Parallel Implementation of Metahueristic Algorithm

Rohit Khetan(20bcs114), Yashu Mittal(20bcs139), Navneet Sen(20bds037), Shubham Sharma(20bds051)
Project Guide- Dr. Pramod Yelmewad
Department of Computer Science and Engineering
Indian Institute of Information Technology Dharwad

***Abstract*—The improved Whale Optimization Algorithm (WOAmM) is one of the most advanced nature-inspired meta-heuristic optimization algorithms available today. It incorporates into the original Whale Optimization Algorithm (WOA) a modified mutualism phase from Symbiotic Organisms Search (SOS). Consequently, it effectively tackles the early convergence observed in WOA. We suggest using the CUDA GPU architecture for a parallel version of the same, and we show the speedups over the sequential technique.**

I. Introduction

In order for meta-heuristic algorithms to function, a population of agents must be created and led through phases of exploration and exploitation. While the exploitatory stage refines potential ideas from the exploratory stage, the exploratory phase aids in the comprehensive exploration of the search space. Resolving data interdependence among populations and producing thread-safe random numbers are the two main problems that parallel meta-heuristic algorithms must overcome. In order to increase cache efficiency, we also set the population size to match the size of the warp and switched to using only the local memory of the registers. This allowed us to use the fast warp primitives for intra-warp communication and completely avoid the slow shared memory.

II. Related Work

Since meta-heuristic algorithms are iterative by nature, swarm-based techniques in particular require fewer operators and conserve search space information over time than evolutionary alternatives. Whale Optimization Algorithm (WOA) is a swarm-based meta-heuristic optimization algorithm that was described by Mirjalili and Lewis [1]. It is modeled after the humpback whale's bubble-net hunting technique. Enhanced Whale Optimization Algorithm (WOAmM) is a hybrid algorithm that combines the best features of WOA with a modified mutualism phase from Symbiotic Organisms Search (SOS) [2]. The shortcomings of WOA—low exploration capabilities, sluggish convergence speed, and ease of becoming trapped in a local solution—are addressed by WOAmM.

III. An Algorithm for Parallel Enhanced Whale Optimization

WOAmM is composed of the original WOA and the Mutualism stage of the symbiotic organism search (mSOS) algorithm. The sequential execution of WOAmM enabled data-dependent stochasticity for both components. The likelihood that a randomly selected person in serial WOAmM has been updated before increases from zero for the first person to one for the last. This gain will have to be forfeited by the parallel implementation of the method. But as we would learn later, this is irrelevant to the optimization if we can ensure the quality of random numbers.

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

Mutualism is a two-way relationship in which the interactions benefit both of the organisms. Honey bees with flowers would be a common example. In mathematical terms, we can write 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_r is an organism chosen at random, and P_i is the i^{th} individual[1].[2] chosen to interact with P_i, and P_s is the most fit individual among P_r & P_s. These are the variables that are still in play.

$$P_s = minfitness(P_n, P_m) \tag{3}$$

$$P_r = maxfitness(P_n, P_m) \tag{4}$$

$$MV = \frac{P_i + P_s}{2} \tag{5}$$

$$BF = round(1 + rnd) \tag{6}$$

B. Whale Optimization Algorithm (WOA)

WOA comprises three phases: scanning for prey, encircling the target, and spiral bubble-net feeding maneuver. It is modeled after the bubble-net hunting tactic used by humpback whales. These three stages are used by WOA to strike a balance between exploration and exploitation.

1) Searching the prey: Depending on where the target is at any given time, whales randomly hunt it. The program makes use of this humpback whale behavior to increase its capacity for exploration.

2) Encircling the prey: In this stage, we consider the best option that exists right now to be fairly near to the ideal answer. The whales then realign themselves in proximity to the optimal answers.

