Attract-Repulse Fireworks Algorithm and its CUDA Implementation Using Dynamic Parallelism

Ke Ding and Ying Tan

Abstract—**Fireworks Algorithm (FWA) is a recently developed Swarm Intelligence Algorithm (SIA), which has been successfully used in diverse domains. When applied to complicated problems, many function evaluations are needed to obtain an acceptable solution. To address this critical issue, a GPU-based variant (GPU-FWA) was proposed to greatly accelerate the optimization procedure of FWA. Thanks to the active studies on FWA and GPU computing, many advances have been achieved since GPU-FWA. In this paper, a novel GPU-based FWA variant, Attract-Repulse FWA (AR-FWA), is proposed. AR-FWA introduces an efficient adaptive search mechanism (AFW Search) and a nonuniform mutation strategy for spark generation. Compared to the state-of-the-art FWA variants, AR-FWA can greatly improve the performance on complicated multimodal problems. Leveraging the edge-cutting dynamic parallelism mechanism provided by CUDA, AR-FWA can be implemented on the GPU easily and efficiently.**

Index Terms—**Fireworks Algorithm (FWA), Swarm Intelligence Algorithms (SIAs), GPU Computing, Compute Unified Device Architecture (CUDA), Dynamic Parallelism**

I. INTRODUCTION

Fireworks Algorithm (FWA) is a novel swarm intelligence algorithm (SIA) under active research. Inspired by the explosion process of fireworks, FWA was originally proposed for solving optimization problems [1]. Comparative study shows that FWA is very competitive with respect to real-parameter problems [2]. FWA has been successfully applied to many scientific and engineering problems, such as non-negative matrix factorization [3], digital filter design [4], parameter optimization [5], document clustering [6], and so forth. New mechanisms and analyses are actively proposed to further improve the performance of FWA [7], [8].

Although FWA, as well as other SIAs, has achieved success in solving many real-world problems where conventional arithmetic and numerical methods fail, it suffers from the drawback of intensive computation which greatly limits its applications where function evaluation is time-consuming.

Facing technical challenges with higher clock speeds in fixed power envelope, modern computer systems increasingly depend on adding multiple cores to improve the performance [9]. Initially designed for addressing highly computational graphics tasks, the Graphics Processing Unit (GPU), from its inception, has many computational cores and can provide massive parallelism (with thousands of cores) at a reasonable price. As the hardware and software for GPU programming grow mature [10], [11], GPUs have become popular for general purpose computing beyond the field of graphics processing, and great success has been achieved in various applications, from embedded systems to high performance supercomputers [12], [13], [14].

Based on interactions within population, SIAs are naturally amenable to parallelism. SIAs' such intrinsic property makes them very suitable to run on the GPU in parallel, thus gain remarkable performance improvement. In effect, GPUs have been utilized to accelerated SIAs from the first days of GPU computing, and significant progress has been achieved along with the emergence of high-level programming platforms such as CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) [15], [16]. In the past few years, different implementations of diverse SIAs were proposed. Targeting on GPUs of various architectures and specifications, many techniques and tricks were introduced [17].

The first GPU-based FWA, named GPU-FWA, was proposed in 2013 which targets on GPUs of Fermi Architecture [18]. GPU-FWA modifies the original FWA to suit the particular architecture of the GPU. It does not need special complicated data structure, thus making it easy to implement; meanwhile, it can make full use of the great computing power of GPUs. In the last few years, however, many advances have been achieved for both FWA and GPU computing. More dedicated and efficient implementations are possible.

In this paper, a novel GPU-based FWA variant, Attract-Repulse FWA (AR-FWA), is proposed and discussed in detail. AR-FWA introduces an efficient adaptive firework search strategy and a novel mutation mechanism for spark generation. Thanks to the dynamic parallelism provided by CUDA, AR-FWA can be implemented on the GPU very easily and efficiently.

The remainder of this paper is organised as follows. In Section II, the related work, Fireworks Algorithm (FWA) and General Purpose Computing on the GPU (GPGPU), is presented briefly. Section III discusses the proposed algorithm, Attract-Repulse Fireworks Algorithm (AR-FWA). The Adaptive Firework Search (AFW Search) and Non-uniform Mutation are presented in detail. Section IV describes how AR-FWA can be implemented on the GPU using dynamic parallelism. Key kernel codes are also given out in this section. The experiments and analyses are given in Section V. The performance of Non-uniform mutation against uniform mutation is studied, as well as AR-FWA against the state-of-the-art FWA variants and the speedup on the basis of extensive experiments. Finally, we conclude the discussion in Section

The authors are with the Key Laboratory of Machine Perception (MOE), Peking University and Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871 China (E-mail: {keding,ytan}@pku.edu.cn.)

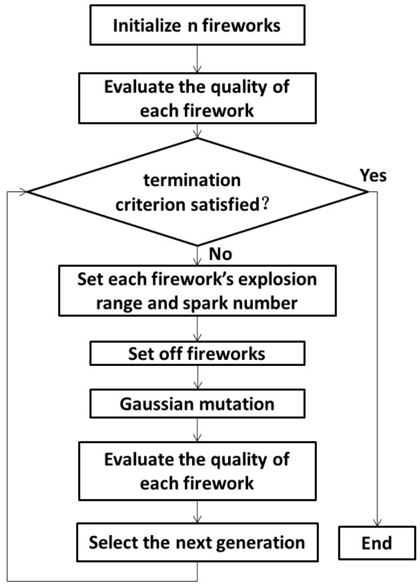


Fig. 1: Framework of Fireworks Algorithm

VI.

II. Related Work

A. Fireworks Algorithm (FWA)

The framework of FWA is illustrated by Fig. 1. FWA utilizes n D-dimensional parameter vectors $\mathbf{x}_i^G$ as the basic population in each generation. Parameter i varied from 1 to n and parameter G stands for the index of generations. Every individual in the population explodes and generates sparks around it. The number of sparks and the amplitude of each individual are determined by certain strategies. Furthermore, a Gaussian explosion is used to generate sparks to keep the diversity of the population. Finally, the algorithm keeps the best individual in the population and selects the rest $n-1$ individuals based on distance for next generation. More specific strategies of fireworks algorithm are described as follows [2].

1) Spark Generation: Spark generation mimics the explosion of fireworks and is the core mechanism in fireworks algorithm. When a firework blasts, many sparks appear around it. The explosion sparks strategy mimicking this phenomenon is used to produce new individuals by explosion. In this strategy, two parameters need to be determined.

The first one is the number of sparks:

$$s_i = \hat{S} \cdot \frac{y_{max} - f(\mathbf{x}_i) + \xi}{\sum_{i=1}^{N}(y_{max} - f(\mathbf{x}_i)) + n \cdot \xi}. \tag{1}$$

where $\hat{S}$ is a parameter controlling the total number of sparks generated by the n fireworks, $y_{max} = max(f(\mathbf{x}_i))\,(i = 1, 2, \ldots, n)$ is the maximum (worst) fitness value of the objective function among the n fireworks, and ξ denotes the machine precision. s_i is rounded to the nearest integer (clamped if beyond a predefined range). (Note that, in the original literature [1] and many following works, the ξ in the denominator is not multiplied by n which will cause the sum of all A surpasses $\hat{A}$ when the finesses are very close. The same argument holds for Eq. (2) as well).

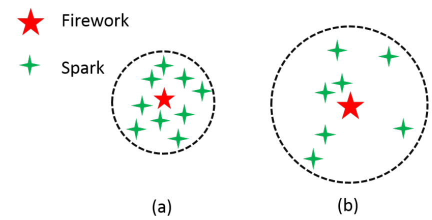


Fig. 2: A firework with better fitness value can generate a larger population of sparks within a smaller range (a), and vice verse (b).

The second parameter in this strategy is the amplitude of sparks:

$$A_i = \hat{A} \cdot \left(\frac{f(\mathbf{x}_i) - y_{min} + \xi}{\sum_{i=1}^{N}(f(\mathbf{x}_i) - y_{min}) + n \cdot \xi} + \Delta \right). \tag{2}$$

where the predefined $\hat{A}$ denotes the maximum explosion amplitude, and $y_{min} = min(f(\mathbf{x}_i))\,(i = 1, 2, \ldots, n)$ i.e. the minimum (best) value of the objective function among the n fireworks, and ξ , which denotes the machine precision. Δ is a small number to guarantee the amplitude is nonzero thus avoid the search process getting stalled. In [19], a minimum amplitude check is conducted instead of using Δ.

Eq. (1) and Eq. (2) guarantee that fireworks with better fitness values generate more sparks within smaller range (cf. Fig. 2). Via this mechanism, more computing resource can be assigned to better space to enhance exploitation, and for the worse space, the search trends to exploration.

2) Mapping Strategy: If an individual is close to the boundary, the generated sparks may lie beyond the feasible space. Therefore, some mapping method is needed to ensure that all the individuals stay in the feasible space. If there are some outlying sparks from the boundary, they will be mapped to their allowable scopes.

A common strategy is mapping by clamping:

$$\mathbf{x}_i = \mathbf{x}_i^{min} + |\mathbf{x}_i - \mathbf{x}_i^{min}| \% (\mathbf{x}_i^{max} - \mathbf{x}_i^{min}). \tag{3}$$

where $\mathbf{x}_i$ represents the positions of any sparks on the i-th dimension that lie out of bounds, while $\mathbf{x}_i^{max}$ and $\mathbf{x}_i^{min}$ stand for the upper and lower limits of a spark position. The symbol $\%$ stands for the modular arithmetic operation.

