# Theoretical Peak FLOPS per instruction set on modern Intel CPUs

Romain Dolbeau
Bull – Center for Excellence in Parallel Programming
Email: romain.dolbeau@atos.net

***Abstract*—It used to be that evaluating the theoretical peak performance of a CPU in FLOPS (floating point operations per seconds) was merely a matter of multiplying the frequency by the number of floating-point instructions per cycles. Today however, CPUs have features such as vectorization, fused multiply-add, hyper-threading or "turbo" mode. In this paper, we look into this theoretical peak for recent full-featured Intel CPUs., taking into account not only the simple absolute peak, but also the relevant instruction sets and encoding and the frequency scaling behavior of current Intel CPUs.**

**Revision 1.41, 2016/10/04 08:49:16**
***Index Terms*—FLOPS**

## I. Introduction

High performance computing thrives on fast computations and high memory bandwidth. But before any code or even benchmark is run, the very first number to evaluate a system is the theoretical peak - how many floating-point operations the system can theoretically execute in a given time. The most common measurement is the FLOPS, floating-point operations per second. The simple view is: the more FLOPS, the better.

However, evaluating the peak FLOPS is not as easy as it looks. It used to be that multiplying the number of floating-point operations per cycle by the number of cycles per second was enough. It was the simple equation in equation 1. A node is a single computer (perhaps part of a cluster), and it's easy to define it's speed. It has only one super-scalar core, no fancy vector, and this is the early 1990s.

$$\frac{flops}{node} = \frac{flops}{second} \times \frac{cores}{node} \tag{1}$$

But since then, many level of parallelism have been introduced. Multiple physical silicon devices per nodes ("socket"), each with multiple cores, and potentially multiple threads per core. Vector of varying sizes. And more sophisticated instructions. Equation 2 describes a more realistic view, that we will explain in details in the rest of the paper, first in general in section II and then for the specific cases of Intel CPUs: first a simple one from the Nehalem/Westmere era in section III and then the full complexity of the Haswell family in section IV. A complement to this paper titled "Theoretical Peak FLOPS per instruction set on less conventional hardware" [1] covers other computing devices.

$$\frac{flops}{node} = \left.\begin{array}{r} \frac{flop}{operation} \\ \times \frac{operations}{instruction} \\ \times \frac{instructions}{cycle} \end{array}\right\} micro-architecture \left.\begin{array}{r} \times \frac{cycles}{second} \\ \times \frac{cores}{socket} \\ \times \frac{sockets}{node} \end{array}\right\} machine \quad architecture \tag{2}$$

## II. Contribution to the peak

In this section, we will detail the meaning of each part of equation 2. Of course, they are not fully independent, as explained in each relevant section. As indicated in equation 2, the first three are part of the micro-architecture (how the CPU

is designed), whereas the last three are part of the machine architecture (how many devices, which devices, what speed).

### A. $^{flop}/_{operation}$: *instructions*

How many floating point "operation" (in the mathematical sense of the word) are embedded in the semantic of the instruction. Usually, most time-consuming operations (such as division) are simply not taken into account, and only the simplest and most relevant are counted: addition, subtraction, multiplication. They all count as one. So any instruction encoding one of those operations is represented by the ratio $^1/_1$, whereas the quite common fused multiply-add (or any variant) $d = a \times b + c$ is represented by the ratio $^2/_1$.

### B. $^{operations}/_{instruction}$: *vector*

How many (group of) operations are executed simultaneously by an instruction. This is the so-called "vector" or "SIMD" mode. If an instruction uses vector of 4 elements as operands, then each of the mathematical operation(s) are executed four times, for a ratio of $^4/_1$. This can vary greatly, from $^1/_1$ for "scalar" instructions, to $^{16}/_1$ for single-precision floating-point in an AVX-512 vector of 64 bytes.

### C. $^{instructions}/_{cycle}$: *super-scalar*

How many instructions can be executed simultaneously in a cycle. This is trickier than it sounds. For instance, do Intel processors take into account the legacy x87 Floating-Point Unit (FPU)? Or only the SIMD units? The general question is, are those instructions homogeneous enough - in terms of the previous two factors - to be represented by a simple fraction ? Usually yes, as we can for instance simply ignore the legacy FPU and the complex instructions such as division. So if a CPU can do two fused multiply-add at once, the ratio will be $^2/_1$; for two arbitrary single-operation instruction (add, sub, mul) again a ratio of $^2/_1$; and a fused multiply-add in parallel with a single-operation instruction can be encompassed by a ratio of $^3/_2$. The real complexity comes from CPU with non-scalar floating-point units where less than one instruction per cycle is possible - then more complex calculations have to be made. Fortunately, few modern CPUs are in that category.

