Vector Function Application Binary Interface Specification for POWER Architecture

Vector Function ABI Overview

This Vector Function ABI provides the ABI for vector functions generated by GCC compilers supporting SIMD constructs of OpenMP 4.0 [1] and above. These SIMD constructs are also available without OpenMP in GCC compilers that implement the __attribute__ ((__simd__)) for function declarations and definitions.
The ABI described here applies only for C/C++ functions.
Use of a SIMD construct for a function declaration or definition enables the creation of vector versions of the function from the scalar version of the function. The vector variants can be used to process multiple instances concurrently in a single invocation in a vector context (e.g., most typically in vectorizing loops during the optimization phase of compilation.)
For a function definition, use of #pragma omp declare simd or __attribute__ ((__simd__)) enables creation of vector versions by the compiler.
For a function declaration, use of #pragma omp declare simd or __attribute__ ((__simd__)) enables the compiler to know the exact list of available vector function implementations provided by a library. The library's vector functions will use the OpenMP pragma or GCC attribute SIMD constructs in their prototypes.
The Vector Function ABI defines a set of rules that caller and callee functions must obey. The rules consist of:

- Calling convention (how arguments are passed to the vector function and how values are returned from the vector function)
- Vector length (the number of concurrent scalar invocations to be processed per invocation of the vector function)
- Mapping from element data types to vector data types
- Ordering of vector arguments
- Vector function masking
- Vector function name mangling
- Compiler generated vector function variants

This specification shall be considered an extension to the OpenPOWER 64-bit ELF V2 ABI Specification for Power Architecture [2]. As such, it applies to Power ISA 2.07 and above, requiring the VSX vector facility described in that ABI and ISA.

Calling convention

The vector functions should use the calling convention described in Section 2.2, Function Calling Sequence, of OpenPOWER 64-bit ELF V2 ABI Specification for Power Architecture [2] document.

Vector Length

Every vector variant of a SIMD-enabled function has a vector length (VLEN).
If OpenMP clause "simdlen" is used, the VLEN is the value of the argument of that clause. The VLEN value must be a power of 2.
In the other cases (GCC simd attribute used or OpenMP simdlen not used) the notion of a function's "characteristic data type" (CDT) is used to compute the vector length. CDT is defined in the following order:

1. For non-void function, the CDT is the return type.
2. If the function has any non-uniform, non-linear parameters, then the CDT is the type of the first such parameter.
3. If the CDT determined by a) or b) above is a homogeneous aggregate (see "Parameter Passing in Registers" in [2]), the CDT is the entire homogeneous aggregate. For example, a parameter "double x[2]" has a CDT of type double[2] and size 16 bytes. The same applies for a complex double type.
4. If the CDT determined by a) or b) above is a nonhomogeneous struct, union, or class type (see "Parameter Passing in Registers" in [2]) which is pass-by-value, the characteristic data type is int.
5. If none of the above three cases is applicable, the CDT is int.

The VLEN is then determined based on the CDT and the size of the vector register for the ISA. VLEN is computed using the formula below:
VLEN = sizeof(vector_register) / sizeof(CDT).
VSX has sizeof(vector_register) = 16.

Mapping from element data type to vector data type

The vector data types for parameters are selected depending on ISA, vector length, data type of original parameter, and parameter specification.
For uniform and linear parameters (detailed descriptions are found in [1]), the original data type is preserved.
For vector parameters, vector data types are selected by the compiler. The mapping from element data type to vector data type is described below.

- The bit size of the vector data type of a parameter is computed as:
  size_of_vector_data_type = VLEN * sizeof(original_parameter_data_type) * 8
  For instance, for a VSX vector function with parameter data type "int":
  VLEN = 4, size_of_vector_data_type = 4 * 4 * 8 = 128 bits, which means one argument of type vector signed int.
- If the size_of_vector_data_type is greater than the width of the vector register, multiple vector registers are used for passing the vector parameter. For instance, a VSX vector function with parameter data type of "double":
  VLEN = 4, size_of_vector_data_type = 4 * 8 * 8 = 256 bits, the vector data type is vector double [2], which means 2 arguments of type vector double are to be passed.

