A NEC-based parallel differential evolution algorithm with MKL/CUDA

Zi-Bin Pan

College of Mathematics and Informatics
South China Agricultural University
Guangzhou,510642
mathscau@163.com

Lei Yang*

College of Mathematics and Informatics
South China Agricultural University
Guangzhou,510642
Guangdong Provincial Key Laboratory of Food Quality and Safety
South China Agricultural University
Guangzhou,510642
Corresponding Author: yanglei_s@scau.edu.cn

Ze-Xin Xu

College of Mathematics and Informatics
South China Agricultural University
Guangzhou,510642
201725050325@stu.scau.edu.cn

Dong-Ya Wang

College of Engineering, Mathematics and Physical Sciences
University of Exeter
Exeter EX4 4QF UK
dw495@exeter.ac.uk

ABSTRACT. *Differential evolution (DE) is an effective global evolutionary optimization algorithm using to solve global optimization problems mainly in a continuous domain. But with the increase enlarge of problems' dimensions, the computational performance of DE don't satisfy the need. In order to further improve the performance of the algorithm, this paper propose A NEC-based parallel differential evolution algorithm with MKL/CUDA. Firstly, we analyzes the design of parallel computation of DE which can easily be executed in Math Kernel Library (MKL) and Compute Unified Device Architecture (CUDA). Then the essence of the exponential crossover operator is described and we point out that it cannot be used for better parallel computation. Later, we propose a New Exponential Crossover (NEC) operator that can be executed parallelly with MKL/CUDA. The simulation results validate the superiority of our parallel DE with MKL/CUDA,when tackling the optimization problem with large number of dimensions or decision variables.*
Keywords: Parallel differential evolution algorithm, Matrx calculation, High performance

1. **Introduction.** Evolutionary algorithm has important applications in many fields [1-4]. Among them, Differential evolution (DE) is simple and useful. DE is similar to the Genetic Algorithm (GA) and able to address optimization problems more efficiently and precisely [5, 6]. Since DE was proposed by Price and Storn in 1995 with several papers [7-9], it has attracted a lot of people to use and do further researches because of its simple realization, wide application, and good optimization results [10, 11].

Most researchers pay more attention to reforming DE in order to make it solve difficult problems better. But as for some large-scale optimization problems, although the evaluation of individuals is the bottleneck of the computational performance most of the time, we still found that if we use CPU/GPU parallel technologies in DE, we can get a dramatic increase in the performance.

In DE, each chromosome is an array containing a list of real numbers. This simple and useful structure makes it possible to calculate parallelly by matrix calculation techniques with the help of Math Kernel Library (MKL) or Compute Unified Device Architecture (CUDA). The former one performs well in matrix calculation with CPU while the other one can speed up the code with Graphics Processing Unit (GPU).

Until now there are many achievements in this field, for example, DE with General Purpose GPU (GPGPU), is a parallel version of DE executing in GPU, introduced by de Veronese and Krohling [12], Zhu [13], and Zhu and Li [14]. Cortes et.al [15] analyzed the influence of parameters on speedup and the quality of solutions of DE on a GPGPU, showing that DE on a GPGPU is not only running faster but also has a good optimization result. Recently, Meselhi et. al. designed a fast differential evolution for big optimization [16], which shows a big increase in the computational performance.

However, these parallel calculation achievements can only perform well in the DE algorithm with binomial crossover. They don't do well in some other versions of DE such as the DE algorithm with exponential crossover, which uses the exponential crossover (exp) instead of the uniform (binomial) crossover in recombination. The work of Tanabe [17] showed that the exp crossover can perform better in some optimization problems where the adjacent variables have some dependencies. Hence, it is still useful in DE. However, in this context, it will show that the exp crossover cannot be calculated parallelly enough as the uniform (binomial) crossover so that when using parallel calculation the former one runs much slower than the latter.

This paper designed the parallel DE with matrix calculation which can easily be executed in Math Kernel Library (MKL) and Compute Unified Device Architecture (CUDA), at the same time, we design a new exponential crossover operator,the extended experiments show that our algorithm has better performance. The rest of the paper is organized as follows. Section 2 shows several parallel-computational algorithms in a different part of DE. In section 3, it will uncover the essence of exponential crossover and changed it with the help of the probability theory so that it can be faster as well as be able to speed up on MKL/GPU. In section 4, we will compare the performances of DE, DE on MKL (MKL-DE), and DE with CUDA (CUDA-DE).

2. **Parallel Differential Algorithm.** The differential evolutionary algorithm processes the population with N individuals. Each individual owns a chromosome which is an n-dimension vector $x_{i,g}$ made of real numbers. i is the index number of the individual in the population, g indicates the generation to which a vector belongs. x_g is a matrix that stores all the $x_{i,g}$ in the population. After initialization, base vector selection, differential mutation, recombination, and selection, a new generation is created and after several iterations, the solutions of the population can get better and better. The pseudocode of DE is shown in Algorithm 1.

Algorithm 1 Differential Evolution (DE).

1: Set parameters C_r, F and N_p and iteration counter $g \leftarrow 0$
2: Initialize population $P_{x,g} = (x_{i,g})$, $i = 0, 1, \dots, N - 1$
3: $fit \leftarrow \text{evaluate}(x_g)$
4: **while** Stop Criteria is False **do**
5:      **for** i = 0 to pop_size-1 **do**
6:         Select a index number: r_0
7:         $v_{i,g} \leftarrow \text{differential\_mutation}(F, x_{i,g}, r_0)$
8:         $u_{i,g} \leftarrow \text{recombination}(Cr, v_{i,g}, x_{i,g})$
9:         $fit' \leftarrow \text{evaluate}(u_{i,g})$
10:        **if** $fit' < fit_i$ **then**
11:           $x_{i,g+1} \leftarrow u_{i,g}$
12:           $fit_i \leftarrow fit'$
13:        **else**
14:           $x_{i,g+1} \leftarrow x_{i,g}$
15:        **end if**
16:      **end for**
17: **end while**

In Algorithm 2, the loop between line 5 and line 14 can be calculated parallelly. Suppose we denote a parallel calculation task as a "job", meaning that the device we used can do all the jobs parallelly. Hence, each iteration of the loop between line 5 and line 16 can be allocated to a job. In addition, in line 7, line 8, and line 12, we need to loop all the elements of the chromosome vector, since each loop is independent. However, it is not good enough to set sub-jobs in a parallel calculation job: As we can't control the running time of line 10 (evaluation), it's not good to allocate each loop from line 5 to line 16 to a single job for a thread to do by including a time-consuming evaluation task in it. It's necessary to change the algorithm flow design of DE to a new one where the procedure of line 10 is separated so that other procedures of it can be calculated parallelly on MKL/CUDA. The pseudocode of the parallel DE framework can be seen in Algorithm 2.

