Arithmetics on Intel's Sandy Bridge and Westmere CPUs: not all FLOPs are created equal

Andrey Vladimirov
Stanford University
for Colfax International

April 30, 2012

Abstract

This paper presents a new arithmetic efficiency benchmark and uses it to compare the Intel Sandy Bridge E5-2680 CPU to the Intel Westmere X5690 CPU performance. The efficiency is measured for single and double precision floating point operations: addition, multiplication, division, square root and the exponential function, and for 32- and 64-bit integer operations: addition, multiplication and division. The SSE2 and AVX instruction sets, as well as scalar operations, in single-threaded and multi-threaded modes are covered. This benchmark eliminates the effects of memory bandwidth and latency by fitting the calculation in the L1 cache. The bandwidth of the L1 cache and main memory (RAM) are estimated for reference, and the LINPACK benchmark result is reported.

Results show that the E5-2680 CPU performs floating point addition and multiplication dramatically faster (up to 2.6x) than the X5690 model. However, the floating point division and square root are the new model's weak spots. AVX floating point operations addition and multiplication are up to 2.0x faster than the SSE2; however, AVX provides no performance gain for division and square root. 32-bit integer arithmetic operations, despite the lack of AVX integer intrinsics, are up to 3.5x faster on E5-2680. At the same time, the Sandy Bridge CPU showed a 1.15x better L1 cache performance and 2.4x greater memory bandwidth than the Westmere model.

These results lead to the conclusion that the edge of the 8-core, 2.70 GHz Sandy Bridge CPU over the 6-core, 3.46 GHz Westmere processor will be most significant in both single and double precision for linear algebra and other tasks based on addition and multiplication. Re-compilation of codes performing addition and multiplication-based tasks with AVX intrinsics instead of SSE2 should lead to additional performance benefits on Sandy Bridge. However, CPU-bound calculations heavily using the division operation and transcendental functions are likely to experience a smaller speedup from using the Sandy Bridge processor in place of Westmere. Likewise, they will benefit less from the migration from SSE2 to AVX.

Contents

1 Introduction

Benchmarks of arithmetic capabilities typically focus on the ability of computers to solve certain test problems, like linear algebraic transformations. For example, the LINPACK benchmark is an industry standard in high performance computing, including the Top500 list. This benchmark solves a system of linear algebraic equations, predominantly employing floating point addition and multiplication operations, along with some division and comparison. Therefore, the number of floating point operations per second (FLOP/s) reported by LINPACK is a measure of the performance of a combination of computing operations.

For the purposes of designing and tuning scientific computing codes, it is helpful to know exactly how long any given computing operation takes. In addition, it is important to know the impact of choosing single or double precision and of using scalar or vector (i.e., SIMD) operations. Comprehensive benchmarks like this have been reported, e.g., by Agner[1]. His reports show the tabulated latency and micro-operation throughput per cycle for a number of architectures. It may not be clear to a non-expert in microprocessor architecture how to translate these values into to practical performance metrics, and how to achieve this performance in a real-world application written in C/C++, rather than in the assembly code.

This paper is published in an effort to provide transparency of the arithmetic capability measurements for CPUs: how this performance can be realized in practice without employing the assembly code, what the performance is in terms of the number of operations per second for specific models of CPUs, and what it all means for real-world applications.

[1] http://www.agner.org/optimize/#manual_instr_tab

1.1 The CIAO Benchmark

This paper introduces an original benchmark[2] designed to test the arithmetic capabilities of CPUs for specific floating point and integer operations. The code of the benchmark relies on the compiler to automatically implement code vectorization. This is done by following the guidelines outlined in a number of widely available publications, including a recent Colfax Research paper 'Auto-vectorization with the Intel Compilers: is Your Code Ready for Sandy Bridge and Knights Corner?'. These guidelines are, in essence, maintaining data locality, ensuring proper address alignment, and providing compiler optimization hints.

