

ACCELERATING LOCAL SEARCH ALGORITHMS FOR TRAVELLING
SALESMAN PROBLEM USING GPU EFFECTIVELY

by
GİZEM ERMİŞ

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Master of Science

Sabancı University
July 2015

ACCELERATING LOCAL SEARCH ALGORITHMS FOR TRAVELLING
SALESMAN PROBLEM USING GPU EFFECTIVELY

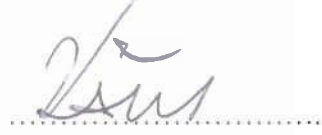
APPROVED BY:

Prof. Dr. Bülent Çatay

(Thesis Supervisor)



Asst. Prof. Dr. Kamer Kaya



Asst. Prof. Dr. Gürkan Öztürk



DATE OF APPROVAL: ...03.09.2015.....

© Gizem ERMİŞ 2015

All Rights Reserved

TABLE OF CONTENTS

1	INTRODUCTION	1
2	LITERATURE REVIEW	6
3	ARCHITECTURE	8
3.1	DEVICE MEMORIES AND DATA TRANSFER	15
3.2	THREAD SCHEDULING AND LATENCY TOLERANCE	17
3.3	MEMORY MODEL AND LOCALITY	24
4	EXPERIMENTAL DESIGN	28
4.1	PARALLELIZATION STRATEGY FOR 2-OPT ALGORITHM ON TSP	32
4.2	EXPERIMENTAL RESULTS	39
4.2.1	<i>Experiment 1-500 cities</i>	<i>41</i>
4.2.2	<i>Experiment 2-1000 Cities</i>	<i>46</i>
4.2.3	<i>Experiment 3-1500 Cities</i>	<i>48</i>
4.2.4	<i>Experiment 4-2000 Cities</i>	<i>50</i>
4.3	SEQUENTIAL VS. PARALLEL 2-OPT PERFORMANCE	51
4.4	ALGORITHM MODIFICATION TO SOLVE LARGE SIZED TRAVELLING SALESMAN PROBLEMS	52
4.4.1	<i>First Step: Decreasing Shared Memory Usage for Each City</i>	<i>52</i>
4.4.2	<i>Second Step: Dividing Problem into Sub Problems</i>	<i>54</i>
4.5	2-OPT ALGORITHM WITH INITIAL SOLUTION	60
4.6	3-OPT ALGORITHM	61
5	CONCLUSION AND FUTURE RESEARCH	65
	APPENDICES	67
	BIBLIOGRAPHY	75
	Appendix A: 2-Opt Algorithm Results for Different Kinds of Resource Allocations....	67
	Appendix B: Results for 2-Opt Large Sized Data and 3-opt Algorithms.....	71
	Appendix C: CUDA Code.....	72

LIST OF FIGURES

Figure 1.1 A basic block diagram of a generic multi-core processor	3
Figure 3.1 The architectural difference between CPU and GPU	9
Figure 3.2 Pipelining	10
Figure 3.3 The thread hierarchy in the CUDA programming model	12
Figure 3.4 The indexes produced by kernel depending on the number of blocks launched in the grid and the number of threads launched in the block	13
Figure 3.5 Overview of CUDA device memory model (Kirk and Hwu, 2013)	16
Figure 3.6 The grid of threads produced because of the kernel launch (Hwu, 2013)	17
Figure 3.7 Warp scheduling (Cooper, 2011)	19
Figure 3.8 Streaming multiprocessor structure of the GPU device (Nvidia, 2012)	22
Figure 3.9 Thread level parallelism (Volkov, 2010)	23
Figure 3.10 Iteration level parallelism (Volkov, 2010)	23
Figure 4.1 2-opt step on a travelling salesman tour	29
Figure 4.2 Sequential 2-opt algorithm	31
Figure 4.3 Assigning thread indices via built-in variables	34
Figure 4.4 Assigning jobs to specified threads	34
Figure 4.5 Device function to calculate the distances between cities	37
Figure 4.6 Inputs of occupancy calculator for problem with 500 nodes	44
Figure 4.7 Output of occupancy calculator for problem with 500 nodes	44
Figure 4.8 The effect of block size on occupancy	45
Figure 4.9 The effect of shared memory usage on occupancy	45
Figure 4.10 The impact of block size on occupancy for the problem with 1000 nodes	47
Figure 4.11 The effect of block size on occupancy in the problem with 1500 cities	48
Figure 4.12 The effect of block size on occupancy in the problem with 2000 cities	50
Figure 4.13 Storing the coordinates in the tour order	53
Figure 4.14 The modifications in the kernel code for big sized problems	53
Figure 4.15 Modification of the distance function for divided coordinates	57
Figure 4.16 Additional code in the host code to divide coordinates	58
Figure 4.17 Modification in kernel code for divided coordinates	59
Figure 4.18 3-opt exchange	61

LIST OF TABLES

Table 3.1 Relationship between the indices, thread id, block id and block dimension ..	14
Table 3.2 CUDA functions and their behaviors	17
Table 3.3 The features of a GPU device with compute capability 3.0	19
Table 3.4 Utilizing all possible warps in the streaming multiprocessor	21
Table 3.5 Features of CUDA variables	25
Table 4.1 Initial tour order in Example 4.1	29
Table 4.2 All possible edge exchanges for a TSP tour with 10 nodes	30
Table 4.3 Required indexes that will be produced by built-in variables in kernel	31
Table 4.4 Assigning jobs to threads	33
Table 4.5 Calculating the job ids using built-in variables and iterations	35
Table 4.6 Thread-level and iteration-level parallelism	38
Table 4.7 Predicted best kernel launches vs. observed best kernel launches for TSP with 500 cities	41
Table 4.8 Restrictions of shared memory and registers	42
Table 4.9 Predicted best kernel launches vs. observed best kernel launches for TSP with 1000 cities	46
Table 4.10 Occupancy information for the problem with 1000 nodes	47
Table 4.11 Occupancy information of the problem with 1500 cities	49
Table 4.12 Predicted best kernel launches vs. observed best kernel launches for TSP with 1500 cities	50
Table 4.13 Occupancy information of the problem with 2000 cities	51
Table 4.14 Predicted best kernel launches vs. observed best kernel launches for TSP with 2000 cities	51
Table 4.15 Comparing sequential and parallel 2-OPT algorithm performances	52
Table 4.16 Division scheme of coordinates for the problem with 9000 cities	54
Table 4.17 Division scheme of coordinates for the problem in Example 4.1	55
Table 4.18 Calculated edge exchange effects depending on the coordinate ranges sent to kernel (see Example 3.1)	56
Table 4.19 Calculation of sequential processes for several sized problems	57
Table 4.20 Predicted best kernel launches vs. observed best kernel launches for TSP with 6000 and 15000 cities	60

Table 4.21 Algorithm performances with a naive initial solution vs. with a good initial solution	61
Table 4.22 Possible edge exchanges for 3-opt.....	62
Table 4.23 Some problematic ids stem from the unrevised formula	63
Table 4.24 Results after fixing formula of “i”	64
Table A-1 2-opt performance for a TSP tour with 500 cities	67
Table A-2 2-opt performance for a TSP tour with 1000 cities	68
Table A-3 2-opt performance for a TSP tour with 1500 cities	69
Table A-4 2-opt performance for a TSP tour with 2000 cities	70
Table B-1 Best 2-Opt Results for Large-Sized Data	71
Table B-2 3-Opt Results for Different Sized Data	71

ABSTRACT

“ACCELERATING LOCAL SEARCH ALGORITHMS FOR TRAVELLING
SALESMAN PROBLEM USING GPU EFFECTIVELY”

GİZEM ERMİŞ

M.Sc. ‘Thesis’, July 2015

Prof. Dr. BÜLENT ÇATAY

Keywords: GPU computing, parallelization, optimization, GPU architecture, TSP

The main purpose of this study is to demonstrate the advantages of the GPU usage to solve computationally hard optimization problems. Thus, to solve the Travelling Salesman Problem, 2-opt and 3-opt methods were implemented in parallel. These search techniques compare every possible valid combination of the certain exchange system. It means that large numbers of calculations and comparisons are required. Through the parallelization of these methods via the GPU, performance has increased remarkably compared to performance in the CPU. Because of the distinctive manner of work and the complicated memory structure of GPU, implementations can be tough. Imprecise usage of GPU causes considerable decrease in the performance of the algorithm. Therefore, in addition to comparisons between GPU and CPU performances, the effect of GPU resource allocations on the GPU performance was examined. Allocating resources in different ways, several experiments on various sized travelling salesman problems were tested. According to the experiments, a technique was specified to utilize GPU resources ideally. Although GPU devices evolve day to day, some resources of them have still quite restricted capacity. For this reason, when it came to large scale problems, a special on-chip memory of the GPU device remained incapable. In order to overcome this issue, some helpful approaches were proposed. Basically, the problem was divided into parts. Parallelism was applied to each part separately. To sum up, the aim of this research is to give some useful insights about effective GPU usage and making researchers in the optimization area familiar with it.

ÖZET

“GRAFİK İŞLEMCI BİRİMİNİN ETKİN KULLANIMIYLA GEZGİN SATICI PROBLEMİ İÇİN YEREL ARAMA ALGORİTMALARININ HIZLANDIRILMASI”

GİZEM ERMİŞ

Yüksek Lisans Tezi, Temmuz 2015

Prof. Dr. BÜLENT ÇATAY

Anahtar sözcükler: Grafik İşlemci Birimi ile programlama, paralelleştirme, optimizasyon, Grafik İşlemci Birimi mimarisi , gezgin satıcı problemi

Çalışmanın temel amacı NP-zor optimizasyon problemlerini çözmede Grafik İşlemci Birimi kullanımının avantajlarını göstermektir. Bu nedenle, gezgin satıcı problemini çözmek üzere 2-opt ve 3-opt yöntemleri paralel olarak uygulanmıştır. Yöntemler belirli bir değişim sisteminin tüm geçerli kombinasyonlarını karşılaştırmaktadır. Bunun anlamı çok fazla sayıda hesaplama ve karşılaştırma işlemine ihtiyaç duyacak olmalarıdır. Bu yöntemlerin Grafik İşlemci Birimi aracılığıyla paralelleştirilmesiyle, Merkezi İşlemci Biriminin performansı ile karşılaştırıldığında performans önemli ölçüde artmıştır. Grafik İşlemci Biriminin kendine özgü çalışma tarzı ve karmaşık mimari yapısı nedeniyle, uygulamalar zorlu olabilmektedir. Grafik İşlemci Biriminin özensiz kullanımı algoritmanın performansında kayda değer bir azalışa yol açabilir. Bu nedenle, Grafik ve Merkezi İşlemci Birimi performanslarının karşılaştırmalarına ek olarak, Grafik İşlemci Biriminin kaynak tahsisinin işlemci performansındaki etkisi de incelenmiştir. Kaynaklar farklı yollarla paylaşılırak, çeşitli büyüklükteki gezgin satıcı problemleri üzerinde birtakım deneyler test edilmiştir. Deneylere göre Grafik İşlemci Birimi kaynaklarını ideal olarak paylaşmak için bir yöntem belirlenmiştir. Grafik İşlemci Birimleri günden güne evrilmesine rağmen, bazı kaynakları hala oldukça sınırlı kapasiteye sahiptir. Bu sebeple, uygulama sırasında söz konusu büyük boyutlu problemler olduğunda, Grafik İşlemci üzerindeki özel bir bellek yetersiz kalmıştır. Sorunun üstesinden gelmek için, bazı yararlı yaklaşımlar önerilmiştir. Temel olarak, problem parçalara ayrılmıştır. Parallelleştirme işlemi her parçaya ayrı ayrı uygulanmıştır. Özetleyecek olursak, bu araştırmanın amacı Grafik İşlemci Biriminin etkin kullanımıyla ilgili faydalı bilgiler vermek ve optimizasyon alanındaki araştırmacıların bu konuya aşına olmalarını sağlamaktır.

<< *To my irreplaceable family* >>

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof. Dr. Bülent Çatay for the continuous support of my research. I have many reasons to appreciate him. In addition to his encouragements to begin this study, during the research he gave me inspirational ideas and shared his immense knowledge with me. His guidance was admirable and I could not have imagined having a better advisor and mentor for my master study.

Besides my advisor, I would like to thank the rest of my thesis committee Asst. Prof. Dr. Kamer Kaya and Asst. Prof. Dr. Gürkan Öztürk for their insightful comments, encouragement and valuable feedback.

As my precious family never left me alone in my hardest times, they also strongly supported me through my research. I feel quite lucky to have such supportive, helpful and lovely family members. I never can thank them enough.

I am also grateful to my special friends in the IE lab. Murat, Sonia, Ece, Bahar, Başak, Burcu, Özgün, Ameer, İhsan, Menekşe and Yağmur. They kept me motivated all the time. Moreover I appreciate to my true friends Tuba, Esra, Çağrı and Hatice for their endless support, friendship and motivation.

1 INTRODUCTION

Optimization problems have maintained their importance in many areas such as industry and the public sector. Efficiency is as much critical factor as solution quality when an optimization algorithm is written. Algorithms of optimization methods such as 2-opt or 3-opt are computationally difficult when they are solved via a CPU. A qualified parallelism can accelerate these kinds of algorithms considerably. While restricted parallelism can be managed via central processing units (CPUs), the modern graphical processing units (GPUs) can provide much more parallelism through their highly parallel structure. Thus they can considerably reduce the execution time of algorithms by performing a wide range of calculations at the same time, in other words in a parallel manner.

In the hardware structure of a computer, the task of reading and executing program instructions belongs to a processor which is a chip in computers. These instructions notify the processor what to do such as reading data from memory or sending data to an output bus. CPU is a common type of processor (Prinslow and Jain, 2011). The processor core or briefly “core” is an individual processor and a modern processor can have multi or many cores.

Modern GPUs are many-core processors that are specifically designed to perform data-parallel computation. Data parallelism means that each processor performs the same task on different pieces of distributed data (Brodtkorb et al., 2013). This data parallel framework of GPUs is referred to as “single instruction multiple data (SIMD).

Before the evolution of nowadays’ advance GPUs, traditional, single-core processors were exploited. A single core processor could provide only concurrency through the “multithreading”, but no parallelism. Multithreading handles the concurrent execution of different parts of the same program and each of these parts referred to as thread. However, it is not possible to execute different tasks or programs in a parallel way via

one single-core processor. There is a crucial fundamental difference between concurrency and parallelism. “In a multithreaded process on a single processor, the processor can switch execution resources between threads, resulting in concurrent execution.” It means that although single-core processors can normally execute one thread at a time, via multithreading the processor can switch between threads, which is that while one of the threads in the program was waiting another thread can execute, giving the impression that threads are running concurrently. “In the same multithreaded process in a multiprocessor environment, each thread in the process can run on a separate processor at the same time, resulting in parallel execution (Oracle, 2010).”

Computationally hard tasks such as solution of optimization problems were taking a great deal of time when they were solved by the help of single-core processors. Faster and faster single-core processors were developed by computer industry, but they were still insufficient for peak performances. Because it was difficult to accelerate individual processors/cores further but possible to provide more processing power by putting more cores onto a single chip/processor, around the year 2000, by fitting more cores in the same chip, single-core processors evolved to multi-core processors (Figure 1.1), which work together to process instructions and thus have higher total theoretical performance (Brodtkorb et al., 2013) (Oxford). Multi-core processors, which have two or more independent processors, achieved greater performance through parallelism rather than shortening the completion period of an operation via higher clock speed, in other words accelerating individual processors. These multi-core CPUs were efficient at task parallel implementations. Consequently the sequential software started to lose its prestige and via multiple CPU cores task parallel implementations were applied to computationally hard tasks.

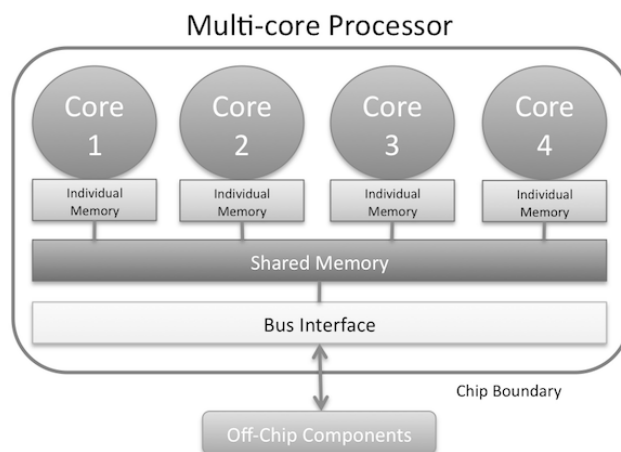


Figure 0.1 A basic block diagram of a generic multi-core processor

After some time, because of gaming industry needs, GPUs which actually were the normal component in common PCs, developed quickly in terms of computational performance. Multi-core GPU processors evolved to massive multi-core or many-core processors which work as massively parallel stream processing accelerators or data parallel accelerators. Because of the rapid advances of GPUs, they became common as accelerators in general purpose programming. Although both multi-core CPUs and GPUs can implement parallel algorithms, the architectural differences between CPUs and GPUs created different usage areas for them depending upon the nature of the problem. While multi-core CPUs are designed for task parallel implementations, many-core processors are specifically designed for data parallel implementations. Instead of distributing different tasks amongst individual processors, in data parallel computations the data is distributed (SIMD). Furthermore, CPU performance is better on latency-sensitive, partially sequential algorithms. However, GPU performance is better on latency-tolerant, highly-parallel algorithms (Prinslow et al., 2011). In other words, CPU aims to minimize the time of a single operation or minimize the latency of a single operation, although GPU tries to maximize the number of operations in unit of time or maximize throughput in per unit time. Lastly, compared to CPUs, GPUs have much more arithmetic logic units, which perform all arithmetic computations and comparison operations. Thus, via GPUs data parallel, throughput-oriented applications with intense arithmetic operations can be accelerated on a large scale. More extensive differences between the GPU and CPU architectures will be elaborated on the architecture part.

Nowadays, GPUs have many processors and with the help of these processors GPU performance can be much better than CPU performance in some specific problems, especially in computational problems. Of course the increase in the number of processors created a need for simpler processors than previous ones, which we will elaborate on the Architecture part. Because of these simpler GPU processors and the limited structure of GPUs with data-parallel computation, it is difficult to solve an entire problem via GPUs. Thus, to benefit from GPU, we do not necessarily have to choose a completely parallelizable problem. It is quite sensible to take advantage of GPU technology in the convenient part of the solution method and continue to use CPU for remaining parts. In other words an algorithm starts at the CPU and whenever data parallelism can be managed the data is sent to the GPU and computations are made in parallel there. This is called general-purpose computation on GPUs (GPGPU) and also heterogeneous programming.

