Parallelized Particle Swarm Optimization on Inverse Kinetics

林宗佑
CSIE
NCYU
Taiwan
jacklin315@gmaill.com

劉禮榮
CSIE
NCYU
Taiwan
louis75g@gmail.com

ABSTRACT

Particle swarm optimization algorithm(PSO) is a gradient-free global optimization algorithm, which can solve multivariable engineering optimization problems in fields like biomechanics, civil engineering and robotics to name just a few.

PSO was first proposed in 1995. Since then many variants were proposed in order either to tailor to their problem or improve the performance specific to their system. What we present here is Parallelized Synchronous Vanilla PSO to solve the inverse kinetics problem. PSO is population-based. It is inherently suitable for parallelization. In addition to OpenMP, we also used CUDA to parallelize it.

1 Introduction

1.1 Introduction to PSO

The pseudocode for serial PSO is shown as the figure below.

1. Initialization
   1.1. For each particle i in a swarm population size P:
      1.1.1. Initialize X_i randomly
      1.1.2. Initialize V_i randomly
      1.1.3. Evaluete the fitness $f(X_i)$
      1.1.4. Initialize $\boldsymbol{pbest}_i$ with a copy of X_i
   1.2. Initialize $\boldsymbol{gbest}$ with a copy of $X_i$with the best fitness
2. Repeat until a stopping criterion is satisfied:
   2.1. For each particle i:
      2.1.1. Update V_i^t and X_i^t according to Eqs. (1) and (2)
      2.1.2. Evaluete the fitness $f(X_i^t)$
      2.1.3. $\boldsymbol{pbest}_i \longleftarrow X_i^t$ if $f(\boldsymbol{pbest}_i) < f(X_i^t)$
      2.1.4. $\boldsymbol{gbest} \longleftarrow X_i^t$ if $f(\boldsymbol{gbest}) < f(X_i^t)$

Figure 1: Serial PSO pseudocode

X_{ij}^t is the j-th element of i-th particle's position vector at t-th iteration. V_{ij}^t is the j-th element of the i-th particle's velocity vector at t-th iteration. X_{ij}^{t+1} is updated by eq1 at (t+1)-th iteration. V_{ij}^{t+1} is updated by eq(2) at (t+1)-th iteration. ω is the weighting for inertia. $c1$, $c2$ are coefficients for random numbers $r1$, $r2$ which are ranged from 0 to 1. $pbest_i$ is the position with best fitness the i-th particle has ever met. $gbest_i$ is the position with best fitness the whole swarm has ever seen.

$$X_{ij}^{t+1} = X_{ij}^t + V_{ij}^{t+1} \tag{1}$$

$$V_{ij}^{t+1} = wV_{ij}^t + c_1 r_1^t \left(pbest_{ij} - X_{ij}^t\right) + c_2 r_2^t \left(gbest_j - X_{ij}^t\right) \tag{2}$$

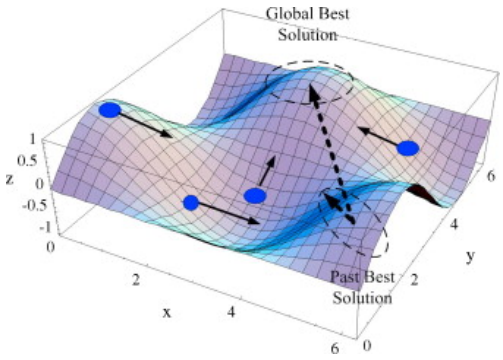

Figure 2: PSO algorithm visualization

1.2 Introduction to Inverse Kinetics

The inverse kinetics is aimed to find one or many sets of parameters to enable a robot arm to reach a prescribed point. We'll use PSO to search for *θ1, θ2, θ3* and *d* so that the end tip can reach the desired *(x, y)* as shown in the figure below.

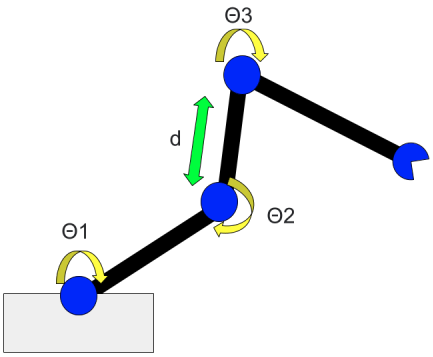

Figure3 :Robot Arms

1.3 PSO applied on Inverse Kinetics

Each particle of PSO is at a four dimensional space-($\theta1$,$\theta2$,$\theta3$,d), We constrain our $\theta1$ in the range [-π,π], $\theta1$ and $\theta2$ in range [-π/2,π/2] and d in range [1,2]. As stated in Serial PSO pseudocode 1.1.1 and 1.1.2, we randomly initialize the position and velocity throughout the four dimensional space within the specified boundary. After evaluating all particles' fitness according to the distance between the end tip and desired (*x,y*), we can assign the $pbest_i$ to each particle, and $gbest_i$ to the whole swarm as stated in pseudocode 1.1.4 and 1.2. Note that the fitness is the lower the better. Then all particle begin searching and hopefully the $gbest_i$ will converge to a point, at which the end tip can reach the desired (*x,y*).

