A GPU accelerated parallel heuristic for the 2D knapsack problem with rectangular pieces

MOHAMMAD HARUN RASHID
Pace University
New York, USA
Email: mr29963n@pace.edu

Abstract: The 2D Knapsack Problem is a NP hard problem in combinatorial optimization which can be described easily, but it is very difficult to solve. If we take a rectangular container as well as a set of rectangular pieces into consideration, the two dimensional knapsack problem (2D-KP) consists of packing a subset of the rectangular pieces in the rectangular container in such a way that the sum of the total values of the packed rectangular pieces is maximized. If we consider the value of a rectangular piece by the area, the goal here is to maximizing the covered area of the rectangular container. It is not only very time consuming for Central Processing Units (CPU) but also very difficult to obtain an optimized solution when solving large problem instances. So, parallelism can be a good technique for reducing the time complexity, as well as improving the solution quality. Nowadays Graphics Processing Units (GPUs) have evolved supporting general purpose computing. In this paper, we propose GPU accelerated parallel heuristics for 2D Knapsack Problem and our experimental evaluation shows that this optimization algorithm efficiently accelerated on GPUs can provide with higher quality solutions within a reasonable time for 2D-KP.

Keywords: GPU, combinatorial optimization, 2D Knapsack Problem, parallel heuristics, algorithms, greedy algorithm, local search

I. Introduction

The 2D knapsack problem (2D-KP) with rectangular pieces is a popular and well-studied combinatorial optimization problem. It has many applications in real world such as metal, wood, paper and glass industries. This paper presents a optimized solution for the 2D-KP problem with a larger rectangle container and a set of small rectangular pieces (R1,R2,....Rn); Ri = (wi, hi) . The job is to efficiently arrange the subset of the small rectangular pieces within the large rectangular container such that the total value of the packed pieces is maximized. If we consider the value of a small piece as its area, the objective here is to maximize the covered area of the rectangular container. We can also call the arrangement of pieces as a packing plan by assuming that each piece is placed completely into the container in parallel to the container edges and the placed pieces won't overlap each other.

This problem is known as a NP hard problem. It has a number of real world applications, for instance: designing integrated circuits with efficient organization of components on chips, packing goods efficiently on pallets in horizontal layers, effective arrangement for saving the space on newspapers pages etc.

Combinatorial optimization problems are often NP-hard. It is often time consuming and very complex for Central Processing Unit (CPU) to solve combinatorial optimization problems, especially large scale problems. Metaheuristic methods can be used to find satisfactory resolution(optimal) within a reasonable time. Local search metaheuristics (LSMs) are single solution-based approaches, as well as one of the most widely researched metaheuristics with various types such as iterated local search, hill climbing, simulated annealing (SA), tabu search, and genetic algorithm etc. Parallelism is a good technique for improving the solution quality as well as reducing the time complexity.

Recently, Graphics Processing Units (GPUs) have been developed gradually from fixed function rendering devices to parallel/programmable processors. GPUs motivated by high definition 3D graphics from real time market demand have become many core processors, multithreaded, highly parallel with high bandwidth memory and tremendous computational power. So, more transistors are devoted to design GPU architecture in order to do more data processing than data caching and flow control. With the fast development of general purpose Graphics Processing Unit (GPGPU) techniques, many companies have promoted GPU programming frameworks, such as CUDA (Compute Unified Device Architecture), OpenCL (Open Computing Language) and direct Compute. GPU based metaheuristics have become much more computational efficient compared to CPU based metaheuristics. Furthermore, making local search algorithms optimized on GPU is an important problem for the maximum efficiency.

978-1-5386-7693-6/18/$31.00@2018 IEEE

In this paper, we present a GPU accelerated optimized parallel heuristics and we have made some experiments to test the proposed algorithm with 2D-KP. Our GPU based parallel approach that integrates genetic algorithm, greedy algorithm and local search method can be highly considered as an efficient parallel heuristic approach to find higher quality optimal solution for 2D-KP.

II. Background

1) Genetic Algorithm: Genetic Algorithm [7] is based on natural evolution that includes the idea of surviving of the most fitted element in a searching algorithm. In nature the most appropriate individuals have more possibilities of surviving and it will bring more adapted successors, those will be healthier than their parents. The same idea is applied by creating a new generation of solutions that are nearer to the demanded solution. A genetic algorithm consists of these steps: 1) encoding, 2) evaluation, 3) selection, 4) crossover, 5) mutation, 6) decoding. The basic genetic algorithm is given below:

```
Basic Genetic Algorithm:

Step-1: Initialize population with random solutions.
Step-2: Evaluate each candidate.
Step-3: Repeat (iterations).
            a) Select parents.
            b) Recombine pairs of parents.
            c) Mutate the resulting children.
            d) Evaluate children.
            e) Select individuals for the next generation.
    Until Terminating condition is satisfied.
```

2) Local Search Heuristic: Local search [8] is single solution-based approaches, as well as one of the most widely researched metaheuristics with various types such as iterated local search, hill climbing, simulated annealing (SA), tabu search, and genetic algorithm etc. A common feature that local search metaheuristics can share is, a candidate solution is iteratively selected from its neighborhood.

