Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

© 2020 The authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/APC200047

Design of an FPGA-Based Matrix Multiplier with Task Parallelism

Yiyu TAN[a,1], Toshiyuki IMAMURA[a], and Daichi MUKUNOKI[a]
[a] *RIKEN Center for Computational Science, Kobe, Hyogo, Japan*

Abstract. Matrix multiplication requires computer systems have huge computing capability and data throughputs as problem size is increased. In this research, an OpenCL-based matrix multiplier with task parallelism is designed and implemented by using the FPGA board DE5a-NET to improve computation throughput and energy efficiency. The matrix multiplier is based on the systolic array architecture with 10 × 16 processing elements (PEs), and all modules except the data loading modules are autorun to hide computation overhead. When data are single-precision floating-point, the proposed matrix multiplier averagely achieves about 785 GFLOPs in computation throughput and 66.75 GFLOPs/W in energy efficiency. Compared with the Intel's OpenCL example with data parallelism on FPGA, the SGEMM routines in the Intel MKL and OpenBLAS libraries executed on a desktop with 32 GB DDR4 RAMs and an Intel i7-6800K processor running at 3.4 GHz, the proposed matrix multiplier averagely outperforms by 3.2 times, 1.3 times, and 1.6 times in computation throughput, and by 2.9 times, 10.5 times, and 11.8 times in energy efficiency, respectively, even though the fabrication technology is 20 nm in the FPGA while it is 14 nm in the CPU. Although the proposed FPGA-based matrix multiplier only delivers 6.5% of the computation throughput of the SGEMM routine in the cuBLAS performed on the Nvidia TITAN V GPU, it outperforms by 1.2 times in energy efficiency even though the fabrication technology of the GPU is 12 nm.

Keywords. Matrix multiplication, FPGA, OpenCL

1. Introduction

Matrix multiplication is one of the fundamental building blocks of linear algebra, and has been widely applied in high performance computing (HPC) to solve scientific and engineering problems, such as deep learning, data analytics, and so on. In general, matrix multiplication requires computing systems to have huge computation capability and memory bandwidth as problem size grows. Nowadays, many methods and algorithms have already been developed to speed up computation through parallel programming techniques or improving the efficiency of memory hierarchy to reduce data access overhead in supercomputers, GPUs, multicores, many-cores, and cluster systems. On the other hand, power problems become more and more serious in HPC systems, and heterogeneous architectures are becoming the mainstream, in which GPUs or FPGAs are tightly integrated with multicore processors as accelerators to reduce power consumption, especially FPGAs. FPGAs deliver much higher energy efficiency through data parallelism and pipelining parallelism using a sea of individually PEs running at low

[1] Corresponding Author: Yiyu Tan, RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo, Japan; E-mail: tan.yiyu@riken.jp.

clock frequency. In recent years, an FPGA contains thousands of hardened floating-point arithmetic units and million-byte on-chip block RAMs (BRAMs). Furthermore, high-level synthesis tools let developers shift their focus from low-level HDL-based designs to C, C++, or OpenCL codes annotated with directives, which makes the system development much easier and development time being shortened. All these lead FPGAs to become attractive in HPC.

FPGA-based acceleration on matrix multiplication has received much attention in industry and academics. Zhuo et al. [1] and Dou et al. [2] proposed a matrix multiplier with PEs being one-dimensional linear array architecture. Zhuo et al. analyzed the design tradeoffs and optimized the design based on the hardware constraints. Dou et al. proposed a parallel block algorithm to improving performance by exploiting the data locality and reusability. Based on their work, Wu et al. [3] developed an I/O and memory optimized blocking algorithm to improve memory efficiency and reduce the required hardware resources. Kumar et al. [4] also presented a one-dimensional array architecture of PEs and applied the rank-1 update algorithm to schedule input data to PEs. Different with the architectures proposed by Dou and Zhou, this architecture distributes the same elements of the first matrix to all PEs through broadcasting. Pedram et al. [5][6] introduced a two-dimensional linear array architecture for matrix multiplication, in which data exchange between PEs was performed through row/column broadcasting buses. Such two-dimensional architecture provides benefits in scalability, addressing, and data movement over one-dimensional array architecture.

