Computing and Informatics, Vol. 36, 2017, 1001–1018, doi: 10.4149/cai_2017_5_1001

USING AVX2 INSTRUCTION SET TO INCREASE PERFORMANCE OF HIGH PERFORMANCE COMPUTING CODE

Pawel Gepner

Intel Corporation
Pipers Way, Swindon, Wiltshire
SN3 1RJ, United Kingdom
e-mail: pawel.gepner@intel.com

Abstract. In this paper we discuss new Intel instruction extensions – Intel Advance Vector Extensions 2 (AVX2) and what these bring to high performance computing (HPC). To illustrate this new systems utilizing AVX2 are evaluated to demonstrate how to effectively exploit AVX2 for HPC types of the code and expose the situation when AVX2 might not be the most effective way to increase performance.

Keywords: FMA operations, performance of AVX2 instruction set, benchmarking, Haswell processor, 4th generation Intel Core processor, Intel Xeon E5-2600v3 series processor

Mathematics Subject Classification 2010: 68M07, 68M20

1 INTRODUCTION

In the field of computational science, a synergistic approach based on separated but connected domains, taking into account knowledge about the problem, tools and environments for obtaining a simulation model of the given problem and a properly chosen computer architecture for research is foreseen. Selection of each element of such an approach is equally important, hence the development of simulation systems is usually a complicated task, requiring a collaborative effort from many domain specialists, built with demanding advances in computing/computer sciences.

Each domain, as the element of the approach is to be evolved separately, but with limitations following from the requirements for collaboration.

Development of High Performance Computing (HPC) architectures is performed at different levels, following from their functional construction, with processors as a primary interest. Since the new processor with a novel implementation of Advanced Vector Extensions (AVX) instruction set – AVX2 has brought new level of performance for floating point operations and changed the landscape of HPC, the new era of calculation began. Therefore, the focus of this paper is on efficient handling of floating point operations, as often used in scientific computing and in linear algebra operations in particular. As linear algebra operations, often used in HPC, operate mainly on matrices and scalar functions with focus on basic-type operations like multiplications and additions, a deep study will be done concentrated on operations throughput and latency in different scenario context.

The Haswell processor is the first Intel CPU supporting the AVX2 instruction set, correspondingly the graphics unit has been completely redefined which doubled the graphics performance and made this product even more attractive for the end-users. In addition to AVX2 a new DDR4 memory interface is available within the Intel Xeon E5 family portfolio. In particular AVX2 appears to be very useful for high-performance computing, however, the new graphics unit is dedicated more towards desktop and laptop users. The scope of this paper will not focus on graphics quality as it is concentrated on HPC-type applications.

The rest of the paper is organized as follows. In the next section the overview of existing approaches on FMA implementations are recapitulated and roughly compared. Section 3 outlines the problem under study together with the experimental setup used for experiments, while the following sections report the problem solution, results and conclusions. Outlook for the future research recapitulates the paper.

2 STATE OF THE ART

In March 2008 Intel introduced the first AVX extensions, new products from the Sandy Bridge and Ivy Bridge family established and brought significant advantage for HPC types of code and started a new era of HPC systems. Nevertheless, Intel's architects realised that a lack of a FMA extension and the possibility to expand vector integer SSE and AVX instructions to 256 bits were still required and worked hard to include these new additions. These capabilities were later added to the x86 instruction set architecture and AVX2 with FMA support has been added to Intel CPU family with the launch of the Haswell CPU.

A fused multiply-addition (FMA) is a floating point multiply-addition operation performed in one step, with a single rounding. A FMA computes the entire operation $a+b\times c$ to its full precision before rounding the final result down to N significant bits. The main advantages of using FMA operations are performance and accuracy. Typical implementation of fused multiply-addition operation performs actually faster than a multiply operation followed by an addition and requires only one step versus two steps needed for separated operations. In addition, separated and following multiply and addition operations would compute the product $b \times c$ first, round it to N significant bits and then add the result to a, and round back again to N significant bits, while again, for fused multiply-addition operation computes the entire sum $a + b \times c$ to its full precision before rounding the final result down to N significant bits. Of course, such implementation requires a large amount of logic and extra transistors, but performance benefits compensate the cost of additional resources, as it is going to be demonstrated in this paper.

Intel's AVX2 is not the first implementation of FMA for the x86 family as AMD processors supported it before Intel. Unfortunately, the implementation of AMD is not compatible with Intel's and, more importantly, it is not as powerful. There are two variants of FMA implementation: FMA4 and FMA3. Both implementations have identical functionality but they are not compatible, FMA3 instructions have three operands while FMA4 ones have four. The 3-operand is Intel's version of FMA and makes the code shorter and the hardware implementation slightly simpler and faster. Other CPU vendors implemented FMA in their products as well. The first implementation of FMA has been done by IBM in the IBM POWER1 processor, and then followed by HP PA-8000, Intel Itanium, Fujitsu SPARC64 VI and other vendors. Today even GPUs and ARM processor are equipped with FMA functionality.

Since the official launch of AVX2, FMA has become the standard operation for Intel's ISA but the evolution did not stop here. In 2013 Intel proposed 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions for x86 and the use of FMA has been extended. This extension is not only limited to 512-bit but also adding new Integer Fused Multiply Add instructions (IFMA52) operating on 52-bit integers data. For the first implementation of AVX-512 we need to wait, but definitely this shows the trend, importance and role FMA plays already and it is going to play in the future.

3 PROBLEM DESCRIPTION AND EXPERIMENTAL SETUP

The main problem studied in this paper is how to effectively utilize AVX2 for HPC code and expose the situation when AVX2 might not be the most effective way to increase performance. The systems and CPU carefully selected for this study are described below.