Algorithm 2 Parallel Differential Evolution framework.

1: Set parameters C_r, F and N_p and iteration counter $g \leftarrow 0$
2: Initialize population chromosome matrix x_g
3: $fit \leftarrow \text{evaluate}(x_g)$
4: **while** Stop Criteria is False **do**
5:      shuffle the order of the individuals
6:      $r_0 \leftarrow \text{base\_index\_selection}(N_p)$
7:      $mask \leftarrow \text{crossover}(Cr, N_p, D)$
8:      $u_g \leftarrow \text{copy}(x_g)$
9:      $u_g[mask] \leftarrow \text{differential\_mutation}(F, x_g, mask, r_0)$
10:     $fit' \leftarrow \text{evaluate}(u_g)$
11:     $idx \leftarrow \text{where}(fit' < fit)$
12:     $x_{g+1} \leftarrow \text{copy}(x_g)$
13:     $x_{g+1}[idx, :] \leftarrow u_g[idx, :]$
14:     $fit[idx] \leftarrow fit'[idx]$
15: **end while**

Algorithm 3 Random Base Index Selection.

Input: N_p.
Output: An array_like r_0.
1: $r_0 \leftarrow N_p \cdot \text{floor}(\text{rand}(0,1,\text{size}=N_p))$

In Algorithm 2, r_0 in line 6 is a vector that stores the indices of individuals chosen by the selection method. The base vectors in DE are equal to $x_g[r_0,:]$. r_0 can be created parallelly. In line 7, $mask$ is a 0-1 matrix storing the message whether which element of the u_g matrix needs to be changed its value to the mutation result or not (1 means yes while 0 means no). It's easy to see that the order between the crossover and the mutation is exchanged, which can decrease the running time but get the same result as the traditional DE.The DE algorithm with binomial crossover can be designed according to Algorithm 2. This framework can be used in many different versions of DE. For example, as to the DE algorithm with binomial crossover, we can use this framework parallelly. Except line 10 (evaluation), all other codes from line 5 to line 14 can be executed parallelly by matrix calculation with the help of MKL/CUDA, which means that in many underlying operating procedures, for example, how many sub-threads are used in the parallel calculation, is setting to a suitable state by MKL/CUDA automatically. There are many kinds of base index selection in DE, some can be seen in Algorithm 3, 4, and 5 The basic parallel mutation of "DE/*/1/bin" can be seen in Algorithm 6, and the parallel uniform (binomial) crossover can be seen in Algorithm 7.

Algorithm 4 Random Offset Base Index Selection.

Input: N_p.
Output: An array_like r_0.
1: $idx \leftarrow [0, 1, \cdot, N_p - 1]$
2: $r_g \leftarrow \text{ceil}((N_p - 1) \cdot \text{rand}(0,1))$
3: $r_0 \leftarrow (idx + r_g) \bmod N_p$

Algorithm 5 Permutation Base Index Selection.

Input: N_p.
Output: An array_like r_0.
1: $idx \leftarrow [0, 1, \cdot, N_p - 1]$
2: $r_0 \leftarrow \text{shuffle}(idx)$

The three selections above can all be easily calculated on MKL/CUDA.

In traditional DE, the crossover is executed after mutation. After the crossover is done, quite many elements of $v_{i,g}$ which cannot be given to $u_{i,g}$ are wasted. So the $mask$ helps to speed up DE.

The Algorithm 7 implements the uniform (binomial) crossover, the formula is as follows:

$$u_{i,j,g} \leftarrow \begin{cases} v_{i,j,g}, & if \quad rand(0,1) < Cr \quad or \quad j = j_{rand} \\ x_{i,j,g}, & otherwise \end{cases} \tag{2.1}$$

j is the index number which values in $0, 1, \cdots, D - 1$. the $mask$ is a 0-1 matrix that marks whether the elements of u_g matrix which are need to be crossed with v_g matrix or not.

The algorithms above can easily be done in MKL/CUDA, because most procedures of them are mainly matrix calculation. In section 4 we will show the acceleration under MKL/CUDA.

Algorithm 6 Basic Parallel Differential Mutation.

Input: $F, x_g, mask, r_0$.
Output: An array_like *results*.
  1: $N_p, Lind \leftarrow$ the size of x_g
  2: $r_1 \leftarrow r_0 + (N_p - 1) \cdot \text{ceil}(\text{rand}(0,1,\text{size}=N_p)) \bmod N_p$
  3: $r_2 \leftarrow r_1 + (N_p - 1) \cdot \text{ceil}(\text{rand}(0,1,\text{size}=N_p)) \bmod N_p$
  4: $results \leftarrow (x_g[r0,:])[mask] + F \cdot ((x_g[r1,:])[mask] - (x_g[r2,:])[mask])$

Algorithm 7 Parallel Uniform (Binomial) Crossover (MKL/CUDA-BIN).

