Adaptive Large Neighborhood Search on the Graphics Processing Unit

Lukas Bach, Geir Hasle, Christian Schulz

Citation:
Lukas Bach, Geir Hasle, Christian Schulz. (2019) Adaptive Large Neighborhood Search on the Graphics Processing Unit. European Journal of Operational Research, 2019, 275 (1) 53-66.
DOI: 10.1016/j.ejor.2018.11.035

Adaptive Large Neighborhood Search on the Graphics Processing Unit

Lukas Bach*, Geir Hasle, Christian Schulz

Department of Mathematics and Cybernetics, SINTEF Digital, Oslo, Norway

Abstract

For computationally hard discrete optimization problems, we rely on increasing computing power to reduce the solution time. In recent years the computational capacity of the Graphics Processing Unit (GPU) in ordinary desktop computers has increased significantly compared to the Central Processing Unit (CPU). It is interesting to explore how this alternative source of computing power can be utilized. Most investigations of GPU-based methods in discrete optimization use swarm intelligence or evolutionary methods. One of the best single-solution metaheuristics for discrete optimization is Adaptive Large Neighborhood Search (ALNS). GPU parallelization of ALNS has not been reported in the literature. We gain knowledge on the difficulties of developing a data parallel version of the ALNS, and investigate the efficiency of ALNS on the GPU. To this end, we develop an ALNS for the much studied Distance Constrained Capacitated Vehicle Routing Problem (DCVRP). We compare the performance of our GPU-based ALNS with a state-of-the-art CPU implementation using standard DCVRP benchmarks. While it proved hard to implement certain commonly used mechanisms efficiently on the GPU, experimental results show that our GPU-based ALNS yields highly competitive performance.

Keywords: Metaheuristics, ALNS, Discrete Optimization, VRP, GPU-programming

*Corresponding author.
Email address: lukas.bach@sintef.no (Lukas Bach)

1. Introduction and Literature Review

Many important tasks in society can be formulated as discrete optimization problems. The computational complexity of these problems is normally high, and real-life instances are often of a size that challenges state-of-the-art solution methods.

The power of an optimization algorithm is influenced by both its algorithmic capabilities as well as by the computer hardware it runs on. As an illustration, Bixby (2002) points to the fact that the average time for solving a Linear Program has been reduced by a factor of one million over a period of ten years, of which he contributes three orders of magnitude to the improvement in algorithms and methods, and the same factor he associates with the improvements in computer hardware during this time. In order to develop the needed new powerful methods, we thus need a thorough understanding of the methods used, in our case a metaheuristic, and the computer hardware at hand.

Modern commodity computers have come a long way from the first microprocessors that emerged in the 1970s. Up to the year 2004, both the number of transistors per chip, as well as its frequency doubled roughly every two years. This led automatically to a free speedup for all sequential programs. However, in 2004 the frequency hit the physical limit of what a chip can withstand. In the following years the continuing increase in the number of transistors per chip thus led away from single core CPUs and towards processors with multiple cores, which, however, are running on a lower frequency than the processors in 2004. This means that modern optimization algorithms need to be able to take advantage of this parallelism, if they want to further benefit from recent and future hardware development.

In addition, there is alternative hardware readily available in modern commodity computers. The task of displaying the pixels of a screen on a display has been moved to dedicated hardware, the Graphics Processing Unit, as early as in the 1980s. By the parallel nature of displaying many pixels, GPUs offer a high level of parallelism, but in a different way than multi-core CPUs. From the year 2000 on, researchers started to explore whether this massive parallelism can be used for non-graphical computations. Since then GPUs have evolved from specialized graphics hardware towards general purpose, relatively easily programmable, massively parallel accelerators. This facilitated the use of GPUs for general purpose computation, for instance in optimization. See Brodtkorb et al. (2013) for an introduction to the GPU

and how to use it in optimization, and Schulz et al. (2013) for a survey of routing related GPU literature.

Several vital mechanisms in metaheuristics are not difficult to parallelize. In particular, it is straightforward to exploit opportunities for task parallelism that multi-core processors offer. There is a sizable literature on parallel metaheuristics, see for instance Alba et al. (2013). However, less work has been reported on utilization of the data parallelism offered by GPUs. In particular, little work has been reported on GPU parallelization of methods for the VRP. Schulz et al. (2013) only found three papers. Until recently, we have seen no paper that presents competitive experimental results regarding solution quality. However, Boschetti et al. (2017) present data parallel versions of important route relaxations used in solving the pricing problem in state-of-the-art column generation methods for the CVRP that solve instances up to 200 customer to optimality. In more detail, the authors design GPU versions of the well-known q-route (Christofides et al. (1981)) and ng-route (Baldacci et al. (2017)) relaxations. They compare the performance of their data parallel GPU versions to a serial version of these algorithms on a set of standard CVRP instances as well as two new instances of large size. The authors report speedup factors varying from 0.3 for a 34 customer instance to 34.1 for an 859 customer instance.

Against this background, we investigate how to implement Adaptive Large Neighborhood Search (ALNS), one of the most competitive and generic metaheuristics for discrete optimization problems, such that the GPU is utilized efficiently. As our target problem, we have chosen the Distance-constrained Capacitated VRP (DCVRP) (see Toth and Vigo (2014)), for which both sequential and task parallel ALNS implementations have been investigated extensively and shown competitive performance (Pisinger and Ropke (2007); Ropke (2009)). In addition to investigations on solution quality vs. time performance, and GPU utilization, our goal is to gain knowledge on the difficulties of adapting the various mechanisms of ALNS to data parallelism. To the best of our knowledge, our paper presents the first investigation of data parallelism for the ALNS metaheuristic. Our research bears quite some resemblance to the one of Boschetti et al. (2017) referred to above. We design GPU versions of a set of well-known and effective destroy and repair operators in ALNS, and compare the performance of the resulting GPU version of ALNS both to serial and task parallel ALNS versions on a set of DCVRP instances ranging from 50 to 4500 customers.

Several authors in academia claim that the DCVRP has been solved for

all practical purposes, as the most competitive approximation methods show a negligible error relative to the best known solution value on standard benchmarks up to 1200 customers. Hence, it is uninteresting to study larger instances. We disagree. In our practical experience, we observe that our industrial VRP solver Spider generates, on a daily basis, approximate solutions to real-life instances of last mile delivery that may contain up to 10.000 delivery points. Users are concerned with response time and solution quality. In our opinion it is both important and relevant to conduct scientific studies of VRP instances with more than 1200 customers, a view that we share with Kytöjoki et al. (2007) (page 2744):

> In practice there are, however, vehicle routing problems of a much larger size. For example, in the context of waste collection and mail and newspaper delivery, some problems can involve thousands of customer points, and sometimes even several tens of thousands customers.

The DCVRP is informally defined as follows. There is a homogeneous fleet (either limited or unlimited in size) of capacitated vehicles for serving a set of request orders, each with a given demand. The travel cost matrix (e.g. distance or time) between the points representing either request locations or the depot is given. The goal is to find a cost minimizing set of tours that visits all requests once. Each tour is performed by one vehicle, starts and finishes at the depot, adheres to a given route length (duration) constraint, and the total demand for the visited requests obeys the vehicle capacity constraint. For a more precise definition and mathematical formulations, we refer to Irnich et al. (2014). Many exact methods for the DCVRP have been proposed, including Branch & Cut and Branch & Cut & Price methods. Currently, state-of-the-art exact methods are able to solve any DCVRP instance to proven optimality with less than some 200 requests, in a few minutes. We refer to Part I of Toth and Vigo (2014) for a survey of exact methods for the DCVRP. For larger DCVRP instances, several metaheuristics have shown good performance. The best competitors find most of the known optimal solutions in a short time. Particularly interesting are metaheuristics that are also competitive across many VRP variants, for instance UHGS by Vidal et al. (2014), and the selected metaheuristic of our study: ALNS by Pisinger and Ropke (2007).

The remainder of this paper is organized as follows. In Section 2 we present the ALNS and its implementation on the CPU. Section 3 presents

GPU-ALNS, our implementation of ALNS on the GPU. For the interested reader we refer to Appendix A where we give insights into how well our algorithm utilizes the GPU. Results from computational experiments are presented in Section 4, and finally, in Section 5, we present our conclusions. Detailed results are given in Appendix B. In addition, we provide a general introduction to GPU programming with CUDA in the on-line appendices. There we also provide detailed information on how we tuned the algorithm on the GPU to achieve better performance.

2. Adaptive Large Neighborhood Search

Adaptive Large Neighborhood Search (ALNS) was introduced by Ropke and Pisinger (2006); Pisinger and Ropke (2007) as a further development of Large Neighborhood Search (LNS) proposed by Shaw (1998). In ALNS we have a set of neighborhoods, each with an assigned *score*. The score specifies the likelihood of selecting the neighborhood, e.g. by using roulette wheel selection. The scores are adjusted during the search process based on neighborhood performance. Whether to accept a new solution generated by the neighborhood is controlled by a metaheuristic, typically Simulated Annealing, see Pisinger and Ropke (2010).

Neighborhoods are defined by destroy and repair operators. Destroy operators remove a number of requests from the current solution, whereas repair operators reinsert the removed requests to form a new solution. A diverse set of destroy and repair operators is important for ALNS performance.

For more information about ALNS we refer the interested reader to Pisinger and Ropke (2010). For easy reference, we present in Algorithm 1 a pseudo-code for ALNS that is similar to the pseudo-code given in Pisinger and Ropke (2010).

After a destroy/repair iteration, the new solution is accepted according to a Simulated Annealing (SA) (Nikolaev and Jacobson, 2010) criterion. An improving solution is always accepted, whereas the probability of selecting a non-improving solution is determined by the size of the deterioration and a temperature parameter that is reduced during the destroy/repair iterations according to a cooling schedule. When finished, the cooling schedule is restarted.

For GPU-ALNS, we select a subset of the destroy and repair operators proposed in Pisinger and Ropke (2007). Below, we give a overall presentation of the workings of these operators for the DCVRP. All operators are

Algorithm 1 *ALNS*

Input: feasible solution x;
    set of destroy operators Ω^-;
    set of repair operators Ω^+;
Initialize: best solution $x^b = x$;
    operator scores $\rho^- = (1, ..., 1)$, $\rho^+ = (1, ..., 1)$
repeat
    $k = random(k_{min}, k_{max})$
    Select $d \in \Omega^-$ and $r \in \Omega^+$ using ρ^- and ρ^+
    $x' = r(d(x, k), k)$ ♯ apply operators
    if $accept(x', x)$ **then**
        $x = x'$
    end if
    if $(cost(x') < cost(x^b))$ **then**
        $x^b = x'$
    end if
    Update ρ^- and ρ^+
until $stop()$

parameterized with k, the selected number of requests to remove and insert on a given iteration. Details on the implementation of these operators on the GPU are provided in Section 3.4.

Random Removal chooses k requests in the current solution randomly and removes them.

Worst Removal iteratively removes the request that provides the highest reduction of the objective, until k requests have been removed.

Related Removal first chooses a seed at random. Successively, the closest request to the one most recently removed according to a relatedness metric defined for the problem at hand, until k requests have been removed. For the DCVRP, the relatedness between two requests is simply the distance between them. Randomization can be added to the relatedness measure in order to select less related requests at a chosen probability.

Historical Node-Pair Removal. For each edge in the graph of the problem, the objective of the best known solution an edge has been a part of is recorded. We remove the k requests that have the largest sum of incident edges.

