Performance Evaluation and Tuning of An OpenCL based Matrix Multiplier

Yiyu Tan[1], **and Toshiyuki Imamura**[1]
[1]RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo, Japan

Abstract - *Matrix multiplication is one of the fundamental building blocks of numerical linear algebra. It requires computer systems have huge computing capability and consumes much more power as problem size is increased. In this research, an OpenCL-based matrix multiplier is presented. When data are single precision floating-points, compared with the software simulations based on the Intel MKL and OpenBLAS libraries carried out on a PC with 32 GB DDR RAM and an Intel i7-6800K processor running at 3.4 GHz, although the proposed matrix multiplier implemented by using the FPGA board DE5-NET achieves much worse performance in computation throughput, it gains 1.17 to 8.47 times, and 1.54 to 11.27 times in energy efficiency, respectively, even if the fabrication technology of the FPGA is 28 nm while it is 14 nm in the Intel i7-6800K processor. Furthermore, the performance tuning results show that the matrix multiplier obtains the best performance when the block size is 64×64 and the kernel vectorization is 4.*

Keywords: Matrix multiplication, FPGA, OpenCL

1 Introduction

Matrix computation has been widely applied in high performance computing (HPC) to solve science and engineering problems, such as medical diagnosis [1], climate prediction [2], drug discovery and molecular simulation [3], deep learning, and bio-molecular simulation in life science [4]. Matrix multiplication is at the heart of matrix computation because it is the fundamental building block of numerical linear algebra, and greatly impacts on the computation performance of HPC applications. Even now, how to improve the performance of matrix multiplication at hardware and software level is still an open problem in academic and industry. In general, matrix multiplication requires computing systems to have huge computation capacity as problem size grows. Many methods and algorithms have already been developed to speed up computation through parallel programming in supercomputers or cluster systems. The blocked algorithms divided matrices into sub-matrices to perform computation and reduce memory bandwidth requirement. Additionally, the tile algorithms divided block computations into several cores to increase computation performance through compensation with higher parallelism. And some methods speed up computation through reducing data access according to the memory hierarchy. Unfortunately, because of the power wall and memory wall problems, performance and power improvements due to technology scaling and instruction level parallelism in general-purpose processors have ended. And the guiding metric to evaluate the performance of computing systems is shifted from computing power (FLOPs: floating-point operations per second) to energy efficiency (computing power/power consumption: FLOPs/W) at post Moore's era. Therefore, how to reduce power consumption while increasing performance is one of the core concerns in future HPC systems. Nowadays, heterogeneous systems with hardware specialization or acceleration by using GPUs, FPGAs, and ASICs, offer a promising path toward major leaps to improve computation capability while achieving high energy efficiency. Particularly, FPGA, due to its high energy efficiency and reconfigurability, has been applied in data center and cloud computing in recent years [5-6].

FPGA delivers high energy efficiency through data parallelism and pipelining parallelism by using a sea of individually slow processing elements that are energy efficient per operation. In the FPGA-based solution, numerical equations are directly implemented by the configurable logic blocks, and data are kept by the D flip-flops or block memory units inside FPGA chips. By cascading hundreds of arithmetic units together, and coordinating them to work in parallel, an FPGA-based accelerator may achieve much higher computation performance than software simulations on generic computer systems. However, FPGA accelerators are usually designed by using hardware description language (HDL) in the past, such as VHDL, Verilog, and SystemVerilog. The HDL-based design flow requires designers has broad hardware knowledge, and development period is long because of system verification and debugging, such as system-level function simulation, gate-level timing simulation, system optimization and debugging. Particularly, it is a big challenge to implement and debug system interfaces to communicate with the external world, such as DDR memory controller, PCIe, and so on. Although current technologies of system on chip make such implementations and debugging easier, it is still time-consuming and requires designer having fluent experiences.

