Assembly micro-benchmark generator for characterizing Floating Point Units

Jean Pourroy, Patrick Demichel, Christophe Denis

Assembly micro-benchmark generator for characterizing Floating Point Units

Jean Pourroy
CMLA-CNRS, ENS Paris-Sclay
Cachan, 94230, France
jean.pourroy@ens-paris-saclay.fr

Patrick Demichel
Hewlett Packard Enterprise
Grenoble, 38000, France
patrick.demichel@hpe.com

Christophe Denis
CMLA-CNRS, ENS Paris-Sclay
Cachan, 94230, France
christophe.denis@ens-paris-saclay.fr

***Abstract*—Making the right platform choice has always been a challenge for the HPC users no matter the applications vertical they are in. The number of references is very large and making the wrong choice can have adverse effects. Formerly users only had to choose between, for example, the different processors and interconnect vendors. Lately, due to the new Intel Skylake processors the choice has become increasingly difficult as different levels of performance are available within the same vendor platforms. To facilitate selection and give possible directions for the real benchmarked applications we introduce the Kernel Generator, an open source tool generating assembly kernels to help the programmer or the benchmarker understand the behavior of the different micro-architectures. We used our tool to study the behavior of the current micro-architectures and compare it to the current synthetic benchmarks which sometimes are not correctly characterizing a platform nor expose its strengths. The Kernel Generator facilitates the discovery of the platforms performance fit. To insure the relevance of our kernel, we are looking at Ansys Fluent behavior to explain the performance on the different Intel processors. In this case, we have that 4100 and 6100 Intel processors families can have equivalent performance on codes not well vectorized: Fluent being one of them. This demonstrates that we can use our tool for initial profiling and understanding of the different platforms.**

Keywords: **Architecture characterization; vectorization; FPU**

I. Introduction

Today, the HPC domain is constrained both from the economic and the feasibility perspective. Using the correct Supercomputer in place became an important pillar which carefully needs to be implemented as it involves taking some decisions like: choice of vendor, the choice of components but also the total cost of ownership and data center foot print for example. From the platform perspective we do have some important choices ahead as Intel released their new Skylake processors [1], AMD just announced the EPYC processors, but one can also look at accelerators like GP-GPUs from Nvidia and AMD [2] Many-core technologies like Intel Knight Landing [3] but also the latest arrivals into HPC like the FGPAs from Intel or Xilinx [4]. Not all applications are able to run on all the different processors or accelerators above. In order to be able to run on GPUs, KNLs or FPGAs applications either need to be written from scratch. Applications can be ported using CUDA (for Nvidia GPUs) or OpenCL (AMD GPUs and FPGAs). As the effort of application porting is usually too big, most of today's applications today are still running on CPUs only. This does not mean that the choice of CPU is an easy task and most companies run platform benchmarks or proof of concepts before purchasing a large scale system. Benchmarks can be micro-benchmarks like: HPL [5], Stream [6], HPCC [7] or HPCG [8] or real applications benchmarks. This exercise is planned very carefully for both the customer and the vendor. The real problem is that the current synthetic benchmarks do not always show the strengths of all platforms thus not even allowing some CPUs to be considered for the application benchmarking. What we have seen in the past months with the releases of the new Skylake processors is that entry level processors like Intel 4110 can perform as good as a top bin processor like the Intel 6148 for real applications thus increasing the $\frac{performance}{price}$ of a large scale cluster. To make this visible and the choice of the CPU more informed we created a tool called the Kernel Generator able to correctly assess in which type of applications a processor would best perform such as for example Computational Fluid Dynamics, Structure, Life Sciences. Together, with the Kernel Generator, we use monitoring tools written by our team to understand what is occurring on processors during runs. This paper is organized as follows: Section 2 introduces the state of the art. Section 3 details the Kernel Generator and its functionality. Section 4 exposes some experimental results using Ansys Fluent. Finally in Section 5 we conclude and propose future directions.

II. Technology background and related benchmarks

A. Intel technology

Since 2007, Intel has become accustomized to releasing its processors based on a tick tock model [10]. When a tock presents a new architecture, a tick represents a shrinking of this micro-architecture. In 2017, Intel introduced its new server processor portfolio with its latest architecture, Skylake, replacing its Broadwell predecessor. For these server-only processors, Intel has separated its portfolio into 4 ranges: Bronze (model 31XX), Silver (model 41xx), Gold (model 51xx and 61xx) and Platinum (model 81xx). On Broadwell, even if the number of type of processors was bigger, we had easier choice as only one type of processor was feasible for HPC (E5-26XXv4). With this new architecture, Intel offers a large number of processors with great technological differences but

TABLE I: Skylake portfolio: major difference between SKUs

	Bronze: 31XX	Silver: 41XX	Gold: 5100	Gold: 6100	Platinium: 8100
Memory channel and speed	6-ch@2133 Ghz	6-ch@**2400**	6-ch@2400	6-ch@**2666**	6-ch@2666
UPI links (Scalability)	2 (2S-2UPI)	2 (2S-2UPI)	2 (4S-2UPI)	3 (2S-3UPI)	3 (8S-3UPI)
UPI bandwidth	9.6 GT/s	9.6 GT/s	10.4 GT/s	10.4 GT/s	10.4 GT/s
HyperThreading [9]	NO	**YES**	YES	YES	YES
FMA-512 FPU	1	1	1	**2**	2

also price differences. Our goal is to understand the differences between these models and take advantage of them (Table I).

