Intel(R) Array Building Blocks Virtual Machine

Specification

Document Number: 324820-003US

Legal Information

Contents

Legal Information

Introducing the Intel® Array Building Blocks

You can view Intel® Array Building Blocks (Intel® ArBB) from several different perspectives.

It can be seen as any of the following:

- A language.

  Intel ArBB is a full programming language that supports the specification and implementation of arbitrary parallel computations. However, it is an embedded language since it expresses all its types, control flow constructs, and operators using standard C++ functionality. Intel ArBB can be seen as a parallel extension to C++ that you can use together with existing C++ functionality.
- A programming model.

  Intel ArBB supports a programming model based on the dynamic composition of structured patterns of parallel computation. It enables the construction of parallel programs by expressing operations at the aggregate data collection level as well as at the element level. Intel ArBB supports a set of deterministic parallel patterns with consistent serial interpretations. Composing programs using these patterns results in deterministic programs that avoid race conditions and deadlocks by construction and are easier to debug and understand than unstructured parallel programs.
- An abstract virtual machine.

  Intel ArBB semantics are based on the underlying abstract model of parallelism often referred to as the Intel ArBB Virtual Machine (VM). Abstract parallel operations expressed in the VM semantics are automatically mapped onto the underlying hardware mechanisms that enhance performance. These mechanisms include vector instructions, multiple cores, prefetching and streaming for parallelizing memory access. However, the high-level abstraction of the Intel ArBB virtual machine avoids dependencies on the details of the underlying system, making Intel ArBB code more portable and easier to write and maintain. Finally, the VM has its own API that can be used to implement front-ends for other languages in addition to C++.
- An Application Programming Interface (API).

  Intel ArBB uses C++ templates to introduce new types. It also uses operator overloading and library calls to introduce new operators. To compile Intel ArBB programs, you can use it as a library with standard C++ compilers. All Intel ArBB functionality is implemented through its library interface. Of particular interest are the call and map functions that compile sequences of operations expressed over Intel ArBB types and can apply them in turn as aggregate and element-wise parallel operations. These functions take as parameters the name and parameters of a standard C++ function that uses Intel ArBB types. The dynamic compiler retargets the expressed sequences of operations for the target architecture and caches the dynamically compiled machine language for reuse on subsequent call or map invocations. You can also translate Intel ArBB functions in advance and store them in the closure objects for later use. Thus, Intel ArBB provides a full metaprogramming API for dynamically generating and manipulating parallel code. This provides a generic programming interface that can help you avoid the modularity overhead of C++.

Notational Conventions

This document uses the Backus Normal Form (BNF) notation for grammar of a textual representation of the Application Programming Interface (API). For example,

```
module          ::= module-item-seq

module-item-seq ::= module-item | module-item module-item-seq

module-item     ::= function-defn | function-decl | global-decl

                  | meta-definition
```

Each sequence of non-blank characters on the right-hand side of `:=` represents a possible substitution for the sequence of characters on the left-hand side of `::=` . A vertical bar indicates a choice among the substitution options. Each sequence on the right of `::=` either appears on the left or is a *terminal*. Terminals never appear to the left of `::=` . However, a sequence that appears only on the right of `::=` in a topic may not be a terminal. It may appear to the left of `::=` in a different topic. In this case, the sequence has a hyperlink to its substitution options.

Font Conventions

The following font conventions are used in this document:

`courier`	Code examples and code fragments of the C interface. For example, `function callee(out $f32 a, in $f32 b, in $f32 c);` Syntax in the BNF notation. For example, `global-decl ::= 'global' type identifier ';'`
`courier italic`	Parameters of the API statements. For example, *`s0`*.

Overview

1

This document specifies the Intel® Array Building Blocks (Intel® ArBB) Virtual Machine (VM) interface. This interface consists of:

- An Intermediate Representation (IR) that defines data structures and code within the virtual machine.
- An execution Application Programming Interface (API) that enables code defined using the Virtual Machine IR to be executed through the virtual machine.

Intermediate Representation Definition Interfaces

There are two ways to define the Intel ArBB VM IR:

- Using the *streaming API*.

  IR operations can be streamed into the VM programmatically, appending operations to an IR representation in a linear fashion. This technique is primarily intended for other front ends to generate IR during static or run-time compilation. The streaming API enables providing IR without any significant parsing effort (for example, look-ahead) and provides instant feedback on IR errors.
- Using the *serialized format*.

  IR operations can be represented in an on-disk storage format. This representation is primarily intended for development and debugging. The serialized format is easy to read, write, and modify.

Overview of the Virtual Machine Intermediate Representation

C API Overview

The Intel ArBB VM API is a set of C APIs to build the Intel ArBB VM IR.

The `arbb_vmapi.h` header file provides declarations for all the functions (see the *Intel ArBB User's Guide* for the location of the header file). For a C++ compiler to parse this header, all declarations must be `extern "C"` declarations, as shown below:

```
#ifndef ARBB_VM_API_H
#define ARBB_VM_API_H
#if defined(_WIN32)
#  if defined(OPENCT_DLL_EXPORTS)
#    define ARBB_VM_EXPORT __declspec(dllexport)
#  else
#    define ARBB_VM_EXPORT __declspec(dllimport)
#  endif
#else
#  if defined(OPENCT_DLL_EXPORTS)
#    define ARBB_VM_EXPORT __attribute__((visibility("default")))
#  else
#    define ARBB_VM_EXPORT
#  endif
#endif
#if defined(_WIN32)
  typedef __int64 int64_t;
  typedef unsigned __int64 uint64_t;
#else
```

```
  #include <stdint.h>
#endif
#ifdef __cplusplus
extern "C" {
#endif
/// @addtogroup arbb_virtual_machine
/// @{
```

Types Overview

The table below lists the types provided in the Intel ArBB VM API:

Type Name	Category	Description
arbb_context_t	Opaque	A context within which VM calls are being made.
arbb_refcountable_t	Opaque	A reference-countable object in the VM API.
arbb_error_details_t	Opaque	A structure representing detailed error information from a function call.
arbb_global_variable_t	Opaque	A global variable or constant.
arbb_variable_t	Opaque	A variable in the VM.
arbb_function_t	Opaque	A function in the VM.
arbb_type_t	Opaque	A type for a variable or function.
arbb_binding_t	Opaque	A binding specification to indicate a binding between host data and global variables.
arbb_string_t	Opaque	A string returned by the VM API.
arbb_cxx_stack_trace_t	Opaque	A C/C++ stack trace.
arbb_cxx_frame_t	Opaque	A frame in a C/C++ stack trace.
arbb_cxx_frame_property_t	Enumeration	A set of frame properties that can be queried.
arbb_attribute_key_t	uint32_t	A key to be used to index attribute values.
arbb_attribute_value_t	Union	An attribute value.
arbb_attribute_key_value_t	Struct	An attribute key-value pair.
arbb_attribute_map_t	Opaque	The set of key-value pairs that may be attached to basic block statements or function definitions.
arbb_attribute_type_t	Enumeration	The set of supported attribute value types.
arbb_source_info_provider_t	Callback	A callback function that the VM calls to request a filename and line number information from the front end.
arbb_opaque_type_t	Opaque	Generic opaque type that represents any opaque type.
arbb_opaque_tag_t	Enumeration	The set of opaque types used by the VM API.
arbb_error_t	Enumeration	An error code returned from a function.
arbb_scalar_type_t	Enumeration	A type code representing a primitive scalar type.

Type Name	Category	Description
`arbb_opcode_t`	Enumeration	An operation code representing one of the instructions available in the VM.
`arbb_call_opcode_t`	Enumeration	An operation code representing an instruction that accepts a function.
`arbb_loop_type_t`	Enumeration	A loop type, for example, `arbb_loop_for` or `arbb_loop_while`.
`arbb_loop_block_t`	Enumeration	A loop block type, for example, `arbb_loop_block_cond` or `arbb_loop_block_body`.
`arbb_range_access_mode_t`	Enumeration	The set of access modes allowed for `arbb_map_to_host()`.

Opaque Types

You can use opaque types only as parameters of functions defined by the API. An application that uses the API cannot manipulate or inspect opaque types directly.

The following are definitions of the opaque types:

```
/// @defgroup arbb_virtual_machine_opaque_types Opaque Types
/// Opaque types provided by the virtual machine API.
/// @{
/// A context within which VM calls are being made.
typedef void* arbb_context_t;
/// A reference-countable object in the VM API.
/// @see @ref arbb_virtual_machine_refcounting
typedef void* arbb_refcountable_t;
/// A structure representing detailed error information from a function call.
///
/// @see @ref arbb_virtual_machine_errors
typedef void* arbb_error_details_t;
/// A global variable or constant.
/// This structure can be converted to an arbb_refcountable_t.
typedef void* arbb_global_variable_t;
/// A variable in the VM.
typedef void* arbb_variable_t;
/// A function in the VM.
///
/// @see @ref arbb_virtual_machine_functions
typedef void* arbb_function_t;
/// A type for a variable or function.
typedef void* arbb_type_t;
/// A binding specification to indicate a binding between host data and global variables.
///
/// @see @ref arbb_virtual_machine_binding
typedef void* arbb_binding_t;
/// A string returned by the VM API.
typedef void* arbb_string_t;
/// A C/C++ stack trace.
/// @see arbb_cxx_store_stack_trace, arbb_cxx_release_stack_trace, arbb_cxx_get_frame_count
typedef void* arbb_cxx_stack_trace_t;
/// A frame in a C/C++ stack trace.
/// @see arbb_cxx_get_frame, arbb_cxx_get_frame_property
typedef void* arbb_cxx_frame_t;
/// A set of associated key-value pairs which may be attached to ArBB basic
/// block statements or function definitions.
/// @see arbb_create_attribute_map
typedef void* arbb_attribute_map_t;
/// Generic opaque type used to represent any opaque type.
typedef void* arbb_opaque_type_t;
/// The set of the opaque types used by the VM API.
typedef enum {
  arbb_opaque_context,
  arbb_opaque_refcountable,
  arbb_opaque_error_details,
  arbb_opaque_global_variable,
  arbb_opaque_variable,
  arbb_opaque_function,
  arbb_opaque_type,
  arbb_opaque_binding,
  arbb_opaque_string,
  arbb_opaque_cxx_stack_trace,
  arbb_opaque_cxx_frame,
  arbb_opaque_attribute_map
} arbb_opaque_tag_t;

/// @}
```

Non-opaque Types

Enumeration, struct, union and other specific types are defined using C syntax. For details of specific types, see Types.

Error Handling

Most Intel ArBB API functions return an error code of type `arbb_error_t`. This error code has one of the following values:

Error code (value of `arbb_error_t`)	**Meaning**
`arbb_error_none`	No error occurs.
`arbb_error_invalid_argument`	One or more parameters provided to a function is invalid, that is, a null object was provided to a parameter that cannot be null.
`arbb_error_scoping`	An API operation is attempted in an illegal scope, that is, an attempt is made to add a control flow operation without having a currently defined function.
`arbb_error_out_of_bounds`	An attempt is made to access a container out of bounds.
`arbb_error_arithmetic`	An arithmetic exception occurs: overflow, underflow, or division by zero.
`arbb_error_bad_alloc`	An attempt to allocate memory fails.
`arbb_error_uninitialized_access`	A variable is used before it is given a value.
`arbb_error_internal`	An unexpected internal error occurs.

In addition to returning an error code, these functions accept a final parameter of type `arbb_error_details_t*`. If a function returns `arbb_error_none` or a null pointer is passed to this parameter, the parameter is ignored. Otherwise, this pointer points to a newly allocated object of type `arbb_error_details_t` containing additional information about the error. Deallocate this object in your application by calling `arbb_free_error_details()`. Manipulate this object using the functions shown below:

```
/// @defgroup arbb_virtual_machine_errors Error handling
/// @{
/// An error code representing the result of a function.
typedef enum {
  arbb_error_none, ///< No error occurred.
  arbb_error_invalid_argument, ///< One or more parameters provided to a function was invalid, e.g. a
null object was provided to a parameter that must not be null.
  arbb_error_scoping, ///< An API operation was attempted in an illegal scope, e.g. attempting to add a
control flow operation without having a currently defined function.
  arbb_error_out_of_bounds, ///< An attempt to access a container out of bounds was made.
  arbb_error_arithmetic, ///< An arithmetic exception occurred - overflow, underflow, divide by 0.
  arbb_error_bad_alloc, ///< An attempt to allocate memory failed.
  arbb_error_uninitialized_access, ///< A variable was used before it was given a value.
  arbb_error_internal, ///< An unexpected internal error occurred.
} arbb_error_t;
/// Returns a character string with an informative error message
/// corresponding to @p error_details.
///
/// @return The descriptive error message. If a null error details object
/// is passed in, a null pointer is returned.
ARBB_VM_EXPORT
const char* arbb_get_error_message(arbb_error_details_t error_details);
/// Returns the error code corresponding to @p error_details
/// object.
///
/// @return The error code. If a null error details object is passed
/// in, ::arbb_error_none is returned.
ARBB_VM_EXPORT
arbb_error_t arbb_get_error_code(arbb_error_details_t error_details);
/// Frees the resources for @p error_details. If @p error_details is a
/// null object, this function has no effect.
ARBB_VM_EXPORT
void arbb_free_error_details(arbb_error_details_t error_details);/
/// @}
```

Contexts

Most Intel ArBB VM API functions take the first parameter of type `arbb_context_t`. In an environment with more than one registered VM implementation the context defines which implementation the VM function call is dispatched to. The ability to register alternate VM implementations is a currently unimplemented feature. Use the following function to retrieve a context for the default ArBB VM implementation:

```
/// @defgroup arbb_virtual_machine_contexts Contexts
/// @{
/// Retrieves the context for the default ArBB VM implementation. VM
/// operations may then be issued to the default ArBB VM implementation using
/// the returned context.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the reference was successfully decremented.
///  - ::arbb_error_invalid_argument if @p out_context is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_get_default_context(arbb_context_t* out_context,
                                      arbb_error_details_t* details);
/// @}
```

Reference Counting

The following types are reference counted and can be converted to an `arbb_refcountable_t` object (as shown in the following fragment of the header file):

- `arbb_function_t`
- `arbb_global_variable_t`

Below is the related fragment of the header file:

```
/// @defgroup arbb_virtual_machine_refcounting Reference counting
/// These functions allow conversion from specific object types that
/// are reference-counted to @ref arbb_refcountable_t instances.
/// @{
/// Converts the given @ref arbb_function_t to an @ref arbb_refcountable_t instance.
///
/// @return An @ref arbb_refcountable_t instance that can be used with arbb_acquire_ref() and
arbb_release_ref()
ARBB_VM_EXPORT
arbb_refcountable_t arbb_function_to_refcountable(arbb_function_t convertible);
/// Converts the given @ref arbb_global_variable_t to an @ref arbb_refcountable_t instance.
///
/// @return An @ref arbb_refcountable_t instance that can be used with arbb_acquire_ref() and
arbb_release_ref()
ARBB_VM_EXPORT
arbb_refcountable_t arbb_global_variable_to_refcountable(arbb_global_variable_t convertible);
/// Increments the reference count of @p refcountable.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the reference was successfully incremented.
///  - ::arbb_error_invalid_argument if @p refcountable is a null object.
ARBB_VM_EXPORT
arbb_error_t arbb_acquire_ref(arbb_refcountable_t refcountable,
                              arbb_error_details_t* details);
/// Decrements the reference count of @p refcountable. If the reference
/// count of the object drops to zero, its resources will be reclaimed.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the reference was successfully decremented.
///  - ::arbb_error_invalid_argument if @p refcountable is a null object.
ARBB_VM_EXPORT
arbb_error_t arbb_release_ref(arbb_refcountable_t refcountable,
                              arbb_error_details_t* details);
/// @}
```

String Objects

Some Intel ArBB VM API functions (notably `arbb_serialize_function()`) need to return text strings across the API boundary. The `arbb_string_t` type is used for this purpose. It enables accessing a string as a C character string and freeing the resources occupied by the string. Below is the related fragment of the header file:

```
/// @defgroup arbb_virtual_machine_strings String objects
/// @{
/// Retrieves the C string associated with @p string.
///
/// @return The C string held by @p string.
ARBB_VM_EXPORT
const char* arbb_get_c_string(arbb_string_t string);
/// Frees any resources associated with @p string.
ARBB_VM_EXPORT
void arbb_free_string(arbb_string_t string);
/// @}
```

Modules

2

Grammar

```
module          ::= module-item-seq

module-item-seq ::= module-item | module-item module-item-seq

module-item     ::= function-defn | function-decl | global-decl

                  | meta-definition
```

Description

The outermost level of nesting in the Intel® Array Building Blocks Virtual Machine Intermediate Representation is a module. A module may contain global declarations, function declarations and definitions, type definitions, and global attributes.

Identifiers

3

Grammar

```
identifier           ::= named-identifier | anonymous-identifier

named-identifier     ::= '_[a-zA-Z][_a-zA-Z0-9]*'

anonymous-identifier ::= '_[1-9][0-9]*' | '_0'
```

Description

Variables, functions, types, attributes, and labels are named with identifiers. Identifiers may be either named or anonymous. The names of identifiers begin with a prefix character that depends on the kind of the name:

- Named variables and functions begin with an underscore (_)
- Named types begin with a dollar sign ($)
- Named attributes begin with an exclamation mark (!)

In named variables, an alphabetical character follows the leading underscore, and a sequence of underscores or alphanumeric characters follows this alphabetical character.

Examples of named variables:

```
_foo

_foo_bar

_foo_123

_fooBarBaz
```

The leading underscore in named variables is used only in the textual serialized format and is missing in the in-memory representation of the Intermediate Representation.

Anonymous identifiers represent variables that have no explicit name. Like named identifiers, anonymous identifiers begin with an underscore. Unlike named identifiers, anonymous identifiers have only numerical characters following the underscore and no leading zeroes.

Examples of anonymous identifiers:

```
_0

_1

_4513
```

Types

4

Grammar

```
type                        ::= primitive-type | composite-type
primitive-type              ::= int-type | float-type | '$boolean' | size-type
int-type                    ::= '$i8' | '$i16' | '$i32' | '$i64' | '$u8' | '$u16'
                              | '$u32' | '$u64'
float-type                  ::= '$f32' | '$f64'
size-type                   ::= '$usize' | '$isize'
composite-type              ::= dense-type | nested-type
dense-type                  ::= 'dense' '<' composable-type dimensionality-specifier[opt]
'>'
dimensionality-specifier ::= ',' dimensionality
dimensionality              ::= '1' | '2' | '3'
nested-type                 ::= 'nested' '<' composable-type '>'
composable-type             ::= primitive-type
```

Description

The valid types that can be used to declare variables are primitive (scalar) types and composite types. The table below lists all the valid primitive types:

Primitive Type	**Description**
`$i8`	8-bit signed integer
`$i16`	16-bit signed integer
`$i32`	32-bit signed integer
`$i64`	64-bit signed integer
`$u8`	8-bit unsigned integer
`$u16`	16-bit unsigned integer
`$u32`	32-bit unsigned integer
`$u64`	64-bit unsigned integer
`$f32`	32-bit floating-point number
`$f64`	64-bit floating-point number

Primitive Type	Description
`$isize`	Signed array offset or size
`$usize`	Unsigned array index or size
`$boolean`	Boolean `true` or `false` value

Each composite type is based on a primitive type but represents a higher-level data structure, such as a dense array, a nested array, or an element binding. The table below lists valid composite types:

Composite Type	Description
`dense<T, N>`	Dense array of primitive type `T` with `N` dimensions (`N` = 1, 2, or 3)
`dense<T>`	Alias for dense<`T`, 1>
`nested<T>`	Nested array of primitive type `T`

Composable Types

Use composable types as element types of containers. The `composable-type` grammar rule defines valid composable types.