Another mapping strategy is proposed in [19]. In this strategy (cf. Eq. (4)), the value is clamped near the bound randomly to enhance the search near boundaries.

$$\mathbf{x}_i = \begin{cases} \mathbf{x}_i^{max} - 0.2 * (\mathbf{x}_i^{max} - \mathbf{x}_i^{min}) & , if\ \mathbf{x}_i > \mathbf{x}_i^{max}, \\ \mathbf{x}_i^{min} + 0.2 * (\mathbf{x}_i^{max} - \mathbf{x}_i^{min}) & , if\ \mathbf{x}_i < \mathbf{x}_i^{min}, \\ \mathbf{x}_i & , otherwise. \end{cases} \tag{4}$$

3) Gaussian Mutation: Aside from the ordinary spark generation, another way to generate sparks is proposed as Gaussian mutation. Gaussian mutation is used to generate sparks with Gaussian distribution in order to keep the diversity of the population. Suppose the position of current individual is stated as $\mathbf{x}_k^j$, the Gaussian explosion sparks are calculated as:

$$\mathbf{x}_k^j = \mathbf{x}_k^j \cdot g. \tag{5}$$

where g is a random number in Gaussian distribution:

$$g = Gaussian(1, 1). \tag{6}$$

Parameter g obeys the Gaussian distribution with both mean value and standard deviation being 1.

4) Selection Strategy: After normal explosions and Gaussian explosions, a selection procedure is conducted to keep n individuals for next generation. In the original literature [1], a distance based selection method was suggested. In the selection strategy, the distances between individuals need to be calculated, which is very time-consuming.

To tackle this issue, a more efficient selection strategy, named Elitism Random Selection (ERS), was proposed and widely adopted in the following works [20]. In ERS, the best individual is always preserved, while the other $n-1$ individuals are selected randomly. In this way, the running time for FWA is largely decreased.

A detailed discussion on selection strategy can be found in [2] and [19] .

B. General-Purpose Computing on GPUs (GPGPU)

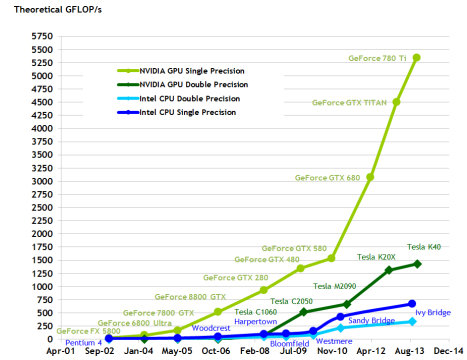
Driven by the insatiable demand for real-time high-definition graphics, GPUs have evolved into highly parallel, many-core processors and are able to execute tens of hundreds threads concurrently. Today's GPUs greatly outperform CPUs in both arithmetic throughput and memory bandwidth (cf. Fig. 3). GPUs can offer great performance at a very low price, and meanwhile GPUs can also be integrated into High Performance Computing (HPC) systems without much difficulty [21], [22]. Moreover, GPUs also have great performance/watt, which is key for achieving super computing performance. In the latest (as of April 2015) Green500 list [1], nine of the top 10 systems on the Green500 are accelerated with GPUs. Much effort has been made to harness the enormous power of GPUs for general-purpose computing, and a great success has been achieved.

Many platforms and programming models have been proposed for GPU computing, of which the most important platforms are CUDA (Compute Unified Device Architecture) [23] and OpenCL [24]. Both platforms are based on C language and share very similar platform model, execution model, memory model and programming model.

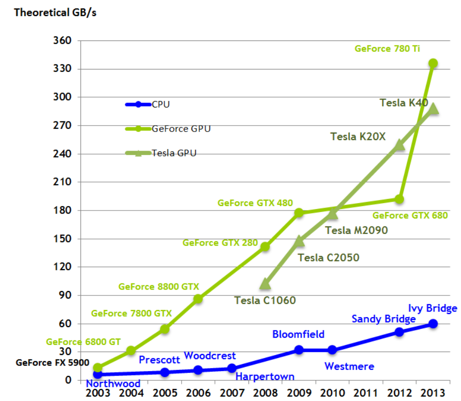
CUDA, a proprietary architecture from NVIDIA, enjoys the advantage of mature ecosystem and it is very easy to use as far as programming procedure is concerned. CUDA comes with a software environment that allows developers to use C as a high-level programming language, thus makes it easier for programmers to fully exploit the parallel feature of GPUs without an explicit familiarity with the GPU architecture [23].

In CUDA programming, GPU computing is conducted by kernels. A kernel is a function that explicitly specifies data parallel computations to be executed on GPUs. When a kernel is launched on the GPU, it is executed by a batch of threads. Threads are organized into independent blocks, and blocks in turn constitute a grid. Closely related to CUDA's thread hierarchy is its memory model. CUDA threads may access data from multiple memory spaces during their execution as illustrated by Fig. 4. Each thread has private registers and local memory. Each thread block has shared memory visible to all threads of the block. All threads have access to the same global memory. Register and shared memory are very fast on-chip memory while global memory is off-chip and has very long access latency.

[1] http://green500.org/lists/green201411



(a) Floating-Point Operations per Second for the CPU and GPU



(b) Memory Bandwidth for the CPU and GPU

Fig. 3: GPUs greatly outperform CPUs in both arithmetic throughput and memory bandwidth [23].

.

III. Attract-Repulse Fireworks Algorithm (AR-FWA)

In this section, the algorithm will be described in detail. We leave the discussion about the GPU-based implementation in

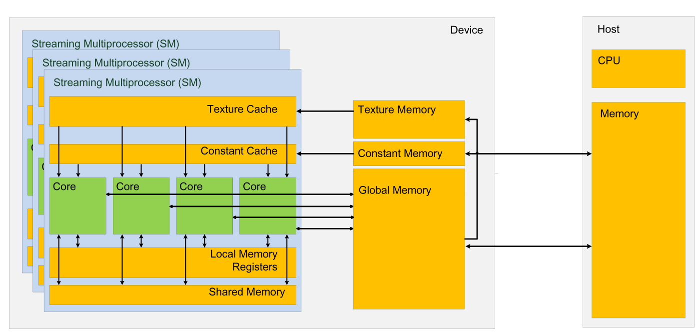


Fig. 4: Memory Model of CUDA

Algorithm 1 AR-FWA

Initialize N fireworks
while terminated conditions not satisfied **do**
  Calculate the fitness values of each firework
  Calculate s according to Eq. (1)
  Calculate A according to Eq. (2)
  for $i = 1$ to n **do**
    Search according to Algorithm 2
  end for
  Mutate according to Algorithm 4
end while

Algorithm 2 Adaptive Firework Search

For the k-th spark
for $i = 1$ to L **do**
  Generate s_k sparks according to Algorithm 3
  Evaluate the fitness
  Find the best spark, and replace it with the firework if better
  if firework is updated **then**
    $A = A * \alpha$
  else
    $A = A * \beta$
  end if
end for

Algorithm 3 Spark Generation

Initialize the location: $\hat{\mathbf{x}} = \mathbf{x}$;
for $i = 1$ to D **do**
  $r = uniform(0, 1)$;
  if $r < \frac{1}{2}$ **then**
    $\hat{\mathbf{x}}_i = \hat{\mathbf{x}}_i + A \cdot RNG(\cdot)$;
  end if
end for

Algorithm 4 Attract-Repulse Mutation

For the k-th firework
Initialize the new location: $\hat{\mathbf{x}} = \mathbf{x}$;
for $d = 1$ to D **do**
  $r = rand(0, 1)$;
  if $r < \frac{1}{2}$ **then**
    $s = U(1 - \delta, 1 + \delta)$;
    $\hat{\mathbf{x}}_d = \hat{\mathbf{x}}_d + (\hat{\mathbf{x}}_d - \mathbf{x}_{best,d}) \cdot s$;
  end if
end for

the next section.

The basic procedure of AR-FWA is depicted by Algorithm 1. In the remainder of this section, each component will be discussed respectively.

A. Adaptive Firework Search (AFW Search)

In [18], a mechanism called Firework Search (FW Search) is suggested for efficient local search. In FW search, each firework generates a fixed number of sparks and the exact number of sparks is determined in accordance with the specific GPU hardware. It was argued that, this fixed encoding of firework explosion is more suitable for parallel implementation on GPUs. However, as the GPU architecture has evolved a lot since then, the argument is not necessarily true any more. In AFW search, the number of sparks is determine dynamically according to Eq. (1). In Section IV, we will see how this can be implemented efficiently using the novel dynamic parallelism mechanism.

One of the key parameter in AFW search (as well as FW search in [18]) is the explosion amplitude determined by Eq. (2). Recently, the adaptive amplitude controlling is been actively discussed. Many proposals have been come up with to adjust the amplitude dynamically according to the history information [7], [8]. Among these proposals, Zheng et al. suggest a decent strategy for dynamic search for FWA [7]. In their proposal, the core firework (i.e. the current best firework) uses a dynamic explosion amplitude for the firework at the currently best position. If the fitness of the best firework is improved, the explosion amplitude increases in order to speed up convergence. On the contrary, if the current position of the best firework is not be improved, the explosion amplitude decreases to narrow the search area. Experiments show that by using the dynamic strategy, the performance can be greatly improved. Based on this insight, we apply the dynamic strategy for all of the fireworks, instead of only for the core firework.

