Improving Performance of DAOPHOT-FIND by Using GPU Architecture

Rubén Hernández Pérez(✉), Ruslan Gabbasov, and Joel Suárez Cansíno

Centro de Investigación en Tecnologías de Información y Sistemas, Instituto de Ciencias Básicas e Ingeniería, Universidad Autónoma del Estado de Hidalgo, Carretera Pachuca–Tulancingo Km. 4.5, Cd. Universitaria, 42090 Mineral de la Reforma, HGO, Mexico
rub3n.hernandez.perez@gmail.com

Abstract. In astronomy there have been big changes in the availability of data, much of it provided freely and publicly via internet, allowing people to access the same data as used by professional astronomers for their own investigations. The data obtained from different telescopes have increased in size too, forcing the community to boost the performance of algorithms for image processing, pattern recognition, and, in particular, tasks for finding and characterizing astronomical objects. One of such packages is DAOPHOT designed to deal with crowded astronomical fields. However, the tasks are computationally intensive because they require the execution of many floating point operations. In order to face such computational challenge, we propose an implementation of DAOPHOT's particular task FIND, using massive parallel computation architecture supported by GPUs, which allows us to process large images at least two times faster. This work presents the analysis and comparison of time complexity between the implementations of the FIND algorithm both in CPU and GPU.

1 Introduction

Automatic source extraction from astronomical images started in the middle of the 60's, with the first automatic machines GALAXY in Edinburg [6] and APMS in Minneapolis [7] that allowed simple flux and position measurements better than what could be done by hand. Since then, software for detection and classification has evolved at a relatively slow pace, mainly because simple techniques meet most scientific specifications. Over the years the popularity of the difficult art of stellar photometry in crowded fields has derived in many computer programs which extract information from two-dimensional digital images. One of those computer programs is DAOPHOT [1], which continues being developed at the Dominion Astrophysical Observatory and allows performing tasks like finding objects by measuring the signal intensity enhancement over the background and Point Spread Function (PSF) fitting [5]. The program shows good accuracy and robustness but is relatively slow [2].

The increasing use of internet and massive databases allows anyone to have access to raw data for astronomy research. Web sites like SkyView [12] provide

I. Gitler and J. Klapp (Eds.): ISUM 2015, CCIS 595, pp. 240–249, 2016.
DOI: 10.1007/978-3-319-32243-8_17

images of any part of the observed sky at wavelengths ranging from Radio to Gamma-Ray. Through its simple web interface, it is possible to acquire images and to perform investigation by our own, but it is still necessary to support those investigations with packages and software to process and extract information from images. As mentioned before, DAOPHOT helps to perform such tasks and with the access to modern computer technology this process becomes increasingly easier and it is possible to analyse large amounts of data in a short period of time. However, it is necessary to make an effort to implement algorithms and routines in a way that exploits all capabilities of such technology.

The necessary operations for finding objects in astronomical images are computationally intensive, mainly because of the size of the images which increase due to the advance of technology of detectors. For example, the Large Synoptic Survey Telescope (LSST) is expected to produce several terabytes of data every night using its 3200 megapixel camera [3]. In addition, many automatic telescopes acquire raw data in real time and the matter of speed is crucial for image processing. Rapid processor development allowed the execution of a continuously increasing number of operations, and the processors became faster every year until 2005, when a physical limit was reached by the fact that energy consumption varies as the cube of clock rate [8], not only because of the operations of the processors themselves, but the energy needed to cool down the chip too. To overcome this obstacle, multi-core technology emerged as a new direction to improve performance without increasing the clock rate of processors. The implications of this new direction are found in the development of new software. The vast majority of the software written for astronomy before 2005 is designed to run sequentially, and for this reason it is necessary to make an effort to rewrite the algorithms to take full advantage of the capabilities of this new hardware.

