FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Finding Longest Common Subsequences by GPU-Based Parallel Ant Colony Optimization

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

David Markvica
Matrikelnummer 0125181

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Univ.Ass. Dipl.-Ing. Christian Schauer

Wien, 22.01.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

David Markvica
Sonnergasse 36, 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

The longest common subsequence (LCS) problem is one of the classic problems in string processing. It is commonly used in file comparison, pattern recognition, and computational biology as a measure of sequence similarity. Given a set of strings, the LCS is the longest string that is a subsequence of every string in the set. For an arbitrary number of strings the LCS problem is NP-complete. Heuristic approaches are needed to process datasets of hundreds of sequences, each thousands of character in length, that are common place in computational biology.

This master thesis presents a parallel hybrid metaheuristic combining an Ant Colony Optimization with a Local Search. The heuristic is designed from the ground up to exploit the capabilities of many-core processor architectures, such as Graphics Processing Units (GPUs). The Ant Colony Optimization constructs numerous solutions simultaneously and the Local Search employs a highly parallel enumeration to explore neighborhoods.

The algorithm was implemented using OpenCL, a framework for parallel programming of heterogeneous systems. The result is a single program that is capable of running on two different processor architectures, on CPUs and on GPUs. A number of micro benchmarks are performed to highlight the different performance characteristics of the tested architectures and to show that the algorithm scales linearly with the number of processor cores used.

Finally the implementation is benchmarked on a dataset commonly used in the LCS literature. It will be shown that the presented approach outperforms previously described methods based on Ant Colony Optimization in terms of solution quality.

Kurzfassung

Die Berechnung der Longest Common Subsequence (LCS) ist ein klassisches Problem der Stringverarbeitung, das unter anderem in der Mustererkennung und Textverarbeitung Anwendung findet. In der Bioinformatik wird die LCS als Maß für die Ähnlichkeit von DNS- und Proteinsequenzen verwendet. Von mehreren Zeichenketten soll die längste gemeinsame Teilfolge gefunden werden. Bei variabler Anzahl von Zeichenketten erweist sich das Problem als NP-schwer. Da die in der Bioinformatik auftretenden Probleminstanzen hunderte von Sequenzen mit mehreren tausend Zeichen umfassen, werden heuristische Verfahren benötigt um Instanzen dieser Größe effizient verarbeiten können.

In dieser Arbeit wird eine parallele Hybridheuristik zur Berechnung der LCS präsentiert. Die Heuristik kombiniert einen Ant Colony Optimization Algorithmus mit einer Lokalen Suche und ist für die Ausführung auf massiv paralleler Hardware (wie beispielsweise gängige Graphikkarten) optimiert. Im Ant Colony Optimization Algorithmus werden zahlreiche Lösungen zeitgleich und unabhängig voneinander erstellt und die Lokale Suche verwendet ein hoch paralleles Enumerationsverfahren um die Nachbarschaften zu erkunden.

Für die Implementierung des Algorithmus wurde OpenCL verwendet, eine Programmierschnittstelle für heterogene Parallelrechner. Das entwickelte Programm ist sowohl auf der Graphikkarte (GPU) als auch auf dem Hauptprozessor (CPU) ausführbar. In einer Reihe von Benchmark-Tests werden die Unterschiede der beiden Prozessorarchitekturen hervorgehoben und gezeigt, dass die Implementierung mit der Anzahl der Prozessorkerne linear skaliert.

Abschließend wird die Implementierung an gängigen Instanzen aus der LCS Literatur getestet. Es konnte gezeigt werden, dass die vorgestellte Hybridheuristik bessere Lösungen liefert als bestehende Verfahren, die auf Ant Colony Optimization beruhen.

Contents

CHAPTER 1

Introduction

In computational biology, sequence alignment is a fundamental technique to measure the similarity of biological sequences, such as DNA and genome sequences. A high sequence similarity often implies molecular structural and functional similarity and can be used to determine if (and how) sequences are related. Finding the longest common subsequence (LCS) is one way to measure the similarity of sequences. It was proposed by Wagner and Fischer in 1974 [81] and is, from the computer science point of view, one of the classical problems in string processing.

Given a set of strings, the LCS is the longest string that is a subsequence of every string in the set. The longest common subsequence can be obtained by deleting characters from the strings until a common substring remains. The relative positions of the remaining elements is unchanged. Figure 1.1 shows an example of 3 sequences and their LCS.

S_1 = **A**TGG**C**CCA**GG** T**GC**AGC**TG**CA G**TC**TAG**A**GAG
S_2 = GTCA**A**G**C**CTT C**GG**A**G**ACC**CT** **G**TCCC**TCA**CC
S_3 = T**AC**TACT**GG**A **GCTG**GA**TC**CG GC**A**GCCCGCC

$LCS(S_1, S_2, S_3)$ = **ACGGGCTGTCA**

Figure 1.1: Three DNA sequences and their longest common subsequence.

Table 1.1: Common structures in computational biology and their approximate size ranges. [58, 83]

Biological Data	Alphabet (Σ)	$\lvert\Sigma\rvert$	Typical Sequence Length
Protein	$A, C, \ldots, W$	20	$\sim 10^2 - 10^4$
RNA	A, C, G, U	4	$\sim 10 - 10^4$
Genome	$gene_1, gene_2, \ldots, gene_k$	$\sim 10^4$	$\sim 10 - 10^4$
DNA	A, C, G, T	4	$\sim 10^4 - 10^{11}$

Apart from being used in computational biology [6, 41, 48], the LCS has a wide variety of applications in computer science. Traditional applications can be found in data compression [75], file comparison [32], and database query optimization [69]. In recent years it has also been used for circuit area minimization in FPGA synthesis [7], document reconstruction [67], malware detection [36], and character recognition [52].

Due to its classical nature and diverse areas of application, the LCS problem is well studied and has attracted a lot of research efforts over the last 30 years. Many algorithms have been proposed for calculating the LCS efficiently but most of them focus on exact methods and/or restrict the number of strings to a fixed number, usually two. This makes them impractical when dealing with large numbers of long strings that are common place in computational biology. Heuristic methods are needed when dealing with hundreds of sequences, each thousands of character in length. Table 1.1 summarizes the properties of strings found in computational biology.

The sheer volume of data that is produced by modern genome sequencing machines each day and the exponentially expanding size of biological sequence databases makes parallel algorithms increasingly important [88]. When the computation is parallelized, larger instances can be solved by assigning more processor cores to the problem and distributing the work among them.

In recent years, graphics processing units (GPUs) have become a widely used platform for parallel computing for a number of reasons. Their theoretical processing power is larger than that of CPUs. They are more efficient than CPUs in terms of floating-point operations (FLOPS) per watt, and cheaper (FLOPS per euro) [65]. Unlike FPGAs or vector-processors they are mass market products and therefore readily available. Almost every desktop computer has a dedicated GPU card and all modern cell phones and consumer CPUs include a GPU co-processor on the processor die.

In this thesis, a parallel algorithm for calculating the longest common subsequence of multiple strings on GPUs will be presented. The thesis is structured as follows. At

the beginning a formal definition of the LCS problem will be given (Chapter 2) and related work will be presented (Chapter 3). Chapter 4 will give a brief overview of the metaheuristics that were used in the implementation of this work and other related metaheuristics. It is followed by an overview of GPU programming in Chapter 5. There the differences to CPUs will be highlighted and various programming models will be discussed with an emphasis on the programming framework OpenCL.

Chapter 6 will present the algorithm that was designed for this master thesis. It solves the longest common subsequence problem with a parallel hybrid metaheuristic combining an Ant Colony Optimization with a Local Search. The implementation of this algorithm is a single program capable of running on either CPU or GPU. This will allow meaningful comparison of the performance of the algorithm on these two architectures. The program will be tested on a series of micro benchmarks (Chapter 7) and a dataset commonly used in the literature. The thesis concludes in Chapter 8 and gives an outlook for future work.

CHAPTER 2

Problem Definition

According to [1, 17], the longest common subsequence problem can be defined as follows.

Let $A = [a_1, a_2, \ldots, a_l]$ be a sequence of l elements, i.e., a string. The elements of the string are members of a finite alphabet Σ, $a_i \in \Sigma, \forall i = 1, 2, \ldots, l$.

A sequence $B = [b_1, b_2, \ldots, b_k]$ is a **subsequence** of A, ($B \prec A$), if there exists a strictly increasing sequence of indices $[i_1, i_2, \ldots, i_k]$ such that $A[i_j] = B[j]$ holds, $\forall j = 1, 2, \ldots, k \leq l$.

Given a finite set S of n strings $S = \{S_1, S_2, \ldots, S_n\}$, C is a **common subsequence**, if $C \prec S_i, \forall i = 1, 2, \ldots, n$.

The **longest common subsequence** of S is the common subsequence of maximum length. The problem can be expressed as an optimization problem of the form

$$\begin{aligned} &\text{maximize} && |C| \\ &\text{subject to} && C \prec S_i,\ \forall i = 1, \ldots, n. \end{aligned} \tag{2.1}$$

with $|C|$ being the length of the common subsequence C. The longest common subsequence does not have to be unique, i.e., there can be more than one common subsequence with maximum length.

2.1 Complexity

In case of two strings and an alphabet of fixed size, an exact solution to the longest common subsequence problem can be found in polynomial time using dynamic programming. Bergroth et al. [1] did an extensive comparison of algorithms for the longest common subsequence problem with two strings (LCS2). In this case the time complexity is $O(l_1 l_2)$, with l_1 and l_2 being the length of the strings.

Maier [55] and Paterson et al. [66] showed that for an arbitrary number of strings the LCS problem is NP-complete, even with binary alphabet. NP-complete means that there is no known algorithm that can compute the optimal solution of every input instance in polynomial time. Given a solution, it is however possible to check the correctness of this particular solution in polynomial time [22].

CHAPTER 3

Related Work

Over the last 30 years many algorithms that compute the LCS of multiple strings have been proposed. Most of the earlier work focuses on exact methods; the majority is based on dynamic programming. Due to the high complexity of the problem these approaches scale badly and are not applicable when dealing with large-sized datasets that became common in recent years. Especially when the number of strings to compare grows, exact approaches reach their limits fast. Focus has now shifted towards heuristic approaches that can deal with the hundreds of gigabytes of data that are generated by genome sequencing machines each week [89]. Approximative and heuristic algorithms have been developed that find reasonably good solutions in a short amount of time.

3.1 Algorithms Based on LCS2

The various algorithms used to compute the LCS of two strings can be extended to handle multiple strings. The *greedy* and the *tournament* algorithms [71] find the LCS by comparing strings pairwise. Both algorithms have time complexity O(n^2l^2), with n being the number of strings and l being the string lengths.

The *greedy* algorithm chooses two strings S_1 and S_2 that yield the longest LCS of all string pairs (S_i, S_j). The algorithm removes strings S_1 and S_2 from the set of strings and adds their longest common subsequence $LCS2(S_1, S_2)$ as a new string. The algorithm then proceeds recursively:

$$Greedy(S_1, S_2, \ldots, S_n) = Greedy(LCS2(S_1, S_2), S_3, \ldots, S_n)$$

The *tournament* algorithm is similar to the *greedy* one but combines $\left\lfloor \frac{n}{2} \right\rfloor$ pairs in each recursive step:

$$Tournament(S_1, S_2, \ldots, S_n) \\ = Tournament(LCS2(S_1, S_2), LCS2(S_3, S_4), \ldots, (LCS2(S_{n-1}, S_n))$$

3.2 Exact Approaches

Many exact algorithms for solving the LCS problem are based on dynamic programming, see for example Hirschberg [28] or Irving and Fraser [35]. They have a time complexity of O(l^n) which means they are exponential in the number of strings. Over the years various improvements have been proposed and the time complexity has been reduced to O(l^{n-1}). One improvement that is often used in practice is reducing the search space by pre-computing dominant points [82].

Parallel implementations of exact approaches were developed, like the *FAST_LCS* by Chen et al. [15] and the current state-of-the-art algorithm by Wang et al. [83]. By parallelizing the computation, larger instances can be solved by assigning more processor cores to the problem and distributing the work among them. In practice these implementations are applicable for finding the LCS of around ten strings ($n = 10$), with more strings their run-time becomes impractical.

A different exact approach is to use integer linear programming techniques as proposed by Singireddy in his master thesis [73]. The complexity remains O(l^n).

3.3 Approximation Approaches

Approximation algorithms find near-optimal solutions to NP-hard problems within affordable time [17]. Unlike heuristics (Chapter 4) they have provable guarantees for solution quality and run-time.

The first approximation algorithm for the LCS was *Long Run* [40]. *Long Run* constructs the longest string containing only a single character that is a valid subsequence in all strings. For each character $a \in \Sigma$, let c_a be the minimum number of occurrences of a in all strings $S_1, S_2, \ldots, S_n$. *Long Run* returns a string of character $\alpha \in \Sigma$ of length c_α where c_α is maximal. Because its solution is restricted to strings of a single character it is not good enough for practical use.

The approximation ratio gives the factor by which the length of the guaranteed solution is at most smaller than the optimal solution. Therefore, a ratio of 1 would describe an exact approach. *Long Run* has an approximation ratio of $|\Sigma|$, according to [77], which makes it not very useful in practice, even on datasets with small alphabets such as DNA sequences ($|\Sigma| = 4$).

The *Expansion Algorithm* by Bonizzoni et al. [5] and the *Best Next for Maximal Available Symbols* (BNMAS) algorithm [31] are approximation algorithms that are not restricted to single character solutions. Still, they guarantee the same approximation ratio as *Long Run*.

3.4 Heuristic Approaches

Like approximation algorithms, heuristic algorithms are used to find near-optimal solutions to NP-hard problems. They differ from approximation algorithms in the way that heuristics cannot provide guarantees regarding the solution quality or run-time. Their focus is on finding a solution that is "good enough" quickly. An overview of commonly used heuristic solution techniques is given in Section 4.

Hinkemeyer and Julstrom [27, 43] proposed a *Genetic Algorithm* (GA) for computing the LCS of an arbitrary number of strings. They compared the algorithm to Irving and Fraser's dynamic programming approach [35] and reported shorter runtime for the GA. However, their results could not be reproduced by Jansen and Weyland [39], who did a theoretical analysis of the performance of evolutionary algorithms for the LCS problem.

In his master thesis Chiang [16] proposed a different GA, but its performance was below the state-of-the-art approach of this time, an *Ant Colony Optimization* (ACO) by Shyu and Tsai [72]. Weng et al. [86] created a hybrid algorithm based on Chiang's GA and Shyu and Tsai's ACO and reported a slight improvement in run-time and solution quality over the ACO. Shyu and Tsai's ACO will be described in detail in Section 6.1.

One heuristic that has often been applied successfully to the LCS problem is *Beam Search*. The first use of *Beam Search* for the LCS was in 2007 by Blum and Blesa [3]. Their *Probabilistic Beam Search* produced solutions that were better than or similar to the *Expansion Algorithm* in much shorter time. An improved version of the Beam Search that added an additional method for pruning the search space was presented by Blum et al. in 2009 [3] and outperformed all other approaches (including *Expansion Algorithm* and Shyu and Tsai's ACO) in terms of solution quality and run-time.

In 2010 Blum et al. [2, 4] presented a hybrid algorithm that combines Beam Search

with Ant Colony Optimization. The so called *Beam-ACO* uses Beam Search as the construction function of the ACO. The *Beam-ACO* provides better solutions than Beam Search alone, especially when applied to DNA sequences. Its run-time depends on the size of the alphabet; for DNA sequences ($|\Sigma| = 4$) it is worse than plain Beam Search, for protein sequences ($|\Sigma| = 20$) it is faster.

Beam search is also used to generate the initial solution in Lozano and Blum's low-level integrative hybrid metaheuristic [54]. This hybrid algorithm uses a Variable Neighborhood Search (VNS) that applies an iterated greedy algorithm and a greedy randomized procedure in its improvement phase. In each iteration of the VNS a large contiguous part of the solution (up to 10 percent) is deleted and re-constructed by the greedy randomized procedure. The solution quality of this approach is similar to the *Beam-ACO* and the authors hint at further work combining the two; using the *Beam-ACO* as a construction heuristic for the VNS.

Other recently published heuristics are the *MLCS-APP* by Wang et al. [84], which is based on the *A** search algorithm, and the *Deposition and Extension* algorithm by Ning [60]. Their solution quality is on par with the 2009 Beam Search from Blum et al. [3] but they are out-classed by later work.

The *Pro-MLCS* algorithm [89] was presented by Yang et al. in 2013 and is a parallel progressive algorithm. This approach can return an approximate solution quickly and progressively generates better solutions until it reaches the optimal one. In their work the authors present a shared-memory and a distributed-memory version of the algorithm. The parallel implementation achieves near linear speedup with long strings and retains the progressiveness property; it finds better solutions in a shorter time when given more hardware resources. Given the same time to execute, its solution quality is comparable to that of *MLCS-APP*.

