GPU-Accelerated Tree-Search in Chapel versus CUDA and HIP

Guillaume Helbecque[a,b], Ezhilmathi Krishnasamy[a], Nouredine Melab[b], Pascal Bouvry[a]

[a] *Université du Luxembourg, DCS-FSTM/SnT, Luxembourg*
[b] *Université de Lille, CNRS/CRIStAL UMR 9189, Centre Inria de l'Université de Lille, France*

Abstract

In the context of exascale programming, the PGAS-based Chapel is among the rare languages targeting the holistic handling of high-performance computing issues including the productivity-aware harnessing of Nvidia and AMD GPUs. In this paper, we propose a pioneering proof-of-concept dealing with this latter issue in the context of tree-based exact optimization. Actually, we revisit the design and implementation of a generic multi-pool GPU-accelerated tree-search algorithm using Chapel. This algorithm is instantiated on the backtracking method and experimented on the N-Queens problem. For performance evaluation, the Chapel-based approach is compared to Nvidia CUDA and AMD HIP low-level counterparts. The reported results show that in a single-GPU setting, the high GPU abstraction of Chapel results in a loss of only 8% (resp. 16%) compared to CUDA (resp. HIP). In a multi-GPU setting, up to 80% (resp. 71%) of the baseline speed-up is achieved for coarse-grained problem instances on Nvidia (resp. AMD) GPUs.

Keywords: Chapel, Tree-Search, GPU computing, CUDA, HIP, N-Queens, AMD, Nvidia

1. Introduction

Graphics Processing Units (GPUs) have emerged as building blocks in modern supercomputers[1], reshaping the landscape of high-performance computing (HPC). Their parallel processing capabilities accelerate computations, making them invaluable in addressing complex applications across diverse domains, including scientific simulation, artificial intelligence, and operations research [1; 2; 3].

In the context of tree-based exact optimization, focus of this paper, the consideration of GPUs in algorithms like backtracking or Branch-and-Bound (B&B), plays a crucial role in expediting decision-making processes, making them a valuable asset in solving complex problems. It also raises multiple challenges related to the irregular workload, dynamic memory requirement, and data exchanges of those methods.

Many works proposed efficient GPU-enhanced B&B algorithms to solve challenging combinatorial optimization problems (COPs), such as the Permutation Flowshop Scheduling Problem (PFSP) or the Knapsack problems [4; 5; 6; 7]. While the latter demonstrate significant improvements compared to CPU-based approaches, as well as a high scalability in terms of GPUs count, they are generally implemented using a combination of a low-level GPU programming model, such as CUDA or HIP, along with other parallel programming environments, such as OpenMP or POSIX standard. The writing of such programs is usually complex (GPU memory (de)allocation, host-device/device-host data transfers, GPU kernels, *etc.*), error-prone, and often specific to a GPU architecture, e.g., Nvidia or AMD.

Moreover, the overall competing power is increasing and modern architectures are getting more complex[1], especially with the arrival of the exaflop-level Frontier supercomputer in the Top500 ranking in June 2022. Even though we can reasonably say that the hardware bet has paid off, the software challenge is currently being addressed in some exascale initiatives such as Partitioned Global Address Space (PGAS) models [8; 9].

In the last two decades, PGAS-based parallel environments based on higher-level abstraction have emerged, e.g., X10, UPC, and Chapel. In addition, some efforts have been made to support GPU programming on such environments [10; 11; 12].

In this work, we focus on the Chapel programming language [13]. It is a versatile PGAS-based parallel programming language specifically designed for high-level, productive development across various architectures, seamlessly supporting multi-core, distributed and GPU computing environments.

To the best of our knowledge, the only work targeting GPU-accelerated tree-search in Chapel is the one of Carneiro *et al.* [14]. It implements a GPU-based tree-search algorithm in Chapel, targeting permutation-based COPs. The proposed algorithm exploits Chapel's iterators by combining a partial search strategy with pre-compiled CUDA kernels for more efficient exploitation of the intra-node parallelism. Extensive experimentation on big N-Queens problem instances shows that up to 90% of the linear speed-up can be achieved.

This work provides a different design and implementation of a GPU-accelerated tree-search in Chapel. This is based on a generic multi-pool approach including a load-balancing mechanism. The implementation is based on the native GPU support of Chapel and does not require any additional programming environment, in contrast to the cited references. More-

[1]Top500 ranking of supercomputers worldwide (11/2023): `https://www.top500.org/lists/top500/2023/11/`.

over, Chapel allows to compile and execute the same code on different GPU architectures, without additional effort. The proposed algorithm is instantiated on the backtracking method and experimented on the N-Queens problem. The experiments are conducted on Nvidia and AMD GPU-powered systems, and compared to CUDA- and HIP-based implementations, respectively.