### D. $^{cycles}/_{second}$: *frequency*

How many cycles in a second. While it sounds as the easiest factor to compute, it can be quite tricky as well. Modern CPUs feature "turbo" mode whereas some of the cores can have a higher speed than nominal. Energy-saving mode lower the frequency. Recent Intel CPUs have a different base frequency when exploiting certain instructions, in particular AVX instructions - the very instructions we are trying to count.

### E. $^{cores}/_{socket}$ & $^{sockets}/_{node}$: *multi-core*

How many cores in a node, computed with the two factors. However, care should be taken: while hyper-threading (or simultaneous) adds additional resources to the processors seen as "cores" by the operating system, it does not add execution units. So the number of cores does not take hyper-threading into account.

## III. An easy Intel CPU: Nehalem/Westmere

The first case study are the older Intel CPUs from the Nehalem or Westmere generation[1]. Those CPUs use the SSE2 SIMD instruction set for floating-point operations. This instruction set features 128 bits (16 bytes) registers, capable of holding either 2 double-precision or 4 single-precision elements. The instruction set does not support fused multiply-add. But it does support a scalar mode, with only one element used per vector register. Table I summarizes the number of FLOPS per cycle one can expect from the Nehalem family in the three possible modes - scalar, vector single-precision and vector double-precision. The value for scalar applies to both single and double precision.

Then for a given CPU model we can add the specific data (frequency, core per socket, socket per node) and obtain the raw per-node theoretical peak, as is done in figure II for the Xeon X5650[2]. This however is an approximation - as the description of the CPU signals, "turbo" mode allows some core to

[1]Most processors even older than Nehalem but supporting SSE2 would fall into the same category. Strictly speaking, SSE only supports single-precision floating-point operations, and SSE2 supports double-precision. Processor without SSE2 have to rely on x87 for double-precision arithmetic, and are not considered. In the remainder of this paper, the term SSE will be used to describe the SSE & SSE2 combination, since both are mandatory on all x86-64 processors.

TABLE I
THEORETICAL PER-CYCLE PEAK FOR NEHALEM/WESTMERE

| | SSE (Scalar) | SSE (DP) | SSE (SP) |
|---|---|---|---|
| $^{flop}/_{operation}$ | 1 | 1 | 1 |
| $\times{}^{operations}/_{instruction}$ | 1 | 2 | 4 |
| $\times{}^{instructions}/_{cycle}$ | 2 | 2 | 2 |
| $=^{flop}/_{cycle}$ | 2 | 4 | 8 |

go up to 3.06 GHz, something we have not taken into account.

TABLE II
THEORETICAL PER-NODE PEAK FOR X5650

| | SSE (Scalar) | SSE (DP) | SSE (SP) |
|---|---|---|---|
| $^{flop}/_{cycle}$ | 2 | 4 | 8 |
| $\times{}^{cycles}/_{second}$ | $2.67G$ | $2.67G$ | $2.67G$ |
| $\times{}^{cores}/_{socket}$ | 6 | 6 | 6 |
| $\times{}^{sockets}/_{node}$ | 2 | 2 | 2 |
| $=^{flops}/_{node}$ | $64G$ | $128G$ | $256G$ |

## IV. THE COMPLEX CASE: THE HASWELL FAMILY

The Intel Haswell (a.k.a. Xeon E5 v3) family is much more complex to evaluate. First we will talk about the various SIMD modes, and then about the frequency of the cores.

### A. Instruction sets

*1) Description:* Up to and including the Westmere family described above, the only vector mode was SSE. However, Intel has since introduced a new encoding scheme and instruction set in the Sandy Bridge micro-architecture, known as AVX, which pushed vector register sizes to 256 bits (32 bytes)[2]. It then extended AVX with new instructions in the Haswell micro-architecture, such as integer operations and a "gather" operation, in an instruction set known as AVX2. The encoding scheme and instructions are described in [5]. Intel also introduced in Haswell the FMA instruction set, which is commonly described as AVX2, even though it is technically distinct with its own flag in the CPUID instruction. Suffice to say that the Haswell family supports all of those - SSE, AVX, AVX2 and FMA. It is therefore possible to access floating point operations in the following ways (always ignoring the legacy x87 FPU):