The Skylake portfolio, divided into 5 groups of SKUs, categorizes processors their architecture and the available features. The portfolio ranges from the simplest (Bronze) to the most powerful (Platinum) processors. Table I presents the major changes. The Bronze range will be quite difficult to use for HPC, these processors not having turbo or HyperThreading [9] and only compatible with the slowest memory (2133 Ghz). As we move up in range new features appear. The silver range brings TurboBoost [11] and HyperThreading. Our experience as an HPC vendor shows that the 4100 and the 5100 SKUs from Intel never were considered as serious candidates for HPC business addressed so far. It represents the lower-end of the Skylake portfolio and never appeared to be the right choice for any application. Except a few of them, they usually offer fewer cores per socket and degraded specs (Intel QuickPath Interconnect (QPI) links and memory frequency supported). This is what triggered our consideration for a new tool correctly assessing the processors for the different HPC applications.

The main difference between 8100/6100 and 5100/4100 series remains their ability to perform 1 or 2 AVX512 instructions per cycle. For years, vectorization has been presented by Intel as the only viable approach to improve the performance of HPC applications, as the hardware does not improve in terms of clock speed. Vectorization is not necessarily sufficient to improve the performance, if all available floating-point units are not fully leveraged. What's the benefit of using 512-bit wide instructions if only one is issued every few cycles? To overcome the need to provide enough data to the Floating Point Unit (FPU), this new generation of processors increases the number of memory channels between the processor and memory from 4 channels (Broadwell) to 6, which are compatible with different memory speed, from 2133 to 2666 GHz.

B. Related work

To measure and compare computers architectures it is common to use code called benchmarks. They can be used as a baseline in the industry as their behavior and their performance is well known. Benchmarks are also used to determine the characteristics of an architecture. Although it can be found in technical manuals, it is common for achievable performance to be lower. The performance of an application can then be estimated correctly by comparing it to practical ($Performance_{max}$) rather than theoretical performance ($Performance_{peak}$). There are several types of benchmarks that have different purposes and can be classified into three families [12]: *benchmarks*, *kernel-based benchmarks* and *micro-benchmarks*.

Benchmarks are complete applications that perform different types of calculations. In some cases, real applications are used as reference benchmarks. We could cite some of them such as BSMBench [13] designed to perform the pattern of computation used in Lattice Gauge Theory (LGT) field.

Kernel-based benchmarks are usually smaller codes than benchmarks. A kernel is a function that has been extracted from a larger application to measure only its performance. The most used and well known benchmarks are HPL [5] and Stream [6]. The High Performance Computing Linpack (HPL) benchmark is used to sort the *Top500* [14] which is ranking the 500 most powerful supercomputers since 1993 [15]. It is used to measure the maximum Floating Points Operation per Second (FLOPS) a supercomputer is able to provide by solving a linear system of equation using LU decomposition. The second worldwide known benchmark is STREAM that was originally developed to understand the performance differences between two architectures for a weather application. The main problem with those benchmarks comes from their artificial code not representing real application. To this end, other specific benchmarks have been developed to reflect how real application would perform. On recent HPC architectures, having a good HPL performance is no longer correlated with having good performance with real applications, so the HPCG was introduced [8]. The HPCG benchmark fills the gap by implementing some micro-benchmarks more representative of the workload used in the industry as sparse matrix-vector multiplication or sparse triangular solvers. Some benchmarks use generators to create benchmarks that cover more cases using different parameters, data sets or resolution functions. There are kernel generators available such as BMG [16] to generate benchmarks with different parameters for the Travelling Salesman Problem (TSP).

Lastly, the *micro-kernel benchmark* approach is an extension of the kernel-based benchmarks that are used to isolate and measure a specific part of the micro-architecture. For example, lmbench [12] is a suite of portable benchmarks used to measure important characteristics of the memory such as bandwidth, memory latency and performance of the different cache levels.We also mention [17], [18] for the characterization of the different levels of the memory hierarchy. Thanks to those benchmarks, the collected information can be used to predict the performance of applications. Also, the recovered information allows to implement optimizations such as *loop tiling* to fit perfectly in the different levels of caches. For example, computational libraries such as ATLAS [19] and

TABLE II: Skylake performance the HPL Benchmark in GFLOP/s: 8 process per core, HyperThreading off, frequency capped at 1.5 Ghz)

MKL instructions set	Silver: 4110	Gold: 5117	Gold: 6130	Platinium: 8160
STREAM (256)	361	362	362	363
STREAM (512)	297	372	714	716

FFTW [20] generate micro-benchmarks to characterize the architecture. Dozens of different versions are executed and measured, to keep only the most efficient version. As the quality of the micro-benchmarks used is important to properly characterize the architecture, work has been done to improve them [21].

III. Kernel Generator

Today, disruptive architectures are released every month in the HPC domain and it becomes hard to exactly understand what performance they are able to provide. In this paper we are presenting an Open Source C++ tool, called Kernel Generator. This tool can be downloaded on our Github repository [22]. It is used to generate assembly kernel to finely characterize some part of the processors architecture and more precisely the Arithmetic Logic Unit (ALU) which is in charge of the execution of floating point units. The Kernel Generator is a tool used to finely describe the behavior of the Floating Point Units (FPU) of the processor and answers questions like: How does the processor behave when running vectorized instructions? How the frequency is changing running AVX-512 on all the processors cores? Indeed, FPU is a crucial component of the processor as most of HPC applications are executing intensive compute on floating point data.

