Novel Algebras for Advanced Analytics in Julia

Jeremy Kepner*, Viral B. Shah†, Jeff Bezanson‡, Stefan Karpinski§ and Alan Edelman¶
*Email: kepner@ll.mit.edu
†Email: viral@mayin.org
‡Email: jeff.bezanson@gmail.com
§Email: stefan@karpinski.org
¶Email: edelman@math.mit.edu

***Abstract*—A linear algebraic based approach to graph algorithms that exploits the sparse adjacency matrix representation of graphs can provide a variety of benefits. These benefits include syntactic simplicity, easier implementation, and higher performance. One way to employ linear algebra techniques for graph algorithms is to use a broader definition of matrix and vector multiplication. We demonstrate through the use of the Julia language system how easy it is to explore semirings using linear algebraic methodologies.**

I. Introduction

A. Semiring algebra

The duality between the canonical representation of graphs as abstract collections of vertices and edges and a sparse adjacency matrix representation has been a part of graph theory since its inception [5], [6]. Matrix algebra has been recognized as a useful tool in graph theory for nearly as long (see [3] and references therein). A linear algebraic based approach to graph algorithms that exploits the sparse adjacency matrix representation of graphs can provide a variety of benefits. These benefits include syntactic simplicity, easier implementation, and higher performance. One way to employ linear algebra techniques for graph algorithms is to use a broader definition of matrix and vector multiplication. One such broader definition is that of a semiring. For example, in semiring notation we write matrix-matrix multiply as:

$$C = A + . * B$$

where

$$+.*$$

denotes standard matrix multiply. In such notation, a semiring requires that addition and multiplication are associative, addition is commutative, and multiplication distributes over addition from both left and right. In graph algorithms, a fundamental operation is matrix-matrix multiply where both matrices are sparse. This operation represents multi source 1-hop breadth first search (BFS) and combine, which is the foundation of many graph algorithms [2]. In addition, it is often the case that operations other than standard matrix multiply are desired, for example:

1)
$$C = A \ \max . + B$$

2)
$$C = A \ \min . \max B$$

3)
$$C = A \ |.\& B \ (\mathrm{or}.and)$$

4)
$$C = Af().g()B$$

With this more general case of sparse matrix multiply, a wide range of graphs algorithms can be implemented [4].

II. Application example

A classic example of the utility of the semiring approach is in finding the minimum path between all vertices in graph (see [12] in [4]. Given a weighted adjacency matrix for a graph where $A(i,j) = w_{ij}$ is the weight of a directed edge from vertex i to vertex j. Let $C(i,j)_2$ be the minimum 2-hop cost from vertex i to vertex j (if such a path exists). C_2 can be computed via the semiring matrix product:

$$C_2 = A \ \min . + A$$

Likewise, C_3 can be computed via

$$C_3 = A \ \min . + A \min . + A$$

and more generally

$$C_k = A^k$$

III. Julia

Julia is emerging as a very popular computing technology for high performance technical computing. It has been designed from the ground up to take advantage of modern techniques for executing dynamic languages efficiently, and still be of appeal to scientists, engineers, data analysts, and other practicitioners of technical computing[1].

Despite advances in compiler technology and execution for high-performance computing, programmers continue to prefer high-level dynamic languages for algorithm development and data analysis in applied math, engineering, the sciences, and general data analysis. High-level environments such as Matlab [8], Octave [10], R [11], SciPy [9], and SciLab [13] provide greatly increased convenience and productivity. However, C and Fortran remain the gold standard languages for computationally-intensive problems because high-level dynamic languages still lack sufficient performance. As a result, the most challenging areas of technical computing

have benefited the least from the increased abstraction and productivity offered by higher level languages.

Two-tiered architectures have emerged as the standard compromise between convenience and performance: programmers express high-level logic in a dynamic language while the heavy lifting is done in C and Fortran. The aforementioned dynamic technical computing environments are all themselves instances of this design. While this approach is effective for some applications, there are drawbacks. It would be preferable to write compute-intensive code in a more productive language as well. This is particularly true when developing parallel algorithms, where code complexity can increase dramatically. Instead, there is pressure to write "vectorized" code, which is unnatural for many problems and might generate large temporary objects which could be avoided with explicit loops. Programming in two languages is more complex than using either language by itself due to the need for mediation between different type domains and memory management schemes. Interfacing between layers may add significant overhead and makes whole-program optimization difficult. Two-tiered systems also present a social barrier, preventing most users from understanding or contributing to their internals.

