A Domain-Specific Language and Compiler for Computation-in-Memory Skeletons

Computation in Memory Group
Computer Engineering Lab
Delft University of Technology

Jintao Yu
Tom Hogervorst
Razvan Nane

Date: May 10th 2017
Motivation

• Conventional computer challenges
  • Memory wall
  • Power wall

Source: Borkar, Shekhar. "Exascale computing – a fact or a fiction.” Keynote presentation at IPDPS (2013)
Motivation

- Conventional computer challenges
  - Memory wall
  - Power wall

- Promising solution: memristor-based Computation-in-Memory (CiM)*
  - Accelerator for highly parallel functions
  - Memristor crossbar and controller

Motivation

- Conventional computer challenges
  - Memory wall
  - Power wall

- Promising solution: memristor-based Computation-in-Memory (CiM)*
  - Accelerator for highly parallel functions
  - Memristor crossbar and controller
  - Logic and memory operations
  - No off-chip communication
  - Low energy consumption

---

Motivation

- Conventional computer challenges
  - Memory wall
  - Power wall

- Promising solution: memristor-based Computation-in-Memory (CiM)*
  - Accelerator for highly parallel functions
  - Memristor crossbar and controller
  - Logic and memory operations
  - No off-chip communication
  - Low energy consumption

CiM can solve memory and power wall!

---

Motivation

• CiM challenges
  • Memristors are passive devices
  • Data movement requires explicit control
  • Routing influences scheduling *
  • Spatial programming

*: L. Xie et al. "Interconnect networks for memristor crossbar.", NANOARCH 2015
Motivation

- **CiM challenges**
  - Memristors are passive devices
  - Data movement requires explicit control
  - Routing influences scheduling *
  - Spatial programming

- **Solution: CiM skeletons #**
  - High-level constructs
  - Scheduling, placement, and routing

Motivation

- **CiM challenges**
  - Memristors are passive devices
  - Data movement requires **explicit** control
  - Routing influences scheduling *
  - Spatial programming

- **Solution: CiM skeletons #**
  - High-level patterns
  - Scheduling, placement, and routing
  - Implemented as C++ functions
    - Not flexible
    - High development time

---

C codes segment:

```c
... recur_ske inner("inner", &mul, &add);
   inner.setSize(16);
   inner.genSystem();
...```

---

Motivation

- **CiM challenges**
  - Memristors are passive devices
  - Data movement requires **explicit** control
  - Routing influences scheduling *
  - Spatial programming

- **Solution: CiM skeletons #**
  - High-level patterns
  - Scheduling, placement, and routing
  - Implemented as C++ functions
    - Not flexible

We need a way to describe CiM skeletons!

---

Contributions

- **DSL** for spatial patterns on CiM crossbar

Program ::= Circuit.decl* Component*
Component ::= ID Sig.decl [Par.decl] Statement*
Sig.decl ::= {ID Int}* {ID Int}*
Par.decl ::= {Type ID}*
  Type ::= “int” | “comp”
Contributions

- **DSL** for spatial patterns on CiM crossbar

- **Compiler** that schedules, places, and routes

Program ::= Circuit_decl* Component*
Component ::= ID Sig_decl [Par_decl] Statement*
Sig_decl ::= (ID Int)* (ID Int)*
Par_decl ::= (Type ID)*
  Type := “int” | “comp”
Contributions

• **DSL** for spatial patterns on CiM crossbar

• **Compiler** that schedules, places, and routes

• **Verification** of language and compiler
Outline

- The need for a new language
- CiM toolchain
- CiM DSL syntax
- CiM compiler implementation
- Results
- Conclusion
The Need for a New Language

• Requirements:
  • Explicit spatial information
  • Skeleton-based
The Need for a New Language

• Requirements:
  • Explicit **spatial** information
  • **Skeleton**-based

• Existing spatial programming languages
  • MaxJ\* -- Extended on Java
  • ANML\# -- Extended on XML

The Need for a New Language

• Requirements:
  • Explicit spatial information
  • Skeleton-based

• Existing spatial programming languages
  • MaxJ* -- Extended on Java
  • ANML# -- Extended on XML

• Existing skeleton-based programming languages
  • GridFOR%
  • Delite$

Spatial languages: MaxJ, ANML
Skeleton languages: GridFOR, Delite
The Need for a New Language

• Requirements:
  • Explicit spatial information
  • Skeleton-based

• Existing spatial programming languages
  • MaxJ* -- Extended on Java
  • ANML# -- Extended on XML

• Existing skeleton-based programming languages
  • GridFOR%
  • Delite$

None has both features

%: Y. Wang & Z. Li. “GridFOR: A Domain Specific Language for Parallel Grid-Based Applications.” IJPP, 2016
CiM Toolchain

- GPL Program: partitioned in three parts

