

# FANG: Fast and Efficient Successor-State Generation for Heuristic Optimization on GPUs<sup>\*</sup>

Marcel Köster<sup>1</sup>, Julian Groß<sup>1</sup>, and Antonio Krüger<sup>1</sup>

Saarland Informatics Campus, Campus D3.2, 66123 Saarbrücken, Germany  
{firstname.lastname}@dfki.de

**Abstract.** Many optimization problems (especially nonsmooth ones) are typically solved by genetic, evolutionary, or metaheuristic-based algorithms. However, these genetic approaches and other related papers typically assume the existence of a neighborhood or successor-state function  $N(x)$ , where  $x$  is a candidate state. The implementation of such a function can become arbitrarily complex in the field of combinatorial optimization. Many  $N(x)$  functions for a huge variety of different domain-specific problems have been developed in the past to solve this general problem. However, it has always been a great challenge to port or realize these functions on a massively-parallel architecture like a Graphics Processing Unit (GPU). We present a GPU-based method called *FANG* that implements a generic and reusable  $N(x)$  for arbitrary domains in the field of combinatorial optimization. It can be customized to satisfy domain-specific requirements and leverages the underlying hardware in a fast and efficient way by construction. Moreover, our method has a high scalability with respect to the number of input states and the complexity of a single state. Measurements show significant performance improvements compared to traditional exploration approaches leveraging the CPU on our evaluation scenarios.

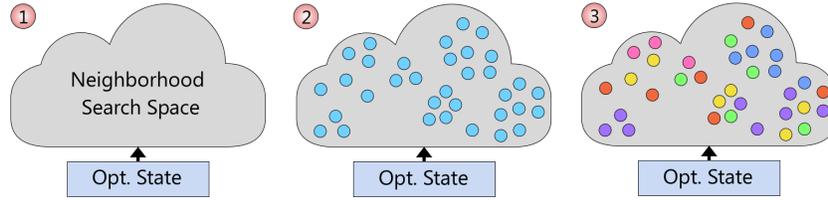
**Keywords:** Heuristic search, combinatorial optimization, successor-state generation, neighborhood exploration, massively-parallel processing, graphics processing units, GPUs

## 1 Introduction

There are many different optimization algorithms for a huge variety of problems. Every problem can be assigned to a category and different methods are used to solve a problem—even in the scope of a single problem category. Convex optimization problems have the advantage that a local optimum is equal to the global optimum. However, so called *nonsmooth optimization problems (NSPs)* are typically assumed to be non-convex. Furthermore, it is not possible in general to determine the direction into which an optimizer has to continue from a certain point in order to find a better solution. Such problems are often optimized by different genetic, evolutionary or in general heuristic-based algorithms.

---

<sup>\*</sup> PREPRINT



**Fig. 1.** The first image shows the abstract and large neighborhood search space around a single optimization state (1). We have to identify possible (with respect to constraints) and beneficial (with respect to the cost function) neighboring successor states within this very large search space (2). Since it is typically not possible to explore all potential successors, the available possibilities are rated according to heuristics (3). Once the different ratings are available, a selection strategy chooses possibly beneficial successor states out of this set.

In order to apply existing optimization algorithm for heuristic search, a local neighborhood or successor-state function is required to enumerate all states nearby (Figure 1). Without such a function, possible candidate states of a single source state cannot be determined. This functionality is typically referred to by  $N(x)$ , where  $x$  is a candidate state in the scope of the optimizer [4]. However,  $N(x)$  can become arbitrarily complex and difficult to implement, even when focusing on the pure algorithms and logics. If we consider an implementation on a high-performance massively-parallel processor like a *Graphics Processing Unit (GPU)*, a default CPU-based implementation of  $N(x)$  has to be manually adapted and tuned to the target hardware. Since many heuristic optimizers track multiple candidate states at the same time, a simple way to parallelize a successor-state generation would be to simply invoke a sequential existing  $N(x)$  function for every state in parallel. However, this typically sacrifices a large amount of performance since it often leads to non-optimal memory-access patterns and does not pay attention to other hardware-specific peculiarities like *single-instruction multiple threads (SIMT)* units<sup>1</sup>. Hence, developers have to adjust their implementation to these hardware characteristics, which is error-prone and time consuming [6]. Consequently, an arbitrary  $N(x)$  function on a GPU, for instance, becomes even more sophisticated to implement.

In this paper we present a new method to implement domain-independent  $N(x)$  functions for combinatorial heuristic optimization on massively-parallel architectures using SIMT units. It is designed to achieve a high utilization of the available processing power and scales well with the number of variables, possibilities per variable and the number of states. This allows for an application to large optimization problems that significantly benefit from parallel processing. Furthermore, it enables the design and the implementation of GPU-based optimizers that can perform nearly all required steps in parallel without high communication overhead to the CPU (the transfer of state information during the optimization process, for example). In order to guide  $N(x)$  to enumerate possibly interesting states, we offer so called *local heuristics* that guide the gen-

<sup>1</sup> We will refer to a single SIMT unit as warp in the scope of this paper.

eration of successors. Since our method only determines the direct neighboring or potential successor states for a given set of states, it can be easily integrated into any existing optimization algorithm like Tabu Search. We demonstrate several use cases in the evaluation section and show the significant improvements in terms of performance and scalability.

In the remainder of this paper, we focus on related work from the field of parallel neighborhood exploration in the context of heuristic optimization. We introduce our method in section 3 and present a detailed explanation how the general algorithm works (subsection 3.1, subsection 3.2) and can be implemented on GPUs (subsection 3.4). The evaluation (section 4) shows different optimization problems that were optimized using our  $N(x)$  implementation.

## 2 Related Work

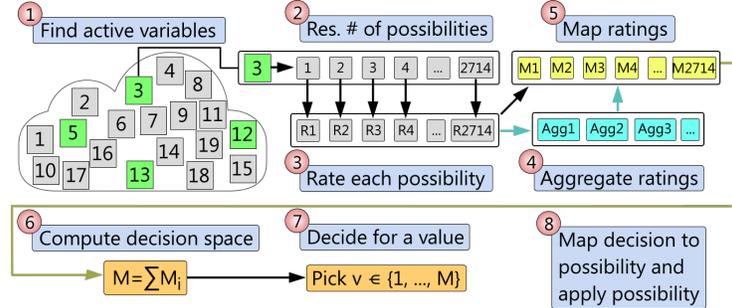
There has been a lot of work on using GPUs for solving optimization problems in general. We focus on a selection of papers that involve  $N(x)$  realizations in favor of purely parallelized optimization algorithms.