A. Basic concept: assembly kernel

The Kernel Generator allows a user to finely benchmark the arithmetic logical unit by generating an assembly loop with different instructions type and precision. Instructions can be scalar or SSE, AVX2, AVX512, and by self monitoring its loop, the micro-benchmark generated will print some relevant information such as the number of instructions that can be executed by cycle (IPC), the frequency and the number of GFLOP/s. In generating an assembly kernel our tool accepts several options:

- `--width {64, 128, 256, 512}`: set instruction's width. Multiple types can be mixed.
- `--operation {a,m,f}`: Type of operations generated: addition (a), multiplication (m) or Fused Multiply Add (f) (can be mixed)
- `--precision {single, double}`: set instruction's precision.
- `--dependency {true, false}`: Is an instruction dependent on the result of the previous one?
- `--loopsize N`: set the number of samples executed
- `--unroll N`: unroll the kernel generated

For example, the generator can be used as follows to create a kernel composed of 4 AVX2 (256-bits wide) single precision instructions: two additions and two multiplications:

```
./kg   -P single -W 256  -O aamm
```

This generates the kernel shown in Listing 1. The code template is stored in a file. The generator copies it and sets some parameters such as *NB_LOOP*. It also inserts the appropriate floating point instruction (line 5 to 8). We measure its performance by surrounding it with two calls to the `rdtsc` [23] assembly instruction [24]. Finally, we also surround the kernel by two calls to the system function `gettimeofday`. To maximize the accuracy of our measure, we actually run this whole code several times and do the average at the end. Listing 1 shows the code generated by the previous command.

```
1  for (i = 0; i < NB_lOOP; i++) {
2     timeStart = mygettime();
3     cycleInStart = rdtsc();
4     __asm__ ("myBench: "
5         "vaddss % % xmm0, %%xmm1, %%xmm2;"
6         "vaddss % % xmm0, %%xmm1, %%xmm3;"
7         "vmulss % % xmm0, %%xmm1, %%xmm4;"
8         "vmulss % % xmm0, %%xmm1, %%xmm5;"
9     "sub  $0x1, %%eax;"
10    "jnz  myBench;"        :
11    "=r"(instructions_executed) : "a"(NB_lOOP_IN))
      ;
12    cycleInEnd = rdtsc();
13    timeEnd = mygettime();
14    cycle_total += (cycleInEnd - cycleInStart);
15    time_total += timeEnd - timeStart;
16 }
```

Listing 1: Generated code with the previous command

```
1                  INSTRUCTIONS SUMMARY
2 NB_INSTRUCTIONS  Time  FREQ Giga_inst/s   IPC
3    32000000000  7.71   2.1         4.15     2
4                      FLOP SUMMARY
5 PRECISION        FLOP/cycle       FLOP/second
6 Single                 1.98          4.14e+09
7 Double                    0                 0
```

Listing 2: Summary of the kernel generator: instructions and FLOP

B. Iteration measurement validation

Our monitoring code is measuring the number of cycles used to execute the assembly code the user wants to profile, and the two instructions used to control the loop iterations. As shown in Listing 1, both `rdtsc` instructions are executed around the loop, one just before the label declaration (line 3) and the second one is executed after the two instructions needed for the loop, `sub` and `jnz` (line 12). The following experience shows that to calculate the number of Instructions per Cycle (IPC) those two instructions should not be taken into account. We have been using a specific executable to measure the actual clock speed a processor would adopt when running

some code (Listing 3).Such a loop can be performed every cycle by Xeon processors since SandyBridge. To be sure of that, we monitored it with the command `perf stat ./kg` and that is measuring an IPC of 1.

```
__asm__ ("myBench: "
  "sub  $0x1, %%eax;"
"jnz  myBench;"
```

Listing 3: Measure the impact of jzn and sub

We believe, the branch instruction (`jnz`) does not affect the time execution as the branch predictor is very efficient and specially for those kind of simple loop [25]. So the only instructions that's spending cycles in this loop is the `sub`. To measure its impact on a more realistic kernel, we have generated the kernel shown in Listing 4 by using the following command:

```
/kg  -P double -W 512 -O ffff
```

Then we have manually added subtraction instructions on `%%ebx` to measure their impact.

```
"myBench: "
 "vfmadd231pd %%zmm0, %%zmm1, %%zmm2; "
 "vfmadd231pd %%zmm0, %%zmm1, %%zmm3; "
 "vfmadd231pd %%zmm0, %%zmm1, %%zmm4; "
 "vfmadd231pd %%zmm0, %%zmm1, %%zmm5; "
 "sub  $0x1, %%ebx;" //fake substraction
 "sub  $0x1, %%ebx;" //fake substraction
 "sub  $0x1, %%ebx;" //fake substraction
 "sub  $0x1, %%eax;"
"jnz  myBench;"
```

Listing 4: Measure the impact of a sub instruction

We used an Intel 4110 processor that we know is able to execute one 512-bits FMA instruction per cycle and it was only after adding 3 fake additions that the performance deteriorated. From this experience we conclude that one sub instruction has no impact on kernel performance (if its not the only instruction in the loop). We have shown that even if the FMA-512 FPU is used by an AVX-512 instruction, then the processor is able to perform up to 4 subtractions on registers in one cycle as we have measured an IPC equal to one by using `perf stat`. We assume those two instructions do not affect the result so we exclude them from the calculation of the IPC. Moreover, to reduce any potential noise caused by these two instructions, we have added an option to unroll the assembly instructions several times inside the loop. Thus the proportion of both instructions is reduced.

C. Frequency validation

The faster a processor runs, the more power it requires. For this reason, Intel has adopted different frequency level for their processors to gain performance when needed and limit the power consumption when not. The instruction `rdtsc` is used to read the hardware counter corresponding to the number of ticks since the last reset of the processor [23] The frequency of those ticks is independent from the actual clock frequency and this frequency correspond to the Rated Base Frequency on Intel processor. If the processor frequencies were to vary, measurements made using `rdtsc` would be wrong. This is why the processor frequency must be set before the execution of the micro-benchmark. In case of a fixed frequency different from the Rated Base Frequency, we have implemented a verification that will then calculate this frequency, and adjust all results measured by `rdtsc`.

