SINTEF

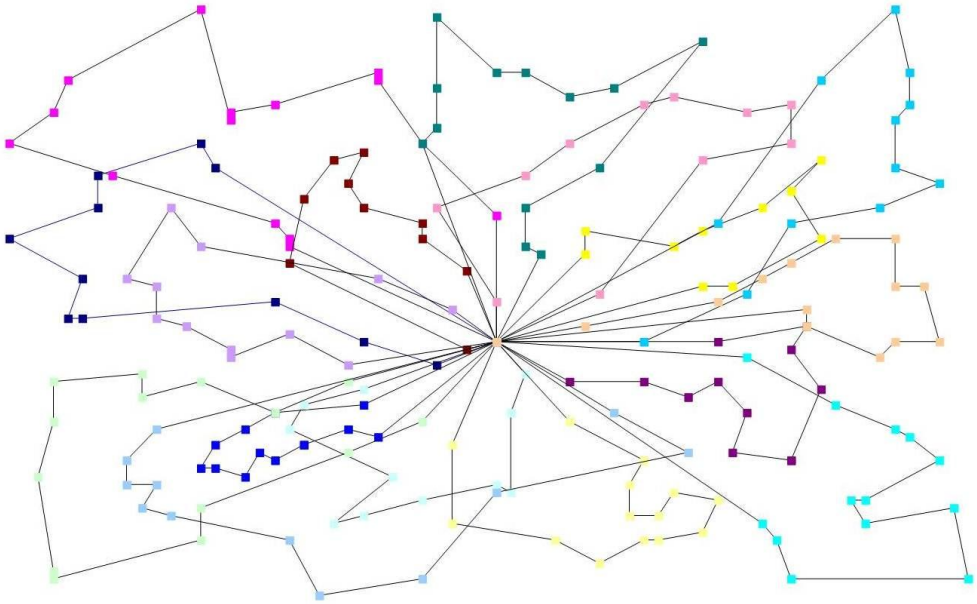
SINTEF A19678 Unrestricted

Report

Efficient Local Search on the GPU

Investigations on the Vehicle Routing Problem

Author(s)
Christian Schulz



SINTEF ICT
Applied Mathematics
2011-05-24

SINTEF

SINTEF IKT
SINTEF ICT

Address:
Postboks 124 Blindern
NO-0314 Oslo
NORWAY

Telephone:+47 73593000
Telefax:+47 22067350

postmottak.IKT@sintef.no
www.sintef.no
Enterprise /VAT No:
NO 948 007 029 MVA

Report

Efficient Local Search on the GPU

Investigations on the Vehicle Routing Problem

KEYWORDS:
GPU, discrete optimization, local search, CUDA, VRP, efficient implementation, parallel computing, stream processing

VERSION

DATE
2011-05-24

AUTHOR(S)
Christian Schulz

CLIENT(S)
The Research Council of Norway

CLIENT'S REF.
192905/I40

PROJECT NO.
90A3 6301

NUMBER OF PAGES/APPENDICES:
21

ABSTRACT
We investigate the performance increase potential of GPU implementations of local search. In particular, we report on how we managed to incrementally improve the implementation of a local search algorithm to a given GPU platform for maximum performance. As our target problem we use the well known Vehicle Routing Problem (VRP). The VRP is a family of computationally very hard problems with high industrial relevance. In particular, we investigate the 2-opt and 3-opt neighborhoods for the Distance constrained Capacitated VRP (DCVRP). Our final GPU implementation utilizes the GPU architecture efficiently. It is nearly one order of magnitude faster than the first implementation.

PREPARED BY
Christian Schulz

SIGNATURE
Christian Schulz

CHECKED BY
Tomas Nordlander

SIGNATURE

APPROVED BY
Geir Hasle

SIGNATURE
Geir Hasle

REPORT NO.
SINTEF A19678

ISBN
978-82-14-04984-8

CLASSIFICATION
Unrestricted

CLASSIFICATION THIS PAGE
Unrestricted

SINTEF

Document history

VERSION	DATE	VERSION DESCRIPTION
1	2011-05-24	First version

Efficient Local Search on the GPU – Investigations on the Vehicle Routing Problem

Christian Schulz

June 15, 2011

Abstract

In the recent years the graphics processing unit, or GPU, changed from being purely graphics oriented to a more general, programmable hardware. It is a very powerful, intrinsicly parallel machine that employs the idea of data parallelism, i.e., the same procedure is performed on different data. At the same time the GPU is nowadays a common part in most PCs, providing a very reasonable yet powerful tool if it can be harnessed. This has also been recognized by the scientific community. Research efforts have been reported on where and how the GPU can accelerate computationally expensive tasks in scientific computing. Discrete optimization is no exception. Local search is a computationally expensive method in discrete optimization. It is a basic ingredient in many so called metaheuristics, which in the recent years have proven to be very successful in providing high quality solutions for real world, large size discrete optimization problems. In local search the same type of operation is performed on a large set of data, exposing exactly the kind of data parallelism the GPU is designed for. Earlier studies of local search implementations on the GPU have verified the potential. However, a thorough investigation of how well the local search process can be adapted to the specific requirements of the GPU has been lacking so far. In this paper, we investigate the performance increase potential of GPU implementations of local search. In particular, we report on how we managed to incrementally improve the implementation of a local search algorithm to a given GPU platform for maximum performance. Our final version uses the GPU architecture efficiently and is nearly one order of magnitude faster than the first implementation. As our target problem we use the well known Vehicle Routing Problem (VRP). The VRP is a family of computationally very hard problems with high industrial relevance.

Keywords: GPU , Discrete Optimization , Local search , CUDA , VRP , Efficient implementation , Parallel computing , Stream processing

1 Introduction

Discrete Optimization Problems (DOPs) abound in business, industry, and the public sector. Their computational complexity (most of them are NP-hard) more often than not makes human problem solving inadequate, leaving a large potential for improvement of critical factors such as economy, customer service, and environmental damage. In many application areas a decision support tool industry has emerged. The performance of such tools is largely depending on two factors: the performance of the DOP solution method, and the computing power of the hardware platform. DOPs are also important in science. An increase in our ability to solve large-size DOPs in reasonable time contributes to scientific progress in fields such as biology, chemistry, and physics.

Solution methods for DOPs can largely be divided in exact and approximative methods. Exact methods guarantee to find an optimal solution, but often the response time is forbidding for real-life DOPs. Approximative methods, so-called metaheuristics [7, 21] in particular, have proven effective in providing high quality solutions to practical instances of many types of DOP under realistic response requirements. A basic ingredient in many metaheuristics[1] is *Local Search* (LS), also called *neighborhood search*. The bulk of the computational effort of sequential implementations of such metaheuristics is typically concentrated on feasibility and objective assessment of neighbors in LS. Other, so-called evolutionary metaheuristics, are based on a population of solutions. Analogously, their bottleneck is normally the assessment of each solution in the population.

The general DOP resolution power has increased tremendously over the past half century. This is due to a combination of methodological improvements and an exponential development of the power of commodity computers. To illustrate, commercial Linear Programming solvers had a speedup factor of roughly six orders of magnitude in the period 1987-2000 according to Bixby [3]. He attributes roughly a factor 1000 to better methods, and the same for more powerful hardware. Linear Programming is a basis for exact DOP methods. Still, there is a large gap between user requirements and state-of-the-art solver performance for many DOP applications.

Around year 2004, the architecture of processors for commodity computers started to change. Due to technological limits, the still prevailing Moore's law no longer materialized in the form of a doubling of clock speed every 18 months or so. Hence, the tongue-in-cheek "Beach law"[2] was no longer

[1] So-called single solution or trajectory based metaheuristics.

[2] One way of doubling the performance of your computer program

true. Multi-core processors with an increasing number of cores and higher theoretical performance than their single core predecessors emerged, but each core had lower clock speed. DOP methods need task parallelization to benefit from this development. In addition, there has been a drastic improvement of performance and general programmability of stream processing accelerators such as Graphics Processing Units (GPUs). To fully profit from the general recent and future hardware development on modern PC architectures, heterogeneous DOP methods that combine task and data parallelism must be developed. Such methods should self-adapt to the hardware resources at hand. The heterogeneous architecture also invites to a fundamental re-thinking of DOP methods.

The GPU has been utilized for scientific computing for a decade or so, for instance in linear algebra, geometry, visualization, and PDE-based simulation. There is a substantial research literature, see [5, 4]. In contrast, the literature on GPU implementations of DOP methods is scarce.

An early approach to implement tabu search on the GPU using the graphics pipeline was done by Janiak et al in 2008 [15]. With the development of CUDA (see later) programming the GPU became easier. In [16] Liu uses the GPU for VLSI circuit optimization. Vidal and Alba apply a cellular genetic algorithm on the GPU in [23]. Evolutionary algorithms as well as versions of local search were also studied by Luong et al. [17, 18]. Their results show that a GPU implementation can outperform the CPU.

In this paper, we investigate GPU implementations of LS. The study is an important step in research on heterogeneous computing for DOP on modern PC architectures. To this end, we utilize a hard DOP, the much studied Vehicle Routing Problem (VRP). In this context, our goal is neither to suggest a competitive VRP solution method, nor to prove once more that the GPU has high computing power. Rather, our main goal is to carefully assess the performance increase potential for GPU implementations of LS. In particular, we report on how we managed to incrementally improve the implementation of an LS algorithm to a given GPU platform for maximum performance.

The remainder of this paper is structured as follows. In Section 2, we informally describe the VRP and define the DCVRP variant that we use in our investigations. In Section 3, we define the Discrete Optimization Problem and give an introduction to local search. In Section 4 we describe the overall DCVRP solution method and the representation used. An introduction to programming the GPU with CUDA and related tools is given in Section 5. In Section 6 we describe in detail our implementation on the GPU and the steps undertaken to tune the algorithm. Section 7 shortly discusses the problem of filtering neighborhoods on the GPU. The paper finishes with our conclusion in Section 8.

is to go to the beach for two years and then buy a new computer.

2 The Vehicle Routing Problem

The VRP is a family of hard discrete optimization problems with high industrial relevance [10]. The classical Capacitated VRP (CVRP) was first described in the Operations Research literature in [6]. Since then, thousands of papers have been written on variants of the VRP. The CVRP is informally described as follows:

> *A number of identical vehicles with a given capacity are located at a central depot. They are available for servicing a set of customer orders, (all deliveries, or, alternatively, all pickups). Each customer order has a specific location and size. Travel costs between all locations are given. The goal is to design a least-cost set of routes for the vehicles in such a way that all customers are visited once and vehicle capacities are adhered to.*

For a thorough treatment of the VRP, we refer to [22] and [8], but for easy reference we give a definition of the CVRP here.

2.1 The Capacitated Vehicle Routing Problem

The classical (symmetrical) CVRP is defined on a weighted graph $G = (N, E)$. $V = \{1, \ldots, K\}$ is a set of identical vehicles, each with capacity Q. The graph nodes $N = C \cup D$, where $C = \{1, \ldots, n\}$ represents the set of customer locations, and $D = \{n + 1, \ldots, n + K + 1\}$ is the set of (artificial) depot nodes, all with the same location.

The edges $e \in E \subseteq N \times N$ represent travel possibilities between nodes. G is usually complete. A non-negative travel cost w_e, $e \in E$ is associated with each edge. For notational convenience, we shall use $w_{i,j}$ instead of $w_{(i,j)}$ for edge costs. In the literature, it is normal to use the Euclidean distance. A potential service cost particular to each customer may be represented by adding half of it to the cost of each incoming and outgoing edge of the customer.

A transportation order exists for each customer j, each with a non-negative demand q_j, requiring that the demand must be delivered by a single vehicle from the depot to the customer[3].

The goal in CVRP is to find a solution consisting of K *circuits*, i.e., one round trip *tour* for each vehicle, starting and stopping at the depot, with minimal total travel cost. Some circuits may be empty, i.e., one does not need to use all available vehicles. All customers must be served exactly once (i.e., all transportation orders must be serviced, and no split deliveries are allowed), and the total demand for each tour must not exceed the vehicle capacity Q. In a consistent CVRP instance we obviously have $Q \geq q_j$, $j \in \{1, \ldots, n\}$, and $KQ \geq \sum_{j=1}^{n} q_j$.

