PyCoRAM: Yet Another Implementation of CoRAM Memory Architecture for Modern FPGA-based Computing

Shinya Takamaeda-Yamazaki
Tokyo Institute of Technology
JSPS Research Fellow
Tokyo, Japan 152-8552
takamaeda@arch.cs.titech.ac.jp

Kenji Kise
Tokyo Institute of Technology
Tokyo, Japan 152-8552
kise@cs.titech.ac.jp

James C. Hoe
Carnegie Mellon University
Pittsburgh, PA 15213
jhoe@ece.cmu.edu

ABSTRACT

Development of FPGA system is very complicated because designers should manage sea of logics and bare-metal components on FPGAs. CoRAM memory architecture is an endeavor to simplify the FPGA accelerator development. Its soft-logic implementation for existing FPGAs gives an abstract of on-chip/off-chip memory and on-chip interconnections. This paper presents PyCoRAM which is yet another implementation of CoRAM memory architecture, for modern integrated development environments (IDE) provided by the FPGA vendors. PyCoRAM supports AMBA AXI4, a major commercial interconnect architecture, as the on-chip interconnection. PyCoRAM synthesizes an application IP-core from a user-defined application RTL designs with abstracted CoRAM memory components and software threads written in Python. Then the designer can easily use the synthesized accelerator as a typical IP-core, as same as the other IP-cores such like soft-core CPU. We evaluated the current implementation of PyCoRAM on two actual FPGA boards. The evaluation result shows that PyCoRAM can sufficiently utilize the memory bandwidth by representation of the memory access pattern in the minimal software model of Python. This paper is for Category 1 (new unpublished manuscript).

1. INTRODUCTION

With the growing demands for higher energy efficiency of computing, computer architects have to explore a sophisticated approach employing heterogeneous computing resources, such as GPGPU, FPGA and ASIC[1]. FPGA-based custom computing is a hopeful way for both high performance and energy efficiency. In FPGA accelerator development, system designers have to implement not only the detailed computing logic, but also a time-consuming and error-prone complicated control logic to manage the data marshaling among the on-chip components and the off-chip memory.

CoRAM[2] (Connected RAM) memory architecture is an endeavor to simplify the FPGA accelerator development flow. Its soft-logic implementation[3] on existing FPGAs gives an abstract of on-chip/off-chip memory components and on-chip interconnections. In CoRAM, the application designer can represent a complicated memory access pattern by using the C-like software model. It simplifies the development with keeping the accelerator performance.

Aside from this, FPGA vendors provide integrated development environments (IDE), such as Xilinx Platform Studio[4] and Altera Qsys[5], for IP-core-based system development. In the IDEs, most hardware components, including off-chip memory interface and I/O interface, are handled as abstract function units connected to the specified interconnection. It helps developers to build a large system, such as FPGA-based SoC.

To integrate CoRAM memory abstraction into modern IDE-based development flow, we developed PyCoRAM, which is yet another implementation of CoRAM memory architecture for modern IDEs provided by the FPGA vendors. Certainly, PyCoRAM also gives users the memory abstraction to simplify the accelerator development, as same as the original CoRAM. Unlike the original CoRAM, PyCoRAM supports AMBA AXI4[6, 7] as FPGA on-chip interconnect way. PyCoRAM tool-chain automatically generates an AXI4 IP-core module of the accelerator from the user-defined application RTL design and a software program of memory access patterns. As the name suggests, PyCoRAM uses Python as the programming language to model the memory access patterns.

In this paper, we present the architecture and microarchitecture of PyCoRAM, including the Python-to-Verilog high-level synthesis compiler to generate a hardware component to manage on-chip/off-chip data movements. As an initial evaluation, we implemented a very simple bandwidth intensive kernel on two kinds of actual FPGA boards. The evaluation result shows that the current PyCoRAM implementation can sufficiently utilize the memory bandwidth by straightforward representation of the memory access pattern in Python.

