Document: WG14 N1408

Submitter: Barry Minor, Michael Wong, Raymond Mak (IBM)
Email:bminor@us.ibm.com, michaelw@ca.ibm.com, rmak@ca.ibm.com
Submission Date: 2009-09-23
Related WG14 documents: N/A
Subject: Proposed Common Vectorized Types for C based on OpenCL

1. Introduction

This aim of this proposal is to suggest a standardized vector types for C. This can be in the form of a TR or with enough interest, with the current C1X. However, this proposal can be viewed as an introduction to vector types.

This paper will look at some history for vectorized types, as well as examples of current implementation experience of vectorized types in various industrial compilers. This paper is not an exhaustive authoritative research on the topic, and we apologize for omissions.

The motivation for a common vectorized type is to:

1. support one of the most promising form of parallel programming and improves the viability of C as a future programming language
2. Reduce the multiple ways that vectorized types can be created even within a single compiler vendor's product, and improve cross-compiler support through standardization.

2. History

The basic concept behind vector processing is to enhance the performance of data-intensive applications by providing hardware support for operations that can manipulate an entire vector (or array) of data in a single operation. The number of data elements operated upon at a time is called the vector length.

Scalar processors perform operations that manipulate single data elements such as fixed-point or floating-point numbers. For example, scalar processors usually have an instruction that adds two integers to produce a single-integer result.

Vector processors perform operations on multiple data elements arranged in groups called vectors (or arrays). For example, a vector add operation to add two vectors performs a pair-wise addition of each element of one source vector with the corresponding element of the other source vector. It places the result in the corresponding element of the destination vector. Typically a single vector operation on vectors of length n is equivalent to performing n scalar operations.

Processor designers are continually looking for ways to improve application performance. The addition of vector operations to a processor architecture is one method that a processor designer can use to make it easier to improve the peak performance of a processor. However, the actual performance improvements that can be obtained for a specific application depend on how well the application can exploit vector operations.

The concept of vector processing has existed since the 1950s. Early implementations of vector processing (known as array processing) were installed in the 1960s. They used special purpose peripherals attached to general purpose computers. An example is the IBM 2938 Array Processor, which could be attached to some models of the IBM System/360. This was followed by the IBM 3838 Array Processor in later years.

By the mid-1970s, vector processing became an integral part of the main processor in large supercomputers manufactured by companies such as Cray Research. By the mid-1980s, vector processing became available as an optional feature on large general-purpose computers such as the IBM 3090TM.

In the 1990s, developers of microprocessors used in desktop computers adapted the concept of vector processing to enhance the capability of their microprocessors when running desktop multimedia applications. These capabilities were usually referred to as Single Instruction Multiple Data (SIMD) extensions and operated on short vectors. Examples of SIMD extensions in widespread use today include:

Intel Multimedia Extensions (MMX™)

Intel Streaming SIMD Extensions (SSE)

AMD 3DNow!

Motorola AltiVec and IBM VMX/AltiVec

The SIMD extensions found in microprocessors used in desktop computers operate on short vectors of length 2, 4, 8, or 16. This is in contrast to the classic vector supercomputers that can often exploit long vectors of length 64 or more.

The VMX/AltiVec extensions to PowerPC Architecture add a vector processor (VXU) to the PowerPC logical processing model.

The VXU operates on vectors that are a total of 128-bits long. These can be interpreted by the VXU as either:

A vector of sixteen 8-bit bytes

A vector of eight 16-bit half words

A vector of four 32-bit words

The VMX/AltiVec extensions to PowerPC Architecture define 32 vector registers that form the vector register file (VRF). The VRF is architecturally distinct from the standard PowerPC FPRs and GPRs.

The VMX/AltiVec extensions to PowerPC also define two additional registers:

- The VMX/AltiVec status and control register (VSCR), which is used to control the operation of the VXU and report the status of some VMX/AltiVec operations
- The VRSAVE register, which can be used to assist the operating system save state across context switches by providing a mechanism for software to indicate what vector registers are in use

The VMX/AltiVec extensions to PowerPC Architecture define new instructions that use the VXU to manipulate vectors stored in the VRF. These instructions fall into these categories:

- Vector integer arithmetic instructions (on 8-bit, 16-bit, or 32-bit integers)
- Vector floating-point arithmetic instructions (32-bit only)
- Vector load and store instructions
- Vector permutation and formatting instructions
    - Processor control instructions used to read and write from the VMX/AltiVec status and control register
    - Memory control instructions used to manage caches

This technology can be thought of as a set of registers and execution units that can be added to the PowerPC architecture in a manner analogous to the addition of floating-point units. Floating-point units were added to provide support for high-precision scientific calculations and the vector technology is added to the PowerPC architecture to accelerate the next level of performance-driven, high-bandwidth communications and computing applications.

3. Current Implementations

I will list a few examples of vector types that our C and other compiler support:

1. Altivec, VMX

From the IBM XLC++ V10.1 [IBMXLC] manual for VMX vector types:

The following table lists the supported vector data types and the size and possible values for each type.

Type	Interpretation of content	Range of values
Table 1. Vector data types		
vector unsigned char	16 unsigned char	0..255
vector signed char	16 signed char	-128..127
vector bool char	16 unsigned char	0, 255
vector unsigned short		
vector unsigned short int	8 unsigned short	0..65535
vector signed short		
vector signed short int	8 signed short	-32768..32767
vector bool short		
vector bool short int	8 unsigned short	0, 65535
vector unsigned int		
vector unsigned long		
vector unsigned long int	4 unsigned int	$0..2^{32}-1$
vector signed int		
vector signed long		
vector signed long int	4 signed int	$-2^{31}..2^{31}-1$
vector bool int		
vector bool long		
vector bool long int	4 unsigned int	$0, 2^{32}-1$
vector float	4 float	IEEE-754 values6.80564694 * 10^{+38}
vector pixel	8 unsigned short	1/5/5/5 pixel

All vector types are aligned on a 16-byte boundary. An aggregate that contains one or more vector types is aligned on a 16-byte boundary, and padded, if necessary, so that each member of vector type is also 16-byte aligned.

