Shuffled Frog Leaping Algorithm for 0/1 Knapsack Problem on the GPU

Pranav Bhandari, Rahul Chandrashekhar, and Peter Yoon
Department of Computer Science
Trinity College
Hartford, CT 06106 USA

Abstract—This paper presents an accelerated implementation of the discrete shuffled frog leaping algorithm (DSFLA) to solve 0/1 Knapsack Problem, an important class of combinatorial optimization problems. The DSFLA is known for its numerical stability, but, as the number of objects in the dataset increases, the number of possible solutions also increases exponentially, making this approach computationally impractical. To that end, the proposed parallel algorithm exploits a massively parallel architecture of modern graphics processing units (GPUs) to make the DSFLA computationally competitive in real situations. The experimental results show that it is very effective in solving 0/1 knapsack problems of various sizes, rendering an optimal solution. When implemented on a multi-GPU system, our parallel algorithm exhibits a high-level of scalability on large datasets.

1. Introduction

Given \( n \) items, each having weight \( w_i \) and value \( v_i \), we consider the problem of collecting the items in the knapsack in such a way that the total value of the knapsack is maximized while the capacity of the knapsack, \( W \), is not exceeded, that is,

\[
\text{maximize } \sum_{i=1}^{n} v_i \delta_i \quad (1)
\]

subject to \( \sum_{i=1}^{n} w_i \delta_i \leq W \quad (2) \)

where \( \delta_i \in \{0, 1\}, \quad i = 1, \ldots, n \quad (3) \)

This problem is known as 0/1 knapsack problem, an important class of combinatorial problems.

There are several applications of the 0/1 knapsack problem, including industrial applications such as cargo loading, project selection, and budget control. For example, a shipping company would want to maximize the volume of shipments that they are sending in each shipment. Choosing the right combination of items to send will help the company to ship more efficiently. Another application is in the real estate industry. The 0/1 knapsack problem can be defined in which the items are real estate properties and the knapsack capacity is the available investment. The aim is to buy properties such that the profit is maximized based on the data from the previous period.

Several methods have been proposed over the years in order to solve the 0/1 knapsack problem. When the brute force method is used, since there are \( n \) items to choose from, there will be \( 2^n \) number of possible combinations. Each combination is then checked to see which one has the highest value and is below the weight limit of the knapsack. This simple approach might be useful for a small data, but, as the data size increase, it becomes impractical because of the computational time it would require. Other approaches like dynamic programming and approximation algorithm have also been investigated [3], [4], [5], [6].

Recently, it has been shown that the shuffled frog leaping algorithm (SFLA), a memetic meta-heuristic algorithm, can be used to efficiently solve the 0/1 knapsack problem [2]. The algorithm creates a population of solutions which are known as "frogs." It uses particle swarm optimization in order to perform the local search and then improve the population to bring it closer to the optimal solution.

However, the SFLA in itself cannot be used to solve the 0/1 knapsack problem. In order to satisfy the condition (3) of the 0/1 knapsack problem, the SFLA has to be modified slightly. For the 0/1 knapsack problem, each solution is a series of bits of length \( n \) where each bit represents an item in the list and the value of the bit shows if the item is included in the knapsack. If the \( i \)th bit is 1, then the item \( i \) is included; otherwise it is excluded. This algorithm is called the discrete shuffled frog leaping algorithm (DSFLA).

Although the DSFLA can lead to an optimal solution, the algorithm might take too long when the problem set increases. As the number of objects in the list increases, the number of frogs that should be processed also increases, making the procedure computationally impractical.

In this paper, we present an efficient implementation of the DSFLA on multi-GPU systems in order to accelerate the overall computations. As the algorithm is embarrassingly parallel, there is a potential to achieve good speed-ups. Using the GPU, threads can be created which can each create a frog at the same time. The division of memeplex is also implemented in parallel. Each memeplex is assigned a thread where it undergoes local search. This is where the most
computation time is saved as the local search is the most computationally expensive part of the algorithm. After each memeplex goes through the local search, the memeplexes are combined to determine the final population at the end. The population converges towards the optimal solutions and the best frog of the final population is the best solution.