The current state-of-the-art heuristic is the *Hyper Heuristic* presented by Tabataba and Mousavi in [77]. At its core the Hyper Heuristic uses a Beam Search with two different candidate heuristic functions. Neither of the heuristic functions alone has a clear advantage over the other in all experimentally tested cases, so the Hyper Heuristic uses both. It applies the functions with a small beam size to find out which of the candidate heuristics performs better on this particular instance. Because of the small beam size this initial step executes quickly. Then the candidate heuristic that yields better results is executed a second time, this time with the full beam width.

CHAPTER 4

Heuristic Solution Techniques

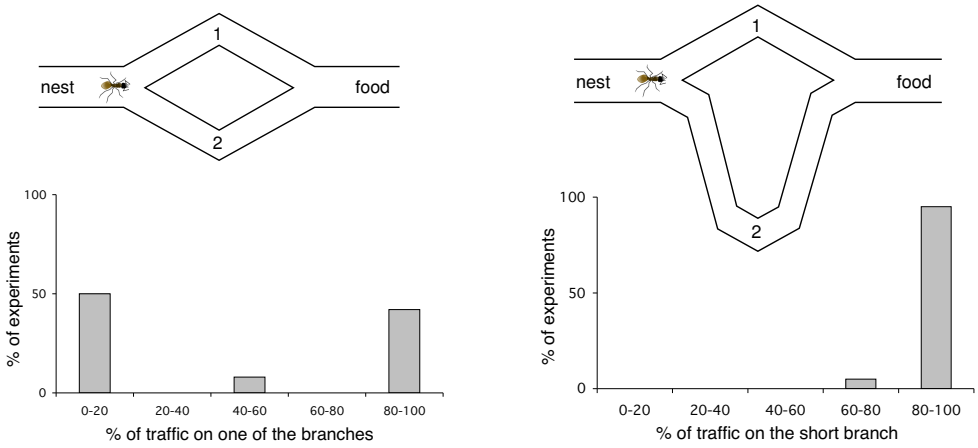
Metaheuristics are designed to find near-optimal solutions fast by exploring large solution search spaces efficiently. Therefore, they are often used for solving hard optimization problems. Metaheuristics are not problem-specific, they make few assumptions about the underlying problem and can be applied to a wide variety of optimization problems.

This chapter will give a brief overview of the metaheuristics that were used in the implementation of this work (Chapter 6) and other related metaheuristics. For a more detailed introduction to metaheuristics see [78].

4.1 Ant Colony Optimization

Ant Colony Optimization (ACO) is a metaheuristic inspired by the foraging behavior of ants. Ants are simple, identical individuals, working together as a highly structured social organism [19]. They have no way of direct communication and are bad sighted, some sub-species even completely blind [30]. Still, ants as a collective are able to accomplish complex tasks such as finding the shortest path between a food source and their nest.

Ants communicate indirectly through pheromones they deposit on the ground. This mechanism of indirect coordination between agents via modification of their environment was termed *stigmergy* by Grassé [24], who observed it on termites. The amount of pheromones an ant deposits depends on the length of the path it travelled and the amount of food it discovered. Pheromones act as a chemical trail that guides other ants, which



(a) Branches of equal length (b) Branches of different length

Figure 4.1: The double bridge experiment. When the branches are of equal length (a), the ants use one branch or the other in approximately the same number of experiments. When the branches are of different length (b), the ants converge towards the shorter branch in the majority of the experiments. [21, 53]

in turn deposit additional pheromones, intensifying and reinforcing the trail. Over time pheromones evaporate, which allows ants to "forget" previous trails and explore new ones.

The process of finding the shortest path by real ants has been highlighted by the *double bridge experiment* (Figure 4.1) by Deneubourg et al. [19, 23]. The first experiment connects the ants' nest to the food source by two bridges (branches) of equal length (Figure 4.1(a)). In the beginning, when there are no pheromones, ants choose both branches with an equal likelihood. Due to random fluctuations, a few more ants would choose one branch over the other leading to an increase of pheromones deposited on this branch. This creates a positive feedback loop and soon almost all ants use only one branch.

In the second experiment, the branches connecting the nest and the food source are of different length, one being twice as long as the other (Figure 4.1(b)). The initial lack of pheromones leads the first ants again to choose randomly between the two branches on their way from the nest to the food source. Over time the shorter branch accumulates more pheromones than the longer as it allows ants to travel faster and therefore more frequently. Eventually almost all ants use the shorter branch. In both experiments there is a small percentage of ants that ignores the established pheromone trail used by the majority of ants and "explores" the other branch.

Algorithm 4.1: Ant Colony Optimization

```
1 initialize pheromone trails
2 while termination condition not met do
3 |   foreach ant do
4 |   |   construct ant solution using pheromone trail information
5 |   end
6 |   Local Search // optional
7 |   update pheromone trail
8 end
```

The *Ant System* proposed by Dorigo in his PhD thesis [20] in 1992 was the first ant-inspired algorithm for solving a combinatorial optimization problem. Similar to how the shortest path problem is solved by real ants, Ant System uses artificial ants to solve the traveling salesman problem. This technique was later generalized into a metaheuristic, called Ant Colony Optimization [21].

Algorithm 4.1 shows the template algorithm for Ant Colony Optimization. First the pheromone trails are set to an initial value (line 1). Unlike it is the case with real ants, the initial pheromone value τ_0 of the trails in an ACO is not necessarily zero. Within the inner loop (lines 3-5), each ant constructs a complete solution to the problem using the pheromone information and a heuristic function.

The solution construction (line 4) can be seen as a stochastic greedy procedure. Starting from an initially empty solution, solution components are added until a complete solution is derived. In each step the next component i is chosen from a set of candidate components $i \in Cand$ with respect to a probabilistic transition rule:

$$p_i = \frac{[\tau_i]^\alpha \cdot [\eta_i]^\beta}{\sum_{j \in Cand} [\tau_j]^\alpha \cdot [\eta_j]^\beta} \tag{4.1}$$

The pheromone factor τ_i represents the past experience of choosing component i as part of a solution. This value will change over time as pheromones are deposited and evaporate. The heuristic factor η_i evaluates the attractiveness of adding component i to the solution by a greedy procedure. The definition of the pheromone and the heuristic factor is highly dependent on the problem and the chosen pheromone structure. Parameters α and β are used to control the balance of the influence of pheromone and heuristic factors in the transition probability p_i. A higher value for α increases the importance of the pheromone factor, favoring exploitation. Exploration of the search space is encouraged by a higher value for β.

Pheromone update (line 7) is done in two steps. In the first step, all the existing pheromone trails are decreased by multiplying them by a factor of $1 - \rho$, with ρ being the evaporation rate:

$$\tau_i \leftarrow (1 - \rho)\tau_i, \quad \rho \in [0, 1] \tag{4.2}$$

Pheromone evaporation allows ants to forget older decisions and to focus on recent constructions.

In the second step of pheromone update, pheromones $\Delta\tau_i^k$ are deposited, giving positive feedback:

$$\tau_i \leftarrow \tau_i + \Delta\tau_i^k \tag{4.3}$$

$$\Delta\tau_i^k = \begin{cases} c_k/c^* & \text{if ant } k \text{ uses component } i \text{ in its solution} \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

Pheromones are deposited on the components that are part of the solutions that were found by the ants. This increases the probability of selecting a component in later iterations of the algorithm.

The amount of pheromone $\Delta\tau_i^k$ deposited is proportional to the quality of the solution c_k found by the k-th ant. It is usually normalized between 0 and 1 by defining it as a ratio between the value c_k and the optimal solution c^*. If the optimal solution is unknown, an estimated upper bound or the best value found so far can be used as c^*.

Different strategies for selecting the ants that participate in the pheromone depositing step have been proposed:

- In Dorigo's original **Ant System** algorithm, all ants deposit pheromones for the solutions generated in the current iteration.
- Dorigo also proposed the first improvement to the Ant System, the **Elitist Ant System**. In addition to the solutions of the current iteration, the best solution found since the start of the algorithm deposits pheromones as well. This reinforces

the best solution and was shown to find better solutions in a lower number of iterations in computer simulations [20].

- The **Rank-based Ant System** proposed by Bullnheimer et al. [10] orders the ants according to their solution quality, assigning them a rank. The amount of pheromones deposited decreases with the rank of an ant. Only a limited number of ants, the ω highest ranked ones, and the best solution found so far deposit pheromones.
- $\mathcal{MAX} - \mathcal{MIN}$ **Ant System** as proposed by Stützle and Hoos [76] strongly exploits good solutions. Only a single solution deposits pheromones, either the best solution found so far or the best solution found in this iteration. To avoid early convergence and stagnation the pheromone trail values are limited to an interval $[\tau_{min}, \tau_{max}]$. This reduces the relative differences between pheromone trails and ensures good diversification.

4.2 Genetic Algorithms

Genetic Algorithms [29] have been developed by Holland in the 1970s. Like ACO they are population-based and were inspired by nature. They simulate the evolution of species through survival of the fittest ("selection"), sexual reproduction ("recombination"), and random changes in the DNA ("mutation"). One main application for Genetic Algorithms is the field of combinatorial optimization. Algorithm 4.2 shows the general structure of a Genetic Algorithm based on [78].

Algorithm 4.2: Genetic Algorithm

```
1  generate P(0)
2  t ← 0
3  while termination condition not met do
4  |  evaluate P(t)
5  |  P'(t) ← select(P(t))
6  |  recombine P'(t)
7  |  mutate P'(t)
8  |  evaluate P'(t)
9  |  P(t+1) ← replace (P(t), P'(t))
10 |  t ← t+1
11 end
```

A population of candidate solutions is usually randomly generated, see Algorithm 4.2 line 1. In each iteration the fitness of all individuals in the current population is evaluated (line 4). Fitness is a measure for the quality of the solution represented by the

individual. A subset P' of the population is selected and used as a starting point for the next generation (line 5). Usually two parent individuals from the population P' are taken to create one or two descendants, which are then added back into the population P'. Fitter individuals have higher probability of being chosen to advance to the next generation. New individuals are created by recombining existing individuals (line 6). Descendants share some characteristics of their parents. Mutations, i.e., small localized changes to the solution are applied to some individuals in the population P' with a certain probability (line 7). This guarantees diversity and avoids pre-convergence to a local optimum, in which the whole population becomes too homogenous. At the end of an iteration a replacement scheme is applied to determine, which individuals are passed on to the next iteration (lines 8-9). The algorithm stops, when the termination condition is met (line 3). Possible termination conditions could be reaching a predefined number of iterations or exceeding a given time limit. For a more detailed introduction to Genetic Algorithms see [29].

4.3 Local Search

Local Search (Algorithm 4.3) is a single-solution based metaheuristic intended to find the local optimum within a given neighborhood. A neighborhood structure N is a function $N : S \rightarrow 2^S$ mapping each solution $s \in S$ with S being the set of feasible solutions to a set of neighbors, the so-called neighborhood $N(s)$ of s. The neighborhood definition depends strongly on the optimization problem and its representation. Its main characterizing property is locality: small changes made in the representation must result in small changes in the solution.

Algorithm 4.3: Local Search

```
1 generate a start solution s
2 define a neighborhood structure N(s)
3 while termination condition not met do
4 |   choose s' ∈ N(s) according to step function
5 |   if s' is better than s then
6 |   |   s ← s'
7 |   end
8 end
```

Local Search starts with a candidate solution s and iteratively moves ("walks") from one solution to another in the solution space. In each step a number of related solutions according to the neighborhood $N(s)$, i.e., the neighbors of s, are examined, see line 4. If the chosen solution $s' \in N(s)$ is an improvement over s, it is accepted as the new

current solution, see lines 5-7. The local optimum is found, when no solution $s' \in N(s)$ yields an improvement over s.

Selecting a new neighbor s' is done via a step function. The step function can pursue one of the following strategies:

- **First improvement** chooses the first neighbor that is better than the current solution.
- **Best improvement** evaluates all possible neighbors in $N(s)$ and chooses the one that yields the best improvement to the current solution.
- **Random selection** selects a random neighbor.

With *first improvement* or *best improvement* as the step function, the algorithm can reach the point where no further improvement can be made because the local optimum is reached; this can be used as termination condition (line 3). When *random selection* is used, it is not possible to state whether the local optimum is reached. Therefore, common termination criteria are exceeding a time limit, reaching a predefined number of iterations or not improving the best solution for a given number of iterations.

4.3.1 Variable Neighborhood Descent

Variable Neighborhood Descent [26] (Algorithm 4.4) defines a deterministic search through multiple neighborhood structures. Different neighborhood structures have different local optima. Thus the search space can be enlarged according to the number of neighborhoods. Because searching through multiple neighborhoods finds the local optimum according to all those neighborhoods. The set of neighborhood structures $N_{1 \leq i \leq k_{max}}$ is ordered and the algorithm iterates over the neighborhood structures one after the other. If an improvement was found a new current solution is chosen and the search is restarted within the first neighborhood, see lines 6-8. Otherwise the next neighborhood is searched for an improvement. Therefore, the ordering has a big impact on performance and commonly the neighborhood structures are ranked in increasing order of their size. Since it is a deterministic search, only *first improvement* and *best improvement* are valid step functions.

Algorithm 4.4: Variable Neighborhood Descent

generate a start solution s
define neighborhood structures $N_i, i = 1, 2, \ldots, k_{max}$
$k \leftarrow 1$
while $k \leq k_{max}$ **do**
    choose $s' \in N_k(s)$ according to step function
    if s' *is better than* s **then**
        $s \leftarrow s'$
        $k \leftarrow 1$
    else
        $k \leftarrow k + 1$
    end
end

Algorithm 4.5: Variable Neighborhood Search

generate a start solution s
define neighborhood structures $\mathcal{N}_i, i = 1, 2, \ldots, l_{max}$
while *termination condition not met* **do**
    $l \leftarrow 1$
    while $l \leq l_{max}$ **do**
        choose s' randomly from $\mathcal{N}_l(s)$
        $s'' \leftarrow$ LocalSearch(s') **or** VariableNeighborhoodDescent(s')
        if s'' *is better than* s **then**
            $s \leftarrow s''$
            $l \leftarrow 1$
        else
            $l \leftarrow l + 1$
        end
    end
end

4.3.2 Variable Neighborhood Search

Variable Neighborhood Search [26] (Algorithm 4.5) is a stochastic search method based on multiple neighborhood structures to escape local optima. Similar to the Variable Neighborhood Descent it iterates through a set of neighborhood structures $\mathcal{N}_i$. At each iteration it performs three steps: shaking (line 6), local improvement (line 7), and move (lines 8-13). A new neighbor s' of the best known solution s is chosen randomly according to the current neighborhood structure $\mathcal{N}_i$, i.e., s is "shaken" to generate a new starting point for the Local Search or VND, respectively. When the local optimum s'' found is not an improvement over s, the algorithm moves to the next neighborhood $\mathcal{N}_{i+1}$. If s'' marks a better solution, the search starts again in the first neighborhood $\mathcal{N}_1$ with s'' being the new best solution s. The search stops when the termination condition, for example a time limit or a given number of iterations, is reached (line 3).

4.4 Hybrid Metaheuristics

Metaheuristics have been applied to a wide variety of real-life and theoretical optimization problems providing good results. Hybrid algorithms yield further improvements by combining a metaheuristic with other techniques. For many optimization problems hybrid approaches represent the current state-of-the-art.

In [78] four different possible combinations for metaheuristics are discussed:

- **Combining metaheuristics with other metaheuristics**: Population based metaheuristics are well suited for exploring a large search space (i.e., diversification). Single-solution based metaheuristics on the other hand start from a given solution and exploit the neighborhood in the search space (i.e., intensification). Combining the two results in a search algorithm that is balanced between intensification and diversification of the search.

- **Combining metaheuristics with exact methods**: Exact methods such as linear programming techniques and branch and bound can be used to calculate upper and lower bounds for solutions, thereby reducing the search space of the metaheuristic. Moreover, partial solutions from exact methods can be used as initial solutions for metaheuristics. Exact methods can also be used to fully explore larger neighborhoods in single-solution based metaheuristics, trading efficiency for effectiveness.

- **Combining metaheuristics with constraint programming** [74]: Constraint programs specify in a declarative way the structure of a feasible solution for a given

problem. A solver for constraint programs can be used in the same way as exact methods to generate partial solutions and bounds. Constraint programs are well suited to express transformations of solutions, such as the recombination and mutation in Genetic Algorithms (Section 4.2).

- **Combining metaheuristics with machine learning**: Techniques from data mining and machine learning can be used to gather knowledge from the history of the search. Acquired positive and negative knowledge guides the search of the metaheuristic and improves its efficiency and effectiveness.

