International Journal of Parallel Programming (2021) 49:776–801
https://doi.org/10.1007/s10766-021-00714-1

High-Level Parallel Ant Colony Optimization with Algorithmic Skeletons

Breno A. de Melo Menezes[1] · **Nina Herrmann**[1] · **Herbert Kuchen**[1] · **Fernando Buarque de Lima Neto**[2]

Received: 13 November 2020 / Accepted: 1 April 2021 / Published online: 29 April 2021
© The Author(s) 2021

Abstract

Parallel implementations of swarm intelligence algorithms such as the ant colony optimization (ACO) have been widely used to shorten the execution time when solving complex optimization problems. When aiming for a GPU environment, developing efficient parallel versions of such algorithms using CUDA can be a difficult and error-prone task even for experienced programmers. To overcome this issue, the parallel programming model of *Algorithmic Skeletons* simplifies parallel programs by abstracting from low-level features. This is realized by defining common programming patterns (e.g. map, fold and zip) that later on will be converted to efficient parallel code. In this paper, we show how algorithmic skeletons formulated in the domain specific language *Musket* can cope with the development of a parallel implementation of ACO and how that compares to a low-level implementation. Our experimental results show that *Musket* suits the development of ACO. Besides making it easier for the programmer to deal with the parallelization aspects, *Musket* generates high performance code with similar execution times when compared to low-level implementations.

Keywords Algorithmic skeletons · Ant colony optimization · High performance computing

✉ Breno A. de Melo Menezes
breno.menezes@uni-muenster.de

Nina Herrmann
nina.herrmann@uni-muenster.de

Herbert Kuchen
kuchen@uni-muenster.de

Fernando Buarque de Lima Neto
fbln@ecomp.poli.br

[1] University of Münster, Leonardo-Campus 3, 48149 Münster, Germany

[2] University of Pernambuco, Rua Benfica ,455, 50720-001 Recife, Brazil

1 Introduction

Nature inspired metaheuristics have been widely used to solve complex optimization problems [1]. When tackling combinatorial problems, approaches using Ant Colony Optimization (ACO) have been widely exploited and can be often found in literature [2]. Initially proposed by Marco Dorigo in his Ph.D. thesis, it is inspired by the social behavior of ant colonies when searching for new sources of food and their ability of finding the shortest path between the colony and the food [3]. ACO was initially created to solve problems such as the Traveling Salesman Problem (TSP), achieving satisfactory results.

In ACO, the process to build a solution is done in several steps, including a certain number of probability calculations. The workload in this process is proportional to the size of the problem and the number of possibilities to be explored. Considering the TSP problem, the number of possible tours grows exponentially as the number of nodes in the graph increases. The same occurs for packing problems such as the Multidimensional Knapsack Problem (MKP) and the Bin Packing Problem (BPP), where the number of combinations rises quickly as more items have to be packed. In order to explore more of these possibilities in the search space, more ants are required in the colony and, therefore, the computational costs increase substantially. Knowing that a considerable part of the computations are performed during the solution construction phase (path construction or packing) and that the runtime increases when tackling a bigger instance of such problems with several ants, it is mandatory to speed up the program in order to run the algorithm in a reasonable amount of time without losing the quality of the solutions.

Parallel implementations of ACO have been introduced aiming for different high-performance hardware, such as multi-core CPUs and GPUs. Low-level frameworks such as OpenMP, MPI and CUDA provide many tools for programmers, assisting the development of such parallel versions of the algorithm. The tools provided by CUDA help programmers to develop parallel programs for Nvidia GPUs. Nevertheless, some expertise is necessary to generate high performance code. Programmers must be aware of data transfers, synchronization points, and many other issues that make the development of the program difficult and error prone.

Aiming to ease the development of such parallel algorithms, high-level parallelization tools provide means to profit from the use of high-performance hardware without the issues inherent to low-level programming. For example, some tools allow the use of predefined common programming patterns, known as algorithmic skeletons [4], here referred to as skeletons. They represent common operations, such as *map, zip*, and *reduce* and can be used in a program. Musket converts those patterns to parallel code. This way, the programmer's job is to translate the methods from the original algorithm into predefined operations which are translated to parallel code.

Muenster Skeleton Tool for High-Performance Code Generation (Musket) is an approach based on a Domain Specific Language (DSL) created to speed-up the development of parallel programs [5]. By using it, programmers are able to create code by first writing it in the DSL *Musket* and then converting the program into parallel CPU or GPU code. Created as a general purpose tool, *Musket* has already been applied and tested in several problems including metaheuristics, presenting promising performance results compared to other parallelization approaches [6].

In this paper, we investigate the use of *Musket* to create a parallel GPU version of ACO in order to understand and compare how it relates to a low-level implementation in terms of performance and development complexity. The identification of positive and negative sides of using the general purpose structures available in *Musket* may also serve as a base for the future development of the framework.

Our paper is organized as follows: The basics about ACO are displayed in Sect. 2. In Sect. 3 we give an overview of related work. The description of *Musket* and how it was applied in this work can be found in Sect. 4. Experiments are detailed in Sect. 6 together with the results. In Sect. 7 we put together the conclusions of this work and point out future work.

2 Ant Colony Optimization

ACO is a metaheuristic in which artificial ants cooperate among each other in order to solve complex discrete optimization problems. It was initially tailored to solve the TSP, therefore it requires some adaptations in order to be applied in other contexts, such as the BPP and MKP. Details about each adaptation and the GPU implementation are described in the following.

2.1 ACO Solving the Traveling Salesman Problem

In the case of TSP, the objective is to find the shortest tour in a graph, starting from a random node, visiting each node once and only once and coming back to the original node [7, 8]. In order to solve such a problem, each ant in the colony tries to create a tour and at the end of each iteration they share their success through pheromone deposits. More successful ants, the ones that generated shorter tours, deposit more pheromone on the visited edges. Pheromone is what will attract other ants in the following iteration, helping them to generate similar tours based on the success of ants from previous iterations.

The process described above can be divided into two steps which compose the execution of ACO, namely tour construction and pheromone deposit. During the tour construction, each ant must create a tour starting at a random node. The decision where to go next from the current node i is done in a probabilistic manner, taking into account the distance and the amount of pheromone between the current node and a candidate node. The probability is calculated as shown in Eq. 1.

$$p_{i,j} \quad = \quad \frac{[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}}{\sum_{l \epsilon N}[\tau_{i,l}]^{\alpha}[\eta_{i,l}]^{\beta}} \quad \forall j \epsilon N \tag{1}$$

where α and β are the parameters used to determine the influence of pheromone quantity and distance between the nodes over the probability, respectively. $\tau_{i,j}$ is the pheromone at the edge from node i to j, $\eta_{i,j}$ is $1/d_{i,j}$, where $d_{i,j}$ is the distance from node i to j. N is the set of unvisited nodes that can follow node i. $p_{i,j}$ is the probability that the ant goes from node i to j. The current node i and visited nodes have the probability equal to zero.

Once each ant has created its tour, the fitness of each ant will be equal to the total distance traveled. Afterwards, the pheromone update will take place. The first step is the pheromone evaporation where each edge loses a certain quantity of pheromone according to the following assignment (Eq. 2).

$$\tau_{i,j} := (1 - \rho) * \tau_{i,j} \tag{2}$$

where, $\rho \in [0..1]$ defines the evaporation rate and $\tau_{i,j}$ is the amount of pheromone between nodes i and j. ρ is used to control the amount of pheromone, enabling the algorithm to focus more on new trails. After the evaporation, it is time for each ant to deposit pheromone on the tour it has created, according to the following assignment (Eq. 3):

$$\tau_{i,j} := \Delta t + \tau_{i,j} \tag{3}$$

where $\Delta t = 1/q_k$, and q_k is the length of the round tour of ant k. By doing so, ants that generated shorter tours deposit more pheromone at visited edges than the ones that generate longer tours.

This process is repeated through as many iterations as needed. Although the pheromones are used to attract ants and help them create tours similar to previous successful ones, the probabilistic way of choosing the next step allows ants to create distinct paths and therefore generate diversity.

