Improving performance through vector optimizations with z/OS XL C/C++

Improving performance through vector optimizations with z/OS XL C/C++

Introduction

In the past, a large portion of the improvements from one release of the IBM Z® hardware to the next came from CPU clock speed increases, which allowed existing programs to bene it from the newer hardware without the need for recompilation. Simply running the existing binary on the new hardware was enough to see noticeable improvement in performance. However, as CPU clock speed has stagnated due to physical limitations, CPU designers have focused their efforts on other areas within the CPU to improve performance. These areas include improved branch prediction, increased cache sizes, and additions to the instruction set architecture (ISA).

One of the biggest additions to the ISA introduced in IBM z13® and IBM z13s® and enhanced in IBM z14™ were the vector instructions and vector registers. Vector instructions use the vector registers to operate on multiple elements of data in parallel. These single instruction, multiple data (SIMD) class of instructions allow the processor to use the data level parallelism inherent in many workloads to improve performance. However, unlike the CPU clock speed improvements of the past, these new SIMD instructions require greater intervention on the part of the user. Source code needs to be recompiled with the latest compiler to properly take advantage of these new features. Not recompiling means missing out on possibly significant performance improvements and under-utilizing the hardware.

This document describes the vector-related optimizations that are available in the z/OS® V2R3 XL C/C++ compiler. Most of these optimizations allow you to take full advantage of the latest hardware with little to no source changes. Simply recompiling with the right set of options is often enough to use the new vector instructions.

Note: While the goal of compiler optimizations is to improve the performance of compiled code, there is no guarantee that performance improvement will always be the case. The benefit and size of impact depends on various factors, including the code to be compiled and the kinds of operations the code uses.

How this document is organized

This document describes the following features in detail:

Hardware vector facilities
: Provides significant performance improvements over the older scalar instructions.

Vector programming support
: Allows the user to use vector instructions directly by using built-in functions.

Automatic compiler optimizations

Vector optimizations require recompilation, but not source code changes.

AutoSIMD
: Analyzes and transforms the source code to take advantage of SIMD instructions where possible.

Vector single element
: Uses vector instructions and registers to perform scalar IEEE-754 binary floating-point operations.

Vector long double
: Uses vector registers and instructions for IEEE-754 quadruple precision operations.

Vector packed decimal
: Uses vector registers and instructions for binary coded decimal (BCD) operations.

Vectorized C string functions
: Uses vector string instructions to implement common C string functions.

Hardware vector facilities

Vector instructions were introduced in the vector facility for z/Architecture® in the z13 hardware. This facility added 32 128-bit wide vector registers and a set of corresponding vector instructions that uses these registers. Depending on the instruction, each 128-bit wide vector register can be treated as having 1, 2, 4, 8 or 16 element(s). The instructions perform an operation on all elements of the vector register in parallel. Most of the vector floating-point instructions can also perform their operation on a single element, which is discussed in details in Vector single element.

In z14, the z/Architecture *vector enhancements facility 1* and *vector packed decimal facility* were added, which build upon the existing z13 vector facility by adding new vector instructions. Among other things, the vector enhancements facility 1 adds support for IEEE-754 single and quadruple precision binary floating-point arithmetic. Previously vector instructions were limited to IEEE-754 double precision only. The

vector packed decimal facility adds support for performing BCD operations that use vector registers instead of memory-to-memory operations.

These vector facilities can provide significant performance improvements for workloads that are able to take advantage of them relative to the older scalar instructions. The ability to operate on multiple elements in parallel, having access to larger number of vector registers, and avoiding expensive memory-to-memory operations, can all help improve performance.

Vector programming support

The vector programming support exposes the hardware vector facilities directly to the user. It provides data types, language extensions, and built-in functions for C/C++ that allows users to write high-performance vector code. The ideal use case for vector programming support is performance critical regions that would benefit the most from hand-tuned vector code.

The vector data types allow C/C++ code to work with vector data in the same way that one would work with scalar integer and floating-point types. The vector data types support the same indexing syntax as arrays, making it easy to access individual elements of a vector.

The language extensions add vector data type support for most unary, binary, and relational operators. When these operators are applied to vector operands, the operation is performed on each element of the vector. The C/C++ languages are also extended to support vector literals, making it easier to initialize vector values.

The vector built-in functions expose many of the underlying hardware vector instructions to the users to use directly in their code. These vector built-ins, along with the data types and language extensions, provide a seamless way to integrate vector instructions into C/C++ code.

Note: To enable the vector programming support, the following options are required: `VECTOR`, `FLOAT(AFP(NOVOLATILE))`, `ARCH(11)` or higher, and `TARGET(zOSV2R1)` or higher.

