CONVOLVE

Seamless design of smart edge processors

GRANT AGREEMENT NUMBER: 101070374

Deliverable D5.5

Initial Memory management and allocation for ULP accelerators

Title of the deliverable	Memory management and allocation for ULP accelerators
WP contributing to the deliverable	WP 5
Task contributing to the deliverable	Task 5.4 and Task 5.6
Dissemination level	PU - Public
Due submission date	30/04/2024
Actual submission date	30/04/2024
Author(s)	Christos Lamprakos [ICCS] Sotirios Xydis [ICCS]
Internal reviewers	Mottaqiallah Taouil [TUD] Bram Verhoef [AXE]

Document Version	Date	Change
V0.1	12.04.2024	Initial document
V0.2	18.04.2024	Submitted for internal review
V0.9	26.04.2024	Reviewers' feedback incorporated
V1.0	29.04.2024	Ready to submit version
V1.1	30.04.2024	Project coordinators' feedback incorporated.

Contents

Deliverable Summary 4
1. Introduction 5
2. Related Work 7
3. Background 8
4. idealloc 8
    4.1 Intuition 8
    4.2 Final Placement Derivation 9
    4.3 Performance and Scalability 10
    4.4 Quality Stochasticity 10
5. Evaluation 11
6. Integration with the CONVOLVE Compiler 13
7. Conclusion 13
8. References 14

Deliverable Summary

Memory management is important for machine learning kernels due to the large amount of data transfers. Memory management in the context of machine learning accelerators is split in two parts: (i) planning and (ii) runtime data transfer management. The planning part is further divided into bufferization, i.e., assigning memory buffers to tensors, and offset assignment, i.e., placing each of the distinct buffers to contiguous chunks of on-chip memory. This document describes our work on the offset assignment front, which is orthogonal to both the bufferization and data transfer parts.

To this end, we present (i) a survey of existing solutions from the state-of-the-art, (ii) a high-level description of our offset assignment agent, (iii) an empirical comparison between our agent and the aforementioned related work, and (iv) a list of the tasks remaining to completion of the whole memory management flow.

In its current state, our solution outperforms most of the state-of-the-art with respect to memory footprint. Work to also make our offset assignment agent fast is in progress, and opens exciting research directions. Integration with CONVOLVE's compiler is straightforward, though not yet done.

1. Introduction

The immense popularity of neural networks coupled to their ever increasing size deems optimal memory handling a vital component of any machine learning (ML) pipeline. This holds true regardless of where a model is deployed: memory is an equally valuable resource in the Cloud and in a constrained edge device.

In the context of ML compilers, memory optimization is a two-stage process. A network described as a sequence of layers, each layer node operating on and emitting tensors, is first *scheduled*--that is, independent computation chunks are identified and destined for simultaneous execution. The CONVOLVE compiler shall utilize STREAM to analyze and guide decisions for that purpose [1].