Campeotto et al. [3] focus on parallelizing constraint solving using GPUs. They use a hybrid design in which they switch between CPU and GPU in every solver step: The actual constraint propagation and consistency checks are executed on the GPU, whereas the main solver runs on the CPU. This can be seen as a parallel evaluation of several potential states in the scope of an abstract  $N(x)$  function, which is primarily evaluated on the CPU side. For general work on local search and constraint programming we refer the interested reader to [4].

A follow-up paper by Campeotto et al. [2] goes into more detail about the neighborhood processing functionality. The CPU selects subsets of variables to explore and copies the required information to the GPU. Afterwards, the GPU can process the different sets and explore the resulting states in parallel. They use different strategies (called *local search strategies*) to select these potentially interesting subsets. This is very related to our approach; however, we perform the whole  $N(x)$  evaluation in parallel on the GPU without the need for additional CPU communication.

Munawar et al. [14] investigated solving of MAX-SAT problems on GPUs using efficient genetic algorithms and local search. The neighborhood exploration is based on a 4D virtual grid which yields four neighbor possibilities (2D) for every individual and four possibilities for every population (2D) in the scope of the genetic algorithm. This can be directly mapped to GPUs using multiple thread groups that process all possibilities in parallel. This is very similar to our approach: the whole neighborhood exploration is performed on the GPU. In contrast to their  $N(x)$  implementation, our method can work with arbitrary local-search criteria and is not tied to a particular mathematical optimization model (MAX-SAT in this case).

A combination of the previously presented approaches is the one by Abdelkafi et al. [1]. They leverage OpenCL to evaluate the neighborhood in parallel by assigning different threads to different neighbors. Afterwards, they evaluate each neighbor sequentially in the scope of their optimization system. They investigated the knapsack problem and the travelling salesman problem (TSP) using

**Fig. 2.** High-level view of our method when it is applied to a single optimization state.

their method by leveraging customized data structures during neighbor generation. In comparison to our approach, they are limited to manually adjusting their method based on the domain and do not support any kind of variable-assignment ratings. The latter one is particularly important when exploring extremely large neighborhoods that cannot be expanded in memory (see Figure 1). Similar to these general approaches is the one by Lam et al. [12]. They investigated a parallelization of TSP on CPUs and GPUs using simulated annealing. As before, the proposed neighborhood generation is wired to and specialized for the underlying mathematical optimization model which makes it difficult to reuse it in a different context.

Luong et al. [8, 9] use the GPU to explore the neighborhood of a single state and to evaluate the consequences of several decisions. In this context, they focus on binary problems using different Hamming Distances. Again, this approach lacks generality: a generic problem of an unknown domain requires a much more sophisticated concept to explore the neighborhood. They also applied their own successor-generation approach in a parallelized implementation of Tabu Search [7]. Ghorpade et al. [5] used the method by Luong et al. to perform a parallel evaluation of all neighboring states in the domain of TSPs using the 2-opt algorithm. They adapted the method in such a way that they have encoded their own structures and strategies to generate successor states according to their use case. In a follow-up work from Melab et al. [11], they extended Luong et al.’s method to perform a whole framework-based approach. They use GPU acceleration for neighborhood exploration based on local-search metaheuristics as before. However, the basic exploration approach in the paper stays the same as before.

Novoa et al. [15] created a parallel search implementation for the quadratic assignment problems on GPUs. They use permutations to generate successor states in the scope of their GPU kernels. Their algorithm is based on a binary decision structure that allows variables to be 0 or 1. This significantly simplifies the permutation process in this domain compared to our generic approach.

The probably most related work is the one by Rashid et al. [13]. They discuss challenges of designing neighborhood generation on GPUs in general and give several proposals. Furthermore, they provide high-level algorithms to realize a

GPU optimization system based on S- and P-metaheuristics [19]. However, they do not give a detailed explanation regarding neighborhood generation. Their high-level approach already relies on a notation of  $N(x)$ . Once they have this abstract concept at hand, they can build upon this and apply their algorithms.

### 3 FANG

As previously mentioned, a heuristic optimization system often tracks multiple candidate states at the same point in time. Thereby, a single optimization state consists of state-dependent information and several variables  $\lambda_k$  that are part of the optimization problem, where  $k \in \{1, \dots, |\lambda|\}$  and  $|\lambda|$  refers to the number of variables. A common task is to find an assignment of all variables  $\lambda_k$  to values  $V_l$  according to the individual constraints of every variable in order to find the best state according to a given cost function, where  $l \in \{1, \dots, |V|\}$  and  $|V|$  refers to the number of possibilities. Our target are large-scale optimization problems that require the evaluation of hundreds of possibilities per variable and a large number of candidate states. As previously mentioned, the number of neighbor states might be very large, and thus, cannot be simply generated or returned by  $N(x)$ . Instead of creating and iterating over all neighbor states, we have to limit the range and want to enumerate only potentially "interesting" successors of a given state. In order to explore the neighborhood of every variable, we build upon the basic ideas for random-based local-exploration by Munawar et al. [14] and Campeotto et al. [2]. However, instead of randomly choosing and assigning variables, we investigate assignment possibilities of all variables and determine probabilities for every possible assignment (see below). We chose a single possibility based on a random value that is determined using a uniform random distribution.

