# Intel® ISPC User's Guide


**Resources**

- GitHub page
- Discussions on GitHub
- Issues on Github
- Release planning board
- Contributing guide
- Wiki on Github


The Intel® Implicit SPMD Program Compiler (Intel® ISPC) is a compiler for writing SPMD (single program multiple data) programs to run on the CPU and GPU. The SPMD programming approach is widely known to graphics and GPGPU programmers; it is used for GPU shaders and CUDA* and OpenCL* kernels, for example. The main idea behind SPMD is that one writes programs as if they were operating on a single data element (a pixel for a pixel shader, for example), but then the underlying hardware and runtime system executes multiple invocations of the program in parallel with different inputs (the values for different pixels, for example).

The main goals behind `ispc` are to:

- Build a variant of the C programming language that delivers good performance to performance-oriented programmers who want to run SPMD programs on CPUs and GPUs.
- Provide a thin abstraction layer between the programmer and the hardware--in particular, to follow the lesson from C for serial programs of having an execution and data model where the programmer can cleanly reason about the mapping of their source program to compiled assembly language and the underlying hardware.
- Harness the computational power of the Single Program, Multiple Data (SIMD) vector units without the extremely low-programmer-productivity activity of directly writing intrinsics.
- Explore opportunities from close-coupling between C/C++ application code and SPMD `ispc` code running on the same processor--lightweight function calls between the two languages, sharing data directly via pointers without copying or reformatting, etc.

**We are very interested in your feedback and comments about ispc and in hearing your experiences using the system. We are especially interested in hearing if you try using ispc but see results that are not as you were expecting or hoping for.** We encourage you to send a note with your experiences or comments to the GitHub Discussions forum or to file bug or feature requests with the `ispc` bug tracker. (Thanks!)

Contents:

## Recent Changes to ISPC

See the file ReleaseNotes.txt in the `ispc` distribution for a list of recent changes to the compiler.

### Updating ISPC Programs For Changes In ISPC 1.1

The major changes introduced in the 1.1 release of `ispc` are first-class support for pointers in the language and new parallel loop constructs. Adding this functionality required a number of syntactic changes to the language. These changes should generally lead to straightforward minor modifications of existing `ispc` programs.

These are the relevant changes to the language:

- The syntax for reference types has been changed to match C++'s syntax for references and the `reference` keyword has been removed. (A diagnostic message is issued if `reference` is used.)
    - Declarations like `reference float foo` should be changed to `float &foo`.
    - Any array parameters in function declaration with a `reference` qualifier should just have `reference` removed: `void foo(reference float bar[])` can just be `void foo(float bar[])`.
- It is now a compile-time error to assign an entire array to another array.
- A number of standard library routines have been updated to take pointer-typed parameters, rather than references or arrays an index offsets, as appropriate. For example, the `atomic_add_global()` function previously took a reference to the variable to be updated atomically but now takes a pointer. In a similar fashion, `packed_store_active()` takes a pointer to a `uniform unsigned int` as its first parameter rather than taking a `uniform unsigned int[]` as its first parameter and a `uniform int` offset as its second parameter.
- It is no longer legal to pass a varying lvalue to a function that takes a reference parameter; references can only be to uniform lvalue types. In this case, the function should be rewritten to take a varying pointer parameter.
- There are new iteration constructs for looping over computation domains, `foreach` and `foreach_tiled`. In addition to being syntactically cleaner than regular `for` loops, these can provide performance benefits in many cases when iterating over data and mapping it to program instances. See the Section Parallel Iteration Statements: "foreach" and "foreach_tiled" for more information about these.

### Updating ISPC Programs For Changes In ISPC 1.2

The following changes were made to the language syntax and semantics for the `ispc` 1.2 release:

- Syntax for the "launch" keyword has been cleaned up; it's now no longer necessary to bracket the launched function call with angle brackets. (In other words, now use `launch foo();`, rather than `launch < foo() >;`.)
- When using pointers, the pointed-to data type is now "uniform" by default. Use the varying keyword to specify varying pointed-to types when needed. (i.e. `float *ptr` is a varying pointer to uniform float data, whereas previously it was a varying pointer to varying float values.) Use `varying float *` to specify a varying pointer to varying float data, and so forth.
- The details of "uniform" and "varying" and how they interact with struct types have been cleaned up. Now, when a struct type is declared, if the struct elements don't have explicit "uniform" or "varying" qualifiers, they are said to have "unbound" variability. When a struct type is instantiated, any unbound variability elements inherit the variability of the parent struct type. See Struct Types for more details.
- `ispc` has a new language feature that makes it much easier to use the efficient "(array of) structure of arrays" (AoSoA, or SoA) memory layout of data. A new `soa<n>` qualifier can be applied to structure types to specify an n-wide SoA version of the corresponding type. Array indexing and pointer operations with arrays SoA types automatically handles the two-stage indexing calculation to access the data. See Structure of Array Types for more details.

### Updating ISPC Programs For Changes In ISPC 1.3

This release adds a number of new iteration constructs, which in turn use new reserved words: `unmasked`, `foreach_unique`, `foreach_active`, and `in`. Any program that happens to have a variable or function with one of these names must be modified to rename that symbol.

### Updating ISPC Programs For Changes In ISPC 1.5.0

This release adds support for double precision floating point constants. Double precision floating point constants are floating point number with `d` suffix and optional exponent part. Here are some examples: 3.14d, 31.4d-1, 1.d, 1.0d, 1d-2. Note that floating point number without suffix is treated as single precision constant.

### Updating ISPC Programs For Changes In ISPC 1.6.0

This release adds support for Operators Overloading, so a word `operator` becomes a keyword and it potentially creates a conflict with existing user function. Also a new library function packed_store_active2()

was introduced, which also may create a conflict with existing user functions.

### Updating ISPC Programs For Changes In ISPC 1.7.0

This release contains several changes that may affect compatibility with older versions:

- The algorithm for selecting overloaded functions was extended to cover more types of overloading, and handling of reference types was fixed. At the same time the old scheme, which blindly used the function with "the best score" summed for all arguments, was switched to the C++ approach, which requires "the best score" for each argument. If the best function doesn't exist, a warning is issued in this version. It will be turned into an error in the next version. A simple example: Suppose we have two functions: max(int, int) and max(unsigned int, unsigned int). The new rules lead to an error when calling max(int, unsigned int), as the best choice is ambiguous.
- Implicit cast of pointer to const type to void* was disallowed. Use explicit cast if needed.
- A bug which prevented "const" qualifiers from appearing in emitted .h files was fixed. Consequently, "const" qualifiers now properly appearing in emitted .h files may cause compile errors in pre-existing codes.
- get_ProgramCount() was moved from stdlib to examples/util/util.isph file. You need to include this file to be able to use this function.

