Heron: Automatically Constrained High-Performance Library Generation for Deep Learning Accelerators

Jun Bi
USTC
SKL of processors, ICT, CAS
Cambricon Technologies
China

Qi Guo*
SKL of processors, ICT, CAS
Beijing, China

Xiaqing Li
SKL of processors, ICT, CAS
Beijing, China

Yongwei Zhao
SKL of processors, ICT, CAS
Beijing, China

Yuanbo Wen
SKL of processors, ICT, CAS
Beijing, China

Yuxuan Guo
USTC
SKL of processors, ICT, CAS
Cambricon Technologies
China

Enshuai Zhou
USTC
SKL of processors, ICT, CAS
Cambricon Technologies
China

Xing Hu
SKL of processors, ICT, CAS
Beijing, China

Zidong Du
SKL of processors, ICT, CAS
Beijing, China

Ling Li
Institute of software, CAS
Beijing, China

Huaping Chen
USTC
Hefei, China

Tianshi Chen
Cambricon Technologies
China

ABSTRACT

Deep Learning Accelerators (DLAs) are effective to improve both performance and energy efficiency of compute-intensive deep learning algorithms. A flexible and portable mean to exploit DLAs is using high-performance software libraries with well-established APIs, which are typically either manually implemented or automatically generated by exploration-based compilation approaches. Though exploration-based approaches significantly reduce programming efforts, they fail to find optimal or near-optimal programs from a large but low-quality search space because the massive inherent constraints of DLAs cannot be accurately characterized.

In this paper, we propose Heron, a novel exploration-based approach, to efficiently generate high-performance libraries of DLAs. The key is to automatically (rather than manually) enforce massive sophisticated while accurate constraints through the entire program generation including constrained space generation and constrained space exploration. By conducting static analysis on compute, sophisticated constraints are automatically generated to properly characterize inherent constraints of DLAs, and thus greatly prune invalid program candidates to produce a high-quality constrained search space. To efficiently explore the resultant search space, we further propose a novel *constraint-based genetic algorithm*, which features that the evolutionary process is conducted on formulated constraint satisfactory problems instead of concrete solutions. Thus, the sophisticated constraints of the search space are strictly preserved during the entire exploration process. We conduct extensive experiments on 3 representative DLAs, i.e., NVIDIA TensorCore, Intel DL Boost Acceleration, and TVM Versatile Tensor Accelerator. Experimental results demonstrate that Heron averagely achieves 2.71× speedup over four state-of-the-art automatic generation approaches. Also, compared to vendor-provided hand-tuned libraries, Heron achieves 2.00× speedup on average.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Source code generation**.

KEYWORDS

code generation, compiler optimization, tensor computation

ACM Reference Format:
Jun Bi, Qi Guo, Xiaqing Li, Yongwei Zhao, Yuanbo Wen, Yuxuan Guo, Enshuai Zhou, Xing Hu, Zidong Du, Ling Li, Huaping Chen, and Tianshi Chen. 2023. Heron: Automatically Constrained High-Performance Library Generation for Deep Learning Accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages.
https://doi.org/10.1145/3582016.3582061

*Corresponding author

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582061

1 INTRODUCTION

Deep Learning Accelerators (DLAs), which offer specialized hardware architecture to greatly improve the efficiency of widely-used deep learning algorithms, have received extensive attention in both academia and industry. In academia, started with DianNao series [23, 27, 28], many DLAs targeting different algorithms [21, 48, 57] or with new technologies [30, 36] emerge. In industry, DLAs including NVIDIA TensorCore [11], Google TPU [44], and Cambricon MLU [3], are developed for commercial use.

To embrace the diversity of DLAs, high-performance software libraries with well-established APIs are served as flexible, portable, and efficient solutions for programmers to exploit high computational abilities. For example, NVIDIA offers CUDA Deep Neural Network library (cuDNN) [9] to utilize TensorCore, and Intel provides oneAPI Deep Neural Network library (oneDNN) [13] to leverage DL Boost acceleration [7]. Unfortunately, these libraries are manually implemented and tuned for high performance, which requires a deep understanding of algorithms, compilers, and hardware architectures with intense human efforts. In practice, the development cycle typically takes months or even years for widely-used DL operators, which cannot meet the stringent time-to-market requirement of hardware and also fall behind the fast evolution of algorithms.

To reduce development costs, exploration-based compilation approaches [17, 25, 50, 67, 71–73] have recently emerged as one of the most effective solutions for automatic generation of low-level programs of DL operators. The key idea is to formulate the program generation as the exploration in a large space consisting of program candidates, which are built either with manually implemented [25, 26, 50, 67] or automatically generated [71–73] templates along with multiple tunable parameters (e.g., tiling factors). Despite delivering comparable or even better performance against hand-tuned counterparts on CPUs and GPUs (with CUDA core), these approaches cannot perform efficiently on various DLAs because they fail to find optimal or near-optimal programs in the search space. Taking the convolution layers in VGG-16 [64] as an example, on TensorCore, state-of-the-art exploration-based approaches such as AutoTVM [26] cannot find the optimal program in the search space, and the performances of generated programs only achieve 46% of that of cuDNN on average.

The inefficiency of existing exploration-based approaches stems from low-quality search spaces, which are large but nearly all the program candidates are invalid to meet the architectural constraints of DLAs. Taking the matrix multiply operator of size $32\times1000\times4096$ as an example, about 95% of the total programs generated by AutoTVM are invalid for TensorCore, though they can be executed on general-purpose architectures such as CPUs and GPUs. The reason is that DLAs typically have inherent architectural constraints and thus impose non-trivial requirements on the programs, e.g., the computational unit of TensorCore has strict requirements on the dimensions of computed matrices, while general-purpose architectures do not have such requirements. To improve the quality of search spaces, it is necessary to accurately constrain the search space by eliminating invalid programs as much as possible.

However, it is quite difficult to accurately constrain the search space because of the diverse and complicated architectural constraints of DLAs. Figure 1(a) shows that a search space can be easily specified with a few simple constraints, represented by the straight lines, while it excludes many valid programs and the optimal one. To include all valid programs, in Figure 1(b), the search space can be greatly improved by enforcing hundreds of sophisticated constraints, which requires domain knowledge and expertise to accurately characterize architectural constraints of DLAs by imposing appropriate constraints on computation. In addition to accurately constraining the search space, another key challenge is that all such constraints should be strictly satisfied during the entire exploration process to find the optimal programs efficiently.

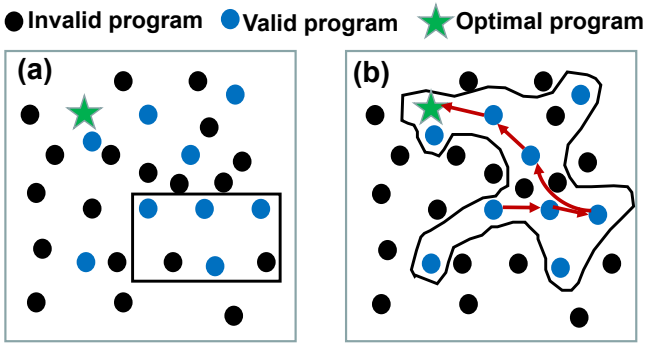

Figure 1: Comparison of two constrained search spaces: (a) a regular space with a few simple constraints excludes many valid programs, and (b) an irregular space with massive sophisticated constraints includes all valid programs. The irregular space poses a key challenge to the search algorithms in that such constraints should be preserved during the entire exploration process.

To address the above challenges, in this paper, we propose an automatically constrained exploration-based approach, i.e., Heron, to generate high-performance libraries for DLAs. The key is to automatically (rather than manually) enforce massive sophisticated while accurate constraints through the entire program generation including constrained space generation and constrained space exploration. During constrained space generation, the first challenge of generating accurate constraints is addressed by conducting static analysis on compute. Specifically, we summarize multiple general rules on variables that are mainly related to key architectural constraints such as requirements of functional units and capacity of on-chip memory and then apply such rules during static analysis to automatically produce both optimization schedules and sophisticated constraints on computation description. During constrained space exploration, the second challenge of exploring the irregular space is addressed by a novel *constraint-based genetic algorithm (CGA)*. Concretely, the original space is formulated as a constraint satisfaction problem (CSP) to generate a set of new CSPs. Then the evolutionary operations (e.g., crossover and mutation) are directly enforced on such CSPs instead of concrete solutions in traditional GAs. Thus, all the initial constraints can be strictly preserved during the entire exploration process to find optimal programs.

To evaluate Heron, we conduct extensive experiments on 3 representative DLAs, i.e., NVIDIA TensorCore, Intel DL Boost, and TVM Versatile Tensor Accelerator (VTA) [55], with a large number of benchmarks (including operators and networks). On TensorCore, compared to 4 state-of-the-art automatic generation paradigms

Table 1: Examples of schedule primitives in TVM

Name	Description
split	split a loop into multiple sub-loops $loop_1, loop_2 = stage.split(loop, split\_factor)$
fuse	merge multiple loops into one loop $loop = stage.fuse(loop_1, loop_2)$
cache	use shared memory for inputs or outputs $cache\_read(stage, "shared")$
unroll	unroll the loop for several times $stage.unroll(loop, unroll\_length)$
compute_at	$stage_2$ is fused into $stage_1$ at *location* $stage_2.compute\_at(stage_1, location)$
tensorize	replace compute to utilize hardware intrinsic $stage.tensorize(loop, intrin\_gemm(m, n, k))$

Table 2: Examples of rules in Ansor, where the state S is the current schedule template and related program, i is the index of current node of the DAG, and S' is the updated template and transformed program.

