**Motivation**

- Extracting more parallelism in sequential loops
  - Massive parallelism in modern CPUs
  - Many cores, wide SIMD units
  - Certain loops are inherently sequential
  - Loop-carried dependence
  - Not benefit from the parallelism in hardware
  - Many sequential loops can be modeled as recurrence equations

---

**Examples**

- **A loop evaluating polynomials**
  
  ```c
  y = 0;
  for (i=0; i<N; i++) { 
    y = c*y + A[i];
  }
  ```

  **Modeled as:**
  
  ```c
  y_{i+1} = f_{i+1}(y_i) = c*y_i + A[i+1]
  ```

---

**Related Works**

- **Template-based approaches** (Ammarguellat et al. PLDI’90, Callahan et al. LCPC’92)
  - Match the computation in the loop with template recurrences
  - Use first-order recurrence templates with fixed parallelization strategy
  - Only work for loops that can be sliced into first-order recurrences

- **Search-based approaches** (Fisher et al. PLDI’94, Raychev et al. SOSP’15, Farzan et al. PLDI’17)
  - Search for closed-form composition of loop body functions
  - More general and powerful than templated-based approaches
  - Relies on the efficiency of the search algorithms
  - Applicability is indefinite (not clear what recurrences can be solve in reasonable amount of time)

---

**Our Work**

- **Main idea**
  - Many recurrence functions can have a point-value representation: a few sample points can reconstruct the function
  - Instead of deriving closed-form composition explicitly, we run the loop with a few different initial values (sampling)
  - The composed function can be reconstructed by the results produced by the different initial values (reconstruction)

---

**Linear function ("polynomial evaluation" loop)**

- `for(i=0; i<N; i++) {`  
  - `y = c*y + A[i];`  
- `y_j = Linr_j(y_j-1) = Linr_j Linr_{j-1}(y_{j-1}) = Linr_j Linr_{j-1} Linr_{j-2}(y_{j-2})`  
- **Compile-time reasoning:** the value of `y` at the end of iteration `j` is a linear function of its value at the end of iteration `i`
- **Runtime:** two sample points are sufficient to reconstruct it

- `y_0 = Linr_{j-1} Linr_{j-2}(0) = B, y_1 = Linr_{j-1} Linr_{j-2}(1) = A + B`  
- `y_j = Linr_{j-1} Linr_{j-2}(y_{j-2}) = (y_{j-2} - y_0) * y_1 + y_0`  
- `y_{j-2}` and `y_{j-1}` are the values of `y` at the end of iteration `j` with initial value of `0` and `1` respectively.

---

**Rectified-linear (loops with conditionals)**

- `for (i=0; i<N; i++) {`  
  - `if (s + A[i] < 0) s = 0;`  
  - `else s += A[i];`  
- `s_{i+1} = max(s_{i} + A[i+1], 0)`  
- **The composition of rectified-linear functions is still a rectified-linear function or simple piecewise-linear function**

- **Compile-time reasoning:** `s_j = func(s_i)` is a rectified-linear function

---

**Evaluation**

<table>
<thead>
<tr>
<th>Function Types</th>
<th>Parsimony?</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>yes</td>
</tr>
<tr>
<td>rectified-linear</td>
<td>yes</td>
</tr>
<tr>
<td>multiple-precision</td>
<td>yes</td>
</tr>
<tr>
<td>single-precision</td>
<td>yes</td>
</tr>
<tr>
<td>linear</td>
<td>no</td>
</tr>
<tr>
<td>rectified-linear</td>
<td>no</td>
</tr>
<tr>
<td>multiple-precision</td>
<td>no</td>
</tr>
<tr>
<td>single-precision</td>
<td>no</td>
</tr>
</tbody>
</table>

---

**Comparison with Farzan et al. PLDI’17**

- **Speedup over Sequential**
  - The speedup increases as the number of threads increases.
  - Linear and rectified-linear functions show better speedup compared to non-linear functions.
  - Multiple-precision functions provide higher speedup compared to single-precision functions.

---

**Running Example**

- Execution of the "polynomial evaluation" loop with `A[6]={1, -5, 3, -7, -1, -10}` and `c=2`

```c
y0 = 2 + 2 * 0 + 1 = 3
y1 = 2 + 2 * 1 + 1 = 5
y2 = 2 + 2 * 1 + 3 = 7
y3 = 2 + 2 * 1 + 7 = 13
y4 = 2 + 2 * 1 + 13 = 27
y5 = 2 + 2 * 1 + 27 = 55
```

---

**Comparison with Farzan et al. PLDI’17**

- **Number of Threads**
  - The speedup increases as the number of threads increases.
  - Linear and rectified-linear functions show better speedup compared to non-linear functions.
  - Multiple-precision functions provide higher speedup compared to single-precision functions.