An Improved CUDA-Based Implementation of Differential Evolution on GPU

A. K. Qin
INRIA Grenoble Rhone-Alpes
655 avenue de l'Europe, Montbonnot
38334 Saint Ismier Cedex, France
kai.qin@inria.fr

Federico Raimondo
INRIA Grenoble Rhone-Alpes
655 avenue de l'Europe, Montbonnot
8334 Saint Ismier Cedex, France
federaimondo@gmail.com

Florence Forbes
INRIA Grenoble Rhone-Alpes
655 avenue de l'Europe, Montbonnot
8334 Saint Ismier Cedex, France
florence.forbes@inria.fr

Yew Soon Ong
School of Computer Engineering
Nanyang Technological University
Nanyang Avenue, 639798, Singapore
asysong@ntu.edu.sg

ABSTRACT

Modern GPUs enable widely affordable personal computers to carry out massively parallel computation tasks. NVIDIA's CUDA technology provides a wieldy parallel computing platform. Many state-of-the-art algorithms arising from different fields have been redesigned based on CUDA to achieve computational speedup. Differential evolution (DE), as a very promising evolutionary algorithm, is highly suitable for parallelization owing to its data-parallel algorithmic structure. However, most existing CUDA-based DE implementations suffer from excessive low-throughput memory access and less efficient device utilization. This work presents an improved CUDA-based DE to optimize memory and device utilization: several logically-related kernels are combined into one composite kernel to reduce global memory access; kernel execution configuration parameters are automatically determined to maximize device occupancy; streams are employed to enable concurrent kernel execution to maximize device utilization. Experimental results on several numerical problems demonstrate superior computational time efficiency of the proposed method over two recent CUDA-based DE and the sequential DE across varying problem dimensions and algorithmic population sizes.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

Keywords

CUDA, Compute Unified Device Architecture, DE, Differential Evolution, GPU, Graphics Processing Unit, Massively Parallel Computing

1. INTRODUCTION

In past decades, evolutionary algorithms (EAs) [1] have shown remarkable efficacy for solving diverse real-world optimization problems. However, they may expend considerable computation time when handling large-scale and complex tasks. Consequently, EAs are off-limits to a range of applications with demanding computational budgets.

EAs consist of a population of candidate solutions that explore a given solution space using various nature-inspired operations, such as selection, reproduction and replacement, to gradually evolve the population in the quest for global optima. This type of algorithms is inherently parallelizable since population members are typically subjected to same operations. However, a majority of the existing EAs had been designed and implemented in the sequential way because hardware and software platforms that facilitate parallel computing were not widely available and affordable in the past.

In recent years, the graphics processing unit (GPU) has emerged as a powerful computing device that can support general-purpose massively data-parallel computation by means of its hundreds of streaming processors (SPs). Nowadays, with affordable prices and wieldy parallel computing platforms, modern GPUs have empowered numerous personal computers (PCs) the capability of developing parallel applications.

Among existing parallel computing platforms on GPU, NVIDIA's compute unified device architecture (CUDA) [2, 3] provides an intuitive and scalable programming model based on an extended C programming language: CUDA-C. Developers can simply write a C-style routine to process one data element, which then gets automatically distributed across hundreds of SPs for thousands of threads to process different data elements. Due to little efforts for developers already familiar with the C language to grasp CUDA-C, many state-of-the-art algorithms from different scientific and engineering fields have been redesigned based on CUDA to speed up their computation. However, computational time efficiency of CUDA-C applications depends on comprehensive consideration of various technical properties of GPUs during development and implementation. Without delicate consideration, parallel programs written in CUDA-C might even run slower than their sequential counterparts.

Differential evolution (DE) [4], as one of the most promising state-of-the-art EAs, has consistently demonstrated superiority for solving challenging optimization problems. In fact, DE is highly suitable for parallelization owing to its data-parallel algorithmic structure. Although several CUDA-based DE implementations already exist [6-13], their computational time efficiency is hampered by excessive low-throughput global memory access and less efficient device utilization. This work presents an improved CUDA-based DE, called cudaDE$_i$ where the subscript "i" stands for "improved", to enhance memory and device utilization. Major features of cudaDE$_i$ are highlighted as follows:

- Several logically-related kernel functions are unified into one composite kernel function to reduce global memory access by maximizing the use of shared memory.
- Each thread during the kernel execution will process a unique data element to maximize the degree of parallelism.
- Kernel execution configuration parameters are automatically determined to maximize the streaming multiprocessor (SM) occupancy by fully utilizing the available shared memory and registers while considering problem properties and compute capability of GPU.
- Streams are used to enable concurrent kernel execution for the maximization of device utilization.

We evaluate the performance of cudaDE$_i$ on several numerical test problems, and compare cudaDE$_i$ with two recent CUDA-based DE implementations and the sequential DE implemented in the C language in terms of computation time. Experiment results demonstrate the consistent superiority of cudaDE$_i$ across varying problem dimensions and algorithmic population sizes.

The remaining paper is organized as follows. Section 2 introduces GPU computing with CUDA followed by a description of DE and its CUDA-based implementations in Section 3. The improved CUDA-based DE is detailed in Section 4 with experimental results reported and analyzed in Section 5. Finally, Section 6 draws conclusions with some future plans.

2. GPU COMPUTING WITH CUDA

Modern GPUs are no longer exclusively designed for and used in graphics and gaming applications. Additionally, they can provide a powerful and budget parallel computing environment in favor of massively data-parallel applications, i.e. different parts of data are subjected to same operations. Nowadays, common PCs have been widely equipped with GPUs having hundreds of SPs, and thus become highly suitable for developing parallel applications.

