CUDA Implementation of Particle Swarm Optimization

Muhammad Muzammil, Mohammad Shanur Rahman

In partial fullfillment of course - High End Simulation and Practices, Masters in AI, Friedrich-Alexander-Universität Erlangen-Nürnberg

September 11, 2023

1 Introduction

1.1 Background

Particle Swarm Optimization (PSO) is a popular optimization algorithm inspired by the social behavior of birds and fish. It was first introduced by Kennedy and Eberhart in 1995. PSO has gained wide recognition due to its simplicity and effectiveness in solving a variety of optimization problems.

1.2 Particle Swarm Optimization (PSO)

Particle Swarm Optimization is a population-based optimization algorithm that simulates the behavior of a swarm of particles. Each particle represents a potential solution to the optimization problem and adjusts its position in the solution space based on its own fitness value and the fitness value of its neighbors. PSO iteratively updates the particles' positions until a stopping criterion is met, ideally converging to the optimal solution.

1.3 Parallel Computing with CUDA

With the growing need for faster and scalable algorithms, it is important to rethink known algorithms in the context of parallelization. We use CUDA, which is a parallel computing platform and API developed by NVIDIA to parallelise PSO for GPUs.

2 Objectives and Goals

The primary objective of this project is to implement the Particle Swarm Optimization algorithm using CUDA to use GPU parallelism. The specific goals are as follows:

- Develop serial CPU-based code for PSO algorithm.
- Develop the CUDA-based PSO algorithm.
- Analyse multiple reduction kernels for finding the particle with minimum fitness value.
- Compare the performance of the CUDA implementation with CPU-based PSO, and differen GPU based kernels.
- Analyze the speedups achieved.

3 Methodology

3.1 Parallelization Techniques

In our CUDA implementation of PSO, we parallelize the particle update process. Each particle's position and velocity updates are computed independently, making it suitable for execution on a GPU with a large number of cores. We use two kernels [Steps 3 - 5] in Algorithm 1, the first one updates the position and velocities as per the velocity update rule. After calculating the new positions, the fitness of each particle is calculated and returned in an array. The second reduction kernel [Step 6] in Algorithm 1 finds the index with minimum fitness value. And these two kernels are iteratively executed until a minima is reached.

Algorithm 1 Particle Swarm Optimization - CUDA

Require: Number of particles N, Number of iterations T
Ensure: Best solution found P_{best}

1: Initialize particle positions and velocities
2: **for** $t \leftarrow 1$ to T **do**
3: Update particle velocities using its current velocity, global best solution and local best solution as per velocity update rule .
4: Update particle positions.
5: Calculate fitness value for each particle and store in an array $fitness[]$.
6: Find the index i_{min} with minimum fitness value in $fitness[]$.
7: Update local best and global best solution P_{best} with the position of particle i_{min}
8: **end for**
9: **return** P_{best}

Since, our step 6 in Algorithm 1 requires finding the minimum of all fitness values, we use tree based reduction algorithms which are extensively studied.

In kernel 2, we iteratively increase the stride in multiples of 2 to find minimum of each thread and corresponding strided thread. The resulting algorithm runs in O(log N) complexity, instead of usual O(N) in case of a sequential traversal in a CPU. However, the divergence among threads is quite evident. Since, at a single time only strided threads are active, remaining threads are idle. To cope this, we change our access pattern in kernel 3 such that each thread reduces its own value and the value after blockDim.x. This causes consecutive threads to be busy and hence less divergence, leading to more free cores, which can be used for further re-scheduling.

Algorithm 2 CUDA Kernel for Finding Minimum Value in an Array with divergent threads

Require: `input` (device array), `output` (device array), `N` (array size)

```
tid ← threadIdx.x + blockIdx.x * blockDim.x
{Store the minimum value for this block in shared memory}
__shared__ int min_shared[256]; {Assumes max threads per block is 256}
min_shared[threadIdx.x] ← input[tid]
__syncthreads()
{Reduce the minimum values from all threads in the block to find the block minimum}
for s ← 1 to blockDim.x, s*=2 do
  if threadIdx.x % (2 * s) == 0 then
    partial_res[threadIdx.x] = MIN(partial_res[threadIdx.x], partial_res[threadIdx.x + s]);
  end if
  __syncthreads()
end for
{Write the block minimum to the output array}
if threadIdx.x == 0 then
  output[blockIdx.x] ← min_shared[0]
end if
```

4 Experimental Setup

4.1 Hardware and Software Environment

The experiments were conducted on a system equipped with an NVIDIA GeForce RTX 3080 GPU. We used the CUDA 12 APIs for GPU programming. And used unified memory as memory interface between GPU and CPU. The CPU-based PSO served as the baseline for comparison.

