INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

Efficient Parallelization and Performance Analysis of Meta-heuristics on Many-core Platforms

by

Manoj Kumar

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Department of Computer Science and Engineering

Under the supervision of

Prof. Pinaki Mitra

June 2023

Declaration of Authorship

I, Manoj Kumar, hereby confirm that:

- The work contained in this thesis is original and has been done by myself under the general supervision of my supervisor.
- This work has not been submitted to any other Institute for any degree or diploma.
- Whenever I have used materials (data, theoretical analysis, results) from other sources, I have given due credit to the authors/researchers by citing them in the text of the thesis and giving their details in the reference.
- Whenever I have quoted from the work of others, the source is always given.

Manoj Kumar
Research Scholar,
Department of CSE,
Indian Institute of Technology Guwahati,
Guwahati, Assam, INDIA 781039,
manoj.kumar@iitg.ac.in, manojngp13@gmail.com

Date: June 8, 2023
Place: IIT Guwahati

Certificate

This is to certify that the thesis entitled **"Efficient Parallelization and Performance Analysis of Meta-heuristics on Many-core Platforms"** being submitted by **Mr. Manoj Kumar** to the department of *Computer science and Engineering, Indian Institute of Technology Guwahati*, is a record of bonafide research work under my supervision and is worthy of consideration for the award of the degree of Doctor of Philosophy of the Institute.

Prof. Pinaki Mitra
Department of CSE,
Indian Institute of Technology Guwahati,
Guwahati, Assam, INDIA 781039,
pinaki@iitg.ac.in

Date: June 8, 2023
Place: IIT Guwahati

Dedicated to

my guru maharaji PREM RAWAT

and

my loving parents & wife.

Acknowledgements

It is an immense pleasure for me to thank all the peoples who have supported me during my Ph.D. stay at IIT Guwahati. First and foremost, I thank my supervisor Prof. Pinaki Mitra for his guidance, encouragement, and extensive help over the last four years. He has given me the freedom to pursue research ideas and develop my research skills. I also thank Prof. Aryabartta Sahu for his in-depth guidance for research works and encouraging me during my Ph.D. work. I profusely thank him for correcting my silly mistakes again and again and keep me engaged on Ph.D. work.

I am thankful to my Doctoral Committee Members: Prof. G. Sajith, Prof. Sushanta Karmakar, and Dr. Rashmi Dutta Baruah for their productive and constructive suggestions for my thesis work. I firmly believe that their opinions and comments help me to shape up my final thesis. Additionally, my sincere thanks to Prof. Jatindra Kumar Deka, the Head of the Department of Computer Science and Engineering, and other department faculty members for their constant support and helps.

I specially thanks to Dr. Rakesh Tripathi for encouraging and saving me from many up and downs during entire Ph.D works. I would also like to thank Dr. Bala for technical discussions and review for the my publications. I also thank to one of my seniors Pradeep for motivating me during the research and Ph.D works.

My friends and well-wishers have greatly helped me and without their help my work would not have been possible. I feel lucky to have a company of Dr. Sukarn, Dr. Saptarshi, Dr. Hema, Dr. Surajit, Deepak(all PhDs). An uncountable number of random discussions with them makes my life not dull and joyful. My thanks are also due for Aditya, Pankaj, Naween, Birendra and many more (all College and School mates) for their tremendous support to me which hardly few people will get.

I sincerely thanks to Mr. Raktajit Pathak, Mr. Nanu Alan Kachari, Mr. Bhriguraj Borah, Mr. Monojit Bhattacharjee, Mrs. Gauri Deori and all other department staff members for helping me different ways and at different times during my stay at IIT Guwahati. I would also like to thanks the student affairs section for providing on-campus hostel facility. Last but not least, I am conveying my appreciation to security guards, janitors, hostel mess, and canteen staffs for making my life smooth in IITG campus.

Above all, I am incredibly fortunate to have moral support and encouragement from my parents and wife Supriya Bharti. I must say without their moral support and continuous motivations, none of this would have been possible. Thank You for everything to me. I feel sorry to my son *Chunmun* who didnt got to much love from father because i am busy in my ph.d work.

Last but not least, I would like to thank those people who discourage, demoralize, and menace me at different stages of life. All these motivate me to take something as a challenge and work hard for it.

Abstract

Meta-heuristics are an efficient method for solving complex problems in science, engineering, and industry. They explore the solution space efficiently to generate a good solution in a reasonable time through a neighborhood or population-based local search. Even if the meta-heuristics do it efficiently, for large instances (practical problems of science, engineering, or industry), generation of neighborhood and evaluation of solution of single-solution based meta-heuristics or population-based meta-heuristics takes a tremendous amount of time.

We used this highly parallel meta-heuristics solver to run on modern days massively parallel Graphics Processing Unit (GPU) to reduce the execution further. In this work, we scheduled and mapped the application to take advantage of the GPU's architectural configuration to run efficiently by taking advantage of the management of local memory, shared memory, and global memory of the accelerator. Many GPU programmers use most coding styles or mapping strategies that are adhoc based. We automated this mapping or coding approach so that the accelerators are adequately used and improve the program's performance.

In the later phase, we analyzed the performance of meta-heuristics on GPU by using static and dynamic auto-tuning of application models on target architecture for better performance. In this study, we evaluated the performance of various meta-heuristics for the quadratic assignment problem (QAP), traveling salesman problem (TSP), and permuted perceptron problem (PPP) using a massively parallel machine such as a GPU. These meta-heuristics included iterated local search (ILS), simulated annealing (SA), genetic algorithm (GA), tabu search (TS), particle swarm optimization (PSO), and crow search algorithm (CSA). We achieved the highest speedup, 127.56 on GPU using PSO for QAP, speedup 50.53 using CSA for TSP, and speedup 165.62 using SA for PPP.

The thesis has thus demonstrated the practical management techniques for achieving the highest speedup and optimal solution on many-core architecture for real-life optimization problems using meta-heuristics.

Contents

List of Figures

List of Tables

Abbreviations

CPU	Central Processing Unit
GPU	Graphics Processing Unit
ILS	Iterated Local Search
SA	Simulated Annealing
GA	Genetic Algorithm
TS	Tabu Search
PSO	Particle Swarm Optimization
CSA	Crow Search Algorithm
QAP	Quadratic Assignment Problem
TSP	Traveling Salesman Problem
ACO	Ant Colony Optimization
PP	Perceptron Problem
PPP	Permuted Perceptron Problem
Pthread	POSIX Thread
PC	Personal Computer
SIMT	Single Instruction Multiple Thread
SIMD	Single Instruction Multiple Data
OS	Operating System
MIMD	Multiple Instructions and Multiple Data
SMP	Symmetric Multiprocessor
I/O	Input Output
ALU	Arithmetic Logic Unit
SMT	Simultaneous Multi-threading
MIC	Many Integrated Core
TD	Tag Directory
VPU	Vector Processing Unit
KB	Kilo Bytes
PCIe	Peripheral Component Interconnect Express
DRAM	Dynamic Random Access Memory

CUDA	Compute Unified Device Architecture
SFU	Special Function Unit
LD/ST	Load or Store
SP	Streaming Processor
SM	Streaming Multiprocessor
GDDR5	Graphics Double Data Rate 5
HPC	High Performance Computing
D	Deviation
ET	Execution Time
S	Speedup
OpenMP	Open Multi-Processing
MPI	Message Passing Interface
FORTRAN	FORmula TRANslation
QAPLIB	Quadratic Assignment Problem Library
TSPLIB	Traveling Salesman Problem Library
EA	Evolutionary Algorithm
GM	Global Memory
LM	Local Memory
CM	Constant Memory
SM	Shared Memory
TM	Texture Memory

Chapter 1

Introduction

Many real-world problems are complex and exciting to solve or to get near-optimal solutions is becoming a sensitive area for researchers. These optimization problems can be solved by the heuristics method for small-size instances, but for large-size instances, it may take a considerable amount of time. There are many meta-heuristics available [1] through which we can get near-optimal solutions to optimization problems using massively parallel devices such as Graphics Processing Unit.

We attempted Quadratic Assignment Problem (QAP) [2], Permuted Perceptron Problem (PPP) [3], and Traveling Salesman Problem (TSP) [4] on targeted multi or many-core architecture like GPU and Xeon-Phi co-processor using s meta-heuristics (single solution based meta-heuristic), and p meta-heuristics (population-based meta-heuristics). Many exact and heuristic methods can solve the QAP problem [5]; however, heuristics take a considerable time for instances greater than 20. Compared to problem-specific heuristic methods, meta-heuristics are suitable for all problems. Thus, they have become one of the alternative methods for solving the QAP, with many nature-inspired meta-heuristics being proposed for this purpose. In this study, we have chosen a subset of single solution-based meta-heuristics (i.e., the iterated local search (ILS), simulated annealing (SA), and tabu search (TS)) and a subset of population-based meta-heuristics (i.e., the

genetic algorithm (GA), particle swarm optimization (PSO), and crow search algorithm (CSA)) to find the optimal solution for QAP. In addition, we have considered 21 test instances from the QAP library (QAPLIB) [6] as a benchmark for QAP, generated task graphs for the meta-heuristics using the *Contech* tool [7] for parallel programs, and analyzed their performance.

Next, we attempted PPP. This problem is an NP-complete problem based on a cryptographic identification scheme, which is well suited for resource-constrained devices such as smart cards. PPP is derived from the perceptron problem (PP), which is also an NP-complete problem. The perceptron problem is motivated by a well-known perceptron in Neural Computing. We used SA meta-heuristics to get the number of possible solutions, record the time on the serial device (CPU) and the parallel device (GPU), and also computed the speedup on GPU as compared to the CPU.

We also attempted TSP. This problem belongs to NP-complete class problems, and it has many applications. Although, many recent meta-heuristics algorithms [8, 9] are used to optimize the TSP. We used six meta-heuristics- ILS, SA, GA, PSO, CSA, and TS to get the optimum tour in a reasonable time. Here we used TSPLIB symmetric instances as a benchmark and computed the optimum tour on multi-core device CPU, Pthread, and many-core device on GPU.

This research uses accelerated systems such as multi-core CPU, Pthread, and GPU. We used two NVIDIA GPU cards: GeForce GTX 980 Ti and GeForce GTX 1050, and *Contech* tools to generate the different task graphs of different parallel sections.

1.1 Objectives

The principle aim of this dissertation has been to parallelize the meta-heuristics to reduce the overall execution time in an accelerated system. In this context, we can use two ways for full utilization of GPU hardware: first is to generate neighborhood

on CPU and evaluate it on GPU, and second is generation and evaluation both are on GPU. In the first approach the large data size generation of neighborhood takes a long time either on CPU or GPU. If it is on CPU, then for evaluation, we need to transfer data from CPU to GPU, which takes a considerable amount of time. After re-evaluating GPU, we need to transfer the solution from GPU to CPU. So this approach is not much more suitable. In the latter approach, both generation and evaluation are on GPU; there is no need to transfer data from CPU to GPU. So this approach is relatively more suitable. In particular, the objectives of this work may be summarized as follows:

1. Performance analysis of meta-heuristics for combinatorial optimization problems through different profiling tools (*gprof, gcov etc.*).
2. Executing meta-heuristics in different levels of parallelism- algorithmic level, iteration level, and solution level, and finding the appropriate parallel sections.
3. Generate and analyze the task graph using *contech tools* for each parallel section.
4. Scheduling and mapping of task graphs on modern many-core architecture.
5. Mapping task graph considering memory architecture of the accelerated system.

1.2 Motivations

In the last few years GPU have become famous for solving the combinatorial optimization problem of large-size instances. Due to its high-performance, multi-threaded, high-memory bandwidth and parallel architecture, it has become a growing interest in high-performance computing applications. Solving real and complex problems by heuristics algorithms is more time-consuming on the CPU, and these kind of algorithms do not apply to other optimization problems. Hence switching

to a meta-heuristic that applies to all problems *i.e.*, not a problem-dependent algorithm, has become attractive. However, for large-size instances, it takes several years or even unlimited time to give acceptable solutions. Hence with the help of GPU, it gives the solutions in a reasonable time. It motivates us to parallelize the meta-heuristics algorithm on GPU. While using a GPU, there are three main challenges in which still optimization is possible.

- Efficient communication must be required to optimize data transfer from CPU to GPU and vice versa.
- Control of parallelization, which issues are thread generation and mapping with data input.
- Efficient memory management, which emphasizes the required operation, can be done on the most suitable memory.

As QAP is one of the most popular optimization problems, it has many applications in different domains. Solving QAP using heuristics methods for large instances takes a lot of time. So it motivates us to solve QAP using different meta-heuristics on GPU to reduce the execution time further and improve the quality of optimal solutions.

PPP is one of the NP-complete class problems which is the best suitable for smart card applications. We can test it with large randomly generated instances, which motivates us to do more parallelization on the GPU to achieve the best performance of the GPU.

TSP is similar to the QAP problem, with few variations, which motivates us to implement TSP and analyze the performance of the GPU.

1.3 Efficient Parallelization of Meta-heuristics

E.G. Talbi [10] classified parallelization of meta-heuristics into three levels:algorithmic level, iteration level and solution level which can be shown in Figure 1.1.

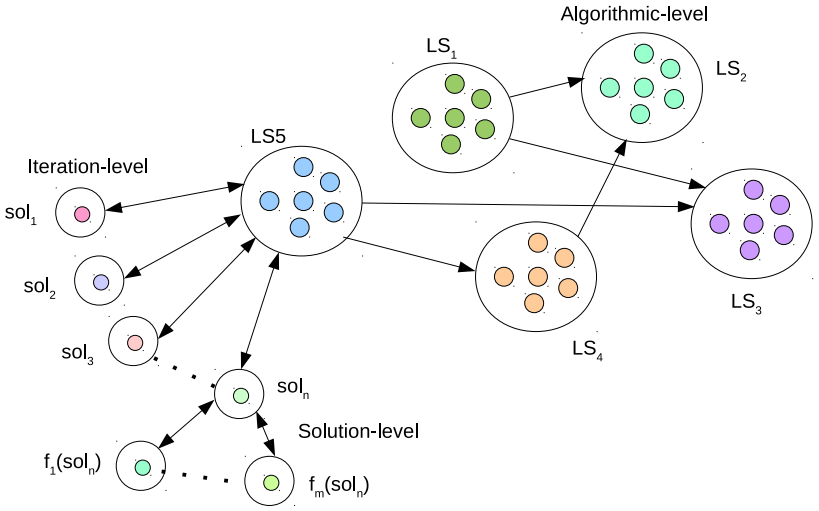

FIGURE 1.1: Parallel model of meta-heuristics

- **Algorithmic Level:** In this level, parallelization of meta-heuristics is problem-independent, *i.e.*, different meta-heuristics are executed in parallel without cooperation. Each meta-heuristic may have different initial solutions for S-meta-heuristics or different initial populations for P-meta-heuristics. In addition to the initial solution, each meta-heuristic may have a different parameter like the tabu list size for tabu search, mutation, cross-over for the evolutionary algorithm, etc. This parallel model is based on the master-worker model in which workers implement meta-heuristics. The master defines the different parameters used by the workers and determines the best solution found among all workers.

- **Iteration Level:** In this iteration level, the aim is to parallelize every iteration of the meta-heuristics, which is problem-independent. Most of the meta-heuristics belong to this level. For S-meta-heuristics, generation and evaluation of the neighborhood can happen parallel. The generation of a neighborhood can be parallelized by dividing the neighborhood into different partitions. Each partition can be evaluated in parallel so that we can find the best solution in a reasonable time. As each iteration progresses, the best solution is improved until the stopping criteria are reached. Similarly, for P-meta-heuristics, the initial population is divided, and the operation on each element is in parallel.

- **Solution Level:** In this solution level, problem-dependent operations performed on solutions are parallelized. This model is particularly used when the objective function or the constraints are time, memory-consuming, and input-output-intensive. Generally, objective functions are partitioned into different partial functions, and each function is evaluated in parallel. So one has to wait for all partial function evaluations before sending the final result. In some problems, objective functions require access to a huge database that a single machine cannot manage. The database is distributed among different sites in this situation, and data parallelism is used to evaluate the objective function. In data parallelism, the same identical function is evaluated on a different partition of the input data of the problem.

The main aim of parallelizing the meta-heuristics is to reduce the execution time. In this context, we can use two ways for full utilization of GPU hardware; the first is to generate neighborhood on CPU and evaluate it on GPU and the second is generation and evaluation both are on GPU. Now, the first approach for the large data generation of neighborhood takes a long time either on CPU or GPU. If it is on CPU, then for evaluation, we need to transfer data from CPU to GPU, which takes a tremendous amount of time, and after evaluation on GPU again, we need to transfer the solution from GPU to CPU. So this approach is not much more suitable. In the latter approach, both generation and evaluation on GPU, there is no need to transfer data from CPU to GPU.

For the generation of the neighborhood, generally used local search algorithm, shown in Figure 1.2. First, generate an initial solution, and at each iteration, a set of neighboring solutions is generated and evaluated. The best solution is selected and replaced with the current one. This process is continuously repeated until the stopping criteria are reached.

1.4 Thesis Contributions

The major contributions of this thesis can be summarized as follows:

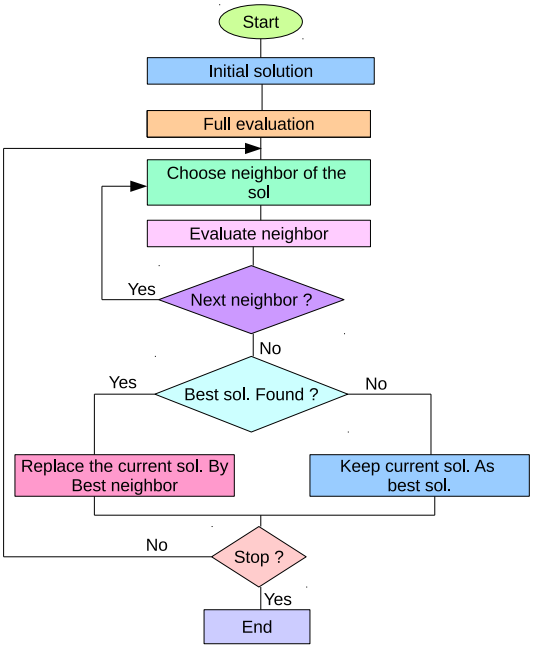

FIGURE 1.2: General model for local search meta-heuristics

1.4.1 Analysis of Meta-heuristics for Quadratic Assignment Problem in Accelerated Systems

First generate random initial solutions. At each iteration from initial solutions, generate several neighbor solutions using adjacent pair-wise exchange permutation method and evaluate; among these, the best solution is found and replaced with the current solution. This process continues until the stopping criteria are reached. For implementing ILS on serial machine (CPU), above all process is repeated and evaluated all the $\frac{n\times(n-1)}{2}$ neighbor solutions where n is the size of each solution. At this stage, local optima reached; thus, for the next iteration, made the minimal local solution the initial solution and repeated all the steps above. This process continues until the stopping criteria reach. To avoid being stuck in local optima or unable to get the best global minimum solution, took 500 random initial solutions, and each initial solution iterated 10 times. *ILS using Pthread:* In this meta-heuristic, instead of executing all initial solutions at a time, we assigned all the initial solutions to several processors so that each could be utilized and get an equal number of the initial solutions. Then, each part is assigned to different processors so that all parts run in parallel. *ILS on GPU:* In general, there are three issues

in GPU where optimization is possible: 1) efficient communication between CPU and GPU, in which the main issue is to optimize data transfer; 2) control of parallelization, in which the issues are thread generation and mapping the threads to data input; and 3) efficient memory management, which can be performed on any memory. The parallel design of meta-heuristics to solve any combinatorial optimization problem, such as the QAP, has a significant performance effect on accelerated machines. In general, to run a program on GPU, three main steps have to be followed: first, the data input is copied from CPU (host) to GPU (device); second, the program is executed on GPU; and last, the result is sent back from GPU to CPU. First, we set the kernel's grid size and block size to generate a number of threads to achieve the best performance with respect to executing the ILS algorithm on GPU. For the implementation, we generated random initial solutions on CPU, and from each initial solution, we generated neighbor solutions and evaluated it on GPU. In this study, we took 500 initial solutions, and each initial solution iterated 10 times to get the best possible optimal solution. All the initial solutions were run in parallel and called the GPU, which also ran in parallel to execute the evaluation cost function of neighbor solutions.

SA on the serial machine: To find the best solution among all neighbor solutions, we applied SA. We fixed the main parameters of SA to be as follows: initial temperature: $10,000$, cooling rate: 0.9999, and absolute temperature: 0.00001. Initially, the algorithm starts with an initial temperature, and at every iteration, the temperature is reduced to [current temperature $\times$ cooling rate], making it the current temperature for the next iteration. This process continues until the current temperature reaches the absolute temperature. *SA using Pthread:* Instead of running all of the initial solutions sequentially on the CPU, we assigned them to an equal number of processors. The optimal solution was found and compared on each processor to get the globally optimal solution. *SA on GPU:* First, we generated neighbor solutions from the initial solutions and evaluated them on GPU. Then, the neighbor solutions' entire assignment cost is copied back from GPU to CPU. Afterward, we performed SA on the CPU to find the optimal solution, and all neighbor solutions were evaluated in parallel by GPU.

GA on the serial machine: GA starts with a random initial population of solutions, where the neighbor solutions are generated with the help of two operators, namely, crossover and mutation. The crossover operator selects two random solutions from the population of solutions. In this study, we used the tournament method. In this method, first, a few solutions are randomly selected. Then, among them, the best solution (with the least assignment cost) is selected. When two solutions are selected, the crossover operator is employed to generate the new offspring and then store it in the new population. In this study, we used one-point crossover and then applied the mutation operator to the newly generated solution. We changed the elements' position for mutation and stored them in the new population. In addition, we fixed the number of initial solutions as 5000 and the number of iterations as 10. From the 5000 initial solutions, we generated 25000 neighbor solutions using the crossover and mutation operators. In each iteration, the best population of solutions was selected (i.e., 5000), becoming the current population, with this process continuing until the termination condition is reached. *GA using Pthread:* In this study, the neighbor solutions are assigned to an equal number of processors, and each processor evaluates the solutions and stores their costs, and then merges all solutions, out of which we chose the best 5000 solutions for the next generation or iteration. *GA on GPU:* Using GA, we generated the random initial population of solutions, then generated the offspring using the crossover and mutation operators for the next generation and iteration. In GPU, we evaluated the solutions in parallel until the stopping condition is reached, then the results were sent back from GPU to CPU.

PSO on the serial machine: It starts with random initial solutions or a population of particles, with each particle having a random initial velocity. Every particle has it's personal best based on its experience or history and global best for the whole group of particles. In every iteration, each particle updates its velocity and position. As the iteration continues, each particle converges toward the optimal solution. In this study, velocity was measured in terms of the number of swaps of positions inside a solution. *PSO using Pthread:* on pthread, an initial population of solutions is assigned to an equal number of processors. As the algorithm

proceeds, every processor reaches the optimal solution. After the termination, all the optimal solutions for the processors are combined, and among them, the best solution is observed. *PSO on GPU:* In PSO, we observed the random position and velocity of the whole swarm, which is generated on the CPU. The particle's position and velocity were used to generate the next position of the particles, which was evaluated in parallel on GPU. Every member of the swarm updates its velocity based on the position from the local optimum. After reaching the stopping criteria, the final global optimum values were copied from GPU to CPU.

CSA on the serial machine: In CSA, first, we fixed the initial parameters such as the flight length and probability of awareness. Initially, for the serial machine, we generated a fixed number of initial solutions as the size of the input; with each initial solution, we generated many neighbor solutions. Among these neighbor solutions, we found the best one, which was used to initialize the memory of each crow for each initial solution. As the iteration increases, each crow updates its memory until the termination condition is reached. Among the memories of all crows, we found the best one giving the optimal solution for the QAP, taking flight length as 2 and probability of awareness as 0.15. *CSA using Pthread:* To implement on Pthread; first, we divided the initial solutions among processors, with each processor running the maximum number of iterations (initially fixed) and updating the memory of each crow. All processors run in parallel, while inside each processor, this algorithm runs serially. After completing the execution of all processors, we found the best-updated memory among all crows that gives our optimal solutions. *CSA on GPU:* To implement it on GPU; first, we fixed the initial parameters. Then, we calculated the cost of the initial solutions on the CPU, and from each initial solution, we generated the neighbor solutions. Afterward, we evaluated the cost of the neighbor solutions on GPU. From each initial solution, we found the best possible neighbor solution cost that is set to the memories of each corresponding crow (for the initial solution). Consequently, we run the CSA on the CPU to find the next positions of the crows, evaluate their cost, and compare them to the solution stored in their memory. If the solution gave the best result, then the corresponding crow's memory was updated until the

termination criteria were reached. Finally, among the memories of all crows, we found the best solution cost.

TS on the serial machine: In TS, first, we fixed the size of the tabulist as the size of the instance. Then, we generated the fixed number of (size of tabulist) random initial solutions, evaluated their cost, and stored them in tabulist. In each iteration, we generated the neighbor solution through an adjacent pair-wise exchange method and evaluated their cost; if it improves from the current solution, then we updated the tabulist. This procedure will continue until it reaches the terminating condition and finds the best optimal solution from tabulist. To avoid being stuck in local minima, we implemented a diversification operator suggested by Glover et al. [11] to generate a new solution. We fixed the maximum number of failures as the size of the instance. In CPU, we fixed the number of iterations as 10. *TS using Pthread:* To implement on pthread; first we fixed tabulist size as the size of the instance, and then generated a random initial solution, evaluated their cost, and stored in tabulist. We call Pthread, and every thread gets the tabulist and generates a neighbor solution; if it improves the current solution, then update the tabulist. When Pthread joins, then merge all the thread in tabulist, and from that, we get the best optimal solution. *TS on GPU:* To implement TS on GPU, first we fixed the GPU parameters and then transferred the input data from CPU to GPU. In GPU, we fixed the number of iterations as 10, and the size of tabulist is the size of the instance. In each iteration of TS, we transfer the tabulist from CPU to GPU. On GPU, we generate and evaluate the neighbor solution, update the tabulist with the best optimal solution, and then update tabulist is transferred back from GPU to CPU. This process continues until it reaches the stopping condition. Finally, the best optimal solution is found from tabulist.

This work is fully discussed in Chapter 4.

1.4.2 Analysis of Iterated Local Search Meta-heuristic on GPU Spatial Memory

Here ILS meta-heuristic is used for solving QAP on GPU spatial memory and compared the execution time and speedup on GPU. The GPU supports programmable memory, where a user can write a program for the use of memory to utilize the resources of GPU architecture. Besides the global memory of GPU, it also has fast memory systems such as the shared, constant, texture, and local memory. By using the GPU spatial memory, it can be further reduce the execution time for solving QAP. The GPU shared memory is used when data is required to be accessed by all threads within a block. The GPU constant memory is used for read-only data accessed uniformly by threads in a warp. It performs best when all threads in warp access the exact location in the constant memory. Texture memory is also read-only; it performs best when all reads in a warp are physically adjacent to each other. The GPU local memory is used when data is required to be accessed by only a particular thread *i.e.* data is only visible to the thread that wrote it and ended when threads are destroyed. We implemented the ILS meta-heuristic using the local, shared, and constant memory, and the results are recorded. Because the QAP input, such as the distance matrix and flow matrix, is always constant, we put it on a read-only memory (the constant memory). So here, GPU is performing best on shared memory, while on constant memory, GPU performance is worst. We used Nvidia GeForce GTX 980 Ti GPU card, which has only 48KB constant memory or cache, so instances of size greater than 100 show out of memory and not be executed. We also implemented texture memory along with varying one with constant and texture memory, like we have put one-time distance matrix input on constant memory and flow matrix on texture memory. We observed that when we put on mixed with constant and texture memory, speedup on GPU is more significant than only constant memory but worst than only texture memory.

More details about this work are given in Chapter 5.

1.4.3 Analysis of Meta-heuristics for Traveling Salesman Problem in Accelerated Systems

In this work, we analyze the performance of meta-heuristics for solving TSP. Here we considered both types: Single solution-based (s-type) and Population-based (p-type) meta-heuristics. We have taken s-type as ILS, SA, and CSA in this study and p-type as GA, PSO, and TS. We also compared the optimum tour and execution time between CPU, Pthread, and GPU and computed the speedup on GPU. Here we have taken instances from TSPLIB [12] symmetric instances. We generated neighbor solutions using the adjacent pair-wise exchange permutation method and used incremental solution evaluation instead of executing the full solution at a time. For incremental solution evaluation, we passed only the position of cities or locations which is changed, and due to this, we calculated the change of cost in the tour. We used the same process and same parameter as used in the QAP for all the above meta-heuristics. We compared all meta-heuristics for each TSP symmetric instance on GPU and calculated the speedup on GPU concerning the CPU. We observed that for instance *pr107*, CSA is giving the highest speedup 50.53, and ILS is giving the best speedup for all other instances on GPU.

A more detailed description of this work is given in Chapter 6.

1.4.4 Analysis of Meta-heuristics for Permuted Perceptron Problem in Accelerated Systems

In this work, we implemented PPP using SA meta-heuristic and used the binary encoding method to generate the neighbor solution from the initial solution. We generated one and two hamming distance neighborhoods. We observed that the number of neighbor solutions increases as the hamming distance increases and the GPU speeds up. *SA on the serial machine:* In CPU, first we randomly generated the ϵ-matrix I of size $m \times n$ and a ϵ-vector Z of size $n \times 1$; then we find the multi-set matrix IZ of size $m \times 1$, using this we calculated the histogram of elements of a multi-set matrix. We again also generated a random ϵ-vector Z

as the candidate vector initial solution, and from that, we generated a neighbor solution using hamming distance one. From each neighbor solution, we calculated the multi-set matrix of size $m \times 1$. We calculated the energy function, then applied SA to find the candidate vector solution. In the next iteration, we reduced the temperature by the reducing factor at each iteration; this iteration will continue until the temperature becomes 1. Finally, we noted the solution and made them an initial solution for the next iteration, and this process will continue until it reaches the maximum number of iterations. *SA on GPU:* For the implementation of SA on GPU, first, we generated random ϵ I matrix of size $m \times n$ and ϵ Z matrix of size $n \times 1$. We find the multi-set matrix IZ on GPU, and results are copied back from GPU to CPU. We also calculated the histogram of each odd element of the multi-set matrix. Again we generated a random initial candidate vector of size $n \times 1$, and from this initial solution, we generated neighbor solutions using binary encoding with hamming distance one. We calculated the objective function for each neighbor solution and applied the SA method to accept or reject the neighbor solution. For each iteration, the SA method continues to run until its temperature parameter T becomes one. We fixed the maximum number of iterations as 1000 and T as the value of n. We also generated a neighbor solution with a hamming distance of 2 and used SA meta-heuristics to solve PPP. We noted the result in Table 7.4. From this table, we can see that the highest speedup on GPU is 196 is obtained for instance size $121 - 81$ while for instance size $151 - 167$, the speedup is 166. We analyzed the performance of the SA meta-heuristic on a sequential machine and a GPU. First, by using a GNU profiling tool (i.e., *gprof*) [13], the performance of SA meta-heuristics is checked for solving PPP on a serial machine. We analyzed the performance of PPP, for instance size $(101 - 117)$, by randomly generating $Imatrix$ and $Zmatrix$. We run 1000 iterations for each solution on the sequential machine (i.e., CPU) and noted the results of the *gprof* profiling tool. We generated the task graphs of each parallel section using the *Contech* tools.

The detail description of this work is given in Chapter 7

1.5 Summary

Many researchers have shown an interest in solving optimization problems and their applications using meta-heuristics in accelerated systems. We attempted QAP using six meta-heuristics: ILS, SA, GA, PSO, CSA, and TS, and took instances from QAPLIB. We considered mainly four different classes of different instances. We implemented it on CPU, pthread, and GPU and examined the speedup on pthread and GPU regarding CPU. The PSO meta-heuristics show the highest speedup on GPU among all six meta-heuristics. We also notice that as the size of the instance increases, the GPU resources are utilized more efficiently, which results in more speedup on GPU. Additionally, as QAP input is read-only, we implemented ILS on GPU spatial memory such as shared, constant, and texture for QAP, which increased the speedup compared to utilizing just the global memory and enhanced GPU performance.

All the above six meta-heuristics are used for TSP in accelerated systems. Here we considered the symmetric instances from TSPLIB. Among all these meta-heuristics, CSA shows the highest speedup on GPU, and TS shows the lowest speedup on GPU. Similarly, we used the SA meta-heuristic to examine the speedup on GPU for PPP. For PPP, we randomly generated the input instance with different sizes and examined the speedup on the GPU.

In this dissertation, main aim to parallelize the different meta-heuristics and compare them to get the most suitable meta-heuristics for accelerated systems optimization problems.

1.6 Organization of Thesis

The rest of this thesis is organized as follows:

- Chapter 2 described the introduction of optimization problems and meta-heuristics.

- Chapter 3 discusses the introduction of multi and many-core architecture.
- Chapter 4 presents the first contribution, which is the parallelization of six meta-heuristics: ILS, SA, GA, PSO, CSA, and TS in accelerated systems: multi-core CPU, Pthread, and many-core GPU for QAP.
- Chapter 5 described the logical extension of GPU hardware spatial memory for QAP using the ILS meta-heuristic.
- Chapter 6 discusses another optimization problem TSP similar to QAP. We used all six meta-heuristics: ILS, SA, GA, PSO, CSA, and TS in accelerated systems.
- Chapter 7 demonstrates the parallelization of the simulated annealing meta-heuristic for PPP in a many-core GPU architecture.
- Chapter 8 finally concludes the thesis.

Chapter 2

Introduction to Meta-heuristics

As described in Chapter 1, solving complex real-life optimization problems using heuristics methods takes tremendous time for instances greater than 20. So, highly massively parallel device architecture like GPU may reduce the execution time and improve the optimal solution. The main goal of this thesis is to find which meta-heuristics is most suitable for the optimization problems we have considered. We initially summarize the preliminary concept of optimization problems and meta-heuristics. We also discussed the basic algorithm used in each meta-heuristic.

The layout of this chapter is as follows: optimization problem is discussed in section 2.1. Section 2.2 describes the meta-heuristics, solution representation is described in section 2.3. Section 2.4 discusses the different meta-heuristics, performance analysis of meta-heuristics is summarized in section 2.5 followed by the summary of the chapter in section 2.6.

In this thesis, the terms instance or instances means that it is the input or benchmark of the optimization problem.

2.1 Optimization Problems

An optimization problem is the minimization or maximization of a cost function, which can be mono-objective or multi-objective. This (these) function (s) is (are) called as objective function (s). In this document, we are considering optimization problems in a minimization context.

$$\begin{aligned} &Mono\text{-}objective && min\, f(x), x \in S \\ &Multi\text{-}objective && \begin{cases} min\, f(x) = (f_1(x), f_2(x), \ldots, f_n(x)) \quad n \geq 2 \\ const.x \in S \end{cases} \end{aligned} \tag{2.1}$$

where n is the number of objectives, $x = (x_1, x_2, \ldots, x_k)$ is the vector representing the decision variables and S is the set of feasible solutions. $f(x) = (f_1(x), f_2(x), \ldots, f_n(x))$ is the vector of objectives to be optimized. For $n = 1$ the objective function is called as mono-objective and for $n \geq 2$ is called as multi-objective optimization [14].