To overcome these disadvantages of the HDL-based design flow, high level synthesis is becoming popular in recent years, such as SDAccel from Xilinx, and OpenCL from Intel. The Intel FPGA SDK for OpenCL provides an exchange data between FPGA and host machine. It allows users to abstract away the traditional HDL-based FPGA design flow, and only focus on designing the kernel by using OpenCL. The I/O interfaces and their drivers are generated by the compiler automatically. As a result, the development period is shortened significantly over the traditional HDL-based FPGA design flow. In addition, how to tune the system performance according to the constraint hardware resources inside an FPGA is one of the open issues in hardware-based acceleration on matrix multiplication. To address these, an OpenCL-based accelerator for matrix multiplication is presented in this study, and its performance is evaluated and tuned. The major contributions of this work are as follows.

(1) Detail performance evaluation of an OpenCL-based matrix multiplier implemented using the FPGA board DE5-NET, including computation throughput, power efficiency, and hardware resource consumption.
(2) Performance tuning according to the constraint hardware resources inside FPGA to investigate the optimization of the accelerator.

The remainder of this paper is organized as follows. The related works are briefly summarized in Section 2. The system design and implementation are introduced in Section 3. In Section 4, performance estimation is presented, followed by the performance tuning in Section 5. Finally, conclusions are drawn in Section 6.

2 Related works

In recent years, owing to the development of fabrication technology, an FPGA contains much more hardware resources, such as thousands of hardened floating-point processing blocks, and is becoming a promising platform for scientific computing. The FPGA-based approaches accelerate computations in matrix multiplication by exploiting spatial and temporal parallelism through circuit design techniques. Furthermore, memory units can be arranged close to arithmetic units to reduce data access overhead, data are kept by on-chip memory, and processing elements (PEs) communicate each other through routing wires instead of message passing to minimize memory bandwidth.

Zhuo et al. [7] and Dou et al. [8] proposed an accelerator based on the one-dimensional linear array architecture of PEs for matrix multiplication. Each PE consists of one multiplier, one adder, and a set of registers and buffers. By identifying the inherent characteristics of each operation, Zhuo et al. analyzed the design tradeoffs and optimized the design according to the hardware constraints, including the number of configurable slices, the available memory bandwidth, the size of on-chip memory. In later work, Wu et al. [9] developed a high performance and memory efficient matrix multiplier for dense matrices by applying transformations and optimizations on the original serial matrix multiplication algorithm to derive an I/O and memory optimized blocking algorithm for matrix multiplication. Through using the similar architectures as Zhuo and Dou, the proposed multiplier demonstrated that the required hardware resources, especially memory, were reduced significantly while the clock frequency was increased. Matam et al. [10] modified the PE architecture by adding buffers to suppress the switching activity in memory to improve energy performance.

Kumar et al. [11] presented a one-dimensional array architecture of PEs and applied the rank-1 update algorithm to schedule input data to PEs in order to fully utilize PEs, data path and existing components, and overlap data communication through I/O and computation completely to sustain the peak performance. Compared with the architectures proposed by Dou and Zhou, in which each PE communicates only with the adjacent ones, this architecture distributes the same elements of the first matrix to all PEs through broadcasting.

The main drawback of one-dimensional linear array architecture of PEs is the system parallelism, scalability, and data movement are limited. To address this, Pedram et al. [12][13] introduced a microarchitecture based on the two-dimensional linear array architecture, in which data exchange between PEs was performed through row/column broadcasting buses. And the microarchitecture was extended with a general memory hierarchy model to evaluate the tradeoffs in performance and energy efficiency. Such architecture provides benefits in scalability, addressing, and data movement over one-dimensional array architecture. The two-dimensional arrangement of PEs has been proven to be scalable relative to the ratio of problem size to local memory size [14], and different types of interconnects can be adopted to speed up data access.

There are other FPGA-based accelerators on matrix multiplication for different purpose. Heiner et al. [15] presented a FPGA-based accelerator architecture for matrix multiplication on a hybrid FPGA/CPU system to study energy efficiency, in which a circular buffer was served as a vector cache for a current block of matrix, a solver module was attached to the circular buffer to compute the dot-products for each matrix row and forward the results to a pack unit, which combined the consecutive data words into one DMA word and sent it to the put-channel. Jiang et al. [16] introduced a scalable macro-pipelined accelerator to perform floating-point matrix multiplication to exploit temporal parallelism and architectural scalability. Wang et al. [17] integrated multiple matrix accelerators with a master processor and built a universal floating-point matrix processor, which consisted of accelerators, master processor, shared matrix caches, and arbitrator. Jang et al. [18] developed new algorithms and architectures for matrix multiplication on configurable devices to reduce power consumption and latency. Z. Jovanovic et al. [19] 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.