The remainder of this paper is organized as follows: Section 2 briefly describes the DSFLA, Section 3 presents a detailed parallel implementation, Section 4 presents the experimental results followed by discussion and future directions in Section 5.

2. Shuffled Frog Leaping Algorithm

The SFLA is a popular choice of solving the 0/1 knapsack problem for its fast convergence and stability. It has been used to solve many nonlinear optimization problems. A detailed description of the algorithm is given in [2].

In this algorithm based on the divide and conquer paradigm, we construct a "population" of "frogs," each of which represents a solution to the problem. These frogs are then partitioned into families of frogs, known as memeplexes. Each memeplex acts as an independent family of frogs and performs a local search. Within each memeplex, the frogs can be influenced by the state of another frog and reorganize themselves by going through a somewhat evolutionary process to determine the best solution.

After performing a number of evolutions, all the frogs are regrouped, shuffled and divided into families again. This causes an exchange of information and can have a greater influence on other memeplexes. As the SFLA is a meta heuristic algorithm, it is not guaranteed that the evolutions will always converge to a unique solution, but it will converge to an optimal solution. Local searches continue until a predetermined convergence criteria is met.

To begin a search process, an initial population of frogs, \( P \), is first created randomly. Here, the number of frogs is predetermined. The frogs are then sorted in a descending order on the basis of their fitness and divided into \( m \) memeplexes each containing \( n \) frogs. The frogs are assigned to each memeplex, so that the first frog goes to the first memeplex, the second frog goes to the second memeplex, and so on. We continue to assign each frog in cyclic fashion, that is, the \((m+1)^{st}\) frog ends up in the first memeplex.

Next, we find the best frog, \( X_b \), and worst frog, \( X_w \), in each memeplex based on their fitness. At this point, the global best frog, \( X_g \), is also determined. As a part of every evolution, the frog with the worst fitness in each memeplex undergoes an improvement step given by

\[
D_i = \text{rand} \cdot (X_b - X_w),
\]

where \( D_i \) is the change in the position of the ith frog and \( \text{rand} \) represents a random number from \((0,1]\). The new position of the frog is then determined by

\[
X_w^{(new)} = X_w + D_i \quad (5)
\]

\[
-D_{max} \leq D_i \leq D_{max} \quad (6)
\]

Here, \( D_{max} \) represents the maximum change that is allowed to a frog’s position. If this step yields a better solution, it replaces the worst frog \( X_w \). If not, just replace \( X_b \) by \( X_g \) in (4). If it still does not improve the solution, a randomly generated frog is used to replace the worst frog. The whole process continues until the convergence criteria is satisfied.

2.1 Discrete Shuffled Frog Leaping Algorithm

Unlike other knapsack problems, the 0/1 knapsack problem does not allow a fraction of an item to be included. It is either included or excluded in entirety. Thus, we use a slightly modified version of the SFLA, called the Discrete Shuffled Frog Leaping Algorithm (DSFLA). With this algorithm the worst frog in each memeplex is modified by

\[
D_i = \begin{cases} 
\min\{\text{rand} \cdot (X_b - X_w), D_{max}\} & \text{for a positive step} \\
\max\{\text{rand} \cdot (X_b - X_w), -D_{max}\} & \text{for a negative step} 
\end{cases} \quad (7)
\]

The new position of the frog is determined in the same way described in (5). The DSFLA is very similar to the SFLA in the sense that the basic approach to perform the local search remains the same. If there are no improvements possible for the worst frog, the best frog is replaced by the global best frog. If that fails as well, we use a randomly generated frog. The other important parameters such as the population \( P \), the number of memeplexes \( m \), the number of frogs per memeplex \( n \), and the number of shuffling iterations remain the same. The DSFLA is summarized in the following:

