Squeezing More Instructions per Cycle out of the Intel Sandy Bridge CPU Pipeline

Andrey Vladimirov
Stanford University
for Colfax International

July 31, 2012

Abstract

Parallelism in modern CPU architectures is supported at hardware level by multiple cores, vector registers, and pipelines. While the utilization of the former two is a shared responsibility of the programmer and the compiler, pipelining is handled completely by the processor. It is, however, useful for the developer to know what types of workloads optimize pipeline utilization. This paper shows one example where a specific workload improves the number of instructions executed per clock cycle, boosting arithmetic performance. This workload is comprised of two independent data processing tasks, one performing the AVX addition instruction and the other — the AVX multiplication instruction. Even though these tasks are executed sequentially on one core, alternating additions and multiplications in the code allows the CPU to complete the task 40% faster than when a sequence of additions is followed by a sequence of multiplications. Such workloads are common in linear algebraic applications. Examples in the paper illustrate how improved performance can be achieved in portable C code using the Intel C/C++ compiler. Performance benchmarking with the Intel Vtune Parallel Amplifier is illustrated.

Contents

1 Counting Gigaflops

The Linpack score of a 2-socket Sandy Bridge E5-2690 system published by Intel[1] is 348 GFLOP/s. The system contains two CPUs, each with 8 cores clocked at 2.90 GHz and supporting AVX instructions. In double precision, an AVX instruction performs `256/(8*sizeof(double))=4` floating point operations at once. If one AVX instruction is issued every clock cycle, the theoretical maximum performance of this system is

$$16 \text{ cores} \times 256 \text{ bits per AVX register}/(8 \times \text{sizeof(double)}) \times 2.90 \text{ GHz} = 186 \text{ GFLOP/s}.$$

Clearly, this estimate is too low to explain the observed performance, which is almost twice greater than the 'naive' estimate[2]. The reason for this discrepancy is in the pipelining functionality of the processor.

The ability of the CPU to complete more than one instruction per cycle depends on the workload. Indeed, recent Colfax Research publication[3] reported the measurements of arithmetic performance on the E5-2680 Sandy Bridge CPU. The performance of double precision AVX addition and multiplication of this system reported in that paper is 13-14 GFLOP/s per core, which falls short of the observed performance on 16 cores[4].

The rest of this paper will illustrate that the number of floating point operations per second can be improved if SIMD additions are alternated with SIMD multiplications. In other words, the CPU is able to complete N additions and N multiplications *faster* if addition instructions are alternated with multiplication instructions, as opposed to N consecutive addition instructions followed by N consecutive multiplication instructions.

[1] http://www.intel.com/content/www/us/en/benchmarks/server/xeon-e5-hpc/xeon-e5-hpc-matrix-multiplication.html

[2] Note that the Sandy Bridge CPU does not support the fused multiply-add (FMA) instruction.

[3] http://research.colfaxinternational.com/post/2012/04/30/FLOPS.aspx

[4] The Linpack score of the E5-2680 system reported in that paper has been corrected. See the *addendum* to the article at the same Web address.

2 Baseline: consecutive additions and multiplications

The code introduced in the CIAO benchmark[5] relies on loop tiling in order to increase the arithmetic intensity of the code and reduce the effect of finite memory bandwidth on the measurement. The function benchmarking the multiplication operation can be implemented as shown in Figure 1. In order for this function to be compiled with AVX instructions, the compiler must be run with the argument `-xAVX`. In all examples in this code, choosing `NBLK=8` seems to result in the best performance. Arrays `double xx[m]` and `double yy[m]` are aligned on a 32-byte boundary using the `_mm_malloc()` function.

```
1  // Returns the arithmetic performance, in FLOP/s, of the multiplication operation
2  double benchmark_flops_mul(const int m, const int nswp,
3                             double* restrict xx, double* restrict yy){
4
5    const double tstart=omp_get_wtime();
6    for (int s=0; s<nswp; s++) {
7  #pragma simd
8  #pragma vector aligned
9      for (int i=0; i<m; i++)
10       for (int r=0; r<NBLK; r++) {
11         yy[i] *= xx[i];
12       }
13   }
14   const double tend=omp_get_wtime();
15
16   return (double)nswp*(double)m*(double)NBLK*(double)NSTR/(tend-tstart);
17 }
```

Figure 1: Benchmarking the performance of the vector multiplication operation. This `C` code relies on the automatic vectorization functionality of the Intel compiler to produce SIMD instructions in the executable. `#pragma simd` instructs the compiler to automatically vectorize the `i`-loop. The constant `NBLK` must be known at compile time in order for the compiler to unroll the `r`-loop. The return value of this function is the number of double precision floating point multiplications performed per second.

