PROCEEDINGS OF SCIENCE

Performance of SSE and AVX Instruction Sets

Hwancheol Jeong*, **and Weonjong Lee**
Lattice Gauge Theory Research Center, CTP, and FPRD,
Department of Physics and Astronomy,
Seoul National University, Seoul, 151-747, South Korea
E-mail: wlee@snu.ac.kr

Sunghoon Kim, and Seok-Ho Myung
Sejong Science High School, Seoul, 152-881, South Korea

SSE (streaming SIMD extensions) and AVX (advanced vector extensions) are SIMD (single instruction multiple data streams) instruction sets supported by recent CPUs manufactured in Intel and AMD. This SIMD programming allows parallel processing by multiple cores in a single CPU. Basic arithmetic and data transfer operations such as sum, multiplication and square root can be processed simultaneously. Although popular compilers such as GNU compilers and Intel compilers provide automatic SIMD optimization options, one can obtain better performance by a manual SIMD programming with proper optimization: data packing, data reuse and asynchronous data transfer. In particular, linear algebraic operations of vectors and matrices can be easily optimized by the SIMD programming. Typical calculations in lattice gauge theory are composed of linear algebraic operations of gauge link matrices and fermion vectors, and so can adopt the manual SIMD programming to improve the performance.

The 30th International Symposium on Lattice Field Theory
June 24 – 29, 2012
Cairns, Australia

*Speaker.

PoS(Lattice 2012)249

1. Introduction

SIMD is an abbreviation of the term *Single Instruction, Multiple Data streams* [1]. It describes a computer architecture that deals with multiple data streams simultaneously by a single instruction. Despite recent CPUs support SIMD instructions, plain C/C++ codes are composed of SISD (Single Instruction, Single Data streams) instructions. However, with SIMD instructions, one can sum multiple numbers simultaneously, or can calculate a product of vectors with less loops. In lattice gauge theory, the code are composed of linear algebraic operations of gauge link matrices and fermion vectors. Hence, by adopting the SIMD programming such as SSE and AVX, one can improve the performance of the numerical simulations in lattice gauge theory [2].

2. SIMD Programming

There are three methods to implement the SIMD programming: (1) inline assembly, (2) intrinsic function, and (3) vector class. Figure 1 shows SSE codes that perform the summation of two arrays using the three methods.

```
asm volatile (
  "movups %1, %%xmm0 \n\t"
  "movups %2, %%xmm1 \n\t"
  "addps %%xmm0, %%xmm1 \n\t"
  "movups %%xmm1, %0"
  :
  "=m" (C[i])
  :
  "m" (A[i]),
  "m" (B[i])
);
```

(a) Inline Assembly

```
__m128 xmm_A = _mm_load_ps(A);
__m128 xmm_B = _mm_load_ps(B);
__m128 xmm_C = _mm_add_ps(xmm_A, xmm_B);
_mm_store_ps(C, xmm_C);
```

(b) intrinsic function

```
F32vec4 vec_A;
F32vec4 vec_B;
F32vec4 vec_C;
loadu(vec_A, A);
loadu(vec_B, B);
vec_C = vec_A + vec_B;
storeu(C, vec_C);
```

(c) vector class

Figure 1: Implementation of SIMD. Codes for the summation of two arrays

- `Inline assembly`: a merit is that one can handle almost every part of a program, so that one can achieve the maximum performance of the system. One drawback is that writing assembly code is so complicated that it requires cautious handling of data transfer between CPUs and memories.
- `Intrinsic function`: a merit is that one can program it using the standard C/C++ language and so it is easy to program. A drawback is that there is no guarantee that the code is optimized to the highest level.
- `Vector class`: a merit is that it is even easier to program compared with intrinsic functions. A drawback is that the performance is even lower than the intrinsic functions.