A mono-objective optimization problem aims to find a feasible solution that minimizes the objective function. In contrast, a multi-objective problem aims to find the set of *Pareto* optimal solution, which is called the *Pareto front*. A solution is called Pareto optimal if it is impossible to improve a given objective without detriment any other objective.

For solving an optimization problem, proper optimization methods depending upon the complexity of the problem, are utilized. Two types of optimization methods are mainly utilized: exact methods and heuristics. Figure 2.1 illustrates the different methods of optimization. Exact methods give the optimal solutions and guarantee their optimality. However, the exact methods (branch and bound, constraint programming, and dynamic programming) become impractical for large instances of the problems [15, 16, 17].

Conversely, heuristic methods generate high-quality optimal solutions, but there is no guarantee that optimal solutions are reached. Further heuristic can be classified

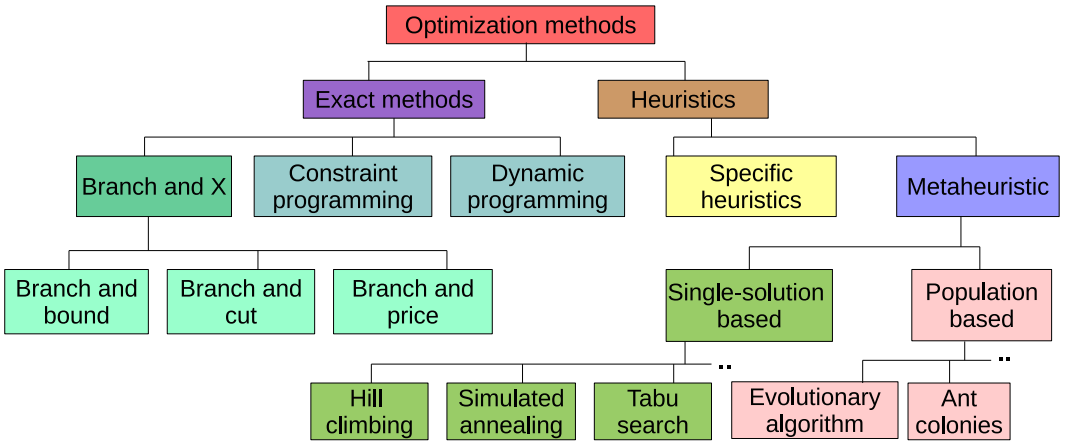

FIGURE 2.1: Types of optimization methods

as: specific heuristic and meta-heuristics [10].

An specific heuristic is designed to solve a particular problem or instance, whereas a meta-heuristic is generic and applicable to different problem types. Meta-heuristics generally use randomization and local search they are based on the iterative improvement of either a single solution or a population of solutions.

2.2 Meta-heuristics

The word meta-heuristic made with two words meta (means upper) and heuristic (means to search). So meta-heuristic is defined as an upper-level heuristic design that helps to find or generate an acceptable optimal solution to an optimization problem. However, it does not guarantee to find a globally optimal solution [18].

Heuristics are problem-dependent and deterministic techniques usually adapted to the problem at hand. In order to maximize their efficiency, they take advantage of the peculiarities of this problem. Thus, heuristics are usually too greedy and tend to become trapped in local optima, which prevents them from obtaining the optimal global solution. As opposed to this, meta-heuristics are non-deterministic techniques independent of the problem. In this way, they can serve as black boxes since they do not exploit any particularities of the problem. Their solution may even deteriorate temporarily as they are not greedy.

2.2.1 Classification of Meta-heuristics

Meta-heuristics are generic, randomized, and use local search to improve the solution based on iterative improvement. These are classified into two categories: Single-solution based meta-heuristic (S-meta-heuristics) and population-based meta-heuristics (P-meta-heuristics).

2.2.1.1 Single-solution-based Meta-heuristic

Single solution-based meta-heuristics (s meta-heuristics) manipulate and transform a single solution during the search. These meta-heuristics are based on exploitation-oriented *i.e.* they can intensify the search in local regions. They are iterative techniques successfully applied to solve many real and complex problems. They usually start with a randomly generated initial solution. As the algorithm iterates, the current solution is replaced by another one chosen from its neighboring solutions. Famous examples of S-meta-heuristics are hill climbing, tabu search, simulated annealing, iterative local search, and variable neighborhood search.

2.2.1.2 Population-based Meta-heuristic

The whole population of solutions is evolved in population-based meta-heuristics (P meta-heuristics). These meta-heuristics are based on exploration-oriented *i.e.* they provide better diversification in the entire search space. They start from an initial population of solutions, and then they iteratively apply to generate the new population and replace the current population with the newly generated population. This process is carried out until the given stopping criteria are satisfied. Famous examples of P-meta-heuristics are evolutionary algorithms, ant colony optimization, scatter search, and particle swarm optimization. The main sub-classes of evolutionary algorithms are genetic algorithms, genetic programming, and evolution strategies.

2.3 Solution Representation

Designing any iterative meta-heuristic requires an encoding (representation) of a solution. The encoding plays a significant role in any meta-heuristics efficiency and effectiveness, so it constitutes an essential step in designing a meta-heuristic. The encoding must be suitable and relevant to tackle the optimization problem. A representation must have:

- Complete, that is, all solutions associated with the problem must be represented.
- A search path must exist between any two solutions of the search space, especially the global optimum solution that can be attained.
- Easy to manipulate by the search operators.

Generally, four primary encoding techniques are used to represent a solution: binary encoding (e.g., Knapsack problem, Satisfiability problem), vector of discrete values (e.g., location problem, assignment problem), permutation (e.g., Traveling Salesperson Problem, Scheduling Problems) and vector of real values (e.g., continuous functions).

2.3.1 Binary Encoding

In a binary representation, a solution is represented by the vector (string) of bits. Binary encoding is based on *hamming distance*. This distance represents the number of positions between two strings of equal length in which corresponding symbols differ.

1-***Hamming distance neighborhood:-*** In this representation, the neighborhood is generated by flipping one bit of the candidate vector solution (as shown in Figure- 2.2). If the solution size is n, then the possible number of neighbors is n, each of size n.

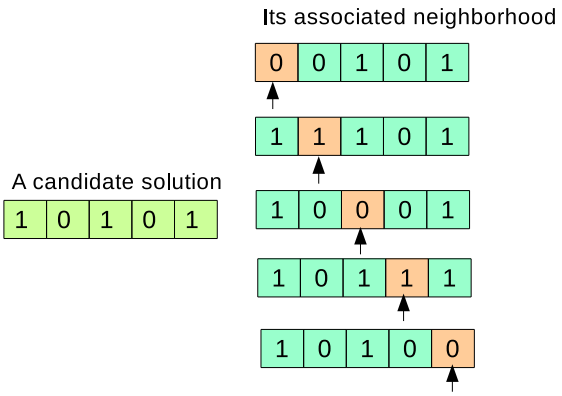

FIGURE 2.2: Binary encoding for 1-hamming distance

2-***Hamming distance neighborhood:-*** In this representation, a neighborhood is generated by flipping two values of a candidate vector solution. For a candidate solution of size n, total number of possible neighborhoods is $\frac{n\times(n-1)}{2}$.

Similarly for 3-*hamming distance neighborhood*, a neighborhood is generated by flipping three values of a candidate solution and total number of possible neighborhoods from a size n is $\frac{n\times(n-1)\times(n-2)}{6}$.

2.3.2 Discrete Vector Representation

It is a variant of binary encoding with an extension that uses the alphabet $\sum$. Each variable in this representation has a value from the alphabet $\sum$. For a discrete vector of size n, the number of the neighborhood is $(k-1)\times n$ if the cardinality of the alphabet $\sum$ is k. Now for example consider a discrete vector of size $n=3$ and alphabet $\sum=\{0,1,2,3,4,5\}$ with cardinality $k=6$. A neighborhood can be generated by replacing every element of the candidate solution with the alphabet, each one at a time. All the neighbors are shown in Figure 2.3. The total number of neighborhoods generated is 15 (5×3).

2.3.3 Vector of Real Values

This representation is mainly used in continuous optimization. Such a representation has the solution space as its neighborhood. The ball is a notion used to

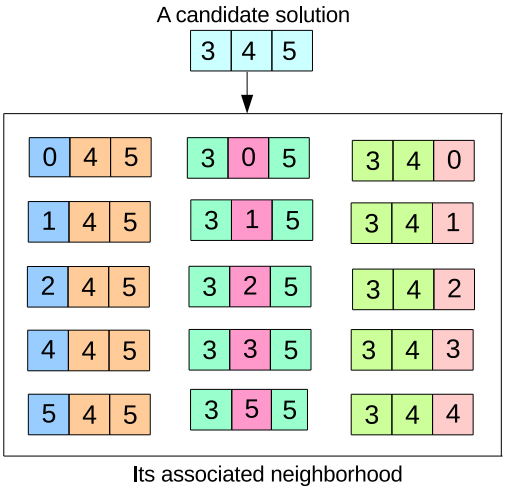

FIGURE 2.3: Discrete vector representation

define the neighborhood by R. Chelouah and P. Siarry [19]. Having a radius of r and being centered on s, a ball with the formula $B(s, r)$ includes all of the points s' such that $\|s' - s\| \leq r$. A collection of balls with a radius of $h_0, h_1, \ldots, h_m$ are considered with the current solution s as their center to achieve the homogeneous exploration of the space.

Thus the space is partitioned into crowns $c_i(s, h_{i-1}, h_i)$ such that

$$c_i(s, h_{i-1}, h_i) = s' | h_{i-1} \leq \|s' - s\| \leq h_i. \tag{2.2}$$

By randomly selecting one point from each crown c_i, with i ranging from 1 to m (as illustrated in Figure 2.4), the m neighbours of s are determined. One thread to at least one neighbour, which corresponds to one point inside each crown, is connected with the mapping. This mapping is applied to the Weierstrass function [20].

2.3.4 Permutation Representation

- 2-Exchange Neighborhood: To generate a neighborhood using the pair-wise exchange method is a standard way in permutation problems. For a permutation of size n, the total number of possible neighborhood is $\frac{n \times (n-1)}{2}$. Figure 2.5 shows all the possible neighbors of candidate solution $\{1, 0, 3, 2\}$ by exchanging the pairs of candidate solutions.

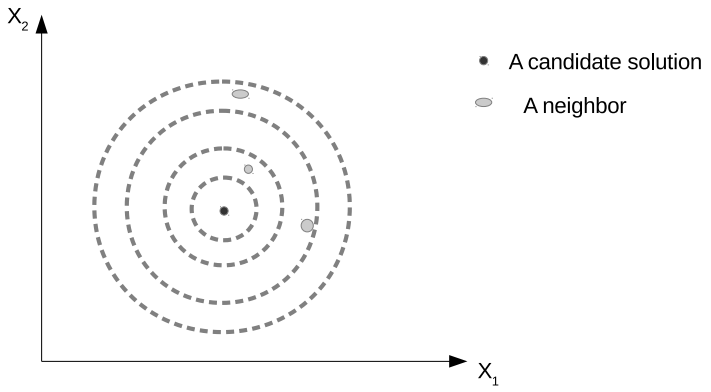

FIGURE 2.4: A neighborhood for a continuous problem of two dimension

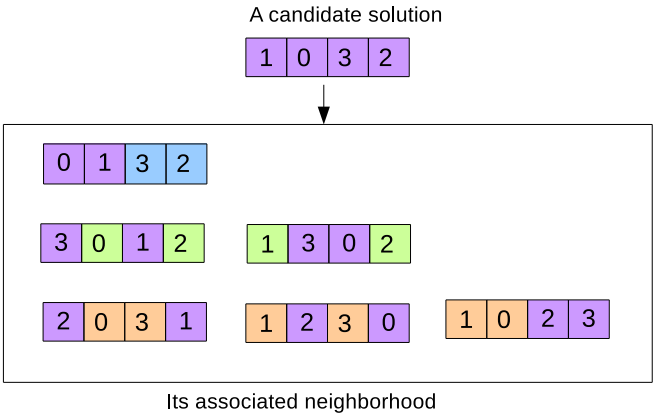

FIGURE 2.5: A neighborhood for permutation representation

Permutation encoding differs from all previous representations because the mapping between neighbor and GPU thread is not straightforward. The neighbor consists of two indexes (a swap in a permutation), and each thread has a unique identifier. So, one mapping has to be considered to convert one index into two, and another one to convert two indexes into one.

Proposition 2.1 (Two-to-one index transformation)**.** *If i and j are the permutation representation's indexes for the two elements to be swapped, then the equivalent index $f(i,j)$ in the neighborhood representation is equal to $i \times (n-1) + (j-1) - \frac{i \times (i+1)}{2}$, where n is the permutation size.*

Proposition 2.2 (One-to-two index transformation)**.** *For an element with index $f(i,j)$ in the neighborhood representation, the equivalent index i in the*

permutation representation is equal to

$$n - 2 - \left\lfloor \frac{\sqrt{8 \times (m - f(i,j) - 1) + 1} - 1}{2} \right\rfloor$$

and j is equal to $f(i,j) - i \times (n-1) + \frac{i \times (i+1)}{2} + 1$ in the permutation representation, where m denotes the neighborhood size and n denotes the permutation size.

The proof of the above two proposition can be found in [21]. This mapping is applied to solve the quadratic assignment problem.

- 3-Exchange Neighborhood: A neighborhood is generated by exchanging three neighborhood values for large instances. For a neighborhood of size n, total number of possible neighborhood is $\frac{n \times (n-1) \times (n-2)}{6}$. For this mapping, a transformation is needed to convert three-to-one and one-to-three index transformations.

2.4 Different Meta-heuristics

Meta-heuristics do not guarantee finding a global optimal solution. They are classified into two types: single solution-based and population-based. In single solution-based meta-heuristics, a single solution is used for manipulation and transformation during the search, whereas in population-based meta-heuristics, the whole population is involved during the search. Moreover, in single solution-based meta-heuristics, we can use the pairwise exchange method to generate neighbor solutions that can be assessed using the incremental solution evaluation. However, in population-based meta-heuristics, the generated neighbor solutions cannot be assessed using the incremental solution evaluation, but we can evaluate a full solution utilizing the constant memory, local memory, and shared memory of GPU.

Algorithm 1 Iterated Local Search (ILS)

Input: Distance matrix and Flow matrix
Output: Optimal solution (local optimum).

1: $s = s_0$; (Generate random initial solution s_0)
2: **while** not Termination_Criteria **do**
3: **repeat**
4: s_b the best neighbor solution near s, and $s_b = s$
5: Generate neighbor solution s' from s (where $s' \in N(s)$)
6: Evaluate the neighbor solution s' using objective function F.
7: **if** $F(s')$ is less than $F(s_b)$ **then**
8: $s_b = s'$
9: **end if**
10: **until** All the neighbor solutions of s are explored
11: for next iteration $s = s_b$
12: **end while**

2.4.1 Iterated Local Search (ILS)

It is a single solution-based meta-heuristic, which starts with an initial solution that is randomly generated. At each iteration, it explores all possible neighbor solutions generated from the initial solution, then evaluates the objective function of each neighbor solution and compares it with the objective function of the current solution. If it improves the cost function, then it replaces the neighbor solution as the current solution. However, if all neighbor solutions are worse than the current solution, it stops, indicating that the local optimum is reached. The basic algorithm of ILS is illustrated using Algorithm 4.1.

2.4.2 Simulated Annealing (SA)

It is a stochastic single solution-based meta-heuristic, which generally starts with an initial solution and an initial temperature parameter T. At each iteration, a neighbor solution is generated, which is always accepted if it improves the cost function, and accepted with probability if it is a non-improving solution, with the probability of an accepted solution decreasing as the algorithm progresses. It can be also formalized as if a current solution S_c has cost function $f(S_c)$, while

Algorithm 2 Simulated Annealing (SA)

Input: Cooling schedule
Output: Optimal solution.

1: $s = s_0$; (Generate random initial solution s_0)
2: $T = T_{max}$; Set initial maximum temperature
3: **while** not Termination_Criteria *i.e.* $T < T_{min}$ **do**
4: **repeat**
5: Generate random neighbor s' from s (where $s' \in N(s)$)
6: Evaluate the neighbor solution s' using objective function f.
7: $\Delta E = f(s') - f(s)$;
8: **if** $\Delta E \leq 0$ **then**
9: $s = s'$ /* Accept the neighbor solution */
10: **else**
11: Accept s' with a probability $e^{\frac{-\Delta E}{T}}$
12: **end if**
13: **until** Equilibrium condition reached
14: $T = g(T)$; /* Update the temperature */
15: **end while**

the next neighbor solution S_n has cost function $f(S_n)$. For a minimization problem, if $f(S_n)$ is less than $f(S_c)$, the solution S_n is accepted and assigned as the next current solution. Otherwise, the solution S_n is accepted with probability $exp(f(S_c) - f(S_n))/T$, where T is the current temperature. The basic SA algorithm is illustrated using Algorithm 2.

2.4.3 Genetic Algorithm (GA)

It is a population-based meta-heuristic that belongs to the larger class of EAs, which are generally good at diversification. It starts with many initial solutions and with the help of three genetic operators (i.e., crossover, mutation, and selection), then it generates the next generation population of solutions, continuing this process until reaching the termination criteria. In this study, we use a single-point crossover operator to combine two randomly chosen solutions to generate two new solutions. The mutation operator is used to randomly change the position of some elements in the selected solution, which generates a new solution that prevents the sticking in local optima. The cost of all solutions is evaluated using the objective function, which is then used to decide whether to eliminate or

Algorithm 3 Genetic Algorithm

Input: Distance matrix and Flow matrix
Output: optimal solution.

```
1: Generate P=(P(O)); /* Generate random initial population */
2: while not Termination_Criteria do
3:     repeat
4:         Generate offspring P' with crossover operation and from P
5:         Generate offspring by applying mutation operation on P'
6:         Evaluate the offspring P' using objective function F.
7:         Find best population P_b
8:     until All offspring of P are explored.
9:     Replace P with P_b for the next generation
10: end while
```

retain these solutions. To generate a new population, low-cost solutions are kept, whereas high-cost solutions are discarded, then the generated new population replaces the current population, and the whole process is repeated until reaching the termination condition. The basic GA is illustrated using Algorithm 3.

2.4.4 Particle Swarm Optimization (PSO)

It is a stochastic population-based meta-heuristic optimization technique, which was developed by Russel Eberhart and James Kennedy [22] in 1995. PSO is a nature-inspired meta-heuristic based on swarm intelligence where the movement of each individual explores the search space. In a swarm, each individual gains experience, called the personal best (*pbest*), which contributes to the group and uses the experience of the group, called the global best (*gbest*), for evaluating its own self. Thus, they pass the information in both directions, namely, from an individual to the group and from the group to an individual. Shi and Eberhart [23] proposed a mathematical formulation based on the inertia weight model for the movement of particles in search space, with the coefficients being independent from the optimization problem. In PSO, each particle is composed of three vectors: $\vec{p_i}(t)$ that denotes the position vector, $\vec{v_i}(t)$ that denotes the velocity vector, and $\vec{p_{pbest_i}}(t)$ that denotes the best local position vector of particle i. The velocity of

Algorithm 4 Particle swarm optimization (PSO)

Input: Distance matrix and Flow matrix
Output: optimal solution.

```
Random initialization of position and velocity of each particle
repeat
    Evaluate the objective function f(x_i)
    for all particles i do
        Update velocities:
        v_i(t+1) = w * v_i(t) + θ_1 × (p_i − x_i(t)) + θ_2 × (p_g − x_i(t));
        Move next position: x_i(t+1) = x_i(t) + v_i(t+1);
        if f(x_i) < f(pbest_i) then
            pbest = x_i;
        end if
        if f(x_i) < f(gbest_i) then
            gbest = x_i;
        end if
        Update (x_i, v_i);
    end for
until Stopping criteria reached
```

a particle can be defined as

$$\begin{aligned} v_{id}(t+1) = w * v_{id}(t) + c_1\theta_{1d}(t)(p_{pbest_{id}}(t) - p_{id}(t)) \\ +c_2\theta_{2d}(t)(p_{gbest_d}(t) - p_{id}(t)) \end{aligned} \tag{2.3}$$

and the next position of the particle can be written as

$$p_{id}(t+1) = p_{id}(t) + v_{id}(t+1) \tag{2.4}$$

where $p_{id}(t)$, $v_{id}(t)$, $p_{pbest_{id}}(t)$, and $p_{gbest_d}(t)$ represent the current position, the velocity, the best local position, and the best global position of particle i in dimension d at iteration t, respectively. θ_{1d} and θ_{2d} denote the random variables between 0 and 1, w represents the inertia, while c_1 and c_2 are two positive constants that show the personal influence and the social influence of a particle. The basic algorithm of PSO is illustrated using Algorithm 4.

2.4.5 Crow Search Algorithm (CSA)

It is a nature-inspired meta-heuristic, which was developed by Askarzadeh [24] in 2016. This algorithm is based on crow intelligence in nature. Crows are very good in observing other birds. For example, when other birds hide their excess food in some places, the crows observe until the other birds leave, and then the crows steal their food. Once the crows commit theft, they take extra precautions by moving to another place and looking for future victims. In a crow of flock size N, the position of a crow i at time (iteration) t in a d-dimensional environment is represented by a vector $p^{i,t}$ $(i = 1, 2, \ldots, N; t = 1, 2, \ldots, t_{max})$, where $p^{i,t} = [p_1^{i,t}, p_2^{i,t}, \ldots, p_d^{i,t}]$, and t_{max} denotes the maximum number of iterations. Because each crow has a memory, $m^{i,t}$ denotes the memory (position of the hiding place) of crow i at iteration t, being the best position obtained by crow i so far. Assuming that at any iteration t, crow j wants to visit its hiding place $m^{j,t}$, and crow i decides to follow crow j, then in this situation, the following two cases may happen.

case 1: Crow j does not know that crow i is following it. Therefore, crow i will approach the hiding place of crow j. Thus, the new position of crow i is represented as

$$p^{i,t+1} = p^{i,t} + r_i \times f_l^{i,t} \times (m^{j,t} - p^{i,t}) \tag{2.5}$$

where r_i denotes a random number between 0 and 1 with a uniform distribution, while $f_l^{i,t}$ represents the flight length of crow i at iteration t.

case 2: Crow j knows that crow i is following it. Then, in this situation, crow j will fool crow i by moving to another position in the search space so that its hiding place remain protected. Therefore, by combining both cases 1 and 2, we can write the position of crow i in the next iteration as

$$p^{i,t+1} = \begin{cases} p^{i,t} + r_i \times f_l^{i,t} \times (m^{j,t} - p^{i,t}) & r_j \geq PA^{j,t}, \\ \text{a random position} & \text{otherwise} \end{cases} \tag{2.6}$$

where r_j denotes a random number between 0 and 1 with a uniform distribution, while $PA^{j,t}$ denotes the probability of awareness of crow j at iteration t. The next

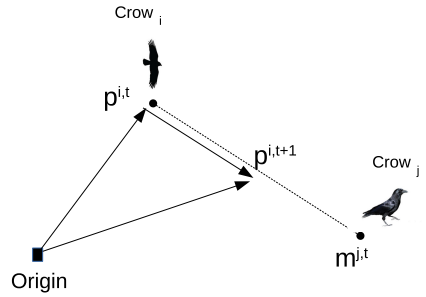

FIGURE 2.6: $f_l < 1$

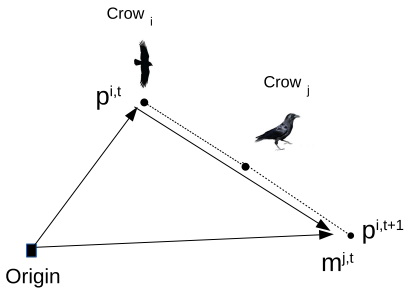

FIGURE 2.7: $f_l > 1$

Algorithm 5 Crow search algorithm (CSA)

Input: Distance matrix and Flow matrix
Output: optimal solution.

Randomly initialize the positions of crows of flock size N in a search space
Evaluate the positions of the crows
Initialize the memory of each crow
while $t < t_{max}$ **do**
    for $i = 1 : N$ **do**
        Randomly choose one of the crows among the flock of size N to follow a crow (for example j)
        Define the probability of awareness
        if $r_j > PA^{j,t}$ **then**
            $p^{i,t+1} = p^{i,t} + r_i \times f_l^{i,t} \times (m^{j,t} - p^{i,t})$
        else
            $p^{i,t+1}$ = a random position in search space
        end if
    end for
    Check the feasibility of the new positions
    Evaluate the new position of the crows
    Update the memory of crows
end while

position of crow i depends on the flight length, that is, if $f_l < 1$, then the crows next position is between $p^{i,t}$ and $m^{j,t}$, as shown in Figure 2.6. Conversely, if $f_l > 1$, then the next position of the crow can be anywhere on the line, that is, it may exceed the position $m^{j,t}$, as illustrated in Figure 2.7. Hence, small values of f_l move toward the local search, whereas a large value of f_l moves toward the global search in the search space. The pseudo code of CSA is shown in Algorithm 5.

2.4.6 Tabu Search (TS)

It is a single solution-based meta-heuristic, which is proposed by Glover [25]. It starts with a random initial solution and a memory-based tabulist. In each iteration of tabu search, all neighbor solution is generated from an initial solution and evaluated their cost to improve the current solution. In TS, to avoid local minima a tabulist is introduced, which is a fixed-length list used to store the best solution. In each iteration, the best solution is found, and compared with the tabulist, if this solution improves from the solution stored in the tabulist, then the tabulist is updated. So in this way, TS accepts not only the improved solution but also accepts the non-improved solution. The performance of this meta-heuristic depends on different parameters like neighbor solution generation and tabulist implementation.

2.5 Performance Analysis of Meta-heuristics

After designing the meta-heuristic algorithm and execution on many-core architecture their performance must be analyzed on a fair basis. The following are the essential steps for evaluation of performance of meta-heuristic algorithm:

- Design of meta-heuristic algorithms done according to the input instance and the solution of the problem.
- Statistical analysis must be done on the obtained results with the help of suitable optimization algorithms.

2.6 Summary

Real-world optimization problems are resolved using a variety of meta-heuristics. Here we considered some single solution-based and population-based meta-heuristics

to analyze their performances in optimization problems effectively. Single solution-based meta-heuristics are initialized with only one solution, while population-based meta-heuristics are initialized with a group of initial solutions. Here we discussed the single solution as ILS, SA, and TS meta-heuristics, whereas population-based GA, PSO, and CSA meta-heuristics. The solution that optimizes the objective cost function is accepted in ILS and TS meta-heuristic. In contrast, in SA, both the solutions that optimize the cost function and non optimize solutions are accepted. In GA, the best populations are selected in every iteration, and optimal solutions are searched. In PSO, each particle optimizes its best in every iteration to get the optimal solution. In CSA, each crow modernizes their memory to find the optimal solution.

Chapter 3

Introduction to Many-core Architecture

Chapters 1 and 2 discussed how meta-heuristics could be used to solve optimization problems. However, they do not guarantee that the optimal solution will be reached. It is possible to reduce execution time and improve optimal solutions using massively parallel architectures like GPUs. We summarized the multi and many-core architecture. In addition, we also discussed the different versions of GPU architecture that are launched till the year 2022 with their necessary improvements from the previous release version from NVIDIA.

The rest of the chapter is organized as follows: multiprocessor architecture is discussed in section 3.1. Section 3.2 described the architecture of GPU. CUDA programming model is presented in section 3.3. Finally section 3.4 summarizes the chapter.

3.1 Multiprocessor Architecture

In the last decades, main personal computers have been switching to include other processing elements to gain high performance. We can classify the multiprocessor architecture of the PC & work stations processor system into three categories (a)

	Number of cores	#transistors	Examples
Multi-core	4, 8, 12, 16	$\geq 55 \times 10^6$	Intel Xeon Processor
Many-core	57, 60, 61	$\geq 5 \times 10^9$	Intel Xeon Phi Co-Processor
GPU	512, 1024, 3072	$\geq 5 \times 10^9$	NVIDIA GeForce GTX Titan

TABLE 3.1: Number of cores and transistors in different processing system

multi-core architecture, (b) many-core architecture, and (c) GPU/SIMT architecture. Table 3.1 shows the number of cores, area, or transistors of the different multiprocessor systems.

3.1.1 Multi-core Architecture

In multi-core architecture, ten or a hundred cores are generally there. In a multi-core processor, two or more processors (also called cores) are built on a single computing platform, and all cores run in parallel so that it enhances the overall speed of the program. Every core is controlled separately by the operating system (OS), and the OS scheduler assigns threads and processes to distinct cores. It is physically possible to connect each processor to the same memory. IBM's POWER4 was the first general-purpose multi-core processor released in 2001 [26].

Multi-core processors are MIMD (multiple instructions and multiple data) *i.e.* different cores execute different threads (multiple instructions), operating on different parts of memory (multiple data). In multi-core devices, designers may couple cores tightly or loosely. For example, cores may or may not share caches; they may implement shared memory or message passing inter-core communication methods.

3.1.2 Symmetric Multiprocessor (SMP)

In SMP architecture, two or more similar processors are connected to a single shared main memory, providing full access to all I/O devices. SMP is controlled by a single operating system that treats all processors equally and reserves no processors for particular purposes. SMP architecture is the multiprocessor system's most popular and more accessible design.

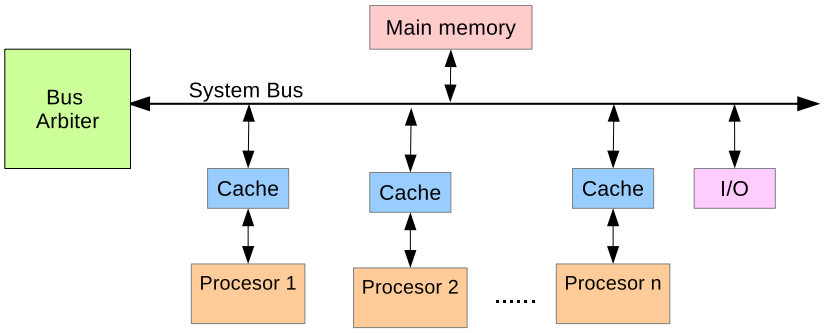

FIGURE 3.1: Architecture of symmetric multiprocessor (SMP)

Figure 3.1 shows the most common design of SMP for personal computers, workstations, and servers. The system bus connects all the processors, main memory, and I/O (input/output) devices. Each processor in a multi-processor system has a control unit, an ALU (arithmetic logic unit), registers, and cache. An SMP system has a shared memory referred to as main memory, which two or more homogeneous processors share. Each processor can communicate through the common data area of shared memory, but the direct signal exchange is also possible [27].

3.1.3 Simultaneous Multi-threading (SMT)

SMT is a technique that permits multiple threads issues to multiple functional units in each cycle. SMT processor design supports instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle. Instruction-level parallelism comes from every single program or thread, and thread-level parallelism comes from either a multi-threaded parallel program or every single program from multi-programming workloads. Because it supports both types of parallelism, it utilizes resources on processors more efficiently to gain instruction throughput or speed up the processors. Hardware features of SMT are a combination of super-scalars and multi-threaded processors, like the ability to issue multiple instructions each cycle is inherited from super scalar and represents the hardware state for several threads from multi-threaded processors. SMT processors perform better than several processor designs such as super-scalar, traditional multi-threaded, and on-chip multiprocessor architectures [28].

3.1.4 Distributed Memory Architecture

Each processor has its local memory in a distributed memory architecture called a multi-processor computer system. This system has several benefits; first, there is no contention on the bus or switches. Each processor may use the bandwidth available to its local memory without interference from other processors. Second, as there is no bus, so no limit on the number of processors; the size of the system depends on the network used to connect processors. Third, there are no cache coherency problems; each processor operates on its data.

The main disadvantage of this technology is the more difficult inter-processor communication. Generally, communication from one processor to another happens through exchanging messages which causes extra overhead and is time-consuming. Also, there is a chance to intercept by other processors.

3.1.5 Intel Xeon Phi

Co-processor Intel Xeon Phi [29] is built on the Intel Many-Integrated Core (MIC) architecture. It has up to 61 cores connected by a high-performance on-die bidirectional interconnect. Figure 3.2 illustrates the basic blocks of Intel Xeon Phi architecture. Each core supports four hardware contexts simultaneously and executes two instructions per clock cycle. Every core has an 8-way set associative 32 KB L1 data cache, 32 KB instruction cache, and 512 KB private L2 cache. The L2 cache is kept fully coherent with each other by the tag directory (TD). Each core has a vector processing unit (VPU) that executes mathematical functions and 16 single, 8 double floating-point operations per clock cycle. A bidirectional ring interconnects all components (cores, TD, memory controller, PCIe, etc.). Every memory address is mapped using a reversible one-to-one address hashing method in 64 tag directories and eight memory controllers [30].

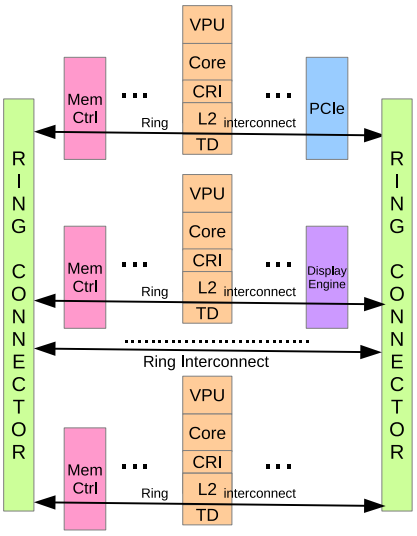

FIGURE 3.2: Intel Xeon Phi architecture

3.1.6 Single Instruction Multiple Thread (SIMT)

SIMT (Single Instruction, Multiple Thread) is a parallel programming model used in NVIDIA GPU in which multi-threading is performed by SIMD (Single Instruction, Multiple Data) processors. In SIMD, all cores execute the same instruction on different data streams; examples are vector computers, and most modern computers have SIMD architecture.

A critical difference between SIMD and SIMT is that SIMD exposes the width to the software where as SIMT instructions specify the execution and branching behavior of a single thread [31]. With the help of SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads and data parallel code for coordinated threads. There are three critical features of SIMT that SIMD does not have, (a) single instruction, multiple register sets, (b) single instruction, multiple addresses, and (c) single instruction, multiple flow paths.

However, despite high latencies, SIMT and SMT use thread mechanisms to achieve high throughput. SIMT is less flexible than SMT as (a) SIMT has low occupancy and flows divergence, which reduces the performance, and (b) SIMT synchronization options are minimal.