Furthermore, hybrid algorithms can be categorized in a coordinate system along two axis: *low-level/high-level* and *relay/teamwork*. Two algorithms can be combined in a way that one algorithm implements a function of another algorithm. This is called low-level hybridization. In high-level hybrid algorithms the algorithms remain separate and self-contained. In relay hybridization the algorithms run in sequence, using the output of the previous as input for the next algorithm. Cooperating algorithms that run in parallel are called teamwork hybrids.

High-level relay hybrids are often used in practice. A common hybridization is using a greedy heuristic or an ACO to create the initial population for a Genetic Algorithm. High-level relay hybrids are also used to further exploit solutions (i.e., intensification of the search) and finding local optima (e.g., VND, Local Search) within metaheuristics (e.g., GA, ACO) that specialize in making good global decisions (i.e., diversification of the search).

When, as it is the case for this thesis, a population based metaheuristic (ACO) is combined with a single-solution based metaheuristic (Local Search), the latter can be applied to:

- **The whole population:** This guarantees to find the local optimum within the whole population but comes at a great computational cost.
- **The best solution of the population:** This approach is fast, but the best solution within the population need not necessarily provide the best starting point for the local improvement.
- **A subpopulation:** This is a compromise between the other two approaches but raises the problem of finding a good subpopulation.

CHAPTER 5

Graphics Processing Units

Graphics Processing Units (GPUs) have become a compelling platform for computationally expensive tasks. In recent years their capabilities expanded beyond that of special purpose graphics accelerators and they are now usable for scientific computing. Critical factors for this advancement were the addition of IEEE-compliant floating-point operations and support for error correcting code (ECC) memory [59]. The key strength of GPUs compared to CPUs is in raw performance. The theoretical peak performance of GPUs is an order of magnitude larger than that of CPUs and the gap continues to widen (Figure 5.1).

While GPUs are now capable of general purpose computation, they do require the use of new special purpose programming models. Their underlying architecture is vastly different from CPUs. Therefore, programs and algorithms have to be redesigned to exploit their potential performance.

Application areas where GPUs have been used successfully tend to have the following characteristics [64]: Their computational requirements are large, parallelism is substantial, and throughput is more important than latency. Typical examples (besides applications in the field of computer graphics) are physics simulations, image processing applications, and statistical modeling.

The remainder of this chapter gives a brief overview of the architecture of state-of-the-art GPUs and explains programming paradigms.

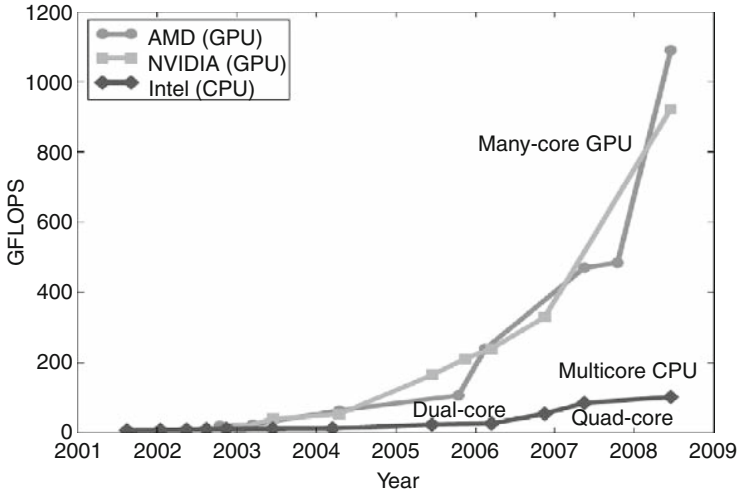


Figure 5.1: Peak performance in gigaFLOPS (billion FLoating-point Operations Per Second) for CPUs and GPUs. [46, 65]

5.1 Architecture

The architecture of modern GPUs is very different from that of modern CPUs mainly because their goals are different. CPUs are designed to make the execution of single-threaded programs as fast as possible. They employ sophisticated branch prediction and speculative execution units that try to guess, what instructions might be executed next. Large caches are used to mitigate memory access latencies by storing recently used data, assuming it will be used again. All of this control logic and cache memory takes up a huge percentage of the die area of modern CPUs, while the actual computing units, ALUs (Arithmetic Logic Units) and FPUs (Floating-Point Units), account for only a fraction of that. Since speculative control logic and caches do not contribute to the peak performance of the chip, it has been argued [62] that they are not the most efficient use of processor die space.

GPUs, on the other hand, started out as accelerator chips for computer graphics. Tasks such as texture mapping, polygon rendering, and vertex processing are inherently parallel. Adding more cores to do work independently is a natural way of increasing performance. GPUs are designed for maximum floating-point performance and use most of their transistors for computation instead of caching and speculative execution. Another advantage of GPUs is that chip designers are not bound by backwards compatibility; there is no standardized instruction set like x86 on desktop PCs, which allows radical changes from one generation to the next.

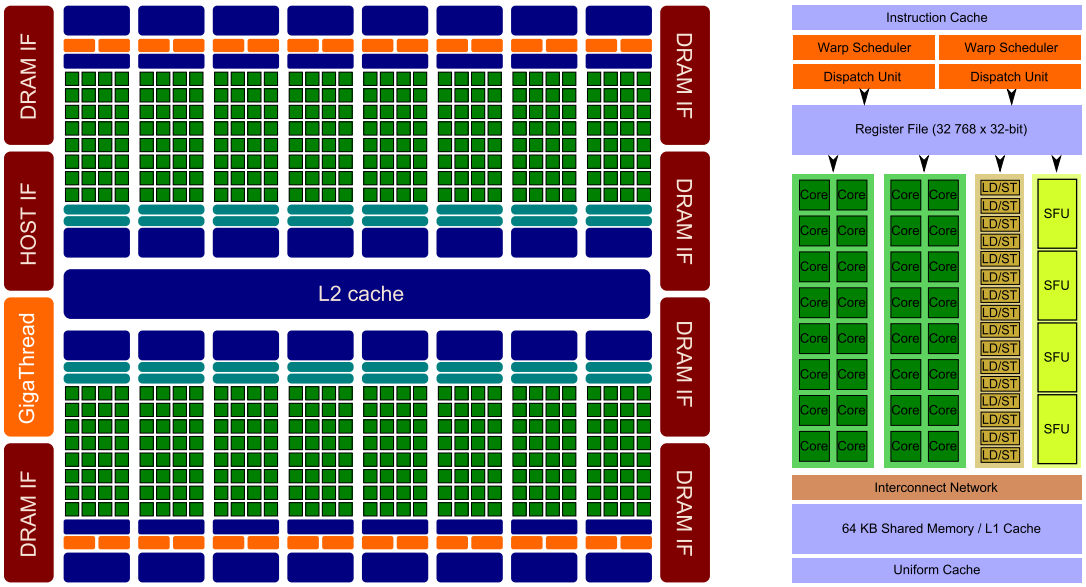


Figure 5.2: NVIDIA's Fermi GPU architecture. A GPU consisting of 16 streaming multiprocessors, a shared L2 cache and interfaces to the host and onboard memory is shown in (left). A single streaming multiprocessor is shown in (right). [8]

Figure 5.2 shows a high-level block diagram of a modern GPU architecture, the Fermi by NVIDIA [8, 87]. The GPU consists of 16 *multiprocessors*, which are similar to the processor cores found in CPUs. Each multiprocessor has 32 *streaming processors*, 16 load/store units and 4 *special function units* that handle intrinsic instructions such as sine, square root or interpolation. A single streaming processor has a full 32-bit precision integer arithmetic logic unit (ALU) and an IEEE 754-2008 compliant floating-point unit (FPU), which allows the execution of one 32-bit integer or floating-point operation per clock cycle.

Multiprocessors execute instructions in an SIMD (Single Instruction Multiple Data) way, i.e., the same instruction is executed simultaneously for multiple different data elements. Fermi is a so-called 32-wide SIMD architecture and therefore executes an instruction on 32 data elements at once before moving to the next instruction. The instruction itself can take multiple clock cycles to complete, because there are for example fewer than 32 load/store units.

Executing instructions in an SIMD fashion has an impact on the way branches are handled. When the thread of execution diverges within a program, the multiprocessor has

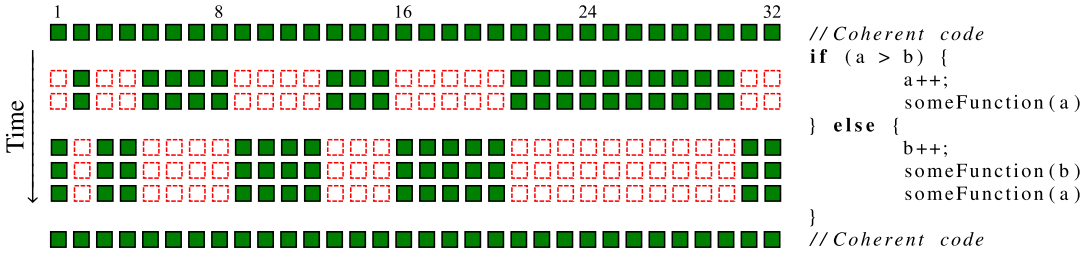


Figure 5.3: Handling a branch on a 32-way SIMD GPU: Elements for which the current branch condition does not hold, are masked out (shown as dashed boxes) and are not affected by the computation. [8]

to execute both parts of a branch one after another (Figure 5.3). The elements, which are not part of the currently executed branch, are masked out. Therefore, they are not affected by the current computation. In the worst case, the execution of all 32 elements diverges, which slows down execution by a factor of 32, compared to the ideal case of all elements executing the same instruction at once.

Multiprocessors have hardware support for hundreds of threads and execute them in a time-sliced fashion. In the absence of large caches, a large number of ready to run threads is used to hide memory access latency. When one thread stalls on a memory fetch, the multiprocessor simply switches to the next one.

5.2 Programming Models

Early experiments on graphics hardware for non-graphics computational tasks were done using OpenGL [70] directly. OpenGL is an API designed for realtime 2D and 3D graphics processing. Therefore, all programs have to be expressed in terms of operations on graphical primitives.

Larsen and McAllister [50] showed in 2001 how matrix multiplication could be done on consumer graphics hardware. They encoded matrix elements as colors of two-dimensional textures and used texture blending to perform arithmetic on them. Due to limitations of the hardware at that time they could only operate on fixed point numbers of 8-bit precision. Thompson et al. [79] showed that, for large matrices (1500×1500 elements), the GPU outperformed the CPU by a factor of 3.2.

In 2004 higher-level programming languages started to emerge, among them *Brook* [9]

and *Sh* [56], that tried to abstract from the graphics part but still exposed many implementation details of the underlying graphics hardware. There was only limited use for those languages but they influenced later languages.

The first widely used general purpose programming languages for GPUs are CUDA (Compute Unified Device Architecture) [68] and OpenCL (Open Computing Language). CUDA was developed by NVIDIA and draws many ideas from *Brook*. It is proprietary technology and works exclusively on GPUs produced by NVIDIA. Despite this CUDA is widely used in the industry and in research [33]. OpenCL was created by Apple and has since become an open standard. The programming model of the two is quite similar and almost all features in one language have a corresponding equivalent in the other. The biggest difference is that CUDA targets GPUs only whereas OpenCL supports heterogenous computing on a broad spectrum of different types of processors.

A wide variety of libraries and domain specific languages build on top of OpenCL and CUDA. *Accelerate* [13] for example is an embedding in Haskell and provides a purely functional array language that is compiled to CUDA, OpenCL and multi-core CPUs. Efforts were made to embed array languages in the dependently typed programming language *Agda* [14]. This would provide static bounds checks at compile time and eliminate a very common source of bugs entirely.

5.2.1 OpenCL

OpenCL [25] is a framework for programming heterogenous computing platforms. It was initially developed by Apple Inc. In 2008 a first proposal was submitted to the Khronos Group, which is an international consortium that manages open standards such as OpenGL. The Kronos Group has since published three versions of the OpenCL standard: version 1.0 in 2008, version 1.1 in 2010, and version 1.2 in 2011.

The goal of OpenCL is to provide a uniform abstraction for programming all the different processors found in modern computers, including CPUs, GPUs, DSPs and dedicated accelerators (like the Cell BE [44] that was used in the Playstation 3 and IBM's Roadrunner supercomputer). There are efforts to compile OpenCL programs directly into hardware circuits designs, which can then be synthesized and uploaded onto FPGAs [37, 63].

The OpenCL framework consists of two parts, the OpenCL API and the OpenCL programming language, which are described in the following sections.

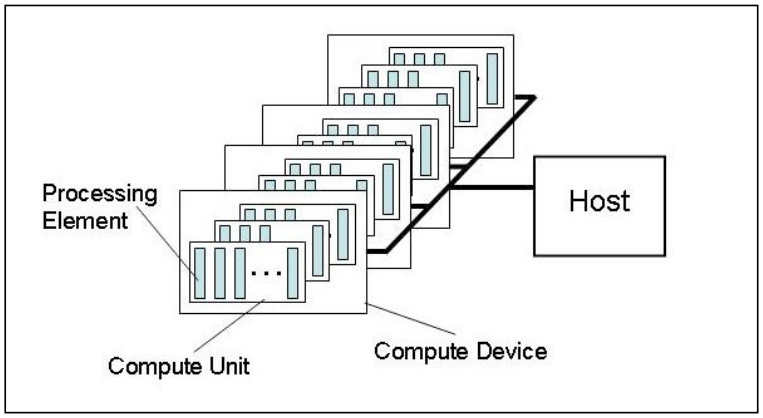


Figure 5.4: The OpenCL platform model. [25]

The OpenCL API

An OpenCL application consists of two parts: a program that runs on the CPU (the *host*) and a part of the program that runs on the OpenCL devices. The program that runs on the host can be written in any programming language that provides OpenCL bindings whereas the part of the program that runs on the OpenCL devices must be written in the OpenCL Programming Language. The host fulfills organizing functions and only few computational intensive tasks are done by the host.

The host queries for available OpenCL devices and adds them to a *context*. A device can be a GPU, a CPU or any other OpenCL compatible processor (Figure 5.4). Each *compute device* has one or more *compute units*. A compute unit is analogous to a processor core on a CPU or a multiprocessor in a GPU. Compute units have local memory and one or more *processing elements*. Processing elements do the actual computation, they correspond to streaming processors on GPUs.

The execution of instructions of the processing elements within a compute unit depends on their underlying architecture. They either execute in an SIMD or SPMD (Single Program Multiple Data) fashion, where each processing element has its own program counter and executes the same program independently. With SPMD the actual sequence of executed instructions can diverge within a compute unit, and as a result, branches and loops do not have the same impact on performance as they do in SIMD (Section 5.1).

The part of the application that will be executed on devices is written in the OpenCL Programming Language. It is usually bundled with the application as raw source code and during the run-time of the application is compiled specifically for the device it is

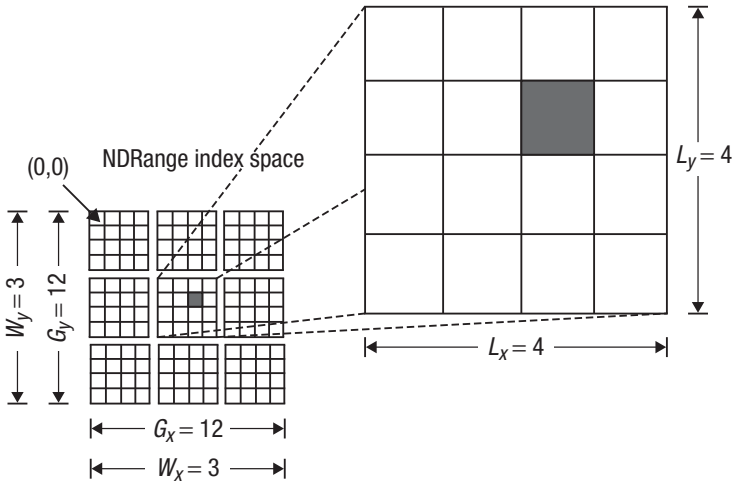


Figure 5.5: A two-dimensional index space with 9 work-groups, arranged in a 3×3 grid, each of which has 16 work-items. The global index of the highlighted work-item is $(6, 5)$. It is part of work-group $(1, 1)$ and its local index within this work-group is $(2, 1)$. [57]

going to be executed on. The resulting binaries are called *kernels*; they are the basic unit of execution in OpenCL, similar to functions in other languages.

When a kernel is executed, it is invoked with an index space. One instance of the kernel is run for each of the indices. A single instance is called a *work-item* and is identified uniquely by its coordinates in the index space. In OpenCL the index space is one- to three-dimensional. Work-items are organized into *work-groups* and can communicate via shared memory and memory barriers with other work-items in the same work-group. Every work-item has a global index and a local index within the work-group. Figure 5.5 shows a concrete example of an index space. All work items execute the same program code. They operate on different data or diverge in their kernel program execution by using their indices in offset and branch computations.