There have been several SIMD instruction sets for different CPUs. SSE or AVX are two of them, which are supported by recent CPUs. Table 1 shows lists of processors (Intel CPUs) which

SSE3	Quad-Core Xeon 73xx, 53xx, 32xx, Dual-Core Xeon 72xx, 53xx, 51xx, 30xx, Core 2 Extreme 7xxx, 6xxx, Core 2 Quad 6xxx, Core 2 Duo 7xxx, 6xxx, 5xxx, 4xxx, Core 2 Solo 2xxx, Pentium dual-core E2xxx, T23xx
SSE4.1	Xeon 74xx, Quad-Core Xeon 54xx, 33xx, Dual-Core Xeon 52xx, 31xx Core 2 Extreme 9xxx, Core 2 Quad 9xxx, Core 2 Duo 8xxx, Core 2 Duo E7200
SSE4.2	i7, i5, i3 series, Xeon 55xx, 56xx, 75xx
AVX1	Sandy Bridge, Sandy Bridge-E, Ivy Bridge
AVX2	Haswell (will be released in 2013)

Table 1: List of SIMD instruction sets for Intel CPUs

support a specific version of SSE and AVX. The higher version instruction sets include more useful extensions which are not supported in the lower version.

SSE (Streaming SIMD Extensions) offers SIMD instruction sets for XMM registers [3]. For 64 bit system, there are 16 XMM registers (xmm0 $\sim$ xmm15) in the CPUs, and an XMM register has 128 bits (16 bytes). Thus, using XMM registers, one can process four single precision float point numbers or two double precision floating point numbers simultaneously.

AVX (Advanced Vector eXtensions) is the next generation of the SIMD instruction sets supported from Intel Sandy Bridge processors [4]. It offers instruction sets for YMM registers. Similar to the XMM registers, there are 16 YMM registers (ymm0 $\sim$ ymm15) in the CPUS. The size of ymm is twice bigger than xmm. Therefore, YMM registers make it possible to process eight single precision floating point numbers or four double precision floating point numbers, simultaneously. Figure 2 shows a diagram that explains XMM and YMM registers.

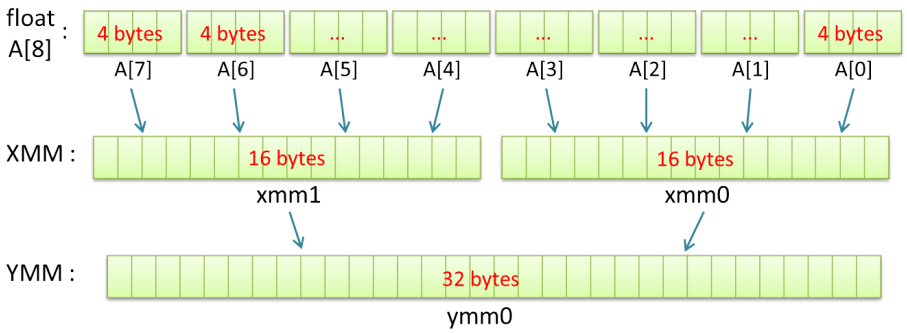

Figure 2: XMM and YMM registers

Besides the increase in size, AVX also provides an extended instruction format which allows three input arguments in contrast to two input arguments allowed for SSE. Because SSE provides SIMD instructions with only two operand, it is limited to $a = a + b$ kind of functions. However, AVX supports three operand SIMD instructions, so that $c = a + b$ kind of functions are available using AVX.

3. Optimization Scheme

3.1 Data Packing

As described in Section 2, an advantage of SIMD programming is the data packing. One can pack multiple data to a single XMM or YMM register and those data can be processed or calculated simultaneously using SSE or AVX instructions. Recent SSE and AVX instructions provide many useful SIMD functions – sum, multiply, square root, shift, *etc.*

```
for(i=0; i<ArraySize; i+=8) {
  b[i] = a[i];
  b[i+1] = a[i+1];
  b[i+2] = a[i+2];
  b[i+3] = a[i+3];
  b[i+4] = a[i+4];
  b[i+5] = a[i+5];
  b[i+6] = a[i+6];
  b[i+7] = a[i+7];
}
```

(a) C++

```
for(i=0; i<ArraySize; i+=8) {
  asm volatile (
  "movups %2, %%xmm0 \n\t"
  "movups %%xmm0, %0 \n\t"
  "movups %3, %%xmm1 \n\t"
  "movups %%xmm1, %1 \n\t"
  : "=m" (b[i]), "=m" (b[i+4])
  : "m" (a[i]), "m" (a[i+4])
  );
}
```

(b) SSE4.2

```
for(i=0; i<ArraySize; i+=8) {
  asm volatile (
  "vmovups %1, %%ymm0 \n\t"
  "vmovups %%ymm0, %0 \n\t"
  : "=m" (b[i])
  : "m" (a[i]) );
}
```

(c) AVX1

Figure 3: Codes for the simple data copy made in (a) C++, (b) SSE4.2, and (c) AVX1.

