2019-02-28

Accelerating Sequence Calculations on Parallel GPU Architecture

Hossain, Roksana

UNIVERSITY OF CALGARY

Accelerating Sequence Calculations on Parallel GPU Architecture

by

Roksana Hossain

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

FEBRUARY, 2019

Abstract

In this thesis, I have implemented a GPU (graphics processor unit) based sequencing algorithm that finds a sequence or order among data points to optimize a given objective. I have studied the sequencing algorithm as a path planner for an unmanned aerial vehicle (UAV) and also, a basecaller for a miniature DNA sequencer. Parallel implementation utilizing GPU enables faster processing and decision making that are important when data is quite large and a real-time response is critical *e.g.*, in UAV based transportation.

The goal of using UAV in my thesis is to construct a wireless sensor network in a remote location by deploying wireless sensor nodes. The proposed path planner, also known as a sequencer, is designed to find the shortest path that is also a safe path using travelling salesman problem. A path is considered safe when the vehicle would not collide with any obstacle. Two sets of heuristic algorithms, one for generating a sequence of waypoints (***sequence generator***) and another one for constructing a path between two waypoints (***path explorer***), are used to find the near optimal solution. The highly parallel multicore GPU is used for the real-time implementation that offloads compute-intensive portions from the traditional CPU to the GPU to make the decision-making process faster. In this thesis, the parallel execution of the sequence generator and the path explorer achieved a 4.82× and 164× speed-up compared to the CPU-only approach respectively.

The second sequencer is a palm-sized miniature DNA sequencer, the so-called MinION device. In the case of the MinION, a vast multitude of DNA strands are introduced into the device and converted into noisy electronic time-series signals; these measurements are essentially physical signatures related to the molecular make-up of the sensed DNA. Among a long "pipeline" of analysis steps to be performed on such measurement sequences, the first, and arguably most intensive, is the so-called ***basecalling*** step which analyzes the time-series and converts it into the equivalent monomeric base of the DNA under test using the Viterbi algorithm.

Acknowledgements

I want to express my sincere gratitude to my supervisors Dr. Sebastian Magierowski and Dr. Geoffrey Messier for providing their invaluable guidance, comments, and suggestions throughout the research project. I would specially thank Dr. Sebastian Magierowski for continually motivating me to work harder and supervising me from Toronto.

I am grateful to my family for their continuous support as it wasn't an easy journey for me. I like to thank my husband, Monir, for all of his help including teaching me algorithms. Without Monir's support, I would not have completed my degree. I also love to appreciate my little one, Nora; for allowing me to work on my thesis. Special thanks to my mother who always motivated me to complete the degree. I am also thankful to my other family members and friends who have supported me along the way.

Extraordinary gratitude goes out to all down at Alberta innovates technology futures and Queen Elizabeth II fund for helping and providing the funding for the work.

Table of Contents

List of Figures and Illustrations

List of Tables

List of Symbols, Abbreviations and Nomenclature

Symbol or abbreviation	Definition
GPU	Graphics processing unit
GPGPU	General purpose graphics processing unit
CUDA	Compute unified device architecture
OpenCL	Open computing language
VTOL	Vertical take-off and landing
WSN	Wireless sensor network
UAV	Unmanned aerial vehicle
TSP	Travelling salesman problem
PRM	Probabilistic roadmap
NGS	Next-generation-sequencing
CMOS	Complementary metal oxide semiconductor
DNA	Deoxyribonucleic acid
AWS	Amazon web services
CPI	Cycles-per instruction
GPS	Global positioning system
NP	Nondeterministic polynomial time
RRT	Rapidly exploring random tree
NBS	Nanopore-based sequencer
DRAM	Dynamic random access memory
GDDR5 SGRAM	Graphics double data rate five synchronous graphics random access memory
SMEM	Shared memory
PCIe	Peripheral component interconnect express
ILP	Instruction-level parallelism
PLP	Processor-level parallelism
SP	Streaming processor
SM	Streaming multiprocessor
SFU	Special Function Unit
IMU	Inertial measurement unit
GA	Genetic algorithm
SIMD	Single instruction multiple data

Symbols from Chapter 4 to 6:

x	x coordinate
y	y coordinate
z	z coordinate
D_{matrix}	Distance matrix
v_i	i-th waypoint
n	Total number of total waypoints
$\{v_i\}$	A set of waypoints
V	A path which is a set of sequenced waypoints i.e. $V = \{v_i\}$
$\{ob_j\}$	A set of obstacles in the environment and $j = 1, 2 \ldots q$
q	Total number of obstacles
$Update_{TSP}$	A boolean variable
$Update_{PRM}$	A boolean variable
E	A set of all possible edges connecting two different waypoints
x_{ij}	A boolean variable
c_{ij}	Cost associated for travelling from waypoint v_i to v_j
k	Total number of generated paths
$\{V_j\}$	A set of generated paths, where $j = 1, 2, \ldots k$
C	Total distance for a path
$\{V'\}$	A set of best paths
$\{V''\}$	A set of regenerated paths
P	The best path, a set of sequenced waypoints, $(v_i : i = 1, 2, \ldots n)$
t	Number of threads
m	Number of clusters
CH_j	jth cluster head, $j = 1, 2, \ldots m$
$\{CH_j\}$	A set of cluster heads
CH_{v_i}	The cluster head assigned to v_i
u	Number of combinations or population for a cluster
v_{start,CH_j}	The start waypoint inside j-th cluster using nearest neighbour technique
v_{end,CH_j}	The end waypoint inside j-th cluster using nearest neighbour technique
b_w	Number of bytes required by a waypoint
b_d	Number of bytes required by a distance matrix
b_b	Number of bytes required by a barrier vector element
n_{trd}	Number of threads per block
n_{block}	Number of blocks
C_{serial}	Length of the paths (serial implementation)

$C_{parallel}$	Length of the paths (parallel implementation)
s	Sample nodes
w	Total number of sample nodes for each segmented path
$\{s_r\}$	A set of sample nodes, where $r = 1, 2 \dots w$
p	Path for each segment between start to destintion
d	Distance of the segmented path, p
Symbols from Chapter 7:	
$\boldsymbol{s}$	The sequence of states
N	Number of observed events
$\boldsymbol{e}$	Observed events
L	Strand length
$f(\boldsymbol{s}, \boldsymbol{e})$	A joint probability density model (pdf)
$P(\boldsymbol{s})$	Probability of prior state sequences
$f(\boldsymbol{e}\|\boldsymbol{s})$	The likelihood model
$f(e_i\|s_i)$	The emission probability: the pdf of an event e_i in response to a nanopore state s_i
$\tau(s_{i-1}, s_i)$	The transition probability: the probability of s_{i-1} transitioning to s_i
$v_i(e_i, s_i)$	The sequence posterior at each index i
M	All the possible states that the nanopore can assume at any particular measured event e_i
b	A base
$\mathcal{N}$	Guassian (normal) distribution
IG	Inverse-Gaussian distribution
x_i	Mean value of event
y_i	Standard deviation of event
μ_j	Model level mean for state j
σ_j	Model level standard deviation for state j
η_j	Model spread mean for state j
λ_j	Shape parameter

Chapter 1

Introduction

Sequence calculation, the selection or identification of a desirable sequential pattern from within a complex space, is a common challenge faced by automation technology. It touches on subjects as diverse as machine learning, robotics, social networks, biomedical applications, etc. In the case of decision-making, a problem formulation that invites sequence calculations considers all possible outcomes and seeks to compute an optimal sequence of decisions needed to realize the most desirable result. Depending on the problem or application, the number of possible outcomes as well as intermediate decisions can vary profoundly.

Often this quantity of outcomes is too large to work with using commodity computational devices such as the ubiquitous microprocessor for applications with real-time performance pressures. In such cases, a limited number of probable outcomes are considered therefore sacrificing accuracy or fidelity. Naturally, the better the processor, the more outcomes may be considered within a given time. To address this problem the graphics processing unit (GPU) is used here.

In general, GPUs are used for graphics related operations. As such they accelerate the graphical computations dominant in consumer electronic devices such as traditional computers and mobile devices. For example, GPUs enabled the fast execution of RenderScript code on the Google Android 4.2 device [4], and Apple introduced their own Apple A11 GPU

device with an API named, Metal [5] for the iPhone. GPUs mainly achieve this acceleration by realizing many 100s of simple cores per die with memory to efficiently handle graphics problems.

In addition, some GPUs allow us to do our own programming on applications of our own choosing, a development spurred by GPU makers who realized the potential of such units beyond graphics alone. Such GPUs are typically referred to as general purpose graphics processing units (GPGPUs) [6]. Companies such as NVIDIA and AMD are now famous for providing such devices to end users. Indeed, such products have been instrumental in the explosive growth of specific machine learning applications, particularly those targeting the deep learning paradigm [7, 8], and have inspired the parallel computing research reported in this thesis as well.

The GPGPU offloads compute-intensive portions of an application from the traditional central processing unit (CPU) to the GPU co-processor, while the remainder of the code still runs on the CPU [9]. In this thesis, for convenience, I do not distinguish between GPUs and GPGPUs as many so-called GPU graphics cards support custom programming. The two main and generally similar programming frameworks for GPGPU computing are CUDA which is NVIDIA's own proprietary programming model and OpenCL which is supported by AMD, Intel, NVIDIA, etc. [10, 11]. In this thesis, I employ the CUDA programming model which was released before OpenCL.

Although GPU use has escalated tremendously in the last decade, it is arguable that a large part of the applications in which it has met considerable success has been confined to problems naturally suited for massively parallel machines. That is, a set of relatively independent calculations that can be done in parallel. These include compute paradigms [12] such as map, scan, reduce, gather, scatter, stencil, etc. ***But how can GPU technology be used in applications like sequence calculation without such a clear-cut mapping onto the traditional GPU framework?***

This is the key question addressed by this thesis. Sequence calculation is used in many

applications like product assembly [13, 14] and disassembly [15], task sequencing [16] for robotics, path planner for autonomous vehicles [17, 18], DNA sequencing [19, 20] etc. Among these applications, this thesis dealt within the context of two emerging embedded-systems applications: autonomous vehicles and biomolecular detectors. Namely, in this thesis, the GPU is used to accelerate the path sequence and path planning calculation for an unmanned aerial vehicle (UAV) and the basecalling step of the DNA sequencing pipeline. Now the question will be ***how can a path planner be related to a DNA sequencer?*** The first similarity between these problems is that both problems require a sequence or order of data points to optimize individual objective. The objective of path planning is to minimize the traveled distance by ordering waypoints. The objective of DNA sequencing is to maximize the likelihood of the genomic sequence. Aside from the sequence calculation, these applications share two other important constraints increasingly encountered in practice: they are deployed in small resource constrained devices and solutions are required in real-time. These added complexities stress the importance of realizing local compute enhancement that can operate in an embedded fashion.

Given the similarities between the path planner and the DNA sequencer, the question will be ***whether these problems are exactly the same***. Although there are similarities, these are not exactly the same problem. The last hypothesis of the thesis studies ***if the problems can be solved using the same algorithm or different algorithms***. The next few sections will address these questions by outlining details of the path planner for UAV and DNA sequencing.

1.1 Sequence Synthesis: Path Planning for UAV

In recent years, due to improvements in technology, robots have been replacing human operators. An example of such replacement includes robotic arms in manufacturing industries [21], medical surgeries [22, 23], robotic vehicles [24, 25, 26] etc. In this vein, one of the subjects

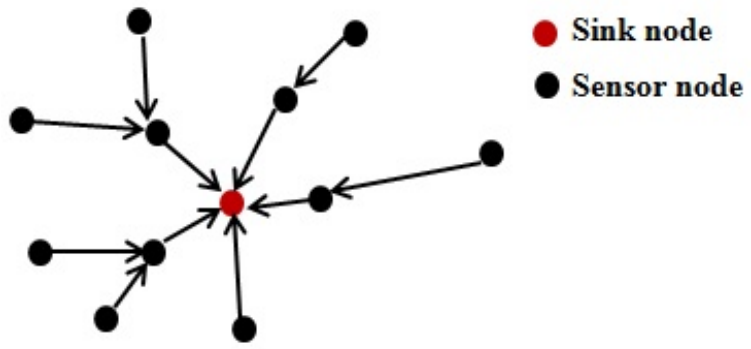


Figure 1.1: Wireless sensor network: example of data funnelling to a sink node.

of this thesis is the UAV, which is a robotic vehicle that can fly without the assistance of a human pilot.

This thesis concentrates on the fast computation of path planning for small vertical take-off and landing (VTOL) UAVs that build wireless sensor networks (WSNs) [27] by dropping sensor nodes at given locations. The decision regarding where the sensor nodes will be deployed is made at the stage of designing the network. The sensor nodes can communicate with each other by transmitting and receiving data. For WSNs, the ultimate goal is often to collect sensing data from all sensors and funnel it to specific *sink nodes* and perform further analysis at the sink nodes. Thus, data collection is one of the most common services used in sensor network applications. Fig. 1.1 shows an example of the data collection process in a WSN. In the figure, a single sink node, red coloured, collects sensing values from every sensor using a collection tree.

The quality of the network depends on the several criteria that make the network construction challenging [27]. Each node has a limited transmission range so the distance between the nodes should not exceed this range. The quality of the data transfer among the nodes depends on the distance between the nodes as well as the physical environment [27]. Due to the limited battery life of the nodes, all of the nodes should be deployed as quickly as possible to get the most efficient performance. Otherwise, if there is a long time interval between the deployments of two sets of nodes in the same network, the batteries of the first set of nodes start discharging before the completion of the network construction. This kind

of unexpected situation reduces the total lifetime of the network.

The advantage of using a robotic helicopter to construct a wireless sensor network is that helicopters can be sent to remote locations instead of human operators. The helicopter can fly close to the ground allowing it to deploy nodes accurately. Flying close to the ground increases the number of obstacles on the path. In such a situation, the benefit of using a helicopter over fixed-wing aerial vehicles is that the former can hover if it finds an obstacle in front of it. In short, the helicopter posses the ability to achieve much finer control over its flight path.

However, a UAV requires a path planner to direct it towards the sequence of desired locations where it will deploy a WSN's sensor nodes. The path planners can be subdivided into two components: a ***global path planner*** and a ***local path planner***. In general, the global path planner finds a path from the starting location to the destination so that the UAV will not hit any obstacles. The local path planner generates the required steering controls so that the UAV can follow the path generated by the global path planner.

In this thesis, a unique global path planner is proposed and designed that allows a VTOL UAV to deploy wireless sensor nodes so that the total distance traveled by the vehicle is minimized. In Fig. 1.2, there are three destination locations, or waypoints shown as blue stars; obstacles are shown in red. In such a cluttered environment (an environment that contains obstacles) the global path planner produces a safe path as illustrated with green dots.

In this thesis, the locations where the sensor nodes are to be deployed, the ***waypoints***, are given to the UAV software control system at mission start. As this thesis considers multiple sensor nodes, the global path planner requires finding a path through various waypoints. In such situations, the UAV decides over which sequence it should deploy the sensor nodes as the length of the path through all the waypoints varies depending on the sequence.

The proposed global path planner finds the sequence of waypoints by essential solving the travelling salesman problem (TSP) [28] and finds the flying path between two waypoints

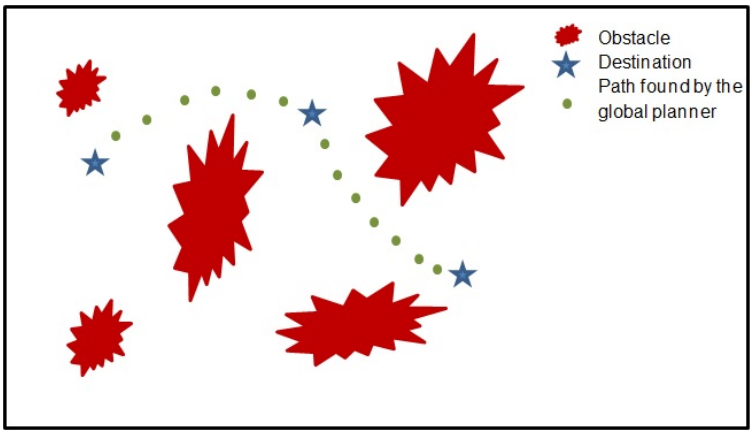


Figure 1.2: Path found by the global planner in a cluttered environment.

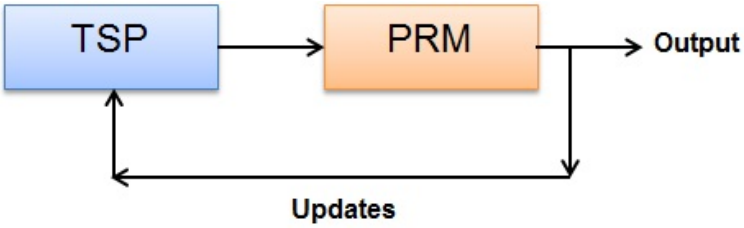


Figure 1.3: Global path planner contains two blocks working on TSP and PRM.

without hitting any obstacle by way of probabilistic roadmap (PRM) methods [29] as shown in Fig. 1.3. A detailed discussion on how the TSP and the PRM work together to generate a final path is discussed in Chapter 4. In this thesis, both the TSP and the PRM have been implemented separately and are discussed in Chapter 5 and 6 respectively.

For the global path planner, both the sequence generator and the path explorer require a significant amount of computational effort. To make a quick final decision from the UAV software system blocks, each block should respond fast with the correct output. Making a quick final decision also saves a UAV from hitting an unmapped obstacle. Parallel computation is used to reduce the computational time for the proposed global path planner. Successful implementation of the proposed global path planner can be used in different robotic applications. Examples of such applications can vary as widely as military operations with UAVs in the battlefield performing a search over multiple target locations to domestic setting where flying robots deliver parcels to different addresses, robots conveying groceries to any grocery

centre or collecting numerous books from a library system, monitoring gas, oil and water pipelines etc.

1.2 Sequence Analysis: Real-Time DNA Sequencing

The complement to the sequence synthesis (construction) problem above which seeks to realize an autonomous vehicle traversal pattern over a set of discrete labels (i.e., waypoints) is the sequence analysis problem which seeks to find discrete labels for a given pattern. Although sequence analysis is not so pressing an issue in emerging UAV applications, it is well ensconced in molecular biology where the measurement of complex strings (life molecules such as proteins and nucleic acids) plays a critical role.

As with autonomous aerial vehicles, exciting advances in the form of miniaturized real-time portable DNA (deoxyribonucleic acid) sequencers has greatly improved the need for efficient and high-speed sequence analysis and its concomitant computational hardware. Presently, the embodiment of this advance is a palm-sized miniature DNA sequencer, the so-called **MinION** device [30], which has been commercially available for nearly three years. In the case of the MinION, a vast multitude of DNA strands are introduced into the device and converted into noisy electronic time-series signals; these measurements are essentially physical signatures related to the molecular make-up of the sensed DNA.

Among a long "pipeline" of analysis steps to be performed on such measurement sequences, the first, and arguably most intensive, is the so-called ***basecalling*** step which analyzes the time-series and converts it into the equivalent monomeric base sequence (from the discrete set of four possible nucleobase molecules, adenine(`A`), cytosine(`C`), guanine (`G`), thymine (`T`)) of the DNA under test [31]. Achieving this analysis at low-cost and in real-time promises substantial breakthroughs in point-of-care clinical analysis and molecular-level environmental analysis. A short description of DNA and DNA sequencing is given in Section 1.2.1.

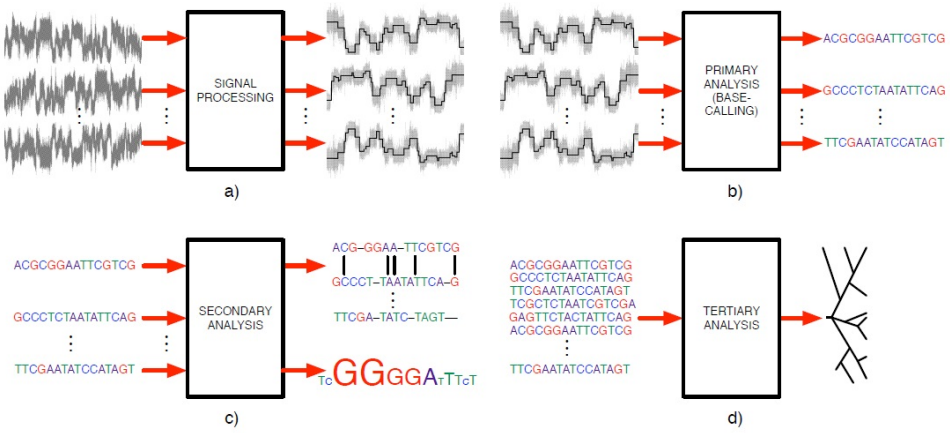


Figure 1.4: A simplified diagram of the main features in a MinION-based DNA sequencing pipeline.

1.2.1 DNA Sequencing

DNA is a molecule that ostensibly carries the genetic instructions used in the growth, development, functioning, and reproduction of all known living organisms and many viruses. Most DNA molecules consist of two biopolymer strands (**template** and **complement**) coiled around each other to form a double helix. Fig. 1.4 shows a simplified block diagram of a DNA sequencing pipeline as may be used in the context of a MinION device.

The first step (Fig. 1.4a), ***signal processing***, involves the extraction of representative signal features from the MinION's raw measurements. One prominent feature is a piecewise-linear representation of the underlying raw signal, also referred to as a "squiggle plot". The mean features of the squiggle plot are referred to as *events* and, as a result, the signal processing stage is sometimes referred to the *event detection* phase.

The next step (Fig. 1.4b), ***basecalling*** is the main component of the so-called ***primary analysis*** calculation. The job of the basecalling is to convert the squiggle plot (or even raw data in some cases) to their text (i.e. `A`, `C`, `G`, `T`) equivalent. As noted, only this step is considered in computational detail in this thesis.

Next, the ***secondary analysis*** phase (Fig. 1.4c) of the sequencing pipeline seeks to reconstruct all the different text files generated during basecalling into a complete genome.

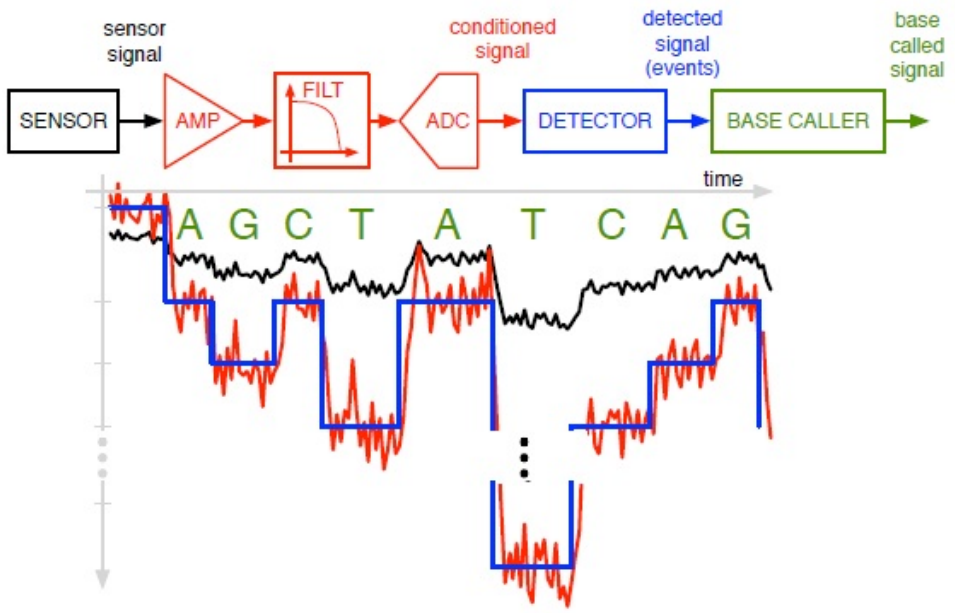


Figure 1.5: The signal flow in a MinION sequencing process form sensing to basecalling.

The final step, the ***tertiary analysis*** phase (Fig. 1.4d), seeks to make classifications and inferences of value to researchers in computational biology and medical clinicians.

Fig. 1.5 illustrates the MinION sequencing process in a little more of its physical detail. It demonstrates how a weak current signal (black) resulting from DNA sensing is first amplified, conditioned, and sampled (red) by analog circuitry. As noted above this is followed by event detection (blue) and basecalling (green).

The rate at which each DNA molecule is measured by a MinION device is about 250 bases per second (bp/s) [32]. In fact, an amazing amount of effort has gone into making DNA translocation through the pore as slow as 250 bp/s. If left to its own devices (i.e., no special protein-based movement control) the rate could be as high as 1M bp/s (1,000,000 bp/s).

In contrast, the computational workload for basecalling [33] requires about 1.5×10^6 arithmetical operations per second (OPS) for each squiggle plot event. A detailed calculation along with the basecalling algorithm itself is discussed in Chapter 7.

A MinION working at a channel rate of 250 bp/s and 250 operational channels at one

time (another critical feature of the device is its ability to measure many DNA samples in parallel), effectively requires 9.5×10^{10} OPS. At [34], an i7 4820K (based on 22-nm Ivy Bridge-E running overclocked at 3.9 GHz) is reported to achieve 12×10^9 DIPS (Dhrystone instructions-per-second) or 2.4^{10} OPS running an optimized version of a Dhrystone 2.1 test. Thus, per second, we are 3.96× behind in terms of calculating DNA sequencing. This OPS calculation is for one event. Usually, the total number of events is in the range of 5k but can easily exceed this value by 10× or 20×. If we consider 1000 events only, a traditional CPU-centred computational system is 3.96×10^3 times slower than the rate of raw signal production by the MinION device.

Speeding up the calculation rate is essential for real-time DNA sequencing in case of urgent care diagnoses. Using a computing cloud option such as the c3.xlarge facilities provided by Amazon Web Services (AWS) imposes a cost of $0.21/hour [35]. For a human genome consisting of 3×10^9 bases the cost of such a service is $55 based on the calculations above. Depending on the accuracy, one may need to sequence a genome repeatedly to attain "coverage" of say 30× resulting in a cost of $55 \times 30 = $1,650. A not insubstantial price for a continuous service. Perhaps even more are field scenarios where only expensive satellite or cellular data services may be available if at all. In such cases the costs incurred in terms of raw data tonnage from sensor to the cloud can easily eclipse the remote computing charge.

Considering the above limitations, a local computing accelerator for the basecalling functions becomes highly desirable. Achieving such in terms of a commodity GPU provides an extra measure of cost savings to the potential user. In this thesis, a pipeline system has been implemented where CPU threads are also used in conjunction with a GPU solution, this facilitates rapid operation over a multitude of DNA squiggle results where the first CPU thread reads the input (squiggle) file which includes the digital input data and does the necessary pre-processing calculations. The second CPU thread starts working with the processed data prepared by the first thread and calls the GPU for parallel computation where GPU threads operate in parallel to accelerate the central basecalling problem.

The basecalling problem is solved using the Viterbi algorithm instead of travelling salesman problem which is used for path planning. Generally speaking, the Viterbi algorithm provides an exact solution when the number of possible states is limited [36]. And for a higher number of states, Viterbi beam serach [37] is used which generates an approximate solution. In my basecalling problem, the number of states was manageable to use a Viterbi algorithm with the number of processors available in GPU. If I would have used TSP, I had to use heuristic algorithms that would provide me with a near-optimal solution.

1.3 Thesis Contribution

This thesis has three main contributions. The contributions are listed below.

Contribution 1: The first contribution is that the thesis solves the larger TSP on the GPU which has smaller on-chip memory. A large problem is subdivided into smaller problems so that the smaller problems can fit into the limited memory of the GPU. Then the smaller problems are solved inside the GPU. In general, a search for the best sequence of waypoints may invoke two opposite strategies: exact search and approximate search. Using the exact search, all possible sequences or combinations are checked to find the best result. As the number of points grows this approach quickly becomes intractable. In such situations, an approximate search is used which may find a near-optimal result. In these approximate searches, instead of checking all possible combinations, a limited number of combinations are tested.

Existing computational systems are not capable of implementing an exact search for any problem of practical interest. But after dividing the waypoints into small groups, it becomes possible to do an exact search for small groups of waypoints. This approach generates the best path through a limited number of waypoints and leads to producing a better final path at the end. A 4.82× speed-up compared to the CPU-only case is achieved.

Contribution 2: The second contribution of this research is generating a path between

two waypoints by solving the PRM more quickly. In particular, two steps (sample generation and k-nearest neighbour search) which take $\approx 99\%$ of the total execution time on a typical CPU-only implementation are parallelized. A $164\times$ speed-up compared to the CPU-only approach is achieved.

Contribution 3: The last contribution is accelerating the execution time for the DNA basecaller. The implemented basecaller is capable of generating 7283 bp/sec whereas the current serial basecaller developed by [38] can generate only 1080 bp/sec. The result is achieved by using CPU threads and GPU threads simultaneously.

1.4 Thesis Organization

The thesis is organized into eight different chapters. A brief literature survey is included in Chapter 2. A detailed description of GPU structure is been presented in Chapter 3. Chapter 4 discusses the total UAV mission strategy, the physical system configuration, I/O and the proposed global planner. Chapter 5 presents the UAV sequence generator implemented in this thesis. Also present therein is the customized clustering algorithm required to address the GPU's limited memory space. Chapter 6 presents the implementation of an accelerated UAV probabilistic roadmap on the GPU. Chapter 7 presents the accelerated DNA basecaller. The thesis concludes in Chapter 8.

Chapter 2

Literature Review

This chapter presents a brief description of the prior work done on key topics related to this thesis. First, the concepts of parallel execution in general, and parallel execution on the GPU in particular, are discussed. Then, this chapter presents the prior work related to sequence calculation using a parallel architecture, the construction of wireless sensor networks (WSNs) using unmanned aerial vehicles (UAVs), path planning for UAVs, and finally DNA sequencing.

2.1 Parallel Computation and Graphics Processing Unit

A brief overview of hardware for parallel computation and, in particular, the GPU, is now given.

In general, computing speed is often gauged by the number of clock pulses needed to complete an instruction, the so-called "cycles-per instruction" (CPI). Although a variety of techniques are used to achieve instruction-level and data-level parallelism, and hence improve CPI, until the turn of the century the most popular means among commodity microprocessors of improving computing speed was to increase the clock frequency. Due to heat dissipation and energy consumption issues, however, this strategy eventually became untenable; the number of tasks that could be processed in each clock period within a practical

power budget stopped improving at its previous rate. Processor developers responded to this circumstance by designing units with multiple processors or *cores* per die and thus offered the possibility of executing instructions in parallel; IBM's POWER4, released in 2002, is the first commercial processor to explore this [39].

Today, two different families of processors are marketed: *multicore* and *many core*. In a multicore approach, a few cores (typically two to ten at present) are integrated into a single microprocessor chip with the intention of speeding up the execution of the programs traditionally run on commodity machines including personal computers. In a many-core approach, several hundred cores (each with a limited computational ability and lower power needs) are oriented in such a way that maximizes the throughput of traditionally parallel problems. The GPU falls into the many-core category and has been adopted as the parallel platform of focus in this thesis.

Many core machines are used for parallel computations that distribute the workload among multiple processors working in parallel. Supercomputers [40], warehouse-scale computers, and GPUs are examples of many-core devices. A warehouse-scale computer is a cluster comprised of tens of thousands of servers connected together by a local area network to act as a single large computer. Such warehouse systems typically consist of 50,000 to 100,000 servers and cost on the order of USD 150 million to set-up [41]. A user can buy a subscription for single or multiple computers depending on the user's needs.

Supercomputers play an important role in the field of computational science, and are used for a wide range of computationally intensive tasks in various fields, including quantum mechanics, weather forecasting [42], climate research [43], and molecular modeling [44]. The most powerful supercomputer in recent days, the **Sunway TaihuLight**, contains more than ten million CPU cores and has been benchmarked with a performance rate of 93-petaflops [45]. The Sunway TaihuLight is also the most energy efficient machine at 6051-megaflops/watt and costs USD 273 million to make.

Many applications have adopted the GPU (energy efficiency for Kepler typed GPU is re-

ported in the range of $12 \sim 18$ gigaflops/watt [46]) for their computing needs including: image processing for computer vision [47], medical imaging [48, 49], environment mapping [50], movement tracking [51] etc. More specifically, the image processing problem, a dominant GPU application, includes blurring, edge detection, filtering, decoding, and encoding. Apart from image processing, the GPU is also used in computational biology for DNA analysis, sequencing [52], and alignment [20]. The GPU is also used in fluid mechanics for solving sparse matrices [53] as often encountered in weather prediction problems [54].

Over the last few years, the GPU has been the subject of intense interest from machine learning researchers, especially following the emergence of the deep learning paradigm. Significant performance improvement over standard CPU architectures has been recorded by using the GPU in this context especially in the training of convolutional neural networks for the GPU's staple application: image processing [55, 56, 57].

2.2 Parallel Sequence Calculation

Researches have already been used a parallel architecture like GPU for calculating sequences. These sequence calculations include, but not limited to, task sequencing [58], path planner for autonomous vehicles [17, 18], DNA sequencing [19, 20] etc. Travelling salesman problem is commonly used to define sequencing problems. Many algorithms, i.e., ant colony, min-max, genetics algorithm etc, are used to solve the TSP. More details discussion on TSP, used for sequence calculation are included in section 2.4.1.

The Viterbi algorithm is another known algorithm of solving sequential problems which is used for DNA sequencing in this thesis that provides an optimal solution. In general, the Viterbi algorithm is used for the decoding problem (defines as, given a sequence of symbols or observations and a trained model, what is the most likely sequence of states that produced the sequence). In the following paragraph, I will discuss a few prior research works that use GPU to solve the Viterbi algorithm in different research areas.

The Viterbi algorithm is widely used in the communication system, i.e., Li et al. [59] have used Viterbi decoding algorithm (VDA) in WLAN, WiMAX, or 3G communications where they have analyzed the parallelism of the Viterbi algorithm. Li et al. [59] have proposed a parallel version of the algorithm executable on the multi-core CPU, graphics processing unit (GPU), and field programmable gate array (FPGA). For more examples, the Viterbi algorithm is used in sparse communication channels [60], for decoding in software defined radio using GPUs [61]. The parallelly implemented Viterbi algorithm is also used to recognize speech [62] as well as to detect the speech of two-person from a single recording [63]. DNA alignments [64], liver image segmentation [65] are the few examples of the parallelly implemented Viterbi algorithm in biomedical technology.

The above paragraphs give a general idea of sequence calculation using a parallel architecture like GPU to the readers. From now, further discussion will be limited to the thesis topics only which are building a wireless sensor network using UAV and DNA sequencing.

2.3 Wireless Sensor Network Construction using Unmanned Aerial Vehicle

UAV-enabled (and GPU assisted) automated wireless sensor network (WSN) construction is a main application outlet of this thesis.

In this study, I have assumed that the WSN consists of a number of spatially distributed sensor/communication nodes whose physical location has been pre-determined according to some objectives/constraints assumed to be outside the scope of this thesis. Thus, the main objective of this thesis aspect deals with the computational challenges of the ensuing stage of such WSN realization: the automated deployment of the nodes at their specified locations via UAV. Since this objective address means of high-level control (i.e. planning) in UAVs, this section first outlines the main characteristics of a typical UAV's control system followed by a discussion of existing examples of UAV use for WSN construction.

2.3.1 Unmanned Aerial Vehicle: A Robotic Aerial Vehicle

Depending on the applications and environment, either fixed wing or rotary wing aerial vehicles are used. The most common example of fixed-wing UAVs are drones which are often used for military purposes. The velocity of these vehicles can be as fast as 16,700 miles/hour at altitudes of thousands of meters [66]. For low flying applications, it is preferable to use rotary wing aerial vehicles like the classic single-rotor helicopter. In such applications, the vehicle's richer maneuverability frees it to operate very close to the ground ostensibly improving its ability to reliably seed WSN from the air. Such an approach naturally requires avoiding collisions with a broad range of obstacles: trees, buildings, wires etc. For such low flying applications, small obstacles might not be visible from afar and it may happen that the vehicle cannot find an alternate path to avoid the obstacles instantly. In that case, the advantage of the rotary wing aerial vehicle is that the UAV can come to a stop position rather than hitting the obstacle [67]. Other advantages are easy landing and take off.

