StellerJ: A Compiled Approach to the Functional Array Language Paradigm

Conor O'Brien
College of William & Mary
Email: csobrien@wm.edu

***Abstract*—StellerJ is a dialect of the functional array programming language J, targeted at LLVM. This paper examines the potential array languages like J have with high-performance computing by exploring the potential benefits of compiling instead of interpreting such languages. Compiling tacit structures involves dissecting the idioms used to construct them into compilable and analyzable fragments. The StellerJ implementation is compared for execution speed against equivalent programs in J, and in two approaches of programming in C++ (vectorized and unvectorized).**

I. Background

Languages like J [1] and APL [2][3] are two common and influential instances of the functional array language paradigm. They are defined primarily by two capabilities: Functional programming and array programming. While usually implemented via interpreters with code being developed via read-eval-print loops, both language support standalone scripts, which are executed through the interpreter. This provides our motivation: We can leverage the static nature of such script files and provide corresponding compiled binaries, which reduces the overhead of interpretation, as well as offers a higher degree of control over static optimization.

There are two main challenges in compiling a functional language such as J. The first challenge designing an effective compiler for first-order functions (verbs). Haskell [4] is an example of an effective compiled functional language, although it achieves compilation of tacit structures by defining equivalences to corresponding explicit structures. The second challenge emerges from J's powerful, often-tacit syntax, which tends to create non-obvious control paths.

With these challenges, however, come great benefits: Languages like J, being array-based, provide great opportunities for parallelism [5]. Since verbs explicitly operate on lists and tables, many operations, especially mathematical ones, are naturally expressed via vectorization, leading to intrinsic and easily-identifiable speed-ups by merely using conventional operators.

The IR we target is LLVM [6]. LLVM is a useful target IR as it is portable, able to be deployed on many systems. Further, it comes with a slew of existing optimizations via the `opt` tool, which relaxes the need to reimplement common optimization techniques within the scope of this compiler. This endeavor will be similar in effect to aplc [7], an existing compiler for APL also targeting LLVM. LLVM is particularly useful because of the native vector support through mathematical commands, such as `add` and `mul`; the LLVM IR can be further targeted towards specific vectorization methods whenever possible.

Fig. 1. The majestic Steller's Jay for which StellerJ is named. Picture Credit: Julietfiss on Wikimedia.

Naturally, there are two parts to the functional array language paradigm: functions and arrays. We will review some principles of each.

A. Functions

Functions, also referred to as *verbs*, are one of the primary constructs in appropriately-named functional languages. Different from imperative programming, functions may be operated upon and combined in different ways to produce new functions, often without need of referencing data by name. One central aspect of functional programming is composition: Let f, g be functions; then, their composition

$$(f \circ g)(x_1, \ldots, x_n) = f(g(x_1, \ldots, x_n)).$$

While functions conventionally can take an arbitrary specified number of parameters, it is usual in the context of J and APL to talk about only one- and two-argument functions, called *monads* and *dyads*, respectively. It is in this context the rationale behind the term verb emerges. Carrying on with this analogy to speech, *nouns* are the arguments to function, and represent data. Verb modifiers are either called *adverbs* or *conjunctions*, depending on whether they act upon one verb, or upon two in concert. Thus, composition as defined above is a conjunction.

One important adverb is *reduction*, also known variously as folding, insertion, and injection. The term insertion makes the adverbs behavior most clear: It modifies a verb by "inserting" it between every element in an input vector. We can consider the summation operation on a vector to be addition reduction. Although there are two directions one can reduce in, we shall assume right-to-left reduction (i.e., computed as $f(a_0, f(a_1, \cdots f(a_{n-1}, a_n))))$ symbolically as $\langle f \rangle$.

Which adverbs and conjunctions a language defines are those which are useful to accomplish tasks. In theory, any modification of one or two verbs can be defined as an adverb or conjunction. For the sake of this paper, looking forward to potential optimizations, we will define here the inner product. Similar to matrix multiplication, the inner product is a conjunction $f \otimes g$ which operates upon two input matrices, say A and B. Suppose $\dim(A) = [N, K]$ and $\dim(B) = [K, M]$. Then, $\dim((f \otimes g)(A, B)) = [N, M]$ and

