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

Yu, Jintao; Hogervorst, Tom; Nane, Razvan

DOI
10.1145/3060403.3060474

Publication date
2017

Document Version
Accepted author manuscript

Published in
GLSVLSI '17 Proceedings of the on Great Lakes Symposium on VLSI 2017

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

Jintao Yu Tom Hogervorst Razvan Nane
{j.yu-1, r.nane}@tudelft.nl

Abstract

Computation-in-Memory (CiM) is a new computer architecture template based on the in-memory computing paradigm. CiM can solve the memory-wall problem of classical Von Neumann-based computer systems by exploiting application-specific computational and *data-flow* patterns with the capability of performing both storage and computations of emerging resistive RAM technologies (e.g., memristors). However, to efficiently explore and design such radically new application-specific CiM architectures, we require fundamentally new algorithm specification and compilation techniques. In this paper, we introduce a domain-specific language to express not only the computational patterns of an algorithm but also its *spatial* characteristics. Furthermore, we design a compiler that is able to transform these patterns into highly-optimized CiM designs. Experiments demonstrate the functional correctness of the language and the compiler as well as an order of magnitude speedup improvement over a multicore system in both performance and energy costs.

Keywords: Domain specific language; computation-in-memory; algorithmic skeleton; memristor

1 Introduction

In classical Von Neumann-based computing systems, memory access and data transfer operations are becoming a *big bottleneck* for Big Data applications [5]. This is not only because of the limited transfer speed at which data is retrieved from and written back to memory, but also because performing a large amount of memory accesses incurs high energy costs. A solution is to design application-specific memristor-based Computation-in-Memory (CiM) architectures [9], which feature a large memristor crossbar to execute massively parallel applications. Because we can perform both computations and storage using memristors [19], no off-crossbar data transfers are required during execution, enabling CiM-based architectures to solve the memory wall bottleneck.

Application-specific CiM-based solutions can therefore result in significant performance gains over multicore systems [10]. To implement highly-optimized CiM architectures, a designer would need to explicitly *spatially program* the application-specific computational and data-flow patterns onto the crossbar. This is a new form of *Spatial Computation* paradigms [3], which map programs into completely distributed hardware. Although general purpose languages can be used for spatial computation [3], Domain-Specific Languages (DSLs) facilitate domain experts to generate optimal solutions [16]. Some DSLs have been developed for specific platforms, such as MaxJ for Data-Flow Engine [15] and ANML for Automata Processor Engines [8]. However, specifying and compiling customized application layouts for a memristor crossbar poses a new challenge. Due to the *passive nature* of the memristor, the mapping and routing results directly influences the scheduling phase [20] because data movements on the crossbar have to be controlled as well. As a result, existing DSLs for spatial computation are not applicable to memristor-based CiM architectures.

To solve aforementioned challenge, a new type of skeleton was proposed: the *CiM skeleton* [20]. Skeletons (formally referred to as *algorithmic skeletons*) are high-level software constructs used to hide the complexity of parallel computer systems from a programmer [6]. Additionally, CiM skeletons pro-

vide the scheduling, mapping, and routing information needed to program applications on the CiM architecture. In this paper, we propose a DSL and a DSL-based compilation flow to express and implement CiM skeletons. We make the following contributions:

- A skeleton-based Domain Specific Language, *CiM DSL*, to describe the low-level crossbar details of the *spatial* patterns of an algorithm.
- A compiler that generates a scheduled, mapped, and routed system from this DSL using *CiM Skeletons*.
- The verification of this compiler and comparison with a multicore system.

The paper is organized as follows. Section 2 provides background information about the CiM architecture. Section 3 and Section 4 describe the CiM DSL and the implementation of its compiler. Thereafter, Section 5 validates the compiler with multiple applications. Finally, Section 6 concludes the paper.