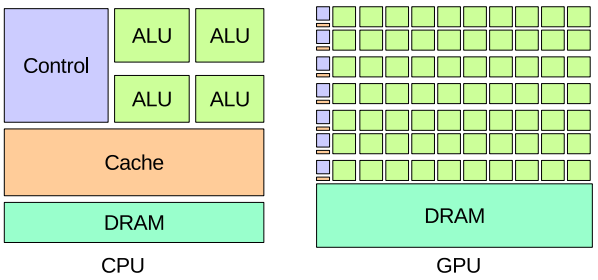

FIGURE 3.3: Approximate area of CPU and GPU

3.2 GPU Architecture

Graphics Processing Unit (GPU) is not a single stand-alone device but a co-processor to a CPU. GPU operates in conjunction with CPU through a PCI-Express bus. In GPU computing terms, CPU is called the host, and GPU is called the device, and the code running on CPU is called host code and the code running on GPU is called device code. A high-performance sequential part runs on the CPU, and a parallel part runs on GPU. GPU programming can run on compute unified device architecture (CUDA), a parallel computing platform and programming model invented by NVIDIA [31]. CUDA is an extension of C-like high-level programming language. There are multiple advances in GPU architecture over the years, some of the popular GPU architecture is Tesla (in 2006), Fermi (2010), Kepler (2012), Maxwell (2014), Volta (2017), Turing (2018), Ampere (2020), and Hopper (2022) developed by NVIDIA.

In Figure 3.3, the approximate CPU and GPU area is illustrated. In CPU number of arithmetic and logic units (ALU) is negligible. However, the ALU, control unit, and cache size are more prominent. But, in GPU, the number of ALU is vast, along with a smaller amount of area dedicated to controlling the cache.

CPU is latency oriented, powerful ALU that reduces the operation latency, fewer registers and SIMD units, and ample cash, which is most beneficial to converting long latency memory accesses to short latency cache accesses sophisticated control unit. It uses branch prediction to reduce branch latency and data forwarding to reduce data latency. Mainly CPU is used for sequential parts where latency matters, so it is more than ten times faster than GPU for sequential code.

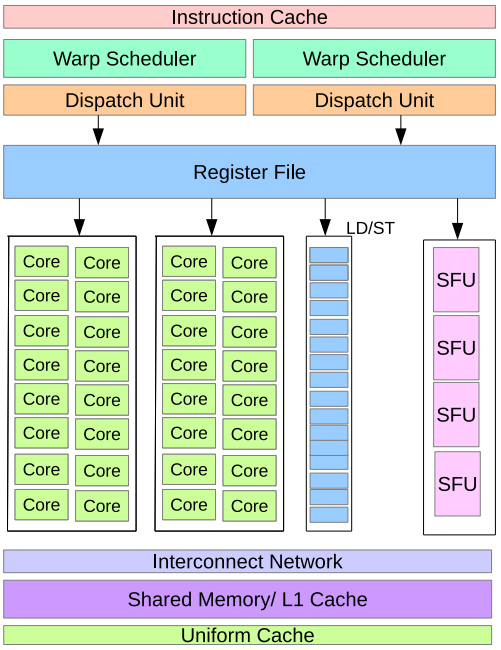

FIGURE 3.4: Streaming multiprocessor of NVIDIA fermi architecture

On the other hand, GPU has throughput-oriented cores, small caches, simple control but a large number of threads to manage, and no branch prediction and no data forwarding. Also, it has many energy-efficient ALU with long latency but is heavily pipe-lined for high throughput, many registers, and many SIMD units. Mainly GPU is used for parallel parts where throughput matters, so it can be more than ten times faster than CPU for parallel code.

Figure 3.4 shows GPU Fermi architecture's streaming multiprocessor (SM). Each GPU consists of many SM, and each SM has a certain number of streaming processors or processor cores. So it is possible to have thousands of threads executing concurrently on a single GPU. Each core executes a single thread instruction in a single instruction multiple data (SIMD) fashion, and the instruction unit distributes the current instruction to the cores. Each core has one arithmetic unit that can perform single-precision floating point operations or 32-bit integer arithmetic. In addition, each SM has special function units (SFUs), which can execute more complex floating point operations such as reciprocal sine, cosine, and square root with low cycle latency.

The SM has load/store units and other components like registers files, warp scheduler, and dispatch unit. Thus GPU architecture consists of SM, interconnect network, shared memory/L1 cache, uniform cache, and DRAM. Some of the GPU architecture released are: Fermi, kepler, Maxwell, Pascal, Volta, Turing, Ampere, and Hopper architecture.

Fermi Architecture: The Fermi architecture was the first complete GPU architecture in high-performance computing (HPC) applications. It has 512 accelerator cores which are also called CUDA cores. Fermi architecture has 16 SM each with 32 CUDA cores. Fermi has six 384-bit GDDR5 DRAM memory interfaces, which can support up to a total of 6GB of global memory. Also, Fermi includes a 768 KB L2 cache, which is shared by all SMs. Each SM has 16 load/store units, two warp schedulers, two instruction dispatch units, and 4 special function units (SFUs). Fermi also supports concurrent kernel execution, multiple kernels launched from the same application executing on the same GPU simultaneously.

Kepler Architecture: The Kepler architecture, released in 2012, is a fast and highly efficient high-performance computing architecture. Kepler K20X has 15 SMs and six 64-bit memory controllers. Each SM consists of 192 single-precision CUDA cores, 64 double-precision units, 32 special function unit and 32 load/store units (LD/ST). The three important features in the Kepler architecture has: first, enhanced SMs; second, dynamic parallelism that allows the GPU to launch new grids; and third, Hyper-Q: It adds more simultaneous hardware connections between the CPU and GPU so that CPU cores simultaneously run more tasks on the GPU.

Maxwell Architecture: The Maxwell architecture, released in 2014, was the successor to Fermi. The first generation of Maxwell GPUs has the following advantages over Fermi:

- It has enhanced streaming multiprocessor (SM), related to control logic partitioning, workload balancing, clock-gating granularity, compiler-based scheduling, number of instructions issued per clock cycle, and many other enhancements which improve energy efficiency.

- It has more significant dedicated shared memory, i.e., 64 KB per SM, unlike Fermi and Kepler, which partitioned 64 KB into shared memory and L1 cache. It remains the per-thread block memory as 48 KB, but the total shared memory increased, which improves the occupancy of SM. Dedicated shared memory improvements are possible by combining the L1 cache and texture memory.

- It has native shared memory atomic operations, which helps implement other atomic operations. While Fermi or Kepler has a lock/update/unlock pattern, which could be expensive for updates to a particular location in shared memory.

- It has dynamic parallelism support in the mainstream that benefits developers to no need to implement special-case algorithms for high-end GPUs.

Pascal Architecture: It has a successor to Maxwell, released in 2016. It has the following improvements over Maxwell architecture:

- It supports NVLink communication which enables a 5X increase in bandwidth between Tesla Pascal GPUs and CPU, which results in significant speed improvement over PCIe.

- It has high bandwidth HBM2 memory, which provides 3X memory performance over Maxwell and Kepler GPUs.

- It has pascal unified memory Which helps developers not copy data from the host to the device and vice-versa. It allows direct access to GPU all memory as well as all system memory.

- It has computed preemption support, allowing higher priority tasks to interrupt currently running tasks.

- It supports dynamic load balancing, which optimizes GPU resource utilization.

Volta Architecture: It was released in 2017, strictly marketed for professional applications, with advanced technologies supporting HPC and artificial intelligence applications. It has the following improvements over Pascal architecture:

- It was the first micro-architecture to use tensor cores (which is mainly for specialized mathematical calculations). Tensor cores perform matrix operations for deep learning, and AI use cases.
- It has new SM architecture optimized for deep learning. Its SM is 50% more energy efficient than previous Pascal architecture.
- It has a second-generation NVIDIA NVLink that gives high-speed interconnect, higher bandwidth, more links, and improved scalability for use in multi-GPU and multi-GPU/CPU systems.
- It has enhanced unified memory that allows more accurate migrations of pages to the processor and access more frequently. It has also enhanced address translation service that allows GPU to access CPU's page table directly.

Turing Architecture: It was released in 2018 with the support of tensor cores and consumer-focused GPUs. It has the following improvements over Volta architecture:

- It has new SM architecture, also called Turing SM, that improves the shading efficiency and 50% improvements in performance per Cuda-core compared to pascal generations.
- It has tensor cores similar to volta architecture with an enhanced feature in precision modes.
- It introduces real-time ray tracing (RTX), which enables a GPU to visualize 3D games into physically accurate shadows, reflection, and refraction, which has applications in virtual reality (VR).

- It has deep learning features for graphics as well as for inference. It utilizes deep learning neural networks and a set of neural services to perform AI-based functions that accelerate and enhance graphics applications.

Ampere Architecture: It was released in 2020, with enhancement from Turing architecture in 3rd generation NVlink, 3rd generation tensor cores, and 2nd generation ray tracing cores. It provides tremendous speedup for AI training and inference applications, HPC workloads, and data analytics.

Hopper Architecture: It is released in 2022, with enhancement from Ampere architecture in 4th generation NVlink, and 4th generation tensor cores to perform faster matrix computations. It supports distributed shared memory that allows direct SM-to-SM communications for loads, stores, and atomics across multiple SM shared memory blocks.

3.3 CUDA Programming Model

The CUDA programming model provides an abstraction for GPU architecture, which helps to link the application program to its implementation on GPU hardware. In this model, the CPU is the host, while the GPU is the device. For executing any CUDA program, there are mainly three steps involved: first, copy the input data from host memory to device memory *i.e.* host to device data transfer; second, load GPU program and execute, and third, copy the results back from device memory to host memory. CUDA kernel is a function that executes on GPU. For running any application on GPU, a parallel portion is executed k times by k threads in parallel while running on the CPU; it performs only once like a regular C function. For declaring a CUDA kernel function *__global__* declaration specifier is used in the beginning of the function name.

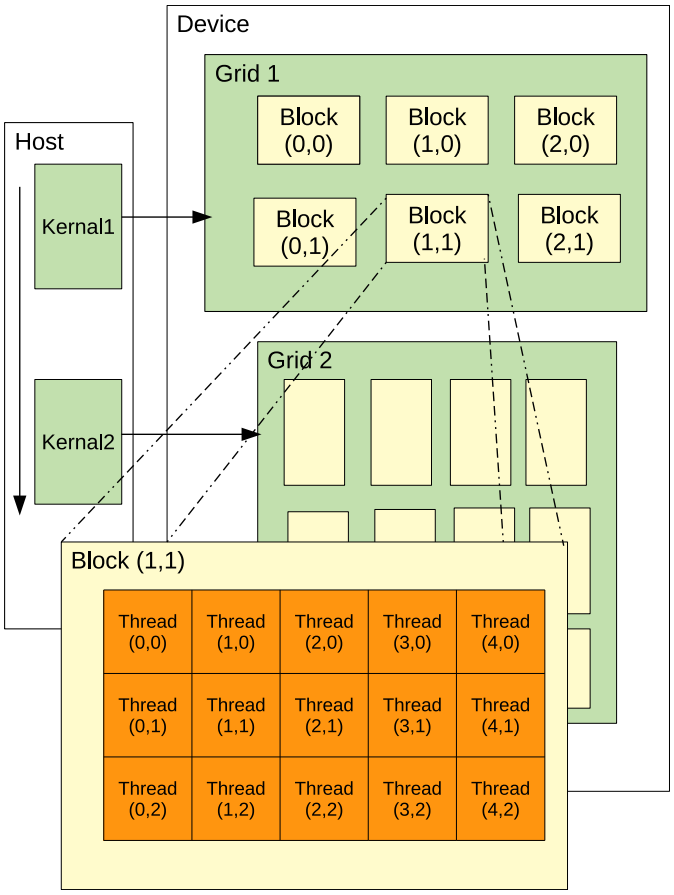

FIGURE 3.5: CUDA programming model

3.3.1 GPU Thread Mapping and Scheduling

A group of threads is called a CUDA block, and a group of blocks is called a grid. Each thread is mapped to one Cuda core, and a cuda block is mapped to the streaming multiprocessor. A streaming multiprocessor can run multiple cuda or thread blocks. All threads in one grid execute the same kernel. Figure 3.5 illustrates the cuda programming model. Each thread and block has bult-in 3D variable index called as threadIdx and blockIdx.

3.3.2 GPU Memory Hierarchy

GPU consists of various types of memory, which is illustrated in Figure 3.6 and described as follows:

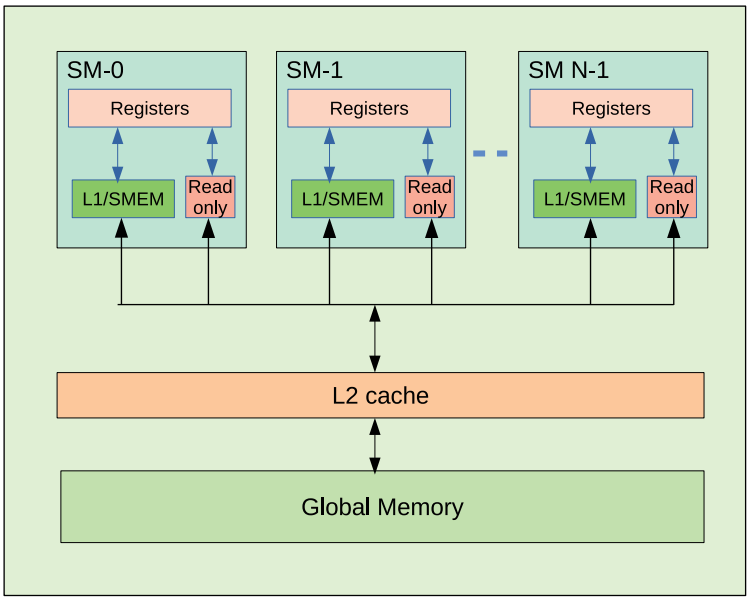

FIGURE 3.6: Memory hierarchy in GPU

Registers: Registers are private to each thread assigned to the thread, which is not visible to other threads. The compiler decides on the registers utilization.

L1/Shared Memory (SMEM): Every SM has an on-chip memory that is very fast and uses an L1 cache and shared memory. All threads in a CUDA block can share the shared memory, and all CUDA blocks can share physical memory resources in an SM.

Read-only Memory: Each SM has a read-only memory to kernel code: an instruction cache, constant memory, texture memory, and RO cache.

L2 Cache: L2 cache is shared across all SMs, so every thread in every block can share this memory.

Global Memory: Global memory is DRAM which is the frame buffer size of GPU.

3.4 Summary

Multi and many-core architecture is improving their performance with release in different versions. In GPU architecture, NVIDIA has released different versions with improvements from the previous version. Recent versions of GPU are

used for deep learning, artificial intelligence, and virtual reality applications. This thesis used Maxwell and Pascal GPU architecture with different CUDA cores. The CUDA programming model helps the GPU architecture to be utilized efficiently.

Chapter 4

Analysis of Meta-heuristics for Quadratic Assignment Problem in Accelerated Systems

In this chapter, we discuss the first contribution of the QAP and parallelization of meta-heuristics: ILS, SA, GA, PSO, CSA, and TS in accelerated systems: multi-core CPU, Pthread, and many-core GPU for QAP. In addition to this, We also discussed the performance and statistical analysis of above all meta-heuristics and compared them among these meta-heuristics. We also calculated the speedup on Pthread and GPU concerning CPU.

4.1 Introduction

The quadratic assignment problem (QAP) is one of the most studied optimization problems, as it is NP-hard and considered a classical challenge [32]. Owing to its complexity and importance in real-life problems, it has attracted considerable attention from many researchers, with many real-life problems having direct applications of the QAP such as the facility layout problem [33], the traveling salesman problem [4], the bin packing problem [34], the maximum clique problem [35], the

scheduling problem [36], the graph partitioning problem [37], the keyboard layout problem [38], the problem of memory layout in digital signal processors [39], and the backboard wiring problem [40].

Many exact and heuristic methods can solve the QAP problem [5]; however, heuristics take a considerable amount of time for instances of size greater than 20. Compared to heuristic methods, which are problem-specific, meta-heuristics are suitable for all types of problems. Thus, they have become one of the alternative methods for solving the QAP, with many nature-inspired meta-heuristics being proposed for this purpose. In this study, we have chosen some single solution-based meta-heuristics and some population-based meta-heuristics to find the optimal solution for QAP. In addition, we have considered 21 test instances from the QAP library (QAPLIB) [6] as a benchmark for QAP, and generated task graphs for the meta-heuristics using the *Contech* tool [7] for parallel programs and analyzed their performance.

The rest of the chapter is organized as follows: literature review for QAP is discussed in section 4.2, and motivation and background are presented in section 4.3. The QAP is defined in section 4.4. Section 4.5 describes the accelerated system we used in this study. A description of the problem of mapping to architecture is given in section 4.6. Section 4.7 presents the experimental results, while Section 4.8 describes the performance analysis of the meta-heuristics. Section 4.9 presents the generation of task graphs using *contech tools*. Finally, we summarize this chapter in the section 4.10.

4.2 Literature Review

Many works [41, 42] have been done related to solving optimization problems using meta-heuristics in accelerated systems. Due to the increasing interest in writing programming languages on GPUs, solving combinatorial optimization problems has become popular. An evolutionary algorithm (EA), a population-based meta-heuristic algorithm, was the first parallel algorithm developed on GPU. Wong et

al. [43] proposed the first GA on GPU, as they implemented fast evolutionary programming based on mutation operators. They divided their work into three phases: first, they generated the initial population in the central processing unit (CPU) and then evaluated it in GPU; second, they compared the results for the CPU or GPU; third, they transferred the final results from GPU to CPU. However, they implemented the replacement and selection operators on the CPU, thus limiting the algorithm's performance owing to the transfer of essential data from CPU to GPU. This drawback has been overcome by Q. Yu et al. [44] who were the first to implement a full parallelization of a G.A. on GPU. They represented the population structure with a 2D toroidal grid in which each grid point denotes one individual, using a crossover operator to generate a child for each individual, selecting the best one among the neighborhood by taking one of its parents and itself. However, this implementation has not worked for the binary encoding scheme of G.A., as it only works for vectors of real values. To overcome this problem, J.M. Li et al. [45] have used specific genetic operators for the binary representation of G.A.

In his study [46], Zhu implemented a parallel evolution strategy algorithm based on a pattern search on GPU. He used multiple kernels on GPU with the selection and crossover operators running on one kernel and the mutation and evaluation operators running on the other kernel of GPU. At the same time, the remaining processes were done on the CPU. He compared the performance of CPU and GPU by considering a selected benchmark and analyzing the quality of solutions under time limits.

Tsutsui et al. [47] used the shared memory management concept on GPU for solving the QAP by GA. In their work, they designed a GA model divided into two types of independent GAs, where each sub-population represents one thread block. While the GPU implementation aims to maximize the utilization of shared memory, the maximum number of threads runs on multiprocessors, thus coalescing the memory access. They compared the results for CPU and GPU, indicating that there is still room for performance improvement because they shuffled the individual on CPU, which is time-consuming, and used a multiple-population

coarse-grained GA and not a single-population fine-grained one. Moreover, they did not use the texture memory, a high-speed memory, and did not apply any local search. Luong et al. [48] proposed a hybrid EA by mixing EA with local search on GPU. They divided their model into three layers: high level (memory allocation and data transfer), intermediate level (thread management), and low level (memory management).

- *High-level layer:* In this layer, both the allocation of memory on GPU and data transfer between CPU and GPU are done.
- *Intermediate-level layer:* This layer focuses on efficient thread mapping to the neighborhood, as the GPU threads are light-weighted compared to the CPU threads; thus, the context switch between the two threads is very slow. Therefore, thread management is necessary for the generation of the neighborhood in local search.
- *Low-level layer:* In this layer, kernel memory management is performed. Additionally, the authors manage the memory coalescing issues of unstructured data by bounding the texture memory with the global memory, which is a read-only memory.

Later, they tested the proposed algorithm on QAP by taking standard *taillard* instances of size greater than 30, noted the CPU time and GPU time as well as the texture memory, and analyzed the performance, showing that GPU is much more efficient than CPU. Many meta-heuristics have been used to solve QAP using GPU [49]; in this paper, the authors survey different works related to QAP using meta-heuristics on GPU. A cooperative parallel tabu search algorithm is proposed by James et al., for solving QAP [50]; in this paper, the authors compared variants of tabu search with the proposed algorithm. In paper [51] E. Sonuc et al., parallel multi-start SA for QAP on GPU is proposed and compared with TS, and the authors indicated the GPU run times are 29 times faster than CPU. A parallel genetic algorithm is proposed for QAP in [52] by H. Alfaif et al., in which authors used new crossover and mutation are implemented on CPU and GPU, that shows

the optimal solutions for a few instances. In [53], R. Matousek et al. proposed a mathematical programming technique for finding the lower bounds for QAP that helps to construct the starting point for QAP. Silva et al., [54] considered four variants of QAP and proposed a single framework to solve all of them; authors also illustrated parallel memetic iterated tabu search as the most successful heuristics for QAP. In [55], R. POVEDA et al. proposed a hybrid parallel meta-heuristics algorithm for QAP. A parallel ILS meta-heuristic is implemented for QAP in paper [56] by Özçetin et al. and calculated the speedup on GPU. Cheng et al. [57], exposed a survey work on parallel GA on GPU architecture.

In paper [58], L. Stoltzfus et al. used to optimize the placement of data on GPU memory architecture. The authors also compared the speedup of matrix multiplication on different GPU architectures. A survey on GPU parallelization strategies is discussed by M. Essaid et al. [59]. Here authors listed the different implementations of parallel meta-heuristics using GPU programming.

4.3 Motivation and Background

Many meta-heuristics are used for solving QAP [60]. Although, QAP belongs to the NP-complete class [61] problem, using meta-heuristics on GPU [62], effective near optimal solution on GPU can be achieved in reasonable time.

4.4 Quadratic Assignment Problem (QAP)

The main objective of the QAP is to minimize the assignment cost, that is, the cost to assign n facilities to n locations. The assignment cost is calculated as the sum of the cost of all pairs, that is, the product of the flows between facilities and the distance between nodes corresponding to these facilities.

Assuming $D = (d_{ij})$ and $F = (f_{ij})$ to be two positive integer matrices of size $n \times n$, then the permutation of $(1, 2, \ldots, n)$ is the solution of QAP that minimizes the

objective function

$$Z(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} f_{\pi(i)\pi(j)} \tag{4.1}$$

where π is any permutation of $(1, 2, \ldots, n)$ representing a solution of QAP, which is also called the permutation representation. The functional value of the objective function Z depends on both the distance and flow matrices, thus making the QAP problem more difficult to solve, with each permuted solution representing a neighbor solution. In this study, we use the pairwise exchange method to generate a neighbor solution.

4.4.1 Solution Evaluation for QAP

To evaluate the solution, the evaluation function takes a substantial amount of time. In this study, in the evaluation function, we pass one full solution at a time. Because the assignment cost is the sum of the cost of all pairs of locations, to calculate the cost for one location, it searches all facilities taking $O(n)$ time, while taking $O(n^2)$ times to calculate the cost for all locations.

4.4.2 Incremental Solution Evaluation for QAP

In the incremental solution evaluation, instead of passing one full solution at a time and calculating the cost for every location, we only pass the changed locations of neighbor solutions, with no changes to the remaining locations. Hence, instead of calculating the cost for all locations, we only calculate the changed cost for that solution; thus, it takes only $O(n)$ times.

Pseudo code of incremental solution evaluation [63]:

```
int evaluateSwapCost(int N, int *t_sol, int Cost,
int r, int s, int **d_m, int **f_m)
{
  int vr = t_sol[r]-1,vs = t_sol[s]-1,k,delta = 0;
```

```
  delta = d_m[r][r] * (f_m[vs][vs]-f_m[vr][vr]) +
          d_m[r][s] * (f_m[vs][vr]-f_m[vr][vs]) +
          d_m[s][r] * (f_m[vr][vs]-f_m[vs][vr]) +
          d_m[s][s] * (f_m[vr][vr]-f_m[vs][vs]);
  for(k=0; k<N; k++)
  {
    int vk = t_sol[k]-1;
    if(k != r && k != s)
     {
       delta +=(d_m[k][r] * (f_m[vk][vs]-f_m[vk][vr]) +
                d_m[k][s] * (f_m[vk][vr]-f_m[vk][vs]) +
                d_m[r][k] * (f_m[vs][vk]-f_m[vr][vk]) +
                d_m[s][k] * (f_m[vr][vk]-f_m[vs][vk]));
      }
   }
 Cost+=delta;
 return Cost;
}
```

4.5 The Accelerated System for QAP

In a multi-core processor, two or more cores are built on a single computing platform, with all cores running in parallel to enhance the overall speed of the program. Each core is treated as a separate processor by the operating system and maps to *threads/processor* by OS scheduler, and each processor is physically connected to the same memory. Multi-core processors execute multiple instructions and process multiple data on different cores at a time. In this study, we have used Intel(R) Core(TM) i5-6500 CPU which has 3.2 GHZ clock speed and 4 CPU cores.

At present, the (GPU)-based system is a popular parallel processing system. GPU is not a stand-alone single device, but it is connected to CPU through a peripheral

component interconnect express (PCI-e) bus. NVIDIA's Compute unified device architecture (CUDA) is a combination of several streaming multiprocessors (SMs). Each SM consists of several scalar processors or CUDA cores that run in parallel, has a load/store unit, a special functional unit (SFU), and shared memory and caches (constant and texture caches), and shares a global memory of GPU. When a kernel is launched by GPU, the threads of that kernel are distributed among the SMs for execution.

In CUDA programming, a program runs on both CPU and GPU, with the task being divided between the CPU host and GPU device. A program that runs on CPU (the host code) calls the program that runs on GPU (the device code), which is also called the kernel. As GPU threads are light weight, thread switching is a low-cost operation. A CUDA core runs a single instruction on multiple threads. A group of 32 threads is called warps. In a warp, all the threads run the same instruction simultaneously. In this study, we used Nvidia GeForce GTX 980 Ti GPU with 2816 CUDA cores and 6 GB DDR5 memory, 22 SM, and 64 warps per SM. Additionally, We installed the *CUDA toolkit* 8.0 on this device.

4.6 Mapping Meta-heuristics for QAP to Multi-core, Pthread, and GPU

The general approach for solving the QAP for all meta-heuristics is as follows. First, we take the input as distance matrix and flow matrix. Then, it generates a random initial solution of QAP, from which we obtain all possible neighbor solutions and assess them using the evaluation function. If a neighbor solution improves the objective of the current solution, then it replaces the current solution as the new solution. This process will continue until reaching the stopping condition.

We have obtained the input instances from QAPLIB [6], with all the instances of QAPLIB being available online on the QAPLIB website. We have taken instances

of size between 30 and 100 of *taillard a*, 60 and 150 of *taillard b*, 64 and 256 of *taillard c*, 72 and 100a from *sko* and *els*19, *kra*30a, *kra*32, and *ste*36a of other set as benchmarks for the QAP problem.

4.6.1 ILS Implementation

4.6.1.1 ILS on Serial Machine

Using ILS, we generated neighbor solutions from the initial solution with the adjacent pairwise exchange permutation method. We started from the first position, and then changed the position of the elements to the next nearby position. After the new neighbor solution was generated, we assessed it using the evaluation function. We compared the assignment cost with the current assignment cost; if it was less than the current assignment cost, we chose the new solution as the current solution. Thereby, we evaluated all the $\frac{n\times(n-1)}{2}$ neighbor solutions. At this stage, local optima were reached; thus, for the next iteration, we made the local minimal solution the initial solution and repeated all the aforementioned steps. This process continued until the stopping criteria were reached. To avoid being stuck in local optima or not being able to get the best global minimum solution, we took 500 random initial solutions, with each initial solution being iterated 10 times. Finally, we observed the objective value (obtained cost) and the execution time (in seconds) to execute the whole program as in Table 4.1. In this table, the percentage deviation (D) is calculated using Equation 4.2. The deviation of the benchmarks on CPU increases slowly as the size of *taillard a, b* and *c* instances increases from 30 to 100, 60 to 150, and 64 to 256 respectively; however, the execution time of the benchmarks increases more rapidly with the size. For instances of class 3, *sko* 90 show less deviation and more execution time than *sko* 81. For class 4 instances *els*19, *kra*30a, *kra*32, and *ste*36a the deviation and the execution time increases as the size of the instances increases from 19 to 36. In Table 4.1, *speedup* (S) is calculated using $\frac{CPU\,exec.\,time}{Pthread\,exec.\,time}$ for Pthread and $\frac{CPU\,exec.\,time}{GPU\,exec.\,time}$ for

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
unstructured, randomly generated instances, class 1								
tai30a	4.87	0.871	4.87	0.240	3.63	4.87	0.192	4.54
tai35a	5.78	1.397	5.78	0.377	3.71	5.78	0.222	6.29
tai40a	7.13	2.058	7.13	0.711	2.89	7.13	0.243	8.47
tai50a	8.04	4.003	8.04	1.081	3.70	8.04	0.286	14.00
tai60a	8.47	7.099	8.47	1.887	3.76	8.47	0.331	21.45
tai80a	8.43	16.950	8.43	5.958	2.84	8.43	0.436	38.88
tai100a	8.87	34.224	8.87	10.431	3.28	8.87	0.618	55.38
Real-life like instances, class 2								
tai60b	14.55	8.023	14.55	2.104	3.81	14.55	0.311	25.80
tai80b	17.29	19.037	17.29	4.967	3.83	17.29	0.419	45.43
tai100b	17.47	37.451	17.47	9.930	3.77	17.47	0.598	62.63
tai150b	17.64	126.210	17.64	33.723	3.74	17.64	1.514	83.36
tai64c	0.00	9.734	0.00	2.539	3.83	0.00	0.334	29.14
tai256c	**4.91**	**769.284**	**4.91**	**206.234**	**3.73**	**4.91**	**7.153**	**107.55**
Instances with grid-distances, class 3								
sko72	8.83	14.397	8.83	3.622	3.97	8.83	0.367	39.23
sko81	9.09	20.491	9.09	5.167	3.97	9.09	0.425	48.21
sko90	9.03	28.188	9.03	7.081	3.98	9.03	0.493	57.18
sko100a	9.45	38.673	9.45	9.730	3.97	9.45	0.598	64.67
Real-life instances, class 4								
els19	0.61	0.242	0.61	0.069	3.51	0.61	0.151	1.60
kra30a	9.20	1.003	9.20	0.266	3.77	9.20	0.190	5.28
kra32	9.41	1.219	9.41	0.322	3.79	9.41	0.200	6.10
ste36a	21.58	1.738	21.58	0.457	3.80	21.58	0.211	8.24

TABLE 4.1: Percentage deviation and exec. time of QAP using ILS on CPU, Pthread, and GPU

GPU.

$$Deviation\% = \frac{(Objective\,Value - QAPLIB\,Cost)}{QAPLIB\,Cost} \times 100. \tag{4.2}$$

4.6.1.2 ILS using Pthread

In this meta-heuristic, instead of executing all initial solutions at a time, we assigned all the initial solutions to a number of processors so that each processor could be utilized and get an equal number of the initial solutions. Then, each part was assigned to different processors so that all parts could be run in parallel.

Finally, Table 4.1 shows the optimal cost and execution time, with the percentage deviation of all instances running on CPU and Pthread being the same; however, the execution time on Pthread was reduced in proportion to the number of processors used.

4.6.1.3 ILS on GPU

Implementing meta-heuristics on a massively parallel architecture is not a straightforward task, as it requires considerable effort at the design and implementation stages. In general, there are three issues in GPU where optimization is possible: 1) efficient communication between CPU and GPU, in which the main issue is to optimize data transfer; 2) control of parallelization, in which the issues are thread generation and mapping the threads to data input; and 3) efficient memory management, which can be performed on any memory.

The parallel design of meta-heuristics to solve any combinatorial optimization problem, such as the QAP, has a significant performance effect on accelerated machines.

First, we set the grid size and block size of a kernel to generate a number of threads to achieve the best performance with respect to executing the ILS algorithm on GPU, which is written in Algorithm 6. For the implementation, we generated random initial solutions on CPU, and from each initial solution we generated neighbor solutions that were evaluated on GPU. In this study, we took 500 initial solutions, with each initial solution being iterated 10 times to get the best possible optimal solution. All the initial solutions were run in parallel and called the GPU, which also ran in parallel to execute the evaluation cost function of neighbor solutions. The percentage deviation and execution time on GPU are reported in Table 4.1, which demonstrates that the execution time on GPU is less than that on CPU. For example, the Pthread for *tai100a* instances takes 34.224 seconds on CPU, while on GPU, it takes only 0.618 seconds, demonstrating that GPU executes 55 times faster than CPU.

Algorithm 6 Iterated Local Search (ILS) On GPU

Input: Distance matrix and Flow matrix
Output: Optimal solution (local optima).

1: Generate a random initial solution
2: Evaluate the objective function of the solution
3: Initialize the ILS parameters
4: **while** not Termination_Criteria **do**
5: **repeat**
6: s_b the best neighbor solution near s, and $s_b = s$
7: Generate all neighbor solutions of s
8: Evaluate all the neighbor solutions of s using the objective function F by GPU kernel in parallel
9: Results are sent back from GPU to CPU
10: **if** $F(s')$ is less than $F(s_b)$ **then**
11: $s_b = s'$
12: **end if**
13: **until** All the neighbor solutions of s are explored
14: for next iteration $s = s_b$
15: **end while**

4.6.2 SA Implementation

4.6.2.1 SA on Serial Machine

Using SA to execute the evaluation function, first, we generated all possible neighbor solutions. Unlike executing the ILS for each neighbor solution to find the best solution, we calculated the assignment cost of all the neighbor solutions at a time and then stored the values for each solution. To find the best solution among all neighbor solutions, we applied SA. We fixed the main parameters of SA to be as follows: initial temperature: $10,000$, cooling rate: 0.9999, and absolute temperature: 0.00001. Initially, the algorithm starts with an initial temperature, and at every iteration, the temperature is reduced to [current temperature×cooling rate] making it the current temperature for the next iteration. This process continues until the current temperature reaches the absolute temperature. The deviation and execution time are reported in Table 4.2, where we can see that as the size of instances increases from 80 to 100, the execution time also increases more rapidly.