2. BACKGROUND

2.1 CoRAM Memory Architecture

CoRAM is a portable memory architecture for efficient FPGA-based computing. The programing view of CoRAM is simplified by the abstraction of memory fabrics, as illustrated in Figure.1. Data movement behavior among on-chip memory fabrics and off-chip memory components is represented in the software model. Application development concern is split into two parts; (1) computing kernels and (2) control threads. Computing kernels tightly connected to CoRAM

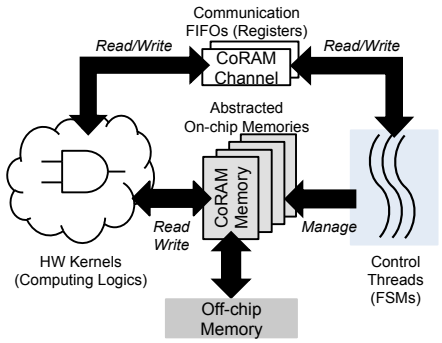

Figure 1: CoRAM Programming Model

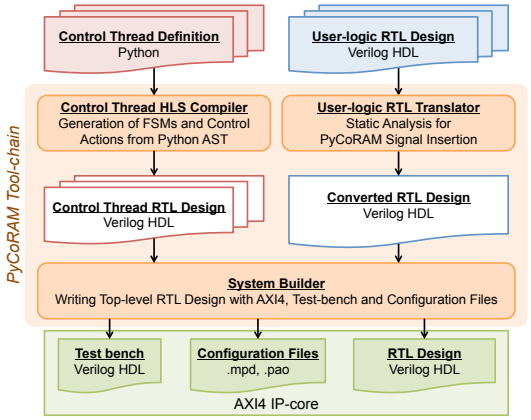

Figure 2: PyCoRAM Tool-chain

function units (CoRAM Memory, which is an abstract memory block, and CoRAM Channel, which is a communication channel between computing kernels and control threads). CoRAM abstract memory blocks are connected to off-chip memory components via on-chip interconnection. Control threads manage the on-chip data movements. Designers specify the behavior of a control thread by the representation of software model of C. This separation simplifies the development with keeping the accelerator performance. It also improves the portability of accelerator design for the other FPGA boards.

A CoRAM memory block has a standard interface as same as on-chip SRAMs of FPGAs (BRAM). The user-logic can be defined in a very simple structure, such like a computation pipeline and on-chip memory blocks supplying data for them. CoRAM channels are used for control between user-logic and control threads. For instance, the user-logic can know that data for the computation are ready on on-chip CoRAM memory block from control threads via the channels, and notify the completion of the computation to the control threads.

Originally, CoRAM ideology is a standardization of memory abstraction for future advanced FPGAs with hard-wired on-chip memory interconnections and sophisticated CPU-based data marshal units. In order to examine the CoRAM philosophy, a soft-logic implementation on existing FPGAs is proposed[3]. In the soft-logic implementation, a control thread is realized as a logic unit of FSM (Finite State Machine) using FPGA reconfigurable resources. The soft-logic CoRAM has an llvm-based HLS (high level synthesis) compiler to translate a software model of control thread into an RTL design with FSMs. As an interconnect system to the off-chip memory, the soft-logic CoRAM employs CONNECT[8], which is a well-tuned FPGA-oriented high throughput NoC (network on chip). Additionally, an optimization method called ShrinkWrap[9] is also proposed, which enhances the on-chip network performance and reduces the resource consumption.

2.2 Vendor-Provided IDE and On-chip Interconnection

FPGA vendors provide useful IDEs for IP-core-based system development. A hardware component is treated as an abstract function units connected to the specified on-chip interconnection. Most IDEs provide also many useful IP-cores as standard. System designers can effectively develop a FPGA system, with the provided standard IP-cores and/or soft-core CPUs. As an interconnection mechanism among hardware components on an FPGA, the IDEs support standardized on-chip interconnect architectures, such as AMBA AXI[6, 7]. Designers can easily expand the system by appending original IP-cores into the design, if the original IP-cores are designed for those standard interconnect architecture.