2 Background

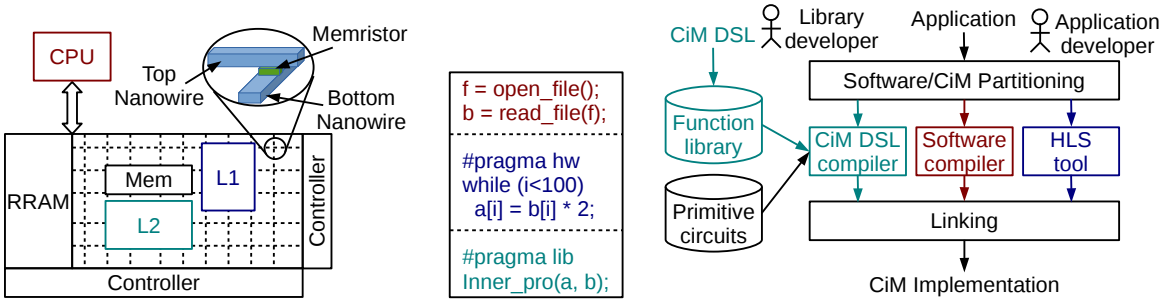
The CiM architecture is a computational template in which application-specific accelerators are instantiated and executed in-memory under the control of a CPU. Figure 1a shows a high-level view of the CPU/CiM heterogeneous system. CiM's hardware consists of two parts: a large reconfigurable crossbar of horizontal and vertical nanowires with memristors on every intersection, and a CMOS-based controller that activates voltages on the nanowires to control the memristors. Memristors can be configured as memory elements as well as computation elements [19], allowing both to be performed in the crossbar. The high density of a memristor crossbar enables this architecture to fully exploit the parallelism in embarrassingly parallel applications. Furthermore, since a CiM crossbar coexists with the general memory space (RRAM in Figure 1a), we do not have any off-chip memory accesses. This leads to significant improvements in both performance and energy consumption.

Figure 1c shows CiM's programming model. Before programmers can use application-specific accelerators in CiM, an algorithm designer is required to create the library functions (i.e., accelerators) optimized for *minimum communication maximum parallelism* on the crossbar. The algorithms are specified in the CiM DSL, using *skeleton operators* to specify not only the floorplan but also the *communication paths*[1] between the computational units. Please note that in the CiM crossbar, each corner point that changes the direction of the path needs to be explicitly controlled [18]. As a result, the scheduling is dependent on the mapping and routing. The defined CiM skeletons are used further to specify the complete algorithm that is stored in the *function library*. Subsequently, an application programmer can use this library in a different high-level language (e.g., C). Therefore, CiM DSL is designed for the library developers rather than the application developers. Functions not contained in the library are processed by a CiM High-Level Synthesis (HLS) tool. Finally, the compilation results of different tools are linked together. Figure 1a illustrates one possible compilation result of the sample program shown in Figure 1b. One circuit is generated using the library functions (colored green), and another one is generated by the CiM HLS tool (colored blue). Code that doesn't need to be executed on the CiM crossbar (colored red), is compiled regularly for the CPU. In this paper we focus only on the CiM DSL compiler.

3 CiM DSL

In this section, we describe the rules of CiM DSL's syntax using one variant of Extended Backus-Naur Form (EBNF) [2]. We design the DSL with the goal to create and use CiM skeletons easily. Figure 2 shows the dependence between different CiM concepts. First, we define *skeleton language operators*, which are language constructs that are used to specify different connections between functional blocks and their relative position. Subsequently, the operators are used to define *CiM Skeletons*. Finally, both skeleton operators and CiM skeletons are used to build library functions.

[1] We refer to communication paths simply as paths in the remainder of the paper



(a) CiM architecture (b) Program sample (c) CiM compilation tool-chain

Figure 1: CiM architecture and compilation tool-chain.

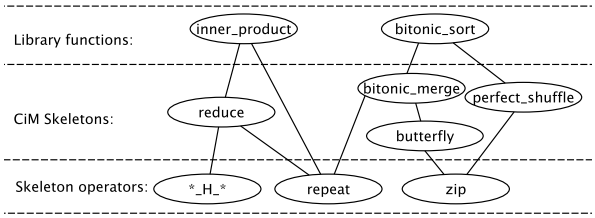
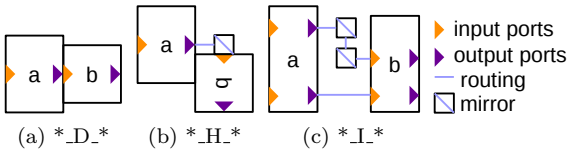


Figure 2: Dependence of CiM skeletons and library functions.