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
unstructured, randomly generated instances, class 1								
tai30a	6.13	0.964	8.06	0.350	2.75	7.33	0.756	1.28
tai35a	8.16	1.568	8.92	0.507	3.09	7.14	0.825	1.90
tai40a	7.69	2.217	8.84	0.666	3.33	8.18	0.857	2.59
tai50a	9.17	4.286	9.60	1.551	2.76	8.98	0.973	4.40
tai60a	9.21	7.469	10.56	2.317	3.22	9.56	1.106	6.75
tai80a	9.18	17.825	9.26	5.235	3.40	9.30	1.458	12.23
tai100a	9.39	35.344	9.22	10.101	3.50	9.22	1.924	18.37
Real-life like instances, class 2								
tai60b	13.36	8.511	13.60	2.394	3.56	14.06	1.050	8.11
tai80b	18.65	20.025	18.38	5.431	3.69	18.33	1.314	15.24
tai100b	18.70	39.161	18.95	10.470	3.74	19.12	1.689	23.19
tai150b	18.21	132.719	16.89	35.699	3.72	17.58	4.170	31.83
tai64c	0.09	10.372	0.19	2.897	3.58	0.00	1.145	9.06
tai256c	**4.78**	**769.235**	**4.52**	**210.690**	**3.65**	**4.66**	**12.613**	**60.99**
Instances with grid-distances, class 3								
sko72	16.16	14.477	16.57	3.759	3.85	16.47	1.209	11.97
sko81	16.22	20.584	16.36	5.268	3.91	15.63	1.341	15.35
sko90	15.60	28.269	15.01	7.216	3.92	15.06	1.490	18.97
sko100a	14.40	38.652	14.01	9.878	3.91	14.58	1.696	22.79
Real-life instances, class 4								
els19	0.52	0.274	0.89	0.176	1.56	0.00	0.338	0.81
kra30a	35.61	1.082	34.18	0.328	3.30	33.41	0.407	2.66
kra32	38.00	1.305	30.78	0.381	3.43	35.33	0.404	3.23
ste36a	74.76	1.841	86.67	0.524	3.51	88.37	0.480	3.84

TABLE 4.2: Percentage deviation and exec. time of QAP using SA on CPU, Pthread, and GPU

4.6.2.2 SA using Pthread

In this study, we applied SA to find the best solution or local optima for every initial solution. Therefore, instead of running all of the initial solutions sequentially on CPU, we assigned them to an equal number of processors. On each processor, the optimal solution was found and compared to get the globally optimal solution. The percentage deviation and total execution time are reported in Table 4.2. In Pthread implementation, processor utilization performs better as compared to serial machine, with each part running in parallel. In Table 4.2, we can see that its percentage deviation is not the same as that of CPU because of the random

generation of solutions. Additionally, the execution time on Pthread is reduced almost in proportion to the number of processors; however, not because of the overhead of Pthread.

4.6.2.3 SA on GPU

For the implementation of SA on GPU, first, we generated neighbor solutions from the initial solutions and evaluated them on GPU. Then, the whole cost of the neighbor solutions was transferred from GPU to CPU. Afterwards, we performed SA on CPU to find the optimal solution after all neighbor solutions were evaluated in parallel by GPU. The modified algorithm of SA is written in Algorithm 7, and the execution results are reported in Table 4.2. In this table, CPU takes 35.344 seconds for *tai100a* execution, while GPU takes only 1.924 seconds. Moreover, we can observe that for the large numbers of instances, GPU performs faster than CPU, as compared to the small numbers of instances.

Algorithm 7 Simulated Annealing (SA) on GPU

Input: Cooling schedule
Output: Optimal solution.

1: Generate a random initial solution
2: Evaluate the objective function of the solution
3: Set the initial parameters for SA
4: **while** not Termination_Criteria *i.e.* $T < T_{min}$ **do**
5: **repeat**
6: Generate random neighbor s' from s (where $s' \in N(s)$)
7: Evaluate the neighbor solution s' using the objective function f on GPU in parallel.
8: Results are sent back from GPU to CPU.
9: $\Delta C = f(s') - f(s)$; /* Difference in the assignment cost */
10: **if** $\Delta C \leq 0$ **then**
11: $s = s'$ /* Accept the neighbor solution */
12: **else**
13: Accept the neighbor solution s' with a probability $e^{\frac{-\Delta C}{T}}$
14: **end if**
15: **until** Absolute Temperature (Equilibrium condition) is reached
16: $T = g(T)$; /* Update the $temperature = temperature \times coolingRate$ */
17: **end while**

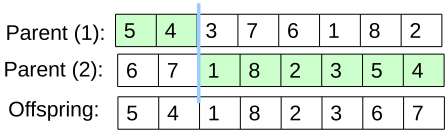

FIGURE 4.1: Crossover operator

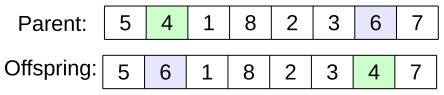

FIGURE 4.2: Mutation operator

4.6.3 GA Implementation

4.6.3.1 GA on Serial Machine

GA starts with a random initial population of solutions, where the neighbor solutions are generated with the help of two operators, namely, crossover and mutation. The crossover operator selects two random solutions from the population of solutions. There are several methods for the random selection of individuals in this algorithm, of which the roulette wheel and tournament methods are the most popular. In this study, we used the tournament method. In this method, first, a few solutions are randomly selected. Then, among them, the best solution (which has the least assignment cost) is selected.

When two solutions are selected, the crossover operator is employed to generate the new offspring as shown in Figure 4.1, and then to store it in the new population. In this study, we used one-point crossover, and then applied the mutation operator to the newly generated solution. For mutation, we simply changed the position of the elements as illustrated in Figure 4.2, and stored it in the new population. In addition, we fixed the number of initial solutions as 5000 and the number of iterations as 10. From the 5000 initial solutions, we generated 25000 neighbor solutions using the crossover and mutation operators. In each iteration, the best population of solutions was selected (5000), becoming the current population, with this process continuing until the termination condition is reached. The percentage deviation and execution time of GA are reported in Table 4.3, which shows that as the instance size increases from 30 to 60, the execution time slightly increases, but as the size increases from 60 to 80 and 100, the execution time increases more rapidly.

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
unstructured, randomly generated instances, class 1								
tai30a	13.39	1.754	11.19	0.622	2.82	13.39	0.443	3.96
tai35a	15.49	2.224	13.84	0.759	2.93	15.49	0.490	4.54
tai40a	14.70	2.771	14.70	0.916	3.03	14.70	0.519	5.34
tai50a	15.22	3.688	14.72	1.147	3.22	15.22	0.599	6.16
tai60a	14.72	4.960	11.67	1.534	3.23	14.72	0.705	7.04
tai80a	12.80	8.192	12.80	2.419	3.39	12.80	0.865	9.47
tai100a	12.32	12.124	11.49	3.428	3.54	12.32	1.064	11.39
Real-life like instances, class 2								
tai60b	45.18	4.985	40.28	1.524	3.27	45.18	0.691	7.21
tai80b	38.61	8.272	37.14	2.417	3.42	38.61	0.859	9.63
tai100b	38.50	12.065	38.50	3.454	3.49	38.50	1.071	11.27
tai150b	25.99	24.632	25.99	6.870	3.59	25.99	1.720	14.32
tai64c	13.77	5.510	13.77	1.703	3.24	13.77	0.720	7.65
tai256c	**12.33**	**66.926**	**12.33**	**18.241**	**3.67**	**12.33**	**3.041**	**22.00**
Instances with grid-distances, class 3								
sko72	16.01	6.732	16.01	2.054	3.28	16.01	0.790	8.52
sko81	14.71	8.475	14.45	2.503	3.39	14.71	0.895	9.47
sko90	15.29	9.812	14.48	2.890	3.40	15.29	0.988	9.93
sko100a	14.14	12.090	12.76	3.515	3.44	14.14	1.070	11.30
Real-life instances, class 4								
els19	64.19	1.097	0.93	0.446	2.46	64.19	0.382	2.87
kra30a	31.70	1.769	24.80	0.648	2.73	31.70	0.441	4.01
kra32	34.90	1.966	34.90	0.697	2.82	34.90	0.459	4.28
ste36a	86.77	2.289	48.62	0.802	2.85	86.77	0.486	4.71

TABLE 4.3: Percentage deviation and exec. time of QAP using GA on CPU, Pthread, and GPU

4.6.3.2 GA using Pthread

In GA, the generated random initial solutions are called the population, with the neighbor solutions being generated with the help of genetic operators, namely, crossover and mutation. Afterwards, the neighbor solutions are evaluated in parallel, among them, the best population is chosen for the next generation. In this study, the neighbor solutions are assigned to an equal number of processors, with the number of neighbor solutions taken being 25000 and the initial population being 5000. Each processor evaluates the solutions and store their costs, and then merges all solutions, out of which we chose the best 5000 solutions for the next

Algorithm 8 Genetic Algorithm (GA) on GPU

Input: Distance matrix and Flow matrix
Output: Optimal solution.

```
1: Generate random initial population P
2: Set the initial parameters for GA
3: while not Termination_Criteria do
4:     repeat
5:         Generate offspring P' with crossover operation and from P
6:         Generate offspring by applying mutation operation on P'
7:         Copy offspring from CPU to GPU
8:         Evaluate the offspring P' using the objective function F on GPU in
   Parallel.
9:         Results are sent back from GPU to CPU
10:        Find the best population P_b
11:    until All offspring of P are explored.
12:    Replace P with P_b for the next generation
13: end while
```

generation or iteration. Table 4.3 shows the percentage deviation and execution time using Pthread. Further, it shows that as the size of instances increases, the execution time increases in proportion to the number of processors used (e.g., *tai100a*).

4.6.3.3 GA on GPU

Using GA, we generated the random initial population of solutions, then we generated the offspring using the crossover and mutation operators for next generation and iteration. In GPU, we evaluated the solutions in parallel until the stopping condition is reached, then the results were sent back from GPU to CPU. The modified algorithm for GA is written in Algorithm 8. The percentage deviation and execution time to run on GPU are reported in Table 4.3. In this table, for *tai100a* instance, CPU takes 12.124 seconds and GPU takes 1.064 seconds for execution.

4.6.4 PSO Implementation

4.6.4.1 PSO on Serial Machine

PSO starts with random initial solutions or a population of particles, with each particle having a random initial velocity. Every particle has its personal best based on its experience or history, and global best for the whole group of particles. In every iteration, each particle updates its velocity according to Equation 2.3 and position according to Equation 2.4. As the iteration continues, each particle converges toward the optimal solution. In this study, velocity was measured in terms of the number of swaps of positions inside a solution. In Equation 2.3, we have taken the inertia factor (ω) 0.9, and two constants c_1 and c_2 as 2. The results of PSO are reported in Table 4.4, showing that as the instance size increases up to 40, the execution time slightly increases, but as the size increases from 50 to 100, the execution time on serial machine increases exponentially. Moreover, we used 500 initial solutions, with each solution being iterated 10 times to obtain the optimal solution.

4.6.4.2 PSO using Pthread

In this algorithm, an initial population of solutions is assigned to an equal number of processors. As the algorithm proceeds, every processor reaches toward the optimal solution. After the termination, all the optimal solutions for the processors are combined, and among them, the best solution is observed. The percentage deviation and execution time are reported in Table 4.4, with the Pthread execution time being significantly reduced as compared to CPU.

4.6.4.3 PSO on GPU

In PSO, we observed the random position and velocity of the whole swarm, which is generated on CPU. The particles position and velocity were used to generate the next position of the particles, which was evaluated in parallel on GPU. Every

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
unstructured, randomly generated instances, class 1								
tai30a	12.37	0.597	12.72	0.184	3.24	12.37	0.181	3.30
tai35a	12.30	1.132	13.44	0.310	3.65	12.30	0.244	4.64
tai40a	14.11	1.958	13.50	0.529	3.70	14.06	0.328	5.97
tai50a	13.15	4.677	14.04	1.279	3.66	13.15	0.500	9.35
tai60a	13.15	9.642	13.78	2.700	3.57	13.15	0.739	13.05
tai80a	12.32	30.559	12.08	8.282	3.69	12.54	1.390	21.98
tai100a	11.67	74.116	11.49	19.984	3.71	11.67	2.508	29.55
Real-life like instances, class 2								
tai60b	35.96	11.479	35.91	3.045	3.77	35.96	0.402	28.55
tai80b	35.59	36.175	32.95	9.694	3.73	35.59	0.760	47.60
tai100b	32.20	87.717	32.20	23.397	3.75	32.20	1.366	64.21
tai150b	23.80	444.755	24.33	118.480	3.75	23.80	4.737	93.89
tai64c	8.63	14.915	6.96	3.912	3.81	8.63	0.454	32.85
tai256c	**10.20**	**3784.968**	**10.20**	**1009.232**	**3.75**	**10.20**	**29.672**	**127.56**
Instances with grid-distances, class 3								
sko72	14.60	23.831	14.68	6.267	3.80	14.60	0.601	39.65
sko81	14.46	38.026	14.38	10.226	3.72	14.46	0.810	46.95
sko90	14.21	57.688	14.28	15.106	3.82	14.45	1.025	56.28
sko100a	13.76	87.812	13.43	23.410	3.75	13.76	1.373	63.96
Real-life instances, class 4								
els19	53.20	0.140	51.10	0.085	1.65	53.20	0.056	2.50
kra30a	29.38	0.717	28.02	0.238	3.01	29.38	0.108	6.64
kra32	28.88	0.919	31.67	0.296	3.10	29.36	0.118	7.79
ste36a	59.16	1.571	61.73	0.437	3.59	59.16	0.145	10.83

TABLE 4.4: Percentage deviation and exec. time of QAP using PSO on CPU, Pthread, and GPU

member of the swarm update its velocity based on the position from the local optimum. After reaching the stopping criteria, the final global optimum values were copied from GPU to CPU. The modified algorithm is written in Algorithm 9. The percentage deviation and execution time on GPU are reported in Table 4.4. It shows that the percentage deviation for both CPU and GPU are approximately equal, but the execution time on GPU is less than that on CPU, as for *tai100a* instance, CPU takes 74.116 seconds for execution, while GPU only takes 2.508 seconds. Moreover, as the size of the instances increases, GPU gives faster results as compared to CPU.

Algorithm 9 Particle Swarm Optimization (PSO) On GPU

Input: Distance matrix and Flow matrix
Output: Optimal solution.

1: Random initialization of the position and velocity of the whole swarm
2: Set the initial parameters for PSO
3: **repeat**
4: Evaluate the objective function $f(p_i)$
5: **for all** particles i **do**
6: Update velocities:
7: $v_i(t+1) = w * v_i(t) + c_1\theta_1(t)(p_{pbest_i}(t) - p_i(t)) + c_2\theta_2(t)(p_{gbest}(t) - p_i(t))$
8: Move next position: $p_i(t+1) = p_i(t) + v_i(t+1)$;
9: **if** $f(p_i) < f(pbest_i)$ **then**
10: $pbest = p_i$;
11: **end if**
12: **if** $f(p_i) < f(gbest_i)$ **then**
13: $gbest = p_i$;
14: **end if**
15: Update (p_i, v_i);
16: **end for**
17: **until** Stopping criteria reached

4.6.5 CSA Implementation

4.6.5.1 CSA on Serial Machine

In CSA, first, we fixed the initial parameters such as the flight length and probability of awareness. When the probability of awareness decreases, then it searches in the local region (exploitation-oriented), whereas when it increases, it explores the search space (exploration-oriented). Initially, for the serial machine, we generated a fixed number of initial solutions as the size of input, with each initial solution generating a neighbor solution using the adjacent pairwise exchange method. Among these neighbor solutions, we found the best one, which was used to initialize the memory of each crow for each initial solution. As the iteration increases, each crow updates its memory until the termination condition is reached. Among the memories of all crows, we found the best one giving the optimal solution for the QAP, taking flight length as 2, and probability of awareness as 0.15.

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
unstructured, randomly generated instances, class 1								
tai30a	16.75	0.195	13.34	0.275	0.71	14.79	0.149	1.31
tai35a	17.29	0.492	13.56	0.648	0.76	16.73	0.174	2.83
tai40a	14.23	0.695	15.05	0.808	0.86	15.80	0.200	3.48
tai50a	6.04	1.500	14.43	0.562	2.67	16.25	0.260	5.77
tai60a	14.38	0.988	14.46	0.460	2.15	15.49	0.300	3.29
tai80a	13.12	0.887	13.26	0.799	1.11	13.19	0.402	2.21
tai100a	10.99	6.232	12.51	3.183	1.96	12.53	0.509	12.24
Real-life like instances, class 2								
tai60b	39.83	0.425	42.82	0.508	0.84	41.36	0.300	1.42
tai80b	25.90	2.534	36.99	1.651	1.53	41.00	0.429	5.91
tai100b	17.76	3.485	36.72	2.884	1.21	37.97	0.506	6.89
tai150b	24.54	9.141	24.79	5.922	1.54	26.08	0.792	11.54
tai64c	10.45	0.723	8.82	0.531	1.36	14.08	0.322	2.25
tai256c	**11.65**	**84.351**	**10.52**	**22.504**	**3.75**	**12.99**	**1.398**	**60.34**
Instances with grid-distances, class 3								
sko72	15.72	3.067	15.63	0.749	4.09	15.68	0.363	8.45
sko81	15.66	4.093	15.40	0.777	5.27	16.11	0.655	6.25
sko90	14.35	3.882	14.74	1.795	2.16	15.16	0.456	8.51
sko100a	6.69	4.751	12.52	1.155	4.11	14.40	0.651	7.30
Real-life instances, class 4								
els19	74.81	0.179	56.17	0.224	0.80	47.73	0.112	1.60
kra30a	34.41	0.244	38.46	0.239	1.02	33.95	0.150	1.63
kra32	13.42	0.153	32.73	0.490	0.31	36.18	0.160	0.96
ste36a	38.30	0.177	84.65	0.281	0.63	88.35	0.180	0.98

TABLE 4.5: Percentage deviation and exec. time of QAP using CSA on CPU, Pthread, and GPU

4.6.5.2 CSA using Pthread

To implement on Pthread, first, we divided the initial solutions among processors, with each processor running the maximum number of iterations (initially fixed) and updating the memory of each crow. All processors run in parallel, while inside each processor this algorithm runs serially. After completing the execution of all processors, we found the best updated memory among all crows that gives our optimal solutions, with the results represented in Table 4.5.

4.6.5.3 CSA on GPU

To implement CSA on GPU, first, we fixed the initial parameters. Then, we calculated the cost of the initial solutions on CPU, and from each initial solution we generated the neighbor solutions using the adjacent pairwise exchange method. Afterwards, we evaluated the cost of the neighbor solutions on GPU. From each initial solution, we found the best possible neighbor solution cost that is set to the memories of each corresponding crow (for initial solution). Consequently, we run the CSA on CPU to find the next positions of the crows and evaluate their cost, and compared them to the solution stored in their memory. If the solution gave the best result, then the memory of the corresponding crow was updated until reaching the termination criteria. Finally, among the memories of all crows, we found the best solution cost, which is reported in Table 4.5. The modified algorithm for CSA is described in Algorithm 10. In Table 4.5, we can see that the percentage deviation has a slight difference on CPU, Pthread, and GPU because of the random generation of the initial positions; however, the execution time on GPU is significantly less that on CPU, as for *tai100a* instance, CPU takes 6.232 seconds, while GPU takes only 0.509 seconds.

4.6.6 TS Implementation

4.6.6.1 TS on Serial Machine

In TS, first, we fixed the size of the tabulist as the size of the instance. Then, we generated the fixed number of (size of tabulist) random initial solution and evaluated their cost and stored it in the tabulist. In each iteration, we generated the neighbor solution, through adjacent pairwise exchange methods and evaluated their cost, if it improves from the current solution then, we updated the tabulist. This procedure will continue until it reaches the terminating condition, and found the best optimal solution from the tabulist. To avoid, stuck in local minima, we implemented a diversification operator suggested by *James et al.* [50] to generate a

Algorithm 10 Crow Search Algorithm (CSA) On GPU

Input: Distance matrix and Flow matrix
Output: Optimal solution found.

Random initialization of positions of each crow in a search space
Evaluate using the objective function for positions of the crows
Random initialization of the memory of each crow
Set initial parameters for CSA
Generate and evaluate neighbor solution on GPU
Update the memory for each crow with the best neighbor solution cost
while $t < t_{max}$ **do**
    for $i = 1 : N$ **do**
        Evaluate objective function $f(p_i)$
        Randomly choose one of the crows among flock of size N to follow crows (e.g., j)
        Define the probability of awareness (PA)
        if $r_j > PA^{j,t}$ **then**
            $p^{i,t+1} = p^{i,t} + r_i \times f_l^{i,t} \times (m^{j,t} - p^{i,t})$
        else
            $p^{i,t+1}$ = a random position in search space
        end if
    end for
    Check the new positions feasibility using the objective function
    Evaluate the newly position of crows using the evaluate function
    Update the memory of crows
end while

new solution. We fixed the maximum number of failures (none of the tabulist elements not get updated in an iteration) as the size of the instance. We implemented diversification for a solution by simply changing the position of elements with step size from 2 to the size of the instance, i.e. if the step size is 2 then all odd position elements get placed first serially, then even position elements get placed there after to generate a new solution. In the CPU, we fixed the number of iterations as 10. In Table 4.6 execution time and percentage deviation are reported.

4.6.6.2 TS using Pthread

To implement on Pthread, first, we fixed tabulist size as the size of the instance, and then generated a random initial solution, evaluated their cost, and stored it in the tabulist. We call Pthread and every thread gets the tabulist, generates a neighbor solution, if it improves the current solution then update the tabulist.

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
unstructured, randomly generated instances, class 1								
tai30a	10.98	0.083	8.14	0.039	2.13	10.98	0.018	4.61
tai35a	11.66	0.144	9.53	0.052	2.77	11.66	0.026	5.54
tai40a	11.76	0.238	9.28	0.073	3.26	11.76	0.036	6.61
tai50a	11.54	0.576	11.30	0.191	3.02	11.54	0.064	9.00
tai60a	12.14	1.218	12.03	0.407	2.99	12.14	0.106	11.49
tai80a	11.25	3.865	11.12	1.231	3.14	11.25	0.259	14.92
tai100a	11.12	10.218	10.81	3.218	3.18	11.12	0.553	18.48
Real-life like instances, class 2								
tai60b	30.52	1.207	28.86	0.398	3.03	30.52	0.105	11.50
tai80b	32.75	3.859	27.80	1.280	3.01	32.75	0.256	15.07
tai100b	32.21	9.581	29.95	2.991	3.20	32.21	0.568	16.87
tai150b	23.88	49.494	22.01	16.992	2.91	23.88	2.399	20.63
tai64c	0.59	1.456	0.59	0.389	3.74	0.59	0.116	12.55
tai256c	**12.96**	**481.051**	**8.62**	**137.210**	**3.51**	**12.96**	**17.449**	**27.57**
Instances with grid-distances, class 3								
sko72	12.98	2.486	11.54	0.781	3.18	12.98	0.187	13.29
sko81	12.92	4.071	12.54	1.322	3.08	12.92	0.265	15.36
sko90	12.72	6.160	11.88	1.921	3.21	12.72	0.380	16.21
sko100a	11.81	9.553	12.04	3.041	3.14	11.81	0.540	17.69
Real-life instances, class 4								
els19	50.0	0.019	3.47	0.013	1.46	50.0	0.007	2.71
kra30a	20.04	0.083	17.39	0.037	2.24	20.04	0.019	4.37
kra32	21.86	0.106	16.91	0.040	2.65	21.86	0.021	5.05
ste36a	50.51	0.161	43.25	0.061	2.64	50.51	0.028	5.75

TABLE 4.6: Percentage deviation and exec. time of QAP using TS on CPU, Pthread, and GPU

When Pthread join then merges all the thread tabulists and from that, we get the best optimal solution, which is reported in Table 4.6.

4.6.6.3 TS on GPU

To implement TS on GPU, first, we fixed the GPU parameters and then transferred the input data from CPU to GPU. In GPU, we fixed the number of iterations as 10 and the size of the tabulist as the size of the instance. In each iteration of TS, we generate and evaluate the neighbor solution on GPU, copy the neighbor solution objective value from GPU to CPU host memory, compared with the tabulist, and

Algorithm 11 Tabu Search (TS) On GPU

Input: Distance matrix and Flow matrix
Output: Optimal solution found.

```
1: Random initialization of an initial solution
2: Evaluate the solution using objective function
3: Set initial parameters for TS
4: Initialize tabulist
5: repeat
6:     for each generated neighbor solution on GPU do
7:         Evaluate neighbor solution using objective function
8:         Insert the resulting objective value into the neighbor solution objective
   value
9:     end for
10:    Copy the neighbor solution objective value from GPU to CPU host memory
11:    for each neighbor solution objective value do
12:        Compare with tabulist
13:        if It improves the objective function then
14:            Select that neighbor solution
15:            Update the tabulist
16:        end if
17:    end for
18:    Copy the selected solution on GPU device memory
19:    if tabulist fails to update and reached maxFailures then
20:        Perform diversification
21:    end if
22: until a maximum number of iterations reached
```

if the objective value improves the objective function, then update the tabulist and select the best neighbor solution. Finally copy the selected neighbor solution on GPU device memory. This process continues until it reaches the stopping condition. The modified algorithm of TS is described in Algorithm 11. Finally, the best optimal solution is found from the tabulist and results are noted in Table 4.6. Here in Table 4.6, we can see that speed up on GPU is increases as the size of the instance increases. The deviation on Pthread is less, as compared to CPU and GPU; because here each thread is executing with a separate tabulist, so it explores more new solutions and gives less deviation.

4.7 Experimental Results

4.7.1 Comparison of the Serial, Pthread, and GPU Versions of QAP Meta-heuristics

In this study, we have considered four classes of QAP instances classified by *Thomas Stutzle* [64]. These are unstructured, randomly generated instances (class 1), real-life like instances (class 2), instances with grid-distances (class 3), and real-life instances (class 4). For each instance, we calculated the objective value of QAP by implementing each meta-heuristic on serial, Pthread, and GPU machines. We calculated the difference between the objective value and the standard QAPLIB cost for each instance. Further, for each meta-heuristic, we have considered the average deviation and average execution time for CPU, Pthread, and GPU for all four classes of instances. The Comparison for all meta-heuristics are reported in Table 4.7, which demonstrates that ILS produces the least deviation from the standard optimal cost, but it takes a slightly more execution time to run on GPU as compared to CSA. Moreover, we can also see that CSA demonstrates more deviation, but it executes too quickly to get the optimal solution and that the GPU version on GPU, average computation is up to 28 times faster than serial machines on CPU for taixxa instances.

We also compared the execution time of different meta-heuristics (ILS, SA, GA, PSO, CSA, and TS) between CPU, Pthread, and GPU for only one instance *tai80a*, which is illustrated in Figure 4.3 shows that the CSA performance is superior to other meta-heuristics. In addition, for *tai80a* instances on CPU and Pthread, PSO takes more time for execution compared to other meta-heuristics, while on GPU, GA takes more time for execution as compared to other meta-heuristics. We analyzed the speedup on GPU of PSO meta-heuristic for all the considered instances, which is illustrated in Figure 4.4, here we can see that for *tai256c* maximum speedup around 128 is achieved.

Figure 4.5 shows the execution time for class 1 instances for all targeted meta-heuristics on GPU. The SA is showing worst-performance up to size 80 (up to *tai80a*), but for *tai100a* the PSO is taking the highest time on GPU. We can also observe that as the size of the instances increases, the execution time on GPU and their *speedup* also increases. In Figure 4.6, *tai256c* is taking more time on GPU among all the class 2 instances. Execution time for class 3 instances is illustrated in Figure 4.7; in this figure, SA is showing the worst performance among all the other meta-heuristics. Figure 4.8 illustrates the execution time for class 4 instances, and the TS demonstrating the best performance among all the other meta-heuristics.

We also compared the execution time (ET) and percentage deviation (D) on CPU, Pthread, and on GPU, for selected instances and noted the result in Table 4.8, here we can see PSO is taking more time to execute in CPU but it gives the highest speedup around 128 on GPU because as in PSO, every particle used to update the velocity based on the target of optimal cost, so for large size instances the amount of parallelization in PSO is increases and velocity of the particle is getting calculated efficiently, and hence a higher speedup on GPU is achieved for large size instances. We compared the percentage deviation of all meta-heuristics by fixing the execution time 2 seconds on GPU, which is illustrated in Table 4.9. In this table for *tai100a* and *tai150b* the ILS approach, for *tai100b* and *sko100a* the TS approach, for *tai256c* the SA approach is showing less deviation from standard QAP libraries.

We fixed the number of evaluations of the cost function for all the considered meta-heuristics as $(500 \times 10 \times \binom{n}{2})$, where n is the size of the input instance. We reported the result in Table 4.10, here we can see that for instances of the same size 100; of *tai100a, tai100b, and sko100a*, GA takes more times and TS gives the best score (or less percentage deviation) as compared to all other considered meta-heuristics.

Metaheuristics	**D**	**ET CPU**	**ET Pthread**	**ET GPU**
ILS	7.37	9.515	2.955	0.333
SA	8.72	9.953	2.961	1.128
GA	13.70	5.102	1.546	0.669
PSO	**12.83**	**17.526**	**4.753**	**0.841**
CSA	14.01	1.570	0.962	0.285
TS	11.10	2.334	0.745	0.152

TABLE 4.7: A comparison of meta-heuristics on CPU, Pthread, and GPU of taixxa instances

I	ILS		SA		GA		PSO		CSA		TS	
CPU	D	ET	D	ET	D	ET	D	ET	D	ET	D	ET
tai100a	8.9	34.22	9.4	35.34	12.3	12.12	11.7	74.12	11.0	6.23	11.1	10.22
tai100b	17.5	37.45	18.7	39.16	38.5	12.07	32.2	87.72	17.8	3.49	32.2	9.58
sko100a	9.5	38.67	14.4	38.65	14.1	12.09	13.8	87.81	6.7	4.75	11.8	9.55
tai150b	17.6	126.21	18.2	132.72	26.0	24.63	23.8	444.76	24.5	9.14	23.9	49.49
tai256c	**4.9**	**769.28**	**4.8**	**769.24**	**12.3**	**66.93**	**10.2**	**3784.97**	**11.7**	**84.35**	**13.0**	**481.05**
GPU												
tai100a	8.9	0.62	9.2	1.92	12.3	1.06	11.7	2.51	12.5	0.51	11.1	0.55
tai100b	17.5	0.60	19.1	1.69	38.5	1.07	32.2	1.37	38.0	0.51	32.2	0.57
sko100a	9.5	0.60	14.6	1.70	14.1	1.07	13.8	1.37	14.4	0.65	11.8	0.54
tai150b	17.6	1.51	17.6	4.17	26.0	1.72	23.9	4.74	26.1	0.79	23.9	2.40
tai256c	**4.9**	**7.15**	**4.7**	**12.61**	**12.3**	**3.04**	**10.2**	**29.67**	**13.0**	**1.40**	**13.0**	**17.45**

TABLE 4.8: A comparison of meta-heuristics on CPU and GPU of selected QAP instances

Instance	ILS	SA	GA	PSO	CSA	TS
tai100a	8.55	8.95	12.32	11.49	12.53	12.27
tai100b	17.47	18.14	38.5	32.2	37.97	10.42
sko100a	9.05	14.02	14.14	12.56	14.4	5.01
tai150b	17.64	17.91	25.99	23.48	26.08	24.7
tai256c	**4.96**	**4.95**	**12.33**	**11.55**	**12.99**	**14.32**

TABLE 4.9: Percentage deviation of meta-heuristics on GPU for fixed exec. time 2 sec.

4.7.2 Statistical Analysis of all the Meta-heuristics on GPU

We analyzed all the meta-heuristics for selected instances of the same size as *tai100a, tai100b, sko100a*, and for large size instance *tai256c* on GPU. We considered execution times of 20 *runs* for each of the selected instances and plotted the *boxplot* for each instance. *Boxplot* is a standardized way for analyzing the

I	ILS		SA		GA		PSO		CSA		TS	
CPU	D	ET	D	ET	D	ET	D	ET	D	ET	D	ET
tai30a	4.9	0.87	6.1	0.96	14.4	17.47	12.4	0.60	15.8	0.02	5.9	1.24
tai64c	0.0	9.73	0.1	10.37	14.7	273.50	8.6	14.91	17.3	0.25	0.1	16.77
tai100a	8.9	34.22	9.4	35.34	12.8	1310.56	11.7	74.12	12.5	1.84	5.2	51.19
tai100b	17.5	37.45	18.7	39.16	41.2	1297.18	32.2	87.72	37.5	2.15	9.2	50.12
sko100a	9.5	38.67	14.4	38.65	15.0	1374.29	13.8	87.81	14.3	2.02	3.8	60.66
GPU												
tai30a	4.9	0.19	7.3	0.76	14.4	4.33	12.4	0.18	14.8	0.01	5.9	0.22
tai64c	0.0	0.33	0.0	1.15	14.7	35.36	8.6	0.45	14.1	0.01	0.1	0.78
tai100a	8.9	0.62	9.2	1.92	12.8	131.44	11.7	2.51	12.5	0.02	5.2	2.44
tai100b	17.5	0.60	19.1	1.69	41.2	131.44	32.2	1.37	38.0	0.02	9.2	2.50
sko100a	9.5	0.60	14.6	1.70	15.0	131.62	13.8	1.37	14.4	0.02	3.8	2.46

TABLE 4.10: A comparison of meta-heuristics on CPU and GPU for common termination criterion (by fixing no. of evaluations)

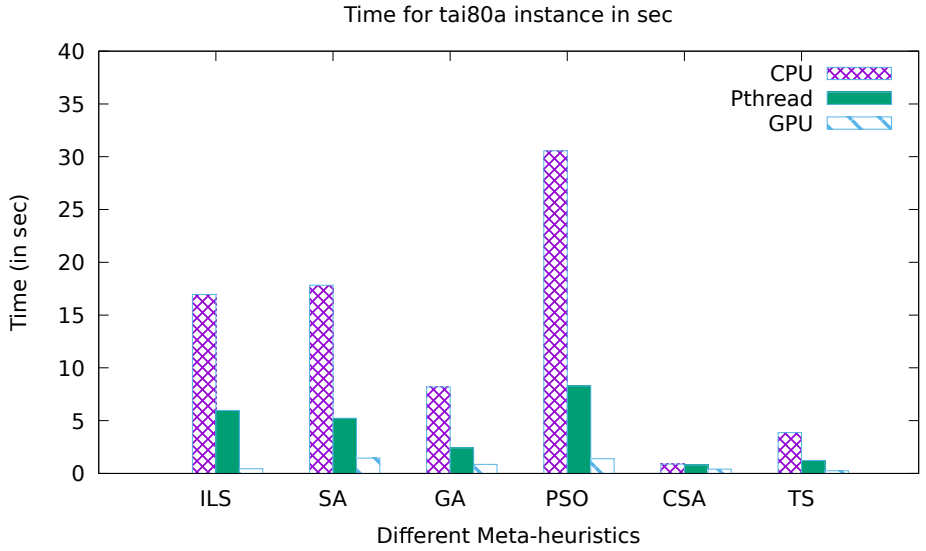

FIGURE 4.3: Exec. time of meta-heuristics for tai80a instance on CPU, Pthread, and GPU

data, using this we can find minimum, first quartile, median, third quartile, maximum, and outliers for the data. Figure 4.9 shows the execution time for *tai100a* instance on GPU. The meta-heuristics SA and TS showing very little variance while CSA shows higher variance among all meta-heuristics. For the *tai100b* instance, Figure 4.10 PSO shows the outliers and CSA gives the highest variance. Figure 4.11 illustrates the execution time for the *sko100a* instance, in this case, also the meta-heuristic CSA has the most variance and PSO has the very little variance in execution time. For large size instance, *tai256c* execution time is shown

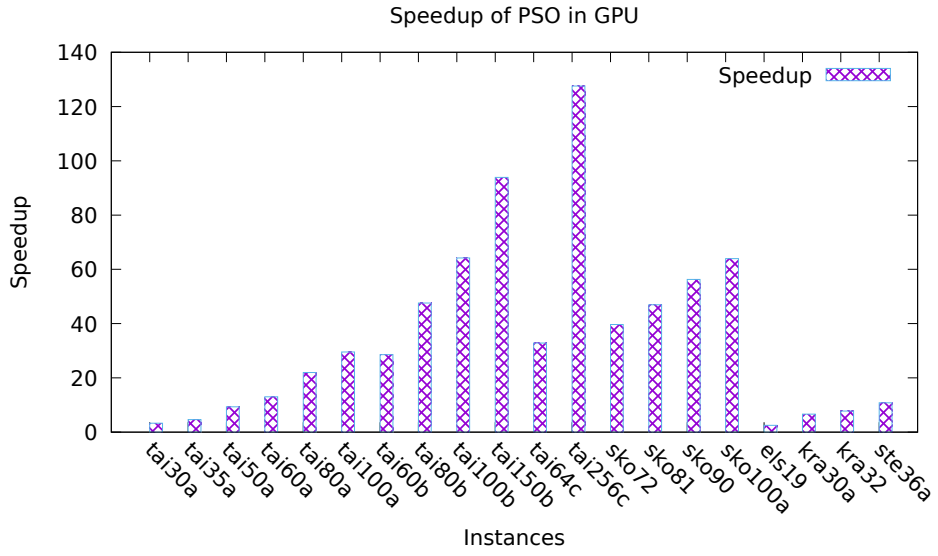

FIGURE 4.4: Speedup on GPU for PSO

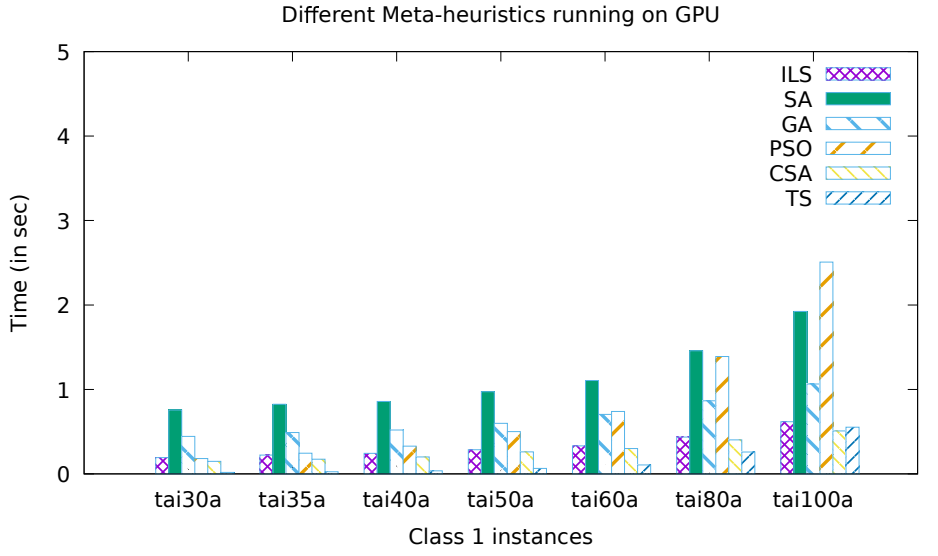

FIGURE 4.5: Exec. time on GPU for class 1 instances

in Figure 4.12, GA have very less and CSA have the highest variance among all the meta-heuristics.

