MENTAI: A Fully Automated CGRA Application Development Environment that Supports Hardware/Software Co-design

Ayaka Ohwada, Takuya Kojima and Hideharu Amano
Graduate School of Science and Technology, Keio University,
Hiyoshi 3–14–1, Kohoku-ku, Yokohama, Kanagawa, 223–8522 Japan
{ohwawa, tkojima, hunga}@am.ics.keio.ac.jp

Abstract

Energy-efficient coarse-grained reconfigurable architectures (CGRAs) have been attracting attention as accelerators for IoT devices. CGRAs have several processing units called PEs (Processing Elements) arranged in a two-dimensional array. CGRAs can achieve high energy efficiency by adaptively changing the operation executed for each PE and the connections between PEs. When running an application on CGRAs, it is necessary to convert the target application to a data flow graph (DFG) and map the DFG to the PE array. Currently, there is not enough research on the CGRA application development environment, and it is hard to say that the environment is in place. In this paper, we propose MENTAI, a fully automated application development environment for the CGRAs based on LLVM. As a case study, we evaluated it using CCSOTB2, which is a kind of CGRA. Thanks to MENTAI, a compute-intensive part of an application is executed on the CCSOTB2, instead of a host processor. Besides, it considers the energy efficiency of the entire system, including the host processor. Thereby, MENTAI achieves around 60 % energy reduction while keeping the execution time.

I. Introduction

In recent years, IoT and wearable computing have become widespread. Devices for them are needed to achieve high processing performance with low power consumption, and reconfigurable devices are attracting attention.

Coarse-grained reconfigurable architecture (CGRA) is a type of reconfigurable device that has features between FPGAs and ASICs. While CGRAs are less programmable than FPGAs, they outperform FPGAs in energy efficiency. In a CGRA, multiple processing elements (PEs) are arranged in a two-dimensional array and interconnected with each other. A PE consisted of an ALU and a Switch Element (SE) can reconfigure (1) the operation to be executed by the ALU and (2) the interconnection between PEs according to the target application.

Various CGRAs such as ADRES [1], DT-CGRA [2], and FloRA [3] have been studied. CGRAs can be divided into two categories according to their reconfiguration style. One changes the configuration of the PE array cycle-by-cycle for a single task [1] [3], whereas the other keeps it during the task [2]. Although code generation for cycle-level CGRA is easier than task-level CGRA, power consumption increases because reconfiguration is performed for each cycle. Thus, the task-level reconfiguration has an advantage of energy efficiency.

In general, an application for the CGRAs is compiled as follows the flow shown below: First, extracting the part to be offloaded to the CGRA in the application, and generate its data flow graph (DFG). Second, mapping the DFG to the target PE array. Finally, scheduling data input/output control.

With the attention to CGRAs growing, programming environments for them have been actively researched, and some of them are available [4] [5] [6]. However, many of them focus on the CGRAs with cycle-level reconfiguration [7], which is relatively easy to generate the operational code by using the dynamic reconfiguration, and others aim to the architectural exploration rather than developing applications on them. Also, although most of CGRAs are used as an accelerator, few tools care about the total system consisting of the host and CGRAs. In this study, we propose such an integrated CGRA application development environment called MENTAI (Macro-based ENergy-efficienT Application Integration) based on LLVM [8].

MENTAI is a tool for CGRAs to support task-level reconfiguration, which has a controller for data transfer between the data memory and the PE array. This tool focuses on energy efficiency. It allows users to describe applications in C language and hides architectural details from the user. Also, MENTAI automatically generates the code required for the operation of both the CGRA and the host processor from the C source code. That is, (1) configuration data necessary for operating the CGRA, (2) assembly code for the controller, and (3) code for the host processor can be generated at once from the source code. Users are not responsible for developing each part separately and can easily use the entire system, including the CGRA. In this paper, CCSOTB2 [9], which is a type of CGRAs that performs task-level reconfiguration, was used as a case study.