In order to observe the advantages of GPU usage in solving computationally expensive optimization problems, we applied the parallel 2-opt and 3-opt local search methods for the Travelling Salesman Problem (TSP) which are proposed by Rocki and Suda (2012, 2013). The 2-opt technique depending on the best improvement searches for all the possible swaps in a route and the aim is finding the swap that will decrease the tour cost most. The method that we applied is a complete 2-opt local search will compare every possible valid combination of the swapping mechanism (Wikipedia). This is why these methods take a great deal of time when the CPU is used unless the data is not too small. As can be realized, 2-opt and 3-opt methods are quite suitable for adapting to the SIMD architecture of the GPU. Possible swaps will have different effects on the tour cost and to calculate these effects the same formula will be used. In this situation, possible swaps can be thought as multiple data and the formula can be thought as single instruction. Thus, to calculate the effect of each possible swap, by applying parallelism through the GPU, we can obtain significantly better results in terms of computation time compared to that of the CPU. Moreover, it is possible to produce accelerations in GPU algorithm time by using its memory more efficiently.

The main reason for this research is to provide insights into powerful usage of GPUs, building efficient techniques, sharing some useful experimental results and sighting the

advantages of GPU usage. Furthermore, restrictions of GPUs and strategies to overcome these restrictions will be mentioned. We aim to encourage researchers who are interested in optimization problems to benefit from the advantages of GPUs.

2 LITERATURE REVIEW

Local search is a fundamental algorithm in optimization problems. This algorithm generates several candidate solutions in the defined neighborhood to improve the current solution and then picks the best or an improving one among them. This process continues until there is no further improvement for the current solution. Because the evaluation of the neighborhood is quite suitable to be performed in parallel, local search algorithms can be accelerated for problems with large neighborhood substantially.

Up to now, the researchers performing local searches through GPU generally reported the speedups in comparison to CPU. During GPU implementations, performance analysis and improvement of system performance is fairly important to provide effective utilization of GPU resources. Schulz (2013) accelerated the naive GPU algorithm using profiling tools and saturating device fully. According to the study of Schulz, to saturate the GPU a large enough problem instance is required. Schulz achieved a speedup of almost an order of magnitude compared to the Benchmark Version. Burke and Riise demonstrated that the evaluation of the entire neighborhood to discover the best improvement can display better performance than applying the first improvement.

The first research applying some kind of local search to routing problems via GPU belongs to Janiak et al. (2008). Janiak presented the implementation of a tabu search algorithm for TSP and flow shop scheduling problem. After CUDA was introduced, performing local search methods in GPU became much easier. To solve TSP problem Luong et al. (2009) used GPU as a coprocessor for extensive computations which is evaluating each solution from a given 2-exchange (swap) neighborhood in parallel. Remaining computations were done in CPU.

A local search has four main steps which are neighborhood generation, evaluation, move selection and solution update. The simplest method is to create the neighborhood on the CPU and transferring it to GPU each time. Luong et al. (2011b) applied this technique which requests copying of a lot of information from the CPU to the GPU. Other way is to generate neighborhood in GPU.

To evaluate the neighborhood the common method used is to assign one or several moves to a thread which is called mapping. Luong et al. (2011b), Burke and Riise (2012) Coelho et al. (2012), Rocki and Suda (2012), Schulz (2013) utilized an explicit formula to provide mapping. Luong et al. (2011b) used an algorithm. The mapping approach, which is done in GPU, removes the need for copying some information from CPU to GPU.

As neighborhood evaluation is the most computationally expensive task, it was generally performed on the GPU. However, choosing the best move may not be performed on the GPU.

Luong et al. (2011b), O'Neil et al. (2011), Coelho et al. (2012), Rocki and Suda(2012), Schulz (2013) presented some implementation details in order to execute kernel efficiently. Among them the only one who demonstrated the profiling analysis of these details is Schulz. Moreover, because of the limited memory of GPU, for large neighborhoods Schulz proposed an implementation that divides the neighborhood in parts. More comprehensive review of GPU computing and its application to Vehicle Routing Problems can be found in the studies of Brodtkorb et al. (2013) and Schulz et al. (2013).

3 ARCHITECTURE

In order to comprehend the possible advantages of GPU usage in certain kinds of applications, firstly the main differences of GPU and CPU architecture should be understood.

GPUs and multi-core CPUs are specifically designed to perform different types of parallel computations. Although the design of CPU is optimized for partially-sequential code performance, GPU is optimized for highly parallel code execution. CPUs can be called as *latency-oriented* devices and GPUs are *throughput-oriented* devices. Latency is the amount of time to complete a task which is measured in units of time, like seconds. Throughput is tasks completed per unit time and it is measured in units as stuff per time, like jobs completed per hour. While CPU aims to minimize latency, GPU tries to maximize throughput.

The main components of a regular processor are arithmetic logic units (ALU), control unit, cache and DRAM. The main difference between GPUs and CPUs is that GPUs devote *proportionally* more transistors to arithmetic logic units (ALU) and less to caches and flow control in comparison to CPUs. As mentioned in the introduction, all arithmetic computations such as multiplication, addition and also comparison operations are performed by ALUs. GPUs also typically have higher memory bandwidth compared to CPUs (Oxford).

As seen in the Figure 3.1, CPUs have larger local cache than GPUs. Cache memory is random access memory (RAM) that a computer microprocessor can access more quickly than it can access regular RAM and it reduces the instruction and data access latencies of large complex applications. Moreover CPUs have more sophisticated control logic in contrast to GPUs. These control logic provides to reduce arithmetic calculation latency and memory access latency.

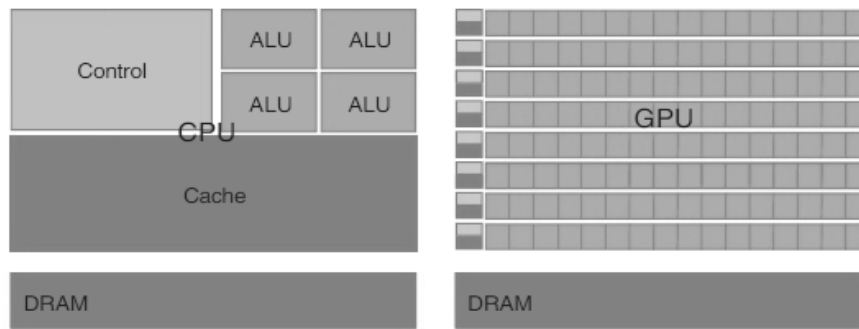


Figure 0.2 The architectural difference between CPU and GPU
(Kirk and Hwu, 2013)

Control logic and cache memories do not help to reach the peak calculation speed because large cache memory and sophisticated control logic consume chip area and power considerably. By using smaller cache and simpler control logic it is possible to have more arithmetic execution units and memory access channels on chip. So the larger control logic and cache memory in CPUs are disadvantageous with regards to time performance of the whole algorithm. (Kirk and Hwu, 2013)

GPUs aim to maximize chip area and power budget dedicated to floating point calculations. Compared to GPUs, CPUs have very powerful arithmetic control units (ALU) that can generate arithmetic results in very few clock cycles which requires more energy. The power of the CPU ALUs stems from sophisticated control unit and big control cache in the CPU architecture. Because ALUs in CPU are quite powerful, they have extremely short latency for producing floating arithmetic operations. However, GPUs have great numbers of energy efficient ALUs which have long latency but heavily pipelined for high throughput. Pipelining helps microprocessor to begin executing a second instruction before the first one has been completed. (Figure 3.2) It means that completion of one operation takes more time, but the total time to complete all operations can be shorter than in that of CPUs if the large number of ALUs (so many threads) in GPUs can be fully utilized. In other words, in GPU system overall throughput is improved, even though the execution of each individual thread is degraded.

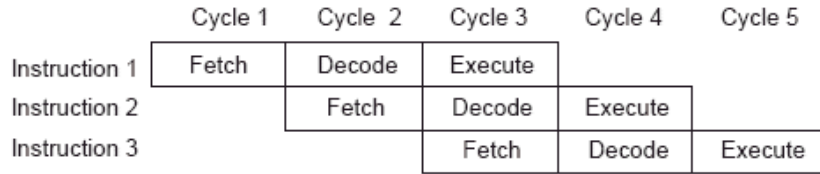


Figure 0.3 Pipelining

As discussed earlier, CPU hardware reduces the execution latency of each individual thread by reducing the latency of operations, while GPU has long latency for a single thread as it uses simpler control logic and smaller cache. In order to tolerate these latencies, massive numbers of threads are required like GPUs have. Through these massive numbers of parallel threads in GPU, the total execution throughput is maximized although individual threads take much longer time than in CPU.

To sum up, the design of GPU saves chip area and power by allowing pipelined memory channels and arithmetic operations to have long latency. As the power and area of the cache, control and individual arithmetic logic unit (ALU) were reduced in the design of GPU, more memory access units and arithmetic units could be used on a chip and this kind of a design increased the total execution throughput. In the working system of GPU, as some of the threads should wait for long latency memory access or arithmetic operations, it is more advantageous to use large number of parallel threads to compensate the waiting time. Otherwise GPU usage can be meaningless. In GPU architecture a small cache memory is provided for each set of multiple threads that access the same memory data. In this way, instead of going to DRAM these multiple threads can go the cache, which takes much shorter time (Kirk and Hwu, 2013).

As mentioned earlier, CPUs minimize the execution latency of a single thread while GPUs maximize execution throughput of all threads. So it can be said that CPU and GPU have different advantages. By using CPUs for sequential parts of the algorithm where latency matters and GPUs for parallel parts where throughput wins, the optimal algorithms can be achieved. This way of programming is called as heterogeneous programming. In our research, accelerated 2-opt and 3-opt algorithms were investigated which were written by utilizing heterogeneous programming. CUDA C language which supports the heterogeneous programming was used. CUDA C is very similar to regular

C language, additionally it has a kernel function which exploits parallelism and some additional functions that provide kernel launch and communication between CPU and GPU.

A CUDA program has two main components. First one is host code which runs locally on CPU and second one is GPU kernel code which is a GPU function that runs on GPU device. A heterogeneous code starts on a CPU host and when parallelization is needed the host code invokes a GPU kernel on a GPU device (Cornell Workshop, 2015).

As seen in the Figure 3.3, kernels have a grid structure which has lots of thread blocks and these thread blocks have lots of threads which exploit parallelism. Through these threads, different parts of the data can be processed independently of each other as parallel. After kernel finishes its execution, the CPU continues to execute the original program. In order to use GPU, firstly a device should be initialized and GPU memory should be allocated in host code. Then the data that will be made parallel should be transferred to the device from the host and kernel should be invoked. Invoking kernel is like calling a function. Differently from C, the kernel functions take configuration parameters or arguments (Hwu, 2015). These configuration parameters consecutively represent number of blocks in the grid and number of threads in a block. After kernel finishes its parallel processes, the result should be transferred from device to host if it is needed.

Each thread in GPU can be thought as a virtualized Von-Neumann processor. Thus, every CUDA thread can execute a program. As mentioned before, the GPU memory has lots of threads, in other words lots of processors and the kernel function is executed by them. Because of the SIMD structure of GPU, all threads in a grid run the same kernel code. To specify memory addresses and make control decisions, each thread has its own indexes, in other words each thread has a unique thread ID. These indices are used by threads in order to decide what data to work on (Hwu, 2015). The threads and blocks have a 3-dimensional structure to ease parallelism of some specific problems. It simplifies memory addressing when processing multidimensional data (Hwu, 2015). However it is not an obligation to use all the dimensions. While applying two

dimensions is a better way in a matrix multiplication, in our study utilizing only one dimension is more appropriate.

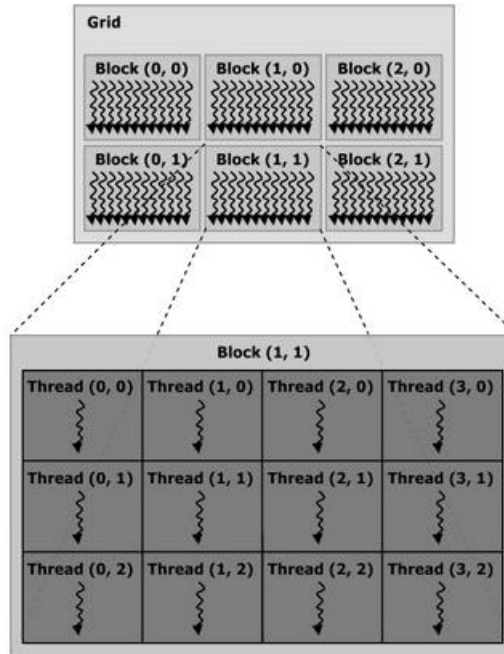


Figure 0.4 The thread hierarchy in the CUDA programming model (Virginia Tech)

In order to access the indexes of threads, CUDA has specific predefined variables such as “threadIdx.x”, “blockIdx.x”, “blockDim.x”, “gridDim.x”, which represent “x” dimension. As we will utilize only one dimension in our problem, we will not emphasize “y” and “z” dimensions. If the number of threads in a block is represented by “n”, which refers to as “block dimension (blockDim.x)”, each thread will have different indexes from “0” up to and including “n-1” in that block. In other words, starting from 0 “threadIdx.x” counts the threads in a block one by one. Through “threadIdx.x” these indexes can be assigned to a variable in the device. “blockDim.x” takes the size of a block which is the number of threads in a block. Like threadIdx.x”, “blockIdx.x” counts the number of blocks in the grid one by one, in a sense it gets the indexes of blocks from 0 up to the specified number of blocks in a grid. Lastly “gridDim.x” represents the size of a grid, in other words number of blocks in a grid.

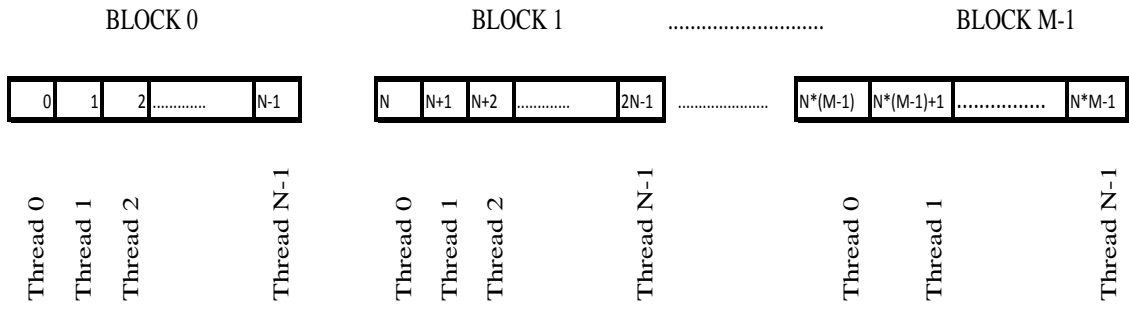


Figure 0.5 The indexes produced by kernel depending on the number of blocks launched in the grid and the number of threads launched in the block

Let's assume that there are N threads in a block and M blocks in the grid. (Figure 3.4) In this situation every thread has a thread index and also a block index which are specified by "threadIdx.x" and "blockIdx.x" consecutively. These are predefined CUDA variables that can be used in a kernel and they actually are initialized by the hardware for each thread like below (Hwu, 2013):

```

threadIdx.x = 0,1,2,3,4.....(N-1)
blockDim.x = N
blockIdx.x = 0,1,2,3,4.....(M-1)

```

"blockDim.x" helps to factor in both the thread index and the block index. In order to obtain all the thread indexes, the block index (blockIdx.x) should be multiplied by the block dimension (blockDim.x) and added to the thread index (threadIdx.x) such as " $\text{blockDim.x} \cdot \text{blockIdx.x} + \text{threadIdx.x}$ ". Via that formula all the indexes of all the threads in Figure 3.4 will be initialized by the system like in Table 3.1. Third and forth columns in this table shows how block ids and thread ids in each block are automatically initialized when kernel launches M blocks including N threads. Second column shows the size of a block. Depending on these built-in variables, indexes specific to each thread in the grid are calculated in the first column.

Table 0.1 The relationship between the indices, thread id, block id and block dimension

indexes (blockDim.x*blockIdx.x+threadIdx.x)	blockDim.x	blockIdx.x	threadIdx.x
0	N	0	0
1	N	0	1
2	N	0	2
..	N	0	..
N-1	N	0	N-1
N	N	1	0
N+1	N	1	1
N+2	N	1	2
..	N	1	...
2*N-1	N	1	N-1
...
...
...
N*(M-1)	N	M-1	0
N*(M-1)+1	N	M-1	1
N*(M-1)+2	N	M-1	2
..	N	M-1	..
N*M-1	N	M-1	N-1

As shown in the Figure 3.4 and in the Table 3.1, from the formula “ $\text{blockDim.x} * \text{blockIdx.x} + \text{threadIdx.x}$ ”, thread 0 in block 0 has the index of 0 as the block index is 0. However thread 0 in block 1 has the index of “N” instead of “0”, as the block index is 1. Then thread 0 in the next block will have the index of “2N”. Consequently the index values of the first block will range from “0” up to and including “N-1”, the index values of the second block will range from “N” up to and including “2N-1” and the index values of the last block will range from “N*(M-1)” up to and including “N*M-1”. All the indexes from 0 up to and including N*M-1 can be obtained in this way, as we have “M” blocks and “N” threads in each block (M*N threads totally). Threads within a block can cooperate via shared memory, atomic operations and barrier synchronization although threads within different blocks cannot interact (Hwu, 2013). This subject will be elaborated later.

Of course there are some restrictions in parallelism because of the GPU design. State of the art GPU cards allow to use maximum 1024 threads in a block and $2^{31}-1$ blocks in the grid.

3.1 Device Memories and Data Transfer

As discussed previously, before the kernel invocation the GPU memory should be allocated and then the necessary data should be moved from CPU (host memory) into GPU (device memory) via API (application programming interface) functions so that the device can be ready to process the data.

API functions are programming interface functions in CUDA host code. In industry standard programming languages are extended via APIs. In order to help C programmers to use GPUs in a heterogeneous environment, CUDA designers and NVIDIA proposed some API functions (Hwu, 2013). These API functions provide the communication and integration between CPU and GPU. Two main API functions are “device memory allocation” and “host-device data transfer” functions.

A conceptual understanding of CUDA memories is necessary in order to understand how API functions work (see Figure 3.5) It is known that device has great numbers of threads and each of these threads is actually a processor. Therefore each thread has registers as displayed in Figure 3.5 and these registers hold variables that are private to the thread (Hwu, 2015). In this figure, global memory is the memory that all threads can have access. The “device memory allocation” functions are specialized functions that allocate global memory. On the other hand, “host-device data transfer” functions copy the data from the host memory to the global memory and from the global memory to the host memory. These API functions should be defined in the host code.

The specific expression of “device memory allocation” function is “`cudaMalloc()`” which allocates object in the device global memory. It has two parameters. The first parameter specifies the address of a pointer to the allocated object and the second one shows the size of allocated object as bytes.

