Speeding Up DSMGA-II on CUDA Platform

Sung-Chi Li
Taiwan Evolutionary Intelligence Laboratory
Department of Electrical Engineering
National Taiwan University
r04921056@ntu.edu.tw

Tian-Li Yu
Taiwan Evolutionary Intelligence Laboratory
Department of Electrical Engineering
National Taiwan University
tianliyu@ntu.edu.tw

ABSTRACT

This paper proposes two CUDA based implementations to speed up the model building process for DSMGA-II, which has shown superior optimization ability to hBOA and LT-GOMEA on various benchmark problems. The first implementation is lossless, which is algorithmically identical to the original version. The second implementation is lossy, which sacrifices some accuracy for further speedup. On several commonly used benchmark problems, the proposed implementations are stable. As the problems become larger, the amount of speedup increases accordingly. The lossless scheme speeds up the first part of model building for more than ten times; the lossy implementation further speeds up the second part of model building for more than 400 times on a 600-bit folded-trap problem. The limitation of such implementations are also discussed in detail in this paper.

KEYWORDS

Evolution strategies, Genetic algorithms, Parallelization

ACM Reference format:
Sung-Chi Li and Tian-Li Yu. 2017. Speeding Up DSMGA-II on CUDA Platform. In *Proceedings of GECCO '17, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3071178.3071228

1 INTRODUCTION

The dependency structure matrix genetic algorithm-II (DSMGA-II) is a model building genetic algorithm (GA) proposed by Hsu and Yu in 2015 [3]. DSMGA-II constructs a dependency structure matrix (DSM) representing the linkage information by detecting the mutual information between variables. Then DSMGA-II extracts the linkage model (or masks) from the DSM and gradually puts solution fragments together. DSMGA-II has shown superior optimization ability to hierarchical Bayesian optimization algorithm [9] and linkage tree gene-pool optimal mixing evolutionary algorithm [1, 15, 16], two state-of-the art model-building GAs, on various benchmark problems. The performance of DSMGA-II highly relies on the DSM and the linkage model; however, the constructions of DSM and the linkage model can be time consuming for large-scale problems.

Fortunately, nowadays the graphic processing unit (GPU) is an affordable parallel processor. With the concept of general purpose GPU (GPGPU), NVIDIA released the compute unified device architecture (CUDA) [7], which is an extension of C/C++ programming with heterogeneous computing model. Developers write serial programs and launch hundreds of thousands of threads to process data when the program executes to a part which can be executed in parallel. After all works on GPU are done, the results are collected, and the program continues the serial part, and so on. We think that GPU may be a good tool to speed up the model building of DSMGA-II.

Although computing the fitness function can be also very time-consuming, it is beyond the scope of this paper for the following reasons. According to the work of Seo *et al.* [12], GPU is not good at branch predicting by nature, which can make the speedup become counterproductive depending the problem. In addition, since GA is a block-box optimizer, it is impractical to customize a speedup method for every single fitness calculation.

Researchers have implemented many evolutionary strategy algorithms on CUDA, such as the particle swarm optimization [6, 14], the differential evolution [4, 8], and GA [10]. Contrarily, there are relatively much fewer researches concerning speeding up the model building process of model building genetic algorithm (MBGA), probably because model building is more centralized and hence is not easily parallelized or distributed by nature. Shao and Yu used CUDA to speed up the model building process on ECGA [13]. Poulding *et al.* proposed a CUDA based Bayesian optimization algorithm [11].

In this paper, we propose two CUDA based implementations aiming to speed up the model building in DSMGA-II. The first implementation, called the lossless scheme, is algorithmically identical to the original DSMGA-II, by launching hundreds of thousands of threads and adopting the CUDA API to speed up the constructions of the DSM and the linkage model. The second implementation, called the lossy scheme, sacrifices some accuracy in building linkage model for further speedup. The remainder of this paper is organized as follows. Sections 2 and 3 give introductions to DSMGA-II and CUDA, respectively. Sections 4 and 5 details the lossless and the lossy implementations, respectively, as well as the ideas behind such designs. Section 6 shows the empirical results and discussions, followed by Section 7, which concludes this paper.

2 DSMGA-II

In this section, we briefly introduce the framework of DSMGA-II and give the flow of the algorithm. We provide a more detailed introduction for the DSM building and linkage model building here because the performances of these two parts are improved more by GPU.

2.1 Framework of DSMGA-II

DSMGA-II is composed of 3 stages: the initializing, the restricted mixing and the back mixing. First, it randomly initializes a population and performs hill climbing on each chromosome in the initializing stage. This procedure reduces the possible patterns, which appears in a building block, and reduces noises. Therefore hill climbing enhances the accuracy to recognize the linkages between bits. After that, DSMGA-II goes into a loop, breaks and terminates until some criteria are fulfilled.

In the loop, DSMGA-II selects the good individuals to build the DSM. After the DSM is built, we have enough information to build an incremental linkage set (ILS), which is the linkage model.