3.1 Architecture and System Configuration

For this study, several Intel Xeon platforms with the Haswell processor have been selected: E3-1270v3 single socket server version, as well as dual socket servers based on Intel Xeon E5-2697v3 and Intel Xeon E5-2680v3.

The Intel Xeon E3-1270v3 processor is a quad core, with 3.5 GHz 80 W CPU in an LGA package. It is dedicated to single socket servers and workstations. The

Intel Rainbow Pass platform has been selected as the validation vehicle with 16 GB of system memory (2 × 8 GB DDR3-1600 MHz) has been installed and with enterprise class Intel SSD X25-E hard disk drives. The system utilizes the Linux kernel installed, known as bullxlinux6.3 (based on Linux 2.6.32 kernel).

The Intel Xeon E5-2697v3 processor has 14 cores, with 2.6 GHz, 145 W CPU in FCLGA2011-3 package. It is dedicated to dual socket servers and workstations. The Intel Xeon E5-2680v3 processor has 12 cores, with 2.5 GHz, 120 W CPU in FCLGA2011-3 package. The Intel Server Wildcat Pass S2600WT platform has been selected as the validation vehicle for all Intel Xeon E5-2600v3 product members. Populated system memory contains 64 GB of (8 × 8 GB DDR4-2133 MHz) for both configurations. The Intel Xeon E5-2600v3 series systems are based on Linux RHEL7 64-bit with a kernel vmlinuz-3.10.0-123.el7.x86_64.

The latest versions of the Intel tools suite are installed on all systems, amongst others, Intel C/C++/Fortran Compiler XE 13.1 Update 1, Intel Math Kernel Library version 1.2 update 2 and Intel VTune Amplifier XE 2013 Update 5. The new compiler and libraries offer advanced vectorization support, including support for AVX-2 and include Intel Parallel Building Blocks (PBB), OpenMP, High-Performance Parallel Optimizer (HPO), Interprocedural Optimization (IPO) and Profile-Guided Optimization (PGO). All performance tools and libraries provide optimized parallel functions and data processing routines for high-performance applications.

3.2 CPU Characteristic

The Haswell processor family is based on the same micro-architecture principles as its predecessor, however, it becomes wider and armed with more resources. As the first members of the Haswell family, the 4th generation Intel Core processors have been designated for the mobile market, the power consumption, performances of the graphics unit as well as performance per Watt characteristic were the primary design goals. Even though those processors are focused on the mobile and desktop markets, some of the technologies, in particular AVX2, are expected to be very applicable for the HPC segment. Of course, the real product for technical computing is coming from the Xeon line as the Intel Xeon E5-2600v3 series. In addition to more cores, bigger cache and more channels of memory, the Intel Xeon E5-2600v3 series supports new DDR4 memory whereas the 4th generation Intel Core processor family still uses DDR3.

It is very important to emphasize that since each Haswell processor is based on the same architecture as other members of the family it does not matter if the product is dedicated for high-end server or desktop or mobile use. The architecture of the single core remains identical for all members, only what is known as "out-core" architecture and configuration details are changed. It is worth understanding that all benefits evaluated on the core level are consistent in the whole family.

The 4th generation Intel Core processor family has been manufactured on 22 nm Hi-k metal Tri-Gate 3-D gate silicon technology and has 1.44 billion transistors for the quad-core version. Almost 30 % of this transistor budget is dedicated to the graphics unit. From a transistor count perspective, this is an increase of 45 % from 995 million transistors implemented on Sandy Bridge. The die size of the new the 4th generation Intel Core processor is 177 mm^2 and it is only 7 mm^2 bigger than quad core version of Ivy Bridge. It operates at the same frequency as Ivy Bridge, however performance will not increase through frequency or core counts growth in this case but via extension to instruction set architecture. The new support for the AVX2 has been added and it is main driver of the floating-point performance increase. To make this improvement powerful two fused multiply-add (FMA) units have been added. Comparing separate dependent multiply and add instructions latency versus FMA difference goes from eight to only five cycles. This is a 60 % instruction latency improvement compared to the previous processors. To support the AVX2 extensions in the Haswell processor the bus width is increased to 256 bits, this effectively doubles the L1 cache bandwidth for AVX type of entries but for integer it remains the same as it was in the Sandy Bridge family.

Utilizing both AVX2 and FMA, the 4th generation Intel Core processor delivers a peak of 16 double-precision flops per cycle – twice as much as the Ivy Bridge and Sandy Bridge processors and four times more then Gulftown and Lynnfield. The AVX2 instruction set also supports integer operations with 256-bit vectors. Since the primary design philosophy of the 4th generation Intel Core processor was to increase performance of a single core – a third ALU, a second store-address generator and a second branch unit have been added. In addition to that two extra ports were added to the unified reservation station, so the 4th generation Intel Core processor can issue and execute eight micro-ops per cycle, which is 33 % more than Ivy Bridge can do, however only four micro-ops per cycle can be decoded and written back [1].

If a code has many branches, the extra branch unit on one of the added ports will definitely help. This new extra branch unit also reduces potential conflicts on one of the existing ports. Similarly, the new store address generation unit on the second added port extends load-address generation operations. Now the 4th generation Intel Core processor can generate two load addresses and one store address in the same cycle [2].