(a) *_D_* (b) *_H_* (c) *_I_*

Figure 3: Implementation of operators.

3.1 Circuits and Expressions

The CiM DSL creates systems by connecting functional blocks together in expressions. The syntax related to this is as follows:

$$Circuit\_decl ::= ID\ File\_name \quad (1)$$

$$Exp ::= Circuit \mid Int \mid ID\ Exp_1\ Exp_2\ ... \quad (2)$$

$$\mid Exp_1\ Op\ Exp_2 \mid Map \mid Fold \mid Repeat \quad (3)$$

$$Op ::= *\_D\_* \mid *\_H\_* \mid *\_I\_* \quad (4)$$

$$Map ::= ID\ Range\ Exp \quad (5)$$

$$Fold ::= Op\ Exp \quad (6)$$

$$Repeat ::= Int\ Exp \quad (7)$$

$$Range ::= Int \mid Int\ [Ar\_Op\ Int]\ Int \quad (8)$$

$$Ar\_Op ::= \text{“}+\text{”} \mid \text{“}-\text{”} \mid \text{“}*\text{”} \mid \text{“}/\text{”} \quad (9)$$

A *circuit declaration* (Circuit_decl) declares the name of the primitive circuit used in the program and specify its library file (Rule 1). This file contains all the circuit's information needed by the CiM compiler, including its latency, area, energy consumption, positions of input and output ports, initiation interval, and VHDL model. An *expression* can be a primitive circuit, an integer, or an instantiation of a CiM skeleton (Rule 2). To instantiate a CiM skeleton, the user needs to specify the component name that represents the skeleton, and assign the parameters. Two expressions and an operator (Op) constitute a new expression (Rule 3). The *operators* are *_D_*, *_H_*, and *_I_*. Operators represent different mapping strategies as shown in Figure 3. The mapping is performed according to the positions of input and output ports. *_D_* puts two circuits next to each other so that an input port is directly linked with the output port. *_H_* rotates one circuit and uses a mirror in between. The mirror changes the direction of a path [18]. *_I_* links two groups of input and output ports between two components using mirrors.

Other forms of expressions include *map* (Rule 5), *fold* (Rule 6), and *repeat* (Rule 7). These expressions are useful to generate larger circuits. For example, *map* applies every number in a *range* to an expression that contains an variable. The *range* is an array of integers, which can be expressed using one to three fields (Rule 8). If the range has one

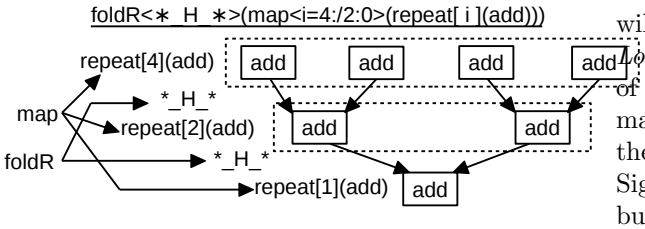


Figure 4: Expression and data-flow graph of an adder tree.

field, it is a single value. If it has two fields, it is a series of incrementing integers from the first up to but not including the second value. If it has three fields, it represents a discontinuous array of integers. In this case, the arithmetic operator (Ar_Op) and the number in the middle indicate the intervals. The *arithmetic operators* are add ("+"), subtract ("-"), multiply ("×"), and divide ("/") (Rule 9). The bracket ("[]") surrounding "*Ar_Op Int*" means they are optional.

We use an example shown in Figure 4 to explain the semantics of *repeat*, *fold*, and *map* expressions. The example is an adder tree that calculates the sum of eight values. The CiM DSL line that describes this circuit is shown at the top of the figure. $Repeat[n](add)$ creates an array of n additions. The *map* expression duplicates the *repeat* expression following a noncontinuous array [4, 2, 1]. These duplicates are linked with *_H_* operator, which is specified by the *fold* expression. The associativity of *fold* expressions is indicated by the letter "L" or "R" at the end.