Rule Name	Condition and Application
Always-Inline	IsStrictInlinable(S, i) $S' \leftarrow Inline(S, i); i' \leftarrow i - 1$
Multi-level-Tiling	HasDataReuse(S, i) $S' \leftarrow MultiLevelTiling(S, i); i' \leftarrow i - 1$
Add-Cache-Stage	HaseDataReuse(S, i)& ¬ HasFusibleConsumer(S, i) $S' \leftarrow AddCacheWrite(S, i); i' \leftarrow i$
User-Defined-Rule	*user-defined transformations and conditions*

including exploration-based approaches (i.e., AutoTVM [26], Ansor [71], and AMOS [72]) and a polyhedral compiler (i.e., AKG [70]), the performance gains of Heron range from 1.52× to 2.85×, with 1.89× on average. Compared to hand-tuned libraries including cuDNN/cuBLAS and PyTorch kernel, the performance gains of Heron range from 1.10× to 8.89×, with 2.69× on average. On the DL Boost, compared to AutoTVM, Ansor, and AMOS, the performance gains of Heron range from 2.72× to 12.0×, with 4.57× on average. Compared to hand-tuned libraries such as Intel oneDNN, the performance gains of Heron range from 1.02× to 6.94×, with 1.49× on average. On the VTA, compared to AutoTVM, the performance gains of Heron range from 1.03× to 2.95×, with 2.32× on average.

This paper makes the following contributions:

- **Automatic generation of sophisticated constraints**. We define multiple rules to automatically generate sophisticated while accurate constraints for obtaining a high-quality constrained search space.
- **Constraint-based genetic algorithm**. We formulate the space exploration as optimizing a constrained optimization problem with a novel constraint-based genetic algorithm. *To our best knowledge, this is the first work to conduct the evolutionary process directly on CSPs.*
- **Comprehensive evaluation**. We conduct extensive experiments on 3 representative DLAs, where Heron significantly outperforms both state-of-the-art automatic generation approaches without additional compilation costs and vendor-provided hand-tuned libraries.

2 BACKGROUND AND MOTIVATION

2.1 Deep Learning Compilers

Deep learning compilers emerge as effective techniques to reduce the development efforts for high-performance programs on DLAs. The basic idea is to translate the original models with various operators to native codes on different hardware platforms. Taking the most representative deep learning compiler, i.e., TVM, as an example, original models first undergo graph-level optimizations such as operator fusion and layout transformation. Then, the optimized models (i.e., graphs) are represented by a declarative tensor expression language, where fine-grained optimizations such as loop tiling can be enforced. Finally, the compiler generates native codes for different back-ends, e.g., CUDA for GPU and assembly for CPU.

The fine-grained optimizations in TVM are usually performed via a *schedule* template consisting of a list of schedule primitives, which are program transformations that generate programs from the input *compute* (i.e., description of the tensor computation consisting of multiple stages such as the load stage) with various optimizations (e.g., loop tiling). As listed in Table 1, a schedule primitive consists of the stage to be transformed, the related loops of the stage, and various schedule parameters. Some of the schedule parameters (e.g., $split\_factor$, $unroll\_length$, and $location$) are tunable for better performance. As a result, the combination of possible schedule templates with their tunable parameters can form an extremely large search space of program candidates.

Apparently, the key factors to the optimization efficiency are the quality of the search space and the effectiveness of the search algorithms. As stated, the space is mainly determined by the schedule templates, which can be generated either manually [26, 67] or automatically [71–73]. The well-known search algorithms include random search [26], simulated annealing [26], and genetic algorithm [71]. Among these algorithms, we observe that the genetic algorithm is most widely used in state-of-the-art deep learning compilers [71, 72]. The background of the schedule template generation and the genetic algorithm will be elaborated later.

2.2 Schedule Template Generation

As stated, there are two main approaches for generating the schedule templates.

Manual approaches. These approaches rely on domain experts to manually write schedule templates and specify the range of schedule parameters as well for each compute [25, 26, 50, 67]. Then, the compiler searches for the best parameter configuration for given input shapes and hardware platforms. These approaches can provide high-performance program implementations and are flexible to support newly emerged DLAs. However, it takes non-trivial efforts for compiler designers to write schedule templates and such manually-implemented templates usually result in limited search spaces.

Automatic approaches. These approaches automatically generate schedule templates [71–73] to overcome the aforementioned drawbacks of the manual approaches. Typically, these approaches rely on predefined rules to generate the schedule templates along with their tunable parameters' ranges. As one of the most representative approaches, Ansor [71], first converts the input compute to a naive program and the corresponding compute graph (i.e., directed

acyclic graph, DAG) by directly expanding the loop indices. Then, the naive program is recursively transformed by the *derivation rules*, which are predefined by compiler designers for general-purpose architectures as listed in Table 2, to generate different schedule templates and related programs. Concretely, Ansor traverses the stages (i.e., nodes of the DAG) in reverse topological order and applies the rules if the conditions (e.g., `HasDataReuse(S,i)` indicates that the node i in the current state S has data reuse opportunity such as matrix multiply) are satisfied, so as to generate new schedule templates and related programs (e.g., perform multi-level tiling for data reusable nodes).

Compared to the manual approaches, the automatic approaches not only remove the programming burden of schedule templates but also generate larger search spaces that contain more high-performance programs. However, they do not consider the diverse architectural constraints of DLAs and thus are not efficient for the code generation of DLAs.

2.3 Genetic Algorithm

Genetic algorithm (GA) is a meta-heuristic algorithm for solving optimization problems. GA solves the problem by simulating it as a population's natural evolutionary process where the parameters to be optimized are encoded as a sequence of variables called chromosome. During optimization, GA randomly samples an initial population of chromosomes as the first generation and then iteratively produces new generations by performing *selection*, *crossover*, and *mutation* on the previous generation until given termination conditions (e.g., a time budget) are met. Finally, the chromosome with the highest fitness score, which measures the quality of represented solutions, is decoded as the optimized parameters.

GA is widely used in existing exploration-based compilation approaches [26, 71, 72] for program optimization space exploration. The parameters (i.e., tunable parameters) are first encoded as a chromosome whose fitness score is calculated as the measured [26] or estimated [71, 72] program performance. In each generation, the chromosomes are selected to survive by the Roulette-wheel selection [37] which selects chromosomes with probability $p_i = \frac{f_i}{\sum f_i}$ where f_i is the fitness score. Offspring chromosomes are then generated by performing crossover (e.g., single point crossover in [26]) and mutation (e.g., tile size or unroll length mutation in [71]) and form the new generation combined with their parents.

Although GA can intuitively optimize the parameters in exploration-based compilation, it does not take the constraints into consideration to guide the exploration process because GA is essentially an unconstrained optimization approach [32].

2.4 Motivation

To understand the design principle of Heron, we examine the challenges of defining and exploring the search spaces for DLAs and obtain three key observations as follows.

Observation #1: *There exists a large number of diverse and complicated architectural constraints in DLAs.*

Table 3 lists typical architectural constraints of representative DLAs including TensorCore, DL Boost, VTA, TPU [44], and Cambricon [49]. Obviously, the constraints vary significantly. For example, TensorCore has relatively strict constraints on the dimensions of

Table 3: Constraint examples of various DLAs.

Platforms	Constraints	Category
TensorCore	$m * n * k == 4096$	computation size
	$m, n, k \in \{8, 16, 32\}$	computation size
	$shared\_mem \leq 48K$	memory capacity
	$vector\_length \in \{1, 2, 4, 8\}$	memory access
DL Boost	$m, n, k == 1, 16, 4$	computation size
VTA	$m, n, k == 1, 16, 16$	computation size
	$2 \leq access\_cycle$	memory access
	$input\_buffer \leq 32K$	memory capacity
	$weight\_buffer \leq 256K$	memory capacity
	$output\_buffer \leq 128K$	memory capacity
TPU	$m, n, k == 1, 256, 256$	computation size
	$m \times 256 \leq 4M$	memory capacity
Cambricon	$V_{out} \times 3 \leq 64K$	memory capacity
	$V_{out} + V_{out} \times V_{in} + V_{in} \leq 768K$	memory capacity

Table 4: Variables for describing GEMM's constraints on TensorCore.

	Architectural Constraint	Loop Length	Tunable Parameter	Others
Numbers	10	82	30	51

Table 5: Number of variables and constraints used for space description.

	GEMM	BMM	C1D	C2D	C3D
Variables	173	236	236	304	363
Constraints	372	529	547	702	861

matrices while Cambricon relaxes the constraints due to more flexible functional units. The constraints of DLAs stem from limitations of on-chip resources (e.g., scratchpad memory and processing elements) and can be grouped into three categories including computation size (e.g., fixed computation sizes for DL Boost), memory capacity (e.g., limited buffer sizes for VTA), and memory access (e.g., aligned load/store vectorization for TensorCore). If constraints are not satisfied, the generated programs result in a compilation or run-time error which significantly degrades the efficiency of the exploration process.

Furthermore, there exists a gap between the architectural constraints and tunable parameters, e.g., the constraints of on-chip memory capacity do not directly associate with tunable parameters, which requires enforcing such constraints on various parameters in an indirect and compositional manner. Hence, the diverse and complicated architectural constraints make it challenging for practitioners to accurately describe the parameter search spaces.

Observation #2: *High-quality search spaces are hard to be accurately described with a small number of intuitive hand-written constraints.*

To describe the relationship between architectural constraints and tunable parameters, it requires to declare massive variables with complicated constraints. Take the GEMM operator (i.e., $C[i, j]$ += $A[i, r] * B[r, j]$) as an example, the computation on TensorCore mainly consists of five stages:

$$\begin{gathered} \text{global} \xrightarrow{stage_1} \text{shared} \xrightarrow{stage_2} \text{register} \xrightarrow{stage_3} \\ \text{TensorCores} \xrightarrow{stage_4} \text{shared} \xrightarrow{stage_5} \text{global} \end{gathered} \tag{1}$$

where $stage_1$ and $stage_2$ are for data loading, $stage_3$ is for computation, and $stage_4$ and $stage_5$ are for storing the results. To describe

the architectural constraints, 10 dedicated variables are declared (e.g., m, n, k for the fixed computation size) and the related constraints are specified (e.g., $m * n * k == 4096$ & $m \in \{8, 16, 32\}$ &...). Then, these architectural constraints confine the loop lengths, which requires 82 loop lengths related variables (e.g., $stage_3.i_6$ refers to the length of loop i_6 at $stage_3$) with constraints (e.g., $m == stage_3.i_6$ & $stage_3.i == stage_3.i_6 * ... * stage_3.i_0$ &...). After that, the loop lengths constrain the choice of tunable parameters, which requires 30 tunable parameter-related variables (e.g., $tile.stage_3.i_6$ for the tile size of $stage_3$ to generate i_6) with constraints (e.g., $tile.stage_3.i_6 == stage_3.i_6$). Finally, 51 other variables are declared to help describe the constraints (e.g., $m \in \{8, 16, 32\}$ is expressed as $m == 8 \mid m == 16 \mid m == 32$ where $m == x$ is a boolean variable). In this case, there are 173 variables and 372 constraints in total, and Table 4 lists the breakdown of such variables.

