An embedded C++ domain-specific language for stream parallelism

**Authors**: Dalvan Griebler\(^1,2\), Marco Danelutto\(^2\), Massimo Torquati\(^2\), Luiz Gustavo Fernandes\(^1\)

**E-mail**: dalvan.griebler@acad.pucrs.br

\(^1\)Faculdade de Informática, Computer Science Graduate Program  
Parallel Application Modeling Group - GMAP  
PUCRS – Porto Alegre – Brazil  
\(^2\)Computer Science Department, University of Pisa, Italy

International Conference on Parallel Computing (ParCo) - 2015

September 2015
Introduction

Stream Processing [1]

- **Application domain**: data, image and network.
- **Pattern of behavior**: process input and produce output.
- **Requirements**: high throughput and low latency.

Our Domain (Stream Parallelism)

- **Problem**: sequential code rewriting and low-level optimization for developers not familiar with parallelism.
- **Goal**: solve the problem, flexibility for fast code prototyping, and efficient parallel code generation.
- **Solution**: standard C++ annotation mechanism by using GCC-Pluings technique for parallel code generation, and the FastFlow runtime.
Introduction

Motivation

- **DSL-POPP**: provides suitable building block for Master/Slave and Pipeline, where the DSL is composed by intrusive annotations using a dedicated compiler [2, 3].
- **REPARA**: intermediate attribute annotations which are not exposed to end users and inserted by appropriate tools [4].

Contributions

- A domain-specific language for expressing stateless stream parallelism.
- An annotation-based programing interface that preserves the semantics of DSL-POPP’s principles and adopts REPARA’s C++ attribute style, avoiding significant code rewriting in sequential programs.
Introducing C++ Attributes with GCC plugin technique

Standard C++ Attributes

- **Originated**: GNU C attributes (`__attribute__((<name>)))` to standard C++ language ([[attr-list]])) [5, 6].
- **Advantage**: can be declared almost everywhere in a program (e.g., types, classes, code blocks, etc.), and the compiler is able to fully recognize

GCC Plugins

- Why to use?
- Which are the drawbacks?
## Attributes for Stream Parallelism

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ToStream()</code></td>
<td>annotates a loop or block of code for implementing a stream parallelism region</td>
</tr>
<tr>
<td><code>Stage()</code></td>
<td>annotates a block of code that computes the stream inside a <code>ToStream</code> region</td>
</tr>
<tr>
<td><code>Input()</code></td>
<td>used to indicate input streams in the ID attributes</td>
</tr>
<tr>
<td><code>Output()</code></td>
<td>used to indicate output streams ID attributes</td>
</tr>
<tr>
<td><code>Replicate()</code></td>
<td>auxiliary attribute to indicate a stage replication</td>
</tr>
</tbody>
</table>

**Table:** The generalized C++ attributes for the DSL.
How to use

1. Where does the stream start?
2. Where does the stream come from?
3. Where are the stages inside the computation?
4. Which are the input streams for my stages?
5. Can I replicate a stage?

Legend:
- Think
- Solve
- Order

Example code snippet:

```java
[[ToStream(...)]]
loop() {
    ... [[Stage(...), Replicate(...)]{ ...
    }
}
```
Design Implementation

- High-level abstractions
- Algorithmic skeleton flexibility [7]
- Interacting mode:
  - **Computation activities**: identify when using the ToStream and Stage ID attributes
  - **Spatial constrains**: identify by using input and output attributes
  - **Temporal constrains**: defined by the order of the declarations and their spatial constrains
  - **Interaction**: based on the users’ stream dependency specification and using lock-free queues of FastFlow
- Persistent nesting when adding Replicate attribute (replicates “R” only sequential code “S” in stage attribute)
Transformation Rules

