Matmap: a Modular, Automatable, Tunable Mapper for Accelerator Programming

Grace Dinh
UC Berkeley

***Abstract*—**Achieving high performance for many computational kernels - especially tensor operations - on modern architectures requires programmers to carefully design and implement *schedules* describing how the kernel is to be mapped onto the target architecture's instruction set. Previous approaches to generating schedules have largely been monolithic systems that are difficult and labor-intensive to modify and adapt to new algorithms and architectures. We introduce Matmap, a schedule metaprogramming system which allows programmers to to programatically construct, modify, and tune schedules that can be used across multiple user-schedulable languages. Using Matmap, we implement several scheduling algorithms and integrate them into existing hand-scheduled implementations in only a few hundred lines of code. We show that in this setting, our automated tuning framework achieves performance comparable to hand-tuned implementations of SGEMM on AVX512.

I. Introduction and Background

Tensor computations, such as convolutions, stencil operations, and linear algebra, are central to many applications, such as machine learning, image processing, and scientific computing. As tensor computations often comprise the bulk of the computational resources required for these applications, optimizing their performance is of significant interest both to researchers and engineers. A key contributor to performance is hardware acceleration, in the form of either vector and tensor instructions on CPUs and GPUs or special-purpose accelerators, such as TPUs [13], NVDLA [23], Eyeriss [6], and GEMMINI [9].

However, exploiting hardware acceleration requires a carefully chosen *schedule* describing how to *map* the tensor computation onto the target hardware hardware; this takes the form of code transformations such as loop operations (tiling, reordering, parallelization, unrolling), memory allocation to specific levels of the memory hierarchy, and substitutions of optimized hardware instructions for equivalent kernels. Scheduling decisions can significantly affect end-to-end performance for many applications [20]; however, the actual schedule required to attain good performance varies significantly, depending on both the computation and the target architecture. As both computations and hardware exist in a wide variety of configurations (further exacerbated by automated exploration of algorithm parameters (e.g. neural architecture search [24]), reconfigurable hardware, and multi-tenancy environments where only part of the resources of a given chip can be devoted to a problem), each pair of which corresponds to a different set of scheduling requirements, the effort required to build good schedules is significant.

Schedules can be constructed using several different strategies, each with their own tradeoffs:

- **Manual development** and hand-tuning can produce very high performance. However, achieving that requires both laborious effort by performance programming experts and significant re-engineering effort to target different hardware platforms.
- **Automated search**, including both brute-force search and machine learning (ML) approaches, [1], [12], [17], [21], [22], can produce high performance, sometimes even beating hand-tuned schedules. However, collecting enough samples to get good performance with brute-force search or to accurately train learned performance models can be infeasibly high in environments where the cost of executing and benchmarking a single run is significant (e.g. when running on a simulated candidate hardware before tapeout). Furthermore, approaches that purely rely on automated search are difficult to adapt to changes to hardware parameters, which often occur in architectures when resources are shared by multiple workloads.
- **Heuristics and optimization-based approaches** make scheduling decisions directly from algorithm and machine parameters, often by representing scheduling problems as relatively tractable numerical optimization problems. Many such approaches only make a subset of required scheduling decisions; for example, CoSA [10] automates the selection of tile sizes, loop ordering, and spatio-temporal mappings, but does not perform any scheduling operations such as machine instruction mapping and loop unrolling. As a result, each such heuristic method must be combined with other techniques (e.g. automated search in [12]) to attain good performance in practice.

In many cases, a combination of multiple approaches is desirable. For instance, a programmer may wish to shrink a large tuning space by using an optimization-based approach to make a subset of scheduling decisions, use an autotuner to make the remaining decisions, then use specific knowledge about the architecture to fine-tune the schedule.