How does one design a C code that will get auto-vectorized to yield the maximum performance of vector intrinsics? Suppose we have arrays `float xx[m]` and `float yy[m]`, and we want to test the performance of floating point addition for the operation `yy[i]+=xx[i]`. An obvious, albeit incorrect, approach would be to allocate the arrays on a properly aligned boundary[3] and form a loop like this:

```
1  for (int t=0; t<nswp; t++) {
2  #pragma simd
3  #pragma vector aligned
4    for (int i=0; i<m; i++)
5      yy[i]+=xx[i];
6  }
```

Note that this is not an optimal solution, however, it illustrates the basic idea of the method. The outer loop is necessary to run the calculation multiple times in order to allow sufficient timer ticks between the start and the end of the `t`-loop. The array size, `m`, should be small enough that all the data fit into the CPU's L1 cache. This way, we eliminate the latency and bandwidth of the L2 cache and the RAM from the benchmark. At the same time, `m` must be large enough, so that the overhead of loop setup is negligible compared to the calculation time. The choice of values `m` and `nswp` is discussed at the end of this section.

The above approach will not yield optimal results, because for each vectorized addition instruction, the code will have to load the data into the vector registers and then store the result into memory (or L1 cache). A better approach would be to re-use the values of `xx[i]` and `yy[i]` loaded into the registers for multiple addition operations, so that the latency of the load/store operations is masked, or becomes negligible. The code listing below demonstrates the improved approach:

```
1  double benchmark_flops_add(const int m, const int nswp,
2                             REAL* restrict xx, REAL* restrict yy) {
3    const double tstart=omp_get_wtime();
4    for (int s=0; s<nswp; s++) {
5  #ifndef SCA
6  #pragma simd
7  #pragma vector aligned
8  #else
9  #pragma novector
10 #endif
11     for (int i=0; i<m; i++)
12       for (int r=0; r<16; r++)
13         yy[i]+=xx[i];
14   }
15   const double tend=omp_get_wtime();
16   return tend-tstart;
17 }
```

In this code, the preprocessor macro `REAL` is set to either `float`, or `double`, in order to benchmark single and double precision operations, respectively. The macro `SCA` may be defined to to suppress auto-vectorization and measure the performance of scalar operations. The `#pragma simd` statement indicates to the compiler that the `i`-loop should be vectorized. The inner `r`-loop wil, be unrolled by the compiler, because the upper bound is known at compile time. I have verified it by manually unrolling the `r`-loop only to see identical performance results. The value of 16 for the upper bound in `r` is chosen empirically: greater values of the upper bound do not lead to higher performance benchmarks on the tested systems.

[2] Pilot name CIAO, which stands for Colfax Individual Arithmetic Operations benchmark

[3] See the above mentioned paper for an illustration of the data alignment and related techniques.

The CIAO benchmark uses the method shown above, employing automatic vectorization and multiple arithmetic operations performed on the same data. The benchmark functions for other arithmetic operations are designed in a similar fashion and shown in the Appendix. The advantage of the auto-vectorizable code is in its portability across instruction sets. Indeed, scalar, SSE and AVX executables can be obtained by recompiling it with the arguments '`-DSCA -no-vec`', '`-xSSE4.2`' and '`-xAVX`', respectively. It is also ready for future instruction sets, such as AVX2.

However, compiler optimizations may interfere with the performance results. In order to verify that the auto-vectorized code produces an accurate benchmark, I have reproduced the same results with explicitly vectorized codes. For example, a single precision AVX code looks like in the following listing:

```
    for (int i=0; i<m; i+=8) {
      __m256 ymm0=_mm256_load_ps(&xx[i]);
      __m256 ymm1=_mm256_load_ps(&yy[i]);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      ymm1=_mm256_mul_ps(ymm1, ymm0);
      _mm256_store_ps(&yy[i], ymm1);
    }
```

Note that the auto-vectorized code must be compiled with the arguments '`-fp-model source`' in order to get identical results with the explicitly vectorized code.

In this work, arithmetic performance is defined as the number of floating point operations per second (FLOP/s) or integer operations per second (IOP/s):

```
const double t=benchmark_flops_add(m, nswp, xx, yy); // Calculation time in seconds
const double gflops=1e-9*16*m*nswp/t; // Performance in GFLOP/s
```

Here the factor $1e-9$ converts the operation rate from FLOP/s to GFLOP/s, the factor 16 accounts for sixteen addition operations performed in the body of the inner loop, and `m` and `nswp` account for the number of loop iterations performed in the course of time `t`.