We propose the concept of *local heuristics* that guide the successor generation in such a way that they rate variable assignments (see Figure 1). In other words, they answer the question: Is an assignment of variable  $\lambda_k$  to value  $V_l$  a good choice? They are called *local*, since the question is answered *locally* per variable and assignment: the rating of the assignment  $\lambda_k \rightarrow V_l$  takes the current state  $x$  into account but does not pay attention to other possible assignments of this variable. This leads to a major benefit: all ratings can be considered in parallel since every potential assignment is treated on its own. However, assignments of other variables have to be considered during the actual successor construction. There might be a constraint that hinders the assignments of two variables  $\lambda_k$  and  $\lambda_l$  to the same value  $V_p$ , for instance. For this reason, we assign different variables sequentially in general. This avoids race conditions during variable assignments and simplifies the rating process: Every rating can be computed by accessing all other already assigned variables since the optimization state is read only at this point in time. This is not a strict limitation and can be relaxed if multiple variables can be assigned without any interference. We further distinguish between different *variable types* (referred to as  $\lambda^T$ ). This is useful for many domain-specific problems that leverage different heuristics for distinct variables in order to simplify the modeling process.

### 3.1 High-Level Algorithm

From a high-level point of view, we distinguish between *active variables* that have to be assigned to a *value* and *inactive ones* that do not require a new assignment. Whether a variable is active or not during successor creation is typically determined by the surrounding optimization system. We assume that the decision step already happened and our task is to find all active variables that require an assignment in this step (Figure 2-1).

First, we have to pick an active variable and have to resolve the number of possibilities to assign the current one (Figure 2-2)<sup>2</sup>. Second, we can rate each possibility using a *local heuristic* (Figure 2-3). Thereby, a rating value  $R$  has a user-defined value type that is opaque to our algorithm. Since the initial rating itself is designed as a local process that handles every possibility independently, we added the opportunity to define a custom rating aggregator that accumulates global information about all ratings (Figure 2-4). An aggregator can carry multiple aggregation values that are combined with the help of custom aggregation functions. This step happens during and immediately after all possibilities have been rated.

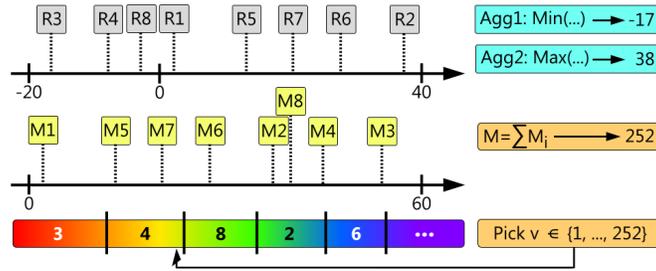
Afterwards, we can perform a mapping operation to convert each user-defined rating  $R$  using the globally aggregated information into a mapped rating  $M \in \mathbb{N}$  (Figure 2-5). From a theoretical point of view, a mapped rating can be seen as a probability of the associated assignment possibility. Note that the value 0 of a mapped rating  $M$  corresponds to a probability of 0. This allows to easily forbid possibilities that cannot/should not be selected for some reason. In practice, however, it is much easier to convert the initial rating into another value  $\in \mathbb{N}$  that can be converted to an assignment possibility (see Figure 3): A larger mapped rating indicates a higher probability with respect to the sum of all mapped ratings  $M$  (Figure 2-6). Next, we pick a uniformly-distributed random value  $v$  between 1 and  $M$  (Figure 2-7). We have to remap the chosen value  $v$  to an associated mapped rating  $M$ , and thus, to its original possibility  $l$  it was computed from (Figure 2-8). Finally, we can apply the selected possibility  $l$  and its associated (domain-specific) value  $V_l$  by assigning the variable  $\lambda_k$ . The whole process will be repeated until no active variable can be found any more.

### 3.2 Low-Level View

The whole algorithm can be realized with the help of a single GPU kernel. Every thread group (consisting of  $N$  threads) on the GPU handles a single optimization state and assigns all active variables in a cooperative way. The variables of a single state are managed with the help of Boolean sets (Figure 4-1, see subsection 3.8). Starting with these sets, we have to iterate over all active variables (Figure 4-2). The number of possibilities per variable is resolved by all threads of the group in parallel. This avoids expensive group-synchronization operations by preferring operations on registers. Alternatively, the number of possibilities could be resolved by the first thread of the group and can be made

<sup>2</sup> The number of possibilities can be seen as a subset of all possible successor states from Figure 1-2.

**Fig. 3.** Sample rating mapping that realizes a non-trivial rating process (see subsection 3.1 for more details). Eight different possibilities are individually rated according to a user-defined rating function (top). In this sample, positive ratings directly correspond to their final mapped value (middle). Negative values that are computed by the rating function should be more important than the largest positive value. For this reason, the custom aggregator stores the maximum value of all ratings which can be directly used to remap negative values. All mapped ratings are accumulated in order to derive their individual probability (bottom). Larger intervals correspond to higher probabilities. We can then choose our decision value  $v$  out of the computed set of values. This value is then mapped to a target interval, and thus, to a target value to assign.



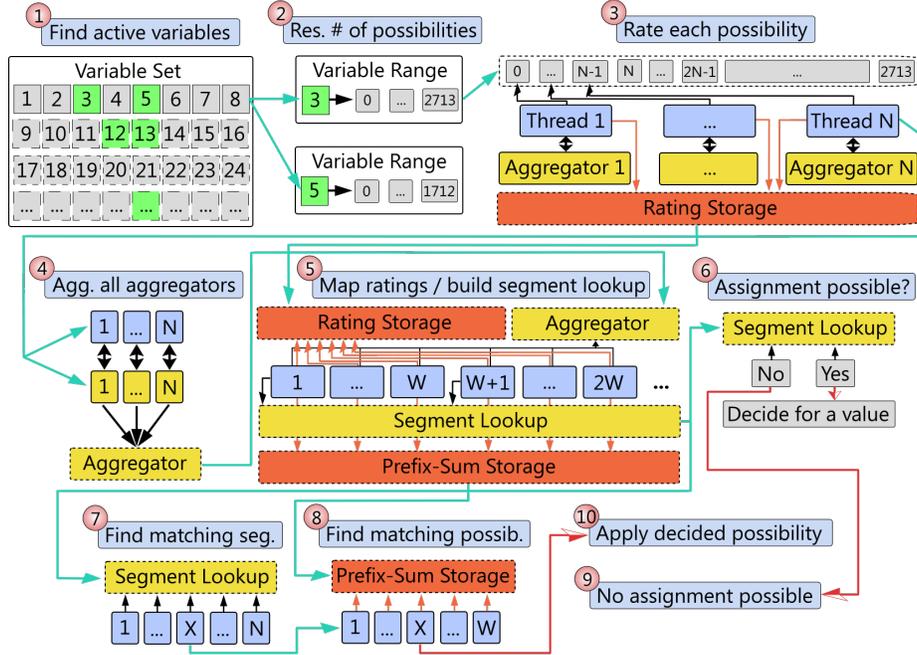
accessible by all other threads via shared memory. It reduces the number of active warps and leaves more opportunities to the warp dispatcher. However, we have not seen any computationally or memory-expensive implementations that made it necessary to rely on shared memory and group synchronization in practice.