C Application Programming Interface

```
/// @defgroup arbb_virtual_machine_types Types
/// All variables in the VM have a corresponding type, represented by
/// a @ref arbb_type_t object.
///
/// @see @ref arbb_virtual_machine_functions
///
/// @{
/// The fixed set of scalar types provided.
typedef enum {
  arbb_i8, ///< 8-bit signed integer
  arbb_i16, ///< 16-bit signed integer
  arbb_i32, ///< 32-bit signed integer
  arbb_i64, ///< 64-bit signed integer
  arbb_u8, ///< 8-bit unsigned integer
  arbb_u16, ///< 16-bit unsigned integer
  arbb_u32, ///< 32-bit unsigned integer
  arbb_u64, ///< 64-bit unsigned integer
  arbb_f32, ///< 32-bit floating point number
  arbb_f64, ///< 64-bit floating point number
  arbb_boolean, ///< boolean true/false value
  arbb_usize, ///< unsigned array index or size
  arbb_isize, ///< signed array offset or size
} arbb_scalar_type_t;

/// Sets @p out_type to the @ref arbb_type_t object corresponding to a
/// scalar of type @p scalar_type.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_type is a null pointer.
///  - ::arbb_error_invalid_argument if @p scalar_type is an invalid value.
ARBB_VM_EXPORT
arbb_error_t arbb_get_scalar_type(arbb_context_t context,
                                  arbb_type_t* out_type,
                                  arbb_scalar_type_t scalar_type,
                                  arbb_error_details_t* details);

// Returns the number of bytes in the representation of @p type.
//
/// @return An error code depending on the result of the operation:
```

```
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_size is a null pointer.
///  - ::arbb_error_invalid_argument if @p type is a null object.
ARBB_VM_EXPORT
arbb_error_t arbb_sizeof_type(arbb_context_t context,
                              uint64_t* out_size,
                              arbb_type_t type,
                              arbb_error_details_t* details);

/// Sets @p out_type to the @ref arbb_type_t object corresponding to a
/// dense container with the specified element type and dimensionality.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_type is a null pointer.
///  - ::arbb_error_invalid_argument if @p dimensionality is not 1, 2, or 3.
///  - ::arbb_error_invalid_argument if @p element_type is a null object.
///  - ::arbb_error_invalid_argument if @p element_type is not a composable type.
ARBB_VM_EXPORT
arbb_error_t arbb_get_dense_type(arbb_context_t context,
                                 arbb_type_t* out_type,
                                 arbb_type_t element_type,
                                 unsigned int dimensionality,
                                 arbb_error_details_t* details);

/// Sets @p *out_type to the @ref arbb_type_t object corresponding to a nested
/// container with the specified element type.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_type is a null pointer.
///  - ::arbb_error_invalid_argument if @p element_type is a null object.
///  - ::arbb_error_invalid_argument if @p element_type is not a composable type.
arbb_error_t arbb_get_nested_type(arbb_context_t context,
                                  arbb_type_t* out_type,
                                  arbb_type_t element_type,
                                  arbb_error_details_t* details);

/// @}
```

Constants 5

Grammar

```
constant          ::= float-constant | int-constant | boolean-constant | size-constant
float-constant    ::= float-type '(' float-literal ')'
int-constant      ::= int-type '(' int-literal ')'
boolean-constant  ::= '$boolean' '(' boolean-literal ')'
size-constant     ::= size-type '(' int-literal ')'
float-literal     ::= hex-int-literal | dec-float-literal
dec-float-literal ::= '-?[0-9]*\.[0-9]*(e-?(0|[1-9][0-9]*))?'
int-literal       ::= dec-int-literal | hex-int-literal
dec-int-literal   ::= '-?[1-9][0-9]*' | '0'
hex-int-literal   ::= '0x[0-9a-fA-F][0-9a-fA-F]*'
boolean-literal   ::= 'true' | 'false'
```

Description

In some parts of code, such as parameters of operations, you can use scalar constants. Each scalar constant consists of a primitive type followed by a parenthesized literal.

Each integer constant consists of an integer type followed by an integer literal. An integer literal is either of the following:

- A decimal number that contains no leading zeroes and begins with an optional plus or minus sign.
- A hexadecimal number, which begins with the "0x" string.

Examples of integer constants:

```
$i32(15)

$u16(55)

$i16(-15)

$i8(0xff)
```

Each floating-point constant consists of a floating-point type followed by a floating-point literal. A floating-point literal is either of the following:

- A decimal number, which begins with an optional minus sign, always includes a decimal point, and ends with an optional exponent specifier.

  A decimal floating-point literal is converted to a floating-point value with correct rounding.
- A hexadecimal number, which begins with the "0x" string and specifies the bit pattern to be interpreted as a floating-point number of the appropriate size.

Examples of floating-point constants:

```
$f32(-5.5) // (1)

$f64(136.83) // (2)

$f32(0xC0B00000) // equivalent to line (1)

$f64(0x40611A8F5C28F5C3) // equivalent to line (2)
```

Each boolean constant consists of the type `$boolean` followed by a parenthesized boolean literal. A boolean literal is either `true` or `false`.

Examples of boolean constants:

```
$boolean(true)

$boolean(false)
```

C Application Programming Interface

```
/// @defgroup arbb_virtual_machine_constants Constants
/// Constants are variables of scalar type whose values never change.
/// @{
/// Creates a scalar constant of the given type. The data for the
/// constant will be copied in from the pointer parameter passed
/// to the @p data parameter.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_var is a null pointer.
///  - ::arbb_error_invalid_argument if @p type is a null object.
///  - ::arbb_error_invalid_argument if @p type is not a scalar type.
///  - ::arbb_error_invalid_argument if @p data is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_create_constant(arbb_context_t context,
                                  arbb_global_variable_t* out_var,
                                  arbb_type_t type,
                                  void* data,
                                  debug_data_description* debug_desc,
                                  arbb_error_details_t* details);
/// @}
```

6 Global Declarations

Grammar

```
global-decl ::= 'global' type identifier ';'
```

Description

A declaration of a variable that is available to all functions within a module is global. A global declaration consists of the keyword `global` followed by the variable type and an identifier naming the variable.

Examples:

```
global dense<$f32> _intensities;
```

```
global $f32 _weight;
```

C Application Programming Interface

In the C Application Programming Interface, a global container may be *bound* upon creation.

```
/// @defgroup virtual_machine_globals Globals
/// Global variables may be used in multiple functions, and may have
/// their values changed at run time.
/// @{
/// Create a new global variable.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_var is a null pointer.
///  - ::arbb_error_invalid_argument if @p type is a null object.
///  - ::arbb_error_invalid_argument if @p type is a function type.
///  - ::arbb_error_invalid_argument if @p binding is not a null object and does not match the type
being constructed
ARBB_VM_EXPORT
arbb_error_t arbb_create_global(arbb_context_t context,
                                arbb_global_variable_t* out_var,
                                arbb_type_t type,
                                const char *name,
                                arbb_binding_t binding,
                                debug_data_description* debug_desc,
                                arbb_error_details_t* details);
/// Sets @p out_var to the @ref arbb_variable_t object wrapped by the
/// given @ref arbb_global_variable_t object.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_var is a null pointer.
///  - ::arbb_error_invalid_argument if @p global_var is a null object.
ARBB_VM_EXPORT
arbb_error_t arbb_get_variable_from_global(arbb_context_t context,
                                           arbb_variable_t* out_var,
                                           arbb_global_variable_t global_var,
                                           arbb_error_details_t* details);
/// @}
```

Functions

7

Grammar

```
function-decl         ::= function-declarator ';'
function-defn         ::= function-declarator attributes[opt]
                          function-body
function-declarator   ::= 'function' identifier '(' param-seq[opt] ')'
function-body         ::= '{' stmt-seq[opt] '}'
param-seq             ::= param | param ',' param-seq
param                 ::= input-param | output-param
input-param           ::= 'in' type identifier
output-param          ::= 'out' type identifier'
```

Description

A function consists of a name, a sequence of input and output variables, and a body of code. A function declaration consists of the `function` keyword followed by an identifier naming the function. List function parameters in parentheses as a comma-separated, possibly empty, sequence of variable declarations.

To declare an input or output parameter, use the `in` or `out` keyword, respectively, followed by a type and an identifier. Declare all output parameters before all input parameters.

Example of a function declaration:

```
function _add_vectors(out dense<$f32> _result,
in dense<$f32> _left,
in dense<$f32> _right);
```

A function definition consists of a function declarator followed by a body. The body is a list of zero or more statements enclosed in braces (`{` and `}`).

Example of a function definition:

```
function _add_vectors(out dense<$f32> _result,
                      in dense<$f32> _left,
                      in dense<$f32> _right)
{
    _result = add<dense<$f32>>(_left, _right);
}
```

C Application Programming Interface

The process of building a function, starting with `arbb_begin_function` and ending with `arbb_end_function`, is not thread safe. Build functions in the main thread. However, you may compile and/or execute them in any thread.

```
///@defgroup arbb_virtual_machine_functions Functions
/// @{
/// Adds a new function type with the specified prototype, which consists of the
/// number of input and output parameters and their corresponding types.
/// input_types and output_types should be arrays with num_inputs and
/// num_outputs entries respectively.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_type is a null pointer.
///  - ::arbb_error_invalid_argument if @p num_inputs is greater than 0 and input_types is a null
pointer.
///  - ::arbb_error_invalid_argument if @p num_outputs is greater than 0 and output_types is a null
pointer.
///  - ::arbb_error_invalid_argument if any entry in @p input_types is a null object.
///  - ::arbb_error_invalid_argument if any entry in @p output_types is a null object.
ARBB_VM_EXPORT
arbb_error_t arbb_get_function_type(arbb_context_t context,
                                    arbb_type_t* out_type,
                                    unsigned int num_outputs,
                                    const arbb_type_t* output_types,
                                    unsigned int num_inputs,
                                    const arbb_type_t* input_types,
                                    arbb_error_details_t* details);
/// Converts the provided @p function_type to a function type with the
/// same signature, but where the outputs are aliased to a subset of
/// the inputs.
///
/// @param context The context in which to perform the operation.
/// @param out_type The location in which to store the new type.
/// @param function_type The function type on which the new type will
/// be based.
/// @param parameter_aliases An array of N integers, where N is the
/// number of outputs, specifying the indices of the inputs aliased
/// with each corresponding output.
/// @param details If non-null, error details will be placed here if
/// an error occurs.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_type is a null pointer.
///  - ::arbb_error_invalid_argument if @p function_type is a null object.
///  - ::arbb_error_invalid_argument if @p parameter_aliases is a null pointer.
///  - ::arbb_error_invalid_argument if any of the entries in @p parameter_aliases are greater than or
equal to the number of inputs in @p function_type.
ARBB_VM_EXPORT
arbb_error_t arbb_get_function_type_parameter_alias(arbb_context_t context,
                                                    arbb_type_t* out_type,
                                                    arbb_type_t function_type,
                                                    const unsigned int* parameter_aliases,
                                                    arbb_error_details_t* details);
/// Creates a new function of the specified function type and begins
/// defining it. If @p output_parameter_names or @p input_parameter_names
/// is non-null, this function expects it to point to an array of C strings
/// that are used to name the function parameters.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_function is a null pointer.
///  - ::arbb_error_invalid_argument if @p function_type is not a valid function type.
///  - ::arbb_error_invalid_argument if @p name is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_begin_function(arbb_context_t context,
                                 arbb_function_t* out_function,
                                 arbb_type_t function_type,
                                 const char* function_name,
                                 const char** output_parameter_names,
                                 arbb_error_details_t* details);
/// Aborts the definition of the specified function.
///
/// This will free any resources, such as local variables, associated with
/// the function. The function will no longer be a valid object.
///
/// This function is intended to be used when an application error occurs
/// during the definition of a function.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if function is a null object.
///  - ::arbb_error_scoping if the given function is not being defined.
ARBB_VM_EXPORT
arbb_error_t arbb_abort_function(arbb_function_t function,
                                 arbb_error_details_t* details);
/// Finishes the definition of the specified function.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if function is a null object.
///  - ::arbb_error_scoping if the given function is not being defined.
ARBB_VM_EXPORT
arbb_error_t arbb_end_function(arbb_function_t function,
                               arbb_error_details_t* details);
/// Sets @p *out_var to the variable corresponding to the input or
/// output of the given function at the position given by @p index.
/// If @p get_output is non-zero, an output is returned, otherwise an
/// input is returned.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p function is a null object.
///  - ::arbb_error_invalid_argument if @p out_var is a null pointer.
///  - ::arbb_error_invalid_argument if @p index is greater than or equal
///    to the number of inputs or outputs of the function, depending on the
///    value of @p get_output.
///  - ::arbb_error_scoping if @p function is not being defined.
ARBB_VM_EXPORT
arbb_error_t arbb_get_parameter(arbb_function_t function,
                                arbb_variable_t* out_var,
                                int get_output,
                                unsigned int index,
                                arbb_error_details_t* details);
/// Serializes the intermediate representation of @p function.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p function is a null object.
///  - ::arbb_error_invalid_argument if @p out_text is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_serialize_function(arbb_function_t function,
                                     arbb_string_t* out_text,
                                     arbb_error_details_t* details);
/// @}
```

Local Declarations

Grammar

```
decl-stmt ::= 'local' type identifier attributes[opt] ';'
```

Description

To declare variables locally, use a local declaration statement, which consists of the keyword `local` followed by the variable type and identifier.

Example:

```
local dense<$f32> _vec1;
```

Declaring a local variable twice in the same function results in an error. If a declaration of a local variable appears in a function, use this local variable as a statement parameter (input or output) only after the declaration and only in the scope of the declaration.

If you are using a serialized format, you do not need to declare variables explicitly. If a function does not contain a declaration of a variable, the variable is declared implicitly. The rules for implicit declaring a local variable are as follows:

- Local variables are recognized by their identifiers that appear as input or output parameters of statements.
- Every local variable that appears as an input parameter of a statement within an evaluation unit must also appear as an output parameter of a statement within the same evaluation unit.
- The result type of statements in which a local variable appears as an output parameter determines the type of the variable. The result type must be the same in all such statements.
- Statements in which a variable is used as an input parameter and statements in which the same variable is used as an output parameter may appear in any order. For example, a variable may appear as an input parameter in an evaluation unit before it appears for the first time as an output parameter.
- Local variables declared implicitly are considered declared at the beginning of the function in the outermost scope within the corresponding function.

C Application Programming Interface

```
/// @defgroup arbb_virtual_machine_locals Local variables
/// @{
/// Creates a new local variable within the given function.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p function is a null object.
///  - ::arbb_error_invalid_argument if @p out_var is a null pointer.
///  - ::arbb_error_invalid_argument if @p type is a null object.
///  - ::arbb_error_invalid_argument if @p type is a function type.
///  - ::arbb_error_scoping if @p function is not being defined.
ARBB_VM_EXPORT
arbb_error_t arbb_create_local(arbb_function_t function,
                               arbb_variable_t* out_var,
                               arbb_type_t type,
                               const char* name,
                               arbb_error_details_t* details);

/// Sets @p out_name to the name of the variable @p var.
///
/// @return An error code depending on the result of the operation:
/// - ::arbb_error_none if the operation succeeded.
/// - ::arbb_error_invalid_argument if @p var is a null pointer.
/// - ::arbb_error_invalid_argument if @p out_name is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_get_variable_name(arbb_variable_t var,
                                    arbb_string_t* out_name,
                                    arbb_error_details_t* details);

/// @}
```

See Also

Overview
Intermediate Representation Definition Interfaces

8 Statements

Grammar

```
stmt-seq ::= stmt | stmt stmt-seq

stmt     ::= simple-stmt  | facility-stmt | reduce-stmt | cflow-stmt

           | decl-stmt | member-stmt | functor-stmt

           | ... //
```

Description

Statements appear within function definitions and may be nested inside control-flow and other hierarchical statements.

