C++ AMP : Language and Programming Model

Version 1.0, August 2012

This specification reflects input from NVIDIA Corporation (Nvidia) and Advanced Micro Devices, Inc. (AMD).

Abstract

C++ AMP (Accelerated Massive Parallelism) is a native programming model that contains elements that span the C++ programming language and its runtime library. It provides an easy way to write programs that compile and execute on data-parallel hardware, such as graphics cards (GPUs).

The syntactic changes introduced by C++ AMP are minimal, but additional restrictions are enforced to reflect the limitations of data parallel hardware.

Data parallel algorithms are supported by the introduction of multi-dimensional array types, array operations on those types, indexing, asynchronous memory transfer, shared memory, synchronization and tiling/partitioning techniques.

1 Overview

C++ AMP is a compiler and programming model extension to C++ that enables the acceleration of C++ code on data-parallel hardware.

One example of data-parallel hardware today is the discrete graphics card (GPU), which is becoming increasingly relevant for general purpose parallel computations, in addition to its main function as a graphics accelerator. While GPUs may be tightly integrated with the CPU and can share memory space, C++ AMP programmers must remain aware that the GPU can also be physically separate from the CPU, having discrete memory address space, and incurring high cost for transferring data between CPU and GPU memory. The programmer must carefully balance the cost of this potential data transfer overhead against the computational acceleration achievable by parallel execution on the device. The programmer must also follow some basic conventions to avoid unnecessary copies on systems that have separate memory (see **Error! Reference source not found. Error! Reference source not found.** and the discard_data() method in **Error! Reference source not found.**).

Another example of data-parallel hardware is the SIMD vector instruction set, and associated registers, found in all modern processors.

For the remainder of this specification, we shall refer to the data-parallel hardware as the *accelerator*. In the few places where the distinction matters, we shall refer to a GPU or a VectorCPU.

The C++ AMP programming model gives the developer explicit control over all of the above aspects of interaction with the accelerator. The developer may explicitly manage all communication between the CPU and the accelerator, and this communication can be either synchronous or asynchronous. The data parallel computations performed on the accelerator are expressed using high-level abstractions, such as multi-dimensional arrays, high level array manipulation functions, and multi-dimensional indexing operations, all based on a large subset of the C++ programming language.

The programming model contains multiple layers, allowing developers to trade off ease-of-use with maximum performance.

C++ AMP is composed of three broad categories of functionality:

1. C++ language and compiler
    a. Kernel functions are compiled into code that is specific to the accelerator.
2. Runtime
    a. The runtime contains a C++ AMP abstraction of lower-level accelerator APIs, as well as support for multiple host threads and processors, and multiple accelerators.
    b. Asychronous execution is supported through an eventing model.
3. Programming model
    a. A set of classes describing the shape and extent of data.
    b. A set of classes that contain or refer to data used in computations
    c. A set of functions for copying data to and from accelerators
    d. A math library
    e. An atomic library
    f. A set of miscellaneous intrinsic functions

1.1 Conformance

All text in this specification falls into one of the following categories:

- *Informative: shown in this style.*
  Informative text is non-normative; for background information only; not required to be implemented in order to conform to this specification.

- *Microsoft-specific: shown in this style.*
  Microsoft-specific text is non-normative; for background information only; not required to be implemented in order to conform to this specification; explains features that are specific to the Microsoft implementation of the C++ AMP programming model. However, implementers are free to implement these feature, or any subset thereof.

- Normative: all text, unless otherwise marked (see previous categories) is normative. Normative text falls into the following two sub-categories:
  - Optional: each section of the specification that falls into this sub-category includes the suffix "(Optional)" in its title. A conforming implementation of C++ AMP may choose to support such features, or not. (Microsoft-specific portions of the text are also Optional.)
  - Required: unless otherwise stated, all Normative text falls into the sub-category of Required. A conforming implementation of C++ AMP *must* support *all* Required features.

Conforming implementations shall provide all normative features and any number of optional features. Implementations may provide additional features so long as these features are exposed in namespaces other than those listed in this specification. Implementation may provide additional language support for amp-restricted functions (section 2.1) by following the rules set forth in section 13.

The programming model utilizes Microsoft's Visual C++ syntax for *properties*. Any such property shall be considered optional. An implementation is free to use equivalent mechanisms for introducing such properties as long as they provide the same functionality of indirection to a member function as Microsoft's Visual C++ properties do.

1.2 Definitions

This section introduces terms used within the body of this specification.

- **Accelerator**
  A hardware device or capability that enables accelerated computation on data-parallel workloads. Examples include:
  - Graphics Processing Unit, or GPU, other coprocessor, accessible through the PCIe bus.
  - Graphics Processing Unit, or GPU, or other coprocessor that is integrated with a CPU on the same die.
  - SIMD units of the host node exposed through software emulation of a hardware accelerator.

- **Array**
  A dense N-dimensional data container.

- **Array View**
  A view into a contiguous piece of memory that adds array-like dimensionality.

- **Compressed texture format**.
  A format that divides a texture into blocks that allow the texture to be reduced in size by a fixed ratio; typically 4:1 or 6:1. Compressed textures are useful when perfect image/texel fidelity is not necessary but where minimizing memory storage and bandwidth are critical to application performance.

- **Extent**
  A vector of integers that describes lengths of N-dimensional array-like objects.

- **Global memory**
  On a GPU, global memory is the main off-chip memory store,
  Informative: Typcially, on current-generation GPUs, global memory is implemented in DRAM, with access times of 400-1000 cycles; the GPU clock speed is around 1 Ghz; and may or may not be cached. Global memory is accessed

in a coalesced pattern with a granularity of 128 bytes, so when accessing 4 bytes of global memory, 32 successive threads need to read the 32 successive 4-byte addresses, to be fully coalesced.

Informative: The memory space of current GPUs is typically disjoint from its host system.

- **GPGPU:** General Purpose computation on Graphics Processing Units, which is a GPU capable of running non-graphics computations.

- **GPU:** A specialized (co)processor that offloads graphics computation and rendering from the host. As GPUs have evolved, they have become increasingly able to offload non-graphics computations as well (see GPGPU).

- **Heterogenous programming**
  A workload that combines kernels executing on data-parallel compute nodes with algorithms running on CPUs.

- **Host**
  The operating system proecess and the CPU(s) that it is running on.

- **Host thread**
  The operating system thread and the CPU(s) that it is running on. A host thread may initiate a copy operation or parallel loop operation that may run on an accelerator.

- **Index**
  A vector of integers that describes an N-dimentional point in iteration space or index space.

- **Kernel; Kernel function**
  A program designed to be executed at a C++ AMP call-site. More generally, a kernel is a unit of computation that executes on an accelerator. A kernel function is a special case; it is the root of a logical call graph of functions that execute on an accelerator. A C++ analogy is that it is the "main()" function for an accelerator program

- **Perfect loop nest**
  A loop nest in which the body of each outer loop consists of a single statement that is a loop.

- **Pixel**
  A pixel, or ***pi**cture **el**ement,* represents a single element in a digital image. Typically pixels are composed of multiple color components such as a red, green and blue values. Other color representation exist, including single channel images that just represent intensity or black and white values.

- **Reference counting**
  Reference counting is a memory management technique to manage an object's lifetime. References to an object are counted and the object is kept alive as long as there is at least one reference to it. A reference counted object is destroyed when the last reference disappears.

- **SIMD unit**
  Single Instruction Multiple Data. A machine programming model where a single instruction operates over multiple pieces of data. Translating a program to use SIMD is known as vectorization. GPUs have multiple SIMD units, which are the streaming multiprocessors.
  Informative: An SSE (Nehalem, Phenom) or AVX (Sandy Bridge) or LRBni (Larrabee) vector unit is a SIMD unit or vector processor.

- **SMP**
  Symmetric Multi-Processor – standard PC multiprocessor architecure.

- **Texel**
  A texel or ***tex**ture **el**ement* represents a single element of a texture space. Texel elements are mapped to 1D, 2D or 3D surfaces during sampling, rendering and/or rasterization and end up as pixel elements on a display.

- **Texture**
  A texture is a 1, 2 or 3 dimensional logical array of texels which is optimized in hardware for spacial access using texture caches. Textures typically are used to represent image, volumetric or other visual information, although they are efficient for many data arrays which need to be optimized for spacial access or need to interpolate between adjacent elements. Textures provide virtualization of storage, whereby shader code can sample a texture object as if it contained logical elements of one type (e.g., float4) whereas the concrete physical storage of the texture is represented in terms of a second type (e.g., four 8-bit channels). This allows the application of the same shader algorithms on different types of concrete data.

- **Texture Format**
  Texture formats define the type and arrangement of the underlying bytes representing a texel value.
  Informative: Direct3D supports many types of formats, which are described under the DXGI_FORMAT enumeration.

- **Texture memory**
  Texture memory space resides in GPU memory and is cached in texture cache. A texture fetch costs one memory read from GPU memory only on a cache miss, otherwise it just costs one read from texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same scheduling unit that read texture addresses that are close together in 2D will achieve best performance. Also, it is designed for streaming fetches with a constant latency; a cache hit reduces global memory bandwidth demand but not fetch latency.

- **Thread group; Thread tile**
  A set of threads that are scheduled together, can share tile_static memory, and can participate in barrier synchronization.

  Tile_static memory
  User-managed programmable cache on streaming multiprocessors on GPUs. Shared memory is local to a multiprocessor and shared across threads executing on the same multiprocessor. Shared memory allocations per thread group will affect the total number of thread groups that are in-flight per multiprocessor
- **Tiling**
  Tiling is the partitioning of an N-dimensional dense index space (compute domain) into same sized 'tiles' which are N-dimensional rectangles with sides parallel to the coordinate axes. Tiling is essentially the process of recognizing the current thread group as being a cooperative gang of threads, with the decomposition of a global index into a local index plus a tile offset. In C++ AMP it is viewing a global index as a local index and a tile ID described by the canonical correspondence:
  compute grid ~ dispatch grid x thread group
  In particular, tiling provides the local geometry with which to take advantage of shared memory and barriers whose usage patterns enable reducing global memory accesses and coalescing of global memory access. The former is the most common use of tile_static memory.

- **Restricted function**
  A function that is declared to obey the restrictions of a particular C++ AMP subset. A function can be CPU-restricted, in which case it can run on a host CPU. A function can be amp-restricted, in which case it can run on an amp-capable accelerator, such as a GPU or VectorCPU. A function can carry more than one restriction.

1.3 Error Model

Host-side runtime library code for C++ AMP has a different error model than device-side code. For more details, examples and exception categorization see Error Handling.

Host-Side Error Model: On a host, C++ exceptions and assertions will be used to present semantic errors and hence will be categorized and listed as error states in API descriptions.

Device-Side Error Model: *Microsoft-specific: The debug_printf instrinsic is additionally supported for logging messages from within the accelerator code to the debugger output window.*

Compile-time asserts: The C++ intrinsic *static_assert* is often used to handle error states that are detectable at compile time. In this way *static_assert* is a technique for conveying static semantic errors and as such they will be categorized similar to exception types.

1.4 Programming Model

The C++ AMP programming model is factored into the following header files:

- **<amp.h>**
- **<amprt.h>**
- **<amp_math.h>**
- **<amp_graphics.h>**
- **<amp_short_vectors.h>**

Here are the types and patterns that comprise C++ AMP.

- **Indexing level (<amp.h>)**
    - index<N>
    - extent<N>
    - tiled_extent<D0,D1,D2>
    - tiled_index<D0,D1,D2>
- **Data level (<amp.h>)**
    - array<T,N>
    - array_view<T,N>, array_view<const T,N>
    - copy
    - copy_async
- **Runtime level (<amprt.h>)**
    - accelerator
    - accelerator_view
    - completion_future
- **Call-site level (<amp.h>)**
    - parallel_for_each
    - copy – various commands to move data between compute nodes
- **Kernel level (<amp.h>)**
    - tile_barrier
    - restrict() clause
    - tile_static
    - Atomic functions
- **Math functions (<amp_math.h>)**
    - Precise math functions
    - Fast math functions
- **Textures (optional, <amp_graphics.h>)**

   - texture<T,N>
   - writeonly_texture_view<T,N>
- **Short vector types (optional, <amp_short_vectors.h>)**
   - Short vector types
- **direct3d interop (optional and Microsoft-specific)**
   - Data interoperation on arrays and textures
   - Scheduling interoperation accelerators and accelerator views
   - **Direct3d intrinsic functions for clamping, bit counting, and other special arithmetic operations.**

2 C++ Language Extensions for Accelerated Computing

C++ AMP adds a closed set[1] of restriction specifiers to the C++ type system, with new syntax, as well as rules for how they behave with respect to conversion rules and overloading.

Restriction specifiers apply to function declarators only. The restriction specifiers perform the following functions:

1. They become part of the signature of the function.
2. They enforce restrictions on the content and/or behaviour of that function.
3. They may designate a particular subset of the C++ language

.

For example, an "amp" restriction would imply that a function must conform to the defined subset of C++ such that it is amenable for use on a typical GPU device.

2.1 Syntax

A new grammar production is added to represent a sequence of such restriction specifiers.

restriction-specifier-seq:
   restriction-specifier
   restriction-specifier-seq restriction-specifier

restriction-specifier:
   restrict *(restriction-seq)*

restriction-seq:
   restriction
   restriction-seq , restriction

restriction:
   amp-restriction
   cpu

amp-restriction:
   amp

The ***restrict*** keyword is a contextual keyword. The restriction specifiers contained within a *restrict* clause are not reserved words.

Multiple restrict clauses, such as *restrict(A) restrict(B)*, behave exactly the same as *restrict(A,B)*. Duplicate restrictions are allowed and behave as if the duplicates are discarded.

[1] There is no mechanism proposed here to allow developers to extend the set of restrictions.

The ***cpu*** restriction specifies that this function will be able to run on the host CPU.

If a declarator elides the restriction specifier, it behaves as if it were specified with ***restrict(cpu)***, except when a restriction specifier is determined by the surrounding context as specified in section 2.2.1. If a declarator contains a restriction specifier, then it specifies the entire set of restrictions (in other words: *restrict(amp)* means will be able to run on the amp target, need not be able to run the CPU).

2.1.1 Function Declarator Syntax

The function declarator grammar (classic & trailing return type variation) are adjusted as follows:

D1 (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *restriction-specifier-seq*$_{opt}$
exception-specification$_{opt}$ *attribute-specifier*$_{opt}$

D1 (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *restriction-specifier-seq*$_{opt}$
exception-specification$_{opt}$ *attribute-specifier*$_{opt}$ *trailing-return-type*

Restriction specifiers shall not be applied to other declarators (e.g.: arrays, pointers, references). They can be applied to all kinds of functions including free functions, static and non-static member functions, special member functions, and overloaded operators.

Examples:

```
auto grod() restrict(amp);
auto freedle() restrict(amp)-> double;

class Fred {
public:
    Fred() restrict(amp)
       : member-initializer
    { }

    Fred& operator=(const Fred&) restrict(amp);

    int kreeble(int x, int y) const restrict(amp);
    static void zot() restrict(amp);
};
```

restriction-specifier-seq$_{opt}$ applies to to all expressions between the *restriction-specifier-seq* and the end of the function-definition, lambda-expression, member-declarator, lambda-declarator or declarator.

2.1.2 Lambda Expression Syntax

The lambda expression syntax is adjusted as follows:

lambda-declarator:
(*parameter-declaration-clause*) *attribute-specifier*$_{opt}$ mutable$_{opt}$ *restriction-specifier-seq*$_{opt}$
exception-specification$_{opt}$ *trailing-return-type*$_{opt}$

When a restriction modifier is applied to a lambda expression, the behavior is as if all member functions of the generated functor are restriction-modified.

2.1.3 Type Specifiers

Restriction specifiers are not allowed anywhere in the type specifier grammar, even if it specifies a function type. For example, the following is not well-formed and will produce a syntax error:

```
typedef float FuncType(int);

restrict(cpu) FuncType* pf; // Illegal; restriction specifiers not allowed in type specifiers
```

The correct way to specify the previous example is:

```
typedef float FuncType(int) restrict(cpu);

FuncType* pf;
```

or simply

```
float (*pf)(int) restrict(cpu);
```

2.2 Meaning of Restriction Specifiers

The restriction specifiers on the declaration of a given function *F* must agree with those specified on the definition of function *F*.

Multiple restriction specifiers may be specified for a given function: the effect is that the function enforces the union of the restrictions defined by each restriction modifier.

*Informative: not for this release: It is possible to imagine two restriction specifiers that are intrinsically incompatible with each other (for example, **`pure`** and **`elemental`**). When this occurs, the compiler will produce an error.*

Refer to section 13 for treatment of versioning of restrictions

The restriction specifiers on a function become part of its signature, and thus can be used to overload.

Every expression (or sub-expression) that is evaluated in code that has multiple restriction specifiers must have the same type in the context of each restriction. It is a compile-time error if an expression can evaluate to different types under the different restriction specifiers. Function overloads should be defined with care to avoid a situation where an expression can evaluate to different types with different restrictions.

2.2.1 Function Definitions

The restriction specifiers applied to a function definition are recursively applied to all function declarators and type names defined within its body that do not have explicit restriction specifiers (i.e.: through nested classes that have member functions, and through lambdas.) For example:

```
void glorp() restrict(amp) {
    class Foo {
        void zot() {…} // "zot" is amp-restricted
    };

    auto f1 = [] (int y) { … }; // Lambda is amp-restricted

    auto f2 = [] (int y) restrict(cpu) { … }; // Lambda is cpu-restricted

    typedef int int_void_amp();  // int_void_amp is amp-restricted
    …
}
```

This also applies to the function scope of a lambda body.

2.2.2 Constructors and Destructors

Constructors can have overloads that are differentiated by restriction specifiers.

Since destructors cannot be overloaded, the destructor must contain a restriction specifier that covers the union of restrictions on all the constructors. (A destructor can achieve the same effect of overloading by calling auxiliary cleanup functions that have different restriction specifiers.)

For example:

```
class Foo {
public:
    Foo() { … }
    Foo() restrict(amp) { … }

    ~Foo() restrict(cpu,amp);
};

void UnrestrictedFunction() {
    Foo a; // calls "Foo::Foo()"
    …
    // a is destructed with "Foo::~Foo()"
}

void RestrictedFunction() restrict(amp) {
    Foo b; // calls "Foo::Foo() restrict(amp)"
    …
    // b is destructed with "Foo::~Foo()"
}

class Bar {
public:
    Bar() { … }
    Bar() restrict(amp) { … }

    ~Bar(); // error: restrict(cpu,amp) required
};
```

A virtual function declaration in a derived class will override a virtual function declaration in a base class only if the derived class function has the same restriction specifiers as the base. E.g.:

```
class Base {
public:
    virtual void foo() restrict(R1);
};

class Derived : public Base {
public:
    virtual void foo() restrict(R2); // Does not override Base::foo
};
```

(Note that C++ AMP does not support virtual functions in the current *restrict(amp)* subset.)

2.2.3 Lambda Expressions

When restriction specifiers are applied to a lambda declarator, the behavior is as if the restriction specifiers are applied to all member functions of the compiler-generated function object. For example:

```
Foo ambientVar;

auto functor = [ambientVar] (int y) restrict(amp) -> int { return y + ambientVar.z; };
```

is equivalent to:

```
Foo ambientVar;

class <lambdaName> {
```

```
public:
    <lambdaName>(const Foo& foo)
        : capturedFoo(foo)
    { }

    ~<lambdaName>() { }

    int operator()(int y) restrict(amp) { return y + capturedFoo.z; }

    const Foo& capturedFoo;
};

<lambdaName> functor;
```

2.3 Expressions Involving Restricted Functions

2.3.1 Function pointer conversions

New implicit conversion rules must be added to account for restricted function pointers (and references). Given an expression of type "pointer to R_1-function", this type can be implicitly converted to type "pointer to R_2-function" if and only if R_1 has all the restriction specifiers of R_2. Stated more intuitively, it is okay for the target function to be more restricted than the function pointer that invokes it; it's not okay for it to be less restricted. E.g.:

```
int func(int) restrict(R1,R2);
int (*pfn)(int) restrict(R1) = func;   // ok, since func(int) restrict(R1,R2) is at least R1
```

(Note that C++ AMP does not support function pointers in the current *restrict(amp)* subset.)

2.3.2 Function Overloading

Restriction specifiers become part of the function type to which they are attached. I.e.: they become part of the signature of the function. Functions can thus be overloaded by differing modifiers, and each unique set of modifiers forms a unique overload.

The restriction specifiers of a function shall not overlap with any restriction specifiers in another function within the same overload set.

```
int func(int x) restrict(cpu,amp);
int func(int x) restrict(cpu);   // error, overlaps with previous declaration
```

The target of the function call operator must resolve to an overloaded set of functions that is *at least* as restricted as the body of the calling function (see Overload Resolution). E.g.:

```
void grod();
void glorp() restrict(amp);

void foo() restrict(amp) {
    glorp(); // okay: glorp has amp restriction
    grod();  // error: grod lacks amp restriction
}
```

It is permissible for a less-restrictive call-site to call a more-restrictive function.

Compiler-generated constructors and destructors (and other special member functions) behave as if they were declared with as many restrictions as possible while avoiding ambiguities and errors. For example:

```
struct Grod {
    int a;
    int b;

    // compiler-generated default constructor: Grod() restrict(cpu,amp);
```

```
    int frool() restrict(amp) {
        return a+b;
    }

    int blarg() restrict(cpu) {
        return a*b;
    }

    // compiler-generated destructor: ~Grod() restrict(cpu,amp);
};

void d3dCaller() restrict(amp) {
    Grod g; // okay because compiler-generated default constructor is restrict(amp)

    int x = g.frool();

    // g.~Grod() called here; also okay
}

void d3dCaller() restrict(cpu) {
    Grod g; // okay because compiler-generated default constructor is restrict(cpu)

    int x = g.blarg();

    // g.~Grod() called here; also okay
}
```

The compiler must behave this way since the local usage of "Grod" in this case should not affect other potential uses of it in other restricted or unrestricted scopes.

More specifically, the compiler follows the standard C++ rules, ignoring restrictions, to determine which special member functions to generate and how to generate them. Then the restrictions are set according to the following steps:

The compiler sets the restrictions of compiler-generated destructors to the intersection of the restrictions on all of the destructors of the data members [*able to destroy all data members*] and all of the base classes' destructors [*able to call all base classes' destructors*]. If there are no such destructors, then all possible restrictions are used [*able to destroy in any context*]. However, any restriction that would result in an error is not set.

The compiler sets the restrictions of compiler-generated default constructors to the intersection of the restrictions on all of the default constructors of the member fields [*able to construct all member fields*], all of the base classes' default constructors [*able to call all base classes' default constructors*], and the destructor of the class [*able to destroy in any context constructed*]. However, any restriction that would result in an error is not set.

The compiler sets the restrictions of compiler-generated copy constructors to the intersection of the restrictions on all of the copy constructors of the member fields [*able to construct all member fields*], all of the base classes' copy constructors [*able to call all base classes' copy constructors*], and the destructor of the class [*able to destroy in any context constructed*]. However, any restriction that would result in an error is not set.

The compiler sets the restrictions of compiler-generated assignment operators to the intersection of the restrictions on all of the assignment operators of the member fields [*able to assign all member fields*] and all of the base classes' assignment operators [*able to call all base classes' assignment operators*]. However, any restriction that would result in an error is not set.

2.3.2.1 Overload Resolution

Overload resolution depends on the set of restrictions (function modifiers) in force at the call site.

```
int func(int x) restrict(A);
int func(int x) restrict(B,C);
int func(int x) restrict(D);

void foo() restrict(B) {
    int x = func(5); // calls func(int x) restrict(B,C)
    …
}
```

A call to function *F* is valid if and only if the overload set of *F* covers all the restrictions in force in the calling function. This rule can be satisfied by a single function *F* that contains all the require restrictions, or by a set of overloaded functions *F* that each specify a subset of the restrictions in force at the call site. For example:

```
void Z() restrict(amp,sse[2],cpu) { }

void Z_caller() restrict(amp,sse,cpu) {
    Z(); // okay; all restrictions available in a single function
}

void X() restrict(amp) { }
void X() restrict(sse) { }
void X() restrict(cpu) { }

void X_caller() restrict(amp,sse,cpu) {
    X(); // okay; all restrictions available in separate functions
}

void Y() restrict(amp) { }

void Y_caller() restrict(cpu,amp) {
    Y(); // error; no available Y() that satisfies CPU restriction
}
```

When a call to a restricted function is satisfied by more than one function, then the compiler must generate an as-if-runtime[3]-dispatch to the correctly restricted version.

2.3.2.2 Name Hiding

Overloading via restriction specifiers does not affect the name hiding rules. For example:

```
void foo(int x) restrict(amp) { ... }

namespace N1 {
    void foo(double d) restrict(cpu) { .... }

    void foo_caller() restrict(amp) {
        foo(10); // error; global foo() is hidden by N1::foo
    }
}
```

The name hiding rules in C++11 Section 3.3.10 state that within namespace N1, the global name "Foo" is hidden by the local name "Foo", and is *not overloaded* by it.

2.3.3 Casting

A restricted function type can be cast to a more restricted function type using a normal C-style cast or *reinterpret_cast*. (A cast is not needed when losing restrictions, only when gaining.) For example:

```
void unrestricted_func(int,int);

void restricted_caller() restrict(R) {
    ((void (*)(int,int) restrict(R))unrestricted_func)(6, 7);
```

[2] Note that "sse" is used here for illustration only, and does not imply further meaning to it in this specification.

[3] Compilers are always free to optimize this if they can determine the target statically.

```
        reinterpret_cast<(void (*)(int,int) restrict(R)>(unrestricted_func)(6, 7);
    }
```

A program which attempts to invoke a function expression after such unsafe casting can exhibit undefined behavior.

2.4 amp Restriction Modifier

The *amp* restriction modifier applies a relatively small set of restrictions that reflect the current limitations of GPU hardware and the underlying programming model.

2.4.1 Restrictions on Types

Not all types can be supported on current GPU hardware. The *amp* restriction modifier restricts functions from using unsupported types, in their function signature or in their function bodies.

We refer to the set of supported types as being *amp-compatible*. Any type referenced within an amp restriction function shall be amp-compatible. Some uses require further restrictions.

2.4.1.1 Type Qualifiers

The *volatile* type qualifier is not supported within an amp-restricted function. A variable or member qualified with volatile may not be declared or accessed in *amp* restricted code.

2.4.1.2 Fundamental Types

Of the set of C++ fundamental types only the following are supported within an amp-restricted function as *amp-compatible* types.

- *bool*
- *int, unsigned int*
- *long, unsigned long*
- *float, double*
- *void*

The representation of these types on a device running an *amp* function is identical to that of its host.

2.4.1.2.1 Floating Point Types

Floating point types behave the same in *amp* restricted code as they do in CPU code. C++ AMP imposes the additional behavioural restriction that an intermediate representation of a floating point expression shall not use higher precision than the operands demand. For example,

```
float foo() restrict(amp) {
    float f1, f2;
    …
    return f1 + f2; // "+" must be performed using "float" precision
}
```

In the above example, the expression “f1 + f2” shall not be performed using *double* (or higher) precision and then converted back to *float*.

Microsoft-specific*: This is equivalent to the Visual C++ “/fp:precise” mode. C++ AMP does not use higher-precision for intermediate representations of floating point expressions even when “/fp:fast” is specified.*

2.4.1.3 Compound Types

Pointers shall only point to *amp-compatible* types or *concurrency::array* or *concurrency::graphics::texture*. Pointers to pointers are not supported. *std::nullptr_t* type is supported and treated as a pointer type. No pointer type is considered *amp-compatible*. Pointers are only supported as local variables and/or function parameters and/or function return types.

References (lvalue and rvalue) shall refer only to *amp-compatible* types and/or *concurrency::array* and/or *concurrency::graphics::texture*. Additionally, references to pointers are supported as long as the pointer type is itself supported. Reference to *std::nullptr_t* is not allowed. No reference type is considered *amp-compatible*. References are only supported as local variables and/or function parameters and/or function return types.

concurrency::array_view and *concurrency::graphics::writeonly_texture_view* are *amp-compatible* types.

A class type (class, struct, union) is amp-compatible if

- it contains only data members whose types are *amp-compatible,* except for references to instances of classes *array* and *texture*, and
- the offset of its data members and base classes are at least four bytes aligned, and
- its data members shall not be bitfields, and
- it shall not have *virtual* base classes, and *virtual* member functions, and
- all of its base classes are *amp-compatible*.

The element type of an array shall be *amp-compatible* and four byte aligned.

Pointers to members (C++11 8.3.3) shall only refer to non-static data members.

Enumeration types shall have underlying types consisting of *int, unsigned int, long*, or *unsigned long.*

The representation of an amp-*compatible* compound type (with the exception of pointer & reference) on a device is identical to that of its host.

2.4.2 Restrictions on Function Declarators

The function declarator (C++11 8.3.5) of an amp-restricted function:

- shall not have a trailing ellipsis (…) in its parameter list
- shall have no parameters, or shall have parameters whose types are *amp-compatible*
- shall have a return type that is *void* or is *amp-compatible*
- shall not be *virtual*
- shall not have a throw specification
- shall not have *extern "C"* linkage when multiple restriction specifiers are present

2.4.3 Restrictions on Function Scopes

The function scope of an amp-restricted function may contain any valid C++ declaration, statement, or expression except for those which are specified here.

2.4.3.1 Literals

A C++ AMP program is ill-formed if the value of an integer constant or floating point constant exceeds the allowable range of any of the above types.

2.4.3.2 Primary Expressions (C++11 5.1)

An identifier or qualified identifier that refers to an object shall refer only to:

- a parameter to the function, or
- a local variable declared at a block scope within the function, or
- a non-static member of the class of which this function is a member, or

- a *static const* type that can be reduced to a integer literal and is only used as an rvalue, or
- a global *const* type that can be reduced to a integer literal and is only used as an rvalue, or
- a captured variable in a lambda expression.

2.4.3.3 Lambda Expressions

If a lambda expression appears within the body of an amp-restricted function, the *amp* modifier may be elided and the lambda is still considered an amp lambda.

A lambda expression shall not capture any context variable by reference, except for context variables of type *concurrency::array* and *concurrency::graphics::texture*.

The effective closure type must be *amp-compatible*.

2.4.3.4 Function Calls (C++11 5.2.2)

The target of a function call operator:

- shall not be a virtual function
- shall not be a pointer to a function
- shall not recursively invoke itself or any other function that is directly or indirectly recursive.

These restrictions apply to all function-like invocations including:

- object constructors & destructors
- overloaded operators, including **`new`** and **`delete`**.

2.4.3.5 Local Declarations

Local declarations shall not specify any storage class other than *register,* or *tile_static.* Variables that are not *tile_static* shall have types that are *amp-compatible*, pointers to *amp-compatible* types, or references to *amp-compatible* types.

2.4.3.5.1 tile_static Variables

A variable declared with the *tile_static* storage class can be accessed by all threads within a tile (group of threads). (The *tile_static* storage class is valid only within a *restrict(amp)* context.) The storage lifetime of a *tile_static* variable begins when the execution of a thread in a tile reaches the point of declaration, and ends when the kernel function is exited by the last thread in the tile. Each thread tile accessing the variable shall perceive to access a separate, per-tile, instance of the variable.

A *tile_static* variable declaration does not constitute a barrier (see 8.1.1). *tile_static* variables are not initialized by the compiler and assume no default initial values.

The *tile_static* storage class shall only be used to declare local (function or block scope) variables.

The type of a *tile_static* variable or array must be *amp-compatible* and shall not directly or recursively contain any concurrency containers (e.g. *concurrency::array_view*) or reference to concurrency containers.

A *tile_static* variable shall not have an initializer and no constructors or destructors will be called for it; its initial contents are undefined.

Microsoft-specific*: The Microsoft implementation of C++ AMP restricts the total size of* tile_static *memory to 32K.*

2.4.3.6 Type-Casting Restrictions

A type-cast shall not be used to convert a pointer to an integral type, nor an integral type to a pointer. This restriction applies to *reinterpret_cast* (C++11 5.2.10) as well as to C-style casts (C++11 5.4).

Casting away *const*-ness may result in a compiler warning and/or undefined behavior.

2.4.3.7 Miscellaneous Restrictions

The pointer-to-member operators `.*` and `->*` shall only be used to access pointer-to-data member objects.

Pointer arithmetic shall not be performed on pointers to *bool* values.

A pointer or reference to an amp-restricted function is not allowed. This is true even outside of an amp-restricted context.

Furthermore, an amp-restricted function shall not contain any of the following:

- *dynamic_cast* or *typeid* operators
- *goto* statements or labeled statements
- *asm* declarations
- Function *try* block, *try* blocks, *catch* blocks, or *throw*.

3 Device Modeling

3.1 The concept of a compute accelerator

A compute accelerator is a hardware capability that is optimized for data-parallel computing. An accelerator may be a device attached to a PCIe bus (such as a GPU), a device integrated on the same die as the GPU, or it might be an extended instruction set on the main CPU (such as SSE or AVX).

Informative: Some architectures might bridge these two extremes, such as AMD's Fusion or Intel's Knight's Ferry.

In the C++ AMP model, an accelerator may have private memory which is not generally accessible by the host. C++ AMP allows data to be allocated in the accelerator memory and references to this data may be manipulated on the host. It is assumed that all data accessed within a kernel must be stored in acclerator memory although some C++ AMP scenarios will implicitly make copies of data logically stored on the host.

C++ AMP has functionality for copying data between host and accelerator memories. A copy from accelerator-to-host is always a synchronization point, unless an explicit asynchronous copy is specified. In general, for optimal performance, memory content should stay on an accelerator as long as possible.

In some cases, accelerator memory and CPU memory are one and the same. And depending upon the architecture, there may never be any need to copy between the two physical locations of memory. C++ AMP provides for coding patterns that allow the C++ AMP runtime to avoid or perform copies as required.

3.2 accelerator

An *accelerator* is an abstraction of a physical data-parallel-optimized compute node. An accelerator is often a GPU, but can also be a virtual host-side entity such as the Microsoft DirectX *REF* device, or *WARP* (a CPU-side device accelerated using SSE instructions), or can refer to the CPU itself.

3.2.1 Default Accelerator

C++ AMP supports the notion of a default accelerator, an accelerator which is chosen automatically when the program does not explicitly do so.

A user may explicitly create a default accelerator object in one of two ways:

1. Invoke the default constructor:

```
accelerator def;
```

2. Use the *default_accelerator* device path:

```
accelerator def(accelerator::default_accelerator);
```

The user may also influence which accelerator is chosen as the default by calling *accelerator::set_default* prior to invoking any operation which would otherwise choose the default. Such operations include invoking *parallel_for_each* without an explicit *accelerator_view* argument, or creating an *array* not bound to an explicit *accelerator_view*, etc. Note that obtaining the default accelerator does not fix the default; this allows users to determine what the runtime's choice would be before attempting to override it.

If the user does not call *accelerator::set_default*, the default is chosen in an implementation specific manner.

Microsoft-specific*:*
The Microsoft implementation of C++ AMP uses the the following heuristic to select a default accelerator when one is not specified by a call to accelerator::set_default*:*

1. *If using the debug runtime, prefer an accelerator that supports debugging.*
2. *If the process environment variable CPPAMP_DEFAULT_ACCELERATOR is set, interpret its value as a device path and prefer the device that corresponds to it.*
3. *Otherwise, the following criteria are used to determine the 'best' accelerator:*
    a. *Prefer non-emulated devices. Among multiple non-emulated devices:*
        i. *Prefer the device with the most available memory.*
        ii. *Prefer the device which is not attached to the display.*
    b. *Among emulated devices, prefer accelerated devices such as WARP over the REF device.*

Note that the cpu_accelerator is never considered among the candidates in the above heuristic.

3.2.2 Synopsis

```
class accelerator
{
public:
    static const wchar_t default_accelerator[]; // = L"default"

    // Microsoft-specific:
    static const wchar_t direct3d_warp[];      // = L"direct3d\\warp"
    static const wchar_t direct3d_ref[];       // = L"direct3d\\ref"

    static const wchar_t cpu_accelerator[];     // = L"cpu"

    accelerator();
    explicit accelerator(const wstring& path);
    accelerator(const accelerator& other);

    static vector<accelerator> get_all();
    static bool set_default(const wstring& path);

    accelerator& operator=(const accelerator& other);

    __declspec(property(get)) wstring device_path;
    __declspec(property(get)) unsigned int version; // hiword=major, loword=minor
```

```
    __declspec(property(get)) wstring description;
    __declspec(property(get)) bool is_debug;
    __declspec(property(get)) bool is_emulated;
    __declspec(property(get)) bool has_display;
    __declspec(property(get)) bool supports_double_precision;
    __declspec(property(get)) bool supports_limited_double_precision;
    __declspec(property(get)) size_t dedicated_memory;
    __declspec(property(get)) accelerator_view default_view;

    accelerator_view create_view();
    accelerator_view create_view(queuing_mode qmode);

    bool operator==(const accelerator& other) const;
    bool operator!=(const accelerator& other) const;
};
```

`class accelerator`
Represents a physical accelerated computing device. An object of this type can be created by enumerating the available devices, or getting the default device, the reference device, or the WARP device. *Microsoft-specific:* *The WARP device may not be available on all platforms, not even all Microsoft platforms.*

3.2.3 Static Members

`static vector<accelerator> accelerator::get_all()`
Returns a std::vector of accelerator objects (in no specific order) representing all accelerators that are available, including reference accelerators and WARP accelerators if available.
Return Value:
A vector of accelerators.

`static bool set_default(const wstring& path);`	
Sets the default accelerator to the device path identified by the "path" argument. See the constructor "accelerator(const wstring& path)" for a description of the allowable path strings. This establishes a process-wide default accelerator and influences all subsequent operations that might use a default accelerator.	
Parameters	
path	The device path of the default accelerator.
Return Value:	
A Boolean flag indicating whether the default was set. If the default has already been set for this process, this value will be *`false`*, and the function will have no effect.	

3.2.4 Constructors

`accelerator()`
Constructs a new accelerator object that represents the default accelerator. This is equivalent to calling the constructor "accelerator(accelerator::default_accelerator)". The actual accelerator chosen as the default can be affected by calling "accelerator::set_default".

Parameters:
None.

```
accelerator(const wstring& path)
```

Constructs a new accelerator object that represents the physical device named by the "path" argument. If the path represents an unknown or unsupported device, an exception will be thrown.