Now we want to parallelize the algorithm with two different methods.

5 Proposed Solution

We'll parallelize it with OpenMP and CUDA.

5.1 Parallelization by OpenMP

Proposed solution for the OpenMP version is listed below. Our program set the desired number of threads by:

```
opm_set_num_thread(NUM_THREAD)//NUM_THREAD = 1..10;
```

The original serial PSO is doing iteration in pseudocode section 2. Since the algorithm updates $gbest_i$ synchronously, we added the OpenMP work-sharing directive to pseudocode section 2.1 without needing to worry about data race.

```
/*2 REPEAT UNTIL A STOPPING CRITERION IS SATISFIED*/
#pragma omp parallel for
/*2.1 FOR EACH PARTICLE i*/
```

It turns out the efficiency was quite good. But we wanted to improve it even further. As a result of the directive being nested in loop 2, the threads were created and destroyed again and again. So we refactor the code so that we can put the directive outside the loop 2 and the threads only need to be created once. That is:

```
#pragma omp parallel for
/*2 REPEAT UNTIL A STOPPING CRITERION IS SATISFIED*/
/*2.1 FOR EACH PARTICLE i*/
/*BARRIER HERE TO SYNCHRONOUS*/
```

Note that in order to achieve this, extra barrier code should be added at the end. Unfortunately, the refactored code doesn't do better. So we came to a conclusion that either in our case creating/destroying threads cost not much or the barrier consumed too much time.

5.2 Parallelization by CUDA

Recently, GPU parallelized technique has been applied to a large variety of computing problems. Due to different architecture from CPU , GPU has surpassed CPU in computing large amounts of data simultaneously. CUDA is a GPU computing platform introduced by NVIDIA which makes it much easier to construct parallelized computing applications with GPU supported by NVIDIA. PSO is an algorithm which concludes a large amount of computing including node position computing, particle best position computing, global best position computing and also pseudo-random number computing. Due to the advantages above, we design a model to parallelize PSO with CUDA.
The following diagram shows how we implement parallelized PSO with CUDA. The parallelized algorithm can be divided into four parts including random factor calculating, speed and position calculating, out of bound checking and tackling, fitness calculating and reducing.

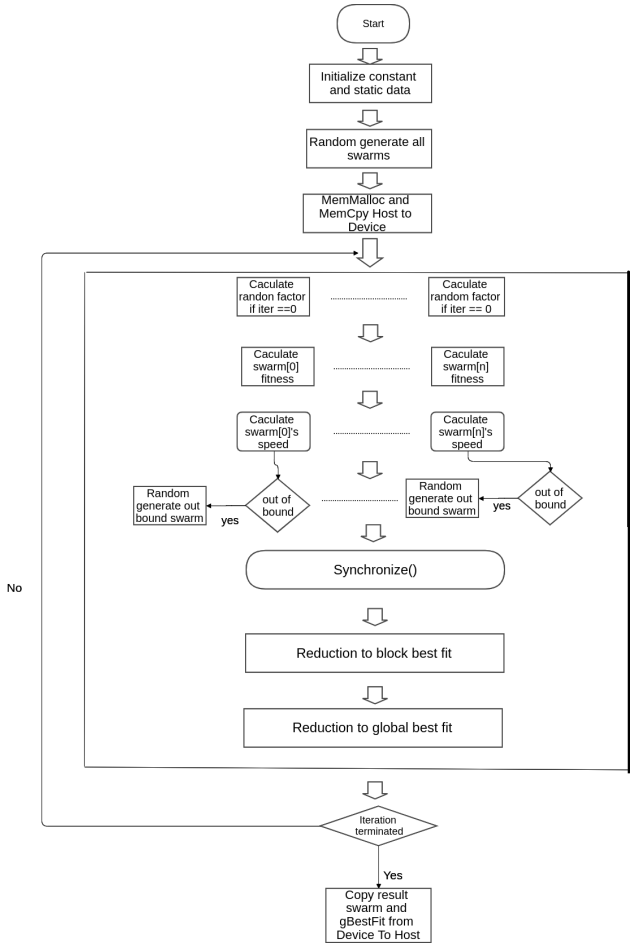

Figure 4: Flowchart for CUDA parallelized PSO

First, we initialize all constant and static data including initial point, target point, bounds etc. Then we generate the swarm randomly using CPU. Due to the high computation effort to calculate fitness, find particle best point, global best point, the task is postponed to the part which GPU is responsible for. After finishing initializing all swarms, we copy the properties of the swarms and all constants needed to the GPU memory.

1. Calculate the particle fitness if the iteration equals zero.
2. Calculate the speeds of all swarms and also its next point.
3. Check if the new point is out of bound and regenerate those by random number generator.
4. Synchronization
5. Reduce particle fitness to block best fitness
6. Reduce block best fitness to global best fitness

.

As the Flowchart shows, the random number generator plays an important row in the algorithm. We have tested that calculating the same big amount of random numbers with GPU will outperform CPU in speed. This will be mentioned later in the experiment section.