An exciting civilian example of rotary wing UAVs is the **Amazon Prime Air**, this drone, created by the retailer Amazon [68], has been in development over the last few years for the purpose of automated aerial parcel delivery. Similarly, one of China's biggest retailers is building a delivery drone that can carry 2000 pounds of cargo [69]. Most of the small rotary winged UAVs that have emerged in recent years are equipped with an autopilot and remote control. The number of fully autonomous UAVs remain quite limited however and although researchers have proposed multiple applications for these small UAVs, most of the work on them remains confined in research laboratories.

2.3.2 Control System of Unmanned Aerial Vehicle

In general, each robot has a software control system which initiates multi-stage decisions as part of the machine's plan execution. According to Kortenkamp and Simmons [70], the design of such a robot software control system refers to two different concepts: *architectural structure* and *architectural style*.

The robotic architectural structure refers to how the system is subdivided into multiple subsystems. These are typically categorized into four groups [70]: *deliberative model* [71] (the robot thinks first, then takes an action), *reactive model* [72] (reacts in real-time according to pre-programmed controls), *hybrid model* [73] (combination of deliberative and reactive), and *behaviour-based control model* (a set of modules are designed with specific behaviours).

The architectural style refers to how each of the subsystems communicate with each other. Two of the most common styles are *client-server* and *publish-subscribe* methods. In the client-server system, the request for any result is paired with the server and the client waits until the result has been received. In the publish-subscribe style, one subsystem completes a goal and broadcasts the commensurate result. Only those subsystems that require the result assert it. The subsystems are often referred to as *modules* when they are built on top of one another. In this case, the result of one module is used by the next module. Each module is designed to implement a specific goal. Such modular design reduces the complexity of the system by lowering the need for communication through the narrow bandwidth of the system. For the implementations considered in this thesis, the software system is designed on a modular basis.

More generally, instead of a single robot, multiple robots can be used to achieve goals. In case of multiple robots, a network control system is required to manage the robots or to facilitate effective communication among them. In some experiments, the unmanned vehicles are controlled from a central unit [74, 75]. Alternatively, in other approaches, all vehicles have their own control system on board but the vehicles also have the option to communicate with other vehicles that are engaged in the same mission [76]. This thesis limits its scope to the single-UAV case.

The control system of a UAV is used for implementing a fixed goal like scanning an environmental disaster, constructing a network, reconnaissance, target acquisition, search and rescue, surveillance, environmental monitoring, mapping, and many other applications. For example, Quaritsch et al. [77] have built a disaster management application where UAVs

fly over the disaster area. The UAVs are equipped with cameras and sensors to collect video and images of the area. The videos and images are sent back to the ground and are used by the rescue team after completing some specific analysis. Again, if we consider an example of mapping, Scherer et al. [78] used a low-flying robot to map a river which requires distinguishing the river's edge from the embankment vegetation. Many other examples of UAV applications can be cited, but presently there is no work done for constructing a WSN from the UAV control perspective according to my knowledge.

2.3.3 Wireless Sensor Network Construction

The concept of deploying sensor nodes from a UAV is not new and a number of researchers have experimented with UAVs for just such a purpose [79, 80, 81]. Corke et al. [79, 80], used a wire coil with a radio controller to deploy nodes to the correct position on the ground from a UAV. After deploying the sensor nodes, the connectivity of the network was checked and additional nodes were deployed on demand.

In [79, 80], the UAV was equipped with GPS (global positioning system) for localization and connected with the ground station through a 2.4 GHz wireless Ethernet link. During the node deployment, instead of an online path planner, the UAV followed either a pre-programmed path or control instructions sent from the ground station [79, 80].

Ho et al. [82] used a UAV to collect data from the cluster heads in a WSN and addressed optimal robot navigation across a sequence of points by solving a version of the travelling salesman problem (TSP). Note that the TSP solves for the sequence of waypoints which will generate the shortest path that visits each of the waypoints in some set only once. Ho et al. [82] mentioned the desired ground velocity and altitude of the UAV, but did not mention how the flight path will be constructed.

In many such studies [79, 80, 81, 82], the authors assumed that the UAV could somehow carry-out a safe journey and did not focus on the flight control: how the obstacles were avoided, how the steering was controlled, how the paths to the destination were found etc.

Generally speaking, the main drawback of works considering UAV-deployed WSNs has been that only the accuracy of the network was considered while largely ignoring the issue of UAV control during the mission.

Again, the number of deployed nodes depends on the transmission range of the sensor nodes and on the desired network performance. For example, Corke et al. [79] deployed 50 nodes, each with a transmission range of 2.5-m, in a 22-m×10-m sized field. In contrast, Tian and Georganas [83] placed 100 nodes in a region whose area is 50 m×50 m. The node transmission range in [83] was 10 m. For larger area and sensor nodes with a shorter transmission range, the number of the sensor nodes can be as large as several thousand, a scale not discussed in [83, 79, 80, 81, 82]. Such a big wireless sensor network is typically used in the oil, gas or water pipeline monitoring system [84, 85]. Jawhar et al. [85] mentioned that in the United Arab Emirates, there is 2,580 km of gas pipelines, 300 km of liquid petroleum gas pipelines, 2,950 km of oil pipelines, and 156 km of refined products pipelines according to the year of 2006. Monitoring such a big infrastructure requires thousands of sensor nodes to cover the entire pipeline.

2.4 Path Planner for a UAV

The problem studied in this thesis, UAV-assisted WSN construction, has multiple waypoints denoting locales for sensor node deployment. Naturally, the UAV is therefore required to fly through all of the given waypoints, seeding a node at each point. For such flights, the UAV's path planner decides which waypoint the UAV should next go in its node deployment sequence and which flying route the UAV should follow in flying from one waypoint to the another.

A review of UAV path planners reported in the literature reveals that few research works concentrate on both the sequence generator (i.e. which waypoint the UAV should go to next), and the point-to-point path constructor together. For example, Requicha and Spitz [86] have

worked on the sequence planner and path construction for coordinate measuring machines (CMMs) where a probe is moved to multiple designated spots for measurement. This involves both path planning for collision avoidance and sequencing the points to minimize the length of the path. In particular, [86] solved the TSP for the sequence planning and another algorithm (the probability roadmap — PRM) for path construction. The entire planner took 12.96 sec for 30 waypoints and 24.48 sec for 100 waypoints using serial execution on a Sun ULTRA 1 machine. In [87], [88], and [89], the planner includes both sequencing and path construction but all of these examples are offline planners.

Most path planner research works, other than the small number of research studies focusing on both sequence generation and path construction, are focused either on computing a safe path between two waypoints or on finding an efficient sequence of waypoints, often called as task planner. In light of this partition, the following discussion on prior works related to UAV path planning, is divided into two sections: 2.4.1, and 2.4.2 which discuss the sequence generator, and the path construction between two waypoints, respectively.

2.4.1 Sequence Generator

A sequence generator computes an optimum sequence of waypoints to follow, depending on the problem objective. Example objectives in the context of UAV path planning over multiple waypoints include minimizing flight length, minimizing fuel cost etc. A rigorous means of addressing this challenge is in terms of the aforementioned ***travelling salesman problem*** (TSP).

The TSP is a very well-known topic with a rich history of research [28]. As mentioned above, classically, it concerns the problem of visiting each of a given set of points exactly once such that the length of the total route through the sequence is a minimum.

This is an NP-hard problem that, as with all such challenges, can be solved in two basic ways: an exact algorithm or an approximate (heuristic) method [28]. The former finds the exact solution at the expense of computation time which increases exponentially with the

number of waypoints [28]. Conversely, the heuristic approach solves the problem faster but can not guarantee the optimal solution although near-optimal results are possible.

Specific examples of strategies employed for exact algorithmic solutions include integer linear programming, branch and bound, breadth-first search etc. [90]. Heuristic examples include nearest neighbour search, genetic algorithm (GA), simulated annealing etc. [28]. Researchers have solved the TSP with different algorithms and executed the computation in either a serial or a parallel manner. Further discussions on the prior TSP work is divided into two parts: TSP with serial implementation, and TSP with parallel implementation.

TSP: Serial Implementation

Most of the research work related to the TSP-based navigation using serial implementations, assumes that the sequence of waypoints will be generated offline due to the relatively long computational time. These contributions often incorporate other scenario features in their calculations such as missile threats [91], radar network [92], flying angle etc.

To date, the best generic TSP solver (i.e. just the classic TSP with no additional feature considerations) is the Concorde TSP [93] that finds the optimal result for all of the instances in an example library, TSPLIB, using multiple heuristic algorithms including the cutting-plane method, the minimum spanning tree, the nearest neighbour, the branch and bound method. Specifically, the TSPLIB is a library of sample instances for the TSP [94] and the largest problem in the TSPLIB contains 85,900 nodes. The execution time for 1000 waypoints (from the TSPLIB file named dsj1000) is listed as 410.32 sec [93] on a 500-MHz Compaq XP1000 workstation. As a pure TSP solver, evaluated by the quality of its results and size of problems handled, Concorde is recognized as the best available tool, but it can't be used online due to its execution time.

Dorigo and Gambardella [95] used the ant colony algorithm to solve the TSP. In general, in the ant colony algorithm, a set of agents called ants cooperate to find a good sequence. In nature, ants are able to find good solutions to the shortest path problems between a food

source and their home colony via a pheromone-based (aromatic substances) communication scheme that they use in variable quantities to mark their trails. An ant's tendency to choose a specific path is positively correlated to the intensity of a found trail. The pheromone trail evaporates over time. i.e., it loses intensity if no more pheromone is laid down by other ants. If many ants choose a certain path and lay down pheromones, the intensity of the trails increases and thus this trail attracts more ants. Dorigo and Gambardella [95] also used 3-opt local search with the ant colony algorithm and were able to find the optimal solution for 318 waypoints in 537 sec and near optimal answer for problems larger than 318. Stutzle and Hoos [96] used a max-min ant system, an updated version of ant colony system, that also cannot guarantee to find an optimal solution like [95].

When multiple algorithms are used to solve a problem, the process is referred to a following a hybrid approach. The hybrid method has become very popular for solving the TSP. For example, Baraglia et al. [97] used the GA and Lin-Kernighan (LK) local search to solve a 1000-node TSP in an average simulation time of 25 min using C++. The authors [97] also claimed that their algorithm is capable of solving 13,509 nodes.

The main drawback of the serially executed TSP solvers [95, 96, 97] is the long execution time. None of them can find the sequence online. For faster sequence generation parallel computation architectures are explored in the next section.

TSP: Parallel Implementation

Due to the large computational workload, it is very common to solve the TSP using parallel execution. For handling parallel execution on modern CPU and GPU platforms, multithreading is the most convenient method to use. For example, Sahingoz and Ozalp [98] used 16 CPU threads to solve the TSP in a 3D environment and added threats of a radar network. The authors [98] used a genetic algorithm and achieved a computational time of more than 600 sec for a 963 waypoints problem.

References [99, 100, 101, 102, 103] etc. are examples of solving the TSP with parallel

GPU threads. O'Neil et al. [99] presented an implementation of the TSP using a GPU where they applied an iterative hill climbing (IHC) method and opt-2 search for hill climbing or, in other words, a 2-opt search was employed to find a better solution than the hill climbing algorithm itself. In general, in hill climbing methods, the result is updated when a better solution than the current state is found. The advantage of using IHC is that no additional memory storage is required as the current state is updated only if a better new solution is found. IHC is often able to find the local minima only. To resolve this drawback, 100,000 random start points are considered in [99]. Finally, [99] was able to find the optimal solution for 100 waypoints. This GPU solution runs 60 times faster than an x86 core and runs as fast as 32 CPUs with 8 cores/CPU.

The main limitation of O'Neil et al. [99] is that the algorithm can only solve a maximum of 110 waypoints as they store pre-calculated waypoint separation distances in the GPU's *shared memory*, a very low latency local storage. Their design and GPU shared memory size limitations cannot hold distances for problems with more than 110 waypoints.

Rocki and Suda [100] addressed the main limitation of [99]'s iterative hill climbing method. In particular, they managed to increase the limit of the number of waypoints by storing coordinates in shared memory and calculating the distances among the waypoints on-the-fly. As a result, [100] managed to solve a 6000 waypoints problem using the IHC method. However, at such scales, the approach was two times slower than the original serial implementation.

Rocki and Suda [101] also have worked on a parallel implementation of TSP using an iterative local search. This effort achieved 10 to 50 times faster execution time depending on the size of the problem. The best thing about the implementation [101] is that the algorithm can handle thousands of waypoints.

Ant colony algorithms have emerged as attractive means of tackling the parallel implementation of the TSP. For example, Fu et al. [102] used an ant colony algorithm for the parallel implementation of a TSP and achieved 30× speed for the problem with 1000

waypoints. The parallel implementation of a similar strategy, the so-called MAX-MIN Ant system algorithm, runs 32 times faster than a CPU for 400 waypoints [103].

Genetic algorithm (GA) are another family that historically has not demonstrated great success in implementing a larger GPU-based parallel TSP implementation. Fujimoto and Tsutsui [104] used a GA to solve the TSP and demonstrated a parallel implementation working 24.2× faster than the CPU implementation. But their approach is limited to problems with a maximum of 512 waypoints. The work done by Chen and his co-workers [105] also supports a very limited number of waypoints using a GA; their parallel execution is less than two times faster than the CPU execution. Li et al. [106] implemented an immune algorithm using a genetic algorithm framework and got only a 11.5 times faster result for 225 waypoints. My research study is able to solve a larger problem with 4096 waypoints which distinguish the research study with the others.

2.4.2 Path Explorer

In general, a UAV's path explorer concentrates on the problem of path construction between two waypoints. Among the many algorithms focused on path construction, two types of general approaches are *deterministic* and *stochastic* [107].

In the deterministic algorithms, the uncertainty due to robot actuators and sensors is not considered while stochastic approaches take these into account. In robotics, component information uncertainties can be due to unmodeled vehicle dynamics, unknown environments, environmental disturbances, uncertainty about the pose information etc. This research work follows the deterministic approach and further discussion is limited to this outlook.

Deterministic methods can be further divided into three main approaches [108]: the skeleton approach, cell decomposition, and potential fields.

In the skeleton approach, a graph is generated around the start and the destination locations. Then a graph search algorithm is applied to find the final path. This approach is also known as a *roadmap* or *highway* approach [108].

In the cell decomposition method, the map is represented by cells; each cell is assigned with a weight depending on the path planner [107, 109]. For finding the desired path, the weight of each cell is updated depending on neighbouring cells which makes the process serialized.

The potential field concept assigns repulsive forces (relative to the UAV) to obstacles and attractive forces to the target, respectively [110]. Potential field planners are quite fast but not guaranteed to find a path [111].

Much research work has been done on path construction using the three approaches listed above. For the sake of simplicity, further discussions are divided in two Sections 2.4.2 and 2.4.2 depending on how the programs are executed, i.e., serial execution or parallel execution.

Path Explorer: Serial Implementation

To date, most of the prior research work on UAV path planning falls into this computational style, where a safe path is constructed between start location to target location and algorithms are executed in a serial manner in the CPU. Many of the studies are related to avoiding radar networks [91] or obstacles [112], and generating smooth paths [113]. Computational time is usually not listed in these reports. For example, Zhang et al. [114] have used one of the most famous algorithms for path planning, the ant colony optimization, for generating a path from start to destination. Mittal and Deb [113] have worked on generating paths offline using a hybrid algorithm that includes the genetic algorithm, B-Spline curve, and clustering to find the safest path. But none of them [114, 113] have reported the runtime.

Where the computational time is mentioned, the runtime is typically very long, relegating the techniques to offline scenarios. For example, Pehlivanoglu [115] used a vibrational genetic algorithm enhanced with a Voronoi diagram for the path construction between two waypoints which took approximately 27 sec. The Voronoi diagram divides the space into several regions

depending on the obstacle locations [116]. Kim et al. [112] used reinforcement learning (the Q learning algorithm) to endow the UAV with obstacle avoidance. But learning takes 26 sec for a 20×20 size map with 52 obstacles. Using such planners requires additional flights at the beginning to train the planner.

There are few UAV-focused path planners which work in real-time. Such planners typically contain two versions of the planner: an offline planner and an online planner. The offline planner is executed before the vehicle takes off and the online planner in executed during flight for making corrections in case of unexpected moving obstacles.

For example, Sujit and Beard [117] worked on multiple UAVs where each UAV had its own start and end location and these UAVs which pass through the same environment. Each UAV generates a path for itself in 60 sec, initially offline. Then, depending on the pop-up and moving obstacles, the UAVs make corrections on their path which takes 5 sec to 6 sec online. Similarly, Kothari et al. [118], used an offline and online version of planner using RRT (rapidly-exploring random tree) algorithm. Qu et al. [119] worked on two types of path planners: real-time suboptimal path planner (with A* and a geometry smoothing algorithm), and an offline optimal path planner (employing GA and potential field methods).

Path Explorer: Parallel Implementation

Research works on path explorers using parallel implementation are reviewed in this section. The skeleton approach is more popular for such realizations on many-core GPUs. In the skeleton approach, multiple sample nodes (in the range of hundreds to thousands, depending on the space size and problem formulation) are used and the same set of codes are executed for all of the sample nodes which can exploit a GPU's high parallelism. Conversely, in the cell decomposition method, individual cells depend on neighbouring cells which limits opportunities for straightforward parallelization.

Two common skeleton-based methods, often called sample based algorithms, are probabilistic roadmaps (PRMs) and rapidly-exploring random trees (RRTs). Manocha and his

team worked rigorously on PRM with 6-DOF (degree of freedom). Pan et al. [111] presented a novel parallel algorithm for real-time motion planning of high DOF robots that exploits the computational capability of a commodity GPU. In general, the problem complexity grows exponentially with increases in the DOF and [111] was able to handle such computational workload.

PRM has two phases: a construction phase and a query phase. In [111], efficient parallel strategies for the construction phase are described that include sample generation, collision detection, connecting nearby samples, and local planning. The query phase is also performed in parallel based on graph search. In order to accelerate the overall performance, [111] also described new hierarchy-based collision detection algorithms. They highlight its performance on multiple benchmarks on a commodity PC with an NVIDIA GTX 285 GPU and observe a 10-80 times performance improvement over CPU-based implementations.

Collision detection and k-nearest neighbour search are the most computationally expensive phases in a sample based planner. Pan et al. [120] implemented a novel algorithm for k-nearest neighbour search with linear space and time complexity and exploits the multiple cores and data parallelism effectively in GPU. The formulation is based on locality sensitive hashing (LSH) and cuckoo hashing techniques, which can compute approximate k-nearest neighbours in higher dimensions. The implemented algorithm improves the performance of the overall planner by 20 to 40 times for CPU based planners and up to 2 times for GPU-based planners.

Pan and Manocha [121] presented parallel algorithms to accelerate collision queries for sample-based motion planning like PRM and RRT. In order to take advantage of many-core GPUs, they present a clustering scheme to appropriately allocate collision queries to different cores, and collision-packet traversal to perform efficient collision queries, from [120], on multiple configurations simultaneously. The implemented code can perform 500,000 collision queries per second, which is 10 times faster than prior GPU-based techniques. Moreover, they can compute collision-free paths for rigid and articulated models in less than 100 ms

for many benchmarks, almost 50 to 100 times faster than CPU-based PRM planners [121].

Yoon et al. [122] presents GPU-based collision detection method that accelerates collision queries for sampling-based motion planning where a robot used 4-DOF robotic arm with each joint having 0 to 360° mobility. For the collision detection, the rectangular shaped oriented bounding box (OBB) is used to represent the robot as well as the obstacles in 3D space. The robot could be a combination of multiple OBBs depending on its shape. Then the collision between the obstacle OBB and the robot OBB is checked. They observed a ten-fold speed-up in the local planner compared to using a CPU.

2.5 DNA Sequencing

The DNA sequencing process is getting faster and more accurate with new innovations in bioscience. Fred Sanger's DNA sequencing technique [123], a means employing randomly interrupted enzymatic extension, was a profound contribution to genomics science. In short, this technique employs some chemical reactions and the outcome of this reaction is a set of dsDNA (double-stranded DNA) of various lengths whose terminating nucleotide can be identified and thus sequenced by resulting molecule size. Sanger's contribution accelerated the process of DNA sequencing from rates of roughly 10 base pairs (bp) per year to about 100 bp/day [124].

Later, in the mid-80s and in the late 90's, two outstanding improvements in the field of sequencing was the adoption of fluorescent labelling [125], and capillary technology [126], respectively. In the fluorescent labelling process, nucleotides are bonded to fluorescent molecules and emitted colours help the automation of the sequencing process. The introduction of capillary transport in place of gel electrophoresis increased sequencing throughput to 360 kilo base pairs (kbp) per day and, through refinements, is today capable of processing roughly 1-2 million bp/day [127].

A substantial advance in DNA sequencing methods started in 2005 with the emergence of

a variety of so-called next-generation-sequencing (NGS) techniques. NGS machines achieve sequencing by adhering to a ssDNA (single-stranded DNA) multiple detectable bases in a controlled and sequential process (and thus synthesize a double-stranded chain), a method generally referred to as sequencing-by-synthesis. This approach makes it possible to carry out sequencing in-situ, without the need for physical transport of the analyte as needed in the Sanger method. As a result, NGS has opened and exploited the possibility of realizing much more complex and compact sequencing platforms. This, in turn, has lead to the construction of multi-channel systems and thus achieved high-throughput operation with institutional NGS machines capable of exceeding 1 Tbp/day and even some desktop NGS machines operating over 100 Mbp/hour [128]. Important examples of NGS technologies employing the sequencing-by-synthesis strategy include the Solexa/Illumina bridge amplification method which dominates the field today [129] as well as Roche's pyrosequencing method [130].

To achieve sequencing with sufficient fidelity, multiple copies (roughly one million) of analyte molecules must be made; the reason for this is the limited sensitivity of detection methods in NGS which also introduces noise. Another limitation of NGS systems is their relatively short read-length, that is, the number of nucleotides (nt) that can be sequenced using NGS processes in a single run to a given accuracy. The availability of only short reads naturally leads to difficulties in the sequence extraction of significant long-range patterns (e.g. genes) from the DNA under investigation [127].

Arguably, the next major step in the evolution of sequencing machines into their third generation centres around the creation of single-molecule detection technologies. Such a development promises to address two critical complications in NGS machines: chemical amplification and short read-lengths. Nanopore-based sequencers are one embodiment of this vision.

The nanopore-based sequencing method breaks both of the key features (synthesis and optics) established by its predecessors: i) its signalling process works directly on the DNA and

the chemical complexities this entails (e.g. nucleotide modification, amplification, nucleotide addition, wash, fluor removal, blockers, etc.); ii) it associates structural DNA features with electronic charge rather than photonic wavelength making an interface to production-grade microelectronic technologies (i.e. complementary metal oxide semiconductor — CMOS) much more amenable.

Nanopore-based sequencers (NBS) are a very recent addition to the sequencing tool marketplace. Most prominently, a nanopore-based molecular sensing device plus accompanying sequence analysis tool chain have been made available by Oxford Nanopore Technologies Inc. (ONT) since the spring of 2014 [131, 132]. These systems currently achieve sequencing speeds on the order of 500-nt/s, orders of magnitude faster than the methods outlined above, with the potentially for vastly higher per-sensor speeds given a quality interface to a sufficient signal processing technology.

The MinION currently relies on a cloud computing platform, Metrichor (metrichor.com), for translating locally generated sequencing data into basecalls. David et al. [38] build the first offline open-source basecaller for the MinION and made the basecaller open source which named as Nanocall. Nanocall can generate about 2500 Kbp of sequence per core hour or 694 bp/sec per core using the hidden Markov model and Viterbi algorithm.

To accelerate the DNA sequencing process, in terms of computation, GPU has been used by researchers [133, 134, 135, 64]. Boža et al. [133] solved the basecaller for MinION based on nueral network. Researchers [135, 64] used GPU for DNA alignment which is a part of secondary analysis discussed in Chapter 1. [134] is an example of solving basecalling with GPU using deep learning method. But my reserach work on basecalling is based on the hidden Markov model and Viterbi algorithm.

Chapter 3

General Purpose Graphics Processing Unit

The general purpose graphics processing unit (GPU) is a promising computing architecture for parallel programming. The GPU's compute hierarchy, memory hierarchy, hardware architecture, and specifications are discussed in this chapter. Energy consumption and performance rate for selected GPUs are listed in the GPU specifications section.

3.1 Overview of GPU

A promising computing architecture for parallel programming is the GPU [6]. Compared to the CPU, the GPU has a thousand times more cores within a similar chip area, a simpler hardware control, and more power efficient operation (i.e., the power requirement as a function of throughput) [136]. Another primary difference between the CPU and the GPU is that the CPU is typically designed to optimize the latency (reduce execution time for a task) while the GPU is designed to maximize the throughput (the number of tasks completed within a time frame). Thus the GPU is widely used for parallel processing a computing technique known for boosting throughput via distribution of operations over many simultaneously operating units.

The CUDA programming framework, developed by NVIDIA, allows a programmer to write code for both processors (CPU and GPU) in a single program [137]. A picture describing a typical CPU/GPU workstation arrangement is shown in Fig. 3.1. Typically, the GPU essentially functions as a coprocessor for the CPU. The CPU is in charge of initiating all the instructions. In general, the CPU is referred to as the *host* and the GPU is referred to as the *device* [137]. Both the CPU and the GPU have their own dedicated memories: the traditional DRAM (Dynamic Random Access Memory) for the CPU and the GDDR5 SGRAM (Graphics Double Data Rate Five Synchronous Graphics Random Access Memory) [138] for the GPU. The GPU also has on-chip shared memories denoted as SMEM (Shared Memory) in Fig. 3.1.

Typically, the CPU and the GPU communicate with each other through the PCIe (Peripheral Component Interconnect Express) bus. The CUDA framework through which GPU resources may be accessed for general program execution supports numerous programming languages. In this thesis, C/C++ has been used with an extension for GPU programming. In such programs, a part of the code is executed in the CPU and rest of the code is executed in the GPU. The code that is executed in the CPU is called the ***host code*** and the code meant for GPU execution is known as the ***device code*** [137]. A compiler from NVIDIA, ***nvcc***, is responsible for the separation of both the host code and device code from within a single program.

Efficient programming for the GPU requires an obvious idea of its architecture and the optimization mechanisms through which it may be best utilized for a problem. Although specific parameter settings may change, well-designed program concepts should scale very well between GPU architectures ranging from mobile devices to large distributed systems [137]. There are three main types of GPU architectures: Fermi, Tesla, and Kepler. Depending on the GPU specification, i.e., the number of CUDA cores, memory size, etc., the performance of the implementation will vary across these only because of access to more processing and memory. In this thesis, the Kepler GPU architecture, as present in the NVIDIA GeForce

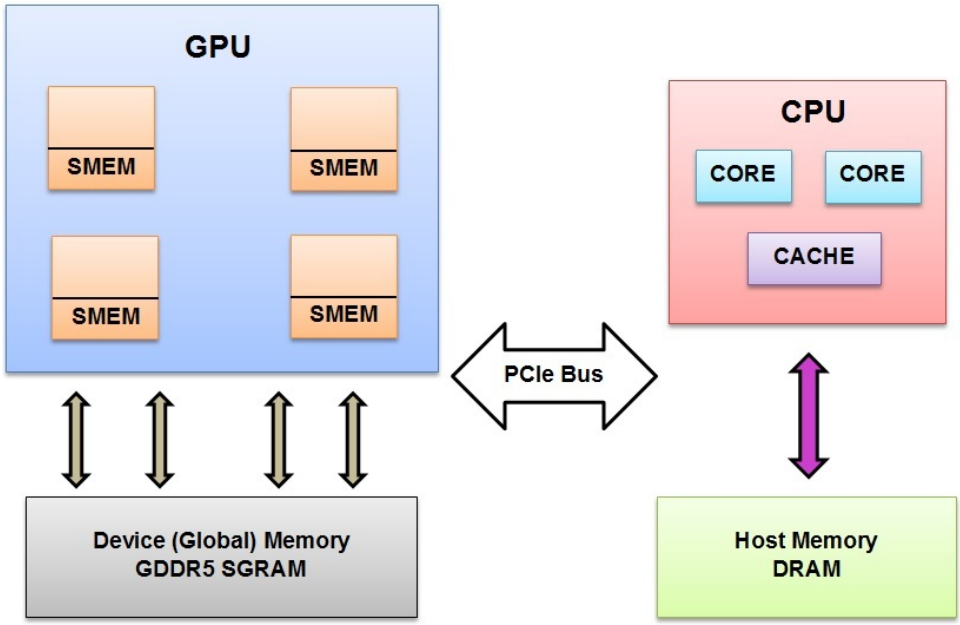


Figure 3.1: CUDA memory management model.

GTX 680 GPU model, was used for generating the experimental results. But an approximate performance update will be estimated for high-end GPUs: Tesla K20 and Tesla K80.

3.2 GPU Architecture

This section introduces and outlines the basic GPU features essential for parallel programming. Specifically, the organizational strategies used to arrange the compute and memory portions of these devices are discussed.

3.2.1 Compute Hierarchy: Thread, Block, Grid, Kernel, Warp

Thread

Command execution in terms of **threads** is a long established practice in computing platforms. It refers to the partitioning of code into distinct "sections" (i.e., threads). Each thread simply encapsulates a piece of serial code. However, within this paradigm, the application is free to decide which thread is best executed at any one time and thus performs intermittent

context switching between threads sharing a processor. For example, if one thread enters a blocking state, the time may be right for the application to shift to another thread of serial execution. Being part of one consistent application allows threads to share the same address space in main memory, therefore, allowing for an efficient transition between threads (relative to switching between individual processes) [6].

In CPUs, the treatment of individual threads is usually quite sophisticated including the concept of simultaneous multithreading (e.g., hyperthreading) over pipelined superscalar architectures with out-of-order execution capability. The result is a very sophisticated control logic scheme intended to extract as much instruction-level parallelism (ILP) as possible from serial thread code. In a commodity desktop multicore CPU, perhaps 16 thread instructions may be executing simultaneously under optimum conditions.

In contrast, even relatively modest GPUs like the GTX 680 can support 1,536 simultaneous threads with the ability to partition an algorithm into many 1000s of non-concurrent threads [138]. This is achieved rather directly, with 1000s of cores, so-called *streaming processors* (SPs), placed adjacent to each other on a single-chip die. Each SP is much simpler than a modern CPU's core and hence is inferior in its ability to achieve high ILP from any thread assigned to it. However, this architecture, via frameworks such as NVIDIA's CUDA, does give the programmer the ability to achieve significant processor-level parallelism (PLP).

Block

For NVIDIA GPU systems, the programmer has the option of lumping threads into larger units called *blocks*. Threads aggregated to a given block have the possibility of at least crudely synchronizing their actions (e.g., via so-called *barrier synchronization* mechanisms). Each GPU allows a limited number of threads to be assigned to any one block, presently 1024. The threads in a block are indexed according to a maximum of three-dimensional array thus giving the programmer more flexibility in assigning parallel threads over multi-dimensional arrays, particularly those directly describing physical phenomena, a relic of GPU origins in

graphics applications. Physically, each block is executed by a *streaming multiprocessor* (SM) unit. SMs are collections of $\sim$ 200 SPs with dedicated control logic and **shared memory** per SM. SM's are further detailed in discussions below.

Grid

A **grid** consists of blocks [137]. As with blocks, a grid can be organized according to a three-dimensional array. Like blocks, a grid also has limits on the number of blocks it can contain. Code run on the host launches **kernels** which is technical jargon for functions executed on the GPU. These kernels are executed by many GPU threads in parallel that are organized, at the highest level, in grids. That is, a kernel is assigned to a grid with the grid being a two-stage hierarchy comprised of the aforementioned blocks and threads.

Warp

During execution, threads in a block are divided into smaller groups called **warps**. Threads in the same warp are executed at the same time. They also read and write data at the same time. The standard warp contains 32 threads. If the total number of threads in a block is not divisible by 32, a warp will contain less than 32 threads. For example, if there are 34 threads (numbered as #0 - #33) in a block, then the first warp will contain 32 threads (from #0 to #31) and the second warp will contain two threads (#32 and #33).

3.2.2 Memory Hierarchy

As mentioned earlier, the program is initiated from the CPU. Thus, at the start of execution, the program beings in the CPU and data is present in the CPU's main memory. When the CPU program reaches its device code execution section, it is required to send data to the device so that the GPU can properly execute the device code. At this point, a memory management issue arises. There are several types of memory present in the GPU which are shown in Fig. 3.2. Fig. 3.2 also shows the direction of data flow.

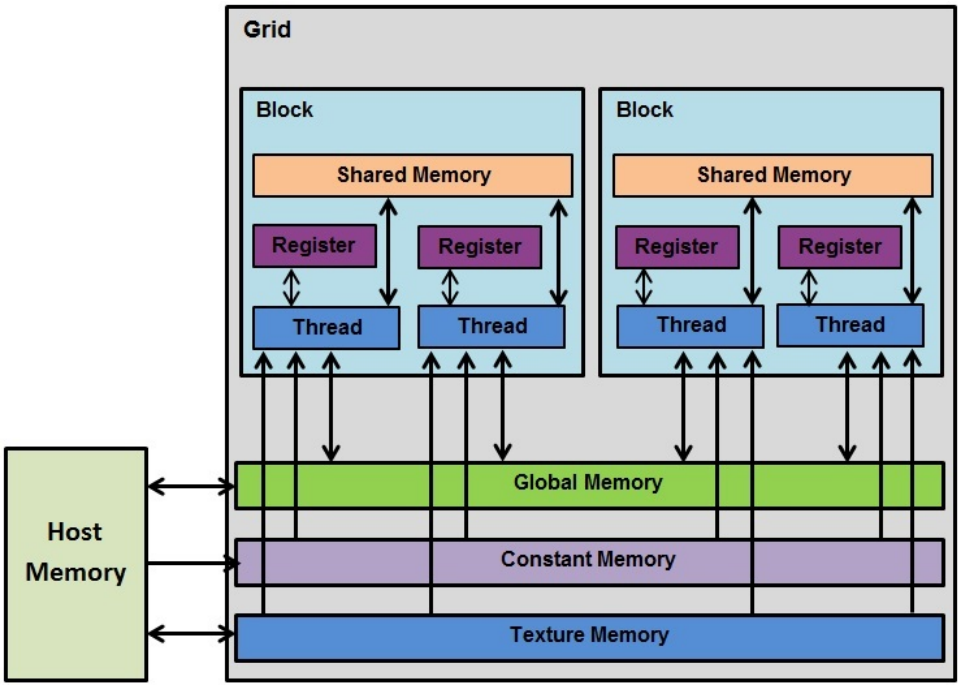


Figure 3.2: CUDA memory management model and data flow direction.

The main types of memory in the GPU and their specifications are given in Table 3.1. The largest memory available to GPUs is the **global memory**. This memory, which is on the order of 10 GB on GPU modules (see Table 3.2 for specific examples), is implemented in memory chips sharing the GPU's motherboard with the GPU computing core. Communication between the CPU's main memory and the GPU's global memory is typically facilitated by a PCIe link. For a typical 16-lane PCIe connection with eight lanes dedicated to each direction, duplex communications on the order of 10 GB/s between the system and global memories may be facilitated in each direction. Although such transfer rates may seem impressive, compared to the GPU's on-board memory's bandwidth, they are quite slow. For example, the GTX 680's GDDR5 SGRAM (*double data rate type five synchronous graphics random-access memory*) operates with an effective memory clock of 6008 MHz over a 256-bit bus thus achieving a 192.2-GB/s memory bandwidth between the GPU and its global memory [138]. As a result, data read/write between CPU host system memory and GPU global memory is considered slow.