The path can be one of the following:

1. accelerator::default_accelerator (or L"default"), which represents the path of the fastest accelerator available, as chosen by the runtime.
2. accelerator::cpu_accelerator (or L"cpu"), which represents the CPU. Note that parallel_for_each shall not be invoked over this accelerator.
3. A valid device path that uniquely identifies a hardware accelerator available on the host system.

Microsoft-specific:

4. *accelerator::direct3d_warp (or L"direct3d\\warp"), which represents the WARP accelerator*
5. *accelerator::direct3d_ref (or L"direct3d\\ref"), which represents the REF accelerator.*

Parameters:	
path	The device path of this accelerator.

```
accelerator(const accelerator& other);
```

Copy constructs an accelerator object. This function does a shallow copy with the newly created accelerator object pointing to the same underlying device as the passed accelerator parameter.

Parameters:	
other	The accelerator object to be copied.

3.2.5 Members

```
static const wchar_t default_accelerator[]
static const wchar_t direct3d_warp[]
static const wchar_t direct3d_ref[]
static const wchar_t cpu_accelerator[]
```

These are static constant string literals that represent device paths for known accelerators, or in the case of "default_accelerator", direct the runtime to choose an accelerator automatically.

default_accelerator: The string L"default" represents the default accelerator, which directs the runtime to choose the fastest accelerator available. The selection criteria are discussed in section 3.2.1 `Default Accelerator`.

cpu_accelerator: The string L"cpu" represents the host system. This accelerator is used to provide a location for system-allocated memory such as host arrays and staging arrays. It is not a valid target for accelerated computations.

Microsoft-specific:
direct3d_warp: *The string L"direct3d\\warp" represents the device path of the CPU-accelerated Warp device. On other non-direct3d platforms, this member may not exist.*

direct3d_ref: *The string L"direct3d\\ref" represents the software rasterizer, or Reference, device. This particular device is useful for debugging. On other non-direct3d platforms, this member may not exist.*

```
accelerator& operator=(const accelerator& other)
```

Assigns an accelerator object to "this" accelerator object and returns a reference to "this" object. This function does a shallow assignment with the newly created accelerator object pointing to the same underlying device as the passed accelerator parameter.

Parameters:	
other	The accelerator object to be assigned from.
Return Value:	
A reference to "this" accelerator object.	

`__declspec(property(get)) accelerator_view  default_view`	
Returns the default accelerator view associated with the accelerator. The queuing_mode of the default accelerator_view is queuing_mode_automatic.	
Return Value:	
The default **`accelerator_view`** object associated with the accelerator.	

`accelerator_view create_view(queuing_mode qmode)`	
Creates and returns a new accelerator view on the accelerator with the supplied queuing mode.	
Return Value:	
The new **`accelerator_view`** object created on the compute device.	
Parameters:	
qmode	The queuing mode of the accelerator_view to be created. See "-`Queuing Mode`".

`accelerator_view create_view()`
Creates and returns a new resource view on the accelerator. Equivalent to "create_view(queuing_mode_automatic)".
Return Value:
The new **`accelerator_view`** object created on the compute device.

`bool operator==(const accelerator& other) const`	
Compares "this" accelerator with the passed accelerator object to determine if they represent the same underlying device.	
Parameters:	
other	The accelerator object to be compared against.
Return Value:	
A boolean value indicating whether the passed accelerator object is same as "this" accelerator.	

`bool operator!=(const accelerator& other) const`	
Compares "this" accelerator with the passed accelerator object to determine if they represent different devices.	
Parameters:	
other	The accelerator object to be compared against.
Return Value:	
A boolean value indicating whether the passed accelerator object is different from "this" accelerator.	

3.2.6 Properties

The following read-only properties are part of the public interface of the class ***accelerator,*** to enable querying the accelerator characteristics:

```
__declspec(property(get)) wstring device_path
```

Returns a system-wide unique device instance path that matches the “Device Instance Path” property for the device in Device Manager, or one of the predefined path constants **cpu_accelerator**, **direct3d_warp**, or **direct3d_ref**.

```
__declspec(property(get)) wstring description
```

Returns a short textual description of the accelerator device.

```
__declspec(property(get)) unsigned int version
```

Returns a 32-bit unsigned integer representing the version number of this accelerator. The format of the integer is major.minor, where the major version number is in the high-order 16 bits, and the minor version number is in the low-order bits.

```
__declspec(property(get)) bool has_display
```

This property indicates that the accelerator may be shared by (and thus have interference from) the operating system or other system software components for rendering purposes. A C++ AMP implementation may set this property to false should such interference not be applicable for a particular accelerator.

```
__declspec(property(get)) size_t dedicated_memory
```

Returns the amount of dedicated memory (in KB) on an accelerator device. There is no guarantee that this amount of memory is actually available to use.

```
__declspec(property(get)) bool supports_double_precision
```

Returns a Boolean value indicating whether this accelerator supports double-precision (**double**) computations. When this returns true, supports_limited_double_precision also returns true.

```
__declspec(property(get)) bool supports_limited_double_precision
```

Returns a boolean value indicating whether the accelerator has limited double precision support (excludes double division, precise_math functions, int to double, double to int conversions) for a parallel_for_each kernel.

```
__declspec(property(get)) bool is_debug
```

Returns a boolean value indicating whether the accelerator supports debugging.

```
__declspec(property(get)) bool is_emulated
```

Returns a boolean value indicating whether the accelerator is emulated. This is true, for example, with the reference, WARP, and CPU accelerators.

3.3 accelerator_view

An *accelerator_view* represents a logical view of an accelerator. A single physical compute device may have many logical (isolated) accelerator views. Each accelerator has a default accelerator view and additional accelerator views may be optionally created by the user. Physical devices must potentially be shared amongst many client threads. Client threads may choose to use the same *accelerator_view* of an accelerator or each client may communicate with a compute device via an independent *accelerator_view* object for isolation from other client threads. Work submitted to an accelerator_view is guaranteed to be executed in the order that it was submitted; there are no such ordering guarantees for work submitted on different accelerator_views.

An *accelerator_view* can be created with a queuing mode of “immediate” or “automatic”. (See “Queuing Mode”).

3.3.1 Synopsis

```
class accelerator_view
{
public:
```

```
    accelerator_view() = delete;
    accelerator_view(const accelerator_view& other);

    accelerator_view& operator=(const accelerator_view& other);

    __declspec(property(get)) Concurrency::accelerator accelerator;
    __declspec(property(get)) bool is_debug;
    __declspec(property(get)) unsigned int version;
    __declspec(property(get)) queuing_mode queuing_mode;

    void flush();
    void wait();
    completion_future create_marker();

    bool operator==(const accelerator_view& other) const;
    bool operator!=(const accelerator_view& other) const;
};
```

`class accelerator_view`
Represents a logical (isolated) accelerator view of a compute accelerator. An object of this type can be obtained by calling the *default_view* property or *create_view* member functions on an accelerator object.

3.3.2 Queuing Mode

An *accelerator_view* can be created with a queuing mode in one of two states:

```
    enum queuing_mode {
        queuing_mode_immediate,
        queuing_mode_automatic
    };
```

If the queuing mode is *queuing_mode_immediate*, then any commands (such as copy or *parallel_for_each*) are sent to the corresponding accelerator before control is returned to the caller.

If the queuing mode is *queuing_mode_automatic*, then such commands are queued up on a command queue corresponding to this *accelerator_view*. There are three events that can cause queued commands to be submitted:

- Copying the contents of an array to the host or another accelerator_view results in all previous commands referencing that array resource (including the copy command itself) to be submitted for execution on the hardware.
- Calling the "accelerator_view::flush" or "accelerator_view::wait" methods.
- The IHV device driver may internally uses a heuristic to determine when commands are submitted to the hardware for execution, for example when resource limits would be exceeded without otherwise flushing the queue.

3.3.3 Constructors

An *accelerator_view* object may only be constructed using a copy or move constructor. There is no default constructor.

`accelerator_view(const accelerator_view& other)`	
Copy-constructs an accelerator_view object. This function does a shallow copy with the newly created accelerator_view object pointing to the same underlying view as the "other" parameter.	
Parameters:	
other	The accelerator_view object to be copied.

3.3.4 Members

`accelerator_view& operator=(const accelerator_view& other)`	
Assigns an accelerator_view object to "this" accelerator_view object and returns a reference to "this" object. This function does a shallow assignment with the newly created accelerator_view object pointing to the same underlying view as the passed accelerator_view parameter.	
Parameters:	
other	The accelerator_view object to be assigned from.
Return Value:	
A reference to "this" accelerator_view object.	

`__declspec(property(get)) queuing_mode queuing_mode`
Returns the queuing mode that this accelerator_view was created with. See "`Queuing Mode`".
Return Value:
The queuing mode.

`__declspec(property(get)) unsigned int version`
Returns a 32-bit unsigned integer representing the version number of this accelerator view. The format of the integer is major.minor, where the major version number is in the high-order 16 bits, and the minor version number is in the low-order bits. The version of the accelerator view is usually the same as that of the parent accelerator. *Microsoft-specific: The version may differ from the accelerator only when the accelerator_view is created from a direct3d device using the interop API.*

`__declspec(property(get)) Concurrency::accelerator accelerator`
Returns the accelerator that this accelerator_view has been created on.

`__declspec(property(get)) bool is_debug`
Returns a boolean value indicating whether the accelerator_view supports debugging through extensive error reporting. The is_debug property of the accelerator view is usually same as that of the parent accelerator. *Microsoft-specific: The is_debug value may differ from the accelerator only when the accelerator_view is created from a direct3d device using the interop API.*

`void wait()`
Performs a blocking wait for completion of all commands submitted to the accelerator view prior to calling *wait*.
Return Value:
None

`void flush()`
Sends the queued up commands in the accelerator_view to the device for execution. An accelerator_view internally maintains a buffer of commands such as data transfers between the host memory and device buffers, and kernel invocations (parallel_for_each calls)). This member function sends the commands to the device for processing. Normally, these commands are sent to the GPU automatically whenever the runtime determines

that they need to be, such as when the command buffer is full or when waiting for transfer of data from the device buffers to host memory. The *flush* member function will send the commands manually to the device. Calling this member function incurs an overhead and must be used with discretion. A typical use of this member function would be when the CPU waits for an arbitrary amount of time and would like to force the execution of queued device commands in the meantime. It can also be used to ensure that resources on the accelerator are reclaimed after all references to them have been removed. Because *flush* operates asynchronously, it can return either before or after the device finishes executing the buffered commands. However, the commands will eventually always complete. If the *queuing_mode* is *queuing_mode_immediate*, this function does nothing.
Return Value:
None

`completion_future create_marker()`
This command inserts a marker event into the accelerator_view's command queue. This marker is returned as a `completion_future` object. When all commands that were submitted prior to the marker event creation have completed, the future is ready.
Return Value:
A future which can be waited on, and will block until the current batch of commands has completed.

`bool operator==(const accelerator_view& other) const`	
Compares "this" accelerator_view with the passed accelerator_view object to determine if they represent the same underlying object.	
Parameters:	
other	The accelerator_view object to be compared against.
Return Value:	
A boolean value indicating whether the passed accelerator_view object is same as "this" accelerator_view.	

`bool operator!=(const accelerator_view& other) const`	
Compares "this" accelerator_view with the passed accelerator_view object to determine if they represent different underlying objects.	
Parameters:	
other	The accelerator_view object to be compared against.
Return Value:	
A boolean value indicating whether the passed accelerator_view object is different from "this" accelerator_view.	

3.4 Device enumeration and selection API

The physical compute devices can be enumerated or selected by calling the following static member function of the class accelerator.

3.4.1 Synopsis

```
vector<accelerator> accelerator::get_all();
```

As an example, if one wants to find an accelerator that is not emulated and is not attached to a display, one could do the following:

```
vector<accelerator> gpus = accelerator::get_all();
auto headlessIter = std::find_if(gpus.begin(), gpus.end(), [] (accelerator& accl) {
    return !accl.has_display && !accl.is_emulated;
});
```

4 Basic Data Elements

C++ AMP enables programmers to express solutions to data-parallel problems in terms of N-dimensional data aggregates and operations over them.

Fundamental to C++ AMP is the concept of an array. An array associates values in an index space with an element type. For example an array could be the set of pixels on a screen where each pixel is represented by four 32-bit values: Red, Green, Blue and Alpha. The index space would then be the screen resolution, for example all points:

```
{ {y, x} | 0 <= y < 1200, 0 <= x < 1600, x and y are integers }.
```

4.1 index<N>

Defines an N-dimensional index point; which may also be viewed as a vector based at the origin in N-space.

The index<N> type represents an N-dimensional vector of *int* which specifies a unique position in an N-dimensional space. The dimensions in the coordinate vector are ordered from most-significant to least-significant. Thus, in Cartesian 3-dimensional space, where a common convention exists that the Z dimension (plane) is most significant, the Y dimension (row) is second in significance and the X dimension (column) is the least significant, the index vector (2,0,4) represents the position at (Z=2, Y=0, X=4).

The position is relative to the origin in the N-dimensional space, and can contain negative component values.

Informative: As a scoping decision, it was decided to limit specializations of index, extent, etc. to 1, 2, and 3 dimensions. This also applies to arrays and array_views. General N-dimensional support is still provided with slightly reduced convenience.

4.1.1 Synopsis

```
template <int N>
class index {
public:
    static const int rank = N;
    typedef int value_type;

    index() restrict(amp,cpu);
    index(const index& other) restrict(amp,cpu);
    explicit index(int i0) restrict(amp,cpu); // N==1
    index(int i0, int i1) restrict(amp,cpu); // N==2
    index(int i0, int i1, int i2) restrict(amp,cpu); // N==3
    explicit index(const int components[]) restrict(amp,cpu);

    index& operator=(const index& other) restrict(amp,cpu);
```

```
    int operator[](unsigned int c) const restrict(amp,cpu);
    int& operator[](unsigned int c) restrict(amp,cpu);

    template <int N>
      friend bool operator==(const index<N>& lhs, const index<N>& rhs) restrict(amp,cpu);
    template <int N>
      friend bool operator!=(const index<N>& lhs, const index<N>& rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator+(const index<N>& lhs,
                                const index<N>& rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator-(const index<N>& lhs,
                                const index<N>& rhs) restrict(amp,cpu);

    index& operator+=(const index& rhs) restrict(amp,cpu);
    index& operator-=(const index& rhs) restrict(amp,cpu);

    template <int N>
      friend index<N> operator+(const index<N>& lhs, int rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator+(int lhs, const index<N>& rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator-(const index<N>& lhs, int rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator-(int lhs, const index<N>& rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator*(const index<N>& lhs, int rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator*(int lhs, const index<N>& rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator/(const index<N>& lhs, int rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator/(int lhs, const index<N>& rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator%(const index<N>& lhs, int rhs) restrict(amp,cpu);
    template <int N>
      friend index<N> operator%(int lhs, const index<N>& rhs) restrict(amp,cpu);

    index& operator+=(int rhs) restrict(amp,cpu);
    index& operator-=(int rhs) restrict(amp,cpu);
    index& operator*=(int rhs) restrict(amp,cpu);
    index& operator/=(int rhs) restrict(amp,cpu);
    index& operator%=(int rhs) restrict(amp,cpu);

    index& operator++() restrict(amp,cpu);
    index  operator++(int) restrict(amp,cpu);
    index& operator--() restrict(amp,cpu);
    index operator--(int) restrict(amp,cpu);
};
```

`template <int N> class index`
Represents a unique position in N-dimensional space.
Template Arguments

N	The dimensionality space into which this index applies. Special constructors are supplied for the cases where N ∈ { 1,2,3 }, but N can be any integer greater than 0.

`static const int rank = N`
A static member of **`index<N>`** that contains the rank of this index.

`typedef int value_type;`
The element type of index<N>.

4.1.2 Constructors

`index() restrict(amp,cpu)`
Default constructor. The value at each dimension is initialized to zero. Thus, "**`index<3> ix;`**" initializes the variable to the position (0,0,0).

`index(const index& other) restrict(amp,cpu)`	
Copy constructor. Constructs a new **`index<N>`** from the supplied argument "other".	
Parameters:	
other	An object of type **`index<N>`** from which to initialize this new index.

`explicit index(int i0) restrict(amp,cpu) // N==1`
`index(int i0, int i1) restrict(amp,cpu) // N==2`
`index(int i0, int i1, int i2) restrict(amp,cpu) // N==3`	

Constructs an **`index<N>`** with the coordinate values provided by $i_{0…2}$. These are specialized constructors that are only valid when the rank of the index N ∈ {1,2,3}. Invoking a specialized constructor whose argument count ≠ N will result in a compilation error.	
Parameters:	
i0 [, i1 [, i2]]	The component values of the index vector.

`explicit index(const int components[]) restrict(amp,cpu)`	
Constructs an **`index<N>`** with the coordinate values provided the array of **`int`** component values. If the coordinate array length ≠ N, the behavior is undefined. If the array value is NULL or not a valid pointer, the behavior is undefined.	
Parameters:	
components	An array of N **`int`** values.

4.1.3 Members

`index& operator=(const index& other) restrict(amp,cpu)`	
Assigns the component values of "other" to this **`index<N>`** object.	
Parameters:	
other	An object of type **`index<N>`** from which to copy into this index.
Return Value:	
Returns **`*this`**.	

`int operator[](unsigned int c) const restrict(amp,cpu)` `int& operator[](unsigned int c) restrict(amp,cpu)`	
Returns the index component value at position **c**.	
Parameters:	
c	The dimension axis whose coordinate is to be accessed.
Return Value:	
A the component value at position **c**.	

1115

1116

4.1.4 Operators

1117

```
template <int N>
  friend bool operator==(const index<N>& lhs, const index<N>& rhs) restrict(amp,cpu)
template <int N>
  friend bool operator!=(const index<N>& lhs, const index<N>& rhs) restrict(amp,cpu)
```

Compares two objects of **index<N>**.

The expression

leftIdx ⊕ rightIdx

is true if leftIdx[*i*] ⊕ rightIdx[*i*] for every *i* from 0 to N-1.

Parameters:	
lhs	The left-hand **index<N>** to be compared.
rhs	The right-hand **index<N>** to be compared.

1118

```
template <int N>
  friend index<N> operator+(const index<N>& lhs, const index<N>& rhs) restrict(amp,cpu)
template <int N>
  friend index<N> operator-(const index<N>& lhs, const index<N>& rhs) restrict(amp,cpu)
```

Binary arithmetic operations that produce a new **index<N>** that is the result of performing the corresponding pair-wise binary arithmetic operation on the elements of the operands. The *result* **index<N>** is such that for a given operator ⊕,

result[*i*] = leftIdx[*i*] ⊕ rightIdx[*i*]

for every *i* from 0 to N-1.

Parameters:	
lhs	The left-hand **index<N>** of the arithmetic operation.
rhs	The right-hand **index<N>** of the arithmetic operation.

1119

```
index& operator+=(const index& rhs) restrict(amp,cpu)
index& operator-=(const index& rhs) restrict(amp,cpu)
```

For a given operator ⊕, produces the same effect as

*(*this) = (*this) ⊕ rhs;*

The return value is "*this".

Parameters:	
rhs	The right-hand **index<N>** of the arithmetic operation.

1120

1121

```
template <int N>
  friend index<N> operator+(const index<N>& idx, int value) restrict(amp,cpu)
template <int N>
  friend index<N> operator+(int value, const index<N>& idx) restrict(amp,cpu)
template <int N>
  friend index<N> operator-(const index<N>& idx, int value) restrict(amp,cpu)
template <int N>
  friend index<N> operator-(int value, const index<N>& idx) restrict(amp,cpu)
template <int N>
  friend index<N> operator*(const index<N>& idx, int value) restrict(amp,cpu)
template <int N>
  friend index<N> operator*(int value, const index<N>& idx) restrict(amp,cpu)
template <int N>
  friend index<N> operator/(const index<N>& idx, int value) restrict(amp,cpu)
template <int N>
  friend index<N> operator/(int value, const index<N>& idx) restrict(amp,cpu)
template <int N>
  friend index<N> operator%(const index<N>& idx, int value) restrict(amp,cpu)
template <int N>
```

`friend index<N> operator%(int value, const index<N>& idx) restrict(amp,cpu)`	
Binary arithmetic operations that produce a new **`index<N>`** that is the result of performing the corresponding binary arithmetic operation on the elements of the index operands. The *result* **`index<N>`** is such that for a given operator ⊕, *result*[*i*] = idx[*i*] ⊕ value or *result[i]* = value ⊕ idx[*i*] for every *i* from 0 to N-1.	
Parameters:	
idx	The **`index<N>`** operand
value	The integer operand

`index& operator+=(int value) restrict(amp,cpu)`
`index& operator-=(int value) restrict(amp,cpu)`
`index& operator*=(int value) restrict(amp,cpu)`
`index& operator/=(int value) restrict(amp,cpu)`
`index& operator%=(int value) restrict(amp,cpu)`	

For a given operator ⊕, produces the same effect as *(*this) = (*this) ⊕ value;* The return value is "*this".	
Parameters:	
value	The right-hand **`int`** of the arithmetic operation.

`index& operator++() restrict(amp,cpu)`
`index  operator++(int) restrict(amp,cpu)`
`index& operator--() restrict(amp,cpu)`
`index  operator--(int) restrict(amp,cpu)`

For a given operator ⊕, produces the same effect as *(*this) = (*this) ⊕ 1;* For prefix increment and decrement, the return value is "*this". Otherwise a new index<N> is returned.

4.2 extent<N>

The extent<N> type represents an N-dimensional vector of *int* which specifies the bounds of an N-dimensional space with an origin of 0. The values in the coordinate vector are ordered from most-significant to least-significant. Thus, in Cartesian 3-dimensional space, where a common convention exists that the Z dimension (plane) is most significant, the Y dimension (row) is second in significance and the X dimension (column) is the least significant, the extent vector (7,5,3) represents a space where the Z coordinate ranges from 0 to 6, the Y coordinate ranges from 0 to 4, and the X coordinate ranges from 0 to 2.

4.2.1 Synopsis

```
template <int N>
class extent {
public:
    static const int rank = N;
    typedef int value_type;

    extent() restrict(amp,cpu);
    extent(const extent& other) restrict(amp,cpu);
    explicit extent(int e0) restrict(amp,cpu); // N==1
    extent(int e0, int e1) restrict(amp,cpu); // N==2
    extent(int e0, int e1, int e2) restrict(amp,cpu); // N==3
    explicit extent(const int components[]) restrict(amp,cpu);

```

```
extent& operator=(const extent& other) restrict(amp,cpu);

int operator[](unsigned int c) const restrict(amp,cpu);
int& operator[](unsigned int c) restrict(amp,cpu);

unsigned int size() const restrict(amp,cpu);

bool contains(const index<N>& idx) const restrict(amp,cpu);

template <int D0>                tiled_extent<D0> tile() const;
template <int D0, int D1>         tiled_extent<D0,D1> tile() const;
template <int D0, int D1, int D2> tiled_extent<D0,D1,D2> tile() const;

extent operator+(const index<N>& idx) restrict(amp,cpu);
extent operator-(const index<N>& idx) restrict(amp,cpu);


extent& operator+=(const index<N>& idx) restrict(amp,cpu);
extent& operator-=(const index<N>& idx) restrict(amp,cpu);
extent& operator+=(const extent<N>& idx) restrict(amp,cpu);
extent& operator-=(const extent<N>& idx) restrict(amp,cpu);

template <int N>
  friend extent<N> operator+(const extent<N>& lhs,
                             const extent<N>& rhs) restrict(amp,cpu);
template <int N>
  friend index<N> operator-(const extent<N>& lhs,
                            const extent<N>& rhs) restrict(amp,cpu);

template <int N>
  friend bool operator==(const extent<N>& lhs, const extent<N>& rhs) restrict(amp,cpu);
template <int N>
  friend bool operator!=(const extent<N>& lhs, const extent<N>& rhs) restrict(amp,cpu);

template <int N>
  friend extent<N> operator+(const extent<N>& lhs, int rhs) restrict(amp,cpu);
template <int N>
  friend extent<N> operator+(int lhs, const extent<N>& rhs) restrict(amp,cpu);
template <int N>
  friend extent<N> operator-(const extent<N>& lhs, int rhs) restrict(amp,cpu);
template <int N>
  friend extent<N> operator-(int lhs, const extent<N>& rhs) restrict(amp,cpu);
template <int N>
  friend extent<N> operator*(const extent<N>& lhs, int rhs) restrict(amp,cpu);
template <int N>
  friend extent<N> operator*(int lhs, const extent<N>& rhs) restrict(amp,cpu);
template <int N>
  friend extent<N> operator/(const extent<N>& lhs, int rhs) restrict(amp,cpu);
template <int N>
  friend extent<N> operator/(int lhs, const extent<N>& rhs) restrict(amp,cpu);
template <int N>
  friend extent<N> operator%(const extent<N>& lhs, int rhs) restrict(amp,cpu);
template <int N>
  friend extent<N> operator%(int lhs, const extent<N>& rhs) restrict(amp,cpu);

extent& operator+=(int rhs) restrict(amp,cpu);
extent& operator-=(int rhs) restrict(amp,cpu);
extent& operator*=(int rhs) restrict(amp,cpu);
```

```
    extent& operator/=(int rhs) restrict(amp,cpu);
    extent& operator%=(int rhs) restrict(amp,cpu);

    extent& operator++() restrict(amp,cpu);
    extent operator++(int) restrict(amp,cpu);
    extent& operator--() restrict(amp,cpu);
    extent operator--(int) restrict(amp,cpu);
};
```

`template <int N> class extent`	
Represents a unique position in N-dimensional space.	
Template Arguments	
N	The dimension to this extent applies. Special constructors are supplied for the cases where N ∊ { 1,2,3 }, but N can be any integer greater than or equal to 1. (Microsoft-specific: N can not exceed 128.)

`static const int rank = N`
A static member of **`extent<N>`** that contains the rank of this extent.

`typedef int value_type;`
The element type of **`extent<N>`**.

4.2.2 Constructors

`extent() restrict(amp,cpu);`
Default constructor. The value at each dimension is initialized to zero. Thus, "**`extent<3> ix;`**" initializes the variable to the position (0,0,0).
Parameters:
None.

`extent(const extent& other) restrict(amp,cpu)`	
Copy constructor. Constructs a new **`extent<N>`** from the supplied argument ix.	
Parameters:	
other	An object of type **`extent<N>`** from which to initialize this new extent.

`explicit extent(int e0) restrict(amp,cpu) // N==1`
`extent(int e0, int e1) restrict(amp,cpu) // N==2`
`extent(int e0, int e1, int e2) restrict(amp,cpu) // N==3`	

Constructs an **`extent<N>`** with the coordinate values provided by $e_{0..2}$. These are specialized constructors that are only valid when the rank of the extent N ∊ {1,2,3}. Invoking a specialized constructor whose argument count ≠ N will result in a compilation error.	
Parameters:	
e0 [, e1 [, e2]]	The component values of the extent vector.

`explicit extent(const int components[]) restrict(amp,cpu);`	
Constructs an **`extent<N>`** with the coordinate values provided the array of **`int`** component values. If the coordinate array length ≠ N, the behavior is undefined. If the array value is NULL or not a valid pointer, the behavior is undefined.	
Parameters:	
components	An array of N **`int`** values.

4.2.3 Members

`extent& operator=(const extent& other) restrict(amp,cpu)`	
Assigns the component values of "other" to this **extent<N>** object.	
Parameters:	
other	An object of type **extent<N>** from which to copy into this extent.
Return Value:	
Returns ***this**.	

`int operator[](unsigned int c) const restrict(amp,cpu)` `int& operator[](unsigned int c) restrict(amp,cpu)`	
Returns the extent component value at position **c**.	
Parameters:	
c	The dimension axis whose coordinate is to be accessed.
Return Value:	
A the component value at position **c**.	

`bool contains(const index<N>& idx) const restrict(amp,cpu)`	
Tests whether the index "idx" is properly contained within this extent (with an assumed origin of zero).	
Parameters:	
idx	An object of type **index<N>**
Return Value:	
Returns **true** if the "idx" is contained within the space defined by this extent (with an assumed origin of zero).	

`unsigned int size() const restrict(amp,cpu)`
This member function returns the total linear size of this extent<N> (in units of elements), which is computed as: extent[0] * extent[1] … * extent[N-1]

`template <int D0> tiled_extent<D0> tile() const restrict(amp,cpu)`
`template <int D0, int D1> tiled_extent<D0,D1> tile() const restrict(amp,cpu)`
`template <int D0, int D1, int D2> tiled_extent<D0,D1,D2> tile() const restrict(amp,cpu)`

Produces a **tiled_extent** object with the tile extents given by D0, D1, and D2. **tile<D0,D1,D2>()** is only supported on **extent<3>**. It will produce a compile-time error if used on an **extent** where N ≠ 3. **tile<D0,D1>()** is only supported on **extent <2>**. It will produce a compile-time error if used on an **extent** where N ≠ 2. **tile<D0>()** is only supported on **extent <1>**. It will produce a compile-time error if used on an **extent** where N ≠ 1.

4.2.4 Operators

`template <int N>`
`  friend bool operator==(const extent<N>& lhs, const extent<N>& rhs) restrict(amp,cpu)`
`template <int N>`
`  friend bool operator!=(const extent<N>& lhs, const extent<N>& rhs) restrict(amp,cpu)`	

Compares two objects of **extent<N>**. The expression leftExt ⊕ rightExt is true if leftExt[*i*] ⊕ rightExt[*i*] for every *i* from 0 to N-1.	
Parameters:	
lhs	The left-hand **extent<N>** to be compared.
rhs	The right-hand **extent<N>** to be compared.

`extent<N> operator+(const index<N>& idx) restrict(amp,cpu)` `extent<N> operator-(const index<N>& idx) restrict(amp,cpu)`
Adds (or subtracts) an object of type **index<N>** from this extent to form a new extent. The *result* **extent<N>** is such that for a given operator ⊕,

result[*i*] = this[*i*] ⊕ idx[*i*]

Parameters:

idx	The right-hand **index<N>** to be added or subtracted.

```
template <int N>
  friend extent<N> operator+(const extent<N>& ext, int value) restrict(amp,cpu)
template <int N>
  friend extent<N> operator+(int value, const extent<N>& ext) restrict(amp,cpu)
template <int N>
  friend extent<N> operator-(const extent<N>& ext, int value) restrict(amp,cpu)
template <int N>
  friend extent<N> operator-(int value, const extent<N>& ext) restrict(amp,cpu)
template <int N>
  friend extent<N> operator*(const extent<N>& ext, int value) restrict(amp,cpu)
template <int N>
  friend extent<N> operator*(int value, const extent<N>& ext) restrict(amp,cpu)
template <int N>
  friend extent<N> operator/(const extent<N>& ext, int value) restrict(amp,cpu)
template <int N>
  friend extent<N> operator/(int value, const extent<N>& ext) restrict(amp,cpu)
template <int N>
  friend extent<N> operator%(const extent<N>& ext, int value) restrict(amp,cpu)
template <int N>
  friend extent<N> operator%(int value, const extent<N>& ext) restrict(amp,cpu)
```

Binary arithmetic operations that produce a new **extent<N>** that is the result of performing the corresponding binary arithmetic operation on the elements of the extent operands. The *result* **extent<N>** is such that for a given operator ⊕,

result[*i*] = ext[*i*] ⊕ value

or

result[i] = value ⊕ ext[*i*]

for every *i* from 0 to N-1.

Parameters:

ext	The **extent<N>** operand
value	The integer operand

```
extent& operator+=(int value) restrict(amp,cpu)
extent& operator-=(int value) restrict(amp,cpu)
extent& operator*=(int value) restrict(amp,cpu)
extent& operator/=(int value) restrict(amp,cpu)
extent& operator%=(int value) restrict(amp,cpu)
```

For a given operator ⊕, produces the same effect as

*(*this) = (*this) ⊕ value*

The return value is "*this".

Parameters:

Value	The right-hand **int** of the arithmetic operation.

```
extent& operator++() restrict(amp,cpu)
extent  operator++(int) restrict(amp,cpu)
extent& operator--() restrict(amp,cpu)
extent  operator--(int) restrict(amp,cpu)
```

For a given operator ⊕, produces the same effect as

*(*this) = (*this) ⊕ 1*

For prefix increment and decrement, the return value is "*this". Otherwise a new **extent<N>** is returned.

4.3 tiled_extent<D0,D1,D2>

A *tiled_extent* is an extent of 1 to 3 dimensions which also subdivides the index space into 1-, 2-, or 3-dimensional tiles. It has three specialized forms: *tiled_extent<D0>*, *tiled_extent<D0,D1>*, and *tiled_extent<D0,D1,D2>*, where $D_{0\text{-}2}$ specify the positive length of the tile along each dimension, with *D0* being the most-significant dimension and *D2* being the least-significant. Partial template specializations are provided to represent 2-D and 1-D tiled extents.

A *tiled_extent* can be formed from an extent by calling *extent<N>::tile<D0,D1,D2>()* or one of the other two specializations of *extent<N>::tile()*.

A *tiled_extent* inherits from *extent*, thus all public members of *extent* are available on *tiled_extent*.

4.3.1 Synopsis

```
template <int D0, int D1=0, int D2=0>
class tiled_extent : public extent<3>
{
public:
    static const int rank = 3;

    tiled_extent() restrict(amp,cpu);
    tiled_extent(const tiled_extent& other) restrict(amp,cpu);
    tiled_extent(const extent<3>& extent) restrict(amp,cpu);

    tiled_extent& operator=(const tiled_extent& other) restrict(amp,cpu);

    tiled_extent pad() const restrict(amp,cpu);
    tiled_extent truncate() const restrict(amp,cpu);

    __declspec(property(get)) extent<3> tile_extent;

    static const int tile_dim0 = D0;
    static const int tile_dim1 = D1;
    static const int tile_dim2 = D2;

    friend bool operator==(const tiled_extent& lhs,
                           const tiled_extent& rhs) restrict(amp,cpu);
    friend bool operator!=(const tiled_extent& lhs,
                           const tiled_extent& rhs) restrict(amp,cpu);
};


template <int D0, int D1>
class tiled_extent<D0,D1,0> : public extent<2>
{
public:
    static const int rank = 2;

    tiled_extent() restrict(amp,cpu);
    tiled_extent(const tiled_extent& other) restrict(amp,cpu);
    tiled_extent(const extent<2>& extent) restrict(amp,cpu);
```

```
    tiled_extent& operator=(const tiled_extent& other) restrict(amp,cpu);

    tiled_extent pad() const restrict(amp,cpu);
    tiled_extent truncate() const restrict(amp,cpu);

    __declspec(property(get)) extent<2> tile_extent;

    static const int tile_dim0 = D0;
    static const int tile_dim1 = D1;

    friend bool operator==(const tiled_extent& lhs,
                           const tiled_extent& rhs) restrict(amp,cpu);
    friend bool operator!=(const tiled_extent& lhs,
                           const tiled_extent& rhs) restrict(amp,cpu);
};

template <int D0>
class tiled_extent<D0,0,0> : public extent<1>
{
public:
    static const int rank = 1;

    tiled_extent() restrict(amp,cpu);
    tiled_extent(const tiled_extent& other) restrict(amp,cpu);
    tiled_extent(const extent<1>& extent) restrict(amp,cpu);

    tiled_extent& operator=(const tiled_extent& other) restrict(amp,cpu);

    tiled_extent pad() const restrict(amp,cpu);
    tiled_extent truncate() const restrict(amp,cpu);

    __declspec(property(get)) extent<1> tile_extent;

    static const int tile_dim0 = D0;

    friend bool operator==(const tiled_extent& lhs,
                           const tiled_extent& rhs) restrict(amp,cpu);
    friend bool operator!=(const tiled_extent& lhs,
                           const tiled_extent& rhs) restrict(amp,cpu);
};
```

```
template <int D0, int D1=0, int D2=0> class tiled_extent
template <int D0, int D1>             class tiled_extent<D0,D1,0>
template <int D0>                     class tiled_extent<D0,0,0>
```

Represents an extent subdivided into 1-, 2-, or 3-dimensional tiles.

Template Arguments

D0, D1, D2	The length of the tile in each specified dimension, where D0 is the most-significant dimension and D2 is the least-significant.

```
static const int rank = N
```

A static member of **`tiled_extent`** that contains the rank of this tiled extent, and is either 1, 2, or 3 depending on the specialization used.

4.3.2 Constructors

`tiled_extent() restrict(amp,cpu)`
Default constructor.   The origin and extent is default-constructed and thus zero.
Parameters:
None.

`tiled_extent(const tiled_extent& other) restrict(amp,cpu)`	
Copy constructor. Constructs a new **`tiled_extent`** from the supplied argument "other".	
Parameters:	
other	An object of type **`tiled_extent`** from which to initialize this new extent.

`tiled_extent(const extent<N>& extent) restrict(amp,cpu)`	
Constructs a **`tiled_extent<N>`** with the extent "extent". The origin is default-constructed and thus zero. Notice that this constructor allows implicit conversions from extent<N> to tiled_extent<N>.	
Parameters:	
extent	The extent of this tiled_extent

4.3.3 Members

`tiled_extent& operator=(const tiled_extent& other) restrict(amp,cpu)`	
Assigns the component values of "other" to this **`tiled_extent<N>`** object.	
Parameters:	
Other	An object of type **`tiled_extent<N>`** from which to copy into this.
Return Value:	
Returns **`*this`**.	

`tiled_extent pad() const restrict(amp,cpu)`
Returns a new tiled_extent with the extents adjusted up to be evenly divisible by the tile dimensions. The origin of the new tiled_extent is the same as the origin of this one.

`tiled_extent truncate() const restrict(amp,cpu)`
Returns a new tiled_extent with the extents adjusted down to be evenly divisible by the tile dimensions. The origin of the new tiled_extent is the same as the origin of this one.

`__declspec(property(get)) extent<N> tile_extent`
Returns an instance of an **`extent<N>`** that captures the values of the **`tiled_extent`** template arguments D0, D1, and D2. For example:

```
tiled_extent<64,16,4> tg;
extent<3> myTileExtent = tg.tile_extent;
assert(myTileExtent[0] == 64);
assert(myTileExtent[1] == 16);
assert(myTileExtent[2] == 4);
```

`static const int tile_dim0`
`static const int tile_dim1`
`static const int tile_dim2`

These constants allow access to the template arguments of **`tiled_extent`**.

