Intel(R) Array Building Blocks for Linux* OS

User's Guide

Document Number: 324171-002US

Legal Information

Legal Information

Contents

What's New

The User's Guide provides a quick guide for the Intel® Array Building Blocks software for all supported architectures on the Linux* OS.

This User's Guide documents Intel® Array Building Blocks 1.0 Beta 1 release.

Getting Support

Intel provides a support web site that contains a rich repository of self-help information, including getting started tips, known product issues, product errata, license information, and more (visit http://www.intel.com/software/products/support/).

Registering your product entitles you to one-year technical support and product updates through Intel(R) Premier Support. Intel Premier Support is an interactive issue management and communication web site providing the following services:

- Submit issues and review their status.
- Download product updates.

To register your product, or contact Intel, or seek product support, visit

http://www.intel.com/software/products/support/

Intel® Array Building Blocks software provides a product web site that offers timely and comprehensive product information, including product features, white papers, and technical articles. For the latest information, see http://www.intel.com/software/products/.

Notational Conventions

The document uses the following font conventions and symbols:

Table 1-1 Notational conventions

Italic	*Italic* is used for emphasis and also indicates document names in body text, for example: see *Intel© Array Building Blocks Application Programming Interface Reference Manual*
`Monospace lowercase`	Indicates filenames, directory names, and paths, for example: `\tools, build_run.bat`
`UPPERCASE MONOSPACE`	Indicates system variables, for example, `JIT_OPTIONS`
`Monospace italic`	Indicates a parameter in discussions, such as function parameters, for example, *`lda`*; makefile parameters, for example, *`functions_list`*; and so forth. When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value: <*`installation directory`*>.
[items]	Square brackets indicate that the items enclosed in brackets are optional.
{ item | item }	Braces indicate that only one of the items listed between braces can be selected. A vertical bar (|) separates the items

Displaying Pseudocode

In this guide the types and operators are provided using C++ syntax, but the semantics of these operators are illustrated using pseudo-code like conventions. In particular, the curly brackets and ordered values, which are separated by commas, denote both the extent and the nesting of a vector, as follows:

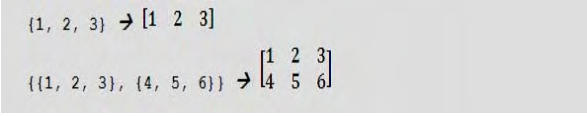
```
{a, b, c, ...}, {{1, 1} {} {0, 1, 2}}
```

Pseudocode is printed on the grey background (above). The syntax in such cases may disobey C/C++ syntactic conventions and specifications.

Actual code or prototypes are printed on no background as follows:

```
i32 aVariable = 2;
```

Matrix notation is occasionally used in explaining the structure of a vector. For example:



Getting Started with Intel® Array Building Blocks Software

1

This chapter provides basic information about Intel® Array Building Blocks and how to confirm installation of the product.

Intel® Array Building Blocks Basics

The Intel® Array Building Blocks (Intel® ArBB) software is a data-parallel programming environment designed to effectively utilize the power of existing and upcoming throughput-oriented features on modern processor architectures, including Intel's multi-core and many-core platforms. Intel ArBB provides a C++ library interface that enables you to continue to write code using standard C++, and works with tools such as standards-conformant C++ compilers

Intel® Array Building Blocks Usage

You can focus your use of Intel® Array Building Blocks (Intel® ArBB) software on those portions of your C/C++ code that have the most parallelism potential. Intel ArBB supports compos ability through dynamic composition and optimization of its operators. For example, you can spread the Intel ArBB code across nested functions that span multiple libraries. Intel ArBB provides a safer parallel programming environment by isolating its data objects from the rest of the C/C++ program and by restricting conflicting concurrent accesses to shared objects through structured primitives. This eliminates the need for low-level constructs like locks and prevents data races.

The Intel ArBB can be described at several levels of abstraction:

Programming model

The Intel ArBB software allows you to express data parallelism with sequential semantics by expressing operations at the aggregate data collection level. The model provides for data isolation between primitive and function invocations to ensure race-free parallel execution. Execution of programs written using Intel ArBB are deterministic (subject to the limited precision of computer arithmetic).

Language

The Intel ArBB software mimics C/C++'s control flow as well as adding new types and operators. While these features give the developer an expressive programming environment, these additions are supported solely through the use of header files and a runtime library using only standard C++ capabilities. Because of this, Intel ArBB usage co-exists with C/C++ code and only specific portions of your program need be rewritten to utilize parallel features of the underlying system in the Intel ArBB software.

Application Programming Interface

The Intel ArBB software introduces new types using C++'s type system. It introduces new operators using operator overloading and library calls. You can use standard C++ compilers to compile Intel ArBB programs. The entry points to data parallel functions are `call` and `map` functions, which express semantics as aggregate (vector-oriented) and element-wise (scalar) operations, respectively, on data collections.

High Performance Virtual Machine

The Intel ArBB high-level interfaces and its dynamic compiler enable improving performance without having to consider the details of the underlying system. The Intel ArBB virtual machine transforms high-level code into optimized and parallelized machine code for the target architecture. From the high-level code point of view, the degree of parallelism supported with threads, vectors, the ISA, the memory model, and the cache sizes are all hidden, making code more portable and easier to write. The dynamic compiler retargets Intel ArBB functions for the target architecture and caches it for reuse (to avoid further compilation) on subsequent invocations.

Data Types

Intel® Array Building Blocks software provides a rich set of data types for representing your data *collections* (aggregations of data, like images, matrices, and arrays). Fundamental elements of these data types are the dense and nested containers, which are collections to which data-parallel operators may be applied.

Safety

Intel® Array Building Blocks (Intel® ArBB) provides safety through isolation and immutable types. Since C/C++ enables unsafe, concurrent write accesses to its objects, Intel ArBB objects are not accessible via C/C++ pointers. Intel ArBB software operates in a data space that is isolated from the native (C/C++) data space. Explicit operators are required to move data between native types and vectors. Intel ArBB operators and types only affect the Intel ArBB data space. This isolation enables allocating vectors from a managed heap. Data races within Intel ArBB code are prevented through unchangeable types. The semantics of Intel ArBB operators are purely functional: vector objects are passed by value, and each Intel ArBB operator logically returns a new vector object. This property ensures the safety of parallelism and enables the compiler to aggressively optimize the code.

Determinism

Intel® Array Building Blocks software implements a deterministic parallel programming model, meaning that the behavior of any program is the same on single-core platforms as it is on multi- or many-core platforms. However, results may change slightly because of the limited precision of computer arithmetic.

Some Definitions

Here are some definitions of terms that are used throughout the document.

Operator - a function that is provided by the Intel® Array Building Blocks (Intel® ArBB) API. Element-wise, collective, permutation operators operate on Intel ArBB types.

Function - a user-defined function, either invoked natively or with the special operators `call` or `map`.

Facility - a helper function that is provided by the Intel ArBB API. Vector facilities such as binding constructors and `copy_in/out` facilities operate exclusively on Intel ArBB types. System facilities do not necessarily operate on such types.

Generic prototype - generic form of operators and functions.

Native - refers to C/C++ types, C/C++ data and name space, and code compiled by the C/C++ compiler.

Steps to Get Started

This section describes the steps you must do to start with Intel® Array Building Blocks (Intel® ArBB).

Checking Your Installation

After installing the Intel® Array Building Blocks (Intel® ArBB) software, verify that the software is properly installed and configured.

Check that the directory you choose for the installation is created: *<installation path>*/arbb/*<release_type>*. By default, *<installation path>* is /opt/intel.

To check the high-level and detailed structure of the Intel ArBB software installation directory, see High-level Directory Structure.

Setting Environment Variables

You can use the shell script file set_env.sh (for bash), or set_env.csh (for csh) to set LIB, INCLUDE, LD_LIBRARY_PATH and CPATH environment variables for Intel® Array Building Blocks (Intel® ArBB) software on the specified architecture. To do this, run set_env.sh arg (set_env.csh arg).

arg specifies the architecture:

arg= ia32 for the IA-32 architecture;

arg= intel64 for the Intel 64 architecture.

You can set the level of optimization by setting the environment variable ARBB_OPT_LEVEL.

O0 enables no runtime optimizations, and uses interpretations. O2 enables vectorization. O3 enables vectorization and thread parallelization.

The default value is O2.

Writing Programs Using Intel® Array Building Blocks

This topic explains the basic steps way for writing a very simple Intel® Array Building Blocks (Intel® ArBB) program.

1. Include the Intel ArBB library header file arbb.hpp.

```
#include <arbb.hpp>
```

2. Add the following statement at the front of your source code:

```
using namespace arbb;
```

3. Move code into a Intel ArBB function by adding these lines before and after the code section;

```
void foo(dense<T> in, dense<T>& out) {
<code section>
}
```

4. Bind vectors so that they to take values from their C++ counterparts using the binding constructors:

```
dense<f32> in;
bind(in, inPtr, length);
dense<f32> out;
bind(out, outPtr, length);
```

5. Invoke the Intel ArBB function using a `call`:

```
call(foo)(in, out);
```

6. Ensure that the call has completed:

```
out.read_only_range();
```

Refer to the *Intel® Array Building Blocks Application Programming Interface Reference Manual* to learn more about developing the Intel ArBB applications. Also refer to the section "Porting C Code to Intel® Array Building Blocks" of this guide.

Intel® Array Building Blocks Package Structure

2

This chapter discusses the structure of the Intel® Array Building Blocks software after installation as well as the application libraries supplied.

High-Level Directory Structure

The following table shows the high-level directory structure of Intel® Array Building Blocks (Intel® ArBB) package after installation.

High-Level Directory Structure

Directory	Description
<*install_directory*>	installation product directory
/Documentation	Intel ArBB related documents
/include	header files
/lib	library files
/samples	sample applications
/tools	script files

The <*install_directory*> is the installation directory.

The `include` directory contains all header files that are needed for the Intel ArBB application development. Typically you must include `arbb.hpp` in your Intel ArBB application source code.

The `lib` directory contains all library files that the you need to link to. For example, `/ia32/libarbbd.so` - the dynamic library for the debug version on Linux* OS.

The `samples` directory contains domain-specific workloads developed using the Intel ArBB technology. Refer to the section Intel® Array Building Blocks Sample Code.

The `tools` directory contains different tools, for example, batch file to set environment variables.

The `Documentation` directory contains the Intel ArBB related documents (refer to the section "Contents of the Documentation Directory").

Contents of the Documentation Directory

The following table shows the content of the `/Documentation` directory in the Intel® Array Building Blocks (Intel® ArBB) software installation directory.

Contents of the `/Documentation` Directory

File name	Description	Notes
`arbb_documentation.htm`	Documentation index. Lists the principal Intel ArBB Technology documents with appropriate links to the documents	
`arbb_release_notes.pdf`	General overview of the product and information about this release	These file can be viewed prior to the product installation
`arbb_support.txt`	Information on package number for customer support reference	
`arbbEULA.txt`	Intel® ArBB license agreement.	

Intel® Array Building Blocks Sample Code

3

Sample applications and tutorials are available for Intel® Array Building Blocks (Intel® ArBB) technology. These can help you start your application development. The tutorials explain the basics of the Intel ArBB. The samples illustrate Intel ArBB implementations of specific kernels. You can use them for evaluating performance in particular domains.

Samples and tutorials offer direct methods to determine:

- Whether the software is working on your system
- How Intel ArBB language constructs (such as operators, functions, collectives, facilities and so on) relate to analogous C++ constructs
- What options are available for implementing Intel ArBB functions (such as large-vector arithmetic versus pre-element scalar arithmetic)
- How to optimize Intel ArBB functions to build performant and scalable applications (for instance, alignment requirements for memory allocation)
- How the performance and precision of Intel ArBB implementations compare to those of analogous C++ applications

Using Tutorials

This section provides general information on the Intel® Array Building Blocks tutorials and provides general instructions on how to use them.

About Tutorials

Tutorials provide step-by-step instructions on how to implement common algorithms, for example convolution or dot product, using features of Intel® Array Building Blocks.

Use this link at http://software.intel.com/en-us/articles/arbb-tutorial/ to browse through available tutorials and download the tutorials that you would like to work with further.

Building and Running Tutorials

After installing the Intel® Array Building Blocks (Intel® ArBB) product, use the following steps to build and run the tutorials:

1. Create a directory in which to install the tutorials. In the following this directory is referred as `<tutorials-dir>`.

   Download and extract the Intel ArBB tutorials to this directory.
2. Go to a tutorial directory such as `<tutorial-dir>/tutorials/tut0`.

```
cd ~/<tutorial-dir>/tutorials/tut0
```

3. Edit the Makefile to select a C++ compiler:

```
CXX=g++
```

4. Optionally override the default target architecture. For example, you can specify IA-32 on a 64-bit operating system. Valid values for the Intel architecture are `ia32` and `intel64`:

```
export ARBB_IA=ia32
```

5. Set the `ARBB_ROOT` environment variable to point to the root directory of the Intel ArBB installation, and rebuild the tutorial:

```
export ARBB_ROOT=<install-dir>
make clean
make
```

6. Set the `LD_LIBRARY_PATH` and run the tutorial:

```
LD_LIBRARY_PATH=$ARBB_ROOT/lib/$ARBB_IA ./tut0
```

As an alternative to step 4, use the following shell script to run a tutorial that is already compiled:

```
cd ~/<tutorial-dir>/tutorials/
./run.sh tut0
```

Using Sample Applications

This section provides general information on the Intel® Array Building Blocks sample applications and provides general instructions on how to use them.

