PARALLEL PARTICLE SWARM OPTIMIZATION FOR IMAGE SEGMENTATION

Agustinus Kristiadi, Pranowo, Paulus Mudjihartono

Atma Jaya Yogyakarta University – Yogyakarta - Indonesia
Jl Babarsari 43 Yogyakarta 55281 Indonesia

090705773@students.uajy.ac.id, pranowo@staff.uajy.ac.id, paul235@staff.uajy.ac.id

ABSTRACT

One of the problems faced with Particle Swarm Optimization (PSO) is that this method is simply time consuming. It is so, especially when it deals with a problem that needs a lot of particles to represent. This paper tries to compare the speed of PSO run at parallel mode to ordinary one. The testing applies an example of an image segmentation to demonstrate the PSO method to find best clusters of image segmentation. Best clustering is determined by viewing it as it is an optimization problem in finding the minimum error of the clustering. The PSO process, especially the iteration; the one that is the most time consuming; can be fastened by the usage of the parallel property of the PSO. We use NVIDIA CUDA for parallelizing the computation occurred in each particle. The results show that PSO run 170% faster when it used Graphic Processing Unit (GPU) in parallel mode other than that used CPU alone, for number of particle=100. This speed is growing as the number of particle gets higher.

Keywords: PSO, Parallel, Image Segmentation, Clustering, CUDA.

1. INTRODUCTION

Particle Swarm Optimization (PSO) is a metaheuristic method using swarm intelligence [1]. The idea behind PSO originated from behaviour of swarm of animals, for example flock of birds or school of fish, to search foods. In PSO, each particle move based of best known position of optima, so that each particle tends to move to that best known position with a hope that it finds global best position in the search space.

Since PSO is a metaheuristic method, it offers many solutions in many ways to one specific problem. Some solutions are satisfied and some others are not. However it is practically accepted.

PSO can be used to solve many optimization problems, for example scheduling [2], neural network weighting, and data mining [3]. Another example is that PSO is used in generating 'university timetable', a kind of lecturing schedule in university, in a manner of its very origin [4]. Moreover, PSO can also be used to solve clustering problems and can be applied to the image segmentations.

As it is metaheuristic methods, PSO implements the stochastic optimization which includes random and trial-error behaviours which is relatively slow [5]. From this point of view, we need to do research for finding the methods that can be accelerated the processing time of PSO. One of these methods is parallel computing. Fortunately, parallel computing library is available. It is NVIDIA CUDA. CUDA is a technology that does calculation concurrently in graphic processing unit (GPU) [6].

To analyze the good of PSO using CUDA, we compare the speed and the quality of an image segmentation using PSO clustering method that run in both CPU and GPU.

2. PARTICLE SWARM OPTIMIZATION

Particle swarm optimization (PSO) developed by Kennedy and Eberhart as a stochastic optimization method based on swarm intelligence. The idea behind PSO is to mimic behaviours of school of fish and flock of birds whenever they looking for food [7]. PSO is modeling population of swarm as search agents that move looking for an optimal solution in search space. Each individual (particle) in the swarm remembers its best known position.

Each particle's movement is affected by its current velocity, personal best known position, and best known position in the entire swarm. Hence, for particle i, in dimension d, its movement is calculated as follows:

$$v_{id} = w * v_{id} + c_1 * r_1 * (p_{id} – x_{id}) + c_2 * r_2 * (p_{gd} – x_{id}) \quad (1)$$

where w is the inertia, c_1 and c_2 are nonnegative constant, r_1 and r_2 are random number between 0 and 1. p_{id} is personal best known position, p_{gd} is best known position in the entire swarm, and x_{id} is current position of particle.

Velocity from equation (1) is used to determine new position of particle. New particle's position can be calculated as follows:

$$x_{id} = x_{id} + v_{id} \tag{2}$$

3. PSO IMAGE SEGMENTATION

One of the methods to perform image segmentation is clustering. The idea is that the pixels in an image are grouped into several smaller groups called clusters. The grouping is done in pixel color basis. Pixels with similar color or relatively similar are grouped into one certain cluster. PSO can be used to solve clustering problem, which in this case, PSO is used to perform image segmentation.

