ONNC: A Compilation Framework Connecting ONNX to Proprietary Deep Learning Accelerators

Wei-Fen Lin, Der-Yu Tsai, Luba Tang, Cheng-Tao Hsieh, Cheng-Yi Chou, Ping-Hao Chang, Luis Hsu
Skymizer Taiwan Inc.

***Abstract*—This paper presents ONNC (Open Neural Network Compiler), a retargetable compilation framework designed to connect ONNX (Open Neural Network Exchange) models to proprietary deep learning accelerators (DLAs). The intermediate representations (IRs) of ONNC have one-to-one mapping to ONNX IRs, thus making porting ONNC to proprietary DLAs much simpler than other compilation frameworks such as TVM and Glow especially for hardware with coarse-grained operators that are not part of the generic IRs in the LLVM backend. ONNC also has a flexible pass manager designed to support compiler optimizations at all levels. A docker image of ONNC bundled with a Vanilla backend is released with this paper to enable fast porting to new hardware targets. To illustrate how an ONNC-based toolkit guides our research and development in DLA design, we present a case study on compiler optimizations for activation memory consumption. The study shows that the Best-Fit algorithm with a proposed heuristic and a reordering scheme may act as a near-optimal strategy, getting the memory consumption close to the ideal lower bound in 11 of 12 models from the ONNX model zoo. To our best knowledge, ONNC is the first open source compilation framework that is specially designed to support the ONNX-based models for both commercial and research projects for deep learning applications.**

***Index Terms*—Deep learning accelerators, Compilers, ONNX, Memory optimization**

I. Introduction

Deep learning applications have been revolutionizing many industries, infusing into increasingly more commercial products that have intelligence capabilities with the potential to impact the everyday experience of people and the standard processes of industry practices. As data scientists propose more domain-specific deep learning models to solve real-world problems, there are increasingly stronger demands for new hardware to accelerate the computation-intensive model training and inference processes. Model training demands high throughput, thus is most often carried out by GPUs, given their massive parallelism, simple control flow, and high energy efficiency. However, model inference aims at low latency to meet real-time requirements. While training is mostly dominated by GPUs, there are several types of deep learning accelerators (DLAs) for inference in the market including GPU, TPU [1], FPGA, and ASIC chips. One of major challenges for DLA design is porting models in high-level language to the executable code on the DLA. To avoid rewriting code and overcome the code optimization challenges, porting a compiler for a proprietary DLA is an effective step to reduce both the software and hardware development cycles. In this paper, we introduce ONNC (Open Neural Network Compiler), a compilation framework aiming at creating a fast track of connecting ONNX to proprietary DLAs.

ONNX [2] is an open format to represent deep learning models. It enables models to be trained in one framework and then transferred to another for inference. With ONNX, AI developers can more easily move models between state-of-the-art tools and choose the combination that is best for them. Industry titans, universities and communities of machine learning researchers worldwide support ONNX. An ecosystem is built upon the drastic need for interoperability and new classes of DLAs that support ONNX have been proposed in the industry and research community. One of the major design challenges is porting a compiler that works for the proprietary accelerator, especially for small-scaled IC design houses who cannot afford dedicated compiler resources. In addition, the research community also shows great interests in proposing new architectures and optimization techniques for DLA implementation. Lacking a good compiler for the research project limits the research scope and slows down the progress in the software development. ONNC presents an opportunity in speeding up the development process of ONNX-based DLAs and facilitates neural network compilation research in general.

ONNC is a collection of open source, modular, reusable compiler algorithms, and tool chains targeted on DLAs. ONNC has been built from ground up for translating ONNX intermediate representations (IRs) to proprietary DLA code. Its software architecture design emphasizes portability and reusability, thus simplifying retargeting. ONNC differentiates itself from other open source compilers such as TVM [3] and Glow [4] in three aspects. First, ONNC has its own backend and IR design besides supporting the LLVM backend. Porting on TVM and Glow requires modification to the LLVM backend, which mainly support fine-grained operators such as multiplier-accumulator (MAC) rather than coarse-grained operators such as convolution (CONV). The mismatch in the design granularity requires more efforts in porting them to ONNX-based DLA. Second, ONNC is an iterative compiler and able to retry from the intermediate pass automatically upon compilation failures. Third, ONNC has a flexible pass manager design that is more suitable for the neural network model compilation. Optimization passes may be implemented as primitive passes with inter-dependency and get scheduled automatically by the pass manager. ONNC is also integrated with the LLVM bit code runtime and backend. Any accelerator that already has the LLVM compiler support can be integrated

