Use the IBM z13 SIMD unit and the IBM z/OS XL C/C++ compiler to add parallelism to your C/C++ programs

Process data efficiently with vector data types and operations

Rajan Bhakta (rbhakta@us.ibm.com)
Technical Architect, z/OS XL C/C++ compilers
IBM

14 July 2015

The IBM z13 hardware provides a new SIMD unit. This article describes how to use the IBM z/OS XL C/C++ language to take advantage of the new processor and exploit the enhanced parallelism it offers. This article also provides an overview of the new data types, the operations that can be done on those data types, and the built-in functions to make vector programming easier.

Parallelism in C and C++ languages

The latest C and C++ language standards provide threading support and atomics, but even with this support, parallelism is still not exploited as deeply as it can be. For example, there is no prescribed way for single instruction/multiple data (SIMD) style coding.

The IBM z/OS XL C/C++ V2.1.1 compiler adds in support for SIMD data level parallelism with new data types that fit in with the existing C and C++ type system, use the existing operators to work on the new types in a way that is natural and intuitive, and a number of new built-in functions (BIFs) that can be used to exploit SIMD functionality at the hardware level.

Programming efficient code to do a series of similar or same transformations to a large amount of data is a common task. As long as the data does not have dependencies during the transformations, it is usually a prime candidate for parallelism.

Encapsulating the data in a form that can be manipulated efficiently, transformed effectively, and performed quickly requires a strong connection between the hardware and software. Without hardware support, the software overhead often results in poorly performing code. Without software support, the code is usually very hard to maintain and has poor readability.
The new IBM z13 processors provide the hardware for increased parallelism and the IBM z/OS XL C/C++ V2.1.1 compiler provides the software to exploit the hardware to allow efficient, effective, and fast performing data processing for various fields such as business analytics.

**SIMD principles**

Large data sets (the *multiple data* part of SIMD) that require a series of operations lend themselves well to data types that encapsulate chunks of the data and allow the operations to take effect on the chunks.

The IBM z13 SIMD unit allows handling of 16 byte chunks of data in various element sizes. It supports instructions (the *single instruction* part of SIMD) to perform operations on 16 individual single byte data elements, 8 individual double byte elements, 4 individual four byte elements or 2 individual eight byte elements (again, the *multiple data* part of SIMD). There are also instructions to perform 1 sixteen byte element operations, but since that does not really fit the SIMD model, that aspect will not be discussed in this article.

**New data types**

The element sizes supported by the SIMD unit map naturally to some of the basic C and C++ data types. You can chunk the data into various sized groups of normal C and C++ types. This chunking is represented by a new family of data types called the vector data types. The data types are the same for both C and C++ allowing interoperability between the two languages.

The vector family of data types share common characteristics in that they all syntactically begin with the *vector* or *__vector* keyword, followed by the actual element type. The *vector* keyword is introduced with the VECTOR option. This article will use the *vector* keyword for simplicity, though *__vector* is valid wherever the *vector* keyword is present.

The elements that are supported by the vector family of types are the normal integral types (*char* to *long long int*) and the double type, along with boolean versions of each of the integral types as seen in Table 1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Interpretation of content</th>
<th>Range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector unsigned char</td>
<td>16 unsigned char</td>
<td>0..255</td>
</tr>
<tr>
<td>vector signed char</td>
<td>16 signed char</td>
<td>-128..127</td>
</tr>
<tr>
<td>vector bool char</td>
<td>16 unsigned char</td>
<td>0 (FALSE), 255 (TRUE)</td>
</tr>
<tr>
<td>vector unsigned short</td>
<td>8 unsigned short</td>
<td>0..65535</td>
</tr>
<tr>
<td>vector unsigned short int</td>
<td>8 unsigned short</td>
<td>0..65535</td>
</tr>
<tr>
<td>vector signed short</td>
<td>8 signed short</td>
<td>-32768..32767</td>
</tr>
<tr>
<td>vector signed short int</td>
<td>8 signed short</td>
<td>0 (FALSE), 65535 (TRUE)</td>
</tr>
<tr>
<td>vector bool short</td>
<td>8 unsigned short</td>
<td>0 (FALSE), 65535 (TRUE)</td>
</tr>
<tr>
<td>vector bool short int</td>
<td>4 unsigned int</td>
<td>0..2^32-1</td>
</tr>
</tbody>
</table>
Use the IBM z13 SIMD unit and the IBM z/OS XL C/C++ compiler to add parallelism to your C/C++ programs

<table>
<thead>
<tr>
<th>Vector Type</th>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector signed int</code></td>
<td>4 signed int</td>
<td>$-2^{31} \ldots 2^{31}-1$</td>
</tr>
<tr>
<td><code>vector bool int</code></td>
<td>4 unsigned int</td>
<td>0 (FALSE), $2^{32}-1$ (TRUE)</td>
</tr>
<tr>
<td><code>vector unsigned long long</code></td>
<td>2 unsigned long long</td>
<td>0..$2^{64}-1$</td>
</tr>
<tr>
<td><code>vector signed long long</code></td>
<td>2 signed long long</td>
<td>$-2^{63} \ldots 2^{63}-1$</td>
</tr>
<tr>
<td><code>vector bool long long</code></td>
<td>2 unsigned long long</td>
<td>0 (FALSE), $2^{64}-1$ (TRUE)</td>
</tr>
<tr>
<td><code>vector double</code></td>
<td>2 double</td>
<td>IEEE-754 double (64 bit) precision floating-point values</td>
</tr>
</tbody>
</table>

Note that the `long long int` types are available since the VECTOR option implies LANGLVL(LONGLONG) and the `double` type is available only in IEEE mode, which can be used by specifying the FLOAT(IEEE) option.