We can mathematically write 1 and 2 as:

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

[1] P_* corresponds to a position vector

[2] To represent a random number with a uniform distribution between $[0, 1]$, we use rnd. In this study, all references to random numbers relate 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:

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

$$A = 2a_1 \times rnd - a_1 \quad (9)$$

$$C = 2 \times rnd \quad (10)$$

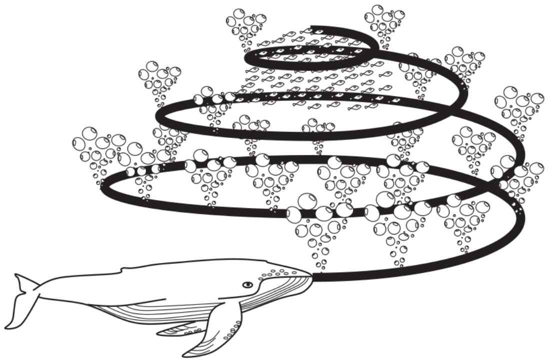

Fig. 1. Bubble-net feeding behavior of humpback whales.

3) Bubble-net attacking strategy: The bubble-net assault surface feeding activity of humpback whales involves the whales traveling in a helix-shaped pattern. In WOA, we apply this tactic as:

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

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

C. Parallelization

We used CUDA on the GPU for parallelization (Algorithm 1). In order to keep the population data and its fitness in the thread's local memory (register), we modeled the individuals P_i of the population as GPU threads. We fixed the population size n to the warp size 32 in order to take advantage of the wrap level communication provided, which enables population data transmission without utilizing the sluggish shared memory. The entire WOAmM algorithm was run by a single kernel in a thread warp.

1) Intra-warp Communication: In order to determine P_{best} and P_{random}, threads must interact. The __shfl_xor_sync () warp primitive has been used in an all-to-all butterfly reduction in order to get P_{best}, which yields the cost and thread id of the best. In a similar manner, we determine the thread id first for P_{random}. We then utilize the __shfl_sync () warp primitive for both scenarios to retrieve each unique P_{best} or P_{random}.

Algorithm 1: Parallel WOAmM for GPU.

```
1  # Start kernel from here for single warp;
2  for each thread do
3      Initialize the population $P_i (i = 1, 2 \dots n)$;
4      Initialize $k = 0$ & $\max_{iter}$;
5      while $k < \max_{iter}$ do
6          Fetch population = $[P_m, P_n]$ from other threads
             randomly, where $P_i \neq P_s \neq P_r$;
7          Calculate the new value of $P_i$ & $P_r$ using
             eq.1 and eq.2;
8          Calculate the fitness of $P_i^{k+1}$ and $P_r^{k+1}$;
9          Update the value of $P_i$ and $P_r$if the new fitness
             value is minimum;
10         Find $P_{best}$;
11         #Procedure WOA starts from here;
12         for every search-agent do
13             Update $A, C, l$ & $\beta$;
14             if $\beta < 0.5$ then
15                 if $|A| \geq 1$ then
16                     Select a random individual $P_{random}$
                         Update the position of $P_i$ by eq 7
                         for $P_{random}$
17                 else
18                     Update the position of $P_i$ by eq 7
                         for $P_{best}$
19                 end
20             else
21                 Update the position of current
                     search-agent by eq.8
22             end
23             Check boundary conditions;
24         end
25         $k = k + 1$
26     end
27 end
28 Return $P_{best}$
```

2) Avoiding Warp Divergence: Conditionals in mSOS and WOA may cause warp divergence. To stop it, we've employed binary boolean values and pointer arrays.

3) Random Numbers: Random numbers are required for the execution of WOAmM because it is a stochastic algorithm. A sequential program we employ has a state size of 19937 bits, or 2.5 kilobytes, for the Mersenne Twister 19937 generator (64 bit). Every thread needs a state when running in parallel for thread safety. The local memory of GPUs will thus become a bottleneck, making CPU-based pseudo-random number generators like mt19937_64 less than optimal. Because of this, we have employed two RNGs with device API: MRG32k3a and Philox_4x32_10. These are far leaner than the robust RNG from the Mersenne Twister family, MTGP32, with host API. We set up the device RNGs in the CUDA kernel and distributed the states for each thread using the sequence option. We used the CPU to initialize the host RNG.

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}

IV. EXPERIMENTS AND RESULTS

A. Experiment Setup

Applying Parallel WOAmM to four popular multi-variate functions taken from the original research allowed for its evaluation. Two multimodal (F3 and F4) and two unimodal (F1 and F2) functions with dimensions of 30 were included in the sample. Additional details about the functions can be found in Table II.