Moreover, warp divergence and occupancy also take critical part in designing CUDA parallelized PSO. Unite tasks for all warps in a block or make it as similar as possible will eliminate warp divergence. Also, by allocating memory carefully and using the technique of reduction can increase the occupancy which means to use the computation ability of the device as far as possible.

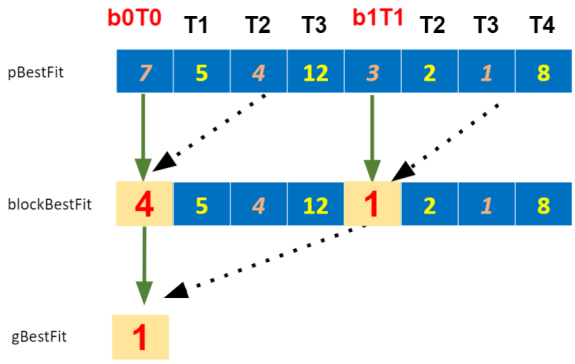

Figure 5: Reduction when calculating BestFit value

The bestFit value calculation is the last parallelized part in a single iteration. We can use the reduction technique to make best use of GPU Streaming Multiprocessor which will increase the occupancy. By the instruction in Nsight profiler, Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps.[5] High occupancy does not mean high performance but low occupancy always lead to low performance. This is intuitive that when a large part of Streaming Multiprocessors are stalled or not working, the parallelization is not successful.

5.2.2 Memory management

There are two strategies to allocate memory for device to use:

1. Structure of arrays.
2. Array of structures.

The following diagram shows how the two strategies work:

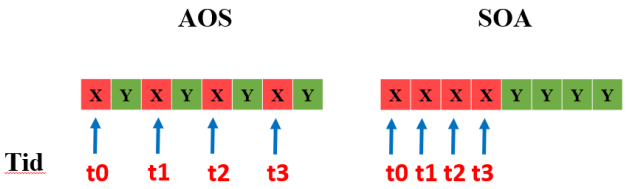

Figure 6: Flowchart for CUDA parallelized PSO

There are two factor that influence the efficiency of CUDA performance:

1. Aligned memory access.
2. Coalesced memory access.

Aligned memory access means that the first address of global memory is an even multiple of the cache length. Coalesced memory access means that 32 threads in a warp access the same chunk of memory but not accessing the same memory. The following diagram shows how these two memory access method works:

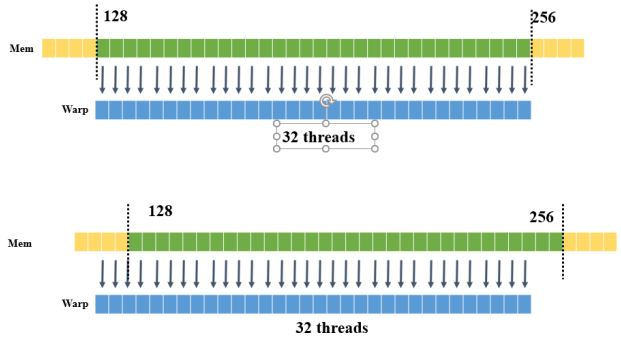

Figure 7: Aligned memory access : The Upper part is aligned memory access. The lower part is not aligned memory access

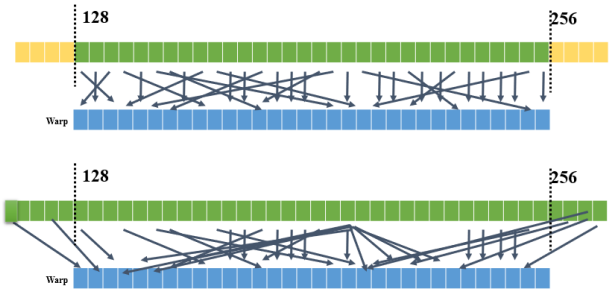

Figure 8: Coalesced memory access : The Upper part is coalesced memory access. The lower part is not coalesced memory access

The memory access in our parallelized PSO remains aligned by not offsetting our global memory. The constant factors is using the read-only memory. Also, the memory access is coalesced due to our memory allocation is structure of arrays. By examining Figure 6 and Figure 4, we can come to the conclusion that SOA is the better strategy due to Coalesced memory access.

6 Experimental Methodology

The experiment platform is shown as follow:

1. Intel Core i7-9750H CPU 12 core.
2. NVIDIA Geforce RTX 2060/PCIe/SSE2.

The detailed content for the experiment platform is as Figure 7:

Figure 9: Experiment platform

We use Qt to construct an experiment environment. The environment is shown in Figure 8.