Input: $Cr \in [0,1], N_p, D$.
Output: $mask$.
  1: $mask \leftarrow$ an $N_p \times D$ matrix in which all elements are 0
  2: $j_{rand} \leftarrow \text{randint}(D, \text{size}=(N_p))$
  3: $idx \leftarrow [0, 1,_{...}, N_p - 1]$
  4: $mask[idx, j_{rand}] \leftarrow 1$
  5: $R \leftarrow \text{rand}(0,1,\text{size}=(N_p, D))$
  6: $mask[R < Cr] \leftarrow 1$

---

3. New Exponential Crossover.

3.1. The Essence of the Exponential Crossover.

The exponential crossover in Differential Evolution is similar to 1 or 2 point crossover in the Genetic Algorithm (GA) [8], which is widely used in the DE algorithm with exponential crossover. Its implementation is shown in Algorithm 8. In each generation g, a mutation vector $v_{i,g}$ created from the parent chromosome vector $x_{i,g}$ by applying some mutation strategies. After that, the mutation vector $v_{i,g}$ is crossed with $x_{i,g}$ in order to create trial vector $u_{i,g}$ by applying the exponential crossover. $Cr \in [0,1]$ is the crossover rate, j_{rand} is a decision variable index randomly selected from $[0, D-1]$.

The Altorithm 6 implements the differential mutation in the "DE/*/1/bin", the formula is as follows:

$$u_i \leftarrow x_{r0} + F \cdot (x_{r1} - x_{r2}) \tag{3.1}$$

Algorithm 8 Exponential Crossover.

Input: $x_g, v_g, Cr \in [0,1]$
Output: u_g
  1: $N_p, D \leftarrow$ the shape of x_g
  2: **for** $i \leftarrow [0, 1,_{...}, N_p - 1]$ **do**
  3:      $u_{i,g} \leftarrow x_{i,g}$
  4:      j is randomly selected from [0,D-1], $L = 0$
  5:      **do**
  6:          $u_{i,j,g} \leftarrow v_{i,j,g}$
  7:          $j \leftarrow (j + 1) \bmod D$
  8:          $L \leftarrow L + 1$
  9:      **while** rand[0,1)¡Cr and $L < D$
10: **end for**

However, the exponential crossover is seldom used in parallel DE practices. We can easily find the reason. It's clear that if we allocate each execution of the exponential crossover of each individual as a parallel job, that each job needs to loop the chromosome

vector to check whether $u_{i,g}$ is needed to be crossed. This procedure cannot run parallelly. As to the uniform (binomial) crossover, the process of checking whether bit of chromosome is needed to be crossed or not is independent. Thus, the latter is more suitable for parallel calculation. Figure 1 shows the runtime comparison between parallel exponential crossover (p-exp) and parallel uniform (binomial) crossover (p-bin). The exponential crossover has a
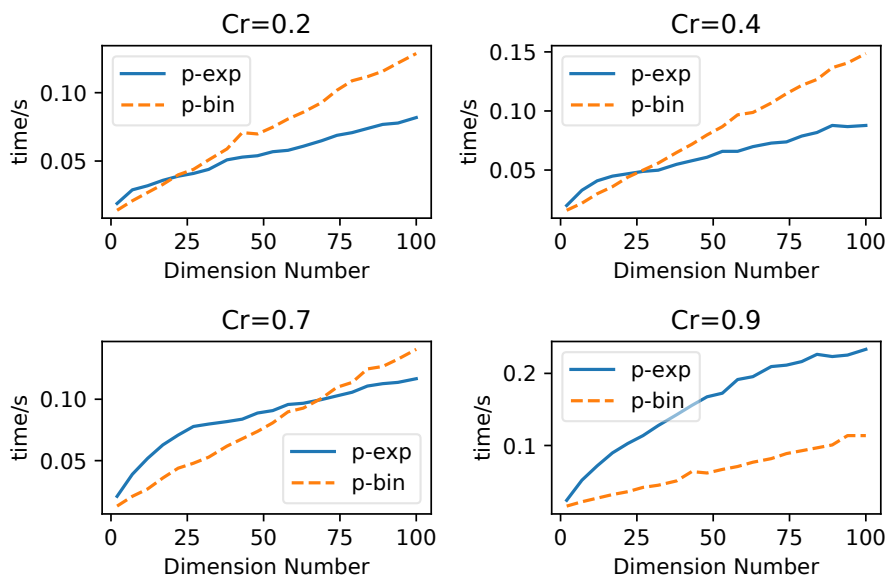

FIGURE 1. Running Time Comparison between p-exp and p-bin in $N_p = 1000$

congenital advantage that when Cr is relatively small and the dimension of $u_{i,g}$ is large, it doesn't need to traverse all bit of $u_{i,g}$ to cross, because the loop will be probability stopped on the half-way. So we can see in Figure 1 that when $Cr = 0.2$ and $Cr = 0.4$, as the dimension of the chromosome vector is getting larger, the p-exp has a higher performance than p-bin. However, when Cr is getting larger, p-exp is slower than p-bin. That is because each parallel job of p-exp has more work to do: it needs to traverse the bits of $u_{i,g}$ to check whether it is needed to cross.

In some related work, the L of each $v_{i,g}$ can be calculated before the crossover, so that remain procedures of the exp crossover can be calculated parallelly at a higher speed. That can be seen in [18]. However, when calculating the L, they still use a loop same to the traditional exp crossover, so that these improvements are still unsuitable for parallel calculation.

If we reconsider the exponential crossover, we can see that the L, which is equal to the length of crossed bits in $u_{i,g}$, is subject to geometric distribution, not the exponential distribution. But as the geometric distribution has some relationship with the exponential distribution, it's unnecessary to change the name of the exponential crossover. Instead, we can use the probability theory to redesign the exponential crossover in order to make it faster.

