Parallel Enhanced Whale Optimization Algorithm

Bevan Stanely
Department of Microbiology and Cell Biology
Indian Institute of Science, Bangalore, India
bevanstanely@iisc.ac.in

***Abstract*—One of the current state-of-the-art nature-inspired meta-heuristic optimization algorithms is the enhanced Whale Optimization Algorithm (WOAmM). It integrates a modified mutualism phase from Symbiotic Organisms Search (SOS) into the original Whale Optimization Algorithm (WOA). As a result, it efficiently addresses the premature convergence seen in WOA. We propose a parallel implementation of the same under the CUDA GPU architecture and demonstrate the speedups achieved compared to the sequential algorithm.**

I. Introduction

Meta-heuristic algorithms work by spawning a population of agents and engaging them in exploration and exploitation phases. The exploratory phase helps to explore the search space extensively, whereas the exploitatory step refines promising solutions from the exploratory stage. Two primary challenges for parallel meta-heuristic algorithms are generating thread-safe random numbers and resolving data dependencies among populations. Additionally, we fixed the population size to that of warp size and resorted to using only the registers' local memory for improved cache efficiency, allowing us to employ the fast warp primitives for intra-warp communication and evade the slow shared memory altogether.

II. Related Work

Meta-heuristic algorithms are iterative by nature, and swarm-based methods, in particular, preserve search space information over subsequent iterations and involve fewer operators compared to evolutionary approaches. Mirjalili and Lewis [1] described a swarm-based meta-heuristic optimization algorithm called Whale Optimization Algorithm (WOA). It is inspired by the bubble-net hunting strategy of humpback whales. A hybrid of WOA with a modified mutualism phase from Symbiotic Organisms Search (SOS) came out designated enhanced Whale Optimization Algorithm (WOAmM) [2]. WOAmM addresses the drawbacks of WOA, namely, low exploration ability, slow convergence speed, and being trapped into a local solution easily.

III. A Parallel Enhanced Whale Optimization Algorithm

WOAmM has two components: the original WOA and the Mutualism phase of the symbiotic organism search (mSOS) algorithm. Sequential execution of WOAmM allowed data-dependent stochasticity for both components. The probability that a randomly selected individual has already been updated increases from zero for the first individual to one for the last individual in serial WOAmM. A parallel implementation of the algorithm will have to forsake this advantage. But as would be later found, this is inconsequential for the optimization if we can assure the quality of random numbers.

A. Modified Mutualism phase of the symbiotic organism search (mSOS) algorithm

Mutualism is a two-way relationship between two organisms, where both of them benefitted from the interaction. A typical example would be honey bees and flowers. Mathematically we can express it as,

$$P_i^{(k+1)} = P_i^{(k)} + rnd \cdot (P_s - MV \cdot BF1) \tag{1}$$
$$P_r^{(k+1)} = P_n^{(k)} + rnd \cdot (P_s - MV \cdot BF2) \tag{2}$$

where P_i is the i^{th} individual[1], and P_r is an organism randomly[2] selected to interact with P_i and P_s is the individual with the best fitness among P_r & P_s. The remaining variables are as follows.

$$P_s = minfitness(P_n, P_m) \tag{3}$$
$$P_r = maxfitness(P_n, P_m) \tag{4}$$
$$MV = \frac{P_i + P_s}{2} \tag{5}$$
$$BF = round(1 + rnd) \tag{6}$$

B. Whale Optimization Algorithm (WOA)

WOA mimics the bubble-net hunting strategy of humpback whales and includes three phases, searching the prey, encircling the target, and spiral bubble-net feeding maneuver. WOA uses these three phases to balance between exploration and exploitation.

1) Searching the prey: Whales randomly search the target, depending on its current location. This behavior of the humpback whale is used in the algorithm to amplify its exploration capability.

2) Encircling the prey: During this phase, we assume the current best solution to be close to the optimal solution. Then, the whales update their positions near the best solutions.