2. OpenCL:1.0

OpenCL [OpenCL] (Open Computing Language) is the first open, royalty-free standard for general-purpose parallel programming of heterogeneous systems. OpenCL provides a uniform programming environment for software developers to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices using a diverse mix of multi-core CPUs, GPUs, Cell-type architectures and other parallel processors such as DSPs.

The specification is implemented by many vendors including Apple, AMD, Nvidia, and RapidMind. All of this is based on the LLVM Compiler technology and use the Clang Compiler as its frontend [OpenCLImpl].

This specification is rapidly becoming an industrial de-factor standard for vectorized types. We will use this as an example of the basis for standardization for the C Programming language.

Type	Description
char*n*	A 8-bit signed two's complement integer vector.
uchar*n*	A 8-bit unsigned integer vector.
short*n*	A 16-bit signed two's complement integer vector.
ushort*n*	A 16-bit unsigned integer vector.
int*n*	A 32-bit signed two's complement integer vector.
uint*n*	A 32-bit unsigned integer vector.
long*n*	A 64-bit signed two's complement integer vector.
ulong*n*	A 64-bit unsigned integer vector.
float*n*	A float vector.

Type	Description
bool*n*	A boolean vector.
double, double*n*	A double precision floating-point number, and double precision vector.
half*n*	A 16-bit float vector.
quad, quad*n*	A 128-bit floating-point number and vectors.
complex half, complex half*n* imaginary half, imaginary half*n*	A complex 16-bit floating-point number, and complex and imaginary 16-bit floating-point vectors.
complex float, complex float*n* imaginary float, imaginary float*n*	A complex single precision floating-point number, and complex and imaginary single precision floating-point vectors.
complex double, complex double*n*, imaginary double, imaginary double*n*	A complex double precision floating-point number, and complex and imaginary double precision floating-point vectors.
complex quad,	A complex 128-bit floating-point number, and

3. GCC vector types using attributes on basic types:
For GCC [GCC], on some targets, the instruction set contains SIMD vector instructions that operate on multiple values contained in one large register at the same time. For example, on the i386 the MMX, 3Dnow! and SSE extensions can be used this way.

The first step in using these extensions is to provide the necessary data types. This should be done using an appropriate typedef:

```
typedef int v4si __attribute__ ((vector_size (16)));
```

The int type specifies the base type, while the attribute specifies the vector size for the variable, measured in bytes. For example, the declaration above causes the compiler to set the mode for the v4si type to be 16 bytes wide and divided

into int sized units. For a 32-bit int this means a vector of 4 units of 4 bytes, and the corresponding mode of foo will be V4SI.

The vector_size attribute is only applicable to integral and float scalars, although arrays, pointers, and function return values are allowed in conjunction with this construct.

All the basic integer types can be used as base types, both as signed and as unsigned: char, short, int, long, long long. In addition, float and double can be used to build floating-point vector types.

4. Sample Proposal

We propose to follow the syntax defined in the current OpenCL specification. Many other models are possible, and there may be value in combining multiple models. The remainder of this paper will describe this proposal.

Data Types

Below is a list of supported vector data types.
Supported values of n are 2, 4, 8, and 16.

char*n* A 8-bit signed two's complement integer vector.
uchar*n* A 8-bit unsigned integer vector.
short*n* A 16-bit signed two's complement integer vector.
ushort*n* A 16-bit unsigned integer vector.
int*n* A 32-bit signed two's complement integer vector.
uint*n* A 32-bit unsigned integer vector.
long*n* A 64-bit signed two's complement integer vector.
ulong*n* A 64-bit unsigned integer vector.
float*n* A float vector.
double*n* A double precision vector.

Alignment of Types

A data item declared to be a data type in memory is always aligned to the size of the data type in bytes. For example, a float4 variable will be aligned to a 16-byte boundary, a char2 variable will be aligned to a 2-byte boundary.

A built-in data type that is not a power of two bytes in size must be aligned to the next larger power of two. This rule applies to built-in types only, not structs or unions.

The compiler is responsible for aligning data items to the appropriate alignment as required by the data type. The behavior of a direct unaligned load/store is considered to be undefined, except for the vector data load and store functions. These vector load and store functions allow you to read and write vectors types from addresses aligned to the size of the vector type or the size of a scalar element of the vector type.

Vector Literals

Vector literals can be used to create vectors from a set of scalars, or vectors. A vector literal is written as a parenthesized vector type followed by a parenthesized set of expressions. Vector literals may be used either in initialization statements or as constants in executable statements.

The number of literal values specified must be one, i.e. referring to a scalar value, or must match the size of the vector type being created. If a scalar literal value is specified, the scalar literal value will be replicated to all the components of the vector type.

Vector Components

The components of vector data types with 1 … 4 components can be addressed as `<vector_data_type>.xyzw`. Vector data types of type `char2, uchar2, short2, ushort2, int2, uint2, long2, ulong2, float2, and double2` can access `.xy` elements. Vector data types of type `char4, uchar4, short4, ushort4, int4,uint4, long4, ulong4, float4, and double4` can access `.xyzw` elements.

Accessing components beyond those declared for the vector type is an error so, for example:

```
float2 pos;

pos.x = 1.0f;  // is legal
pos.z = 1.0f; // is illegal
```

The component selection syntax allows multiple components to be selected by appending their names after the period (`.`).

```
float4 c, a, b;

c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f;      // is a float
```

```
c.xy = (float2)(3.0f, 4.0f); // is a float2
```

The order of the components can be different to swizzle them, or replicated:

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

float4 swiz= pos.wzyx; // swiz = (4.0f, 3.0f, 2.0f, 1.0f)

float4 dup = pos.xxyy; // dup = (1.0f, 1.0f, 2.0f, 2.0f)
```

The component group notation can occur on the left hand side of an expression. To form an l- value, swizzling must be applied to an l-value of vector type, contain no duplicate components, and it results in an l-value of scalar or vector type, depending on number of components specified.

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

pos.xw = (float2)(5.0, 6.0);  // pos =(5.0f, 2.0f, 3.0f, 6.0f)
pos.wx = (float2)(7.0f, 8.0f);// pos =(8.0f, 2.0f, 3.0f, 7.0f)
pos.xx = (float2)(3.0f, 4.0f);// illegal - 'x' used twice

// illegal - mismatch between float2 and float4
pos.xy = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
```

Elements of vector data types can also be accessed using a numeric index to refer to the appropriate element in the vector. The numeric indices that can be used are given in the table below:

Vector Components Numeric indices that can be used

2-component	0, 1
4-component	0, 1, 2, 3
8-component	0, 1, 2, 3, 4, 5, 6, 7
16-component	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F

The numeric indices must be preceded by the letter `s` or `S`.