The second function is called “cudaMemcpy()” which is a “host-device data transfer” function. “cudaMemcpy()” helps to transfer the data from host memory to device memory and vice versa. It has four parameters which are consecutively pointer to the destination, pointer to the source, the size of the data to be copied as bytes and the direction of the transfer (host to device or device to host).

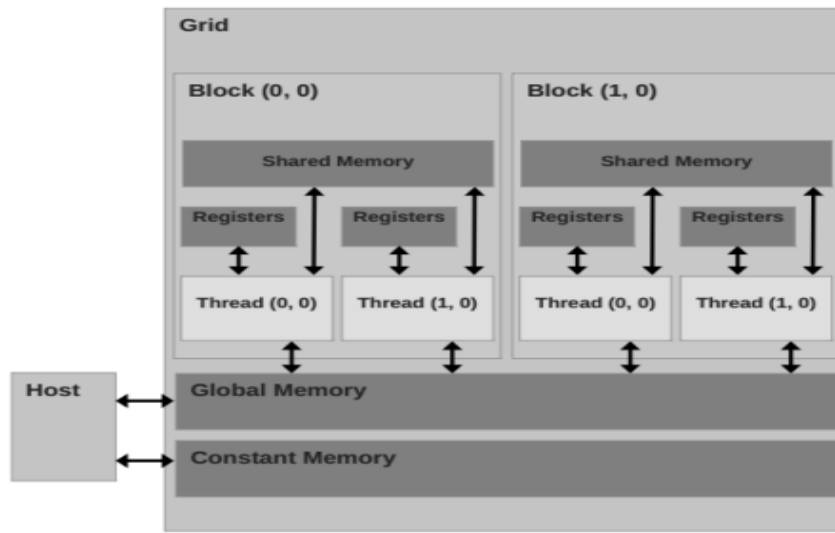


Figure 0.6 Overview of CUDA device memory model (Kirk and Hwu, 2013)

After device memory allocation and data transferring from host to device, the kernel function can be invoked. In addition to regular features of C functions, CUDA kernel functions should be preceded by “__global__” keyword so that compiler can understand that it is a kernel function. Launching the kernel function differs from calling a traditional C function. It has special parenthesis syntax of “<<< >>>” between the name of the function and the parameters of the function. This special parenthesis includes two configuration parameters for the kernel. The first one is the number of blocks in the grid and the second one is the number of threads in a block.

During the configuration of the kernel, the important point is determining the number of required threads according to the solution method. When the kernel is launched in the host code, the kernel function is called and the hardware produces a grid of threads according to the configuration parameters like in Figure 3.6.

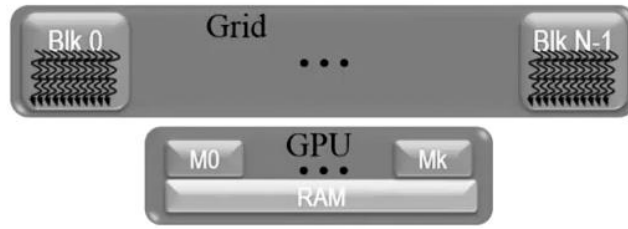


Figure 0.7 The grid of threads produced because of the kernel launch (Hwu, 2013)

As mentioned earlier each thread in the grid has the built in variables `blockIdx.x`, `blockDim.x` and `threadIdx.x`; these predefined variables allow threads to generate different data indices so that each thread can process a different part of the data. In addition to these functions it should be known that from kernel or from other device functions only the device functions can be called and these functions should be preceded by “`__device__`”.

The whole CUDA function types are described in the first column of Table 3.2. The second and third columns consecutively present the places that these functions are executed and called from. Host functions are actually functions in CPU. They are called from the host and also executed in the host. “`__global__`” defines a kernel function and kernel function has to return “`void`”. Although kernel functions are called from the host they are executed on the device. Lastly, “`__device__`” defines device function which is called from the device and executed on the device.

Table 0.2 CUDA functions and their behaviors

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

3.2 Thread Scheduling and Latency Tolerance

Up to now, the basic concepts of GPU and the mechanism of a CUDA program are discussed. In order to reach peak calculation speeds the resources of GPU should be utilized carefully, rather than using the advantages of GPU randomly. It should be

ensured that hardware execution resources are utilized efficiently. In order to manage this, number of blocks and number of threads should be specified according to the structure of the execution resources of GPU, in other words CUDA thread blocks should be assigned to execution resources efficiently. The capacity constraints of execution resources should be considered and zero-overhead thread scheduling should be provided to tolerate the latencies in individual threads (Hwu, 2013).

In order to perform thread scheduling properly, firstly the features of the GPU card should be investigated. GPU cards maintain to evolve, their qualifications change and improve in time. Current GPU technology is much better than before and in the future most probably it will be much better than today.

In our study, one of the best GPU cards which named as Quadro K600 is used and its compute capability is 3.0. Compute capability shows the general specifications and features of a compute device (Nvidia, 2015). Table 3.3 presents the features of a device with compute capability 3.0. The capacity constraints of execution resources depend on the type of GPU device. As illustrated in table, our device can have maximum 1024 threads in a block and “ $2^{31}-1$ ” blocks in the grid.

As mentioned before, when a kernel is launched CUDA system produces equivalent grid of threads and assigns them to execution resources. The execution resources in GPU hardware are organized into streaming multiprocessors (SM) (Kirk and Hwu, 2013). Streaming multiprocessors executes the threads in block granularity. All the threads in a block are assigned to the same SM. Quadro K600 has a very efficient and advance multiprocessor, which specifically named as “SMX” (Figure 3.8). SMXs have also some resource limitations. From Table 3.3 it can be seen that each SMX can have maximum 16 resident blocks and 2048 resident threads.

When the CUDA system assigns a block to a streaming multiprocessor, this block is divided into 32 thread units which is called warps. In other words, in CUDA each block is executed as warps and each warp has 32 parallel threads. Each warp has sequential indexes, for example the first warp has the indexes from 0 up to 31, the second one has the indexes from 32 up to 63 etc. In each warp the same instruction is executed. When

an instruction in a warp should wait for a result of a previous long-latency operation, this warp cannot be executed. While this warp stalls, another ready warp is selected to be executed. (see Figure 3.7) This process of tolerating the latency arising from the long-latency operations with other group of threads is called latency hiding (Kirk and Hwu, 2013). Consequently, by providing enough active warps, latency hiding can be managed as the hardware can find a warp for execution at any time rather than waiting for the busy warps. The GPU hardware doesn't waste time while choosing the ready warps, for this reason it is called zero-overhead thread scheduling. As seen in the Table 3.3, the restriction about warps is that one SMX can have maximum 64 warps. As seen in the Figure 3.8, one SMX has 4 warp schedulers which allow 4 warps to be executed parallel.

Utilizing great numbers of warps can be the one way of achieving enough parallelism and increasing the performance, but not necessarily. This kind of parallelism is called "thread level parallelism". Thread level parallelism is assessed by the "occupancy" which is the number of active warps over the maximum number of active warps supported on one SM. Thus, increasing the occupancy can provide thread level parallelism. Maximum number of active warps per multiprocessor is 64 in our device, in addition to the previous restrictions. By considering these memory restrictions, different combinations of block and grid dimensions can be exploited to increase occupancy. Table 3.4 demonstrates the configuration parameters when all warps are utilized on the device. First two columns consecutively show number of threads in a block and number of blocks in a grid. Depending on the block dimension, the number of active warps in each block is calculated in the third column and multiplying number of active warps in each block by number of blocks in the grid the number of active warps in the system is calculated in the last column.

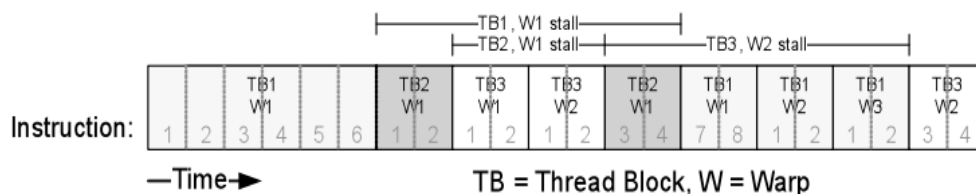


Figure 0.8 Warp scheduling (Cooper, 2011)

Table 0.3 The features of a GPU device with compute capability 3.0

Technical Specifications	Compute Capability 3.0
Maximum dimensionality of grid of thread blocks	3
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$
Maximum y- or z-dimension of a grid of thread blocks	65535
Maximum dimensionality of thread block	3
Maximum x- or y-dimension of a block	1024
Maximum z-dimension of a block	64
Maximum number of threads per block	1024
Warp size	32
Maximum number of resident blocks per multiprocessor	16
Maximum number of resident warps per multiprocessor	64
Maximum number of resident threads per multiprocessor	2048
Number of 32-bit registers per multiprocessor	64 K
Maximum number of 32-bit registers per thread block	64 K
Maximum number of 32-bit registers per thread	63
Maximum amount of shared memory per multiprocessor	48KB
Maximum amount of shared memory per thread block	48KB
Number of shared memory banks	32
Amount of local memory per thread	512KB
Constant memory size	64KB
Cache working set per multiprocessor for constant memory	8KB
Cache working set per multiprocessor for texture memory	Between 12 KB and 48 KB

Table 0.4 Utilizing all possible warps in the streaming multiprocessor

Block Dimension (threads in a block)	Grid Dimension (number of blocks)	Number of Warps in a Block	Number Of Warps in a SM
1024	2	$1024/32=32$	$32*2=64$
512	4	$512/32=16$	$16*4=64$
256	8	$256/32=8$	$8*8=64$
128	16	$128/32=4$	$16*4=64$

To sum up, the resource restrictions are like following:

- Number of threads in a block can be maximum 1024.
- A Streaming Multiprocessor can have maximum 16 blocks.
- Blocks are divided by warps. One warp has 32 threads. As a result, one block can have $1024/32=32$ warps most.
- One SM can have maximum 64 warps.

As seen from Table 3.4;

- Number of threads in a block is less than or equal to 1024. Also it is divisible by the size of a warp which is 32.
- Number of blocks in a grid, i.e., in a streaming multiprocessor, is less than or equal to 16.
- As one warps contains 32 threads, to find number of warps in a block number of threads in a block should be divided by 32. For each combination number of warps in a block is less than or equal to 32 in the table.
- Number of warps in a SM is found multiplying number of warps in a block by number of blocks in a SM. In the table, number of warps in a SM is equal to 64 which means that all warps in a SM are utilized.

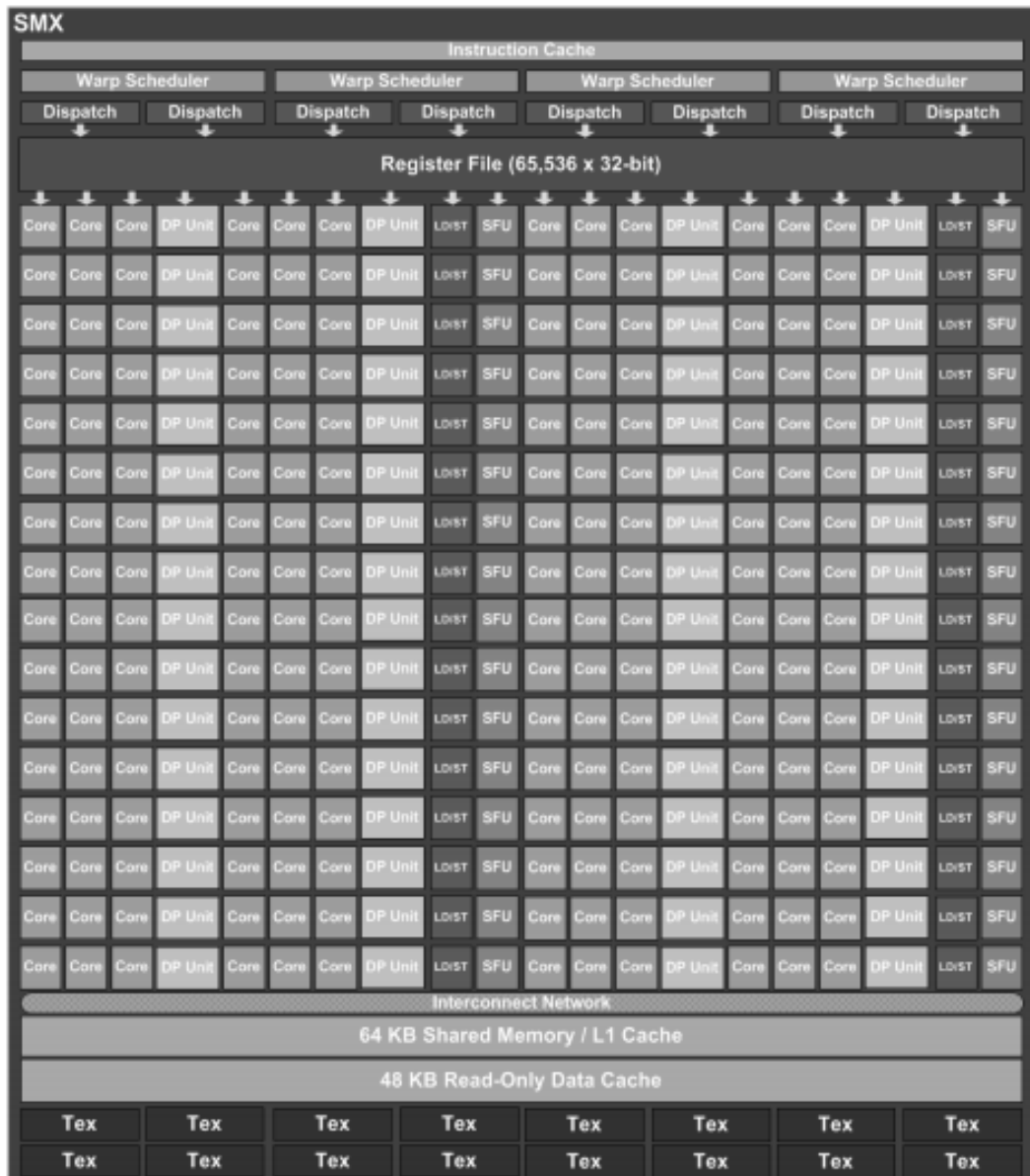


Figure 0.9 Streaming multiprocessor structure of the GPU device (Nvidia, 2012)

In addition to thread level parallelism, instruction level parallelism can be applied to expose enough parallelism and achieve good performance. In instruction level parallelism an individual thread executes concurrent operations, while independent and parallel operations are assigned to different threads in thread level parallelism. In other words, parallel tasks are executed by different threads in TLP (see Figure 3.9) and parallel tasks are executed by one thread in ILP (see Figure 3.10).

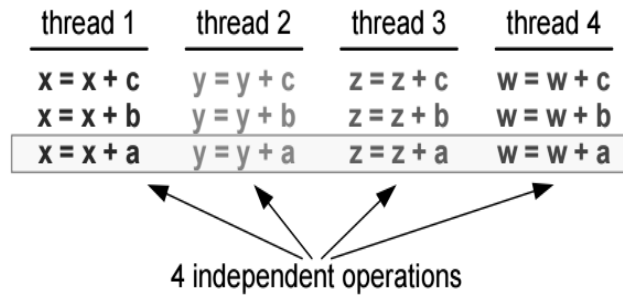


Figure 0.10 Thread level parallelism (Volkov, 2010)

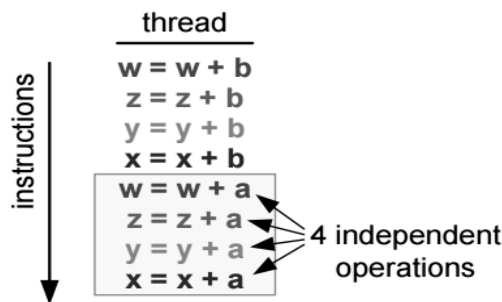


Figure 0.11 Iteration level parallelism (Volkov, 2010)

Although thread level parallelism is a good way of increasing performance, the limitations in kernel resources may prevent hiding latencies at some point. When resource consumption of a kernel is too large, it restricts the number of concurrent threads on a streaming multiprocessor. In this situation, instruction level parallelism can be applied or instruction level parallelism and thread level parallelism can be combined. In some cases low thread level parallelism with higher instruction level parallelism may exploit better performance as lower occupancy increases the number of registers per thread. However, the register pressure also increases. Because all the loads are grouped or batched together through a thread-private array in register memory in addition to having each thread execute multiple concurrent operations. Thread private arrays consume registers and may further add to register pressure (Ruetsch and Fatica, 2013). Thus, determination of how much thread level or instruction level parallelism will produce optimal results depends on type of the problem. In this research, certain experimental studies were done about this subject using 2-opt TSP problem that will be elaborated on the experimental design section.

3.3 Memory Model and Locality

It is possible and important to manage scalable parallel programs via CUDA. A scalable system is a system whose performance improves after adding hardware, proportionally to the capacity added. If a system, algorithm or program maintains its efficiency and practicability when applied to large instances, it is said to scale.

Quickly memory access is a critical factor for a scalable and parallel execution. It is very important to be careful about using the different memory parts of the GPU efficiently. In addition to global memory which is mentioned before, shared memory and registers will be introduced in this chapter. In Figure 3.5, different CUDA memories can be observed. In this figure, 2 blocks and 2 threads in each block are representatively demonstrated. Normally a grid has the capability of including a lot more blocks and threads.

In Figure 3.5, registers and shared memory are called on-chip memories as they are situated in GPU device. Variables of these memories can be accessed quite quickly and in a highly parallel way. CUDA memory types specify the visibility of a variable in addition to its access speed.

As discussed earlier, host code copies the data into the global memory and out of the global memory through “`cudaMemcpy`”. All the threads in a grid/kernel can access to the global memory. Thus, all the threads in a grid can see the contents of the global memory. In CUDA another memory level is “registers” which are generally used for frequently used variables. Each thread in the grid has a certain number of registers to hold its private variables. The variables that are placed into registers by the respective thread of these registers can only be visible to that same thread. Other threads in the grid cannot identify the value of these variables. Next memory that we will talk about it is shared memory. All the blocks in the grid can use shared memory. In shared memory, some locations are allocated to each block. When a block uses its allocated locations in the shared memory (i.e. own shared memory), all the threads in that block are able to see the contents of that locations of the shared memory. Nonetheless, the threads in other blocks cannot see the contents of these locations. Although each block can read

from and write to their own shared memory, the data in the shared memory of different blocks cannot be visible to each other.