2.2 ACO Solving the Bin Packing Problem

The bin packing problem (BPP) is an NP-complete combinatorial optimization problem where a set of items with a given volume has to be packed into as few bins with a fixed capacity as possible. A certain quantity tells how many identical copies of an item are available. Being a combinatorial problem, the BPP can also be solved using ACO. The goals are similar but the process is a bit different because of the weight limitation imposed in the BPP and the method used to build a solution. In this section, these differences will be explained together with the approach used in this work to build a solution. The algorithm used here follows the approach of Levine and Ducatelle [9].

Differences between both problems emerge directly from the setup. When solving the BPP, the algorithm has to ensure continuously that the capacity of the currently considered bin is not violated, while the TSP has no such restrictions. Nevertheless, the basic structure of the algorithm remains unchanged. Modifications of the algorithm come from the fact that the order of inclusion is not important any more. Instead the grouping of items that fit together in one bin is relevant. Therefore, all items included in the current bin should be considered when calculating the probability of selecting the next item. Algorithm 1 illustrates the process of calculating τ and η.

Algorithm 1 Calculate Probabilities Pseudo-code

```
1:  $current\_bin = empty$
2:  $avail\_capacity = capacity$
3:  for $j \leftarrow items$ do
4:      $phero\_sum = 0.0$
5:      while $(j.quantity\text{--} > 0)$ do
6:          if $(j.volume <= avail\_capacity)$ then
7:              $avail\_capacity$ -= $j.volume$
8:              for $i \leftarrow current\_bin$ do
9:                  $phero\_sum$ += $pheromones[i][j]$
10:             $current\_bin = current\_bin \cup \{j\}$
11:         else
12:             $current\_bin = empty$
13:             $avail\_capacity = capacity$
14:     $\tau_j = phero\_sum/number\_of\_items\_in\_current\_bin$
15:     $\eta_j = j.volume^{\beta}$
```

Once τ and η are calculated for each item, the probability of choosing item j is calculated in the same way as explained previously. Items that have already been included (*item.volume* $= 0$) and items that would exceed the bin's capacity will be excluded and their probabilities are set to zero. If there are still items available but none fits into the current bin, a new bin is started. The process is repeated until all items have been packed and the fitness of the solution is equal to the number of bins used in this solution.

Using the same concept, the pheromone deposit process considers the fitness of one solution and re-visits each of the bins created to deposit pheromones. If items A, B and C are part of a bin created in a solution that resulted in a good fitness, the connections between these items will receive the same high amount of pheromone. This method of depositing pheromones, informs ants in future packing phases that packing A, B and C together helped creating a good solution and increases the chances of them being packed together again.

Other parts of the algorithm, such as the pheromone evaporation, are performed in the same way as mentioned previously.

2.3 ACO Solving the Multidimensional Knapsack Problem

The multidimensional knapsack problem (MKP) is as the previously discussed optimization problems NP-hard. In its original form, a set of items ($J = \{1, 2, ..., n\}$) is given together with a set of constraints ($I = \{1, 2, ..., m\}$). As each item has a value, the goal is to select a subset of J aiming to maximize profit respecting all given constraints. The approach used in this work to solve the MKP is inspired by the one proposed by Soh-Yee Lee and Yoon-Teck Bau [10]. In this approach, the probability of ant k adding a new item to the knapsack is calculated considering the partial solution ($\widetilde{S}_k(t)$) constructed until step t. It can be calculated using Eq. 4:

$$p_j^k(t) = \begin{cases} \dfrac{[\tau_j(t)]^\alpha \cdot [\eta_j(\widetilde{S}_k(t))]^\beta}{\sum_{l \in A_k(t))}[\tau_l]^\alpha[\eta_l(\widetilde{S}_k(t))]^\beta}, & \text{if } j \in A_k(t) \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where $p_j^k(t)$ is the probability of ant k choosing item j at step t, $A_k(t)$ is the set of available items for ant k at step t, $\tau_j(t)$ is the amount of pheromones assigned to item j at step t and $\eta_j(\widetilde{S}_k(t))$ is the heuristic factor which is calculated considering the value of item j and how it complies with the constraints as shown in Eq. 5:

$$\eta_j(\widetilde{S}_k(t)) = \frac{v_j}{\bar{\delta}_j(k,t)} \tag{5}$$

where v_j is the value of item j and $\bar{\delta}_j(k,t)$ is the average tightness considering all constraints and the actual state of the partial solution $\widetilde{S}_k(t)$. The tightness of an item j for a certain constraint i is defined by Eq. 6:

$$\delta_{ij}(k,t) = \frac{r_{ij}}{c_i - \sum_{l \in \widetilde{S}_k(t)} r_{il}} \tag{6}$$

where r_{ij} is the size of item j for constraint i and c_i is the capacity of the knapsack for dimension i. Summing the values for each dimension and dividing it by the number of constraint dimensions m gives the average tightness $\bar{\delta}_j(k,t)$.

Once all probabilities are calculated, the same probabilistic method as mentioned before is used to select the next item to be inserted in the knapsack. The process is repeated until the knapsack is full and none of the remaining items fits. Differently from the BPP, the solution proposed for the MKP can be smaller than the size of the initial set of objects.

In the approach used in this work, pheromone values are an association with one item and a construction step. Unlike the BPP, where groups of items are important, in MKP the order in which items are added has more value. The amount of pheromone to be deposited to item j by ant k is calculated using Eq. 7:

$$\Delta\tau_j^k(t) = \begin{cases} QL_k, & \text{if item } j \text{ is used by ant} k \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

where Q is constant (defined by $1/\sum_{j=1}^n p_j$) and L_k is the fitness of the solution constructed by ant k, which is equal to the sum of the values of all items inserted.

2.4 GPU-ACO

Targeting a GPU environment, we present here one possible implementation approach for parallelizing ACO using CUDA. The concepts explained in this section are applied in the same way for both problems investigated, adapting only to their peculiarities. The steps that compose the ACO algorithm as described above are quite simple and the general process makes the algorithm suitable for parallelization. Even so, extra care is required when dealing with the same steps in a parallel way.

The CUDA framework makes it possible to run sequential instructions on the CPU, while the computational intensive tasks can run on the GPU in parallel. In our approach, the whole algorithm runs on the GPU and, therefore, operations such as routing and pheromone updates are declared as CUDA kernels. One advantage about this approach is that no data transfer between host and device is necessary along the iterations. These data transfers are performed at the beginning (reading data and copying to GPU) and at the end of the execution (retrieving results from GPU to host). A general view of the proposed implementation used to solve the TSP is represented in Algorithm 2.

Algorithm 2 Pseudo-code GPU-ACO

```
1: $initialize\_ACO$
2: $copy\_data\_to\_Device();$
3: $initialize\_GPU$
4: $calculate\_distance\_kernel <<< n\_blocks, n\_threads >>> (...);$
5: $create\_random\_generators\_kernel <<< n\_blocks, n\_threads >>> (...);$
6: $n\_threads = warp\_size;$
7: $n\_blocks = n\_ants/warp\_size;$
8: $iterations = 0;$
9: while $(iterations{+}{+} < max\_iterations)$ do
10:     $solution\_construction\_kernel <<< n\_blocks, n\_threads >>> (...);$
11:     $pheromone\_evaporation\_kernel <<< n\_cities, n\_cities >>> (...);$
12:     $pheromone\_deposit\_kernel <<< n\_ants, n\_cities >>> (...);$
13: end while
14: $copy\_results\_to\_Host();$
15: $clear\_GPU();$
```

The first step of this implementation is the initialization of the structures that compose the problem. It includes reading the data from a file that contains the *x* and *y* coordinates of each city present in the graph of the TSP instance or the volumes and quantities of the items for the BPP. Afterwards, the data can be copied to the GPU and, already on the device side, other data structures necessary to run ACO can be created directly on the GPU, i.e. random number generators, distance matrix, pheromone matrix and route matrix. After everything is created and initialized properly, the algorithm can loop through its iterations and perform the solution constructions (tour or packing) and pheromone updates.