Compared to the central processing unit (CPU) that contains one or several sophisticated processors working at high clock speed, GPU consists of hundreds of SPs having simplified structures and working at lower clock speed. Although CPU can rapidly tackle many general-purpose tasks owing to its high clock speed, operation re-scheduling ability and large cache memory, it is less efficient in massively data-parallel applications. In contrast, GPU that operates based on the single instruction multiple data (SIMD) model can allow the simultaneous execution of same operations on many data elements, and thus can lead to high computational speed in data-parallel applications via massive parallelism.

As the graphics hardware advances, its application programming interfaces (APIs) keep being improved to facilitate development. In the early stage of general-purpose GPU computing, developers needed to transform scientific calculations into problems that can be represented by triangles and polygons so as to solve them on GPU using graphics APIs for programming [5]. Nowadays, user-friendly APIs suitable for general-purpose GPU computing have been developed to reduce programming difficulty.

NVIDIA's CUDA technology [2, 3] provides a parallel computing architecture on modern NVIDIA's GPUs in which hundreds of SPs are grouped into several SMs. Each SM contains a number of SPs that share control logic, instruction cache, shared memory with low latency, registers, and so on. All SMs share global memory with high latency. The number of SMs and the size of global memory vary as per GPU models and brands. However, in each SM, the number of SPs, the size of shared memory and the number of registers depend on GPU's compute capability [2].

CUDA also refers to an intuitive and scalable programming model based on an extended C programming language, called CUDA-C [2, 3]. This model unifies CPU and GPU, so-called host and device, into a heterogeneous computing system to make the best advantage out of both of them. Specifically, CUDA-C contains three types of functions: (1) *host functions*, called and executed only by the host, which are exactly the same functions available in the C language; (2) *kernel functions*, only called by the host and executed by the device, which require a qualifier **"__global__"** being declared before the function's return value type that must be "void"; (3) *device functions*, called and executed only by the device, which requires a quantifier "**__device__**" being declared before the function's return value type that can be any types. The sequential operations should be programmed as host functions that are executed on CPU. The parallelizable operations should be programmed as kernel or device functions that are executed on GPU. Both host and kernel functions will be encapsulated and called in a main host function.

In fact, each kernel function will be executed on GPU by a large number of threads at the same time following the SIMD model. In the CUDA programming model, these threads are organized into a grid of thread blocks with each block containing a certain number of threads. The grid can have up to three dimensions of blocks. Its size is denoted by a predefined struct variable *gridDim* with three fields x, y and z storing the block numbers in three dimensions respectively. Each block in a grid can be indexed by a predefined struct variable *blockIdx* with three fields x, y and z storing the position of the corresponding block in the grid. The block can have up to three dimensions of threads with its size denoted by a predefined struct variable *blockDim* having three fields x, y and z storing the thread numbers in three dimensions respectively. Each thread in a block can be indexed by a predefined struct variable *threadIdx* with three fields x, y and z storing the position of the corresponding thread in the block.

The dimension and size of the grid and block usually depend on the characteristics of problems and algorithms. Once a grid of blocks is determined, each block of threads will be executed on one SM. Since each SM allows a maximal number of resident blocks determined by the device's compute capability, once all SMs are fully occupied the remaining blocks have to wait for any available slots released by the completed blocks on any SMs. This scheme ensures the transparent scalability of CUDA-C programs executed on future generations of GPUs that contain more SMs.

When launching a kernel function, its associated kernel execution configuration parameters, such as *gridDim* and *blockDim*, must be specified within "<<<…>>>". When determining configuration parameters, we should consider: each thread block has a maximal number of threads determined by the device's compute capability; each SM has the limited shared memory size and register number, which will influence the allowed number of threads per block and the allowed number of blocks per SM; all threads in one block can access the same data stored on shared memory while threads in different blocks can only communicate via global memory.

3. DIFFERENTIAL EVOLUTION

Differential evolution (DE) [4] emerged as a simple and powerful EA more than a decade ago and has now developed into a very promising research area in the field of evolutionary computation.

3.1 Overview of DE

DE evolves a population of candidate solutions to seek for global optima using three major operations, i.e. mutation, crossover and replacement. The quality of each candidate solution is evaluated as per specific objective functions.

Firstly, a size-fixed population is randomly initialized within the given solution space. Then, each population member, so-called target vector, undergoes three operations in sequence:

- **Mutation**: a base vector is generated from the population, determining the reference point of the mutation. Then, the vector difference of randomly selected population members excluding the target vector under consideration is scaled and added to the base vector to produce a mutant vector.
- **Crossover**: crossover is applied with a certain probability (*CR*) between the above-generated mutant vector and the target vector under consideration to generate a trial vector.
- **Replacement**: if the trial vector has better quality than the target vector under consideration, it will replace the target vector and enter the population of the next generation. Otherwise, the target vector will remain in the population of the next generation.

The population is updated iteratively using these three operations until certain termination criteria are met, e.g. a pre-specified maximal number of function evaluations is reached.