We can mathematically write 1 and 2 as:

$$P^{(k+1)} = P_{random/best}^{(k)} - A \cdot \overline{D} \tag{7}$$

[1] P_* corresponds to a position vector

[2] We use rnd to denote a random number with a uniform distribution between $[0, 1]$. All references to random numbers in this paper corresponds to rnd

TABLE I
PARAMETERS OF WOA

a_1: A number decreasing linearly from 2 to 0
a_2: A number decreasing linearly from -1 to -2
$b = 1$
$l = (a_2 - 1)rnd + 1$

The vectors A, C, and $\overline{D}$ are calculated as follows:

$$\overline{D} = |C \cdot P^{(k)}_{random/best} - P^{(k)}| \tag{8}$$

$$A = 2a_1 \times rnd - a_1 \tag{9}$$

$$C = 2 \times rnd \tag{10}$$

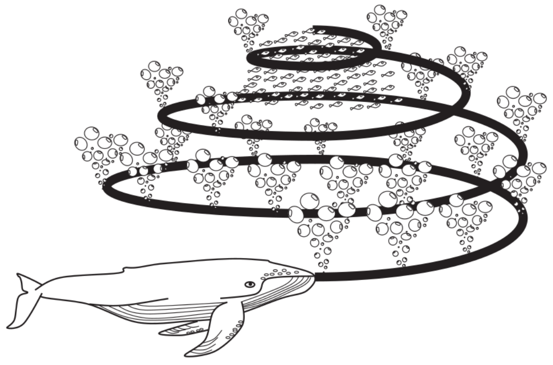

Fig. 1. Bubble-net feeding behavior of humpback whales.

3) Bubble-net attacking strategy: Humpback whales follow a surface feeding behavior known as bubble-net attack, where the whales move on a helix-shaped path. We implement this strategy in WOA as:

$$P^{(k+1)} = D^* \cdot e^{bl} \cdot \cos 2\pi l + P^{(k)}_{best} \tag{11}$$

$$D^* = P^{(k)}_{best} - P^{(k)} \tag{12}$$

C. Parallelization

For parallelization, we employed CUDA under GPU. We modeled the individuals P_i of the population as GPU threads and stored the population data and its fitness in the thread's local memory (register). To capitalize on the wrap level communication offered, we fixed the population size n to the warp size 32, allowing communication of population data without involving the slow shared memory. A single kernel executed the complete WOAmM algorithm within a thread warp.

1) Intra-warp Communication: Threads need to communicate to find P_{best} and to find P_{random}. We have resorted to an all-to-all butterfly reduction with __shfl_xor_sync() warp primitive to find P_{best}, which returns the cost and thread id of the best. Similarly, for P_{random}, we calculate the thread id first. Later for both cases, we use __shfl_sync() warp primitive to fetch the individual P_{best} or P_{random}.

Algorithm 1: Parallel WOAmM for GPU.

```
# Start kernel from here for single warp;
for each thread do
    Initialize the population $P_i (i = 1, 2 \ldots n)$;
    Initialize $k = 0$ & $\max_{iter}$;
    while $k < \max_{iter}$ do
        Fetch population = $[P_m, P_n]$ from other threads
          randomly, where $P_i \neq P_s \neq P_r$;
        Calculate the new value of $P_i$ & $P_r$ using
          eq.1 and eq.2;
        Calculate the fitness of $P_i^{k+1}$ and $P_r^{k+1}$;
        Update the value of $P_i$ and $P_r$if the new fitness
          value is minimum;
        Find $P_{best}$;
        #Procedure WOA starts from here;
        for every search-agent do
            Update $A, C, l$ & $\beta$;
            if $\beta < 0.5$ then
                if $|A| \geq 1$ then
                    Select a random individual $P_{random}$
                      Update the position of $P_i$ by eq 7
                      for $P_{random}$
                else
                    Update the position of $P_i$ by eq 7
                      for $P_{best}$
                end
            else
                Update the position of current
                  search-agent by eq.8
            end
            Check boundary conditions;
        end
        $k = k + 1$
    end
end
Return $P_{best}$
```

2) Avoiding Warp Divergence: mSOS and WOA have conditionals that can lead to warp divergence. We have used pointer arrays and binary boolean values to prevent it.