Constant memory is a small (64 KB) subset of the GPU's main GDDR5 SGRAM

intended to store variables whose values do not change over the course of a kernel's execution. As a result, these values are preferentially cached on-chip by the GPU and thus exhibit faster access rates than standard global memory contents. However, threads can only *read* data from constant memory. Constant memory is optimized for broadcasting [137] which means reading data from the same address of the constant memory by threads from the same warp. This is as fast as reading from an on-chip register as long as all threads read from the same address. Accesses to different addresses by threads of the same warp are serialized, so cost scales linearly with the number of different addresses read by all threads within a warp.

Texture memory is also a read-only cached memory and can be accessed by all the threads present in different blocks. Texture memory is optimized for 2D spatial access patterns.

Shared memory is a fast, SRAM-based, on-chip memory for threads in the same block. Each thread has its own on-chip registers which have the fastest accessibility among all the memories. The difference between the shared memory and the register memory is that the shared memory can be accessed by all the threads in the same block and the register memory is for a single thread itself which can be accessed by that single thread only. If the threads exceed the memory limit of these registers, they start using **local memory** which is an off-chip memory. The local memory is part of global device memory, that is, it is accessible only by the thread that declares it. As the access speed to the local memory is not as fast as for the registers, the program gets slower.

Besides, sending data from host memory to global memory, there is another option that does not send data to the device which is known as **page locked memory**. Using this feature, the time required for transferring data from host to device and device to host can be saved as the data resides in the host memory. In this process, a pointer (pointing to the host memory where the data is saved) is sent to the device. This approach is helpful when a large amount of data needs to be transferred to and received from the device. But multiple accesses of a single data from the device ruins the time saved by not transferring data.

Memory type	Read/Write options	On/Off chip	Cached	Access speed	Accessibility
Global	Read and Write	Off	Cached	High latency	All threads and host
Constant	Read only	Off	Cached	Low latency	All threads and host
Texture	Read only	Off	Cached	High latency	All threads and host
Shared	Read and Write	On	N/A	Low latency, very high bandwidth per multiprocessor	Threads in the same blocks only
Register	Read and Write	On	N/A	Low latency	Individual threads only
Local	Read and Write	Off	Cached	High latency	Individual threads only

Table 3.1: GPU's memory description.

3.2.3 Processors/Hardware Architecture

In the GeForce GTX 680, there are 4 *Graphics Processing Clusters,* GPCs [138]. Inside of each GPC, there are two next-generation *streaming multiprocessors*, SMs. Thus, in the GeForce GTX 680, there are eight next-generation SMs. SMs are considered to be the heart of the GPU architecture as processes distributed to SMs can be conveniently coordinated by the programmer [139]. Fig. 3.3 shows a simplified block diagram of the SM. The working principle in the SM is as follows:

- Instructions are cached in the instruction cache unit. During device code execution, blocks are assigned to SMs. One block can be assigned to one SM only. This assignment is not controlled by the programmer.
- Each block is broken into warps with 32 threads or less, as discussed in Section 3.2.1.
- Warps are assigned to the warp schedulers.
- Dispatch unit is coupled with the warps schedulers. An example of warp schedulers coupled with dispatch unit is given in A.1.4. Note that, the programmer does not

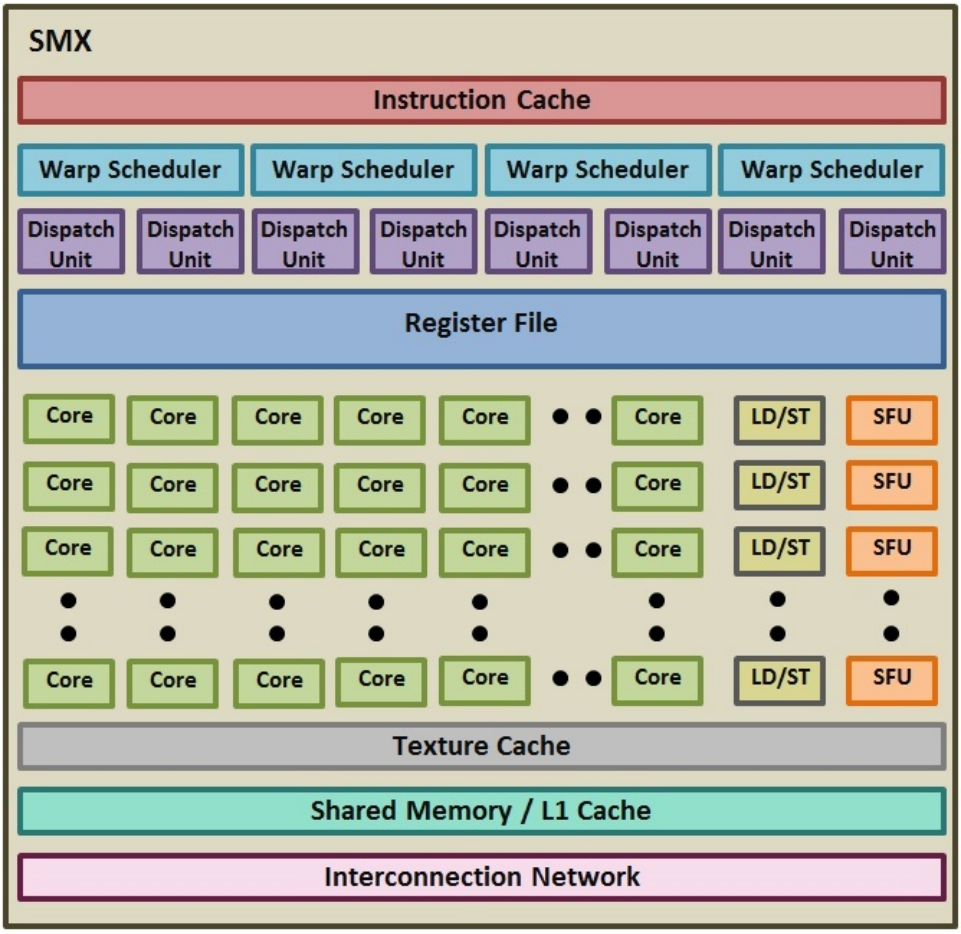


Figure 3.3: Simplified block diagram of the next generation streaming multiprocessor (SM).

control the warp scheduler and dispatch unit.

- Instructions are assigned for execution from the dispatch unit depending upon the availability of the execution units.

- There are several execution units like core and *Special Function Unit,* SFU. Cores are dedicated for executing arithmetic operation, and SFUs are dedicated to executing the special operations: sine function, cosine function, square root function, etc.

	GTX 680	**K20**	**K80**
Number of GPUs	1	1	2
CUDA Capability	3.0	3.5	3.7
Global memory	2 GB	5 GB	24GB
Number of SMX Units	8	13	26
CUDA Cores	1536	2496	4992
GPU Clock rate	1059 MHz	706 MHz	560 MHz
Memory Bus Width	256-bit GDDR5	320-bit GDDR5	384-bit GDDR5
Memory Bandwidth	192.2 GB/s	208 GB/s	480 GB/s
Graphics Card Power	195W	225W	300W
Performance (single precision)	3090 Gflops	3.52 Tflops	8.74 Tflops
Shared memory per block	48KB	48KB	48KB
Max Registers available per thread	64	255	255
Warp size	32	32	32
Max. threads per block	1024	1024	1024
Unit price	CAD 566	CAD 3650	CAD 13094

Table 3.2: Device specifications for GeForce GTX 680, Tesla K20 and Tesla K80.

3.3 GPU Specifications

NVIDIA's GeForce GTX 680 is used in this thesis. Table 3.2 shows the detailed technical specification of the GeForce GTX 680, Tesla K20 and Tesla K80. From the comparative table, it is clear that K80 has more resources for parallelism compare to the two other GPU models. The unit price is taken from the amazon.ca on September 13, 2017.

Chapter 4

System Model and The Path Planner

This chapter discusses the proposed UAV path planner for WSN deployment. The proposed planner consists of two functional blocks: a TSP (travelling salesman problem) solver and a PRM (probabilistic roadmap) solver. The high-level TSP solver computes an optimum global path for the UAV to follow across all network nodes while the low-level PRM solver ensures safe, obstacle-free, node-to-node routes. Preceding a detailed discussion on the proposed path planner, background information on the UAV's operating environment and software/hardware system models are given.

4.1 Environment Description

A rural environment is considered in this thesis for the network construction. The motivating assumption here is that the need for automated WSN deployment is more likely in such locales rather than heavily resourced urban areas. Further, the environment under consideration is assumed to be a cluttered one, meaning that it has many obstacles and thus requires more than a rudimentary point-to-point plan of any autonomous vehicle operating therein. Generally speaking, the obstacles may be trees, mountains, forests, electrical transmission apparatus, etc.

Due to the obstacles in the environment, the UAV requires an effective path planner

and control system so that the UAV will not hit any of the obstacles on the way to its destination. The size and the shape of the obstacles vary, for example, [112] used square shaped obstacles in a grid map, [140] used circle shaped obstacle in a 2D environment, [141] used cylinder shaped obstacle in a 3D environment. In this thesis, sphere-shaped obstacles are used in a 3D environment as abstractions of realistic environmental features. In this study, less than 5% of the total volume is occupied with obstacles. The size of the area (on the ground) where the network is constructed is assumed to be 10,000 m^2. Assuming sensor nodes with a 2-m communications range(as in [79]), approximately 2500 nodes are required to build a network over this space. The maximum flying height is assumed to be 50 m. The spatial dimensions, the number of sensor nodes, and altitude of the UAV are inspired by [142] where a new communication protocol is proposed for the communication system between a large WSN and a UAV. Though for the considered space 2500 nodes are required, the experimental section of Chapter 5 and 6 includes the result for 4096 nodes to show the reader the work capability of the planner.

4.2 System Model

This section provides background information on the possible UAV and its load for the aforementioned deployment challenge. A popular autonomous helicopter model, the Yamaha RMAX [143] is considered as the UAV in this thesis. This UAV can carry a payload maximum of 28-kg and itself has a maximum takeoff weight of 94-kg. This weight capacity allows the UAV to load a complicated control system that can be coarsely subdivided into the hardware and the software blocks. A simplified example of the control system model is given in Fig. 4.1. In this figure, the hardware devices are presented with blue boxes, and the software blocks are presented with green boxes. The data flow is presented with arrows. The white boxes show the parameters measured by the sensors.

As shown in Fig. 4.1, the UAV is assumed to be equipped with camera, radar, sonar,

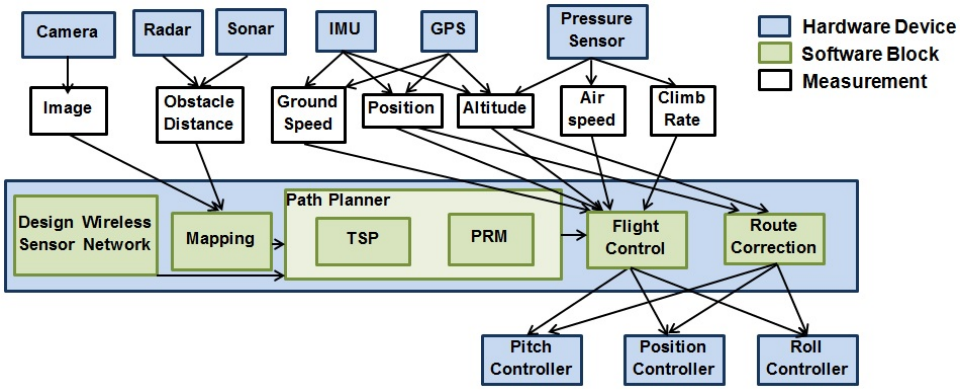


Figure 4.1: Control system of a UAV.

inertial measurement unit (IMU), global positioning system (GPS), pressure measurement unit to generate the 3D map of the environment, computation devices (CPU and GPU) etc. An Intel Core i7 with 3.4 GHz is used as the central processor and assumed to run a Linux operating system. The data collected from the input devices are processed in the processing units, i.e, the software blocks, with multiple algorithms and the resultant data is sent to the output devices to control the UAV physically. The output devices include the actuators for the UAV's critical movement modalities: yaw, pitch, roll, speed control. These actuators receive a signal from the software blocks to control the vehicle physically and to direct the vehicle towards the goal locations.

However, in the process of a UAV flight, the three most important steps are perception, path planning and control of the vehicle according to a plan. Perception refers to the act of generating and updating the map used by the robot to effectively describe its environment. The path planner computes the flight's path from the map (as updated by the mapping block as shown in Fig. 4.1). The control refers to the steering control of the UAV. In short, this block controls the actuators responsible for physically propelling the helicopter and maintaining stable flight. In Fig. 4.1, flight control and route correction works as a control block.

4.3 The Proposed Path Planner

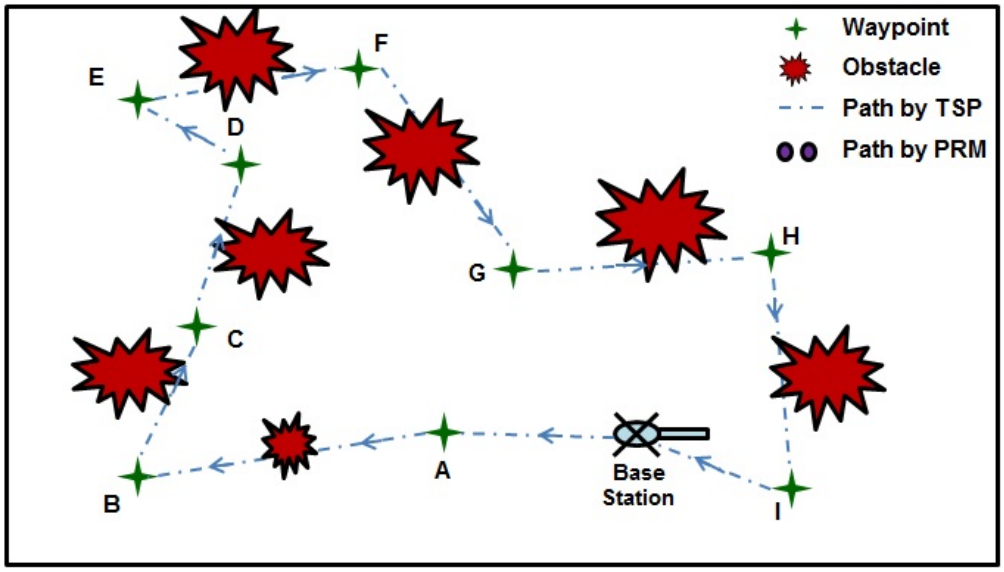
As mentioned earlier, the proposed planner consists of a TSP block and a PRM block which are implemented separately in this thesis. A global distance matrix is considered where the distance between two waypoints is listed. Before the path planner starts working, distances are initialized with the diagonal distance between the two waypoints. In this thesis, it is considered that the 3D waypoints which the UAV must visit are given.

The TSP considers only 2D (x-y) values to calculate the distance. As a simple visual example, the TSP finds the sequence (A-B-C-D-E-F-G-H-I) of the given waypoints in Fig. 4.2a. In its calculations, the TSP considers the distance listed in the distance matrix but does not consider the obstacles in the environment.

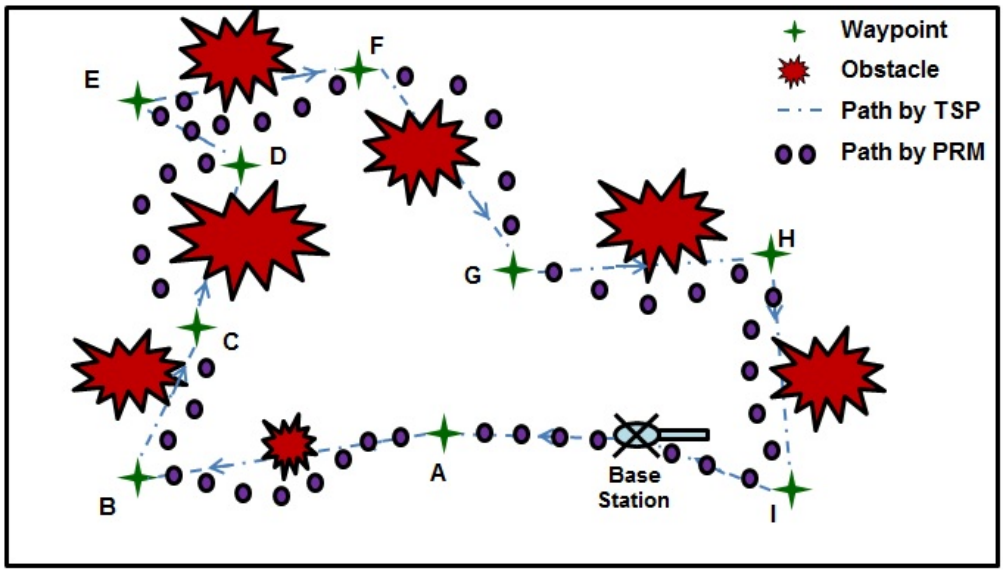
In practice, a diagonal path might not be a safe for the flight due to the obstacles. Thus, as soon as the TSP completes computing the sequence of waypoints, the PRM takes environmental obstacles into account and starts computing a realistic obstacle-free path between connected waypoints (i.e., waypoints linked by the TSP step) as shown in Fig. 4.2b. Essentially, the PRM finds a detour path and the distance of the safe path between two waypoints will be longer than the TSP identified the diagonal path. The PRM considers the given waypoints in 3D-space.

Although the sensor network is built on the ground, it is assumed that instead of landing on the ground, the UAV will deploy its sensors nodes while hovering. The given z coordinate parameter for the waypoint is interpreted as the hovering height.

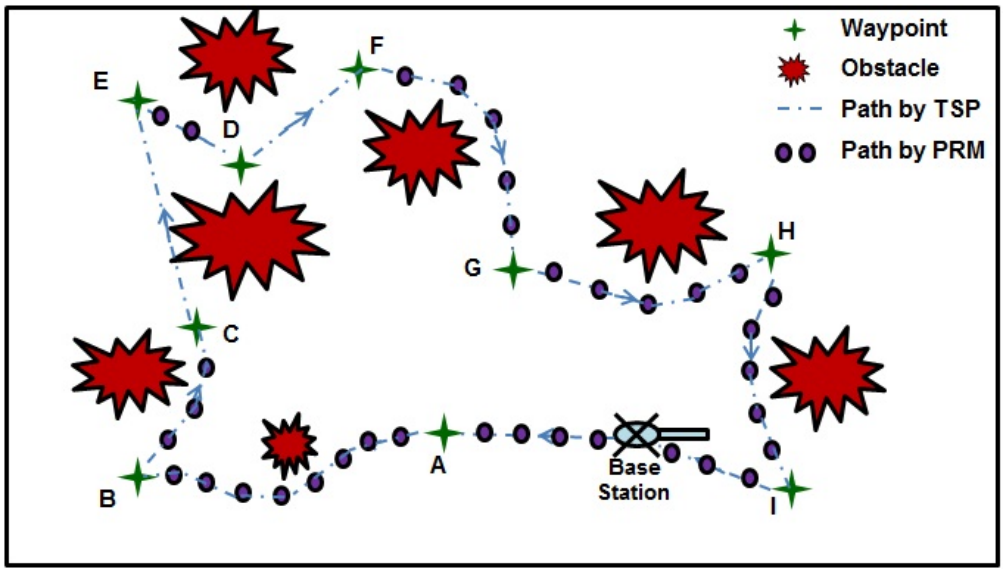
The PRM updates the global distance matrix with its newly calculated distances. The distance is considered infinite when there is no path found between two waypoints. As soon as the distance matrix is updated, the TSP starts calculating again to find a better solution in Fig. 4.2c. In the new solution from the TSP, if there is any new pair of waypoints whose actual flying path is not calculated yet by the PRM block, then the PRM calculates those paths (in Fig. 4.2d). This process continues two or three times to find the final shortest path. The iteration stops when there is no update on the sequence generated by the TSP.



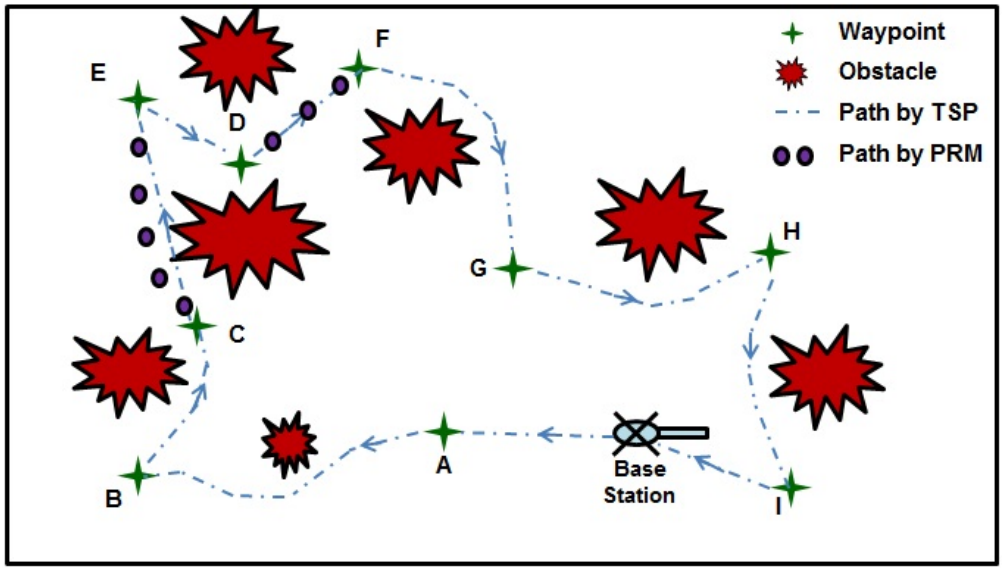
(a) The sequence found by the TSP. Initially, the global distance matrix has the diagonal distances. So TSP finds the shortest path considering the diagonal distances.



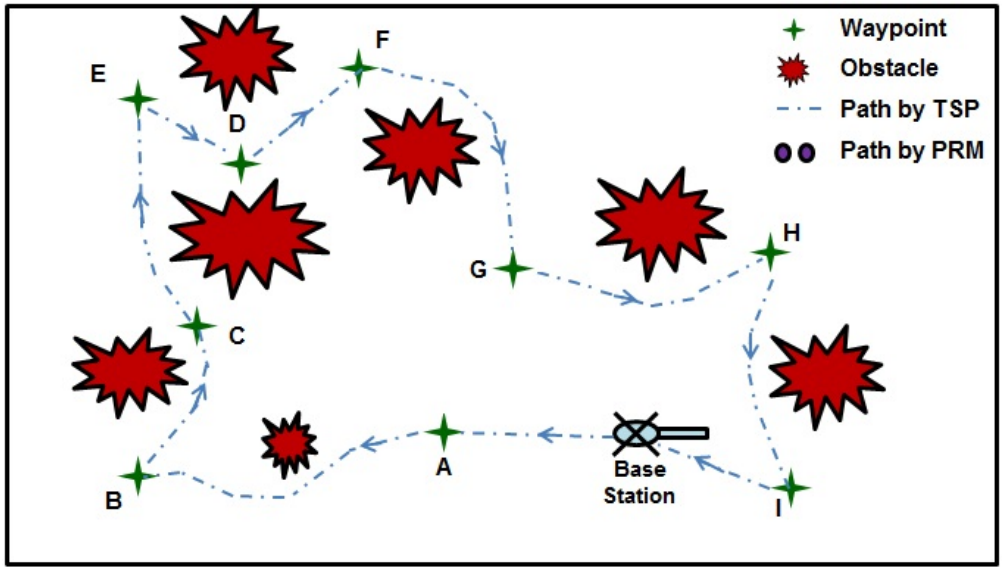
(b) The PRM finds a flying path of the sequence generated by the TSP. The PRM make detours so that the UAV will not hit any obstacles, if required.



(c) The TSP again search the shortest path after the PRM updating the realistic flying distance between two waypoints. In the new path, C-D-E-F path is updated as C-E-D-F.



(d) In the recent updated sequence by the TSP, the flying path C-E and D-F are unknown. The PRM finds the path between C-E and D-F.



(e) For the new update by the PRM, the TSP finds the path again and generates the final shortest path.

Figure 4.2: The proposed path planner generates the shortest path among the given waypoints. The UAV starts from the base station and fly through all of the waypoints before returning the base station again.

Algorithm 1 presents the pseudocode for the proposed path planner (motivated by [86]) that is described in the above paragraphs. Here, D_{matrix} is the distance matrix, v_i is the i-th waypoint and $i = 1, 2, \ldots n$, n is the number of total waypoints. $\{v_i\}$ is the set of waypoints. V is the path which is a set of sequenced waypoints, i.e., $V = \{v_i\}$. $\{ob_j\}$ is the set of obstacles in the environment and $j = 1, 2 \ldots q$, q is the total number of obstacles. In this piece of code, the function TSP 7 finds the shortest path, V according to the distance matrix. It returns the calculated path and a boolean variable whether the path has been updated or not. The function PRM 10 finds the path between the waypoints according to the sequence generated by the TSP and updates the distance matrix. The boolean variable $Update_{TSP}$ and $Update_{PRM}$ becomes true when there is an update made in the process of the TSP and the PRM respectively. The TSP and the PRM block has been implemented and will be discussed in detail in Chapters 5 and 6.

Algorithm 1 The Path Planner

```
1: function PathPlanner({v_i}, {ob_j})
2:     Initialize: D_matrix;
3:     bool Update_TSP ← true;
4:     bool Update_PRM ← true;
5:     while Update_TSP or Update_PRM do
6:         if Update_PRM then
7:             Update_TSP, V ← TSP(D_matrix);
8:         end if
9:         if Update_TSP then
10:            Update_PRM, D_matrix ← PRM(V);
11:        else
12:            Update_PRM ← false;
13:        end if
14:    end while
15:    return V
16: end function
```

4.3.1 Performance Measure

Let us consider that 512 waypoints are given to a network deployment system. The TSP takes 4.27 sec to generate the sequence of waypoints (the detail experimental results are shown in Chapter 5). In the next step, the PRM constructs 511 paths in 1.32 sec (from Chapter 6). The iterative process continues until the final result does not change more than 5%. I am assuming that this iterative process will converge quickly. Though the number of iteration will vary depending on the number of obstacles present in the environment. The more number of obstacles are present in the environment, the initial Euclidian distances stored in the distance matrix will be less accurate. In this iterative process, TSP will take the same computational time in each iteration, but the PRM will require less computational time after the first iteration. The reason for taking less time for the PRM from the second iteration is that PRM requires finding only the new edges updated in the TSP generated sequence. So, we can consider another 1.5 sec for the PRM calculation for the second iteration and so on.

The maximum theoretical performance of a parallel computation can be measured by

Amdahl's law [144]. In general, when multiprocessors are used to run a program in parallel, a part of the program is executed parallelly, and the rest of the program is executed serially. Depending on the percentage of code that can be executed in parallel, Amdahl's law helps to calculate maximum theoretical speedup of the given program regardless of the number of processors. For example, a program takes 1 hour to execute, and 20% of the code can be executed in parallel. That means 80% of the code will be executed serially which will take 48 min and the remaining 20% of the code will be executed parallelly. So regardless of the number of processors, the total execution time will be higher than 48 min. According to Amdahl's law, the maximum theoretical speedup is calculated using (4.1).

$$S_{latency}(s) \leq \frac{1}{(1-p)} \tag{4.1}$$

where $S_{latency}$ is the theoretical speedup of the execution of the program, s is the speedup of the part of the program that benefits from improved system resources, p is the proportion of execution time that the part benefiting from improved resources originally occupied. For the above-stated example, p=0.2, so the maximum theoretical speedup will be 1.25×.

Chapter 5

Sequence Generator for The Path Planner

The sequence generator is the part of the path planner tasked with finding the shortest flying path through a set of given waypoints. This chapter details the sequence generator implementation and the results recorded from the sequence generator during this study. A description of the sequence generator as designed for standard CPU-based serial execution is first given followed by a description of its design for parallel execution on a GPU. The parallel design works 4.82× faster than its serial counterpart. The sequence generator presented can operate on sequences of up to 4096 waypoints.

5.1 Problem Definition

The problem addressed in this chapter is that of quickly finding the shortest path through a large number (1000s) of waypoints using an on-board computational system suitable for the UAV under consideration (UAV specifics are outlined in Chapter 4). This thesis assumes that, besides some standard central processing unit, the UAV's on-board computational system also hosts a graphics processing unit.

For UAVs, as well as many autonomous robot systems, finding a sequence of multiple

destinations or waypoints through physical space is a common challenge and may be suitably addressed in the context of the travelling salesman problem (TSP) [145]. The TSP refers to the challenge of finding the shortest route through all vertices in a graph. The goal of aerially deploying wireless sensor nodes using a UAV conforms to this problem statement as well. In this thesis, the number of waypoints is assumed to equal the number of sensor nodes required for deployment.

The parameters given in a TSP consist of a set of waypoints plus the costs associated with travelling from one waypoint to any other waypoint. The cost can be the spatial distance or the fuel consumption or any other metric that needs to be optimized in the context of full graph traversal. As a result, the TSP has been applied to many problems beyond those seeking to minimize travel distance.

The TSP is an NP-hard problem that can be solved in two basic ways: via an exact algorithm or an approximate (heuristic) algorithm [28]. The former finds the global optimum at the expense of computation time which increases exponentially with the number of waypoints. Conversely, the heuristic approach solves the problem faster but cannot guarantee the best optimal solution although nearly optimal results are possible. Researchers have solved the TSP using a number of different algorithms and demonstrated computation in both serial and parallel manners.

Most of the serial-computing approaches to the TSP essentially serve as offline solutions due to the relatively long computational time required. To date, the best TSP solver is the Concorde TSP [93] which finds the optimal result for all of the problem instances in the reference TSP library, TSPLIB [94] (the largest problem in the TSPLIB contains 85,900 cities/nodes).

In Concorde TSP, multiple heuristic algorithms are used including the cutting-plane method, the minimum spanning tree, the nearest neighbour, the branch and bound method [93]. Depending on the number of cutting planes and the structure of the search tree, the execution time varies markedly for different problems with nearly identical node counts in

Concorde TSP. For example, the computation times for 1000 waypoints and 1002 waypoints (from the TSPLIB files named dsj1000 and pr1002, respectively) are listed as 410.32 sec and 34.30 sec [93] respectively on a 500-MHz Compaq XP1000 workstation.

Although Concorde TSP is generally recognized as the best available tool, it is inappropriate for use in online settings due to its unpredictable execution time. Other important serial TSP solvers using hybrid algorithms (discussed in Chapter 2) suitable for offline use are described in [95, 96, 97].

Due to the large computational workload presented by it, parallel computational solutions based on CPU [98] and GPU threads [99, 100, 101, 104] are often used for near optimal solution. For example, O'Neil et al. [99] implemented a TSP solution using a GPU employing the iterative hill climbing (IHC) method. This solution runs 60× faster than its CPU-only counterpart on 100 waypoints. Rocki and Suda [100], working with an improved IHC technique, managed to solve a 6000-waypoint problem using a GPU, but their parallel implementation is two times slower than their serial implementation.

Rocki and Suda [101] also worked on a parallel implementation of the TSP using an iterative local search method that achieves 10× to 50× faster time-to-solution than their serial implementation for thousands of waypoints. Fu et al. [102] used the "ant colony" algorithm for a parallel implementation of the TSP and achieved a 30× speed-up for the problem over serial with 1000 waypoints.

Another approach, the genetic algorithm (GA), has achieved 24.2× speed-up over serial for only 512 waypoints [146]. Although the nature of the GA maps well to the GPU's single-instruction-multiple-data (SIMD) nature, GA has not been used for solving the TSP using GPU with thousands of waypoints by 2013. There were research works solving TSP with thousands of waypoints using GPU with algorithms other than GA at 2013. This presented an attractive opportunity to exploit the GA's properties for GPU-enabled online path planning on large network deployment problems. So, this thesis solves the TSP with 4096 waypoints using the GA for the first time using GPU in 2013 [147], and the parallel

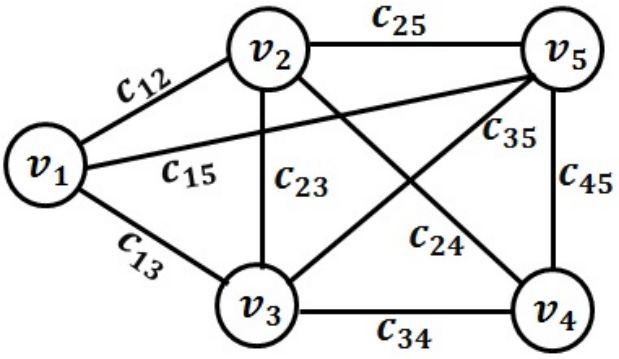


Figure 5.1: Travelling salesman problem: graph representation, $G(V, E)$.

implementation executes 4.8× faster than the serial implementation. Later, in 2014, a 2932 waypoint, GA-based, the solution was described in [18]. In [18], the authors reported up to 51026× speed-up of their parallel implementation over their CPU-only implementation using 2-opt local search with GA. But the time for parallel implementation presented in the paper [18] is 10× to 60× higher than this research study (the same GPU model is used in both studies) which essentially relegates the approach for offline use.

Sections 5.1.1, and 5.1.2 detail the TSP and the GA, respectively.

5.1.1 Travelling Salesman Problem

The classic TSP states that the salesman wants to travel through a collection of cities, visiting each only once, using the shortest possible path. The TSP may be studied in terms of a weighted graph as exemplified by the illustration in Fig. 5.1. In Fig. 5.1, the graph, $G(V, E)$ has two parameters V and E; V is the set of all waypoints present in the problem (in this example the individual waypoints are denoted with v_i with index $i = 1, 2, \ldots, 5$). E is the set of all possible edges connecting two different waypoints. All of these edges have a cost (weight). Spatially speaking, longer edges correspond to higher costs.

Considering a problem with n waypoints, the binary variable, x_{ij} ($i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$) denotes the presence ($x_{ij} = 1$) or not ($x_{ij} = 0$) of an edge from waypoint v_i to waypoint v_j.

Formally, (5.1) states the objective of the optimization problem:

$$\min \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} c_{ij} x_{ij} \tag{5.1}$$

Such that,

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1, 2, \ldots, n; \quad i \neq j \tag{5.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, 2, \ldots, n; \quad j \neq i \tag{5.3}$$

$$x_{ij} \in 0, 1 \tag{5.4}$$

Here c_{ij} is the cost associated for travelling from waypoint v_i to v_j. In this work, the cost is considered as the spatial distance between the two waypoints. So for every different pair of waypoints, the corresponding c_{ij} is different. Eq. (5.1) indicates that the value of x_{ij} needs to be selected in such a way that the summation of costs, c_{ij} will be minimized. The first constraint (5.2) makes sure that each waypoint will be entered only once and the second constraint (5.3) makes sure that each waypoint will be exited only once.

5.1.2 Genetic Algorithm

The genetic algorithm is used to solve the TSP as mentioned in Section 5.1. In the field of artificial intelligence, the GA is a search heuristic that mimics the process of natural selection. This heuristic is routinely used to generate useful solutions to optimization and search problems. The GA belongs to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution. The GA finds application in many fields including bioinformatics, phylogenetics, computational science, engineering, economics, chemistry, manufacturing, mathematics, physics, etc. [148].

The working principle of the GA algorithm can be divided into five steps that are discussed in the following sections. These steps continue until the final result is found. The

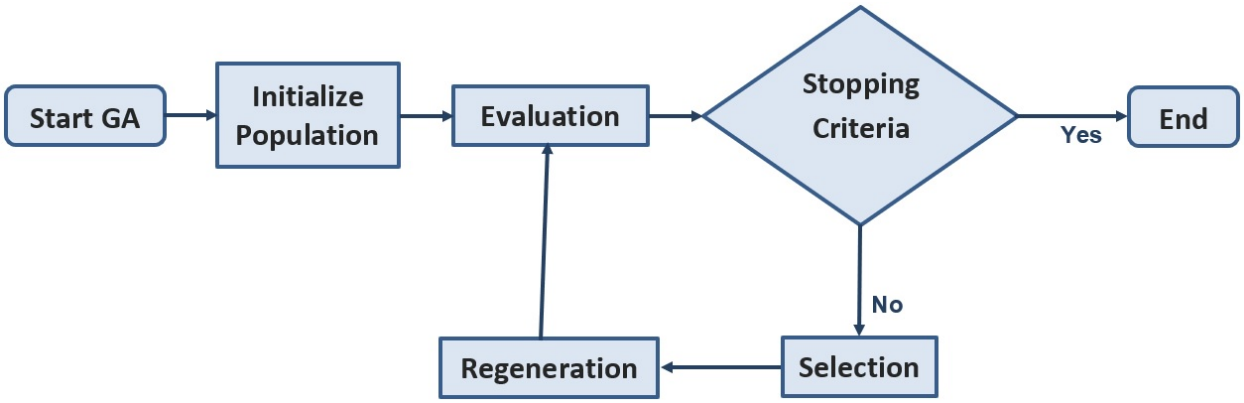


Figure 5.2: Flowchart for the genetic algorithm.

steps are presented in flowchart form in Fig. 5.2.