With all these considerations in mind, we end up with the adaptive firework search. The pseudo-code of AFW Search is listed in Algorithm 1, where $\alpha = 0.9$, $\beta = 1.2$ according to [7].

B. Attract-Repulse Mutation (AR Mutation)

While AFW search is leveraged to guide local search, other strategies should be taken to keep the diversity of the firework swarm, which is is crucial for the success of optimization procedure. The mechanism, Attract-Repulse Mutation (after which we name the proposed algorithm in this paper), proposed in [18] is adopted in AR-FWA to achieve this aim explicitly. AR mutation is described by Algorithm 4, where $\mathbf{x}_{best}$ depicts the firework with the best fitness.

The philosophy behind AR Mutation, as illustrated by Fig. 5, is that, for non-best fireworks, they either attracted by the best firework to 'help' exploit the current best location or repulsed by the best firework to explore more space. The choice between 'attract' and 'repulse' reflects balance between exploitation and exploration.

For detailed discussion on AR mutation, readers can refer to [18]. (Notice that in both Algorithm 2 and 4, the outrange check and the corresponding mapping are omitted for the purpose of clarity.)

C. Non-uniform Mutation

Sparks are generated following Algorithm 3. In the conventional FWA, $RNG(\cdot)$ is uniform distributionc[18], [7]. To be more general, it can be any distribution that meets the following conditions:

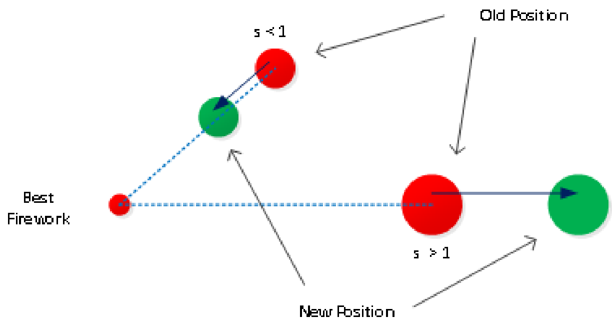


Fig. 5: Schematic Diagram of Attract-Repulse Mutation

- symmetric with respect to the original
- distributed very close to 0.

There are many distributions satisfying these conditions. Here, we only discus two of them, the Gaussian distribution and the Cauchy distribution.

1) Gaussian Distribution: The probability distribution function (PDF) of Gaussian distribution is illustrated by Eq. (7).

$$gauss(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{7}$$

where μ is the expectation and σ is the standard deviation.

2) Cauchy Distribution: The PDF of Cauchy distribution is illustrated by Eq. (8).

$$cauchy(x) = \frac{1}{\pi\gamma[1 + (\frac{x-\mu}{\gamma})^2]} \tag{8}$$

where μ is the location parameter which determines the location of the peak of the distribution (the mode of the distribution); γ is the scale parameter, specifying half the width of the PDF at half the maximum height. Similar to the Gaussian distribution the Cauchy distribution has a symmetric bell shaped probability density function, however it is more peaked at the center and has fatter tails than a Gaussian distribution.

Fig. 6 shows the probability density functions of uniform distribution, the standard Gaussian distribution and the standard Cauchy distribution. Fig. 7 illustrates a 2-D simulation results of standard Cauchy distribution ($\mu = 0$, $\gamma = 1$), standard Gaussian distribution ($\mu = 0$, $\sigma = 1$) and uniform distribution s.t. $[-1, 1]$. In the simulation, up to 100 points are drawn independently from each distribution. As can be seen, the points from uniform distribution are only located between $[-1, 1]$, while Gaussian and Cauchy distributions are more scattered. Most of the points are within the range of 3 σ for Gaussian distribution, and more outliers are generated for Cauchy distribution.

We expect that Gaussian and Cauchy distributions can result in better diversity for the firework swarm, which will be verified by experiments in Section V.

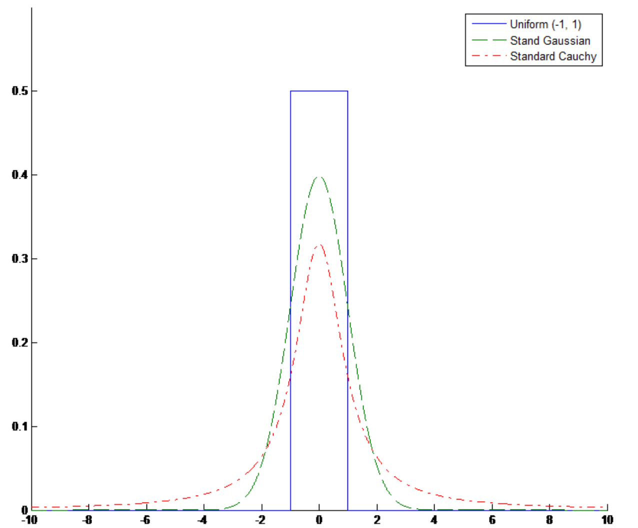


Fig. 6: Probability Density Functions of Uniform Distribution, Gaussian Distribution and Cauchy Distribution

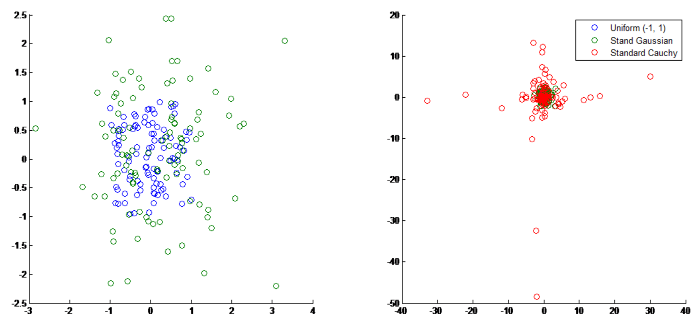


Fig. 7: 2-D Simulation Results

IV. Implementation

A. Dynamic Parallelism

Dynamic Parallelism in CUDA is supported via an extension to the CUDA programming model that enables a CUDA kernel to create and synchronize new nested work (cf. Fig. 8). Basically, a child CUDA kernel can be called from within a parent CUDA kernel and then optionally synchronize on the completion of that child CUDA kernel. The parent CUDA kernel can consume the output produced from the child CUDA kernel, all without CPU involvement [23].

Dynamic parallelism enjoys many advantages. Firstly, with dynamic parallelism, additional parallelism can be exposed to the GPUs hardware schedulers and load balancers dynamically, adapting in response to data-driven decisions or workloads (cf. Fig. 9). Secondly, algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism can be more transparently expressed. Besides, with dynamical parallelism, top-level loops can be moved to GPU, thus reduce kernel launch time

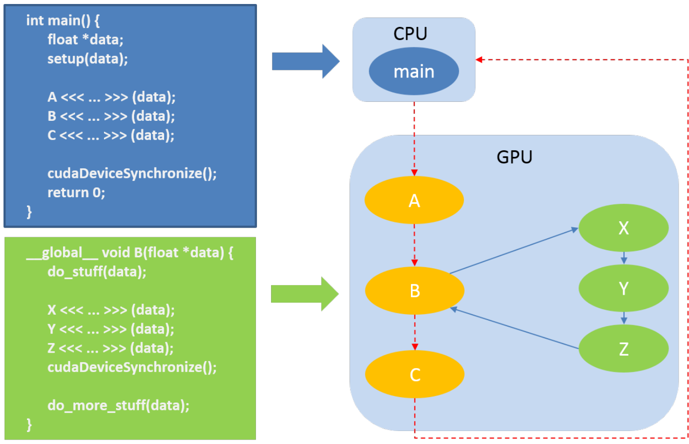


Fig. 8: Dynamica Parallel

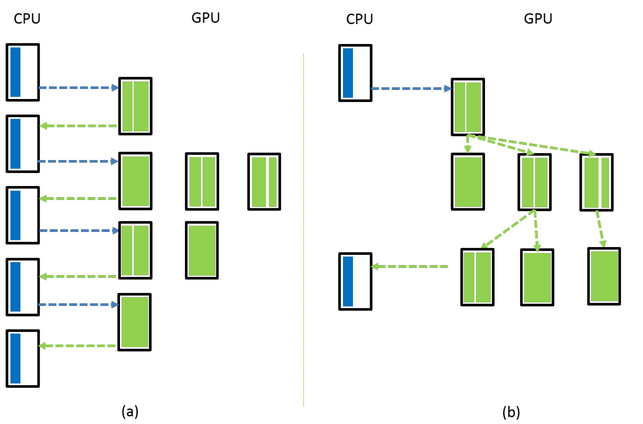


Fig. 9: Dynamic parallelism allow allocating resource in response to data-driven decisions or workloads.

B. Framework

AR-FWA is implemented by all-GPU parallel model [17] and relies heavily on dynamic parallelism. The framework based on dynamic parallelism is depicted by Fig. 10. Differing form GPU-FWA, AR-FWA move both the outer and inner loops to GPU side, thus releasing CPU from the scheduling. The whole optimization procedure is totally dependent from CPU.

In the following subsections, the implementation of the key components of AR-FWA will be described in detail respectively.

C. Random Number Generation

Random number generation is an integral component of FWA, which should be disposed with care [25].

In [18], random numbers are generated by invoking cuRAND's host API [26]. To avoid the overhead of kernel launch, device API [26] is used in AR-FWA. Another advantage of using device API is that the API calling can be easily integrated with other operations to get diverse non-uniform distributions. Listing 1 demonstrates how to use cuRAND's device API to get random numbers subject to cauchy distribution from one single kernel call.