For all the meta-heuristics, we plotted *boxplots* by setting the same termination criteria as the number of evaluations of the cost function $(500 \times 10 \times \binom{n}{2})$ on GPU, where n denotes the size of input instance. For this, we choose instances of the same size as *tai100a, tai100b, sko100a*, and other instances as *tai64c*. We

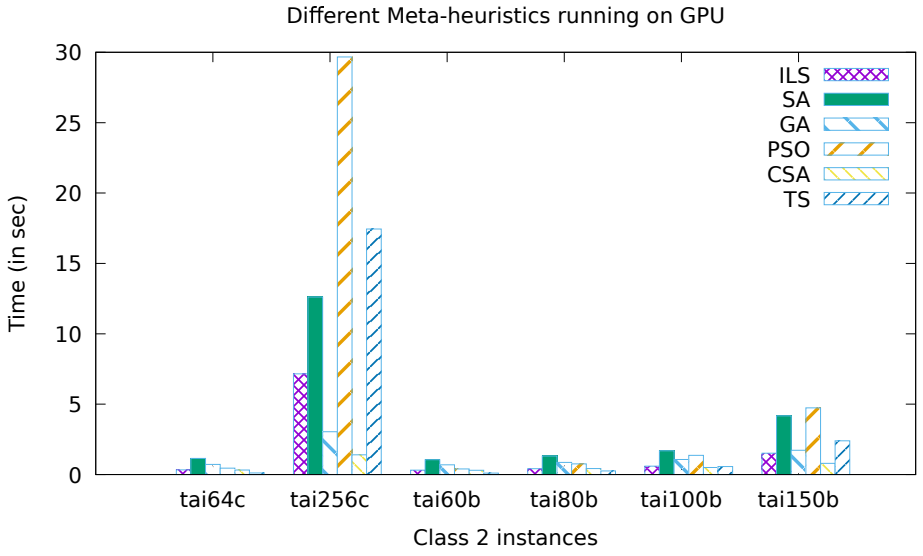

FIGURE 4.6: Exec. time on GPU for class 2 instances

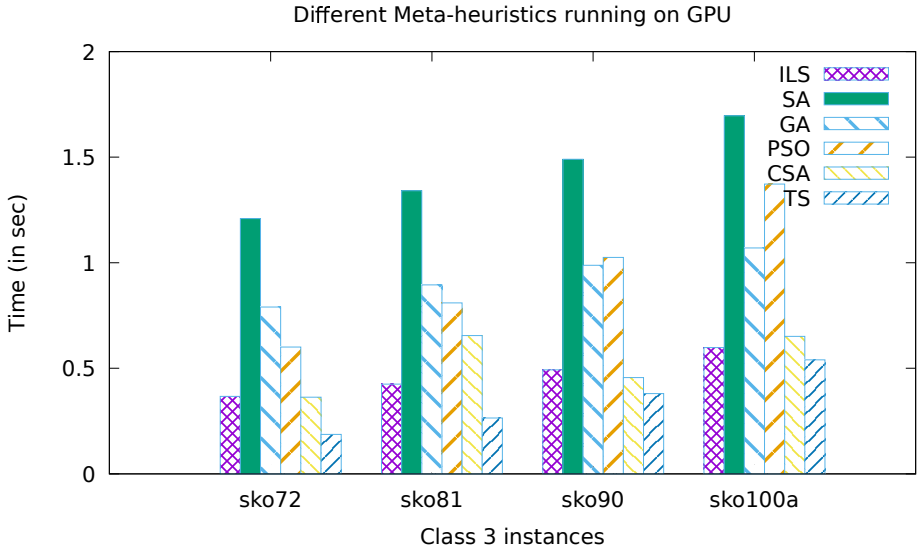

FIGURE 4.7: Exec. time on GPU for class 3 instances

considered the optimal cost of 20 *runs* for each selected instances. Figure 4.13 shows the optimal cost of all the considered meta-heuristics for *tai100a* instance, here SA shows the high variance while GA shows the high deviation, and TS gives the best optimal cost (or less deviation) from standard QAPLIB. Similarly in Figure 4.14 for *tai100b* instance. For *sko100a* instance; in Figure 4.15, SA shows the high variance and high deviation from PSO, while TS shows the least deviation among all the considered meta-heuristics. We also considered other instance as

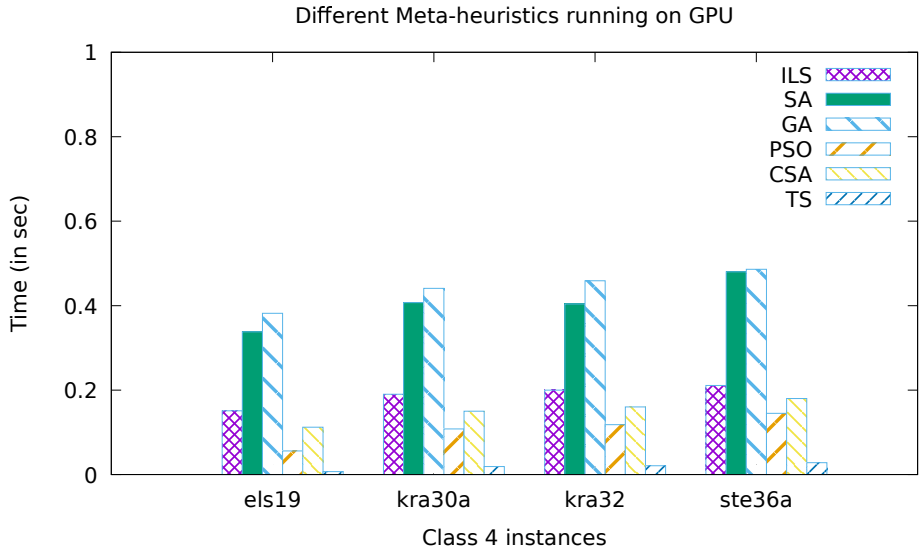

FIGURE 4.8: Exec. time on GPU for class 4 instances

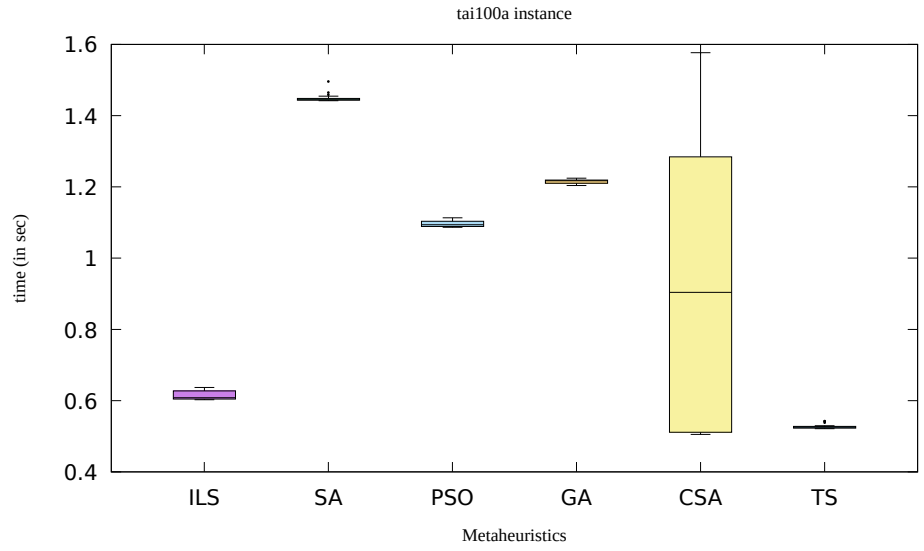

FIGURE 4.9: Boxplot of exec.time on GPU for tai100a instance

tai64c of taillard c type instance and the result for this shown in Figure 4.16, here again, SA shows the high variance and GA high deviation, but SA demonstrates the less deviation as compared to PSO, GA, and CSA and the winner TS shows the best optimal cost among all the considered meta-heuristics.

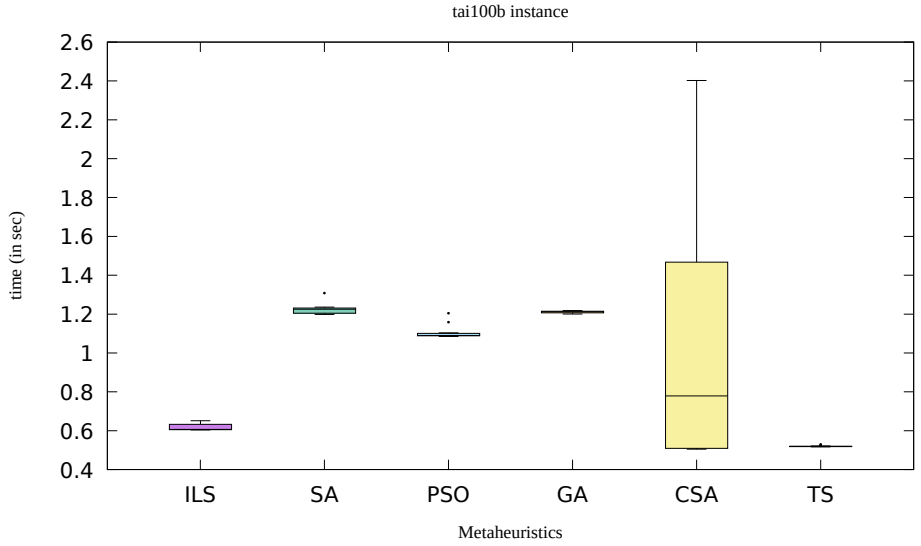

FIGURE 4.10: Boxplot of exec.time on GPU for tai100b instance

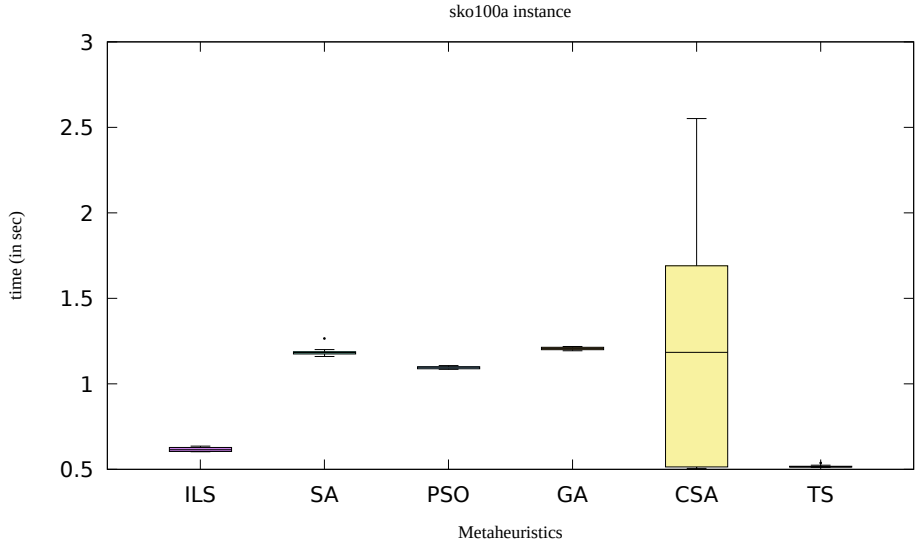

FIGURE 4.11: Boxplot of exec.time on GPU for sko100a instance

4.8 Performance Analysis of Meta-heuristics

In this section, we conduct a performance analysis of the targeted meta-heuristics on a sequential machine. We used a GNU profiling tool (i.e., gprof) [13] to check the performance of meta-heuristics on a QAP program on a serial machine. We analyzed the performance of QAP by fixing 500 initial solutions, with each solution running 10 iterations on the sequential machine (i.e., CPU) by taking the standard

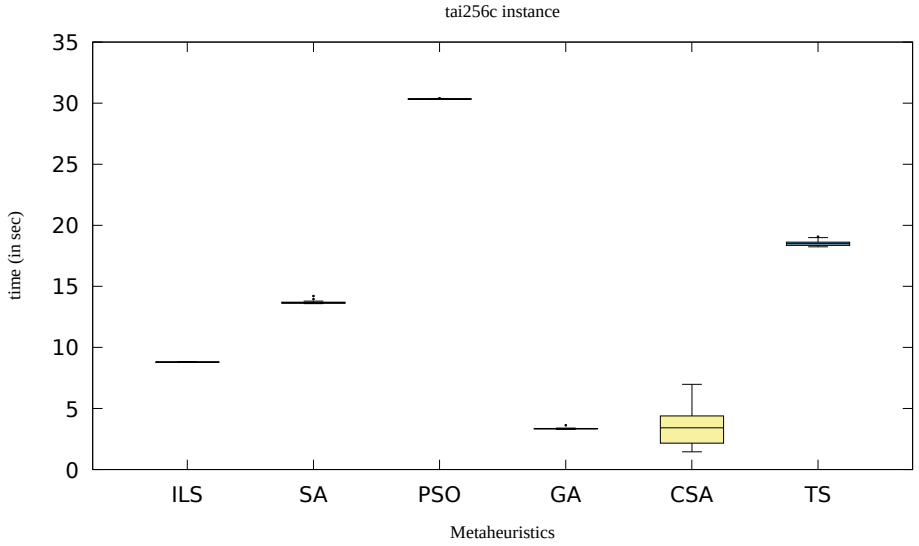

FIGURE 4.12: Boxplot of exec.time on GPU for tai256c instance

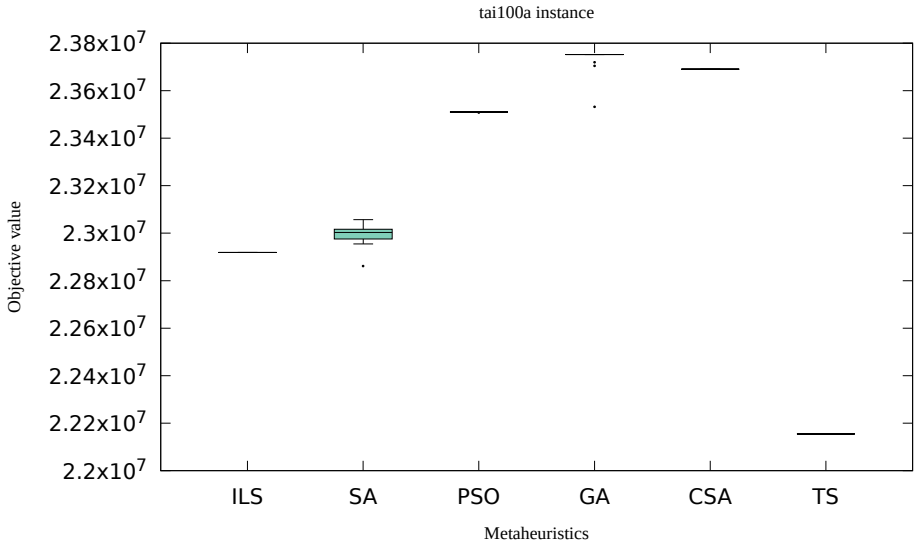

FIGURE 4.13: Boxplots of objective value on GPU for tai100a instance

taillard instances from QAPLIB [6]. The results of one instance *tai35a* on the gprof profiling tool are shown below in Table 4.11.

The above gprof results (in Table 4.11) shows that QAP spent 99.94% of time on *evaluateCost* function. Thus, if we parallelize the *evaluateCost* function, then the performance of QAP will improve. When we analyzed the program, we found the following three main parallel sections:

- Parallel section (P1): Generation of the best solution from many initial solutions

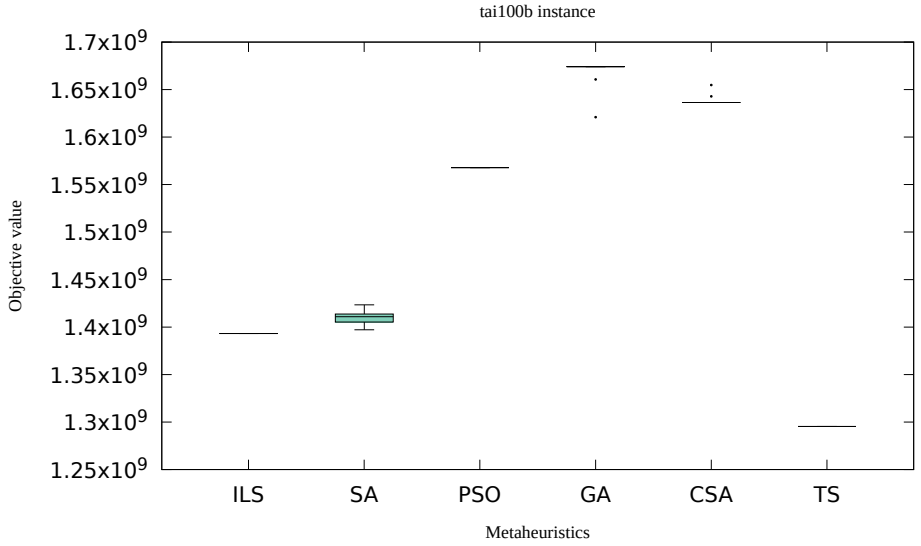

FIGURE 4.14: Boxplots of objective value on GPU for tai100b instance

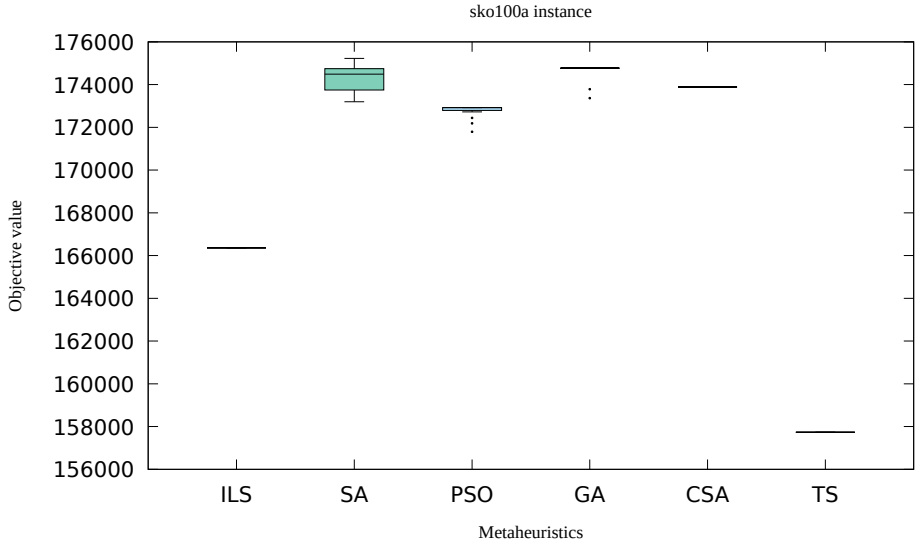

FIGURE 4.15: Boxplots of objective value on GPU for sko100a instance

gprof output

```
Flat profile:
Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  us/call  us/call  name
99.94    26.61     26.61  6125500     4.34     4.34  evaluateCost
0.19     26.66     0.05                              main
0.04     26.67     0.01   6125000     0.00     0.00  swap
0.00     26.67     0.00      8640     0.00     0.00  cost_solution
```

TABLE 4.11: gprof output

in parallel, where each processor handles the work of one initial solution. It

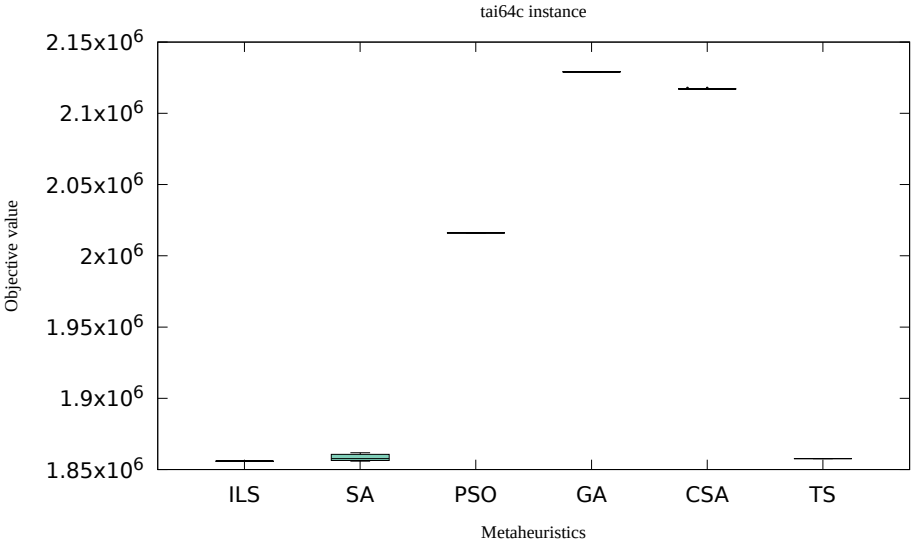

FIGURE 4.16: Boxplots of objective value on GPU for tai64c instance

involves many iterations of the neighbor solutions generation, evaluation, and selection.

- Parallel section (P2): Generation of many neighbor solutions and then conducting evaluation of these neighbor solutions in parallel, where each solution is evaluated by a processor.
- Parallel section (P3): Evaluation of a single solution in parallel.

4.9 Task Graph Generation using Contech

Contech is an open source compiler-based framework developed by Railing [65] to generate task graphs from parallel programming. This framework generates task graphs by following a combination of programming languages such as C, C++, FORTRAN, pthreads, OpenMP, or MPI. The task graphs generated by Contech are represented using four tasks: creates (C), joins (J), sync (S), and barrier (B), while the other task work (W) represents all the tasks in the task graph. The *Create (C)* task increases the possibility of parallelism, where the execution of two or more actions can be done simultaneously. The *Join (J)* task is the opposite of the create task, as two or more actions are joined together, which

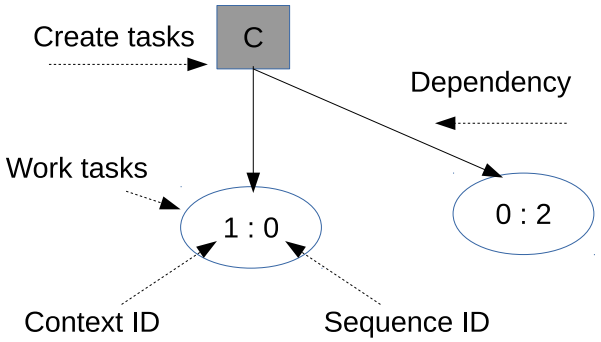

FIGURE 4.17: Contech Task Graph Visualization

reduces the parallelism. The *Sync (S)* task gives the orders between the tasks without affecting the parallelism in the task graph. Generally, the sync task may has a semaphore or condition variable, when the tasks are waiting for each other. Contrary to the sync task, where the order between the tasks is one to one (e.g., lock or one to many: condition variable), the *Barrier (B)* task is all to all. The *Work* task includes an execution sequence and memory action which is same as in the program. The Contech task graph representation is shown in Figure 4.17, in which each Contech task is denoted by $< ContextID >:< SequenceID >$, where *context ID* denotes the thread ID and *sequence ID* denotes the consistently increasing or non-decreasing identifier for each context in the task graph. The Dependency between the two tasks is denoted by the arrow in the task graph.

We generated the task graph for ILS using Contech tools by fixing the initial numbers of solutions and iteration, and wrote the ILS program using the C and OpenMP programming languages.

The task graph for parallel section $P1$ is shown in Figure 4.18. This figure demonstrates that there are many create (c) and join (j) actions that can be parallelized, which is the function of parallel section $P2$. Additionally, the main task with thread id 0 has 12 dependencies.

The task graph of parallel section $P2$ is represented in Figure 4.19. This figure shows that there is one more parallel section $P3$ that can be parallelized. In parallel section $P2$ the main task with thread id 0 has 8 dependencies.

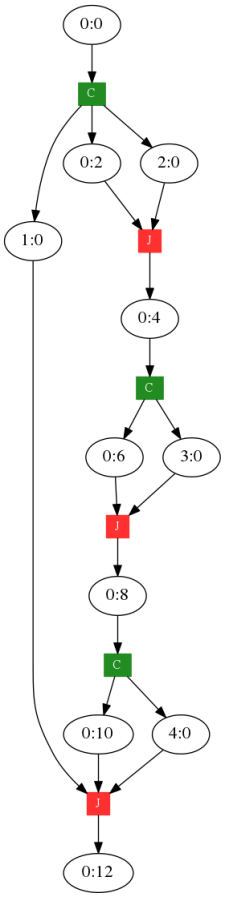

FIGURE 4.18: Task graph of ILS for P1

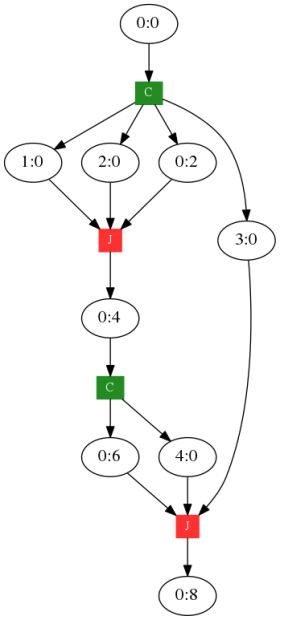

FIGURE 4.19: Task graph of ILS for P2

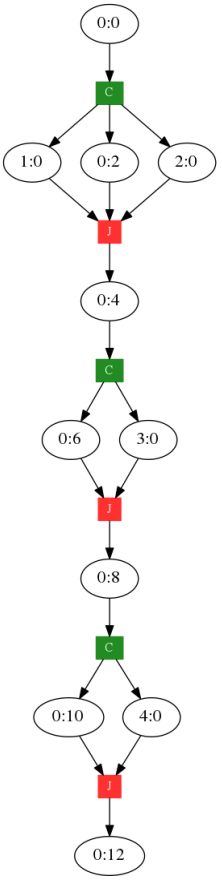

FIGURE 4.20: Task graph of ILS for P3

The last parallel section $P3$ is represented in Figure 4.20, demonstrating the existence of many create and join actions that have a small dependency. Thus, each dependency can be parallelized using many small cores.

4.9.1 Analysis of the Task Graphs

We analyzed the task graph based on the suitable input from *gprof* profiling tools and *Contech* task graph generation tools. Based on the analysis, we suggest the suitable target architecture for each parallel section of the task graph.

- Target for parallel section $P1$: From Figure 4.18, we can see that the P1 parallel section is suitable for the OpenMP type of parallelism with executing threads on a bigger core (e.g., Intel Xeon, Pentium cores). The reasons behind this

suggestion are: (a) it generates neighbor solutions, evaluate them, find the best solution, and then repeat the process with many iterations for each initial solution; and (b) one processor is responsible for finding the best solution from one initial solution.

- Target for parallel section $P2$: From Figure 4.19, this parallel section generates many neighbor solutions from the initial solutions, with the evaluation of all neighbor solutions done in parallel independently. Each neighbor solution evaluation takes a small amount of time, but is called for numerous times. Thus, the GPU or SIMD architecture is suitable for this parallel section.

- Target for parallel section $P3$: From figure 4.20, as we evaluate the individual solutions in parallel, this parallel section involves a reduction operation. Thus, OpenMP is suitable for this parallel section, and many chunks of reduction can be done efficiently on a smaller number of bigger cores.

4.10 Summary

This chapter explored which meta-heuristic is the most suitable for solving the QAP using the best available, accelerated machine. We compared the execution time and optimal cost with the recent optimal cost from QAPLIB; we achieved 127 times speedup on GPU compared to the serial version on CPU.

In this study, we compared the sequential and parallel execution of algorithms by applying both types of meta-heuristics (*i.e.,* single solution-based and population-based) on an accelerated machine, finding that among the six studied meta-heuristics, ILS (for small instance) and SA (for large instance) for all the four classes [64] demonstrates the least deviation from the recent standard online library QAPLIB. For small-size instances, the meta-heuristic TS shows the least average run-time on GPU, and CSA has the least average run-time on CPU, while for large-size instances, CSA shows the least execution time in GPU. By using the uniform termination condition for all the meta-heuristics by fixing the number of

evaluations of the cost function, the meta-heuristic TS shows the least deviation among all other considered meta-heuristics. Moreover, we analyzed the performance of meta-heuristics by finding the most appropriate parallel section, based on which we mapped each parallel section to the architecture.

Chapter 5

Analysis of Iterated Local Search Meta-heuristic on GPU Spatial Memory

This chapter deals with a logical extension of GPU hardware spatial memory for QAP using the ILS meta-heuristic. Here we discuss the use of memory properties for efficient utilization of GPU hardware and performance analysis of the ILS meta-heuristic.

5.1 Introduction

As we discussed, the detailed analysis of meta-heuristics for QAP is in Chapter 4. Here we focused mainly on the utilization of GPU different memories. GPU has a highly complex memory hierarchy to exploit its potential for paralyzing the data. An NVIDIA GPU Kepler architecture generally has more than eight types of memories (global, shared, constant, texture, and various caches). In paper [58] by Stoltzfus et al., the works have shown that using proper utilization of GPU memory hierarchy for placing the data improves the performance of the GPU. In QAP, we have taken the input in matrix form, which is *distance matrix* and

flow matrix; these matrices do not change during the entire program execution. So, mapping it on the read-only memory (constant, and texture) improves the performance from using only global memory.

The layout of this chapter is as follows: GPU memory architecture is discussed in section 5.2, then we describe the accelerated system or GPU used in this work in section 5.3. We illustrated the utilization of GPU memory in section 5.4. Experimental results are recorded in the section 5.4 followed by summary in section 5.6.

5.2 GPU Memory Architecture

To effectively utilize the computational capability of GPUs, memory access efficiency is crucial. GPU is equipped with different levels of memory with different characteristics, namely, global, constant, texture, shared, and registers, as shown in Figure 5.1. A high level overview of each memory is described in the following section.

5.2.1 Global Memory

Global memory is the most significant off-chip memory, considered the GPU's main memory by default. It has limited bandwidth and long latencies compared to on-chip memory or cache. Global memory is located in device memory, accessible by memory transactions of 32, 64, or 128 bytes. The memory transactions, which are in terms of 32, 64, or 128 bytes, can be only read or written. The throughput of GPU memory varies based on the compute capability of the GPU device.

5.2.2 Shared Memory

Shared memory is on-chip memory, which has low latency and high bandwidth. GPUs use shared memory to distribute processing among all the active threads in the streaming multiprocessor (SM).