Table 0.5 Features of CUDA variables

Variable declaration	Memory	Scope	Lifetime
<code>intLocalVar;</code>	register	thread	thread
<code>__device__ __shared__ intSharedVar;</code>	shared	block	block
<code>__device__ intGlobalVar;</code>	global	grid	application
<code>__device__ __constant__ intConstantVar;</code>	constant	grid	application

In Table 3.5 some features of CUDA variables are demonstrated. It shows the memory that each variable occupies, the scope of them and how much they exist. In order to use registers for a CUDA variable, we can declare that variable as an automatic variable in the kernel function or device function. In a kernel function all the variables that are declared like in the traditional C function become register variables. Thus, these variables are located into the registers and the scope of a variable is within one thread. Each thread in the grid will have the different/own version of that variable. When a thread changes the value of its private variable, other threads cannot see that modification. Moreover, the lifetime of a register variable is same with the life of a thread. It means that the register variable destroyed when its thread finishes execution. In Table 3.5, the second row shows the declaration of the shared memory variables. When the identifier “`__device__ __shared__`” is seen in front of a variable, it means that this variable will be placed into shared memory. As the scope of the shared memory variable is within one block, a variable that is declared for a block will only be detectable for the threads in that block. However, each block will have the different/own version of that shared memory variable. Thus, the contents of the variables in one block will not be visible to other blocks in shared memory. Also the lifetime of a shared memory variable is equal to the lifetime of a block. When a block finishes its execution, shared memory variables belong to that block are destroyed. In the third row of Table 3.5, the features of global memory can be observed. Global variables can be declared via “`__device__`” statement. Unlike previous variables, global variable is declared in the host code. Rather than using the “`__device__`” expression, more common declaration of the global memory is provided via “`cudaMalloc`” and “`cudaMemCpy`”. Although accesses to global variables are slow, they are visible to all threads in the kernel and their lifetime lasts through the execution. For this reason, global variables

can be used to cooperate across blocks. But, when a thread changes the value of particular global variable, other threads may not realize this modification immediately. Stopping the kernel execution is the only way to provide synchronization between threads from different blocks. This is why global variables are generally used to send information from one kernel invocation to another one.

Briefly we will mention some details about CUDA memory variables. Firstly we actually do not need to use identifier “__ device__” to declare shared memory and constant memory variables as constant and shared memory are already in the device. The second detail is that although all automatic variables are placed into registers, automatic variable arrays are stored in the global memory. Thus the access to a large automatic array is quite slow.

It is critical to decide where the variables should be declared. If we want host access to a variable, then the variables should be declared outside any function. The variables that are declared in a kernel or device function cannot be accessed from host. Constant and global memory variables are in this class. On the other hand, the variables that host don't need to access such as registers and shared memory variables, can be declared in kernel.

Lastly we should mention shared memory a bit more in detail as it is crucial in terms of the speed of an algorithm. Shared memory, whose contents are explicitly declared, is an exceptional memory type in CUDA. Explicit memories are the memories that can be intentionally and consciously declared. Each streaming multiprocessor (SM) has a shared memory. Shared memory is much faster accessible than global memory and also its performance is much better in both latency and throughput. Shared memory is generally used to store specific part of the data in the global memory which are frequently used during the execution of kernel (Kirk and Hwu, 2013). Although shared memory is quite fast, the memory of it is pretty small as it needs to fit into the processor.

In CUDA there is a common programming method for shared memory. The most important point is that the data should be divided into parts called tiles that fit into

shared memory. As mentioned before the shared memory is allocated to blocks. So these tiles are actually blocks. All the threads in a block cooperate and copy the tile from global memory to the shared memory. As there can be wide range of threads in a block, a good parallelism can be managed while transferring data to the shared memory. This kind of parallelism refers to as memory level parallelism. Once the data is moved to shared memory, the computations can be managed in a much faster way as shared memory gives the data to the processing units at quite high speed.

4 EXPERIMENTAL DESIGN

In this study, to solve symmetric TSP accelerated 2-opt and 3-opt algorithms were implemented utilizing SIMD structure of GPU. Fundamentally “Accelerating 2-opt and 3-opt Local Search Using GPU in the Travelling Salesman Problem (KamilRocki and Reiji Suda, 2012) ” and “High Performance GPU Accelerated Local Optimization in TSP (KamilRocki and Reiji Suda, 2013)” papers were considered. In addition to observing how much GPU accelerated sequential 2-opt and 3-opt CPU algorithms, some tests were performed on these algorithms to discover the best way of allocating GPU resources and also take the advantages of different parallelism strategies.

Repeating 2-opt exchanges on a travelling salesman tour, which considerably improves the solution, is an efficient local-search method for solving TSP.

As shown in Figure 4.1, in a 2-opt exchange step, two edges are removed from the current tour and after this removal process two sub-tours emerge. There is only one way to reconnect these sub-tours by protecting the validity of the travelling salesman tour and they are connected in this way. 2-opt exchange is performed in the event that the cost of the two edges that reconnects two new sub-tours created is lower than the cost of removed edges. As the other parts of the tour remain same, there is no need for further calculations.

2-opt algorithm calculates the effect of each possible edge exchange on the current tour cost. From among these possible exchanges, it performs the one with the best improvement, in other words the exchange that decreases current tour cost most. Algorithm repeats this step until there is no further improvement.

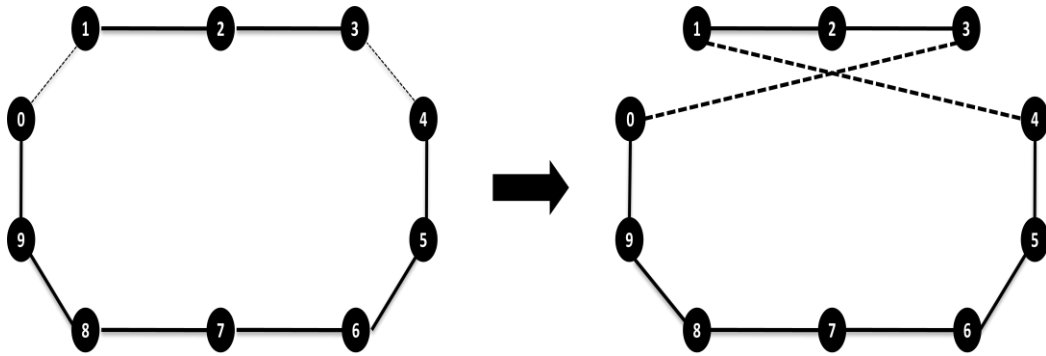


Figure 0.12 2-opt step on a travelling salesman tour

If the number of nodes in the tour is “n”, the number of possible edge exchanges/swaps in each iteration is $\frac{n \times (n-1)}{2}$.

Example 4.1: Assuming that there are 10 nodes/cities in a tour, all possible edge exchanges are presented in Table 4.2 in which “t” represents the array that keeps tour order (check Table 4.1 for the initial tour order).

Table 0.6 Initial tour order in Example 4.1

t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]	t[10]
0	1	2	3	4	5	6	7	8	9	0

As shown in Table 4.2, there are $\frac{10 \times 9}{2} = 45$ possible edge swaps in a tour consisting of 10 nodes. In this table, the entries (the nodes) in bold consecutively represent the row and the column indexes of a triangular matrix which will be important in the parallelization phase of 2-opt algorithm. Note that in the last column of each row deleted and added edges are same. We don’t save any time when we eliminate these exchanges as they will be performed in parallel. For this reason we won’t let our program to make an effort to control unnecessary exchanges.

If node “i” symbolizes the row indexes and node “j” symbolizes the column indexes, “i” and “j” variables will take the values in Table 4.3. In this table, in each cell the first number in the parenthesis will be assigned to “i” variable and the second one will be assigned to “j” variable.

Table 0.7 All possible edge exchanges for a TSP tour with 10 nodes

Remove (t{2},t{1}) (t{1},t{0}) Add (t{2},t{1}) (t{1},t{0})								
Remove (t{3},t{2}) (t{1},t{0}) Add (t{3},t{1}) (t{2},t{0})	Remove (t{3},t{2}) (t{2},t{1}) Add (t{3},t{2}) (t{2},t{1})							
Remove (t{4},t{3}) (t{1},t{0}) Add (t{4},t{1}) (t{3},t{0})	Remove (t{4},t{3}) (t{2},t{1}) Add (t{4},t{2}) (t{3},t{1})	Remove (t{4},t{3}) (t{3},t{2}) Add (t{4},t{3}) (t{3},t{2})						
Remove (t{5},t{4}) (t{1},t{0}) Add (t{5},t{1}) (t{4},t{0})	Remove (t{5},t{4}) (t{2},t{1}) Add (t{5},t{2}) (t{4},t{1})	Remove (t{5},t{4}) (t{3},t{2}) Add (t{5},t{3}) (t{4},t{2})	Remove (t{5},t{4}) (t{4},t{3}) Add (t{5},t{4}) (t{4},t{3})					
Remove (t{6},t{5}) (t{1},t{0}) Add (t{6},t{1}) (t{5},t{0})	Remove (t{6},t{5}) (t{2},t{1}) Add (t{6},t{2}) (t{5},t{1})	Remove (t{6},t{5}) (t{3},t{2}) Add (t{6},t{3}) (t{5},t{2})	Remove (t{6},t{5}) (t{4},t{3}) Add (t{6},t{4}) (t{5},t{3})	Remove (t{6},t{5}) (t{5},t{4}) Add (t{6},t{5}) (t{5},t{4})				
Remove (t{7},t{6}) (t{1},t{0}) Add (t{7},t{1}) (t{6},t{0})	Remove (t{7},t{6}) (t{2},t{1}) Add (t{7},t{2}) (t{6},t{1})	Remove (t{7},t{6}) (t{3},t{2}) Add (t{7},t{3}) (t{6},t{2})	Remove (t{7},t{6}) (t{4},t{3}) Add (t{7},t{4}) (t{6},t{3})	Remove (t{7},t{6}) (t{5},t{4}) Add (t{7},t{5}) (t{6},t{4})	Remove (t{7},t{6}) (t{6},t{5}) Add (t{7},t{6}) (t{6},t{5})			
Remove (t{8},t{7}) (t{1},t{0}) Add (t{8},t{1}) (t{7},t{0})	Remove (t{8},t{7}) (t{2},t{1}) Add (t{8},t{2}) (t{7},t{1})	Remove (t{8},t{7}) (t{3},t{2}) Add (t{8},t{3}) (t{7},t{2})	Remove (t{8},t{7}) (t{4},t{3}) Add (t{8},t{4}) (t{7},t{3})	Remove (t{8},t{7}) (t{5},t{4}) Add (t{8},t{5}) (t{7},t{4})	Remove (t{8},t{7}) (t{6},t{5}) Add (t{8},t{6}) (t{7},t{5})	Remove (t{8},t{7}) (t{7},t{6}) Add (t{8},t{7}) (t{7},t{6})		
Remove (t{9},t{8}) (t{1},t{0}) Add (t{9},t{1}) (t{8},t{0})	Remove (t{9},t{8}) (t{2},t{1}) Add (t{9},t{2}) (t{8},t{1})	Remove (t{9},t{8}) (t{3},t{2}) Add (t{9},t{3}) (t{8},t{2})	Remove (t{9},t{8}) (t{4},t{3}) Add (t{9},t{4}) (t{8},t{3})	Remove (t{9},t{8}) (t{5},t{4}) Add (t{9},t{5}) (t{8},t{4})	Remove (t{9},t{8}) (t{6},t{5}) Add (t{9},t{6}) (t{8},t{5})	Remove (t{9},t{8}) (t{7},t{6}) Add (t{9},t{7}) (t{8},t{6})	Remove (t{9},t{8}) (t{8},t{7}) Add (t{9},t{8}) (t{8},t{7})	
Remove (t{10},t{9}) (t{1},t{0}) Add (t{10},t{1}) (t{9},t{0})	Remove (t{10},t{9}) (t{2},t{1}) Add (t{10},t{2}) (t{9},t{1})	Remove (t{10},t{9}) (t{3},t{2}) Add (t{10},t{3}) (t{9},t{2})	Remove (t{10},t{9}) (t{4},t{3}) Add (t{10},t{4}) (t{9},t{3})	Remove (t{10},t{9}) (t{5},t{4}) Add (t{10},t{5}) (t{9},t{4})	Remove (t{10},t{9}) (t{6},t{5}) Add (t{10},t{6}) (t{9},t{5})	Remove (t{10},t{9}) (t{7},t{6}) Add (t{10},t{7}) (t{9},t{6})	Remove (t{10},t{9}) (t{8},t{7}) Add (t{10},t{8}) (t{9},t{7})	Remove (t{10},t{9}) (t{9},t{8}) Add (t{10},t{9}) (t{9},t{8})

Table 0.8 Required indexes that will be produced by built-in variables in kernel
(all different city combinations)

j i	1	2	3	4	5	6	7	8	9
1									
2	(2,1)								
3	(3,1)	(3,2)							
4	(4,1)	(4,2)	(4,3)						
5	(5,1)	(5,2)	(5,3)	(5,4)					
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)				
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)			
8	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)		
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	
10	(10,1)	(10,2)	(10,3)	(10,4)	(10,5)	(10,6)	(10,7)	(10,8)	(10,9)

Based on the edge exchange operations in Table 4.2 and specified “i” and “j” values in Table 4.3, the sequential 2-opt algorithm is demonstrated in Figure 4.2 where “n” is the number of nodes, “change” is the decrease in tour cost and “global_min” is the minimum of the “change” values. Searching for minimum of the “change” values gives the maximum decrease in the tour cost. Algorithm sequentially calculates the decrease in the tour cost for all possible edge exchanges. Meanwhile, it compares the effect of current edge exchange with the previous ones. If the current edge exchange improves the tour cost more than previous best exchange, it stores the related “i” and “j” values. At the end it reaches the best improvement for the current solution.

```

global_min = 0;
for (int j=1; j<= (n-1); j++)
    for (int i=j+1; i < n; i++)
    {
        change = distance(t[i],t[j]) + distance(t[i-1],t[j-1]) -
                 distance(t[i-1],t[i]) - distance(t[j-1],t[j]);
        if (change < global_min)
        {
            remember i and j;
            global_min = change;
        }
    }

```

Figure 0.13 Sequential 2-opt algorithm

4.1 Parallelization Strategy for 2-Opt Algorithm on TSP

In 2-opt method, the same task is performed on different parts of the travelling salesman tour. As can be seen in the sequential algorithm (see Figure 4.2), the calculation of “distance(t[i],t[j]) + distance(t[i-1],t[j-1]) - distance(t[i-1],t[i]) -distance(t[j-1]+t[j])” is applied to all city combinations in the current tour as the best improvement strategy is implemented. Thus, it can be said that 2-opt algorithm is quite suitable for “single instruction multiple data (SIMD)” structure of GPU.

Parallelism can be managed by distributing $\frac{n*(n-1)}{2}$ possible edge exchanges (see Table 4.2) among different threads equally so that each thread can calculate the effects of relevant exchanges on the tour cost. In order to achieve distributing process one or more city combinations (Table 4.3) should be assigned to different threads. In this way each thread can perform exchange effect calculations on different parts of the tour using city pairs assigned to it.

In this study “Calculating the effect of single edge exchange” will be called as a “job” and all jobs will be made parallel to each other to discover the effect of all possible edge exchanges. As discussed earlier, it is important to combine thread level parallelism and iteration level parallelism in some situations. In our problem, one thread can perform several jobs in a parallel way which exploits iteration level parallelism and also different threads can perform different jobs in parallel to each other which utilizes thread level parallelism.

To decide how many jobs a thread will perform, number of possible edge exchanges should be divided by the number of threads.

$$\text{Number of jobs for each thread} = \frac{n*(n-1)}{2*\text{totalNumberOfThreadsInGPU}} \quad (4.1)$$

If 5 threads are used to solve TSP in Example 4.1, each thread will perform $\frac{10*(10-1)}{2*5} = 9$ jobs, which will be called “number of iterations” or “iterations” from now on. It means that each thread will iterate 9 times to calculate the effect of 9 possible swaps.

The most critical part in the algorithm is generating common formulas for all threads so that they can produce “i” and “j” values in Table 4.3. Table 4.4, in which id represents index of a job, shows the division of jobs between threads and also (i,j) values associated with these jobs . In order to obtain “i” values which represent rows, we should relate i values to the job ids that derive from predefined variables in CUDA.

Table 0.9 Assigning jobs to threads
(ids represent jobs that will be performed in parallel)

i \ j	1	2	3	4	5	6	7	8	9
1									
2	thread 0 id = 0 (2,1)								
3	thread 1 id=1 (3,1)	thread 2 id=2 (3,2)							
4	thread 3 id=3 (4, 1)	thread 4 id=4 (4,2)	thread 0 id=5 (4,3)						
5	thread 1 id=6 (5, 1)	thread 2 id=7 (5,2)	thread 3 id=8 (5,3)	thread4 id=9 (5,4)					
6	thread 0 id=10 (6, 1)	thread 1 id=11 (6,2)	thread 2 id=12 (6,3)	thread 3 id=13 (6,4)	thread 4 id=14 (6,5)				
7	thread 0 id=15 (7, 1)	thread 1 id=16 (7,2)	thread 2 id=17 (7,3)	thread 3 id=18 (7,4)	thread 4 id=19 (7,5)	thread 0 id=20 (7,6)			
8	thread 1 id=21 (8, 1)	thread 2 id=22 (8,2)	thread 3 id=23 (8,3)	thread 4 id=24 (8,4)	thread 0 id=25 (8,5)	thread 1 id=26 (8,6)	thread 2 id=27 (8,7)		
9	thread 3 id=28 (9, 1)	thread 4 id=29 (9,2)	thread 0 id=30 (9,3)	thread 1 id=31 (9,4)	thread 2 id=32 (9,5)	thread 3 id=33 (9,6)	thread 4 id=34 (9,7)	thread 0 id=35 (9,8)	
10	thread 1 id=36 (10, 1)	thread 2 id=37 (10,2)	thread 3 id=38 (10,3)	thread 4 id=39 (10,4)	thread 0 id=40 (10,5)	thread 1 id=41 (10,6)	thread 2 id=42 (10,7)	thread 3 id=43 (10,8)	thread 4 id=44 (10,9)

In example 4.1, there are 45 jobs totally. To complete these jobs 5 threads are allocated for thread level parallelism and 9 iterations will be performed for iteration level parallelism. As each of 5 threads iterates through 9 different jobs, at the end all of the jobs will be performed.

In order to allocate 5 threads, block dimension should be initialized as “5” and grid dimension is 1 in the host code. If “idx” represents thread ids, it should be defined as in Figure 4.3.

```
idx = blockDim.x*blockIdx.x+threadIdx.x
where
blockDim.x = 5
blockIdx.x = 0
threadIdx.x = 0, 1, 2, 3, 4
```

Figure 0.14 Assigning thread indices via built-in variables

Thus, the variable of “idx” will get the values of “0,1,2,3,4” automatically.