```cpp
f = open_file();
b = read_file(f);
#pragma hls
while (i<100)
a[i] = b[i] * 2;
#pragma lib
inner_pro(a, b);
```
CiM Toolchain

- GPL Program: partitioned in three parts
- CiM DSL User: library developer
  - Knowledgeable about CiM
  - Familiar with the algorithms

```c
f = open_file();
b = read_file(f);

#pragma hls
while (i<100)
a[i] = b[i] * 2;

#pragma lib
Inner_pro(a, b);
```
CiM Toolchain

- GPL Program: partitioned in three parts
- CiM DSL User: library developer
  - Knowledgeable about CiM
  - Familiar with the algorithms
- CiM compiler
  - Function library
  - Primitive circuit library

```plaintext
f = open_file();
b = read_file(f);
#include "function.h"
#pragma hls
while (i<100)
a[i] = b[i] * 2;
#include "library.h"
#pragma lib
Inner_pro(a, b);
```
CiM DSL Syntax - Overview

- Data-flow language

- Language constructs
  - Program
  - Component
  - Statement
    - Signal
  - Expression
    - Functional block
    - Operator

Program
CiM DSL Syntax - Overview

- Data-flow language
- Language constructs
  - Program
  - Component
CiM DSL Syntax - Overview

• Data-flow language

• Language constructs
  • Program
  • Component
  • Statement
    • Signal
CiM DSL Syntax - Overview

• Data-flow language

• Language constructs
  • Program
  • Component
  • Statement
    • Signal
  • Expression
    • Functional block
    • Operator
CiM DSL Syntax - Expressions

- Functional blocks
  - Single operations
  - Referred to with one word
  - Represent circuits

```
Circuit_decl ::= ID File_name

Exp ::= Circuit | Int | ID Exp1 Exp2 ...
     | Exp1 Op Exp2 | Map | Fold | Repeat

Op ::= *_D_* | *_H_* | *_I_*
```

May 3, 2017
CiM DSL Syntax - Expressions

- Functional blocks
  - Single operations
  - Referred to with one word
  - Represent circuits
- Operators
  - Work on (arrays of) functional blocks
  - Implicitly define scheduling, placement, and routing

Circuit_dec ::= ID File_name

Exp ::= Circuit | Int | ID Exp1 Exp2 ...
    | Exp1 Op Exp2 | Map | Fold | Repeat

Op ::= *_D_* | *_H_* | *_I_*
CiM DSL Syntax - Expressions

- **Functional blocks**
  - Single operations
  - Referred to with one word
  - Represent circuits

- **Operators**
  - Work on (arrays of) functional blocks
  - Implicitly define scheduling, placement, and routing

---

**Circuit Decl**: `ID File_name`

**Exp**: `Circuit | Int | ID Exp1 Exp2 ...`

| Exp1 Op Exp2 | Map | Fold | Repeat |

**Op**: `*_D_* | *_H_* | *_I_*`

---

May 3, 2017
CiM DSL Syntax - Expressions

- Functional blocks
  - Single operations
  - Referred to with one word
  - Represent circuits

- Operators
  - Work on (arrays of) functional blocks
  - Implicitly define scheduling, placement, and routing

\[
\begin{align*}
\text{Circuit}_\text{decl} & := ID \text{ File_name} \\
\text{Exp} & := \text{Circuit} \mid \text{Int} \mid ID \text{ Exp}_1 \text{ Exp}_2 \ldots \\
& \quad \mid \text{Exp}_1 \text{ Op Exp}_2 \mid \text{Map} \mid \text{Fold} \mid \text{Repeat} \\
\text{Op} & := \*_D_* \mid \*_H_* \mid \*_I_*
\end{align*}
\]
CiM DSL Syntax - Expressions

- Expressions describe circuits
- Functions for larger circuits

Example: binary tree of adders