In addition to the GEMM operator, Table 5 further lists the number of variables and constraints of other operators for accurately describing the irregular search spaces. The numbers of variables and constraints vary for different operators, e.g., the C3D (3D convolution) operator needs more constraints than the C1D (1D convolution) operator since it is more complicated. Even for the same operator, the constraints are different when the shapes of the input tensors vary. Apparently, writing such massive and sophisticated constraints by hand is infeasible even for domain experts.

Observation #3: *Existing search algorithms fail to explore such high-quality while irregular search space efficiently.*

Although traditional simulated annealing algorithms (SA) and genetic algorithms (GA) work well in previous work [26, 71], they fail to perform better than a random searching algorithm (RAND) in such high-quality while extremely irregular search spaces. Figure 2(a) illustrates that SA easily gets stuck at the local optimum at the early stage of exploration, because the valid programs can only be obtained by changing values of less-constrained tunable parameters such as `unroll`. Figure 2(b) shows that the exploration process of GA is almost random, because GA fails to generate valid programs after crossover and mutation, which incurs frequent random restarts during exploration. All these inefficiencies stem from not taking the massive constraints into consideration when exploring such irregular search spaces.

In summary, the above observations motivate us to employ an automatic rather than manual approach to enforce sophisticated while accurate constraints to not only search space generation but also space exploration.

3 DESIGN OVERVIEW

To automatically enforce constraints through program generation, Heron contains two stages, i.e., generation and exploration, and implements four modules (see Figure 3), including *Space Generator*, *Space Explorer*, *DLA Measurer*, and *Cost Model*. The first module is responsible for the generation stage and the other three modules are used for the exploration stage.

Space Generator. To generate the search space, *Space Generator* takes the compute as the input and outputs a space defined by tunable parameters and their constraints. Specifically, the *Space Generator* mainly performs two generation steps including *Schedule Template Generation* and *Constraint Generation* using *Generation Rules*. The *Generation Rules* contain special rules for generating the schedule templates and the constraints. Based on the compute, the *Schedule Template Generation* generates a schedule template consisting of schedule primitives whose tunable parameters form an initial search space. By analyzing the generated schedule primitives, the *Constraint Generation* then generates massive constraints which can be used to limit the range of the tunable parameters and finally transform the search space into a constrained one. More details are provided in Section 4.

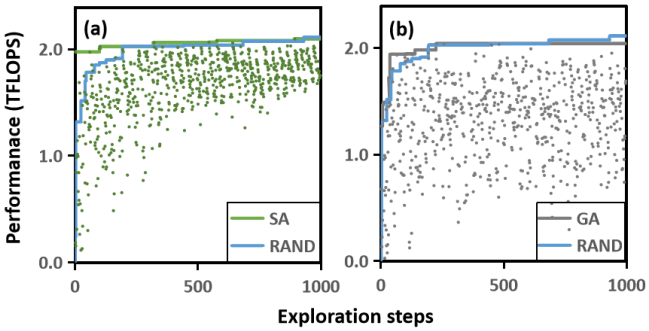

Figure 2: Comparison of random search (i.e., RAND) with simulated annealing (i.e., SA) and genetic algorithm (i.e., GA) in an irregular search space. The points represent measured performances. RAND randomly samples valid parameter configurations under the constraints using a solver [4]. GA and SA adopt the same setups as [26] and use RAND to generate initial valid programs.

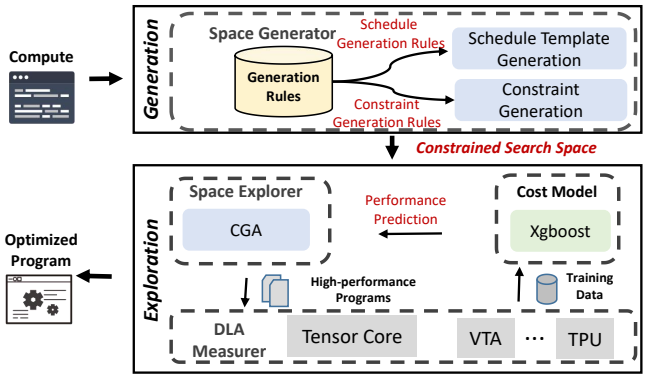

Figure 3: The overview of Heron, which contains two main stages: generation and exploration. The generation stage contains Space Generator, and the exploration stage consists of Space Explorer, DLA Measurer, and Cost Model.

Space Explorer. It conducts the exploration in the constrained search space with a constraint-based genetic algorithm (i.e., CGA) to obtain the programs with optimal or near-optimal performance. CGA uses predicted performance numbers as the fitness scores and outputs a set of high-performance programs for hardware measurements. More details of CGA are provided in Section 5.

DLA Measurer. It measures the accurate performances (i.e., the latency) of the generated programs. Concretely, it measures a single program multiple times to obtain the average execution

time as the final performance. To obtain stable performance, Heron conducts such measurement by adopting different configurations for different DLAs. Finally, the measured performance is recorded by Heron for later cost model training.

Cost Model. It estimates the performance of the programs quickly via different designed features [17, 26, 71]. Although the employed cost model (i.e., XGBoost [24]) is similar to AutoTVM's [26], Heron uses different features taken from defined variables of the *Constraint Generation* step. These variables contain different kinds of important information, such as the lengths of loops, the vector length, and the memory usage. More importantly, the values of the defined variables can be obtained without any compilation, which makes the exploration process faster.

4 CONSTRAINED SPACE GENERATION

In Heron, the representation of the constrained search space is formulated as a constraint satisfaction problem called $CSP_{initial}$. Algorithm 1 shows the basic generation process of $CSP_{initial}$, which requires a naive program (i.e., P) of the given compute and its corresponding computation graph (i.e., DAG) as the inputs. Then two key steps are enabled to generate the schedule template and the constraints respectively. Finally, the algorithm offers $CSP_{initial}$ as the output that consists of generated constraints as the constrained search space.

Algorithm 1 Constrained Search Space Generation

```
1:  Input: P, DAG
2:  Output: CSP_initial
3:  Step 1. schedule template generation
4:  nodes ← post_order_traverse(DAG)
5:  template ← [ ]; nodes_scheduled ← ϕ
6:  while nodes ≠ ϕ do
7:      node ← nodes.pop()
8:      for rule ∈ predefined_schedule_rules do
9:          if rule.condition(P, node) == True then
10:             P, DAG, primitives, next ← rule.apply(P, node)
11:             template.append(primitives)
12:             if next then
13:                 break
14:     nodes_scheduled ← nodes_scheduled ∪ {node}
15:     nodes ← post_order_traverse(DAG)
16:     nodes ← nodes \ nodes_scheduled
17: Step 2. constraints generation
18: CSP_initial ← ϕ
19: for primitives ∈ template do
20:     for rule ∈ predefined_constraint_rules do
21:         if rule.condition(schedule) == True then
22:             constraints ← rule.apply(primitives)
23:             CSP_initial ← CSP_initial ∪ constraints
24: return CSP_initial
```

Regarding the schedule template generation (Step-1), Heron traverses the computation stages (i.e., DAG's unscheduled nodes) in reverse topological order and applies predefined schedule generation rules if their conditions are satisfied. For a given node, Heron performs the transformations by iterating over all different types of rules. In each iteration, Heron checks the rule's condition according to the current program and the graph node. Here, if the condition is satisfied, the rule is applied correspondingly based on the program and the node, resulting in four outputs as shown in line 10 of Algorithm 1: (1) a new program after transformation; (2) a new computation DAG after transformation; (3) new schedule primitives for updating the schedule template; and (4) a predicate *next* (e.g., $i' == i-1$) which is true if the next node should be transformed after the current step. If all the applicable transformations are performed or the *next* is set to be true, the transformation will be terminated and the given node will be inserted into scheduled nodes either. After the transformation of the given node, Heron updates the set of unscheduled nodes according to the current DAG and the set of scheduled nodes. Repeating the above process, Heron generates the schedule template until all nodes are processed.

Table 6: Schedule generation rules and the related conditions.

No.	Rule	Condition and Application
S1	Tensorize	Tensorizable(S, i) $S' \leftarrow Tensorize(S, i); i' \leftarrow i$
S2	Add Multi-Level SPM	HasDataReuse(S, i) & HasMultiLevelCache(S, i) $S' \leftarrow AddMultiLevelCacheRead(S, i); i' \leftarrow i$
S3	Add Multi-Scope SPM	HasDataReuse(S) & HasMultiScopeCache(S, i) $S' \leftarrow AddCacheReadInputs(S, i)$; $S'' \leftarrow AddCacheWrite(S', i); i' \leftarrow i-1$

Regarding the constraints generation (Step-2), Heron scans through the generated schedule primitives and applies predefined constraint generation rules if conditions are satisfied. Concretely, an empty search space is created at first. Then, Heron scans the schedule primitives and iterates all the rules to generate new constraints. In each iteration, Heron analyzes the current schedule primitives to check whether the rule's condition is satisfied. If satisfied, this rule is applied to generate the related variables as well as their constraints. Finally, these generated variables and constraints are fed into the $CSP_{initial}$ to update the search space.

Schedule generation rules. Heron adopts generation rules for DLAs as well as for general-purpose architectures (i.e., rules in Table 2). Generally, the rules of DLAs are designed by considering how to efficiently leverage the hardware intrinsics and dedicated scratchpad memory (SPM). From the perspective of programmers, compared to general-purpose architectures, DLA mainly features coarse-grained hardware intrinsics (e.g., `wmma` in TensorCore) and user-programmable dedicated SPM (e.g., Unified Buffer in TPU). Thus, we design rules by abstracting such common architectural characteristics of various DLAs. Table 6 summarizes three kinds of generation rules, where Rule-S1 corresponds to hardware intrinsics, and Rule-S2 and Rule-S3 correspond to dedicated SPM.