The success of DE attributes to its unique differential mutation scheme, which distinguishes DE from other existing EAs and accordingly coins its name. One most popular DE algorithm "DE/rand/1/bin" randomly selects a population member as the base vector to which the scaled vector difference of two randomly sampled population members is added to generate a mutant vector:

$$\mathbf{V}_i = \mathbf{X}_{r_1^i} + F \cdot \left(\mathbf{X}_{r_2^i} - \mathbf{X}_{r_3^i}\right) \tag{1}$$

where r_1^i, r_2^i and r_3^i are mutually exclusive indices of randomly chosen population members with respect to the target vector $\mathbf{X}_i$. The scale factor F controls the mutation step size. Crossover is then applied between the mutant vector and its corresponding target vector to generate a trial vector $\mathbf{U}_i$**:**

$$u_{i,j} = \begin{cases} v_{i,j}, & \text{if } \operatorname{rand}_j(0,1) \le CR \textit{ or } j = k \\ x_{i,j}, & \text{otherwise} \end{cases} \tag{2}$$

where j is the element index and rand$_j$(0,1) denotes a uniform random number generated in (0,1) with respect to the j^{th} element. The element index k is randomly chosen to ensure that at least one element of the trial vector to come from the mutant vector.

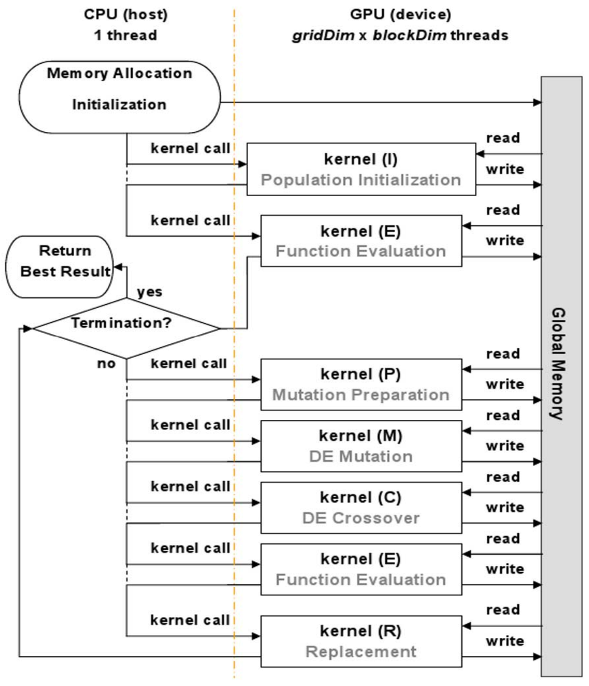

Figure 1. Flow chart of a basic CUDA-based DE.

3.2 CUDA-based DE implementation

A basic CUDA-based DE implementation is illustrated in Figure 1, which encapsulates different DE operations into separate kernel functions that are communicated via global memory. Specifically, there are six kernel functions:

kernel(I): initializes the population and writes it into global memory. This kernel requires random numbers.

kernel(E): evaluates the objective function values of population members as per the problem being solved and writes them into global memory. The evaluation of each population member can take up one or more threads. This kernel needs to read the population to be evaluated from global memory.

kernel(P): prepares the mutually exclusive indices of randomly sampled population members for each target vector to generate the mutant vector. The indices are structured into a matrix and written into global memory. This kernel requires random numbers.

kernel(M): performs the DE mutation to generate mutant vectors, which are then written into global memory. This kernel needs to read the current population from global memory and requires random numbers.

kernel(C): performs the DE crossover to generate trial vectors, which are written into global memory. This kernel needs to read the current population and mutant vectors generated in kernel(M) from global memory and requires random numbers.

kernel(R): performs the objective function values comparison between trial vectors and their corresponding target vectors to form the population of the next generation, which is then written into global memory. This kernel needs to read the objective function values of trial and target vectors, trial vectors and the current population from global memory.

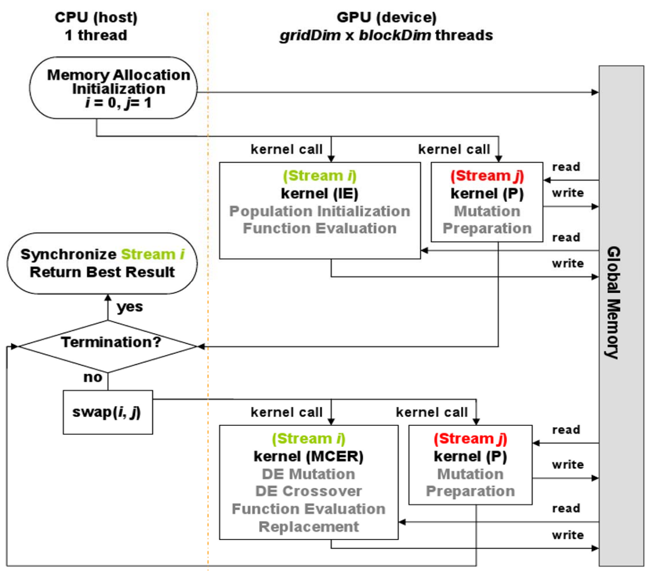

Figure 2. Flow chart of the proposed cudaDE$_i$.

After necessary memory allocation (host and device memories) and initialization (random number generators and algorithmic variables), kernel(I) and kernel (E) are invoked in sequence by the host. Then, the host repeatedly invokes the sequence of kernel(P), kernel(M), kernel(C), kernel(E) and kernel(R) until the main loop terminates.

Although the implementation depicted in Figure 1 is very intuitive, its computational time efficiency can be influenced by excessive global memory access since the main loop consists of several kernels involving low-throughput data transfer via global memory. Furthermore, if the execution configuration parameters of kernels in the main loops result in the low occupancy of SMs during the kernel execution, computational speed can be much reduced.