4.3.4 Operators

```
friend bool operator==(const tiled_extent& lhs,
                       const tiled_extent& rhs) restrict(amp,cpu)
```

```
friend bool operator!=(const tiled_extent& lhs,
                       const tiled_extent& rhs) restrict(amp,cpu)
```

Compares two objects of **`tiled_extent<N>`**. The expression lhs ⊕ rhs is true if lhs.extent ⊕ rhs.extent and lhs.origin ⊕ rhs.origin.	
Parameters:	
lhs	The left-hand **`tiled_extent`** to be compared.
rhs	The right-hand **`tiled_extent`** to be compared.

4.4 tiled_index<D0,D1,D2>

A *tiled_index* is a set of indices of 1 to 3 dimensions which have been subdivided into 1-, 2-, or 3-dimensional tiles in a *tiled_extent*. It has three specialized forms: *tiled_index<D0>*, *tiled_index<D0,D1>*, and *tiled_index<D0,D1,D2>*, where $D_{0\text{-}2}$ specify the length of the tile along each dimension, with *D0* being the most-significant dimension and *D2* being the least-significant. Partial template specializations are provided to represent 2-D and 1-D tiled indices.

A *tiled_index* is implicitly convertible to an *index<N>*, where the implicit index represents the global index.

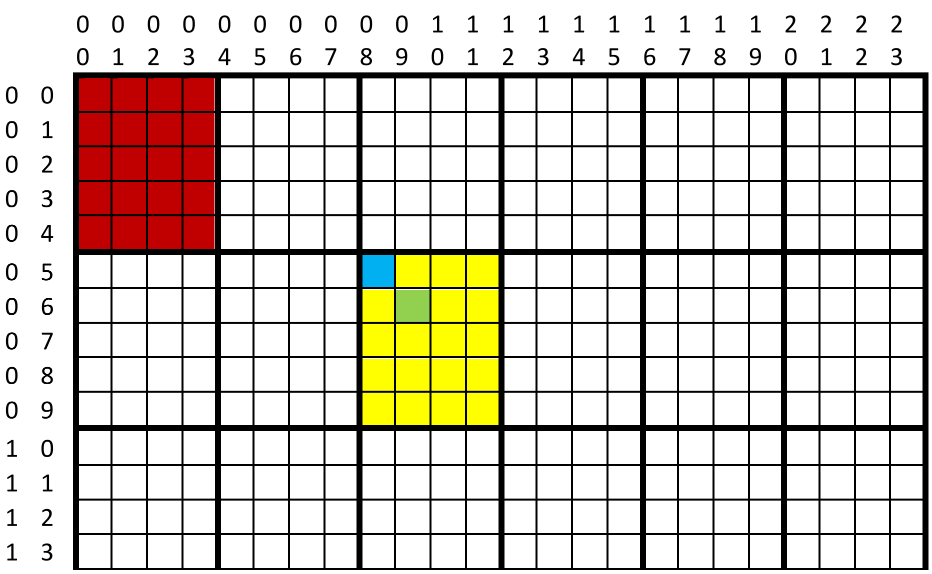
A *tiled_index* contains 4 member indices which are related to each other mathematically and help the user to pinpoint a global index to an index within a tiled space.

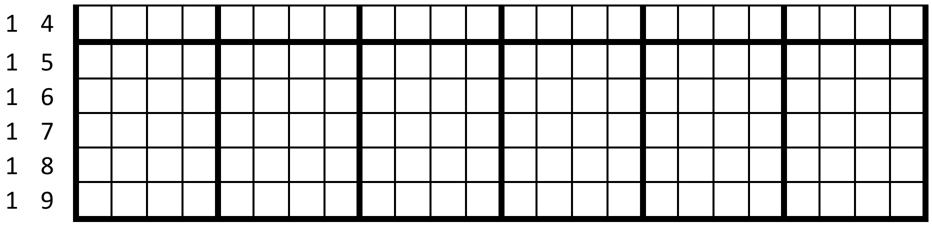
A *tiled_index* contains a global index into an extent space. The other indices obey the following relations:

.local ≡ .global % (D0,D1,D2)
.tile ≡ .global / (D0,D1,D2)
.tile_origin ≡ .global - .local

This is shown visually in the following example:

```
parallel_for_each(extent<2>(20,24).tile<5,4>(),
                  [&](tiled_index<5,4> ti) { /* ... */ });
```





1. Each cell in the diagram represents one thread which is scheduled by the *parallel_for_each* call. We see that, as with the non-tiled *parallel_for_each*, the number of threads scheduled is given by the extent parameter to the *parallel_for_each* call.
2. Using vector notation, we see that the total number of tiles scheduled is <20,24> / <5,4> = <4,6>, which we see in the above diagram as 4 tiles along the vertical axis, and 6 tiles along the horizontal axis.
3. The tile in red is tile number <0,0>. The tile in yellow is tile number <1,2>.
4. The thread in blue:
   a. has a global id of <5,8>
   b. Has a local id <0,0> within its tile. i.e., it lies on the origin of the tile.
5. The thread in green:
   a. has a global id of <6,9>
   b. has a local id of <1,1> within its tile
   c. The blue thread (number <5,8>) is the green thread's tile origin.

4.4.1 Synopsis

```
template <int D0, int D1=0, int D2=0>
class tiled_index
{
public:
    static const int rank = 3;

    const index<3> global;
    const index<3> local;
    const index<3> tile;
    const index<3> tile_origin;
    const tile_barrier barrier;

    tiled_index(const index<3>& global,
                const index<3> local,
                const index<3> tile,
                const index<3> tile_origin,
                const tile_barrier& barrier) restrict(amp,cpu);
    tiled_index(const tiled_index& other) restrict(amp,cpu);

    operator const index<3>() const restrict(amp,cpu);

    __declspec(property(get)) extent<3> tile_extent;

    static const int tile_dim0 = D0;
    static const int tile_dim1 = D1;
    static const int tile_dim2 = D2;
};

template <int D0, int D1>
class tiled_index<D0,D1,0>
{
```

```
public:
    static const int rank = 2;

    const index<2> global;
    const index<2> local;
    const index<2> tile;
    const index<2> tile_origin;
    const tile_barrier barrier;

    tiled_index(const index<2>& global,
                const index<2> local,
                const index<2> tile,
                const index<2> tile_origin,
                const tile_barrier& barrier) restrict(amp,cpu);
    tiled_index(const tiled_index& other) restrict(amp,cpu);

    operator const index<2>() const restrict(amp,cpu);


    __declspec(property(get)) extent<2> tile_extent;

    static const int tile_dim0 = D0;
    static const int tile_dim1 = D1;
};

template <int D0>
class tiled_index<D0,0,0>
{
public:
    static const int rank = 1;

    const index<1> global;
    const index<1> local;
    const index<1> tile;
    const index<1> tile_origin;
    const tile_barrier barrier;

    tiled_index(const index<1>& global,
                const index<1> local,
                const index<1> tile,
                const index<1> tile_origin,
                const tile_barrier& barrier) restrict(amp,cpu);
    tiled_index(const tiled_index& other) restrict(amp,cpu);

    operator const index<1>() const restrict(amp,cpu);

    __declspec(property(get)) extent<1> tile_extent;

    static const int tile_dim0 = D0;
};
```

```
template <int D0, int D1=0, int D2=0> class tiled_index
template <int D0, int D1>             class tiled_index<D0,D1,0>
template <int D0 >                    class tiled_index<D0,0,0>
```

Represents a set of related indices subdivided into 1-, 2-, or 3-dimensional tiles.

Template Arguments	
D0, D1, D2	The length of the tile in each specified dimension, where D0 is the most-significant dimension and D2 is the least-significant.

`static const int rank = N`
A static member of **`tiled_index`** that contains the rank of this tiled extent, and is either 1, 2, or 3 depending on the specialization used.

4.4.2 Constructors

The tiled_index class has no default constructor.

```
tiled_index(const index<N>& global,
            const index<N>& local,
            const index<N>& tile,
            const index<N>& tile_origin,
            const tile_barrier& barrier) restrict(amp,cpu)
```

Construct a new tiled_index out of the constituent indices. Note that it is permissible to create a tiled_index instance for which the geometric identities which are guaranteed for system-created tiled indices, which are passed as a kernel parameter to the tiled overloads of parallel_for_each, do not hold. In such cases, it is up to the application to assign application-specific meaning to the member indices of the instance.	
Parameters:	
global	An object of type **`index<N>`** which is taken to be the global index of this tile.
local	An object of type **`index<N>`** which is taken to be the local index within this tile.
tile	An object of type **`index<N>`** which is taken to be the coordinates of the current tile.
tile_origin	An object of type **`index<N>`** which is taken to be the global index of the top-left corner of the tile.
barrier	An object of type **`tile_barrier`**.

`tiled_index(const tiled_index& other) restrict(amp,cpu)`	
Copy constructor. Constructs a new **`tiled_index`** from the supplied argument "other".	
Parameters:	
other	An object of type **`tiled_index`** from which to initialize this.

4.4.3 Members

`const index<N> global`
An index of rank 1, 2, or 3 that represents the global index within an extent.

`const index<N> local`
An index of rank 1, 2, or 3 that represents the relative index within the current tile of a tiled extent.

`const index<N> tile`
An index of rank 1, 2, or 3 that represents the coordinates of the current tile of a tiled extent.

`const index<N> tile_origin`
An index of rank 1, 2, or 3 that represents the global coordinates of the origin of the current tile within a tiled extent.

`const tile_barrier barrier`
An object which represents a barrier within the current tile of threads.

`operator const index<N>() const restrict(amp,cpu)`
Implicit conversion operator that converts a tiled_index<D0,D1,D2> into an index<N>. The implicit conversion converts to the .global index member.

`__declspec(property(get)) extent<N> tile_extent`

Returns an instance of an **`extent<N>`** that captures the values of the **`tiled_index`** template arguments D0, D1, and D2. For example:
 	
`index<3> zero;`
`tiled_index<64,16,4> ti(index<3>(256,256,256), zero, zero, zero, mybarrier);`
`extent<3> myTileExtent = ti.tile_extent;`
`assert(myTileExtent.tile_dim0 == 64);`
`assert(myTileExtent.tile_dim1 == 16);`
`assert(myTileExtent.tile_dim2 == 4);`

`static const int tile_dim0`
`static const int tile_dim1`
`static const int tile_dim2`

These constants allow access to the template arguments of **`tiled_index`**.

4.5 tile_barrier

The *tile_barrier* class is a capability class that is only creatable by the system, and passed to a tiled *parallel_for_each* function object as part of the *tiled_index* parameter. It provides member functions, such as *wait*, whose purpose is to synchronize execution of threads running within the thread tile.

A call to *wait* shall not occur in non-uniform code within a thread tile. Section 8 defines uniformity and lack thereof formally.

4.5.1 Synopsis

```
class tile_barrier
{
public:
    tile_barrier(const tile_barrier& other) restrict(amp,cpu);

    void wait() const restrict(amp);
    void wait_with_all_memory_fence() const restrict(amp);
    void wait_with_global_memory_fence() const restrict(amp);
    void wait_with_tile_static_memory_fence() const restrict(amp);
};
```

4.5.2 Constructors

The tile_barrier class does not have a public default constructor, only a copy-constructor.

`tile_barrier(const tile_barrier& other) restrict(amp,cpu)`	
Copy constructor. Constructs a new **`tile_barrier`** from the supplied argument "other".	
Parameters:	
other	An object of type **`tile_barrier`** from which to initialize this.

4.5.3 Members

The tile_barrier class does not have an assignment operator. Section 8 provides a complete description of the C++ AMP memory model, of which class *tile_barrier* is an important part.

`void wait() const restrict(amp)`
Blocks execution of all threads in the thread tile until all threads in the tile have reached this call. Establishes a memory fence on all tile_static and global memory operations executed by the threads in the tile such that all memory operations issued prior to hitting the barrier are visible to all other threads after the barrier has completed and none of the memory operations occurring after the barrier are executed before hitting the barrier. This is identical to *wait_with_all_memory_fence*.

`void wait_with_all_memory_fence() const restrict(amp)`
Blocks execution of all threads in the thread tile until all threads in the tile have reached this call. Establishes a memory fence on all tile_static and global memory operations executed by the threads in the tile such that all memory operations issued prior to hitting the barrier are visible to all other threads after the barrier has completed and none of the memory operations occurring after the barrier are executed before hitting the barrier. This is identical to *wait*.

`void wait_with_global_memory_fence() const restrict(amp)`
Blocks execution of all threads in the thread tile until all threads in the tile have reached this call. Establishes a memory fence on global memory operations (but not tile-static memory operations) executed by the threads in the tile such that all global memory operations issued prior to hitting the barrier are visible to all other threads after the barrier has completed and none of the global memory operations occurring after the barrier are executed before hitting the barrier.

`void wait_with_tile_static_memory_fence() const restrict(amp)`
Blocks execution of all threads in the thread tile until all threads in the tile have reached this call. Establishes a memory fence on tile-static memory operations (but not global memory operations) executed by the threads in the tile such that all tile_static memory operations issued prior to hitting the barrier are visible to all other threads after the barrier has completed and none of the tile-static memory operations occurring after the barrier are executed before hitting the barrier.

4.5.4 Other Memory Fences

C++ AMP provides functions that serve as memory fences, which establish a happens-before relationship between memory operations performed by threads within the same thread tile. These functions are available in the concurrency namespace. Section 8 provides a complete description of the C++ AMP memory model.

`void all_memory_fence(const tile_barrier&) restrict(amp)`
Establishes a thread-tile scoped memory fence for both global and tile-static memory operations. This function does not imply a barrier and is therefore permitted in divergent code.

`void global_memory_fence(const tile_barrier&) restrict(amp)`
Establishes a thread-tile scoped memory fence for global (but not tile-static) memory operations. This function does not imply a barrier and is therefore permitted in divergent code.

`void tile_static_memory_fence(const tile_barrier&) restrict(amp)`
Establishes a thread-tile scoped memory fence for tile-static (but not global) memory operations. This function does not imply a barrier and is therefore permitted in divergent code.

4.6 completion_future

This class is the return type of all C++ AMP asynchronous APIs and has an interface analogous to std::shared_future<void>. Similar to std:shared_future, this type provides member methods such as ***wait*** and ***get*** to wait for C++ AMP asynchronous operations to finish, and the type additionally provides a member method ***then***, to specify a completion callback *functor* to be executed upon completion of a C++ AMP asynchronous operation. Further this type also contains a member method

to_task (Microsoft specific extension) which returns a *concurrency::task* object which can be used to avail the capabilities of PPL tasks with C++ AMP asynchronous operations; viz. chaining continuations, cancellation etc. This essentially enables "wait-free" composition of C++ AMP asynchronous tasks on accelerators with CPU tasks.

4.6.1 Synopsis

```
class completion_future
{
public:

    completion_future();
    completion_future(const completion_future& _Other);
    completion_future(completion_future&& _Other);
    ~completion_future();
    completion_future& operator=(const completion_future& _Other);
    completion_future& operator=(completion_future&& _Other);

    void get() const;

    bool valid() const;

    void wait() const;
    template <class _Rep, class _Period>
    std::future_status::future_status wait_for(const std::chrono::duration<_Rep, _Period>&
_Rel_time) const;
    template <class _Clock, class _Duration>
    std::future_status::future_status wait_until(const std::chrono::time_point<_Clock,
_Duration>& _Abs_time) const;

    operator std::shared_future<void>() const;

    void then(const _Functor &_Func) const;

    concurrency::task<void> to_task() const;
};
```

4.6.2 Constructors

completion_future()
Default constructor. Constructs an empty uninitialized completion_fuure object which does not refer to any asynchronous operation. Default constructed completion_future objects have **valid() == false**

completion_future (const completion_future& other)	
Copy constructor. Constructs a new **completion_future** object that referes to the same asynchronous operation as the **other** completion_future object.	
Parameters:	
other	An object of type **completion_future** from which to initialize this.

completion_future (completion_future&& other)	
Move constructor. Move constructs a new **completion_future** object that referes to the same asynchronous operation as originally refered by the **other** completion_future object. After this constructor returns, **other.valid() == false**	
Parameters:	
other	An object of type **completion_future** which the new completion_future object is to be move constructed from.

completion_future& operator=(const completion_future& other)	
Copy assignment. Copy assigns the contents of **other** to **this.** This method causes **this** to stop referring its current asynchronous operation and start referring the same asynchronous operation as **other**.	
Parameters:	
other	An object of type **completion_future** which is copy assigned to **this**.

completion_future& operator=(completion_future&& other)	
Move assignment. Move assigns the contents of **other** to **this.** This method causes **this** to stop referring its current asynchronous operation and start referring the same asynchronous operation as **other**. After this method returns, **other.valid() == false**	
Parameters:	
other	An object of type **completion_future** which is move assigned to **this**.

4.6.3 Members

void get() const
This method is functionally identical to **std::shared_future<void>::get**. This method waits for the associated asynchronous operation to finish and returns only upon the completion of the asynchronous operation. If an exception was encountered during the execution of the asynchronous operation, this method throws that stored exception.

bool valid() const
This method is functionally identical to **std::shared_future<void>::valid**. This returns true if **this** completion_future is associated with an asynchronous operation.

```
void wait() const

template <class Rep, class Period>
std::future_status::future_status wait_for(const std::chrono::duration<Rep, Period>&
rel_time) const

template <class Clock, class Duration>
std::future_status::future_status wait_until(const std::chrono::time_point<Clock,
Duration>& abs_time) const
```

These methods are functionally identical to the corresponding **std::shared_future<void>** methods.

The **wait** method waits for the associated asynchronous operation to finish and returns only upon completion of the associated asynchronous operation or if an exception was encountered when executing the asynchronous operation.

The other variants are functionally identical to the **std::shared_future<void>** member methods with same names.

operator shared_future<void>() const
Conversion operator to **std::shared_future<void>**. This method returns a **shared_future<void>** object corresponding to **this** completion_future object and refers to the same asynchronous operation.

```
template <typename Functor>
void then(const Functor &func) const
```

This method enables specification of a completion callback **func** which is executed upon completion of the asynchronous operation associated with **this** completion_future object. The completion callback **func** should have an operator() that is valid when invoked with non arguments, i.e., "**func()**".	
Parameters:	
func	A function object or lambda whose operator() is invoked upon completion of **this**'s associated asynchronous operation.

`concurrency::task<void> to_task() const`
This method returns a **`concurrency::task<void>`** object corresponding to **`this`** completion_future object and refers to the same asynchronous operation. This method is a Microsoft specific extension.

5 Data Containers

5.1 array<T,N>

The type *array<T,N>* represents a dense and regular (not jagged) N-dimensional array which resides on a specific location such as an accelerator or the CPU. The element type of the array is *T*, which is necessarily of a type compatible with the target accelerator. While the rank of the array is determined statically and is part of the type, the extent of the array is runtime-determined, and is expressed using class *extent<N>*. A specific element of an array is selected using an instance of *index<N>*. If “idx” is a valid index for an array with extent “e”, then 0 <= idx[k] < e[k] for 0 <= k < N. Here each “k” is referred to as a dimension and higher-numbered dimensions are referred to as less significant.

The array element type *T* shall be an *amp-compatible* whose size is a multiple of 4 bytes and shall not directly or recursively contain any concurrency containers or reference to concurrency containers.

Array data is laid out contiguously in memory. Elements which differ by one in the least significant dimension are adjacent in memory. This storage layout is typically referred to as *row major* and is motivated by achieving efficient memory access given the standard mapping rules that GPUs use for assigning compute domain values to warps.

Arrays are logically considered to be value types in that when an array is copied to another array, a deep copy is performed. Two arrays never point to the same data.

The *array<T,N>* type is used in several distinct scenarios:

- As a data container to be used in computations on an accelerator
- As a data container to hold memory on the host CPU (to be used to copy to and from other arrays)
- As a staging object to act as a fast intermediary for copying data between host and accelerator.

An array can have any number of dimensions, although some functionality is specialized for *array<T,1>*, *array<T,2>*, and *array<T,3>*. The dimension defaults to 1 if the template argument is elided.

5.1.1 Synopsis

```
template <typename T, int N=1>
class array
{
public:
    static const int rank = N;
    typedef T value_type;

    array() = delete;

    explicit array(const extent<N>& extent);
    array(const extent<N>& extent, accelerator_view av);
    array(const extent<N>& extent, accelerator_view av, accelerator_view associated_av); // staging
```

```
  template <typename InputIterator>
   array(const extent<N>& extent, InputIterator srcBegin);
  template <typename InputIterator>
   array(const extent<N>& extent, InputIterator srcBegin, InputIterator srcEnd);
  template <typename InputIterator>
   array(const extent<N>& extent, InputIterator srcBegin,
      accelerator_view av, accelerator_view associated_av); // staging
  template <typename InputIterator>
   array(const extent<N>& extent, InputIterator srcBegin, InputIterator srcEnd,
      accelerator_view av, accelerator_view associated_av); // staging
  template <typename InputIterator>
   array(const extent<N>& extent, InputIterator srcBegin, accelerator_view av);
  template <typename InputIterator>
   array(const extent<N>& extent, InputIterator srcBegin, InputIterator srcEnd,
      accelerator_view av);

  explicit array(const array_view<const T,N>& src);
  array(const array_view<const T,N>& src,
     accelerator_view av, accelerator_view associated_av); // staging
  array(const array_view<const T,N>& src, accelerator_view av);

  array(const array& other);
  array(array&& other);

  array& operator=(const array& other);
  array& operator=(array&& other);

  array& operator=(const array_view<const T,N>& src);

  void copy_to(array& dest) const;
  void copy_to(const array_view<T,N>& dest) const;

  __declspec(property(get)) extent<N> extent;

  __declspec(property(get)) accelerator_view accelerator_view;
  __declspec(property(get)) accelerator_view associated_accelerator_view;

  T& operator[](const index<N>& idx) restrict(amp,cpu);
  const T& operator[](const index<N>& idx) const restrict(amp,cpu);
  array_view<T,N-1> operator[](int i) restrict(amp,cpu);
  array_view<const T,N-1> operator[](int i) const restrict(amp,cpu);

  const T& operator()(const index<N>& idx) const restrict(amp,cpu);
  T& operator()(const index<N>& idx) restrict(amp,cpu);
  array_view<T,N-1> operator()(int i) restrict(amp,cpu);
  array_view<const T,N-1> operator()(int i) const restrict(amp,cpu);

  array_view<T,N> section(const index<N>& idx, const extent<N>& ext) restrict(amp,cpu);
  array_view<const T,N> section(const index<N>& idx, const extent<N>& ext) const
restrict(amp,cpu);
  array_view<T,N> section(const index<N>& idx) restrict(amp,cpu);
  array_view<const T,N> section(const index<N>& idx) const restrict(amp,cpu);
  array_view<T,N> section(const extent<N>& ext) restrict(amp,cpu);
  array_view<const T,N> section(const extent<N>& ext) const restrict(amp,cpu);

  template <typename ElementType>
   array_view<ElementType,1> reinterpret_as() restrict(amp,cpu);
  template <typename ElementType>
```

```
      array_view<const ElementType,1> reinterpret_as() const restrict(amp,cpu);

    template <int K>
      array_view<T,K> view_as(const extent<K>& viewExtent) restrict(amp,cpu);
    template <int K>
      array_view<const T,K> view_as(const extent<K>& viewExtent) const restrict(amp,cpu);

    operator std::vector<T>() const;

    T* data() restrict(amp,cpu);
    const T* data() const restrict(amp,cpu);
};

template<typename T>
class array<T,1>
{
public:
    static const int rank = 1;
    typedef T value_type;

    array() = delete;

    explicit array(const extent<1>& extent);
    explicit array(int e0);
    array(const extent<1>& extent,
          accelerator_view av, accelerator_view associated_av); // staging
    array(int e0, accelerator_view av, accelerator_view associated_av); // staging
    array(const extent<1>& extent, accelerator_view av);
    array(int e0, accelerator_view av);

    template <typename InputIterator>
      array(const extent<1>& extent, InputIterator srcBegin);
    template <typename InputIterator>
      array(const extent<1>& extent, InputIterator srcBegin, InputIterator srcEnd);
    template <typename InputIterator>
      array(int e0, InputIterator srcBegin);
    template <typename InputIterator>
      array(int e0, InputIterator srcBegin, InputIterator srcEnd);
    template <typename InputIterator>
      array(const extent<1>& extent, InputIterator srcBegin,
            accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(const extent<1>& extent, InputIterator srcBegin, InputIterator srcEnd,
            accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(int e0, InputIterator srcBegin,
            accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(int e0, InputIterator srcBegin, InputIterator srcEnd,
            accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(const extent<1>& extent, InputIterator srcBegin, accelerator_view av);
    template <typename InputIterator>
      array(const extent<1>& extent, InputIterator srcBegin, InputIterator srcEnd,
            accelerator_view av);
    template <typename InputIterator>
      array(int e0, InputIterator srcBegin, InputIterator srcEnd, accelerator_view av);

```

```
    array(const array_view<const T,1>& src);
    array(const array_view<const T,1>& src,
          accelerator_view av, accelerator_view associated_av); // staging
    array(const array_view<const T,1>& src, accelerator_view av);

    array(const array& other);
    array(array&& other);

    array& operator=(const array& other);
    array& operator=(array&& other);

    array& operator=(const array_view<const T,1>& src);

    void copy_to(array& dest) const;
    void copy_to(const array_view<T,1>& dest) const;

    __declspec(property(get)) extent<1> extent;

    __declspec(property(get)) accelerator_view accelerator_view;
    __declspec(property(get)) accelerator_view associated_accelerator_view;


    T& operator[](const index<1>& idx) restrict(amp,cpu);
    const T& operator[](const index<1>& idx) const restrict(amp,cpu);
    T& operator[](int i0) restrict(amp,cpu);
    const T& operator[](int i0) const restrict(amp,cpu);

    T& operator()(const index<1>& idx) restrict(amp,cpu);
    const T& operator()(const index<1>& idx) const restrict(amp,cpu);
    T& operator()(int i0) restrict(amp,cpu);
    const T& operator()(int i0) const restrict(amp,cpu);

    array_view<T,1> section(const index<1>& idx, const extent<1>& ext) restrict(amp,cpu);
    array_view<const T,1> section(const index<1>& idx, const extent<1>& ext) const
restrict(amp,cpu);
    array_view<T,1> section(const index<1>& idx) restrict(amp,cpu);
    array_view<const T,1> section(const index<1>& idx) const restrict(amp,cpu);
    array_view<T,1> section(const extent<1>& ext) restrict(amp,cpu);
    array_view<const T,1> section(const extent<1>& ext) const restrict(amp,cpu);
    array_view<T,1> section(int i0, int e0) restrict(amp,cpu);
    array_view<const T,1> section(int i0, int e0) const restrict(amp,cpu);

    template <typename ElementType>
      array_view<ElementType,1> reinterpret_as() restrict(amp,cpu);
    template <typename ElementType>
      array_view<const ElementType,1> reinterpret_as() const restrict(amp,cpu);

    template <int K>
      array_view<T,K> view_as(const extent<K>& viewExtent) restrict(amp,cpu);
    template <int K>
      array_view<const T,K> view_as(const extent<K>& viewExtent) const restrict(amp,cpu);

    operator std::vector<T>() const;

    T* data() restrict(amp,cpu);
    const T* data() const restrict(amp,cpu);
};
```

```
template<typename T>
class array<T,2>
{
public:
    static const int rank = 2;
    typedef T value_type;

    array() = delete;
    explicit array(const extent<2>& extent);
    array(int e0, int e1);
    array(const extent<2>& extent,
          accelerator_view av, accelerator_view associated_av); // staging
    array(int e0, int e1, accelerator_view av, accelerator_view associated_av); // staging
    array(const extent<2>& extent, accelerator_view av);
    array(int e0, int e1, accelerator_view av);

    template <typename InputIterator>
      array(const extent<2>& extent, InputIterator srcBegin);
    template <typename InputIterator>
      array(const extent<2>& extent, InputIterator srcBegin, InputIterator srcEnd);
    template <typename InputIterator>
      array(int e0, int e1, InputIterator srcBegin);
    template <typename InputIterator>
      array(int e0, int e1, InputIterator srcBegin, InputIterator srcEnd);
    template <typename InputIterator>
      array(const extent<2>& extent, InputIterator srcBegin,
            accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(const extent<2>& extent, InputIterator srcBegin, InputIterator srcEnd,
            accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(int e0, int e2, InputIterator srcBegin,
            accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(int e0, int e2, InputIterator srcBegin, InputIterator srcEnd,
            accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(const extent<2>& extent, InputIterator srcBegin, accelerator_view av);
    template <typename InputIterator>
      array(const extent<2>& extent, InputIterator srcBegin, InputIterator srcEnd,
            accelerator_view av);
    template <typename InputIterator>
      array(int e0, int e1, InputIterator srcBegin, accelerator_view av);
    template <typename InputIterator>
      array(int e0, int e1, InputIterator srcBegin, InputIterator srcEnd, accelerator_view av);

    array(const array_view<const T,2>& src);
    array(const array_view<const T,2>& src,
          accelerator_view av, accelerator_view associated_av); // staging
    array(const array_view<const T,2>& src, accelerator_view av);

    array(const array& other);
    array(array&& other);

    array& operator=(const array& other);
    array& operator=(array&& other);

```

```
    array& operator=(const array_view<const T,2>& src);

    void copy_to(array& dest) const;
    void copy_to(const array_view<T,2>& dest) const;

    __declspec(property(get)) extent<2> extent;

    __declspec(property(get)) accelerator_view accelerator_view;
    __declspec(property(get)) accelerator_view associated_accelerator_view;


    T& operator[](const index<2>& idx) restrict(amp,cpu);
    const T& operator[](const index<2>& idx) const restrict(amp,cpu);
    array_view<T,1> operator[](int i0) restrict(amp,cpu);
    array_view<const T,1> operator[](int i0) const restrict(amp,cpu);

    T& operator()(const index<2>& idx) restrict(amp,cpu);
    const T& operator()(const index<2>& idx) const restrict(amp,cpu);
    T& operator()(int i0, int i1) restrict(amp,cpu);
    const T& operator()(int i0, int i1) const restrict(amp,cpu);

    array_view<T,2> section(const index<2>& idx, const extent<2>& ext) restrict(amp,cpu);
    array_view<const T,2> section(const index<2>& idx, const extent<2>& ext) const
restrict(amp,cpu);
    array_view<T,2> section(const index<2>& idx) restrict(amp,cpu);
    array_view<const T,2> section(const index<2>& idx) const restrict(amp,cpu);
    array_view<T,2> section(const extent<2>& ext) restrict(amp,cpu);
    array_view<const T,2> section(const extent<2>& ext) const restrict(amp,cpu);
    array_view<T,2> section(int i0, int i1, int e0, int e1) restrict(amp,cpu);
    array_view<const T,2> section(int i0, int i1, int e0, int e1) const restrict(amp,cpu);

    template <typename ElementType>
      array_view<ElementType,1> reinterpret_as() restrict(amp,cpu);
    template <typename ElementType>
      array_view<const ElementType,1> reinterpret_as() const restrict(amp,cpu);

    template <int K>
      array_view<T,K> view_as(const extent<K>& viewExtent) restrict(amp,cpu);
    template <int K>
      array_view<const T,K> view_as(const extent<K>& viewExtent) const restrict(amp,cpu);

    operator std::vector<T>() const;

    T* data() restrict(amp,cpu);
    const T* data() const restrict(amp,cpu);
};


template<typename T>
class array<T,3>
{
public:
    static const int rank = 3;
    typedef T value_type;

    array() = delete;

    explicit array(const extent<3>& extent);
```

```
    array(int e0, int e1, int e2);
    array(const extent<3>& extent,
          accelerator_view av, accelerator_view associated_av); // staging
    array(int e0, int e1, int e2,
          accelerator_view av, accelerator_view associated_av); // staging
    array(const extent<3>& extent, accelerator_view av);
    array(int e0, int e1, int e2, accelerator_view av);

    template <typename InputIterator>
      array(const extent<3>& extent, InputIterator srcBegin);
    template <typename InputIterator>
      array(const extent<3>& extent, InputIterator srcBegin, InputIterator srcEnd);
    template <typename InputIterator>
      array(int e0, int e1, int e2, InputIterator srcBegin);
    template <typename InputIterator>
      array(int e0, int e1, int e2, InputIterator srcBegin, InputIterator srcEnd);
    template <typename InputIterator>
      array(const extent<3>& extent, InputIterator srcBegin,
            accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(const extent<3>& extent, InputIterator srcBegin, InputIterator srcEnd,
            accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(int e0, int e2, int e2, InputIterator srcBegin,
           accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(int e0, int e2, int e2, InputIterator srcBegin, InputIterator srcEnd,
           accelerator_view av, accelerator_view associated_av); // staging
    template <typename InputIterator>
      array(const extent<3>& extent, InputIterator srcBegin, accelerator_view av);
    template <typename InputIterator>
      array(const extent<3>& extent, InputIterator srcBegin, InputIterator srcEnd,
            accelerator_view av);
    template <typename InputIterator>
      array(int e0, int e1, int e2, InputIterator srcBegin, accelerator_view av);
    template <typename InputIterator>
      array(int e0, int e1, int e2, InputIterator srcBegin, InputIterator srcEnd,
            accelerator_view av);

    array(const array_view<const T,3>& src);
    array(const array_view<const T,3>& src,
          accelerator_view av, accelerator_view associated_av); // staging
    array(const array_view<const T,3>& src, accelerator_view av);

    array(const array& other);
    array(array&& other);

    array& operator=(const array& other);
    array& operator=(array&& other);

    array& operator=(const array_view<const T,3>& src);

    void copy_to(array& dest) const;
    void copy_to(const array_view<T,3>& dest) const;

    __declspec(property(get)) extent<3> extent;

    __declspec(property(get)) accelerator_view accelerator_view;
```

```
    __declspec(property(get)) accelerator_view associated_accelerator_view;

    T& operator[](const index<3>& idx) restrict(amp,cpu);
    const T& operator[](const index<3>& idx) const restrict(amp,cpu);
    array_view<T,2> operator[](int i0) restrict(amp,cpu);
    array_view<const T,2> operator[](int i0) const restrict(amp,cpu);

    T& operator()(const index<3>& idx) restrict(amp,cpu);
    const T& operator()(const index<3>& idx) const restrict(amp,cpu);
    T& operator()(int i0, int i1, int i2) restrict(amp,cpu);
    const T& operator()(int i0, int i1, int i2) const restrict(amp,cpu);

    array_view<T,3> section(const index<3>& idx, const extent<3>& ext) restrict(amp,cpu);
    array_view<const T,3> section(const index<3>& idx, const extent<3>& ext) const
restrict(amp,cpu);
    array_view<T,3> section(const index<3>& idx) restrict(amp,cpu);
    array_view<const T,3> section(const index<3>& idx) const restrict(amp,cpu);
    array_view<T,3> section(const extent<3>& ext) restrict(amp,cpu);
    array_view<const T,3> section(const extent<3>& ext) const restrict(amp,cpu);
    array_view<T,3> section(int i0, int i1, int i2,
                            int e0, int e1, int e2) restrict(amp,cpu);
    array_view<const T,3> section(int i0, int i1, int i2,
                                  int e0, int e1, int e2) const restrict(amp,cpu);

    template <typename ElementType>
      array_view<ElementType,1> reinterpret_as() restrict(amp,cpu);
    template <typename ElementType>
      array_view<const ElementType,1> reinterpret_as() const restrict(amp,cpu);

    template <int K>
      array_view<T,K> view_as(const extent<K>& viewExtent) restrict(amp,cpu);
    template <int K>
      array_view<const T,K> view_as(const extent<K>& viewExtent) const restrict(amp,cpu);

    operator std::vector<T>() const;

    T* data() restrict(amp,cpu);
    const T* data() const restrict(amp,cpu);
};
```

`template <typename T, int N=1> class array`	
Represents an N-dimensional region of memory (with type T) located on an accelerator.	
Template Arguments	
T	The element type of this array
N	The dimensionality of the array, defaults to 1 if elided.

`static const int rank = N`
The rank of this array.

`typedef T value_type;`
The element type of this array.

5.1.2 Constructors

There is no default constructor for *array<T,N>*. All constructors are restricted to run on the CPU only (can't be executed on an amp target).

```
array(const array& other)
```

Copy constructor. Constructs a new **array<T,N>** from the supplied argument other. The new array is located on the same accelerator_view as the source array. A deep copy is performed.

Parameters:

Other	An object of type **array<T,N>** from which to initialize this new array.

```
array(array&& other)
```

Move constructor. Constructs a new **array<T,N>** by moving from the supplied argument other.

Parameters:

Other	An object of type **array<T,N>** from which to initialize this new array.

```
explicit array(const extent<N>& extent)
```

Constructs a new array with the supplied extent, located on the default view of the default accelerator. If any components of the extent are non-positive, an exception will be thrown.

Parameters:

Extent	The extent in each dimension of this array.

```
explicit array<T,1>::array(int e0)
array<T,2>::array(int e0, int e1)
array<T,3>::array(int e0, int e1, int e2)
```

Equivalent to construction using "**array(extent<N>(e0 [, e1 [, e2]]))**".

Parameters:

e0 [, e1 [, e2]]	The component values that will form the extent of this array.

```
template <typename InputIterator>
    array(const extent<N>& extent, InputIterator srcBegin [, InputIterator srcEnd])
```

Constructs a new array with the supplied extent, located on the default accelerator, initialized with the contents of a source container specified by a beginning and optional ending iterator. The source data is copied by value into this array as if by calling "**copy()**".

If the number of available container elements is less than this->extent.size(), undefined behavior results.

Parameters:

extent	The extent in each dimension of this array.
srcBegin	A beginning iterator into the source container.
srcEnd	An ending iterator into the source container.

```
template <typename InputIterator>
  array<T,1>::array(int e0, InputIterator srcBegin [, InputIterator srcEnd])
template <typename InputIterator>
  array<T,2>::array(int e0, int e1, InputIterator srcBegin [, InputIterator srcEnd])
template <typename InputIterator>
  array<T,3>::array(int e0, int e1, int e2, InputIterator srcBegin [, InputIterator srcEnd])
```

Equivalent to construction using "**array(extent<N>(e0 [, e1 [, e2]]), src)**".

Parameters:

e0 [, e1 [, e2]]	The component values that will form the extent of this array.

srcBegin	A beginning iterator into the source container.
srcEnd	An ending iterator into the source container.

2044

`explicit array(const array_view<const T,N>& src)`	
Constructs a new array, located on the default view of the default accelerator, initialized with the contents of the array_view "src". The extent of this array is taken from the extent of the source array_view. The "src" is copied by value into this array as if by calling "**`copy(src, *this)`**" (see 5.3.2).	
Parameters:	
src	An **`array_view`** object from which to copy the data into this array (and also to determine the extent of this array).

