SpMV Sections

ACM Reference Format:
. 2019. SpMV Sections. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SPARSE MATRIX-VECTOR MULTIPLICATION

Sparse linear algebra libraries and the methods they employ are vitally important in the domain of scientific computing (TODO: why?). Of particular interest is the sparse matrix-vector product, which solves $y = \boldsymbol{A}x$ where $\boldsymbol{A}$ is a sparse matrix and the vectors y and x are dense. Because SpMV usage is often highly repetitive within, e.g., iterative solvers, performance improvements via novel algorithms and the exploitation of modern hardware are areas of ongoing research (TODO: citations).

...why is SpMV difficult? Things like array indirection, varying sparsity patterns, it's memory bound...

As modern hardware introduces higher levels of parallelism and SpMV algorithms become more complex, the likelihood of introducing subtle bugs becomes decidedly higher. In this section we present a method for reasoning about the structural complexities and verifying the correctness of SpMV algorithms. We first model an abstract matrix-vector multiplication and subsequently demonstrate that a CSR SpMV algorithm is a valid functional refinement.

1.1 Abstract Matrix-Vector Multiplication

In order to demonstrate that an SpMV algorithm is correct, we must first create a model of matrix-vector multiplication to act as the arbiter of correctness. The result of the matrix-vector multiplication $y = \boldsymbol{A}x$ is a densely populated vector, y, in which each value is the dot product of a single row of the matrix A with the vector x. A dot product is simply a sum of products in which each product is of two values located at the same index within two vectors, as shown in Figure 1b.

Thus far, our matrix and sparse matrix models are capable of representing the structure of the matrix and vectors, as well as the values contained in $\boldsymbol{A}$ and x. However, because we wish to model the *structural* behavior of the algorithm, the resulting vector, y, cannot simply contain abstract values. We require a mechanism that allows us to determine the origin of values in the dot product–such a mechanism will allow us to reason about whether or not the dot product is the correct composition of values. This is achieved using the `SumProd` signature, introduced in Section 1.1.1. Additionally, we require some mechanism for equating dot products, allowing us to determine if the solutions generated by differing algorithms are equivalent. This is achieved using the `valEqv` predicate introduced in Section 1.1.2.

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \times \begin{Bmatrix} x_0 \\ x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{A_0 \cdot x_0} \\ A_1 \cdot x_1 \\ A_2 \cdot x_2 \end{Bmatrix}$$

(a)

$$\boldsymbol{A_0 \cdot x_0} = A_{00} * x_0 + A_{01} * x_1 + A_{02} * x_2$$

(b)

0	A_{00}	x_0
1	A_{01}	x_1
2	A_{02}	x_2

(c)

Figure 1: (a) a matrix-vector multiplication, (b) the components of the first dot product in (a), and (c) the relational form of the same dot product.

1.1.1 Sum of Products Model. The `SumProd` signature found in Figure 2 represents the result of a dot product operation, a sum of products. Because each dot product evaluates to a single value that must be stored in one of our existing matrix models, the `SumProd` signature extends `Value`. Note that this evaluation is never actually realized, we merely wish to assemble values such that they represent a sum of products. To do so, `SumProd` contains a single field, `values`, that maps unique integers to `Value` pairs. We do not allow `SumProd`s to nest, as we only wish to model dot products of numerical values, not complex numerical expressions, and so `SumProd` is removed from the set of values that may be stored by using the expression `(Value-SumProd)`. Using the `values` field, the `SumProd` signature is able to represent a sum of products as a table in which each row contains a product of two values, and the summation is composed of all rows, as shown in Figure 1c. Additionally, the integer value associated with each row indicates the index from which the values in that row originate.

1.1.2 Dot Product Equivalence. Hi.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
sig SumProd extends Value {
    values: Int
        → lone (Value-SumProd)
        → (Value-SumProd)
} {
    all i: values.univ.univ | i ≥ 0
}
```

Figure 2: The `SumProd` signature, representing the result of a dot product.