In the **Cluster Removal** operator, a single route is selected and split into

two components by solving a minimum spanning tree problem with Kruskal's algorithm. Stopping just before the last iteration results in two components. One of those is chosen at random and removed.

In k iterations, **Greedy Insertion** picks the unserviced request that can be inserted into the solution in the cheapest way, and inserts it in its best position.

Regret-2 Insertion The Regret-2 value is a measure of criticality. It is defined as the cost difference of inserting a given request in its second best and best positions. In k iterations, regret values are calculated for unserviced requests, and the request with the highest regret value is inserted in its best position.

3. ALNS on the GPU

In this section we describe our design for GPU-ALNS and discuss implementation challenges. We select CUDA as programming language as it is mature and comes with a set of useful tools like the NVIDIA NSight Development Platform with debugging, profiling, and tracing functionality. For the reader new to CUDA and GPU programming, there is an introduction in the on-line appendices.

A major challenge when designing a new GPU based algorithm, is to choose a design that not only ensures enough parallelism in all major steps such that the GPU is fully utilized, but also leads to all these computations actually being useful, i.e. they lead to better solutions and/or shorter computation times. Especially in optimization using neighborhood exploration, it is easy to resort to searching through larger parts of the neighborhood, or even bigger neighborhoods, as means of providing enough work to fill the GPU. However, evaluating larger parts or neighborhoods does not necessarily lead to better solutions or faster algorithms (see for example Pisinger and Ropke (2007); Basseur and Goëffon (2015)). We describe our general design of the GPU based ALNS in Section 3.1, including reasoning for the choices taken. The reasoning may prove helpful to other researchers interested in developing GPU based optimization algorithms. The details of our design are explained in the remaining subsections. Our solution representation is presented in Section 3.2. Section 3.3 explains how we compute the initial solution(s), whereas in Section 3.4 we go into more detail about which operators were implemented on the GPU and how. The stopping criterion for

GPU-ALNS is explained in Section 3.5. Post processing of the final solution is explained in Section 3.6.

3.1. General Design

The first, and maybe most important, design decision for a GPU version of ALNS is the decision on how to distribute the work on kernels, blocks, warps, and threads. Two main points are of importance. The kernel(s) need(s) to be computationally intensive enough to avoid the problem of kernel launches and other setup work dominating the time. This would lead to the GPU being idle major parts of the time. Regarding kernel execution, our design needs to support enough parallel work to fill the whole capacity of the GPU. In general, we have two main alternatives: A *single solution* design, in which all kernels and elements of each kernel work together, or a *multiple solutions* design. For the latter alternative, we need to decide which elements in the compute grid work together on one solution.

To fill the GPU sufficiently, we need to provide a large number of parallel threads with ample work. The Titan GPU, for example, supports up to 28,672 active threads. A destroy/repair neighborhood requires sequential execution of the destroy and repair operators. In combination with the inability to synchronize between blocks in CUDA, this sequentiality leads to the need of a separate kernel for each operator for the single solution design. Although, at least for large problem instances, the destroy/repair operator computations might support a large enough number of threads, this design will lead to few operations per thread. The result would be many small kernels, each with a short runtime, leading to the setup problem mentioned above.

A potential remedy, at least for large size instances, would be to increase the amount of work performed by an operator by using higher values for k, the number of requests to remove and reinsert in an ALNS iteration. However, the literature tells us that k values higher than a certain constant do not yield improved solution quality, see Pisinger and Ropke (2007). Hence, even with the assumption of perfect parallelism in the operators, the single solution design will lead to inefficient usage of the GPU as computational resource. In addition, not all steps in the ALNS algorithm, nor all operators, will provide perfect parallelism. For example, with the right data structures, updating the solution will require very little work. We conclude that single solution is not a promising design for a GPU based ALNS.

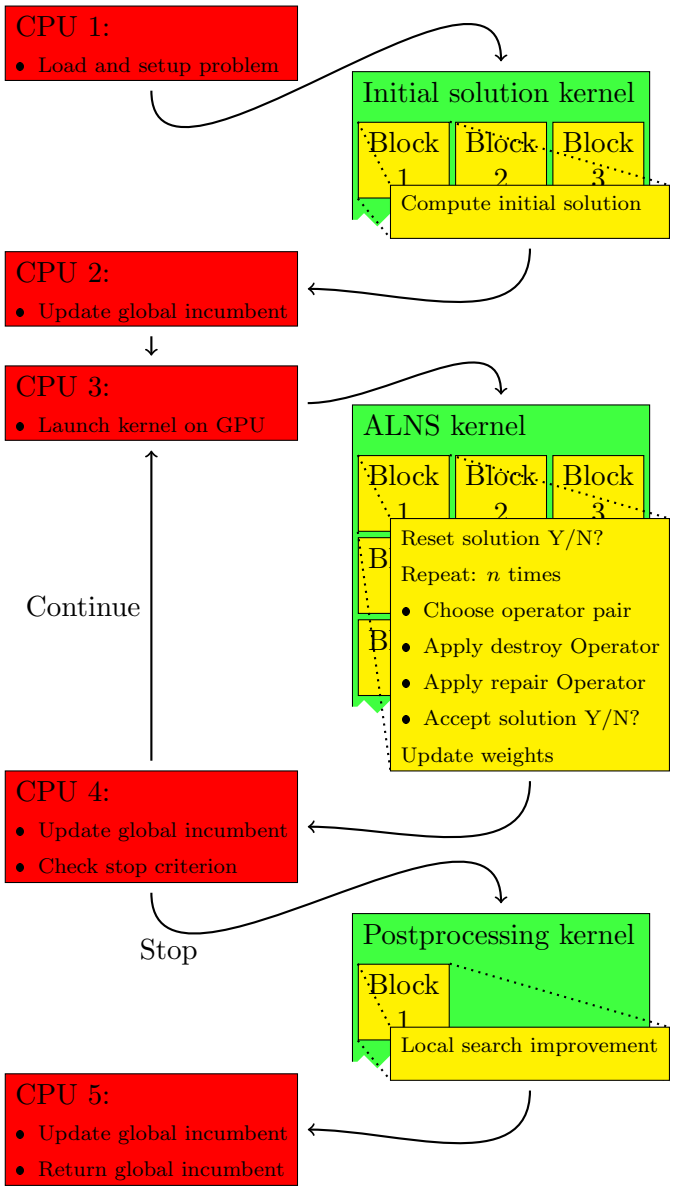


Figure 1: General design of GPU-ALNS. *Incumbent* stands for the best solution found so far.

We therefore choose the multiple solution design for GPU-ALNS by running multiple independent ALNS search processes, each starting with its own initial solution. With ALNS process we here refer to the ALNS as defined in Section 2. This design leads to a 2-level hierarchy of parallelism - at the higher level we have the parallel running processes, and at the lower level we have the parallelism provided within the steps of a single ALNS process. The advantage of this design is that the parts of ALNS that do not lend themselves easily to parallelism will not stall the whole algorithm, but only delay a single process. The different ALNS processes work independently; hence, also the choice of operator pair in each iteration is independent between the processes. Another advantage of our multiple solution design is that the different ALNS processes can exchange information. How we utilize this opportunity for cooperative search in GPU-ALNS is described in detail below. A potential disadvantage of the multiple solution design is that it may take long for each individual search process to converge to a final solution. However, the typical sequential ALNS algorithm as described in Algorithm 1 will not only be executed once, but instead with multiple restarts based on different initial solutions. In GPU-ALNS we keep the multiple restart approach; hence it does not matter that it will take some time to find the first solution.

The multiple processes design suggests having one block running one ALNS process. Since all the threads within a block can be synchronized easily, the need for separate kernels for the separate aspects of the ALNS process disappears. This reduces the amount of overhead time spent on launching and synchronizing the different kernels. However, independent ALNS search processes based on different initial solutions may take widely different times to converge. If a block would run the whole ALNS process, this will lead to the problem of having many ALNS processes, i.e. blocks, waiting for the last one to finish. We still choose to let the ALNS kernel have one block per process. However, the kernel performs a fixed number of ALNS-iterations. The stopping criterion is checked afterwards on the CPU. In this way, we still have all the advantages of the one-block-one-ALNS approach, but avoid stalling the GPU while waiting for the last block to finish. The stopping criterion is extended to include a test of whether to restart with a new initial solution because of convergence or lack of solution quality progress.

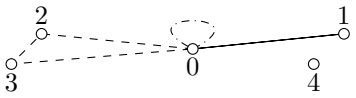
This setup yields the general design illustrated in Figure 1. We have a synchronized CPU-GPU behavior; the CPU is idle while the GPU is working, and vice versa. However, the design is such that the GPU will be busy almost

all the time since the CPU operations between kernel calls are negligible compared to the time spent on kernel execution. Of course this implies that the CPU is basically idle, leaving it free to do other work.

The overall process illustrated in Figure 1 is as follows. First, the problem instance is loaded and data structures are initialized on the CPU. Then the *Initial solution kernel* is launched, whereby the GPU data structures are initialized, and a set of initial solutions is calculated on the GPU, to be used also for later restarts. Details on the construction of initial solutions are found in Section 3.3. After initialization of the global incumbent (i.e., the best solution found so far among all ALNS searches) on the CPU, the next step is to start a number of independent ALNS processes by launching the ALNS kernel that performs a fixed number of iterations of the ALNS loop (we use 100). In GPU-ALNS, each destroy/repair *pair* of operators has its own score. The score is not updated on every ALNS iteration, but after the batch of iterations has finished. After the scores are updated the ALNS kernel finishes, and the CPU continues by accessing data on the GPU to decide whether GPU-ALNS has converged, or whether it should continue. The stopping criterion is explained in more detail in Section 3.5. If the decision is to continue, another instance of the ALNS kernel is launched. If not, the GPU-ALNS stops after post-processing of the global incumbent (see Section 3.6).

For the execution configuration of the ALNS kernel, our goal is to provide just enough blocks to fully saturate each Streaming Multiprocessor. In experiments, we found that using a combination of 512 threads per block and 2 blocks per Streaming Multiprocessor yields the best overall performance. This setting ensures a good balance between the number of independent ALNS search processes, i.e., the number of solutions being worked on simultaneously, and the number of threads available for parallel work for each process.

We previously mentioned that running multiple independent search processes in parallel yields information that can be exploited. We use it in GPU-ALNS in the following ways. First, some operators utilize historical search information that ideally should be based on all solutions considered so far, and hence should be shared among all search processes. Moreover, the stopping criterion for GPU-ALNS is based on the *local incumbents* of all search processes. Finally, the criterion for deciding whether a given process should restart from a new initial solution uses information from all local incumbents. Restart takes place when the process is considered to have



	Depot	Requests				Artificial Depots			
Index	0	1	2	3	4	5	6	7	8
Next	-8	6	3	0	-1	7	2	1	$-\infty$
Prev	3	7	6	2	x	-1	1	5	x
Tour id	-1	2	1	1	x	0	1	2	3

Figure 2: Doubly linked list representation of a solution with three tours (one is empty) and one not served request. x stands for undefined value.

converged, meaning that the local incumbent has not changed for a certain number of iterations. A second criterion for restart is that the solution process is not promising. If the local incumbent is substantially worse than the global incumbent and it has not improved for some time, we consider it futile to continue and restart with another initial solution.