In the clustering context, each particle represents set of cluster centroids as many as N_c [8]. Each particle is represented as follows:

$$x_i=(m_{i1},\ldots m_{ij},\ldots m_{iNc}) \tag{3}$$

where m_{ij} is j-th centroid in i-th particle. Swarm of particles represents several candidate solutions for clustering problems. Furthermore, in each of iterations PSO tries to improve the candidate solution.

The quality of clustering can be measured using quantization error:

$$f = \frac{\sum_{j=1}^{N_c} \frac{\sum_{\forall z_p \in Z} d(z_p, m_j)}{|Z|}}{N_c} \tag{4}$$

where Z is data vector to be clustered, and d function is the Euclidean distance:

$$d(z_p, m_j) = \sqrt{\sum_{i=1}^{d_m} (z_{pi} - m_{ji})^2} \tag{5}$$

where d_m is dimension of data to be clustered.

4. NVidia CUDA

CUDA (Compute Unified Device Architecture) is a GPU architecture from NVidia that enables program run at GPU [9]. This refutes that GPU can merely do graphics computing; on the contrary it can also do general purpose computing such as those on CPU. CUDA architecture was first introduced in 2007; which is GPU NVidia series G80.

The model of CUDA programming divides work into many smaller works that will be handled by smallest processor unit, called **thread**. Each thread has its own memory and process to perform the works altogether with all other threads. The group of all threads is divided into several groups of threads called **blocks**. Each block shares a memory for communicating among the threads in that block. All blocks are eventually grouped to become a **grid**; which is used to perform a computation. See Figure 1, which is taken from [10].

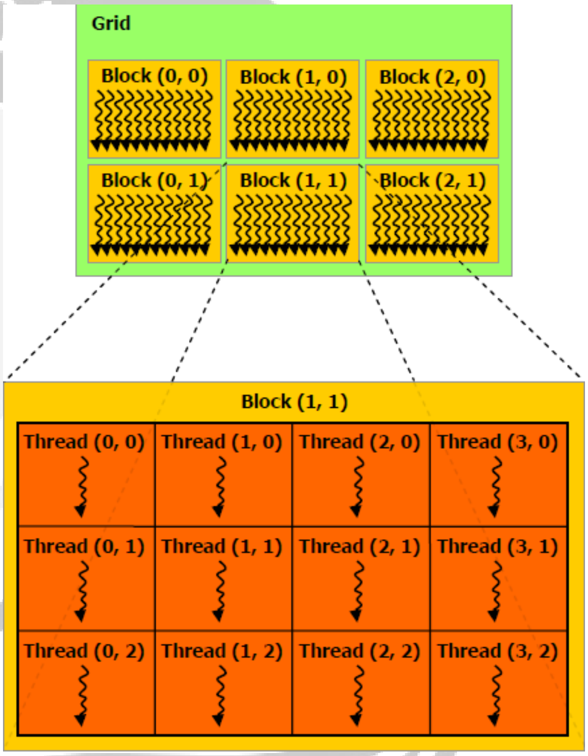

Figure 1. CUDA Processor Structure (source [10])

In CUDA, CPU is called the **host**, and GPU is called the **device**. Furthermore CUDA program has two parts; which are function that runs at CPU and function that runs at GPU. Function on CUDA program that runs at GPU is called the **kernel**. During the execution of the function, programmer should provide GPU a piece of information of how many blocks are needed and how many threads in each block should be.

Programming Language that used by CUDA is C/C++ with the specific extension of functions and syntax particularly set to CUDA. CUDA compiler, with its driver nvcc, will compile the code running at the device while the code which runs at the host will be compiled by the compiler resides at the host, such as GCC/G++ in Linux or Visual C++ in Windows.