**Algorithm 1 Algorithm for DSFLA**

1: Generate random population of \( P \) solutions (or frogs)
2: for each individual \( i \in P \) do
3: Calculate fitness\((i)\)
4: Sort the population \( P \) in descending order of their fitness
5: Divide \( P \) into \( m \) memeplexes
6: for each memeplex do
7: Determine the best frog, \( X_b \), and the worst frog, \( X_w \).
8: Improve the position of \( X_w \) using (5) and (7).
9: Repeat the process for \( p \) times,
10: where \( p \) is a predefined number of iterations.
11: Combine the evolved memeplexes
12: Sort the population \( P \) according to their fitness
13: if terminated then return the best solution
2.2 Modifications for the 0/1 Knapsack Problem

It has been empirically observed that some modifications must be made to the DSFLA in order to make the local search process more suitable for the 0/1 knapsack problem. The local search procedure accounts for high convergence speed, and the purpose of using the DSFLA is to take advantage of this property. But at the same time, we also risk of a possibility of the local search oscillating between a range of optimum solutions. Thus, we must maintain the balance between the convergent and divergent properties of the approach.

2.2.1 Discretization

A critical part of the DSFLA is where we need to determine which frog is the worst frog, the one above or below the weight limit. Thus, we first check the magnitude of the frogs at two extremes. The one which is the farthest is the one that needs improvement. Therefore, the points farther away from the weight limit converge first.

Based on this, we can make two types of improvements: a positive improvement and a negative improvement. The positive improvement is done when the frog below the weight limit is selected. To improve this frog, a 0 bit is selected at random and changed into a 1. Therefore, the weight and the value of the item added will be a part of the solution. On the other hand, the negative improvement is possible when the frog above the weight limit is selected. To improve this frog, a 1 bit is selected at random and changed into a 0. Therefore, the weight and the value of the item is removed from the solution as illustrated in the following:

\[
\begin{align*}
t & = X_w + D \\
X_{w_{\text{new}}}^{(new)} & = \begin{cases} 
0 & t \leq 0 \\
\text{round}(t) & 0 < t < 1 \\
1 & t \geq 1
\end{cases}
\end{align*}
\]  

This method ensures that we get only integral values and makes the convergence faster.

2.2.2 Stopping criteria

As the DSFLA is a meta heuristic algorithm, the choice of termination condition is critical. But as the algorithm is highly sensitive to the underlying datasets, we cannot always be sure as to when to terminate. To that end, we use two approaches.

The first approach is to terminate when we reach the maximum number of iterations. The maximum number of iterations is one of the parameters that has to be passed to the algorithm. Once the local search runs for the maximum number of times the algorithm is terminated and the memeplexes are then combined to determine the solution.

Another strategy is to maintain a threshold and terminate the algorithm when a suitable level of accuracy has been obtained. In other words, the program terminates when a desired accuracy has been reached or if there is no further improvement. We can conclude that there is no further improvement when every frog in the memeplex becomes equivalent to the best frog.

3. GPU Implementation

The main purpose of our GPU implementation is to exploit the embarrassingly parallel nature of the algorithm. Recently, GPUs have been widely used for graphics processing on multiple platforms. It has been also found that they can be used for massively parallel numerical, analytical and scientific problems. By using the numerous GPU cores, the overall processing can be accelerated significantly. Our implementation was done using CUDA, a computational platform specifically designed for general-purpose computing with Nvidia GPUs.

3.1 CUDA

The availability of massive number of cores in the GPU makes CUDA a powerful programming environment to develop parallel algorithms. The CPU and the main memory are known as host while the GPUs are referred to as device. A kernel is a routine that executes on the device. The kernels designated as __global__ can only be invoked by the host while those designated as __device__ can only be called by a kernel. To successfully exploit parallelism, CUDA employs units called blocks and threads. A GPU device assigns a copy of kernel code to every block which is further broken down into threads, which work simultaneously.

Unlike CPU cores, GPU devices do not share their memory. CUDA’s unified memory model provides a framework where the CPU and the GPU can share their memory in one unit. We make an extensive use of the unified memory model in order to carry out our algorithm in parallel. While this is just from the programmer’s perspective, CUDA manages the memory and transfers of data in the background. We use a hybrid computational model in order to carry out our implementation. This model employs both CPUs and GPUs where the CPUs coordinate the communication between the GPUs and the GPUs perform the actual computations. Because the limited GPU memory for the program, we fixed the size of the GPU workspace in order to avoid running out of GPU memory irrespective of the input size. As the size of the input is independent of the GPU memory, we must account for dynamic partitioning of the data and tasks. The following sections further explain the hybrid model used in our implementation.