In the following example

```
float8  f;
```

`f.s0` refers to the 1st element of the `float8` variable `f` and `f.s7` refers to the 8th element of the `float8` variable `f`.

The numeric indices used to refer to an appropriate element in the vector cannot be intermixed with `.xyzw` notation used to access elements of a 1 .. 4 component vector.

For example

```
float4  f, a;

a = f.x12w; // illegal use of numeric indices with .xyzw

a.xyzw = f.s0123; // valid
```

Vector data types can use the `.lo` (or `.odd`) and `.hi` (or `.even`) suffixes to get smaller vector types or to combine smaller vector types to a larger vector type. Multiple levels of `.lo` (or `.odd`) and `.hi` (or `.even`) suffixes can be used until they refer to a scalar term.

The `.lo` suffix refers to the lower half of a given vector. The `.hi` suffix refers to the upper half of a given vector.

Some examples to help illustrate this are given below:

```
float4 vf;

float2 low = vf.lo; // returns vf.xy
float2 high = vf.hi // returns vf.zw
```

The `.odd` suffix refers to the odd elements of a vector. The `.even` suffix refers to the even elements of a vector.

Some examples are given below:

```
float8 vf;
float4 left = vf.odd;
float4 right = vf.even;
float2 high = vf.even.hi;
float2 low = vf.odd.lo;

// interleave L+R stereo stream
float4  left,  right;
float8  interleaved;
interleaved.even = left;
interleaved.odd = right;

// deinterleave
left = interleaved.even;
right = interleaved.odd;

// transpose a 4x4 matrix
void transpose( float4 m[4] )
{
 // read matrix into a float16 vector
```

```
  float16 x = (float16)( m[0], m[1], m[2], m[3] );
  float16 t;


  //transpose
  t.even = x.lo;
  t.odd = x.hi;
  x.even = t.lo;
  x.odd = t.hi;
  //write back
  m[0] = x.lo.lo; // { m[0][0], m[1][0], m[2][0], m[3][0] }
  m[1] = x.lo.hi; // { m[0][1], m[1][1], m[2][1], m[3][1] }
  m[2] = x.hi.lo; // { m[0][2], m[1][2], m[2][2], m[3][2] }
  m[3] = x.hi.hi; // { m[0][3], m[1][3], m[2][3], m[3][3] }
 }
```

Explicit Casts

Explicit casts between vector types are not legal. The example below will generate a compilation error.

```
 float4 f;
 int4  i = (int4) f;  ← not allowed
```

Scalar to vector conversions may be performed by casting the scalar to the desired vector data type. Type casting will also perform appropriate arithmetic conversion. The round to zero rounding mode will be used for conversions to built-in integer vector types. The current rounding mode will be used for conversions to floating-point vector types.

In the examples below:

```
 float  f  = 1.0f;
 float4 va = (float4)f;

// va is a float4 vector with elements (f, f, f, f).

 uchar  u = 0xFF;

float4 vb = (float4)u;

// vb is a float4 vector with elements((float)u, (float)u,
//             (float)u, (float)u).

float f = 2.0f;
int2 vc = (int2)f;
```

```
// vc is an int2 vector with elements ((int)f, (int)f).
```

Explicit Conversions

Explicit conversions may performed using the

`convert_<dest type name>(srctype)`

suite of functions. These provide a full set of type conversions between supported types. The number of elements in the source and destination vectors must match.

In the example below:

```
uchar4  u;
int4    c = convert_int4(u);
```

`convert_int4` converts a `uchar4` vector `u` to a `int4` vector `c`.

```
float   f;
int     i = convert_int(f);
```

`convert_int` converts a `float` scalar `f` to a `int` scalar `i`.

Explicit conversions from a type to the same type has no effect on the type or value of an expression.

The behavior of the conversion may be modified by one or two optional modifiers that specify saturation for out-of-range inputs and rounding behavior.

The full form of the scalar convert function is:

`destType` **`convert_destType<_sat><_roundingMode>`** `(sourceType)`

The full form of the vector convert function is:

`destType`*`n`* **`convert_destType`***`n`***`<_sat><_roundingMode>`** `(sourceType`*`n`*`)`

Data Types

Conversions are available for the following scalar types: `bool`, `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, double and built-in vector types derived therefrom. The operand and result type must have the same number of elements. The operand and result type may be the same type.

Rounding Modes

Conversions to and from floating-point type shall conform to IEEE-754 rounding rules. Conversions involving a floating-point or integer source operand or destination type may have an optional rounding mode modifier. These are described in the table below:

Modifier Rounding Mode Description

`_rte` Round to nearest even
`_rtz` Round towards zero
`_rtp` Round toward positive infinity
`_rtn` Round toward negative infinity

no modifier specified Use the default rounding mode for this destination type, `_rtz` for conversion to integers or the current rounding mode for conversion to floating-point types.

By default, conversions to integer type use the `_rtz` (round toward zero) rounding mode and conversions to floating-point type use the current rounding mode. The only default floating- point rounding mode supported is round to nearest even i.e the current rounding mode will be `_rte` for floating-point types.

Out of Range Behavior and Saturated Conversions

When the conversion operand is either greater than the greatest representable destination value or less than the least representable destination value, it is said to be out of range. When converting between integer types, the resulting value for out of range inputs will be equal to the set of least significant bits in the source operand element that fit in the corresponding destination element. When converting from a floating-point type to integer type, the behavior is implementation- defined.

Conversions to integer type may opt to convert using the optional saturated mode by appending the `_sat` modifier to the conversion function name. When in saturated mode, values that are outside the representable range shall clamp to the nearest representable value in the destination format. (NaN should be converted to 0).