Expressing these scheduling decisions by manually implementing low-level code is a labor-intensive and error-prone process. Therefore, schedules are most often implemented using compilers or code generation tools. Since traditional compilers both generate and apply scheduling internally, changing how scheduling decisions are made requires modifying the compiler itself, a challenging task for performance program-

mers who may not be compiler experts. To address this, *user-schedulable* compilers, such as Halide [19], TVM [5], and Exo [11], expose scheduling decisions to users while leaving code generation to the compiler; a user only needs to specify an unoptimized source code and a list of code transforms to be run. For instance, a user can use the following code to instruct TVM to transform a computation `f` by first tiling its loop into 32×32 blocks and then reordering its axes, without needing to manually rewrite the loops involved:

```
ax1, ax2, ax3, ax4 =
↪  schedule[f].tile(f.op.axis[0], f.op.axis[1],
↪  32, 32)
schedule[f].reorder(ax1, ax3, ax2, ax4)
```

Fig. 1. Tiling and reordering in TVM

User scheduling has facilitated the creation of schedule generators, especially automated search tools such as Halide's autoscheduler [1], [17] and TVM's Ansor [21], which can systematically explore the space of schedules by using the compiler to generate code for a large variety of schedules and using the resulting performance measurements as feedback for learning algorithms.

However, these schedulers have largely been designed to operate as monolithic blocks tightly integrated into their compiler frameworks. While these schedules can be inspected and tweaked by hand, they are not intended to be programmatically modified, or to be used in concert with other scheduling tools; for example, Halide's documentation[1] suggests that programmers who wish to iteratively improve on auto-generated schedules (for instance, to adapt them to different hardware) dump its output to Halide C++ source, paste it into the original program, and then modify it, which is not particularly amenable to integration with other tools.

As a result, implementing and testing new optimizations (e.g. using optimizers to make a subset of scheduling decisions while leaving others to existing tools) requires either manually modifying the individual schedules outputted by these monolithic tools, or modifying the generator tools themselves - both of which are difficult and laborious. Similarly, responding to changes in hardware backends requires new schedules to be constructed, either by modifying the schedule generators or (in the case of largely tuning-generated schedules) expensive retraining of models. Furthermore, these tools are tightly integrated components of compilers, making it more difficult to port schedules from one software stack to another.

In this paper, we introduce Matmap, a framework for programatically constructing, modifying, and tuning schedules. Matmap is a **modular** system, where schedules are *declaratively* defined as arbitrary combinations of subschedule blocks which represent sequences of scheduling operations. These blocks are simple Python objects or JSON files which can be generated and programatically modified either manually or by **automated** search tools and optimizers. Furthermore, using GPTune [15] we provide a simple interface for a variety of **tuners**, including OpenTuner [2] and HPBandSter [8], as well as a unified performance database for performance experiments.

II. Programming Model

In order to schedule a program onto a target architecture, a compiler executes a series of *passes* over the code, each time applying a *transform* to change the code. In prior work, these schedule passes are expressed *imperatively*, as a series of directives to be applied in order, as in Figure 1; as a result, a schedule, once generated, cannot be modified without manually editing either the private internal representation of the tool that generated it (e.g. Ansor's undocumented internal schedule format[2]) or the code it generates (as with Halide's autoscheduler).

Instead, Matmap allows users to define scheduling transforms *declaratively*, by representing schedules as static Python objects with publicly accessible interfaces. Specifically, each scheduling transform (e.g. loop tiling) is a subclass of a `Transform` object, which overrides an abstract method `apply(fn, backend)` describing how it should be applied to a function `fn` within the compiler `backend`. Specific scheduling decisions (e.g. tile sizes for each axis), are represented by instances of individual transform objects.

For instance, to express loop tilings, we define the `TilingTransform` class to contain and be initialized with a dictionary `tile_dict` mapping loop indices to tile sizes. The `apply` function, depending on the value of `backend`, will call the appropriate compiler function(s) (e.g. TVM's `tile` or Exo's `split` and `reorder`) on a function represented in the given compiler. There is no need for a one-to-one mapping between Matmap scheduling's transforms and the backend compiler's scheduling directives; for instance, calling

```
TilingTransform({'i':4, 'j':8}).apply(fn)
```

can issue a single 2D `tile()` instruction to a TVM backend or *four* instructions to an Exo backend (which lacks the higher-level `tile` instruction): two calls to `split()` to turn each loop index into two, followed by two calls to `reorder()` to move the inner-tile loops below the outer-tile loops; a similar transform object created for *three* tiling axes would issue three instructions to TVM (a `tile()`, a `split()`, and a `reorder()`) as TVM's `tile()` function only supports two simultaneous axes. As a result, Matmap transforms are compiler-independent, which enables performance programmers to implement optimizations on a variety of target systems.