Along with the development of multi-core CPUs, the powerful gaming industry had devised its own processors called Graphics Processing Units or GPUs, which are based on different principles than those of CPUs [8]. A GPU is an specialized processor that is equipped with a large number of special hardware units for mathematical and fast floating point operations. The primary goal of GPUs is to provide high-performance for gaming and rich 3D experience [9], but the demand for high performance computation has caused this processors to have a more general purpose, having recently burst onto the scientific computing scene as an innovative technology that has demonstrated substantial performance and energy efficiency improvements [10].

In this paper we present a modification of a popular stellar photometry package DAOPHOT that makes use of GPU hardware. In particular, the task FIND devoted to finding the stars in the images was optimized. We show that substantial time reduction can be obtained, although there is still space for improvement.

2 FIND Algorithm

DAOPHOT package deals with the difficult problem of performing accurate photometry in crowded fields. It consists of several routines and algorithms to perform tasks that include finding objects, aperture photometry, obtaining the point spread function among others. The focus of this work is on the first task mentioned, and we briefly describe the FIND algorithm. More details can be found in the documentation made by Stetson [1], the reference guide by Davis [4], and in the original code itself which is well commented. We use the latest version of Daophot II package (v1.3-6) included as a part of Starlink software available at [13].

The FIND algorithm attempts to find stars in a two-dimensional image by going through the image pixel by pixel asking the question, “If there is a star centred in this pixel, how bright is it?”. The answer for this question is estimated numerically by fitting a truncated Gaussian profile to the values in a surrounding sub-array of pixels (sub-picture). To explain the operations performed, let the brightness in the $(i, j) - pixel$ represented by $D_{i,j}$ and let G represent the unit-height, circular, bivariate Gaussian function:

$$G(\Delta i, \Delta j; \sigma) = \mathrm{e}^{-(\Delta i^2 + \Delta j^2)/2\sigma^2}, \tag{1}$$

where σ is the standard deviation. Then the central brightness which best fits the pixels around the point (i_0, j_0) in the sub-picture $H_{i_0 j_0}$ is given by:

$$D_{ij} \doteq H_{i_0 j_0} G(i - i_0, j - j_0; \sigma) + b \qquad (i, j) \text{ near } (i_0, j_0) \tag{2}$$

Where b is the estimated value of the background of the image which can be obtained from the images in several ways, DAOPHOT’s implementation uses a function *mode* to calculate this value. The symbol “$\doteq$” denotes a least-squares fit to the data for some set of pixels (i, j) in a defined region around and including (i_0, j_0). Then the numerical value of $H_{i_0 j_0}$ can be obtained by a simple linear least squares using the number of pixels, n, involved in the fit as:

$$H_{i_0 j_0} = \frac{\Sigma(GD) - (\Sigma G)(\Sigma D)/n}{\Sigma(G^2) - (\Sigma G)^2/n} \tag{3}$$

Equation 3 is the arithmetic equivalent of convolving the original image with the kernel function. Having calculated the H array of the same size of the original data D the routine runs through H looking for positions of local maxima, producing a list of positive brightness enhancements. There are two other criteria for detecting objects performed by the routine, the first one is a sharpness criterion that compares H_{i_0,j_0} to the height of a two-dimensional delta-function, d, defined by taking the observed intensity difference between the central pixel of the presumed star and the mean of the remaining pixels used in the fits of Eq. 3:

$$d_{i_0,j_0} \equiv D_{i_0,j_0}/\langle D_{i,j}\rangle, \quad (i, j) \neq (i_0, j_0), \quad sharp \equiv d_{i_0,j_0}/H_{i_0,j_0} \tag{4}$$

The second one is a roundness criterion; images of stars are strongly peaked functions of both x and y so that for round, well-guided images $h_x \approx h_y$. The roundness criterion readily distinguishes stars ($round \approx 0$) from bad rows and columns ($round \approx \pm 2$):

$$round \equiv 2\left(\frac{h_y - h_x}{h_y + h_x}\right) \tag{5}$$

3 Implementation

The FIND algorithm was implemented both in GPU and CPU in order to compare the response time between implementations and to identify reduction on time complexity. For the CPU implementation, the source code of DAOPHOT-FIND was translated from the original Fortran language to C language and compiled with GCC version 4.9.2 by adding the flag $-O3$ in order to turn on all optimizations provided by the compiler, and ran on a Intel Core i7 of 2.5 GHz. In the same way, GPU implementation was made by translating the original code to CUDA C language. The NVCC compiler provided with the NVIDIA Toolkit 7.0 was used including flags sm_50 and fast math library $-use\_fast\_math$. This new code was executed in a GeForce GTX 860M with 5 streaming multiprocessors and 640 CUDA Cores.