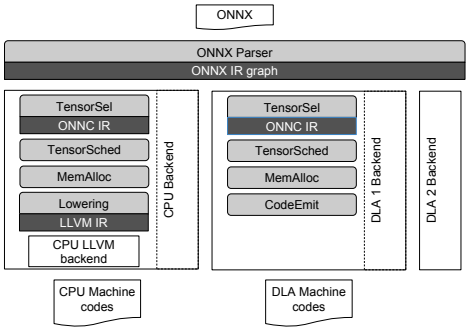


Fig. 1. ONNC software architecture diagram

with ONNC seamlessly. The compatibility with the LLVM backend helps ONNC to be ported to most CPUs, GPUs, and DSPs in a very short time.

The rest of the paper is organized as the follows. Section 2 discusses the design challenges and the ONNC architecture. Section 3 describes the major internal features for porting ONNC to proprietary DLAs. Section 4 covers the utilities including tools, unit tests, and the benchmarks. Section 5 presents a case study on memory optimizations to illustrate how ONNC helps to guide DLA designs. A docker image [5] with a full set of ONNC development package is released with this paper to the research community. Industrial partners are welcome to use ONNC for product development as well.

II. DESIGN CHALLENGES AND COMPILER ARCHITECTURE

User applications written in standard machine learning frameworks, like TensorFlow or PyTorch, require an efficient hardware with massive computation power to deliver quick and accurate results. The key metrics for embedded machine learning applications are accuracy, energy consumption, throughput/latency, and cost. There is a huge design space to explore given a specific priority of design goals. This poses a design challenge for architectural exploration at early design stage, especially for those who favor software/hardware co-design methodology. ONNC provides a modular and reusable software stack built from ground up to support ONNX-based DLAs. Our vision is to empower the hardware designers in the ONNX community to rapidly build a compiler and optimized deep learning libraries for their hardware proposals. Software support is essential for full system evaluation and deployment. To facilitate software and hardware co-design process, ONNC's roadmap is to become a full system ESL tool for the deep learning hardware design.

A. ONNC Software Architecture

Figure 1 depicts the top-level block diagram of ONNC software stacks. The software stack illustrates the functional blocks from importing an ONNX computation graph model to emitting corresponding hardware binaries. In addition to leveraging the LLVM backend, ONNC paves another fast track for proprietary DLAs to execute ONNX models by defining ONNC IR, an intermediate representation (IR) that has one-to-one mapping to the ONNX IR. Two other popular compilation frameworks in deep learning systems, TVM and Glow, built their software stacks on top of the LLVM backend. The intermediate representations of LLVM have a finer granularity than ONNC IRs while mapping to hardware operators. For accelerators built with coarse-grained operators such as convolution, it requires more porting efforts to hack the LLVM backend. Many DLA designs such as Nvidia's NVDLA [6] and Bitmain's Sophon BM168X series [7] favor coarse-grained operators over LLVM operators. In those cases, ONNC provides a more straightforward way to convert ONNX models to target binaries using its own Vanilla backend. thus speeding up porting a compiler to new hardware. For fast porting, users only need to copy the Vanilla backend as a template, override two software pipes at minimum, add optional optimization passes and the framework will handle the rest of work like a charm.

B. Minimal Porting Efforts

ONNC was designed with portability in mind so that it can easily support diverse accelerator hardware. The target-dependent support to certain hardware is implemented as a backend in the ONNC software stack. As Figure 1 shows, there are two kinds of backends, the LLVM backend and the ONNC backend. If a DLA already supports the LLVM compiler, it can be connected to ONNC seamlessly via the LLVM IR. ONNC will continue to integrate more CPU, GPU, and even DSP backends. On the other hand, if a DLA has unique computation features and is not compatible to LLVM, it can implement an ONNC proprietary backend using a Vanilla template to jump-start the porting work.