There are other FPGA-based accelerators on matrix multiplication for different purposes. Giefers et al. [7] presented an FPGA-based accelerator for matrix multiplication on a hybrid FPGA/CPU system to study energy efficiency. Jiang et al. [8] introduced a scalable macro-pipelined accelerator to perform matrix multiplication to exploit temporal parallelism and architectural scalability. Wang et al. [9] integrated multiple matrix accelerators with a master processor and built a universal matrix processor. Z. Jovanovic et al. [10] presented an accelerator to minimize resource utilization and maximum clock frequency by returning the computation results to the host processor as soon as they were computed. Andrade et al. [11] adopted high-level synthesis approach to generate two-dimensional embedded processor arrays for matrix algorithms. Holland [12] proposed high-level synthesis optimization strategies to maximize the utilization of the DSPs and BRAMs in blocked matrix multiplication. In this research, an FPGA-based matrix multiplier with task parallelism is presented for large-scale matrix multiplication. The major contributions of this work are as follows.

(1) Design and implementation of a matrix multiplier with task parallelism. The matrix multiplier is based on the systolic array architecture with 10×16 PEs, and data reuse and optimization techniques are applied to improve computing performance and energy efficiency.
(2) Task parallelism and data vectorization. The system is partitioned into different single-work-item kernels according to dataflow, and most of the kernels work at the autorun mode to reduce computation overhead. High-speed and high-bandwidth buffers are adopted to exchange data between PEs and kernels. Data vectorization is applied to compute the dot product of multiple data inside a PE to enhance the computation capability. The proposed system averagely achieves about 785 GFLOPs in computation throughput and 66.75 GFLOPs/W in energy efficiency in the case of data being single-precision floating-point.

The remainder of this paper is organized as follows. The system design and implementation are introduced in Section 2. In Section 3, performance evaluation results are presented, followed by conclusions drawn in Section 4.

2. System Design and Implementation

A matrix multiplication $C = A \times B$ is generally defined as follows:

$$C[i][j] = \sum_{k=0}^{P-1} A[i,k] \times B[k,j] \qquad (0 \leq i < M,\ \ 0 \leq j < N)$$

Where A, B, and C are $M \times P$, $P \times N$, and $M \times N$ matrices, respectively. The computations from the above formula can be described by the pseudocode shown in Figure 1. In Figure 1, a matrix multiplication consists of three loops, and their positions can be changed. The computations require $2{\times}M{\times}P{\times}N$ floating-point operations and $M{\times}P{+}P{\times}N{+}M{\times}N$ memory accesses. To speed up the computations, firstly, the computation load in a clock cycle should be maximized, which means more floating-point multipliers and adders are involved into computation and work in parallel to get one or more of the scalar product C_{ij} at a clock cycle. This may be achieved by unrolling the inner loop (k) and outer loops (i and j) to get more parallel iterations at arithmetic level. At circuit level, it is achieved using deep pipelining of floating-point multiplication and addition units inside a PE, and many PEs are applied to calculate C_{ij}. Although the loops can be unrolled completely, they are limited by the number of DSP blocks inside an FPGA, which are applied to implement the floating-point arithmetic units. Secondly, parallel data stream is needed to feed the pipeline and shorten the overhead of data access. Thus, on-chip buffers are demanded to store the elements of matrices A and B in advance. In general, multiple-port and large-size buffers can read/write data in parallel and keep more data, but they are constrained by the size of BRAMs inside an FPGA. In addition, the overheads to writing data from external memory to on-chip buffers are affected by memory bandwidth. To address these, the blocked matrix multiplication algorithm is applied to partition the matrices into smaller blocks (sub-matrices), and parallelisms, such as data parallelism and task parallelism, are put on the sub-matrix multiplications. In this research, the matrix multipliers based on such two parallelisms are designed using OpenCL programming language and implemented using FPGA, respectively.

```
for i = 0; i < M; i++
   for j = 0; j < N; j++ {
     sum = 0.0;
     for k = 0; k < P; k++
        sum = sum + A[i][k] × B[k][j];
     C[i][j] = sum;}
```

Figure 1. Pseudocode of a matrix multiplication.

