Accelerating Genetic Algorithms using Constant Memory in GPUs
Graphics Processing Units Project Report

Prasoon Goyal, Sahil Goel

Abstract

We propose a novel GPU variant for genetic algorithm that uses constant memory effectively to store and share the elite population across different blocks to obtain faster convergence rates. We compare our algorithm to a previous work which does not use constant memory, and show that using constant memory significantly improves the quality of solution for a fixed execution time. We show performance improvement on an instance of Traveling Salesman Problem and Ackley’s function, which is a commonly used function to test optimization algorithms.

1 Problem Definition and Related Work

In this work, we implement genetic algorithm on GPUs. The algorithm is provided with a fitness function, which takes a bit string as argument and returns the fitness score of the string. The objective is to find the bit string that is the maximizer of the fitness score. The algorithm starts with a random set of bit strings (chromosomes), picks two of these randomly in proportion to their fitness score, performs crossover and mutation over these chromosomes to generate new chromosomes.

Most existing implementations do not use constant memory for the task. [1] and [2] divide the population into groups (islands), assigning each group to one block. The population in each group evolves independently. The groups are arranged in the form of a unidirectional ring. Migration across groups is effected by passing the best $M$ chromosomes from one group to the next along the ring every few iterations using global memory. The shortcoming of this approach is that if the chromosomes in one group are much better than in all the others, then it will take a lot of iterations before they are propagated to all groups.

[3] and [4] use texture memory for the task. The chromosomes are arranged in a 2-dimensional grid, and only neighbouring chromosomes are crossed in any iteration, thereby utilizing the spatial locality of texture memory.

In this work, we study the effectiveness of constant memory for genetic algorithms. The population will be partitioned into groups, and each group will evolve in one block, as in [1] and [4]. However, instead of a ring to migrate chromosomes across groups, each block will write its best chromosomes to a designated location in the global memory, from which
it will be copied to constant memory periodically by the host. Each block, in addition to using the population in its own group, will use the “elite” population in constant memory to generate new chromosomes. Thus, the best chromosomes generated so far are shared across all blocks.

2 Design

The host function is provided with a fitness function, the maximizer of which is to be computed. The input to the fitness function is a bit string.

For ease of exposition, we assume the following values of parameters. The description can easily generalized to other values of the parameters.

Size of input to fitness function = 64 bits
Shared memory size per block = 16 KB
Constant memory = 64 KB
Number of blocks = NUM_BLOCKS
Number of threads per block = THREADS_PER_BLOCK
Number of iterations per kernel call = ITER_PER_KERNEL_CALL

Each block stores one chromosome per thread in the shared memory, and the constant memory holds 1024 chromosomes.

In global memory, memory is allocated to store (NUM_BLOCKS * THREADS_PER_BLOCK) chromosomes.

2.1 Initialization phase

The initialization phase generates THREADS_PER_BLOCK random chromosomes for each block to fill the respective shared memories. Similarly, 1024 chromosomes are randomly generated and filled in constant memory.

2.2 Score computation

After the initialization phase, fitness scores of all chromosomes in shared memory and constant memory are computed. The scores within each block are added and the sum is stored in global memory for use in evolution phase. Similarly, the scores within the constant memory are added.

2.3 Evolution phase

The evolution kernel is called repeatedly in loop by the host function passing it the sum of scores for this block (sharedMemScoreSum) and the sum of scores for chromosomes in constant memory (constantMemScoreSum). In every call, each block starts with THREADS_PER_BLOCK chromosomes in shared memory. These are evolved using the following process for ITER_PER_KERNEL_CALL iterations:
• Each thread randomly chooses one parent from shared memory or constant memory, in proportion to their fitness scores. That is, a chromosome is picked from shared memory with probability

$$\frac{\text{sharedMemScoreSum}}{\text{sharedMemScoreSum} + \text{constantMemScoreSum}/\text{NUM_BLOCKS}}$$

and from constant memory with probability

$$\frac{\text{constantMemScoreSum}/\text{NUM_BLOCKS}}{\text{sharedMemScoreSum} + \text{constantMemScoreSum}/\text{NUM_BLOCKS}}$$

This ensures that, on the average, each chromosome gets picked with probability proportional to its fitness score.

• The threads then perform crossover and mutation using these parents to generate two new chromosomes. The scores of these new chromosomes are compared to the score of the chromosome in shared memory corresponding to this thread, and the best one is written to the shared memory. This is different from [1], where they always overwrite the old chromosome with the new one irrespective of their scores.

Typical values of crossover probabilities and mutation probabilities used in previous implementations are 70% and 5%, respectively.

After completion of \texttt{ITER_PER_KERNEL_CALL} iterations, each block sorts the chromosomes in its shared memory according to their fitness scores, and writes them to the designated part of the global memory.

Finally, when the control returns to the host, it copies $1024/\text{NUM_BLOCKS}$ best chromosomes from each block to constant memory. The host also computes and stores the best chromosome found so far, and updates this value.

The score computation and evolution kernels are called repeatedly until termination condition is reached.

3 Genetic operations

3.1 Crossover

Two chromosomes \texttt{chromosome1} and \texttt{chromosome2} are crossed as follows: for each bit, a random number is generated in $[0, 1]$ and the corresponding bits in the chromosomes are exchanged if the random number is less than \texttt{CROSSOVER_PROB}.