3.2 Creating Frogs

The creation of the frogs is the first step of the algorithm where each frog is a structure with attributes: weight,
value and a bitstring. The bitstring is of the length $n$, the number of objects in context. Each bit is either 0 or 1, indicating whether the respective object is included or excluded from the solution. Similarly, every frog represents a unique solution to the given data set. The algorithm, being embarrassingly parallel, gives us the freedom to implement this step on the device as well. Since the creation of every frog is independent from the other frogs, a GPU thread is assigned to carry out this task. Each thread first spawns a random potential solution which is a series of bits of 0’s and 1’s of length $n$. Then the thread calculates the weight and the value of the frog using the bits. An algorithm outlining the parallel creation of frogs is as follows:

Algorithm 2 Algorithm for creating frogs

1: Input : The population of frogs $P$ of size $m$, set of objects $S$ of size $n$
2: for each frog $P_i$ in the population in parallel do
3: Spawn a new GPU thread $t_j$
4: Assign $t_j$ to a frog $P_i$
5: Generate a random bitstring $b_i$ in $t_j$
6: Calculate the weight and value of $P_i$ based on $b_i$
7: return $P$

3.3 Construction of Memeplexes

After each thread completes creating and evaluating the frog, these frogs are sorted in the descending order of their value. As the sorting process involves the interaction of multiple frogs, this process cannot be carried out in parallel. Since the aim of the implementation is optimization, using conventional sorting algorithms such as bubble sort or selection sort do not provide as much of a speed up. To harness the computing power of GPUs, we use Nvidia’s Thrust library [7] to perform the sorting. Thrust is a C++ template library for CUDA which is based on the Standard Template Library. Its sorting procedure implemented on CUDA gives a speed-up of as much as 10x running on the GPU as compared to the sort() function of the C standard library. We next divide the population into memeplexes. This division is also performed in parallel by assigning one thread to one memeplex. For every iteration, a thread will select a frog from the sorted population and place it in the respective memeplex. This is carried out simultaneously by all the threads in the block making this process parallelizable as well.

Algorithm 3 Algorithm for constructing memeplexes

1: Input : The population of frogs $P$ of size $m$
2: Spawn a GPU thread to sort the population $P$ with the population as the parameter
3: Sort the population $P$ in descending order of their fitness using the Thrust library
4: for each frog $P_i$ in the population in parallel do
5: Spawn a new GPU thread $t_j$
6: Assign $t_j$ to a frog $P_i$
7: Determine the index $tid$ of $t_j$ using the formula:
   threadIdx.x + blockIdx.x * blockDim.x
8: Assign $P_i$ to the memeplex number ($tid$ mod $m$)
9: return set of memeplexes $M$

3.4 Local Search

Each memeplex is assigned to a thread where the local search takes place. To maintain the uniqueness of every thread, we associate an ID with each memeplex. This helps to relate a thread to its respective block on the device and can be used as an index for the thread. Note that this is the most computationally expensive part of the algorithm. As mentioned earlier, since the evolution of every memeplex is independent from the other in a single iteration of the local search, we employ a single GPU thread to perform a local search on every memeplex. This continues until the termination criteria is met. After the local search is completed, the frogs in the memeplex are sorted in the descending order again. This is also performed through Nvidia’s Thrust Library.

After the local search is completed, the memeplexes are combined back together and sorted in descending order.
again. After this is done, the frog with the highest value under the knapsack limit is the best solution. All the frogs in the memeplex converge to a central value with each iteration as each frog is improved in each step. A detailed algorithm is given in Algorithm 4.