1) Measuring the Base Frequency: We have developed a code to measure the base frequency. It uses the *sleep* function which wait a certain number of micro-seconds. And also the `rdtsc` instructions that returns the number of cycles spent based on the base frequency. Finally we can calculate the frequency with the ratio $cycleSpent/timeSpent$ as shown on Listing 5

```
timeStart = mygettime();
cycleInStart = rdtsc();
    usleep(10000);
cycleInEnd = rdtsc();
timeEnd = mygettime();
cycleSpent = (cycleInEnd - cycleInStart);
freq_Base = cycleSpent / (timeEnd - timeStart);
```

Listing 5: Code used to measure the base frequency of the processor

2) Measuring the current frequency: On any modern processor, a sub instruction on a register can be executed every cycle. Listing 6 showns a loop that will take 800000000 iterations. Knowing that what is inside the loop is supposed to run with an IPC of 1, this loop should be executed in 80000000 cycles.

```
cycleInStart = rdtsc();
__asm__ ("aloop: "
    "sub $0x1,%%eax;"
    "sub $0x1,%%eax;"
    "sub $0x1,%%eax;"
    "sub $0x1,%%eax;"
    "jnz aloop" : : "a" (80000000UL)
);
cycleInEnd = rdtsc();
cycleSpent = (cycleInEnd - cycleInStart);
```

Listing 6: Measuring the current frequency by subtracting 1 from a register

We can calculate the number of instructions executed by cycle as follows $80000000/cycleSpent$, and with this result we can conclude:

- `IPC == 1`: The processor is running at its based frequency.
- `IPC < 1`: The processor is running at a lower frequency than his based frequency (capping).
- `IPC > 1`: The processor is running at a higher frequency than his based frequency (Turbo).

3) Adjusting rdtsc results: In our experimentation, we use an external script to lock the frequency of processor to be sure it will run at this specific frequency. We locked the frequency of the Intel Xeon E5-2690v4 at 2.00 Ghz. This processor has a base frequency equal to 2.60GHz. Our code presented in the previous section measures an IPC for the Listing 6's code of 0.768. This means that it is actually running

at 76.8% of its base frequency, or 2.00Ghz. Thats experiment is here to validate our methodology based on `rdtsc`, but also to validate before any experimentation that the frequency is correctly set. User can choose to activate the frequency check by using the option *–frequency true*. Then the Kernel Generator will execute this code, to check the value of the current frequency and adjust the results given by `rdtsc`. It will multiply or divide the original value by the ratio $BasedFreq/MeasuredFreq$.

D. Validation of the Kernel Generator results

As we are working on very fine, critical measurements, such as counting the number of cycles, we had to verify the results provided by the Kernel Generator. To do this, we used 3 different ways: first we used an internal tool called `mygflops`. Then we developed a self-check option directly in the tool and finally we simply monitored the execution by using *perf*.

1) Validation with mygflop: The first way we used to control the correctness of our tool was one of our internal script based on hardware counters, which allows us to count and sort the floating point instructions that are executed. The Listing 7 is showing the results given by `mygflops` tool that allows us to validate two values: we can validate that the micro-benchmark generated really executes only 32-bits wide instructions and that the throughput of the program is effectively 4.15 GFlop/s (see Listing 2).

```
++ Single-p: 4.155284 GFlop/s -- 100.0%
100.0%  scalar  32-bit inst (100.0% )
0.0%    packed 128-bit inst (  0.0% )
0.0%    packed 256-bit inst (  0.0% )
0.0%    packed 512-bit inst (  0.0% )

++ Double-p: 0.000004 GFlop/s --   0.0%
100.0%  scalar  64-bit inst (  0.0% )
0.0%    packed 128-bit inst (  0.0% )
0.0%    packed 256-bit inst (  0.0% )
0.0%    packed 512-bit inst (  0.0% )
```

Listing 7: Validation of the type of instructions executed with *mygflops*

2) Self validation: Next, we have added an option that checks within the kernel that the correct number of operations have effectively been executed. Processors having become very sophisticated, they execute a useless code and can skip it. For example, to check the number of additions that have been done, we initialize the two registers used by those instructions as source, with the value one, and perform an addition in other registers. At the end, we do a reduction on those registers to count exactly the number of operation that have been executed. And this number should correspond to the *Number Of Iterations * Number Of Additions*.

3) Perf validation: The last way we use to validate our measurements is to use the *perf* command as follows: `perf stats ./micro-benchmark`, the ouput of this command is shown in the Listing 8.

```
2,008,852,256  cycles       # 2.099 GHz
6,010,024,733  instructions # 2.99  insn per cycle
```

Listing 8: Output of the perf command used to validate the frequency and the IPC

With this output, one user could check if the frequency was correctly set before the experiment (2.10 Ghz which is the base frequency of a Skylake 4110). By looking at the IPC measured, `perf` finds an IPC of 3 instead of 2. As mentioned above we considered that both instructions `jnz` and `sub` had no impact on the execution of the loop. Here we check it, in this loop 6 instructions are executed (2 add, 2mul, 1 sub and 1 jnz). Since 2 add and 2 mul can be executed every cycle, the processor executes those 6 instructions in 2 cycles, i. e. an IPC of 3.