Julia is designed from the ground up to take advantage of modern techniques for executing dynamic languages efficiently. As a result, Julia has the performance of a statically compiled language while providing interactive dynamic behavior and productivity like Python, LISP or Ruby. The key ingredients of performance are:

- Rich type information, provided naturally by multiple dispatch;
- Aggressive code specialization against run-time types;
- JIT compilation using the LLVM compiler framework [7].

Although a sophisticated type system is made available to the programmer, it remains unobtrusive in the sense that one is never required to specify types. Type information flows naturally from having actual values (and hence types) at the time of code generation, and from the language's core paradigm: by expressing the behavior of functions using multiple dispatch, the programmer unwittingly provides the compiler with extensive type information.

We recommend trying the code examples in this paper. Just a few moments may convince the reader that Julia is simple yet powerful. All of the code may be found on GITHUB: `github.com/ViralBShah/SemiringAlgebra.jl`

IV. Semiring algebra in Julia

Julia's ordinary matrix multiplication is recognizable to users of many high level languages:

```
A=rand(m,n);
B=rand(n,p);
C=A*B
```

or one can equally well use the prefix notation

```
C = *(A,B)
```

instead of the infix notation.

We believe that users of matrix-multiply who are use to such compact expressions would prefer not to have overly complicated syntax to express the more general semirings. Julia offers two simple possibilities that are readily available for exploration:

1) Star overloading: This method is recommended for interactive exploration of many semiring operators. Without introducing any types, define `*(f,g)(A,B)` to perform the semiring operation very generally, with no a-priori restriction on the binary functions `f` or `g`. Upon overloading the star operator, and defining the ringmatmul, the following immediately work in Julia:

```
*(max,+)(A,B)
*(min,max) (A,B)
*(|,&) (A,B)
*(+,*) (A,B)
```

The last example returns the same value as `*(A,B)` or `A*B`.

We note users can also equally well now type the infix notation, `(max * +)(A,B)` though users should avoid potentially ambiguous situations such as `(+ * *)`.

2) Creation of semiring objects: This method is recommended for users who are working exclusively in one semiring and wish to optimize notation and performance. In this method, a semiring type is created. One overloads scalar + and scalar * only, and thanks to the care taken in Julia's abstractions, automatically `*(A,B)` or `A*B` provides the semiring matrix multiply without any notational hindrance.

A. Star Overloading

First time users should readily and effortlessly be able to google, download, and install Julia. The star overloading functionality may be explored by typing directly into a Julia session:

```
function ringmatmul(pl,ti,A,B)
m=size(A,1); n=size(B,2); p=size(A,2)
C=[ti(A[i,1],B[1,j]) for i=1:m,j=1:n]
for i=1:m, j=1:n, k=2:p
   C[i,j]=pl(C[i,j],ti(A[i,k],B[k,j]))
end
end

function *(f::Function,g::Function)
   (A,B)->ringmatmul(f,g,A,B)
end
```

We believe first time Julia users can readily understand the function `ringmatmul`, which is mainly the usual matrix multiply loop. The function takes four arguments, the first two, `pl` and `ti`, are functions that will play the role of plus and times. Functions are first class variables in Julia and thus may be arguments to other functions. The second two variables are the arrays `A` and `B`. The function `ringmatmul` is redefining matrix multiply in terms of `pl` and `ti`.

The second function overloads the symbol "*", for the convenience of the user using the multiple dispatch mechanism. It says that if "*" is called with two arguments, both of which have the type `Function`, then the result is a function itself which takes two arrays as arguments, and proceeds to compute the semiring `ringmatmul` with arguments `f,g,A,B`.