Every possibility will be processed by a single thread using a group-stride loop (Figure 4-3). This ensures coalesced memory accesses in the scope of the heuristic  $h$  and the *rating storage*. The latter one stores all computed intermediate user-defined rating values  $R$  from the heuristic in global memory. Storing computed ratings avoids expensive re-computations during the mapping process. If a re-computation of all possibilities is much cheaper than storing the values in global memory, the rating storage can be omitted. However, storing the values in global memory does not impose a significant overhead when processing a large number of states since the memory latency could be hidden by the GPU.

The rating aggregators are locally maintained in register space in the scope of every thread. All computed ratings are directly accumulated using the local aggregators which avoids synchronization with other threads. After computing and storing all  $R$  values, the individual intermediate aggregators are combined into a globally available aggregator (Figure 4-4). The global aggregator can be stored in shared memory, registers or local memory depending on the individual requirements and the user-defined heuristic implementation (since it is readonly after the aggregation step).

Next, we map all rating values  $R$  to their mapped counterparts  $M$  using the previously computed aggregator information (Figure 4-5). We conceptionally split all mapped ratings into *segments* of the size of a single warp. This enables

**Fig. 4.** Low-level view of our method that is closely related to Algorithm 1. Black arrows indicate memory accesses. Red/organge arrows indicate global memory accesses to allocated regions our approach requires. Blue/green arrows indicate logical associations. Red arrows mark decisions. Dashed boxes are intermediate information that is available to all threads.



us to build an acceleration structure that significantly speeds up the possibility resolving step in the end. The acceleration structure is completely stored in shared memory and consumes a single 64bit unsigned integer per segment. Leveraging shared memory ensures high-performance random-access lookups and avoids consumption of additional global memory. The disadvantage of this concept is the limited number of possibilities per variable that can be stored in shared memory: Assuming 24kb of shared memory per thread group<sup>3</sup> and a warp size of 32 [17], yields a total number of  $\frac{24 \cdot 1024}{8} \cdot 32 = 98304$  possibilities.

All individually computed mapped ratings  $M$  will be processed using a group-stride loop. They are on-the-fly accumulated in the form of a prefix sum that is computed for the whole group after every thread has mapped its associated rating value  $R$ . Using a prefix sum allows us to represent the different probabilities that are implied by all  $M$  values (see Figure 3). The resulting prefix-sum values will be written into a *prefix-sum storage* that resides in global memory. Again, this avoids expensive prefix-sum re-computations (see above). In the same run, the first thread of every warp stores the right boundary value of all threads in the warp by writing its value into the shared-memory segment lookup.

<sup>3</sup> Without limiting the parallel execution of multiple groups per multiprocessor.

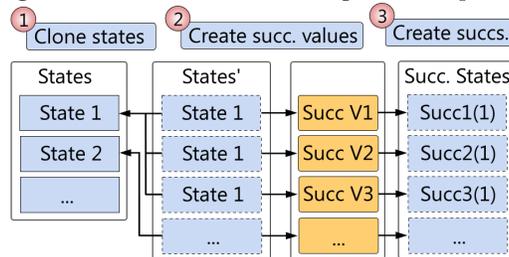
In the 6th step (Figure 4-6) all threads determine whether an assignment is possible or not. This breaks down to a simple check whether the value of the last segment in the used lookup is zero or not. If no assignment is possible, we will have to deactivate the current variable  $\lambda_k$  (Figure 4-9) and continue with the next active one. If an assignment is possible, every thread in the group picks the same decision value  $v$  that is less than the maximum prefix sum value of the last segment.

The last two phases resolve the possibility that belongs to the chosen decision value  $v$ . First, we have to identify the target segment that narrows the search space to the number of threads in a warp (Figure 4-7). We do not leverage any advanced search algorithm to find the target segment (like a binary-search, for instance). Instead, we simply check each segment using a single thread and group-stride loop. Once the target segment has been found, the first warp of every group investigates all potential matches in the target segment in parallel (Figure 4-8). Only one thread can win during the comparison of its neighboring prefix-sum values to the chosen value  $v$ . Finally, this thread applies the decided possibility  $V_l$  to the current variable  $\lambda_k$  and deactivates it. Afterwards, we continue with the next active variable by repeating the whole assignment process.

### 3.3 Successor Generation

The actual successor generation process happens in three steps (see Figure 5). We leverage a double-buffer approach: a source and a target buffer containing all state information. First, the source states from the readonly source buffer will be cloned into the target buffer. A user defined parameter specifies the number of successors per input state. A common scenario in this context is a single input state that will be expanded several times in the target buffer. Second, the successor indices will be generated using an external random number generator and assigned to every new cloned state. The successor index is used as random seed for all subsequent assignment decisions in the scope of the associated state. Third, the already presented variable-assignment algorithm will be applied to every target state resulting in different successor states. Finally, the source and target buffers are swapped to delete all source states and to "free" memory for the next successor-generation step.

**Fig. 5.** The visualized successor generation process



### 3.4 Algorithm

The main algorithm can be found in Algorithm 1 which represents a single GPU kernel and the actual implementation of our  $N(x)$  function. It is designed in a way that it can be directly converted into code with minor adjustments (see subsection 3.8). The kernel is launched with a group size that is a multiple of the warp size in order to achieve high occupancy and to ensure that the segment lookup works as intended. The term *group index* refers to the index of the  $i$ -th thread inside the group.

