A Dataflow Programming Language and Its Compiler for Streaming Systems

Haitao Wei, Stephane Zuckerman, Xiaoming Li, and Guang Gao

University of Delaware
Why a new dataflow programming language?

DDDAS Applications

A

B

C

D

E

F
Why a new dataflow programming language?

Spontaneous Data flow

Sensor input

Distributed and localized execution

Exit
Why a new dataflow programming language?

Spontaneous Data flow

- Sensor input

Distributed and localized execution

- Sensor input

Thread

Existing Parallel Programming Model and System Support
Why a new dataflow programming language?

- Arbitrary dataflow graph with multiple input/output operators
  - Beyond static mapping
  - True asynchronous processing

- Primitive and compositional coding
  - Code reuse: using subgraph template
  - Easy to construct dataflow graph: using explicit variable/name passing to connect the computation node

- Efficiency under dynamic program configurations
  - Automatic software pipelining
  - Optimized resource allocation and mapping
**COStream Programming Language**

- **Multiple I/O Operators VS Single I/O Operator**

**Fork-join style** dataflow graph in Streamit with single input/output and split-join structure

**Arbitrary** Dataflow graph in COSTream with multiple Input/Output Operator and stream
COStream Programming Language

- **composite**
  - Connect operators to construct a dataflow subgraph

- Can be instantiated to reuse the code to explore the parallelism
COStream Programming Language

- Stream variable and dataflow graph construct

```java
composite Main(){
    graph
        stream<float x> S = A(){…}
    }

    stream<float x> T = B(){…}

    stream<float x> P = C(T,S){…}
}
```

Costream: Easy to construct and make it simple

StreamIt: Introduce useless split-join nodes which makes programming hard and brings the overhead
# Example: Moving Average

```c
composite Main() {
    Int N=10;
    graph
        stream<float x> S=Source() {
            int x;
            init{x=0;}  
            work{
                S[0].x=x;
                x++;
            }  
        }  //initialize an instance of composite MovAve  
    stream<float x> P=MovAver(S)(N);
    Sink(P) {
        work{
            print(P[0].x);
        }
    }
}
```

```c
composite MovAver(output O, input rawIn) {
    param
        int N;
    graph
        stream<float x> O=Aver(rawIn) {
            float w[N];
            init{
                for(i=0;i<N;i++)
                    w[i]=i;
            }
            work{
                int sum=0, i;
                for(i=0;i<N;i++)
                    sum += rawIn[i].x*w[i];
                O[0].i=sum/N;
            }
    }
    window{
        rawIn sliding(N,1)
        O tumbling(1);}
}
```
Example: Moving Average

composite Main(){
    Int N=10;
    graph
        stream<float x> S=Source()
        int x;
        init{x=0;}
        work{
            S[0].x=x;
            x++;
        }
    //initialize an instance of
    composite
    MovAve
        stream<float x>
    P=MovAver(S)(N);
    Sink(P){
        work{
            print(P[0].x);
        }
    }
}

composite MovAver(output O, input rawIn){
    param
        int N;
    graph
        stream<float x> O=Aver(rawIn){
            float w[N];
            init{
                for(i=0;i<N;i++)
                    w[i]=i;
                }
            work{
                int sum=0, i;
                for(i=0;i<N;i++)
                    sum += rawIn[i].x*w[i];
                O[0].i=sum/N;
            }
        window{
            rawIn sliding(N,1)
            O tumbling(1);
        }
    }
}
Example: Moving Average

```cpp
composite Main(){
  Int N=10;
  graph
    stream<float x> S=Source(){
      int x;
      init{x=0;}
      work{
        S[0].x=x;
        x++;
      }
    //initialize an instance of composite MovAve
    stream<float x> P=MovAver(S)(N);
    Sink(P){
      work{
        print(P[0].x);
      }
    }
}}

composite MovAver(output O, input rawIn){
  param
    int N;
  graph
    stream<float x> O=Aver(rawIn){
      float w[N];
      init{
        for(i=0;i<N;i++)
          w[i]=i;
      }
      work{
        int sum=0, i;
        for(i=0;i<N;i++)
          sum += rawIn[i].x*w[i];
        O[0].i=sum/N;
      }
    window{
      rawIn sliding(N,1)
      O tumbling(1);
    }
  }
Example: Moving Average

```cpp
composite Main(){
  Int N=10;
  graph
    stream<float x> S=Source(){
      int x;
      init{x=0;}
      work{
        S[0].x=x;
        x++;
      }
      //initialize an instance of
      composite
      MovAve
      stream<float x> P=MovAver(S)(N);
      Sink(P){
        work{
          print(P[0].x);
        }
      }
    }
}

composite MovAver(output O, input rawln){
  param
    int N;
  graph
    stream<float x> O=Aver(rawln){
      float w[N];
      init{
        for(i=0;i<N;i++)
          w[i]=i;
      }
      work{
        int sum=0, i;
        for(i=0;i<N;i++)
          sum +=
          rawln[i].x*w[i];
        O[0].i=sum/N;
      }
      window{
        rawln sliding(N,1)
        O tumbling(1);
      }
    }
```
Example: Moving Average

```plaintext
composite Main(){
    Int N=10;
    graph
        stream<float x> S=Source(){
            int x;
            init{x=0;}
            work{
                S[0].x=x;
                x++;
            }
        } //initialize an instance of composite
    MovAver
        stream<float x> P=MovAver(S)(N);
    Sink(P){
        work{
            print(P[0].x);
        }
    }
}

composite MovAver(output O, input rawln){
    param
        int N;
    graph
        stream<float x> O=Aver(rawln){
            float w[N];
            init{
                for(i=0;i<N;i++)
                    w[i]=i;
            }
            work{
                int sum=0, i;
                for(i=0;i<N;i++)
                    sum += rawln[i].x*w[i];
                O[0].i=sum/N;
            }
        }
    window{
        rawln sliding(N,1)
        O tumbling(1);
    }
}
```
Compiler Framework

1. Front-end: Lexer, parser and construct AST
2. MIR: Construct stream graph from AST
3. Symbol Execute: Initial && Periodic schedule of SG
4. Optimization: • Parallel exploring • Buffer allocation • Synchronization
5. Code Generation
6. Runtime System
7. Low-level c/ c++ compiler
8. Multi/Many-core

Synchronization Data Flow
Software Pipelining

Step 1: Task Assignment
Software Pipelining

Step 2: Stage Assignment
Software Pipelining

Core1

A0

A0

A0

Core2

A1

A1

A1

Core3

B0

B1

B2

B0

B1

B2

...
Experimental Results

Speedup: 3.0x~7.4x, average: 5.9x

FMRadio: small workload and lots of memory access
Vocoder: bad work balance
Experimental Results

**Speedup:** 3.0x~7.4x, average: 5.9x

FMRadio: small workload and lots of memory access
Vocoder: bad work balance
Ongoing Research

- **Compilation**
  - Software Pipelining within codelets

- **Backend**
  - Currently using pthread
  - Will use Fresh Breeze runtime
Ongoing Research (cont’d)

- **Runtime**
  - Currently re-optimize for every new configuration
  - Graceful transition between configurations
Transition between configurations
Transition between configurations
Transition between configurations
Transition between configurations
Acknowledgements

- **AFOSR** FA9550-13-1-0213

- **NSF** CCF-0925863, CCF-0937907 and OCI-0904534