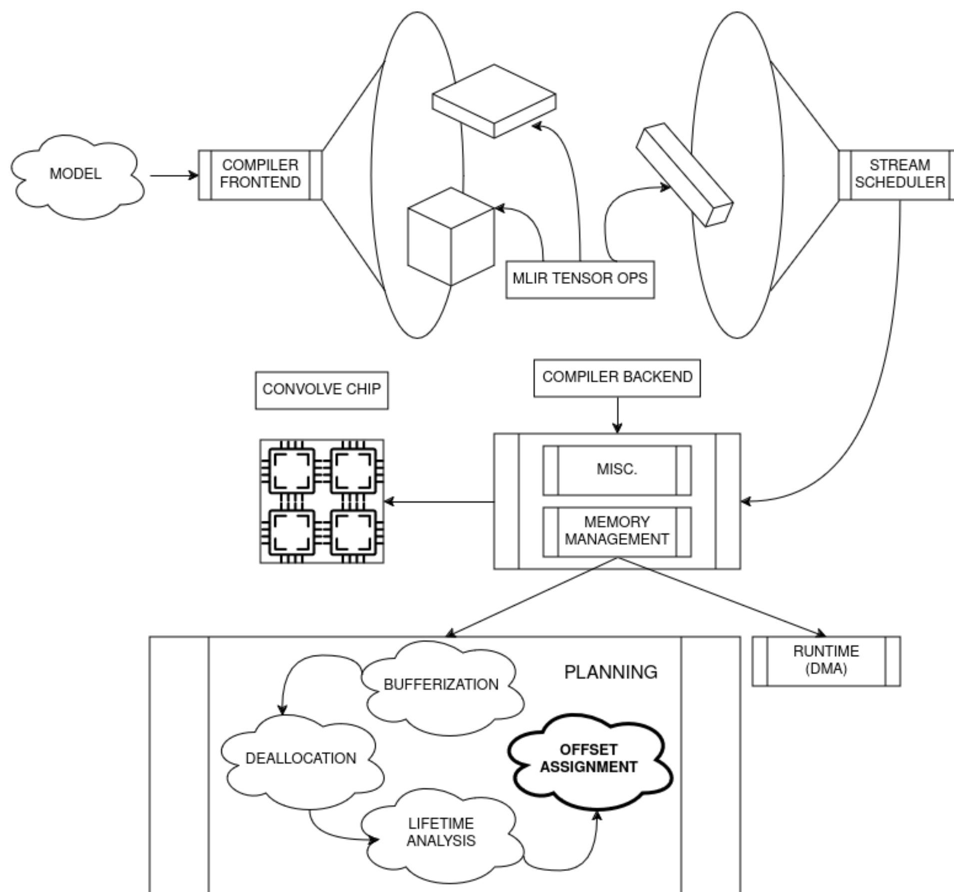

Figure 1: A conceptual overview of CONVOLVE's compilation process, and memory management's place therein.

The derived schedule is then lowered from tensor operations to accelerator-specific ones. A part of said lowering decides and keeps track of which memory buffers hold the data of which tensor, with the overall goal of keeping the total number of tensors to a minimum. This step, called *bufferization*, concludes the first stage of memory optimization.

Bufferization is usually conducted independently of the next stage [2], [3], but synergistic approaches do exist [4], [5]. At the time of writing this deliverable, CONVOLVE plans to keep bufferization and offset assignment separate, but its infrastructure is such that future attempts towards combined memory management remain feasible (if such attempts are viewed as an iterative process repeatedly invoking existing parts).

Buffers are annotated with lifetimes, i.e., moments of logical time between which a buffer is considered "live". Temporally overlapping buffers must remain spatially disjoint, while non-overlapping ones may share the same memory space. As long as the compiler has a concrete definition of logical time, this lifetime analysis is trivial; an indirect consequence of the scheduling and bufferization steps, given the network's static architecture.

The final sequence of lifetime-annotated buffers is the input of memory optimization's last stage. In the literature it comes by many names: *buffer allocation* [6], *static memory allocation* [7], *static memory planning* [8], [9], *memory layout optimization* [4], *offset assignment* [10]. We will be using the terms "buffer allocation" and "offset assignment" interchangeably, though the essence remains the same: each buffer receives an address space offset during that analysis, its goal being to utilize as little memory as possible.

Deliverable D5.2 presents the work conducted by ICCS in the context of offset assignment. Integration with the rest of CONVOLVE's compiler is left as a task for the final memory management deliverable. Scheduling is already handled by STREAM, while bufferization is merely an off-the-shelf MLIR transformation. As will be shown later, even in its current state, our buffer allocator outperforms the vast majority of the state-of-the-art with respect to memory footprint.

