Improving Performance of Particle Tracking Velocimetry Analysis with Artificial Neural Networks and Graphics Processing Units

Rubén Hernández Pérez, Ruslan Gabbasov, and Joel Suárez Cansino

Universidad Autónoma del Estado de Hidalgo,
Centro de Investigación en Tecnologías de Investigación y Sistemas,
Cuerpo Académico de Computación Inteligente,
Mexico

rub3n.hernandez.perez@gmail.com

Abstract Flow analysis has a wide range of applications both in engineering and science, and many techniques have been developed over the years that allow the data extraction and measurement of the flow dynamics. In particular, Particle Tracking Velocimetry (PTV) techniques have shown good results when combined with Artificial Neural Networks (ANN) techniques, especially using Self-Organizing Maps (SOM). In order to improve the performance and reduce the time consumption of proposed SOMs for PTV analysis, the parallel nature of modern architectures such as Graphics Processing Units (GPU) can be used. In this paper we describe how the GPU architecture can be exploited for the implementation of the inherent parallelism of a SOM for PTV analysis, and measure the performance obtained with different optimizations techniques. We show that it is possible to gain a speedup of ~ 5 when running in parallel.

Keywords: Particle tracking velocimetry analysis, artificial neural networks, GPU

1 Introduction

Flow visualization techniques are used to reveal the fluid motions and allow to study the complex phenomena such as turbulence. Techniques such as Particle Tracking Velocimetry (PTV) give us quantitative two-dimensional information about the velocity field by extracting the positions of illuminated particles suspended in the flow and tracing their motion between two or more frames. This information can be used later to compute other flow quantities such as vorticity [9].

The applications of PTV cover a wide range of areas of engineering, science and industry [3]. Some examples are found in medical applications [1], [5], where the PTV analysis provides information for validation and testing of models and procedures that can be used later for surgical treatment of arteries and veins.

This kind of studies demand techniques and tools that enable data extraction from the images of PTV and more important, these tools must have response as fast as possible for PTV analysis, which can be critical, especially in cases when a large number of images is obtained in real time.

The PTV technique can be divided into two steps. First is the image capture and preprocessing where the positions of tracers are determined. The second one is the position match of the tracers between the images (pairing) and measurement of their displacement. Here we study only the second step.

In order to perform an efficient particle tracking when the number of tracers is very large, i.e., the field is crowded, Artificial Neural Networks (ANN) have been proposed as an efficient way to achieve a good level of accuracy [2] and, in particular, Self-Organizing Maps (SOM) have demonstrated to give excellent results [7] for such images. There have been some improvements to the Labonte's algorithm by adding a distance-dependent schema for updating the weights for the neurons [6] and by adding a Delta-Bar-Delta rule to the net to reduce the computation time [4].

Many ANNs, including SOMs, are already implemented in software and there are many available libraries that can be used [8]. The advantage of such implementations is that they can be directly invoked and used as a black-box, nevertheless, they are not necessarily optimized for custom problems and hardware. The problem of performance can be somehow mitigated by using modern microprocessor architectures that include set of instructions for specific tasks and the many-core architecture. Another way to improve the performance is by using special hardware implemented with digital or analogue circuits representing the neurons, but there are many issues yet to be solved like poor flexibility to change the net structure and the way to send and retrieve information from it.

In this work we focus on modern hardware that allows parallel execution, such as Graphics Processing Units or GPUs, the multi-core architecture that offers flexibility and is able to perform arithmetic operations, which represents a cheap alternative to the CPUs. The main characteristic of such architecture is the number of computing cores available which is much larger than any other commercial microprocessor currently available [8]. For programming these devices NVIDIA Corp. provides proprietary NVIDIA Toolkit, which is a set of tools, including it's C/C++ compiler and CUDA libraries. This tool is distributed for free and allows to exploit all supported GPU capabilities.

The multicore architecture allows to take advantage of the parallel nature of the SOM by implementing it on the GPU.

2 A SOM for PTV Analysis

The Labonte's original algorithm [7] starts with the coordinates vectors of particles in two consecutive frames denoted by x_i $(i = 1, ..., N)$ and y_j $(j = 1, ..., M)$, according to this vectors, two sub-nets of N and M neurons are created. Each neuron have two weight vectors denoted by v_i and w_j whose values at the beginning are assigned to x_i and y_j respectively. First the stimulus vector v_i

from the first layer is present to the second layer, then a winner neuron w_c is selected as the one closest to v_i. Having this, the neurons in the second layer are subjected to the next displacement.