Additionally, these interconnect architectures supported by the vendor IDEs have large capability for accelerator development. For instance, AMBA AXI4, supported by Xilinx Platform Studio[4], has several useful functions to improve the system throughput and development efficiency; automatic data width conversion and frequency conversion between multiple interfaces, Out-of-Order transmission using interface IDs, and so on.

3. PYCORAM: YET ANOTHER IMPLEMENTATION OF CORAM

3.1 Overview

PyCoRAM is an orchestrated framework of CoRAM memory abstraction and a vendor-provided IDE with standard on-chip interconnect interface. PyCoRAM synthesizes an IP-core package from user-defined RTL designs of computing kernels and software description of memory access patterns.

PyCoRAM does not generate an RTL design of on-chip network. Instead, PyCoRAM utilizes a standard on-chip interconnection architecture supported by vender IDEs. In the current implementation, PyCoRAM supports AMBA AXI4 as the interconnection. As the IDE, Xilinx Platform Studio (XPS) is supported. At synthesis time of FPGA circuit image, XPS automatically synthesizes an AXI4 interconnection network and connects the IP-core instances.

Figure.2 illustrates the IP-core generation flow of PyCoRAM. PyCoRAM provides (1) a Python-to-Verilog high-level synthesis compiler (Control Thread HLS Compiler) to generate a hardware design to manage data movement among on-chip memory components and off-chip DRAMs, (2) an RTL design translator (User-logic Translator) to automatically insert CoRAM memory signals into the input user design, and (3) a coordinated RTL synthesizer (System Builder) to generate a top-level IP-core design with AXI4 interconnect bridges, a test-bench and some essential configuration files for the IDE tool.

The microarchitecture overview of a generated IP-core is illustrated in Figure.3. The hardware kernel that the designer modeled with CoRAM memory abstraction is located

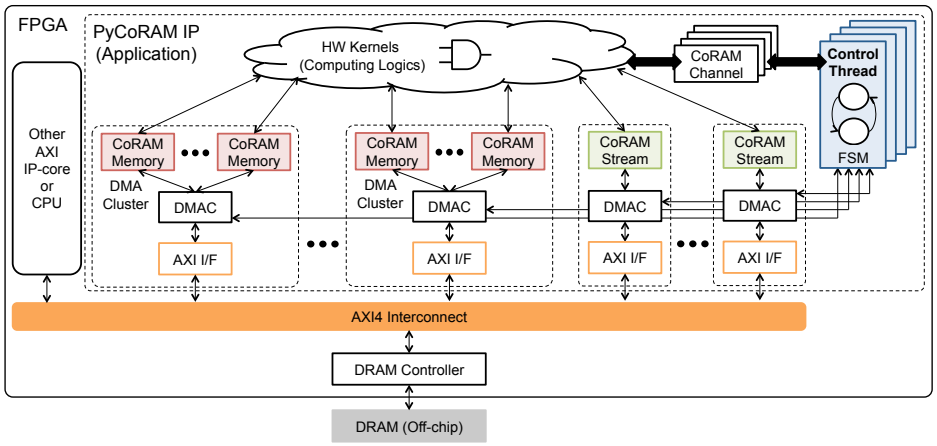

Figure 3: PyCoRAM Microarchitecture Overview

on the top of design. Abstract CoRAM memory objects in the user-defined design are replaced with a combination of physical CoRAM memory objects (Block RAM) and DMA controllers. Several CoRAM memory objects may work as a cluster within a single DMA controller. The owner DMA controller treats these CoRAM memory objects as a virtual single Block RAM, in order to support scatter/gather operation and multi-banked Block RAM. Additionally, PyCoRAM supports CoRAM stream interface connected with off-chip DRAM in standard. CoRAM stream interfaces can be accessed as FIFO from the user-defined logics.