The solution construction is the first step in an iteration and also the most demanding task of ACO. In order to create a parallel version of it, the straight forward approach was chosen where the calculations necessary for calculating one solution (e.g. one route) are performed by one thread. The number of CUDA blocks will be determined by dividing the number of ants in the colony by the desired number of threads per block, meaning that it can be changed and adapted according to the necessity and capabilities of the hardware. The simplicity to implement this approach is one of its positive aspects. Once there is a sequential implementation of ACO, it is not too difficult to port it to CUDA using this approach.

After the solution-construction phase, the pheromone update is broken down into two different steps. The first one is the pheromone evaporation, in which each connection in the graph loses a certain amount of pheromone as explained before in Eq. 2. In our parallel implementation, one CUDA block is generated for each city and, inside this block, one thread is generated for each other city in order to decrease the amount of pheromone (Listing 1). For the BPP problem, the same approach is chosen with connections between each type of item available to be packed in bins.

```
__global__ void evaporation_kernel(double* c_phero) {
    int edge_index = blockIdx.x * blockDim.x + threadIdx.x;
    double RO = 0.5; //Evaporation rate = 50%

    if(blockIdx.x !=  threadIdx.x){  // no edge from city_i to itself
        c_phero[edge_index] = (1 - RO) * c_phero[edge_index];
    }
}
```

Listing 1: Evaporation Kernel (TSP)

The pheromone deposit phase starts with one CUDA block assigned to one ant in the colony. Each thread inside a block is responsible for updating an edge visited by the ant in the tour constructed previously. As the pheromone matrix is stored in the GPU's global memory and it is being updated by different threads, racing conditions might appear. In order to overcome that, the pheromone deposit is performed using CUDA's atomic operations, guaranteeing the integrity of the data without losing performance. For the BPP problem the pheromone value of items which are packed together in good solutions in one bin is increased. In this way items which are often packed together in good solutions have a connection with a higher pheromone value.

The process is repeated for a number n of iterations. At the end, the best results are copied to the host and the execution is ended. With such a simple approach it is already possible to achieve a considerable speedup when compared to sequential implementations.

3 Related Work

The use of ACO to solve combinatorial problems has been already deeply investigated. Levine and Ducatelle [9], used pure ACO to solve many instances to the BPP and Cutting Stock Problem (CSP). They state that exact methods work well for small instances of such problems, but would require too much time to solve bigger instances. In this situation, ACO comes as a convenient tool which is able to achieve good results in a reasonable amount of time. Furthermore, they also state the need of having a certain number of evaluations in order to achieve good results. For bigger instances, the necessity of having these evaluations together with the complexity of generating solutions result in a notable increase in the execution times. Although they do not apply any parallelization, it would come to hand when solving these bigger instances as the execution take already a considerable amount of time compared to the smaller problems.

Low-level parallel implementations of ACO have been already investigated in literature. The approaches include mainly the introduction of data parallelism into the code, like in the work of Uchida et al. [11]. Other works focus on improvements in the algorithm that would favor data parallelism. Cecilia et al. developed a new mechanism called I-Roulette in order to enhance parallelism during the path creation process. Furthermore, they introduce strategies for parallelization of the pheromone update process suitable for GPU architectures [12].

Another approach to reduce the execution time of the algorithm is to improve the parallelization itself. Instead of modifying the algorithm by introducing new mechanisms that would simplify the execution, the idea is to make better use of the hardware available. This can be achieved by organizing the structures needed by the algorithm in an optimal way and dividing the work in such a way that hardware use would be optimized. This can be done for example by introducing different levels of parallelism. In ACO, this can be done by parallelizing not only the job of each ant but also all internal calculations that belong to tour construction as demonstrated by Menezes et al. [13]. Also, in another work, the same authors investigate the use of atomic operations to enhance the process of updating the pheromone matrix [14]. The results indicate that different levels of parallelism can be useful according to the size of the problem and that the use of atomic operations can speedup the pheromone update phase.

Rieger et al. introduced *Musket*, a DSL for parallel programming [5]. Their idea is to offer a language with algorithmic skeletons built in and with a syntax similar to C++ in order to help programmers to write high performance distributed parallel programs without the need of expertise in low-level frameworks. High performance low-level code for different architectures (Multi-core CPUs, GPUs or clusters) is generated from Musket files. The authors point out the benefits of using a DSL compared to other high-level approaches. Also, among some examples, the Fish School Search (FSS) metaheuristic is used as a benchmark. Further analysis of FSS and *Musket* are done by Wrede et al. [6]. Both studies show the possibility of using such general purpose tools for the application in the metaheuristics field. The major criticism of using high-level frameworks has been the possible loss in performance. The present paper serves to evaluate the performance of a skeleton based implementation and the hand written implementation previously discussed [13, 14].

There are also many other approaches providing high-level parallel programming based on algorithmic skeletons, including Fastflow [15], SkePU [16], Muesli [17, 18], eSkel [19] and many others.

4 Musket

Musket is a DSL which enables programmers to develop parallel applications and generate optimized code without requiring knowledge about low-level parallel programming frameworks. For interested readers the code can be found in a public repository [20].

The syntax of *Musket* is based on C++ which is widely used for high performance computing. It was defined using the Xtext framework and it includes a parser and an editor that can be incorporated into Eclipse [21]. In this way, programmers can use helpful features such as syntax highlighting, code completion and validation. Creating parallel programs in *musket* is simplified in multiple ways. Common parallel programming structures are simplified by representing them as skeletons. Moreover, the division and allocation of data structures to distinct processes is done by the code generator which transforms Musket code to C++ with CUDA operations in case that GPU code is generated. Furthermore, it is totally abstracted from specifying the number of threads to be started. Those and other advantages become more apparent by illustrating an exemplary program (Listing 2).

A *musket* program is divided into four parts, namely *meta-information*, *data structure declaration*, *user function declaration*, and *main program declaration*. The *meta-information* block (lines 1–5) specifies for which type of hardware code should be generated. Firstly, the platform argument distinguished between a program for GPUs or CPUs. In case of stating multiple platforms multiple programs are created. For our application context only the GPU code generator is required. Afterwards, the number of processes, cores, and GPUs which shall be used are specified. Information about the targeted architecture is essential to generate a distribution for data structures which is efficient and to organize the parallel execution of the skeletons.

In *musket*, global data structures are declared before writing functions in the *data structure declaration* block (line 7). On the one hand, for each additional data structure type which is offered, the effort for the implementation rises. On the other hand, it is possible to include additional information for the data structures e.g. on the data distribution. Here, the elements of the array are distributed among the GPUs. Available distribution modes are *dist* for distributed, *copy* to make the whole data structure available for all processes, and local which is a specific form of copy where no global copy needs to be created. Similar to the specification of arrays, matrices can be created, which require the same parameters despite having two parameters to specify the number of rows and columns.

```
1  #config PLATFORM GPU CPU_MPMD
2  #config PROCESSES 4
3  #config CORES 24
4  #config GPUS 4
5  #config MODE release
6
7  array<int,16384,dist> a;
8
9  int double_values(int i){
10   return i + i;
11 }
12
13 main{
14   mkt::roi_start();
15   a.mapInPlace(double_values());
16   mkt::roi_end();
17 }
```

Listing 2: Musket Code Sample

The *user-function declaration* part (lines 9–11) includes custom user functions which will be passed as arguments to the skeletons calls and in the final program executed among the nodes and cores available. Inside user functions, programmers can make use of control structures, such as *if-else statements* and *for loops*, moreover, a selection of C++ library methods and external functions are available. Global structures declared in the previous section can be used either with a local index or a global index. Moreover, local variables can be created.