The algorithm takes a state  $x$  and a set of heuristics  $h^T$  for the different variable types  $\lambda^T$  that can occur in the scope of the problem instance. We allocate the required amount of shared memory, initialize the random seed based on the successor index and iterate over all active variables sequentially. Lines 6–8 and Algorithm 2 correspond to the FANG steps 1–4 in Figure 4. Lines 9–18 and Algorithm 3 correspond to the FANG steps 5,6 and 9. Note that we also call the user-defined heuristic in line 14 that can perform state-dependent adjustments in the case of no assignment before deactivating the variable. Lines 20–31 reflect the remaining FANG steps 7–10. Again, we execute user-defined code to enable customized assignment logic. Finally, we store the updated random-number-generator seed in the optimization state. Upcoming assignment steps will then use the updated seed to generate "new" random numbers.

Note that there cannot be any race-conditions between different variable assignments. Only one thread at a time executes the actual (no-)assignment, activation or deactivation procedures. The thread barriers in line 17 and 33 ensure that the changes of these threads will be visible to all other threads in the group after an assignment loop. This also guarantees that the influence of a variable assignment can be taken into account by all other variables that are assigned afterwards.

### 3.5 Assignment Order

The sequential assignment of all variables can lead to a bias, and thus, to unintended behavior of the general optimization system. Assume two variables  $\lambda_1$  and  $\lambda_2$  that have the following possibilities:

- $\lambda_1 \Rightarrow \{1\}$  and
- $\lambda_2 \Rightarrow \{1\}$ ,

with the constraint that  $V(\lambda_1) \neq V(\lambda_2)$ . Consider a state in which both variables are active at the same time. Let us also assume, we assign both variables sequentially in a pre-defined order, e.g.  $\lambda_1$  before  $\lambda_2$ . Then, variable  $\lambda_2$  cannot be assigned in any case to the value  $V(\lambda_2) = 1$ . Instead the variable assignment will always yield  $V(\lambda_1) = 1$  and  $V(\lambda_2) = \perp$  (could not be assigned). This can be seen as the intended behavior that was desired by the user. If not, we will also have to change the order in which we assign variables from time to time. This can be directly achieved by randomly permuting the order in which we iterate over all active variables.

---

**Algorithm 1: The FANG Algorithm**

---

**Input:** state  $x$ , heuristics  $h^T$  for the variables types  $\lambda^T$

```

1 sharedSegments := shared memory rating type[#ratings per group];
2 sharedPrefixSum := shared memory int64[group size];
3 seed := SuccessorValue( $x$ );
4 foreach  $\lambda \in$  active variables do
5    $h :=$  DetermineHeuristic( $\lambda$  based on its type  $\lambda^T$ );
6    $|R| :=$  GetNumberOfPossibilities( $h, x, \lambda$ );
7   /* Algorithm 2 */
8   agg := RateAllPossibilities( $x, h, \text{sharedPrefixSum}$ );
9   /* Algorithm 3 */
10  MapAllRatings( $x, h, \text{agg}, \text{sharedSegments}, \text{sharedPrefixSum}$ );
11  /* Check for a possible assignment */
12  globalRating := PrefixSumValue(last segment);
13  if  $\text{globalRating} < 1$  then
14    if is first lane in warp then
15      /* No assignments possible */
16      CouldNotAssign( $h, x, \lambda$ );
17      DeactivateVariable( $\lambda$ );
18    end
19    /* Wait for all threads... */
20    group barrier;
21    continue;
22  end
23  /* Find decision value  $v$  */
24   $v, \text{seed} :=$  DrawRandomNumber()  $\in [0, \dots, \text{globalRating} - 1]$ ;
25  /* Algorithm 4 */
26  segment := FindMatchingSegment( $x, v,$ 
27    sharedSegments, sharedPrefixSum);
28  /* Select possibility based on  $v$  */
29  if is first warp then
30     $p :=$  segment * warp size + group index;
31    leftSum := prefixSumStorage( $x$ )[ $p - 1$ ];
32    rightSum := prefixSumStorage( $x$ )[ $p$ ];
33    if  $\text{leftSum} \leq v \wedge \text{rightSum} < v$  then
34      Assign( $h, x, \lambda$ );
35      DeactivateVariable( $\lambda$ );
36    end
37  end
38  /* Wait for all threads... */
39  group barrier;
40  /* Store updated successor value */
41  if is first thread in group then
42    SuccessorValue( $s$ ) := seed;
43  end
44 end

```

---

---

**Algorithm 2: RateAllPossibilities**

---

```

Input: state  $x$ , heuristic  $h$ , shared memory
1  $\text{agg} := \text{CreateAggregator}(h)$ ;
  /* Iterate over all possibilities and aggregate information */
2 for  $i := \text{group index}$ ;  $i < |R|$ ;  $i += \text{group size}$  do
3    $R_i := \text{ComputeRating}(h, x, \lambda, i)$ ;
4    $\text{ratingStorage}(x)[i] := R_i$ ;
5    $\text{agg} := \text{Aggregate}(\text{agg}, R_i)$ ;
6 end
  /* Wait for all threads... */
7 group barrier;
  /* Aggregate all thread-local aggregators */
8  $\text{agg} := \text{reduce}(\text{sharedPrefixSum}, \text{agg})$ ;
9 return  $\text{agg}$ ;

```

---



---

**Algorithm 3: MapAllRatings**

---

```

Input: state  $x$ , heuristic  $h$ , aggregator, shared memory
1  $\text{lowerBound} := 0$ ;
  /* Iterate over ratings in the rating storage */
2 for  $i := \text{group index}$ ;  $i < |R|$ ;  $i += \text{group size}$  do
3    $\text{initialRating} := \text{ratingStorage}(x)[i]$ ;
4    $M_i := \text{MapRating}(h, \text{initialRating}, \text{agg})$ ;
5    $\text{prefix} := \text{lowerBound} + \text{prefix sum}(\text{sharedPrefixSum}, \text{initialRating})$ ;
  /* Store prefix sum, update lower bound, and update segment */
6    $\text{lookup}$ 
7    $\text{prefixSumStorage}(x)[i] := \text{prefix}$ ;
8    $\text{lowerBound} := \text{PrefixSumValue}(\text{last thread in group})$ ;
9   if is first lane in warp then
10     $\text{sharedSegments}[\frac{i}{\text{warp size}}] := \text{lowerBound}$ ;
11 end
  /* Wait for all threads... */
12 group barrier;

```

---

### 3.6 Duplicate States

A disadvantage of our parallel-processing method is the (potential) generation of duplicate successor states from a single state. This is caused by our random selection process that chooses from different assignment possibilities. The probability that this actually happens depends on the optimization domain, e.g. the number of variables, individual constraints, the rating functionality and the number of possibilities per assignment. This can be safely neglected in general, since the probability that two identical successors will end up in the same state after  $n$  successor generation steps is  $p^n$ , where  $p \in [0, \dots, 1]$  refers to the probability that a duplicate state emerges.