E. Experimental work on Broadwell

We performed experimental tests on Broadwell processors during the development of Kernel Generator. We have run a simple test: generate a kernel containing only multiplications (Listing 9), on a Broadwell e5-2690v4 processor. The kernel was generated with the following command:

```
./kg  -P double -W 64 -O mmmmm
```

The expected result, IPC of 2, being well displayed by our tool (Listing 10).

```
for (i = 0; i < NB_lOOP; i++) {
  timeStart = mygettime();
  cycleInStart = rdtsc();
  __asm__ (""
    "myBench:"
      "vmulsd %%xmm0, %%xmm1, %%xmm2; "
      "vmulsd %%xmm0, %%xmm1, %%xmm3; "
      "vmulsd %%xmm0, %%xmm1, %%xmm4; "
      "vmulsd %%xmm0, %%xmm1, %%xmm5; "
      "vmulsd %%xmm0, %%xmm1, %%xmm6; "
    "sub  $0x1, %%eax;"
    "jnz  myBench;"   :: "a" (NB_lOOP_IN));
  cycleInEnd = rdtsc();
  timeEnd = mygettime();
  cycle_total += (cycleInEnd - cycleInStart);
  time_total += timeEnd - timeStart;
}
```

Listing 9: Generated micro-benchmark

```
NB_INSTRUCTIONS  Time  FREQ Giga_inst/s   IPC
    40000000000  7.71   2.1        4.15     2
```

Listing 10: The Broadwell architecture is able to execute 2 scalar multiplications per cycle

Then, we checked the correctness of the tool with addition instructions.

```
./kg  -P double -W 64 -O aaaaa
```

And we obtained the following results:

```
NB_INSTRUCTIONS  Time  FREQ Giga_inst/s   IPC
    40000000000  14.43  2.1        2.71     1
```

Listing 11: Broadwell experiment: only one scalar additions per cycle

We first thought that a mistake was done as it is known that Broadwell processor can fetch and execute two instructions per cycle. In fact it is a specific behaviour of the micro-architecture which has already been found before here. Even if Broadwell processors are not being requested anymore, we needed to expose how simple it is for an HPC user to characterize his processor and find interesting behaviors of the micro-architecture.

F. Describe Silver and Gold SKUs

To understand how the Skylake architecture is behaving, we have generated several micro-benchmark with the Kernel Generator. This section presents some relevant results on Silver and Gold.

1) Addition latency: : To avoid any dependency between instructions, the tool is generating them with different destination registers (3rd arguments of the assembly instructions), this can be controlled by an option if one wants to have dependent instructions. We used the `--dependency` option to generate instructions that are using as source, the result of the previous instruction. By running the following command we can find the latency of an instruction:

```
./kg  -O a --dependency true
```

```
"myBench: "
      "vaddsd %%xmm0, %%xmm2, %%xmm2; "
"sub  $0x1, %%eax;"
"jnz  myBench;"
```

Listing 12: Code used to measure the addition instruction latency

```
NB_INSTRUCTIONS  Time  FREQ Giga_inst/s   IPC
     1000000000  1.49  2.72       0.672  0.25
```

Listing 13: Result of a kernel generated with the dependency option

The Listing 13 shows that a Skylake processor is able to execute the kernel with a CPI (cycle per instruction) of 4 cycles. Such instruction uses, as source, the result of its previous executions, 4 cycles represent the latency for an addition, which is validated by Intel documentation [26].

2) FMA-512 FPU for AVX512 instruction: : In the 4110 processor documentation [27], Intel indicates that it has only one FMA-512 unit. We also know that to be able to run AVX-512 instructions, processors run at their AVX Base frequency, which is lower than the Base Frequency. This of drop depends on two things: the number of cores actually running those instructions, and the size of the instructions. In order to get rid of any frequency drop, we measured this frequency, and we used it to cap the processor (1.60 Ghz for the 4110). As the frequency is not changing, our tool can find and adjust the results if it is launched with the *–frequency* option (see subsubsection III-C3).

```
./kg  -P double -W 512 -O ffffffff -F true
```

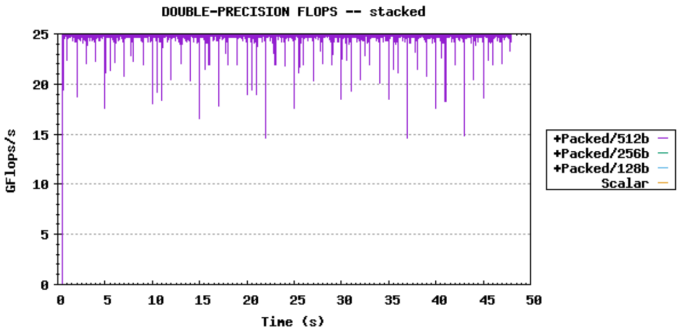

Fig. 1: Graphic view of the kernel run generated with an intern tool based on mygflops

```
-- CHECK FREQUENCY  --
+ Base      frequency is 2.10Ghz
+ Current   frequency is 1.60Ghz
+ /!\ The frequency seems to be capped: -24.0%
--    RESULTS       --
NB_INSTRUCTIONS  Time  FREQ Giga_inst/s   IPC
     1000000000  5.04  1.60        1.59  1.00
```

Listing 14: Generation of AVX-512 FMA micro-kernel

We monitored the run with an internal tool that is based on `mygflops` is showing the evolution of the flops during the run:

Figure 1 shows that the processor is executing only 512-bits wide instructions with a throughput of 25 GFlop/s. As the processor is running at 1.6 Ghz, we calculate how many `flop` it is doing per cycle $\frac{25}{1.6} \approx 16 \frac{FLOP}{cycle}$. As the instructions are running on double precision values, an AVX512 instruction can process 8 elements and as an FMA is a fused instruction, every cycle it executes 2 operations. So, theoretically an FMA-512 can execute 16 FLOP/cycle, which matches to the results we have measured.