$$\Delta W_j(c) = \alpha_j(v_i - w_c), \qquad j = 1, ..., M \tag{1}$$

Where $\alpha \in [0, 1]$ is an scalar value between 0 and 1 given by the condition $\alpha_j = \alpha$ if neuron $j \in S_c(r)$ and $\alpha_j = 0$ if not. In the case of the Ohmi's algorithm [6] the condition changes outside the radius r where the displacement is modified by the distance-dependent Gaussian function:

$$\alpha * \exp\{-(|w_c - w_j| - r)^2/(2r^2)\} \tag{2}$$

For both cases, $S_c(r)$ is the radius of the closed circle centred on the point y_c. Each time the first layer presents the weight vectors as stimuli for the second layer, the second layer is then updated according to the next operation.

$$w_j \leftarrow w_j + \sum_{i=1}^{N} \Delta w_j(c_i), \qquad j = 1, ..., M \tag{3}$$

And in the same way, in the next step, the second layer presents the weight vectors as stimuli for the first layer and update its values with the same formula just in the opposite direction.

$$v_i \leftarrow v_i + \sum_{j=1}^{M} \Delta v_i(c_j), \qquad i = 1, ..., N \tag{4}$$

At each step, the radius r of the circle, within which the neuron weights are changed, is decreased and the amplitude α of the weight translation is increased according to the following equations respectively:

$$r \leftarrow \beta r, \quad 0 < \beta < 1 \tag{5}$$

$$\alpha \leftarrow \alpha/\beta \tag{6}$$

These steps are iterated until r reaches a given value of r_f, which should be small enough to cover only the winner neuron. The value of β is an scalar between 0 and 1. Finally, the matching between particles in the frames is done by a last nearest-neighbour check with a small radius ϵ.

3 Implementation

The need for high performance computing through the use of GPU, is based on the complexity of the algorithm itself, since it involves two subnets interacting with each other every iteration, and each of them has as many neurons as particles in the corresponding frame. The operations performed by all neurons

of v_i against all neurons of w_j are executed in both direction each iteration, implying an exponential behaviour. Besides the number of iterations required by the net depends on the scalar value of the compression factor β and the boundaries delimited by r and r_f as will be shown in section 4.

In order to show the improvement that can be achieved by implementing the proposed SOM in a GPU, we have three approaches. The first one consists in use of the displacement of neurons in equation 3 and simply transforming the loop for processing all neurons (w_j in the first step and v_i in the second step) in parallel for each neuron. This means unrolling the nested loop shown below:

```
for (j=0; j<M; j++) {
   for (i=0; i<N; i++) {
      Calculate displacements...
   }
}
```

Then a parallel form in CUDA, where the code is executed once per neuron will be:

```
j = blockDim.x * blockIdx.x + threadIdx.x;
if (j<M) {
   for (i=0; i<N; i++) {
      Calculate displacementes...
   }
}
```

Here it is necessary to add a condition $i < N$ in order to guarantee that all neurons have one thread assigned.

This quick approach provides substantial speed enhancement in comparison with the traditional serial code and parallel implementations using OpenMP, but it doesn't exploit all the capabilities of the GPU. The architecture of a GPU is composed of blocks of threads that can be referenced by a 9-dimensional index and with these is possible to establish a mapping for all operations between the neurons on v_i and w_j. To illustrate how this can be done, we divide the algorithm in 5 basic steps as shown in Fig. 1.

There are two steps (Calculate Distances and Calculate Displacements) that involve calculation of values between each neuron of v_i and each neuron of w_j. In the first approach, these operations are done by executing in parallel a serial loop for each neuron of v_i. In the new approach the GPU can execute all operations as single threads and is possible for each thread to have two indexes to make reference both to v_i and to w_j and a third index to identify the calculation as unique.

```
idvi = threadIdx.x + blockIdx.x * blockDim.x;
idwj = threadIdx.y + blockIdx.y * blockDim.y;
idviwj = idwj + idvi * N;
```

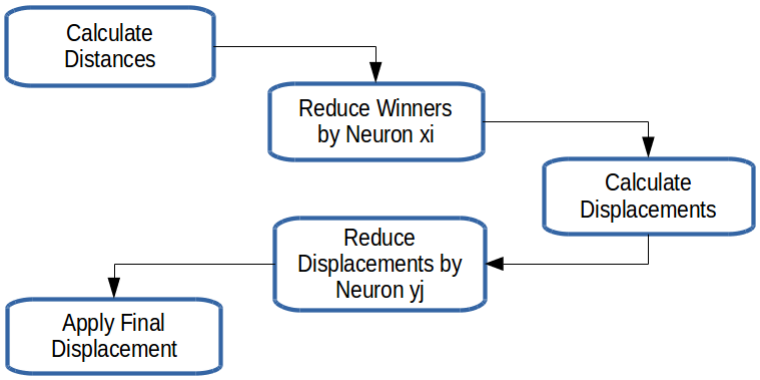

Fig. 1. Steps involved in the iterations over the net. Based on equations 1, 3 and 4, is is possible separate the calculations in well defined segments.

