GUIDE TO AUTOMATIC VECTORIZATION WITH INTEL AVX-512 INSTRUCTIONS IN KNIGHTS LANDING PROCESSORS

Bonan Zhang

Colfax International

May 11, 2016

Abstract

This publication is part of a developer guide focusing on the new features in 2nd generation Intel® Xeon Phi™ processors code-named Knights Landing (KNL). In this document, we focus on the new vector instruction set introduced in Knights Landing processors, Intel® Advanced Vector Extensions 512 (Intel® AVX-512). The discussion includes:

- Introduction to vector instructions in general,
- The structure and specifics of AVX-512, and
- Practical usage tips: checking if a processor has support for various features, compilation process and compiler arguments, and pros and cons of explicit and automatic vectorization using the Intel® C++ Compiler and the GNU Compiler Collection.

This publication and other white papers on KNL processors can be found on the Colfax Research Website: colfaxresearch.com/knl-guide

Table of Contents

1. VECTOR INSTRUCTIONS

Intel® Xeon Phi™ products are highly parallel processors with the Intel® Many Integrated Core (MIC) architecture. Parallelism is present in these processors at two levels: task parallelism and data parallelism. Task parallelism comes from the massive number of cores in the processor (up to 72). Data parallelism comes from support for vector instructions, which make every core a single instruction multiple data (SIMD) processor. 1st generation Intel Xeon Phi processors, released in 2012 and code-named Knights Corner (KNC), were first Intel architecture processors to support 512-bit vectors. 2nd generation Intel Xeon Phi processors, to be released in 2016 and code-named Knights Landing (KNL), also support 512-bit vectors, but in a new instruction set called Intel® Advanced Vector Extensions 512 (Intel® AVX-512).

The Knights Landing architecture has a peak theoretical performance of 6 TFLOP/s in single precision, which is triple than what Knights Corner had. This performance gain is partly due to the presence of two vector processing units (VPUs) per core, doubled compared to the previous generation. Each VPU operates independently on 512-bit vector registers, which can hold 16 single precision or 8 double precision floating-point numbers.

The need for using vector instructions cannot be understated, as it is the key to efficiently utilizing the VPUs for high performance.

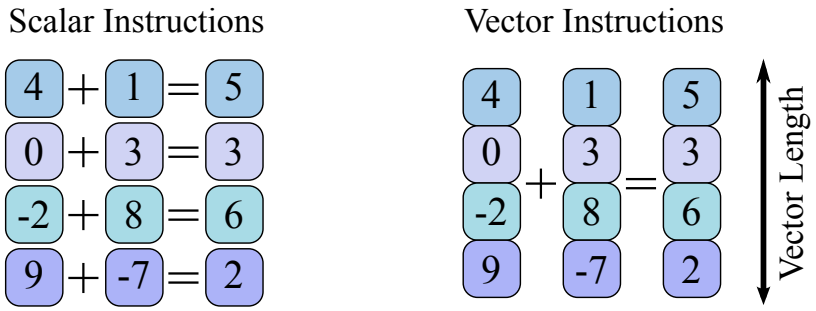

Figure 1: Illustration of SIMD instruction

Figure 1 shows an example of a vector instruction adding multiple numbers together at the same time, compared to doing the additions sequentially. Where a scalar processor would have to perform a load, an addition and a store instruction for every element, a vector processor performs one load, one addition and one store for multiple elements. AVX-512 wide vectors and 2 VPUs allow for 32 additions to happen at the same time for single precision numbers.

To use vector instructions in applications, developers have two options: (i) explicitly calling vector instructions from assembly or via instrinsic functions, or (ii) relying on automatic vectorization by the compiler. The former option allows fine control over the instruction stream, while the latter is portable and "future-proof". This means that to adapt code to a future generation of processors, only re-compilation is needed. At the same time, to effectively use automatic vectorization, the programmer must follow guidelines for vectorizable code (see, e.g., [1]) and be aware of the specifics of the instructions supported by the processor. This paper aims to provide necessary details about AVX-512 for developers following the automatic vectorization route.

