Differential Evolution with parallelised objective functions using CUDA

Primož Kralj
primoz@codehunter.eu

***Abstract*—Differential Evolution (DE) algorithms can be used in various fields for problem solving where we need to find an optimal (or close to optimal) solution but we don't have a clear, straightforward method to compute it. Unfortunately it can take a very long time to produce such a solution when implemented serially or even parallel on a Central Processing Unit (CPU). To reduce computation time we can utilise the power of Graphics Processing Units (GPU) using Compute Unified Device Architecture (CUDA) technology. Since GPUs are designed purely for computational tasks as opossed to CPUs which are more general-work oriented they can provide a great architecture for parallelising our DE method.**

Keywords - Differential Evolution, Compute Unified Device Architecture (CUDA), Graphics Processing Unit (GPU), Computational Performance

I. Introduction

Differential Evolution is a widely known optimization technique proposed by R. Storn and K. Price [1] where the main idea is to generate new population in each generation. The population consist of multiple vectors which represent a candidate solution each containing a set of real-valued numbers. These are the parameters of the optimization problem we are solving usually referred to as *genes* from biology jargon. The new population is formed from the population of previous generation with the means of genetic operators - mutation, crossover and selection all of which is described more profoundly in Section II Related work.
Implemented DE is tested using 28 functions from CEC'13 single objective real-parameter optimization competition [2] all of which are minimization problems. These are evaluated using sequential DE algorithm and CUDA-optimized parallel algorithm.

II. Related work

Our endeavour to develop a DE algorithm for minimizing the problem using CUDA is based upon L. P. Veronese's and R. A. Krohling's research paper *"Differential Evolution Algorithm on the GPU with C-CUDA"* [3]. However, we took a different approach to parallelizing DE using CUDA; in the mentioned paper authors are parallelizing the execution of genetic operators which are running on the GPU, speeding-up the process by atleast a factor of 20 (figure 2 and 3 in the related research paper). We are, on the other hand, trying to boost the execution by parallelizing the test functions themselves which is covered in Section III Implementation.
In the related article DE idea is explained as follows. Each candidate solution is described as a vector denoted by $x = [x_{i1}, x_{i2}, x_{i3}, ..., x_{in}]^T$ where $i = 1, ..., N_p$ represents individual's index in the population and n represents the individual's dimension which is in this case the dimension of the problem - in other words, number of parameters in our test functions. Before the DE is started the initial population is generated randomly. For our case it is important to take into consideration the evaluation criteria from CEC13 competition which states that the search range is in the interval $[-100, 100]]$, meaning random numbers for initial population need to be generated from within this interval. After the initialization the DE can be started where in each generation the genetic operators are applied to the population. For each individual mutation is performed first, then crossover and lastly selection.

A. Genetic operators

Mutation is the first operator which takes a random individual to which a weighted difference of two other random individuals is added. None of the three should be the same. The mutated vector v_i is calculated as

$$v_i = x_{r_1} + F_m(x_{r_2} - x_{r_3}) \tag{1}$$

where r_1,r_2 and r_3 are random numbers representing individuals indexes in the population, i is the index of mutated individual and F_m is the *mutation factor* which weights the difference between two random individuals by which we avoid search stagnation, as noted in the linked paper. Authors are suggesting the values in the range of $[0.5, 1]$.

Crossover is the second operator which produces a new set of N_p individuals. Each individual has some parameters (*genes*) from the parent in the original population and some from the mutated individual where the ratio is determined by *crossover rate* C_r for which authors recommend the interval $[0.8, 1]$. This leads to N_p *trial* individuals, each denoted as:

$$u_{i,j} = \{ \begin{array}{cc} v_{i,j} & \text{if rand(0,1)} \leq \text{C}_r \text{ or j=k} \\ x_{i,j} & \text{otherwise} \end{array} \tag{2}$$

where i stands for current index in the population and j for current parameter. Number k is a random number from interval $\{1, ..., N_p\}$ to make sure that at least one parameter from mutant individual is selected into trial individual.