For parallel runs, OpenMP threads are spawned, and each thread calls the benchmark function on the respective pre-allocated arrays. The performance is then defined via the average calculation time:

```
double avtime=0;
#pragma omp parallel reduction(+: avtime)
{
#pragma omp barrier
  const double tthisthread=
    benchmark_flops_add(m, nswp, xx[omp_get_thread_num()], yy[omp_get_thread_num()]);
  avtime+=tthisthread/omp_get_max_threads();
}
const double gflops=1e-9*16*m*nswp/avtime*omp_get_max_threads(); // Performance in GFLOP/s
```

In practice, when benchmarks are run, `m` and `nswp` should be chosen empirically, i.e., by testing different values and picking the ones that provide the best performance. I have found that on the tested systems, the optimal performance is achieved for `m` from 2^6 to 2^{11} in single precision. As for the value `nswp`, it only has to be large enough for an accurate benchmark, because the performance plateaus for `nswp`$\gtrsim 2^{14}$.

1.2 Supplementary Benchmarks: STREAM, LINPACK

The STREAM benchmark[4] was used to estimate the rate of copying data from/to the RAM. The measured bandwidth for the `COPY` test was used in order to estimate the number of floating point (or integer) numbers that can be fetched from, and written back to, the RAM per second. In order to convert the STREAM bandwidth in MBytes/s into a value compatible with our GFLOP/s performance measurements, the following formula was used:

```
1 const double gflops=(bandwidth/1024.0)/sizeof(REAL);
```

Here `REAL=float` or `REAL=double` for single and double precision tests, respectively.

Additionally, in order to compare the CIAO results with other well-known tests, the LINPACK benchmark optimized for Intel CPUs[5] was used.

Results of STREAM and LINPACK are provided in Section 2, along with the CIAO benchmark.

1.3 Tested System Configuration

Two systems were benchmarked in this work:

1. a two-way machine with 6-core Westmere X5690 CPUs at 3.46 GHz (max turbo frequency 3.73 GHz) and 12 GB of 1333 MHz DDR3 RAM in 2 GB modules, and
2. a two-way machine with 8-core Sandy Bridge E5-2680 CPUs at 2.70 GHz (max turbo frequency 3.50 GHz) and 64 GB of 1333 MHz DDR3 RAM in 8 GB modules.

In order to bind the execution of all benchmarks to a single CPU, the `numactl` tool was used with the following arguments: `--cpunodebind=0 --membind=0`. The environment variable `KMP_AFFINITY` was set to `compact`. Both systems were running CentOS 6.2 with the Linux kernel version number `2.6.32-220.el6.x86_64`.

The code was compiled using the Intel C++ compiler from the Intel Parallel Studio XE 12.1.3. In order to compile the code for AVX, SSE2 or scalar instructions, the compiler was given the arguments `-xAVX`, `-xSSE4.2`[6] and '`-DSCA -no-vec`', respectively. The latter defines an internal macro, which disables auto-vectorization (see Appendix). The optimization level was set to `-O2`, which enables automatic vectorization, unless `-no-vec` is used. Additionally, `-fp-model source` was required in order to produce results consistent with the explicitly vectorized code results. In practical applications, the default value of the floating point model in the Intel compiler may lead to better optimization and higher performance of some arithmetic operations.

[4] http://www.cs.virginia.edu/stream/

[5] http://software.intel.com/en-us/articles/intel-math-kernel-library-linpack-download/

[6] All instructions used in this research are available in SSE2, however, the `-xSSE4.2` was used for future-proofing.

2 Results

2.1 Serial (Single-threaded) Performance: Sandy Bridge

The benchmark results for serial (single-threaded) code on the Sandy Bridge E5-2680 CPU are shown in Figure 1. All measurements are obtained with the original CIAO benchmark described in Section 1.1, except the 'Copy to/from RAM' test, which is the result of the STREAM benchmark converted to compatible units (see Secion 1.2). The single-threaded 64-bit LINPACK benchmark with the problem size and the leading dimension equal to 8192 performed at 24.6 GFLOP/s on this system.