3) Using the Kernel Generator to understand HPL results: If we look at the results of the HPL benchmark compiled with AVX-256 instructions (Table III), we can see an abnormal behavior. Indeed, the two SKUs Silver and Gold obtain rigorously the same results while the second one has 1 FMA-512 more than the first one. With the Kernel Generator, it has been easy to generate a kernel composed of AVX2 instructions and benchmark both processors.

```
./kg  -P double -W 256 -O ffffffff -F true
```

```
"myBench: "
  "vfmadd231pd %%ymm0, %%ymm1, %%ymm2; "
  "vfmadd231pd %%ymm0, %%ymm1, %%ymm3; "
  "vfmadd231pd %%ymm0, %%ymm1, %%ymm4; "
  "vfmadd231pd %%ymm0, %%ymm1, %%ymm5; "
  "vfmadd231pd %%ymm0, %%ymm1, %%ymm6; "
"sub  $0x1, %%eax;"
"jnz  myBench;"
```

Listing 15: Generated AVX2 Kernel

The results of those runs are shown in Table II. And a graphic view from `mygflop` is shown Figure 2. This test allowed us to discover a major functionality of the Skylake

TABLE III: Xeon Gold and Silver results for AVX2 instructions

	Silver: 4110	Gold: 6130
IPC	2	2
GFLOP/s	2.38e+10	2.37e+10

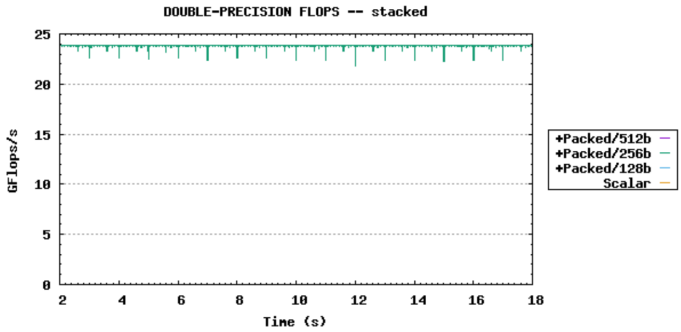

Fig. 2: Xeon Gold and Silver obtains equivalent results when running AVX2 instructions

architecture. The FMA-512 FPU of the Intel 4100 processor able to fuse two 256-bits instructions to fit the FMA-512 unit and to execute two 256-bits instructions per cycle. However, the 6100 processor is not able to perform this, and does not therefore take advantage of having two 512-bits FPUs [28]. This discovery is in fact an opportunity, as a large majority of HPC applications do not use AVX512, they could have similar performance on both architectures which are far from having similar prices. To complete this analysis we wrote a script using a batch scheduler, *Slurm*, to execute kernels on all our Skylake processor. With few lines of code, we were able to execute and to compare the performance of scalar and vectorized instructions between Skylake SKUs. Figure 3 is showing the results for double precision instructions.

4) FMA-512 FPU benchmark: : We then used the generator to create kernels with mixed instructions of different sizes. For example, we have generated a kernel with AVX-512 and AVX2 instructions mixed as shown in Listing 16.

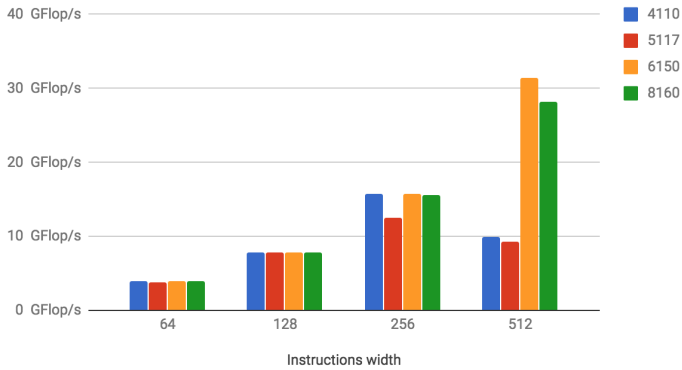

Fig. 3: Performance overview of a processor core depending on the instruction width. (double precision)

TABLE IV: Skylake portfolio: FPU micro-benchmark with different instruction width combination

Instructions mix	4110 (1)	5117 (1)	6130 (2)	8160 (2)
128 and Scalar	2	2	2	2
256 and Scalar	2	2	2	2
256 and 128	2	2	2	2
512 and Scalar	1	1	2	2
512 and 128	1	1	2	2
512 and 256	1	1	2	2

```
"myBench: "
  "vfmadd231pd %%zmm0, %%zmm1, %%zmm2; "
  "vfmadd231pd %%ymm0, %%ymm1, %%ymm3; "
  "vfmadd231pd %%zmm0, %%zmm1, %%zmm4; "
  "vfmadd231pd %%ymm0, %%ymm1, %%ymm5; "
  "vfmadd231pd %%zmm0, %%zmm1, %%zmm6; "
  "vfmadd231pd %%ymm0, %%ymm1, %%ymm7; "
"sub  $0x1, %%eax;"
"jnz  myBench;"
```

Listing 16: Kernel with mixed AVX-512 and AVX2 instructions

In the same way we have carried out other tests by mixing the different types of instructions so the results are presented in Table IV. It shows how many instructions can be retired every cycle depending on their width. From these numbers, we can conclude that an FMA-512 FPU can retire the following combinations of floating point instructions every cycle, assuming no dependency between registers:

- Any single 512-bit wide instruction
- Two 256-bit wide instructions
- Two 128-bit wide instructions
- Two scalar instructions
- Any combination of 2 instructions providing the aggregated width does not exceed 512 bits.

IV. Real case: Fluent, a CFD application

There are a lot of processors released within the Skylake architecture and finally this year AMD released its new Epyc processor as well. To find the best platform for ones application, one would have to benchmark all of them. On the one hand, benchmarking all available processors would be time consuming as some applications take several hours to execute. On the other hand, as we have shown in our experiments, targeting only high end processors may lead to missing some opportunities with cheaper processors and equivalent performance could be missed. In this section we are investigating a methodology using the kernel generator to help users in their choice of platforms. To conduct our study, we placed ourselves in a real situation by studying Fluent [29], an application widely used by our customers, particularly in Formula 1. As we want to make fair comparisons of processors from different ranges, we blocked the processor frequency at 2.1 Ghz, which is the highest frequency common to the 6 processors studied. In the same way, as some processors have less cores than others, we used the maximum number of cores common to the processors which is 8 cores per processor.