The remainder of the paper is organised as follows. Section 2 presents some background on parallel tree-search. Then, we describe in Section 3 the design and implementation of the proposed algorithm in Chapel, along with the CUDA-based baseline. The latter are then evaluated and compared in terms of performance in Section 4. Finally, Section 5 draws the conclusions and highlights the future perspectives.

2. Parallel tree-search

2.1. Serial tree-search

Tree-search algorithms are strategies that implicitly enumerate a solution space, dynamically building a tree. They start with a root node representing the initial problem to be solved, and nodes are branched during the search process, generating children nodes more constrained than their father node. The generated nodes are evaluated, and then, the valid and feasible ones are stored in a pool-like data structure. The search generates and evaluates nodes until the data structure is empty or another termination criterion is satisfied.

During the search, if an undesirable state is reached, the algorithm discards this node and then chooses an unexplored node in the pool. This action prunes some regions of the solution space, preventing the algorithm from unnecessary computations. However, the pruning of subproblems makes the shape of the tree irregular, which might result in load imbalance when parallel computing is used.

Different strategies exist to expand tree nodes, such as Depth-First Search (DFS) or Breadth-First Search (BFS). In DFS, we explore the node branch as far as possible before backtracking and expanding other nodes. It is easily implemented by storing generated, but not yet evaluated, nodes in a stack (last-in, first-out, LIFO). In contrast, BFS explores all nodes at the present depth prior to moving on to the nodes at the next depth level and is generally implemented using a queue (first-in, first-out, FIFO). In this work, DFS is preferred since the memory requirements of BFS often become excessive.

2.2. Parallel tree-search

The most general and most frequently used model to parallelize tree-search algorithms is the parallel tree-exploration model [15]. It consists in exploring several disjoint subspaces in parallel, meaning that multiple DFS, rooted in different tree nodes, are performed in parallel, as shown in Fig. 1. Searching these parts requires no (e.g., in backtracking) or minimal (e.g., in B&B) communication between workers.

We adopt a collegial multi-pool approach, in which each worker manages its own pool [16]. This approach alleviates the bottleneck problem that occurs in single-pool approaches

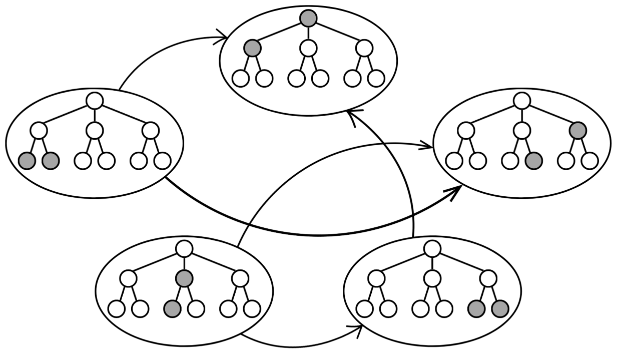
Figure 1: Illustration of the parallel tree-exploration model.

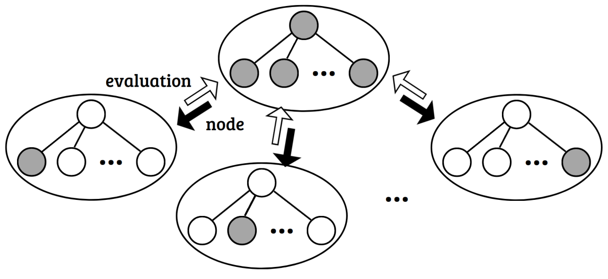

Figure 2: Illustration of the parallel evaluation of nodes model.

but raises the issue of balancing the workload between multiple pools. In this work, we adopt a static load balancing mechanism, which consists in evenly distributing the workload between workers before the parallel tree-exploration starts.

2.3. GPU-accelerated tree-search

We exploit GPU devices using the parallel evaluation of nodes model in which the generated nodes are evaluated in parallel, as shown in Fig. 2. More precisely, we combine this model with the parallel tree-exploration one by parallelizing the evaluation operator of each independent worker. This model is data-parallel, intrinsically synchronous, and fine-grained (the cost of the node evaluation), which is an execution model that fits GPUs well. It is well-adapted in cases where the cost of the node evaluation function is high compared with the rest of the algorithm.

3. Design and Implementation

We first present in Section 3.1 a background on GPU programming in Chapel. Then, we provide in Section 3.2 the design and implementation of the proposed GPU-accelerated tree-search algorithm in Chapel. Finally, we present in Section 3.3 the CUDA-based baseline implementation used in our experimental evaluation.

3.1. GPU programming in Chapel

Chapel is a versatile parallel programming language specifically designed for productive parallel computing at scale [13]. It supports a multi-threaded execution model via high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. Chapel's *locale* type enables users to specify and reason about the placement of data and tasks on a target architecture in order to tune for locality and affinity. Chapel applies the partitioned global address space paradigm and supports global-view data aggregates with user-defined implementations, permitting operations on distributed data structures to be expressed in a natural manner. Chapel supports code reuse and rapid prototyping via object-oriented design, type inference, and features for generic programming. Existing code can be integrated into Chapel programs (and vice-versa) via interoperability features.