The multiple solutions design of GPU-ALNS could also be exploited by adaptive memory procedures or Evolutionary Algorithms, in order to generate new solutions from local incumbents. We experimented with this idea, using a simple Genetic Algorithm, but it neither improved the quality of solutions, nor did it speed up the algorithm. We therefore decided not to include the Genetic Algorithm in our final version. We intend to investigate such ideas in our future work.

3.2. Solution Representation

We use a giant tour representation for our solutions. As data structure we basically have two options: an array, or a linked list. The array approach is more traditional and quite popular especially on the GPU. This is due to the intuitive representation and the fact that the i-th thread can work on the i-th node in the solution. However, it has several disadvantages. It involves a lot of copying when inserting and removing a request from the solution. Also, when calculating the cost of inserting a request between nodes at position i and $i+1$, we first need to read which nodes are at those positions, and then find the distance costs. A linked list in comparison has the advantage that inserting and removing a request is trivial and takes constant time. Moreover,

we do not need to read which node is at which position, since we work directly on the nodes. However, this is also the biggest weakness of the linked list, as we cannot deduce the position of a node without iterating through the list. The latter is probably the reason why linked list representations have not been popular on the GPU.

We decide to use a doubly linked list representation for our solution, as illustrated in Figure 2. The main idea here is that the i-th thread will not work on the node in i-th position, but on the $f(i)$-th node. Of course, this node might not be part of the solution at this moment, then the corresponding thread simply jumps to the next node it shall consider. This might lead to some warp divergence, but with the right design of the mapping f, this can be minimized. For example, when considering the cost of inserting requests into the solution, threads i and $i+1$ can consider inserting requests r_i and r_i+1 after node $f(i)$. In this way, if a node is not part of the solution, whole warps might be able to jump at the same time. Our experiments have shown that the linked list representation outperforms the array representation in our code, for further details we refer to on-line appendices.

To make the linked list representation more effective, we use the following rules. The linked list is actually represented as a 2-dimensional array with $n+m+1$ columns and 3 rows, with n being the number of requests and m being the maximal number of tours possible in the solution. We introduce m artificial depot nodes, which have the indices $m+1,\ldots, n+m+1$ and the same location as the depot. They represent the start depots of each tour, and each tour ends a the start depot of another tour, except the last one which ends at the original depot at index zero. The columns 1,. . .,n are the request nodes. The three rows in principle contain the index of the next node, the previous node, and the tour id. The tour id is simply an integer identifying which tour a node belongs to, the tour ids do not need to be in a specific order with respect to the order of tours. Each artificial node has its own fixed tour id that never changes. The next entry is positive if the node is in the solution and negative if not, except for the depot at column zero, which always has a negative next entry as it is the last node in the solution. If a node is not in the solution, the entry for the previous node is undefined, as is the tour id for a request that is not in the solution. The next entries for the artificial depots not in the solution are creating a simple linked list of unused artificial depots, just with negative index. The next entry of the depot node (column zero) is the negative index of the first unused artificial depot in that list, or minus infinity if no unused artificial depot node exists.

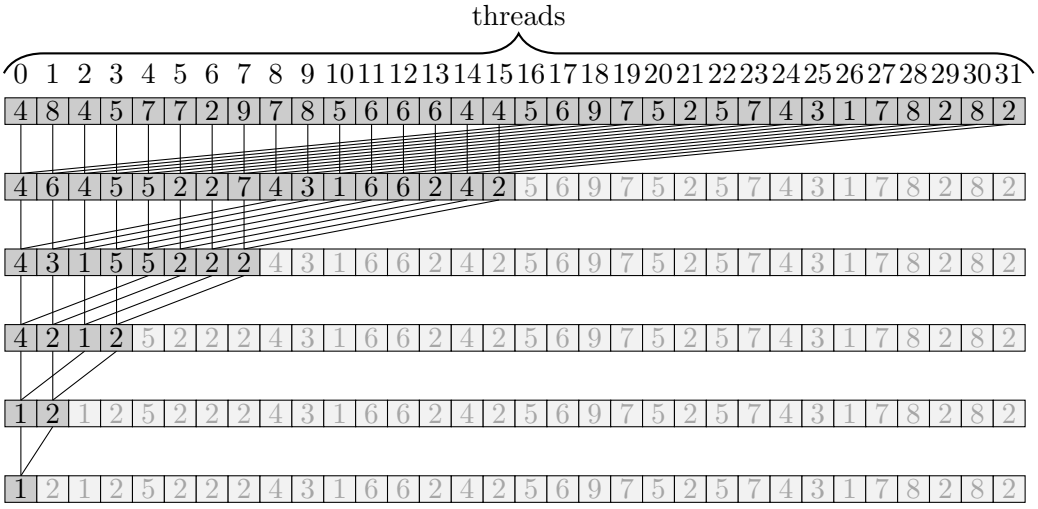


Figure 3: A SIMD reduction algorithm - Find minimum example.

The advantage of this approach is that first of all we do not need to consider all artificial depot nodes when checking for inserting a request, as long as we have some bound telling us that all artificial depot nodes above that bound are unused. This bound can be computed by simple bookkeeping.

3.3. Initial Solutions

The initial solutions are generated directly on the GPU by the Initial solution kernel. As for the ALNS kernel, each block in the Initial solution kernel computes initial solutions independently of the other blocks, but with different parameters to minimize the chance of generating the same initial solution multiple times. There are three construction methods. The first uses the Greedy Insertion operator to insert all requests into the solution. For the second, a random number of requests are placed randomly into the solution. The remaining requests are inserted using Greedy Insertion. The third method is maximum dispersion.

3.4. GPU Operators

The operators are at the heart of the ALNS algorithm and therefore need to support enough parallelism in order to utilize the GPU to a high degree. Of course, the operators also need to perform tasks that improve solution quality. The operators selected for GPU-ALNS have been briefly described in Section

2. We do not try to replicate those CPU operators exactly. Instead, our aim is to create operators that utilize the GPU efficiently, and whose effects mimic those of the CPU versions. For some operators it is simple to perform the conversion to a data parallel device, whereas for others a straightforward conversion would be too inefficient. In the following we describe the GPU implementation of the selected operators. Since we have a design where each block works on its own solution, all the operators work in a single block.

Most operators involve finding the minimum or maximum across a set of possible changes. Using a sequential algorithm, this is straightforward as each of the changes is evaluated one at a time while keeping track of the best move. In a parallel context these moves are evaluated in parallel, thus we need a mechanism to compare the best move of each thread with the best moves of the other threads. This can be performed by a SIMD reduction algorithm, which in our setup will work on a single block. Assume we have $N = 2^i$ for some integer i threads in the block, and $m > N$ variables to find the minimum of. We first split the m variables into N subsets of nearly equal size, and each of the N threads finds the minimum in one of those subsets by the simple sequential algorithm, leading to exactly N variables. The SIMD reduction algorithm now finds the minimum among those N variables with a complexity of $O(log(N))$. In Figure 3 we show an example of a reduction algorithm for finding the minimum value across a warp (32 threads). This algorithm can be scaled up to N threads. In our figure each thread initially has a value between 1 and 9 in the top row. Then for each row each active thread finds the minimum among its own and another value as shown in the pattern. This is repeated $\log(32) = 5$ times, until the first thread ends up with the minimum value.

In the following, we describe the GPU implementation of the selected operators. Greedy Insertion will be described in full detail as it is rather simple and serves as a good example for the method used in all operators.

Greedy Insertion is simple to implement on the GPU as each thread can evaluate the insertion of a specific request into a position in the current solution. This can be done in parallel, and thus we can fully exploit the GPU.

We have given a set of requests R to be inserted into the solution, with $|R| = k$. The solution provides a set of positions where each request can be inserted. However, we have a linked list solution representation, where we do not have easy access to the positions. Instead, we have n requests and a set of artificial depot nodes D that might be in the solution as well. Observe

that $|D| \leq n$ as we might have a limit on the amount of tours and we also might have an upper bound on the number of artificial depot nodes used at the moment. Let $n + |D| = s$.

We define a potential insertion position identified by i as inserting a request after node i if the node is in the solution. If the corresponding node i is not in the solution, the potential insertion position is invalid. The node here might be either a request or an artificial depot node. A potential insertion is a combination of a request to insert and a potential insertion position. In total this gives $k \cdot s$ potential insertions to evaluate, where some may be invalid and others infeasible.

Algorithm 2 *GreedyInsertion*(k, s)

```
1: while 0 ≤ k do
       ♯ Initialize best insertion variables:
2:     bestIndex = −1
3:     minCost = ∞
       ♯ Index iterating thread's subset of pot. insertions:
4:     x = threadId
       ♯ Iterate thread's subset of potential insertions:
5:     while x < (k · s) do
6:         c = CostFunction(x, k, s)      ♯ See Alg. 3
7:         if c < minCost then
8:             minCost = c
9:             bestIndex = x
10:        end if
11:        x = x + N
12:    end while
       ♯ Find global best potential insertion:
13:    minReduction(minCost, bestIndex)
       ♯ Insert the best potential insertion:
14:    if threadId == 0 then
15:        insertIntoSolution(bestIndex, k, s)
16:    end if
17:    k = k − 1
18: end while
```

The Greedy Insertion operator works on this set of potential insertions just as a reduction would. We split the set of potential insertions into N

subsets, where N is the number of threads in a block. Then each thread evaluates one of those subsets sequentially and in this way finds the best of those insertions. Afterwards we apply the reduction algorithm to find the best potential insertion. To illustrate this process, we provide the algorithm in pseudo-code in Algorithm 2. The maybe most interesting point here is to see exactly how the set of potential insertions is split into subsets. This split is actually partially hidden in the cost function, whose pseudo code is presented in Algorithm 3. Observe that the potential insertions are organized such that if k is big enough, then whole warps will jump on invalid insertion positions.

Algorithm 3 $CostFunction(x, k, s)$

```
    ♯ Find potential insertion position node:
1:  i = ⌊x/k⌋
    ♯ Find request for potential insertion:
2:  j = x − (i · k)
    ♯ Initialize cost:
3:  c = ∞
4:  if InSolution(i) && Feasible(i, j) then
        ♯ Calculate cost of insertion:
        ♯ next(i) returns the node after i in solution
5:      c = dist(i, j) + dist(j, next(i)) − dist(i, next(i))
6:  end if
7:  return c
```

For simplicity, only thread 0 performs insertion of the request into the solution in Algorithm 2. Of course, this does not need to be the case. Updating the linked list structure is easy, so using only one thread is acceptable. However, it is possible to use several threads in the block to update the solution, and all other data that needs to be modified, in parallel. This requires that all threads know the best potential insertion index, e.g. by copying it to shared memory.

The GPU version of the **Regret-2 operator** performs exactly the same task as the CPU version, see Section 2. It is implemented using reduction, but with a twist. To compute the regret value of a request to be inserted, we need to find its best position. Let t be the tour where this position is. In addition we need the best insertion position for all tours except t. The set of potential insertion positions for the request are split into $M \leq N$

subsets, which are processed by M threads. Each thread keeps track of the best position and the best in a different tour. Subsequently, reduction is applied to the M pairs to find the best and second best position. This yields the Regret-2 value for this request. Once all Regret-2 values for all requests that shall be inserted have been computed in this way, a simple reduction on the regret values determines which request and where to insert it. Note that due to $M \leq N$, several or even all Regret-2 values can be calculated simultaneously. In this way we are able to perform Regret-2 in a highly parallel way with the same result as on the CPU.