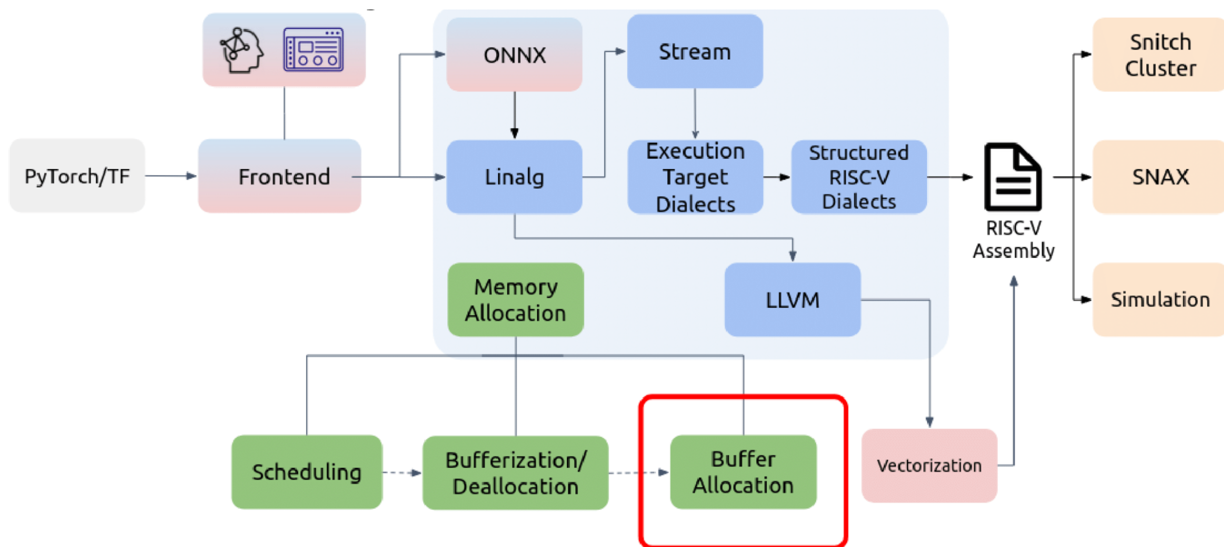

Figure 2: A more detailed map of work done in WP5. The scope of deliverable D5.2 is marked in red.

2. Related Work

To emphasize offset assignment's relevance to real-world ML deployments, we structure this section according to the various frameworks and compilers used in production. We shall conclude with academic works that are not yet in production but bear strong resemblance to our topic.

XLA [11]**:** an open-source compiler developed by Google, used by popular frameworks such as PyTorch [12], TensorFlow [13] and JAX [14]. The offset assignment algorithm followed by XLA focuses on speed rather than memory efficiency, since it relies on simple best-fit placement of the buffers after it has sorted them by lifetime[1].

TFLite [15]**:** a specialized version of TensorFlow tailored to embedded devices and microcontrollers. There are *five* different memory allocation algorithms available in TFLite: greedy-by-size, greedy-by-breadth, greedy-in-order, strip packing, and minimum-cost flow [16]. This variety is reflected on different footprint/latency trade-offs[2].

TVM [17]**:** open-source compiler maintained by the Apache Foundation and targeting a wide array of accelerator backends such as CPUs, GPUs, even FPGAs. TVM also employs more than one buffer allocator: greedy-by-size, greedy-by-conflict, and a hillclimb version alternating between the former two[3].

MindSpore [18]**:** an open-source framework developed by Huawei Technologies, targeting CPUs, GPUs and the Ascend NPU. MindSpore addresses an extended version of offset assignment for training networks in parallel on the same accelerator. The core logic remains identical to most other works: apply some sorting to the buffer sequence, and then place them with an either first- or best-fit heuristic. Due to the requirement for parallel training, these algorithms are enriched by the notion of *safety* [10][4].

Triton [19]**:** a GPU focusing compiler and IR adopted by OpenAI. Triton uses a linear-time storage allocation algorithm formulating the problem as interval graph coloring[5].

Two more works that have ended up being integrated in XLA are TelaMalloc [6] and minimalloc [7]. Both of them use rectangle packing as their problem formulation--in line with our own allocator. Rectangle packing for memory allocation is known to be NP-hard, so

---

[1] github.com/openxla/xla/blob/main/xla/service/memory_space_assignment/best_fit_repacker.cc

[2] github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/delegates/gpu/common/memory_management

[3] github.com/apache/tvm/tree/main/src/tir/usmp/algo

[4] github.com/mindspore-ai/mindspore/tree/5f329d13e3154083d7ee03afd544be64d5d4b765/mindspore/ccsrc/backend/common/somas

[5] github.com/openai/triton/blob/a791746de8f49c99c9e4cc2bf21f6eb639dbf07d/lib/Analysis/Allocation.cpp

both algorithms devise their own heuristics: TelaMalloc an ML-assisted ILP program, and minimalloc a lattice theory isomorphism.

3. Background

The research described in this document utilizes the formulation of static memory allocation as an instance of two-dimensional rectangle (or bin) packing. Let us now elaborate on this abstraction.

As stated earlier, the memory requirements of a neural network's inference stage are viewed as a set of lifetime-annotated buffers. Each buffer is a rectangle in a 2D space where the vertical axis represents the accelerator's address space and the horizontal axis denotes logical time. The allocator may slide rectangles up and down, the operation standing for different offset assignments; it cannot, however, slide them sideways. Buffers are also restricted from overlapping vertically.