2.1 Matrix Multiplier with Data Parallelism

The matrix multiplier with data parallelism is referenced from the Intel's OpenCL design example [13], and the related code is shown in Figure 2. Both matrices A and B are stored by row-major in the host, and two local buffers, A_local and B_local, are defined to store the block data of matrices A and B, respectively (lines 4 and 5). Matrices A and B are written into the on-board DDR memory from the host machine before computation, and then a block of A and a block of B are read into A_local and B_local to perform the product of two blocks by the computation engine, which is defined by the N-dimensional index space (NDRange) in the OpenCL execution model through the attribute option

__attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1))). The calculated product C[i][j] is written into the on-board DDR memory (line 24), and finally transferred to the host machine after all computations are finished. Along with data reading from the external memory (lines 14 - 17), the block data of B are transposed before they are stored in the local buffer (line 17) to ensure consecutive data access during computation (line 21). Furthermore, the inner loop in Figure 1 is fully unrolled by adding the directive (line 19) to instruct the OpenCL compiler to implement parallelism by using more DSP blocks, and the outer loop in Figure 1 is parallelized and realized through introducing the two-dimensional computing engine defined by the NDRange. The main disadvantage of this matrix multiplier is that system will be stalled until computations in a block are completed (line 22) and new blocks are fed into the local buffers (line 18) to maintain data synchronization.

OpenCL code for blocked matrix multiplication

```
1:  __kernel void matrixmult ( __global float *restrict C, __global float *A,
2:                                    __global float *B, int A_width, int B_width)  {
3:  //define local storage for a block of input matrices A and B
4:    __local float A_local[BLOCK_SIZE][BLOCK_SIZE];
5:    __local float B_local[BLOCK_SIZE][BLOCK_SIZE];
6:   int block_x = get_group_id(0);   //define block index: row
7:   int block_y = get_group_id(1);   //define block index: column
8:   int local_x = get_local_id(0);   //define local index: row
9:   int local_y = get_local_id(1);   //define local index: column
10:  int a_start = A_width*BLOCK_SIZE*block_y;   //loop start and stop points
11:  int a_end = a_start + A_width - 1;
12:  int b_start = BLOCK_SIZE * block_x;
13:  float sum = 0.0f;
14:  for (int aa=a_start, bb=b_start; aa<=a_end; aa+=BLOCK_SIZE, bb+= (BLOCK_SIZE*B_width)) {
15:    // load the matrices into local memory, and perform B<=B^T
16:    A_local[local_y][local_x]=A[aa+A_width*local_y+local_x];
17:    B_local[local_x][local_y]=B[bb+B_width*local_y+local_x];
18:    barrier(CLK_LOCAL_MEM_FENCE); // wait for the entire block to be loaded.
19:    #pragma unroll
20:    for (int k = 0; k< BLOCK_SIZE; ++k) {
21:        sum += A_local[local_y][k]*B_local[local_x][k]; }
22:    barrier(CLK_LOCAL_MEM_FENCE);  // wait for completion of computation.
23:  }
24   C[get_global_id(1)*get_global_size(0)+get_global_id(0)]=sum; // store result in matrix C
25:  }
```

Figure 2. OpenCL code for blocked matrix multiplication with data parallelism.

As shown in Figure 2, blocks of both matrices are read into the local buffers at each iteration, and multiplications are then carried out by the computation engine. The number of iterations is determined by the block size and matrix scale. The required hardware resources are affected by the complexity of the kernel shown in Figure 2, and not associated with the dimensions of matrices A and B, which only affect the number of iterations. In Figure 2, the consumed hardware resources are mainly determined by the size of local buffers (A_local and B_local), the unrolling of inner loop, the scale of the arithmetic array defined by the NDRange, and the kernel vectorization to specify the number of work items within a work group to execute in a single instruction multiple data (SIMD) fashion. Except for the kernel vectorization and the unrolling of inner loop being specified individually, the size of local buffers and the scale of the arithmetic array are determined by the block size. Consequently, the block size has great impacts on the required hardware resources and system performance.

