these parameters, 94% of the samples created from the combined hybrid Tausworthe and *MINSTD* generator passed the

block test. For comparison, only 4% of samples created from *MINSTD* generators seeded with values from another *MINSTD* generator passed the test.

Effect of Iteration Limits

The number of swarm iterations is a key factor in the success of the *PSFIT* algorithm. If the number of iterations, which must be fixed for all images being fit, is too few, the swarm is not able to converge to a meaningful solution. However, if the number of iterations is too large, the execution time increased so as to erase the benefit of using the GPU since the algorithm of $O(N_{iter})$.

The fits produced by the *PSFIT* algorithm were compared to those produced by the IDL `CURVEFIT` routine. This routine is based on the `CURFIT` routine in Bevington [15] and is a standard gradient-descent, nonlinear, curve fitting algorithm. In this work, the output of `CURVEFIT` is taken to be a gold standard. The comparison considers the fit parameters to be a point in a six-dimensional space and calculates the Euclidean distance between the two sets of parameters as a function of the number of swarm iterations. Specifically, 50,000 images were fit with *PSFIT* and `CURVEFIT` for each swarm iteration value. The Euclidean distance between the corresponding fits was calculated and the median of this set was plotted versus the iteration limit as in Figure 6. As can be, the swarms quickly converge to values virtually identical to those found by `CURVEFIT`. For the iPALM reconstruction, the iteration limit was set to between 25 and 30.

The *PSFIT* algorithm running on two Tesla cards achieved performance comparable to some 30 nodes of the Janelia Farm Linux cluster [16]. This is an encouraging report as a key goal of this work is to eventually replace the cluster with a single desktop machine using GPUs.

For a test sample of 50,000 images, 30 swarm iterations per image, the median parameter distance when compared to *CURVEFIT* was 0.844 indicating excellent agreement in the majority of cases. It should be noted that 6.3% of the test images failed to fit using *CURVEFIT*. The failures can be explained by the need to provide initial guess values for the parameters which occassionally lead to unstable results. Particle swarm optimization is a population-based optimization algorithm and as such does not require initial guesses. This view gives insight on why it is more robust than simple gradient descent, even for a straightforward function like a 2D Gaussian.

Parameter	$\bar{x}$	95% CI
x_{center}	4.57196	[4.57194, 4.57198]
y_{center}	4.81209	[4.81205, 4.81213]

Table 1: Mean and 95% confidence intervals for the calculated 2D Gaussian peak position for 100,000 fits of a single peak image.

The stability of the *PSFIT* algorithm was examined. A single photoactivation peak image was fit 100,000 times. The large number of repetitions ensures that the 95% confidence intervals for the parameter mean values is very tight as can be seen for the center position of the 2D Gaussian in Table 1.

Performance Versus Number of Images

Another key performance factor is the number of images to be fit. Figure 7 shows the run time performance for *PSFIT* and *CURVEFIT* as a function of the number of images to fit. The *PSFIT* performance includes the time to transfer data to and from the Tesla C1060. The *CURVEFIT* results were run on a Dell computer (Intel 2.66 GHz, 2 cores, 4 GB RAM, Red Hat Linux 4.3, IDL 6.4). As can be seen, the CPU case quickly becomes very time consuming and makes iPALM image reconstruction tedious if done solely on a desktop PC. The slight performance gain seen by changing the number of swarm iterations per image from 30 to 25 needs to be balanced against the imprecision of the results (see Figure 6). For iPALM reconstruction this imprecision is acceptable.