In Chapel, GPU devices are seen as *sub-locales*, and code can be deployed on them using the `on`-clause. This clause allows the user to specify the locale, potentially remote, on which a task is to be executed. Inside such GPU sub-locales, Chapel supports two memory strategies to manage data: `array_on_device` (default) which stores array data directly on the device and store other data on the CPU host in a page-locked manner, and `unified_memory` which implicitly manages the migration of data to and from the GPU as necessary.

Chapel will launch kernels for all eligible loops that are encountered by a task executing on a GPU sub-locale. Loops are eligible when: (1) they are order-independent, (2) they only make use of known compiler primitives that are fast and local, (3) they do not call out to extern functions and (4) they are free of any call to a function that fails to meet the other criteria or accesses outer variables. Any code in a GPU sub-locale that is not within an eligible loop will be executed on the CPU.

It is worth to mention that the GPU support of Chapel is a work in progress and still suffers from some limitations:

- Interoperability: The use of most extern functions within a GPU eligible loop is not supported; only a limited set of functions used by Chapel's runtime library are supported. This is particularly restrictive given that rewriting existing software is generally prohibitively expensive.

- Portability: Not all GPU architectures are yet supported, e.g., Intel GPUs. In addition, it is not currently possible to compile for multiple AMD GPU architectures at the same time.

- Functionality: Some Chapel-specific features, such as distributed arrays, are not supported within GPU kernels. The latter implements arrays whose indices are mapped to different (remote) locales, and could be extended to GPU sub-locales as well.

3.2. GPU-accelerated tree-search in Chapel

3.2.1. Single-GPU approach

Fig. 3 shows the flowchart of our GPU-accelerated tree-search algorithm. The tree exploration starts on the CPU, and each node taken from the work pool is evaluated, potentially pruned, and branched. We evaluate and prune before branching in order to avoid the generation of the non promising nodes. The children nodes resulting from the branching operation are then inserted back into the pool. This process is repeated until the pool is empty. In order to exploit GPU-acceleration, we

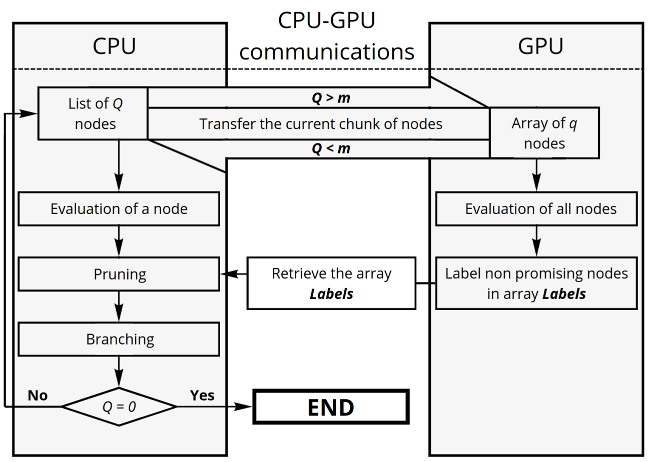

Figure 3: Flowchart of the proposed GPU-accelerated tree-search algorithm.

Algorithm 1: Single-GPU tree-search in Chapel

```
var pool = new Pool();
var root = new Node();
pool.add(root);

while pool.notEmpty() {
  if (pool.getSize() < m) {
    var parent = getNode(pool);
    evaluate_generate_children(parent, pool);
  }
  else {
    var parents: [] Node = getNodes(pool, M);
    var labels: [] uint(8) = noinit;
    on here.gpus[0] { //execute on GPU
      const parents_d = parents; //host-to-device
      var labels_d: [] uint(8) = noinit;
      evaluate_gpu(parents_d, labels_d); //kernel
      labels = labels_d; //device-to-host
    }
    generate_children(parents, labels, pool);
  }
}
```

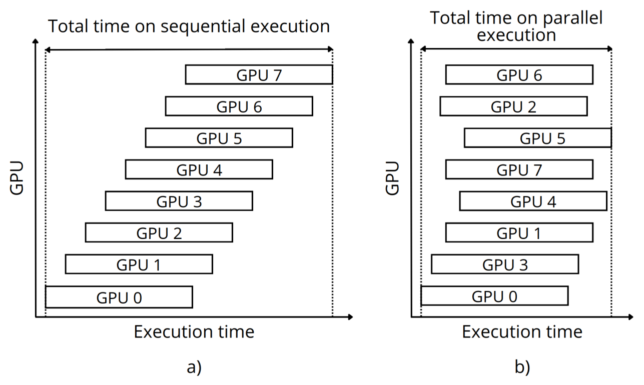

Figure 4: Approaches for launching many GPUs (kernels): a) sequential version; b) parallel version using multi-threading.

offload a chunk of nodes on the GPU when the pool size Q is sufficiently large, i.e., $Q > m$, where m is the minimum number of nodes to offload on the GPU. The number of transferred nodes is $q = min(Q, M)$, where M is the maximum number of nodes to offload on the GPU. m and M are parameters that allow us to avoid the GPU starvation that usually occurs at the beginning and the end of the search process, and to control the size of the data transfer, respectively. When the GPU retrieves the nodes, the latter are then evaluated in parallel, and the non promising nodes are labelled. Finally, the array of labels is sent back to the CPU, which uses it to prune and branch.