Let $\mathbf{s} = \{R_1, \ldots, R_K\}$ denote a candidate solution, where $R_k = (n+k, c_{k,1}, \ldots, c_{k,m_k}, n+k+1)$ denotes the route for

[3] Equivalently, all orders specify pickups from the customers to the depot.

vehicle k, $k = 1, \ldots, K$, i.e., the sequence of nodes starting and finishing with the depot and visiting m_k customers. Let $R_k(j)$, $j = 0, \ldots, m_k + 1$ denote the j-th node in route R_k, starting and finishing with the depot. The associated *route cost* $z(R_k)$ is simply the sum of costs for all edges defined by the node sequence:

$$z(R_k) = \sum_{j=0}^{m_k} w_{R_k(j),R_k(j+1)}$$

Let $C(R_k) = \{c_{k,1}, \ldots, c_{k,m_k}\}$ denote the set of customer nodes in route R_k. For a feasible solution $\mathbf{s}$ we have $\cup_{k=1}^{K} C(R_k) = C$ (all customers visited), $C(R_i) \cap C(R_j) = \emptyset$ for $i \neq j$ (no customer in more than one route), $\forall i, j \in \{1, \ldots, m_k\}$, $i \neq j : R_k(i) \neq R_k(j)$ (no route visits the same customer more than once), and

$$\sum_{j \in C(R_k)} q_j \leq Q, \ k = 1, \ldots, K$$

(no route violates the vehicle capacity).

The objective to be minimized for the CVRP is the sum of route costs over all routes:

$$\min Z(\mathbf{s}) = \sum_{i=1}^{K} z(R_i)$$

The optimal solution may require fewer than K routes. If there is a goal of minimizing the number of vehicles used, fixed costs for non-empty routes can be added to the arcs from the depot nodes to customers.

There is a basic extension to the CVRP: the Distance Constrained Capacitated VRP (DCVRP). For the DCVRP, there is an additional constraint: a fixed maximum cost (or length or duration) of each route must not be exceeded: $z(R_i) \leq L, \ i = 1, \ldots, K$.

Current state-of-the-art exact methods for the DCVRP can consistently solve instances with up to some 100 customers [2] in reasonable time. Real life VRPs may have thousands or tens of thousands of customers. For solving large-size practical cases one needs to resort to some type of approximative solution method, for instance metaheuristics based on LS[4].

In this paper we use the DCVRP as the DOP for which we investigate efficient GPU implementations of local search.

3 Discrete Optimization Problems and Local Search

Here, we give a formal definition of the DOP and the sequential variant of local search (LS). For a comprehensive treatment, we refer to [1].

We define a *Discrete Optimization Problem* (DOP) in the following way. A (usually finite) set of *solutions* $\mathcal{S}'$ called *the solution space* is given. A subset $\mathcal{S} \subseteq \mathcal{S}'$ is defined as the set of *feasible solutions*, also called the *search space*.

A function $z : \mathcal{S}' \to \mathbb{R}$, called *the objective* is defined on the solution space. The goal is to find a *global optimum*, i.e., a feasible solution $\mathbf{s}^* \in \mathcal{S}$ such that the objective z is minimized[5]:

$$z(\mathbf{s}^*) \leq z(\mathbf{s}), \forall \, \mathbf{s} \in \mathcal{S}$$

Hence, a DOP may be defined by a pair $(\mathcal{S}, z)$. Often, the solution space is given by the combinatorial nature of the problem at hand, for instance all permutations of cities in a travelling salesman problem. Also, the search space $\mathcal{S}$ is typically not given explicitly but defined implicitly through *variables* and their *domains*, together with a set of *constraints* $\mathcal{Z} = \{\zeta\}$ on these variables. Each constraint is a function $\zeta : \mathcal{S}' \to \{0, 1\}$. Let $\mathbf{s}' \in \mathcal{S}'$ be a (candidate) solution. A constraint has the property:

$$\zeta(\mathbf{s}') = \begin{cases} 0 & \text{if the solution } \mathbf{s}' \text{ is infeasible with respect to } \zeta, \\ 1 & \text{if the solution } \mathbf{s}' \text{ is feasible with respect to } \zeta. \end{cases}$$

Given the constraint set $\mathcal{Z}$, the set of feasible solutions (search space) $\mathcal{S}$ is defined by:

$$\mathcal{S} = \{\mathbf{s}' \in \mathcal{S}' : \zeta(\mathbf{s}') = 1, \forall \zeta \in \mathcal{Z}\}$$

We note that the DCVRP is a DOP according to the definition above.

Local Search (LS), also called *Neighborhood Search*, is based on the idea of improvement through modifications of a *current solution*. More formally, a neighborhood $\mathcal{N}$ is defined as a function $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$, where $2^{\mathcal{S}}$ denotes the set of all subsets of $\mathcal{S}$. Given a solution $\mathbf{s}$, $\mathcal{N}(\mathbf{s}) \subseteq \mathcal{S}$ is called the *Neighborhood* of $\mathbf{s}$. Normally, but not necessarily, a neighborhood is a small subset of the solution space.

A neighborhood is typically defined on the basis of a certain type of operation defined by an *operator* on the solution. Several operators may be defined for a given DOP. For the DCVRP we may for instance use the k-opt operator that selects k edges in a solution and replaces them with k others.

We define a neighborhood *move* as the acceptance of a neighbor solution as the next current solution. The neighborhood generated by an operator represents *potential moves* or *neighbors*. The neighbors must be checked for feasibility and change in objective value before a move is selected. These checks are typically the computational bottleneck in LS.

Let $\mathcal{N}$ be a Neighborhood function for the DOP $(\mathcal{S}, z)$. We define a solution $\hat{\mathbf{s}}$ to be a *local optimum* if the following holds:

$$z(\hat{\mathbf{s}}) \leq z(\mathbf{s}), \forall \, \mathbf{s} \in \mathcal{N}(\hat{\mathbf{s}})$$

Observe that a local optimum is defined relative to a specific neighborhood. Different neighborhoods will typically give rise to different sets of local optima.

[4] Other remedies to contain complexity are decomposition and abstraction.

[5] A maximization problem may easily be transformed to a minimization problem.

LS is an iterative improvement procedure. In its basic form, LS moves between feasible solutions. It needs an *initial solution* $\mathbf{s_0} \in \mathcal{S}'$ that is the first current solution. There are alternative general methods for producing an initial solution. These include random generation, greedy construction, exact resolution of a reduced or relaxed problem, and perturbation of a previously found solution. A common approach is greedy construction. Here, one starts with an empty solution and uses a more or less myopic and computationally cheap heuristic to incrementally build a feasible solution.

Starting with the initial solution $\mathbf{s_0}$, LS moves iteratively from one solution to a better solution. In the kth iteration, it searches the neighborhood of the current solution $\mathbf{s_{k-1}}$ for an improving solution. LS stops when there are no improving solutions in the neighborhood of the current solution. It follows from the definition above that the final solution is a local optimum relative to the neighborhood used. If the search space $\mathcal{S}$ is finite, LS must stop in some iteration T, where $T < |\mathcal{S}|$.

Note that the above does not precisely define LS. There may be many improving solutions in the neighborhood of the current solution. A *strategy* for the acceptance of improving neighbors is needed. The most common acceptance strategy is *Best Improving*: the best (or one of the best, with a deterministic tie-breaker) improving neighbor is accepted. Alternative strategies may be employed. For instance, one may accept the *First Improving* neighbor. Given an initial solution, LS has the *anytime property*: It may be interrupted at any time and still provide a useful solution. LS, with a specific search strategy, will define a path in the search space from the initial solution to a local optimum: $(\mathbf{s_0}, \ldots, \mathbf{s_T})$.

Pseudo-code for a sequential version of local search with the Best Improving strategy is shown in Listing 1. It should be clear that LS has parts that are embarrassingly parallel, most notably, the neighborhood exploration.

```
Procedure LocalSearch(s0,z,N,C)
begin
  current:=s0; // This is the initial solution
  localopt:=false;
  while not localopt do
  begin
    (current,localopt):=
     ExploreNeighborhood
     (current,N(current),f,C);
    if localopt then return current;
  end
end

Procedure ExploreNeighborhood
          (current,Neighbors,f,Constraints)
begin
  bestneighbor:=current;
  for n in Neighbors do
  begin
    feasible:=CheckFeasibility(n,Constraints);
    if feasible and f(n) < f(bestneighbor) then
    begin
       bestneighbor:=n
    end
  end
  return (bestneighbor,bestneighbor=current)
end

Procedure CheckFeasibility
          (solution,Constraints)
begin
  for c in Constraints do
  begin
    if c(solution)=0 then return false;
  end
  return true;
end
```

Listing 1: Local Search with Best Improving strategy.

4 Solution Method and Representation

Here, we describe the overall DCVRP solution method and the representations used in our investigations of efficient GPU implementations of local search.

4.1 Solution method

The optimization method is LS with the k-opt operator, for $k = 2, 3$, see Section 3. These operators remove k edges from the current solution and combine the resulting segments in all possible ways. The reason for selecting these operators are their widespread use, their generic nature, and the fact that their cardinalities are $O(n^2)$ and $O(n^3)$, respectively. Normally, the 3-opt neighborhood is regarded as being too expensive for large VRPs.

The initial solution is generated either with a cheapest insertion heuristic or by assigning one vehicle to each customer. Neither construction method can guarantee that a

given constraint on the number of vehicles is respected. In order to remove such infeasibilities, we utilize an augmented objective z with a penalty term z_K during the subsequent LS phase:

$$z = z_w + \lambda_K z_K, \qquad z_w = Z(\mathbf{s}).$$

If the number of vehicles in the solution is valid, $z_K = 0$, otherwise it is equal to the number of vehicles in the solution. The weight λ_K is such that a solution with fewer vehicles is always preferable. However, there is no guarantee that LS finds a feasible solution if the initial solution is infeasible. Since the goal of this paper is not to provide a new method for solving the DCVRP but to study GPU implementations of local search, we believe that this approach is reasonable.

4.2 Representation

For candidate VRP solutions, we use a giant tour representation, i.e., a sequence of customer nodes with artificial depot nodes as delimiters between routes:

$$\mathbf{s} = (n{+}1, c_{1,1}, \ldots, c_{1,m_1}, n{+}2, \ldots, n{+}K, c_{K,1}, \ldots, c_{K,m_K}, n{+}K{+}1)$$

As edge cost we simply use the Euclidean distance.

Most of the work in a LS iteration consists of generating the neighborhood, evaluating the neighbors with respect to the objective and constraints, and comparing them to the current solution [14]. A neighbor, or potential move, in a k-opt neighborhood is defined by the k-opt operator that replaces k current edges with new ones. The k-opt operator combines $k+1$ segments of the current solution in the giant tour representation.

More formally, a segment P is simply a part of the solution $\mathbf{s}$, i.e., an arbitrary subsequence of nodes $\eta_i \in N$ in the giant tour representation. A segment may span more than one route. When combining two segments P_1 and P_2, we will always assume that P_1 ends on the same node as P_2 starts from, i.e., $P_1 = \eta_0, \ldots, \eta_n$ and $P_2 = \eta_n, \ldots, \eta_k$.

The cost associated with a segment is simply the sum of edge costs for the path from the start node to the finish node:

$$z_w(P) = \sum_{i=1}^{n} w_{\eta_{i-1},\eta_i}.$$

The cost of a combined segment is the sum of the two segment costs:

$$z_w(P_1 + P_2) = z_w(P_1) + z_w(P_2).$$

In the following we describe a way to represent the influence of each segment to the constraints and the objective, and how this representation can be used in order to compute the influence of the combined segments. For details, we refer to [14].

4.3 Resource Extension Functions

We use classical resource extension functions (CREFs) as explained by Irnich [13, 14]. In a resource extension function (REF), the consumption related to a constraint or objective is represented by a resource t. The resource can for example be the cost of the route so far, or the current load of the vehicle. The change in the resource along an edge (η_i, η_j) is modelled by a REF $f_{i,j} : \mathbb{R} \to \mathbb{R}$. In addition there is a resource interval $[a_i, b_i]$ associated with each node η_i in the graph, which describes the feasible values for the resource at this node. A segment P is considered feasible with respect to the resource t if and only if there exists for each node $\eta_i \in P$ a resource value $T_i \in [a_i, b_i]$ such that for every edge $(\eta_i, \eta_{i+1}) \in P$ we have

$$f_{i,i+1}(T_i) \leq T_{i+1}.$$

A CREF is a REF of the form

$$f_{i,j}(T) = \max(a_j, T + t_{i,j}),$$

where $t_{i,j} \in \mathbb{R}$ models the (constant) change in the resource along the edge (η_i, η_j). For the capacity constraint we simply use $a_i = 0$, $b_i = Q$, $t_{i,j} = -Q, j \in D$ and $t_{i,j} = q_j$, $i \in N$, $j \in C$. For information about how two CREFs of two segments can be merged to generate the CREF of the combined segment, we refer to [13, 14].

The route distance constraint in a DCVRP can also be modeled using CREFs with $a_i = 0$, $b_i = L$ with L being the maximum cost on a single route and $t_{i,j} = w_{\eta_i,\eta_j}$. This approach has however one problem, the resource needs to be reset in the finishing depot node of a route. This is one of the reasons why Irnich suggests to split depot nodes into start depot nodes and end depot nodes. We choose not to do this, as it would yield the need for another constraint related to end depot nodes being followed by start depot nodes. Instead we apply an implicit separation of each depot node in two nodes. This means, that any CREF for an edge ending in a depot node is combined with a reset CREF f_R with $a_R = 0$, $b_R = L$ and $t_R = -L$ where $R \in D$.