   In science and industry, local search methods are a class of heuristic methods to solve very complex optimization problems. Even though these iterative methods can reduce the computational time significantly, the iterative process can still be costly when dealing with very large problem instances. Although local search algorithms can also reduce the computational complexity for the search process, still it is very time consuming for CPU in case of objective function calculations, especially when the search space size is too large. Therefore, instead of traditional CPUs the GPUs can be used to find efficient alternative solutions for calculations.

3) Greedy Algorithm: A greedy algorithm is a problem solving heuristic that makes the optimal choice locally at each stage in order to find a global optimum. Greedy algorithms having the properties of mastoids can solve combinatorial optimization problems.

   We can make a greedy choice that looks best at that moment and later on, it can solve the sub problems. The greedy choice for a greedy algorithm doesn't depend on future choices, nor the solutions to the sub problem, but it may depend on choices made so far. It reduces each problem into a smaller one by iteratively making one greedy choice after another.

III. GPU ARCHITECTURAL CONSTRAINTS

Just a while ago, the conventional single core or multicore CPU processor was the only viable choice for parallel programming. Usually many of them were arranged either tightly as multiprocessors by sharing a single memory space, or loosely as multicomputer with the communication among them done indirectly due to the isolated memory spaces. CPU has a large cache as well as an important control unit, but it doesn't have many arithmetic logic units. CPU can manage different tasks in parallel which requires a lot of data, but data are stored in a cache for accelerating its accesses. Nowadays most of the personal computers have GPUs which offer a multithreaded, highly parallel and many core environments, and can potentially reduce the computational time. The performance of the modern GPU architecture is outstanding in regards to power consumption, cost and occupied space.

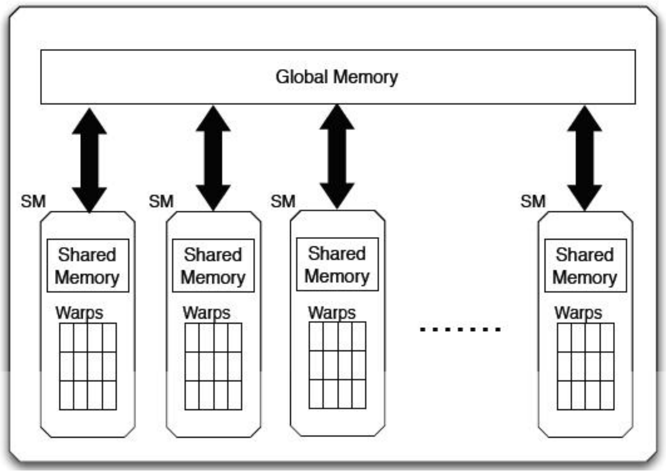

Fig. 1. GPU Architecture

A GPU consists of a number of Stream Multiprocessors (SM). Each stream multiprocessor includes multiple process-ing units which has the ability of concurrently executing thousands of operations. Threads are grouped into warps inside a SM. A warp can execute 32 threads in a SIMD (Single Instruction Multiple Data) manner, which means all threads in a warp can execute the same operation on multiple data points. There are at least two kinds of memory: global memory and shared memory. Global memory allows to store a large amount of data (e.g., 8GB), whereas shared memory can usually store only few Kilobytes per SM.

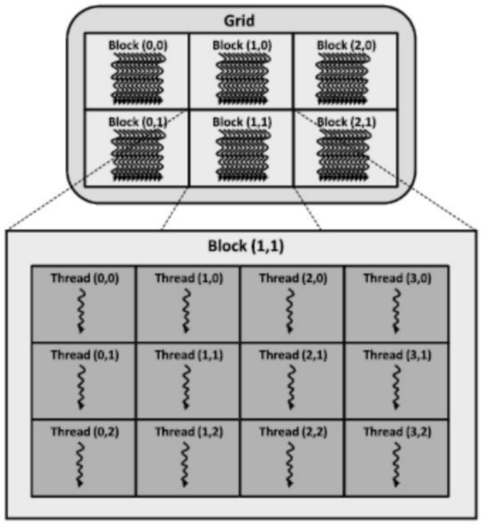

Fig. 2. GPU threads model

A GPU thread can be considered as a data element to be processed. GPU threads are very lightweight in comparison with CPU threads. So, it is not a costly operation when two threads change the context among each other. GPU threads are organized in blocks.