II. Related Work

There are two main types of CGRA-related tools: (1) mainly for design space exploration, and (2) for software development. Black Diamond [6], CGRA-ME [4], and DAEGEN [10] fall in the former classification.

Black Diamond is a compiler for dynamically reconfigurable architectures. By giving the target architecture as a Graph with Configuration Information (GCI), applications can be compiled for various architectures. Users write applications using a C-based front-end representation. A description example is shown in Code1.

Code 1 Black Diamond sample code

```
1  call [in0] MEM_OUT_00();
2  [@"PE_101"]call [r0] AND(in0,maskr);
3  [@"PE_001"]call [r0] SR(r0, sixteen);
4  [@"PE_200"]calculate r0 = r0 * c_r;
5  [@"PE_201"]call [g0] AND(in0,maskg);
6  [@"PE_100"]call [g0] SR(g0, eight);
7  [@"PE_300"]calculate g0 = g0 * c_g;
8  [@"PE_401"]calculate out0 = r0 + g0;
9  [@"PE_601"]call [out0] SR(out0,ten);
10 call [] MEM_IN_00(out0);
```

Users describe which PE performs which operation and the dependencies between the operations. The DFG is generated from such a description by the user, that is, the abstraction level is low in this tool.

CGRA-ME is a framework that includes CGRA architecture description, modeling, application mapping, and system simulation. It uses the LLVM for the front end, and can generate DFG in the *dot* file format by analyzing LLVM IR from a program written in C language. While CGRA-ME focuses on CGRA alone, such as architecture description, DFG generation, and mapping, this study implements a library to support control such as data transfer between the CGRA and the host processor. In other words, the support range is different from MENTAI.

DAEGEN is a modular accelerator design framework. It targets accelerators on which the application is mapped spatially. It can separate memory access and computation, such as CGRAs, to perform task-level reconfiguration. DAEGEN is a tool to seek the best architecture to execute an application, while the proposed tool is a software-hardware co-development environment to apply an application to an input architecture.

CCF [7], DFGenTool [5], and Musketeer [11] are tools not for the design space exploration. CCF is a compiler framework for CGRAs with the cycle-level reconfiguration. Users can annotate performance-critical loop with pragmas, and CCF compiler can generate the code for the application's execution on the CGRA. As mentioned, the cycle-level reconfiguration, which CCF considers, has few restrictions on executable applications. In contrast, the task-level reconfiguration has many restrictions. Therefore, loosely constrained pragma-based methods such as CCF are not suitable for task-level CGRAs.

DFGenTool is a tool to generate the DFG for CGRAs in the *dot* file format using the LLVM from a high-level language such as C language with a pragma annotation. The *dot* file generated by DFGenTool emphasizes readability and cannot be mapped as it is. However, it does not include a specific back-end compiler to map the DFG to the PE array. On the other hand, MENTAI integrates our mapping tool GenMap [12] into the compilation flow.

Musketeer [11] is an application framework for DRP (Dynamically Reconfigurable Processor). Musketeer can write applications in the C language as well as MENTAI and CGRA-ME. While Musketeer is a framework for DRP, i.e., available for CGRA using cycle-level reconfiguration, MENTAI is available for CGRA using task-level reconfiguration.