$$(f \otimes g)(A, B)_{i,j} = f(G(i, j)),$$

where

$$G(i, j)_k = g(a_{i,k}, b_{k,j}),\ 0 \leq k < K.$$

In addition to composition and inner product, we give meaning to a *train* of consecutive verbs. While not strict conjunctions, *fork* and *hook* are important verb modifiers. A fork of 3 verbs $\{f, g, h\}$ is defined monadically as dyadically:

$$\begin{aligned} \{f, g, h\}(y) &= g(f(y), h(y)) \\ \{f, g, h\}(x, y) &= g(f(x, y), h(x, y)) \end{aligned}$$

g is always a dyad, and the arity of f and g varies with the arity of the invocation. A hook $\{g, h\}$ is defined for both arities:

$$\begin{aligned} \{g, h\}(y) &= g(y, h(y)) \\ \{g, h\}(x, y) &= g(x, h(y)) \end{aligned}$$

In general, a train of verbs is interpreted by parenthesizing the rightmost 3 verbs into a fork, or the remaining 2 verbs into a hook, if less than 3, until a single verb remains. For example, the 6-train $\{f_0, f_1, f_2, f_3, f_4, f_5\}$ is interpreted as $\{f_0, \{f_1, f_2, \{f_3, f_4, f_5\}\}\}$: two forks and a hook.

These mathematical concepts correlate directly with J concepts. We may therefore express the discussed mathematical notation in the language of J. Let + and $\star$ represent their usual mathematical operations, `/` be a postfix adverb representing reduction, and `.` be an infix conjunction representing the inner product $\otimes$, we may define the matrix product as

```
mat_prod =: +/ . *
```

B. Arrays

Arrays are a common data structure in program, among the most simple. In general, array languages generalize the concept of the 1D array (the list) to *tensors*, an array whose dimension can be arbitrary. Matrices are equivalent to 2-tensors. We choose to work with packed (i.e., non-sparse) arrays for the sake of simplicity. We also use a simple linear representation of arrays, where multiple dimensions are concatenated, and dimension boundaries are implied by an additional dimension container. So, to store an array, we store the following information:

1) A linear array of the data;
2) The total amount of data present;
3) A linear array of the dimension of the data; and
4) The total amount of dimensions present.

II. Design

We designed StellerJ as a dialect of J which serves as a frontend for LLVM. The language's syntax is a subset taken directly from J. It features a drastically reduced vocabulary for the scope of this project, although it still features the usual arithmetic operators.

A. Vectorization

Vectorization emerges here from the union between tensors and verb modifiers. When manually optimizing reduction operations using SIMD (Single Instruction, Multiple Data), the usual approach is to perform that same reduction on groups of SIMD registers and the corresponding SIMD operation. Summation is usually optimized this way: Read consecutive elements into a SIMD register and add that to a running SIMD register sum until no elements remain. This optimization, though it computes the sum in a different manner (as, say, $(a_0 + a_4 + \cdots) + (a_1 + a_5 + \cdots) + \cdots$ instead of $a_0 + a_1 + \cdots$), works due to the associativity of addition. However, even non-associative operators can be optimized similarly, as long as they can be restated. Right-folding subtraction is equivalent to summing the original vector multiplied pairwise by the repeating vector $[1, -1]$: $a - (b - (c - d)) = 1 \cdot a + -1 \cdot b + 1 \cdot c + -1 \cdot d$, for example.

There are a limited set of SIMD instructions, so the set of operations explicitly able to be formed using them is also limited. If an operation is able to be efficiently formed using SIMD instructions, we term it *vectorizable*. We call a verb *vector reducible* if it is can be expressed in terms of vectorizable, associative operators.

Thus, for any vector reducible verb f, we have the the reduction $\langle f \rangle$ can be SIMD optimized. For any vector reducible verb f, and for any vectorizable verb g, we have the inner product $\langle f \rangle \otimes g$ can be SIMD optimized. Using these two optimizations, we can generally provide SIMD optimizations for verbs following these constructions.

B. StellerJ

As mentioned, StellerJ is a compiler frontend, which means it consists of three steps: Tokenization, Parsing, and IR generation.

1) Tokenization: The J grammar is defined relatively simply, and can be inferred from the behavior of the J verb primitive `;:` (Words). Tokenization in StellerJ proceeds by first scanning the input for tokens and thereafter classifying them with a classifier. We first approach the problem by creating a regular expression which scans through the input string

greedily for tokens, according to the following preference order.