```plaintext
foldR<*>H*>(map<i=4:/2:0>(repeat[i](add)))
```

```
Repeat ::= Int Exp
Map ::= ID Range Exp
Fold ::= Op Exp
Range ::= Int | Int [Ar_Op Int] Int
Ar_Op ::= " + " | " - " | " * " | " / "
```
Expressions describe circuits
Functions for larger circuits
  • Repeat

Example: binary tree of adders

repeat[4](add)
• Expressions describe circuits
• Functions for larger circuits
  • Repeat
  • Map

Example: binary tree of adders

\[
\text{map}\langle i=4:/2:0\rangle(\text{repeat}[i](\text{add}))
\]

\[
\begin{align*}
\text{repeat}[4](\text{add}) & \quad \text{add} \quad \text{add} \quad \text{add} \quad \text{add} \\
\text{repeat}[2](\text{add}) & \quad \text{add} \\
\text{repeat}[1](\text{add}) & \quad \text{add}
\end{align*}
\]
Expressions describe circuits
- Functions for larger circuits
  - Repeat
  - Map
  - Fold
- Example: binary tree of adders

\[
\text{foldR} \langle \ast \_H\_\ast \rangle (\text{map}\langle i=4:/2:0\rangle (\text{repeat}[i](\text{add})))
\]
CiM DSL Syntax - Expressions

- Example: binary tree of adders
  - Placed in HTree pattern*

\[ \text{foldR}(<\ast \ H \ \ast>)(\text{map}<i=4:/2:0>(\text{repeat}[i](\text{add}))) \]

CiM DSL Syntax - Statements

- Statement functions:
  - Define variables
  - Provide conditionals and loops
  - Control data flow to/from circuits

\[
\text{Statement ::= Signal [Exp] Signal | Loop}
\]
\[
\text{Loop ::= ID Range Statement+}
\]
\[
\text{Signal ::= ID Range | Signal ++ Signal}
\]
\[
\text{Zip(Signal, Signal)}
\]
CiM DSL Syntax - Statements

- Statement functions:
  - Define variables
  - Provide conditionals and loops
  - Control data flow to/from circuits

\[
\text{Statement ::= Signal [Exp] Signal | Loop} \\
\text{Loop ::= ID Range Statement+} \\
\text{Signal ::= ID Range | Signal ++ Signal} \\
\text{ | Zip(Signal, Signal)}
\]

- Signals
  - Represent paths
  - Allow complex connection schemes
CiM DSL Syntax - Statements

- **Statement functions:**
  - Define variables
  - Provide conditionals and loops
  - Control data flow to/from circuits

- **Signals**
  - Represent paths
  - Allow complex connection schemes

```
Statement ::= Signal [Exp] Signal | Loop
Loop ::= ID Range Statement+
Signal ::= ID Range | Signal ++ Signal
    | Zip(Signal, Signal)
```

![Diagram of signal connections](image_url)
CiM DSL Syntax - Statements

- **Statement functions:**
  - Define variables
  - Provide conditionals and loops
  - Control data flow to/from circuits

- **Signals**
  - Represent paths
  - Allow complex connection schemes
CiM DSL Syntax - Components

• Act like functions
  • Represent library function or CiM skeleton

• Example program:

```plaintext
libmod add(add.lib);
libmod mul(mul.lib);

comp main<in[16] | out[1]> (){
  in[0:16] => inner_product(8) => out[0];
}

comp inner_product<a[n], b[n] | out[1]>(int n){
  zip(a[0:n], b[0:n]) => copy[n](mul) *H_* reduce(n/2, add) => out[0];
}

comp reduce<in[2*n] | out[1]>(int n, comp c){
  in[0:2*n] => foldR<*H*>((map<i = n: /2: 0>(repeat[i](c))) => out[0];
}
```

Program ::= Circuit_decl* Component*
Component ::= ID Sig_decl [Par_decl] Statement*
Sig_decl ::= {ID Int}* {ID Int}*
Par_decl ::= {Type ID}*
Type ::= “int” | “comp”
CiM DSL Syntax - Components

- Act like functions
  - Represent library function or CiM skeleton
  - Input/output signals

Example program:

