Assignment 1: Maximum CPU rates - ITCS 4182

Meghana Gudaram
ID: 800962452

Machine used for this experiment ***: Intel(R) Xeon(R) CPU E5-2667 v3 @ 3.20GHz***

```
processor       : 0 // there are 2 processors with 8 cores each
vendor_id       : GenuineIntel // Intel processors, Haswell arch
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2667 v3 @ 3.20GHz
stepping        : 2
microcode       : 0x36
cpu MHz         : 3321.375 // CPU clk is approximately 3.2 GHz
cache size      : 20480 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 8  // 8 Cores on each processor
```

Intel Haswell, Intel Broadwell and Intel Skylake	16 DP FLOPs/cycle: two 4-wide FMA instructions	32 SP FLOPs/cycle: two 8-wide FMA instructions ➔ **S = 32**

Maximum number of floating point operations this machine can perform per second
Socket * Core * Frequency * FPA * OP / S
*= 2 * 8 * 3.2 * 10^9 * 2 * 2 * 256 / 32 =* ***1638 GFlops/cycle or 1.638 TFlops/cycle***

Maximum number of integer operations this machine can perform per second
Socket * Core * Frequency * FPA * OP / S
*= 2 * 8 * 3.2 * 10^9 * 3 * 1 * 256 / 32 =* ***1228 GIops/cycle or 1.228 TFlops/cycle***

A code that gets peak Flops

```
__m256 fma()
{
   __m256 x = _mm256_setr_ps (1.2, 2.2, 3.2, 4.2, 5.2, 6.2, 7.2, 8.2);
   __m256 y = _mm256_setr_ps (10.1, 20.1, 30.1, 40.1, 50.1, 60.1, 70.1, 80.1);
   __m256 z = _mm256_setr_ps (1.1, 2.1, 3.1, 4.1, 5.1, 6.1, 7.1, 8.1);
  for (long long i=0; i< 10000000; ++i)
   {
     z= _mm256_fmadd_ps(x, y, z); // this intrinsic function fuses fma
   }
  return z;
 } // complete code at https://github.com/MeghanaGudaram/HighPerformanceComputing
```

A code that gets peak Iops

```
__m256i iop()
{
   __m256i x = _mm256_set_epi16 (1, 2, 3, 4, 5, 6, 7, 8, 9,10, 11, 12, 13,14,15,16);
   __m256i y = _mm256_set_epi16 (1, 2, 3, 4, 5, 6, 7, 8, 9,10, 11, 12, 13,14,15,16);
   __m256i z ;
  for (long long i=0; i< 10000000; ++i)
   {
    z= _mm256_adds_epu16(x, y); // this intrinsic function fuses avx2
   }
  return z;
 } // complete code at https://github.com/MeghanaGudaram/HighPerformanceComputing
```

Assembly code to make sure if fma and avx are fused along with for loop :

```
.LFB6238:
        .cfi_startproc
        movl    $10000000, %eax          // for loop is not optimized
        vmovaps .LC0(%rip), %ymm0
        vmovaps .LC1(%rip), %ymm2
        vmovaps .LC2(%rip), %ymm1
```

	Double Precision Packed FP	Single Precision Packed FP	Double Precision Scalar FP	Single Precision Scalar FP
Fused Multiply-Add A = A x B + C C += A x B	VFMADD132PD VFMADD213PD VFMADD231PD _mm_fmadd_pd() _mm256_fmadd_pd()	VFMADD132PS VFMADD213PS VFMADD231PS _mm_fmadd_ps() _mm256_fmadd_ps()	VFMADD132SD VFMADD213SD VFMADD231SD _mm_fmadd_sd() _mm256_fmadd_sd()	VFMADD132SS, FMADD213SS VFMADD231SS _mm_fmadd_ss() _mm256_fmadd_ss()

```
.L2:
        vfmadd231ps     %ymm1, %ymm2, %ymm0  // vfmadd231ps make sure fma is fused
        subq    $1, %rax
        jne     .L2
        rep ret
```

Similarly for iops :

```
.L15:

        .cfi_startproc
        movl $10000000, %eax
        vmovdqa .LC0(%rip), %ymm1
        vpaddusw %ymm1, %ymm1,%ymm1 // vaddsubps make sure avx fused
```

Flops code achieved :

```
[mgudaram@mba-i1 ~]$ g++ -std=c++11 -mavx2 -mfma flop.cpp
[mgudaram@mba-i1 ~]$ ./a.out
Time taken for 10^7 flops : 59604.9 useconds.
[mgudaram@mba-i1 ~]$ g++ -std=c++11 -mavx2 -mfma -O flop.cpp
[mgudaram@mba-i1 ~]$ ./a.out
Time taken for 10^7 flops : 0.115 useconds.
[mgudaram@mba-i1 ~]$ g++ -std=c++11 -mavx2 -mfma -O1 flop.cpp
[mgudaram@mba-i1 ~]$ ./a.out
Time taken for 10^7 flops : 0.094 useconds.
[mgudaram@mba-i1 ~]$ g++ -std=c++11 -mavx2 -mfma -O2 flop.cpp
[mgudaram@mba-i1 ~]$ ./a.out
Time taken for 10^7 flops : 0.094 useconds.
```

From the figure time taken before optimization:
10^7 Flops = 5.9 * 10^{-2} ➔ 1 Flop = 5.9 * 10^{-9} sec, frequency is 0.16 GFlops

From the figure time taken after optimization:
10^7 Flops = 1.15 * 10^{-5} ➔ 1 Flop = 1.15 * 12 sec, frequency is 868 GFlops or 0.86 TFlops

Iops code achieved :

```
[mgudaram@mba-i1 ~]$ g++ -std=c++11 -mavx2 -mfma Iop.cpp
[mgudaram@mba-i1 ~]$ ./a.out
Time taken for 10^7 operations : 22656.2 useconds.
[mgudaram@mba-i1 ~]$ g++ -std=c++11 -mavx2 -mfma -O Iop.cpp
[mgudaram@mba-i1 ~]$ ./a.out
Time taken for 10^7 operations : 0.104 useconds.
[mgudaram@mba-i1 ~]$ g++ -std=c++11 -mavx2 -mfma -O1 Iop.cpp
[mgudaram@mba-i1 ~]$ ./a.out
Time taken for 10^7 operations : 0.093 useconds.
[mgudaram@mba-i1 ~]$ g++ -std=c++11 -mavx2 -mfma -O2 Iop.cpp
[mgudaram@mba-i1 ~]$ ./a.out
Time taken for 10^7 operations : 0.096 useconds.
```

From the figure time taken before optimization:
10^7 Iops = 2.2 * 10^{-2} ➔ 1 Iops = 2.2 * 10^{-9} sec, frequency is 0.4 GIops

From the figure time taken after optimization:
10^7 Iops = 1.04 * 10^{-5} ➔ 1 Iops = 1.04 * 12 sec, frequency is 961 GIops or 0.96 TIops

Expectation Vs Results
FLOPs : 0.86/1.638 = 0.525 ➔ only 52.5% is utilized by the CPU
IOPSs : 0.96/1.228 = 0.781 ➔ only 78.1% is utilized by the CPU

Conclusion : Yes, code could be written better to utilize the CPU (there by utilize architecture to the fullest). Lower FLOP/s are often an indication of significant latencies and overall performance bottlenecks. Workloads with low mask register utilization may have over counted the FLOPs value. In order to vectorize traditional SIMD reduction loops, compilers often have to do special post-processing, which could be seen as expensive vectorization overhead.

References :
https://github.com/MeghanaGudaram/HighPerformanceComputing