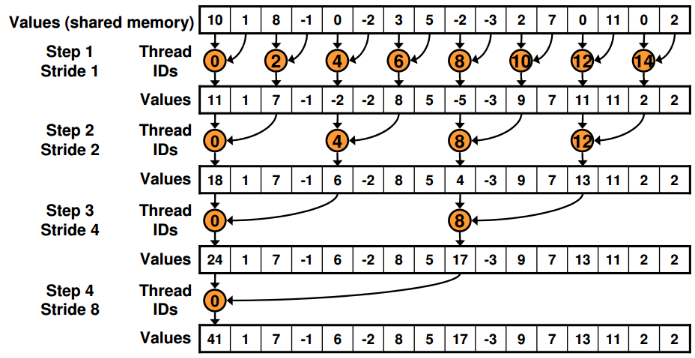


Fig. 11: Shared Memory based Reduction

D. Initialization

All fireworks are initialized within the whole feasible domain. Thus, the implementation can be based on the uniform random number generation and a simple element-wise scaling, which can be implemented in a fine-grained manner. To deal with lareg scale (high dimension) problems, grid-stride loops trick can be utilized. For a D dimension problem solved by n fireworks, the CUDA code snippet is listed in Listing 2.

E. Reduction

Reduction is not a specific kernel in AR-FWA. It is a primitive used by several different kernels instead. Reduction is used for calculating the summation and finding the maximum or minimum value of an array. Here, we discuss two methods for efficient reduction operation.

1) Shared Memory Based Reduction: The procedure of shared memory based reduction is as Fig. 11. This way, take advantage of fast shared memory, and avoid bank conflict.

Listing 3 gives out the code snippet for a summation reduction. the extension to other reductions is obvious and is given by the comment.

2) Shuffle Based Reduction: Shuffle (SHFL) is a new machine instruction introduced in Kepler architecture. The shuffle intrinsics (index, up, down and butterfly) permit exchanging of a variable between threads within the same warp without use of shared memory. The exchange occurs simultaneously for all active threads within the warp, moving 4 bytes of data per thread. Refer to [23] B.14 for details.

Compared to shared memory based reduction, shuffle based implementation can be faster. Fig. 12 compares these two reduction implementations on NVIDIA GTX 970 GPU. As can be seen, when used for reducing $10M$ data, shuffle can achieve 30%$\sim$40% improvement under various block sizes.

Code snippet for reduction using shuffle is given by Listing 4 whose intuition can be illustrated by Fig. 13. Listing 5 demonstrates how shuffle based reduction can be operated on the block level. For larger scale problem, a two-path strategy can be used with the help of global memory.

F. AFW Search

Each AFW search is executed by a single kernel which is launched dynamically by the parent thread. The new kernel

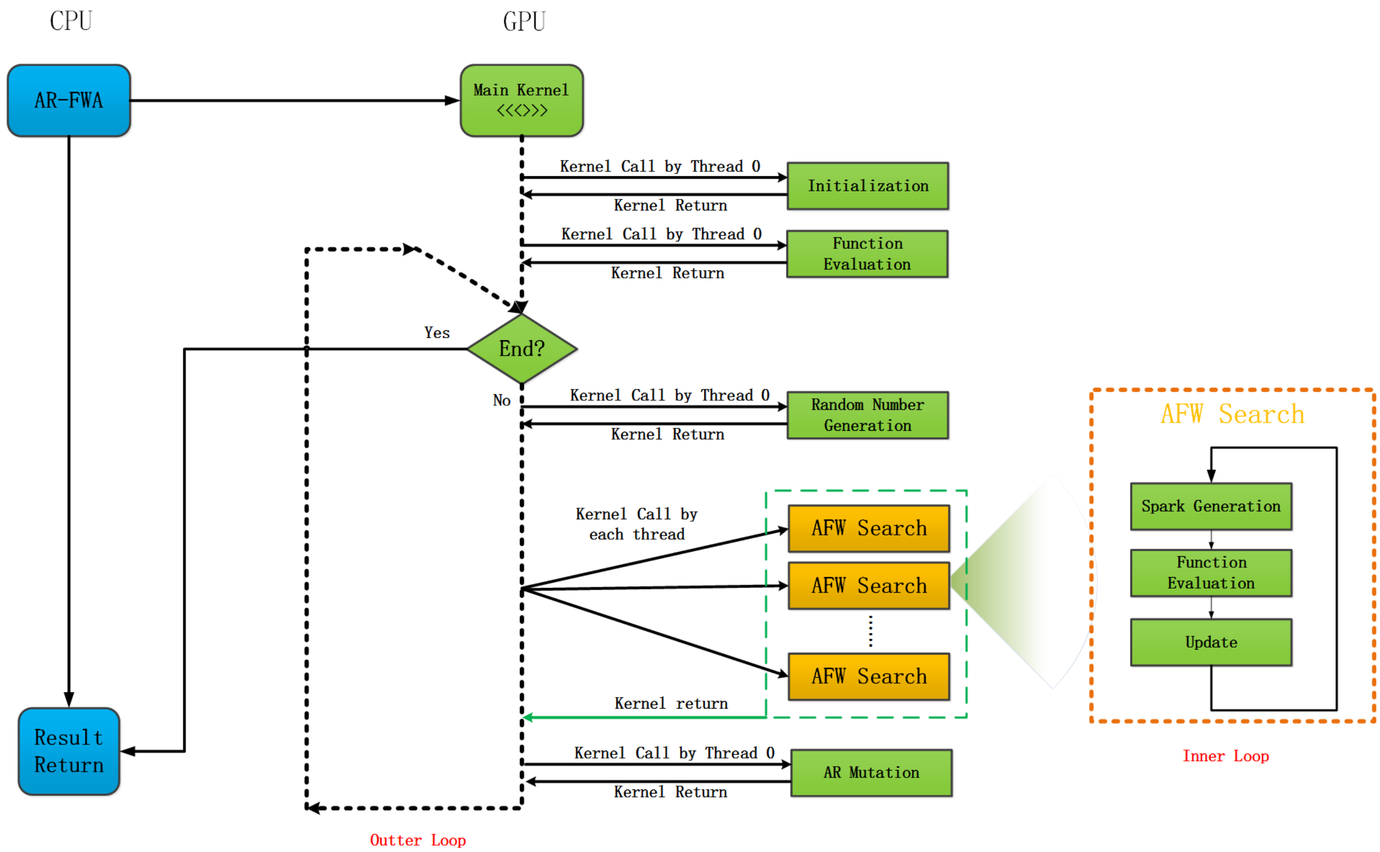


Fig. 10: Framework of GPU-based AR-FWA

Listing 1 Cauchy Distribution Random Number Generation

```
__global__ void generate_cauchy(int num, float *result, float mu, float gamma) {
    int tidx = threadIdx.x + blockIdx.x * blockDim.x;
    for (; tidx < num; tidx += blockDim.x * gridDim.x) {
        // Fetch a random number subject to uniform distribution
        float r = curand_uniform(&state[blockIdx.x]);
        // Transform r into Cauchy distribution using inverse transform methods
        result[tidx] = mu + gamma * tanf(PI * (r - 0.5f));
    }
}
```

Listing 2 Intialization

```
__global__
void initialize(float *fireworks, // fireworks to be initialized
                float *rng,       // random number pool
                float upper,      // upper bound for search
                float lower       // lower bound for search
                ) {
    int tidx = threadIdx.x + blockIdx.x * blockDim.x;
    float t;
    for (; tidx < n * D; tidx += blockDim.x * gridDim.x) {
        t = rng[tidx]; // uniform random number
        t = lower + (upper - lower) * t; // scale to (lower, upper)
        fireworks[tidx] = t;        // write back to memory
    }
}
```

Listing 3 Parallel Reduction Based on Shared Memory (Summation)

```
__inline__ __device__
float reduceSum(float *arr, int num) {
    // Reduction for finding maximum is followed as comments.
    __shared__ float sdata[];

    int tidx = threadIdx.x;
    sdata[tidx] = 0;  // sdata[tidx] = -inf;

    // Read all data to be reduced into shared memory
    for (int i = tidx; i < num; i += blockDim.x) {
        sdata[tidx] += arr[i]; // sdata[tidx] = max(sdata, arr[i]);
    }
    __syncthreads();

    // Reduce using shared memory by the 1st warp
    for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s]; // sdata[tid] = max(sdata[tid], sdata[tid + s]);
        }
        __syncthreads();
    }

    return sdata[0];
}
```