3.2 Statements and Signals

Statements connect the circuits declared in expressions to signals, to give explicit control over the data flow to and from those circuits. A *Signal* is a connection path between two data locations. The relevant CiM DSL syntax is as follows:

$$Statement ::= Signal\ [Exp]\ Signal \mid Loop \quad (10)$$
$$Loop ::= ID\ Range\ Statement+ \quad (11)$$
$$Signal ::= ID\ Range \mid Signal\ +\!+\ Signal \quad (12)$$
$$\mid Zip(Signal,\ Signal) \quad (13)$$

A statement contains an input signal, an output signal and optionally an expression (Rule 10). It will be translated into a group of primitive circuits. *Loops* can be used to compactly express a group of similar statements, which is similar to the semantics of *map* (Rule 11). The symbol "+" means the loop structure accepts one or more statements. Signals have a name and a range, and they can be built using *signal operators* "++" (Rule 12) and "zip" (Rule 13). "++" concatenates two signals sequentially and "zip" builds a single signal from the elements of two signals interleaved with one another.

Statements are allowed to specify only the connection between input and output signals, without including any component (Rule 10). This feature is useful for shuffling the signals. Listing 1 shows an example of a signal shuffle, specifically one named the butterfly pattern, which is used in a bitonic sort function.

Listing 1: Butterfly Shuffle Statement

```
1  zip(in[0:2:m/2], in[m/2:2:m]) ++
2      zip(in[1:2:m/2], in[1+m/2:2:m]) => out[0:
           m];
```

3.3 Programs and Components

A complete CiM DSL program consists of one or more circuit declarations and one or more components (Rule 14). A *component* describes a library function or a CiM skeleton. The CiM DSL syntax concerning these two language constructs is as follows:

$$Program ::= Circuit\_decl^*\ Component^* \quad (14)$$
$$Component ::= ID\ Sig\_decl\ [Par\_decl]\ Statement^* \quad (15)$$
$$Sig\_decl ::= \{ID\ Int\}^*\ \{ID\ Int\}^* \quad (16)$$
$$Par\_decl ::= \{Type\ ID\}^* \quad (17)$$
$$Type := \text{"}int\text{"} \mid \text{"}comp\text{"} \quad (18)$$

The head of a component contains its name (ID), signal declaration (Sig_decl), and parameter declaration (Par_decl) (Rule 15). The square brackets surrounding Par_decl means it is optional. Among all the components in a program, one and only one component should be named as "main". It represents the library function defined by the program. The *signal declaration* declares input and output

signals, containing names (ID) and their sizes (Int) (Rule 16). *Parameter declaration* specifies the type and names of the parameters (Rule 17). Currently, CiM DSL supports two *types* of parameters, which are integer ("int") and components ("comp") (Rule 18). The body of a component is one or more statements (Rule 15), which link expressions with input and output signals (Rule 10).

As an example of a complete CiM DSL program, Listing 2 shows the code for matrix multiply $A_{m\times n} \times B_{n\times k}$ in CiM DSL. It is built based on the inner product function. To achieve the best performance, the primitive circuits used in the inner product are arranged following an H-tree pattern [10]. Please refer to Figure 6a for a visualization of the target layout. The inner product hardware is duplicated to calculate all the elements of the result matrix in parallel.

Listing 2: Matrix Multiply in CiM DSL

```
1  libmod add(add.lib);
2  libmod mul(mul.lib);
3  comp main<in[32] | out[16]>(){
4    in[0:32]=> matrix_multiply(4, 4, 4) =>out
         [0:16];
5  }
6  comp matrix_multiply<A[m*n], B[n*k] | out[m*k]>
7      (int m, int n, int k){
8    forV i=0:m do
9      A[n*i: n*i+n] ++ B[0: n*k] =>
10         row(n,k) => out[k*i:k*i+k];
11   end
12 }
13 comp row<a[n], b[n*k] | out[n]>(int n, int k){
14   forH i=0:k do
15     a[0:n]++b[n*i:n*i+n]=>inner_product(n)=>out
           [i];
16   end
17 }
18 comp inner_product<a[n], b[n] | out[1]>(int n){
19   zip(a[0: n], b[0: n]) => repeat[n](mul)
20       *_H_* reduce(n/2, add) => out[0];
21 }
22 comp reduce<in[2*n] | out[1]>(int n, comp c){
23   in[0: 2*n] => foldR<*_H_*>(map<i = n: /2: 0>
24       (repeat[i](c))) => out[0];
25 }
```