The host interacts with devices via a *command-queue*, which is used to issue commands to the OpenCL context. Commands can be used to execute kernels or to manipulate *memory objects*. Memory objects are regions of memory containing values that are used for several operations by instances of the kernel. The host can create memory objects of various sizes, transfer data between them, and copy data between them and the host address space.

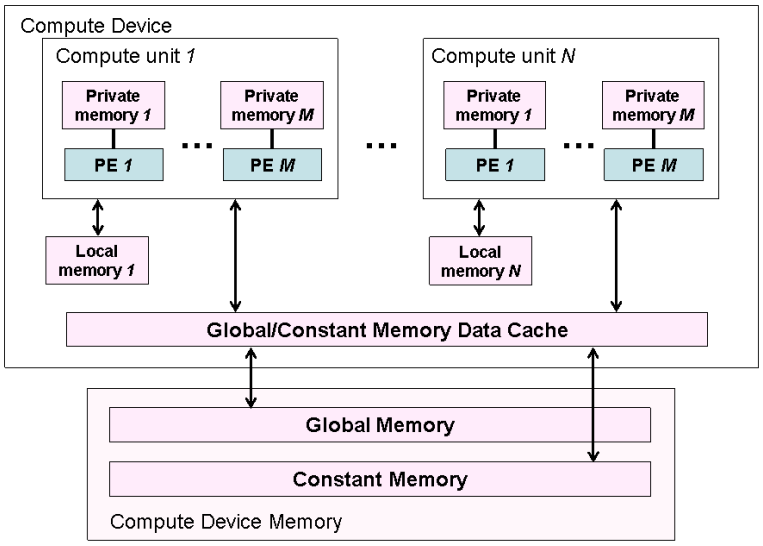


Figure 5.6: The OpenCL memory model. [25]

OpenCL exposes the memory hierarchy of the device to the application programmer and defines four distinct memory regions (Figure 5.6):

- **Private memory** can only be accessed by a single work-item. It is statically allocated by the kernel and used for storing thread local variables.
- **Local memory** is memory shared by a work-group. It can be allocated statically by the kernel or dynamically by host. The host can only allocate the memory region, it has no read or write access to it. Memory consistency can be enforced by using work-group-wide barrier synchronization.
- **Global memory** can be allocated by the host only. All work-groups have read and write access to it. Depending on the capabilities of the device access to global memory may be cached.
- **Constant memory** is a part of the global memory. The host allocates and initializes the memory region before a kernel is executed and its content remains constant for the duration of the execution.

The OpenCL Programming Language

The OpenCL programming language (OpenCL C) is based on the C programming language [45] (to be more precise, on the C standard from 1999, also known as C99 [34]). It is an extended subset of C99, which means, it removes some features and adds different features. Features not supported in OpenCL C are function pointers, recursion, bit fields and variable-length arrays.

OpenCL C adds the following features to C99: [57]

- **Vector data types**, i.e., literals and functions that allow writing portable vector code.
- **Address space qualifiers** are used to specify the region of memory that is used to allocate objects.
- **Additions to the language for parallelism.** Support for work-groups, work-items and synchronization within work-groups via memory barriers.
- **Image** data type and functions for reading and writing images.
- **Built-in functions** for integer and floating point math, geometric, relational, and atomic functions.

5.2.2 PyOpenCL

OpenCL bindings are available for most common programming languages, including C++, C#, Java, Python, Ruby, and even Javascript. Most bindings are simple wrappers around the standard C headers and provide little benefit over using them directly. PyOpenCL [47], a binding for the Python programming language [80], is different in that it uses run-time code generation for the OpenCL kernels.

OpenCL kernel code can run on a wide variety of devices, but to run efficiently it needs to take into account the specifics of the hardware it is actually running on. At run-time properties that are crucial for performance such as the width of native vector types and size of work groups and local memory can be queried. This information is used to generate a version of the kernel code that is highly optimized for this particular device.

PyOpenCL provides an array class that implements the same interface as the *numpy* Python library [61], which is widely-used for scientific computing. This class supports element-wise operations and reductions that run on the GPU and are automatically tuned for the device via run-time code generation. There are built-in functions for often used

operations such as element-wise addition of two arrays and calculating the maximum value in an array. PyOpenCL also provides extension points for user defined functions.

Integrated in PyOpenCL is a library for generating random numbers directly on the GPU (based on RANLUX [38]) and a small templating language. The templating language can be used for simple compile-time metaprogramming in OpenCL kernels. Kernels can be parameterized with information about the dataset at hand, which results in more specialized and efficient code.

PyOpenCL itself and all the Python code runs on the host. PyOpenCL loads kernels written in OpenCL C (or generates code for them), compiles, and executes them. The kernels can be executed on the GPU, not the Python code itself. This is different from *Copperhead* [12], which is an embedded domain specific language that transforms annotated Python code and executes it on the GPU.

By using a dynamic language like Python on the host side, all the benefits commonly associated with scripting languages are gained: interactive development in a REPL (Read Eval Print Loop) is possible, productivity is increased, and programs are easier to debug. All of this has no negative impact on performance as the performance-sensitive code that runs on the GPU is compiled OpenCL C code.

CHAPTER 6

Implementation

The main part of this thesis was to design and implement a hybrid metaheuristic, namely an Ant Colony Optimization (ACO) incorporating a Local Search procedure (LS) for solving the longest common subsequence problem. Algorithm 6.1 shows the general outline of the algorithm and highlights that apart from initialization, the entire algorithm is capable of running in parallel on the GPU. The ACO is based on the work by Shyu and Tsai [72], and will be presented in Section 6.1. A Local Search to further improve the best solution found by the ACO is described in Section 6.2. The parameter settings used in this thesis are listed in Table 6.1.

6.1 Ant Colony Optimization

Algorithm 6.1 starts off by calculating an upper bound c^* for the length of the expected solution. For this it computes the optimal longest common subsequence of two randomly selected strings (line 1). This part of the algorithm is not performance critical and can be done on the CPU by any LCS2 algorithm, for example the ones mentioned in Bergroth et al. [1].

During the initialization phase (lines 2-5) a number of data structures are allocated and set up in GPU memory. First, the set of n input strings $S = \{S_1, S_2, \ldots, S_n\}$ is copied. If the strings are not of equal length, shorter ones are padded with a character o that does not occur in any of the strings, $o \notin \Sigma$. All strings are now of length len.

The pheromones are stored in a matrix of floating point numbers of size $n \times len$, i.e., the same dimensions as the input strings. At the start of the algorithm, all pheromones

Algorithm 6.1: Hybrid ACO-LS Algorithm

```
1  c* ← |LCS2(S_x, S_y)|                                   // CPU
2  copy strings to GPU memory                               // CPU
3  initialize pheromone trails & ant memory
4  BS ← ∅
5  i ← 0
6  while i < iterations do
7  |   reset ant memory
8  |   generate random numbers
9  |   foreach ant do
10 |   |   construct ant solution
11 |   end
12 |   IB ← best solution of this iteration
13 |   if perform Local Search then
14 |   |   Local Search
15 |   end
16 |   update pheromone trail
17 |   if IB is better than BS then
18 |   |   BS ← IB
19 |   end
20 |   i ← i + 1
21 end
22 copy BS to main memory                                   // CPU
```

are set to τ_0. The value of each element $\tau_{i,j}$ in the matrix represents the experience of choosing the corresponding character $S_{i,j}$ as part of the solution in previous iterations. Figure 6.1 shows an example of a pheromone matrix after 2000 iterations.

Let m be the number of ants. Each ant has its own "memory", which it uses to keep record of the progress during one iteration of the algorithm and where it stores its current solution. The memory of each ant is a matrix of size $n \times len$ storing positions of characters in S that are part of the solution. BS (best solution) is of the same shape as the memory of one ant ($n \times len$) and is used to store the best solution found so far. Figure 6.2 shows a visualization of a solution stored in the memory of an ant.

At the beginning of each iteration the memory of all ants is reset (line 7). Each ant has its own pool of pseudo random numbers that are pre-generated at the start of each iteration (line 8) using a built-in function of PyOpenCL that implements the RANLUX algorithm [38].

Table 6.1: Parameter values used for the Ant Colony Optimization.

Parameter		Value
α	Pheromone influence	1.0
β	Heuristic influence	2.0
ρ	Evaporation rate	0.004
τ_0	Initial pheromone value	0.5
τ_{min}	Minimum pheromone value	0.01
τ_{max}	Maximum pheromone value	0.99
d	Number of candidates	10 for DNA ($\|\Sigma\| = 4$) 32 for protein ($\|\Sigma\| = 20$)
q_0	Exploitation probability factor	0.9
q_1	Exclusion probability factor	0.95
Φ_{IB}	Iteration-best pheromone factor	0.1
Φ_{BS}	Best-so-far pheromone factor	0.1

Constructing the ant solution is done in parallel for each ant (lines 9-11). An explanation of the construction procedure follows in Section 6.1.1. All solutions found by the ants in this iteration are compared and the best solution of the iteration, i.e., the longest one, is stored in IB (line 12). The iteration's best solution can be further improved by an optional Local Search procedure (lines 13-15) that will be presented in Section 6.2.

The pheromone update is computed in line 16. Pheromones evaporate and the best solution of this iteration IB and the overall best solution BS deposit pheromones. Section 6.1.2 shows how the pheromone matrix is updated.

If a new best solution is found in the current iteration, it is stored in BS (lines 17-19). The algorithm terminates after a fixed number of iterations (lines 6,20) and copies the best solution found from the memory of the GPU into the main memory (line 22).

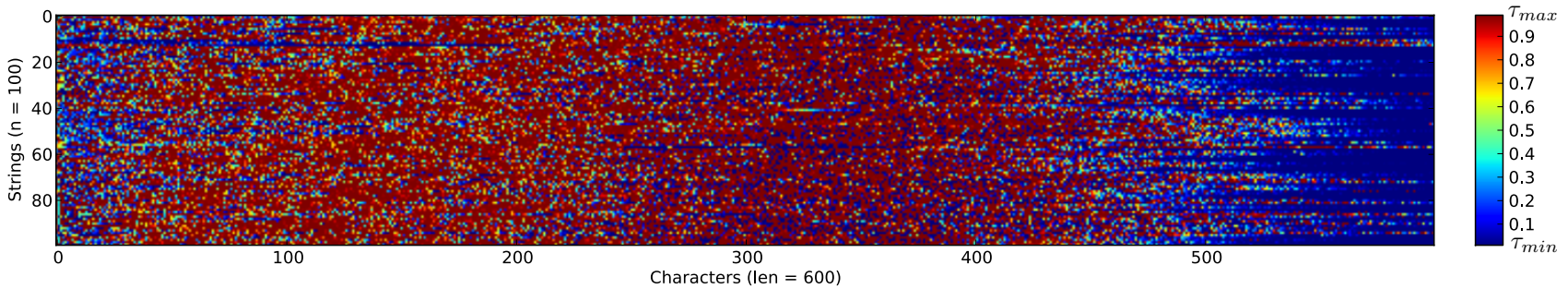


Figure 6.1: The pheromone matrix after 2000 iterations on the "DNA rat" dataset with 100 strings of length 600. Pheromone intensities are within the interval $[\tau_{min}, \tau_{max}]$, the higher the value the better the experience of choosing the character in previous iterations.

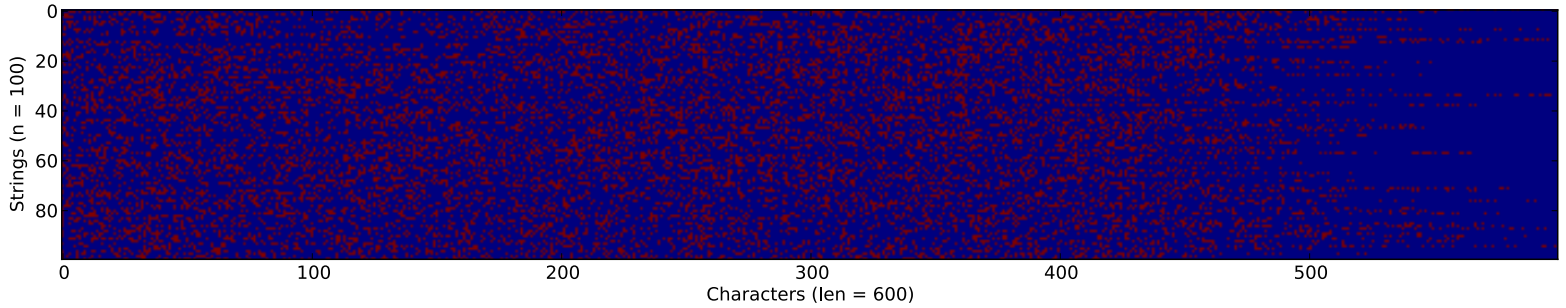


Figure 6.2: A snapshot of the memory of an ant, storing a solution found by the algorithm. Red dots indicate characters that are part of the solution. The dataset used is "DNA rat" with 100 strings of length 600.

6.1.1 Construct Ant Solution

Intuitively, the construction process of a solution can be seen as one ant walking along a randomly assigned string, looking for "good" characters and incrementally constructing a solution. In Algorithm 6.2 this process is formalized.

Algorithm 6.2: Construct Ant Solution

```
1  solution ← ∅
2  randomly choose r,    1 ≤ r ≤ n
3  u_i ← 0,    ∀i : 1 ≤ i ≤ n
4  while u_r ≤ |S_r| do
5      Cand ← [u_r + 1, u_r + 2, ..., max(u_r + d, |S_r|)]
6      foreach c ∈ Cand do
7          ch ← character at position c in S_r
8          v ← calculate next occurrences of character ch in all strings
9          calculate probabilistic transition factor pf(c)
10     end
11     q ← random number ,    0 ≤ q ≤ 1
12     choose c from Cand by probabilistic function p(v, q)
13     solution ← solution ∪ c
14     for i ← 1 to n do
15         u_i ← v_i
16     end
17 end
```

At the beginning of the solution construction algorithm, each ant is assigned a random string S_r (line 2). An array u is used to track the current position of the ant in all strings (line 3 and Figure 6.3a). The ant walks along the assigned string S_r (lines 4-17) from left to right until it reaches the end of the string.

At first a number of characters as candidates is selected to be added to the solution (line 5). From the current position of the ant u_r, the next d characters to the right are considered as candidates $Cand$ (Figure 6.3a). For each of the candidate characters, the position where the character occurs next (v_i) is calculated for all strings (lines 7-8, Figure 6.3b). This information will be used in the heuristic function of the probabilistic transition factor pf.

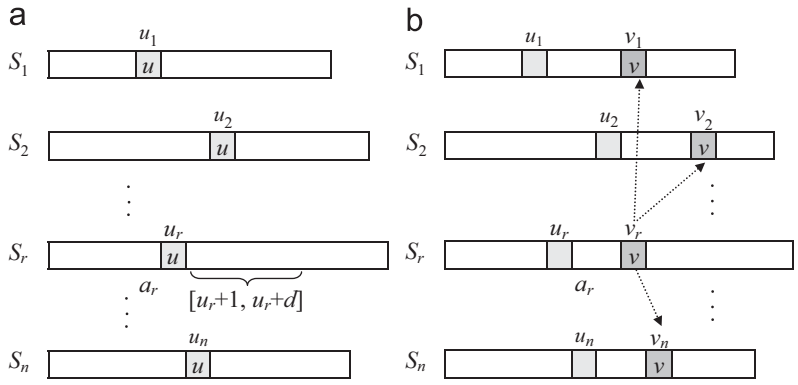


Figure 6.3: (a) The current position u of the ant in all strings and the range of potential candidates ($[u_r + 1, u_r + d]$). (b) The next occurrences of the candidate character v. [72]

The probabilistic transition factor $pf(c)$ determines the attractiveness of adding character c to the solution (line 9). It is calculated from previous experience in form of pheromones τ and the heuristic factor $\eta \in \{\eta_1, \eta_2\}$ that is based on greedy local decisions:

$$pf(c) = \frac{[\tau_{r,c}]^\alpha \cdot [\eta_{r,c}]^\beta}{\sum_{z \in Cand} [\tau_{r,z}]^\alpha \cdot [\eta_{r,z}]^\beta} \tag{6.1}$$

The greedy heuristic function used by Shyu and Tsai is based solely on the number of characters that would have to be skipped if v_r were to become part of the solution. Non-chosen characters in all strings are simply summed up:

$$\eta_1 = \frac{1}{\sum_{1 \leq i \leq n} (v_i - u_i)} \tag{6.2}$$

The implementation of this thesis uses a different heuristic function. Like η_1 it tries to skip as few characters as possible. In addition to that it also takes into account the number of remaining characters in each string. Preliminary tests showed that this function gives superior results in all cases (Section 7.1.4):

$$\eta_2 = \frac{1}{\sum_{1 \leq i \leq n} \frac{(v_i - u_i)}{(|s_i| - u_i)}} \tag{6.3}$$

The probabilistic transition factor is used by the probabilistic function p to determine which of the candidate characters, if any, is added to the solution (lines 11-13). A random number q is taken from the pseudo random number pool of the ant and is used to control exploitation, biased exploration and exclusion. Parameters q_0 and q_1, with $0 \leq q_0 < q_1 \leq 1$, are used to regulate the probability of choosing one of the three strategies:

$$p(c,q) = \begin{cases} 1 & \text{if } q \leq q_0 \text{ and } pf(c) = \underset{z \in Cand}{arg\,max}\, pf(z) \quad \text{(exploitation)} \\ pf(c) & \text{if } q_0 < q \leq q_1 \quad \text{(biased exploration)} \\ 0 & \text{otherwise } (q > q_1) \quad \text{(exclusion)} \end{cases} \tag{6.4}$$

In the exploitation case, the candidate with the best probabilistic transition factor is chosen, regardless of all others. When following the exclusion strategy, none of the candidates is chosen and the ant updates its position u_r to $u_r + d + 1$ without adding a character to the solution, effectively skipping all the candidates. For biased exploitation the probability of selecting a candidate character is proportional to its probabilistic transition factor. This is similar to the idea of roulette wheel selection [78] commonly used in Genetic Algorithms.