2.2 Matrix Multiplier with Task Parallelism

In the matrix multiplier with task parallelism, the system is divided into different function modules in accordance to the data flow, and each function module is described as a kernel in OpenCL. As illustrated in Figure 3, the system consists of the computing kernel, data-feeding modules (feed_mat_A_kernel and feed_mat_B_kernel), matrix-loading modules (load_mat_A_kernel and load_mat_B_kernel), and data output module. Each kernel works with single work-item. Except for the matrix-loading modules, other kernels run at the autorun mode to reduce computation overhead. The function of each module is described as follows in detail.

- Matrix-loading modules. The matrix-loading modules read data from the on-board DDR memory into buffers according to the data vectorization, block size, block position, and data reuse.
- Data-feeding modules. The data-feeding modules read data from the buffer and feed the related data to the computing kernel. As shown in Figure 3, the number of data-feeding modules for matrices A and B equals to the number of rows and columns of the systolic array in computing kernel, respectively. Each data-feeding module will check and feed the corresponding data to the specific row or column of the computing kernel, and then forward the other data to the next neighbor data-feeding module.
- Computing kernel. The computing kernel is a systolic array with 10 × 16 PEs, and high-speed and high-bandwidth channels are applied to connect PEs and kernels. In the computing kernel, data of matrix A are shifted from the left to right while data of matrix B are moved from the top to bottom in the systolic array to reuse data.
- PE. The PE is the arithmetic unit to perform computation. The block diagram of a PE is presented in Figure 4, which consists of eight arithmetic units to compute the dot product of eight data at a clock cycle through pipelining, namely the data vectorization is eight. In Figure 4, each arithmetic unit is generated by the IP generator from the Quartus Prime Pro to perform different operations, and is implemented by using the hardened DSP blocks inside FPGA. In the current design, each arithmetic unit consumes one DSP block, which contains an adder and a multiplier with single-precision.
- Data output module. The data output module outputs the computation results to the on-board DDR memory. Similar as the data-feeding module for matrix B, several data output modules will be applied in accordance to the column of the systolic array in the computing kernel.

2.3 System Implementation

The matrix multipliers with data parallelism and task parallelism are compiled by using the Intel FPGA SDK for OpenCL 16.1 and implemented by using an FPGA board DE5a-NET from Terasic [16], which includes an Intel Arria 10 GX FPGA 10AX115N2F45E1SG and 8 GB on-board DDR3 memory. The FPGA contains 1518 hardened single-precision floating-point units, 427,200 ALMs, and 53 Mb M20K memory. The on-board memory is arranged at two independent channels with each being 4 GB. The matrices A and B are written into different banks of the on-board memory through PCIe bus, and they are accessed independently through different channels. The product, namely the matrix C, is firstly stored into the same memory bank as the matrix A, and finally written back to the host machine. In the matrix multiplier with data

parallelism, the block size is 128 × 128 and the kernel vectorization is four. In contrast, in the matrix multiplier with task parallelism, the data vectorization is eight, and the block size of matrix A is 320 × 256 while the block size of matrix B is 256 × 512.

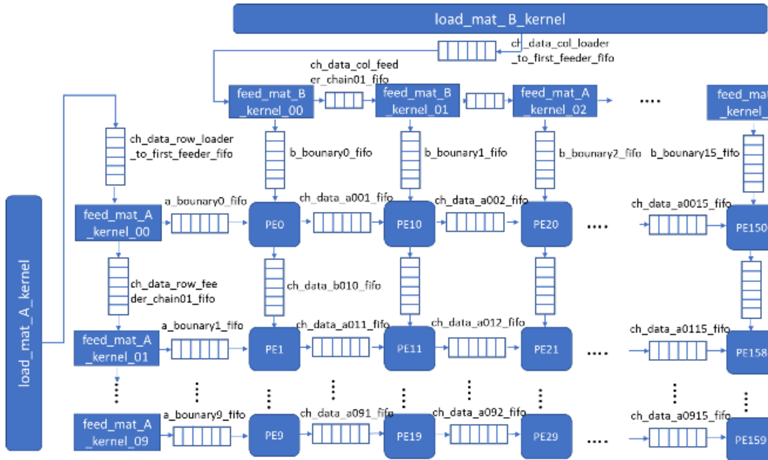

Figure 3. System block diagram.

