Proposal: Accelerating Structural Sum Types

Luuk de Graaf

August 2023

Contents

1 Introduction

Abstractions in programming languages simplify code and allow for implicit exploitation of arising properties. Pure functional array languages utilize the independence between computations to implicitly parallelize array operations. This is complicated in languages that are not inherently thread-safe. Many high-performance applications adopt a data-oriented design, which centres around data transformations and avoiding abstractions around data. Both parallelization opportunities and cache efficiency are focal points of the data-oriented design paradigm.

A challenge with data-oriented design is representing non-uniform data without abstracting on the data itself. Game-engines, which deal with many clusters of data, have started using the Entity-Component-System (ECS) pattern. Systems are top-level functions that operate on all entities that implement a set of components. An entity is idiomatically a set of components, but instantiated as a globally unique index that can identify the associated data components. Rather than iterating with this index, systems utilize the implicit structure of an entity to directly iterate on the data components. An entity is effectively a persistent index across iterations and used to invoke structural changes.

Accelerate is an array processing language embedded within Haskell, which uses Algebraic Data Types (ADT's). Choosing a performant internal representation for ADT's in parallel languages is challenging due to their non-uniform size, which is a prerequisite for arrays. Currently Accelerate[25], and other functional array languages such as Futhark[23], use tagged unions to represent types with multiple variants (sum types). This increases the memory footprint and introduces branching, which can limit performance in data parallel languages. The ECS pattern preserves the functionality of heterogeneous arrays while allowing for more flexibility for the internal representation. This flexibility aligns with structural sum types, as variant changes can be defined with more accuracy.

The idea is proposed of utilizing the ECS pattern to preserve existing functionality of an array language, while extending the scope of data structures for ADT's beyond tagged unions. An extensive performance analysis is required, such that use cases of several data structures can be established. Results will be guiding in the creation of a type interface that aligns with the established properties and existing functionality of ADT's.

2 Background

2.1 Performance

There are many components that can influence the performance of a program. This grows the importance of being able to identify *bottlenecks* but also understanding the underlying technological performance considerations[13]. Within the first section fundamental optimizations related to the interaction between data and hardware are introduced. This is used to identify architecture agnostic performance considerations for array operations in the second section.

2.1.1 Optimizations

In the early days of computing, memory was seen as a way to store data indefinitely. As computational power of processors increased, the importance of main memory increased. Main memory is dependant on the advancements of random-access-memory (RAM), which stagnated due to both cost and physical limitations[7]. This put pressure on the software side to adapt to hardware components for optimizations, rather than merely the computational complexity of algorithms.

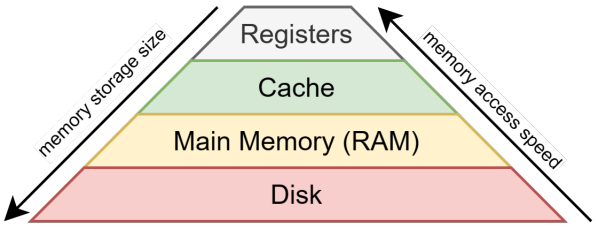


Figure 1: memory hierarchy

One of these hardware optimizations is cache storage, which accelerates memory accesses of predetermined data. The data is decided based on a cache replacement policy, often based on a temporal property. The cache operates independently of the operating system[7], and no direct control can be exercised.

Instructions Interfacing with processors is done through computer instructions. Fetching of an instruction is a memory operation, as it retrieves the instruction at the target of the program counter. Instructions operate on registers, which have distinct sizes depending on the architecture and their respective function. There are many instruction set architectures (ISA) and devices that implement distinct instruction sets. An intermediate representation (IR), such as the LLVM IR[15], can be used to create a uniform interface between these instruction sets[6]. Explicit use of these architecture exclusive instructions can be achieved through compiler intrinsics. Hardware design sometimes allows for executing specialized instructions[1], which are faster than their semantically equivalent instruction(s). This includes sacrificing accuracy for performance (floating-point), combining a sequence of instructions (arithmetic) or by parallel execution on multiple data elements (SIMD). SIMD instructions in particular are often very performant, as several steps within the execution pipeline can be parallelized. This process of instruction parallelization is called *vectorization*.

[1]Note that the term *complex instruction* is avoided, as this concerns a compact *representation* of several instructions.

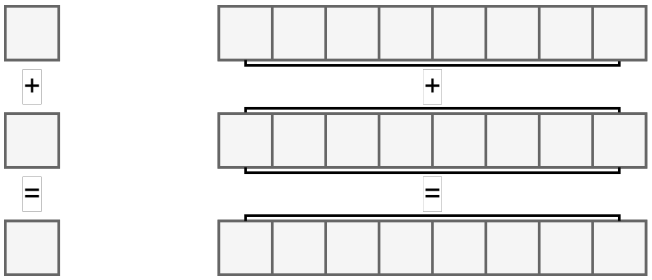


Figure 2: vectorization of a scalar addition operation

Register Pressure Registers can be considered the fastest available memory, as the data is ready to be used by an instruction. Within the context of registers this data is commonly referred to as a variable. Variables existing within registers before execution is a prerequisite for reasonable performance. This scheduling problem is to be considered a NP-complete problem[5]. Programming languages with any form of abstraction delegate this process to the compiler. This simplifies a lot of complexity, as only which data is being used by what instruction is relevant. In some cases there are too many live variables for the available registers, which *spills* the variable. This requires a variable to be stored outside registers, in a slower form of memory, and incites a delay upon use. This can be prevented by reducing the live-time of variables, reordering instructions and diversifying execution units.