Next, DSMGA-II goes into the restricted mixing stage. In this stage, DSGMA-II randomly picks one chromosome and a randomly chosen gene as starting position. DSMGA-II then gets an order sequence by using the linkage set inside the ILS. The restricted mixing tries to flip bits on the picked chromosome according to the order sequence. During this process, the picked chromosome is treated as a receiver, and we check whether the flipped bit pattern exists in the population. If yes, at least one donor exists; otherwise, the mixing process of this chromosome is stopped and restarts from picking another chromosome. If the receiver exists, we use the optimal mixing (OM) operator proposed by Thierens *et al.* [16] to perform the donation process. If the OM succeeds, the algorithm steps into the back mixing stage. Otherwise, continues the order sequence flipping. In the back mixing stage, we believe that the pattern in the previous restricted mixing stage is good enough to make the donation succeed, so we try to donate that pattern to all of the chromosomes by OM.

The criteria of breaking the loop contain: finding the optimum, running out of the given number of function evaluations (NFEs), and running out of the given generations. After leaving the loop, DSMGA-II also terminates and returns the population that contain solutions, which we believe are good enough for us. The pseudo code is given in Algorithm 1

Algorithm 1: DSMGA-II

```
Input  : $\ell$: problem size, $n$: population size
Output : $\mathcal{P}$: population
$DSM$: dependency matrix, $s$: selection pressure, $R$: constant
randomly initialize $\mathcal{P}$
$\mathcal{P} \leftarrow$ RunLocalSearch ($\mathcal{P}$)
while $\neg$ShouldTerminate ($\mathcal{P}$) do
    $S \leftarrow$ TournamentSelection ($\mathcal{P}, s$)
    $DSM \leftarrow$ UpdateMatrix ($S$)
    $ILS \leftarrow$ FindClique ($DSM$)
    for $k = 1$ to $R$ do
        $\mathcal{I} \leftarrow$ random permutation from 1 to $|\mathcal{P}|$
        for $i = 1$ to $|P|$ do
            if RestrictedMixing($\mathcal{P}_{I_i}, ILS$) then
                $\mathcal{P} \leftarrow$ BackMixing $(\mathcal{P}_{I_i}, ILS)$
```

2.2 Building DSM

DSM is a graph representation of the dependency between the two genes, where each entry e_{ij} is the measure of gene i and gene j. The larger the e_{ij} is, the stronger the dependency between the two genes is. Here, DSMGA-II uses mutual information proposed by Kullback *et al.* [5] as a measurement, which is shown below:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}$$

2.3 Building Linkage Model

In DSMGA-II, the process of building linkage model is the FindClique in Algorithm 1. The output of FindClique is a list of incremental linkage set (ILS) with each gene as a starting node. ILS is a list of linkage set, and a linkage set is an approximation maximum-weight connected graph (AMWCS), which gives an order that for a linkage set L and index $i, j, i < j$, the L_i is the index such that $\sum_{k<i} I(L_i, L_k) > \sum_{k<j} I(L_j, L_k)$, where $I(L_i, L_k)$ is the mutual information between genes at loci of L_i and L_k. The order in a linkage set indicates the linkage strength from strong to weak about how a gene is connected to those genes in the linkage set with index smaller than itself.

3 CUDA

In this section, we briefly introduce the background and programming model of CUDA, thus mapping out a concept about how to implement CUDA program. Then we discuss the design constraints and the optimization guidelines, for those parts highly influence our algorithm design.

3.1 Background

Originally, GPU is designed for graphic rendering tasks such as texture rendering. GPU is composed of thousands of cores, thus we can launch hundreds of thousands of threads. Each thread is processed by a GPU core. When a thread is waiting for data, the GPU task scheduler switches the core to process another thread and hides the memory access latency, thus making these threads acting like they are executed concurrently.

Because GPU is specialized for highly parallel tasks with intensive computation, the ability of caching and flow control are far worse than CPU. Hence, GPU is usually used only in rendering graphics. Traditionally, for those developers who want to use GPU to do calculation, they have to map the non-graphics computation to graphic renderings.

Since the new design of GPU structure and the enhancement of GPU cores, the concept of GPGPU has aroused. CUDA is a programming interface of GPGPU announced by NVIDIA in 2007, aiming for an easier programming environment for combining the computation power of CPU and GPU to develop GPU program.

3.2 Programming Model

CUDA adopts the concept of heterogeneous computing. When a program is launched, CPU executes the program, generates data and copies the data to the global memory of GPU. When a section of execution can be highly paralleled, CPU launches a kind of function called the "kernel" function, which makes all of the cores on GPU