In the last part, similar to C programs, a main function is declared which defines the entrance of the program. In the example, lines 13–17 contain the *main program declaration*. There, general instructions of the program are listed using control structures, musket functions, and the parallelization instructions in the form of algorithmic skeletons. Musket functions are typically used functions for writing parallel programs which do not need to be executed in parallel, for example measuring the runtime and getting maximal and minimal values of data types. For example in line 14 the time measurement is started. Wrapping such functions relieves the user of the framework to think about target specific functions. In order to simplify parallelization, *musket* offers (different versions of) the Skeletons *fold*, *map*, *reduce* (which in contrast to fold only accepts a selection of commonly used reduction operators), *zip*, *gather*, *scatter* and *shift partition*. For *zip* and *map*, in place and index variants and their combinations are available. The given example doubles the value of every element of the array. The implementation of the ACO algorithm will show how those structures can also be used for more complex programs.

The written DSL code will then be transformed into low-level code. With each program generated (in case of multiple platforms) CMake Files and execution scripts are generated. The generated code is not meant to be further adjusted.

5 Our Proposal

ACO is a metaheuristic which is suitable for parallelization. Many tasks, such as the path or packing construction, are independent of each other as each ant is able to create its own solution without the interference from other ants. Even though, the task of creating a parallel version for it can be quite challenging. A few steps require extra care since reduction steps are executed before performing general calculations. For example, as shown in Eq. 1, the probability is calculated using the product of the amount of pheromone and the distance divided by the sum of all products. Furthermore, steps like the pheromone-update phase include changes which shall be performed by each ant in the colony over the pheromone matrix, which is a structure used by the whole colony. Therefore, the programmer must be careful to avoid race conditions and perform the right data transfers without harming performance.

In order to overcome such difficulties, *musket* is helpful. Generally, high-level frameworks have the advantage that the user does not need expertise in the specific area (in this context parallel programming). Musket DSL code is also more concise than e.g. C++ code with calls to a (skeleton) framework.

However, using a DSL also has its disadvantages. The developer of the DSL has to decided which functionalities are essential. Missing necessary functionalities limit the user of the DSL. While, in contrast, including too much functionalities increases the complexity of the code generator and confuses inexperienced users.

In order to evaluate the usability and practicability of a high-level framework for the exemplary case of the ACO algorithm, a *musket* program will be compared to a hand written program. The aspects of major interest are the performance of the programs and the complexity of the syntax and structure provided by the different approaches. As part of the comparison the creation process of a program implementing the ACO algorithm in *musket* will be described, to illustrate advantages and disadvantages of using a high-level framework.

5.1 Musket-ACO

In the following, interesting aspects of the high-level implementation of the ACO algorithm will be discussed. Of particular interest is how the single steps proposed in the abstract solution approach in Algorithm 2 are translated to one or multiple skeletons. Taking two problems enriches the analysis since similarities of the implementations independent from the problem can be highlighted.

The first difference between the algorithms is that for solving the TSP problem firstly the distances between all cities are calculated, and secondly, the 32 closest cities are determined. Those calculations speed up the route searching process of the algorithm since it favours close cities. The steps are executed once; therefore, they can be considered as data pre-processing. For the BPP and the MKP, no data pre-processing is necessary with the used data set.

In Listings 3, 4 and 5 extracts of the program are shown. Both programs nest those steps in a for loop to find a good solution. The number of iterations is left to the programmer. A program which stops when a sufficiently good result is achieved would also be feasible. Both programs start in line 1 with the most calculation intense step. For solving the TSP problem each ant calculates one possible route to visit all cities, for the BPP problem each ant packs bins until all items are packed. The parallelization of those methods is obvious as each ant can independently perform calculations. The return value of the two methods `route_kernel` and `packing_kernel` is different. The program solving the TSP problem writes the sequence of visited cities into an array and does not save the overall distance. The program solving the BPP problem returns the number of bins used. The functions require different parameters, however from these two problems it can be concluded that the first part of solving a problem with the ACO algorithm is feasible by mapping each result an ant produces to a result array.

```
1  antss.mapIndex(route_kernel());
2  d_fitness.mapIndexInPlace(update_delta_phero());
3  int new_bestroute = d_fitness.reduce(min);
4  antss.mapIndex(update_best_sequence(bestroute));
5  city.mapIndex(update_phero());
```

Listing 3: Skeleton Calls of TSP Program

```
1  d_fitness.mapIndexInPlace(packing_kernel(itemtypes, itemcount,
       BETA, bin_capacity));
2  int new_best_fitness = d_fitness.reduce(min);
3  if (new_best_fitness < best_fitness)
4          best_fitness = new_best_fitness;
5  d_phero.mapIndexInPlace(evaporation_kernel(itemtypes, EVA));
6  d_fitness.mapIndexInPlace(update_pheromones_kernel(itemtypes,
       itemcount, bin_capacity));
```

Listing 4: Skeletons Calls of BPP Program

```
1  d_ant_fitness.mapIndexInPlace(generate_solutions(n_objects,
       n_constr));
2  best_fitness = d_ant_fitness.reduce(max);
3  d_ant_fitness.mapIndexInPlace(update_best_solution(best_fitness,
       n_ants, n_objects));
4  d_pheromones.mapIndexInPlace(evaporate(evaporation));
5  d_ant_solutions.mapIndexInPlace(pheromone_deposit(n_objects,
       n_ants));
```

Listing 5: Skeletons Calls of MKP Program

Obviously, the program solving the TSP uses five skeleton calls while the program solving the BPP uses four skeleton calls. This shows that although the same algorithm is used the steps for finding a solution are adjusted dependent on the problem solved. In this case, the changes are caused due to the differences in the steps executed to find a good solution and measuring the fitness of a solution.

The differences in finding a good solution are that for the TSP problem good results are found by choosing from the 32 closest cities. For the BPP problem, good results are found by firstly packing the heaviest object and proceed by favoring as heavy objects. Both approaches are influenced by the pheromone and a random factor. The fitness of the TSP problem is measured by the overall distance while the fitness of the BPP problem is measured by the number of bins. Therefore, for the BPP problem the fitness is required during calculating a solution. Moreover, calculating the fitness for the BPP problem does not require a lot of memory space as it is sufficient to increase an integer for each bin used. In contrast, this is not the case for the TSP problem. Calculating the distance of the whole route is not necessary as any route with all cities is a valid solution. Moreover, calculating the fitness while finding a solution would result in additional read operations of the data structure storing the distance between different cities. Therefore, it would be required to load an array which stores the distance between all cities in the limited Graphic Processing Unit (GPU) memory. Instead the implementation loads an array with the closest 32 cities calculated in the data pre-processing which requires remarkably less memory. This is essential as less threads can be started with less memory available.

Therefore, the program solving the TSP problem uses another map skeleton to calculate the distance of each route found by ants (Listing 3 l.2). Depending on the distances, values for updating the pheromone are saved. This skeleton is not necessary for the the BPP problem as the number of bins used are already saved in the data structure `d_fitness`. Both programs continue by finding the minimum of all found solutions (Listing 3 l.3, Listing 4 l.2). For other problems this could also be the maximum (e.g. the Knapsack problem searches for the maximum value of packed items).

For the solution of the BPP problem, the following steps are to evaporate the previous pheromone, and to update the pheromone between items (Listing 4 l.5+6). For the TSP problem it was decided to evaporate the pheromone while updating the pheromone in the same skeleton call (Listing 4 l.5). No run-time differences could be found for having two or one skeleton call. Starting two skeletons allows more parallelization in the generated code as all entries can be changed at the same time, but cost more time as the start of an additional kernel requires time. Having one skeleton call saves time for starting kernels, but allows for less parallelization as not all entries are updated in parallel but dependent on the fitness all entries used by that solution are updated by one thread.

As a result, the *musket* implementation of ACO is very similar to the low-level approach. Adjustments have been made according to the necessities and restrictions imposed by Musket. For example, atomic operations applied in the pheromone update phase were not used in the high-level implementations because they are not implemented in *musket* until now. For this specific case, the lack of this feature has no impact in the final program since other approaches can be used to perform the same steps. One approach is to run the pheromone updates sequentially which would in theory consume more time. The second approach is to perform the updates in parallel without any lock controls. In practice, the sequential approach does not consume much more time and does not change the fact that the pheromone update consumes a minimal fraction of the total execution time. For the parallel approach, there is small chance of two distinct thread trying to update the same value and, if they do, the loss of information is minimal and has no major impact in the behavior of the algorithm.