2. STRUCTURE AND FUNCTIONALITY OF AVX-512

2.1. SUBSETS

The Intel AVX-512 has 9 subsets of instructions, of which some are supported by Intel Xeon Phi processors starting with Knights Landing, and some will be supported by future Intel® Xeon® processors [2]. They are listed and illustrated in Figure 2 below.

- Features common to Knights Landing and future Intel Xeon processors include AVX512-F and AVX512-CD. If it is desirable to maintain binary compatibility between the Knights Landing processor and a future Intel Xeon processor, these are the subsets to use.
- All AVX-512 features of the Knights Landing processors include, in addition to AVX512-F and AVX512-CD, the AVX512-ER, and AVX512-PF subsets. If it is not necessary to maintain binary compatibility between the Knights Landing processor and a future Intel Xeon processor, then the additional AVX512-ER and AVX512-PF instructions can be leveraged to vectorize previously unvectorizable code.
- All AVX-512 features of the future Intel Xeon processors include, in addition to AVX512-F and AVX512-CD, the AVX512-BW, AVX512-DQ, AVX512-VL, AVX512-IFMA, and AVX512-VBMI subsets. Since these instructions are not available on the Knights Landing processors, they are out of the scope of this document.

2.2. AVX512-F

This feature set is the "Fundamental" instruction set, available on Knights Landing processors and future Intel Xeon processors. It contains vectorized arithmetic operations, comparisons, type conversions, data movement, data permutation, bitwise logical operations on vectors and masks, and miscellaneous math functions like min/max [3]. This is similar to the core feature set of the AVX2 instruction set, with the difference of wider registers, and more double precision and integer support.

2.3. AVX512-CD

This feature set is the "Conflict Detection" instruction set, available on Knights Landing processors and future Intel Xeon processors. The following code sample illustrates when conflict detection instructions are useful.

```
1 void hist(const float * samples, int * const hist, const float group_width) {
2     float group_width_rec = 1.0f / group_width;
3 #pragma vector aligned
4     for (int sample_index = 0; sample_index < 1024; sample_index++) {
5         const int bin = (int) (samples[sample_index] * group_width_rec);
6         hist[bin]++;
7     }
8 }
```

Listing 1: Binning Code with Conflicting Memory Accesses

This code illustrates the computation of a histogram. This educational example represents a greater class of real-world problems that involve binning (for example, Monte Carlo simulations of particle transport). Operations in line 5 have SIMD semantics, and therefore the performance of this application can benefit from vectorization. However, the loop in line 4 cannot be automatically vectorized by the compiler because the final binning to the histogram array in line 6 has memory accesses that could have a conflict. The conflict arises from the value of the variable *bin*, which could be the same for different iterations of the loop. The compiler sees this as a vector dependence. This entire loop was not vectorizable with previous instruction sets, and code changes (strip-mining and loop splitting) was required to vectorize the calculation in line 5. However, with the addition of conflict detection instructions, like `vpconflictd`, which will detect conflicting memory accesses, it is now possible for the compiler to generate vectorized code to resolve the vector dependence [4]. For more on how to vectorize this with previous instructions sets, refer to this publication from Colfax Research [5].

2.4. AVX512-ER

This feature set is the "Exponential and Reciprocal" instruction set, available on Knights Landing processors. It contains instructions for base 2 exponential functions (i.e., 2^x), reciprocals, and inverse square root. These instructions are available in both single and double precision, with rounding and masking options. The max relative error is 2^{-23} for exponential and 2^{-28} for reciprocal and inverse square roots. Previously, the IMCI instruction set in the KNC architecture only had single precision support for these features. In even earlier instruction sets like AVX2, only the reciprocal is available; the vector exponential is computed via software implementations without hardware support.