Line 1 and line 2 declare two primitive circuits, i.e. the adder (add) and the multiplier (mul). The keyword of this declaration is "libmod", and the name suffix of the library files is ".lib". After these declarations, the program defines five components with the key word "comp". The "main" component (line 3 to line 5) specifies that the sizes of two matrices are both four by four. The declaration of input and output ports are surrounded with a pair of angle brackets and separated with a delimiter ("|"). On the other hand, the parameter declarations are marked with parentheses (line 7, line 13, etc.). The body of components are surrounded with curly braces and consists of one or more statements. For each statement, the expression and signals are linked by "=>", which means that input locations (e.g., registers/storage locations in the crossbar) are transferred via signals to the matrix multiplication's input ports. Note the use of the "++" signal operator in lines 10 and 15 to concatenate two arrays into one, and of the "zip" signal operator in line 19 to zip the elements of two arrays into one. *matrix_multiply* and *row* components both employ a *for-loop*, which expresses a group of statements. They arrange the circuits that are obtained from the loop statements vertically (forV) or horizontally (forH). Therefore, line 15 duplicates k *inner_product* function in a row, and line 10 further extends these rows into a matrix. The *repeat* expression in lines 19 and 24 represents duplicates of a circuit, which are executed in parallel. It does not contain mapping information and will be further mapped using operators. Line 23 builds a binary tree as shown in Figure 4. The map expression duplicates a circuit using a variable (i) in the range (n:/2:0). The range is divided into three fields by colons (:). The first field (n) is the starting number. The second field (/2) is applied to n repetitively until the value reaches the third field (0). All these intermediate values, not including the third field, constitute the range. Finally, the group of circuits represented by *map* are linked with the operator specified in the *fold* expression (i.e., *_H_*).

4 Implementation

We developed CiM DSL compiler using Spoofax, a language workbench in the Eclipse IDE [11]. The development consists of two phases. First, we define the syntax and the name binding rules, which are used to parse user programs into an Intermediate Representation (IR). Second, we define transformation and generation rules, which annotate and transform the IR according to our defined skeleton operators, and emit the code for the desired circuits, respectively. Spoofax has good support for both phases so that the workload of developing the new DSL is significantly reduced. After defining all the rules, the CiM DSL compiler is generated by Spoofax.

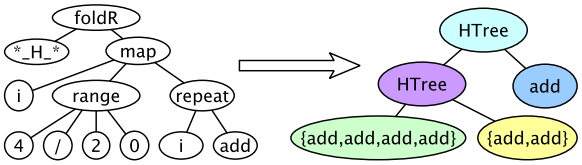


Figure 5: Dereferencing of *reduce(4, add)*.

A CiM DSL program is compiled into circuits following four steps, namely parsing, dereferencing, transforming, and emitting. The parsing builds an Abstract Syntax Tree (AST), which is subsequently dereferenced to eliminate language structures such as *for-loop*, *fold*, *map*, and *repeat*. During dereferencing, the loops are fully unrolled, and *fold*, *map*, and *repeat* are applied. Figure 5 shows this process for the expression in the *reduce* component in Listing 2 (line 22). In this case, we assume n is four. The left side illustrates the AST obtained after parsing, where the range is calculated into a noncontinuous array. Then, *map*, *repeat*, and *foldR* are applied successively as described in Section 3. The result of the deferencing is also an AST in which all the leaf nodes are primitive circuits, and all the internal nodes are skeleton operators. The result AST is shown in the right part of Figure 5, where the *_H_* operator is replaced by the name *HTree* that implements this pattern.

The transform process constructs circuits based on the AST that is obtained in the previous step. The circuit construction is performed using a post-order, depth-first tree traversal using the *Transform_ske* function that is dynamically dispatched according to the type of the operators. The function contains the scheduling, placement, and routing algorithms that will be applied to the circuits according to the particular CiM skeleton that it implements. Figure 7 shows an example of transforming the AST on the right of Figure 5. Its left part shows the leaf nodes of the dereferenced AST and the circuits they represent. The right part is the transformation of this AST, which is done in two steps. The lower Htree node is transformed first, which combines four adders and two adders into two group adders. Next, these two groups are transformed by the root node to form a whole circuit together with another adder. The small squares shown in this figure represent mirrors.