III. CMA Architecture

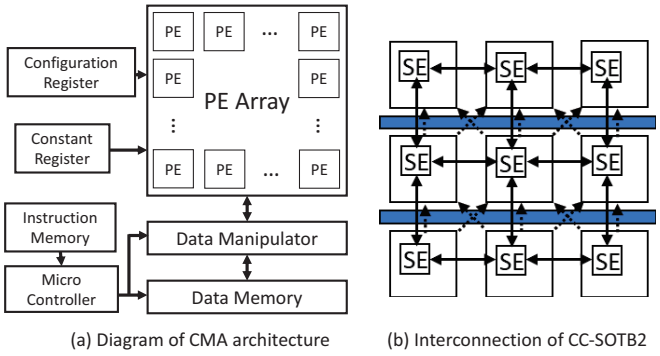

Fig. 1. CMA architcture and PE interconnection

As a case study, we carry out evaluations targeting CCSOTB2 [9]. CCSOTB2 belongs to the CMA (Cool Mega Array) architecture proposed as an energy-aware CGRA [13]. In CMA architecture, the configuration of the PE array is changed task-by-task. The power consumption is cut down by removing the clock tree and registers from a PE and build the PE array as a combinational circuit. Fig. 1 (a) shows a diagram of CMA architecture. In CMA architecture, computation and memory access are separated. Data transfer between the PE array and the data memory is done by using a μ controller, which is a processor to execute 16-bit instructions. A data manipulator is placed between the PE array and the data memory. It is composed of several multiplexers to support a flexible data transfer. The PEs form an island-style interconnection network, as shown in Fig. 1 (b). CCSOTB2 has a 12 $\times$ 8 PE array and configurable pipeline registers placed between every PE row. Each PE has one SE (Switching Element), and each SE has one channel.

A. Configuration Data

The configuration data specifies operations executed by PEs, and interconnection between other PEs, constant

registers, and pipeline registers. The configuration can be changed by writing configuration data to a configuration register outside the PE array.

B. Operation mapping and application execution

After the application is converted to the DFG, it is mapped to the PE array by the CGRA mapping tool GenMap [12], and configuration data are generated. It is necessary to prepare an assembly code for the μ controller corresponding to the mapped application. Here is an example of running RGB 24-bit gray-scale on the CGRA. This application has three inputs and three outputs when converted to the DFG. Code 2 runs the gray-scale in a 7-stage pipeline. Code 2 has a loop length of 24

Code 2 Assembly code for the μ controller

```
1     DELAY 7
2     LDI r3, #24
3     SET_LD #0x0, #3
4     SET_ST #0x48, #3
5  LP: LDST_ADD #0, #0
6     BNZD r3, LP
7     DONE
```

and means that three data are loaded from the address 0x0 of the data memory of each loop, and three data are stored from the address 0x48.

IV. Proposal: CGRA Application Development Environment

In this paper, we propose MENTAI, an application development environment that supports the entire system consisted of a host processor and the CGRA using LLVM. Here, the application, kernel, and context are used in the following meaning.

- Application: The entire program, including both where it runs on the host processor and where it offloads to the CGRA.
- Kernel: The part of the application that is offloaded to the CGRA.
- Context: The part of the kernel that is offloaded to the CGRA in one reconfiguration. The kernel may consist of multiple contexts or a single context.

A. Application Development Flow

Executing an application on CGRA, especially CMA, mainly requires the following two codes: (1) configuration data corresponding to the target DFG and (2) assembly code for the μ controller. In this proposal, we implemented the LLVM Passes that automate the generation of (1) DFG and (2) assembly code in Fig. 2. Light blue portions represent the implemented functionality described in this paper. Shaded portions represent existing tools. Besides, it includes a CGRA programming library implemented in C language. The development flow using this tool is as follows. Here, LLVM version 4.0.1 was used. The user writes the entire application including CGRA and host processor in C language and compile it using Clang. The LLVM IR of CGRA part is passed through a Multi Context Data Pass (MCDPass) for extracting context information. Duplicated load instructions are eliminated by Eliminate Duplicate Load Pass (EDLPass). The DFG is generated by the DFGPass, then information on the number of inputs and outputs of the DFG is generated as LLVM IR. AsmPass generates assembly code for the μ controller in CGRA. Finally, LLVM IR for the host processor is linked to the codes above and compiled to executable code.

A.1. CGRA kernel description rules

The following rules are given in describing the CGRA kernel.

- In the file that describes the application, the *PRELUDE* macro is described at the beginning.
- Each context is described as a function where the return value is void, and the arguments are all int type pointers.
- Describe the arguments in the order of the CGRA input data array and the output data storage destination array.
- Specify the number of inputs and outputs with the *CGRA* macro.
- Specify the loop length and operating frequency with *REGISTER_KERNEL* macro or *REGISTER_KERNEL_CUSTOM*.