Initialize Population

The GA starts with an **initializing population** phase. The population is often referred as *combinations* in this thesis. Here, a combination refers to a possible sequence of all waypoints (the optimal sequence being indicative of the shortest route and hence the solution sought). The sequence of the waypoints is chosen randomly in this step. The total number of combinations depends on the size of the problem and how the program is designed.

Evaluation

Next, the **evaluation** step is executed whose function is to compute the total distance of each combination identified in the previous step. Then, the combinations are sorted in ascending order (according to the total distance).

Selection

The **selection** step comes after the evaluation phase. Specifically, combinations with shorter paths are selected as parent combinations. New combinations are then generated from the parent combinations with the anticipation that combinations of superior waypoint sequences will themselves lead to be better waypoint combinations. Combinations other than the

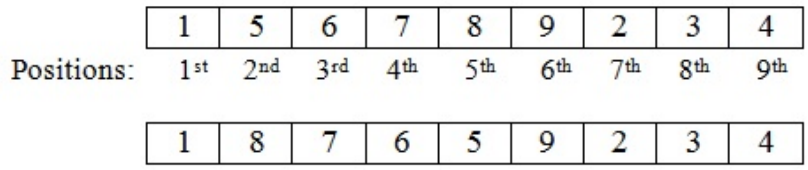


Figure 5.3: Flipping.

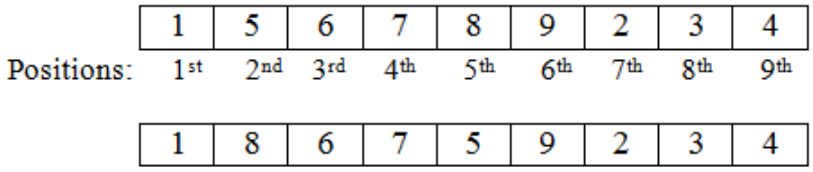


Figure 5.4: Swapping.

selected parents are then replaced by the newly generated combinations which are called child combinations.

Regeneration

Regeneration is the process of generating new combinations from the parent combinations, defined above as the combination selected in the previous selection step. In this thesis, three different actions are taken to achieve regeneration: flip, swap and slide. The next three paragraphs detail these actions.

Flip: During the regeneration process, a part of a parent combination is flipped to generate a child combination. In the code, any two positions between 1 to n are randomly selected. For example, 2 and 5 are selected randomly (in Fig. 5.3). After applying the flipping instruction, the waypoints in the 2nd position to the 5th position will be flipped. Fig. 5.3 shows an example of the flipping action. The first row shows the given parent combination and the second row shows the child combination after the flipping action.

Swap: In the swapping process, a new combination is generated by swapping two waypoints randomly. Fig. 5.4 shows the swap action assuming the random positions are 2 and 5.

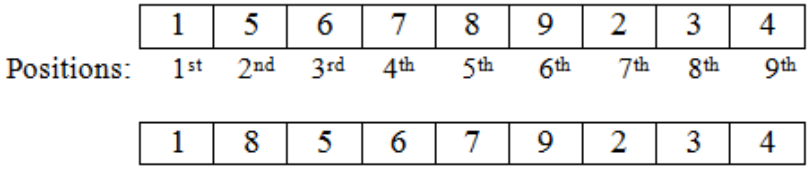


Figure 5.5: Sliding.

Slide: The last technique used for regeneration is combination sliding. In this step one position is selected randomly and the waypoints from the selected position to the end are shifted one position to the right from their original position. Fig. 5.5 shows the sliding action with the same example as considered above.

Stopping Criteria

The combination with the shortest distance from the evaluation step is reported as the final result when the **stopping criteria** is satisfied. In general, the stopping criteria is set either as a fixed number of iterations or when only marginal result improvements are noted over a few consecutive iterations. A fixed number of such criteria-defined iterations will make sure that the program will not run for an infinite time.

5.2 Serial Execution

This section discusses the process of solving the waypoint sequencing problem serially in this thesis on a CPU using the GA discussed above in Section 5.1.2.

The pseudo code for implementing the TSP with GA is given in Algorithm 2. The process starts with reading the set of waypoints $\{v_i\}$, where $i = 1, 2, \ldots n$ and n is the total number of waypoints, from an input file. It is assumed that the input file contains the list of waypoints which are generated and written by the sensor network constructor. The start location for the UAV is the base station.

The first step towards solving the TSP is to generate the initial combinations or paths randomly which traverse all of the waypoints. Here, $\{V_j\}$ is the set of generated paths in

line 2, where $j = 1, 2, \ldots k$, k is the total number of generated paths, and V is the complete path through the set of sequenced waypoints $(v_i : i = 1, 2, \ldots n)$.

Then the iterative process starts and continues until the stopping criteria are satisfied.

In each iteration, the total distance C for each path is calculated; a part of the GA's evaluation step. The best path out of every four combinations is selected and the rest of the three combinations are replaced by regenerating the three remaining paths from the selected best combination. $\{V^{'}\}$ denotes the set of best paths and $\{V^{''}\}$ denotes the set of regenerated paths shown in line 6. By selecting the best path in every four paths, the algorithm might select a path which is worse than a replaced path in some other group. As a result, the algorithm does not follow the greedy approach. At the end, the best path, P, a set of sequenced waypoints, $(v_i : i = 1, 2, \ldots n)$, is selected.

Algorithm 2 Pseudo code for the serial computation

```
1: Input: {v_i} where i = 1, 2, ... n
2: Initialize {V_j} where j = 1, 2, ... k
3: while !stopping condition do
4:     {C_j} ← {V_j}.
5:     {V'} ← {V_j}.
6:     {V_j} ← ({V'}, {V''})
7: end while
8: P ← {V_j}
9:  return P
```

5.3 Parallel Execution

In this thesis, the TSP has been implemented in a parallel form using a novel algorithm and memory architecture on a commercial GPU. In general, GPUs have limited on-chip shared memory (the high-speed counterpart to the large, but the relatively slow off-chip memory). Using the on-chip shared memory, one can solve a limited number of waypoints only as is discussed in Section 5.3.1. For a more effective parallel implementation, the waypoint sequencing problem is divided into smaller sub-problems to accommodate sufficient data

distribution to the shared memory. In other words, waypoints are divided into small groups, and the shortest paths through the waypoints in the same small group are calculated using the GPU. At the end, all of the shortest paths from different small groups are connected together to find the final shortest path through all of the waypoints.

For dividing the problem into several sub-problems or small groups, clustering and nearest neighbour techniques have been used in this thesis. The next sections first describe the limitations of implementing a larger problem in the GPU and then discuss how the larger problem is divided into multiple sub-problems and how each of the smaller problems is solved using a GPU.

5.3.1 Challenges/Limitation

Generally speaking, the main challenge to distributing a problem across a GPU is the issue of providing enough memory space for the problem as a whole. This limitation is illustrated with a simple quantification below.

The GPU has both off-chip and on-chip memory components. Naturally, the off-chip *global memory* is larger but slower than its on-chip *shared memory* counterpart. So it is a better idea to bring the combinations and the n-by-n distance matrix (whose elements represent distances between two waypoints) to the faster shared memory.

Fig. 5.6 shows an example distance matrix for a problem with four waypoints. The diagonal elements of the table are zero. The upper and the lower triangular matrix elements (highlighted with dotted outlines in the figure) contain the same value assuming symmetric distance metrics (an ongoing assumption in this thesis). Therefore, the upper triangular matrix contains all the necessary problem information in $n(n-1)/2$ elements. A separate distance matrix is required to calculate the total distance for each combination being processed by the GA.

As mentioned earlier in this section, it is a better idea to bring the GA waypoint combinations and the distance matrix to the faster shared memory; now I will discuss how many

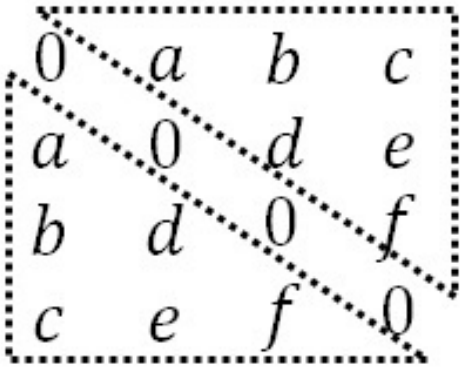


Figure 5.6: A distance matrix example for a four-node waypoint combination.

waypoints can be fit within the shared memory. The on-chip shared memory has very limited memory space. For example, the size of the shared memory is 49152 bytes per block, in NVIDIA's GeForce GTX 680, as well as in high-end GPUs like the Tesla K80. Thus, if we assume that each waypoint requires 2 bytes of memory (unsigned integer types are used to hold sequence number waypoints) and each element of the distance matrix is 4 bytes (float type is used), then, as detailed below, the TSP algorithm outlined above encounters several critical limitations.

Generally, problem size and shared memory must adhere to the following inequality

$$n \cdot t \cdot (2\ [\text{bytes}]) + \frac{n(n-1)}{2}(4\ [\text{bytes}]) \leq 49152\ [\text{bytes}] \tag{5.5}$$

The left side of (5.5) presents the total shared memory required per block where n is the number of waypoints, t is the number of threads and $n(n-1)/2$ is the number of elements in the symmetric distance table. The right side of (5.5) presents the total amount of shared memory available per block that should not be exceeded by the left side of (5.5). In (5.5), the variables are t and n. The value of t should be in the range of 32 to 1024 as the warp size is 32 threads (as discussed in Ch.3, a group of 32 threads makes full use of the resource), and the maximum limit of threads per block is 1024. So, a widely available GPU can accommodate only $n = 142$ (when $t = 32$) or $n = 23$ (when $t = 1024$) waypoints in its shared memory.

In these cases, the combinations and distance matrix will occupy 49132 and 48116 bytes of memory respectively. Again, relying on maximized shared memory usage is not advisable due to the communication delay imposed by regular exchanges with this off-chip component.

5.3.2 Clustering

A clustering technique is used to divide the problem into several smaller problems. In this process, closer waypoints are grouped together, and each group is generally referred to as a *cluster*. Each cluster is formed around uniformly seeded *cluster heads*. Specifically, after selecting a set of evenly distributed positions for cluster head locations, a pre-determined number of waypoints are selected to form a cluster around each head. The waypoints are selected according to those whose total distance to the cluster head is a minimum. Algorithm 3 presents the pseudo code used for clustering in this thesis.

The total number of clusters is decided at the beginning. The number of clusters is set in such a way that each cluster will have the same number of waypoints. But if the total number of waypoints is not evenly divisible, then there will be two types of cluster groups defined by their waypoint count. That is, a cluster group type will contain the same number of waypoints. For example, if there are 64 waypoints and 8 clusters, each cluster can contain 8 waypoints. But if there are 67 waypoints and 8 clusters, 5 clusters will contain 8 waypoints each and 3 clusters will contain 9 waypoints each ($5 \times 8 + 3 \times 9 = 67$). Maintaining an equal number of waypoints per cluster helps to distribute the workload of threads in the GPU evenly.

The waypoints inside the clusters are tagged with a sequence number between 0 and $n/m - 1$, where n is the number of waypoints, and m is the number of clusters. So, the combinations of waypoints for each cluster should consist of 0 to $n/m - 1$ waypoints (though their locations: x and y coordinates are different). The advantage of tagging a generic sequence number to the waypoints inside all clusters is that the waypoint combination for one cluster can be used by all other clusters. In this way, the waypoint combination data

that needs to be transferred from CPU to GPU is reduced by a factor of m.

Another advantage to using clusters is that the data transferred from the CPU to the GPU for the distance table is also reduced. Without the clustering techniques, for an n-waypoint symmetric problem, the size of the distance table is $4n(n-1)/2$-bytes considering float data types. After applying the clustering techniques, the size of the distance table becomes $4\frac{n}{m}(\frac{n}{m}-1)/2$-bytes, where the 4-factor accounts for the use of the float data type.

As outlined above, in the process of clustering, the cluster heads are placed (with an equal distance from one another) in the area where the waypoints exist at the beginning. Then the waypoints are assigned to their closest cluster head, and the location of the cluster heads are updated in each step so that the total distance from the head to each of the waypoints in that cluster will be minimized.

Algorithm 3 shows the pseudo code for the clustering algorithm where $\{CH_j\}$ is the set of cluster heads, CH_j is the jth cluster head, $j = 1, 2, \ldots m$, m is the total number of cluster heads. In line 3, the ith waypoint, v_i is assigned to a cluster head CH_{v_i}. Then the position of cluster heads are updated in line 4. The process stated in line 3 and 4 iterates q times. The work complexity of the algorithm is $O(n + m + nm)$. Fig. 5.7 shows an example of the clustering technique.

Algorithm 3 Pseudo code for the clustering

1: *Initialize:* $\{CH_j : j = 1, 2, \ldots m\}$
2: **for** $l = 1$ to q **do**
3: $\quad CH_{v_i} \leftarrow \forall v_i$
4: $\quad$ Update $CH_j \forall j$
5: **end for**

5.3.3 Nearest Neighbour Technique

A nearest neighbour technique is used for finding the start and the end waypoints for each cluster. Using this approach, the number of waypoints participating in the generation of combinations reduces by two since the start, and the end point are fixed. This approach

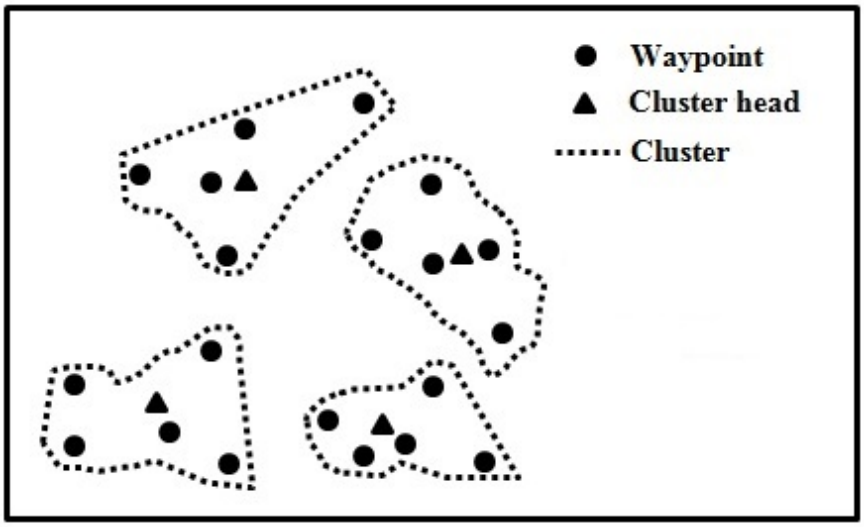


Figure 5.7: An example of clustering.

also reduces the amount of data required to transfer from CPU to GPU. Fig. 5.8 shows an example of using the nearest neighbour technique on top of the clustering method.

5.3.4 GPU Implementation

After dividing the problem into multiple smaller problems, the GA is applied to each of the smaller problems, and a part of the algorithm is executed on the GPU. The initial waypoint combinations for the smaller problem are generated by the CPU and transferred to the global memory of the GPU. The combinations are then brought to the on-chip shared memory.

The program is designed in such a way that each thread is made responsible for a single combination of a cluster. For example, if the number of waypoint combinations for a cluster is u and there are m clusters, there will be $m \cdot u$ threads in total.

The number of threads per block is kept as an integer multiple of the warp size for better efficiency. Each block is assigned to a single cluster, but a single cluster is distributed among multiple blocks. The data are kept in the shared memory in such a way that the read and write operation on the shared memory will be coherent.

The evaluation step is applied next where distance for the corresponding combination is calculated. A bitonic sort [149] (discussed in Appendix A) is applied to sort the combinations

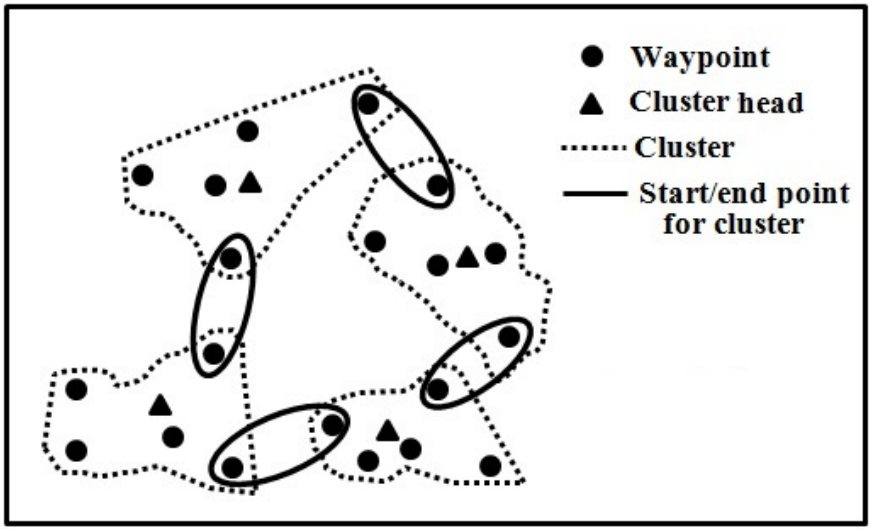


Figure 5.8: Example of clustering in combination with nearest neighbour identification.

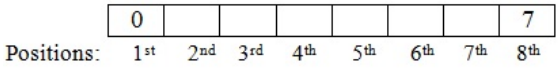


Figure 5.9: Combination for 8 waypoints.

according to the calculated distances within a block.

The regeneration step is applied only when the number of waypoints per cluster is greater than 8. Conversely, when the number of waypoints per cluster is 8, the waypoints inside a cluster are tagged as 0 to 7, and the start and the end points are 0 and 7 respectively as shown in Fig. 5.9. The rest of the 6 positions (2nd to 7th) will be filled out by the other waypoints inside the cluster. The maximum number of possible combinations for these empty spots (in Fig. 5.9) is the factorial of 6, which is 720. If we want to check all possible combinations of such a case, 720 threads per cluster are required. Using 720 threads per cluster is affordable for a GPU like the GeForce GTX 680, if the number of total blocks will not exceed its maximum limit. In such situations, the regeneration step is skipped, and all possible combinations are checked.

When the number of waypoints per cluster exceeds 8, it is not possible to use all combinations. In this case, the regeneration is made inside the threads within the same block which reduces the memory access time (as it is not accessing the off-chip global memory).

At the end, the best result from the block is written to the global memory so that the results from the other blocks for the same cluster can be compared. At this step, a barrier is used to keep the blocks waiting while other blocks are lagging behind. I have used separate barriers for different clusters. Atomic instruction [137] has been used for the barrier calculation. The clusters are independent of each other. Using separate barriers for different clusters also reduces the waiting time. The results are compared using bitonic sort, and the best result is sent to the CPU.

After getting the result from the GPU, the final path is generated by combining the paths inside the cluster by maintaining the sequence of cluster heads generated in the previous step. Finally, the GA algorithm is applied for the recently calculated path for a few iterations.

Algorithm 4 shows the pseudo code for the TSP implementation as described above. The input of the TSP is the set of waypoints, $\{v_i\}$ and the distance matrix, D_{matrix}. A set of cluster heads are initialized and sequenced, $(CH_j : j = 1, 2, \ldots m)$ using GA. The set of waypoints under CH_j cluster head is presented as $\{v_i : i = 1, 2, \ldots u\}_{CH_j}$. The start, v_{start,CH_j} and the end, v_{end,CH_j} waypoint inside each cluster is defined using the nearest neighbour technique. Line 5 has been implemented on a GPU. The sequenced path, inside the cluster, is V'. So the GPU returns a set of sequenced path for all the clusters. Finally, all paths from the cluster are combined, and a few iterations of the GA are applied to find the final result, the sequenced path $(v_i : i = 1, 2, \ldots n)$.

Algorithm 4 Pseudo code for the parallel TSP

1: Input: $\{v_i : i = 1, 2, \ldots n\}, D_{matrix}$
2: Initialize: $\{CH_j : j = 1, 2, \ldots m\}$
3: $(CH_j : j = 1, 2, \ldots m) \leftarrow \{CH_j : j = 1, 2, \ldots m\}$using GA.
4: $(v_{start,CH_j}, v_{end,CH_j}) \leftarrow \{v_i\}_{CH_j}$
5: $\{V_l^{'} : l = 1, 2, \ldots m\} \leftarrow$ GPU$(\{\{v_i\}_{CH_j}\}, \{v_{start,CH_j}\}, \{v_{end,CH_j}\}, D_{matrix})$
6: $(v_i : i = 1, 2, \ldots n) \leftarrow \{V_l^{'} : l = 1, 2, \ldots m\}$
7: **return** $(v_i : i = 1, 2, \ldots n)$

5.4 Analytic Model of Communication

As is well known, for effective parallel programming the communication load must be minimized. The critical communications in this context include the data transfer between the CPU and GPU as well as between the different memory spaces inside the GPU.

The communication from the CPU to the GPU is influenced by the combination of waypoints (which is $(\frac{n}{m}-2)\cdot u$), the distance matrix (which is $\frac{n}{m}(\frac{n}{m}-1)/2$ or $\frac{n(n-m)}{2\cdot m}$), and the vector for barrier m. By summing up all the values and considering b_w, b_d and b_b as the bytes required by each waypoint, distance matrix, and barrier vector element respectively, the data required to transfer from the CPU to the GPU is expressed as (5.6). In this mathematical expression, 2 is deducted as a result of assuming a fixed start and end waypoint.

$$\left(\frac{n}{m}-2\right)\cdot u\cdot b_w+\frac{n(n-m)}{2\cdot m}\cdot b_d+m\cdot b_b \tag{5.6}$$

Inside the GPU, data was copied from the global memory to the shared memory. Considering u as the number of combinations per cluster, n_{trd} as the number of threads per block, and n_{block} as the total number of blocks, the total data copied to shared memory is:

$$\left[\left(\frac{n}{m}-2\right)\cdot n_{trd}\cdot b_w+\frac{n(n-m)}{2\cdot m^2}\cdot b_d\right]\cdot n_{block} \tag{5.7}$$

Similarly, data are copied back from the shared memory to the global memory. It copies the best result among the threads to the global memory. The expression (5.8) shows the data transferred from the shared memory to the global memory. Here the b_d refers to the distance of the best combination considering the data size of the distance is same as the elements in the distance matrix.

$$\left[\left(\frac{n}{m}-2\right)\cdot b_w+b_d\right]\cdot n_{block} \tag{5.8}$$

Finally, the best combinations for each cluster are sent to the CPU. In this time, the

least amount of data are transferred which is equal to:

$$\left[\left(\frac{n}{m} - 2\right) \cdot b_w\right] \cdot m \tag{5.9}$$

5.5 Experimental Results

Table 5.1 shows the simulation results. The algorithm has been implemented for 128, 256, 512, 1024, 2500, and 4096 waypoints. The data set was generated randomly in a 100×100 area where the wireless sensor network is constructed (as discussed in Chapter 4).

For the serial implementation, Algorithm 2 has been used where combinations are generated first. Each combination contains all of the waypoints, and the child combination is generated from the parent combinations. The process of regeneration and searching for better results repeats iteratively.

The execution time for the serial CPU codes (in Table 5.1) is the time spanning initialization of the random combinations to the generation of the final path result. This time does not include reading or writing the data to any memory.

In the parallel code, additional steps (clustering, preparing and transferring data to the GPU and the CPU) are required on top of the core GA functions. The execution time calculated for the parallel code includes these additional steps. Similarly, the execution time for the parallel TSP code, listed in Table 5.1 starts from the clustering step and ends after the final result.

Reading from Table 5.1, it can be seen that both the serial and the parallel codes give similar results. As mentioned earlier that the sequence generator can generate the near optimal result, the result from the serial implementation is not same as the result from the parallel implementation. For larger problems like 4096 waypoints, the serial implementation's solution is 1.4× longer than that found by the parallel implementation. The reason is that inside each cluster, better paths are generated compare to the serial implementation.

	Serial implementation		Parallel implementation		
Number of waypoints	**Result**	**Time (in sec)**	**Result**	**Time (in sec)**	**Speed up**
128	6917.08	5.1	6869	1.08	4.72
256	13635.74	10.1	14414.1	2.1	4.81
512	24952.67	20.51	21418.6	4.27	4.8
1024	57514.28	44.79	45465.83	9.39	4.78
2500	145098.36	112.3	110939.95	23.4	4.8
4096	246057.18	184.59	174756.52	38.3	4.82

Table 5.1: Results from serial and parallel implementation for different number of waypoints.

Fig. 5.10 shows the percent difference between the tour distance calculated using the serial and parallel implementations. The percent difference is calculated using the following equation where C_{serial} and $C_{parallel}$ is the length of the paths found from the serial and parallel implementation respectively.

$$\frac{C_{serial} - C_{parallel}}{C_{serial}} \times 100\% \tag{5.10}$$

From Fig. 5.10, it can be seen that the parallel implementation finds a better path as the number of waypoints becomes larger. The parallel code also runs 4.82× faster compared to its serial counterpart. The bar graph in Fig. 5.11 shows the execution time for the serial and the parallel code. After investigating the executed serial code, 80% of the serial code executed in parallel. So according to Amdahl's law as stated in Chapter 4, the maximum theoretical speedup will be $(1/(1-0.8)) = 5\times$, which is very close to our experimental result.

For the parallel execution, the execution time on the GPU is in the range of milliseconds. The rest of the time is required to execute the processing in the CPU. Table 5.2 shows the GPU execution time for different combinations of problem parameters. 128 threads were considered for each block for the results listed in Table 5.2. The results show that the parallel TSP calculation works 5.4× to 14.9× faster when the waypoints used per cluster is reduced.

The occupancy of the shared memory is low when the number of waypoints per cluster is small. This result also proves that the performance of the GPU execution decreases when

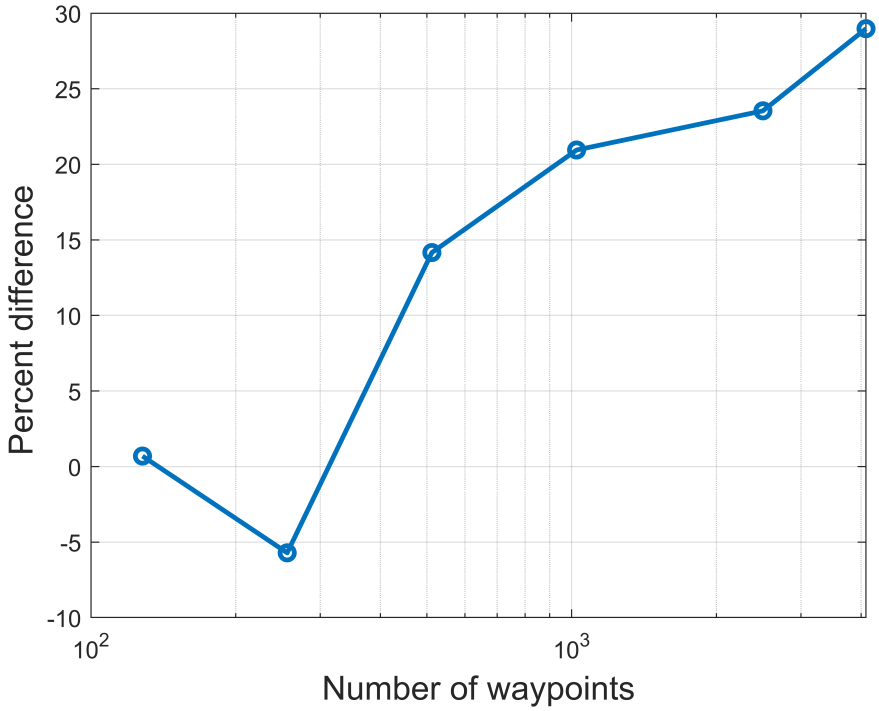


Figure 5.10: Percent difference between the CPU-based serial implementation and the GPU-based parallel implementation.

the occupancy of the shared memory is high. Fig. 5.12 graphically presents the results from Table 5.2. When the number of waypoints per cluster increases, it is not possible to find the exact solution for each cluster as the number of all possible combinations exceeds the limit of our computational system. In that time, the heuristic approach is used. In the heuristic approach, the chance of finding an optimal solution may be increased by increasing the number of combinations. The increased number of combinations require more blocks per cluster. As a result, this increases the execution time nonlinearly for the GPU.

To validate the performance of the algorithm, the parallel code is implemented for multiple TSPLIB problems. The parallel code can solve problems with thousands of waypoints but the average percent of errors compared to the known optimal solution (listed in [93]) increases with the size of the problem. Table 5.3 presents the execution time for the parallel version of the code. The percentage of error for a 280-waypoint problem is 0.8%. The percentage of error compared to the known optimal value increases by up to 38% for the larger

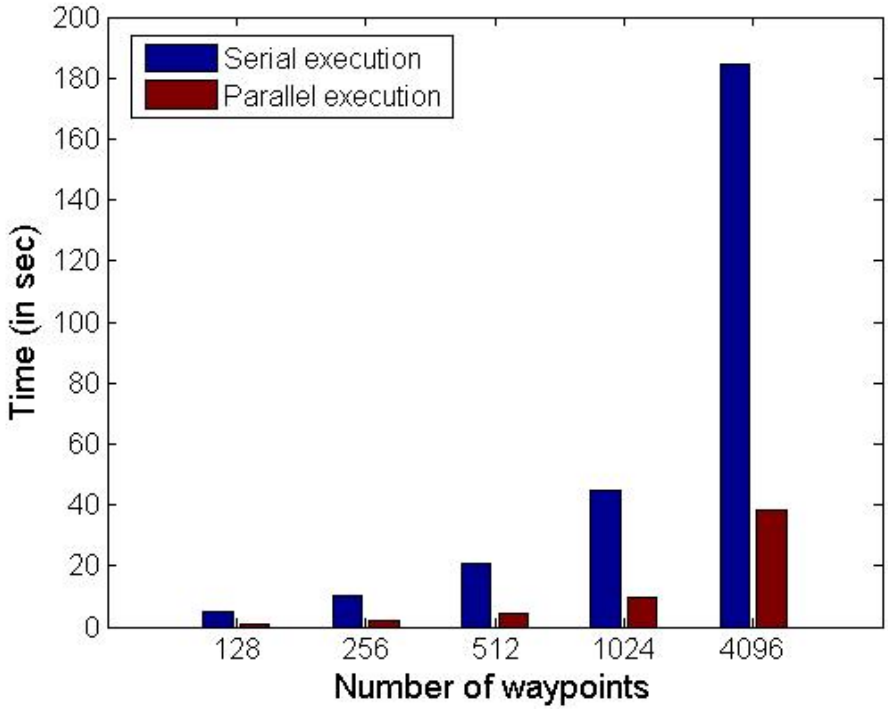


Figure 5.11: Computation time required by the serial and the parallel implementation for different waypoint settings.

Number of waypoints	**Number of clusters**	**Waypoints per cluster**	**Number of blocks**	**Shared memory size (in bytes)**	**Time (in ms)**
256	32	8	192	2416	0.53
256	16	16	5040	4832	2.42
256	8	32	2520	10432	2.87
512	64	8	384	2416	0.55
512	32	16	10080	4832	4.44
512	16	32	5056	10432	5.84
1024	128	8	768	2416	0.79
1024	64	16	20160	4832	8.87
1024	32	32	10080	10432	11.78

Table 5.2: GPU execution results.

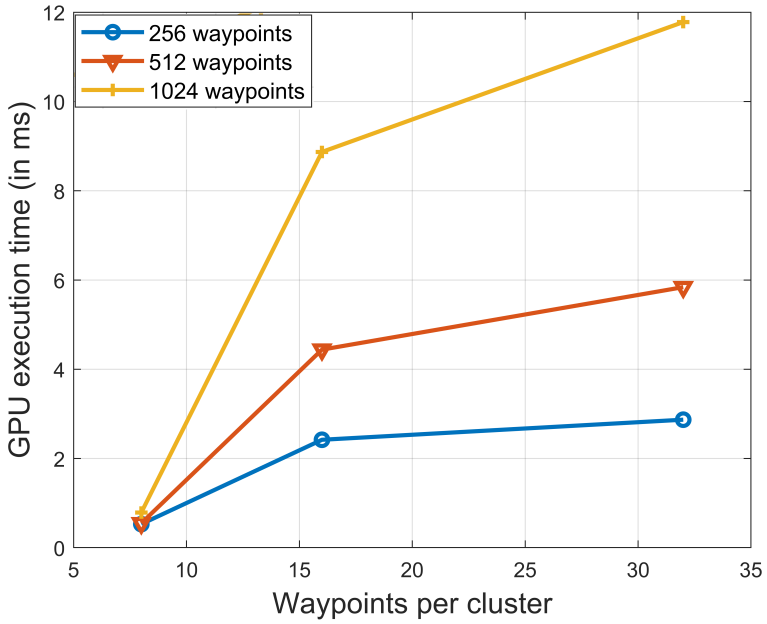


Figure 5.12: GPU execution time varies with the number of waypoints per cluster.

Problem instance	Execution time (in sec)	Average Error
a280	2.3	0.8%
att532	4.67	12%
fl3795	38.59	38%

Table 5.3: GPU execution results for TSPLIB problems.

problem with 3795 waypoints.

5.6 Conclusion

This chapter discussed GPU-based sequence generator implementations that can easily be used in UAVs and hence impose a limited contribution to their payload budget. The sequence generator has been implemented using a genetic algorithm and a customized clustering technique. The data transfer rate from the CPU to the GPU has been reduced by using clusters with the same number of waypoints. Using the clustering approach, the memory size required in the on-chip GPU has also been reduced resulting in faster computations. Although

the parallel implementation requires some computational overhead like, clustering, searching the sequence of cluster heads, etc., the simulation time is still reduced by 4.82×, and the generated path is 1.4× shorter relative to the CPU-only execution. The main advantage of using the algorithm is that it can find a path for large problems with approximately 4096 waypoints using the genetic algorithm. The implemented planner can also generate paths for larger problems consisting of more than 4096 waypoints. In those cases, we should use more clusters and multiple levels of clustering, e.g., generating clusters with the cluster heads.

For future work, using a high-end GPU for the sequence calculation, the implemented sequence generator will be able to generate faster results as the number of available CUDA cores and the memory bandwidth is higher which reduces the time required for read/write instructions for the threads. With a high-end GPU like Tesla K80 (weights 2.2 lbs), we cannot increase the number of waypoints per cluster per block used in this research as the size of shared memory remains constant for all Nvidia GPUs. As a result, instead of shared memory, other memories (e.g., texture memory and global memory) need to be explored for storing the distance matrix so that larger problem can be solved. The drawback of using such high-end GPU is higher energy consumption. For example, the power rating of Geforce GTX 680 and K80 is 195 W and 300 W respectively. The weight difference between these two GPU is not high, e.g., the weight of Geforce GTX 680 and K80 are 3 lbs and 2.2 lbs respectively

Chapter 6

Path Explorer: Probabilistic Roadmap

In this chapter, the algorithmic design and implementation of the UAV's local path planner are discussed. This planner is based on the probabilistic roadmap (PRM) technique which is outlined herein. PRM-based planners designed for serial and parallel execution, respectively, are described. The parallel execution is specifically targeted for GPUs. As key performance demonstrations, the execution times of parallel and serial planner implementations are compared and discussed.