1) SSE-encoded scalar mode;
2) SSE-encoded double-precision vector mode;
3) SSE-encoded single-precision vector mode;
4) AVX-encoded scalar mode;
5) AVX-encoded 128 bits double-precision vector mode;
6) AVX-encoded 128 bits single-precision vector mode;
7) AVX-encoded 256 bits double-precision vector mode;
8) AVX-encoded 256 bits single-precision vector mode;
9) AVX-encoded scalar mode (with FMA)
10) AVX-encoded 128 bits double-precision vector mode (with FMA);
11) AVX-encoded 128 bits single-precision vector mode (with FMA);
12) AVX-encoded 256 bits double-precision vector mode (with FMA);
13) AVX-encoded 256 bits single-precision vector mode (with FMA);

That is no less than thirteen different ways of computing on floating-point data. The first three (1–3) and last two (12, 13) seems pretty obvious. The first three are mentioned in the previous section. The last two are the new 256 bits-wide instructions with FMA support from Haswell, which are well known. However, AVX-encoded instructions are not behaving well with the older, SSE-encoded 128 bits instructions and should not be mixed. Also, SSE is a two-operand instruction set (the output share a register with an input) whereas AVX is a three-operand instruction set (neither input are overwritten by the output). For those reasons, the AVX encoding scheme includes the re-encoding with three operands of all SSE instructions (later shortened to AVX-128), both scalar and 128 bits-wide vector instructions, creating another three ways (9–11). Finally, in the middle (4–8) there is the non-FMA version of AVX (from the "Sandy Bridge" and "Ivy Bridge" families), which can be compiled and run on Haswell without the benefit of FMAs.

It may seem pedantic to mention all those, but they are quite important. First, as mentioned, the new encoding scheme allows three operands, thus

[2]Another encoding scheme will follow with the processor codenames "Knights Landing" and "Skylake" [3][4])

potentially removing spurious "move" instructions, and generally making life easier for the compiler or the assembly programmer. Second, for code written using Intel intrinsics functions for SSE, the compiler can generate them using AVX-128 instructions, benefiting both from the new three operands scheme and compatibility with AVX-256. Third, codes that fail to vectorize properly and are therefore "scalar" still can exploit the new fused multiply-add operations, available in AVX-encoded scalar but not SSE-encoded scalar. Finally, all those variant can be generated by the compiler - knowing how the code was compiled will constrain which variant was used, and therefore the attainable peak. Table III summarizes the $^{flop}/_{cycle}$ for the Haswell family. The non-FMA variants of AVX are not in the tables: they have exactly half the $^{flop}/_{operation}$ of the corresponding FMA variants, and for AVX-128 without FMA they have the same peak as the corresponding SSE variant.

We can now add the specific data (frequency, core per socket, socket per node) for a model and obtain the raw per-node theoretical peak, as is done in figure IV for the Xeon E5-2695 v3[6]. Again we only consider the nominal frequency, something we will revisit in the next section. As can be seen in the last line of the table, theoretical peak is a relative terms, since there is a factor of 16 between a non-vectorized code encoded in SSE (which doesn't have access to the fused multiply-add) and a full-vector single-precision code encoded in AVX (which has a vector width of 8, and can do two FMAs per cycle).

*2) Note on instruction mix:* As mentioned in section II-C, the number of instructions per cycle is an approximation in some cases, since some CPU will support only some mix of instructions. This is the case for Intel processor, although it does not affect the ratio itself. It is nonetheless interesting to specify the capabilities of the instructions pipeline.

**SSE, AVX** SSE and AVX (without FMA) are implemented by Intel with the same capabilities. They have the ability to issue one multiplication and one addition (or subtraction) simultaneously, hence the two instructions per cycle. Independent multiplications cannot run together, and neither can independent additions.

**FMA aka AVX2** The Haswell core has the capability to execute two FMA per cycle, again hence the two instructions per cycle. However the two pipelines are not identical. Pipeline 0 has the ability to execute multiplications and long-running (and mostly ignored here) instructions such as division and reciprocal, but it does not support additions. Pipeline 1 has the ability to execute multiplications and additions. So a pair of independent additions will not be able to execute simultaneously. It is possible to use a FMA as a substitute for an addition, but care has to be taken since the latency of the FMA is higher than that of an addition (5 cycles vs. 3 cycles).

### B. Frequency

Unfortunately, the base frequency is no longer a reliable measure to estimate peak performance. As detailed by Intel in [7], Intel now specifies no less than four different frequencies for its CPUs, with the last two being new in server-class Haswell CPUs:

- The "marked TDP frequency", the one used up to that point in this paper and usually referred to in marketing materials, the minimum frequency for all (now only non-AVX) workloads;
- The "turbo frequency", the maximum frequency any one core can attain;
- The "AVX base frequency", the minimum frequency for AVX workloads;
- The "AVX max all core turbo", the maximum frequency reachable on any one core for AVX workloads.