```plaintext
libmod add(add.lib);
libmod mul(mul.lib);

comp main<in[16] | out[1]> () {
  in[0:16] => inner_product(8) => out[0];
}

comp inner_product<a[n], b[n] | out[1]>(int n) {
  zip(a[0:n], b[0:n]) => copy[n](mul) *H* reduce(n/2, add) => out[0];
}

comp reduce<in[2*n] | out[1]>(int n, comp c) {
  in[0:2*n] => foldR<*H*>(map<i = n: /2: 0>(repeat[i](c))) => out[0];
}
```

Program ::= Circuit_decl* Component*
Component ::= ID Sig_decl [Par_decl] Statement*
Sig_decl ::= {ID Int}* {ID Int}*
Par_decl ::= {Type ID}*
Type ::= “int” |“comp”
CiM DSL Syntax - Components

- Act like functions
  - Represent library function or CiM skeleton
  - Input/output signals + additional parameters

- Example program:

```
libmod add(add.lib);
libmod mul(mul.lib);

comp main<in[16] | out[1]>() {
    in[0:16] => inner_product(8) => out[0];
}

comp inner_product<a[n], b[n] | out[1]>(int n) {
    zip(a[0:n], b[0:n]) => copy[n](mul) *H* reduce(n/2, add) => out[0];
}

comp reduce<in[2*n] | out[1]>(int n, comp c) {
    in[0:2*n] => foldR<*>H*>(map<i = n: /2: 0>(repeat[i](c))) => out[0];
}
```
CiM DSL Syntax - Components

- Act like functions
  - Represent library function or CiM skeleton
  - Input/output signals + additional parameters

- Example program:

  ```
  Program ::= Circuit_decl* Component*

  Component ::= ID Sig_decl [Par_decl] Statement*

  Sig_decl ::= {ID Int}* {ID Int}*

  Par_decl ::= {Type ID}*

  Type ::= “int” |“comp”

  libmod add(add.lib);
  libmod mul(mul.lib);

  comp main<in[16] | out[1]>(){
    in[0:16] => inner_product(8) => out[0];
  }

  comp inner_product<a[n], b[n] | out[1]>(int n){
    zip(a[0:n], b[0:n]) => copy[n] (mul) * H * reduce(n/2, add) => out[0];
  }

  comp reduce<in[2*n] | out[1]>(int n, comp c){
    in[0:2*n] => foldR<* H *>(map<i = n: /2: 0>(repeat[i](c))) => out[0];
  }
  ```
CiM DSL Syntax - Components

- Act like functions
  - Represent library function or CiM skeleton
  - Input/output signals + additional parameters
  - Referred to by name and parameters

- Example program:

```
libmod add(add.lib);
libmod mul(mul.lib);

comp main<in[16] | out[1]>(){
  in[0:16] => inner_product(8) => out[0];
}

comp inner_product<a[n], b[n] | out[1]>(int n){
  zip(a[0:n], b[0:n]) => copy[n](mul) *H* reduce(n/2, add) => out[0];
}

comp reduce<in[2*n] | out[1]>(int n, comp c){
  in[0:2*n] => foldR<*_H_*>(map<i = n: /2: 0>(repeat[i](c))) => out[0];
}
```

