Exploiting GPU Parallelism to Optimize Real-World Problems

Md. Hasan Furhad[1], Fahmida Ahmed[2], Md. Faisal Faruque[3], Md. Iqbal Hasan Sarker[4]
[1]Department of Computer Engineering & Information Technology, University of Ulsan, South Korea
[2]Department of Computer Science & Engineering, Chittagong University, Bangladesh
[3]Department of Computer Science & Engineering, University of Information Technology & Sciences, Bangladesh
[4]Department of Computer Science & Engineering, Chittagong University of Engineering & Technology, Bangladesh
Email:hfurhad@gmail.com; sarzana.cse.cu14@gmail.com; faisal_uits@yahoo.com; iqbal@cuet.ac.bd

***Abstract*— Construction of optimal schedule for airline crew-scheduling requires high computation time. The main objective to create this optimal schedule is to assign all the crews to available flights in a minimum amount of time. This is a highly constrained optimization problem. In this paper, we implement co-evolutionary genetic algorithm in order to solve this problem. Co-evolutionary genetic algorithms are inherently parallel in nature and they require high computation time. This high computation time can be reduced by exploiting the parallel architecture of graphics processing units (GPU). In this paper, compute unified device architecture (CUDA) provided for NVIDIA GPU is used. Experimental results demonstrate that computation time can significantly be reduced and the algorithm is capable to find some good solutions in a feasible time bound.**

Keywords: GPU;CUDA; Co-evolutionary genetic algorithm; Crew-scheduling; Min-max optimization.

I. INTRODUCTION

The scheduling of resources in the transportation industry is very complex and is a time consuming process. For instance, the airline industry faces largest scheduling problems in its daily operations [1]. The main problem every airline must solve is to construct an optimal schedule with the most efficient use of aircraft and crew resources in the timetable at a minimum cost to achieve maximum revenue. It is a real time optimization problem which should be solved in a given time to prevent any propagation of the disruption [2]. Therefore, these difficult optimization problems deserve a great deal of attention. However, these problems can be effectively solved by formulating min-max optimization problems.

Min-max optimization problems are found in different areas, for instance, in game theory, scheduling applications, network design, mechanical engineering, constrained optimization and function optimization [3]. These problems are considered difficult to solve deterministically in polynomial time bound. However, co-evolutionary genetic algorithms are found to be reliable to solve min-max optimization problems [4]. Co-evolutionary genetic algorithms usually operate on two or more populations of individuals. The populations evolve independently, but they are coupled together through fitness evaluation. The fitness of an individual in one population is evaluated on its performance against the individuals in the other population [5].

Using co-evolutionary genetic approach to solve min-max optimization problems requires high computation time to evaluate fitness of the individuals of each population [6]. Let us consider two populations each containing n individuals. The number of objective function calls required to evaluate each population is n^2. The individuals of one population must be evaluated against all the individuals with the other population. The evaluation of population needs to be performed in a number of times during the optimization process which is the most time consuming process [7]. Hence, this approach suffers from scalability problems if n is large or the objective function is complex. However, this scalability can be improved by exploiting the parallelism of GPU. In the proposed approach, fitness evaluation of each individual of one population against every individual in the other population is done in parallel. As a result, we can achieve a significant reduction in the computation time.

The remainder of the paper is organized as follows: in Section 2, the basic definition of min-max optimization problem is discussed. Section 3, describes the problem definition and co-evolutionary genetic algorithm. In Section 4, the implementation of the algorithm in CUDA architecture to solve the crew-scheduling problem has been discussed. Section 5 presents the experimental results and Section 6 concludes the paper.

II. LITERATURE REVIEW

Min-Max optimization problems allow one to find solutions by using scenarios to structure uncertainty [8]. In general, it can be defined as follows. Let's consider X is a set of all solutions and S is the set of all possible scenarios. If $F(x,s)$ is considered to be the cost of a solution $x \in X$ in a scenario $s \in S$, then the task is to find some solutions which can minimize this cost over

some scenarios. This is same as minimizing the maximum cost. According to this, the problem can be defined as follows:

$$\min_{x \in X} \max_{s \in S} F(x, s) \quad (1)$$

From [9], we find that the min-max problems were originally formulated by game theorists, which can be seen as an antagonist game where two players have a set of options. The player trying to find the solution *x*, tries to minimize the cost, while the player determining the scenario *s*, tries to maximize the cost. Later, these min-max problems were studied mathematically by many researchers. However, this problem is well suited with co-evolutionary genetic algorithms, since we can generate two different populations for *x* & *s* from two different search spaces.