The *PRELUDE* macro sets up to use the information which is analyzed by the passes. The *CGRA* macro has the role of giving information necessary for DFG generation and telling the subsequent pass which loop is mapped to the CGRA. *REGISTER_KERNEL* and *REGISTER_KERNEL_CUSTOM* are responsible for providing data necessary for generating assembly code. When the former is used, the operating frequency becomes the default value of 20 MHz, and when the latter is used, the user specifies it.

Here is Code 3 that describes the application converting 24-bit RGB to grayscale according to the above rules.

B. LLVM Passes

Four LLVM Passes are implemented as described below.

B.1. MCD Pass

Multi Context Data Pass (MCDPass) is a pass to convert context information described as a function into a form easily used in the subsequent pass. When writing an application, the *PRELUDE* macro defines some globale variables called dummy variables. Information, such as the number of input data arrays to the kernel and the number of kernel loops, is assigned to dummy variables by the *CGRA* macro or *REGISTER_KERNEL* macro. The

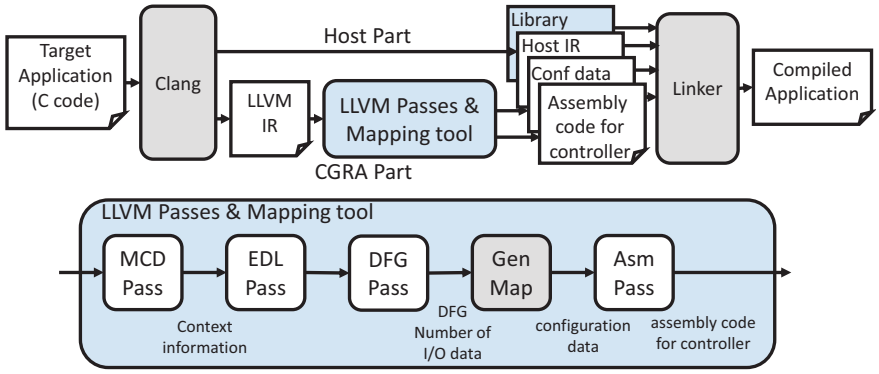

Fig. 2. Application development flow

Code 3 Entire code to execute 24-bit grayscale

```
1  PRELUDE;
2
3  void gray(int *in, int *out) {
4          CGRA(1, 1);
5          int i = 0;
6          out[i] = (((in[i] & 0x00ff) >> 16) * 306 +
7                  ((in[i] & 0x0000ff) >> 8) * 601 +
8                  (in[i] & 0x000000ff) * 117) >> 10;
9  }
10
11 REGISTER_KERNEL("gray");
12
13 int main() {
14         int result[48];
15     int index = get_kernel_index("gray");
16     cgra_setup(index);
17     send_host_to_cgra(index, 1, dmem, 48, 0);
18     cgra_run(index, 0, 0, 0);
19     wait_cgra();
20     send_cgra_to_host(index, 1, result, 48, 0);
21 }
```

substitution location is analyzed by MCDPass to generate data to be used in subsequent passes and libraries. Instructions for assigning values to dummy variables are unnecessary in the process after MCDPass. Therefore, they are deleted after the information is extracted.

B.2. EDLPass

Eliminate Duplicated Load Pass (EDLPass) is a pass to remove duplicate loads from the same memory address in the input sequence. Code 4 shows the LLVM IR fragment with overlapping loads. In Code 4, values are loaded from the same index of the same array of %7 and %16. In EDLPass, duplicate load instructions are deleted by replacing %16 of the multiply instruction with %7 and deleting the load instruction to assign a value to %16.