On the other hand, high level synthesis becomes popular because of its easy system development. Andrade et al. [20] proposed a high-level synthesis approach to generate two-dimensional embedded processor arrays for matrix algorithms. The solution provided efficient data memory accesses and data transferring for feeding the PE array. Holland proposed high-level synthesis optimization strategies to maximize the utilization of the DSPs and BRAMs in blocked matrix multiplication [21]. In this research, a matrix multiplier based on OpenCL is presented, and its performance is evaluated and tuned in accordance to the constraint hardware resources inside FPGA.

3 System design and implementation

As shown in Fig. 1, a matrix multiplication consists of three loops. When both matrices are square (n×n), the computation requires $2n^3$ floating-point operations. To speed up computation, 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 like an PE array 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 read 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 BRAM inside an FPGA. In addition, the overhead to writing data from the external memory to the on-chip buffers is affected by the memory bandwidth. To address this, the blocked matrix multiplication algorithm is generally applied to divide matrices into small sub-matrices, and parallelisms are put on the sub-matrix multiplications.

```
for i=0; i<n; i++
   for j=0; j<n; j++ {
   sum=0.0;
      for k=0; k<n; k++
         sum+=A[i][k] * B[k][j];
         C[i][j] = sum; }
```

Fig. 1 Design of a matrix multiplication

3.1 System design

In this research, the blocked matrix multiplication is applied, in which matrices are divided into blocks (sub-matrices), one or more blocks are read into the local buffers at each time, and computations are carried out. Once computations for the data inside the local buffers are completed, another block data are read into to perform computation. The procedure is iterated until all the product C_{ij} are obtained. Based on the considerations mentioned above, the matrix multiplier is designed using OpenCL. As shown in Fig. 2 [22][23], both the 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 6 and 7). Before computation, matrices A and B are written into the on-board DDR memory from the host. During computation, a block of A and a block of B are read into the local buffers at each time, and the product of two blocks is carried out by a two-dimensional PE array, which is defined by the N-dimensional index space (NDRange) in the OpenCL execution model. Along with reading data from the external memory (lines 17 - 20), the block data of the matrix B is firstly transposed and then stored in the local buffer (line 20) to ensure consecutive data access during computation (line 25). The inner loop in Fig. 1 is unrolled completely by adding the directive (line 23) to instruct the OpenCL compiler to generate the system pipeline and implement parallelism by using more DSP blocks. And the parallelism of the outer loop in Fig. 1 is realized through introducing the two-dimensional PE array to perform computation. Therefore, the computation throughput will be enhanced significantly. In addition, after product of two blocks is finished, system will be stalled until new block data are fed into the local buffers (line 22) to maintain data synchronization. The calculated product is written into the on-board DDR memory (line 30), and finally transferred to the host machine.

3.2 System implementation

To verify the proposed matrix multiplier and evaluate its performance, the matrix multiplier is implemented using the FPGA board DE5-NET from Terasic, which includes an Altera FPGA Stratix V GX5SGXEA7N2F45C and 4 GB on-board DDR3 memory. The FPGA chip contains 256 DSP blocks, 234,720 ALMs, and 50 Mb M20K memory. The on-board memory is arranged at two independent banks with each being 2 GB. Before computation, the elements of 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 Fig. 2, blocks of both matrices are read into the local buffers from the on-board DDR memory at each iteration, and multiplications are carried out by the kernel. 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 Fig. 2, and not associated with the matrix scale of A and B, which only affects the number of iterations. In Fig. 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 PE 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 are specified individually, the size of local buffers and the scale of the PE array are determined by the block size. Consequently, the block size has a great impact on the required hardware resources.

