INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING
ICPP / 2021 / CHICAGO / USA
acm In-Cooperation
sighpc
AUGUST 9-12, 2021

FastPSO: Towards Efficient Swarm Intelligence Algorithm on GPUs

Hanfeng Liu[1], **Zeyi Wen**[2], Wei Cai[1]

[1]The Chinese University of Hong Kong, Shenzhen, China
[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

THE UNIVERSITY OF WESTERN AUSTRALIA

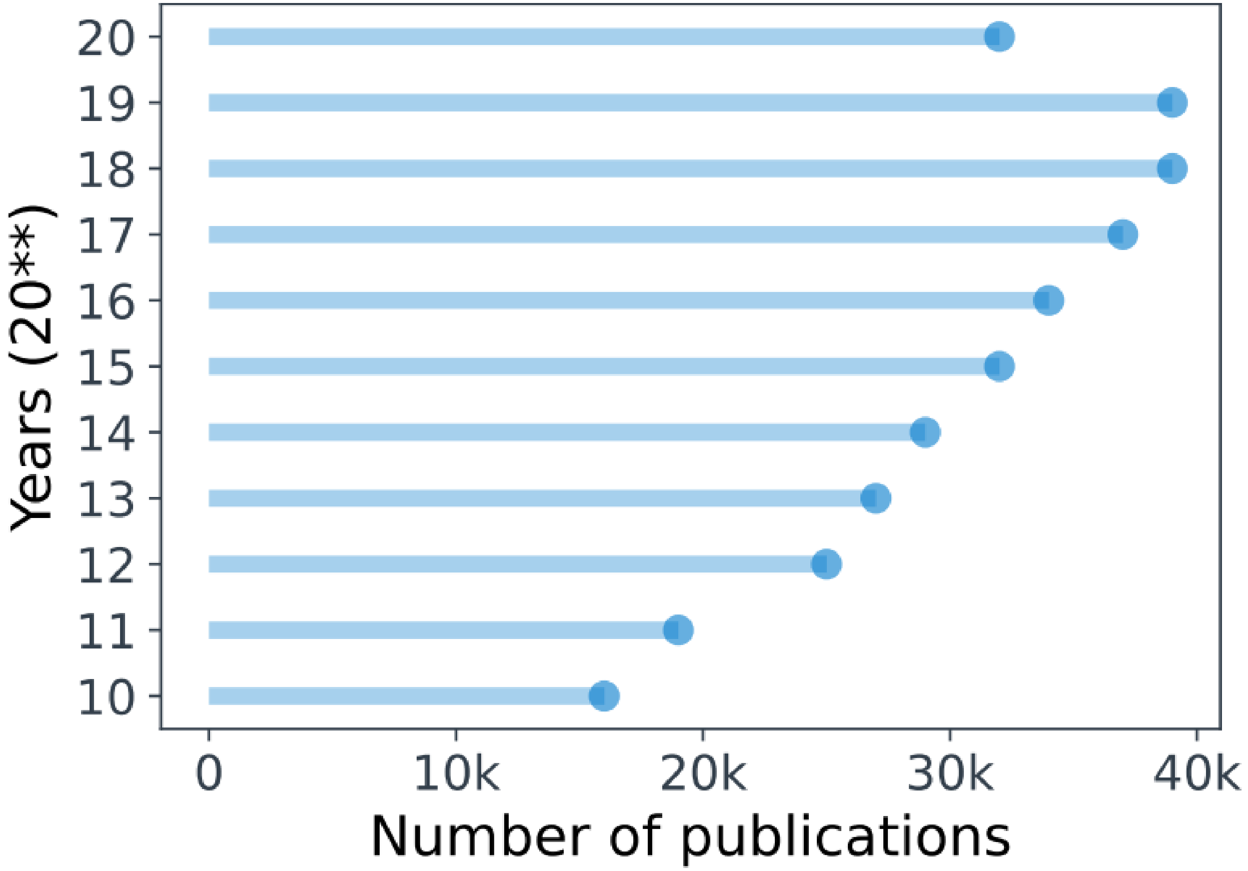
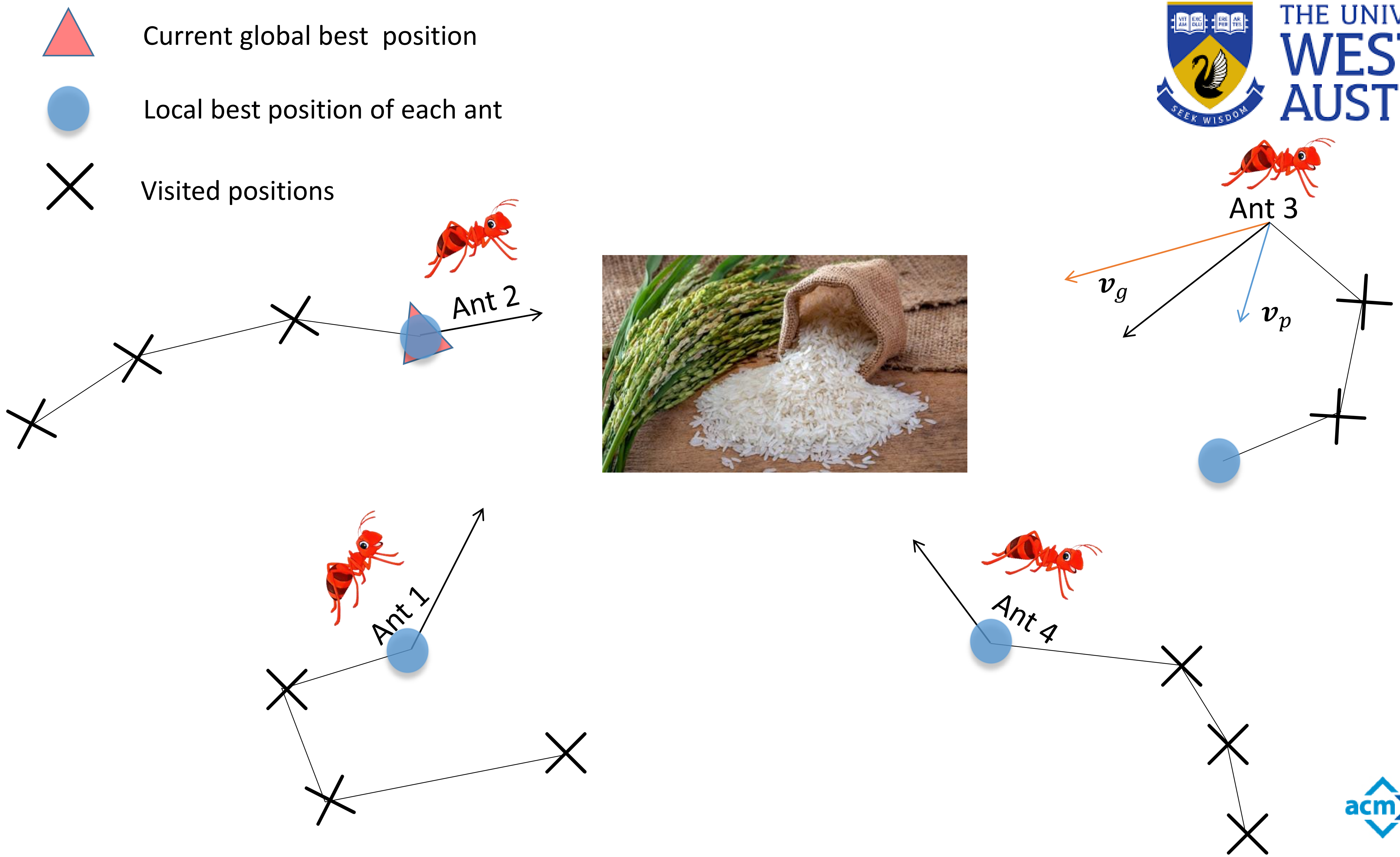