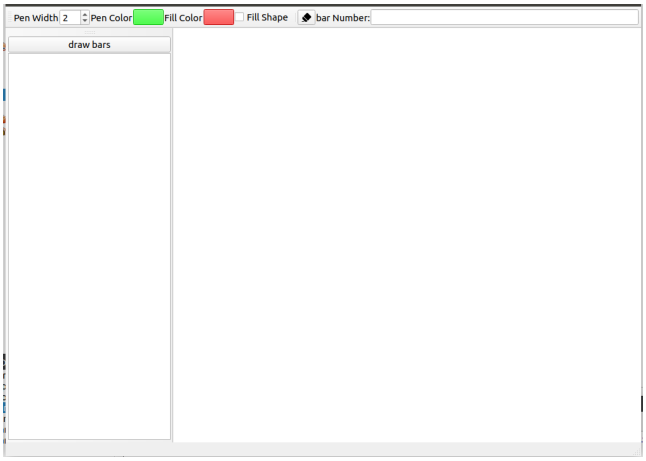

Figure 10: Experiment environment

The experiment environment is designed to import all the factor , initial point and target point needed for PSO algorithm to solve inverse kinetics by txt file. Moreover, serialized or parallelized by specific platform should be mentioned clearly in the file. After importing the environment will call the specific executable and communicate with it by posix shared memory before, during and after the algorithm is performing. When it is performing, the environment will draw the global best bar. When it is done, the fitness, final swarm and time interval should be shown at the left segment of the main window. Figure 9 shows the main window when the algorithm is performing. Note that using this application will cause synchronization, latency and also additional memory shifting to the PSO algorithm for Inverse Kinetics no matter if it is serialized or parallelized by any platforms.

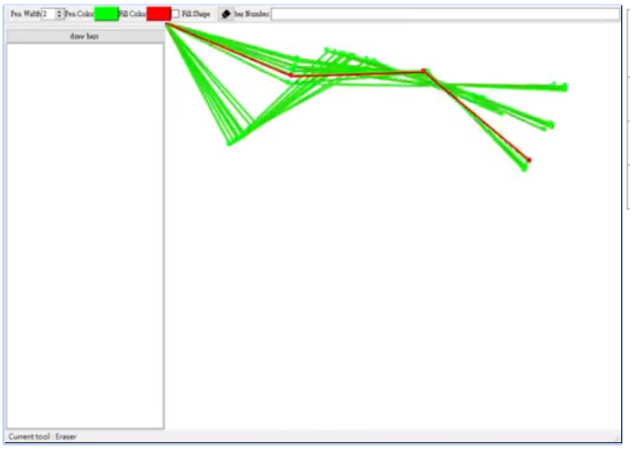

Figure 11: Experiment environment when serialize 3000iteration PSO Inverse kinetics is running.

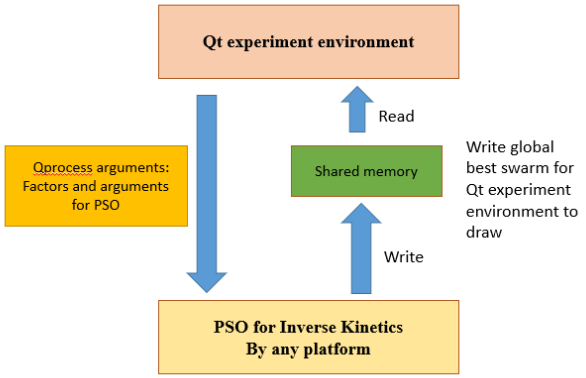

Figure 12: Experiment environment when serialize 3000iteration PSO Inverse kinetics is running.

Figure 10 shows how this experiment environment works. The initial factor and arguments are passed by experiment environment when calling the PSO for Inverse Kinetics executable. The temporary global best swarm , the final global best swarm, the final fitness and the time elapse is passed by shared memory which is a Inter Process Communication in linux platform to the experiment environment.

The advantage of this platform is that it is decoupled to parallelized platforms or algorithms. And also, it mimics how real world application works. The computation core is usually separated from the control core. Moreover, this platform can be modified to a demo application in part 7 which will use the PSO for Inverse Kinetics application to trace mouse.

7 Experimental result

7.1 OpenMP parallelized PSO evaluation

The analysis tool for the OpenMP version is Linux Perf_events, which is a sampled-based analysis tool. It means the tool Perf_event doesn't affect the performance of the program (too much). We ran the program several times with a certain number of threads specified each and then averaged the statistics out. We increased the threads from 1 to 10 by the following function call:

```
opm_set_num_thread(NUM_THREAD)//NUM_THREAD = 1..10;
```

The program ran on our another **6-core** Intel-i5 series machine.We present the the following outcome from perf_events :

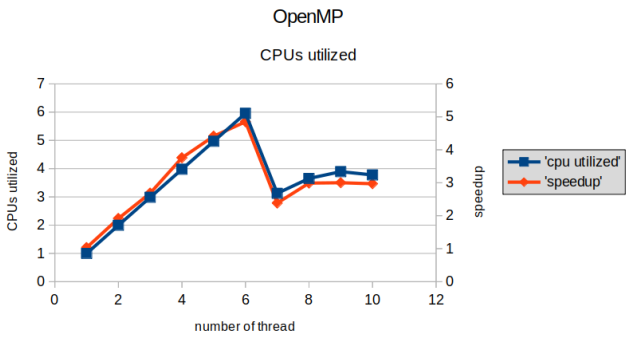

Figure 13: CPU utilized vs speedup

As we can see, the trend between speed up and CPU utilized is pretty matched. The scalability is limited to the same amount of the number of the cores. As the number of threads increased further than 6, the efficiency dropped quite dramatically. The trend of context switching might be able to explain the phenomena.