6.1 Problem Definition

The global path planner for a UAV, discussed in Chapter 4, is proposed in this thesis which generates the flight path through all of the waypoints given to the planner. The sequence generator, as discussed in Chapter 5, is studied for finding the sequence of waypoints for the *global path planner*. This chapter has described a *local path planner* that constructs a safe *and* short flying path between each pair of sequenced waypoints. A safe path between two waypoints refers to a route calculation that does not overlap with any obstacles. In the process of building the flying path, the local path planner also calculates the flying distance

between the pair of waypoints.

Constructing a flight path between a pair of waypoints, which is done by the local path planner, is typically done using one of three main approaches [108]: the skeleton approach (a graph is created and searched), cell decomposition (each cell is assigned with a weight depending on the weight of its neighbouring cells), and the potential field approach (the gradient of the potential field is used) as discussed in Chapter 2.

Among these approaches, the skeleton approach is more suitable to implement in parallel as in the skeleton approach multiple "sample nodes" (as discussed in the next paragraph), generated between the start and the destination location, execute the same set of instructions independently. In case of the cell decomposition method, each cell in the grid executes the same set of instructions but the data depends on the neighbouring cells which is not a good fit for parallel execution. In this research, the *probabilistic roadmap* (PRM) algorithm, a skeleton approach, is used as the local planner.

The local planner is required due to the presence of obstacles in the environment. In other words, the UAV might not be able to fly in a straight line from the start to the destination due to obstacles and hence requires a local planner to find an obstacle-avoiding path. In the process of constructing a path from the start to the destination, the PRM generates so-called non-weighed sample nodes, which serve as intermediate waypoints from the start to the destination. To start, the sample nodes are randomly spread over an area encompassing the start and the destination nodes. A list of "neighbour nodes" for each sample node is calculated using the k-nearest neighbour technique [107]. In this thesis, a node is considered as a neighbour node only, if there exists an uninterrupted euclidean path between the node and the neighbouring node. The distance of the Euclidean path is recorded too. After building a graph over the generated sample nodes, a graph search algorithm is used to find the path between the start and the destination by optimally connecting the intermediate sample waypoints.

The process discussed above constructs the flying path between two waypoints only. In

the proposed global planner the total number of considered waypoints is 4096 as in Chapter 5. For that problem, the local planner needs to be applied to 4095 pairs of waypoints, an extensive computational workload. Moreover, the proposed path planner should find the shortest path quickly thus requiring the PRM to work fast too. Again, a GPU-based parallel computing approach, with a number of customizations, is introduced in this thesis to arrive at a PRM suitable for large UAV navigation problems.

6.2 PRM Algorithm

The PRM algorithm studied in this chapter has the following inputs: the start location, the destination location and a map with its obstacles. The output of the algorithm is a path running from the start to the destination that is theoretically negotiable by the machine tasked with traversing it; that is, it is a path that does not intersect obstacles. More specifically, the PRM-generated path is a set of sequential sample nodes. The algorithmic mechanism behind this path generation can be divided into four main steps:

1. sample node generation
2. milestone calculation
3. nearest neighbour search
4. graph search

These steps are discussed in the following sections.

6.2.1 Sample Node Generation

The PRM's objective is to compute a path from the start to the destination through intermediate waypoints. Considering the presence of obstacles, I have assumed that the UAV may not be able to fly in a straight line from the start to the destination. Instead, the UAV

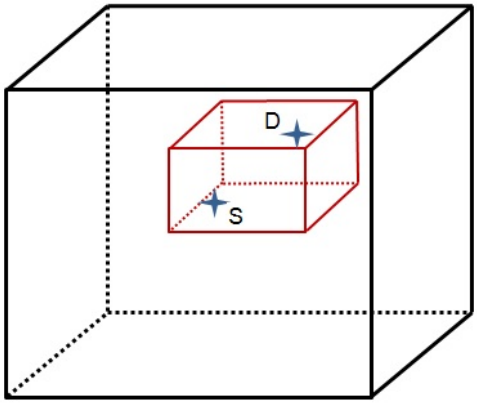


Figure 6.1: A space is separated where the PRM will build a path from the start, S to the destination, D. The sample nodes are placed in the separated (highlighted in red) space.

will fly through intermediate waypoints located between the start and the destination. For the intermediate waypoints, sample nodes are generated and placed in the area where WSN need to be built. The sample nodes can be generated randomly or organized in a grid or over any custom designated spots of the user's choosing.

In this thesis, instead of placing the sample nodes anywhere in the space where the WSN needs to be built, the nodes are placed around the local start and the destination points (i.e. the node couples found by the TSP global planner) where the PRM will find a path as shown in Fig. 6.1. Placing the sample nodes around the start and the destination will prevent us searching for a shorter path far from the destination which also optimizes the calculation process. The number of total sample nodes is defined by the user. Some of these sample nodes are placed in a grid, and the remainder of the sample nodes are placed randomly within the identified space. There will be more discussion on sample node distribution in the next section.

6.2.2 Milestone Calculation

The milestone calculation step checks if any sample nodes generated in the previous step collide with any of the obstacles. Sample nodes colliding with obstacles are discarded. Naturally, the total number of sample nodes is reduced after this step.

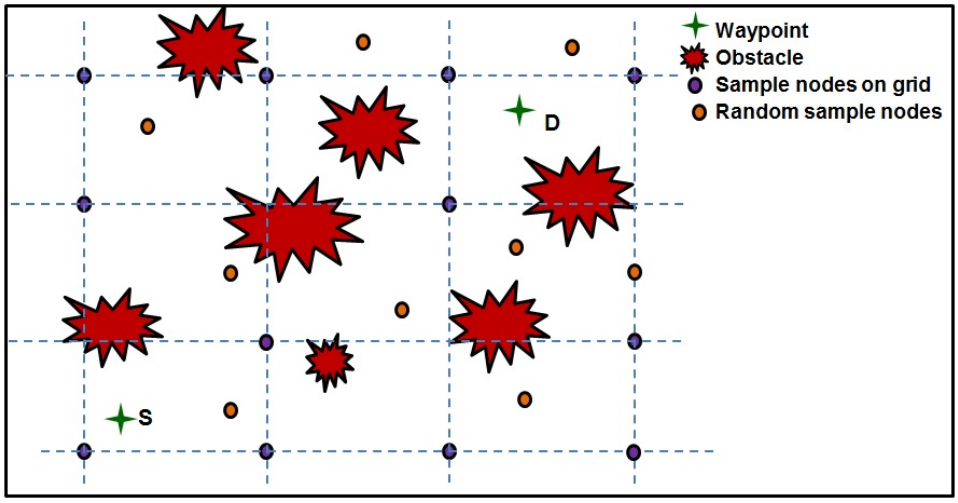
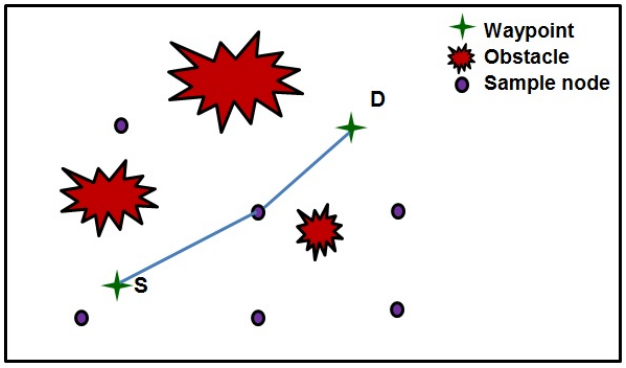


Figure 6.2: An example of how the samples are distributed.

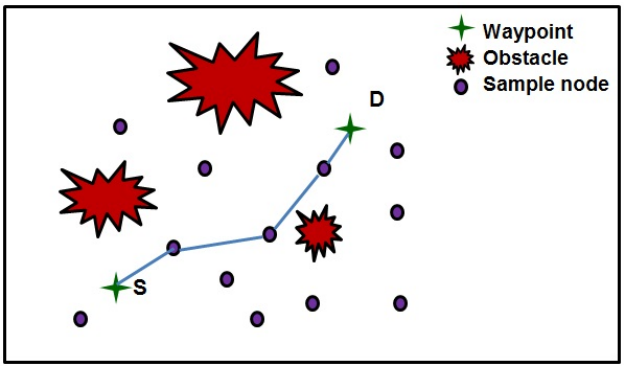
During parallel execution on a GPU, multiple paths are built at the same time using the GPU, i.e., multiple threads work together as discussed in Chapter 3. For balanced work-load distribution to the threads, the number of sample nodes for all the paths are kept equal in this experiment. To keep the number of sample nodes equal for all the paths, the sample node generation step and the milestone calculation step are combined together.

In detail, the confined space (as previously discussed in section 6.2.1) is divided into grids. The size of the grids depends on the total number of samples. Then, the sample nodes are placed on those grid locations if the grid locations are free of any obstacle or far enough from any obstacle for the UAV to execute a safe flight. After placing the sample nodes on the appropriate grid locations, there might be remaining sample nodes that have not been placed yet. For those nodes, random locations are iteratively assigned until placement is achieved in unobstructed space.

Fig. 6.2 shows an example of sample node distribution. In the implementation, a three-dimensional space is considered. But for simpler visualization, a two-dimensional space is shown in Fig. 6.2.



(a) An example of the path search problem with 6 sample nodes and 3 nearest neighbours.



(b) An example of the path search problem with 13 sample nodes and 3 nearest neighbours.

Figure 6.3: An example of how the number of sample nodes and the number of nearest neighbour affects the final path

6.2.3 Nearest Neighbour

After the sample generation and the milestone calculation steps, the nearest neighbouring nodes are calculated. In this step, the k closest nodes are listed for each node. The criterion used for this calculation is the un-obstructed euclidean distance between two nodes. The number of neighbouring nodes considered depends on the number of sample nodes. Fig. 6.3 shows how the number of nearest neighbours affects the final result. After this step, all nodes will have a list of neighbouring nodes and the corresponding distances which creates a graph.

6.2.4 Dijkstra Graph Search

The Dijkstra graph search [107] is used after the construction of the graph from the nearest neighbour search step. The final path search starts from the start location and continues until the shortest path to the destination is found. The search also terminates when there is no path found from the start location to the destination. In the Dijkstra graph search, the search starts from the start location. Then all the neighbours of the start location with their distance from the start node are kept in a list named *frontier*. In the next step, from the frontier, the closest node towards the start node is taken. If the taken node is not the destination, the neighbouring nodes of the taken nodes with their distances towards the start node are brought to the frontier. This process continues until the destination is taken from the frontier.

Fig. 6.4 shows an example of a graph. In the graph, four nearest neighbours are considered. The blue solid lines are used for connecting neighbours. The distances are noted beside the solid line. The blue dotted line shows the closer nodes that are not considered as neighbours because there is no safe flying path between them. This kind of graph is generated from the nearest neighbour step.

During the path search process, as discussed above, the min heap algorithm is used to implement the frontier. In the min heap, a tree data structure is used. In the tree data structure, a parent node can have a maximum of two children, and the parent node must be smaller than its children. The advantage of using min heap over searching the smallest value is the work complexity. The work complexity of searching the smallest value among n numbers is $O(n)$. But the work complexity of getting the smallest number from the min heap is $O(\log n)$. Fig. 6.5 shows the frontier implemented with the min heap algorithm for the graph search example shown in Fig. 6.4.

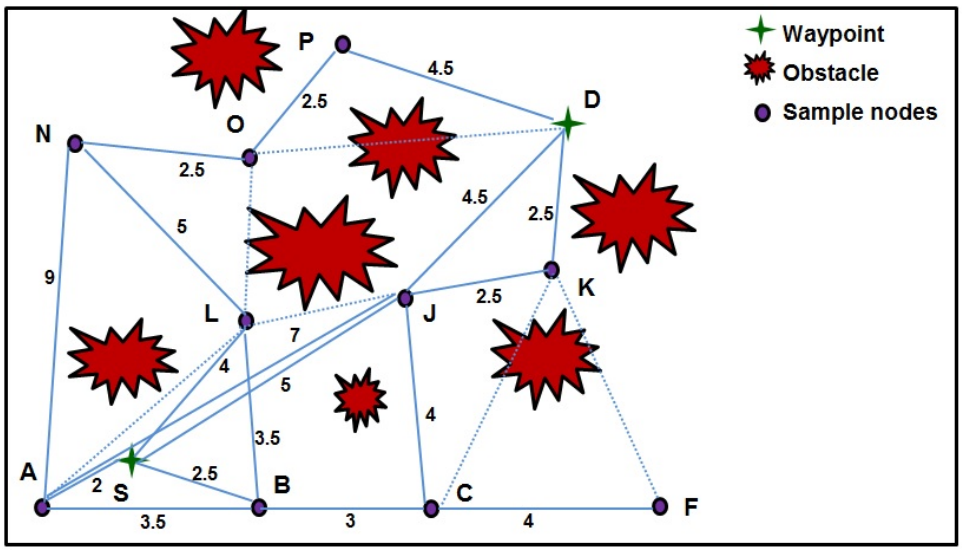


Figure 6.4: Graph for the path search

6.3 PRM: Serial Implementation

A conventional serial implementation of the PRM for a standard CPU has been realized as part of this thesis following the algorithm discussed in Section 6.2. In short, the process starts by reading the start and destination node location as well as the obstacles. Then, the sample nodes are generated within a cubic area whose two, diagonally opposing, corners are set at the start and destination nodes, respectively. To keep the number of sample nodes constant, the milestone calculation and the sample node generation steps are combined in this thesis. After generating the sample nodes, the k-nearest neighbours are calculated. The built-in approximate nearest neighbour (ANN) library [150] is used to find the nearest neighbour for each node. In the ANN library, the confined space containing the sample nodes is divided into smaller groups using a kd tree [151] and the neighbouring nodes are searched in the tree structure. The library returns the neighbouring nodes, and the Dijkstra graph search algorithm is used to find the final path.

Algorithm 5 shows the serial-PRM pseudocode. The input for the PRM is a set of sequenced waypoints, $(v_i : i = 1, 2 \ldots n)$ generated by the TSP. n is the total number of waypoints. Another input is the set of obstacles ob_j and q is the total number of obstacles.

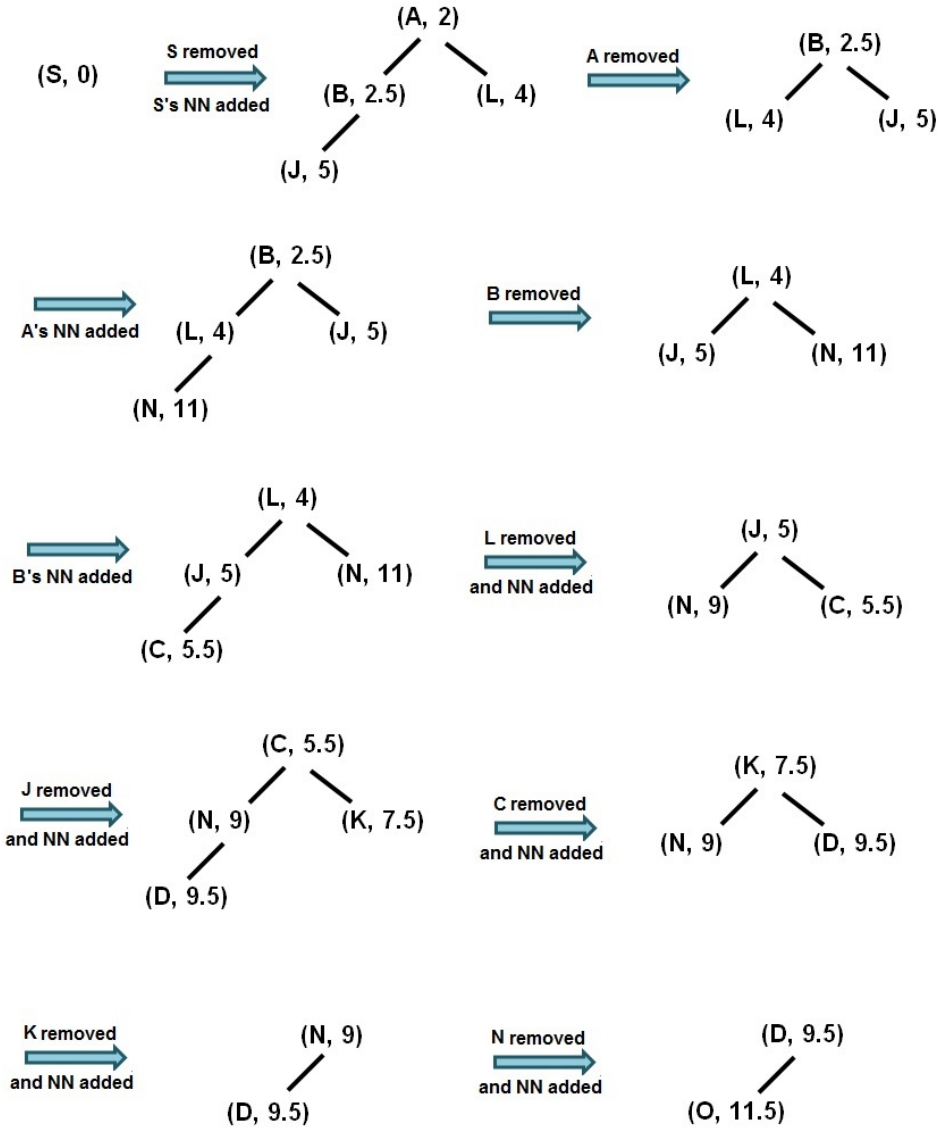


Figure 6.5: Min heap used for the graph search.

The sequential waypoints are segmented into $n-1$ pairs where the first waypoint is the start location and the second waypoint is the destination. p is the path for each segment. s denotes the sample nodes, $\{s_r\}$ is the set of sample nodes, where $r = 1, 2 \dots w$, and w is the total number of sample nodes for each segmented path where the PRM is applied. d is the distance of the segmented path, p, from the start to the destination.

Algorithm 5 The PRM

1: **function** PRM($(v_i : i = 1, 2 \dots n), \{ob_j : j = 1, 2 \dots q\}$)
2: $\{p_a : a = 1, 2 \dots (n-1)\} \leftarrow (v_i : i = 1, 2 \dots n)$;
3: **for** $l = 1$ to $(n-1)$ **do**
4: $\{s_r : r = 1, 2 \dots w\} \leftarrow$ SampleGeneration$(p_l, \{ob_j : j = 1, 2 \dots q\})$
5: $\{s_r : r = 1, 2 \dots w\} \leftarrow$ k-nn$(\{s_r : r = 1, 2 \dots w\})$
6: $((s_r : r \in [1, w]), d_l) \leftarrow$ GraphSearch$(\{s_r : r = 1, 2 \dots w\})$
7: Update $D_{matrix} \leftarrow d_l$
8: **end for**
9: **return** $D_{matrix}, \{(s_r : r \in [1, w])\}_a$
10: **end function**

Before executing the PRM in parallel, its serial execution has been analyzed closely. The time required for each block of codes are observed. It was found that 96% to 99% of the total execution time is elapsed for finding the k nearest neighbour (k-nn). The block of code calculating the sample generation and the milestone takes 1% to 3% of the total time. The final path search takes 0.1% of the total time. Reading data and the rest of the code takes less than 0.1% of the total time execution time for the serial execution.

6.4 PRM: Parallel Implementation

For faster execution of the PRM algorithm, considering the execution time of the serial code, the k-nn search and the step combining the sample generation and the milestone calculation has been implemented on the GPU. Fig. 6.6 shows the program execution pattern with time. The data regarding the waypoints and the obstacles is read by the CPU. Then, the main program in the CPU or the host code calls the GPU to calculate the sample nodes for each pair of waypoints. The GPU then returns the sample data to the CPU. In the next

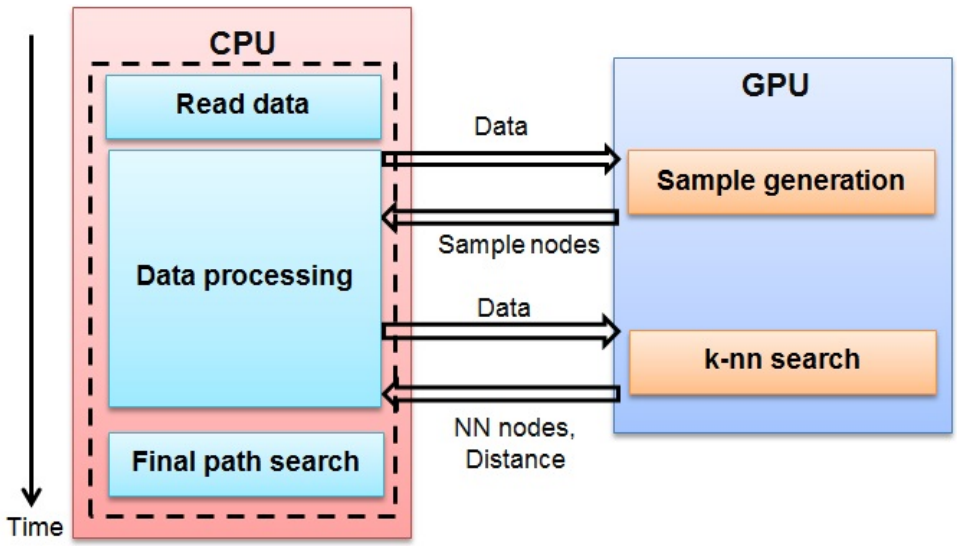


Figure 6.6: The blocks of code executed either CPU or GPU.

step, the required data is sent to GPU to calculate the k-nearest neighbour nodes and the corresponding distances. The host code in the CPU starts searching the final paths once the GPU returns the results to the CPU. The sample generation and the k-nn implementation are discussed in sections 6.4.1 and 6.4.2, respectively.

6.4.1 Parallel Sample Generation and Milestone

The sample generation in the GPU is relatively straight forward. In general, when the total number of waypoints is n, the PRM finds the $(n-1)$ paths. In each of these $(n-1)$ paths, there is a start location and a destination location. The PRM finds the intermediate waypoints to reach the destination from the start location. The sample nodes are considered as intermediate waypoints. If there are w sample nodes for each path, $w \times (n-1)$ sample nodes are generated in total for the $(n-1)$ paths in the GPU. In this process, $(n-1)$ number of GPU blocks are used. Threads in the block generate sample nodes either on the grid locations or randomly using *curand* function [152] provided by CUDA.

6.4.2 Parallel k-Nearest Neighbour

The input for this block is the obstacles and the sample nodes as well as the start and destination for all of the $(n-1)$ paths. As discussed in section 6.4.2, for the $(n-1)$ paths, there are $w \times (n-1)$ sample nodes. During the implementation of the parallel k-nn, for each sample node, a block is assigned which requires in total $w \times (n-1)$ blocks. Every w blocks work for a single path. As each block works for a single sample node, threads in the block find the distance from that single sample node to all other sample nodes $(w-1)$ from the same path. The distances with the corresponding sample node numbers are kept in shared memory. After the distance calculation, a bitonic sort is applied to find the k-nearest neighbour nodes. The description of the bitonic sort is given the Appendix A. The final results are copied to the global memory and sent back to the CPU. During the implementation, when $n-1$ gets larger than 100, the k-nn is calculated iteratively on the GPU. The k-nn for every 100 paths are calculated on the GPU at a single kernel call.

6.5 Experimental Results

The section includes the experimental results recorded from both the serial and the parallel local-path planner PRM implementation. The dimension of the considered environment is $100 \times 100 \times 50$ m^3. Three different obstacle-infused environment types are studied. Each environment contains different amounts of obstacles. In this thesis, the environments are named: En.1, En.2, and En.3. Approximately 0.35%, 2.58%, and 5.3% of the total volume is obstructed in En.1, En.2, and En.3 respectively. Fig. 6.7 shows En.2. The obstacles are modeled as spheres and shown in red.

The PRM algorithm is implemented for the three different environments as well as different numbers of waypoints. As discussed in Section 6.3, the k-nn block, and the sample generation block take 96% to 99% and 1% to 3% of the total execution time for the serial implementation. Based on this observation and according to Amdahl's law, the theoretical

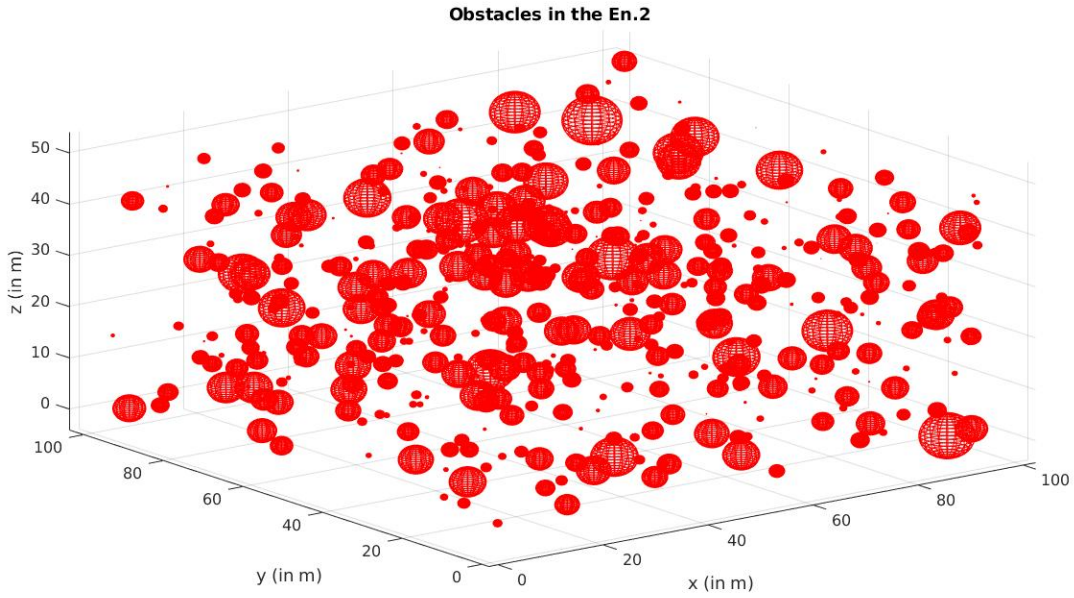


Figure 6.7: The environment, En.2 with the obstacles.

speedup will be $(1/(1-0.99)) = 100\times$.

In this thesis, both the sample generation and the k-nn blocks are implemented in parallel. The times are taken for sample generation, k-nn calculation, and the whole program for both the serial and parallel implementations are listed in Table 6.1. The corresponding speed ups for the parallel implementation with regards to sample generation, k-nn calculation, and the total execution time are also listed in Table 6.1. The unit of the listed time in the table is seconds. The number of total sample nodes is 64, and 40 sample nodes are considered as k in the k-nn calculation.

From the results listed in the Table 6.1, the execution time increases as the number of obstacles increases. Naturally, this is not surprising, but the nature of the problem is such that the parallel implementation is far less sensitive to this parameter. As an example, for 128 waypoints, the serial implementation takes 7.521 sec, 33.832 sec, and 61.161 sec in total for En.1, En.2, and En.3 respectively. On the other hand, for 128 waypoints, the parallel implementation takes, 0.205 sec, 0.309 sec, and 0.455 sec in total for En.1, En.2, and En.3 respectively. So in both cases, the execution time increases because when the number of

	Number of waypoints	Serial			Parallel			Sample speed up	k-nn speed up	Total speed up
		Sample time (sec)	k-nn time (sec)	total time (sec)	Sample time (sec)	k-nn time (sec)	total time (sec)			
En.1	2	0.002	0.074	0.077	0.055	0.007	0.063	0.03×	10×	1.21×
	10	0.008	0.536	0.548	0.063	0.014	0.083	0.13×	37.18×	6.64×
	128	0.14	7.328	7.521	0.075	0.072	0.205	1.86×	101.83×	36.7×
	256	0.215	14.716	15.035	0.09	0.133	0.337	2.4×	110.31×	44.57×
	512	0.32	29.461	29.99	0.078	0.258	0.563	4.1×	114.24×	53.31×
	1024	0.532	58.955	59.907	0.078	0.503	1.03	6.78×	117.21×	58.19×
	2500	1.323	143.99	146.34	0.118	1.21	2.389	11.19×	119×	61.25×
En.2	2	0.009	0.268	0.279	0.055	0.022	0.086	0.16×	11.97×	3.24×
	10	0.037	2.364	2.405	0.055	0.026	0.088	0.67×	90.9×	27.29×
	128	0.454	33.332	33.832	0.0707	0.185	0.309	6.42×	180.47×	109.52×
	256	0.776	66.976	67.844	0.087	0.36	0.555	8.91×	186.1×	122.28×
	512	1.266	134.23	135.68	0.071	0.704	0.988	17.85×	190.75×	137.32×
	1024	2.294	268.72	271.38	0.083	1.399	1.911	27.75×	192.05×	142.04×
	2500	5.94	656.99	663.82	0.131	3.372	4.543	45.46×	194.81×	146.13×
	4096	9.974	1082.63	1095.39	0.16	5.44	7.36	62.34×	199.01×	148.83×
En.3	2	0.018	0.487	0.508	0.056	0.015	0.074	0.33×	32.88×	6.88×
	10	0.07	4.264	4.34	0.059	0.035	0.101	1.19×	120.36×	43.19×
	128	0.981	60.141	61.161	0.076	0.326	0.455	12.97×	184.76×	134.33×
	256	1.589	121.444	123.11	0.09	0.649	0.843	17.7×	187.28×	146.05×
	512	2.562	243.466	246.18	0.081	1.27	1.56	31.78×	191.73×	157.8×
	1024	4.758	486.944	492.01	0.096	2.526	3.038	49.75×	192.8×	161.93×
	2500	12.256	1193.36	1206.36	0.151	6.18	7.356	81.18×	193.11×	164×

Table 6.1: PRM execution time for the serial and the parallel implementation.

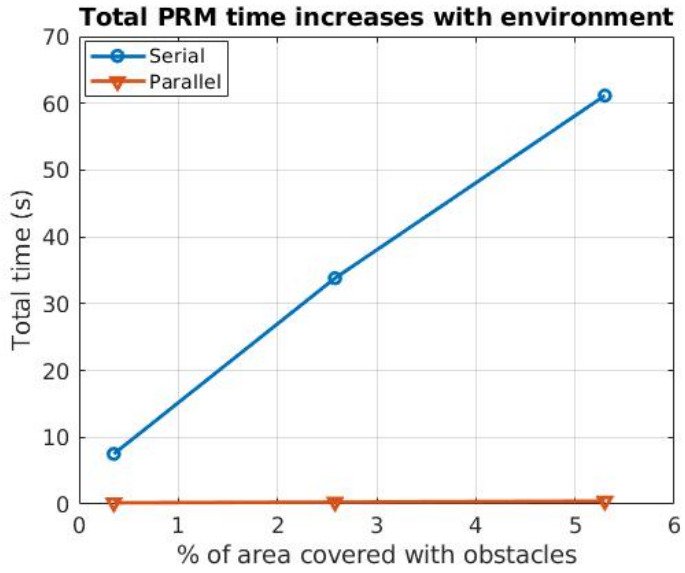
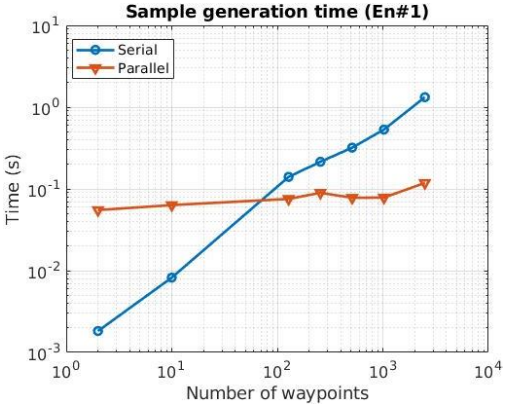


Figure 6.8: The PRM execution time changes with the increase of obstacles in the environment.

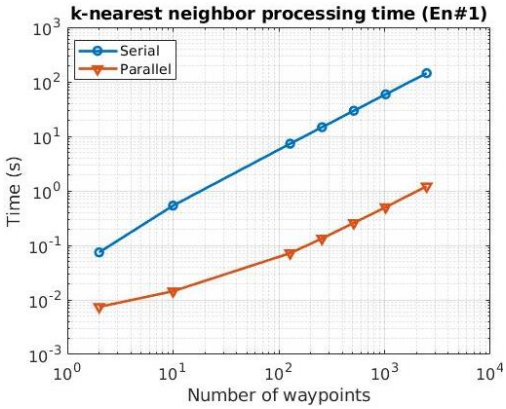
obstacles increases, the program needs to check more obstacles for path blocking. Fig. 6.8 shows how the total execution time changes with the change of the percentage of area covered by the obstacles. For the serial implementation, the time increases linearly. For the parallel implementation, the execution time increases as well but remains almost flat compared to the serial implementation.

In any environment, the execution time also increases with the increase of the number of waypoints. As the number of waypoints n increases, the PRM needs to find more $(n-1)$ paths. Fig. 6.9, 6.10, and 6.11 shows how the execution time for the sample generation, the k-nn calculation, and the total execution time for the PRM changes with the number of waypoints in both the serial and the parallel implementations for all three environments. For the sample node generation, for a limited number of waypoints (less than about 150) the serial code works faster than the GPU. The reason is that the GPU does not have enough work to utilize its maximum capability.

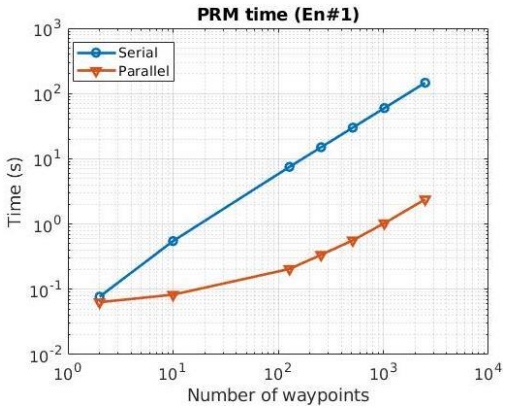
Finally, the speed-up of the parallel implementation is calculated and recorded in the last three columns of Table 6.1. The speed-up of the parallel implementation over its serial complement is more pronounced as the number of waypoints and the density of obstacles



(a) sample generation time for EN1.

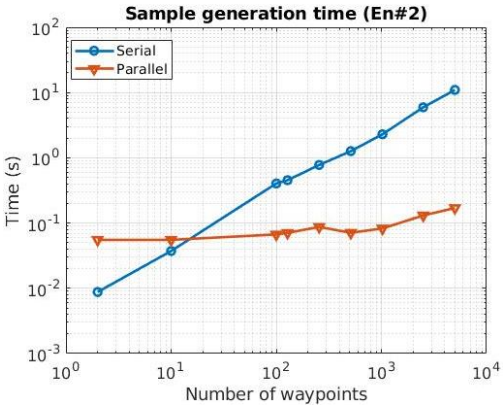


(b) k-nn time for EN1.

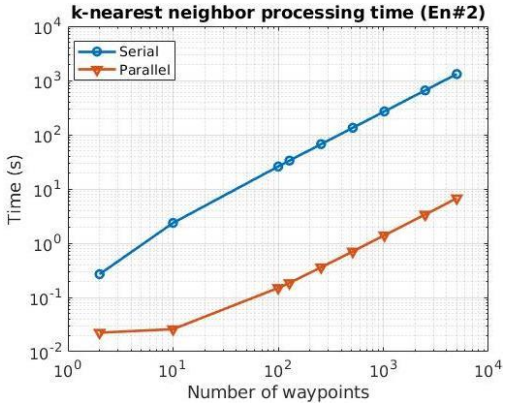


(c) The PRM time for En1.

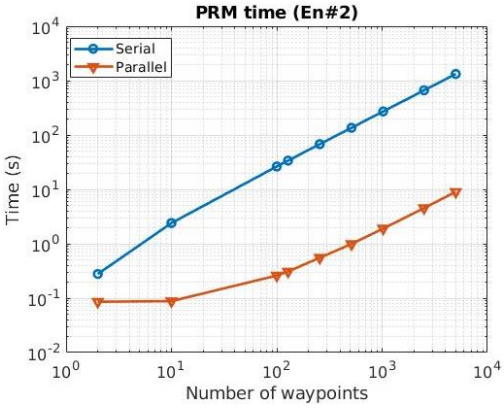
Figure 6.9: The execution time increases with the number of waypoints in En.1



(a) sample generation time for EN2.