Empirically-Derived Parameter Uncertainties

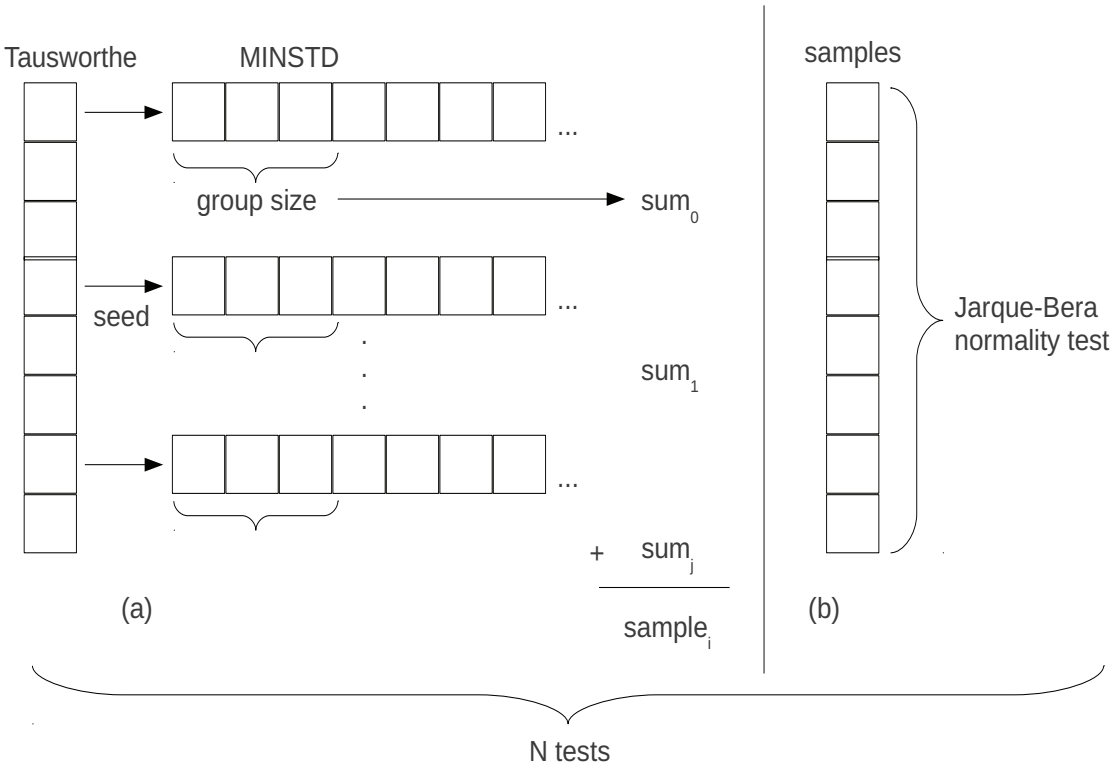

Figure 5: The structure of the block test. (a) Streams of MINSTD generators seeded from a single hybrid Tausworthe generator. Sequential samples for each MINSTD, of group size, are totaled to create a single sum value. This is repeated for each MINSTD stream. The total of the sum values forms a single sample. (b) The collection of samples is given to the Jarque-Bera test for normality as described in the text. The entire process, (a) and (b), is repeated to form N tests, always with a new hybrid Tausworthe generator seeded with the Unix system time.

The empirically-derived *PSFIT* parameter uncertainties were compared to the *CURVEFIT* parameter uncertainties in a maner analogous to the parameter values themselves. Figure 8 shows the Euclidean distance between the *PSFIT* and *CURVEFIT* uncertainty values, again viewing each as a point in a six-dimensional space, as a function of the number of swarm iterations. As the second plot shows, the minimum for the 2D Gaussians fit in this case, occurs around 33 iterations reaching a measured minimum of 0.295. After this point, the distance increases slightly before leveling off at a mean value of 0.655. The reason for this decrease followed by an increase may be due to zigzagging [17], a known effect where the particle swarm oscillates about the minimum point.