2045

`explicit array(const extent<N>& extent, accelerator_view av)`	
Constructs a new array with the supplied extent, located on the accelerator bound to the **`accelerator_view`** "av".	
Parameters:	
extent	The extent in each dimension of this array.
av	An **`accelerator_view`** object which specifies the location of this array.

2046

`array<T,1>::array(int e0, accelerator_view av)`
`array<T,2>::array(int e0, int e1, accelerator_view av)`
`array<T,3>::array(int e0, int e1, int e2, accelerator_view av)`	

Equivalent to construction using "**`array(extent<N>(e0 [, e1 [, e2 ]]), av)`**".	
Parameters:	
e0 [, e1 [, e2]]	The component values that will form the extent of this array.
av	An **`accelerator_view`** object which specifies the location of this array.

2047

`template <typename InputIterator>`
`  array(const extent<N>& extent, InputIterator srcBegin [, InputIterator srcEnd],`
`        accelerator_view av)`	

Constructs a new array with the supplied extent, located on the accelerator bound to the **`accelerator_view`** "av", initialized with the contents of the source container specified by a beginning and optional ending iterator. The data is copied by value into this array as if by calling "**`copy()`**".	
Parameters:	
extent	The extent in each dimension of this array.
srcBegin	A beginning iterator into the source container.
srcEnd	An ending iterator into the source container.
av	An **`accelerator_view`** object which specifies the location of this array.

2048

`array(const array_view<const T,N>& src, accelerator_view av)`
Constructs a new array initialized with the contents of the array_view "src". The extent of this array is taken from the extent of the source array_view. The "src" is copied by value into this array as if by calling "**`copy(src, *this)`**" (see 5.3.2). The new array is located on the accelerator bound to the **`accelerator_view`** "av".
Parameters:

src	An **`array_view`** object from which to copy the data into this array (and also to determine the extent of this array).
av	An **`accelerator_view`** object which specifies the location of this array

```
template <typename InputIterator>
  array<T,1>::array(int e0, InputIterator srcBegin [, InputIterator srcEnd],
                    accelerator_view av)
template <typename InputIterator>
  array<T,2>::array(int e0, int e1, InputIterator srcBegin [, InputIterator srcEnd],
                    accelerator_view av)
template <typename InputIterator>
  array<T,3>::array(int e0, int e1, int e2, InputIterator srcBegin [, InputIterator srcEnd],
                    accelerator_view av)
```

Equivalent to construction using "**`array(extent<N>(e0 [, e1 [, e2 ]]), srcBegin [, srcEnd], av)`**".

Parameters:

e0 [, e1 [, e2]]	The component values that will form the extent of this array.
srcBegin	A beginning iterator into the source container.
srcEnd	An ending iterator into the source container.
av	An **`accelerator_view`** object which specifies the location of this array.

5.1.2.1 Staging Array Constructors

Staging arrays are used as a hint to optimize repeated copies between two accelerators (in V1 practically this is between the CPU and an accelerator). Staging arrays are optimized for data transfers, and do not have stable user-space memory.

Microsoft-specific: On Windows, staging arrays are backed by DirectX staging buffers which have the correct hardware alignment to ensure efficient DMA transfer between the CPU and a device.

Staging arrays are differentiated from normal arrays by their construction with a second accelerator. Note that the *accelerator_view* property of a staging array returns the value of the first accelerator argument it was constructed with (*av*, below).

It is illegal to change or examine the contents of a staging array while it is involved in a transfer operation (i.e., between lines 17 and 22 in the following example):

```
1. class SimulationServer
2. {
3.     array<float,2> acceleratorArray;
4.     array<float,2> stagingArray;
5. public:
6.     SimulationServer(const accelerator_view& av)
7.         :acceleratorArray(extent<2>(1000,1000), av),
8.          stagingArray(extent<2>(1000,1000), accelerator("cpu").default_view,
9.          accelerator("gpu").default_view)
10.    {
11.    }
12.
13.    void OnCompute()
14.    {
15.        array<float,2> &a = acceleratorArray;
16.        ApplyNetworkChanges(stagingArray.data());
17.        a = stagingArray;
18.        parallel_for_each(a.extents, [&](index<2> idx)
19.        {
20.           // Update a[idx] according to simulation
```

```
        }
        stagingArray = a;
        SendToClient(stagingArray.data());
    }
};
```

`array(const extent<N>& extent, accelerator_view av, accelerator_view associated_av)`	
Constructs a staging array with the given extent, which acts as a staging area between accelerator views "av" and "associated_av". If "av" is a cpu accelerator view, this will construct a staging array which is optimized for data transfers between the CPU and "associated_av".	
Parameters:	
extent	The extent in each dimension of this array.
av	An **`accelerator_view`** object which specifies the home location of this array.
associated_av	An **`accelerator_view`** object which specifies a target device accelerator.

`array<T,1>::array(int e0, accelerator_view av, accelerator_view associated_av)`
`array<T,2>::array(int e0, int e1, accelerator_view av, accelerator_view associated_av)`
`array<T,3>::array(int e0, int e1, int e2, accelerator_view av, accelerator_view associated_av)`	

Equivalent to construction using "**`array(extent<N>(e0 [, e1 [, e2 ]]), av, associated_av)`**".	
Parameters:	
e0 [, e1 [, e2]]	The component values that will form the extent of this array.
av	An **`accelerator_view`** object which specifies the home location of this array.
associated_av	An **`accelerator_view`** object which specifies a target device accelerator.

`template <typename InputIterator>` `array(const extent<N>& extent, InputIterator srcBegin [, InputIterator srcEnd], accelerator_view av, accelerator_view associated_av)`	
Constructs a staging array with the given extent, which acts as a staging area between accelerators "av" (which must be the CPU accelerator) and "associated_av". The staging array will be initialized with the data specified by "src" as if by calling "**`copy(src, *this)`**" (see 5.3.2).	
Parameters:	
extent	The extent in each dimension of this array.
srcBegin	A beginning iterator into the source container.
srcEnd	An ending iterator into the source container.
av	An **`accelerator_view`** object which specifies the home location of this array.
associated_av	An **`accelerator_view`** object which specifies a target device accelerator.

`array(const array_view<const T,N>& src, accelerator_view av, accelerator_view associated_av)`
Constructs a staging array initialized with the array_view given by "src", which acts as a staging area between accelerators "av" (which must be the CPU accelerator) and "associated_av". The extent of this array is taken from the extent of the source array_view. The staging array will be initialized from "src" as if by calling "**`copy(src, *this)`**" (see 5.3.2).

Parameters:	
src	An **array_view** object from which to copy the data into this array (and also to determine the extent of this array).
av	An **accelerator_view** object which specifies the home location of this array.
associated_av	An **accelerator_view** object which specifies a target device accelerator.

```
template <typename InputIterator>
  array<T,1>::array(int e0, InputIterator srcBegin [, InputIterator srcEnd], accelerator_view av, accelerator_view associated_av)
template <typename InputIterator>
  array<T,2>::array(int e0, int e1, InputIterator srcBegin [, InputIterator srcEnd],
                    accelerator_view av, accelerator_view associated_av)
template <typename InputIterator>
  array<T,3>::array(int e0, int e1, int e2, InputIterator srcBegin [, InputIterator srcEnd],
                    accelerator_view av, accelerator_view associated_av)
```

Equivalent to construction using "**array(extent<N>(e0 [, e1 [, e2]]), src, av, associated_av)**".

Parameters:	
e0 [, e1 [, e2]]	The component values that will form the extent of this array.
srcBegin	A beginning iterator into the source container.
srcEnd	An ending iterator into the source container.
av	An **accelerator_view** object which specifies the home location of this array.
associated_av	An **accelerator_view** object which specifies a target device accelerator.

5.1.3 Members

```
__declspec(property(get)) extent<N> extent
extent<N> get_extent() const restrict(cpu,amp)
```

Access the extent that defines the shape of this array.

```
__declspec(property(get)) accelerator_view accelerator_view
```

This property returns the accelerator_view representing the location where this array has been allocated. This property is only accessible on the CPU.

```
__declspec(property(get)) accelerator_view associated_accelerator_view
```

This property returns the accelerator_view representing the preferred target where this array can be copied.

```
array& operator=(const array& other)
```

Assigns the contents of the array "other" to this array, using a deep copy. This function can only be called on the CPU.

Parameters:	
other	An object of type **array<T,N>** from which to copy into this array.

Return Value:

Returns ***this**.

```
array& operator=(array&& other)
```

Moves the contents of the array "other" to this array. This function can only be called on the CPU.

Parameters:	
other	An object of type `array<T,N>` from which to move into this array.
Return Value:	
Returns `*this`.	

2103

`array& operator=(const array_view<const T,N>& src)`	
Assigns the contents of the array_view “src”, as if by calling “copy(src, *this)” (see 5.3.2).	
Parameters:	
src	An object of type `array_view<T,N>` from which to copy into this array.
Return Value:	
Returns `*this`.	

2104

`void copy_to(array<T,N>& dest)`	
Copies the contents of this array to the array given by “dest”, as if by calling “copy(*this, dest)” (see 5.3.2).	
Parameters:	
dest	An object of type `array <T,N>` to which to copy data from this array.

2105

`void copy_to(const array_view<T,N>& dest)`	
Copies the contents of this array to the array_view given by “dest”, as if by calling “copy(*this, dest)” (see 5.3.2).	
Parameters:	
dest	An object of type `array_view<T,N>` to which to copy data from this array.

2106

`T* data() restrict(amp,cpu)` `const T* data() const restrict(amp,cpu)`
Returns a pointer to the raw data underlying this array.
Return Value:
A (const) pointer to the first element in the linearized array.

2107

`operator std::vector<T>() const`
Implicitly converts an array to a std::vector, as if by “`copy(*this, ` ***vector***`)`” (see 5.3.2).
Return Value:
An object of type vector<T> which contains a copy of the data contained on the array.

2108

2109
5.1.4 Indexing
2110

`T& operator[](const index<N>& idx) restrict(amp,cpu)` `T& operator()(const index<N>& idx) restrict(amp,cpu)`	
Returns a reference to the element of this array that is at the location in N-dimensional space specified by “idx”. Accessing array data on from a location where it is not resident (e.g. from the CPU when it is resident on a GPU) results in an exception or undefined behavior.	
Parameters:	
idx	An object of type `index<N>` from that specifies the location of the element.

2111

`const T& operator[](const index<N>& idx) const restrict(amp,cpu)` `const T& operator()(const index<N>& idx) const restrict(amp,cpu)`	
Returns a const reference to the element of this array that is at the location in N-dimensional space specified by “idx”. Accessing array data on from a location where it is not resident (e.g. from the CPU when it is resident on a GPU) results in an exception or undefined behavior.	
Parameters:	
idx	An object of type `index<N>` from that specifies the location of the element.

2112

`T& array<T,1>::operator()(int i0) restrict(amp,cpu)` `T& array<T,1>::operator[](int i0) restrict(amp,cpu)`

```
T& array<T,2>::operator()(int i0, int i1) restrict(amp,cpu)
T& array<T,3>::operator()(int i0, int i1, int i2) restrict(amp,cpu)
```

Equivalent to "**`array<T,N>::operator()(index<N>(i0 [, i1 [, i2 ]]))`**".	
Parameters:	
i0 [, i1 [, i2]]	The component values that will form the index into this array.

```
const T& array<T,1>::operator()(int i0) const restrict(amp,cpu)
const T& array<T,1>::operator[](int i0) const restrict(amp,cpu)
const T& array<T,2>::operator()(int i0, int i1) const restrict(amp,cpu)
const T& array<T,3>::operator()(int i0, int i1, int i2) const restrict(amp,cpu)
```

Equivalent to "**`array<T,N>::operator()(index<N>(i0 [, i1 [, i2 ]])) const`**".	
Parameters:	
i0 [, i1 [, i2]]	The component values that will form the index into this array.

```
array_view<T,N-1> operator[](int i0) restrict(amp,cpu)
array_view<const T,N-1> operator[](int i0) const restrict(amp,cpu)
```

This overload is defined for array<T,N> where N ≥ 2.

This mode of indexing is equivalent to projecting on the most-significant dimension. It allows C-style indexing. For example:

```
array<float,4> myArray(myExtents, …);

myArray[index<4>(5,4,3,2)] = 7;
assert(myArray[5][4][3][2] == 7);
```

Parameters:	
i0	An integer that is the index into the most-significant dimension of this array.
Return Value:	
Returns an array_view whose dimension is one lower than that of this array.	

5.1.5 View Operations

```
array_view<T,N> section(const index<N>& offset, const extent<N>& ext) restrict(amp,cpu)
array_view<const T,N> section(const index<N>& offset, const extent<N>& ext) const restrict(amp,cpu)
```

See "**`array_view<T,N>::section(`**`const index<N>&, const extent<N>&)` in section 5.2.5 for a description of this function.

```
array_view<T,N> section(const index<N>& idx) restrict(amp,cpu)
array_view<const T,N> section(const index<N>& idx) const restrict(amp,cpu)
```

Equivalent to "**`section(idx, this->extent – idx)`**".

```
array_view<T,N> section(const extent<N>& ext) restrict(amp,cpu)
array_view<const T,N> section(const extent<N>& ext) const restrict(amp,cpu)
```

Equivalent to "**`section(index<N>(), ext)`**".

```
array_view<T,1> array<T,1>::section(int i0, int e0) restrict(amp,cpu)
array_view<const T,1> array<T,1>::section(int i0, int e0) const restrict(amp,cpu)
array_view<T,2> array<T,2>::section(int i0, int i1, int e0, int e1) restrict(amp,cpu)
array_view<const T,2> array<T,2>::section(int i0, int i1,
                                          int e0, int e1) const restrict(amp,cpu)
array_view<T,3> array<T,3>::section(int i0, int i1, int i2,
                                    int e0, int e1, int e2) restrict(amp,cpu)
```

```
array_view<const T,3> array<T,3>::section(int i0, int i1, int i2,
                                          int e0, int e1, int e2) const restrict(amp,cpu)
```

Equivalent to "`array<T,N>::section(index<N>(i0 [, i1 [, i2 ]]), extent<N>(e0 [, e1 [, e2 ]])) const`".

Parameters:

i0 [, i1 [, i2]]	The component values that will form the origin of the section
e0 [, e1 [, e2]]	The component values that will form the extent of the section

```
template<typename ElementType>
  array_view<ElementType,1> reinterpret_as() restrict(amp,cpu)
template<typename ElementType>
  array_view<const ElementType,1> reinterpret_as() const restrict(amp,cpu)
```

Sometimes it is desirable to view the data of an N-dimensional array as a linear array, possibly with a (unsafe) reinterpretation of the element type. This can be achieved through the `reinterpret_as` member function. Example:

```
struct RGB { float r; float g; float b; };

array<RGB,3>  a = ...;
array_view<float,1> v = a.reinterpret_as<float>();

assert(v.extent == 3*a.extent);
```

The size of the reinterpreted ElementType must evenly divide into the total size of this array.

Return Value:

Returns an `array_view` from this `array<T,N>` with the element type reinterpreted from `T` to `ElementType`, and the rank reduced from `N` to 1.

```
template <int K>
  array_view<T,K> view_as(extent<K> viewExtent) restrict(amp,cpu)
template <int K>
  array_view<const T,K> view_as(extent<K> viewExtent) const restrict(amp,cpu)
```

An array of higher rank can be reshaped into an array of lower rank, or vice versa, using the `view_as` member function. Example:

```
array<float,1> a(100);

array_view<float,2> av = a.view_as(extent<2>(2,50));
```

Return Value:

Returns an `array_view` from this `array<T,N>` with the rank changed to K from N.

5.2 array_view<T,N>

The *array_view<T,N>* type represents a possibly cached view into the data held in an *array<T,N>*, or a section thereof. It also provides such views over native CPU data. It exposes an indexing interface congruent to that of *array<T,N>*.

Like an *array*, an *array_view* is an N-dimensional object, where N defaults to 1 if it is omitted.

The array element type *T* shall be an *amp-compatible* whose size is a multiple of 4 bytes and shall not directly or recursively contain any concurrency containers or reference to concurrency containers.

.

*array_view*s may be accessed locally, where their source data lives, or remotely on a different accelerator_view or coherence domain. When they are accessed remotely, views are copied and cached as necessary. Except for the effects of automatic caching, *array_view*s have a performance profile similar to that of arrays (small to negligible access penalty when accessing the data through views).

There are three remote usage scenarios:

1. A view to a system memory pointer is passed through a *parallel_for_each* call to an accelerator and accessed on the accelerator.
2. A view to an accelerator-residing array is passed using a *parallel_for_each* to another accelerator_view and is accessed there.
3. A view to an accelerator-residing array is accessed on the CPU.

When any of these scenarios occur, the referenced views are implicitly copied by the system to the remote location and, if modified through the *array_view*, copied back to the home location. The Implementation is free to optimize copying changes back; may only copy changed elements, or may copy unchanged portions as well. Overlapping *array_view*s to the same data source are *not guaranteed to maintain aliasing between* arrays*/*array_views on a remote location.

Multi-threaded access to the same data source, either directly or through views, must be synchronized by the user.

The runtime makes the following guarantees regarding caching of data inside array views.

1. Let A be an array and V a view to the array. Then, all well-synchronized accesses to A and V in program order obey a serial happens-before relationship.
2. Let A be an array and V1 and V2 be overlapping views to the array.
   - When executing on the accelerator where A has been allocated, all well-synchronized accesses through A, V1 and V2 are aliased through A and induce a total happens-before relationship which obeys program order. (No caching.)
   - Otherwise, if they are executing on different accelerators, then the behaviour of writes to V1 and V2 is undefined (a race).

When an *array_view* is created over a pointer in system memory, the user commits to:

1. only changing the data accessible through the view directly through the view class, **or**
2. adhering to the following rules when accessing the data directly (not through the view):
   a. Calling *synchronize()* before the data is accessed directly, **and**
   b. If the underlying data is modified, calling *refresh()* prior to further accessing it through the view.

(Note: The underlying data of an array_view is updated when the last copy of an array_view having pending writes goes out of scope or is otherwise destructed.)

Either action will notify the *array_view* that the underlying native memory has changed and that any accelerator-residing copies are now stale. If the user abides by these rules then the guarantees provided by the system for pointer-based views are identical to those provided to views of data-parallel arrays.

The memory allocation underlying a concurrency::array is reference counted for automatic lifetime management. The array and all array_views created from it hold references to the allocation and the allocation lives till there exists at least one array or array_view object that references the allocation. Thus it is legal to access the array_view(s) even after the source concurrency::array object has been destructed.

When an array_view is created over native CPU data (such as raw CPU memory, std::vector, etc), it is the user's responsibility to ensure that the source data outlives all array_views created over that source. Any attempt to access the array_view contents after native CPU data has been deallocated has undefined behavior.

5.2.1 Synopsis

The *array_view<T,N>* has the following specializations:

- *array_view<T,1>*
- *array_view<T,2>*

- *array_view<T,3>*
- *array_view<const T,N>*
- *array_view<const T,1>*
- *array_view<const T,2>*
- *array_view<const T,3>*

5.2.1.1 array_view<T,N>

The generic *array_view<T,N>* represents a view over elements of type *T* with rank *N*. The elements are both readable and writeable.

```
template <typename T, int N = 1>
class array_view
{
public:
    static const int rank = N;
    typedef T value_type;

    array_view() = delete;
    array_view(array<T,N>& src) restrict(amp,cpu);
    template <typename Container>
      array_view(const extent<N>& extent, Container& src);
    array_view(const extent<N>& extent, value_type* src) restrict(amp,cpu);

    array_view(const array_view& other) restrict(amp,cpu);

    array_view& operator=(const array_view& other) restrict(amp,cpu);

    void copy_to(array<T,N>& dest) const;
    void copy_to(const array_view& dest) const;

    __declspec(property(get)) extent<N> extent;

    // These are restrict(amp,cpu)
    T& operator[](const index<N>& idx) const restrict(amp,cpu);
    array_view<T,N-1> operator[](int i) const restrict(amp,cpu);

    T& operator()(const index<N>& idx) const restrict(amp,cpu);
    array_view<T,N-1> operator()(int i) const restrict(amp,cpu);

    array_view<T,N> section(const index<N>& idx, const extent<N>& ext) restrict(amp,cpu);
    array_view<T,N> section(const index<N>& idx) const restrict(amp,cpu);
    array_view<T,N> section(const extent<N>& ext) const restrict(amp,cpu);

    void synchronize() const;
    completion_future synchronize_async() const;

    void refresh() const;
    void discard_data() const;

};

template <typename T>
class array_view<T,1>
{
public:
    static const int rank = 1;
```

```
    typedef T value_type;

    array_view() = delete;
    array_view(array<T,1>& src) restrict(amp,cpu);
    template <typename Container>
      array_view(const extent<1>& extent, Container& src);
    template <typename Container>
      array_view(int e0, Container& src);
    array_view(const extent<1>& extent, value_type* src) restrict(amp,cpu);
    array_view(int e0, value_type* src) restrict(amp,cpu);

    array_view(const array_view& other) restrict(amp,cpu);

    array_view& operator=(const array_view& other) restrict(amp,cpu);

    void copy_to(array<T,1>& dest) const;
    void copy_to(const array_view& dest) const;

    __declspec(property(get)) extent<1> extent;

    T& operator[](const index<1>& idx) const restrict(amp,cpu);
    T& operator[](int i) const restrict(amp,cpu);

    T& operator()(const index<1>& idx) const restrict(amp,cpu);
    T& operator()(int i) const restrict(amp,cpu);

    array_view<T,1> section(const index<1>& idx, const extent<1>& ext) const restrict(amp,cpu);
    array_view<T,1> section(const index<1>& idx)  const restrict(amp,cpu);
    array_view<T,1> section(const extent<1>& ext)  const restrict(amp,cpu);
    array_view<T,1> section(int i0, int e0) restrict(amp,cpu);

    template <typename ElementType>
      array_view<ElementType,1> reinterpret_as() const restrict(amp,cpu);

    template <int K>
      array_view<T,K> view_as(extent<K> viewExtent) const restrict(amp,cpu);

    T* data() const restrict(amp,cpu);

    void synchronize() const;
    completion_future synchronize_async() const;

    void refresh() const;
    void discard_data() const;
 };


template <typename T>
class array_view<T,2>
{
public:
    static const int rank = 2;
    typedef T value_type;

    array_view() = delete;
    array_view(array<T,2>& src) restrict(amp,cpu);
    template <typename Container>
      array_view(const extent<2>& extent, Container& src);
```

```
    template <typename Container>
      array_view(int e0, int e1, Container& src);
    array_view(const extent<2>& extent, value_type* src) restrict(amp,cpu);
    array_view(int e0, int e1, value_type* src) restrict(amp,cpu);

    array_view(const array_view& other) restrict(amp,cpu);

    array_view& operator=(const array_view& other) restrict(amp,cpu);

    void copy_to(array<T,2>& dest) const;
    void copy_to(const array_view& dest) const;

    __declspec(property(get)) extent<2> extent;

    T& operator[](const index<2>& idx) const restrict(amp,cpu);
    array_view<T,1> operator[](int i) const restrict(amp,cpu);

    T& operator()(const index<2>& idx) const restrict(amp,cpu);
    T& operator()(int i0, int i1) const restrict(amp,cpu);

    array_view<T,2> section(const index<2>& idx, const extent<2>& ext) const restrict(amp,cpu);
    array_view<T,2> section(const index<2>& idx) const restrict(amp,cpu);
    array_view<T,2> section(const extent<2>& ext) const restrict(amp,cpu);
    array_view<T,2> section(int i0, int i1, int e0, int e1) const restrict(amp,cpu);

    void synchronize() const;
    completion_future synchronize_async() const;

    void refresh() const;
    void discard_data() const;
};

template <typename T>
class array_view<T,3>
{
public:
    static const int rank = 3;
    typedef T value_type;

    array_view() = delete;
    array_view(array<T,3>& src) restrict(amp,cpu);
    template <typename Container>
      array_view(const extent<3>& extent, Container& src);
    template <typename Container>
      array_view(int e0, int e1, int e2, Container& src);
    array_view(const extent<3>& extent, value_type* src) restrict(amp,cpu);
    array_view(int e0, int e1, int e2, value_type* src) restrict(amp,cpu);

    array_view(const array_view& other) restrict(amp,cpu);

    array_view& operator=(const array_view& other) restrict(amp,cpu);

    void copy_to(array<T,3>& dest) const;
    void copy_to(const array_view& dest) const;

    __declspec(property(get)) extent<3> extent;

    T& operator[](const index<3>& idx) const restrict(amp,cpu);
```

```
    array_view<T,2> operator[](int i) const restrict(amp,cpu);

    T& operator()(const index<3>& idx) const restrict(amp,cpu);
    T& operator()(int i0, int i1, int i2) const restrict(amp,cpu);

    array_view<T,3> section(const index<3>& idx, const extent<3>& ext) const restrict(amp,cpu);
    array_view<T,3> section(const index<3>& idx) const restrict(amp,cpu);
    array_view<T,3> section(const extent<3>& ext) const restrict(amp,cpu);
    array_view<T,3> section(int i0, int i1, int i2, int e0, int e1, int e2) const
restrict(amp,cpu);

    void synchronize() const;
    completion_future synchronize_async() const;

    void refresh() const;
    void discard_data() const;
};
```

5.2.1.2 array_view<const T,N>

The partial specialization *array_view<const T,N>* represents a view over elements of type const *T* with rank *N*. The elements are readonly. At the boundary of a call site (such as *parallel_for_each*), this form of *array_view* need only be copied to the target accelerator if it isn't already there. It will not be copied out.

```
template <typename T, int N=1>
class array_view<const T,N>
{
public:
    static const int rank = N;
    typedef const T value_type;

    array_view() = delete;
    array_view(const array<T,N>& src) restrict(amp,cpu);
    template <typename Container>
      array_view(const extent<N>& extent, const Container& src);
    array_view(const extent<N>& extent, const value_type* src) restrict(amp,cpu);

    array_view(const array_view<T,N>& other) restrict(amp,cpu);
    array_view(const array_view<const T,N>& other) restrict(amp,cpu);

    array_view& operator=(const array_view& other) restrict(amp,cpu);

    void copy_to(array<T,N>& dest) const;
    void copy_to(const array_view<T,N>& dest) const;

    __declspec(property(get)) extent<N> extent;

    const T& operator[](const index<N>& idx) const restrict(amp,cpu);
    array_view<const T,N-1> operator[](int i) const restrict(amp,cpu);

    const T& operator()(const index<N>& idx) const restrict(amp,cpu);
    array_view<const T,N-1> operator()(int i) const restrict(amp,cpu);

    array_view<const T,N> section(const index<N>& idx, const extent<N>& ext) const
restrict(amp,cpu);
    array_view<const T,N> section(const index<N>& idx) const restrict(amp,cpu);
    array_view<const T,N> section(const extent<N>& ext) const restrict(amp,cpu);
```

```
    void refresh() const;
 };

template <typename T>
class array_view<const T,1>
{
public:
    static const int rank = 1;
    typedef const T value_type;

    array_view() = delete;
    array_view(const array<T,1>& src) restrict(amp,cpu);
    template <typename Container>
      array_view(const extent<1>& extent, const Container& src);
    template <typename Container>
      array_view(int e0, const Container& src);
    array_view(const extent<1>& extent, const value_type* src) restrict(amp,cpu);
    array_view(int e0, const value_type* src) restrict(amp,cpu);

    array_view(const array_view<T,1>& other) restrict(amp,cpu);
    array_view(const array_view<const T,1>& other) restrict(amp,cpu);

    array_view& operator=(const array_view& other) restrict(amp,cpu);

    void copy_to(array<T,1>& dest) const;
    void copy_to(const array_view<T,1>& dest) const;

     __declspec(property(get)) extent<1> extent;

    // These are restrict(amp,cpu)
    const T& operator[](const index<1>& idx) const restrict(amp,cpu);
    const T& operator[](int i) const restrict(amp,cpu);

    const T& operator()(const index<1>& idx) const restrict(amp,cpu);
    const T& operator()(int i) const restrict(amp,cpu);

    array_view<const T,1> section(const index<N>& idx, const extent<N>& ext) const
restrict(amp,cpu);
    array_view<const T,1> section(const index<1>& idx) const restrict(amp,cpu);
    array_view<const T,1> section(const extent<1>& ext) const restrict(amp,cpu);
    array_view<const T,1> section(int i0, int e0) const restrict(amp,cpu);

    template <typename ElementType>
      array_view<const ElementType,1> reinterpret_as() const restrict(amp,cpu);

    template <int K>
      array_view<const T,K> view_as(extent<K> viewExtent) const restrict(amp,cpu);

    const T* data() const restrict(amp,cpu);

    void refresh() const;
 };

template <typename T>
class array_view<const T,2>
{
public:
```

```
    static const int rank = 2;
    typedef const T value_type;

    array_view() = delete;
    array_view(const array<T,2>& src) restrict(amp,cpu);
    template <typename Container>
      array_view(const extent<2>& extent, const Container& src);
    template <typename Container>
      array_view(int e0, int e1, const Container& src);
    array_view(const extent<2>& extent, const value_type* src) restrict(amp,cpu);
    array_view(int e0, int e1, const value_type* src) restrict(amp,cpu);

    array_view(const array_view<T,2>& other) restrict(amp,cpu);
    array_view(const array_view<const T,2>& other) restrict(amp,cpu);

    array_view& operator=(const array_view& other) restrict(amp,cpu);

    void copy_to(array<T,2>& dest) const;
    void copy_to(const array_view<T,2>& dest) const;

     __declspec(property(get)) extent<2> extent;

    const T& operator[](const index<2>& idx) const restrict(amp,cpu);
    array_view<const T,1> operator[](int i) const restrict(amp,cpu);

    const T& operator()(const index<2>& idx) const restrict(amp,cpu);
    const T& operator()(int i0, int i1) const restrict(amp,cpu);

    array_view<const T,2> section(const index<2>& idx, const extent<2>& ext) const
restrict(amp,cpu);
    array_view<const T,2> section(const index<2>& idx) const restrict(amp,cpu);
    array_view<const T,2> section(const extent<2>& ext) const restrict(amp,cpu);
    array_view<const T,2> section(int i0, int i1, int e0, int e1) const restrict(amp,cpu);

    void refresh() const;
 };

template <typename T>
class array_view<const T,3>
{
public:
    static const int rank = 3;
    typedef const T value_type;

    array_view() = delete;
    array_view(const array<T,3>& src) restrict(amp,cpu);
    template <typename Container>
      array_view(const extent<3>& extent, const Container& src);
    template <typename Container>
      array_view(int e0, int e1, int e2, const Container& src);
    array_view(const extent<3>& extent, const value_type* src) restrict(amp,cpu);
    array_view(int e0, int e1, int e2, const value_type* src) restrict(amp,cpu);

    array_view(const array_view<T,3>& other) restrict(amp,cpu);
    array_view(const array_view<const T,3>& other) restrict(amp,cpu);

    array_view& operator=(const array_view& other) restrict(amp,cpu);

```

```
    void copy_to(array<T,3>& dest) const;
    void copy_to(const array_view<T,3>& dest) const;

     __declspec(property(get)) extent<3> extent;

     // These are restrict(amp,cpu)
    const T& operator[](const index<3>& idx) const restrict(amp,cpu);
    array_view<const T,2> operator[](int i) const restrict(amp,cpu);

    const T& operator()(const index<3>& idx) const restrict(amp,cpu);
    const T& operator()(int i0, int i1, int i2) const restrict(amp,cpu);

    array_view<const T,3> section(const index<3>& idx, const extent<3>& ext) const
restrict(amp,cpu);
    array_view<const T,3> section(const index<3>& idx) const restrict(amp,cpu);
    array_view<const T,3> section(const extent<3>& ext) const restrict(amp,cpu);
    array_view<const T,3> section(int i0, int i1, int i2, int e0, int e1, int e2) const
restrict(amp,cpu);

    void refresh() const;
 };
```

5.2.2 Constructors

The *array_view* type cannot be default-constructed. It must be bound at construction time to a contiguous data source.

No bounds-checking is performed when constructing *array_view*s.

`array_view<T,N>::array_view(array<T,N>& src) restrict(amp,cpu)` `array_view<const T,N>::array_view(const array<T,N>& src) restrict(amp,cpu)`	
Constructs an array_view which is bound to the data contained in the "src" array. The extent of the array_view is that of the src array, and the origin of the array view is at zero.	
Parameters:	
Src	An array which contains the data that this array_view is bound to.

`template <typename Container>`
`  array_view<T,N>::array_view(const extent<N>& extent, Container& src)`
`template <typename Container>`
`  array_view<const T,N>::array_view(const extent<N>& extent, const Container& src)`	

Constructs an array_view which is bound to the data contained in the "src" container. The extent of the array_view is that given by the "extent" argument, and the origin of the array view is at zero.	
Parameters:	
Src	A template argument that must resolve to a linear container that supports .data() and .size() members (such as std::vector or std::array)
Extent	The extent of this array_view.

`array_view<T,N>::array_view(const extent<N>& extent, value_type* src) restrict(amp,cpu)`
`array_view<const T,N>::array_view(const extent<N>& extent,`
`                                  const value_type* src) restrict(amp,cpu)`

Constructs an array_view which is bound to the data contained in the "src" container. The extent of the array_view is that given by the "extent" argument, and the origin of the array view is at zero.
Parameters:

Src	A pointer to the source data that will be copied into this array.
Extent	The extent of this array_view.

```
template <typename Container>
  array_view<T,1>::array_view(int e0, Container& src)
template <typename Container>
  array_view<T,2>::array_view(int e0, int e1, Container& src)
template <typename Container>
  array_view<T,3>::array_view(int e0, int e1, int e2, Container& src)

template <typename Container>
  array_view<const T,1>::array_view(int e0, const Container& src)
template <typename Container>
  array_view<const T,2>::array_view(int e0, int e1, const Container& src)
template <typename Container>
  array_view<const T,3>::array_view(int e0, int e1, int e2, const Container& src)
```

Equivalent to construction using "**array_view(extent<N>(e0 [, e1 [, e2]]), src)**".

Parameters:

e0 [, e1 [, e2]]	The component values that will form the extent of this array_view.
Src	A template argument that must resolve to a contiguous container that supports .data() and .size() members (such as std::vector or std::array)

```
array_view<T,1>::array_view(int e0, value_type* src) restrict(amp,cpu)
array_view<T,2>::array_view(int e0, int e1, value_type* src) restrict(amp,cpu)
array_view<T,3>::array_view(int e0, int e1, int e2, value_type* src) restrict(amp,cpu)

array_view<const T,1>::array_view(int e0, const value_type* src) restrict(amp,cpu)
array_view<const T,2>::array_view(int e0, int e1, const value_type* src) restrict(amp,cpu)
array_view<const T,3>::array_view(int e0, int e1, int e2,
                                  const value_type* src) restrict(amp,cpu)
```

Equivalent to construction using "**array_view(extent<N>(e0 [, e1 [, e2]]), src)**".

Parameters:

e0 [, e1 [, e2]]	The component values that will form the extent of this array_view.
Src	A pointer to the source data that will be copied into this array.

```
array_view(const array_view<T,N>& other) restrict(amp,cpu)
array_view(const array_view<const T,N>& other) restrict(amp,cpu);
```

Copy constructor. Constructs a new **array_view<T,N>** from the supplied argument other. A shallow copy is performed.

Parameters:

Other	An object of type **array_view<T,N>** or **array_view<const T,N>** from which to initialize this new array_view.

5.2.3 Members

```
__declspec(property(get)) extent<N> extent
extent<N> get_extent() const restrict(cpu,amp)
```

Access the extent that defines the shape of this array_view.

array_view& operator=(const array_view& other) restrict(amp,cpu)	
Assigns the contents of the array_view "other" to this array_view, using a shallow copy. Both array_views will refer to the same data.	
Parameters:	
other	An object of type **array_view<T,N>** from which to copy into this array.
Return Value:	
Returns ***this**.	

void copy_to(array<T,N>& dest)	
Copies the data referred to by this array_view to the array given by "dest", as if by calling "copy(*this, dest)" (see 5.3.2).	
Parameters:	
dest	An object of type **array <T,N>** to which to copy data from this array.

void copy_to(const array_view& dest)	
Copies the contents of this array_view to the array_view given by "dest", as if by calling "copy(*this, dest)" (see 5.3.2).	
Parameters:	
dest	An object of type **array_view<T,N>** to which to copy data from this array.

T* array_view<T,1>::data() const restrict(amp,cpu) const T* array_view<const T,1>::data() const restrict(amp,cpu)
Returns a pointer to the first data element underlying this array_view. This is only available on array_views of rank 1. When the data source of the array_view is native CPU memory, the pointer returned by data() is valid for the lifetime of the data source. When the data source underlying the array_view is an array, the pointer returned by data() in CPU context is ephemeral and is invalidated when the original data source or any of its views are accessed on an accelerator_view through a parallel_for_each or a copy operation.
Return Value:
A (const) pointer to the first element in the linearized array.

void array_view<T, N>::refresh() const void array_view<const T, N>::refresh() const
Calling this member function informs the array_view that its bound memory has been modified outside the array_view interface. This will render all cached information stale.

void array_view<T, N>::synchronize() const
Calling this member function synchronizes any modifications made to "this" array_view to its underlying data container. For example, for an array_view on system memory, if the contents of the view are modified on a remote accelerator_view through a parallel_for_each invocation, calling synchronize ensures that the modifications are synchronized to the source data and will be visible through the system memory pointer which the array_view was created over.

completion_future array_view<T, N>::synchronize_async() const
An asynchronous version of *synchronize*, which returns a completion future object. When the future is ready, the synchronization operation is complete.

void array_view<T, N>::discard_data() const
Indicates to the runtime that it may discard the current logical contents of this array_view. This is an optimization hint to the runtime used to avoid copying the current contents of the view to a target accelerator_view, and its use is recommended if the existing content is not needed.

5.2.4 Indexing

Accessing an *array_view* out of bounds yields undefined results.

`T& array_view<T,N>::operator[](const index<N>& idx) const restrict(amp,cpu)` `T& array_view<T,N>::operator()(const index<N>& idx) const restrict(amp,cpu)`	
Returns a reference to the element of this array_view that is at the location in N-dimensional space specified by "idx".	
Parameters:	
Idx	An object of type **`index<N>`** from that specifies the location of the element.