AutoSIMD

The AutoSIMD optimization is available starting from z/OS V2R2 XL C/C++ and supports integers and double precision floating-point values. AutoSIMD automatically transforms source code to use vector instructions for SIMD. The first step is safety analysis phase that identifies whether the transformation is safe to apply. The next step is profitability analysis to determine whether the transformation will be beneficial. Finally, the scalar source code is transformed to use vector instructions. AutoSIMD is able to transform both loops and basic blocks to use vector instructions.

Figure 1 shows a `for` loop that is performing a summation of the first 256 values in an array. Figure 2 shows the optimized code of Figure 1 by AutoSIMD to use vector instructions. This transformed code is expressed by using vector programming support. The code in Figure 2 is longer because it is more explicit, using built-in functions to show the loads and stores. The transformed code does four sums in parallel using vector registers since a single vector register can hold four floating-point single precision values. This means that we have a quarter of the total number of iterations, 64 versus 256. Finally, the four summations are themselves added up to get the final value. The AutoSIMD optimization is able to transform the code to achieve four times as much work in each loop iteration relative to the original scalar code.

```
float sum = 0.0f;
for (int i = 0; i < 256; i++)
{
    sum += arr[i];
}
```

Figure 1: Simple loop computing the sum of the first 256 element of an array

```
vector float tmpA, tmpB;
vec_xst((vector float){0.0f,0.0f,0.0f,0.0f}, 0, tmpA);
int i = 0;
do {
   tmpA = vec_xl(0, tmpA);
   tmpB = vec_xl(0, ((char *)arr + (4)*(i * 4)));
   vec_xst(tmpA + tmpB, 0, tmpA);
   i = i + 1;
} while (i < 64u);
sum = tmpA[0] + tmpA[1] + tmpA[2] + tmpA[3];
```

Figure 2: Simple loop after AutoSIMD, expressed by using vector programming support

Notes:

- The AutoSIMD optimization requires the following options: `HOT`, `VECTOR(AUTOSIMD)`, `FLOAT(AFP(NOVOLATILE))`, `ARCH(11)` or higher, and `TARGET(zOSV2R1)` or higher.
- The `FLOAT(IEEE)` option is required to enable AutoSIMD for IEEE floating-point values.
- The `ARCH(12)` option and the z/OS V2R3 XL C/C++ compiler is required to enable AutoSIMD for IEEE single precision floating-point values.

Vector single element

One of the limiting factors for scalar floating-point instructions is that they have access to only 16 floating-point registers. The compiler tries to keep as many values in these registers as possible. However, there are times where the number of live values exceeds the number of registers. In these high register pressure situations, the values are temporarily stored out to memory and loaded back later. This spilling operation can be expensive, especially if the value gets evicted from the L1 cache between the store and the load. The simplest solution is to have more registers and that is exactly what the vector facility provides for IEEE floating-point values.

Vector instructions are able to access all 32 128-bit vector registers and most vector floating-point instructions have a single element control bit. When this bit is set, the operation is performed only on the zero-indexed element of a vector register, the values in the other elements are unpredictable and are ignored. The single element control bit enables the use of vector instructions, and therefore vector registers, for scalar floating-point operations. This is very useful for code that is suffering from a larger number spills due to register pressure. Additionally, most scalar floating-point instructions are 2-operand. Instructions of this form have only two operands with one of the operands being used as both a source and a destination. If that value is required in subsequent operations, then it needs to be copied into another register. Most vector floating-point instructions are 3-operand, providing distinct operands for the sources and destination and eliminating the need for register copies to preserve values.

Another aspect that makes this feature useful is that floating-point registers are overlaid with the vector registers. Referring to floating-point registers 0 - 15 is the same as referring to the most significant half of the corresponding vector registers 0 - 15. This makes it easier to mix floating-point vector and scalar instructions since extra instructions for moving between register types are not required. Simply ensuring that the result of a vector instruction is placed in vector registers 0 - 15 makes it available to scalar floating-point instructions. This is especially useful when a single element vector alternative is not available for a scalar instruction, such as some conversion instructions.

With z/OS V2R3 XL C/C++, the vector single element feature automatically use this single element control bit to improve IEEE floating-point code. Twice as many values can be live and kept in registers before spilling is necessary. The use of vector single element instructions can improve performance, reduce code size, and reduce stack space usage. The only down side of using the vector instructions is the larger size of each instruction compared to the scalar floating-point instruction. Most scalar floating-point instructions are 4 bytes versus the 6-byte length of vector instructions. However, in internal benchmarks this has not been an issue since the benefits of reduced spills more than makes up for any negative performance impact of the larger instruction size.