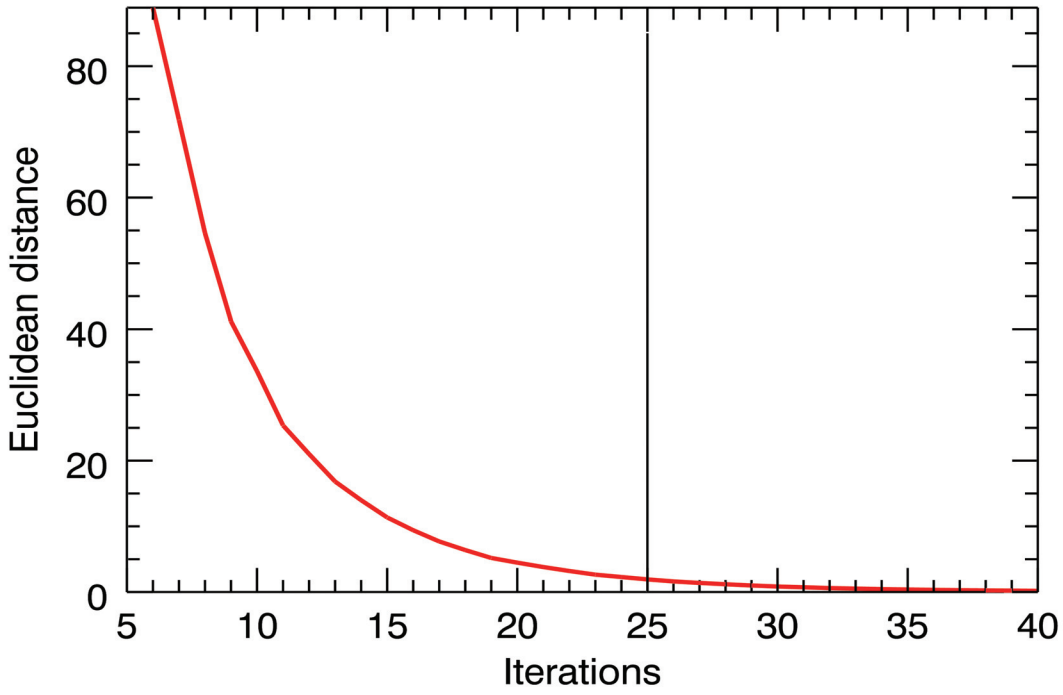

Figure 6: The median distance between the parameters found by *PSFIT* and those found by `CURVEFIT` for 50,000 photoactivation peak images as a function of the number of swarm iterations. The vertical line marks the iteration limit used in iPALM reconstruction.

4. Discussion

A pure IDL implementation of the *PSFIT* algorithm runs approximately 1000x slower than the IDL intrinsic *CURVEFIT* for the same 2D Gaussian (30 parameter iterations, dual core 2.4 GHz Intel CPU, 2 GB RAM, Fedora 12 Linux) so the performance gain seen by using CUDA is due to the brute force speed of the GPU and the number of simultaneous fits that it can do. This fact helps determine when the *PSFIT* algorithm is appropriate, namely, in cases where there are many, many fits of the same kind. If only a few or even a few hundreds of fits are needed, the overhead is such that another approach is likely to be more performant.

The number of iterations of the swarm necessary for convergence must be determined empirically. It is directly related to the complexity of the fit function. In this present case, the number of necessary iterations to achieve good convergence was found to be small, a definite advantage. Like the number of fits to be performed, the iteration limit must also be kept in mind when evaluating this approach for a particular task.

Standard curve fitting algorithms return some notion of the uncertainty in the parameters. The empirical approach to parameter uncertainty used in *PSFIT* tracks well with these uncertainties. Placing this approach on a surer mathematical foundation is an area for future work.

The *PSFIT* algorithm scales directly with the number of CUDA cards in the system. Improvements in CUDA hardware, therefore, will immediately translate into improvements in the performance of the algorithm thereby ensuring that the appropriateness of this approach will not fade with time, in the near