When parallelizing an optimization algorithm, we must prioritize both execution speed and result correctness. To reach our objective, we must overcome these two major obstacles. Prior to addressing the reduced quality of GPU RNGs, we need to compensate for the loss of data reliance. We have therefore changed three parameters for the experiments. They are listed below:

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.

After implementing Parallel WOAmM, we evaluated the speed-up to the Sequential algorithm using 30 iterations of WOAmM across all parameters and compared the optimization efficiency with the Sequential algorithm. The CDS Turing Cluster GPU node, equipped with an NVIDIA Tesla K40M GPU and a Xeon E5 2620 V2 CPU with 24 GB of RAM, was used for all of the studies.

We have isolated RNG state initialization from the actual Parallel WOAmM execution in order to compare device RNGs with host RNG for the Parallel WOAmM effectively. Therefore, the total runtime of Parallel WOAmM does not include the memory allocation and RNG initialization times. All runtime, however, includes memory copy calls from the device to the host and computation expenses.

The comparative analysis of Parallel WOAmM's efficiency with its sequential version using 30 iterations yielded good optimization results for the sequential algorithm. We gathered data for four function runs using different parameters for Parallel WOAmM. As a result, the algorithm for a given function was run 50 times for each execution.

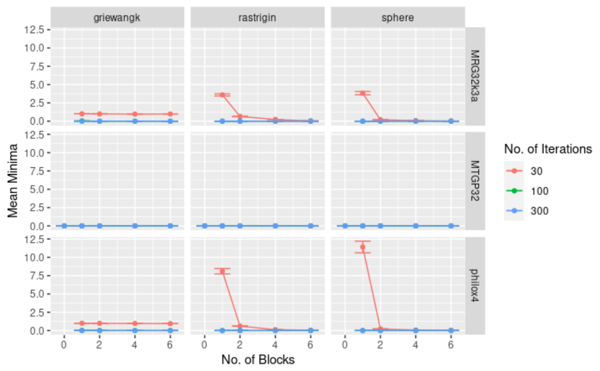

Fig. 2. The mean value of evaluated function with error bars. Note sequential program(MT64 RNG) is included under MTGP32 for comparison as block 0.

B. Results

The optimization performance of the Parallel WOAmM in comparison to sequential execution for variable parameters is summarized in Fig. 2. **note**Since F2 fared worse with successive WOAmM, we decided to include it in our tests as an acceptable control. However, it produced results three orders of magnitude poorer than the sequential optimization using single block Parallel WOAmM and Philox_4x32_10 RNG,thus we had to eliminate it from our investigation. Our misgivings about the GPU RNGs were not without merit. Figures 2 and 3 show that, although providing the most ideal value, the CPU RNG only significantly increases speed (in the range $(0.5, 5.5)$). Out of the two GPU RNGs, MRG32k3a provided significantly better optimization, suggesting a better deal in terms of randomness quality and RNG state size.

As anticipated, speedups don't change much as the number of blocks does. Furthermore, we find that MRG32k3a with 100 iterations and four blocks of GPU threads provides

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

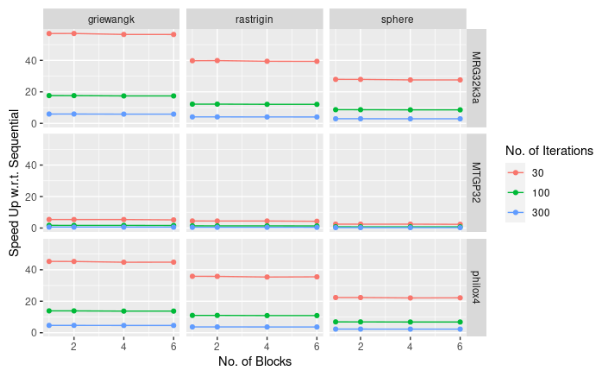

Fig. 3. Speed-ups of parallel implementations w.r.t. sequential program with 30 iterations and MT64 RNG

good optimization and speedup, however this depends on the particular function we are evaluating.

We would like to explore with new GPU RNGs that strike a decent compromise between size and randomness quality for our next work on Parallel WOAmM. We discover a profitable variation of WOA called CWOA [3], which substitutes chaotic maps for random integers. If there are any parallel execution-friendly chaotic maps in literature, that would be nice to see.

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