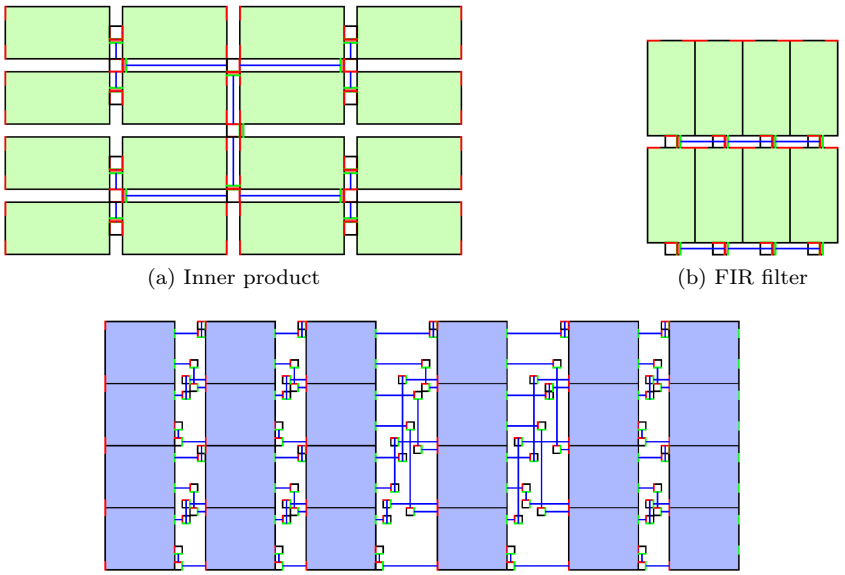
Finally, the code generation phase emits three types of files, which are VHDL, mapping constraints, and graphic outputs. The VHDL contains the port maps and an FSM, which are produced according to the mapped, routed, and scheduled Data Flow Graph (DFG) of the algorithm. This VHDL can be used for behavioral simulation. We generate graphic output to examine the placement and routing results. This output is in the C language, invoking a graphic library named *pslib* to produce a graph in postscript (.ps) format.

5 Experimental Results

We use four functions to validate CiM DSL and its compiler, which are vector inner product, matrix multiply, Finite Impulse Response (FIR) filter, and bitonic sort. The code of the first two functions and a small part of bitonic sort is shown in Listing 2 and Listing 1. The data type used in these functions are 32-bit integer.

5.1 Compiler Outputs

The attributes of primitive circuits we used in this case study are listed in Table 1. The *adder* (Add) and the *multiplier* (Mul) are designed by previous works [12, 1]. The original design is not based on a crossbar, so the area and latency are moderately different when we adapt it to CiM. We will not discuss these changes since the hardware design is beyond the scope of this work. The *Greater than* (Gt) component, which is used in bitonic sort, cannot be found in existing works. Therefore we estimate their attributes. The latency is listed in terms of Clock Cycle (CC). The area is represented by the required number of rows (Height) and columns (Width). The energy consumption of the adder and the multiplier is not given in the original papers [12, 1]. Actually, it is impossible to report accurate energy consumption at the design phase because this is input data dependent. However, we can estimate the maximum value by assuming every IMPLY or FALSE operation [12] consumes the energy of switching states. This energy varies among technologies, from 0.1 fJ [17] to 230 fJ [13]. In this paper, we use 100 fJ. We also calculated the energy consumption for copy operation following the same way. Its implementation, which is essentially two NOT operations, is taken from [1].



(a) Inner product (b) FIR filter

(c) Bitonic sort

Figure 6: Graphic output.

