ASPE: an Abstraction Framework using ALU Arrays for Scalable Multiple FPGAs System

Kenta Inakagata
Dept. of ICS,
Keio University,
Yokohama JAPAN 223-8522,
cfd@am.ics.keio.ac.jp

Takayuki Akamine
Dept. of ICS,
Keio University,
Yokohama JAPAN 223-8522,
cfd@am.ics.keio.ac.jp

Hirokazu Morishita
Dept. of ICS,
Keio University,
Yokohama JAPAN 223-8522,
cfd@am.ics.keio.ac.jp

Yasunori Osana
Dept. of EEE,
Ryukyu University,
Nishihara JAPAN 903-0213,
cfd@am.ics.keio.ac.jp

Naoyuki Fujita
ARD Japan Aerospace
Exploration Agency,
Chofu JAPAN 182-8522,
cfd@am.ics.keio.ac.jp

Hideharu Amano
Dept. of ICS,
Keio University,
Yokohama JAPAN 223-8522,
cfd@am.ics.keio.ac.jp

Abstract— Multi-FPGA systems have attracted attentions as cost-efficient accelerators for high performance scientific computation. The major problem of such systems for users is programmability. It is difficult especially for Multi-FPGA systems to find the best structure considering the resource and communication capability with HDL-based design.

Here, ASPE, a design framework using arrays of processing elements on FPGAs is proposed to address the problem. Instead of HDL-coding, ASPE makes the application executed by defining operations and communication in the ALU arrays on multiple FPGAs.

MUSCL, the core program in the computational fluid dynamics is implemented on the ASPE as an example, and evaluation results show that about 4.1 times performance compared with software on Intel Core 2 Duo is achieved.

I. Introduction

Scientific computing is widely utilized in various fields including physics, biology and finance. In most cases, performance cannot be satisfied by a general-purpose processor because of a large number of floating point calculations.

A lot of accelerators using diverse hardware devices have been developed. For example, GPGPUs (Genaral-Purpose Computing on GPU) and dedicated hardware like GRAPE(GRAvity piPE)[1] have been achieved dramatic performance. However, the former cannot always resolve problems caused by complicated memory access, and the latter cannot be developed without paying vast amount of cost.

An FPGA (Field-Programmable Gate Array) has been taken notice as an accelerator because of its flexibility and cost effeciency. FPGAs were regarded as unsuitable devices for scientific computing due to their insufficient resources. However, as a rapid growth of FPGA technology [2][3], the reconfigurable devices have been introduced into such a large scale computation [4][5], and realized some successful examples are reported[6]. Besides, in order to obtain more computational capability, multi-FPGA systems have been studied. BEE3 (Berkeley Emulation Engine) and CUBE as a large-scale multiple FPGA platform are good examples[7][8].

The problem is FPGA is difficult to program. In most scientific computation with FPGAs, a lot of computational units are connected with a data flow of target application. Additionally, HDL(Hardware Description Language) are used for FPGA programming. The distribution of logic into multi-FPGAs system will increase the designers' burden.

In order to address this problem, building an array of programmable processing elements on FPGAs is a hopeful approach. SIMD array[9] and systoric array[10] are practical exmaples. However, they are limited to specific algorithms which can be also executed efficiently in GPU or other accelerators. A coarse-grain architecture[11] is one of approaches, which runs at higher speed and less energy consumption. but didn't forcus on high-performance computing on multiple FPGA platforms. Floating point ALUs are also introduced into an FPGA[12], but it did not treat multi-FPGA systems.

Here, we propose a design model called ASPE (ALU array based Stream Processing for multiple FPGA Environment) which constructs of computational circuits by ready-made ALU array on FPGAs. ASPE contributes following three : (1) the user can port an application just by setting the operation of ALU and their interconnection attached with the data. (2) The user don't have to mind the partitioning of multiple FPGAs. For that, ALUs

are prepared on multiple FPGAs in advance. (3) Unlike other processing arrays on FPGAs, the program by using tokens makes configuration of FPGAs flexible. Following the above concept, we designed ASPE framework. Then, we evaluated the framework using MUSCL algorithm in UPACS, which is CFD package software.

The rest of this paper is organized as follows. Section II. introduces the design concept of ASPE. Section **??** shows the FLOPS-2D, which is target platform. Section III. illustrates the detail of implementation. Section IV. shows the evaluation results and Section V. explains conclusion and future work.