Figure 3 shows partial codes of a program which performs a simple data copy between two arrays, written in plain[1] C++, SSE4.2 and AVX1 language respectively. For SIMD code in the figure (SSE and AVX), the data copy code is implemented using inline assembly method in order to obtain maximum performance.

Table 2 shows the performances of the three different methods when the size of the array is 10^9. Without optimization option for compiler, SIMD methods (SSE4.2 and AVX1) are much faster than plain C++ code. When the maximum optimization option is applied to the C++ compiler, the SIMD method is as fast as C++. This result indicates that the compiler optimizes the given C++ code by converting to SIMD code, automatically. Indeed, we can convert the C++ object code into an assembler code using the `objdump` command in LINUX, and then we can have a look at the assembler source code. In this way, we find out that the optimization option convert the C++ code into an assembler code using the SIMD instruction sets automatically.

[1]Here *plain* C/C++ denotes a normal C/C++ language without SSE or AVX. Since SSE or AVX are also implemented in C/C++ code, we use *plain* to distinguish those programming methods from the normal C/C++ language.

We also find out that, regardless of the optimization, performances of SSE4.2 and AVX1 are almost the same. This indicates that use of AVX1, i.e., YMM registers, does not improve the speed of data transfer.

	C++	SSE4.2	AVX1
no opt.	163	94.5	97.7
max. opt.	75	71	75

Table 2: CPU clocks required for simple data transfer of array of 10^9 size, using the codes given in Figure 3 in units of 10^4 clocks. Here 'no opt.' corresponds to the results that the optimization option is turned off with the C++ compiler. The 'max. opt.' corresponds to the results with the maximum optimization option turned on. We use Intel Core i7-3820 Sandy Bridge-E with Fedora 17 of kernel 3.5.2-3.fc17. The compiler is GCC 4.7.0.

3.2 Data Reuse

In the previous example of data transfer, we find that the C++ optimization option makes the program to use the SIMD instructions. However, one can increase the performance of the SIMD code using data reuse. In the inline assembly, a programmer has a full control over registers. Thus, if there are some data which are repeatedly used, one can reuse existing register, so that one may remove unwanted data transfer.

```
for(i=0; i<LoopNum; i++) {
  c[0] = a[0] + b[0];
  c[1] = a[1] + b[1];
  c[2] = a[2] + b[2];
  c[3] = a[3] + b[3];
  c[4] = a[4] + b[4];
  c[5] = a[5] + b[5];
  c[6] = a[6] + b[6];
  c[7] = a[7] + b[7];
}
```

(a) C++

```
asm volatile (
   ⋮
"LOOP: \n\t"
"addps %%xmm0, %%xmm1 \n\t"
"addps %%xmm2, %%xmm3 \n\t"
"dec %%ecx \n\t"
"jnz LOOP \n\t"
   ⋮
```

(b) SSE4.2

```
asm volatile (
   ⋮
"LOOP: \n\t"
"vaddps %%ymm0, %%ymm1, %%ymm2 \n\t"
"dec %%ecx \n\t"
"jnz LOOP \n\t"
   ⋮
```

(c) AVX1

Figure 4: Codes for simple summation made in (a) C++, (b) SSE4.2, and (c) AVX1.

Figure 4 shows part of the codes that does a simple summation, written in C++, SSE4.2, and AVX1. To illustrate the advantage of data reuse, we repeat summations many times. Since the same data are used repeatedly, the SIMD codes (SSE4.2 and AVX1) remove unwanted data transfer of reloading the same variables, and reuse the existing data in registers.

Table 3 compares the performances of the three methods by performing a simple summation 10^9 times over the same data. We find out that regardless of the compiler optimization, the SIMD methods (SSE4.2 and AVX1) are significantly faster by an order of magnitude than C++. Furthermore, AVX1 is much faster (almost 3 times) than SSE4.2. This result indicates that by adjusting the AI (arithmetic intensity) ratio[2] using such an optimization method as data reuse and trading the data transfer with the floating point calculation by SU(3) reconstruction, we can increase the performance of the SIMD programming dramatically.