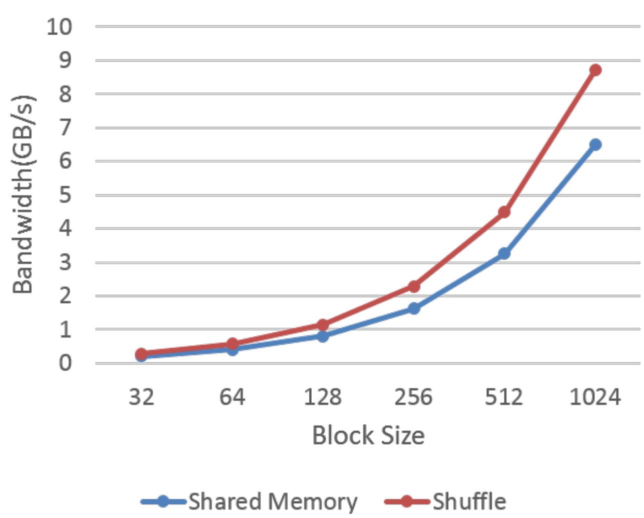


Fig. 12: Shared memory based reduction vs. shuffle based reduction

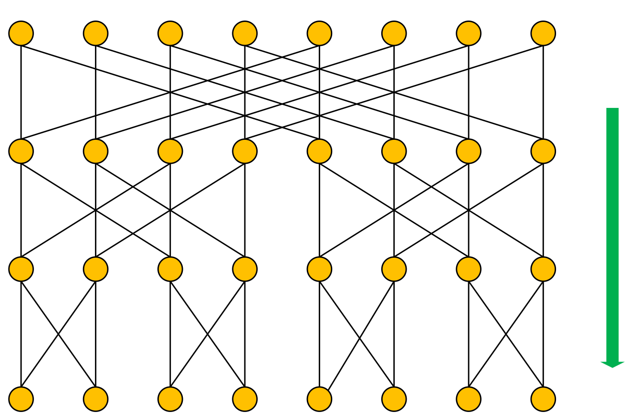


Fig. 13: Principal Diagram of Shuffle Based Reduction

Listing 4 Parallel Reduction Using Shuffle (Warp)

```
__inline__ __device__
int warpReduceSum(int val) {
    // Reduce using shuffle (cf. Fig. 13)
    for (int m = warpSize/2; m > 0; m >>= 1)
        val += __shfl_xor(val, m);
    return val;
}
```

Listing 5 Parallel Reduction Using Shuffle (Block)

```
__inline__ __device__
int blockReduceSum(int val) {
  // Shared memory for 32 partial sums
  static __shared__ int shared[32];
  int lane = threadIdx.x % warpSize;
  int wid = threadIdx.x / warpSize;

  // Each warp performs partial reduction
  val = warpReduceSum(val);

  // Write reduced value to shared memory
  if (lane == 0) shared[wid]=val;

  // Wait for all partial reductions
  __syncthreads();

  // read only if that warp existed
  val = (lane < blockDim.x / warpSize) ?
        shared[lane] : 0;

  // Final reduce within first warp
  if (wid == 0) val = warpReduceSum(val);

  return val;
}
```

launches its own child kernel, dynamically, for spark generation, objective evaluation and update.

G. AR Mutation and Spark Generation

The implementation of AR mutation is very similar to initialization. The random numbers are drawn from the random numbers pool in a fine-grained manner. The code snippet is illustrated by Listing 6.

The implementation of spark generation is very similar to that of AR mutation. So the code is omitted here.

H. Objective Function Evaluation

In the implementation, the fine-grained strategy is adopted to parallelized function evaluation [17]. The code snippet is given by Listing 7.

Listing 7 Fine-grained Function Evaluation (Sphere Function)

```
__inline__ __device__
float evaluate(float *x) {
    // Shared memory
    extern __shared__ float sdata[];

    // Initialize shared memory to 0
    int tidx = threadIdx.x;
    sdata[tidx] = 0;

    // Read all data into shared memory
    float tmp;
    for (int i = tidx; i < D; i += blockDim.x) {
        tmp = x[i];
        sdata[tidx] += tmp * tmp;
    }
    __syncthreads();

    // Reduce using shared memory by the 1st warp
    for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tidx < s) {
            sdata[tidx] += sdata[tidx + s];
        }
        __syncthreads();
    }

    return sdata[0];
}
```

1) Firework Update: To update the firework using the best spark, the spark with best fitness value should be located. This can be implemented by reduction operation, which we have discussed in the previous subsection. Having the best spark, the update can be conducted in a fine-grained way (each thread for a dimension), which is very similar to the initialization AR mutation and spark generation.

V. Experiments and Analysis

A. Benchmark functions

In our experiments, the GPU-based benchmark, cuROB, is used [27]. cuROB is implemented with CUDA and can support any dimension within the limit of hardware. The current release of cuROB includes 37 single objective real-parameter optimization functions. The test functions fall into four categories: unimodal functions (No. 0∼6), basic multi-modal functions (No. 7∼22), hybrid functions (No. 23∼28) and composition functions (No. 29∼36). The summary of the suit is listed in Tab. I. Detailed information for each function is given in [27].

B. Algorithm Performance

In the experiments, all algorithm are implemented using the Naive Parallel Model [17] with double precision float operation.

1) Uniform Mutation vs. Non-uniform Mutation: To verify the feasibility of non-uniform mutation, we implement Algorithm 3 using uniform distribution s.t. $[-1, 1]$, standard Gaussian distribution and Cauchy distribution, respectively. The test functions are all with dimension of 30 ($D = 30$), and up to $D * 10000$ function evaluations are conducted for each run. The number of firework is $n = 5$, and the number of sparks $s = 150$, $A = 40$, $\Delta = 0.00001$. For AFW search, $L = 100$, $\alpha = 1.2$, $\beta = 0.9$, and for AR mutation $\delta = 0.5$. 151 independent runs are conducted for each test function. The finally results are listed in Tab. IV.

The experimental results are listed in Tab. II.

TABLE II: Results for AR-FWA with Uniform and Non-uniform Mutation

NO.	Uniform		Gaussian		Cauchy	
	Mean	Std.	Mean	Std.	Mean	Std.
0	1.00E+02	1.32E-14	1.00E+02	2.26E-14	1.00E+02	3.55E-14
1	1.00E+02	1.93E-13	1.00E+02	4.03E-12	1.00E+02	5.25E-07
2	5.28E+05	2.14E+05	5.42E+05	2.44E+05	1.03E+06	4.35E+05
3	8.97E+02	4.14E+02	8.02E+02	4.01E+02	3.09E+03	1.55E+03
4	6.97E+03	7.67E+03	7.21E+03	6.87E+03	9.86E+03	8.37E+03
5	1.00E+02	2.13E-05	1.00E+02	2.60E-05	1.00E+02	5.41E-05
6	1.01E+02	1.04E+00	1.01E+02	1.40E+00	1.02E+02	2.20E+00
7	1.00E+02	5.46E-01	1.00E+02	5.12E-01	1.00E+02	4.04E-01
8	1.10E+02	2.17E+00	1.09E+02	2.16E+00	1.11E+02	2.43E+00
9	1.00E+02	2.57E-03	1.00E+02	3.38E-03	1.00E+02	4.34E-03
10	1.84E+02	1.48E+01	1.73E+02	1.36E+01	1.05E+02	3.00E+00
11	1.86E+02	1.43E+01	1.77E+02	1.17E+01	1.82E+02	1.37E+01
12	1.28E+02	6.62E+00	1.17E+02	4.44E+00	1.16E+02	4.40E+00
13	1.05E+02	1.27E+00	1.05E+02	1.46E+00	1.06E+02	1.62E+00
14	1.19E+02	2.65E+00	1.19E+02	2.81E+00	1.22E+02	2.62E+00
15	2.48E+03	3.41E+02	2.14E+03	3.27E+02	2.65E+02	1.16E+02
16	2.54E+03	3.60E+02	2.35E+03	3.37E+02	2.50E+03	3.79E+02
17	1.00E+02	7.02E-02	1.00E+02	7.43E-02	1.00E+02	8.98E-02
18	1.30E+02	4.79E-01	1.30E+02	4.76E-01	1.30E+02	6.71E-01
19	1.20E+02	6.36E-04	1.20E+02	1.47E-04	1.20E+02	4.06E-04
20	1.00E+02	6.26E-02	1.00E+02	5.70E-02	1.00E+02	6.28E-02
21	1.00E+02	6.13E-02	1.00E+02	3.73E-02	1.00E+02	9.74E-02
22	1.11E+02	4.68E-01	1.11E+02	3.92E-01	1.10E+02	3.82E-01
23	4.62E+04	1.70E+04	4.59E+04	1.47E+04	3.84E+04	1.25E+04
24	4.10E+04	8.01E+03	4.14E+04	9.11E+03	7.94E+03	5.78E+03
25	1.16E+02	6.86E+00	1.19E+02	1.84E+01	1.17E+02	1.83E+01
26	7.53E+03	5.31E+03	7.58E+03	4.87E+03	5.57E+03	3.04E+03
27	2.57E+04	1.03E+04	2.70E+04	1.12E+04	3.55E+04	3.29E+04
28	1.22E+02	5.45E-01	1.22E+02	5.71E-01	1.21E+02	6.48E-01
29	3.76E+02	2.58E-06	3.76E+02	2.84E-06	3.76E+02	2.41E-02
30	4.17E+02	1.80E+01	4.07E+02	1.47E+01	4.05E+02	1.44E+01
31	3.23E+02	4.82E+00	3.24E+02	4.04E+00	3.22E+02	3.48E+00
32	2.01E+02	5.75E-02	2.01E+02	5.46E-02	2.01E+02	6.68E-02
33	4.62E+02	4.41E+00	4.61E+02	4.72E+00	4.63E+02	4.46E+00
34	1.62E+03	1.52E+02	1.49E+03	1.36E+02	1.30E+03	1.12E+02
35	2.42E+07	2.55E+06	2.31E+07	2.40E+06	1.90E+07	3.81E+05
36	5.05E+06	1.04E+06	4.39E+06	7.46E+05	3.02E+06	3.18E+05

Via t-test, the comparison results are listed in Tab III. Obviously, Non-uniform mutation gains no benefit on the simple unimodal problems. However, for the more complicated multimodal problems, both Gaussian and Cauchy distributions improve the performance in some degress. Cauchy (11 better) can achieve more significant improvement compared to Gaussian (9 better).