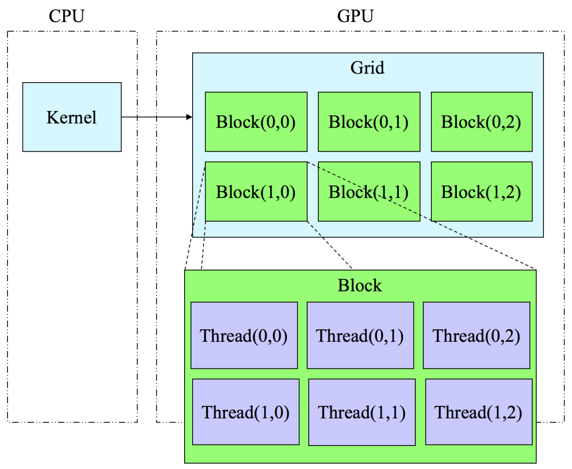

Figure 1: The hierarchical threads management in CUDA

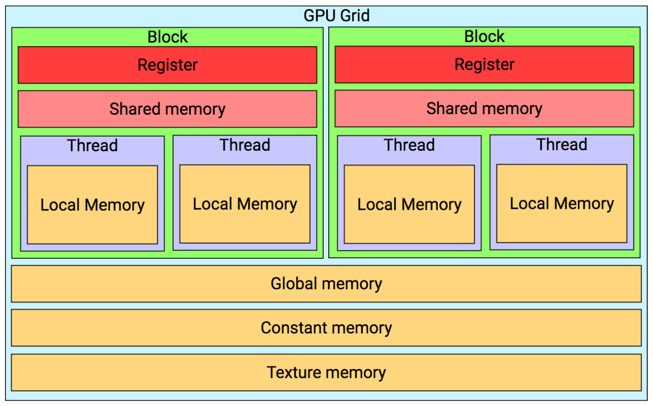

Figure 2: The memory architecture in CUDA. CPU only communicates with the global, constant and texture memory.

execute the same function and stores the computing results in GPU memory. The computing results are waiting for CPU to copy them back when needed and CPU continues the serial program.

The threads are managed in a hierarchical structure. The hierarchical structure is composed of three levels: thread, block and grid. The reason why to manage threads in this way is because the architecture of GPU itself. A basic processing unit in GPU is the streaming processor (SP); several SPs with some other parts forms a streaming multiprocessor (MP); a bunch of MPs forms a texture processing cluster (TPC). Usually, a GPU has 2 ∼ 6 MP. Roughly said, a kernel is a grid and is processed by the whole GPU. A block is processed by an MP, and a thread is processed by an SP. One thing to note is that 32 threads are packed as a "warp", and an MP processes a block each time with a warp. A grid can have three-dimensional blocks, and a block can have three-dimensional threads. Figure 1 shows the hierarchical structure.

There are 64KB L1 cache memory in each block. L1 cache memory is a high speed memory and can be more than 100 times faster than the global memory in GPU. A barrier function $\_\_syncthreads()$ can hold all threads in the same block to wait until all threads execute to that part.

3.3 Design Constraints and Optimization Guidelines

Due to the architecture of GPU, there are many design constraints: high speed memory and low speed memory in GPU, memory transfer between GPU and CPU, the high speed memory in each block only has 64KB, memory access conflict, and etc.

The bottleneck for most CUDA programs optimization is the memory access. CUDA contains six types of memories (see Figure 2). The global, constant and texture memory are large, but they are slow. Each of them has different size of cache. Each block has its own shared memory, which is about 100 times faster than global memory. However, the size of shared memory is only 64KB. Local memory belongs to each threads. Although local memory is drawn inside the block, it actually uses registers to store variables if the registers are not run out. If the registers are run out or the variable is declared as an array, the variable is stored in the global memory, which can be as slow as accessing global memory. However, there is a shared memory access conflict called bank conflict. Coalescing access global memory is an another way to enhance the performance of memory access.

As mentioned before, an MP only processes a warp at a time. Hence, if the total threads in one block can not be divided by 32, the remainder threads are packed as a warp. Thus there are some SPs are idle when processing that warp, which results in a waste. Also, more threads in a block can hide the memory access latency more. Hence, it is a good idea to launch more threads in one block. However, more threads in a block means that each thread can get less shared memory. Besides, more active blocks are distributed to different SM, this can utilize the GPU resources.

In CUDA architecture, data generated from CPU need to be copied to the global memory on GPU so that GPU is able to process them. The data transfer is through the PCI-Express bus, thus costing time to setup the communication and data transfer.

Using CUDA API is an another optimizing tip. Since CUDA API implements lots of function using native instructions on GPU, using CUDA API does improve the performance of a program significantly.

4 LOSSLESS SCHEME

This is the first implementation using CUDA to speed up the DSMGA-II. This scheme is algorithmically identical to the original version except the double precision error produced by the log function. In this section, we first have a glimpse about the flow and then introduce *FastCounting*, which is an idea proposed by [13] that reduces the storage to record the occurrence of each gene and to speed up the counting distribution. We then present our improved implementation on *FastCounting* to use more threads and blocks to do the calculation. Next, we discuss using CUDA to speed up the DSM building process. At the end of this section is about speeding up the process of building linkage model.