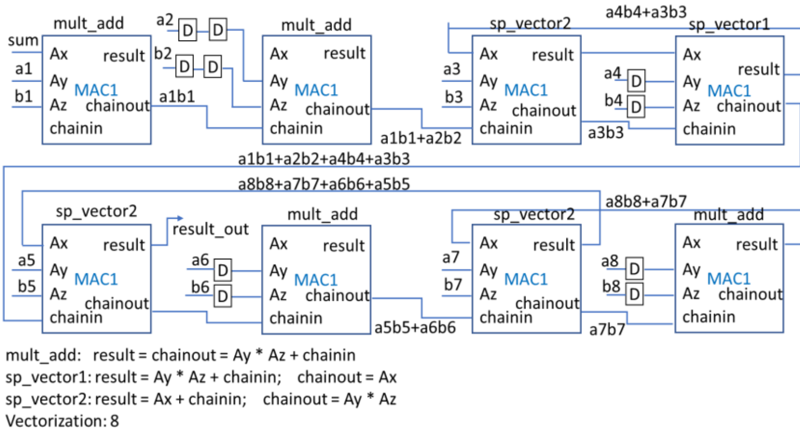

Figure 4. Block diagram of a PE.

The kernel code is compiled into the intermediate representations, applied necessary optimizations, converted to the Verilog files, and then performed synthesis, placement and routing to generate the final FPGA bitstream by the Intel FPGA SDK for OpenCL. Table 1 presents the hardware resource utilization in the two matrix multipliers in the case of data being single-precision floating-point. As shown in Table 1, the matrix multiplier with task parallelism utilizes more DSP blocks and gains much higher clock frequency over the matrix multiplier with data parallelism on FPGA. In both systems, the system performance is constrained by the size of on-chip BRAM blocks. Although the matrix multiplier with data parallelism consumes less DSP blocks (34%), if the block size is increase further to use more DSP blocks to improve system performance, such as 256 × 256, the on-chip BRAM blocks will be exhausted and the system cannot be synthesized because the block size affects the utilization of the BRAM blocks and DSP

blocks. On the other hand, in the matrix multiplier with task parallelism, the systolic array and data vectorization are determined by the available DSP blocks and BRAM blocks inside the FPGA. In the current design, the optimal systolic array contains 10 × 16 PEs, and the data vectorization is eight, which means each PE consumes eight DSP blocks. Therefore, the matrix multiplier requires 1280 (10×16×8) DSP blocks. Although the systolic array can be scaled up to 11 × 16 PEs according to the available hardware resources of the FPGA, the system is not synthesizable by the Intel FPGA SDK for OpenCL. In addition, the clock frequency of the proposed matrix multiplier with task parallelism is much higher than the matrix multiplier with data parallelism because of more optimized data path in accordance to the dataflow.

Table 1. Hardware resource utilization

Matrix multiplier	Hardware resource			
	Logic utilization	DSP blocks	RAM blocks	Clock frequency
Task parallelism	237128(56%)	1280(84%)	2529(93%)	305 MHz
Data parallelism	92002(22%)	520 (34%)	1774(65%)	235 MHz