Another Intel document [8] is full of tables detailing the maximum turbo frequencies for the normal and AVX case depending on how many cores are in use, including the above values. We learn about our chosen CPU that our four frequencies are:

- Base: 2.3 GHz (table 2 page 8 of [8])
- Turbo: 3.3 GHz
- AVX Base: 1.9 GHz (table 3 page 10 of [8])
- AVX all Max: 3 GHz

Apparently, the computation in table IV are significantly overestimating the peak - we lose over 17% due to the 1.9 GHz instead of 2.3 GHz frequency. But that's not the whole story - those frequencies are in fact not enough to specify a

TABLE III
THEORETICAL PER-CYCLE PEAK FOR HASWELL

| | SSE (Scalar) | SSE (DP) | SSE (SP) | AVX+FMA (scalar) | AVX-128 +FMA (DP) | AVX-128 +FMA (SP) | AVX+FMA (DP) | AVX+FMA (SP) |
|---|---|---|---|---|---|---|---|---|
| $^{flop}/_{operation}$ | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| $\times^{operations}/_{instruction}$ | 1 | 2 | 4 | 1 | 2 | 4 | 4 | 8 |
| $\times^{instructions}/_{cycle}$ | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| $=^{flop}/_{cycle}$ | 2 | 4 | 8 | 4 | 8 | 16 | 16 | 32 |

TABLE IV
THEORETICAL PER-NODE PEAK FOR E5-2695 V3

| | SSE (Scalar) | SSE (DP) | SSE (SP) | AVX+FMA (scalar) | AVX-128 +FMA (DP | AVX-128 +FMA (SP) | AVX+FMA (DP) | AVX+FMA (SP) |
|---|---|---|---|---|---|---|---|---|
| $^{flop}/_{cycle}$ | 2 | 4 | 8 | 4 | 8 | 16 | 16 | 32 |
| $\times^{cycles}/_{second}$ | $2.3G$ | $2.3G$ | $2.3G$ | $2.3G$ | $2.3G$ | $2.3G$ | $2.3G$ | $2.3G$ |
| $\times^{cores}/_{socket}$ | 14 | 14 | 14 | 14 | 14 | 14 | 14 | 14 |
| $\times^{sockets}/_{node}$ | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| $=^{flops}/_{node}$ | $128.8G$ | $257.6G$ | $515.2G$ | $257.6G$ | $515.2G$ | $1030.4G$ | $1030.4G$ | $2060.8G$ |

part. The maximum "turbo" frequency are fully defined depending on how many cores are in use, and even with all cores running the maximum turbo frequency (with or without AVX) is much higher than its respective base frequency, although lower than the one core "Turbo" frequency. If we take those minimum and maximum "turbo" frequency into account, we obtain table V for a node of two sockets. Note that even when using all 14 cores in each socket, we can still reach a frequency of 2.6 GHz - not only 0.7 GHz more than the AVX Base Frequency, but still 0.3 GHz more than the Base frequency.

### C. Practical measurements

The best way to confirm a theory is to try it in practice. In this section, we are going to measure observed throughput of instructions, match it against the known characteristics of the Haswell architecture, and deduce the operating frequency in various cases.

*1) Methodology:* We start with a very simple code that simply feed sequence of identical instructions, and measure the time taken to execute the sequence. Knowing the sequence and the pipeline, we can define the theoretical throughput - the number of cycle per instructions the core will take to execute the sequence. From this, we can deduce the total number of cycles. Combined with the wall-clock measurement, we can deduce the operating frequency. There is no point in directly using the time-stamp counter (TSC) in recent Intel CPUs for reason other than validation, as in Haswell the TSC always run at the nominal (base) frequency of the CPU, a feature known as an "invariant TSC".

So for instance, if we only compute AVX vector shift, then we know from [9] page 193 that we can execute one per cycle and the producer-consumer latency is one cycle. Therefore any sequence of such vector shifts will take as many cycles as there is instructions. So if we build a sequence of $8192$ and loop over it $2048$ times, we expect an execution time of $8192 \times 2048 = 16777216$ cycles. If we measure a time of approximately $5.6$ milliseconds, then we are running at $16777216/(5.6 \times 10^{-3}) \approx 3 \times 10^9$, or approximately 3 GHz.