Worst Removal can be implemented in a highly parallel way. It is essentially the opposite of Greedy Insertion, as all positions are considered for removal and we find the request with the highest saving and remove it.

In the **Random Removal** operator we remove a number of requests at random from the current solution. Due to our linked list solution representation, it is complicated to remove several random requests in parallel, although it is possible. However, due to the simplicity of drawing a random number and removing a request in the linked list solution representation, we remove the requests purely sequentially, basically yielding a non-parallel operator.

Related Removal is basically a version of Worst Removal with the distance to the seed request node as cost function. Hence this operator is implemented just as Worst Removal, it is using repeated reduction to find the requests to remove. We also experimented with another version that only removes requests from the same tour as the seed, but this idea did not perform equally well and was thus discarded.

As on the CPU, our GPU version of **Historical Node-Pair Removal** maintains a record of the value of the best solution an edge has been a part of. We then are able to find the request with the worst combined edge values using reduction, just as for the Worst Removal operator. To store the historical information, we use a common memory for the multiple ALNS processes running in parallel. In this way the processes exchange information in a collaborative way.

As mentioned in Section 2, **Cluster Removal** in Pisinger and Ropke (2007) splits a route in two components by solving a minimum spanning tree problem with Kruskal's algorithm. Although there exist GPU versions of minimum spanning tree algorithms, Kruskal's sequential algorithm does not lend itself easily to efficient GPU implementation. We developed a new cluster identification algorithm that exploits the GPU better. For each route we first find the average edge length, ignoring the edges from and to the

depot. Then we search for cluster defining edges, i.e., edges with a length greater than the average multiplied by a given factor. Hence there can be more than two clusters in a tour. This detection can be performed in a highly parallel manner. We remove a number of clusters, making sure not to delete entire tours on the way.

3.5. Stopping Criterion

The stopping criterion is based on multiple sub-criteria. First, we measure how many times the ALNS kernel has been called without any improvement to the global incumbent. If this is greater than some threshold, GPU-ALNS terminates. The threshold is a piecewise constant function depending on the number of iterations performed so far, with the following values: $(0-125, \infty)$, $(126-250, 30)$, $(>250, 10)$.

Second, we calculate a convergence factor. We start by computing the "standard deviation" of the costs of local incumbents, but use the global incumbent as "mean" during these calculations. The convergence factor is the "standard deviation" divided by the cost of the global incumbent. If the convergence factor is below a threshold T, and there has been a certain number K of ALNS kernel calls without improvement, the algorithm will terminate. Two combinations of T and K are monitored: $(T = 0.005,\ K = 20)$ and $(T = 0.00125,\ K = 5)$.

Finally, we count the percentage of how many of the local incumbents are equal to the global incumbent. If this is larger than 15 %, we add a third combination of T and K to be monitored: $(T = 0.002,\ K = 10)$.

3.6. Post Processing

When the stop criterion is satisfied, the algorithm performs local search on the global incumbent as a post processing step on the GPU, for final intensification. Two-opt is run to a local optimum.

Alternatively, local search could also be applied inside the ALNS kernel, either for each kernel call, or less frequently. Our experiments show that this does not provide a significant improvement.

4. Experiments & Results

The main goal of this paper is to study how well the ALNS algorithm can be parallelized on a stream processing device such as the GPU. We compare the performance of our GPU-ALNS with a state-of-the-art ALNS for

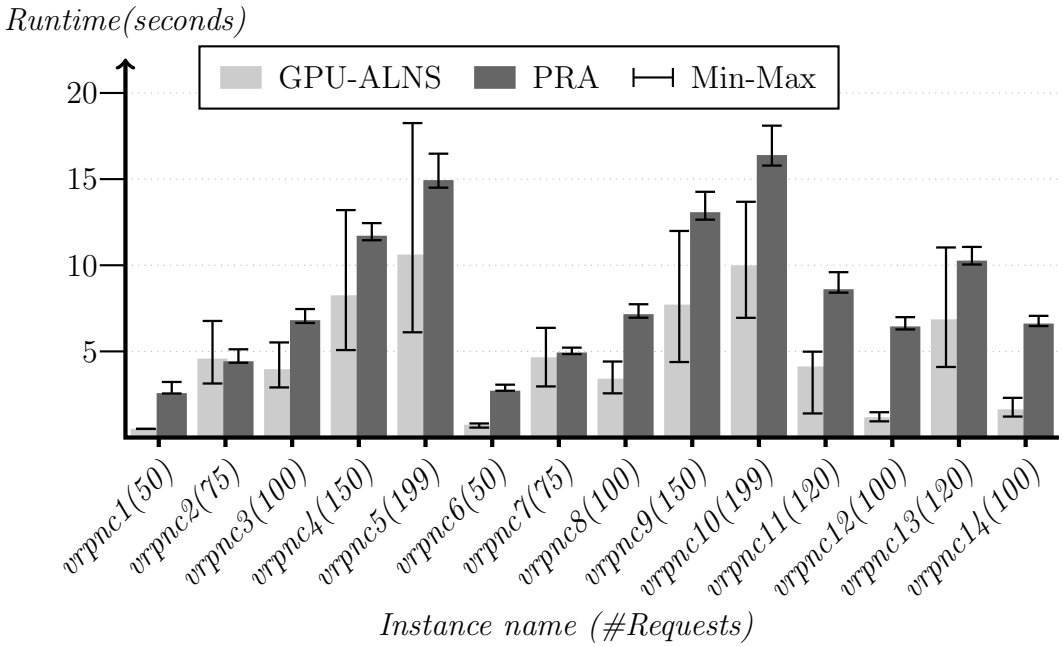


Figure 4: Runtimes for the Christofides et al. instances.

multi-core CPUs, namely the actual implementation by Pisinger and Røpke compiled on the 11^{th} of December 2014. Below we use PRA when referring to this implementation executed on a single thread, and PRA-8 when it is running on 8 threads. The platform for our computational experiments is an Intel Core i7-4820K 3.7 GHz with 4 cores, hyperthreading, and 32 GB memory, with a GeForce TITAN GPU with 6 GB global memory. We used two heavily used sets of DCVRP instances: the Christofides, Mingozzi, and Toth benchmark that has 14 instances with 50-199 requests (Christofides et al. (1979)), and the 12 Li, Golden, and Wasil instances with 560-1200 requests (Li et al. (2005)). For both algorithms, we performed 100 repetitions on these instances. Furthermore, we have created 16 new larger instances that we refer to as BHS. On these instances we have exclusively compared the GPU-ALNS to the PRA version using 8 threads (PRA-8), performing 50 repetitions. For detailed results on all instances presented we refer to Appendix B.

In Tables 1 and 2 for each instance we report the number of requests,

			GPU-ALNS		PRA		Comparison	
Instance	Req.	BKS	Error	Time	Error	Time	Error	τ
1	50	524.6	0.00%	0.50	0.00%	2.58	0.00%	5.12
2	75	835.3	0.29%	4.58	0.08%	4.43	0.21%	0.97
3	100	826.1	0.00%	3.97	0.01%	6.81	-0.02%	1.72
4	150	1,028.4	0.32%	8.26	0.16%	11.71	0.16%	1.42
5	199	1,291.3	0.88%	10.62	0.32%	14.94	0.56%	1.41
6	50	555.4	0.00%	0.74	0.00%	2.72	0.00%	3.66
7	75	909.7	0.38%	4.66	0.00%	4.94	0.37%	1.06
8	100	866.0	0.00%	3.42	0.00%	7.16	0.00%	2.09
9	150	1,162.6	0.39%	7.72	0.10%	13.08	0.29%	1.69
10	199	1,395.9	0.65%	10.00	0.60%	16.40	0.05%	1.64
11	120	1,042.1	0.02%	4.13	0.00%	8.61	0.02%	2.09
12	100	819.6	0.00%	1.18	0.00%	6.45	0.00%	5.48
13	120	1,541.1	0.15%	6.86	0.13%	10.27	0.02%	1.50
14	100	866.4	0.00%	1.64	0.00%	6.62	0.00%	4.05
Total/Average		13,664.4	0.27%		0.13%		0.14%	1.71

Table 1: Comparison of GPU-ALNS and PRA for the Christofides et al. instances.

the best known upper bound (BKS), and the error percentage relative to BKS and runtime in seconds for GPU-ALNS and PRA, respectively. The final two columns show the difference in error percentage (negative number means lower cost for GPU-ALNS), and the time ratio τ, with

$$\tau = (\text{Time PRA[-8]}) \ / \ (\text{Time GPU-ALNS}).$$

We use the term time ratio and not speedup, to emphasize that there are differences in the implementations of ALNS that we compare.

The results when comparing GPU-ALNS to PRA for the smaller Christofides et al. instances in Table 1 show that the difference in solution quality is very small or zero for eight of the instances. For the remaining six instances, differences are small, with GPU-ALNS being 0.56% worse in the most extreme case. The average results for both algorithms are close to the best known solutions, with average errors of 0.27% and 0.13% for GPU-ALNS and PRA, respectively. In Figure 4, the minimum, average, and maximum runtime for each instance are depicted for both algorithms. It can be seen that GPU-ALNS is faster than PRA on most instances, but the runtime

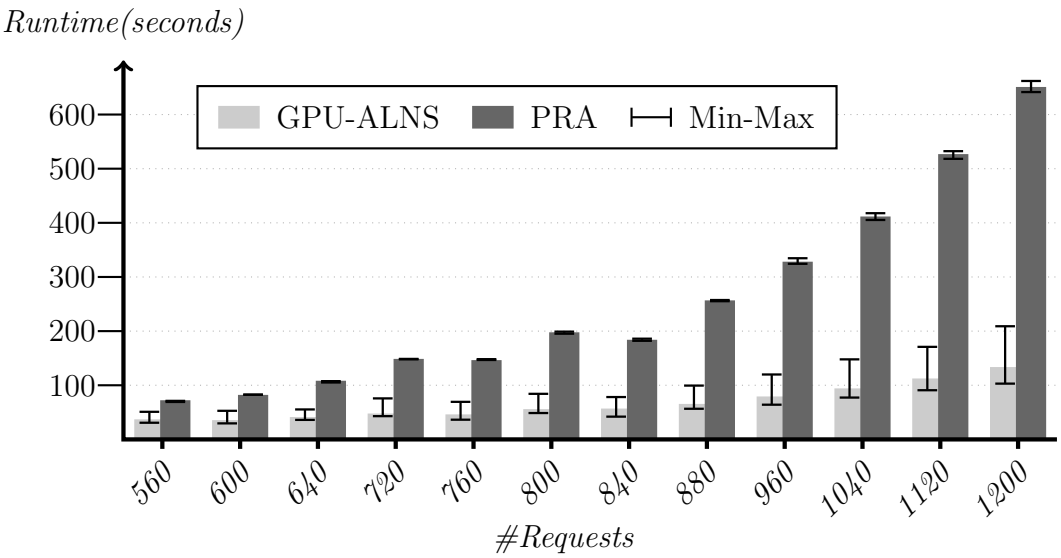


Figure 5: Runtimes for the Li et al. instances.