Control threads, which represent the memory access pattern, are synthesized into FSM-based units by the HLS compiler. Control threads can send/receive tokens to/from the user-defined logics via CoRAM channels. CoRAM channel has a FIFO interface. The user-defined logics can be controlled from the control thread, and the control threads can use the dynamic value from the user-defined logics. Control thread units should manage the data on physical CoRAM memory objects, by using the DMA controllers. DMA controllers receive a request of data movement between on-chip Block RAM and off-chip DRAM. Then the DMA controllers send a transfer request to the off-chip DRAM interface via AXI interconnection.

AXI interface (AXI I/F) has a general port for handshaking with the DMA controller and an AXI port that treats a raw AXI4 protocol. Note that the data width of DMA controller equals to the width of CoRAM memory object. However, the data width of root AXI interconnection may be wider than the width of DMA controller. In this case, the IDE tool automatically inserts a data width converter between two interfaces that the different data width have.

With PyCoRAM supports, PyCoRAM accelerator IP-core and other IP-cores can share a same AXI4 interconnection in a system. System designers can develop a heterogeneous SoC with multiple IP-cores on a single FPGA, in direct way using the vendor IDE.

3.2 Python-to-Verilog Compiler for Control Thread

As described, every data movement between on-chip CoRAM memory (Block RAM) and off-chip DRAM is managed by control threads. In PyCoRAM, control threads' behavior is represented in Python, a major script programing language.

```
0  # PyCoRAM memory object and channel object
1  ram = CoramMemory(idx=0, datawidth=32, size=1024)
2  channel = CoramChannel(idx=0, datawidth=32)
3
4  ram_size = 128
5
6  # function definition
7  def array_sum(repeat_time):
8      addr = 0
9      sum = 0
10     for i in range(repeat_time):
11         # transfer from DRAM to CoRAM memory
12         ram.write(0, addr, ram_size)
13         # write a value to the channel
14         channel.write(addr)
15         # wait for a token from the user-logic
16         sum = channel.read()
17         addr += ram_size * 4
18     # display statement for Verilog simulation
19     print('sum=', sum)
20
21 # function call
22 array_sum(8)
```

Figure 4: PyCoRAM Control Thread

An actual control thread is implemented as a state-machine by using some FPGA hardware resources. We developed Python-to-Verilog HLS compiler to generate an RTL design with FSMs to control DMA controllers. Figure.4 shows an example of acceptable Python code of control thread. The example thread just transfers the data from an off-chip DRAM to on-chip CoRAM memory for several times.

A control thread in PyCoRAM consists of (1) instance creations of CoRAM components (memory, stream, channel and register), (2) method invocations of CoRAM instance, and (3) general value definitions and method definitions. In line 1 in Figure.4, CoRAM memory instance is defined for this control thread. In line 2, CoRAM channel instance is defined as a bridge between the control thread and the user-defined logic. To issue a DMA transfer for CoRAM memory object, read/write methods of them are called, as shown in line 12. As arguments of the method, a local address on the CoRAM memory, remote address of DRAM and transfer size (in word) are passed. User-defined logics and control threads can communicate with each other. User-defined logics accept their controls from control threads via CoRAM channels. Control threads can use dynamic values sent from the connected user-defined logics.

As well as general software programming, the basic control syntaxes (if/for/while statements and return/break/-cotninue statements) are available. Designers can easily put

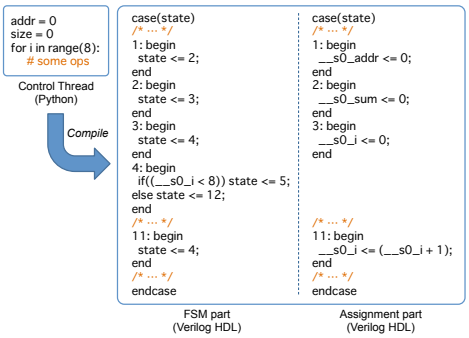

Figure 5: HSL Compilation Example

memory access patterns of the application in software manner.