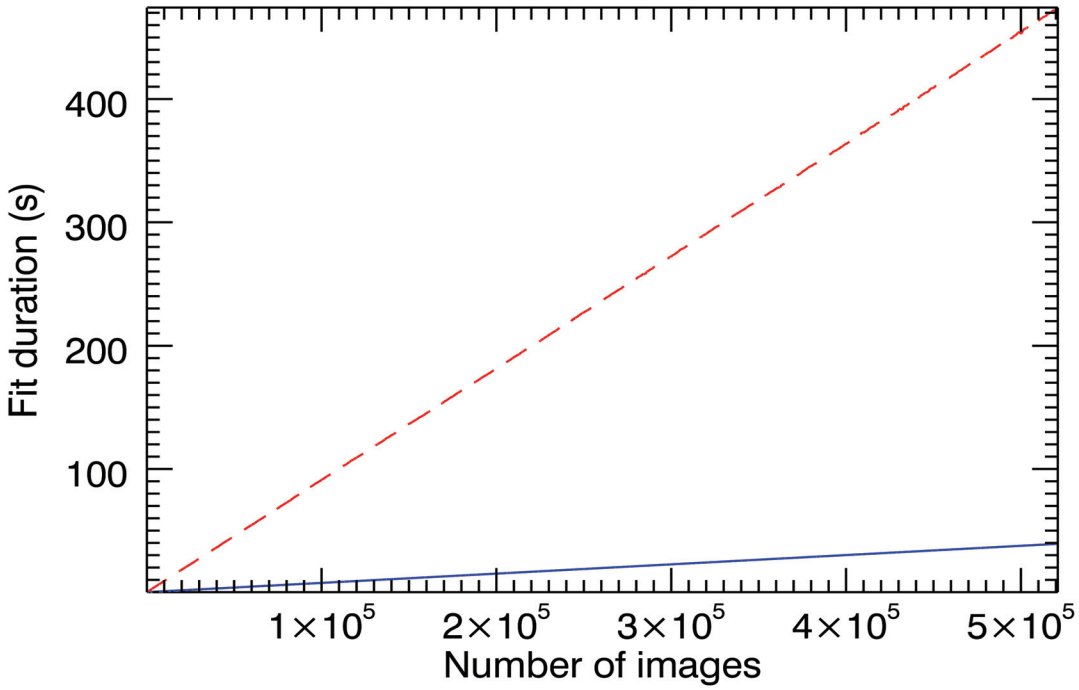

Figure 7: 2D Gaussian fit execution time as a function of the number of images for both *CURVEFIT* (dashed) and *PSFIT* (solid). The CPU execution time quickly becomes unacceptable while the GPU runtime increases much more slowly. The PSFIT iteration limit was set to 25. Test performed on a Pentium 4 computer (Intel 2.66 GHz, 2 cores, 4 GB RAM, Red Hat Linux 4.3, IDL 6.4) with a GTX-280 NVIDIA GPU.

term, but only become more attractive.

The treatment of random number generation differs from that of other researchers. For example, Zhou and Tan [8] simply generate a large set of random numbers on the CPU and transfer them to the GPU before starting the swarm. Here, only the seeds for the specific particles of the swarms are generated ahead of time. Mussi and Cagnoni [9] make use of the Mersenne Twister kernel supplied by NVIDIA though they do not state in any detail how it was incorporated into the actual particle swarm kernel.

As shown above, the use of two separate random number generation algorithms, one for the seeds (hybrid Tausworthe) and one on the GPU for the particles (MINSTD) clearly provides excellent results. Srinivasan *et al* [13] warn that when using the same generator for each stream two conditions must be tested. First is the intra-stream correlation to show that the values of each stream are suitable random. This is already known for the MINSTD algorithm. The inter-stream correlation is shown above to be good via the block test. Therefore, the approach chosen here is a sound one.

Multicore CPU implementations, to speed the seed generation, have not been investigated at present. This is an area for future work, though it is clear that the scaling of multiple GPU cards will also apply to multiple CPUs.

5. Conclusion

In this work a particle swarm optimization based curve fitting algorithm was developed and implemented on NVIDIA GPUs using CUDA. This algo-

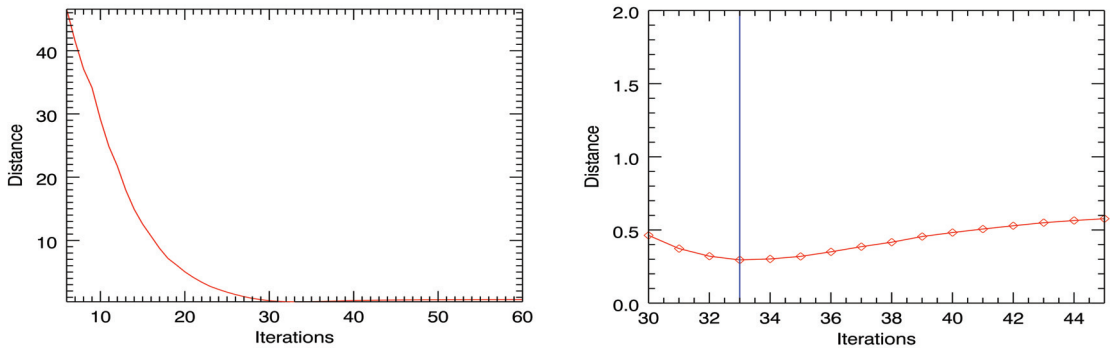

Figure 8: A comparison of the Euclidean distance between the parameter uncertainties returned by *CURVEFIT* and those of *PSFIT* as a function of the iteration limit. The same image was fit each time. The distance was used as a convergence of the uncertainties would be reflected as a decrease of the distance when the parameter uncertainties are treated as a point in a six-dimensional space. On the left, the full plot show the rapid convergence of the swarm with iteration limit. On the right, a close up of the region near the minimum shows a slight increase in the distance after 33 iterations (blue line) possibly due to the zig-zag effect as mentioned in the text.