2) Compared to the State-of-the-Art FWA variants: In this part, we compare AR-FWA to the-state-of-the-art FWA variants, dynFWA [7] and EFWA [19].

The experimental results are listed in Tab. IV, which is the mean on 151 independent runs with $D * 10000$ function

Listing 6 AR Mutation

```
__global__ void AR_Mutate(floatX *fireworks, floatX* best_position, floatX *rng) {
    // Move pointer to the proper locations
    int tidx = threadIdx.x;
    fireworks += blockIdx.x * D;
    rng += blockIdx.x * D * 2;

    // for thread 0 to thread D - 1, each for one dimension
    if (tidx < D) {
        floatX c = best_position[tidx];
        floatX t = fireworks[tidx];

        // Mutate accordingly
        c += (t - c) * (rng[tidx] * 2 * delta + 1 - delta);
        fireworks[tidx] = rng[tidx + dim] < 0.5 ? c : t;
    }
}
```

TABLE I: Summary of cuROB's Test Functions

	No.	Functions	ID	Description
Unimodal Functions	0	Rotated Sphere	SPHERE	Optimum easy to track
	1	Rotated Ellipsoid	ELLIPSOID	
	2	Rotated Elliptic	ELLIPTIC	Optimum hard to track
	3	Rotated Discus	DISCUS	
	4	Rotated Bent Cigar	CIGAR	
	5	Rotated Different Powers	POWERS	
	6	Rotated Sharp Valley	SHARPV	
Basic Multi-modal Functions	7	Rotated Step	STEP	With adepuate global structure
	8	Rotated Weierstrass	WEIERSTRASS	
	9	Rotated Griewank	GRIEWANK	
	10	Rastrigin	RARSTRIGIN_U	
	11	Rotated Rastrigin	RARSTRIGIN	
	12	Rotated Schaffer's F7	SCHAFFERSF7	
	13	Rotated Expanded Griewank plus Rosenbrock	GRIE_ROSEN	
	14	Rotated Rosenbrock	ROSENBROCK	With weak global structure
	15	Modified Schwefel	SCHWEFEL_U	
	16	Rotated Modified Schwefel	SCHWEFEL	
	17	Rotated Katsuura	KATSUURA	
	18	Rotated Lunacek bi-Rastrigin	LUNACEK	
	19	Rotated Ackley	ACKLEY	
	20	Rotated HappyCat	HAPPYCAT	
	21	Rotated HGBat	HGBAT	
	22	Rotated Expanded Schaffer's F6	SCHAFFERSF6	
Hybrid Functions	23	Hybrid Function 1	HYBRID1	With different properties for different variables subcomponents
	24	Hybrid Function 2	HYBRID2	
	25	Hybrid Function 3	HYBRID3	
	26	Hybrid Function 4	HYBRID4	
	27	Hybrid Function 5	HYBRID5	
	28	Hybrid Function 6	HYBRID6	
Composition Functions	29	Composition Function 1	COMPOSITION1	Properties similar to particular sub-function when approaching the corresponding optimum
	30	Composition Function 2	COMPOSITION2	
	31	Composition Function 3	COMPOSITION3	
	32	Composition Function 4	COMPOSITION4	
	33	Composition Function 5	COMPOSITION5	
	34	Composition Function 6	COMPOSITION6	
	35	Composition Function 7	COMPOSITION7	
	36	Composition Function 8	COMPOSITION8	
Search Space: $[-100, 100]^D$, $f_{opt} = 100$				

TABLE III: Performance Comparison between Uniform and Gaussian & Cauchy Distributions (better/inconclusive/worse)

	Unimodal	Basic Multimodal	Hybrid	Composition	Summary
Uniform vs Gaussian	3/3/1	2/6/8	3/2/1	3/3/2	11/14/12
Uniform vs Cauchy	3/4/0	3/8/5	1/1/4	2/0/5	9/14/14

evaluations. The parameters of dynFWA and EFWA are as in the original paper. Considering the better performance of Cauchy distribution (cf. the last subsection), for AR-FWA, Cauchy distribution is adopted. The parameters are the same as in the previous subsection.

TABLE IV: AR-FWA vs. dynFWA and EFWA

ID	AR-FWA		dynFWA		EFWA	
	Mean	Std.	Mean	Std.	Mean	Std.
0	1.000E+02	3.551E-14	1.000E+02	7.965E-14	1.000E+02	3.764E-04
1	1.000E+02	5.250E-07	1.000E+02	2.282E-13	1.004E+02	1.378E-01
2	1.034E+06	4.351E+05	8.128E+05	3.858E+05	7.552E+05	2.803E+05
3	3.092E+03	1.555E+03	6.621E+02	3.104E+02	1.005E+02	2.011E-01
4	9.860E+03	8.371E+03	9.365E+03	1.112E+04	4.814E+03	4.843E+03
5	1.000E+02	5.413E-05	1.000E+02	1.479E-05	1.000E+02	9.879E-05
6	1.018E+02	2.198E+00	1.070E+02	1.060E+01	1.092E+02	9.949E+00
7	1.002E+02	4.039E-01	1.109E+02	3.725E+00	1.023E+02	1.825E+00
8	1.110E+02	2.434E+00	1.291E+02	4.527E+00	1.330E+02	3.489E+00
9	1.000E+02	4.340E-03	1.000E+02	1.243E-02	1.000E+02	1.287E-02
10	1.045E+02	3.000E+00	2.682E+02	4.835E+01	2.689E+02	3.461E+01
11	1.823E+02	1.374E+01	3.243E+02	4.638E+01	3.115E+02	4.713E+01
12	1.155E+02	4.404E+00	1.632E+02	9.136E+00	1.634E+02	9.270E+00
13	1.060E+02	1.615E+00	1.179E+02	1.315E+01	1.170E+02	5.887E+00
14	1.223E+02	2.625E+00	1.285E+02	2.327E+01	1.418E+02	3.055E+01
15	2.648E+02	1.162E+02	2.332E+03	6.405E+02	3.226E+03	6.073E+02
16	2.500E+03	3.794E+02	4.184E+03	6.845E+02	4.424E+03	6.520E+02
17	1.002E+02	8.980E-02	1.006E+02	2.719E-01	1.004E+02	2.221E-01
18	1.301E+02	6.709E-01	1.300E+02	2.794E-13	1.300E+02	8.733E-04
19	1.200E+02	4.057E-04	1.200E+02	1.480E-04	1.200E+02	4.465E-04
20	1.004E+02	6.281E-02	1.006E+02	1.443E-01	1.005E+02	1.326E-01
21	1.002E+02	9.736E-02	1.005E+02	2.860E-01	1.003E+02	1.140E-01
22	1.104E+02	3.820E-01	1.117E+02	5.439E-01	1.121E+02	5.059E-01
23	3.836E+04	1.246E+04	8.773E+04	5.074E+04	4.546E+04	1.677E+04
24	7.937E+03	5.784E+03	6.564E+03	8.112E+03	4.564E+03	5.273E+03
25	1.171E+02	1.834E+01	8.852E+02	2.876E+03	5.610E+02	1.540E+03
26	5.568E+03	3.038E+03	2.912E+04	3.473E+04	6.446E+03	9.266E+03
27	3.548E+04	3.289E+04	7.684E+04	9.194E+04	2.835E+04	2.076E+04
28	1.213E+02	6.476E-01	2.647E+02	1.866E+02	1.328E+02	4.867E+01
29	3.764E+02	2.410E-02	3.768E+02	4.146E+00	3.764E+02	7.962E-04
30	4.048E+02	1.443E+01	6.162E+02	8.352E+01	7.539E+02	9.603E+01
31	3.222E+02	3.483E+00	3.401E+02	2.623E+01	3.684E+02	1.768E+01
32	2.014E+02	6.678E-02	2.115E+02	5.037E+01	2.032E+02	2.299E+01
33	4.634E+02	4.461E+00	5.045E+02	1.996E+01	5.181E+02	1.375E+01
34	1.304E+03	1.120E+02	1.154E+03	8.460E+02	3.569E+03	6.951E+02
35	1.897E+07	3.813E+05	2.257E+02	5.844E+02	1.409E+07	9.768E+06
36	3.016E+06	3.181E+05	1.889E+03	3.020E+03	2.573E+03	2.263E+03

a) Unimodal Functions: The convergence trends on unimodal functions for the three algorithms are depicted by Fig. 14. Over all, AR-FWA has slower convergence rate on unimodal functions campared to dynFWA and EFWA.

The t-test results are listed in Tab. V, and the comparison results are listed in Tab. VI, where $+1$ denotes significantly better and -1 significant worse, 0 inconclusive.

AR-FWA is significantly worse on function NO. $2 \sim 4$, and for 2 and 3, the disparity is as mush as an order of magnitude. The result is no surprise. As dynFWA and EFWA select the next generation using the ERS strategy (cf. II-A4), population converge to a location quickly due to the high competition pressure. Therefor dynFWA and EFWA can outperform AF-FWA unimodal functions.