Memory Access Time Semantically random-access-memory (RAM) implies that memory operations take around the same amount of time. In practice this does not hold for several reasons.

SRAM/DRAM On a modern system there often exist several different types of RAM, mostly driven by cost differences. The main forms of RAM are static RAM (SRAM) and dynamic RAM (DRAM). SRAM uses six transistors to represent a single bit, while DRAM only uses one transistor with a single capacitor. A capacitor loses electrons over time which means data has to be refreshed repeatedly to preserve its data. A refresh requires both read and write operations, which interferes with other memory operations. This makes DRAM inconsistent and on average significantly slower but much cheaper to produce due to requiring less transistors.[7]

Propagation Data is transferred by using electrical charges through semi-conductors. This creates a physical limitation dictated by physical distance and temperature. This is called propagation delay and a hard limitation to the rate at which components can operate on. SRAM is often located physically closer to the execution units to utilize the faster memory access more effectively.

DMA A processing unit needs to forward the requested data to the targeted location, which takes up processing time. Direct-memory-access (DMA) is an interface for hardware components and allows memory operations to be more organized. This allows for large scale memory operations to be performed efficiently and independently of the main processor. It requires use of several buses which means some processors must idle at seemingly random periods of time. This means that other hardware components can influence the memory access time.

Caching Due to hardware related discrepancies in memory access time, it can be beneficial to organize data according to the memory access time. One way to achieve this is by caching data, that is storing a *copy* of the data in faster accessible medium. A cache is generally made of SRAM and resides close to the processor, which allows memory accesses to be magnitudes faster than the equivalent main memory access[7]. When data already exist in the cache it is referred to as a *cache hit*, otherwise a *cache miss*. Deciding which data is cached and for how long is a cache replacement policy. Adapting to these policies simplifies the scheduling and minimizes the amount of cache misses.

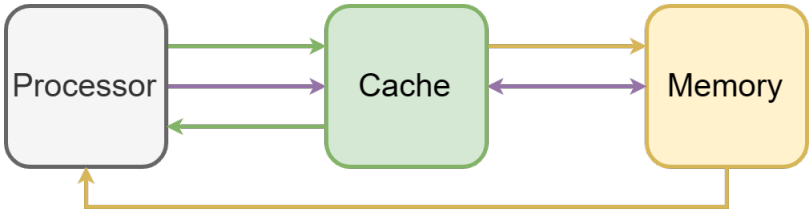


Figure 3: cache hit (green), cache miss (yellow) and replacement (purple)

Caching of data can be done after the data has been retrieved, which means the delay already has occurred. This can be avoided by requesting data in advance and storing it a cache prematurely, so called cache prefetching. This is done by analyzing future instructions (hardware) and instructions that *hint* at the future use of data (software)[3].

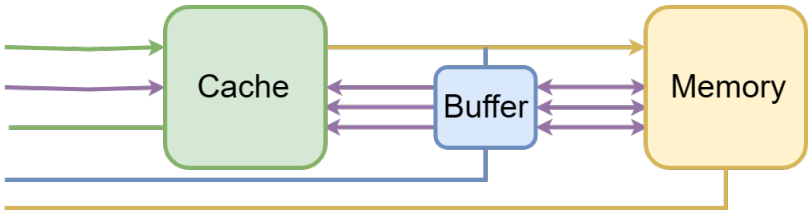


Figure 4: prefetch based on information (blue) from cache miss and processor instructions

This is harder when a branch is encountered, as both the data and the next instructions are uncertain. Speculatively executing these uncertain instructions can be performant if the overhead of redundant work remains small enough. Rather than executing unconditionally, some processors execute the most likely to happen branch based on some parameters (branch prediction)[24].

Parallelism Instruction-level parallelism is the parallel execution of multiple instructions[24]. This can be done by dividing instructions into several steps and outsourcing each step to a distinct processor unit (instruction pipelining). Shuffling the order of instructions can allow more processor units to work in parallel (out-of-order execution). Duplicate units and independent instructions allows for additional parallelism (superscalar execution).

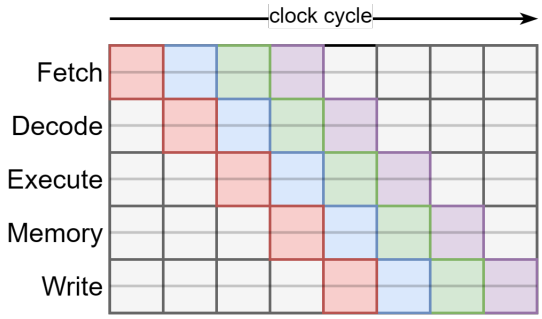


Figure 5: instruction pipelining: each color represents an instruction

Data-level parallelism executes an instruction on several data elements, such as the previously discussed SIMD instructions. Specialized processors sometimes either fully pipeline the data (vector processing) or allow for some form of autonomy (multithreading). Both share instruction fetching and decoding, but threads have their own program counter which allows for an independent sequence of instructions.

