FastPSO: Towards Efficient Swarm Intelligence Algorithm on GPUs

Hanfeng Liu\textsuperscript{1}, Zeyi Wen\textsuperscript{2}, Wei Cai\textsuperscript{1}

\textsuperscript{1}The Chinese University of Hong Kong, Shenzhen, China
\textsuperscript{2}The University of Western Australia

zeyi.wen@uwa.edu.au
Outline

• Background
• Our proposed techniques
• Experimental results
• Conclusion
Swarm Intelligence

• mimics behaviors of social animals (e.g., ants and bees),
• exploits information exchanges among individuals in the group to achieve intelligence.

Ant Colony Optimization

The Bee Algorithm
Particle Swarm Optimization (PSO)

- A type of swarm intelligence; simple but effective
- Wide range of applications (e.g., neural architecture search)
- The update of the swarm can be extremely slow
  - dealing with high dimensional problems
  - having to use a large number of particles

Bees, ants, migratory birds, ...
Research papers with PSO

Figure 1: Number of publications of PSO in recent years
Current global best position

Local best position of each ant

Visited positions

\[ \mathbf{v}_g \]

\[ \mathbf{v}_p \]

Ant 1

Ant 2

Ant 3

Ant 4
Key Steps of PSO

• swarm initialization
• swarm evaluation
• $p_{best}$ and $g_{best}$ update
• swarm update
Bottleneck of PSO—Breakdown
The PSO Algorithm

- The goal is to find the global optimum by the particles.
- Each particle has a velocity ($v_i$) and a position ($p_i$).

$$v_i' = \omega v_i + c_1 l_i \odot (p_{best_i} \cdot e - p_i) + c_2 g \odot (g_{best_i} \cdot e - p_i) \quad (1)$$

$$p_i' = p_i + v_i' \quad (2)$$

- $\omega$: particle momentum
- $c_1, c_2 \in (0, 1)$: random weight vector
- $l_i, g_i$: preference to explore locally/globally
- $e = [1, 1, \ldots, 1] \in \mathbb{R}^d$

**Figure 1**: Updating position and velocity of $i$-th particle
Potential GPU Acceleration

• One GPU thread per particle [SYNASC’16]

• GPU for swarm update and CPUs for the rest [TPDS’17]
  ▪ one GPU thread per particle

• One GPU thread for a dimension of a particle [ours]
Outline

• Background
• Our proposed techniques
• Experimental results
• Conclusion
Overview of Our Method
Swarm Initialization

• Initialization of multiple $d \times n$ matrices
  ▪ Initialization of position ($p$)
  ▪ Initialization of velocity ($v$)
  ▪ Initialization of $l$ and $g$

• Each particle has a velocity ($v_i$) and a position ($p_i$).

$$v'_i = \omega v_i + c_1 l_i \odot (p_{best_i} \cdot e - p_i) + c_2 g \odot (g_{best_i} \cdot e - p_i)$$  \hspace{1cm} (1)