Concretely, Rule-S1 uses the tensor intrinsics to accelerate computation. Take the `mma_sync` in TensorCore as an example, to verify the applicability of tensorization for the current node, Heron mainly checks whether the computation matches the matrix multiplication's expression pattern (e.g., $C[...] \mathrel{+}= A[..] * B[..]$). If the node can be tensorized, Heron decides how to map the loop iterators by analyzing the indices. For example, the matrix multiplication operator (e.g., $C[i,j] \mathrel{+}= A[i,r] * B[r,j]$)) can be tensorized by directly mapping $[i,j,r]$ to $[m,n,k]$. The 2D convolution operator with NCHW layout (e.g., $C[n,c,h,w] \mathrel{+}= A[n,rc,h+rh,w+rw] * B[c,rc,rh,rw]$) can be tensorized by first transforming the convolution to a matrix multiplication via *im2col* and then mapping the iterators directly.

Rule-S2 generates multiple nodes for data movement between different levels of on-chip SPMs. To apply this rule, Heron first checks whether the node has data reuse opportunity (e.g., matrix

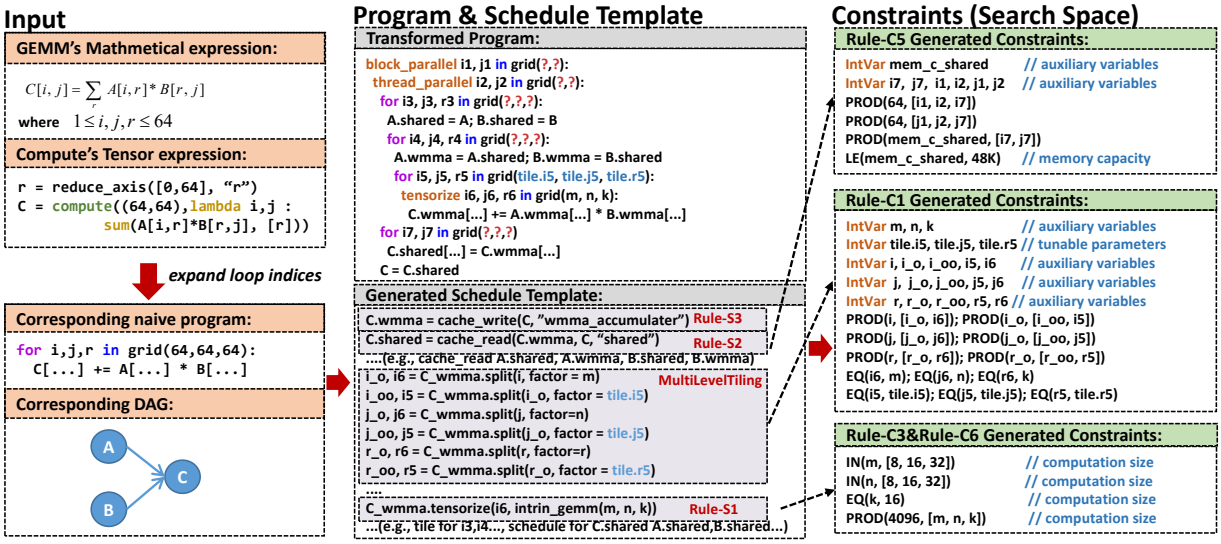

Figure 4: An illustrative example of the constrained space generation for an GEMM operator. Variables that are not tunable are referred as auxiliary variables.

multiplication) and then checks whether the DLA has multiple levels of SPMs (e.g., TensorCore has two levels of SPMs for wmma fragments and shared memory). When the conditions are satisfied, Heron inserts `cache_read` primitives for both the node and its producers to perform data movement.

Rule-S3 generates multiple nodes for moving data in or out of different SPMs that are used for different types of data. To apply this rule, Heron first checks whether the node has data reuse opportunity and then checks whether the DLA has multiple scopes of SPMs (e.g., Synapse Buffer and Neuron Buffer in DianNao [23]). When the conditions are satisfied, Heron inserts `cache_write` primitive for the node and `cache_read` primitives for its producers.

Table 7: Constraint types and their descriptions

No.	Type	Description
T1	PROD(v, $[v_1, ..., v_n]$)	$v = v_1 * ... * v_n$
T2	SUM(v, $[v_1, ..., v_n]$)	$v = v_1 + ... + v_n$
T3	EQ(v1, v2)	$v_1 = v_2$
T4	LE(v1, v2)	$v_1 \le v_2$
T5	IN(v, $[c_1, ..., c_n]$)	$v = c_1$ or ... or $v = c_n$
T6	SELECT(v, u, $[v_1, ..., v_n]$)	$v = v_u$

Constraint generation rules. Based on the common characteristics of generated schedule primitives, we group the related constraints into 6 types and propose predefined rules to automatically generate the constraints.

Table 7 summarizes 6 types of constraints, which can well describe the composable relationship of different variables. Concretely, *PROD* or *SUM* produce constraints that the variable equals to the product or sum of an array of variables. *EQ* and *LE* produce constraints that are used for comparing the values of variables. *IN* produces a constraint that the variable can only take a value from a list of constants. *SELECT* produces a constraint that the variable takes value in a list of variables and the index of the selected value is determined by another tunable parameter (e.g., $stage_2.i == X_{location}$ where X is a list of loop length related variables and *location* is the computation location when $stage_2$ is fused into $stage_1$).

Table 8 illustrates that the constraint generation rules are abstracted from the common characteristics of generated schedule primitives. Concretely, when there exists a loop that has been split or fused (i.e. *HasLoopSplit* or *HasLoopFused*), Rule-C1 and Rule-C2 generate a constraint that the product of two loops' lengths equals to the length of another loop. When there exists a variable whose value is restricted to the candidates (i.e., *HasCandidates*), Rule-C3 generates the *IN* constraint to constrain the variable. When there exist two stages where $stage_2$ is fused into $stage_1$ (i.e., *HasStageFused*), Rule-C4 leverages *SELECT* to generate constraints that the loop lengths of $stage_2$ depend on the compute location. When there exists a schedule primitive that explicitly uses SPMs (i.e., *HasSPM*), Rule-C5 generates 3 constraints. The first one (i.e., *PROD*) implies that the memory consumption of each input tensor is determined by the product of loops' lengths. The second one (i.e., *SUM*) implies that the total memory consumption is determined by the sum of each tensor. The third one (i.e., *LE*) implies that the total memory consumption is less than the SPM capacity. When DLA has special architectural constraints (i.e., *HasSpecialArchConstraints*), Rule-C6 generates dedicated constraints for different DLAs. For example, TensorCore limits the vector lengths to $\{1, 2, 4, 8\}$, while VTA has constraints on the order of loops.

An example. Figure 4 illustrates the main generation process of the search space for a GEMM operator, where the process is roughly divided into three parts: the input, generated schedule template, and generated constraints. For the input, the compute is first represented as a tensor expression. Then it is converted to a naive program and a DAG by directly expanding loop indices. For the schedule template, it is generated by applying rules on the stage nodes of the DAG in reverse topological order (i.e., C, A, B). Specifically, the template and the transformed program can be generated by:

Table 8: Constraint generation rules, rule conditions and descriptions, where P is the schedule primitive to be constrained.

No.	Rule	Condition	Description
C1	AddLoopSplit	HasLoopSplit(P)	$PROD(l, [l_o, l_i])$
C2	AddLoopFuse	HasLoopsFused(P)	$PROD(l, [l_1, l_2])$
C3	AddCandidates	HasCandidates(P)	$IN(v, [c_1, ..., c_n])$
C4	AddStageFuse	HasStagesFused(P)	$PROD(v_k, [l_{k+1}, .., l_n])$ for each k; $SELECT(v, loc, [v_1, ..., v_k])$
C5	AddMemLimit	HasSPM(P)	$PROD(mem_i, [l_1, ..., l_n])$ for each i; $SUM(total, [mem_1, .., mem_t])$; LE$(total, limit)$
C6	AddDLASpecific	HasSpecialArchConstraints(P)	Specialized for each DLA

$$\text{input} \longrightarrow \text{C} \overset{Rule-S3}{\longrightarrow} \text{C.wmma} \overset{Rule-S2}{\longrightarrow} \text{C.wmma} \overset{Rule-S1}{\longrightarrow} \text{C.wmma} \overset{MultiLevelTiling}{\longrightarrow} \text{output} \quad (2)$$

According to the given order, node C is firstly transformed by Rule-S3 which generates a new node C.wmma. Then C.wmma is transformed with Rule-S2, Rule-S1, and the *MultiLevelTiling* Rule in Table 2 to generate the schedule template. The schedule template covers different program candidates. Take loop ordering as an example, the order of spatial loops, i.e., ($i3$, $j3$, $i4$, $j4$, ...), can be reordered to ($j3$, $i4$, $j4$, $i5$, ...) by setting $i3 == 1 \& j6 == 1$. For the constraints, Heron traverses all the schedule primitives to apply constraint generation rules. For example, schedules generated by the *MultiLevelTiling* satisfy the condition of *AddLoopSplit* (i.e., Rule-C1), which results in a declaration of tunable parameter-related variables and auxiliary variables with *PROD* or *EQ* types of constraints. Finally, the generated variables and their constraints form $CSP_{initial}$ which defines the generated constrained search space.

Customization. Heron is able to support new DLAs by slightly modifying both the schedule and constraint generation rules in order to recognize the architectural constraints. Take TensorCore for example, compiler designers can modify Rule-S1 to properly leverage the hardware intrinsic (i.e., use *mma_sync* for computation and *mma_load_matrix_sync* for data loading). Rule-S2 and Rule-S3 also can be modified for proper *cache_read* or *cache_write* according to the memory hierarchy of the target DLA (i.e., multiple levels of SPMs including shared memory and wmma fragments, multiple scopes of SPMs including *wmma_matrix_a* and *wmma_matrix_b*). For constraint generation rules, Rule-C5 can be modified to set the proper capacity constraint for allocated buffers. To describe the computation size and memory access constraints, designers can declare dedicated variables (e.g., *vector_length*) with corresponding constraints (e.g., EQ(*loop, vector_length*) & IN(*vector_length*, {1,2,4,8})) in Rule-C6. Heron can easily adapt to a new DLA with a small modification of the proposed generation rules.