The difference between CPU and GPU code can be easily illustrated through the following snippet of code, representing the construction of the kernel for the profile fitting according to Eq. 1.

```
void constructKernel(
      float *kernel, int nbox, int nhalf, float sigsq) {

   int i, j, id;
   float rsq;

   for(i=0; i<nbox; i++) {
      for(j=0; j<nbox; j++) {
         id = i *  nbox + j;
         rsq = (j-nhalf)*(j-nhalf) + (i-nhalf)*(i-nhalf);
         kernel[id] = expf(-0.5*rsq/sigsq);
      }
   }
}
```

The GPU code looks almost the same, the main difference is the absence of loop instructions for; instead there is an index composed by the variables $blockIdx.x$ and $threadIdx.x$ provided by CUDA to identify each process as unique. By mapping this index to the position of the kernel, it is possible to calculate all their values in parallel. For this part of the FIND algorithm, the process for GPU was launched with as many blocks and threads as the dimensions of the kernel, `<<<nbox, nbox>>>`. The rest of the tasks, including convolution and criteria for roundness and sharpness were transformed in a similar way.

```
__global__ void constructKernel(
      float *kernel, int nhalf, float hsigsq) {

   int id = blockIdx.x * blockDim.x + threadIdx.x;
   float rsq;
```

```
        rsq = (blockIdx.x-nhalf)*(blockIdx.x-nhalf);
        rsq += (threadIdx.x-nhalf)*(threadIdx.x-nhalf);
        kernel[id] = __expf(rsq*hsigsq);
    }
```

There are two details that can improve the performance of calculation for the Gaussian function. First, the arguments of `__expf(rsq*hsigsq)` differ since the value of $hsigsq = -1/(2 * sigsq)$ can be calculated on the CPU before passing it as the argument. The second detail is the use of the function `__expf()`, which is optimized for GPU execution in combination with the flag $-use\_fast\_math$ mentioned before.

Another way to use indexes provided by CUDA, is by launching the GPU kernel with threads distributed in two dimensions as shown in the next snippet of code. In this case the kernel should be called with just one block and as many threads as pixels of the kernel for profile fitting `<<<1, dim3(nbox, nbox)>>>` or `<<<1, nbox*nbox>>>`. Since this part of the process is made only once, and the space of the kernel is really small (7×7 pixels used for the tests), no important difference was detected in the response time between the different forms for launching this GPU kernel.

```
    Launching <<<1, dim3(nbox, nbox)>>>
        id = threadIdx.x * nbox + threadIdx.y;
        rsq = (threadIdx.x-nhalf)*(threadIdx.x-nhalf);
        rsq += (threadIdx.y-nhalf)*(threadIdx.y-nhalf);
        kernel[id] = __expf(rsq*hsigsq);

    Launching <<<1, nbox*nbox>>>
        id = threadIdx.x;
        rsq = (id/nbox-nhalf)*(id/nbox-nhalf);
        rsq += (id%nbox-nhalf)*(id%nbox-nhalf);
        kernel[id] = __expf(rsq*hsigsq);
```

According to Eq. (3), the image convolution is the next step in the image analysis of FIND algorithm. In this process another important aspect of GPUs can be considered which is memory management. Since Shared memory resides on chip it's i/o operations are much faster than operations in Global memory. One of the implementations proposed basically splits the image in tiles, each of them launched as blocks of threads. With this approach, each thread is responsible for loading data from Global to Shared memory and perform mathematical operations for the convolution. One limitation of this approach, is the size of the tile, where the threads of one block are used for mapping the data needed for a single tile. A GPU such as GM107 having a limit of 1024 threads per block, could manage tiles of 32×32 pixels maximum (1024 elements).