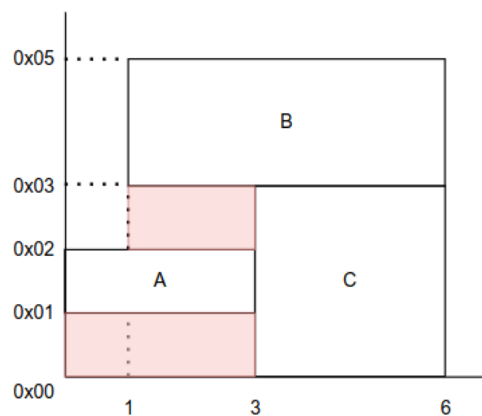

Figure 3: A rectangle packing example.

Rectangle packing algorithms place buffers in such a way in order to minimize the resulting placement's *makespan*, i.e., the largest offset occupied by any buffer. The problem is NP-hard and although it has been extensively studied in the theoretical computer science literature, solution optimality is still at quest. If we define a placement's *load* as the maximum sum of simultaneously live buffers--in other words a lower bound for the ideal placement, the state-of-the-art algorithm, guarantees solutions at most $(2+\epsilon)$ times bigger than the load [20].

4. idealloc

We have created `idealloc`, an implementation of the state-of-the-art rectangle packing algorithm originally described in [20]. Its design has been published in extensive detail as an open-access research paper [21]. This section presents the main key points; we more than encourage curious readers to consult our paper.

4.1 Intuition

In its original paper, the algorithm is demonstrated as a series of theorems and corollaries, each proved by construction. Each next theorem builds on top of the previous ones, starting from elementary cases with, e.g., restrictions on the possible sizes of treated buffers, and

culminating to a general solution. Our implementation is nothing more than a translation of each proof into source code.