5 CONSTRAINED SPACE EXPLORATION

Constrained space exploration searches for the optimal assignments of tunable parameters that can maximize the performance while satisfying the constraints specified in $CSP_{initial}$. Algorithm 2 shows the detailed workflow of the exploration process. The inputs of the explorer include $CSP_{initial}$, total trials for hardware measurements, and generations of genetic algorithm. The exploration process consists of multiple iterations. In each iteration, four steps are included: (1) forming the first generation with randomly sampled assignments as well as the high-performance ones in the previous iteration, (2) evolving the population for several generations to optimize the fitness scores, (3) selecting promising assignments for hardware measurement using $\epsilon - greedy$ strategy [26], and (4) updating the cost model based on measured performances. Finally, Algorithm 2 outputs the program with the highest measured performance as the optimized program.

Algorithm 2 **CGA-based Exploration**

```
1: Input: CSP_initial, Trials, Generations
2: D ← ϕ; Pop ← ϕ
3: while i ≤ Trials do
4:     Step-1. Forming CGA's first generation of population.
5:     Pop_random ← RandSAT(CSP_initial, solver)
6:     Pop ← Pop + Pop_random
7:     Step-2. Evolving the population for several generations.
8:     for j ∈ {0, 1, ..., Generations − 1} do
9:         Pop ← Select_rw(Pop, model)        ▷ The Roulette-wheel Selection
10:        CSPs ← constraint-based crossover and mutation
11:        Pop ← RandSAT(CSPs, solver) ∪ Pop
12:    Step-3. Selecting assignments for hardware measurements.
13:    programs ← Select_ε−greedy(Pop, model)
14:    perfs ← Measure(programs)
15:    Step-4. Updating cost model.
16:    D ← D + {perfs}; i ← i + |perfs|
17:    Update(model, D).
18: return program with highest performance
```

5.1 Constraint-Based Genetic Algorithm (CGA)

Heron leverages CGA to search for high-performance assignments that satisfies $CSP_{initial}$. Different from the traditional GA-based exploration process introduced in Section 2.3, CGA encodes the tunable parameters as well as other auxiliary variables declared in $CSP_{initial}$ to form the chromosome. Instead of performing crossover and mutation on concrete solutions to generate new chromosomes, CGA performs such transformation (i.e., constraint-based crossover and mutation) on $CSP_{initial}$ to generate a set of new *CSP*s. The new *CSP*s are then for generating concrete value assignments to form the new generation of the population with the help of a constraint problem solver.

CSP solver. For a given *CSP*, Heron searches for valid assignments of the variables by solving the *CSP* and thus generates chromosomes with concrete solutions for fitness evaluation. This process can be implemented in either grid search or random search. Specifically in Heron, since random search is more efficient than grid search in achieving comparable performance [19], we adopt a *random constraint satisfaction* approach (i.e., *RandSAT*) which returns multiple chromosomes with valid and concrete value assignments that are randomly generated by the solver.

Constraint-based crossover and mutation. CGA generates a set of new *CSP*s by specially designed constraint-based crossover and mutation operator as shown in Algorithm 3. The algorithm requires four inputs including: (1) *Pop* is the population after selection, (2) *model* is the cost model, (3) N is the size of the offspring population, and (4) $CSP_{initial}$ is the generated constrained satisfaction problem. For each new *CSP*, the generation process consists

Algorithm 3 Constraint-based Crossover and Mutation

Input: Pop, $model$, N, $CSP_{initial}$
$CSPs \leftarrow []$
for $i \in \{0, 1, ..., N-1\}$ **do**
  $constraints \leftarrow \phi$
  Step-1. Key variable extraction
  $V \leftarrow$ extract key variables from $model$ by feature importance
  Step-2. Constraint-based crossover
  $c_1, c_2 \leftarrow$ two random chromosomes from Pop
  for $v \in V$ **do**
    $constraints \leftarrow constraints + \{IN(c_v, [c_{1,v}, c_{2,v}])\}$
  Step-3. Constraint-based mutation
  $constraints \leftarrow$ remove one constraint from $constraints$ randomly.
  $CSP \leftarrow CSP_{initial} + constraints$; $CSPs$.append(CSP)
return $CSPs$

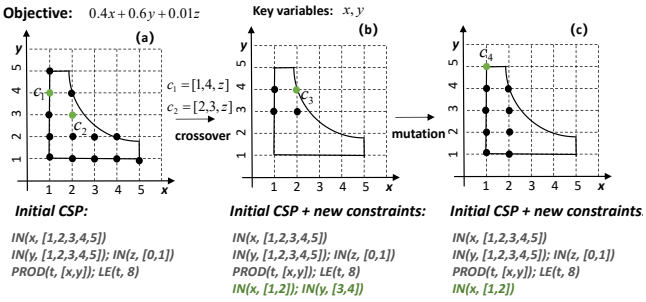

Figure 5: An example of constraint-based crossover and mutation. The points represent valid assignments for $CSP_{initial}$ or the new CSPs.

of three steps including key variable extraction, constraint-based crossover, and constraint-based mutation.

CGA extracts key variables from the cost model according to predicted importance scores of the input features. In the employed XGBoost model, the importance scores are measured by the improvement of achieved performance (e.g., the gini index). The features are then ranked according to their scores and the Top-k of them are selected as the key variables. The key variables (e.g., memory size) are closely related to the predicted performance. Thus two chromosomes with the same values on key variables share similar predicted performance, which helps crossover to retain good genes.

CGA performs the constraint-based crossover and mutation by exerting new constraints on $CSP_{initial}$. For crossover, CGA first chooses two random chromosomes (i.e., c_1 and c_2) from the population and then creates new constraints for the key variables. For a specific key variable v and a given chromosome c, the corresponding value assignments of chromosomes c, c_1, and c_2 on v are denoted as c_v, $c_{1,v}$, and $c_{2,v}$ respectively. CGA then creates a constraint $IN(c_v, [c_{1,v}, c_{2,v}])$ that constrains c_v to be equal to either $c_{1,v}$ or $c_{2,v}$. For mutation, CGA randomly removes one constraint from the $constraints$ generated by the crossover operator. The resultant $constraints$ are then added to $CSP_{initial}$ to form the new CSP.

The proposed constraint-based crossover and mutation not only guarantee the validity of offspring chromosomes but also retain good genes during evolution. Figure 5 shows an example of the process. The objective function of the constrained optimization problem is $0.4x + 0.6y + 0.01z$ and the constraints (i.e., $CSP_{initial}$) are $xy \leq 8 \& x \in \{1, 2, 3, 4, 5\} \& y \in \{1, 2, 3, 4, 5\} \& z \in \{0, 1\}$. In Step-1, variables x and y are considered as key variables since they are more related to the value of the objective function. In Step-2, two chromosomes (i.e., $c_1 = [1, 4, z]$ and $c_2 = [2, 3, z]$) are randomly selected for crossover which results in two new constraints (i.e., $x \in \{1, 2\}$ and $y \in \{3, 4\}$). In Step-3, the mutation operator then removes the $y \in \{3, 4\}$ constraint and makes it possible for finding the optimal solution (i.e., c_4). As shown in Figure 5(b) and (c), the new CSPs consist of both $CSP_{initial}$ and new constraints. Thus assignments (e.g., c_3 and c_4) that satisfy the new CSPs also satisfy $CSP_{initial}$, which means that offspring chromosomes by solving the new CSPs are guaranteed to be valid. Moreover, the good genes from parents can be retained by crossover when generating offspring chromosomes (e.g., x from c_2 and y from c_1 are retained to generate c_3 with higher fitness score). In summary, the proposed approach preserves validity and retains good genes during evolution to make the exploration process efficient.

6 EXPERIMENTAL METHODOLOGY

6.1 Evaluation Platforms

We extensively evaluate Heron on 3 representative platforms:

- **NVIDIA TensorCore** is integrated into NVIDIA V100 GPUs (Volta architecture) [12], which has 640 TensorCores to achieve the peak performance of 112 TFLOPS. We also evaluate Heron on TensorCores of other GPUs including NVIDIA T4 (Turing architecture) [10] and NVIDIA A100 (Ampere Architecture) [8], to demonstrate its generality.
- **Intel DL Boost** is integrated into Intel's Xeon Gold 6240 CPU with 18 cores, achieving 23 TOPS peak performance.
- **TVM Versatile Tensor Accelerator (VTA)** is configured with 256 processing elements to perform 8-bit computation on the Xilinx PYNQ-Z2 platform, achieving 51TOPS peak performance.

6.2 Evaluated Benchmarks

We evaluate Heron with both deep learning (DL) operators and networks. Regarding the operators, we select 9 typical and widely-used operators, including GEMM, 1D convolution (C1D), 2D convolution (C2D), 3D convolution (C3D), transposed 2D convolution (T2D), dilated convolution (DIL), batch matrix multiplication (BMM), GEMV, and scan (SCAN). Specifically, for each operator, we evaluate 6-10 different combinations of shapes and show the geometry average speedups normalized to Heron. The detailed shape configurations are the same as Ansor [71] and AMOS [72] for a fair comparison. Regarding the networks, we select 4 commonly-used neural networks, including ResNet-50 [38], Inception-V3 [65], VGG-16 [64], and BERT [34] with a batch size of 16.