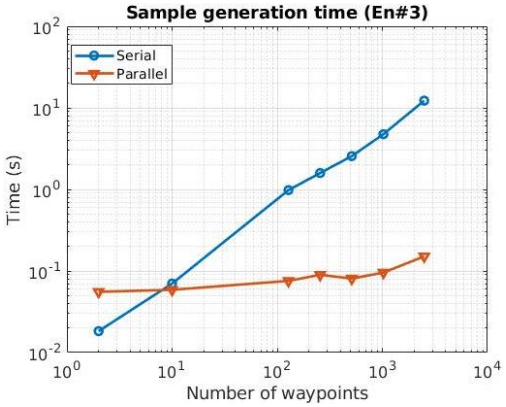


(b) k-nn time for EN2.

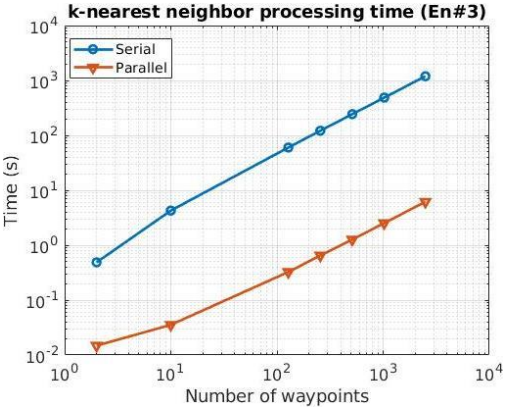


(c) The PRM time for En2.

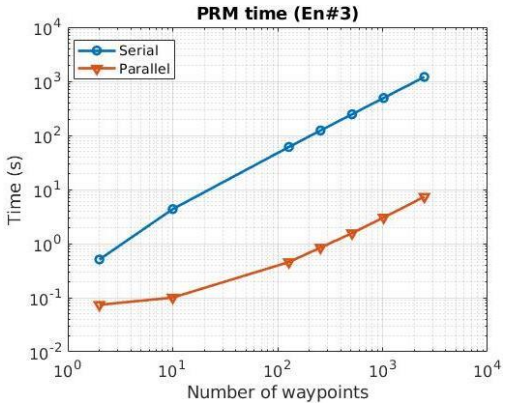
Figure 6.10: The execution time increases with the number of waypoints in En.2



(a) sample generation time for EN3.



(b) k-nn time for EN3.



(c) The PRM time for En3.

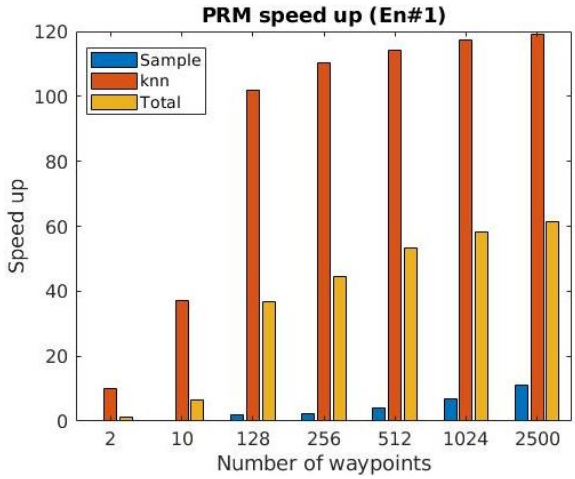
Figure 6.11: The execution time increases with the number of waypoints in En.3

increase. The maximum speed ups recorded for the sample generation, the k-nn calculation, and the total time for PRM are 81.18×, 194.81×, and 164× respectively which are close to theoretical speed ups (which was calculated 100× earlier in this section) calculated by Amdahl's law. Fig. 6.12 shows the bar plots for the speedups of the parallel implementation. From the peak of the bars in Fig. 6.12 the readers can see that there is a tremendous speed up for 2 waypoints to 128 waypoints. But for the number of waypoints higher than 128, the speed up gets linear. The reason is when the number of waypoints gets more than 100 waypoints, the parallel implementation solves every 100 waypoints on GPU iteratively. In other words, if there are 1000 waypoints, the device code is called 10 times from the host code to solve the 1000 waypoints problem.

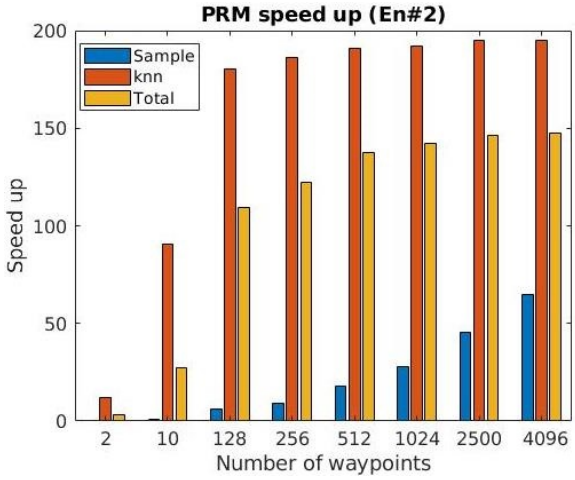
Using the parallel PRM implementation, the total execution time reduces. Table 6.2 shows the distance calculated in En.2. The results are not very different as most of the segments of the paths generated from the serial and the parallel implementation are the same. Fig. 6.13 shows an example of the paths generated from the serial and the parallel implementation. In both cases, the same set of sample nodes and the same number of nearest neighbours are used. But the results found from the two implementations are not the same. The difference between the distances calculated during the serial and parallel implementations are varied by less than 0.7% (according to Table 6.2). During the parallel implementation, due to the presence of more processors, more intermediate waypoints are considered to find a path. As the more intermediate waypoints are considered, the better results are found during the parallel implementation.

6.6 Conclusion

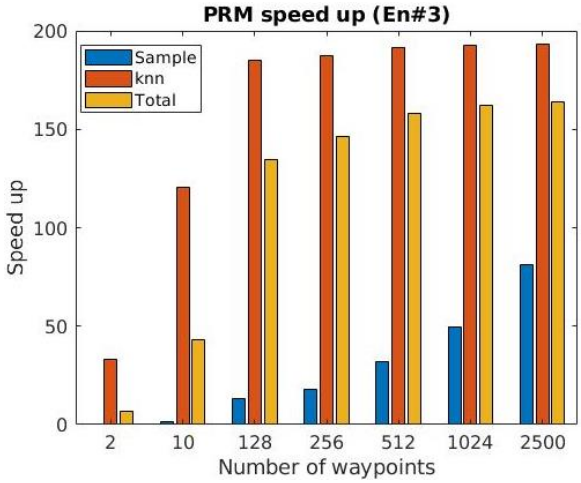
In this chapter the PRM is implemented for the proposed path planner and used to find a safe flying path between all waypoints couples provided by the global TSP-based planner. At first, the PRM is implemented in serial form on the CPU with three different environments. The density of the obstacles in these environments are different. From the results of the



(a) PRM speed up for EN.1.



(b) PRM speed up for EN.2.

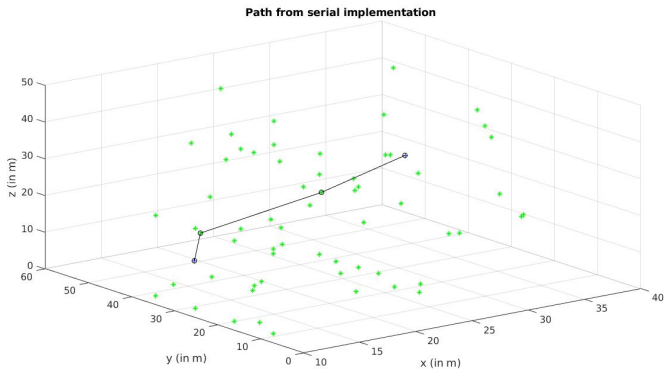


(c) PRM speed up for EN.3.

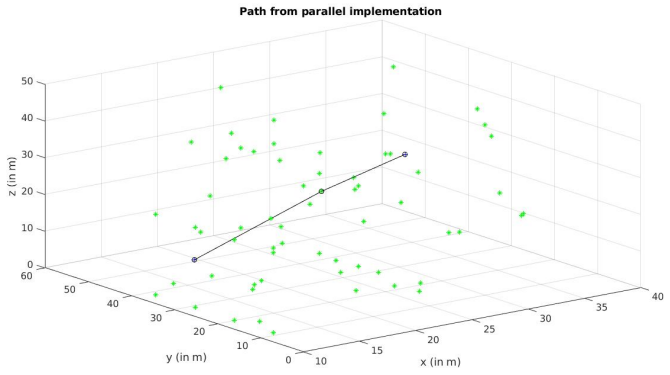
Figure 6.12: PRM speed ups in different environment.

Number of waypoints	Distance (m) (serial)	Distance (m) (parallel)
2	61.637	61.756
10	492.943	491.593
128	7173.99	7125.79
256	14090.2	14043.2
512	29168.6	29108.8
1024	59018.8	58855.2
2500	143050	142321
4096	287280	285842

Table 6.2: The quality of the parallel implementation over the serial implementation.



(a) The path from the serial implementation



(b) The path from the parallel implementation

Figure 6.13: A comparison of the paths generated from the serial and the parallel implementation using the same set of sample nodes (64) and the same number of nearest neighbour (40).

serial implementation it was found that 96% to 99% of the total program execution time is spent for the k-nn calculation and 1% to 3% time is spent for the sample generation step; that is, the milestone calculation is merged with the sample generation step in this thesis.

A parallel GPU implementation of the PRM for accelerated flying path calculation is described. Guided by the above serial execution profile, the parallel implementation is focused on the PRM algorithm's k-nn calculation block and the sample generation block. The performance of the parallel implementation relative to its serial counterpart increases as the number of waypoints and the number of environmental obstacles is increased. The total execution time falls in the range of a few milliseconds to less than 10 s. Finally, the maximum speed ups recorded for the sample generation, the k-nn calculation, and the total time for the PRM are 81.18×, 194.81×, and 164× respectively.

Chapter 7

Sequence Analysis: DNA Sequencing

This chapter extends the thesis investigation into parallel-accelerated sequential problems by considering DNA sequencing. A miniaturized sequencing process is discussed that can sense and analyze DNA in a continuous fashion at the rate of 7242-base pairs per second. A GPU-enhanced sequence analyzer capable of operating in a streaming fashion, at-rate with the DNA sensor is discussed. An overview of DNA sequencing, an explanation of the GPU accelerator, and the presentation of the experimental results are given in this chapter.

7.1 DNA Sequencing: an Overview

As already discussed in Chapter 1, DNA is a biological macromolecule present in the cells of all known cellular organisms. A segment of this molecule is illustrated in Fig. 7.1, the double-helix famously identified by Watson and Crick [153]. The ladder-like structure of the DNA molecule is distinguished by its twisted outer support, made of phosphate and sugar molecules, and ladder "rungs" corresponding to a sequence of smaller molecules called *bases*. In fact, each rung is made out of two bases, a *base pair* (bp), bonded to one another along the central axis of the helix. There are only four different types of base molecules, each typically referred to by the first letter of its chemical label, namely, `A`, `C`, `G`, `T`. Although any sequence of bases can make up a DNA molecule, the bases pair according to complementary

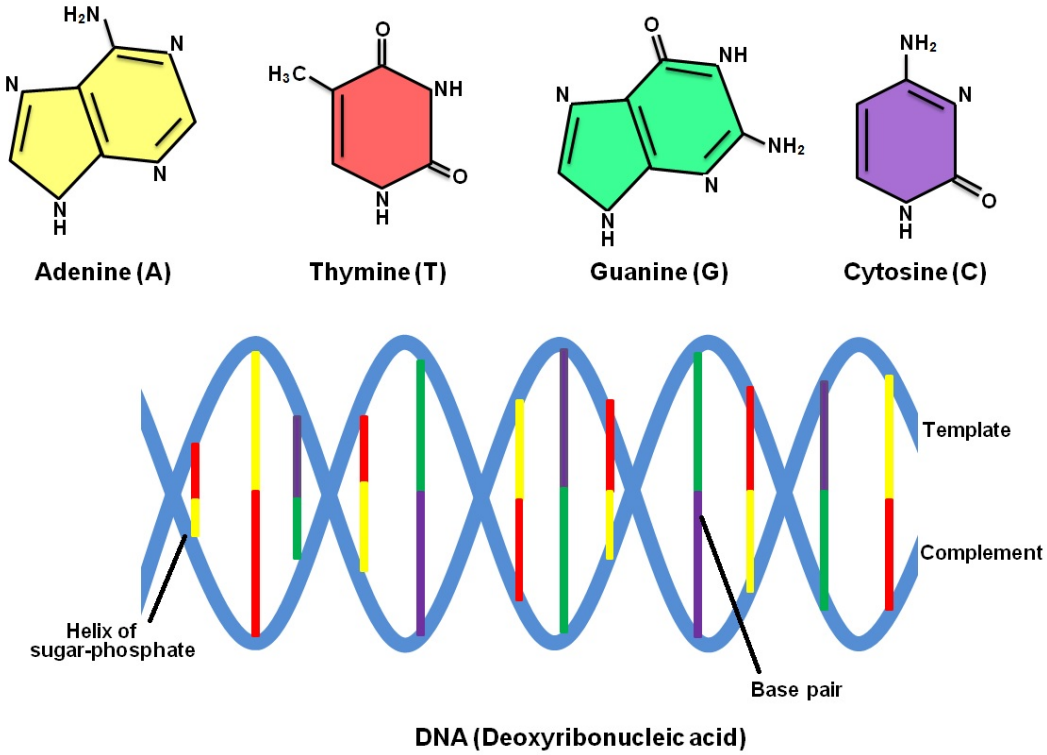


Figure 7.1: A schematic diagram of DNA and nucleobases of DNA : Adenine (A), Thymine (T), Guanine (G), and Cytosine (C) [1].

bonding rules such that, unless otherwise damaged, DNA base pairs consist of A-T couples or C-G couples.

DNA is commonly referred to as the "blueprint of life" because it is used by other *transcription/translation* mechanisms within the cell to generate complex molecules (proteins) needed to execute the biochemical functions necessary for life. The type of proteins coded by DNA (by certain intervals within the molecule called *genes*), as well as a number of other functions, are fundamentally defined by the sequence of bases that comprise the DNA polymer. Identifying the sequence of bases that comprise a DNA molecule defines the goal of the *sequencing process*. More generally, our ability to sequence the whole of an organism's intrinsic DNA identifies its *genome* and thus results in a record of its entire set of genes. Of course, extracting actionable information (e.g., susceptibility to disease) from such a record requires substantial scientific insights and research from fields such as *molecular and popu-*

lation genetics. These latter issues are not the concern of this thesis, indeed only an early, but computationally intensive part of the so-called *sequencing pipeline* is the focus of this work, this step is called *basecalling*. Keeping this in mind, and for simplicity, I will occasionally refer to any component of the sequencing pipeline (e.g., chemistry, measurement, signal processing, basecalling, bioinformatics, etc.) as simply "sequencing" although it should be clear that the process itself consists of many steps.

Basecalling is the step of translating signals from sensors operating on DNA molecules into a sequence of text labels (i.e. from the set {`A`, `C`, `G`, `T`}) identifying the measured DNA's base sequence. The reader should appreciate that current technological limits prevent sequencers from handling entire genomes without interruption. In practice, these molecules need to be randomly partitioned into much smaller "samples" before they can be robustly measured in a sequencing machine. For example, it is very common to randomly partition the chromosomes of the human genome (i.e., a continuous segment of DNA roughly 150 million bp long) into minute segments of only 100 bp. It is the job of the basecaller to label such measured segments. Ensuing processes then correct and re-assemble these samples back into complete genomes.

In the vast majority of today's sequencing technologies, optical sensing techniques are used, and the basecaller is then left to sift through a sequence of recorded light pulses in order to identify (i.e., sequence) the DNA from which such signals originated. In March 2014 a new sequencing technology (based on decades of research) entered the market with profound implications for the field [154]. This is a method wherein DNA is threaded through a very small molecular opening, a sensor called a *nanopore*, resulting in the generation of an electrochemical time-series signature of the DNA that is converted into an electronic signal directly processed by microelectronic components. This method arguably sets the stage for the next major advance in sequencing technologies (there have been at least three such innovations in the last 40 years [155]).

A number of benefits accrue due to this nanopore-based sequencing advance. Most ob-

Figure 7.2: MinION: nanopore-based sequencing device [2].

vious is the physical scales achievable by such a device (an example is pictured in Fig. 7.2, a hand-sized machine weighing 100 grams) which are orders of magnitude smaller than the mainframe or desktop-sized sequencing units established in the market today.

This size reduction is largely due to the implementation of an electronic sensing modality in place of an optical one. The ability to translate DNA directly into electronic time-series makes these signals amenable to direct handling by semiconductor microelectronics components. With integrated circuits present very close to the sensing site, significant downsizing is achieved with the promise of substantial computing resources traditionally associated with microelectronic chips.

Another major benefit of the nanopore-based approach is the dynamic nature of its measurement process. Whereas arduous laboratory methods are needed to load and measure DNA samples in legacy sequencers, the nanopore-based approach allows continuous loading and sequencing of molecules. In effect, the sequencing process has been turned into a real-time sensing technology.

The attributes listed above, coupled with other advantages, make nanopore-based sequencing highly promising for a multitude of applications detailed elsewhere [156, 157, 158]. However, new engineering challenges are also presented by this technology and one of these,

namely basecalling, is the main concern of this chapter. Specifically, the nanopore-based sequencers can achieve their "raw" sensor measurements extremely economically. Quantitatively, for roughly 1 W of power consumption, a hand-held sequencer can operate over 512 simultaneous DNA measurement channels processing nearly 500 bp per second (bp/s) and offering their signals for downstream analysis such as basecalling. This throughput is roughly equivalent to 1 entire human genome every 3.5 hours. Keeping in mind that this rate is already a 10× improvement over this sequencer's performance levels only three years ago and that at least a 100× increase in sensor speed is theoretically possible, the potential for this technology is immense.

But as efficient as this data gathering is, it happens that its raw measured signal has a relatively low signal-to-noise ratio (SNR) compared to established sequencing methods. This presents a substantial statistical signal processing challenge that is first encountered by the basecalling step of the broader sequencing pipeline. Numerous algorithms exist to deal with such an SNR problem, but they all require intense computing resources. Thus, one is faced with an incongruity, a small portable sequencer capable of gathering real-time data within a 1-W power budget, but requiring follow-up computation approaching 1000 W to keep up. This scenario exactly parallels the UAV discussions explored earlier, and once again I consider the utility of GPUs for dealing with it; in this case the application of GPUs for basecalling. I detail this approach next.

7.2 Problem Definition

As outlined above, basecalling is essentially a sequence labeling problem: a time-series is converted to a sequence of letters from the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. The aim of this research is to make the conversion faster and more power efficient to enable embedded operations. The serial execution of basecalling is discussed in Section 7.3. Section 7.4 discusses two different methods used for the parallel execution of basecalling where the first method executes a

single file at any instance and the second method executes multiple files at any instance.

7.3 Serial Basecalling

A well established sequence detection method, the *hidden Markov model* (HMM), is used for the basecalling calculation [159] in this thesis. In general, this method seeks to identify the "hidden states" (of some hypothesized state-space) that a system under study traverses by observing only a noisy signal probabilistically related to the underlying hidden states. In the case of basecalling, the hidden states essentially denote the string, or "sub-sequence" of DNA bases (a so-called k-mer, a string of k bases, depending on the nanopore sensor realistic values of k may vary between about 3 to 6) in the nanopore sensor at any one time. Hence, if the states are identified from the observations so are these base k-mer strings, and consequently so are the bases comprising these k-mers. Thus, the process of basecalling is achieved.

In a *Markov model*, the states of a system under study, are directly visible to the observer, and therefore the state transition probabilities, a model of the chances that one state transitions into some other state of the system during some instant, are the only parameters required for the probability calculation. But, as noted, in the HMM formalism, the state occupied by the system under study is not directly visible, it may only be inferred from the noisy observations/outputs. Each state has a probability distribution over the possible output. Therefore, the sequence of outputs generated by an HMM gives some information about the sequence of underlying (hidden) states.

In this research study, the measured data from the sensor is the observed output and the hidden states are the aforementioned k-mer sub-sequences ($k = 6$ in this work), assumed to contribute to the measured signal at any observation instant. That is, the nanopore is not precise enough to produce a signal proportional to a single base, but rather, according to some models, a string of $k = 6$ bases. Given that there are four possible bases, the number

of states accounting for all possible k-mers is 4^k.

In this thesis, the HMM is solved in a dynamic programming fashion using the *Viterbi algorithm (VA)* [160]. Further discussion on the HMM and the Viterbi algorithm is in Sections 7.3.1 and 7.3.2.

7.3.1 HMM-Based Basecalling

A succinct expression of the basecaller's underlying detection strategy is

$$\hat{\boldsymbol{s}} = \arg\max_{\boldsymbol{s}} f(\boldsymbol{s}, \boldsymbol{e}) \tag{7.1}$$

where $\boldsymbol{s}$ and $\boldsymbol{e}$ denote the sequence of states and N observed *events* respectively. The *read length* N depends on the DNA length (the *strand length*, L), in terms of bases, of the molecular sample that translocates through the nanopore sensor. Ideally the sensor's observation sequence $\boldsymbol{e}$ will correspond to the actual DNA base sequence in a one-to-one manner making $N = L$. However, as discussed in Section 7.3.1 complications arise in practice resulting in $N \approx L$. The value of L is randomly distributed around a mean dependent on how the DNA samples to be sequenced are prepared. A typical experiment [161] may achieve median $L \approx 5$ kbp with some lengths easily exceeding 50 kbp.

In a *maximum a posteriori* (MAP) sense, the basecaller constructs estimates, $\hat{\boldsymbol{s}}$, of the underlying state sequence $\boldsymbol{s}$ by considering the joint probability density model (pdf), $f(\boldsymbol{s}, \boldsymbol{e})$.

Rephrasing (7.1) via Bayes's formula gives

$$\hat{\boldsymbol{s}} = \arg\max_{\boldsymbol{s}} f(\boldsymbol{e}|\boldsymbol{s}) P(\boldsymbol{s}) \tag{7.2}$$

where the *likelihood* $f(\boldsymbol{e}|\boldsymbol{s})$ models the dependence of event observation sequences on underlying nanopore state sequences with *prior*, $P(\boldsymbol{s})$.

Assuming a Markovian state progression property, the basecalling mechanism can be

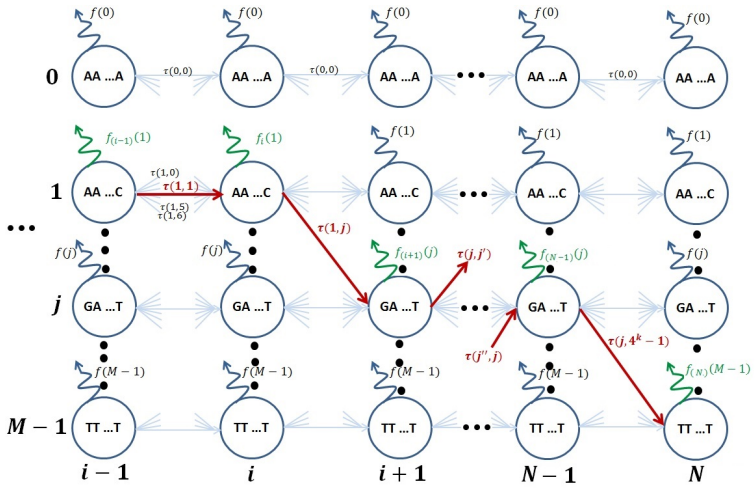


Figure 7.3: The HMM expressing the translocation of a molecule through a nanopore sensor.

further distilled to the product series

$$f(\boldsymbol{e}|\boldsymbol{s})P(\boldsymbol{s}) = \prod_{i=1}^{N} f(e_i|s_i) \prod_{i=2}^{N} \tau(s_{i-1}, s_i) P(s_1) \tag{7.3}$$
$$= \prod_{i=1}^{N} v_i(e_i, s_i)$$

where $f(e_i|s_i)$ denotes the *emission probability*: the pdf of an event e_i in response to a nanopore state s_i and where $\tau(s_{i-1}, s_i)$ denotes the *transition probability*: the probability of s_{i-1} transitioning to s_i between consecutive observations. The sequence posterior at each index i is denoted with $v_i(e_i, s_i)$.

The decision metric summarized by (7.3) assumes the familiar HMM approach and can be pictured in terms of the trellis diagram sketched in Fig. 7.3. As illustrated, the trellis nodes represent all the possible $M = 4^k$ states that the nanopore can assume at any particular measured event e_i. The edges between states in the trellis denote the transition probabilities; the undulating vectors emerging from each node denote the emission probabilities. Fig. 7.3 employs a more compact notation; states, s_i, are denoted with labels $j, j' \in \{0, \ldots, M-1\}$ and the emission probability is denoted with $f_i(j) = f(e_i|s_i)$. Similarly, the posterior can more succinctly be denoted with $v_i(j) = v(e_i, s_i)$.

In the context of the trellis, and as implied by (7.2), the goal of basecalling is to find the optimum sequence of states, an optimum path, through the trellis (i.e., highlighted path in Fig. 7.3). This effort requires a more detailed consideration of the HMM's component parts, the transition and emission characteristics and I turn to these now.

Transitions

Ideal nanopore sequencing operation would imply that only one of four transitions are possible from s_{i-1} to s_i. This ideal scenario is based on the anticipation that each new event e_i corresponds to the entry of a new base $b_i \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ into the sensor and hence a transition into state $s_i = b_{i-k+1} \dots b_i$ from state $s_{i-1} = b_{i-k} \dots b_{i-1}$. Such *step* transitions should be the most common transition-type in a well designed system, but, practically-speaking, they cannot be assumed to be the only one.

In alignment with the transition mechanisms outlined in [38], two more transition types should also be considered: 1.) the possibility that a new event observation e_i corresponds to the previously determined state (a *stay* transition) where $s'_i = b_{i-k} \dots b_{i-1}$; 2.) the possibility that a base observation was missed and hence that e_i corresponds to a state s_i which was preceded by $s'_{i-1} = b_{i-k-1} \dots b_{i-2}$ (a *skip* transition).

An example of the transition types outlined above is illustrated in Fig. 7.4 for changes from a state $s_{i-1} = \mathtt{AAAAAC}$. As indicated, accounting for step, stay, and skip mechanisms results in 21 possible transitions per state. I denote the labels of states that transition to a state j with ν belonging to the set of such states $\boldsymbol{\omega}(j)$.

Emissions

While the transitions $\tau(j, j')$ quantify state progression through the trellis, the emissions, $f_i(j)$ allow the basecaller to measure the association between measurements and those states. Depending on the measurements acquired, a variety of associations can be made.

Among these, the measurement process can extract events from noisy data (e.g. as shown

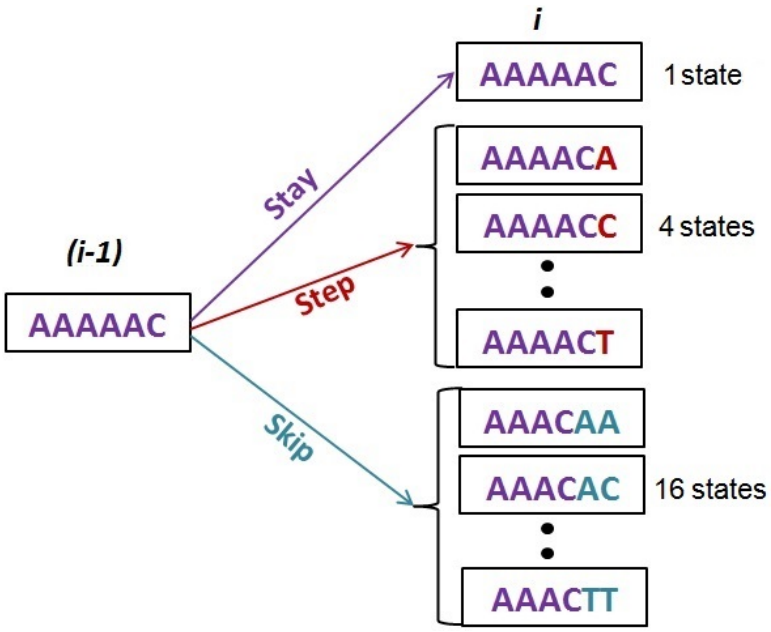


Figure 7.4: One model of possible transitions from one state to another. Stays, steps, and single base skips are the transitions accounted for. In this model a total of 21 transitions form the state at i are possible.

in Fig. 7.5) in terms of the pair $e_i = (x_i, y_i)$ where x_i is the mean event level and y_i is its standard deviation. Treating these components as independent metrics, one can associate them with an underlying sensor model via appropriate pdfs. Following [38] I consider

$$f_i(j) = \mathcal{N}(x_i; \mu_j, \sigma_j) \cdot \mathrm{IG}(y_i; \eta_j, \lambda_j) \tag{7.4}$$

where $\mathcal{N}$ denotes the Guassian (normal) distribution and IG denotes the inverse-Gaussian distribution and μ_j, σ_j, η_j, and λ_j represent nanopore model settings. Specifically, these settings parameterize the pdfs

$$\mathcal{N}(x_i; \mu_j, \sigma_j) = \frac{1}{\sqrt{2\pi\sigma_j^2}} e^{-(x_i-\mu_j)^2/2\sigma_j^2} \tag{7.5}$$

and

$$\mathrm{IG}(y_i; \eta_j, \lambda_j) = \left[\frac{\lambda_j}{2\pi y_i^3}\right]^{1/2} e^{-\lambda_j(y_i-\eta_j)^2/2\eta_j^2 y_i} \tag{7.6}$$

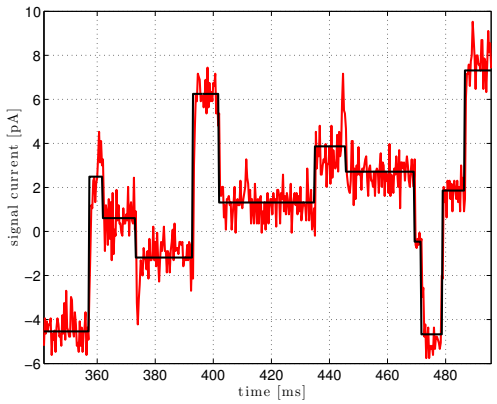


Figure 7.5: Nanopore-based electronic current (around a mean) generated by the movement of DNA through a nanopore sensor. Shown is the raw (noisy) time-series and a piecewise-constant approximation extracted from it [3].

7.3.2 Viterbi Algorithm for Basecalling

The transition and emission components of the HMM as discussed in Sections 7.3.1 and 7.3.1 complete (7.3) and hence the basecaller's decision criterion (7.2). In its most general sense, this calculation calls for the identification of $\hat{\boldsymbol{s}}$ through the analysis of all possible paths of length N through the trellis, a computationally daunting problem given the scales at play. However, this process is amenable to solution via the Viterbi algorithm (VA).

The VA, as appropriate for the basecaller's state identification goal, is summarized in Algorithm 6.

Algorithm 6 Viterbi Serial Base Caller Calculation

Initialize:
$v_0(j) \leftarrow f_0(j) \;\forall\; j \in \{0, \ldots, M-1\}$
Iterate: ▷ Update posterior and traceback $\forall\; j$
for $i \leftarrow 1, N-1 \;\;\&\;\; \forall\; j$ **do**
    $v_i(j) \leftarrow f_i(j) \max_{\nu \in \boldsymbol{\omega}(j)} [v_{i-1}(\nu)\tau(\nu, j)]$
    $\text{ptr}_i(j) \leftarrow \arg\max_{\nu \in \boldsymbol{\omega}(j)} [v_{i-1}(\nu)\tau(\nu, j)]$
end for
Optimal End State:
$\hat{s}_{n-1} \leftarrow \arg\max_j [v_j(n-1)]$
Traceback:
for $i \leftarrow n-1, 1$ **do**
    $\hat{s}_{i-1} \leftarrow \text{ptr}_i(\hat{s}_i)$
end for

The VA basecaller computes M posteriors $\{v_i(j)\}$ over N event indexes (line 5 of Algorithm 6). Generally speaking, the VA's iterations occur over the event index i. At each increment of i, the posterior probability of all M states is updated. This running posterior is $v_i(j)$, the probability of state j having occurred at i (given the event e_i). This update (see line 5 of Algorithm 6) is

$$v_j(i) \leftarrow f_j(e_i) \max_{\nu \in \boldsymbol{\omega}(j)} [v_\nu(e_{i-1})\tau(\nu, j)]$$

This line considers all transitions from states ν into the state j weighted by the transition's probability to that state, $\tau(\nu, j)$. The product of $\tau(\nu, j)$ and the accumulated sequential posterior probability, $v_{i-1}(\nu)$, of its origin state (i.e. the origin state ν at index $i-1$) is proportional to the probability of transitioning to state j at index i. Note that among $|\boldsymbol{\omega}(j)|$ products $v_{i-1}(\nu)\tau(\nu, j)$ just one maximum is chosen. This maximum is the key in ultimately extracting the most likely path, $\hat{\boldsymbol{s}}$. To complete the net likelihood of the sequence to state j, the maximum product is multiplied by the posterior of that state (alone); that is, one multiplies by $f_i(j)$.

Line 5 only enumerates the maximal sequence state probabilities. The solution algorithm also has to keep track of which states ν at $i-1$ connect to states j at i. Line 6 of Algorithm 6,

$$\mathrm{ptr}_i(j) \leftarrow \arg\max_{\nu \in \boldsymbol{\omega}(j)} [v_{i-1}(\nu)\tau(\nu, j)],$$

is responsible for this. This line assigns to the pointer $\mathrm{ptr}_i(j)$ the state ν that maximizes the product $v_{i-1}(\nu)\tau(\nu, j)$ for all $\nu \in \boldsymbol{\omega}(j)$.

After the VA's iterative procedure completes, the terminal state of the most likely path is identified by choosing the state j corresponding to the maximum value of the sequential probability. This is calculated by line 9 in Algorithm 6, that is:

$$\hat{s}_{i-1} \leftarrow \arg\max_j [v_j(n-1)].$$

Finally, the rest of the states of $\hat{\boldsymbol{s}}$, from $\hat{s}_{n-2}$ to $\hat{s}_0$, making up the most likely sequence are recovered by applying a traceback procedure relying on the pointers accumulated during the iteration phase.

The core implementation outlined above is suitable for offline post processing of sequencer measurements. Given the nascent status of market-ready nanopore sequencers and the arrested clock frequency scaling demonstrated by commodity microprocessor technology, however, it is clear that this approach is not suitable for the massive real-time analysis requirements of nanopore sequencers.

7.4 Parallel Basecaller

This research study now proposes means by which the VA nanopore basecaller may be implemented in a parallel processing architecture that can enhance the computational throughput and allow the computational system to keep pace with advances in the sequencer's measurement technology. The GPU's parallel architecture provides direct opportunities to achieve this [162].

In the VA nanopore basecaller algorithm, there are a number of components that can be executed independently. Identifying these elements and porting their execution to the GPU in an efficient manner affords immediate and substantial improvements in throughput. The acceleration of these algorithm aspects and the complications imposed by their translation into GPU procedures are elaborated upon in the following sections.

At first one file, containing the measured output of a DNA sample, is executed in parallel which is discussed in Section 7.4.1. In the next Section 7.4.2, multiple files (each file containing the measured output of a DNA sample) are executed in parallel.

7.4.1 Parallel Basecaller: Single File

The first step in porting to a parallel implementation involves identifying the algorithm components that can be executed independently. So, a re-phrasing of the VA (i.e. line numbers 4 to 7 of Algorithm 6) is presented in Algorithm 7.