Iteration level parallelism will be managed in the kernel function. If “no” represents the number of iterations and “packSize” represents all the threads in the grid, job ids will be obtained as in Figure 4.4.

```
id = idx + no*packSize
where
idx = 0, 1, 2, 3, 4
packSize = blockDim.x*gridDim.x = 5*1=5
no = 0, 1, 2, 3, 4, 5, 6, 7, 8
```

Figure 0.15 Assigning jobs to specified threads

So the ids will be as follows:

id = 0,1,2,3,4,5,6,7,8,9,10.....43,44

Table 4.5 summarizes calculation of first fourteen job ids depending on built-in variables and iterations.

Table 0.10 Calculating the job ids using built-in variables and iterations

threadIdx.x	blockIdx.x	blockDim.x	idx	iteration(no)	packSize	id
0	0	5	0	0	5	0
1	0	5	1	0	5	1
2	0	5	2	0	5	2
3	0	5	3	0	5	3
4	0	5	4	0	5	4
0	0	5	0	1	5	5
1	0	5	1	1	5	6
2	0	5	2	1	5	7
3	0	5	3	1	5	8
4	0	5	4	1	5	9
0	0	5	0	2	5	10
1	0	5	1	2	5	11
2	0	5	2	2	5	12
3	0	5	3	2	5	13
4	0	5	4	2	5	14

Because up to and including the k^{th} row there are $\frac{k*(k+1)}{2}$ jobs in Table 4.4, it can be said that $\frac{k*(k+1)}{2}$ counts the number of jobs. Through this approach Rocki and Suda achieved the formula 4.2.

$$\frac{k * (k + 1)}{2} = id \tag{4.2}$$

By finding the roots of that quadratic equation and modifying it, the formula of “i” indices were obtained as in 4.3 where “i” values are rounded down in order to generate integer numbers.

$$i = \frac{3 + \sqrt{8 * id + 1}}{2} \tag{4.3}$$

Utilizing “i” and “id” values, the formula of “j” was obtained like in 4.4 where “j” values are rounded down in order to generate integer numbers.

$$j = id - \frac{(i - 2) * (i - 1)}{2} + 1 \tag{4.4}$$

For example, when id is equal to “25”,

$$i = \frac{3 + \sqrt{8 * 25 + 1}}{2} = 8,5 \cong 8 \text{ and } j = 25 - \frac{6 * 7}{2} + 1 = 5.$$

When i is equal to “11”,

$$i = \frac{3 + \sqrt{8 \cdot 11 + 1}}{2} = 6,2 \cong 6 \text{ and } j = 11 - \frac{4 \cdot 5}{2} + 1 = 2. \text{ (Check Table 4.4)}$$

After calculating all the values of (i, j) pairs like in Table 4.4, the 2-opt exchange effects of related edges will be calculated in parallel. In the algorithm, for each (i, j) pair the cost of the deleted edges will be subtracted from cost of the added edges in order to discover how much 2-opt exchange decreases the current tour cost. The result will be assigned to a variable called as “change”. Each thread will store different “change” values in its own registers. From among these values, after the positive values are eliminated as they signify increase in the tour cost, the minimum value will be chosen as it gives the maximum decrease in the tour cost. Then 2-opt exchange will be applied to relevant edges and the current tour will be updated. The 2-opt exchange operations will continue until there is no further improvement in the tour cost.

While solving a large scale TSP using GPU, it is not logical to store the calculated distances between cities in the off-chip global memory since the access of threads in kernel to the global memory is very slow. As outlined in the “Architecture” on-chip shared memory is very fast but also very limited which is 48KB in our device, therefore it cannot be used too. The best way is to store only city coordinates and tour order in fast shared memory and calculating the necessary distances each time utilizing the high-power computational power of GPU. The city coordinates will be defined as “structure” which includes integer x and y variables representing x and y coordinates of the cities.

As the city coordinates and tour order will be stored in the shared memory, it should be calculated that until which problem size our algorithm will be feasible. The size of a city coordinate defined as integer is 4 bytes. Moreover tour order will be defined as “unsigned short” with the size of 2 bytes. The memory needed for each city will be 10 bytes because a city has two coordinates and one tour order.

Shared memory is 48 KB which equals to $48 \times 1024 = 49152$ bytes. Consequently, shared memory can store the coordinates and tour order of $\frac{49152 \text{ bytes}}{10 \text{ bytes}} = 4915$ cities.

Under these circumstances solvable maximum problem size is 4915.

Formula 4.5 represents the “distance” function which will calculate the Euclidean distances between cities where “coords[i].x” represents the x coordinate of the city “i” and “coords[i].y” represents the y coordinate of the city i.

$$\sqrt{(\text{coords}[i].x - \text{coords}[j].x)^2 + (\text{coords}[i].y - \text{coords}[j].y)^2} \quad (4.5)$$

Figure 4.5 presents the “distance” function on device. As this function will be called from kernel function, which executes on the device, and only a device function can be called from a device function (Table 3.2), the distance function should be preceded by “__device__” declaration. In order to call “distance” function from host, it should be preceded by “__host__”.

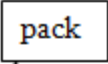
```
__device__ int distance (int i, int j, city_coords *coords)
{
    register float dx, dy;
    dx = coords[i].x - coords[j].x;
    dy = coords[i].y - coords[j].y;
    return(int) (sqrtf(dx*dx+dy*dy));
}
```

Figure 0.16 Device function to calculate the distances between cities

In Appendix C the draft view of the algorithm can be examined. Code starts at host defining necessary variables such as for number of nodes/cities, number of possible exchanges/swaps in the tour. Variable “iter” (number of iterations) is defined to assign more than one job to a thread, in other words to decide the amount of iteration level parallelism. The way of distributing jobs among threads and iterations can be observed in Table 4.6. In this table, the numbers in the cells show the ids of possible edge exchange effects (job ids).

Table 0.11 Thread-level and iteration-level parallelism

iterations (no) \ threads (idx)	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24
5	25	26	27	28	29
6	30	31	32	33	34
7	35	36	37	38	39
8	40	41	42	43	44



After defining host variables, the device variables are defined for transferring to kernel and device memory is allocated for them. Firstly city coordinates are transferred from host to device. Because city coordinates will be fixed throughout the algorithm, it is enough to do this transfer operation just once. The initial tour order is specified in ascending order (Table 4.1). Thereby an initial solution is determined. The configuration parameters (block and grid dimension) and number of iterations are set. It is important to reiterate that multiplication of iterations, block and grid dimension should be arranged as equal to the number of possible edge exchanges. As mentioned before, 2-opt exchange is applied until there is no further decrease in the tour cost. Thus the kernel function which searches for the best improvement via 2-opt method will be invoked as long as the best improvement in the tour cost is less than zero. Each time the updated tour order should be transferred from host to device and then the kernel should be invoked to begin a new 2-opt search on the new tour.

In kernel function, through built-in CUDA variables the indexes of threads in SM are assigned to a register variable called “idx”. “packSize” calculates the number of threads in SM. Via for loop jumping as “packSize” distance iteration times, new jobs are assigned to threads until all jobs are completed (check Table 4.6). This process provides iteration level parallelism. Ids of all jobs, in other words all possible exchanges, are assigned to threads in this way. According to these ids, the indexes of each possible city

pairs are calculated (Formula 4.3 and 4.4). Utilizing the indexes of city pairs the relevant edge exchange effects are calculated in parallel. The exchange which decreases the tour cost most is detected via CUDA-specific “atomicMin” function and the id of it is stored. Then this id is copied to host where the necessary edge exchange is performed in the current tour. Acquired new tour is copied to kernel and next 2-opt search is performed on this new tour.

General draft of CUDA Code is given in Appendix C.

4.2 Experimental Results

In order to utilize the GPU optimally, different kinds of kernel configuration parameters will be tested. Considering resource restrictions of our GPU device and occupancy of streaming multiprocessor in the device, the results will be commented. As discussed previously, occupancy is calculated by dividing the number of active warps in an SM by the number of warps supported on an SM of the GPU. Busy warps in SM are called as active. Occupancy helps to exploit GPU memory efficiently by increasing thread level parallelism. Through a high occupancy many resources of GPU device can be kept busy and more jobs can be done in parallel.

Our device, Quadro K600, has 1 streaming multiprocessor which is called SMX and the compute capability of Quadro K600 is 3.0 (check Table 3.4). Therefore SMX have maximum 2048 resident threads and 16 resident blocks. As each warp consists of 32 threads, an SMX have maximum $\frac{2048}{32} = 64$ resident warps. In order to utilize all the resident warps in an SMX, blocks and grid should be arranged like in Table 3.5. However other resource restrictions such as maximum number of registers and shared memory limit per SMX may prevent to utilize all the warps. These factors should be checked in order to understand whether they decreased the number of active warps or not. Total number of registers per SMX is 65536 and shared memory per SMX is $48KB * 1024 = 49152$ bytes in our device. After discovering the number of registers and amount of shared memory that each block uses, the number of active blocks and active warps can be calculated. These factors do not decrease the number of active

blocks or warps if there are enough registers and shared memory for each block in the optimal warp system, in which all the warps are used (check Table 3.5). Otherwise, occupancy may decrease. Let's assume that there are 512 threads in a block and 4 blocks in the grid which means 64 possible active warps are available. But, if the number of registers or amount of shared memory in the SMX is enough for only 3 blocks, it means that there are only 3 active blocks and $\frac{512\text{threads in a block}}{32\text{threads in a warp}} \times 3\text{blocks} = 48$ active warps. A programmer tool called CUDA Occupancy Calculator automatically performs these calculations when user enters the necessary information about GPU device and some resource usage. Considering additional factors such as “register allocation unit size” or “warp allocation granularity”, its calculations are more precise.

For different sized TSPs, the performance changes with different resource allocations can be examined in the following experiments. As reported in Table 4.6, in our algorithm some part of the possible swaps should be distributed among launched threads via block dimension and grid dimension. Then remaining possible swaps should be assigned to the same threads again via “iterations (no)”. In other words, “number of iterations” will help to assign more than one job to a thread.

In these experiments the block dimensions will be arranged as multiples of warp size which is 32. The configurations with the same block dimensions will be grouped and there will be 4 groups in our experiments which are 1024, 512, 256 and 128. For each group different [grid dimension, iterations] combinations will be executed. Among each group a launch with the best performance will be selected. In the table italic and bold entries represent them. Then, among all launches the best one or ones will be selected and the entries in bold indicate them.

In order to select best performance, the kernel performances will be compared because number of 2-opt iterations will change each time. This change arises from the behavior of CUDA-specific “atomicMin” function. When different edge exchanges produce the same amount of decrease in the tour cost, this function may choose any of them at each program run. For this reason minimized tour cost and 2-opt iteration results are not identical, but quite close to each other. Nevertheless, in some situations comparing “cpu+gpu” times will not be fair enough.

It is important to reiterate that “grid dimension” and “block dimension” together will help us to observe the effect of thread level parallelism while “number of iterations” will help to observe the effect of iteration level parallelism.

4.2.1 Experiment 1-500 cities

We assume that the best kernel launches will be achieved when all warps in SMX are utilized (i.e when occupancy is 100%). Predicted best kernel launches and observed best kernel launches are compared in Table 4.7. More detailed performance results for different sized TSP problems with various configuration parameter settings can be found in Appendix A.

Table 0.12 Predicted best kernel launches vs. observed best kernel launches for TSP with 500 cities

Predicted best kernel launches			Observed best launches						
Block Dim.	Grid Dim.	# of iterations	Block Dim.	Grid Dim.	# of iterations	kernel time (ms)	Minimized cost	# of 2-opt iterations	CPU+GPU time (ms)
1024	2	61	1024	2	61	0.203	16292	529	325
512	4	61	512	4	61	0.203	16292	529	326
256	8	61	256	8	61	0.209	16196	538	340
128	16	61	128	9	109	0.307	16304	529	346

In Table 4.7 it can be seen that in the first three groups best performances are obtained when all warps are active. It means that shared memory and registers are enough for all launched blocks. However, in the last group out of 16 blocks, only 9 blocks are used when the best performance is obtained.

Restriction of Resources

In our algorithm, each thread uses 20 registers and each block occupies 5012 bytes shared memory for 500 nodes/cities. For each group calculation of the resource usage in the SMX is given in Table 4.8. In the table the number of active blocks that registers

and shared memory allows is determined. After analyzing restrictions, occupancy is calculated.

Table 0.13 Restrictions of shared memory and registers

When All Warps Are Used in SM		Restriction of Registers		Restriction of Shared Memory		After Restrictions	
Block Dim	Grid Dim	# of Registers Used in SM	max. # of active blocks registers allow	Shared memory used in SM	max. # of active blocks shared memory allows	Grid Dim	Occupancy
1024	2	$1024 * 20 = 20480$	$\frac{65536}{20480} = 3 > 2$	5012 bytes	$\frac{49152}{5012} = 9 > 2$	2	100%
512	4	$512 * 20 = 10240$	$\frac{65536}{10240} = 6 > 4$		$\frac{49152}{5012} = 9 > 4$	4	100%
256	8	$256 * 20 = 5120$	$\frac{65536}{5120} = 12 > 8$		$\frac{49152}{5012} = 9 > 8$	8	100%
128	16	$128 * 20 = 2560$	$\frac{65536}{2560} = 25 > 16$		$\frac{49152}{5012} = 9 < 16$	9	56%

1st Group-Block Dimension: 1024

To utilize all the warps in the multiprocessor 2 blocks should be launched.

1st check: Restriction of Registers

In the first group, because each block has 1024 threads, a block exploits $1024 * 20 = 20480$ registers. We know that maximum number of registers per SMX is 65536. Thus each multiprocessor can have maximum $\frac{65536}{20480} = 3$ blocks. Since there are enough registers for 3 blocks, which is greater than 2, we conclude that register capacity of the SMX does not restrict the usage of all active warps.

2nd check: Restriction of Shared Memory

For 500 nodes, shared memory usage per block is 5012 bytes in our program. We know that maximum shared memory size per streaming multiprocessor is 49152 bytes. So, shared memory allows $\frac{49152}{5012} = 9$ active blocks which is greater than 2. Shared memory capacity of the SMX does not restrict the usage of all active warps.

To sum up, in the first group all 64 warps can be utilized by launching 2 blocks and 1024 threads in a block. As anticipated the best performance is obtained when all warps are active, in other words when occupancy is %100.

In the second and third group consecutively 4 and 8 blocks are required in order to utilize all the active warps. Their behaviors are similar to first one. Shared memory and registers do not give rise to any restrictions and all warps can be utilized which provides 100% occupancy and best performances.

Rather than the first three groups, in the last group shared memory capacity prevents the usage of all warps.

4th Group-Block Dimension: 128

To utilize all the warps in the multiprocessor 16 blocks should be launched.

Restriction of Shared Memory

Shared memory usage varies with the change of the problem size. Thus occupied shared memory doesn't change for this configuration. Although shared memory still allows $\frac{49152}{5012} = 9$ active blocks, to exploit all warps 16 active blocks are required this time. It means that in the SMX there is enough memory for only 9 blocks which is lower than 16. In this group $\frac{128}{32} * 9 = 36$ warps out of 64 can be utilized by launching 9 blocks and 128 threads in each block. Unlike %100 occupancy in other groups, in this group occupancy is $\frac{\text{number of active warps in SMX}}{\text{maximum number of warps in SMX}} = \frac{36}{64} = 56\%$ which creates worse performances than other groups. Nevertheless, the best performance within fourth group is achieved when all warps that shared memory and register allows are used.

Consequently, for the problem with 500 cities best performances are obtained when occupancy is 100%. It seems that higher occupancy gives better results in our algorithm.

As discussed previously, CUDA Occupancy Calculator performs above calculations when necessary information about GPU device and resource usage of algorithm is provided. It asks for the compute capability and shared memory size of the device.

Moreover, number of threads in a block of the launched kernel, number of the registers used per thread and the shared memory occupied by per block should be entered as inputs. Figure 4.6 displays the Occupancy Calculator inputs that should be entered by user for the first group.

1.) Select Compute Capability (click):	3,0
1.b) Select Shared Memory Size Config (bytes)	49152
2.) Enter your resource usage:	
Threads Per Block	1024
Registers Per Thread	20
Shared Memory Per Block (bytes)	5012

Figure 0.17 Inputs of occupancy calculator for problem with 500 nodes

After the inputs in Figure 4.6 are entered, the Occupancy Calculator shows the results consisting of the number of active threads, warps and blocks per multiprocessor and also occupancy of the multiprocessor as in Figure 4.7.

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	100%

Figure 0.18 Output of occupancy calculator for problem with 500 nodes

From Figure 4.7, it can be seen that our calculations are in line with the results of Occupancy Calculator. Occupancy Calculator also presents certain plots in which one can identify impact of varying block size (Figure 4.8), varying register count per thread and varying shared memory usage per block (Figure 4.9).