The assembly code produced by the Intel compiler for the body of the `i`-loop in Figure 1 is shown in Figure 2. It is evident that vectorization has been successful, and the CPU will issue 8 sequential AVX multiplication instructions. But how well will this code perform?

```
1  vmovupd    (%r14,%rax,8), %ymm7
2  vmulpd     (%r12,%rax,8), %ymm7, %ymm0
3  vmulpd     %ymm7, %ymm0, %ymm1
4  vmulpd     %ymm7, %ymm1, %ymm2
5  vmulpd     %ymm7, %ymm2, %ymm3
6  vmulpd     %ymm7, %ymm3, %ymm4
7  vmulpd     %ymm7, %ymm4, %ymm5
8  vmulpd     %ymm7, %ymm5, %ymm6
9  vmulpd     %ymm7, %ymm6, %ymm8
10 vmovupd    %ymm8, (%r12,%rax,8)
```

Figure 2: Assembly code for the body of the `i`-loop in line 9 of Figure 1. This code is generated by the Intel compiler using automatic vectorization. The assembly code was multiversioned, and VTune (see below) was used to identify the assembly lines that were used used when the benchmark was running.

Running the multiplication code (Figure 1) with parameters `m`=2^5 and `nswp`=2^{26} on a single core of the E5-2680 CPU resulted in a performance of 13.6 GFLOP/s for the AVX multiplication operation. The

[5] http://research.colfaxinternational.com/post/2012/04/30/FLOPS.aspx

double precision addition operation was also benchmarked using a similar code, where the operator "$\star=$" is replaced with "$\star=$". The addition operation was clocked at 13.7 GFLOP/s.

The code was also analyzed using the Intel Vtune Parallel Amplifier in the *Lightweight Hotspots* mode. This tool collects profiling information with hardware-based event sampling. In this case, let us consider only two events: *Instructions Retired* and *Cycles Unhalted*. The ratio of the latter to the former is known as *CPI*, the cycles per instruction ratio. This ratio is the inverse of IPC (instructions per cycle), and a CPI of 1.0 means that one instruction is issued in every clock cycle. CPI is often used as a cumulative performance metric: high values of CPI indicate that the code performs sub-optimally, which may be caused by execution stalls due to cache misses, branch mispredictions, and other undesirable effects. While a CPI of 1.0 is generally considered adequate performance, today's Intel CPU are able to achieve even lower values of CPI by utilizing pipelines to issue more than one instruction per cycle.

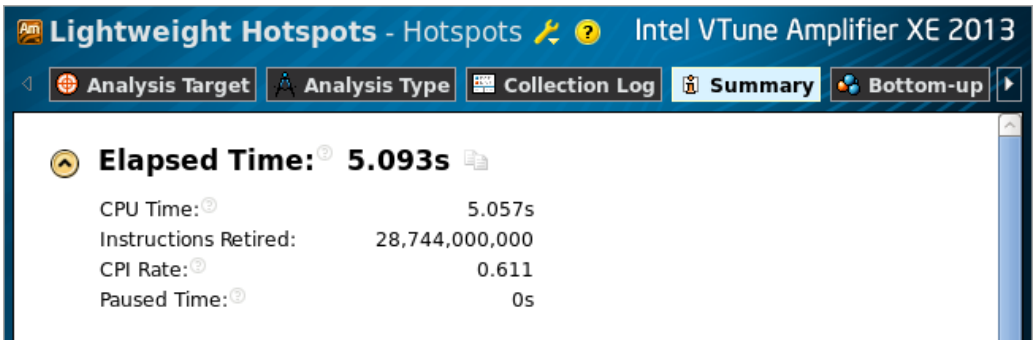

Figure 3: Summary of the Lightweight Hotspots profile for the multiplication benchmark shown in Figure 1 with `m`=2^5 and `nswp`=2^{26}. Profiling was performed in Intel VTune Amplifier XE.

The result of the Lightweight Hotspots analysis of the code in Figure `fig-mul` is shown in Figure 3. Note that the measured value of CPI is 0.611, and the elapsed time of the benchmark with `m`=2^5 and `nswp`=2^{26} is 5.057 s. While this metric can be generally considered as good performance, Section 3 shows that with a different workload, the performance in terms of GFLOP/s and CPI can be improved.

3 Additions alternating with multiplications

In Section 2, the performance of individual addition and multiplication SIMD instructions was benchmarked. Section 3 considers a workload in which independent additions and multiplications are alternated. The code representing this workload is shown in Figure 4.

```
1  double benchmark_flops_addmul(const int m, const int nswp,
2                                double* restrict xx1, double* restrict yy1,
3                                double* restrict xx2, double* restrict yy2){
4
5    const double tstart=omp_get_wtime();
6    for (int s=0; s<nswp; s++) {
7  #pragma simd
8  #pragma vector aligned
9      for (int i=0; i<m; i++)
10       for (int r=0; r<NBLK; r++) {
11         yy1[i]+=xx1[i];
12         yy2[i]*=xx2[i];
13       }
14   }
15   const double tend=omp_get_wtime();
16
17   return (double)nswp*(double)m*(double)NBLK*2.0/(tend-tstart);
18 }
```

Figure 4: Benchmarking the performance of the vector multiplication operations alternated with vector addition operations. Note the factor 2.0 in the return value of the function, which accounts for the fact that every iteration of the `r`-loop performs two independent instructions: one addition and one multiplication. See the caption of Figure 1 for more information.