3) Random Numbers: WOAmM is a stochastic algorithm and hence needs random numbers for execution. The state size of the Mersenne Twister 19937 generator (64 bit) that we use in a sequential program is 19937 bits or 2.5 kilobytes. In parallel execution for thread safety, we need a state for each thread. Hence the pseudo-random number generators like mt19937_64 from the CPU world are not ideal; the local memory for GPUs will become a bottleneck. Therefore we have used two RNGs MRG32k3a, and Philox_4x32_10, with device API, which is much leaner and a robust RNG from the Mersenne Twister family MTGP32 with host API. We initialized the device RNGs within the CUDA kernel and used

TABLE II
FIXED DIMENSION UNIMODAL AND MULTIMODAL FUNCTIONS.

ID	Function	Equation	Search Space	Dimension D	Optimum Value
		Unimodal			
F1	Sphere	$F(x) = \sum_{k=1}^{D} x_k^2$	$[-100, 100]$	30	0
F2	Rosenbrock	$F(x) = \sum_{k=1}^{D-1} [100(x_{k-1} - x_k^2)^2 + (x_k - 1)^2$	$[-30, 30]$	30	0
		Multimodal			
F3	Rastrigin	$F(x) = \sum_{k=1}^{D} [x_k^2 - 10\cos(2\Pi x) + 10]$	$[-5.12, 5.12]$	30	0
F4	Griewank	$F(x) = \frac{1}{4000}\sum_{k=1}^{D} x_k^2 - \prod_{k=1}^{D} \cos(\frac{x_k}{\sqrt{k}}) + 1$	$[-600, 600]$	30	0

TABLE III
PARAMETERS VARIED IN THE EXPERIMENTS

Parameter	Set

RNG	Host: {MTGP32}
	Device: {MRG32k3a,Philox_4x32_10}

# Iterations	{30,100,300}
# Blocks	{1,2,4,6}

the sequence option to distribute states for individual threads. We initialized the host RNG from the CPU.

IV. EXPERIMENTS AND RESULTS

A. Experiment Setup

Parallel WOAmM was evaluated by applying it to four well-known multi-variate functions selected from the original paper. The sample included two unimodal (F1 and F2) and two multimodal (F3 and F4) functions with dimensions 30. Further information regarding the functions is available in Table II.

For an optimization algorithm, while parallelizing, we have to focus on the accuracy of the result and execution speed. Thus, we face two main challenges to achieve our goal. First, we have to make up for the loss of data dependency and then address the lower quality of GPU RNGs. Hence for the experiments, we have varied three parameters. They are as follows:

1) Types of RNGs.
2) Number of iterations of WOAmM
3) Number of blocks that run simultaneously in a single experiment

The exact details of the parameters are available in Table III.

We implemented Parallel WOAmM, compared optimization efficiency with the sequential algorithm, and then analyzed the speed-up to the sequential algorithm with 30 iterations of WOAmM across all parameters. All the experiments were performed with the GPU node of CDS Turing Cluster, with NVIDIA Tesla K40M GPU and Xeon E5 2620 V2 CPU with 24 GB memory.

To effectively compare device RNGs with host RNG for the Parallel WOAmM, we have separated RNG state initialization from the actual execution of Parallel WOAmM. Hence the memory allocation and RNG initialization times are not included in the total runtime of Parallel WOAmM. However, the total runtime includes computation costs with memory copy call from device to host.

The efficiency of Parallel WOAmM has been measured by comparing it to the sequential counterpart with 30 iterations [3]. For Parallel WOAmM, we collected data for executions with four functions by varying parameters. Thus, each execution of the algorithm for a particular function was carried out 50 times.

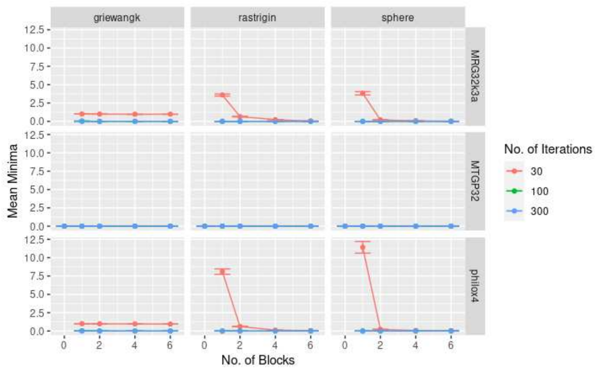

Fig. 2. The mean value of evaluated function with error bars. Note sequential program(MT64 RNG) is included under MTGP32 for comparison as block 0.