6.1.2 Pheromone Updating

Following the $\mathcal{MAX} - \mathcal{MIN}$ Ant System, the pheromones are updated in three steps for all elements $\tau_{i,j}$ of the pheromone matrix, $\forall i, j : 1 \leq i \leq n, 1 \leq j \leq len$.

The first step is the pheromone evaporation:

$$\tau_{i,j} \leftarrow (1 - \rho) \cdot \tau_{i,j} \tag{6.5}$$

In the second step, the best solution found so far (BS) and the best solution found in the current iteration (IB) deposit pheromones. The amount of pheromones is defined as the ratio between the length of the solution and the conservative upper bound c^*, calculated earlier (Algorithm 6.1, line 1). This is a slight extension to the $\mathcal{MAX} - \mathcal{MIN}$ Ant System as proposed by Stützle and Hoos, where either BS or IB deposit pheromones.

$$\tau_{i,j} \leftarrow \tau_{i,j} + \Delta\tau_{i,j}^{IB} \cdot \Phi_{IB} + \Delta\tau_{i,j}^{BS} \cdot \Phi_{BS} \tag{6.6}$$

$$\Delta\tau_{i,j}^{IB} = \begin{cases} \frac{|IB|}{c^*} & \text{if the } j\text{-th character in } S_i \text{ is part of } IB \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

$$\Delta\tau_{i,j}^{BS} = \begin{cases} \frac{|BS|}{c^*} & \text{if the } j\text{-th character in } S_i \text{ is part of } BS \\ 0 & \text{otherwise} \end{cases} \tag{6.8}$$

In the third and final step, the pheromone values are restricted to the interval $[\tau_{min}, \tau_{max}]$:

$$\tau_{i,j} = \begin{cases} \tau_{max} & \text{if } \tau_{i,j} > \tau_{max} \\ \tau_{min} & \text{if } \tau_{i,j} < \tau_{min} \\ \tau_{i,j} & \text{otherwise} \end{cases} \tag{6.9}$$

6.2 Local Search

The Local Search procedure takes a solution and tries to improve its quality, i.e., find a solution with greater length. It starts off with the best solution found so far (BS) by the ACO and explores its neighborhood $N(BS)$ exhaustively. The neighborhood is defined as follows. First, x adjacent characters are removed from the best solution BS. The neighbors of BS are all feasible strings that fill the gap with a new character sequence of length ≥ 0. If the new character sequence is longer than x, an improved solution has been found. New solutions are adopted following the first improvement strategy.

Algorithm 6.3: Local Search

```
1 while improve do
2 |   improve ← false
3 |   p ← 1
4 |   while p < |BS| − x do
5 |   |   improve ← improve ∨ enumerate(BS, p, x)
6 |   |   p ← p + 1
7 |   end
8 end
```

Algorithm 6.3 shows the outer loop of the Local Search procedure. The search continues as long as it yields further improvements to the solution (lines 1,2,5). In each iteration

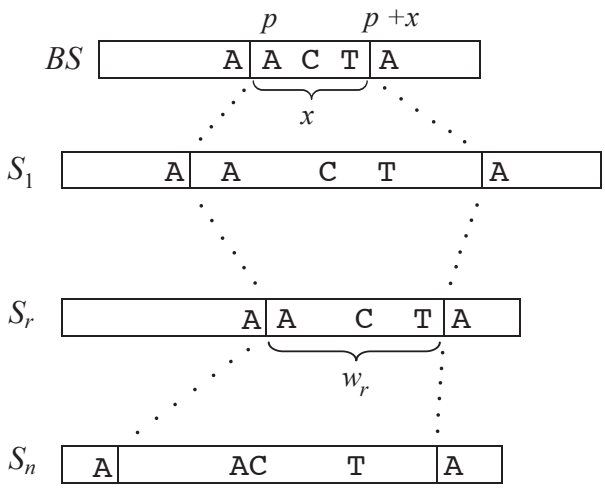


Figure 6.4: The characters from the subsequence $[p-1, p+x]$ in BS and their respective positions in the strings $S_1, S_2, \ldots, S_n$.

of the loop the whole solution is traversed (lines 3,4,6) and the function `enumerate` is called for every position p in BS (line 5). The parameters passed are the solution BS, the current position p, and x, which specifies the number of adjacent characters to be replaced with new characters. The function `enumerate` returns $true$ if an improvement to the solution was found and $false$ otherwise.

As the name suggests, `enumerate` (Algorithm 6.4) explores the neighborhood exhaustively by complete enumeration. Performing this computationally expensive task is possible through parallelization by distributing the validation of possible candidate solutions among the numerous cores of the GPU. Each scheduled thread checks if a candidate solution is valid in all strings. Breaking down the task of checking one solution further and performing the check in all strings in parallel as well would expose more parallelism but then each sub-task would be too fine-grained and the coordination and thread creation overhead would outweigh the actual computational task.

In the first step of the algorithm (line 2) an interval of length w_i, starting at position f_i and ending at position g_i, is calculated for all strings S_i, $\forall i : 1 \leq i \leq n$. This interval marks the gap in each string that is created when x characters are removed from the solution BS, starting at position p in BS (Figure 6.4). On the left hand side each interval is limited by the occurrence of the character at position $p-1$ in BS and on the right hand side by that of the character at position $p+x$.

Algorithm 6.4: Local Search - Enumeration

```
1  function enumerate(BS, p, x)
2      calculate w_i for S_i,    ∀i : 1 ≤ i ≤ n
3      r ← i, where w_i = arg min_{1≤i≤n} w_i
4      if w_r > ω then
5          return false
6      end
7      improve ← false
8      b ← bit array of length w_r
9      foreach b ← ⟨0, ..., 0⟩ to ⟨1, ..., 1⟩ do                    // parallel
10         if b is contained in S_i,    ∀i : 1 ≤ i ≤ n and |b| > x then
11             best ← b
12             improve ← true
13         end
14     end
15     if improve then
16         replace subsequence of BS from position p to p + x − 1 with best
17     end
18     return improve
19 end
```

Without loss of generality, let r be the index of the interval with the smallest length (line 3). Choosing r this way minimizes the run-time of the algorithm which is proportional to the length of the selected interval. If the length of the shortest interval (w_r) is longer than ω characters, the enumeration is deemed too costly and is skipped (lines 4-6). This sets an upper bound to the run-time. The value of ω is dependent on the capabilities of hardware the program is executed on and will be discussed in more detail at the end of this section.

All subsequences of the character sequence in the interval $[f_r, g_r]$ are possible candidate solutions (i.e., neighbors) for the Local Search. A candidate solution can be represented by a bit array b of length w_r with $b = \langle b_1, b_2, \ldots, b_{w_r} \rangle$ and $b_e \in \{0, 1\}, \forall e : 1 \leq e \leq w_r$. Bit b_e set to 1 indicates that the character at position $f_r + e - 1$ is used in the candidate solution, while set to 0 indicates that the corresponding character is left out.

All possible candidate solutions are enumerated in parallel, see line 9. A sequence of characters t represented by the bit array b is considered to be a valid candidate solution if and only if it is a subsequence of all intervals ($t \prec [f_i, g_i], \forall i : 1 \leq i \leq n$). If a

candidate solution is valid and contains more characters than x, it is stored as the best choice $best$ (lines 10-13). If an improved sequence has been found by the enumeration, the best solution found so far BS is updated and the function returns $true$ (lines 15-18).

Given a bit array of length w_r, it follows that there are 2^{w_r} possible candidate solutions to explore. The hardware used for benchmarking the implementation of this thesis (see Table 7.1) is capable of checking millions of possible solutions in a short time. A high number of threads executing in parallel is needed to utilize all cores of the GPU, but once all cores are fully saturated, doubling the number of threads also doubles the execution time. For this setup, the upper bound ω was set to 25, which corresponds to exploring over 33.5 million candidate solutions in parallel.

The number of characters to be replaced, as defined by x, has a direct impact on the length w_r. The more characters are skipped, the more likely it is that even the smallest gap ($[f_r, g_r]$) is longer than ω and the enumeration is skipped. Following preliminary tests, the value used for x in the rest of this thesis is randomly chosen from the interval $[2, 5]$ in each invocation of Algorithm 6.3.

CHAPTER 7

Tests

For testing the implementation, a subset of the dataset proposed by Shyu and Tsai in [72] was used. This subset consists of DNA and protein sequences and has been used by many of the recently published papers that present algorithms to solve the longest common subsequence problem [2, 54, 77, 86]. Shyu and Tsai randomly selected sequences from the *GenBank*[1] of the *National Center for Biotechnology Information* (NCBI) to create their test set. The exact list of sequences they used is available online[2] and the sequences themselves can be downloaded with tools such as *genbank-download*[3].

The test dataset consists of 20 sets of sequences in total. Half of them are DNA sequences with an alphabet size of $|\Sigma| = 4$ and the other half are protein sequences with $|\Sigma| = 20$ different characters. For both types of sequences, there are ten sets of strings with the following number of strings: $n \in \{10, 15, 20, 25, 40, 60, 80, 100, 150, 200\}$. Strings that were longer than 600 characters have been truncated, strings that were shorter have been padded with a character that is not part of the alphabet. Therefore, all strings used for testing have a length of 600.

Unless stated otherwise, each instance was tested with 20 runs on an idle machine. Two machines were used for testing the implementation. Tests that run primarily on the CPU were performed on a laptop with a quad-core processor. The tests that stress the GPU were run on a desktop computer with an NVIDIA GPU. For details on the system configuration of both machines see Table 7.1.

[1] `https://www.ncbi.nlm.nih.gov/genbank`
[2] `http://tmue.edu.tw/~sjshyu/public/aco_lcs/accno-aco_lcs.html`
[3] `http://simon.net.nz/articles/genbank-download/`

Table 7.1: The two system configurations used for the tests. The CPU is a quad-core processor with Hyper-Threading (HT) technology. Therefore, to the operating system and to OpenCL it appears to have 8 virtual cores.

	CPU	GPU
Manufacturer	Intel	NVIDIA
Model name	i7-2820QM	GeForce GTX 560 Ti
Market launch	Early 2011	Early 2011
Operating system	Mac OS X 10.8.3	Kubuntu 12.04 LTS
Driver version	1.1	310.14
OpenCL version	1.2	1.1
PyOpenCL version	2013.1 pre-release	2013.1 pre-release
Cores	4 (8 HT)	384 CUDA cores
Clock frequency	2300 MHz	1645 MHz
Available memory	8 GB	1 GB

All times reported are wall-clock times elapsed from the start of the first step of the ACO, the pheromone initialization, to end of the last iteration of the ACO or LS, when the final solution has been written to device memory (Algorithm 6.1, lines 3-21).

7.1 Micro Benchmarks

This section presents the results of a series of micro benchmarks. The benchmarks were chosen to illustrate certain properties of the algorithm or the architecture (CPU or GPU) the program is executed on. Results obtained from these benchmarks influenced the parameter settings used for the comparison of the implementation with the work by Shyu and Tsai (Section 7.2).

7.1.1 Multi-core Speedup

The first micro benchmark measures the possible speedup obtained, when the program is run on a multi-core CPU, compared to execution on a single-core. The CPU used for benchmarking uses *Intel Turbo Boost Technology* to speed up the execution of single threaded applications. When an application uses only a single core and the other cores are idle, it automatically over-clocks the busy core. For this test *Turbo Boost* was disabled with a kernel extension that sets the appropriate CPU processor flags[4]. On Mac OS X the number of cores can be restricted with the *Instruments* tool, which is part of Apple's development toolchain. Unfortunately similar tools do not exist for NVIDIA's OpenCL implementation so this can be tested on the CPU only.

[4] `https://github.com/nanoant/DisableTurboBoost.kext`

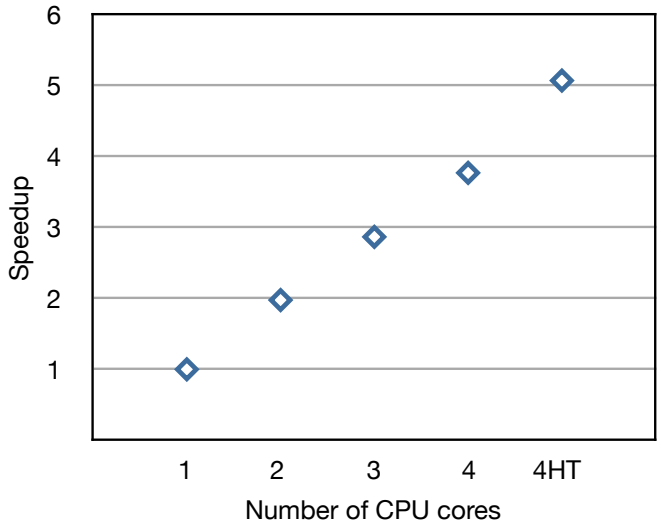


Figure 7.1: Speedup when executing the program on multiple CPU cores compared to a single core. Using up to four physical cores the program achieves near linear speedup. Enabling Hyper-Threading, shown as "4HT" in the figure, gives a total speedup of factor 5 using 8 virtual cores compared to a single core.

The CPU used for benchmarking is also capable of Hyper-Threading. Hyper-Threading is Intel's implementation of simultaneous multithreading, which makes each physical processor core appear to the operating system and the application as two virtual processor cores [49]. The operating system can schedule twice as many threads at once, leaving the processor with more independent instructions that can be issued simultaneously. This allows the processor to better utilize all execution units and reduces the performance penalty of pipeline stalls.

To test the performance on a varying number of cores the "DNA rat" dataset with 10 strings is taken for testing purposes using 500 iterations and 80 ants. The number of ants is high enough to saturate all cores. As shown in Figure 7.1, the program is able to achieve near linear speedup, also called ideal speedup. The speedup S_p is the ratio between the execution time on a single core T_1 and on multiple cores T_p (i.e., $S_p = \frac{T_1}{T_p}$ with p being the number of processors). Therefore, linear speedup means doubling the number of processors halves the execution time of the program. In this specific case the program executes 3.77 times faster using 4 cores than on a single core. Enabling Hyper-Threading gives an additional speedup of 35%, allowing the program to execute 5.07 times faster than on a single core.

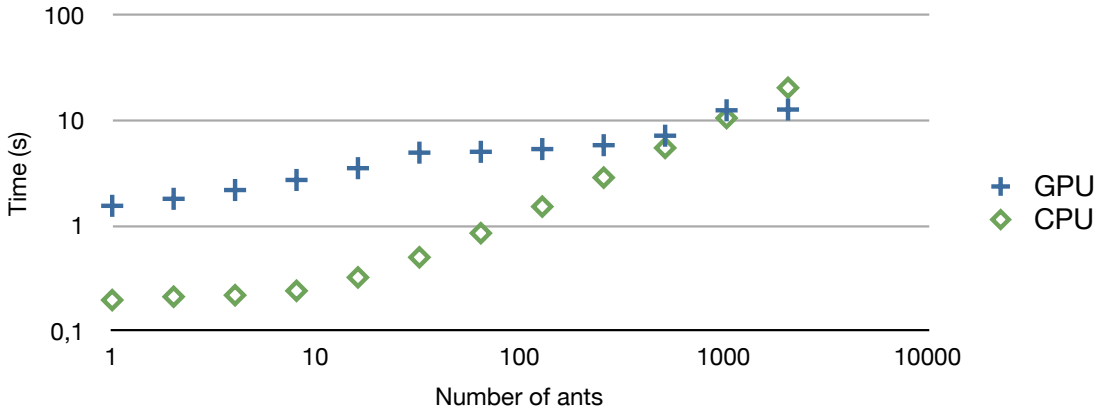


Figure 7.2: The run-time of the program, in seconds, with varying number of ants. Note that both axes are of logarithmic scale.