---

**Algorithm 4:** FindMatchingSegment

---

**Input:** state  $x$ , value  $v$ , shared memory

```

1 ResetSharedMemory();
2 targetSegmentIdx := -1;
  /* Iterate over segment lookup in shared memory */
3 for  $i :=$  group index;  $i < \lceil \frac{|R|}{\text{warp size}} \rceil$ ;  $i +=$  group size do
4   leftBoundary := sharedMemory[i - 1];
5   rightBoundary = sharedMemory[i];
6   match := leftBoundary  $\leq v \wedge$  rightBoundary  $< v$ ;
7   targetSegmentIdx := reduce (sharedPrefixSum, match ? i : -1);
8   if target segment match then
9     | break;
10  end
11 end

```

---

### 3.7 Memory Consumption

Our algorithm requires a temporary array to store all custom rating values and all prefix-sum information. The size of a single array entry in bytes is  $|entry| = \text{sizeof}(\text{rating type}) + 8$ , since we store all accumulated mapped ratings as unsigned 64bit integers to avoid overflows. Segment-lookup information is stored in shared memory on the multiprocessor and does not require additional global memory. However, this array is required per state. Consequently, the global memory consumption is

$$\max(|R(\lambda_0)|, \dots, |R(\lambda_{n-1})|) \cdot |X| \cdot |entry|, \quad (1)$$

where  $|R(\lambda_i)|$  is the maximum number of possibilities of the variable  $\lambda_i$  and  $|X|$  is the number of states.

If we process multiple variables in parallel, we require several instances of the array in memory to store all intermediate values. Hence, Equation 1 has to be adapted in order to reflect the additional memory consumption:

$$\left( \sum_i |R(\lambda_i)| \right) \cdot |X| \cdot |entry|, \quad (2)$$

if we assume that all variables can be assigned in parallel. Since we disable parallel variable assignment in our real-world applications, the memory consumption is the one from Equation 1.

### 3.8 Implementation Details

We have implemented our algorithm in C++ using Cuda for all GPU kernels. Variables are managed with the help of bit-sets in form of unsigned integers. They also act as acceleration structures to skip over larger regions of inactive variables. Active variables are identified with the help of hardware bit-manipulation

Load	X	\lambda	R	1080 Ti	$\sigma$	Titan X	$\sigma$	i9 7940X	$\sigma$	R. 2700X	$\sigma$
1	1024	8	1224	<b>2.20</b>	0.09	<b>3.04</b>	0.12	<b>3.26</b>	0.72	<b>4.61</b>	0.20
*	*	*	4488	<b>3.34</b>	0.14	<b>4.73</b>	0.10	<b>10.42</b>	1.16	<b>14.95</b>	1.21
*	*	*	9800	<b>5.05</b>	0.33	<b>7.66</b>	0.10	<b>22.28</b>	1.43	<b>31.06</b>	1.12
*	*	32	1224	<b>6.97</b>	0.63	<b>10.85</b>	0.44	<b>11.06</b>	1.40	<b>16.03</b>	0.91
*	*	*	4488	<b>9.85</b>	0.89	<b>17.13</b>	0.13	<b>40.02</b>	1.29	<b>56.18</b>	1.82
*	*	*	9800	<b>16.90</b>	1.22	<b>28.49</b>	0.22	<b>87.31</b>	1.81	<b>121.75</b>	2.64
*	4096	8	1224	<b>6.48</b>	0.57	<b>11.54</b>	0.11	<b>12.08</b>	1.44	<b>16.73</b>	0.49
*	*	*	4488	<b>9.84</b>	0.71	<b>17.75</b>	0.12	<b>40.18</b>	1.54	<b>55.50</b>	1.40
*	*	*	9800	<b>16.94</b>	0.98	<b>30.13</b>	0.10	<b>85.09</b>	1.79	<b>118.61</b>	2.25
*	*	32	1224	<b>23.27</b>	1.47	<b>44.33</b>	0.44	<b>42.21</b>	1.79	<b>60.32</b>	1.31
*	*	*	4488	<b>37.78</b>	2.19	<b>70.08</b>	0.69	<b>154.70</b>	1.96	<b>217.85</b>	3.86
*	*	*	9800	<b>66.30</b>	1.75	<b>118.56</b>	0.53	<b>336.196</b>	3.96	<b>471.87</b>	6.80

**Table 1.** Influence of the number of states, the number of possibilities and the number of variables on the overall run time. GPUs: GeForce GTX 1080 Ti and GeForce GTX Titan X. CPUs: Intel Core i9 7940X and AMD Ryzen 7 2700X.

instructions. For performance reasons, we typically leverage an *XorShift\** or an *XorShift1024\** random-number generator [10]. They yield excellent results on our common optimization domains and are not too expensive to compute in every assignment step.

We further leverage template specialization to instantiate different assignment kernels for each heuristic. Based on our experience, the majority of heuristics use uniform control flow that does not diverge into too many distinct sections. This results in specialized GPU kernels for each variable kind that benefit from the common uniform-control-flow pattern of each heuristic, which significantly reduces thread divergences. Moreover, we assign variables of different types typically sequentially to ensure that decisions from previous categories are visible to the heuristic of the next variable kind.

All memory buffers are allocated before the optimization process starts, since the required memory size is already known during initialization. This avoids unnecessary dynamic memory allocations during runtime. We use warp shuffles to improve performance of all prefix-sum and reduction computations [16]. Note that the loops in Algorithms 3 and 4 have thread divergences the way they are described in the paper. In order to leverage group barriers inside reduce and prefix-sum computations, these loop bounds are padded to avoid any divergences in our implementation. Furthermore, accesses to shared memory in Algorithm 4 and to the prefix-sum storage in Algorithm 1 are modified to avoid out-of-bounds accesses.