C Application Programming Interface

The `arbb_opcode_t` enumeration values correspond to those used in the textual Intermediate Representation that are prefixed by `arbb_op_`.

```
/// @defgroup arbb_virtual_machine_operations Operations
/// @{
/// The set of operations that can be performed using arbb_op() or
/// arbb_op_dynamic().
typedef enum {
  // element-wise operations
  arbb_op_abs,
  arbb_op_acos,
  arbb_op_asin,
  arbb_op_atan,
  arbb_op_ceil,
  arbb_op_copy,
  arbb_op_cos,
  arbb_op_cosh,
  arbb_op_exp,
  arbb_op_exp10,
  arbb_op_floor,
  arbb_op_ln,
  arbb_op_log10,
  arbb_op_log_not,
  arbb_op_bit_not,
  arbb_op_rcp,
  arbb_op_round,
  arbb_op_rsqrt,
  arbb_op_sin,
  arbb_op_sinh,
  arbb_op_sqrt,
  arbb_op_tan,
  arbb_op_tanh,
  arbb_op_neg,
  arbb_op_add,
  arbb_op_bit_and,
  arbb_op_atan2,
  arbb_op_compare,
  arbb_op_div,
  arbb_op_equal,
  arbb_op_geq,
  arbb_op_greater,
  arbb_op_bit_or,
  arbb_op_leq,
  arbb_op_less,
  arbb_op_log_and,
  arbb_op_log_or,
  arbb_op_lsh,
  arbb_op_max,
  arbb_op_min,
  arbb_op_mod,
  arbb_op_mul,
  arbb_op_neq,
  arbb_op_pow,
  arbb_op_rsh,
  arbb_op_sub,
  arbb_op_bit_xor,
  arbb_op_select,
  // reorder operations
  arbb_op_gather,
  arbb_op_scatter,
  arbb_op_pack,
  arbb_op_unpack,
  arbb_op_shuffle,
  arbb_op_unshuffle,
  arbb_op_repeat,
  arbb_op_distribute,
  arbb_op_repeat_row,
  arbb_op_repeat_col,
  arbb_op_repeat_page,
  arbb_op_transpose,
  arbb_op_swap_col,
  arbb_op_swap_row,
  arbb_op_swap_page,
  arbb_op_shift_constant,
  arbb_op_shift_clamp,
  arbb_op_shift_constant_reverse,
  arbb_op_shift_clamp_reverse,
  arbb_op_rotate,
  arbb_op_rotate_reverse,
  arbb_op_reverse,
  // facility operations
  arbb_op_length,
  arbb_op_apply_nesting,
  arbb_op_get_nesting,
  arbb_op_cat,
  arbb_op_cast,
  arbb_op_extract,
  arbb_op_split,
  arbb_op_unsplit,
  arbb_op_index,
  arbb_op_mask,
  arbb_op_copy_nesting,
  arbb_op_flatten,
  arbb_op_const_vector,
  arbb_op_sort,
  arbb_op_sort_rank,
  arbb_op_replace,
  arbb_op_set_regular_nesting,
  arbb_op_replace_row,
  arbb_op_replace_col,
  arbb_op_replace_page,
  arbb_op_get_nrows,
  arbb_op_get_ncols,
  arbb_op_get_npages,
  arbb_op_extract_row,
  arbb_op_extract_col,
  arbb_op_extract_page,
  arbb_op_section,
  arbb_op_segment,
  arbb_op_replace_segment,
  arbb_op_alloc,
  arbb_op_replace_element,
  arbb_op_get_elt_coord,
  arbb_op_bitwise_cast,
  arbb_op_get_neighbor,
  arbb_op_expect_size,
  // collective operations
  arbb_op_add_reduce,
  arbb_op_mul_reduce,
  arbb_op_max_reduce,
  arbb_op_max_reduce_loc,
  arbb_op_min_reduce,
  arbb_op_min_reduce_loc,
  arbb_op_and_reduce,
  arbb_op_ior_reduce,
  arbb_op_xor_reduce,
  arbb_op_add_scan,
  arbb_op_mul_scan,
  arbb_op_max_scan,
  arbb_op_min_scan,
  arbb_op_and_scan,
  arbb_op_ior_scan,
  arbb_op_xor_scan,
  arbb_op_add_merge,
  arbb_op_add_merge_scalar
} arbb_opcode_t;
/// Adds a new instruction to the given function or executes the
/// provided operation. If @p function is a null pointer, all inputs
/// and outputs must be globals and the operation executes
/// immediately. The number of inputs and outputs of the given opcode
/// must be static. The arguments passed to the @p outputs and @p inputs
/// parameters must be arrays of length matching the operation
/// arity. If @p attributes is not a null pointer, the attribute map pointed to
/// by @p attributes is attached to the operation. ArBB assumes ownership
/// of the attribute map and * @p attributes is set to a null object.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p outputs is a null pointer.
///  - ::arbb_error_invalid_argument if @p inputs is a null pointer.
///  - ::arbb_error_invalid_argument if any of the entries in @p outputs are null objects.
///  - ::arbb_error_invalid_argument if any of the entries in @p inputs are null objects.
///  - ::arbb_error_invalid_argument if @p opcode does not have a static number of parameters.
///  - ::arbb_error_invalid_argument if @p attributes is not a null pointer but
///    points to a null object
///  - ::arbb_error_scoping if @p function is not currently being defined.
ARBB_VM_EXPORT
arbb_error_t arbb_op(arbb_function_t function,
                     arbb_opcode_t opcode,
                     const arbb_variable_t* outputs,
                     const arbb_variable_t* inputs,
                     void* debug_data_ptrs[],
                     arbb_attribute_map_t* attributes,
                     arbb_error_details_t* details);
/// Adds a new instruction to the given function or executes the
/// provided operation. If @p function is a null pointer, all inputs
/// and outputs must be globals and the operation executes
/// immediately. The provided opcode must have a dynamic number of
/// inputs and/or outputs. The parameters passed to the @p outputs and
/// @p inputs parameters must be arrays of length @p num_outputs and
/// @p num_inputs, respectively. If @p attributes is not a null pointer, the
/// attribute map pointed to by @p attributes is attached to the operation
/// and ArBB assumes ownership of the attribute map and * @p attributes is
/// set to a null object.
/
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if outputs is a null pointer.
///  - ::arbb_error_invalid_argument if inputs is a null pointer.
///  - ::arbb_error_invalid_argument if any of the entries in outputs are null objects.
///  - ::arbb_error_invalid_argument if any of the entries in inputs are null objects.
///  - ::arbb_error_invalid_argument if the opcode has a static number of parameters.
///  - ::arbb_error_invalid_argument if either the inputs or outputs of the
///    opcode has a static size, and the size provided does not match.
///  - ::arbb_error_invalid_argument if @p attributes is not a null pointer but
///    points to a null object
///  - ::arbb_error_scoping if the given function is not currently being defined.
ARBB_VM_EXPORT
arbb_error_t arbb_op_dynamic(arbb_function_t function,
                             arbb_opcode_t opcode,
                             unsigned int num_outputs,
                             const arbb_variable_t* outputs,
                             unsigned int num_inputs,
                             const arbb_variable_t* inputs,
                             void* debug_data_ptrs[],
                             arbb_attribute_map_t* attributes,
                             arbb_error_details_t* details);
/// @}
```

Function Execution Semantics

To execute virtual machine functions, use the `arbb_execute` function or a function call statement.

The semantics of the execution are as follows:

- Temporary variables are allocated for all input and output parameters.
- All input parameters are copied to the corresponding temporary variables.
- Constant input parameters are copied into variables.
- The appropriate temporary variables are passed to the function as input and output parameters.
- After completing the execution, the output is copied from the corresponding temporary variables to the actual output parameters.

If two output parameters are bound to host data so that their bindings are aliases, the behavior of this function is undefined. Input parameters can be aliases of the output parameters according to the rules above.

See Also

Execution and Compilation
Function Call Statements
Container Bindings

Basic Block Statements

Grammar

```
simple-stmt    ::= arg '=' simple-stmt-id '<' type '>' '(' param-seq ')'
attributes[opt] ';'

simple-stmt-id ::= ewise-stmt-id | scan-stmt-id | reorder-stmt-id
```

Description

Basic block statements are statements unrelated to control flow. Most basic block statements are simple statements. A simple statement starts with a destination identifier, which is followed by an assignment symbol, a statement identifier, a statement type, and a parenthesized list of parameters (`param-seq`).

Example of a simple statement:

```
_vec1 = add<dense<$f32>>(_vec2, _vec3);
```

The first parameter of a simple statement represents the type of the statement. It is usually the type of the output parameter of that statement, but for some statements (such as those performing logical comparison operations) it may relate to the type of the input parameters. All the rest of parameters are input.

Notational Conventions for Statements

Syntax of specific statements in the Intel® Array Building Blocks (Intel® ArBB) Virtual Machine (VM) Application Programming Interface (API) is represented using placeholders for input and output types. Values of the placeholders are as follows:

`any`	Any primitive type.
`val`	Any primitive type except for `$boolean`.
`fp`	Any primitive floating-point type.
`int`	Any primitive integer type.
`size`	Any primitive signed or unsigned size type.
`C<T>`	Any dense or nested container of element type T.

See Also

Notational Conventions

Elementwise Statements

Grammar

```
ewise-stmt-id ::  'abs' | 'acos' | 'asin' | 'atan' | 'ceil'
                | 'copy' | 'cos' | 'cosh' | 'exp' | 'exp10'
                | 'floor' | 'ln' | 'log10' | 'log_not'
                | 'bit_not' | 'rcp' | 'round' | 'rsqrt'
                | 'sin' | 'sinh' | 'sqrt' | 'tan' | 'tanh'
                | 'neg' | 'add' | 'bit_and' | 'atan2'
                | 'compare' | 'div' | 'equal'
                | 'geq' | 'greater' | 'bit_or'
                | 'leq' | 'less' | 'log_and' | 'log_or' |
                | 'lsh' | 'max' | 'min' | 'mod' | 'mul'
                | 'neq' | 'pow' | 'rsh' | 'sub'
                | 'bit_xor' | 'select'
```

Description

Elementwise statements are executed on scalars or on each element in dense or nested containers. The result of an elementwise statement is a scalar if all input parameters are scalars, and a container if any input parameter is a container. If a parameter of an elementwise statement is a container, use containers of exactly the same shape for all input and output parameters.

The behavior of an elementwise statement with at least one container input is the same as of a `map` statement where the function called is the scalar version of the elementwise statement.

The following table lists available elementwise statements.

In the **Syntax** column, names of input parameters begin with the letter `s`.

Contents of the **Type** column indicate the type to use in the type field of the statement. An explicit primitive type indicates that this type or a dense container of this type should be used. Placeholders stand for the types as specified in the Notational Conventions for Statements.

The **Tolerance** column indicates the maximum floating-point error in the result of floating-point operations, in Units of Least Precision (ULPs). One ULP is the value the least-significant bit represents if it is one. If the values of tolerance are different for a regular Intel ArBB and for Intel ArBB on Intel® Many Integrated Core (Intel® MIC) Architecture, both values are provided.

Name	Syntax	Type	Description	Tolerance (ULPs)
Unary Statements				
`abs`	`fp = abs(fp s0);`	`fp`	Computes the absolute value of *s0*	0
`acos`	`fp = acos(fp s0);`	`fp`	Computes the arccosine of *s0*	4
`asin`	`fp = asin(fp s0);`	`fp`	Computes the arcsine of *s0*	4
`atan`	`fp = atan(fp s0);`	`fp`	Computes the arctangent of *s0*	4
`ceil`	`fp = ceil(fp s0);`	`fp`	Rounds *s0* to the nearest integer towards the positive infinity	0
`copy`	`any = copy(any s0);`	`any`	Copies *s0* to the output	0
`cos`	`fp = cos(fp s0);`	`fp`	Computes the cosine of *s0*	4
`cosh`	`fp = cosh(fp s0);`	`fp`	Computes the hyperbolic cosine of *s0*	4
`exp`	`fp = exp(fp s0);`	`fp`	Computes the base-e exponent of *s0*	4
`exp10`	`fp = exp10(fp s0);`	`fp`	Computes the base-10 exponent of *s0*	4
`floor`	`fp = floor(fp s0);`	`fp`	Rounds *s0* to the nearest integer towards the negative infinity	0
`ln`	`fp = ln(fp s0);`	`fp`	Computes the base-e logarithm of *s0*	4
`log10`	`fp = log10(fp s0);`	`fp`	Computes the base-10 logarithm of *s0*	4
`log_not`	`$boolean = log_not($boolean s0);`	`$boolean`	Computes the logical negation of *s0*	
`bit_not`	`int = bit_not(int s0);`	`int`	Computes the bitwise negation of *s0*	
`rcp`	`fp = rsp(fp s0);`	`fp`	Computes the reciprocal of *s0*	0.5 4 on Intel MIC Architecture

Name	Syntax	Type	Description	Tolerance (ULPs)
`round`	`fp = round(fp s0);`	`fp`	Rounds *s0* to the nearest integer	0
`rsqrt`	`fp = rsqrt(fp s0);`	`fp`	Computes the reciprocal square root of *s0*	1 4 on Intel MIC Architecture
`sin`	`fp = sin(fp s0);`	`fp`	Computes the sine of *s0*	4
`sinh`	`fp = sinh(fp s0);`	`fp`	Computes the hyperbolic sine of *s0*	4
`sqrt`	`fp = sqrt(fp s0);`	`fp`	Computes the square root of *s0*	4
`tan`	`fp = tan(fp s0);`	`fp`	Computes the tangent of *s0*	4
`tanh`	`fp = tanh(fp s0);`	`fp`	Computes the hyperbolic tangent of *s0*	4
`neg`	`fp = neg(fp s0);`	`val`	Computes the numerical negation of *s0*	0
Binary Statements				
`add`	`val = add(val s0, val s1);`	`val`	Adds *s0* to *s1*	0.5
`bit_and`	`int = bit_add(int s0, int s1);`	`int`	Computes the bitwise *and* of *s0* and *s1*	
`atan2`	`fp = atan2(fp s0, fp s1);`	`fp`	Computes *atan2*(*s0*/*s1*) as defined by C89.	4
`compare`	`$isize = compare(any s0, any s1);`	`any`	Compares *s0* and *s1*, returning {-1, 0, 1} depending on whether s0 is {smaller, equal, greater} than s1	
`div`	`val = div(val s0, val s1);`	`val`	Divides *s0* by *s1*	0.5 4 on Intel MIC Architecture
`equal`	`$boolean = equal(any s0, any s1);`	`any`	Computes whether *s0* equals s1	
`geq`	`$boolean = geq(any s0, any s1);`	`any`	Computes whether *s0* is greater than or equal to *s1*	
`greater`	`$boolean = greater(any s0, any s1);`	`any`	Computes whether *s0* is strictly greater than *s1*	
`bit_or`	`int = bit_or(int s0, int s1);`	`int`	Computes the bitwise *or* of *s0* and *s1*	
`leq`	`$boolean = leq(any s0, any s1);`	`any`	Computes whether *s0* is less than or equal to *s1*	

Name	Syntax	Type	Description	Tolerance (ULPs)
`less`	`$boolean = less(any s0, anys1);`	`any`	Computes whether *s0* is strictly less than *s1*	
`log_and`	`$boolean = log_not($boolean s0, $boolean s1);`	`$boolean`	Computes the logical *and* of *s0* and *s1*	
`log_or`	`$boolean = log_or($boolean s0, $boolean s1);`	`$boolean`	Computes the logical *or* of *s0* and *s1*	
`lsh`	`int = lsh(int s0, $u8 s1);`	`int`	Computes the left shift of *s0* by *s1* bits. The result is unspecified if *s1* ≥ (bit width of *s0*).	
`max`	`any = max(any s0, any s1);`	`any`	Computes the maximum of *s0* and *s1*	
`min`	`any = min(any s0, any s1);`	`any`	Computes the minimum of *s0* and *s1*	
`mod`	`int = mod(int s0, int s1);`	`int`	Computes the remainder when *s0* is divided by *s1*. The equality `mod(`*s0*, *s1*`)` = *s0* - (*s0* / *s1*) * *s1* rounded toward 0 holds.	
`mul`	`val = mul(val s0, val s1);`	`val`	Multiplies *s0* and *s1*	0.5
`neq`	`$boolean = neq(any s0, any s1);`	`any`	Computes whether *s0* does not equal *s1*	
`pow`	`fp = pow(fp s0, fp s1);`	`fp`	Raises *s0* to the power of *s1*	4
`rsh`	`int = rsh(int s0, $u8 s1);`	`int`	Computes the right shift of *s0* by *s1* bits. If the type of *s0* is signed, performs sign extension. Otherwise, fills vacated bits with zeroes. The result is unspecified if *s1* ≥ 0 (bit width of *s0*).	
`sub`	`val = sub(val s0, val s1);`	`val`	Subtracts *s1* from *s0*	0.5
`bit_xor`	`int = bit_xor(int s0, int s1);`	`int`	Computes the bitwise exclusive *or* of *s0* and *s1*	
Ternary Statements				
`select`	`any = select($boolean s0, any s1, any s1);`	`$boolean`	Returns *s1* if *s0* is true, *s1* otherwise	

See Also

Function Call Statements

Function Call Statements

Grammar

```
functor-stmt ::= 'call' identifier '(' param-seq[opt] ')' ';'

               | 'map' identifier '(' param-seq[opt] ')' ';'
```

Description

The `call` and `map` statements enable execution of functions. A `call` statement performs a single call to a function, whereas a `map` statement executes a function in parallel over multiple instances of data passed in. The initial keyword in these statements indicates the type of call (`call` or `map`), which is followed by a function identifier with a list of parameters (`param-seq`).

For `call` statements, types of input and output parameters exactly match the function signature. For `map` statements, the output parameters are containers (of matching dimensionality, size, and container type) of the corresponding parameter types defined in the function signature, and input parameters can either be such containers or exactly match the corresponding parameter types defined in the function signature.

CAUTION Reading from or writing to global Intel ArBB variables inside a `call` or `map` statement is not supported and will result in a run-time error.

The following code fragment illustrates the use of the `call` statement:

```
function callee(out $f32 a, in $f32 b, in $f32 c);
function caller(out $f32 o, in $f32 i1, in $f32 i2) {
call callee(o, i1, i2);
}
```

The following code fragment illustrates the use of the `map` statement:

```
function callee(out $f32 a, in $f32 b, in $f32 c);
function caller(out dense<$f32> o, in dense<$f32> i1, in $f32 i2) {
map callee(o, i1, i2);
}
```

C Application Programming Interface

```
/// @defgroup arbb_virtual_machine_calls Function calls
/// @{
/// The set of ways in which functions may be called using arbb_call_op().
///
/// @see arbb_call_op()
typedef enum {
  arbb_op_call, ///< Perform a plain call operation.
  arbb_op_map //< Perform a map operation.
} arbb_call_opcode_t;
/// Adds a new calling instruction to the given function. The number and
/// types of inputs and outputs passed in must match the function
/// signature. For plain call operations, the function signature must
/// match exactly. For map operations, output types must be containers
/// of the corresponding function signature types, and input types must
/// either match exactly or be containers of the corresponding function
/// signature types.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p caller is a null object.
///  - ::arbb_error_invalid_argument if @p callee is a null object.
///  - ::arbb_error_invalid_argument if @p outputs is a null pointer.
///  - ::arbb_error_invalid_argument if @p inputs is a null pointer.
///  - ::arbb_error_invalid_argument if any of the entries in @p outputs are null objects.
///  - ::arbb_error_invalid_argument if any of the entries in @p inputs are null objects.
///  - ::arbb_error_invalid_argument if any of the variables provided in @p inputs
///      or @p outputs do not match the function's signature appropriately.
///  - ::arbb_error_scoping if @p caller is not currently being defined.
///  - ::arbb_error_scoping if @p callee has not been defined already.
ARBB_VM_EXPORT
arbb_error_t arbb_call_op(arbb_function_t caller,
                          arbb_call_opcode_t opcode,
                          arbb_function_t callee,
                          const arbb_variable_t* outputs,
                          const arbb_variable_t* inputs,
                          arbb_error_details_t* details);
/// @}
```

See Also

Function Execution Semantics

Functions

Reordering Statements

Grammar

```
reorder-stmt-id ::= 'gather' | 'scatter' | 'pack' | 'unpack'
                  | 'shuffle' | 'unshuffle' | 'repeat' | 'distribute'
                  | 'repeat_row' | 'repeat_col' | 'repeat_page' | 'transpose'
                  | 'swap_row' | 'swap_col' | 'swap_page'
                  | 'shift_constant' | 'shift_clamp'  | 'shift_constant_reverse'
                  | 'shift_clamp_reverse' | 'rotate' | 'rotate_reverse' | 'reverse'
```

Description

Reordering statements rearrange one or more containers in some manner, for example, by creating a container of multiple copies of an existing container or by scattering data into a container.

gather

Copies data out of a container using a sequence of indices.

Syntax

dense<any, 1> = gather(dense<any, 1> *in*, dense<$usize, 1> *cols*, any);

dense<any, 2> = gather(dense<any, 2> *in*, dense<$usize, 2> *rows*, dense<$usize, 2> *cols*, any);

dense<any, 3> = gather(dense<any, 3> *in*, dense<$usize, 3> *pages*, dense<$usize, 3> *rows*, dense<$usize, 3> *cols*, any);

Description

A `gather` statement copies data out of a container using a sequence of indices. A `scatter` statement performs the inverse operation.

A `gather` statement accepts a dense container of data, a dense container of indices for each dimension of the data, and a default value. The default value is used when an out-of-bounds access is performed. Provide the default value that matches the element type of the data container.

The associated type of a `gather` statement is the type of the result.