7.1.2 Parallel Execution of Threads

It is to be expected that running only a single thread at a time, in our case a single ant, is a task where the GPU lags behind the CPU by a good margin. The GPU has a lower clock speed and is not designed and optimized for this kind of workload. When the number of threads is increased, the parallel architecture of the GPU should kick in and the GPU should be able to outperform the CPU. The number of iterations in this test is set to 100 and the dataset used is "DNA rat" with 10 strings.

When only a single core is used (i.e., the number of ants is 1), execution on the CPU is 7.9 times faster than on the GPU, as shown in Figure 7.2 and Table 7.2. On the CPU, execution time stays almost the same when the number of ants is increased from one until it exceeds the number of physical processor cores (in this case 4). After that point it increases linearly, when the number of ants is doubled, the execution time doubles.

On the GPU the execution time appears to raise in a stepwise pattern. The execution time of the configurations between 32 and 256 ants is very similar (4.94 to 5.82 seconds). This pattern can be observed again when the number of ants is doubled from 1024 to 2048. The run-time remains almost unchanged, increasing only by less than 2%, from 12.49 to 12.70 seconds.

In terms of wall-clock execution time, the GPU is able to outperform the CPU when there are more than about 1300 ants working in parallel and the CPU is running at full capacity, with four cores and Hyper-Threading enabled. If the CPU is restricted to single core execution, the GPU would be faster with 85 ants or more.

Table 7.2: The run-time of the program, in seconds, with varying number of ants and the respective speedup when executing on the GPU.

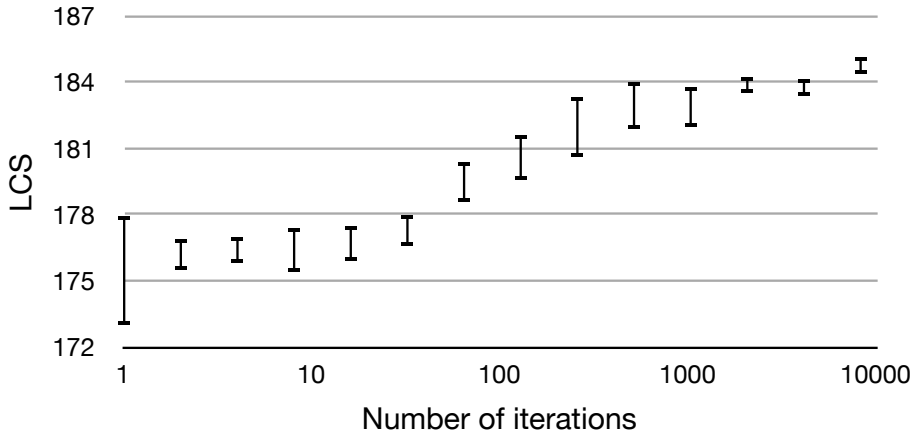
$ants$	CPU time	GPU time	GPU speedup
1	0.20	1.54	0.13
2	0.21	1.80	0.12
4	0.22	2.18	0.10
8	0.24	2.72	0.09
16	0.32	3.52	0.09
32	0.50	4.94	0.10
64	0.85	5.03	0.17
128	1.52	5.34	0.28
256	2.86	5.82	0.49
512	5.48	7.16	0.77
1024	10.54	12.49	0.84
2048	20.45	12.70	1.61

7.1.3 Number of Ants vs. Number of Iterations

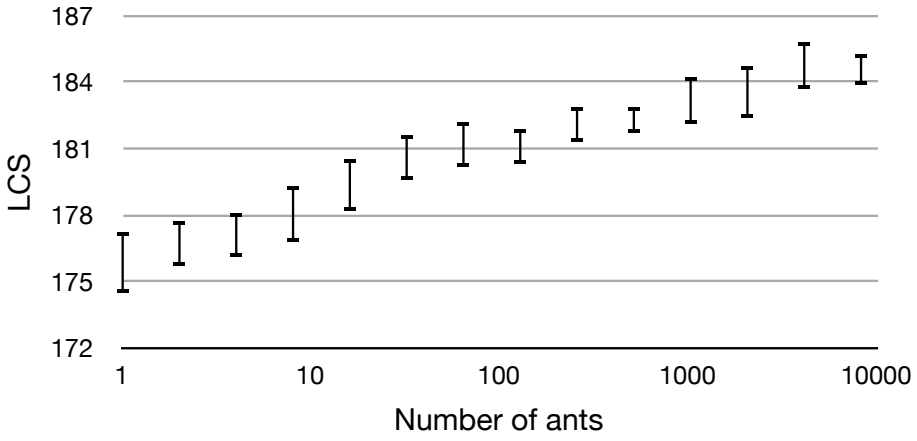
This scenario compares the impact of two parameters on the solution quality: the number of iterations and the number of ants. With Ant Colony Optimization, increasing the number of iterations should lead to an increase in the solution quality. Pheromones accumulated in earlier iterations guide the ants towards good solution components and a balanced heuristic function ensures new solutions are explored. Figure 7.3(a) shows this is indeed the case.

When the number of ants working simultaneously is increased, the search in the solution space is more diversified and a greater number of new solutions is found in each iteration. Recommended values for the number of ants are between 10 and 50 [78]. As Figure 7.3(b) shows, increasing the number of ants further, while keeping the iteration count constant, does improve the solution quality even more.

The impact of the number of iterations on the run-time is independent of the device the programs runs on. Individual iterations are performed sequentially, doubling the number of iterations always doubles the execution time of the program. As shown in Section 7.1.2 and Figure 7.4, the same consideration does not hold true for the number of ants when the program is run on the GPU. On the GPU, increasing the number of iterations from 32 to 2048 slows the program down by a factor of 32 (Figure 7.4(a)), while an increase in the number of ants from 32 to 2048 slows it down by only a factor of 2.6 (Figure 7.4(b)).

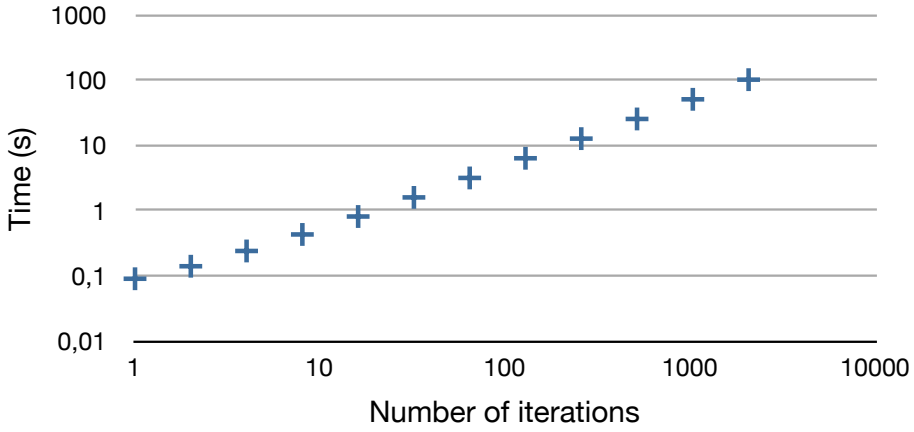


(a) 32 ants, varying number of iterations

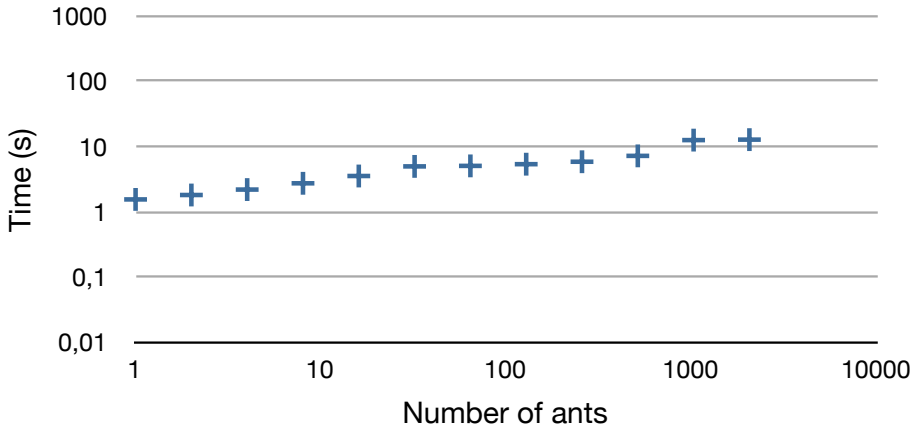


(b) 100 iterations, varying number of ants

Figure 7.3: The length of the solutions found by varying the number of iterations (a) and ants (b), displayed as the range of $mean \pm standard\ deviation$. Note the logarithmic scale on the x-axis. The dataset is "DNA rat" with 10 strings.



(a) 32 ants, varying number of iterations



(b) 100 iterations, varying number of ants

Figure 7.4: The run-time of the program on the GPU, in seconds, with varying number of iterations (a) and ants (b). Note the logarithmic scale on both axes. The dataset is "DNA rat" with 10 strings.

Table 7.3: The average length of the solutions found by two different heuristic functions on DNA and protein datasets with varying number of strings (n). The heuristic function used by Shyu and Tsai is listed as η_1 and compared with the heuristic function of this implementation, η_2. In all cases η_2 provides better or equal results. Statistical significance was tested with a Wilcoxon rank-sum test.

n		10	15	20	25	40	60	80	100	150	200
DNA	η_1	181.2	164.2	149.5	150.9	139.2	132.1	121.7	120.8	108.8	104.7
	η_2	183.8	167.8	155.6	154.4	141.0	134.1	124.8	124.2	114.0	107.2
Protein	η_1	66.5	56.5	54.0	50.9	46.5	43.7	42.5	41.8	41.5	40.6
	η_2	68.0	59.0	55.6	52.6	47.0	44.4	43.0	42.0	42.0	41.4

7.1.4 Heuristic Function

The implementation described in Section 6.1 uses a different heuristic function than the one used by Shyu and Tsai. Their greedy function is solely based on the number of characters skipped:

$$\eta_1 = \frac{1}{\sum_{1 \leq i \leq n}(v_i - u_i)}$$

The heuristic function of this thesis also takes into account the number of remaining characters:

$$\eta_2 = \frac{1}{\sum_{1 \leq i \leq n} \frac{(v_i - u_i)}{(|s_i| - u_i)}}$$

The impact of the new function η_2 is tested with 32 ants and 2000 iterations. As shown in Table 7.3, the modified heuristic function η_2 consistently yields better results than η_1. The results were tested for statistical significance using the SciPy [42] implementation of the Wilcoxon rank-sum test. The obtained error probability is always less than 1.5%, except in the instance "Protein 100" (12%).

7.2 Results

The implementation presented in this work uses OpenCL, which supports execution of the same program on CPU and GPU, allowing a fair side-by-side comparison of the algorithm on both architectures. This is in stark contrast to a lot of recently published

papers that compare a parallel GPU implementation with a sequential one that runs only on a single core of the CPU. A detailed critique of this practice can be found in Lee et al. [51].

Based on the micro benchmarks of Section 7.1, two sets of parameters were chosen and their performance was tested on CPU and GPU. The first parameter set is called *setCPU* and is a conservative setting using 32 ants and 2000 iterations. As the name implies, it is designed to perform well on the CPU.

The second set of parameters is optimized for execution on the GPU and therefore called *setGPU*. This setting uses a high number of ants and a lower number of iterations (200). The number of ants is restricted only by the amount of memory available. When operating on datasets with 10 to 100 strings, the number of ants is set to 3000. With larger datasets, the number of ants has to be decreased to fit in the 1GB of memory available on the GPU used for testing. The number of ants is set to 2000 with datasets of 150 strings and to 1500 with datasets of 200 strings.

All the other parameters used for the ACO are the same for both parameter sets and listed in Table 6.1. The Local Search was configured to use a width of two to five adjacent characters. Local Search is performed twice on the best solution found so far, once after half of the iterations and once after the last iteration. Applying the Local Search more often did not result in an increase in solution quality.

Tables A.1 to A.4 in Appendix A show the results of running the program with the parameter sets *setCPU* and *setGPU* on the CPU and the GPU, respectively. Table A.1 and Table A.2 use the "DNA rat" dataset while Table A.3 and Table A.4 use the "protein virus" dataset. The ACO was executed without Local Search for Table A.1 and Table A.3. Table A.2 and Table A.4 show the results of the ACO using Local Search.

Running the program with the same parameters on the CPU and the GPU gives almost the same solution quality. This is to be expected because the same program is run on both devices.

Both parameter sets, *setCPU* and *setGPU*, produce solutions of similar quality. *setGPU* gives slightly better solutions in general, the difference in terms of length of the solution is up to 2 characters. The solution quality of the ACO alone is comparable to the results reported by Shyu and Tsai in [72] for their ACO plus Local Search hybrid. On DNA sequences the solution quality is equally good or slightly worse (Table A.7) and on protein sequences it is consistently better (Table A.8).

Table 7.4: The run-times (in seconds) when executing the program on the CPU and on the GPU using the same parameter set *setGPU* on both architectures with varying number of strings (n). The GPU speedup is provided for each instance and additionally the average speedup and its standard deviation is shown in the last line.

	DNA			Protein		
	CPU	GPU	GPU	CPU	GPU	GPU
n	time [s]	time [s]	speedup	time [s]	time [s]	speedup
10	43.6	25.9	1.68	89.8	48.0	1.87
15	52.8	42.3	1.25	102.7	63.7	1.61
20	164.2	75.7	2.17	126.3	85.5	1.48
25	214.5	107.0	2.00	149.8	110.9	1.35
40	316.7	169.9	1.86	220.0	177.3	1.24
60	502.1	251.6	2.00	316.7	250.8	1.26
80	984.9	420.6	2.34	480.2	320.4	1.50
100	777.7	382.5	2.03	588.4	401.9	1.46
150	1634.0	1056.5	1.55	786.2	670.1	1.17
200	684.5	477.6	1.43	768.9	698.8	1.10
	Average speedup:		$1.83\ (0.35)$	Average speedup:		$1.41\ (0.23)$

The Local Search used in this implementation has only a small impact on the solution quality. The solution quality of the ACO alone and ACO plus Local Search combined are compared in Table A.5 for DNA sequences and Table A.6 for protein sequences. Only in three cases, the "DNA rat" dataset with $n \in \{10, 15, 40\}$, Local Search improves the solution quality in a statistically significant way. Therefore, the remainder of this chapter discussing the run-time of the program will focus only on the ACO.

With regard to the run-time, the parameter sets obviously favor the architecture they were designed for. Using *setCPU* on the CPU and *setGPU* on the GPU results in much shorter run-times than switching the parameter sets on the architectures.

When both architectures use the parameter set *setGPU*, the GPU is on average 1.6 times faster than the CPU (1.83 on DNA sequences, 1.41 on protein sequences), as shown in Table 7.4. As a point of reference, Cagnoni et al. [11] report a speed gain of "1 to no more than 5-6" for the GPU on almost identical hardware for their implementation of a particle swarm optimization, "probably one of the algorithms that is most suitable for parallelization on massively parallel architectures".

Comparing the wall-clock time of the CPU using the *setCPU* parameters with the GPU using the *setGPU* parameters, the CPU is faster in all tests. As shown in Table 7.5,

Table 7.5: The run-times (in seconds) when executing the program on the CPU and on the GPU using the same parameter set *setCPU* on both architectures with varying number of strings (n). The GPU speedup is provided for each instance and additionally the average speedup and its standard deviation is shown in the last line.

	DNA			Protein		
	CPU	GPU	GPU	CPU	GPU	GPU
n	time [s]	time [s]	speedup	time [s]	time [s]	speedup
10	6.9	25.9	0.27	11.7	48.0	0.24
15	9.0	42.3	0.21	13.4	63.7	0.21
20	21.2	75.7	0.28	16.6	85.5	0.19
25	27.0	107.0	0.25	19.3	110.9	0.17
40	37.8	169.9	0.22	26.1	177.3	0.15
60	53.5	251.6	0.21	37.2	250.8	0.15
80	103.7	420.6	0.25	52.0	320.4	0.16
100	78.8	382.5	0.21	64.5	401.9	0.16
150	258.6	1056.5	0.24	131.9	670.1	0.20
200	132.0	477.6	0.28	175.0	698.8	0.25
	Average speedup:		0.24 (0.03)	Average speedup:		0.19 (0.04)

in the DNA tests the CPU is faster than the GPU by a factor of 3.5 to 4.85 and in the protein tests by a factor of 4 to 6.8. This is because we compare the GPU to a multi-core CPU, not just to a single core. Compared to a single core, the GPU would be up to 40% faster in the DNA tests when the multi-core speedup shown in Section 7.1.1 is taken into account.