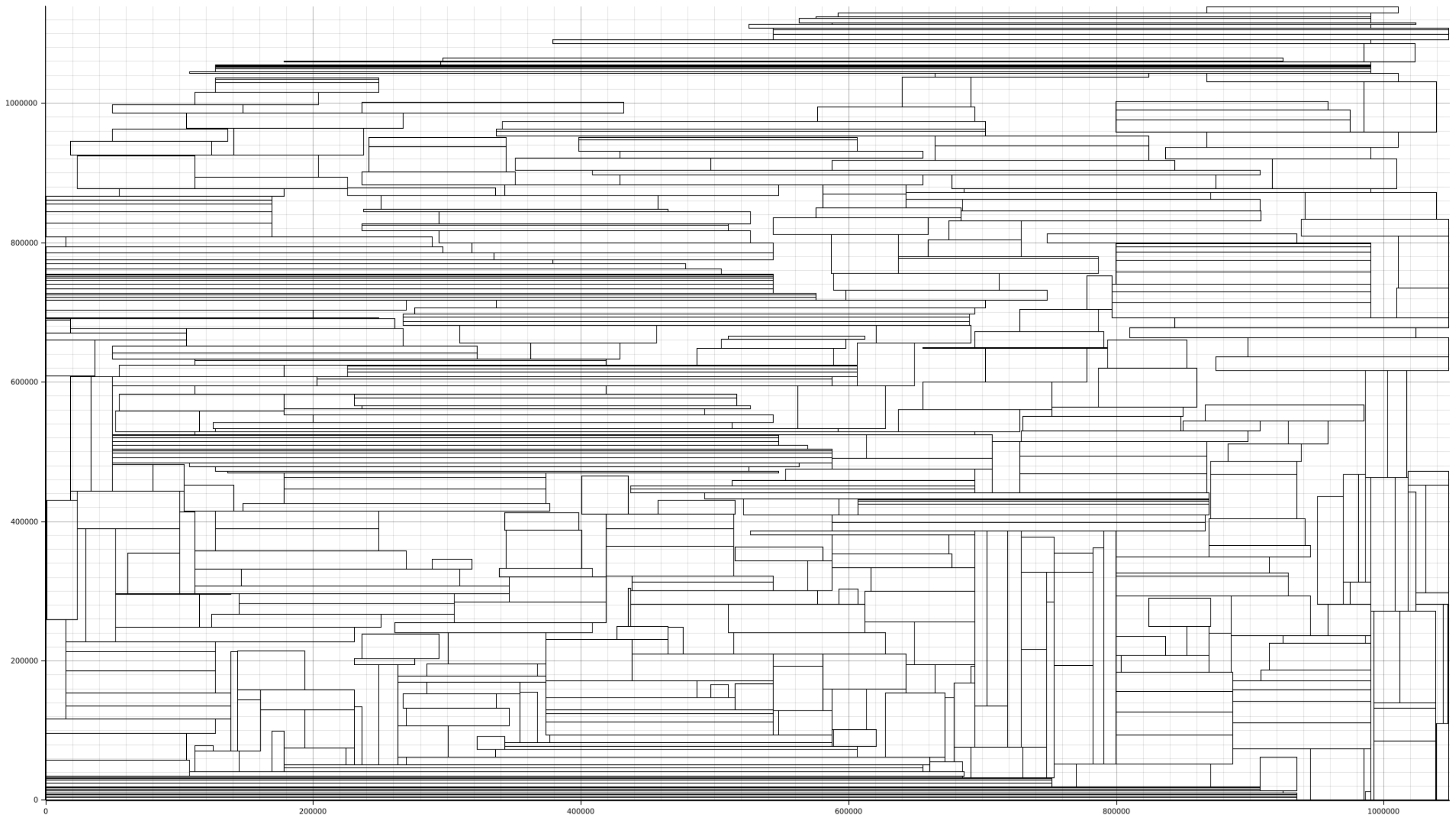

Figure 4: A placement derived by `idealloc` for one of minimalloc's production-grade TPU benchmarks.

The cornerstone operation introduced in the very first lemma of [20] is *boxing*: groups of buffers are boxed together into larger rectangles that are themselves viewed as buffers by the algorithm. For instance, a boxing of buffers A and C in Figure 3 is a rectangle of height 3 and lifetime spanning from 0 to 6. Most of the algorithm can be seen as an iterative boxing process with nested recursions--on a high level.

When a specific criterion regarding the ratio of the formed boxes' minimum and maximum heights is met, the algorithm has converged.

4.2 Final Placement Derivation

Boxing is only the first part of our implementation. It yields a set of boxes which therefore contain boxes and so on until some level where only original buffers reside. In the second part, we iteratively unbox and place until we reach that base level.

One final part deals with the fact that after all the unboxing is done, the initially derived placement is pretty sparse--this is probably owed to the algorithm's theoretical needs toward formal guarantees. We compensate for this sparsity with a "tightening" pass.

4.3 Performance and Scalability

Contrary to our original publication, `idealloc` does not suffer neither from high latency nor from Out-Of-Memory (OOM) killers. We have made several enhancements in order to be able to deal with arbitrarily large buffer sets (in the order of *millions*).

A simple example is that the original version was making *new* objects in memory to represent the contents of each new box: thus one buffer had as many clones on the heap as boxes it belonged to, which, due to the recursive nature of the boxing procedure, was consuming exponential DRAM amounts. We are currently storing pointer vectors instead, thus keeping memory consumption linear to the number of boxes.

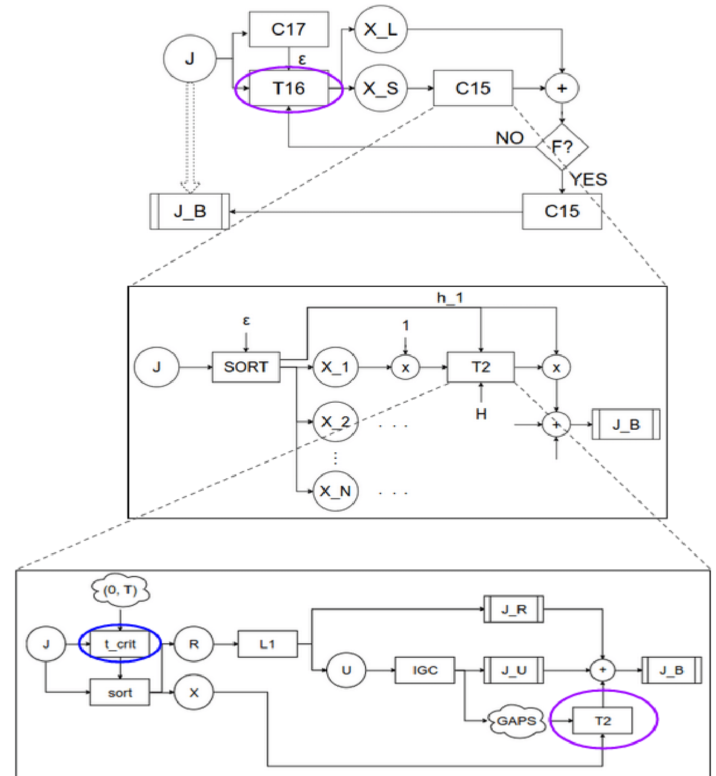

Figure 5: High-level boxing schematic. Recursion marked in purple, and random point selection in blue.