## 4 Evaluation

The evaluation section does not cover benchmarks of our  $N(x)$  algorithm in the scope of different optimization systems using hard-to-reproduce and difficult to understand optimization benchmarks. Instead, we evaluated the pure successor-state generation process using a well known heuristic from the field of shortest-path optimization. We consider a discrete 2D grid using the Manhattan distance,

Load	$ X $	$ \lambda $	$ R $	1080 Ti	$\sigma$	i9 7940X	$\sigma$
1	1024	8	1224	<b>2.20</b>	0.09	<b>3.26</b>	0.72
8	*	*	*	<b>1.88</b>	0.05	<b>7.81</b>	0.11
16	*	*	*	<b>1.82</b>	0.02	<b>14.05</b>	0.15
1	*	*	4488	<b>3.34</b>	0.14	<b>10.42</b>	1.16
8	*	*	*	<b>3.40</b>	0.13	<b>27.39</b>	0.47
16	*	*	*	<b>2.75</b>	0.02	<b>50.11</b>	0.55
1	4096	*	1224	<b>6.48</b>	0.57	<b>12.08</b>	1.44
8	*	*	*	<b>6.26</b>	0.03	<b>29.42</b>	0.28
16	*	*	*	<b>6.26</b>	0.02	<b>53.70</b>	0.44
1	*	*	4488	<b>9.84</b>	0.71	<b>40.18</b>	1.54
8	*	*	*	<b>10.00</b>	0.89	<b>104.61</b>	0.86
16	*	*	*	<b>9.91</b>	0.70	<b>193.39</b>	0.98

**Table 2.** Influence of the compute load on the overall runtime. Please note that the table is not meant to compare the theoretical computational power of GPUs against CPUs. GPU: GeForce GTX 1080 Ti. CPU: Intel Core i9 7940X.

which is given by

$$d(p, q) = \sum_{i=1}^n |p_i - q_i|. \quad (3)$$

We iteratively move a point  $p$  to its neighboring cell according to the evaluation result of the Manhattan distance. Hence, we consider neighboring grid cells in 2D as potential target points for the next step during heuristic evaluation. The number of variables  $|\lambda|$  indicates the number of points that we want to compute a shortest path for. Consequently, every variable  $\lambda_i$  corresponds to a single current point  $p$  and a corresponding goal point  $q$  that we want to reach. We choose  $|\lambda|$  to be  $\in \{8, 32\}$  to demonstrate the effect of a small number of variables on the overall run time. The number of possibilities  $|R|$  for every variable is derived from the number of neighboring cells that we can move to in a single step. This number can be computed using

$$|R| = j \cdot j - 1, \quad (4)$$

where  $j$  refers to the number of neighboring cells in one dimension. Since we are interested in large-scale optimization problems, we chose  $j \geq 35$  to have some reasonable number of possibilities per variable ( $j \in \{35, 67, 99\}$ ). Choosing  $j$  to be smaller does not make any sense for a GPU-based optimizer since the number of possibilities is too small. The number of states  $|X|$  is chosen to be  $\in \{1024, 4096\}$  to create some workload. However, in reality  $|X| \ll |R|$ , which also avoids duplicate states. For the sake of completeness, we also included performance measurements of such cases in which  $|X| > |R|$ . In order to simulate more- and less-expensive rating computations based on the Manhattan distance, we introduce the *load* factor. It indicates the number of  $d(p, q)$  computations that are performed using different goal positions  $q$  inside a loop. As baseline we chose *load* to be 1, which is the worst-case for our algorithm since the workload of every single possibility evaluation is extremely small.

Our CPU implementation is derived from an object-oriented design, that instantiates state objects encapsulating bit-fields of active variables. During suc-

cessor generation, these states are cloned and are assigned in parallel to leverage all cores. Schulz et al. [18] reported on many benchmarks in the field of discrete optimization that GPU and CPU comparisons of algorithms often lack comparability of the results. For this reason, our CPU version uses the same method as shown in Figure 2 to ensure that the solutions from all scenarios are identical on the CPU and the GPU.

We used two GPUs from NVIDIA (a GeForce GTX 1080 Ti and a GeForce GTX Titan X) and two CPUs: one from Intel (a Core i9 7940X) and one from AMD (a Ryzen 7 2700X). Every performance measurement is the median execution time of 100 algorithm executions. Moreover, all variables  $\lambda_i$  were considered to be active at the same time to increase the number of assignment steps per state. The CPU code was compiled with *msvc v19.16.27030* with all compiler optimizations and the AVX2 instruction set enabled. The Cuda code was compiled with *nvcc v10.1.105* with all compiler optimizations enabled.

Table 1 shows the main evaluation table demonstrating the impact of the number of states, possibilities and variables on the overall run time. As previously mentioned, the load factor was set to 1 in order to measure the worst case of our method. On the GPU side, the 1080 Ti is roughly between  $1.5\times$  and  $2\times$  as fast on all benchmarks compared to its older generation counterpart. Both GPUs scale well with the overall complexity of the assignment problem. However, they are heavily influenced by the number of active variables  $|\lambda|$ , which are assigned sequentially. This holds also true for the CPUs: doubling  $|\lambda|$  roughly results in a doubled execution time, which corresponds to the expected behavior. Fixing  $|\lambda|$  and changing the number of possibilities  $|R|$  per variable results in a very good scalability of our method on the GPUs: increasing the number of possibilities by a factor of 8 (from 1224 to 9800) results in an increase of the run time of roughly a factor of 4. This is caused by the parallel evaluation of many possibilities in the scope of a single thread group, which is very work efficient. The CPUs have a dramatic slowdown that is tightly coupled to  $|R|$  since they are processing the assignment possibilities one by one. Fixing  $|\lambda|$  and  $|R|$  while changing  $|X|$  yields comparable results: The GPUs scale well when increasing the number of states, whereas the CPUs register a bad scalability. This is due to the fact that the GPU scheduler can choose between more thread groups to hide memory latency and improve the overall occupancy.