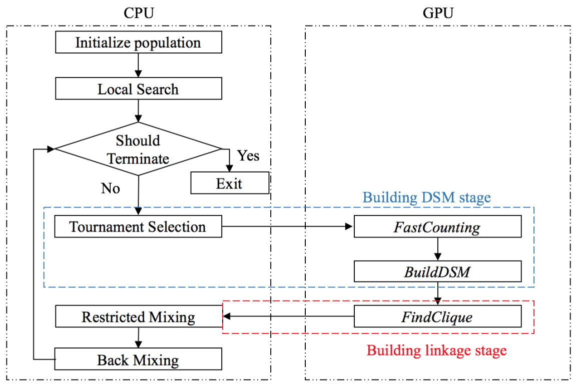

Figure 3: The flow chart of lossless scheme

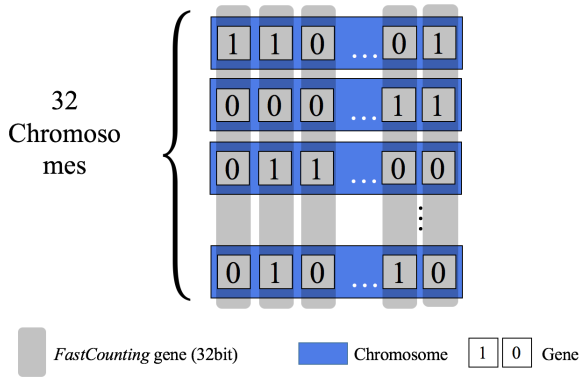

Figure 4: ***FastCounting*** **gene**

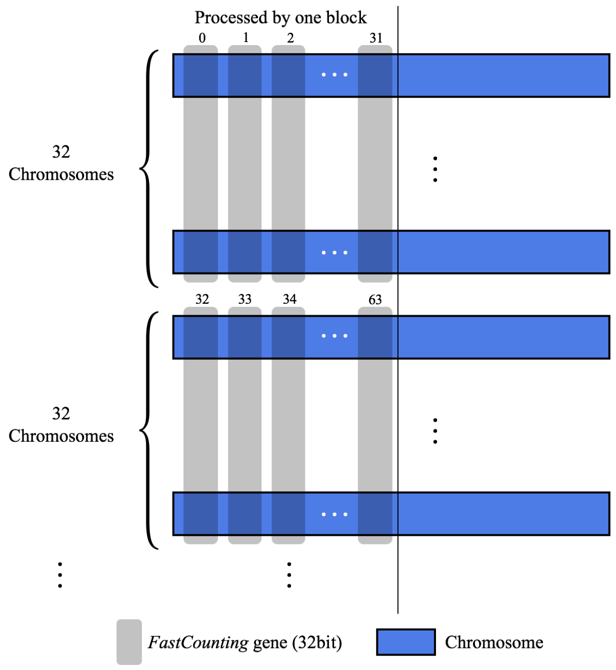

Figure 5: In CUDA, a thread only processes one *FastCounting* gene. The number along side the gene shows that it is process by the thread with id equal to the number.

4.1 Flow

As Figure 3 shows, we use GPU to speed up the *FastCounting*, *buildDSM* and *FindClique*. The data transfer between CPU and GPU happens in two parts: before starting the *FastCounting* and after *FindClique*. We decide to do tournament selection in CPU because the implementation of tournament selection is to generate permutation list, which the list size is equal to the selection pressure. Each permutation list size is equal to the population size. The generation of permutation list can not be highly paralleled. Besides, finishing the selection in CPU enables us to only copy a population with all of the selection winners. This implementation makes the accessing of the population in kernel functions simpler, and the access pattern is a coalescing memory access.

The GPU computing result, which needs to be copied back to CPU, is a two dimensional array, which is the ILS. The DSM built by the function *BuildDSM* becomes a mid product and is stayed in the GPU memory without copying back to CPU, which reduces a huge data transfer between CPU and GPU.

4.2 FastCounting

FastCounting uses a utility table to store the occurrence of each gene. For the ith entry in the utility table, it stores all chromosome's the ith genes value. When calculating the mutual information of the ith and jth gene, we need to calculate the occurrence of n_{11}, n_{10}, n_{01} and n_{00}. To calculate the number of occurrences of n_{11}, we only need to count how many 1s in ith entry as one_i, how many 1s in jth entry as one_j and XOR the ith and the jth entries, then calculate how many 1s in the XOR result as $one_{XOR_{ij}}$. At last, $n_{11} = \frac{one_i + one_j - one_{XOR_{ij}}}{2}$. Similarly, the occurrence of $n_{10} = one_i - n_{11}$, etc. In the implementation, each entry in the table is an integer array, and each bit in the integer represent a chromosome's gene value. Take Figure 4 for example, the first bit in the integer of the first *FastCounting* gene is the first gene value of fist chromosome's gene value, which is 1.

In CUDA, each thread calculates just one *FastCounting* gene. We also use an integer array to store each chromosome's gene. Each bit in the integer represents a gene value, so for a ℓ bit problem with population size n, we launch $\lceil \frac{n}{32} \rceil$ blocks and ℓ threads in each block. The total threads we launch is $\lceil \frac{n}{32} \rceil \cdot \ell$. The concept is drawn in Figure 5. Using this implementation, each thread calculates one *FastCounting* gene, which is 32, and each block calculates 32 rows of the utility table. To reduce the global memory access, we only need to access and cache 32 gene of each chromosome, which is at most $32 \cdot n$ for calculating the occurrence, and we can store the *FastCounting* gene in local registers for each thread only calculates one *FastCounting* gene, which is exactly 32 bits. With this implementation, we reduce the memory access latency and also launch appropriate blocks to make SM busy.

4.3 Speedup Building DSM

Given problem size = ℓ, the DSM is a matrix with size = $\ell \cdot \ell$, for entry e_{ij} stores the mutual information between gene i and gene j. In reality, the DSM is just a triangular matrix with all the diagonal entries value = 1, so we implement the DSM using one-dimensional double array with size $graphSize = \frac{\ell \cdot (\ell-1)}{2}$. If given the row i and column j, $i < j$, the index on the double array = $i \cdot (\ell - 1) - \frac{i \cdot (i+1)}{2} + j - 1$; if given the array index idx, we can follow the Algorithm 2 to transform the idx into row index i and column index j.