4.4 Quality Stochasticity

In the deliverable summary we mention that we are still trying to make `idealloc` faster; yet the last subsection boasts that it's behaving fine latency-wise. This contradiction is owed to the--as of yet undisclosed--stochastic nature of the boxing algorithm. We try to illustrate what we mean in Figure 5.

Recall that the whole process may be viewed as iterative nested recursions. In the figure we have marked recursion nodes (Theorems 2 and 16) in purple. Theorem 2 also exhibits a *randomness* component: more precisely, given a subset of boxes and a time horizon, the algorithm must select a random point in that interval under the constraint that at least one buffer is live at that moment. This detail makes `idealloc` inherently stochastic, since we

have observed that different point selections yield different placement makespans. Moreover, each "move" made affects the moves available in the next stage.

Thus, `idealloc` doesn't suffer from high latency as long as we run the algorithm only once, i.e., as long as a single sequence of random moves is made. But our early experiments showed that yielding the best possible placements in a single iteration is unlikely (see next section). Our current remedy is thus to run `idealloc` a high enough number of times in order to be certain that a broad area of the search space has been covered. We recently realized that our case is a single-player game: each random moment selection instance may be viewed as a node in a search tree. To this end, we are in the process of implementing Monte Carlo Tree Search [22] towards reducing total iterations.

5. Evaluation

To test our allocator's performance, we performed an offline, i.e., not integrated to some ML framework, comparison of `idealloc`'s performance against the state-of-the-art solutions presented in the Related Work section.

More specifically, we isolated the allocator source code of each compiler and fed it with the open-source TPU benchmarks of [the `minimalloc` repo].

