AoS to SoA Transformation of Recursive Tree-like ADTs

Vidush Singhal; Milind Kulkarni
Purdue University

Abstract

Algebraic data types are a language feature available in many functional programming languages like Haskell. These make it easier for programmers to write complex data types. However, the performance of such programs depends on various factors of which we are interested mainly in **data-layout**. Gibbon[1] is a compiler that supports a small language of tree traversals written in a subset of Haskell. It transforms programs that perform traversals on tree-like ADTs to corresponding traversals over the serialized representation of the ADT. This **enhances spatial locality** resulting in enhanced runtime performance. We take this data representation a step further by allowing such ADTs to be **represented as a struct of arrays (SoA)** serialization format. The structure of arrays layout can allow us to do performance crucial optimizations such as **vectorization** over irregular, tree like recursive data types and transform recursive traversals over them to iterative loops. We build our tool on top of the Gibbon compiler and for a subset of programs, the SoA transformation performs better due to enhanced locality. Preliminary results show that for certain reduction-like programs we see ~2x speedup compared to baseline Gibbon.

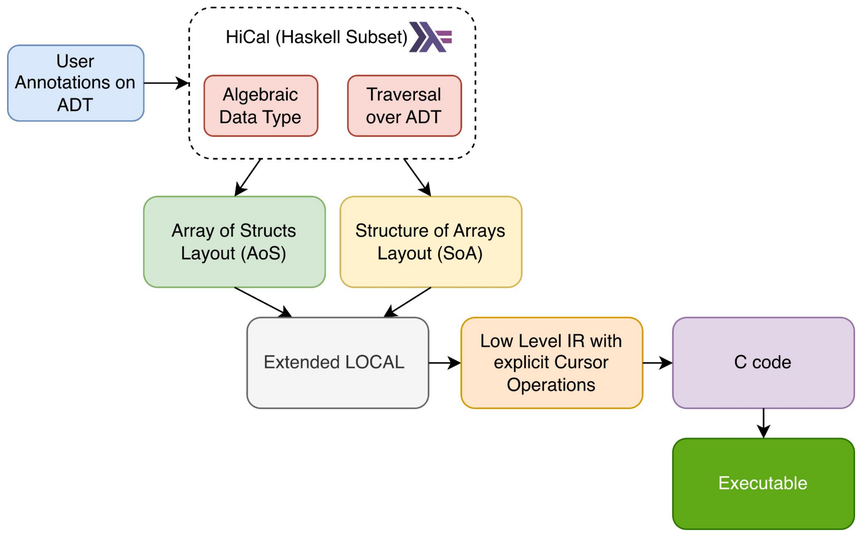
Introduction

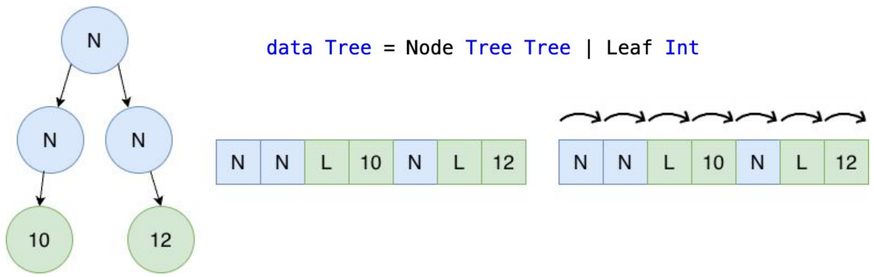
Gibbon compiler

- A tree-traversal accelerator that serializes algebraic data types (ADTs).
- Gibbon uses *regions* — chunk-allocated buffers — to store serialized algebraic data types.
- Serialization is guided by **LOCAL[2]**, a formal language that describes datatype layouts using statically computed constraints.
- Gibbon automatically rewrites traversals over ADTs into traversals over a **byte array** representation.

Hardware Benefits

- Modern hardware prefetchers are optimized for array-based access patterns.
- Traversals exhibit enhanced **spatial locality** and better **cache performance**.





Tree Representation

- The figure shows a Haskell **Tree** ADT and its **preorder serialized layout**.
- The arrows indicate order in which nodes are accessed during a **preorder traversal**.

Serialized Format

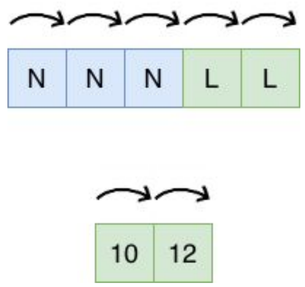
- Each **tag** (e.g., N, L) occupies **1 byte**.
- Each **integer value** in the tree occupies **4 bytes**.
- The serialized representation is laid out **contiguously in memory**.