**DSL attributes**

- `[[ToStream]]{ [[Stage]]{} }` → `ToStream(S,S)` → `Pipe(S,S)`
- `[[ToStream]]{ [[Stage,Replicate]]{} }` → `ToStream(S,R(S))` → `Farm(Emitter(S),Worker(S))`
- `[[ToStream]]{ [[Stage]]{} [[Stage]]{} }` → `ToStream(S,S,S)` → `Pipe(S,S,S)`
- `[[ToStream]]{ [[Stage,Replicate]]{} [[Stage]]{} }` → `ToStream(S,R(S),S)` → `Farm(Emitter(S),Worker(S),Collector(S))`
- `[[ToStream]]{ [[Stage,Replicate]]{} [[Stage,Replicate]]{} }` → `ToStream(S,R(S),R(S))` → `Pipe(Farm(Emitter(S),Worker(S)),Farm(Worker(S)))`
Methodology

Experiments

- Sobel Application.
- Prime Number Application.

Evaluation

- When possible, equivalent code with OpenMP pragmas is tested
- Best execution times are reported for each test
- The average value is obtained over 40 runs

Environment

Dual-socket NUMA Intel multi-core Xeon E5-2695 Ivy Bridge micro-architecture running at 2.40GHz featuring 24 cores 2-way Hyperthreading. Each core had 32KB L1 and 256KB L2 private, and 30MB L3 shared. Linux 2.6.32 x86_64 (CentOS 6.5) and GNU GCC 4.9.2 with the –O3 flag.
Sobel Application \((S \rightarrow R(S) \rightarrow S)\)

```cpp
1 using namespace spar;
2 // global declaration
3 int main(int argc, char *argv[])
4 {
5     // open directory ...
6     DIR *dptr = opendir(...);
7     struct dirent *dfptr;
8     [[ToStream(Input(dfptr, dptr, argv), Output(tot_not, tot_img))]
9         while((dfptr = readdir(dptr)) != NULL) {
10         // preprocessing
11         if (file_extension == "bmp") {
12             // Reads the image ...
13             tot_img++;
14             image = read(filename, height, width);
15             [[Stage(Input(image, height, width), Output(new_image)), Replicate(workers)]]
16                 // Applies the Sobel
17                 new_image = sobel(image, height, width);
18             }
19         }[[Stage(Input(new_image, height, width))]]
20             // Writes the image ...
21             write(new_image, height, width);
22         } // end stage
23     } else {
24         tot_not++;
25     }
26 } // end of stream computing
27 // end processing
28 return 0;
29 ```
Sobel Application ($S \rightarrow R(S)$)

```cpp
using namespace spar;

// global declaration
int main(int argc, char *argv[]){
    // open directory ...
    DIR *dptr = opendir(...);
    struct dirent *dfptr;
    [[ToStream(Input(dfptr, dptr, argv), Output(tot_not, tot_img))]] while((dfptr = readdir(dptr)) != NULL){
        // preprocessing
        if (file_extension == "bmp"){
            // Reads the image ...
            tot_img++;
            image = read(filename, height, width);
            [[Stage(Input(image, height, width)), Replicate(workers)]]{
                // Applies the Sobel
                new_image = sobel(image, height, width);
                // Writes the image ...
                write(new_image, height, width);
            } // end stage
        } else{
            tot_not++;
        }
    } // end of stream computing
    // end processing
    return 0;
}
```
Sobel Application (Parallel Activity Graph)
Sobel Application (Performance)

Results (Size=800x600 -- N=400)

<table>
<thead>
<tr>
<th>Tested versions</th>
<th>DSL</th>
<th>OMP-4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>21.7</td>
<td>21.7</td>
</tr>
<tr>
<td>S-&gt;R(S)-&gt;S</td>
<td>5.60</td>
<td>4.71</td>
</tr>
<tr>
<td>S-&gt;R(S)</td>
<td>4.85</td>
<td>3.90</td>
</tr>
</tbody>
</table>

Results (Size=mixed -- N=400)

<table>
<thead>
<tr>
<th>Tested versions</th>
<th>DSL</th>
<th>OMP-4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>108.9</td>
<td>108.9</td>
</tr>
<tr>
<td>S-&gt;R(S)-&gt;S</td>
<td>22.79</td>
<td>21.12</td>
</tr>
<tr>
<td>S-&gt;R(S)</td>
<td>25.88</td>
<td>20.32</td>
</tr>
<tr>
<td>S-&gt;R(S)-&gt;R(S)</td>
<td>24.39</td>
<td></td>
</tr>
</tbody>
</table>
Prime Number Application \((S \leftrightarrow R(S))\)