6.3 Comparison Baselines

Our baselines include 4 state-of-the-art automatic generation approaches (AutoTVM [26], Ansor [71], AKG [70], and AMOS [72]) and 4 vendor-provided hand-tuned libraries (cuDNN, cuBLAS, PyTorch, and oneDNN). More specifically, AutoTVM supports all three selected platforms via hand-written templates and shows high performance on the baseline operators. AMOS systematically explores different software mappings of loop iterations onto DLAs and is the

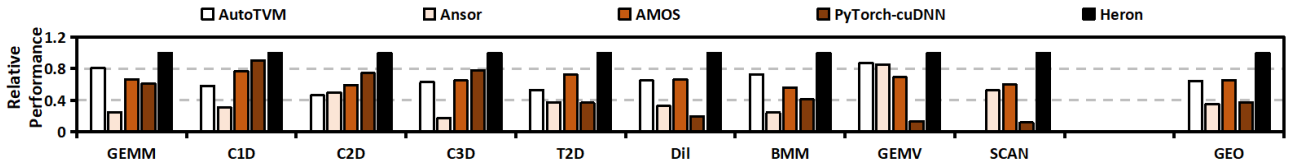

Figure 6: Performances relative to Heron on NVIDIA V100 TensorCore.

state-of-the-art approach on operators with multiple feasible mappings (e.g., C1D and C2D). Ansor is the state-of-the-art approach for code generation of GPU CUDA Core and CPUs, but it cannot utilize DLAs due to its architectural constraints. However, it is still worth comparing Heron with Ansor to show the benefits of using DLAs like TensorCore. AKG is the state-of-the-art polyhedral-based approach for code generation of TensorCore, and we use it as a baseline to show the effectiveness of exploration-based approaches. For a fair comparison, we run AutoTVM, Ansor, AMOS, and Heron up to 2,000 measurement trials per test case and report their best performance. Regarding the hand-tuned libraries, on TensorCore, we use the cuDNN-v8.2.1, cuBLAS-v11.2.1 and kernels in PyTorch-v1.10.2 as baselines, and on DL Boost, we select oneDNN-v2.5.0 as the baseline. Heron uses TVM for code generation and or-tools [4] for solving CSPs.

7 EXPERIMENTAL RESULTS

We compare the operator and network performance of Heron against the baselines on three DLAs, including 3 GPU TensorCores, DL Boost, and VTA. To detail the effectiveness of Heron, we further present generated search spaces' quality and exploration efficiency. Finally, we show that Heron does not introduce additional compilation costs.

7.1 Operator Performance

We first evaluate the operator performance on the three DLAs.

TensorCore. Figure 6 shows that Heron achieves 1.55×, 2.85×, 1.52×, and 2.69× performance improvement over AutoTVM, Ansor, AMOS, and hand-tuned PyTorch (with cuDNN/cuBLAS as backend), respectively[1]. Compared to AutoTVM and AMOS, Heron is able to perform more powerful multi-level loop tiling [71] by automatically generating accurate constraints for the tiling factors. Besides of tiling, AMOS cannot use the *storage_align* scheduling primitive to reduce shared memory bank conflicts. The tunable parameters of this primitive are closely related to the lengths of loops and the compute location of shared memory load, which cannot be described by a small number of simple constraints. Compared to Ansor, Heron combines the advantage of the high-performance TensorCore computing units and the efficient auto-scheduling strategies [71, 73] to achieve better program optimization.

We further compare Heron against baselines on more GPUs including NVIDIA T4 and NVIDIA A100 with the most widely-used GEMM and C2D (i.e., detailed configurations are listed in Table 9). All these configurations are from well-known networks [34, 38, 65] or baselines [71]. We select 5 different shape configurations for each operator and various batch sizes (e.g., 1, 8, 16, and 32) that cover different scenarios aiming at both low latency and high throughput. We also report the absolute performances on these two platforms to show the hardware utilization of different approaches on these workloads. Figure 7 shows that Heron consistently outperforms other generation approaches including AutoTVM, Ansor, AKG, and AMOS. Moreover, compared to hand-tuned libraries (i.e., cuDNN and cuBLAS) and the polyhedral-based approach (i.e., AKG), Heron as well as other exploration-based approaches are more scalable across different platforms since they can adjust their scheduling strategies according to the hardware and software environments.

DL Boost. Figure 8 shows that Heron achieves 2.93×, 12.0×, 2.71× and 1.49× performance improvement over AutoTVM, Ansor, AMOS, and hand-tuned oneDNN, respectively. For convolution operators, we use the same weight layout as oneDNN (e.g., *Ohwl64o4i* or *Ohwl32o4i* for C2D and *Olhw4i16o4i* for T2D). Such layouts are cache-friendly and improve the performance by nearly 30% compared to the normal layouts. AutoTVM has fixed tiling structures while Heron explores more complicated tiling structures that are controlled by the *tile_factors*. AMOS cannot tune different compute locations for fused stages (i.e., cached results), since different compute locations change the inner loops' lengths whose values also need to satisfy the alignment constraints of *tensorize*. Heron uses automatically generated constraints to describe such dependencies and thus explores this kind of program transformation easily.

VTA. We evaluate Heron on GEMM, C2D, and BMM with AutoTVM as the baseline. Figure 9 shows that Heron achieves an average 2.32× performance improvement over AutoTVM on VTA. Regarding the C2D, Heron achieves comparable performance with AutoTVM on all configurations. The main reason is that it is easy to optimize on such a simple architecture with flexible GEMM computation units, where both AutoTVM and Heron can easily reach more than 90% of the peak performance. Regarding the GEMM and BMM, Heron outperforms AutoTVM by exploring more complicated multi-level tiling structures. VTA has special constraints on the tiling structures due to the constraints on the cycles of writing the same address. Multi-level tiling becomes error-prone if such constraints are not considered. Heron handles this challenge by directly applying the constraints on the variables that represent the lengths of loops.

7.2 Network Performance

Figure 10 shows the performance of Heron on 4 evaluated networks. On average, Heron achieves 1.69×, 1.46×, and 1.44× improvement over AutoTVM, AMOS, and PyTorch-cuDNN, respectively. Since ResNet-50 and Inception-V3 use many 1×1 kernels for convolution layers that are well optimized by AutoTVM and AMOS, PyTorch-cuDNN cannot perform better on these two networks. However, both AutoTVM and AMOS perform much worse than

[1]The polyhedral-based approach, i.e., AKG, only works for GEMM and C2D, and it is compared in later experiments.

Table 9: Evaluated configurations for C2D and GEMM on T4 and A100 respectively.

C2D Names	Batch	H	W	CI	CO	R	S	Padding	Stride	GEMM Names	M	N	K
C1	1	56	56	64	64	1	1	0	1	G1	1024	1024	1024
C2	8	28	28	512	128	1	1	1	1	G2	4096	4096	4096
C3	16	14	14	1024	512	1	1	0	2	G3	32	1000	2048
C4	32	7	7	512	512	3	3	0	1	G4	32	4096	4096
C5	32	14	14	256	256	3	3	1	1	G5	32	1000	4096

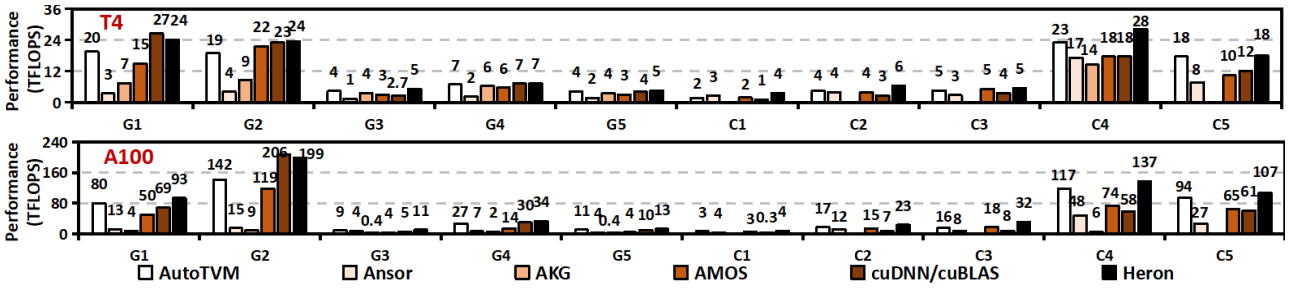

Figure 7: Performance comparison on other GPU platforms including NVIDIA T4 and NVIDIA A100.

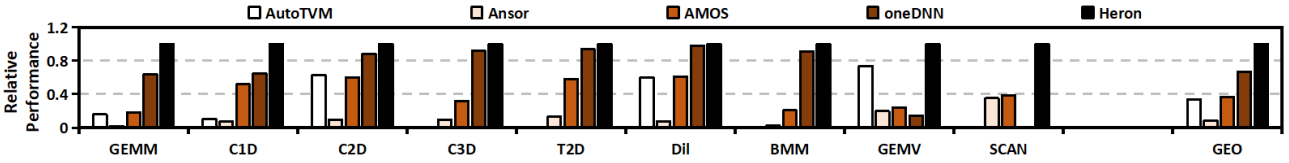

Figure 8: Performances relative to Heron on Intel DL Boost.

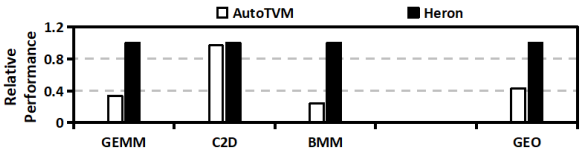

Figure 9: Performances relative to Heron on TVM VTA.

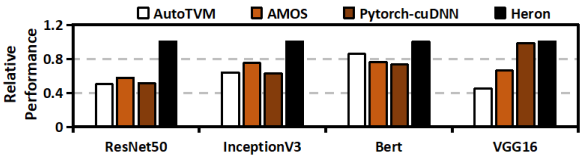

Figure 10: Performances of different networks relative to Heron on TensorCore.

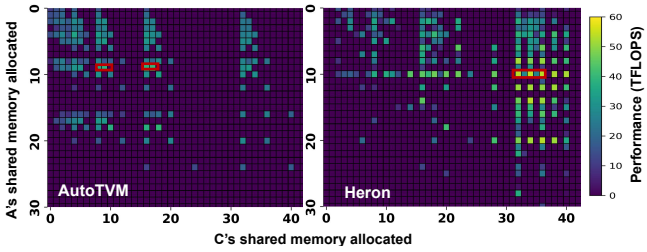

Figure 11: Visualization of the search spaces of AutoTVM and Heron on GEMM. Each point represents a sub-space of the search space and the color of the point represents the highest performance sampled in the sub-space. Thus points with the same coordinate can have different colors since other features (e.g., unroll lengths) can be different. Sub-spaces containing optimal or near-optimal programs are marked.