A. Run the application only once

We developed the Kernel Generator to help us in our search for efficient architectures for certain applications. To speed up this research, we wanted to drastically reduce the number of benchmarks to be executed as they can last several hours before their results are released. To do this, we will only run Fluent once, with Ansys public use-case, *sedan4m* [30] and monitor it with `mygflops` to get its computational profile. Listing 17 shows the profile collected.

```
++ Single-p: 10.298097 GFlop/s -- 97.2%
91.8%   scalar  32-bit inst (94.9% )
8.2%    packed 128-bit inst (  2.1% )
0.0%    packed 256-bit inst (  0.0% )
0.0%    packed 512-bit inst (  0.0% )

++ Double-p: 0.292344  GFlop/s -- 2.8%
100.0%  scalar  64-bit inst (  2.9% )
0.0%    packed 128-bit inst (  0.0% )
0.0%    packed 256-bit inst (  0.0% )
0.0%    packed 512-bit inst (  0.0% )
```

Listing 17: Monitoring of Fluent with mygflop using sedan4m case

If the profile is simple (only one type of instructions), then it can refer to the table we have done previously (see Figure 3). Then one would be able to estimate the performance of the code on processors which have already been analyzed with the Kernel Generator but also predict the performance of a processor not available.

B. Kernel Generator

If the profile of the application is made of a mix of instructions type or with different precisions, one will have to generate a micro-benchmark using the Kernel Generator which is equivalent, in terms of calculation operations, to the profile previously gathered. By using the different parameters available, the user will be able to generate a kernel that fits the original application (precision, width, dependency). In our case, we have generated a kernel composed of single precision instructions (scalar, and 128-bit wide) and double precision instructions (scalar). Listing 18 shows the profile of the kernel measured on the same processor (Xeon Platinium 8150).

```
++ Single-p: 49.5707 GFlop/s -- 97.1%
91.2%   scalar  32-bit inst (94.7% )
8.8%    packed 128-bit inst (  2.3% )
0.0%    packed 256-bit inst (  0.0% )
0.0%    packed 512-bit inst (  0.0% )

++ Double-p: 1.45494 GFlop/s --  2.9%
100.0%  scalar  64-bit inst (  3.0% )
0.0%    packed 128-bit inst (  0.0% )
0.0%    packed 256-bit inst (  0.0% )
0.0%    packed 512-bit inst (  0.0% )
```

Listing 18: Generating a kernel with the same profile as fluent's use case

Since the code is artificial and does not make any memory access, it is much more powerful than the fluent code (51 GFlop/s for the kernel counter 10.6 for Fluent). As the purpose of this study is to compare different processor's core we will only focus our measure on the FPU efficiency and what is important in this evaluation is that the proportion of the different instructions be respected.

TABLE V: Benchmark of Skylake processors with a kernel behaving like Fluent

	4110	5117	6148	6150	6130	8160
GFLOP/s (SP)	2.09	2.06	2.01	2.02	2.09	2.04
GFLOP/s (DP)	68.31	68.29	68.35	68.71	68.15	68.79
IPC	2	2	2	2	2	2

TABLE VI: Benchmark of Skylake processors on processors selected with the Kernel Generator

	Xeon 6150 (ref)	Xeon 4110	Xeon 5117
Total Solve Time (s)	57.32	58.20	57.43
GFLOP/s	10.63	10.54	10.65

C. Micro-benchmark every processors

As the micro-benchmark does not last long, it is now really fast to benchmark different processors and one will be able to execute it on all the potential processors. Table V shows the results of the kernel's execution on processors at our disposition. From these results we can conclude that fluent should have the same performance on all processors and even on the entry-level ones as the 4110 assuming that the stalls of the two processors will be equivalent.

D. Run the application on selected processors

By using the Kernel Generator as a filter, a user can accelerate his study of the multitude of available processors. Finally, to confirm those results, he would only have to execute the application on the processors selected. If for example we had, for economic reasons, a preference for the 4110 and 5117 we would only have to execute the real application on these two processors. Table VI shows the results of those execution that correspond to the previsions we made with the Kernel Generator.

V. Conclusion and future work

Building and planning for an HPC cluster has become a very interesting but in the same time a very difficult exercise due the large amount of choices available at the different levels of components. Benchmarking and properly understanding the performance delivered by the different processors, powering those clusters implies large efforts and a lot of time. In most cases, due to the lack of time not all the viable options are properly investigated and some very good possibilities may be easily missed. To tackle that problem, we have introduced in this paper a micro-benchmark candidate that allows its users to quickly understand the capabilities of the chosen processors. We know very well that a real HPC applications performance is dependent not only on the processor but also on the memory bandwidth, interconnect and so on. In this context, we state that our tool can be used as an initial assessment exercise by simulating the correct

instructions kernel which the real applications are executing. In doing so, we could start the application benchmarking on a solid list of processors candidates that our tool recommends.

Future work. Until now we have targeted our tool to applications which are mainly used in manufacturing like Computation Fluid Dynamics or Structures. We are planning to investigate other types of applications from different industry and vertical and research. The Kernel Generator has only been used for Intel processors, but once AMD and ARM64 processors are available in our lab, we would extend the use to include those new platforms as well. We would also expand the tool to allow users to dynamically decide on the mix and type of the instructions generated eventually via a graphical interface. Finally, the dependency between instructions can be more complex than the currently implemented configuration. Future developments could a allow the user to more accurately describe the dependencies.