Algorithm 7 A Part of Viterbi Serial Basecaller Calculation

```
1: for strand ← 0, 1 do
2:     for model ← 0, 1 do
3:         for i ← 1, N − 1 do
4:             for j ← 0, M − 1 do
5:                 v_i(j) ← max_ν[v_{i−1}(ν)τ(ν, j)]
6:                 v_i(j) ← f_i(j)v_i(j)
7:                 ptr_i(j) ← arg max_{ν∈ω(j)}[v_{i−1}(ν)τ(ν, j)]
8:             end for
9:         end for
10:    end for
11: end for
```

Before embarking on a parallel implementation of this code it is clear that a logarithmic implementation of the update function will reduce the computational burden. That is, the implementation

$$\ln v_i(j) \leftarrow \ln f_i(j) + \max_{\nu \in \boldsymbol{\omega}(j)} [\ln v_{i-1}(\nu) + \ln \tau(\nu, j)] \tag{7.7}$$

will substitute additions for costly multiplications. Although the multiplicative exposition is retained, throughout the remainder of the chapter, for consistency with the VA as outlined initially, it is to be understood that the logarithmic equivalent of the posterior is computed.

As noted earlier, the total number of states $M = 4096$ for a 6-mer nanopore sensor model. This makes the posterior calculation iterate M times in Algorithm 7. Again, the total number of events N (the read length), represents the number of samples generated from the sequencer per read of a DNA. Given that events are observations of bases as the DNA translocates through the sensor, they (the events) are approximately (accounting for stays and skips) equal to the length L of the DNA (in terms of bases).

The average number of events per read $\bar{N}$ is approximately 5k with the possibility of

some reads exceeding 10× or 20× this value. For all these events, the posterior calculation iterates N times in line 3 in Algorithm 7.

As seen in Algorithm 7, there are two more outer loops (in lines 1 and 2) present for the posterior calculation. The outer loop at 1 iterates either one or two times depending the input file. This *strand* variable refers to the two strands of a DNA molecule: complement and template strand (as shown in Fig. 7.1). In most event input files handled by the basecaller, the measured data from both strands are given (i.e., the nanopore manages to unfurl the DNA and measure each of its two strands in sequence). Also, in the input file, one or two trained data set or models are given for each strand which explains the remaining loop in line 2.

Considering $L = 5000$, $M = 4096$, two strands, and two models per strand, the total number of serial iterations becomes $5000 \times 4096 \times 2 \times 2 \approx 82$ million. Roughly, the total number of basic operations per event (M states per event are included) is 1,523,714 OP [33]. The basic operations include addition, subtraction, division, etc. Thus, for a single file with $L = 5000$ the total number of operations will be $5000 \times 2 \times 2 \times 1{,}523{,}714 \approx 3 \times 10^{10}$ OP.

Now, considering a miniature sequencer working with a channel rate of 250-bp/s and 250 operational channels, the total number of operation per second becomes $1{,}523{,}714 \times 250 \times 250 = 9.5 \times 10^{10}$ OPS ≈ 95 GOPS. To estimate the execution time, one may consider a Core i7 4820K (based on a 22-nm Ivy Bridge-E) running overclocked at 3.9 GHz which achieves about 12×10^9 DIPS (Dhrystone instructions-per-second) on Dhrystone 2.1 [163]. Assuming 2 OPS = 1 DIPS [33] these approximations predict that a Core i7 4820K requires at least 3.97 sec (excluding the read/write, scaling operations) to complete the computation tasks for the data generated by the sequencer in one second.

This speed disparity strongly suggests the importance of using parallel computational architecture for basecalling. Indeed, aspects of this computational load can be handled quite adeptly by a massively parallel device such as a GPU. Mapping the problem onto the GPU and efficiently managing the workload is now discussed.

In this section, a parallel basecaller is described that solves a single measured DNA file at-a-time. Line 3 to line 9 from Algorithm 7 are executed on the GPU with multiple kernel calls. In each kernel call $q = 32$ events are calculated on the GPU and all the threads calculate the probability, $v_i(j)$ in parallel for different states $j \in \{0, \ldots, M-1\}$ of the same i-th event e_i. Once the calculations for the ith event are complete, ensuing threads start the calculation for $(i+1)$th event. Threads cannot be assigned to calculating the probabilities for the ith and $(i+1)$th events in parallel because the probabilities of the $(i+1)$th event depend on the probabilities of ith event. Global memory is used to store the calculated probabilities, $v_i(j)$ so that $v_i(j)$s are accessible during the probability calculation of $(i+1)$th event.

A detailed discussion on how the threads are distributed in the blocks is included later in this section. Events e_i consist of two components: the mean x_i, and standard deviation y_i; for a given event index i. At any time instant, threads in the GPU work in parallel calculating different states probabilities for the same event. During these calculation, threads use the same mean and standard deviation values (as each event has a unique value of mean and standard deviation). As a result, for parallel implementation, the GPU's constant memory block is the best option for retaining these event components.

The variable $\boldsymbol{\omega}(j)$, contains the set of transition probabilities denoting the chance of moving from one state from to another state. The pore model data (i.e. μ_j, σ_j, λ_j, and η_j as present in (7.5) and (7.6)) is used for the emission probability calculation. This data structure contains M elements, and each element has multiple variables. Global memory is used for keeping both $\boldsymbol{\omega}(j)$ and the pore model data in this design.

Given the above considerations, the code is implemented and executed in parallel on a GPU using CUDA C++ [164]. Fig. 7.6 shows the CPU and GPU threads executing host and device codes respectively with time. The pseudo code is provided in Algorithm 8 where q is the number of events calculated in a single kernel call. In this experiment $q = 32$, as the traceback step (line 10 of Algorithm 6) is applied every 32 events. In the device code,

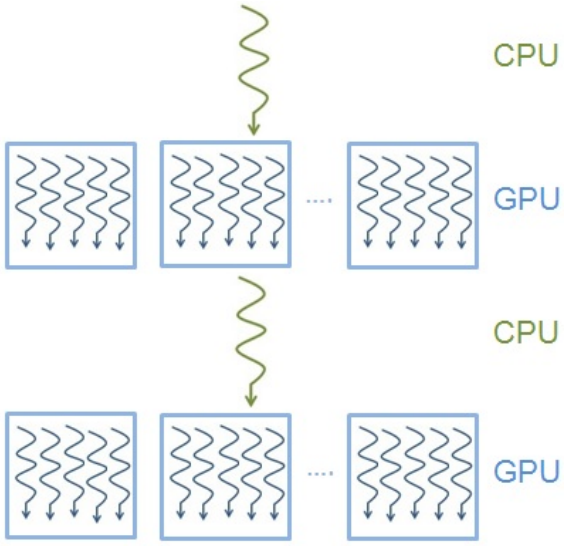


Figure 7.6: CPU and GPU threads working within a single-file basecaller design.

Algorithm 8 Viterbi Parallel Basecaller Calculation

procedure DEVICE CODE
    $v_i(id) \leftarrow \max_\nu [v_{i-1}(\nu)\tau(\nu, id)]$
    $v_i(id) \leftarrow f_i(id)v_i(id)$
    $\text{ptr}_i(id) \leftarrow \arg\max_{\nu \in \boldsymbol{\omega}(j)} [v_{i-1}(\nu)\tau(\nu, id)]$
end procedure
procedure HOST CODE
    for $strand \leftarrow 0, 1$ **do**
        for $model \leftarrow 0, 1$ **do**
            for $i \leftarrow 1, N-1$ for every q **do**
                Call Device code
            end for
        end for
    end for
end procedure

id refers to the global thread number which is expressed in (7.8) for 1D blocks.

$$id = \text{threads per block} \times \text{block.Idx} + \text{thread.Idx} \tag{7.8}$$

For each event calculation, the highest emission probability $f_{(i-1)}(j)$ from the previous event is required during line 3 in Algorithm 8. Finding the highest value of a data set on a GPU is challenging when the data set is distributed in several blocks. In such case, the easiest implementation involves calculating all the states (4096 states) of one event in a single kernel call, transferring the results to the CPU and then finding the maximum value on the CPU itself. This scenario would be repeated for each event. The main drawback for such an implementation is clearly the large memory traffic between the CPU and the GPU compared to the GPU workload.

To solve the problem, instead of one event, multiple events are calculated in a kernel call, and line 3 is implemented on the GPU which requires the highest emission probability from the previous event. At the beginning, I tried 4096 threads distributed in multiple blocks works for each state. But threads from different block cannot be synchronized, i.e., one thread will wait for another thread. Again using the lock option like mutex, makes the code serialize. In general, in parallel implementation when multiple threads wants to access the same variable, a race condition arises. Mutex is used to prevent such race condition. Using mutex only one thread can access to the variable at a single instant and rest of the threads wait to access to the variable.

Ultimately the problem was handled by using 1024 threads from one block to 4096 states where each thread is responsible for calculating four states. For example, Thread# 0 calculates $f_i(0)$, $f_i(1024)$, $f_i(2048)$, and $f_i(3072)$ first, then the thread finds the maximum value among these four values and copies its local maximum value into the shared memory. Up to this point, a thread works independently. Globally, threads wait until all the threads copy their local maxima to shared memory (i.e. the GPU forces the threads to synchronize) and

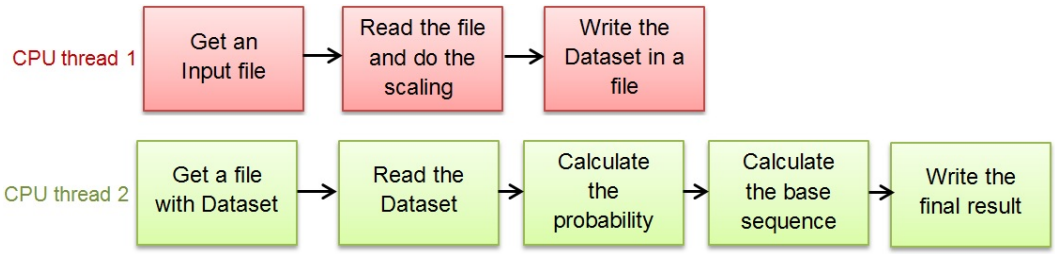


Figure 7.7: Block diagram for multi file basecaller.

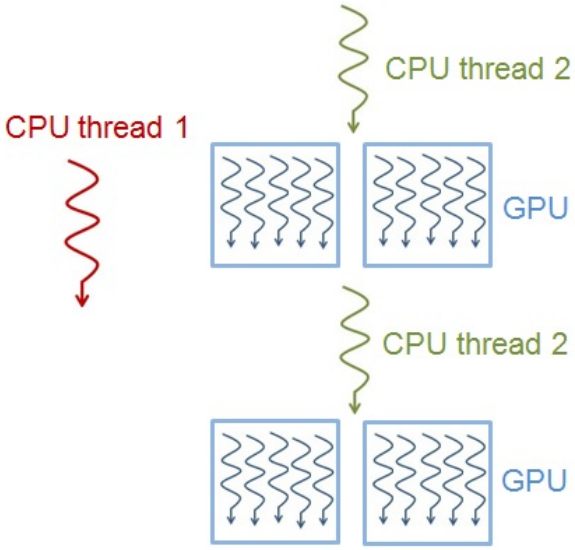


Figure 7.8: CPU and GPU threads working simultaneously for multi file basecaller.

finally a reduction method (discussed in Appendix A, Section A.2.1) is used to find the global maximum value which is subsequently used in the $(i+1)$th event and allows the threads to start working independently again.

7.4.2 Parallel Basecaller: Multiple Files

Using only 1024 threads is not the best solution (as in Section 7.4.1) when the number of cores is higher than 1024. In this section, I have increased the workload for GPU and used additional CPU thread to preprocess data for GPU use, as shown in Fig. 7.7 and 7.8.

In Fig. 7.7, the CPU thread#1 reads the input file, scales the input data with the given model, and copies the scaled data to a file which is later accessed by the CPU thread#2. As mentioned in Section 7.4.1, the input file might have data for two strands and two models

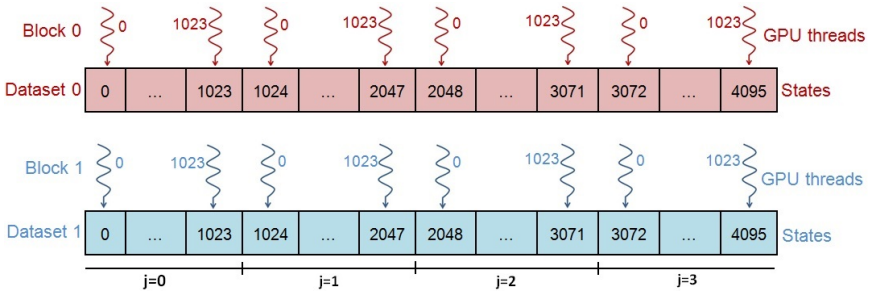


Figure 7.9: GPU threads accessing memory locations during multi files basecaller.

per strand. The term *dataset* is used denote data corresponding to one strand scaled by one model. For example, if there are two stands and two models per strand are given, then CPU thread#1 will generate four datasets after preprocessing.

CPU thread#1 keeps working until all of the input files is read. CPU thread#2 waits for one second in the beginning so that CPU thread#1 can preprocess at least one file before CPU thread#2 starts working on the final probability calculation. CPU thread#2 never needs to wait for CPU thread#1, as the preprocessing time for a file is less than the final probability calculation time.

The *Calculate the probability* block denoted in Fig. 7.7 is executed on the GPU. The memory arrangement and concepts of kernel call remain the same as discussed in Section 7.4.1. The only difference is that instead of one dataset, multiple datasets are executed in a single kernel call. 32 events from each dataset are executed in a single kernel call, and one block with 1024 threads is assigned for each dataset. Fig. 7.9 shows 1024 threads from the same block accessing memory for one dataset.

7.5 Experimental Results

The sample and the training data used in this study is available at [165]. The training data set used in this thesis is independent of any specific sequencing device like MinION as European Nucleotide Archive uses data from different sources (i.e., submissions of raw data, data provision from the major European sequencing centers and routine, etc) to train a dataset.

Read Number	Nanocall Serial Implementation (in bp/sec)	VA Basecaller Serial Implementation (in bp/sec)	VA Basecaller Parallel Implementation (in bp/sec)
Read 1	1033	1695	2653
Read 2	1171	2048	3496
Read 3	1152	2234	3472
Read 4	1003	1873	3026
Read 5	1061	1754	2814
Average	1080 +/- 70	1900 +/- 200	3100 +/- 400

Table 7.1: Parallel basecalling with single file.

Table 7.1 shows the experimental results of basecalling as discussed in Section 7.4.1. An open-source serial nanopore basecaller implementation ("Nanocall") described in [38] is used for comparison to the GPU implementation (in Table 7.1). The GPU computations presented in this section match the results computed by Nanocall which operates with roughly an 80% accuracy. In the last column of the Table 7.1, results from the parallel implementation are presented for a single file scenario as discussed in Section 7.4.1 and named as *VA basecaller parallel implementation*. Results using five different input files are presented in the Table 7.1.

The VA basecaller parallel implementation is capable of generating 3100 bp/sec which is $\approx 2.9\times$ more bp than Nanocall.

Table 7.2, presents the results from the implementation of the basecaller with multiple files. In each kernel call, two datasets are used. As discussed earlier, 1024 threads are used for one data set, 2048 threads are used for two datasets here. Using 2048 threads on a GPU with 1536 CUDA cores is reasonable to keep the cores busy. In Table 7.2, the speed for single-file execution is shown in the first row, and the rate of calling bp is higher than the rate listed in Table 7.1 due to executing multiple datasets in one kernel call.

For higher numbers of files, the bp generation rate saturates to around 7.2 kbp/s as the number of datasets executed in each kernel call remains constant. The bp calling rate for one file and 100 files are different though the number of datasets executed in the kernel for both cases are the same. The reason could be that the initial delay time applied to CPU

Number of files	Parallel VA basecaller (in bp/sec)
1	5683
100	7283
500	7203
1000	7242

Table 7.2: Parallel basecalling with multiple files.

thread#2, as discussed in Section 7.4.2, has a bigger effect on the total execution time of one file only. Conversely, the one second time delay has less effect on the total execution time of hundreds of files.

7.6 Summary and Conclusion

In this chapter, the basecalling step in an emerging DNA sequencing pipeline is implemented on a GPU. At first, one file is executed in parallel, and the bp calling rate is recorded as 3100 bp/s. In the next step, two CPU threads are used where the first CPU thread preprocessed the input file and the second CPU thread completes the remaining computational tasks including kernel calls. In each kernel call, two datasets are executed with 32 events each, and the implemented basecaller is capable of calling 7283 bp/sec. With high-end GPUs consisting of more CUDA cores, faster bp calling can be achieved. For example, with a Nvidia K80, which contains 3.25× more CUDA cores than GeForce GTX 680 and 12× larger memory, it should be possible to call at least 4× to 5× faster than the experimental results presented in this chapter, by executing more dataset in a kernel call.

Chapter 8

Conclusion and Future Work

In this thesis, GPU based sequencers have been implemented as key parts in a planner for a miniature unmanned aerial vehicle and a basecaller for a miniature real-time DNA sequencer. In the following sections, the results from the experiments in this thesis are summarized followed by a discussion of prospective future works related to the current research.

8.1 Global Path Planner

A global path planner for a UAV is proposed in this thesis where the UAV deploys wireless sensor nodes from the air to the ground for the purpose of constructing wireless sensor network in remote locations. The global path planner consists of two building blocks: a TSP solver as the sequencer and a PRM as the path explorer. A global distance matrix is considered which contains distances between each pair of waypoints.

At the beginning of a mission, Euclidean distances are inserted in the global distance matrix. The distance matrix is considered as global because the matrix is made accessible to both the TSP and the PRM. The TSP generates the shortest (or near optimal) sequence of waypoints considering the distances from the distance matrix. Next, the PRM constructs the paths between each pair of sequenced waypoints generated by the TSP. As there are obstacles in the environment, the PRM finds a safe flying path between all waypoints connected

by the TSP. This effectively modifies the initially anticipated distance between waypoints. Therefore, the PRM updates the calculated distance in the distance matrix so that the TSP can use the accurate distances in the next iteration. In the experiments, the TSP takes 4.27 sec to generate a sequence of 512 waypoints, and the PRM block takes 1.32 sec to construct paths between 511 pairs of waypoints. The iteration continues until the error becomes less than 5%. In the following sections, the TSP and the PRM blocks are summarized along with suggestions for future works, and at the end of this section, the future tasks related to the global planner are discussed.

8.1.1 Sequence Generator

For sequence generation, the travelling salesman problem is solved using a heuristic employing the genetic algorithm (GA), clustering, and nearest neighbour techniques, to find the near optimal solution. The TSP is solved for sequence generation over 4096 waypoints. A manycore GPU is used to accelerate the computational process. Due to the GPU's limited on-chip shared memory, and to reduce the data traffic between CPU and GPU, the problem is divided into multiple subproblems using the clustering technique. The GA is applied individually to the waypoints inside each cluster which are executed inside the GPU. When the number of waypoints inside the cluster is less than or equal to 8, instead of applying GA, all possible combinations are checked to find the best sequence among the waypoints in that cluster. Finally, the sequenced paths for each block are transferred to the CPU and the GA is applied to all of the waypoints using the CPU to generate the final sequence.

The parallel sequence generator executed 4.8× faster to a solution than its serial counterpart. The generator in both cases (serial and parallel) finds near optimal results although these are not necessarily identical with each run. The serial implementation generates 1.4× longer path. The execution time also varies with the cluster size. Experimental results show that the parallel TSP calculation works 5.4× to 14.9× faster when the waypoints used per cluster are reduced. The reason for this is that the data for clusters are kept in on-chip shared

memory, and the occupancy of the shared memory is low when the number of waypoints per cluster is small. As a result, the performance of the GPU execution decreases when the occupancy of the shared memory is high. The main advantage of using the algorithm is that it can find a path for a significant number (up to 4096) of waypoints.

The implemented planner can also generate paths for more significant problems consisting of more than 4096 waypoints. But the error rate compared to the optimal solution of the sequenced path increases. In such cases, we should use more clusters and multiple levels of clustering, e.g., generating clusters with the cluster heads themselves. We should also try not to overload the shared memory by using read-only memories. Another option to explore is to use a high-end GPU for the sequence calculation. The implemented sequence generator will be able to generate faster results as the number of available CUDA cores, and the memory bandwidth is higher which reduces the time required for read/write instructions for the threads. With a high-end GPU like Tesla K80 (weights 2.2 lbs), we cannot increase the number of waypoints per cluster per block used in this research as the size of shared memory remains constant for all NVIDIA GPUs. As a result, instead of shared memory, other memories (e.g., texture memory and global memory) need to be explored for storing the distance matrix so that more extensive problem can be solved. The drawback of using such high-end GPU is higher energy consumption.

8.1.2 Path Explorer

A sample-based probabilistic roadmap algorithm is used to find an obstacle-free flying path for the UAV. In short, the PRM process starts by reading the start and destination node locations as well as the record of obstacles followed by the four major steps: sample node generation, milestone calculation, nearest neighbour search, and graph search. The sample nodes are generated within a cubic area whose two, diagonally opposing, corners are set as the start and destination nodes, respectively. The number of sample nodes is kept constant by combining the milestone calculation and the sample node generation steps in this thesis.

For the nearest neighbour search, ANN library is used for serial implementation, and a customized nearest neighbour algorithm is used for parallel implementation which was executed on a GPU. A Dijkstra graph search is used for finding the final result. During the graph search, instead of sorting, a min-heap data structure is used to find the shortest path from the frontier which reduces the time complexity.

The execution of the k-nearest neighbour search and the sample node generation steps take 96% to 99% and 1% to 3% of the total execution time during the serial implementation. So these two steps are executed in parallel using a GPU. Three different obstacle-infused environment types are studied in this thesis. Each environment contains varying amounts of obstacles: 0.35%, 2.58%, and 5.3% of the total volume. The entire execution time for the parallel implementation falls in the range of a few milliseconds to less than 10 sec. The maximum speedups recorded for the sample generation, the k-nn calculation, and the total time for the PRM are 81.18×, 194.81×, and 164× respectively.

For the future work on the local planner, PRM, in the current implementation, for finding the k-nn nodes, all nodes are examined to see the nn nodes. For example, if there are 60 nodes and 30 nearest neighbour nodes are required, in the current code, a node finds the distances from itself to the remaining 59 nodes and then uses bitonic sort for selecting the 30 closest nodes. But considering more intermediate sample nodes, this process can be cumbersome. So additional features like Voronoi diagrams, BVH, clustering, etc. can be applied to separate the plane which will reduce the complexity of the k-nn procedure, with some added computational cost.

Now I will discuss the future tasks related to the global path planner. The global planner can be implemented and examined for the number of iterations it takes to converge to the final path. In the proposed planner, the TSP block executes first, then the PRM block is engaged. The process continues iteratively until the result does not change by more than 5% (a reasonable choice for the scale of delivery problems considered herein). The convergence rate to the final result can be examined when the global planner is ultimately implemented.

Alternatively, one may explore the possibility of executing the PRM first then the TSP. In such cases (implementing PRM first), PRM needs to calculate all possible edges first; then one needs to call the TSP once. Another option when executing the PRM first, instead of calculating all possible edges, limited edges will be calculated first. The limited edges refer to the edges that might have a higher probability to exist in the final sequenced path. Then TSP block will generate the path. The working loop of the TSP and the PRM will continue until the result does not change by more than 5%.

To measure the probable edges for the PRM, as mentioned in the previous paragraph, clustering can be used, i.e., there is a high chance that nodes close to each other will be connected directly to the final path. If one prefers to use clustering, he/she can also combine the TSP and the PRM block. In more details, the given waypoints will be divided into smaller groups or clusters. The coordinate values of the waypoints will be transmitted and stored in the GPU's shared memory. One block of threads inside the GPU will be assigned to one cluster. So for multiple clusters, several blocks will be allocated. Threads will execute PRM for all the waypoints of the designated cluster first; then the GA may be applied. After the kernel calculation, the sequenced path should be transferred to the CPU.

8.2 DNA Sequencing

Basecalling, one of the critical steps in the DNA sequencing pipeline, is implemented in this thesis which gets the event signal as input and generates a series of equivalent letters representing the four bases comprising the molecular alphabet. The uncertainty of the bases can be expressed using a hidden Markov model, and a Viterbi algorithm is used to solve the problem. In the sequencing process, each event depends on its previous event, and there are 4096 probably states for each event assuming a 6-mer sensitive DNA sensor.

As each event depends on its last event, two adjacent events cannot be calculated in parallel which limits the proper usage of the computational ability of the GPU. So in this

thesis, two DNA strands: the template and the compliment, are solved in parallel which means 2×4096 states are calculated in parallel on the GPU. The basecalling requires additional scaling before the state calculation which is computed using CPU threads. Using the CPU threads, the GPU does not need to wait for the next DNA strand's scaling and works as a pipeline configuration. The implemented basecaller is capable of generating 7283 bp/sec.

With high-end GPUs consisting of more CUDA cores, faster basecalling can be achieved. For example, with a NVIDIA K80, which contains 3.25× more CUDA cores than GeForce GTX 680 and 12× larger memory. By handling more datasets in a kernel call, the NVIDIA K80 should execute basecalling at least 4× to 5× faster than the experimental results presented in this thesis. Such implementation requires more complex code as the events in each file are not the same, and the code must keep track of the source file of every event.

Finally, the accuracy of the basecalling does not precisely depend on how fast the result is computed, rather it depends on the quality and the volume of training data. To improve accuracy, we can add more training data when it is possible. We can also apply anomaly detection techniques to identify any erroneous data. Anomaly detection techniques allow identifying data items that differ significantly from the majority of the records in a given dataset.

Bibliography

[1] S. Magierowski, "Hmm-based basecalling for nanopore systems," York University, Tech. Rep., July 2016.

[2] S. Magierwski, "Minion notes," York University, Tech. Rep., July 2016.

[3] R. Hossain, R. Mittmann, E. Ghafar-Zadeh, G. G. Messier, and S. Magierowski, "Gpu base calling for dna strand sequencing," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2017, pp. 96–99.

[4] "Android 4.2 apis." [Online]. Available: https://developer.android.com/about/versions/android-4.2.html

[5] B. B. Gear, "Apple's making its own gpu to control its own destiny," [Online], March 4, 2017. [Online]. Available: https://www.wired.com/2017/04/apples-making-gpu-control-destiny/

[6] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, ser. Applications of GPU Computing Series. Elsevier Science, 2010.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[8] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[9] "What is gpu accelerated computing?" [Online]. Available: http://www.nvidia.ca/object/what-is-gpu-computing.html

[10] "Opencl overview." [Online]. Available: https://www.khronos.org/opencl/

[11] "Opencl." [Online]. Available: https://en.wikipedia.org/wiki/OpenCL

[12] W. Hwu, *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 2011.

[13] X. Zha, S. Y. Lim, and S. Fok, "Integrated knowledge-based assembly sequence planning," *The International Journal of Advanced Manufacturing Technology*, vol. 14, no. 1, pp. 50–64, 1998.

[14] P. Gu and X. Yan, "Cad-directed automatic assembly sequence planning," *International Journal of Production Research*, vol. 33, no. 11, pp. 3069–3100, 1995.

[15] W. Hui, X. Dong, and D. Guanghong, "A genetic algorithm for product disassembly sequence planning," *Neurocomputing*, vol. 71, no. 13-15, pp. 2720–2726, 2008.

[16] T. Cao and A. C. Sanderson, "Task sequence planning using fuzzy petri nets," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 25, no. 5, pp. 755–768, 1995.

[17] P. Perazzo, K. Ariyapala, M. Conti, and G. Dini, "The verifier bee: A path planner for drone-based secure location verification," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2015 IEEE 16th International Symposium on a*. IEEE, 2015, pp. 1–9.

[18] U. Çekmez, M. Özsığınan, M. Aydın, and Ö. K. Şahingöz, "Uav path planning with parallel genetic algorithms on cuda architecture," 2014.

[19] S.-Y. Shin, I.-H. Lee, D. Kim, and B.-T. Zhang, "Multiobjective evolutionary optimization of dna sequences for reliable dna computing," *IEEE transactions on evolutionary computation*, vol. 9, no. 2, pp. 143–158, 2005.

[20] C. Trapnell and M. C. Schatz, "Optimizing data intensive gpgpu computations for dna sequence alignment," *Parallel Computing*, vol. 35, no. 8, pp. 429–440, 2009.

[21] R. Bischoff, J. Kurth, G. Schreiber, R. Koeppe, A. Albu-Schäffer, A. Beyer, O. Eiberger, S. Haddadin, A. Stemmer, G. Grunwald *et al.*, "The kuka-dlr lightweight robot arm-a new reference platform for robotics research and manufacturing," in *Robotics (ISR), 2010 41st international symposium on and 2010 6th German conference on robotics (ROBOTIK)*. VDE, 2010, pp. 1–8.

[22] L. R. Kavoussi, R. G. Moore, J. B. Adams, and A. W. Partin, "Comparison of robotic versus human laparoscopic camera control," *The Journal of urology*, vol. 154, no. 6, pp. 2134–2136, 1995.

[23] H. J. Tovey, K. Ratcliff, K. E. Toso, and P. W. Hinchliffe, "Robotic arm dlus for performing surgical tasks," Dec. 7 2004, uS Patent 6,827,712.

[24] S. Zimmerman and A. Abdelkefi, "Review of marine animals and bioinspired robotic vehicles: Classifications and characteristics," *Progress in Aerospace Sciences*, 2017.

[25] Z. Tao, P. Bonnifait, V. Frémont, J. Ibanez-Guzman, and S. Bonnet, "Road-centered map-aided localization for driverless cars using single-frequency gnss receivers," *Journal of Field Robotics*, 2017.

[26] F. Vanegas, D. Campbell, N. Roy, K. J. Gaston, and F. Gonzalez, "Uav tracking and following a ground target under motion and localisation uncertainty," in *Aerospace Conference, 2017 IEEE*. IEEE, 2017, pp. 1–10.

[27] R. R. Selmic, V. V. Phoha, and A. Serwadda, *Wireless Sensor Networks: Security, Coverage, and Localization*. Springer, 2016.

[28] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The traveling salesman problem: a computational study*. Princeton University Press, 2007.

[29] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[30] M. Jain, I. T. Fiddes, K. H. Miga, H. E. Olsen, B. Paten, and M. Akeson, "Improved data analysis for the minion nanopore sequencer," *Nature methods*, vol. 12, no. 4, pp. 351–356, 2015.

[31] B. Ewing, L. Hillier, M. C. Wendl, and P. Green, "Base-calling of automated sequencer traces usingphred. i. accuracy assessment," *Genome research*, vol. 8, no. 3, pp. 175–185, 1998.

[32] "Oxford nanopore technologies," [Online]. [Online]. Available: https://nanoporetech.com/

[33] S. Magierowski, "Nanopore basecaller analysis: Computations, comparisons, costs," York University, Tech. Rep., December 2016.

[34] R. Longbottom, "Roy longbottom's pc benchmark collection. [online]." 2016. [Online]. Available: http://www.roylongbottom.org.uk/dhrystone%20results.htm

[35] (2018) Amazon emr pricing. [Online]. Available: https://aws.amazon.com/emr/pricing/

[36] G. D. Forney, "The viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[37] G. Bonaccorso, *Machine Learning Algorithms*. Packt Publishing Ltd, 2017.

[38] M. David, L. J. Dursi, D. Yao, P. C. Boutros, and J. T. Simpson, "Nanocall: An Open Source Basecaller for Oxford Nanopore Sequencing Data." *Bioinformatics (Oxford, England)*, pp. 1–8, Sep. 2016.

[39] J. M. Tendler, J. S. Dodson, J. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, 2002.

[40] W. J. Kaufmann and L. L. Smarr, *Supercomputing and the Transformation of Science*. WH Freeman & Co., 1992.

[41] "The true cost of building a data warehouse," September 2017. [Online]. Available: https://www.cooladata.com/cost-of-building-a-data-warehouse/

[42] T. Gneiting and A. E. Raftery, "Weather forecasting with ensemble methods," *Science*, vol. 310, no. 5746, pp. 248–249, 2005.

[43] D. Menemenlis, C. Hill, A. Adcrocft, J.-M. Campin, B. Cheng, B. Ciotti, I. Fukumori, P. Heimbach, C. Henze, A. Köhl *et al.*, "Nasa supercomputer improves prospects for ocean climate research," *Eos, Transactions American Geophysical Union*, vol. 86, no. 9, pp. 89–96, 2005.

[44] A. Ganesan, M. L. Coote, and K. Barakat, "Molecular 'time-machines' to unravel key biological events for drug design," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 2017.

[45] "World's fastest supercomputer now has chinese chip technology." [Online]. Available: https://www.bloomberg.com/news/articles/2016-06-20/world-s-fastest-supercomputer-now-has-chinese-chip-technology

[46] D. Pricea, M. Clarka, B. Barsdella, R. Babichb, and L. Greenhilla, "Optimizing performance per watt on gpus in high performance computing: temperature, frequency and voltage effects," *arXiv preprint arXiv:1407.8116*, 2014.

[47] J.-P. Farrugia, P. Horain, E. Guehenneux, and Y. Alusse, "Gpucv: A framework for image processing acceleration with graphics processors," in *Multimedia and Expo, 2006 IEEE International Conference on*. IEEE, 2006, pp. 585–588.

[48] J. E. Cates, A. E. Lefohn, and R. T. Whitaker, "Gist: an interactive, gpu-based level set segmentation tool for 3d medical images," *Medical Image Analysis*, vol. 8, no. 3, pp. 217–231, 2004.

[49] Y. Heng and L. Gu, "Gpu-based volume rendering for medical image visualization," in *Engineering in Medicine and Biology Society, 2005. IEEE-EMBS 2005. 27th Annual International Conference of the*. IEEE, 2006, pp. 5145–5148.

[50] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on cuda," in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 3. IEEE, 2008, pp. 198–201.

[51] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Gpu-based video feature tracking and matching," in *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, vol. 278, 2006, p. 4321.

[52] A. Tumeo and O. Villa, "Accelerating dna analysis applications on gpu clusters," in *Application Specific Processors (SASP), 2010 IEEE 8th Symposium on*. IEEE, 2010, pp. 71–76.

[53] M. Griebel and P. Zaspel, "A multi-gpu accelerated solver for the three-dimensional two-phase incompressible navier-stokes equations," *Computer Science-Research and Development*, vol. 25, no. 1-2, pp. 65–73, 2010.

[54] J. Mielikainen, B. Huang, H. Huang, and M. Goldberg, "Gpu acceleration of the updated goddard shortwave radiation scheme in the weather research and forecasting (wrf) model," *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of*, vol. 5, no. 2, pp. 555–562, 2012.

[55] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[56] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. Xing, "Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines," *arXiv preprint arXiv:1512.06216*, 2015.

[57] R. Socher, B. Huval, B. Bath, C. D. Manning, and A. Y. Ng, "Convolutional-recursive deep learning for 3d object classification," in *Advances in Neural Information Processing Systems*, 2012, pp. 656–664.

[58] N. Ganesan, R. D. Chamberlain, J. Buhler, and M. Taufer, "Accelerating hmmer on gpus by implementing hybrid data and task parallelism," in *Proceedings of the first ACM international conference on bioinformatics and computational biology.* ACM, 2010, pp. 418–421.

[59] R. Li, Y. Dou, and D. Zou, "Efficient parallel implementation of three-point viterbi decoding algorithm on cpu, gpu, and fpga," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 3, pp. 821–840, 2014.

[60] N. C. McGinty, R. A. Kennedy, and P. Hocher, "Parallel trellis viterbi algorithm for sparse channels," *IEEE Communications Letters*, vol. 2, no. 5, pp. 143–145, 1998.

[61] C.-S. Lin, W.-L. Liu, W.-T. Yeh, L.-W. Chang, W.-M. W. Hwu, S.-J. Chen, and P.-A. Hsiung, "A tiling-scheme viterbi decoder in software defined radio for gpus," in *Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference on.* IEEE, 2011, pp. 1–4.

[62] J. Chong, E. Gonina, and K. Keutzer, "Efficient automatic speech recognition on the gpu," in *GPU Computing Gems Emerald Edition.* Elsevier, 2011, pp. 601–618.