Example

```
local dense<$f32, 1> _data;
local dense<$f32, 1> _result;
local dense<$isize, 1> _indices;
_result = gather<dense<$f32, 1>>(_data, _indices, $f32(0.0));
```

See Also

Notational Conventions for Statements

scatter

Copies data out of a container into a new container with a given set of indices.

Syntax

`dense<any, 1> = scatter(dense<any, 1> in, dense<$usize, 1> cols, dense<any, 1> out);`

`dense<any, 2> = scatter(dense<any, 2> in, dense<$usize, 2> rows, dense<$usize, 2> cols, dense<any, 2> out);`

`dense<any, 3> = scatter(dense<any, 3> in, dense<$usize, 3> pages, dense<$usize, 3> rows, dense<$usize, 3> cols, dense<any, 3> out);`

Description

A `scatter` statement copies data out of a container into a new container with a given set of indices. A `gather` statement performs the inverse operation.

A `scatter` statement accepts a dense container of data, a dense container of destination indices for each dimension of the data, and a dense container to scatter the data into.

The associated type of a `scatter` statement is the type of the resulting container.

The behavior of the statement is undefined if the set of indices contains a duplicate index.

Example

```
local dense<$f32, 1> _data;
local dense<$f32, 1> _destination;
local dense<$f32, 1> _result;
local dense<$isize, 1> _indices;
_result = scatter<dense<$f32, 1>>(_data, _indices, _destination);
```

See Also

Notational Conventions for Statements

pack

Compacts data in a container using a mask container.

Syntax

`C<any> = pack(C<any>, C<$boolean>);`

Description

A `pack` statement compacts data in a container using a mask container. An `unpack` statement performs the inverse operation.

A `pack` statement accepts a data container and a mask container and returns a container of the same type as the data container. The associated type of a `pack` statement is the type of the resulting container. The mask container in a `pack` statement has a `$boolean` element type. Provide the data and mask containers for the `pack` statement that match in size and shape.

A `pack` statement generates the resulting container by including, in order, every element from the data container that has a `true` value in the corresponding location of the mask container. Values that have a corresponding `false` value in the mask container do not appear in the resulting container. Hence the size of the resulting container equals the number of `true` elements in the mask container.

Example

```
local dense<$f32> _data;
local dense<$boolean> _mask;
local dense<$f32> _packed;
_packed = pack<dense<$f32>>(_data, _mask);
```

See Also

Notational Conventions for Statements

unpack

Expands data in a container using a mask container.

Syntax

```
C<any> = unpack(C<any>, C<$boolean>, any);
```

Description

An `unpack` statement expands data in a container using a mask container. A `pack` statement performs the inverse operation.

An `unpack` statement accepts a data container, a mask container, and a default value and returns a container of the same type as the data container. The associated type of an `unpack` statement is the type of the resulting container. The mask container has a `$boolean` element type. Provide the default value of the same type as the element type of the data container. If the number of `true` values in the mask container does not equal the number of elements in the data container, the behavior of the statement is undefined.

The unpack statement generates a resulting container with the same size as the mask container. For every `false` value in the mask container, the default value is placed in the resulting container. The statement draws other elements, corresponding to `true` values in the mask container, from the data container in order.

Example

```
local dense<$f32> _data;
local dense<$boolean> _mask;
local dense<$f32> _packed;
_packed = unpack<dense<$f32>>(_data, _mask, $f32(0.0));
```

See Also

Notational Conventions for Statements

shuffle

Interleaves sequences from two containers.

Syntax

`C<any> = shuffle(C<any> in1, C<any> in2, $usize grain, $usize level);`

Description

A `shuffle` statement interleaves sequences from two containers. An `unshuffle` statement performs the inverse operation.

A `shuffle` statement accepts two data containers, a sequence size, and a nesting level and produces a resulting container. Provide the data containers for `shuffle` that match in type and size. The resulting container has the same type as the data containers. The associated type of a shuffle statement is that of the resulting container. The statement alternately draws into the resulting container a number of elements/ segments from each data container, starting with the first data container. The sequence size determines the number of elements drawn between alternating draws.

The level information is relevant only to nested containers. Valid values of the *level* parameter mean the following:

- If *level* = 0, the statement operates individual elements
- If *level* = 1, the statement operates segments

Example

```
local dense<$f32> _data1;
local dense<$f32> _data2;
local dense<$f32> _zipped;
_zipped = shuffle<dense<$f32>>(_data1, _data2, $usize(1), $usize(0));
```

See Also

Notational Conventions for Statements

unshuffle

Performs an inverse operation of interleaving sequences from two containers.

Syntax

`C<any> = unshuffle(C<any> in, $usize grain, $usize level);`

Description

A `shuffle` statement interleaves sequences from two containers. An `unshuffle` statement performs the inverse operation.

An `unshuffle` statement accepts a single data container *in*, a sequence size *grain*, and a nesting level *level* and produces a resulting container. The resulting container has the same type as the data container. The associated type of an `unshuffle` statement is that of the resulting container.

The unshuffle operation draws a number of elements/segments from the input container and appends those elements/segments to two alternating containers. The sequence size determines this number. The resulting container is a concatenation of the two containers.

The level information is relevant only to nested containers. Valid values of the *level* parameter mean the following:

- If *level* = 0, the statement operates individual elements
- If *level* = 1, the statement operates segments

Example

```
local dense<$f32> _zipped;
local dense<$f32> _sequential;
_sequential = unshuffle<dense<$f32>>(_zipped, $usize(1), $usize(0));
```

See Also

Notational Conventions for Statements

repeat

Repeats each element or segment of a container the same number of times.

Syntax

`dense<any> = repeat(dense<any> in, $usize times, $usize level);`

`nested<any> = repeat(nested<any> in, $usize times, $usize level);`

Description

A `repeat` statement repeats each element or segment of a dense container the same number of times.

A `repeat` statement accepts a data container *`in`*, a size *`times`*, and a nesting level *`level`* and generates a container of the same type as the data container. The associated type of a `repeat` statement is that of the resulting container. The size value determines how many times each element/segment of the input container is repeated to generate the resulting container. The statement collates repeated values.

The level information is relevant only to nested containers. Valid values of the *`level`* parameter mean the following:

- If *`level`* = 0, the statement operates individual elements
- If *`level`* = 1, the statement operates segments

Example

```
local dense<$f32> _data;
local dense<$f32> _repeated;
_repeated = repeat<dense<$f32>>(_data, $usize(5), $usize(0));
```

See Also

Notational Conventions for Statements
repeat_row, repeat_col, repeat_page
distribute

distribute

Repeats each element or segment of a container a given number of times.

Syntax

`dense<any> = distribute(dense<any> in, dense<$usize> times, $usize level);`

`dense<any> = distribute(dense<any> in, $usize times, $usize level);`

`nested<any> = distribute(nested<any> in, nested<$usize> times, $usize level);`

`nested<any> = distribute(nested<any> in, $usize times, $usize level);`

Description

A `distribute` statement repeats each element or segment of a container a given number of times. It accepts a data container *`in`*, a container of sizes or a size scalar *`times`*, and a nesting level *`level`* and generates a container of the same type as the data container.

Provide *in* and *times* containers of the same size. To generate the result, the statement repeats each element/segment in the data container as many times as indicated by the corresponding entry in the *times* container. The statement does not collate repeated values. If the *times* parameter is a scalar, it is equivalent to the *times* container of the same shape as the data container with each element equal to the given value.

The level information is relevant only to nested containers. Valid values of the *level* parameter mean the following:

- If *level* = 0, the statement operates individual elements
- If *level* = 1, the statement operates segments

Example

```
local dense<$f32> _data;
local dense<$usize> _sizes;
local dense<$f32> _repeated;
_repeated = distribute<dense<$f32>>(_data, _sizes, $usize(0));
```

See Also

Notational Conventions for Statements
repeat

repeat_row, repeat_col, repeat_page

Create multiple copies of a container along the respective dimension.

Syntax

`dense<any, 2> = repeat_row(dense<any, 1>, $usize);`

`dense<any, 2> = repeat_col(dense<any, 1>, $usize);`

`dense<any, 2> = repeat_page(dense<any, 1>, $usize);`

Description

`repeat_row`, `repeat_col`, and `repeat_page` statements take a dense container and create multiple copies of it along the respective dimension. The operation increases the dimensionality of the container by one.

Each of these statements takes a container to be repeated and the number of times the container is to be repeated. `repeat_row`, and `repeat_col` statements accept one-dimensional containers and return two-dimensional containers. The repeat_page statement accepts a two-dimensional container and returns a three-dimensional container.

The associated type of each of these statements is the element type of the resulting container.

Example

```
local dense<$f32, 1> _1;
local dense<$f32, 2> _2;
local dense<$f32, 2> _3;
_2 = repeat_row<$f32>(_1, $usize(16));
_2 = repeat_col<$f32>(_1, $usize(8));
_3 = repeat_page<$f32>(_2, $usize(4));
```

See Also

Notational Conventions for Statements
repeat

transpose

Transposes the first two dimensionalities of a two- or three-dimensional dense or nested container.

Syntax

```
C<any> = transpose(C<any>);
```

Description

A `transpose` statement transposes the first two dimensionalities of a two- or three-dimensional dense or nested container to generate a resulting container of the same type. The associated type of a `transpose` statement is that of the resulting container.

Example

```
local dense<$f32, 2> _regular_2d;
local dense<$f32, 3> _regular_3d;
local dense<$f32, 2> _transposed_2d;
local dense<$f32, 3> _transposed_3d;
_transposed_2d = transpose<dense<$f32, 2>>(_regular_2d);
// Note the following statement transposes the first two dimensions
_transposed_3d = transpose<dense<$f32, 3>>(_regular_3d);
```

See Also

Notational Conventions for Statements

swap_row, swap_col, swap_page

Swaps two rows, columns, or pages of a dense container.

Syntax

```
dense<any, 2> = swap_row(dense<any, 2>, $usize, $usize);

dense<any, 3> = swap_row(dense<any, 3>, $usize, $usize);

dense<any, 2> = swap_col(dense<any, 2>, $usize, $usize);

dense<any, 3> = swap_col(dense<any, 3>, $usize, $usize);

dense<any, 3> = swap_page(dense<any, 3>, $usize, $usize);
```

Description

`swap_row`, `swap_col`, and `swap_page` statements take a dense container and return a new container with two rows, columns, or pages, respectively, swapped.

The first parameter of each of these statements is a container whose elements are to be swapped, and the second and third parameters are the indices of the rows, columns, or pages to be swapped. The `swap_row` and `swap_col` statements accept two- or three-dimensional dense containers. The `swap_page` statement accepts only three-dimensional dense containers.

The associated type of each of these statements is the type of the resulting container.

Example

```
local dense<$f32, 2> _a;
local dense<$f32, 2> _b;
local dense<$f32, 3> _c;
local dense<$f32, 3> _d;
_b = swap_row<dense<$f32,2>>(_a, $usize(1), $usize(2));
_d = swap_col<dense<$f32,3>>(_c, $usize(3), $usize(2));
_d = swap_page<dense<$f32,3>>(_c, $usize(6), $usize(7));
```

See Also

Notational Conventions for Statements

shift_constant, shift_constant_reverse

Shift elements of a dense container by a given offset amount along each dimension filling vacated elements with a default value.

Syntax

```
dense<any, 1> = shift_constant(dense<any, 1>, $isize, any);

dense<any, 2> = shift_constant(dense<any, 2>, $isize, $isize, any);

dense<any, 3> = shift_constant(dense<any, 3>, $isize, $isize, $isize, any);

dense<any, 1> = shift_constant_reverse(dense<any, 1>, $isize, any);

dense<any, 2> = shift_constant_reverse(dense<any, 2>, $isize, $isize, any);

dense<any, 3> = shift_constant_reverse(dense<any, 3>, $isize, $isize, $isize, any);
```

Description

`shift_constant` and `shift_constant_reverse` statements generate a new container from an existing container by looking up elements at a certain offset along each dimension.

A `shift_constant` statement accepts a container, one, two, or three scalars of the size type storing offset amounts at each dimension, and a default value that matches the element type of the container. `shift_constant` generates a resulting container of the same type as the input container.

`shift_constant` statements operate on dense containers.

The result of a `shift_constant` statement is a container of the same size as the input container, whose values are shifted by the appropriate amount. An element at index [i, j, k] of the resulting container corresponds to an element at index [i + a, j + b, k + c], where (a, b, c) are the offset amounts as passed to the statement. The statement fills an element in the resulting container with the default value if the index of the corresponding element lies outside the bounds of the input container.

A `shift_constant_reverse` statement operates the same way, but with offset parameters negated. An element at index [i, j, k] of the resulting container corresponds to an element at index [i - a, j - b, k - c].

The associated type of a `shift_constant` and `shift_constant_reverse` statements is the type of the input (and resulting) container.

Example

```
local dense<$f32> _source_1d;
local dense<$f32> _dest_1d_const;
_dest_1d_const = shift_constant<dense<$f32>>(_source_1d, $isize(1), $f32(0.0));
local dense<$f32, 2> _source_2d;
local dense<$f32, 2> _dest_2d_const;
_dest_2d_const = shift_constant<dense<$f32, 2>>(_source_2d, $isize(1),$isize(-1), $f32(0.0));
```

See Also

Notational Conventions for Statements

shift_clamp, shift_clamp_reverse

rotate, rotate_reverse

shift_clamp, shift_clamp_reverse

Shift elements of a dense container by a given offset amount along each dimension with boundary clamping.

Syntax

```
dense<any, 1> = shift_clamp(dense<any, 1>, $isize);

dense<any, 2> = shift_clamp(dense<any, 2>, $isize, $isize);

dense<any, 3> = shift_clamp(dense<any, 3>, $isize, $isize, $isize);

dense<any, 1> = shift_clamp_reverse(dense<any, 1>, $isize);

dense<any, 2> = shift_clamp_reverse(dense<any, 2>, $isize, $isize);

dense<any, 3> = shift_clamp_reverse(dense<any, 3>, $isize, $isize, $isize);
```

Description

`shift_clamp` and `shift_clamp_reverse` statements generate a new container from an existing container by looking up elements at a certain offset along each dimension. Each of these statements behaves the same way as a `shift_constant` or `shift_constant_reverse` statement, respectively, except when the index of the corresponding element of the input container lies outside its bounds. `shift_clamp` and `shift_clamp_reverse` clamp such indices in each dimension for them to lie within the bounds of the input container.

The associated type of a `shift_clamp` and `shift_clamp_reverse` statements is the type of the input (and resulting) container.

Example

```
local dense<$f32> _source_1d;
local dense<$f32> _dest_1d_clamped;
_dest_1d_clamped = shift_clamp<dense<$f32>>(_source_1d, $isize(1));
local dense<$f32, 2> _source_2d;
local dense<$f32, 2> _dest_2d_clamped;
_dest_2d_clamped = shift<dense<$f32, 2>>(_source_2d, $isize(1),$isize(-1));
```

See Also

Notational Conventions for Statements
rotate, rotate_reverse

rotate, rotate_reverse

Shift elements of a dense container along each dimension by a given offset amount taken modulo the respective size of the container.

Syntax

```
dense<any, 1> = rotate(dense<any, 1>, $isize);

dense<any, 2> = rotate(dense<any, 2>, $isize, $isize);

dense<any, 3> = rotate(dense<any, 3>, $isize, $isize, $isize);

dense<any, 1> = rotate_reverse(dense<any, 1>, $isize);

dense<any, 2> = rotate_reverse(dense<any, 2>, $isize, $isize);

dense<any, 3> = rotate_reverse(dense<any, 3>, $isize, $isize, $isize);
```

Description

`rotate` and `rotate_reverse` statements generate a new container from an existing container by looking up elements at a certain offset along each dimension.

`rotate` takes each offset modulo the respective size of the container to avoid going out of bounds. A `rotate` statement accepts a container and one, two, or three scalars of the size type storing offset amounts at each dimension. `rotate` generates a resulting container of the same type as the input container.

`rotate` statements operate on dense containers.

The result of a `rotate` statement is a container of the same size as the input container. The resulting container is the input container rotated by the appropriate amount. An element at index [i, j, k] of the resulting container corresponds to the element at index [i + a mod w, j + b mod h, k + c mod d] of the input container, where (a, b, c) are the offset amounts passed to the statement and (w, h, d) are the width, height, and depth of the input container, respectively.

A `rotate_reverse` statement operates the same way, but with offset parameters negated. An element at index [i, j, k] of the resulting container corresponds to the element at index [i - a mod w, j - b mod h, k - c mod d] of the input container.

The associated type of a `rotate` and `rotate_reverse` statements is the type of the input (and resulting) container.

Example

```
local dense<$f32> _source_1d;
local dense<$f32> _dest_1d;
_dest_1d = rotate<dense<$f32>>(_source_1d, $isize(1));
local dense<$f32, 2> _source_2d;
local dense<$f32, 2> _dest_2d;
_dest_2d = rotate<dense<$f32, 2>>(_source_2d, $isize(1), index(-1));
```

See Also

Notational Conventions for Statements
shift_constant, shift_constant_reverse
shift_clamp, shift_clamp_reverse

reverse

Reverses the order of elements in a one-dimensional dense container.

Syntax

```
dense<any, 1> = reverse(dense<any, 1>);
```

Description

A `reverse` statement generates a new one-dimensional dense container from an existing one-dimensional dense container by reversing the order of elements within the container. It accepts a single parameter specifying the source container and returns a container of the same type and size.

The associated type of a `reverse` statement is the type of the input (and resulting) container.

Example

```
local dense<$f32> _source;
local dense<$f32> _dest;
_dest = reverse<dense<$f32>>(_source);
```

See Also

Notational Conventions for Statements

Facility Statements

Grammar

```
facility-stmt ::= arg-seq '=' facility-stmt-id '(' arg-seq ')'attributes[opt] ';'

facility-stmt-id ::= 'const_vector' | 'bitwise_cast' | 'cast' | 'cat' | 'extract'
```

```
| 'replace' | 'replace_row' | 'replace_col' | 'replace_page'
| 'replace_element' | 'section' | 'index'
| 'extract_row' | 'extract_col' | 'extract_page'
| 'get_elt_coord' | 'get_neighbor' | 'expect_size'
| 'mask' | 'sort ' | 'sort_rank' | 'alloc'
| 'length' | 'get_nrows' | 'get_ncols' | 'get_npages'
| 'set_regular_nesting' | 'apply_nesting' | 'copy_nesting'
| 'get_nesting' | 'split' | 'unsplit'
| 'flatten' | 'segment' | 'replace_segment'
```

const_vector

Creates a one-dimensional dense container.

Syntax

```
dense<any> = const_vector( any value, $usize );
```

Description

A `const_vector` statement creates a one-dimensional dense container. The statement accepts two scalar input parameters. The first parameter provides the initial value for each element of the resulting container. The second parameter specifies the size of the resulting container.

See Also

Notational Conventions for Statements

bitwise_cast

Performs the bit-pattern conversion of the type (type-cast).

Syntax

```
any = bitwise_cast(any);

dense<any, N> = bitwise_cast(dense<any, N>);

nested<any> = bitwise_cast(nested<any>);
```

The dimensionality `N` = 1, 2, 3.