The data types in the vector family are all aligned on an 8 byte boundary.

Specifying literals of the data types is done similar to the C99 compound literal mechanism with the vector type as the cast part and the vector element values in the brace (or parenthesized) list. The brace or parenthesized element list can also be used to initialize the vector types.

**Operations on vectors**

Having the vector data types is not very useful without being able to do transformations or operations on the data. Using a number of the normal C and C++ operators just as if the vector data types were normal basic arithmetic types allows a concise and readable mechanism to write SIMD algorithms.

**Arithmetic operations**

Common arithmetic operations such as addition, subtraction, increment, decrement, negation, and others are supported. These operations, like most of the vector operations, act element by element, following the SIMD model. For example, the following code results in the variable `result` having the value of four integers, all with the value 55, being displayed to stdout.

```c
#include <stdio.h>
int main(void) {
    vector unsigned int a = {1, 2, 3, 4};
    vector unsigned int b = {54, 53, 52, 51};
    vector unsigned int result = a + b;
    printf("Result: %vld\n", result);
    return result[2];
}
```

To compile this code in USS, the command is as follows:

```
xlc -qvector -qarch=11 getResult.c -o getResult
```

The example above also uses the `[ ]` vector element selection operator. It allows access to individual vector elements for both read and write purposes and acts just like the array index.
operator. Note that since vector double types were not used, the -qfloat=ieee option did not need to be specified.

Other operations

Along with the arithmetic and logical operators, there are the usual set of data type operators such as sizeof, __alignof__, address of, typeof, casts, and assignment. A new vec_step operator that gives the number of elements in a vector is also available. This operator can be used to help iterate over the underlying scalar type arrays for example.

For comparison purposes, the relational operators are also supported. For use in conditionals, the relational operators result in either a 1 or 0 value based on the relation between all the elements of one operand and the other. For more information on how to get element by element results and an OR reduced result, see the "Built-in functions" section.

The default argument promotion rules that C and C++ provide do not take effect on vector types. This means that the operands might have to be more strictly related than an equivalent scalar operation is related. For example, the bit-wise left shift operator requires the same (ignoring signed-ness) vector type (to allow element by element shifting) or an unsigned long data type (for a shift by a single number of bits) as the right operand. It also does not allow vector boolean types since the logical values for those are only the maximum element type value and 0.

Built-in functions

For the most efficient access to machine instructions and operations that do not lend themselves to normal C and C++ operators, the IBM z/OS XL C/C++ compiler has added in a number of built-in functions (BIFs) to handle those operations.

These vector BIFs are prototyped in builtins.h just like most of the other BIFs. They are only active when the VECTOR option is specified.

The BIFs that are available include specialized arithmetic operations (which are not available with traditional C and C++ operators), compare, range compare, finding elements, gather/scatter, mask generation, copy until zero, load and store, specialized logical operations, merge, pack/unpack, replicate, rotate and shift, rounding and conversion, test and ANY and ALL predicates.

These BIFs usually correspond to the machine SIMD instructions to allow efficient and high performing code. For example, in the following program code, for vec_any_lt you see the VFCH (VECTOR FP COMPARE HIGH) instruction being generated in the pseudo-assembly listing.

```c
#include <builtins.h>
#include <math.h>
#include <stdio.h>

double getMinimumValue(vector double *list, int numberOfArrayElements) {
    vector double currentMinimum = { INFINITY, INFINITY }; // Infinities
    for (int i = 0; i < numberOfArrayElements; i++) {
        if (vec_any_lt(list[i], currentMinimum)) {
            // At least one element in the list element is smaller
        }
    }
    return currentMinimum[0];
}
```
Use the IBM z13 SIMD unit and the IBM z/OS XL C/C++ compiler
to add parallelism to your C/C++ programs

```c
if (list[i][0] < list[i][1]) {
    currentMinimum = vec_splat(list[i], 0);
} else {
    currentMinimum = vec_splat(list[i], 1);
}
}
return currentMinimum[0];

#define NUM_ELEMENTS 3
int main(void) {
    vector double array[NUM_ELEMENTS] = {
        { 5.5, 6.6 },
        { 7.7, -3.14 },
        { 2.73, 0.0 }
    };
    double minValue = getMinimumValue(array, NUM_ELEMENTS);
    printf("Minimum Value: %e\n", minValue);
    return 55;
}
```

To compile this code in USS, the command is as follows:

```
xlc -qvector -qarch=11 -qfloat=ieee -qlist -qlanglvl=extc99 getMinimum.c -o getMinimum
```

The BIFs often translate directly into the associated machine instruction, but the compiler can still optimize it better (perhaps even removing the instruction) if it sees a better way of doing things.

Although the example above is simple, it does show the use of array indexing with vector element indexing, for both initialization and for selecting elements of the vector. It also shows vector BIFs and vector types being used as parameters and arguments to functions.

**Conclusion**

The IBM z13 SIMD unit provides a very rich and important mechanism to help crunch data faster and write code that exploits parallelism at the data level. With the new data types, operations, and BIFs in C and C++, writing programs to exploit the new unit is very easy and intuitive.

Along with the other features of the IBM z/OS XL C/C++ compiler, such as architecture sections, inline assembly, and high optimizations, programs can be written to fully exploit the new processor providing a huge boost to data analytics and processing.

**READ:** For more information on vector programming, read the "IBM z/OS XL C/C++ V2R1 Programming Guide" (SC14-7315-01)
About the author

Rajan Bhakta

Rajan Bhakta has six years of development experience for IBM XL C. He is currently the ISO C Standards representative for Canada, and the C representative for IBM in INCITS. He is also the Technical Architect for z/OS XL C/C++.