rithm was applied to the fitting of many 2D Gaussians to photoactivation peak images as a key part of the iPALM reconstruction process. The resulting GPU implementation was substantially faster at fitting many hundreds of thousands to millions of peak images when compared to a single CPU. The GPU implementation was also effective when scaled to two NVIDIA Tesla cards achieving performance comparable to 30 nodes of a Linux cluster. The implementation required some care in the use of pseudo-random numbers, which are key elements of the particle swarm optimization algorithm. The algorithm itself is general and applicable to other functions to be fit. Future work will investigate this potential.

Acknowledgment

The author would like to thank Harald F. Hess and Gleb Shtengel, Howard Hughes Medical Institute, Janelia Farm Research Campus, Ashburn, VA 20147, for their helpful comments and assistance with this work.

1. G. Shtengel, *et. al*. *Interferometric fluorescent super-resolution microsopy resolves 3D cellular ultrastructure*, PNAS March 3, 2009, vol 106, no 9, 3125-3130.
2. J. Kennedy and R. Eberhart. *Particle Swarm Optimization*. Proc. IEEE Int'l. Conf. on Neural Networks (Perth, Australia), 1995.
3. R. Eberhart, J. Kennedy. *A New Optimizer Using Particle Swarm Theory*, IEEE Sixth Intl. Symp. on Micro Machine and Human Science, 39-43, 1995.
4. Y. Chen, A. Mimori. *Hybrid Particle Swarm Optimization for Medical Image Registration*, Fifth International Conference on Natural Computation, 26-30, August 2009.
5. M. R. AlRashidi, M. E. El-Hawary. *A Survey of Particle Swarm Optimization Applications in Electric Power Systems*, IEEE Transactions on Evolutionary Computation, vol 13, no 4, August 2009.
6. X. Wang, J. Yang, X. Teng, W. Xia, R. Jensen. *Feature selection based on rough sets and particle swarm optimization*, Pattern Recognition Letters, vol 28, 459-471, 2007.
7. H. Shinzawa, J. Jiang, M. Iwahashi, Y. Ozaki. *Robust Curve Fitting Method for Optical Spectra by Least Median Squares (LMedS) Estimator with Particle Swarm Optimization (PSO)*, Analytical Sciences, vol 23, 781-785, July 2007.
8. Y. Zhou, Y. Tan. *GPU-based Parallel Particle Swarm Optimization*, IEEE Congress on Evolutionary Computation, 1493-1500, 2009.
9. L. Mussi, S. Cagnoni. *Particle Swarm Optimization within the CUDA Architecture*, Genetic and Evolutionary Computation Conference, Montréal, Canada, July 2009.
10. S. K. Park, K. W. Miller. *Random Number Gener-*

ators: Good Ones are Hard to Find, Comm. of the ACM, vol 31, no 10, 1192-1201, October 1988.
11. L. Howes, D. Thomas. *Efficient Random Number Generation and Application using CUDA*, in *GPU Gems 3*, H. Nguyen, ed. Addison-Wesley Professional, 2007.
12. G. Marsaglia. *Remarks on Choosing and Implementing Random Number Generators*, Comm. of the ACM, vol 36, no 7, July 1993.
13. A. Srinvasan, M. Mascagni, D. Ceperley. *Testing Parallel Random Number Generators*, Parallel Computing (29), 69-94, 2003.
14. C. M. Jarque, A. K. Bera. *Efficient tests for normality, homoscedasticity and serial independence of regression residuals*, Economics Letters, vol 6, no 3, 255-259, 1980.
15. P. R. Bevington. *Data Reduction and Error Analysis for the Physical Sciences*, McGraw-Hill, New York, 237-240, 1969.
16. H. Hess, Howard Hughes Medical Institute, Janelia Farm Research Campus, Ashburn, VA, private communication, November 2009.
17. I. Trelea. *The particle swarm optimization algorithm: convergence analysis and parameter selection*, Information Processing Letters (85), 317-325, 2003.