### Updating ISPC Programs For Changes In ISPC 1.8.2

The release doesn't contain language changes, which may affect compatibility with older versions. Though you may want be aware of the following:

- Mangling of uniform types was changed to not include varying width, so now you may use uniform structures and pointers to uniform types as return types in export functions in multi-target compilation.

### Updating ISPC Programs For Changes In ISPC 1.9.0

The release doesn't contains language changes, which may affect compatibility with older versions. It introduces new AVX512 target: avx512knl-i32x16.

### Updating ISPC Programs For Changes In ISPC 1.9.1

The release doesn't contains language changes, which may affect compatibility with older versions. It introduces new AVX512 target: avx512skx-i32x16.

### Updating ISPC Programs For Changes In ISPC 1.9.2

The release doesn't contain language changes, which may affect compatibility with older versions.

### Updating ISPC Programs For Changes In ISPC 1.10.0

The release has several new language features, which do not affect compatibility. Namely, new streaming stores, aos_to_soa/soa_to_aos instrinsics for 64 bit types, and a "#pragma ignore".

One change that potentially may affect compatibility - changed size of short vector types. If you use short vector types for data passed between C/C++ and ISPC, you may want to pay attention to it.

### Updating ISPC Programs For Changes In ISPC 1.11.0

This release redefined -O1 compiler option to optimize for size, so it may require adjusting your build system accordingly.

Starting 1.11.0 version auto-generated headers use `#pragma once`. In the unlikely case when your C/C++ compiler is not supporting that, please use `--no-pragma-once` `ispc` switch.

This release also introduces new AVX512 target avx512skx-i32x8. It produces code, which doesn't use ZMM registers.

### Updating ISPC Programs For Changes In ISPC 1.12.0

This release contains the following changes that may affect compatibility with older versions:

- `noinline` keyword was added.
- Standard library functions `rsqrt_fast()` and `rcp_fast()` were added.
- AVX1.1 (IvyBridge) targets and generic KNC and KNL targets were removed. Note that KNL is still supported through avx512knl-i32x16.

The release also introduces static initialization for varying variables, which should not affect compatibility.

This release introduces experimental cross OS compilation support and ARM/AARCH64 support. It also contains a new 128-bit AVX2 target (avx2-i32x4) and a CPU definition for Ice Lake client (--device=icl).

### Updating ISPC Programs For Changes In ISPC 1.13.0

This release contains the following changes that may affect compatibility with older versions:

- Representation of `bool` type in storage was changed from target-specific to one byte per boolean value. So size of `varying bool` is target width (in bytes), and size of `unform bool` is one. This definition is compatible with C/C++, hence improves interoperability.
- type aliases for unsigned types were added: `uint8`, `uint16`, `uint32`, `uint64`, and `uint`. To detect if these types are supported you can check if ISPC_UINT_IS_DEFINED macro is defined, this is handy for writing code which works with older versions of `ispc`.
- `extract()`/`insert()` for boolean arguments, and `abs()` for all integer and FP types were added to standard library.

### Updating ISPC Programs For Changes In ISPC 1.14.0

This release contains the following changes that may affect compatibility with older versions:

- "generic" targets were removed. Please use native targets instead.

New i8 and i16 targets were introduced: avx2-i8x32, avx2-i16x16, avx512skx-i8x64, and avx512skx-i16x32.

Windows x86_64 target now supports `__vectorcall` calling convention. It's off by default, can be enabled by `--vectorcall` command line switch.

### Updating ISPC Programs For Changes In ISPC 1.14.1

The release doesn't contain language changes, which may affect compatibility with older versions.

### Updating ISPC Programs For Changes In ISPC 1.15.0

The release has several new language features, which do not affect compatibility. Namely, packed_[load|store]_active() stdlib functions for 64 bit types, and loop unroll pragmas: "#pragma unroll" and "#pragma nounroll".

### Updating ISPC Programs For Changes In ISPC 1.16.0

The release has several new functions in the standard library, that can possbily affect compatibility:

- `alloca()` - refer to Stack Memory Allocation for more details.
- `assume()` - refer to Compiler Optimization Hints for more details.
- `trunc()` - refer to Basic Math Functions for more details.

The language got experimental feature for calling LLVM intrinsics. This should not affect compatibility with existing programs. See LLVM Intrinsic Functions for more details.

### Updating ISPC Programs For Changes In ISPC 1.17.0

The release introduces new data type `float16` and floating point literals with `f16` suffix.

For the sake of unification with C/C++, capital letter X may be used in hexadecimal prefix (`0X`) and capital letter P as a separator for exponent in hexadecimal floating point. For example: `0X1P16`.

The naming of Xe targets, architectures, device names has changed.

Standard library library got new `prefetchw_{l1,l2,l3}()` intrinsics for prefetching in anticipation of write.

The algorithms used for implementation of `rsqrt(double)` and `rcp(double)` standard library functions have changed on AVX512 and may affect the existing code.

### Updating ISPC Programs For Changes In ISPC 1.18.0

AVX512 targets were renamed to drop "base type" (or "mask size"), old naming is accepted for compatibility. New names are avx512skx-x4, avx512skx-x8, avx512skx-x16, avx512skx-x32, avx512skx-x64, and avx512knl-x16.

Standard library gained full support for `float16` type. Note that it is fully supported only on the targets with native hardware support. On the other targets emulation is still not guaranteed, but may work in some cases.

The compiler gained support for `-E` switch for running preprocessor only, which is similar to the switch of C/C++ compilers. Also, as a result of bug fix, in case of preprocessor error, the compiler will crash now. It used not to crash and produced some output (sometimes correct!). As it was a convenient feature for some users running experiments in isolated environment (like ignoring missing includes when compiling of Compiler Explorer), `--ignore-preprocessor-errors` switch was added to preserve this behavior.

## Getting Started with ISPC

### Installing ISPC

The ispc downloads web page has prebuilt executables for Windows*, Linux* and macOS* available for download. Alternatively, you can download the source code from that page and build it yourself; see the ispc wiki for instructions about building `ispc` from source.