This varies from the crossover operation in [1], in which they generate an aggregation weight ($w$) randomly in $[0, 1]$, and obtain the new chromosomes as:

new\_chromosome1 = $w \times$ \texttt{chromosome1} + $(1 - w) \times$ \texttt{chromosome2}
new\_chromosome2 = $(1 - w) \times$ \texttt{chromosome1} + $w \times$ \texttt{chromosome2}

3.2 Mutation

A single chromosome is mutated by flipping each bit with probability \texttt{MUTATION_PROB}.
4 Application: Travelling Salesman Problem

4.1 Problem definition

Travelling salesman problem (TSP) consists of finding the shortest tour in a given weighted graph, where a tour is defined as a path that visits each vertex exactly once.

We test our algorithm on the following instance of travelling salesman problem: The graph consists of 16 vertices, with the edge weights given by

\[
w(i, j) = \begin{cases} 
1 & \text{if } |i - j| = 1 \\
5 & \text{otherwise}
\end{cases}
\]

The corresponding fitness function is given by

\[
fitnessScore(path) = 75 - \text{sumWeights}(path) - 75 \times \text{numDuplicates}(path)
\]

where \text{sumWeights()} calculates the sum of weights of edges along the path and \text{numDuplicates()} returns the number of pairs of duplicate nodes in the path. 75 is the maximum possible weight for valid paths (no duplicates); hence, a single duplicate node will have lower score than any valid path. The maximum value of this function is 60, and the goal is to find the corresponding chromosome that represents that path.

We represent a path (chromosome) as a long variable, where each chunk of 4 bits represents a node index (0 to 15). Consecutive chunks represent consecutive nodes of the path.

5 Experiments and Results

In this section, we report the experimental results, comparing implementation of our proposed algorithm (denoted as WithConst), implementation of [1] (denoted as NoConst) and a sequential implementation of genetic algorithm (denoted as Sequential).

We ran all our experiments on cuda1.cims.nyu.edu, the configuration for which are as follows:

Device Number: 0
Device name: GeForce GTX TITAN Black
Total Global Memory: 2147287040
Shared Memory per block : 49152
Max threads per block : 1024
Max Threads Dimension : 1024 x 1024 x 64
Max grid size : 2147483647 x 65535 x 65535
Total Constant Memory : 65536
Memory Clock Rate (KHz): 3500000
Memory Bus Width (bits): 384
Peak Memory Bandwidth (GB/s): 336.000000

Device Number: 1
5.1 Travelling Salesman Problem

We experimented with all combinations of the following values of number of blocks, number of threads per block, and number of iterations per kernel call, and ran WithConst, NoConst and Sequential algorithms for each of these configurations:

Number of blocks = 32, 1024
Number of threads per block = 128, 1024
Number of iterations per kernel call = 100, 1000

For each configuration, the results (fitness score and execution time) were averaged over 5 executions of the program. The results are summarized in Figure 1. (For larger configurations, we could not run sequential version as it takes fairly long to run.)

It is easy to see that for each configuration, WithConst finds a good quality solution much faster than NoConst and Sequential implementations.

5.2 Other fitness functions

To study how our algorithm generalizes to other applications, we compared our implementation with [1] on Ackley’s function, which is a standard function used to test the performance of optimization algorithms. As can be seen from the figure, it is a highly non-convex 2D function with optimal value of 0 at (0,0). We represented the chromosome as long variable, and mapped the first 32 bits to a floating point value to get the x-coordinate, and the last 32 bits to get the y-coordinate.

To convert the problem from minimization to maximization, we defined our fitness score as $\text{fitnessScore}(x, y) = 25 - \text{ackley}(x, y)$, so that the optimal fitness score is 25. We recorded the number of kernel calls required by each algorithm to reach value of 24.99 or better. The results are summarized in Table 1. As can be seen from the table, WithConst finds a value of 24.99 or higher in 2 kernel calls for all configurations, demonstrating the effectiveness of using constant memory. NoConst, on the other hand, takes significantly more iterations, particularly for smaller configurations.

Finally, we also experimented with the following two synthetic fitness functions:

---

1For sequential implementation, the number of chromosomes was taken to be Number of blocks * Number of threads per block.
Figure 1: Results for experiments on TSP
Table 1: Comparison of NoConst and WithConst implementations on Ackley’s function. No. of kernel calls refer to the kernel calls required to reach a fitness score of 24.99 or better.
• **CountOnes**: Returns the number of bits equal to 1 in the input bit string (taken as a `long` variable). This function takes a maximum value of 64 for bit string containing all ones.

• **FirstOne**: Returns the position of the first bit equal to 1 from the left in the input bit string (taken as a `long` variable). If there are no ones, it returns 65. This function takes a maximum value of 65 for bit string with all zeros.

Note that the first function is highly non-convex, while the second one is convex. We observed that both WithConst and NoConst found the optimal solutions within a few seconds (few kernel calls), and hence precise comparisons will not be very accurate.

## 6 Conclusions

We proposed a variant of genetic algorithms to effectively make use of the constant memory to store elite population. This allows all blocks to share the elite population, allowing much faster convergence. Our experiments demonstrate that using constant memory results in same or better quality solutions in orders of magnitude less time.
7 References