C. Iterative Compilation Framework with Flexible Pass Manager Design

Pass manager is responsible for managing the execution order of all the optimization and analysis (called passes) during compilation. This concept is from LLVM, but we further enhanced its functionality with automatic iterative compilation so that ONNC's pass manager is more suitable for the neural network model compilation. Traditional pass managers stop immediately when something fails (e.g., constraints not met) at any pass. It relies on human to adjust parameters and retry. We found that compilation failures occur much more frequently in the neural network compilation than in the traditional (e.g., C/C++) compilation. The reasons are briefed as follows. In ONNC design, a complex optimization problem is usually divided and conquered with many sub-steps. However, it is common that those small steps may be coupled with each other, so a step needs to guess or estimate the results of the posterior steps for its optimization to continue. We found that in the neural network compilation, the memory usage is really hard to guess because activation sizes are not as regular as scalar sizes in traditional languages. In addition, the execution time of an operator is not deterministic given various operand size (e.g., small convolution vs large convolution). Therefore,

the iterative compilation is favored over traditional compiler for neural network applications.

D. Heterogeneous Compiler Support

Domain-specific chips increasingly reply on heterogeneity to achieve greater performance and scalability. Most DLAs are part of a heterogeneous system that comprises multiple execution contexts with different programming abstractions and runtimes. In order to reduce programming efforts, a heterogeneous compiler is expected to evaluate the cost functions of each compute unit, partition the workload, and dispatch work to the compute unit with the highest execution efficiency. In the V1.0 release, ONNC has an interface for cost function implementation. However, the rest of support for heterogeneous systems is still under development. Making ONNC a heterogeneous compiler is the top priority in the future ONNC roadmap.

III. ONNC INTERNALS

The ONNC V1.0 release is intended to facilitate fast porting and compiler research for DLA design. Three references are bundled in the release to document the internals of the ONNC compiler including how to create a new IR, how to add a new pass, and how to port to a new target.

A. ONNC IR and Extension

Users may extend ONNC IR for their proprietary design. For example, if a DLA supports a compound operator, CONV-POOL, which computes convolution followed by max pooling in series, a new IR is desired such that the compound operator is regarded as a singleton in scheduling and memory allocation. The ONNC IR has defined a set of common operators among which 116 IRs respectively correspond to 116 ONNX operators. To create new ONNC IRs, users may refer to the ONNC IR Extension Guide [8] that uses concrete examples to describe the detailed steps of adding a new IR including creating a class for the new operator, creating a pass to deploy the new operator, and extending the code emitting pass to translate the new operator into machine code.

B. Pass Manager

ONNC's pass manager supports automatic scheduling based on the dependency defined by the pass designer. For modularity and reusability, a complete optimization pass can be decomposed into several primitive passes and have dependency specified in the pass implementation. The pass manager will check whether the dependency is satisfied or not and take proper actions for automatic scheduling.

As mentioned in earlier section, if a pass fails to achieve an optimization goal, it can opt to return a retry request and then pass manager will re-schedule the retry pass as well as all its dependent passes. There is a shared data structure among passes, so the retry pass can feed the retry reason back to its dependent passes for them to adjust parameters accordingly. More details are documented in the ONNC Pass Manager Getting Started Guide [9].

C. Vanilla Backend

ONNC provides a Vanilla backend as a template to ease the development of a new DLA backend. New IRs or new passes might be required in porting to a new target. ONNC's modular design allows users to reuse passes from other backend development. However, each DLA design has its own application and architecture advantages. To make most out of the DLA hardware, customized optimization passes are the key and the default passes in the Vanilla backend just give users a jump-start. Users may refer to the ONNC Backend Porting Guide [10] for more details.

IV. UTILITIES

In addition to making ONNC an open source, we also released a couple of utilities to help the research community. Users may refer to the ONNC Utilities [11] for more information.

A. ONNC Docker Image

A docker image [5] with the ONNC framework and development environment is available for download in the Docker Hub repository. Users may run and hack ONNC in an isolated container and synchronize the same development environment with others. The idea is to package all necessary dependencies, source code, utilities and documents for ONNC users.

B. Unit Tests and Statistics

Users who modify ONNC are provided with a set of unit tests for regression purpose. Correctness checking might be a hassle without unit tests. More regression tests will be added to improve the test coverage in future work. Once the correctness is checked, users may also desire to gather data for performance analysis. ONNC provides a set of statistics APIs for users to embed performance counters in the compiler, dump statistics in the compiling process, and allows users to parse output for data analysis and visualization tools.