Figure 1: Number of publications of PSO in recent years

acm In-Cooperation sighpc

THE UNIVERSITY OF WESTERN AUSTRALIA

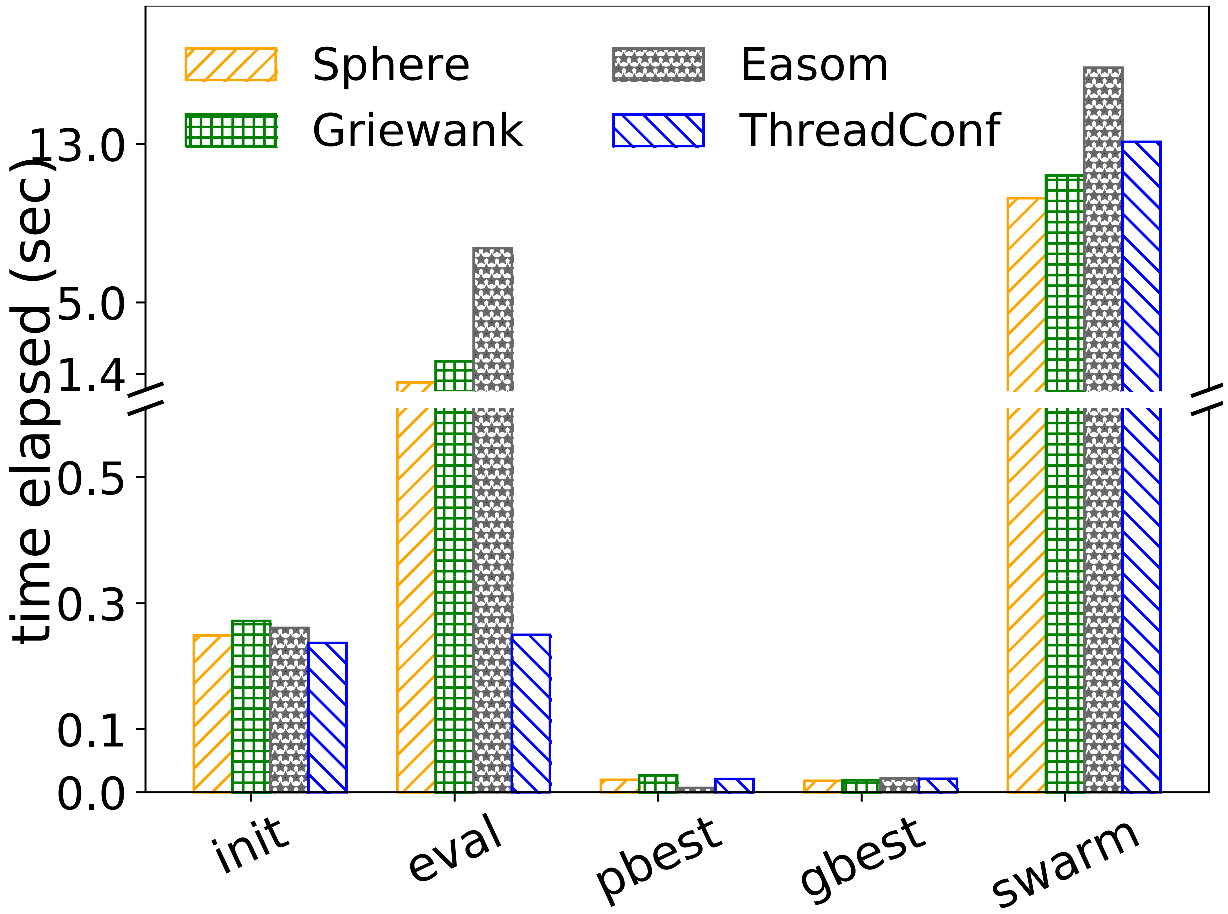


acm In-Cooperation sighpc

Key Steps of PSO

- swarm initialization
- swarm evaluation
- *pbest* and *gbest* 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** ($\boldsymbol{v}_i$) and a **position** ($\boldsymbol{p}_i$).