About Sample Applications

Sample applications illustrate the Intel® Array Building Blocks (Intel® ArBB) implementation of kernels commonly used in specific areas, including financial services, graphics, image processing, medical imaging, seismic reconstruction, and others.

By default, samples are installed to the `<install-dir>/samples` directory**.** All samples are grouped into categories corresponding to their application areas. The `misc` category includes samples applicable to a variety of different areas. It also includes the `spec-samples` project that demonstrates the usage of individual Intel ArBB language constructs, complementing the *Intel® Array Building Blocks Application Programming Interface Reference Manual*.

Each sample compares the performance of serial and Intel ArBB implementations of a given kernel operation. The difference in execution time is measured and can be taken as a rough indication of the speedup that you can expect for comparable patterns of data transfer and computation. Each sample also includes validation to ensure that the serial baselines and parallel Intel ArBB implementations are producing similar results. For floating-point computations, the comparisons are made within a specified tolerance.

Building and Running Sample Applications

To build and run sample applications:

Go to the `<install-dir>/tools` directory and run one of the following shell scripts:

- `build_run-icc.sh` - automatically builds and runs the sample applications using the Intel® C++ Compiler
- `build_run-gcc.sh` - automatically builds and runs the sample applications using the GNU Compiler Collection (GCC)

You can inspect the shell scripts to determine how to build individual samples.

Understanding Sample Performance

Most samples run in a benchmarking mode that compares the performance of Intel® Array Building Blocks (Intel® ArBB) implementations against a reference serial implementation. For example, the output of the 3D-dilate sample looks like the following:

`Running 3D-dilate.exe`

Version	Time (s)	Speed Up
C	27.037554	1.000
ArBB1	1.249182	21.644

It is important to treat these numbers only as a rough indication of the performance benefits of Intel ArBB. The following information can help to interpret the relative speedup of Intel ArBB over the reference implementation:

Reference Implementation: The baseline is a serial implementation using 'C' that produces the same results as the Intel ArBB implementation. However:

- By default, the baseline uses a single core and does not explicitly take advantage of SIMD instruction sets.

  To tune the baseline implementation (for example, to isolate other performance advantages of the Intel ArBB implementation), use an auto-vectorizing compiler. For example, if we compile the 3D-dilate sample with the Intel compiler with appropriate options to autovectorize the baseline, we obtain the following:

  `Running 3D-dilate.exe`

Version	Time (s)	Speed Up
C	8.319997	1.000
ArBB1	1.146859	7.255

- Slight algorithmic variations can exist between the serial and Intel ArBB implementations. For example, the logic to handle boundary conditions can vary.
- Of course the speedup also varies with the processor generation and instruction set supported, number of cores, number of processors, size of the cache, speed of the memory system, operating system activity, and other factors.

Optimization Level: By default, samples run at the default optimization level of O2. At O2, the code emission is fully optimized, but Intel ArBB applications only use a single core. To use the sample applications as a parallelization benchmark, set the environment variable `ARBB_OPT_LEVEL=O3`. By default at O3, Intel ArBB applications are vectorized and use all available cores. Set the environment variable `ARBB_NUM_CORES` to a specific number less than the number of physical cores to reserve cores for other concurrent processing.

Workload: You can choose between the following pre-processor options when building samples. In all cases, the Intel ArBB and reference implementations use the same problem size and time the same number of iterations:

- `SAMPLE_DATA_SET`: By default, a single repetition is timed with a significant (that is non-trivial) problem size.
- `BIG_DATA_SET`: Multiple repetitions are timed with a significant (that is non-trivial) problem size.

  The reported times are averages that reduce the impact of operating system activity on the benchmark output.

- `PERF_REGRESSION` or `SMALL_DATA_SET`: Multiple repetitions are timed with a medium problem size.
  1. Failures related to the accuracy of single-precision computation occur less often at these problem sizes.
  2. Large variations in performance are typical when competing tasks such as operating system services are run concurrently at 03 level.
  3. This option provides a fast way to determine if a sample is working correctly

NOTE. In some cases where workloads are time-consuming at large problem sizes, a single repetition is always performed. Also, repetitions (necessary to get an accurate measurement of average performance) increase the wall-clock time to execute the operation.

Throughput: The workloads have not all been tuned to illustrate optimal throughput. For instance, there are cases where implementations include a large number of calls to Intel ArBB functions with small workloads. While these workloads can often be re-factored to improve throughput, the sample implementations as given do provide the opportunity to compare the static overheads of invoking Intel ArBB functions.

Sample Browser

The table below lists the available samples, with short descriptions. The column **Category** corresponds to the subfolder of the `<install-dir>/samples` directory where each `Sample` subfolder is located.

In addition, characterizations of each algorithm and the implementation of algorithms through Intel® Array Building Blocks (Intel® ArBB) language constructs are provided to help you find relevant samples:

- The column **Description** explains the sample operation using terms specific to industry verticals. For example, *option pricers* are used in computational finance to price options, derivatives and other financial instruments.
- The column **Algorithm** provides information about the parallel patterns (for example, stream or reduction) and the arithmetic intensity of the kernels (the relative balance of data transfer and computation). This information can assist you in finding Intel ArBB samples with similar characteristics as the bottlenecks in your applications.
- The column **Implementations** compares the variations of the Intel ArBB kernels provided in the each sample project. In some cases, when variations are present for a given algorithm, they illustrate techniques to optimize that algorithm for different problem sizes. In other cases, the variations just illustrate different strategies for implementing the same algorithms using Intel ArBB.

  For example, many algorithms can be implemented using either **`map`** or **`call`** operators as it is possible to express the math using either scalar *elemental* functions or large-vector arithmetic. Large-vector arithmetic is implemented in the body of an Intel ArBB function using operations on dense and nested containers. The same algorithm can usually be implemented using element-wise arithmetic using Intel ArBB scalars, then replicating this function over a collection using a map.

NOTE. The terms in **bold text** refer here to Intel ArBB language constructs.

Sample	Description	Algorithm	Implementation
		Category: finance	
binominal-tree	Numerical lattice for pricing European options.	Stream of option pricing evaluations with high arithmetic intensity (**exp**, **sqrt**).	(1) A **map** to parallelize over pricing multiple options. Uses a series of **_for** loops for each time step and calls **replace()** on elements of local (temporary) containers as well as containers for output of the option prices.
black-scholes	Analytical method for pricing European options. Optionally evaluates or approximates polynomials.	Data-parallel random number generation using **scan**. Option stream with arithmetic intensity (**ln**, **exp**, **sqrt**).	(1) Uses the **call** operator to inline an Intel ArBB function whose outer loop parallelizes over options. Illustrates the use of **select** to choose between two terms during polynomial evaluation.
monte-carlo	Stochastic method for computing financial options using the Blackscholes formula given randomly varying prices. Can optionally generate the sequence of random numbers using a multiplicative congruential generator (MCG).	Data-parallel random number generation using **scan**. Option stream with arithmetic intensity (**exp**). 1D and 2D accumulation (reductions).	(1) Uses the **call** operator to invoke an Intel ArBB function whose outer loop parallelizes over options. Generates a normally distributed random sequence using a transformation of a uniform random sequence. Uses a nested **_for** loop over prices to perform 1D vector arithmetic, and uses **add_reduce** and **replace** to accumulate a result. (2) Uses **reshape** and **repeat_col** to perform an equivalent 2D implementation. Illustrates the use of **add_reduce** for accumulation.
poisson-solver	Monte-Carlo method to solve Poisson functions (MCP solver). Uses a sequence of random numbers from a linear congruential generator (LCG).	Data-parallel random number generation using **scan**. Kernel with nested loops and high arithmetic intensity (**sin**, **cos**). Minimum distance computation using a series of thresholds in the inner loop. Unbalanced load where the number of iterations depends on random input.	(1) Uses the **call** operator to generate a large vector of scalar random numbers. Followed by a **map** over points illustrating nested **_for** and **_while** loops for a random walk. The inner loop is a series of **_if** statements to compute a minimum distance. (2) Equivalent implementation using **map** to perform scalar arithmetic.

Sample	Description	Algorithm	Implementation
randomlib	Code that can be in-lined to generate a normally distributed random sequence using the following algorithms: • Linear Congruential Generator (LCG) • Multiplicative Congruential Generator (MCG) • Combined multiple recursive generator with two components of order 3 (MRG) • Generalized feedback shift register generator (R250) Mersenne twister (MT)	Data-parallel random number generation using **scan**. Scan collectives, bitwise operations	(MCG) Uses the **call** operator to invoke native code stubs from within an Intel ArBB function. The actual native implementation can be switched at link time. (General) Use of **mul_scan** to generate indices. Illustrates the use of **rotate** and **select** on seeds. Illustrates the use of a bitwise & operation (a mask) to simulate a vector modulus operation.
	Category: graphics		
raytracing1	A kernel used to create a realistic visualization of a scene when tracing rays from a camera through an image plane to a light source. For each pixel in a 2D array, the kernel determines the closest ray-triangle intersection and evaluates the pixel shade using a lighting calculation. This implementation is brute-force and does not use an	For each triangle in the scene, compute the intersection and distance to triangle. Compute the minimum distance and shade the triangle closest to the camera. A simplified lighting calculation is given by a proportional sum of diffuse, specular and ambient light. The ray tracing algorithm is parameterized over 1-component or 2-component inputs and outputs.	(1) Uses the **call** operator over a 2D pixel array. Within the **call** body, a **_for** loop over the height of the pixel array is performed. For each row, a **map** over 1D lines of pixels is performed. Illustrates the use of **index** to generate an arithmetic sequence and **replace_row** to populate rows of the 2D outputs. Uses **at** to extract a one-component **array** from an N-component **array**. (2) An equivalent **map** over a 2D pixel array is performed. Illustrates the use of **index** to generate a 2D sequence. **Note:** The bulk of the implementation differs only in the use of scalars versus

Sample	Description	Algorithm	Implementation
	accelerator: every ray is compared to every triangle.		2-tuples for positions and directions. Therefore, ray tracing is parameterized over different data types. (3) A variation on (2) using 3-component tuples and large vectors of 3-component tuples instead of separate variables (i.e. for RGB and XYZ). Illustrates the use of **get** and **set** for components of a tuple
raytracing2	A variation on raytracing1 where ray-triangle intersection is limited to triangles in grid cells that intersect with rays. In other words, a uniform spatial partition is used for acceleration.	A variation on raytracing1 where the triangles are indexed by a uniform grid accelerator to limit the number of triangle-ray tests that are required.. For each cell in a 3D grid, an initial test is performed to determine if the ray intersects the cell. Ray-triangle intersection is performed only when rays intersect grid cells.	(1-3) See (1), (2) and (3) for raytracing1. **NOTE.** Uses **_break** to perform an early exit from a **_for** loop or a **_while** loop.
	Category: image processing		
convolve	Convolution of a 2D image with a discrete Gaussian function.	A local gather over a fixed neighborhood around each pixel of a 2D image. 1D convolution along X and Y axis of a pre-computed 2D stencil of coefficients. Clamps output to 255 to prevent saturation of the 8-bit unsigned image data. Optionally runs with convolution stencils of 5x5 or 9x9 pixels.	(1) Uses the **call** operator to implement separable convolution using large vector math. Calls **shift** to perform vector arithmetic on neighbors. Optionally performs an averaging filter. (2) A variation on (1) with manual unrolling rather than **_for** loops for separable convolution. Only works with a 5x5 pixel convolution stencil. (3) A **map** operation using nested **_for** loops to perform convolution. Calls **num_rows** on the 2D stencil to operate on square kernels of any size. (4) This tuned version casts the unsigned image data to single-precision float. Next, a **_for** loop is performed over 64-pixel wide strips of the image. For each block, nested for loops are used to unroll the convolution. Uses **section** and**replace** to operate on a

Sample	Description	Algorithm	Implementation
			64-pixel wide strip of the image. This is followed by a similar loop to process the portion of the image that does not fit into strips of 64 pixels.
gauss-convolve	Convolution of a 2D image with a discrete Gaussian function.	Similar to convolve. Uses different stencil sizes and does not assume odd stencil sizes.	(1) Two **_for** loops to perform separable convolution using large vector math. Uses **shift_row** and **shift_col** for the 1D convolution along X and Y axis. (2) Equivalent implementation using **map** to perform scalar arithmetic. Illustrates in-lining of multiple C++ routines (one for each axis) into a single Intel ArBB function.
sobel	An edge detection filter for a 2D image that uses the gradient (rate of change) of image intensities.	A gather over a fixed neighborhood around each pixel of a 2D image. Separately computes the gradient along the X and Y axes. This variation on a Sobel filter outputs the largest of the two gradients (with clamping to avoid saturation of 8-bpp image data).	(1) Uses **call** to invoke an Intel ArBB function that in turn inlines separate functions to compute the gradient in X and Y using large vector arithmetic. Calls **shift** to perform vector arithmetic on neighbors. (2) Equivalent implementation using **map** to perform scalar arithmetic.
		Category: medical	
3D-dilate	A morphological operator for dilation applied to 3D grayscale images.	Loops over a neighborhood defined by a 3D binary mask (structuring element). For each neighbor corresponding to a non-zero mask entry, the image is updated with the largest difference between the neighbor and a height field matrix. Features are dilated when voxels neighboring the structuring element are incorporated (assigned similar intensities). The height matrix provides intensities for non-flat structuring elements.	(1) Three nested **_for** loops are used to iterate through the mask. A call to **create** makes a local buffer to store maximums. Calls **shift** to perform vector arithmetic on neighbors. Calls **num_cols**, **num_rows** and **num_pages** to operate on a mask of arbitrary size.
3D-Erode	A morphological operator for erosion applied to 3D grayscale images.	Similar to 3D-dilate, except that the smallest difference is output.	Similar to 3D-dilate. Uses **min_reduce**.