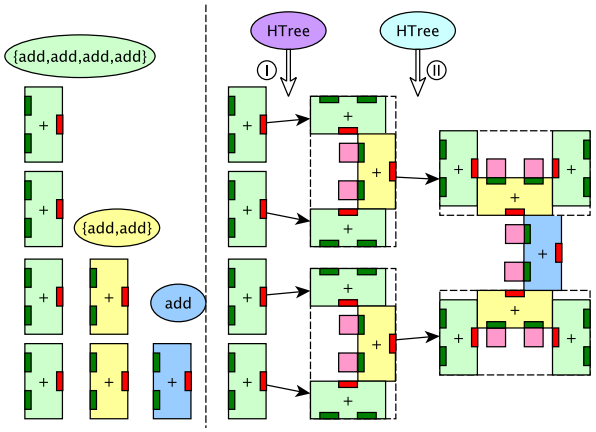


Figure 7: Transformation of *reduce(4, add)*.

Table 1: Attributes of Primitive Circuits and Copy Operation

	Latency	Width	Height	Energy	Ref.
Add	178	9	32	124.8	[12]
Mul	803	256	128	4407.8	[1]
Gt	27	128	192	93	–
Copy	3	–	–	12.8	[1]

We generated the graphic outputs of inner product, FIR filter, and bitonic sort as shown in Figure 6. Matrix multiply is not presented because it is a matrix of inner products, which is very large but contains little information. The vector size of inner product is 16. The tap size of FIR filter is four and the input size is two. The input size of bitonic sort is eight. In Figure 6a and Figure 6b, the large and small boxes denote multipliers and mirrors respectively. The adders are tiny bars beside the mirrors. In Figure 6c, the small boxes are also mirrors while the large ones are *gt*s. The red and green dashes in these figures indicate the input and output ports while the blue lines show the routing. The layouts are the same as we designed, which demonstrates the DSL works as intended.

5.2 Performance Evaluation

We enlarged the problem sizes and compared the performance of generated circuits with a multicore platform. The problem sizes and the attributes of the generated circuits are listed in Table 2. For the FIR filter, the tap size is 64 and the input size is 512. We calculated the area of the generated circuits based on the memrisor density predicted by ITRS [7], which is 2.38×10^{11} bit/cm^2. These library functions are simulated using Sniper [4],

Table 2: Experimental Results

	Problem size	CIM					Multicore		
		Lat/CC	Width	Height	Area/mm^2	Energy/mJ	Lat/μs	Energy/mJ	Speedu
Inner product	32768	3653	20448	73696	0.6332	0.1502	272.7	15.99	74.65
Matrix multiply	32×32	1753	19456	72704	0.5943	0.1500	174.5	21.96	99.54
FIR filter	64/512	12773	8192	147456	0.5075	0.1498	302.7	35.95	23.70
Bitonic Sort	256	1401	58240	32768	0.8019	0.0008	–	–	–

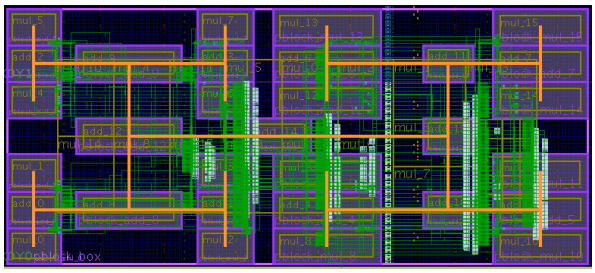


Figure 8: FPGA floorplanning for inner product.

and the energy consumption is reported by McPAT [14]. The targeted multicore system is Intel Xeon X7460 processor that consists of six cores on a die of 503 mm^2, running at 2.66 GHz each. These cores have 64 kB L1 cache each and share a 16 MB L3 cache. Every two cores share an L2 cache of 3 MB. The latency (Lat) and energy consumption are also listed in Table 2. The speedup of CiM's latency over multicore is calculated, which is between 23x and 99x. The area of the circuits generated by CiM compiler is very small compared with this processor. The energy consumption is less than 1% of the multicore. Please note that we do not include the CMOS controller in the energy evaluation because there is no backend synthesis tool yet for the CiM system. However, we do not expect the controller to have a big impact on the reported numbers due to its simplicity.