Once you have an executable for your system, copy it into a directory that's in your PATH. Congratulations--you've now installed `ispc`.

### Compiling and Running a Simple ISPC Program

The directory `examples/simple` in the `ispc` distribution includes a simple example of how to use `ispc` with a short C++ program. See the file `simple.ispc` in that directory (also reproduced here.)

```
export void simple(uniform float vin[], uniform float vout[],
                   uniform int count) {
    foreach (index = 0 ... count) {
        float v = vin[index];
        if (v < 3.)
            v = v * v;
        else
            v = sqrt(v);
        vout[index] = v;
```

```
    }
}
```

This program loops over an array of values in `vin` and computes an output value for each one. For each value in `vin`, if its value is less than three, the output is the value squared, otherwise it's the square root of the value.

The first thing to notice in this program is the presence of the `export` keyword in the function definition; this indicates that the function should be made available to be called from application code. The `uniform` qualifiers on the parameters to `simple` indicate that the corresponding variables are non-vector quantities--this concept is discussed in detail in the "uniform" and "varying" Qualifiers section.

Each iteration of the `foreach` loop works on a number of input values in parallel--depending on the compilation target chosen, it may be 4, 8, 16, 32, or even 64 elements of the `vin` array, processed efficiently with the CPU's or GPU's SIMD hardware. Here, the variable `index` takes all values from 0 to `count-1`. After the load from the array to the variable `v`, the program can then proceed, doing computation and control flow based on the values loaded. The result from the running program instances is written to the `vout` array before the next iteration of the `foreach` loop runs.

To build and run examples go to `examples` and create `build` folder. Run `cmake -DISPC_EXECUTABLE=<path_to_ispc_binary> ../`. On Linux* and macOS*, the makefile will be generated in that directory. On Windows*, Microsoft Visual Studio solution `ispc_examples.sln` will be created. In either case, build it now! We'll walk through the details of the compilation steps in the following section, Using The ISPC Compiler.) In addition to compiling the `ispc` program, in this case the `ispc` compiler also generates a small header file, `simple.h`. This header file includes the declaration for the C-callable function that the above `ispc` program is compiled to. The relevant parts of this file are:

```
#ifdef __cplusplus
extern "C" {
#endif // __cplusplus
    extern void simple(float vin[], float vout[], int32_t count);
#ifdef __cplusplus
}
#endif // __cplusplus
```

It's not mandatory to `#include` the generated header file in your C/C++ code (you can alternatively use a manually-written `extern` declaration of the `ispc` functions you use), but it's a helpful check to ensure that the function signatures are as expected on both sides.

Here is the main program, `simple.cpp`, which calls the `ispc` function above.

```
#include <stdio.h>
#include "simple.h"

int main() {
    float vin[16], vout[16];
    for (int i = 0; i < 16; ++i)
        vin[i] = i;

    simple(vin, vout, 16);

    for (int i = 0; i < 16; ++i)
        printf("%d: simple(%f) = %f\n", i, vin[i], vout[i]);
}
```

Note that the call to the `ispc` function in the middle of `main()` is a regular function call. (And it has the same overhead as a C/C++ function call, for that matter.)

When the executable `simple` runs, it generates the expected output:

```
0: simple(0.000000) = 0.000000
1: simple(1.000000) = 1.000000
2: simple(2.000000) = 4.000000
3: simple(3.000000) = 1.732051
...
```

For a slightly more complex example of using `ispc`, see the Mandelbrot set example page on the `ispc` website for a walk-through of an `ispc` implementation of that algorithm. After reading through that example, you may want to examine the source code of the various examples in the `examples/` directory of the `ispc` distribution.

## Using The ISPC Compiler

To go from a `ispc` source file to an object file that can be linked with application code, enter the following command

```
ispc foo.ispc -o foo.o
```

(On Windows, you may want to specify `foo.obj` as the output filename.)

### Basic Command-line Options

The `ispc` executable can be run with `--help` to print a list of accepted command-line arguments. By default, the compiler compiles the provided program (and issues warnings and errors), but doesn't generate any output.

If the `-o` flag is given, it will generate an output file (a native object file by default).

```
ispc foo.ispc -o foo.obj
```

To generate a text assembly file, pass `--emit-asm`:

```
ispc foo.ispc -o foo.s --emit-asm
```

To generate LLVM bitcode, use the `--emit-llvm` flag. To generate LLVM bitcode in textual form, use the `--emit-llvm-text` flag.

To run only the preprocessor, use the `-E` flag.

```
ispc foo.ispc -E -o foo.i
ispc foo.ispc -E -o foo.ispi
```

In this mode, the output will be directed to `stdout` if no output file is specified. The standard suffixes `.i` or `.ispi` are assumed for preprocessor output.

By default the compilation will fail if preprocessor encountered an error. To ignore the preprocessor errors and proceed with normal compilation flow, `--ignore-preprocessor-errors` switch may be used.

Optimizations are on by default; they can be turned off with `-O0`:

```
ispc foo.ispc -o foo.obj -O0
```

There is support for generating debugging symbols; this is enabled with the `-g` command-line flag. Using `-g` doesn't affect optimization level; to debug unoptimized code pass `-O0` flag.

The `-h` flag can also be used to direct `ispc` to generate a C/C++ header file that includes C/C++ declarations of the C-callable `ispc` functions and the types passed to it.

The `-D` option can be used to specify definitions to be passed along to the pre-processor, which runs over the program input before it's compiled. For example, including `-DTEST=1` defines the pre-processor symbol `TEST` to have the value `1` when the program is compiled.

The compiler issues a number of performance warnings for code constructs that compile to relatively inefficient code. These warnings can be silenced with the `--wno-perf` flag (or by using `--woff`, which turns off all compiler warnings.) Furthermore, `--werror` can be provided to direct the compiler to treat any warnings as errors.

Position-independent code (for use in shared libraries) is generated if the `--pic` command-line argument is provided.

### Selecting The Compilation Target

There are four options that affect the compilation target: `--arch`, which sets the target architecture, `--device` (also may be spelled as `--cpu`), which sets the target CPU or GPU, `--target`, which sets the target instruction set, and `--target-os`, which sets the target operating system.

If none of these options is specified, `ispc` generates code for the host OS and for the architecture of the system the compiler is running on (i.e. 64-bit x86-64 (`--arch=x86-64`) on x86 systems and ARM NEON on ARM systems.