Selection is the last genetic operator which simply determines which one of the two individuals is better comparing the parent vector $\mathbf{x_i}$ and the trial vector $\mathbf{u_i}$. To determine which is actually better we need to use some objective function *f()* which, in our case, are the CEC2013 competition test functions. Since they are minimization problems the individual selected into new population is

$$x_i = \{ \begin{array}{cc} u_i & \text{if } f(u_i) \leq \text{f(x}_i) \\ x_i & \text{otherwise} \end{array} \tag{3}$$

III. Implementation

Firstly the basic DE was implemented using genetic operators described above. The pseudocode is displayed in Listing 1 which also places other listings in context. To paralellize the execution of evalution functions the memory on GPU must be allocated as shown in Listing 2. The variable *POP_SIZE* holds the size of the population, *PROB_DIM* holds the number of parameters in an individual thus the dimension of an individual and *TEST_FUNCS* the number of test functions which is, in our case, 28. Points of interest are *d_mutants* which is a one-dimensional array containing all the parameters for every individual in the population. It would be more intuitive if 2D array would be used but that would cause overhead by copying each individual to the device allocated memory whereas with 1D copying occurs just once. and *d_f* which is also a 1D array containing all the calculated fitnesses in a way that for each individual it contains 28 values (28 objective functions results) which are then reduced to one fitness value per individual in the host code as shown in last line of Listing 3. Other arrays are holding the internal values and are not of a big importance to us. Still in Listing 1, the arrays are allocated using *cudaMalloc* function and some data is coppied from host to device - *OShift* and *M* arrays which have been filled in some other function with existing data from file. Since this data is used by test functions running on GPU it needs to be coppied to the device memory. Lastly in Listing 1 we set the contents of the remaining arrays to zero. Listing 3 containts the most important part of the code which is the execution of the *kernel* function. The kernel is the parallel portion of the application which is executed by many threads on the device. Before the kernel invocation the altered (by genetic operators) population is coppied to the device memory. Next the for loop is started with the kernel call. The purpose of the loop is to process 32 individuals per one kernel call meaning in each iteration 32 fitnesses - each consisting of 28 subfitnesses (28 test functions) - are calculated. We came to this optimal number by trial and error. The values inside angled brackets in kernel function invocation $(\langle\langle\langle 32, 28\rangle\rangle\rangle$ imply that 32 blocks will be launched, each consisting of 28 threads. This way each block represents one individual and each thread calculates one of the test functions. GPU actually process threads in *warps* of 32 but since there are only 28 test functions 4 threads will do no work. After each kernel call *cudaDeviceSynchronize()* function is called which blocks the execution until all threads finish. After the for loop we copy the results back to host and sum it up so each individual gets one fitness value. This could be implemented on the GPU too decrease the execution time even further.
In Listing 4 the kernel code is displayed. First we check that current thread number is under 28 since we have only 28 test functions with *threadIdx.x* which is a CUDA built-in variable. Then we check the thread's index again in a switch clause to determine which test function it will execute.
Lastly, in Listing 5, test function's declaration is shown. Because it executes on the GPU and is called from kernel function, *__device__* qualifier is written in front of it. Notice that kernel function too has a CUDA qualifier, *__global__*, which means that it is executed on the GPU but called from a host environment (CPU). Functions with *__device__* qualifier on the other hand can not be called from host.

```
initialize()
evaluate_population()

while(best_fitness>THRESHOLD || generation<MAX_GENERATIONS)
{
    apply_genetic_operators()
    evaluate_population()
}
```