`const T& array_view<const T,N>::operator[](const index<N>& idx) const restrict(amp,cpu)` `const T& array_view<const T,N>::operator()(const index<N>& idx) const restrict(amp,cpu)`	
Returns a const reference to the element of this array_view that is at the location in N-dimensional space specified by "idx".	
Parameters:	
Idx	An object of type **`index<N>`** from that specifies the location of the element.

`T& array_view<T,1>::operator()(int i0) const restrict(amp,cpu)`
`T& array_view<T,1>::operator[](int i0) const restrict(amp,cpu)`
`T& array_view<T,2>::operator()(int i0, int i1) const restrict(amp,cpu)`
`T& array_view<T,3>::operator()(int i0, int i1, int i2) const restrict(amp,cpu)`	

Equivalent to "**`array_view<T,N>::operator()(index<N>(i0 [, i1 [, i2 ]]))`**".	
Parameters:	
i0 [, i1 [, i2]]	The component values that will form the index into this array.

`const T& array_view<const T,1>::operator()(int i0) const restrict(amp,cpu)`
`const T& array_view<const T,2>::operator()(int i0, int i1) const restrict(amp,cpu)`
`const T& array_view<const T,3>::operator()(int i0, int i1, int i2) const restrict(amp,cpu)`	

Equivalent to "**`array_view<T,N>::operator()(index<N>(i0 [, i1 [, i2 ]])) const`**".	
Parameters:	
i0 [, i1 [, i2]]	The component values that will form the index into this array.

`array_view<T,N-1> array_view<T,N>::operator[](int i0) const restrict(amp,cpu)` `array_view<const T,N-1> array_view<const T,N>::operator[](int i0) const restrict(amp,cpu)`	

This overload is defined for array_view<T,N> where N ≥ 2.
 	
This mode of indexing is equivalent to projecting on the most-significant dimension. It allows C-style indexing. For example:
 	
`array<float,4> myArray(myExtents, …);`
 	
`myArray[index<4>(5,4,3,2)] = 7;`
`assert(myArray[5][4][3][2] == 7);`	

Parameters:	
i0	An integer that is the index into the most-significant dimension of this array.
Return Value:	
Returns an array_view whose dimension is one lower than that of this array_view.	

5.2.5 View Operations

`array_view<T,N> array_view<T,N>::section(const index<N>& idx, const extent<N>& ext) const restrict(amp,cpu)` `array_view<const T,N> array_view<const T,N>::section(const index<N>& idx, const extent<N>& ext) const restrict(amp,cpu)`
Returns a subsection of the source array view at the origin specified by "idx" and with the extent specified by "ext

Example:

```
array<float,2> a(extent<2>(200,100));
array_view<float,2> v1(a); // v1.extent = <200,100>
array_view<float,2> v2 = v1.section(index<2>(15,25), extent<2>(40,50));
assert(v2(0,0) == v1(15,25));
```

Parameters:

idx	Provides the offset/origin of the resulting section.
ext	Provides the extent of the resulting section.

Return Value:

Returns a subsection of the source array at specified origin, and with the specified extent.

```
array_view<T,N> array_view<T,N>::section(const index<N>& idx) const restrict(amp,cpu)
array_view<const T,N> array_view<const T,N>::section(const index<N>& idx) const
restrict(amp,cpu)
```

Equivalent to "`section(idx, this->extent – idx)`".

```
array_view<T,N> array_view<T,N>::section(const extent<N>& ext) const restrict(amp,cpu)
array_view<const T,N> array_view<const T,N>::section(const extent<N>& ext) const
restrict(amp,cpu)
```

Equivalent to "`section(index<N>(), ext)`".

```
array_view<T,1> array_view<T,1>::section(int i0, int e0) const restrict(amp,cpu)
array_view<const T,1> array_view<const T,1>::section(int i0, int e0) const restrict(amp,cpu)

array_view<T,2> array_view<T,2>::section(int i0, int i1, int e0, int e1) const
restrict(amp,cpu)
array_view<const T,2> array_view<const T,2>::section(int i0, int i1,
                                          int e0, int e1) const restrict(amp,cpu)

array_view<T,3> array_view<T,3>::section(int i0, int i1, int i2,
                                         int e0, int e1, int e2) const restrict(amp,cpu)
array_view<const T,3> array_view<const T,3>::section(int i0, int i1, int i2,
                                          int e0, int e1, int e2) const restrict(amp,cpu)
```

Equivalent to "`section(index<N>(i0 [, i1 [, i2 ]]), extent<N>(e0 [, e1 [, e2 ]]))`".

Parameters:

i0 [, i1 [, i2]]	The component values that will form the origin of the section
e0 [, e1 [, e2]]	The component values that will form the extent of the section

```
template<typename ElementType>
  array_view<ElementType,1> array_view<T,1>::reinterpret_as() const restrict(amp,cpu)
template<typename ElementType>
  array_view<const ElementType,1> array_view<const T,1>::reinterpret_as() const
restrict(amp,cpu)
```

This member function is similar to "`array<T,N>::reinterpret_as`" (see 5.1.5), although it only supports array_views of rank 1 (only those guarantee that all elements are laid out contiguously).

The size of the reinterpreted ElementType must evenly divide into the total size of this array_view.

Return Value:

Returns an **array_view** from this **array_view<T,1>** with the element type reinterpreted from **T** to **ElementType**.

```
template <int K>
  array_view<T,K> array_view<T,1>::view_as(extent<K> viewExtent) const restrict(amp,cpu)
template <int K>
  array_view<const T,K> array_view<const T,1>::view_as(extent<K> viewExtent) const
restrict(amp,cpu)
```

This member function is similar to **array<T,N>::view_as**" (see 5.1.5), although it only supports array_views of rank 1 (only those guarantee that all elements are laid out contiguously).
Return Value:
Returns an **array_view** from this **array_view<T,1>** with the rank changed to K from 1.

5.3 Copying Data

C++ AMP offers a universal *copy* function which covers all synchronous data transfer requirements. In all cases, copying data is not supported while executing on an accelerator (in other words, the copy functions do not have a *restrict(amp)* clause). The general form of copy is:

```
copy(src, dest);
```

Informative: Note that this more closely follows the STL convention (destination is the last argument, as in std::copy) and is opposite of the C-style convention (destination is the first argument, as in memcpy).

Copying to *array* and *array_view* types is supported from the following sources:

- An *array* or *array_view* with the same rank and element type as the destination *array* or *array_view*.
- A standard container whose element type is the same as the destination *array* or *array_view*.

Informative: Containers that expose .size() and .data() members (e.g., std::vector, and std::array) can be handled more efficiently.

The copy operation always performs a deep copy.

Asynchronous copy has the same semantics as synchronous copy, except that they return a `completion_future` that can be waited on.

5.3.1 Synopsis

```
template <typename T, int N>
  void copy(const array<T,N>& src, array<T,N>& dest);
template <typename T, int N>
  void copy(const array<T,N>& src, const array_view<T,N>& dest);

template <typename T, int N>
  void copy(const array_view<const T,N>& src, array<T,N>& dest);
template <typename T, int N>
  void copy(const array_view<const T,N>& src, const array_view<T,N>& dest);

template <typename T, int N>
  void copy(const array_view<T,N>& src, array<T,N>& dest);
template <typename T, int N>
  void copy(const array_view<T,N>& src, const array_view<T,N>& dest);
```

```
template <typename InputIter, typename T, int N>
  void copy(InputIter srcBegin, InputIter srcEnd, array<T,N>& dest);
template <typename InputIter, typename T, int N>
  void copy(InputIter srcBegin, InputIter srcEnd, const array_view<T,N>& dest);

template <typename InputIter, typename T, int N>
  void copy(InputIter srcBegin, array<T,N>& dest);
template <typename InputIter, typename T, int N>
  void copy(InputIter srcBegin, const array_view<T,N>& dest);

template <typename OutputIter, typename T, int N>
  void copy(const array<T,N>& src, OutputIter destBegin);
template <typename OutputIter, typename T, int N>
  void copy(const array_view<T,N>& src, OutputIter destBegin);

template <typename T, int N>
  completion_future copy_async(const array<T,N>& src, array<T,N>& dest);
template <typename T, int N>
  completion_future copy_async(const array<T,N>& src, const array_view<T,N>& dest);

template <typename T, int N>
  completion_future copy_async(const array_view<const T,N>& src, array<T,N>& dest);
template <typename T, int N>
  completion_future copy_async(const array_view<const T,N>& src, const array_view<T,N>& dest);

template <typename T, int N>
  completion_future copy_async(const array_view<T,N>& src, array<T,N>& dest);
template <typename T, int N>
  completion_future copy_async(const array_view<T,N>& src, const array_view<T,N>& dest);

template <typename InputIter, typename T, int N>
  completion_future copy_async(InputIter srcBegin, InputIter srcEnd, array<T,N>& dest);
template <typename InputIter, typename T, int N>
  completion_future copy_async(InputIter srcBegin, InputIter srcEnd, const array_view<T,N>&
dest);

template <typename InputIter, typename T, int N>
  completion_future copy_async(InputIter srcBegin, array<T,N>& dest);
template <typename InputIter, typename T, int N>
  completion_future copy_async(InputIter srcBegin, const array_view<T,N>& dest);

template <typename OutputIter, typename T, int N>
  completion_future copy_async(const array<T,N>& src, OutputIter destBegin);
template <typename OutputIter, typename T, int N>
  completion_future copy_async(const array_view<T,N>& src, OutputIter destBegin);
```

5.3.2 Copying between array and array_view

An *array<T,N>* can be copied to an object of type *array_view<T,N>*, and vice versa.

```
template <typename T, int N>
  void copy(const array<T,N>& src, array<T,N>& dest)

template <typename T, int N>
  completion_future copy_async(const array<T,N>& src, array<T,N>& dest)
```

The contents of "src" are copied into "dest". The source and destination may reside on different accelerators. If the extents of "src" and "dest" don't match, a runtime exception is thrown.	
Parameters:	
Src	An object of type **array<T,N>** to be copied from.
Dest	An object of type **array<T,N>** to be copied to.

```
template <typename T, int N>
  void copy(const array<T,N>& src, const array_view<T,N>& dest)

template <typename T, int N>
  completion_future copy_async(const array<T,N>& src, const array_view<T,N>& dest)
```

The contents of "src" are copied into "dest". If the extents of "src" and "dest" don't match, a runtime exception is thrown.	
Parameters:	
src	An object of type **array<T,N>** to be copied from.
dest	An object of type **array_view<T,N>** to be copied to.

```
template <typename T, int N>
  void copy(const array_view<const T,N>& src, array<T,N>& dest)

template <typename T, int N>
  void copy(const array_view<T,N>& src, array<T,N>& dest)

template <typename T, int N>
  completion_future copy_async(const array_view<const T,N>& src, array<T,N>& dest)

template <typename T, int N>
  completion_future copy_async(const array_view<T,N>& src, array<T,N>& dest)
```

The contents of "src" are copied into "dest". If the extents of "src" and "dest" don't match, a runtime exception is thrown.	
Parameters:	
src	An object of type **array_view<T,N>** (or **array_view<const T,N>**) to be copied from.
dest	An object of type **array<T,N>** to be copied to.

```
template <typename T, int N>
  void copy(const array_view<const T,N>& src, const array_view<T,N>& dest)

template <typename T, int N>
  completion_future copy_async(const array_view<const T,N>& src, const array_view<T,N>& dest)
```

The contents of "src" are copied into "dest". If the extents of "src" and "dest" don't match, a runtime exception is thrown.	
Parameters:	
src	An object of type **array_view<T,N>** (or **array_view<const T,N>**) to be copied from.
dest	An object of type **array_view<T,N>** to be copied to.

5.3.3 Copying from standard containers to arrays or array_views

A standard container can be copied into an *array* or *array_view* by specifying an iterator range.

Informative: Standard containers that present a .size() and a .data() (such as std::vector and std::array) operation can be handled very efficiently.

```
template <typename InputIter, typename T, int N>
  void copy(InputIter srcBegin, InputIter srcEnd, array<T,N>& dest)

template <typename InputIter, typename T, int N>
  void copy(InputIter srcBegin, array<T,N>& dest)

template <typename InputIter, typename T, int N>
  completion_future copy_async(InputIter srcBegin, InputIter srcEnd, array<T,N>& dest)

template <typename InputIter, typename T, int N>
  completion_future copy_async(InputIter srcBegin, array<T,N>& dest)
```

The contents of a source container from the iterator range [srcBegin,srcEnd) are copied into “dest”. If the number of elements in the iterator range is not equal to “dest.extent.size()”, an exception is thrown.

In the overloads which don’t take an end-iterator it is assumed that the source iterator is able to provide at least dest.extent.size() elements, but no checking is performed (nor possible).

Parameters:	
srcBegin	An iterator to the first element of a source container.
srcEnd	An iterator to the end of a source container.
dest	An object of type **array<T,N>** to be copied to.

```
template <typename InputIter, typename T, int N>
  void copy(InputIter srcBegin, InputIter srcEnd, const array_view<T,N>& dest)

template <typename InputIter, typename T, int N>
  void copy(InputIter srcBegin, const array_view<T,N>& dest)


template <typename InputIter, typename T, int N>
  completion_future copy_async(InputIter srcBegin, InputIter srcEnd, const array_view<T,N>&
dest)

template <typename InputIter, typename T, int N>
  completion_future copy_async(InputIter srcBegin, const array_view<T,N>& dest)
```

The contents of a source container from the iterator range [srcBegin,srcEnd) are copied into “dest”. If the number of elements in the iterator range is not equal to “dest.extent.size()”, an exception is thrown.

Parameters:	
srcBegin	An iterator to the first element of a source container.
srcEnd	An iterator to the end of a source container.
Dest	An object of type **array_view<T,N>** to be copied to.

5.3.4 Copying from arrays or array_views to standard containers

An array or array_view can be copied into a standard container by specifying the begin iterator. Standard containers that present a **.size()** and a **.data()** (such as **std::vector** and **std::array**) operation can be handled very efficiently.

```
template <typename OutputIter, typename T, int N>
  void copy(const array<T,N>& src, OutputIter destBegin)

template <typename OutputIter, typename T, int N>
  completion_future copy_async(const array<T,N>& src, OutputIter destBegin)
```

The contents of a source array are copied into "dest" starting with iterator destBegin. If the number of elements in the range starting destBegin in the destination container is smaller than "src.extent.size()", an exception is thrown.

Parameters:

src	An object of type **array<T,N>** to be copied from.
destBegin	An output iterator addressing the position of the first element in the destination container.

```
template <typename OutputIter, typename T, int N>
  void copy(const array_view<T,N>& src, OutputIter destBegin)

template <typename OutputIter, typename T, int N>
  completion_future copy_async(const array_view<T,N>& src, OutputIter destBegin)
```

The contents of a source array are copied into "dest" starting with iterator destBegin. If the number of elements in the range starting destBegin in the destination container is smaller than "src.extent.size()", an exception is thrown.

Parameters:

src	An object of type **array_view<T,N>** to be copied from.
destBegin	An output iterator addressing the position of the first element in the destination container.

6 Atomic Operations

C++ AMP provides a set of atomic operations in the *concurrency* namespace. These operations are applicable in *restrict(amp)* contexts and may be applied to memory locations within *concurrency::array* instances and to memory locations within *tile_static* variables. Section 8 provides a full description of the C++ AMP memory model and how atomic operations fit into it.

6.1 Synposis

```
int atomic_exchange(int * dest, int val) restrict(amp)
unsigned int atomic_exchange(unsigned int * dest, unsigned int val) restrict(amp)
float atomic_exchange(float * dest, float val) restrict(amp)

bool atomic_compare_exchange(int * dest, int * expected_value, int val) restrict(amp)
bool atomic_compare_exchange(unsigned int * dest, unsigned int * expected_value, unsigned int
val) restrict(amp)

int atomic_fetch_add(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_add(unsigned int * dest, unsigned int val) restrict(amp)

int atomic_fetch_sub(int * dest, int val) restrict(amp)
```

```
unsigned int atomic_fetch_sub(unsigned int * dest, unsigned int val) restrict(amp)

int atomic_fetch_max(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_max(unsigned int * dest, unsigned int val)

int atomic_fetch_min(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_min(unsigned int * dest, unsigned int val)

int atomic_fetch_and(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_and(unsigned int * dest, unsigned int val)

int atomic_fetch_or(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_or(unsigned int * dest, unsigned int val)

int atomic_fetch_xor(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_xor(unsigned int * dest, unsigned int val) restrict(amp)

int atomic_fetch_inc(int * dest) restrict(amp)
unsigned int atomic_fetch_inc(unsigned int * dest) restrict(amp)

int atomic_fetch_dec(int * dest) restrict(amp)
unsigned int atomic_fetch_dec(unsigned int * dest) restrict(amp)
```

6.2 Atomically Exchanging Values

```
int atomic_exchange(int * dest, int val) restrict(amp)
unsigned int atomic_exchange(unsigned int * dest, unsigned int val) restrict(amp)
float atomic_exchange(float * dest, float val) restrict(amp)
```

Atomically read the value stored in *dest*, replace it with the value given in *val* and return the old value to the caller. This function provides overloads for *int*, *unsigned int* and *float* parameters.

Parameters:	
dst	An pointer to the location which needs to be atomically modified. The location may reside within a *concurrency::array* or within a *tile_static* variable.
val	The new value to be stored in the location pointed to be *dst*.

Return value:

These functions return the old value which was previously stored at *dst*, and that was atomically replaced. These functions always succeed.

```
bool atomic_compare_exchange(int * dest, int * expected_val, int val) restrict(amp)
bool atomic_compare_exchange(unsigned int * dest, unsigned int * expected_val, unsigned int
val) restrict(amp)
```

These functions attempt to atomically perform these three steps atomically:

1. Read the value stored in the location pointed to by *dest*
2. Compare the value read in the previous step with the value contained in the location pointed by *expected_val*
3. Carry the following operations depending on the result of the comparison of the previous step:
   a. If the values are identical, then the function tries to atomically change the value pointed by *dest* to the value in *val*. The function indicates by its return value whether this transformation has been successful or not.
   b. If the values are not identical, then the function stores the value read in step (1) into the location pointed to by *expected_val*, and returns *false*.

In terms of sequential semantics, the function is equivalent to the following pseudo-code:

```
auto t = *dest;
bool eq = t == *expected_val;
if (eq)
    *dst = val;
*expected_val = t;
return eq;
```

The function may fail spuriously. It is guaranteed that the system as a whole will make progress when threads are contending to atomically modify a variable, but there is no upper bound on the number of failed attempts that any particular thread may experience.

Parameters:

dst	An pointer to the location which needs to be atomically modified. The location may reside within a *concurrency::array* or within a *tile_static* variable.
expected_val	A pointer to a local variable or function parameter. Upon calling the function, the location pointed by *expected_val* contains the value the caller expects *dst* to contain. Upon return from the function, *expected_val* will contain the most recent value read from *dst*.
val	The new value to be stored in the location pointed to be *dst*.

Return value:

The return value indicates whether the function has been successful in atomically reading, comparing and modifying the contents of the memory location.

6.3 Atomically Applying an Integer Numerical Operation

```
int atomic_fetch_add(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_add(unsigned int * dest, unsigned int val) restrict(amp)

int atomic_fetch_sub(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_sub(unsigned int * dest, unsigned int val) restrict(amp)

int atomic_fetch_max(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_max(unsigned int * dest, unsigned int val)

int atomic_fetch_min(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_min(unsigned int * dest, unsigned int val)

int atomic_fetch_and(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_and(unsigned int * dest, unsigned int val)

int atomic_fetch_or(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_or(unsigned int * dest, unsigned int val)

int atomic_fetch_xor(int * dest, int val) restrict(amp)
unsigned int atomic_fetch_xor(unsigned int * dest, unsigned int val) restrict(amp)
```

Atomically read the value stored in *dest*, apply the binary numerical operation specific to the function with the read value and *val* serving as input operands, and store the result back to the location pointed by *dest*.

In terms of sequential semantics, the operation performed by any of the above function is described by the following piece of pseudo-code:

```
*dest = *dest ⊗ val;
```

Where the operation denoted by ⊗ is one of: addition (atomic_fetch_add), subtraction (atomic_fetch_sub), find maximum (atomic_fetch_max), find minimum (atomic_fetch_min), bit-wise AND (atomic_fetch_and), bit-wise OR (atomic_fetch_or), bit-wise XOR (atomic_fetch_or).

Parameters:	
Dst	An pointer to the location which needs to be atomically modified. The location may reside within a *concurrency::array* or within a *tile_static* variable.
val	The second operand which participates in the calculation of the binary operation whose result is stored into the location pointed to be *dst*.
Return value:	
These functions return the old value which was previously stored at *dst*, and that was atomically replaced. These functions always succeed.	

```
int atomic_fetch_inc(int * dest) restrict(amp)
unsigned int atomic_fetch_inc(unsigned int * dest) restrict(amp)

int atomic_fetch_dec(int * dest) restrict(amp)
unsigned int atomic_fetch_dec(unsigned int * dest) restrict(amp)
```

Atomically increment or decrement the value stored at the location point to by *dest*.	
Parameters:	
Dst	An pointer to the location which needs to be atomically modified. The location may reside within a *concurrency::array* or within a *tile_static* variable.
Return value:	
These functions return the old value which was previously stored at *dst*, and that was atomically replaced. These functions always succeed.	

7 Launching Computations: parallel_for_each

Developers using C++ AMP will use a form of *parallel_for_each()* to launch data-parallel computations on accelerators. The behavior of *parallel_for_each* is similar to that of *std::for_each*: execute a function for each element in a range. The C++ AMP specialization over ranges of type *extent* and *tiled_extent* allow execution of functions on accelerators.

The *parallel_for_each* function takes the following general forms:

1. Non-tiled:
```
template <int N, typename Kernel>
void parallel_for_each(extent<N> compute_domain, const Kernel& f);
```

2. Tiled:
```
template <int D0, int D1, int D2, typename Kernel>
void parallel_for_each(tiled_extent<D0,D1,D2> compute_domain, const Kernel& f);

template <int D0, int D1, typename Kernel>
void parallel_for_each(tiled_extent<D0,D1> compute_domain, const Kernel& f);

template <int D0, typename Kernel>
void parallel_for_each(tiled_extent<D0> compute_domain, const Kernel& f);
```

A *parallel_for_each* invocation may be explicitly requested on a specific accelerator view

1. Non-tiled:
```
template <int N, typename Kernel>
void parallel_for_each(const accelerator_view& accl_view,
                       extent<N> compute_domain, const Kernel& f);
```

2. Tiled:

```
template <int D0, int D1, int D2, typename Kernel>
void parallel_for_each(const accelerator_view& accl_view,
                       tiled_extent<D0,D1,D2> compute_domain, const Kernel& f);

template <int D0, int D1, typename Kernel>
void parallel_for_each(const accelerator_view& accl_view,
                       tiled_extent<D0,D1> compute_domain, const Kernel& f);

template <int D0, typename Kernel>
void parallel_for_each(const accelerator_view& accl_view,
                       tiled_extent<D0> compute_domain, const Kernel& f);
```

A *parallel_for_each* over an *extent* represents a dense loop nest of independent serial loops.

When *parallel_for_each* executes, a parallel activity is spawned for each index in the compute domain. Each parallel activity is associated with an index value. (This index is an *index<N>* in the case of a non-tiled *parallel_for_each*, or a *tiled_index<D0,D1,D2>* in the case of a tiled *parallel_for_each*.) A parallel activity typically uses its index to access the appropriate locations in the input/output arrays.

A call to *parallel_for_each* behaves as if it were synchronous. In practice, the call may be asynchronous because it executes on a separate device, but since data copy-out is a synchronizing event, the developer cannot tell the difference.

There are no guarantees on the order and concurrency of the parallel activities spawned by the non-tiled *parallel_for_each*. Thus it is not valid to assume that one activity can wait for another sibling activity to complete for itself to make progress. This is discussed in further detail in section 8.

The tiled version of *parallel_for_each* organizes the parallel activities into fixed-size tiles of 1, 2, or 3 dimensions, as given by the *tiled_extent<>* argument. The *tiled_extent* provided as the first parameter to *parallel_for_each* must be divisible, along each of its dimensions, by the respective tile extent. Tiling beyond 3 dimensions is not supported. Threads (parallel activities) in the same tile have access to shared *tile_static* memory, and can use *tiled_index::barrier.wait* (4.5.3) to synchronize access to it.

When launching an *amp*-restricted kernel, the implementation of tiled *parallel_for_each* will provide the following minimum capabilities:

- The maximum number of tiles per dimension will be no less than 65535.
- The maximum number of threads in a tile will be no less than 1024.
    - In 3D tiling, the maximal value of D0 will be no less than 64.

Microsoft-specific:
When launching an amp-restricted kernel, the tiled parallel_for_each provides the above portable guarantees and no more. i.e.,

- *The maximum number of tiles per dimension is 65535.*
- *The maximum nuimber of threads in a tile is 1024*
    - *In 3D tiling, the maximum value supported for D0 is 64.*

The execution behind the *parallel_for_each* occurs on a certain accelerator, in the context of a certain accelerator view. This accelerator view may be passed explicitly to *parallel_for_each* (as an optional first argument). Otherwise, the target accelerator and the view using which work is submitted to the accelerator, is chosen from the objects of type *array<T,N>* and *texture<T>* that were captured in the kernel lambda. An implementation may require that all arrays and textures captured in the lambda must be on the same accelerator view; if not, an implemention is free to throw an exception. An implementation may also arrange for the specified data to be accessible on the selected accelerator view, rather than reject the call.

Microsoft-specific: the Microsoft implementation of C++ AMP requires that all array and texture objects are co-located on the same accelerator view which is used, implicitly or explicitly in a parallel_for_each call.

If the parallel_for_each kernel functor does not capture an array/texture object and neither is the target accelerator_view for the kernel's execution is explicitly specified, the runtime is allowed to execute the kernel on any accelerator_view on the default accelerator.

Microsoft-specific: In such a scenario, the Microsoft implementation of C++ AMP selects the target accelerator_view for executing the parallel_for_each kernel as follows:

a. *Determine the set of accelerator_views where ALL array_views referenced in the p_f_e kernel have cached copies*
b. *From the above set, filter out any accelerator_views that are not on the default accelerator. Additionally filter out accelerator_views that do not have the capabilities required by the p_f_e kernel (debug intrinsics, number of UAVs)*
c. *The default accelerator_view of the default accelerator is selected as the target, if the resultant set from b. is empty, or contains, that accelerator_view*

Otherwise, any accelerator_view from the resultant set from b., is arbitrarily selected as the target
The *tiled_index<>* argument passed to the kernel contains a collection of indices including those that are relative to the current tile.

The argument *f* of template-argument type *Kernel* to the *parallel_for_each* function must be a lambda or functor offering an appropriate function call operator which the implementation of *parallel_for_each* invokes with the instantiated index type. To execute on an accelerator, the function call operator must be marked *restrict(amp)* (but may have additional restrictions), and it must be callable from a caller passing in the instantiated index type. Overload resolution is handled as if the caller contained this code:

```
template <typename IndexType, typename Kernel>
void parallel_for_each_stub(IndexType i, const Kernel& f) restrict(amp)
{
    f(i);
}
```

Where the *Kernel f* argument is the same one passed into *parallel_for_each* by the caller, and the index instance *i* is the thread identifier, where *IndexType* is the following type:

- Non-Tiled *parallel_for_each*: *index<N>,* where *N* must be the same rank as the *extent<N>* used in the *parallel_for_each*.
- Tiled *parallel_for_each*: *tiled_index<D0 [, D1 [, D2]]>,* where the tile extents must match those of the *tiled_extent* used in the *parallel_for_each.*

The value returned by the kernel function, if any, is ignored.

Microsoft-specific:
In the Microsoft implementation of C++ AMP, every function that is referenced directly or indirectly by the kernel function, as well as the kernel function itself, must be inlineable[4].

[4] An implementation can employ whole-program compilation (such as link-time code-gen) to achieve this.

7.1 Capturing Data in the Kernel Function Object

Since the kernel function object does not take any other arguments, all other data operated on by the kernel, other than the thread index, must be captured in the lambda or function object passed to *parallel_for_each*. The function object shall be any amp-compatible class, struct or union type, including those introduced by lambda expressions.

7.2 Exception Behaviour

If an error occurs trying to launch the *parallel_for_each*, an exception will be thrown. Exceptions can be thrown the following reasons:

1. Failure to create shader
2. Failure to create buffers
3. Invalid extent passed
4. Mismatched accelerators

8 Correctly Synchronized C++ AMP Programs

Correctly synchronized C++ AMP programs are correctly synchronized C++ programs which also adhere to a few additional C++ AMP rules, as follows:

1. Accelerator-side execution
    a. Concurrency rules for arbitrary sibling theads launched by a *parallel_for_each* call.
    b. Semantics and correctness of tile barriers.
    c. Semantics of atomic and memory fence operations.
2. Host-side execution
    a. Concurrency of accesses to C++ AMP containers between host-side operations: *copy*, *synchronize*, *parallel_for_each* and the application of the various subscript operators of arrays and array views on the host.
    b. Accessing *array*s or *array_view* data on the host.

8.1 Concurrency of sibling threads launched by a parallel_for_each call

In this section we will consider the relationship between sibling threads in a single *parallel_for_each* call. Interaction between separate *parallel_for_each* calls, copy operations and other host-side operations will be considered in the following sub-sections.

A *parallel_for_each* call logically initiates the operation of multiple sibling threads, one for each coordinate in the *extent* or *tiled_extent* passed to it.

All the threads launched by a *parallel_for_each* are potentially concurrent. Unless barriers are used, an implementation is free to schedule these threads in any order. In addition, the memory model for normal memory accesses is weak, that is operations could be arbitrarily reordered as long as each thread perceives to execute in its original program order. Thus any two memory operations from any two threads in a *parallel_for_each* are by default concurrent, unless the application has explicitly enforced an order between these two operations using atomic operations, fences or barriers.

Conversely, an implementation may also schedule only a single logical thread at a time, in a non-cooperative manner, i.e., without letting any other threads make any progress, with the exception of hitting a tile barrier or terminating. When a thread encounters a tile barrier, an implementation must wrest control from that thread and provide progress to some other thread in the tile until they all have reached the barrier. Similarly, when a thread finishes execution, the system is obligated to execute steps from some other thread. Thus an implementation is obligated to switch context between threads only when a thread has hit a barrier (barriers pertain just to the tiled *parallel_for_each*), or is finished. An implementation doesn't have to admit any concurrency at a finer level than that which is dictated by barriers and thread termination. All implementations, however, are obligated to ensure progress is continually made, until all threads launched by a *parallel_for_each* are completed.

An immediate corollary is that C++ AMP doesn't provide a mechanism using which a thread could, without using tile barriers, poll for a change which needs to be effected by another thread. In particular, C++ AMP doesn't support locks which are implemented using atomic operations and fences, since a thread could end up polling forever, waiting for a lock to become available. The usage of tile barriers allows for creating a limited form of locking scoped to a thread tile. For example:

```
void tile_lock_example()
{
  parallel_for_each(
    extent<1>(TILE_SIZE).tile<TILE_SIZE>(),
    [] (tiled_index<TILE_SIZE> tidx) restrict(amp)
    {
      tile_static int lock;

      // Initialize lock:
      if (tidx.local[0] == 0) lock = 0;
      tidx.barrier.wait();

      bool performed_my_exclusive_work = false;
      for (;;) {
        // try to acquire the lock
        if (!performed_my_ exclusive _work  && atomic_compare_exchange(&lock, 0, 1)) {
          // The lock has been acquired - mutual exclusion from the rest of the threads in the tile
          // is provided here....
          some_synchronized_op();

          // Release the lock
          atomic_exchange(&lock, 0);
          performed_my_exclusive_work = true;
        }
        else {
          // The lock wasn't acquired, or we are already finished. Perhaps we can do something
          // else in the meanwhile.
          some_non_exclusive_op();
        }

        // The tile barrier ensures progress, so threads can spin in the for loop until they
        // are successful in acquiring the lock.
        tidx.barrier.wait();
      }
    });
}
```

Informative: More often than not, such non-deterministic locking within a tile is not really necessary, since a static schedule of the threads based on integer thread ID's is possible and results in more efficient and more maintainable code, but we bring this example here for completeness and to illustrate a valid form of polling.

8.1.1 Correct usage of tile barriers

Correct C++ AMP programs require all threads in a tile to hit all tile barriers uniformly. That is, at a minimum, when a thread encounters a particular *tile_barrier::wait* call site (or any other barrier method of class *tile_barrier*), all other threads in the tile must encounter the same call site.

Informative: This requirement, however, is typically not sufficient in order to allow for efficient implementations. For example, it allows for the call stack of threads to differ, when they hit a barrier. In order to be able to generate good quality code for vector targets, much stronger constraints should be placed on the usage of barriers, as explained below.

C++ AMP requires all *active control flow expressions* leading to a tile barrier to be *tile-uniform*. Active control flow expressions are those guarding the scopes of all control flow constructs and logical expressions, which are actively being executed at a time a barrier is called. For example, the condition of an *if* statement is an active control flow expression as long as either the true or false hands of the *if* statement are still executing. If either of those hands contains a tile barrier, or leads to one through an arbitrary nesting of scopes and function calls, then the control flow expression controlling the *if* statement must be *tile-uniform*. What follows is an exhaustive list of control flow constructs which may lead to a barrier and their corresponding control expressions:

```
if (<control-expression>) <statement> else <statement>
switch (<control-expression> { <cases> }
for (<init-expression>; <control-expression>; <iteration-expression>) <statement>
while (<control-expression>) <statement>
do <statement> while(<control-expression>);
<control-expression> ? <expression> : <expression>
<control-expression> && <expression>
<control-expression> || <expression>
```

All active control flow constructs are strictly nested in accordance to the program's text, starting from the scope of the lambda at the *parallel_for_each* all the way to the scope containing the barrier.

C++ AMP requires that, when a barrier is encountered by one thread:

1. That the same barrier will be encountered by all other threads in the tile.
2. That the sequence of active control flow statements and/or expressions be identical for all threads when they reach the barrier.
3. That each of the corresponding control expressions be *tile-uniform* (which is defined below).
4. That any active control flow statement or expression hasn't been departed (necessarily in a non-uniform fashion) by a *break*, *continue* or *return* statement. That is, any breaking statement which instructs the program to leave an active scope must in itself behave as if it was a barrier, i.e., adhere to these preceding rules.

Informally, a *tile-uniform expression* is an expression only involving variables, literals and function calls which have a uniform value throughout the tile. Formally, C++ AMP specifies that:

5. *Tile-uniform* expressions may reference literals and template parameters
6. *Tile-uniform* expressions may reference *const* (or effectively *const*) data members of the function object parameter of *parallel_for_each*
7. *Tile-uniform* expressions may reference *tiled_index<,,>::tile*
8. *Tile-uniform* expressions may reference values loaded from *tile_static* variables as long as those values are loaded immediately and uniformly after a tile barrier. That is, if the barrier and the load of the value occur at the same function and the barrier dominates the load and no potential store into the same *tile_static* variable intervenes between the barrier and the load, then the loaded value will be considered *tile-uniform*
9. Control expressions may reference *tile-uniform local variables and parameters*. Uniform local variables and parameters are variables and parameters which are always initialized and assigned-to under uniform control flow (that is, using the same rules which are defined here for barriers) and which are only assigned *tile-uniform* expressions
10. *Tile-uniform* expressions may reference the return values of functions which return *tile-uniform* expressions
11. *Tile-uniform* expressions may not reference any expression not explicitly listed by the previous rules

An implementation is not obligated to warn when a barrier does not meet the criteria set forth above. An implementation may disqualify the compilation of programs which contain incorrect barrier usage. Conversely, an implementation may accept programs containing incorrect barrier usage and may execute them with undefined behavior.

8.1.2 Establishing order between operations of concurrent `parallel_for_each` threads

Threads may employ atomic operations, barriers and fences to establish a happens-before relationship encompassing their cumulative execution. When considering the correctness of the synchronization of programs, the following three aspects of the programs are relevant:

1. The types of memory which are potentially accessed concurrently by different threads. The memory type can be:
    a. Global memory
    b. Tile-static memory
2. The relationship between the threads which could potentially access the same piece of memory. They could be:
    a. Within the same thread tile

    b. Within separate threads tiles or sibiling threads in the basic (non-tiled) parallel_for_each model.
3. Memory operations which the program contains:
    a. Normal memory reads and writes.
    b. Atomic read-modify-write operations.
    c. Memory fences and barriers

Informally, the C++ AMP memory model is a weak memory model consistent with the C++ memory model, with the following exceptions:

1. Atomic operations do not necessarily create a sequentially consistent subset of execution. Atomic operations are only coherent, not sequentially consistent. That is, there doesn't necessarily exist a global linear order containing all atomic operations affecting all memory locations which were subjects of such operations. Rather, a separate global order exists for each memory location, and these per-location memory orders are not necessarily combinable into a single global order. (Note: this means an atomic operation does not constitute a memory fence.)
2. Memory fence operations are limited in their effects to the thread tile they are performed within. When a thread from tile A executes a fence, the fence operation doesn't necessarily affect any other thread from any tile other than A.
3. As a result of (1) and (2), the only mechanism available for cross-tile communication is atomic operations, and even when atomic operations are concerned, a linear order is only guaranteed to exist on a per-location basis, but not necessarily globally.
4. Fences are bi-directional, meaning they have both acquire and release semantics.
5. Fences can also be further scoped to a particular memory type (global vs. tile-static).
6. Applying normal stores and atomic operations concurrently to the same memory location results in undefined behavior.
7. Applying a normal load and an atomic operation concurrently to the same memory location is allowed (i.e., results in defined bavior).

We will now provide a more formal characterization of the different categories of programs based on their adherence to synchronization rules. The three classes of adherence are

1. *barrier-incorrect* programs,
2. *racy programs,* and,
3. *correctly-synchronized programs*.

8.1.2.1 Barrier-incorrect programs

A *barrier-incorrect* program is a program which doesn't adhere to the correct barrier usage rules specified in the previous section. Such programs always have undefined behavior. The remainder of this section discusses barrier-correct programs only.

8.1.2.2 Compatible memory operations

The following definition is later used in the definition of racy programs.

Two memory operations applied to the same (or overlapping) memory location are *compatible* if they are both aligned and have the same data width, and either both operations are reads, or both operation are atomic, or one operation is a read and the other is atomic.

This is summarized by the following table in which T_1 is a thread executing Op_1 and T_2 is a thread executing operation Op_2.

Op_1	Op_2	Compatible?
Atomic	Atomic	Yes
Read	Read	Yes
Read	Atomic	Yes

Write	Any	No

8.1.2.3 Concurrent memory operations

The following definition is later used in the definition of racy programs.