PyTorch-cuDNN on VGG-16 which only uses 3×3 kernels for convolutional layers. Heron overcomes such deficiency by generating accurate search spaces that contain most of the high-performance programs and a highly efficient searching algorithm for finding the optimal or sub-optimal programs.

7.3 Quality of Search Spaces

Figure 11 compares the search space automatically generated by Heron against the manually constrained search space of AutoTVM. More specifically, the search spaces of the GEMM operator *G1* are visualized with an obtained performance by considering two key parameters, i.e., the allocated shared memory of output matrix C (X-axis) and input matrix A (Y-axis). The search space of Heron differs from that of AutoTVM in two aspects: 1) either the average or maximal performance of all valid programs in the search space of Heron is much better than that of AutoTVM, and 2) the search space of Heron is much more complicated because the performance of two neighboring programs differs significantly as shown in the marked sub-spaces in Figure 11, which advances higher challenges on the exploration efficiency of search algorithms.

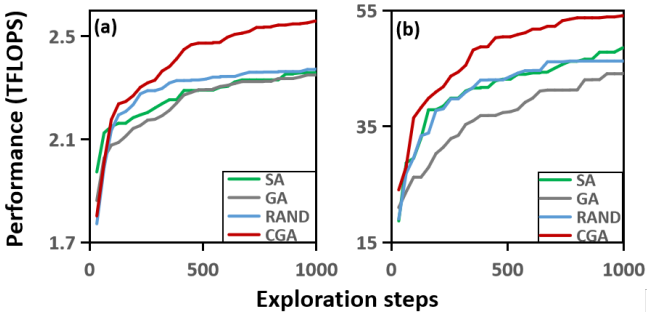

Figure 12: Comparison of CGA, SA, and GA on (a) the C2D and (b) the GEMM operator. The setup of baseline is the same as Figure 2.

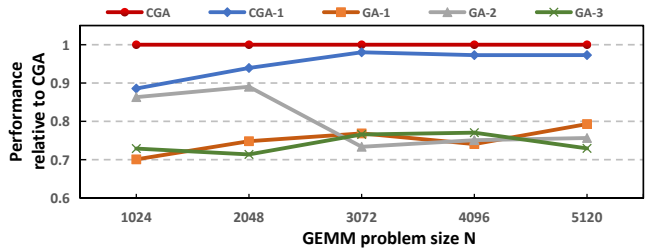

Figure 13: Comparison of CGA (i.e., CGA and CGA-1) and other constraint-handling techniques for GA (i.e., GA-1, GA-2, and GA-3). The y-axis is the performance relative to CGA (the higher the better). The x-axis is the size N of the GEMM operator with shape (N, N, N).

7.4 Exploration Efficiency of CGA

To demonstrate the efficiency of the proposed CGA, we first compare CGA with 3 popular exploration algorithms (i.e., SA, GA, and RAND), and then compare CGA against a modified version of CGA (CGA-1) and 3 state-of-the-art constraint-handling techniques of genetic algorithms (GA-1 [62], GA-2 [51], and GA-3 [61]).

Figure 12 shows that CGA searches faster and finds better programs compared to popular exploration algorithms. Take the GEMM (i.e., Figure 12(b)) for example, CGA finds better programs in 500 exploration steps compared to the baselines with 1000 exploration steps. The efficiency of CGA mainly stems from the ability to explore more high-performance while valid programs within the given time budget.

Figure 13 compares CGA against other constraint-handling techniques of GA including: (1) CGA-1 refers to CGA with key variables chosen randomly, (2) GA-1 refers to the GA with stochastic ranking [62], (3) GA-2 refers to the GA with a SAT-Decoder [51], and (4) GA-3 refers to the GA based on multi-objective optimization [61]. Experimental results show that CGA outperforms CGA-1 since the key variables predicted by the cost model are more related to the fitness score, which helps retain the good genes of the parent chromosomes. The gap between CGA and CGA-1 becomes smaller as the problem size grows since the accuracy of the cost model trained with a fixed number of samples degrades, which makes both approaches hard to find high-performance programs. GA-1 and GA-3 cannot guarantee the validity of the offspring chromosomes and thus perform worse than CGA. GA-2 guarantees the validity of offspring chromosomes using a SAT-decoder and thus outperforms GA-1 and GA-3 on smaller problem sizes. However, GA-2 has to generate invalid intermediate chromosomes which are later decoded into valid ones. Thus, GA-2 cannot retain the good genes of parent chromosomes, which results in performance degradation as the problem size grows.

In general, the efficiency of CGA stems from the ability to explore more high-performance while valid programs since the constraint-based crossover and mutation retain the good genes and guarantee the validity of offspring chromosomes.

Table 10: Compilation time of different approaches on TensorCore.

Operator	AutoTVM (min)	AMOS (min)	Heron (min)
GEMM	91	96	92
BMM	90	97	90
Conv1D	98	87	56
Conv2D	97	91	90
Conv3D	105	147	97

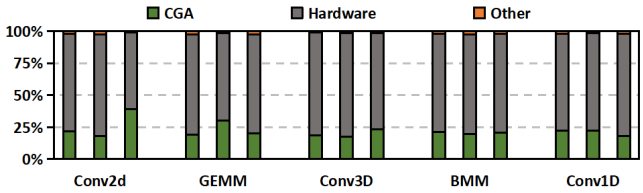

Figure 14: Breakdown of Heron's compilation time.

7.5 Compilation Time

The compilation time is critical for practical usage, and we compare the compilation time of Heron against two exploration-based approaches, i.e., AutoTVM and AMOS. Table 10 shows the average compilation time on 5 commonly used operators, and all these approaches perform 2,000 trials of hardware measurement. The results show that Heron's compilation time is 87% of AutoTVM's and 82% of AMOS's on average because Heron can quickly find optimal programs with lower measurement costs. Figure 14 further shows the breakdown of Heron's compilation time, where we show 3 test cases for each operator. Thanks to the improvement in the efficiency of modern constraint satisfaction problem solvers, the cost of CGA is 23% on average. Most of the compilation time is spent on hardware measurement which ranges from 61% to 79%, with 76% on average. Other costs include training cost model are less than 1% of the total time. Thus Heron does not introduce additional compilation costs.

8 RELATED WORK

In addition to reviewing deep learning accelerators, we summarize the related work of manually optimized libraries, automatic code generation for deep learning, and genetic algorithms for constrained optimization.

Deep learning accelerators. DLAs are customized for different DL applications, aiming at elevating both performance and energy efficiency to the extreme. In academia, by leveraging specialized functional units, memory hierarchy, and interconnect, DianNao family (i.e., DianNao [23], DaDianNao [28], PuDaianNao [48], and ShiDianNao [35]) greatly improve the performance of DL computation. Eyeriss [29] introduces a brand-new dataflow, which minimizes data movement energy consumption on a spatial architecture. Following this trend, many of emerging accelerators that target different algorithms [21, 48, 57] or are with new technologies [30, 36] are proposed. Meanwhile, spurred by the rapid expansion of DL applications in the industry, hardware vendors (e.g., NIVIDA TensorCore [20] and Intel NNP [40]), internet giants (e.g., Google TPU [44] and Apple Bonic [6]), and even startups (e.g., Cambricon MLU [3] and Graphcore IPU [42]) have released a number of DLAs. In fact, dedicated DLAs, whether in academia or industry, are essentially domain-specific rather than general-purpose architecture, thus inevitably incurring complicated and diverse architectural constraints.

Manually optimized libraries. Manually-optimized libraries are flexible and efficient techniques to improve DL performance. Traditional high-performance libraries in HPC domain such as MKL [1], OpenBLAS [14], and cuBLAS [2] are used for accelerating DL computation on general-purpose architecture. Dedicated software libraries have also been developed by DLA vendors. To better exploit TensorCore, cuDNN v7.0 [16] utilizes TensorCore intrinsics to accelerate various DL operators such as convolution. Intel released a new library oneDNN [13] for leveraging its DL Boost acceleration mechanism. Other DLA vendors also released libraries along with their hardware products. For example, GraphCore released PopLIBS [15] and Habana released SynapseAI [5] to achieve both programmability and high performance. However, these high-performance libraries are heavily optimized and tuned in a hardware-specific fashion, which requires intense engineering efforts and a deep understanding of algorithms, compilers, and hardware architecture in order to obtain outstanding performance.

Automatic code generation. Halide [59] presents a scheduling language for high-performance computing. This language is well suitable for both automatic search and manual optimization, and thus is followed by a number of researches [17, 47, 56]. TVM automatically generates low-level optimized code for various hardware, but the users are required to write schedule templates manually. Benefiting by exploring larger search spaces with generated templates, FlexTensor [73] and Ansor [71] can identify better-optimized kernels but cannot support typical DLAs such as TensorCore, TPU, and VTA. Although UNIT [67] can support tensor instructions of DLAs, it still relies on the manual-implemented schedule. Spatial accelerator compilers [22, 33, 39] leverage cost models designed by human experts to accelerate space exploration. dMazeRunner [33] enumerates and evaluates all feasible tile sizes (e.g., 1.75e+07 for a C2D operator), which is not feasible for larger spaces. MindMappings [39] uses a gradient-based approach to explore unconstrained space, which is hard for spaces with complicated discrete constraints. Recent work [18, 66, 70] leverage polyhedral models to solve the code generation problems. Tiramisu [18] provides fine-grained control of optimizations. Tensor Comprehensions [66] generates high-performance GPU kernel codes for arbitrary mathematical operations on tensors. AKG [70] is the state-of-the-art that leverages polyhedral schedulers to perform much wider transformations and it can be used for TensorCore. Although automatic code generation techniques evolve very fast, the key deficiency is that they do not carefully consider the pervasive and complicated architectural constraints of DLAs during the program generation process, which might result in low optimization efficiency.