Algorithm 1 shows a pseudo-code of the proposed algorithm in Chapel. First of all, we initialize the pool of nodes, as well as the root node (lines 1-3). The pool is implemented as a dynamic-sized stack supporting the basic `pushBack` and `popBack` operations, and each `Node` contains only the information required to generate its children nodes. Then, we iterate over each node in the pool until the latter is empty (line 5). When the pool contains less than m elements, the exploration is done on the CPU, meaning that one node is taken, evaluated, and its children are generated before their insertion back to the pool (lines 6-9). Otherwise, we offload a bulk of nodes on GPU for their evaluation in parallel. The parent nodes are first taken from the pool on the CPU (line 11), offloaded on the GPU (line 14), and their children are then evaluated in parallel (line 16). `labels` contains the results and allows the CPU to generate and prune the children nodes (line 19).

3.2.2. *Multi-GPU approach*

This variant of the previous algorithm exploits multiple GPU devices to allow more computational resources. In addition, we use task-parallelism in order to launch the GPU kernels concurrently, as shown in Fig. 4. Basically, each task will be responsible for handling a specific GPU. More particularly, each task manages a privatized pool of nodes and follows the flowchart of Fig. 3. In order to avoid critical load imbalance between tasks during execution, we preliminary perform a partial search sequentially until we have $G \times m$ nodes in the pool, where G is the number of GPU devices. This set of nodes is then statically distributed between tasks in a round-robin fashion (node l goes

Algorithm 2: Multi-GPU tree-search in Chapel

```
var pool = new Pool();
var root = new Node();
pool.add(root);

//partial search
while (pool.getSize() < G*m) {
  var parent = getNode(pool);
  evaluate_generate_children(parent, pool);
}

coforall taskId in 0..#G with (ref pool) {
  var pool_l = new Pool();
  fillPool(pool_l, pool);

  while pool_l.notEmpty() {
    if (pool_l.getSize() < m) {
      var parent = getNode(pool_l);
      evaluate_generate_children(parent, pool_l);
    }
    else {
      var parents: [] Node = getNodes(pool_l, M);
      var labels: [] uint(8) = noinit;
      on here.gpus[taskId] { //execute on GPU
        const parents_d = parents; //host-to-device
        var labels_d: [] uint(8) = noinit;
        evaluate_gpu(parents_d, labels_d); //kernel
        labels = labels_d; //device-to-host
      }
      generate_children(parents, labels, pool_l);
    }
  }
}
```

Algorithm 3: Single-GPU tree-search in C+CUDA

```
1  Pool pool;
2  Node root;
3  pool.add(root);
4
5  while (pool.notEmpty()) {
6    if (pool.getSize() < m) {
7      Node parent = getNodes(pool);
8      evaluate_generate_children(parent, pool);
9    }
10   else {
11     Node* parents = malloc(q * sizeof(Node));
12     parents = getNodes(pool);
13     Node* parents_d;
14     uint8_t* labels_d;
15     cudaMalloc(parents_d);
16     cudaMalloc(labels_d);
17     cudaMemcpy(parents_d, parents, HostToDevice);
18     evaluate_gpu<<<nBlocks, blockSize>>>(parents_d,
       labels_d);
19     cudaMemcpy(labels, labels_d, DeviceToHost);
20     cudaFree(parents_d);
21     cudaFree(labels_d);
22     generate_children(parents, labels, pool);
23     free(parents);
24     free(labels);
25   }
26 }
```

to GPU l mod G).

Algorithm 2 shows a pseudo-code of the proposed algorithm in Chapel. In contrast to the previous algorithm, lines 6-9 contain the implementation of the partial search on CPU. Then, we use the Chapel's `coforall`-statement to create task parallelism (line 11). The latter creates and launches one task per loop iteration, i.e., here, the number of GPU devices. Its `with`-clause indicates the shared variables between tasks; here, the pool of nodes resulting from the partial search. This pool is then read by each task to fill their privatized pool, hereafter called `pool_l` (lines 12-13). The `fillPool` procedure contains the static load balancing mechanism. Finally, the remaining algorithm is similar to lines 5-21 of Algorithm 1, except that now, each task uses its privatized local pool.

3.3. *CUDA-based baseline implementation*

In this section, we present the CUDA-based baseline implementation. The latter implements the algorithm presented in the previous section.