Informally, two memory operations by different threads are considered *concurrent* if no order has been established between them. Order can be established between two memory operations only when they are executed by threads within the same tile. Thus any two memory operations by threads from different tiles are always concurrent, even if they are atomic. Within the same tile, order is established using fences and barriers. Barriers are a strong form of a fence.

Formally, Let $\{T_1,...,T_N\}$ be the threads of a tile. Fix a sharable memory type (be it global or tile-static). Let M be the total set of memory operations of the given memory type performed by the collective of the threads in the tile.

Let $F = \langle F_1,\ldots,F_L \rangle$ be the set of memory fence operations of the given memory type, performed by the collective of threads in the tile, and organized arbitrarily into an ordered sequence.

Let P be a partitioning of M into a sequence of subsets $P = \langle M_0,\ldots,M_L \rangle$, organized into an ordered sequence in an arbitrary fashion.

Let S be the interleaving of F and P, $S = \langle M_0,F_1,M_1,\ldots,F_L,M_L \rangle$

S is *conforming* if both of these conditions hold:

1. **Adherence to program order**: For each T_i, S respects the fences performed[5] by T_i. That is any operation performed by T_i before T_i performed fence F_j appears strictly before F_j in S, and similarly any operations performed by T_i after F_j appears strictly after F_j in S.
2. **Self-consistency**: For $i<j$, let M_i be a subset containing at least one store (atomic or non-atomic) into location L and let M_j be a subset containing at least a single load of L, and no stores into L. Further assume that no subset in-between M_i and M_j stores into L. Then S provides that all loads in M_j shall:
   a. Return values stored into L by operations in M_i, and
   b. For each thread T_i, the subset of T_i operations in M_j reading L shall all return the same value (which is necessarily one stored by an operation in M_i, as specified by condition (a) above).
3. **Respecting initial values**. Let M_j be a subset containing a load of L, and no stores into L. Further assume that there is no M_i where $i<j$ such that M_i contains a store into L. Then all loads of L in M_j will return the initial value of L.

In such a conforming sequence S, two operations are *concurrent* if they have been executed by different threads and they belong to some common subset M_i. Two operations are *concurrent in an execution history* of a tile, if there exists a conforming interleaving S as described herein in which the operations are concurrent. Two operations of a program are *concurrent* if there possibly exists an execution of the program in which they are concurrent.

A barrier behaves like a fence to establish order between operations, except it provides additional guarantees on the order of execution. Based on the above definition, a barrier is like a fence that only permits a certain kind of interleaving. Specifically, one in which the sequence of fences (F in the above formalization) has the fences , corresponding to the barrier execution by individual threads, appearing uninterrupted in S, without any memory operations interleaved between them. For example, consider the following program:

```
C1
Barrier
C2
```

[5] Here, performance of memory operations is assumed to strictly follow program order.

Assume that C1 and C2 are arbitrary sequences of code. Assume this program is executed by two threads T1 and T2, then the only possible conforming interleavings are given by the following pattern:

T1(C1) || T2(C1)
T1(Barrier) || T2(Barrier)
T1(C2) || T2(C2)

Where the || operator implies arbitrary interleaving of the two operand sequences.

8.1.2.4 Racy programs

Racy programs are programs which have possible executions where at least two operations performed by two separate threads are both (a) incompatible AND (b) concurrent.

Racy programs do not have semantics assigned to them. They have undefined behavior.

8.1.2.5 Race-free programs

Race-free programs are, simply, programs that are not racy. Race-free programs have the following semantics assigned to them:

1. If two memory operations are ordered (i.e., not concurrent) by fences and/or barriers, then the values loaded/stored will respect such an ordering.
2. If two memory operations are concurrent then they must be atomic and/or reads performed by threads within the same tile. For each memory location X there exists an eventual total order including all such operations concurrent opertions applied to X and obeying the semantics of loads and atomic read-modify-write transactions.

8.2 Cumulative effects of a `parallel_for_each` call

An invocation of *parallel_for_each* receives a function object, the contents of which are made available on the device. The function object may contain: *concurrency::array* reference data members, *concurrency::array_view* value data members, *concurrency::graphics::texture* reference data members, and *concurrency::graphics::writeonly_texture_view* value data members. (In addition, the function object may also contain additional, user defined data members.) Each of these members of the types *array*, *array_view*, *texture* and *write_only_texture_view*, could be constrained in the type of access it provides to kernel code. For example an *array<int,2>&* member provides both read and write access to the array, while a *const array<int,2>&* member provides just read access to the array. Similarly, an *array_view<int,2>* member provides read and write access, while an *array_view<const int,2>* member provides read access only.

The C++ AMP specification permits implementations in which the memory backing an *array*, *array_view* or *texture* could be shared between different accelerators, and possibly also the host, while also permitting implementations where data has to be copied, by the implementation, between different memory regions in order to support access by some hardware. Simulating coherence at a very granular level is too expensive in the case disjoint memory regions are required by the hardware. Therefore, in order to support both styles of implementation, this specification stipulates that *parallel_for_each* has the freedom to implement coherence over *array*, *array_view*, and *texture* using coarse copying. Specifically, while a *parallel_for_each* call is being evaluated, implementations may:

1. Load and/or store any location, in any order, any number of times, of each container which is passed into *parallel_for_each* in read/write mode.
2. Load from any location, in any order, any number of times, of each container which is passed into *parallel_for_each* in read-only mode.

A *parallel_for_each* always behaves synchronously. That is, any observable side effects caused by any thread executing within a *parallel_for_each* call, or any side effects further affected by the implementation, due to the freedom it has in moving memory around, as stipulated above, shall be visible by the time *parallel_for_each* return.

However, since the effects of *parallel_for_each* are constrained to changing values within *array*s, *array_view*s and *texture*s and each of these objects can synchronize its contents lazily upon access, an asynchronous implementation of *parallel_for_each* is possible, and encouraged. Nonetheless, implementations should still honor calls to *accelerator_view::wait* by blocking until all lazily queued side-effects have been fully performed. Similarly, an implementation should ensure that all lazily queued side-effects preceding an *accelerator_view::create_marker* call have been fully performed before the *completion_future* object which is retuned by *create_marker* is made ready.

Informative: Future versions of parallel_for_each *may be less constrained in the changes they may affect to shared memory, and at that point an asynchronous implementation will no longer be valid. At that point, an explicitly asynchronous* parallel_for_each_async *will be added to the specification.*

Even though an implementation could be coarse in the way it implements coherence, it still must provide true aliasing for *array_view*s which refer to the same home location. For example, assuming that *a1* and *a2* are both *array_view*s constructed on top of a 100-wide one dimensional *array*, with *a1* referring to elements [0…10] of the *array* and *a2* referring to elements [10...20] of the same *array*. If both *a1* and *a2* are accessible on a *parallel_for_each* call, then accessing *a1* at position 10 is identical to accessing the view *a2* at position 0, since they both refer to the same location of the *array* they are providing a view over, namely position 10 in the original *array*. This rules holds whenever and wherever *a1* and *a2* are accessible simultaneously, i.e., on the host and in *parallel_for_each* calls.

Thus, for example, an implementation could clone an *array_view* passed into a *parallel_for_each* in read-only mode, and pass the cloned data to the device. It can create the clone using any order of reads from the original. The implementation may read the original a multiple number of times, perhaps in order to implement load-balancing or reliability features.

Similarly, an implementation could copy back results from an internally cloned *array*, *array_view* or *texture*, onto the original data. It may overwrite any data in the original container, and it can do so multiple times in the realization of a single *parallel_for_each* call.

When two or more overlapping array views are passed to a *parallel_for_each*, an implementation could create a temporary array corresponding to a section of the original container which contains at a minimum the union of the views necessary for the call. This temporary array will hold the clones of the overlapping *array_view*s while maintaining their aliasing requirements.

The guarantee regarding aliasing of *array_view*s is provided for views which share the same *home location*. The home location of an *array_view* is defined thus:

1. In the case of an *array_view* that is ultimately derived from an array, the home location is the array.
2. In the case of an *array_view* that is ultimately derived from a host pointer, the home location is the original array view created using the pointer.

This means that two different *array_view*s which have both been created, independently, on top of the same memory region are not guaranteed to appear coherent. In fact, creating and using top-level *array_view*s on the same host storage is not supported. In order for such *array_view* to appear coherent, they must have a common top-level *array_view* ancestor which they both ultimately were derived from, and that top-level *array_view* must be the only one which is constructed on top of the memory it refers to.

This is illustrated in the next example:

```
#include <assert.h>
#include <amp.h>

using namespace concurrency;

void coherence_buggy()
{
```

```
  int storage[10];
  array_view<int> av1(10, &storage[0]);
  array_view<int> av2(10, &storage[0]); // error: av2 is top-level and aliases av1
  array_view<int> av3(5,  &storage[5]); // error: av3 is top-level and aliases av1, av2

  parallel_for_each( extent<1>(1), [=] (index<1>) restrict(amp) { av3[2] = 15; });
  parallel_for_each( extent<1>(1), [=] (index<1>) restrict(amp) { av2[7] = 16; });
  parallel_for_each( extent<1>(1), [=] (index<1>) restrict(amp) { av1[7] = 17; });

  assert(av1[7] == av2[7]); // undefined results
  assert(av1[7] == av3[2]); // undefined results
}

void coherence_ok()
{
  int storage[10];
  array_view<int> av1(10, &storage[0]);
  array_view<int> av2(av1);                // OK
  array_view<int> av3(av1.section(5,5));   // OK

  parallel_for_each( extent<1>(1), [=] (index<1>) restrict(amp) { av3[2] = 15; });
  parallel_for_each( extent<1>(1), [=] (index<1>) restrict(amp) { av2[7] = 16; });
  parallel_for_each( extent<1>(1), [=] (index<1>) restrict(amp) { av1[7] = 17; });

  assert(av1[7] == av2[7]); // OK, never fails, both equal 17
  assert(av1[7] == av3[2]); // OK, never fails, both equal 17
}
```

An implementation is not obligated to report such programmer's errors.

8.3 Effects of copy and copy_async operations

Copy operations are offered on *array*, *array_view* and *texture*.

Copy operations copy a source host buffer, *array*, *array_view* or a *texture* to a destination object which can also be one of these four varieties (except host buffer to host buffer, which is handled by *std::copy*). A *copy* operation will read all elements of its source. It may read each element multiple times and it may read elements in any order. It may employ memory load instructions that are either coarser or more granular than the width of the primitive data types in the container, but it is guaranteed to never read a memory location which is strictly outside of the source container.

Similarly, `copy` will overwrite each and every element in its output range. It may do so multiple times and in any order and may coarsen or break apart individual store operations, but it is guaranteed to never write a memory location which is strictly outside of the target container.

A synchronous copy operation extends from the time the function is called until it has returned. During this time, any source location may be read and any destination location may be written. An asynchronous copy extends from the time *copy_async* is called until the time the *std::future* returned is ready.

As always, it is the programmer's responsibility not to call functions which could result in a race. For example, this program is racy because the two copy operations are concurrent and *b* is written to by the first parallel activity while it is being updated by the second parallel activity.

```
      array<int> a(100), b(100), c(100);
      parallel_invoke(
```

```
        [&] { copy(a,b); },
        [&] { copy(b,c); });
```

8.4 Effects of array_view::synchronize, synchronize_async and refresh functions

An *array_view* may be constructed to wrap over a host side pointer. For such *array_view*s, it is generally forbidden to access the underlying *array_view* storage directly, as long as the *array_view* exists. Access to the storage area is generally accomplished indirectly through the *array_view*. However, *array_view* offers mechanisms to synchronize and refresh its contents, which do allow accessing the underlying memory directly. These mechanisms are described below.

Reading of the underlying storage is possible under the condition that the view has been first *synchronized* back to its home storage. This is performed using the *synchronize* or *synchronize_async* member functions of *array_view***.**

When a top-level view is initially created on top of a raw buffer, it is synchronized with it. After it has been constructed, a top-level view, as well as derived views, may lose coherence with the underlying host-side raw memory buffer if the *array_view* is passed to *parallel_for_each* as a mutable view, or if the view is a target of a copy operation. In order to restore coherence with host-side underlying memory *synchronize* or *synchronize_async* must be called. Synchronization is restored when *synchronize* returns, or when the completion_future returned by *synchronize_async* is ready.

For the sake of composition with *parallel_for_each*, *copy*, and all other host-side operations involving a view, *synchronize* should be considered a read of the entire data section referred to by the view, as if it was the source of a copy operation, and thus it must not be executed concurrently with any other operation involving writing the view. Note that even though synchronize does potentially modify the underlying host memory, it is logically a no-op as it doesn't affect the logical contents of the array. As such, it is allowed to execute concurrently with other operations which read the array view. As with *copy*, *synchronize* works at the granularity of the view it is applied to, e.g., synchronizing a view representing a sub-section of a parent view doesn't necessarily synchronize the entire parent view. It is just guaranteed to synchronize the overlapping portions of such related views.

*array_view*s are also required to synchronize their home storage:

1. Before they are destructed if and only if it is the last view of the underlying data container.
2. When they are accessed using the subscript operator or the .data() method (on said home location)

As a result of (1), any errors in synchronization which may be encountered during destruction of arrays views will not be propagated through the destructor. Users are therefore encouraged to ensure that *array_view*s which may contain unsynchronized data are explicitly synchronized before they are destructed.

As a result of (2), the implementation of the subscript operator may need to contain a coherence enforcing check, especially on platforms where the accelerator hardware and host memory are not shared, and therefore coherence is managed explicitly by the C++ AMP runtime. Such a check may be detrimental for code desiring to achieve high performance through vectorization of the array view accesses. Therefore it is recommended for such performance-sensitive code to obtain a pointer to the beginning of a "run" and perform the low-level accesses needed based off of the raw pointer into the *array_view*. *array_view*s are guaranteed to be contiguous in the unit-stride dimension, which enables this style of coding. Furthermore, the code may explicitly synchronize the *array_view* and at that point read the home storage directly, without the mediation of the view.

Sometimes it is desirable to also allow refreshing of a view by directly from its underlying memory. The *refresh* member function is provided for this task. This function revokes any caches associated with the view and resynchronizes the view's contents with the underlying memory. As such it may not be invoked concurrently with any other operation that accesses the view's data. However, it is safe to assume that *refresh* doesn't modify the view's underlying data and therefore concurrent read access to the underlying data is allowed during *refresh*'s operation and after *refresh* has returned, till the point when coherence may have been lost again, as has been described above in the discussion on the *synchronize* member function.

9 Math Functions

C++ AMP contains a rich library of floating point math functions that can be used in an accelerated computation. The C++ AMP library comes in two flavors, each contained in a separate namespace. The functions contained in the *concurrency::fast_math* namespace support only single-precision (*float*) operands and are optimized for performance at the expense of accuracy. The functions contained in the *concurrency::precise_math* namespace support both single and double precision (*double*) operands and are optimized for accuracy at the expense of performance. The two namespaces cannot be used together without introducing ambiguities. The accuracy of the functions in the *concurrency::precise_math* namespace shall be at least as high as those in the *concurrency::fast_math* namespace.

All functions are available in the *<amp_math.h>* header file, and all are decorated *restrict(amp)*.

9.1 fast_math

Functions in the *fast_math* namespace are designed for computations where accuracy is not a prime requirement, and therefore the minimum precision is implementation-defined.

Not all functions available in *precise_math* are available in *fast_math*.

C++ API function	Description
float acosf(float x) float acos(float x)	Returns the arc cosine in radians and the value is mathematically defined to be between 0 and PI (inclusive).
float asinf(float x) float asin(float x)	Returns the arc sine in radians and the value is mathematically defined to be between -PI/2 and PI/2 (inclusive).
float atanf(float x) float atan(float x)	Returns the arc tangent in radians and the value is mathematically defined to be between -PI/2 and PI/2 (inclusive).
float atan2f(float y, float x) float atan2(float y, float x)	Calculates the arc tangent of the two variables x and y. It is similar to calculating the arc tangent of y / x, except that the signs of both arguments are used to determine the quadrant of the result.). Returns the result in radians, which is between -PI and PI (inclusive).
float ceilf(float x) float ceil(float x)	Rounds x up to the nearest integer.
float cosf(float x) float cos(float x)	Returns the cosine of x.
float coshf(float x) float cosh(float x)	Returns the hyperbolic cosine of x.
float expf(float x) float exp(float x)	Returns the value of e (the base of natural logarithms) raised to the power of x.
float exp2f(float x) float exp2(float x)	Returns the value of 2 raised to the power of x.
float fabsf(float x) float fabs(float x)	Returns the absolute value of floating-point number
float floorf(float x) float floor(float x)	Rounds x down to the nearest integer.
float fmaxf(float x, float y) float fmax(float x, float y)	Selects the greater of x and y.

float fminf(float x, float y) float fmin(float x, float y)	Selects the lesser of x and y.
float fmodf(float x, float y) float fmod(float x, float y)	Computes the remainder of dividing x by y. The return value is x - n * y, where n is the quotient of x / y, rounded towards zero to an integer.
float frexpf(float x, int * exp) float frexp(float x, int * exp)	Splits the number x into a normalized fraction and an exponent which is stored in exp.
int isfinite(float x)	Determines if x is finite.
int isinf(float x)	Determines if x is infinite.
int isnan(float x)	Determines if x is NAN.
float ldexpf(float x, int exp) float ldexp(float x, int exp)	Returns the result of multiplying the floating-point number x by 2 raised to the power exp
float logf(float x) float log(float x)	Returns the natural logarithm of x.
float log10f(float x) float log10(float x)	Returns the base 10 logarithm of x.
float log2f(float x) float log2(float x)	Returns the base 2 logarithm of x.
float modff(float x, float * iptr) float modf(float x, float * iptr)	Breaks the argument x into an integral part and a fractional part, each of which has the same sign as x. The integral part is stored in iptr.
float powf(float x, float y) float pow(float x, float y)	Returns the value of x raised to the power of y.
float roundf(float x) float round(float x)	Rounds x to the nearest integer.
float rsqrtf(float x) float rsqrt(float x)	Returns the reciprocal of the square root of x.
int signbit(float x) int signbit(double x)	Returns a non-zero value if the value of X has its sign bit set.
float sinf(float x) float sin(float x)	Returns the sine of x.
void sincosf(float x, float* s, float* c) void sincos(float x, float* s, float* c)	Returns the sine and cosine of x.
float sinhf(float x) float sinh(float x)	Returns the hyperbolic sine of x.
float sqrtf(float x) float sqrt(float x)	Returns the non-negative square root of x
float tanf(float x) float tan(float x)	Returns the tangent of x.
float tanhf(float x) float tanh(float x)	Returns the hyperbolic tangent of x.
float truncf(float x) float trunc(float x)	Rounds x to the nearest integer not larger in absolute value.

The following list of standard math functions from the “std::” namespace shall be imported into the concurrency::fast_math namespace:

```
using std::acosf;
using std::asinf;
using std::atanf;
using std::atan2f;
using std::ceilf;
```

```
    using std::cosf;
    using std::coshf;
    using std::expf;
    using std::fabsf;
    using std::floorf;
    using std::fmodf;
    using std::frexpf;
    using std::ldexpf;
    using std::logf;
    using std::log10f;
    using std::modff;
    using std::powf;
    using std::sinf;
    using std::sinhf;
    using std::sqrtf;
    using std::tanf;
    using std::tanhf;

    using std::acos;
    using std::asin;
    using std::atan;
    using std::atan2;
    using std::ceil;
    using std::cos;
    using std::cosh;
    using std::exp;
    using std::fabs;
    using std::floor;
    using std::fmod;
    using std::frexp;
    using std::ldexp;
    using std::log;
    using std::log10;
    using std::modf;
    using std::pow;
    using std::sin;
    using std::sinh;
    using std::sqrt;
    using std::tan;
    using std::tanh;
```

Importing these names into the fast_math namespace enables each of them to be called in unqualified syntax from a function that has both "restrict(cpu,amp)" restrictions. E.g.,

```
void compute() restrict(cpu,amp) {
  …
  float x = cos(y); // resolves to std::cos in "cpu" context; else fast_math::cos in "amp" context
  …
}
```

9.2 precise_math

Functions in the *precise_math* namespace are designed for computations where accuracy is required. In the table below, the precision of each function is stated in units of "ulps" (error in last position).

Functions in the *precise_math* namespace also support both single and double precision, and are therefore dependent upon double-precision support in the underlying hardware, even for single-precision variants.

C++ API function	Description	Precision (float)	Precision (double)
float acosf(float x) float acos(float x) double acos(double x)	Returns the arc cosine in radians and the value is mathematically defined to be between 0 and PI (inclusive).	3	2
float acoshf(float x) float acosh(float x) double acosh(double x)	Returns the hyperbolic arccosine.	4	2
float asinf(float x) float asin(float x) double asin(double x)	Returns the arc sine in radians and the value is mathematically defined to be between -PI/2 and PI/2 (inclusive).	4	2
float asinhf(float x) float asinh(float x) double asinh(double x)	Returns the hyperbolic arcsine.	3	2
float atanf(float x) float atan(float x) double atan(double x)	Returns the arc tangent in radians and the value is mathematically defined to be between -PI/2 and PI/2 (inclusive).	2	2
float atanhf(float x) float atanh(float x) double atanh(double x)	Returns the hyperbolic arctangent.	3	2
float atan2f(float y, float x) float atan2(float y, float x) double atan2(double y, double x)	Calculates the arc tangent of the two variables x and y. It is similar to calculating the arc tangent of y / x, except that the signs of both arguments are used to determine the quadrant of the result.). Returns the result in radians, which is between -PI and PI (inclusive).	3	2
float cbrtf(float x) float cbrt(float x) double cbrt(double x)	Returns the (real) cube root of x.	1	1
float ceilf(float x) float ceil(float x) double ceil(double x)	Rounds x up to the nearest integer.	0	0
float copysignf(float x, float y) float copysign(float x, float y) double copysign(double x, double y)	Return a value whose absolute value matches that of x, but whose sign matches that of y. If x is a NaN, then a NaN with the sign of y is returned.	N/A	N/A
float cosf(float x) float cos(float x) double cos(double x)	Returns the cosine of x.	2	2
float coshf(float x) float cosh(float x) double cosh(double x)	Returns the hyperbolic cosine of x.	2	2
float cospif(float x) float cospi(float x) double cospi(double x)	Returns the cosine of pi * x.	2	2
float erff(float x)	Returns the error function of x; defined as erf(x) = 2/sqrt(pi)* integral from 0 to x of exp(-t*t) dt	3	2

float erf(float x) double erf(double x)			
float erfcf(float x) float erfc(float x) double erfc(double x)	Returns the complementary error function of x that is 1.0 - erf (x).	6	5
float erfinvf(float x) float erfinv(float x) double erfinv(double x)	Returns the inverse error function.	3	8
float erfcinvf(float x) float erfcinv(float x) double erfcinv(double x)	Returns the inverse of the complementary error function.	7	8
float expf(float x) float exp(float x) double exp(double x)	Returns the value of e (the base of natural logarithms) raised to the power of x.	2	1
float exp2f(float x) float exp2(float x) double exp2(double x)	Returns the value of 2 raised to the power of x.	2	1
float exp10f(float x) float exp10(float x) double exp10(double x)	Returns the value of 10 raised to the power of x.	2	1
float expm1f(float x) float expm1(float x) double expm1(double x)	Returns a value equivalent to 'exp (x) - 1'	1	1
float fabsf(float x) float fabs(float x) double fabs(double x)	Returns the absolute value of floating-point number	N/A	N/A
float fdimf(float x, float y) float fdim(float x, float y) double fdim(double x, double y)	These functions return max(x-y,0). If x or y or both are NaN, Nan is returned.	0	0
float floorf(float x) float floor(float x) double floor(double x)	Rounds x down to the nearest integer.	0	0
float fmaf(float x, float y, float z) float fma(float x, float y, float z) double fma(double x, double y, double z)	Computes (x * y) + z, rounded as one ternary operation: they compute the value (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur.	0	0[6]
float fmaxf(float x, float y) float fmax(float x, float y) double fmax(double x, double y)	Selects the greater of x and y.	N/A	N/A
float fminf(float x, float y) float fmin(float x, float y) double fmin(double x, double y)	Selects the lesser of x and y.	N/A	N/A

[6] IEEE-754 round to nearest even.

float fmodf(float x, float y) float fmod(float x, float y) double fmod(double x, double y)	Computes the remainder of dividing x by y. The return value is x - n * y, where n is the quotient of x / y, rounded towards zero to an integer.	0	0

int fpclassify(float x);
 	
int fpclassify(double x);
	Floating point numbers can have special values, such as infinite or NaN. With the macro fpclassify(x) you can find out what type x is. The function takes any floating-point expression as argument. The result is one of the following values:
	• FP_NAN : x is "Not a Number".
	• FP_INFINITE: x is either plus or minus infinity.
	• FP_ZERO: x is zero.
	• FP_SUBNORMAL : x is too small to be represented in normalized format.
	• FP_NORMAL : if nothing of the above is correct then it must be a normal floating-point number.
		N/A	N/A

float frexpf(float x, int * exp) float frexp(float x, int * exp) double frexp(double x, int * exp)	Splits the number x into a normalized fraction and an exponent which is stored in exp.	0	0
float hypotf(float x, float y) float hypot(float x, float y) double hypot(double x, double y)	Returns sqrt(x*x+y*y). This is the length of the hypotenuse of a right-angle triangle with sides of length x and y, or the distance of the point (x,y) from the origin.	3	2
int ilogbf (float x) int ilogb(float x) int ilogb(double x)	Return the exponent part of their argument as a signed integer. When no error occurs, these functions are equivalent to the corresponding logb() functions, cast to (int). An error will occur for zero and infinity and NaN, and possibly for overflow.	0	0
int isfinite(float x) int isfinite(double x)	Determines if x is finite.	N/A	N/A
int isinf(float x) int isinf(double x)	Determines if x is infinite.	N/A	N/A
int isnan(float x) int isnan(double x)	Determines if x is NAN.	N/A	N/A
int isnormal(float x) int isnormal(double x)	Determines if x is normal.	N/A	N/A
float ldexpf(float x, int exp) float ldexp(float x, int exp) double ldexpf(double x, int exp)	Returns the result of multiplying the floating-point number x by 2 raised to the power exp	0	0
float lgammaf(float x) float lgamma(float x) double lgamma(double x)	Computes the natural logarithm of the absolute value of gamma ofx. A range error occurs if x is too large. A range error may occur if x is a negative integer or zero.	6[7]	4[8]
float logf(float x) float log(float x) double log(double x)	Returns the natural logarithm of x.	1	1
float log10f(float x)	Returns the base 10 logarithm of x.	3	1

[7] Outside interval -10.001 ... -2.264; larger inside.
[8] Outside interval -10.001 ... -2.264; larger inside.

float log10(float x) double log10(double x)			
float log2f(float x) float log2(float x) double log2(double x)	Returns the base 2 logarithm of x.	3	1
float log1pf (float x) float log1p(float x) double log1p(double x)	Returns a value equivalent to 'log (1 + x)'. It is computed in a way that is accurate even if the value of x is near zero.	2	1
float logbf(float x) float logb(float x) double logb(double x)	These functions extract the exponent of x and return it as a floating-point value. If FLT_RADIX is two, logb(x) is equal to floor(log2(x)), except it's probably faster. If x is de-normalized, logb() returns the exponent x would have if it were normalized.	0	0
float modff(float x, float * iptr) float modf(float x, float * iptr) double modf(double x, double * iptr)	Breaks the argument x into an integral part and a fractional part, each of which has the same sign as x. The integral part is stored in iptr.	0	0
float nanf(int tagp) float nanf(int tagp) double nan(int tagp)	return a representation (determined by tagp) of a quiet NaN. If the implementation does not support quiet NaNs, these functions return zero.	N/A	N/A
float nearbyintf(float x) float nearbyint(float x) double nearbyint(double x)	Rounds the argument to an integer value in floating point format, using the current rounding direction	0	
float nextafterf(float x, float y) float nextafter(float x, float y) double nextafter(double x, double y)	Returns the next representable neighbor of x in the direction towards y. The size of the step between x and the result depends on the type of the result. If x = y the function simply returns y. If either value is NaN, then NaN is returned. Otherwise a value corresponding to the value of the least significant bit in the mantissa is added or subtracted, depending on the direction.	N/A	N/A
float powf(float x, float y) float pow(float x, float y) double pow(double x, double y)	Returns the value of x raised to the power of y.	8	2
float rcbrtf(float x) float rcbrt(float x) double rcbrt(double x)	Calculates reciprocal of the (real) cube root of x	2	1
float remainderf(float x, float y) float remainder(float x, float y) double remainder(double x, double y)	Computes the remainder of dividing x by y. The return value is x - n * y, where n is the value x / y, rounded to the nearest integer. If this quotient is 1/2 (mod 1), it is rounded to the nearest even number (independent of the current rounding mode). If the return value is 0, it has the sign of x.	0	0
float remquof(float x, float y, int * quo) float remquo(float x, float y, int * quo) double remquo(double x, double y, int * quo)	Computes the remainder and part of the quotient upon division of x by y. A few bits of the quotient are stored via the quo pointer. The remainder is returned.	0	0
float roundf(float x) float round(float x) double round(double x)	Rounds x to the nearest integer.	0	0
float rsqrtf(float x)	Returns the reciprocal of the square root of x.	2	1

float rsqrt(float x) double rsqrt(double x)			
float sinpif(float x) float sinpi(float x) double sinpi(double x)	Returns the sine of pi * x.	2	2
float scalbf(float x, float exp) float scalb(float x, float exp) double scalb(double x, double exp)	Multiplies their first argument x by FLT_RADIX (probably 2) to the power exp.	0	0
float scalbnf(float x, int exp) float scalbn(float x, int exp) double scalbn(double x, int exp)	Multiplies their first argument x by FLT_RADIX (probably 2) to the power exp. If FLT_RADIX equals 2, then scalbn() is equivalent to ldexp(). The value of FLT_RADIX is found in <float.h>.	0	0
int signbit(float x) int signbit(double x)	Returns a non-zero value if the value of X has its sign bit set.	N/A	N/A
float sinf(float x) float sin(float x) double sin(double x)	Returns the sine of x.	2	2
void sincosf(float x, float * s, float * c) void sincos(float x, float * s, float * c) void sincos(double x, double * s, double * c)	Returns the sine and cosine of x.	2	2
float sinhf(float x) float sinh(float x) double sinh(double x)	Returns the hyperbolic sine of x.	3	2
float sqrtf(float x) float sqrt(float x) double sqrt(double x)	Returns the non-negative square root of x	0	0[9]
float tgammaf(float x) float tgamma(float x) double tgamma(double x)	This function returns the value of the Gamma function for the argument x.	11	8
float tanf(float x) float tan(float x) double tan(double x)	Returns the tangent of x.	4	2
float tanhf(float x) float tanh(float x) double tanh(double x)	Returns the hyperbolic tangent of x.	2	2
float tanpif(float x) float tanpi(float x) double tanpi(double x)	Returns the tangent of pi * x.	2	2
float truncf(float x) float trunc(float x) double trunc(double x)	Rounds x to the nearest integer not larger in absolute value.	0	0

The following list of standard math functions from the “std::” namespace shall be imported into the concurrency::precise _math namespace:

[9] IEEE-754 round to nearest even.

```
using std::acosf;
using std::asinf;
using std::atanf;
using std::atan2f;
using std::ceilf;
using std::cosf;
using std::coshf;
using std::expf;
using std::fabsf;
using std::floorf;
using std::fmodf;
using std::frexpf;
using std::ldexpf;
using std::logf;
using std::log10f;
using std::modff;
using std::powf;
using std::sinf;
using std::sinhf;
using std::sqrtf;
using std::tanf;
using std::tanhf;

using std::acos;
using std::asin;
using std::atan;
using std::atan2;
using std::ceil;
using std::cos;
using std::cosh;
using std::exp;
using std::fabs;
using std::floor;
using std::fmod;
using std::frexp;
using std::ldexp;
using std::log;
using std::log10;
using std::modf;
using std::pow;
using std::sin;
using std::sinh;
using std::sqrt;
using std::tan;
using std::tanh;
```

Importing these names into the precise_math namespace enables each of them to be called in unqualified syntax from a function that has both “restrict(cpu,amp)” restrictions. E.g.,

```
void compute() restrict(cpu,amp) {
  …
  float x = cos(y); // resolves to std::cos in “cpu” context; else fast_math::cos in “amp” context
  …
}
```

9.3 Miscellaneous Math Functions (Optional)

The following functions allow access to Direct3D intrinsic functions. These are included in *<amp.h>* in the *concurrency::direct3d* namespace, and are only callable from a *restrict(amp)* function.

`int abs(int val) restrict(amp)`	
Returns the absolute value of the integer argument.	
Parameters:	
val	The input value.
Returns the absolute value of the input argument.	

`int clamp(int x, int min, int max) restrict(amp)` `float clamp(float x, float min, float max) restrict(amp)`	
Clamps the input argument "x" so it is always within the range [min,max]. If x < min, then this function returns the value of min. If x > max, then this function returns the value of max. Otherwise, x is returned.	
Parameters:	
val	The input value.
min	The minimum value of the range
max	The maximum value of the range
Returns the clamped value of "x".	

`unsigned int countbits(unsigned int val) restrict(amp)`	
Counts the number of bits in the input argument that are set (1).	
Parameters:	
val	The input value.
Returns the number of bits that are set.	

`int firstbithigh(int val) restrict(amp)`	
Returns the bit position of the first set (1) bit in the input "val", starting from highest-order and working down.	
Parameters:	
val	The input value.
Returns the position of the highest-order set bit in "val".	

`int firstbitlow(int val) restrict(amp)`	
Returns the bit position of the first set (1) bit in the input "val", starting from lowest-order and working up.	
Parameters:	
val	The input value.
Returns the position of the lowest-order set bit in "val".	

`int imax(int x, int y) restrict(amp)`
Returns the maximum of "x" and "y".
Parameters:

x	The first input value.
y	The second input value
Returns the maximum of the inputs.	

`int imin(int x, int y) restrict(amp)`	
Returns the minimum of "x" and "y".	
Parameters:	
x	The first input value.
y	The second input value
Returns the minimum of the inputs.	

`float mad(float x, float y, float z) restrict(amp)`
`double mad(double x, double y, double z) restrict(amp)`
`int mad(int x, int y, int z) restrict(amp)`
`unsigned int mad(unsigned int x, unsigned int y, unsigned int z) restrict(amp)`	

Performs a multiply-add on the three arguments: x*y + z.	
Parameters:	
x	The first input multiplicand.
y	The second input multiplicand
z	The third input addend
Returns x*y + z.	

`float noise(float x) restrict(amp)`	
Generates a random value using the Perlin noise algorithm. The returned value will be within the range [-1,+1].	
Parameters:	
x	The first input value.
Returns the random noise value.	

`float radians(float x) restrict(amp)`	
Converts from "x" degrees into radians.	
Parameters:	
x	The first input in degrees.
Returns the radian value.	

`float rcp(float x) restrict(amp)`	
Calculates a fast approximate reciprocal of "x".	
Parameters:	
x	The input value.
Returns the reciprocal of the input.	

`unsigned int reversebits(unsigned int val) restrict(amp)`	
Reverses the order of the bits in the input argument.	
Parameters:	
val	The input value.
Returns the bit-reversed number.	

`float saturate(float x) restrict(amp)`	
Clamps the input value into the range [-1,+1].	
Parameters:	
x	The input value.
Returns the clamped value.	

`int sign(int x) restrict(amp)`	
Returns the sign of "x"; that is, it returns -1 if x is negative, 0 if x is 0, or +1 if x is positive.	
Parameters:	
x	The first input value.
y	The second input value
Returns the sign of the input.	

`float smoothstep(float min, float max, float x) restrict(amp)`	
Returns a smooth Hermite interpolation between 0 and 1, if x is in the range [min, max].	
Parameters:	
min	The minimum value of the range.
max	The maximum value of the range.
x	The value to be interpolated.
Returns the interpolated value.	

`float step(float x, float y) restrict(amp)`	
Compares two values, returning 0 or 1 based on which value is greater.	
Parameters:	
x	The first input value.
y	The second input value.
Returns 1 if the x parameter is greater than or equal to the y parameter; otherwise, 0.	

10 Graphics (Optional)

Programming model elements defined in *<amp_graphics.h>* and *<amp_short_vectors.h>* are designed for graphics programming in conjunction with accelerated compute on an accelerator device, and are therefore appropriate only for proper GPU accelerators. Accelerator devices that do not support native graphics functionality need not implement these features.

All types in this section are defined in the *concurrency::graphics* namespace.

10.1 texture<T,N>

The *texture* class provides the means to create textures from raw memory or from file. *texture*s are similar to *array*s in that they are containers of data and they behave like STL containers with respect to assignment and copy construction.

*texture*s are templated on *T*, the element type, and on *N*, the rank of the texture. *N* can be one of 1, 2 or 3.

The element type of the *texture*, also referred to as the texture's logical element type, is one of a closed set of short vector types defined in the *concurrency::graphics* namespace and covered elsewhere in this specification. The below table briefly enumerates all supported element types.

Rank of element type, (also referred to as "number of scalar elements")	Signed Integer	Unsigned Integer	Single precision floating point number	Single precision singed normalized number	Single precision unsigned normalized number	Double precision floating point number
1	int	unsigned int	float	norm	unorm	double
2	int_2	uint_2	float_2	norm_2	unorm_2	double_2
3	int_3	uint_3	float_3	norm_3	unorm_3	double_3
4	int_4	uint_4	float_4	norm_4	unorm_4	double_4

Remarks:

1. *norm* and *unorm* vector types are vector of *float*s which are normalized to the range [-1..1] and [0...1], respectively.
2. Grayed-out cells represent vector types which are defined by C++ AMP but which are not necessarily supported as *texture* value types. Implementations can optionally support the types in the grayed-out cells in the above table.

Microsoft-specific: grayed-out cells in the above table are not supported.