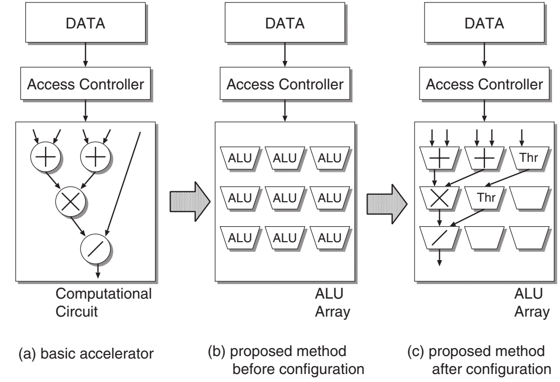


Fig. 1. Existing Accelerator and Proposed Method

II. Design of ASPE

In most cases, an accelerator for stream-processing consists of an access controller and a computational circuit as shown in Fig.1(a). It is difficult for researchers to construct these circuits since optimizing circuits structure is required with HDL.

In this study, ASPE shown in Fig.2, a stream-processing machine for single precision floating point computation by a ready-made ALU array is proposed. In ASPE framework, users can use multi-FPGAs system according to the following design flow.

Step 1. Optimize inputs and outputs for the target application.

Step 2. Set the configuration data such as the number of FPGA chips.

Step 3. Set the operations of ALUs in each FPGA chips.

Step 4. Using API, send data from host CPUs

Step 5. After calculation, extract result data from FPGA using API.

In this paper, we focus on the mechanism for Step 2. and Step 3. in this flow. APIs are now under development.

First, overview of ASPE is described in this section. Configuration method is then introduced and finally modules and operations of ASPE are shown.

A. ASPE Overview

ASPE consists of IFPGAs (FPGAs for input), CFPGAs (FPGAs for Computation) and OFPGAs (FPGAs for output) as shown in Fig.2. The left side of the figure shows 2D-mesh connected FPGAs which perform computation by sending data from upper FPGAs to lower ones, while the right side shows the architecture of the CFPGA. IFPGAs manage the data access, thus, their design depends on applications. Also, OFPGAs store or send calculation results to the host CPU. One or more number of host CPUs are connected to the ASPE, and the calculation data and operations for ALU are sent from them.

Calculation starts along with the following steps.

1. Operations are sent from the host machine to each ALU.
2. After configuration of all ALUs, `configuration completion signals`, which informs the host CPU of the end of FPGAs configuration, are sent.
3. Calculation data from the host CPU are stored in BlockRAMs of each FPGA or memory modules in each FPGA board.
4. After storing, data from IFPGA are sent, and calculated in CFPGAs.

After Step.2, CFPGAs can calculate valid data sent from the connected host CPU to CFPGAs directly, that is, an CFPGA does not store data in calculation. In the system, changing configuration of ALU during calculation is not supposed.

B. Transferring Operations

CFPGAs are formed in a uniform structures, and no information for identification such as address or ID is attached due to adding further FPGAs easily. Thus, the mechanism for transferring operations uses the location in the coordinate. Configuration data are transferred according to the following steps.

1. Assign the location of the target FPGA into the coordinate field in the header: Fig. 3(a).
2. If both x-field and y-field in the coordinate field are 0, transferred data is stored as its own: Fig. 3(d).
3. if y-location is 0, x-field in the coordinate field is decremented, and the data is transferred to the right FPGA: Fig. 3(b).
4. if x-field is 0, y-field is decremented, and the data is transferred to the lower FPGA: Fig. 3(c).