Description

A `bitwise_cast` statement performs the bit-pattern conversion of the type (type cast) of an input parameter. The input parameter and result can be both scalars, dense containers, or nested containers of the same shape. The bit pattern of the output data is the same as the bit pattern of the input data.

NOTE Intel® Array Building Blocks (Intel® ArBB) Virtual Machine Application Programming Interface does not support implicit type conversion of Intel ArBB types.

See Also

Notational Conventions for Statements
cast

cast

Performs conversion of the type (type cast).

Syntax

```
any = cast(any);
```

```
dense<any, N> = cast(dense<any, N>);
```

```
nested<any> = cast(nested<any>);
```

The dimensionality `N` = 1, 2, 3.

Description

A `cast` statement performs conversion of the type (type cast) of an input parameter. The input parameter and the result can be both scalars, dense containers, or nested containers of the same shape. `cast` converts data from the input type to the output type. For a `cast` statement, the width or bit orientation of the input data may change in the output. The bit pattern of the output data may differ from the one of the input data.

NOTE Intel® Array Building Blocks (Intel® ArBB) Virtual Machine Application Programming Interface does not support implicit type conversion of Intel ArBB types.

See Also

Notational Conventions for Statements
bitwise_cast

cat

Concatenates two one-dimensional or nested containers.

Syntax

```
dense<any> = cat(dense<any> in1, dense<any> in2);
```

```
nested<any> = cat(nested<any> in1, nested<any> in2);
```

Description

A `cat` statement concatenates two one-dimensional containers or two nested containers to generate the resulting container. For one-dimensional input containers, the length of the resulting container equals the sum of the lengths of these containers. For nested input containers, the number of segments in the resulting container equals the sum of the numbers of segments in these containers.

See Also

Notational Conventions for Statements

extract

Extracts an element from a dense container.

Syntax

```
any = extract(dense<any,1> in, $usize index);
```

```
any = extract(dense<any,2> in, $usize row, $usize col);
```

```
any = extract(dense<any,3> in, $usize row, $usize col, $usize page);
```

```
any = extract(nested<any,3> in, $usize segment, $usize index);
```

Description

An `extract` statement extracts an element from an input dense or nested container *`in`*. The following parameters of the statement determine the element to be extracted:

- *`index`* - index of the element in case of a one-dimensional input dense container
- *`row`*, *`col`* - row and column indices in case of a two-dimensional input dense container
- *`row`*, *`col`*, *`page`* - row, column, and page indices in case of a three-dimensional input dense container
- *`segment`* and *`index`* - segment and index in case of a nested input container

The result of the statement is a scalar of the same type as the container.

See Also

Notational Conventions for Statements

replace

Replaces specified elements in a dense container with elements from another dense container.

Syntax

```
dense<any,1> = replace(dense<any,1> in, $usize first, $usize length, $usize stride,
dense<any,1> values);
```

```
dense<any,2> = replace(dense<any,2> in, $usize first_row, $usize num_rows, $usize
row_stride, $usize first_col, $usize num_cols, $usize col_stride, dense<any,2> values);
```

Description

A `replace` statement replaces elements in a dense container *`in`* with elements from a dense container *`values`*. The statement returns a dense container of the same size as *`in`*. Parameters of the statement determine the elements to be replaced:

- *`first, length`*, and *`stride`* - for the one-dimensional flavor
- *`first_row, num_row, row_stride`* and *`first_col, num_cols, col_stride`* - for the two-dimensional flavor

If the values of the scalar parameters specify an element to be replaced that is out of bounds of the *`in`* container, the result is undefined.

See Also

Notational Conventions for Statements
replace_element
replace_row, replace_col, replace_page

replace_row, replace_col, replace_page

Replace a specified row, column, or page in a dense container with another dense container.

Syntax

```
dense<any,2> = replace_row( dense<any,2> in, $usize row, dense<any,1> new_row);
```

```
dense<any,2> = replace_col( dense<any,2> in, $usize col, dense<any,1> new_col);
```

```
dense<any,3> = replace_page( dense<any,3> in, $usize page, dense<any,2> new_page);
```

Description

`replace_row`, `replace_col`, and `replace_page` statements replace a given row at index *`row`*, column at index *`col`*, or page at index *`page`* in a given dense container *`in`* with a given dense container *`new_row`*, *`new_col`*, or *`new_page`* of an appropriate dimension.

See Also

Notational Conventions for Statements
replace
replace_element

replace_element

Replaces an element in a dense container with a given value.

Syntax

```
dense<any,1> = replace_element(dense<any,1> in, $usize index, any new_value);
```

```
dense<any,2> = replace_element(dense<any,2> in, $usize row, $usize col, any new_value);
```

```
dense<any,3> = replace_element(dense<any,3> in, $usize page, $usize row, $usize col,
any new_value);
```

Description

A `replace_element` statement replaces an element in a dense container *`in`* with a given value *`new_value`*. The parameters determine the element to be replaced:

- *`index`* - index of the element for the one-dimensional flavor
- *`row`*, *`col`* - row and column indices for the two-dimensional flavor
- *`page`*, *`row, col`*, - page, row, and column indices for the three-dimensional flavor

See Also

Notational Conventions for Statements
replace
replace_row, replace_col, replace_page

section

Clips specified elements out of a dense container.

Syntax

```
dense<any,1> = section(dense<any,1> in, $usize offset, $usize length, $usize stride);
```

```
dense<any,2> = section(dense<any,2> in, $usize row_offset, $usize num_rows, $usize
row_stride, $usize col_offset, $usize num_cols, $usize col_stride);
```

```
dense<any,3> = section(dense<any,3> in, $usize page_offset, $usize num_pages, $usize
page_stride, $usize row_offset, $usize num_rows, $usize row_stride, $usize col_offset,
$usize num_cols, $usize col_stride);
```

Description

A `section` statement clips elements out of a dense container *`in`* and places them into the resulting container. The following parameters of the statement determine the elements to be clipped out:

- *`offset, length`*, and *`stride`* - for the one-dimensional flavor
- *`row_offset, num_rows, row_stride`* and *`col_offset, num_cols, col_stride`* - for the two-dimensional flavor

- *page_offset, num_pages, page_stride, row_offset, num_rows, row_stride*, and *col_offset, num_cols, col_stride* - for the three-dimensional flavor

The `section` statement is not in-place. Stride values are given as the distance from one element to the next one to be included. To calculate row and page strides, be aware that consecutive rows in a given page are laid end-to-end in linear memory and the last row of a page comes immediately before the first row on the next page. For example, for a container with a width of five, to include every other row in the result of the `section` statement, use a row stride of ten.

See Also

Notational Conventions for Statements

index

Fills a one- or two-dimensional dense container with non-negative integer values using a linear pattern.

Syntax

```
dense<any,1> = index(val start, $usize length, val stride);
```

```
dense<any,2> = index(val start, $usize length, val stride, $usize num_times, $boolean
along_row);
```

Description

An `index` statement fills a dense container with non-negative values using a linear pattern. For the one-dimensional flavor of the statement, the pattern is determined by *start*, *length*, and *stride*. For example,

`index(0, 5, 1)` returns a one-dimensional container with the values [0, 1, 2, 3, 4].

`index(1, 10, 2)` returns a one-dimensional container with the values [1, 3, 5, 7, 9, 11, 13, 15, 17, 19].

The two-dimensional flavor fills a one-dimensional container the same way as the one-dimensional flavor and repeats this container *num_times* times. The value of the *along_row* parameter determines how this repetition is done:

- true - along rows
- false - along columns

See Also

Notational Conventions for Statements

extract_row, extract_col, extract_page

Extract a specified row, column, or page from a dense container.

Syntax

```
dense<any,1> = extract_row(dense<any,2> in, $usize row);
```

```
dense<any,1> = extract_col(dense<any,2> in, $usize col);
```

```
dense<any,2> = extract_page(dense<any,3> in, $usize page);
```

Description

`extract_row` and `extract_col` statements extract a given row at index *row* or column at index *col* from a two-dimensional dense container *in* to generate a one-dimensional dense container of size equal to the width or height of the input container, respectively. An `extract_page` statement extracts a given page at index *page* from a three-dimensional dense container *in* to generate a two-dimensional dense container of the same width and height as the input container.

See Also

Notational Conventions for Statements

extract

get_elt_coord

Returns the current location of an induction variable inside a `map` *function.*

Syntax

```
$usize, $usize, $usize = get_elt_coord();
```

Description

A `get_elt_coord` statement returns the current location of an induction variable inside a map function, that is, a function called through a `map` statement. For unused indices, the statement returns 0.

See Also

Notational Conventions for Statements

Function Call Statements

get_neighbor

For an element of container used in a map, returns the container element in a specified position relative to the current location in the map.

Syntax

```
any = get_neighbor(any in, $isize page_offset, $isize row_offset, $isize col_offset);
```

Description

A `get_neighbor` statement accepts a scalar *`in`* and three scalar offsets. The output is a scalar of the same type as *`in`*.

If *`in`* is an element of a container that is used in a map, the output is the element in the position (*`page_offset, row_offset, col_offset`*) relative to the current location in the map. If the resulting position is out of bounds, the statement returns 0. For example, if the map induction variables are (0, 1, 2), `get_neighbor(x, 0, 0, -1)` returns the element in the position (0, 1, 1).

If *`in`* is not an element of a container, the output is *`in`*.

See Also

Notational Conventions for Statements

Function Call Statements

expect_size

Associates a width, height, and depth with a given dense container.

Syntax

```
expect_size(dense<any, N> in, $usize width, $usize height, $usize depth);
```

Description

An `expect_size` statement associates an assumed width, height, and depth with a given dense container. This information is used to perform specific optimizations during compilation. If the actual dimensions of a dense container do not match the dimensions specified in the `expect_size` statement, the execution results in an error.

See Also

Notational Conventions for Statements

mask

Creates a boolean one-dimensional dense container.

Syntax

```
dense<$boolean> = mask($usize nelts, $usize start, $usize ones, $usize stride);
```

Description

A mask statement creates a one-dimensional dense container of the boolean type defined by the parameters:

- *nelts* is the size of the container.
- *start, ones*, and *stride* determine the location and number of true values.

See Also

Notational Conventions for Statements

sort, sort_rank

Sort a one-dimensional dense container.

Syntax

```
dense<any> = sort(dense<any> in, $usize sort_direction);
```

```
dense<any>, dense<$usize> = sort_rank(dense<any> in, $usize sort_direction);
```

Description

A `sort` and `sort_rank` statements sort a one-dimensional dense container *in* and return the sorted container. `sort_rank` also returns a dense container storing the location from which each element in the result was retrieved.

The *sort_direction* input scalar specifies the sort order:

- If *sort_direction* = 0, the sort order is ascending
- If *sort_direction* = 1, the sort order is descending

See Also

Notational Conventions for Statements

alloc

Allocates memory for a dense container.

Syntax

```
dense<any,1> = alloc($usize size);
```

```
dense<any,2> = alloc($usize nrows, $usize ncols);
```

```
dense<any,3> = alloc($usize npages, $usize nrows, $usize ncols);
```

Description

An `alloc` statement allocates memory for a dense container. For each dimension of the dense container, it accepts a scalar indicating the size of the container in the number of elements.

See Also

Notational Conventions for Statements

length

Returns the total size of a dense container.

Syntax

```
size = length(dense<any, N>);
```

The dimensionality `N` = 1, 2, 3.

Description

A `length` statement returns the total size of a one-, two-, or three-dimensional dense container.

See Also

Notational Conventions for Statements

get_nrows, get_ncols, get_npages

Return the number of rows, columns, or pages for a given container.

Syntax

```
$usize = get_nrows(dense<any,2>);

$usize = get_nrows(dense<any,3>);

$usize = get_ncols(dense<any,2>);

$usize = get_ncols(dense<any,3>);

$usize = get_npages(dense<any,3>);
```

Description

`get_nrows` and `get_ncols` statements return the number of rows and the number of columns, respectively, for a given two- or three-dimensional container. A `get_npages` statement returns the number of pages for a given three-dimensional container.

See Also

Notational Conventions for Statements

Nesting Statements

Nesting statements, which refer to facility statements, transform one-dimensional dense containers to nested containers and vice versa. Nested containers are used to represent irregular vectors.

Nesting Descriptors

To define an irregular vector, you need to define its nesting structure. An irregular vector consists of subvectors, or segments. Nesting descriptors are one-dimensional dense containers that specify the segments of a nested container. The nesting descriptor, along with the descriptor type, determines the shape of a nested container.

The following table specifies possible types of nested containers that you can create and describes the respective nesting descriptors.

Type	Value	Nesting Descriptor
`VS_LENGTH`	1	A `$usize` dense container with one element for every segment in the nested container. The elements of the nesting descriptor specify the lengths of the segments of the nested container.

Type	Value	Nesting Descriptor
`VS_OFFSET`	2	A `$usize` dense container with one element for every segment in the nested container. Each element of the nesting descriptor specifies the offset of the start of the corresponding segment from the start of the nested container.
`VS_FLAG`	3	A `$boolean` dense container of the same length as the nested container. Each `true` value in the nesting descriptor denotes that a new segment starts at the corresponding element of the nested container.

For example, the following nesting descriptors are equivalent and all produce a nested container with the shape:
{ {a, b, c}, {d, e, f, g}, {h, i} }:

Type	Value
`VS_LENGTH`	{3, 4, 2}
`VS_OFFSET`	{0, 3, 7, 9}
`VS_FLAG`	{`true, false, false, true, false, false, false, true, false`}

set_regular_nesting

Generates a two- or three-dimensional nesting descriptor for regular nesting.

Syntax

`dense<any, 2> = set_regular_nesting(dense<any,1>, $usize` *`height`*`, $usize` *`width`*`);`

`dense<any, 3> = set_regular_nesting(dense<any,1>, $usize` *`depth`*`, $usize` *`height`*`, $usize` *`width`*`);`

Description

A `set_regular_nesting` statement generates regular nesting, that is, a two- or three-dimensional nesting descriptor. The statement accepts a one-dimensional dense container, *`depth`* for the three-dimensional flavor, *`height`*, and *`width`*.

See Also

Notational Conventions for Statements

apply_nesting, copy_nesting, get_nesting

Create a nested container; get the nesting descriptor of a nested container.

Syntax

`nested<any> = apply_nesting(dense<any>` *`in`*`, dense<$usize>` *`desc`*`, $usize` *`descriptor_type`*`);`

`nested<any> = apply_nesting(dense<any>` *`in`*`, dense<$boolean>` *`desc`*`, $usize` *`descriptor_type`*`);`

`nested<any> = copy_nesting(dense<any>` *`in`*`, dense<any>` *`desc_source`*`);`

`dense<$usize> = get_nesting(nested<any>` *`in`*`, const $usize` *`descriptor_type`*`);`

`dense<$boolean> = get_nesting(nested<any>` *`in`*`, const $usize` *`descriptor_type`*`);`

Description

`apply_nesting` and `copy_nesting` statements operate on both regular and irregular nested containers, but are more often used for irregular containers.

An `apply_nesting` statement applies to a dense container *`in`* the nesting structure determined by a nesting descriptor *`desc`* and the type of a nested container provided in the *`descriptor_type`* parameter.

A `copy_nesting` statement creates a nested container using the element type and data provided in the dense container *`in`* and a nesting descriptor of the nested container *`desc_source`*. The element types of the input dense and nested containers can be different. The lengths of the input dense and nested containers must match.

A `get_nesting` statement takes a nested container and returns a copy of its nesting descriptor in a dense container. `get_nesting` returns:

`descriptor_type` specifies how to interpret the nesting descriptor data. For all these statements, the type of the input or output nesting descriptor container must match the value of *`descriptor_type`*, that is, be:

- `dense<$usize>` for *`descriptor_type`* = 1 or *`descriptor_type`* = 2.
- `dense<$boolean>` for *`descriptor_type`* = 3.

See Also

Nesting Descriptors
Notational Conventions for Statements

split, unsplit

Creates a nested container by splitting an input container according to index values; performs an inverse operation.

Syntax

```
nested<any> = split(dense<any> in1, dense<int> in2);

nested<any> = split(nested<any> in1, nested<int> in2);

nested<any> = unsplit(nested<any> in, dense<int> ind);

nested<any> = unsplit(nested<any> in, nested<int> ind);
```

Description

A `split` statement takes two dense containers or two nested containers (*`in1`* and *`in2`*) of the same shape. The indices have the values of -1, 0, or 1. The result is a nested vector resulting from sorting the first parameter according to the values in the second parameter and with a segment added per each index value.

An `unsplit` statement takes a nested container *`in`* to specify the values and a dense or nested container *`ind`* to specify indices and returns a nested container with the provided values at the corresponding indices. The resulting container has the same shape as *`ind`*. The *`ind`* container must have the same shape as the resulting container.

See Also

Notational Conventions for Statements

flatten

Turns a nested container or two- or three-dimensional dense container into the one-dimensional dense container with the same values.

Syntax

```
dense<any, 1> = flatten(nested<any>);
```

```
dense<any, 1> = flatten(dense<any, 2>);
```

```
dense<any, 1> = flatten(dense<any, 3>);
```

Description

A `flatten` statement accepts a nesting container and returns a dense container with the same values as the nested container but without a nesting descriptor. `flatten` also turns a two- or three-dimensional dense container into a one-dimensional dense container.

See Also

Notational Conventions for Statements

segment

Returns a dense segment of a nested container.

Syntax

```
dense<any> = segment(nested<any>, $usize index);
```

Description

For a nested container, a `segment` statement returns the dense segment indicated by *`index`*.

See Also

Notational Conventions for Statements

replace_segment

Replaces a dense segment of a nested container.

Syntax

```
nested<any> = replace_segment(nested<any>, $usize index, dense<any> values);
```

Description

In an input nested container, a `replace_segment` statement replaces the dense segment indicated by *`index`* with the specified values.

See Also

Notational Conventions for Statements

Collective Statements

Collective statements consist of the following groups of statements:

- Reduction
- Scan
- Merge

Each of these groups is described in a separate section which contains a table that lists the statements in this group.

Collective statements are executed on containers. In a collective statement, the output containers always have the same element type as the input containers.

Scans and reductions operate on multi-dimensional containers along a particular dimension, or level. The following table describes the effect the level has on an operation.

Level	Description of the Operation
0	Operates on elements in the same row

Level	Description of the Operation
1	Operates on elements in the same column
2	Operates on elements in the same row and column of each page

Reduction Statements

Grammar

```
reduce-stmt    ::= arg-seq '=' reduce-stmt-id '<' type '>' '(' arg-seq
')'attributes[opt] ';'

reduce-stmt-id ::= 'add_reduce' | 'mul_reduce ' | 'max_reduce' | 'max_reduce_loc'

                 | 'min_reduce' | 'min_reduce_loc'

                 | 'and_reduce' | 'ior_reduce' | 'xor_reduce'
```

Description

Reduction statements apply a combining operation across a container to reduce it to a smaller set of values.

Each reduction statement accepts a container to reduce and an optional parameter specifying the level along which to perform the reduction. If level is not specified, 0 is assumed. A reduction statement returns a reduced container or scalar of the same primitive type, and, in the case of the "loc" variants, a reduced container of indices.