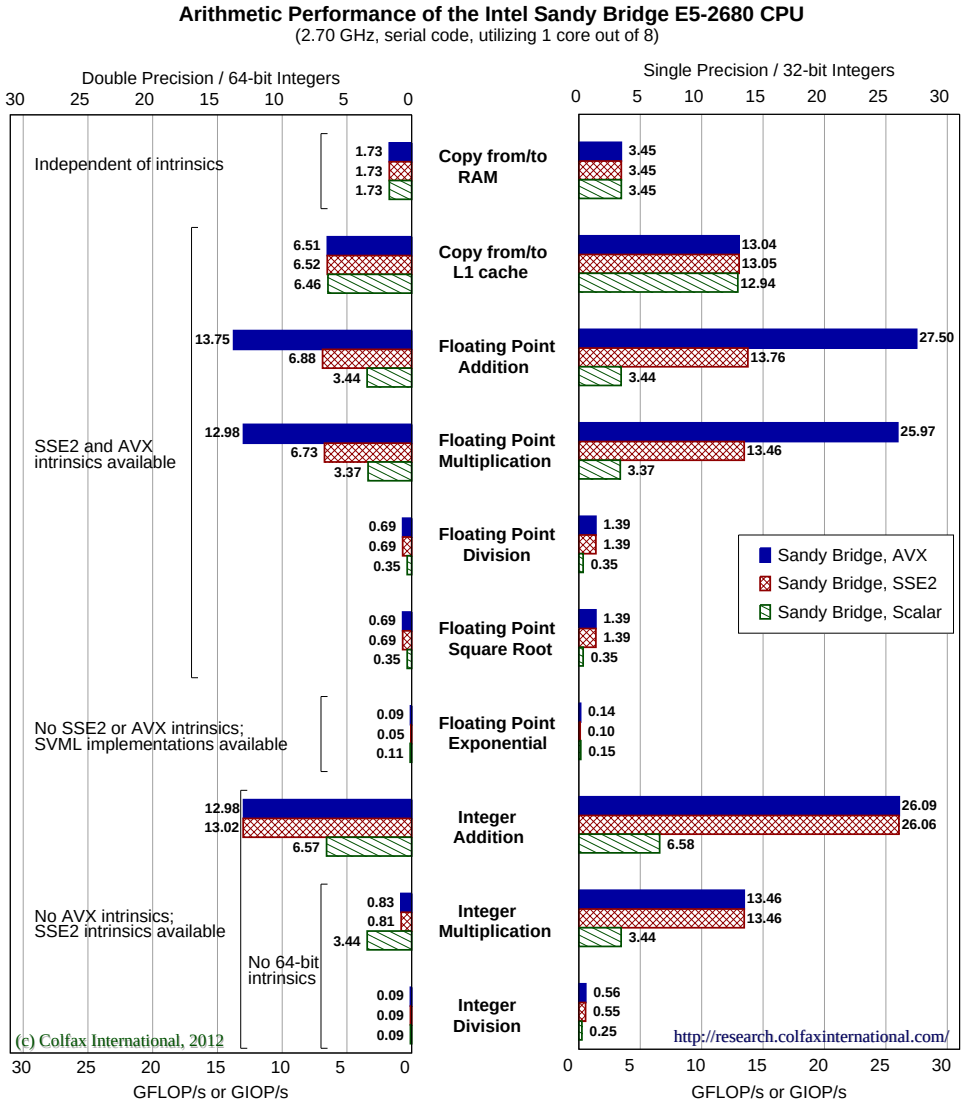

Figure 1: Scalar, SSE2 and AVX operations benchmarked on the Sandy Bridge E5-2680 CPU with a serial (single-threaded) code. The AVX instruction set does not support integer operations, and auto-vectorized code compiled with `-xAVX` falls back to SSE2. SSE2 intrinsics are not available for the multiplication and division of 64-bit integers. The exponential function is not available in either instruction set, and a vectorized implementation of the Intel SVML is used; its execution time, in general, depends on the value of the argument.

2.2 Serial (Single-threaded) Performance: Westmere

The Westmere X5690 CPU performance results for serial (single-threaded) execution are shown in Figure 2. Like for the Sandy Bridge system, these measurements are obtained with our original CIAO benchmark described in Section 1.1, except the 'Copy to/from RAM' test, which is the result of the STREAM benchmark converted to compatible units (see Secion 1.2). The single-threaded 64-bit LINPACK with the problem size and the leading dimension equal to 8192 was clocked at 12.7 GFLOP/s on this system.