B. Semiring Objects

Here we create a "multiply-plus" semiring. While this seems hardwired, one can at time change the definitions of "+" and "*" which below are set to max and plus. The semiring does not need to be taught matrix multiply, and will work on matrices of any type (e.g. ints, floats, ...) and will also work on dense or sparse matrices.

```
immutable MPNumber{T} <: Number
    val::T
end

+(a::MPNumber,b::MPNumber)
      = MPNumber(max(a.val,b.val))
*(a::MPNumber,b::MPNumber)
      = MPNumber(a.val+b.val)
show(io::IO, k::MPNumber)
      = print(io, k.val)

zero{T}(::Type{MPNumber{T}})
      = MPNumber(typemin(T))
one{T}(::Type{MPNumber{T}})
      = MPNumber(zero(T))
promote_rule(::Type{MPNumber},::Type{Number})
      = MPNumber

mparray(A::Array) = map(MPNumber, A)
array{T}(A::Array{MPNumber{T}})
      = map(x->x.val, A)
mpsparse(S::SparseMatrixCSC)
      = SparseMatrixCSC(S.m, S.n, S.colptr,
            S.rowval,mparray(S.nzval))
```

The first function `MPNumber` is the constructor for a Max Plus Number. Then plus and times are defined as max and plus respectively, followed by a routine IO function.

The zero and one in this semiring are defined as the identity elements for max and plus, and the promote rule is a Julia construction which allows semi-ring numbers and ordinary numbers to work together.

Finally the last three functions allow for the conversion between ordinary arrays and the semi-ring, with a special extra constructor for the important use case of sparse arrays.

Some examples of using the code:

```
A=mparray(rand(3,3)); B=mparray(rand(3,3));
A*B
A*A
A^2
C=mpsparse(sparse(eye(3,3)));
C*C
```

V. Performance

The ultimate performance for an ordinary matrix multiply on floating point numbers would be the expected outcome of a highly tuned BLAS routine perhaps for a targeted architecture. The goal of Julia is to have the right combination of reasonable performance for the machine and productivity for the human. We have already demonstrated how expressive Julia is.

On a Macbook Pro, with dual-core 2.4 GHz Intel core i5, 8GB RAM, we compared a dense BLAS run of matmul with an MPNumberFloat64 run in the semiring. The BLAS time was 0.26msec (practically free!) for a 100x100 array. The semiring time was 313 msec. This is no surprise.

More interesting and more useful is the comparison for sparse matrices. For an n=100,000 by 100,000 sparse random array with density 1/n, the ordinary matrix multiply took 21 msec, and the semi-ring matrix multiply took a comperable 22 msec.

For a higher density (5/n) we found 461 msec for the matmul compared to 414 msec in the semi-ring.

VI. Conclusion

Julia facilitates the implementation and exploration of graph algorithms through the semiring notation of generalized matrix mutliply. Thhese algorithms appear in applications to data analysis and related fields. Exploitation of sparse data strucutres provides simple easy implemeentations with reasonable performance. Sparse semiring performance is likely to be comparable to ordinary sparse matmul in the Julia language.

References

[1] Jeff Bezanson, Stefan Karpinski, Viral Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. www-math.mit.edu/ edelman/homepage/papers/juliafast.pdf, 2012.

[2] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. A unified framework for numerical and combinatorial computing. *Computing in Sciences and Engineering*, 10(2):20–25, Mar/Apr 2008.

[3] Frank Harary. *Graph Theory*. Addison-Wesley Publishing, 1969.

[4] Jeremy V. Kepner and J. R. Gilbert. *Graph algorithms in the language of linear algebra*. Society for Industrial and Applied Mathematics, 2011.

[5] Dénes Kőnig. Gráfok és mátrixok. *Matematikaiés Fizikai Lapok*, 38:116–119, 1931.

[6] Dénes Kőnig. *Theorie der Endlichen und Unendlichen Graphen: Kombinatorische Topologie der Streckenkomplexe*. Akad. Verlag., 1936.

[7] LLVM. http://llvm.org.

[8] MATLAB. http://www.mathworks.com.

[9] Numpy. http://www.numpy.org.

[10] Octave. http://www.octave.org.

[11] R. http://www.r-project.org.

[12] Charles M. Rader. Connected components and minimum paths. In Jeremy V. Kepner and J. R. Gilbert, editors, *Graph algorithms in the language of linear algebra*. Society for Industrial and Applied Mathematics, 2011.

[13] Scilab. http://www.scilab.org.