Listing 1. DE pseudocode

```
double *d_mutants,*d_f,*d_OShift,*d_M,*d_y,*d_z;

int dbl_bytes=sizeof(double);
cudaMalloc((void**)&d_mutants,POP_SIZE*PROB_DIM*dbl_bytes);
cudaMalloc((void**)&d_f,POP_SIZE*TEST_FUNCS*dbl_bytes);
cudaMalloc((void**)&d_OShift,PROB_DIM*10*dbl_bytes);
cudaMalloc((void**)&d_M,10*PROB_DIM*PROB_DIM*dbl_bytes);
cudaMalloc((void**)&d_y,POP_SIZE*28*PROB_DIM*dbl_bytes);
cudaMalloc((void**)&d_z,POP_SIZE*28*PROB_DIM*dbl_bytes);

cudaMemcpy(d_OShift, OShift, PROB_DIM*10*dbl_bytes,
                              cudaMemcpyHostToDevice);
cudaMemcpy(d_M, M, 10*PROB_DIM*PROB_DIM*dbl_bytes,
                              cudaMemcpyHostToDevice);

cudaMemset(d_y,0,POP_SIZE*28*PROB_DIM*dbl_bytes);
cudaMemset(d_z,0,POP_SIZE*28*PROB_DIM*dbl_bytes);
cudaMemset(d_f,0,POP_SIZE*TEST_FUNCS*dbl_bytes);
```

Listing 2. Initialization

```
cudaMemcpy(d_mutants,mutants,POP_SIZE*PROB_DIM*dbl_bytes,
                                       cudaMemcpyHostToDevice);
  for(i=0; i<POP_SIZE/32; i++)
  {
    test_func<<<32,28>>>(&d_mutants[i*32],
                         &d_f[i*32*TEST_FUNCS],
                         &d_y[i*32*TEST_FUNCS*PROB_DIM],
                         &d_z[i*32*TEST_FUNCS*PROB_DIM],
                         d_OShift, d_M, PROB_DIM);
    cudaDeviceSynchronize();
  }
cudaMemcpy(f, d_f, POP_SIZE*TEST_FUNCS*dbl_bytes,
                              cudaMemcpyDeviceToHost);
sumFitnesses(fitnesses, f);
```

Listing 3. Evaluation

```
__global__ void test_func(double *mutants,double *fs,
                          double *y_ptr,double *z_ptr,
                          double *Os,double *Mr,int nx)
{
  if(threadIdx.x<28){
    switch(threadIdx.x){
      case 0:
        sphere_func(&mutants[blockIdx.x*nx],
                 &fs[blockIdx.x*28+threadIdx.x],nx,
                 &y_ptr[blockIdx.x*(28*nx)+threadIdx.x*nx],
                 &z_ptr[blockIdx.x*(28*nx)+threadIdx.x*nx],
                                              Os,Mr,0);
        break;
      case 1:
        ellips_func(&mutants[blockIdx.x*nx],
                 &fs[blockIdx.x*28+threadIdx.x],nx,
                 &y_ptr[blockIdx.x*(28*nx)+threadIdx.x*nx],
                 &z_ptr[blockIdx.x*(28*nx)+threadIdx.x*nx],
                                              Os,Mr,1);
        break;
      case 3:
        [...]
    }
  }
}
```

Listing 4. Kernel function

```
__device__ void shiftfunc (double *x,double *xshift,
                                int nx,double *Os){...}
```

Listing 5. One of the test functions

IV. Performance evaluation

The experiments were conducted on a laptop with Intel core i7 2GHz quadcore CPU and 8GB DDR3 1066MHz of main memory. GPU is NVIDIA GeForce GT540M with 2GB memory, 2 multiprocessors and Fermi architecture able to execute 96 threads at once, while the CUDA architecture is revision 2.1. Firstly experiments were run to determine the optimal size of the population size (although this should not matter in the real world tests). We found out that the ratio of consumed time was in favour to GPU while the population size was kept low, that is 64 individuals. If the population size exceeds this number the GPU and CPU execution times become the same or even GPU becomes slower. This is a major flaw of the current implementation, caused by the appliance of CUDA to the algorithm and will be inspected thoroughly in the future. In this field a lot of trial and error is needed to find out the optimal solution. The major improvement concerning execution time would also be the calculation of the sums of test functions on GPU inside of the kernel function rather than on the CPU. Despite this the results of the execution on the GPU are almost 4-times faster than on the CPU when the population size is small enough as shown in Figure 1.