$$p'_i = p_i + v'_i \hspace{1cm} (2)$$
Swarm Evaluation on GPUs

• One GPU thread evaluates the fitness of a particle
• A schema to customize swarm evaluation functions

```cpp
template<typename L>
__global__ void evaluation_kernel(int dim, L lambda){
    for(int i = blockIdx.x * blockDim.x + threadIdx.x;
        i < dim; i += blockDim.x * gridDim.x) {
        lambda(i);
    }
}

Figure 2: Swarm evaluation schema
Element-wise Operations

\[ v' = \omega \cdot v + c_1 \cdot L \odot (\varepsilon_i - p) + c_2 \cdot G \odot (\varepsilon_g - p) \]  

(3)

\[ v = \begin{bmatrix} v_{11} & \cdots & v_{1d} \\ v_{21} & \cdots & v_{2d} \\ \vdots & \ddots & \vdots \\ v_{n1} & \cdots & v_{nd} \end{bmatrix}, \quad p = \begin{bmatrix} p_{11} & \cdots & p_{1d} \\ p_{21} & \cdots & p_{2d} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nd} \end{bmatrix}, \quad L = \begin{bmatrix} l_{11} & \cdots & l_{1d} \\ l_{21} & \cdots & l_{2d} \\ \vdots & \ddots & \vdots \\ l_{n1} & \cdots & l_{nd} \end{bmatrix}, \]

\[ G = \begin{bmatrix} g_{11} & \cdots & g_{1d} \\ g_{21} & \cdots & g_{2d} \\ \vdots & \ddots & \vdots \\ g_{n1} & \cdots & g_{nd} \end{bmatrix}, \quad \varepsilon_i = \begin{bmatrix} p_{best_1} & \cdots & p_{best_1} \\ p_{best_2} & \cdots & p_{best_2} \\ \vdots & \ddots & \vdots \\ p_{best_n} & \cdots & p_{best_n} \end{bmatrix}, \]

\[ \varepsilon_g = \begin{bmatrix} \text{gbest} & \cdots & \text{gbest} \\ \text{gbest} & \cdots & \text{gbest} \\ \vdots & \ddots & \vdots \\ \text{gbest} & \cdots & \text{gbest} \end{bmatrix} \]

- Each particle has a velocity \((v_i)\) and a position \((p_i)\).

\[ v'_i = \omega v_i + c_1 l_i \odot \left( p_{best_i} \cdot e - p_i \right) + c_2 g \odot \left( gbest_i \cdot e - p_i \right) \]  

(1)

\[ p'_i = p_i + v'_i \]  

(2)
Workload Allocation

Each thread in the GPU is responsible for the update of a particle value, the GPU thread workload of FastPSO as follows.

\[ tw = \frac{n \times d}{mem} \]

where \( n \) denotes to the number of particles, \( d \) is the particle dimension and \( mem \) is the GPU memory.

The velocity and particle update is constrained by the follow equation:

\[ v_{ij} = \begin{cases} 
    lower\_bound_{ij} & \text{if } v_{ij} < lower\_bound_{ij} \\
    upper\_bound_{ij} & \text{if } v_{ij} > upper\_bound_{ij} \\
    v_{ij} & \text{otherwise}
\end{cases} \]
Shared Memory and Tensor Cores

• The velocity/position matrix may be too large.
  ▪ segmented into multiple sub-matrices, copied to shared memory

• Making use of Tensor Cores
  ▪ assigned to the fragment of tensor cores

• Using memory caching to reduce GPU memory allocation

• Two approaches to support multiple GPUs
  ▪ Each GPU is responsible for a subset of particles
  ▪ Each GPU is responsible for a tiled sub-matrix
Outline

• Background
• Our proposed techniques
• Experimental results
• Conclusion
Experimental Setup

• 2 Xeon E5-2640v4 10 core CPUs, a Tesla V100 16G GPU
• CUDA-C and compiled with -O3 option
• 2000 particles, 2000 iterations, 200 dimensions

• Search problems
  ▪ Sphere: \( f(x) = \sum_{i=1}^{d} x_i^2 \), \( x \in (-5.12, 5.12) \)
  ▪ Griewank: \( f(x) = \frac{1}{4000} \sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d} \cos \left( \frac{x_i}{\sqrt{i}} \right) + 1 \), \( x \in (-600, 600) \)
  ▪ Easom: \( f(x) = -(-1)^d (\prod_{i=1}^{d} \cos^2(x_i)) \exp[-\sum_{i=1}^{d} (x_i - \pi)^2] \), \( x \in (-2\pi, 2\pi) \)
  ▪ TheadConf: Optimize the block configuration for a GPU program
Experimental Setup

• Baselines
  ▪ *pyswarms*: a PSO algorithm implemented in Python
  ▪ *scikit-opt*: a toolkit of swarm intelligence algorithms in Python
  ▪ *gpu-ps* [SYNASC’16]: a GPU based PSO implementation
  ▪ *hgpu-ps* [TPDS’17]: a heterogeneous multi-core implementation

• Our implementations
  ▪ *fastps-seq*: CPU based sequential PSO
  ▪ *fastps-omp*: CPU based multi-threads PSO using *OpenMP*
  ▪ *fastps*: GPU based PSO implementation
Overall Comparison

Table 1: Overall comparison of FastPSO against other implementations

<table>
<thead>
<tr>
<th>problem</th>
<th>elapsed time (sec)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pyswarms</td>
<td>scikit-opt</td>
</tr>
<tr>
<td>Sphere</td>
<td>129.67</td>
<td>88.98</td>
</tr>
<tr>
<td>Griewank</td>
<td>80.94</td>
<td>172.17</td>
</tr>
<tr>
<td>Easom</td>
<td>126.89</td>
<td>12.77</td>
</tr>
</tbody>
</table>

Table 2: Errors to the optimal values

<table>
<thead>
<tr>
<th>implementation</th>
<th>Sphere</th>
<th>Griewank</th>
<th>Easom</th>
</tr>
</thead>
<tbody>
<tr>
<td>pyswarms</td>
<td>1031.99</td>
<td>2965.27</td>
<td>0.00</td>
</tr>
<tr>
<td>scikit-opt</td>
<td>2483.61</td>
<td>8892.36</td>
<td>0.00</td>
</tr>
<tr>
<td>gpu-pso</td>
<td>23.72</td>
<td>0.69</td>
<td>0.00</td>
</tr>
<tr>
<td>hgpu-pso</td>
<td>15.06</td>
<td>0.31</td>
<td>0.00</td>
</tr>
<tr>
<td>fastpso-seq</td>
<td>26.98</td>
<td>0.66</td>
<td>0.00</td>
</tr>
<tr>
<td>fastpso-omp</td>
<td>22.01</td>
<td>0.72</td>
<td>0.00</td>
</tr>
<tr>
<td>fastpso</td>
<td>23.62</td>
<td>0.71</td>
<td>0.00</td>
</tr>
</tbody>
</table>
FLOPs and Memory Bandwidth

- Much higher throughput than other GPU implementations.
- GFLOPs: similar on different implementations.

**Table 3: FLOPs and memory bandwidth**

<table>
<thead>
<tr>
<th>metrics</th>
<th>dram read throughput (GB/s)</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpu-pso</td>
<td>61.83</td>
<td>1.19</td>
</tr>
<tr>
<td>hgpu-pso</td>
<td>57.41</td>
<td>0.97</td>
</tr>
<tr>
<td>fastpso</td>
<td>106.94</td>
<td>8.68</td>
</tr>
</tbody>
</table>
Conclusion

• We present FastPSO which is 5-7 times faster than its GPU-based counterparts and is two orders of magnitude faster than the existing CPU-based libraries.

• We located the bottleneck of PSO and investigated different techniques to accelerate PSO on GPUs.

• Varieties of experiments were conducted to study the efficiency of our techniques.
The End

Thanks!