Blocks are organized in a one or two dimensional grid of threaded blocks and equally threaded multiple blocks execute a kernel. Threads are grouped in a similar way inside a block and each thread is assigned a unique id. The advantage for group-ing is that the number of blocks processed simultaneously by the GPU is closely linked to hardware resources. The threads within the same block are assigned to a single multiprocessor as a group. So, different thread blocks are assigned to different multiprocessors. Therefore, a major issue is to control the threads parallelism for meeting the memory constraints. From a hardware point of view, graphic cards consist of stream multiprocessors and each with registers, processing units, and on chip memory. As multiprocessors are mainly organized based on the SPMD (Single Program Multiple Data) model, the threads can share the same code as well as access to different memory areas.

GPU is used as a device coprocessor and CPU is used as a host. Each GPU has its own processing elements and

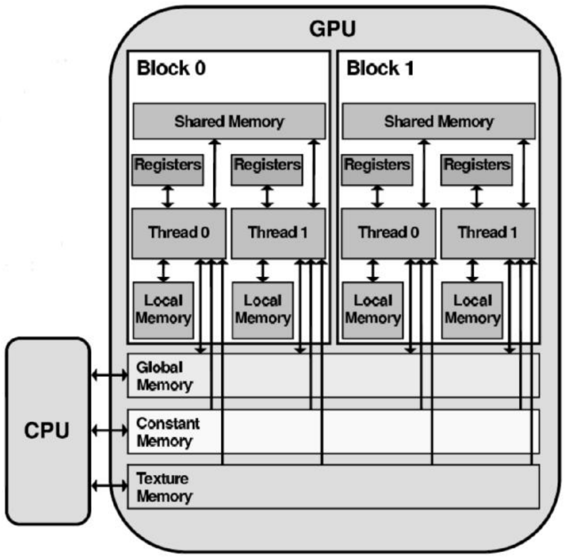

Fig. 3. CPU - GPU communications

memory which are separate from the host computer. Data is transferred between the host memory and the GPU memory during the execution of programs. Each device processor on GPU supports SPMD model, which means multiple autonomous processors can simultaneously execute same program on different data. To achieve this, we can define the kernel concept. The kernel is basically a method or function which is executed by several processors simultaneously on the specified device in parallel and callable from the host as well. The Communications between CPU host and the device coprocessors are accomplished via the global memory.

IV. RELATED WORK

Andreas Bortfeldt and Tobias Winter [1] proposed a genetic algorithm, that can address both guillotine case and non-guillotine case of the 2D-KP. In addition, an orientation constraint can be considered optionally and the given set of pieces can be constrained or unconstrained. They used genetic algorithm and made extensive test by using well known benchmark instances. According to them, their experiment gave competitive results compared to recently published work.

Defu Zhang, Stephen C. H. Leung, Yain Whar Si and Furong Ye and [2] proposed a variable neighbourhood based hybrid algorithm. Their proposed hybrid heuristic integartes a heuristic algorithm with a variable neighbourhood search. Based on the block patterns concept, they constructed different neighborhoods. Their experimental validation with a set of problem instances gave better results compared to recently published work.

Also, previously I (Mohammad Rashid) [3] proposed a hybrid algorithm that combines genetic algorithm, greedy algorithm and local search algorithm. In our previous approach, we kept the evolutionary technique for genetic algorithms by using operators such as crossover, mutation etc., but a local search algorithm was used to enhance the crossover operator. Our greedy appraoch generated the start generation and carried out the post optimization for the previous best solution.

V. Design and Methods

With our new approach, our previously proposed heuristics is parallelized on GPU by using so many number of GPU threads and making an efficient communication between CPU and GPU in order to find a better solution than all other previous works.

```
Proposed Parallel Heuristic:

Step-1: Allocate 200 GPU threads.
Step-2: For each GPU threads.
        a) Run the Main 2D-KP Algorithm in parallel to find the best packing plan.
        b) Send each thread obtained packing plan (sum of the values of the packed pieces) to CPU.

Step-3: CPU compares all the packing plans sent by all 200 GPU threads and determine the best packing plan that maximize the covered area of the container.
```

Below is our previously proposed algorithms to find optimal solution for 2D-KP.

```
Our Main 2D-KP Algorithm:

Step-1: Generate a series of random initial solutions (Packing plan with a sequence of rectangle pieces).
Step-2: Calculate fitness (height and width) of every rectangle pieces of initial solutions.
Step-3: Repeat (iterations).
        a) Pick a random solution and apply local search heuristic to select parents from initial population for cross over and mutation.
        b) Apply a greedy heuristic GreedyPacking() to generate greedy solution from parents.
        c) Calculate fitness of newly generated solution and add newly generated solution in initial population.
        d) Sort combined population (initial population + new greedy solution) by height/width and select best solution(Packing plan) for next population.
        e) Set initial population = next population
    Until Terminating condition is satisfied (for given number of generations ).
Step-4: Call GreedyPostOptimization() algorithm to carry out post-optimization of the previously best solution
```

In our greedy approach, we considered the following conditions and fitness while building a greedy heuristic.