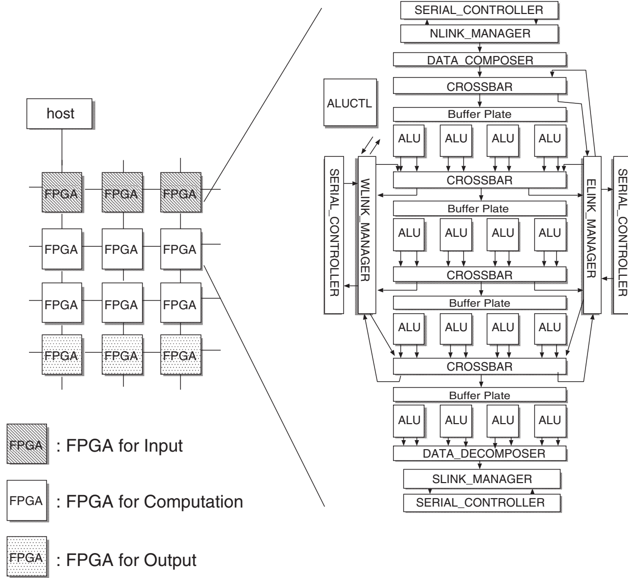
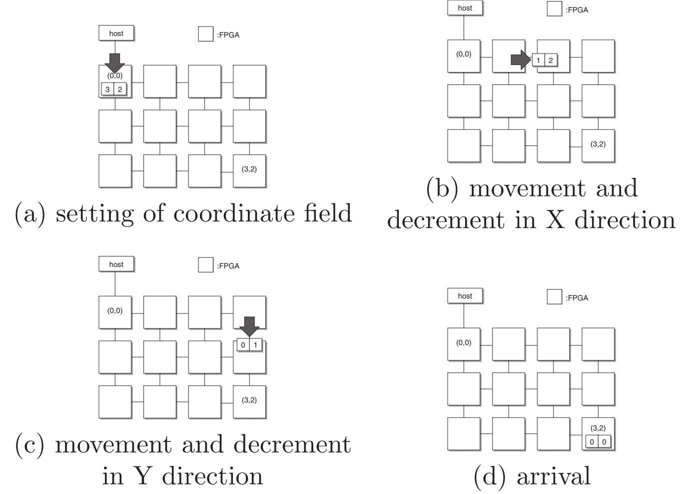


Fig. 2. The Diagram of ASPE



(a) setting of coordinate field
(b) movement and decrement in X direction
(c) movement and decrement in Y direction
(d) arrival

Fig. 3. Mechanism of Operation Transfer

This mechanism enables to transfer data to the target.

Next, the technique for sending `configuration completion signal` to the host CPU is explained. The host CPU starts to send calculation data after receiving the signals, so completion of configuration of all CFPGAs must be guaranteed at the time. Thus, the signal is propagated over each CFPGA by the following rules.

1. Even though a CFPGA receives its own configuration data, the `configuration completion signal` is sent after receiving `configuration completion signal` of right and lower CFPGAs.
2. `Configuration completion signal` from a CFPGA is sent to upper and left CFPGAs.

This mechanism configures $m \times n$ segment of CFPGAs from upper left most FPGA connected to the host CPU and doesn't influence $x > m \quad || \quad y > n$ segment of CFPGAs, Thus, configuration by multiple hosts is possible in the same way.

III. Implementation

Following to concept of ASPE, we should design the system for setting configuration of ALU, the modules for synchronizing the timing of calculation , and the connector of communication between FPGAs. Especially, we have taken care of the communication bandwidth between CFPGAs and the timing of calculation. Then, we have designed the architecture shown as Fig. 2. This system has 16 ALUs, some synchronizer, and some connector between an FPGA to another. Additionally, in order to connect further more FPGA to this system easily, we would not like to give some identification to FPGAs. Then, we have prepared unique configuration headers. Users can alter ALUs and a way to stream data by changing these headers. Here, modules in ASPE and the header style are explained.

A. Modules in ASPE

DATA_COMPOSER/DECOMPOSER

This module manages data transfer between `ALUs` and `SERIAL_CONTROLLERs`. Here, assume the system including 4x4 ALUs with two data inputs and outputs port. In this case, 8 sets of data are inputted or outputted simultaneously in total. Since the IEEE 754 single precision floating-point data are transferred through 32bits 4-lanes XGMII interface, it takes 2 clock cycles to send or receive 8 data. Then, `DATA_DECOMPOSER` sends data from `ALU` to `SERIAL_CONTROLLER` in two clock cycles. `DATA_COMPOSER` performs the reverse operation.

ALUCTL

This module delivers configuration data to another FPGA and stores the data from the host CPU appropriately. When configuration data are received, the header is checked as mentioned in Section.B.. If the data are for itself, configuration for `ALUs` and `CROSSBAR` in the data are stored in corresponding register. Otherwise, the coordinate field in the header is decremented properly. After finishing configuration of itself, right and lower CFPGA, the FPGA informs of the end of configuration by sending signal to `LINK_MANAGER`. This module doesn't influence calculation data.