The 4th generation Intel Core processor has implemented an extra multiplier and consequently two ports can do two multiplications simultaneously. In addition, the integer unit reduces the load on two ports, liberating them for supplementary vector operations. The unified L2 TLB was doubled in size to 1024 entries and now can support 4K and 2M pages. The 4th generation Intel Core processor also improves some of the technology that is not suitable for client market but will attract the server segment, e.g., virtualization, crypto algorithms and highly threaded workloads. Fast context switching implementation improves multiple virtualization sessions and makes them more elastic. Crypto algorithms also benefit from new instructions. A good example could be the PCLMULQDQ instructions. The PCLMULQDQ instruction performs carry-less multiplication of two 64-bit operands, which can be used for CRC computation and block cipher encryption and runs over three times faster on Haswell compared with Ivy Bridge.

The 4th generation Intel Core processor has a number of other system enhancements such as support for up to three monitors, dynamic (no reboot required) overclocking control and integrated voltage regulator. For HPC the most important enhancements are FMA and AVX2.

Of course, the Intel Xeon E5-2600v3 series has a lot of more enhancements and new technologies like DDR4 memory support, L3 cache or Intel Turbo Boost Technology 2.0 and many others that provide significant improvement for HPC type of the code being outside the scope of this paper. This study does not focus on them – it just concentrates on single core performance and innovations and look what type of implication they may have for the HPC type of workloads.

4 DESCRIPTION OF A PROBLEM

The main concerns of the paper is how to effectively utilize AVX2 for HPC type of the code and expose the situation when AVX2 might not be the most effective way to increase performance. As mentioned before, the linear algebra operations benefit from AVX2 instructions, deep analysis of throughput and latency of multiplication and addition operations are essentially taken into consideration. As the multiplication and addition operations are critical for scalars and matrices computations it is very interesting to compare and analyse previous generations of Intel Instruction Set Architectures (ISA) for these operations.

The presented performance review allows us to demonstrate the improvement of AVX2. Through the process of an extensive and analytic approach the performance of Intel AVX2 instruction extension in a HPC scenario utilizing Intel software tools including Intel MKL [3, 4, 5] is discussed.

General innovation of Intel AVX2 is based on the new way of promoting the majority of 128-bit integer SIMD instruction sets to operate with 256-bit wide YMM registers. AVX2 instructions are encoded using the VEX prefix and require the same operating system support as AVX. Most of the promoted 256-bit vector integer instructions follow the 128-bit lane operation, similar to the promoted 256-bit floating-point SIMD instructions in AVX. Novel functionalities in AVX2 fall into the following categories:

- integer data types expanded to 256-bit SIMD,
- bit manipulation instructions,
- gather instructions,
- vector to vector shifts,
- Floating Point Multiply Accumulate (FMA).

The AVX2 instruction operates on scalars, 128-bit packed single and double precision data types, and 256-bit packed single and double-precision data types. Double amount of Flops reached compared to previous generations are complemented by doubled cache bandwidth to feed FMA's execution units. Table 1 shows theoretical performance and bandwidth of four generations Intel Microarchitecture based CPUs.

Intel Microarchitectures	Peak Flop/Clock	Peak Cache Bytes/Clock
Banias	4	8
Merom	8	16
Sandy Bridge	16	32
Haswell	32	64

Table 1. Performance of Intel Microarchitectures

5 RESULTS

Having performance in mind it is important to take into account that the transition between AVX/AVX2 and its predecessor, the legacy Intel Streaming SIMD Extensions (SSEx), may result in latency penalties. Because the two instruction sets operate on different levels of YMM registers the hardware during the transition has to first save and then retrieve the contents of the YMM register which penalizes each operation with several tens of additional cycles.

There are ways to avoid penalties related to transitions between AVX/AVX2 and SSEx. One of them is to zero the upper 128 bits of the YMM register with the VZEROUPPER instructions. The complier introduces the VZEROUPPER instruction code at the beginning and the end of the function with AVX/AVX2 instruction code that prevents the transitions from happening when calling the functions in those files from routines containing SSEx instructions.

Another method to avoid penalty during the transition is to have legacy Intel SSEx code converted to AVX/AVX2-128 bit instructions. This can be done by using QxCORE-AVX2 function to convert the auto-generate 128-bit AVX/AVX2 from legacy SSEx code and add VZEROUPPER instruction code.

5.1 Importance of VZEROUPPER

It is important to handle registers and conversions related to transitions between AVX/AVX2 and SSEx, a good example is illustrated by Algorithm 1 which demonstrates performance implications of basic atomic instructions and consequences of appropriate usage of them. This example uses 5 analogous loops (only two of them fully displayed but loop1, loop2, loop3 are constructed based on the same analogy as loop and loop4), where each of them contains 8 interchangeable instructions such as vaddpd, vmulpd or vfmadd132pd.

```
loop:
  vaddpd   ymm0, ymm0, ymm15
  vaddpd   ymm1, ymm1, ymm15
  vaddpd   ymm2, ymm2, ymm15
  vaddpd   ymm3, ymm3, ymm15
  vaddpd   ymm4, ymm4, ymm15
  vaddpd   ymm5, ymm5, ymm15
  vaddpd   ymm6, ymm6, ymm15
  vaddpd   ymm7, ymm7, ymm15
  add rcx, 1
  cmp rcx, repeat
  jb   loop
loop1:
  vmulpd   ymm0, ymm0, ymm15
  ...
  add rcx, 1
  cmp rcx, repeat
  jb   loop1
loop2:
  vmulpd   ymm0, ymm15, ymm15
  ...
  add rcx, 1
  cmp rcx, repeat
  jb   loop2
loop3:
  vfmadd132pd        ymm0, ymm15, ymm15
  ...
  add rcx, 1
  cmp rcx, repeat
  jb   loop3
  ...
loop4:
  vfmadd132pd ymm0, ymm15, ymm15
  vfmadd132pd ymm1, ymm15, ymm15
  vfmadd132pd ymm2, ymm15, ymm15
  vfmadd132pd ymm3, ymm15, ymm15
  vfmadd132pd ymm4, ymm15, ymm15
  vfmadd132pd ymm5, ymm15, ymm15
  vfmadd132pd ymm6, ymm15, ymm15
  vfmadd132pd ymm7, ymm15, ymm15
  vzeroupper
  movaps xmm15, xmm0
  add rcx, 1
  cmp rcx, repeat
  jb loop4
```