The best block size options can be identified from the Figure 4.8. Clearly block dimensions 256, 512 and 1024 produce best performances which supports our results in Table 4.7

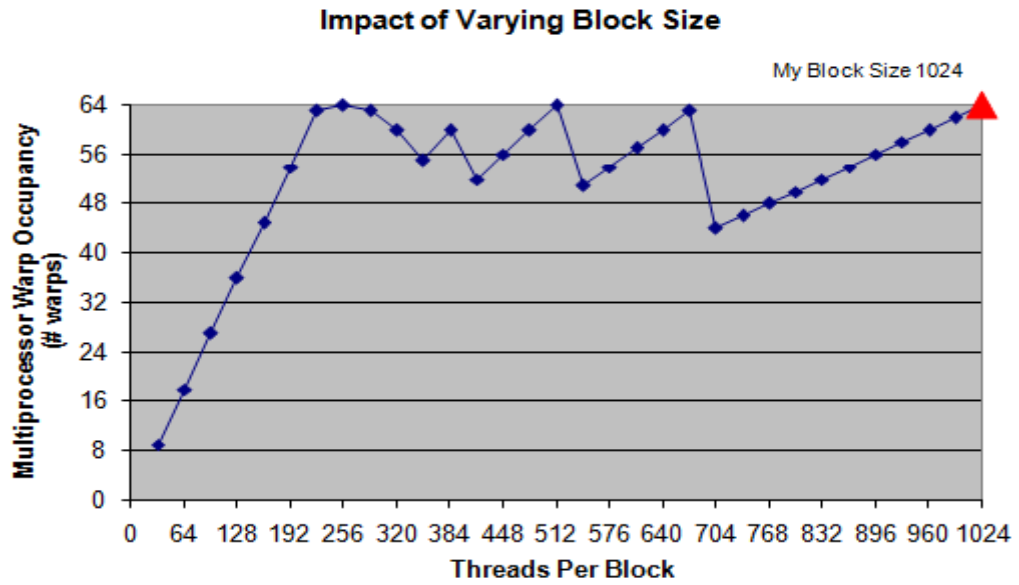


Figure 0.19 The effect of block size on occupancy

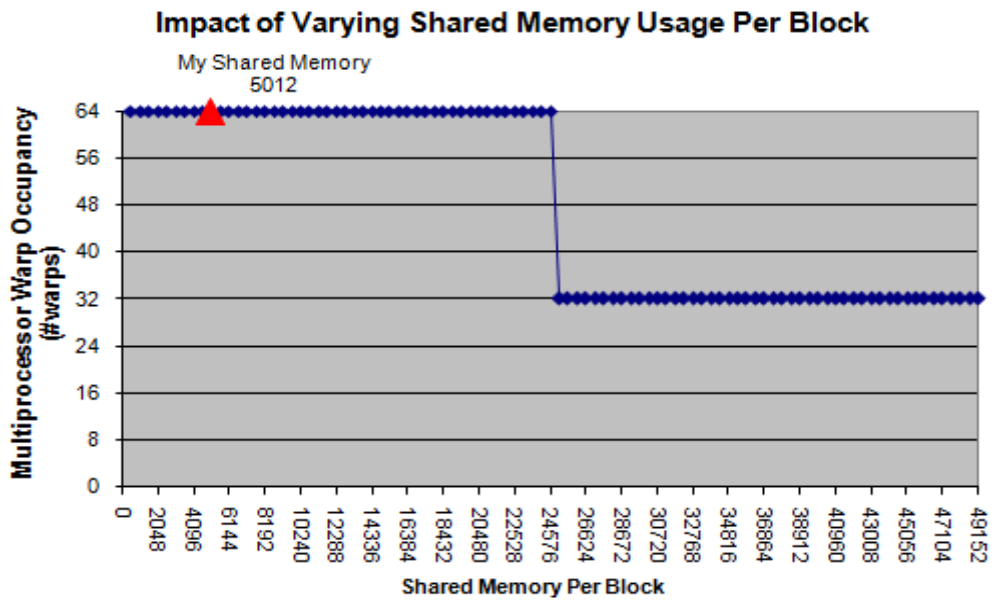


Figure 0.20 The effect of shared memory usage on occupancy

4.2.2 Experiment 2-1000 Cities

Table 0.14 Predicted best kernel launches vs. observed best kernel launches for TSP with 1000 cities

Predicted best kernel launches			Observed best launches						
Block Dim.	Grid Dim.	# of iterations	Block Dim.	Grid Dim.	# of iterations	Kernel time (ms)	Minimized cost	# of 2-opt iterations	CPU+GPU time (ms)
1024	2	242	1024	2	242	0.7	35985	1094	1096
512	4	242	512	4	242	0.7	36197	1090	1067
256	8	242	256	4	484	1	36193	1091	1742
128	16	242	128	4	967	2	36544	1089	3036

As illustrated by Table 4.9, from among all groups algorithm performs best when

- Block dimension is 1024 and grid dimension is 2.
- Block dimension is 512 and grid dimension is 4.

Apparently the best performance in third group is worse than the best performances in the first two groups. In the last group, the best performance is even worse than the best performance in the third group.

Let's discover the occupancies for the four groups and decide the best block size for this problem via Occupancy Calculator. It is important to reiterate that occupied shared memory changes with the increasing size of the problem. For 1000 nodes necessary shared memory size is 10012 bytes. Remaining settings will stay as before.

Figure 4.10 shows the best block sizes for this problem as 1024 and 512 which is consistent with our results obtained in Table 4.9. It comes from activating all the warps in the SMX. As all warps can be active when block size is 1024 or 512, the best performances are obtained from among these groups. It appears that occupancy calculator verifies our results.

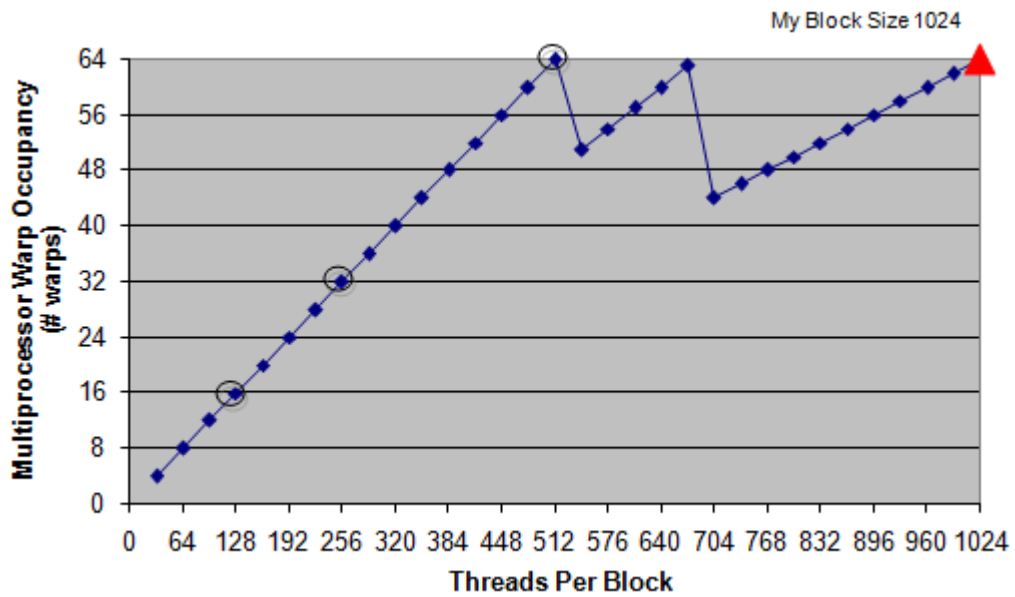


Figure 0.21 The impact of block size on occupancy for the problem with 1000 nodes

Table 4.10 summarizes the resource usage and GPU occupancy data for all groups. As listed in the table 32 warps out of 64 are active when block dimension is 256 and only 16 warps can be active when block dimension is set to 128 which gives rise to the decrease in the occupancy. Worse performances in last two groups (see Table 4.9) are the consequence of this lower occupancy.

Table 0.15 Occupancy information for the problem with 1000 nodes

RESOURCE USAGE:				
Threads Per Block	1024	512	256	128
Registers Per Thread	20	20	20	20
Shared Memory Per Block (bytes)	10012	10012	10012	10012
GPU OCCUPANCY DATA:				
Active Threads per Multiprocessor	2048	2048	1024	512
Active Warps per Multiprocessor	64	64	32	16
Active Thread Blocks per Multiprocessor	2	4	4	4
Occupancy of each Multiprocessor	100%	100%	50%	25%

Inspection of Table 4.10 indicates that in the third group only 4 blocks out of 8 are active and in the fourth group only 4 blocks out of 16 are active because shared memory restricts the block size that can be used per multiprocessor with $\frac{49152}{10012} = 4$. Thus, the

algorithm displays best performances within these two groups when kernel is launched with 4 blocks (check Table 4.9).

4.2.3 Experiment 3-1500 Cities

As seen in Figure 4.11 and Table 4.11, occupancy calculator shows that the best block size is 1024 for the problem with 1500 nodes as 100% occupancy can be managed. According to the results of occupancy calculator, the performances are expected to be as follows:

$$\text{BKT of (Group 1)} < \text{BKT of (Group 2)} < \text{BKT of (Group 3)} < \text{BKT of (Group 4)}$$

(BKT: Best kernel time)

which means

$$\text{P of (Group 1)} > \text{P of (Group 2)} > \text{P of (Group 3)} > \text{P of (Group 4)}$$

(P: performance, >: better)

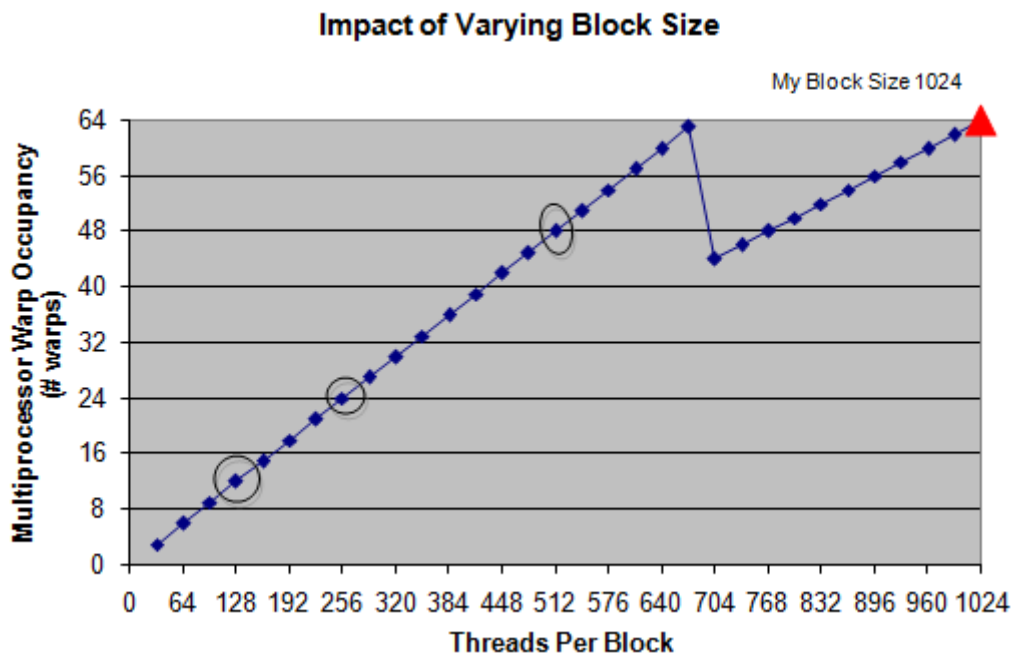


Figure 0.22 The effect of block size on occupancy in the problem with 1500 cities

Table 0.16 Occupancy information of the problem with 1500 cities

RESOURCE USAGE:				
Threads Per Block	1024	512	256	128
Registers Per Thread	20	20	20	20
Shared Memory Per Block (bytes)	15012	15012	15012	15012
GPU OCCUPANCY DATA:				
Active Threads per Multiprocessor	2048	1536	768	384
Active Warps per Multiprocessor	64	48	24	12
Active Thread Blocks per Multiprocessor	2	3	3	3
Occupancy of each Multiprocessor	100%	75%	38%	19%

Thus, the best performance should be observed in Group 1. In Group 1 two blocks and in other groups three blocks can be active. So, the best performances within each group will be obtained when the launch is arranged considering active thread blocks per multiprocessor found by occupancy calculator (Table 4.11). Similar to Experiment 2, in this experiment shared memory prevents utilizing all warps in the multiprocessor. Shared memory is enough only for 3 blocks.

Table 4.12 displays observed performances of different kernel launches for the problem size 1500. As anticipated the performance declines with the decrease in the occupancy as we can see from the table.

Occupancy (>:greater)

Group 1 (100%) > Group 2 (75%) > Group 3 (38%) > Group 4 (19%)

Performance (in terms of kernel time for 1 step 2-opt search) (>:better)

Group 1 (1.739 ms) > Group 2 (2.125 ms) > Group 3 (3.851 ms) > Group 4 (7.515 ms)

First 3 experiments indicates that the best way for the peak performances is after utilizing all resident warps as shared memory and register limits of the device allow, to perform remaining jobs through iteration level parallelism. Further experiments will be demonstrated in order to show the robustness of this approach.

Table 0.17 Predicted best kernel launches vs. observed best kernel launches for TSP with 1500 cities

Predicted best kernel launches			Observed best launches						
Block Dim.	Grid Dim.	# of iterations	Block Dim.	Grid Dim.	# of iterations	Kernel time (ms)	Minimized cost	# of 2-opt iterations	CPU+GPU time (ms)
1024	2	549	1024	2	549	1	47585	1645	3262
512	4	549	512	3	732	2	47293	1657	3935
256	8	549	256	3	1464	3	47297	1657	6693
128	16	549	128	3	2928	7	47604	1643	12655

4.2.4 Experiment 4-2000 Cities

As illustrated by Table 4.13, the occupancy declines with the decrease in the block dimension. For each group 2 blocks can be active which causes the reduction in the number of active warps of the groups excluding first group. As before the restriction to active warps stems from the shared memory limit. Since increasing the problem size accompanies more shared memory usage per block, number of active blocks in the multiprocessor decreases.

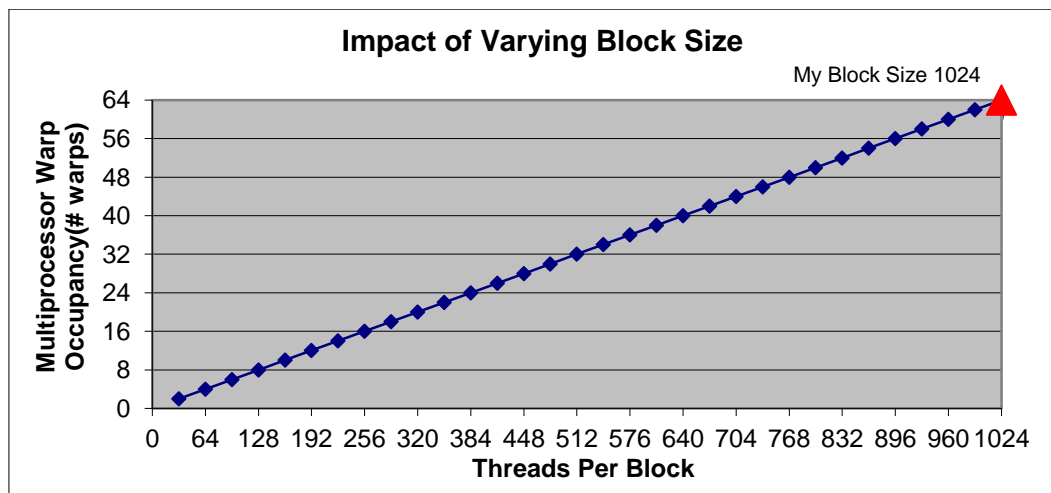


Figure 0.23 The effect of block size on occupancy in the problem with 2000 cities

Table 0.18 Occupancy information of the problem with 2000 cities

RESOURCE USAGE:				
Threads Per Block	1024	512	256	128
Registers Per Thread	20	20	20	20
Shared Memory Per Block (bytes)	20012	20012	20012	20012
GPU OCCUPANCY DATA:				
Active Threads per Multiprocessor	2048	1024	512	256
Active Warps per Multiprocessor	64	32	16	8
Active Thread Blocks per Multiprocessor	2	2	2	2
Occupancy of each Multiprocessor	100%	50%	25%	13%

Table 0.19 Predicted best kernel launches vs. observed best kernel launches for TSP with 2000 cities

Predicted best kernel launches			Observed best launches						
Block Dim.	Grid Dim.	# of iterations	Block Dim.	Grid Dim.	# of iterations	kernel time (ms)	Minimized cost	# of 2-opt iterations	CPU+GPU time (ms)
1024	2	977	1024	2	977	3	60926	2166	7099
512	4	977	512	2	1953	5	61120	2164	11654
256	8	977	256	2	3905	10	61336	2128	21832
128	16	977	128	2	7809	19	61359	2129	42611

4.3 Sequential vs. Parallel 2-opt Performance

Table 4.15 compares the performances of sequential and parallel 2-opt algorithms for different sized TSP problems. As can be observed that parallel 2-opt algorithm is much faster than sequential one.

Table 0.20 Comparing sequential and parallel 2-Opt algorithm performances

No of Nodes	Initial tour cost	Sequential 2-opt algorithm			Parallel 2-opt algorithm		
		Total time (ms)	No of 2-opt iterations	Minimized cost	Total time (CPU+GPU) (ms)	No of 2-opt iterations	Minimized cost
500	380825	253	541	16117	181	533	16278
1000	807793	5512	1081	34448	1096	1094	35985
2000	1705333	79233	2219	61279	7099	2166	60926
3000	3020898	315715	3292	91778	50754	3268	91132
4000	4535148	1287758	4470	121050	93671	4450	121396

4.4 Algorithm Modification to Solve Large Sized Travelling Salesman Problems

Maximum number of cities that can be stored in the shared memory was 4915 in the previous algorithm and the algorithm was not performing accurately when the problem size exceeded 4915. In order to solve bigger problems, previous algorithm will be modified. Looking through earliest formed CUDA code in Appendix C can provide better understanding about modifications.

4.4.1 First Step: Decreasing Shared Memory Usage for Each City

To save more space in the shared memory, the city coordinates will be sent to kernel in the tour's order instead of sending the tour order and city coordinates separately. As the tour order will not be stored in the shared memory anymore, solvable problem size will

$$\text{increase to } \frac{\text{shared memory}}{\text{memory used to store _ordinates of a city}} = \frac{49152 \text{ bytes}}{8 \text{ bytes}} = 6144.$$

Modifications in the Host Code

A new array called "orderedCoords" which will store the city coordinates in the tour order will be created and before each kernel launch it will be updated with the new order of city coordinates as in the Figure 4.12.

```

for(int n=0; n < (noOfNodes+1); n++)
{
    // stores x and y coordinates of the nth of the current tour in the
    new array
    orderedCoords[n].x=coords[tour[n]].x;
    orderedCoords[n].y=coords[tour[n]].y;
}

```

Figure 0.24 Storing the coordinates in the tour order

Modifications in the Kernel Code

As indicated in Figure 4.13, only the array of “orderedCoords” will be stored in the shared memory, instead of “Coords” and “Tour”. Thus, the “change” formula will also be modified.

Note: Commented rows in the Figure show the code before modification.

```

/**__shared__ city_ coords c[ ];
//__shared__ unsigned short t[ ];
__shared__ city_ coords c[ ];

**      For(int i = threadIdx.x; i<noOfNodes; i+=blockDim.x)
        {
            // c[i] = coords[i];
            // t[i] = tour[i];
            c[i] = orderedCoords[i];
        }

*** // change = distance(t[i],t[j],c)+distance(t[i-1],t[j-1],c)-
distance(t[i-1],t[i],c)-distance(t[j-1],t[j],c)
        change = distance(i,j,c)+ distance(i-1,j-1,c)-distance(i-1,i,c) -
distance(j-1,j,c)

```

Figure 0.25 The modifications in the kernel code for big sized problems

4.4.2 Second Step: Dividing Problem into Sub Problems

In order to increase solvable problem size further, city coordinates will be divided into partitions as can fit into shared memory and each partition will be sent to kernel sequentially.

In Table 4.16, the method for dividing coordinates is illustrated for a TSP with 9000 cities. To simplify calculations we will suppose that shared memory can store maximum 6000 cities. The size of partitions should be 6000 and each partition should include different combinations of city coordinates. In order to provide these combinations there should be two arrays of city coordinates (coordinates A and B) consisting of 3000 nodes in one array. Then for each partition the cities in these different arrays will be combined and 2-opt search will be applied to these combinations. The number of partitions will change with the number of cities in the problem. For 9000 nodes, there are three possible arrays of coordinates which are (6000,9000], (3000,6000] and (0,3000]. $\frac{3 \times 4}{2} = 6$ different combinations of these arrays accompany 6 partitions. Apparently, parallelism will be managed only within partitions, but not among partitions. In other words, partitions will be sent to kernel sequentially. For example, the second partition will be sent to kernel after the computations are performed on partition one.