Algorithm 2: Index Convert

```
Input  : $\ell$: problem size, $idx$: array index
Output : $i, j$: row index and column index
$i \leftarrow 0$
$j \leftarrow 0$
$layerSize \leftarrow \ell - 1$
while $idx - layerSize >= 0$ do
    $i \leftarrow i + 1$
    $idx \leftarrow layerSize$
    $layerSize \leftarrow layerSize - 1$
$j \leftarrow idx + i + 1$
return $i, j$
```

In CUDA implementation, we launch $\frac{graphSize}{\ell} + 1$ blocks, and each block contains ℓ threads. At beginning of the kernel *BuildDSM*, we cache the whole utility table generated by *BuildFastCounting* to each block, then each thread computes one DSM entry. A thread uses its thread id as an array index with Algorithm 2 to decide which DSM entry to process. We use CUDA API *__logf*(), which is a function that returns the result of log function with single precision, to compute the mutual information. Using only single precision to store the mutual information of entry is enough because the size of the population is usually not that big, and the single precision error for one entry is acceptable for comparing relative value with the same row. However, it is necessary to use double precision in building linkage model, and we discuss it in the next subsection.

4.4 Speedup Building Linkage Model

In CUDA, because the size of DSM is so large that it can not be cached inside a block, we launch ℓ blocks with ℓ threads in each block and each block computes one ILS using block id as starting gene. We also use the technique *reduction* [2] to speed up finding the max connections index and cache the *ILS* and *connections*, then write the *ILS* back to global memory. The pseudo code is listed in Algorithm 3. In our implementation, although ILS is a set that is composed of linkage sets, each linkage set is just a subset of the next linkage set. Therefore, we only find the largest linkage set.

Some details worth noting is that although we use the technique *reduction* to find the *maxMIIdx*. We conserve the randomness to pick a random candidate when there are two index candidates with the same connection strength. The randomness can highly influence the stability of the DSMGA-II. Also, this is the reason why we use double precision to store the accumulated linkage value to reduce building the incorrect linkage model caused by accumulated precision error.

Algorithm 3: CUDA Construct ILS

```
Input  : $DSM$: dependency matrix, $\ell$: problem size
Output : $ILS$: ILS, integer array represent index
$used$: bool array, cached in shared memory
$connections$: double array cached in shared memory
$reductionMaxIdx$: int array, cached in shared memory
if $threadIdx == blockIdx$ then
    $used[threadIdx] \leftarrow True$
    $ILS[0] \leftarrow threadIdx$
else
    $used[threadIdx] \leftarrow False$
    $connections[threadIdx] \leftarrow DSM[blockIdx][threadIdx]$
$\_\_syncthreads()$
for $ILSSize = 1$ to $\ell$ do
    $DoReduction(reductionMaxIdx, connections)$
    $maxMIIdx \leftarrow reductionMaxIdx[0]$
    if $threadIdx == 0$ then
        $used[maxMIIdx] \leftarrow True$
        $ILS[ILSSize] \leftarrow maxMIIdx$
    $\_\_syncthreads()$
    if $used[threadIdx]$ then
        $connections[threadIdx] \leftarrow 0$
    else
        $connections[threadIdx] += DSM[maxMIIdx][threadIdx]$
```

Under this scheme, the algorithm is limited by the size of shared memory, which only depends on the problem size. The theoretical maximum problem size applicable in our implementation is 26,213 bits.

5 LOSSY SCHEME

The result of lossless scheme is shown in Figures 6 and 7. The speedup on building linkage model is very subtle. However, the building linkage model is actually the most time-consuming part. For example, a 400-bit concatenated trap with building block size 5, the time in building linkage model versus building DSM is about 16, with the population found by sweep method, which is proposed in [3]. Thus, we decide to sacrifice some accuracy on building linkage model for further speedup. In the original version, the *FindClique* output all ILS with each size ℓ, but we don't need to construct such long ILS. For example, in concatenated trap function with building block size 5, an ILS is used by the restricted mixing operator with the first 5-th linkage set under most circumstances and seldom with the first 10 linkage set. This gives us an inspiration that constructing an ILS with only moderate element size in *FindClique*.

The flow of the lossy scheme is the same as lossless scheme, but different inside the *FindClique* part. As mentioned before, the ILS is used only in the restricted mixing operator, and only when there is a complementary bit pattern exists in the population, the restricted mixing can try the donation. Thus, the idea of the lossy scheme is that when constructing an ILS, we randomly choose a representative from the current population. Every time before we take the next index to construct the next linkage set, we check whether there exists a complementary bit pattern in the current population; if yes, we append the linkage set into the ILS; otherwise, stop the construction. This scheme shows that the output ILS is a shrunk version of the original ILS. There are some advantages in the lossy scheme. First, it highly speeds up the procedure of *findClique* in one generation and is applicable to CPU version. Second, this method reduces the cross competition rate because of the small size of donation pattern. Third, donation pattern increases the efficiency of restricted mixing and back mixing. However, this method sacrifices correctness and the possibility that some restricted mixing procedure can success with longer ILS. The pseudo code is shown in Algorithm 4