Table 1 shows the listing snippet without and with vector single element respectively. This listing snippet is from a floating-point intensive workload that is constrained by register pressure. Without vector single element, we notice several loads and stores that are spilling and restoring values. This is because this code has exhausted all 16 floating-point registers. We also see the need for an LDR, which is a register-to-register copy of floating-point registers. This LDR preserves the value that is about to be overwritten by the subsequent 2-operand instruction. The listing snippet without vector single element shows one of the ways of dealing with register pressure without using vector instructions, register-storage form instructions such as MADB. These register-storage form instructions have a memory reference as one of their operands, which eliminates the need for a floating-point register. This can help in some cases but still has several tradeoffs:

- The memory reference cannot act as a destination.
- Memory needs to be referenced every time that operand is referenced.
- The use of a floating-point register has been replaced by the use of one or more general purpose registers for addressing. However, general purpose registers are also limited to 16, so this might cause general purpose register to spill instead.

Table 1: Pseudo-assembly listing without vector single element

Without vector single element

```
STDY     f15,rns18.0.(r14,r8,-128)
MADBR    f2,f4,f3
MADBR    f13,f11,f3
LD       f15,#vsSPILL11(,r4,2240)
MADBR    f9,f2,f3
LD       f2,#vsSPILL2(,r4,2168)
SDBR     f13,f5
STD      f14,#vsSPILL17(,r4,2288)
SDBR     f9,f5
LD       f11,#vsSPILL0(,r4,2152)
LDR      f14,f0
MDBR     f15,f9
LD       f9,#vsSPILL5(,r4,2192)
MADB     f15,f9,rns18.0.(r14,r9,40)
LD       f9,#vsSPILL9(,r4,2224)
STDY     f15,rns18.0.(r14,r10,-497112)
```

With vector single element

```
STDY     f2,rns18.0.(r14,r8,-128)
WFMADB   v24,v6,v24,v10
VLEG     v27,rns18.0.(r14,r9,40),0
WFMDB    v2,v0,v15
WFSDB    v24,v24,v3
WFMDB    v25,v7,v24
WFMADB   v24,v6,v14,v13
WFMADB   v0,v6,v24,v10
WFADB    v24,v4,v5
WFSDB    v26,v0,v3
WFMADB   v0,v27,v9,v25
WFMDB    v7,v7,v26
STDY     f0,rns18.0.(r14,r10,-497112)
```

Enabling vector single element eliminates all spill-related loads and stores in this snippet since we are able to keep all values in registers. The 3-operand form of the vector instructions eliminates the need for LDR instructions to preserve operands. Vector instructions lack register-storage form instructions, but the larger number of registers makes them unnecessary. Overall, the version of the benchmark with vector single element enabled is 20% faster, has 80% fewer spills, and uses 25% less stack space.

Notes:

- The following options are necessary to enable vector single element for IEEE-754 double precision: `OPT(2)` or higher, `VECTOR`, `FLOAT(AFP(NOVOLATILE), IEEE)`, `ARCH(11)` or higher, and `TARGET(zOSV2R1)` or higher.
- Additionally, `ARCH(12)` is required to enable vector single element for IEEE-754 single-precision.

Vector long double

The `vector long double` feature is similar to vector single element, except that it works on IEEE-754 quadruple precision values. Existing, quadruple precision instructions require the use of floating-point register pairs since the 128-bit values are too large for a single 64-bit floating-point register. However, with the vector enhancements facility 1 in z14, support was added for quadruple precision operations by using 128-bit vector registers and vector instructions.

Moving from eight floating-point register pairs to 32 vector registers can improve the performance of code that is constrained by register pressure. The 3-operand form of the vector instructions can also avoid the extra copy code that is needed to preserve operands. In this case, the savings in copy code are even greater because we save on two instructions, one for each register in the pair. Similarly, the savings from reduction in spill code are also greater because the register pairs required two load and two store instructions. Unlike the scalar single and double precision floating-point instructions, quadruple precision instructions don't have register-storage form instructions.

Table 2 shows the listing from a floating-point intensive workload without and with `vector long double` enabled respectively. This is the same work load from Table 1, but operating on quadruple precision values instead of double precision. The first thing to note is the need for pairs of loads and store due to paring of registers. Sequential even registers or sequential odd registers are paired together to store a full 128-bit value. For example, the loads into f4 and f6 constitute a single 128-bit value. Similarly, the two store instructions (STD) are spilling a single 128-bit value. With `vector long double`, the two load and store instructions can be replaced with a single vector load or store instruction. The listing with `vector long double` is shorter, but manages to do more meaningful work because it does not need as many loads before an actual operation.