[63] M. H. Radfar, R. M. Dansereau, and W. Wong, "Speech separation using gain-adapted factorial hidden markov models," *arXiv preprint arXiv:1901.07604*, 2019.

[64] Z. Du, Z. Yin, and D. A. Bader, "A tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

[65] J. P. Walters, V. Balu, S. Kompalli, and V. Chaudhary, "Evaluating the use of gpus in liver image segmentation and hmmer database searches," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[66] "Us military to launch fastest-ever plane." [Online]. Available: https://www.theguardian.com/world/2011/aug/10/us-military-fastest-plane-falcon

[67] S. Scherer, S. Singh, L. Chamberlain, and M. Elgersma, "Flying fast and low among obstacles: Methodology and experiments," *The International Journal of Robotics Research*, vol. 27, no. 5, pp. 549–574, 2008.

[68] D. D. V. Pandit and A. Poojari, "A study on amazon prime air for feasibility and profitability–a graphical data analysis," *IOSR Journal of Business and Management*, vol. 16, no. 11-1, pp. 06–11, 2014.

[69] A. Glaser, "One of china's biggest online retailers is building a delivery drone that can carry 2000 pounds of cargo," May 2017, [Online; posted 22-May-2017]. [Online]. Available: https://www.recode.net/2017/5/22/15666446/jd-china-drone-delivery-two-thousand-2000-pounds

[70] B. Siciliano and O. Khatib, Eds., *Springer Handbook of Robotics*. Springer, 2008. [Online]. Available: http://www.libreka.de/9783540239574/

[71] H. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, vol. 2. IEEE, 1985, pp. 116–121.

[72] P. E. Agre and D. Chapman, "Pengi: An implementation of a theory of activity." in *AAAI*, vol. 87, no. 4, 1987, pp. 286–272.

[73] E. Gat, "Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots," in *AAAi*, vol. 1992, 1992, pp. 809–815.

[74] K. Doganay, H. Hmam, S. P. Drake, and A. Finn, "Centralized path planning for unmanned aerial vehicles with a heterogeneous mix of sensors," in *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2009 5th International Conference on*. IEEE, 2009, pp. 91–96.

[75] W. Li and C. G. Cassandras, "Centralized and distributed cooperative receding horizon control of autonomous vehicle missions," *Mathematical and computer modelling*, vol. 43, no. 9, pp. 1208–1228, 2006.

[76] A. Ryan, X. Xiao, S. Rathinam, J. Tisdale, M. Zennaro, D. Caveney, R. Sengupta, and J. K. Hedrick, "A modular software infrastructure for distributed control of collaborating uavs," in *Proceedings of the AIAA Conference on Guidance, Navigation, and Control*, 2006.

[77] M. Quaritsch, K. Kruggl, D. Wischounig-Strucl, S. Bhattacharya, M. Shah, and B. Rinner, "Networked uavs as aerial sensor network for disaster management applications," *e & i Elektrotechnik und Informationstechnik*, vol. 127, no. 3, pp. 56–63, 2010.

[78] S. Scherer, J. Rehder, S. Achar, H. Cover, A. Chambers, S. Nuske, and S. Singh, "River mapping from a flying robot: state estimation, river detection, and obstacle mapping," *Autonomous Robots*, vol. 33, no. 1-2, pp. 189–214, 2012.

[79] P. Corke, S. Hrabar, R. Peterson, D. Rus, S. Saripalli, and G. Sukhatme, "Deployment and connectivity repair of a sensor net with a flying robot," in *Experimental Robotics IX*. Springer, 2006, pp. 333–343.

[80] ——, "Deployment and repair of a sensor network using an unmanned aerial vehicle," in *Proc. IEEE Int. Conf. Robots and Auto.*, Apr. 2004, pp. 3602–3608.

[81] A. Ollero, M. Bernard, M. La Civita, L. Van Hoesel, P. Marron, J. Lepley, and E. de Andres, "Aware: Platform for autonomous self-deploying and operation of wireless sensor-actuator networks cooperating with unmanned aerial vehicles," in *Safety, Security and Rescue Robotics, 2007. SSRR 2007. IEEE International Workshop on*, sept. 2007, pp. 1 –6.

[82] D.-T. Ho, E. I. Gr, P. Sujit, T. A. Johansen, J. B. Sousa *et al.*, "Cluster-based communication topology selection and uav path planning in wireless sensor networks," in *Unmanned Aircraft Systems (ICUAS), 2013 International Conference on*. IEEE, 2013, pp. 59–68.

[83] D. Tian and N. D. Georganas, "A coverage-preserving node scheduling scheme for large wireless sensor networks," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. ACM, 2002, pp. 32–41.

[84] H. Yu and M. Guo, "An efficient oil and gas pipeline monitoring systems based on wireless sensor networks," in *Information Security and Intelligence Control (ISIC), 2012 International Conference on*. IEEE, 2012, pp. 178–181.

[85] I. Jawhar, N. Mohamed, and K. Shuaib, "A framework for pipeline infrastructure monitoring using wireless sensor networks," in *Wireless telecommunications symposium, 2007. WTS 2007*. IEEE, 2007, pp. 1–7.

[86] S. N. Spitz and A. A. Requicha, "Multiple-goals path planning for coordinate measuring machines," in *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, vol. 3. IEEE, 2000, pp. 2322–2327.

[87] F. Adolf, A. Langer, L. d. M. P. e Silva, and F. Thielecke, "Probabilistic roadmaps and ant colony optimization for uav mission planning," *IFAC Proceedings Volumes*, vol. 40, no. 15, pp. 264–269, 2007.

[88] F.-M. Adolf and A. Müller, "Asymmetric 2-opt scheduling for roadmap-based task planning in urban terrain," *AIAA Infotech@ Aerospace*, 2013.

[89] B. Englot and F. Hover, "Multi-goal feasible path planning using ant colony optimization," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 2255–2260.

[90] S. J. Russell, P. Norvig, E. Davis, S. J. Russell, and S. J. Russell, *Artificial intelligence: a modern approach*. Prentice hall Upper Saddle River, NJ, 2010.

[91] S.-G. Choi, W.-J. Jung, and J.-H. Choi, "3d-based uav path-planning algorithm considering altitude and reconnaissance areas," 2017.

[92] S. A. Bortoff, "Path planning for uavs," in *American Control Conference, 2000. Proceedings of the 2000*, vol. 1, no. 6. IEEE, 2000, pp. 364–368.

[93] "Concorde tsp solver." [Online]. Available: http://www.math.uwaterloo.ca/tsp/concorde/index.html

[94] "Discrete and combinatorial optimization." [Online]. Available: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

[95] M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," *Evolutionary Computation, IEEE Transactions on*, vol. 1, no. 1, pp. 53–66, 1997.

[96] T. Stutzle and H. Hoos, "Max-min ant system and local search for the traveling salesman problem," in *Evolutionary Computation, 1997., IEEE International Conference on*. IEEE, 1997, pp. 309–314.

[97] R. Baraglia, J. I. Hidalgo, and R. Perego, "A hybrid heuristic for the traveling salesman problem," *Evolutionary Computation, IEEE Transactions on*, vol. 5, no. 6, pp. 613–622, 2001.

[98] N. Özalp and O. K. Sahingoz, "Optimal uav path planning in a 3d threat environment by using parallel evolutionary algorithms," in *Unmanned Aircraft Systems (ICUAS), 2013 International Conference on*. IEEE, 2013, pp. 308–317.

[99] M. A. O'Neil, D. Tamir, and M. Burtscher, "A parallel gpu version of the traveling salesman problem," in *2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2011, pp. 348–353.

[100] K. Rocki and R. Suda, "An efficient gpu implementation of a multi-start tsp solver for large problem instances," in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, ser. GECCO Companion '12. New York, NY, USA: ACM, 2012, pp. 1441–1442. [Online]. Available: http://doi.acm.org/10.1145/2330784.2330978

[101] K. M. Rocki and R. Suda, "Brief announcement: a gpu accelerated iterated local search tsp solver," in *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '12. New York, NY, USA: ACM, 2012, pp. 188–189. [Online]. Available: http://doi.acm.org/10.1145/2312005.2312041

[102] J. Fu, L. Lei, and G. Zhou, "A parallel ant colony optimization algorithm with gpu-acceleration based on all-in-roulette selection," in *Advanced Computational Intelligence (IWACI), 2010 Third International Workshop on*, 2010, pp. 260–264.

[103] H. Bai, D. Ouyang, X. Li, L. He, and H. Yu, "Max-min ant system on gpu with cuda," in *Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference on*, 2009, pp. 801–804.

[104] N. Fujimoto and S. Tsutsui, "A highly-parallel tsp solver for a gpu computing platform," in *International Conference on Numerical Methods and Applications*. Springer, 2010, pp. 264–271.

[105] S. Chen, S. Davis, H. Jiang, and A. Novobilski, "Cuda-based genetic algorithm on traveling salesman problem," in *Computer and Information Science 2011*. Springer, 2011, pp. 241–252.

[106] J. Li, L. Zhang, and L. Liu, "A parallel immune algorithm based on fine-grained model with gpu-acceleration," in *Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference on*, 2009, pp. 683–686.

[107] S. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[108] J. Y. Hwang, J. S. Kim, S. S. Lim, and K. H. Park, "A fast path planning by path graph optimization," *IEEE Transactions on systems, man, and cybernetics-part a: systems and humans*, vol. 33, no. 1, pp. 121–129, 2003.

[109] A. R. Soltani, H. Tawfik, J. Y. Goulermas, and T. Fernando, "Path planning in construction sites: performance evaluation of the dijkstra, a*, and ga search algorithms," *Advanced engineering informatics*, vol. 16, no. 4, pp. 291–303, 2002.

[110] J. Barraquand, B. Langlois, and J.-C. Latombe, "Numerical potential field techniques for robot path planning," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 2, pp. 224–241, 1992.

[111] J. Pan, C. Lauterbach, and D. Manocha, "g-planner: Real-time motion planning and global navigation using gpus." in *AAAI*, 2010.

[112] J. Kim, S. Shin, J. Wu, S.-D. Kim, and C.-G. Kim, "Obstacle avoidance path planning for uav using reinforcement learning under simulated environment," in *IASER 3rd International Conference on Electronics, Electrical Engineering, Computer Science, Okinawa*, 2017, pp. 34–36.

[113] S. Mittal and K. Deb, "Three-dimensional offline path planning for uavs using multiobjective evolutionary algorithms," in *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on.* IEEE, 2007, pp. 3195–3202.

[114] C. Zhang, Z. Zhen, D. Wang, and M. Li, "Uav path planning method based on ant colony optimization," in *Control and Decision Conference (CCDC), 2010 Chinese.* IEEE, 2010, pp. 3790–3792.

[115] Y. V. Pehlivanoglu, "A new vibrational genetic algorithm enhanced with a voronoi diagram for path planning of autonomous uav," *Aerospace Science and Technology*, vol. 16, no. 1, pp. 47–55, 2012.

[116] C. Goerzen, Z. Kong, and B. Mettler, "A survey of motion planning algorithms from the perspective of autonomous uav guidance," *Journal of Intelligent & Robotic Systems*, vol. 57, no. 1, pp. 65–100, 2010.

[117] P. Sujit and R. Beard, "Multiple uav path planning using anytime algorithms," in *American Control Conference, 2009. ACC'09.* IEEE, 2009, pp. 2978–2983.

[118] M. Kothari, I. Postlethwaite, and D.-W. Gu, "Multi-uav path planning in obstacle rich environments using rapidly-exploring random trees," in *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on.* IEEE, 2009, pp. 3069–3074.

[119] Y.-h. Qu, Q. Pan, and J.-g. Yan, "Flight path planning of uav based on heuristically search and genetic algorithms," in *Industrial Electronics Society, 2005. IECON 2005. 31st Annual Conference of IEEE.* IEEE, 2005, pp. 5–pp.

[120] J. Pan, C. Lauterbach, and D. Manocha, "Efficient nearest-neighbor computation for gpu-based motion planning," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. IEEE, 2010, pp. 2243–2248.

[121] J. Pan and D. Manocha, "Gpu-based parallel collision detection for fast motion planning," *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 187–200, 2012.

[122] J. Yoon, J. Park, and M. Baeg, "Gpu-based collision detection for sampling-based motion planning," in *Ubiquitous Robots and Ambient Intelligence (URAI), 2013 10th International Conference on*. IEEE, 2013, pp. 215–218.

[123] F. Sanger, S. Nicklen, and A. R. Coulson, "Dna sequencing with chain-terminating inhibitors," *Proceedings of the national academy of sciences*, vol. 74, no. 12, pp. 5463–5467, 1977.

[124] L. H. Harwell, L. Hood, and M. L. Goldberg, *Genetics from genes to genomes*. McGraw-Hill,, 2004, no. 576.5 G328g.

[125] L. M. Smith, J. Z. Sanders, R. J. Kaiser, P. Hughes, C. Dodd, C. R. Connell, C. Heiner, S. B. Kent, and L. E. Hood, "Fluorescence detection in automated dna sequence analysis," *Nature*, vol. 321, no. 6071, p. 674, 1986.

[126] M. Marsh, O. Tu, V. Dolnik, D. Roach, N. Solomon, K. Bechtol, P. Smietana, L. Wang, X. Li, P. Cartwright *et al.*, "High-throughput dna sequencing on a capillary array electrophoresis system." *Journal of capillary electrophoresis*, vol. 4, no. 2, pp. 83–89, 1997.

[127] S. Magierowski, Y. Huang, C. Wang, and E. Ghafar-Zadeh, "Nanopore-cmos interfaces for dna sequencing," *Biosensors*, vol. 6, no. 3, p. 42, 2016.

[128] N. J. Loman, R. V. Misra, T. J. Dallman, C. Constantinidou, S. E. Gharbia, J. Wain, and M. J. Pallen, "Performance comparison of benchtop high-throughput sequencing platforms," *Nature biotechnology*, vol. 30, no. 5, p. 434, 2012.

[129] S. Balasubramanian, "Polynucleotide sequencing," Dec. 21 2004, uS Patent 6,833,246.

[130] P. Nyren, "Method of sequencing dna based on the detection of the release of pyrophosphate and enzymatic nucleotide degradation," Jul. 10 2001, uS Patent 6,258,568.

[131] J. Clarke, H.-C. Wu, L. Jayasinghe, A. Patel, S. Reid, and H. Bayley, "Continuous base identification for single-molecule nanopore dna sequencing," *Nature nanotechnology*, vol. 4, no. 4, p. 265, 2009.

[132] C. L. Ip, M. Loose, J. R. Tyson, M. de Cesare, B. L. Brown, M. Jain, R. M. Leggett, D. A. Eccles, V. Zalunin, J. M. Urban *et al.*, "Minion analysis and reference consortium: Phase 1 data release and analysis," *F1000Research*, vol. 4, 2015.

[133] V. Boža, B. Brejová, and T. Vinař, "Deepnano: Deep recurrent neural networks for base calling in minion nanopore reads," *PloS one*, vol. 12, no. 6, p. e0178751, 2017.

[134] M. Ratković, "Deep learning model for base calling of minion nanopore reads," Ph.D. dissertation, Fakultet Elektrotehnike i Računarstva, Sveučilište u Zagrebu, 2017.

[135] H. Jiang, N. Ganesan, and Y.-D. Yao, "Cudampf++: A proactive resource exhaustion scheme for accelerating homologous sequence search on cuda-enabled gpu," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[136] "Social impact of the gpu: Energy efficiency," NVIDIA, http://www.nvidia.ca/object/gcr-energy-efficiency.html.

[137] *CUDA C PROGRAMMING GUIDE*, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, NVIDIA, October 2012, ver. 4.2.

[138] "Nvidia geforce gtx 680," NVIDIA, whitepaper. [Online]. Available: http://www.nvidia.com/content/PDF/product-specifications/GeForce_GTX_680_Whitepaper_FINAL.pdf

[139] "Geforce gtx 680." [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications

[140] A. Pandey and D. R. Parhi, "Matlab simulation for mobile robot navigation with hurdles in cluttered environment using minimum rule based fuzzy logic controller," *Procedia Technology*, vol. 14, pp. 28–34, 2014.

[141] K. Yang and S. Sukkarieh, "3d smooth path planning for a uav in cluttered natural environments," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. IEEE, 2008, pp. 794–800.

[142] D.-T. Ho and S. Shimamoto, "Highly reliable communication protocol for wsn-uav system employing tdma and pfs scheme," in *GLOBECOM Workshops (GC Wkshps), 2011 IEEE*. IEEE, 2011, pp. 1320–1324.

[143] *The high performance unmanned helicopter designed for a wide range of industrial uses: RMAX Type II G/ Type II*, Yamaha. [Online]. Available: http://rmax.yamaha-motor.com.au/sites/rmax/files/pdf/RMAX_Brochure.pdf

[144] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[145] R. Soleymani-Fard, C.-Y. Shih, M. Baudewig, and P. J. Marron, "Coplanner: A wireless sensor network deployment planning architecture using unmanned vehicles as deployment tools," in *SENSORCOMM 2012, The Sixth International Conference on Sensor Technologies and Applications*, 2012, pp. 73–76.

[146] N. Fujimoto and S. Tsutsui, "A highly-parallel tsp solver for a gpu computing platform," in *Numerical Methods and Applications*. Springer, 2011, pp. 264–271.

[147] R. Hossain, S. Magierowski, and G. G. Messier, "Gpu enhanced path finding for an unmanned aerial vehicle," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 1285–1293.

[148] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[149] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "Fast in-place sorting with cuda based on bitonic sort," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2009, pp. 403–410.

[150] D. M. Mount and S. Arya, "Ann: A library for approximate nearest neighbor searching." [Online]. Available: https://www.cs.umd.edu/~mount/ANN/

[151] H. Samet, *The design and analysis of spatial data structures*. Addison-Wesley Reading, MA, 1990, vol. 199.

[152] C. Toolkit, "4.2: Curand guide," *NVIDIA Corporation, Santa Clara*, 2012.

[153] J. D. Watson, F. H. Crick *et al.*, "Molecular structure of nucleic acids," *Nature*, vol. 171, no. 4356, pp. 737–738, 1953.

[154] K. Dolan and C. WRIGHT, "Analysis of a polymer from multi-dimensional measurements," 2015, priority: Mar. 12, 2014, wO Patent App. PCT/GB2015/050,776. [Online]. Available: https://www.google.com/patents/WO2015140535A1?cl=nl

[155] J. M. Heather and B. Chain, "The sequence of sequencers: the history of sequencing dna," *Genomics*, vol. 107, no. 1, pp. 1–8, 2016.

[156] P. M. Ashton, S. Nair, T. Dallman, S. Rubino, W. Rabsch, S. Mwaigwisya, J. Wain, and J. O'grady, "Minion nanopore sequencing identifies the position and structure of a

bacterial antibiotic resistance island," *Nature biotechnology*, vol. 33, no. 3, pp. 296–300, 2015.

[157] E. Karlsson, A. Lärkeryd, A. Sjödin, M. Forsman, and P. Stenberg, "Scaffolding of a bacterial genome using minion nanopore sequencing," *Scientific reports*, vol. 5, p. 11996, 2015.

[158] A. Kilianski, J. L. Haas, E. J. Corriveau, A. T. Liem, K. L. Willis, D. R. Kadavy, C. N. Rosenzweig, and S. S. Minot, "Bacterial and viral identification and differentiation by amplicon sequencing on the minion nanopore sequencer," *Gigascience*, vol. 4, no. 1, p. 12, 2015.

[159] P. Boufounos, S. El-Difrawy, and D. Ehrlich, "Basecalling using hidden markov models," *Journal of the Franklin Institute*, vol. 341, no. 1, pp. 23–36, 2004.

[160] W. Timp, J. Comer, and A. Aksimentiev, "Dna base-calling from a nanopore using a viterbi algorithm," *Biophysical journal*, vol. 102, no. 10, pp. L37–L39, 2012.

[161] P. M. Ashton, S. Nair, T. Dallman, S. Rubino, W. Rabsch, S. Mwaigwisya, J. Wain, and J. O'Grady, "MinION nanopore sequencing identifies the position and structure of a bacterial antibiotic resistance island," *Nat. Biotechnol.*, vol. 33, no. 3, pp. 296–300, Dec. 2014.

[162] R. Farber, *CUDA application design and development*. Elsevier, 2011.

[163] R. Longbottom, "Roy longbottom's pc benchmark collection," 2016. [Online]. Available: http://www.roylongbottom.org.uk/dhrystone%20results.htm

[164] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[165] "European nucleotide archive." [Online]. Available: https://www.ebi.ac.uk/ena

[166] "Cuda memory types." [Online]. Available: http://www.math-cs.gordon.edu/courses/cps343/presentations/CUDA_Memory.pdf

[167] M. Harris, "How to overlap data transfer in cuda c/c++," NVIDIA Developer Zone, December 2012, https://developer.nvidia.com/content/how-overlap-data-transfers-cuda-cc.

[168] "Nvidia gf100: World's fastest gpu delivering great gaming, performance with true geometric realism," NVIDIA, whitepaper.

Appendix A

GPU: Performance Optimization and Algorithms

A.1 Performance Optimization

For faster execution, parallel programming requires several optimizing methods. All of them are not controlled by the programmer. But a programmer can write a program that will utilize the optimizing approaches. Few of such methods will be discussed in the following sections.

A.1.1 Coalesced Memory Access

For better performance, it is recommended that the memory access should be coalesced. Fig. A.1 shows an example of coalesced and non-coalesced memory access. In Fig. A.1a, adjacent threads or threads in the same warp are reading/writing the data from the adjacent memory which is considered as coalesced memory access. In contrast Fig. A.1b shows the non-coalesced memory access where adjacent threads or threads in the same warp reading/writing data spreaded over the total memory space. Misaligned, discontinuous, random memory accesses serialize the program [6, 137].

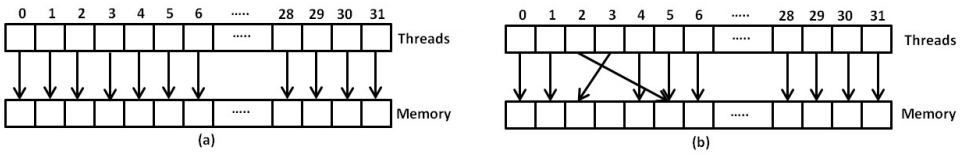


Figure A.1: Example of coalesced (a) and non-coalesced (b) memory access.

Variable Declaration	Memory	Scope	Performance Penalty
int locakVar;	register	thread	1×
int localArray[5];	local	thread	100×
__shared__ int sharedVar;	shared	block	1×
__device__ int globalVar;	global	grid	100×
__constant__ int constantVar;	constant	grid	1×

Table A.1: GPU's memory description

A.1.2 CUDA Memory Types

Table A.1 shows multiple variable declarations that resides in different types of memories [166]. The table also shows the performance penalty for different types of memory.

A.1.3 Data Transfer Parallelism / Asynchronous Data Transfer

Typically, a CUDA program sends data from host to device, executes the instructions in the device and then sends data back from the device to the host. It is suggested that instead of sending several variables' data (say three integers as three different variables), it is better to send them as a matrix or an array [6]. Apart from that, for large size of data, it is better to chop the data into several pieces and do the data transfer and execution simultaneously using asynchronous method [167].

This step also depends on the number of available copy engines. If the device has 2 copy engines (one is responsible for transferring data from host to device and another copy engine is responsible for transferring data from device to host), then it can both send and receive data to and from GPU simultaneously. But the some device has single copy engine. For those devices, that copy engine is responsible for the both way data transfers (host to device and device to host). In general, when a part of code is executed in device the main three

Copy data from host to device synchronously *Execute the instructions in device* *Copy the back to host from device synchronously*

Table A.2: Algorithm for synchronous data transfer.

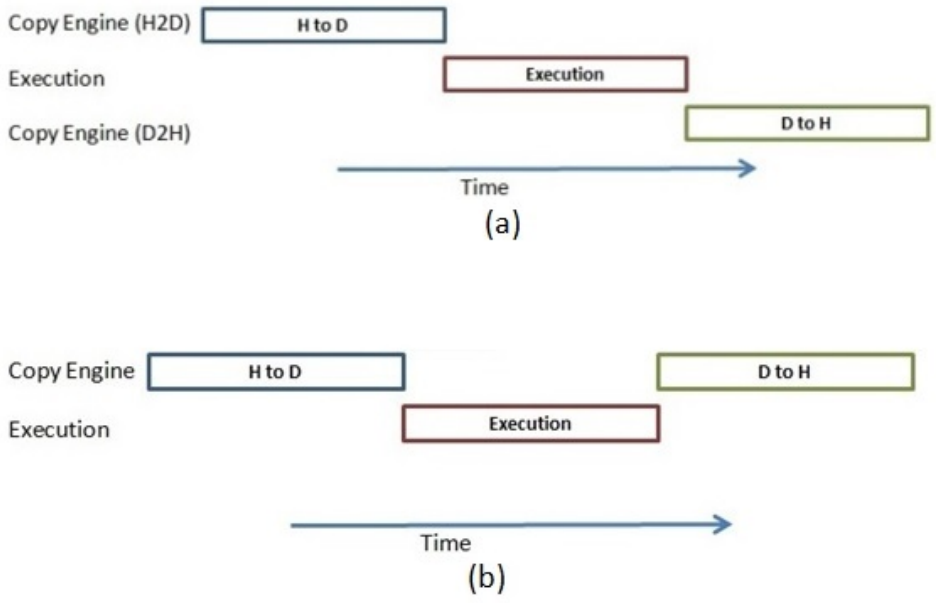


Figure A.2: Device with two (a) and one (b) copy engines and synchronous operation.

steps are shown in Table A.2.

Fig. A.2 shows how the operations are executed in time frame. In synchronous execution the pointer does not move to the next command until it finishes the current execution. In this thesis, the data are divided into groups and asynchronous data transfer is used. In asynchronous operation, the pointer will move to the next command while it is executing the current command. The device that is used for this research has single copy engine. So the code was updated for single copy engine machine and shown in Table A.3. With this update the performance of the device increases and the performance updates are shown in Fig. A.3. In Fig. A.3, it is assumed that the data is divided into 3 small groups. The blue outlined boxes represent the data transfer from the host to device. The red outlined boxes represent the parallel execution of the device codes for the 3 small sets of the data. The green outlined boxes represent the data transfer from the device to the host.

Divide the data into n pieces *For n pieces of data* *Copy data from host to device asynchronously* *End for loop* *For n pieces of data* *Execute the instructions in device* *End for loop* *For n pieces of data* *Copy the back to host from device asynchronously* *End for loop*

Table A.3: Algorithm for asynchronous data transfer for a device with single copy engine.

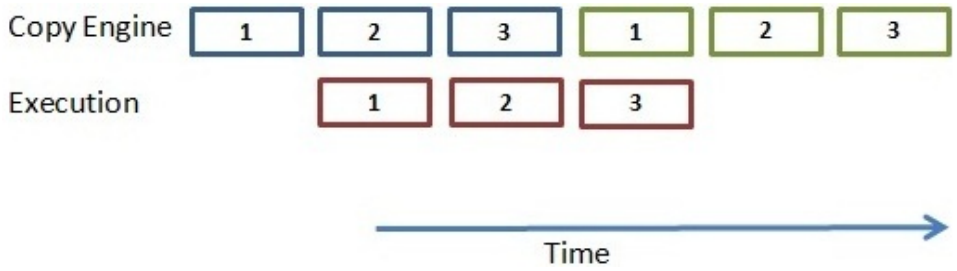


Figure A.3: Updated asynchronous operation for single copy engine.

A.1.4 Warp Schedulers Pipeline

Warp scheduler is another pipelining option which is mostly not controlled by the user. Instructions wait in the warp scheduler pipeline. And dispatch unit coupled with the warps schedulers distribute the work to the Cores or SFUs [168].

Blocks are assigned to a fix SM. Then blocks are divided into warps (a group of 32 threads). Now warps are assigned to warp schedulers without maintaining any sequence. Warp scheduler works as a pipeline. Instructions from the same warp waits in the pipeline. Instructions from of the next warp also waits in the pipeline after the current warp (in Fig. A.4, in the first row blue box warp instruction is waiting after the red box instructions which instructions are from the current warp). The Dispatch unit works as a supervisor for the warp schedulers and the execution units (cores, SFU). If any cores or SFUs are available for computation, the dispatch unit assign the warp instructions from the scheduler pipeline to the cores or SFUs depending on the type of the instruction. Note that, in the GeForce

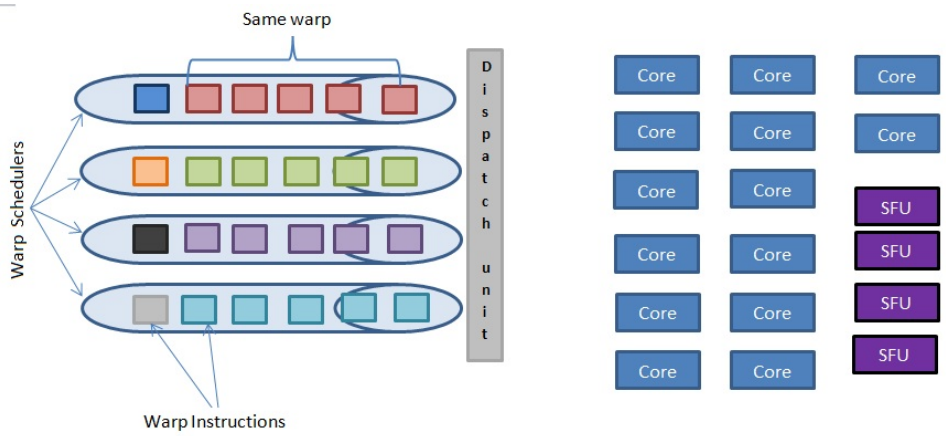


Figure A.4: Warp scheduler pipeline.

GTX 680 GPU, there is a similar pipelining option is available for SFUs.

A.2 Algorithm

This section includes two algorithms: reduction and bitonic sort that are used in this thesis.

A.2.1 Reduction

Reduction is one of the common algorithms used in device codes. Reduction is used for finding the maximum/ minimum value, summation of a large data set etc. In general, for large data set, threads responsible for the data set are distributed over the blocks. Threads in the same block can wait of each other if requires. Threads need to wait for another threads in the same block when an updated data from previous step is used. A detail example of finding maxima is included in the next paragraph.

Instead of employing a pure serial loop for finding maximum value in large dataset, the GPU can search for its maximum using a reduction method in K threads compare K number-pairs from an array of size $2K$. Applied iteratively this procedure can obtain its answer in $O(\log 2K)$ steps. An example of this is illustrated in illustrated in Fig. A.5 for $2K = 10$. In Fig. A.5, there are 10 numbers in a block and we need to find the maximum

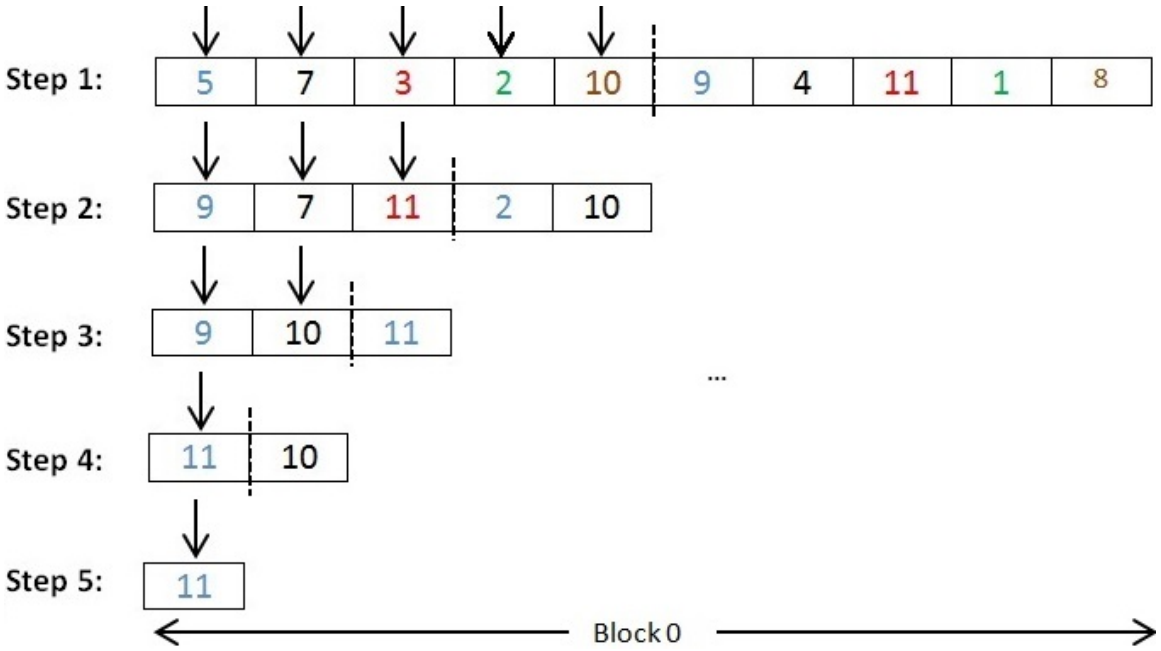


Figure A.5: Reduction method is used for searching maximum value within a block

number among them. In every step half of the threads will work. At Step 1, first 5 threads is pointing to the first 5 numbers in the row. These 5 threads will compare the pointing number to another number from the rest of the half numbers (same color is used to show which two numbers are compared). After step 1, there will be 5 numbers to consider. In the step 2, three threads will work and get 3 numbers for the next step and so on. At the final step the highest number will come to the first cell. The reduction process is executed independently in every block. Finding maximum value among the blocks requires lock option so that threads from different block will wait for each other.

A.2.2 Bitonic Sort

Bitonic sort is one of the most common parallel sorting algorithms. Fig. A.6 shows an example of bitonic sort. In every step two numbers are compared and each black straight line represents a thread. So in this example, 16 threads are used to sort a list numbers ranging from 0 to 15. The algorithm was implemented manually (no library was used) in this thesis.

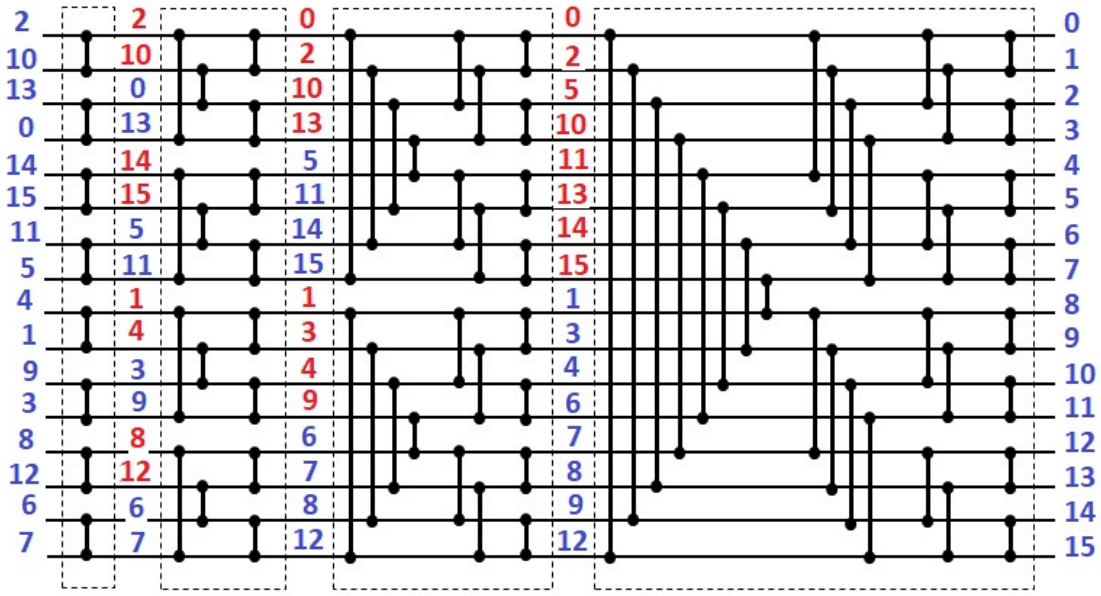


Figure A.6: An example of bitonic sort.