Algorithm 4: CUDA Construct lossy ILS

```
Input  : $DSM$: dependency matrix, $P$: population, $\ell$: problem
         size, $n$: population size
Output : $ILS$: ILS, integer array represent index
$used$: bool array, cached in shared memory
$connections$: double array cached in shared memory
$reductionMaxIdx$: int array, cached in shared memory
$hasDonor$: bool variable, a shared variable used by whole block
if $threadIdx == blockIdx$ then
    $used[threadIdx] \leftarrow True$
    $ILS[0] \leftarrow threadIdx$
else
    $used[threadIdx] \leftarrow False$
    $connections[threadIdx] \leftarrow DSM[blockIdx][threadIdx]$
$\_\_syncthreads()$
for $ILSSize = 1$ to $\ell$ do
    $DoReduction(reductionMaxIdx, connections)$
    $maxMIIdx \leftarrow reductionMaxIdx[0]$
    if $threadIdx == blockIdx$ then
        $hasDonor \leftarrow False$
    $\_\_syncthreads()$
    if $threadIdx$ ¡ $n$ then
        if $isComplement(threadIdx, ILS, maxMIIdx)$ then
            $hasDonor \leftarrow True$
    $\_\_syncthreads()$
    if $hasDonor$ then
        if $threadIdx == 0$ then
            $used[maxMIIdx] \leftarrow True$
            $ILS[ILSSize] \leftarrow maxMIIdx$
        $\_\_syncthreads()$
        if $used[threadIdx]$ then
            $connections[threadIdx] \leftarrow 0$
        else
            $connections[threadIdx] += $
            $DSM[maxMIIdx][threadIdx]$
    else
        break
```

Under this scheme, the algorithm is also limited by the size of shared memory, which depends on the problem size and population. If we assume the population size is equal to problem size, the theoretical applicable max problem size is 24,422 bits.

6 RESULT

In this section, we present our experimental results. First, we provide our hardware specification and describe our test problems. Next, we show our speedup results of the lossless scheme and end this section with the results of lossy scheme.

6.1 Hardware Specification

We conduct the experiment on a computer with a 4 cores Intel i5-2400 CPU at 3.10 GHz, and an NVIDIA GTX 980 Ti with 6 GB of DDR5 global memory. The compute capability of the GPU card is 5.2 and it has 2816 CUDA cores at 1.10 GHz max clock rate. The memory interface width is 384-bit with memory clock rate 3305 MHz. The operation system is Debian with kernel version 3.16.0. The driver version is NVIDIA-DRIVERS 375.26 with CUDA toolkit 8.0.

6.2 Test Problems

The test functions are chosen from the original paper of DSMGA-II and are listed as follows:

6.2.1 *Concatenated trap (MK).*

The concatenated trap is composed of m equal contributed non-overlaping trap function. Each subfunction contains k variables. The number of variables of the concatenated trap function is $m \cdot k$. The fitness function is listed in the following equation:

$$f_{m,k}(x) = \sum_{i=1}^{m} f_k^{trap}\left(\sum_{j=i\cdot k-k+1}^{i\cdot k} x_j\right)$$

where

$$f_k^{trap}(u) = \begin{cases} 1 & \text{if } u = k, \\ 0.8 \cdot \frac{k-1-u}{k-1} & \text{otherwise} \end{cases} \tag{1}$$

6.2.2 *Folded trap (FTRAP).*

The folded trap function in our experiment is composed of several bipolar trap function with $k = 6$. A bipolar trap function contains two global optima and a number of local optima. The fitness function in our experiment is expressed as:

$$f_{m,k=6}(x) = \sum_{i=1}^{m} f_{k=6}^{folded}\left(\sum_{j=i\cdot k-k+1}^{i\cdot k} x_j\right)$$

where

$$f_{k=6}^{folded}(u) = \begin{cases} 1 & \text{if } |u-3| = 3, \\ 0.8 & \text{if } |u-3| = 0, \\ 0.4 & \text{if } |u-3| = 1, \\ 0 & \text{otherwise} \end{cases}$$

6.2.3 *Cyclic trap (CYC).*

Cyclic trap is composed of overlapping trap functions with wraparound. It is similar to the $m \cdot k$ concatenated trap function. All subfunctions inside are the trap function, but all of them are overlapped by one bit.

$$f_{m,k}(x) = \sum_{i=1}^{m} f_k^{trap}\left(\sum_{j=i\cdot(k-1)-k+2}^{i\cdot(k-1)+1} x_j\right)$$

where f_k^{trap} is the same as Equation 1.

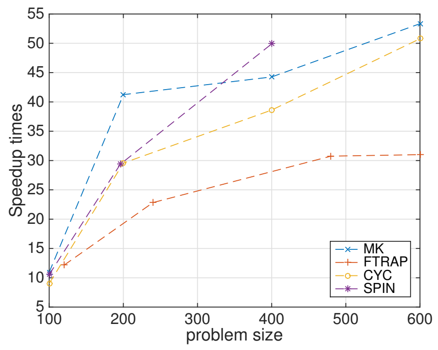

Figure 6: The results of building DSM speedup times in lossless scheme.

