**Introduction into AVX instruction**

1. **introduction**

Intel Pentium III where the first processors to introduce SSE (Streaming SIMD extensions) which extended intel processors with specialized Single Instruction Multiple Data (SIMD) instructions, which allowed 128 bit vector operation to be perform simultaneously thus achieving much higher thorough in workloads such as digital signal processing and vector graphics.

SSE would get multiple revision to enable additional functionality like double operations, till SSE4.

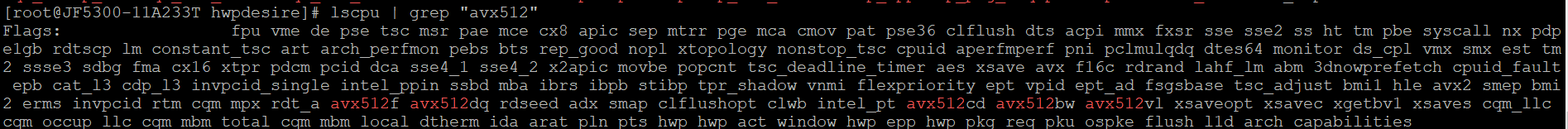
AVX was released in 2011 with sandy bridge (2nd gen core) to support 256 wide bit operations.

AVX2 is an additional expansion of AVX in the 4th gen core processor adding additional functionality for AVX while keeping the 256 registers and operations

AVX3 is 512 bit wide extension that was first seen on intel Xeon Phi (knight landing), and later on with skylake-x CPUs.

To check whether AVX 512 is enabled on your machine we have to can use:

$lscpu | grep "avx512"



To zoon in further



Hence our machine supports the following avx 512 subset of instructions:

**avx512f** **avx512dq** **avx512cd** **avx512bw** **avx512vl**

VX-512 Foundation (F) – expands most 32-bit and 64-bit based AVX instructions with the EVEX coding scheme to support 512-bit registers, operation masks, parameter broadcasting, and embedded rounding and exception control, supported by Knights Landing and Skylake Xeon

AVX-512 Conflict Detection Instructions (CD) – efficient conflict detection to allow more loops to be vectorized, supported by Knights Landing and Skylake X

<https://software.intel.com/en-us/blogs/2013/avx-512-instructions>

AVX-512 Vector Length Extensions (VL) – extends most AVX-512 operations to also operate on XMM (128-bit) and YMM (256-bit) registers

VX-512 Doubleword and Quadword Instructions (DQ) – adds new 32-bit and 64-bit AVX-512 instructions

AVX-512 Byte and Word Instructions (BW) – extends AVX-512 to cover 8-bit and 16-bit integer operations

<https://software.intel.com/en-us/blogs/additional-avx-512-instructions>

We can use the following look to see the intrinsic instructions and function that our processer can use:

<https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

In order to execute a single AVX instruction, the CPU must use all 8 ports in the backend of the processor.

By using AVX512 we can essentially do 8 double calculations at once or 16 floats or int operations at the same time per CPU.

However since we are using all ports at the same time we are stressing the CPU to the maximum of its potential and we expect a spike in power usage thus producing more heat, and can cause thermal of power throttling.

In order to compile avx instruction we need a complier which can compile such code

* [GCC](https://en.wikipedia.org/wiki/GNU_Compiler_Collection) 4.9 and newer
* [Clang](https://en.wikipedia.org/wiki/Clang) 3.9 and newer
* [ICC](https://en.wikipedia.org/wiki/Intel_C%2B%2B_Compiler) 15.0.1 and newer
* Microsoft Visual Studio 2017 C++ Compiler
* Java 9

Also each compiler would need a special compiler flag to allow it to use 512 AVX instructions

1. **Testing methodology**

In order to test the performance and power of using AVX instructions we wrote three different scripts. They all do a dot product of two 83040 long vectors.

One is using standard code, one using avx 2 code and one using avx 512 code the code are implemented the following:

#include <immintrin.h> // this include is needed to use intrinsic functions



Using avx 512 instructions



Using avx 256 instructions:



All three version of the code will produce the same output given the same input.

Most complier can automatically vectorize SSE instructions but they need to enable complier flags to compile avx 2 and avx 512.

In this test we used openMP to fork run 112 threads (test machine is dual CPU each with 56 cores and 112 threads), running the same dot product millions of time, divided by the number of threads.

The following is the main function which reads the request number of threads and attempts to fork them. We also measure the start and end time of the program. Note that openmp does implicit synchronization of thread after the pragma block, so thread\_join or barrier are not required.