Conversions to floating-point type shall conform to IEEE-754 rounding rules. The `_sat` modifier may not be used for conversions to floating-point formats.

Explicit Conversion Examples

Example 1:

```
short4  s;
```

```
// -ve values clamped to 0
ushort4 u = convert_ushort4_sat( s );

// values > CHAR_MAX converted to CHAR_MAX
// values < CHAR_MIN converted to CHAR_MIN
char4 c = convert_char4_sat( s );
```

Example 2:

```
float4 f;

// values implementation defined for
// f > INT_MAX, f < INT_MIN or NaN
int4 i = convert_int4( f );

// values > INT_MAX clamp to INT_MAX, values < INT_MIN clamp
// to INT_MIN. NaN should produce 0.
// The _rtz rounding mode is
// used to produce the integer values.
int4 i2 = convert_int4_sat( f );

// similar to convert_int4, except that
// floating-point values are rounded to the nearest
// integer instead of truncated
int4 i3 = convert_int4_rte( f );

// similar to convert_int4_sat, except that
// floating-point values are rounded to the
// nearest integer instead of truncated
int4 i4 = convert_int4_sat_rte( f );
```

Example 3:

```
int4 i;
// convert ints to floats using the current rounding mode.
float4 f = convert_float4( i );

// convert ints to floats. integer values that cannot
// be exactly represented as floats should round up to the
// next representable float.
float4 f = convert_float4_rtp( i );
```

Reinterpreting Data As Another Type

It is frequently necessary to reinterpret bits in a vector data type as another vector data type. This is typically required when direct access to the bits in a floating-point type is needed, for example to mask off the sign bit or make use of the result of a vector

relational operator.

Reinterpreting Types Using as_typen()

All data types may be also reinterpreted as another data type of the same size using the **as_typen()** operator. When the operand and result type contain the same number of elements, the bits in the operand shall be returned directly without modification as the new type. The usual type promotion for function arguments shall not be performed.

For example, `as_float(0x3f800000)` returns `1.0f`, which is the value that the bit pattern `0x3f800000` has if viewed as a IEEE-754 single precision value.

When the operand and result type contain a different number of elements, the result shall be implementation-defined. That is, a conforming implementation shall explicitly define a behavior, but two conforming implementations need not have the same behavior when the number of elements in the result and operand types does not match. The implementation may define the result to contain all, some or none of the original bits in whatever order it chooses. It is an error to use `as_typen`() operator to reinterpret data to a type of a different number of bytes.

Examples:

```
 float f = 1.0f;
 uint u = as_uint(f); // Legal. Contains:  0x3f800000

 float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
// Legal. Contains: (int4)
// (0x3f800000, 0x40000000, 0x40400000, 0x40800000)
int4 i = as_int4(f);

 float4 f, g;
 int4  is_less = f < g;

// Legal.  f[i] = f[i] < g[i] ? f[i] : 0.0f
 f = as_float4(as_int4(f) & is_less);

 int i;
// Legal. Result is implementation-defined.
 short2 j = as_short2(i);

 int4 i;
// Legal. Result is implementation-defined.
 short8 j = as_short8(i);

 float4 f;
 //Error.  result and operand have different size
 double4 g = as_double4(f);
```

Implicit Conversions

Implicit conversions are conversions from one type to another without the use of the as_type , explicit casts or explicit conversions using the convert_<dest type name>(srctype) function . When an implicit conversion is done, it is not just a re-interpretation of the expression's value but a conversion of that value to an equivalent value in the new type. For example, the integer value 5 will be converted to the floating-point value 5.0.

Implicit conversions may occur from scalar and vector built-in types. The conversions shall proceed as described in *section 6.3* of the ISO C99 standard TC2, with the following amendments:

- Implicit conversions of non-void scalar types to vector types are allowed, if the conversion to the corresponding scalar type is allowed (see below).
- Conversion to a scalar or vector floating point type shall be correctly rounded as described by IEEE-754 using the current rounding mode.
- Conversion to a scalar or vector floating point type from integer with zero value shall produce +0, per IEEE-754-2008.
- Vector types may not be implicitly converted to any other type.

Conversions are classified into three varieties:

- **promotion** -- the conversion is of type "promotion" if the new type can exactly represent all numeric values representable by the old type.
- **conversion** -- a conversion wherein the new type cannot represent all numeric values representable by the old type.
- **widening** -- a conversion is a widening operation if it is a conversion of a scalar type to a vector type containing elements of the same type.

A conversion may be both a promotion and a widening, or a conversion and a widening in cases where both the type and number of elements in the type change. In such cases, the conversion is considered a single conversion. The compiler shall implement conversions from all non-void scalar types directly to all non-void built-in scalar and vector types.

Implicit scalar promotion to vector types will be extended to all operators, and built-in functions, except for the as_type() operator, and inside a vector constructor with more

than one argument. In cases where the correct operator or function to use may be ambiguous because of the arguments passed to an overloaded function or operator do not exactly match the function or operator prototype, the correct function to be used shall be determined as follows:

1. A set of legal candidate functions (or operators) is prepared. Qualifying functions have the following properties:
    a. They have the correct name.
    b. They are in the scope of the statement.
    c. They have the same number of arguments as the number of arguments provided to the operator or function.
    d. For each argument that does not match the argument type for the function, there must exist a single legal implicit conversion to that type.

2. Candidate functions are evaluated to find the best candidate function. If there is exactly one candidate function that is a better function than all the other candidate functions, then it is selected. Otherwise, the call is ill-formed.

A candidate function A is defined to be better than another candidate function B if for all arguments, there is not an implicit conversion to conform to A's prototype that is worse than the corresponding conversion to conform to B for the same argument, and for some argument the conversion to A is a better conversion than to B.

A conversion is defined to be better than another conversion according to the following ranking:

Conversion Class		
1. Exact match i.e. no conversion	–	**Best**
2. Promotion only		
3. Widening only		
4. Promotion + Widening		
5. Conversion only		
6. Conversion + Widening	–	**Worst**

In addition, when comparing two conversions in the same conversion class, conversions that widen to a vector with fewer elements are better than conversions that widen to a vector with more elements.