			GPU-ALNS		PRA		PRA-8		ΔPRA		ΔPRA-8	
Instance	Req.	BKS	Error	Time	Error	Time	Error	Time	Error	τ	Error	τ
1	560	16,213	0.49%	37.10	1.18%	72.34	1.25%	16.05	-0.68%	1.95	-0.74%	0.43
2	600	14,499	1.15%	35.60	0.92%	82.21	0.91%	17.56	0.22%	2.31	0.23%	0.49
3	640	18,801	0.68%	41.34	0.98%	108.36	0.88%	23.40	-0.29%	2.62	-0.19%	0.57
4	720	21,389	0.96%	47.83	1.93%	148.71	1.69%	31.93	-0.95%	3.11	-0.71%	0.67
5	760	16,669	3.40%	46.23	1.67%	146.46	1.83%	30.45	1.70%	3.17	1.54%	0.66
6	800	23,978	1.20%	55.92	2.72%	197.82	2.64%	43.64	-1.48%	3.54	-1.40%	0.78
7	840	17,373	3.44%	56.99	1.63%	184.03	1.70%	38.20	1.78%	3.23	1.71%	0.67
8	880	26,566	1.17%	65.57	2.71%	256.78	2.45%	56.53	-1.50%	3.92	-1.25%	0.86
9	960	29,154	1.13%	79.32	2.90%	328.22	2.76%	73.05	-1.72%	4.14	-1.59%	0.92
10	1040	31,743	1.23%	94.22	2.96%	411.70	2.69%	96.84	-1.68%	4.37	-1.42%	1.03
11	1120	34,331	1.11%	112.52	2.98%	526.92	2.48%	129.73	-1.81%	4.68	-1.34%	1.15
12	1200	37,159	1.12%	133.66	2.32%	650.92	2.06%	170.91	-1.18%	4.87	-0.92%	1.28
Total/Average:		287,875	1.34%		2.27%		2.09%		-0.90%	3.86	-0.73%	0.90

Table 2: Comparison of GPU-ALNS, PRA and PRA-8 for the Li et al. instances.

fluctuates more. Except for a few of the instances that seem particularly easy, the GPU-ALNS is only moderately faster than the PRA. We conclude that on the Christofides et al. instances, there is no clear winner. This is not unexpected, as GPU-ALNS is designed for massive parallelism that will only be effective if the computational demand is large.

In Table 2, results comparing GPU-ALNS to both version PRA and PRA-8 for the larger Li et al. instances are reported. First we consider the comparison with PRA. On solution quality, GPU-ALNS outperforms PRA on 8 instances, is worse on 2 instances, and performs equally well on 2 instances. On average, the objective value is 0.91% better for GPU-ALNS. The runtimes are also improved, with a time ratio of 3.86 on average. Figure 5 clearly indicates that time ratio increases with instance size. Although GPU-ALNS has a higher variance in runtime, its worst case runtime is still less than the best case of PRA for all instances. Regarding solution quality we can draw the same solution for the comparison to PRA-8. In general the two version of the Pisinger and Røpke algorithm (PRA and PRA-8) have a very similar performance w.r.t. solution quality. PRA-8 is faster than GPU-ALNS for all instances with less than 1000 requests, and slower for the instances with more than 1000 requests. Figure 6 shows the time ratio for GPU-ALNS relative to PRA and PRA-8. The y-axis to the left gives the time ratio relative to PRA, whereas the y-axis to the right gives the time ratio relative to PRA-8. The curves show a similar pattern – a clear trend of increasing time ratios for larger instances.

To investigate whether the trend shown in Figure 6 continues for larger instances, we have created new instances with sizes from 1400 to 4500 requests. There are two classes of new instances: a set based on the existing Li et al. instances (BHS1-BHS4), and a set of new generated instances (BHS5-BHS10). To create the first set we start by merging the existing Li et al. instances and removing overlapping nodes, yielding a base instance with 2000 requests. We create instances with fewer requests by randomly removing requests. The second set is generated using the method for generating new instances described by Li et al..

As it has already been shown that the PRA and PRA-8 algorithms have very similar solution quality and that PRA-8 is always faster, experiments for the new instances have only been performed using the PRA-8 algorithm. The detailed results can be found in Tables B.9 and B.10. The results show that the solution quality is very similar with a slightly higher variation between the two algorithms with the GPU-ALNS outperforming on 5 of 10

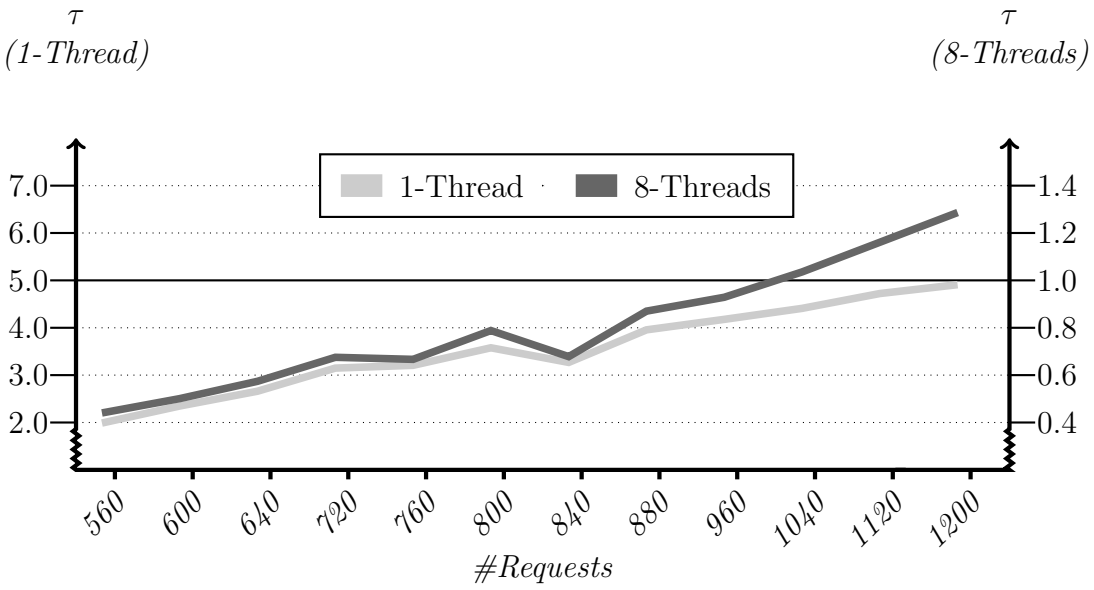


Figure 6: Time ratio for GPU-ALNS relative to PRA and PRA-8 respectively.

of the new BHS instances, PRA-8 on 4 of 10 and 1 of 10 that is too close to call. With the goal of this paper being to compare the time ratio between the algorithms, we believe it is fair to conclude that the two algorithms show approximately equal performance on the new instances. It is therefore fair to compare the runtime between the two.

Figure 7 shows the runtimes for the new instances. The runtime fluctuates more on the GPU-ALNS, which is due to the stopping criterion being more adaptive, hence it sometimes needs extra time to finish, whereas the PRA-8 is based on a fixed number of iterations and therefore is rather stable with respect to runtime. In Figure 8 the time ratio of GPU-ALNS to PRA and PRA-8 respectively is shown for both the Li et al. instances and their extensions BHS1 to BHS10. It illustrates that the clear trend of increasing time ratios for larger instances continues.

5. Conclusions & Future Research

In this paper we show, as far as we know for the first time, that it is possible to develop an efficient GPU implementation of one of the best

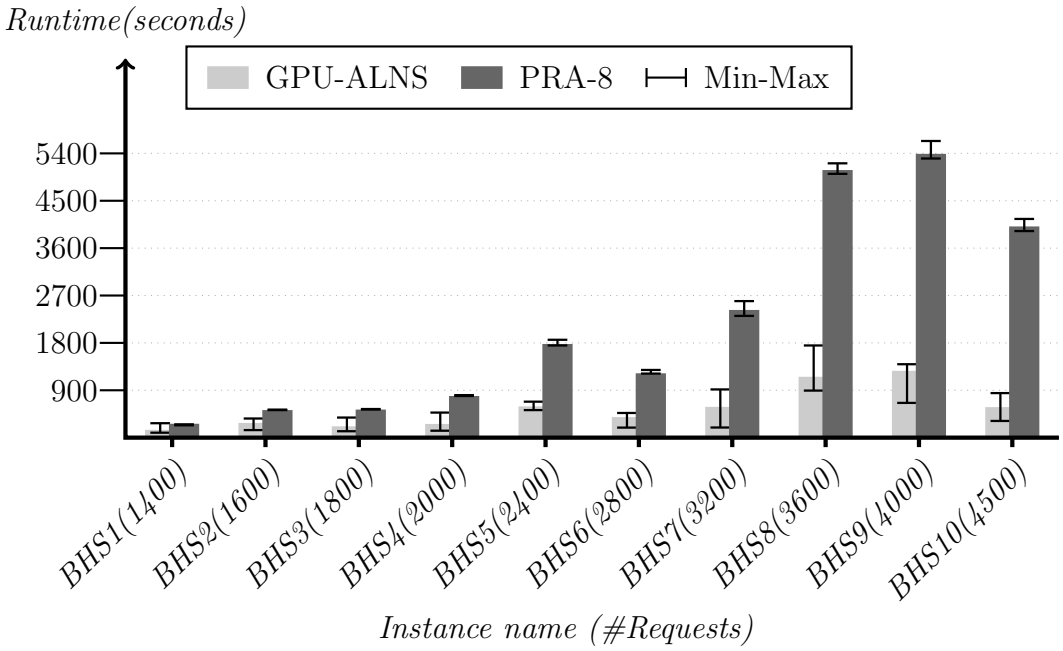


Figure 7: Runtimes for the new instances.

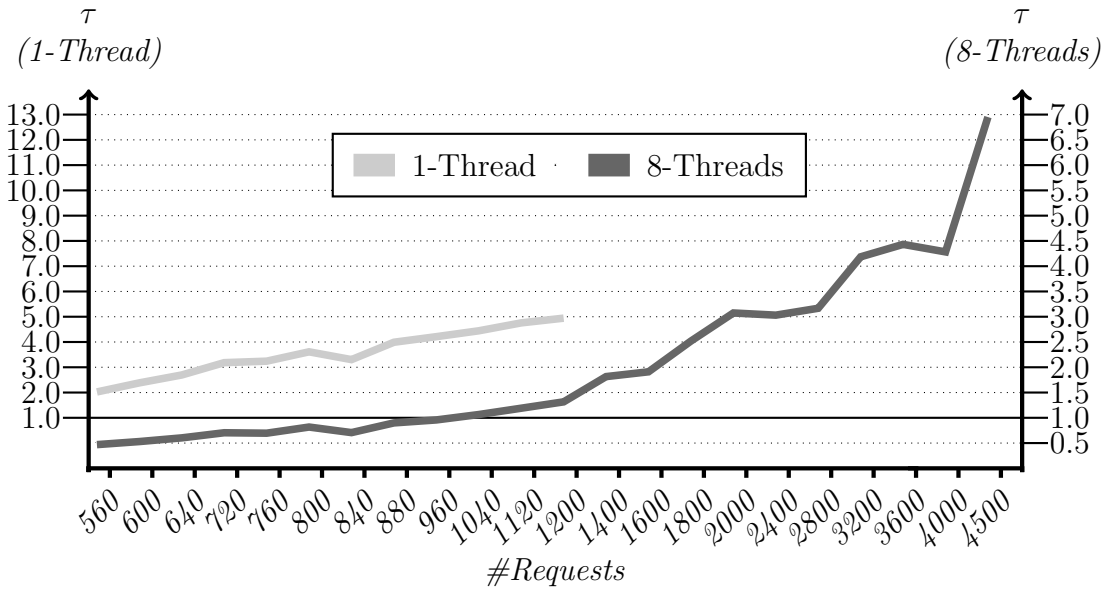


Figure 8: Time ratio for GPU-ALNS relative to PRA and PRA-8, respectively, for the Li et al. instances and for BHS1 to BHS10.

metaheuristics for vehicle routing problems: Adaptive Large Neighborhood Search. Experiments show that our data parallel ALNS implementation is competitive regarding solution quality, and outperforms a state-of-the-art CPU-based algorithm on runtime.