To compile to a 32-bit x86 target, for example, supply `--arch=x86` on the command line:

```
ispc foo.ispc -o foo.obj --arch=x86
```

To compile for Intel Xe LP platform:

```
ispc foo.ispc -o foo.bin --target=xelp-x16 --device=tgllp --emit-zebin
```

Currently-supported architectures are `x86`, `x86-64`, `xe32`, `xe64`, `arm`, and `aarch64`.

The target CPU determines both the default instruction set used as well as which CPU architecture the code is tuned for. `ispc --help` provides a list of all of the supported CPUs. By default, the CPU type of the system on which you're running `ispc` is used to determine the target CPU.

```
ispc foo.ispc -o foo.obj --device=corei7-avx
```

Next, `--target` selects the target instruction set. For targets without hardware support for masking, the target string is of the form `[ISA]-i[mask size]x[gang size]`. For example, `--target=avx2-i32x16` specifies a target with the AVX2 instruction set, a mask size of 32 bits, and a gang size of 16. For targets with hardware masking support, which are AVX512 and GPU targets, the target string is of the form `[ISA]-x[gang size]`. For example, `--target=xehpg-x16` specifies Intel XeHPG as a target ISA and defines a gang size of 16.

By default, the target instruction set is chosen based on the most capable one supported by the system on which you're running `ispc`. In this case a warning will be issued noting the target used for compilation. It is recommended to always use `--target` switch to explicitly specify the target.

To get the complete list of supported targets, please use `--help` switch and note the list in the description of `--target`, or use `--support-matrix` switch, which will give the complete information of supported combinations of target, arch and target OS.

The following target ISAs are supported:

| Target | Description |
|---|---|
| avx, avx1 | AVX (2010-2011 era Intel CPUs) |
| avx2 | AVX 2 target (2013- Intel "Haswell" CPUs) |
| avx512knl | AVX 512 target (Xeon Phi chips codename Knights Landing) |
| avx512skx | AVX 512 target (future Xeon CPUs) |
| neon | ARM NEON |
| sse2 | SSE2 (early 2000s era x86 CPUs) |
| sse4 | SSE4 (generally 2008-2010 Intel CPUs) |
| gen9 | Intel Gen9 GPU |
| xehpg | Intel XeHPG GPU |
| xelp | Intel XeLP GPU |

Consult your CPU's manual for specifics on which vector instruction set it supports.

The mask size may be 8, 16, 32, or 64 bits, though not all combinations of ISA and mask size are supported. For best performance, the best general approach is to choose a mask size equal to the size of the most common datatype in your programs. For example, if most of the computations are done using 32-bit floating-point values, an `i32` target is appropriate. However, if you're mostly doing computation with 8-bit data types, `i8` is a better choice.

See Basic Concepts: Program Instances and Gangs of Program Instances for more discussion of the "gang size" and its implications for program execution.

The naming scheme for compilation targets changed in August 2013; the following table shows the relationship between names in the old scheme and in the new scheme:

| Target | Former Name |
|---|---|
| avx1-i32x8 | avx, avx1 |
| avx1-i32x16 | avx-x2 |
| avx2-i32x8 | avx2 |
| avx2-i32x16 | avx2-x2 |
| neon-8 | n/a |
| neon-16 | n/a |
| neon-32 | n/a |
| sse2-i32x4 | sse2 |
| sse2-i32x8 | sse2-x2 |
| sse4-i32x4 | sse4 |
| sse4-i32x8 | sse4-x2 |
| sse4-i8x16 | n/a |
| sse4-i16x8 | n/a |

Finally, `--target-os` selects the target operating system. Depending on your host `ispc` may support Windows, Linux, macOS, Android, iOS and PS4/PS5 targets. Running `ispc --help` and looking at the output for the `--target-os` option gives the list of supported targets. By default `ispc` produces the code for your host operating system.

```
ispc foo.ispc -o foo.obj --target-os=android
```

Note that cross OS compilation is in experimental stage. We encourage you to try it and send us a note with your experiences or to file a bug or feature requests with the `ispc` bug tracker.

### Selecting 32 or 64 Bit Addressing

By default, `ispc` uses 32-bit arithmetic for performing addressing calculations, even when using a 64-bit compilation target like x86-64. This implementation approach can provide substantial performance benefits by reducing the cost of addressing calculations. (Note that pointers themselves are still maintained as 64-bit quantities for 64-bit targets.)

If you need to be able to address more than 4GB of memory from your `ispc` programs, the `--addressing=64` command-line argument can be provided to cause the compiler to generate 64-bit arithmetic for addressing calculations. Note that it is safe to mix object files where some were compiled with the default `--addressing=32` and others were compiled with `--addressing=64`.

### The Preprocessor

`ispc` automatically runs the C preprocessor on your input program before compiling it. Thus, you can use `#ifdef`, `#define`, and so forth in your ispc programs. (This functionality can be disabled with the `--nocpp` command-line argument.)

A number of preprocessor symbols are automatically defined before the preprocessor runs:

Predefined Preprocessor symbols and their values

| Symbol name | Value | Description |
|---|---|---|
| ISPC | 1 | Enables detecting that the `ispc` compiler is processing the file |
| ISPC_TARGET_{NEON, SSE2, SSE4, AVX, AVX2, AVX512KNL, AVX512SKX} | 1 | One of these will be set, depending on the compilation target |
| ISPC_POINTER_SIZE | 32 or 64 | Number of bits used to represent a pointer for the target architecture |
| ISPC_MAJOR_VERSION | 1 | Major version of the `ispc` compiler/language |
| ISPC_MINOR_VERSION | 13 | Minor version of the `ispc` compiler/language |
| PI | 3.1415926535 | Mathematics |
| TARGET_WIDTH | Vector width of the target, e.g., 8 for sse2-i32x8 | Can be used for code versioning for static varying initialization |
| TARGET_ELEMENT_WIDTH | Element width in bytes, e.g., 4 for i32 | Can be used for code versioning for static varying initialization |
| ISPC_UINT_IS_DEFINED | 1 | The macro is defined if uint8/uint16/uint32/uint64 types are defined in the `ispc` (it's defined in 1.13.0 and later) |
| ISPC_FP64_SUPPORTED | 1 | The macro is defined if double type is supported by the target |
| ISPC_LLVM_INTRINSICS_ENABLED | 1 | The macro is defined if LLVM intrinsics support is enabled |

`ispc` supports the following `#pragma` directives.