Algorithm 1. Independent 5 loops operating on vaddpd, vmulpd (running with no dependency and with dependency) vfmadd132pd (running with VZEROUPPER and without VZEROUPPER)

Instructions	Cycles per Iteration
ADD	8.02
MUL	4.01
MUL (add dependency)	5.01
FMA (VZEROUPPER)	5.04
FMA (SSE/AVX – without proper register handling)	160.78

Table 2. Number of cycles per iteration running Algorithm 1

By running each of the loops in Algorithm 1 independently, it can be seen that ADDs are twice as slow as MULs if there is no register dependency. FMA is a bit slower than MUL but almost twice faster than ADD. When insert dependencies to MUL (i.e. accumulate values) then time goes up to match almost the FMA time. It also demonstrates how important VZEROUPPER is in this code. Using it in an appropriate way makes all the difference. It almost triggers a 30× improvement versus the code without a proper code converting mechanism. Table 2 shows an average execution time of instructions for different atomic types of instruction.

5.2 Linpack

Certainly FMA brings the most impressive performance improvement and accelerates computations significantly. To verify that further the Linpack benchmark has been adopted, with the results summarized in Table 3. The SMP version of Linpack has been executed for the problem size 45 000 compiled against different instruction sets: SSE 4.2, AVX and AVX2 utilizing Intel MKL version 1.2 update 2. The switching between SSE 4.2, AVX and AVX2 has been done using Intel MKL library dispatch (MKL_DEBUG_CPU_TYPE) [6, 7]. For this experiment the Intel Server Wildcat Pass S2600WT platform has been selected as the validation vehicle based on Intel Xeon E5-2697v3 processor.

Instruction Set Architecture	GFlop/s	Improvement AVXx/SSE 4.2
SSE 4.2	297.16	1
AVX	500.56	1.68
AVX2	906.01	3.05

Table 3. SMP Linpack performance

You can clearly see that the AVX2 version utilizing FMA is 81 % faster than version operating on the AVX instructions set and 3 times more powerful than code, which is relying on the SSE 4.2 instruction set architecture only. The improvement between SSE 4.2 versus AVX is smaller than AVX2/AVX because of frequency scaling issue. The problem appears when AVX or AVX2 instructions are executed, processor may run at less than the rated frequency of the CPU, see Section 5.5 [8, 9, 10].

5.3 Inefficiency of FMA

FMA does not guarantee a perfect solution for every multiplication and addition scenario. For instance in the situation of calculating the formula: $x = a \times b + c \times d$ using FMA is presented in Algorithm 2.

```
x = VMULPS(a,b);
x = VFAMDD213PS(c,d,x);
```

Algorithm 2. Calculating $x = a \times b + c \times d$ using FMA

These two operations are performed in 10 cycles while the same calculation could be performed with separate multiply and addition instructions as Algorithm 3 shows.

```
x = VMULPS(a,b);
y = VMULPS(c,d);
x = VADDPS(x,y);
```

Algorithm 3. Calculating $x = a \times b + c \times d$ using multiply and addition instructions

These require only 8 cycles due to parallel execution of two multiplications $x = \mathrm{VMULPS}(a, b)$ and $y = \mathrm{VMULPS}(c, d)$, performed simultaneously on different ports with the latency of 5 cycles plus additional 3 cycles for $x = \mathrm{VADDPS}(x, y)$.

To illustrate this scenario we ran these small kernels 100 000 times each and after all these iterations the non FMA enable kernel was able to provide 10 % less cycles than version utilizing the FMA instructions.

The latency gap will be even larger for a formula like this one $x = a \times b + c \times d + e \times f + g \times h$. FMA implementation with dependency chain through the accumulator will require 20 cycles. This implementation is presented in Algorithm 4

```
x = VMULPS(a,b);
x = VFAMDD213PS(c,d,x);
x = VFAMDD213PS(e,f,x);
x = VFAMDD213PS(g,h,x);
```

Algorithm 4. Calculating $x = a \times b + c \times d + e \times f + g \times h$ using FMA

Alternative parallel accumulator version without usage of FMA instructions is illustrated in Algorithm 5. This implementation costs only 13 cycles as $x = \mathrm{VMULPS}(a, b)$ and $y = \mathrm{VMULPS}(e, f)$ are completed concurrently during the 5 cycles, the same as $x = \mathrm{VFAMDD213PS}(c, d, x)$ and $y = \mathrm{VFAMDD213PS}(g, h, y)$. So with 10 cycles 4 instructions are completed concurrently utilizing two FMA ready ports. The last $x = \mathrm{VADDPS}(x, y)$ operation will be executed in 3 cycles.

```
x = VMULPS(a,b);
y = VMULPS(e,f);
x = VFAMDD213PS(c,d,x);
y = VFAMDD213PS(g,h,y);
x = VADDPS(x,y);
```

Algorithm 5. Calculating $x = a \times b + c \times d + e \times f + g \times h$ without FMA

For this bigger scenario presented in Algorithms 4 and 5 the 100 000 times iterations test verifies that the kernels show 28 % less cycle utilizing non FMA enabled versions and clearly demonstrates that FMA is not always the best solution for every ADD and MUL coding situation.