This process can be done for many instruction types, and with two different pipeline filling: either running many independent instructions to fill the pipeline as much as possible, or with sequentially dependent instructions. We can then run from one to as many threads as there is core in the socket.

For this paper, we tried the following cases:

**vfma** AVX2-256 FMA, filling the pipeline to $^4/_5$ (8 FMAs in 5 cycles);

**vfmadep** AVX2-256 FMA, sequentially dependent

TABLE V
TURBO-DEPENDANT PEAK DUAL E5-2695 V3 NODE USING AVX+FMA DP

| $^{activecores}/_{socket}$ | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AVX (min) | freq | 1.9 | 1.9 | 1.9 | 1.9 | 1.9 | 1.9 | 1.9 | 1.9 | 1.9 | 1.9 | 1.9 | 1.9 | 1.9 | 1.9 |
| AVX (max) | freq | 3 | 3 | 2.8 | 2.7 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 | 2.6 |
| AVX Flops (min) | DP | 60.8 | 121.6 | 182.4 | 243.2 | 304.0 | 364.8 | 425.6 | 486.4 | 547.2 | 608.0 | 668.8 | 729.6 | 790.4 | 851.2 |
| AVX Flops (max) | DP | 96 | 192 | 268.8 | 345.6 | 416.0 | 499.2 | 582.4 | 665.6 | 748.8 | 832.0 | 915.2 | 998.4 | 1081.6 | 1164.8 |

(1 FMA every 5 cycle);

**vshift** AVX2-256 vector shift (an integer instruction, one per cycle);

**vpaddd** AVX2-256 vector add (an integer instruction, two per cycle);

**vmulpd** AVX-256 MUL, filling the pipeline to $^4/_5$ (8 MULs in 5 cycles, on Haswell);

**vmulpd128** AVX-128 MUL, filling the pipeline to $^4/_5$ (8 MULs in 5 cycles, on Haswell);

**mulpd** SSE MUL, filling the pipeline to $^4/_5$ (8 MULs in 5 cycles, on Haswell).

This way we can verify the throughput and latency of the main FPU instruction (FMA), integer workload filling one or two vector pipeline, and compare AVX and SSE encoding.

The $^4/_5$ figure (8 operations in 5 cycles) for FMAs and MULs are a side-effect of the way the sequences are built. A block of 8 operations use 8 different AVX (or SSE) registers as both input and output (this is mandatory for SSE where the output register is always one of the input, and we re-use the same pattern for three-operands AVX instructions). The block is then repeated to create the long sequence, thus creating a dependency chain between each consecutive block of 8 operations: no instruction in a block can be scheduled until the same instruction of the previous block is complete. Since the producer-consumer latency for FMAs and MULs is 5 cycles, it takes at least 5 cycles between each block, thus 8 operations in 5 cycles. As on the Haswell architecture two such instructions can be executed in one cycle, or 10 instructions in 10 cycles, the pipeline is only filled to $^8/_{10}$ or $^4/_5$.

*2) Effect of threads:* When running single thread, we expect AVX instructions to run (per table V) at 3 GHz. The actual output for the peak frequency observed is (output is rounded to 33 MHz for clarity):

```
----------------
0
----------------
     vfma: 3000 MHz
  vfmadep: 3000 MHz
   vshift: 3300 MHz
   vpaddd: 3300 MHz
   vmulpd: 3000 MHz
vmulpd128: 3300 MHz
    mulpd: 3300 MHz
```

Apparently, the AVX frequency is limited to 256 bits floating-point AVX-encoded instructions. All of the vector integer operations, the 128 bits floating-point AVX-encoded operations and the SSE-encoded floating point operations are running at the maximum frequency of 3.3 GHz on this particular E5-2695 v3. Whether the pipelines are saturated by the FMAs or not does not change the running frequency.

We get the same results for 2 threads, as documented, and the following for three threads:

```
----------------
[0-2]
----------------
     vfma: 2900 MHz
  vfmadep: 2900 MHz
   vshift: 3100 MHz
   vpaddd: 3100 MHz
   vmulpd: 2900 MHz
vmulpd128: 3100 MHz
    mulpd: 3100 MHz
```