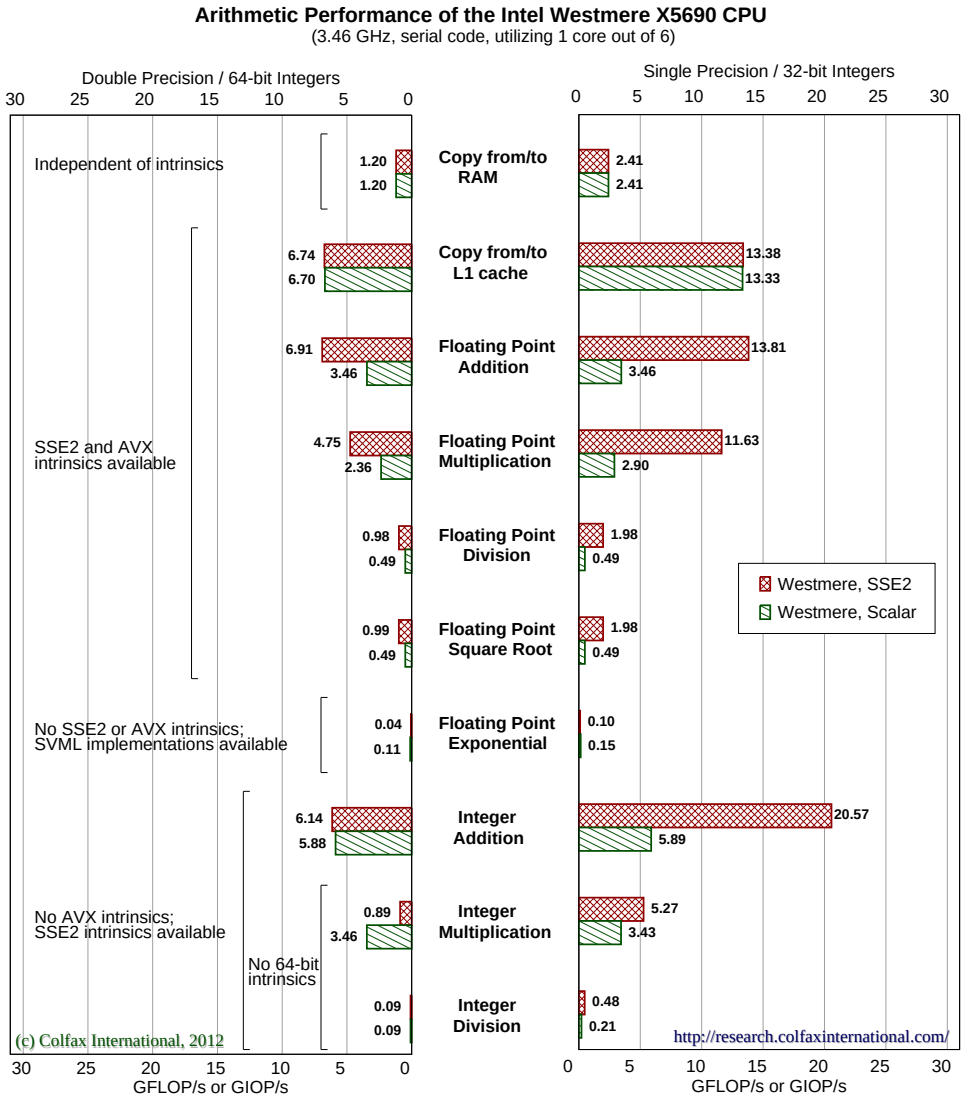

Figure 2: Scalar and SSE2 operations benchmarked on the Westmere X5690 CPU with a serial (single-threaded) code. The AVX instruction is not supported in the Westmere architecture. SSE2 intrinsics are not available for the multiplication and division of 64-bit integers. The exponential function is not available in either instruction set, and a vectorized implementation of the Intel SVML is used; its execution time, in general, depends on the value of the argument.

2.3 Parallel (Multi-threaded) Performance: Sandy Bridge vs Westmere

The parallel (multi-threaded) performance results for the Sandy Bridge E5-2680 and Westmere X5690 CPUs are combined in Figure 3. Hyperthreading was disabled in both systems. The number of threads is set equal to the number of CPU cores. The `numactl` library is used to bind the threads and memory to a single CPU on these dual-socket machines. The multi-threaded LINPACK benchmark with the problem size and the leading dimension equal to 16384 performed at 157.7 GFLOP/s on the E5-2680 and 71.8 GFLOP/s on the X5690 system.

