Parallel Mematic Algorithm Implementation on CUDA
Umut Cinar, Alptekin Temizel
Graduate School of Informatics, Middle East Technical University, Ankara, Turkey

ABSTRACT
In this paper, a parallel memetic algorithm implementation for CUDA platform is described. The conventional genetic operators are adapted to the GPU considering the GPU architecture. In this population based optimization technique, there are one more islands and each island consists of constant number of individuals. Each CUDA thread is responsible for evolution of one individual, and islands are mapped to CUDA blocks to benefit from the shared memory. The results show up to 38x speedup compared to the CPU implementation.

KERNEL PSEUDO CODE

Initialize shared memory buffers

WHILE termination criterion is not met; DO
begin local search
    tournament selection
    IF thread_id < warp_size
        offsprings = crossover(parents)
        offsprings = mutation(offsprings)
        replace offsprings with worst_individuals
    ELSE IF thread_id < warp_size*2 AND
       (MIGRATION_INTERVAL is true)
       migrate(best_individuals)
    END IF
END WHILE

- Threads within a block continues in parallel until the variation and migration operators take place.
- At this point, two warp of threads executes crossover/mutation and migration operations. First warp is assigned to crossover/mutation while the second warp is assigned to migration (when it is migration interval).
- Hence, this part of the implementation is completely unrolled by using a warp for each operation. This approach decreases the cost of warp divergence significantly.

INFLUENCE OF GENOME SIZE ON EXECUTION TIME

Test function: Schwefel

Function: $f_s(x) = \sum_{i=1}^{n} x_i^2$
Global Minimum: $s^* = (0, 0, \ldots, 0)$
Properties: Not separable

In this paper, a parallel memetic algorithm implementation for CUDA platform is described. The conventional genetic operators are adapted to the GPU considering the GPU architecture. In this population based optimization technique, there are one more islands and each island consists of constant number of individuals. Each CUDA thread is responsible for evolution of one individual, and islands are mapped to CUDA blocks to benefit from the shared memory. The results show up to 38x speedup compared to the CPU implementation.

GENERAL DESIGN

Experiments have been investigated using several benchmarking functions including Rosenbrock, Rastrigin, Schwefel and Griewangk, and various parameters.

<table>
<thead>
<tr>
<th>Function</th>
<th>Formula</th>
<th>Global Minimum</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rosenbrock</td>
<td>$f_{rb}(x) = \sum_{i=1}^{n} 100(x_{i+1} - x_i^2)^2 + (x_1 - 1)^2$</td>
<td>$s^* = (1, 1, \ldots, 1)$</td>
<td>$f_{rb}(s^*) = 0$</td>
</tr>
<tr>
<td>Rastrigin</td>
<td>$f_{ra}(x) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)]$</td>
<td>$s^* = (0, 0, \ldots, 0)$</td>
<td>$f_{ra}(s^*) = 0$</td>
</tr>
<tr>
<td>Griewangk</td>
<td>$f_{gw}(x) = 1 + \sum_{i=1}^{n} \left( \frac{x_i}{\sqrt{n}} \right)^2 - \prod_{i=1}^{n} \cos \left( \frac{x_i}{\sqrt{n}} \right)$</td>
<td>$s^* = (0, 0, \ldots, 0)$</td>
<td>$f_{gw}(s^*) = 0$</td>
</tr>
<tr>
<td>Schwefel</td>
<td>$f_{sw}(x) = \sum_{i=1}^{n} x_i^2$</td>
<td>$s^* = (0, 0, \ldots, 0)$</td>
<td>$f_{sw}(s^*) = 0$</td>
</tr>
</tbody>
</table>

EXPERIMENTS

Speedup has been investigated using several benchmarking functions including Rosenbrock, Rastrigin, Schwefel and Griewangk, and various parameters.

CONCLUSION

- The experiments show that the results are heavily dependent on the fitness function type. In a memetic algorithm, the most computationally complex routine is the fitness function, and this is reflected in the dependency of performance on fitness function type.
- The maximum speedups achieved with Griewangk function as x38.

REFERENCES