C. Benchmarking Using ONNX Model Zoo

When porting a compiler to a DLA, proper benchmarking using deep learning models is essential. Building deep learning models from scratch is not trivial, at least not for most ONNC users. In the released docker image, the ONNX Model Zoo [12], which is a collection of pre-trained, state-of-the-art models in the ONNX format, is also integrated for easy benchmarking and performance evaluation.

V. CASE STUDY ON MEMORY OPTIMIZATION

Memory is one of the biggest challenges in deep neural networks (DNNs) today. For application-specific DNN chip designs, researchers and engineers have been struggling with the tradeoff between on-chip memory cost and required memory capacity to meet the minimal performance goal. In this study, we illustrate how ONNC helps us to explore this problem by adding a memory optimization pass and generating useful profiling data to guide the design choices.

While making inferences for a specific DNN model, it is always desirable to minimize the memory consumption

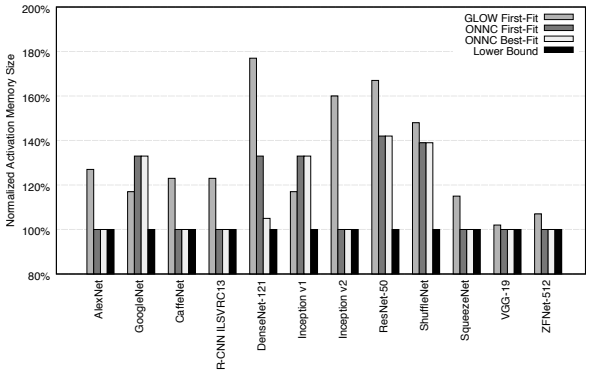


Fig. 2. Memory consumption comparison. The activation memory sizes are normalized to the lower bound.

of intermediate activations of the network. Liveness analysis is a common approach used in compiler optimization for efficient register allocation [13] [14]. Furthermore, analogy between memory management algorithms in operating systems and memory allocation algorithms for computational graphs enables us to start with some well-known algorithms in the literatures [15]. In the context of DNN inference, similar techniques can be applied to static activation memory allocation. By tracing the Glow compiler code, we found that the Glow compiler performs the First-Fit algorithm on activation memory allocation. In Glow's bundle example source code, the size of the activation memory can be extracted from its data structure. Therefore, we use Glow bundles as the baseline in our case study.

To begin with, ONNC implements the First-Fit algorithm on memory allocation as well. First-Fit begins search at the start of the memory and allocates memory from the first hole it encounters large enough to satisfy the request. Another popular algorithm, Best-Fit, is also implemented in ONNC as an alternative. Best-Fit places activation in the smallest free space of allocated memory in which it will fit. In either algorithm, if no free space can be found to satisfy the request, a new memory block following the allocated region will be allocated. In order to understand how close the results are to the "optimal" algorithm, a lower bound is derived for comparison by summing up the sizes of intermediate activations that are in use at memory allocation time and find the maximum over time. Figure 2 depicts the experiment results for all models in the ONNX model zoo.

Compared to Glow First-Fit, ONNC First-Fit performs better in 10 models and worse in 2 models in the ONNX model zoo. The memory consumption is normalized to the lower bound that assumes an optimal policy exists. In 7 out of 12 models, ONNC First-Fit does quite well and gets very close to the lower bound. Glow First-Fit apparently only gets close to the lower bound for the VGG-19 model. After digging deeper into Glow's implementation, we found the following reasons that contribute to the memory consumption differences between ONNC First-Fit and Glow First-Fit: (1) Glow extracts the transpose operation to an independent layer that contains more than one node in the computational graph but ONNC handle the transpose operation in the runtime without