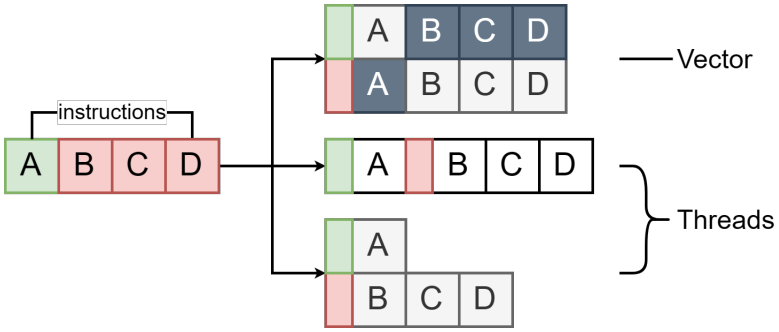


Figure 6: branch instructions: masking (vector) vs independent sequence (threads)

Execution of threads can be done concurrently, which can be useful to hide latencies (context-switching). In parallel is also possible with multi-core processors, which have several processors units (cores) that can support multiple threads. Cores are often not independent processors and might share several components with other cores, such as caches[16].

2.1.2 Principles

The are many components that can influence the performance of a program, some of which were discussed in previous sections. This makes general statements on optimizations often weak, as the interaction between these components is complex. Focusing on a particular area, such as iterating on data elements, allows for stronger arguments. Within this section previously discussed optimizations will be discussed in the context of iterating on many elements.

Contiguous A rudimentary reason for contiguously allocated data is that it creates structure, which can be used to organize data. Arrays utilize their structure to align elements, such that each element can be identified in constant time[2] through a linear function. This is also used for compound datatypes, where structure and the type can identify the memory location of each field. The structure also simplifies work distribution between threads, as it is a matter of constant offsets. For vectorization contiguous data is a prerequisite as instructions operate on singular contiguous blocks of data. If data is not spatial adjacent in memory, data must aligned temporarily or complex interleaving methods must be used[21]. In the general case compound datatypes interfere with vectorization, as spatial adjacent data is not of the same type. Parallel arrays solve this by creating a distinct array for each primitive type (Struct of Array) so that each field can be vectorized.

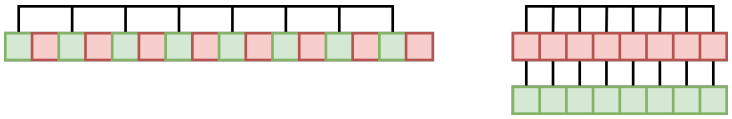


Figure 7: compound datatype array (1) and a parallel array (2)

Caches also operate with contiguous blocks of memory, which means spatial adjacent data within a fixed alignment are stored together. This lends itself well to contiguous allocated data, as it means the least amount of cache blocks are required irrespective of block size and alignment. In addition all memory accesses use the same linear function, a *constant stride* access behavior, which makes it receptive to hardware cache prefetching[3].

Access Patterns As the cache is finite a cache block can be ejected prematurely. This exists for data within the same block but also when the same block is required at multiple times. Increasing temporal locality is done by avoiding random accesses and organizing computations order around data usage. This is non-trivial in iterations where multiple indices are accessed (stencils) or computations that inherently involve random access.

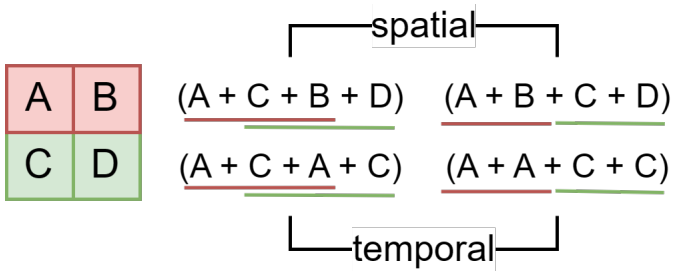


Figure 8: exploitation of spatial and temporal locality

[2]Both in *time complexity* and within *computer architecture* norms, as data access is a single instruction, unlike pointer trees and hash tables.

One way to apply this to iterating on many elements, is to iterate one subset of elements at a time (tiling). This can be further improved by also accounting for shared resources, by grouping elements that use the same resource in their instruction sequence. This is explored in raytracing[1], where spatial locality of rays is used to exploit the cache coherence within the traversal of a tree. These techniques are also important for multi-core processors, as it reduces the need for data to exist in multiple caches.

Branching Pipelining instructions is not possible when the sequence of instructions is dependant on the result of a previous instruction. This limits instruction-level parallelism, which is solved through various unconditional instruction executions[24]. Either by discarding the computed results or by *flushing* the pipeline when the wrong branch is predicted, both of which intuitively have an overhead. A compiler can eliminate[3] branches or move loop-invariant code to facilitate instruction-level parallelism[10]. These optimizations are not absolute, as an increase in instructions can pressure registers and the cache. It is also limited to instructions that cannot fail or overflow, as both can introduce unintentional side-effects.