2.5. AVX512-PF

This feature set is the "Prefetch" instruction set, available on Knights Landing processors. This contains prefetch instructions for gather and scatter instructions. Even though these instructions provide software prefetch support, Knights Landing processors have a much heavier emphasis on hardware prefetching. The reason for that is the instructions within this subset are all only hints to the processor [6]. These correspond to `#pragma prefetch` directives in software, which are also hints. For most cases, memory prefetching happens with reasonably good performance for accesses with good patterns.

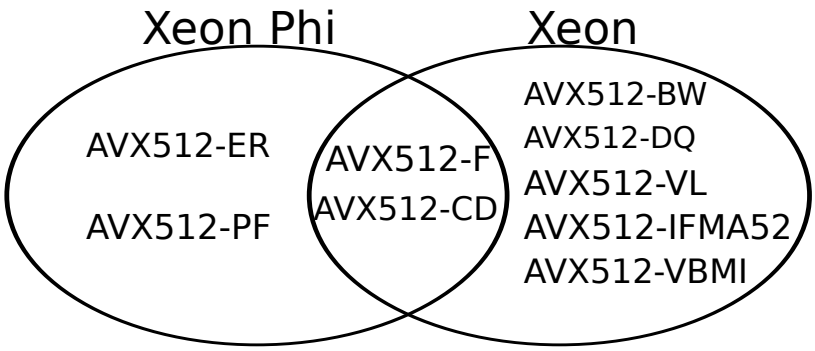

Figure 2: Instructions available on Intel Xeon and Intel Xeon Phi processors

3. FEATURE CHECK

There are two ways for determining if a processor has support for various features. You can determine feature support on a processor from the command line, or using library calls from source code.

3.1. COMMAND LINE

From the command line, you can run the command in Listing 2, and look for avx512f, avx512cd, etc., in the list of processor features. Note that you can also find flags for legacy features like avx or sse2.

```
vega@lyra% cat /proc/cpuinfo
...
flags                : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cm
ov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aper
fmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 fma cx16 xtp
r pdcm sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdr
and lahf_lm abm 3dnowprefetch ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi
flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms avx512f rdseed
adx avx512pf avx512er avx512cd
...
```

Listing 2: Processor Feature Support

3.2. SOURCE CODE

This information ultimately comes from the `CPUID` instruction, where the processor returns a string of information including the product family, model, and list of supported features [7]. The Intel C++ Compiler has a built-in wrapper, which is the `_may_i_use_cpu_feature` intrinsic. Example usage for checking the Knights Landing features is shown in Listing 3. For other compilers, the code is less abstracted because it is hardware dependent. For more details, refer to this blog post for implementation details [8].

```
1 #include <immintrin.h>
2 int has_intel_knl_features() {
3     const unsigned long knl_features =
4         (_FEATURE_AVX512F | _FEATURE_AVX512ER |
5         _FEATURE_AVX512PF | _FEATURE_AVX512CD );
6         return _may_i_use_cpu_feature( knl_features );
7 }
```

Listing 3: Processor feature support intrinsic

4. COMPILING

4.1. USAGE MODELS

There are two ways to take advantage of vector instructions: explicit vectorization and automatic vectorization by the compiler. Using explicit vectorization gives the developer control over every instruction, but comes at the cost of portability. Using automatic vectorization means letting the compiler look for patterns in the code to determine what vector instructions to use. In the following example, two arrays of eight double precision floating-point numbers are added together.

```
1 double A[vec_width], B[vec_width];
2 //this loop will be automatically vectorized
3 for(int i = 0; i < vec_width; i++)
4   A[i]+=B[i];
```

Listing 4: Automatic vectorization

```
1 double A[vec_width], B[vec_width];
2 __m512d A_vec = _mm512_load_pd(A);
3 __m512d B_vec = _mm512_load_pd(B);
4 A_vec = _mm512_add_pd(A_vec,B_vec);
5 _mm512_store_pd(A,A_vec);
```