Sample	Description	Algorithm	Implementation
3D-gauss-convolve	Convolution of a 3D image with a discrete Gaussian function.	Similar to guass-convolve, except that a **3D** convolution stencil is applied to **3D** image data.	Similar to guass-convolve. Uses **shift_page** in addition to **shift_row** and **shift_col** to handle the Z axis.
back_projection	A technique for image reconstruction used with inputs from computed axial tomography (CAT) scans.	A spatially-coherent gather along projections (rays) through each pixel of a 2D image. Applies the inverse Radon transform to reconstruct a 2D image given a set of projections through that image. Uses 1D interpolation to update the output image with the contribution from the nearest projections. **NOTE.** A simple scan geometry is assumed (radically symmetric 1D orthographic projections rather than a helical scan). In addition, it is assumed that sharpening of sets of input projections (sinograms) has already been performed. **Category: misc**	(1) Uses the **call** operator to parallelize over pixels in the 2D output image. Uses a **_for** loop in the **call** body to iterate through projection angles. Uses a table lookup to compute the **sin** and **cos** of each projection angle. Calls **floor** and **ceiling** on large vectors prior to interpolation. Uses the **+=** operator to integrate contributions. (2) A variation on (1) that uses **reshape** to create a 2D product of angles and projections rather than a packed 1D vector. Within the **call** body, the indexing is modified to perform a 2D gather using a two-component index.
mandelbrot	Fractal data set generation.	Iteratively applies a quadratic polynomial map over complex numbers and computes it escape time to compute a fractal set.	(1) Uses a**_for** loop in a **map** operator to iteratively refine the output. Uses the complex numbers (using std::complex over Intel ArBB floating-point types) to perform complex multiplication and addition. Calls **abs** to compute the complex norm, and uses **_break** to exit early when the hard-coded bounds are exceeded. (2) An alternative implementation using a **_for** loop in a call operator. Creates a large vector that is local to the Intel ArBB function, complex and 2D. Performs a fixed number of iterations, and

Sample	Description	Algorithm	Implementation
			stops updating the output when the fractal bounds have been exceeded (this performs more work than the version (1) using an early exit).
spec-samples	Calling code that details the behavior of various Intel ArBB operations on dense and nested containers. The operations are divided into three categories: • Collectives used for reductions and scans. • Facilities for building and querying the structure of nested data containers. • Operations on dense and nested containers, permutation operations on elements or nested segments of containers.	(1) Full and partial collective operations are performed. The partial collectives reduce the dimensionality of the input set rather than returning a single value. Collective operation is illustrated using dense and nested containers. (2) Illustrates the reshaping of dense containers as nested containers, flattening of nested containers, and **split/unsplit/cat** operations. Also shows how to extract sizes of dense containers and nested segments. (...2) Illustrates the creation and initialization of large vectors and index sets. Also shows how to section large vectors and update sections of large vectors. (3) Permutes data using swizzle, pack, shift, rotate, sort and shuffle operations. Many of these operations have inverses, such as pack/unpack.	(1) The calling code, inputs and outputs are detailed for full/partial reductions using **add_reduce,** as well as exclusive and inclusive scans (**add_scan** and **add_iscan**). (2) Uses **reshape_nested_lengths** to generate nested vectors from dense vectors based on segment descriptors. Couples this operation with a type cast using **reshape_as**. Calls **split**, **unsplit** and **cat** with inputs and/or outputs that are nested containers. (...2) Calls **value**, **lengths**, **flags** and **offsets** to extract information about nested containers. (...2) Calls **create** for large vectors and illustrates the construction of index sets. Uses section and replace to operate on pieces of large vectors. (3) Performs swizzle, mask, pack/unpack and scatter operations on large vectors using large vectors to specify the output indices. (...3) Calls **shift**, **shift_sticky** and **rotate** with options to permute **dense** and **nested** containers both left and right. Note that full segments of nested containers can be permuted. (...3) Calls **sort** to perform direct and indirect sorts on dense containers. (...3) Calls **shuffle**/**unshuffle** to perform strided interleave/de-interleave of dense containers.

Sample	Description	Algorithm	Implementation
			(...3) Shows how to use repeat and **repeat_row** variants to replicate data in dense containers.
		Category: seismic	
3dstencil	Convolution used in reverse time migration (RTM).	Convolution using a 7x7x7 cross-shaped kernel.	(1) Uses the **map** operator to perform scalar arithmetic. Uses relative indices to gather values of neighbors.
convolution	1D and 2D convolution for a seismic image.	Separable 2D convolution using a cross-shaped kernel.	(X) Uses the **call** operator to implement 1D convolution on the x-axis between a seismic trace and a large array of weights. Calls **shift** to access neighbors within a **_for** loop to perform convolution with an arbitrarily sized array of weights. Uses **create** to generate a large vector output of any specified size. (Y) An equivalent operation on the Y axis performed on half of the input data set. (2D) Uses the **call** operator to perform a 2D convolution with a cross-shaped stencil of fixed size. Uses **shift_sticky** to perform vector arithmetic with neighbors using a zero-flux assumption for out-of-bounds accesses (clamped to the nearest boundary value). Uses a stride of 2 on the x-axis when gathering neighbors.
kirchhoff	Generic Kirchhoff migration assuming constant velocity of seismic waves through a sub-surface.	Accumulates the contributions of each seismic trace to a sub-surface reconstruction. Uses a constant velocity model where the time from source to receiver is proportional to the distance between the source and receiver. Uses the equation of a circle to determine the possible reflection points. Uses correlation between multiple source-receiver pairs to identify the location of the reflecting sub-surface.	(1) Uses the **call** operator to implement migration with large vector arithmetic. Uses **create** to allocate a large vector output. Constructs sets of **indices<>** with the user-specified resolution. Uses a **_for** loop to parallelize over circle centers. Uses a **select** statement to perform a boundary check. (2) A 2D variation on (1) where the output and index sets are 2D X-Z datasets. Uses

Sample	Description	Algorithm	Implementation
			repeat_col and **repeat_row** to generate the 2D index sets. Uses **create** to initialize a 2D large vector containing two-component tuples used to perform a gather. Specifically, the 2-tuples are used to index the trace data to determine the appropriate contribution for the output reconstruction.

Configuring Your Development Environment

4

This chapter explains how to configure your development environment for use with the Intel® Array Building Blocks software.

Creating an Intel® Array Building Blocks Project on Linux* OS

This section describes how to create and set up manually the project on Linux* OS.

You must use the common build rule file `include/makefile.inc`. The Intel® Array Building Blocks (Intel® ArBB) software supports different compilers. See their list in the *Release Notes* documents. You can set the default compiler in `samples/common/makefile.inc`. `icc` denotes the Intel C++ Compiler, `gcc` - GNU C/C++ Compiler. For example,

```
ifndef compiler
export compiler=gcc
endif
```

You can easily build a new makefile for a project by including this common rule file. Please refer to `samples/finance/black-scholes/Makefile`.

```
WKLD_DIR=../../..

VPATH=$(WKLD_DIR)/samples/common

SRCS=bs_c.cpp bs.cpp main.cpp data.cpp util.cpp

include $(WKLD_DIR)/samples/common/makefile.inc

ifneq ($(MOS),win)

C_EXTRA_FLAG = -msse2

endif
```

`WKLD_DIR` defines the installation directory of the Intel ArBB package. You must properly set it.

The second line is optional, it defines the location (other than current directory) of the source files included in this application if.

The sixth line is also optional, and defines extra G++ compilation flags. The common rule file defines the basic flags. You can append extra flags at the end of this line.

The third line contains the required variable SRCS that defines the list of source files. If the source files are not in the current directory, you must define the VPATH variable.

The fourth line includes the common rule file. This line is required, too.

In summary, you must define `WKLD_DIR` and SRCS, and include the file `makefile.inc`.

You can use the command shown below to build the project. The argument following the `make` command is one of the target objects, such as `all`, `dbg`, `opt`, and `clean` . The default target object is `all`.

```
make [all | dbg | opt | clean]
```

Debugger Integration

The debug facilities of Intel® Array Building Blocks (Intel® ArBB) are called *debugger integration* since there is no separate debugger needed to debug Intel ArBB code. The debugger integration consists of scripts that help existing debuggers to work with Intel ArBB code.

The intention of the debugger integration is to show the content of variables in the usual way, and to appear similar to non-ArBB variables. Without using the Intel ArBB debugger integration opaque types as specified by the Intel ArBB VM API cannot be inspected, and no actual user data is shown. The debugger integration helps to format these opaque types.

The scripts enable the following:

- Perform introspection of Intel ArBB scalars and dense containers and visualization of their values
- Provide insight into Intel ArBB opaque types (C++ space)

The debugging of Intel ArBB code requires `ARBB_OPT_LEVEL=O0` ("O-zero"), which is *immediate*, or *emulation mode* execution. This mode execution of Intel ArBB provides the following:

- Triggering non-JIT code execution, that is no IR is recorded, and the execution happens in the C++ space
- Capturing and closure creation is not supported in immediate mode

The properties delivered by the immediate mode execution are:

- Common debugger features are working as expected, for example break points
- Control flow can be directly monitored

Debugging in the immediate mode technically implies:

- Execution on the host system, that is independent of the compilation target (remote execution)
- No memory allocation, and no copies are introduced over regular immediate mode execution

Intel ArBB supports GNU Debugger (gdb). Refer to the *Release Notes* to find out the exact version needed.

GNU Debugger Integration

The requirements of Intel® Array Building Blocks (Intel® ArBB) debugger integration with the GNU debugger (GDB) are as follows:

- Python script to pretty-print Intel ArBB data objects
- Suitable GDB front-ends, for example, DDD or GNU Emacs
- GNU debugger GDB 7.0, or later

As an example, assuming a GDB session is running where the following code has been executed in the current scope:

```
i16                 i(16);
boolen              b(true);
dense<i32>          d1;
```

```
dense(boolean>     d2(8);
dense<i32, 2>      d3(2, 4);
dense<i32, 3>      d4(2, 2, 2);
```

If you type

```
print i
```

the result is:

```
$1 = 16
```

If you type

```
print b
```

the result is:

```
$2 = true
```

If you type

```
print d1
```

the result is:

```
$3 = ArBB dense, uninitialized = {
  [0] uninitialized
}
```

If you type

```
print d2
```

the result is:

```
$4 = ArBB dense<arbb_boolean, 1> = {
  [0] = container (8) = {false, false, false, false, false, false, false, false)
}
```

If you type

```
print d3
```

the result is:

```
$5 = ArBB dense<arbb_u32, 2> = {
   [0] = container (2, 4) = {[0] = {0, 0}, [1] = {0, 0}, [2] = {0, 0}, [3] = {0, 0}}
}
```

If you type

```
print d4
```

the result wil be:

```
$6 = ArBB dense<arbb_u32, 3> = {
  [0] = container (2, 2, 2) = {[0] = {[0] = {0,0}, [1] = {0, 0}}, [1] = {[0] = {0,0}, [1]
= {0, 0}}}
}
```

The formatting of the containers above is subject to GDB settings for printing arrays and may not exactly match the above examples. Among the relevant options are those set by the following GDB commands:

- set print array
- set print array-indexes
- set print elements
- set print pretty

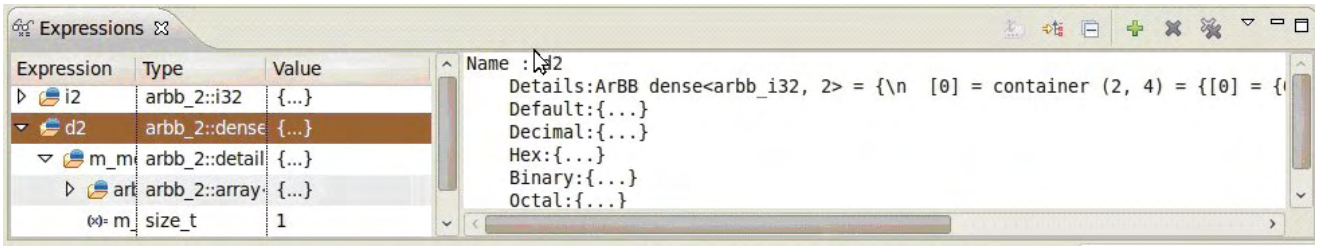
Refer to the GDB documentation for details.