We provide experimental results for the well-known Christofides et al. and Li et al. DCVRP benchmarks and new larger instances (BHS). We compare our GPU-ALNS implementation to an optimized CPU-ALNS implementation by Pisinger and Røpke, in two versions: a single threaded (PRA) and a task parallel version running on 8 threads (PRA-8).

For the smaller Christofides et al. instances, we get a modestly shorter runtime relative to the PRA. For the medium sized Li et al. instances we are faster than PRA on all instances. Compared to the PRA-8, GPU-ALNS is slower for instances with less than 1000 requests and slightly faster on the larger instances. We observe that the time ratio increases with the number of requests. On the new BHS instances with up to 4500 requests, the GPU-ALNS shows a further improved time ratio of up to 6.9 when comparing to PRA-8 (eight parallel threads). Again, time ratios generally increase with instance size. The vital part of ALNS is the large neighborhood search, where destroy operators remove parts of an existing solution and repair operators reinsert them. The literature shows that good performance requires a diverse repertoire of operators. For GPU-ALNS, we selected a subset of the operators proposed in the literature for vehicle routing problems. Compared to the CPU based implementation, it is necessary to adapt some of the operators to achieve a good utilization on the GPU. For further improvement, research on operators that are specifically designed to utilize the advantages of the GPU is necessary. Our investigations indicate that adding operators does not necessarily have detrimental effects on runtime.

Acknowledgments

The authors would like to thank David Pisinger and Stefan Røpke for kindly providing us with their state-of-the-art ALNS code for the CPU.

The work presented in this paper was funded by the Research Council of Norway as a part of the Collab project [Contract 192905/I40, SMARTRANS], the Collab II project [Contract 227071/O70, SMARTRANS] and the DynamITe project [Contract 246825,TRANSPORT].

References

Alba, E., Luque, G., Nesmachnow, S., 2013. Parallel metaheuristics: recent advances and new trends. International Transactions in Operational Research 20 (1), 1–48.

Baldacci, R., Mingozzi, A., Toth, P., 2017. New route relaxation and pricing strategies for the vehicle routing problem. Operations Research 59 (2), 1269–1283.

Basseur, M., Goëffon, A., 2015. Climbing combinatorial fitness landscapes. Applied Soft Computing 30, 688–704.

Bixby, R. E., 2002. Solving Real-World Linear Programs: A Decade and More of Progress. Operations Research 50, 3–15.

Boschetti, M. A., Maniezzo, V., Strappaveccia, F., 2017. Route relaxations on gpu for vehicle routing problems. European Journal of Operational Research 258 (2), 456–466.

Brodtkorb, A. R., Hagen, T. R., Schulz, C., Hasle, G., 2013. GPU Computing in Discrete Optimization – Part I: Introduction to the GPU. EURO Journal on Transportation and Logistics 2 (1-2), 129–157.

Christofides, N., Mingozzi, A., Toth, P., 1979. The vehicle routing problem. In: Christofides, N., Mingozzi, A., Toth, P., Sandi, C. (Eds.), Combinatorial Optimization. Wiley, Chichester, pp. 315–338.

Christofides, N., Mingozzi, A., Toth, P., 1981. Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. Mathematical Programming 10, 255–280.

Irnich, S., Toth, P., Vigo, D., 2014. The Family of Vehicle Routing Problems. In: Toth, P., Vigo, D. (Eds.), Vehicle Routing: Problems, Methods, and Applications. SIAM, Ch. 1, pp. 1–33.

Kytöjoki, J., Nuortio, T., Bräysy, O., Gendreau, M., 2007. An efficient variable neighborhood search heuristic for very large scale vehicle routing problems. Computers & Operations Research 34 (9), 2743 – 2757.

Li, F., Golden, B., Wasil, E., 2005. Very large-scale vehicle routing: new test problems, algorithms, and results. Computers & Operations Research 32 (5), 1165 – 1179.

Nikolaev, A., Jacobson, S., 2010. Simulated annealing. In: Gendreau, M., Potvin, J.-Y. (Eds.), Handbook of Metaheuristics. Vol. 146 of International Series in Operations Research & Management Science. Springer US, pp. 1–39.

Pisinger, D., Ropke, S., 2007. A general heuristic for vehicle routing problems. Computers & Operations Research 34 (8), 2403 – 2435.

Pisinger, D., Ropke, S., 2010. Large neighborhood search. In: Gendreau, M., Potvin, J.-Y. (Eds.), Handbook of Metaheuristics. Vol. 146 of International Series in Operations Research & Management Science. Springer US, pp. 399–419.

Ropke, S., 2009. Parallel large neighborhood search - a software framework. In: Proceedings of MIC 2009: The VIII Metaheuristics International Conference.

Ropke, S., Pisinger, D., 2006. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation Science 40 (4), 455–472.

Schulz, C., Hasle, G., Brodtkorb, A. R., Hagen, T. R., 2013. GPU Computing in Discrete Optimization – Part II: Survey Focused on Routing Problems. EURO Journal on Transportation and Logistics 2 (1-2), 159–186.

Shaw, P., 1998. Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.-F. (Eds.), Principles and Practice of Constraint Programming CP98. Vol. 1520 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 417–431.

Toth, P., Vigo, D., 2014. Vehicle Routing: Problems, Methods, and Applications, 2nd Edition. SIAM Monographs on Discrete Mathematics and Applications, Philadelphia.

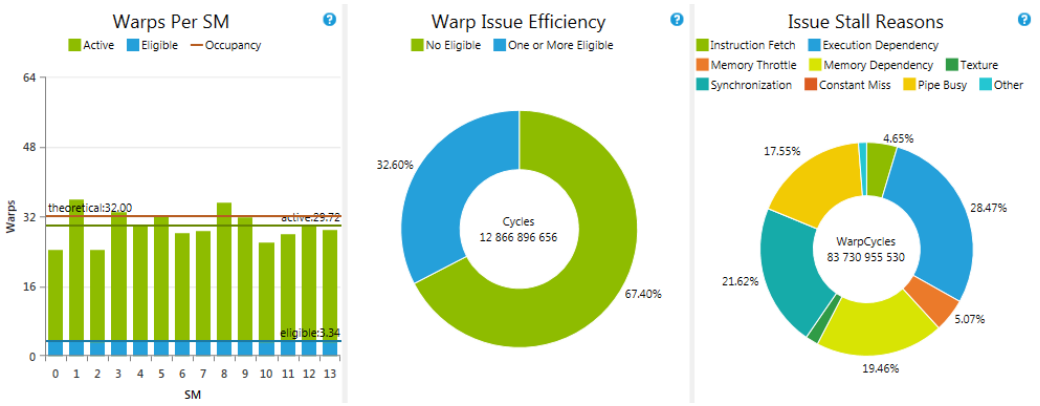


Figure A.9: Issue efficiency for one call of the ALNS kernel. The left column shows the theoretical occupancy and the average number of active and eligible warps. In the middle, the percentage of cycles with no eligible warps is shown, whereas the right displays the percentagewise distribution of reasons for stalled warps.

Vidal, T., Crainic, T. G., Gendreau, M., Prins, C., 2014. A unified solution framework for multi-attribute vehicle routing problems. European Journal of Operational Research 234 (3), 658–673.

Appendix A. GPU Efficiency

On the GPU, just as on the CPU, the speed of a specific algorithm depends on how well it uses the resources at hand. In this section we shortly discuss on how well GPU-ALNS uses the GPU resources. This section may safely be skipped by readers not interested in the more technical ascpects of GPU programming. The implementation is performed in C++ using CUDA 7.5 on a GeForce TITAN with 14 Streaming Multiprocessors and 6 GB global memory for all experiments. All profiling is performed using NVIDIA NSight in Visual Studio. The figures shown in this section are screenshots from the analysis generated by NSight.

We profiled our code in order to tune it towards efficient usage of the GPU, but did not perform exhaustive profiling and tuning. Instead we chose to profile at a strategic point in the development process, to maximize the effect while keeping the time spent at a low level. Our profiling was performed when the Simulated Annealing framework, the Worst Removal, Random Removal, Cluster Removal, Greedy Insertion, and Regret-2 Insertion operators

were implemented. At that point, changes would not introduce too much rewriting, but at the same time the central parts of the code could be profiled. We only profiled the Worst Removal / Greedy Insertion operator pair, and focused on tuning the corresponding code, and code common with, or similar to, code for other operators. With this approach we managed to decrease the runtime of the code by a factor five. For the interested reader, we present the details of the tuning process in on-line appendices.

Once the tuning process was finished, we continued to develop GPU-ALNS . We added the remaining operators, the stopping criterion, and the restart mechanism. During that we of course took the lessons learned during the profiling into account. When timing the final implementation with only using the Worst Removal / Greedy Insertion pair, the functionality added after the tuning increased the runtime of the kernel by 21% only. The additions, however, improve the performance regarding solution quality vs. number of iterations needed, and thus reduce the time until the stop criterion is satisfied. The additional functionality comes at a low price and does not destroy our improved GPU usage achieved through the profiling process. This is also an indication that adding operators to improve solution quality performance would not necessarily be computationally expensive.

The general design of GPU-ALNS ensures that most of the time is spent on the ALNS kernel. NSight tracing tells us that the GPU is busy 97.9% of the time, confirming our claim that the synchronous CPU-GPU approach does not hinder high GPU utilization. It also means that the efficiency of GPU-ALNS is directly related to the efficiency of the ALNS kernel.

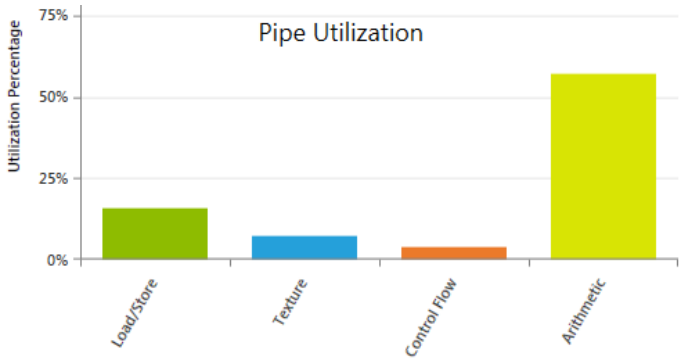


Figure A.10: Average pipe utilization percentage across the duration of one ALNS kernel call for the four major logical pipelines of the SMs.