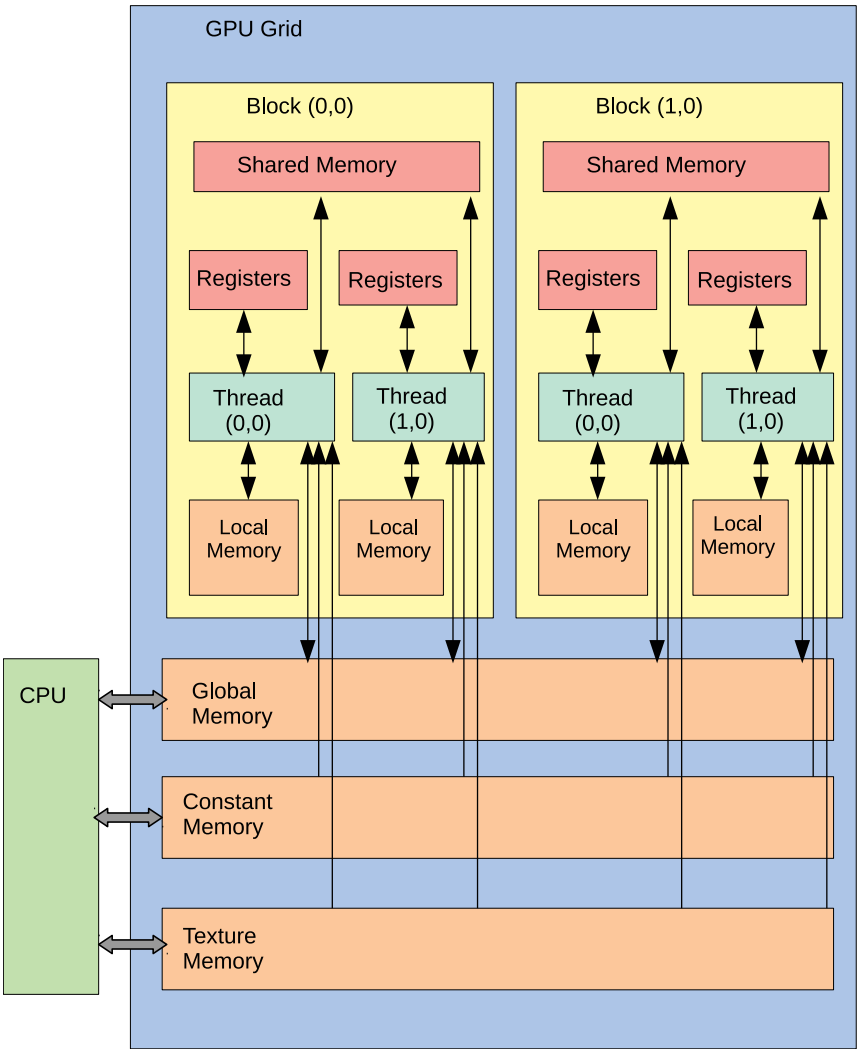

FIGURE 5.1: GPU memory hierarchy

5.2.3 Constant Memory

Constant memory is read-only memory, which is stored in the device memory. It physically has the exact location as global memory. This memory space is globally visible to all threads.

5.2.4 Texture Memory

Texture memory is also a read-only memory. It has off-chip memory space, which is optimized for the 2D spatial locality, so the threads in a warp that use it to

access 2D data will achieve the best performance. It is mainly suited for those threads that access memory addresses close to each other in 2D.

5.3 The Accelerated System

We used a GPU card from Nvidia, the GeForce GTX 980 Ti, which has 2816 CUDA cores, 6 GB DDR5 RAM, 22 streaming multiprocessor (SM), and 64 warps per SM. This device has the CUDA toolkit 8.0 installed. Table 5.1 describes the short overview of GPU cards.

TABLE 5.1: Nvidia GeForce GTX 980 Ti Configuration

No. of multiprocessors	22
No. of registers per block	65536
Global memory	5.94 MB
Constant memory	64 KB
Shared memory per block	48 KB
Warp size	32
Maximum no. of threads per block	1024
Maximum no. of threads per multiprocessors	2048
Maximum no. of warps per multiprocessors	64

5.4 Utilization of GPU Memory

The GPU supports programmable memory, where a user can write a program for the use of memory to utilize the resources of GPU architecture. Besides the global memory of GPU, it also has fast memory systems such as the shared, constant, texture, and local memory. By using the GPU spatial memory, we can further reduce the execution time for solving QAP. GPU shared memory is used when all threads within a block need to access the same data. The GPU constant memory is used for read-only data accessed uniformly by threads in a warp. It performs best when all threads in warp access the exact location in the constant memory. Texture memory is also read-only; it performs best when all reads in a warp are physically adjacent to each other. The GPU local memory is used when data is

Instance	GPU(GM+LM)			GPU(GM+CM)			GPU(GM+SM)		
	D (%)	**ET time**	**S**	**D (%)**	**ET time**	**S**	**D (%)**	**ET time**	**S**
tai30a	4.87	0.205	4.26	4.87	0.511	1.7	4.87	0.175	4.98
tai35a	5.78	0.236	5.93	5.78	0.612	2.28	5.78	0.196	7.13
tai40a	7.13	0.261	7.89	7.13	0.757	2.72	7.13	0.211	9.77
tai50a	8.04	0.306	13.07	8.04	1.01	3.96	8.04	0.249	16.1
tai60a	8.47	0.36	19.74	8.47	1.268	5.6	8.47	0.287	24.69
tai80a	8.43	0.48	35.3	8.43	3.267	5.19	8.43	0.403	42.08
tai100a	8.87	0.677	50.52	8.87	Out	Out	8.87	0.674	50.77

TABLE 5.2: Percentage deviation and exec. time of QAP on GPU local, constant, and shared memory

required to be accessed by only a particular thread i.e. data is only visible to the thread that wrote it and ended when threads are destroyed.

5.5 Experimental Results

We implemented the ILS meta-heuristic using the local, shared, and constant memory and the results are reported in Table 5.2. Because the QAP input, such as the distance matrix and flow matrix, is always constant, we put it on a read-only memory i.e. the Constant memory. In this table 5.2, we can see that for instance *tai80a* the speedup on GPU for local memory is 35.3, constant memory is 5.19, and shared memory is 42.08. So here, on Shared memory, GPU is performing best, while on constant memory, GPU performance is worst. We used Nvidia GeForce GTX 980 Ti GPU card, which has only 48KB Constant memory or cache, so instances of size greater than 100 show out of memory and are not executed. We also implemented texture memory by varying one with constant and texture memory, like we have put one-time distance matrix input on constant memory and flow matrix on texture memory. Similarly, we put the distance matrix on texture memory and the flow matrix on constant memory for another variation. We noted the results in Table 5.3. In this table, we observed that when we put on mixed with constant and texture memory, speedup on GPU is more significant than only constant memory but worst than only texture memory. For instance,

Instance	GPU (GM+TM)			GPU(GM+ CM(d)+TM(f))			GPU(GM+ TM(d)+CM(f))		
	D (%)	ET time	S	D (%)	ET time	S	D (%)	ET time	S
tai30a	4.87	0.193	4.52	4.87	0.226	3.85	4.87	0.224	3.88
tai35a	5.78	0.22	6.34	5.78	0.301	4.64	5.78	0.312	4.47
tai40a	7.13	0.241	8.52	7.13	0.37	5.55	7.13	0.388	5.3
tai50a	8.04	0.285	14.02	8.04	0.505	7.92	8.04	0.548	7.3
tai60a	8.47	0.33	21.51	8.47	0.623	11.39	8.47	0.683	10.4
tai80a	8.43	0.435	39	8.43	0.931	18.2	8.43	0.949	17.86
tai100a	8.87	0.619	55.31	8.87	1.626	21.04	8.87	2.941	11.63

TABLE 5.3: Percentage deviation and exec. time of QAP on GPU texture, and constant memory

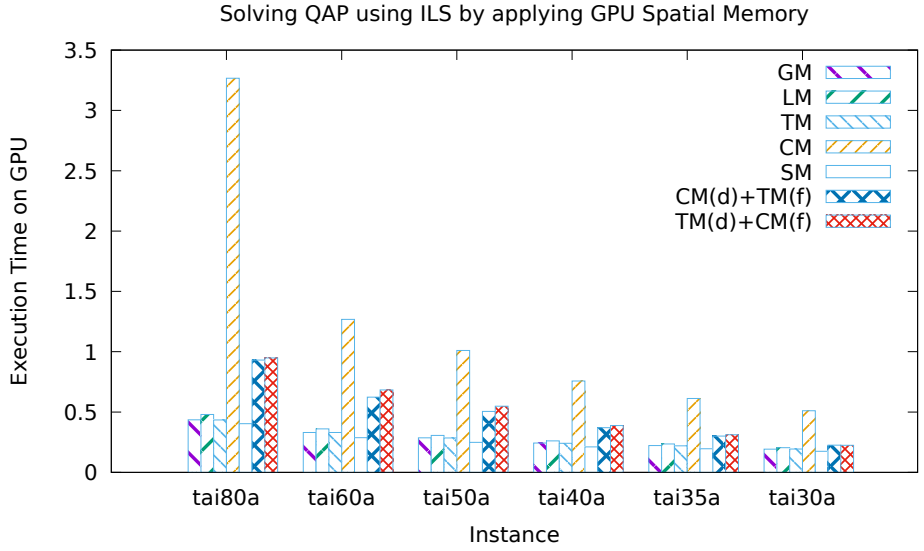

FIGURE 5.2: Execution Time (seconds) on GPU

tai80a speedup on GPU for distance matrix on constant memory and flow matrix on texture memory is 18.2, which is greater than for only constant memory 5.19 and lesser than for only texture memory 39. We compared the execution time on GPU with different memory as illustrated in Figure 5.2, and it shows that as the size of the instance grows, the execution time on the GPU also grows. We compared the speedup on GPU concerning CPU is shown in Figure 5.3, here in this figure, for instance, *tai80a* we got the highest speedup on GPU using shared memory.

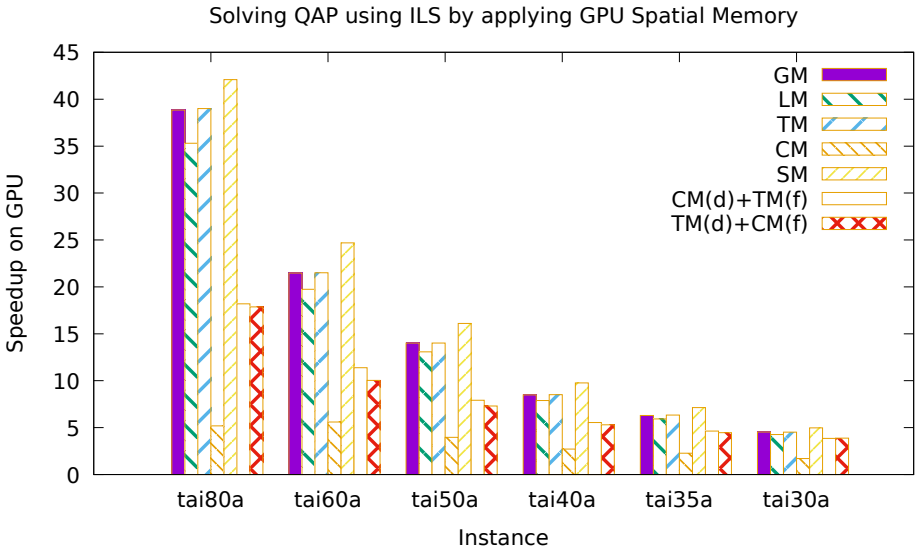

FIGURE 5.3: Speedup on GPU with respect to CPU

5.6 Summary

In this chapter, we used GPU spatial memory and compared the execution time and speedup on GPU. We have found that by adequately utilizing GPU memory, we can reduce the execution time and increase the speedup on GPU. Here, shared memory performs better for instances of size up to 80 when we mix the constant and texture memory, the performance increases from using only constant memory while decreasing from using only texture memory.

Chapter 6

Analysis of Meta-heuristics for Traveling Salesman Problem in Accelerated Systems

In this chapter, we discuss another optimization problem TSP similar to QAP. We again used all six meta-heuristics: ILS, SA, GA, PSO, CSA, and TS in accelerated systems. We also did the performance analysis of all meta-heuristics and mapped each parallel section to the appropriate architecture. Finally, we compared all meta-heuristics and calculated the speedup on GPU.

6.1 Introduction

In this work, we analyze the performance of meta-heuristics for solving TSP. Here we considered both types of meta-heuristics: single solution-based (s-type) and population-based (p-type) meta-heuristics. We have taken s-type as ILS, SA, and CSA in this study and p-type as GA, PSO, and TS. We also compared the optimum tour and execution time between CPU, Pthread, and GPU and computed the speedup on GPU. Here we have taken instances from TSPLIB [12] symmetric instances.

This chapter is organized as follows: literature review is discussed in section 6.2, brief description of TSP is described in section 6.3, method for generating of neighbor solution is discussed in section 6.4. Section 6.5 shows the accelerated system used for TSP, various meta-heuristics implementation are illustrated in section 6.6. Section 6.7 described the experimental results and analysis. Finally we summarized this chapter in section 6.8.

6.2 Literature Review

TSP belongs to NP-complete class problems [66]. It has many applications in engineering problems such as electronic circuit design, network optimization, vehicle routing, etc. [67, 68]. Grefenstette et al. [69] used GA to solve TSP, and many transform operators are used to improving a solution in each generation. In paper [70], Scholz et al. described the historical development of GA for TSP. In paper [71], Uchida et al. used ant colony optimization for solving TSP on GPU. He used GPU coalesced access memory to optimize the results further. Semin Kang and others [72] used GA on GPU for solving TSP by using a constructive crossover operator. In paper [73], Fosin et al. used local search operators on GPU for symmetric TSP. ILS is parallelized on GPU using 2-opt and 3-opt local search operators [74]. In paper [75], Bali et al. used PSO to accelerate on GPU for TSP. In paper [76], Yelmewad et al. used a parallel iterative hill climbing algorithm for solving TSP large-size instances on GPU. The authors also demonstrated the impact of storing data in different GPU memory and showed improvement using texture memory in place of global memory and speedup 181 on GPU compared to sequential part on CPU. Ant colony optimization (ACO) meta-heuristic is parallelized on GPU [77], Menezes et al. compare coarse- grain and fine-grain parallel ACO. In paper [78], Abbasi et al. used GA for solving TSP on multi-core and many-core systems. Here, the authors used three different GPU kernels to run concurrently for GA operators and showed the highest speedup of 58.35 on GPU. Another work of TSP based on GPU is done by Qiao et al. [79], where authors used multiple k-opt heuristics to parallelize on GPU and show speedup on GPU.

Now researchers are looking for machine learning approach [80, 81] to select the best meta-heuristics for solving TSP, in this context Huerta et al. [82], proposed a any-time automatic algorithm selection methods for TSP.

6.3 Traveling Salesman Problem

Given n cities and a distance matrix $d_{n,n}$, where each element d_{ij} represents the distance between the cities i and j, then a problem is to find a tour that minimizes the total distance. A tour visits each city exactly once.

6.4 Generating Neighbor Solution

We used the adjacent pairwise exchange method in this work to generate neighbor solutions. If a solution has n cities then $n \times (n-1)/2$ neighbor solutions are generated.

6.4.1 Incremental Solution Evaluation

Instead of passing the whole solution, we passed the only index position to swap the elements of a solution; with that help, we got more speedup compared to giving the complete solution every time. Due to the swapping of two components, four positions are affected in a complete solution: the distance between swapped elements and their neighbor elements. Let us take an example of a complete solution with two positions r and s out of the full size N. If in a solution, position r and s are to be swapped, then the changes in the solution cost are represented as $\Delta(x)$, which is as follows-

FIGURE 6.1: Solution before swap

FIGURE 6.2: Solution after swap

$$\Delta(x) = \Big\{ + \Big(d(s, r-1) + d(s, r+1)\Big) - \Big(d(r, r-1) + d(r, r+1)\Big) + \Big(d(r, s-1) + d(r, s+1)\Big) - \Big(d(s, s-1) + d(s, s+1)\Big)\Big\} \tag{6.1}$$

Where $d(x_1, x_2)$ denotes the distance between points x_1 and x_2. In Equation 6.1, if the index $r = 1$, then we consider $r - 1 = N$ and if $s = N$, then we consider $s + 1 = 1$, where N is the size of the solution or position of the last element in a solution.

6.5 Accelerated System for TSP

In this work, we used Nvidia GeForce GTX 980 Ti GPU with 2816 CUDA cores and 6 GB DDR5 memory, 22 SM, and 64 warps per SM. Additionally, We installed the CUDA toolkit 8.0 on this device.

6.6 Meta-heuristics Implementation

We analyzed the solution of TSP by applying both types of meta-heuristics (single solution and population-based meta-heuristics) on the serial machine, *POSIX* thread (Pthread) and GPU. The general approach for solving TSP is for all meta-heuristics is as first we take the input as TSPLIB instances in the form of city number, x-coordinate, y-coordinate (*tsp matrix*). Then generated a random initial solution (*tour*) of TSP; from that initial solution, we generated all possible neighbor solutions and evaluated them using the evaluation function. If it improves the

objective of the solution, then replace the current solution with a new solution. This process will continue until it reaches the stopping condition.

We have taken TSPLIB [12] instances from online TSP library. Here we considered only symmetric TSP instances from size 51 to 1379 of different instances. In this work, we used 25 different instances to analyze the performance of different meta-heuristics.

6.6.1 ILS Implementation

6.6.1.1 ILS on Serial Machine

Using ILS, we generated neighbor solutions from the initial solution with the adjacent pairwise exchange permutation method. We started from the first position and then changed the position of the elements to the next nearby position. After generating the new neighbor solution, we assessed it using the evaluation function. We compared the assignment cost with the current assignment cost; if it was less than the current one, we chose the new solution as the current one. Thereby, we evaluated all the $\frac{n\times(n-1)}{2}$ neighbor solutions. At this stage, local optima were reached; thus, for the next iteration, we made the minimal local solution the initial solution and repeated all the above-mentioned steps. This process continued until the stopping criteria were reached. To avoid being stuck in local optima or unable to get the best global minimum solution, we took 500 random initial solutions, with each initial solution being iterated 10 times. Finally, we observed the objective value and the execution time (in seconds) to execute the whole program as in Table 6.1. In this table, the percentage deviation (D) is calculated using Equation 6.2. We considered 20 TSPLIB symmetric instances. Here in Table 6.1, we can observe that some instances have the highest deviation, like instance *lin318* has the 1038 deviation percentage because it may require more exploration of initial solutions and number of iterations. Here instance *a280* shows the highest

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
a280	927	22.31	927	4.22	5.29	922	0.92	24.17
berlin52	13	0.54	13	0.15	3.55	13	0.11	4.81
bier127	313	3.23	313	0.87	3.73	313	0.28	11.48
eil51	142	0.52	142	0.14	3.69	120	0.11	4.65
eil76	210	1.17	210	0.31	3.75	205	0.16	7.47
gil262	812	13.88	812	3.68	3.77	811	0.82	17.01
kroa100	368	2.01	368	0.53	3.79	367	0.21	9.65
kroa150	559	4.52	559	1.24	3.65	559	0.35	13.08
kroa200	705	8.05	705	2.15	3.74	705	0.53	15.24
krob100	371	2.02	371	0.53	3.8	371	0.21	9.53
krob150	541	4.53	541	1.19	3.79	541	0.35	12.86
krob200	712	8.07	712	2.16	3.74	711	0.53	15.15
kroc100	403	2.01	403	0.54	3.69	402	0.21	9.63
krod100	379	2	379	0.54	3.71	379	0.2	9.97
kroe100	377	2	377	0.53	3.79	376	0.21	9.53
lin105	428	2.22	428	0.59	3.79	427	0.23	9.52
lin318	1038	20.38	1038	5.42	3.76	1037	1.16	17.5
pr107	595	2.3	595	0.61	3.75	594	0.23	9.91
pr124	668	3.09	668	0.84	3.67	668	0.27	11.36
pr136	473	3.73	473	0.99	3.76	473	0.31	12.08

TABLE 6.1: Percentage deviation and exec. time of TSP using ILS on CPU, Pthread, and GPU

speedup 24.17 on GPU.

$$Deviation\% = \frac{(Objective\,Value - TSPLIB\,Cost)}{TSPLIB\,Cost} \times 100. \tag{6.2}$$

6.6.1.2 ILS using Pthread

In this meta-heuristic for implementing on pthread, instead of executing all initial solutions at a time, we assigned all the initial solutions to several processors so each could be utilized and get an equal number of the initial solutions. Then, each part was assigned to different processors so that all parts could be run in parallel. Finally, Table 6.1 shows the optimal cost and execution time using pthread. Here deviation of all instances is the same as on CPU; only execution time is different. Here for instance *a280*, the speedup on Pthread is 5.29.

6.6.1.3 ILS on GPU

In general, to run a program on GPU, three main steps have to be followed: first, the data input is copied from CPU (host) to GPU (device); second, the program is executed on GPU; and last, the result is sent back from GPU to CPU.

First, we set the kernel's grid size and block size to generate several threads to achieve the best performance with respect to executing the ILS algorithm on GPU. For the implementation, we generated random initial solutions on CPU, and from each initial solution, we generated neighbor solutions that were evaluated on GPU. In this study, we took 500 initial solutions, with each initial solution being iterated 10 times to get the best possible optimal solution. All the initial solutions were run in parallel and called the GPU, which also ran in parallel to execute the evaluation cost function of neighbor solutions. The percentage deviation and execution time on GPU are reported in Table 6.1, demonstrating that the execution time on GPU is less than that on CPU. For example, the instance *a280* has execution time on CPU is 22.31, on Pthread is 4.22 while on GPU is 0.92, which shows 24.17 faster on GPU. For instance *berlin52*, we got the minimum deviation 13% among all other instances.

6.6.2 SA Implementation

6.6.2.1 SA on Serial Machine

Using SA to execute the evaluation function, first, we generated all possible neighbor solutions. Unlike executing the ILS for each neighbor solution to find the best solution, we calculated the assignment cost of all the neighbor solutions simultaneously and then stored the values for each solution. To find the best solution among all neighbor solutions, we applied SA. We fixed the main parameters of SA to be as follows: initial temperature: $10,000$, cooling rate: 0.9999, and absolute temperature: 0.00001. First, the algorithm starts with an initial temperature, and at every iteration, the temperature is reduced to [current temperature$\times$cooling rate],

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
a280	1100	20.54	1104	19.99	1.03	1101	4.22	4.87
berlin52	102	0.71	118	0.76	0.94	112	0.32	2.23
bier127	357	4.51	347	4.96	0.91	348	1.26	3.59
eil51	212	0.68	211	0.67	1.01	202	0.3	2.27
eil76	283	1.5	287	1.51	1	288	0.47	3.17
gil262	912	18.02	930	17.85	1.01	924	3.7	4.87
kroa100	568	2.72	581	2.83	0.96	587	0.77	3.52
kroa150	723	6.13	735	5.97	1.03	739	1.49	4.11
kroa200	886	11.17	919	11.53	0.97	939	2.5	4.47
krob100	534	2.7	516	2.8	0.97	524	0.77	3.52
krob150	746	6.1	752	6.61	0.92	732	1.51	4.05
krob200	884	10.9	861	12	0.91	912	2.49	4.38
kroc100	574	2.74	592	3.1	0.88	565	0.78	3.52
krod100	529	2.7	524	3.14	0.86	533	0.77	3.51
kroe100	545	2.72	529	3.09	0.88	546	0.77	3.54
lin105	623	2.95	597	3.21	0.92	618	0.81	3.65
lin318	1186	27.63	1192	31.62	0.87	1183	6	4.61
nrw1379	2299	569.93	2300	518.83	1.1	2315	101.94	5.59
pr1002	2289	284.57	2271	405.17	0.7	2275	70.7	4.03
pr107	908	3.24	967	3.95	0.82	924	1.01	3.22
pr124	873	4.33	881	5.2	0.83	874	1.27	3.4
pr136	626	5.22	630	6.46	0.81	621	1.51	3.45

TABLE 6.2: Percentage deviation and exec. time of TSP using SA on CPU, Pthread, and GPU

making it the current temperature for the next iteration. This process continues until the current temperature reaches the absolute temperature. The deviation and execution time are reported in Table 6.2, where we can see that as the size of instances increases, the execution time increases more rapidly.

6.6.2.2 SA using Pthread

In this study, we applied SA to find the best solution or local optima for every initial solution. Therefore, instead of running all of the initial solutions sequentially on the CPU, we assigned them to an equal number of processors. The optimal solution was found and compared on each processor to get the globally optimal

solution. The percentage deviation and total execution time are reported in Table 6.2. In Pthread implementation, processor utilization performs better than a serial machine, with each part running in parallel. In Table 6.2, we can see that its percentage deviation is not the same as that of the CPU because of the random generation of solutions. Additionally, the execution time on Pthread is reduced, and because of overhead for some of the instances like *lin318* it gives the worst than CPU.

6.6.2.3 SA on GPU

For the implementation of SA on GPU, first, we generated neighbor solutions from the initial solutions and evaluated them on GPU. Then, the entire cost of the neighbor solutions was transferred from GPU to CPU. Afterward, we performed SA on the CPU to find the optimal solution after GPU evaluated all neighbor solutions in parallel. The execution results are reported in Table 6.2. In this table, for instance, *nrw1379*, the execution time on CPU is 569.93 seconds while on GPU is 101.94, which shows the 5.59 times fast on GPU as compared to CPU.

6.6.3 GA Implementation

6.6.3.1 GA on Serial Machine

GA starts with a random initial population of solutions, where the neighbor solutions are generated with the help of two operators, namely, crossover and mutation. The crossover operator selects two random solutions from the population of solutions. Several methods for the random selection of individuals in this algorithm, of which the roulette wheel and tournament methods are the most popular. In this study, we used the tournament method. In this method, first, a few solutions are randomly selected. Then, among them, the best solution (with the minor assignment cost) is selected. When two solutions are selected, the crossover operator is employed to generate the new offspring and then store it in the new population.

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
a280	1119	7.35	1108	3.51	2.1	1114	5.98	1.23
berlin52	136	1.46	93	0.76	1.91	136	1.3	1.12
bier127	378	3.34	387	1.67	2	378	2.74	1.22
eil51	238	1.65	221	0.75	2.18	233	1.29	1.28
eil76	312	2.09	303	1.07	1.97	299	1.76	1.19
gil262	939	8.67	939	3.33	2.61	935	5.17	1.68
kroa100	609	2.76	583	1.37	2.01	609	2.2	1.25
kroa150	765	4.05	750	1.96	2.06	764	3.08	1.32
kroa200	919	5.39	945	2.6	2.07	919	4.01	1.34
krob100	559	2.75	542	1.35	2.03	559	2.19	1.26
krob150	763	4.67	759	1.94	2.41	762	3.18	1.47
krob200	885	5.76	926	2.59	2.23	884	4.27	1.35
kroc100	574	2.74	578	1.36	2.02	574	2.18	1.26
krod100	544	3.6	551	1.35	2.67	544	2.2	1.63
kroe100	583	2.77	562	1.37	2.02	582	2.16	1.28
lin105	614	3.69	616	1.4	2.62	614	2.27	1.62
lin318	1205	11.05	1169	4.23	2.61	1204	6.4	1.73
nrw1379	2324	39.67	2317	365.03	0.11	2323	34.17	1.16
pr1002	2292	25.15	2294	151.91	0.17	2292	21.8	1.15
pr107	989	2.93	972	1.46	2.01	989	2.37	1.24
pr124	942	4.4	936	1.67	2.64	942	2.69	1.63
pr136	656	3.69	646	1.79	2.06	656	2.86	1.29

TABLE 6.3: Percentage deviation and exec. time of TSP using GA on CPU, Pthread, and GPU

In this study, we used one-point crossover and then applied the mutation operator to the newly generated solution. For mutation, we changed the position of the elements and stored them in the new population. In addition, we fixed the number of initial solutions as 500 and the number of iterations as 10. From the 500 initial solutions, we generated 25000 neighbor solutions using the crossover and mutation operators. In each iteration, the best population of solutions was selected (i.e., 500), becoming the current population, with this process continuing until the termination condition is reached. The percentage deviation and execution time of GA are reported in Table 6.3, which shows that as the size of the instance increases, the execution time also increases.

6.6.3.2 GA using Pthread

In GA, the generated random initial solutions are called the population, with the neighbor solutions being generated with the help of genetic operators, namely, crossover and mutation. Afterward, the neighbor solutions are evaluated in parallel; among them, the best population is chosen for the next generation. In this study, the neighbor solutions are assigned to an equal number of processors, with the number of neighbor solutions taken being 25000 and the initial population being 500. Each processor evaluates the solutions, stores their costs, and then merges all solutions, out of which we chose the best 500 solutions for the next generation or iteration. Table 6.3 shows the percentage deviation and execution time using Pthread. Further, it shows that as the size of instances increases, the execution time increases in proportion to the number of processors used (e.g., *lin318*).

6.6.3.3 GA on GPU

Using GA, we generated the random initial population of solutions; then, we generated the offspring using the crossover and mutation operators for the next generation and iteration. In GPU, we evaluated the solutions in parallel until the stopping condition was reached, then the results were sent back from GPU to CPU. The percentage deviation and execution time to run on GPU are reported in Table 6.3. In this table, for instance, *berlin52*, the execution time on CPU is 1.46 seconds while on GPU is 1.3 seconds, which shows the speedup on GPU.

6.6.4 PSO Implementation

6.6.4.1 PSO on Serial Machine

PSO starts with random initial solutions or a population of particles, with each particle having a random initial velocity. Every particle has its personal best based on its experience or history and global best for the whole group of particles. In every iteration, each particle updates its velocity according to Equation 2.3

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
a280	1083	81.88	1071	21.92	3.74	1077	16.67	4.91
berlin52	109	0.55	114	0.19	2.92	113	0.15	3.68
bier127	361	7.67	370	2.08	3.69	383	1.41	5.44
eil51	203	0.51	205	0.17	3.01	199	0.14	3.67
eil76	286	1.67	278	0.48	3.47	274	0.36	4.67
gil262	912	67.14	895	17.74	3.78	902	13.64	4.92
kroa100	546	3.79	546	1.03	3.66	556	0.73	5.18
kroa150	724	12.62	725	3.41	3.7	711	2.33	5.42
kroa200	881	30.02	902	8	3.75	908	5.94	5.06
krob100	517	3.76	501	1.05	3.57	521	0.73	5.15
krob150	730	12.61	727	3.41	3.69	730	2.29	5.5
krob200	859	29.88	890	8.01	3.73	879	5.99	4.99
kroc100	552	3.77	563	1.05	3.6	566	0.73	5.14
krod100	520	3.76	484	1.04	3.6	494	0.73	5.16
kroe100	532	3.77	529	1.06	3.55	527	0.73	5.16
lin105	588	4.35	598	1.12	3.87	596	0.83	5.23
lin318	1174	119.73	1161	32.14	3.73	1168	24.25	4.94
pr107	937	4.62	921	1.26	3.66	962	0.87	5.29
pr124	870	7.13	841	1.94	3.68	902	1.31	5.43
pr136	599	9.43	620	2.55	3.69	634	1.71	5.51

TABLE 6.4: Percentage deviation and exec. time of TSP using PSO on CPU, Pthread, and GPU

and position according to Equation 2.4. As the iteration continues, each particle converges toward the optimal solution. In this study, velocity was measured in terms of the number of swaps of positions inside a solution. In Equation 2.3, we have taken the inertia factor (ω) 0.9, and two constants c_1 and c_2 as 2. The results of PSO are reported in Table 6.4, showing that for instance of equal size like *kroc100, krod100, kroe100* execution time is almost same. Moreover, we used 500 initial solutions, with each solution being iterated 10 times to obtain the optimal solution.

6.6.4.2 PSO using Pthread

In this meta-heuristic for implementing on pthread, an initial population of solutions is assigned to an equal number of processors. As the algorithm proceeds,

every processor reaches the optimal solution. After the termination, all the optimal solutions for the processors are combined, and among them, the best solution is observed. The percentage deviation and execution time are reported in Table 6.4, with the Pthread execution time being significantly reduced as compared to CPU.

6.6.4.3 PSO on GPU

In PSO, we observed the random position and velocity of the whole swarm, which is generated on the CPU. The particle's position and velocity were used to generate the next position of the particles, which was evaluated in parallel on GPU. Every member of the swarm updates its velocity based on the position from the local optimum. After reaching the stopping criteria, the final global optimum values were copied from GPU to CPU. The percentage deviation and execution time on GPU are reported in Table 6.4. Here from this table, for instance, *pr136* the speedup on GPU is 5.51.

6.6.5 CSA Implementation

6.6.5.1 CSA on Serial Machine

In CSA, first, we fixed the initial parameters such as the flight length and probability of awareness. When the probability of awareness decreases, then it searches in the local region (exploitation-oriented), whereas when it increases, it explores the search space (exploration-oriented). Initially, for the serial machine, we generated a fixed number of initial solutions as the size of the input, with each initial solution generating a neighbor solution using the adjacent pairwise exchange method. Among these neighbor solutions, we found the best one, which was used to initialize the memory of each crow for each initial solution. As the iteration increases, each crow updates its memory until the termination condition is reached. Among the memories of all crows, we found the best one giving the optimal solution for the TSP, taking flight length as 2 and probability of awareness as 0.15.

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
a280	1109	4.78	1051	27.44	0.17	1107	3.94	1.21
berlin52	92	0.37	95	1.48	0.25	101	0.33	1.14
bier127	361	1.15	349	4.99	0.23	407	1.92	0.6
eil51	218	0.29	198	1.61	0.18	185	0.58	0.51
eil76	282	0.5	275	2.65	0.19	287	1.06	0.48
gil262	912	3.33	875	22.85	0.15	904	2.79	1.19
kroa100	570	1.55	532	5.13	0.3	550	0.73	2.14
kroa150	584	1.49	693	8.78	0.17	720	1.68	0.89
kroa200	931	3.59	866	14.27	0.25	919	3.19	1.12
krob100	541	0.72	488	3.62	0.2	509	1.29	0.56
krob150	725	1.99	695	8.51	0.23	751	3.29	0.6
krob200	868	2.48	850	13.68	0.18	913	2.24	1.11
kroc100	576	1.64	525	3.43	0.48	559	1.14	1.43
krod100	540	0.5	500	3.96	0.13	534	0.61	0.81
kroe100	558	1.13	509	3.71	0.3	562	0.98	1.15
lin105	476	1.11	586	4.79	0.23	630	1.63	0.68
lin318	1177	9.29	1149	35.23	0.26	1194	2.91	3.19
pr1002	2238	92.06	2209	327.59	0.28	2273	20.57	4.48
pr107	951	92.06	846	4.33	21.26	959	1.82	50.53
pr124	876	1.2	866	6.56	0.18	950	0.76	1.58
pr136	637	0.77	597	7.68	0.1	648	2.4	0.32

TABLE 6.5: Percentage deviation and exec. time of TSP using CSA on CPU, Pthread, and GPU

6.6.5.2 CSA using Pthread

To implement on Pthread, first, we divided the initial solutions among processors, with each processor running the maximum number of iterations (initially fixed) and updating the memory of each crow. All processors run in parallel, while inside each processor, this algorithm runs serially. After completing the execution of all processors, we found the best-updated memory among all crows that gives our optimal solutions, with the results represented in Table 6.5.

6.6.5.3 CSA on GPU

To implement CSA on GPU, first, we fixed the initial parameters. Then, we calculated the cost of the initial solutions on the CPU, and from each initial

solution, we generated the neighbor solutions using the adjacent pairwise exchange method. Afterward, we evaluated the cost of the neighbor solutions on GPU. From each initial solution, we found the best possible neighbor solution cost that is set to the memories of each corresponding crow (for the initial solution). Consequently, we run the CSA on the CPU to find the next positions of the crows, evaluate their cost, and compare them to the solution stored in their memory. If the solution gave the best result, then the corresponding crow's memory was updated until the termination criteria were reached. Finally, among the memories of all crows, we found the best solution cost, which is reported in Table 6.5. In Table 6.5, the percentage deviation and execution time are different on CPU, Pthread, and GPU. Here for instance *pr107* the speedup on GPU is 50.53.