Using CREFs to model the capacity and route distance constraint gives the possibility to aggregate segments, which in turn leads to constant time evaluation of a potential move as we will see in the next paragraph. However, the penalty component of the objective can not be represented by CREFs. Nevertheless, we can represent all necessary information per segment and aggregate it. To do this, we only need the number of inner routes l_P in a segment P, meaning the number of non-empty complete routes inside the segment. In addition we need to know whether the segment starts with a depot node, ends with one, and whether it contains one at all. The information for the aggregated segment P from P_1 and P_2 is then

$$l_P = l_{P_1} + l_{P_2} + \gamma_{P_1,P_2}$$

where $\gamma_{P_1,P_2} = 0$ if one of the segments does not contain a depot node or if P_1 ends at a depot node and P_2 start at it. Otherwise, $\gamma_{P_1,P_2} = 1$.

Checking the feasibility of a segment with respect to the number of vehicles using our model is straightforward, as incomplete routes do not count and the giant tour will start and end with a depot node and thus l_P will be the number of all (non-empty) routes. However, it is more difficult to check the feasibility of a segment using REFs without iteratively combining all edges and checking for the bounds along the way. In [13] Irnich explains how using inverse REFs together with REFs enables a feasibility check of an aggregated segment using just the REFs and inverse REFs of the two segments which are combined. This means, the recombination and feasibility check of the segments generated during a k-opt neighbor evaluation can be done in constant time as long as we have the REFs and their inverse for the generated segments. By pre-computing those for all possible segments of the current solution, we get constant time neighbor evaluation.

Pre-computing the data for every possible segment in the current solution leads to $O(n^2)$ segments which need to be calculated and stored, where $n = |N|$ is the number of nodes in the current solution. In order to reduce this amount, Irnich suggests to use a hierarchy of segments [13]. Here the current solution is split into parts where each part contains the same number of edges (except the last one, which might contain less). Then all possible segments inside each part are pre-computed. In addition one calculates all possible segments that start and end at the first or last node of the current solution and at the nodes which split the parts. For the latter calculations the previously computed segments for each part can be used. This leads to a one-level hierarchy (following Irnich's terminology), where level 0 contains the segments of the parts and level 1 contains the segments with several parts. Of course, this idea can be extended by splitting the first level again, leading to a two-level hierarchy. In theory, we can continue that way to generate a l-level hierarchy. According to Irnich [12], the computational effort is minimal when each part contains $n^{1/3}$ nodes for a one-level hierarchy. For a two-level hierarchy, the minimal effort is achieved for $n^{1/7}$ nodes in each part of level zero and $n^{3/7}$ nodes in the parts of level one[6].

5 GPU and Development Tools

The primary task of a graphics processing unit (GPU) used to be to compute the image which is to be displayed on the screen. Such images consist of thousands of independent pixels that, in dynamic settings such as computer gaming, need to be computed several times per second. These requirements have lead to a highly parallel GPU architecture. Driven by the constant hunger for better graphics from the gaming industry, combined with the interest of non-graphic use of GPUs, these units developed into general purpose GPUs with high computing power [4]. The scientific community soon started to utilize such GPUs for computationally intensive tasks through parallel algorithms.

Originally, a GPU was designed to compute the image which is to be displayed on the screen by a given set of functions [4]. Any algorithm that was intended to run on a GPU had to be expressed as part of the graphics pipeline using graphics APIs such as OpenGL and DirectX. A modern general purpose GPU, however, is equipped with more programmer-friendly environments. Most important to the scientific community are programming languages that map data parallel algorithms to GPU architectures and accompanying development environments. The most important languages are OpenCL and CUDA. Whereas OpenCL is an open standard, CUDA is a proprietary programming language from NVIDIA, one of the main GPU manufacturers. For the work presented here we selected CUDA, primarily because it allows for more detailed control of the GPU. Also, there were sophisticated and mature development tools.

5.1 The Fermi architecture and CUDA

In this paper we only consider the Fermi architecture, the latest architecture for NVIDIA GPUs. Different architectures support different aspects of CUDA. This is indicated by the *compute capability* of the GPU. Fermi GPUs have compute capability ≥ 2.0. A GPU with compute capability 2.0 has on board main memory (GPU memory) and several streaming multiprocessors (SM), each consisting of 32 cores. Each SM has 64 KB of memory, 32768 32-bit registers, and other elements which we will not discuss further. The memory of a SM is used as shared memory and cache, as will be discussed later.

Programming the GPU using CUDA is by itself a relatively simple process. CUDA extends the C++ language by a set of keywords that enable the user to write functions, so called *kernels*, that are executed on the GPU. Inside a kernel nearly the whole C++ language can be used [7]. Moreover CUDA provides functions for copying data to and from the GPU as well as configurating the GPU.

The difficult part of GPU programming is to write an efficient program that uses the specifics of the GPU for maximum performance. For this, one needs to understand the architecture and how a kernel is executed on the GPU. Arguably, substantial parts of the scarce literature on GPU implementations of metaheuristics describe a very basic, straightforward approach that does not utilize the full computing power of GPUs. Our main goal here is to show that substantial speedups follow from careful implementation, and to identify the main steps and relevant implementation issues of a performance optimization process.

5.2 Kernel execution

A kernel is executed on a *compute grid* consisting of a number of *blocks*, where each block again contains a number of *threads*, as illustrated in Figure 1. All of the threads execute the same kernel. The number of blocks in the three

[6] This implies that $n^{2/7}$ parts of level zero form one part of level one.

[7] For CUDA v3.2, only virtual inheritance is missing.

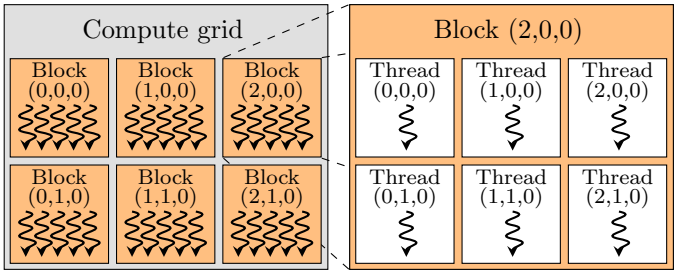


Figure 1: Illustration of CUDA compute grid, block and thread hierarchy.

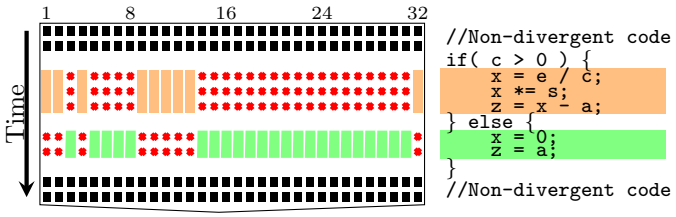


Figure 2: Warp serialization and masking.

dimensional grid is defined by the programmer. However, the maximum number of blocks in the z-direction is currently 1 (for CUDA v3.2). The threads in a block also form a 3 dimensional layout whose limits are again defined by the programmer. Nevertheless, the maximum number of total threads per block is currently 1024. CUDA provides each thread with the index of its block and its index within the block. All threads in one block are executed on the same SM, but different blocks in the compute grid can be executed on different SM. When a block is executed, the threads are split into so called *warps* of 32 threads. Therefore it makes sense to have a multiple of 32 threads in each block. The 32 threads of one warp execute the same instruction at any time. Divergence in the code within one warp is handled by the GPU through warp serialization and masking, as illustrated in Figure 2. This means that while some of the threads in the warp execute the instructions of one branch, the other threads are idle. In the worst case all 32 threads would take different branches. Therefore it is important to minimize divergence in a warp.

The result of an arithmetic instruction, or data read from global GPU memory in a thread, is not available to this thread in the next cycle. The GPU solves this latency problem by switching to a different warp when one of the threads within the current warp needs to wait for a result. This switching comes without any overhead, thanks to the GPU hardware. The new warp can be from the same or a different block as the current warp. A Fermi GPU can run up to 8 blocks per SM simultaneously. The latency of an arithmetic operation is around 18 cycles. As a rule of thumb one can therefore say that each SM should have at least $18 \cdot 32 = 576$ threads to work on. The maximum number of threads a SM can run at the same time is 1536, which corresponds to 48 warps.

When analyzing an algorithm with respect to its utilization of the GPU there are many measures one can and should consider. One of these is *occupancy*, which is the ratio of active warps on a SM to the maximum number of warps[8].

Figure 3: Illustration of the CUDA memory hierarchy.

5.3 Memory hierarchy

Similar to the thread hierarchy, there is also a memory hierarchy on the GPU as shown in Figure 3. The registers of an SM are distributed evenly over the currently running threads. Thus each thread has some amount of registers available. Calculations are performed on the data stored in these registers, the results again are stored in registers. A register is private to the thread, meaning no other thread can read from or write to it. Since there is a limit of registers available per SM, there is a limit on registers per thread, depending on how many threads are running simultaneously on the SM. If a thread would need more registers than it has, data is stored in local memory. This is called *register spilling*.

Local memory is again private to each thread, however has the same high latency as global memory[9]. Actually, local memory is created by taking a part of the global memory and assigning it to the thread, making it private and local. Arrays where the indexing is not constant are also placed in local memory. In practice however it is difficult to know in advance whether an array is placed in the registers or local memory, as this depends on whether the compiler can detect constant access or not.

One part of the memory on a SM is used as shared memory. Shared means it is shared between all threads in one block, data stored in it can be read and written by all threads in a block. If a SM has several blocks running at the same time, the shared memory is split into parts and each block has its own shared memory. Accessing shared memory is slower than using registers, but faster than reading or writing to global memory.

Finally the main memory on the GPU is called global memory and can be accessed by all threads, blocks and grids. While data in local and shared memory is lost once the thread or block is finished, global memory will keep its data until explicitly released. Unfortunately, accessing GPU global memory is slow, just as on the CPU. On the Fermi GPU each SM therefore uses some if its memory as a cache for global (and local) memory access. The 64 KB memory of a SM can be split into either 48 KB shared memory and 16 KB cache or 16 KB shared memory and 48 KB cache. When

[8] 48 for the Fermi architecture.

[9] Here and in the following global memory always refers to global GPU memory.

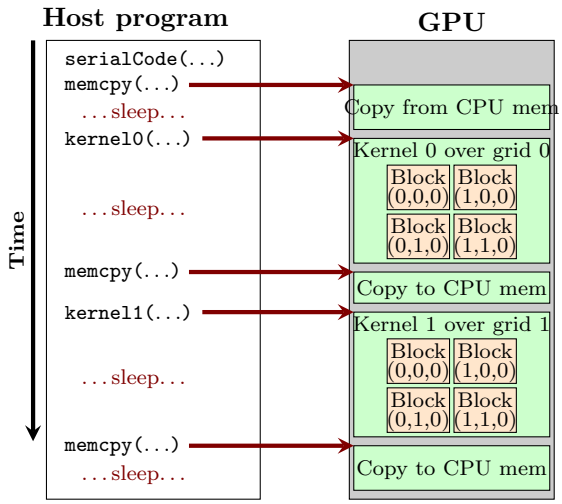


Figure 4: Simple use of GPU with single stream.

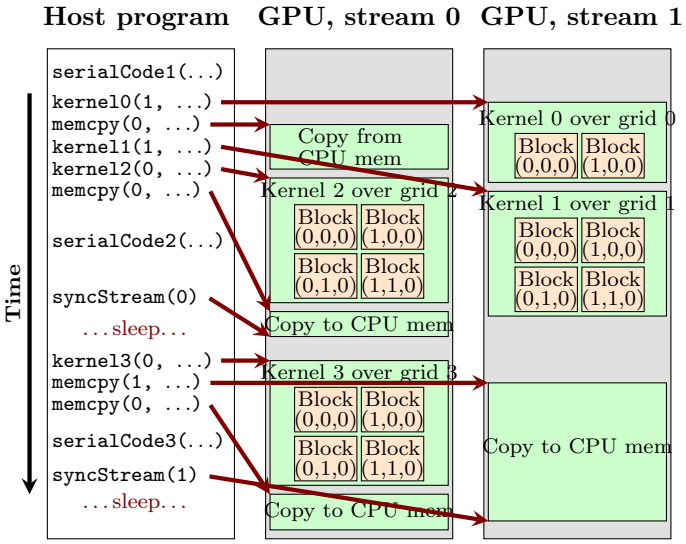


Figure 5: Asynchronous multi stream GPU usage.

a warp accesses the global memory, the access pattern influences the efficiency. It is more efficient when adjacent threads access adjacent memory locations than if those locations are scattered throughout the whole global memory. In the former case the access can be coalesced, meaning the whole area of memory can be read/written in few operations, ideally only one.