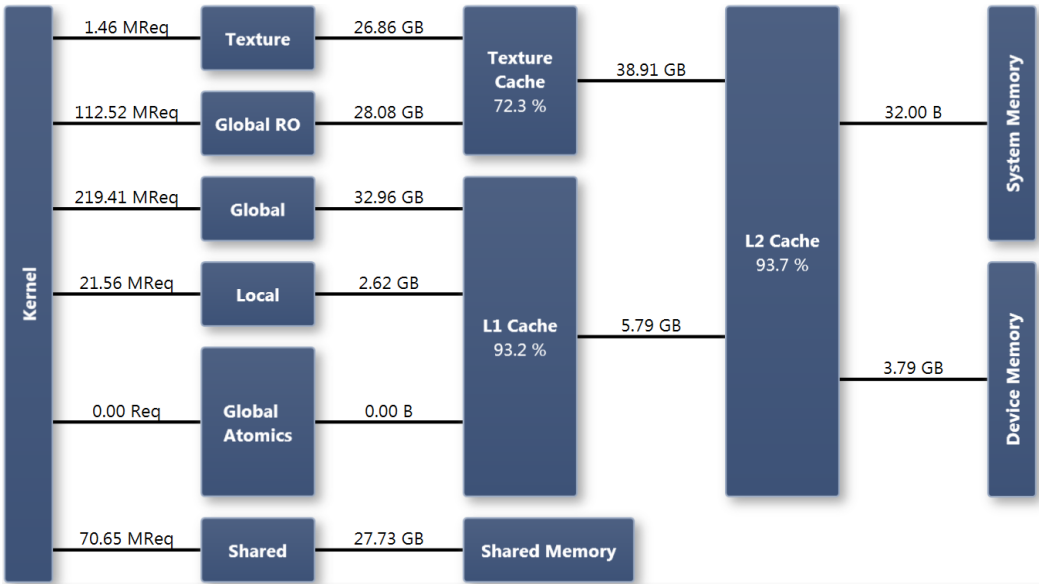


Figure A.11: Memory statistics for one ALNS kernel call. The connections from the kernel area report the total amount of memory requests, whereas the other links display the total amount of transferred memory in bytes. In addition, the cache hit rate percentage is shown for the caches.

As mentioned before we use a grid size which yields 2 blocks per Streaming Multiprocessor with 512 threads per block. The GeForce Titan has 14 Streaming Multiprocessors, yielding a grid size of 28 blocks.

Each Streaming Multiprocessor can run up to 2048 threads in parallel, leading to a theoretical occupancy of 50%. NSight reports that the achieved occupancy is 46.3%, very close to the theoretical maximum. Using only 1024 threads per Streaming Multiprocessor gives each thread 64 registers to use.

More profiling of the final version of our ALNS kernel is presented in Figures A.9, A.10, and A.11. It shows that we have on average 3.34 eligible warps per Streaming Multiprocessor in combination with having an eligible warp 32.6% of the time. We see that we have four major reasons which together stand for more than 81% of the warp stalls. These are, in decreasing order: execution dependency, synchronization, memory dependency, and pipe busy. The pipe utilization analysis shows that the arithmetic pipe is the one most busy, suggesting that the pipe busy reason for stalling warps is due too many arithmetic operations at one time.

Both execution dependency and pipe busy are caused by the order of

instructions executed. There are too many arithmetic instructions issued at the same time to fit into the pipeline. The results of previous instructions are needed but not yet available. Hence, the next operation can not be started yet. Both these issues could in theory be resolved if one could modify the order of operations, i.e., reformulate the code. Also memory dependency could be reduced if the operation needing memory access could be executed later. However, reformulating the code to achieve less dependencies and pipe pressure is a labor intensive task, with an uncertain guarantee of success. We therefore decided to focus on higher level optimization algorithm improvements instead of reformulating our code.

Memory dependency is also related to the way the memory is used, i.e., memory latency. Tuning led to the memory access pattern illustrated in Figure A.11. In the final configuration, we load the distance matrix through the read-only-data cache and read selected global memory loads through the L1 Cache, and local memory usage is minimized. The L1 Cache and the shared memory are configured to have the same size. In shared memory we keep the necessary data for the reduction operations used in the operators. If the problem instance size allows, we also keep data related to the solution in shared memory, otherwise these data reside in global memory. For all instances used in our experiments (up to 1200 requests), the solution related data always fitted in shared memory.

Appendix B. Detailed Experimental Results

On the following pages, the detailed results for all experiments are reported in Tables B.3 - B.8. Runtime and objective values for GPU-ALNS and PRA are reported for the Cristofides et al. and Li et al. instances. PRA-8 results are only reported for the Li et al. instances. Explanations for the column heading abbreviations in the Tables B.3 - B.8 are as follows: The "Diff" columns show the difference between the values for the two algorithms. The number of requests in an instance is given by "Req.". The best known upper bound is reported in the "BKS" columns. For each algorithm, 100 repetitions have been performed. The average, minimum, maximum and standard deviation values are given in the column "Avg.", "Min.", "Max" and "Std.", respectively.

Instance			GPU-ALNS				PRA				Diff			
Number	Req.	BKS	Avg.	Min.	Max.	Std.	Avg.	Min.	Max	Std.	Avg.	Min.	Max	Std.
1	50	524.61	524.61	524.61	524.61	0.00	524.61	524.61	524.61	0.00	0.00%	0.00%	0.00%	0.00
2	75	835.26	837.68	835.26	842.87	1.83	835.89	835.26	839.77	1.10	0.21%	0.00%	0.37%	0.72
3	100	826.14	826.14	826.14	826.14	0.00	826.26	826.14	827.39	0.37	-0.02%	0.00%	-0.15%	-0.37
4	150	1,028.42	1,031.74	1,028.42	1,035.34	1.33	1,030.05	1,028.42	1,033.73	1.52	0.16%	0.00%	0.16%	-0.19
5	199	1,291.29	1,302.62	1,293.12	1,312.35	4.10	1,295.37	1,291.45	1,308.56	3.44	0.56%	0.13%	0.29%	0.66
6	50	555.43	555.44	555.43	556.68	0.12	555.43	555.43	555.43	0.00	0.00%	0.00%	0.22%	0.12
7	75	909.68	913.10	909.68	925.32	2.53	909.70	909.68	911.76	0.21	0.37%	0.00%	1.49%	2.33
8	100	865.95	865.95	865.95	865.95	0.00	865.94	865.94	865.94	0.00	0.00%	0.00%	0.00%	0.00
9	150	1,162.55	1,167.05	1,163.31	1,171.16	1.58	1,163.67	1,162.55	1,164.24	0.51	0.29%	0.07%	0.59%	1.07
10	199	1,395.85	1,404.88	1,401.27	1,409.87	1.51	1,404.20	1,399.65	1,408.64	1.60	0.05%	0.12%	0.09%	-0.09
11	120	1,042.12	1,042.36	1,042.12	1,043.89	0.50	1,042.12	1,042.12	1,042.12	0.00	0.02%	0.00%	0.17%	0.50
12	100	819.56	819.56	819.56	819.56	0.00	819.56	819.56	819.56	0.00	0.00%	0.00%	0.00%	0.00
13	120	1,541.14	1,543.49	1,541.14	1,546.92	1.50	1,543.16	1,542.86	1,543.93	0.28	0.02%	-0.11%	0.19%	1.22
14	100	866.37	866.37	866.37	866.53	0.02	866.37	866.37	866.37	0.00	0.00%	0.00%	0.02%	0.02
Total/Average		13,664.37	13,700.98	13,672.36	13,747.18		13,682.32	13,670.02	13,712.05		0.14%	0.02%	0.26%	

Table B.3: Objective values for GPU-ALNS and PRA on the Christofides et al. instances.

Instance		GPU-ALNS				PRA				τ			
Number	Req.	Avg.	Min.	Max.	Std.	Avg.	Min.	Max	Std.	Avg.	Min.	Max	Std.
1	50	0.50	0.43	0.57	0.02	2.58	2.49	3.29	0.10	5.12	5.74	5.78	-0.08
2	75	4.58	3.07	6.83	0.83	4.43	4.28	5.19	0.11	0.97	1.39	0.76	0.72
3	100	3.97	2.84	5.59	0.46	6.81	6.58	7.52	0.18	1.72	2.32	1.35	0.28
4	150	8.26	5.01	13.27	1.45	11.71	11.39	12.51	0.20	1.42	2.27	0.94	1.25
5	199	10.62	6.04	18.32	2.40	14.94	14.44	16.54	0.34	1.41	2.39	0.90	2.07
6	50	0.74	0.51	0.88	0.09	2.72	2.65	3.13	0.06	3.66	5.19	3.55	0.02
7	75	4.66	2.90	6.43	0.73	4.94	4.78	5.28	0.10	1.06	1.65	0.82	0.63
8	100	3.42	2.50	4.48	0.35	7.16	6.89	7.80	0.18	2.09	2.76	1.74	0.18
9	150	7.72	4.32	12.06	1.75	13.08	12.58	14.33	0.30	1.69	2.91	1.19	1.45
10	199	10.00	6.88	13.75	1.56	16.40	15.72	18.17	0.38	1.64	2.28	1.32	1.18
11	120	4.13	1.33	5.05	1.07	8.61	8.34	9.66	0.21	2.09	6.26	1.91	0.86
12	100	1.18	0.88	1.54	0.20	6.45	6.21	7.05	0.15	5.48	7.10	4.59	0.06
13	120	6.86	4.03	11.10	2.05	10.27	9.98	11.13	0.20	1.50	2.48	1.00	1.85
14	100	1.64	1.15	2.37	0.27	6.62	6.40	7.13	0.13	4.05	5.56	3.01	0.14
Total/Average		68.27	41.91	102.23		116.72	112.73	128.75		1.71	2.69	1.26	

Table B.4: Runtimes for GPU-ALNS and PRA on the Christofides et al. instances.

Instance			GPU-ALNS				PRA				Diff			
Number	Req.	BKS	Avg.	Min.	Max.	Std.	Avg.	Min.	Max	Std.	Avg.	Min.	Max	Std.
1	560	16,212.83	16,293.05	16,224.30	16,424.70	67.78	16,404.78	16,222.41	16,587.79	148.96	-0.68%	0.01%	-0.98%	-81.18
2	600	14,499.04	14,665.31	14,637.70	14,719.90	17.09	14,632.94	14,616.50	14,649.92	7.90	0.22%	0.15%	0.48%	9.19
3	640	18,801.13	18,929.65	18,812.60	19,094.80	78.25	18,984.84	18,810.71	19,255.76	172.08	-0.29%	0.01%	-0.84%	-93.83
4	720	21,389.43	21,595.54	21,402.90	21,822.50	94.42	21,802.24	21,400.21	22,259.01	217.61	-0.95%	0.01%	-1.96%	-123.20
5	760	16,668.51	17,235.46	17,119.40	17,324.50	38.49	16,947.26	16,810.76	17,131.98	86.29	1.70%	1.84%	1.12%	-47.80
6	800	23,977.73	24,265.91	23,992.60	24,559.20	112.93	24,630.02	23,989.25	25,401.09	272.41	-1.48%	0.01%	-3.31%	-159.48
7	840	17,372.64	17,969.94	17,757.50	18,095.10	60.73	17,654.99	17,529.55	17,866.78	78.12	1.78%	1.30%	1.28%	-17.40
8	880	26,566.03	26,876.12	26,578.70	27,277.90	121.40	27,284.83	26,576.82	27,987.24	330.41	-1.50%	0.01%	-2.53%	-209.01
9	960	29,154.34	29,482.96	29,355.20	30,140.80	92.56	29,999.53	29,166.32	30,766.66	346.01	-1.72%	0.65%	-2.03%	-253.45
10	1040	31,742.64	32,133.99	31,944.00	32,430.30	123.85	32,681.52	31,752.23	33,730.35	377.86	-1.68%	0.60%	-3.85%	-254.01
11	1120	34,330.94	34,712.23	34,355.00	34,962.60	118.92	35,353.73	34,531.05	36,520.76	462.14	-1.81%	-0.51%	-4.27%	-343.22
12	1200	37,159.41	37,575.98	37,341.90	38,301.00	148.14	38,022.78	37,309.88	39,331.30	505.75	-1.18%	0.09%	-2.62%	-357.61
Total/Average			291,736.13	289,521.80	295,153.30		294,399.48	288,715.69	301,488.64		-0.90%	0.28%	-2.10%	

Table B.5: Objective values for GPU-ALNS and PRA on the Li et al. instances.