	C++	SSE4.2	AVX1
no opt.	732	83.7	26.9
max. opt.	317	78.8	26.9

Table 3: CPU clocks required for a simple summation using the codes given in Figure 4 in units of 10^4 clocks. The index convention is the same as in Table 2.

3.3 Asynchronous Data Transfer

Data transfer from the data memory to the registers is a slow process. As discussed in Subsection 3.1, the gain of using XMM and YMM registers for the data transfer is only a factor of 1.2. Hence, in a real code, it is hard to obtain the full advantage of the data packing. Fortunately, the overload of the data transfer can be minimized by using the asynchronous data transfer method. Asynchronous data transfer is a technique that performs calculation and data transfer, simultaneously.

SSE and AVX instructions provide some basic prefetching methods. Prefetching is a technique which pre-loads data to the cache memory, before CPU initiates the calculation. However, the prefetching in SSE and AVX does not support a full control over the memory caching, but only gives hints to CPU about the memory caching. In other words, it does not force data to be preloaded to the cache memory, but just give information on which data hope to be pre-loaded. This prefetching method does not work successfully because there are many background processes from OS or other applications to be handled with higher priority from the standpoint of CPU.

```
   ⋮
"LOOP: \n\t"
"prefetch1 0x200(%%rax) \n\t"
"movups (%%rax), %%xmm0 \n\t"
"movups %%xmm0, (%%rdx) \n\t"
"movups 0x10(%%rax), %%xmm1 \n\t"
"movups %%xmm1, 0x10(%%rdx) \n\t"
   ⋮
```

(a) SSE4.2

```
   ⋮
"LOOP: \n\t"
"prefetch1 0x200(%%rax) \n\t"
"movups (%%rax), %%ymm0 \n\t"
"movups %%ymm0, (%%rdx) \n\t"
"movups 0x20(%%rax), %%ymm1 \n\t"
"movups %%ymm1, 0x20(%%rdx) \n\t"
   ⋮
```

(b) AVX1

Figure 5: Codes for data copy using prefetching techniques.

[2]Here, we use the standard definition of AI, which is the ratio of the amount of floating point calculation and the amount of data transfer.

Figure 5 shows SIMD codes of data transfer with prefetching method. The results presented in Table 4 shows that the prefetching method improves about 1.5% for SSE and 5% for AVX method. Hence, we need a more powerful asynchronous data transfer method.

	C++	SSE4.2	AVX1
no prefetching	75	71	75
512 bytes prefetching		69.9	70.9

Table 4: CPU clocks required for data transfer of array of 10^9 single precision floating point numbers with prefetching method. We use the codes in Figure 5. Units are 10^4 clocks.

4. Conclusion

Recent CPUs support the SIMD instructions such as SSE and AVX. The SIMD instruction sets provide methods of parallel processing in a single CPU level. The standard C/C++ compilers support those SIMD instructions with optimization options. It turns out that one can achieve significantly higher performance by programming SIMD codes using the inline assembly as demonstrated in this paper. The optimization techniques of SIMD programming such as data packing, data reuse, and asynchronous data transfer can be easily applied to the physics code in lattice gauge theory.

5. Acknowledgement

The research of W. Lee is supported by the Creative Research Initiatives Program (2012-0000241) of the NRF grant funded by the Korean government (MEST). W. Lee would like to acknowledge the support from the KISTI supercomputing center through the strategic support program for the supercomputing application research [No. KSC-2011-G2-06]. Computations were carried out in part on QCDOC computing facilities of the USQCD Collaboration at Brookhaven National Lab, on GPU computing facilities at Jefferson Lab, on the DAVID GPU clusters at Seoul National University, and on the KISTI supercomputers. The USQCD Collaboration are funded by the Office of Science of the U.S. Department of Energy.

References

[1] http://en.wikipedia.org/wiki/SIMD

[2] M. Lüscher, Nucl. Phys. Proc. Suppl. **106**, (2002), 21-28 ; [arXiv:hep-lat/0110007].

[3] http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

[4] http://en.wikipedia.org/wiki/Advanced_Vector_Extensions