`#pragma ignore warning` directives direct the compiler to ignore compiler warnings for individual lines.

`#pragma ignore warning` directives and their functions:

| `#pragma` name | Use |
|---|---|
| `#pragma ignore warning(all)` | Turns off all `ispc` compiler warnings including performance warnings for the following line of code. |
| `#pragma ignore warning(perf)` | Turns off only performance warnings for the following line of code. |
| `#pragma ignore warning` | Turns off all `ispc` compiler warnings including performance warnings for the following line of code. |

When using `#pragma ignore warning` before a call to a macro, it suppresses warnings from the expanded macro code.

`#pragma unroll` and `#pragma nounroll` directives provide loop unrolling optimization hints to the compiler. This pragma is placed immediately before a loop statement. Currently, this functionality is limited to uniform `for` and `do-while`.

`#pragma unroll` and `#pragma nounroll` directives and their functions:

| `#pragma` name | Use |
|---|---|
| `#pragma unroll COUNT` | Directs the loop unroller to unroll the loop `COUNT` times. The parameter may optionally be enclosed in parentheses: `#pragma unroll (COUNT)`. |
| `#pragma unroll` | Directs the loop unroller to fully unroll the loop if possible. |
| `#pragma nounroll` | Directs the loop unroller to not unroll the loop. |

### Debugging

The `-g` command-line flag can be supplied to the compiler, which causes it to generate debugging symbols. The debug info is emitted in DWARF format on Linux* and macOS*. The version of the DWARF can be controlled by command-line switch `--dwarf-version={2,3,4}`. On Windows* CodeView format is used (not PDB), it's natively supported by Microsoft Visual Studio*. Running `ispc` programs in the debugger, setting breakpoints, printing out variables is just the same as debugging C/C++ programs. Similarly, you can directly step up and down the call stack between `ispc` code and C/C++ code.

One limitation of the current debugging support is that the debugger provides a window into an entire gang's worth of program instances, rather than just a single program instance. (These concepts will be introduced shortly, in Basic Concepts: Program Instances and Gangs of Program Instances ). Thus, when a `varying` variable is printed, the values for each of the program instances are displayed. Along similar lines, the path the debugger follows through program source code passes each statement that any program instance wants to execute (see Control Flow Within A Gang for more details on control flow in `ispc`.)

While debugging, a variable, `__mask`, is available to provide the current program execution mask at the current point in the program

Another option for debugging is to use the `print` statement for `printf()` style debugging. (See Output Functions for more information.) You can also use the ability to call back to application code at particular points in the program, passing a set of variable values to be logged or otherwise analyzed from there.

### Other ways of passing arguments to ISPC

In addition to specifying arguments on the command line, if the `ISPC_ARGS` environment variable has been set it is split into arguments and these arguments are appended to any provided on the command line.

It is also possible to pass arguments to `ispc` in a file. If an argument has the form `@<filename>`, where `<filename>` exists and is readable, it is replaced with the content of the file split into arguments. Note that it *is* allowed for a file to contain a further `@<filename>` argument.

Where a file or environment variable is split into arguments, this is done based on the arguments being separated by one or more whitespace characters, including tabs and newlines. There is no means of escaping or quoting a character to allow an argument to contain a whitespace character.

## The ISPC Parallel Execution Model

Though `ispc` is a C-based language, it is inherently a language for parallel computation. Understanding the details of `ispc`'s parallel execution model that are introduced in this section is critical for writing efficient and correct programs in `ispc`.

`ispc` supports two types of parallelism: task parallelism to parallelize across multiple processor cores and SPMD parallelism to parallelize across the SIMD vector lanes on a single core. Most of this section focuses on SPMD parallelism, but see Tasking Model at the end of this section for discussion of task parallelism in `ispc`.

This section will use some snippets of `ispc` code to illustrate various concepts. Given `ispc`'s relationship to C, these should be understandable on their own, but you may want to refer to the The ISPC Language section for details on language syntax.

### Basic Concepts: Program Instances and Gangs of Program Instances

Upon entry to a `ispc` function called from C/C++ code, the execution model switches from the application's serial model to `ispc`'s execution model. Conceptually, a number of `ispc` *program instances* start running concurrently. The group of running program instances is a called a *gang* (harkening to "gang scheduling", since `ispc` provides certain guarantees about the control flow coherence of program instances running in a gang, detailed in Gang Convergence Guarantees.) An `ispc` program instance is thus similar to a CUDA* "thread" or an OpenCL* "work-item", and an `ispc` gang is similar to a CUDA* "warp".

An `ispc` program expresses the computation performed by a gang of program instances, using an "implicit parallel" model, where the `ispc` program generally describes the behavior of a single program instance, even though a gang of them is actually executing together. This implicit model is the same that is used for shaders in programmable graphics pipelines, OpenCL* kernels, and CUDA*. For example, consider the following `ispc` function:

```
float func(float a, float b) {
     return a + b / 2.;
}
```

In C, this function describes a simple computation on two individual floating-point values. In `ispc`, this function describes the computation to be performed by each program instance in a gang. Each program instance has distinct values for the variables `a` and `b`, and thus each program instance generally computes a different result when executing this function.

The gang of program instances starts executing in the same hardware thread and context as the application code that called the `ispc` function; no thread creation or context switching is done under the covers by `ispc`. Rather, the set of program instances is mapped to the SIMD lanes of the current processor, leading to excellent utilization of hardware SIMD units and high performance.

The number of program instances in a gang is relatively small; in practice, it's no more than 2-4x the native SIMD width of the hardware it is executing on. Thus, four or eight program instances in a gang on a CPU using the 4-wide SSE instruction set, eight or sixteen on a CPU using 8-wide AVX/AVX2, eight, sixteen, thirty two, or sixty four on AVX512 CPU, and eight or sixteen on a Intel GPU.

### Control Flow Within A Gang

Almost all the standard control-flow constructs are supported by `ispc`; program instances are free to follow different program execution paths than other ones in their gang. For example, consider a simple `if` statement in `ispc` code:

```
float x = ..., y = ...;
if (x < y) {
   // true statements
}
else {
   // false statements
}
```

In general, the test `x < y` may have different result for different program instances in the gang: some of the currently running program instances want to execute the statements for the "true" case and some want to execute the statements for the "false" case.

Complex control flow in `ispc` programs generally works as expected, computing the same results for each program instance in a gang as would have been computed if the equivalent code ran serially in C to compute each program instance's result individually. However, here we will more precisely define the execution model for control flow in order to be able to precisely define the language's behavior in specific situations.