The first CUDA-based DE (cudaDE) [6] was developed in 2010. Since then, other CUDA-based DE implementations emerged [7-13]. Among them, the fully parallel differential evolution (FPDE) [8] is worthy of note, which had participated in the competition on "GPUs for Genetic and Evolutionary Computation" held at the *2011 Genetic and Evolutionary Computation Conference*. We choose cudaDE and FPDE to compare with our work. The codes of cudaDE and FPDE are provided by the original authors.

In contrast to the implementation in Figure 1, cudaDE does not contain kernel(P) but generates the indices matrix for mutation by the host, and transfers the generated matrix from host memory to device global memory. Moreover, cudaDE combines kernel(I) and kernel(E) for initialization and unifies kernel(M) and kernel(C) into one kernel as well as kernel(E) and kernel(R) into one kernel in the main loop. FPDE mostly conforms to the implementation in Figure 1. Both cudaDE and FPDE use pre-specified kernel execution configuration parameters: cudaDE employs a 1D grid of 240 blocks with each block composed of a 1D array of 64 threads. FPDE uses a 1D grid of N or $\lceil N/16 \rceil$ blocks with each block containing a 1D array of 16 threads. N denotes the population size. FPDE utilizes texture memory in several kernels to accelerate data reading, and uses CURAND library [14] to generate random numbers from the device to overcome the difficulty encountered by cudaDE of generating random numbers in a multithreaded way.

4. IMPROVED CUDA-BASED DE

Existing CUDA-based DE implementations suffer from two major deficiencies:

- *Excessive low-throughput global memory access*: existing CUDA-based DE implementations usually involve several kernel functions performing different DE operations. In fact, most of these kernels rely on results produced by previously launched kernels. However, since global memory is the only data communication media between kernels, results generated by a currently executing kernel have to be written into global memory in order to be used by subsequently launched kernels. When too much such global memory data transfer is invoked in the main loop, the program will become fairly slow.

- *Less-efficient device utilization*: most of the existing CUDA-based DE implementations use pre-specified kernel execution configuration parameters, which are independent of problem properties, e.g. the problem dimension. Consequently, when problem properties do not match the specified configuration, computational speed tends to decrease due to the reduced occupancy of SMs. For example, when each thread block handles a single population member [6], if the problem dimension is much smaller than the block size, SMs will have the very low occupancy since each SM can accommodate a limited number of blocks while only a small fraction of threads are active in each block. On the other hand, if the population dimension is much higher than the block size, some threads in the block have to sequentially handle multiple population member elements, which degrades computational time efficiency.

To address the above issues, we propose an improved CUDA-based DE implementation (cudaDE$_i$) as shown in Figure 2, which consists of only three kernel functions, i.e. kernel(IE), kernel(P), kernel(MCER), and uses CURAND library [14] to generate random numbers. Its major characteristics are as follows:

- Several logically-related kernel functions are unified into one composite kernel to minimize global memory access via the maximal use of shared memory. Specifically, two kernel functions: kernel(I) and kernel(E), as shown in Figure 1, are combined into kernel(IE) shown in Figure 2. Four kernel functions: kernel(M), kernel(C), kernel(E) and kernel(R) are unified into kernel(MCER) shown in Figure 2. Since the latter combination occurs in the main loop, much global memory access can be reduced.

- Kernel execution configuration parameters are automatically determined to maximize the SM occupancy by fully utilizing the available shared memory and registers while considering the device's compute capability. Specifically, each population member element is assigned with a unique thread while the whole population is partitioned into a number of blocks with each block containing an integer number of members. The number of population numbers per block is maximized provided that (1) each thread should reserve enough registers to use; (2) the total shared memory required by a block, calculated according to problem and algorithm characteristics, should not exceed the size of shared memory in SM; (3) the block number and size per SM should not exceed the device's compute capability. The finally determined kernel execution configuration parameters will satisfy all these constraints. Note that configuration parameters derived in the above way will limit the maximal problem dimension. However, higher dimensions can be handled by letting some threads to process more than one population member elements.

- Streams are used to enable concurrent kernel execution to maximize device utilization. Specifically, two streams are created to make the concurrent execution of kernel(MCER) and kernel(P) in the main loop. Two stream indices and two memory space addresses are swapped per generation for kernel(MCER) and kernel(P) to ensure data consistence. For example, after completing an initial execution of kernel(P) in stream0 to generate a mutation indices matrix in global memory addressed by M0, kernel(MCER) is launched in stream0, which uses the generated matrix stored at address M0. At the same time, kernel(P) is launched in stream1 to generate a new mutation indices matrix in global memory addressed by M1. Since the next execution of kernel(MCER) requires a new mutation indices matrix, the next launch of kernel(MCER) will be invoked in steam1 to make use of the generated matrix stored at address M1. Simultaneously, the next launch of kernel(P) will be invoked in stream0 to generate another new mutation indices matrix in global memory addressed by M0.

5. EXPERIMENTAL RESULTS

We evaluate the performance of cudaDE$_i$ and compare cudaDE$_i$ with two recent CUDA-based DE implementations (cudaDE and FPDE) and the sequential DE implementation (sDE) in terms of computation time using four numerical problems of 10D, 50D and 100D respectively, where "D" denotes the dimension. Different algorithmic population sizes (P50, P100, P500 and P1000) are examined to reveal their impacts on computational time efficiency.