5. GPU PSO IMAGE SEGMENTATION

The ordinary method of PSO remains the same with the original PSO itself. In the case of image segmentation, one particle is represented by the sequence of cluster centroids, while the position of particle is represented by the RGB values of those centroids. Suppose we intend to cluster the image data into two colors then there are two RGBs stand for the position of the particles; which is six dimensions. This algorithm can be written as follows (two clusters division):

```
1. Initialize N particles taken from N/2 couple
   pixels of the sample image and set their
   position accordingly. Set pBest (personal best
   position of particle p), and gBest (global best
   position of particle p) with the random two
   RGB values of the sample image data and
   also set the speed of each particle with zero.
2. For iteration = 1 to max_iteration do:
   a. For each particle p do:
         i. For each cluster c do:
              Update speed of p of dimension d
              on cluster c
              Update position p of dimension d
              on cluster c
        ii. p.posisi ← position of p.
   b. For each particle p do:
         i. If (fitness(p.posisi) <
            fitness(p.pBest)):
              p.pBest ← p.posisi
        ii. If (fitness(p.pBest) < fitness(gBest)):
              gBest ← p.pBest
```

After some iteration, gBest will get its optimal solution of clustering. The optimal solution is the solution that satisfies the objective function, in this case, is to minimize the quantization error of the cluster. The solution is a couple of pixel (the case of 2 clusters) that compels the segmentation falling into two colors.

On the other hand, the parallel PSO algorithm uses the parallel property to generate the solutions. PSO naturally is an algorithm that inherently parallel not serial. This means that the process of one certain particle does not depend on other particle processes. One particle should not wait for other particles to finish their processes to finish its own.

There are three kinds of parallel PSO implementation; they are naïve, asynchronous and full device implementation. Firstly, in naïve implementation pBest computation (concurrently among particles) is done in device but the gBest computation is in the host. It needs frequent copies of pBest from the device to the host so that the computation is not optimized. Secondly, asynchronous implementation is just like the naïve one unless it performs the kernel concurrently with the copying computation. Lastly, full device implementation that we did in our research needs all PSO functions are concurrently accomplished on the device. The algorithm of this **full device** implementation is as follows:

```
1. Initialize some particles; pBest; gBest and the
   speed of each particle as in ordinary PSO
   mode.
2. For iteration = 1 to maximum iteration do:
   a. Update the speed and position of the
      particles concurrently on the device.
   b. Update pBest of all particles concurrently
      on the device.
   c. Set Idx with the index of particle which its
      gBest is minimum.
   d. Update gBest with pBest of particle with
      the index Idx.
```

(Each particle carries out its computations independently in a single thread. This has removed the loop 'for each particle'.)

6. EXPERIMENTAL RESULTS

Experiment is done using a PC with Intel Core 2 Duo T6600 2.2 GHz as CPU, and NVIDIA GeForce 110M with 256MB of memory as GPU.

In this experiment, we study the time processing of PSO when it runs in GPU compared to CPU, relative to number of particle. The experiment needs the sample image to be segmented. It is a color image with 160 x 120 pixels of dimension. Other parameters that are also needed will be the number of cluster and the maximum iteration, where we pick 2 and 60 respectively.

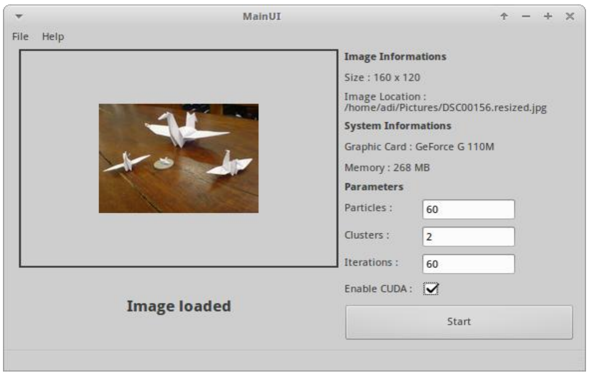

Image Informations

Size : 160 x 120

Image Location : /home/adi/Pictures/DSC00156.resized.jpg

System Informations

Graphic Card : GeForce G 110M

Memory : 268 MB

Parameters

Particles : 60

Clusters : 2

Iterations : 60

Enable CUDA : ☑

Image loaded

Figure 2. The Interface of the Tester Prototype Displaying Sample Image

Figure 2 shows the sample image to be clustered. In this case, image is clustered into two clusters by doing PSO with 60 particles and 60 iterations. Similarly, the testing also applied for 20 and 100 particles with the same number of cluster and iteration.