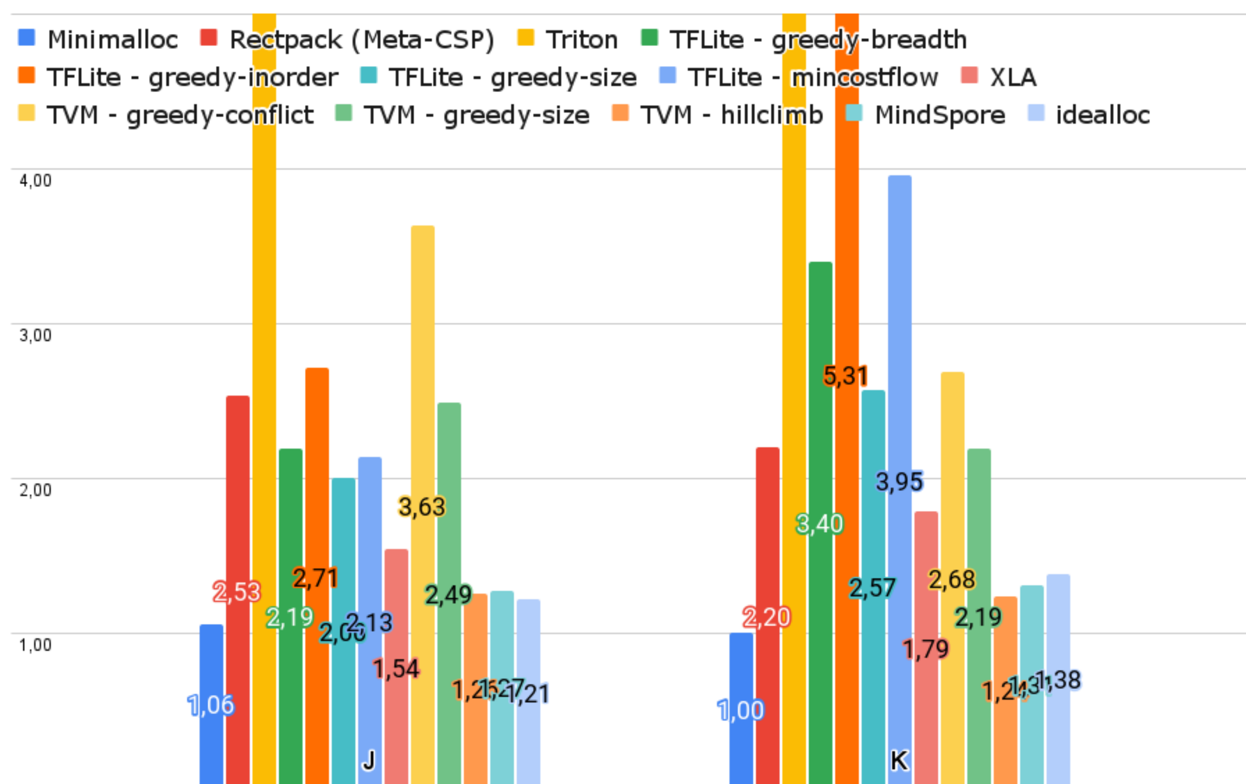

Figure 6: A subset of our memory footprint measurements normalized to each benchmark's load.

Table 1: Single-iteration latencies (ms).		
# of buffers	**minimalloc**	**idealloc**
154	6	8
296	6	9
454	6	13
1821	601*	87
5620	12949*	1703
18692	**SEGFAULT**	1393

Then we gave the same benchmarks as inputs to our implementation, which we executed repeatedly *1 million times*. We were interested in answering the following research questions:

- how do we perform against the state-of-the-art with respect to memory footprint?
- how do we perform against the best allocator memory-wise with respect to single-iteration execution time?

Figure 6 answers the first question. We have kept only the two hardest benchmarks (J, K) in order for it to remain as informative and easy-to-read as possible. The numbers are normalized to the benchmarks' lower bound, i.e., their load. We observe that `idealloc` stays within a 20-40% range from the theoretical optimum (while, quite impressively, `minimalloc` consistently achieves it). The two next-best allocators, MindSpore and TVM hillclimb, perform very close to our implementation. This pattern was consistent across all 11 benchmarks. In **7 out of the total 11** cases, only `minimalloc` beat our implementation. This sparked our second research question, to which we now turn.

Table 2: Empirical comparison benchmark characteristics.		
Bench mark	**# of buffer s**	**Load (x4 KiB pages)**
A	154	256
B	170	256
C	203	254
D	213	241
E	215	256
F	296	256
G	308	256
H	316	256
I	374	256
J	409	242
K	454	256

Table 1 shows that `minimalloc` is characterized by poor scalability as regards allocation time, while our allocator does not have the same problem. Moreover, the two asterisks denote that we had to *manually* configure `minimalloc` in order for it to converge in reasonable time (when we did not do that, allocation time for the 5K case proved more than 80 minutes long, at which point we stopped the execution).

Arguably, given `idealloc`'s stochastic nature, this comparison is unfair to `minimalloc`. Nevertheless, both these results and our plan to tame randomness via Monte Carlo Tree Search as remarked in the previous section, hint towards exciting prospects for our implementation in the future.

6. Integration with the CONVOLVE Compiler

The offset assignment agent presented in this deliverable has been implemented as a shared library in the Rust programming language. The compiler expected as the main output of WP5 utilizes the Python MLIR framework xDSL[6]. In order for `idealloc` to be integrated with CONVOLVE's compiler, the following steps need to take place:

- the tensor lowering, bufferization/deallocation, and lifetime annotation operations noted in the Introduction section must be implemented in xDSL. Particularly regarding lifetimes, (i) a definition of logical time must be formed and (ii) it seems more natural to do the annotations on the tensor level and propagate them downwards from there.
- the result of the previous step must be raised up again to (or maintained all along as) some Python collection of objects representing the derived buffers. This seems to be the approach taken by most--if not all--compilers and frameworks listed in the Related Work section.
- a Python ⟵⟶ Rust Foreign Function Interface (FFI) such as PyO3[7] must be used to allow xDSL and `idealloc` to communicate.

7. Conclusion

We have reported on our recent work on memory management for ML compilers, specifically the problem of buffer allocation. The solution presented stands very competitive against the state-of-the-art, and exposes a very simple interface to the CONVOLVE compiler in both the technical and the semantics sense.

