Reasoning About Sparse Vectors for Loops Code Generation

Vadim Zaliva, Franz Franchetti, Carnegie Mellon University

Motivating example

Pointwise operator vectorization:

$$f\left(\begin{bmatrix}a_0\\a_1\\a_2\\a_3\end{bmatrix}\right)=\begin{bmatrix}f(a_0)\\0\\0\\0\end{bmatrix}+\begin{bmatrix}0\\f(a_1)\\0\\0\end{bmatrix}+\begin{bmatrix}0\\0\\f(a_2)\\0\end{bmatrix}+\begin{bmatrix}0\\0\\0\\f(a_3)\end{bmatrix}=\begin{bmatrix}f(a_0)\\f(a_1)\\0\\0\end{bmatrix}+\begin{bmatrix}0\\0\\f(a_2)\\f(a_3)\end{bmatrix}=\begin{bmatrix}f(a_0)\\f(a_1)\\f(a_2)\\f(a_3)\end{bmatrix}$$

Loop Parellelization:

$$P^n_{f_j} = \sum_{j=0} S_{(j)} \circ A_{f_j} \circ G_{(j)}$$

Sample non-vectorized function processing single FP element.

```
void f1(float *src, float *dst);
```

Pointwise application of 'f' to src, storing results in dst, one at a time. It requires 4 iterations and 4 function calls.

```
for(i=0;i<4;i++)
    f1(src+i,dst+i);
```

$$P^n_{f_j} = \sum_{j=0}^{1} S_{h_{j,0}} \circ \left(\bigtimes_{k=0}^{1} A_{f_k} \right) \circ G_{h_{j,2}}$$

Sample vectorized function processing 2 FP elements at a time.

```
void f2(float *src, float *dst);
```

Pointwise application of 'f' to src, storing results in dst, two at a time. It requires just 2 iterations and 2 function calls.

```
for(i=0;i<2;i++)
    f2(src+2*i,dst+2*i);
```

The Problem

- Dense vectors are decomposed into iterative sums of sparse vectors.
- Various decompositions (number or vectors and location of non-sparse values) could represent a variety of memory access patterns.
- This allows applying a variety of algebraic transformations to reshape a computation to optimize for vectorization, parallelization, sequential memory access.
- However in such iterative sum, the addition has a special semantics:
    - Mathematically, the sparse values could be treated as zeroes.
    - Operationally, combining sparse and non-sparse value could be seen as an assignment.
- The expressions produced are naturally mapped to SSA form only if certain constraints on structure of sparse vectors under iterative sums are maintained.
- Tracking and enforcing such constraints for correctness proofs is difficult, as they are not adequately enforced by mathematica abstraction used.
- In this work we present a working approach for structural constraints tracking and propagation used in Coq proofs of correctness.