The following table shows the shape of the output of a reduction statement and possible values of level.

Shape of the Input Parameter	Shape of the Output Parameter	Possible Values of Level
`dense<T,3>`	`dense<T,2>`	0, 1, 2
`dense<T,2>`	`dense<T,1>`	0, 1
`dense<T,1>`	`T`	0
`nested<T>`	`dense<T,1>`	0

The following table describes available reduction statements. The table uses the placeholders for types as specified in the Notational Conventions for Statements. Additionally the placeholder `R<T>` stands for the reduced type of the input source container.

Name	Syntax	Description
`add_reduce`	`R<val> = add_reduce<val>(C<val> `*`in`*`, $usize `*`level`*`);`	Adds all elements along the specified level.
`mul_reduce`	`R<val> = mul_reduce<val>(C<val> `*`in`*`, $usize `*`level`*`);`	Multiplies all elements along the specified level.
`max_reduce`	`R<val> = max_reduce<val>(C<val> `*`in`*`, $usize `*`level`*`);`	Computes the maximum of all elements along the specified level.

Name	Syntax	Description
`max_reduce_loc`	`R<val>, R<$usize> = max_reduce_loc<val>(C<val> in, $usize level);`	Computes the maximum of all elements along the specified level and returns their locations.
`min_reduce`	`R<val> = min_reduce<val>(C<val> in, $usize level);`	Computes the minimum of all elements along the specified level.
`min_reduce_loc`	`R<val>, R<$usize> = min_reduce_loc<val>(C<val> in, $usize level);`	Computes the minimum of all elements along the specified level and returns their locations.
`and_reduce`	`R<$boolean> = and_reduce< $boolean>(C<$boolean> in, $usize level);`	Computes the logical *and* of all elements along the specified level.
`ior_reduce`	`R<$boolean> = ior_reduce< $boolean>(C<$boolean> in, $usize level);`	Computes the logical inclusive *or* of all elements along the specified level.
`xor_reduce`	`R<$boolean> = xor_reduce< $boolean>(C<$boolean> in, $usize level);`	Computes the logical exclusive *or* of all elements along the specified level.

The following algorithm explains execution of a reduction statement in general:

Algorithm of `op_reduce(C<val> source, $usize level = 0)`:

1. If the container "source" has one dimension,
   a. Initialize a set of "reducibles" to all elements of the provided container
   b. If the set of "reducibles" contains exactly one element, return that element
   c. Otherwise, continue execution
   d. Pick two distinct elements from "source"
   e. Remove these elements from the set
   f. Run an instance of the operation on these two elements as an input
   g. Add the resulting element to the set of "reducibles"
   h. Go to step "b"
2. Otherwise,
   a. Divide the container "source" into as many subcontainers as possible according to the level selected
   b. Call `op_reduce(C<val> sub_source, level)` for each subcontainer
   c. For dense containers, append the results to a new container in a way corresponding to the level chosen
   d. For nested containers, append the results to a new dense container, where each nested segment is reduced to an element in the new dense container
   e. Return the resulting container

NOTE This algorithm imposes no requirement on the order of processing elements during execution. Though the reduce statement yields the same result, the statement may be implemented differently.

Scan Statements

Grammar

```
scan-stmt-id ::= 'add_scan' | 'mul_scan | 'max_scan' | 'min_scan'
               | 'and_scan' | 'ior_scan' | 'xor_scan'
```

Description

A scan statement performs an exclusive prefix-sum equivalent to the corresponding reduction statement, storing the intermediate values in each element of the resulting sequence.

Each scan statement accepts three parameters: a container to scan, a the direction of the scan, and the level along which to perform the scan.

The values of the second parameter (*`dir`*) specify the direction as follows:

- 0 - from low to high indices
- 1 - from high to low indices

Possible values of the level (*`level`* parameter) are the same as for reduce statements.

Scan statements return a container of the same type and dimension as the input container.

The following table lists scan statements. The table uses placeholders for types as specified in the Notational Conventions for Statements.

Name	Syntax	Description
`add_scan`	`C<val> = add_scan<val>(C<val> in, $usize dir, $usize level);`	Computes the prefix sum along the specified level in the specified direction.
`mul_scan`	`C<val> = mul_scan<val>(C<val> in, $usize dir, $usize level);`	Computes the prefix product along the specified level in the specified direction.
`max_scan`	`C<val> = max_scan<val>(C<val> in, $usize dir, $usize level);`	Computes the prefix maximum along the specified level in the specified direction.
`min_scan`	`C<val> = min_scan<val>(C<val> in, $usize dir, $usize level);`	Computes the prefix minimum along the specified level in the specified direction.
`and_scan`	`C<$boolean> = and_scan<$boolean>(C<$boolean> in, $usize dir, $usize level);`	Computes the prefix logical *and* along the specified level in the specified direction.
`ior_scan`	`C<$boolean> = ior_scan<$boolean>(C<$boolean> in, $usize dir, $usize level);`	Computes the prefix logical inclusive *or* along the specified level in the specified direction.
`xor_scan`	`C<$boolean> = xor_scan<$boolean>(C<$boolean> in, $usize dir, $usize level);`	Computes the prefix logical exclusive *or* along the specified level in the specified direction.

Depending on the *`dir`* parameter, the following element in the resulting container acquires the initial value of the operation:

- first element for *`dir`*=0
- last element for *`dir`*=1

The following table describes the initial values of each scan operation:

Scan Operation	**Initial Value**	**Description**
`add`	0	
`mul`	1	
`max`	min(*<type>*)	Minimum value for each type. For example: • 0 for `u8` • 0x80 for `i8`
`min`	max(*<type>*)	Maximum value for each type. For example: • 0xFF for `u8` • 0x7F for `i8`
`and`	`true`	
`ior`	`false`	
`xor`	`false`	

The following algorithm explains execution of a scan statement in general:

Algorithm of `op_scan(C<val> source, $usize direction = 0, $usize level = 0)`:

1. If the container "source" has one dimension,

  a. If the direction is 1, reverse the sequence of elements in the set "source"
  b. Add the initial value of the operation to the start of "source"
  c. Remove the last element from "source"
  d. Run an instance of `op_scan_dimension` passing "source", direction, and level
  e. If the direction is 1, return the reversed result of step "d'

2. Otherwise,

  a. If the container "source" is nested, divide it into subcontainers according to its segments
  b. Otherwise, divide "source" into as many subcontainers as possible according to the level specified
  c. Run an instance of `op_scan` for each subcontainer, replacing the subcontainer with the result
  d. Return "source"

Algorithm of `op_scan_dimension(C<val> source, $usize direction = 0, $usize level = 0)`:

1. Create a pivot point within the container "source" and partition the data into two new sets: "left" and "right"
2. If the "left" set is empty, then return the "right" set
3. Otherwise, if there is exactly one element in "left",

  a. Run an instance of `op` on the only element of "left" and the first element of "right"
  b. Replace the first element in the set "right" with the result of step "a"
  c. Run an instance of `op_scan_dimension` on the "right" set
  d. Return a new set that is the concatenation of sets "left" and "right"

4. Otherwise,

  a. Recursively call `op_scan` with the "left" set being passed in as "source"
  b. Run an instance of `op` on the last element of "left" and the first element of "right"
  c. Replace the first element in the set "right" with the result of step "b"
  d. Return a new set that is the concatenation of sets "left" and "right"

NOTE This algorithm imposes no requirement on the order of processing elements during execution. Though the scan statement yields the same result, the statement may be implemented differently.

See Also

Reduction Statements

Merge Statements

Grammar

```
merge-stmt-id ::= 'add_merge' | 'add_merge_scalar'
```

Description

The following table lists available merge statements. In the table, placeholders stand for types as specified in the Notational Conventions for Statements.

Name	Syntax	Description
`add_merge`	`dense<val> = add_merge(dense<val> *in*, dense<$usize> *indices*, $usize *length*);`	Computes the merge sum.
`add_merge_scalar`	`dense<val> = add_merge(val *in*, dense< $usize> *indices*, $usize *length*);`	

A merge operation is similar to a scatter operation, except that merge permits duplicate indices and resolves the duplication by reductions over the data items sharing an index.

An `add_merge` statement accumulates data in a new container according to given set of indices. An `add_merge` statement accepts a one-dimensional dense container of input data. An `add_merge_scalar` statement accepts an input scalar. Both statements accept a one-dimensional dense container of indices (index vector) and a $usize parameter specifying the length of the resulting one-dimensional dense container. The length must be greater than any index. An input scalar is equivalent to a dense one-dimensional container that has the same length as the index vector and is filled with the value of the scalar.

An `add_merge` statement successively performs the following steps:

1. The statement creates a new container having the requested length and filled with zeros.
2. For each element of the index container, the statement places the corresponding element from the input container into the element of the result at that index.

See Also

scatter

Control Flow Statements

Grammar

```
cflow-stmt ::= if-stmt | loop-stmt | break-stmt | continue-stmt
             | return-stmt
```

NOTE All the control flow statements in the Intel ArBB VM API execute serially.

if Statements

Grammar

```
if-stmt   ::= 'if' '(' arg ')' '{' stmt-seq[opt] '}'
              else-stmt[opt]
else-stmt ::= 'else' '{' stmt-seq[opt] '}'
```

Description

An if statement consists of a condition, a true-body block and optionally an else clause with a false-body. The condition is an identifier or constant that causes the true-block or false-block to be chosen. If the condition is true, the true-body is evaluated, otherwise the false body is evaluated. The condition must be of a scalar primitive type.

Example

```
_t1 = less<$f32>(_a, _b);
if (_t1) {
  // ...
} else {
  // ...
}
```

C Application Programming Interface

An if statement starts with a call to `arbb_if`. After this call, an optional call to `arbb_else` occurs. An if statement ends with a call to `arbb_end_if`.

For example:

```
arbb_if(fn, condition1, 0);
// statements executed if condition1 is true
arbb_else(fn, 0);
// statements executed if neither condition1 nor condition2 are true
arbb_end_if(fn, 0);
/// @defgroup arbb_virtual_machine_if_statements If statements
/// @{
/// Begins an if statement, the body of which is conditionally executed if
/// the condition provided is met.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p function is a null object.
///  - ::arbb_error_invalid_argument if @p condition is a null object.
///  - ::arbb_error_invalid_argument if @p condition is not a scalar of type ::arbb_boolean.
///  - ::arbb_error_scoping if the given @p function is not currently being defined.
ARBB_VM_EXPORT
arbb_error_t arbb_if(arbb_function_t function,
                     arbb_variable_t condition,
                     arbb_error_details_t* details);
/// Adds an else portion to the current if statement.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p function is a null object.
///  - ::arbb_error_scoping if there is no if statement being defined.
///  - ::arbb_error_scoping if there was already a call to arbb_else() in the current if statement.
ARBB_VM_EXPORT
arbb_error_t arbb_else(arbb_function_t function,
                       arbb_error_details_t* details);
/// Ends the current if statement.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p function is a null object.
///  - ::arbb_error_scoping if there is no if statement being defined.
ARBB_VM_EXPORT
arbb_error_t arbb_end_if(arbb_function_t function,
                         arbb_error_details_t* details);
/// @}
```

Loops

Grammar

```
loop-stmt ::= while-stmt | do-stmt | for-stmt

loop-block-seq ::= loop-block | loop-block loop-block-seq

loop-block ::= init-block | cond-block | step-block | body-block

init-block ::= 'init' '{' stmt-seq[opt] '}'

cond-block ::= 'cond' '{' stmt-seq[opt] when-stmt '}'

step-block ::= 'step' '{' stmt-seq[opt] '}'

body-block ::= 'body' '{' stmt-seq[opt] '}'
```

C Application Programming Interface

A loop starts with a call to `arbb_begin_loop()`, which is followed by several calls to `arbb_begin_loop_block()` for each part of the statement, and ends with a call to `arbb_end_loop()`. For example, a loop statement can be as follows:

```
arbb_begin_loop(fn, arbb_loop_while, 0);
arbb_begin_loop_block(fn, arbb_loop_block_cond, 0);
// Statements to define the condition go here
arbb_loop_condition(fn, condition, 0);
arbb_begin_loop_block(fn, arbb_loop_block_body, 0);
// Loop body statements go here
arbb_end_loop(fn, 0);
```

You can define loop blocks in any order, but you must define only one block of a given type in a given loop. The following table illustrates valid types of loops and blocks they must contain:

Value of arbb_loop_type_t	**Values of arbb_loop_block_t**
`arbb_loop_for`	`arbb_loop_block_init`, `arbb_loop_block_cond`, `arbb_loop_block_step`, and `arbb_loop_block_body`
`arbb_loop_while`	`arbb_loop_block_cond` and `arbb_loop_block_body`

for Loops

Grammar

```
for-stmt ::= 'for' loop-block-seq
```

Description

A *for* loop consists of a single initializer block, a single condition block, a single step block, and a single body block, in any order. The initializer block, step block and body block can consist of arbitrary statements. The condition block ends in a `when` statement, which specifies the condition that causes the loop to begin or continue executing.

Example

```
for init {
  _i = copy<dense<$usize>>($i32(0));
} cond {
  _t1 = copy<dense<$usize>>($i32(10));
  _t2 = less<dense<$usize>>(_i, _t1);
  _t4 = any<$boolean>(_t2);
  when(_t4);
} step {
  _t3 = copy<dense<$usize>>($i32(1));
  _i = add<dense<$usize>>(_i, _t3);
} body {
  // ...
}
```

while Loops

Grammar

```
while-stmt ::= 'while' loop-block-seq
```

Description

A *while* loop consists of a single condition block and a single body block, in any order. The condition block ends in a `when` statement, which specifies the condition that causes the loop to begin or continue executing.

Example

```
_i = copy<dense<$usize>>($i32(0));
while cond {
  _t1 = copy<dense<$usize>>($i32(10));
  _t2 = less<dense<$usize>>(_i, _t1);
  _t4 = any<$boolean>(_t2);
  when(_t4);
} body {
  // ...
  _t3 = copy<dense<$usize>>($i32(1));
  _i = add<dense<$usize>>(_i, _t3);
}
```

do Loops

Grammar

```
do-stmt ::= 'do' loop-block-seq
```

Description

A *do* loop consists of a single body block, a single condition block, and an optional single step block, in any order. The condition block ends in a `when` statement, which specifies the condition that causes the loop to continue executing. The program enters the loop body at least once and enters the step block after every execution of the loop body but before the condition block is executed.

Example

```
_i = copy<dense<$usize>>($i32(0));
do body {
  // ...
  _t3 = copy<dense<$usize>>($i32(1));
  _i = add<dense<$usize>>(_i, _t3);
} cond {
  _t1 = copy<dense<$usize>>($i32(10));
  _t2 = less<dense<$usize>>(_i, _t1);
  _t4 = any<$boolean>(_t2);
  when(_t4);
}
```

break Statements

Grammar

```
break-stmt ::= 'break' attributes[opt] ';'
```

Description

A `break` statement immediately exits the current innermost loop. A break statement must be contained in a block nested within the body block of a loop or the body block itself.

Example

```
break;
```

continue Statements

Grammar

```
continue-stmt ::= 'continue' attributes[opt] ';'
```

Description

A `continue` statement causes the current innermost loop to proceed to the next iteration, if any. A `continue` statement must be contained in a block nested within the body block of a loop or the body block itself. A `continue` statement skips the rest of the execution of the body block and proceeds to the step block (or the condition block if a loop has no step block).

Example

```
continue;
```

return Statements

Grammar

```
return-stmt ::= 'return' attributes[opt] ';'
```

Description

A `return` statement causes the current function to return its caller.

Example

```
return;
```

Special Statements

when Statements

Grammar

```
when-stmt ::= 'when' '(' arg ')' attributes[opt] ';'
```

Description

A `when` statement occurs at the end of a condition block of a loop. It indicates a variable to be used to determine whether to continue a loop. This variable must be a primitive boolean variable.

Example

```
while cond {
  // ...
  when(_t2);
  // Must be last statement in block
} body {
  // ...
}
```

See Also

Loops

9 Attributes

Grammar

```
attributes        ::= '[[' attr-seq[opt] ']]'

attr-seq          ::= attr | attr ',' attr-seq

attr              ::= attr-keyword '(' attr-arg ')'

attr-keyword      ::= 'debug_info' | 'align' | attr-identifier

attr-identifier ::= '_' string-literal

attr-arg          ::= attr-value | attr-id

attr-value        ::= int-literal | string-literal | void-pointer

attr-id           ::= '![a-zA-Z][0-9a-zA-Z_]*

attr-definition ::= 'attribute' attr-id '=' attr-value

string-literal  ::= '".*"'
```

Description

Function definitions, basic block statements, and type definitions can optionally have attributes attached to them in the form of a possibly empty sequence of key-value pairs encapsulated in an `arbb_attribute_map_t` object. Such sequences are called *attribute maps*.

C Application Programming Interface (API) of the Intel® Array Building Blocks Virtual Machine provides a set of functions to create and query attribute maps. An attribute key has a string identifier and an associated type. The key identifier may be built-in or user-defined. A user-defined key identifier begins with an underscore and is unrelated to other identifiers, such as variable names. The type of a key `is` `arbb_attribute_value_type_t`. It represents the data type of the associated value.

Example of a Statement with Attributes

```
// The following statement has two attributes:
//  - a built-in kind of data called "debug_info" of type void-pointer
//  - a user defined kind of data called "_marked" of type integer
_a = add<$i32>(_b, _c) [[*debug_info(0xcafecafe), _marked(1)]];
```

Global Attributes

Global attribute values are located separately from the locations in which they are used. Therefore, use of these values makes the code more readable in cases such as including debugging information in your code. A global attribute value is defined in the global scope with the `attribute` keyword, followed by an identifier (possibly anonymous), an assignment symbol, and a literal providing the actual value. Global attribute values can be referenced in attributes by using the "!" symbol followed by the attribute value identifier.

Example of Global Attributes

```
function foo(out $f32 _a, in $f32 _b, in $f32 _c) [[_sourceline(!_0)]] {
  _a = add<$f32>(_b, _c) [[_sourceline(!_1)]];
  _a = add<$f32>(_a, _c) [[_sourceline(!_1)]];
}
attribute !_0 = "/foo/bar/baz/really_long_filename.cpp:1246";
attribute !_1 = "/foo/bar/baz/really_long_filename.cpp:1248";
```

You cannot define the same global attribute value more than once.

Built-in Identifiers of Attribute Keys

The following identifiers of attribute keys are available:

Identifier	Value Type	Description
`debug_info`	`void*`	Stores additional information collected during closure capture, which may be useful for debugging and profiling.
`align`	`int_32t`	Specifies alignment of a user-defined type or variable declaration, in bytes. Must be greater than or equal to zero. A value of zero indicates the default alignment.