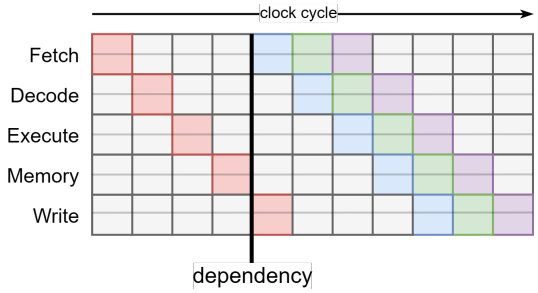


Figure 9: branch introduces dependency on execution of instruction (red)

Branching is also problematic for vectorization, as all data within a certain alignment must follow the same sequence of instructions. This can be resolved through the use of a *bitmask*, which can nullify parts of a result[10]. The additional instructions and computing bit masks can prevents performance from vectorization in certain situations. A notable application is sequential loops, where unrolling creates an opportunity to vectorize the scalar instructions. Automatic vectorization is an active field of research, and limitations have been primarily attributed to the lack of analysis information available to compilers[8]. This means branchless code and simplifying control flow allows the compiler to vectorize in more instances.

Specialized processors where an instruction sequence is distributed over many cores are limited to executing all branches. This is minimized through the use of Streaming Multiprocessors (SM), which contains several cores and fetch their own instructions. Streaming Multiprocessors operate and schedule warps, which often contain 32 threads. When divergence between these threads occurs (*branch divergence*) the instructions will in the general case be executed in lockstep[22].

[3]Either by proofing the branch will never be executed or by replacing the branch with a *conditional move* instruction, which only writes the result on true.

2.2 Data structures

A fundamental aspect of computing is data structures, which is a constant overhead for all computations. For collective operations arrays are essential; as they have a constant access time, are contiguously allocated and access can be parallelized. Composite datatypes within arrays introduce some considerations. One is the *implicit* use of parallel arrays, where each primitive datatype is stored in a distinct array. This enables vectorization opportunities, but a random access pattern might cause additional cache blocks to be cycled between. Since collective operations control the access pattern, parallel arrays are often a natural choice for array languages. The consideration for both structurally and functionally distinct data, now referred to as variant, is often complex. Variants can be represented on an individual basis (element-wise) or collectively (variant-wise). Usage and implementations of these approaches are explored in this chapter. Within this chapter the assumption is made that parallel arrays are used, as they align with the intention to vectorize operations. The example composite datatype has type A, and either has type B or type C.

2.2.1 Element-wise

For each element the choice of variant is represented, which introduces branching and in the general case will break vectorization. As variants are not grouped, functions cannot iterate on a specific variant without iterating on the complete array. The main advantage is that a variant change can be done independent of other elements, and thus can be parallelized. A practical consideration is that each element in an array must be structurally the same, that is they occupy the same memory space. This is a limitation which enforces that each index can determine the location of an element. For parallel arrays this restriction applies for all arrays individually[25].

Tagged Union Multiple variants can be represented through a tag and a fixed size data component with multiple representations, so called *union*. The tag is used to identify the current representation of the union. A naive implementation creates an array for each field of each variant, which means the memory usage is cumulative for each variant. A compact tagged union overlaps fields of variants, as only one representation can be valid at a time. This can in theory reduce the size to the largest variant and the accompanied tag, but this is complicated due to alignment requirements[25].

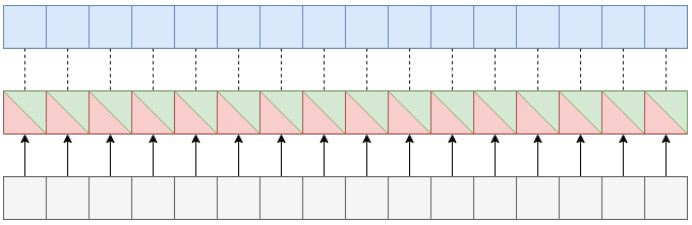


Figure 10: index implicit and tag (gray) identify the data representation

Tagged Pointer Another way to comply with elements being structurally the same is to use a form of indirection, in this case a pointer to memory. The indirection allows variants to escape the uniform size restriction, but there are several notable complications. General complications around pointers, such as being unsafe to operate on and complicating garbage collection apply. In addition, pointers that point to the same data (alias) can prevent parallelization due to possible race conditions. These can be partly solved through language constructs; such as smart pointers, immutable data or abstracting the use of pointers altogether.

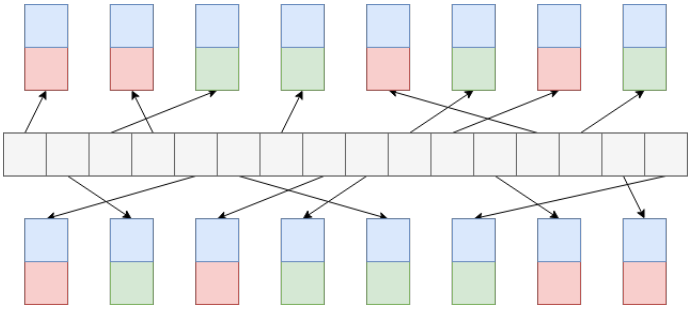
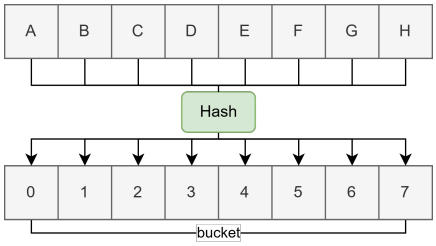


Figure 11: tag and pointer (gray) identify the location and representation.

The key issue is that a change in variant requires new data to be allocated and the pointer to be adjusted. This allocation means there is no guarantee that the data is contiguous, which in addition to the required branching prevents any vectorization efforts. The indirection and fragmented memory is also problematic for cache efficiency, as it is unpredictable and a cache block is not used effectively.