Approach

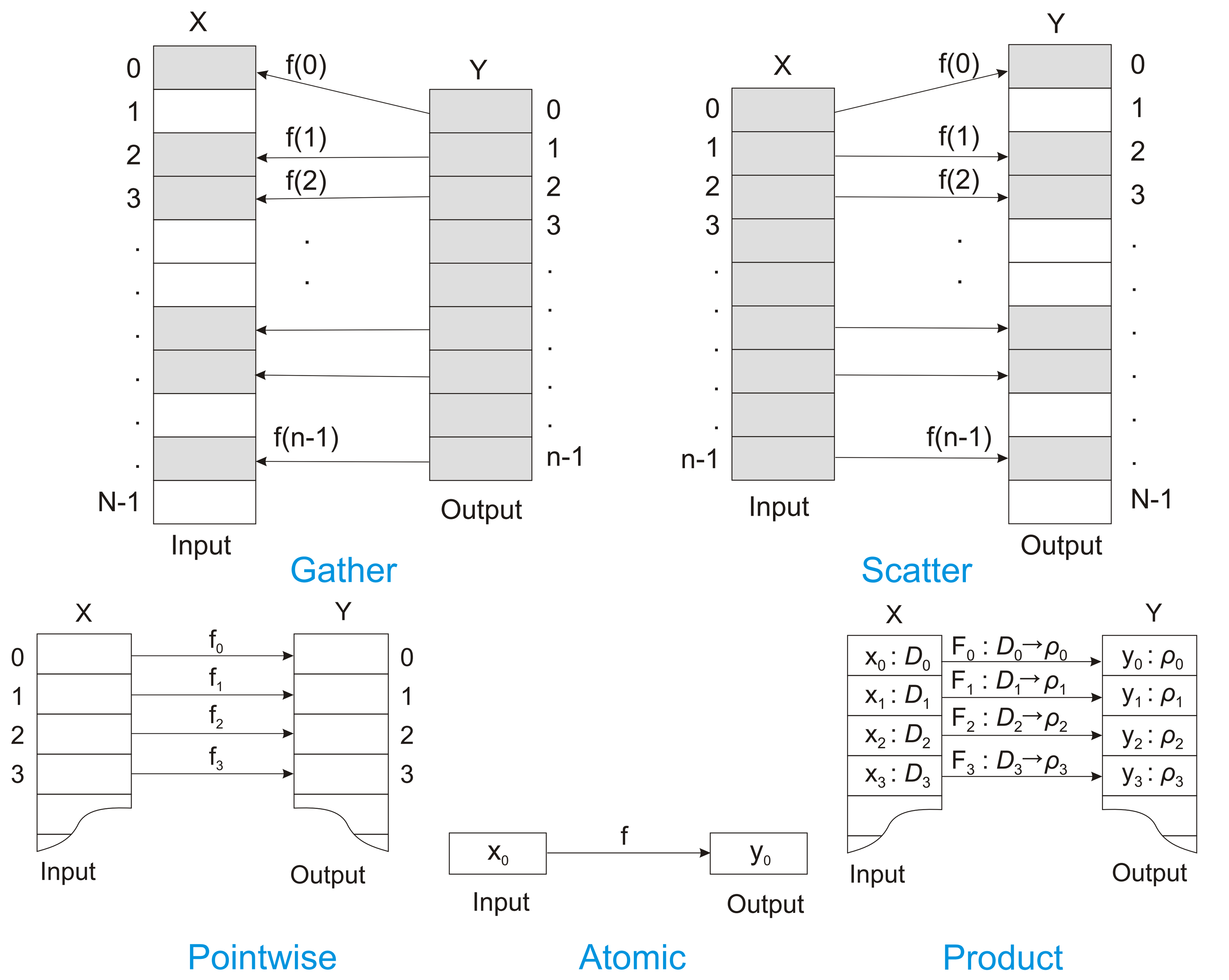
Sparsity Requirements

1. Distinguish empty and assigned cells.
2. Treat empty cells as some "default" value. Such default value could depend on the context (e.g. 0 for addition but 1 for multiplication).
3. In case of "a loop as sparse vector sum" we should never attempt to combine two non-sparse elements. This type of error we will call a "collision".
4. Sparsity tracking should be easy to perform during computations.
5. Collisions should be seamlessly tracked and propagated across the computation.
6. The collision and sparsity tracking should be proof-friendly (easy to deal with in Coq)
7. Separate sparsity tracking from actual operations on values as they represent two different aspects of computation.

State and Collision Tracking Monad

```
Record Rflags : Type := mkRFlags {is_struct:  bool ; is_collision: bool  }.
Definition rFlagsZero := mkRFlags true false.
Definition mappend (a b: Rflags): Rflags :=
  mkRFlags
    (is_struct a && is_struct b)
    (is_collision a || (is_collision b ||
              (negb (is_struct a || is_struct b)))).
Definition RMonoid : Monoid Rflags := Build_Monoid mappend rFlagsZero.
Definition R_θ := writer Rmonoid ℝ.
```

Operators



Implementation

"Diamond" Abstraction

Scalar $\quad \diamond : \mathcal{A} \to \mathcal{A} \to \mathcal{A}$

Vector $\quad \vec{\diamond} : \mathcal{A}^n \to \mathcal{A}^n \to \mathcal{A}^n$

$$((a_0, a_1, \dots, a_{n-1}), (b_0, b_1, \dots, b_{n-1})) \mapsto (a_0 \diamond b_0, a_1 \diamond b_1, \dots, a_{n-1} \diamond b_{n-1})$$

Operation $\quad \mathring{\diamond} : (\mathcal{A}^n \to \mathcal{A}^m) \to (\mathcal{A}^n \to \mathcal{A}^m) \to (\mathcal{A}^n \to \mathcal{A}^m)$

$$(F, G) \mapsto (\mathbf{x} \mapsto F(\mathbf{x})\ \vec{\diamond}\ G(\mathbf{x}))$$

Iterative $\quad \displaystyle\diamond_{i=0}^{n-1} F_i : \mathcal{A}^n \to \mathcal{A}^n$

$$\mathbf{x} \mapsto \begin{cases} \mathbf{0}^n & \text{if } n = 0, \\ \left(F_{n-1}\ \mathring{\diamond} \left(\diamond_{j=0}^{n-2} F_j\right)\right)(\mathbf{x}) & \text{otherwise} \end{cases}$$

Iterative Sum with Sparsity and Collision Tracking

Let us apply the diamond abstraction demonstrated to R_θ type (which represents $\mathbb{R}$ with R_{flags} state) and summation operator. To do so we specalize previous notation as follows:

```
Definition 𝒜 := ℛθ.
Definition ◇ := liftM2 (+).
Definition ◇⃗ := vector.map2 ◇.
Definition ◇̊ f g := λ x ⇒ (f x) ◇⃗ (g x).
Definition 0ⁿ := vector.const (ret 0) n.
```

This gives us a sparse, collision-tracking PointWise:

$$\text{Pointwise}_{n,f} = \diamond_{j=0}^{n-1} \left(S_{(j)_n} \circ A_{f_j} \circ G_{(j)_n} \right)$$

Summary

1. Implementation in Coq proof assistant.
2. Each value is tagged with two boolean flags: is_struct and is_collision.
3. Flags structure along with combining operation forms a Monoid.
4. Two Monoid instances are used: with and without collision tracking.
5. Flags are tracked using Writer Monad.
6. Operations on values can not directly examine sparsity flags and thus can not depend on them.
7. Sparsity is automatically tracked by the monad. No implicit flags handling in operators implementation.
8. Collision is automatically tracked and propagated by the monad.

Contact info

Vadim Zaliva <vzaliva@cmu.edu>
SPIRAL: http://spiral.net/

Carnegie Mellon