Algorithm 4 Algorithm for parallel local search

1: **Input**: Set of memeplexes $M$, set of objects, Global best frog $X_g$
2: **for** each memeplex $M_i$ **in parallel do**
3: Spawn a new GPU thread $t_i$
4: Assign $t_i$ to a memeplex $M_i$
5: Determine the best frog, $X_{b}$, and the worst frog, $X_{w}$ for $M_i$
6: Improve the position of $X_{w}$ using (5) and (7).
7: Calculate number of frogs within optimum threshold
8: **if** accuracy has improved **then** repeat local search
9: Combine the evolved memeplexes
10: Sort the population $P$ according to their fitness
11: **if** terminated **then return** the best solution

3.5 Synchronization

As we are using a hybrid approach, synchronization becomes a key factor for our implementation to function smoothly on a CPU-GPU hybrid model. To exploit the massively parallel nature of GPUs, we employ thousands of threads to carry out the tasks. Threads are the smallest unit of execution path on the GPU. Even though they seem to be working simultaneously, there is a fraction of a delay between the scheduling of threads. Owing to this, the threads need to be synchronized at multiple points in the algorithm. This is achieved through the \_syncthreads() call. The threads belonging to a specific block are all scheduled under one streaming microprocessor. Hence the synchronization is computationally inexpensive on the GPU.

4. Experimental Results

We tested our implementation of the DSFLA on a system which comprises a dual Intel Xeon E5-2620 CPUs with 64 GB main memory, and four NVIDIA Tesla K20c GPU, each with 5 GB global memory.

4.1 Datasets

We tested our implementation using two types of datasets: random and real estate data.

4.1.1 Random Data

This dataset contains randomly generated data without correlation, with positive correlation and with negative correlation as seen in Figure 1. To ensure that a solution exists to the input data, we use a data generator which prepares a data set on the basis of several parameters. The parameters that we use are the number of objects and the capacity of the knapsack. To be able to test our implementation successfully, it is essential to know that a solution exists to the input data hence the first step is to generate solutions to the given input set. To achieve this, the generator determines the average weight of the objects and distributes the weights across the objects so that the total weight is the weight limit. This guarantees the existence of a few solutions in the generated frogs and we can be sure that the algorithm will eventually converge to some solution. The other possible solutions are generated keeping the value to weight ratio of the known solutions much higher than the value to weight ratio of the newly generated ones. This accounts for easier verification of the solution set and fast convergence.

4.1.2 Real Estate Data

We also experimented our GPU-based DSFLA with real-world data in the form of real estate information. In this application, we consider a real estate property to be an item. The weight of the item here is the price of the property and the value is the net increase in the price of the property from last year. Thus, here what the DSFLA will do is try to maximize the price increase. For the price increase we use percent increase or exact monetary values. Instead of data of particular properties, average data can be used when looking to choose between different areas to invest in.

There are various ways this approach can be used to get information in the real estate area. Instead of the price increase, we can use other information such as the value of the frog to extract additional facts. If the value is set to decrease in price, it will show the properties that lost the most value.

Figure 1 shows the the real estate data from the state of Connecticut we used for the experiment [8]. The datasets provide information such as the number of properties sold in the particular county, the median price at which it was sold and the change in the price of the property from the last quarter. We collected the increase in price and average cost of a property of various counties and ran the DSFLA on this data. It must be noted that since this is an application of the algorithm, the solution set is purely computational and not very realistic. In order to achieve pragmatic results, pre-processing of the datasets was required. We also observed that purchasing more properties in the same county can be more productive and hence alter the number of frogs for a property accordingly.

For the pre-processing we first calculated the expected number of houses that will be sold in a county in the next quarter depending on the data of the previous quarter. After the prediction, we assigned each county with market influence constant, $k$, which estimates the percentage of market that you can influence. It basically is an estimation of how many houses one will realistically be able to buy considering all of them are on sale. It is very unlikely that a
### Table 1: Real Estate Data from Berkshire Hathaway with Market Influence Constant $k = 0.5$