5.1 Experimental Setup

Experiments are conducted on a PC equipped with an Intel XEON E5410 CPU at 2.33 Ghz and a NVIDIA GTX560 GPU with 1GB of GDDR5 global memory. GTX560 supports compute capability 2.1, which has 336 SPs evenly deployed in seven SMs, i.e. each SM consists of 48 SPs. Our development environment is made of Ubuntu 11.10 operating system, CUDA toolkit 4.0, NVIDIA driver version 290.10.

All of four DE implementations in comparison, i.e. sDE, cudaDE$_i$, cudaDE and FPDE, are based on the "DE/rand1/bin" algorithm with parameters set as: $CR = 0.3$, $F = 0.5$. Population sizes are set to 50, 100, 500 and 1000 respectively for each test case.

For each test problem of a specific dimension, each DE implementation under a specific parameter setting is executed 25 times starting from different random states while all of four implementations share the same initial random state for any individual run. The algorithm terminates once the maximal number of function evaluations is reached, which is set to 10^4 times the problem dimension, e.g. for a 10D problem, the maximal number of function evaluations is 10^5.

5.2 Test Problems

We choose four numerical problems [15] for testing:

F$_1$: Shifted sphere function

$$f_1(\mathbf{x}) = \sum_{i=1}^{D} z_i^2, \mathbf{z} = \mathbf{x} - \mathbf{o}$$

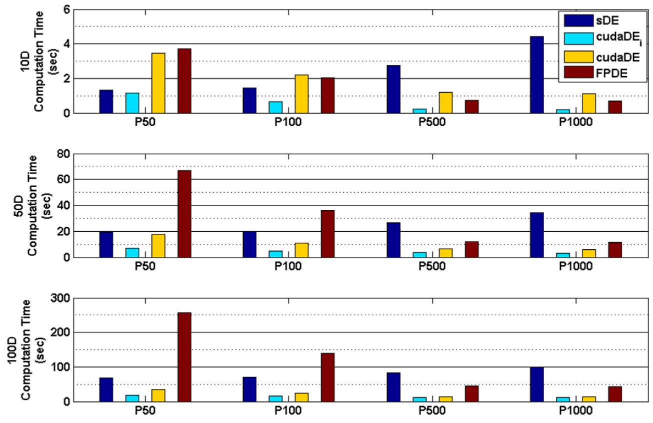

Figure 3. Comparison of the average computation time (seconds) over 25 runs of four DE implementations (sDE, cudaDE$_i$, cudaDE and FPDE) across varying population sizes (P50, P100, P500 and P1000) on test problem F$_1$ of different dimensions (10D, 50D and 100D).

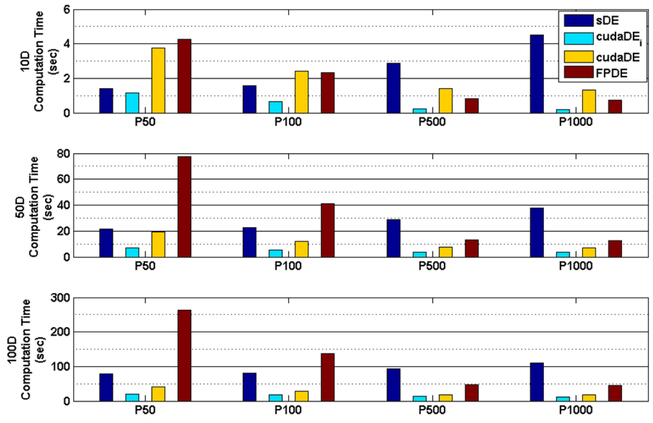

Figure 4. Comparison of the average computation time (seconds) over 25 runs of four DE implementations (sDE, cudaDE$_i$, cudaDE and FPDE) across varying population sizes (P50, P100, P500 and P1000) on test problem F$_2$ of different dimensions (10D, 50D and 100D).

F$_2$: Shifted Rosenbrock's function

$$f_2(\mathbf{x}) = \sum_{i=1}^{D-1} \left(100 \cdot \left(z_i^2 - z_{i+1}\right)^2 + \left(z_i - 1\right)^2\right), \mathbf{z} = \mathbf{x} - \mathbf{o}$$

F$_3$: Shifted Griewank's function

$$f_3(\mathbf{x}) = \sum_{i=1}^{D} \frac{z_i^2}{4000} - \prod_{i=1}^{D} \cos\left(\frac{z_i}{\sqrt{i}}\right) + 1, \mathbf{z} = \mathbf{x} - \mathbf{o}$$

F$_4$: Shifted Rastrigin's function

$$f_4(\mathbf{x}) = \sum_{i=1}^{D} \left(z_i^2 - 10 \cdot \cos(2\pi z_i) + 10\right), \mathbf{z} = \mathbf{x} - \mathbf{o}$$

Here, $\mathbf{o} = [o_1,\ldots,o_D]$ is the shifted global optimum. For all of three CUDA-based DE implementations in comparison, the function evaluation routines are programmed in a parallel manner, i.e. the objective function values are calculated using multiple threads with each thread processing function components corresponding to one or more elements of one or more population members.