OpenCL code for blocked matrix multiplication

```
__kernel void matrixmult( __global float *restrict C,
                __global float *A, __global float *B,
              int A_width, int B_width)
{
//define local storage for a block of input matrices A and B
  __local float A_local[BLOCK_SIZE][BLOCK_SIZE];
  __local float B_local[BLOCK_SIZE][BLOCK_SIZE];
 int block_x = get_group_id(0);  //define block index: row
 int block_y = get_group_id(1);  //define block index: column
 int local_x = get_local_id(0); //define local index: row
 int local_y = get_local_id(1); ////define local index: column
 // loop start and stop points
 int a_start = A_width*BLOCK_SIZE*block_y;
 int a_end = a_start + A_width - 1;
 int b_start = BLOCK_SIZE * block_x;
 float sum = 0.0f;
 for (int aa=a_start, bb=b_start; aa<=a_end; aa+=BLOCK_SIZE,
   bb+= (BLOCK_SIZE*B_width))  {
   // load the matrices into local memory, and perform B<=B^T
   A_local[local_y][local_x]=A[aa+A_width*local_y+local_x];
   B_local[local_x][local_y]=B[bb+B_width*local_y+local_x];
   // wait for the entire block to be loaded.
   barrier(CLK_LOCAL_MEM_FENCE);
   #pragma unroll
   for (int k = 0; k< BLOCK_SIZE; ++k) {
       sum += A_local[local_y][k] * B_local[local_x][k]; }
     // wait for completion of computation.
     barrier(CLK_LOCAL_MEM_FENCE);
   }
  // store result in matrix C
C[get_global_id(1)*get_global_size(0)+get_global_id(0)]=sum;
}
```

Fig. 2 OpenCL code for blocked matrix multiplication.

Table 1 presents the hardware resource consumption of the matrix multiplier and the clock frequency in the case of different block sizes when matrices A and B are 16384× 16384, the SIMD is 4, and data are single-precision floating-points. In Table 1, the multiplier consumes more hardware resources as the block size is increased, especially the DSP blocks, which are utilized to implement multipliers and adders. When the block size is 64×64, the DSP blocks are used up while the block RAMs are only consumed 32% of the capacity in the FPGA. The block size can not be increased further because of the exhausted DSP blocks. Hence, the DSP blocks limit the performance improvement in the current design. From Fig. 2, the innermost loop is unrolled fully, which means BLOCK_SIZE floating-point multiply-accumulators run in parallel. Furthermore, because the kernel vectorization (SIMD) is 4, 4×BLOCK_SIZE multiply-accumulators works parallelly in total in computation. Each single precision floating-point multiply-accumulator usually consumes one DSP block. Therefore, the number of DSP blocks inside an FPGA affects the computation throughput significantly.

Table 1. Hardware resources and clock frequency

Block size	Logic utilization (in ALMs)	RAM blocks	DSP blocks	Frequency (MHz)
8×8	58278 (28%)	441 (17%)	40 (16%)	301
16×16	66681(28%)	490 (19%)	72 (28%)	303
32×32	82939 (35%)	587 (23%)	136 (53%)	290
64×64	115397 (49%)	817 (32%)	256 (100%)	269

On the other hand, when the DSP blocks are used up, the BRAMs are only utilized 32% of the total capacity, the size of the local buffers may be enlarged to keep more matrix blocks to reduce the overhead of access the external memory. In addition, the clock frequency is reduced because of the long data path as the block size is increased. In Table 1, when the block size is increased from 32×32 to 64×64, the clock frequency is reduced from 290 MHz to 269 MHz (about 7%).

4 Performance Estimation

The matrix multipliers with different block sizes and matrix scale are implemented using the FPGA board DE5-NET, and their performance is evaluated. As a comparison, the matrix multiplication functions in the highly optimized libraries OpenBLAS and Intel MKL are executed on a PC with 32 GB RAM and an Intel i7-6800K processor running at 3.4 GHz, and their performance is estimated. The operating system of the PC is CentOS 7.0, and the compilers for MKL and OpenBLAS libraries are icc 18.0.0 from Intel and gcc 4.9.4, respectively. The compiler for the OpenCL is the Intel FPGA SDK for OpenCL 16.0. The host machine of the FPGA system is the same PC as the software simulation. The technology specification of FPGA and the PC used in the software simulation are presented in Table 2. The execution time, computation throughput, and power consumption are measured, and the related system performance is estimated. During evaluation, matrices and blocks are square (n×n).