We will specify the notion of a *program counter* and how it is updated to step through the program, and an *execution mask* that indicates which program instances want to execute the instruction at the current program counter. The program counter is shared by all of the program instances in the gang; it points to a single instruction to be executed next. The execution mask is a per-program-instance boolean value that indicates whether or not side effects from the current instruction should effect each program instance. Thus, for example, if a statement were to be executed with an "all off" mask, there should be no observable side-effects.

Upon entry to an `ispc` function called by the application, the execution mask is "all on" and the program counter points at the first statement in the function. The following two statements describe the required behavior of the program counter and the execution mask over the course of execution of an `ispc` function.

> 1. The program counter will have a sequence of values corresponding to a conservative execution path through the function, wherein if *any* program instance wants to execute a statement, the program counter will pass through that statement.
>
> 2. At each statement the program counter passes through, the execution mask will be set such that its value for a particular program instance is "on" if and only if the program instance wants to execute that statement.

Note that these definitions provide the compiler some latitude; for example, the program counter is allowed to pass through a series of statements with the execution mask "all off" because doing so has no observable side-effects.

Elsewhere, we will speak informally of the *control flow coherence* of a program; this notion describes the degree to which the program instances in the gang want to follow the same control flow path through a function (or, conversely, whether most statements are executed with a "mostly on" execution mask or a "mostly off" execution mask.) In general, control flow divergence leads to reductions in SIMD efficiency (and thus performance) as different program instances want to perform different computations.

### Control Flow Example: If Statements

As a concrete example of the interplay between program counter and execution mask, one way that an `if` statement like the one in the previous section can be represented is shown by the following pseudo-code compiler output:

```
float x = ..., y = ...;
bool test = (x < y);
mask originalMask = get_current_mask();
set_mask(originalMask & test);
if (any_mask_entries_are_enabled()) {
  // true statements
}
set_mask(originalMask & ~test);
if (any_mask_entries_are_enabled()) {
  // false statements
}
set_mask(originalMask);
```

In other words, the program counter steps through the statements for both the "true" case and the "false" case, with the execution mask set so that no side-effects from the true statements affect the program instances that want to run the false statements, and vice versa. However, a block of statements does not execute if the mask is "all off" upon entry to that block. The execution mask is then restored to the value it had before the `if` statement.

### Control Flow Example: Loops

`for`, `while`, and `do` statements are handled in an analogous fashion. The program counter continues to run additional iterations of the loop until all of the program instances are ready to exit the loop.

Therefore, if we have a loop like the following:

```
int limit = ...;
for (int i = 0; i < limit; ++i) {
    ...
}
```

where `limit` has the value 1 for all of the program instances but one, and has value 1000 for the other one, the program counter will step through the loop body 1000 times. The first time, the execution mask will be all on (assuming it is all on going into the `for` loop), and the remaining 999 times, the mask will be off except for the program instance with a `limit` value of 1000. (This would be a loop with poor control flow coherence!)

A `continue` statement in a loop may be handled either by disabling the execution mask for the program instances that execute the `continue` and then continuing to step the program counter through the rest of the loop, or by jumping to the loop step statement, if all program instances are disabled after the `continue` has executed. `break` statements are handled in a similar fashion.

### Gang Convergence Guarantees

The `ispc` execution model provides an important guarantee about the behavior of the program counter and execution mask: the execution of program instances is *maximally converged*. Maximal convergence means that if two program instances follow the same control path, they are guaranteed to execute each program statement concurrently. If two program instances follow diverging control paths, it is guaranteed that they will reconverge as soon as possible in the function (if they do later reconverge). [1]

[1] This is another significant difference between the `ispc` execution model and the one implemented by OpenCL* and CUDA*, which doesn't provide this guarantee.

Maximal convergence means that in the presence of divergent control flow such as the following:

```
if (test) {
  // true
}
else {
  // false
}
```

It is guaranteed that all program instances that were running before the `if` test will also be running after the end of the `else` block. (This guarantee stems from the notion of having a single program counter for the gang of program instances, rather than the concept of a unique program counter for each program instance.)

Another implication of this property is that it would be illegal for the `ispc` implementation to execute a function with an 8-wide gang by running it two times, with a 4-wide gang representing half of the original 8-wide gang each time.

It also follows that given the following program:

```
if (programIndex == 0) {
    while (true)  // infinite loop
        ;
}
print("hello, world\n");
```

the program will loop infinitely and the `print` statement will never be executed. (A different execution model that allowed gang divergence might execute the `print` statement since not all program instances were caught in the infinite loop in the example above.)

The way that "varying" function pointers are handled in `ispc` is also affected by this guarantee: if a function pointer is `varying`, then it has a possibly-different value for all running program instances. Given a call to a varying function pointer, `ispc` must maintain as much execution convergence as possible; the assembly code generated finds the set of unique function pointers over the currently running program instances and calls each one just once, such that the executing program instances when it is called are the set of active program instances that had that function pointer value. The order in which the various function pointers are called in this case is undefined.

### Uniform Data

A variable that is declared with the `uniform` qualifier represents a single value that is shared across the entire gang. (In contrast, the default variability qualifier for variables in `ispc`, `varying`, represents a variable that has a distinct storage location for each program instance in the gang.) (Though see the discussion in Struct Types for some subtleties related to `uniform` and `varying` when used with structures.)

It is an error to try to assign a `varying` value to a `uniform` variable, though `uniform` values can be assigned to `uniform` variables. Assignments to `uniform` variables are not affected by the execution mask (there's no unambiguous way that they could be); rather, they always apply if the program counter pointer passes through a statement that is a `uniform` assignment.

### Uniform Control Flow

One advantage of declaring variables that are shared across the gang as `uniform`, when appropriate, is the reduction in storage space required. A more important benefit is that it can enable the compiler to generate substantially better code for control flow; when a test condition for a control flow decision is based on a `uniform` quantity, the compiler can be immediately aware that all of the running program instances will follow the same path at that point, saving the overhead of needing to deal with control flow divergence and mask management. (To distinguish the two forms of control flow, will say that control flow based on `varying` expressions is "varying" control flow.)