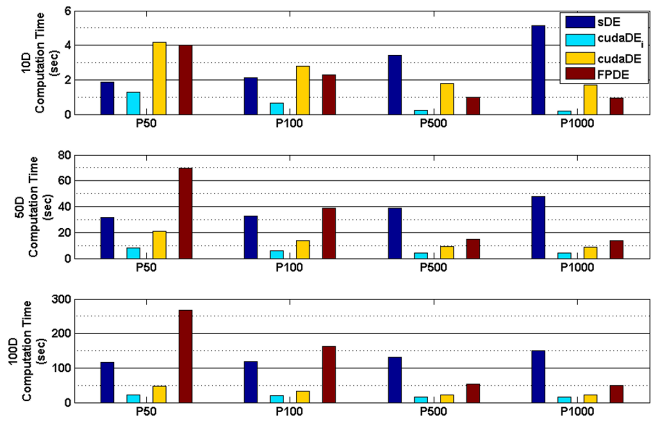

Figure 5. Comparison of the average computation time (seconds) over 25 runs of four DE implementations (sDE, cudaDE_i, cudaDE and FPDE) across varying population sizes (P50, P100, P500 and P1000) on test problem F_3 of different dimensions (10D, 50D and 100D).

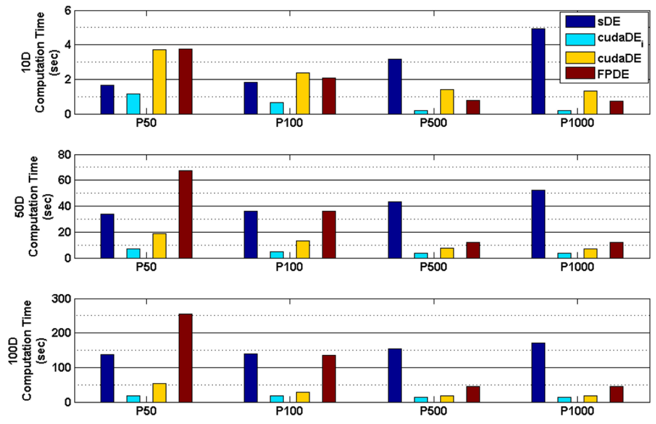

Figure 6. Comparison of the average computation time (seconds) over 25 runs of four DE implementations (sDE, cudaDE_i, cudaDE and FPDE) across varying population sizes (P50, P100, P500 and P1000) on test problem F_4 of different dimensions (10D, 50D and 100D).

5.3 Results

We evaluate and compare computational time efficiency of four DE implementations, i.e. sDE, cudaDE_i, cudaDE and FPDE, using four numerical problems under different problem dimensions and algorithmic population sizes.

Figures 3~6 illustrate, for each of four test problems, the average computation time over 25 runs of four DE implementations across varying algorithmic population sizes (P50, P100, P500 and P1000) and under different problem dimensions (10D, 50D and 100D). Major observations are as follows:

(1) Different test problems reveal similar comparison results with magnitudes of computation time depending on problem complexity.

(2) For all of test problems, cudaDE_i consistently demonstrates superior computational time efficiency with respect to any population sizes and any problem dimensions under test.

(3) Given any specific problem dimension, as the population size increases, computation time of three CUDA-based DE implementations decreases. Among the three, FPDE shows the most remarkable decrease, followed by cudaDE and cudaDE_i.

(4) When solving 10D problems using smaller population sizes, e.g. P50 and P100, both cudaDE and FPDE perform worse than sDE while cudaDE_i outperforms sDE. When the problem dimension increases, cudaDE starts to outperform sDE. However, FPDE can beat sDE only when using larger population sizes, e.g. P500 and P1000.

(5) When solving 100D problems using larger population sizes, e.g. P500 and P1000, the difference of computation time between cudaDE_i and cudaDE is less significant.

Given a specific problem dimension that corresponds to a specific maximal number of function evaluations, the population size determines the total number generations (main loops) and thus the execution times of those kernels involving low-throughput global memory access in the main loop. The more such kernels exist, the more computation time tends to increase with increasing the total number of generations via decreasing the population size. Since FPDE contains the most such kernels while cudaDE_i has the least ones, varying population sizes will influence most on FPDE while least on cudaDE_i, which explains observation (3).

Given pre-specified kernel execution configuration parameters, both problem dimensions and population sizes may influence the degree of parallelism and consequently impact computation time. Furthermore, problem dimensions may also significantly influence the SM occupancy. Specifically, cudaDE specifies 240 thread blocks with each block having 64 threads to cope with a single population member. When the problem dimension exceeds 64, some threads within one block have to sequentially process several population member elements. When the population size is over 240, some blocks have to process several population members in sequence. Such a configuration can handle any problem dimensions and population sizes at the expense of the loss of parallelism. Moreover, it allows at most 512 threads per SM due to the limitation of eight blocks per SM, which results in the maximum of 33% SM occupancy. When solving 10D problems, the SM occupancy will reduce to 5%, which explains observation (4): using smaller population sizes, e.g. P50 and P100, cudaDE will execute kernels involving low-throughput global memory access in the main loop for considerable times under such low SM occupancy, which causes its computation time longer than sDE. However, as the problem dimension increases, e.g. 50D and 100D, cudaDE operates under the much increased SM occupancy and thus outperforms sDE.