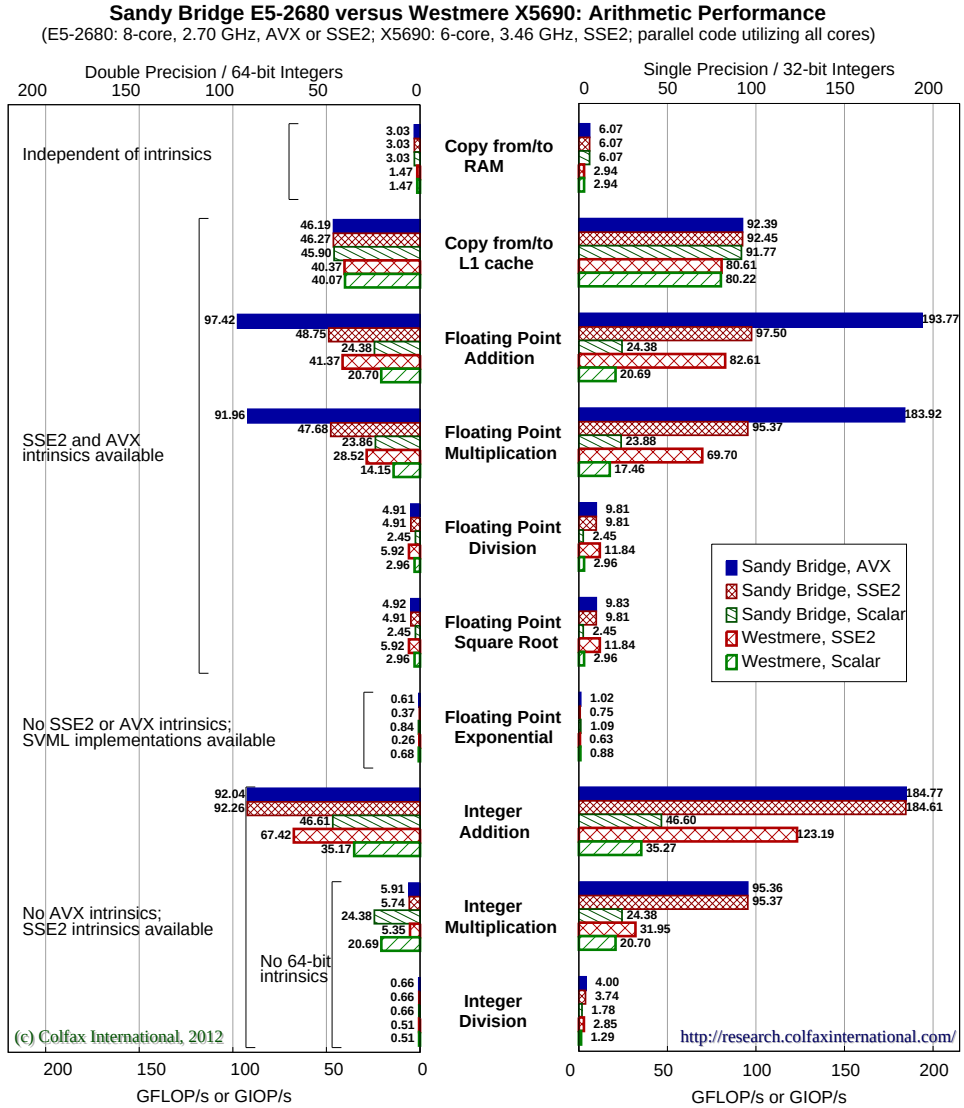

Figure 3: Scalar, SSE2 and AVX operations benchmarked on the Sandy Bridge E5-2680 CPU with a serial (single-threaded) code. The AVX instruction is not supported in the Westmere architecture. The AVX instruction set does not support integer operations, and auto-vectorized code compiled with `-xAVX` falls back to SSE2. SSE2 intrinsics are not available for the multiplication and division of 64-bit integers. The exponential function is not available in either instruction set, and a vectorized implementation of the Intel SVML is used; its execution time, in general, depends on the value of the argument.

3 Discussion

Double Precision vs Single Precision

Good news for codes utilizing scalar arithmetics is that in both Westmere and Sandy Bridge tests, most *scalar* double precision operations scored as many GFLOP/s as their single precision counterparts (see Figure 3, green bars). That means, there is no performance penalty for using double precision, if the code is already operating in a sub-optimal regime due to the lack of vectorization[7]. The exceptions is the exponential function, for which no intrinsic is available. Likewise, scalar 64-bit arithmetics performs as many GIOP/s as scalar 32-bit arithmetics, except for integer division, which is 2.8 times slower for 64-bit integers than for 32-bit.

However, the practical performance of double precision scalar calculations is not determined solely by the arithmetic operations rate. In practice, performance will be additionally affected by the memory and cache traffic, which is doubled by going from single to double precision.