REFERENCES

[1] Lisa Spelman, "The Intel Xeon Scalable a Truly Big Day for the Data Center," 2017. [Online]. Available: https://newsroom.intel.com/editorials/intel-xeon-scalable-processor-family-data-center/#gs.dmvxgu

[2] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-m. Hwu, "GPU clusters for high-performance computing," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–8.

[3] A. Sodani, "Knights landing (KNL): 2nd Generation Intel Xeon Phi processor," in *2015 IEEE Hot Chips 27 Symposium, HCS 2015*. IEEE, 2015, pp. 1–24.

[4] C. H. Martin, V. Tom, G. Yongfeng, S. Bharat, C. Al, M. Josh, and D. Doug, "Achieving High Performance with FPGA-Based Computing," *Computer*, vol. 40, no. 3, pp. 50–57, 2007.

[5] J. J. Dongarra, P. Luszczek, and A. Petite, "The LINPACK benchmark: Past, present and future," *Concurrency Computation Practice and Experience*, vol. 15, no. 9, pp. 803–820, 8 2003. [Online]. Available: http://doi.wiley.com/10.1002/cpe.728

[6] J. D. McCalpin, "STREAM benchmark," *Link: www. cs. virginia. edu/stream/ref. html# what*, vol. 22, 1995.

[7] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi, "Introduction to the HPC challenge benchmark suite," *Challenge*, no. December, p. 13, 2005. [Online]. Available: http://escholarship.org/uc/item/6sv079jp.pdf

[8] J. Dongarra, M. A. Heroux, and P. Luszczek, "A new metric for ranking high-performance computing systems," *National Science Review*, vol. 3, no. 1, pp. 30–35, 2016.

[9] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. a. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 1–12, 2002. [Online]. Available: http://download.intel.com/technology/itj/2002/volume06issue01/art01_hyper/vol6iss1_art01.pdf

[10] T. Jain and T. Agrawal, "The haswell microarchitecture-4th generation processor," *International Journal of Computer Science and Information Technologies*, vol. 4, no. 3, pp. 477–480, 2013.

[11] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the Intel®Core i7 Turbo Boost feature," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 188–197.

[12] C. Staelin and H.-P. Israel, "Lmbench3: Measuring Scalability," 2002.

[13] E. Bennett, L. Del Debbio, K. Jordan, B. Lucini, A. Patella, C. Pica, and A. Rago, "BSMBench: a flexible and scalable supercomputer benchmark from computational particle physics," 2014. [Online]. Available: http://arxiv.org/abs/1401.3733%0Ahttp://dx.doi.org/10.1109/HPCSim.2016.7568421

[14] J. J. Dongarra, H. W. Meuer, E. Strohmaier, and others, "TOP500 supercomputer sites," *Supercomputer*, vol. 13, pp. 89–111, 1997.

[15] Y. Deng, P. Zhang, C. Marques, R. Powell, and L. Zhang, "Analysis of Linpack and power efficiencies of the world's TOP500 supercomputers," *Parallel Computing*, vol. 39, no. 6-7, pp. 271–279, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2013.04.007

[16] A. Younes, O. Basir, and P. Calamai, "A benchmark generator for dynamic optimization," in *Proceedings of the 3rd International Conference on Soft Computing, Optimization, Simulation & Manufacturing Systems*, 2003.

[17] R. H. Saavedra and A. J. Smith, "Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes," *IEEE Transactions on Computers*, vol. 44, no. 10, pp. 1223–1235, 1995. [Online]. Available: http://ieeexplore.ieee.org/document/467697/

[18] J. González-Domínguez, G. L. Taboada, B. B. Fraguela, M. J. Martín, and J. Touriño, "Servet: A benchmark suite for autotuning on multicore clusters," in *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*. IEEE, 2010, pp. 1–9.

[19] R. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," in *Proceedings of the IEEE/ACM SC98 Conference*. IEEE, 1998, pp. 38–38. [Online]. Available: http://ieeexplore.ieee.org/document/1437325/

[20] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 3. IEEE, 1998, pp. 1381–1384.

[21] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1, pp. 181–192, 2005.

[22] J. Pourroy, "Performance Modelisation," in *GitHub repository*, 2019. [Online]. Available: https://github.com/PourroyJean/performance_modelisation

[23] I. Coorporation, "Using the rdtsc instruction for performance monitoring," *Techn. Ber., tech. rep., Intel Coorporation*, p. 22, 1997.

[24] "Using the GNU Compiler Collection (GCC): Extended Asm," 2019. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

[25] M. Milenkovic, A. Milenkovic, and J. Kulick, "Demystifying Intel branch predictors," *Proceedings of the 2002 Workshop on Duplicating, Deconstructing and Debunking (WDDD'02)*, no. Figure 1, 2002.

[26] P. Guide, "Intel®64 and ia-32 architectures software developers manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.

[27] Intel, "Intel Xeon Silver 4110 Processor (11M Cache, 2.10 GHz) Product Specifications." [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/123547/intel-xeon-silver-4110-processor-11m-cache-2-10-ghz.html

[28] WikiChip, "Skylake (client) - Microarchitectures - Intel - WikiChip." [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)#Execution_engine_2

[29] Ansys, "ANSYS High Performance Computing: Compute Clusters." [Online]. Available: https://www.ansys.com/products/platform/ansys-high-performance-computing

[30] ——, "External Flow Over a Passenger Sedan." [Online]. Available: https://www.ansys.com/solutions/solutions-by-role/it-professionals/platform-support/benchmarks-overview/ansys-fluent-benchmarks/ansys-fluent-benchmarks-release-16/external-flow-over-a-passenger-sedan