Consider for example an image filtering operation where the program loops over pixels adjacent to the given (x,y) coordinates:

```
float box3x3(uniform float image[32][32], int x, int y) {
    float sum = 0;
    for (int dy = -1; dy <= 1; ++dy)
        for (int dx = -1; dx <= 1; ++dx)
            sum += image[y+dy][x+dx];
    return sum / 9.;
}
```

In general each program instance in the gang has different values for `x` and `y` in this function. For the box filtering algorithm here, all of the program instances will actually want to execute the same number of iterations of the `for` loops, with all of them having the same values for `dx` and `dy` each time through. If these loops are instead implemented with `dx` and `dy` declared as `uniform` variables, then the `ispc` compiler can generate more efficient code for the loops. [2]

[2] In this case, a sufficiently smart compiler could determine that `dx` and `dy` have the same value for all program instances and thus generate more optimized code from the start, though this optimization isn't yet implemented in `ispc`.

```
for (uniform int dy = -1; dy <= 1; ++dy)
    for (uniform int dx = -1; dx <= 1; ++dx)
        sum += image[y+dy][x+dx];
```

In particular, `ispc` can avoid the overhead of checking to see if any of the running program instances wants to do another loop iteration. Instead, the compiler can generate code where all instances always do the same iterations.

The analogous benefit comes when using `if` statements--if the test in an `if` statement is based on a `uniform` test, then the result will by definition be the same for all of the running program instances. Thus, the code for only one of the two cases needs to execute. `ispc` can generate code that jumps to one of the two, avoiding the overhead of needing to run the code for both cases.

### Uniform Variables and Varying Control Flow

Recall that in the presence of varying control flow, both the "true" and "false" clauses of an `if` statement may be executed, with the side effects of the instructions masked so that they only apply to the program instances that are supposed to be executing the corresponding clause. Under this model, we must define the effect of modifying `uniform` variables in the context of varying control flow.

In general, modifying `uniform` variables under varying control flow leads to the `uniform` variable having a value that depends on whether any of the program instances in the gang followed a particular execution path. Consider the following example:

```
float a = ...;
uniform int b = 0;
if (a == 0) {
    ++b;
    // b is 1
}
else {
    b = 10;
    // b is 10
}
// whether b is 1 or 10 depends on whether any of the values
// of "a" in the executing gang were 0.
```

Here, if any of the values of `a` across the gang was non-zero, then `b` will have a value of 10 after the `if` statement has executed. However, if all of the values of `a` in the currently-executing program instances at the start of the `if` statement had a value of zero, then `b` would have a value of 1.

### Data Races Within a Gang

In order to be able to write well-formed programs where program instances depend on values that are written to memory by other program instances within their gang, it's necessary to have a clear definition of when side-effects from one program instance become visible to other program instances running in the same gang.

In the model implemented by ispc, any side effect from one program instance is visible to other program instances in the gang after the next sequence point in the program. [3]

[3] This is a significant difference between ispc and SPMD languages like OpenCL* and CUDA*, which require barrier synchronization among the running program instances with functions like barrier() or __syncthreads(), respectively, to ensure this condition.

Generally, sequence points include the end of a full expression, before a function is entered in a function call, at function return, and at the end of initializer expressions. The fact that there is no sequence point between the increment of i and the assignment to i in i=i++ is why the effect that expression is undefined in C, for example. See, for example, the Wikipedia page on sequence points for more information about sequence points in C and C++.

In the following example, we have declared an array of values v, with one value for each running program instance. In the below, assume that programCount gives the gang size, and the varying integer value programIndex indexes into the running program instances starting from zero. (Thus, if 8 program instances are running, the first one of them will have a value 0, the next one a value of 1, and so forth up to 7.)

```
int x = ...;
uniform int tmp[programCount];
tmp[programIndex] = x;
int neighbor = tmp[(programIndex+1)%programCount];
```

In this code, the running program instances have written their values of x into the tmp array such that the ith element of tmp is equal to the value of x for the ith program instance. Then, the program instances load the value of neighbor from tmp, accessing the value written by their neighboring program instance (wrapping around to the first one at the end.) This code is well-defined and without data races, since the writes to and reads from tmp are separated by a sequence point.

(For this particular application of communicating values from one program instance to another, there are more efficient built-in functions in the ispc standard library; see Cross-Program Instance Operations for more information.)

It is possible to write code that has data races across the gang of program instances. For example, if the following function is called with multiple program instances having the same value of index, then it is undefined which of them will write their value of value to array[index].

```
void assign(uniform int array[], int index, int value) {
    array[index] = value;
}
```

As another example, if the values of the array indices i and j have the same values for some of the program instances, and an assignment like the following is performed:

```
int i = ..., j = ...;
uniform int array[...] = { ... };
array[i] = array[j];
```

then the program's behavior is undefined, since there is no sequence point between the reads and writes to the same location.

While this rule that says that program instances can safely depend on side-effects from by other program instances in their gang eliminates a class of synchronization requirements imposed by some other SPMD languages, it conversely means that it is possible to write ispc programs that compute different results when run with different gang sizes.

### Tasking Model

ispc provides an asynchronous function call (i.e. tasking) mechanism through the launch keyword. (The syntax is documented in the Task Parallelism: "launch" and "sync" Statements section.) A function called with launch executes asynchronously from the function that called it; it may run immediately or it may run concurrently on another processor in the system, for example.

If a function launches multiple tasks, there are no guarantees about the order in which the tasks will execute. Furthermore, multiple launched tasks from a single function may execute concurrently.

A function that has launched tasks may use the sync keyword to force synchronization with the launched functions; sync causes a function to wait for all of the tasks it has launched to finish before execution continues after the sync. (Note that sync only waits for the tasks launched by the current function, not tasks launched by other functions).

Alternatively, when a function that has launched tasks returns, an implicit sync waits for all launched tasks to finish before allowing the function to return to its calling function. This feature is important since it enables parallel composition: a function can call second function without needing to be concerned if the second function has launched asynchronous tasks or not--in either case, when the second function returns, the first function can trust that all of its computation has completed.

## The ISPC Language

ispc is an extended version of the C programming language, providing a number of new features that make it easy to write high-performance SPMD programs for the CPU and GPU. Note that between not only the few

small syntactic differences between `ispc` and C code but more importantly `ispc`'s fundamentally parallel execution model, C code can't just be recompiled to correctly run in parallel with `ispc`. However, starting with working C code and porting it to `ispc` can be an efficient way to quickly write `ispc` programs.