Listing 5: Explicit vectorization

A good practice is to write code that the compiler can vectorize, to maintain clarity and simplicity. This publication will mainly discuss automatic vectorization, and compiler behavior with the Intel C++ Compiler and the GNU Compiler Collection's C++ Compiler. The example code in Listing 4 and Listing 5 is compiled with:

```
vega@lyra% icpc -xCOMMON-AVX512 -S -o vec_add.s vec_add.cc
```

Listing 6: Compiling with source for assembly code

The `-S` option makes the compiler produce intermediate assembly files so we can check for vectorized instructions. The assembly code produced by the Intel C++ Compiler will have line numbers in the original source for reference, snippet shown in Listing 7. Vectorized instructions begin with the letter v. They operate on vector registers named zmm (or xmm, and ymm for legacy AVX, and AVX2 instructions, respectively).

```
vega@lyra% cat vec_add.s
        vldmxcsr  64(%rsp)                                      #1.12
        vaddpd    (%rsp), %zmm0, %zmm1                          #5.9
        vmovupd   %zmm1, (%rsp)                                 #5.9
```

Listing 7: A few lines of the resulting assembly

You can compile for the Knights Landing processor with any compiler that supports the AVX-512 instruction set. The following sections will show how to compile for the Knights Landing processor with the Intel C++ Compiler and the GNU Compiler Collection, two current implementations of such compilers. The compiler binaries are called icpc, and g++, respectively. With Intel C++ compiler, the minimum version required for these flags is 15.0. In GCC, the flags for the Knights Landing processor are supported since version 4.9.1 and on. The flags for the future Intel Xeon processors are supported since GCC version 5.1 and on. Required compilation flags are detailed below, and summarized in Table 1.

4.2. Intel C++ Compiler

Before you begin, you may wish to check the version of the compiler you have, you can do this as follows:

```
vega@lyra% icpc -v
icpc version 16.0.1 (gcc version 4.9.2 compatibility)
```

Listing 8: Version check

You can see your code in assembly by using the `-S` option. The assembly code is useful for checking if your code was vectorized. As shown in Listing 6, the compiler produces a foo.s assembly file for the input file foo.cc.

Of course, reading through long assembly code is not feasible for any non-trivial amount of code. The Intel C++ Compiler provides a helpful utility, the optimization report. The `-qopt-report=5` option tells the compiler to produce a verbose optimization report. The verbosity level is a number between 1 and 5, 5 being the most verbose.

```
vega@lyra% icpc -xCOMMON-AVX512 -qopt-report=5 -o vec_add vec_add.cc
```

Listing 9: Compiling for auto-vectorization report

Listing 10 shows part of the optimization report for this compilation. The report is organized into sections, and this one shows the loop optimization done for this particular loop.

```
LOOP BEGIN at vec_add.cc(5,5)
   remark #15305: vectorization support: vector length 8
   remark #15427: loop was completely unrolled
   remark #15300: LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 2
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 8
   remark #15477: vector loop cost: 0.620
   remark #15478: estimated potential speedup: 12.800
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

Listing 10: icpc auto-vectorization report

As described in Section 2.1, there are multiple subsets belonging to AVX-512. The following `-x` options correspond to the three pieces of Figure 2, left to right.

```
vega@lyra% # this compiles for Intel Xeon Phi processors (KNL)
vega@lyra% icpc foo.cc -xMIC-AVX512
vega@lyra%
vega@lyra% # this compiles for binary compatibility
vega@lyra% icpc foo.cc -xCOMMON-AVX512
vega@lyra%
vega@lyra% # this compiles for future Intel Xeon processors
vega@lyra% icpc foo.cc -xCORE-AVX512
```

Listing 11: Compiling for with Intel C++ Compiler for various AVX-512 subsets

4.3. The GNU Compiler Collection

GCC also supports AVX-512, and the following is a list of common use cases with the C++ Compiler. Before you begin, you may wish to check the version of the compiler you have, you can do this as follows:

```
vega@lyra% g++ -v
gcc version 4.9.2 (GCC)
```

Listing 12: Version check

To produce an executable or the assembly code, the syntax is just as one might expect. One notable difference is that automatic vectorization is not turned on by default as it is with icpc. You need to specify a high enough optimization level with the `-O` option.