Table 2: Pseudo-assembly listing without vector long double

Without vector long double

```
LD      f4,(*)ldouble(r9,r1,16)
LD      f6,(*)ldouble(r9,r1,24)
LDR     f5,f0
LDR     f7,f2
STD     f6,#SPILL0(,r4,1856)
STD     f4,#SPILL1(,r4,1848)
AXBR    f5,f4
LD      f4,(*)ldouble(r9,r1,32)
LD      f6,(*)ldouble(r9,r1,40)
LD      f8,(*)ldouble(r9,r1,48)
LD      f10,(*)ldouble(r9,r1,56)
```

With vector long double

```
VL      v13,(*)ldouble(r9,r1,16)
VL      v15,(*)ldouble(r9,r1,32)
VL      v5,(*)ldouble(r9,r1,48)
WFAXB   v7,v10,v13
VL      v3,(*)ldouble(r9,r1,64)
VL      v12,(*)ldouble(r9,r1,128)
WFAXB   v7,v15,v7
WFSXB   v11,v5,v3
```

Vector long double enabled code provides the same benefits that we saw earlier with vector single element. However, the benefits in this case are amplified due to the move from register pairs to single vector registers. The vector long double enabled version in Table 2 is over twice as fast, has 92% fewer spills and uses 76% less stack space.

Note: The z/OS V2R3 XL C/C++ compiler will automatically use the `vector long double` feature where possible with the following set of options: `OPT(2)` or higher, `VECTOR`, `FLOAT(AFP(NOVOLATILE), IEEE)`, `ARCH(12)`, and `TARGET(zOSV2R1)` or higher.

Vector packed decimal

The z/OS XL C/C++ compiler provides a fixed-point decimal type for working with packed decimal values in C code. For operations on these fixed-point decimal values, the compiler uses storage-and-storage form packed decimal instructions where all the operands are memory references. These storage-and-storage instructions are necessary because the decimal values, which can be up to 16 bytes in length, are too large to fit in general purpose registers.

With the release of z14, the new vector packed decimal facility adds vector instructions for operating on packed decimal values in vector registers. Vector registers are able to hold all 16 bytes of a signed packed decimal value. Instead of repeatedly referencing storage, as is the case with storage-and-storage instructions, values can instead be loaded into vector registers. Vector decimal instructions are then able to use these registers as operands. Additionally, the new vector packed decimal instructions are 3-operand versus the 2-operand storage-and-storage instructions, which means that a storage-to-storage copy is not necessary to preserve an operand when using vector decimal instructions. So in addition to better performance from the reduced delays between register to register dependencies versus storage to storage dependencies, we may also have fewer overall instructions.

Note: When compiling with the z/OS V2R3 XL C/C++ compiler, the vector packed decimal feature requires the following options: `VECTOR`, `ARCH(12)`, `FLOAT(AFP(NOVOLATILE))`, and `TARGET(zOSV2R1)` or higher.

Vectorized C string functions

The z/OS XL C/C++ compiler performs inline expansion of several C string functions, which allows the code to be better optimized based on how the string function is being used at each invocation site. Before z/OS V2R3 XL C/C++, these inline expansions used instructions like Compare Logical Character (CLC) to implement the C string functions. The vector facility for z/Architecture, introduced in z13, added vector string instructions for the same operations. However, the vector string instructions are able to operate on up to 16 characters in parallel, potentially improving the performance of these functions.

The z/OS V2R3 XL C/C++ compiler may decide to use these new vector string instructions when expanding certain C string functions. The following C string functions are candidates for vectorized inline expansions: `memchr`, `memcmp`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strlen`, `strncat`, `strncmp`, `strncpy`, and `strrchr`. For cases where the parameters to these functions are known at compile time, the z/OS XL C/C++ compiler may evaluate them at compile time and replace the invocation directly with the result. For the cases where this compile-time evaluation is not possible, the compiler will decide between the scalar and vector expansions based on what it thinks will give the best performance.

Table 3 shows the possible expansion for the `strchr` function, which searches for a character in a null terminated string given a pointer to the string and the character to search for. In the included listings, the pointer to the string is in register `r1` and the character to search for is in register `r2`. The result, the address of the first occurrence of the character in the string if found and null otherwise, is placed into register `r3`.

The scalar expansion uses the Translate and Test (TRT) instruction to do the search. The listing can be broken up into three parts: setup, search, and cleanup. Everything up to label `@1L2` is setting up the table that will be used by the TRT later. MVI instructions are used to place a value of two at index zero and a value of one at the index corresponding to the numerical value of the search character. All other entries in the table are zero. The code between the `@1L2` and `@1L3` labels uses this table with the TRT instruction to search the string for the character. Finally, after we find either the search character or the null terminator, we reset the nonzero values in the table back to zero.