4.1 Computation throughput

Fig. 3 and Fig. 4 show the computation throughput in the FPGA system and the software simulations on the PC when data are single precision and double precision floating-points, respectively. In the FPGA system, the block size and SIMD are 64×64 and 4 in Fig. 3, respectively, while they are 32×32 and 2 in Fig. 4. In Fig. 3 and Fig. 4, computation throughput of the software simulations in the optimized libraries is much higher than that of the FPGA system no matter data format is single precision or double precision. Furthermore, since operations become computation-bound in the FPGA system as the matrix scale is increased, computation performance is firstly increased, and varies very small even if the problem size is further increased. In contrast, the data throughput fluctuates as the problem size is increased in the software simulations based on the MKL and OpenBLAS, especially in OpenBLAS. For example, when the matrix scale is increased from 1024×1024 to 16384×16384 in Fig. 3, the computation throughout is only increased from 135.21 GFLOPs to 137.09 GFLOPs in the FPGA system while it is changed from 417.07 GFLOPs to 513.95 GFLOPs in the OpenBLAS. In Fig. 3, when matrices are 16384×16384, the computation performance of the libraries MKL and OpenBLAS is about 4.27 (585.92/137.09) times and 3.74 times (513.95/137.09) over the FPGA system, respectively, and the related performance gains in computation throughput are expanded to 9.83 (312.07/31.73) times and 6.46 (205.11/31.73) times, respectively, when data are in double precision and matrix scales are 8192×8192.

Table 2. Technology specification

	FPGA	Software simulation
Model	Stratix V GX 5SGXEA7N2F45C2	i7-6800K
Cores	256 DSP blocks	6 cores (12 threads)
Clock frequency	About 300 MHz	3.4 GHz
On-chip memory	6.25 MB block RAMs	L1 cache: 384 KB L2 cache: 1.5 MB L3 cache: 15 MB
Programming language	OpenCL	C
Fabrication	28 nm	14 nm

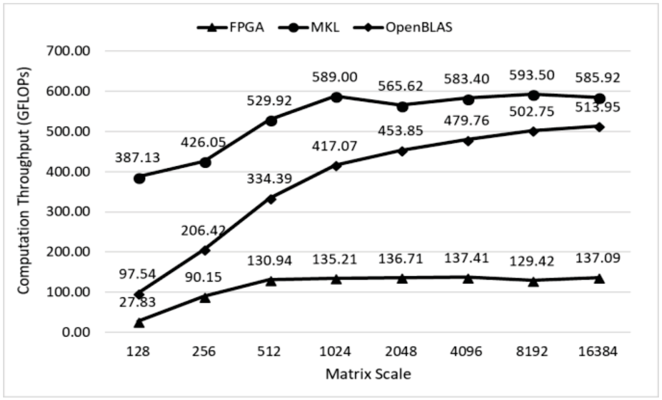

Fig. 3 Computation throughput when data are single precision

In the FPGA system, when data are in double precision, more DSP blocks are required to implement a multiplier-accumulator. Because of hardware resources constraints, the block size is reduced from 64×64 to 32×32, and the SIMD is changed from 4 to 2. This results in the computation throughput being decreased from about 137 GFLOPs in single precision data to about 32 GFLOPs in double precision data.