5.3 FPGA Prototyping

To confirm the validity of the graphical output, we built also an FPGA prototype to simulate the layout of the CiM inner product design on FPGA. We synthesized the generated area constraint file and the VHDL file with Xilinx Vivado. The implemented design of inner product is shown in Figure 8. Primitive circuits are recognizable by their yellow borders, and connections between circuit are represented by orange lines. The sizes of the adders and multipliers are different from Figure 6a. Despite this difference in visualization of the connections, we can verify that the floorplanning and interconnect information was included in the VHDL and constraint files correctly.

6 Conclusion

In this paper, we introduce a DSL and compiler to design programs to be run on future CiM-based systems. The skeleton-based DSL allows for the modular, high-level description of a system, and the compiler schedules, places, and routes the system using information provided by CiM skeletons. The functional correctness of the DSL is verified using VHDL files generated by the compiler, and the mapping and routing results are confirmed by generating graphical output files. This DSL can also be used for FPGA designs and it will be investigated in future work.

References

[1] K. Bickerstaff and E. E. Swartzlander. *Memristor-Based Addition and Multiplication*, pages 473–486. Springer International Publishing, Cham, 2014.

[2] T. Bray, J. Paoli, C. M. Sperberg-McQueen *et al.* *Extensible markup language (XML).* World Wide Web Consortium Recommendation, 1998.

[3] M. Budiu, G. Venkataramani, T. Chelcea *et al.* Spatial computation. ASPLOS XI, pages 14–26, New York, NY, USA, 2004. ACM.

[4] T. E. Carlson, W. Heirman, S. Eyerman, *et al.* An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.*, 11(3):28:1–28:25, Aug. 2014.

[5] C. P. Chen and C.-Y. Zhang. Data-intensive applications, challenges, techniques and technologies: A

survey on big data. *Information Sciences*, 275:314 – 347, 2014.

[6] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, USA, 1991.

[7] I. R. Committee. International technology roadmap for semiconductors 2.0. Technical report, 2015.

[8] P. Dlugosch, D. Brown, P. Glendenning, *et al.* An efficient and scalable semiconductor architecture for parallel automata processing. *TPDS*, 25(12):3088–3098, Dec 2014.

[9] S. Hamdioui, L. Xie, H. A. D. Nguyen *et al.*. Memristor based computation-in-memory architecture for data-in- tensive applications. DATE '15, pages 1718–1725, San Jose, CA, USA, 2015. EDA Consortium.

[10] A. Haron, J. Yu, R. Nane, *et al.* Parallel matrix multiplication on memristor-based computation-in-memory architecture. HPCS '16, pages 759–766. IEEE, July 2016.

[11] L. C. Kats and E. Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. OOPSLA '10, pages 444–463, New York, USA, 2010. ACM.

[12] S. Kvatinsky, G. Satat, N. Wald, *et al.* Memristor-based material implication (imply) logic: Design principles and methodologies. *VLSI*, 22(10):2054–2066, Oct 2014.

[13] S. Lee, J. Sohn, Z. Jiang *et al.* Metal oxide-resistive memory using graphene-edge electrodes. *Nature communications*, 6(8407):1–7, 2015.

[14] S. Li, J. H. Ahn, R. D. Strong *et al.* Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. MICRO 42, 2009. ACM.

[15] O. Pell, O. Mencer, K. H. Tsoi *et al.* *Maximum Performance Computing with Dataflow Engines*, pages 747–774. Springer New York, New York, NY, 2013.

[16] The OpenSPL Consortium. Openspl: Revealing the power of spatial computing. Technical report, Dec. 2013.

[17] C.-L. Tsai, F. Xiong, E. Pop *et al.* Resistive random access memory enabled by carbon nanotube crossbar electrodes. *Acs Nano*, 7(6):5360–5366, 2013.

[18] L. Xie, H. A. D. Nguyen, M. Taouil *et al.* Interconnect networks for memristor crossbar. NANOARCH '15, pages 124–129. IEEE, July 2015.

[19] J. J. Yang and R. S. Williams. Memristive devices in computing system: Promises and challenges. *J. Emerg. Technol. Comput. Syst.*, 9(2):11:1–11:20, May 2013.

[20] J. Yu, R. Nane, A. Haron *et al.* Skeleton-based design and simulation flow for computation-in-memory architectures. NANOARCH '16, pages 165–170. IEEE, July 2016.