Schedules can be constructed either by calling the object constructor directly with the desired parameters, as in the tiling example above, or by calling a factory function makes decisions automatically using optimization or tuning. In order to build more sophisticated schedules, several transform objects may be chained into a `CompoundTransform`, which

[1] https://halide-lang.org/tutorials/tutorial_lesson_21_auto_scheduler_generate.html

[2] https://discuss.tvm.apache.org/t/explanation-of-autoscheduler-transform-steps/9512

simply provides a wrapper around a list of `Transform` objects and whose `apply()` function calls them in sequence. For instance, [10] simultaneously optimizes for loop tiling, loop ordering, and spatial mapping; in Matmap, this is represented as a `CompoundTransform` whose `subtransforms` include a `TilingTransform`, a `ReorderingTransform`, and a `SpatialMappingTransform`; this allows users to easily extract parts of the schedule for reuse elsewhere and to modify individual components of the schedule. For example, in order to retarget an architecture-optimized schedule to run on a similar chip with a larger buffer, we may wish to swap out the tiling for one that uses larger block sizes but leaves the loop ordering and spatial mapping untouched, which can be done by simply assigning a new `TilingTransform` object to `subtransforms[0]`.

Matmap is extensible: users may define both new transforms (for instance, polyhedral loop transformations) by subclassing `Transform` and specifying its implementation in individual scheduling instructions in the target compilers; and new optimization algorithms by constructing the appropriate factory methods. A set of utilities allow these algorithms to query the AST to automatically obtain problem parameters, such as tensor sizes, exists for Exo backends; making this compiler-independent is ongoing work.

We have implemented several optimization-based approaches in Matmap, including automated tiling of nested loops using linear and sigmoidal programming [4], [7], the systolic array mapper CoSA [10], and automatic loop ordering with IOOpt [18] in Matmap. With under 150 lines of additional code each, we were able to take existing research code designed to generate schedules to run specific experiments on a small set of hardware targets and convert them to portable Matmap schedules.

III. Tuning and Runtime

In addition to supporting a variety of optimization-based schedule generators, we provide an automated search capability using GPTune [15], an autotuning framework that provides both its own optimizers based on Bayesian learning and clustered Gaussian processes [16] and an interface to third-party tuners such as HPBandSter [8], OpenTuner [2], and ytopt [3]. GPTune keeps a centralized database containing a list of tunable parameters and performance benchmarks from previous runs, which is used to construct a surrogate model to predict and optimize the performance of future runs using transfer learning.

Within Matmap, each `Transform` object which is denoted `Tunable` provides a list of "knobs"- performance parameters that can be adjusted - and associated constraints, which Matmap exposes to the GPTune; users can then define a specific tuning space over some or all of them and use them to train a surrogate model, analogously to the template-based AutoTVM scheduler.

Note that the method used to embed each knob into a tuning space can vary. For example, consider the case of loop reordering. Loop orders can be described as a set of *categorical* choices, expressing a finite set of options (loop orders) that must be chosen, or (following [14]) as a set of *numeric importance scores*, one for each loop, which indicate the loop order to be chosen by sorting the loops in order of importance scores. Determining the optimal embedding scheme for each scheduling parameter is currently a work in progress; the user must currently specify the embedding manually.

As all Matmap schedules (including manually defined schedules) can be represented in a standard format, we are able automatically collect performance data for not only schedules tested by an autotuner but also those generated by one-shot optimizers and by hand. This data can be used for algorithm selection (GPTune will automatically select the best parameters seen so far, regardless of how they were generated) and to improve the convergence speed and quality of the tuning results.

To validate our strategy, we took an existing *hand-scheduled* AVX512 implementation of matrix multiply written in Exo [11] and replaced its hand-generated tiling with a TilingTransform. We were able to both able to fit the Matmap schedule into an existing codebase without significant refactoring, by replacing the hand-tune's (imperative) tiling instructions for L1 cache with a call to TilingTransform.apply(), and to refactor the hand-tuned codebase to use our generated Matmap schedule by relocating the remaining schedule directives into `apply()` function of a special single-backend `Transform` object.