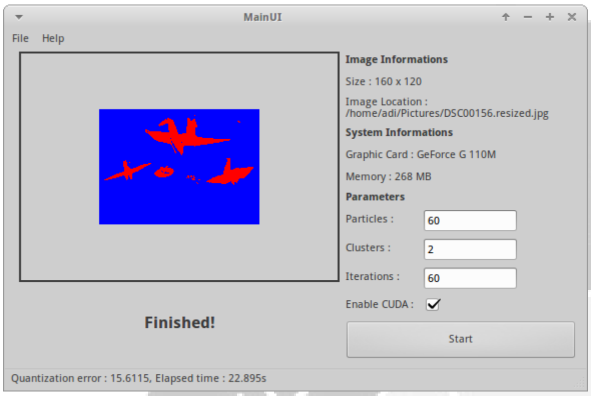

Image Informations

Size : 160 x 120

Image Location : /home/adi/Pictures/DSC00156.resized.jpg

System Informations

Graphic Card : GeForce G 110M

Memory : 268 MB

Parameters

Particles : 60

Clusters : 2

Iterations : 60

Enable CUDA : ✓

Finished!

Quantization error : 15.6115, Elapsed time : 22.895s

Figure 3. The Interface of the Tester Prototype Displaying the Image Segmentation Result

Next, figure 3 presents the cluster results. There are two segments in the image that fall in their two color clusters, red and blue respectively.
In Addition, table 1, 2 and 3 display the testing results of both methods.

Table 1. Test Result for Cluster=2; Epoch=60; p=20

Method	Fitness	Time
CPU	15.729	9.805
Full Device GPU	15.642	18.982

Table 2. Test Result for Cluster=2; Epoch=60; p=60

Method	Fitness	Time
CPU	15.616	27.8478
Full Device GPU	15.613	23.6246

Table 3. Test Result for Cluster=2; Epoch=60; p=100

Method	Fitness	Time
CPU	15,611	46.346
Full Device GPU	15,614	27.363

In the same notion of presenting results, the results can be viewed through graphics visualization as in figure 4 and 5.

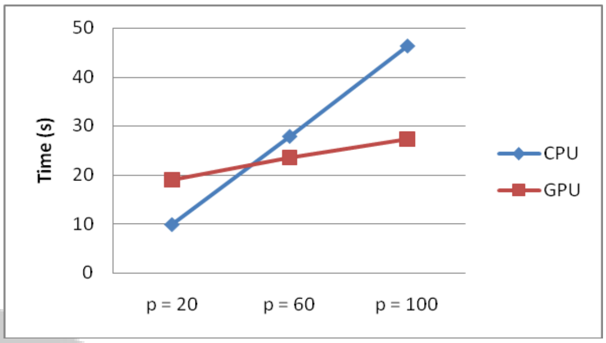

Figure 4. Time Consumption Relative to the Number of Particle with Cluster=2, Epoch=60.

Figure 4 shows the processing time used by PSO for image segmentation in GPU and CPU, relative to the number of the particles p. PSO for image segmentation that run in GPU is generally faster than that run sequentially in CPU. It is about 1.7 times as fast as that in CPU for a big enough number of the particles (p=100). However, in the case of a small number of the particles, PSO for image segmentation in GPU run slower than that in CPU due to the overhead of the parallel processing such as memory copies from CPU to GPU. This overhead time becomes the major portion of the whole processing time in small p while in CPU mode this overhead time does not exist.
In the case of big p, CPU mode is getting worse. The sequential process takes place and fills up the most time of computation. On the other hand, time process in GPU mode relatively remains constants at its portion of the overhead time. It implies that the overhead time is likely not affected by the number of the particles in parallel mode. It shows that the bigger the p number, the clearer that the parallel mode (in GPU) win the competition over the sequential one (in CPU only).
.