The compiler creates a FSM (finite state machine) from a software model. The compiler obtains AST (Abstract Syntax Tree) by using the native code parser of Python. The AST is scanned by the general visitor pattern, and is converted to Verilog HDL. In the current implementation, each value assignment is converted basically into an identical state of FSM, as illustrated in Figure.5. Control syntaxes are realized as conditional transitions. The generated RTL contains two "always" statements: FSM definition and value assignment at each FSM state.

The current compiler implementation is still naive, but it includes a basic optimizer that simplifies arithmetic/logic expressions, replaces high-cost operators (multiplication and division) with equivalent low-cost operators, and converts single-assigned value into constant. In Python, variable definition should be done with a value assignment. Basically, a register ("reg" entry in Verilog HDL) is prepared for every variable, and the value of a register is assigned in the always statement. To handle constant value (like "parameter" in Verilog HDL) in control threads, the compiler optimizer automatically converts the definition of variables with just a single assignment into constant definitions of parameter in Verilog HDL.

The current compiler implementation has some limitations. (1) Only "integer" is supported as the data type. Object-oriented programming, such like class definition, is not supported. Array structures ("list", "dict", "tuple" and "set") are not supported. (2) Nested function definition is not supported. (3) Recursive call of methods is not supported.

3.3 User RTL Design Conversion

Hardware kernels are dominant logics of computing working with CoRAM objects. Therefore they should have an optimized computing pipeline implemented by traditional HDLs or well-tuned HLS environments. In the current implementation of PyCoRAM, only Verilog HDL is supported as the language of input hardware kernels. When HLS tools or other HDLs are used, RTL codes of hardware kernels should be translated into Verilog HDL in some way.

When a hardware kernel is modeled, designers can use stubs as CoRAM objects in their kernel code. Figure.6(a) shows an example instance creation of the CoRAM memory object. To identify the personality of CoRAM object, some parameters are essential; (1) thread name, (2) object id, (3) data width of the interface and (4) address length to express the capacity.

Table 1: Target FPGA Boards

Parameter	Digilent Atlys[10]	Xilinx ML605[11]
FPGA	Xilinx Spartan-6 LX45	Xilinx Virtex-6 LX240T
Logic Cells	43,661	241,152
BRAM Capacity	261 KB	1849 KB
DRAM	DDR2-800	DDR3-800
DRAM Bandwidth	1.2 GB/s	6.4 GB/s
DRAM Capacity	128 MB	512 MB
Clock Frequency	100 MHz	
Interconnection	AMBA AXI4	
Interconnection Type	Crossbar (Performance Optimized)	
Interconnection Width	128 bit	256 bit
Interconnection Frequency	100 MHz	200 MHz

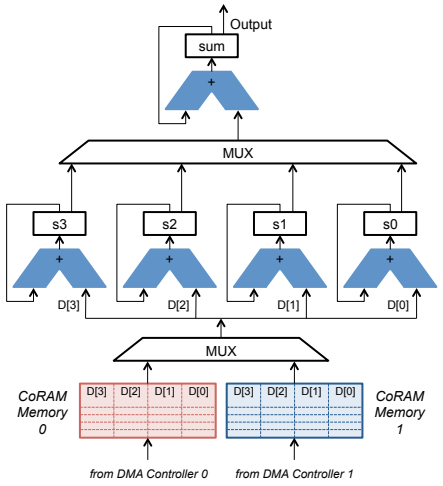

Figure 7: User-logic Design of Array Sum (# SIMD ways = 4)

In PyCoRAM compilation step, the user RTL translator, shown in Figure.2, analyzes the kernel code to replace the stubs. The translator automatically inserts additional signals of CoRAM objects into the design. As illustrated Figure.6(a), a "generate" statement can be used to represent a parameterized module/instance hierarchy, unlike the current soft-logic implementation of the original CoRAM. As shown in Figure.6(b), some CoRAM-related signals are inserted to the instance port. When a generate statement is used, some "if" statements are appended to switch the inserted signals. If a module with CoRAM objects is used twice or more, the module definition in the design is replicated to avoid a name confliction by additional signals.