Furthermore, parts of the code needed to be split into separate steps in order to fit Musket's structure. The implemented programs for the problems varied in the number of skeletons used, even so they are based on the same algorithm.

6 Our Case Study

The comparison between the parallel implementations of ACO used in this work can be done from different perspectives. As we propose the use of a high-level parallelization, the first point to analyse is the applicability of the tool. As mentioned in the previous section, the skeletons available in *musket* suffice to create high-level programs of ACO. Another aspect which can be discussed is the usability of *musket*. In *Musket*, 215 lines where needed to solve the TSP, 175 to solve the BPP and 168 to solve the MKP, compared to 374, 356 and 334 lines of the low-level implementation, respectively. For this comparison the code methods which read the data from files were excluded in both programs.

Furthermore, the cyclomatic complexity was measured for both programs [22]. We are aware that it does not reflect directly the effort to implement the programs, but it depicts evidence and is commonly used because of its simplicity. In this evaluation, the *musket* program has a cyclomatic complexity of 54 compared to a slightly higher complexity of 67 of the low-level implementation, meaning that it has less linearly independent paths, which makes it simpler and easier to maintain.

Most of the difference between the codes from both version comes from the *main* method, since kernel calls do not have to be written in *musket*. But most importantly it is abstracted from all data transfers, which consume several code lines in the low-level program. Moreover, it should be considered that the lines-of-code metric is admittedly debatable since it misses to evaluate how complex the written lines are. In addition to requiring only 57% and 49% of the lines of code compared to the low-level program, *musket* abstracts from complex decisions such as choosing the number of threads or moving data between the CPU and GPU. Therefore, it could be argued that creating a *musket* program does not only require less lines of code but additionally is written faster since the programmer is relieved from complex low-level decisions as in a low-level implementation using pure CUDA.

Another aspect to be evaluated in this work is the runtime. In order to test this aspect in both parallel implementations mentioned in this work, a NVIDIA GeForce RTX 2080 Ti accelerator containing 4352 CUDA cores, 11 GB memory and running CUDA 7.5 was used. Furthermore, we applied the implementations to two different problems to evaluate the behavior in different scenarios. The programs and results are publicly available [23].

6.1 TSP Experiments

The first experiments performed in this work include instances of TSP taken from popular repositories with different graph sizes [24, 25]. The selected TSP instances have different numbers of vertices so that the performance of low- and high-level ACO implementations can be evaluated on various problem sizes (Table 1). Unfortunately, the selection from the different repositories does not grow linearly in size. While some maps are very close in size, e.g. qa194 and d198, other maps have big differences, e.g. d1291 and pr2392.

For experimental purposes, each version was tested using different colony sizes (1024, 2048, 4096, and 8192 ants) for all TSP instances in order to simulate different levels of computational load. An important remark is that in this work, the fitness achieved is not relevant and the focus of the analysis is rather on the execution time. Since in essence both implementations represent the same algorithm and just vary in the parallelization approach and both achieve similar fitness values when using the same setup.

Aiming at a fair comparison between both approaches, the execution times registered in the experiments are denoted in seconds and represent the whole execution of the algorithm, including the initialization process and data transfers between host and GPU. The runtimes are the average of 30 runs excluding the first runs due to the warm up of the GPU. Figure 1 puts the values beside each other graphically for an easier comparison.

The results show that for the smaller problems, where less resources are needed, both implementations achieve very similar, almost identical, results. When more resources are needed, the low-level version tends to scale better and provide shorter execution times. The values for the last and biggest map are excluded in this graph since they impede the readability and will be discussed afterwards.

Intuitively, the graph underlines how close the runtime values from the low-level program and the *musket* program are. Also, as the TSP instance increases in size, the low-level program tends to be slightly faster, especially with higher values for the colony size. For example, when tackling the biggest problem, the low-level implementations is 0.4% faster with 1024 ants, 3.9% with 2048, 7.9% with 4096 and around 7.9% with 8192.

The execution times follow a similar pattern also for the biggest problem tackled (pr2392). Figure 2 illustrates this. This pattern appears in all test cases and is directly connected to how the programs are organized. In the low-level version, the operations are executed specifically for a certain task, against general purpose operations present in the *musket* program.

Table 1 TSPLIB

Instance	dj38	qa194	d198	lin318	pcb442	rat783	pr1002	pcb1173	d1291	pr2392
# Vertices	38	194	198	318	442	783	1002	1173	1291	2392

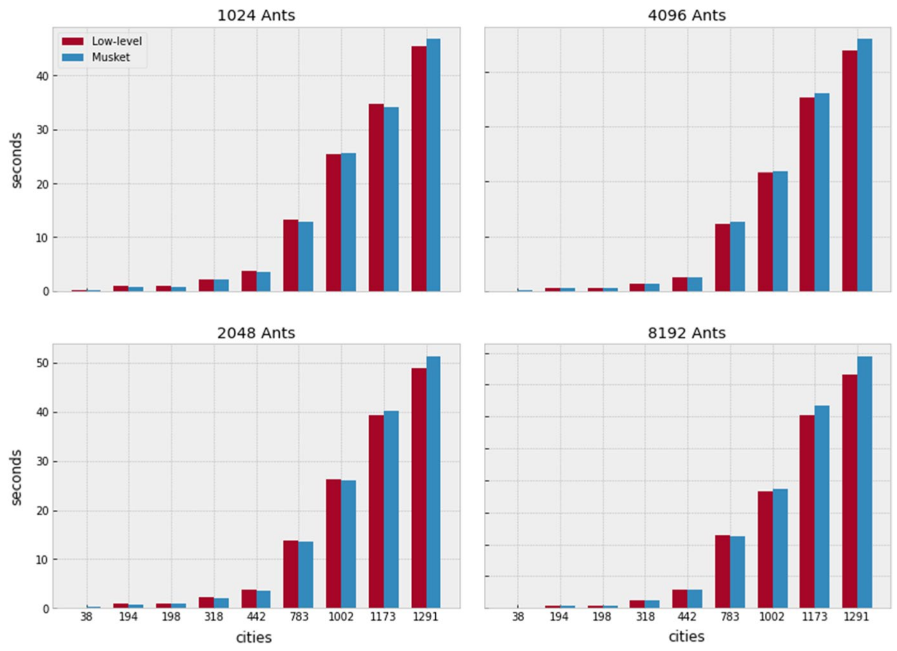


Fig. 1 TSP execution times comparison

Table 2 shows the overall execution times for both parallel ACO implementations considering the different problems and setups.

Another factor that affects the execution times is the setup regarding the number of blocks and block size. As CUDA does not accept more that 1024 threads per block for most architectures and some kernels used the block size equal to the number of cities, some balancing was necessary. Using more blocks with less threads each, enables CUDA to run the algorithm but it also adds some overhead, which explains the growth in the execution times when changing from 4096 to 8192 ants. Adaptations to solve this matter are easily done in the low-level program which generates a program with a better configuration of numbers of blocks and threads,

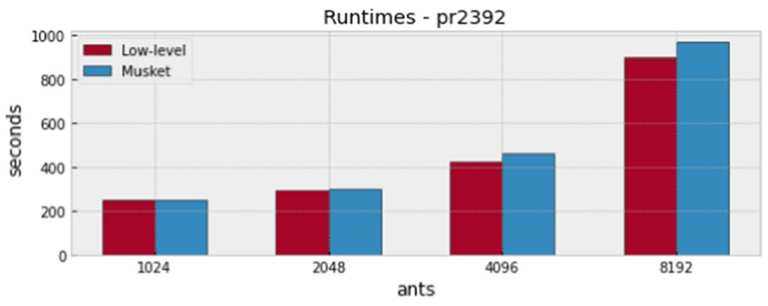


Fig. 2 Execution times for pr2392

which fits the problem, compared to high-level approaches. Of course the kernel instructions can be changed in order to optimize the execution time, but for comparison purposes only the numbers of blocks and threads were changed. In order to investigate further the runtime differences between both implementations, the runtime of single kernels was isolated and investigated separately.