It is necessary to allocate a space of memory of size $N \times M$ to store all calculations between sub-nets and after it is necessary to perform two reduction operations, the first one searching for winners at each neuron v_i, and the second one calculating the cumulative weight displacement for each neuron in w_j. With this approach is possible to obtain a speedup of 3.2X just by calculating all distances and displacements in one step according to GPU capabilities. Note however, that the reduction operations were not parallelized, leaving room for a further improvement.

The parallel reduction is easy to implement but hard to get it right. As an example of optimization, we changed the the serial implementation by a parallel one as a binary three shown in Fig. 2.

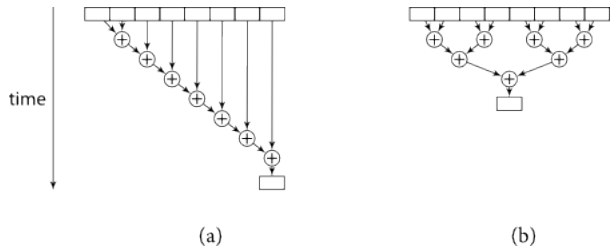

Fig. 2. Difference between simple serial reduction (a) and parallel reduction (b) implementations. It can be easily noted that the parallel reduction requires less execution time.

With this strategy we get a cumulative speedup of 4X with respect to the

original implementation, and as we can observed in Fig. 2, the binary three reduces the number of steps by executing parallel operations at deep nodes of the three. Here is a part of the code implemented in C code that performs this task:

```
idvi = threadIdx.x;
idwj = blockIdx.x;

originalId = idvi + idwj * N;

for(i=512; i>0; i>>=1) {
   if(idwj < i) {
      newId = idwj + (idvi + i) * N;
      Add operations to reduce elements...
   }
}
```

This code reduces blocks of 512 elements at once.

Finally, the last improvement has relation with the calculations of weight displacements. For the Labonte's algorithm it is a simple condition whether to apply or not a displacement based on r, while for Ohmi's algorithm this condition changes and depending on r value, the displacement is applied by the Gaussian function in equation 2. To improve performance of the net, the latter can be replaced by a function that describes the similar behaviour, but avoiding the *if* condition.

In this work we chose a sigmoid function as follows:

$$\alpha * (1 - 1/(1 + \exp\{-(|w_c - w_j| - r) * \lambda\})) \tag{7}$$

In this new equation, the r value is used to displace the function from the center and adds a new parameter λ that can be used to control the smoothness of the curve described by the function. This final approach by replacing conditions in CUDA code, improves performance up to 5.1X with respect to the original implementation.

4 Experiment

All implementations were tested using synthetic images of laminar flow containing 1024 particles in each frame. The performance was measured by changing the value of β, which due to the boundaries defined by r, r_f and equation 5 affects directly the number of performed iterations.

The total number of iterations performed by the net, can be considered as twice the operations performed (see Fig. 1). Each iteration updates weights in both sub-nets vi and wj, so if, for example, the net performed 135 iterations, it actually performed 270 updates over the weight vectors of the 1024 neurons.

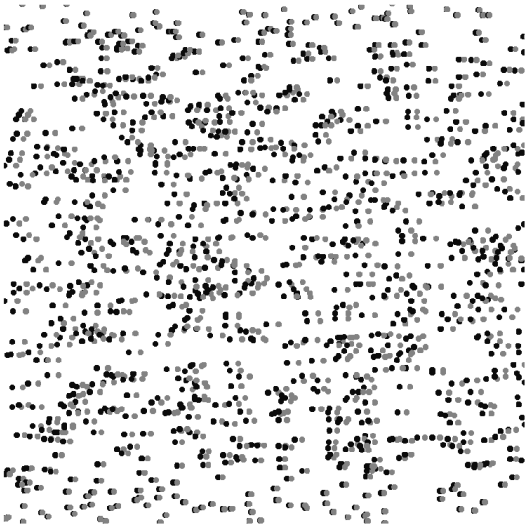

Fig. 3. Example of synthetic image of laminar flow with two frames overlapped. Particles in dark gray are the positions of the first time frame, and light gray dots are the positions of the second frame.

r	rf	β	iterations
100	0.1	0.70	20
100	0.1	0.75	25
100	0.1	0.80	31
100	0.1	0.85	43
100	0.1	0.90	66
100	0.1	0.95	135

Table 1. Values of parameters used to measure the performance of the net. The number of iterations depends on the initial radius r, the final radius r_f and the parameter β. The table shows only the 6 most significant values used for the test. The radius units are in pixels.