10.1.1 Synopsis

```
template <typename T, int N>
class texture
{
public:
    static const int rank = _Rank;
    typedef typename T value_type;
    typedef short_vectors_traits<T>::scalar_type scalar_type;


    texture(const extent<N>& _Ext);

    texture(int _E0);
    texture(int _E0, int _E1);
    texture(int _E0, int _E1, int _E2);

    texture(const extent<N>& _Ext, const accelerator_view& _Acc_view);

    texture(int _E0, const accelerator_view& _Acc_view);
    texture(int _E0, int _E1, const accelerator_view& _Acc_view);
    texture(int _E0, int _E1, int _E2, const accelerator_view& _Acc_view);

    texture(const extent<N>& _Ext, unsigned int _Bits_per_scalar_element);
```

```
    texture(int _E0, unsigned int _Bits_per_scalar_element);
    texture(int _E0, int _E1, unsigned int _Bits_per_scalar_element);
    texture(int _E0, int _E1, int _E2, unsigned int _Bits_per_scalar_element);

    texture(const extent<N>& _Ext, unsigned int _Bits_per_scalar_element,
            const accelerator_view& _Acc_view);

    texture(int _E0, unsigned int _Bits_per_scalar_element, const accelerator_view&
_Acc_view);
    texture(int _E0, int _E1, unsigned int _Bits_per_scalar_element,
            const accelerator_view& _Acc_view);
    texture(int _E0, int _E1, int _E2, unsigned int _Bits_per_scalar_element,
            const accelerator_view& _Acc_view);

    template <typename TInputIterator>
      texture(const extent<N>&, TInputIterator _Src_first, TInputIterator _Src_last);

    template <typename TInputIterator>
      texture(int _E0, TInputIterator _Src_first, TInputIterator _Src_last);
    template <typename TInputIterator>
      texture(int _E0, int _E1, TInputIterator _Src_first, TInputIterator _Src_last);
    template <typename TInputIterator>
      texture(int _E0, int _E1, int _E2, TInputIterator _Src_first,
              TInputIterator _Src_last);

    template <typename TInputIterator>
      texture(const extent<N>&, TInputIterator _Src_first, TInputIterator _Src_last,
              const accelerator_view& _Acc_view);

    template <typename TInputIterator>
      texture(int _E0, TInputIterator _Src_first, TInputIterator _Src_last,
              const accelerator_view& _Acc_view);
    template <typename TInputIterator>
      texture(int _E0, int _E1, TInputIterator _Src_first, TInputIterator _Src_last,
              const accelerator_view& _Acc_view);
    texture(int _E0, int _E1, int _E2, TInputIterator _Src_first, TInputIterator _Src_last,
const accelerator_view& _Acc_view);

    texture(const extent<N>&, const void * _Source, unsigned int _Src_byte_size,
            unsigned int _Bits_per_scalar_element);

    texture(int _E0, const void * _Source, unsigned int _Src_byte_size,
            unsigned int _Bits_per_scalar_element);
    texture(int _E0, int _E1, const void * _Source, unsigned int _Src_byte_size,
            unsigned int _Bits_per_scalar_element);
    texture(int _E0, int _E1, int _E2, const void * _Source,
            unsigned int _Src_byte_size, unsigned int _Bits_per_scalar_element);

    texture(const extent<N>&, const void * _Source, unsigned int _Src_byte_size,
            unsigned int _Bits_per_scalar_element, const accelerator_view& _Acc_view);

    texture(int _E0, const void * _Source, unsigned int _Src_byte_size,
            unsigned int _Bits_per_scalar_element, const accelerator_view& _Acc_view);
    texture(int _E0, int _E1, const void * _Source, unsigned int _Src_byte_size,
            unsigned int _Bits_per_scalar_element, const accelerator_view& _Acc_view);
    texture(int _E0, int _E1, int _E2, const void * _Source, unsigned int _Src_byte_size,
            unsigned int _Bits_per_scalar_element, const accelerator_view& _Acc_view);


    texture(const texture& _Src);
    texture(const texture& _Src, const accelerator_view& _Acc_view);
    texture& operator=(const texture& _Src);
```

```
    texture(texture&& _Other);
    texture& operator=(texture&& _Other);

    void copy_to(texture& _Dest) const;
    void copy_to(const writeonly_texture_view<T,N>& _Dest) const;

    unsigned int get_bits_per_scalar_element() const;
    __declspec(property(get= get_bits_per_scalar_element)) int bits_per_scalar_element;

    unsigned int get_data_length() const;
    __declspec(property(get=get_data_length)) unsigned int data_length;

    extent<N> get_extent() const restrict(cpu,amp);
    __declspec(property(get=get_extent)) extent<N> extent;

    accelerator_view get_accelerator_view() const;
    __declspec(property(get=get_accelerator_view)) accelerator_view accelerator_view;

    const value_type operator[] (const index<N>& _Index) const restrict(amp);
    const value_type operator[] (int _I0) const restrict(amp);
    const value_type operator() (const index<N>& _Index) const restrict(amp);
    const value_type operator() (int _I0) const restrict(amp);
    const value_type operator() (int _I0, int _I1) const restrict(amp);
    const value_type operator() (int _I0, int _I1, int _I2) const restrict(amp);
    const value_type get(const index<N>& _Index) const restrict(amp);

    void set(const index<N>& _Index, const value_type& _Val) restrict(amp);
};
```

10.1.2 Introduced typedefs

```
typedef ... value_type;
```

The logical value type of the texture. e.g., for texture <float2, 3>, value_type would be float2.

```
typedef ... scalar_type;
```

The scalar type that serves as the component of the texture's value type. For example, for texture<int2, 3>, the scalar type would be "int".

10.1.3 Constructing an uninitialized texture

```
texture(const extent<N>& _Ext);

texture(int _E0);
texture(int _E0, int _E1);
texture(int _E0, int _E1, int _E2);

texture(const extent<N>& _Ext, const accelerator_view& _Acc_view);

texture(int _E0, const accelerator_view& _Acc_view);
texture(int _E0, int _E1, const accelerator_view& _Acc_view);
texture(int _E0, int _E1, int _E2, const accelerator_view& _Acc_view);

texture(const extent<N>& _Ext, unsigned int _Bits_per_scalar_element);

texture(int _E0, unsigned int _Bits_per_scalar_element);
texture(int _E0, int _E1, unsigned int _Bits_per_scalar_element);
texture(int _E0, int _E1, int _E2, unsigned int _Bits_per_scalar_element);
```

```
texture(const extent<N>& _Ext, unsigned int _Bits_per_scalar_element, const accelerator_view&
_Acc_view);

texture(int _E0, unsigned int _Bits_per_scalar_element, const accelerator_view& _Acc_view);
texture(int _E0, int _E1, unsigned int _Bits_per_scalar_element, const accelerator_view&
_Acc_view);
texture(int _E0, int _E1, int _E2, unsigned int _Bits_per_scalar_element, const
accelerator_view& _Acc_view);
```

Creates an uninitialized texture with the specified shape, number of bits per scalar element, on the specified accelerator view.

Parameters:

_Ext	Extents of the texture to create
_E0	Extent of dimension 0
_E1	Extent of dimension 1
_E2	Extent of dimension 2
_Bits_per_scalar_element	Number of bits per each scalar element in the underlying scalar type of the texture.
_Acc_view	Accelerator view where to create the texture
Error condition	**Exception thrown**
Out of memory	concurrency::runtime_exception
Invalid number of bits per scalar elementspecified	concurrency::runtime_exception
Invalid combination of value_type and bits per scalar element	`concurrency::unsupported_feature`
accelerator_view doesn't support textures	`concurrency::unsupported_feature`

The table below summarizes all valid combinations of underlying scalar types (columns), ranks(rows), supported values for bits-per-scalar-element (inside the table cells), and default value of bits-per-scalar-element for each given combination (highlighted in green). Note that unorm and norm have no default value for bits-per-scalar-element. Implementations can optionally support textures of double4, with implementation-specific values of bits-per-scalar-element.

Microsoft-specific: the current implementation doesn't support textures of double4.

Rank	int	uint	float	norm	unorm	double
1	8, 16, 32	8, 16, 32	16, 32	8, 16	8, 16	64
2	8, 16, 32	8, 16, 32	16, 32	8, 16	8, 16	64
4	8, 16, 32	8, 16, 32	16, 32	8, 16	8, 16	

10.1.4 Constructing a texture from a host side iterator

```
template <typename TInputIterator>
texture(const extent<N>& _Ext, TInputIterator _Src_first, TInputIterator _Src_last);
texture(int _E0, TInputIterator _Src_first, TInputIterator _Src_last);
texture(int _E0, int _E1, TInputIterator _Src_first, TInputIterator _Src_last);
texture(int _E0, int _E1, int _E2, TInputIterator _Src_first, TInputIterator _Src_last);

template <typename TInputIterator>
```

```
texture(const extent<N>&, TInputIterator _Src_first, TInputIterator _Src_last, const accelerator_view& _Acc_view);

template <typename TInputIterator>
texture(const extent<N>& _Ext, TInputIterator _Src_first, TInputIterator _Src_last, const accelerator_view& _Acc_view);
texture(int _E0, TInputIterator _Src_first, TInputIterator _Src_last, const accelerator_view& _Acc_view);
texture(int _E0, int _E1, TInputIterator _Src_first, TInputIterator _Src_last, const accelerator_view& _Acc_view);
texture(int _E0, int _E1, int _E2, TInputIterator _Src_first, TInputIterator _Src_last, const accelerator_view& _Acc_view);
```

Creates a texture from a host-side iterator. The data type of the iterator must be the same as the value type of the texture. Textures with element types based on norm or unorm do not support this constructor (usage of it will result in a compile-time error).

Parameters:	
_Ext	Extents of the texture to create
_E0	Extent of dimension 0
_E1	Extent of dimension 1
_E2	Extent of dimension 2
_Src_first	Iterator pointing to the first element to be copied into the texture
_Src_last	Iterator pointing immediately past the last element to be copied into the texture
_Acc_view	Accelerator view where to create the texture
Error condition	**Exception thrown**
Out of memory	concurrency::runtime_exception
Inadequate amount of data supplied through the iterators	concurrency::runtime_exception
Accelerator_view doesn't support textures	`concurrency::unsupported_feature`

10.1.5 Constructing a texture from a host-side data source

```
texture(const extent<N>&, const void * _Source, unsigned int _Src_byte_size, unsigned int _Bits_per_scalar_element);

texture(int _E0, const void * _Source, unsigned int _Src_byte_size, unsigned int _Bits_per_scalar_element);
texture(int _E0, int _E1, const void * _Source, unsigned int _Src_byte_size, unsigned int _Bits_per_scalar_element);
texture(int _E0, int _E1, int _E2, const void * _Source, unsigned int _Src_byte_size, unsigned int _Bits_per_scalar_element);

texture(const extent<N>&, const void * _Source, unsigned int _Src_byte_size, unsigned int _Bits_per_scalar_element, const accelerator_view& _Acc_view);

texture(int _E0, const void * _Source, unsigned int _Src_byte_size, unsigned int _Bits_per_scalar_element, const accelerator_view& _Acc_view);
texture(int _E0, int _E1, const void * _Source, unsigned int _Src_byte_size, unsigned int _Bits_per_scalar_element, const accelerator_view& _Acc_view);
texture(int _E0, int _E1, int _E2, const void * _Source, unsigned int _Src_byte_size, unsigned int _Bits_per_scalar_element, const accelerator_view& _Acc_view);
```

Creates a texture from a host-side provided buffer. The format of the data source must be compatible with the texture's vector type, and the amount of data in the data source must be exactly the amount necessary to initialize a texture in the specified format, with the given number of bits per scalar element. For example, a 2D texture of uint2 initialized with the extent of 100x200 and with _Bits_per_scalar_element equal to 8 will require a total of 100 * 200 * 2 * 8 = 320,000 bits available to copy from _Source, which is equal to 40,000 bytes. (or in other words, one byte, per one scalar element, for each scalar element, and each pixel, in the texture).	
Parameters:	
_Ext	Extents of the texture to create
_E0	Extent of dimension 0
_E1	Extent of dimension 1
_E2	Extent of dimension 2
_Source	Pointer to a host buffer
_Src_byte_size	Number of bytes of the host source buffer
_Bits_per_scalar_element	Number of bits per each scalar element in the underlying scalar type of the texture.
_Acc_view	Accelerator view where to create the texture
Error condition	**Exception thrown**
Out of memory	concurrency::runtime_exception
Inadequate amount of data supplied through the host buffer (_Src_byte_size < texture.data_length)	concurrency::runtime_exception
Invalid number of bits per scalar elementspecified	concurrency::runtime_exception
Invalid combination of value_type and bits per scalar element	`concurrency::unsupported_feature`
Accelerator_view doesn't support textures	`concurrency::unsupported_feature`

10.1.6 Constructing a texture by cloning another

```
texture(const texture& _Src);
```

Initializes one texture from another. The texture is created on the same accelerator view as the source.	
Parameters:	
_Src	Source texture or texture_view to copy from
Error condition	**Exception thrown**
Out of memory	concurrency::runtime_exception

```
texture(const texture& _Src, const accelerator_view& _Acc_view);
```

Initializes one texture from another.	
Parameters:	
_Src	Source texture or texture_view to copy from
_Acc_view	Accelerator view where to create the texture
Error condition	**Exception thrown**
Out of memory	concurrency::runtime_exception

Accelerator_view doesn't support textures	concurrency::unsupported_feature

10.1.7 Assignment operator

```
texture& operator=(const texture& _Src);
```

Release the resource of this texture, allocate the resource according to _Src's properties, then deep copy _Src's content to this texture.	
Parameters:	
_Src	Source texture or texture_view to copy from
Error condition	**Exception thrown**
Out of memory	concurrency::runtime_exception

10.1.8 Copying textures

```
void copy_to(texture& _Dest) const;
void copy_to(const writeonly_texture_view<T,N>& _Dest) const;
```

Copies the contents of one texture onto the other. The textures must have been created with exactly the same extent and with compatible physical formats; that is, the number of scalar elements and the number of bits per scalar elements must agree. The textures could be from different accelerators.	
Parameters:	
_Dest	Destination texture or writeonly_texture_view to copy to
Error condition	**Exception thrown**
Out of memory	concurrency::runtime_exception
Incompatible texture formats	concurrency::runtime_exception
Extents don't match	concurrency::runtime_exception

10.1.9 Moving textures

```
texture(texture&& _Other);
texture& operator=(texture&& _Other);
```

"Moves" (in the C++ R-value reference sense) the contents of _Other to "this". The source and destination textures do not have to be necessarily on the same accelerator originally. As is typical in C++ move constructors, no actual copying or data movement occurs; simply one C++ texture object is vacated of its internal representation, which is moved to the target C++ texture object.	
Parameters:	
_Other	Object whose contents are moved to "this"
Error condition	**Exception thrown**
None	

10.1.10 Querying texture's physical characteristics

```
unsigned int get_Bits_per_scalar_element() const;
__declspec(property(get=get_Bits_per_scalar_element)) unsigned int bits_per_scalar_element;
```

Gets the bits-per-scalar-element of the texture. Returns 0, if the texture is created using Direct3D Interop (10.1.15).
Error conditions: none

```
unsigned int get_data_length() const;
__declspec(property(get=get_data_length)) unsigned int data_length;
```

Gets the physical data length (in bytes) that is required in order to represent the texture on the host side with its native format.

Error conditions: none

10.1.11 Querying texture's logical dimensions

```
extent<N> get_extent() const restrict(cpu,amp);
__declspec(property(get=get_extent)) extent<N> extent;
```

These members have the same meaning as the equivalent ones on the array class

Error conditions: none

10.1.12 Querying the accelerator_view where the texture resides

```
accelerator_view get_accelerator_view() const;
__declspec(property(get=get_accelerator_view)) accelerator_view accelerator_view;
```

Retrieves the accelerator_view where the texture resides

Error conditions: none

10.1.13 Reading and writing textures

This is the core function of class texture on the accelerator. Unlike *array*s, the entire value type has to be get/set, and is returned or accepted wholly. *texture*s do not support returning a reference to their data internal representation.

Due to platform restrictions, only a limited number of *texture* types support simultaneous reading and writing. Reading is supported on all *texture* types, but writing through a *texture&* is only supported for *texture*s of *int*, *uint*, and *float*, and even in those cases, the number of bits used in the physical format must be 32. In case a lower number of bits is used (8 or 16) and a kernel is invoked which contains code that could possibly both write into and read from one of these rank-1 *texture* types, then an implementation is permitted to raise a runtime exception.

Microsoft-specific: the Microsoft implementation always raises a runtime exception in such a situation.

Trying to call "set" on a *texture&* of a different element type (i.e., on other than *int*, *uint*, and *float*) results in a static assert. In order to write into *texture*s of other value types, the developer must go through a *writeonly_texture_view<T,N>*.

```
const value_type operator[] (const index<N>& _Index) const restrict(amp);
const value_type operator[] (int _I0) const restrict(amp);
const value_type operator() (const index<N>& _Index) const restrict(amp);
const value_type operator() (int _I0) const restrict(amp);
const value_type operator() (int _I0, int _I1) const restrict(amp);
const value_type operator() (int _I0, int _I1, int _I2) const restrict(amp);
const value_type get(const index<N>& _Index) const restrict(amp);
void set(const index<N>& _Index, const value_type& _Value) const restrict(amp);
```

Loads one texel out of the texture. In case the overload where an integer tuple is used, if an overload which doesn't agree with the rank of the matrix is used, then a static_assert ensues and the program fails to compile.

In the texture is indexed, at runtime, outside of its logical bounds, behavior is undefined.

Parameters

_Index	An N-dimension logical integer coordinate to read from
_I0, _I1, _I0	Index components, equivalent to providing index<1>(_I0), or index<2>(_I0,_I1) or index<2>(_I0,_I1,_I2). The arity of the function used must agree with the rank of the matrix. e.g., the overload which takes (_I0,_I1) is only available on textures of rank 2.
_Value	Value to write into the texture
Error conditions: if set is called on texture types which are not supported, a static_assert ensues.	

10.1.14 Global texture copy functions

```
template <typename T, int N>
void copy(const texture<T,N>& _Texture, void * _Dst, unsigned int _Dst_byte_size);
```

Copies raw texture data to a host-side buffer. The buffer must be laid out in accordance with the texture format and dimensions.	
Parameters	
_Texture	Source texture or texture_view
_Dst	Pointer to destination buffer on the host
_Dst_byte_size	Number of bytes in the destination buffer
Error condition	**Exception thrown**
Out of memory (*)	
Buffer too small	

(*) Out of memory errors may occur due to the need to allocate temporary buffers in some memory transfer scenarios.

```
template <typename T, int N>
void  copy(const void * _Src, unsigned int _Src_byte_size, texture<T,N>& _Texture);
```

Copies raw texture data to a device-side texture. The buffer must be laid out in accordance with the texture format and dimensions.	
Parameters	
_Texture	Destination texture
_Src	Pointer to source buffer on the host
_Src_byte_size	Number of bytes in the destination buffer
Error condition	**Exception thrown**
Out of memory	
Buffer too small	

10.1.14.1 Global async texture copy functions

For each *copy* function specified above, a *copy_async* function will also be provided, returning a `completion_future`.

10.1.15 Direct3d Interop Functions

The following functions are provided in the direct3d namespace in order to convert between DX COM interfaces and textures.

```
template <typename T, int N>
texture<T,N> make_texture(const Concurrency::accelerator_view &_Av, const IUnknown* pTexture);
```

Creates a texture from the corresponding DX interface. On success, it increments the reference count of the D3D texture interface by calling “AddRef” on the interface. Users must call “Release” on the returned interface after they are finished using it, for proper reclamation of the resources associated with the object.	
Parameters	
Av	A D3D accelerator view on which the texture is to be created.
pTexture	A pointer to a suitable texture

Return value	Created texture
Error condition	**Exception thrown**
Out of memory	
Invalid D3D texture argument	

```
template <typename T, int N>
IUnknown * get_texture<const texture<T, N>& _Texture);
```

Retrieves a DX interface pointer from a C++ AMP texture object. Class texture allows retrieving a texture interface pointer (the exact interface depends on the rank of the class). On success, it increments the reference count of the D3D texture interface by calling "AddRef" on the interface. Users must call "Release" on the returned interface after they are finished using it, for proper reclamation of the resources associated with the object.

Parameters	
_Texture	Source texture
Return value	Texture interface as IUnknown *
Error condition: no	

10.2 writeonly_texture_view<T,N>

C++ AMP write-only texture views, coded as *writeonly_texture_view<T, N>*, which provides write-only access into any *texture*.

10.2.1 Synopsis

```
template <typename T, int N>
class writeonly_texture_view<T,N>
{
public:
    static const int rank = _Rank;
    typedef typename T value_type;
    typedef short_vectors_traits<T>::scalar_type scalar_type;

    writeonly_texture_view(texture<T,N>& _Src) restrict(cpu,amp);

    writeonly_texture_view(const writeonly_texture_view&) restrict(cpu,amp);

    writeonly_texture_view operator=(const writeonly_texture_view&) restrict(cpu,amp);

    ~writeonly_texture_view() restrict(cpu,amp);

    unsigned int get_Bits_per_scalar_element()const;
    __declspec(property(get= get_Bits_per_scalar_element)) int bits_per_scalar_element;

    unsigned int get_data_length() const;
    __declspec(property(get=get_data_length)) unsigned int data_length;

    extent<N> get_extent() const restrict(cpu,amp);
    __declspec(property(get=get_extent)) extent<N> extent;

    accelerator_view get_accelerator_view() const;
    __declspec(property(get=get_accelerator_view)) accelerator_view accelerator_view;

    void set(const index<N>& _Index, const value_type& _Val) const restrict(amp);
};
```

10.2.2 Introduced typedefs

```
typedef ... value_type;
```

The logical value type of the writeonly_texture_view. e.g., for writeonly_texture_view<float2,3>, value_type would be float2.

```
typdef ... scalar_type;
```

The scalar type that serves as the component of the texture's value type. For example, for writeonly _texture_view<int2,3>, the scalar type would be "int".

10.2.3 Construct a writeonly view over a texture

```
writeonly_texture_view(texture<T,N>& _Src) restrict(cpu);
writeonly_texture_view(texture<T,N>& _Src) restrict(amp);
```

Creates a write-only view to a given texture.

When create the writeonly_texture_view in a direct3d function, if the number of scalar elements of T is larger than 1, a compilation error will be given.

Parameters	
_Src	Source texture

10.2.4 Copy constructors and assignment operators

```
writeonly_texture_view(const writeonly_texture_view& _Other) restrict(cpu,amp);
writeonly_texture_view operator=(const writeonly_texture_view& _Other) restrict(cpu,amp);
```

writeonly_texture_views are shallow objects which can be copied and moved both on the CPU and on an accelerator. They are captured by value when passed to parallel_for_each

Parameters	
_Other	Source writeonly_texture view to copy
Error condition	**Exception thrown**

10.2.5 Destructor

```
~writeonly_texture_view() restrict(cpu,amp);
```

texture_view can be destructed on the accelerator.

Error conditions: none

10.2.6 Querying underlying texture's physical characteristics

```
unsigned int get_Bits_per_scalar_element() const;
__declspec(property(get=get_Bits_per_scalar_element)) unsigned int bits_per_scalar_element;
```

Gets the bits-per-scalar-element of the texture

Error conditions: none

```
unsigned int get_data_length() const;
__declspec(property(get=get_data_length)) unsigned int data_length;
```

Gets the physical data length (in bytes) that is required in order to represent the texture on the host side with its native format.

Error conditions: none

10.2.7 Querying the underlying texture's accelerator_view

```
accelerator_view get_accelerator_view() const;
__declspec(property(get=get_accelerator_view)) accelerator_view accelerator_view;
```

Retrieves the accelerator_view where the underlying texture resides.
Error conditions: none

10.2.7.1 Querying underlying texture's logical dimensions (through a view)

```
extent<N> get_extent() const restrict(cpu,amp);
__declspec(property(get=get_extent)) extent<N> extent;
```

These members have the same meaning as the equivalent ones on the array class
Error conditions: none

10.2.7.2 Writing a write-only texture view

This is the main purpose of this type. All *texture* types can be written through a write-only view.

```
void set(const index<N>& _Index, const value_type& _Val) const restrict(amp);
```

Stores one texel in the texture. If the texture is indexed, at runtime, outside of its logical bounds, behavior is undefined.	
Parameters	
_Index	An N-dimension logical integer coordinate to read from
_I0, _I1, _I0	Index components
_Val	Value to store into the texture
Error conditions: none	

10.2.8 Global writeonly_texture_view copy functions

```
template <typename T, int N>
void  copy(const void * _Src, unsigned int _Src_byte_size, const writeonly_texture_view<T,N>&
_TextureView);
```

Copies raw texture data to a device-side writeonly texture view. The buffer must be laid out in accordance with the texture format and dimensions.	
Parameters	
_TextureView	Destination texture view
_Src	Pointer to source buffer on the host
_Src_byte_size	Number of bytes in the destination buffer
Error condition	**Exception thrown**
Out of memory	
Buffer too small	

10.2.8.1 Global async writeonly_texture_view copy functions

For each *copy* function specified above, a *copy_async* function will also be provided, returning a `completion_future`.

10.2.9 Direct3d Interop Functions

The following functions are provided in the *direct3d* namespace in order to convert between DX COM interfaces and *writeonly_texture_view*s.

```
template <typename T, int N>
IUnknown * get_texture<const writeonly_texture_view<T, N>& _TextureView);
```

Retrieves a DX interface pointer from a C++ AMP writeonly_texture_view object. On success, it increments the reference count of the D3D texture interface by calling “AddRef” on the interface. Users must call “Release” on the returned interface after they are finished using it, for proper reclamation of the resources associated with the object.	
Parameters	
_TextureView	Source texture view
Return value	Texture interface as IUnknown *
Error condition: no	

10.3 norm and unorm

The *norm* type is a single-precision floating point value that is normalized to the range [-1.0f, 1.0f]. The *unorm* type is a single-precision floating point value that is normalized to the range [0.0f, 1.0f].

10.3.1 Synopsis

```
class norm
{
public:
    norm() restrict(cpu, amp);
    explicit norm(float _V) restrict(cpu, amp);
    explicit norm(unsigned int _V) restrict(cpu, amp);
    explicit norm(int _V) restrict(cpu, amp);
    explicit norm(double _V) restrict(cpu, amp);
    norm(const norm& _Other) restrict(cpu, amp);
    norm(const unorm& _Other) restrict(cpu, amp);

    norm& operator=(const norm& _Other) restrict(cpu, amp);

    operator float(void) const restrict(cpu, amp);

    norm& operator+=(const norm& _Other) restrict(cpu, amp);
    norm& operator-=(const norm& _Other) restrict(cpu, amp);
    norm& operator*=(const norm& _Other) restrict(cpu, amp);
    norm& operator/=(const norm& _Other) restrict(cpu, amp);
    norm& operator++() restrict(cpu, amp);
    norm operator++(int) restrict(cpu, amp);
    norm& operator--() restrict(cpu, amp);
    norm operator--(int) restrict(cpu, amp);
    norm operator-() restrict(cpu, amp);
};

class unorm
{
public:
    unorm() restrict(cpu, amp);
    explicit unorm(float _V) restrict(cpu, amp);
    explicit unorm(unsigned int _V) restrict(cpu, amp);
    explicit unorm(int _V) restrict(cpu, amp);
    explicit unorm(double _V) restrict(cpu, amp);
    unorm(const unorm& _Other) restrict(cpu, amp);
    explicit unorm(const norm& _Other) restrict(cpu, amp);

    unorm& operator=(const unorm& _Other) restrict(cpu, amp);

    operator float() const restrict(cpu,amp);
```

```
    unorm& operator+=(const unorm& _Other) restrict(cpu, amp);
    unorm& operator-=(const unorm& _Other) restrict(cpu, amp);
    unorm& operator*=(const unorm& _Other) restrict(cpu, amp);
    unorm& operator/=(const unorm& _Other) restrict(cpu, amp);
    unorm& operator++() restrict(cpu, amp);
    unorm operator++(int) restrict(cpu, amp);
    unorm& operator--() restrict(cpu, amp);
    unorm operator--(int) restrict(cpu, amp);
};

unorm operator+(const unorm& lhs, const unorm& rhs) restrict(cpu, amp);
norm operator+(const norm& lhs, const norm& rhs) restrict(cpu, amp);

unorm operator-(const unorm& lhs, const unorm& rhs) restrict(cpu, amp);
norm operator-(const norm& lhs, const norm& rhs) restrict(cpu, amp);

unorm operator*(const unorm& lhs, const unorm& rhs) restrict(cpu, amp);
norm operator*(const norm& lhs, const norm& rhs) restrict(cpu, amp);

unorm operator/(const unorm& lhs, const unorm& rhs) restrict(cpu, amp);
norm operator/(const norm& lhs, const norm& rhs) restrict(cpu, amp);

bool operator==(const unorm& lhs, const unorm& rhs) restrict(cpu, amp);
bool operator==(const norm& lhs, const norm& rhs) restrict(cpu, amp);

bool operator!=(const unorm& lhs, const unorm& rhs) restrict(cpu, amp);
bool operator!=(const norm& lhs, const norm& rhs) restrict(cpu, amp);

bool operator>(const unorm& lhs, const unorm& rhs) restrict(cpu, amp);
bool operator>(const norm& lhs, const norm& rhs) restrict(cpu, amp);

bool operator<(const unorm& lhs, const unorm& rhs) restrict(cpu, amp);
bool operator<(const norm& lhs, const norm& rhs) restrict(cpu, amp);

bool operator>=(const unorm& lhs, const unorm& rhs) restrict(cpu, amp);
bool operator>=(const norm& lhs, const norm& rhs) restrict(cpu, amp);

bool operator<=(const unorm& lhs, const unorm& rhs) restrict(cpu, amp);
bool operator<=(const norm& lhs, const norm& rhs) restrict(cpu, amp);

#define UNORM_MIN ((unorm)0.0f)
#define UNORM_MAX ((unorm)1.0f)
#define UNORM_ZERO ((norm)0.0f)
#define NORM_ZERO ((norm)0.0f)
#define NORM_MIN  ((norm)-1.0f)
#define NORM_MAX  ((norm)1.0f)
```

10.3.2 Constructors and Assignment

An object of type *norm* or *unorm* can be explicitly constructed from one of the following types:

- *float*
- *double*
- *int*
- *unsigned int*
- *norm*

- *unorm*

In all these constructors, the object is initialized by first converting the argument to the *float* data type, and then clamping the value into the range defined by the type.

Assignment from *norm* to *norm* is defined, as is assignment from *unorm* to *unorm*. Assignment from other types requires an explicit conversion.

10.3.3 Operators

All arithmetic operators that are defined for the *float* type are defined for *norm* and *unorm* as well. For each supported operator ⊕, the result is computed in single-precision floating point arithmetic, and if required is then clamped back to the appropriate range.

Both *norm* and *unorm* are implicitly convertible to *float*.

10.4 Short Vector Types

C++ AMP defines a set of short vector types (of length 2, 3, and 4) which are based on one of the following scalar types: {*int, unsigned int, float, double, norm, unorm*}, and are named as summarized in the following table:

Scalar Type	Length		
	2	3	4
int	int_2, int2	int_3, int3	int_4, int4
unsigned int	uint_2, uint2	uint_3, uint3	uint_4, uint4
float	float_2, float2	float_3, float3	float_4, float4
double	double_2, double2	double_3, double3	double_4, double4
norm	norm_2, norm2	norm_3, norm3	norm_4, norm4
unorm	unorm_2, unorm2	unorm_3, unorm3	unorm_4, unorm4

There is no functional difference between the type scalar_N and *scalarN*. *scalarN* type is available in the *graphics::direct3d* namespace.

Unlike *index<N>* and *extent<N>*, short vector types have no notion of significance or endian-ness, as they are not assumed to be describing the shape of data or compute (even though a user might choose to use them this way). Also unlike extents and indices, short vector types cannot be indexed using the subscript operator.

Components of short vector types can be accessed by name. By convention, short vector type components can use either Cartesian coordinate names (“x”, “y”, “z”, and “w”), or color scalar element names (“r”, “g”, “b”, and “a”).

- For length-2 vectors, only the names “x”, “y” and “r”, “g” are available.
- For length-3 vectors, only the names “x”, “y”, “z”, and “r”, “g”, “b” are available.
- For length-4 vectors, the full set of names “x”, “y”, “z”, “w”, and “r”, “g”, “b”, “a” are available.

Note that the names derived from the color channel space (rgba) are available only as properties, not as getter and setter functions.

10.4.1 Synopsis

Because the full synopsis of all the short vector types is quite large, this section will summarize the basic structure of all the short vector types.

In the summary class definition below the word "scalartype" is one of { *int, uint, float, double, norm, unorm* }. The value *N* is 2, 3 or 4.

```
class scalartype_N
{
public:
    typedef scalartype value_type;
    static const int size = N;

    scalartype_N() restrict(cpu, amp);
    scalartype_N(scalartype value) restrict(cpu, amp);
    scalartype_N(const scalartype_N& other) restrict(cpu, amp);

    // Component-wise constructor… see 10.4.2.1 Constructors from components

    // Constructors that explicitly convert from other short vector types…
    // See 10.4.2.2 Explicit conversion constructors.

    scalartype_N& operator=(const scalartype_N& other) restrict(cpu, amp);

    // Operators
    scalartype_N& operator++() restrict(cpu, amp);
    scalartype_N operator++(int) restrict(cpu, amp);
    scalartype_N& operator--() restrict(cpu, amp);
    scalartype_N operator--(int) restrict(cpu, amp);
    scalartype_N& operator+=(const scalartype_N& rhs) restrict(cpu, amp);
    scalartype_N& operator-=(const scalartype_N& rhs) restrict(cpu, amp);
    scalartype_N& operator*=(const scalartype_N& rhs) restrict(cpu, amp);
    scalartype_N& operator/=(const scalartype_N& rhs) restrict(cpu, amp);

    // Unary negation: not for scalartype == uint or unorm
    scalartype_N operator-() const restrict(cpu, amp);

    // More integer operators (only for scalartype == int or uint)
    scalartype_N operator~() const restrict(cpu, amp);
    scalartype_N& operator%=(const scalartype_N& rhs) restrict(cpu, amp);
    scalartype_N& operator^=(const scalartype_N& rhs) restrict(cpu, amp);
    scalartype_N& operator|=(const scalartype_N& rhs) restrict(cpu, amp);
    scalartype_N& operator&=(const scalartype_N& rhs) restrict(cpu, amp);
    scalartype_N& operator>>=(const scalartype_N& rhs) restrict(cpu, amp);
    scalartype_N& operator<<=(const scalartype_N& rhs) restrict(cpu, amp);

    // Component accessors and properties (a.k.a. swizzling):
    // See 10.4.3 Component Access (Swizzling)
};

scalartype_N operator+(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
scalartype_N operator-(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
scalartype_N operator*(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
scalartype_N operator/(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
bool operator==(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
bool operator!=(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);

// More integer operators (only for scalartype == int or uint)
scalartype_N operator%(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
scalartype_N operator^(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
scalartype_N operator|(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
```

```
scalartype_N operator&(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
scalartype_N operator<<(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
scalartype_N operator>>(const scalartype_N& lhs, const scalartype_N& rhs) restrict(cpu, amp);
```

10.4.2 Constructors

*scalartype*_N()restrict(cpu,amp)
Default constructor. Initializes all components to zero.

*scalartype*_N(*scalartype* value) restrict(cpu,amp)	
Initializes all components of the short vector to 'value'.	
Parameters:	
value	The value with which to initialize each component of this vector.

*scalartype*_N(const *scalartype*_N& other) restrict(cpu,amp)	
Copy constructor. Copies the contents of 'other' to 'this'.	
Parameters:	
other	The source vector to copy from.

10.4.2.1 Constructors from components

A short vector type can also be constructed with values for each of its components.

*scalartype*_2(*scalartype* v1, *scalartype* v2) restrict(cpu,amp) // only for length 2
*scalartype*_3(*scalartype* v1, *scalartype* v2, *scalartype* v3) restrict(cpu,amp) // only for length 3
*scalartype*_4(*scalartype* v1, *scalartype* v2,
scalartype v3, *scalartype* v4) restrict(cpu,amp) // only for length 4	

Creates a short vector with the provided initialize values for each component.	
Parameters:	
v1	The value with which to initialize the "x" (or "r") component.
v2	The value with which to initialize the "y" (or "g") component
v3	The value with which to initialize the "z" (or "b") component.
v4	The value with which to initialize the "w" (or "a") component

10.4.2.2 Explicit conversion constructors

A short vector of type *scalartype1_N* can be constructed from an object of type *scalartype2_N*, as long as *N* is the same in both types. For example, a *uint_4* can be constructed from a *float_4*.

explicit *scalartype*_N(const int_N& other) restrict(cpu,amp) explicit *scalartype*_N(const uint_N& other) restrict(cpu,amp) explicit *scalartype*_N(const float_N& other) restrict(cpu,amp) explicit *scalartype*_N(const double_N& other) restrict(cpu,amp) explicit *scalartype*_N(const norm_N& other) restrict(cpu,amp) explicit *scalartype*_N(const unorm_N& other) restrict(cpu,amp)
Construct a short vector from a differently-typed short vector, performing an explicit conversion. Note that in the above list of 6 constructors, each short vector type will have 5 of these.

Parameters:	
other	The source vector to copy/convert from.

10.4.3 Component Access (Swizzling)

The components of a short vector may be accessed in a large variety of ways, depending on the length of the short vector.

- As single scalar components (N ≥ 2)
- As pairs of components, in any permutation (N ≥ 2)
- As triplets of components, in any permutation (N ≥ 3)
- As quadruplets of components, in any permutation (N = 4).

Because the permutations of such component accessors are so large, they are described here using symmetric group notation. In such notation, S_{xy} represents all permutations of the letters *x* and *y*, namely *xy* and *yx*. Similarly, S_{xyz} represents all 3! = 6 permutations of the letters *x*, *y*, and *z*, namely *xy*, *xz*, *yx*, *yz*, *zx*, and *zy*.

Recall that the *z* (or *b*) component of a short vector is only available for vector lengths 3 and 4. The *w* (or *a*) component of a short vector is only available for vector length 4.

10.4.3.1 Single-component access

scalartype get_x() const restrict(cpu,amp)
scalartype get_y() const restrict(cpu,amp)
scalartype get_z() const restrict(cpu,amp)
scalartype get_w() const restrict(cpu,amp)

void set_x(*scalartype* v) restrict(cpu,amp)
void set_y(*scalartype* v) restrict(cpu,amp)
void set_z(*scalartype* v) restrict(cpu,amp)
void set_w(*scalartype* v) restrict(cpu,amp)

__declspec(property(get=get_x, put=set_x)) *scalartype* x
__declspec(property(get=get_y, put=set_y)) *scalartype* y
__declspec(property(get=get_z, put=set_z)) *scalartype* z
__declspec(property(get=get_w, put=set_w)) *scalartype* w
__declspec(property(get=get_x, put=set_x)) *scalartype* r
__declspec(property(get=get_y, put=set_y)) *scalartype* g
__declspec(property(get=get_z, put=set_z)) *scalartype* b
__declspec(property(get=get_w, put=set_w)) *scalartype* a

These functions (and properties) allow access to individual components of a short vector type. Note that the properties in the “rgba” space map to functions in the “xyzw” space.