6.2.4 Ising spin-glass (SPIN). The Ising spin-glass is a famous problem of statistical mechanics. Given a set of spins, each of them is in one of two states of {+1, -1}. For any two adjacent spins i and j, there is a coupling constant J_{ij}. The objective is to find a set of spin assignment such that the fitness function is maximum. The fitness function is listed below:

$$f_n(x) = -\sum_{i,j=0}^{n} x_i \cdot x_j \cdot J_{ij}$$

We use a 2-D grid spin-glass with $J_{ij} \in$ {+1, -1} in our experiment.

6.3 Results of Lossless Scheme

The speedup results are shown in Figures 6 and 7. Each data point is tested under the population size averaged with 10 population found by sweep method, which requires 10 consecutive times finding global optima, with average of 100 times. There is only one instance running on the whole machine with no other heavy load process at the same time. We present the speedup in two stages (Figure 3). The first stage is the DSM model building, which starts from selection, including data transfer between CPU and GPU, and ends at finishing the *buildDSM*. The second stage is the linkage model building, which starts after the *BuildDSM*, and ends after transferring the whole ILS back to CPU. We record the time spent in each stage, and cumulate them in each stage respectively through every generation. We also use CUDA APIs *cudaEventRecord* and *cudaEventSynchronize* to synchronize CPU and GPU, then get the precise running time. The APIs return a accuracy level of millisecond.

As the results show, both stages gain more speedup as the problem size grows. This is because we can launch more threads in larger problem size, and the cost of data transfer between GPU and CPU becomes less significant. We show the speedup applies to all test functions stably and gain more speedup in *BuildDSM* but not very much in *FindClique*. The reason why speedup ratio is different within problems is that the convergence level of gene is different, thus influence the need of log function computation. The gene is usually converged after the first generation. Thus, for problems that needs more generation to solve, the speedup gain becomes less significant. The speedup bottleneck in *FindClique* is the global memory access and several *__syncthreads*().

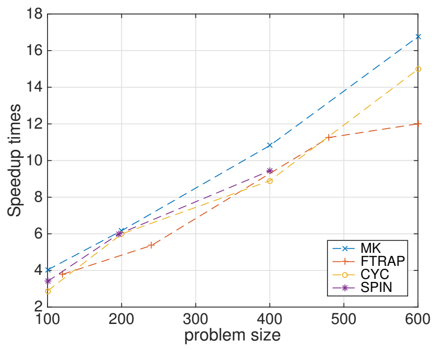

Figure 7: The results of building linkage model speedup times in lossless scheme.

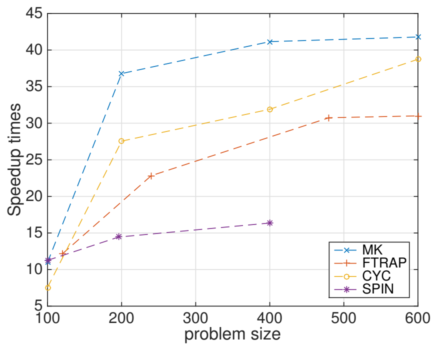

Figure 8: The results of building DSM speedup times in lossy scheme.

6.4 Results of Lossy Scheme

The speedup results are shown in Figures 8 and 9. With this scheme, the generation is usually slightly larger than the original version, and the NFEs are almost the same (see Figure 10). The speedup results of building DSM model are almost the same as lossless scheme, but most speedup results are slightly smaller because of the larger generation. The speedup results of building linkage model are huge breakthroughs, and the greatest speedup ratio is up to 400.205 times. This shows that if the model building algorithm is properly designed for parallel scheme, it benefits from the huge speedups.

7 CONCLUSION

In this paper, we used CUDA to speed up the model building process for DSMGA-II. We first proposed a lossless scheme, which aims at building DSM and building linkage model, to speed up the algorithm itself. We got speedups that is at least ten times faster when building DSM, but only subtle speedups in building linkage model. Besides, the speedup also varied for different problem, and we made an explanation about this phenomenon. Next, since the process of building linkage model is one of the most time-consuming parts, we proposed another lossy scheme for further speedup, and we got significant breakthroughs. Nevertheless, this modification slightly

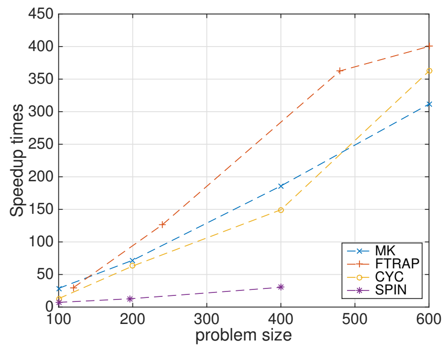

Figure 9: The results of building linkage model speedup times in lossy scheme.