Despite the poor performance on unimodal functions, we expect AR-FWA can achieve better results for complicated multi-modal functions. In the following subsection, we will verify this hypothesis with experiments.

b) Basic Multimodal Functions: Similar to the analysis on unimodal functions, the t-test and comparison results are listed by Tab. VII and Tab. VIII respectively. Out of the 18 functions, AR-FWA outperforms dynFWA on 13 and 1 not statistic significant. AR-FWA outperforms EFWA on 11 functions and 4 not statistic significant. Obviously, AR-FWA can achieve better performance than dynFWA and EFWA do on the basic multimodal functions.

TABLE VIII: AR-FWA v.s. EFEA and dynFWA (Basic Multimodal)

	7	8	9	10	11	12	13	14
dynFWA	+1	0	+1	+1	+1	+1	+1	+1
EFWA	+1	+1	+1	+1	0	+1	+1	+1
	15	**16**	**17**	**18**	**19**	**20**	**21**	**22**
dynFWA	+1	+1	+1	-1	-1	+1	+1	+1
EFWA	+1	0	+1	-1	+1	+1	0	0

c) Hybrid Functions: Hybrid functions are more complicated than basic multimodal functions, thus can simulate the complicated real world scenarios better. The results on hybrid functions are listed by Tab. IX and Tab. X, respectively. Out of the 6 hybrid functions, AR-FWA outperforms dynFWA on 5 of them, and get worse result on 1 of them. AR-FWA outperforms EFWA on 3 functions, worse on 2, and 1 inconclusive. Overall, AF-FWA performs better than EFWA and dynFWA for hybrid functions.

TABLE X: AR-FWA v.s. EFEA and dynFWA (Hybrid)

	23	24	25	26	27	28
dynFWA	+1	-1	+1	+1	+1	+1
EFWA	+1	-1	+1	+1	-1	0

TABLE XII: AR-FWA v.s. EFEA and dynFWA (Composition)

	29	30	31	32	33	34	35	36
dynFWA	+1	+1	+1	+1	+1	-1	-1	-1
EFWA	-1	+1	+1	+1	+1	+1	-1	-1

d) Composition Functions: Composition functions are more complicated than basic multimodal and hybrid functions. The results on composition functions are listed by Tab. XI and Tab. XII. Out of the 8 composition functions, AR-FWA outperforms dynFWA on 5 of them, get worse result on 3 of them. AR-FWA outperforms EFWA on 5 functions, worse on 3. At least AF-FWA performs no worse than dynFWA and EFWA.

All comparison results are summarized by Tab. XIII. AR-FWA is worse than dynFWA and EFWA on unimodal functions, but outperforms the other tow algorithms on multimodal functions generally.

TABLE XIII: Summary of Comparison Results

	better	inconclusive	worse
AR-FWA vs. dynFWA	24	2	11
AR-FWA vs. EFWA	23	5	9

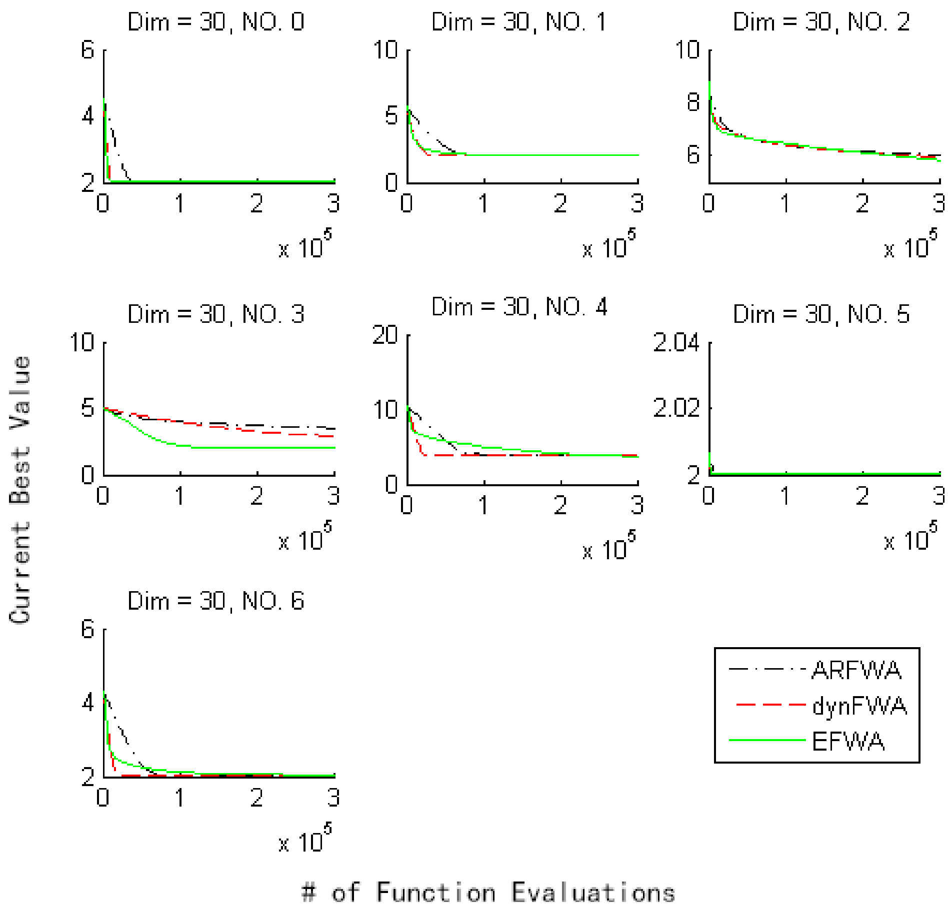


Fig. 14: Convergence on Unimodal Functions

TABLE V: p Values of t-test (Unimodal)

	0	1	2	3	4	5	6
dynFWA	1.00E+00	1.55E-109	1.00E-06	1.43E-96	4.70E-06	1.85E-10	1.50E-52
EFWA	3.07E-02	8.30E-03	2.45E-54	4.09E-53	9.04E-19	4.38E-26	6.89E-16

TABLE VI: AR-FWA v.s. EFWA and dynFWA (Unimodal)

	0	1	2	3	4	5	6
dynFWA	0	-1	-1	-1	-1	-1	+1
EFWA	+1	+1	-1	-1	-1	+1	+1

TABLE VII: p Values of t-test (Basic Multimodal)

	7	8	9	10	11	12	13	14
dynFWA	3.08E-70	6.64E-1	6.33E-10	3.58E-53	9.48E-131	1.61E-08	7.39E-17	6.70E-108
EFWA	4.24E-72	2.77E-101	7.4E-26	4.05E-05	9.25E-2	2.51E-07	1.20E-3	4.8E-4
	15	16	17	18	19	20	21	22
dynFWA	3.46E-34	1.15E-130	9.09E-176	8.37E-19	1.05E-16	5.11E-126	5.73E-165	9.00E-111
EFWA	4.29E-15	2.71E-1	3.9303E-07	2.55E-2	1.33E-18	4.1132E-3	2.17E-1	1.52E-1

TABLE IX: p Values of t-test (Hybrid)

	23	24	25	26	27	28
dynFWA	1.97E-99	3.08E-164	2.50E-163	6.21E-24	2.05E-64	1.28E-03
EFWA	4.53E-94	5.80E-133	4.00E-15	7.80E-97	1.43E-02	3.19E-01

TABLE XI: p Values of t-test (Composition)

	29	30	31	32	33	34	35	36
dynFWA	9.95E-14	3.15E-119	2.03E-166	4.60E-80	2.66E-96	2.60E-48	1.33E-20	2.77E-02
EFWA	5.84E-74	1.13E-138	3.22E-02	1.23E-120	0.00E+00	3.02E-09	2.74E-251	2.91E-251

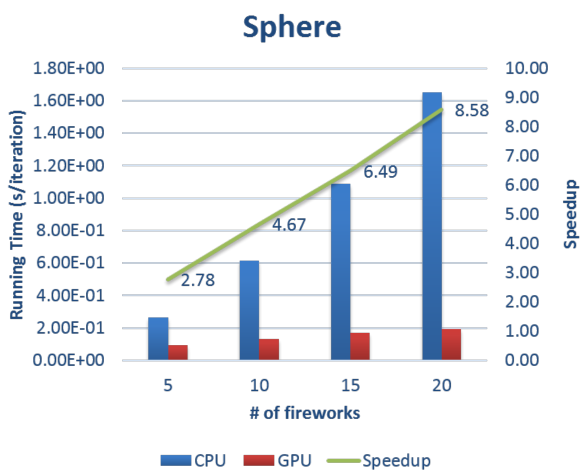


Fig. 15: Speedup with Different Population Size ($D = 30$)

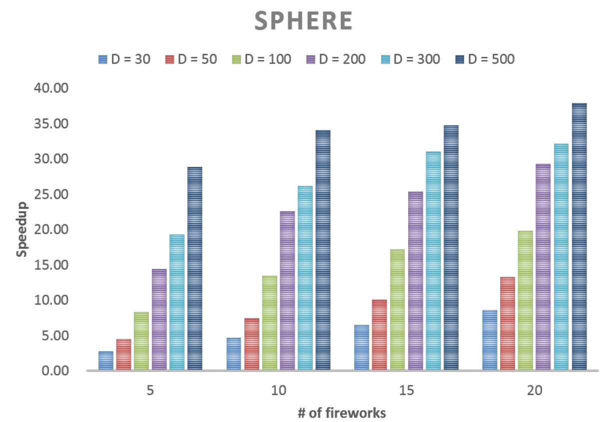


Fig. 16: Speedup with Different Dimensions

C. Parallel Performance

In this part, we study the parallel performance of the GPU-based implementation of AR-FWA. All experiments are conduct on Windows 7 x64 with 8G DDR3 memory and Intel core I5-2310 and NVIDIA GeForce GTX 970 GPU. The programs are compiled with VS 2013 with CUDA 6.5. Single precision float number is adopted by both CPU and GPU implementation.