The assembly code produced by the compiler for the body of the `r`-loop in Figure 4 is shown in Figure 5. Note how the AVX addition instruction `vmulpd` is alternated with the AVX multiplication instruction `vaddpd`.

```
1  vmovupd    (%r14,%rsi,8), %ymm0
2  vmovupd    (%r12,%rsi,8), %ymm14
3  vmulpd     (%r15,%rsi,8), %ymm0, %ymm2
4  vaddpd     (%r13,%rsi,8), %ymm14, %ymm1
5  vmulpd     %ymm0, %ymm2, %ymm4
6  vaddpd     %ymm14, %ymm1, %ymm3
7  vmulpd     %ymm0, %ymm4, %ymm6
8  vaddpd     %ymm14, %ymm3, %ymm5
9  vmulpd     %ymm0, %ymm6, %ymm8
10 vaddpd     %ymm14, %ymm5, %ymm7
11 vmulpd     %ymm0, %ymm8, %ymm10
12 vaddpd     %ymm14, %ymm7, %ymm9
13 vmulpd     %ymm0, %ymm10, %ymm12
14 vaddpd     %ymm14, %ymm9, %ymm11
15 vmulpd     %ymm0, %ymm12, %ymm1
16 vaddpd     %ymm14, %ymm11, %ymm13
17 vmulpd     %ymm0, %ymm1, %ymm0
18 vaddpd     %ymm14, %ymm13, %ymm15
19 vmovupd    %ymm0, (%r15,%rsi,8)
20 vmovupd    %ymm15, (%r13,%rsi,8)
```

Figure 5: Assembly code for the body of the `i`-loop in line 10 of Figure 4.

In order to benchmark the addition/multiplication code, parameter `m` was chosen as 2^4 (i.e., one half of the value for the multiplication benchmark), and `nswp` was chosen as 2^{26} (the same value as for the multi-

plication benchmark). With these parameters, the size of the problem and the total number of floating point operations are the same in this benchmark and the previous one. Running the benchmark yields a performance of 19.1 GFLOP/s, which is 40% greater than 13.6–13.7 GFLOP/s measured for individual additions and multiplications. This means that alternating independent addition and multiplication instructions allows the CPU to utilize the pipeline hardware more efficiently.

This result is further confirmed by the Lightweight Hotspots analysis in VTune. The result of the profiling run is shown in Figure 6.

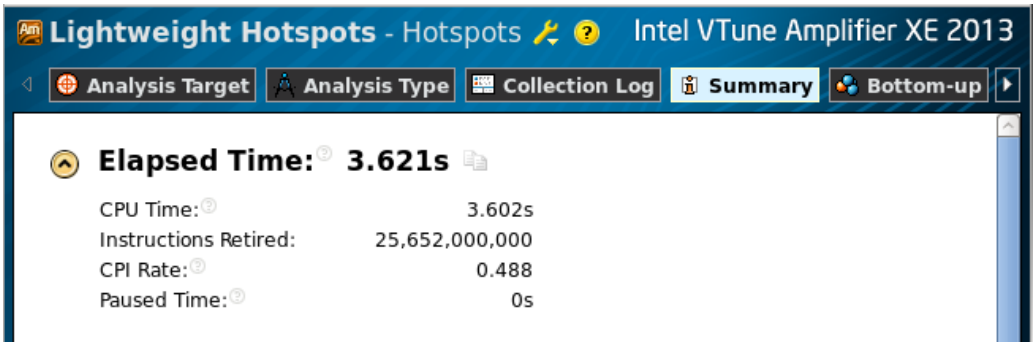

Figure 6: Summary of the Lightweight Hotspots profile for the multiplication benchmark shown in Figure 4 with `m`=2^4 and `nswp`=2^{26}. Profiling was performed in Intel VTune Amplifier XE.

Notably, the number of the Instructions Retired events in the second benchmark is comparable to that counter in the first benchmark; however, the calculation took only 3.621 s instead of 5.093 s, and a lower (i.e., better) CPI of 0.488 was achieved.

4 Conclusions

The case presented in this paper shows here that alternating additions and multiplications are a more favorable workload for the Sandy Bridge E5 series processor than a stream of additions or a stream of multiplications. Naturally, such a workload is highly important for linear algebraic calculations. The Linpack score for this CPU reported in the literature would not be achievable if this pipelining functionality was not exploited in BLAS routines.

In general, arithmetic code optimization on a multi-core system involves a number of methods for which the developer is responsible, including: minimizing synchronization and communication in order to improve scalability across multiple cores; implementing SIMD vectorization with unit-stride data access; maintaining data locality in order to increase memory bandwidth through cache utilization, and other. However, even when the developer produces a code that delivers a near-optimal execution environment for the CPU's arithmetic units, performance still depends on the inner workings of the pipeline. The developer has no direct control over the CPU pipeline. At the same time, the developer can *indirectly* assist the pipelining mechanism by designing the code in a way that delivers a favorable workload to the CPU.

Acknowledgements

I thank Prof. Torben Larsen of Aalborg University, Denmark, for a helpful email discussion that lead to the publication of this paper.

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