3.3.1. *Single-GPU implementation*

Algorithm 3 shows a pseudo-code of the single-GPU tree-search algorithm in C+CUDA. The only differences with Algorithm 1 are the explicit calls to the standard procedures from the CUDA programming model: `cudaMalloc` and `cudaFree` for the allocation and deallocation of memory on the GPU (lines 15-16,20-21), respectively, `cudaMemcpy` for the data transfers between the CPU host and the GPU device (lines 17,19), and the `<<<...>>>` syntax for the configuration of the kernel launch (line 18).

In addition to this, we also have to handle CUDA-specific syntax not shown in these code snippets, such as the `__global__`, `__host__`, and `__device__` qualifiers which control where a function is to be executed (CPU or GPU) and whether it can be called from both depend on their functionality and requirements. Moreover, the multiple threads of the GPU SIMT architecture are handled using built-in unique identifiers for threads and blocks, such as `blockIdx.x`, `blockDim.x`, and `threadIdx.x`.

3.3.2. *Multi-GPU implementation*

Algorithm 4 shows a pseudo-code of the multi-GPU tree-search in C+OpenMP+CUDA. OpenMP is used to handle multi-threading and, more particularly, the concurrent launching of GPU kernels, as already shown in Fig. 4. The OpenMP's `#pragma omp parallel for` construct is used similarly to the Chapel's `coforall` in order to create one thread per loop iterations (lines 11-12), here again the number of GPU devices. This is followed by the `cudaSetDevice` CUDA procedure to map each thread to a device, similarly to the `here.gpus` built-in variable of Chapel. The remaining of the algorithm is similar to lines 5-26 of Algorithm 3.

4. Experimental evaluation

4.1. *Experimental protocol*

We evaluate our algorithm on the N-Queens problem, which consists of placing N non-attacking queens in an $N \times N$ chessboard, *i.e.*, two queens must not share the same row, column, or diagonal. Fig. 5 illustrates a valid solution to the 4-Queens problem. More precisely, we determine the exact number of solutions of the instances. The largest known solution count to date is for the $N = 27$ instance that contains approximately $235e15$ solutions [17].

It is worth mentioning that we use the N-Queens problem as a proof-of-concept that motivates further improvements in solving related COPs. As those problems generally require more computational effort per node, we introduce a parameter g to control the granularity of the N-Queens instances. More particularly, each node evaluation contains g safety check(s). This allows one to investigate different scenarios and is implemented as follows:

```
for each node to evaluate:
    for i from 0 to g:
        check if node safe;
    if safe:
        process it;
```

In this paper, the following experiments are performed on different GPU architectures:

Algorithm 4: Multi-GPU tree-search in C+OpenMP+CUDA

```
Pool pool;
Node root;
pool.add(root);

// partial search
while (pool.getSize() < G*m) {
  Node parent = getNode(pool);
  evaluate_generate_children(parent, pool);
}

#pragma omp parallel for num_threads(G) shared(pool)
for (int taskId = 0; taskId < G; taskId++) {
  cudaSetDevice(taskId);
  Pool pool_l;
  fillPool(pool_l, pool);

  while (pool_l.notEmpty()) {
    if (pool_l.getSize() < m) {
      Node parent = getNodes(pool_l);
      evaluate_generate_children(parent, pool_l);
    }
    else {
      Node* parents = malloc(q * sizeof(Node));
      parents = getNodes(pool_l);
      Node* parents_d;
      uint8_t* labels_d;
      cudaMalloc(parents_d);
      cudaMalloc(labels_d);
      cudaMemcpy(parents_d, parents, HostToDevice);
      evaluate_gpu<<<nBlocks, blockSize>>>(
    parents_d, labels_d);
      cudaMemcpy(labels, labels_d, DeviceToHost);
      cudaFree(parents_d);
      cudaFree(labels_d);
      generate_children(parents, labels, pool_l);
      free(parents);
      free(labels);
    }
  }
}
```

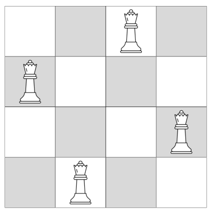
Figure 5: Valid solution of the 4-Queens problem.

Table 1: Summary of the Chapel environment configuration for compilation and execution on GPU.

Variable	Value
`CHPL_RT_NUM_THREADS_PER_LOCALE`	8
`CHPL_RT_NUM_GPUS_PER_LOCALE`	8
`CHPL_LOCALE_MODEL`	*gpu*
`CHPL_GPU`	*nvidia* / *amd*
`CHPL_GPU_ARCH`	*sm_70* / *gfx*906
`CHPL_GPU_MEM_STRATEGY`	*array_on_device*

- *single-GPU*: We evaluate and compare the execution time of our Chapel single-GPU implementation against the baseline implementations.
- *multi-GPU*: We evaluate and compare the performance of our Chapel multi-GPU implementation against the baseline implementations in terms of speed-up.