CHAPTER 8

Conclusion

In this master thesis a hybrid metaheuristic for calculating the longest common subsequence of multiple strings was presented. Based on the work by Shyu and Tsai [72] a parallel Ant Colony Optimization (ACO) incorporating a Local Search (LS) was developed. The algorithm was designed for execution on modern multi-core CPUs and massively parallel processor architectures such as graphics processing units (GPUs).

The ACO part of the implementation makes use of an improved heuristic function, allowing the ACO alone to perform on par with Shyu and Tsai's ACO + LS hybrid in terms of solution quality. The highly parallel Local Search employed by this implementation cannot improve the solutions significantly. This approach can outperform the work of Shyu and Tsai, but the overall performance still falls behind the state-of-the-art approach, a Hyper Heuristic by Tabataba et al. [77].

Using OpenCL for the implementation of the algorithm allows a fair comparison of the performance of CPU and GPU as the same program can be executed on both architectures. As tested on a multi-core CPU, the implementation of the ACO scales linearly with the number of cores and makes good use of virtual cores exposed by Hyper-Threading. Tests on DNA datasets show that the same program executes on average 1.8 times faster on the GPU than on the multi-core CPU when the ACO uses thousands of parallel ants and few iterations. When using a CPU optimized parameter set (with few ants ant more iterations) this speedup shifts towards the multi-core CPU (average 4.2).

8.1 Future Work

In this work the ACO is parallelized by performing the search procedure of the ants in parallel. In each iteration of the ACO, the ants construct their solutions in tandem. Having more ants leads to a higher level of parallelism, which in turn leads to higher performance on parallel architectures such as GPUs.

Using a higher number of ants does not necessarily improve the solution quality of ACOs in general beyond a certain point. There is little benefit in using more than a couple of thousand ants, as seen in the micro benchmarks in Section 7.1.3 and reported by Weiss [85]. Splitting ants into colonies that are able to operate independently of each other is one approach to make use of a very high number of ants. The work of Delévacq et al. [18] shows promising results with several ant colonies on GPUs.

The Local Search used in this work explores neighborhoods that only slightly improve the solutions despite exploring the neighborhoods exhaustively. A promising approach would be to design very large neighborhood structures that can be explored in parallel.

APPENDIX A

Results

Table A.1: DNA rat - ACO. The average length of the longest common subsequence (LCS) in characters, the average runtime in seconds, and the corresponding standard deviations. The dataset is "DNA rat" with a varying number of strings (n). The parameter sets used, *setCPU* and *setGPU*, are explained in Section 7.2 and the general ACO parameters are listed in Table 6.1. The two parameter sets were tested on both architecture (CPU and GPU).

	CPU				GPU			
	setCPU		setGPU		setCPU		setGPU	
n	LCS	time [s]	LCS	time [s]	LCS	time [s]	LCS	time [s]
10	183.7 (0.7)	6.9 (0.5)	184.5 (0.6)	43.6 (0.5)	183.8 (0.6)	99.1 (0.0)	185.3 (1.0)	25.9 (0.0)
15	167.7 (0.6)	9.0 (0.3)	169.5 (0.9)	52.8 (0.5)	169.8 (1.7)	141.8 (0.0)	168.7 (0.6)	42.3 (0.0)
20	155.4 (0.5)	21.2 (0.5)	155.2 (0.4)	164.2 (4.8)	155.1 (0.2)	448.8 (0.2)	155.6 (0.7)	75.7 (0.1)
25	154.5 (1.1)	27.0 (0.9)	155.4 (0.9)	214.5 (5.9)	154.1 (0.2)	588.3 (0.1)	155.6 (0.6)	107.0 (0.1)
40	141.0 (0.3)	37.8 (1.3)	142.7 (0.5)	316.7 (9.0)	141.2 (0.8)	877.9 (1.1)	142.3 (0.7)	169.9 (0.2)
60	134.1 (0.9)	53.5 (1.9)	135.4 (0.6)	502.1 (52.2)	133.8 (0.5)	1280.9 (0.4)	135.3 (0.8)	251.6 (0.6)
80	124.8 (0.9)	103.7 (2.5)	127.2 (0.8)	984.9 (90.0)	126.7 (1.3)	2433.6 (1.3)	127.5 (0.9)	420.6 (1.2)
100	124.0 (0.6)	78.8 (3.4)	125.4 (0.7)	777.7 (77.8)	124.8 (0.5)	1983.7 (2.3)	125.6 (0.8)	382.5 (1.4)
150	114.0 (0.9)	258.6 (15.6)	115.3 (0.8)	1634.0 (176.2)	113.2 (0.4)	5959.9 (3.3)	115.1 (1.0)	1056.5 (1.1)
200	106.8 (1.0)	132.0 (8.9)	108.4 (1.0)	684.5 (74.8)	107.9 (1.0)	3193.0 (2.4)	108.8 (1.1)	477.6 (3.1)

Table A.2: DNA rat - ACO + Local Search. The average length of the longest common subsequence (LCS) in characters, the average run-time in seconds, and the corresponding standard deviations. The dataset is "DNA rat" with a varying number of strings (n). The parameter sets used, *setCPU* and *setGPU*, are explained in Section 7.2 and the general ACO parameters are listed in Table 6.1. The two parameter sets were tested on both architecture (CPU and GPU). Some values are missing because the GPU ran out of memory during the Local Search.

	CPU				GPU			
	setCPU		setGPU		setCPU		setGPU	
n	LCS	time [s]	LCS	time [s]	LCS	time [s]	LCS	time [s]
10	185.0 (1.2)	8.5 (0.7)	184.5 (0.7)	57.9 (3.3)	183.7 (0.6)	99.6 (0.0)	184.9 (1.3)	39.7 (2.8)
15	168.8 (0.5)	9.7 (0.5)	170.1 (0.8)	69.1 (3.7)	168.6 (2.0)	142.3 (0.1)	168.4 (0.7)	60.4 (3.5)
20	155.7 (0.5)	21.6 (1.2)	155.5 (1.0)	197.7 (7.3)	155.1 (0.3)	449.4 (0.2)	155.8 (0.7)	97.7 (4.4)
25	154.7 (1.1)	27.9 (1.5)	155.8 (0.7)	262.1 (9.6)	154.0 (0.0)	588.8 (0.1)	155.8 (0.8)	133.4 (4.4)
40	141.6 (0.7)	39.2 (2.3)	143.2 (1.1)	392.5 (16.7)	141.5 (0.8)	878.8 (0.2)	142.6 (0.6)	209.6 (7.7)
60	133.8 (0.8)	54.9 (3.6)	135.4 (0.9)	565.3 (38.5)	134.1 (0.5)	1281.8 (0.3)	135.8 (0.7)	311.6 (12.2)
80	125.3 (1.3)	105.0 (5.2)	127.5 (0.8)	1070.8 (38.5)	126.5 (1.3)	2434.3 (1.0)		
100	124.4 (0.7)	81.7 (3.8)	125.7 (0.8)	865.3 (47.2)	124.9 (0.5)	1987.1 (2.9)		
150	114.3 (0.9)	258.8 (11.2)	115.3 (0.8)	1723.7 (69.7)	113.6 (0.5)	5962.1 (1.8)		
200	107.7 (1.3)	133.9 (5.7)	108.7 (1.1)	775.8 (27.4)	108.1 (0.9)	3192.2 (3.3)		

Table A.3: Protein virus - ACO. The average length of the longest common subsequence (LCS) in characters, the average run-time in seconds, and the corresponding standard deviations. The dataset is "Protein virus" with a varying number of strings (n). The parameter sets used, *setCPU* and *setGPU*, are explained in Section 7.2 and the general ACO parameters are listed in Table 6.1. The two parameter sets were tested on both architecture (CPU and GPU).

	CPU				GPU			
	setCPU		setGPU		setCPU		setGPU	
n	LCS	time [s]	LCS	time [s]	LCS	time [s]	LCS	time [s]
10	68.0 (0.0)	11.7 (1.0)	69.0 (0.4)	89.8 (1.7)	68.0 (0.0)	295.3 (0.2)	68.9 (0.3)	48.0 (0.1)
15	59.0 (0.0)	13.4 (0.4)	59.7 (0.5)	102.7 (3.7)	59.0 (0.0)	370.5 (0.2)	60.0 (0.5)	63.7 (0.3)
20	55.6 (0.5)	16.6 (0.6)	55.8 (0.4)	126.3 (3.4)	55.5 (0.5)	467.2 (0.4)	55.1 (0.4)	85.5 (0.2)
25	52.6 (0.5)	19.3 (0.7)	52.8 (0.4)	149.8 (8.8)	51.8 (0.4)	577.3 (0.5)	52.2 (0.4)	110.9 (0.2)
40	47.0 (0.0)	26.1 (1.7)	47.8 (0.4)	220.0 (10.0)	47.0 (0.0)	837.0 (0.9)	47.2 (0.4)	177.3 (0.2)
60	44.4 (0.5)	37.2 (3.0)	44.3 (0.5)	316.7 (7.7)	44.0 (0.0)	1163.8 (1.6)	44.3 (0.5)	250.8 (0.8)
80	43.0 (0.2)	52.0 (2.4)	43.0 (0.2)	480.2 (56.6)	42.2 (0.4)	1488.6 (1.0)	43.0 (0.0)	320.4 (0.6)
100	42.0 (0.2)	64.5 (3.1)	43.0 (0.0)	588.4 (98.7)	42.1 (0.3)	1858.2 (1.9)	43.0 (0.2)	401.9 (2.4)
150	42.0 (0.0)	131.9 (5.2)	42.6 (0.5)	786.2 (109.2)	42.0 (0.0)	3463.9 (0.9)	42.6 (0.5)	670.1 (1.0)
200	41.4 (0.5)	175.0 (6.9)	42.0 (0.2)	768.9 (71.7)	41.0 (0.0)	4632.8 (6.2)	42.0 (0.2)	698.8 (1.2)

Table A.4: Protein virus - ACO + Local Search. The average length of the longest common subsequence (LCS) in characters, the average run-time in seconds, and the corresponding standard deviations. The dataset is "Protein virus" with a varying number of strings (n). The parameter sets used, *setCPU* and *setGPU*, are explained in Section 7.2 and the general ACO parameters are listed in Table 6.1. The two parameter sets were tested on both architecture (CPU and GPU). Some values are missing because the GPU ran out of memory during the Local Search.

	CPU				GPU			
	setCPU		setGPU		setCPU		setGPU	
n	LCS	time [s]	LCS	time [s]	LCS	time [s]	LCS	time [s]
10	68.1 (0.3)	17.3 (3.2)	69.1 (0.3)	99.0 (10.8)	68.0 (0.0)	296.8 (1.5)	69.0 (0.7)	53.6 (1.2)
15	59.0 (0.0)	15.4 (1.0)	59.6 (0.6)	112.6 (5.4)	59.0 (0.0)	370.9 (0.2)	60.0 (0.4)	69.6 (0.9)
20	55.5 (0.5)	19.8 (1.8)	55.8 (0.5)	150.0 (9.7)	55.3 (0.5)	467.4 (0.6)	55.2 (0.4)	96.0 (6.8)
25	52.5 (0.5)	22.3 (1.8)	52.9 (0.4)	180.3 (16.8)	52.0 (0.0)	578.2 (0.9)	52.3 (0.5)	120.5 (1.7)
40	47.0 (0.0)	27.7 (1.6)	47.8 (0.4)	240.6 (11.5)	46.7 (0.5)	838.0 (0.6)	47.1 (0.4)	189.5 (1.7)
60	44.3 (0.5)	43.5 (5.1)	44.4 (0.5)	342.4 (11.1)	44.0 (0.0)	1166.7 (1.7)	44.0 (0.2)	268.0 (1.0)
80	43.0 (0.2)	58.5 (14.8)	43.0 (0.0)	494.5 (56.5)	42.0 (0.0)	1488.2 (1.2)		
100	42.1 (0.3)	62.4 (8.5)	43.0 (0.0)	605.7 (42.2)	42.0 (0.0)	1858.7 (1.0)		
150	42.0 (0.0)	155.3 (26.9)	42.6 (0.5)	799.4 (36.4)	42.0 (0.0)	3467.8 (4.6)		
200	41.1 (0.4)	190.3 (28.8)	41.9 (0.3)	780.6 (47.7)	41.4 (0.5)	4633.5 (5.5)		

Table A.5: DNA rat - ACO compared with ACO + Local Search. The average length of the longest common subsequence in characters and the corresponding standard deviation. Statistical significance according to a Wilcoxon rank-sum test with an error level of 5% is shown in column p. The dataset is "DNA rat" with a varying number of strings (n). The parameter sets used, *setCPU* and *setGPU*, are explained in Section 7.2 and the general ACO parameters are listed in Table 6.1. The two parameter sets were tested on both architecture (CPU and GPU).

	CPU						GPU					
	setCPU			setGPU			setCPU			setGPU		
n	ACO	p	ACO+LS	ACO	p	ACO+LS	ACO	p	ACO+LS	ACO	p	ACO+LS
10	183.7 (0.7)	$<$	185.0 (1.2)	184.5 (0.6)	$\approx$	184.5 (0.7)	183.8 (0.6)	$\approx$	183.7 (0.6)	185.3 (1.0)	$\approx$	184.9 (1.3)
15	167.7 (0.6)	$<$	168.8 (0.5)	169.5 (0.9)	$\approx$	170.1 (0.8)	169.8 (1.7)	$>$	168.6 (2.0)	168.7 (0.6)	$\approx$	168.4 (0.7)
20	155.4 (0.5)	$\approx$	155.7 (0.5)	155.2 (0.4)	$\approx$	155.5 (1.0)	155.1 (0.2)	$\approx$	155.1 (0.3)	155.6 (0.7)	$\approx$	155.8 (0.7)
25	154.5 (1.1)	$\approx$	154.7 (1.1)	155.4 (0.9)	$\approx$	155.8 (0.7)	154.1 (0.2)	$\approx$	154.0 (0.0)	155.6 (0.6)	$\approx$	155.8 (0.8)
40	141.0 (0.3)	$<$	141.6 (0.7)	142.7 (0.5)	$\approx$	143.2 (1.1)	141.2 (0.8)	$\approx$	141.5 (0.8)	142.3 (0.7)	$\approx$	142.6 (0.6)
60	134.1 (0.9)	$\approx$	133.8 (0.8)	135.4 (0.6)	$\approx$	135.4 (0.9)	133.8 (0.5)	$\approx$	134.1 (0.5)	135.3 (0.8)	$\approx$	135.8 (0.7)
80	124.8 (0.9)	$\approx$	125.3 (1.3)	127.2 (0.8)	$\approx$	127.5 (0.8)	126.7 (1.3)	$\approx$	126.5 (1.3)	127.5 (0.9)		
100	124.0 (0.6)	$\approx$	124.4 (0.7)	125.4 (0.7)	$\approx$	125.7 (0.8)	124.8 (0.5)	$\approx$	124.9 (0.5)	125.6 (0.8)		
150	114.0 (0.9)	$\approx$	114.3 (0.9)	115.3 (0.8)	$\approx$	115.3 (0.8)	113.2 (0.4)	$\approx$	113.6 (0.5)	115.1 (1.0)		
200	106.8 (1.0)	$\approx$	107.7 (1.3)	108.4 (1.0)	$\approx$	108.7 (1.1)	107.9 (1.0)	$\approx$	108.1 (0.9)	108.8 (1.1)		

Table A.6: Protein virus - ACO compared with ACO + Local Search. The average length of the longest common subsequence in characters and the corresponding standard deviation. Statistical significance according to a Wilcoxon rank-sum test with an error level of 5% is shown in column p. The dataset is "Protein virus" with a varying number of strings (n). The parameter sets used, *setCPU* and *setGPU*, are explained in Section 7.2 and the general ACO parameters are listed in Table 6.1. The two parameter sets were tested on both architecture (CPU and GPU).