According to Algorithm 8, we can get

$$L \sim G\left(1 - Cr\right)$$

Where G denotes the geometric distribution, meaning that

$$P\left\{L = n\right\} = Cr^{n-1}\left(1 - Cr\right), n \geq 1 \tag{3.2}$$

What we use here is the property that

$$G(n) = P\{L \le n\} = 1 - \sum_{i>n}(1-Cr)Cr^{i-1} = 1 - Cr^n \tag{3.3}$$

Now we need to generate a random number that is subject to the geometric distribution. Denote X as an exponentially distributed random variable with parameter λ, so we have $L = \lceil X \rceil$, where $\lceil \quad \rceil$ is the ceil (or smallest integer) function. L is a geometrically distributed random variable with parameter $p = 1 - e^{-\lambda}$. p is the parameter of the geometric distribution. In the exponential crossover, p is equal to $1 - Cr$. Thus, we obtain $\lambda = -\ln Cr$. If we denote $F_X(x)$ as the distribution function of the exponential distribution, we can get

$$F_X(x) = 1 - Cr^x \tag{3.4}$$

According to the inversion method [19] and equation 3.4, we can get

$$U_1 = F_X(x) \tag{3.5}$$

where U_1 is a random number in (0,1). Is clear that $U = 1 - U_1$ is also a uniform distribution random number in (0,1). Thus, we can get a random sampling real number x which is exponentially distributed.

$$x = \frac{\ln(U)}{\ln(Cr)} \tag{3.6}$$

So, the L in the exponential crossover can be sampled randomly by the following equation instead of traversing elements of the chromosome vector.

$$L = \left\lceil \frac{\ln U}{\ln Cr} \right\rceil \tag{3.7}$$

3.2. **New Exponential Crossover (NEC).** In the new exponential crossover, since $Cr \in [0,1]$, it is necessary to duel with the three special cases: $Cr = 0$, $Cr = 1$, and L cannot be bigger than the dimension number D of the $u_{i,g}$. Thus, the equation to generate L is as follows.

$$L = \begin{cases} 1, Cr = 0 \\ \min\left(\left\lceil \frac{\ln U}{\ln Cr} \right\rceil, D\right), Cr \in (0,1) \\ D, Cr = 1 \end{cases} \tag{3.8}$$

After generating L, we can know which bits of $u_{i,g}$ are needed to be crossed directly. So we can redesign the exponential crossover. Algorithm 9 shows the new design of the exponential crossover.

Figure 2 shows the frequency of L in the new exponential crossover (NEC) and the traditional exponential crossover (exp) in 1000000 times experiments with $D = 10$. We can see that we can get similar results as the traditional exponential crossover by using NEC.

Figure 3 and 4 show the higher speed of NEC in 100 experiments with different Cr.

Comparing the traditional exponential crossover and NEC, the latter calculates the total crossed length L at first so that when looping $u_{i,g}$, it only needs to compare the crossed length l with L, which makes it much faster than the traditional exponential crossover.

Moreover, the procedure of generating all L of all individuals in DE can be also executed parallelly, so that we can make full use of MKL/CUDA to speed up the procedure of the new exponential crossover. Firstly, the column vector Ls, that stores all the L of the $u_{i,g}$, $i = 0, 1, 2, \ldots, N_p$ are sampled parallelly by applying equation 3.8. Then, with the help of

---

Algorithm 9 New Exponential Crossover (NEC).

Input: x_g(the current individual), v_g(mutation), Cr(probability of variation)$\in [0,1]$
Output: u_g(experimental individuals)
  1: $N_p, D \leftarrow$ the shape of x_g
  2: **for** $i \leftarrow [0, 1,_{...}, N_p - 1]$ **do**
  3:     $u_{i,g} \leftarrow x_{i,g}$
  4:     j is randomly selected from [0,D-1], $l = 0$
  5:     calculate L by equation 3.8
  6:     **do**
  7:        $u_{i,j,g} \leftarrow v_{i,j,g}$
  8:        $j \leftarrow (j + 1) \bmod D$
  9:        $l \leftarrow l + 1$
 10:    **while** $l < L$
 11: **end for**
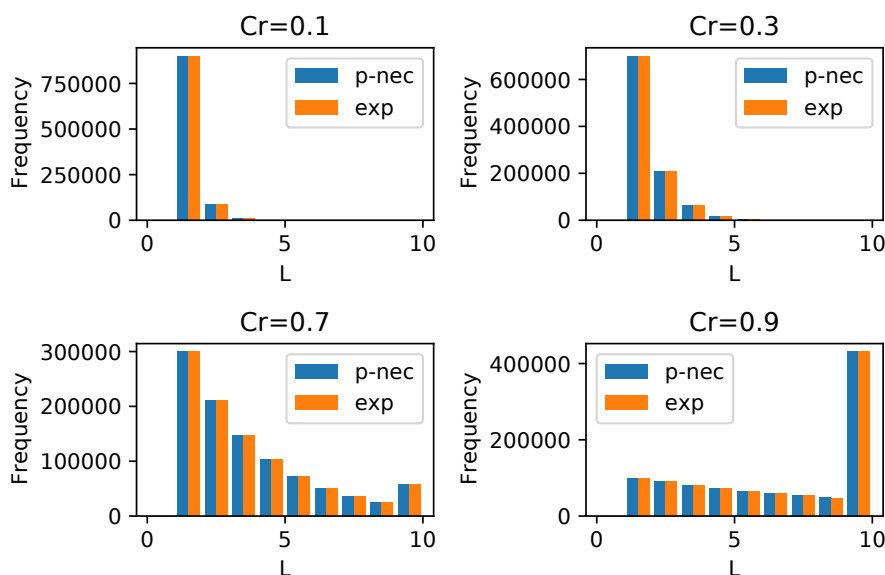
FIGURE 2. The frequencies of L between nec and exp

the Ls, we can create a matrix $mask$ parallelly which denotes whether every bit of all the $u_{i,g}$ of all chromosome vectors need to be crossed or not. Finally, we can do the cross of every bit of the chromosomes parallelly on MKL/CUDA according to the matrix $mask$. It is worth noting that the new exponential crossover on MKL/CUDA (MKL/CUDA-NEC) only returns a signal matrix $mask$, which is similar to Algorithm 7, that is because in parallel differential evolution on MKL/CUDA (see Algorithm 2), the mutation is executed after the crossover. With the help of the $mask$, the mutation only needs to mutate the elements where the element of the $mark$ is equal to 1. Algorithm 10 is the pseudocode of the parallel new exponential crossover (p-nec).