$$\boldsymbol{v}_i' = \omega \boldsymbol{v}_i + c_1 \boldsymbol{l_i} \odot (pbest_i \cdot \boldsymbol{e} - \boldsymbol{p}_i) + c_2 \boldsymbol{g} \odot (gbest_i \cdot \boldsymbol{e} - \boldsymbol{p}_i) \qquad (1)$$

$$\boldsymbol{p}_i' = \boldsymbol{p}_i + \boldsymbol{v}_i' \qquad \mathbf{(2)}$$

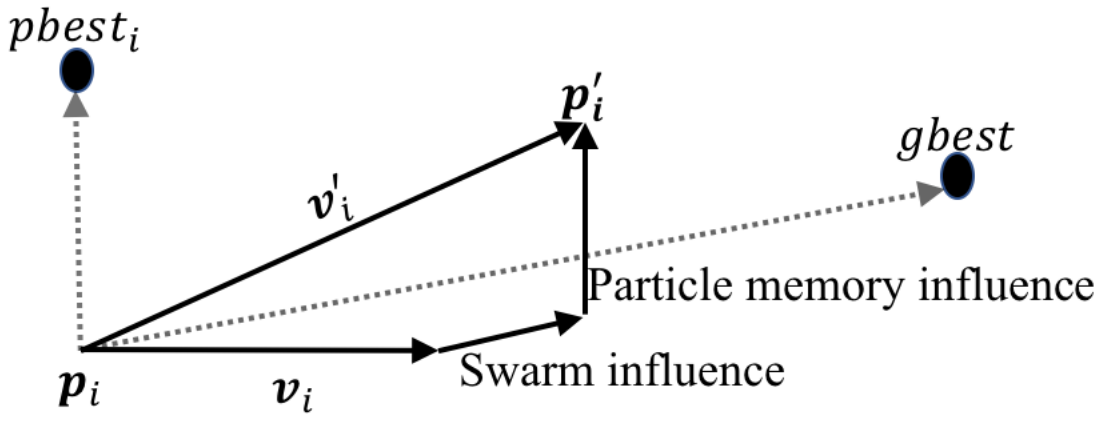


Figure 1: Updating position and velocity of *i* -th particle

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

Potential GPU Acceleration

- One GPU thread per particle [*SYNASC'16*]