In addition to providing information using the standard **print** command, you can use a custom command to print individual elements of a dense variable. The syntax of the command is as follows:

```
arbb print var[page][row][col]
```

The number of indexes provided must match the dimensionality of variable. If the dense variable holds an array or user type, the elements in each sub-container matching the given indexes is printed. Expression evaluation is not supported. A single variable name must be provided and indexes must be numeric. This command does not support options to print values in other than base 10.

The GDB integration also provides some level of support for GUI debuggers that are built on top of GDB. For instance, when you add an Intel ArBB variable to the **Expressions** window in the Eclipse CDT debugger, the *pretty printed* value is shown in the **Details** field for the variable as follows:



Programming with Intel® Array Building Blocks

5

Writing Simple Functions Using Scalars

This section introduces scalars, the simplest types available in Intel® Array Building Blocks (Intel® ArBB), and shows you how to write simple functions to express computations using these types. Functions based purely on scalars are compiled and optimized by Intel ArBB when passed to the `arbb::call()` function. However, such functions do not contain any explicit parallelism because all parallelism in Intel ArBB is based on container types.

See also the following sections in the *Intel® Array Building Blocks Application Programming Interface Reference Manual*:

- Scalars
- Function Invocation
- Scalar and Element-wise Functions
- Small Arrays
- Complex Numbers
- Control Flow

See Also

- Adding Parallelism with Containers

Scalar Types

The scalar types provided by Intel® Array Building Blocks (Intel® ArBB) enable declaring variables such as integers, floating-point numbers, and Boolean values. The scalar types for integers and floating-point numbers correspond to the C++ primitive types such as `int` and `float`, but are processed by the Intel ArBB run-time library. Computations expressed using these types can therefore be captured and compiled by Intel ArBB.

Like all Intel ArBB types, all scalar types are declared in the arbb namespace.

The table below summarizes the Intel ArBB scalar types. The C/C++ equivalents are listed as "typical" because C and C++ do not impose exactly how many bits the representation of a given type should have.

Intel ArBB Scalar Types

Scalar Type	Description	Typical C/C++ Equivalent
`i8`	8-bit signed integer	signed char
`i16`	16-bit signed integer	short

Scalar Type	Description	Typical C/C++ Equivalent
`i32`	32-bit signed integer	`int`
`i64`	64-bit signed integer	`long long`
`u8`	8-bit unsigned integer	unsigned char
`u16`	16-bit unsigned integer	unsigned short
`u32`	32-bit unsigned integer	unsigned int
`u64`	64-bit unsigned integer	unsigned long long
`f32`	32-bit floating-point number	`float`
`f64`	64- bit floating-point number	`double`
`boolean`	Boolean (`true` or `false`) value	bool
`isize`	Signed integer large enough to store indices	`ssize_t` (non-standard)
`usize`	Unsigned integer large enough to store indices	`size_t`

The `isize` and `usize` types are used to store values related to the size of a container, such as the size itself or indices used to look up elements in a container. These types do not have a defined bit-width because their representation depends on the target architecture. For example, on an architecture supporting 64-bit addresses, these types are usually 64-bit, but on a 32-bit architecture, they may be 32-bit.

You can declare scalars like any other C++ objects. Upon construction, a scalar is initialized to a zero value of the appropriate type. You can also construct a scalar by copying its value from another scalar or by initializing it with a C++ primitive value of an appropriate type:

```
arbb::i32 foo; // initialized to zero

arbb::i32 bar = foo; // initialized by copy from foo

arbb::i32 baz = 17; // initialized to the value 17
```

The Intel ArBB scalar types provide many operations, including arithmetic, logical, comparison, transcendental, and bitwise functions. You can use these operations to express computations on scalars in the same way as on C++ primitive types. For details of the operations available on scalars, see the *Intel© Array Building Blocks Application Programming Interface Reference Manual*.

Unlike C++ primitives such as float variables, Intel ArBB scalars are not values themselves. Instead, scalars *represent* values. Use the `arbb::value()` function to access the value stored in a scalar as a regular C++ type. This enables Intel ArBB control over the actual storage and computation resulting in a particular scalar. For example, the computation defining a particular scalar may run asynchronously at the same time as some other C++ code. Calling `arbb::value()` on such a scalar ensures that the computation has completed so that the actual resulting value can be returned.

```
// Declare some scalars.

arbb::i32 i = 5;

arbb::i32 j = 6;
```

```
// Obtain the value of i.
int i_value = arbb::value(i); // i_value is 5.

// Declare another scalar and perform some computation.
arbb::i32 k;
k = i + j;
int k_value = arbb::value(k); // k_value is 11.
```

Writing and Calling Functions

For Intel® Array Building Blocks (Intel® ArBB) to optimize code expressed using Intel ArBB types, place such code in a C++ function and use the `arbb::call()` function to invoke it. The function must return `void`, and accept Intel ArBB type parameters. The parameters can be references or const references, but cannot be pointers. Up to 35 parameters can be used in such functions.

The following example shows a simple function defined using scalar types and a call to this function using `arbb::call()`:

```
#include <iostream>
#include <arbb.hpp>

// Define a function using Intel(R) ArBB types.
void my_function(arbb::f32& result, arbb::f32 a, arbb::f32 b)
{
result = a + b;
}

int main()
{
// Declare some scalar variables.
arbb::f32 x = 1.0f;
arbb::f32 y = 2.0f;
arbb::f32 z;
// Compile and execute the function using Intel(R) Array Building Blocks.
arbb::call(my_function)(z, x, y);

// Print out the value of z after the function execution.
std::cout << “z = “ << value(z) << std::endl;
```

```
}
```

On the first use of a function with `arbb::call()`, Intel ArBB optimizes and compiles the computations expressed in the function. Subsequent calls to `arbb::call()` reuse the compiled code from this first invocation, so they will not incur any execution overhead.

Within an `arbb::call()` invocation, you can perform other `arbb::call()` invocations or call C++ functions directly. Insert such C++ functions directly into the resulting code.

Do not use `arbb::call()` on variables declared locally within an `arbb::call()`. Such variables are placeholders for the computation to be performed in the code compiled by `arbb::call()` and therefore do not have a single corresponding value.

Control Flow

Because scalars *represent* a value rather than *have* a value inside of `arbb::call()`, they cannot be used directly in places where C++ value types such as float and bool can be used. Therefore those scalars cannot be used directly in C++ control flow constructs such as if-statements and for-loops. To enable control flow in functions, Intel® Array Building Blocks (Intel® ArBB) provides a set of control flow constructs corresponding to those used in C++.

Intel ArBB provides control flow constructs through macros such as `_if` ,`_for` and `_while`. Unlike their C++ counterparts, pair all these macros with a corresponding ending macro such as `_end_if`.

Use the `_if` construct to express code that is conditionally executed depending on the value of a scalar. Use the `_else_if` macro to chain together execution of parts of code under multiple conditions.

```
f32 x = ...;
f32 result;

// Start an if statement.
_if (x > 1) {
  // This code will execute if x is greater than one.
  result = x;
} _else_if (x < 0) {
  // This code will execute if x is less than zero.
  result = x * x * x;
} _else {
  // This code will execute if x is between zero and one.
  result = -x;
_end_if;
```

You can express while loops, for loops, and do-until loops using the `_while` , `_for` and _do macros. Unlike in C++ for statements, separate the three parameters of `_for` by commas instead of semicolons. The _do construct differs from C++ do-while loops: Intel ArBB inverts the C++ loop condition and expresses it using the `_until` macro.

```
// Initialize i to zero, and keep incrementing until it reaches a million.
_for(i32 i = 0, i < 1000000, ++i) {
  // This code will be execute a million times.
  some_function(i);
} _end_for;

f32 x = 100.0f;
// Repeat the loop body until x is no longer greater than one.
_while (x > 1.0f) {
  x = x * 0.5f;
} _end_while;
```

```
i32 y = 0;
// Always execute the loop body at least once, then repeat until y is greater than ten.
_do {
  y++;
} _until (y > 10);
```

The _break and _continue statements enable terminating a loop early or continue to the next loop iteration.

```
f32 x = ...;
_for(i32 i = 0, i < 100, ++i) {
  _if (i % 2 == 0) {
    // Skip even iterations
    _continue;
  } _end_if;

  x = x * (1.0f + 1.0f/f32(i));
  _if (x > 10.0f) {
    // Stop the loop completely if x exceeds ten.
    _break;
  } _end_if;
} _end_for;
```

Important. All the Intel ArBB control flow statements, including loops, execute serially.

Complex Numbers and Small Arrays

Intel® Array Building Blocks (Intel® ArBB) specializes the `std::complex` class template from the C++ standard library to work correctly with floating-point scalar types `arbb::f32` and `arbb::f64`. Instances of this specialization behave like those on primitive types, such as `std::complex<float>`, but can be used to express computations optimized and compiled by Intel ArBB.

```
// Declare a complex arbb::f32 variable.
std::complex<arbb::f32> z(1.0, -1.0);

// Perform a complex multiplication.
z = z * z;

// Extract the real part, imaginary part,
// and the magnitude of the complex number.
arbb::f32 r = std::real(z);
arbb::f32 i = std::imag(z);
arbb::f32 mag = std::abs(z);

// Print out the value of z.
```

```
std::cout << "z = (" << value(r) << "," << value(i) << ")\n";
```

Intel ArBB provides the `arbb::array` class template to represent and perform operations on short arrays of Intel ArBB scalars and other types with array sizes being determined at compile time. This type is based on the `tr1::array` template supported by many C++ compilers, but adds all the operations available on scalar types for convenience. For example, you can add two instances of `arbb::array` using `operator+()` if they have the same size and element type. This operation adds each pair of elements from the two arrays at corresponding indices.

The `arbb::array` template accepts two template parameters. The first parameter specifies the element type contained in the array. The second parameter specifies the number of elements in the array. Both parameters are required.

Instances of `arbb::array` are C++ *aggregates*, which means that you must initialize them using brace-enclosed initializer lists like regular C++ arrays.

```
// Declare and initialize some arrays.
arbb::array<arbb::f32, 4> a = {1.0f, 2.0f, 3.0f, 4.0f};
arbb::array<arbb::f32, 4> b = {5.0f, 6.0f, 7.0f, 8.0f};


// Declare another array, and perform an array computation
arbb::array<arbb::f32, 4> c;
c = a + b;
// The c variable now contains {6.0f, 8.0f, 10.0f, 12.0f}.
```

As the name implies, small arrays are best used for small sequences of a fixed size due to their interface and implementation. For larger sequences or sequences whose size is determined at run time, instead use one of the Intel ArBB container types such as `arbb::dense`.

You can use both the Intel ArBB complex and small array types as parameter types for functions to be used with `arbb::call()`.

See Also

- Adding Parallelism with Containers

Adding Parallelism with Containers

This section introduces the Intel® Array Building Blocks (Intel® ArBB) container types, `arbb::dense` and `arbb::nested`. Use these Intel ArBB types to allocate and manipulate arbitrary sequences of scalars, small arrays, complex values, and user-defined types with sequence sizes determined at run time. Operations that take place on containers are a primary means of expressing parallelism using Intel ArBB.

See also the following sections in the *Intel® Array Building Blocks Application Programming Interface Reference Manual*:

- Dense Containers
- Nested Containers
- Scalar and Element-wise Functions
- Container Functions

Using Dense Containers

Dense containers are the simplest containers available in Intel® Array Building Blocks (Intel® ArBB). They can be one, two, or three-dimensional, and represent sequences of simpler Intel ArBB types such as scalars. The `arbb::dense` class template represents dense containers. Use this template similarly to the standard C++ `std::vector` class template to store data whose size is determined at run time and to perform operations (such as arithmetic) directly on dense containers.

The `arbb::dense` template accepts two template parameters. The first parameter specifies the element type of the container (for example, `arbb::f32`). The second parameter specifies the number of dimensions in the container and can be one, two, or three. If it is not specified, the number of dimensions defaults to one.