The execution time was measured using CUDA events instructions as shown below. It is possible to use CPU or operating system timers, but measurements can be biased by external processes and operating system thread scheduler. Using CUDA instructions eliminate such problems when measuring the GPU execution time. The CUDA event instructions are essentially a GPU time stamp that is recorded at a specified point in execution time:

```
cudaEvent_t begin, end;

cudaEventCreate(&begin);
cudaEventCreate(&end);

cudaEventRecord(begin, 0);

      Iterations over the net...

cudaEventRecord(end, 0);
cudaEventSynchronize(end);

cudaEventElapsedTime(&elapsedTime, begin, end);
```

Finally, the GPU time was measured for each implementation described above and for different β values as a total time used to computation and to copy and retrieve data to/from the memory since we found that for considered size of subnets (1024) the latter does not impact the final performance.

5 Results

The experiment with 4 implementations of the SOM algorithm, shows improvements of 3.2X, 4X and 5.1X in speedup respectively as compared to serial CPU version (see Fig. 4). The first approach shows that high speedup can be obtained by executing operations between frames at the same step, while the second was obtained by parallelizing the inner loop.

Finally, using a Sigmoid function in order to avoid taking decisions as implemented in original Labonte's and Ohmi's algorithms, allows to fully exploit SIMD architecture of the GPU hardware.

6 Conclusions

A GPU can be used to improve performance in a SOM for PTV analysis, however the speedup obtained heavily depends on parallelism and optimisation of the code. The easiest way is to take a serial code and transform it to a parallel one, but full capabilities of a GPU can be exploited by using optimizations of code, using indexes available for identifying threads and avoiding divergence statements. The algorithms from Labonte and Ohmi have proven to be efficient

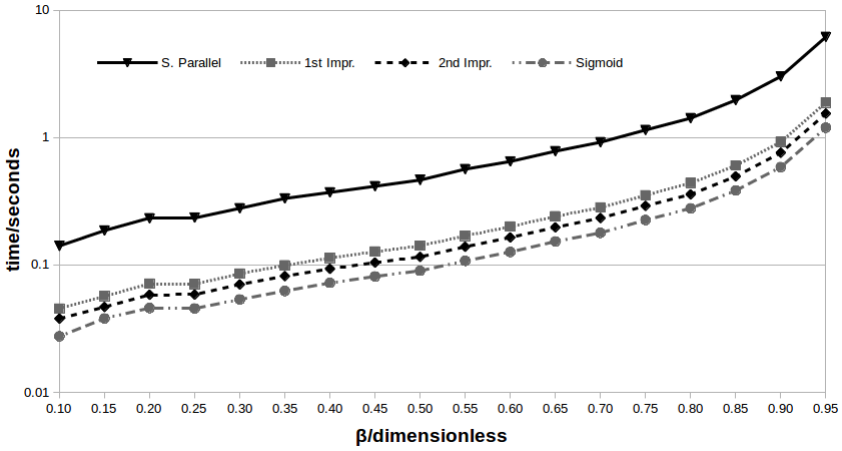

Fig. 4. This chart shows the execution time in seconds for each implementation and each value of β tested.

to deal with the problem of pairing particles, but the functions used to displace weights of neurons can be expressed in terms of a simpler function instead a condition, and it is possible to keep control of the radius and smoothness of the displacements in a more efficient way.

References

1. Grus, T., et al.: Particle image velocimetry measurement in the model of vascular anastomosis. Prague Medical Report 108, 75–86 (2007)
2. Ian Grant, X.P.: An investigation of the performance of multi layer, neural networks applied to the analsis of piv images. Experiments in Fluids 19, 159–166 (1995)
3. J. Hassan, H.Z.: Effects of vortex generator on junction flow. Applied Sciences and Technology (IBCAST) pp. 449–452 (2015)
4. Joshi, S.R.: Improvement of algorithm in the particles tracking velocimetry using self-organizing maps. Journal of the Institute of Engineering 7, 6–23 (2009)
5. Kabinejadian, F., et al.: Particle image velocimetry (piv) flow measurements of carotid artery bifurcation with application to a novel covered carotid stent design. IFMBE Proceedings 39, 1441–1444 (2012)
6. Kazuo Ohmi, A.S.: Cellular neural network based ptv. 13th Int Symp on Applications of Laser Techniques to Fluid Mechanics pp. 26–29 (2006)
7. Labonte, G.: A new neural network for particle-tracking velocimetry. Experiments in Fluids 26, 340–346 (1999)
8. Verber, D.: Implementation of Massive Artificial Neural Networks with CUDA. University of Maribor (2012)
9. Westerweel, J.: Digtal Particle Velocimetry, Theory and Application. Delft University Press (1993)