Comparing MKL/CUDA-nec to p-exp we mentioned above, we can see that all the procedures of the former one are the matrix calculation, which can be easily run thoroughly parallelly with the help of MKL/CUDA. Thus, it runs much faster than p-exp. Figure 5, 6, and 7 will show the runtime comparison among CUDA-nec (using CUDA to calculate NEC), MKL-nec (using MKL to calculate NEC), and p-exp when the population size N_p is 100, 1000 and 5000 in 100 repeat experiments. The D (dimension of $u_{i,g}$) is set from 2
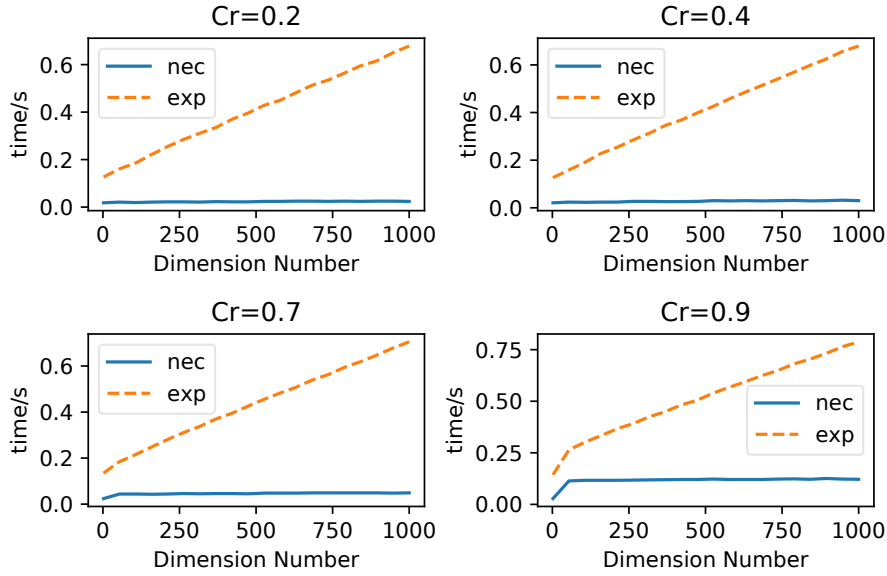
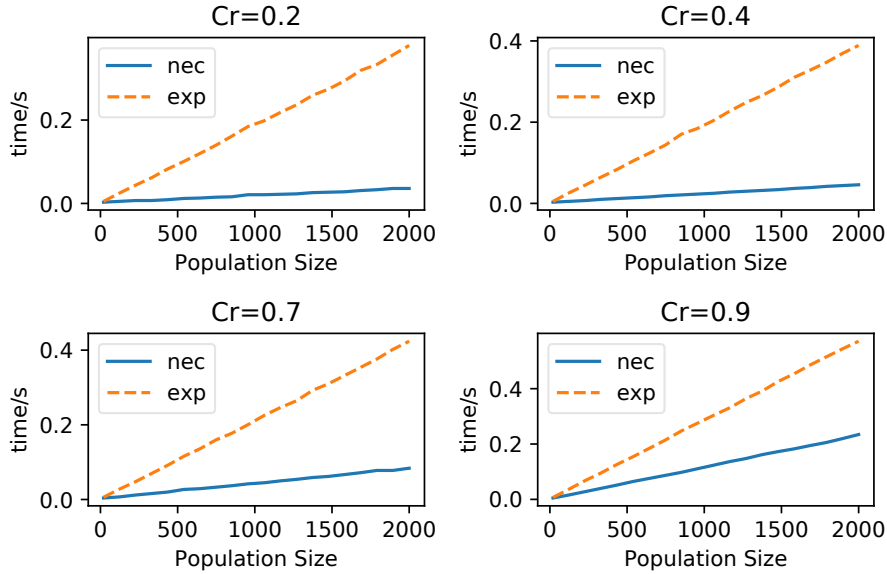FIGURE 3. Running Time Comparison between nec and exp in $N_p = 1000$.

FIGURE 4. Running Time Comparison between nec and exp in $D = 100$.

to 5000. We can see from these pictures that when N_p is small, MKL-NEC is faster than the others, but when N_p is large and the dimension number of the chromosome is getting larger and larger, CUDA-NEC can execute faster. That is because in big optimization, the runtime of data's transport between the memory device and the GPU is no longer the bottleneck, thus, we get a higher speedup by using GPU to calculate.

Table 1 shows more experiment results. Each experiment are done in Cr=0.2, 0.4, 0.6, 0.8 and 1.0 for 100 times then records the total runtime. We can see that when the dimension of the decision variable (D) and the population size (N_p) is relatively small, the new exponential crossover on MKL has a higher performance. But when D and N_p are relatively large, it's better to use CUDA-NEC as well as CUDA-bin. Moreover, no matter

Algorithm 10 Parallel Version of New Exponential Crossover (MKL/CUDA-NEC).

Input: $Cr \in [0,1], N_p, D$.
Output: $mask$.
1: $j_{rand} \leftarrow$ randint(D, size=(N_p))
2: create a column vector Ls by applying equation 3.8 parallelly
3: $Seq \leftarrow$ repmat($[0, 1,_{...}, D-1]$, size=$(N_p,1)$)
4: $j \leftarrow$ repmat(j_{rand},size=$(1,D)$)
5: $j' \leftarrow$ repmat(Ls,size=$(1,D)$)
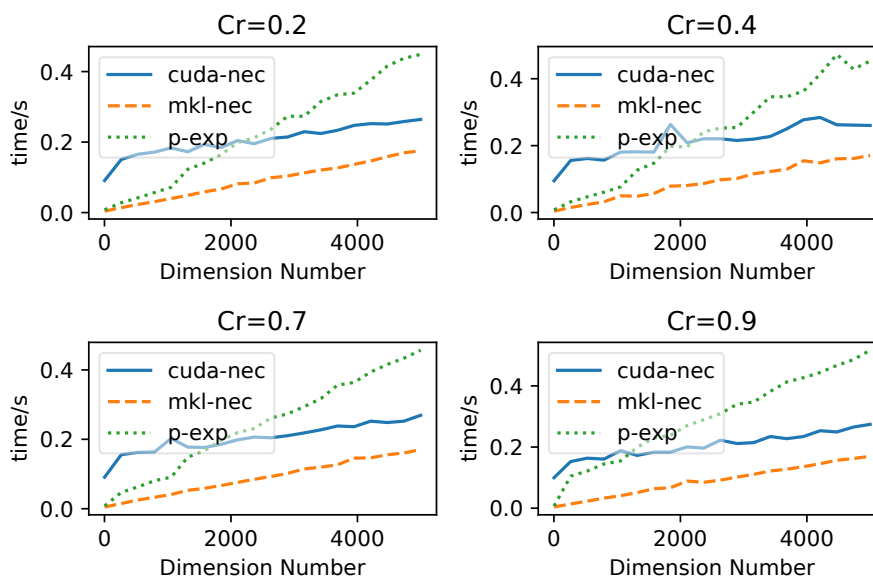6: $mask \leftarrow (j <= Seq)$ XOR $(j' < Seq)$