4.2. Parameters settings

The following system configurations are used:

- Nvidia Tesla V100: Intel Xeon E5-2698 v4 (Broadwell) @ 2.20GHz, 512 GiB, equipped with 8 Nvidia Tesla V100-SXM2-32GB (32 GiB);
- AMD Radeon Instinct MI50: AMD EPYC 7642 (Zen 2) @ 2.3GHz, 512 GiB, equipped with 8 AMD Radeon Instinct MI50 32GB (32 GiB).

The implementation is based on Chapel 1.33.0 and uses LLVM 15.0.7 as Chapel's back-end compiler. Moreover, Table 1 summarizes the environment configurations used for compilation and execution on GPU. The CUDA baseline is compiled and executed using `gcc` 10.4.0 and CUDA 11.7.1. While Chapel allows one to compile and execute the same code on both Nvidia and AMD GPU architectures without any change in the code[2], we translated our CUDA-based source code into portable HIP C++ automatically using the `hipify-perl` tool [18] to target AMD GPUs. In that case, the ROCm/HIP `hipcc` 4.5.0 compiler is used.

In this work, we consider the N-Queens instance from 14 to 17. Table 2 summarizes the solution count of these instances along with the tree size obtained with our algorithms. The smallest instance ($N = 14$) exhibits a tree size of $27e6$ nodes and is solved sequentially in less than a second, while the biggest one ($N = 17$) contains $8e9$ nodes and requires approximately five minutes. In the following, each experiment is performed 5 times, and the results shown correspond to the average. Additionally, in the absence of explicit mention, the variable g is assigned to 1.

4.3. Experimental results

4.3.1. Single-GPU

We can see in Fig. 6 the normalized execution time of our Chapel implementation compared to the baselines. The re-

[2]Chapel, however, requires different environment configurations.

Table 2: Summary of the N-Queens instances solved in this paper.

N	solution count	tree size
14	365,596	27,358,552
15	2,279,184	171,129,071
16	14,772,512	1,141,190,302
17	95,815,104	8,017,021,931

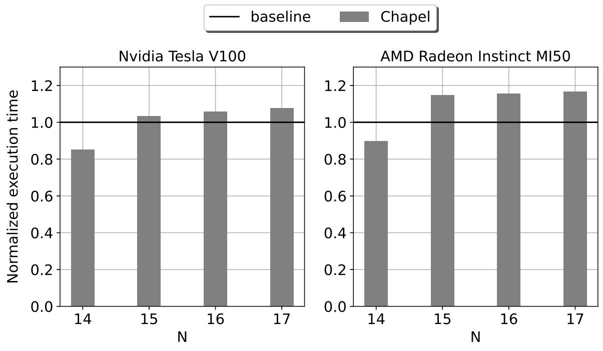

Figure 6: Normalized execution time of the Chapel implementation compared to the CUDA and HIP baselines on the Nvidia V100 and AMD MI50 GPU architectures, respectively.

sults are given for both the Nvidia and AMD GPU architectures. Using the Nvidia GPU, the Chapel implementation is surprisingly 15% faster than the CUDA baseline solving the 14-Queens instance, while it is between 3% and 8% slower on the bigger instances. Using the AMD GPU, the Chapel implementation is 10% faster than the HIP-based one solving the 14-Queens instance, but then 16% slower on average on the other instances. Chapel's expressive and user-friendly syntax results in a more readable and maintainable code, as seen in Section 3.2.1, but may incur a performance cost for certain computational tasks, particularly those with fine-grained parallelism, where the more explicit control provided by CUDA and HIP lead to optimizations not easily achieved in Chapel. Finally, the observed twofold increase in performance degradation on the AMD GPU architecture compared to Nvidia is likely attributable to the more recent and potentially less optimized support for AMD GPUs in Chapel (introduced in version 1.30.0), as opposed to the more established and optimized support for Nvidia GPUs (available since version 1.26.0).

4.3.2. Multi-GPU

Fig. 7 shows the speed-up achieved by the Chapel implementation compared to the CUDA-based baseline on the Nvidia GPU architecture. We can first see that at the finest granularity ($g = 1$), the performance of the Chapel implementation is quite limited. Indeed, we note a speed-up of 70% of the linear one using 4 GPUs, while there is no performance gain using more GPUs. In contrast, we observe in the coarser-grained experiment ($g = 10,000$) that the performance results are much better since we achieve up to 75% of the linear speed-up using 8 GPUs. When the granularity is large, the computational time tends to be equivalent or larger than the generated parallel overheads, which produces better speed-up results. Us-