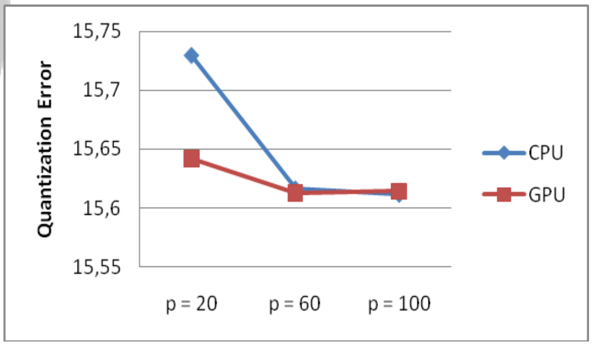

Figure 5. Fitness Value Relative to the Number of Particle with Cluster=2, Epoch=60.

Lastly, figure 5 shows the quality of the clustering of PSO for image segmentation that runs in CPU and GPU, relative to number of particle. In general, the quality of the two only differs a little. In p=20, the difference does not exceed 0.1 while in p=60, and p=100 the difference is relatively none. This means the objective function is as well achieved in parallel mode as in ordinary one.

7. CODE IMPLEMENTATION

This section will only list code from basic data structure, digital image capturing, and PSO GPU-related codes. A data structure is needed to represent data digital, particle, and GBest. This data structure resides on host only. First, we need to represent digital image data; which is a pixel. The data structure of image data is simply an array of three integer since pixel is composed from three colors; Red, Green and Blue. Moreover, we also need to represent Particle and GBest. Particle contains as many positions, pBests and velocities as clusters whereas GBest as another important structure consisting centroids and quantization error values. Here are the data structures:

```
struct Data
{
    int info[DATA_DIM];
};
struct Particle
{
    Data *position;
    Data *pBest;
    Data *velocity;
};
struct GBest
{
    short *gBestAssign;
    Data *centroids;
    int *arrCentroids;
    float quantError;
};
```

The Digital image data should be saved in the host by an assignment. Since we have only `Data` which represents pixel then we need to create array of `Data` to accommodate all pixels of the image. The image data capturing code is as follows:

```
IplImage* input = NULL;
input =
cvLoadImage(file_name.toStdString().c_st
r(), -1);
arr_image_ = new char[w * h * c];
flat_datas_ = new int[w * h * c];
datas_ = new Data[w * h];

for (int i = 0; i < w * h; i++)
{
    Data d;

    for(int j = 0; j < c; j++)
    {
      arr_image_[i*c+j] = (unsigned
      char)input->imageData[i*c+j];
      flat_datas_[i*c+j] = (unsigned
      char)input->imageData[i*c+j];
      d.info[j] = (unsigned char)input-
      >imageData[i*c+j];
    }
   datas_[i] = d;
}
```

Clustering functions are needed by both methods; PSO CPU or PSO GPU. The following text is the clustering function header in host and device respectively:

```
GBest hostPsoClustering(Data *datas, int
data_size, int channel,int
particle_size, int cluster_size, int
max_iter);

extern "C" GBest devicePsoClustering
(Data *datas, int *flatDatas, int
data_size,int channel, int
particle_size,int cluster_size, int
max_iter);
```

There one extra parameter in the device side function header that is `flatDatas`. This parameter is needed since the PSO GPU functions needs the array 1-dimension data variabel.

Next, the PSO GPU-related functions are listed in the next code.

```
cudaMalloc((void**)&devPositions, size);
cudaMalloc((void**)&devVelocities,
size);
cudaMalloc((void**)&devPBests, size);
cudaMalloc((void**)&devGBest,
sizeof(int) * cluster_size * DATA_DIM);
cudaMalloc((void**)&devPosAssign,
assign_size);
cudaMalloc((void**)&devPBestAssign,
assign_size);
cudaMalloc((void**)&devDatas,
sizeof(int) * data_size * DATA_DIM);
```