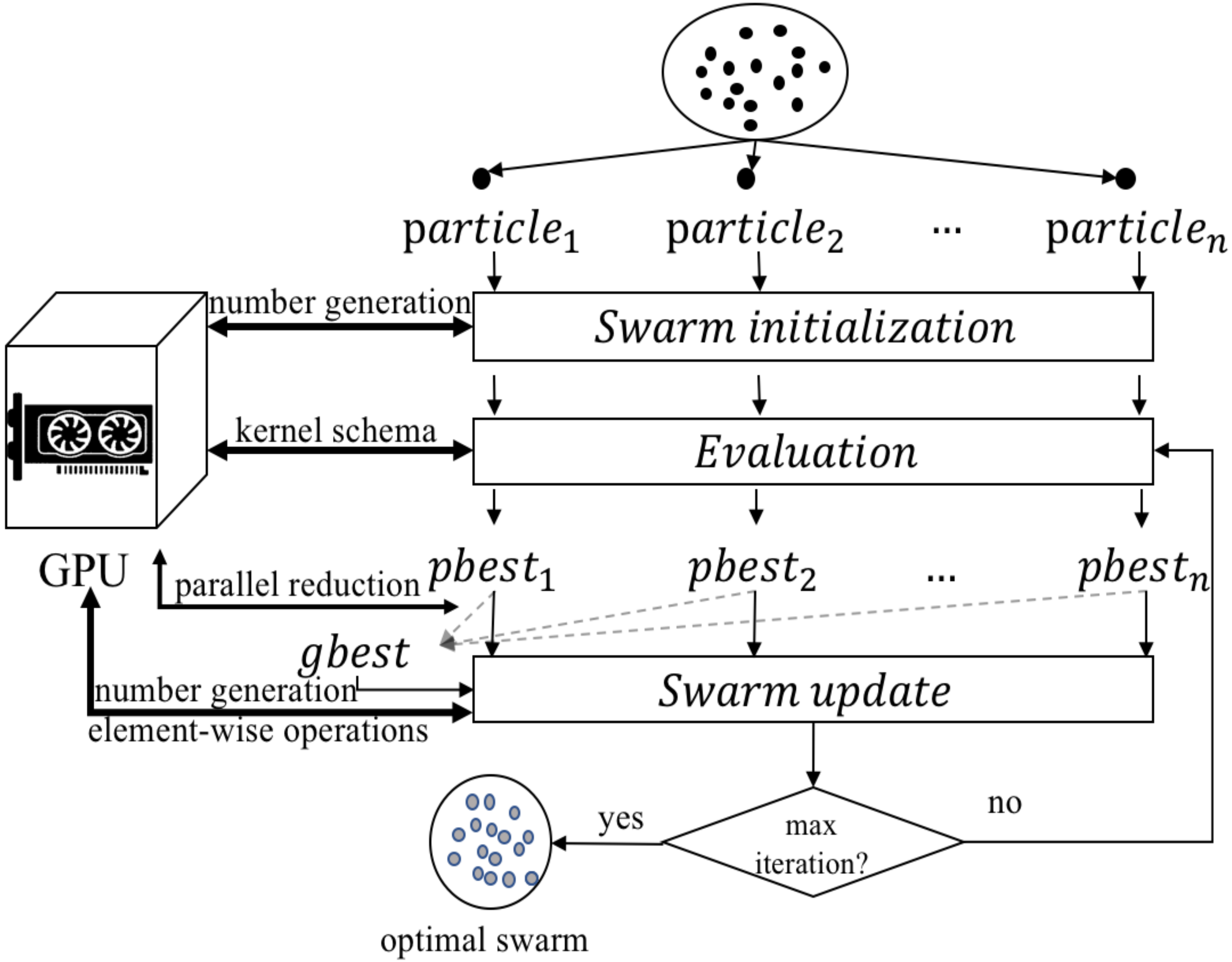
- GPU for swarm update and CPUs for the rest [*TPDS'17*]
  - one GPU thread per particle

- 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 × n matrices
  - Initialization of position (***p***)
  - Initialization of velocity (***v***)
  - Initialization of ***l*** and ***g***

- Each particle has a **velocity** ($\boldsymbol{v}_i$) and a **position** ($\boldsymbol{p}_i$).

$$\boldsymbol{v}_i' = \omega \boldsymbol{v}_i + c_1 \boldsymbol{l}_i \odot (pbest_i \cdot \boldsymbol{e} - \boldsymbol{p}_i) + c_2 \boldsymbol{g} \odot (gbest_i \cdot \boldsymbol{e} - \boldsymbol{p}_i) \qquad (1)$$