In addition, a GPU has texture memory and a constant cache. The latter speeds up uniform access, i.e., all threads in a warp access the same address, and access to constant data. Texture memory is usually used in graphics and provides faster access than global memory. It comes with free interpolation operations targeted at graphics. However, the interpolation precision is low, and the texture memory is optimized for accessing elements close together in a 2 dimensional setting. On older GPUs with compute capability 1.x, reading from texture memory could lead to speedups. On the Fermi architecture, however, global memory access is cached. Thus it can outperform the texture memory if used correctly [20].

5.4 GPU-CPU coordination

So far we only discussed how a kernel is executed on the GPU. The kernels have to be coordinated with the process running on the CPU. The CPU in this setting is also called *host*, whereas the GPU is called *device*. The device memory can be allocated, accessed and freed from the host process. Device memory access from the host means that data can be copied from the host memory to the device memory and vice versa using memory copy functions that are part of the CUDA library. In addition to explicit copying of data between device and host, host memory can be mapped to device memory so copying will occur automatically.

Figure 4 illustrates a simple program utilizing the GPU in a synchronized way. This means that each task is performed after the previous one, whether it involves the CPU, the GPU or both. Hence most of the time one of them is waiting for the other. Having a sleeping CPU part is usually no problem, since the program can use different CPU threads to perform work while waiting. However, Fermi GPUs have additional features to avoid latency. Host-device memory copy can overlap kernel execution. Several kernels can run concurrently if the grids are small enough. In addition, Fermi GPUs can have a host to device copy at the same time as a device to host copy. Hence, a synchronized work flow is often not utilizing the full power of the GPU. Instead it makes sense to have an asynchronous work flow as illustrated in Figure 5.

To utilize these features we need to be able to group tasks on the GPU together. This is done using *streams*. All tasks in one stream are executed in sequence, but tasks from different streams can be performed in arbitrary order. Coordination between a GPU stream and the host can be done by stream synchronization, as illustrated in Figure 5. Synchronizing a stream induces overhead. Subsequently started kernels on the stream are only transferred to the GPU after synchronization.

Often it is not necessary to synchronize the host with a stream, it suffices to coordinate them. As an example, assume we have a kernel k_1 on stream s. Another kernel k_2 can be performed right after k_1. Kernel k_3 can only be issued after the host gets the result of k_1. Synchronizing host and stream s after k_1 leads to an idle GPU because k_2 can only be issued after the synchronization. What one would like to implement on the host is the following: issue k_1, issue k_2, wait for k_1 to finish, issue k_3. This is what *events* are for. An event can be recorded on a stream and queried. An event is completed once all tasks on the stream before the event are finished. CUDA provides functionality to synchronize the host or a stream to an event. This has the advantage that the stream where the event is recorded does not have to synchronize itself with another stream or

the host, it can simply continue with its further tasks once an event is completed.

5.5 Performance measurement

NVIDIA provides an analysis tool called Compute Visual Profiler which can trace the performance of the program and provide various performance metrics. In addition, the NSIGHT plugin to Visual Studio provides timeline visualization of the program flow. Figure 7 is an example of such a timeline that will be used to illustrate performance of alternative GPU implementations.

The timeline is split into several rows, each showing activity related to a different source. The first row shows activity of the driver API, such as scheduling a kernel or a copying or synchronizing with the GPU. The second row displays memory transfer between the CPU and the GPU. The third row presents the activity of the GPU in terms of kernel execution or GPU to GPU copy operations. As explained above, kernels can be executed in different streams. The fourth row is a grouping row, followed by a row for each stream that shows the GPU activity in this stream. The counters group follows, consisting of three more rows: GPU device utilization, host to device bandwidth utilization, and device to host bandwidth utilization.

6 Implementations and Results

As mentioned in the introduction there is a wide variety of GPU implementation papers in different areas of scientific computing, amongst them a few in discrete optimization. A typical "selling point" of these papers are speedup factors of the GPU implementation relative to a CPU implementation. There are two problems with this. First, the comparison is only fair if both the CPU and GPU code are optimized. Considering the efforts involved in a proper GPU implementation, the CPU implementation would need to utilize multiple cores, SSE-instructions, and caching strategies. Such efforts are rarely seen in the literature. Second, it is already well known and accepted that GPUs are formidable and powerful tools for computationally intensive tasks. Despite the first problem described above, the GPU speedup factors reported in discrete optimization papers are generally not impressive. They seem to reveal a basic and fairly naive GPU implementation. Such investigations are not uninteresting, but the main challenge in *GPU programming proper* is to adapt a given algorithm to the GPU architecture in order to use as much of the computational power as possible.[10]

In this section we demonstrate how we adapted our local search algorithm for the DCVRP to the GPU through an incremental improvement process with careful tuning, experiments, and performance analysis. We used CUDA as programming language. All experiments were performed on a NVIDIA GeForce GTX 480, which is a Fermi architecture GPU with compute capability 2.0. The code was compiled using CUDA 3.2. As test instances, we have used a selection of 10 CVRP and DCVRP instances from the literature where the number of nodes in our giant tour representation vary between 50 and 2400. For instances that do not originally have a distance or number of vehicles constraint, such constraints have been added. The main performance measure is speedup relative to the previously best GPU implementation in the incremental improvement process. Our initial reference point is the first, basic GPU implementation that we refer to as *The Benchmark Version*.

[10] In a broader context, a main challenge and opportunity is to design fundamentally new methods that fully utilize the stream processing model and the GPU architecture, also in a heterogeneous architecture with a multi-core CPU.

The Benchmark Version
Setup problem instance data on CPU
Copy problem instance data to GPU
Create initial solution on CPU
Copy initial solution to GPU
Evaluate initial solution on CPU
Create k-opt mapping on CPU
Copy k-opt mapping to GPU
do
Create hierarchies on GPU
Evaluate all constraints and objectives on GPU
Find best neighbor on GPU
Execute best move on GPU
Copy best move to CPU
Execute best move on CPU
Evaluate new current solution on CPU
until local optimum or stop criterion

Figure 6: The Benchmark Version.

6.1 The Benchmark Version

Due to the fact that CUDA is basically C and C++, it is fairly easy to implement a first GPU implementation of LS for the DCVRP. The algorithm is outlined in Figure 6. As a preprocessing step the problem instance data is established on the CPU and copied to the GPU. Then the initial solution is created and evaluated on the CPU and copied to the GPU, before the mapping necessary to identify the k-opt neighbors on the GPU (see below) is generated on the CPU and copied to the GPU.

In each local search iteration, we first create the hierarchy for each objective and constraint on the GPU. We use a one-level hierarchy. The neighborhood evaluation is also completely executed on the GPU and the results are stored in global GPU memory. We then apply a classical reduction operation on the GPU [9] to find the best feasible neighbor and then execute the corresponding move directly to the solution on the GPU. In this way there is neither a need for copying the hierarchy nor *the neighborhood fitness structure*, i.e., the objective delta value and feasibility information for each neighbor, between CPU and GPU. The only copy operation necessary in one iteration is the copy of the selected move from the GPU to the CPU. One could use the value of the best neighbor to decide whether a local optimum has

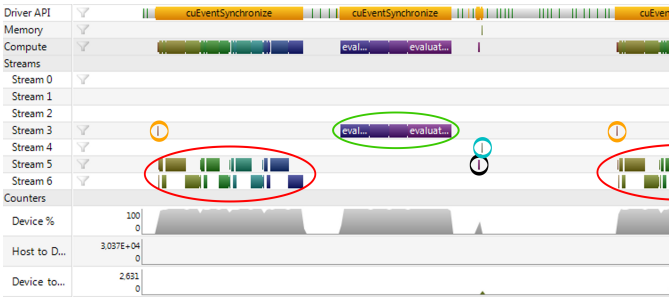


Figure 7: Timeline of one iteration in a 2-opt neighborhood for The Benchmark Version and 399 nodes in solution.

Figure 8: Timeline of one iteration in a 2-opt neighborhood for The Benchmark Version and 2401 nodes in solution.

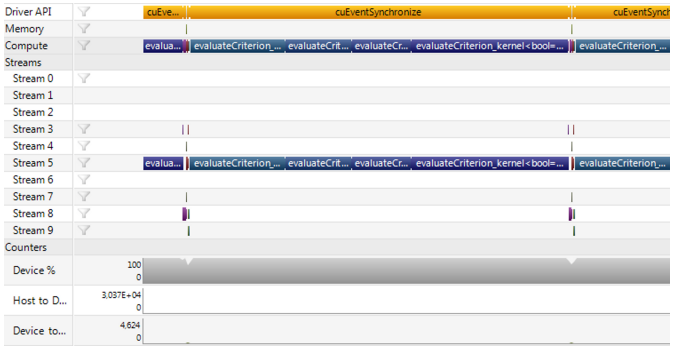


Figure 9: Timeline of one iteration in a 3-opt neighborhood for The Benchmark Version and 399 nodes in solution.

been reached. A CPU copy of the solution is redundant. However, we choose to execute the move on the CPU side also to keep an updated solution. This costs nearly no time relative to the neighborhood evaluation and it gives more flexibility.

A 3-opt neighbor is identified by a four dimensional index x, y, z, p where the $x < y < z$ coordinates specify the locations of the edges in the current solution that shall be removed. A 3-opt neighbor describes only pure 3-opt neighbors that can not be expressed by a 2-opt, yielding only four possibilities $p = 0, 1, 2, 3$ of recombining the resulting segments. A 2-opt neighbor is similarly described by a two dimensional index x, y where again $x < y$ describe the edges to be removed. There is only one way of recombining the segments, so there is no need for a p index. During the neighborhood evaluation one thread is responsible for one neighbor and identified by a one-dimensional index. The mapping between the one-dimensional thread index and the 3-opt neighbor index simply orders the 3-opt neighbors lexicographically. Since for each $x < y < z$ combination there are exactly four 3-opt neighbors, only the lexicographical order of all possible x, y, z values needs to be precalculated on the CPU and copied to the GPU. The mapping is constant during the whole local search and needs therefore to be generated and copied only once. The p component of the 3-opt neighbor index can easily be computed on the GPU during the neighbor evaluation.

The mapping from thread index to the 2-opt index also simply orders the 2-opt indices lexicographically. In contrast to the 3-opt mapping however, there is an explicit formula available for the mapping, see [18]. Let i be the thread index and $n+1$ the number of nodes in the solution, then

$$x = n - 2 - \left\lfloor \left(\sqrt{8\big((n(n-1))/2\big) - 8i - 7} - 1 \right) \Big/ 2 \right\rfloor, \qquad (1)$$
$$y = 1 + i - x(n-1) + \big(x(x+1)\big)/2.$$

In Figure 7 the different tasks during one 2-opt iteration are marked by colored ellipse. The orange circles indicate the initialization of delta values to zero and feasibility to true for all neighbors. The red ellipse show the generation of the hierarchies, followed by the green ellipse that marks the neighborhood evaluation. The reduction to the best neighbor is indicated by the black circle, and copying the best neighbor to the CPU is marked by the cyan circle. In Figure 7 the current solution contains 399 nodes. We can observe that here the generation of the hierarchies actually takes more time than the neighborhood evaluation. Most interesting is actually that the GPU is idle about 40% of the time. This picture changes when considering the 2-opt neighborhood of a solution with 2401 nodes as shown in Figure 8. We can see that for this instance, the GPU is busy most of the time. The neighborhood evaluation is the largest part of the computational effort. This result is expected, since the cost for creating the one-level hierarchy is $O(n^{4/3})$ whereas the number of neighbors in the 2-opt neighborhood is $O(n^2)$. The picture becomes even more clear when considering a 3-opt neighborhood with $O(n^3)$ neighbors, as shown in Figure 9.

Kernel		Time (ms)	Occ	Reg	Bw (Gb/s)	L1 (%)	Lm	Div
3-opt	ζ_c	346	0.50	37	137	84	24e6	69e4
	z_w	241	0.67	32	140	75	10e6	68e4
	z_K	217	0.50	35	98	88	13e6	48e4
	ζ_L	562	0.42	47	138	75	25e6	11e5
2-opt	ζ_c	0.56	0.50	36	131	80	35e3	1560
	z_w	0.38	0.50	36	117	76	15e3	1590
	z_K	0.36	0.50	36	93	82	20e3	1598
	ζ_L	0.85	0.42	45	129	76	37e3	2192

Table 1: Performance of kernels of The Benchmark Version applied to a solution with 399 nodes.