{N, S}LINK_MANAGER

This module manages data transfer between upper/lower `SERIAL_CONTROLLER` and each inside module. Data

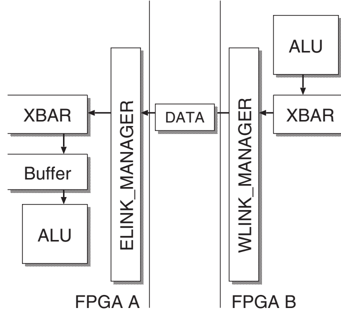


Fig. 4. Data Transfer during Calculation

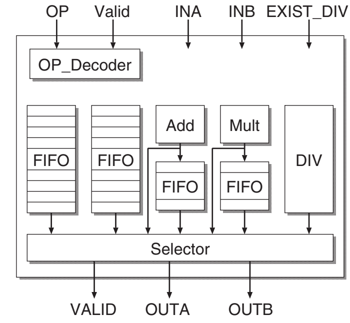


Fig. 5. The Architecture of ALU

are sent to ALUCTL during configuration, and also to DATA_COMPOSER and DATA_DECOMPOSER during calculation. The destination is switched according to the signal from ALUCTL.

{W, E}LINK_MANAGER

This module manages data transfer between right/left SERIAL_CONTROLLER and each inside module. The data are sent to ALUCTL during configuration and to CROSSBAR during calculation for transfering computational data to right and left adjacent CFPGAs as shown in Fig.4. The destination is switched as well as {N, S}LINK_MANAGER.

The number of lanes of links limits the data transfer as following.

- The number of 32bits data transferred between each CROSSBAR and LINK_MANAGER is 4 in maximum.
- The number of 32bits data transferred between CFPGAs is 4 in maximum.

The first limitation is required due to XGMII performance. The second is required in case that all 4 data from one CROSSBAR are outputted through SERIAL_CONTROLLER. The arbitration of the data from a number of CROSSBARs is relegated in this module with configuration data from the host CPU.

CROSSBAR

The module sends data to ALUs adequately. Here, CROSSBAR has 16 inputs. Right and left SERIAL_CONTROLLER provide four inputs respectively, and 8 inputs are provided by ALU or DATA_COMPOSER. 16 outputs are also provided by the module.

BUFFER_PLATE

This module synchronizes the timing of sending data to ALUs when data transfer such as Fig. 4 is needed. The module consists of eight FIFOs and inputs of those FIFOs are connected with the output of CROSSBAR. For synchronization, valid signal included in calculation data and data from SERIAL_LINK are utilized.

ALU

Here, the detail of ALU is described. There are 4×4 ALUs in each CFPGA, and each ALU supports adder, subtractor, multiplier, divider and through operation. Arithmetic units in ALU are provided by Xilinx Core Generator. According to performances of the arithmetic units, the number of pipeline stages of divider must be 24, and that of adder and multiplier is 10.

Outputs of ALUs on the same row must be synchronized for stream-processing. However, the ALUs including 10-stages pipelined dividers decreases the whole system performance. Also, we would like not to increase clock in vain. Then, FIFO and EXIST_DIV signal synchronizes the time for calculation. EXIST_DIV signal indicates whether division exists on the same row. If division exists, calculation requires 24 clock cycles. Otherwise it takes 10 clock cycles as shown in Fig. 5. Each ALU has two output ports, one of which is used for outputting calculation result, and other is assigned for outputting inputted data directly.

B. Header Data for Configuration of CFPGA

Operations for CFPGAs are explained here. Header data of configuration for the same CFPGA are sent in 3 steps continuously limited by the number of required configuration data and bandwidth of serial link. The first header includes the destination.

The structure of header is shown in Fig. 6. The numbers in parenthesis mean bitwidth of the field. Bitwidth of each lane is 32 with XGMII. Fields which have not mentioned previously are explained as below.

- exist_recv(exist_send)
  It indicates whether there are data received (sent) from (to) right-and-left adjacent CFPGA on each row.
- x_max, (y_max)
  They mean whether the destination is the right(or lower)-edge FPGA. Additionally, these signals are substitutes for signals from right(or lower) FPGA which informs of the end of configuration.