10.4.3.2 Two-component access

*scalartype*_2 get_S_{xy}() const restrict(cpu,amp)
*scalartype*_2 get_S_{xz}() const restrict(cpu,amp)
*scalartype*_2 get_S_{xw}() const restrict(cpu,amp)
*scalartype*_2 get_S_{yz}() const restrict(cpu,amp)
*scalartype*_2 get_S_{yw}() const restrict(cpu,amp)
*scalartype*_2 get_S_{zw}() const restrict(cpu,amp)

void set_S_{xy}(*scalartype*_2 v) restrict(cpu,amp)

void set_S_{xz}(*scalartype*_2 v) restrict(cpu,amp)
void set_S_{xw}(*scalartype*_2 v) restrict(cpu,amp)
void set_S_{yz}(*scalartype*_2 v) restrict(cpu,amp)
void set_S_{yw}(*scalartype*_2 v) restrict(cpu,amp)
void set_S_{zw}(*scalartype*_2 v) restrict(cpu,amp)

__declspec(property(get=get_S_{xy}, put=set_S_{xy})) *scalartype*_2 S_{xy}
__declspec(property(get=get_S_{xz}, put=set_S_{xz})) *scalartype*_2 S_{xz}
__declspec(property(get=get_S_{xw}, put=set_S_{xw})) *scalartype*_2 S_{xw}
__declspec(property(get=get_S_{yz}, put=set_S_{yz})) *scalartype*_2 S_{yz}
__declspec(property(get=get_S_{yw}, put=set_S_{yw})) *scalartype*_2 S_{yw}
__declspec(property(get=get_S_{zw}, put=set_S_{zw})) *scalartype*_2 S_{zw}
__declspec(property(get=get_S_{xy}, put=set_S_{xy})) *scalartype*_2 S_{rg}
__declspec(property(get=get_S_{xz}, put=set_S_{xz})) *scalartype*_2 S_{rb}
__declspec(property(get=get_S_{xw}, put=set_S_{xw})) *scalartype*_2 S_{ra}
__declspec(property(get=get_S_{yz}, put=set_S_{yz})) *scalartype*_2 S_{gb}
__declspec(property(get=get_S_{yw}, put=set_S_{yw})) *scalartype*_2 S_{ga}
__declspec(property(get=get_S_{zw}, put=set_S_{zw})) *scalartype*_2 S_{ba}

These functions (and properties) allow access to pairs of components. For example:

```
int_3  f3(1,2,3);
int_2  yz = f3.yz;  // yz = (2,3)
```

10.4.3.3 Three-component access

*scalartype*_3 get_S_{xyz}() const restrict(cpu,amp)
*scalartype*_3 get_S_{xyw}() const restrict(cpu,amp)
*scalartype*_3 get_S_{xzw}() const restrict(cpu,amp)
*scalartype*_3 get_S_{yzw}() const restrict(cpu,amp)

void set_S_{xyz}(*scalartype*_3 v) restrict(cpu,amp)
void set_S_{xyw}(*scalartype*_3 v) restrict(cpu,amp)
void set_S_{xzw}(*scalartype*_3 v) restrict(cpu,amp)
void set_S_{yzw}(*scalartype*_3 v) restrict(cpu,amp)

__declspec(property(get=get_S_{xyz}, put=set_S_{xyz})) *scalartype*_3 S_{xyz}
__declspec(property(get=get_S_{xyw}, put=set_S_{xyw})) *scalartype*_3 S_{xyw}
__declspec(property(get=get_S_{xzw}, put=set_S_{xzw})) *scalartype*_3 S_{xzw}
__declspec(property(get=get_S_{yzw}, put=set_S_{yzw})) *scalartype*_3 S_{yzw}
__declspec(property(get=get_S_{xyz}, put=set_S_{xyz})) *scalartype*_3 S_{rgb}
__declspec(property(get=get_S_{xyw}, put=set_S_{xyw})) *scalartype*_3 S_{rga}
__declspec(property(get=get_S_{xzw}, put=set_S_{xzw})) *scalartype*_3 S_{rba}
__declspec(property(get=get_S_{yzw}, put=set_S_{yzw})) *scalartype*_3 S_{gba}

These functions (and properties) allow access to triplets of components (for vectors of length 3 or 4). For example:

```
int_4  f3(1,2,3,4);
int_3  wzy = f3.wzy;  // wzy = (4,3,2)
```

10.4.3.4 Four-component access

*scalartype*_4 get_S_{xyzw}() const restrict(cpu,amp)

void set_S_{xyzw}(*scalartype*_4 v) restrict(cpu,amp)

__declspec(property(get=get_S_{xyzw}, put=set_S_{xyzw})) *scalartype_4* S_{xyzw} __declspec(property(get=get_S_{xyzw}, put=set_S_{xyzw})) *scalartype_4* S_{rgba}

These functions (and properties) allow access to all four components (obviously, only for vectors of length 4). For example:
`int_4  f3(1,2,3,4);`
`int_4  wzyx = f3.wzyw;  // wzyx = (4,3,2,1)`

10.5 Template Versions of Short Vector Types

The template class short_vector provides metaprogramming definitions of the above short vector types. These are useful for programming short vectors generically. In general, the type "*scalartype_N*" is equivalent to "short_vector<*scalartype*,*N*>::type".

10.5.1 Synopsis

```
template<typename _Scalar_type, int _Size> struct short_vector
{
    short_vector()
    {
        static_assert(false, "short_vector is not supported for this scalar type (_T) and length
(_N)");
    }
};

template<>
struct short_vector<unsigned int, 1>
{
    typedef unsigned int type;
};

template<>
struct short_vector<unsigned int, 2>
{
    typedef uint_2 type;
};

template<>
struct short_vector<unsigned int, 3>
{
    typedef uint_3 type;
};

template<>
struct short_vector<unsigned int, 4>
{
    typedef uint_4 type;
};

template<>
struct short_vector<int, 1>
{
    typedef int type;
};

template<>
struct short_vector<int, 2>
```

```
{
    typedef int_2 type;
};

template<>
struct short_vector<int, 3>
{
    typedef int_3 type;
};

template<>
struct short_vector<int, 4>
{
    typedef int_4 type;
};

template<>
struct short_vector<float, 1>
{
    typedef float type;
};

template<>
struct short_vector<float, 2>
{
    typedef float_2 type;
};

template<>
struct short_vector<float, 3>
{
    typedef float_3 type;
};

template<>
struct short_vector<float, 4>
{
    typedef float_4 type;
};

template<>
struct short_vector<unorm, 1>
{
    typedef unorm type;
};

template<>
struct short_vector<unorm, 2>
{
    typedef unorm_2 type;
};

template<>
struct short_vector<unorm, 3>
{
    typedef unorm_3 type;
};
```

```
template<>
struct short_vector<unorm, 4>
{
    typedef unorm_4 type;
};

template<>
struct short_vector<norm, 1>
{
    typedef norm type;
};

template<>
struct short_vector<norm, 2>
{
    typedef norm_2 type;
};

template<>
struct short_vector<norm, 3>
{
    typedef norm_3 type;
};

template<>
struct short_vector<norm, 4>
{
    typedef norm_4 type;
};

template<>
struct short_vector<double, 1>
{
    typedef double type;
};

template<>
struct short_vector<double, 2>
{
    typedef double_2 type;
};

template<>
struct short_vector<double, 3>
{
    typedef double_3 type;
};

template<>
struct short_vector<double, 4>
{
    typedef double_4 type;
};
```

10.5.2 short_vector<T,N> type equivalences

The equivalences of the template types “short_vector<*scalartype*,*N*>::type” to “*scalartype*_*N*” are listed in the table below:

short_vector template	Equivalent type
short_vector<unsigned int, 1>::type	unsigned int
short_vector<unsigned int, 2>::type	uint_2
short_vector<unsigned int, 3>::type	uint_3
short_vector<unsigned int, 4>::type	uint_4
short_vector<int, 1>::type	int
short_vector<int, 2>::type	int_2
short_vector<int, 3>::type	int_3
short_vector<int, 4>::type	int_4
short_vector<float, 1>::type	float
short_vector<float, 2>::type	float_2
short_vector<float, 3>::type	float_3
short_vector<float, 4>::type	float_4
short_vector<unorm, 1>::type	unorm
short_vector<unorm, 2>::type	unorm_2
short_vector<unorm, 3>::type	unorm_3
short_vector<unorm, 4>::type	unorm_4
short_vector<norm, 1>::type	norm
short_vector<norm, 2>::type	norm_2
short_vector<norm, 3>::type	norm_3
short_vector<norm, 4>::type	norm_4
short_vector<double, 1>::type	double
short_vector<double, 2>::type	double_2
short_vector<double, 3>::type	double_3
short_vector<double, 4>::type	double_4

10.6 Template class short_vector_traits

The template class short_vector_traits provides the ability to reflect on the supported short vector types and obtain the length of the vector and the underlying scalar type.

10.6.1 Synopsis

```
template<typename _Type> struct short_vector_traits
{
    short_vector_traits()
    {
        static_assert(false, "short_vector_traits is not supported for this type (_Type)");
    }
};

template<>
struct short_vector_traits<unsigned int>
{
    typedef unsigned int value_type;
```

```
    static int const size = 1;
};

template<>
struct short_vector_traits<uint_2>
{
    typedef unsigned int value_type;
    static int const size = 2;
};

template<>
struct short_vector_traits<uint_3>
{
    typedef unsigned int value_type;
    static int const size = 3;
};

template<>
struct short_vector_traits<uint_4>
{
    typedef unsigned int value_type;
    static int const size = 4;
};

template<>
struct short_vector_traits<int>
{
    typedef int value_type;
    static int const size = 1;
};

template<>
struct short_vector_traits<int_2>
{
    typedef int value_type;
    static int const size = 2;
};

template<>
struct short_vector_traits<int_3>
{
    typedef int value_type;
    static int const size = 3;
};

template<>
struct short_vector_traits<int_4>
{
    typedef int value_type;
    static int const size = 4;
};

template<>
struct short_vector_traits<float>
{
    typedef float value_type;
    static int const size = 1;
};
```

```
template<>
struct short_vector_traits<float_2>
{
    typedef float value_type;
    static int const size = 2;
};

template<>
struct short_vector_traits<float_3>
{
    typedef float value_type;
    static int const size = 3;
};

template<>
struct short_vector_traits<float_4>
{
    typedef float value_type;
    static int const size = 4;
};

template<>
struct short_vector_traits<unorm>
{
    typedef unorm value_type;
    static int const size = 1;
};

template<>
struct short_vector_traits<unorm_2>
{
    typedef unorm value_type;
    static int const size = 2;
};

template<>
struct short_vector_traits<unorm_3>
{
    typedef unorm value_type;
    static int const size = 3;
};

template<>
struct short_vector_traits<unorm_4>
{
    typedef unorm value_type;
    static int const size = 4;
};

template<>
struct short_vector_traits<norm>
{
    typedef norm value_type;
    static int const size = 1;
};

template<>
```

```
struct short_vector_traits<norm_2>
{
    typedef norm value_type;
    static int const size = 2;
};

template<>
struct short_vector_traits<norm_3>
{
    typedef norm value_type;
    static int const size = 3;
};

template<>
struct short_vector_traits<norm_4>
{
    typedef norm value_type;
    static int const size = 4;
};

template<>
struct short_vector_traits<double>
{
    typedef double value_type;
    static int const size = 1;
};

template<>
struct short_vector_traits<double_2>
{
    typedef double value_type;
    static int const size = 2;
};

template<>
struct short_vector_traits<double_3>
{
    typedef double value_type;
    static int const size = 3;
};

template<>
struct short_vector_traits<double_4>
{
    typedef double value_type;
    static int const size = 4;
};
```

10.6.2 Typedefs

typedef scalar_type *value_type*

Introduces a typedef identifying the underling scalar type of the vector type. scalar_type depends on the instantiation of class short_vector_types used. This is summarized in the list below

Instantiated Type	**Scalar Type**
short_vector_traits<unsigned int>	unsigned int
short_vector_traits<uint_2>	unsigned int

short_vector_traits<uint_3>	unsigned int
short_vector_traits<uint_4>	unsigned int
short_vector_traits<int>	int
short_vector_traits<int_2>	int
short_vector_traits<int_3>	int
short_vector_traits<int_4>	int
short_vector_traits<float>	float
short_vector_traits<float_2>	float
short_vector_traits<float_3>	float
short_vector_traits<float_4>	float
short_vector_traits<unorm>	unorm
short_vector_traits<unorm_2>	unorm
short_vector_traits<unorm_3>	unorm
short_vector_traits<unorm_4>	unorm
short_vector_traits<norm>	norm
short_vector_traits<norm_2>	norm
short_vector_traits<norm_3>	norm
short_vector_traits<norm_4>	norm
short_vector_traits<double>	double
short_vector_traits<double_2>	double
short_vector_traits<double_3>	double
short_vector_traits<double_4>	double

10.6.3 Members

```
static int const size;
```

Introduces a static constant integer specifying the number of elements in the short vector type, based on the table below:

Instantiated Type	Size
short_vector_traits<unsigned int>	1
short_vector_traits<uint_2>	2
short_vector_traits<uint_3>	3
short_vector_traits<uint_4>	4
short_vector_traits<int>	1
short_vector_traits<int_2>	2
short_vector_traits<int_3>	3
short_vector_traits<int_4>	4
short_vector_traits<float>	1
short_vector_traits<float_2>	2
short_vector_traits<float_3>	3
short_vector_traits<float_4>	4
short_vector_traits<unorm>	1
short_vector_traits<unorm_2>	2
short_vector_traits<unorm_3>	3
short_vector_traits<unorm_4>	4

short_vector_traits<norm>	1	
short_vector_traits<norm_2>	2	
short_vector_traits<norm_3>	3	
short_vector_traits<norm_4>	4	
short_vector_traits<double>	1	
short_vector_traits<double_2>	2	
short_vector_traits<double_3>	3	
short_vector_traits<double_4>	4	

11 D3D interoperability (Optional)

The C++ AMP runtime provides functions for D3D interoperability, enabling seamless use of D3D resources for compute in C++ AMP code as well as allow use of resources created in C++ AMP in D3D code, without the creation of redundant intermediate copies. These features allow users to incrementally accelerate the compute intensive portions of their DirectX applications using C++ AMP and use the D3D API on data produced from C++ AMP computations.

The following D3D interoperability functions are available in the *direct3d* namespace:

```
accelerator_view create_accelerator_view(IUnknown *_D3d_device_interface)
```

Creates a new *accelerator_view* from an existing Direct3D device interface pointer. On failure the function throws a *runtime_exception* exception. On success, the reference count of the parameter is incremented by making a *AddRef* call on the interface to record the C++ AMP reference to the interface, and users can safely *Release* the object when no longer required in their DirectX code.

The *accelerator_view* created using this function is thread-safe just as any C++ AMP created *accelerator_view*, allowing concurrent submission of commands to it from multiple host threads. However, concurrent use of the *accelerator_view* and the raw *ID3D11Device* interface from multiple host threads must be properly synchronized by users to ensure mutual exclusion. Unsynchronized concurrent usage of the *accelerator_view* and the raw *ID3D11Device* interface will result in undefined behavior.

The C++ AMP runtime provides detailed error information in debug mode using the Direct3D Debug layer. However, if the Direct3D device passed to the above function was not created with the *D3D11_CREATE_DEVICE_DEBUG* flag, the C++ AMP debug mode detailed error information support will be unavailable.

Parameters:

_D3d_device_interface	An AMP supported D3D device interface pointer to be used to create the accelerator_view. The parameter must meet all of the following conditions for successful creation of a accelerator_view: 1) Must be a supported D3D device interface. For this release, only ID3D11Device interface is supported. 2) The device must have an AMP supported feature level. For this release this means a D3D_FEATURE_LEVEL_11_0. or D3D_FEATURE_LEVEL_11_1 3) The D3D Device should not have been created with the “D3D11_CREATE_DEVICE_SINGLETHREADED” flag.

Return Value:

The newly created accelerator_view object.

Exceptions:

runtime_exception	1) "Failed to create accelerator_view from D3D device.", E_INVALIDARG 2) "NULL D3D device pointer.", E_INVALIDARG

IUnknown * get_device(const accelerator_view &_Rv)	
Returns a D3D device interface pointer underlying the passed accelerator_view. Fails with a "runtime_exception" exception if the passed accelerator_view is not a D3D device accelerator view. On success, it increments the reference count of the D3D device interface by calling "AddRef" on the interface. Users must call "Release" on the returned interface after they are finished using it, for proper reclamation of the resources associated with the object. Concurrent use of the accelerator_view and the raw ID3D11Device interface from multiple host threads must be properly synchronized by users to ensure mutual exclusion. Unsynchronized concurrent usage of the accelerator_view and the raw ID3D11Device interface will result in undefined behavior.	
Parameters:	
_Rv	The accelerator_view object for which the D3D device interface is needed.
Return Value:	
A IUnknown interface pointer corresponding to the D3D device underlying the passed accelerator_view. Users must use the *QueryInterface* member function on the returned interface to obtain the correct D3D device interface pointer.	
Exceptions:	
runtime_exception	"Cannot get D3D device from a non-D3D accelerator_view.", E_INVALIDARG

template <typename T, int N> array<T,N> make_array(const extent<N> &_Extent, const accelerator_view &_Rv, IUnknown *_D3d_buffer_interface)	
Creates an array with the specified extents on the specified accelerator_view from an existing Direct3D buffer interface pointer. On failure the member function throws a *runtime_exception* exception. On success, the reference count of the Direct3D buffer object is incremented by making an *AddRef* call on the interface to record the C++ AMP reference to the interface, and users can safely *Release* the object when no longer required in their DirectX code.	
Parameters:	
_Extent	The extent of the array to be created.
_Rv	The accelerator_view that the array is to be created on.
_D3d_buffer_interface	AN AMP supported D3D device buffer pointer to be used to create the array. The parameter must meet all of the following conditions for successful creation of a accelerator_view: 1) Must be a supported D3D buffer interface. For this release, only ID3D11Buffer interface is supported. 2) The D3D device on which the buffer was created must be the same as that underlying the accelerator_view parameter *rv*. 3) The D3D buffer must additionally satisfy the following conditions: a. The buffer size in bytes must be greater than or equal to the size in bytes of the field to be created (g.get_size() * sizeof(_Elem_type)). b. Must not have been created with D3D11_USAGE_STAGING.

	c. SHADER_RESOURCE and/or UNORDERED_ACCESS bindings should be allowed for the buffer. d. Raw views must be allowed for the buffer (e.g. D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS).
Return Value:	
The newly created array object.	
Exceptions:	
runtime_exception	1) "Invalid extents argument.", E_INVALIDARG 2) "NULL D3D buffer pointer.", E_INVALIDARG 3) "Invalid D3D buffer argument.", E_INVALIDARG 4) "Cannot create D3D buffer on a non-D3D accelerator_view.", E_INVALIDARG

```
template <size_t RANK, typename _Elem_type>
IUnknown * get_buffer(const array<_Elem_type, RANK> &_F)
```

Returns a D3D buffer interface pointer underlying the passed array. Fails with a "runtime_exception" exception of the passed array is not on a D3D device resource view. On success, it increments the reference count of the D3D buffer interface by calling "AddRef" on the interface. Users must call "Release" on the returned interface after they are finished using it, for proper reclamation of the resources associated with the object.	
Parameters:	
_F	The `array` for which the underlying D3D buffer interface is needed.
Return Value:	
A IUnknown interface pointer corresponding to the D3D buffer underlying the passed `array`. Users must use the QueryInterface member function on the returned interface to obtain the correct D3D buffer interface pointer.	
Exceptions:	
runtime_exception	"Cannot get D3D buffer from a non-D3D array.", E_INVALIDARG

12 Error Handling

12.1 static_assert

The C++ intrinsic *static_assert* is often used to handle error states that are detectable at compile time. In this way *static_assert* is a technique for conveying static semantic errors and as such they will be categorized similar to exception types.

12.2 Runtime errors

On encountering an irrecoverable error, C++ AMP runtime throws a C++ exception to communicate/propagate the error to client code. (Note: exceptions are not thrown from *restrict(amp)* code.) The actual exceptions thrown by each API are listed in the API descriptions. Following are the exception types thrown by C++ AMP runtime:

12.2.1 runtime_exception

A *runtime_exception* instance comprises a textual description of the error and a *HRESULT* error code to indicate the cause of the error.

`class runtime_exception`
The exception type that all AMP runtime exceptions derive from. A *runtime_exception* instance comprises of a textual description of the error and a HRESULT error code to indicate the cause of the error.

`runtime_exception(const char * _Message, HRESULT _Hresult) throw()`	
Construct a runtime_exception exception with the specified message and HRESULT error code.	
Parameters:	
_Message	Descriptive message of error
_Hresult	HRESULT error code that caused this exception

`runtime_exception (HRESULT _Hresult) throw()`	
Construct a runtime_exception exception with the specified HRESULT error code.	
Parameters:	
_Hresult	HRESULT error code that caused this exception

`HRESULT get_error_code() const throw()`
Returns the error code that caused **this** exception.
Return Value:
Returns the HRESULT error code that caused **this** exception.

12.2.1.1 Specific Runtime Exceptions

Exception String	Source	Explanation
No supported accelerator available.	Accelerator constructor, array constructor	No device available at runtime supports C++ AMP.
Failed to create buffer	Array constructor	Couldn't create buffer on accelerator, likely due to lack of resource availability.

12.2.2 out_of_memory

An instance of this exception type is thrown when an underlying OS/DirectX API call fails due to failure to allocate system or device memory (*E_OUTOFMEMORY HRESULT* error code). Note that if the runtime fails to allocate memory from the heap using the C++ *new* operator, a *std::bad_alloc* exception is thrown and not the C++ AMP *out_of_memory* exception.

`class out_of_memory : public runtime_exception`
Exception thrown when an underlying OS/DirectX call fails due to lack of system or device memory.

`explicit out_of_memory(const char * _Message) throw()`	
Construct a out_of_memory exception with the specified message.	
Parameters:	
_Message	Descriptive message of error

`out_of_memory() throw()`
Construct a out_of_memory exception.
Parameters:
None.

12.2.3 invalid_compute_domain

An instance of this exception type is thrown when the runtime fails to devise a dispatch for the compute domain specified at a *parallel_for_each* call site.

`class invalid_compute_domain : public runtime_exception`
Exception thrown when the runtime fails to launch a kernel using the compute domain specified at the parallel_for_each call site.

`explicit invalid_compute_domain(const char * _Message) throw()`	
Construct an invalid_compute_domain exception with the specified message.	
Parameters:	
_Message	Descriptive message of error

`invalid_compute_domain() throw()`
Construct an invalid_compute_domain exception.
Parameters:
None.

12.2.4 unsupported_feature

An instance of this exception type is thrown on executing a *restrict(amp)* function on the host which uses an intrinsic unsupported on the host (such as *tiled_index<>::barrier.wait()*) or when invoking a *parallel_for_each* or allocating an object on an accelerator which doesn't support certain features which are required for the execution to proceed, such as, but not limited to:

1. The accelerator is not capable of executing code, but serves as a memory allocation arena only
2. The accelerator doesn't support the allocation of textures
3. A texture object is created with an invalid combination of bits_per_scalar_element and short-vector type
4. Read and write operations are both requested on a texture object with bits_per_scalar != 32

`class unsupported_feature : public runtime_exception`
Exception thrown when an unsupported feature is used.

`explicit unsupported_feature (const char * _Message) throw()`	
Construct an unsupported_feature exception with the specified message.	
Parameters:	
_Message	Descriptive message of error

`unsupported_feature () throw()`
Construct an unsupported_feature exception.
Parameters:
None.

12.2.5 accelerator_view_removed

An instance of this exception type is thrown when the C++ AMP runtime detects that a connection with a particular accelerator, represented by an instance of class accelerator_view, has been lost. When such an incident happens, all data allocated through the accelerator view and all in-progress computations on the accelerator view may be lost. This exception may be thrown by *parallel_for_each*, as well as any other copying and/or synchronization method.

`class accelerator_view_removed : public runtime_exception`
HRESULT error code indicating the cause of removal of the accelerator_view

explicit accelerator_view_removed(const char * _Message, HRESULT _View_removed_reason) throw(); explicit accelerator_view_removed(HRESULT _View_removed_reason) throw();	
Construct an accelerator_view_removed exception with the specified message and HRESULT	
Parameters:	
_Message	Descriptive message of error
_HRESULT	HRESULT error code indicating the cause of removal of the accelerator_view

HRESULT get_view_removed_reason() const throw();
Provides the HRESULT error code indicating the cause of removal of the accelerator_view
Return Value:
The HRESULT error code indicating the cause of removal of the accelerator_view

12.3 Error handling in device code (amp-restricted functions) (Optional)

The use of the *throw* C++ keyword is disallowed in C++ AMP vector functions (*amp* restricted) and will result in a compilation error. C++ AMP offers the following intrinsics in vector code for error handling.

Microsoft-specific: *the Microsoft implementation of C++ AMP provides the methods specified in this section, provided all of the following conditions are met.*

1. *The debug version of the runtime is being used (i.e. the code is compiled with the _DEBUG preprocessor definition).*
2. *The debug layer is available on the system. This, in turn requires DirectX SDK to be installed on the system on Windows 7. On Windows 8 no SDK intallation is necessary..*
3. *The accelerator_view on which the kernel is invoked must be on a device which supports the printf and abort intrinsics. As of the date of writing this document, only the REF device supports these intrinsics.*

When the debug version of the runtime is not used or the debug layer is unavailable, executing a kernel that using these intrinsics through a parallel_for_each call will result in a runtime exception. On devices that do not support these intrinsics, these intrinsics will behave as no-ops.

`void direct3d_printf(const char *_Format_string, …) restrict(amp)`	
Prints formatted output from a kernel to the debug output. The formatting semantics are same as the C Library printf function. Also, this function is executed as any other device-side function: per-thread, and in the context of the calling thread. Due to the asynchronous nature of kernel execution, the output from this call may appear anytime between the launch of the kernel containing the printf call and completion of the kernel's execution.	
Parameters:	
_Format_string	The format string.
…	An optional list of parameters of variable count.
Return Value:	
None.	

`void direct3d_errorf(char *_Format_string, …) restrict(amp)`	
This intrinsic prints formatted error messages from a kernel to the debug output. This function is executed as any other device-side function: per-thread, and in the context of the calling thread. Note that due to the asynchronous nature of kernel execution, the actual error messages may appear in the debug output asynchronously, any time between the dispatch of the kernel and the completion of the kernel's execution. When these error messages are detected by the runtime, it raises a "runtime_exception" exception on the host with the formatted error message output as the exception message.	
Parameters:	
_Format_string	The format string.
…	An optional list of parameters of variable count.

`void direct3d_abort() restrict(amp)`
This intrinsic aborts the execution of threads in the compute domain of a kernel invocation, that execute this instruction. This function is executed as any other device-side function: per-thread, and in the context of the calling thread. Also the thread is terminated without executing any destructors for local variables. When the abort is detected by the runtime, it raises a "runtime_exception" exception on the host with the abort output as the exception message. Note that due to the asynchronous nature of kernel execution, the actual abort may be detected any time between the dispatch of the kernel and the completion of the kernel's execution.

Due to the asynchronous nature of kernel execution, the *direct3d_printf*, *direct3d_errorf* and *direct3d_abort* messages from kernels executing on a device appear asynchronously during the execution of the shader or after its completion and not immediately after the async launch of the kernel. Thus these messages from a kernel may be interleaved with messages from other kernels executing concurrently or error messages from other runtime calls in the debug output. It is the programmer's responsibility to include appropriate information in the messages originating from kernels to indicate the origin of the messages.

13 Appendix: C++ AMP Future Directions (Informative)

It is likely that C++ AMP will evolve over time. The set of features allowed inside *amp*-restricted functions will grow. However, compilers will have to continue to support older hardware targets which only support the previous, smaller feature set. This section outlines possible such evolution of the language syntax and associated feature set.

13.1 Versioning Restrictions

This section contains an informative description of additional language syntax and rules to allow the versioning of C++ AMP code. If an implementation desires to extend C++ AMP in a manner not covered by this version of the specification, it is recommended that it follows the syntax and rules specified here.

13.1.1 *auto* restriction

The *restriction* production (section 2.1) of the C++ grammar is amended to allow the contextual keyword **auto**.

restriction:
  amp-restriction
  cpu
  auto

A function or lambda which is annotated with *restrict(auto)* directs the compiler to check all known restrictions and automatically deduce the set of restrictions that a function complies with. *restrict(auto)* is only allowed for functions where the function declaration is also a function definition, and no other declaration of the same function occurs.

A function may be simultaneously explicitly and *auto* restricted, e.g., *restrict(cpu,auto)*. In such case, it will be explicitly checked for compulsory conformance with the set of explicitly specified (non-auto) restrictions, and implicitly checked for possible conformance with all other restrictions that the compiler supports.

Consider the following example:

```
int f1() restrict(amp);

int f2() restrict(cpu,auto)
{
    f1();
}
```

In this example, *f2* is verified for compulsory adherence to the *restrict(cpu)* restriction. This results in an error, since *f2* calls *f1*, which is not *cpu*-restricted. Had we changed f1's restriction to *restrict(cpu)*, then *f2* will pass the adherence test to the explicitly specified *restrict(cpu)*. Now with respect to the *auto* restriction, the compiler has to check whether *f2* conforms to *restrict(amp)*, which is the only other restriction not explicitly specified. In the context of verifying the plausibility of inferring an *amp*-restriction for *f2*, the compiler notices that *f2* calls *f1*, which is, in our modified example, not *amp*-restricted, and therefore *f2* is also inferred to be not *amp*-restricted. Thus the total inferred restriction for *f2* is *restrict(cpu)*. If we now change the restriction for *f1* into *restrict(cpu,amp)*, then the inference for *f2* would reach the conclusion that *f2* is *restrict(cpu,amp)* too.

When two overloads are available to call from a given restriction context, and they differ only by the fact that one is explicitly restricted while the other is implicitly inferred to be restricted, the explicitly restricted overload shall be chosen.

13.1.2 Automatic restriction deduction

Implementations are encouraged to support a mode in which functions that have their definitions accompany their declarations, and where no other declarations occur for such functions, have their restriction set automatically deduced.

In such a mode, when the compiler encounters a function declaration which is also a definition, and a previous declaration for the function hasn't been encountered before, then the compiler analyses the function as if it was restricted with *restrict(cpu,auto)*. This allows easy reuse of existing code in *amp*-restricted code, at the cost of prolonged compilation times.

13.1.3 *amp* Version

The *amp-restriction* production of the C++ grammar is amended thus:

amp-restriction:
  amp *amp-version*$_{opt}$

amp-version:
  : *integer-constant*
  : *integer-constant* **.** *integer-constant*

An *amp* version specifies the lowest version of amp that this function supports. In other words, if a function is decorated with *restrict(amp:1)*, then that function also supports any version greater or equal to 1. When the *amp* version is elided, the implied version is implementation-defined. Implementations are encouraged to support a compiler flag controlling the default version assumed. When versioning is used in conjunction with *restrict(auto)* and/or automatic restriction deduction, the compiler shall infer the maximal version of the *amp* restriction that the function adheres to.

Section 2.3.2 specifies that restriction specifiers of a function shall not overlap with any restriction specifiers in another function within the same overload set.

```
int func(int x) restrict(cpu,amp);
int func(int x) restrict(cpu);   // error, overlaps with previous declaration
```

This rule is relaxed in the case of versioning: functions overloaded with *amp* versions are not considered to overlap:

```
int func(int x) restrict(cpu);
int func(int x) restrict(amp:1);
int func(int x) restrict(amp:2);
```

When an overload set contains multiple versions of the amp specifier, the function with the highest version number that is not higher than the callee is chosen:

```
void glorp() restrict(amp:1) { }
void glorp() restrict(amp:2) { }

void glorp_caller() restrict(amp:2) {
    glorp(); // okay; resolves to call "glorp() restrict(amp:2)"
}
```

13.2 Projected Evolution of *amp*-Restricted Code

Based on the nascent availability of features in advanced GPUs and corresponding hardware-vendor-specific programming models, it is apparent that the limitations associated with *restrict(amp)* will be gradually lifted. The table below captures one possible path for future *amp* versions to follow. If implementers need to (non-normatively) extend the *amp*-restricted language subset, it is recommended that they consult the table below and try to conform to its style.

Implementations may not define an amp version greater or equal to 2.0. All non-normative extensions shall be restricted to the patterns 1.x (where x > 0). Version number 1.0 is reserved to implementations strictly adhering to this version of the specification, while version number 2.0 is reserved for the next major version of this specification.

Area	Feature	amp:1	amp:1.1	amp:1.2	amp:2	cpu
Local/Param/Function Return	char (8 - signed/unsigned/plain)	No	Yes	Yes	Yes	Yes
Local/Param/Function Return	short (16 - signed/unsigned)	No	Yes	Yes	Yes	Yes
Local/Param/Function Return	int (32 - signed/unsigned)	Yes	Yes	Yes	Yes	Yes
Local/Param/Function Return	long (32 - signed/unsigned)	Yes	Yes	Yes	Yes	Yes
Local/Param/Function Return	long long (64 - signed/unsigned)	No	No	Yes	Yes	Yes
Local/Param/Function Return	half-precision float (16)	No	No	No	No	No
Local/Param/Function Return	float (32)	Yes	Yes	Yes	Yes	Yes
Local/Param/Function Return	double (64)	Yes[10]	Yes	Yes	Yes	Yes
Local/Param/Function Return	long double (?)	No	No	No	No	Yes
Local/Param/Function Return	bool (8)	Yes	Yes	Yes	Yes	Yes
Local/Param/Function Return	wchar_t (16)	No	Yes	Yes	Yes	Yes
Local/Param/Function Return	Pointer (single-indirection)	Yes	Yes	Yes	Yes	Yes
Local/Param/Function Return	Pointer (multiple-indirection)	No	No	Yes	Yes	Yes
Local/Param/Function Return	Reference	Yes	Yes	Yes	Yes	Yes
Local/Param/Function Return	Reference to pointer	Yes	Yes	Yes	Yes	Yes
Local/Param/Function Return	Reference/pointer to function	No	No	Yes	Yes	Yes
Local/Param/Function Return	static local	No	No	Yes	Yes	Yes
Struct/class/union members	char (8 - signed/unsigned/plain)	No	Yes	Yes	Yes	Yes
Struct/class/union members	short (16 - signed/unsigned)	No	Yes	Yes	Yes	Yes
Struct/class/union members	int (32 - signed/unsigned)	Yes	Yes	Yes	Yes	Yes
Struct/class/union members	long (32 - signed/unsigned)	Yes	Yes	Yes	Yes	Yes
Struct/class/union members	long long (64 - signed/unsigned)	No	No	Yes	Yes	Yes
Struct/class/union members	half-precision float (16)	No	No	No	No	No
Struct/class/union members	float (32)	Yes	Yes	Yes	Yes	Yes
Struct/class/union members	double (64)	Yes	Yes	Yes	Yes	Yes
Struct/class/union members	long double (?)	No	No	No	No	Yes
Struct/class/union members	bool (8)	No	Yes	Yes	Yes	Yes
Struct/class/union members	wchar_t (16)	No	Yes	Yes	Yes	Yes
Struct/class/union members	Pointer	No	No	Yes	Yes	Yes
Struct/class/union members	Reference	No	No	Yes	Yes	Yes
Struct/class/union members	Reference/pointer to function	No	No	No	Yes	Yes
Struct/class/union members	bitfields	No	No	No	Yes	Yes
Struct/class/union members	unaligned members	No	No	No	No	Yes
Struct/class/union members	pointer-to-member (data)	No	No	Yes	Yes	Yes
Struct/class/union members	pointer-to-member (function)	No	No	Yes	Yes	Yes
Struct/class/union members	static data members	No	No	No	Yes	Yes

[10] Double precision support is an optional feature on some amp:1-compliant hardware.

Struct/class/union members	static member functions	Yes	Yes	Yes	Yes	Yes
Struct/class/union members	non-static member functions	Yes	Yes	Yes	Yes	Yes
Struct/class/union members	Virtual member functions	No	No	Yes	Yes	Yes
Struct/class/union members	Constructors	Yes	Yes	Yes	Yes	Yes
Struct/class/union members	Destructors	Yes	Yes	Yes	Yes	Yes
Enums	char (8 - signed/unsigned/plain)	No	Yes	Yes	Yes	Yes
Enums	short (16 - signed/unsigned)	No	Yes	Yes	Yes	Yes
Enums	int (32 - signed/unsigned)	Yes	Yes	Yes	Yes	Yes
Enums	long (32 - signed/unsigned)	Yes	Yes	Yes	Yes	Yes
Enums	long long (64 - signed/unsigned)	No	No	No	No	Yes
Structs/Classes	Non-virtual base classes	Yes	Yes	Yes	Yes	Yes
Structs/Classes	Virtual base classes	No	Yes	Yes	Yes	Yes
Arrays	of pointers	No	No	Yes	Yes	Yes
Arrays	of arrays	Yes	Yes	Yes	Yes	Yes
Declarations	tile_static	Yes	Yes	Yes	Yes	No
Function Declarators	Varargs (…)	No	No	No	No	Yes
Function Declarators	throw() specification	No	No	No	No	Yes
Statements	global variables	No	No	No	Yes	Yes
Statements	static class members	No	No	No	Yes	Yes
Statements	Lambda capture-by-reference (on gpu)	No	No	Yes	Yes	Yes
Statements	Lambda capture-by-reference (in p_f_e)	No	No	No	Yes	Yes
Statements	Recursive function call	No	No	Yes	Yes	Yes
Statements	conversion between pointer and integral	No	Yes	Yes	Yes	Yes
Statements	new	No	No	Yes	Yes	Yes
Statements	delete	No	No	Yes	Yes	Yes
Statements	dynamic_cast	No	No	No	No	Yes
Statements	typeid	No	No	No	No	Yes
Statements	goto	No	No	No	No	Yes
Statements	labels	No	No	No	No	Yes
Statements	asm	No	No	No	No	Yes
Statements	throw	No	No	No	No	Yes
Statements	try/catch	No	No	No	No	Yes
Statements	__try/__except	No	No	No	No	Yes
Statements	__leave	No	No	No	No	Yes