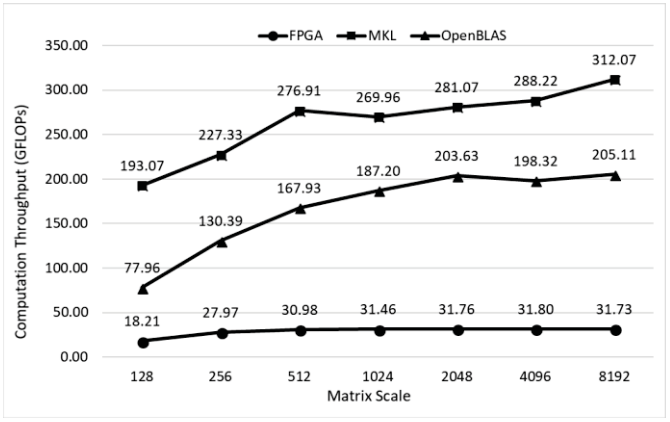

Fig.4 Computation throughput when data are double precision

4.2 Energy efficiency

To estimate the energy efficiency, the input current and voltage of the host machine in the FPGA system and the PC used for simulation are measured every 200 ms by using a digital multimeter PC720M from the Sanwa when the FPGA system and the PC are idle and active. The power consumption is estimated by multiplying the voltage and the current difference. And the energy efficiency is calculated by using equation 1.

$$P_{efficiency} = \frac{E_{computation\_throughput}}{V \times (I_{active} - I_{idle})} \qquad (1)$$

where E is the computation throughput, V is the voltage, and I is the current. To measure the current difference, the multimeter is applied to measure the current flowing through the power cable of the host PC in case the PC is idle and the simulation or the FPGA board system is active. The I_{active} and I_{idle} are the average of the measured values.

Fig. 5 and Fig. 6 show the energy efficiency of the FPGA system and the software simulations when data are single and double precision floating-points. As shown in Fig. 5 and Fig. 6, the energy efficiency is changed relatively smoothly as the problem size is increased in the libraries MKL and OpenBLAS. However, it is fluctuated in the FPGA system. The highest and lowest energy efficiency occur at the matrix scale being 512 and 16,384, which are 50.03 GFLOPs/W and 9.01 GFLOPs/W in single precision data, and 8.19 GFPLOs/W and 2.43 GFLOPss/W in double precision data, respectively. As shown in Fig. 3 and Fig. 4, the computation throughput is changed very small after the problem size is larger than 1,024 in the FPGA system while it is increased significantly when the matrix scale is increased from 128 to 512. Hence, the FPGA system achieves the highest and lowest energy efficiency in the case of the problem size being 512 and 16,384, respectively. From Fig. 5, we can observe that the speedup gain of the energy efficiency of the FPGA system over the software simulations based on the MKL and OpenBLAS ranges from 1.17 to 8.47 times, and 1.54 to 11.27 times, respectively, even if the fabrication technology of the Altera Stratix V (28 nm) falls much behind that of the i7-6800K processor (14 nm). When data format is double precision, although the computation throughput of the FPGA system is much worse than the software libraries, it still provides a little better performance in energy efficiency.