Instance		GPU-ALNS				PRA				τ			
Number	Req.	Avg.	Min.	Max.	Std.	Avg.	Min.	Max	Std.	Avg.	Min.	Max	Std.
1	560	37.10	28.68	53.06	6.03	72.34	71.25	73.30	0.52	1.95	2.48	1.38	5.51
2	600	35.60	27.52	54.92	6.25	82.21	80.61	84.66	0.72	2.31	2.93	1.54	5.54
3	640	41.34	33.95	57.44	6.16	108.36	107.23	109.70	0.53	2.62	3.16	1.91	5.63
4	720	47.83	40.84	77.85	6.71	148.71	146.60	149.95	0.59	3.11	3.59	1.93	6.12
5	760	46.23	34.31	71.47	9.51	146.46	144.58	150.30	1.04	3.17	4.21	2.10	8.46
6	800	55.92	46.61	86.28	9.58	197.82	193.12	201.16	1.22	3.54	4.14	2.33	8.36
7	840	56.99	39.95	80.34	11.20	184.03	180.01	188.12	1.41	3.23	4.51	2.34	9.79
8	880	65.57	54.49	101.50	10.81	256.78	253.14	259.54	1.27	3.92	4.65	2.56	9.53
9	960	79.32	62.01	122.00	15.18	328.22	322.23	336.71	2.31	4.14	5.20	2.76	12.87
10	1040	94.22	75.18	149.93	17.60	411.70	403.17	419.88	3.30	4.37	5.36	2.80	14.30
11	1120	112.52	88.72	173.03	20.01	526.92	516.07	534.44	3.62	4.68	5.82	3.09	16.39
12	1200	133.66	100.93	211.09	30.54	650.92	639.38	664.08	5.74	4.87	6.33	3.15	24.80
Total/Average		806.31	633.19	1,238.92		3,114.47	3,057.39	3,171.84		3.86	4.83	2.56	

Table B.6: Runtimes for GPU-ALNS and PRA on Li et al. instances.

Instance			GPU-ALNS				PRA				Diff			
Number	Req.	BKS	Avg.	Min.	Max.	Std.	Avg.	Min.	Max	Std.	Avg.	Min.	Max	Std.
1	560	16,212.83	16,293.05	16,224.30	16,424.70	67.78	16,415.20	16,223.14	16,587.79	125.97	-0.74%	0.01%	-0.98%	-58.19
2	600	14,499.04	14,665.31	14,637.70	14,719.90	17.09	14,631.47	14,612.85	14,649.92	7.85	0.23%	0.17%	0.48%	9.24
3	640	18,801.13	18,929.65	18,812.60	19,094.80	78.25	18,965.88	18,810.71	19,253.62	142.19	-0.19%	0.01%	-0.82%	-63.95
4	720	21,389.43	21,595.54	21,402.90	21,822.50	94.42	21,750.49	21,400.21	22,256.79	207.18	-0.71%	0.01%	-1.95%	-112.77
5	760	16,668.51	17,235.46	17,119.40	17,324.50	38.49	16,974.00	16,822.17	17,157.87	100.32	1.54%	1.77%	0.97%	-61.83
6	800	23,977.73	24,265.91	23,992.60	24,559.20	112.93	24,610.39	24,177.38	25,390.95	241.82	-1.40%	-0.76%	-3.28%	-128.89
7	840	17,372.64	17,969.94	17,757.50	18,095.10	60.73	17,667.50	17,511.16	17,870.76	96.54	1.71%	1.41%	1.26%	-35.82
8	880	26,566.03	26,876.12	26,578.70	27,277.90	121.40	27,216.47	26,765.68	27,519.52	267.30	-1.25%	-0.70%	-0.88%	-145.91
9	960	29,154.34	29,482.96	29,355.20	30,140.80	92.56	29,960.28	29,165.12	30,765.05	374.35	-1.59%	0.65%	-2.03%	-281.79
10	1040	31,742.64	32,133.99	31,944.00	32,430.30	123.85	32,595.26	31,753.42	33,542.47	377.82	-1.42%	0.60%	-3.32%	-253.97
11	1120	34,330.94	34,712.23	34,355.00	34,962.60	118.92	35,182.33	34,341.72	36,334.28	432.97	-1.34%	0.04%	-3.78%	-314.05
12	1200	37,159.41	37,575.98	37,341.90	38,301.00	148.14	37,923.49	37,191.80	39,318.05	495.61	-0.92%	0.40%	-2.59%	-347.46
Total/Average			291,736.13	289,521.80	295,153.30		293,892.75	288,775.38	300,647.08		-0.73%	0.26%	-1.83%	

Table B.7: Objective values for GPU-ALNS and PRA-8 on Li et al. instances.

Instance		GPU-ALNS				PRA				τ			
Number	Req.	Avg.	Min.	Max.	Std.	Avg.	Min.	Max	Std.	Avg.	Min.	Max	Std.
1	560	37.10	28.68	53.06	6.03	16.05	15.76	16.56	0.15	0.43	0.55	0.31	5.88
2	600	35.60	27.52	54.92	6.25	17.56	17.22	18.53	0.17	0.49	0.63	0.34	6.08
3	640	41.34	33.95	57.44	6.16	23.40	22.99	24.26	0.19	0.57	0.68	0.42	5.97
4	720	47.83	40.84	77.85	6.71	31.93	31.35	33.17	0.27	0.67	0.77	0.43	6.44
5	760	46.23	34.31	71.47	9.51	30.45	29.70	31.32	0.26	0.66	0.87	0.44	9.24
6	800	55.92	46.61	86.28	9.58	43.64	42.82	44.57	0.40	0.78	0.92	0.52	9.18
7	840	56.99	39.95	80.34	11.20	38.20	37.57	41.20	0.41	0.67	0.94	0.51	10.79
8	880	65.57	54.49	101.50	10.81	56.53	55.63	58.00	0.44	0.86	1.02	0.57	10.36
9	960	79.32	62.01	122.00	15.18	73.05	71.57	76.14	0.86	0.92	1.15	0.62	14.32
10	1040	94.22	75.18	149.93	17.60	96.84	94.14	101.52	1.34	1.03	1.25	0.68	16.26
11	1120	112.52	88.72	173.03	20.01	129.73	125.59	137.83	2.13	1.15	1.42	0.80	17.89
12	1200	133.66	100.93	211.09	30.54	170.91	162.15	177.08	2.80	1.28	1.61	0.84	27.74
Total/Average		806.31	633.19	1,238.92		728.28	706.48	760.18		0.90	1.12	0.61	

Table B.8: Runtimes for GPU-ALNS and PRA-8 on Li et al. instances.

Instance		GPU-ALNS				PRA-8				Diff			
Number	Req.	Avg.	Min.	Max.	Std.	Avg.	Min.	Max	Std.	Avg.	Min.	Max	Std.
1	1400	43,005.69	42,823.00	43,477.80	132.43	43,398.06	42,757.56	44,199.65	536.83	-0.91%	0.15%	-1.65%	-404.40
2	1600	40,936.85	40,272.60	41,350.30	229.44	40,613.76	40,050.44	41,301.49	249.17	0.79%	0.55%	0.12%	-19.73
3	1800	45,965.07	45,431.80	47,104.60	314.65	46,040.78	45,069.34	47,351.50	578.58	-0.16%	0.80%	-0.52%	-263.93
4	2000	46,469.19	45,657.50	47,924.30	415.36	46,375.01	45,271.70	47,782.87	639.21	0.20%	0.85%	0.30%	-223.85
5	2400	77,649.10	77,649.10	77,649.10	0.00	77,919.45	77,649.05	81,465.30	922.75	-0.35%	0.00%	-4.80%	-922.75
6	2800	42,373.94	41,803.30	42,988.50	191.19	41,494.72	40,845.03	41,893.93	177.84	2.10%	2.32%	2.58%	13.35
7	3200	114,962.31	112,637.00	118,037.00	1,422.50	116,051.34	112,111.22	119,098.04	1,430.76	-0.94%	0.47%	-0.89%	-8.26
8	3600	113,652.07	113,652.00	113,656.00	0.43	113,686.40	113,652.28	117,063.87	341.16	-0.03%	0.00%	-2.95%	-340.73
9	4000	122,317.82	121,047.00	123,802.00	648.67	120,745.84	119,236.06	122,153.61	588.30	1.29%	1.51%	1.34%	60.37
10	4500	53,099.79	52,633.70	53,618.50	221.01	54,294.41	53,701.46	54,955.25	299.59	-2.22%	-2.01%	-2.46%	-78.57
Total		700,431.84	693,607.00	709,608.10		700,619.76	690,344.15	717,265.51		-0.03%	0.47%	-1.07%	

Table B.9: Objective values for GPU-ALNS and PRA-8 on the BHS instances.

Instance		GPU-ALNS				PRA-8				τ			
Number	Req.	Avg.	Min.	Max.	Std.	Avg.	Min.	Max	Std.	Avg.	Min.	Max	Std.
1	1400	148.45	72.54	295.82	69.64	264.03	256.26	274.29	4.62	1.78	3.53	0.93	65.02
2	1600	279.62	122.04	385.15	80.72	523.86	510.81	545.02	7.49	1.87	4.19	1.42	73.23
3	1800	215.03	100.45	403.30	98.03	533.63	522.36	556.25	8.00	2.48	5.20	1.38	90.03
4	2000	260.51	110.20	497.76	118.12	791.29	769.56	829.53	11.13	3.04	6.98	1.67	106.98
5	2400	594.25	501.93	705.17	35.60	1,779.47	1,729.62	1,880.86	26.52	2.99	3.45	2.67	9.07
6	2800	390.47	169.30	490.52	69.73	1,222.46	1,195.54	1,307.51	19.80	3.13	7.06	2.67	49.93
7	3200	584.80	170.98	937.59	208.34	2,426.10	2,291.30	2,615.43	61.97	4.15	13.40	2.79	146.37
8	3600	1,156.73	872.49	1,772.71	293.77	5,083.92	4,990.02	5,232.80	54.50	4.40	5.72	2.95	239.27
9	4000	1,269.74	638.58	1,416.37	146.17	5,391.59	5,281.02	5,656.98	73.53	4.25	8.27	3.99	72.64
10	4500	580.30	295.64	868.11	110.50	4,012.31	3,902.98	4,176.14	65.35	6.91	13.20	4.81	45.15
Total		5,479.91	3,054.14	7,772.50		22,028.66	21,449.46	23,074.82		4.02	7.02	2.97	

Table B.10: Runtimes for GPU-ALNS and PRA-8 on the BHS instances.