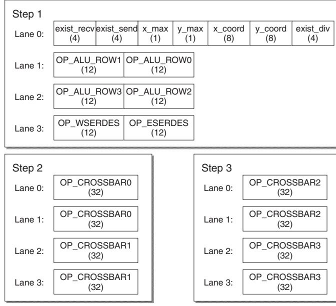


Fig. 6. Configuration Header for CFPGA

- x_coord, y_coord
  They are coodinate fields mentioned in Section B.
- OP_ALU_ROW0 - 3
  They mean operations for each `ALU` in CFPGA.
- OP_WSERDES, OP_ESERDES
  They inform which inputs should be outputted for `LINK_MANAGER`.
- CROSSBAR
  They mean operations for each `CROSSBAR` in CFPGA.

IV. Evaluation

Here, we evaluate ASPE by means of resource usage and performance. The performance is estimated by using an implementation example of MUSCL, an algorithm in CFD(Computational Fluid Dynamics).

A. Resource Usage

Resource usage of each module in CFPGA is shown in Table I. We used Xilinx ISE-12.2 in order to measure resources. Each `ALU` includes four DSP48 primitives. 12 BlockRAMs in total are utilized, an `ALU` has four and a `BUFFER_PLATE` has 8 BlockRAMs. The resource consumption of `SERIAL_CONTROLLER` for four lanes is estimated from the consumption of 2-lanes controller[13]. It is found that 64 BlockRAMs are used from the estimated result.

Fig. II shows the total resource usage of all modules except `SERIAL_CONTROLLER` in case of 4 by 4 ALU array. Since the number of slices of XC4VLX100 on FLOPS-board is 49,152, the usage is 74.4%. By adding `SERIAL_CONTROLLER`'s resource estimated, the total used slices become up to about 90.7%. Though this result means that there are about 10% available slices, it's reported that frequency of design tends to decrease as increasing resource usage[?]. Then, adding further modules may decrease performance.

B. Performance

Performance of CFPGA is evaluated by implementing MUSCL on ASPE. First, MUSCL algorithm is described.

TABLE I
Resource Usage of Each Module

Module	Slices	FlipFlops	LUTs
`ALUCTL`	382	613	510
`ALU`	1,472	1,617	1,864
`DATA_COMPOSER`	131	260	131
`DATA_DECOMPOSER`	163	307	134
`BUFFER_PLATE`	316	336	346
{`N, S`}`LINK_MANAGER`	266	4	396
{`W, E`}`LINK_MANAGER`	959	5	1,910
`CROSSBAR`	2,112	4,224	4,224
`SERIAL_CONTROLLER`	7,996	7,950	8,190

TABLE II
Total Resource Usage of CPFGA

Slices	FlipFlops	LUTs	BlockRAMs	DSPs
36,580	31,290	53,950	96	64

The elapsed time is then compared with that on Core 2 Duo and a dedicated computational circuits.

B..1 MUSCL

MUSCL (Monotone Upstream-centered Schemes for Conservation Laws) is a method to improve spacial accuracy. In CFD, the target space is divided into mesh. Equations are solved at each grid of the mesh. MUSCL extrapolates contact surface values from cell center values as shown in equations (1) to (4).

$$q'_{i+1/2} = \frac{q_{i+1} - q_i}{\Delta_{i+1} + \Delta_i}, \tag{1}$$

$$q'_{i-1/2} = \frac{q_i - q_{i-1}}{\Delta_i + \Delta_{i-1}}, \tag{2}$$

$$q_{i\pm 1/2} \cong q_i \pm \psi(r)\Delta_i q'_{i-1/2}, \tag{3}$$

$$r = \frac{q'_{i+1/2}}{q'_{i-1/2}}, \tag{4}$$

where q represents five physical values consisting of veracity, pressure and temperature. i means the direction in the mesh, and Δ_i is the distance between the cell center and the contact surface. $\psi_(r)$ is a limiter function to suppress the divergence. There are various limiter functions, and Van Albada limiter as the equation (5) was used here.

$$\psi(r) = (r^2 + r)/(r^2 + 1) \tag{5}$$

Then, with above known parameters, the physical values of the neighboring grid $q_{i\pm 1/2}$ are computed.