Addition and Multipication vs Division and Square Root

It is evident from Figures 1, 2 and 3 that some operations perform much faster than other, especially when vector instructions are used.

- *Floating Point*: When arithmetically intensive codes can be (auto-)vectorized, it is absolutely worth doing so, especially with the AVX instructions. In double precision, the performance gain for addition and multiplication is almost a factor of 2x with SSE2, and 4x with AVX; in single precision, it is almost a factor of 4x with SSE2, and 8x with AVX. See Figure 3, red and blue bars (SSE and AVX) compared to green bars (scalar).

  However, here is where the performance burden of double precision will be painful: the division operation and square root. While SSE2 accelerates double precision division and square root twofold as it should, AVX brings no additional performance gain. In fact, the Sandy Bridge CPU scored slightly lower than Westmere for these operations in SSE2.

- *Integer*: Speaking of integer arithmetics, we only need to consider SSE2, because the AVX instruction set does not support integer operations. SSE2 does accelerate integer addition and 32-bit integer multiplication. However, for 64-bit integers, intrinsics are not available, and an attempt of the compiler to vectorize the code resulted in a performance drop.

  Going to 64-bit numbers from 32-bit may be painful for codes heavily using a lot of division and multiplication: there are no SSE2 intrinsics for 64-bit integer division or multiplication, and code vectorization is impractical in this case.

AVX vs SSE2

Code migration from SSE2 to AVX is non-trivial when the source code is not available (e.g., for proprietary applications), or when the code is written with explicit calls to vector intrinsics, and migration requires a significant development effort. The questions that users of these codes will be asking if they buy a Sandy Bridge CPU should be: is it worth the effort to migrate the code to AVX or buy an AVX-enabled application?

Our results (Figure 1, blue versus red bars) show that the answer really depends on the precision and type of arithmetic operations that the code is performing:

- For linear algebra-like codes, mostly using addition and multiplication, the expected performance gain from migration to AVX is a factor of 2, in both single and double precision.
- However, for codes bottlenecked by floating point division, square root, or derived operations, AVX provides no additional gain.
- Obviously, no gain from migration to AVX is expected for integer-based calculations, as AVX does not support integer arithmetics, and we will have to wait until AVX2 for that support.

[7] Vectorization may be impossible in some arithmetically intensive calculations.

On the other hand, for a code that relies on compiler auto-vectorization, it is a 'no brainer' to recompile it with AVX support in order to squeeze more performance out of the Sandy Bridge CPU. Granted, the code must conform with auto-vectorization guidelines in order to make this transition effective[8]. Regardless of the precision and the type of operations performed by the code, AVX performs at least no worse than SSE2 on Sandy Bridge.

Sandy Bridge vs Westmere

How much faster will my code run on a Sandy Bridge CPU than on a Westmere? Obviously, the answer depends on the type of code. In particular, for arithmetics-bound codes, the difference between Sandy Bridge and Westmere greatly depends on whether the codes are vectorized:

- For scalar (i.e., not vectorized) codes, the E5-2680 is only marginally faster than X5690, which is not surprising, considering that 8×2.70 GHz cores on the E-2680 deliver only 4% as many clock cycles per second as 6×3.46 GHz cores on the X5690. However, a greater difference is expected in the Turbo mode.
- On the other hand, for vector addition and multiplication, the Sandy Bridge beats the Westmere architecture by a large factor, thanks to the wide 256-bit vector registers. Comparing the parallel performance of AVX operations on Sandy Bridge to SSE2 operations on Westmere, we find 2.3x to 2.6x more GFLOP/s for floating point addition and multiplication.
- For division and square root operations, Sandy Bridge performs slightly worse than Westmere, in both single and double precision, with and without vector intrinsics. For some problems, the Sandy Bridge division and square root performance may be improved by using other floating point models used in the compiler (see the -fp-mode compiler argument). However, the nature and amount of these improvements depend on the problem at hand, and therefore are not suitable for a benchmark.
- Surprisingly, Sandy Bridge beats Westmere in the integer arithmetics, even though AVX, which gives the floating point of Sandy Bridge its floating point performance edge, does not support integers. Integer arithmetics on Sandy Bridge falls back to SSE2, but runs 1.5x to 3x times faster than on Westmere.

At the same time, for most codes, the performance of the RAM and the CPU cache are critical. In this study, the Sandy Bridge E5-2680 CPU demonstrated more than double the RAM copy throughput of the Westmere X5690 CPU, and a 15% faster L1 cache.