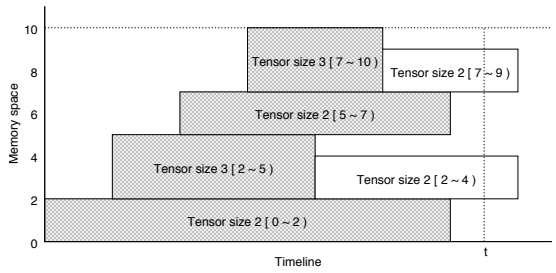
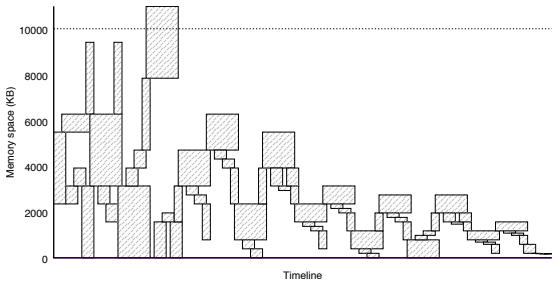


Fig. 3. Fragmentation at the boundaries. At time t, both the top and bottom memory spaces have free space for allocation.

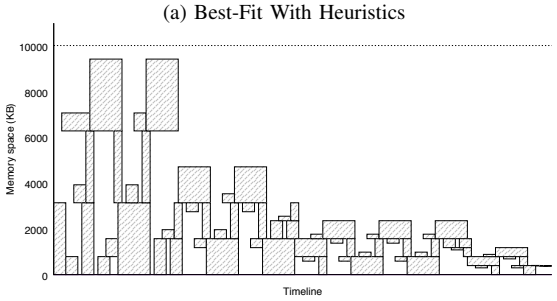
generating additional nodes and perform index transformation instead. (2) Glow expands broadcasting at compile time but ONNC handles broadcasting at runtime. For example, if there is a X[1000] array that need to be broadcasted into 3 channels to be X[3][1000], Glow expands the array into 3 channels at compile time and allocate 3*1000 unit space. On the other hand, ONNC only allocates 1000 unit space at compile time and adjust the indexes to access accordingly at runtime, thus avoiding the extra memory expansion cost. (3) Glow allocates memory to some data that never get used at runtime just for supporting training and inference with the same APIs. Some data are only used for automatic differentiation in training but not used in inference.

ONNC Best-Fit significantly improves the memory consumption for the DenseNet-121 model. It implies that the DenseNet-121 model suffers from the memory fragmentation seriously and the Best-Fit algorithm resolves this issue. This conclusion can be further validated by adding an ONNC profiling tool to calculate the average number of empty slots and the average empty slot size in the allocated activation memory. For DenseNet-121, the average maximal empty slot size for Best-Fit is about 23% less than that of First-Fit. Apparently, DenseNet-121 suffers from serious fragmentation in memory allocation.

Among all the fragmentations, there is a special type of fragmentation that occurs at the boundaries. Figure 3 depicts an example where both bottom and top boundaries of allocated memory have free space at time t. In the First-Fit and Best-Fit algorithms, when a new block of memory is allocated, the new request is always assigned to memory in the higher address space. The fragmentation at the lower boundary never gets attention. To resolve this problem, we tweak both the First-Fit and Best-Fit algorithms with a heuristic to reduce the fragmentation at the lower boundary. The proposed heuristic is simple and straightforward. When no fit can be found for a new request, instead of always allocating memory at the higher memory address space, the lower memory address boundary is also checked to see if a more efficient allocation is possible. For the example in Figure 3, there is more space available at the lower address boundary than the higher address boundary at time t, so the heuristic will favor the lower address space for allocating new memory. A minor adjustment in the data placement needs to be done in the proposed heuristic. In general, data is placed to the left of the free memory space on a fit hit. The proposed heuristic places data to the right



(a) Best-Fit With Heuristics



(b) Best-Fit With Heuristics & Reordering

Fig. 4. Memory allocation diagram for ResNet-50

of the free memory space if the free space is found at the lower address boundary and the start address of the allocated memory does not exceed half of the allocated address space. The idea is to avoid the fragmentation at the lower address boundary.

The experiment results show that the heuristic successfully reduces the memory consumption significantly for the 5 models that suffer from memory fragmentation. The Best-Fit with Heuristic algorithm acts as a near-optimal strategy and reduce the activation memory consumption to ideal lower bound in 10 out of 12 models. Inception-V1 still requires 2.45% more memory than ideal lower bound. For ResNet-50, although it cuts down the memory consumption by 25%, it still requires 16.67% more memory than the ideal lower bound. In order to understand the memory allocation layout on ResNet-50, ONNC is embedded with performance counters to gather runtime information and a visualization tool is developed to display the results in Figure 4. In 4a, a block of 3MB is allocated to the top of the address space while there is still empty space at the lower address region. To fix this issue for ResNet-50, we sort the allocation requests by size before starting memory allocation.