Examples:

Implicit conversion of float to double is preferred over float to float2, because promotion is better than widening only.

Implicit conversion of float to float4 is better than float to float8, because float4 is narrower.

Implicit conversion of float to double2 is better than float to int2, because the former is a promotion, and the latter is a conversion.

Implicit conversion from double to float2 is better than double to int4 because float2 has fewer elements than int4. However, implicit conversion from double to float2 is the same as double to int2 – both are in the same conversion class and widen to the same number of elements. The call is ill-formed.

Implicit conversions for pointer types follow the rules described in the C99 specification.

There are no implicit conversions for members of an array or structure. For example, an array of int cannot be implicitly converted to an array of float.

Operators

a. The arithmetic operators add (+), subtract (-), multiply (*) and divide (/) operate on built-in integer and floating-point scalar, and vector data types. The remainder (%) operates on built- in integer scalar and integer vector data types only. All arithmetic operators result in the same fundamental type (integer or floating-point) as the operand they operate on, after operand type conversion. After conversion, the following cases are valid:

The two operands are scalars. In this case, the operation is applied, resulting in a scalar.

One operand is a scalar, and the other is a vector. In this case, the scalar is promoted and/or up-converted to the type used by the vector operand (down-conversion of the scalar type is illegal and will result in a compile time error). The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

The two operands are vectors of the same size. In this case, the operation is done component-wise resulting in the same size vector.

All other cases are illegal. Division on integer types which results in a value that lies outside of the range bounded by the maximum and minimum representable values of the integer type will not cause an exception but will result in an unspecified value. A divide by zero with integer types does not cause an exception but will result in an unspecified value. Division by zero for floating-point types will result in ±infinity or NaN as prescribed by the IEEE-754 standard.

b. The arithmetic unary operators (+ or -), operates on built-in scalar and vector types.

c. The arithmetic post- and pre-increment and decrement (**--** and **++**) operate on built-in scalar and vector types except the built-in scalar and vector float types [21]. All unary operators work component-wise on their operands. These result with the same type they operated on. For post- and pre-increment and decrement, the expression must be one that could be assigned to (an l-value). Pre-increment and pre-decrement add or subtract `1` to the contents of the expression they operate on, and the value of the pre-increment or pre-decrement expression is the resulting value of that modification. Post-increment and post-decrement expressions add or subtract `1` to the contents of the expression they operate on, but the resulting expression has the expression's value before the post-increment or post-decrement was executed.

d. The relational operators greater than (**>**), less than (**<**), greater than or equal (**>=**), and less than or equal (**<=**) operate on scalar and vector types. If the source operands are a vector float, the result is a vector signed integer.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `char`*n* and `uchar`*n* return a `char`*n* result; vector source operands of type `short`*n* and `ushort`*n* return a `short`*n* result; vector source operands of type `int`*n*, `uint`*n* and `float`*n* return an `int`*n* result; vector source operands of type `long`*n*, `ulong`*n*, and *`doublen`* return a `long`*n* result. For scalar types, the relational operators shall return `0` if the specified relation is *false* and `1` if the specified relation is *true*. For vector types, the relational operators shall return `0` if the specified relation is *false* and `–1` (i.e. all bits set) if the specified relation is *true*. The relational operators always return `0` if either argument is not a number (NaN).

e. The equality operators equal **(==**), and not equal (**!=**) operate on built-in scalar and vector types. For built-in vector types, the operators are applied component-wise.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `char`*n* and `uchar`*n* return a `char`*n* result; vector source operands of type `short`*n* and `ushort`*n* return a `short`*n* result; vector source operands of type `int`*n*, `uint`*n* and `float`*n* return an `int`*n* result; vector source operands of type `long`*n*, `ulong`*n* and `double`*n* return a `long`*n* result.

For scalar types, the equality operators return `0` if the specified relation is *false* and return `1` if the specified relation is *true*. For vector types, the equality operators shall return `0` if the specified relation is *false* and `–1` (i.e. all bits set) if the specified relation is *true*. The equality operator equal (==) returns `0` if one or both arguments are not a number (NaN). The equality operator not equal (!=) returns `1` (for scalar source operands) or `-1` (for vector source operands) if one or both arguments are not a number (NaN).

f. The bitwise operators and (**&**), or (**|**), exclusive or (**^**), not (**~**) operate on all scalar and vector built-in types except the built-in scalar and vector float types. For vector built-in types, the operators are applied component-wise.

g. The logical operators and (**&&**), or (**||**) operate on all scalar and vector built-in types except the built-in scalar and vector float types. And (**&&**) will only evaluate the right hand operand if the left hand operand compares unequal to 0. Or (**||**) will only evaluate the right hand operand if the left hand operand compares equal to 0. For built-in vector types, the operators are applied component-wise.

The logical operator exclusive or (**^^**) is reserved.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `char`*n* and `uchar`*n* return a `char`*n* result; vector source operands of type `short`*n* and `ushort`*n* return a `short`*n* result; vector source operands of type `int`*n*, and `uint`*n* return an `int`*n* result; vector source operands of type `long`*n* and `ulong`*n* return a `long`*n* result.

For scalar types, the logical operators shall return `0` if the result of the operation is *false* and `1` if the result is *true*. For vector types, the logical operators shall return `0` if the result of the operation is *false* and `–1` (i.e. all bits set) if the result is *true*.

h. The logical unary operator not (**!**) operates on all scalar and vector built-in types except the built-in scalar and vector float types. For built-in vector types, the operators are applied component-wise.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `char`*n* and `uchar`*n* return a `char`*n* result; vector source operands of type `short`*n* and `ushort`*n* return a `short`*n* result; vector source operands of type `int`*n*, and `uint`*n* return an `int`*n* result; vector source operands of type `long`*n* and `ulong`*n* return a `long`*n* result.