```cpp
1 using namespace spar;
2 // global declarations ...
3 int prime_number(int n){
4    int total = 0;
5    [[ToStream(Input(total,n),Output(total))]]
6    for (int i = 2; i <= n; i++){
7        int prime = 1;
8        [[Stage(Input(i,prime),Output(prime)),Replicate(workers)]]
9        for (int j = 2; j < i; j++){
10            if (i % j == 0){
11                prime = 0;
12                break;
13            }
14        }
15        total = total + prime;
16    }
17    return total;
18 }
```
Prime Number Application \((\mathbf{S} \rightarrow R(\mathbf{S}) \rightarrow \mathbf{S})\)

```cpp
1 using namespace spar;
2 // global declarations ...
3 int prime_number(int n){
4    int total = 0;
5    [[ToStream(Input(total,n),Output(total))]]
6    for (int i = 2; i <= n; i++){
7        int prime = 1;
8        [[Stage(Input(i,prime),Output(prime),Replicate(workers))]]
9        for (int j = 2; j < i; j++){
10           if ( i % j == 0 ){
11              prime = 0;
12              break;
13           }
14        }
15        [[Stage(Input(total,prime),Output(total))]] { total = total + prime; }
16    }
17    return total;
18 }
```
Prime Number Application (Parallel Activity Graph)
Prime Number Application (Performance)

Results (N=500000)

<table>
<thead>
<tr>
<th>Tested versions</th>
<th>DSL</th>
<th>OMP-4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[S]</td>
<td>70.4</td>
<td>70.4</td>
</tr>
<tr>
<td>[S&lt;-&gt;R(S)]</td>
<td>6.17</td>
<td>7.91</td>
</tr>
<tr>
<td>[S-&gt;R(S)-&gt;S]</td>
<td>4.84</td>
<td></td>
</tr>
</tbody>
</table>

Execution Time (S)
Related Work

Standard runtimes
- OpenMP [8]
- Cilk [9]
- TBB [10]

Research runtimes
- Programming Language: StreamIt [11]
- Skeleton library: FastFlow [12]
- Standard extensions: Cilk-Piper [13] and OpenStream [14]
Conclusions

Overview of Results

- A new C++ embedded DSL for expressing simple stream-based parallelism
- Standard annotation-based interface is flexible enough by providing five attributes
- Efficient parallel code transformations using FastFlow
- The performance results are comparable or better than with OpenMP
- Code productivity compared to FastFlow code: 23.4% (sobel) and 27.5% (prime number)

Future Works

- Implement automatic source-to-source transformation
- Perform more experiments using our DSL
References

Henrique C. M. Andrade, Bugra Gedik, and Deepak S. Turaga.
*Fundamentals of Stream Processing.*

Dalvan Griebler and Luiz G. Fernandes.
Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming.
In *Programming Languages - 17th Brazilian Symposium - SBLP*, volume 8129 of *LNCS*, pages 105–119, Brasilia, Brazil, October 2013. Springer.

Dalvan Griebler, Daniel Adornes, and Luiz G. Fernandes.
Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures.

REPARA Project.
D6.2: Dynamic Runtimes for Heterogeneous Platforms.
Technical report, University of Pisa, Pisa, Italy, November 2014.

Jens Maurer and Michael Wong.
Towards Support for Attributes in C++ (Revision 6).

ISO/IEC.
Information Technology - Programming Languages - C++.

Anne Benoit and Murray Cole.
Two Fundamental Concepts in Skeletal Parallel Programming.
In *International Conference on Computational Science (ICCS)*, volume 3515 of *LNCS*, pages 764–771, USA,
Questions