	CPU						GPU					
	setCPU			setGPU			setCPU			setGPU		
n	ACO	p	ACO+LS	ACO	p	ACO+LS	ACO	p	ACO+LS	ACO	p	ACO+LS
10	68.0 (0.0)	$\approx$	68.1 (0.3)	69.0 (0.4)	$\approx$	69.1 (0.3)	68.0 (0.0)	$\approx$	68.0 (0.0)	68.9 (0.3)	$\approx$	69.0 (0.7)
15	59.0 (0.0)	$\approx$	59.0 (0.0)	59.7 (0.5)	$\approx$	59.6 (0.6)	59.0 (0.0)	$\approx$	59.0 (0.0)	60.0 (0.5)	$\approx$	60.0 (0.4)
20	55.6 (0.5)	$\approx$	55.5 (0.5)	55.8 (0.4)	$\approx$	55.8 (0.5)	55.5 (0.5)	$\approx$	55.3 (0.5)	55.1 (0.4)	$\approx$	55.2 (0.4)
25	52.6 (0.5)	$\approx$	52.5 (0.5)	52.8 (0.4)	$\approx$	52.9 (0.4)	51.8 (0.4)	$\approx$	52.0 (0.0)	52.2 (0.4)	$\approx$	52.3 (0.5)
40	47.0 (0.0)	$\approx$	47.0 (0.0)	47.8 (0.4)	$\approx$	47.8 (0.4)	47.0 (0.0)	$\approx$	46.7 (0.5)	47.2 (0.4)	$\approx$	47.1 (0.4)
60	44.4 (0.5)	$\approx$	44.3 (0.5)	44.3 (0.5)	$\approx$	44.4 (0.5)	44.0 (0.0)	$\approx$	44.0 (0.0)	44.3 (0.5)	$\approx$	44.0 (0.2)
80	43.0 (0.2)	$\approx$	43.0 (0.2)	43.0 (0.2)	$\approx$	43.0 (0.0)	42.2 (0.4)	$\approx$	42.0 (0.0)	43.0 (0.0)		
100	42.0 (0.2)	$\approx$	42.1 (0.3)	43.0 (0.0)	$\approx$	43.0 (0.0)	42.1 (0.3)	$\approx$	42.0 (0.0)	43.0 (0.2)		
150	42.0 (0.0)	$\approx$	42.0 (0.0)	42.6 (0.5)	$\approx$	42.6 (0.5)	42.0 (0.0)	$\approx$	42.0 (0.0)	42.6 (0.5)		
200	41.4 (0.5)	$\approx$	41.1 (0.4)	42.0 (0.2)	$\approx$	41.9 (0.3)	41.0 (0.0)	$\approx$	41.4 (0.5)	42.0 (0.2)		

Table A.7: DNA rat - ACO compared with Shyu and Tsai's ACO+LS [72]. The average length of the longest common subsequence in characters and the corresponding standard deviation. Tests were performed on the GPU using parameter set *setGPU*. The dataset is "DNA rat" with a varying number of strings (n).

	setGPU ACO	Shyu & Tsai ACO+LS
n	GPU	CPU
10	185.3 (1.0)	182.0 (2.4)
15	168.7 (0.6)	166.6 (1.3)
20	155.6 (0.7)	160.0 (1.3)
25	155.6 (0.6)	155.8 (1.3)
40	142.3 (0.7)	143.4 (0.8)
60	135.3 (0.8)	142.4 (1.7)
80	127.5 (0.9)	128.8 (0.7)
100	125.6 (0.8)	124.6 (2.0)
150	115.1 (1.0)	115.6 (1.3)
200	108.8 (1.1)	114.6 (2.3)

Table A.8: Protein virus - ACO compared with Shyu and Tsai's ACO+LS [72]. The average length of the longest common subsequence in characters and the corresponding standard deviation. Tests were performed on the GPU using parameter set *setGPU*. The dataset is "Protein virus" with a varying number of strings (n).

	setGPU ACO	Shyu & Tsai ACO+LS
n	GPU	CPU
10	68.9 (0.3)	65.6 (0.8)
15	60.0 (0.5)	55.8 (1.3)
20	55.1 (0.4)	53.6 (1.3)
25	52.2 (0.4)	49.6 (0.8)
40	47.2 (0.4)	46.4 (0.8)
60	44.3 (0.5)	43.4 (0.8)
80	43.0 (0.0)	43.0 (0.4)
100	43.0 (0.2)	42.0 (1.1)
150	42.6 (0.5)	42.6 (0.8)
200	42.0 (0.2)	41.0 (0.2)

Bibliography

[1] Lasse Bergroth, Harri Hakonen, and Timo Raita. A Survey of Longest Common Subsequence Algorithms. *Proceedings Seventh International Symposium on String Processing and Information Retrieval SPIRE 2000*, pages 39–48, 2000.

[2] Christian Blum. Beam-ACO for the Longest Common Subsequence Problem. In *Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2010.

[3] Christian Blum and Maria J Blesa. Probabilistic beam search for the longest common subsequence problem. In *Proceedings of the 2007 international conference on Engineering stochastic local search algorithms: designing, implementing and analyzing effective heuristics*, pages 150–161, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Christian Blum and Monaldo Mastrolilli. Using Branch & Bound Concepts in Construction-Based Metaheuristics: Exploiting the Dual Problem Knowledge. In *Proceedings of the 4th international conference on Hybrid Metaheuristics*, pages 123–139, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] Paola Bonizzoni, Gianluca Della Vedova, and Giancarlo Mauri. Experimenting an Approximation Algorithm for the LCS. *Discrete Applied Mathematics*, 110(1):13–24, 2001.

[6] Peer Bork and Eugene V Koonin. Protein sequence motifs. *Current Opinion in Structural Biology*, 6(3):366–376, 1996.

[7] Philip Brisk, Adam Kaplan, and Majid Sarrafzadeh. Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs. In *Proceedings of the 41st annual Design Automation Conference*, pages 395–400, New York, NY, USA, 2004. ACM.

[8] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[9] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[10] Bernd Bullnheimer, Richard F Hartl, and Christine Strauß. A New Rank Based Version of the Ant System - A Computational Study. *Central European Journal for Operations Research and Economics*, 7:25–38, 1997.

[11] Stefano Cagnoni, Alessandro Bacchini, and Luca Mussi. OpenCL Implementation of Particle Swarm Optimization: A Comparison between Multi-core CPU and GPU Performances. In *Proceedings of the 2012 European conference on Applications of Evolutionary Computation*, pages 406–415, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an Embedded Data Parallel Language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 47–56, New York, NY, USA, 2011. ACM.

[13] Manuel M T Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, pages 3–14, New York, NY, USA, 2011. ACM.

[14] Manuel M T Chakravarty and Peter Thiemann. Agda Meets Accelerate. In *Revised Papers of the 24th Symposium on Implementation and Application of Functional Languages (IFL 2012)*, pages 174–189, 2013.

[15] Yixin Chen, Andrew Wan, and Wei Liu. A fast parallel algorithm for finding the longest common sequence of multiple biosequence. *BMC Bioinformatics*, 7, 2006.

[16] Chung-Han Chiang. A Genetic Algorithm for the Longest Common Subsequence of Multiple Sequences. Master's thesis, National Sun Yat-sen University, July 2008.

[17] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 3rd edition, 2009.

[18] Audrey Delévacq, Pierre Delisle, Marc Gravel, and Michaël Krajecki. Parallel Ant Colony Optimization on Graphics Processing Units. *Journal of Parallel and Distributed Computing*, 73(1):52–61, 2013.

[19] Jean Louis Deneubourg, Serge Aron, Shyamali Goss, and Jacques M Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3:159–168, 1990.

[20] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[21] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, USA, 1st edition, 2004.

[22] Lance Fortnow. The Status of the P versus NP Problem. *Communications of the ACM*, 52(9):78–86, 2009.

[23] Shyamali Goss, Serge Aron, Jean Louis Deneubourg, and Jacques M Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76:579–581, 1989.

[24] Pierre-Paul Grassé. La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. La théorie de la stigmergie: Essai d'interprétation des termites constructeurs. *Insectes Sociaux*, 6:41–84, 1959.

[25] Khronos OpenCL Working Group. The OpenCL Specification Version 1.0. Accessed April 19, 2013.
`http://www.khronos.org/opencl`

[26] Pierre Hansen and Nenad Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449–467, 2001.

[27] Brenda Hinkemeyer and Bryant A Julstrom. A Genetic Algorithm for the Longest Common Subsequence Problem. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 609–610, New York, NY, USA, 2006. ACM.

[28] Dan S Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18:341–343, June 1975.

[29] John H Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[30] Bert Hölldobler and Edward O Wilson. *The Ants*. Springer-Verlag, 2nd edition, 1998.

[31] Kuo-Si Huang, Chang-Biau Yang, and Kuo-Tsung Tseng. Fast Algorithms for Finding the Common Subsequence of Multiple Sequences. In *Proceedings of International Computer Symposium, 2004*, pages 90–95, Taipei.

[32] James W Hunt and M Douglas McIlroy. An Algorithm for Differential File Comparison. *Bell Laboratories Computing Science Technical Report 41*, July 1976.

[33] Wen-mei W Hwu, editor. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[34] International Organization for Standardization, International Electrotechnical Commission, and others. ISO/IEC 9899: 1999. Technical report.

[35] Robert W Irving and Campbell Fraser. Two Algorithms for the Longest Common Subsequence of Three (or More) Strings. In *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, pages 214–229, London, UK, UK, 1992. Springer-Verlag.

[36] Makoto Iwamura, Mitsutaka Itoh, and Yoichi Muraoka. Towards Efficient Analysis for Malware in the Wild. In *Proceedings of the 2011 IEEE International Conference on Communications (ICC)*, pages 1–6, 2011.

[37] Pekka O Jääskeläinen, Carlos S de La Lama, Pablo Huerta, and Jarmo H Takala. OpenCL-based design methodology for application-specific processors. In *Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 223–230, 2010.

[38] Fred James. RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Lüscher. *Computer Physics Communications*, 79(1):111–114, 1994.

[39] Thomas Jansen and Dennis Weyland. Analysis of Evolutionary Algorithms for the Longest Common Subsequence Problem. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, pages 939–946, New York, NY, USA, 2007. ACM.

[40] Tao Jiang and Ming Li. On the Approximation of Shortest Common Supersequences and Longest Common Subsequences. In *Proceedings of the 21st International Colloquium on Automata, Languages and Programming*, pages 191–202, London, UK, UK, 1994. Springer-Verlag.

[41] Tao Jiang, Guohui Lin, Bin Ma, and Kaizhong Zhang. A General Edit Distance between RNA Structures. *Journal of Computational Biology*, 9:371–388, 2002.

[42] Eric Jones, Travis Oliphant, and Pearu Peterson. SciPy: Open source scientific tools for Python. Accessed April 19, 2013.
`http://www.scipy.org/`

[43] Bryant A Julstrom and Brenda Hinkemeyer. Starting from Scratch: Growing Longest Common Subsequences with Evolution. In *Proceedings of the 9th international conference on Parallel Problem Solving from Nature*, pages 930–938, Berlin, Heidelberg, 2006. Springer-Verlag.

[44] Jim A Kahle, Michael N Day, Hans Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David J Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005.

[45] Brian W Kernighan and Dennis M Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[46] David B Kirk and Wen-mei W Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[47] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012.

[48] Dmitry Korkin and Lev Goldfarb. Multiple genome rearrangement: a general approach via the evolutionary genome graph. *Bioinformatics*, 18(suppl 1):S303–S311, 2002.

[49] David Koufaty and Deborah T Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.

[50] E Scott Larsen and David McAllister. Fast Matrix Multiplies Using Graphics Hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 55–55, New York, NY, USA, 2001. ACM.

[51] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, 38(3):451–460, 2010.

[52] Chao-Lin Liu, Min-Hua Lai, Kan-Wen Tien, Yi-Hsuan Chuang, Shih-Hung Wu, and Chia-Ying Lee. Visually and Phonologically Similar Characters in Incorrect Chinese Words: Analyses, Identification, and Applications. *ACM Transactions on Asian Language Information Processing*, 10(2):10:1–10:39, 2011.

[53] Manuel López-Ibáñez. Ant colony optimization. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, pages 2353–2384, New York, NY, USA, 2010. ACM.

[54] Manuel Lozano and Christian Blum. A hybrid Metaheuristic for the Longest Common Subsequence Problem. In *Proceedings of the 7th international conference on Hybrid metaheuristics*, pages 1–15, Berlin, Heidelberg, 2010. Springer-Verlag.

[55] David Maier. The Complexity of Some Problems on Subsequences and Supersequences. *Journal of the ACM*, 25:322–336, April 1978.

[56] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Transactions on Graphics*, 23(3):787–795, 2004.

[57] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[58] Anton Nekrutenko and Wen-Hsiung Li. Transposable elements are found in a large number of human protein-coding genes. *Trends in Genetics*, 17(11):619–621, 2001.

[59] John Nickolls and William J Dally. The GPU Computing Era. *IEEE Micro*, 30:56–69, March 2010.

[60] Kang Ning. Deposition and extension approach to find longest common subsequence for thousands of long sequences. *Computational Biology and Chemistry*, 34(3):149–157, 2010.

[61] Travis E Oliphant. Python for Scientific Computing. *Computing in Science Engineering*, 9(3):10–20, 2007.

[62] Kunle Olukotun, Basem A Nayfeh, Lance Hammond, Kenneth G Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. *Operating Systems Review*, 30:2–11, 1996.

[63] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D Antonopoulos. Synthesis of Platform Architectures from OpenCL Programs. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 186–193, Washington, DC, USA, 2011. IEEE Computer Society.

[64] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[65] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[66] Mike Paterson and Vlado Dančík. Longest Common Subsequences. In *Proceedings of the 19th International Symposium on Mathematical Foundations of Computer Science*, pages 127–142. Springer, 1994.

[67] Andre Pimenta, Edson Justino, Luiz S Oliveira, and Robert Sabourin. Document reconstruction using dynamic programming. In *Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1393–1396, Washington, DC, USA, 2009. IEEE Computer Society.

[68] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 1st edition, 2010.

[69] Timos K Sellis. Multiple-Query Optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[70] Dave Shreiner, editor. *OpenGL Reference Manual*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 4th edition, 2004.

[71] Amit Shukla and Suneeta Agarwal. A Relative Position based Algorithm to find out the Longest Common Subsequence from Multiple Biological Sequences. In *Proceedings of the 2010 International Conference on Computer and Communication Technology (ICCCT)*, pages 496–502, 2010.

[72] Shyong Jian Shyu and Chun-Yuan Tsai. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Computers and Operations Research*, 36:73–91, January 2009.

[73] Abhilash Singireddy. Solving the Longest Common Subsequence Problem in Bioinformatics. Master's thesis, Kansas State University, 2003.

[74] Christine Solnon. *Ant Colony Optimization and Constraint Programming*. Wiley-IEEE Press, 1st edition, 2010.

[75] James A Storer. *Data compression: methods and theory*. Computer Science Press, Inc., New York, NY, USA, 1988.

[76] Thomas Stützle and Holger H Hoos. MAX-MIN Ant system. *Future Generation Computer Systems*, 16:889–914, June 2000.

[77] Farzaneh Sadat Tabataba and Sayyed Rasoul Mousavi. A hyper-heuristic for the Longest Common Subsequence problem. *Computational Biology and Chemistry*, 36(0):42–54, 2012.

[78] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 1st edition, 2009.

[79] Chris J Thompson, Sahngyun Hahn, and Mark Oskin. Using Modern Graphics Architectures for General-purpose Computing: A Framework and Analysis. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 306–317, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[80] Guido van Rossum. The Python programming language. Accessed April 19, 2013. `http://www.python.org/`

[81] Robert A Wagner and Michael J Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, 1974.

[82] Qingguo Wang, Dmitry Korkin, and Yi Shang. Efficient Dominant Point Algorithms for the Multiple Longest Common Subsequence (MLCS) Problem. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, pages 1494–1499, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[83] Qingguo Wang, Dmitry Korkin, and Yi Shang. A Fast Multiple Longest Common Subsequence (MLCS) Algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 23(3):321–334, 2011.

[84] Qingguo Wang, Mian Pan, Yi Shang, and Dmitry Korkin. A Fast Heuristic Search Algorithm for Finding the Longest Common Subsequence of Multiple Strings. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010*, pages 1287–1292, 2010.

[85] Robin M Weiss. GPU-Accelerated Ant Colony Optimization. In Wen-mei W Hwu, editor, *GPU Computing Gems Emerald Edition*, pages 325–340. Morgan Kaufmann Publishers Inc., 2011.

[86] Hsiang-Yi Weng, Shyue-Horng Shiau, Kuo-Si Huang, and Chang-Biau Yang. A Hybrid Algorithm for the Longest Common Subsequence Problem. In *Proceedings of the 26th Workshop on Combinatorial Mathematics and Computation Theory*, pages 122–129, 2009.

[87] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31:50–59, March 2011.

[88] Jiaoyun Yang, Yun Xu, and Yi Shang. An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs. *Lecture Notes in Engineering and Computer Science*, 2183(1):499–504, 2010.

[89] Jiaoyun Yang, Yun Xu, Guangzhong Sun, and Yi Shang. A New Progressive Algorithm for a Multiple Longest Common Subsequences Problem and Its Efficient Parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):862–870, 2013.