Code 4 LLVM IR fragment with overlapping load instructions

```
1  define void @sepia(i32* nocapture readonly, i32*
       nocapture) local_unnamed_addr #0 {
2    %7 = load i32, i32* %0, align 4, !tbaa !1
3    %8 = mul nsw i32 %7, 357
4    // Omitting Code
5    %16 = load i32, i32* %0, align 4, !tbaa !1
6    %17 = mul nsw i32 %16, 278
```

B.3. DFGPass

DFGPass is a pass to generate DFG from the input LLVM IR and to output it in dot format [14]. Generated DFG is passed to GenMap, which is the CGRA mapping tool. It also analyzes the number of inputs and outputs of the DFG and passes them to the subsequent Pass.

B.4. AsmPass

AsmPass is a pass to generate an assembly code that can be used by the μ controller. The pass outputs the transfer destination and the transfer source addresses of the data required for application execution in the LLVM IR format.

C. CGRA Library

The CGRA library provides functions for application execution, such as reconfiguration, data transfer between the host processor and CGRA, and instructions on the timing of starting the operations in CGRA. Here is Code. 3 for an application that converts 24-bit RGB to grayscale with the library functions. *get_kernel_index* function returns the number assigned to the kernel *gray*. This number is used to identify the context to be executed by the library functions. *cgra_setup* transfers configuration data to the CGRA and *send_host_to_cgra* send data to be processed. In Code 3, the host processor sends 48 data to the CGRA. *cgra_run* starts processing and *wait_cgra* checks DONE signal sent by the CGRA. *send_cgra_to_host* collects results from the CGRA.

V. Evaluation and Application Development Case Study

The goal of the evaluation is to investigate the usability of the MENTAI through the followings: (1) comparing the amount of code when using this tool with the conventional application description method; (2) demonstrating that the application developed by this tool works on a system consisted of the CGRA and the host processor.

Table I shows the application, the data amount, the number of DFG nodes, and the number of contexts used for the evaluation. alpha is 24-bit alpha blender. gray is 24-bit grayscale. sf is 8-bit sepia filter. sepia is 24-bit sepia filter. The calculated data volume of sepia consists of two contexts, and the number of nodes of DFG is shown as the total value.

TABLE I
Applications

App	Data	# op ops.	Context
alpha	248	30	1
gray	192	20	1
sf	192	20	1
sepia	84	50	2

A. Code Amount

We compared the amount of code for writing the application shown in Table I with Black Diamond and with the MENTAI. Fig. 3 provides the comparison result in the amount of code. On average, it is reduced by 87.2 %.

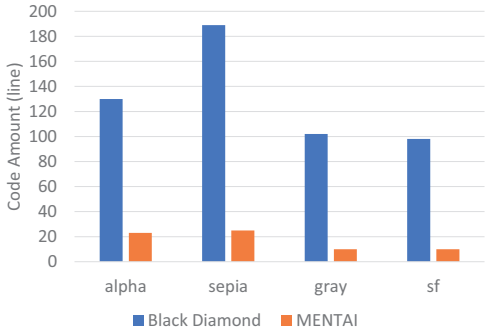

Fig. 3. Code amount

B. Case Study: CCSTOB2+GeyserTT

An evaluation was performed on a building-block type stacking system: a chip stack with CCSOTB2 and GeyserTT [15]. GeyserTT is an embedded processor that operates as a host with a MIPS R3000 compatible CPU core. The CPU core has an 8-KB 2-way set-associative cache for data and instructions and a 16-entry TLB. Data transfer between the stacked chips is performed via a ThruChip Interface (TCI) [16]. TCI is a technique of transferring data by inductive coupling between coils of stacked chips and transferring digital signals as a change in the magnetic field of the coils. All of the above architectures were design with Verilog HDL and synthesized by Synopsys Design Compiler using Renesas SOTB 65nm. They were placed and routed with Synopsys IC Compiler. The evaluation was done by simulation. We use Cadence NC-Verilog for HDL simulation and Synopsys Prime Time for power simulation. All systems operate at 20MHz. The supply voltage is 0.55V for CCSOTB2 and 0.75V for GeyserTT.