$$\boldsymbol{p}_i' = \boldsymbol{p}_i + \boldsymbol{v}_i' \qquad (2)$$

Swarm Evaluation on GPUs

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

```
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

$$\mathcal{V}' = \omega \cdot \mathcal{V} + c_1 \cdot \mathcal{L} \odot (\mathcal{E}_l - \mathcal{P}) + c_2 \cdot \mathcal{G} \odot (\mathcal{E}_g - \mathcal{P}) \qquad (3)$$

$$\mathcal{V} = \begin{bmatrix} v_{11} & \cdots & v_{1d} \\ v_{21} & \cdots & v_{2d} \\ \vdots & \ddots & \vdots \\ v_{n1} & \cdots & v_{nd} \end{bmatrix}, \quad \mathcal{P} = \begin{bmatrix} p_{11} & \cdots & p_{1d} \\ p_{21} & \cdots & p_{2d} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nd} \end{bmatrix}, \mathcal{L} = \begin{bmatrix} l_{11} & \cdots & l_{1d} \\ l_{21} & \cdots & l_{2d} \\ \vdots & \ddots & \vdots \\ l_{n1} & \cdots & l_{nd} \end{bmatrix},$$

$$\mathcal{G} = \begin{bmatrix} g_{11} & \cdots & g_{1d} \\ g_{21} & \cdots & g_{2d} \\ \vdots & \ddots & \vdots \\ g_{n1} & \cdots & g_{nd} \end{bmatrix}, \mathcal{E}_l = \begin{bmatrix} pbest_1 & \cdots & pbest_1 \\ pbest_2 & \cdots & pbest_2 \\ \vdots & \ddots & \vdots \\ pbest_n & \cdots & pbest_n \end{bmatrix},$$

$$\mathcal{E}_g = \begin{bmatrix} gbest & \cdots & gbest \\ gbest & \cdots & gbest \\ \vdots & \ddots & \vdots \\ gbest & \cdots & gbest \end{bmatrix}$$

- Each particle has a **velocity** ($\boldsymbol{v}_i$) and a **position** ($\boldsymbol{p}_i$).

$$\boldsymbol{v}_i' = \omega \boldsymbol{v}_i + c_1 \boldsymbol{l}_i \odot (pbest_i \cdot \boldsymbol{e} - \boldsymbol{p}_i) + c_2 \boldsymbol{g} \odot (gbest_i \cdot \boldsymbol{e} - \boldsymbol{p}_i) \qquad (1)$$

$$\boldsymbol{p}_i' = \boldsymbol{p}_i + \boldsymbol{v}_i' \qquad (2)$$

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 tenser 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: $f(x) = \sum_1^d x_i^2 \, , x \in (-5.12, 5.12)$
  - Griewank: $f(x) = \frac{1}{4000} \sum_1^d x_i^2 - \prod_1^d cos\left(\frac{x_i}{\sqrt{i}}\right) + 1, x \in (-600, 600)$
  - Easom: $f(x) = -(-1)^d (\prod_1^d cos^2(x_i)) exp[-\sum_1^d (x_i - \pi)^2], x \in (-2\pi, 2\pi)$
  - 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-pso [SYNASC'16]: a GPU based PSO implementation*
  - *hgpu-pso [TPDS'17]: a heterogeneous multi-core implementation*

- Our implementations
  - *fastpso-seq*: CPU based sequential PSO
  - *fastpso-omp*: CPU based multi-threads PSO using *OpenMP*
  - *fastpso*: GPU based PSO implementation

Overall Comparison

THE UNIVERSITY OF WESTERN AUSTRALIA

Table 1: Overall comparison of FastPSO against other implementations

problem	elapsed time (sec)							speedup					
	pyswarms	scikit-opt	gpu-pso	hgpu-pso	fastpso-seq	fastpso-omp	fastpso	pyswarms	scikit-opt	gpu-pso	hgpu-pso	fastpso-seq	fastpso-omp
Sphere	129.67	88.98	4.90	6.01	11.56	8.74	**0.67**	194.41	133.40	7.34	9.01	17.33	13.10
Griewank	80.94	172.17	5.08	7.32	13.78	9.58	**0.66**	123.38	262.46	7.74	11.16	21.00	14.60
Easom	126.89	12.77	5.07	7.22	33.91	24.71	**0.87**	146.35	14.72	5.85	8.33	39.11	28.50
TheadConf	117.670	81.320	4.498	5.477	11.459	6.736	**0.47**	251.97	174.13	9.63	11.73	24.54	14.42

Table 2: Errors to the optimal values

implementation	Sphere	Griewank	Easom
pyswarms	1031.99	2965.27	0.00
scikit-opt	2483.61	8892.36	0.00
gpu-pso	23.72	0.69	0.00
hgpu-pso	15.06	0.31	0.00
fastpso-seq	26.98	0.66	0.00
fastpso-omp	22.01	0.72	0.00
fastpso	23.62	0.71	0.00

acm In-Cooperation sighpc

FLOPs and Memory Bandwidth

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

Table 3: FLOPs and memory bandwidth

metrics	dram read throughput (GB/s)	GFLOPS
gpu-pso	61.83	1.19
hgpu-pso	57.41	0.97
fastpso	106.94	8.68

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!