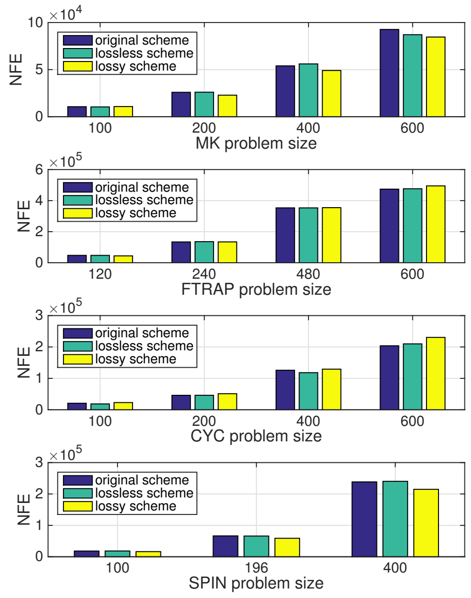

Figure 10: The NFE comparison between origianl, lossless and lossy scheme. The NFE is tested under the population found by sweep method which requires 10 consecutive hit. The lossless scheme is algorithmically the same as the original scheme, so the NFE is almost the same. The NFE of lossy and original scheme are neck and neck, but the relative NFE differences are still small. For the largest relative NFE difference between original and lossy scheme is on the CYC 600-bit with 13.12%

increased the generation in most problems, and cause the speedup in building DSM to slightly decrease. In the lossy scheme, the costs of NFEs are almost the same as the original version.

As for the future work, we would like to combine the parallelism of CPU with GPU. Because for some operations, for example, calculating the fitness function on GPU may be counterproductive, we may use the parallelism of CPU to speed up these operations, and gain more speedup for the whole program. We also want to further investigate the lossy scheme because of the advantages we mentioned before, hoping to develop a new technique on DSMGA-II not only to improve the computational performance, but also to reduce NFEs.

Traditional GAs are highly parallelizable; however, the model-building process in MBGAs is centralized and hence makes MBGAs difficult to be parallelized. Since there is a limit of speed on single CPU core, and as multi-core CPU, GPU and cluster computing become more available, it is worth considering the parallelism when designing MBGAs. This paper demonstrates adopting parallelism for a high-performance MBGA. Such design may be applied to other MBGAs (including estimation distribution algorithms) as well.

8 ACKNOWLEDGMENT

The authors would like to thank the support by Ministry of Science and Technology in Taiwan under Grant No. MOST 105-2221-E-002-191.

REFERENCES

[1] Peter AN Bosman and Dirk Thierens. 2012. Linkage neighbors, optimal mixing and forced improvements in genetic algorithms. *Annual conference on Genetic and evolutionary computation* (2012), 585–592.
[2] Mark Harris. 2007. Optimizing CUDA. *SC07: High Performance Computing With CUDA* (2007).
[3] Shih-Huan Hsu and Tian-Li Yu. 2015. Optimization by pairwise linkage detection, incremental linkage set, and restricted/back mixing: DSMGA-II. *Annual Conference on Genetic and Evolutionary Computation* (2015), 519–526.
[4] Pavel Krömer, Jan Platoš, Václav Snášel, and Ajith Abraham. 2013. Many-threaded Differential Evolution on the GPU. In *Massively Parallel Evolutionary Computation on GPGPUs*. Springer, 121–147.
[5] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics* 22, 1 (1951), 79–86.
[6] Luca Mussi, Fabio Daolio, and Stefano Cagnoni. 2011. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. *Information Sciences* 181, 20 (2011), 4642–4657.
[7] CUDA Nvidia. 2011. Nvidia CUDA C programming guide. *Nvidia Corporation* 120, 18 (2011), 8.
[8] Chhaya Patel. 2014. Different Optimization Strategies and Performance Evaluation of Reduction on Multicore CUDA Architecture. *International Journal of Engineering* 3, 4 (2014).
[9] Martin Pelikan and David E Goldberg. 2003. Hierarchical BOA solves Ising spin glasses and MAXSAT. *Annual conference on Genetic and evolutionary computation* (2003), 1271–1282.
[10] Petr Pospichal, Jiri Jaros, and Josef Schwarz. 2010. Parallel genetic algorithm on the CUDA architecture. *European Conference on the Applications of Evolutionary Computation* (2010), 442–451.
[11] Simon Marcus Poulding, Jan Peter Staunton, and Nathan John Burles. 2011. Full Implementation of an Estimation of Distribution Algorithm on a GPU. *Annual Conference on Genetic and Evolutionary Computation* (2011).
[12] Jae-Hyun Seo, Eun-Sol Ko, and Yong-Hyuk Kim. 2014. Performance Comparison of GPUs with a Genetic Algorithm based on CUDA. *Advanced Science and Technology Letters* 65 (2014), 36–40.
[13] Chung-Yu Shao and Tian-Li Yu. 2013. Speeding up model building for ECGA on CUDA platform. *Annual Conference on Genetic and Evolutionary Computation* (2013), 1197–1204.
[14] Daniel Leal Souza, Glauber Duarte Monteiro, Tiago Carvalho Martins, Victor Alexandrovich Dmitriev, and Otávio Noura Teixeira. 2011. PSO-GPU: accelerating particle swarm optimization in CUDA-based graphics processing units. *Annual Conference on Genetic and Evolutionary Computation* (2011), 837–838.
[15] Dirk Thierens. 2010. The linkage tree genetic algorithm. *International Conference on Parallel Problem Solving from Nature* (2010), 264–273.
[16] Dirk Thierens and Peter AN Bosman. 2011. Optimal mixing evolutionary algorithms. *Annual Conference on Genetic and Evolutionary Computation* (2011), 617–624.