Entity A notable observation is that the re-allocation of a variant change causes the data to be not contiguous, not the indirection in itself. This can be illustrated through a hash table data structure, where a key is mapped to a value within an array (bucket). Any collective operation on the hash table can be vectorized by disregarding the hashing and using the internal array directly, as computations are inherently independent and order is irrelevant.



Entities within the ECS pattern function similarly, a form of indirection that is not used by the collective operations. Represented as a single heterogeneous array, which internally consists of several variant-wise collections. It will mean that variant choice is not *directly* represented on an element basis, which has several implications.

Stable The same entity is not guaranteed to refer to the same data, the entity is no longer stable across structural changes. The reverse also holds, the data is not guaranteed to have the same entity along iterations. This can solved by tracking the location of entities *or* annotating the data with their entity. These approaches can be complementary for performance reasons, but they are collectively isomorphic[4] through gather and scatter operations.

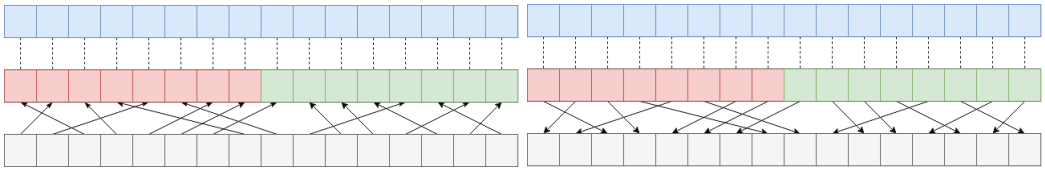


Figure 12: persistent array with indirection (1) or annotate the data with their index (2).

For many array operations stableness is excessive, as it means data is discriminated on the basis of their index. The implicit connection that data with the same index has can be represented through a (temporary) datatype.

Independent Elements can no longer change their variant independently of other elements, which can be problematic for parallelism. Depending on the way variants are grouped, there can also be a significant cost associated with regrouping variants. This can be minimized by delaying structural changes indefinitely, by using a tagged union approach. Regrouping variants is a performance consideration between the cost of regrouping and having to branch for future iterations.

[4]A single entity cannot identify the data in constant time, without an array of pointers. A data element cannot identify the entity in constant time, without the entity as data.

2.2.2 Variant-wise

Grouping variants means all data is uniform, contiguously allocated and there exist no inherit branching within the same grouping of variants. This can be achieved through an array for each variant, but also grouping *within* the same array and using segment descriptors. The latter is effectively an untagged union, where the representation is determined by the index within the array. Both allow instructions to be vectorized, but there exist several other considerations.

grouping As stated in the previous section, regrouping variants to a variant-wise collection is a performance consideration. When variants are stored in separate arrays, the amount of a certain variant must be known before allocation. When this is dependant on a computation, it can be retrieved through an additional scan or atomically[5] counting any structural change, which adds an overhead. This is not required for a singular array if the total remains the same.

immutable An important consideration for purely functional languages is that values, and therefore arrays, are to be considered immutable. This means that *updating* parts of an array efficiently is non-trivial. It must be proven that the array before update will never be used again, otherwise both arrays must co-exist in memory. This is inefficient for small updates and grows the necessity to *destructively update*[14].

automatic Most compilers support automatic vectorization of iterations with flexible bounds, where the final leftover iteration is not vectorized. This overhead can be a significant when the loop is extensively unrolled. This is minimized through epilogue vectorization, which (re-)applies loop vectorization to the remaining scalar code. In practice data must be aligned along specific byte boundaries to be vectorized, which is challenging for (dynamic) regions within an array and not always analyzed by compilers[8].

operable An undiscussed benefit of parallel arrays is that fields can be operated on independent of other fields, as they are distinct arrays. This is also possible for *regions* within an array, but this is less trivial and often requires explicit support in array languages[9].

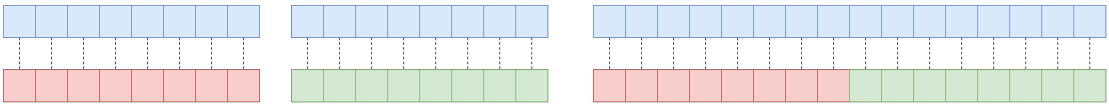


Figure 13: distinct arrays (1) or distributed in a single array (2)

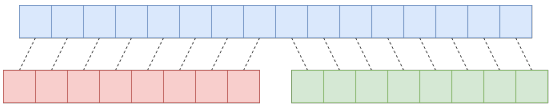


Figure 14: combination of both approaches

[5] Atomic instructions prevent interruptions by other processes and are thread-safe.

2.3 Types

The ECS pattern is arguably a reaction to the prevalence of object-oriented languages within game engines. The premise is to organizes game-logic within systems rather than components, where the relation between components is flexible[17]. In contrast to inheritance, where relations are statically determined and game-logic is embedded within an hierarchy. In this chapter the type interpretation of entities is discussed, and linked to functional languages.

2.3.1 Entity

An entity is fundamentally a composition of components, irrespective of which components can be combined. As many object-oriented languages are nominally typed, an entity is often implemented to only have one component of each. This means an entity is effectively a *set* of components. Systems operate on all entities that contain such a set of components, where components are *mutated*. Structural changes to entities are invoked through statements.