III. PROBLEM DEFINITION & ALGORITHM

A. *Crew-Scheduling Problem Definition*

Given a set of trips that an agency must manage, the crew scheduling problem is to assign the trips to each crew in such a way that no trips are left unassigned in a given time. Let $\{T_1, T_2, T_3, \ldots\ldots T_n\}$ be the set of trips. Each trip T_i has a minimum processing time p_i and a maximum processing time q_i where $0 < p_i < q_i$. There are *m* crews $C_1, C_2, C_3, \ldots\ldots C_m$. We consider decision binary variables x_{ik} where,

$x_{ik} = 1$ [if T_i is assigned to crew C_k]

$x_{ik} = 0$ [if T_i is not assigned to crew C_k]

An assignment *x* is a feasible assignment (solution) if, for each trip T_i,

$$\sum_{k=1}^{m} x_{ik} = 1 \quad (2)$$

Let *X* be the set of all possible solutions. A scenario *s* is the combination of processing times. Thus $s = (p_1^s \ldots\ldots p_n^s)$. For each trip T_i, $p_i \le p_i^s \le q_i$, *S* is the set of all possible scenarios. *F(x,s)* is the time or cost of a solution *x* in scenario *s*. The cost is the maximum processing time to assign a trip to crew. Thus, we can define it as follows:

$$F(x, s) = \max_{1 \le k \le m} \left(\sum_{i=1}^{n} x_{ik} p_i^s \right) \quad (3)$$

Now the problem is to minimize this cost which can be formulated as min-max problems discussed earlier:

$$\min_{x \in X} \max_{s \in S} F(x, s) \quad (4)$$

We use co-evolutionary genetic algorithm to solve this problem.

B. *Co-evolutionary genetic algorithm*

The co-evolutionary genetic algorithm maintains two populations. The basic steps of the algorithm are illustrated below:

1. Initialize population *A* and population *B* at *t = 0 /* Initialize Populations */*
   / for i=1 to Maxitetrations */*
   / for j=1 to Maxgeneration1 */*
2. For each individual $x \in A(t)$, we evaluate *h(x) = max(F(x,s): s ϵ B(t))*
   / fitness evaluation */*
3. Create new generation *A(t+1)* by reproduction, crossover and mutation
   / end of Maxgeneration1 for loop */*
   / for k=1 to Maxgeneration2 */*
4. For each individual $s \in B(t)$, we evaluate *g(s) = min(F(x,s): x ϵ A(t))*
   / fitness evaluation */*
5. Create new generation *B(t+1)* by reproduction, crossover and mutation
   / end of Maxgeneration2 for loop */*
6. *t = t+1,* Unless *t* equals the maximum number of iterations go to step 2
   / end for loop of Maxiterations */*
7. Return the best solutions

From the Algorithm, we can observe that the number of evaluations of populations is *Maxiterations(Maxgeneration1 + Maxgeneration2).* Considering *n* individuals in both populations, the number of evaluations of the objective function per populations is n^2. This evaluation is done in conventional CPU, which is also known as sequential evaluation. The formulation can be written as following:

$$Eval_{seq} = Maxit(Maxgen1 * n^2 + Maxgen2 * n^2) \quad (5)$$

For instance, if we consider *Maxiterations = 100, Maxgeneration1 = 10, Maxgeneration2 = 10, n = 50, we have* $Eval_{seq} = 5000000$. This computation is inevitable, but it is possible to evaluate the fitness of all *n* individuals in parallel. For this case, we can write the formulation as following:

$$Eval_{parl} = Maxit(Maxgen1 * n + Maxgen2 * n) \quad (6)$$

Considering the same parameters we have $Eval_{parl} = 100000$, which shows a clear advantage over sequential evaluation. Hence, we can obtain a significant reduction in computation by exploiting the parallel advantage of GPU.

IV. IMPLEMENTATION OF THE ALGORITHM USING CUDA

A. *Basic Concepts of GPU & CUDA*

The Graphics Processing Unit (GPU) has emerged as a powerful computing device in this era of technology. The operational speed of GPU is much faster than the CPU (Central Unit Processing) [11]. The CPU mainly concentrates on arbitrary operations whereas on-the-other side GPU mainly concentrates on performance optimization related tasks. NVIDIA developed a software platform named compute unified device architecture (CUDA) for programmers to code in GPU. The language syntax consists of extensions of basic C-Language [10]. The depiction of

basic CUDA architecture has been illustrated in Fig. 1. To support the parallelism of a program: threads, blocks and grids are used. In CUDA the functions are grouped into three categories: The *host* functions, which are called and executed only by the CPU. These functions are similar to those implemented in C. The *kernel* functions are only executed by the GPU device and called by the CPU. We have to use the qualifier, __global__, for this type of function. The return type of this type of functions is always void. Finally, there are device functions, which are both called and executed only by the GPU device. The qualifier, __device__, precedes the function definition for this type of functions. In case of device functions, it is allowed to return any type of value [11].