Conclusion

There is no doubt that Intel's Sandy Bridge 'tock' brought about a lot of improvement over its Westmere predecessor. The arithmetic performance gain is mostly due to wider vector registers supporting the AVX instruction set. The division operation and square root are the only spots where Sandy Bridge has no edge over Westmere. However, additional performance gains are expected from increased cache size and speed, as well as greater RAM bandwidth.

The CIAO benchmark presented here provides a breakdown of a microprocessor's arithmetic performance for different arithmetic operations in available instruction sets. This tool will be expanded to include more functions and architectures. Ultimately, CIAO will be released as an open source project. The results of CIAO allow one to make informed decisions regarding the utility of benchmarked hardware systems for specific arithmetically intensive problems.

Please visit `http://research.colfaxinternational.com/` to learn more about the Colfax Research project, comment on this article, and subscribe for updates.

[8] See http://software.intel.com/file/38565/ and http://research.colfaxinternational.com/post/2012/03/12/AVX.aspx

Appendix: Benchmark Function Code

```
1  double benchmark_flops_add(const int m, const int nswp,
2                             REAL* restrict xx, REAL* restrict yy){
3    const double tstart=omp_get_wtime();
4    for (int s=0; s<nswp; s++) {
5  #ifndef SCA
6  #pragma simd
7  #pragma vector aligned
8  #else
9  #pragma novector
10 #endif
11     for (int i=0; i<m; i++)
12       for (int r=0; r<16; r++)
13         yy[i]+=xx[i];
14   }
15   const double tend=omp_get_wtime();
16   return tend-tstart;
17 }
18
19
20 double benchmark_flops_mul(const int m, const int nswp,
21                            REAL* restrict xx, REAL* restrict yy){
22 ...
23        // xx[i] are initialized to (REAL)1 to avoid overflow
24        yy[i]*=xx[i]; // xx[i] are initialized to (REAL)1 to avoid overflow
25 ...}
26
27
28 double benchmark_flops_div(const int m, const int nswp,
29                            REAL* restrict xx, REAL* restrict yy){
30 ...
31        // xx[i] are initialized to (REAL)1 to avoid overflow
32        yy[i]/=xx[i];
33 ...}
34
35
36 double benchmark_flops_sqrt(const int m, const int nswp,
37                            REAL* restrict xx, REAL* restrict yy){
38 ...
39         // xx[i] are initialized to (REAL)1 to avoid overflow
40 #ifdef SP
41         xx[i]=sqrtf(xx[i]);
42 #elif defined DP
43         xx[i]=sqrt(xx[i]);
44 #endif
45 ...}
46
47
48 double benchmark_flops_exp(const int m, const int nswp,
49                            REAL* restrict xx, REAL* restrict yy){
50 ...
51        // xx[i] are initialized to (REAL)1 to avoid overflow
52 #ifdef SP
53       yy[i]=expf(-xx[i]);
54       for (int r=1; r<NBLK; r++)
55         yy[i]=expf(yy[i]);
56 #elif defined DP
57       yy[i]=exp(-xx[i]);
58       for (int r=1; r<NBLK; r++)
59         yy[i]=exp(yy[i]);
60 #endif
61 ...}
```

```
62
63
64  double benchmark_iops_add(const int m, const int nswp,
65                            INT* restrict xx, INT* restrict yy){
66  ...
67          yy[i]+=xx[i];
68  ...}
69
70
71  double benchmark_iops_mul(const int m, const int nswp,
72                            INT* restrict xx, INT* restrict yy){
73  ...
74          // xx[i] are initialized to 1 to avoid overflow
75          yy[i]*=xx[i];
76  ...}
77
78
79  double benchmark_iops_div(const int m, const int nswp,
80                            INT* restrict xx, INT* restrict yy){
81  ...
82          // xx[i] are initialized to 1 to avoid division by zero
83          yy[i]/=xx[i];
84  ...}
85
86
87  double benchmark_flops_copy(const int m, const int nswp,
88                              REAL* restrict xx, REAL* restrict yy){
89    const double tstart=omp_get_wtime();
90    for (int s=0; s<nswp*16; s++) {
91  #ifndef SCA
92  #pragma vector aligned
93  #else
94  #pragma novector
95  #endif
96      for (int i=0; i<m; i++)
97        xx[i]=yy[i];
98    }
99    const double tend=omp_get_wtime();
100   return tend-tstart;
101 }
```