Algorithm 3 CUDA Kernel for Finding Minimum Value in an Array with reduced idle threads

Require: `input` (device array), `output` (device array), `N` (array size)

```
tid ← threadIdx.x + blockIdx.x * blockDim.x
stride ← blockDim.x * gridDim.x
min_val ← input[tid]
for i ← tid + stride to N step stride do
  if input[i] < min_val then
    min_val ← input[i]
  end if
end for
{Store the minimum value for this block in shared memory}
__shared__ int min_shared[256]; {Assumes max threads per block is 256}
min_shared[threadIdx.x] ← min_val
__syncthreads()
{Reduce the minimum values from all threads in the block to find the block minimum}
for s ← blockDim.x / 2 to 1 do
  if threadIdx.x < s then
    min_shared[threadIdx.x] = MIN(min_shared[threadIdx.x], min_shared[threadIdx.x + s]);
  end if
  __syncthreads()
end for
{Write the block minimum to the output array}
if threadIdx.x == 0 then
  output[blockIdx.x] ← min_shared[0]
end if
```

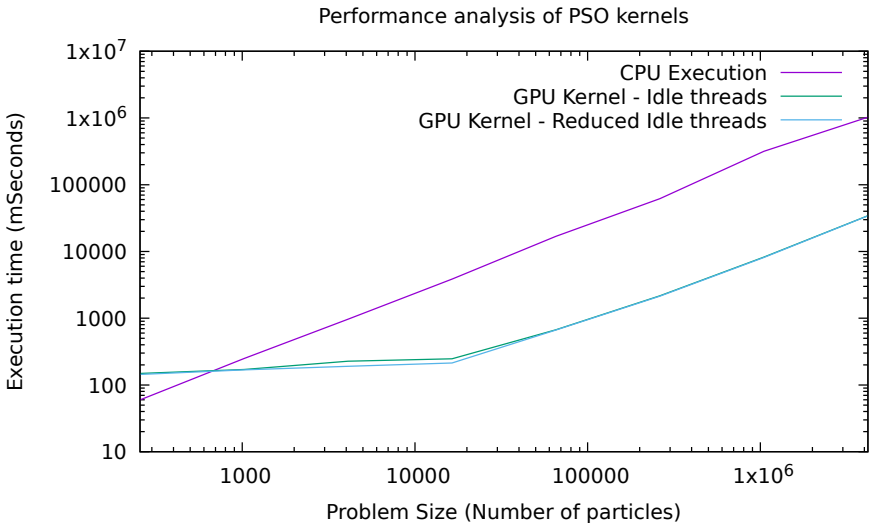

Figure 1: Performance comparison of CPU execution and two GPU kernels. The x and y are log-scaled. It can be clearly seen that at largest problem size, GPUs take execution time in orders of 1e4, while CPU takes in orders of 1e6. And the kernel with idle threads follows the one with reduced idle threads.

4.2 Experimental Parameters

We configured the PSO algorithm with the following parameters:

- Swarm size: [256 - 4,194,304] particles
- Dimensionality: 2 dimensions (Altrhough the code is generalised for N dimensions)
- Maximum iterations: 10000

5 Results and Discussion

The experimental results in Fig. 1 indicate a significant speedup in the CUDA implementation compared to the CPU-based PSO. And a visible speedup for the kernel with reduced idle threads. Since, on increasing problem size, the execution time is not saturated, this implies the program is memory bound instead of compute bound.

6 Conclusion

In this project, we successfully implemented Particle Swarm Optimization using CUDA. The CUDA version demonstrated superior performance over the CPU-

based implementation.

7 References

- Kennedy, J., Eberhart, R. (1995). Particle Swarm Optimization. Proceedings of IEEE International Conference on Neural Networks.