Like the implementation of PSO CPU, PSO GPU also allocates memory for the operating vaiables by using `cudaMalloc` function. The difference of the memory allocation between both methods is that in PSO GPU, all variables changed to be an array of 1-dimension types. This fits to the so called ‘coalesced memory access’ in CUDA where accessing consecutive memory is faster than not.

```
cudaMemcpy(devPositions, positions,
size, cudaMemcpyHostToDevice);
cudaMemcpy(devVelocities, velocities,
size, cudaMemcpyHostToDevice);
cudaMemcpy(devPBests, pBests, size,
cudaMemcpyHostToDevice);
cudaMemcpy(devGBest, gBest, sizeof(int)
* cluster_size * DATA_DIM,
cudaMemcpyHostToDevice);
cudaMemcpy(devPosAssign, posAssign,
assign_size, cudaMemcpyHostToDevice);
cudaMemcpy(devPBestAssign, pBestAssign,
assign_size,  cudaMemcpyHostToDevice);
cudaMemcpy(devDatas, flatDatas,
sizeof(int) * data_size * DATA_DIM,
cudaMemcpyHostToDevice);
```

After allocating memory on the device side, the memory needs to be initialized with the data comes from the host. The initialization process is done by using function `cudaMemcpy` that resembles `memcpy` in C. The parameters of `cudaMemcpy` encompass the pointer of the destination memory, the pointer of the source memory, the amount (bytes) of the memory needs to be copied and flag that determines the direction of copying process whether from the device to the host or vice versa.

```
__global__ void kernelUpdateGBest  (int
*gBest, int *pBests, int offset, int
cluster_size)
{
    int i = blockIdx.x * blockDim.x
    + threadIdx.x;
    if(i >= cluster_size * DATA_DIM)
        return;
    gBest[i] = pBests[offset + i];
}

__global__ void
kernelUpdateGBestAssign(short
*gBestAssign, short *pBestAssign, int
offset, int data_size)
{
    int i = blockIdx.x * blockDim.x
       + threadIdx.x;
    if(i >= data_size)
        return;
    gBestAssign[i] =
       pBestAssign[offset + i];
}
```

The code listed above displays that every single thread is used to update one single index of gBest. This means that we need as many thread as gBest index in the array. The final gBest that found on the last iteration will eventually become the solution itself. Lastly, the value of final gBest will be copied from the device back onto the host for presentation purpose.

8. CONCLUSION

From the experimental results, we can say that parallel PSO for image segmentation is more efficient than that run in CPU. In the case of the big number of particle (p=100), the speed in parallel mode is up to around 1.7 times as fast as that in CPU. This number is growing as the number of particle p gets higher. While in the case of the small number of particle, the application of the parallel PSO does not make any good. In addition, the quality between the two has no significant difference.

For future research, the speed of PSO for image segmentation can be further improved by using faster and newer model of GPU. Moreover, PSO for image segmentation can also be developed in OpenCL.

9. REFERENCES

1. Poli, R., Kennedy J., Blackwell T.: Particle Swarm Optimization: An Overview, Springer Science (2007).
2. Tasgetiren F., Sevkli M., Liang Y.C., Geneyilmaz G.: Particle Swarm Optimization Algorithm for Single Machine Total Weighted Tardiness Problem, 2004 IEEE Congress on Evolutionary Computation, Volume 2 (2004).
3. Weiss, R.M.: GPU-Accelerated Data Mining with Swarm Intelligence, Department of Computer Science Macalaster College, PhD Thesis (2010).
4. Mudjihartono P., Gunawan W.T., Ai T.J.: University Timetabling Problems With Customizable Constraints Using Particle Swarm Optimization Method, International Conference on Soft Computing, Intelligent System and Information Technology (ICSIIT), Petra University, Indonesia. ISBN: 978-602-97124-0-7 (2010)
5. Chen, X., Li Y.: Neural Network Training Using Stochastic PSO, ICONIP'06 Proceedings of the 13th International Conference on Neural Information Processing- Volume Part II (2006).

6. Zhou, Y., Tan Y.: GPU-based Parallel Particle Swarm Optimization, 2009 IEEE Congress on Evolutionary Computation, Volume 2 (2009).
7. Kennedy, J., Eberhart, R.C.: Swarm Intelligence, Morgan Kauffman (2001).
8. Merwe V. D., Engelbrecht A.P.: Data Clustering using Particle Swarm Optimization, Proceedings of IEEE Congress on Evolutionary Computation (2003).
9. Kirk, D.B., Hwu W.W.: Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kauffman (2010).
10. NVidia CUDA C Programming Guide Version 3.2, NVidia, pp. 9 (2010).