In order to convert designs in this way, syntax parser and dataflow analysis of Verilog HDL are used. We used Pyverilog for design analysis in PyCoRAM, which is our original Verilog HDL design analysis tool-kit.

4. EVALUATION

4.1 Methodology

We evaluated the performance of generated IP-core design by PyCoRAM on two actual FPGA boards. We used Digilent Atlys (Xilinx Spartan-6 LX45) and Xilinx ML605 (Xilinx Virtex-6 LX240T). The configurations of target FPGA boards are listed in Table.1. We used Xilinx Platform Studio

```
generate for(i=0; i<2; i=i+1 ) begin: loop
 CoramMemory1P
  #(
   .CORAM_THREAD_NAME("thread_name"),
   .CORAM_ID(i),
   .CORAM_ADDR_LEN(W_A),
   .CORAM_DATA_WIDTH(W_D)
   )
 inst_memory_name
   (
   .CLK(CLK),
   .ADDR(mem_addr),
   .D(mem_d),
   .WE(mem_we),
   .Q(mem_q)
   );
end endgenerate
```

(a) Instantiation Declaration in Input RTL

```
generate for(i=0; i<2; i=i+1) begin: loop
 if((i == 0)) begin
  CoramMemory1P
   #(
    .CORAM_THREAD_NAME("thread_name"),
    .CORAM_ID(i),
    .CORAM_ADDR_LEN(W_A),
    .CORAM_DATA_WIDTH(W_D)
    )
  inst_memory_name
   (
    .CLK(CLK),
    .ADDR(mem_addr),
    .D(mem_d),
    .WE(mem_we),
    .Q(mem_q),
    .coram_addr(thread_name_corammemory_0_addr),
    .coram_d(thread_name_corammemory_0_d),
    .coram_we(thread_name_corammemory_0_we),
    .coram_q(thread_name_corammemory_0_q)
    );
end
else if((i == 1)) begin
  CoramMemory1P
   #(
    .CORAM_THREAD_NAME("thread_name"),
    .CORAM_ID(i),
    .CORAM_ADDR_LEN(W_A),
    .CORAM_DATA_WIDTH(W_D)
    )
  inst_memory_name
   (
    .CLK(CLK),
    .ADDR(mem_addr),
    .D(mem_d),
    .WE(mem_we),
    .Q(mem_q),
    .coram_addr(thread_name_corammemory_1_addr),
    .coram_d(thread_name_corammemory_1_d),
    .coram_we(thread_name_corammemory_1_we),
    .coram_q(thread_name_corammemory_1_q)
    );
 end
end endgenerate
```

(b) Instantiation Declaration in the Converted RTL

Figure 6: RTL Conversion: Instance Declaration in generate statement

```
DATA_SIZE = 1024 * 1024 * 128 # byte
RAM_SIZE = 1024 * 1 # entry
USED_RAM_SIZE = 1024 * 1 # entry
SIMD_SIZE = 128 # bit
REPEAT = ((DATA_SIZE / (USED_RAM_SIZE * (SIMD_SIZE / 8))) - 2) / 2

ram0 = CoramMemory(0, SIMD_SIZE, RAM_SIZE)
ram1 = CoramMemory(1, SIMD_SIZE, RAM_SIZE)
channel = CoramChannel(0, 64)

read_addr = 0
sum = 0

ram0.write_nonblocking(0, read_addr+0*USED_RAM_SIZE*(SIMD_SIZE/8), USED_RAM_SIZE)
ram0.wait()
channel.write(USED_RAM_SIZE)

ram1.write_nonblocking(0, read_addr+1*USED_RAM_SIZE*(SIMD_SIZE/8), USED_RAM_SIZE)
ram1.wait()
channel.write(USED_RAM_SIZE)

for i in range(REPEAT):
  read_addr += (USED_RAM_SIZE*(SIMD_SIZE/8)) * 2

  sum = channel.read()
  ram0.write_nonblocking(0, read_addr, USED_RAM_SIZE)
  ram0.wait()
  channel.write(USED_RAM_SIZE)

  sum = channel.read()
  ram1.write_nonblocking(0, read_addr+USED_RAM_SIZE*(SIMD_SIZE/8), USED_RAM_SIZE)
  ram1.wait()
  channel.write(USED_RAM_SIZE)

sum = channel.read()
sum = channel.read()
print('sum=', sum)
```

Figure 8: Control Thread for Array Sum

14.6 to synthesize a bit file and to generate an AXI interconnect.