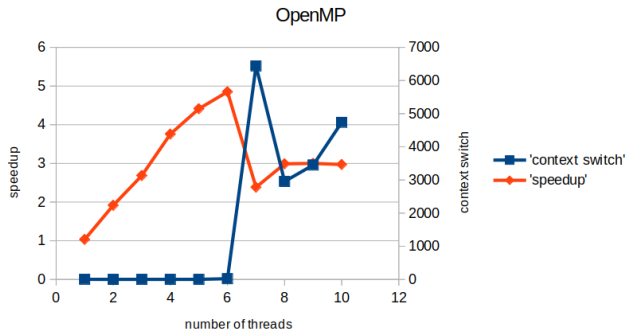

Figure 14: context switch vs speedup

The context switch peaks as the number of threads is equal to 7. Context switch takes wall time to I/O instead of CPU time. The context switch(per second) is over 6000 when NUM_THREAD=7 versus about 22 (per seconds) when NUM_THREAD = 6.

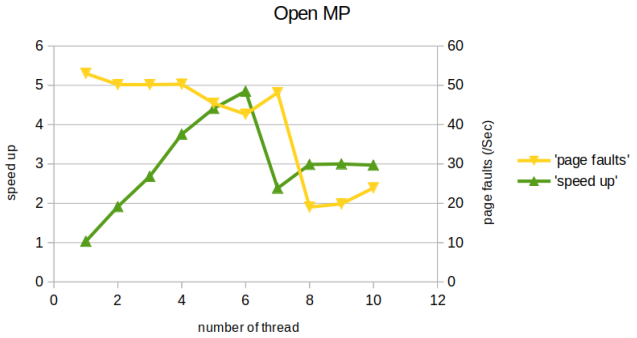

Figure 15: page faults vs speedup

Interestingly, the page fault drop as the number of threads is greater than 6. It seems, however, barely help. As for the reason for the drop of page fault might be that multiple threads shared the same core. So threads can use shared cache.

In addition to what've been stated above, we also observed that OS didn't spread thread to every core. Take the following figure for example. Though the number of threads was set to be 7, the CPU cores actually got used was not 6 but 5.

```
thread 0 executes on CPU:4
thread 6 executes on CPU:1
thread 1 executes on CPU:2
thread 5 executes on CPU:3
thread 4 executes on CPU:0
thread 2 executes on CPU:2
thread 3 executes on CPU:0
```

The following figure shows another example as the number of threads was set to 6 and only 5 cores were used. We tested it several times, however , in most case all CPU tend to be utilized.

```
thread 2 executes on CPU:1
thread 3 executes on CPU:3
thread 4 executes on CPU:4
thread 1 executes on CPU:0
thread 0 executes on CPU:3
thread 5 executes on CPU:0
```

Those phenomena may have something to do with CPU Affinity. We should able to specify it with:

```
int sched_setaffinity(pid_t pid, size_t cpusetsize,cpu_set_t *mask);
```

But we are not going to discuss it in this report and we believe it might improve the performance even further.

7.2 CUDA parallelized PSO evaluation

The evaluation of the results of CUDA parallelized PSO can be divided into three part:

1. Factor to fitness.
2. Factor to time..
3. Compare to serialized PSO
4. Occupancy after optimization

Figure16 to Figure 18 shows the difference of the log2(Particle Number) to fitness when iteration varies. Figure 19 shows log2(Particle Number) to the time diagram. Figure 20 shows the iteration to time diagram. Figure 21 shows the log2(Particle Number) to time ratio between Serialized PSO and CUDA parallelized PSO. . Figure 22 shows the iteration to time ratio between Serialized PSO and CUDA parallelized PSO. Figure 23 shows the improvement of occupancy when using SOA technique.. Figure16 to Figure18 belongs to part1. Figure19 and Figure20 belong to part 2. Figure21 and Figure22 belongs to part 3. Figure23 belongs to part4.