Ordering of Vector Arguments

When a parameter in the original data type results in one argument in the vector function, the ordering rule is a simple one-to-one match with the original argument order.
For example, when the original argument list is (int a, float b, int c), VLEN is 4, and all a, b, and c are classified vector parameters, the vector function argument list becomes (vector int vec_a, vector float vec_b, vector int vec_c).
There are cases where a single parameter in the original data type results in multiple arguments in the vector function. Those additional second and subsequent arguments are inserted in the argument list right after the corresponding first argument, not appended to the end of the argument list of the vector function. For example, if the original argument list is (int a, double b, int c), VLEN is 4, and all a, b, and c are classified as vector parameters, the vector function argument list becomes (vector int vec_a, vector double vec_b1, vector double vec_b2, vector int vec_c). For an example involving homogeneous aggregates, if the original argument list is (int a, double b[2], int c), VLEN is 4, and all a, b, and c are classified as vector parameters, the vector function argument list becomes (vector int vec_a, vector double vec_b0_0, vector double vec_b0_1, vector double vec_b1_0, vector double vec_b1_1, vector int vec_c).

Masking of Vector Functions

Masking of vector functions is not currently supported by the Power ISA. Compilers should not generate code for masked variants of vector functions until such time (if ever) as masked vector instructions are supported.

Vector Function Name Mangling

The name mangling of generated vector functions based on standardized annotation is an important part of this ABI. It allows caller and callee functions to be separately compiled. Using the function prototypes in header files to communicate vector function annotation information, the compiler can perform function matching when vectorizing code at call sites. The vector function name is mangled as the concatenation of the following items:
<prefix> <isa> <mask> <len> <parameters> "_" <original_name>
<prefix> := "_ZGV"
<original_name> := name of scalar function, including C++ mangling
<isa> := "b" (VSX)
<mask> := "N" (No Mask)
| "M" (Mask)
<len> := VLEN
<parameters> := /* empty */
| <parameter> <opt-align> <parameters>
<parameter> := "l" <stride> // linear(x:linear_step) or
// linear(val(x):linear_step) when x is a
// pointer
| "R" <stride> // linear(ref(x):linear_step)
| "U" <stride> // linear(uval(x):linear_step)
| "L" <stride> // linear(val(x):linear_step) or
// linear(x:linear_step) when x is a reference

| "u" // uniform parameter
| "v" // vector parameter
<stride> := /* empty */ // linear_step is equal to 1
| "s" <non-negative-decimal-number> // linear_step is passed
// in another argument, decimal number is the position # of linear_step
// argument, which starts from 0
| <number> // linear_step is literally constant stride
<number> := [n] non-negative decimal integer // n indicates negative
<opt-align> := /* empty*/
| "a" non-negative decimal integer
Please refer to section 2.7, Compiler generated variants of vector functions, below, for examples of vector function name mangling.
Note that the value "M" for the <mask> field is reserved until such time (if ever) as masked vector instructions are supported in the Power ISA.

Compiler generated variants of vector functions

The compiler should generate vector variants, masked and/or umasked as appropriate, depending on the SIMD construct used to enable vectorization. Compiler implementations must not generate calls to versions that are unavailable unless some non-standard pragma or clause is used to declare those other versions available.
Example 1.

```
#pragma omp declare simd notinbranch uniform(q) aligned(q:16) linear(k:1)
float foo (float *q, float x, int k)
{
q[k] = q[k] + x;
return q[k];
}
```

Below is the vector function’s prototype given "foo" and its associated pragma.

- vector float _ZGVbN4ua16vl_foo (float *, vector float, int)

Where "foo" is the original function name, "_ZGV" is the prefix of vector function names, "b" indicates the VSX ISA, "N" indicates an unmasked version, "4" is the vector length for the ISA, "ua16" indicates uniform(q) and align(q:16), "v" indicates second argument x is a vector argument, "l" indicates linear(k:1) - k is a linear variable whose stride is 1.
Example 2.

```
#pragma omp declare simd notinbranch
double foo (double x)
{
return x * x;
}
```

Below is the vector function’s prototype given "foo" and its associated pragma.

- vector double _ZGVbN2v_foo (vector double)

Where "foo" is the original function name, "_ZGV" is the prefix of vector function names, "b" indicates the VSX ISA, "N" indicates an unmasked version, "2" is the vector length for the ISA/CDT combination, "v" indicates single argument x is a vector argument.

References

[1] OpenMP 4.0 Specification
http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
[2] OpenPOWER 64-bit ELF V2 ABI Specification - Power Architecture
https://openpowerfoundation.org/?resource_lib=64-bit-elf-v2-abi-specification-power-architecture
[3] Section 6.33 Declaring Attributes of Functions
https://gnu.org/onlinedocs/gcc/Function-Attributes.html
[4] Section 6.33.1 Common Function Attributes
https://gnu.org/onlinedocs/gcc/Common-Function-Attributes.html
[5] Libmvec
https://sourceware.org/glibc/wiki/libmvec