The vector expansion uses the Vector Find Element Equal (VFEEB) to perform the search. Again, the listing starts with some setup, though in this case there is no table, instead the search character is replicated across all 16 elements of vector register v2 through the VREPB instruction. The code then enters a loop, which loads up to 16 bytes of the string and searches it for either the search character or the null terminator. When a possible match is found, there is an extra check to differentiate between the search character and the null terminator before the result address is computed into register r3.

Table 3: Scalar and vector pseudo-assembly listing for the strchr function

Scalar expansion

```
     NILF    r2,F'255'
     L       r5,TRT_PTR(,r12,572)
     LR      r3,r1
     MVI     TRT(r5,0),2
     LA      r8,TRT(r2,r5,0)
     MVI     TRT(r8,0),1
@1L2 DS      0H
     TRT     Cuchar(256,r3,0),TRT(r5,0)
     LA      r3,Cuchar(,r3,256)
     JE      @1L2
     LA      r3,0
     NILF    r2,F'255'
     BRCT    r2,@1L3
     LR      r3,r1
@1L3 DS      0H
     MVI     TRT(r5,0),0
     MVI     TRT(r8,0),0
```

Vector expansion

```
     LR      r0,r2
     LA      r3,0
     STC     r2,VTEMP1(,r4,2032)
     NILF    r0,F'255'
     LA      r2,0
     VL      v0,VTEMP1(,r4,2032)
     VREPB   v2,v0,0
@1L2 DS      0H
     VLBB    v0,Cuchar(r2,r1,0),2
     LCBB    r5,Cuchar(r2,r1,0),2
     LR      r9,r2
     ALR     r2,r5
     VFEEB   v0,v0,v2,b'0010'
     VLGVB   r10,v0,7
     CLRJNL  r10,r5,@1L2
     ALRK    r2,r10,r9
     LLC     r5,Cuchar(r2,r1,0)
     CLRJNE  r5,r0,@1L1
     LA      r3,Cuchar(r2,r1,0)…
```

The benefit of the vector expansion over its scalar counterpart is that the VFEEB instruction is able to search up to 16 characters at a time. The TRT instruction must proceed one character at a time, looking up each character in the table before moving on to the next one. There is some additional bookkeeping that needs to happen with the vector expansion. The Vector Load to Block Boundary (VLBB) and Load Count to Block Boundary (LCBB) instructions need to be used to avoid crossing a page boundary when loading 16 bytes. This is something the scalar expansion does not have to deal with. Additionally, code for indexing into the string also needs to be generated for the vector expansion, whereas it is encapsulated in the TRT instruction in the scalar case. Overall, the vector expansion is able to provide a speed-up of over 2x in most cases over the scalar expansion.

Notes:

- To allow the compiler to generate these vectorized inline expansions, the `string.h` header must be included in any source to use these string functions.

- The following compiler options are required: `OPT(2)` or higher, `VECTOR(AUTOSIMD)`, `ARCH(11)` or higher, `FLOAT(AFP(NOVOLATLE))`, and `TARGET(zOSV2R1)` or higher.

Summary

The improvements that are brought by the vector facilities in the latest Z hardware can offer significant performance improvements for code that takes advantage of them. The set of vector optimizations that is provided by the z/OS V2R3 XL C/C++ compiler allows users to use these newest additions to the ISA to improve the performance of their code. In most cases, source changes are not necessary and only recompilation with the appropriate compiler options is required. The compiler uses the new vector facilities in two distinct ways. The first is by exposing inherent data level parallelism to the hardware through optimizations like AutoSIMD and vectorized C string functions. The second is by using new hardware features to speed up scalar operations such as vector single element, `vector long double`, and vector packed decimal. The vector optimizations in the z/OS V2R3 XL C/C++ compiler are key to getting the most out of the latest hardware.

References

- For more information about performance tuning, see Part 5. Performance optimization in the *z/OS V2R3 XL C/C++ Programming Guide (SC14-7315-30)*
- Chapter 35. Using vector programming support in the *z/OS V2R3 XL C/C++ Programming Guide (SC14-7315-30)*.
- AutoSIMD compiler optimizations for z/OS XL C/C++ programs

Purchasing

Information about purchasing z/OS XL C/C++ is available at the Marketplace website:

https://www.ibm.com/us-en/marketplace/xl-cpp-compiler-zos

Contacting IBM

IBM welcomes your comments. You can send them to compinfo@cn.ibm.com.

September 2018