FPDE encounters similar issues. For kernels related to algorithmic operations, FPDE uses $\lceil N/16 \rceil$ thread blocks with each block having 16 threads to handle at least 16 population members. For the function evaluation kernel, FPDE uses N thread blocks with each block containing 16 threads to process at least 16 elements of one or more population members. N denotes the population size. Such configurations may significantly reduce the degree of parallelism. Meanwhile, they can allow at most 128 threads per SM due to the limitation of eight blocks per SM, which leads to the maximum of 8% SM occupancy. When solving 10D problems, FPDE will operate under the 5% SM occupancy. When solving 50D and 100D problems, FPDE will operate under the 8% SM occupancy. Due to operating under the very low SM occupancy and involving many kernels that require low-throughput global memory access in the main loop, FPDE cannot beat sDE unless using larger population sizes, e.g. P500 and P1000, which can significantly reduce the maximal number of generations. This fact explains observation (4).

cudaDE$_i$ needs not to pre-specify kernel execution configuration parameters. Instead, it can automatically determine configuration parameters to maximize the SM occupancy by fully utilizing the available shared memory and registers while satisfying various constraints detailed in Section 4. Therefore, cudaDE$_i$ can operate under the high SM occupancy for any problem dimensions and population sizes. Furthermore, cudaDE$_i$ consists of less kernels requiring low-throughput global memory access in the main loop. These benefits explain observation (2).

cudaDE$_i$ tends to maximize the block size and consequently may decrease the maximal number of active blocks per SM. When solving 100D problems using larger population sizes, both cudaDE and cudaDE$_i$ operate under the desirable SM occupancy while cudaDE has eight active blocks per SM and cudaDE$_i$ has one active block per SM. Although cudaDE$_i$ involves less kernels requiring low-throughput global memory access in the main loop than cudaDE, a large block may perform less time-efficiently than several small blocks containing the same number of threads in total when the kernel involves thread synchronization operations. This fact explains observation (5) and deserves further study.

Tables 1~3 elaborate the performance of the proposed cudaDE$_i$, measured by: (1) the mean value and standard deviation of the best error function values (EFVs), i.e. the difference of objective function values between the best solution found so far and the global optimum, achieved when the algorithm terminates over 25 runs; (2) the average computation time (seconds) over 25 runs; (3) the success rate (SR) over 25 runs. One run is claimed to succeed once it achieves an EFV smaller than 10^{-8}. It can be observed that larger population sizes may not definitely result in better solution quality although leading to less computation time.

6. CONCLUSIONS AND FUTURE WORK

We proposed an improved CUDA-based DE implementation to optimize memory and device utilization. Several logically-related kernels are unified into one composite kernel to minimize global memory access by maximizing the use of shared memory. Kernel execution configuration parameters are automatically determined to maximize the SM occupancy by fully utilizing available shared memory and registers while considering problem properties and compute capability of GPU. Streams are employed to enable concurrent kernel execution to maximize device utilization. The consistent superiority of the proposed cudaDE$_i$ over two recent CUDA-based DE and the sequential DE in terms of computational time efficiency is verified using several numerical test problems across varying problem dimensions and algorithmic population sizes.

Ongoing and planned research agendas include: further improving cudaDE$_i$ by studying the relation between computation time and the number of active blocks per SM given the high occupancy of SMs; enabling cudaDE$_i$ to solve problems of more than 1000D; evaluating cudaDE$_i$ on complex and high-dimensional problems as well as real-world applications.

Table 1. Performance of cudaDE$_i$ in terms of the mean value and standard deviation (in bracket) of the best EFVs achieved when the algorithm terminates, the mean value and standard deviation of computation time (seconds) as well as the success rate over 25 runs with respect to four population sizes (P50, P100, P500 and P1000) on four 10D test problems.

		P50	P100	P500	P1000
F_1	Best EFV	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)
	Time (sec)	1.159 (0.002)	0.633 (0.001)	0.228 (0.000)	0.192 (0.000)
	SR	1.00	1.00	1.00	1.00
F_2	Best EFV	0.388 (1.235)	0.044 (0.068)	1.306 (1.388)	2.597 (1.847)
	Time (sec)	1.159 (0.003)	0.632 (0.001)	0.229 (0.000)	0.196 (0.000)
	SR	0.44	0.32	0.00	0.00
F_3	Best EFV	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)
	Time (sec)	1.261 (0.002)	0.648 (0.002)	0.222 (0.000)	0.210 (0.000)
	SR	1.00	1.00	1.00	1.00
F_4	Best EFV	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)
	Time (sec)	1.158 (0.003)	0.635 (0.001)	0.207 (0.000)	0.192 ((0.001)
	SR	1.00	1.00	1.00	1.00

Table 2. Performance of cudaDE$_i$ in terms of the mean value and standard deviation (in bracket) of the best EFVs achieved when the algorithm terminates, the mean value and standard deviation of computation time (seconds) as well as the success rate over 25 runs with respect to four population sizes (P50, P100, P500 and P1000) on four 50D test problems.

		P50	P100	P500	P1000
F_1	Best EFV	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)
	Time (sec)	6.881 (0.011)	4.937 (0.004)	3.453 (0.001)	3.348 (0.001)
	SR	1.00	1.00	1.00	1.00
F_2	Best EFV	213.544 (625.959)	12.244 (7.100)	46.968 (19.274)	56.365 (24.123)
	Time (sec)	6.940 (0.009)	5.055 (0.004)	3.549 (0.005)	3.443 (0.007)
	SR	0.00	0.04	0.00	0.00
F_3	Best EFV	0.001 (0.002)	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)
	Time (sec)	8.142 (0.009)	5.902 (0.005)	4.349 (0.005)	4.282 (0.001)
	SR	0.92	1.00	1.00	1.00
F_4	Best EFV	18.286 (3.97)	2.936 (1.472)	28.345 (14.084)	65.628 (8.607)
	Time (sec)	7.006 (0.010)	4.914 (0.004)	3.851 (0.007)	3.692 (0.002)
	SR	0.00	0.04	0.04	0.00

Table 3. Performance of cudaDE$_i$ in terms of the mean value and standard deviation (in bracket) of the best EFVs achieved when the algorithm terminates, the mean value and standard deviation of computation time (seconds) as well as the success rate over 25 runs with respect to four population sizes (P50, P100, P500 and P1000) on four 100D test problems.