<table>
<thead>
<tr>
<th>County</th>
<th>Median price (in USD)</th>
<th>Change in price (in %)</th>
<th>Profit (in USD)</th>
<th>Sales</th>
<th>Change in sales (in %)</th>
<th>Expected sales</th>
<th># frogs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin</td>
<td>265,000</td>
<td>7.1</td>
<td>18,815</td>
<td>51</td>
<td>6.3</td>
<td>54</td>
<td>27</td>
</tr>
<tr>
<td>Bloomfield</td>
<td>156,000</td>
<td>-16.6</td>
<td>-25,896</td>
<td>53</td>
<td>12.8</td>
<td>59</td>
<td>29</td>
</tr>
<tr>
<td>Farmington</td>
<td>397,000</td>
<td>10.3</td>
<td>40891</td>
<td>42</td>
<td>9.95</td>
<td>46</td>
<td>31</td>
</tr>
<tr>
<td>Southington</td>
<td>274,500</td>
<td>-0.1</td>
<td>-274.5</td>
<td>127</td>
<td>8.5</td>
<td>137</td>
<td>68</td>
</tr>
<tr>
<td>New Britain</td>
<td>140,000</td>
<td>12</td>
<td>16800</td>
<td>73</td>
<td>-21.5</td>
<td>57</td>
<td>28</td>
</tr>
<tr>
<td>Manchester</td>
<td>162,500</td>
<td>-2.7</td>
<td>-4387.5</td>
<td>143</td>
<td>-10.1</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>West Hartford</td>
<td>322,000</td>
<td>5.6</td>
<td>18032</td>
<td>137</td>
<td>-8.23</td>
<td>125</td>
<td>57</td>
</tr>
<tr>
<td>Windsor</td>
<td>204,500</td>
<td>-9.1</td>
<td>-18609.5</td>
<td>84</td>
<td>18.3</td>
<td>99</td>
<td>49</td>
</tr>
<tr>
<td>Enfield</td>
<td>168,000</td>
<td>8.4</td>
<td>14112</td>
<td>129</td>
<td>31.6</td>
<td>169</td>
<td>84</td>
</tr>
<tr>
<td>Hartland</td>
<td>270,000</td>
<td>0.8</td>
<td>2160</td>
<td>55</td>
<td>-20.3</td>
<td>43</td>
<td>21</td>
</tr>
<tr>
<td>Marlborough</td>
<td>275,000</td>
<td>13.6</td>
<td>37400</td>
<td>19</td>
<td>-9.5</td>
<td>17</td>
<td>8</td>
</tr>
<tr>
<td>New Britain</td>
<td>140,000</td>
<td>12</td>
<td>16800</td>
<td>73</td>
<td>-21.5</td>
<td>57</td>
<td>28</td>
</tr>
<tr>
<td>Plainville</td>
<td>175,000</td>
<td>-15.5</td>
<td>-27125</td>
<td>45</td>
<td>-15.1</td>
<td>38</td>
<td>19</td>
</tr>
<tr>
<td>Rocky Hill</td>
<td>304,300</td>
<td>10.6</td>
<td>32255.8</td>
<td>32</td>
<td>-8.6</td>
<td>29</td>
<td>14</td>
</tr>
<tr>
<td>Suffield</td>
<td>330,000</td>
<td>15.2</td>
<td>50160</td>
<td>36</td>
<td>37.5</td>
<td>50</td>
<td>26</td>
</tr>
</tbody>
</table>

A person or company will purchase all the properties within a certain county. So depending on the estimation the value of $k$ can be set, $k$ times the number of estimated houses sold gives the number of frogs for that particular county. For this purpose we have set $k$ to 0.5.

The weight, in our case, is the cost of the property and the maximum weight is the amount of money that is available for investment. The profit made by the property in the last quarter is considered as the value. We assume that the rate of change of the price will be the same and consider the potential profit that could be made.

### 4.2 Discussions and Conclusion

The results obtained indicated good speedups. For lower dimensional problems, it is expected that the serial algorithm will run faster than the parallel one, but as we increase the dimension of the problem the speedup becomes more and more evident. The multi-CPU vs multi-GPU comparison also shows a favorable speedup in the case of the latter.