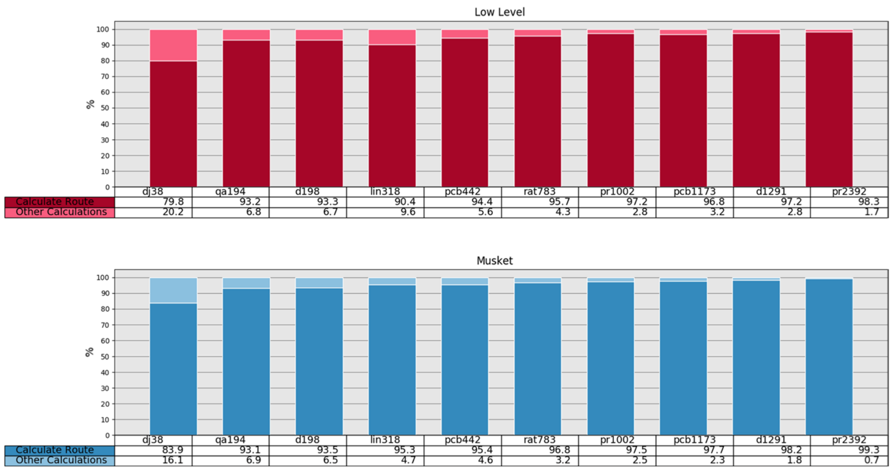
The most important and time consuming step in ACO is the tour construction. Performed many times during the execution, it is affected by the colony size and also the graph size. Therefore, special attention was given to the time spent by each parallel implementation on creating routes. Figure 3 shows for the example of 1024 ants the proportional amount of time spent on each kernel by the *musket* implementation and the low-level implementation. Obviously, even for the smallest map the route-construction kernel requires for both programs by far most of the runtime. Therefore, we investigated in the calculations of the route.

In order to compare the two implementations regarding the tour construction step, we have investigated the average time spent in the tour construction per iteration as shown in Fig. 4. The graphs show similar results to the total execution times mentioned previously and it is no wonder since the tour construction is the reason for a great part of the general execution times shown before.

The kernels responsible for executing the other steps of the algorithm have almost equal execution times for both implementations. Furthermore, they represent a small, almost irrelevant, part of the whole execution time. Therefore no deeper analysis becomes necessary.

Table 2 Execution times comparison: low-level vs. *musket*

	GeForce RTX 2080 Ti							
	1024		2048		4096		8192	
Problem	LL	*Musket*	LL	*Musket*	LL	*Musket*	LL	*Musket*
dji38	0.103	0.186	0.106	0.189	0.107	0.192	0.158	0.199
cat194	0.890	0.782	0.896	0.797	1.022	0.938	1.742	1.789
d198	0.906	0.852	0.914	0.861	1.037	1.001	1.784	1.804
lin318	2.217	2.103	2.250	2.103	2.726	2.665	6.175	6.261
pcb442	3.711	3.464	3.783	3.474	5.022	4.961	14.318	14.224
rat783	13.365	12.870	13.766	13.621	24.651	25.191	57.208	56.407
pr1002	25.409	25.529	26.307	26.116	43.194	43.657	91.539	93.51
pcb1173	34.680	34.201	39.201	40.23	70.605	72.321	151.416	158.965
d1291	45.37	46.744	48.832	51.306	87.808	91.888	182.741	197.17
pr2392	251.412	252.648	292.913	304.26	428.534	462.256	899.670	969.704



Low Level

	dj38	qa194	d198	lin318	pcb442	rat783	pr1002	pcb1173	d1291	pr2392
Calculate Route	79.8	93.2	93.3	90.4	94.4	95.7	97.2	96.8	97.2	98.3
Other Calculations	20.2	6.8	6.7	9.6	5.6	4.3	2.8	3.2	2.8	1.7

Musket

	dj38	qa194	d198	lin318	pcb442	rat783	pr1002	pcb1173	d1291	pr2392
Calculate Route	83.9	93.1	93.5	95.3	95.4	96.8	97.5	97.7	98.2	99.3
Other Calculations	16.1	6.9	6.5	4.7	4.6	3.2	2.5	2.3	1.8	0.7

Fig. 3 Proportional execution times of route calculation

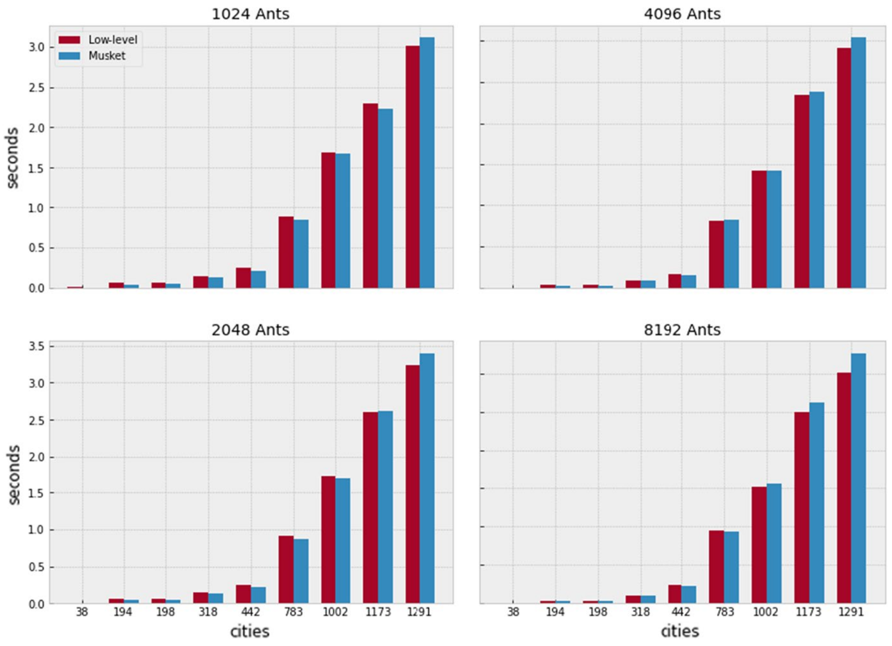


Fig. 4 Execution times comparison for the tour construction kernel

6.2 BPP Experiments

The BPP instances used in this work were extracted from different sources from literature [26, 27]. They vary not only in the number of items to be packed but also in

Table 3 BPP

Instance	0	1	2	3	4	5
# Item types	50	166	358	522	712	915
Total # items	60	201	402	600	801	1002
Bin capacity	1000	2456	7552	16,256	31,616	65,088

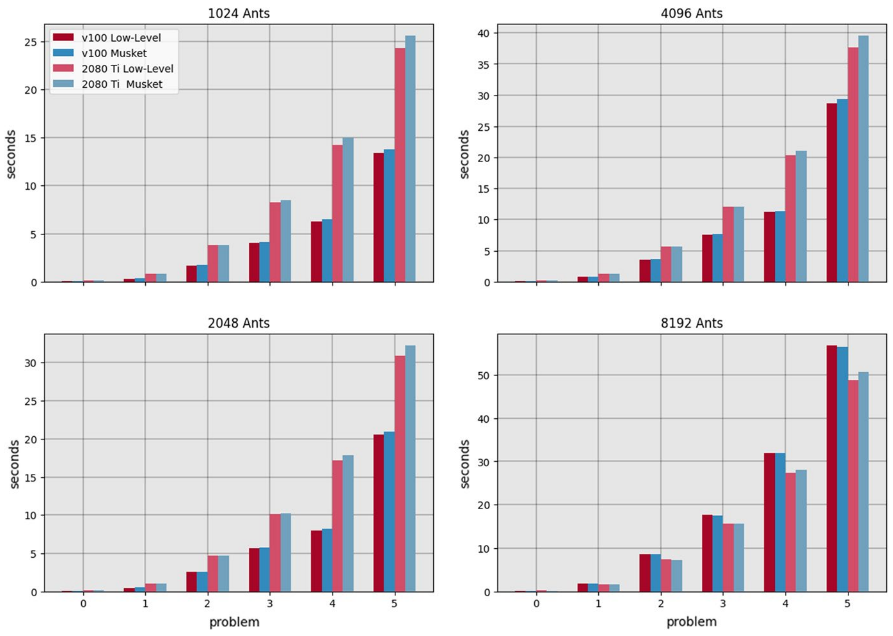


Fig. 5 BPP execution times

the degree of difficulty to solve. Details about each instance can be seen in Table 3.