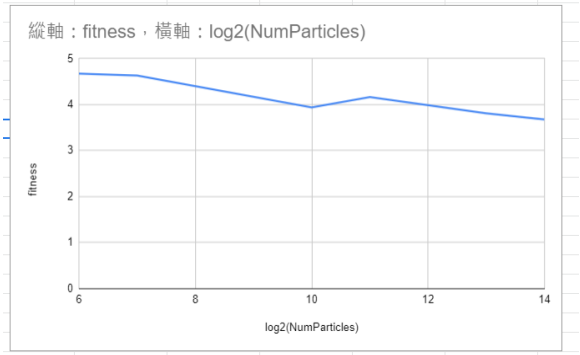

Figure 16: 10 iteration: log2(NumParticle) to fitness

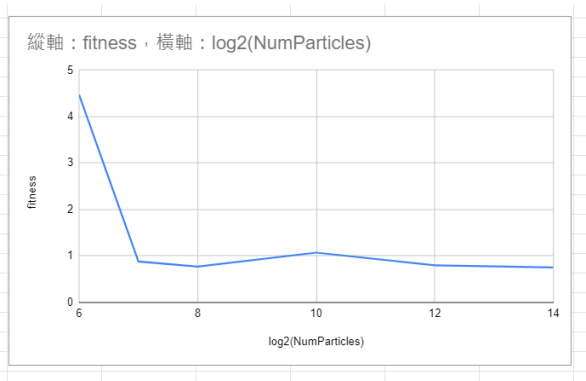

Figure 17: 100 iteration: log2(NumParticle) to fitness

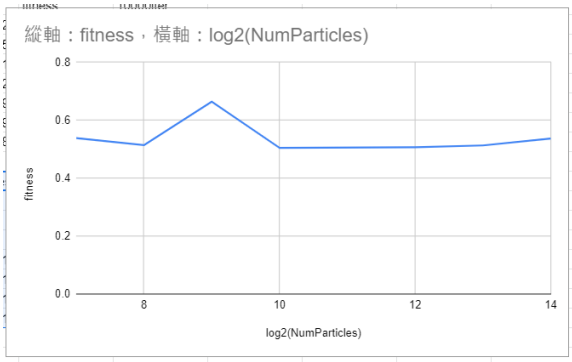

Figure 18: 10000 iteration: log2(NumParticle) to fitness

Interestingly, we can come to the conclusion below by examining the results from Figure16 to Figure18:

1. The particle number plays an important role only when iteration is less..
2. 10000 iterations is much more accurate than others no matter the particle number is.
3. We can decrease the fitness by increase the Particle number, iterations or both

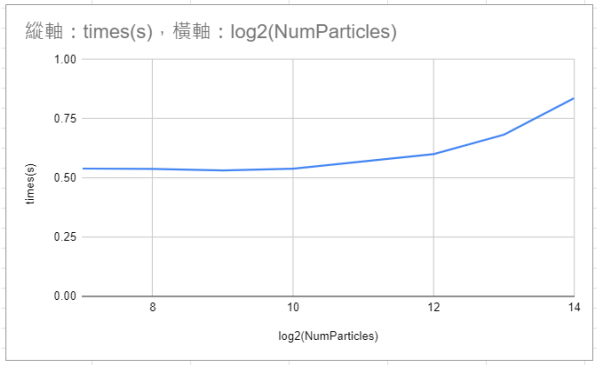

Figure 19:10000 iteration: log2(NumParticle) to time

By Figure19, we can figure out that the time needed for computing remains flat at first and slightly increases at last when the particle number increases. This result shows that the efficient gain by parallelization is significant. The slight increase at last can be result in the latency caused by synchronization become significant when the particle number is large..

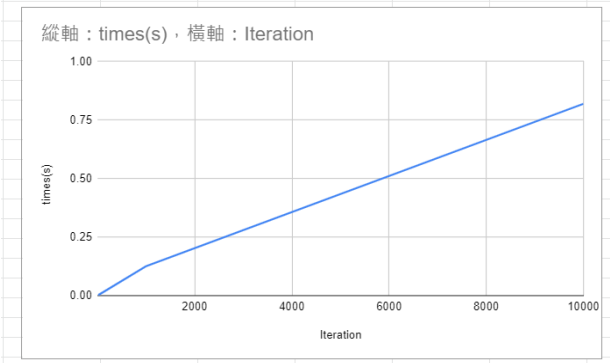

Figure 20:16384NumParticles: iteration to time

Due to the modal didn't parallelizing through iteration, The time needed for computation increases as the iteration increases. It's almost a straight line. This result meets the expectation of the parallelization model introduced in Section 5.2.

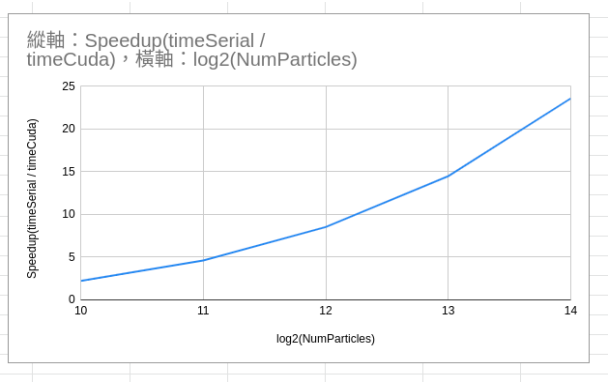

Figure 21: Flowchart for CUDA parallelized PSO

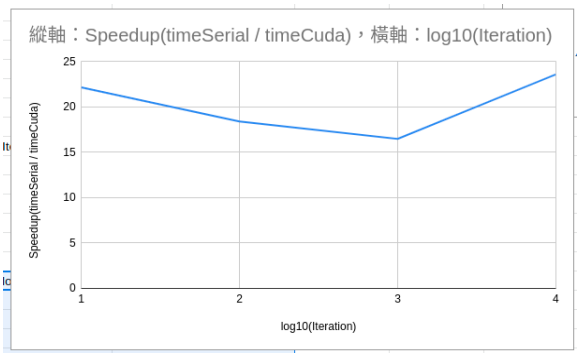

Figure 22: Flowchart for CUDA parallelized PSO

Figure21 shows that the time ratio between Serialized PSO and CUDA parallelized PSO when log2(particle number) increases. Figure22 shows that the time ratio between Serialized PSO and CUDA parallelized PSO when log10(iteration) increases. We can figure out that the speedup ratio increases from about 1 to nearly 25 when the log2(particle number) increases. But, the speedup ratio remains between 15 to 25 when log10(iteration) increases. we can come to the conclusion below by examining the results :

1. The speedup is significant if the particle number is not very low.
2. Apart from low particle number, other factors like iterations won't decrease the efficiency of CUDA parallelized PSO.
3. We can gain more than 20X speedup when the particle number is high. This may also get more accurate results.