The main function along with the required libraries are written below in the code excerpt



To compile the code we can use the following pseudo-code commands:



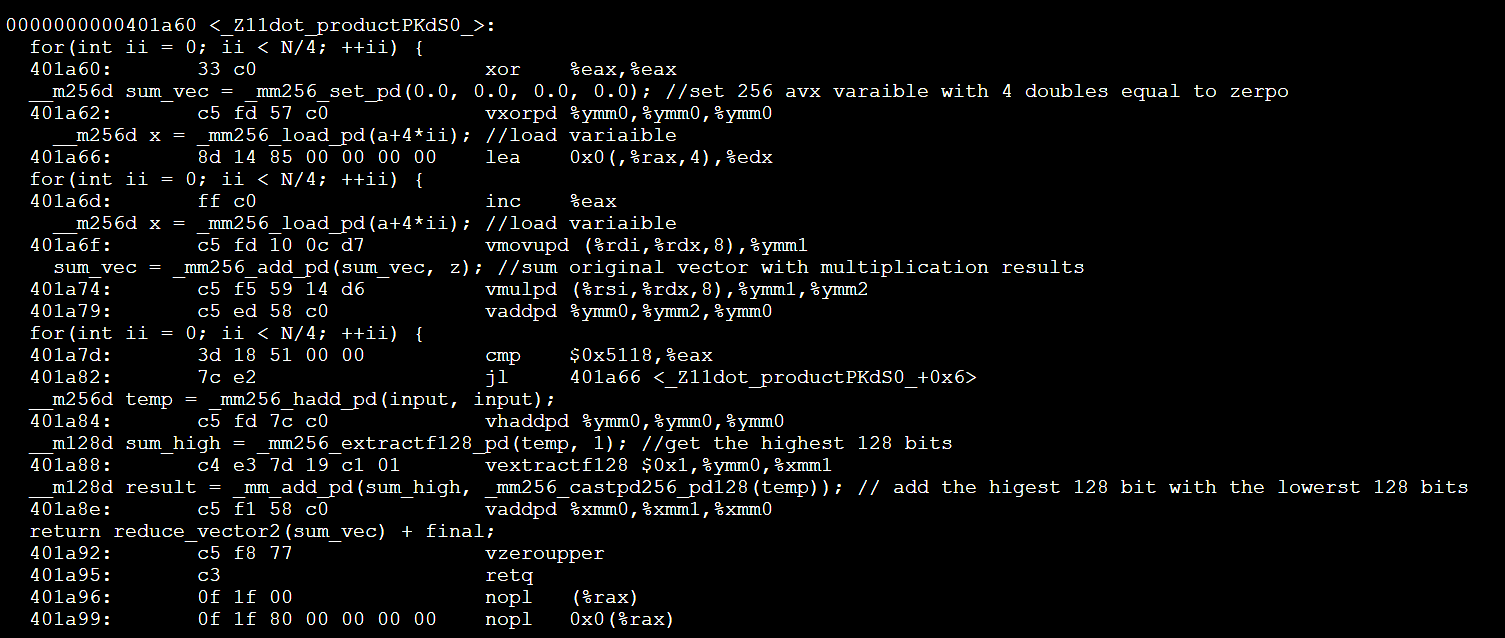
Notice –axAVX2 to compile and run AVX2 code.

1. **Testing results**

To verify that AVX2 is compiled correctly we can use objdump

$objdump –S –W cpu\_stress\_avx2

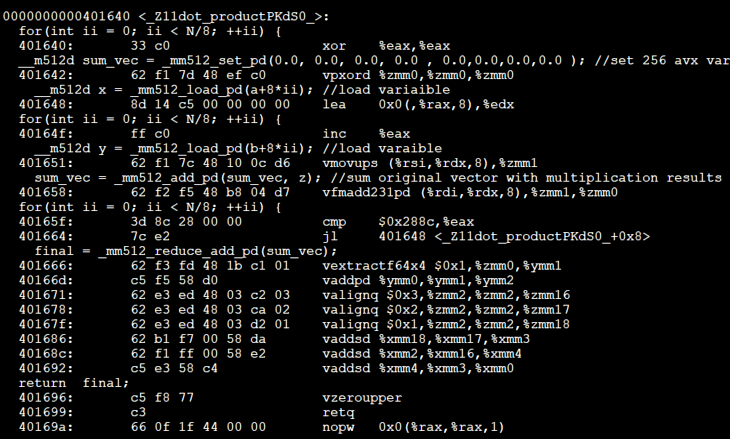
If scroll where our dot\_product avx2 is located we can see the assembly instructions used to verify that our AVX2 compiled correctly