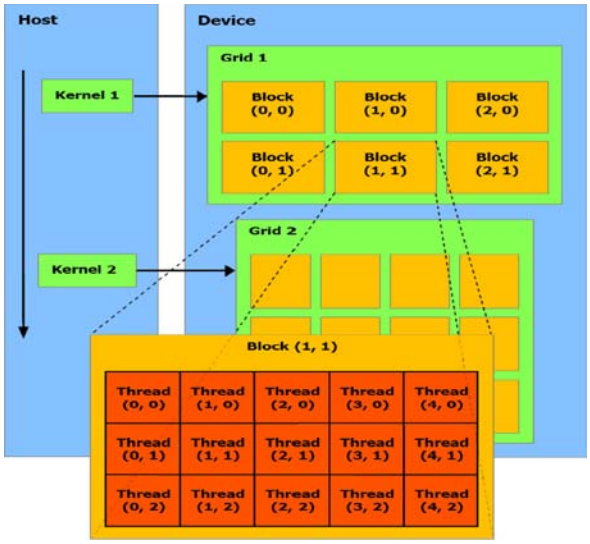

Figure 1. Basic CUDA Architecture

B. *Problem Formulation*

To evaluate the performance of this algorithm, first we try to find the worst-case scenario. For a solution, one of the worst-case scenarios is the one when all the trip assignments require maximum processing time. Thus, if we can reduce this worst-case scenario as much as we can, then we can create an optimal schedule. We define a lower bound on this algorithm. It is defined as follows:

$$L = \frac{1}{m}\sum_{i=1}^{n} q_i \qquad (7)$$

where q is the maximum processing time required for m crews. In addition, we formulate some problem sets to evaluate the performance of the algorithm. We consider two variables α_1 & α_2 to govern the processing time. The minimum processing time p_i is selected from a uniform distribution with the range [5, 20 α_1], where we set some arbitrary values for α_1. The maximum delay time d_i is selected from a uniform distribution with the range [0, α_2 p_i], where we set some arbitrary values for α_2. Hence, the maximum processing time is $q_i = p_i + d_i$. We try to find some good solutions closer to the lower bound as mentioned in Eq. (1).

C. *Implementation of the Algorithm in CUDA*

The main objective of this work is to minimize the processing time and generate an optimal schedule. In order to observe the robustness, we implement this algorithm in both sequential (CPU) and parallel (GPU) environment. The coding structure for these two environments is similar except the fitness evaluation and population generation part. Hence, in this section we discuss about these two parts regarding to GPU by which we can achieve parallelism. The algorithm first initializes the populations and then calculates the fitness in both randomly initialized populations. To note, we consider processing time as Population *A* and solution as Population *B*. Fig. 2 depicts the code which is responsible for invoking the kernel fitness function.

```
Fitness<<<nPop,nPop>>>(oldPopA,oldPop
B,nPop,nPar);
……..
__global__ void Fitness(population A,
population B, int nPop, int nPar){
     __shared__ float fitness[NPOP];
    int popAid = blockIdx.x;
    int i;
    int popBid = threadIdx.x;
    float fit2 = mp( A[popAid].s,
B[popBid].g);
    fitness[popBid] = fit2;
        __syncthreads();
    if(popBid==0){
       float min = 100000;
       for(i=0; i<nPop; i++){
           if(fitness[i]<min){
              min = fitness[i];
           }
       }
       B[popAid].fit=min;
    }else if(popBid==1){
       float max = -100000;
       for(i=0; i<nPop; i++){
           if(fitness[i]>max){
              max = fitness[i];
           }
       }
       A[popAid].fit=max;
        }
}
```

Figure 2. Sample CUDA code for Kernel Fitness Function

The computation is done in parallel to calculate the fitness for each individual of populations A and B. Here each block indexed by blockIdx calculates the fitness of the individual in its corresponding threads indexed by threadIdx. This evaluation is done in two populations concurrently. However, there is some