During the neighborhood evaluation we use up to eight kernels, one for each constraint and objective, and different kernels for 2-opt and 3-opt neighbors. All floating point arithmetic is done in double precision. The performance of these kernels is shown in Table 1, where the numbers are taken from the Compute Visual Profiler. Here, and in the following tables, z_w is the cost objective, ζ_c the capacity constraint, z_K the number of vehicles objective, and ζ_L the route distance constraint. The columns show the runtime of the kernel in milliseconds (time), the achieved occupancy (Occ), the number of registers used (Reg), the achieved bandwidth in Gbyte per second (Bw), the percentage of cache hits (L1), the number of local memory access operations (Lm) and the number of divergent branches (Div). Please note that L1, Lm and Div are measured only over one streaming multiprocessor and thus do not provide absolute numbers. The values however are useful to compare different implementations with each other [19].

The kernels use between 32 and 47 registers, hence there are not enough registers on a SM for 48 warps. This explains the observed occupancies of around 0.5. Although occupancy does not equal speed, a higher occupancy could enable the SM to hide more latency. The cache usage of 75 percent and above is not perfect, but a reasonable result.

The achieved bandwidth rates of 130 to 140 related to a theoretical maximum of 177.4 Gbyte per second indicate that the bandwidth is nearly fully used. Hence memory bandwidth is probably a limiting factor. This impression is strengthened by the fact that all kernels make heavy use of local memory which is slow and fills the bandwidth. In our CUDA compiler configuration each thread can use up to 63 registers, so register spilling is unlikely to be the cause of the local memory usage. Finally, all 3-opt kernels have a high number of divergent branches, which indicates that the parallelism in the code flow for different neighbors is far from ideal.

From the timeline analysis we know that for 3-opt neighborhoods in general and for 2-opt ones of large enough solutions, the neighborhood evaluation is the bottleneck of our algorithm. The performance study of the related kernels reveals that they are far from optimal. We will therefore show how to improve the kernels in the next subsections.

In the following we will present several figures of the type of Figure 10. They show the speedup of a new implementation relative to the current one. The speedup is computed by considering the time for one LS iteration. This time is calculated by measuring the time for 100 iterations. If a local optimum is found earlier, the search is restarted to reach 100 iterations.

6.2 Segments in registers

A major problem with the kernels in The Benchmark Version is the heavy local memory usage. This is due to the fact that for each k-opt neighbor we split the current solution in $k+1$ segments and store them in local memory. To reduce local memory usage we therefore move the segments into registers, which is possible as we only need to store the aggregated version of each segment. Figure 10 shows that this indeed leads to a good speedup compared to The Benchmark Version, although the register usage increases.

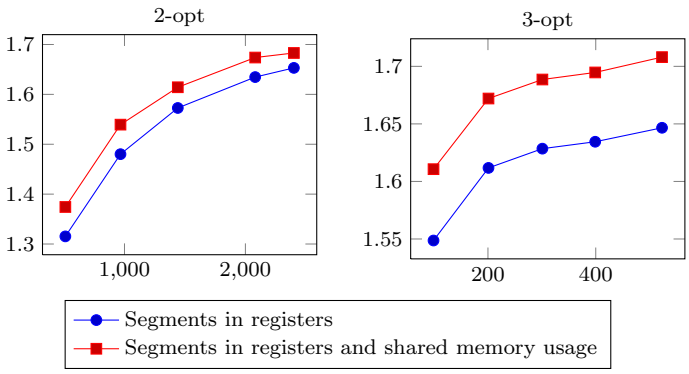


Figure 10: Speedup due to less local memory and use of shared memory. The x-axis shows the number of nodes in the current solution, the y-axis the speedup.

Our implementation consists of eight different kernels, two for each objective and constraint: one for the pure 2-opt neighborhood, and one for the pure 3-opt neighborhood. When examining certain performance measures of the implementation, such as the usage of local memory, they could in theory be different for all kernels. However, that would mean that each kernel needs a complete and independent implementation. Between the 2-opt and 3-opt kernels this is not always possible, as they use the same hierarchy if one does not want to represent it twice. Moreover, the concept of how a neighbor is evaluated is the same for all objectives and constraints. The only difference is the way the aggregation of two segments is performed. Therefore our implementation splits those two tasks.

We have one 2-opt and one 3-opt kernel that takes care of aggregating the correct segments by using aggregation operators. In this way each objective or constraint only needs to specify these operators to use the kernels, leading to an easy to use and extendible implementation. However, this design forces us in several cases to use the same aspects of the implementation for all objectives and constraints. Therefore the measurements in Figure 10 and similar ones later show the speedup for evaluating all objectives and constraints. For the 3-opt comparisons this means that also the 2-opt times are included, as the complete 3-opt neighborhood is defined by the union of the pure 3-opt neighbors and the 2-opt neighborhood.

6.3 Shared memory

When computing the aggregation of one segment, we need to traverse the pre-computed hierarchy. During this traversal the number of nodes in each part of the hierarchy have to be read several times by each thread from global memory. These numbers are stored in global memory as the hierarchy traversal implementation is generic for hierarchies with any

Kernel		Time (ms)	Occ	Reg	Bw (Gb/s)	L1 (%)	Lm	Div
3-opt	ζ_c	198	0.42	46	3	97	24e5	60e5
	z_w	171	0.50	36	6	84	0	60e5
	z_K	184	0.50	38	4	97	26e4	60e5
	ζ_L	249	0.33	63	8	86	34e5	96e5
2-opt	ζ_c	0.31	0.50	40	9	95	4187	1010
	z_w	0.26	0.50	36	9	90	331	1038
	z_K	0.28	0.50	36	5	96	989	1034
	ζ_L	0.41	0.33	54	13	89	5404	1593

Table 2: Performance of kernels with segments in registers and shared memory usage.

number of levels. At the same time these numbers are constant. Hence, by loading them once into the shared memory of each block, access should be faster. Figure 10 shows that this is in fact the case. Table 2 shows the performance of the new kernels with segments in registers and shared memory usage.

Moving the segments to the registers and using shared memory increases clearly register usage, which influences occupancy. But the reduced usage of local memory and improved cache hit percentage clearly led to improved running times. Also the bandwidth usage went down to below 10 Gbyte per second for most kernels. This indicates that memory bandwidth is not a major issue. However, the used memory layout might have bad access patterns which in turn will not use the whole bandwidth. On the other hand the low memory bandwidth could indicate that the algorithm is now compute bound, meaning that the number of instructions related to computing are limiting the kernel speed. In addition the divergence in code flow has increased quite a bit for the 3-opt kernels.

Figure 10 shows speedup relative to The Benchmark Version. In the following we use the best current implementation as reference for speedup computations. This means that in the next one we compare to the implementation using segments in registers and shared memory.

6.4 Avoiding expensive arithmetic operations

During the aggregation of segments the algorithm needs to traverse the pre-computed hierarchy. Computing the index of a segment in the hierarchy includes several modulo operations and integer divisions, which are quite expensive operations on the GPU. However, if the divisor is a power of two, those operations can be replaced by bitwise operations which are much faster. By restricting the part lengths in the hierarchy to powers of two, we can replace most of the costly integer operations by bitwise operations. The resulting speedup, as shown in Figure 11, has to be examined more closely. If we compute the number of nodes in a part such that it is close to optimal, using only parts with power of two number of nodes will lead to different parts than before. We therefore study the speedup once for parts with the

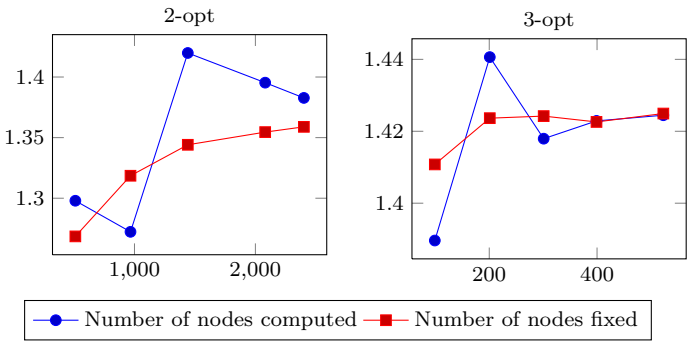


Figure 11: Speedup due to hierarchy containing only parts with power of two number of nodes. The x-axis shows the number of nodes in the current solution, the y-axis the speedup.

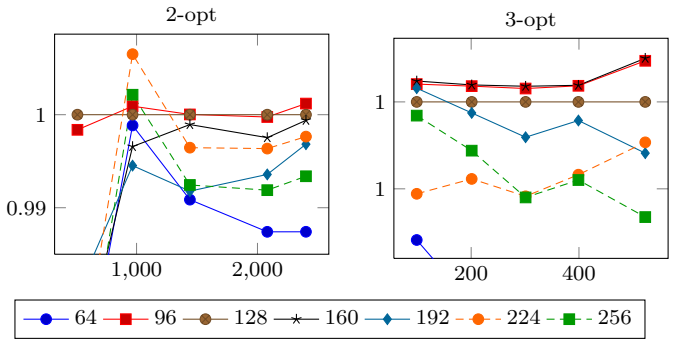


Figure 12: Speedup due to block size for cost objective. The x-axis shows the number of nodes in the current solution, the y-axis the speedup.

size computed and once for parts where the size is fixed to 8 nodes per part. Nevertheless, in both cases we can observe a significant speedup.

6.5 Block size

Before continuing the discussion on the implementation of kernels, let us focus on block size. So far each kernel was executed with blocks of 128 threads. However, due to different register usage as well as memory access and caching patterns, different kernels might perform better with different block sizes. In Figure 12, the effect of different block sizes for the cost objective in the 2-opt and 3-opt kernel is illustrated. We can see that for the 2-opt kernel using 96 or 128 threads per block is good, whereas for the 3-opt kernel we should either use 96 or 160 threads per block. We do the same examination for the other six kernels. For all kernel changes reported below, we also adjust block size.

6.6 Choice of data structures

In The Benchmark Version we use a simple array of structures (AoS) to store a hierarchy. This means that each segment is contained in a structure which is simply put in an array. The array itself is a jagged three-dimensional array

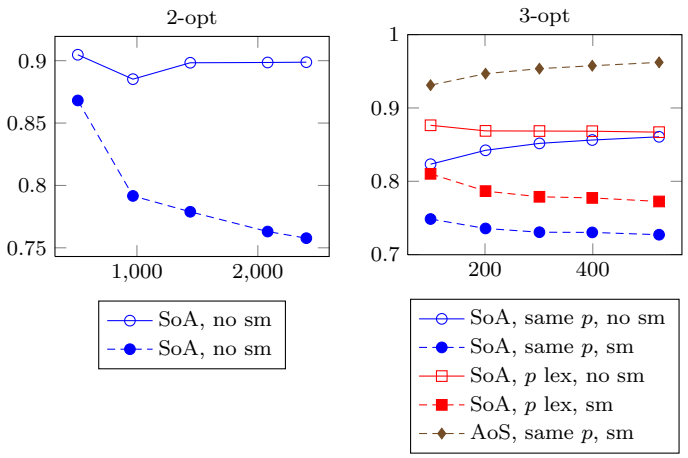


Figure 13: Speedup due to SoA/AoS and order related to the p component of the 3-opt neighbor index. The x-axis shows the number of nodes in the current solution, the y-axis the speedup. The acronym sm means shared memory, lex means lexicographically.

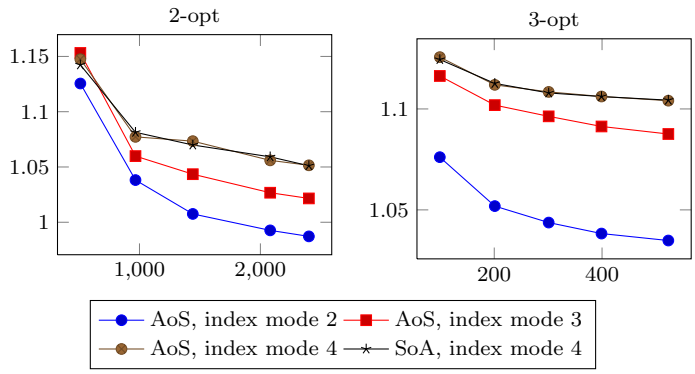


Figure 14: Speedup due to index mode. The x-axis shows the number of nodes in the current solution, the y-axis the speedup.

with constant width and height. This gives opportunities for improving the memory access pattern of our kernels. Firstly, it is often recommended in the GPU literature to use a structure of arrays (SoA) instead. This has the advantage that the same elements of adjacent segments lie next to each other in memory.

Another option is to modify the mapping between the one dimensional thread index and the corresponding neighbor index to improve the memory access pattern to the hierarchy. Or, instead of modifying the mapping, we can modify the layout of the segments in the hierarchy to change the memory access pattern. First we will study the use of AoS vs SoA. In combination we will modify the mapping to the 3-opt neighbor index with respect to the p component. Instead of having the indices ordered lexicographically with respect to p, meaning $(x,\ y,\ z-1,\ 3)$ $(x,\ y,\ z,\ 0)$, $(x,\ y,\ z,\ 1)$ $(x,\ y,\ z,\ 2)$ $(x,\ y,\ z,\ 3)$ $(x,\ y,\ z+1,\ 0)$, we first use all neighbors with $p = 0$, then all neighbors with $p = 1$, and so on.