For scalar types, the result of the logical unary operator is `0` if the value of its operand compares unequal to `0`, and `1` if the value of its operand compares equal to `0`. For vector types, the unary operator shall return a `0` if the value of its operand compares unequal to `0`, and `-1` (i.e. all bits set) if the value of its operand compares equal to `0`.

i. The ternary selection operator (**?:**) operates on three expressions (*exp1* **?** *exp2* **:** *exp3*). This operator evaluates the first expression *exp1*, which can be a scalar or vector result except float. If the result is a scalar value then it selects to evaluate the second expression if the result is *true*, otherwise it selects to evaluate the third expression. If the result is a vector value, then this is equivalent to calling **select**(*exp2*, *exp3*, *exp1*). The second and

third expressions can be any type, as long their types match. This resulting matching type is the type of the entire expression.

j. The operators (**~**), right-shift (**>>**), left-shift (**<<**) operate on all scalar and vector built-in types except the built-in scalar and vector float types. For built-in vector types, the operators are applied component-wise. For the right-shift (**>>**), left-shift (**<<**) operators, the rightmost operand must be a scalar if the first operand is a scalar, and the rightmost operand can be a vector or scalar if the first operand is a vector.

The result of `E1 << E2` is `E1` left-shifted by $\log_2(N)$ least significant bits in `E2` viewed as an unsigned integer value, where `N` is the number of bits used to represent the scalar data type or each component of a vector data type; vacated bits are filled with zeros.

The result of `E1 >> E2` is `E1` right-shifted by $\log_2(N)$ least significant bits in `E2` viewed as an unsigned integer value, where `N` is the number of bits used to represent the scalar data type or each component of a vector data type. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the empty bits are cleared. If `E1` has a signed type and a negative value, the empty bits are set.

k. The `sizeof` operator yields the size (in bytes) of its operand, including any padding bytes (needed for alignment, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array[22] type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

When applied to an operand that has type `char`, `uchar`, the result is `1`. When applied to an operand that has type `short`, `ushort`, or `half` the result is `2`. When applied to an operand that has type `int`, `uint` or `float`, the result is `4`. When applied to an operand that has type `long`, `ulong` or `double`, the result is `8`. When applied to an operand that is a vector type, the result is number of components * size of each scalar component. When applied to an operand that has array type, the result is the total number of bytes in the array. When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding. The `sizeof` operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member.

l. The comma (,) operator operates on expressions by returning the type and value of the right- most expression in a comma separated list of expressions. All expressions are evaluated, in order, from left to right.

m. The unary (*****) operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating

the object. If the operand has type ‘‘pointer to *type*’’, the result has type ‘‘*type*’’. If an invalid value has been assigned to the pointer, the behavior of the unary ***** operator is undefined[23].

n. The unary (**&)** operator returns the address of its operand. If the operand has type ‘‘*type*’’, the result has type ‘‘pointer to *type*’’. If the operand is the result of a unary ***** operator, neither that operator nor the **&** operator is evaluated and the result is as if both were omitted,

except that the constraints on the operators still apply and the result is not an lvalue. Similarly, if the operand is the result of a **[]** operator, neither the **&** operator nor the unary ***** that is implied by the **[]** is evaluated and the result is as if the **&** operator were removed and the **[]** operator were changed to a **+** operator. Otherwise, the result is a pointer to the object or function designated by its operand[24].

o. Assignments of values to variable names are done with the assignment operator (=), like

```
lvalue = expression
```

The assignment operator stores the value of *expression* into *lvalue*. The *expression* and *lvalue* must have the same type, or the expression must have a type in *table 6.1*, in which case an implicit conversion will be done on the expression before the assignment is done.

If *expression* is a scalar type and *lvalue* is a vector type, the scalar is promoted and/or up-converted to the type used by the vector operand (down-conversion of the scalar type is illegal and will result in a compile time error). The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

Any other desired type-conversions must be specified explicitly. L-values must be writable. Variables that are built-in types, entire structures or arrays, structure fields, l-values with the field selector (**.**) applied to select components or swizzles without repeated fields, l-values within parentheses, and l-values dereferenced with the array subscript operator (**[ ]**) are all l- values. Other binary or unary expressions, function names, swizzles with repeated fields, and constants cannot be l-values. The ternary operator (**?:**) is also not allowed as an l-value.

The order of evaluation of the operands is unspecified. If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined. Other assignment operators are the assignments add into (+=), subtract from (-=), multiply into (*****=), divide into (/=), modulus into (**%**=), left shift by (<<=), right shift by (>>=), and into (&=), inclusive or into (|=), and exclusive or into (^=).

The expression

```
lvalue op= expression
```

is equivalent to

```
lvalue = lvalue op expression
```

and the l-value and expression must satisfy the semantic requirements of both *op* and equals (=).

Vector Operations

Vector operations are component-wise. Usually, when an operator operates on a vector, it is operating independently on each component of the vector, in a component-wise fashion.

For example,

```
float4 v, u;
float  f;

v = u + f;
```

will be equivalent to

```
v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;
v.w = u.w + f;
```

And

```
float4 v, u, w;

w = v + u;
```

will be equivalent to

```
w.x = v.x + u.x;
w.y = v.y + u.y;
w.z = v.z + u.z;
w.w = v.w + u.w;
```

and likewise for most operators and all integer and floating-point vector types.

Built-in Functions

Math Functions

The vector versions of the math functions operate component-wise. The description is per- component.

The built-in math functions are not affected by the prevailing rounding mode in the calling environment, and always return the same value as they would if called with the round to nearest even rounding mode.