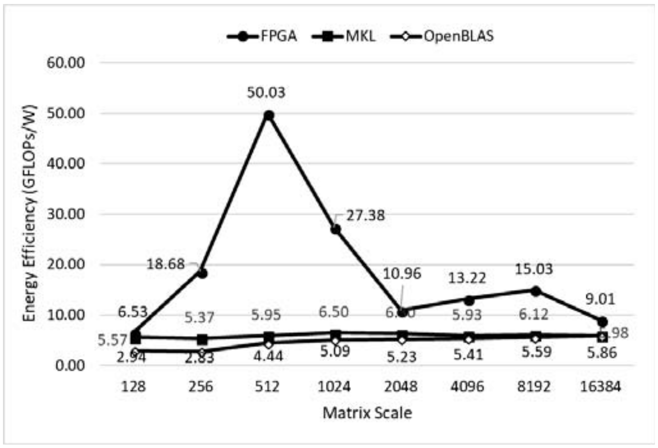

Fig. 5 Energy efficiency when data are single precision

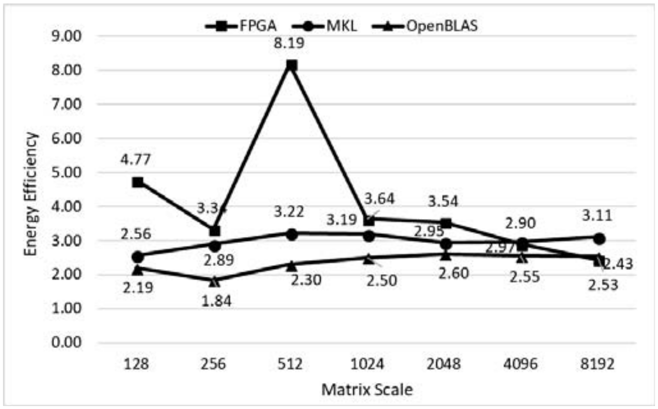

Fig. 6 Energy efficiency when data are double precision

5 Performance tuning

As mentioned in Section 3, the block size and kernel vectorization affect hardware resource consumption significantly. In general, large block size and SIMD enhance system performance, but they are limited by the hardware resources inside FPGA. To investigate the impacts of the constraint hardware resources on the performance, the matrix multipliers with different block sizes are implemented, and their performance is evaluated. During the investigation, the matrices A and B are both 16384 × 16384, data are single precision floating-points.

5.1 Block size

Fig. 7 shows the execution time and computation throughput of the matrix multiplier when the SIMD equals 4. In Fig. 7, as the block size is increased, the PE array becomes larger, and the number of iterations and data access to the external DDR memory are reduced. As a result, the overhead of data access is shortened, and computations to get the product of two blocks are speeded up. These results in the decreasing of execution time and increasing of computation throughput. When the block size is increased from 8×8 to 64×64, the execution time is reduced from about 58 minutes to 1 minute while the computation throughput is increased from 2.5 GFPLOPs to 137.09 GFLOPs. Thus, the block size has a great impact on the system performance. As discussed in Section 3, the block size is determined by the hardware resources inside an FPGA, especially DSP blocks and BRAMs. From Table 1, we can know that the number of DSP blocks constraints system performance.

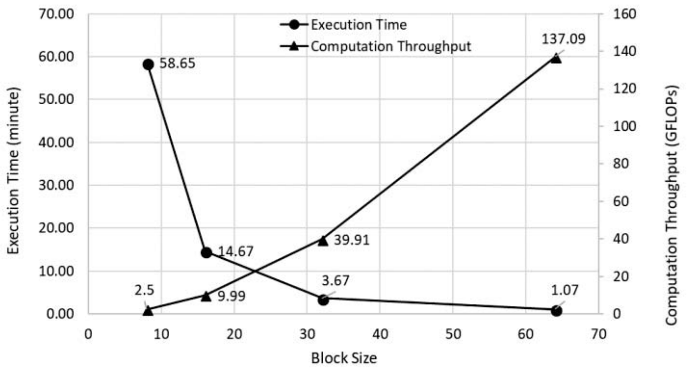

Fig. 7 Execution time and computation throughput

5.2 SIMD

Kernel vectorization allows multiple work-items within a work group to execute in a single instruction multiple data fashion to achieve higher computation throughput. The offline compiler can be directed to translate multiple scalar arithmetic operations in the kernel, such as addition or multiplication, to a SIMD operation during system synthesis. Table 3 presents the computation throughput in the case of different SIMDs and block sizes. Large SIMD value usually achieves higher computation throughput. When the SIMD is fixed, the computation throughput becomes larger as the block size is increased. In Table 3, when the SIMD is 4 and the block size is 64, the FPGA system achieves the highest computation throughput.

Table 3. SIMD and computation throughput

SIMD	Block size	Computation throughput (GFLOPs)
2	128	108.49
4	8	2.5
4	16	9.99
4	32	39.91
4	64	137.09
16	16	9.99

6 Conclusions

Matrix multiplication is a fundamental operation of numerical linear algebra. In this research, an OpenCL-based matrix multiplier is designed and implemented using the FPGA board DE5-NET, and its performance is evaluated and tuned according to the constraint hardware resources. Compared with the software simulations based on the highly optimized numerical libraries MKL and OpenBLAS, the proposed FPGA-based matrix multiplier achieves much higher energy efficiency although the fabrication technology of the FPGA chip lags much behind that of the processors in the software simulation. In future work, the local buffer size and its effect on the system performance will be investigated, and data reuse techniques will be examined to improve system performance.