		P50	P100	P500	P1000
F$_1$	**Best EFV**	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)	0.000 (0.000)
	Time (sec)	18.624 (0.013)	16.396 (0.008)	12.289 (0.003)	11.557 (0.002)
	SR	1.00	1.00	1.00	1.00
F$_2$	**Best EFV**	15.720 (24.466)	42.293 (32.790)	113.292 (22.349)	118.43 (33.239)
	Time (sec)	19.003 (0.012)	16.919 (0.008)	12.679 (0.002)	11.917 (0.003)
	SR	0.00	0.00	0.00	0.00
F$_3$	**Best EFV**	0.002 (0.006)	0.005 (0.027)	0.000 (0.002)	0.000 (0.000)
	Time (sec)	22.376 (0.012)	20.929 (0.011)	15.784 (0.003)	14.794 (0.002)
	SR	0.84	0.96	0.96	1.00
F$_4$	**Best EFV**	63.166 (8.984)	48.329 (9.172)	40.891 (8.539)	114.234 (47.738)
	Time (sec)	18.862 (0.032)	17.417 (0.020)	13.236 (0.038)	12.644 (0.015)
	SR	0.00	0.00	0.00	0.00

7. ACKNOWLEDGMENTS

We thank authors of cudaDE and FPDE for providing the codes. This work was partially supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61005051, the Specialized Research Fund for the Doctoral Program of Higher Education (SRFDP) under Grant No. 20100092120027, and the Media Development Authority of Singapore, Singapore-MIT GAMBIT Game Lab.

8. REFERENCES

[1] De Jong, K. A. *Evolutionary Computation: A Unified Approach*. The MIT Press, 2006.

[2] *NVIDIA CUDA C Programming Guide Version 4.0*. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[3] Kirk D., and Hwu, W.-M. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[4] Price, K., Storn, R., and Lampinen, J. *Differential Evolution: A Practical Approach to Global Optimization*. Springer-Verlag, Berlin, Germany, 2005.

[5] Fok, K.-L., Wong, T. T., and Wong, M.-L. Evolutionary computing on consumer-level graphics hardware. *IEEE Intelligent Systems*, 22, 2, 2007, 69-78.

[6] Veronese, L. D. P., and Krohling, R. A. Differential evolution algorithm on the GPU with C-CUDA. In *Proc. of the 2010 IEEE Congress on Evolutionary Computation (CEC'2010)*, Barcelona, Spain, July 18-23, 2010.

[7] Zhu, W., and Li, Y. GPU-accelerated differential evolutionary Markov Chain Monte Carlo method for multi-objective optimization over continuous space. In *Proc. of the 2nd Workshop on Bio-inspired Algorithms for Distributed Systems*, New York, NY, USA, June 7-11, 2010.

[8] Domínguez González, S. J., and Barriga, N. G. Fully parallel differential evolution. In *Competition of GPUs for Genetic and Evolutionary Computation at the 2011 Genetic and Evolutionary Computation Conference (GECCO'2011)*, Dublin, Ireland, July, 2011.

[9] Kromer, P., Platos, J., Snasel, V., and Abraham, A. A comparison of many-threaded differential evolution and genetic algorithms on CUDA. In *Proc. of the 2011 World Congress on Nature and Biologically Inspired Computing (NaBIC'2011)*, Salamanca, October 19-21, 2011.

[10] Kromer, P., Platos, J., and Snasel, V. Differential evolution for the linear ordering problem implemented on CUDA. In *Proc. of the 2011 IEEE Congress on Evolutionary Computation (CEC'2011)*, New Orleans, LA, USA, June 05-08, 2011.

[11] Fabris, F., and Krohling, R. A. A co-evolutionary differential evolution algorithm for solving min–max optimization problems implemented on GPU using C-CUDA. *Expert Systems with Applications*, available online, 2011. DOI = http://dx.doi.org/10.1016/j.eswa.2011.10.015.

[12] Ramirez-Chavez, L. E., Coello Coello, C. A., and Rodriguez-Tello, E. A GPU-based implementation of differential evolution for solving the gene regulatory network model inference problem. In *Proc. of the 4th International Workshop on Parallel Architectures and Bioinspired Algorithms (WPABA'2011)*, Galveston Island, TX, USA, October 10-14, 2011.

[13] Krömer, P., Snåšel, V., Platoš, J., and Abraham, A. Many-threaded implementation of differential evolution for the CUDA platform. In *Proc. of the 2011 Genetic and Evolutionary Computation Conference (GECCO'2011)*, Dublin, Ireland, July, 2011.

[14] *CURAND Library User Guide*. http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CURAND_Library.pdf.

[15] Suganthan, P. N., Hansen, N., Liang, J. J., Deb, K., Chen, Y.-P., Auger A., and Tiwari, S. Problem definitions and evaluation criteria for the CEC 2005 special session on real parameter optimization. *Technical Report*, Nanyang Technological University, Singapore, May 2005 and KanGAL Report #2005005, IIT Kanpur, India.