The resulting speedups are shown in Figure 13. We can observe that using AoS with lexicographically ordered p as in The Benchmark Version is best, as all other variants are slower. We would nevertheless like to point out that for the SoA layout, the kernels not using shared memory perform better. One possible explanation is that the numbers of nodes per part are already in the cache most of the time, so uploading them to shared memory is only additional work.

6.7 Index mode

In The Benchmark Version we used a quite complicated indexing of the hierarchy, taking into account whether the segment is in the middle of a hierarchy part or starts or ends at a node separating two parts. The goal for this was to maximize the probability for adjacent threads to access adjacent segments.

The fact that AoS is better than SoA indicates that this is actually not so important for the performance of the kernels. Hence we can use simpler and faster indexing modes for the hierarchy. One possibility is to generate an array of segments for each node. This array contains the segments which start at this node and end at a later one. For each node we add another array containing the segments that end at this node but start at a later one, basically the inverted segments of the first array. Altogether this leads to a three dimensional jagged array, where the first dimension has only two elements, not-inverted and inverted. The second dimension always contains $n-1$ elements, where n is the number of nodes in the current solution. We call this indexing scheme index mode two.

A different approach is to merge both arrays per node into one, having a non-inverted segment being followed by its inverted one. This leads to a two dimensional jagged array and is our index mode three. We implement jagged arrays by having additional information about the starting positions for the different arrays in global memory. Accessing an element in a jagged array therefore needs two global memory access, one for reading the starting position of the corresponding array and a second for reading the element in that array.

An alternative to index mode three with a jagged array is therefore to simply use a two dimensional array where the non-used entries are empty. This will waste some global memory, but saves one additional memory access. This is index mode four.

The results of using the different indexing modes are shown in Figure 14. We can see that index mode two is better than the previous indexing scheme for 2-opt neighborhoods for solutions with up to $\approx$1750 nodes, but above that it is worse. It is better for the 3-opt neighborhood for all tested instances. Indexing mode three is better than two and in all cases better than the previous scheme, and, as is to be expected, index mode four is better than index mode three. An interesting fact is that using SoA with shared memory and index mode four is equally good as AoS with

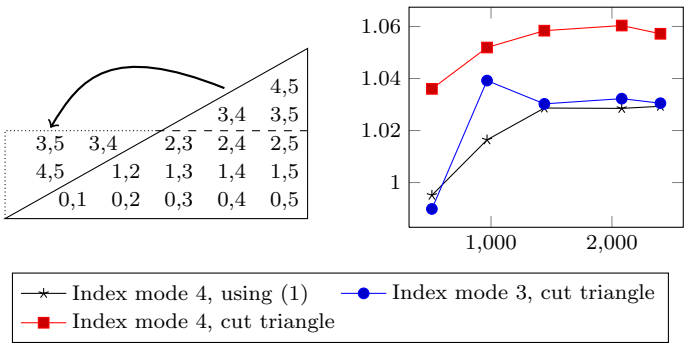


Figure 15: *Left:* 2-opt thread - neighbor index mapping using the cut triangle idea. *Right:* Speedup due to cut triangle mapping. The x-axis shows the number of nodes in the current solution, the y-axis the speedup.

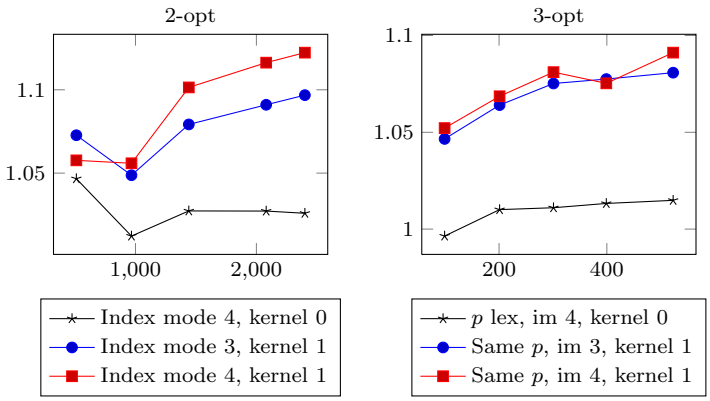


Figure 16: Speedup due to kernel mode. The x-axis shows the number of nodes in the current solution, the y-axis the speedup. The acronym im is used for index mode, lex for lexicographically. Kernel mode zero is the one where all segments are computed and stored, kernel mode one computes only the next one needed.

these settings.

It is not easy to decide whether to use index mode three or four. Index mode four is faster, but wastes global memory, which could become a problem for large solutions. We therefore chose index mode three as our reference implementation, but will in following comparisons also consider index mode four.

6.8 2-opt mapping

The current mapping between the thread index and the 2-opt neighbor index orders the neighbors lexicographically. In this way adjacent neighbors can often use the same first segment. However, the mapping itself involves a square root and a rounding operation. A different type of mapping would be to simply consider the 2-opt neighbor indices to be laid out in a triangle, as illustrated in the left part of Figure 15. By cutting the top of the triangle and moving it around one can create a rectangular mapping which is easy to compute. The disadvantage is that inside the rectangle the adjacent neighbors along the line separating base and top of the triangle have less correlation. Nevertheless, we can see in the right part of Figure 15 that this simplified mapping yields a speedup over the mapping given by formula (1).

6.9 Kernel mode

At this point, the current solution is split into three or four segments and all of those are kept in registers. This was done so that the algorithm steps through the solution in a linear way and thus linearly through the corresponding hierarchy segments. This is to ensure that adjacent threads access the same or adjacent segments in the hierarchy. As we learned above, however, such proximity is not always beneficial. We therefore investigate an implementation where each segment is computed at the time it is needed. In this way we only need to store two segments in the registers, the first representing the new computed solution so far, and the second the next needed segment of the current solution. This reduces register usage and could thus lead to higher occupancy and better performance. That this is in fact the case is shown in Figure 16. We notice that for the new way of computing the segments, changing the order related to p in the mapping for the 3-opt indices back to the one where neighbors with the same p component are next to each other yields the best results.

6.10 Number of hierarchy levels

The improvements in the kernels so far were done using a one-level hierarchy. In general using a l-level hierarchy to compute one solution segment leads to at most $2l-1$ combinations of hierarchy segments. On the other hand, creating a l-level hierarchy is faster with increasing l. Depending on the situation, a low or a high l should be beneficial. We therefore study our implementation with respect to the previous mentioned aspects also for a zero-level and a two-level hierarchy. A zero-level hierarchy is basically just the collection of all possible segments in the current solution. The results are displayed in Figure 17.

For all hierarchies the AoS layout performs best as does the cut triangular mapping for 2-opt neighbor indices and the zero-level hierarchies do not need the shared memory. We observe that for a 2-opt neighborhood using a one-level hierarchy actually performs best. For a 3-opt neighborhood instead the zero-level hierarchy is best. This makes sense, as the the 3-opt neighborhood has cardinality $O(n^3)$, whereas creating the zero-level hierarchy is $O(n^2)$. In addition, only having a collection of segments removes the need of traversing a hierarchy.

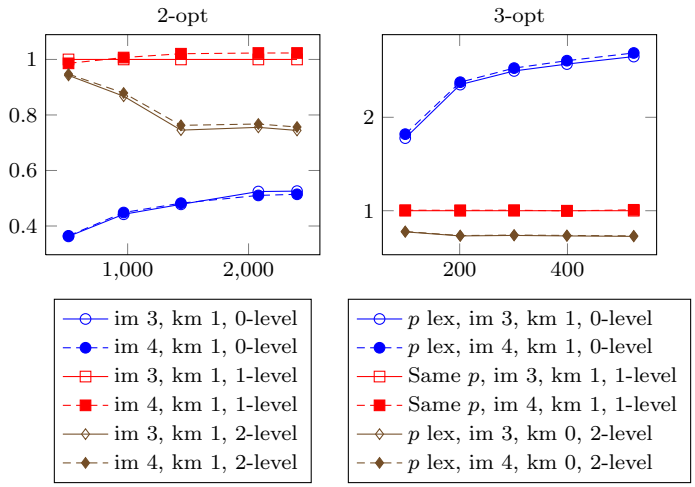


Figure 17: Speedup due to number of levels. The x-axis shows the number of nodes in the current solution, the y-axis the speedup. Used acronyms are im for index mode, km for kernel mode, and lex for lexicographically.

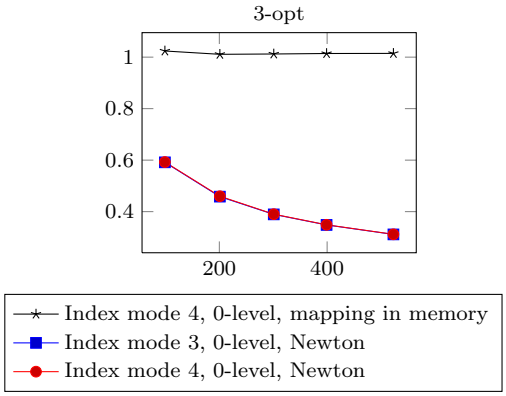


Figure 18: Speedup due to Newton based evaluation of 3-opt mapping. The x-axis shows the number of nodes in the current solution, the y-axis the speedup.

6.11 Newton based evaluation of 3-opt mapping

Before we discuss a final change to the implementation, we would like to shortly mention why we pre-compute the mapping between thread index and x, y and z components of the 3-opt neighbor index on the CPU and copy it to the GPU. As this part of the mapping is a lexicographical order, it is very easy to compute sequentially. On the GPU side, reading from global memory is not very fast, but it is one simple operation. In [18] it is suggested to use Newton's method to solve a cubic equation in order to compute the mapping during the neighborhood evaluation. This approach is elegant as it reduces the need for explicitly storing the mapping. Unfortunately, it is also slower, as demonstrated in Figure 18.

Kernel		Time (ms)	Occ	Reg	Bw (Gb/s)	L1 (%)	Lm	Div
3-opt	ζ_c	31	1.00	18	12	94	0	17e3
	z_w	26	1.00	20	42	77	0	2
	z_K	27	1.00	18	35	91	0	0
	ζ_L	76	0.52	28	12	83	0	52e4
2-opt	ζ_c	0.07	1.00	18	35	83	0	31
	z_w	0.05	0.83	21	61	72	0	0
	z_K	0.06	1.00	18	44	84	0	0
	ζ_L	0.16	0.67	30	41	83	0	768

Table 3: Performance of kernels for zero-level hierarchy applied to current solution with 399 nodes.

Kernel		Time (ms)	Occ	Reg	Bw (Gb/s)	L1 (%)	Lm	Div
2-opt	ζ_c	0.18	0.67	27	<1	95	0	865
	z_w	0.14	0.67	29	14	84	0	884
	z_K	0.16	0.67	27	8	94	0	820
	ζ_L	0.24	0.38	39	4	90	0	1271

Table 4: Performance of 2-opt kernels for one-level hierarchy applied to current solution with 399 nodes.

6.12 Combined speedup so far

The process of tuning the kernels has yielded substantial speedups. The resulting performance of the kernels is shown in Table 3 for a zero-level hierarchy (3-opt neighborhood) and in Table 4 for the 2-opt kernels for a one-level hierarchy.

The 2-opt kernels also perform better for a zero-level hierarchy than a one-level hierarchy. The hierarchy creation process plays a more important role for 2-opt than for 3-opt. Therefore, in a 2-opt neighborhood, the overall performance is best with a one-level hierarchy. Nevertheless, we can observe that the kernels perform much better than The Benchmark Version in Table 1 for both cases. All kernels use less registers than before. In fact, some of them use less than 20 registers enabling an occupancy of 1. This leads to the idea that it might now be worth to combine the evaluation of constraints / objectives in one kernel. In this way the computation of indices, mapping computations or memory reads can be shared and only need to be done once. This could lead to a synergy effect.

6.13 Combined evaluation

As a last kernel improvement study we examine the possibility of evaluating two of the objectives or constraints together. As the results in Figure 19 show, this leads to an improvement in most cases. It shows that combining the cost objective with the route distance constraint and the capacity constraint with the number of vehicles objective yields the most improvement[11].

[11] We also tried the other combinations and combining three objectives/constraints together, but this led to the CUDA compiler crashing.