To construct dense containers do either of the following:

- Use a default constructor
- Pass in a size along each dimension
- Bind the dense container to regular C++ data

```
// An empty, one-dimensional dense container of f32 elements.
arbb::dense<arbb::f32> a;
// A sized but uninitialized two-dimensional dense container of i8 elements.
arbb::dense<arbb::i8, 2> b(1024, 768);
// Allocate some data and bind a dense container to it.
std::vector<float> my_data(256);
arbb::dense<arbb::f32> c;
arbb::bind(c, &my_data[0], 256);
```

You can use all of the operators and functions available to perform computations on scalars directly on dense containers of those scalars. Like `arbb::array`, operations on containers execute in an element-wise fashion, and the element sizes of containers used together in such an operation must match. Additionally, any element-wise operation on a dense container that takes more than one parameter can be mixed with arguments of the element type of the container. Because element-wise operations operate on each element of a container independently, Intel ArBB executes such operations in parallel.

```
arbb::dense<arbb::f32> container_a, container_b;
arbb::f32 scalar;

// Add every element in container_a to every corresponding
// element in container_b
container_a = container_a + container_b;

// Multiply every element in container_a with scalar.
container_a = container_a * scalar;
```

The `arbb::dense` template provides several member functions to obtain properties such as the size of a container along each dimension. Using the square bracket (`operator[]`) and round bracket (`operator()`) operators to read and write single elements of dense containers. The two forms of the operators are equivalent in behavior, but the `operator[]` form accepts only a single index, whereas the `operator()` form accepts multiple indices to access multi-dimensional containers. Use these operators in arbitrary computations involving containers. For the special case of accessing the values in dense containers directly using C++ types, refer to the *range* functions.

For details on the member functions available in `arbb::dense`, the *Intel® Array Building Blocks Application Programming Interface Reference Manual*.

See Also

- Binding and Accessing Dense Container Data

Binding and Accessing Dense Container Data

Like you access the values of Intel® Array Building Blocks (Intel® ArBB) scalar variables using the `arbb::value()` function, use the *range* interface provided by dense containers to access their values as plain C++ data. Accessing a dense container as a range enables your direct operating on the data in the dense container as on a regular standard C++ container such as `std::vector`. Because containers can be large and may be stored by Intel ArBB internally on another device or in an internally optimal data format, you need to indicate whether you are accessing the dense container to read its elements, write to its elements, or both. For this reason, `arbb::dense` provides three member functions to obtain ranges:

- `read_only_range()`
- `write_only_range()`
- `read_write_range()`

All range member functions return an `arbb::range` or `arbb::const_range` instance. This type provides access to the data in its range through iterators with a `begin()` and `end()` member function, as well as random access to the data using the square bracket indexing operator `operator[]`.

```
arbb::dense<arbb::f32> foo(...);

// Construct a range that can be read from and written to.
arbb::range<arbb::f32> rw_range = foo.read_write_range();
rw_range[0] = 1.0f; // Set first element to 1.0f.
float second_element = rw_range[1]; // Read value of second element.

// Construct a range that can only be read.
arbb::const_range<arbb::f32> read_range = foo.read_only_range();
float third_element = read_range[2]; // Read value of third element.

// Use range iterators.
arbb::const_range<arbb::f32>::iterator begin = read_range.begin();
arbb::const_range<arbb::f32>::iterator end = read_range.end();

for (arbb::range<arbb::f32>::iterator I = begin; I != end; ++I) {
  std::cout << *I << std::endl; // Print each element in the range.
}

// Fill entire range with zeros.
std::fill(rw_range.begin(), rw_range.end(), 0.0f);
```

Because a range access may involve some synchronization, such as waiting for a pending operation to complete or transferring data from another device, the lifetime of a range is limited until the next Intel ArBB operation involving the container is accessed. Once the lifetime of a range has expired, you can no longer perform operations such as accessing its iterators on it.

```
arbb::dense<arbb::f32> container(...);

 // Obtain a range from the container.
 arbb::range<arbb::f32> range1 = container.read_write_range();

 // Fill entire range with zeroes.
 std::fill(rw_range.begin(), rw_range.end(), 0.0f);

 // Call a function that uses the given container.
 // range1 is now invalid.
 arbb::call(some_function)(container);

 // At this point, it is no longer safe to perform any operations
 // on range1.

 // Obtain another range from the container.
 arbb::range<arbb::f32> range2 = container.read_write_range();

 // Output the first element from the range.
 std::cout << range2[0] << std::endl;

 // Perform another operation on the container.
 // range2 is now invalid.
 container += container;

 // At this point, it is no longer safe to perform any operations
 // on range1 or range2.
```

If you already have data allocated using plain C++ types, you can permit Intel ArBB to directly access this data by *binding* a container to the data using the `arbb::bind()` family of functions. A container that was initialized in this manner is called a *bound* container. When you assign to a bound container, the size of the source container must exactly match the destination container because Intel ArBB updates the data bound to the container appropriately.

You can bind any dense container, including two-dimensional and three-dimensional containers. You must always pass a pointer to the data being bound, as well as the number of elements along each dimension. You can specify dimensions using either of the following:

- A sequence of scalar arguments
- `arbb::array` of scalars of the appropriate size

By default, Intel ArBB assumes that your data is contiguously stored in page/row/column order. All forms of `arbb::bind()` also accept optional pitch parameters specifying the number of bytes between each column, row, and page. This enables binding to arbitrary regularly ordered sequences of data, such as two-dimensional column-major padded data, or portions of arrays of structures.

You must still use the range functions described in this section before you access the data on the host, even in its original bound location.

NOTE. This version of Intel ArBB does not check for violations of this rule, and performs synchronization frequently enough for the data to be directly accessed. To enable your project's work with future versions of Intel ArBB, always use the range access functions regardless.

If you have control over the allocation of data being bound, you can use the Intel ArBB alignment functions and macros to ensure that your data is aligned well in memory for optimal performance. The ARBB_CPP_ALIGN macro enables allocating global, class member, or function-local data with an appropriate alignment. You can use the ARBB_CPP_ALIGN_ALLOCA macro to perform aligned data allocations on the stack with data size determined at run time. Use the `arbb::aligned_malloc()` and `arbb::aligned_free()` functions to allocate aligned data on the heap.

The following code example illustrates the use of `arbb::bind()` and its interaction with range functions.

```
// Allocate 1024 floating point elements on the stack.
ARBB_CPP_ALIGN(float data[1024]);

// Fill the data with some values.
std::fill(data, data + 1024, 1.0f);

// Construct a dense container and bind it to the data.
arbb::dense<arbb::f32> container;
arbb::bind(container, data, 1024);

// At this point, attempting to read or write to the user data
// results have undefined behavior.

// The following operation affects the bound data.
container = container + 1.0f;

// You must call a read range function before reading the values.
container.read_only_range(); // ...or read_write_range()

// Read some data and print it to standard output.
std::cout << data[0] << std::endl;

// At this point, you must call a write range function before
// you modify the data.
container.write_only_range(); // ...or read_write_range()

// Write to the user data.
data[0] = 1.0f;

// Perform another Intel ArBB operation on the container.
container = container * 2.0f;

// Now you must call a range function again before you read or
// write the data, because the previous operation invalidated
// the above ranges.
```

Dense Container Operations

In addition to performing element-wise operations on dense containers, Intel® Array Building Blocks (Intel® ArBB) also provides a wide range of operations that operate on a container as a whole to rearrange or combine its elements.

All functions operating on containers, not only element-wise, semantically always produce a new container with the results rather than operate on the data in place. When compiling a function through `arbb::call()`, Intel ArBB looks for opportunities to reuse data without breaking these semantics when it can.

Most of the operations for dense containers process multiple elements independently. Intel ArBB takes advantage of this feature where possible and parallelizes these operations.

For details on the operations available on dense containers, refer to the *Intel® Array Building Blocks Application Programming Interface Reference Manual*.

Rearranging Dense Containers

The following functions can be used to rearrange values in a dense container:

Function	Description
`shift`	Returns a dense container whose elements are shifted by some amount. Out-of-bounds accesses result in a default value or a provided value.
`shift_sticky`	Returns a dense container whose elements are shifted by some amount. Out-of-bounds accesses are clamped to the edges of the container.
`rotate`	Returns a dense container whose elements are shifted by some amount. Out-of-bounds accesses are wrapped to repeat the container.
`shift_row[_sticky]`	Equivalent to `shift[_sticky](container, 0, distance, …)`.
`shift_col[_sticky]`	Equivalent to `shift[_sticky](container, distance, 0, …)`.
`shift_page[_sticky]`	Equivalent to `shift[_sticky](container, 0, 0, distance, …)`.
`repeat`	Returns a one-dimensional dense container storing a given number of copies of the one-dimensional source.
`repeat_row`	Returns a two-dimensional dense container where each row corresponds to the one-dimensional source.
`repeat_col`	Returns a two-dimensional dense container where each column corresponds to the one-dimensional source.
`repeat_page`	Returns a three-dimensional dense container where each page corresponds to the two-dimensional source.
`replace`	Returns a dense container with a sub-section replaced by another container.
`replace_row`	Returns a dense container with a row replaced by another container.
`replace_col`	Returns a dense container with a column replaced by another container.
`replace_page`	Returns a dense container with a page replaced by another container.
`replace_dim3`	Returns a dense container with a pillar replaced by another container.
`swap_rows`	Returns a dense container with the two given rows swapped.
`swap_cols`	Returns a dense container with the two given columns swapped.
`swap_pages`	Returns a dense container with the two given pages swapped.
`transpose`	Returns a transposed dense container. For three-dimensional containers, only the rows and columns are transposed.
`section`	Returns a dense container containing a subsection of the source container.

Function	Description
`sort`	Returns a dense container sorted by its values or keys provided in another container.
`cat`	Returns a dense container containing the concatenation of two source containers.
`reshape`	Returns a two-dimensional or three-dimensional dense container of a given size filled with values from a one-dimensional dense container.
`reshape_as`	Returns a two-dimensional or three-dimensional dense container with the same size as another container filled with values from a one-dimensional dense container.
`shuffle`	Returns a dense container storing elements alternately drawn from two source containers.
`unshuffle`	Returns the concatenation of two dense containers obtained by alternately drawing elements from a source container.

Gathering and Scattering

The following functions use a container of indices to read from an existing or write to a new dense container.

Function	Description
`gather()`	Returns a dense container with elements drawn from a provided source based on a provided sequence of indices.
`scatter()`	Returns a dense container with elements at provided indices set to corresponding elements from a source container, and all other elements initialized to default values or drawn from another container. Duplicate indices lead to unspecified values in those indices.
`add_merge()`	Performs the same operation as scatter(), but merges duplicate indices by adding their values together.

Mask Operations

Use the following functions to rearrange data in a dense container based on a mask:

Function	Description
`pack()`	Returns a dense container containing only those elements from a given source container whose corresponding values in a provided mask are true.
`unpack()`	Returns a dense container containing with elements at a given index drawn from a source container if corresponding values in a provided mask are true. Values where the mask contains false elements are initialized using a default or provided element.

Filling Dense Containers with Patterns

Intel® Array Building Blocks provides the following functions to create a dense container with a regular pattern:

Function	Description
`fill()`	Returns a dense container containing the same element repeatedly.
`indices()`	Returns a dense container with a regularly increasing sequence of indices.
`mask()`	Returns a dense container containing a repeating pattern of true and false values.

Nested Containers

Nested containers provide the ability to represent a container of sub-containers. Nested containers are limited to one level of nesting, which means that nested containers are equivalent to a sequence of dense containers. The dense containers stored in a nested containers are called *segments*.

The `arbb::nested` class template represents nested containers. It takes a single template parameter specifying the element type of its segments. Unlike dense containers, nested containers are always one-dimensional. You create instances of arbb::nested by transforming dense containers into nested containers, or by performing operations on nested containers.

Like dense containers, any function available on the element type of a nested container can also be applied to nested containers as a whole. In that case the number of segments, as well as each segment size, must match amongst any nested containers used in an operation taking more than one nested container. The operation will be applied segment-by-segment to matching elements in matching segments.

Nested containers provide a set of member functions to access properties such as the number and lengths of segments. It also provides `segment()` member function to extract a single segment from a nested container, and a round-bracket operator (`operator()`) to extract a single element from a given segment. For details on the member functions provided by `arbb::nested`, refer to the *Intel® Array Building Blocks Application Programming Interface Reference Manual*.

Nested Container Operations

Intel® Array Building Blocks (Intel® ArBB) provides several functions that operate on nested containers as a whole, as well as several ways to construct nested containers from dense containers. As with the element-wise operations, and operations provided on dense containers, most of these operations involve processing several independent elements at once. Intel ArBB parallelizes these operations when possible.

For details on all the operations available on nested containers, refer to the *Intel® Array Building Blocks Application Programming Interface Reference Manual*.

Creating Nested Containers from Dense Containers

The following functions can be used to create a nested container from a dense container:

Function	Description
`reshape_nested_offsets()`	Returns a nested container with segment elements drawn from a given dense container and segment boundaries provided as an increasing list of offsets.

Function	Description
`reshape_nested_lenghts()`	Returns a nested container with segment elements drawn from a given dense container and segment boundaries provided as a list of segment lengths.
`reshape_nested_flags()`	Returns a nested container with segment elements drawn from a given dense container and segment boundaries provided as a mask indicating new segment boundaries.
`split()`	Returns a nested vector containing up to three segments, with each segment containing data from a dense container depending on whether a corresponding value in another dense container is -1, 0 or 1.

Rearranging Nested Containers

The following functions can be used to rearrange the segments of nested containers or elements in segments of nested containers. In general these functions accept an optional level argument as their last parameter, which can be used to adjust whether the operation applies to the segments (level 0) or the elements of the segments (level 1). By default, operations apply at the segment level.