As it was already discussed, one benefit of the Haswell microarchitecture is the introduction of the FMA instruction in the AVX2 instruction set architecture. Haswell microarchitecture is able to handle two FMA instructions at the same time, (on two dedicated execution ports) with a latency of five cycles and a throughput of one instruction per clock cycle and per execution port (two FMA instructions total per cycle). Intel also designed the Haswell microarchitecture to handle two MUL operations in the same cycle (with latency of 5 cycles on two ports) but with 'only' one ADD operation on port (with latency of 3 cycles). So in theory, a floating point numerical code, which is using more addition than multiply operations cannot take advantage of the Haswell microarchitecture compared to prior generation (Ivy Bridge).

The benefit of having two FMA execution ports for the code, when using a lot of addition instructions is based on the fact that $a + b = a \times 1.0 + b$. Comparing this to the standard ADD instruction, numerically the result is expected to be the same, however the latency of the two instructions will differ, increasing from 3 to 5 cycles but second execution port counts. The latency for FMA instruction is higher but because two FMA instructions can be executed simultaneously two operations $a \times 1.0 + b$ are executed on both ports in 5 cycle when two $a + b$ on single port require 6 cycles.

To demonstrate this, a very simple kernel of sum reductions of an array of simple precision floating-point numbers will be considered. That array will be correctly aligned in memory and the number of elements has been chosen to fit the L1 data cache of a processor core (32 Kbytes).

Pseudo C-code of the kernel is presented in Algorithm 6:

```
T0 = rdtsc();
for(j=0; j<M; j++)
{
  sum = 0.0f;
  for(i=0; i<N; i++) {sum += a[i];}
}
T1 = rdtsc() - T0 - rdtsc_overhead;
```

Algorithm 6. Pseudosum reduction C-code

The main sum reduction is done in the inner loop with the index `i`. Performance is measured in terms of rate of data moved. In order to get accurate performance estimation, time measured is performed using the `rdtsc` (read time-stamp counter) instruction. The processor time stamp records the number of clock cycles since the last reset. Moreover, the test is repeated `M` times (outer loop with index `j`) to increase the statistical significance of the test, lowering the overhead of the `rdtsc` instruction can be done. `rdtsc` overhead is also estimated on runtime by a simple loop training and the instruction latency `rdtsc_overhead` is then deducted and cycle count measurement is accordingly corrected.

Hiding all latencies and considering a throughput of 1 instruction per cycle of the ADD instruction, to move one 256-bits vector per cycle, representing a theoretical peak of 32 bytes/cycle or 41.6 GFlop/s/core can be expected.

The code is compiled using the Intel C compiler (icc) and with the compiler flags -xCORE-AVX2 -O2 -ip. The code is monothreaded and thread is pinned on the core using `sched_affinity` instruction in order to avoid operating system context switching penalties. The performance measured is 31.5 bytes/cycle so close to the expected theoretical peak of 32 bytes/cycle, on both Haswell and Ivy Bridge architecture as this is presented in Table 4.

Assembly code inspection shows that the main loop is unrolled eight times. Assembly code has been generated using the compiler options `-S -masm=intel` to generate Intel syntax assembly output. Loop unrolling is of course necessary to hide the 3-cycle latency of the ADD instruction as expected.

5.4 Influence of the Unrolling Factor

To measure the influence of the unrolling factor the loop has been re-written using C compiler intrinsics and tested on Haswell and Ivy Bridge architecture. Algorithm 7 shows pseudocode for an unrolling factor of eight.

```
T0 = rdtsc();
for(j=0; j<M; j++) {
y7 = y6 = y5 = y4 = y3 = y2 = y1 = y0 = _mm256_setzero_ps();
for(i=0; i<N; i+=64) {  // Unrolled 8 times
  y0 = _mm256_add_ps(y0,_mm256_load_ps(a+i));
  y1 = _mm256_add_ps(y1,_mm256_load_ps(a+i+8));
  y2 = _mm256_add_ps(y2,_mm256_load_ps(a+i+16));
  y3 = _mm256_add_ps(y3,_mm256_load_ps(a+i+24));
  y4 = _mm256_add_ps(y4,_mm256_load_ps(a+i+32));
  y5 = _mm256_add_ps(y5,_mm256_load_ps(a+i+40));
  y6 = _mm256_add_ps(y6,_mm256_load_ps(a+i+48));
  y7 = _mm256_add_ps(y7,_mm256_load_ps(a+i+56));
  k += 1;
}
T1 = rdtsc();
// Final sum reduction
y1  = _mm256_hadd_ps(
_mm256_add_ps(_mm256_add_ps(y0,y1),_mm256_add_ps(y6,y7)),
_mm256_add_ps(_mm256_add_ps(y2,y3),_mm256_add_ps(y4,y5)));

f0  = ((float*)&y1)[0] + ((float*)&y1)[1] +
      ((float*)&y1)[2] + ((float*)&y1)[3] +
      ((float*)&y1)[4] + ((float*)&y1)[5] +
      ((float*)&y1)[6] + ((float*)&y1)[7];
```

Algorithm 7. Pseudocode for an unrolling factor of eight

The code is compiled exactly in the same way as the legacy standard C code. The results presented in Table 4 clearly show that the loop has to be unrolled by at least a factor of 4 to hide the 3-cycle latency of the ADD instruction as expected. All results are in accordance with the theoretical expectation.