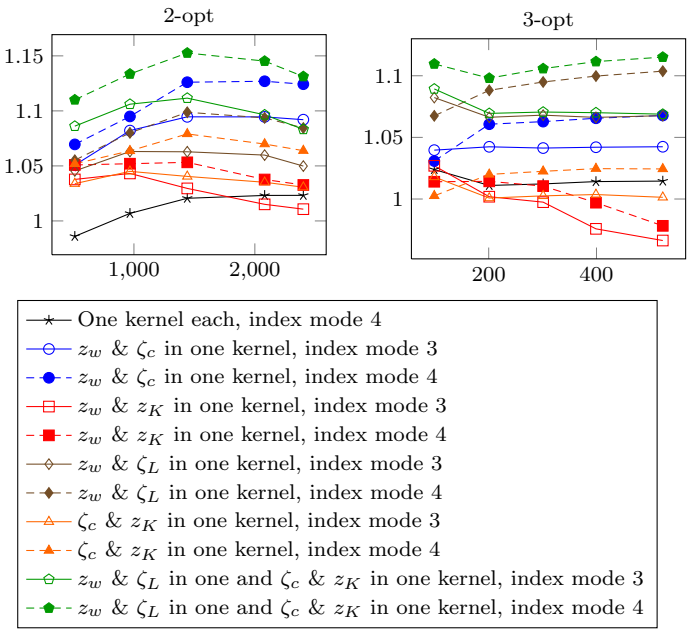


Figure 19: Speedup due combined evaluation of several objectives/constraints in one kernel. The x-axis shows the number of nodes in the current solution, the y-axis the speedup.

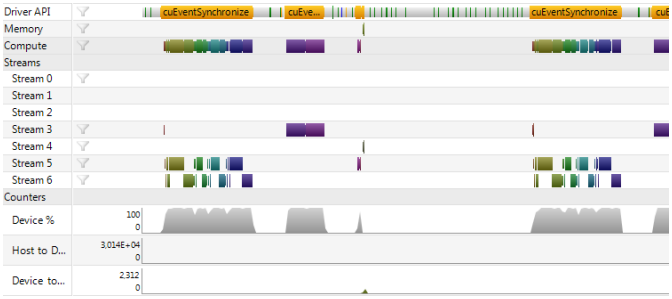


Figure 20: Timeline of one iteration in a 2-opt neighborhood for improved kernels and 399 nodes in solution.

6.14 Improving GPU-CPU coordination

In the previous subsections we described how we improved the performance of the neighborhood evaluation kernels. This is the bottleneck for both the 3-opt neighborhood and the 2-opt neighborhood for solutions of reasonable size. However, in Figure 7 we saw that the GPU is idle about 40% of the time for a 2-opt neighborhood for a solution with 399 nodes. This utilization has worsened with the improved kernel performance, as shown in Figure 20. The GPU is now around 60% idle.

The first row in a timeline plot shows the activity of the driver API by gray or colored bars. A white background without any bars means the CPU thread is busy with computations that are not related to the driver API. When the CPU is synchronizing with the GPU and thus waiting for it, the driver API will perform a synchronization task. Those tasks are marked with yellow bars in the driver API row of the timeplot. As we can see in Figure 20, our implementation uses a simple synchronized GPU and CPU program flow. This basically means that either the CPU or the GPU is busy. However, when the GPU is creating the hierarchy or evaluating the neighborhood, the CPU does not need to wait. In fact, the first four tasks in our algorithm, as described in Figure 6, can be performed in sequence without the CPU intervening. Thus the work of the driver API that is necessary to schedule those kernels can be done while the GPU is working on the previous tasks.

The most significant idle time for the GPU appears after copying the neighbor to the CPU. This is not due to the move execution on the CPU, but to kernel scheduling for the hierarchy creation. When creating a hierarchy, several kernels calls are necessary, and all of them need to be scheduled. This scheduling is clearly marked in the driver API row after copying the move from the GPU to the CPU.

To better make use of the neighborhood evaluation time we therefore make use of the following scheme. We schedule the hierarchy generation directly after setting up the copying of the move. This means that the CPU will not wait until the best neighbor is computed, and the move is executed and copied, before scheduling the hierarchy creation. This leads to the following setup. Let us assume that the hierarchy is already computed for the current iteration. Then the CPU schedules the neighborhood evaluation, reduction to best neighbor, move execution on GPU, copy of move to CPU. It then sets an event for the copy to be finished and can thus schedule to create the hierarchy for the new solution on the GPU right away. It then waits for the event signalizing that the copied move is available. Using streams, events, and stream synchronization towards events we can also synchronize the tasks on the GPU with each other without the need of CPU intervention.

With this approach, scheduling all kernels related to the hierarchy creation can in theory be performed while the GPU is busy with the other tasks. In Figure 21 we see that the asynchronous execution is able to reduce the GPU idle time, but unfortunately is not able to remove it for a solution with 399 nodes. There are just too many kernels calls needed to create the one-level hierarchy for all objectives and constraints. For a solution with around 1000 nodes however, the kernel scheduling can be hidden completely, leading to the GPU being busy practically the whole time. This is shown in Figure 22.

In Figures 21 and 22 the timelines do not contain a device to host copy row in the counters section. This has the following reason. For the programmer it is not possible in CUDA to specify when an asynchronous copy shall happen and whether it is overlapping with kernel execution or not. Using an asynchronous copy operation for transferring the best move from the GPU to the CPU led unfortunately to unwanted and unnecessary idle time for the GPU instead of overlapping copy operations and kernel execution. We therefore use a kernel with host mapped memory to copy the move. Host mapped memory means that the global GPU memory the kernel writes to is automatically, implic-

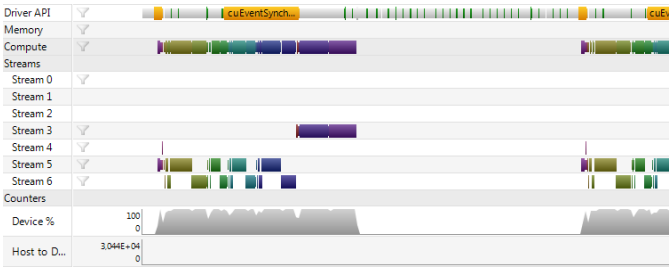


Figure 21: Timeline of one iteration in a 2-opt neighborhood for improved kernels, asynchronous execution and 399 nodes in solution.

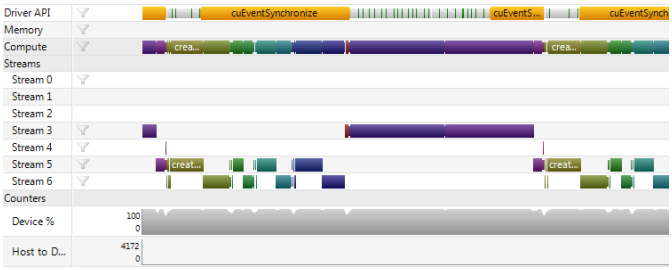


Figure 22: Timeline of one iteration in a 2-opt neighborhood for improved kernels, asynchronous execution and 967 nodes in solution.

itly transferred to the host and accessible there. It therefore does not show up in the device to host copy counter row in the timeline plot.

Obviously it would be good to reduce the number of kernel calls for the hierarchy creation. However, CUDA does not provide a GPU side synchronization between blocks. This is why we have one kernel call per level as the efficient hierarchy creation relies on using level l to compute the segments of level $l+1$. One can of course implement a creation kernel that does not rely on such re-use. We did try that, but this leads to basically the same as creating all possible segments for one solution. This is the same as using a zero-level hierarchy, which we already know is slower than the current implementation for a one-level hierarchy.

As a conclusion of this problem, *we observe that for the GPU to be fully used all the time, the neighborhood that is evaluated needs to be large enough to hide the scheduling of all kernels.* This is the case when the solution contains more than 900 nodes for a 2-opt neighborhood and more than 110 nodes for a 3-opt neighborhood.

6.15 Splitting large neighborhoods

So far we have assumed that we can store the whole neighborhood fitness structure in the global memory of the GPU. This requires either enough memory or a limited size neighborhood. We would like our algorithm to be able to perform well for any reasonable neighborhood and memory size.

The simple idea here is to split the neighborhood into parts, such that the fitness structure of each part fits into approximately half of the memory. We compute the first

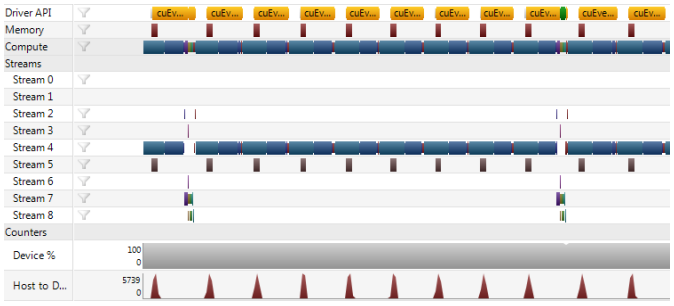


Figure 23: Timeline of one iteration in a 3-opt neighborhood and 735 nodes in solution.

part and store the fitness structure in the first half of the memory. After that we compute the second part and store its fitness structure in the second half of the memory. Then for each stored neighbor in the first fitness structure we compare it with the corresponding neighbor in the second fitness structure and the best feasible of the two is stored back into the first half of the memory. In this way the first part of the global memory contains a subset of the evaluated neighborhood which is guaranteed to contain the best feasible neighbor so far. For the remaining parts we do the same, they are evaluated and the resulting fitness structures combined with the previously computed subset. Thus after all parts of the neighborhood are evaluated, we have a subset of the neighborhood fitness structure that contains the best feasible neighbor. We perform a reduction of this set to get the best feasible neighbor.

Of course one could reduce the fitness structures of each neighborhood part to the best feasible neighbor and thus only keep the best feasible in memory. However, a goal is to provide an as large subset of the neighborhood fitness structure as possible at the end of the neighborhood evaluation.

This approach works fine for a 2-opt neighborhood. For a 3-opt neighborhood we so far used a mapping in memory that was constant. With large neighborhoods we do not want to or may not be able to store the whole mapping in global memory. Instead we utilize the fact that the GPU is able to copy data from the CPU while running kernels.

Thus we split the mapping into mappings for each part. While part i is evaluated, we copy the mapping for part $i+1$ from the CPU to the GPU, hiding the whole copy operation. The copying of the mapping for one part takes less time than the neighborhood evaluation of a part. Hence, the whole copying is completely hidden with respect to runtime and does not slow down the algorithm. This can be nicely seen in Figure 23.

Unfortunately, one is not able to explicitly tell the GPU to overlap a copy operation with a kernel call. The driver API uses some heuristic or other method to determine overlap or not. In the first iteration of the local search, not every copy operation is overlapped. However, after one local search iteration, the driver API is able to overlap all following mapping copy operations with kernel execution. During the evaluation of the final neighborhood part in an iteration

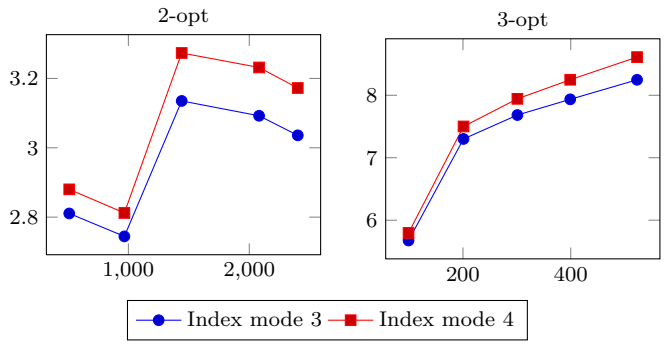


Figure 24: Overall combined speedup. The x-axis shows the number of nodes in the current solution, the y-axis the speedup.

we already copy the mapping for the first part of the mapping for the next iteration. In this way, there is no need to wait for any copy operation between iterations either.

6.16 Final results

In the previous subsections we analyzed the speedup in an incremental fashion. Figure 24 shows the combined speedup of the best implementation compared to The Benchmark Version. The time it takes to perform one whole iteration in the local search process using the best implementation is shown for different solutions in Table 5.

Due to the use of aggregation, one potential move can be evaluated in constant time. This can be observed in the times in Table 5 with taking into account that for small neighborhoods the other parts of the local search process play a more important role. We can observe that the ratio time/(neighborhood size) first decreases as the neighborhood evaluation part becomes the dominant factor of the local search process and then stays roughly constant. This factor also provides us with some interesting comparison of the GPU version to the sequential world. If the whole 2-opt neighborhood of a solution with more than 200 nodes should be evaluated with the same speed in a sequential version, we would in average have less than 36 ns to evaluate each potential move. For a 3-opt neighborhood the time is even less than 5 ns in average per potential move, which corresponds to 10 cycles for a 2 GHz core.

7 Filtering Neighbors

Using the GPU we are able to evaluate the whole 3-opt neighborhood for a solution with 2401 nodes in less than 40 seconds. The size of this neighborhood is $9 \cdot 10^9$. For such huge neighborhoods the question arises whether it makes sense to evaluate the whole neighborhood. On the CPU, large neighborhoods are impossible to evaluate simply due to the time it would take. On the GPU, the problem is basically the same, but the critical size will be different.