Function	Description
`repeat()`	Returns a nested vector containing several copies of the segments or segment elements in a provided nested container.
`replace()`	Returns a nested container with one element of a particular segment replaced by a given value.
`replace_segment()`	Returns a nested container with one segment replaced by a provided dense container.
`reshape_as()`	Returns a nested container with the same segment shape as a given nested container, containing data drawn from a given dense container.
`split()`	Returns a nested vector containing up to three segments, with each segment containing data from a nested container depending on whether a corresponding value in another nested container is -1, 0 or 1.
`unsplit()`	Returns a container given a container of -1/0/1 values such that splitting it would recreate the source container.
`cat()`	Returns a nested container containing the concatenation of two source containers at the element or segment level.
`shuffle()`	Returns a nested container containing segments or segment elements alternately drawn from two source containers.
`unshuffle()`	Returns the concatenation of two nested containers obtained by alternately drawing segments or segment elements from a source container.

Function	Description
`unpack()`	Returns a nested container with elements at a given index drawn from a source container if corresponding values in a provided mask are true. Values where the mask contains false elements are initialized using a default or provided element.

Reductions and Scans

Intel© Array Building Blocks (Intel© ArBB) provides reduction and scan functions that can be applied to dense and nested containers. These functions combine the elements of a container, e.g. by summing them. In the case of a reduction, all elements along one or more dimensions are reduced into a single element. In the case of a scan, each element in the resulting container contains the partial combination of all values up to that point.

For every operation supported in a reduction or scan, at least three versions are provided: a reduction function, an exclusive scan function, and an inclusive scan function. The exclusive scan always returns an initial element containing the identity for the given operation (for example, a zero value for an addition, or a one value for a product), and includes all elements up to a given index in the computation of a given result element. The inclusive scan always returns the first element of the source container as its first element, and includes all elements *up to and including* a given index in the computation of a given result element.

Reductions and scans can apply to a particular dimension of a multi-dimensional nested container. Reductions and scans on nested containers always apply to each segment independently.

Some common reduction functions, for example, sum reductions, also have short-hand versions that reduce along all dimensions, producing a single element as the result.

This table lists all of the scan and reduction functions available in Intel ArBB:

Operation	Reduction	Exclusive Scan	Inclusive Scan	Full Reduction
Addition	`add_reduce()`	`add_scan()`	`add_iscan()`	`sum()`
Multiplication	`mul_reduce()`	`mul_scan()`	`mul_iscan()`	
Minimum	`min_reduce()`	`min_scan()`	`min_iscan()`	
Maximum	`max_reduce()`	`max_scan()`	`max_iscan()`	
Logical and	`and_reduce()`	`and_scan()`	`and_iscan()`	`all()`
Logical or	`ior_reduce()`	`ior_scan()`	`ior_iscan()`	`any()`
Logical xor	`xor_reduce()`	`xor_scan()`	`xor_iscan()`	

Adding Parallelism Using map()

This section introduces the `arbb::map()` function and explains how to use it.

The `arbb::map()` function behaves very similarly to `arbb::call()` that passes a C++ function to Intel© Array Building Blocks (Intel© ArBB) for optimized execution. Such call can contain parallelism in the form of operations applied to containers but is not parallel itself. The `arbb::map()` function permits you to invoke a function written in terms of scalars across all elements of one or more dense containers.

See Also

- Writing Simple Functions Using Scalars
- Adding Parallelism with Containers

Using the map() Function

To use a function with `arbb::map()`, you need to write it in terms of individual elements. When invoking the function using `arbb::map()`, you can pass one or more arguments to these elements using containers of the element type. You can also pass arguments that exactly match the type of the parameter as you do for `arbb::call()`. Container arguments passed to element type parameters are called *varying arguments*. Arguments that exactly match the parameter type are called *fixed arguments*.

You must ensure that the dimensionalities and sizes of all varying arguments passed to `arbb::map()` invocation match exactly, otherwise a run-time error occurs. At least one of the varying arguments must be an output argument that is passed to a parameter of non-const reference type.

As the compilation of a function using `arbb::map()` depends on the arguments being varying or fixed, you can only use `arbb::map()` within an `arbb::call()` invocation. You cannot place invocations of `arbb::map()`, `arbb::call()`, or allocations of containers inside a function invoked through `arbb::map()`.

This code example illustrates the use of `arbb::map()`:

```
#include <iostream>
#include <arbb.hpp>

 // You may pass this function to arbb::map(), because it returns
 // void, takes fewer than 35 parameters (three), and the parameters
 // are all of Intel(R) ArBB types, or references thereof.
 // It also does not contain any arbb::map() or arbb::call()
 // invocations, or declare any local containers, and has a
 // non-container parameter (a) declared as a reference.
 void blend_in_scalar(arbb::f32& a, arbb::f32 factor, arbb::f32 b)
 {
   a += factor * b;
 }

 // As arbb::map() can only occur within a function passed to
 // arbb::call(), and not within functions such as main() below,
 // this function is written to perform the actual arbb::map().
 void blend_in(arbb::dense<arbb::f32>& a, arbb::f32 factor, arbb::dense<arbb::f32> b)
 {
   // This applies blend_in_scalar to all elements of a and b
   // (which are thus varying arguments), with the same value of
   // factor corresponding to all elements (a fixed argument).
   arbb::map(blend_in_scalar)(a, factor, b);
 }
 int main()
 {
   // Create some arbitrary sample data
   arbb::dense<arbb::f32> base = arbb::fill(0.0f, 1024);
   arbb::dense<arbb::f32> addition = arbb::fill(10.0f, 1024);
   arbb::f32 factor = 0.1f;

   // Apply the function a few times.
   for (int i = 0; i < 4; ++i) {
     arbb::call(blend_in)(base, factor, addition);
   }
```

```
   // Output the resulting values
   arbb::const_range<arbb::f32> base_data = base.read_only_range();
   for (std::size_t j = 0; j < 1024; ++j) {
      std::cout << arbb::value(base_data[j]) << std::endl;
   }
}
```

For additional information, see the "Function Invocation" section of the *Intel® Array Building Blocks Application Programming Interface Reference Manual*.

User-defined Types

Intel® Array Building Blocks (Intel® ArBB) supports the use of user-defined C++ classes and structures in functions passed to `arbb::call()` and as element types of containers.

See also the User-defined Types section of the *Intel® Array Building Blocks Application Programming Interface Reference Manual*.

Overview of User-defined Type Support

To be used with functions executed through Intel® Array Building Blocks (Intel® ArBB) and in Intel ArBB containers, a user-defined type must contain only data members belonging to Intel ArBB types or other valid user-defined types and follow the specified rules. Functions using types that follow these rules or member functions of such types can be used in Intel ArBB computations and apply to containers of these types.

With user-defined types, you can write structured C++ code using Intel ArBB and retain the existing structure of C++ code ported to Intel ArBB. You can use most of the template types directly with the Intel ArBB types.

For example, the `tr1::array` class provided in many standard libraries of C++ compilers can be used with any Intel ArBB scalar type or user-defined type, even in computations provided to Intel ArBB through `arbb::call()`. As `tr1::array` is not supported by some compilers, Intel ArBB provides its own type `arbb::array` based on `tr1::array` extending the functionality of this type. For example, `arbb::array` supports element-wise operations.

If you define a function on a user-defined type that has the same name as a function available on Intel ArBB scalar types and a matching function signature, you can use this function on containers of your user-defined types. For example, if you define an addition operator by overloading `operator+()` on a user-defined type, you can add containers of that user-defined type. Classes representing values of a type, especially numerical classes, are likely to define many operators and scalar functions provided by Intel ArBB. If they satisfy the rules for user-defined types, you can automatically use these functions on containers of these types. The functions are applied in an element-wise manner, as if applied to the containers using an `arbb::map()` invocation containing a call to the user-defined function.

Intel ArBB provides a set of macros to declare your functions to Intel ArBB. These macros can process any functions and member functions applying to your types that you may want to use for containers of these types. You can use these macros to declare functions, member functions, and function templates to enable their application to containers of user-defined types. This mechanism makes your types as useful as scalar types, as it permits the `arbb::array` class and `std::complex` specialization on floating-point scalar types to work as first-class types in Intel ArBB.

All these functionalities are especially useful because they permit Intel ArBB to capture all computations of a function using user-defined types, and optimize them in the same way as the computations applied directly to built-in Intel ArBB types. The nature of the Intel ArBB C++ frontend eliminates inherent performance penalty for using user-defined types. Like C++ templates, the user-defined type support in Intel ArBB permits functions to be easily inlined and thus optimized well.

The following code example shows several user-defined type features and their usage:

```
// A user-defined class that satisfies the requirements for use inside
// of Intel(R) ArBB containers and functions.
class quaternion {
public:
  // Because this class does not define any constructors or assignment
  // operations, it will get the compiler-generated implicit versions,
  // which are suitable for Intel(R) ArBB user-defined types.

  // Since operator* is one of the built-in operations on Intel(R)
  // ArBB scalars, it will automatically become available to
  // containers of quaternions.
  quaternion operator*(const quaternion& o) const
  {
    quaternion r;
    r.a = a*o.a - b*o.b - c*o.c - d*o.d;
    r.b = a*o.b + b*o.a + c*o.d - d*o.c;
    r.c = a*o.c - b*o.d + c*o.a + d*o.b;
    r.d = a*o.d + b*o.c - c*o.b + d*o.a;

    return r;
  }

  // We need to declare this function to Intel(R) ArBB to make it
  // available on containers of quaternions, since norm() is not a
  // built-in function on scalars. We do this using
  // ARBB_ELTWISE_METHOD_0 below.
  f32 norm() const
  {
    return sqrt(a*a + b*b + c*c + d*d);
  }

  // ...

private:
  // Because all non-static member variables are instance of
  // Intel(R) ArBB types, and none of them are containers, this
  // class can be used as an Intel(R) ArBB user-defined type.
  f32 a, b, c, d;
};

// Declare the quaternion::norm() member function to Intel(R)
// ArBB. This will provide a free function called norm() that takes a
// single container of quaternions and returns a container of f32
// elements.
//
// You must specify the return type, the class to which the member
// function belongs, and the member function name. Because this member
// function does not accept any parameters, no parameter types need to
// be supplied.
//
// Note that this member function returns an f32. Regular
// Intel(R) ArBB types can be mixed with user-defined types in this way.
ARBB_ELTWISE_METHOD_0(f32, quaternion, norm);
```

See Also

- Writing Simple Functions Using Scalars
- Adding Parallelism with Containers

Rules for User-defined Types

You can pass a user-defined type as a parameter to a function processed by `arbb::call()` if it satisfies the following constraints:

- The type has a public default constructor, copy constructor, assignment operator, and non-virtual destructor. All these functions must be either implicitly declared or behave as implicitly declared.
- All non-static member variables of the types must be instances of Intel® Array Building Blocks (Intel® ArBB) scalar types, container types, or other valid user-defined types.
- If the type has any base classes, they must be valid user-defined types.

For example, a simple C++ `struct` containing only data members of Intel ArBB types satisfies these constraints as its constructors and assignment operator are implicitly declared. If you add simple functions to such structure, it remains a valid user-defined type. The rules above allow a type to be passed to a function and compiled and executed through Intel® Array Building Blocks. If a class complying to these rules does not contain any container members, you can also use it as an element type for an Intel ArBB container such as `arbb::dense`.

Declaring Functions on User-defined Types

You can use the `ARBB_ELTWISE` family of macros to declare free functions and member functions to Intel® Array Building Blocks (Intel® ArBB). This enables using these functions with containers of their parameter types.

All `ARBB_ELTWISE` macros fall into three categories:

- `ARBB_ELTWISE_FUNCTION` macros to declare free functions
- `ARBB_ELTWISE_METHOD` macros to declare member functions
- `ARBB_ELTWISE_TMETHOD` macros to declare member functions of class templates

Intel ArBB supports declaration of functions with up to 35 parameters. Each category of macros provides specific macros for functions with different number of parameters. The naming pattern for the macros is *<category_name>_<number>*, where *<number>* specifies the number of parameters to the function being declared. For example, `ARBB_ELTWISE_METHOD_3` declares a member function that takes three parameters.

For member functions, every function declaration macro takes the parameters in the following order:

1. the return type of the function being declared
2. the class name
3. the function name with parameter types, if any

Only member functions can be declared without parameters. You should include a const keyword and/or use a reference type for all types passed to the macros, depending on the way the function is declared.

Free functions declared through `ARBB_ELTWISE_FUNCTION` macros can be used on containers of their parameter types. You can use the container versions of member functions by calling a free function of the same name and passing the desired class instance of the member function as the first argument.

When using free or member functions, you can also mix containers and single instances of the parameter types.

Specializing Computations with Closures

This section explains how you can use closures for specialing computations.

For additional details, see the "Closures" section and the appendix on Virtual Machine API in *Intel® Array Building Blocks Application Programming Interface Reference Manual* and also *Intel® Array Building Blocks Virtual Machine Specification 1.0 Beta 1*.