Unrolling Factor	Bytes/Cycle	
	Ivy Bridge	Haswell
2	21.2	21.2
4	30.7	31.3
8	30.6	31.5
10	30.0	30.5
16	30.2	30.6

Table 4. Comparing Haswell and Ivy Bridge bytes/cycle

The Algorithm 7 contains the code modified by replacing ADD instruction $y[\cdot] = y[\cdot] + a[\cdot]$ by the FMA instruction sequence $y[\cdot] = 1.0 \times y[\cdot] + a[\cdot]$. Theoretically, optimally unrolled, a bunch of two FMA instructions can be scheduled in the same cycle, theoretically doubling the performance comparing to the usual ADD kernel implementation.

```
_m256 one = _mm256_set_ps(1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f,1.0f);
T0 = rdtsc();
for(j=0; j<M; j++) {
y7 = y6 = y5 = y4 = y3 = y2 = y1 = y0 = _mm256_setzero_ps();
for(i=0;i<N;i+=64) {
  y0 = _mm256_fmadd_ps(one,y0,_mm256_load_ps(a+i));
  y1 = _mm256_fmadd_ps(one,y1,_mm256_load_ps(a+i+8));
  y2 = _mm256_fmadd_ps(one,y2,_mm256_load_ps(a+i+16));
  y3 = _mm256_fmadd_ps(one,y3,_mm256_load_ps(a+i+24));
  y4 = _mm256_fmadd_ps(one,y4,_mm256_load_ps(a+i+32));
  y5 = _mm256_fmadd_ps(one,y5,_mm256_load_ps(a+i+40));
  y6 = _mm256_fmadd_ps(one,y6,_mm256_load_ps(a+i+48));
  y7 = _mm256_fmadd_ps(one,y7,_mm256_load_ps(a+i+56));
  k += 1;
}
T1 = rdtsc();
// Final sum reduction
y1  = _mm256_hadd_ps(
_mm256_add_ps(_mm256_add_ps(y0,y1),_mm256_add_ps(y6,y7)),
_mm256_add_ps(_mm256_add_ps(y2,y3),_mm256_add_ps(y4,y5)));

f0  = ((float*)&y1)[0] + ((float*)&y1)[1] +
      ((float*)&y1)[2] + ((float*)&y1)[3] +
      ((float*)&y1)[4] + ((float*)&y1)[5] +
      ((float*)&y1)[6] + ((float*)&y1)[7];
```

Algorithm 8. Modified version of Algorithm 7 by replacing ADD instruction by the FMA instruction sequence

After modification, the code is compiled and executed exactly in the same way as the previous kernel exposed in Algorithm 7. Results are presented in Table 5.

Unrolling Factor	Bytes/Cycle		
	ADD	FMA	Improvement by Replacing ADD by FMA
1	11.1	6.7	−65 %
2	21.2	13.1	−62 %
4	31.3	25.6	−22 %
8	31.5	45.2	43 %
10	30.5	45.8	50 %
16	30.6	43.9	43 %

Table 5. Comparing Haswell bytes/cycle ratio replacing ADD by FMA

The performance improved by 50 % compared to the previous and usual algorithms. Latency of FMA instruction is 5 cycles (versus 3 cycle latency of the ADD instruction), a higher level of unrolling is necessary. It looks that on average, 2 or 3 extra cycles are observed per iteration explaining the difference between our theoretical expectation and the measurements. These extra cycles can be explained by higher pressure on the cache access in AVX2 (higher computational or numerical intensity). The result with an unrolling factor of 16 is significantly lower than our expectation because of register spilling; 16 YMM registers are not sufficient to permit to hide all instructions or access latencies.

5.5 Benefits of Using AVX2 in Professional Libraries Implementations

The next example of effective use of AVX/AVX2 comes from the DSP field. The company NA Software distributes Digital Signal Processing (DSP) libraries used within the industries requiring very fast processing such as aerospace or defence sectors. As part of a study a very popular AVX Vector Signal Processing Library

(VSIPL) has been tested on Haswell and Ivy Bridge platforms. In order to benchmark DSP processes, various libraries including NAS, IPP (Intel Integrated Performance Primitives) and MKL (Math Kernel Library) DSP have been utilised as part of VSIPL [11].

The majority of the benchmarks, including 2D in-place Complex-to-Complex FTT, Multiple 1D Complex to Complex FFTs, Complex Vector Multiply and Vector Scatter show that the tested libraries execute more successfully on the Haswell platform during testing. In some exceptions the benchmarks including 1D in-place Complex-to-Complex FFT and Complex Transpose proved that NAS is optimised better for complex vector operations. It is also the case when the data is not reduced during the sin/cosine operations (Vector Sine and Vector Cosine benchmarks).

The Complex Vector Multiply performed using VSIPL library depicts very clearly the difference that the Advanced Vector Extensions make to the benchmarking process as Figure 1 illustrates. In the case of both NAS AVX (Ivy Bridge) and AVX2 (Haswell) libraries across all data lengths the results are better than the IPP and MKL implementation without AVX/AVX2 optimisation. With that in mind the peak performance of AVX2 on Haswell was better by over 50 % than AVX on Ivy Bridge.