For large neighborhoods, the evaluation on the GPU is split into parts which are executed in sequential order. It is therefore easy to change from the *Best Improving* strategy to a *First Improving* strategy. After a given part has been evaluated, it is completely reduced to its best feasible neighbor. If this neighbor is improving, we select the corresponding move and ignore the remaining parts of the neighborhood. Unfortunately, this will destroy some of the asynchronous nature of the LS procedure. However, since each part is large in itself, the evaluation time should still be the dominating runtime factor.

Filtering is a popular approach in the DOP community for handling very large neighborhoods. By an efficient procedure[12] one eliminates a large number of neighbors. Filtering works very nicely in a sequential setting, but does it make sense on the GPU? The goal of the remaining part of this section is not to identify the best implementation of filtering on the GPU, but to study the effects and illustrate problems. To this end we use a simple model of filtering, and neighborhoods that do not need splitting.

For our experiments we use a boolean *filter vector* where each element says whether the corresponding neighbor shall be evaluated or ignored. In reality, this vector will be generated by the filtering method. For our investigation, we may choose the elements to be filtered randomly, as filtering procedures will utilize some structural characteristic of the DOP rather than memory structures on the GPU. Therefore, we generate random filter vectors with a controlled filtering factor.

The first naive way of implementing filtering is to check whether the corresponding potential move shall be evaluated or not inside each thread. However, as explained in Section 5.2, a thread is not executed alone but as part of a warp. Each thread inside a warp does exactly the same instructions, with divergence realized by masking. Even if only the potential move of one thread inside a warp needs to be evaluated, the execution time of the whole warp is determined by this thread. Only if all potential moves related to the threads of a warp are filtered, the whole warp can drop the evaluation process, leading to a speedup.

This effect is clearly noticeable in the results shown in Figure 25. Even filtering out as much as 75% of all neighbors does not lead to a significant speedup of computation time. Only when we apply a stronger filter, keeping only 1% or even 0.1% of the neighbors, a significant speedup is noticeable. This is understandable, as a warp contains 32 threads. Hence, if evaluating more than 1/32 of all neighbors, chances are high for a warp to contain at least one thread which has to evaluate its potential move. Evaluating only 1% or less neighbors gives high probability that whole warps do not need to evaluate anything, thus leading to speedups.

A better implementation is a *mapping vector* that contains identifiers of the neighbors that need to be evaluated. In this way we can use consecutive threads to evaluate those neighbors. The effect is either fewer threads in the evaluation kernel call, or whole warps having no neighbors to

[12] Filtering procedures may be exact or heuristic.

		57	99	201	301	399	511	967	1441	2081	2401
2-opt	index mode 3	5.03e−04	6.53e−04	7.11e−04	1.04e−03	1.47e−03	2.07e−03	5.63e−03	1.01e−02	2.04e−02	2.71e−02
	index mode 4	4.99e−04	6.47e−04	7.03e−04	1.02e−03	1.45e−03	2.02e−03	5.50e−03	9.68e−03	1.95e−02	2.59e−02
3-opt	index mode 3	1.25e−03	3.72e−03	2.40e−02	7.75e−02	1.77e−01	3.56e−01	2.57e+00	8.29e+00	2.49e+01	3.83e+01
	index mode 4	1.23e−03	3.64e−03	2.34e−02	7.50e−02	1.70e−01	3.41e−01	2.45e+00	7.88e+00	2.36e+01	3.63e+01

Table 5: Times per iteration in seconds for different solution sizes and both 2-opt and 3-opt neighborhoods. The header row shows the number of nodes in the solution.

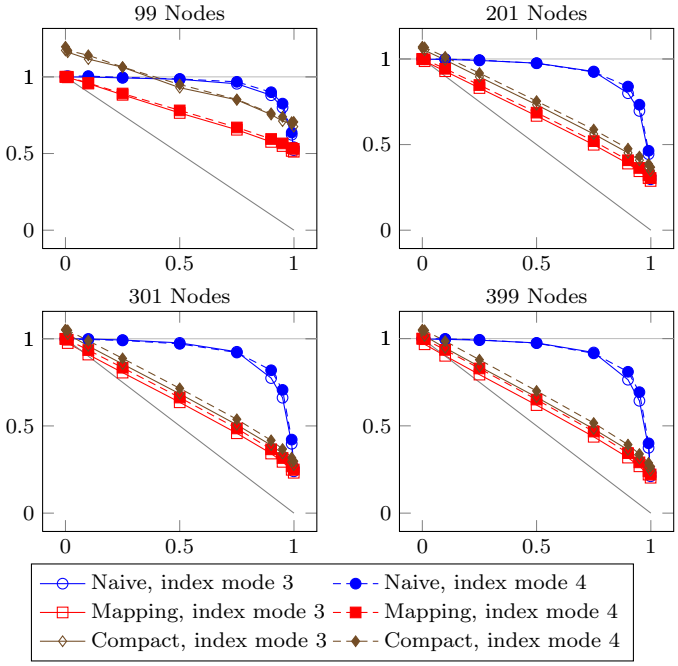


Figure 25: Simulating filtering for solutions with different number of nodes. The x-axis shows the percentage of filtered neighbors, the y-axis the percentage of time related to no filtering.

evaluate. This should lead to a speedup dependent on the percentage of neighbors filtered out. This is in fact the case, as can be seen in Figure 25. However, the stronger the filter, the less two adjacent neighbors in the mapping vector have in common. Therefore we will get less good memory access patterns and probably also more divergence in the code flow than when evaluating the whole neighborhood. These effects explain why filtering out 75% of the potential moves does not lead to a 75% speedup.

In a real setting it is unlikely that a mapping vector is given. Instead we need to use information from the filter vector to create the mapping vector. The operation in question is well known as a compact operation in the GPU community. We use the compact algorithm from the Thrust library [11] in our experiments. Compaction has to be performed in each iteration as the filter vector might change. In Figure 25 one can observe that the computational cost for the compact operation is significant for small neighborhoods, whereas it becomes less significant as neighborhood size grows.

8 Conclusion

In this paper we investigated different GPU implementations for local search. Starting from The Benchmark Version we studied the effects of changing different parts of the implementation, leading to an incremental improvement of our program. We focused both on having efficient evaluation kernels and a well designed CPU-GPU interaction. In addition we showed how memory limitations can be overcome such that the implementation can also efficiently evaluate very large neighborhoods whose fitness structures do not fit into GPU memory. At the end of this paper we shortly touched the topic of filtering moves and the problems related to it.

The presented experiments show clearly that carefully adjusting the GPU implementation of local search yields a significant payoff. We have observed almost an order of magnitude speedup. Arguably, the most important and general lesson learnt is the importance of keeping the GPU busy at all time. It does not help to have a very efficient kernel if the GPU is only used 30% of the time. A busy GPU also includes a well designed coordination strategy between the CPU and the GPU. Tasks that seem to be inherently sequential, such as executing the move and then evaluating the new hierarchies, can be performed at different times on

the GPU and GPU, respectively, allowing one of them to 'race ahead'.

For large neighborhoods the evaluation of potential moves dominates the runtime, whether performed on CPU or GPU. Thus it makes sense to not just optimize the GPU-CPU coordination, but also to carefully optimize the GPU implementation of the evaluation itself. The type of neighborhood and the way it is represented is very important to efficiency. Adjacent threads in a warp always perform the same operation. Their memory access patterns influence the time needed for accessing data. Hence, the mapping between neighbors and threads has a strong effect on GPU utilization. Adjacent neighbors should virtually always be processed in the same way. The data needed should be either identical or adjacent in memory.

Tuning the algorithm and its parameters is also important to performance. In our investigations it turned out to be better to use a zero-level hierarchy for the 3-opt operator, in contrast with a one-level hierarchy for 2-opt. Evaluating a segment when it is needed turned out to be better than computing all at once.

Of course there are many more aspects that influence efficiency, for instance the type of memory or instructions used. There is wide range of possibilities that can be explored in order to find a good, if not the optimal GPU implementation. The challenge is to identify the most promising alternatives related to the current implementation, as one seldom has the time to try all. In this process, performance tools proved to be very helpful. However, one also needs experience to be able to identify bottlenecks and possible remedies for a given implementation.

The main goal of this paper was to develop an efficient GPU implementation of local search. Through a careful improvement process we managed to arrive at an implementation where the GPU is busy basically the whole time, given a not too small problem. However, the CPU is almost idle.

CPU's of today have several cores and can execute multiple threads simultaneously. In the broader picture of developing efficient discrete optimization methods on modern architectures, we need to learn how to distribute the work load between GPU and CPU so that both units are busy all the time. Ideally, methods for computationally hard problems should be able to self-adapt to the available hardware for maximum system performance. Such heterogeneous methods will become more and more important in the future, as there will still be significant improvements in processor hardware. Producers are starting to combine the GPU and CPU into one chip, drastically reducing costs for transferring data between them. In some architectures they actually use the same memory, so data transfer is no issue at all.

In order to profit from current and future hardware development, we need to understand how to distribute the work between heterogeneous computing units in a self-adaptable way. The demands from industry and the scientific communities for computing power for discrete optimization problems are virtually without limits.

Acknowledgement

The author would like to thank Christopher Dyken at SINTEF for providing access to his tikz-generated pictures illustrating CUDA.

References

[1] Emile Aarts and Jan Karel Lenstra, editors. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003.

[2] Roberto Baldacci, Nicos Christofides, and Aristide Mingozzi. An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. *Mathematical Programming*, 115:351–385, 2008. 10.1007/s10107-007-0178-5.

[3] R. E. Bixby. Solving real-world linear programs: A decade and more of progress. *Oper. Res.*, 50(1):3–15, 2002.

[4] A. R. Brodtkorb. *Scientific Computing on Heterogeneous Architectures*. Ph.D. thesis, 1501-7710, No. 1031. University of Oslo, 2010.

[5] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[6] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Manag. Sci.*, 6(80), 1959.

[7] M. Gendreau and J.Y. Potvin. *Handbook of Metaheuristics*. International Series in Operations Research & Management Science. Springer, 2010.

[8] Bruce Golden, S. Raghavan, and Edward Wasil, editors. *The Vehicle Routing Problem – Latest Advances and New Challenges*. SIAM Monographs on Discrete Mathematics and Applications. 2002.

[9] Mark Harris. *Optimizing Parallel Reduction in CUDA*. NVIDIA, 2007.

[10] G. Hasle and O. Kloster. Industrial vehicle routing problems. *Chapter in Hasle G., K-A Lie, E. Quak (eds): Geometric Modelling, Numerical Simulation, and Optimization. ISBN 978-3-540-68782-5, Springer*, pages 397–436, 2007.

[11] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[12] Stefan. Irnich. Online supplement to: A unified modeling and solution framework for vehicle routing and local search-based metaheuristics. *INFORMS Journal on Computing*, 20, 2008. URL: http://www.informs.org/Pubs/IJOC/Online-Supplements/Volume-20-2008.

[13] Stefan Irnich. Resource extension functions: properties, inversion, and generalization to segments. *OR Spectrum*, 30:113–148, 2008.

[14] Stefan. Irnich. A unified modeling and solution framework for vehicle routing and local search-based metaheuristics. *INFORMS Journal on Computing*, 20:270–287, April 2008.

[15] A. Janiak, W. Janiak, and M. Lichtenstein. Tabu search on GPU. *Journal of Universal Computer Science*, 14(14):2416–2427, 2008.

[16] Yifang Liu. *Algorithms for VLSI circuit optimization and GPU-based parallelization*. PhD thesis, Texas A&M University, 2010.

[17] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. GPU-based island model for evolutionary algorithms. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 1089–1096, New York, NY, USA, 2010. ACM.

[18] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Neighborhood structures for gpu-based local search algorithms. *Parallel Processing Letters*, 20(4):307–324, 2010.

[19] NVIDIA. *NVIDIA Compute Visual Profiler*, 2010. Version 3.2.

[20] NVIDIA. *Tuning CUDA Applications for Fermi*, August 2010. Version 1.3.

[21] E.G. Talbi. *Metaheuristics: from design to implementation*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, 2009.

[22] P. Toth and D. Vigo, editors. *The Vehicle Routing Problem*. SIAM Monographs on Discrete Mathematics and Applications. 2002.

[23] Pablo Vidal and Enrique Alba. Cellular genetic algorithm on graphic processing units. In Juan Gonzlez, David Pelta, Carlos Cruz, Germn Terrazas, and Natalio Krasnogor, editors, *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, volume 284 of *Studies in Computational Intelligence*, pages 223–232. Springer Berlin / Heidelberg, 2010.

Technology for a better society

www.sintef.no