See Also

- Writing Simple Functions Using Scalars :: Control Flow
- Writing Simple Functions Using Scalars :: Writing and Calling Functions

Using Closures with arbb::call()

To execute a function using `arbb::call()`, use the following syntax:

```
arbb::call(function)(argument1, argument2/*, more arguments */);
```

This expression contains two sets of parentheses because `arbb::call()` can only take one parameter, that is the function being called. The second set of parentheses containing the arguments passed to the function is actually an invocation of the call operator (operator()()) on the object returned by `arbb::call()`. The return type of `arbb::call()` is an instance of the `arbb::closure` class template.

Closures are functions introduced to Intel® Array Building Blocks (Intel® ArBB) by a process called *closure capture*, or *capture* for short. Once captured, a closure can be re-executed many times without incurring any compilation overhead.

The `arbb::call()` function itself is quite simple. It checks whether it has already seen the address of the function being passed in. If it sees the function for the first time, it captures the function into an `arbb::closure` using the `arbb::capture()` function and returns the resulting closure. Otherwise, it returns the previously captured closure. As closures are lightweight references to the actual object code corresponding to a closure capture, copying them around does not affect performance, allowing `arbb::call()` to be used in inner loops.

The following code example shows several user-defined type features and their usage:

```
 void my_function(arbb::dense<arbb::f32>& a, arbb::dense<arbb::f32> b);

 int main() {
  // ...
  while (processing) {
    // arbb::call can be used in inner loops because it is relatively cheap.
    arbb::call(my_function)(foo, bar);
  }
}
```

However, using closures this way you can save small amount of run time by moving an `arbb::call()` invocation out of a loop and using a closure to store the result. Since the result of an `arbb::call()` invocation over the same function is the same in every iteration of the loop, you can use the returned closure from the first call directly in your loop.

```
void my_function(arbb::dense<arbb::f32>& a, arbb::dense<arbb::f32> b);
int main() {
  // ...

  // The template parameter to arbb::closure is the signature of the
  // function.
  arbb::closure<void (arbb::dense<arbb::f32>&, arbb::dense<arbb::f32>)> c;
  // Capture my_function and store the resulting closure in c.
  c = arbb::call(my_function);
  while (processing) {
    // Using the closure directly avoids the small cost of
    // retrieving the closure previously captured from my_function.
```

```
    c(foo, bar);
  }
}
```

See Also

- Writing Simple Functions Using Scalars
- Adding Parallelism with Containers

Closure Capture

Closure capture uses a simple mechanism, but the process has important implications for code that mixes operations on Intel® Array Building Blocks (Intel® ArBB) types and regular C++ types, such as float or std::ostream. Closure capture simply *executes* the function passed to it, passing proxy values as parameters to the function.

Before executing the function, arbb::capture() records the new closure as the one currently being captured by Intel ArBB. As a result, all operations on primitive Intel ArBB types add an instruction to the new closure specifying what operation they correspond to.

After executing the function, the closure contains a sequence of instructions exactly corresponding to all the operations on Intel ArBB types performed by the function. The control flow macros such as _if also simply record their presence, executing both bodies of an if-statement, and executing a loop iteration exactly once, but maintaining the control flow structure in the closure.

Once a closure is captured, Intel ArBB can compile it for a supported architecture and use the compiled object to execute the computations captured from the function repeatedly without any compilation overhead. As this process occurs at run time, the generated code can be optimized for the exact machine configuration in use.

Run-time Specialization Using Closure Capture

As closure capture simply executes a function, it also executes any operations in the function that do not use Intel® Array Building Blocks (Intel® ArBB) types. For example, computations expressed with C++ primitive types such as float and int execute during capture. They do not execute again when the captured closure itself is executed. If the result of such computation is subsequently used in a computation involving Intel ArBB types, it becomes part of the closure.

Similarly, regular C++ control flow, such as if statements and for loops, also execute in the usual manner. Intel ArBB computations inside of such control flow execute as many times as the control flow body executes, and insert as many computations to be performed into the new closure. For example, a for loop containing Intel ArBB computations results in an unrolled loop containing one copy of all computations in the loop body for each iteration during capture.

In the following example, if `my_function` is captured using `arbb::capture()`, the value of the variable *use_for* at capture time determines whether the operations on the Intel ArBB scalar passed to the function are expressed using a `for` loop or manually unrolled. In either case, the same set of Intel ArBB operations is stored in the captured closure. This makes manual unrolling using Intel ArBB unnecessary.

NOTE. Intel ArBB can also unroll constructs such as `_for` loops automatically during the compilation of a closure. Still, using `for` loops can be helpful if you want to force the unrolling.

```
bool use_for = true;

void my_function(i32& a)
{
```

```
  if (use_for) {
   // Equivalent to the else case below.
   for (unsigned int i = 0; i < 4; ++i) {
    a++;
   }
  } else {
    // Equivalent to the if case above.
    a++; a++; a++; a++;
  }
}
```

This simple mechanism enables writing arbitrary C++ code that generates Intel ArBB code. As you have access to all the run-time state of your C++ program, you can decide what code to generate based on information available only at run time. For example, you can include algorithmic choices in your function being captured and pick appropriate values based on characteristics of a data set loaded at run time. You can also easily wrap a capture call in a loop and generate multiple versions of the same function to be picked at a later time, or run performance tests on multiple versions of a function and pick the fastest one.

Since you are writing plain C++ code, you can use all the modularity features of C++ to keep your program well-structured and maintainable, without paying any cost for that modularity during the execution of your closure.

If you have an interpreter for a domain-specific language, you can turn the interpreter into a compiler using Intel ArBB:

1. Implement the parts of the interpreter that execute operations using Intel ArBB types.
2. Wrap an interpretation of a program in your domain-specific language in an `arbb::capture()` call.

The resulting closure contains the code specified in your language, translated into Intel ArBB operations. This closure can be compiled and executed using the Intel ArBB run-time library.

If you are using Intel ArBB to implement a language frontend, consider using the Intel ArBB Virtual Machine (VM) layer to build VM functions. These functions are the underlying type from which closures are constructed. The VM layer provides a simple C89 (the 1989 version of the C standard, with a few later additions supported by common compilers and foreign function interfaces such as use of const) Application Programming Interface (API).

The VM provides all the semantics available through the C++ frontend. Its syntax is less convenient, but it provides a simpler interface to access from interpreters, compilers, and languages other than C++. In the VM, variables such as scalars and dense containers are dynamically typed instances of the arbb_variable_t VM type. This makes them usable from dynamically typed languages and easier to use for statically compiled frontends. All Intel ArBB C++ frontend features are built on the fully-documented virtual machine API, so you can build the same features using your language as a frontend.

Closure Type Safety and Auto Closures

The `arbb::closure` type is a class template statically typed on the captured functions that can be assigned to it. Assigning a function closure to a closure not matching that function signature results in a compile-time error because the closure types do not match. Attempting to pass arguments to a closure that do not match the parameter types of the captured function also results in a compile-time error. These type checks permit to catch errors during C++ compilation.

While the type safety of `arbb::closure` is usually desirable, you may sometimes want to defer error checking until run time, or perform assignments between closures of different types. The `arbb::auto_closure` class enables you doing this by declaring *auto closures*. As `arbb::auto_closure` class is not a class template, you

can assign to it regardless of the types of the closures involved. Attempting to assign an `arbb::auto_closure` to an `arbb::closure` with a type that does not match the captured function stored in the auto closure results in a run-time error, to preserve the type safety of `arbb::closure`.

An auto closure can be executed like a regular closure. However, calling an auto closure does not perform any compile-time checking. Attempting to call an auto closure with arguments that do not match the parameter types of the captured function results in a run-time error.

Error Handling

You can meet the following kinds of errors when you use the Intel® Array Building Blocks (Intel® ArBB) C++ frontend:

- **compile-time errors** caught by your C++ compiler; such errors cause compilation failure, when you use Intel ArBB to compile a C++ file containing a syntactical or type checking related error.
- **runtime errors** that cause an `arbb::exception` (or derived class) instance to be thrown.

Run-time Exceptions

All exceptions thrown by Intel® Array Building Blocks (Intel® ArBB) are instances of the `arbb::exception` class or its derived classes. The `arbb::exception` class itself derives from `std::exception`, so that instances of `arbb::exception` can be caught as part of standard C++ exception handling.

The only member function provided by `arbb::exception` is `arbb::exception::what()`. Like the member function from `std::exception`, it returns a `const char*` containing a descriptive message of the exception that occurred.

See the table below for the possible exceptions from Intel ArBB C++ frontend.

Exception Type	Description
`arbb::exception`	An error that does not correspond to any of the categories below occurred.
`arbb::out_of_bounds`	An attempt was made to access a container outside of its bounds.
`arbb::arithmetic_error`	An arithmetic error, such as an integer division by zero, occurred.
`arbb::bad_alloc`	A memory allocation attempt failed.
`arbb::uninitialized_access`	A variable was accessed before it was initialized with a value.
`arbb::invalid_op_within_map`	An unsupported operation was attempted inside of `arbb::map()`.
`arbb::internal_error`	An unexpected internal error occurred.

The `arbb::internal_error` exception type is thrown only when an unexpected internal error, such as an assertion failure inside of the Intel ArBB implementation, occurred. Unless a program has overwritten memory contents in an undefined manner, this exception usually indicates a bug in Intel ArBB. If you encounter an `arbb::internal_error`, please file a bug report so that we can fix the problem.

Porting C Code to Intel® Array Building Blocks

6

This book explains how C++ data space interacts with the Intel® Array Building Blocks data space and gives the examples on how to port C++ code.

Dot Product

A simple example of calculating the dot product:

Using C Loops

```
for (i = 0; i < n; i++) {
dst += src1[i] * src2[i];}
```

Using Intel® Array Building Blocks:

```
dense<f64> vsrc1; bind(src1, n);
dense<f64> vsrc2; bind(src2, n);
f64 dst = add_reduce(vsrc1 * vsrc2);
```

Black-Scholes

Black-Scholes is a well-accepted analytical model for European option pricing which is a computation-intensive financial engineering application. The pseudocode below shows the sequential C++ code to be ported to the Intel® Array Building Blocks (Intel® ArBB).

```
0  float s[n], x[n], r[n], v[n], t[n];
1  float result[n];
2  for(int i = 0; i < n; i++) {
3   float d1 = s[i] / ln(x[i]);
4   d1 += (r[i] + v[i] * v[i] * 0.5f) * t[i];
5   d1 /= sqrt(t[i]);
6   float d2 = d1 - sqrt(t[i]);
7   result[i] = x[i] * exp(r[i] * t[i]) *
8    ( 1.0f - CND(d2)) + (-s[i]) * (1.0f - CND(d1));
9 }
```

The pseudocode below shows its Intel ArBB counterpart after porting. These two pieces of code are very similar (lines 4-9, 10-12).

```
0  #include <arbb.hpp>
1  using namespace arbb;
```

```
 2  float s[n], x[n], r[n], v[n], t[n];
 3  float result[n];
 4  dense <f32> vs, vx, vr, vv, vt, vresult;
 5  bind(vs, v, n); bind(vx, x, n); bind(vr, r, n); bind(vv, v, n); bind(vt, t, n);
 6  bind(vresult, result, n);
 7  call(black_scholes)(vs, vx, vr, vv, vt, vresult);

 8  template <typename T>
 9  void black_scholes(dense<T> s, dense<T> x, dense<T> r, dense<T> v, dense<T> t,
                       dense<T>& result) {
10    dense<T>d1 = s / ln(x);
11    d1 += (r + v * v * 0.5f) * t;
12    d1 /= sqrt(t);
13    dense<T>d2 = d1 - sqrt(t);
14    result = x * exp(r * t) * (1.0f - CND(d2)) + (-s) * (1.0f - CND(d1));
15 }
```

The only differences are the following:

- Intel ArBB needs to include the `arbb.hpp` header file and use the namespace `arbb`(lines 0, 1).
- Intel ArBB adds the `dense` declarations (line 4-5) and the `call` operation (line 7).
- Intel ArBB exempts you from having to manipulate arrays in loops and subscripts. Instead, it encapsulates the application logic into a Intel ArBB function (lines 10-12).
- Intel ArBB code can co-exist well with C++'s parametric polymorphism, enabling the code to be instantiated with different types `T`.

Computing Pi

The following pseudocode computes Pi number.

```
 1  srand(time(NULL));
 2  num_inside = 0.0f;
 3  for( i = 0; i < NSET; i++ ) {
 5    // Generate x, y random numbers in [0,1)
 6   float x = float( rand() ) / float( RAND_MAX );
 7   float y = float( rand() ) / float( RAND_MAX );
 8   float distance_from_zero = sqrt( x*x + y*y);
 9   if ( distance_from_zero <= 1.0f )
10    num_inside += 1.0f;
11  else
12    num_inside += 0.0f;
13  }
14  float pi = 4.0f * ( num_inside / NSET );
```

The pseudocode below shows its Intel® Array Building Blocks (Intel® ArBB) counterpart after porting.

```
0  #include <../samples/finance/randomlib/arbb_random.hpp>
1  Uniform<> rng(NSET);
2  dense<f32> x = rng.randUnit();
3  dense<f32> y = rng.randUnit();
4  dense<f32> distance_from_zero = sqrt( x*x + y*y);
5  dense<f32> inside_circle =  select(dist_from_zero <= 1.0f, 1.0f, 0.0f);
```

```
6   f32 pi = 4.0f * (add_reduce(inside_circle) / NSET)
```

The main differences are the following:

- Intel ArBB uses Uniform Random Generator for C's `rand()` (line 1).
- Intel ArBB uses `select()` (line 5) instead of C's conditional (lines 9-12).