The approach we use for this work is a little bit different. By analysing FIND algorithm in the part of convolution, there are many conditional statements in the convolution process to ensure the correct values for each pixel. A GPU executes threads in groups of 32 (*warp*), which are efficiently used if all 32 threads follow the same execution path, but conditional branches diverge the threads, since the warp serially executes each branch path leading to poor performance. The other important thing about the operations of FIND, is data access which is made just once for each pixel of the image and once for the values of the kernel. This means that improving performance by using shared memory to load data (as splitting the image in tiles) may not be a good idea, because FIND does not perform a simple convolution and the operations are made over variables, rather than data stored in Global memory.

For these reasons, the approach used for testing executes as many threads as pixels in the image, while fixes the number of threads per block, and changes the number of blocks according to the image size `<<<ImageSize/128, 128>>>`, and each thread performing all operations needs to convolve a single pixel. We found that in the testing process the use of 128 threads per block produces better results than other configurations. Within the code, the index of each thread is mapped to each pixel of the image as given by the following sequence of instructions

```
id = blockIdx.x * blockDim.x + threadIdx.x;
jx = id / ncol;
jy = id % ncol;
```

Furthermore, in order to improve performance based on the same concept of using different segments of GPU memory, a modification of this last approach was made. This new implementation takes advantage of Constant Memory by copying the values of the kernel for profile fitting, reducing operations on Global Memory. However, the implementation does not show much improvement, and in fact, we found that the complexity of FIND resides in the calculations and decision making rather than on the data access.

The testing procedure uses seven square images with different resolutions taken from the SkyView database [12] and centered on the galaxy Malin 1. Figure 1 shows an example of the image with a resolution of 512 pixels per side. We chose a rather uncrowded field since here we are more interested in the measurement of the performance than in the accuracy of the method. The range of the resolution of each image goes from 64 to 4096 pixels per side. Over this set of images, both CPU and GPU implementations of FIND use the same size of kernel in order to compare the execution time between them. The size of the kernel is calculated according to description of DAOPHOT [1] by applying the formula $2 * max(2.0, 0.637 * FWHM) + 1$, where the value of $FWHM$ is set to 5, resulting in a kernel size of 7 pixels. Table 1 shows the obtained results. In order to ensure that the results are not influenced by operating system processes or other operations of CPU or GPU, each test is executed five times and the averaged results are displayed.

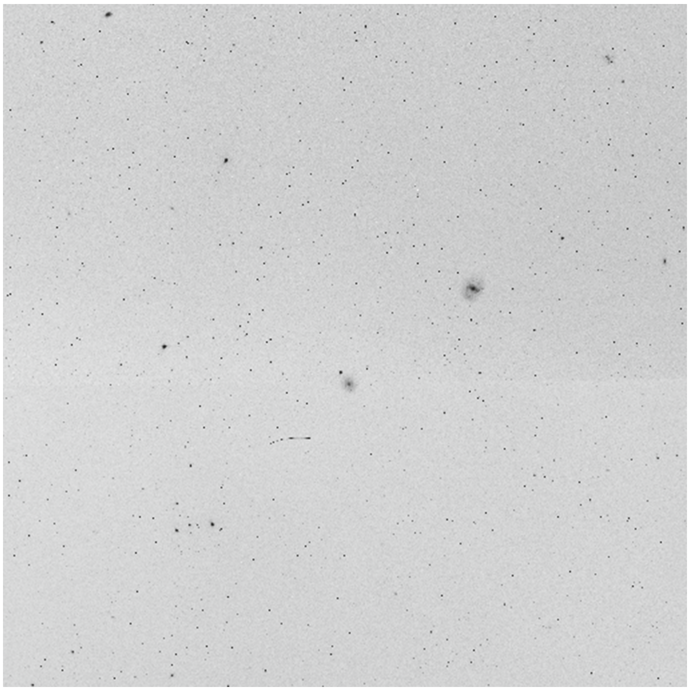

Fig. 1. DSS Blue image of sky $2 \times 2\,^{\circ}$ centred at Malin 1, a giant low surface brightness (LSB) spiral galaxy (Color figure online).