The of $ymm0 register and vaddpd instructions verify that our AVX2 did compile with the desired instruncitons

The same for avx512 instructions

$objdump –S –W 512\_cpu\_stress\_avx



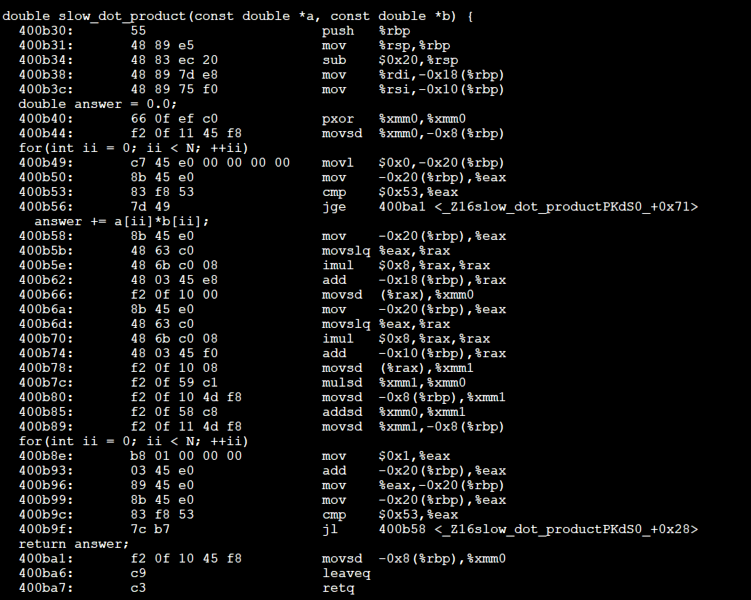
The usage of vmovups and zmm1 register verify the usage of 512-avx instructions.

When compiled without the icpc was using the SSE 128 bit instructions. A 128 bit run took around 180 seconds, which is significantly slower than the avx2 or avx3.

Compiling the slow\_dot\_product function with –axAVX2 or –xCOMMON-AVX512 with intel icpc compiler will automatically vectorize the loop to avx2 or 3 respectively without the manually using intrinsic functions

In our test runs we saw the avx2 workload managed to maintain the base clock (2.7 GHz) on all thread when running the avx2 dot product with 28 seconds, however in AVX 3 the frequency dropped to below 2.5 GHz when running on all cores, however the benchmark finished in ~22 seconds.

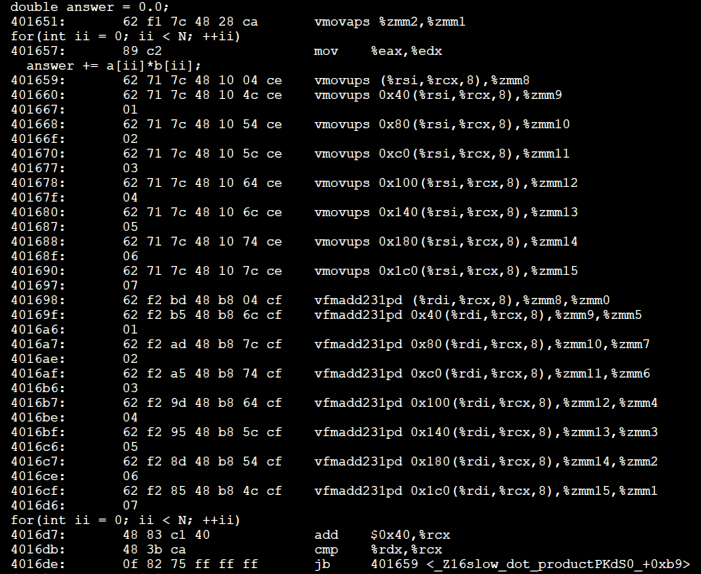
Emon data also shows same package power of both instances at ~405 watts indicating a power throttling for the AVX3 workload.



The following results from objdump from a gcc compiled code, we can notice the usage mulsd and xmm0 register both indicating a 128 bit instructions.

Modern compilers can automatically vectorize some loops if the conditions are right and we provided the correct compiler argument.

The example below is an excerpt from the slow\_dot\_product function but compiled with –xCOMMON-AVX512, the compiler automatically unrolled some loops and ran the multiplication and addition similar to that of our intrinsic function. The automatically vectorized code performed on par with manual intrinsic functions in both power consumption and speed and frequency.



1. **Conclusion**

These runs shows that AVX instruction can give performance benefits at the cost of power consumption. But it more energy efficient since it can finish the same workload earlier while using the same amount of power at least in our optimizard benchmark.