C API

```
/// @defgroup arbb_virtual_machine_attributes Attributes
/// @{

/// A key that you can use to index attribute values.
/// @see arbb_get_attribute_key
typedef uint32_t arbb_attribute_key_t;

/// An attribute value.
typedef union {
    int32_t as_int32;
    char* as_string;
    void* as_pointer;
} arbb_attribute_value_t;

/// An attribute key-value pair.
/// @see arbb_create_attribute_map
typedef struct {
  arbb_attribute_key_t key;
  arbb_attribute_value_t value;
} arbb_attribute_key_value_t;

/// The set of supported attribute value types.
/// @see arbb_lookup_attribute
typedef enum {
  arbb_attribute_int32, ///< A 32-bit signed integer. The result of @ref arbb_lookup_attribute is of
type <tt>int32_t</tt>
  arbb_attribute_string, ///< A null-terminated character string. The result of @ref
arbb_lookup_attribute is of type <tt>const char*</tt>
  arbb_attribute_pointer, ///< A void pointer. The result of @ref arbb_lookup_attribute is only safe to
cast to the same type as passed to @ref arbb_create_attribute_map.
} arbb_attribute_type_t;
/// Returns an attribute_key_t object with identifier @p name
/// used to index attributes of type specified by @p type
/// in an @ref arbb_attribute_map_t object.
///
/// @return An error code depending on the result of the operation
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p name is a null pointer.
///  - ::arbb_error_invalid_argument if @p out_key is a null pointer.
///  - ::arbb_error_invalid_argument if @p name is not a built-in
///      attribute key identifier and does not start with a '<tt>_</tt>'
///      character
///  - ::arbb_error_invalid_argument if an attribute key has already been
///      retrieved with the identifier @p name but with a different
///      type than specified by the @p type parameter
ARBB_VM_EXPORT
```

```
arbb_error_t arbb_get_attribute_key(arbb_context_t context,
                                   const char* name,
                                   arbb_attribute_type_t type,
                                   arbb_attribute_key_t* out_key,
                                   arbb_error_details_t* details);
/// Creates an arbb_attribute_map_t object which contains the key-value pairs
/// specified by the @attributes array.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p attribute_count is 0
///  - ::arbb_error_invalid_argument if @p attributes is a null pointer
///  - ::arbb_error_invalid_argument if @p attributes contains the same
///      key more than once
///  - ::arbb_error_invalid_argument if @p out_map is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_create_attribute_map(arbb_context_t context,
                                  unsigned int attribute_count,
                                  const arbb_attribute_key_value_t* attributes,
                                  arbb_attribute_map_t* out_map,
                                  arbb_error_details_t* details);
/// Frees the resources for @p object.
ARBB_VM_EXPORT
void arbb_free_attribute_map(arbb_attribute_map_t object);
/// Sets the pointer @p out_value to the attribute value associated with
/// @p key in @p attributes.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p attributes is a null object.
///  - ::arbb_error_invalid_argument if @p out_value is a null pointer.
///  - ::arbb_error_invalid_argument if @p attributes contains no values
///      associated with @p key.
ARBB_VM_EXPORT
arbb_error_t arbb_lookup_attribute(arbb_context_t context,
                                   arbb_attribute_map_t attributes,
                                   arbb_attribute_key_t key,
                                   void** out_value,
                                   arbb_error_details_t* details);
```

Run-time Application Programming Interface

10

To interact with a host environment where functions in the virtual machine are executed, the C Application Programming Interface (API) of the Intel® Array Building Blocks (ArBB) Virtual Machine (VM) provides functions to control execution of functions and data access of variables.

Data Access

Scalar Data Access

To copy data in scalar global variables to and from the host, use the `arbb_read_scalar` and `arbb_write_scalar` functions, respectively. It is up to the client application to ensure that the variable passed in is global and that the data has the appropriate size and is formatted correctly.

The following are the declarations of these functions:

```
/// @defgroup arbb_virtual_machine_scalar_access Scalar data access
/// @{
/// Copies data out of the given global scalar to the host.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p variable is a null object.
///  - ::arbb_error_invalid_argument if @p out_data is a null pointer.
///  - ::arbb_error_invalid_argument if @p variable is not a global variable.
///  - ::arbb_error_invalid_argument if @p variable is not a scalar.
ARBB_VM_EXPORT
arbb_error_t arbb_read_scalar(arbb_context_t context,
                              arbb_variable_t variable,
                              void* out_data,
                              arbb_error_details_t* details);
/// Copies data into the given global variable from the host.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p variable is a null object.
///  - ::arbb_error_invalid_argument if @p out_data is a null pointer.
///  - ::arbb_error_invalid_argument if @p variable is not a global variable.
///  - ::arbb_error_invalid_argument if @p variable is not a scalar.
ARBB_VM_EXPORT
arbb_error_t arbb_write_scalar(arbb_context_t context,
                               arbb_variable_t variable,
                               const void* data,
                               arbb_error_details_t* details);
/// @}
```

Container Bindings

When creating a new global container, you can pass a binding specification (of type `arbb_binding_t`) to the `arbb_create_global` function. The `arbb_create_dense_binding` function creates a binding suitable for a dense container. The binding specifies the dimensionality and sizes of each dimension. Sizes are in units of container elements. A null binding, where a null pointer is passed to `user_data`, initializes a container with a certain size without binding it to user data. If `user_data` is not null, `byte_pitches` specifies pitch values (in bytes) between columns, rows, and pages (up to the appropriate dimensionality). The following table illustrates how an element of a container is mapped to an appropriate location in user memory:

Dimensionality	Valid Range	Container Element	Bound Location
1	0 <= col < sizes[0], byte_pitches[i] >= element_size	C[col]	(char*)user_data + col * byte_pitches[0]
2	0 <= col < sizes[0], 0 <= row < sizes[1], byte_pitches[i] >= element_size	C[row, col]	(char*)user_data + col * byte_pitches[0] + row * byte_pitches[1]
3	0 <= col < sizes[0], 0 <= row < sizes[1], 0 <= page < sizes[2], byte_pitches[i] >= element_size	C[page, row, col]	(char*)user_data + col * byte_pitches[0] + row * byte_pitches[1] + page * byte_pitches[2]

Binding specifications are not reference counted. The client must free them explicitly by using the `arbb_free_binding` function.

The following are the function declarations:

```
/// @defgroup arbb_virtual_machine_binding Container bindings
/// @{
/// Creates a binding suitable for a dense container with the
/// specified parameters.
///
/// The number of elements in the @p sizes array must match the value of the
/// @p dimensionality parameter.
///
/// The number of elements in the @p byte_pitches arrays must match the value of
/// the @p dimensionality parameter.
///
/// @p byte_pitches[i] specifies the number of bytes between columns/rows/pages in
/// the user data for i = 0, 1, 2 respectively.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p dimensionality is not 1, 2, or 3.
///  - ::arbb_error_invalid_argument if @p sizes is a null pointer.
///  - ::arbb_error_invalid_argument if @p user_data is a null pointer.
///  - ::arbb_error_invalid_argument if @p byte_pitches is a null pointer.
///  - ::arbb_error_invalid_argument if any of the values in @p byte_pitches are smaller than the
element size.
///  - ::arbb_error_invalid_argument if @p out_binding is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_create_dense_binding(arbb_context_t context,
                                       arbb_binding_t* out_binding,
                                       void* user_data,
                                       unsigned int dimensionality,
                                       const uint64_t* sizes,
                                       const uint64_t* byte_pitches,
                                       arbb_error_details_t* details);
/// Releases all resources associated with the @p binding.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p binding is a null object.
ARBB_VM_EXPORT
arbb_error_t arbb_free_binding(arbb_context_t context,
                               arbb_binding_t binding,
                               arbb_error_details_t* details);
/// @}
```

See Also

Global Declarations

Container Data Mapping

Access data with a dense container by mapping the internal data of the container onto the host address space. The mapping can be read-only, write-only, or read-write. The result is a `void*` to the mapped data and a `byte_pitch` which indicates the spacing between dense elements.

The mapping is valid until the next virtual machine operation that directly or indirectly accesses the given variable.

The following is the related fragment of the header file.

```
/// @defgroup arbb_virtual_machine_container_mapping Container data mapping
/// @{
/// The set of access modes allowed for arbb_map_to_host().
///
/// @see arbb_map_to_host()
typedef enum {
  arbb_read_only_range,
  arbb_write_only_range,
  arbb_read_write_range,
} arbb_range_access_mode_t;
/// Maps the global container provided in @p container into the host
/// address space.
///
/// If @p container has no size, @p out_data is set to null.
/// If @p container was not previously written to, @p out_data
/// points to memory with uninitialized contents.
/// The mapping is valid until the next virtual machine operation
/// that directly or indirectly accesses the given variable.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p container is a null object.
///  - ::arbb_error_invalid_argument if @p out_data is a null pointer.
///  - ::arbb_error_invalid_argument if @p out_byte_pitch is a null pointer.
///  - ::arbb_error_invalid_argument if @p container is not a global variable.
///  - ::arbb_error_invalid_argument if @p container is not a dense container.
ARBB_VM_EXPORT
arbb_error_t arbb_map_to_host(arbb_context_t context,
                              arbb_variable_t container,
                              void** out_data,
                              uint64_t* out_byte_pitch,
                              arbb_range_access_mode_t mode,
                              arbb_error_details_t* details);
/// @}
```

See Also

Global Declarations

Execution and Compilation

For information on execution of VM functions, see Function Execution Semantics.

To compile VM functions, call the `arbb_compile_for_args()` function. This function compiles a version of the VM function that accepts parameters sharing the same attributes as the given parameters. The attributes that affect compilation depend on the parameter type and relate to how that parameter was created as follows:

- For input or output container parameters, whether or not the parameter was created using a non-null binding specification.
- For two- and three-dimensional container parameters created with a non-null binding specification, whether or not the binding specification was created with `byte_pitch` values indicating data striding.

Calling `arbb_compile_for_args()` on a function already compiled for a given set of parameters has no effect.

Calling `arbb_execute` on a function that has not been compiled for a given set of parameters implicitly compiles it.

Functions `arbb_compile_for_args()` and `arbb_execute` are thread safe.

To finish asynchronous executions, call the `arbb_finish` function. Current executions are synchronous, and there is no need to wait until the execution is completed. The `arbb_finish` function is useful when you are timing operations to ensure the entire operation is measured.

The following are the function declarations:

```
/// @defgroup arbb_virtual_machine_execution Execution and compilation
/// @{
/// Executes the given function.
///
/// This call is thread safe.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p function is a null object.
///  - ::arbb_error_invalid_argument if @p function has not been completely defined.
///  - ::arbb_error_invalid_argument if @p outputs is a null pointer and the function has at least one
output.
///  - ::arbb_error_invalid_argument if @p inputs is a null pointer and the function has at least one
input.
///  - ::arbb_error_invalid_argument if any of the variables in @p inputs or @p outputs do not match
the function's parameters.
///  - ::arbb_error_invalid_argument if any of the variables in @p inputs or @p outputs are not global.
///  - ::arbb_error_out_of_bounds if an attempt to access a container out of bounds was made during
execution.
///  - ::arbb_error_arithmetic if an arithmetic exception occurred during execution, such as overflow,
underflow, or an integer division by zero.
///  - ::arbb_error_bad_alloc if memory allocation failed during execution.
///  - ::arbb_error_uninitialized_access if uninitialized variable was used during execution.
ARBB_VM_EXPORT
arbb_error_t arbb_execute(arbb_function_t function,
                          const arbb_variable_t* outputs,
                          const arbb_variable_t* inputs,
                          arbb_error_details_t* details);

/// Compiles the given function for a given set of arguments.
/// The provided arugments give information required for dynamic
/// recompilation. Assuming the same arguments, or arguments that are
/// bound/not-bound the same way, are passed to arbb_execute, no compilation
/// overhead will be incurred.
///
/// This call is thread safe.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p function is a null object.
///  - ::arbb_error_invalid_argument if @p function has not been completely defined.
///  - ::arbb_error_invalid_argument if @p outputs is a null pointer and the function has at least one
output.
///  - ::arbb_error_invalid_argument if @p inputs is a null pointer and the function has at least one
input.
///  - ::arbb_error_invalid_argument if any of the variables in @p inputs or @p outputs do not match
the function parameters
///  - ::arbb_error_invalid_argument if any of the variables in @p inputs or @p outputs are not global.
ARBB_VM_EXPORT
arbb_error_t arbb_compile_for_args(arbb_function_t function,
                          const arbb_variable_t* outputs,
                          const arbb_variable_t* inputs,
                          arbb_error_details_t* details);
/// Waits until any previously issued asynchronous operations have
/// completed.
///
/// This is useful when timing operations to ensure the entire operation is measured.
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
ARBB_VM_EXPORT
arbb_error_t arbb_finish(arbb_error_details_t* details);
/// @}
```

See Also

Container Bindings

Immediate Operations

When an operation function, `arbb_op()` or `arbb_op_dynamic()`, is called with a null `arbb_function_t` parameter, it executes immediately. In this case, all input and output parameters of the operation must be global variables, otherwise an error is returned.

NOTE Execution of operations in this mode is not expected to be as fast as executing a function containing an equivalent sequence of operations because there is no optimization across operations and execution may occur on the host, possibly using emulation code.

See Also

Global Declarations

Immediate map Operations and the Coordinate Stack

The `arbb_op_get_neighbor` and `arbb_op_get_elt_coord` operations are normally used within functions later invoked during an `arbb_op_map` form of an `arbb_call_op()` invocation. They provide information that depends on the context of the map invocation, such as the current map coordinate being processed and the source containers corresponding to a scalar input element. To enable these operations to work immediately (with a null `arbb_function_t` argument), use the following additional interfaces:

```
/// @defgroup arbb_virtual_machine_immediate_map Immediate map operation support functions
/// @{
/// Pushes a new coordinate on the map stack with values <tt>[x, y, z]</tt>
/// representing the coordinate along the first, second and
/// third dimension respectively.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if context is a null object.
ARBB_VM_EXPORT
arbb_error_t arbb_push_map_coord(arbb_context_t context,
                                 uint64_t x,
                                 uint64_t y,
                                 uint64_t z,
                                 arbb_error_details_t* details);
/// Pops the current coordinate from the map coordinate stack.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if context is a null object.
///  - ::arbb_error_scoping if the map stack is currently empty.
ARBB_VM_EXPORT
arbb_error_t arbb_pop_map_coord(arbb_context_t context,
                                arbb_error_details_t* details);
/// Associates the scalar variable @p scalar with the container
/// @p source_container. Immediate invocations of ::arbb_op_get_neighbor operations
/// over the provided scalar will fetch data from the provided container using the
/// current map coordinate.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p scalar is a null object.
///  - ::arbb_error_invalid_argument if @p source_container is a null object.
///  - ::arbb_error_invalid_argument if @p source_container is not a valid dense container.
ARBB_VM_EXPORT
arbb_error_t arbb_set_scalar_source_element(arbb_context_t context,
                                            arbb_global_variable_t scalar,
                                            arbb_global_variable_t source_container,
                                            arbb_error_details_t* details);
/// Sets a flag to copy the source container of @p scalar that was
/// previously associated using @ref arbb_set_scalar_source_element
/// the next time when it is used as an input to ::arbb_op_copy in
/// immediate mode.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p scalar is a null object.
///  - ::arbb_error_invalid_argument if @p scalar is not a valid scalar.
ARBB_VM_EXPORT
arbb_error_t
arbb_set_copy_scalar_source_element(arbb_context_t context,
                                    arbb_variable_t scalar,
                                    arbb_error_details_t* out_error_details);
/// @}
```

Auxiliary Operations

Function Stack

To enable clients of the VM API to have a single place to store a stack of functions being defined, the API provides a simple function stack. The VM itself does not ever use this function stack because all VM operations related to functions explicitly specify which function to use.

The following functions support the stack:

```
/// @defgroup virtual_machine_function_stack Auxiliary operations - function stack
/// @{
/// Pushes the given function onto the function stack. The @p function
/// parameter may be a null object.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
ARBB_VM_EXPORT
arbb_error_t arbb_push_function(arbb_context_t context,
                                arbb_function_t function,
                                arbb_error_details_t* details);
/// Pops the top function from the function stack.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_scoping if the function stack is empty.
ARBB_VM_EXPORT
arbb_error_t arbb_pop_function(arbb_context_t context,
                               arbb_error_details_t* details);
/// Retrieves the top function from the function stack, placing it in
/// @p *out_function. If the function stack is empty, @p *out_function
/// will be set to a null object.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_function is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_top_function(arbb_context_t context,
                               arbb_function_t* out_function,
                               arbb_error_details_t* details);
/// @}
```

Stack of Variable Names

To enable clients of the VM API to have a single place to store a stack of pointers to variable names, the API provides a simple stack. The VM itself does not ever use this stack because all VM operations that create variables explicitly specify the variable names.

The following functions support the stack:

```
/// @defgroup arbb_virtual_machine_variable_stack Auxiliary Operations - Stack of Variables   /// @{

/// Pushes the given variable name onto the function stack. The @p variable_name
/// parameter may be a null object. This function does not create a string copy
/// of @p variable_name.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
ARBB_VM_EXPORT
arbb_error_t arbb_push_variable_name(arbb_context_t context,
                                     const char* variable_name,
                                     arbb_error_details_t* details);

/// Pops the top variable name from the variable name stack.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_scoping if the variable name stack is empty.
ARBB_VM_EXPORT
arbb_error_t arbb_pop_variable_name(arbb_context_t context,
                                    arbb_error_details_t* details);

/// Retrieves the top variable name from the stack of variable names, placing it
/// in @p *out_variable_name . If the stack is empty, @p *out_variable_name
/// is set to a null object.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_variable_name is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_top_variable_name(arbb_context_t context,
                                    char** out_variable_name,
                                    arbb_error_details_t* details);

/// @}
```

C/C++ Stack Traces

The run-time API provides a set of C API functions to record C/C++ stack traces and to query properties, such as filename and line numbers, from individual frames. A stack trace is a data structure that represents the active functions invoked to reach a certain execution state. A stack trace consists of stack frames. Each stack frame represents a single function. The order of frames in a callstack represents the calling hierarchy of a program at a given point. For example, to enable debugging or profiling by C/C++ based front ends, you can use these functions to record the stack traces during insertion of VM operations.