B. Results

Fig 2 summarizes the optimization efficiency of the Parallel WOAmM compared against sequential execution for variable parameters. [4] Our reservations against the GPU RNGs were not unwarranted. From Fig 2 and 3, we see that the CPU RNG gives the most optimal value but with considerably low speedups(in the range $(0.5, 5.5)$). MRG32k3a gave much better optimization among the two GPU RNGs, pointing to a much better bargain with randomness quality and RNG state size.

As expected, speed-ups remain relatively constant when increasing the number of blocks. Further, although it depends

[3]Thirty iterations alone give good optimization for the sequential algorithm.

[4]We included F2 in our experiments because it performed worse with sequential WOAmM; it was a reasonable control. But we had to exclude it from our analysis because it gave results three orders of magnitude worse than the sequential optimization with single block Parallel WOAmM and Philox_4x32_10 RNG.

TABLE IV
OPTIMIZATION AND SPEED-UPS WITH PARALLEL WOAmM

RNG	Blocks	Iteration	F1 Sphere			F3 Rastrigin			F4 Griewank		
			Mean	Best	Speed-Up	Mean	Best	Speed-Up	Mean	Best	Speed-Up
MRG32k3a	1	30	3.8E+00	2.5E-03	27.9	3.6E+00	3.6E-03	39.8	1.0E+00	9.9E-01	57.0
		100	1.5E-10	8.2E-20	8.7	4.2E-05	0.0E+00	12.2	3.1E-02	0.0E+00	17.6
		300	6.2E-43	0.0E+00	2.9	6.1E-07	0.0E+00	4.1	2.4E-09	0.0E+00	5.9
	2	30	2.0E-01	4.2E-04	27.9	6.5E-01	2.7E-04	39.9	9.9E-01	7.9E-01	57.1
		100	5.8E-12	8.4E-19	8.6	0.0E+00	0.0E+00	12.2	1.7E-04	6.0E-08	17.6
		300	0.0E+00	0.0E+00	2.9	0.0E+00	0.0E+00	4.1	0.0E+00	0.0E+00	5.9
	4	30	7.4E-02	4.2E-04	27.6	2.3E-01	2.4E-04	39.4	9.7E-01	6.0E-01	56.5
		100	3.3E-14	3.2E-20	8.6	6.1E-07	0.0E+00	12.1	1.0E-06	0.0E+00	17.4
		300	0.0E+00	0.0E+00	2.9	0.0E+00	0.0E+00	4.0	0.0E+00	0.0E+00	5.8
	6	30	1.6E-02	8.6E-05	27.6	3.9E-02	4.6E-04	39.4	9.8E-01	7.8E-01	56.4
		100	2.3E-15	2.7E-19	8.6	0.0E+00	0.0E+00	12.0	4.0E-07	0.0E+00	17.4
		300	0.0E+00	0.0E+00	2.9	0.0E+00	0.0E+00	4.0	0.0E+00	0.0E+00	5.8
MTGP32	1	30	9.9E-28	1.8E-31	2.5	0.0E+00	0.0E+00	4.5	0.0E+00	0.0E+00	5.4
		100	3.0E-35	3.0E-43	0.8	0.0E+00	0.0E+00	1.4	0.0E+00	0.0E+00	1.7
		300	0.0E+00	0.0E+00	0.3	0.0E+00	0.0E+00	0.5	0.0E+00	0.0E+00	0.6
	2	30	1.9E-20	6.1E-28	2.4	0.0E+00	0.0E+00	4.4	0.0E+00	0.0E+00	5.4
		100	0.0E+00	0.0E+00	0.8	0.0E+00	0.0E+00	1.4	0.0E+00	0.0E+00	1.7
		300	0.0E+00	0.0E+00	0.3	0.0E+00	0.0E+00	0.5	0.0E+00	0.0E+00	0.6
	4	30	2.5E-17	3.8E-29	2.5	3.7E-06	0.0E+00	4.5	0.0E+00	0.0E+00	5.4
		100	1.7E-41	0.0E+00	0.8	0.0E+00	0.0E+00	1.4	0.0E+00	0.0E+00	1.7
		300	0.0E+00	0.0E+00	0.3	0.0E+00	0.0E+00	0.5	0.0E+00	0.0E+00	0.6
	6	30	2.4E-16	1.1E-21	2.4	0.0E+00	0.0E+00	4.3	0.0E+00	0.0E+00	5.2
		100	0.0E+00	0.0E+00	0.7	0.0E+00	0.0E+00	1.3	0.0E+00	0.0E+00	1.6
		300	0.0E+00	0.0E+00	0.2	0.0E+00	0.0E+00	0.4	0.0E+00	0.0E+00	0.5
Philox_4x32_10	1	30	1.1E+01	1.4E-03	22.4	8.1E+00	4.0E-04	35.8	1.0E+00	9.7E-01	45.3
		100	1.5E-10	4.6E-19	6.9	2.1E-05	0.0E+00	11.0	2.4E-02	6.0E-08	13.9
		300	2.3E-40	0.0E+00	2.3	2.4E-06	0.0E+00	3.7	3.6E-09	0.0E+00	4.7
	2	30	2.3E-01	8.4E-04	22.3	6.1E-01	8.9E-04	35.8	9.8E-01	6.8E-01	45.3
		100	9.7E-13	5.8E-20	6.9	0.0E+00	0.0E+00	11.0	1.8E-03	0.0E+00	13.9
		300	0.0E+00	0.0E+00	2.3	0.0E+00	0.0E+00	3.7	0.0E+00	0.0E+00	4.7
	4	30	6.3E-02	1.9E-04	22.1	1.2E-01	1.0E-03	35.4	9.7E-01	2.8E-01	44.8
		100	1.5E-14	1.6E-19	6.8	0.0E+00	0.0E+00	10.9	5.1E-07	0.0E+00	13.7
		300	0.0E+00	0.0E+00	2.3	0.0E+00	0.0E+00	3.7	0.0E+00	0.0E+00	4.6
	6	30	4.1E-02	7.4E-04	22.2	4.9E-02	1.4E-03	35.5	9.5E-01	4.6E-01	44.8
		100	4.5E-15	1.3E-19	6.8	0.0E+00	0.0E+00	10.9	3.8E-07	0.0E+00	13.7
		300	0.0E+00	0.0E+00	2.3	0.0E+00	0.0E+00	3.7	0.0E+00	0.0E+00	4.6

