An Evaluation of Double-Double Precision Operation for Iterative Solver Library using AVX

Toshiaki Hishinuma†, Akihiro Fujii†, Teruo Tanaka† and Hidehiko Hasegawa††

†Department of Computer Science, Faculty of Informatics, Kogakuin University, Tokyo, Japan
††Faculty of Library, Information and Media Science, University of Tsukuba
E-mail: j209098@ns.kogakuin.ac.jp

Abstract

As computing performance improves generation after generation, high precision calculation is required in many situations. One of the efficient methods to perform quadruple precision is to use Double-Double precision routines which use two double precision variables for one quadruple precision variable. The iterative solver library *"Lis"* has vectorized Double-Double precision routines with SSE2. To accelerate these routines, we implemented Double-Double precision routines of *Lis* by using AVX instructions instead of SSE2. Although AVX has theoretically twice the performance of SSE2 routines, the speedup ratio of our routines varied from 1.0 to 2.5.

The high precision operation sometimes requires large and complex programs because of the rounding error.

One of high precision operation routines is Double-Double precision operation which uses two double precision variables for one quadruple precision variable [1].

A Library of Iterative Solvers for Liner Systems (*Lis*) [2] has vectorized Double-Double precision routines with Streaming SIMD Extensions (SSE2). In order to accelerate these routines, we implemented Double-Double precision operation of *Lis* by using Intel Advanced Vector Extensions (AVX) instructions instead of SSE2. We behavior to analyze AVX [3]

First we give summary of Double-Double precision operation and AVX, second we give summary of performance of Double-Double precision vector operations from numerical tests, and finally we analyze performance of Double-Double precision vector operations.

IEEE754 quadruple precision variable consists of one bit sign part, 15 bit exponent part and 112 bit significant part. But Double-Double precision variables consist of 1 bit sign part and 11 bit exponent and 104($=52 \times 2$) bit significant part.

Double-Double precision variable's exponent and significant part is shorter than IEEE754 quadruple variable. In many cases Double-Double precision routines are faster than IEEE754 quadruple routines because there is no special quadruple hardware instruction. Fig.1 shows number of bit for Double-Double precision variable and IEEE754 quadruple precision variable.

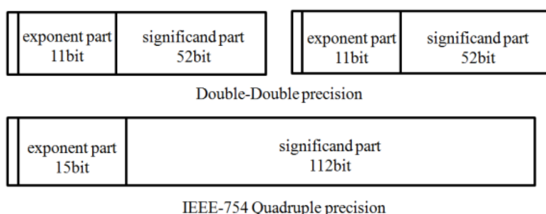

Fig.1 Number of bit for Double-Double precision

Now, *Lis* uses SSE2 for Double-Double precision routines. SSE2 is single instruction multiple data (SIMD) instructions and made by Intel in 2000. SSE2 has 16 128 bit SIMD registers. Therefore SSE2 can calculate two double precision's variables at once.

In 2011, Intel released AVX instructions on Sandy Bridge Microarchitecture. AVX has 16 256 bit SIMD registers and can calculate three operand instructions. Therefore AVX can calculate four double precision variables at once. AVX has twice the performance of SSE2 routines theoretically. As sandy Bridge has floating point adder and multiplier, it can multiply and add can be processed at once.

Table 1 shows the execution environment. For SSE2 execution and AVX execution, we use same CPU and change compile-option.

Table 2 shows experimental items of Double-Double precision vector operations. "α" and "val" are Double-Double precision values. $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ are Double-Double precision vectors.

Table 1 Execution environment

CPU	Intel Core i7 2600K 3.4GHz (16GB) Intel Sandy Bridge Microarchitecture
Compiler	Intel C/C++ Compiler 12.0.3
Compile-options(AVX)	–O3–xAVX–openmp–fp-model precise
Compile-options(SSE2)	–O3–xSSE2–openmp–fp-model precise
OS	Fedora 16

Table 2 list of operations

Name of operations	operation	Load, store
axpy	$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$	2, 1
axpyz	$\boldsymbol{z} = \alpha\boldsymbol{x} + \boldsymbol{y}$	2, 1
xpay	$\boldsymbol{y} = \boldsymbol{x} + \alpha\boldsymbol{y}$	2, 1
scale	$\boldsymbol{x} = \alpha\boldsymbol{x}$	1, 1
dot	$val = \boldsymbol{x} \cdot \boldsymbol{y}$	2, 0
nrm2	$val = \lVert\boldsymbol{x}\rVert_2$	1, 0

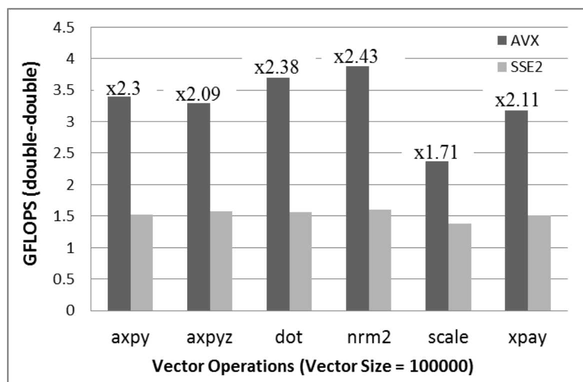

Fig.3 Performance of vector operations (4 threads)

Fig.3 shows the performances when vector size is 1.0×10^5 on four threads. Figure shows vector operation performances named as GFLOPS. It is Double-Double precision FLOPS.