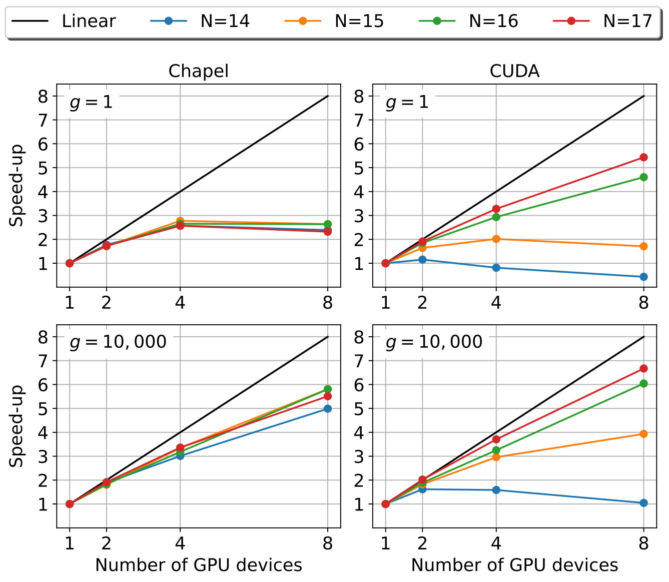

Figure 7: Speed-up achieved by our Chapel implementation compared to the CUDA-based implementation executed on the Nvidia V100 GPUs. Results are shown for different instances and granularities.

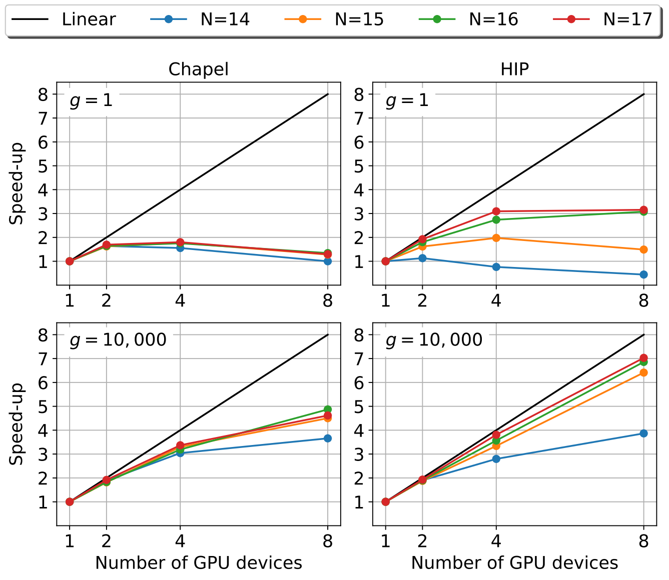

Figure 8: Speed-up achieved by our Chapel implementation compared to the HIP-based implementation executed on the AMD MI50 GPUs. Results are shown for different instances and granularities.

ing the CUDA baseline, we achieves up to 69% of the ideal speed-up on 8 GPUs for the biggest instance ($N = 17$) and the finest granularity, while it reaches up to 84% at coarser-grain. Compared to Chapel, the CUDA implementation is scalable in terms of GPU counts for large instances even at low granularity, meaning that the low-level programming of CUDA allows minimal overheads. For its best results, Chapel achieves 80% of the baseline speed-up using 8 GPUs.

Another major observation is that increasing the instance size does not significantly increase the speed-up using Chapel, while this is the case using CUDA. This behaviour happens in scenarios where the overheads generated by the execution of the program are proportional to the workload. In particular, increasing the number of nodes in the tree increases the communication between the CPU hosts and the GPU devices, as well as the access to the work pool, which increases the general overheads of the Chapel program.

The previous experiments have been conducted similarly on the AMD GPU architecture. Fig. 8 shows the results compared to the HIP-based implementation, which are quite similar to the ones obtained comparing Chapel to CUDA on the Nvidia architecture. However, we note in this configuration that the performance gap between Chapel and the baseline is a little further away than on the Nvidia GPUs. For its best results, Chapel achieves 71% of the baseline speed-up using 8 GPUs. This can still be related to the less optimized Chapel support for AMD GPUs than for Nvidia ones, as observed in the single-GPU setting.

5. Conclusions and future works

This paper provides the design and implementation of a GPU-accelerated tree-search algorithm in Chapel. The latter is based on a generic multi-pool approach including a static load balancing mechanism. As a proof-of-concept, it has been instantiated on the backtracking method and experimented on the N-Queens problem. The approach has been evaluated in terms of performance and compared to CUDA- and HIP-based baseline implementations on a Nvidia and an AMD GPU architecture, respectively.

In a single-GPU setting, we demonstrate that the Chapel's high-level of abstraction generates a performance loss of only 8% on the Nvidia architecture, while it is at least twice as much on the AMD GPU. Moreover, the Chapel multi-GPU version is outperformed by its counterparts on fine-grained problem instances, but achieves on coarse-grained ones up to 80% and 71% of speed-up on the Nvidia and AMD GPUs, respectively. In conclusion, we demonstrate experimentally in the context of GPU-accelerated tree-search that Chapel's high-level expressive and user-friendly syntax results in a more readable and maintainable code but incurs a performance cost for computational tasks, especially with fine-grained parallelism, where the more explicit control provided by CUDA or HIP leads to optimizations not easily achieved in Chapel. Nevertheless, we can expect the Chapel implementation to be quite efficient on big COPs, since those problems generally involve a large computational workload.