Binomial Tree for Options Pricing

The C code to compute binomial tree for options pricing:

```
for (int k = 0; k < NUM_OPTIONS; k++) {
 ...
 float opt_price[TIMESTEPS];
 ...
 for (int i = TIMESTEPS - 1; i >=0; i--) {
   for (int j = 0; j <= i; j++) {
     float compute_value =
         (p * opt_price[j] + (1-p) * opt_price[j+1])
          * multiplier;
     float t1 = upow_tbl[i - j] * dpow_tbl[j];
     float early_value = max(exe_price - t1, (T)0.0);

     opt_price[j] =
          max(early_value, compute_value);
  }
 }
}
```

Intel® Array Building Blocks code, red color shows the difference between C and Intel Array Building Blocks code:

```
f32 map_binomial_tree(...) {
 ...
 dense<f32> opt_price = fill(0, TIMESTEPS)
 ...
  _for (i = TIMESTEPS - 1, i >=0, i--) {
    _for (j = 0, j <= i, j++) {
      f32 compute_value = (p * opt_price[j] + (1-p) * opt_price[j+1])* multiplier;
      f32 t1 = upow_tbl[i - j] * dpow_tbl[j];
```

```
      f32 early_value = max(exe_price - t1, f32(0.0f));

     opt_price = replace(opt_price, j, max(early_value, compute_value));

  }_end_for
 }_end_for
  return f32(opt_price[0]);
}
dense<f32> options = map(map_binomial_tree)(... );
```

Monte Carlo Poisson Solver

This section shows how to port C code for Monte Carlo Poisson Solver, red color shows the difference between C and Intel® Array Building Blocks code.

C code:

```
for (i=0; i<NUM_POINTS; i++ ) {
  xx = x[i];
  yy = y[i];
  while (1) {
     min_dist = minDist(xx, yy);
     if (min_dist < acceptDist )
        break;
     double ran = double(rand())/RAND_MAX;
     xx += min_dist * sin(2*pi*ran);
     yy += min_dist * cos(2*pi*ran);
  }
}
```

Intel Array Building Blocks code:

Start porting by applying the above described porting rules:

```
Uniform<> rng(NUM_POINTS);
_while ( true ) {
 dense<f64> min_dist = minDist(xx, yy);
 ...
 dense<f64> ran = rng.randUnit();
 xx += min_dist * sin( 2*pi*ran );
 yy += min_dist * cos( 2*pi*ran );
```

```
}_end_while
```

Translate the `if... break` conditional:

```
Uniform<> rng(NUM_POINTS);
_while ( true ) {
 dense<f64> min_dist = minDist(xx, yy);
dense<bool> mask = minDist < acceptDist;
 _if(all(mask)) {
    _break;
 }_end_if
 dense<f64> ran = rng.randUnit();
 xx += min_dist * sin(2*pi*ran);
 yy += min_dist * cos(2*pi*ran);
}_end_while
```

The C program iterates different times for different `xx, yy`. In the Intel Array Building Blocks code use `select(mask, ...)` to prevent further iterations for those already with mask 1

```
Uniform<> rng(NUM_POINTS);
_while ( true ) {
 dense<f64> min_dist = minDist(xx, yy);
 dense<bool> mask = minDist < acceptDist;
 _if(all(mask)) {
    _break;
 }_end_if
 dense<f64> ran = rng.randUnit();
 xx = select(mask, xx, xx + min_dist * sin(2*pi*ran);
 yy = select(mask, yy, yy + min_dist * cos(2*pi*ran);
}_end_while
```

Different `x[i], y[i]` converge at different rates, `_if(all(mask))` approach converges at the slowest rate until the last one to satisfy `minDist < acceptDist`.

If most elements converge fast, it might be a good idea to do `pack()`, so that further iterations only happen for those with mask 0:

```
_while ( true ) {
 ...
 dense<bool> mask = minDist < acceptDist;
 _if(all(mask)) {
   _break;
 }_end_if
```

```
 xx = pack(xx, !mask);
 yy = pack(yy, !mask);
 ...
}_end_while
```

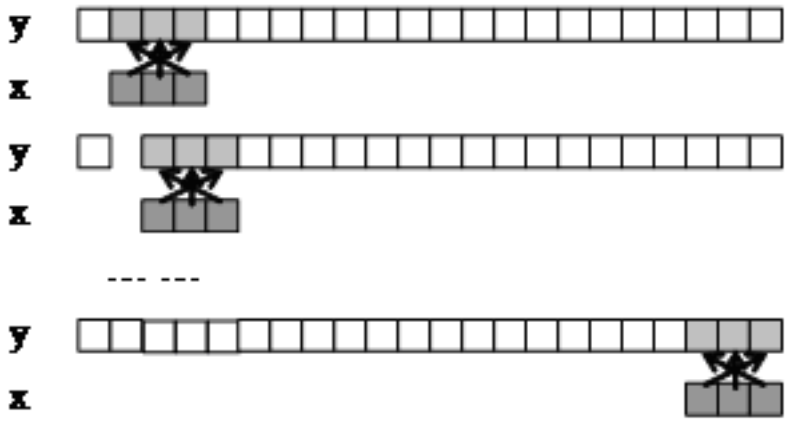
But the best solution is to use `map` function:

```
void poissonSolver(f64 &x, f64&y) {
 Uniform<> rng(1); //assuming length is 1
 _while ( true ) {
   f64 min_dist = minDist(x, y);
   _if (minDist < acceptDist) {
     _break;
   }_end_if
   f64 ran = rng.randUnit()[0];
   x += min_dist * sin(2*pi*ran);
   y += min_dist * cos(2*pi*ran);
 }_end_while
}
```

General Convolution

Convolution is a widely used function in many application domains ranging from signal/image processing to statistics and geophysics. The computation pattern of convolution is slightly trickier.

Figure below shows a typical 1D convolution algorithm, where y is a data set, and x is the kernel sliding through the data set.



The C implementation is illustrated below. As compared to Figure above, you may find the loop structure is more complicated and the array access pattern is more irregular (particularly, `y[i - j]`.

```
float x[M], y[N], z[N];

for (int i = 0; i < N; i++) {
   z[i] = 0.0f;
   for (int j = 0; j < M; j++)
      if (i >= j) {
        z[i] += x[j] * y[i-j];
     }
   }
}
```

You must map the doubly-nested loops to dense objects. Obviously, you want to abstract the data set y to be a dense object Y. At this point, you peel the outer loop and change all the instances of y to Y. This porting uses a lighter-weight operator `shift`. You can consider the kernel X fixed while the data set Y is sliding leftward, which makes the result equivalent. The optimized implementation isshown below:

```
dense<f32> X, Y, Z;

Z = fill(0.0f, N);
_for (i32 j = 0, j < M, j++) {
  Z += X[j] * shift(Y, -j);
} _end_for;
```

Note that the 1D Intel Array Building Blocks implementation can be extended to a 2D convolution implementation with very little effort as shown in Image Convolution section.

Image Convolution

The C code below performs image convolution with a generic kernel and zero boundary conditions:

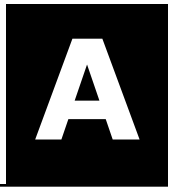
```
Void convolution(float* intput, int width, int height,
   float* kernel, int kernel_width, int kernel_height, float* output)
{
  for (int y = 0; y < height; ++y) {
   for (int x = 0; x < width; ++x) {
     float temp = 0;
     for (int i = 0; i < kernel_height; ++i) {
       for (int j = 0; j < kernel_width; ++j) {
         int yp = y + i - kernel_height / 2;
         int xp = x + j - kernel_width / 2;
         if (yp >=0 && xp >=0 && yp < height && xp < width) {
```

```
            temp += input[yp * width + xp] * kernel[i * kernel_width + j];
          }
        }
      }
      output[y * width + x] = temp;
    }
  }
}
```

It can be easily ported to Intel® Array Building Blocks (Intel® ArBB) code: allocate dense<T, 2> with the given shape and add shift. Since the size of the input and the kernel can be obtained from the dense containers there is no need to provide those values. Red color shows the difference between C and Intel ArBB code:

```
void arbb_convolution( dense<f32, 2> input, dense<f32, 2> kernel, dense<f32, 2> & output)
{
  i32 kernel_width = kernel.num_cols();
  i32 kernel_height = kernel.num_rows();
  dense<f32, 2> temp = fill(f32(0), input.num_cols(), input.num_rows());
  _for (i32 i = 0, i < kernel_height, ++i) {
    _for (i32 j = 0, j < kernel_width, ++j) {
      temp += shift(input, j - kernel_width / 2, i - kernel_height / 2) * kernel(j, i);
    } _end_for;
  }_end_for;
  output = temp;
}
```

Environment Variables



You can set the environment variables to control the execution of the Intel® Array Building Blocks (Intel® ArBB) software.

Intel ArBB Environment Variables

Environment Variable	Values
ARBB_OPT_LEVEL	O0 , O2 or O3
ARBB_VERBOSE	1/y or 0/n
ARBB_TRT_VERBOSE	1/y or 0/n
ARBB_NUM_CORES	n - specifies the max number of cores used by the software
ARBB_DECOMP_DEGREE	N - specifies the number of tasks for each parallel region
ARBB_SPAWN_JOIN_MOD	"TREE_SPAWN_REFCT_JOIN", "TREE_SPAWN_FGDEP_JOIN", "TREE_SPAWN_TREE_JOIN", "LINEAR_SPAWN_FGDEP_JOIN"
ARBB_PROF	"itt", "tp", "cm", "cm_itt", "cm_tp", "cmenh", "ittenh", "tpenh", "cm_ittenh", "cm_tpenh","cmenh_itt", "cmenh_tp"
ARBB_INIT_HEAP	initial heap size
ARBB_MAX_HEAP	maximum heap size
ARBB_DUMPJIT	"1"/"y" or "0"/"n
ARBB_JIT_OPTIONS	[-]oobcheck, [-]scatter_index_check [-]immediate_copy_in [-]allow_fma_fusing [-]allow_fp_reassoc

Environment Variable	Values
ARBB_RT_OPTIONS	special run-time options
ARBB_EXCEPTION_MODE	0-2

ARBB_OPT_LEVEL

Sets the level of optimization. O0 enables no runtime optimizations, and uses interpretations. O2 enables vectorization. O3 enables vectorization and thread parallelization.

The default value of ARBB_OPT_LEVEL is O2.

ARBB_VERBOSE

When ARBB_VERBOSE is turned on, the Intel ArBB compiler/runtime displays information that is related to execution traits and performance.

ARBB_TRT_VERBOSE is a special case of ARBB_VERBOSE, focused only on the threading runtime.

ARBB_PROF

When ARBB_PROF is set to a value that includes `itt` or `tp`, thread profiling is enabled. Information is collected during the runtime that may be viewed with VTune"™s thread profiling facilities. When it is set to a value that includes `cm`, the software counter monitoring facilities are enabled. There are enhanced versions of each of these, indicated by a enh suffix , which turns on enhanced functionality.

ARBB_INIT_HEAP, ARBB_MAX_HEAP

Specifies the heap sizes. They must be a decimal number with a suffix in the set `{k, K, m, M}`.

ARBB_DUMPJIT

When ARBB_DUMPJIT is turned on, the Intel ArBB compiler dumps the intermediate representations and generated code with different optimization levels in the working directory.

ARBB_JIT_OPTIONS

You can turn on or off some optimization options by setting the ARBB_JIT_OPTIONS variable. These settings override the default set specified by the ARBB_OPT_LEVEL variable. If a "-" precedes a setting, it turns off the option.

The possible values of the ARBB_JIT_OPTIONS variable:

`oobcheck` - turn on the bounds checks for scatter/gather operations.

`scatter_index_check` - turn on the index duplication checking for scatter indices, to limit non-determinism.

NOTE. This checking may take a long time, use this option only to debug for correctness.

`immediate_copy_in` - turn on the `copy_in` operator to do an immediate copy instead of lazy copy to enforce greater separation between the C/C++ and Intel ArBB data spaces .

Binding constructors do not support this option .

ARBB_RT_OPTIONS

Specifies one or more runtime options. Now only one option is supported: initial and maximum heap size settings. For example, ARBB_RT_OPTIONS=AMM:`init=96m|max=96m`. `init` and `max` must be lower case.

ARBB_EXCEPTION_MODE

Specifies exception modes: 0 throws an exception, 1 only prints it, and 2 both prints and throws the exception.

Index

S

T

U

W