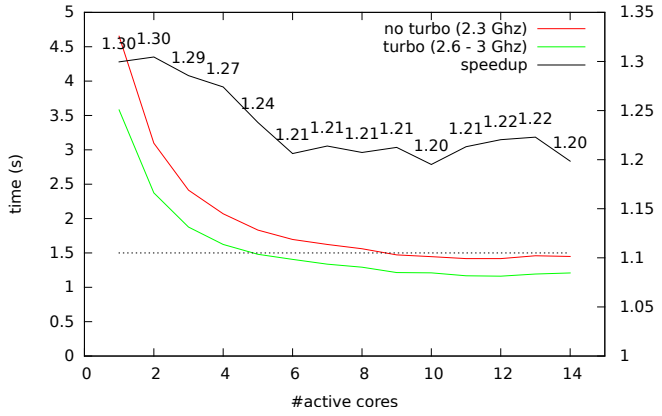
The 3.1 GHz is a match for table 2 in [8], but the 2.9 GHz for AVX instructions is 100 MHz

higher than documented. This intriguing behavior is also visible at four and five cores. From six cores onward, the frequency are stuck at 2.6 and 2.8 GHz, exactly as documented:

```
----------------
[0-13]
----------------
      vfma: 2600 MHz
   vfmadep: 2600 MHz
    vshift: 2800 MHz
    vpaddd: 2800 MHz
    vmulpd: 2600 MHz
 vmulpd128: 2800 MHz
     mulpd: 2800 MHz
```

*3) Example:* The practical results are illustrated in figure 1, representing the speed of execution of a basic finite difference kernel inside a socket. The kernel is full vectorized in AVX2 (with FMA). As is immediately noticeable, the kernel does not scale very well with the number of threads for that particular test case (this is a stand-alone kernel extracted from a real-life application). Activating the turbo gives an instant speed-up for every possible number of active threads, but the benefits are clearly higher for a small number of threads, where the turbo is the most effective. The light grey line represents a time of 1.5 seconds, close to the asymptotic speed without turbo, and shows that with the turbo that level of performance can be achieved with only five active cores.

Fig. 1. Turbo effect on a finite difference kernel




*4) Hyperthreading:* Hyperthreading allows to run more than one flow of instructions in a core, thus allowing a higher throughput, possibly at the expense of single-thread performance. We can use our test code to observe the phenomenon. The reported frequency should no longer be valid, since the number of cycles for a given sequence of instructions will be affected by the other thread. The results for two threads sharing a core are as follow.

```
----------------      ----------------
0                     1
----------------      ----------------
      vfma: 1867 MHz        vfma: 1900 MHz
   vfmadep: 3000 MHz     vfmadep: 3000 MHz
    vshift: 1667 MHz      vshift: 1667 MHz
    vpaddd: 1633 MHz      vpaddd: 1667 MHz
    vmulpd: 1867 MHz      vmulpd: 1900 MHz
 vmulpd128: 2067 MHz   vmulpd128: 2067 MHz
     mulpd: 2067 MHz       mulpd: 2067 MHz
```

The `vshift` case is quite obvious: since the test fill up the only available pipeline, each case only observe about half the "available" frequency - that is, each copy takes twice as long. Or, we do twice as much work in the measured amount of time. We need to sum the computed frequency to obtain the effective frequency. Recall now that the `vfma` test only fill the pipelines to $^4/_5$, that is $20\%$ of the instruction slots are unused by the instruction flow. If the `vfma` was behaving like the `vshift` test, we would measure an apparent frequency of only 1.5 GHz, summing to 3.0 GHz. But the sum (each half rounded to 33 MHz) is 3767 MHz - or about $^5/_4$ of 3000 MHz (with rounding approximation). The empty available slots of one thread are now fully filled by the other thread, so no slots are unused. By utilizing the hyperthreading, we have made the processor slightly more efficient. It is much more obvious for the `vfmadep` test, where the observed frequency is 3 GHz on both thread, for no loss compared to the single-thread case. In that instance, as each instruction flow only fills $20\%$ of the instruction slots, the second one execute for free in the remaining slots. The efficiency is effectively doubled.

## V. Compiler behavior and assembly output

In this section, we will study how to ensure the compiler is targeting the preferred micro-architecture. We will then study how to check that

an assembly code (output from the compiler or disassembled by a specific tool or a debugger) is indeed matched to the micro-architecture. Please note that only a superficial view of each compiler is given, and the compiler documentation should be used as reference for the exact behavior.

### A. Compiler behavior

High-performance computing requires performance and therefore tuning the code to the architecture. However, most code is destined to run on a large number of systems of varying micro-architectures: system code, common libraries from the operating system, many ISV codes. As a result, most compilers target a subset of common capability among several micro-architectures, so as to produce a code that will run anywhere. On the x86-64 architecture, this means only SSE and SSE2 in addition to the basic micro-architecture. Most other instructions, including all AVX flavors, are out-of-bounds as they are not universally available. Even if the underlying micro-architecture supports some instructions, some specific model may not. For instance, a low-end low-power CPU such as the Intel® Celeron® Processor 1007U is based on the Ivy Bridge micro-architecture, but it doesn't support any AVX instructions or the encryption-oriented AES instructions. As a result, the compiler must be explicitly told to use the newer instructions.