6.6.6 TS Implementation

6.6.6.1 TS on Serial Machine

In TS, first, we fixed the size of the tabulist as the size of the instance. Then, we generated the fixed number of (size of tabulist) random initial solutions and evaluated their cost, and stored them in tabulist. In each iteration, we generated the neighbor solution through an adjacent pair-wise exchange method and evaluated their cost; if it improves from the current solution, then we updated the *tabulist*. This procedure will continue until it reaches the terminating condition and finds the best optimal solution from tabulist. To avoid being stuck in local minima, we implemented a diversification operator suggested by James et al. [50] to generate a new solution. We fixed the maximum number of failures as the size of the instance. In CPU, we fixed the number of iterations as 10. In Table 6.6 execution time and percentage deviation are reported.

6.6.6.2 TS using Pthread

To implement on Pthread, we first fixed tabulist size as the instance's size, then generated random initial solutions, evaluated their cost, and stored them in the

Instance	CPU		Pthread			GPU		
	D (%)	ET time	D (%)	ET time	S	D (%)	ET time	S
a280	1055	223.91	969	99.7	2.25	1049	156.56	1.43
berlin52	-49	0.34	-62	0.12	2.75	-49	0.21	1.62
bier127	227	10.86	239	4.76	2.28	227	7.46	1.46
eil51	29	0.32	-9	0.13	2.55	27	0.21	1.55
eil76	132	1.4	136	0.55	2.54	129	0.93	1.5
gil262	898	173.96	806	75.82	2.29	894	120.24	1.45
kroa100	422	4.2	360	1.85	2.28	422	2.87	1.47
kroa150	600	20.08	550	9.08	2.21	600	14.18	1.42
kroa200	861	66.03	754	27.57	2.4	861	43.52	1.52
krob100	387	4.43	314	1.7	2.6	387	2.84	1.56
krob150	667	20.56	553	9.41	2.18	666	14.04	1.46
krob200	812	60.59	734	29.47	2.06	812	43.81	1.38
kroc100	404	4.45	343	1.78	2.5	404	2.86	1.56
krod100	395	4.49	355	1.82	2.46	395	2.86	1.57
kroe100	401	4.36	366	1.73	2.52	401	2.87	1.52
lin105	486	5.47	358	2.19	2.5	486	3.45	1.58
lin318	1195	381.31	1076	188.33	2.02	1195	268.05	1.42
pr107	751	5.37	624	2.22	2.42	751	3.7	1.45
pr124	761	9.79	676	4.41	2.22	761	6.64	1.47
pr136	548	13.17	478	6.28	2.1	548	9.54	1.38

TABLE 6.6: Percentage deviation and exec. time of TSP using TS on CPU, Pthread, and GPU

tabulist. We call pthread, and every thread gets the tabulist and generates a neighbor solution; if it improves the current solution, then update the tabulist. When Pthread join, then merge all the thread tabulist, and from that, we get the best optimal solution, which is reported in Table 6.6.

6.6.6.3 TS on GPU

To implement TS on GPU, first, we fixed the GPU parameters and then transferred the input data from the CPU to GPU. In GPU, we fixed the number of iterations as 10, and the size of the tabulist is the size of the instance. In each iteration of TS, we transfer the tabulist from CPU to GPU. On GPU, we generate and evaluate the neighbor solution, update the tabulist with the best optimal solution, and then the updated tabulist is transferred back from GPU to CPU. This process

Instance	ILS		SA		GA		PSO		CSA		TS	
	D (%)	S	D (%)	S	D (%)	S	D (%)	S	D (%)	S	D (%)	S
a280	922	24.17	1101	4.87	1114	1.23	1077	4.91	1107	1.21	1049	1.43
berlin52	13	4.81	112	2.23	136	1.12	113	3.68	101	1.14	-49	1.62
bier127	313	11.48	348	3.59	378	1.22	383	5.44	407	0.6	227	1.46
eil51	120	4.65	202	2.27	233	1.28	199	3.67	185	0.51	27	1.55
eil76	205	7.47	288	3.17	299	1.19	274	4.67	287	0.48	129	1.5
gil262	811	17.01	924	4.87	935	1.68	902	4.92	904	1.19	894	1.45
kroa100	367	9.65	587	3.52	609	1.25	556	5.18	550	2.14	422	1.47
kroa150	559	13.08	739	4.11	764	1.32	711	5.42	720	0.89	600	1.42
kroa200	705	15.24	939	4.47	919	1.34	908	5.06	919	1.12	861	1.52
krob100	371	9.53	524	3.52	559	1.26	521	5.15	509	0.56	387	1.56
krob150	541	12.86	732	4.05	762	1.47	730	5.5	751	0.6	666	1.46
krob200	711	15.15	912	4.38	884	1.35	879	4.99	913	1.11	812	1.38
kroc100	402	9.63	565	3.52	574	1.26	566	5.14	559	1.43	404	1.56
krod100	379	9.97	533	3.51	544	1.63	494	5.16	534	0.81	395	1.57
kroe100	376	9.53	546	3.54	582	1.28	527	5.16	562	1.15	401	1.52
lin105	427	9.52	618	3.65	614	1.62	596	5.23	630	0.68	486	1.58
lin318	1037	17.5	1183	4.61	1204	1.73	1168	4.94	1194	3.19	1195	1.42
pr107	594	9.91	924	3.22	989	1.24	962	5.29	959	50.53	751	1.45
pr124	668	11.36	874	3.4	942	1.63	902	5.43	950	1.58	761	1.47
pr136	473	12.08	621	3.45	656	1.29	634	5.51	648	0.32	548	1.38

TABLE 6.7: percentage deviation and speedup on GPU for all meta-heuristics for TSP

continues until it reaches the stopping condition. Finally, the best optimal solution is found from tabulist, and results are noted in Table 6.6. Here in Table 6.6, we can see that, for instance, *berlin52* the optimum tour is under-performed than the TSPLIB optimum tour, and the speedup on GPU is 1.62.

6.7 Experimental Results and Analysis

We compared all the meta-heuristics for each TSP symmetric instance on GPU and calculated the speedup on GPU concerning the CPU, which is noted in Table 6.7. Here from this table, we can observe that for input instance *pr107*, CSA is giving the highest speedup 50.53, and ILS offers the best speedup for all other instances on GPU.

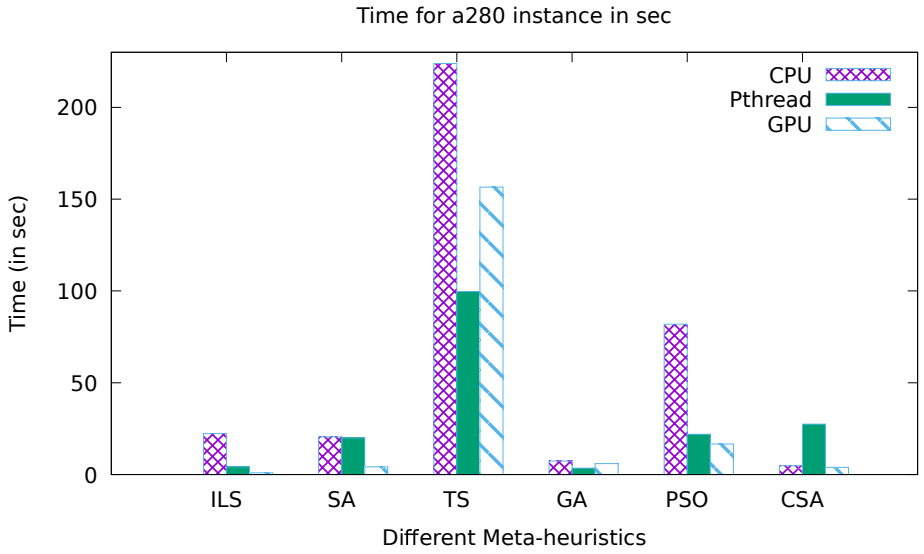

FIGURE 6.3: Exec. time on CPU, pthread, and GPU for instance a280

Figure 6.3, illustrates the execution time (in seconds) for all above meta-heuristics on CPU, pthread, and GPU for one TSPLIB instance *a280*. Here we observed that the TS meta-heuristic took the highest time on CPU, pthread, and GPU, while the ILS meta-heuristic took less time on GPU and performed best for *a280* instance.

According to Figure 6.4, when we compare the speedup of GPU for all the meta-heuristics and all the considered TSPLIB instances, the GPU provides the highest speedup of 50.53 for input instance *pr107*, and ILS performs best in most of the remaining input instances. We have illustrated the speedup on GPU for one meta-heuristic PSO in Figure 6.5; here, we observe that for all the considered input instances, the speedup on GPU has significantly less variation.

6.8 Summary

We used six meta-heuristics: ILS, SA, GA, PSO, CSA, and TS for solving TSP and implemented on CPU, Pthread, and GPU. We have considered 21 symmetric instances from TSPLIB. We observe that the CSA meta-heuristic records the highest speedup 50.53 for instance *pr107* and ILS records second highest speedup

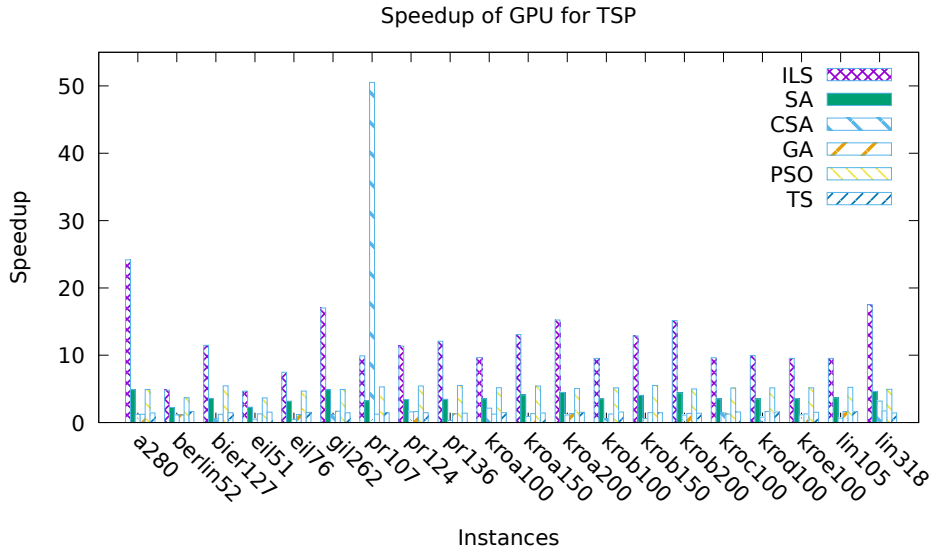

FIGURE 6.4: Speedup on GPU

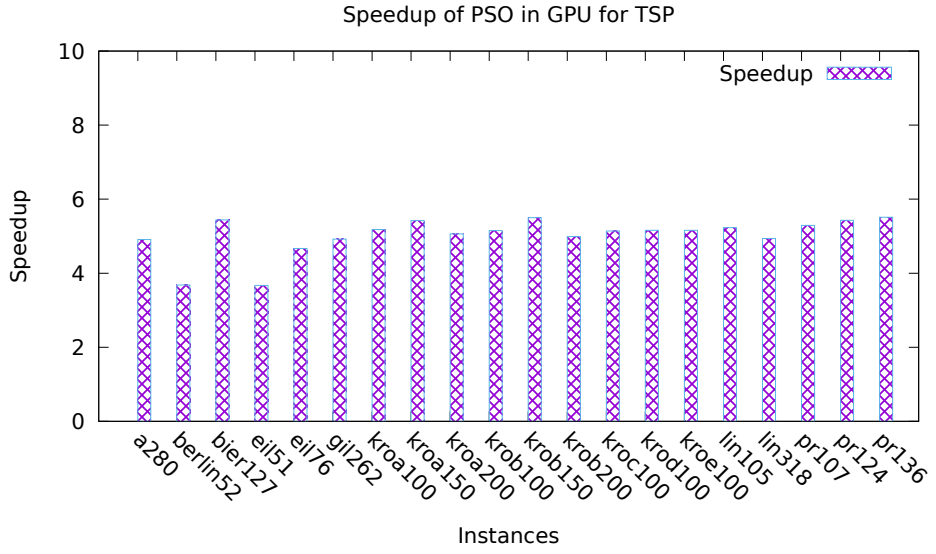

FIGURE 6.5: Speedup on GPU for PSO meta-heuristic

24.17 for *a280* input instance. The PSO meta-heuristics records speedup 5.51 on GPU. We also observe that the instance with similar nature like *kroa100, kroa150, and kroa200* shows the speedup increases as their sizes increases for most of the meta-heuristics ILS, SA, and GA while for others PSO, CSA, and TS gives the speedup with slightly variation.

Chapter 7

Analysis of Meta-heuristics for Permuted Perceptron Problem in Accelerated Systems

In this chapter, we proposed an efficient method for permuted perceptron problems using simulated annealing meta-heuristic in massively parallel many-core architecture graphics processing units. We parallelize the simulated annealing meta-heuristic for PPP in many-core architecture GPU. We discussed the performance and statistical analysis of the SA meta-heuristic and calculated the speedup on GPU concerning multi-core CPU.

7.1 Introduction

Nowadays, solving complex and real-world problems using meta-heuristics is ever-increasing; however, it gives the approximate solution, but it takes a huge amount of time for large instances. The execution time can be reduced with modern massively parallel accelerated systems. The permuted perceptron problem (PPP) introduced by Pointcheval [3] belongs to the NP-complete class problem. This problem is well suited for smart cards.

Many meta-heuristics such as genetic algorithm, tabu search, and ant colony optimization have been widely used in the crypt-analysis of classical cipher [83] and vigenere cipher [84]. In this paper, we analyzed the performance of PPP using the simulated annealing (SA) meta-heuristic in accelerated systems like- multi-core processors and graphics processing units (GPUs).

The chapter is organized as follows: literature review is described in section 7.2, the definition of PPP is described section 7.3. The method for neighbor solutions generation is described in Section 7.4. The overview of accelerated systems is discussed in section 7.5. The mapping of SA meta-heuristic to architecture is explained in section 7.6. The experimental results are reported in Section 7.7. Section 7.8 illustrates the performance analysis of SA meta-heuristic. Finally, we summarize this chapter in section 7.9.

7.2 Literature review

In [3] Pointcheval introduced the identification scheme based on permuted perceptron problem, which is derived from the simpler one but still NP-complete problem known as perceptron problem (PP). In this paper, the author suggested that the matrix size (m, $n = m + 16$) is harder to solve for an attack on a smart card application. Here authors used SA to find the number of solutions and time for PPP.

Knudsen and Meier [85] describe the attack on Pointcheval identification scheme by carrying out a set of runs. The authors noted the result for carrying out a new set of runs and fixed those elements where there is complete agreement; By repeating runs, if the same values are obtained for particular bits, the authors assume that those bits are actually set correctly. Through this approach, authors solve PP instances 180 times faster than Pointcheval for instance size $(151, 167)$ but no upper bound given on sizes achievable. The authors also used a new cost function to solve PPP with histogram punishment. In paper [86], fault injection and timing analysis of PPP is introduced. Here fault injection is used in the

SA meta-heuristic where terminating criteria are fixed as the maximum number of failures to not accepting any solutions. The authors also modified the cost function by inducing some constant in the cost function.

VanLuong [87] has taken as a case study of PPP of a local search algorithm on GPU. In this paper, the authors generated neighbor solutions from the initial solution by binary encoding technique and solved PPP using the tabu search meta-heuristic. The authors used one hamming distance to generate neighbor solutions from an initial solution, executed them on GPU, and compared the results between GPU and GPU with texture memory. He also used two hamming distances to generate neighbor solutions from an initial solution and again compared GPU and GPU texture memory. He found that GPU with texture memory gives 40 times faster than GPU.

7.3 Permuted Perceptron Problem (PPP)

A new identification scheme based on the perceptron problem was introduced by Pointcheval [3], which is used in resource-constrained devices like smart cards. Authors used ϵ-vector and ϵ-matrix to define perceptron and permuted perceptron problems. An ϵ-vector contains all the elements of a vector that are either $+1$ or -1, similarly an ϵ-matrix contains elements only $+1$ and -1. PP can de defined as, given an input of an ϵ-matrix I of size $m \times n$, then to find an ϵ-vector Z of size n such that $IZ \geq 0$. In a similar way, PPP can be defined as, given an input of an ϵ-matrix I of size $m \times n$, and a multi-set S of non-positive integers of size m, then to find an ϵ-vector Z of size n such that-

$$\{\{(IZ)_j j = \{1, \ldots, m\}\}\} = S. \tag{7.1}$$

7.4 Generating Neighbor Solutions

We generated many neighbor solutions from each initial solution using the binary encoding method in this work. The binary encoding method represents any solution as a vector (string) of bits. This method is based on the Hamming distance, which represents the number of positions between two strings of equal length in which corresponding symbols differ.

1-***Hamming distance neighborhood:-*** A neighbor solution is generated by flipping one bit of the initial (candidate) solution in this representation. This representation is shown in Figure 7.1. In this representation n neighbor solutions, each of size n is generated from an initial solution of size n.

2-***Hamming distance neighborhood:-*** In this representation, a neighbor solution is generated by flipping two values of an initial (candidate) solution. Thus $\frac{n\times(n-1)}{2}$ neighbor solutions each of size n are generated from an initial solution of size n.

In a similar way for a 3-*hamming distance neighborhood*, neighbor solutions are generated by flipping three values from an initial (candidate) solution. Thus $\frac{n\times(n-1)\times(n-2)}{6}$ neighbor solutions each of size n are generated from an initial solution of size n.

7.5 Accelerated System used for PPP

Generally, multi-core processors have two or more cores assembled on a single computing platform. To enhance the execution speed of a program, all cores of the multi-core processor are running in parallel. The operating system (OS) treats each core as a separate processor, and the OS scheduler maps each core to *threads/processor*, and the same memory is physically mapped with them. Multi-core processors can process multiple instructions and multiple data. In this

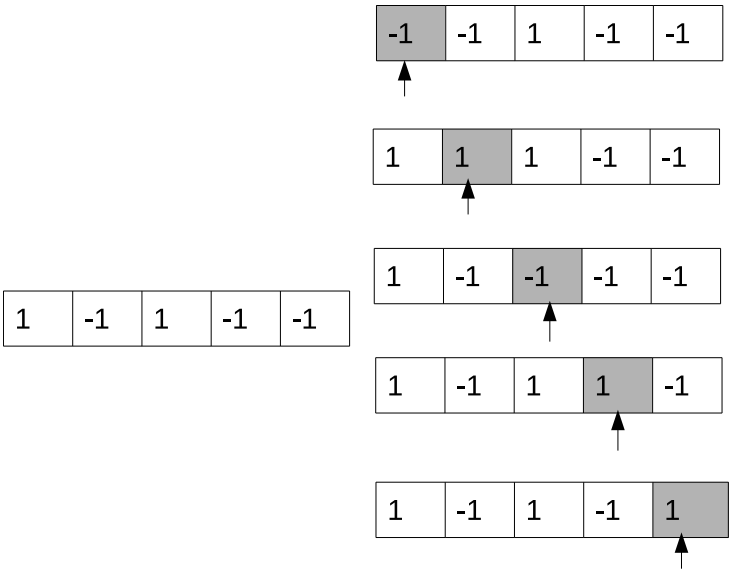

FIGURE 7.1: Binary encoding for one hamming distance

work, we used a CPU with 4 cores and a 3.2 GHz clock speed processor of Intel(R) Core (TM) $i5-6500$.

Here it is used Nvidia GeForce GTX 1050 GPU which has 4 GB DDR5 memory, 5 SMs, and each SMs has 128 CUDA cores *i.e.* a total of 640 Cuda cores. We installed and used Cuda toolkit 10.1 on Nvidia GeForce GTX 1050 GPU card.

7.6 Mapping SA for PPP to Multi-core and Many-core Architecture

In [3], the authors expressed the energy function mathematically using Equation 7.2 for solving PP by applying a simulated annealing meta-heuristic. The candidate vector Z is a solution for PP if and only if the energy function $E(Z)$ is minimum (*i.e.* $E(Z) = 0$).

$$E(Z) = \frac{1}{2}\sum_{i=1}^{m}(\|(IZ)_i\| - (IZ)_i) \tag{7.2}$$

However, this energy function is not suited for finding PPP solutions. Knudsen and Meier [85] mathematically expressed a new energy function for simulated

annealing meta-heuristic to find a solution for PPP in Equation 7.3.

$$E(Z) = g_1 \sum_{i=1}^{m} (\|(IZ')_i\| - (IZ')_i) + g_2 \sum_{i=1}^{n} (\|H_i - H'_i\|) \tag{7.3}$$

Where H represents the histogram vector over the integers. If m, n is odd then H_i is set for only odd values of i, $1 \leq i \leq n$. Z' is the candidate vector, H'_i is the histogram for finding a ϵ-vector Z, and $g_1 \geq 30$, $g_2 = 1$.

7.6.1 SA Implementation

We implemented SA meta-heuristic on multi-core CPU and many-core GPU architecture. We fixed the initial parameter of the SA meta-heuristic as Temperature T as 1000 and cooling rate as 0.001.

7.6.1.1 SA on Multi-core Architecture

In multi-core architecture (CPU), first, we randomly generated the ϵ-matrix I of size $m \times n$ and a ϵ-vector Z of size $n \times 1$; then, we find the multiset matrix IZ of size $m \times 1$, using this we calculated the histogram of elements of multiset matrix. We again also generated a random ϵ-vector Z as the candidate vector initial solution, and from that, the neighbor solution is generated using one hamming distance. We found $m \times 1$ multiset matrix from each neighbor solution. We calculated the energy function using Equation 7.3, then applied SA to find the candidate vector solution. In the next iteration, we reduced the temperature by the reducing factor at each iteration; this iteration will continue until the temperature becomes 1. Finally, we noted the solution and made them the initial solution for the next iteration, and this process will continue until it reaches the maximum number of iterations.

Algorithm 12 Simulated Annealing (SA) on GPU

Input: size of matrix m and n.
Output: Desired solution.

1: Generate a random ϵ I and Z matrix
2: Evaluate the objective function of the solution
3: Generate random initial candidate solution
4: Set the initial parameters for SA
5: **for all** neighbor solutions Z' **do**
6: **while** not Termination_Criteria *i.e.* $T < 1$ **do**
7: **repeat**
8: Evaluate the multi-set matrix IZ' on GPU in parallel.
9: Results are sent back from GPU to CPU.
10: $\Delta C = g_1(\|f(IZ') - f(IZ)\|) + g_2(\|H - H'\|)$; /* H and H' are histogram of initial and neighbor solution */
11: **if** $\Delta C = 0$ **then**
12: $Z = Z'$ /* Accept the neighbor solution */
13: **else**
14: Accept the neighbor solution Z' with a probability $e^{\frac{-\Delta C}{T}}$
15: **end if**
16: **until** Absolute Temperature (Equilibrium condition) is reached
17: $T = g(T)$; /* Update the $temperature = temperature \times coolingRate$ */
18: **end while**
19: **end for**

7.6.1.2 SA on Many-core Architecture

In many-core architecture (GPU), we implemented SA. We first, generated a random ϵ I matrix of size $m \times n$ and ϵ Z matrix of size $n \times 1$. We find the multi-set matrix IZ on GPU, and results are copied back from GPU to CPU. We also calculated the histogram of each odd element of the multi-set matrix. Again we generated a random initial candidate vector of size $n \times 1$, and from this initial solution, we generated neighbor solutions using binary encoding with hamming distance one. We calculated the objective function's cost for each neighbor solution and applied the SA meta-heuristic to accept or reject the neighbor solution. For each iteration, the SA meta-heuristic continues to run until its temperature parameter T becomes one. We fixed the maximum number of iterations as 1000 and T as the value of n. The modified algorithm of SA on GPU is described in Algorithm 12; here in this algorithm, we have taken g_1 and g_2 as constant whose value is 30 and 1 respectively.

7.7 Experimental Results

Here speedup and number of solutions are compared between multi and many-core architecture for SA meta-heuristic. We have taken instance size with different variations and also we reported the results with the hamming distance one and two. We also fixed the number of solutions space and analyzed the speedup on GPU.

7.7.1 Comparison of the Multi and Many-core Architecture of PPP with SA Meta-heuristic

We compared the execution time and the number of solutions between multi-core (CPU) and many-core (GPU) for different instance sizes from 73 to 1301 by varying the randomly generated $Imatrix$. In this report we considered the instance size by varying the m and n like $m = n, m > n, m < n$, and $n = m + 16$. In one experiment, we fixed the number of $Imatrix$ as 10 and 1000; in another, we randomly generated the number of $Imatrix$ as 1000. We reported the result by considering the number of $Imatrix$ as 10 in Table 7.1. Here in this table, we can see that as the size of the instance increases speedup on GPU, it also increases. Here, for instance size $101 - 117$, we found 16 number of solutions and execution time 0.168 seconds on CPU, while on GPU number of solutions is 70 and execution time is 0.034 seconds, so here GPU gives more number of solutions and also speedup is 4.87. Similarly, we reported the result in Table 7.2 for 1000 *Imatrices*. Here we observed that as the no. of *Imatrices* increased no. of solutions also increased as well as speedup on GPU is also increased. In Table 7.2 for instance size $101 - 117$, no. of solutions, execution time on CPU and GPU are 2318, 20.189 seconds and 5327, 2.383 seconds respectively.

We computed the PPP using randomly generated *Imatrix*, and neighbor solutions are generated using hamming distance one. Each neighbor solution is iterated 1000 times, and the best neighbor solution is chosen using the SA meta-heuristic. After execution, we noted the result in Table 7.3. Here again, we observed that as

Instance	CPU		GPU		speedup
	no. of sols.	Time (in sec.)	no. of sols.	Time (in sec.)	
73-73	9	0.048	30	0.010	4.86
81-81	27	0.148	40	0.014	10.74
101-101	27	0.217	50	0.024	8.91
101-81	35	0.140	70	0.026	5.41
121-81	35	0.294	67	0.036	8.28
101-117	16	0.168	70	0.034	4.87
121-137	18	0.433	50	0.036	11.87
151-167	20	0.702	40	0.047	14.90
301-317	22	5.806	47	0.209	27.80
601-617	31	47.519	66	1.133	41.93
801-817	30	118.524	67	2.074	57.16
1001-1017	30	224.113	68	3.137	71.45
1301-1317	42	573.927	90	7.118	80.63

TABLE 7.1: No. of solutions and execution time of PPP on CPU and GPU where no. of Imatrix is 10

Instance	CPU		GPU		speedup
	no. of sols.	Time (in sec.)	no. of sols.	Time (in sec.)	
73-73	1825	5.453	5918	1.301	4.19
81-81	2286	9.555	2436	0.879	10.87
101-101	2296	17.947	4394	2.111	8.50
101-81	1922	10.814	3453	1.316	8.22
121-81	1989	18.234	2948	1.399	13.04
101-117	2318	20.189	5327	2.383	8.47
121-137	2595	39.593	4251	2.663	14.87
151-167	164	81.144	5115	5.227	15.52
301-317	1764	617.803	5593	18.706	33.03

TABLE 7.2: No. of solutions and execution time of PPP on CPU and GPU where no. of Imatrix is 1000

the instance size increased, the GPU speedup also increased. We also generated neighbor solutions using hamming distance 2 and results are noted in Table 7.4. Here in this Table 7.4, as the neighbor solutions space increased, speedup on GPU is also increased and it gives up to 165 for instance size $(151 - 167)$. Here for instance size $101 - 117$, no. of solutions, execution time on CPU and GPU are 2586, 1733.122 seconds and 3795, 16.515 seconds respectively.

When we fixed the total number of solution space or test cases as 1000 and neighbor solutions are generated using hamming distance one, then we reported the results in Table 7.5. In Table 7.5, as the no. of solutions space is reduced, the no. of

Instance	CPU		GPU		speedup
	no. of sols.	Time (in sec.)	no. of sols.	Time (in sec.)	
73-73	1567	4.420	4000	0.989	4.47
81-81	1949	7.999	5986	1.639	4.88
101-101	2593	20.649	3970	1.682	12.27
101-81	1951	10.062	4101	1.486	6.77
121-81	2168	13.913	3883	1.691	8.23
101-117	2233	20.383	5907	2.500	8.15
121-137	2267	33.072	8271	4.422	7.48
151-167	322	78.634	5999	5.213	15.08
301-317	2940	644.409	5702	17.938	35.92

TABLE 7.3: No. of solutions and execution time of PPP on CPU and GPU where $hd = 1$ and no. of iteration 1000

Instance	CPU		GPU		speedup
	no. of sols.	Time (in sec.)	no. of sols.	Time (in sec.)	
73-73	1931	296.891	4043	5.440	54.58
81-81	2091	412.136	4613	6.085	67.72
101-101	2399	1110.418	4741	12.013	92.43
101-81	764	529.524	4491	6.946	76.23
121-81	1713	831.306	2394	4.247	195.75
101-117	2586	1733.122	3795	16.515	104.94
121-137	2750	3982.322	4817	29.278	136.02
151-167	2794	8764.841	5195	52.923	165.62

TABLE 7.4: No. of solutions and execution time of PPP on CPU and GPU where $hd = 2$ and no. of iteration 1000

solutions and speedup on GPU is also reduced. Here, for instance size $101 - 117$, no. of solutions, execution time on CPU and GPU are 5, 0.051 seconds and 970, 0.449 seconds, respectively. Here on GPU, instead of taking less time, it takes more time from CPU because of less no. of solution space, and GPU takes more time in setting up and initialization than execution. We analyzed the speedup on GPU for all four above versions $Imatrix10, Imatrix1000$, hamming distance one with iteration 1000, and hamming distance two with iteration 1000 in Figure 7.2. From Figure 7.2, we can see that for hamming distance two, speedup on GPU is high for all the instance sizes, and the highest speedup is achieved for the instance size $(121 - 81)$.

Instance	CPU		GPU		speedup
	no. of sols.	Time (in sec.)	no. of sols.	Time (in sec.)	
73-73	8	0.023	1000	0.342	0.07
81-81	8	0.048	1000	0.306	0.16
101-101	5	0.042	1000	0.404	0.10
101-81	8	0.040	1000	0.345	0.12
121-81	6	0.046	1000	0.435	0.11
101-117	5	0.051	970	0.449	0.11
121-137	4	0.061	1000	0.532	0.12
151-167	3	0.093	1000	0.920	0.10
301-317	2	0.503	1000	2.949	0.17
601-617	1	1.954	1000	17.404	0.11
801-817	0	2.376	1000	31.376	0.08
1001-1017	0	4.082	1000	48.916	0.08
1301-1317	0	6.311	1000	85.257	0.07

TABLE 7.5: No. of solutions and execution time of PPP on CPU and GPU where $hd = 1$ and fixed test case 1000

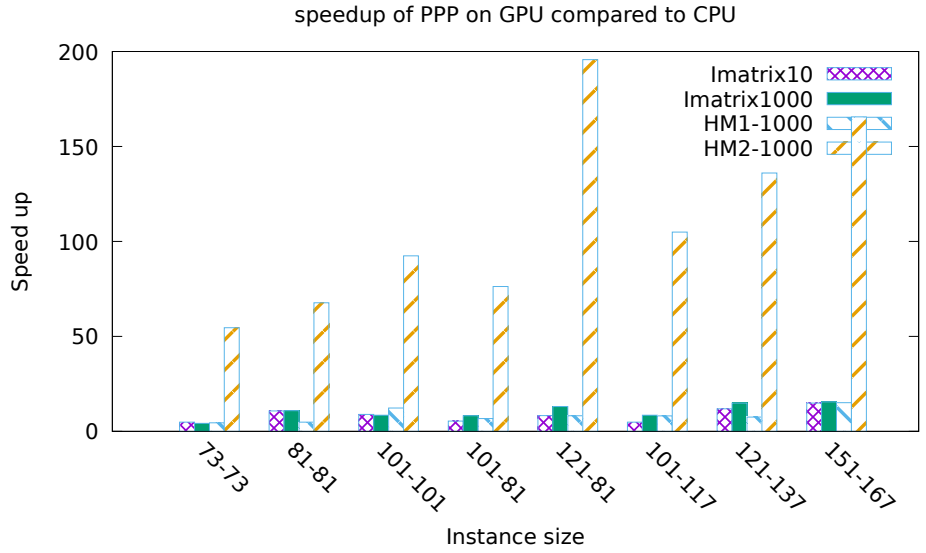

FIGURE 7.2: Speedup of PPP on GPU as compared to serial machine CPU

7.8 Performance Analysis of SA Meta-heuristic

In this section, we analyzed the performance of the SA meta-heuristic on a sequential machine and a GPU. First, by using a GNU profiling tool (i.e., gprof) [13], the performance of SA meta-heuristic is checked for solving PPP on a serial machine. We analyzed the performance of PPP, for instance size $(101 - 117)$, by randomly generating *Imatrix* and *Zmatrix*. For each solution, we run 1000 iterations on

gprof output

```
Flat profile:
Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds  calls    us/call  us/call name
66.80    14.38     14.38  466481    30.82    30.82  IZ_matrix
31.26    21.10     6.73   466481    14.42    14.42  findHistogram
0.93     21.30     0.20   465481     0.43     0.43   energyFunction
0.74     21.46     0.16                              main
0.37     21.54     0.08   103000     0.78     0.78   rpermute
0.00     21.54     0.00        4     0.00     0.00   cpuSecond
```

TABLE 7.6: gprof output

the sequential machine (i.e., CPU) and noted the results of the *gprof* profiling tool, which are shown in Table 7.6.

Here from the above *gprof* results, we can see that only the IZ_matrix function is taking 66.80% of the whole execution time. So parallelization of the IZ_matrix function will increase the performance of PPP. After analyzing the program, we found the three most probable parallel sections, which are as follows:

- *Parallel section (P1):* In this parallel section, we generated the best solutions from many initial solutions involving many iterations. Neighbor solutions are generated and evaluated in each iteration, and the best solution is selected for the next iteration.
- *Parallel section (P2):* We generated many neighbor solutions from each initial solution that can be parallelized in this parallel section.
- *Parallel section (P3):* In this parallel section, each solution is evaluated to get the cost function, which can be done in parallel.

The task graph of each parallel section is generated by some specific tools-*Contech*.