redundancy in calculations. This is because; each evaluation is done twice since shared memory is block-wise. We avoid this redundancy by utilizing global memory. Fig. 3 shows the corresponding CUDA code which is responsible for the generation of the new individuals of a population.

```
generateA<<<nPop,nPop>>>(newPopA,
oldPopA, oldPopB, cudaShuffle, nPop,
nPar, RAND, mut_rate, cross_rate,
bound, nRes);
……..
__global__ void generateA(population
newPopA, population oldPopA,
population oldPopB, int* shuffled,
int nPop, int nPar, float* RAND,
float mut_rate, float cross_rate,
float *bound, int nRes){
    individual trialVector;
    int i, r1, r2, r3, j;
    __shared__ float fitness[NPOP];
    i = blockIdx.x;
    j = threadIdx.x;
    // uniform crossover
    if(j==0){

Crossover(oldPopA,newPopA,&trialVecto
r,i,nPar cross_rate, RAND);
……..
    }
    // mutation
    mutation(shuffled, nPop, &r1,
&r2, &r3, RAND);

makeTrial(oldPopA,&trialVector,r1,r2,
r3, mut_rate,nPar);
        __syncthreads();
    fitness[j]=proctime(newPopA[i].s,
oldPopB[j].g);
    __syncthreads();
    int k;
    float max = -100000;
    for(k=0; k<nPop; k++){
       if(fitness[k]>max){
           max = fitness[k];
       }
    }
    newPopA[i].fit = max;

    if (oldPopA[i].fit <
newPopA[i].fit){

       copyIndividual(&oldPopA[i],&ne
wPopA[i],nPar,nRes);
    }
}
```

Figure 3. Sample CUDA code for Kernel generation Function

The kernel function responsible for Population *A* generation and Population *B* is similar. Here, we represent for the case: Population *A*. The crossover needs an array to store the individuals to perform the operation and they are stored in the array named *trialvector*. This is generated by the host and indexed using the *threadId*. After mutation and picking the trial solutions the most fit values are stored in the required processing time by calling the device function *proctime()*. In this way, we can utilize the advantage of GPU for co-evolutionary genetic algorithm to ensure parallelism.

V. EXPERIMENTAL RESULTS

A. *Experimental Environment*

To evaluate the algorithm performance we have carried out our simulations for sequential and parallel cases. To perform parallel evaluation we carried our experiment on linux server using NVIDIA GeForce GTX 580 driver which supports the CUDA compiler. The system specifications of the driver are listed in Table 1. To perform sequential evaluation we carried our experiment on Intel Pentium G620 processor working at 2.60 GHz clock speed. We adjust our machine to operate at 32-bit operating system including 8GB RAM memory.

Table 1. System Specifications for Parallel Evaluation

Parameter	*Values*
CUDA version	4.2
Global memory	3 GB
Local memory	48KB
Warp size	32
Maximum number of threads per block	1024
Maximum size of each Dimension of a block	1024x1024x64
GPU clock speed	1.54 GHz

Table 2, lists the parameters considered for co-evolutionary genetic algorithm.

Table 2. Parameters Considered for Co-evolutionary Genetic Algorithm

Parameter	*Values*
Maximum number of Iterations	100
Maximum number of allowable generations	100
Population size	50
Number of bits required by a string to represent an individual	64
Number of genes in a String	16
Number of bits restriced for a gene	4
Crossover rate	0.7
Mutation rate	0.003

Recall that the main objective of this work is to minimize the computation time by exploiting the GPU parallelism. Hence, we calculate the algorithm execution time in both CPU and GPU environments to observe the speedup. It is an indication of how much faster a parallel processing is over its counterpart, i.e., sequential processing [12]. The speed up is calculated by the following equation within the defined time bound.

$$Speedup = \frac{T_{seq}}{T_{parl}} \quad (8)$$

Here T_{seq} and T_{parl} are the total time taken by the sequential processing and parallel processing of the algorithm.

B. *Experimental Results*

To evaluate the algorithm, we compare our solutions to the lower bound as we defined earlier. In addition, the algorithm is able to create an optimal schedule by satisfying the requirements. In order to address this issue, we devise a schedule as illustrated in Table 3. In this schedule, we assume a normalized value to 1 to observe the performance of the algorithm. If the solutions can achieve closer to 1 then we can state that the algorithm have a good lower bound, and it can generate good solutions.

Table 3. Results for the Crew-Scheduling Problem

Values considered for governing the processing time	*Algorithm Performance for Sequential Evaluation (CPU)*	*Algorithm Performance for Parallel Evaluation (GPU)*
$\alpha_1 = 0.2$ $\alpha_2 = 0.6$	1.12	1.00
	1.15	1.01
	1.09	1.00
	1.08	1.03
	1.01	1.02
	1.07	1.03
	1.09	1.05
	1.02	1.00
	1.03	1.00
	1.10	1.02

From Table 3, we can observe that the algorithm performs well when it is performed in parallel situation rather than sequential evaluation. Here, we consider one problem set in which 10 instances are considered. In 1 instance, we consider a set of crews those are being assigned to their available trips in the processing time. The objective can be achieved by finding good lower bounds for the algorithm. Here, the values those are closer to the normalized value 1, are considered as good solutions. Hence, we can achieve an optimal schedule by minimizing the maximum processing time which exploits the parallel advantage of GPU.