In order to evaluate the performance of both implementations and compare it with the results of the TSP experiments, similar setups were used in the BPP experiments. The same numbers are used for the colony sizes. Furthermore, the BPP experiments were also executed in a second GPU, the NVIDIA Tesla V100, so that the programs could be evaluated using devices with different configurations. The average execution times for each of the BPP instances are displayed in Fig. 5.

The results show that the low-level implementation has slightly shorter execution times for most of the test cases and for both GPUs. Also, both implementations presented a reduction in the execution times in a similar degree when using the Tesla V100. Similarly to the TSP results, the tendency of having slightly shorter execution times when running the low-level program is more pronounced when observing the results from the bigger instances with more items to be packed. Table 4 includes the execution times and also the percentage comparison from the low-level to the musket program, both using the GeForce RTX 2080 Ti.

It is also interesting to observe that the execution times for both implementations scale differently as the colony gets bigger in the BPP experiments when compared to the TSP experiments. The speedup rates for the BPP experiments remain quite stable, independently of the colony sizes. This might be explained by the fact that the data is structured differently for the bin packing problem when compared to the TSP. In the BPP, most of the data are integer values, which occupy less space and favour the internal calculations. Furthermore, the data structures that store the information regarding the items to be packed are also smaller due to the repetition of items, impacting directly the space required by each thread and reducing the number of probability calculations during the packing phase. Having less demanding threads to execute makes it possible for the GPU to run more threads simultaneously, meaning that doubling the colony size will not necessarily double the execution time.

In order to investigate further, we measured also the time spent during the packing phase of the algorithm. Figure 6 shows the values for Problem 3.

6.3 MKP Experiments

The MKP instances used in this work were extracted from the OR-Library, first described in [28]. The library offers a couple of instances of the MKP with different numbers of objects and different numbers of constraints. Details about each instance can be seen in Table 5.

For these experiments the same ACO setup was used as in the experiments previously mentioned. Regarding the hardware used, a third GPU was used in the benchmarks, namely Quadro RTX 6000. By doing so, the evaluation of both implementations was extended and the performance could be compared also for different scenarios since the GPUs have different specs. A comparison of the execution times for the low-level and musket implementations is displayed in Fig. 7.

The execution time graph shows very similar performances for both implementations when using the same GPU. The differences are in the magnitude of hundredths of a second and did not follow any pattern. The size of the MKP instances and the overall short execution times show that the problems did not pose a greater challenge to the programs which makes it difficult to identify where the differences

Table 4 BPP execution times: proportional comparison

P.	GeForce RTX 2080 Ti											
	1024			2048			4096			8192		
	LL	*Musket*	%	LL	*Musket*	%	LL	*Musket*	%	LL	*Musket*	%
0	0.14	0.14	0.99	0.15	0.16	1.02	0.17	0.18	1.08	0.21	0.21	1.00
1	0.81	0.82	1.02	1.01	1.05	1.05	1.21	1.29	1.06	1.59	1.69	1.07
2	3.82	3.87	1.01	4.70	4.72	1.00	5.65	5.68	1.01	7.40	7.38	1.00
3	8.26	8.49	1.03	10.11	10.25	1.01	12.06	12.15	1.01	15.64	15.75	1.01
4	14.21	15.07	1.06	17.23	17.93	1.04	20.32	21.14	1.04	27.35	28.16	1.03
5	24.24	25.60	1.06	30.94	32.28	1.04	37.70	39.60	1.05	48.85	50.82	1.04

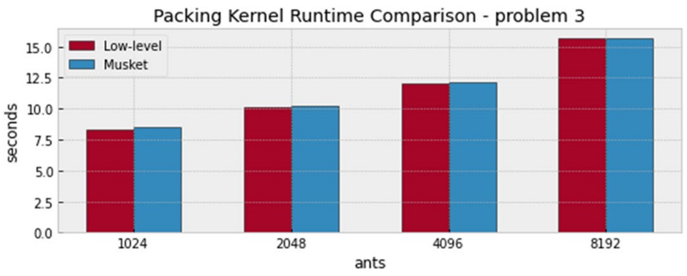


Fig. 6 Packing kernel execution times–problem 3

Table 5 MKP problems

Instance	1	2	3	4	5	6	7
Objects	6	10	15	20	28	39	50
Constraints	10	10	10	10	10	5	5
Optimal	3800	8706.1	4015	6120	12,400	10,618	16,537

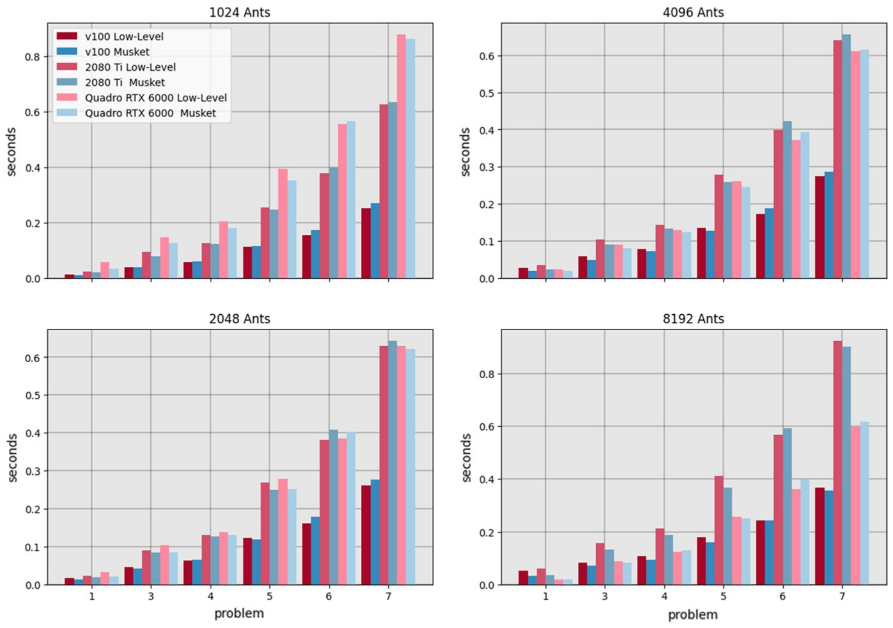


Fig. 7 MKP execution times–GeForce RTX 2080 Ti, Tesla V100 and Quadro RTX 6000

come from. In addition to that, the solution sizes differ, which adds an uncertainty factor to how long a step to build a solution should take.

Performance differences between implementations using different GPUs can be also be seen in Fig. 7. It is interesting to observe that the runtimes from the Musket implementation scale in the same proportion as those from the low-level implementation for all colony sizes.

Figure 8 puts the execution times from the experiments using the Quadro RTX 6000 GPU into another perspective. The graph shows how the execution times are similar for both implementations and how they remain unchanged even when the colony size is doubled. For the same problem, the execution times grow only when the colony size is increased to 8192 ants. This behavior shows how similar the workload generated by the implementation using Musket is when compared to the low-level implementation. For both, the workload generated was not enough to fully occupy the GPU even when the colony size was doubled. It took 8192 ants to generate a workload that would fully occupy the Quadro RTX 6000 GPU and its 4608 cores.

As an overall result, plenty of similarities between the execution times of both implementations investigated in this work can be observed for all three problems. They show how the use of *musket* can simplify the development of parallel programs, as the use of general purpose skeletons provided out of the box suffices to develop a parallel version of ACO in fewer lines of code and on a much lower complexity level when compared to the low-level CUDA implementation without impairing the performance.