FIGURE 5. Running Time Comparison among CUDA-nec, MKL-nec, and p-exp when $N_p = 100$

MKL is used or not, when D and N_p are getting larger, the uniform (binomial) crossover costs more time than the new exponential crossover. That is because, in the uniform (binomial) crossover, we need to create an $N_p \times D$ matrix in which elements are random number. As to the new exponential crossover, with the help of equation 3.8, we only need to create a random column vector which length is N_p. But if we use CUDA to speed up by GPU when the computation scale is large, the runtime gap between CUDA-nec and CUDA-bin is not so significant.

4. **Performance Evaluation of DE based on MKL/CUDA.** Our DE programs are implemented in Python. The MKL is implemented by Numpy-MKL and the GPU environment is set up in Cupy [20] with CUDA 10.0. In order to overcome the performance deficiency of Python, the key codes are implemented in Cython, which is the C-Extensions for Python. There are mainly three different versions of DE in this experiment: no-parallel DE, DE on MKL (MKL-DE), and DE in CUDA (CUDA-DE). In addition, as to the DE, we arrange two sub-versions: "DE/rand/1/bin" and "DE/rand/1/L". It is worth to note that the crossover of "DE/rand/1/L" is the parallel version of the new exponential crossover mentioned in Algorithm 10. All of them are running on an Intel® i5-9600K, 16GB memory, and a common NVIDIA GeForce™ GTX 760 GPU.
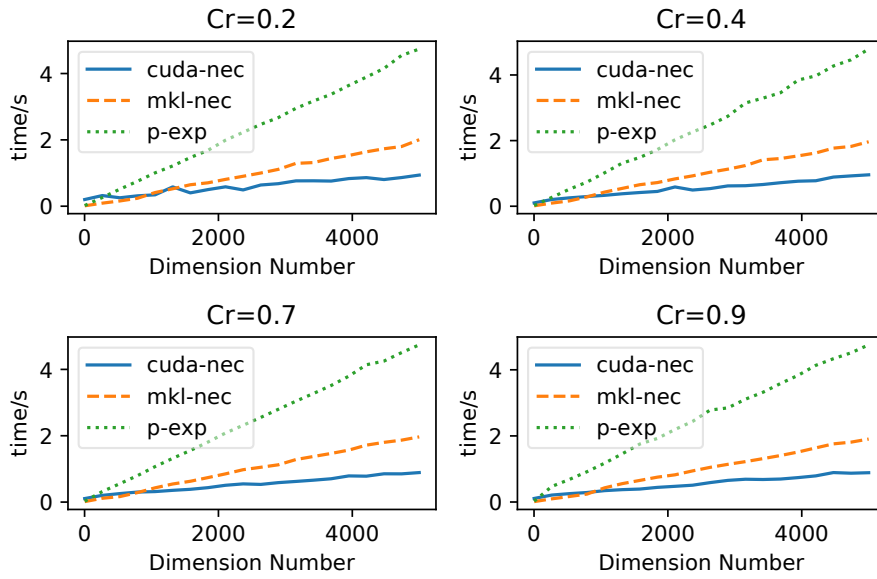
FIGURE 6. Running Time Comparison among CUDA-nec, MKL-nec, and p-exp when $N_p = 1000$
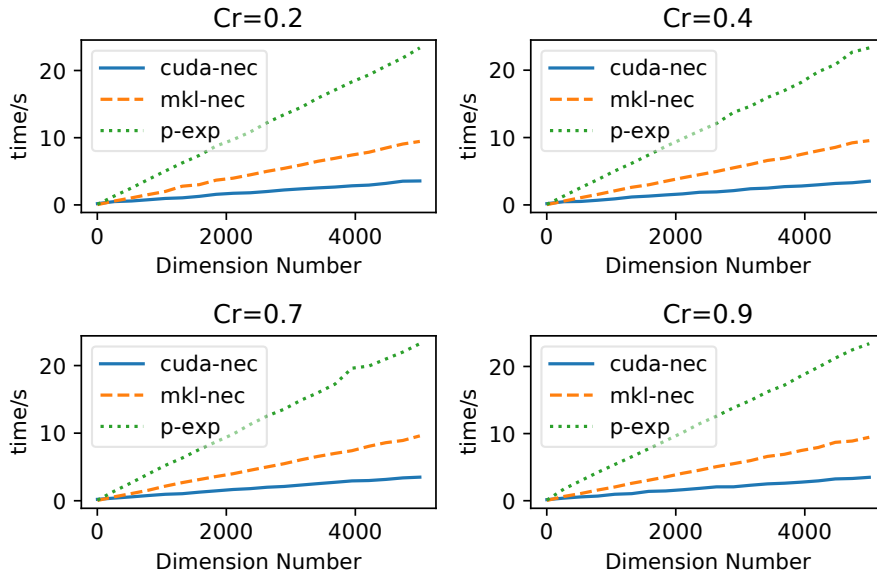

FIGURE 7. Running Time Comparison among CUDA-nec, MKL-nec, and p-exp when $N_p = 5000$.