B.1. Execution Time

We compare the execution times of the stacking system and the host processor alone. Fig. 4 provides the comparison result in the execution time. The execution time was evaluated using a cycle-accurate simulator for stacking systems based on VMIPS [17]. Here, memory access is pipelined, and its latency is eight cycles for this evaluation. Data transfer between GeyserTT and CCSOTB2 is treated in the same way as the general memory access via the inter-chip routers. On average, the execution time of GeyserTT alone is shorter by 0.6 %. This is due to the long time required to transfer the configuration data and calculation data.

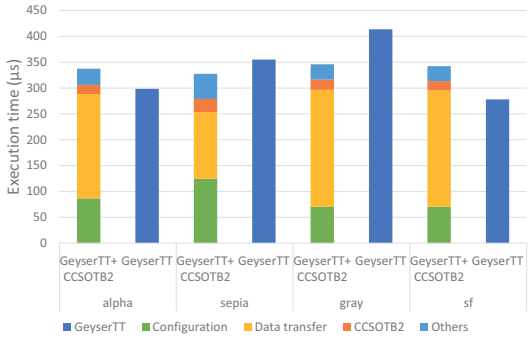

Fig. 4. Comparison of execution time between GeyserTT+CCSOTB2 and GeyserTT

B.2. Energy Efficiency

We compare the energy efficiency of the stacking system and the host processor alone. Fig. 5 provides the comparison result of the energy consumption. The numbers above the bars represent the amount of energy consumed by TCI. Thanks to the integrated development flow of MENTAI, data transfer time and execution time of CCSOTB2 can be approximately estimated. Thus, the GeyserTT sleeps during the estimated time, and clock signal for the data and instruction caches are gated. The power consumption of GeyserTT and CCSOTB2 is obtained by post-layout simulation, as shown in Table II. For CCSOTB2, both the power consumption during the run time and the idle time are shown in the table. The power consumption of the TCI was calculated using the value reported by [18]. On average, the energy of GeyserTT+CCSOTB2 is smaller by 63.8 %.

VI. Conclusion

We implemented an application development tool for the entire system consisting of the CGRA and a host processor, called MENTAI. Unlike existing CGRA-related tools, MENTAI is not for design exploration, but software development on various CGRAs. The development tools were implemented as four LLVM Passes. Thanks

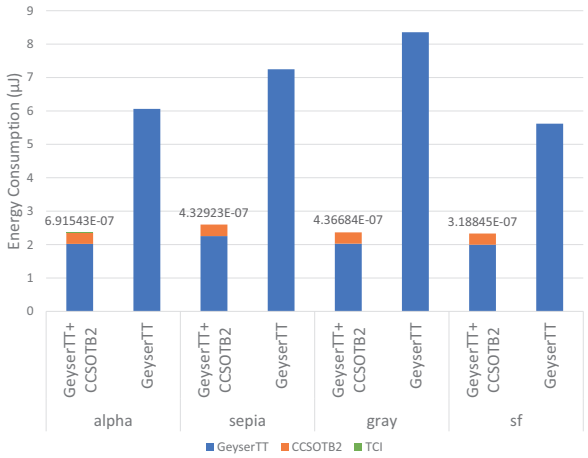

Fig. 5. Comparison of energy consumption between GeyserTT+CCSOTB2 and GeyserTT

TABLE II
Power consumption

Power consumption (mW)				
App	alpha	sepia	gray	sf
GeyserTT	20.3	20.4	20.2	20.2
GeyserTT (gated)	4.51			
CCSOTB2 (active)	2.66	2.86	2.24	2.26
CCSOTB2 (idle)	0.897			

to MENTAI, the DFG and the assembly code of the controller to perform memory access control was automatically generated from the kernel written in C language. Four applications were evaluated for a CGRA and host processor chip-stack system. MENTAI achieves around 60 % energy reduction while keeping the execution time.

Acknowledgment

This research was conducted through the University of Tokyo Large-scale Integrated System Design Education Center. This was done in cooperation with Synopsys Corporation. I would like to thank all those involved.