In this study, we used array-sum illustrated in Figure.7, a tiny application to measure the memory bandwidth utilization. The array-sum calculates the summation value of an array. We used a parameterized design to vary the number of simultaneous computations at 1 cycle. The computing kernel uses two CoRAM memory instances as double-buffered. The data size for each computation is 32-bit. A data combination is stored in a same row on CoRAM memory. The total size of accessed data on the DRAM is 128MB. The control thread description in Python is very short, illustrated in Figure.8. For double-buffered, two CoRAM memory instances are defined. This control thread issues DMA requests to transfer the data from the DRAM to the on-chip CoRAM memories.

The operation frequency of computing kernels is 100 MHz in both FPGA boards. However, each AXI4 interconnection to off-chip DRAM interface is optimized for each memory bandwidth. For Altys board, the maximum memory bandwidth is 1.2 GB/s [1]. The data width of AXI4 interconnection is 128 bit, and the clock frequency is 100MHz. Therefore the interconnection bandwidth is 1.6 GB/s. For ML605, the maximum memory bandwidth is 6.4 GB/s. The data width is 256 bit, and the clock frequency is 200MHz. Therefore the interconnection bandwidth is 6.4 GB/s. The FPGA system is controlled by a host computer via RS-232C. The execution cycle count is measured by using a hardware counter, implemented as a user-logic on FPGA. The bandwidth utilization is calculated as below.

$$BWUtil = \frac{TotalDataSize[B] \times Freq[Hz]}{Cycle[Cycle] \times MaxBW[B/s]}$$

4.2 Memory Bandwidth Utilization

We observed the impact of the number of SIMD ways to the memory bandwidth utilization. Figure.9 shows the measured bandwidth utilization on Digilent Atlys. Figure.10 shows the measured bandwidth utilization on Xilinx ML605.

On Atlys board, 85.5% of the maximum memory bandwidth is utilized when the SIMD size is 16 bytes. When the SIMD size is 32 bytes, 86.3% of the bandwidth is utilized. In these cases, the required bandwidths are 1.6 GB/s and 3.2GB/s, respectively. They are greater than the maximum memory bandwidth. The bandwidth inefficiency of 14% thought to be due to manipulation overhead on the control thread, interconnection latency and DRAM latency. On ML605, 84.9% of the maximum memory bandwidth is utilized, when the SIMD size is 64 bytes. In this case, the required memory bandwidth is 6.4 GB/s, which equals the maximum memory bandwidth. As alike as Atlys, about 15% loss of bandwidth is observed.

The current PyCoRAM can achieve sufficiently memory performance with abstracted memory system and easy-to-use software-like model. However, a certain amount of performance inefficiency is observed. In the current implementation of PyCoRAM, the DMA controller cannot hide the memory access latency because the DMA controller handles up to only one memory transaction at a time. Due to the memory intensive kernel in this evaluation, the elapsed time of computation is smaller than the time of the data marshaling for each memory block, so that the memory latency impact directly affected the performance. In order to improve the memory performance, the DMA controller should have an advanced microarchitecture to handle multiple outstanding requests, but it will introduce some extra hardware overhead.

[1]Official maximum bandwidth is 1.6 GB/s. However, the operation frequency of the memory clock is degraded to 300MHz from 400MHz. Therefore the maximum bandwidth is limited up to 1.2 GB/s.