In this experiment, we focus our attention on three well-known benchmark functions: Ackley, Griewank and Rosenbrock [21]:

- Ackley function—multimodal, separable,

$$f(x) = -20e^{-0.02\sqrt{\frac{1}{D}\sum\limits_{j=1}^{D} x_j^2}} - e^{\frac{1}{D}\sum\limits_{j=1}^{D} \cos 2\pi x_j} + 20 + e$$

$x_j \in [-30, 30]$, $x^* = (0, 0, \dots, 0)$, $f(x^*) = 0$.

TABLE 1. Running times' comparison of several DE Crossovers.

D	Np	exp	nec	bin	MKL -nec	CUDA -nec	MKL -bin	CUDA -bin
10	100	0.05	0.03	0.06	0.02	1.76	0.02	6.99
10	1000	0.44	0.12	0.51	0.05	1.18	0.07	0.64
10	10000	4.34	1.17	4.99	0.31	1.34	0.53	0.94
100	100	0.11	0.08	0.23	0.03	1.16	0.05	0.66
100	1000	1.09	0.61	2.18	0.12	1.23	0.32	0.64
100	10000	10.86	6.15	22.57	0.98	1.11	4.18	1.36
1000	100	0.75	0.57	1.9	0.13	0.66	0.32	0.56
1000	1000	7.53	5.44	19.16	0.73	1.07	3.98	1.00
1000	10000	74.77	54.52	192.46	9.16	4.31	41.76	4.56
10000	100	7.1	5.46	18.67	0.99	1.48	4.1	0.97
10000	1000	70.82	53.58	187.6	7.97	4.43	41.49	3.91
10000	10000	710.4	538.05	1868.47	77.82	33.26	421.17	39.14

- Griewank function—multimodal, nonseparable,

$$f(x) = \sum_{j=1}^{D} \frac{x_j^2}{4000} - \prod_{j=1}^{D} \cos\left(\frac{x_j}{\sqrt{j}}\right) + 1$$

$x_j \in [-400, 400], x^* = (0, 0,_{,...,} 0), f(x^*) = 0.$
- Rosenbrock function—multimodal, separable,

$$f(x) = \sum_{j=1}^{D-1} \left[100\left(x_j^2 - x_{j+1}\right)^2 + (1 - x_j)^2 \right]$$

$x_j \in [-5.12, 5.12], x^* = (1, 1,_{,...,} 1), f(x^*) = 0.$

The parameters setting of DEs in the experiment can be seen in Table 2, where the F and Cr are referred to K. Price and R. Storn's work [21]. The dimension D of these three benchmark functions and the population size N_p are set to different values in order to compare the calculational performance in different computational scales. The results can be seen in the tables range from Table 3 to Table 5.

TABLE 2. Parameters Setting

Function	D	Np	F	Cr	MAX GEN
Ackley	1000	500	0.5	0.2	30000
Griewank	500	300	0.25	0.45	20000
Rosenbrock	50	100	0.65	0.95	10000

Figure 5, Figure 6 and Figure 7 are the comparison diagrams of algorithm running time when the population is 100, 1000, 5000 respectively. Table 1 is the running times'comparison of several DE Crossovers. Table 3 is the comparison of variant DEs in 1000-D Ackley with 500 individuals and 30000 generations. Table 4 is the comparison of variant DEs in 500-D Griewank with 300 individuals and 20000 generations. Table 5 is the comparison of variant DEs in 50-D Rosenbrock with 100 individuals and 10000 generations.

From the results, we can see that the DE (see Algorithm 2) on MKL/CUDA is much faster than traditional DE (see Algorithm 1). And between the DE on MKL (MKL-DE) and the DE in CUDA (CUDA-DE), when the dimension of the variable is relatively small,

TABLE 3. Comparison of variant DEs in 1000-D Ackley with 500 individuals and 30000 generations

Algorithms	best value	best gen	time/s	speedup
DE/rand/1/L	2.32E-05	30000	2742.93($\pm$105)	-
MKL-DE/rand/1/L	2.09E-05	29997	2314.79($\pm$52)	1.18
CUDA-DE/rand/1/L	2.62E-05	29991	323.97($\pm$5)	8.47
rand/1/bin	2.15E-04	29998	3323.53($\pm$93)	-
MKL-DE/rand/1/bin	2.84E-04	29951	2784.57($\pm$98)	1.19
CUDA-DE/rand/1/bin	2.28E-04	29983	321.45($\pm$11)	10.34

TABLE 4. Comparison of variant DEs in 500-D Griewank with 300 individuals and 20000 generations

Algorithms	best value	best gen	time/s	speedup
DE/rand/1/L	8.88E-16	18366	563.84($\pm$84)	-
MKL-DE/rand/1/L	3.71E-16	19855	475.81($\pm$45)	1.19
CUDA-DE/rand/1/L	1.73E-16	19936	149.03($\pm$5)	3.78
rand/1/bin	2.22E-16	8125	732.84($\pm$114)	-
MKL-DE/rand/1/bin	2.22E-16	6291	524.01($\pm$95)	1.40
CUDA-DE/rand/1/bin	2.22E-16	6041	143.95($\pm$8)	5.09

TABLE 5. Comparison of variant DEs in 50-D Rosenbrock with 100 individuals and 10000 generations

Algorithms	bestValue	best_gen	time/s	speedup
DE/rand/1/L	2.04E-07	9991	15.43($\pm$2)	-
MKL-DE/rand/1/L	4.28E-07	9989	9.96($\pm$1)	1.55
CUDA-DE/rand/1/L	1.16E-07	9996	58.92($\pm$3)	0.26
rand/1/bin	3.71E-10	10000	19.13($\pm$2)	-
MKL-DE/rand/1/bin	2.17E-10	9998	10.01($\pm$1)	1.91
CUDA-DE/rand/1/bin	9.93E-10	10000	56.72($\pm$4)	0.34

for example, $D = 50$, MKL-DE is the fastest. While D is relatively larger, for example, $D = 500$ or $D = 1000$, and the population size N_p is large, CUDA-DE is fastest. That is because Algorithm 2 is mainly matrix calculations, which can be executed very fast on MKL/CUDA. And when the matrix scale is quite large, the CUDA can do the calculation much faster by using GPU to accelerate.