Figure 2 shows the Accuracy Result and the Timing Result of the algorithm on randomly generated data for different dimensions of the memeplexes on a single GPU. It shows how the dimension of the memeplex also slightly affects the performance of the algorithm. The Accuracy Result shows how having fewer memeplexes with more frogs is favorable for the accuracy as compared to more memeplexes with fewer frogs. The Timing Result also favors the former when it comes to faster running time.

For most cases, the data must be exchanged between the CPU and the GPU and for smaller dimensional problems, this is what slightly sets back the GPU based implementation as compared to the serial version. For smaller data sets, the time taken for this data transfer is very small and almost negligible. And as the problem size grows, the time taken for exchange of data gets magnified hence giving us an exponential increase in the time taken. The number of items that we have in the problem affects the time taken by the algorithm. When we increase the number of items, there is an exponential increase in the amount of time taken.

This can be seen clearly in Table 2 where the increase in the step size of the objects exhibits an exponential increase in the running time of the implementation. The data used is positively correlated. As seen in the case of the multi-CPU system, the running time is faster for smaller cases but the multi-GPU system dominates when it comes to large test cases. The speed-up is at least 7x for all sizes of the problem. The accuracy achieved was 82.7% on average.

Through multiple test runs across various implementations, there are some patterns that were observed which affect the performance of the algorithm. It is observed that increasing the number of frogs in each memeplex increases...
the number of steps taken to converge to the result in most cases. This is expected because adding more frogs also accounts for more frogs to be improved through the local search. But since the algorithm is a meta-heuristic algorithm and the correct solution is not guaranteed, increasing the number of frogs improves the accuracy of the solution set generated. As the number of frogs increases, there are more frogs that converge near the correct solution giving us a more accurate result.

The tests performed on correlated data have shown better accuracy than that in a general case with randomly generated data. The correlated data follows a trend and since the frogs are sorted before dividing them up into memeplexes, they are evenly spread out across the memeplexes. Owing to this, the number of changes required in the bitstring of the frog to make it converge are less. And since the number of bit flips is lower, the number of steps required also decreases. This factor accounts for quicker convergence with fewer steps.

The number of iterations also plays an important role in the results while considering larger test cases. For smaller cases, the program terminates on converging to the most optimal solution. But as we increase the number of frogs, it becomes increasingly difficult to keep all the frogs within the optimal threshold. It was observed in a few cases that some frogs, after reaching the optimal solution tend to diverge from it over the course of multiple iterations. No specific pattern has been observed with respect to this anomaly due to which the accuracy has suffered.

The real estate data required a significant amount of polishing before it could have been used in order to obtain practical results. The results that we obtained were mathematically correct but were not feasible at all which made us do the pre-processing in the data after which the results have been satisfactory.

For our application of the DSFLA on real estate data, we used $2$ million as the maximum "weight", W. As the data set was relatively small, the results could be verified for plausibility. After performing multiple test runs on the real estate data, the results proved to be consistent with what would be a good investment in terms of maximizing profit. On an average, the set of estates that would be ideal for investing within the allotted budget were: Suffield, Farmington, Marlborough, Rocky Hill, Berlin and West Hartford. The results from this application are consistent with those achieved for the datasets with randomly generated data as presented in Table 2. The average speedup was nearly 8x.

Through the analysis performed and results collected, it has become clear that there exists a trade-off in the implementation when it comes to accuracy and the size of the problem. As the problem size increases so does the number of frogs. An increase in the number of frogs exhibits a better accuracy up until a certain point but for data sets of size $> 3,000$ and above, the accuracy suffers as the problem size increases. This leads us to a trade-off between the accuracy and the size of the problem.

It must be noted that these observations are based on multiple test runs and the average of the cases has been considered. The DSFLA, being a meta-heuristic algorithm does not guarantee convergence to a unique solution and so the solution set that is generated might be different for the same input set run multiple times. However, the overall performance of our GPU implementation of the DSFLA has been proven effective in solving the 0/1 knapsack problem.

**Acknowledgments**

The authors would like to thank CUDA Teaching Center Program, Nvidia Research, and Interdisciplinary Science Program, Trinity College, for supporting this research.

**References**