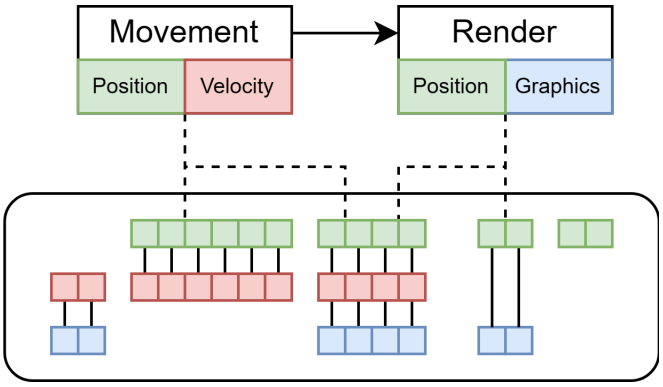


Figure 15: conceptual entity-component-system representation of systems

The ECS design pattern is arguably inherently imperative, due to prevalence of mutating state. Apecs, an ECS implementation in Haskell, achieves this imperative style through monads[4]. All these considerations are related to the interpretation of the relation between components, in essence the *type* of an entity. Rather than emulating ECS, existing type systems can be used to handle the relation between data.

2.3.2 Algebraic Data Type

An algebraic data type is a composite datatype, which is used to compose new datatypes in many functional languages. A product type (x) is the combination of datatypes, while a sum type (+) is the alternation between datatypes. It is often useful to discriminate between variations of a datatype, which is generally done through a data constructor.

```
data Maybe a = Just a | Nothing
```

Deconstructing an algebraic data type to the 'original' datatypes is done by pattern matching on a data constructor. The pattern match can exhaustively match on all variants, as the data constructors of variants are known at compile-time.

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f (Just a) = Just (f a)
fmap f Nothing  = Nothing
```

In Haskell, algebraic data types are distinguishable by name (nominally typed) and therefore explicitly declared. This means data constructors are local to the declared type and pattern matching happens within the same type. The type signature of the `fmap` function provides no information on the potential structural change of a datatype, which is relevant for the internal representation. It is possible to return `Nothing` for both cases[6]. A function that takes `Just a` and returns `Just b` ensures that the *collective identity* is preserved in a collective operation, irrespective of the data transformation. This exact definition is not possible due to being nominally typed, as `Just a` is not a type but a data constructor under the type `Maybe a`. In some cases, such as a safe division function, the introduced branching is inherit to function and is now explicit in the type definition.

```
fmap :: (a -> b) -> Just a -> Just b
fmap f (Just a) = Just (f a)

divide :: Int -> Int -> (Just Int | Nothing)
divide _ 0 = Nothing
divide n m = Just (n `div` m)
```

In this case a function is defined on the structure of a type, the data constructor of algebraic data types. `Maybe a` is now an alias for the mutually exclusive relationship `Just a | Nothing`, rather than a unique and standalone type. This generalizes variance to be between all types. OCaml calls these *polymorphic variants*[7], while other functional languages generally refer to them as *extensible* or *open sum types*. A motivating example is that a collection of `Maybe a` can be an heterogeneous collection or two variant-wise collections of `Just a` and `Nothing`[8]. While the former involves branching for `fmap`, the latter can ignore the `Nothing` collection and vectorize the `fmap` function. This flexibility aligns with the intention to create a modular system that is agnostic to the internal representation.

[6] `Just b` is not possible as it can only be inferred through `Just a` and the `a -> b` function.
[7] OCaml also implements nominally typed sum types, so called *variants*.
[8] As `Nothing` does not hold data, a size descriptor is sufficient.

3 Related Work

Within this chapter Accelerate, the implementation language, will be discussed in the context of iterating on non-uniform data. This is extended by looking at integration of non-uniform data within other functional data-parallel array languages. It will be concluded by comparing existing ECS libraries and their interpretation of the underlying data structure and structural changes.

3.1 Accelerate

Accelerate is a data-parallel array language deeply embedded within Haskell. An abstract syntax tree (AST) is created and optimized by Accelerate within Haskell's runtime system. This greatly improves useability, as it can function as an Haskell library, at the cost of executing code within another runtime system. The garbage collection of the Haskell runtime system is speculated to hinder performance[26]. A type-safe interface to the compiler infrastructure LLVM enabled the creation of two backends: GPU and multi-core CPU's[18]. These backends can be used to execute a small set of collective operators in parallel; such as `map`, `fold` and `stencil`.

```
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

The inherit thread-safety and fixed set of collective operators guarantee a consistent application of data-level parallelization. It in addition allows for these collective operators to be heavily optimized in isolation, but also in relation to other collective operators. A naive implementation of `dotp` would create an intermediate array for the results of the `zipWith` function[19]. Fusing these operations would eliminate an iteration and the intermediate array, at the cost of potential register pressure. Accelerate fuses these collective operations, unless the fusion introduces duplicate work or the `compute` function is explicitly called.