We benchmarked our implementation on a system with two Intel Gold 6126 CPUs (24 cores each) and 768GB of RAM. As seen in Fig. 2, we achieve near comparable performance to a hand-tuned implementation in real-world benchmarks after roughly 255 samples. As we did not fine-tune the remainder of the hand-tuned code (including code optimized for the original tiling, e.g. tile sizes that were multiples of AVX vector lengths), we were unable to close the gap fully, especially in the 64×64 case where traffic on and off L1 cache was not the dominating factor in performance. Further experiments including fine-tuning of the remaining schedule (restricted to a 'local neighborhood' of the original hand-tuned schedule to limit the size of the search space) and theoretically modeled cost functions are ongoing; we believe this will enable us to exceed the performance of hand-tuned code without significantly increasing the number of samples required.

IV. Ongoing and Future Work

Matmap is currently under active development along several lines:

- *Improved schedule composability*: Currently, Matmap schedules use hardcoded variable names or indices, which are not robust to changes in the schedule structure. For example, consider three nested loops indexed by (i, j, k). Splitting i into i_{out} and i_{in} and swapping the order of j and k are independent operations that can be carried out in any order. However, the current implementation of `ReorderTransform` in Matmap (and, in fact, both

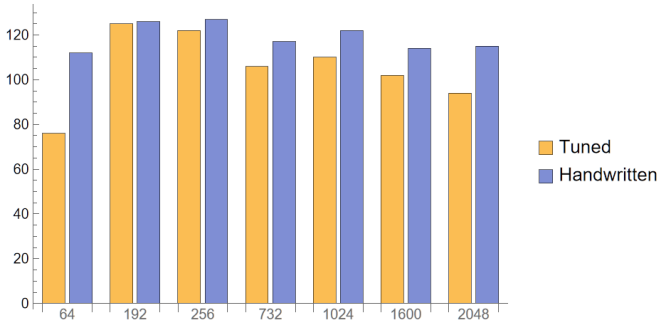

Fig. 2. Square matrix multiply: Matmap tuned (after 255 samples) vs. manually-generated schedule - flops (vertical) vs. matrix size (horizontal)

TVM's and Exo's `reorder()` operations) can be affected by extraneous changes in the loop nest such as this. Other existing scheduling systems encounter similar issues; for instance, the TVM code in Figure 1 relies on the programmer to manually pass the new loop axes generated and returned by `tile()` to `reorder()`, making composability significantly more difficult. We are currently developing a way to describe references to specific nodes of an AST (e.g. specific loop axes) in a manner invariant to schedule transformations, and semantics to allow the user to describe how to resolve conflicts in the case of a potential conflict between two transformations (e.g. an attempt to tile a fully unrolled loop).

- *Improved support for multiple compiler backends:* Our system is most well developed for Exo, with basic support for TVM. We are working on increasing our coverage of these frameworks, and may consider supporting additional backends in the future, possibly using a foreign function interface (FFI) for C++-based languages such as Halide.
- *Porting more automatic schedule generators*: We would like to support additional scheduling algorithms, such as the ability to fuse loops that depend on each other in (e.g. successive layers in neural nets).
- *Automatic tuning space generation*: Tuning frameworks such as Ansor [21] can tune not only specific parameters, but also the scheduling space to search. We believe that this can be encapsulated in Matmap by carefully embedding the choice of schedule space into a GPTune-supported seas space; how to do so in the most computationally efficient manner is an open problem.
- *Hardware-software codesign*: Recently, co-optimizing machine learning algorithms, hardware architectures, and the mappings between them have produced significant speedups [12], [14]. We are using Matmap to accelerate this process. In particular, the ability to easily fuse rapidly generatable schedules such as CoSA [10] with automatic fine-tuning can attain high performance on rapidly changing hardware much faster than optimization-based methods alone, driving down the cost of an iteration; we conjecture that this will help attain higher performance by allowing much more of the codesign state to be searched.