Theoretical Occupancy [%]	100
Theoretical Active Warps per SM [warp]	32
Achieved Occupancy [%]	60.89

Figure 23: Flowchart for CUDA parallelized PSO

Figure23 shows that the achieved occupancy increases from nearly 25 to 60.89 after the AOS memory structure is changed to SOA structure. Also, the Aligned memory access remains and the data structure is tuned slightly.

7.3 Drawing application result evaluation

The results of the PSO algorithm in the Inverse kinetics problem with the application experiment environment is shown below. Figure 5-1 and Figure 5-2 shows the result of serialized PSO. Figure 5-3 and Figure 5-4 shows the result of parallelized PSO by openMP. Figure 5-5 and Figure 5-6 show the result of parallelized PSO by CUDA.

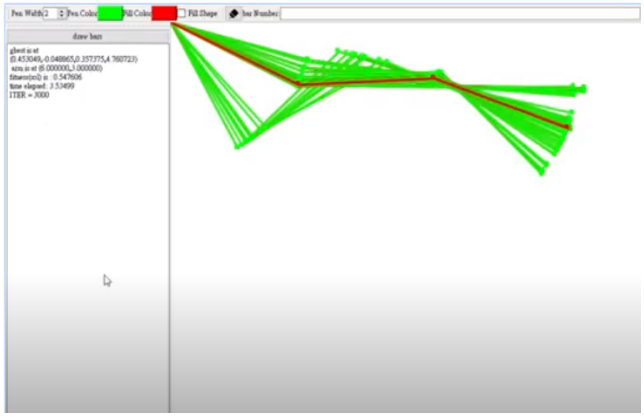

Figure 24:Serialized PSO for Inverse Kinetics demo 15 times slower

gBest is at: a1:0.453, a2:0.489, a3:0.357,d2: 4.791
arm position:(6.0000, 3.000)
time elapsed: 3.53499s
fitness: 0.505182

Figure 25: The performance table of serialized PSO for Inverse Kinetics demo 15 times slower

Figure24 and Figure25 show the result of serialized PSO. By the recorded video, we can figure out that sometimes it needs a little amount of iterations to converge, but sometimes it needs a large amount of iterations to converge. PSO is an algorithm that uses a lot of random

numbers, which makes it an algorithm full of randomness. However, it can always converge to a satisfied fitness after 10000 iterations.

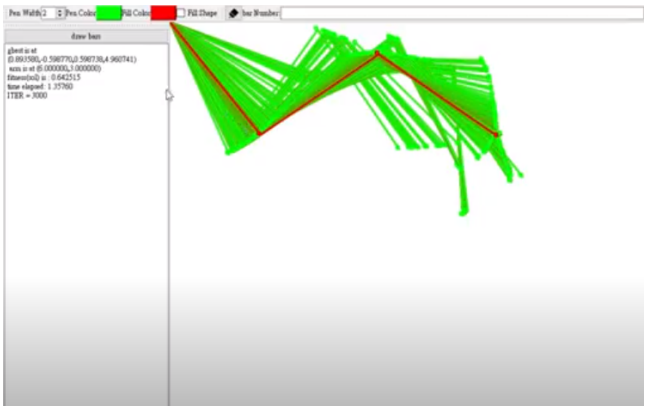

Figure 26: Parallelized PSO by openMP for Inverse Kinetics demo 15 times slower

gBest is at: a1:0.894, a2:-0.599, a3:0.599,d2: 4.960
arm position:(6.0000, 3.000)
time elapsed: 0.6425s
fitness: 0.505182

Figure 27: The performance table of parallelized PSO by openMP for Inverse Kinetics demo 15 times slower

Figure26 and Figure27 show the result of parallelized PSO by openMP. The speed up is 6.99745X which is significant. The time needed for parallelized PSO by openMP meets the requirement for tracing mouse applications in 7.4. The fitness is also fine enough. Though in this experiment, it took a lot of iterations to converge as the diagram shows. It's not always the case. In some other experiments, it took only a little iterations to converge.