The results in 4b show that reordering allocation requests fixes the issue for ResNet-50. As shown in Figure 5, Best-Fit with Heuristics and Reordering gets all models close to the lower bound except that the DenseNet-121 model still requires 4% more memory than the lower bound.

VI. CONCLUSION AND FUTURE WORK

This paper has described ONNC, an open source compilation framework for systems with deep learning accelerators. We have also shown how we tackled a popular DLA design issue by extending ONNC with proprietary memory allocation algorithm, extracting profiling data, and designing visualization tools. The case study has demonstrated a typical workflow

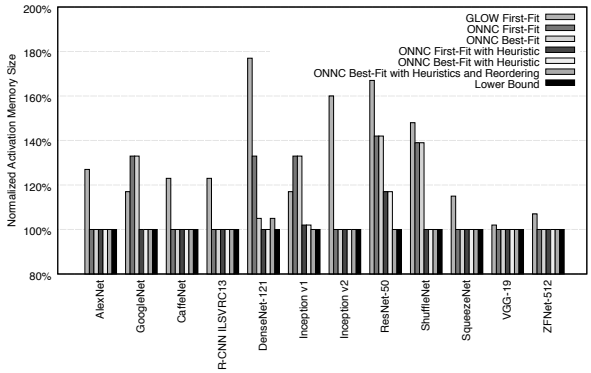


Fig. 5. Memory consumption comparison with heuristics and reordering. The activation memory sizes are normalized to the lower bound.

using ONNC to guide our customers in their customized neural network inference engine design. More generic features in the ONNC framework are planned to be released in the future including heterogeneous compiler support, more optimization passes, more model profiling tools and more target backends. In our vision, ONNC will bridge the gap between research community and commercial products by sharing a common framework. We hope ONNC can facilitate more advanced research and development in deep learning applications.

REFERENCES

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 2017, pp. 1–12.

[2] (2017) Onnx. [Online]. Available: https://github.com/onnx/onnx

[3] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *CoRR*, vol. abs/1802.04799, 2018.

[4] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph lowering compiler techniques for neural networks," *CoRR*, vol. abs/1805.00907, 2018.

[5] (2018) Onnc community edition. [Online]. Available: https://hub.docker.com/r/onnc/onnc-community

[6] (2017) Nvdla deep learning accelerator. [Online]. Available: http://nvdla.org/

[7] *Tensor Computing Processor BM1682*, 2018. [Online]. Available: https://sophon.ai/product/introduce/bm1682.html

[8] *ONNC IR Extension Guide*, Skymizer Taiwan Inc. [Online]. Available: https://github.com/ONNC/onnc/blob/1.0.0-preview/docs/ONNC-IR-Extension-Guide.md

[9] *ONNC Pass Manager Getting Started Guide*, Skymizer Taiwan Inc. [Online]. Available: https://github.com/ONNC/onnc/blob/1.0.0-preview/docs/ONNC-Pass-Manager-Getting-Started-Guide.md

[10] *ONNC Backend Porting Guide*, Skymizer Taiwan Inc. [Online]. Available: https://github.com/ONNC/onnc/blob/1.0.0-preview/docs/ONNC-Backend-Porting-Guide.md

[11] *ONNC Utilities*, Skymizer Taiwan Inc. [Online]. Available: https://github.com/ONNC/onnc/blob/1.0.0-preview/docs/ONNC-Utilities.md

[12] (2018) Onnx model zoo. [Online]. Available: https://github.com/onnx/models

[13] A. W. Appel, *Modern Compiler Implementation in ML: Basic Techniques*. New York, NY, USA: Cambridge University Press, 1997.

[14] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '82. New York, NY, USA: ACM, 1982, pp. 98–105.

[15] C. Bays, "A comparison of next-fit, first-fit, and best-fit," *Commun. ACM*, vol. 20, no. 3, pp. 191–192, Mar. 1977.