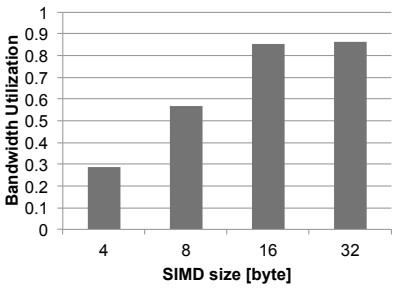

Figure 9: Bandwidth Utilization (Digilent Atlys)

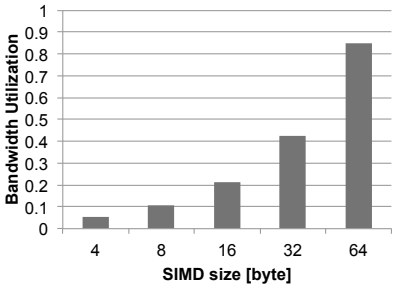

Figure 10: Bandwidth Utilization (Xilinx ML605)

4.3 Resource Utilization

In the viewpoint of resource utilization, the generated IP-core uses 50% of slices that whole the accelerator system used. The rest 50% is used by the AXI interconnect generated by the IDE and system functions for DRAM, clocking and reset. The AXI interconnect consumes 33% slices of whole the system. In this evaluation, it is synthesized under the performance optimized mode. Its resource utilization can be reduced by using area optimized mode.

The user-logic consumes 42% of slices that the IP-core totally used. The control thread utilizes 12% slices of whole the IP-core. Both the DMA controllers and the AXI interfaces consume 37% slices of whole the IP-core. The rest is used for glue logic to connect them. These results show that the hardware overhead is not small, because the application used for this evaluation is relatively small. If the more realistic and larger application is adopted, the ratio of hardware overhead will be diminished.

5. CONCLUSION

This paper presented PyCoRAM which is yet another implementation of CoRAM memory architecture for modern IDEs. We evaluated the current implementation of PyCoRAM on two actual FPGA boards. The evaluation result shows that PyCoRAM can sufficiently utilize the memory bandwidth under the abstraction of memory and memory access patterns in Python.

PyCoRAM is still under research and development. We have to evaluate the performance and the hardware overhead in more realistic applications. We already have implemented some memory intensive applications, such like matrix-matrix multiplication and stencil computation.

In order to establish a more sophisticated programming model for FPGAs, the programmability of PyCoRAM should be compared to the traditional HDLs and the other hardware modeling environments. Actually CoRAM and PyCoRAM can abstract complex memory systems by software-based modeling. We think, however, another easy way should be introduced to improve the programming efficiency to construct a well-tuned pipeline design.

Finally, we are planning to make our PyCoRAM toolkit public in the near future for efficient development of FPGA-based computing system.

6. ACKNOWLEDGMENT

This work is supported in part by Core Research for Evolutional Science and Technology (CREST), JST, Japan. We thank Michael K. Papamichael, Gabriel Weisz and Eric S. Chung for their help with the original CoRAM environment.

7. REFERENCES

[1] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society.

[2] Eric S. Chung, James C. Hoe, and Ken Mai. Coram: an in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 97–106, New York, NY, USA, 2011. ACM.

[3] Eric S. Chung, Michael K. Papamichael, Gabriel Weisz, James C. Hoe, and Ken Mai. Prototype and evaluation of the coram memory architecture for fpga-based computing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, pages 139–142, New York, NY, USA, 2012. ACM.

[4] Xilinx platform studio. http://www.xilinx.com/tools/xps.htm.

[5] Qsys - altera's system integration tool. http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html.

[6] Amba open specifications. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php.

[7] Ug761 axi reference guide - xilinx. http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[8] Michael K. Papamichael and James C. Hoe. Connect: re-examining conventional wisdom for designing nocs in the context of fpgas. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, pages 37–46, New York, NY, USA, 2012. ACM.

[9] E.S. Chung and M.K. Papamichael. Shrinkwrap: Compiler-enabled optimization and customization of soft memory interconnects. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 113–116, 2013.

[10] Atlys spartan-6 fpga development board. http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,836&Prod=ATLYS.

[11] Virtex-6 fpga ml605 evaluation kit. http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm.