Acceleration of the aggregation process in a Hall-thruster simulation using Intel FPGA SDK for OpenCL

Hiroyuki Noda
Keio University
3-14-1 Hiyoshi, Yokohama,
223-8522, Japan
noda@am.ics.keio.ac.jp

Ryotaro Sakai
Keio University
3-14-1 Hiyoshi, Yokohama,
223-8522, Japan
ryotaro@am.ics.keio.ac.jp

Takaaki Miyajima
Japan Aerospace Exploration
Agency (JAXA)
7-44-1 Jindaiji-higashi, Chofu,
182-8522, Japan
miyajima.takaaki@jaxa.jp

Naoyuki Fujita
Japan Aerospace Exploration
Agency (JAXA)
7-44-1 Jindaiji-higashi, Chofu,
182-8522, Japan
fujita@chofu.jaxa.jp

Hideharu Amano
Keio University
3-14-1 Hiyoshi, Yokohama,
223-8522, Japan
asap@am.ics.keio.ac.jp

ABSTRACT

The Full Particle-In-Cell (Full-PIC) method is a numerical simulation technique used in the research and development of Hall-thrusters which are a type of electric propulsion engines. It treats ions, neutrons, and electrons as particles and is highly accurate compared with other methods which treat them as a fluid. However, it requires a large computational cost. The Japan Aerospace Exploration Agency (JAXA) is developing a software package called NSRU-Full-PIC that implements such a method. One of the important computing tasks in NSRU-Full-PIC is the aggregation process, which causes Read-After-write (RAW) hazards, and hence makes parallel computation difficult.

In this paper, we tackle this problem by introducing a reduction operation with an FPGA accelerator. We use Intel's mid-range SoC, Arria 10 which embeds floating-point DSPs for high performance numerical computation. Intel FPGA SDK for OpenCL is available for this platform for easy offloading of complex tasks. We implemented 4 types reduction kernels and compared their performance. As a result, the aggregation process becomes 76.4 times faster than the single-thread version on an ARM Cortex-A9 1.5 GHz, and 14.1 times faster than that on a Xeon E5-2660 2.9 GHz in our fastest implementation, *Read-16-Vect*. In this implementation, we achieved 93.5% of theoretical performance with optimized FPGA resources.

1. INTRODUCTION

Electric propulsions such as Hall and ion thrusters are highly efficient in propulsion power compared with chemical propulsions. In such electric propulsions, Hall-thrusters superior to ion thrusters from the viewpoint of power efficiency in the low propulsion range and light weight because of its simple structure. That is, Hall-thrusters are considered to be an ideal propulsion system for satellite missions like station keeping, orbit transfer, or deep space exploration[1]. So, they are currently intensively studied and developed in various institutes and companies.

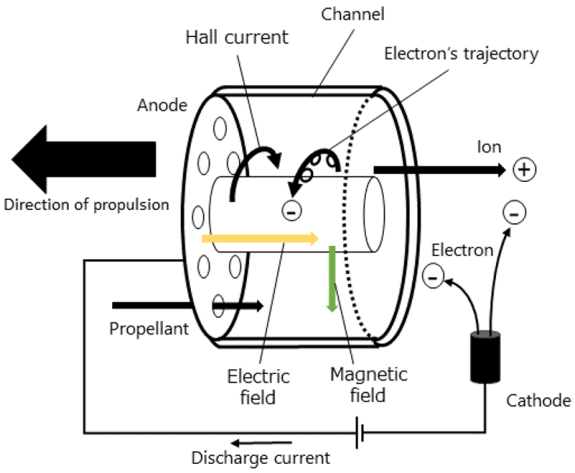


Figure 1: Operating principle of Hall-thrusters

Figure 1 shows an operating principle of Hall-thrusters. In an annular plasma accelerator called a channel, electrons emitted from the cathode are trapped and drift in the circumferential direction by applying a radial magnetic field (green line) and an axial electric field (yellow line). Electrons that move circumferentially generate the Hall current in a circumferential direction of the channel. Propellant that flow from a anode into the channel collides with electrons performing circumferential movement and turns into

HEART2017, June 7-9, 2017, Bochum, Germany ACM ISBN 978-1-4503-5316-8/17/06$15.00
https://doi.org/10.1145/3120895.3120915

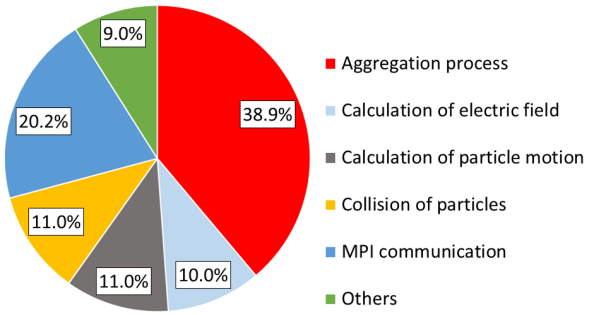


Figure 2: Profile of NSRU-Full-PIC on CRAY XE 6 with 128 processes

plasma. The Lorentz force generated by the Hall current and the radial magnetic fields hinders the movement of electrons in the plasma to cancel the electric fields in axial direction. As a result, the electric fields in the axial direction is maintained. By this, only ions in the plasma are accelerated and emitted as a beam outside the thruster. The thruster itself obtains thrust by a reaction force of the generated Lorentz force. The channel is kept a quasi-neutral state because the electrons flowing from the cathode are scattered toward the anode in the channel.

For development of Hall-thrusters, the numerical simulation is essential, since it is much more cost effective than real experiments. The Full-PIC method which is classified as the particle method that discretizes the motion of a continuum as the motion of a finite number of particles is used to analyze the state of Hall-thrusters. It doesn't adopt any modelization, so takes a long time to compute [2][3][4][5].

The JAXA has been developing in-house Full-PIC program called NSRU-Full-PIC. An important computing tasks in the code is called the aggregation process, which can cause the RAW hazards. In the original code, this process is executed sequentially. Miyajima et al. introduced an atomic operation to implement the aggregation process on a GPGPU [6]. They could only achieve 10% of performance improvement. Also, a few subroutines of NSRU-Full-PIC were offloaded to a Zynq, which is an ARM-based SoC with FPGA has been proposed [7]. However, their focus was not on the aggregation process due to the limited resources of the used platform.

This paper addresses the RAW hazard of the aggregation process by changing the algorithm for introducing reduction operations. We implement 4 types reduction kernels on Intel's mid-range SoC, Arria10 using Intel FPGA SDK for OpenCL. The contribution of the paper is as follows.

- We avoided the RAW hazards of the aggregation process modifying loops to reduction operations.
- We implemented an OpenCL reduction kernel for optimizing the performance and FPGA resource usage of Intel's Arria 10 SoC using Intel FPGA SDK for OpenCL.

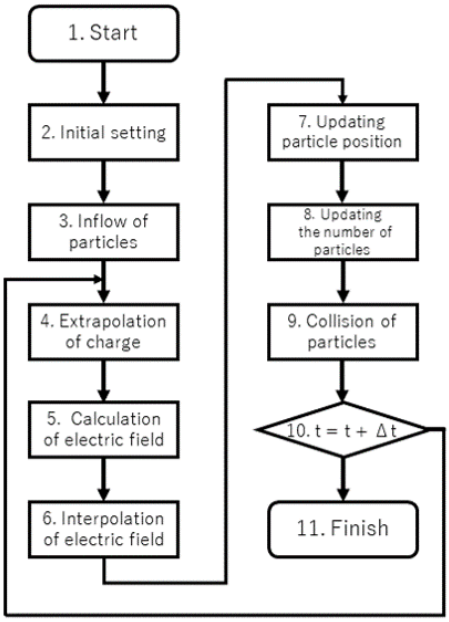


Figure 3: Flow chart of NSRU-Full-PIC

2. AGGREGATION PROCESS IN NSRU-FULL-PIC

NSRU-Full-PIC is a numerical simulation program for Hall-thrusters under development by JAXA. The code is written in Fortran90 of about 7000 lines. In this research, we adopted this code as a target of acceleration. NSRU-Full-PIC analyses the plasma behavior and state of the electrostatic field in the channel in each time step by updating particle and field physical quantities alternately, which has a large loop structure. One-time step corresponds to real time $1 \times 10^{-12}\,sec.$ A computational field called a cell divides the channel inside the thruster. There are 270×310 cells, and the distance between them is $0.2\,mm$. In addition, the number of particles necessary for thruster analysis is tens - hundreds of millions. Figure 3 shows the processing flow of NSRU-Full-PIC.

We conducted a preliminary experiment of NSRU-Full-PIC. We used a CRAY XE 6[8] supercomputer in Kyoto University with 128 processes. The evaluation environment is as follows, CPU: an AMD Opteron 6200 2.5 GHz, Memory: 64 GB/node, OS: a Cray Compute Node Linux, MPI Process: 128. Figure 2 shows the profiling result of NSRU-Full-PIC by the CRAY XE 6 with 128 MPI processes. The aggregation process accounted for about 40% of the total processing time. Thus, the aggregation processing is a largest part of NSRU-Full-PIC. The aggregation process adds the physical quantities held by each particle to the four corners of the cell containing them. Figure 4(a) shows how it is performed on multiple particles. Here, the values held by two particles p1 and p2 are added to GP[0-3], and the values are then updated. If we execute the computation of two particles in parallel, RAW hazards can occur, which means that it has to be done sequentially. This is a vital problem to be

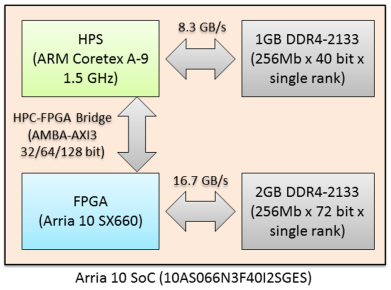


Figure 4: Arria10 consists of HPS and FPGA.

addressed to enable efficient parallel processing.

For the practical simulation, 270×310 cells are used and each cells has 256 particles at most. Thus, $270 \times 310 \times 256 = 21,427,200$ particles are in the simulation. In Figure 3, The aggregation process is included in both the step 4 (extrapolation of charge) and the step 9 (updating the number of particles).

3. INTEL FPGA SDK FOR OPENCL AND ARRIA 10 SOC

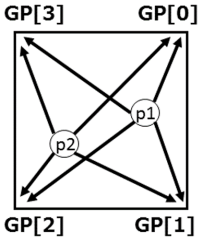
3.1 Intel's Arria 10 SoC

Arria 10 SoC is a mid-range SoC FPGA developed by Intel [9]. It consists of Hard Processor System (HPS) with dual core ARM Cortex-A9 MPCore and the FPGA logic as shown in Figure 4. The HPS unit is consisting of a processor unit including a CPU, cache, on-chip memories, external memory interface, a communication interface controller, and AXI interconnect. The FPGA logic part embeds hardened floating-point DSPs for high performance numerical computation. Arria 10 SoC can compute floating point operations using this much faster than common FPGAs such as Zynq.

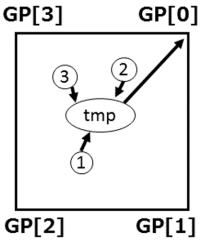
3.2 Intel FPGA SDK for OpenCL

Intel FPGA SDK for OpenCL is an OpenCL-based High-Level Synthesis environment for FPGAs[10]. It is designed for describing high-performance FPGA circuits in a short time. OpenCL is a parallel programming framework that can be used in a multiprocessor environment composed of various processors such as CPUs, GPUs, and FPGAs. Intel FPGA SDK for OpenCL introduces two kinds of code, kernel code and host code. The kernel code is for the operation of an arithmetic processor (OpenCL device), and the host code is for an operation of the control processor (host). They are described in OpenCL C language and C++ with OpenCL runtime Application Programming Interface (API). In addition, Intel FPGA SDK for OpenCL provides a board support package (BSP) that supports peripheral circuits such as PCIe bus between the OpenCL device and host, and an interface with external memory. By using the BSP, users can operate FPGA without designing the peripheral interface.

The OpenCL programming model consists of two hierarchical layer, work-group and work-item. Work-group is a set of work-items, and the OpenCL device executes work-item based processing. Global memory and constant memory can



(a) Aggregation process on multiple particles sequentially could cause RAW hazards.



(b) Reduction and temporary value are used.

Figure 5: The aggregation process in a cell

be accessed from all work-groups. The global memory is readable and writable, but the constant memory is read only. On the other hand, local memory is used for work-group and can be accessed from all work-items belonging to it. Also, private memory is work-item specific.

There are two types of kernel program in the Intel FPGA SDK for OpenCL, a Single work-item kernel and a NDRange kernel. The Single work-item kernel corresponds to task parallel model. There is only one work-group and one work-item in Single work-item kernel, and so kernel code can be described like sequential programming. The compiler extracts the parallelism in the kernel code, and makes pipelines in the loop. On the other hand, the NDRange kernel corresponds to data parallel model. Each work-item corresponds to a thread space and is executed in a pipelined manner. With NDRange kernel, it is possible to specify kernel pipeline multiplexing and vectorization for multiple work-items, which contribute to improvement of throughput. However, it can cause an increase in FPGA resource usage.

4. IMPLEMENTATION

4.1 Avoiding the RAW hazards

As described in Section 1, the step including the aggregation process requires high computational cost, and the avoidance of the RAW hazard is essential for parallelization. Figure 4(b) shows the outline of our implementation. Here, in order to avoid the RAW hazard, particle basis computing is changed into cell by cell computing in the source code level. That is, values of particles are added to four corners of the cell. As shown in Figure 4(b), the value of GP[0] is updated by a temporal variable which gathers the values of all particles in the cell. Since the update is done at once after adding values of all particles into the temporal variable, the RAW hazard never occurs. The same processing is performed for GP[1-3]. In this case, the reduction which is a common computation pattern in high speed computing can be used for computing temporal variables. Although the parallelism of the reduction is decreased at the later steps of communication, there are a lot of cells and the tree structure implemented on an FPGA logic can be used in the pipelined manner.

4.2 Reduction Kernels

We implemented 4 types of reduction codes. All codes adopted *Single work-item kernel*. In the implementation, the reduction is single precision floating point operations. For data communication between host CPU and FPGA, we used *clEnqueueWriteBuffer* and *clEnqueueReadBuffer* provided by OpenCL runtime API.

4.2.1 Full-Unroll implementation

Full-Unroll implementation is adopted for the eight-loops structure as shown in the following partial pseudo code. Each level of reduction was described in independent loop. For each loop, unrolling is performed by adding *#pragma unroll* in the source code, and all the loops are fully expanded. Additionally, we added *volatile* to the input argument of the kernel code. Caching generation was invalidated for reducing the FPGA resource usage.

Listing 1: Partial pseudo code of Full-Unroll

```
kernel void
full_unroll (__global volatile const float* restrict input,
                    __global float* restrict output)
...
// 1st level (256 -> 128)
#pragma unroll
for (i = 0; i < 256; i++) // level1 (256->128)
  buf1[i] = input[i*2 + 0] + input[i*2 + 1];

// 2nd level (128 -> 64)
#pragma unroll
for (i = 0; i < 128; i++) // level2 (128->64)
  buf2[i] = buf1[i*2 + 0] + buf1[i*2 + 1];
...
// 64, 32, 16, 8, and 4 iteration loops are followed.
...
// 8th level (2 -> 1)
  buf8[i] = buf7[i*2 + 0] + buf7[i*2 + 1];
...
```

4.2.2 Read-1 implementation

Read-1 implementation was adopted for the loop structure as shown in the following partial pseudo code. This loop structure performs reduction of 16 elements, and is repeated 16 times. Each level of reduction is manually described in tree-type structure. In the next section, we evaluated the number of unrolls, N.

Listing 2: Partial pseudo code of Read-1

```
kernel void
blueread-1 (__global volatile const float* restrict input,
                    __global float* restrict output)
...
#pragma unroll N
for (int i = 0; i < cells*16; i++){
  // 1st level (16 -> 8)
  level1[0] = input[i+0] + input[i+1];
  level1[1] = input[i+2] + input[i+3];
  ...
  level1[7] = input[i+14] + input[i+15];

  // 2nd level (8 -> 4)
  level2[0] = level1[0] + level1[1];
  ...
  level2[3] = level1[6] + level1[7];

  // 3rd and 4th level are followed.
}
...
```

4.2.3 Read-16 implementation

Read-16 implementation is similar to the structure for *Read-1* implementation, except that float 16, a vector data type is used as a kernel argument. In the next section, we evaluated the number of unrolls, N.

Listing 3: Partial pseudo code of Read-16

```
kernel void
blueread-16 (__global volatile const float16* restrict input,
                    __global float* restrict output)
...
#pragma unroll N
for (int i = 0; i < cells*16; i++){
  // 1st level (16 -> 8)
  level1[0] = input[i+0] + input[i+1];
  level1[1] = input[i+2] + input[i+3];
  ...
  level1[7] = input[i+14] + input[i+15];

  // 2nd level (8 -> 4)
  level2[0] = level1[0] + level1[1];
  ...
  level2[3] = level1[6] + level1[7];

  // 3rd and 4th level are followed.
}
...
```

4.2.4 Read-16-Vect implementation

Read-16-Vect implementation used vector addition for reduction processing of *Read-16* implementation. In the next section, we evaluated the number of unrolls, N.

Listing 4: Partial pseudo code of Read-16-Vect

```
kernel void
blueread-16 (__global volatile const float16* restrict input,
                    __global float* restrict output)
...
#pragma unroll N
for (int i = 0; i < cells*16; i++){
  // 1st level (16 -> 8)
  level1 = input[i].s01234567 + input[i].s89abcdef;

  // 2nd level (8 -> 4)
  level2 = level1.s0123 + level1.s4567;

  // 3rd level (4 -> 2)
  level3 = level2.s01 + level2.s23;

  // 4th level (2 -> 1)
  level4[i\%16] = level3.s0 + level3.s1;
...
}
...
```

5. EVALUATION

We compared the computation speed of the reduction circuits implemented on the FPGA and that of the software

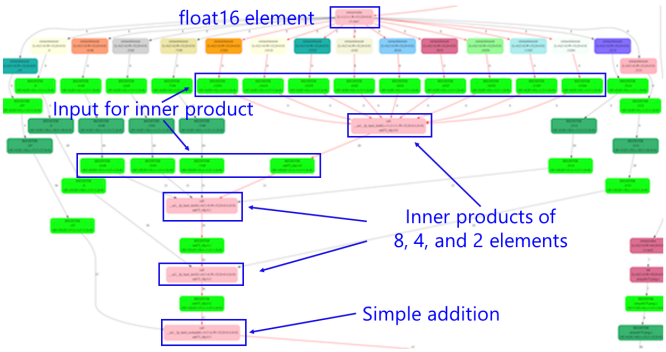


Figure 6: Generated schematic view of Read-16 implementation. Inner product was used instead of simple addition.

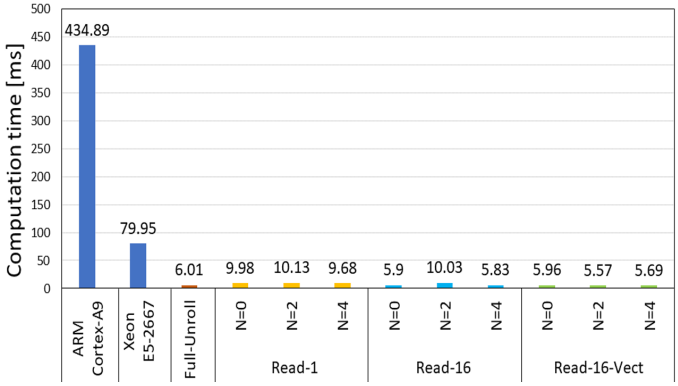


Figure 7: The computation time with Arria 10 SoC and the above CPUs when the size of the input data is 270 x 310 sets of 256 particles

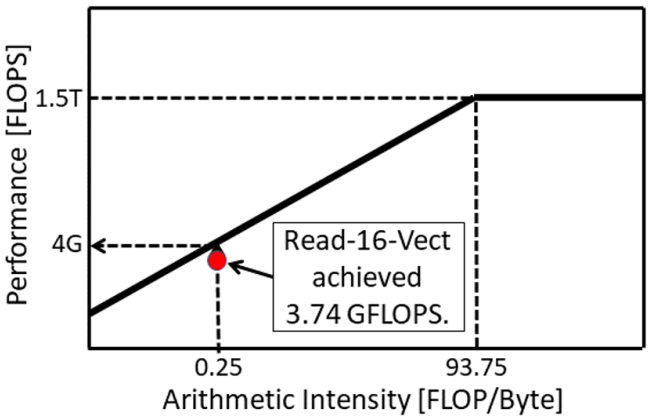


Figure 8: Evaluation using the roof-line model

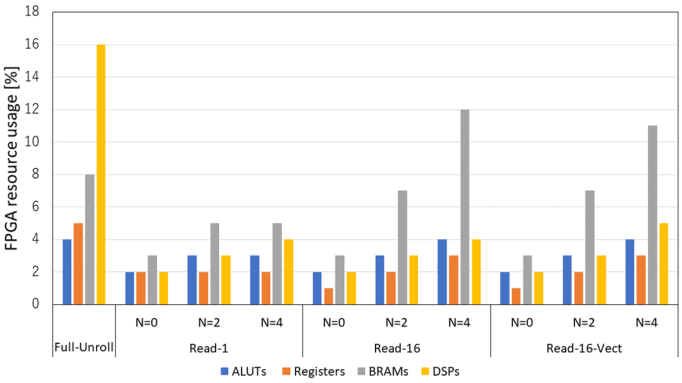


Figure 9: FPGA resource usage

execution on two CPUs, an ARM Cortex-A9 1.5GHz on Arria 10 SoC and a Xeon E5-2660 2.90 GHz. The evaluation environment is as follows. For the FPGA, we used Intel FPGA SDK for OpenCL 64-bit Offline Compiler ver. 16.0.2 with -v, -g and –fp-relaxed option for the kernel code compilation, and gcc compiler ver. 4.8.3 with -O3 option for the host code compilation. For the ARM Cortex-A9, we used gcc compiler ver.4.8.4 with -O3 option. For the Xeon E5-2660, we used gcc compiler ver. 4.4.7 with -O3 option.

First, we analyzed generated code. We examined the kernels adding –dot option to Intel FPGA SDK for OpenCL Kernel Compiler (*aoc* command). In the case of *Read-16*, four DSPs and inner products were used to calculate as shown in Figure 6. The inner product of 8, 4, and 2 elements were performed. Finally, simple addition was performed. The latency of each computation was 11, 8, 6, and 4. Input interval (II) was 1.

Figure 7 shows the computation time with our implementation and the above CPUs when the size of the input data is 270 x 310 sets of 256 particles. As a result, *Full-Unroll* implementation was about 72.4 times faster than that of an ARM Cortex-A9, and about 13.3 times faster than that of a Xeon E5-2667. In *Read-1* implementation, that with N = 4 was the fastest. This was 1.6 times slower than *Full-Unroll* implementation. In *Read-16* and *Read-16-Vect*, we achieved speedup as fast as *Full-Unroll* with the N = 0, 4 implementations. The fastest in all implementations was *Read-16-Vect* with N = 4, about 76.4 times faster than that of an ARM Cortex-A9 and about 14.1 times faster than that of a Xeon E5-2667.

Figure 8 shows the results using the roof-line model. In Arria 10 SoC, the peak computing performance is 1.5 TFLOPS and the peak bandwidth is 16 GB/s. Because the number of single precision floating point operations is 255 times and the number of memory accesses is 260 times in a cell of the aggregation process, the arithmetic intensity is about 0.25 FLOP / Byte. Hence, the theoretical performance is about 4.0 GFLOPS. Since *Read-16-Vect* implementation achieved 3.74 GFLOPS, we can conclude that this implementation achieved 93.5% of theoretical performance by memory limitation. Table 1 summarizes the performance of our implementations.

Figure 9 shows the FPGA resource usage in our implementation. In *Read-16* with N = 0 and *Read-16-Vect* with N = 0, we could reduce the ALUTs by 50.0%, the Registers by 80.0%, the BRAMs by 62.5%, and the DSPs by 87.5% without increasing computation time compared to *Full-Unroll* implementation.

Here, we estimate the speed up ratio in the overall off-

Table 1: Performance

	Unroll	Clock frequency [MHz]	Computation time [ms]	GFLOPS	BW [MB/s]	Ave. Burst Read	Logic Utilization [%]	ALUTs [%]	Dedicated logic registers [%]	Memory blocks [%]	DSP blocks[%]
Full-Unroll	Full	236.7	6.01	3.54	14544.2	16	9	4	5	8	16
Read-1	-	244.4	9.98	2.13	1008.6	16	3	2	2	3	2
	2	224.1	10.13	2.10	1617.8	1	5	3	2	5	3
	3	210.4	1026.95	0.02	210.4	1	6	3	3	10	5
	4	231.0	9.68	2.20	2974.1	16	5	3	2	5	4
Read-16	-	250	5.90	3.61	14963.1	16	3	2	1	3	2
	2	247.5	10.03	2.12	14989.1	16	5	3	2	7	3
	3	213	1014.67	0.02	89.8	4	8	4	4	12	5
	4	246.9	5.83	3.65	15015.1	16	8	4	3	12	4
Read-16-Vect	-	243.8	5.96	3.57	14730.7	16	3	2	1	3	2
	2	262.5	5.57	3.82	15746.5	16	5	3	2	7	3
	3	215.6	1002.14	0.02	90.9	4	8	4	4	11	6
	4	254.2	5.69	3.74	15404.8	16	7	4	3	11	5

loading of NSRU-Full-PIC incorporating *Read-16-Vect* implementation. According to the NSRU-Full-PIC profiling described in Section 2, the aggregation process accounted for 38.9% of the total processing time. Therefore, we can estimate that the speed up ratio is about 1.61 times as the execution by an ARM Cortex-A9 and about 1.55 times as the case with a Xeon E5-2667.

6. CONCLUSION

We off-loaded an aggregation process of NSRU-Full-PIC which is a particularly high computation cost to an Arria 10 SoC using Intel FPGA SDK for OpenCL. The reduction computation is adopted to avoid the RAW hazard in aggregation process. We implemented and evaluated 4 types Single work-item kernels, *Full-Unroll*, *Read-1*, *Read-16*, and *Read-16-Vect*. The fastest in all implementations was *Read-16-Vect* with N = 4, about 76.4 times faster than that of an ARM Cortex-A9 and about 14.1 times faster than that of a Xeon E5-2667. In this implementation, we achieved 93.5% of theoretical performance. And we could reduce the FPGA resource usage without increasing computation time compared to *Full-Unroll* implementation.

As a future work, we plan to extend the current implementation of aggregation process to all cells in the code. We also plan to evaluate the overall off-loading of NSRU-Full-PIC incorporating the parallelization of the aggregation processing to the cell base.

ACKNOWLEDGMENT

The present study was supported in part by the JST/CREST program entitled "Research and Development on Unified Environment of Accelerated Computing and Interconnection for Post-Petascale Era" in the research area of "Development of System Software Technologies for post-Peta Scale High Performance Computing".

7. REFERENCES

[1] Kuriki Kyoichi and Arakawa Yoshihiro. *Introduction to electric propulsion rockets*. Tokyo Shuppan, 2003.

[2] Yokota Shigeru, Komurasaki Kimiya, and Arakawa Yoshihiro, . Plasma Density Fluctuation Inside a Hollow Anode in an Anode-layer Hall Thruster . In *42th Joint Propulsion Conference and Exhibit, AIAA-2006-5170*, 2006.

[3] Hirakawa Miharu . Electron Transport Mechanism in a Hall Thruster . In *IEPC-97-021*, 1997.

[4] Justin M. Fox . *Advances in Fully-Kinetic PIC Simulation of a Near-Vacuum Hall Thruster and Other Plasma Systems* . PhD thesis, 2007.

[5] James Joseph Szabo . *Fully Kinetic Numerical Modeling of a Plasma Thruster* . PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.

[6] Takaaki Miyajima, Shinatora Cho, and Naoyuki Fujita. A study of gpu acceleration of "source" part in hall-thruster simulation. In *IEICE Tech. Rep.*, Vol. 115 of *CPSY2015-62*, pp. 7–12, Dec. 2015.

[7] R. Sakai, N. Sugimoto, T. Miyajima, N. Fujita, and H. Amano. Acceleration of full-pic simulation on a cpu-fpga tightly coupled environment. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pp. 8–14, Sept 2016.

[8] supercomputer system (from October, 2016) — Supercomputer System —Academic Center for Computing and Media Studies, Kyoto University. http://www.iimc.kyoto-u.ac.jp/ja/services/comp/supercomputer/. 2016/12/29/20:50.

[9] Intel Corporation. Arria 10 SoC - Features:. https://www.altera.com/products/soc/portfolio/arria-10-soc/features.html. 2017/01/29/14:04.

[10] Intel Corporation. Intel FPGA SDK for OpenCL Programming Guide - aocl_programming_guide. https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdkaocl_programming_guide.pdf. 2016/11/17/14:00.