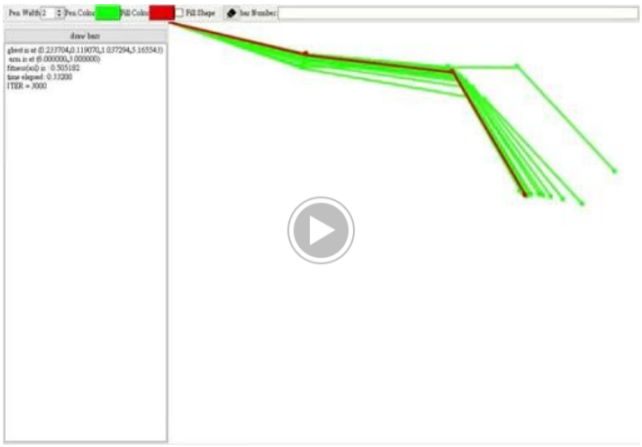

Figure28: Parallelized PSO by CUDA for Inverse Kinetics demo 15 times slower

gBest is at: a1:0.234, a2:0.119, a3:1.037,d2: 5.163
arm position:(6.0000, 3.000)
time elapsed: 0.332s
fitness: 0.505182

Figure29: The performance table of parallelized PSO by CUDA for Inverse Kinetics demo 15 times slower

Figure28 and Figure29 show the result of parallelized PSO by CUDA. The speedup is up to 10.6468X which is much higher than openMP and serialized one. However, we can observe that the speedup is not as much as the result in 7.2. We know that in this experiment. The gBest swarm and fitness need to be passed from algorithm application to experiment environment application. This is the same with the serialized one. Moreover, in each iteration, the data has to pass from device to host. This causes more latency, synchronization, and memory shifting time than serialized one. As a result, the speedup is not as good as the parallelized PSO algorithm in 7.3.

7.4 Tracing mouse application evaluation

Figure30: Mouse trace application

According to the results in section 7.3, the two parallelized PSO algorithms for Inverse Kinetics all meet the requirement for real time tracing. The demo of mouse tracing using parallelized PSO algorithms by CUDA is shown in Figure30. The three bars can adjust it's rotation and translation immediately by getting the result from shared memory and let the endpoint follow the mouse closely. The red line is the trace of the mouse and the three green lines and the three green points represent the bars, the joints and the endpoint.

8 Related work

The parallelized PSO research can be sorted into two categories:Synchronous and Asynchronous parallelized PSO. or just SPPSO and APPSO. J. F. Schutte et. al. publish [1] in 2004. They first compare the performance of **Serial** Synchronous PSO (or SSPSO) and **Serial** Asynchronous PSO (or SAPSO). In most cases, the ASPSO outperforms the SSPSO. Since it can share information more efficiently. When it comes to parallelization. The asynchronous implementation lost the advantage. They deemed it as the overhead of parallelization. Then They discussed coherence. Parallelization should have no adverse effect on algorithm operation. The parallelized algorithm should lead to exactly the same result as the original program. This is the reason why asynchronous implementation is complicated and loses the advantage.

we found [2] discussed asynchronous implementation. They stated that SPPSO can better make efficient use of computational power. When three conditions meet, however, SPPSO work best. First, the optimization has total and undivided access to a homogeneous cluster of computers without interruptions from other users. Second, the analysis function takes a constant amount of time to evaluate any set of design variables throughout the optimization. Third, the number of parallel tasks can be equally distributed among the available processors. As a result, we implemented SPOSO in our project.

On the other hand, solving the equation (1) and equation(2) shown in 1.1 is actually solving linear algebra problems. Plus the nature of Synchronous implementation, we believe we can gain extraordinary computational efficiency using GPU, which is famous for solving linear algebra, matrix operation and tensor etc. We were motivated by the memory model proposed in [3]. It also mentioned Rastrigin's function, which is regularly used when it comes to evaluate global optimization algorithms. So that we can focus more on developing the algorithm instead of generating a library of test cases.

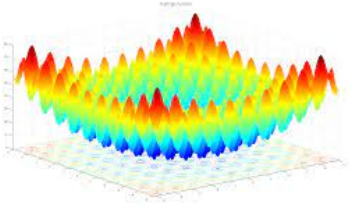

Figure31: Rastrigin's function in 2D space

9 Conclusion

All the results in section 7 prove that our parallelized PSO for Inverse Kinetics works much better than serialized PSO. Moreover, the fitness of the parallelized PSO does not show significant decreases than serialized PSO. We explored the details of the technique of parallelization including occupancy, latency, memory access model, CPU affinity issue and further increased the speedup time.

Second, we construct an experiment environment by inter process communication using posix shared memory. This environment is highly decoupled from the algorithm which is suitable for evaluating performance.

At last, an application mimicking real world application in tracing moving things by Inverse Kinetics is introduced. We can successfully tracing mouse moving by parallelized PSO for Inverse Kinetics.

ACKNOWLEDGMENTS

THANK YOU TEACHER AND T.A. WE LOVE YOU!!!

REFERENCES

[1] J. F. Schutte , J. A. Reinbolt.,2004. Parallel global optimization with the particle swarm algorithm, International Journal for Numerical Methods in Engineering. 2004 December 7

[2] Byung-Il Koh , Alan D. George ,2006. Parallel asynchronous particle swarm optimization, International Journal for Numerical Methods in Engineering. 2006 July 23

[3] Luca Mussi,Stefano Cagn, 2009. Particle swarm optimization within the CUDA architecture, Conference: Genetic and Evolutionary Computation Conference - GECCO 2009