1) String: A single quote, followed by 0 or more of any combination of non-single quotes, or paired single quotes, followed by a final single quote.
2) Comment: The phrase `NB.` not directly followed by other postfix ngraph characters (`.` or `:`), expanded to the next newline character, or the end of the input, if no more newline characters remain.
3) Word: An alphabetic character of any case, followed by 0 or more alphanumeric characters and/or underscores, followed by 0 or more postfix ngraph characters.
4) Spaces: 0 or more space characters (0x20).
5) Numbers: A number or underscore, followed by any combination of alphanumeric characters, underscores (J's negative sign), and periods. Each number may recursively be followed by a space and another number. The entire number expression may be followed optionally by 0 or more postfix ngraph characters.

To match J's parsing behavior, we do not emit space tokens in our tokenizer, nor do we group together numbers which include a colon. We classify these scanned tokens according to their lexical content, as follows. This also follow's J behavior of first producing an untagged list of tokens, and classifying them using simple lexical rules.

1) The set of available verbs, adverbs, conjunctions, control words, and copula are defined explicitly in a corresponding list, and tagged accordingly. We also tag `for_var.` manually, since it is a single expression with variable content, namely, `var`.
2) We group comments by matching against the same behavior described above.
3) Words are defined to be any token whose initial character is alphabetic.
4) Numbers are defined to be any token whose initial character is numeric or the underscore.
5) Strings are defined to be any token whose initial character is a single quote.
6) Each parenthesis is grouped separately.
7) No other tokens are recognized.

After this, all tokens are correctly categorized and emitted as a list to the next stage.

2) Parsing: Parsing in StellerJ is divided into two steps. First, we have an initial parsing step, which performs elementary static analysis to ensure variables make sense, and groups the different parts of speech into separate tree nodes. The second step, grouping, gives names to tree nodes, as well as traverses nodes to pack the conceptual representation of a tree structure. The main conceptual categories proceed as follows:

1) Variable Assignment (labeling data)
2) Verb Assignment (unimplemented)
3) `echo` statements (prints to standard output)
4) `time` statements (used for the evaluation)

3) IR Generation: The final step in StellerJ is the IR Generation step. LLVM is emitted in accordance to the identified group structures. The most meaningful statements are copula (assignment) statements, which modify the state of user-specified variables. For certain functions, such as state generation for the evaluation tasks and the timing functions, we wrote C++ code and extracted the generated LLVM code into a header file which we ship with every compiled StellerJ program.

Between our two main datatypes (integer scalars and integer tensors), we implemented the usual arithmetic operators (addition, subtraction, multiplication, and division represented respectively by `+`, `-`, `*`, and `%`) for both scalar and tensor cases.

We design an LLVMEmitter capable of writing LLVM code through a Ruby API to targeted functions, allowing iterative development of multiple functions concurrently while compiling. Types are abstracted, but it also still allows fine-grained control over the LLVM output. It consolidates the effort required for instantiating `structs` in LLVM, and allows interfacing with the custom `JITensor` datatype on a high level. The emitter also keeps track of variable size to ensure proper alignment and function signatures to allow the LLVM programmer to specify arguments with the emitter deducing the types.

Each of the four described statement types in Parsing has its own compilation strategy with various substeps. Different structures must be compiled for variable assignments to reflect the corresponding datatype. While integer assignment is fairly straightforward, any initialization of a tensor (temporary or permanent) requires allocating memory in a few different locations, on top of maintaining correct labels for the contents and parameters of the tensor. Beyond the top-level assignment statements, we used a depth-first recursive compilation strategy for converting from expressions to LLVM instructions. In general, temporary registers are created every time the algorithm references data a variable points to; although this sometimes results in duplicate loads, this is trivially optimized away through `opt`.

For compiling adverbs and conjunctions, it was necessary to create intermediate LLVM functions, which are then passed through the program as data via function pointers. While some compilations could be inlined, such as `echo +/ 1 2 3`, more complex compilations such as `echo x +/ . * y` necessitate either intermediate functions or to be fully inlined, the latter of which is incredibly difficult to get correct.

C. Internal Representation

This implementation of StellerJ uses LLVM 64-bit signed integers as scalars. Since J's native vectors are regular arrays, they can be stored as flat arrays with an appropriate associated dimension. (E.g., a vector whose shape is 3x4x2 may be represented as the pair consisting of an array of 24 elements and the length-3 array `3 4 2`.)