*1) Intel Compiler:* The Intel compiler (in version 15.0 at the time this is written) default to limiting the capability to SSE2. There is several way to specify how to generate code, depending on how much tuning and backward compatibility is required. The most commons are the `-ax`, `-x` and `-march` & `-mtune` options. The following are the one-line description from the Intel documentation.

**`-ax`** Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel processors if there is a performance benefit.

**`-x`** Tells the compiler which processor features it may target, including which instruction sets and optimizations it may generate.

**`-march`** Tells the compiler to generate code for processors that support certain features.

**`-mtune`** Performs optimizations for specific processors.

The usual way is to use either one of the first two (`-ax` or `-x`), or the last two as a pair. The easiest solution is probably the special architecture parameter "native" to `-march` & `-mtune`[3], which instructs the compiler to generate and optimize code for whatever features is supported by the current CPU. It has the advantage that as long as a code is compiler on the machine it will run on, it will always use all available features. In case the compilation and running host are different, then a specific kind of architecture should be picked, i.e. for AVX2 `-march=core-avx2 -mtune=core-avx2`.

*2) GNU compiler:* Recent GNU compilers "gcc" offer a similar interface to the Intel compiler for `-march` & `-mtune`, and also support the special "native" architecture. It does not have the `-ax` or `-x` options, but it offers a large set of instruction set-specific switches, such as `-mavx`, `-mavx2` or `-maes`.

*3) CLANG compiler:* Recent LLVM-based compilers "clang" supports a similar interface to the GNU compiler, including the "native" micro-architecture on x86-64.

### B. Identifying assembly

It is sometimes useful to assert what kind of code is the binary made of. It can be because the code is part of a binary-only librarie, or because it is not known how the code was compiled, or simply to check the behavior of the compiler and whether the code was properly vectorized or not. It is therefore useful to be able to identify the most common floating-point operations in their various flavors, including the obsolete x87 FPU.

*1) The x87 FPU:* The x87 FPU does not make use of numbered registers, but is stack-based. Most instructions are readily identifiable, as they will reference this special FPU hardware stack. The name of the stack in assembly language is `%st`, with a possible suffix indicating an element not at the top of the stack such as `%st(4)`. So if the code makes use of the x87 FPU (which is to be avoided), then the assembly code will show instructions such as `fmul %st(4),%st` (a multiplication) or the

[3] i.e. `-march=native -mtune=native`

very common `fxch %st(1)` (exchanging the top of the stack with another element).

*2) SIMD instructions: SSE, AVX:* There is two things to identify when checking SIMD assembly code: the width of the registers (scalar i.e. non-vectorized, 128 bits, 256 bits) and the encoding (SSE or AVX). Floating-point SIMD instructions in assembly are made according to a simple pattern:

1) An optional prefix `v` indicating AVX encoding;
2) The mnemonic indicating the type of operation (`mul`, `add`, ...);
3) Two single-letter suffixes indicating first scalar (`s`) or vectorized ("packed", in the Intel documentation) (`p`), and second the type of data (single precision `s` or double precision `d`).

So the instruction `mulss` is a SSE-encoded (no `v` prefix) scalar single-precision multiplication, while `vaddpd` is an AVX-encoded packed/vectorized double-precision addition. In this second case, the instruction mnemonic doesn't specify the width of the registers (128 or 256). 128 bits registers (SSE or AVX-encoded) are identified by the prefix `xmm`, whereas 256 bits registers are identified by the prefix `ymm`. Whenever an instruction is scalar, registers normally use the `xmm` prefix. SSE-encoded instructions cannot use `ymm` registers. And additional clue is that common binary operations have two registers in SSE, but three in AVX.

So for instance, a double-precision addition might be:

**`addsd %xmm0,%xmm1`** SSE-encoded scalar addition;

**`addpd %xmm0,%xmm1`** SSE-encoded 128 bits ($2 \times 64$) addition;

**`vaddsd %xmm0,%xmm1,%xmm1`** AVX-encoded scalar addition;

**`vaddpd %xmm0,%xmm1,%xmm1`** AVX-encoded 128 ($2 \times 64$) bits addition;

**`vaddpd %ymm0,%ymm1,%ymm1`** AVX-encoded 256 ($4 \times 64$) bits addition;

## VI. Conclusion

In this paper, we investigate what is the theoretical peak in floating-point operations of a compute node. We note that it is no longer a single number, but the actual value must take into account the instruction set and its encoding, and also the level of parallelism and the specific of the CPU such as frequency scaling. Since the implementations detail of the actual binary is involved, the compiler and its options will play an important role in constraining the available peak. The binary code must be suited to the underlying architecture.