V. Acknowledgments

We would like to thank Gilbert Bernstein, Yuka Ikarashi, and Alex Reinking for providing tremendous assistance with Exo; Iniyaal Kannan for implementing the loop ordering transform and the CoSA integration; Younghyun Cho and Hengrui Luo for helping us with GPTune; and James Demmel for mentorship and feedback. We would also like to thank Kostadin Ilov for invaluable help with setting up software and build environments. Research was partially funded by SLICE Lab industrial sponsors and affiliates Amazon, Apple, Google, Intel, Qualcomm, and Western Digital. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

[1] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley, "Learning to optimize halide with tree search and random programs," *ACM Trans. Graph.*, vol. 38, no. 4, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3306346.3322967

[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, Aug. 2014. [Online]. Available: http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf

[3] P. Balaprakash, R. Egele, P. Hovland, X. Wu, J. Koo, and B. Videau. Ytopt. [Online]. Available: https://github.com/ytopt-team/ytopt

[4] A. Chen, J. Demmel, G. Dinh, M. Haberle, and O. Holtz, "Communication Bounds for Convolutional Neural Networks," arXiv, Tech. Rep., arXiv:2204.08279 [cs] type: article. [Online]. Available: http://arxiv.org/abs/2204.08279

[5] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[6] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.

[7] G. Dinh and J. Demmel, "Communication-optimal tilings for projective nested loops with arbitrary bounds," 2020, arXiv:2003.00119 [cs] type: article. [Online]. Available: http://arxiv.org/abs/2003.00119

[8] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and Efficient Hyperparameter Optimization at Scale," in *35th International Conference on Machine Learning*, ser. ICML '18. PMLR, pp. 1437–1446. [Online]. Available: https://proceedings.mlr.press/v80/falkner18a.html

[9] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 769–774, iSSN: 0738-100X.

[10] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzynek, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2021, pp. 554–566. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISCA52012.2021.00050

[11] Y. Ikarashi, G. L. Bernstein, A. Reinking, and H. G. andJonathan Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '22.

[12] G. Jeong, G. Kestor, P. Chatarasi, A. Parashar, P.-A. Tsai, S. Rajamanickam, R. Gioiosa, and T. Krishna, "Union: A Unified HW-SW Co-Design Ecosystem in MLIR for Evaluating Tensor Operations on Spatial Accelerators," arXiv, Tech. Rep., arXiv:2109.07419 [cs] type: article. [Online]. Available: http://arxiv.org/abs/2109.07419

[13] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. Association for Computing Machinery, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3079856.3080246

[14] Y. Lin, M. Yang, and S. Han, "NAAS: Neural Accelerator Architecture Search," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 1051–1056, iSSN: 0738-100X.

[15] Y. Liu, W. M. Sid-Lakhdar, O. Marques, X. Zhu, C. Meng, J. W. Demmel, and X. S. Li, "GPTune: multitask learning for autotuning exascale applications," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. Association for Computing Machinery, pp. 234–246. [Online]. Available: https://doi.org/10.1145/3437801.3441621

[16] H. Luo, J. W. Demmel, Y. Cho, X. S. Li, and Y. Liu, "Non-smooth Bayesian Optimization in Tuning Problems," arXiv, Tech. Rep., arXiv:2109.07563 [cs, stat] type: article. [Online]. Available: http://arxiv.org/abs/2109.07563

[17] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, Jul. 2016. [Online]. Available: https://doi.org/10.1145/2897824.2925952

[18] A. Olivry, G. Iooss, N. Tollenaere, A. Rountev, P. Sadayappan, and F. Rastello, "IOOpt: Automatic Derivation of I/O Complexity Bounds for Affine Programs," in *PLDI 2021 - 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual, Canada, Jun. 2021. [Online]. Available: https://hal.inria.fr/hal-03200539

[19] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," vol. 48, no. 6, pp. 519–530. [Online]. Available: https://doi.org/10.1145/2499370.2462176

[20] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 369–383. [Online]. Available: https://doi.org/10.1145/3373376.3378514

[21] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating High-Performance Tensor Programs for Deep Learning," arXiv, Tech. Rep., arXiv:2006.06762 [cs, stat] type: article. [Online]. Available: http://arxiv.org/abs/2006.06762

[22] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, *FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 859–873. [Online]. Available: https://doi.org/10.1145/3373376.3378508

[23] G. Zhou, J. Zhou, and H. Lin, "Research on NVIDIA Deep Learning Accelerator," in *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, pp. 192–195, iSSN: 2163-5056.

[24] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," arXiv, Tech. Rep., arXiv:1611.01578 [cs] type: article. [Online]. Available: http://arxiv.org/abs/1611.01578