In the future, we plan to optimize our Chapel code further in order to reduce the performance gap with the baseline implementations. In addition, we are looking to instantiate our approach on the B&B tree-search method to solve large COPs, such as PFSP. Some of its best-known instances have remained unsolved for 25 years and still represent a substancial challenge for HPC. In order to face such a large computational workload, we plan to extend the current approach to distributed multi-GPU systems, involving many more GPU devices. This will probably require the design and implementation of a more scalable load balancing mechanism. Finally, we also intend to investigate the use of the Chapel's `DistBag_DFS` [19] predefined data structure, which implements a parallel-safe distributed dynamic multi-pool for large scale tree-based applications.

Acknowledgement

The experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several Universities as well as other organizations. This work is supported by the Agence Nationale de la Recherche (ref. ANR-22-CE46-0011) and the Luxembourg National Research Fund (ref. INTER/ANR/22/17133848), under the UltraBO Project.

Data availability statement

All code written in support of this publication is publicly available on GitHub at `https://github.com/Guillaume-Helbecque/GPU-accelerated-tree-search-Chapel` and archived on Zenodo [20].

References

[1] E. Krishnasamy, "Hybrid CPU-GPU Parallel Simulations of 3D Front Propagation," 2014, Dissertation, Linköping University.

[2] E. Krishnasamy, M. Sourouri, and X. Cai, "Multi-GPU Implementations of Parallel 3D Sweeping Algorithms with Application to Geological Folding," *Procedia Computer Science*, vol. 51, pp. 1494–1503, 2015.

[3] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and Scalability of GPU-Based Convolutional Neural Networks," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 317–324.

[4] M. E. Lalami and D. El-Baz, "GPU Implementation of the Branch and Bound Method for Knapsack Problems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, pp. 1769–1777.

[5] N. Melab, I. Chakroun, M. Mezmaz, and D. Tuyttens, "A GPU-accelerated Branch-and-Bound Algorithm for the Flow-Shop Scheduling Problem," in *2012 IEEE International Conference on Cluster Computing*, 2012, pp. 10–17.

[6] I. Chakroun, N. Melab, M. Mezmaz, and D. Tuyttens, "Combining multi-core and GPU computing for solving combinatorial optimization problems," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1563–1577, 2013.

[7] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens, "IVM-based parallel branch-and-bound using hierarchical work stealing on multi-GPU systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 9, p. e4019, 2017.

[8] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, 2007, p. 24–32.

[9] T. M. Mintz, O. Hernandez, and D. E. Bernholdt, "A Global View Programming Abstraction for Transitioning MPI Codes to PGAS Languages," in *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, 2014, pp. 120–133.

[10] D. Cunningham, R. Bordawekar, and V. Saraswat, "GPU programming in a high level language: compiling X10 to CUDA," in *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, 2011, pp. 1–10.

[11] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou, "Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation," in *Languages and Compilers for Parallel Computing*, 2011, pp. 151–165.

[12] A. Hayashi, S. R. Paul, and V. Sarkar, "A Multi-Level Platform-Independent GPU API for High-Level Programming Models," in *High Performance Computing. ISC High Performance 2022 International Workshops*, 2022, pp. 90–107.

[13] B. L. Chamberlain, E. Ronaghan, B. Albrecht, L. Duncan, M. P. Ferguson, B. Harshbarger, D. Iten, D. Keaton, V. Litvinov, P. Sahabu, and G. Titus, "Chapel Comes of Age: Making Scalable Programming Productive," 2018, Cray Inc.

[14] T. Carneiro, N. Melab, A. Hayashi, and V. Sarkar, "Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators," in *18th International Conference on High Performance Computing & Simulation*, 2021.

[15] A. Grama and V. Kumar, "Parallel Search Algorithms for Discrete Optimization Problems," *ORSA Journal on Computing*, vol. 7, no. 4, pp. 365–385, 1995.

[16] B. Gendron and T. G. Crainic, "Parallel Branch-and-Branch Algorithms: Survey and Synthesis," *Operations Research*, vol. 42, no. 6, pp. 1042–1066, 1994.

[17] T. B. Preußer and M. R. Engelhardt, "Putting Queens in Carry Chains, №27," *J Sign Process Syst*, vol. 88, pp. 185–201, 2017.

[18] "HIPIFY Documentation," 2024, Advanced Micro Devices Inc., https://rocm.docs.amd.com/projects/HIPIFY/en/latest/.

[19] G. Helbecque, J. Gmys, N. Melab, T. Carneiro, and P. Bouvry, "Parallel distributed productivity-aware tree-search using Chapel," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 27, p. e7874, 2023.

[20] G. Helbecque, E. Krishnasamy, N. Melab, and P. Bouvry, "GPU-accelerated tree-search in Chapel," Zenodo, version 1.0, 2024.