As Accelerate is embedded within Haskell, it uses algebraic data types and tuples for composite datatypes. Datatypes must be *lifted* into the abstract syntax tree of Accelerate, which is implemented for all native types. As algebraic data types are not native, pattern synonyms are used to create an isomorphic mapping *at compile-time* between the Haskell and the Accelerate datatype. As pattern matching cannot be overloaded in Haskell, the `match` operator is used to inject the required branching[20].

```
mkPattern ''Maybe

apply :: (a -> b) -> Maybe a -> Maybe b
apply f = match (fmap f)

fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f (Just_ a) = Just_ (f a)
fmap f Nothing_  = Nothing_
```

Currently Accelerate uses a non-compact tagged union representation for sum types. Research has been done on a compact tagged union representation for parallel arrays, which has been named a *Recursive Tagged Union*[25]. The representation uses a unified tag for nested sum types, which optimizes memory usage at the cost of tag (de-)construction. It has been partly implemented in Accelerate, and has not yet been benchmarked.

3.2 Futhark

Futhark is a functional structurally typed data-parallel array language. Research has been done on including sum types to the Futhark compiler[23], which has been implemented. A non-compact representation is used, but references the potential to overlap only fields of the same *type*. While not the most compact, it avoids re-interpretation of undefined data.

3.3 Apecs

Apecs[4] is an ECS library in Haskell, with experimental support for concurrency through the Software Transactional Memory (STM) monad[12]. The STM monad utilizes atomic instructions to achieve concurrency within a monadic context. Entities are stored with an integer dictionary, but can be explicitly stored in a sparse fixed size *cache* which allows for constant access time.

3.4 Specs

Specs is an ECS library in Rust[11], with native support for parallelism. It supports several storages such as: a tree data structure, dense vector, sparse vector and an hashmap. Iteration on entities is done by specifying the relation between the components: join, optional and excluded. An entity can be *flagged* (tag) to denote events such as the destruction of the entity.

3.5 Unity

Unity is a game-engine which released their production ready Data-Oriented-Tech-Stack (DOTS) in 2022, which centres around an ECS implementation which they have patented[2]. It uses LLVM to optimize and vectorize C# code (burst compiler) and has access to their internal c++ library for parallel and concurrent tasks (job system). The ECS implementation heavily employs the .NET Compiler Platform Roslyn to generate source-code for iterations on user-defined datatypes. An `Entity` is idiomatically a composite datatype but implemented as an `identifier` with a `version`[9]. A single array keeps track of all entities, without data components. Each `Entity` and related data components are stored in a `Chunk`, a fixed size buffer of 16000 bytes, alongside other entities with the same `Archetype`[10]. Instead of pattern matching, a `Query` matches on certain properties of entities, such as containing a certain component. Collective operations, such as a `Job`, operate on a `Chunk` directly. Structural changes can be queued in parallel, but create a sync-point when executed and involves entities to be moved between chunks. This is not required for an `IEnableComponent`, which functions as an tagged component akin to the `Maybe a` type. A notable observation is that delaying structural changes has side-effects as queued structural changes are not taking in account.

[9] Destroying an entity increments the `version`, which means copies of entities know they are destroyed and the `identifier` can be reused at a later point.

[10] An `ISharedComponent` extends this to the value of a component, which means each value has a unique `Chunk` and the value can be stored within the `Chunk` header.

4 Methodology

The premise is to efficiently iterate on non uniform data, which aligns with existing functionality of pure functional array languages. This can be separated into three distinct chapters. The first chapter investigates compatibility with existing collective array operators in Accelerate. The second chapter attempts to identify *run-time* performance considerations between the discussed data structures. This will be guiding in the third chapter, which aims to create an interface that is integrable within the type system of Haskell and provides sufficient type information for optimizations.

The research on the *Recursive Tagged Union* in Accelerate has centered around memory usage[25]. The Futhark paper goes in depth on type theory of structural sum types and minimal research on the performance implications[23]. Within this a wide-range of data structures will be explored from a performance standpoint, which sets it apart from previous research on this subject. There also exist a lot of research on vectorization of non-array data structures, which employ a similar approach of directly operating on the internal data structure. While the ECS pattern is prevalent in many applications, to my knowledge there is limited (public) research on the optimizations done within these discussed libraries. The goal is not to implement ECS, but to use applicable optimizations from ECS within the context of sum types in functional data-parallel languages.

1. Which collective array operations in Accelerate can be supported?
2. What are the performance considerations between the data structures?
3. What interface provides sufficient information to support optimizations on non-uniform data?

4.1 Implementation

As array languages operate on arrays, the functionality must be similar to remain compatible. In principle this means *delegating* the collective operation to the underlying data structure, based on the type of the function. This cannot be done when an index is explicitly used, such as the function `imap` as the index would be local to the underlying data structure. Internally `imap` is implemented using an array of generated indices through the `generate` function. In theory this means that it can be supported with minimal interference, by reimplementing these functions and replacing the way indices are provided for those functions. This is likely less trivial for element indexing, as it requires an array of pointers/indices to be maintained. An initial assumption is that this is unfeasible to do without sacrificing performance *or* it requires integration within Accelerate. An initial implementation should provide additional context to the feasibility of this.

4.2 Comparison

Preliminary benchmarks must be done on the *runtime* performance of the discussed data structures, for both LLVM backends. As these are all forms of arrays and regions within arrays, existing functionality in Accelerate allows for such benchmarks. A non-exhaustive list of these data structures is:

1. Element-wise
   (a) Naive tagged union
   (b) Recursive tagged union
   (c) Futhark suggested tagged union
   (d) Entity (without using the underlying data structure)
2. Variant-wise
   (a) An array for each variant
   (b) An array for each distinct fields
   (c) Grouping within a single array

Within these comparisons and benchmarks, the focus lies on identifying properties and overall performance. For this the optimizations discussed within the background will be used. A non-exhaustive list of these properties are:

1. Vectorization (automatic)
2. Array fusion (memory bandwidth, register pressure)
3. Structural changes of variants (scan, sorting, grouping, allocation)
4. Sorted arrays (cache coherence, thread divergence, branch prediction)
5. Transform into persistent array (gather, scatter)
6. Suitability for destructive updates and task parallelism

The results will be guiding in deciding the feasibility of the internal representations. This will also give an initial idea on what information is relevant for optimizations and the manual control required to create a performant program.