D. Syntax

Our program compiles definition statements of arbitrary complexity, restricted to the usual arithmetic operators. For

example, we have the following:

```
id3       =: 3 3 $ 1 0 0  0 1 0  0 0 1
all_fives =: 3 3 $ 5 5 5  5 5 5  5 5 5
id3_fives =: id3 * all_fives
my_array  =: 3 6 $ 1 2 3
echo id3_fives +/ . * my_array
echo id3_fives +/ . + my_array
id3       =: id3 + id3 + id3
echo id3
echo */ 1 1 1 1 + i. 4
```

This code showcases the `$` operator retrofitted to determine static array initialization; pairwise multiplication of tensors; recycling; the generality of the inner product and reduction modifiers; the sanctity of variable read-writes; and the correct evaluation of complex expressions.

III. Evaluation

We compare four modes of operation: StellerJ, J, C++, and C++ with SIMD. We compile the compiled modes using maximal optimization `-O3`. We compare for performance on the following 3 tasks:

1) Array generation and summation: **Description:** Allocate an array of 3.2×10^8 64-bit integers, populate the array with consecutive integers starting at 1, and iterate over the array, procuring the sum. **Approach:** In J, a script version of the natural approach `+/i.320000000` suffices. In unvectorized C++, pre-allocating a `std::vector` using the `reserve` method, and implementing the description imperatively . In vectorized C++, SIMD programming is used for the vector summation step. In StellerJ, we use the following code for 5 trials:

```
arr =: i. 320000000
time +/ arr
time +/ arr
time +/ arr
time +/ arr
time +/ arr
```

2) Matrix multiplication: **Description:** Allocate space for two 1024x512 matrices (dimensions related by transpose), populate them with random values, and compute their matrix product. **Approach:** In J, we leverage the dot-product conjunction to define matrix multiplication verb defined earlier as `mat_prod =: +/ . *` (sum together multiplied corresponding elements). In unvectorized C++, we implement a naïve imperative matrix multiplication algorithm. In vectorized C++, we again use SIMD programming in the computation step by stepping along a fixed size. In StellerJ, we use the following code for 5 randomized trials:

```
A =: 1024 task2 512
B =: 512 task2 1024
time A +/ . * B
A =: 1024 task2 512
B =: 512 task2 1024
time A +/ . * B
A =: 1024 task2 512
B =: 512 task2 1024
time A +/ . * B
A =: 1024 task2 512
B =: 512 task2 1024
time A +/ . * B
A =: 1024 task2 512
B =: 512 task2 1024
time A +/ . * B
```

3) Multiplying multidimensional arrays: **Description:** Allocate space for three multidimensional arrays (each of size 70x40x30x64), and populate the first two with random integers. Then, compute their elementwise product and store the result in the third. **Approach:** In J, multiplication implicitly vectorizes, so this is achieved using the `*` verb readily. In unvectorized C++, we simply iterate using the same iteration basis over each array and multiply corresponding values. In vectorized C++, we batch multiplications together using SIMD programming. In StellerJ, we used the following code for 5 randomized trials:

```
A =: task3 70 40 30 64
B =: task3 70 40 30 64
time A * B
A =: task3 70 40 30 64
B =: task3 70 40 30 64
time A * B
A =: task3 70 40 30 64
B =: task3 70 40 30 64
time A * B
A =: task3 70 40 30 64
B =: task3 70 40 30 64
time A * B
A =: task3 70 40 30 64
B =: task3 70 40 30 64
time A * B
```

IV. Results

In our evaluation, we compared the performance of four different language configurations on a common hardware setup, the Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz processor. The reported results are based on the average execution times from five independent trials. It is important to note that the reported times exclude any initialization or input/output operations, focusing solely on the execution of the respective programs. The results are presented both in tabular format in Figure 2 and visually in Figure 3.

Among the tested configurations, StellerJ consistently performed the worst across all three tasks, exhibiting significantly slower execution times. On the other hand, J consistently outperformed the competition by a wide margin in tasks 1 and 2, demonstrating its superior efficiency in these scenarios. Perhaps unsurprisingly, the unadorned C++ implementation showed the best performance in task 3, albeit by a narrow margin. This result suggests that the additional overhead

Execution time (s)

task	StellerJ	J	C++	C++ SIMD
1	1.515	**0.1629**	0.8182	0.8452
2	3.954	**0.1306**	1.742	0.7552
3	0.04424	0.01431	**0.003862**	0.006830

Fig. 2. Table showcasing our results; lower is better.