Table 2 shows the impact of the compute load on the run time of the assignment process. The measurements clearly show the great scalability of our algorithm on the GPUs when it comes to more complex rating functions. A larger computational load shifts the focus from a memory-dependent execution to a computation-dependent execution. This affects the behavior of the GPU scheduler, such that the scheduling overhead can be significantly reduced to spent more time on the actual computations. Hence, an increased load does cause any slowdowns on the GPUs. The CPU versions suffer dramatically from the additional load, as their maximum occupancy was already reached.

The general speedup that can be achieved using our algorithm on the GPU over traditional CPU versions yields speedups between  $7.7\times$  on problems with small computation load and up to  $27\times$  using more workload depending on the

actual hardware. In general, our method provides great scalability that significantly outperforms the CPU versions.

## 5 Conclusion

We present a new method to implement a generic and reusable  $N(x)$  function for heuristic optimization systems. The neighborhood is explored using *local heuristics* that rate all assignment possibilities of every variable. These ratings are converted to probabilities which form the basis to find a decision value. The value itself is resolved using a random-number generator that has a unique seed for every optimization state. The decision is mapped to a variable-assignment possibility using specially designed lookup tables that can be stored in fast on-chip-memory.

Our approach scales very well with the complexity/cost of the rating functions. It also scales excellently with the number of states and assignment possibilities per variable. For instance, an assignment of 32 variables that are active at the same time in 1024 states with 4488 possibilities each require only  $\approx 10$  milliseconds on current hardware to complete. Comparing the performance to traditional  $N(x)$  implementations on the CPU yields significant speedups of up to  $27\times$  on our evaluation scenarios. Moreover, it allows the design and implementation of fully GPU-based heuristic-driven optimization systems without the need to perform neighbor search or successor-state generation on the CPU. This makes it a perfect extension for every modern heuristic-based optimizer.

Probably the main downside of our approach is the high memory consumption. We require a single array entry of at least 12 bytes in memory for every assignment possibility in every state that should be processed in parallel. However, we do not believe that this is a major limitation in practice since large optimization problems require huge amounts of memory anyway.

In the future we would like to extend the concept to support parallel assignment of variables. This would require specific compiler extensions to automatically determine whether or not some variables can be assigned in parallel. In addition, we want to experiment with locally cached ratings in shared memory, since the recent trend has shown increasing sizes of on-chip-memory.

## Acknowledgments

The authors would like to thank Wladimir Panfilenko and Thomas Schmeyer for their suggestions and feedback regarding our method. Furthermore, we would like to thank Gian-Luca Kiefer for additional feedback on the paper. Special thanks to Wladimir at this point for adding the concept of integer-based bit sets for active variables in a single state. This reduces global-memory consumption and improves performance of searching for active variables.

## References

1. Abdelkafi, O., Chebil, K., Khemakhem, M.: Parallel local search on GPU and CPU with OpenCL Language. In: Proceedings of the first international conference on Reasoning and Optimization in Information Systems (09 2013)
2. Campeotto, F., Dovier, A., Fioretto, F., Pontelli, E.: A GPU Implementation of Large Neighborhood Search for Solving Constraint Optimization Problems. In: Proceedings of the Twenty-first European Conference on Artificial Intelligence (2014)
3. Campeotto, F., Dal Palù, A., Dovier, A., Fioretto, F., Pontelli, E.: Exploring the Use of GPUs in Constraint Solving. In: Practical Aspects of Declarative Languages (2014)
4. Focacci, F., Laburthe, F., Lodi, A.: Local Search and Constraint Programming. Springer US (2004)
5. Ghorpade, S., Kamalapur, S.: Solution Level Parallelization of Local Search Metaheuristic Algorithm on GPU. In: International Journal of Computer Science and Mobile Computing (2014)
6. Köster, M., Leiða, R., Hack, S., Membarth, R., Slusallek, P.: Code Refinement of Stencil Codes. *Parallel Processing Letters (PPL)* **24** (2014)
7. Luong, T.V., Loukil, L., Melab, N., Talbi, E.: A GPU-based iterated tabu search for solving the quadratic 3-dimensional assignment problem. In: ACS/IEEE International Conference on Computer Systems and Applications (AICCSA) (2010)
8. Luong, T.V., Melab, N., Talbi, E.G.: Large Neighborhood Local Search Optimization on Graphics Processing Units. In: Workshop on Large-Scale Parallel Processing (LSPP) in Conjunction with the International Parallel & Distributed Processing Symposium (IPDPS) (2010)
9. Luong, T.V., Melab, N., Talbi, E.G.: Neighborhood Structures for GPU-based Local Search Algorithms. *Parallel Processing Letters* (2010)
10. Marsaglia, G.: Xorshift RNGs. *Journal of Statistical Software, Articles* **8** (2003)
11. Melab, N., Luong, T.V., Boufaras, K., Talbi, E.G.: ParadisEO-MO-GPU: a framework for parallel GPU-based local search metaheuristics. In: 11th International Work-Conference on Artificial Neural Networks (2011)
12. Ming Lam, Y., Hung Tsoi, K., Luk, W.: Parallel neighbourhood search on many-core platforms. *International Journal of Computational Science and Engineering* **8** (2013)
13. Mohammad Harun Rashid, L.T.: Parallel Combinatorial Optimization Heuristics with GPUs. *Advances in Science, Technology and Engineering Systems Journal* **3** (2018)
14. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework. *Genetic Programming and Evolvable Machines* (2009)
15. Novoa, C., Qasem, A., Chaparala, A.: A SIMD Tabu Search Implementation for Solving the Quadratic Assignment Problem with GPU Acceleration. In: Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure (2015)
16. NVIDIA: Faster Parallel Reductions on Kepler (2014)
17. NVIDIA: CUDA C Programming Guide v10 (2019)
18. Schulz, C., Hasle, G., Brodtkorb, A.R., Hagen, R., T.: GPU computing in discrete optimization. Part II: Survey focused on routing problems. *EURO Journal on Transportation and Logistics* (2013)
19. Talbi, E.G.: Metaheuristics: From Design to Implementation. Wiley Publishing (2009)