3. Performance Evaluation

The proposed matrix multiplier is implemented by using the FPGA board DE5a-NET, and its performance is evaluated and compared with the matrix multiplier with data parallelism on FPGA [14-15], the SGEMM routine in the cuBLAS performed on the Nvidia TITAN V GPU [17], the SGEMM routines in the Intel MKL(version:2018.0.128) and OpenBLAS (version: 0.2.20) executed on a desktop machine with 32 GB DDR RAMs and an Intel i7-6800K processor running at 3.4 GHz. The operating system of the desktop machine is CentOS 7.0, and the compiler for the OpenBLAS is gcc 4.9.4. The compiler for the OpenCL is Intel FPGA SDK for OpenCL 16.1. In the GPU system, the host machine contains an Intel Xeon W-2123 processor with CentOS 7.4, CUDA 10.0, and the GPU driver version being 410.73. The fabrication technology in the FPGA Arria 10 is 20 nm while it is 14 nm and 12 nm in the Intel i7-6800K processor and TITAN V GPU, respectively. The execution time and power consumption are measured, and the computation throughput and energy efficiency are estimated. During estimation, the matrices are square and data are single-precision.

3.1 Computation throughput

Figure 5 shows the computation performance in the case of different matrix scales. The computation throughput is almost fixed on the FPGA system as the matrix scale is increased because the operations become computation-bound. However, it fluctuates in the SGEMM routines on the GPU with cuBLAS and the desktop machine with the MKL and OpenBLAS libraries. For example, when the matrix scale is increased from 7680 × 7680 to 20480 × 20480, the computation throughput in the FPGA-based matrix multiplier with data parallelism and task parallelism is about 240 GFLOPs and 780 GFLOPs, respectively. But the computation throughput of the SGEMM routine on the GPU is decreased from 13.47 TFLOPs to 11.28 TFLOPs. Similarly, the computation throughput is firstly increased from 586.23 GFLOPs to 631.03 GFLOPs, and then dropped to 532.43 GFLOPs in the SGEMM routine on the desktop machine with the MKL library. In Figure 5, the proposed matrix multiplier with task parallelism averagely

offers about 785 GFLOPs in computation throughput, which is about 3.2 times, 1.3 times, 1.6 times, and 6.5% in average over the matrix multiplier with data parallelism on FPGA, the SGEMM routines in the MKL and the OpenBLAS executed on the desktop machine, and the SGEMM routine in the cuBLAS performed on the TITAN V GPU, respectively. In addition, the block size of matrices A and B impacts on the computation throughput. If matrices A and B are 5120 × 4096 and 4096 × 8192, and the block sizes of A and B are 160 × 128 and 128 × 256, the computation throughput of the matrix multiplier with task parallelism is increased to about 868 GFLOPs.

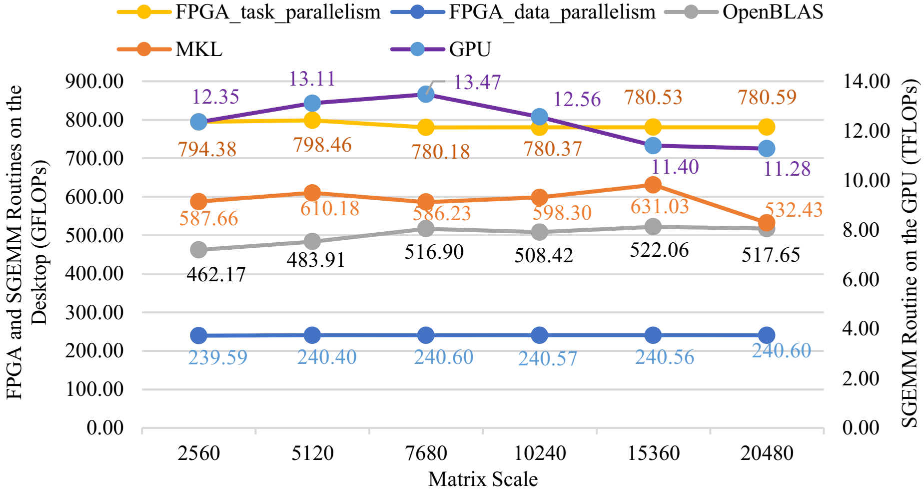

Figure 5. Computation throughput.