```
Program ::= Circuit_decl* Component*
Component ::= ID Sig_decl [Par_decl] Statement*
Sig_decl ::= {ID Int}* {ID Int}*
Par_decl ::= {Type ID}*
Type := “int” “comp”
```
CiM Compiler Implementation

• Developed using Spoofax *

CiM Compiler Implementation

• Developed using Spoofax *

• Four stages:
  • Parsing
  • Dereferencing
  • Scheduling, placement, and routing
  • Code generation

CiM Compiler Implementation

- Developed using Spoofax *

- Four stages:
  - Parsing
  - Dereferencing
  - Scheduling, placement, and routing
  - Code generation

- Parsing
  - Generated from syntax definition
  - Abstract Syntax Tree

• Dereferencing
  • Loop unrolling
  • Constant propagation
CiM Compiler Implementation

- Dereferencing
  - Loop unrolling
  - Constant propagation

- Scheduling, Placement, and Routing
  - Operator dependent
  - Creates system in IR
CiM Compiler Implementation

- Dereferencing
  - Loop unrolling
  - Constant propagation

- Scheduling, Placement, and Routing
  - Operator dependent
  - Creates system in IR
CiM Compiler Implementation

- Dereferencing
  - Loop unrolling
  - Constant propagation

- Scheduling, Placement, and Routing
  - Operator dependent
  - Creates system in IR
CiM Compiler Implementation

- Dereferencing
  - Loop unrolling
  - Constant propagation

- Scheduling, Placement, and Routing
  - Operator dependent
  - Creates system in IR

- Code Generation
  - System parameters
  - VHDL
  - C code using pslib library
Results - Testing

- Testing Functions:
  - Inner product & matrix multiplication
  - Finite Impulse Response filter
  - Bitonic Sorting Network
Results - Testing

- Testing Functions:
  - Inner product & matrix multiplication
  - Finite Impulse Response filter
  - Bitonic Sorting Network

- Functional blocks:

<table>
<thead>
<tr>
<th>Func. Block</th>
<th>Latency [CC]</th>
<th>Width</th>
<th>Height</th>
<th>Energy [nJ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add *</td>
<td>178</td>
<td>9</td>
<td>32</td>
<td>124,8</td>
</tr>
<tr>
<td>Mul #</td>
<td>803</td>
<td>256</td>
<td>128</td>
<td>4407,8</td>
</tr>
<tr>
<td>GT</td>
<td>27</td>
<td>128</td>
<td>192</td>
<td>93</td>
</tr>
<tr>
<td>Copy #</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>12,8</td>
</tr>
</tbody>
</table>

Results – Expected Output Inner Product
Results – Graphical Output Inner Product

\[
\begin{array}{cccc}
  \text{mul} & \text{mul} & \text{mul} & \text{mul} \\
  \text{mul} & \text{mul} & \text{mul} & \text{mul} \\
  \text{mul} & \text{mul} & \text{mul} & \text{mul} \\
  \text{mul} & \text{mul} & \text{mul} & \text{mul} \\
\end{array}
\]
Results – Graphical Output Inner Product

May 3, 2017
Results – Graphical Output Inner Product
Results – Graphical Output FIR filter
Results – Expected Output BSN
Results – Graphical Output BSN
Results – Multicore Comparison

- Multicore simulation:
  - Sniper * and McPAT #
  - Intel Xeon X7460

#: S. Li et al. “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures.” MICRO 42, 2009. ACM.
## Results – Multicore Comparison

- **Multicore simulation:**
  - Sniper * and McPAT #
  - Intel Xeon X7460

<table>
<thead>
<tr>
<th>Problem</th>
<th>Problem size</th>
<th>CiM</th>
<th>Multicore</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inner product</td>
<td>32768</td>
<td>3653</td>
<td>20448</td>
<td>73696</td>
</tr>
<tr>
<td>Matrix multiply</td>
<td>32x32</td>
<td>1753</td>
<td>19456</td>
<td>72704</td>
</tr>
<tr>
<td>FIR filter</td>
<td>64/256</td>
<td>12773</td>
<td>8192</td>
<td>147456</td>
</tr>
<tr>
<td>BSN</td>
<td>256</td>
<td>1401</td>
<td>58240</td>
<td>32768</td>
</tr>
</tbody>
</table>

#: S. Li et al. “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures.” MICRO 42, 2009. ACM.
Results – Multicore Comparison

- Multicore simulation:
  - Sniper * and McPAT #
  - Intel Xeon X7460

- Controller not considered
  - Not expected to impact performance or area

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inner product</td>
<td>32768</td>
<td>3653</td>
<td>20448</td>
<td>0.6332</td>
<td>0.1502</td>
<td>272.7</td>
<td>15.99</td>
<td>74.7</td>
</tr>
<tr>
<td>Matrix multiply</td>
<td>32x32</td>
<td>1753</td>
<td>19456</td>
<td>0.5943</td>
<td>0.1500</td>
<td>174.5</td>
<td>35.95</td>
<td>95.5</td>
</tr>
<tr>
<td>FIR filter</td>
<td>64/256</td>
<td>12773</td>
<td>8192</td>
<td>0.5075</td>
<td>0.1498</td>
<td>302.7</td>
<td>35.95</td>
<td>23.70</td>
</tr>
<tr>
<td>BSN</td>
<td>256</td>
<td>1401</td>
<td>58240</td>
<td>0.8019</td>
<td>0.0008</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

#: S. Li et al. “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures.” MICRO 42, 2009. ACM.
Results – FPGA Prototyping

- Behavioural verification using VHDL
- Generation of constraint file

Generated by Vivado
Conclusion

• Summary
  • Modular language for spatial patterns
  • Compiler schedules, places, and routes simultaneously
  • Verified correctness
Conclusion

• Summary
  • Modular language for spatial patterns
  • Compiler schedules, places, and routes simultaneously
  • Verified correctness

• Future work
  • More CiM functions and patterns
  • Connection with a CiM backend
  • Viability of language for FPGA design
Thanks