```
vega@lyra% g++ -O2 -o foo foo.cc
vega@lyra% g++ -O2 -o foo.s -S foo.cc
```

Listing 13: Compiling with source for executable or assembly code

GCC also has support for giving the user an optimization report. GCC is able to tell the user what it was able to and not able to vectorize and why with the option `-fopt-info-vec-all` [9].

```
vega@lyra% g++ -O2 -mavx512f -mavx512cd -fopt-info-vec-all -o foo foo.cc
```

Listing 14: Compiling for auto-vectorization report

Listing 15 shows the part of a GCC optimization report for a successfully vectorized loop.

```
vec_add.cc:5:22: note: Cost model analysis:
  Vector inside of loop cost: 4
  Vector prologue cost: 4
  Vector epilogue cost: 0
  Scalar iteration cost: 4
  Scalar outside cost: 1
  Vector outside cost: 4
  prologue iterations: 0
  epilogue iterations: 0
  Calculated minimum iters for profitability: 1
vec_add.cc:5:22: note:   Runtime profitability threshold = 7
vec_add.cc:5:22: note:   Static estimate profitability threshold = 7
vec_add.cc:5:22: note: loop vectorized
```

Listing 15: GCC auto-vectorization report

Unlike the Intel C++ Compiler, compiling for the various subsets require a more granular control. Instead of using the groups, you must specify the individual subsets.

```
vega@lyra% # this compiles for Intel Xeon Phi processors (KNL)
vega@lyra% g++ foo.cc -mavx512f -mavx512cd -mavx512er -mavx512pf
vega@lyra%
vega@lyra% # this compiles for future Intel Xeon processors
vega@lyra% g++ foo.cc -mavx512f -mavx512cd -mavx512bw \
> -mavx512dq -mavx512vl -mavx512ifma -mavx512vbmi
vega@lyra%
vega@lyra% # this compiles for binary compatibility
vega@lyra% g++ foo.cc -mavx512f -mavx512cd
```

Listing 16: Compiling for with GCC for various AVX-512 subsets

Table 1 summarizes the compiler arguments necessary for automatic vectorization with AVX-512 with the Intel C++ compiler and with GCC.

	Intel Compilers	GCC
Cross-platform	-xCOMMON-AVX512	-mavx512f -mavx512cd
KNL processors	-xMIC-AVX512	-mavx512f -mavx512cd -mavx512er -mavx512pf
Intel Xeon processors	-xCORE-AVX512	-mavx512f -mavx512cd -mavx512bw -mavx512dq -mavx512vl -mavx512ifma -mavx512vbmi

Table 1: summary of compilation flags

REFERENCES

[1] Colfax Hands On Workshop (HOW) series.
http://colfaxresearch.com/how-series.

[2] AVX-512 instructions.
https://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[3] Intel Intrinsic Guide.
https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[4] GNU Support for AVX 512.
https://gcc.gnu.org/wiki/cauldron2014?action=AttachFile&do=get&target=Cauldron14_AVX-512_Vector_ISA_Kirill_Yukhin_20140711.pdf.

[5] Andrey Vladimirov. Optimization Techniques for the Intel MIC Architecture. Part 2 of 3: Strip-Mining for Vectorization , 2015.
http://colfaxresearch.com/optimization-techniques-for-the-intel-mic-architecture-part-2-of-3-strip-mining-for-vectorization.

[6] Intel Architecture Instruction Set Extensions Programming Reference.
https://software.intel.com/en-us/intel-architecture-instruction-set-extensions-programming-reference.

[7] Intel® 64 and IA-32 Architectures Software Developer Manuals.
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[8] How to detect Knights Landing AVX-512 support (Intel Xeon Phi processor).
https://software.intel.com/en-us/articles/how-to-detect-knl-instruction-support.

[9] GNU Compiler Collection Optimzation Report.
https://gcc.gnu.org/onlinedocs/gccint/Dump-examples.html.