From these results, the speedup ratio varied from around 1.7 to 2.4 according to the routines of arithmetic operations.

GFLOPS performance of "axpy" is higher than those of "axpyz" and "xpay". Because input and output of "axpy" is smaller than "axpyz" and "xpay" needs temporally variable to store αy. "scale" is the worst performance, because "scale" only calculates Double-Double precision additional operation.

AVX has twice the performance of SSE2 routines theoretically. Some experimental results are faster than theoretical value, because AVX can calculate three operand instructions. AVX does not need move instruction.

Fig.4 shows "axpy" performance when vector size is 1.0×10^3 - 4.0×10^5 on one thread. From these results, the speedup ratio varied from around 1.8 to 2.4. When vector size is around 1.0×10^3 - 1.3×10^5, the speedup ratio varied widely from around 2.4. But, when vector size is large, speedup ratio varied widely from around 1.7, because cache stall comes about when vector size is large.

Fig.5 shows "axpy without calculation" which has six load instructions and two store instructions. In this result, performance of memory follows a similar pattern in "axpy".

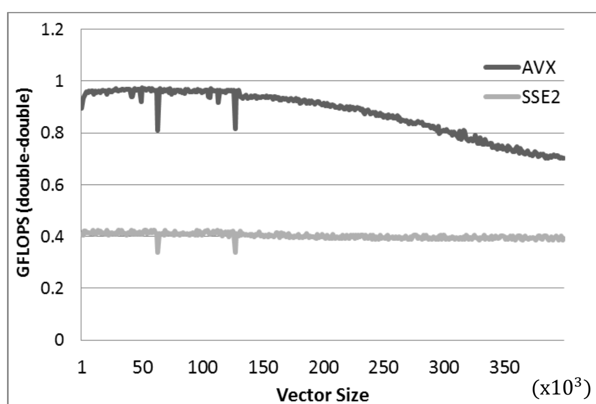

Fig.4. Performance of "axpy" operation (1 thread)

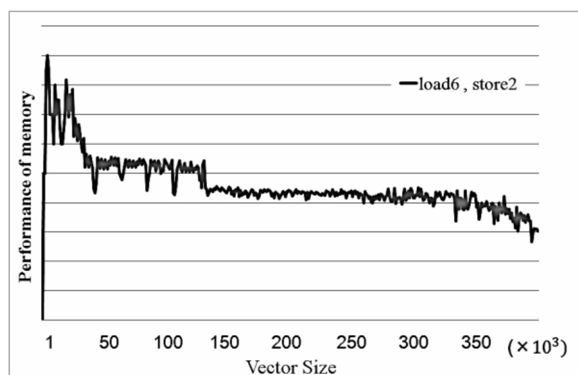

Fig.5 Performance of memory test (1 thread)

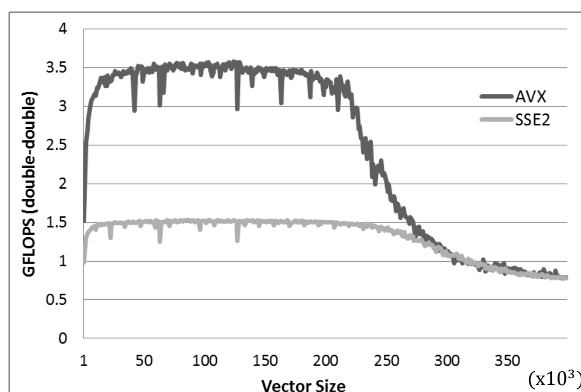

Fig.6 Performance of "axpy" operation (4 threads)

Fig.6 shows "axpy" performance when vector size is 1.0×10^3 - 4.0×10^5 on four threads. The speedup ratio varied from around 1.0 to 2.3. On four threads, when vector size is large, AVX Double-Double precision vector operation's performance becomes same as SSE2, because only has memory access increased by many threads and AVX is the same memory access as SSE2.

We implemented Double-Double precision operation of *Lis* by using AVX instructions instead of SSE2. From numeral tests, the speedup ratio varied from around 1.7 to 2.4 when vector can be stored in cache.

AVX has theoretically twice the performance of SSE2. However, in our results, AVX has more than twice performance of SSE2. This result is achieved by the reduction of move instruction because AVX uses three operand instructions.

References:

[1] Bailey, D, H.: *High-Precision Floating-Point Arithmetic in Scientific Computation*, Computing in Science and Engineering, pp.54–61 (2005).

[2] A Library of Iterative Solver for Liner Systems http://www.ssisc.org/lis/

[3] T. Hishinuma, A. Fujii, T. Tanaka, K. Asakawa, H. Hasegawa, *Acceleration of Double-Double Precision Operation for Iterative Solver Library using AVX (in Japanese)*, IPSJ-HPC12135016, pp.1-6 (2012)