The following functions for a C/C++ stack trace are available:

```
/// @defgroup arbb_virtual_machine_stack_trace C/C++ Stack Traces
/// @{

/// The set of frame properties that can be queried.
/// @see arbb_cxx_get_frame_property
typedef enum {
  arbb_function_name, ///< The name of the function in the frame. C++ identifiers are left in mangled
form. The result of arbb_cxx_get_frame_property is of type <tt>const char*</tt>.
  arbb_line_number, ///< The line number corresponding to the next function call in the stack frame.
The result of arbb_cxx_get_frame_property is of type <tt>const unsigned int*</tt>.
  arbb_file_name ///< The name of the file in which the function was defined. The result
arbb_cxx_get_frame_property is of type <tt>const char*</tt>.
} arbb_cxx_frame_property_t;

/// Returns an arbb_cxx_stack_trace_t object that contains debug information about
/// all frames in the C/C++ stack trace starting with the frame
/// that called arbb_cxx_store_stack_trace. If a stack trace could not be
/// constructed, e.g., due to missing or incomplete debug information,
/// @p out_stack_trace is a null object.
///
/// @see arbb_cxx_stack_trace_t, arbb_cxx_is_stack_trace_null
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out_stack_trace is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_cxx_store_stack_trace(arbb_context_t context,
                                        arbb_cxx_stack_trace_t* out_stack_trace,
                                        arbb_error_details_t* details);

/// Releases all resources belonging to @p stack_trace. If @p stack_trace is
/// a null object, this function has no effect.
///
ARBB_VM_EXPORT
void arbb_cxx_release_stack_trace(arbb_cxx_stack_trace_t stack_trace);

/// Returns the number of stack frames in @p stack_trace.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p stack_trace is a null object.
///  - ::arbb_error_invalid_argument if @p out_num_frames is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_cxx_get_frame_count(arbb_context_t context,
                                      arbb_cxx_stack_trace_t stack_trace,
                                      unsigned int* out_num_frames,
                                      arbb_error_details_t* details);

/// Returns the stack frame at position @p frame_num in @p stack_trace.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p stack_trace is a null object.
///  - ::arbb_error_invalid_argument if @p frame_num is larger than or equal to the number of stack
frames in @p stack_trace.
///  - ::arbb_error_invalid_argument if @p out_frame is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_cxx_get_frame(arbb_context_t context,
                                arbb_cxx_stack_trace_t stack_trace,
                                unsigned int frame_num,
                                arbb_cxx_frame_t* out_frame,
                                arbb_error_details_t* details);

/// Sets the pointer @p out to the frame property specified by @p type. The data
/// that @p out points to after this function is executed is
/// safely castable to the data type that corresponds to the @p type parameter
/// and is valid until the @ref arbb_cxx_stack_trace that contains @p frame
/// is released with @ref arbb_cxx_release_stack_trace. See
/// @ref arbb_cxx_frame_property_t for the supported query types.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p out is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_cxx_get_frame_property(arbb_context_t context,
                                         arbb_cxx_frame_t frame,
                                         arbb_cxx_frame_property_t type,
                                         void** out,
                                         arbb_error_details_t* details);

/// Sets @p out_name to the demangled name of the C/C++ function
/// pointed to by @p function. When using this function, you must call
/// @ref arbb_free_string on the returned string object
/// once it is no longer required.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p function is null, or @p out_name is null
ARBB_VM_EXPORT
arbb_error_t arbb_cxx_resolve_function_name(arbb_context_t context,
                                            void* function,
                                            arbb_string_t* out_name,
                                            arbb_error_details_t* details);

/// Sets @p out_name to the demangled name of the C/C++ variable
/// pointed to by @p variable that must be accessible from @p frame.
/// Member variables are named as when accessing them using the "."
/// operator. For example, a member variable named "bar" in a
/// structure named "foo" is named "foo.bar". If @p class_instance_name
/// is non-null, the name of the outermost object that contains
/// @p variable is replaced with the string @p class_instance_name.
/// When using this function, call @ref arbb_free_string on the
/// returned string object once it is no longer required. If debug
/// information is not available, the function succeeds and @p out_name
/// is set to null.
///
/// @return An error code depending on the result of the operation:
/// - ::arbb_error_none if the operation succeeded.
/// - ::arbb_error_invalid_argument if @p context is a null object.
/// - ::arbb_error_invalid_argument if @p variable is null, or @p out_name is null
ARBB_VM_EXPORT
arbb_error_t arbb_cxx_resolve_variable_name(arbb_context_t context,
                                            void* variable,
                                            arbb_cxx_frame_t frame,
                                            const char* class_instance_name,
                                            arbb_string_t* out_name,
                                            arbb_error_details_t* details);

/// Fills the array @p out_names with the arbb_string_t name of the
/// corresponding C/C++ parameter of the function pointed to by @p function.
/// Use @ref arbb_get_c_string to access individual names.  When using this
/// function, call @ref arbb_free_string on each string returned
/// once it is no longer required.  If debug information is not available,
/// the function succeeds, but the elements of @p out_names are set
/// to null.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p context is a null object.
///  - ::arbb_error_invalid_argument if @p function is null.
///  - ::arbb_error_invalid_argument if @p num_args does not match the actual number of arguments for
@p function.
///  - ::arbb_error_invalid_argument if @p out_names is null
ARBB_VM_EXPORT
arbb_error_t arbb_cxx_resolve_function_parameter_names(arbb_context_t context,
                                                       void* function,
                                                       unsigned int num_params,
                                                       arbb_string_t* out_names,
                                                       arbb_error_details_t* details);

/// Registers a callback function that the VM should use to permit the front
/// end to extract the file name and line number information from the attribute
/// associated with <tt>arbb_op</tt>. If a callback function was previously
/// registered with the same key, the new callback function replaces the
/// previous callback function.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_invalid_argument if @p callback is a null pointer.
ARBB_VM_EXPORT
arbb_error_t arbb_register_source_info_provider(arbb_context_t context,
                                                arbb_attribute_key_t key,
                                                arbb_source_info_provider_t callback,
                                                arbb_error_details_t* details);

/// @}
```

Properties

The Intel ArBB VM provides the following C API functions to modify run-time properties that affect compilation and execution of functions. The table below lists properties that can be modified and VM API functions that modify them:

Property	Description	Default Value	Function
Generation of threaded code	Controls whether the VM generates threaded code during function compilation. The value of this property can be as follows: 0 — VM generates serial code 1 — VM can generate threaded code	1	`arbb_set_generate_threaded_code`
Number of worker threads	Sets the number of threads used when executing kernels if generation of threaded code is enabled.	The number of physical cores on the device where kernels are executed	`arbb_set_num_threads`
Decomposition degree	Sets the number of tasks spawned per core during parallel execution.	The number of available software threads	`arbb_set_decomp_degree`
Heap size	Sets the initial and maximum size of the heap used by the VM (in bytes).	• Initial: 134217728 (128MB) • Maximum (32-bit): 536870912 (512MB) • Maximum (64-bit): 1073741824 (1GB)	`arbb_set_heap_size`

The following are the declarations of those functions:

```
/// @defgroup arbb_properties Run-time Properties
/// @{

/// Enables or disables the creation of threaded code.
/// If @p value is not zero, the VM will generate threaded code during function
/// compilation.  See @ref arbb_virtual_machine_functions.  If @p value is zero,
/// the VM will only generate serial code.  This operation is only valid before
/// the first call to arbb_begin_function.
///
/// @return An error code depending on the result of the operation:
```

```
///  - ::arbb_error_none if the operation succeeded
///  - ::arbb_error_scoping if a function has already been compiled.
ARBB_VM_EXPORT
arbb_error_t arbb_set_generate_threaded_code(arbb_context_t context,
                                             int value,
                                             arbb_error_details_t* details);

/// Sets the number of threads used when executing kernels while threaded code
/// generation is enabled. Defaults to a number equal to the number of
/// physical cores present in the device upon which the kernel is being
/// executed.
///
/// Specifying the special value 0 for @p num_cores restores the number of
/// threads back to the default.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded
ARBB_VM_EXPORT
arbb_error_t arbb_set_num_threads(arbb_context_t context,
                                  unsigned int num_cores,
                                  arbb_error_details_t* details);

/// Sets the number of tasks spawned per core during parallel execution.
/// A larger number may help with dynamic load balancing across threads, but may
/// result in an eventually noticeable overhead.
///
/// By default the decomposition degree is the same as the number of available
/// software threads. If hyper-threading is enabled, the decomposition degree
/// will be multiplied by 2 internally. On the host, if you set
/// @p decomp_degree_multiplier to 0, it behaves the same as the default case
/// and any other value will be multiplied by the number of available software
/// threads.
///
/// Changing the decomposition degree impacts concurrent task execution. Optimal
/// settings are target and workload dependent.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded
ARBB_VM_EXPORT
arbb_error_t arbb_set_decomp_degree(arbb_context_t context,
                                    unsigned int decomp_degree_multiplier,
                                    arbb_error_details_t* details);

/// Sets the initial and maximum heap size for heap memory management.
/// The setting of heap memory management can only be done before instatiating
/// any VM variables or compiling a VM function. That is, this API can only be
/// called before any VM API that requires an arbb_context_t variable.
///
/// @return An error code depending on the result of the operation:
///  - ::arbb_error_none if the operation succeeded.
///  - ::arbb_error_scoping if this function is invoked after heap setup.
ARBB_VM_EXPORT
arbb_error_t arbb_set_heap_size(uint64_t init_heap_size,
                                uint64_t max_heap_size,
                                arbb_error_details_t* details);
/// @}
```

Final API Portions

The following code completes the start of the C API declarations:

```
/// @}

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif // ARBB_VM_API_H
```

Data Exposure for Debugging

To expose variable data within the Microsoft Visual Studio* debugger or GNU debugger (GDB), VM callers need a structure (within their scalar and container variables) that receives a pointer to the underlying data and a description of the format of the data. The VM initializes this structure when VM variables are created (using `arbb_create_global` and `arbb_create_constant`) and keeps it updated when `arbb_op` and `arbb_op_dynamic` are called if the address of the structure is passed to these calls. The description of the debug data has the following format:

```
typedef struct {
  int num_cols;
  int num_rows;
  int num_pages;
  int col_pitch;
  int row_pitch;
  int page_pitch;
  void* data_pointer;
} debug_data_description;
```

The content of the data and the validity of the other fields depend on the type of data being represented.

For scalars, `num_cols`, `num_rows`, and `num_pages` are set to 1, and `col_pitch`, `row_pitch`, and `page_pitch` are zero, and `data_pointer` points to a buffer containing the scalar value. You can retrieve the value by casting `data_pointer` as a pointer to the underlying data type corresponding to the scalar and dereferencing the pointer.

For containers, the members of the structure describe the layout and dimensionality of the data. For one-dimensional or two-dimensional containers, `num_rows` and `num_cols` are set to one, as needed to indicate that the corresponding dimension is not used. The pitch values indicate the number of bytes between columns, rows, and pages in each dimension.

When data has not yet been allocated for a container, `data_pointer` is NULL.

The `arbb_create_global` and `arbb_create_constant` functions require a pointer to the `debug_data_description` structure and initialize the structure to describe the variable that has been created. Debug data is not tracked for local variables. Therefore, there is no corresponding parameter for `arbb_create_local`.

The `arbb_op` and `arbb_op_dynamic` functions have a parameter that receives an array of void pointers to debug information. You can set the value of this parameter to zero if the caller does not require debug data to be maintained (for instance, if the operation will never be executed in the immediate mode). Otherwise, set the elements of this array to point to the `debug_data_description` structure corresponding to the variables in the output parameters.

For details of the use of the debuggers with Intel ArBB, see the *Intel ArBB User's Guide* for the respective OS.

Guidelines for Front Ends

A

elif translation

Some languages, such as Python*, have `elif` statements that execute a portion of an `if` statement only if no preceding portions are executed and a given clause is satisfied. When translating into `if-else` statements of the Intel® Array Building Blocks Virtual Machine, translate `elif` into nested `if-else` statements and enclose the evaluation of the condition expression in `elif` in the corresponding `else` body.

Example

The following is `elif` pseudocode:

```
if (_a < _b) {
  // code A...
} elif (_a > _b) {
  // code B...
} else {
  // code C...
}
```

Translate this pseudocode as follows:

```
_t1 = less<boolean>(_a, _b);
if (_t1) {
  // code A...
} else {
  _t2 = greater<boolean>(_a, _b);
  if (_t2) {
    // code B...
  } else {
    // code C...
  }
}
```

Textual IR Grammar Definition

B

Textual Intermediate Representation (IR) of the Intel® Array Building Blocks Virtual Machine can be defined using the following grammar. It puts together the contents of all the Grammar sections in this specification. The character set is ASCII.

```
module          ::= module-item-seq
module-item-seq ::= module-item | module-item module-item-seq
module-item     ::= function-defn | function-decl | global-decl
                  | meta-definition
```

```
identifier           ::= named-identifier | anonymous-identifier
named-identifier     ::= '_[a-zA-Z][_a-zA-Z0-9]*'
anonymous-identifier ::= '_[1-9][0-9]*' | '_0'
```

```
type                     ::= primitive-type | composite-type
primitive-type           ::= int-type | float-type | '$boolean' | size-type
int-type                 ::= '$i8' | '$i16' | '$i32' | '$i64' | '$u8' | '$u16'
                           | '$u32' | '$u64'
float-type               ::= '$f32' | '$f64'
size-type                ::= '$usize' | '$isize'
composite-type           ::= dense-type | nested-type
dense-type               ::= 'dense' '<' composable-type dimensionality-
specifier[opt] '>'
dimensionality-specifier ::= ',' dimensionality
dimensionality           ::= '1' | '2' | '3'
nested-type              ::= 'nested' '<' composable-type '>'
composable-type          ::= primitive-type
```

```
constant           ::= float-constant | int-constant | boolean-constant | size-constant
float-constant     ::= float-type '(' float-literal ')'
int-constant       ::= int-type '(' int-literal ')'
boolean-constant   ::= '$boolean' '(' boolean-literal ')'
size-constant      ::= size-type '(' int-literal ')'
```

```
float-literal     ::= hex-int-literal | dec-float-literal
dec-float-literal ::= '-?[0-9]*.[0-9]*(e-?(0|[1-9][0-9]*))?'
int-literal       ::= dec-int-literal | hex-int-literal
dec-int-literal   ::= '-?[1-9][0-9]*' | '0'
hex-int-literal   ::= '0x[0-9a-fA-F][0-9a-fA-F]*'
boolean-literal   ::= 'true' | 'false'


global-decl ::= 'global' type identifier ';'


function-decl       ::= function-declarator ';'
function-defn       ::= function-declarator metadata[opt]
                        function-body
function-declarator ::= 'function' identifier '(' param-seq[opt] ')'
function-body       ::= '{' stmt-seq[opt] '}'
param-seq           ::= param | param ',' param-seq
param               ::= input-param | output-param
input-param         ::= 'in' type identifier
output-param        ::= 'out' type identifier'


decl-stmt ::= 'local' type identifier attributes[opt] ';'


stmt-seq ::= stmt | stmt stmt-seq
stmt     ::= simple-stmt | facility-stmt | reduce-stmt | cflow-stmt
           | decl-stmt | member-stmt | functor-stmt
           | ...


simple-stmt    ::= arg '=' simple-stmt-id '<' type '>' '(' arg-
seq ')'attributes[opt] ';'
simple-stmt-id ::= ewise-stmt-id | scan-stmt-id | reorder-stmt-id


ewise-stmt-id ::= 'abs' | 'acos' | 'asin' | 'atan' | 'ceil'
                | 'copy' | 'cos' | 'cosh' | 'exp' | 'exp10'
                | 'floor' | 'ln' | 'log10' | 'log_not'
                | 'bit_not' | 'rcp' | 'round' | 'rsqrt'
                | 'sin' | 'sinh' | 'sqrt' | 'tan' | 'tanh'
                | 'neg' | 'add' | 'bit_and' | 'atan2'
                | 'compare' | 'div' | 'equal'
                | 'geq' | 'gradate' | 'greater' | 'bit_or'
                | 'leq' | 'less' | 'log_and' | 'log_or' |
                | 'lsh' | 'max' | 'min' | 'mod' | 'mul'
                | 'neq' | 'pow' | 'rsh' | 'sub'
                | 'bit_xor' | 'select'


functor-stmt ::= 'call' identifier '(' arg-seq[opt] ')' ';'
               | 'map' identifier '(' arg-seq[opt] ')' ';'


reorder-stmt-id ::= 'gather' | 'scatter' | 'pack' | 'unpack'
                  | 'shuffle' | 'unshuffle' | 'repeat' | 'distribute'
                  | 'repeat_row' | 'repeat_col' | 'repeat_page' | 'transpose'
                  | 'swap_row' | ' swap_col' | 'swap_page'
                  | 'shift_constant' | 'shift_clamp'  | 'shift_constant_reverse'
                  | 'shift_clamp_reverse' | 'rotate' | 'rotate_reverse' | 'reverse'


facility-stmt ::= arg-seq '=' facility-stmt-id '(' arg-seq ')'attributes[opt] ';'
facility-stmt-id ::= 'const_vector' | 'bitwise_cast' | 'cast' | 'cat' | 'extract'
                   | 'replace' | 'replace_row' | 'replace_col' | 'replace_page'
                   | 'replace_element' | 'section' | 'index'
                   | 'extract_row' | 'extract_col' | 'extract_page'
                   | 'get_elt_coord' | 'get_neighbor' | 'expect_size'
                   | 'mask' | 'sort' | 'sort_rank | 'alloc'
                   | 'length' | 'get_nrows' | 'get_ncols' | 'get_npages'
                   | 'set_regular_nesting' | 'apply_nesting' | 'copy_nesting'
                   | 'get_nesting' | 'split' | 'unsplit'
                   | 'flatten' | 'segment' | 'replace_segment'


reduce-stmt    ::= arg-seq '=' reduce-stmt-id '<' type '>' '(' arg-seq
')'attributes[opt] ';'
reduce-stmt-id ::= 'add_reduce' | 'mul_reduce ' | 'max_reduce' | 'max_reduce_loc'
                 | 'min_reduce' | 'min_reduce_loc'
                 | 'and_reduce' | 'ior_reduce' | 'xor_reduce'
```

```
scan-stmt-id ::= 'add_scan' | 'mul_scan | 'max_scan' | 'min_scan'
               | 'and_scan' | 'ior_scan' | 'xor_scan'


merge-stmt-id ::= 'add_merge' | 'add_merge_scalar'


cflow-stmt ::= if-stmt | loop-stmt | break-stmt | continue-stmt
             | return-stmt


if-stmt   ::= 'if' '(' arg ')' '{' stmt-seq[opt] '}'
            else-stmt[opt]
else-stmt ::= 'else' '{' stmt-seq[opt] '}'


loop-stmt ::= while-stmt | do-stmt | for-stmt
loop-block-seq ::= loop-block | loop-block loop-block-seq
loop-block ::= init-block | cond-block | step-block | body-block
init-block ::= 'init' '{' stmt-seq[opt] '}'
cond-block ::= 'cond' '{' stmt-seq[opt] when-stmt '}'
step-block ::= 'step' '{' stmt-seq[opt] '}'
body-block ::= 'body' '{' stmt-seq[opt] '}'


for-stmt ::= 'for' loop-block-seq


while-stmt ::= 'while' loop-block-seq


do-stmt ::= 'do' loop-block-seq


break-stmt ::= 'break' attributes[opt] ';'


continue-stmt ::= 'continue' attributes[opt] ';'
```

```
return-stmt ::= 'return' attributes[opt] ';'


when-stmt ::= 'when' '(' arg ')' attributes[opt] ';'


attributes       ::= '[[' attr-seq[opt] ']]'
attr-seq         ::= attr | attr ',' attr-seq
attr             ::= attr-keyword '(' attr-arg ')'
attr-keyword     ::= 'debug_info' | 'align' | attr-identifier
attr-identifier ::= '_' string-literal
attr-arg         ::= attr-value | attr-id
attr-value       ::= int-literal | string-literal | void-pointer
attr-id          ::= '![a-zA-Z][0-9a-zA-Z_]*
attr-definition ::= 'attribute' attr-id '=' attr-value
string-literal  ::= '".*"'
```

Index