The speed up achieved by the parallel evaluation is compared with sequential evaluation. To conduct this task, we fix every parameter except varying the generations. The results are depicted in Table 4. From Table 4, we can observe that, the computation time can be reduced significantly with parallel implementation to achieve robust solutions.

Table 4. Algorithm Execution Comparison

Number of generations	*T_{seq}[CPU (s)]*	*T_{parl}[GPU (s)]*	*Speedup*
20	6.721	0.099	67.88x
40	7.821	0.212	36.89x
60	10.38	0.582	17.83x
80	12.17	0.841	14.47x
100	14.69	1.061	13.84x

From Table 4, we can observe a reduction in speedup while we increase the generations. This is because; the search space increases while we increase the number of generations, which affects the algorithm execution time. The algorithm execution time is lower, when it creates an optimal schedule by finding good solutions in less number of generations compared to higher number of generations. Overall, the approach helps us to achieve our goal.

VI. CONCLUSION

This paper presents a co-evolutionary genetic algorithm to solve crew-scheduling problem by formulating min-max optimization problems. The main time consuming area of the co-evolutionary genetic algorithm is fitness evaluation of the individuals. In case of serial implementation, the evaluation of fitness function requires $O(n^2)$ time. The parallel evaluation of the fitness of the individuals brings down this time to $O(n)$. Finally, we can state that, the highly constrained real world optimization problems can be solved easily by inheriting the parallel advantage of GPU.

REFERENCES

[1] X. Chen, X. Chin, and X. Zhang, "Crew scheduling models in airline disruption management", In Proceedings of the 17th IEEE International Conference on Industrial Engineering and Engineering Management (IE & EM), pp. 1032-1037, 2010. **(Conference)**

[2] T. H. Yunes, A.V. Moura, and C. C. d. Souza, "Solving very large crew scheduling problems to optimality", In Proceedings of the ACM symposium on Applied computing, pp. 446-451, 2000. **(Conference)**

[3] T. Alamo, D. M. de la paena, and E. F. Camacho, "An Efficient Maximization Algorithm with implications in Min-

Max Predictive Control," IEEE Transactions on Automatic Control, vol. 53, no. 09, pp. 2192-2197, 2008. **(Journal)**

[4] K. Deb, S. Gupta, J. Dutta, and B. Ranjan, "Solving dual problems using a coevolutionary optimization algorithm," Journal of Global Optimization, 2012. **(Journal)**

[5] A. M. Cramer, S. D. Sudhoff, and E. L. Zivi, "Evolutionary Algorithms for Minimax Problems in Robust Design," IEEE Transactions on Evolutionary Computation, vol. 13, no. 02, pp. 444-453, 2009. **(Journal)**

[6] L. P. Veronese, and R. A. Krohling, "Differential evolution algorithm on the GPU with C-CUDA," In Proceedings of the IEEE International Conference on Evolutionary Computation, pp. 18-23, 2010. **(Conference)**

[7] A. K. Qin, V. L. Huang, and P. N. Suganthan, "Differential Evolution Algorithm With Strategy Adaptation for Global Numerical Optimization," IEEE Transactions on Evolutionary Computation, vol. 13, no. 02, pp. 398-417, 2009. **(Journal)**

[8] L. Liu, and Y. Zhang, "Gemetic Algorithm design of the Min-max weighted distance problem," In Proceedings of IEEE International Conference on Computer Application and System Modeling (ICCASM), pp. 620-623, 2010. **(Conference)**

[9] K. Masuda, K. Kurihara, and E. Aiyoshi, "A novel method for solving min-max problems by using a modified particle swarm optimization," In Proceedings of the IEEE International Conference on Systems, Man, and Cybernatics (SMC), pp. 2113-2120, 2011. **(Conference)**

[10] NVIDIA Homepage. Available: http://www.nvidia.com **(link)**

[11] C. C. Boyer, M. Meng, J. Tarjan, S. Sheafler, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," Journal of Parallel and Distributed Computing, vol. 68, no. 10, pp. 1370-1380, 2008. **(Journal)**

[12] J. Verdu, A. Pauelo, and M. Valero, "The Problem of Evaluating CPU-GPU Systems with 3D Visualization Applications," IEEE Micro, vol. 32, no. 06, pp. 17-27, 2012. **(Journal)**