Traditional Pointer-Based Representation

- Each tree node is allocated by **malloc**, storing pointers to left and right children.
- Tree is an **irregular pointer mesh**, with values accessed via pointer dereferencing.

Performance Issues (Pointer Based)

- Traversing pointer-based trees causes **pointer chasing[3]**, which is inefficient.
- Leads to **unpredictable cache behavior** and **slowdowns** due to poor spatial locality.



Structure-of-Arrays (SoA) Representation

- The figure shows an alternative layout where **tags** and **integer values** are stored in **separate regions**.
- One region contains all **tags** (N, L), each **1 byte**; another contains all **leaf integers**, each **4 bytes**.
- Unlike the single-region serialized format, this layout uses **two disjoint regions** (non-overlapping memory).
- Because the ADT structure is **statically known**, the traversal can **compute the next node's address** without following pointers.
- This enables more predictable access patterns and prepares the data for **vectorization-friendly** layouts.

Current Implementation Progress

- We implemented the full transformation in the **Gibbon** compiler.
- We have the ability to represent the memory representation of any datatype as **either fully factored SoA** (Each field of the data type gets its own buffer) **or Linear** which is the standard AoS representation.
- The **user can annotate each data type** in the high level haskell as either fully factored or Linear.
- For nested data types, for instance, a Tree containing a List, if the Tree is annotated as being fully factored, the user can choose the List to be either fully factored or linear.
- The **user can choose to mix SoA and AoS layouts** in the code.
- The memory representation for a datatype is global at the moment.

Benefits of an SoA Optimization

- Many architectures have **prefetchers** that optimize regular access patterns over memory buffers.

AoS representation:

- Traverses one buffer, but accesses different elements with **different constant strides**.
- Example: data constructor tags increment by 1 byte, integers/floats require different strides.
- Accessing buffer with multiple offsets can **hinder prefetching and caching**.

SoA representation:

- Each buffer contains **homogeneous data**, so access stride is constant.
- Prefetchers can **predict access patterns** more effectively because of constant stride.

SoA also benefits vectorization:

- Map-like operations over trees/lists often update leaf integers/floats.
- Example: adding a constant c to leaf values.
- Traversals have **no parent-child dependencies**, enabling parallel updates.
- All integers are in a **single buffer**, allowing **4/8-lane vectorized loops**.
- Vector loads are **efficient** since values are contiguously packed.

AoS is bad for vectorization:

- Integers are interleaved with other data, leading to **poor vectorization**.

Coming Up Next

- Optimize functions for **tail call optimization**.
- Use **mutable cursors** to update address in place to avoid copying costs.
- Exploit **vectorization** potential in recursive functions.
- Use **iterative loops** to traverse data types instead of recursion.

Conclusions

- Serializing Algebraic data types offer performance benefits because of **enhanced spatial locality**.
- Our compiler allows to convert to a **structure of arrays (SoA)** layout which can be beneficial for various optimization benefits.
- SoA can benefit locality due to more **predictable strided access**.
- SoA can benefit **vectorization** by moving homogenous data in the same region.
- Our compiler can allow the user to annotate the ADT with its layout (AoS or SoA) allowing the user to **mix AoS and SoA data types** in the same program.
- For a certain class of programs **(reduction-like)** preliminary experiments show that SoA layouts can offer faster runtime than their AoS counterparts.

Contact

Vidush Singhal
Purdue University
Email: singhav@purdue.edu
Website: vidsinghal.github.io

Milind Kulkarni
Purdue University
Email: milind@purdue.edu
Website: https://engineering.purdue.edu/~milind/

References

1. Vollmer, M., Spall, S., Chamith, B., Sakka, L., Koparkar, C., Kulkarni, M., Tobin-Hochstadt, S., & Newton, R. R. (2017). Compiling tree transforms to operate on packed representations. In P. Müller (Ed.), 31st European Conference on Object-Oriented Programming (ECOOP 2017) (Vol. 74, pp. 26:1–26:29). Schloss Dagstuhl–Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2017.26
2. Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: a language for programs operating on serialized data. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019). Association for Computing Machinery, New York, NY, USA, 48–62. https://doi.org/10.1145/3314221.3314631
3. Vidush Singhal, Chaitanya Koparkar, Joseph Zullo, Artem Pelenitsyn, Michael Vollmer, Mike Rainey, Ryan Newton, and Milind Kulkarni. Optimizing Layout of Recursive Datatypes with Marmoset: Or, Algorithms + Data Layouts = Efficient Programs. In 38th European Conference on Object-Oriented Programming (ECOOP 2024). Leibniz International Proceedings in Informatics (LIPIcs), Volume 313, pp. 38:1-38:28, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2024) https://doi.org/10.4230/LIPIcs.ECOOP.2024.38