When h1>=h2,

if w = wi and h1 = li, then fitness = 6 if w = wi and h2 = li, then fitness = 5 if w = wi and h1 <li, then fitness = 4 if w >wi and h1 = li, then fitness = 3 if w = wi and h1 >li, then fitness = 2 if w >wi and h2 = li, then fitness = 1 if w >w, then fitness = 0

When h1<h2,

if w = wi and h2 = li, then fitness = 6 if w = wi and h1 = li, then fitness = 5 if w = wi and h2 <li, then fitness = 4 if w >wi and h2 = li, then fitness = 3 if w = wi and h2 >li, then fitness = 2 if w >wi and h1 = li, then fitness = 1 if w >w, then fitness = 0

In case of fitness = 3 or 1 or 0, if we pack the piece i, then the remaining width (w - wi) will be smaller than the minimum width of all unpacked pieces. So, there will definitely have some waste. In that case, we propose a least waste strategy. That is, one unpacked piece j can be selected such that the width of j can be largest amongst all the unpacked pieces. Also, wj will be less than or equal to w but the perimeter of j will not be less than that of piece i. Therefore, this strategy might reduce the waste by making up the limitations of the scoring strategy.

Our proposed algorithm is as follows:

```
Our GreedyPacking() Algorithm:

Step-1: for all unpacked pieces do.
Step-2: select the piece i with the highest fitness according to the above mentioned conditions and fitness
Step-3: if the fitness of the piece i is 3 or 1 or 0 then
            select the piece j according to the lease waste strategy.
            if the piece j can be found then pack the piece j instead of packing i
            else pack the piece i
        else pack the piece i
Step-4: Return greedy solution.
```

In addition, we proposed the following GreedyPostOptimization() algorithm to carry out post-optimization of the previously best solution. This algorithm stores a list of free rectangles that represents the free area left in the bin at some packing step.

Our GreedyPostOptimization() Algorithm:

```
foreach Free Rectangle
            Compute the free area left in the bin
            Select an unpacked rectangle, which has smaller height/width than the free area
            Pack the piece in that free area
    end
```

VI. Experimental results

We set up an environment with NVIDIA GPU, CUDA technology, C++ programming language and Visual Studio to test our proposed parallel heuristic for the 2DKP. We found the following results returned by different GPU threads; which finally provided with maximized covered area:

GPU Threads	Covered Area	Maximized Covered Area
1	19	
2	21	
10	18	
30	17	23
80	20	
140	23	
200	22	

We can see that different GPU threads return different covered areas. As we consider only the maximum (23), the sum of the covered area can be maximized by approximately 35.29%. So, we can confirm that parallelizing our proposed heuristics with GPUs can provide us with higher quality solutions for 2D-KP.

VII. Conclusions

In this work, we proposed a parallel heuristic that combines a genetic algorithm, a greedy algorithm and local search method, and we also parallelized this heuristic with GPU. Our experimental result shows that the covered area is maximized by 35.29%. Therefore, we can confirm that because of the parallelization with GPU we achieved a good improvement compared to previous work.

References

[1] Andreas Bortfeldt and Tobias Winter, A Genetic Algorithm for the Two-Dimensional Knapsack Problem with Rectangular Pieces, http://www.fernuni-hagen.de. German, 2008.

[2] Defu Zhang and Furong Ye and Yain Whar Si and Stephen C. H. Leung, A hybrid algorithm based on variable neighbourhood for the strip packing problem, Journal of Combinatorial Optimization. China, 2016.

[3] Mohammad Harun Rashid, A Greedy Local Search Metaheuristic with Genetic Algorithm for the 2D Knapsack Problem with Rectangular Pieces, Pace University. USA, 2017.

[4] Jakob Puchinger and Guunther Raidl, An Evolutionary Algorithm for Column Generation in Integer Programming: An Effective Approach for 2D Bin Packing, https://hal.archives-ouvertes.fr/hal-01299556. UK, 2016.

[5] Jukka Jylnki, A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing, 2010.

[6] E. HOPPER and B. C. H. TURTON, A Review of the Application of Meta-Heuristic Algorithms to 2D Strip Packing Problems, Artificial Intelligence Review. UK, 2001.

[7] Andrea Lodi, Algorithms for Two-Dimensional Bin Packing and Assignment Problems, UNIVERSITA DEGLI STUDI DI BOLOGNA. 1999.