Below is a list of built-in math functions that can take scalar or vector arguments. We use the generic type name `gentype` to indicate that the function can take `float, float2, float4, float8, float16, double, double2, double4, double8,` or `double16` as the type for the arguments. For any specific use of a function, the actual type has to be the same for all arguments and the return type, unless otherwise specified.

gentype **acos** (gentype)
gentype **acosh** (gentype)
gentype **acospi** (gentype *x*)
gentype **asin** (gentype)
gentype **asinh** (gentype)
gentype **asinpi** (gentype *x*)
gentype **atan** (gentype *y_over_x*)
gentype **atan2** (gentype *y*, gentype *x*)
gentype **atanh** (gentype)
gentype **atanpi** (gentype *x*)
gentype **atan2pi** (gentype *y*, gentype *x*)
gentype **cbrt** (gentype)
gentype **ceil** (gentype)
gentype **copysign** (gentype *x*, gentype *y*)
gentype **cos** (gentype)
gentype **cosh** (gentype)
gentype **cospi** (gentype *x*)
gentype **erfc** (gentype)
gentype **erf** (gentype)
gentype **exp** (gentype *x*)
gentype **exp2** (gentype)
gentype **exp10** (gentype)
gentype **expm1** (gentype *x*)
gentype **fabs** (gentype)
gentype **fdim** (gentype *x*, gentype *y*)
gentype **floor** (gentype)
gentype **fma** (gentype *a*, gentype *b*, gentype *c*)

gentype **fmax** (gentype x, gentype y)
gentype **fmax** (gentype x, float y)
gentype **fmin** (gentype *x*, float *y*)
gentype **fmod** (gentype *x*, gentype *y*)
gentype **frexp** (gentype *x*, int*n* *exp)
gentype **hypot** (gentype x, gentype y)
intn **ilogb** (gentype x)
gentype **ldexp** (gentype *x*, int*n n*)
gentype **ldexp** (gentype *x*, int *n*)
gentype **lgamma** (gentype *x*)
gentype **lgamma_r** (gentype *x*, int*n* **signp*)
gentype **log** (gentype)
gentype **log2** (gentype)
gentype **log10** (gentype)
gentype **log1p** (gentype *x*)
gentype **logb** (gentype x)
gentype **maxmag** (gentype x, gentype y)
gentype **minmag** (gentype x, gentype y)
gentype **modf** (gentype x, gentype *iptr)
gentype **nan** (uint*n nancode*)
gentype **pow** (gentype *x*, gentype *y*)
gentype **pown** (gentype x, intn y)
gentype **powr** (gentype x, gentype y)
gentype **remainder** (gentype x, gentype y)
gentype **remquo** (gentype x, gentype y, intn *quo)
gentype **rint** (gentype)
gentype **rootn** (gentype *x*, int*n y*)
gentype **round** (gentype *x*)
gentype **rsqrt** (gentype)
gentype **sin** (gentype)
gentype **sincos** (gentype *x*, gentype **cosval*)
gentype **sinh** (gentype)
gentype **sinpi** (gentype *x*)
gentype **sqrt** (gentype)
gentype **tan** (gentype)
gentype **tanh** (gentype)
gentype **tanpi** (gentype *x*)
gentype **tgamma** (gentype)
gentype **trunc** (gentype)

Integer Functions

Below is a list of built-in integer functions that take scalar or vector arguments. We use the generic type name `gentype` to indicate that the function can take `char`, `char{2|4|8|16}, uchar, uchar{2|4|8|16}, short,`

`short{2|4|8|16}, ushort, ushort{2|4|8|16}, int, int{2|4|8|16}, uint, uint{2|4|8|16}, long, long{2|4|8|16} ulong,` or `ulong{2|4|8|16}` as the type for the arguments. We use the generic type name `ugentype` to refer to unsigned versions of `gentype`. For example, if `gentype` is `char4, ugentype` is `uchar4`.

For any specific use of a function, the actual type has to be the same for all arguments and the return type unless otherwise specified.

ugentype **abs** (gentype *x*)
ugentype **abs_diff** (gentype *x*, gentype *y*)
gentype **add_sat** (gentype *x*, gentype *y*)
gentype **hadd** (gentype *x*, gentype *y*)
gentype **rhadd** (gentype *x*, gentype *y*)
gentype **clamp** (gentype x, gentype minval, gentype maxval)
gentype **clz** (gentype *x*)
gentype **mad_hi** (gentype *a*, gentype *b*, gentype *c*)
gentype **mad_sat** (gentype *a*, gentype *b*, gentype *c*)
gentype **mad_hi_sat** (gentype *a*, gentype *b*, gentype *c*)
gentype **max** (gentype *x*, gentype *y*)
gentype **min** (gentype *x*, gentype *y*)
gentype **mul_hi** (gentype *x*, gentype *y*)
gentype **rotate** (gentype *v*, gentype *i*)
gentype **sub_sat** (gentype *x*, gentype *y*)
short*n* **upsample** (char*n hi*, uchar*n lo*)
ushort*n* **upsample** (uchar*n hi*, uchar*n lo*)
int*n* **upsample** (short*n hi*, ushort*n lo*)
uint*n* **upsample** (ushort*n hi*, ushort*n lo*)
long*n* **upsample** (int*n hi*, uint*n lo*)
ulongn*n* **upsample** (uint*n hi*, uint*n lo*)
intn **msum** (short2n a, short2n b, intn c)
intn **msum** (short2n a, ushort2n b, intn c)
uintm **msum** (ushort2n a, ushort2n b, uintn c)

Common Functions

Below is a list of built-in common functions. These all operate component-wise. The description is per-component. We use the generic type name `gentype` to indicate that the function can take `float, float2, float4, float8, float16, double, double2, double4, double8,` or `double16` as the type for the arguments.

The built-in common functions are implemented using the round to nearest even rounding mode.

gentype **clamp** (gentype *x*, gentype *minval*, gentype *maxval*)
gentype **clamp** (gentype *x*, float *minval*, float *maxval*)
gentype **degrees** (gentype *radians*)
gentype **max** (gentype *x*, gentype *y*)
gentype **max** (gentype *x*, float *y*)
gentype **min** (gentype *x*, gentype *y*)
gentype **min** (gentype *x*, float *y*)
gentype **mix** (gentype *x*, gentype *y*, gentype *a*)
gentype **mix** (gentype *x*, gentype *y*, float *a*)
gentype **radians** (gentype *degrees*)
gentype **step** (gentype *edge*, gentype *x*)
gentype **step** (float *edge*, gentype *x*)
genType **smoothstep** (genType *edge0*, genType *edge1*, genType *x*)
genType **smoothstep** (float *edge0*, float *edge1*, genType *x*)
gentype **sign** (gentype *x*)

Relational Functions

The relational and equality operators (<, <=, >, >=, !=, ==) can be used with scalar and vector built-in types and produce a scalar or vector signed integer result respectively as described in *section 6.3*.