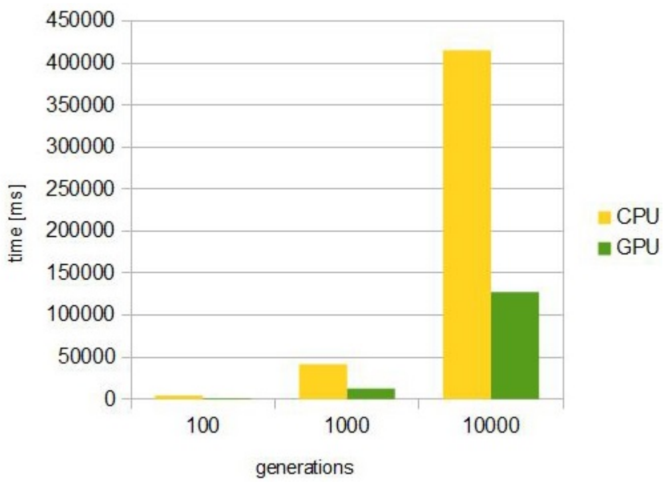

Fig. 1. computing time for test functions with population size 16

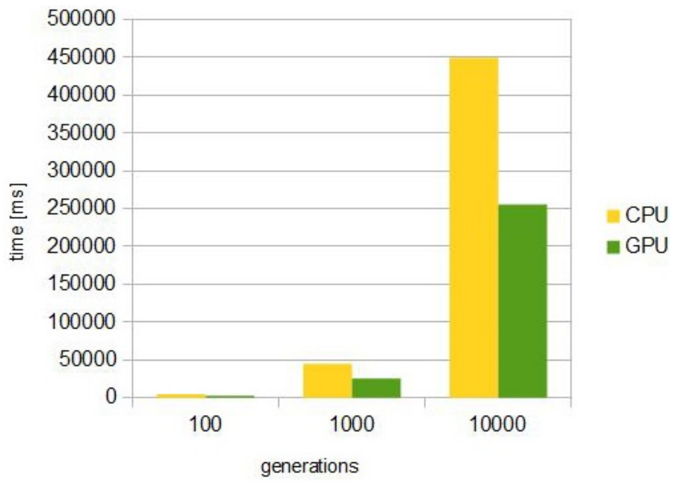

Fig. 2. computing time for test functions with population size 32

V. Conclusion

In this research paper we describe differential evolution algorithm along with the genetic operators needed to implement it. The implementation of CUDA parallelized test functions is also presented resulting in drastical reducement of the time needed to find a solution to a problem. This technology has proved its benefits unnumbered times by now in different fields of computing. With new hardware CUDA is providing more power as time goes on with the ability to run more and more concurrent threads on the GPU. A big role in the effectiveness plays the manner in which we apply the technology to our problem. If designed well, it takes the computing to the next level.

References

[1] R. Storn and K. Price, *"Differential Evolution — a simple and efficient heuristic for global optimization over continuous spaces"* Journal of Global Optimization, vol. 11, no. 4, pp. 341–359, 1997.

[2] J. J. Liang, B. Y. Qu, P. N. Suganthan, Alfredo G. Hernández-Díaz, *"Problem Definitions and Evaluation Criteria for the CEC 2013 Special Session on Real-Parameter Optimization"* http://www.ntu.edu.sg/home/EPNSugan/index_files/CEC2013/Definitions%20of%20%20CEC%2013%20benchmark%20suite%200117.pdf.

[3] De Veronese, L. P., and Renato A. Krohling. *"Differential evolution algorithm on the GPU with C-CUDA."* Evolutionary Computation (CEC), 2010 IEEE Congress on. IEEE, 2010.