Genetic algorithms for constrained optimization. Genetic algorithm is an unconstrained optimization approach, therefore, special constraint-handling techniques are introduced to deal with the complicated constraints. Typical constraint-handling techniques mainly consist of four categories including penalty functions, repair algorithms, decoders, and multi-objective optimization [32, 46, 53]. Penalty functions [41, 43, 60, 62, 63, 69] are used to punish constraint violations by lowering the score on invalid assignments. Repair algorithms [54, 58, 68] repair invalid assignments by local search and replace invalid assignments with modified assignments. Decoders [45, 51] is designed to map the genotypes (i.e., chromosomes in the original space) to the phenotypes (i.e., chromosomes in a manually designed space) which are guaranteed to be valid. Multi-objective optimization methods [31, 52, 61] model the constraints as additional objectives and thus the chromosomes are selected based on the Pareto dominance relationship. All such approaches either fail to guarantee the validity of offspring chromosomes (e.g., penalty functions and multi-objective optimization methods) or may generate invalid intermediate chromosomes (e.g., repair algorithm and decoders) since they perform crossover and mutation on concrete solutions. On contrary, CGA guarantees validity through the entire optimization process by evolving on CSPs.

9 CONCLUSION

We propose a novel exploration-based compilation approach, Heron, for efficiently generating high-performance software libraries of various DLAs. Heron automatically enforces massively sophisticated while accurate constraints through the entire program generation including constrained space generation and constrained space exploration. The generated search space is efficiently explored by a novel constraint-based genetic algorithm (CGA) for generating high-performance programs. Experimental results on 3 representative DLAs demonstrate that Heron averagely outperforms state-of-the-art automatic generation approaches and vendor-provided hand-tuned libraries by 2.71× and 2.00×, respectively.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their valuable suggestions. This work is partially supported by the NSF of China (under Grants U22A2028, 61925208, 62102398, 62222214, 62002338, U19B2019, U20A20227), CAS Project for Young Scientists in Basic Research (YSBR-029), Youth Innovation Promotion Association CAS and Beijing Academy of Artificial Intelligence (BAAI).

REFERENCES

[1] [n.d]. Accelerate Fast Math with Intel® oneAPI Math Kernel Library. http://software.intel.com/en-us/intel-mkl.

[2] [n.d]. Basic Linear Algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas.
[3] [n.d]. Cambricon MLU. https://www.cambricon.com/.
[4] [n.d]. Googles Operations Research Tools. https://github.com/google/or-tools.
[5] [n.d]. GOYA INFERENCE PRODUCTS. https://habana.ai/inference/.
[6] [n.d]. Inside Apple's new A11 Bionic processor. https://www.zdnet.com/article/inside-apples-new-a11-bionic-processor/.
[7] [n.d]. Intel Deep Learning Boost - Intel AI. https://www.intel.com/content/www/us/en/artificial-intelligence/deep-learning-boost.html.
[8] [n.d]. NVIDIA A100 TENSOR CORE GPU. https://www.nvidia.com/en-us/data-center/a100/.
[9] [n.d]. NVIDIA cuDNN. https://developer.nvidia.com/cudnn.
[10] [n.d]. NVIDIA T4 Tensor Core GPU for AI inference. https://www.nvidia.com/en-us/data-center/tesla-t4/.
[11] [n.d]. NVIDIA Tensor Core. https://www.nvidia.cn/data-center/tensor-cores/
[12] [n.d]. NVIDIA V100 TENSOR CORE GPU. https://www.nvidia.com/en-us/data-center/v100/.
[13] [n.d]. oneAPI Deep Neural Network Library (oneDNN). https://github.com/intel/mkl-dnn.
[14] [n.d]. OpenBLAS: An optimized BLAS library. https://www.openblas.net.
[15] [n.d]. Poplar Graph Framework Software. https://www.graphcore.ai/products/poplar.
[16] [n.d]. Programming Tensor Cores in CUDA 9. https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/
[17] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.
[18] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205.
[19] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research (JMLR)* 13 (2012), 281–305.
[20] John Burgess. 2020. Rtx on—the nvidia turing gpu. *IEEE Micro* 40, 2 (2020), 36–44.
[21] Ruizhe Cai, Ao Ren, Ning Liu, Caiwen Ding, Luhao Wang, Xuehai Qian, Massoud Pedram, and Yanzhi Wang. 2018. VIBNN: Hardware Acceleration of Bayesian Neural Networks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 476–488.
[22] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. 2020. Marvel: A Data-centric Compiler for DNN Operators on Spatial Accelerators. *arXiv: Distributed, Parallel, and Cluster Computing* (2020).
[23] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 269–284.
[24] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016).
[25] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 578–594.
[26] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*. 3393–3404.
[27] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao Family: Energy-Efficient Hardware Accelerators for Machine Learning. *Commun. ACM* 59, 11 (2016), 105–112.
[28] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 609–622.
[29] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 367–379.
[30] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 27–39.
[31] Carlos A. Coello Coello. 2000. CONSTRAINT-HANDLING USING AN EVOLUTIONARY MULTIOBJECTIVE OPTIMIZATION TECHNIQUE. *Civil Engineering and Environmental Systems* 17 (2000), 319–346.
[32] Carlos A. Coello Coello. 2022. Constraint-handling techniques used with evolutionary algorithms. *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2022).
[33] Shail Dave, Aviral Shrivastava, Youngbin Kim, Sasikanth Avancha, and Kyoungwoo Lee. 2020. dMazeRunner: Optimizing Convolutions on Dataflow Accelerators. *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2020), 1544–1548.
[34] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. 4171–4186.
[35] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of the 42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 92–104.
[36] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 751–764.
[37] David E Golberg. 1989. Genetic algorithms in search, optimization, and machine learning. *Addion Wesley* 1989, 102 (1989), 36.
[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.
[39] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W. Fletcher. 2021. Mind mappings: enabling efficient algorithm-accelerator mapping space search. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021).
[40] Brian Hickmann, Jieasheng Chen, Michael Rotzin, Andrew Yang, Maciej Urbanski, and Sasikanth Avancha. 2020. Intel Nervana Neural Network Processor-T (NNP-T) Fused Floating Point Many-Term Dot Product. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*. 133–136.
[41] Abdollah Homaifar, Charlene X. Qi, and Steven H. Lai. 1994. Constrained Optimization Via Genetic Algorithms. *Simulation* 62 (1994), 242 – 253.
[42] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413* (2019).
[43] Jeffrey A. Joines and Christopher R. Houck. 1994. On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with GA's. *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence* (1994), 579–584 vol.2.
[44] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 1–12.
[45] Slawomir Koziel and Zbigniew Michalewicz. 1999. Evolutionary Algorithms, Homomorphous Mappings, and Constrained Parameter Optimization. *Evolutionary Computation* 7 (1999), 19–44.
[46] Oliver Kramer. 2010. A Review of Constraint-Handling Techniques for Evolution Strategies. *Appl. Comput. Intell. Soft Comput.* 2010 (2010), 185063:1–185063:11.
[47] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–13.
[48] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao : A Polyvalent Machine Learning Accelerator. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 369–381.
[49] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 393–405.
[50] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (ATC)*. 1025–1040.
[51] Martin Lukasiewycz, Michael Glaß, Christian Haubelt, and Jürgen Teich. 2007. SAT-decoding in evolutionary algorithms for discrete constrained optimization problems. *2007 IEEE Congress on Evolutionary Computation* (2007), 935–942.
[52] Efrén Mezura-Montes and Carlos A. Coello Coello. 2008. Constrained Optimization via Multiobjective Evolutionary Algorithms. In *Multiobjective Problem Solving from Nature*.

[53] Efrén Mezura-Montes and Carlos A. Coello Coello. 2011. Constraint-handling in nature-inspired numerical optimization: Past, present and future. *Swarm Evol. Comput.* 1 (2011), 173–194.

[54] Zbigniew Michalewicz and Girish Nazhiyath. 1995. Genocop III: a co-evolutionary algorithm for numerical optimization problems with nonlinear constraints. *Proceedings of 1995 IEEE International Conference on Evolutionary Computation* 2 (1995), 647–651 vol.2.

[55] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A Hardware–Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro* 39, 5 (2019), 8–16.

[56] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–11.

[57] Surya Narayanan, Karl Taht, Rajeev Balasubramonian, Edouard Giacomin, and Pierre-Emmanuel Gaillardon. 2020. SpinalFlow: An Architecture and Dataflow Tailored for Spiking Neural Networks. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 349–362.

[58] David Orvosh and Lawrence Davis. 1994. Using a genetic algorithm to optimize problems with feasibility constraints. *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence* (1994), 548–553 vol.2.

[59] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 519–530.

[60] Khaled M. Rasheed. 1998. An Adaptive Penalty Approach for Constrained Genetic-Algorithm Optimization.

[61] Tapabrata Ray, Hemant Kumar Singh, Amitay Isaacs, and Warren F. Smith. 2009. Infeasibility Driven Evolutionary Algorithm for Constrained Optimization.

[62] Thomas Philip Runarsson and Xin Yao. 2000. Stochastic ranking for constrained evolutionary optimization. *IEEE Trans. Evol. Comput.* 4 (2000), 284–294.

[63] Hans-Paul Schwefel. 1995. Evolution and optimum seeking. In *Sixth-generation computer technology series*.

[64] K. Simonyan and A. Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*.

[65] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2818–2826.

[66] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[67] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying Tensorized Instruction Compilation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 77–89.

[68] Jing Xiao, Zbigniew Michalewicz, Lixin Zhang, and Krzysztof Trojanowski. 1997. Adaptive evolutionary planner/navigator for mobile robots. *IEEE Trans. Evol. Comput.* 1 (1997), 18–28.

[69] Ozgur Yeniay. 2005. Penalty Function Methods for Constrained Optimization with Genetic Algorithms. *Mathematical & Computational Applications* 10 (2005), 45–56.

[70] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 1233–1248.

[71] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-performance Tensor Programs for Deep Learning. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 863–879.

[72] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: Enabling Automatic Mapping for Tensor Computations On Spatial Accelerators with Hardware Abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*. 874–887.

[73] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 859–873.

Received 2022-10-20; accepted 2023-01-19