8. References

[1] A. Symons, L. Mei, S. Colleman, P. Houshmand, S. Karl, and M. Verhelst, "Towards Heterogeneous Multi-core Accelerators Exploiting Fine-grained Scheduling of Layer-Fused Deep Neural Networks." arXiv, Dec. 20, 2022. Accessed: Apr. 16, 2024. [Online]. Available: http://arxiv.org/abs/2212.10612

[2] N. Vasilache *et al.*, "Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction." arXiv, Feb. 07, 2022. Accessed: Apr. 16, 2024. [Online]. Available: http://arxiv.org/abs/2202.03293

[3] "Bufferization - MLIR." Accessed: Apr. 16, 2024. [Online]. Available:

[6] github.com/xdslproject/xdsl

[7] github.com/PyO3/pyo3

https://mlir.llvm.org/docs/Bufferization/

[4] H. Shu, A. Wang, Z. Shi, H. Zhao, Y. Li, and L. Lu, "ROAM: memory-efficient large DNN training via optimized operator ordering and memory layout." arXiv, Oct. 30, 2023. Accessed: Mar. 05, 2024. [Online]. Available: http://arxiv.org/abs/2310.19295

[5] Y. Li, A. Gupta, and S. Malik, "Combined Scheduling, Memory Allocation and Tensor Replacement for Minimizing Off-Chip Data Accesses of DNN Accelerators." arXiv, Nov. 29, 2023. Accessed: Mar. 05, 2024. [Online]. Available: http://arxiv.org/abs/2311.18246

[6] M. Maas, U. Beaugnon, A. Chauhan, and B. Ilbeyi, "TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, Vancouver BC Canada: ACM, Dec. 2022, pp. 123–137. doi: 10.1145/3567955.3567961.

[7] M. D. Moffitt, "MiniMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, Vancouver BC Canada: ACM, Mar. 2023, pp. 238–252. doi: 10.1145/3623278.3624752.

[8] M. Levental, "Memory Planning for Deep Neural Networks." arXiv, Feb. 23, 2022. Accessed: Mar. 07, 2024. [Online]. Available: http://arxiv.org/abs/2203.00448

[9] "[RFC] Unified Static Memory Planning - Development / pre-RFC," Apache TVM Discuss. Accessed: Apr. 16, 2024. [Online]. Available: https://discuss.tvm.apache.org/t/rfc-unified-static-memory-planning/10099/3

[10] I. Lamprou and Z. Zhang, "Safe Optimized Static Memory Allocation for Parallel Deep Learning".

[11] "openxla/xla." OpenXLA, Apr. 16, 2024. Accessed: Apr. 16, 2024. [Online]. Available: https://github.com/openxla/xla

[12] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library".

[13] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning".

[14] "google/jax." Google, Apr. 16, 2024. Accessed: Apr. 16, 2024. [Online]. Available: https://github.com/google/jax

[15] "TensorFlow Lite | ML for Mobile and Edge Devices," TensorFlow. Accessed: Apr. 16, 2024. [Online]. Available: https://www.tensorflow.org/lite

[16] Y. Pisarchyk and J. Lee, "Efficient Memory Management for Deep Neural Net Inference." arXiv, Feb. 15, 2020. Accessed: Mar. 05, 2024. [Online]. Available: http://arxiv.org/abs/2001.03288

[17] T. Chen *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning".

[18] Ltd. Huawei Technologies Co., Ed., "Huawei MindSpore AI Development Framework," in *Artificial Intelligence Technology*, Singapore: Springer Nature, 2023, pp. 137–162. doi: 10.1007/978-981-19-2879-6_5.

[19] P. Tillet, H. T. Kung, and D. Cox, "Triton: an intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, Phoenix AZ USA: ACM, Jun. 2019, pp. 10–19. doi: 10.1145/3315508.3329973.

[20] A. L. Buchsbaum, H. Karloff, C. Kenyon, N. Reingold, and M. Thorup, "OPT versus LOAD in dynamic storage allocation," in *Proceedings of the thirty-fifth annual ACM symposium*

on Theory of computing, in STOC ’03. New York, NY, USA: Association for Computing Machinery, Jun. 2003, pp. 556–564. doi: 10.1145/780542.780624.

[21] C. P. Lamprakos, S. Xydis, F. Catthoor, and D. Soudris, “The Unexpected Efficiency of Bin Packing Algorithms for Dynamic Storage Allocation in the Wild: An Intellectual Abstract,” in *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management*, Orlando FL USA: ACM, Jun. 2023, pp. 58–70. doi: 10.1145/3591195.3595279.

[22] C. B. Browne *et al.*, “A Survey of Monte Carlo Tree Search Methods,” *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012, doi: 10.1109/TCIAIG.2012.2186810.