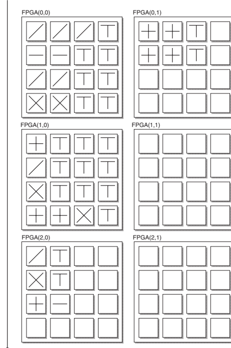


Fig. 7. MUSCL on CFPGAs

B..2 Comparison with Execution on Software and an HDL described system

The flow of MUSCL on CFPGAs is shown in Fig. 7, and when the calculation is done along this flow, it takes 166 clock cycles over arithmetic units. The number of clock cycles consumed in `SERIAL_CONTROLLER` is 33 and communication delay over wire is about 181nsec. Operation frequency measured by TimingAnalyzer is 78 MHz, so it takes $(166+2\times33\times2)/78\text{MHz}+182\text{nsec}\times2$ cycles to finish a computation. Then, in the ideal case that data is provided every clock cycle, it is found that calculation with ASPE is approximately 4.1 times faster than the software execution which runs on Intel Core 2 Duo(2.4 GHz). Source code is compiled by g95 FORTRAN compiler -O2 option.

Next, comparison with HDL described dedicated design is considered. The dedicated hardware solver for MUSCL was designed also for FLOPS-2D. From the number of arithmetic units, the dedicated circuits, and an access controller can be implemented on a single XC4VLX100. The dedicated solver of 8,000 times computations runs at about a half of ASPE's execution time. The performance decrease because of I/O bandwidth for CFPGAs, not communication delay between multiple FPGAs.

V. Conclusion

ASPE is a design framework using an ALU array for easy development of accelerators of stream processing on a platform with multiple-FPGAs. It mitigates the burden for designers of accelerators including HDL-coding and circuits optimization by using programmable ALU array. Evaluation by implementing MUSCL reveals that about 4.1-fold acceleration is expected compared with software execution.

As future work, in order to make ASPE more familiar to users, a programming environment with library for IFPGA should be developed. Additionally, efficient method to utilize unused ALUs should be considered.

Acknowledgments

This work is supported in part by Grants-in-Aid for Scientific Research Japan (200061). The authors also thank to VLSI Design and Education Centor Japan (VDEC) for suppoting simulation tools.

References

[1] Junichiro Makino, et al. GRAPE-DR:2-Pflops Massively-Parallel Computer with 512-Core, 512-GFlops Processor Chips for Scietific Computing. *Supercomputing*, 2007.

[2] Keith Underwood. FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. *Proc. of the International Symposium on Field-Programmable Gate Arrays*, pages 171–180, 2004.

[3] Jian Liang, et al. Floating Point Unit Generation and Evaluation for FPGAs. *Proc. of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

[4] T.El-Ghazawi, et al. The Promise of High-Performance Reconfigurable Computing. *IEEE Computer*, Feb 2008.

[5] M.C.Herbordt, et al. Achieving High Performance with FPGA-Based Computing. *IEEE Computer*, Mar 2007.

[6] Gerald R.Morris, et al. An FPGA-Based Floating-Point Jacobi Iterative Solver. *Proc. of the International Symposium on Parallel Architectures, Algorithms and Networks*, 2005.

[7] S.Kestur, et al. Accelerating the Nonuniform Fast Fourier Transform using FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2010.

[8] O.Mencer, et al. CUBE: A 512-FPGA CLUSTER. *Southern Programmable Logic Conference*, 2009.

[9] Kentaro Sano, et al. Domain-Specific Programmable Design of Scalable Streaming-Array for Power-Efficient Stencil Computation. In *International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2011.

[10] Kentaro Sano, et al. Systolic Architecture for Computational Fluid Dynamics on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007.

[11] C.Brunelli, et al. A coarse-grain reconfigurable architecture for multimedia applications supporting subword and floating-point calclulations. *Journal of Systems Architecture: the EUROMICRO journal, Vol.56, Issue 1*, Jan. 2010.

[12] C.-W. Yu, et al. Optimizing Coarse-Grained Units in Floating Point Hybrid FPGA. *IEEE International Conference on Field Programmable Technology*, Dec. 2008.

[13] et al. Hirokazu Morishita. Exploiting Memory Hierarchy for a Computational Fluid Dynamics Accelerator on FPGAs. *International Conference on Field Programmable Technology*, Dec 2008.

[14] e. a. Hirokazu Morishita, "Implementation and evaluation of an arithmetic pipeline on FLOPS-2D: multi-FPGA system," *ACM SIGARCH Computer Architecture News*, vol. 38, pp. pp.8–13, Sep 2010.