3.2 Energy Efficiency

To estimate the energy efficiency, the input current and voltage of the FPGA board and the desktop machine with the MKL and OpenBLAS libraries are measured every 200 ms by using a digital multimeter PC720M from the Sanwa when the FPGA system and the desktop machine are idle and active, respectively. The power consumption is calculated by multiplying the voltage and the current difference. The energy efficiency is computed by using equation 1.

$$P_{efficiency} = \frac{E_{computation_throughput}}{V \times (I_{active} - I_{idle})} \tag{1}$$

where E is the computation throughput, V is the voltage, and I is the current. I_{active} is the current when the SGEMM routine is performed or the FPGA board is active, and I_{idle} is the system current without computation. The term $I_{active} - I_{idle}$ denotes the actual consumed current by computations in the FPGA and the desktop machine with the SGEMM routine. The I_{active} and I_{idle} are the average of the measured values. In the GPU, the energy consumption is obtained through the "nvmlDeviceGetTotalEnergyConsumption" function provided by the Nvidia management library.

Figure 6 shows the energy efficiency of the FPGA-based matrix multipliers, the SGEMM routines in the MKL and OpenBLAS executed on the desktop machine, and the SGEMM routine in the cuBLAS performed on the GPU. The energy efficiency of the proposed matrix multiplier with task parallelism ranges from 71.74 GFLOPs/W to 63.32 GFLOPs/W, and is about 66.75 GFLOPs/W in average, which is about 2.9 times, 1.2 times, 10.5 times, and 11.8 times, over the FPGA-based matrix multiplier with data parallelism, the SGEMM routine on the GPU, and the SGEMM routines on the desktop machine with the Intel MKL and the OpenBLAS, respectively, even though the fabrication technology of the FPGA (20 nm) is significantly lagged behind that of the CPU (14 nm) and the GPU (12 nm). In addition, the proposed system gains much higher energy efficiency in the case of small problem size because the computation throughput is almost fixed while the consumed current by the FPGA is increased as the problem size grows.

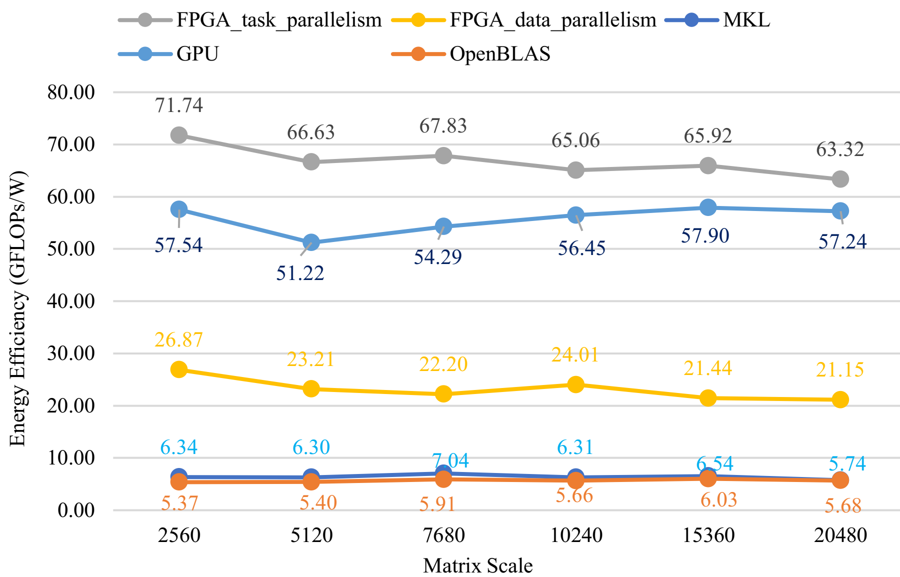

Figure 6. Energy efficiency.