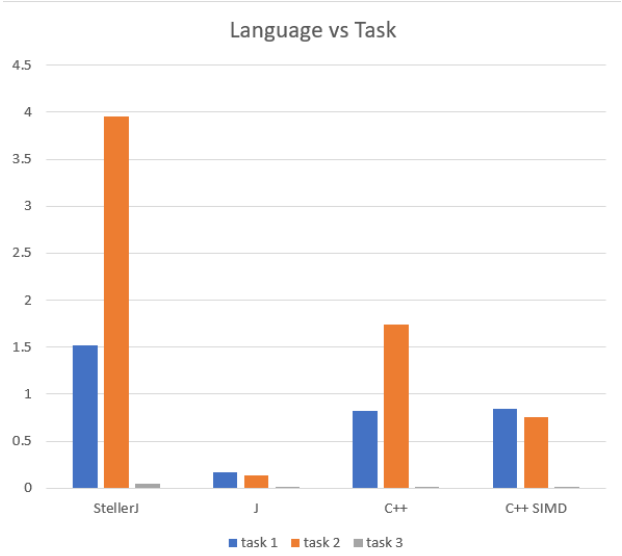

Fig. 3. Visualization of Fig. 2.

introduced by SIMD vectorization did not yield substantial performance improvements in this particular task.

Figure 2 provides a comprehensive overview of our results, depicting the execution times (in seconds) for each task and language configuration. Lower values indicate better performance. The table clearly illustrates the disparities in execution times across the different configurations, reinforcing the conclusions mentioned earlier. Additionally, Figure 3 offers a visual representation of the results presented in Figure 2, providing a graphical perspective on the performance variations.

V. Conclusion

In this study, we evaluated the performance of four different language configurations on three computational tasks. While J emerged as the most efficient configuration for tasks 1 and 2, and unadorned C++ proved to be the best option for task 3, our architecture StellerJ significantly underperformed across all three tasks, despite using the same optimizations as C++ `-O3`.

We suspect that the underperformance of StellerJ may be due to the way functions are tagged for specific optimizations, which is currently beyond our scope of understanding and debugging. While we initially planned to continue optimizing StellerJ by taking advantage of the vectorizable nature of functional operators, our results suggested that this approach was not effective. Thus, we did not pursue further optimization in this direction.

Despite the limitations we encountered, there are many avenues for future work. For example, we only investigated the potential for SIMD optimization on a limited set of modifiers. There is ample opportunity to explore other modifiers for potential SIMD optimization. Furthermore, there are many useful features of the J language that could be incorporated into StellerJ, such as more of J's verbs, support for additional datatypes, and direct functions and control structures.

Our study highlights the importance of carefully considering language choice and optimization strategies for specific computational tasks, and in part demonstrates that being an interpreted language is not necessarily a burden to its performance. Although our findings were not ideal for StellerJ, we learned valuable lessons about language implementation, design and optimization, as well as an increased familiarity with the tools and functionality LLVM has to offer.

References

[1] R. K. Hui, K. E. Iverson, E. E. McDonnell, and A. T. Whitney, "APL\?" in *Conference Proceedings on APL 90: for the future*, 1990, pp. 192–200.

[2] P. S. Abrams, *An interpreter for" Iverson notation"*. Stanford University, 1966.

[3] D. Bowman, "Dyalog APL/W," *ACM SIGAPL APL Quote Quad*, vol. 23, no. 2, pp. 16–23, 1992.

[4] S. P. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler, "The Glasgow Haskell compiler: a technical overview," in *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, vol. 93, 1993.

[5] R. Bernecky, "The role of APL and J in high-performance computation," in *Proceedings of the international conference on APL*, 1993, pp. 17–32.

[6] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[7] jiixyj, "aplc," https://github.com/jiixyj/aplc, 2012.

[8] "`5!:` Representation," https://www.jsoftware.com/help/dictionary/dx005.htm, [Online; accessed 6-April-2023].

[9] A. J. Perlis and S. Rugaber, "Programming with idioms in APL," *ACM SIGAPL APL Quote Quad*, vol. 9, no. 4-P1, pp. 232–235, 1979.

[10] J Wiki, "NuVoc — J Wiki," https://code.jsoftware.com/mediawiki/index.php?title=NuVoc&oldid=41417, 2023, [Online; accessed 6-April-2023].