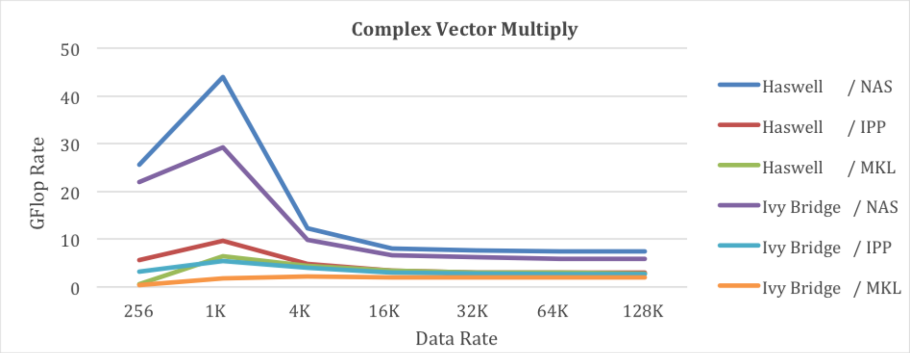

Figure 1. Complex Vector Multiply. NAS vsip_cvmul_f with complex split data

Similarly in both Vector Gather and Vector Scatter the AVX NAS VSIP DSP library performs significantly better on Haswell compared to Ivy Bridge at almost all data lengths. Haswell gains an advantage during Vector Gather after a vector length of 4K on NAS where in Vector scatter it maintains constant and reasonably predictable lead.

Also all three benchmarks of complex-to-complex FFT operations show that the Haswell systems with the advantage of AVX2 are notably better than Ivy Bridge with AVX support only. In the cases where Haswell with AVX2 has not perform as well as Ivy Bridge (AVX) the advantage is usually re-gained with longer data lengths being available [12].

SiSoftware Sandra 2014 has released a study about Cryptohash acceleration through SHA extensions where it shows the impact of the new instruction sets (including AVX and AVX2) available on modern CPUs. In essence, the authors have tested several types of extensions and compared how quickly the data can be encrypted/decrypted per second in MB/s. The Advanced Vector Extensions appear not to be available on the low end/power CPUs [13].

In Figure 2 an almost double performance increase across all three benchmarks is presented. The major improvement on AVX2 is due to double the amount of buffers as it functions on 256-bit integer operations, as opposed to 128-bit integer on AVX [14].

Very similar improvements have been observed during the evaluation of the various small synthetic benchmarks, e.g. MANDELBROT, this small test developed by IT4Innovations National Supercomputing Centre, VSB-Technical University of Ostrava, Czech Republic reported the same level of the benefits delivered by AVX2 versus AVX only version. The second test MULTIDIS that is the chemistry-oriented solver (developed also by IT4Innovations), comparing trajectories of Argon gas, shows 134 % performance improvement when is running on Haswell versus AVX only enabled system [15, 16, 17].

Unfortunately, usage of AVX2 has some negative effect on frequency and Turbo scaling. As outlined, the addition of AVX2 instructions provides drastic performance improvement compared to non-AVX workloads, but when executing AVX2/AVX instructions, the processor may run at less than rated frequency. Of course global performance when using AVX2/AVX instructions is significantly greater than non-AVX instructions even when the processor is operating at a slightly lower frequency. At the same time Intel Turbo Boost Technology continues to provide opportunistic frequency increases based on workload, number of active cores, temperature, power and current. It is important to notice that the amount of turbo frequency achieved depends on all the above elements. Due to workload dependency, separate AVX base and turbo frequencies have been defined for the Haswell product family. There is a new AVX disclaimer explaining that when AVX/AVX2 instructions are executed, it may run at less than the rated frequency.

6 CONCLUSION

The presented study has proven direct evidence of the doubling of the theoretical floating-point capacity with the new AVX2 capability to increase HPC code performance, e.g. the BLAS function of DGEMM or Linpack. For these subroutine and benchmarks, the performance to near double over SSE 4.2 code is observed. Also for the bigger code where a lot of matrix and vector operations are used, new AVX2 extension brings a lot of value, although frequency and Turbo may interfere with ideal scaling. Nevertheless, it is very important to notice that to achieve this level of performance increase, the use of MKL is essential as it simplifies the process of optimization. However, not everything is fully automatic and even a powerful

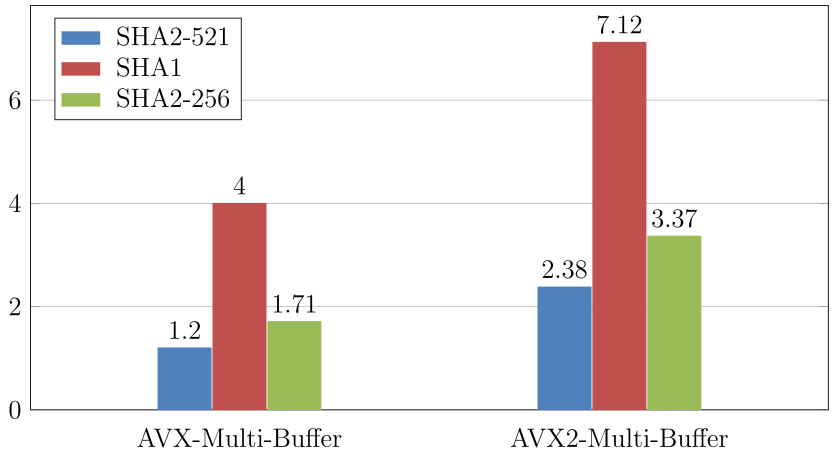

Figure 2. SHA performance utilizing AVX and AVX2

instrument like the FMA instructions are not always the best option to use. The best recommended practice is to always look at the nature of the problem and seek to find the optimal instrumentation.

As the instruction set architecture is evolving and bringing new extensions, e.g. Intel AVX512, the future study is going to be concentrated on the evolution of the new extension and benefits they may bring to the HPC field in particular.