4 Results

Table 1 shows the difference between time consumed among the implementations in CPU, GPU without splitting the images in tiles and GPU by splitting the images in tiles. This difference is directly proportional to the size of the image. For large source images such as 2048 or 4096 pixels the CPU time starts to increase faster than GPU time. One would expect that by splitting the image in tiles and using the Shared and Constant Memory of a GPU, the performance should increase against GPU without splitting, but very surprisingly this is not the case. The analysis of the FIND algorithm shows that there are many more operations than in traditional convolution, including many branch statements and few memory access operations. In this case, the number of launched threads is the main cause for the low response time of the GPU that uses splitting of the image, when comparing with the GPU implementation without splitting.

Every time the image is processed using the splitting in tiles approach, blocks with a number of $(tileSize{+}kernelSize)^2$ threads each are launched to load data and perform operations, but actually only $tileSize^2$ threads perform operations, the rest of threads just load data. This means that for an image of 4096 pixels by side, using a kernel of 7 pixels by side and split in tiles of 16 pixels, there are $4096^2/23^2 = 65,536$ blocks each of them with 23^2 threads, but only 16^2 threads

Table 1. Results of the experiments over the 7 uncrowded images. The kernel size was fixed to require the same number of calculations in both implementations. Size of images and kernel are in pixels by side and the execution time is in seconds.

Input data		Average time/second		
Image Size	Kernel size	CPU	GPU	GPU (Tiles)
64	7	0.00038	0.00239	0.00048
128	7	0.00198	0.00413	0.00133
256	7	0.00562	0.00769	0.00502
512	7	0.02627	0.01524	0.01741
1024	7	0.09047	0.04563	0.06869
2048	7	0.33287	0.16260	0.25277
4096	7	1.32711	0.63014	1.01233

are really working, implying that less than 50 % of the threads of the block, perform the operations of FIND.

In a formal way, time complexity for the above implementations is analysed by fitting the following polynomial function with three parameters a, b and c

$$t = ax^b + c \tag{6}$$

In Eq. (6), t is the time consumption of the implementation as a function of the pixels per side of the image, x. Since the variable x represents the resolution per side of the image and the real input of the implementations is the complete image, then one should expect that a nonlinear regression will produce an approximated value of 2 and 0 for the parameters b and c, respectively. Table 2 shows the coefficients of the best fitted function for the obtained data through the conducted experiment.

Table 2. Results of the best fitted function in the experiment over the 7 images, the kernel size was fixed to have same number of calculations needed in all the implementations.

Implementation	a	b	c
CPU	8.58E-8	1.989	0.0021
GPU	4.63E-8	1.974	0.0040
GPU (Tiles)	6.22E-8	1.996	0.0010

The reduction in time is mainly obtained through the parameter a and Table 2 shows that the implementation on GPU consumes a smaller time, which is represented by the value of the parameter a. The parameter b, which is of major interest, was reduced too, but not in such a large amount. Figure 2 illustrates the behaviour of the equation using the parameters calculated in the CPU case and the two approaches in GPU.