References

[1] B. Mei, F.-J. Veredas, and B. Masschelein, "Mapping an H. 264/AVC decoder onto the ADRES reconfigurable architecture," in *Field Programmable Logic and Applications, 2005. International Conference on*. IEEE, 2005, pp. 622–625.

[2] Xitian Fan, Huimin Li, Wei Cao, and Lingli Wang, "DT-CGRA: Dual-track coarse-grained reconfigurable architecture for stream applications," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.

[3] D. Lee, M. Jo, K. Han, and K. Choi, "FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability," in *2009 International Conference on Field-Programmable Technology*, 2009, pp. 376–379.

[4] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: A unified framework for CGRA modelling and exploration," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2017, pp. 184–189.

[5] M. Mukherjee, A. Fell, and A. Guha, "DFGenTool: A Dataflow Graph Generation Tool for Coarse Grain Reconfigurable Architectures," in *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, Jan 2017, pp. 67–72.

[6] V. TUNBUNHENG and H. AMANO, "A retargetable compiler based on graph representation for dynamically reconfigurable processor arrays," *IEICE Transactions on Information and Systems*, vol. E91.D, no. 11, pp. 2655–2665, 2008.

[7] S. Dave and A. Shrivastava, "Ccf: A cgra compilation framework," *https://github.com/MPSLab-ASU/ccf*.

[8] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, March 2004, pp. 75–86.

[9] Kojima, Takuya and Ando, Naoki and Matshushita, Yusuke and Okuhara, Hayate and Doan, Ng. Anh Vu and Amano, Hideharu, "Real Chip Evaluation of a Low Power CGRA with Optimized Application Mapping," in *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, ser. HEART 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3241793.3241806

[10] J. Weng, S. Liu, V. Dadu, and T. Nowatzki, "Daegen: A modular compiler for exploring decoupled spatial accelerators," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 161–165, July 2019.

[11] T. Toi, N. Nakamura, T. Fujii, T. Kitaoka, K. Togawa, K. Furuta, and T. Awashima, "Optimizing time and space multiplexed computation in a dynamically reconfigurable processor," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 106–111.

[12] T. Kojima, N. A. V. Doan, and H. Amano, "GenMap: A Genetic Algorithmic Approach for Optimizing Spatial Mapping of Coarse-Grained Reconfigurable Architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–14, 2020.

[13] N. Ozaki, Y. Yasuda, M. Izawa, Y. Saito, D. Ikebuchi, H. Amano, H. Nakamura, K. Usami, M. Namiki, and M. Kondo, "Cool Mega-Arrays: Ultralow-Power Reconfigurable Accelerator Chips," *IEEE Micro*, vol. 31, no. 6, pp. 6–18, Nov 2011.

[14] J. Ellson, "Graphviz-graph visualization software," *http://www. graphviz. org/*, 2008.

[15] K. Usami, S. Kogure, Y. Yoshida, R. Magasaki, and H. Amano, "Level-shifter free approach for multi-vdd sotb employing adaptive vt modulation for pmosfet," in *2017 IEEE SOI-3D-Subthreshold Microelectronics Unified Conference, S3S 2017*, vol. 2018-March. Institute of Electrical and Electronics Engineers Inc., Mar. 2018, pp. 1–3, 2017 IEEE SOI-3D-Subthreshold Microelectronics Unified Conference, S3S 2017 ; Conference date: 16-10-2017 Through 18-10-2017.

[16] T. Kuroda, "ThruChip interface (TCI) for 3D networks on chip," in *2011 IEEE/IFIP 19th International Conference on VLSI and System-on-Chip*, Oct 2011, pp. 238–241.

[17] B. Gaeke, "The VMIPS Project, Version 1.5.1," http://vmips.sourceforge.net/vmips/, 2019.

[18] N. Miura, H. Ishikuro, T. Sakurai, and T. Kuroda, "A 0.14pJ/b Inductive-Coupling Inter-Chip Data Transceiver with Digitally-Controlled Precise Pulse Shaping," in *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, Feb 2007, pp. 358–608.