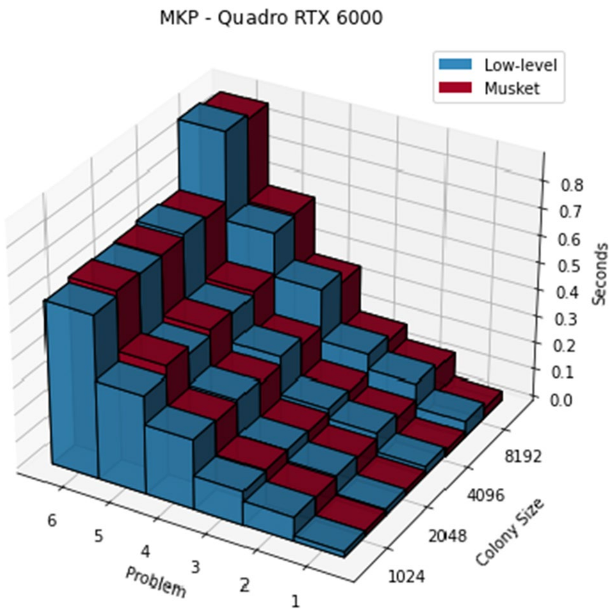


Fig. 8 MKP execution times–Quadro RTX 6000

7 Conclusion

The use of a high-level parallelization approach can be of great help for programmers aiming to run swarm intelligence algorithms on high-performance hardware, such as graphical processing units. In this work we have evaluated *musket* as an approach for the parallelization of the ACO algorithm in order to identify the pros and cons of using such a tool regarding the development aspect and also the performance aspect when compared to a low-level implementation.

Considering the development aspect, in its actual state, the skeletons embedded in *musket* provide enough features for the development of parallel ACO programs. Furthermore, the experiments have shown that *musket* offered some advantages in terms of simplicity, requiring less skills to develop a high performance parallel version of ACO. Not only less lines of code were necessary, but it is also much simpler to program without having the concerns that regard the parallel aspects of programming a CUDA-based version of the code, such as data initialization, data transfers, and the allocation of blocks and threads.

In terms of runtime, the ACO version implemented using *musket* achieved reasonably good execution times compared to the low-level CUDA based implementation for both problems investigated here. As algorithm enhancements and handcrafted adaptations to a certain problem instance were left aside, both implementations were evaluated in equal conditions. By doing so, we were able to observe how the *musket* implementation reacts in scenarios where the colony size was increased and more resources were needed. In these experiments, a bit more overhead was generated but nothing that would compromise *musket*'s overall performance. At the end there is a positive balance, as the results showed that we were able to simplify the implementation phase without compromising the runtime of the experiments.

The ACO version used in this work was idealized to be a simple implementation for a single GPU environment. Many optimizations could be introduced in order to enhance the performance of the algorithm e.g. the usage of shared memory. Also, if the goal is to run in a new environment such as multiple GPUs or multiple computational nodes with multiple GPUs, more complex changes are necessary, which can be tricky even for experienced programmers. In this aspect, *musket* has the advantage that the same program can be used to generate code for different architectures once there is a code generator for it.

Future works include the evaluation on different hardware, such as multiple GPUs on one computational node and also a cluster environment with many nodes and many GPUs per node. Furthermore, we want to further investigate the possibility to enhance *musket* to provide metaheuristic-specific skeletons in order to make better use of the hardware and reduce even more the execution times for such problems.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Talbi, E-G.: *Metaheuristics*. Wiley, Hoboken, NJ (2009)
2. Kallioras, N.A., Kepaptsoglou, K., Lagaros, N.D.: Transit stop inspection and maintenance scheduling: A GPU accelerated metaheuristics approach. Transp. Res. Part C Emerg. Technol., **55**, 246–260 (2015)
3. Dorigo, M.: Optimization, Learning and Natural Algorithms[in Italian]. PhD thesis, Dipartimentodi Elettronica, Politecnico di Milano, Milan (1992)
4. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Pitman London (1989)
5. Rieger, C., Wrede, F., Kuchen, H.: Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons. Proc. ACM Symp. Appl. Comput. Part **F147772**, 1534–1543 (2019)
6. Wrede, F., Rieger, C., Kuchen, H.: Generation of high-performance code based on a domain-specific language for algorithmic skeletons. J. Supercomput. 0123456789 (2019)
7. Dorigo, M., Birattari, M., Stutzle, T.: Ant colony optimization. IEEE Comput. Intell. Mag. **1**(4), 28–39 (2006)
8. Dorigo, M., Caro, G.D.: Ant colony optimization: a new meta-heuristic (1999)
9. Levine, J., Ducatelle, F.: Ant colony optimization and local search for bin packing and cutting stock problems. J. Oper. Res. Soc. **55**(7), 705–716 (2004)
10. Lee, S.Y., Bau, Y.-T.: An ant colony optimization approach for solving the Multidimensional Knapsack Problem. In: 2012 International Conference on Computer & Information Science (ICCIS), pp. 441–446. IEEE (2012)
11. Uchida, A., Ito, Y., Nakano, K.: Accelerating ant colony optimisation for the travelling salesman problem on the GPU. Int. J. Parallel Emergent Distrib. Syst. **29**(4), 401–420 (2014)
12. Cecilia, J.M., García, J.M., Nisbet, A., Amos, M., Ujaldón, M.: Enhancing data parallelism for ant colony optimization on GPUs. J. Parallel Distrib. Comput. **73**(1), 42–51 (2013)
13. Menezes, B.A., Kuchen, H., Neto, H.A.A., de Lima Neto, F.B.: Parallelization strategies for GPU-based ant colony optimization solving the traveling salesman problem. In: 2019 IEEE Congress on Evolutionary Computation, CEC 2019 - Proceedings, pp. 3094–3101 (2019)
14. Menezes, B.A.D.M., Pessoa, L.F.D.A., Kuchen, H., Neto, F.B.D.L.: Parallelization strategies for GPU- ased ant colony optimization applied to TSP. Adv. Parallel Comput., **36**, 321–330 (2020)
15. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. Programming Multi-Core and Many-Core Computing Systems, Parallel and Distributed Computing (2017)
16. Öhberg, T., Ernstsson, A., Kessler, C.: Hybrid cpu-gpu execution support in the skeleton programming framework skepu. J. Supercomput. **76**(7), 5038–5056 (2020)
17. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-gpu systems and clusters. Int. J. High Perform. Comput. Networking **7**(2), 129–138 (2012)
18. Ernsting, S., Kuchen, H.: Data parallel algorithmic skeletons with accelerator support. Int. J. Parallel Prog. **45**(2), 283–299 (2017)
19. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eskel. In: European Conference on Parallel Processing, pp. 761–770. Springer, Berlin (2005)
20. Menezes, B.A.D.M., Herrmann, N.: Musket repository. https://github.com/wwu-pi/musket_dsl (2020)
21. The Eclipse Foundation. Xtext documentation. https://eclipse.org/Xtext/documentation/ (2020)
22. Riguzzi, F.: A survey of software metrics. Technical report (1996)
23. Menezes, B.A.D.M., Herrmann, N.: Ant colony optimization project. https://github.com/brenoamm/ant-colony-optimization-project (2021). Accessed 24 March 2021

24. University of Waterloo. National traveling salesman problems. http://www.math.uwaterloo.ca/tsp/world/countries.html. Accessed 14 March 2018
25. Heidelberg University. Discrete and combinatorial optimization. https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/XML-TSPLIB/instances/. Accessed 14 March 2018
26. Delorme, M., Iori, M., Martello, S.: Bin packing and cutting stock problems: mathematical models and exact algorithms. Eur. J. Oper. Res. **255**, 1–20 (2016)
27. Falkenauer, E.: A hybrid grouping genetic algorithm for bin packing. J. Heuristics **2**, 5–30 (1996)
28. Beasley, J.E.: OR-Library: distributing test problems by electronic mail. J. Oper. Res. Soc. pp. 1069–1072 (1990)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.