Table 0.21 Division scheme of coordinates for the problem with 9000 cities

Partitions	Coordinates A	Coordinates B
1	(6000,9000]	(0,3000]
2	(3000,6000]	(0,3000]
3	(0,3000]	(0,3000]
4	(6000,9000]	(3000,6000]
5	(3000,6000]	(3000,6000]
6	(6000,9000]	(6000,9000]

As can be realized, the range of array coordinates A corresponds to “i” values in the kernel and the range of array coordinates B corresponds to “j” values. Thus in Partition 1, possible exchange effects between the cities in the range of “i= [6000, 9000]” and “j= [0, 3000]” are computed. As soon as kernel finishes its search in these 2 ranges combination, the city coordinates belongs to Partition 2 are sent to kernel. Then the possible exchange effects between the cities in the range of “i= [3000, 6000]” and “j=

[0, 3000]” are computed. After all the six partitions has been sent to the kernel, the necessary 2-opt exchange which decreases the tour cost most is performed and the same process is applied to the new tour.

To clarify the discussed method, it will be applied to a small sized example, which we analyzed before (see Example 4.1), assuming that the shared memory can have maximum 6 cities. Division of the city coordinates into partitions is illustrated in Table 4.17.

Table 0.22 Division scheme of coordinates for the problem in Example 4.1
(9 cities-45 possible exchanges)

Partitions	Coordinates A	Coordinates B
1	(6,9]	(0,3]
2	(3,6]	(0,3]
3	(0,3]	(0,3]
4	(6,9]	(3,6]
5	(3,6]	(3,6]
6	(6,9]	(6,9]

Exchanges performed sequentially are represented by different colors in Table 4.18. Jobs/exchanges belongs to the same partition are performed in parallel. It means that to complete one 2-opt search through all cities, 6 different city combinations (partitions) should be transferred to kernel and kernel should be called 6 times. There is a strong possibility that these sequential processes will extend the execution time of the algorithm.

Table 0.23 Calculated edge exchange effects depending on the coordinate ranges sent to kernel (see Example 3.1)

i \ j	1	2	3	4	5	6	7	8	9
1									
2	Part. 3 Id=0 (2; 1)								
3	Part. 3 Id=1 (3; 1)	Part. 3 Id=2 (3; 2)							
4	Part. 2 Id=3 (4; 1)	Part. 2 Id=4 (4; 2)	Part. 2 Id=5 (4; 3)						
5	Part. 2 Id=6 (5; 1)	Part. 2 Id=7 (5; 2)	Part. 2 Id=8 (5; 3)	Part. 5 Id=9 (5; 4)					
6	Part. 2 Id=10 (6; 1)	Part. 2 Id=11 (6; 2)	Part. 2 Id=12 (6; 3)	Part. 5 Id=13 (6; 4)	Part. 5 Id=14 (6; 5)				
7	Part. 1 Id=15 (7; 1)	Part. 1 Id=16 (7; 2)	Part. 1 Id=17 (7; 3)	Part. 4 Id=18 (7; 4)	Part. 4 Id=19 (7; 5)	Part. 4 Id=20 (7; 6)			
8	Part. 1 Id=21 (8; 1)	Part. 1 Id=22 (8; 2)	Part. 1 Id=23 (8; 3)	Part. 4 Id=24 (8; 4)	Part. 4 Id=25 (8; 5)	Part. 4 Id=26 (8; 6)	Part. 6 Id=27 (8; 7)		
9	Part. 1 Id=28 (9; 1)	Part. 1 Id=29 (9; 2)	Part. 1 Id=30 (9; 3)	Part. 4 Id=31 (9; 4)	Part. 4 Id=32 (9; 5)	Part. 4 Id=33 (9; 6)	Part. 6 Id=34 (9; 7)	Part. 6 Id=35 (9; 8)	
10	Part. 1 Id=36 (10; 1)	Part. 1 Id=37 (10; 2)	Part. 1 Id=38 (10; 3)	Part. 4 Id=39 (10; 4)	Part. 4 Id=40 (10; 5)	Part. 4 Id=41 (10; 6)	Part. 6 Id=42 (10; 7)	Part. 6 Id=43 (10; 8)	Part. 6 Id=44 (10; 9)

Number of sequential processes will rise with the increasing size of the problem. Sequential processes can be calculated via following formula:

$$noofdifferentranges = \frac{Numberofnodes/cities}{(sharedmemorycitystoragecapacity/2)} \quad (4.6)$$

$$noofparts = \frac{noofdifferentranges * (noofdifferentranges + 1)}{2} \quad (4.7)$$

Numbers of sequential processes of some different large sized problems are illustrated in Table 4.19. In despite of many sequential processes in a problem with 60000 cities, it is absolutely better than operating all $\frac{60000 \times 60001}{2} = 1800030000$ jobs sequentially.

Table 0.24 Calculation of sequential processes for several sized problems

	9000 cities	15000 cities	30000 cities	60000 cities
Number of different ranges/arrays	$\frac{9000}{(6000/2)} = 3$	$\frac{15000}{(6000/2)} = 5$	$\frac{30000}{(6000/2)} = 10$	$\frac{60000}{(6000/2)} = 20$
Number of partitions (different combinations of city ranges)	$\frac{3 \times 4}{2} = 6$	$\frac{5 \times 6}{2} = 15$	$\frac{10 \times 11}{2} = 55$	$\frac{20 \times 21}{2} = 210$

Modifications in the Device Code

The “distance” function should be modified like in Figure 4.14.

```

__device__ int distance (int i, int j, city_coords *coordsA, city_coords
*coordsB )
{
    register float dx, dy;
    // dx = coords[i].x - coords[j].x;
    // dy = coords[i].y - coords[j].y;
    dx = coordsA[i].x - coordsB[j].x;
    dy = coordsA[i].y - coordsB[j].y;
    return(int) (sqrtf(dx*dx+dy*dy)); }

```

Figure 0.26 Modification of the distance function for divided coordinates

Modifications in the Host Code

Two device arrays should be defined for coordinates A and coordinates B. Then the ordered coordinates should be copied to these arrays piece by piece. The code in Figure 4.15 generates all the possible range combinations via a, a_end, b, b_end variables. For the problem with 9000 cities, in the first loop “a” and “a_end” consecutively represent the beginning and the ending of the city range A in Partition 1. Similarly “b” and “b_end” consecutively represent the beginning and the ending of the city range B in Partition 1 (see Table 4.19). After the city coordinates in Partition 1 has been transferred to the last defined arrays and kernel has performed its jobs on them, the coordinates in the Partition 2 are copied to these arrays and kernel is invoked again. Until 6 parts are transferred to kernel, this process is repeated.

```

int a,b,a_end,b_end;
for( b=0; b<noOfNodes; b+=half_sm_capacity)
{
if(b>=(noOfNodes-half_half_sm_capacity))
    b=noOfNodes-half_half_sm_capacity;
for( a=(noOfNodes-half_sm_capacity); a>(b-half_sm_capacity);
a-=half_sm_capacity)
{
    if((a<b) || (b==(noOfNodes-half_sm_capacity)))
        a=b;
        a_end = a+half_sm_capacity;
        b_end = b+half_sm_capacity;
    cudaMemcpy(d_orderedCoordsA, orderedCoords+a,
sizeof(city_coords)*(half_sm_capacity+1), cudaMemcpyHostToDevice);
    cudaMemcpy(d_orderedCoordsB, orderedCoords+b,
sizeof(city_coords)*(half_sm_capacity+1), cudaMemcpyHostToDevice);
    kernel<<<gridDimX,blockDimX>>>(d_orderedCoordsA ,d_orderedCoordsB,
device_global_min, device_index, noOfNodes, noOfSwaps,
iter,a,a_end,b,b_end);

}
}

```

Figure 0.27 Additional code in the host code to divide coordinates

Note: “half_sm_capacity” represents the size of defined arrays for coordinates A and coordinates B. “half_sm_capacity” is equal to half of the shared memory capacity which is 3000.

Modifications in the Kernel Code

Figure 4.16 illustrates several modifications in the kernel code. Shared memory should be allocated for two arrays which represent coordinates of A and coordinates of B. Then the elements of ordered coordinates of A and B should be transferred the arrays in the shared memory.

```

*
//__shared__ city_coords c[6001];

__shared__ city_coords cA[3001];
__shared__ city_coords cB[3001];

**
//for(int i= threadIdx.x; i<noOfNodes; i+= blockDim.x)
// { c[i] = orderedCoords[i]; }

for(int i= threadIdx.x; i<(sm_capacity+1); i+= blockDim.x)
{
    cA[i] = orderedCoordsA[i];
    cB[i] = orderedCoordsB[i];
}

***
// change = distance(i,j,c)+distance(i-1,j-1,c)-distance(i-1,i,c)-
distance(j-1,j,c)

if(i>a & i<=a_end & j>b & j<=b_end)
{
    change = distance(i-a,j-b,cA,cB)+distance(i-1-a,j-1-b,cA,cB)-
distance(i-1-a,i-a,cA,cA)-distance(j-1-b,j-b,cB,cB);
}

```

Figure 0.28 Modification in kernel code for divided coordinates

Because “i” and “j” values are produced according to the number of cities, “i” should be restricted with the bounds of array “coordinates A” and “j” should be restricted with the bounds of array “coordinates B”. Thus, there is an additional “if” condition in order to control this. Moreover, while transferring the specific range of the data in the array of “orderedCoords” into the array of “orderedCoordsA” the indices of each element in the first array decreases as “a” unit. Therefore “a” is extracted from i and “b” is extracted from j in the formula of “change”.

Experimental Results for Big Sized Problems

Table 4.20 summarizes the best performances for each group when the problem size is “6000” and “15000”. Substantial increase in the problem size created shared memory restriction in each group. Because shared memory allowed only 1 active block, first group which has more threads in its single block compared to others, performed best performance. Although none of the groups manage to utilize all warps in the SMX, the first group with the highest occupancy performed better.

Table 0.25 Predicted best kernel launches vs. observed best kernel launches for TSP with 6000 and 15000 cities

CITIES:6000 EXCHANGES: 18003000							CITIES: 15000 EXCHANGES: 1112507500						
Predicted best kernel launches			Observed best launches				Predicted best kernel launches			Observed best launches			
Bloc k Dim	Grid Dim	iters	Bloc k Dim	Gri d Dim	iters	kerne l time (ms)	Bloc k Dim	Gri d Dim	iters	Bloc k Dim	Gri d Dim	iters	kerne l time (ms)
1024	2	8791	1024	1	1758 2	61	1024	2	5493 6	1024	1	10987 1	1172
512	4	8791	512	1	3516 3	117	512	4	5493 6	512	1	21974 1	2216
256	8	8791	256	1	7032 5	230	256	8	5493 6	256	1	43948 3	4369

4.5 2-Opt Algorithm with Initial Solution

We thought that starting 2-opt algorithm with a good solution can decrease the number of 2-opt iterations and accordingly shorten the total time of the algorithm. Because of this reason, we applied nearest neighborhood search to obtain an initial solution with good quality. Table 4.21 provides an overview of comparisons between algorithm with a good initial solution and algorithm with a naive initial solution. Experimental results in the table are in line with our expectations. Qualified initial solution decreased the number of 2-opt iterations. Therefore it decreased the total time (CPU+GPU time) of the algorithm although nearest neighborhood search is sequentially applied in CPU.

Table 0.26 Algorithm performances with a naive initial solution vs. with a good initial solution

No of Nodes	Naive Initial Solution				Nearest Neighborhood Initial Solution			
	Initial Cost	Minimized Cost	No of 2-opt iterations	Time (CPU+GPU)	Initial Cost	Minimized Cost	No of 2-opt iterations	Time (CPU+GPU)
500	380825	16278	533	181	278155	15707	417	173(ms)
1000	807793	35985	1094	1096	643904	39763	867	970(ms)
2000	1705333	60926	2166	7099	1486417	60521	1877	6276(ms)
2500	2348116	78383	2772	23011	1894196	77113	2273	19449(ms)
3000	3020898	91132	3268	50754	2622620	89700	2847	34611(ms)
3500	3774387	105356	3883	62493	3479454	103951	3396	55915(ms)
4000	4535148	121946	4462	93820	4224065	120219	3812	81460(ms)
LARGE SIZE DATA								
6000	31433083	10480046	1554	95925	24711210	8559520	1212	74821(ms)
9000	47099249	15206013	2362	493356	37822521	12344314	1876	392864(ms)

4.6 3-Opt Algorithm

3-opt algorithm is quite similar to 2-opt algorithm. The only difference is that 3 edges will be cut and reconnected this time. Figure 4.17 illustrates one 3-opt move.

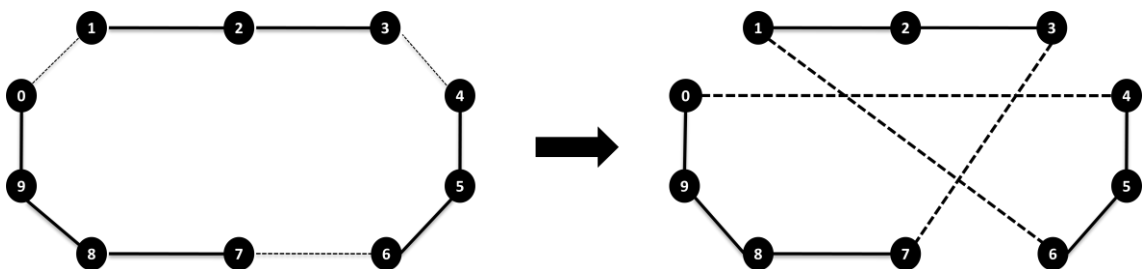


Figure0.293-opt exchange

Rocki and Suda obtained the 3-opt formulas in similar way to previous one. Because there are 3 edge exchanges in 3-opt, there should be a new id of “k” in addition to “i” and “j”. Table 4.22 demonstrates the id combinations for some of possible 3-opt edge exchange combinations. In this Table, the values in each cell consecutively represent the ids of “i”, “j” and “k”. Considering that in this table there are $\frac{n*(n+1)*(n+2)}{6}$ elements up to and including n^{th} row, Rocki and Suda proposed the Formula 4.8 for “i”.

$$n \{i\} = \sqrt[3]{3 * id + \sqrt{9 * id^2 - \frac{1}{9}}} + \sqrt[3]{3 * id - \sqrt{9 * id^2 - \frac{1}{9}}} + 1 \quad (4.8)$$

Table 0.27 Possible edge exchanges for 3-opt

2,1,0									
3,1,0	3,2,0	3,2,1							
4,1,0	4,2,0	4,2,1	4,3,0	4,3,1	4,3,2				
5,1,0	5,2,0	5,2,1	5,3,0	5,3,1	5,3,2	5,4,0	5,4,1	5,4,2	5,4,3

We achieved the formula of “j” based on “i” and job ids. (see Formula 4.9)

$$j = \frac{3 + \sqrt{8 * \left(id - \frac{i * (i - 1) * (i - 2)}{6} \right) + 1}}{2} \quad (4.9)$$

Based on “i”, “j” and job ids, we obtained the Formula 4.10 for “k”.

$$k = id - \frac{i * (i - 1) * (i - 2)}{6} - \frac{j * (j - 1)}{2} \quad (4.10)$$

Exception: The formulas above don’t work accurately when id is equal to “0”. We specified an “if” condition to solve this problem.

Note that “i”, “j” and “k” values should be rounded down to acquire integer values.

As an example these formulas are applied to a 3-opt exchange below:

$$\text{If id=2} \quad i = \sqrt[3]{3 * 2 + \sqrt{9 * 2^2 - 1/9}} + \sqrt[3]{3 * 2 - \sqrt{9 * 2^2 - 1/9}} + 1 = 3,49888 \cong 3$$

$$j = \frac{3 + \sqrt{8 * \left(2 - \frac{3 * 2 * 1}{6}\right) + 1}}{2} = 2 \text{ and } k = 2 - \frac{3 * 2 * 1}{6} - \frac{2 * 1}{2} = 0$$

Although these formulas give the accurate results up to some point, the formula of *i* starts to be problematic at id 454. Because “*j*” and “*k*” values depend on *i*, their formulas also don’t work. They cannot even produce any number because of inaccurate “*i*” values. We solved this issue fixing the formula of “*i*”.

As seen in Table 4.22,

There are $\frac{D*(n+1)*(n+2)}{6}$ elements up to and including *n*th row.

Since $\frac{1*2*3}{6} = 1$, in the first row there should be *one* “**2**”

Since $\frac{2*3*4}{6} - \frac{1*2*3}{6} = 3$, in the second row there should be *three* “**3**”

Since $\frac{3*4*5}{6} - \frac{2*3*4}{6} = 6$, in the third row there should be *six* “**4**”

Since $\frac{13*14*15}{6} - \frac{12*13*14}{6} = 91$, in the 13th row there should be *ninetyone* “14”. However, the problematic id 454, which is in the 13th row, gives the result as 15 when unrevised formula of “*i*” is applied. This situation leads to inaccurate “*j*” and “*k*” values. Similarly formula doesn’t give the correct results for 559th, 679th, 815th, 968th and some other indexes. It causes more missing results in the further indexes as illustrated in Table 4.23. This table includes some sample ids in which the formula produces inaccurate “*i*” values. The first two rows of this table show that the “*i*” values that should be produced in specific id gaps and the third row demonstrates the ids which gives inaccurate “*i*” values. For example, 6th column of the table explains that starting from id 3276 up to and including id 3653, “I” should be generated as “28”. However, 3652th and 3652th ids do not produce that value.

Table 0.28 Some problematic ids stem from the unrevised formula

id range	[364,454]	[455,559]	[560,679]	[680,815]	[816,968]	[3276,3653]	[30856,32508]
<i>i</i>	14	15	16	17	18	28	58
problematic ids	454	559	679	815	968	3652,3653	32505,32506 32507,32508

Thus, we modified the formula of “*i*” as follows:

$$\text{integer } i = \sqrt[3]{3 * id + \sqrt{9 * id^2 - 1/9}} + 1$$

Some of the generated “i”, “j” and “k” values through revised formula can be observed in the left Partition of Table 4.24. As realized some of the results are inaccurate, but at least “j” and “k” produce some values based on the “i” values obtained from the new formula. Thus, inaccurate results are controlled through a simple “if” statement. In the right Partition of the table, the corrected results are demonstrated. If “i” and “j” values are equal to each other, “i” value is incremented 1 unit and then using this updated “i” value “j” and “k” are calculated again.