4. Conclusions

Matrix multiplication is one of the basic building blocks of linear algebra, and widely applied in the HPC to solve scientific and engineering problems. Its performance significantly affects the whole system performance, especially when the problem size is large. In addition, power problem becomes more and more serious in HPC systems. In this research, an FPGA-based matrix multiplier with task parallelism is developed to improve system performance and energy efficiency for large-scale matrix multiplication, in which system is divided into different kernels in accordance to the data flow, a systolic array is adopted to carry out computations, and high-speed and high-bandwidth buffers are used to connect PEs in the systolic array and different kernels. It outperforms the FPGA-based matrix multiplier with data parallelism and the SGEMM routines in the highly optimized MKL and OpenBLAS libraries executed on a desktop machine in

computation throughput and energy efficiency. Compared with the SGEMM routine in the cuBLAS performed on the Nvidia TITAN V GPU, the proposed matrix multiplier is significantly defeated in computing performance, but it wins in energy efficiency. In future work, we will port and optimize the proposed design on other FPGA platforms with more hardware resources and multiple FPGAs to evaluate its performance, and compare the performance with the popular Xeon gold processor in HPC.

Acknowledgments

Thanks for Intel's donation of the FPGA board DE5a-NET and the related EDA tools through University Program. This work was partly supported by the Grant-in-Aid from Foundation for Computational Science (FOCUS). The performance on GPUs was obtained using the GPU cluster installed in Tokyo Woman's Christian University.

References

[1] L. Zhuo, and V. Prasanna, High-performance designs for linear algebra operations on reconfigurable hardware, *IEEE Transactions on Computers* 57(8) (2008), 1057-1072.
[2] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. Gaydadjiev, 64-bit floating-point FPGA matrix multiplication, *ACM/SIGDA 13th international symposium on Field Programmable Gate Arrays*, 2005, pp. 86-95.
[3] G. Wu, Y. Dou, and M. Wang, High performance and memory efficient implementation of matrix multiplication on FPGAs, *2010 International Conference on Field-Programmable Technology*, 2010, pp. 134-137.
[4] V. Kumar, S. Joshi, S. Patkar, and H. Narayanan, FPGA based high performance double-precision matrix multiplication, *International Journal of Parallel Programming* 38 (2010), 322–338.
[5] A. Pedram, A. Gerstlauer, and R. Geijn, Algorithm, architecture, and floating-point unit codesign of a matrix factorization accelerator, *IEEE Transactions on Computers* 63(8), 1854-1867 (2014).
[6] A. Pedram, and R. Geijn, Co-design tradeoffs for high-performance, low-power linear algebra architectures, *IEEE Transactions on Computers* 61(12) (2012), 1724-1736.
[7] H. Giefers, R. Polig, and C. Hagleitner, Analyzing the energy-efficiency of dense linear algebra kernels by power-profiling a hybrid CPU/FPGA system, *25th International Conference on Application-Specific Systems, Architectures and Processors*, 2014, pp. 92-99.
[8] J. Jiang, V. Mirian, K. Tang, P. Chow, and Z. Xing, Matrix multiplication based on scalable macro-pipelined FPGA accelerator architecture, *International Conference on Reconfigurable Computing and FPGAs*, 2009, pp. 48-53.
[9] W. Wang, K. Guo, M. Gu, Y. Ma, and Y. Wang, A universal FPGA-based floating-point matrix processor for mobile systems, *International Conference on Field-Programmable Technology*, 2014, pp. 139-146.
[10] Z. Jovanovic, and V. Milutinovic, FPGA accelerator for floating-point matrix multiplication, *IET Computers & Digital Techniques* 6(4) (2012), 249–256.
[11] R. Andrade, C. Huitzil, and R. Cumplido, Processor arrays generation for matrix algorithms used in embedded platforms implemented on FPGAs, *Microprocessors and Microsystems* 39 (2015), 576–588.
[12] E. H. D'Hollander, High-level synthesis optimization for blocked floating-point matrix multiplication, *ACM SIGARCH Computer Architecture News*, (2016) 74-79.
[13] https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html
[14] Y. Tan and T. Imamura, Performance evaluation and tuning of an OpenCL based matrix multiplier, *24th International Conference on Parallel and Distributed Processing Techniques and Applications*, 2018, pp. 107-113.
[15] Y. Tan and T. Imamura, An energy-efficient FPGA-based matrix multiplier, *24th IEEE International Conference on Electronics, Circuits and Systems*, 2017.
[16] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=231&No=970
[17] https://www.nvidia.com/en-gb/titan/titan-v/