Acknowledgments

The special acknowledgments for contribution and support provided go to Victor Gamayunov, Wieslawa Litke, Hudson Pyke from Intel EMEA Technical Group, Jamie Wilcox from Intel EMEA Technical Marketing HPC Lab and Ludovic Sauge, Cyril Mazauric, Johann Peyrard and Sylvain Cohard from Bull Extreme Computing Applications & Performance Team.

REFERENCES

[1] Optimize for Intel® AVX Using Intel® Math Kernel Library's Basic Linear Algebra Subprograms (BLAS) with DGEMM Routine, The Intel Developer Zone Web Site: `https://software.intel.com/en-us/articles/optimize-for-intel-avx-using-intel-math-kernel-librarys-basic-linear-algebra-subprograms-blas-with-dgemm-routine`.

[2] Intel® Architecture Instruction Set Extensions Programming Reference, Intel Corp., 2015.

[3] Intel® AVX2 optimization in Intel® MKL. Web site: `http://software.intel.com/en-us/articles/intel-mkl-support-for-intel-avx2`.

[4] Demmel, J.—Dongarra, J.—Eijkhout, V.—Fuentes, E.—Petitet, A.—Vuduc, R.—Whaley, R.—Yelick, K.: Self-Adapting Linear Algebra Algorithms and Software. Proceedings of the IEEE, Vol. 93, 2005, pp. 293–312, doi: 10.1109/JPROC.2004.840848.

[5] Dongarra, J.—Du Croz, J.—Hammarling, S.—Duff, I. S.: A Set of Level 3 Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software (TOMS), Vol. 16, 1990, pp. 1–17, doi: 10.1145/77626.79170.

[6] Henry, G.: Optimize for Intel AVX Using Intel Math Kernel Library's Basic Linear Algebra Subprograms (BLAS) with DGEMM Routine. Intel Software Network, July 2009.

[7] Jeffers, J.—Reinders, J.: Intel Xeon Phi Coprocessor High Performance Programming. Morgan Kaufmann, 2013.

[8] Gepner, P.—Gamayunov, V.—Fraser, D. L.: Early Performance Evaluation of AVX for HPC. Procedia Computer Science, Vol. 4, 2011, pp. 452–460, doi: 10.1016/j.procs.2011.04.047.

[9] Gepner, P.—Gamayunov, V.—Fraser, D. L.: Evaluation of Executing DGEMM Algorithms on Modern MultiCore CPU. Proceedings of Parallel and Distributed Computing and Systems 2011 Conference, Dallas, December 2011, doi: 10.2316/P.2011.757-029.

[10] Goto, K.—van de Geijn, R. A.: Anatomy of High-Performance Matrix Multiplication. ACM Transactions on Mathematical Software (TOMS), Vol. 34, 2008, No. 3, Art. No. 12.

[11] Intel AVX VSIPL Benchmarks. `http://www.nasoftware.co.uk/home/attachments/NAS_vsipl_benchmarks.pdf`.

[12] Kopta, P.—Kulczewski, M.—Kurowski, K.—Piontek, T.—Gepner, P.—Puchalski, M.—Komasa, J.: Parallel Application Benchmarks and Performance Evaluation of the Intel Xeon 7500 Family Processors. Procedia Computer Science, Vol. 4, 2011, pp. 372–381.

[13] Crypto Hash Acceleration Through SHA Extensions (HWA). `http://www.sisoftware.co.uk/?d=qa&f=cpu_sha_hw`.

[14] Strassen, V.: Gaussian Elimination Is Not Optimal. Numerische Mathematik, Vol. 13, 1969, No. 4, pp. 354–356, doi: 10.1007/BF02165411.

[15] Krotkiewski, M.—Dabrowski, M.: Parallel Symmetric Sparse Matrix-Vector Product on Scalar Multi-Core CPUs. Parallel Computing, Vol. 36, 2010, No. 4, pp. 181–198, doi: 10.1016/j.parco.2010.02.003.

[16] Bell, N.—Garland, M.: Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. Proceedings of High Performance Computing Networking, Storage and Analysis (SC '09), ACM, 2009, Art. No. 18.

[17] Gepner, P.—Gamayunov, V.—Fraser, D. L.—Houdard, E.—Sauge, L.—Declat, D.—Dubois, M.: Evaluation of DGEMM Implementation on Intel Xeon Phi Coprocessor. ICCSIT 2013, Journal of Computers, Vol. 9, 2014, No. 7, pp. 1566–1571, doi: 10.4304/jcp.9.7.1566-1571.

Pawel Gepner is Intel Corporation Platform Architect focused on high performance computing at DCG FAE Team EMEA. He has joined Intel in 1996 as Field Application Engineer for Central and Eastern Europe. In 2001, he became EMEA Architect focused on HPC area. Currently he is responsible for the development and design of future server platforms and HPC systems at one of Intel's leading EMEA partners. He is Intel Corporation spoke person responsible for communication with the press regarding technical and technology aspect of Intel's products and technology. He led couple of server development projects including first Fault Tolerance Systems based on IA-32 from Stratus Technology. He was responsible for driving Pentium III server project at IBM Development Center in Greenock. He also led the team of Intel architects that developed Bull's Itanium 2 system. He was also involved in Itanium 2 projects at Siemens AG and Eriksson. He led the development team for the first teraflop computing projects in EMEA and first Itanium 2 teraflop installations. He was driving many of the HPC projects in including TASK, SKODA, VW, CERN, and many others. He is graduated in computer science and he holds Master's and Ph.D. degrees from Warsaw University of Technology, Poland and habilitation from Czestochowa University of Technology. He has written 50 technical papers on computer science and technology. He is also a board member and technology advisor for many international scientific and commercial HPC projects.