In the future, it will be necessary to take into account the upcoming AVX-512 instruction set [3][4] which again double the vector width. Fortunately, the encoding is compatible with AVX and FMA, so it should only add two entries in our tables, for single and double precision using the new vector width - doubling the peak against the current AVX+FMA leader.

## Glossary

**AVX** The third SIMD instruction set for the x86 architecture. It uses 256 bits wide registers. The original AVX instruction set only support floating point operations. AVX2 introduced integer operations. 2, 9, 10

**AVX2** First extension to the AVX instruction set, adding integer operations. 3, 9

**CPUID instruction** The CPUID instruction is a mechanism for x86 and x86-64 architectures by which software can tell among other things which instruction sets are supported in a particular CPU. 3

**dependency chain** A sequence of sequentially dependent instructions. If an instruction I1 writes to a register RA and a later instruction I2 reads register RA, then I2 is said to be dependent on I1. If I2 writes to RB, and I3 reads RB, then I3 is dependent on I2, and I1-I2-I3 forms a dependency chain. The total latency for a dependency chain is never lower than the sum of all the latencies for all the instructions in it. 6

**encoding scheme** The encoding is the binary representation of an instruction, directly usable by the processor. An encoding scheme is the description of how to encode the various instructions. The AVX encoding scheme differs from the SSE encoding scheme, as it adds some extra bytes to enable the use of wider registers and extra operands. 3

**Floating-Point Unit** The computation unit responsible for floating-point operations. It can be composed of multiple execution units, each of

which are sometimes also called FPU. It can be scalar (such as the x87 FPU, or SIMD (such as SSE or AVX). 2, 10

**FPU** Floating-Point Unit. 2

**instruction set** A group of instructions that are available - or not - in a micro-architecture. The instruction set with the mandatory instructions is the base instruction set, such as x86 or its 64 bits variant x86-64. Additional instruction sets such as x87, SSE or AVX provide additional instructions to expand the capability of a processor. 2, 3, 9, 10

**micro-architecture** The details of how an architecture is implemented in a given micro-processor. 1

**SSE** The second SIMD instruction set for the x86 architecture. It uses 128 bits wide registers. The original SSE instruction set only support integer and single-precision floating point operations. SSE2 introduced double-precision floating-point operations. Both SSE and SSE2 are required in x86-64 (64 bits) processors. 9, 10

**SSE2** First extension to the SSE instruction set, adding double-precision floating point operations. 2, 10

**x87** It is the common abbreviation for the first, now obsolete, floating-point extension to the x86 architecture. See e.g. [10] for details. 2, 10

## REFERENCES


[1] R. Dolbeau, "Theoretical Peak FLOPS per instruction set on less conventional hardware," 2015. [Online]. Available: http://www.dolbeau.name/dolbeau/publications/peak-alt.pdf

[2] Intel®, "Intel® Xeon® Processor X5650 (12M Cache, 2.66 GHz, 6.40 GT/s Intel® QPI)." [Online]. Available: http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI

[3] ——. (2013) AVX–512 instructions. [Online]. Available: http://software.intel.com/en-us/blogs/2013/avx-512-instructions

[4] ——. (2014) Additional AVX–512 instructions. [Online]. Available: https://software.intel.com/en-us/blogs/additional-avx-512-instructions

[5] ——, Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A–Z, September 2014, no. 325383-052. [Online]. Available: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-instruction-set-reference-manual-html

[6] ——, "Intel® Xeon® Processor E5-2695 v3 (35m Cache, 2.30 GHz)." [Online]. Available: http://ark.intel.com/products/81057/Intel-Xeon-Processor-E5-2695-v3-35M-Cache-2_30-GHz

[7] ——, "Optimizing Performance with Intel® Advanced Vector Extensions," 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf

[8] ——, "Intel® Xeon® Processor E5 v3 Product Families Specification Update," 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v3-spec-update.pdf

[9] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA cpus," Copenhagen University College of Engineering, 1996 – 2014. [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf

[10] Wikipedia. x87. [Online]. Available: https://en.wikipedia.org/wiki/X87