Table 0.29 Results after fixing formula of “i”

Results of revised formula				Results after control statement		
id	i	j	k	i	j	k
1	2	2	0	3	1	0
2	3	2	0	3	2	0
3	3	2	1	3	2	1
4	3	3	0	3	1	0
5	4	2	0	4	2	0
6	4	2	1	4	2	1
7	4	3	0	4	3	0
8	4	3	1	4	3	1
9	4	3	2	4	3	2
10	4	4	0	4	1	0

5 CONCLUSION AND FUTURE RESEARCH

In this study we aimed to give some insights about parallelization strategies in CUDA and propose some methods to utilize GPU resources in a most advantageous way. Considering the parallelization approaches of Rocki and Suda (2012), best improvement 2-opt and 3-opt local search algorithms were accelerated for solving Travelling Salesman Problem. Process of searching neighborhood which means calculating the effect of each possible edge exchange in the tour made in parallel. We performed detailed performance analysis on 2-opt search algorithm configuring kernel parameters in different ways. The best performances are obtained when all resident warps in SMX are utilized which provides 100% occupancy of device. However shared memory started to restrict the usage of all possible active warps in SMX with the growing size of the problem. In this situation the best strategy was utilizing all resident warps that shared memory allows. To sum up, we should keep the warps in device busy as far as possible which provides thread level parallelism. If the number of edge exchanges is greater than the total number of threads in launched warps, all exchange effect calculations should be distributed among the launched threads equally. It means that one thread will perform more than one exchange effect calculation concurrently and this accompanies iteration level parallelism. In implemented algorithm after exploiting thread level parallelism by increasing occupancy until possible highest level, iteration level parallelism should be applied. For large sized problems fast on-chip shared memory wasn't enough to store all city coordinates. Thus, the coordinates are divided into partitions with the proposed technique of Rocki and Suda (2013). Moreover we compared the 2-opt algorithm with a nearest neighborhood initial solution and with a naive initial solution. We observed that qualified initial solution decreased the number of 2-opt iterations and correspondingly decreased the total time (CPU+GPU) of the algorithm although nearest neighborhood search is sequentially applied in CPU. Lastly we modified the 3-opt formula proposed by Rocki and Suda (2012). Although it was working for many index calculations, it was giving inaccurate results for some of them.

In addition to 2-opt and 3-opt search methods we also accelerated exchange of two nodes and relocate algorithms in similar way. In the future, by combining all algorithms in this study we are planning to implement Variable Neighborhood Search on Travelling Salesman Problem to get solutions with better quality. As can be observed, at some point the search method used cannot improve the tour cost further. In order to approach optimal solution more, we will switch into another search method which can improve the current solution. We will combine several methods and try to find the best combination method with regards to time and solution performance. Because GPU has peak performances, one may want to reach better solutions in a bit more time and variable neighborhood can help to get them.

APPENDICES

Appendix A: 2-Opt Algorithm Results for Different Kinds of Resource Allocations

TableError! No text of specified style in document.-1 2-opt performance for a TSP tour with 500 cities

Block Dimension	Grid Dimension	Number of Iterations	GPU Time (ms)	Minimized Cost	Number Of 2-Opt Iterations	CPU + GPU Time (ms)
1024	122	1	0.369	16298	530	428
1024	61	2	0.295	16304	529	354
1024	2	61	0.205	16278	533	181
1024	1	122	0.333	16148	543	318
512	244	1	0.348	16304	529	264
512	122	2	0.281	16304	529	222
512	4	61	0.212	16298	529	189
512	2	122	0.334	16304	529	249
512	1	244	0.635	16304	529	425
256	488	1	0.366	16304	529	371
256	244	2	0.280	16304	529	301
256	122	4	0.246	16304	529	337
256	8	61	0.209	16193	539	318
256	4	122	0.335	16304	529	267
256	1	488	1.246	16304	529	888
128	975	1	0.624	16304	529	422
128	488	2	0.458	16304	529	423
128	244	4	0.386	16304	529	342
128	122	8	0.353	16304	529	319
128	16	61	0.342	16196	538	313
128	8	122	0.337	16304	529	327
128	1	975	2.469	16304	529	1426

Table Error! No text of specified style in document.-2 2-opt performance for a TSP
tour with 1000 cities

Block Dimension	Grid Dimension	Number of Iterations	Kernel Time (ms)	Minimized Cost	Number Of 2-Opt Iterations	CPU + GPU Time (ms)
1024	484	1	1.556	36193	1091	1914
1024	244	2	1.220	34547	1081	1539
1024	122	4	1.024	34370	1084	1357
1024	61	8	0.921	34546	1082	1201
1024	2	242	0.776	35985	1094	1096
1024	1	484	1.310	36188	1092	1667
512	967	1	1.645	36534	1090	2078
512	488	2	1.216	34547	1081	1592
512	244	4	1.008	34499	1088	1381
512	122	8	0.909	34546	1081	1206
512	61	16	0.869	34691	1074	1234
512	4	242	0.791	36197	1090	1067
512	2	484	1.309	36193	1091	1682
512	1	967	2.515	36539	1090	3021
256	1934	1	2.969	36544	1089	3473
256	967	2	2.129	36543	1091	2518
256	488	4	1.743	34576	1083	2098
256	244	8	1.547	34546	1082	1860
256	122	16	1.453	34691	1074	1770
256	61	32	1.422	34455	1075	1739
256	8	242	1.311	36503	1080	1770
256	4	484	1.309	36193	1091	1742
256	1	1934	4.964	36544	1089	5569
128	3868	1	6.625	36539	1090	7475
128	1934	2	4.548	36548	1090	5227
128	975	4	3.588	34395	1083	4082
128	488	8	3.069	34546	1082	3534
128	244	16	2.796	34691	1074	3259
128	122	32	2.697	34455	1075	3099
128	16	242	2.535	36404	1085	3004
128	8	484	2.529	36000	1093	3017
128	4	967	2.515	36544	1089	3036
128	1	3868	9.866	36544	1089	10896

Table Error! No text of specified style in document.-3 2-opt performance for a TSP
tour with 1500 cities

Block Dimension	Grid Dimension	Number of Iterations	GPU Time (ms)	Minimized Cost	Number Of 2-Opt Iterations	CPU + GPU Time (ms)
1024	1098	1	3.963	47202	1639	6863
1024	488	3	2.787	47587	1646	4596
1024	244	5	2.326	47173	1646	4167
1024	61	18	1.939	47542	1635	3483
1024	2	549	1.739	47585	1645	3262
1024	1	1098	2.944	47715	1643	5170
512	2196	1	4.824	47297	1654	8411
512	1098	2	3.510	47431	1646	6128
512	488	5	2.776	47171	1647	4966
512	244	9	2.475	47599	1630	4376
512	61	36	2.254	47312	1635	4055
512	4	549	3.005	47513	1636	5329
512	3	732	2.125	47293	1657	3935
512	2	1098	2.947	47715	1643	5224
512	1	2196	5.697	47297	1654	9662
256	4392	1	10.927	47297	1654	18436
256	2196	2	7.090	47431	1646	12171
256	1098	4	5.494	47173	1642	9269
256	488	9	4.546	47599	1630	7827
256	244	18	4.266	47624	1629	7270
256	61	72	4.060	47249	1637	7007
256	8	549	4.328	47615	1570	7067
256	4	1098	5.695	47592	1642	9594
256	3	1464	3.851	47297	1657	6693
256	1	4392	11.244	47297	1654	18729
128	8784	1	24.467	47297	1654	40994
128	4392	2	15.982	47466	1644	26675
128	2196	4	11.861	47173	1642	19753
128	1098	8	9.663	47247	1634	16168
128	488	18	8.568	47532	1633	14249
128	244	36	8.053	47312	1635	13489
128	61	144	7.885	47455	1650	13323
128	16	549	8.479	47603	1580	13610
128	8	1098	8.472	47511	1651	14244

128	3	2928	7.515	47604	1643	12655
128	1	8784	23.458	47297	1654	38780

Table Error! No text of specified style in document.-4 2-opt performance for a TSP tour with 2000 cities

Block Dimension	Grid Dimension	Number of Iterations	GPU Time (ms)	Minimized Cost	Number Of 2-Opt Iterations	CPU + GPU Time (ms)
1024	1953	1	8.025	61999	2136	17532
1024	977	2	5.552	61240	2171	12537
1024	488	5	4.428	61391	2159	10059
1024	244	9	3.839	60869	2179	8830
1024	61	33	3.309	61266	2140	7466
1024	2	977	3.093	60926	2166	7099
1024	1	1953	5.224	61359	2129	11477
512	3905	1	14.766	60768	2214	33141
512	1953	2	10.097	61913	2134	21924
512	977	4	7.655	61156	2161	17125
512	488	9	6.470	60967	2194	14658
512	244	17	5.885	60918	2182	13285
512	61	65	5.490	61183	2171	12274
512	4	977	5.265	60698	2194	11811
512	2	1953	5.216	61120	2164	11654
512	1	3905	10.109	61336	2128	21763
256	7809	1	30.611	61956	2137	65936
256	3905	2	20.495	61913	2134	44092
256	1953	4	15.229	61253	2136	33095
256	977	8	12.685	61226	2135	27616
256	488	17	11.497	61320	2158	25176
256	244	33	10.810	61337	2137	23334
256	122	65	10.467	61510	2149	22777
256	8	977	10.144	60758	2205	22611
256	4	1953	10.091	61276	2187	22418
256	2	3905	10.090	61336	2128	21832
256	1	7809	20.026	60941	2169	43408
128	15618	1	76.194	61956	2137	163289
128	7809	2	48.314	61913	2134	103469
128	3905	4	34.036	61253	2136	73465
128	1953	8	27.173	61021	2156	58938
128	977	16	23.616	60942	2196	52301

128	488	33	21.830	61339	2134	46835
128	244	65	20.933	61510	2149	45153
128	122	129	20.351	61106	2176	44535
128	16	977	21.984	61144	2170	42611
128	2	7809	19.897	61359	2129	42611
128	1	15618	39.655	61359	2129	84226

Appendix B: Results for 2-Opt Large Sized Data and 3-opt Algorithms

Table Error! No text of specified style in document.-5 Best 2-Opt Results for Large-Sized Data

# of Cities	Block Dim	Grid Dim	iters	kernel time (ms)
6000	1024	1	17582	71.420
15000	1024	1	109871	1588.713
30000	1024	1	439468	21681.576
45000	1024	1	988792	104357.210
60000	1024	1	1757842	321000.937

Table Error! No text of specified style in document.-6 3-Opt Results for Different Sized Data

# of Cities	kernel time (ms)	Minimized Cost	Number Of 2-Opt Iterations	CPU + GPU Time (ms)
100	5	5242	83	396
200	19	7969	175	2318
300	33	10333	270	8714
400	61	12960	342	21012
500	103	15998	466	50732
600	175	18909	557	100612
700	270	22783	632	176567
800	358	25945	731	270441
900	502	28570	836	437143

1000	631	34945	963	630951
------	-----	-------	-----	--------

Appendix C : CUDA Code

```

//KERNEL FUNCTION (DEVICE CODE)
__global__ void kernel( unsigned short *tour, city_coords * coords, int * global_min,
int * index, unsigned intnoOfNodes, unsigned intnoOfSwaps, unsigned intiter)
{
//variable for the ids of threads launched in SM
intidx=blockDim.x*blockIdx.x + threadIdx.x;

//the number of threads in SM
registerintpackSize = blockDim.x*gridDim.x;

registerint i, j, change, id;
register intlocal_min_change = 0;

//Allocating shared memory for tour order and city coordinates.
__shared__ unsigned short t[noOfNodes+1];
__shared__ city_coords c[noOfNodes+1];

//transferring elements of the tour order and city coordinates to shared memory
for(int i= threadIdx.x; i<noOfNodes; i+= blockDim.x)
{ t[i] = tour[i];
c[i] = coords[i]; }
t[noOfNodes]= tour[noOfNodes];
__syncthreads();

//loop to assign multiple jobs to a thread
for(register int no=0; no<iter; no++)
{

//Calculating the ids of total jobs
id = idx + no*packSize;
if(id<noOfSwaps)
{

//calculating the index of the all possible node pairs
i=int(3+sqrtf(8.0f*(float)id+1.0f))/2;
j=id-(i-2)*(i-1)/2+1;

//Calculating the edge exchange effect for each node pair in parallel
change = distance(t[i],t[j],c) + distance(t[i-1],t[j-1],c) - distance(t[i-1],t[i],c)
- distance(t[j-1],t[j],c);

```

```

//Finding the minimum change among all possible exchanges
if(change<local_min_change)
{
local_min_change = change;
atomicMin(&global_min[0], change);
}

//finding the index of the minimum change
if(change == global_min[0])
index[0] = id;
}
}
}
// HOST CODE
constintnoOfNodes;
unsignedintnoOfSwaps = noOfNodes*(noOfNodes-1)/2;
unsignedintiter;
unsigned short * tour = new unsigned short[(noOfNodes+1)];
city_coords *coords = (city_coords*)malloc(sizeof(city_coords)*noOfNodes);
int * tour_cost = new int[1];
int *global_min = new int[1];
int * index = new int[1];

// defining device variables and arrays
unsigned short * d_tour;
city_coords *d_coords;
int * device_global_min;
int * device_index;

//allocating device memory for the device variables and arrays
cudaMalloc( (void**) &d_tour, sizeof(unsigned short)*(noOfNodes+1)) ;
cudaMalloc( (void**) &d_coords, sizeof(city_coords)*noOfNodes);
cudaMalloc( (void**) &device_global_min, sizeof(int)*1) ;
cudaMalloc( (void**) &device_index, sizeof(int)*1) ;

//reading the city coordinates from the ".txt" file
intvertex_sentinel = 0;
while (vertex_sentinel<noOfNodes)
{
DataFile>>vertex;
DataFile>>coord_x;
DataFile>>coord_y;
coords[vertex-1].x=coord_x;
coords[vertex-1].y=coord_y;
vertex_sentinel++;
}

//transferring the city coordinates from host to device
cudaMemcpy(d_coords, coords, sizeof(city_coords)*noOfNodes, cudaMemcpyHostToDevice);

// storing current order of travelling salesman tour into the tour order array.
for(int i=0; i<noOfNodes; i++)
{ tour[i]=i; }
tour[noOfNodes] =0;

```

```

tour_cost[0] = 0;
for(int i=0; i<noOfNodes; i++)
tour_cost[0] = tour_cost[0] + dist(tour[i], tour[i+1],coords);

// configuring the block dimension, grid dimension, number of iterations.
intblockDimX = ;
intgridDimX = ;
iter = ;

global_min[0]=-1;

//call kernel function as long as there is an improvement in the current tour cost
while(global_min[0] < 0)

{
count = count +1 ;
global_min[0] = 0;

index[0] =0;

//Transferring the minimum change and current tour order from host to device.
cudaMemcpy(device_global_min,global_min,sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(d_tour,tour,sizeof(unsigned short)*(noOfNodes+1),cudaMemcpyHostToDevice);

//Invoking the kernel function
kernel<<<gridDimX,blockDimX>>>(d_tour,d_coords,device_global_min,device_index,
noOfNodes, noOfSwaps,iter);

//Host Code Continued

// Transferring the minimum "change" value and the index of it from device to host.
cudaMemcpy(global_min, device_global_min, sizeof(int)*1, cudaMemcpyDeviceToHost);
cudaMemcpy(index, device_index, sizeof(int)*1, cudaMemcpyDeviceToHost);

//Performing the edge exchange in the current tour according to the index
int id = index[0];
int i=int(3+sqrtf(8.0f*(float)id+1.0f))/2;
int j=id-(i-2)*(i-1)/2+1;
int swap2 = i-1;
int swap1 = j;
while(swap1 != swap2 && swap1<swap2)
{
int a = tour[swap1];
tour[swap1] = tour[swap2];
tour[swap2]= a;
swap1++;
swap2--;
}

```

BIBLIOGRAPHY

Brodtkorb, A. R., Hagen, T. R., Schulz, C., & Hasle, G. (2013). GPU computing in discrete optimization. Partition I: Introduction to the GPU. *EURO Journal on Transportation and Logistics*, 2(1-2), 129-157.

Christopher Coopern (2011), "GPU Computing with CUDA Lecture 2 - CUDA Memories", <http://www.bu.edu/pasi/files/2011/07/Lecture2.pdf>, accessed on July 2013

Coelho, I. M., Ochi, L. S., Munhoz, P. L. A., Souza, M. J. F., Farias, R., & Bentes, C. (2012). The single vehicle routing problem with deliveries and selective pickups in a CPU-GPU heterogeneous environment. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on* (pp. 1606-1611). IEEE.

Cornell University Virtual Workshop, "Introduction to GPGPU and CUDA Programming" <https://www.cac.cornell.edu/vw/gpu/structure.aspx>, accessed on July 2013

<https://en.wikipedia.org/wiki/2-opt>, accessed on July 2013

Janiak, A., Janiak, W. A., & Lichtenstein, M. (2008). Tabu Search on GPU. *J. UCS*, 14(14), 2416-2426.

Kirk, D. B., & Wen-me, W. H. (2012). *Programming massively parallel processors: a hands-on approach*. Newnes.

NVIDIA CUDA Programming Guide, “Compute Capabilities”
<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>, last accessed on July 2013

Nvidia, C. (2012). *NVIDIAs next generation CUDA compute architecture: Kepler GK110*. Technical report.

O’neil, M. A., Tamir, D., &Burtscher, M. (2011, July). A parallel gpu version of the traveling salesman problem. In *2011 International Conference on Parallel and Distributed Processing Techniques and Applications* (pp. 348-353).

Oracle Corporation, <http://docs.oracle.com/cd/E19455-01/806-5257/mtintro-6/index.html>, last accessed on July 2013

Oxford University, “GPU Parallelizable Methods”, <http://www.oxford-man.ox.ac.uk/gpu/ss/simd.html>, last accessed on July 2013

Prinslow, G. (2011). Overview of Performance Measurement and Analytical Modeling Techniques for Multi-core Processors. URL: <http://www.cse.wustl.edu/~jain/cse567-11/ftp/multicore.pdf>.

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

Rocki, K., &Suda, R. (2013, May). High Performance GPU Accelerated Local Optimization in TSP. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* (pp. 1788-1796). IEEE.

Ruetsch, G., &Fatica, M. (2013). *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming*. Elsevier.

Schulz, C. (2013). Efficient local search on the GPU—investigations on the vehicle routing problem. *Journal of Parallel and Distributed Computing*, 73(1), 14-31.

Talbi, E. G. (2009). Parallel Local Search on GPU.

Van Luong, T., Melab, N., & Talbi, E. G. (2013). GPU computing for parallel local search metaheuristic algorithms. *Computers, IEEE Transactions on*, 62(1), 173-185.

Virginia Tech Advanced Research Computing, “CUDA Programming Model” <http://www.arc.vt.edu/resources/software/cuda/>, last accessed on July 2013

Volkov, V. (2010, September). Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC* (Vol. 10).

Wen-mei W. Hwu, Illinois University Online Course, “Heterogeneous Parallel Programming”, <https://www.coursera.org/course/hetero>, last accessed on July 2013