The functions described in *table 6.13* can be used with built-in scalar or vector types as arguments and return a scalar vector integer result. The argment type `gentype` can be `char`, `char`*n*, `uchar`, `uchar`*n*, `short`, `short`*n*, `ushort`, `ushort`*n*, `int`, `int`*n*, `uint`, `uint`*n*, `long`, `long`*n*, `ulong`, `ulong`*n*, `float`, `float`*n*, `double`, and `double`*n*. The argument type `igentype` refers to signed integer vector types i.e. `char`, `char`*n*, `short`, `short`*n*, `int`, `int`*n*, `long` and `long`*n*. The argument type `ugentype` refers to unsigned integer vector types i.e. `uchar`, `uchar`*n*, `ushort`, `ushort`*n*, `uint`, `uint`*n*, `ulong` and `ulong`*n*.

The functions **isequal**, **isnotequal**, **isgreater**, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, **isfinite**, **isinf**, **isnan**, **isnormal**, **isordered**, **isunordered** and **signbit** shall return a 0 if the specified relation is *false* and a 1 if the specified relation is true for scalar argument types. These functions shall return a 0 if the specified relation is *false* and a –1 (i.e. all bits set) if the specified relation is *true* for vector argument types.

The relational functions **isequal**, **isgreater**, **isgreaterequal**, **isless**, **islessequal**, and **islessgreater** always return 0 if either argument is not a number (NaN). **isnotequal** returns 1 if one or both arguments are not a number (NaN) and the argument type is a scalar and returns -1 if one or both arguments are not a number (NaN) and the argument type is a vector.

int **isequal** (float *x*, float *y*)
int*n* **isequal** (float*n x*, float*n y*)

int **isnotequal** (float *x*, float *y*)
int*n* **isnotequal** (float*n x*, float*n y*)
int **isgreater** (float *x*, float *y*)
int*n* **isgreater** (float*n x*, float*n y*)
int **isgreaterequal** (float *x*, float *y*)
int*n* **isgreaterequal** (float*n x*, float*n y*)
int **isless** (float *x*, float *y*)
int*n* **isless** (float*n x*, float*n y*)
int **islessequal** (float *x*, float *y*)
int*n* **islessequal** (float*n x*, float*n y*)
int **islessgreater** (float *x*, float *y*)
int*n* **islessgreater** (float*n x*, float*n y*)
int **isfinite** (float)
int*n* **isfinite** (float*n*)
int **isinf** (float)
int*n* **isinf** (float*n*)
int **isnan** (float)
int*n* **isnan** (float*n*)
int **isnormal** (float)
int*n* **isnormal** (float*n*)
int **isordered** (float *x*, float *y*)
int*n* **isordered** (float*n x*, float*n y*)
int **isunordered** (float *x*, float *y*)
int*n* **isunordered** (float*n x*, float*n y*)
int **signbit** (float)
int*n* **signbit** (float*n*)
int **any** (igentype *x*)
int **all** (igentype *x*)
gentype **bitselect** (gentype *a*, gentype *b*, gentype *c*)
gentype **select** (gentype *a*, gentype *b*, igentype *c*)
gentype **select** (gentype *a*, gentype *b*, ugentype *c*)

Vector Data Load Store and Prefetch Functions

Below is a list of supported functions that allow you to read and write vector types from a pointer to memory. We use the generic type `gentype` to indicate the built-in data types `char, uchar, short, ushort, int, uint, long, ulong, float, double`. We use the generic type name `gentype`*n* to indicate the built-in data types `char{2|4|8|16}, uchar{2|4|8|16}, short{2|4|8|16}, ushort{2|4|8|16}, int{2|4|8|16}, uint{2|4|8|16}, long{2|4|8|16}, ulong{2|4|8|16}, float{2|4|8|16},` or `double{2|4|8|16}` as the type for the arguments unless otherwise stated. The suffix used in gentype*n* or the function name (i.e. **vload*n***, **vstore*n*** etc.) represents the number of elements in the built-in vector type (*n* = 2, 4, 8 or 16).

gentype*n* **vload*n*** (size_t *offset*, const gentype **p*)
void **vstore*n*** (gentype*n* *data*, size_t *offset*, gentype **p*)
void **prefetch** (const gentype **p*, size_t *num_elements*)

Miscellaneous Vector Functions

The OpenCL C programming language implements the following additional built-in vector functions. We use the generic type name gentype*n* (or gentype*m*) to indicate the built-in data types char{2|4|8|16}, uchar{2|4|8|16}, short{2|4|8|16}, ushort{2|4|8|16}, int{2|4|8|16}, uint{2|4|8|16}, long{2|4|8|16}, ulong{2|4|8|16}, float{2|4|8|16} or double{2|4|8|16} as the type for the arguments unless otherwise stated. We use the generic name ugentype*n* to indicate the built-in unsigned integer data types.

int **vec_step** (gentype*n* *a*)
int **vec_step**(*type*)
gentype*n* **shuffle** (gentype*m* *x*, ugentype*n* *mask*)
gentype*n* **shuffle** (gentype*m* *x*, gentype*m* *y*, ugentype*n* *mask*)

5. Summary

This paper proposes a vectorized type for C. It follows the OpenCL specification. OpenCL is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL (Open Computing Language) greatly improves speed and responsiveness for a wide spectrum of applications in numerous market categories from gaming and entertainment to scientific and medical software.

Our motivation for a standardized common vectorized syntax is to improve portability of vector programs, especially with the current interest in using vectors in parallel programming. Furthermore, it will reduce the current proliferation of existing vector programming languages, and make standard existing implementations, for which we already have extensive experience.

6. References

[IBMXLC] http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp
[GCC] http://gcc.gnu.org/onlinedocs/gcc-4.4.0/gcc/Vector-Extensions.html#Vector-Extensions
[OpenCL] http://www.khronos.org/opencl/
[OpenCLImpl] http://en.wikipedia.org/wiki/OpenCL

7. Acknowledgement

This proposal would not be possible without the work of the Khronos Group, as well as a number of people from various companies.