7.8.1 Contech Tools for Task Graph Generation

First, we convert the program from *C programming* to *OpenMP* to generate the task graph using *contech* tools. Then we generated the task graph for each parallel

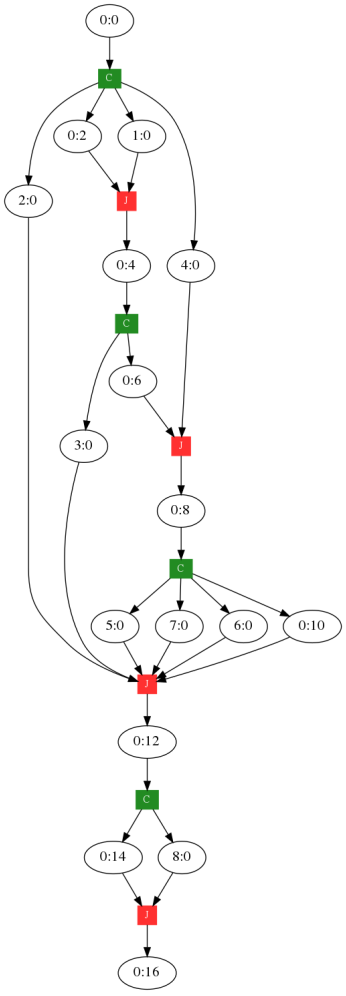

FIGURE 7.3: Task graph of SA for P1

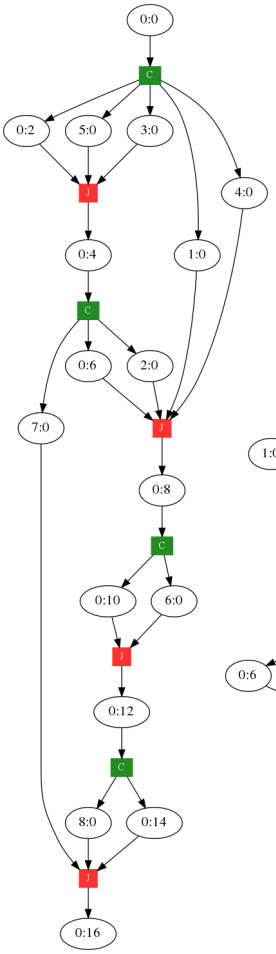

FIGURE 7.4: Task graph of SA for P2

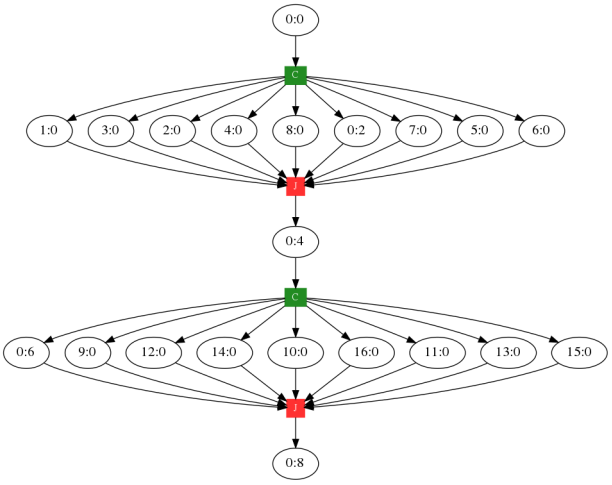

FIGURE 7.5: Task graph of SA for P3

section by taking the number of initial solutions as two and the number of iterations as one.

Figure 7.3 represents the task graph of parallel section $P1$. If additional initial solutions and iterations are performed, we can observe from this figure that there are numerous create (C) and join (J) activities that can be parallelized. Here the thread id 0 is the main task with 16 dependencies.

Figure 7.4 represents the task graph of parallel section $P2$. In this section also, there are many *create*, and *join* functions that can be parallelized concurrently.

Here also, the thread id 0 is the main task with 16 dependencies.

Figure 7.5 represents the last parallel section $P3$, in which many small *create* and *join* functions are there which can be parallelized simultaneously.

7.8.1.1 Analysis of the Task Graphs

We analyzed the results of *gprof* tools, and we found the probable parallel section using this. We generated the task graph of *SA* meta-heuristic through *Contech* tools. After analysis of the task graph, we suggest the most suitable target architecture for each parallel section.

- Suitable architecture for parallel section $P1$: In Figure 4.18, we can see that there are many create and join has long sequence id so more significant core like Intel Xeon is the most suitable architecture for best parallelization. In this parallel section, several neighbor solutions are generated and evaluated for each initial solution, and this process is repeated for each iteration. Here for each initial solution, one processor is responsible for finding the best optimal solution.

- Suitable architecture for parallel section $P2$: In Figure 4.19, the generation of many neighbor solutions from one initial solution can be parallelized. Parallelizing this section requires calling each neighbor's solution numerous times to evaluate the cost function. GPUs or SIMD architectures are best for this purpose.

- Suitable architecture for parallel section $P3$: In Figure 4.20, many *create* and *join* functions can be executed simultaneously. In this parallel section, each neighbor solution is evaluated which involves a reduction operation. So for a less number of bigger cores, *OpenMP* is the most suitable.

7.9 Summary

The main aim of this work is to find the speedup on many-core architecture like GPU concerning the serial machine-like CPU for solving PPP using SA meta-heuristic. In this work, first, we briefly introduce the SA meta-heuristic and the many-core architecture. We analyzed the performance of the SA meta-heuristic on GPU as well as on CPU in terms of execution time and the resultant number of solutions of PPP.

In this chapter, we compared the execution time between CPU and GPU; we found that as the size of the instance increases speedup of GPU is also increased, i.e., for large size instances, GPU performance is best. Here, for input instance size $151 - 167$ using hamming distance two, we got the highest speedup around 165.

Chapter 8

Conclusion

This research uses many-core architecture to parallelize meta-heuristics for combinatorial optimization problems. We considered the quadratic assignment, traveling salesman, and permuted perceptron problems in this work. Towards this, we worked in the following two directions: (i) to find the parallel sections and efficiently map them to the architecture, and (ii) to test the different meta-heuristics for different optimization problems. Towards proposing a solution for the former direction, we considered the serial architecture as a multi-core CPU and parallel architecture as NVIDIA GPU. We generated a task graph using *contech* tools and different profiling tools like *gprof, Valgrind, etc.*. We manipulated the parallel sections and mapped them to the serial and parallel architecture. We also used shared, constant, texture, local, and global memory in this context. Whereas, for proposing a solution for the latter direction, we used two different types of meta-heuristics: (i) single solution-based meta-heuristics: ILS, SA, and TS, and (ii) population-based meta-heuristics: GA, PSO, and CSA. This chapter sums up all the proposed contributions of this dissertation along with the future directions for research.

Meta-heuristics	Highest	Instance of size 100		
	class 2 tai256c S	class 1 tai100a S	class 2 tai100b S	class 3 sko100a S
ILS	*107.55*	55.38	62.63	**64.67**
SA	*60.99*	18	**23.19**	22.79
GA	*22.0*	**11.39**	11.27	11.30
PSO	***127.56***	29.55	**64.21**	63.96
CSA	*60.34*	**12.24**	6.89	7.30
TS	*27.57*	**18.48**	16.87	17.69

TABLE 8.1: Highest speedup on GPU for instances of same size 100 for QAP

8.1 Summary of Contributions

- **Analysis of Meta-heuristics for QAP in Accelerated Systems:** Here, we used different meta-heuristics (ILS, SA, GA, PSO, CSA, and TS) on the massively parallel device *GPU*. We have taken standard instances from QAPLIB and categories among four classes: class (1) unstructured, randomly generated instances, e.g. *taixxa*, class (2) real-life like instances, e.g. *taixxb, taixxc*, class (3) instances with grid-distances, e.g. *skoxx, skoxxa* class (4) real life instances, e.g. *elsxx, kraxxa, kraxx, stexxa*. We calculated the speedup on pthread and GPU compared to CPU, and deviation of optimal solution from the standard QAPLIB library. We used adjacent pairwise exchange methods to generate the neighbor solutions from initial solutions. We implemented all the above six meta-heuristics and analyzed their performances by generating the task graph using *contech* tools. We also compared all the above six meta-heuristics and the results reported in Table 8.1. Here we can see that the highest speedup, 127.56 is achieved with the PSO meta-heuristic. When we compare with instances of the same size for different classes, then ILS shows the highest speedup 64.67 on GPU. We found that the speedup for a given meta-heuristic increased as the size of the instances increased because the GPU's hardware and resources are used more effectively for large-size instances. This is the probable reason that GPU is showing the highest speedup for instance *tai256c* for all meta-heuristics.

- **Analysis of ILS Meta-heuristic for QAP on GPU Spatial Memory:** In this work, we utilized the GPU architectural properties in terms of memory

Meta-heuristics	Highest		Instance of size 100		
	Instance	S	kroa100 S	krob100 S	kroc100 S
ILS	a280	*24.17*	**9.65**	9.53	9.63
SA	nrw1379	*5.59*	**3.52**	**3.52**	**3.52**
GA	lin318	*1.73*	1.25	**1.26**	**1.26**
PSO	pr136	*5.51*	**5.18**	5.15	5.14
CSA	pr107	***50.53***	**2.14**	0.56	1.43
TS	berlin52	*1.62*	1.47	**1.56**	**1.56**

TABLE 8.2: Highest speedup on GPU for instances of same size 100 for TSP

to efficiently parallelize the ILS meta-heuristic. We used shared, texture, constant, local, and global memories of the GPU. We put the read-only instances like distance and flow matrices on read-only memories constant and texture memories. By putting the instances of different memories, shared memory gives the best while constant memory gives the worst performance on GPU speedup. When we mix the input instances with constant and texture memory by varying one time on constant and another time on texture memory, then the speedup on GPU is more significant than only constant memory but worse than only texture memory.

- **Analysis of Meta-heuristics for TSP in Accelerated Systems:**

  Like QAP, we used the above six meta-heuristics (ILS, SA, GA, PSO, CSA, and TS) to implement TSP. We have taken standard symmetric instances from TSPLIB. We used adjacent pairwise exchange permutation methods to generate neighbor solutions from initial solutions. Instead of evaluating a complete neighbor solution in each iteration, we used incremental solution evaluation to minimize the data transfer time from CPU to GPU. We implemented all the above six meta-heuristics on CPU, pthread, and GPU and recorded the speedup on GPU. We compared the performance of the six meta-heuristics mentioned above on the GPU, and Table 8.2 shows the results.

  Here we can see that in Table 8.2, CSA is recorded the highest speedup 50.53 for instance *pr107*, and TS shows the lowest speedup 1.62 for instance *berlin52* on GPU, among all the above meta-heuristics. When we compared the same size 100 of three different instances (*kroa100, krob100, kroc100*), the ILS, PSO,

and CSA recorded highest speedup for *kroa100*, SA equal speedup, and GA and TS showed equal for *krob100, kroc100* instances.

- **Analysis of Meta-heuristics for PPP in Accelerated Systems:** In this work, we used only the SA meta-heuristic on GPU. Here we considered the initial solutions in terms of 1 and -1, which we generated randomly of size from 73 to 1301. We used a binary encoding method with hamming distances one and two to generate the neighbor solutions from an initial solution. Here we considered the instance size by varying the m and n like $m = n, m > n, m < n$, and $n = m + 16$. Generally, input instances of size with a difference of 16 are challenging to solve for PPP and take a huge amount of time. In this work, we fixed the number of *Imatrix* as 10 and 1000 both ways manually and randomly. We analyzed the performance of the SA meta-heuristic and recorded the desired number of solutions and execution time on CPU and GPU. We also calculated the speedup on the GPU. We achieved the highest speedup, 165.62, for instance 151-167 using hamming distance two and number of iteration 1000 among all considered instances from 73 to 151. We obtained the speedup 27.80 and 33.03 when we fixed the no. of *Imatrix* 10 and 1000, respectively for instance 301-317. Here we observed that as the size of the instance increases, the generation of neighbor solutions also increases, and the resource utilization of GPU is more efficient, resulting in more GPU speedup.

8.2 Scope for Future Work

The contributions of this thesis can be extended in several ways. Some of these possible future research directions are listed below:

- For each considered optimization problem in this thesis, with the help of GPU hardware properties like setting the GPU device parameter performance of considered meta-heuristics may be improved.

- Our proposed methods can be extended to multi GPU and cluster systems. Here again performance of these meta-heuristics may be improved.
- By applying recent new noble meta-heuristics, solutions for QAP, PPP, and TSP optimization problems can be improved further.
- By considering other optimization problems apart from three (QAP, PPP, TSP), performance of accelerated machine may be improved.

Appendix A

Experimental Setup Parameter

This appendix focuses on the experimental setup used in our approaches. In this dissertation, we used two hardware: CPU and GPU. In GPU, we used two different cards GPU the first is the NVIDIA Ge Force GTX 1050 GPU card, and another one is NVIDIA Ge Force GTX 980 Ti GPU card. We used the CUDA C programming model to write and run the code on a heterogeneous system. We used Intel(R) Core (TM) i5-6500 CPU, with a 3.2 GHz clock speed and 4 CPU cores. We also used some profiling tools like *gprof* and *gcov* to check the memory leaks. *Contech tools* we used for generating the task graph for different parallel sections of meta-heuristics.

A.1 GPU Parameter

A.1.1 *NVIDIA GeForce GTX 980 Ti* Configuration

The detailed configuration of *NVIDIA GeForce GTX 980 Ti* is listed in Table A.1.

Table A.1: Nvidia GeForce GTX 980 Ti Configuration

Architecture	Maxwell 2.0
No. of CUDA cores	2816
No. of multiprocessors	22
CUDA cores/multiprocessor	128
No. of registers per block	65536
Global memory	6 GB DDR5
Constant memory	64 KB
Shared memory per block	48 KB
Warp size	32
Maximum no. of threads per block	1024
Maximum no. of threads per multiprocessors	2048
Maximum no. of warps per multiprocessors	64

Table A.2: Nvidia GeForce GTX 1050 Configuration

Architecture	Pascal
No. of CUDA cores	640
No. of multiprocessors	5
CUDA cores/multiprocessor	128
No. of registers per block	65536
Global memory	4 GB DDR5
Constant memory	64 KB
Shared memory per block	48 KB
Warp size	32
Maximum no. of threads per block	1024
Maximum no. of threads per multiprocessors	2048
Maximum no. of warps per multiprocessors	64

A.1.2 *NVIDIA GeForce GTX 1050* Configuration

The detailed configuration of *NVIDIA GeForce GTX 1050* is listed in Table A.2.

A.2 Meta-heuristics Parameter for QAP

In this work, we have taken the following parameters for the meta-heuristics of solving QAP.

- **Iterated local search (ILS):** Number of initial solution=500, number of iteration=10.

- **Particle swarm optimization (PSO):** Number of initial solution=500, number of iteration=10
- **Simulated annealing (SA):** Number of initial solution=500, number of iteration=10
- **Crow search algorithm (CSA):** Number of initial solution=size of input instance, Number of iteration=5000
- **Tabu search (TS):** Number of initial solution=size of input instance, Number of iteration=10.
- **Genetic algorithm (GA):** Number of initial solution=5000, number of iteration=10, Number of neighbor solution=25000

A.3 Contech Tools Installations

Following are the steps used for installing the Contech tools-

1. First download the Contech software http://bprail.github.io/contech/
2. unzip in ubuntu home using tar xvzf bprail-contech-f085829.tar.gz
3. Download llvm3.8.0 prebuilt binaries from
   http://releases.llvm.org/download.html
4. Rename llvm as llvm_temp in bprail-contech-f085829 and
   paste prebuilt binaries and rename as llvm
5. In Makefile set the path as
   CONTECH_HOME=/home/manoj/bprail-contech-f085829/
   LLVM_HOME=/home/manoj/bprail-contech-f085829/llvm/
6. Then make locally
7. Make the the envtsetting.sh file as
   export LLVM_DIR=/home/manoj/bprail-contech-f085829/llvm/
   export LLVM_HOME=/home/manoj/bprail-contech-f085829/llvm/
   export CONTECH_LLVM_HOME=/home/manoj/bprail-contech-f085829/llvm/
   export CONTECH_HOME=/home/manoj/bprail-contech-f085829/

```
   export LD_LIBRARY_PATH=/home/manoj/bprail-contech-f085829/common/taskLib/
   export PATH=$PATH:/home/manoj/bprail-contech-f085829/llvm/bin/
8. Run envtsetting.sh using source envtsetting.sh
9. Go to llvm-contech then do cmake .  Or cmake And  make or make .
   If -flto error comes then go to that file and
   remove the argument -flto from -flto as
   If  some .so file or llvm lib file is not linking then
   make the link using pointer like
   ls -l file2
   lrwxrwxrwx 1 root root 5 Mar 31 03:54 file2 -> file1
   (here file 2 is pointing file 1)
10. Make the new dir and put C or OMP file to run
11. Make run.sh in new dir as
    # if libomp.so not found after running ./a.out then set path
    # export LD_LIBRARY_PATH=/home/manoj/bprail-contech-f085829/llvm/lib
    ../scripts/contech_wrapper_par.py -fopenmp pi.c   #this is used for running
                                                      omp program from Contech
    ./a.out                                           #this is generated
                                                      binary and simply run
                                                      as usually
    ../middle/middle /tmp/contech_fe Abstest1.graph   #this is used after
                                                      generating trace file
                                                      contech_fe which is in
                                                      temp, used for generating
                                                      task graph
    ../backend/TaskGraphVisualizer/taskViz Abstest1.graph
                                                      #used for
                                                      visuallizing the
                                                      graph in .png format
     Run either as a whole like sh run.sh or one by one.
```

Bibliography

[1] I. Boussad, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Information Sciences*, vol. 237, pp. 82 – 117, 2013.

[2] T. C. Koopmans and M. Beckmann, "Assignment Problems and the Location of Economic Activities," *Econometrica: journal of the Econometric Society*, pp. 53–76, 1957.

[3] D. Pointcheval, "A new identification scheme based on the perceptrons problem." Springer, 1995, pp. 319–328.

[4] M. Dorigo and L. M. Gambardella, "Ant colonies for the travelling salesman problem," *Biosystems*, vol. 43, no. 2, pp. 73 – 81, 1997.

[5] C. W. COMMANDER, "A survey of the quadratic assignment problem, with applications," Ph.D. dissertation, UNIVERSITY OF FLORIDA, 2003.

[6] R. E. Burkard, S. E. Karisch, and F. Rendl, "QAPLIB &Ndash; A Quadratic Assignment ProblemLibrary," *J. of Global Optimization*, vol. 10, no. 4, pp. 391–403, Jun. 1997.

[7] B. P. Railing, E. R. Hein, P. Vassenko, and T. M. Conte, "Contech: A tool for analyzing parallel programs," Georgia Institute of Technology, Tech. Rep., 2013.

[8] K. Panwar and K. Deep, "Discrete grey wolf optimizer for symmetric travelling salesman problem," *Applied Soft Computing*, vol. 105, p. 107298, 2021.

[9] P. Stodola, P. Otřísal, and K. Hasilová, "Adaptive ant colony optimization with node clustering applied to the travelling salesman problem," *Swarm and Evolutionary Computation*, p. 101056, 2022.

[10] E.-G. Talbi, *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[11] F. Glover, "A template for scatter search and path relinking," in *European conference on artificial evolution*. Springer, 1997, pp. 1–51.

[12] G. Reinelt, "TSPLIB- A Traveling Salesman Problem Library," *ORSA Journal of Computing*, vol. 3, no. 4, pp. 376–384, 1991.

[13] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A Call Graph Execution Profiler," *SIGPLAN Not.*, vol. 39, no. 4, pp. 49–57, Apr. 2004.

[14] T. Luong, "Parallel metaheuristics on gpu," Ph.D. dissertation, Ph. D. thesis, INRIA Lille, 2011.

[15] D. Bertsimas and R. Demir, "An approximate dynamic programming approach to multidimensional knapsack problems," *Management Science*, vol. 48, no. 4, pp. 550–565, 2002.

[16] C. Blum, J. Puchinger, G. R. Raidl, A. Roli *et al.*, "A brief survey on hybrid metaheuristics," *Proceedings of BIOMA*, pp. 3–18, 2010.

[17] H. Wang and B. Alidaee, "A new hybrid-heuristic for large-scale combinatorial optimization: A case of quadratic assignment problem," *Computers & Industrial Engineering*, vol. 179, p. 109220, 2023.

[18] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM computing surveys (CSUR)*, vol. 35, no. 3, pp. 268–308, 2003.

[19] R. Chelouah and P. Siarry, "Tabu Search Applied to Global Optimization," *European Journal of Operational Research*, vol. 123, no. 2, pp. 256 – 270, 2000.

[20] E. Lutton and J. Levy Vehel, "Holder Functions and Deception of Genetic Algorithms," *Evolutionary Computation, IEEE Transactions on*, vol. 2, no. 2, pp. 56–71, Jul 1998.

[21] T. V. Luong, N. Melab, and E.-G. Talbi, "GPU Computing for Parallel Local Search Metaheuristic Algorithms," *Computers, IEEE Transactions on*, vol. 62, no. 1, pp. 173–185, Jan 2013.

[22] R. C. Eberhart, J. Kennedy *et al.*, "A new optimizer using particle swarm theory," in *Proc. of symposium on micro machine and human science*, vol. 1. New York, NY, 1995, pp. 39–43.

[23] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," May 1998, pp. 69–73.

[24] A. Askarzadeh, "A novel metaheuristic method for solving constrained engineering optimization problems: Crow search algorithm," *Computers and Structures*, vol. 169, pp. 1 – 12, 2016.

[25] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers operations research*, vol. 13, no. 5, pp. 533–549, 1986.

[26] A. Vajda, "Multi-core and Many-core Processor Architectures," in *Programming Many-Core Chips*. Springer US, 2011, pp. 9–43.

[27] M. Rantonen, T. Frantti, and K. Leivisk, "Fuzzy Expert System for Load Balancing in Symmetric Multiprocessor Systems," *Expert Systems with Applications*, vol. 37, no. 12, pp. 8711 – 8720, 2010.

[28] S. Eggers, J. Emer, H. Leby, J. Lo, R. Stamm, and D. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors," *Micro, IEEE*, vol. 17, no. 5, pp. 12–19, Sep 1997.

[29] P. Panigrahi, S. Kanchiraju, A. Srinivasan, P. Baruah, and C. Sudheer, "Optimizing MPI Collectives on Intel MIC through Effective use of Cache," Dec 2014, pp. 88–93.

[30] Intel Xeon Phi Coprocessor System Software Developers Guide, march, 2014. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-system-software-developers-guide.html

[31] NVIDIA, "CUDA C Programming Guide," September 2015, version 7.5.

[32] E. M. Loiola, N. M. M. de Abreu, P. O. Boaventura-Netto, P. Hahn, and T. Querido, "A survey for the quadratic assignment problem," *European Journal of Operational Research*, vol. 176, no. 2, pp. 657 – 690, 2007.

[33] R. D. Meller and K.-Y. Gau, "The facility layout problem: Recent and emerging trends and perspectives," *Journal of Manufacturing Systems*, vol. 15, no. 5, pp. 351 – 366, 1996.

[34] A. Stawowy, "Evolutionary based heuristic for bin packing problem," *Computers and Industrial Engineering*, vol. 55, no. 2, pp. 465 – 474, 2008.

[35] R. Carraghan and P. M. Pardalos, "An exact algorithm for the maximum clique problem," *Operations Research Letters*, vol. 9, no. 6, pp. 375 – 382, 1990.

[36] A. M. Geoffrion and G. W. Graves, "Scheduling parallel production lines with changeover costs: Practical application of a quadratic assignment/lp approach," *Operations Research*, vol. 24, no. 4, pp. 595–610, 1976.

[37] P. Kadluczka and K. Wala, "Tabu search and genetic algorithms for the generalized graph partitioning problem," *Control and cybernetics*, vol. 24, pp. 459–476, 1995.

[38] M. Pollatschek, H. Gershoni, and Y. Radday, "Optimization of typewriter keyboard by computer-simulation," *Angewandte Informatik*, no. 10, pp. 438–439, 1976.

[39] B. Wess and T. Zeitlhofer, "On the phase coupling problem between data memory layout generation and address pointer assignment." Springer, 2004, pp. 152–166.

[40] L. Steinberg, "The backboard wiring problem: A placement algorithm," *Siam Review*, vol. 3, no. 1, pp. 37–50, 1961.

[41] E. Alba, G. Luque, and S. Nesmachnow, "Parallel metaheuristics: recent advances and new trends," *International Transactions in Operational Research*, vol. 20, no. 1, pp. 1–48, 2013.

[42] P. Krömer, J. Platoš, and V. Snášel, "Nature-inspired meta-heuristics on modern gpus: state of the art and brief survey of selected algorithms," *International Journal of Parallel Programming*, vol. 42, no. 5, pp. 681–709, 2014.

[43] M.-L. Wong, T.-T. Wong, and K.-L. Fok, "Parallel Evolutionary Algorithms on Graphics Processing Unit," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, vol. 3, Sept 2005, pp. 2286–2293 Vol. 3.

[44] Q. Yu, C. Chen, and Z. Pan, "Parallel Genetic Algorithms on Programmable Graphics Hardware," in *Advances in Natural Computation*, ser. Lecture Notes in Computer Science, L. Wang, K. Chen, and Y. Ong, Eds. Springer, 2005, vol. 3612, pp. 1051–1059.

[45] J.-M. Li, X.-J. Wang, R.-S. He, and Z.-X. Chi, "An Efficient Fine-grained Parallel Genetic Algorithm Based on GPU-Accelerated," in *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, Sept 2007, pp. 855–862.

[46] W. Zhu, "A Study of Parallel Evolution Strategy: Pattern Search on a GPU Computing Platform," ser. GEC '09. ACM, 2009, pp. 765–772.

[47] S. Tsutsui and N. Fujimoto, "Solving Quadratic Assignment Problems by Genetic Algorithms with GPU Computation: A Case Study," ser. GECCO '09. ACM, 2009, pp. 2523–2530.

[48] T. Van Luong, N. Melab, and E. Talbi, "Parallel Hybrid Evolutionary Algorithms on GPU," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, July 2010, pp. 1–8.

[49] O. Abdelkafi, L. Idoumghar, and J. Lepagnot, "A survey on the metaheuristics applied to qap for the graphics processing units," *Parallel Processing Letters*, vol. 26, no. 03, p. 1650013, 2016.

[50] T. James, C. Rego, and F. Glover, "A cooperative parallel tabu search algorithm for the quadratic assignment problem," *European Journal of Operational Research*, vol. 195, no. 3, pp. 810–826, 2009.

[51] E. Sonuc, B. Sen, and S. Bayir, "A cooperative gpu-based parallel multistart simulated annealing algorithm for quadratic assignment problem," *Engineering Science and Technology, an International Journal*, vol. 21, no. 5, pp. 843–849, 2018.

[52] H. Alfaifi and Y. Daadaa, "Parallel improved genetic algorithm for the quadratic assignment problem," *International Journal of Advanced Computer Science and Applications*, vol. 13, no. 5, 2022.

[53] R. Matousek, L. Dobrovsky, and J. Kudela, "How to start a heuristic? utilizing lower bounds for solving the quadratic assignment problem," *International Journal of Industrial Engineering Computations*, vol. 13, no. 2, pp. 151–164, 2022.

[54] A. Silva, L. C. Coelho, and M. Darvish, "Quadratic assignment problem variants: A survey and an effective parallel memetic iterated tabu search," *European Journal of Operational Research*, vol. 292, no. 3, pp. 1066–1084, 2021.

[55] R. POVEDA, E. CARDENAS, and O. GARCIA, "Hybrid of cellular parallel genetic algorithm and greedy 2-opt local search to solve quadratic assignment problem using cuda," *Journal of Engineering Science and Technology*, vol. 15, no. 5, pp. 3082–3095, 2020.

[56] E. Özçetin and G. Öztürk, "A parallel iterated local search algorithm on gpus for quadratic assignment problem," *International Journal of Engineering Technologies IJET*, vol. 4, no. 2, pp. 124–128, 2018.

[57] J. R. Cheng and M. Gen, "Accelerating genetic algorithms with gpu computing: A selective overview," *Computers & Industrial Engineering*, vol. 128, pp. 514–525, 2019.

[58] L. Stoltzfus, M. Emani, P.-H. Lin, and C. Liao, "Data placement optimization in gpu memory hierarchy using predictive modeling," in *Proceedings of the Workshop on Memory Centric High Performance Computing*, 2018, pp. 45–49.

[59] M. Essaid, L. Idoumghar, J. Lepagnot, and M. Brévilliers, "Gpu parallelization strategies for metaheuristics: a survey," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 34, no. 5, pp. 497–522, 2019.

[60] M. Bashiri and H. Karimi, "Effective heuristics and meta-heuristics for the quadratic assignment problem with tuned parameters and analytical comparisons," *Journal of Industrial Engineering International*, vol. 8, no. 1, pp. 1–9, 2012.

[61] S. Sahni and T. Gonzalez, "P-Complete Approximation Problems," *J. ACM*, vol. 23, no. 3, pp. 555–565, Jul. 1976.

[62] S. Memeti, S. Pllana, A. Binotto, J. Kołodziej, and I. Brandic, "Using metaheuristics and machine learning for software optimization of parallel computing systems: a systematic literature review," *Computing*, vol. 101, no. 8, pp. 893–936, 2019.

[63] T. V. T. Van Luong, L. Loukil, N. Melab, and E.-G. Talbi, "A gpu-based iterated tabu search for solving the quadratic 3-dimensional assignment problem," in *ACS/IEEE International Conference on Computer Systems and Applications-AICCSA 2010*. IEEE, 2010, pp. 1–8.

[64] T. Stützle, "Iterated local search for the quadratic assignment problem," *European Journal of Operational Research*, vol. 174, no. 3, pp. 1519–1539, 2006.

[65] B. P. Railing, E. R. Hein, and T. M. Conte, "Contech: Efficiently generating dynamic task graphs for arbitrary parallel programs," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, pp. 25:1–25:24, Jul. 2015.

[66] R. M. Karp, "On the computational complexity of combinatorial problems," *Networks*, vol. 5, no. 1, pp. 45–68, 1975.

[67] J. Wu, L. Zhou, Z. Du, Y. Lv *et al.*, "Mixed steepest descent algorithm for the traveling salesman problem and application in air logistics," *Transportation Research Part E: Logistics and Transportation Review*, vol. 126, pp. 87–102, 2019.

[68] P. Baniasadi, M. Foumani, K. Smith-Miles, and V. Ejov, "A transformation technique for the clustered generalized traveling salesman problem with applications to logistics," *European Journal of Operational Research*, vol. 285, no. 2, pp. 444–457, 2020.

[69] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht, "Genetic algorithms for the traveling salesman problem," in *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, vol. 160, no. 168. Lawrence Erlbaum, 1985, pp. 160–168.

[70] J. Scholz, "Genetic algorithms and the traveling salesman problem a historical review," *arXiv preprint arXiv:1901.05737*, 2019.

[71] A. Uchida, Y. Ito, and K. Nakano, "Accelerating ant colony optimisation for the travelling salesman problem on the gpu," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 4, pp. 401–420, 2014.

[72] S. Kang, S.-S. Kim, J. Won, and Y.-M. Kang, "Gpu-based parallel genetic approach to large-scale travelling salesman problem," *The Journal of Supercomputing*, vol. 72, no. 11, pp. 4399–4414, 2016.

[73] J. Fosin, D. Davidović, and T. Carić, "A gpu implementation of local search operators for symmetric travelling salesman problem," *Promet-Traffic&Transportation*, vol. 25, no. 3, pp. 225–234, 2013.

[74] K. Rocki and R. Suda, "Accelerating 2-opt and 3-opt local search using gpu in the travelling salesman problem," in *2012 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2012, pp. 489–495.

[75] O. Bali, W. Elloumi, P. Krömer, and A. M. Alimi, "Gpu particle swarm optimization applied to travelling salesman problem," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 2015, pp. 112–119.

[76] P. Yelmewad and B. Talawar, "Near optimal solution for traveling salesman problem using gpu," in *2018 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. IEEE, 2018, pp. 1–6.

[77] B. A. Menezes, H. Kuchen, H. A. A. Neto, and F. B. de Lima Neto, "Parallelization strategies for gpu-based ant colony optimization solving the traveling salesman problem," in *2019 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2019, pp. 3094–3101.

[78] M. Abbasi and M. Rafiee, "Efficient parallelization of a genetic algorithm solution on the traveling salesman problem with multi-core and many-core systems," *International Journal of Engineering*, vol. 33, no. 7, pp. 1257–1265, 2020.

[79] W.-B. Qiao and J.-C. Créput, "Multiple k- opt evaluation multiple k- opt moves with gpu high performance local search to large-scale traveling salesman problems," *Annals of Mathematics and Artificial Intelligence*, vol. 88, no. 4, pp. 347–365, 2020.

[80] A. François, Q. Cappart, and L.-M. Rousseau, "How to evaluate machine learning approaches for combinatorial optimization: Application to the travelling salesman problem," *arXiv preprint arXiv:1909.13121*, 2019.

[81] U. J. Mele, X. Chou, L. M. Gambardella, and R. Montemanni, "Reinforcement learning and additional rewards for the traveling salesman problem," in

2021 The 8th International Conference on Industrial Engineering and Applications (Europe), 2021, pp. 198–204.

[82] I. I. Huerta, D. A. Neira, D. A. Ortega, V. Varas, J. Godoy, and R. Asín-Achá, "Improving the state-of-the-art in the traveling salesman problem: An anytime automatic algorithm selection," *Expert Systems with Applications*, vol. 187, p. 115948, 2022.

[83] P. Garg, "Evolutionary computation algorithms for cryptanalysis: A study," *arXiv preprint arXiv:1006.5745*, 2010.

[84] A. K. Bhateja, A. Bhateja, S. Chaudhury, and P. Saxena, "Cryptanalysis of vigenere cipher using cuckoo search," *Applied Soft Computing*, vol. 26, pp. 315–324, 2015.

[85] L. R. Knudsen and W. Meier, "Cryptanalysis of an identification scheme based on the permuted perceptron problem," in *Advances in Cryptology — EUROCRYPT '99*, J. Stern, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 363–374.

[86] J. A. Clark and J. L. Jacob, "Fault injection and a timing channel on an analysis technique," in *Advances in Cryptology — EUROCRYPT 2002*, L. R. Knudsen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 181–196.

[87] T. Van Luong, N. Melab, and E.-G. Talbi, "Local search algorithms on graphics processing units. a case study: The permutation perceptron problem," in *EvoCOP*, 2010, pp. 264–275.

Publications Related to Thesis

Journal(s):

- **M. Kumar**, A. Sahu, and P. Mitra, "A comparison of different metaheuristics for the quadratic assignment problem in accelerated systems," *Applied Soft Computing, Elsevier*, Volume 100, March 2021, Pages 106927.

Conferences:

- **M. Kumar**, P. Mitra, "Solving Quadratic Assignment Problem Using Crow Search Algorithm in Accelerated Systems", *International Conference on Machine Learning, Image Processing, Network Security and Data Sciences (MIND), Springer*, 2020.
- **M. Kumar**, P. Mitra, "Solving Quadratic Assignment Problem using Iterated Local Search on GPU Spatial Memory", *International Conference on Applied Computational Intelligence & Analytics (ACIA), AIP*, 2022.

Under Review:

- **M. Kumar**, A. Sahu, and P. Mitra, "A comparative study on permuted perceptron problem using simulated annealing in accelerated systems", *The Journal of Supercomputing, Springer*, 2022. [Submitted]
- **M. Kumar**, A. Sahu, and P. Mitra, "A comparison of different metaheuristics for the travelling salesman problem in accelerated systems". [To be communicated]