4.3 Interface

The main issue with sum types in Haskell (for this cause) is that structural changes to elements is not apparent. There are various ways to implement structural sum types in Haskell, many of which have been implemented in numerous libraries. A non-exhaustive list of used concepts:

1. Type Families
2. Template Haskell
3. Generics
4. GADT's
5. Type Roles (GHC)

Implementations will be based on the results of the previous section. In Accelerate the tag and union type has not yet been fully implemented[25], which I intend to continue on. The representation/interface for nominal types has also not yet been completed[25], which I do not intend to continue on.

5 Planning

01-08-2023 Official Start

08-08-2023 Break (1 week)

15-08-2023 Preliminary implementation (1 week)

05-09-2023 Complete previous research / union (3 weeks)

12-09-2023 Preliminary Benchmarks (1 week)

26-09-2023 Draft Chapter 1 (2 weeks)

10-10-2023 Draft Chapter 2 (2 weeks)

24-10-2023 Reformulate background information (2 weeks)

31-10-2023 Complete implementation (1 week + parallel to writing)

07-11-2023 Break (1 week)

14-11-2023 Preliminary experimenting with interface (1 week)

28-11-2023 Draft Chapter 3 (2 weeks)

12-12-2023 Implementation optimizations (2 weeks)

19-12-2023 Complete interface (1 weeks + parallel to writing)

02-01-2024 Break (2 weeks)

30-01-2024 Write Thesis (4 weeks)

13-02-2024 Buffer (2 weeks)

References

[1] Daniel Meister 0002, Jakub Boksanský, Michael Guthe, and Jirí Bittner. On ray reordering techniques for faster gpu ray tracing. In Dan Casas, Eric Haines, Sheldon Andrews, Natalya Tatarchuk, and Zdravko Velinov, editors, *I3D '20: Symposium on Interactive 3D Graphics and Games, San Francisco, CA, USA, September 15-17, 2020*, pages 13:1–13:9. ACM, 2020.

[2] Joachim Christoph Ante and Tim Johansson. Method and system for improved performance of a video game engine, March 2020.

[3] J. L. Baer and T. F. Chen. An effictive on-chip preloading scheme to reduce data access penalty. In *Supercomputing 1991*, pages 179–186, November 1991.

[4] Jonas Carpay. apecs: Fast entity-component-system library for game programming. `https://hackage.haskell.org/package/apecs`.

[5] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[6] David Chisnall. The challenge of cross-language interoperability, 2013.

[7] U. Drepper. What every programmer should know about memory. *Red Hat, Inc*, 2007.

[8] Jing Ge Feng, Ye Ping He, and Qiu Ming Tao. Evaluation of compilers' capability of automatic vectorization based on source code analysis. *Scientific Programming*, 2021, 2021.

[9] Martijn Fleuren. Independently computed regions in a data parallel array language, 2020.

[10] A. Fog. Optimizing subroutines in assembly language: An optimization guide for x86 platforms, 2008.

[11] Amethyst Foundation. The specs book. `https://specs.amethyst.rs/docs/tutorials/`.

[12] Tim Harris, Simon Marlow, and Simon Peyton Jones. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM Press, January 2005.

[13] Paul Hsieh. Programming optimization, 2016.

[14] Georgios Korfiatis, Michalis A. Papakyriakou, and Nikolaos Papaspyrou. A type and effect system for implementing functional arrays with destructive updates. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Federated Conference on Computer Science and Information Systems, FedCSIS 2011, Szczecin, Poland, 18-21 September 2011, Proceedings*, pages 879–886, 2011.

[15] Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[16] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[17] Adam Martin. Entity systems are the future of mmog development, 2007.

[18] Trevor L. McDonell, Manuel M T Chakravarty, Vinod Grover, and Ryan R Newton. Type-safe Runtime Code Generation: Accelerate to LLVM. In *Haskell '15: The 8th ACM SIGPLAN Symposium on Haskell*, pages 201–212. ACM, September 2015.

[19] Trevor L. McDonell, Manuel M T Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP '13: The 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2013.

[20] Trevor L. McDonell, Joshua D. Meredith, and Gabriele Keller. Embedded pattern matching. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. ACM, sep 2022.

[21] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. *PLDI'06*, pages 132–143, June 2006.

[22] Nvidia. Nvidia tesla v100 gpu architecture.

[23] Robert Schenck. Sum types in futhark, 2019.

[24] Michael S. Schlansker, B. Ramakrishna Rau, Scott Mahlke, Vinod Kathail, Richard Johnson, Sadun Anik, and Santosh G. Abraham. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Technical Report HPL-96-120, Hewlett–Packard Corporation, 2000.

[25] Rick van Hoef. Accelerating sum types, 2022.

[26] Bart Wijgers. Investigating the performance of the implementations of embedded languages in haskell, 2022.