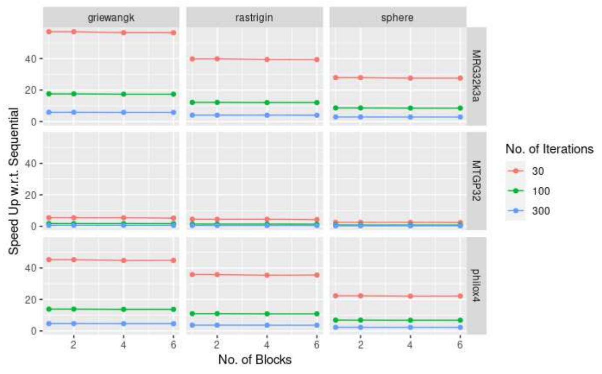

Fig. 3. Speed-ups of parallel implementations w.r.t. sequential program with 30 iterations and MT64 RNG

on the specific function we are evaluating, we find MRG32k3a with 100 iterations and four blocks of GPU threads to give good optimization and speedup.

For future works on Parallel WOAmM, we wish to experiment with more GPU RNGs with a good balance between randomness quality and size. We find a variant of WOA, CWOA [3], which uses chaotic maps instead of random numbers lucrative. It would be interesting to see if parallel execution-friendly chaotic maps exist in literature.

REFERENCES

[1] S. Mirjalili and A. Lewis, "The whale optimization algorithm," *Advances in Engineering Software*, vol. 95, pp. 51–67, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0965997816300163

[2] S. Chakraborty, A. Kumar Saha, S. Sharma, S. Mirjalili, and R. Chakraborty, "A novel enhanced whale optimization algorithm for global optimization," *Computers and Industrial Engineering*, vol. 153, p. 107086, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835220307567

[3] G. Kaur and S. Arora, "Chaotic whale optimization algorithm," *Journal of Computational Design and Engineering*, vol. 5, no. 3, pp. 275–284, 01 2018. [Online]. Available: https://doi.org/10.1016/j.jcde.2017.12.006