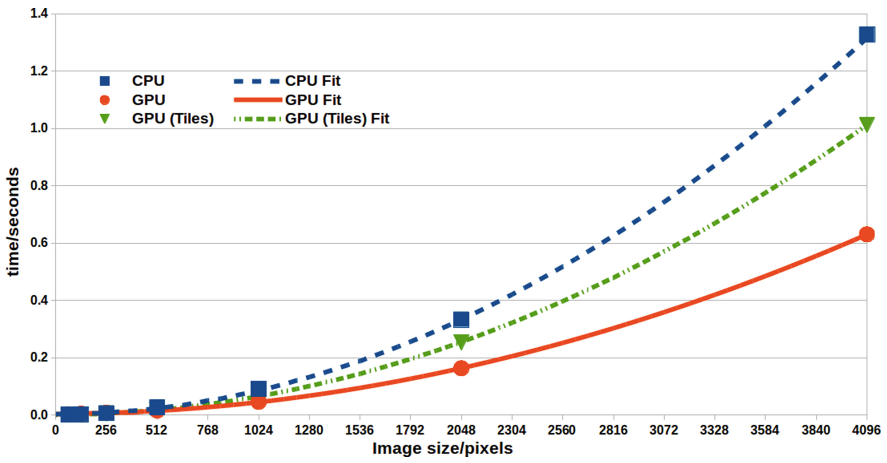

Fig. 2. Time consumption for CPU and GPU implementations of FIND task for different sizes of the images. The symbols represent the experimental data, while the lines correspond to the respective fitted functions.

5 Conclusions

This work presents advances in the implementation of DAOPHOT package on parallel GPU architecture using CUDA programming platform. As a first step we implemented the FIND algorithm and compared its performance against the sequential CPU version. The performed test shows that it is possible to reduce the time consumption of the FIND algorithm, through the implementation on a GPU architecture that is commonly found in modern computers. The time consumption reduction gives an advantage to perform the task of finding objects, specially when dealing with large image resolutions and image sequences. However, our results do not show a reduction in time complexity of GPU as large as expected. Our simple implementation of FIND is a demonstration of how easy is to exploit GPU architecture, but some additional programming changes are needed in order to get their full capabilities. It is possible to improve the performance of this parallel implementation by using nearly all the blocks and thread dimensions available to calculate all independent operations, and subsequently reducing the results [11]. On the other hand, the effort to do these improvements must analyse which of the operations can be calculated independently, and how they can be classified in order to perform the required reductions. This is a point that needs special attention because, as we have shown, reductions cannot be implemented as a trivial program on a GPU, and it is necessary to keep the way in which data are arranged along the whole task execution. Finally, it is necessary to consider that using the Shared Memory of a GPU, does not better performance in all cases. The identification of the kind of operations performed by the algorithm and the detection of memory interactions, rather than calculations with variables in a thread, are necessary conditions to further improve the performance of the algorithm.

References

1. Stetson, P.B.: DAOPHOT - a computer program for crowded-field stellar photometry. Publ. Astron. Soc. Pac. **99**, 191–222 (1987). ISSN 0004–6280
2. Becker, A.C., et al.: In pursuit of LSST science requirements: a comparison of photometry algorithms. Publ. Astron. Soc. Pac. **119**, 1462–1482 (2007)
3. http://www.lsst.org/
4. Davis, L.E.: A Reference Guide to the IRAF/DAOPHOT Package. NOAO, Tucson (1994)
5. Sterken, C., Manfroid, J.: A Guide. Astrophysics and Space Science Library, vol. 175. Springer, The Netherlands (1992)
6. Stoy R.H.: In: Proper Motions. International Astronomical Union colloquium, vol. 7, p. 48. University of Minnesota, Minneapolis (1970)
7. La Bonte A.E.: In: Proper Motions. International Astronomical Union colloquium, vol. 7, p. 26. University of Minnesota, Minneapolis (1970)
8. Tsutsui, S., Collet, P. (eds.): Massively Parallel Evolutionary Computation on GPGPUs. Natural Computing Series. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37959-8-1
9. Kyrkou, C.: Architectures for high performance computing. In: Survey on Stream Processor and Graphics Processing Units (2010). http://sokryk.tripod.com/Stream_Processors_and_GPUs_-_Architectures_for_High_Performance_Computing.pdf
10. Farber, R.: CUDA Application Desing and Development. Morgan Kaufmann, USA (2011)
11. Kirk, D.B., Hwu, W.W.: Programming Massively Parallel Processors A Hands-on Approach. Elsevier, USA (2010)
12. http://skyview.gsfc.nasa.gov/
13. http://starlink.eao.hawaii.edu/starlink