5. **Conclusions and Future Work.** This paper firstly presents an available parallel differential evolution framework which is suitable to be built on MKL/CUDA. The procedure of this parallel DE is mainly matrix calculation so that with the help of the MKL/CUDA, the DE program can be run in a high speed even parallelly if possible. Moreover, it does not need to worry about the parameters of the core process of parallel computing, such as the number of CPU cores, GPU cores, threads, etc. These parameters are automatically and appropriately set in MKL / CUDA.

Later, we analyzed the disadvantage of the exponential crossover of DE, which is inefficient and isn't suitable for MKL/CUDA to calculate totally parallelly. Hence, a new exponential crossover is presented in Algorithm 9, which can run faster. Moreover, a parallel version of the new exponential crossover is proposed in 10, which is all the matrix calculation so that it can be executed in a high speed with the help of MKL/CUDA.

From the experimental results presented in Section 3, we can see that the computational performance of the parallel version of the new exponential crossover catches up with the parallel binomial (uniform) crossover on MKL/CUDA.

The experimental results presented in Section 4 shows the higher performance of the parallel DE on MKL/CUDA. Especially when the dimension of the decision variable of the optimization problem is quite large, the DE on CUDA has a higher acceleration ratio.

Since differential evolution is a stochastic optimization evolution algorithm that is inherently parallel and it is similar to many other evolutionary algorithms, the whole area of evolutionary computation may also get benefit from the MKL/CUDA. In the future, we will make more effort to speed up more evolutionary algorithms.

REFERENCES

[1] E. K. Wang, C. M. Chen, S. M. Yiu, M. M. Hassan, M. Alrubaian, G. Fortino, Incentive evolutionary game model for opportunistic social networks, *Future Generation Computer Systems,* vol. 102, pp. 14-29, 2020.

[2] M. T. Wu, G. Srivastava, S. Tayeb, C. W. Lin, A PSO-Based Sanitization Pro-cess with Multi-thresholds Model, 2021, https://doi.org/10.1007/978-3-030-68799-1_32

[3] M. T. Wu, G. Srivastava, J. C. W. Lin, Q. Teng, A multi-threshold ant colony system-based sanitization model in shared medical environments, *ACM Transactions on Internet Technology (TOIT),* vol. 21, no. 2, pp. 1-26, 2021.

[4] Q. Yang, S. C. Chu, J. S. Pan, C. M. Chen, Sine cosine algorithm with multigroup and multistrategy for solving CVRP, *Mathematical Problemsin Engineering,* Vol. 2020, 8184254, 2020.

[5] K. R. Opara, J. Arabas, Differential evolution: a survey of theoretical analyses, *Swarm and Evolutionary Computation,* vol. 44, pp. 546–558, 2019.

[6] L. Pham, A. Malinowski, T. Ba Rtczak, Comparative study of derivative free optimization algorithms, *IEEE Transactions on Industrial Informatics,* vol. 7, no. 4, pp. 592-600, 2011.

[7] K.V. Price, Differential evolution: a fast and simple numerical optimizer, *Proceedings of North American Fuzzy Information Processing,* pp. 524-527, 2016.

[8] R. Storn, and K. Price, Differential evolution: a simple and efficient adaptive scheme for global optimization over continuous spaces, *Journal of Global Optimization,* vol. 11, pp. 341-359, 1997.

[9] R. Storn, On the usage of differential evolution for function optimization, *Proceedings of North American Fuzzy Information Processing,* pp. 519-523, 1996.

[10] H. Liu, S. L. Wang, D. H. Zhao, Initialalignment for point cloud registration by improved differential evolution algorithm, *Optik - International Journal for Light and Electron Optics,* vol. 2021, 166856, 2021.

[11] L. B. Deng, C. L. Li , R. Q. Han, L. L. Zhang , L. Y. Qiao, Tpde: a tri-population differential evolution based on zonal-constraint stepped division mechanism and multiple adaptive guided mutation strategies, *Information Sciences,* vol. 575, pp. 22-40, 2021.

[12] L. D. P. Veronese, R. A. Krohling, Differential evolution algorithm on the GPU with C-CUDA, *IEEE Congress on Evolutionary Computation,* 2010, https://doi.org/10.1109/CEC.2010.5586219

[13] W. Zhu, Massively parallel differential evolution - pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization, *Journal of Global Optimization,* vol.50, no.3, pp. 417-437, 2011.

[14] W. Zhu, Y. H. Li, GPU-Accelerated differential evolutionary markov chain monte carlo method for multi-objective optimization over continuous space, *2nd workshop on Bio-inspired algorithms for distributed systems,* pp. 1-8, 2010.

[15] O. A. C. Cortes, S. Pais,N. M. Filipo, Differential evolution on a GPGPU: the influence of parameters on speedup and the quality of solutions, *2015 IEEE International Parallel and Distributed Processing Symposium Workshop,* pp. 299-306, 2015.

[16] M. A. Meselhi, S. M. Elsayed, D. L. Essam, R. A. Sarker, Fast differential evolution for big optimization, *International Conference on Software, Knowledge Information, Industrial Management and Applications,* 2017, https://doi.org/10.1109/SKIMA.2017.8294137

[17] R. Tanabe, A. Fukunaga, Reevaluating exponential crossover in differential evolution, *International Conference on Parallel Problem Solving from Nature.* pp. 201-210, 2014.

[18] R. Z. Zhao, P. N. Suganthan, Empirical investigations into the exponential crossover of differential evolutions, *Swarm and Evolutionary Computation,* vol. 9, pp. 27-36, 2013.

[19] L. Devroye, Complexity questions in non-uniform random variate generation, *19th International Conference on Computational Statistics,* pp. 3-18, 2010.

[20] R.Okuta, Y. Unno, D. Nishino,S. Hido , C. Loomis, CuPy: a numPy-compatible library for NVIDIA GPU calculations, *Workshop on ML Systems at NIPS,* 2017, http://learningsys.org/nips17/assets/papers/paper

[21] K. V. Price, R. M. Storn, J. A. Lampinen, Differential evolution: a practical approach to global optimization, 2005, *Springer-Verlag Berlin Heidelberg.*