In practice, the speedup is closely related with the characteristics of the objective function. Here, we use Sphere function as benchmark for evaluating the speedup under different conditions. In the experiments, the total number of sparks ($\hat{S}$) are set to 20 fold of the number of fireworks (n).

1) Speedup against Population Size: Fig. 15 illustrates the speedup achieved by GPU-based AR-FWA with respect to its CPU-based counterpart, under various population sizes. In this experiment, the dimension of the test function is set to 30 ($D = 30$).

Even with small population ($n = 5$), the GPU-based version can achieve up to 3x speedup. As the population size goes larger, the speedup become more significant (~9x with $n = 20$).

2) Speedup against Parallelism: Besides population size, parallelism of the objective function is one of the key factors impacting the overall speedup. In our implementation, the objective function is parallelized in a fine-grained way. Therefore, by controlling the dimension, we can alter the parallelism of the test function. Fig. 16 compares the speedups under various dimensions. Similar to the impact of population size, the speedup is increasingly larger along with the dimensions. With high parallelism, AR-FWA can achieve approximately 40x speedup.

VI. Conclusions

In this paper, an efficient FWA variant, AR-FWA, is proposed. AR-FWA leverages the recently developed techniques from both FWA study and GPU computing, ending up with an adaptive firework search mechanism and a novel non-uniform mutation strategy. Compared to the state-of-the-art FWA variants, dynFWA and EFWA, AR-FWA can improve the performance greatly with respect to the complicated multimodal functions. AR-FWA relies heavily on the cutting-edge CUDA techiques, e.g. dynamic parallelism, shuffle instruction, et al. Compared to the CPU-based implementation, the GPU-based AR-FWA can achieve significant speedup under different population sizes and objective function parallelisms.

Acknowledgments

This work was supported by the Natural Science Foundation of China (NSFC) under grant no. 61375119, 61170057 and 60875080, and partially supported by National Key Basic Research Development Plan (973 Plan) Project of China with grant no. 2015CB352300.

References

[1] Y. Tan and Y. Zhu, “Fireworks Algorithm for Optimization,” in *Advances in Swarm Intelligence*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, vol. 6145, pp. 355–364.

[2] Y. Tan, C. Yu, S. Zheng, and K. Ding, “Introduction to Fireworks Algorithm,” *International Journal of Swarm Intelligence Research*, vol. 4, no. 4, pp. 39–70, October 2013.

[3] A. Janecek and Y. Tan, “Using Population Based Algorithms for Initializing Nonnegative Matrix Factorization,” in *Advances in Swarm Intelligence*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6729, pp. 307–316.

[4] H. Gao and M. Diao, “Cultural Firework Algorithm and its Application for Digital Filters Design,” *International Journal of Modelling, Identification and Control*, vol. 14, no. 4, pp. 324–331, January 2011.

[5] W. He, G. Mi, and Y. Tan, "Parameter Optimization of Local-Concentration Model for Spam Detection by Using Fireworks Algorithm," in *Advances in Swarm Intelligence*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7928, pp. 439–450.
[6] X. Yang and Y. Tan, "Sample Index Based Encoding for Clustering Using Evolutionary Computation," in *Advances in Swarm Intelligence*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8794, pp. 489–498.
[7] S. Zheng, A. Janecek, Y. Tan, and J. Li, "Dynamic Search in Fireworks Algorithm," in *Evolutionary Computation (CEC), 2014 IEEE Congress on*, 2014, pp. 3222–3229.
[8] J. Li, S. Zheng, and Y. Tan, "Adaptive Fireworks Algorithm," in *Evolutionary Computation (CEC), 2014 IEEE Congress on*, July 2014, pp. 3214–3221.
[9] P. Ross, "Why CPU Frequency Stalled," *Spectrum, IEEE*, vol. 45, no. 4, p. 72, april 2008.
[10] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Beijing, China: Tsinghua University Press, 2010.
[11] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*. Addison-Wesley Professional, 2011.
[12] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.
[13] AMD Inc., "Application Showcase." [Online]. Available: http://developer.amd.com/community/application-showcase/
[14] NVIDIDA Corp., "CUDA in Action - Research & Apps." [Online]. Available: https://developer.nvidia.com/cuda-action-research-apps
[15] Y. Zhou and Y. Tan, "GPU-Based Parallel Particle Swarm Optimization," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, May 2009, pp. 1493–1500.
[16] W. Zhu and J. Curry, "Parallel Ant Colony for Nonlinear Function Optimization with Graphics Hardware Acceleration," in *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*, 2009, pp. 1803–1808.
[17] Y. Tan and K. Ding, "Survey of GPU-Based Implementation of Swarm Intelligence Algorithms," *Cyber, IEEE Trans. on* (TBA).
[18] K. Ding, S. Zheng, and Y. Tan, "A GPU-based Parallel Fireworks Algorithm for Optimization," in *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, ser. GECCO '13. New York, NY, USA: ACM, 2013, pp. 9–16.
[19] S. Zheng, A. Janecek, and Y. Tan, "Enhanced Fireworks Algorithm," in *Evolutionary Computation (CEC), 2013 IEEE Congress on*, 2013, pp. 2069–2077.
[20] Y. Pei, S. Zheng, Y. Tan, and H. Takagi, "An Empirical Study on Influence of Approximation Approaches on Enhancing Fireworks Algorithm," in *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, 2012, pp. 1322–1327.
[21] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep Learning with COTS HPC Systems," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, S. Dasgupta and D. McAllester, Eds., vol. 28, no. 3. JMLR Workshop and Conference Proceedings, May 2013, pp. 1337–1345.
[22] K. Ballou and N. Mohammad Mousa, "OpenCUDA+MPI," Student Research Initiative, Tech. Rep. Paper 14, 2013.
[23] NVIDIA Corp., *CUDA C Programming Guide v7.0*, March 2015. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/
[24] Khronos OpenCL Working Group, *The OpenCL 1.2 Specification*, November 2011. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
[25] K. Ding and Y. Tan, "Comparison of Random Number Generators in Particle Swarm Optimization Algorithm," in *the Proceeding of 2014 IEEE Congress on Evolutionary Computation, (IEEE CEC 2014)*, 2014, pp. 2664–2671.
[26] NVIDIA Corp., *CURAND Library Programming Guide v7.0*, March 2015. [Online]. Available: http://docs.nvidia.com/cuda/curand/
[27] K. Ding and Y. Tan, "cuROB: A GPU-Based Test Suit for Real-Parameter Optimization," in *Advances in Swarm Intelligence*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 66–78.

Ke Ding is currently majoring in Computer Science and working towards the Ph.D. degree at the Key Laboratory of Machine Perception (MOE) and School of Electronics Engineering and Computer Science, Peking University, China. His research interests are related to Swarm Intelligence, GPU computing, parallel programming and machine learning.

Ying Tan is a full professor and PhD advisor at the School of Electronics Engineering and Computer Science of Peking University, and director of Computational Intelligence Laboratory at Peking University (PKU). He received his PhD from Southeast University in 1997. From 1997, he was a postdoctoral fellow then an associate professor at University of Science and Technology of China (USTC), then served as director of Institute of Intelligent Information Science and a full professor since 2000. He worked with the Chinese University of Hong Kong (CUHK) in 1999 and 2004-2005. He was an electee of 100 talent program of the Chinese Academy of Science (CAS) in 2005. He is the inventor of Fireworks Algorithm (FWA). He serves as the Editor-in-Chief of International Journal of Computational Intelligence and Pattern Recognition (IJCIPR), the Associate Editor of IEEE Transaction on Cybernetics (Cyb), International Journal of Artificial Intelligence (IJAI), International Journal of Swarm Intelligence Research (IJSIR), International Journal of Intelligent Information Processing (IJIIP), IES Journal B, Intelligent Devices and Systems, and Advisory Board of International Journal on Knowledge Based Intelligent Engineering (KES), and The Editorial Board of Journal of Computer Science and Systems Biology (JCSB), Journal of Applied Mathematics (JAM), Applied Mathematical and Computational Sciences (AMCOS), International Journal of Advance Innovations, Thoughts and Ideas, Immune Computing (ICJ), Defense Technology (DT), CAAI Transactions on Intelligent Systems. He also served as an Editor of Springers Lecture Notes on Computer Science for more than 10 volumes, and Guest Editors of several referred Journals, including Information Science, Softcomputing, Neurocomputing, IJAI, IJSIR, BB, etc. He is a member of Emergent Technologies Technical Committee (ETTC), Computational Intelligence Society of IEEE since 2010. He is/was the general chair of the International Conference on Swarm Intelligence (ICSI 2010-14) and joint general chair of 1st BRICS CCI, program committee co-chair of WCCI 2014, ICACI2012, ISNN 2008 and so on. His research interests include computational intelligence, swarm intelligence, data mining, pattern recognition, intelligent information processing for information security. He has published more than 280 papers in refereed journals and conferences in these areas, and authored/co-authored 8 books and 10 chapters in book, and received 3 invention patents.