Acknowledgment

Thanks for Intel's donation of the FPGA board DE5-NET through University Program. This work was partly supported by Grant-in-Aid from Foundation for Computational Science (FOCUS).

7 References

[1] L. Wang, H. Zhang, K. Wong, H. Liu, and P. Shi.: Physiological-model-constrained noninvasive reconstruction of volumetric myocardial transmembrane potentials. IEEE Trans Bio-medical Engineering, Vol. 57, No.2, 2010, pp. 296-315.

[2] T. Miyoshi, M. Kunii, J. Ruiz, et al.: Big data assimilation revolutionizing severe weather prediction. Bulletin of the American Meteorological Society, August 2016, pp. 1347-1354.

[3] T. Liu, D. Lu, H. Zhang, et al.: Applying high-performance computing in drug discovery and molecular simulation. National Science Review 3(1), 49-63 (2016).

[4] F. Allen, G. Almasi, W. Andreoni, et al.: Blue Gene: a vision for protein science using a petaflop supercomputer. IBM System Journal 40(2), 310-327 (2001).

[5] A. Putnam, A. Caulfield, E. Chung, et al.: A reconfigurable fabric for accelerating large-scale datacenter services. the 41st annual international symposium on Computer architecture, pp. 13-24, 2014.

[6] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, S. Jiang: SDA: software-defined accelerator for large-scale DNN systems. 2014 IEEE Hot Chips Symposium.

[7] L. Zhuo, and V. Prasanna: High-performance designs for linear algebra operations on reconfigurable hardware. IEEE Transactions on Computers 57(8), 1057-1072 (2008).

[8] Y. Dou, S. Vassiliadis, G. Kuzmanov, G. Gaydadjiev: 64-bit floating-point FPGA matrix multiplication. ACM/SIGDA 13th international symposium on Field-programmable gate arrays, pp. 86-95, 2005.

[9] 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, pp. 134-137, 2010.

[10] K. Matam, H. Le, and V. Prasanna: Energy efficient architecture for matrix multiplication on FPGAs. IEEE International Conference on Field Programmable Logic and Applications, pp.1-4, 2013.

[11] V. Kumar, S. Joshi, S. Patkar, and H. Narayanan: FPGA based high performance double-precision matrix multiplication. International Journal of Parallel Programming 38, 322–338 (2010).

[12] 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).

[13] A. Pedram, and R. Geijn: Co-design tradeoffs for high-performance, low-power linear algebra architectures. IEEE Transactions on Computers 61(12), 1724-1736 (2012).

[14] V. Eijkhout: Introduction to High Performance Scientific Computing, www.lulu.com, 2015.

[15] H. Giefers, R. Polig, and C. Hagleitner: Analyzing the energy-efficiency of dense linear algebra kernels by power-profiling a hybrid CPU/FPGA system. the 25th International Conference on Application-Specific Systems, Architectures and Processors, pp. 92-99, 2014.

[16] 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, pp. 48-53, 2009.

[17] 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, pp. 139-146, 2014.

[18] J. Jang, S. Choi, and V. Prasanna: Energy and time-efficient matrix multiplication on FPGAs. IEEE Transactions on Very Large-Scale Integration Systems 13(11), 1305-1319 (2005).

[19] Z. Jovanovic, and V. Milutinovic: FPGA accelerator for floating-point matrix multiplication. IET Computers & Digital Techniques 6(4), 249–256 (2012).

[20] R. Andrade, C. Huitzil, and R. Cumplido: Processor arrays generation for matrix algorithms used in embedded platforms implemented on FPGAs. Microprocessors and Microsystems 39, 576–588 (2015).

[21] E. H. D'Hollander: High-level synthesis optimization for blocked floating-point matrix multiplication. ACM SIGARCH Computer Architecture News, pp. 74-79, 2016.

[22] Y. Tan and T. Imamura: An energy-efficient FPGA-based matrix multiplier", The 24th IEEE International Conference on Electronics, Circuits and Systems, 2017.

[23] https://www.altera.com/products/design-software/embedded-software-developers/opencl/developer-zone.html