This section describes the syntax and semantics of the `ispc` language. To understand how to use `ispc`, you need to understand both the language syntax and `ispc`'s parallel execution model, which was described in the previous section, The ISPC Parallel Execution Model.

### Relationship To The C Programming Language

This subsection summarizes the differences between `ispc` and C; if you are already familiar with C, you may find it most effective to focus on this subsection and just focus on the topics in the remainder of section that introduce new language features. You may also find it helpful to compare the `ispc` and C++ implementations of various algorithms in the `ispc examples/` directory to get a sense of the close relationship between `ispc` and C.

Specifically, C89 is used as the baseline for comparison in this subsection (this is also the version of C described in the Second Edition of Kernighan and Ritchie's book). (`ispc` adopts some features from C99 and from C++, which will be highlighted in the below.)

`ispc` has the same syntax and features for the following as is present in C:

- Expression syntax and basic types
- Syntax for variable declarations
- Control flow structures: `if`, `for`, `while`, `do`, and `switch`.
- Pointers, including function pointers, `void *`, and C's array/pointer duality (arrays are converted to pointers when passed to functions, etc.)
- Structs and arrays
- Support for recursive function calls
- Support for separate compilation of source files
- "Short-circuit" evaluation of `||`, `&&` and `? :` operators
- The preprocessor

`ispc` adds a number of features from C++ and C99 to this base:

- A boolean type, `bool`, as well as built-in `true` and `false` values
- Reference types (e.g. `const float &foo`)
- Comments delimited by `//`
- Variables can be declared anywhere in blocks, not just at their start.
- Iteration variables for `for` loops can be declared in the `for` statement itself (e.g. `for (int i = 0; ...)`
- The `inline` qualifier to indicate that a function should be inlined
- Function overloading by parameter type
- Hexadecimal floating-point constants
- Dynamic memory allocation with `new` and `delete`.
- Limited support for overloaded operators (Operators Overloading).

`ispc` also adds a number of new features that aren't in C89, C99, or C++:

- Parallel `foreach` and `foreach_tiled` iteration constructs (see Parallel Iteration Statements: "foreach" and "foreach_tiled")
- The `foreach_active` and `foreach_unique` iteration constructs, which provide ways of iterating over subsets of the program instances in the gang. See Iteration over active program instances: "foreach_active" and Iteration over unique elements: "foreach_unique".)
- Language support for task parallelism (see Task Parallel Execution)
- "Coherent" control flow statements that indicate that control flow is expected to be coherent across the running program instances (see "Coherent" Control Flow Statements: "cif" and Friends)
- A rich standard library, though one that is different than C's (see The ISPC Standard Library.)
- Short vector types (see Short Vector Types)
- Syntax to specify integer constants as bit vectors (e.g. `0b1100` is 12)

There are a number of features of C89 that are not supported in `ispc` but are likely to be supported in future releases:

- There are no types named `char`, `short`, or `long` (or `long double`). However, there are built-in `int8`, `int16`, and `int64` types
- Character constants
- String constants and arrays of characters as strings
- `goto` statements are partially supported (see Unstructured Control Flow: "goto")
- `union` types
- Bitfield members of `struct` types
- Variable numbers of arguments to functions
- Literal floating-point constants (even without a `f` suffix) are currently treated as being `float` type, not `double`. To have a double precision floating point constant use `d` suffix.
- The `volatile` qualifier
- The `register` storage class for variables. (Will be ignored).

The following C89 features are not expected to be supported in any future `ispc` release:

- "K&R" style function declarations
- The C standard library

- Octal integer constants

The following reserved words from C89 are also reserved in `ispc`:

`break`, `case`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `int`, `NULL`, `return`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `unsigned`, `void`, and `while`.

`ispc` additionally reserves the following words:

`bool`, `delete`, `export`, `cdo`, `cfor`, `cif`, `cwhile`, `false`, `float16`, `foreach`, `foreach_active`, `foreach_tiled`, `foreach_unique`, `in`, `inline`, `noinline`, `__vectorcall`, `int8`, `int16`, `int32`, `int64`, `launch`, `new`, `print`, `uint8`, `uint16`, `uint32`, `uint64`, `soa`, `sync`, `task`, `true`, `uniform`, and `varying`.

### Lexical Structure

Tokens in `ispc` are delimited by white-space and comments. The white-space characters are the usual set of spaces, tabs, and carriage returns/line feeds. Comments can be delineated with `//`, which starts a comment that continues to the end of the line, or the start of a comment can be delineated with `/*` at the start and with `*/` at the end. Like C/C++, comments can't be nested.

Identifiers in `ispc` are sequences of characters that start with an underscore or an upper-case or lower-case letter, and then followed by zero or more letters, numbers, or underscores. Identifiers that start with two underscores are reserved for use by the compiler.

### Integer Literals

Integer numeric constants can be specified in base 10, hexadecimal, or binary. (Octal integer constants aren't supported). Base 10 constants are given by a sequence of one or more digits from 0 to 9. Hexadecimal constants are denoted by a leading `0x` or `0X` and then one or more digits from 0-9, a-f, or A-F. Finally, binary constants are denoted by a leading `0b` and then a sequence of 1s and 0s.

Here are three ways of specifying the integer value "15":

```
int fifteen_decimal = 15;
int fifteen_hex     = 0xf;
int fifteen_binary  = 0b1111;
```

A number of suffixes can be provided with integer numeric constants. First, "u" denotes that the constant is unsigned, and "ll" denotes a 64-bit integer constant (while "l" denotes a 32-bit integer constant). It is also possible to denote units of 1024, 1024*1024, or 1024*1024*1024 with the SI-inspired suffixes "k", "M", and "G" respectively:

```
int two_kb = 2k;   // 2048
int two_megs = 2M; // 2 * 1024 * 1024
int one_gig = 1G;  // 1024 * 1024 * 1024
```

### Floating Point Literals

ISPC supports 3 floating point types : `float16`, `float` and `double`.

- `float16` is an IEEE 754 half-precision (16 bit format) floating point type.
- `float` is an IEEE 754 single-precision (32 bit format) floating point type.
- `double` is an IEEE 754 double-precision (64 bit format) floating point type.

Floating-point constants of all three types can be specified in one of three ways.

- Decimal floating-point with radix separator - a sequence of zero of more 0-9 digits, followed by a period, followed by zero or more 0-9 digits. There must be at least one digit before or after the period. If floating-point suffix is used, radix separator is optional.
- Scientific notation - a decimal base followed by an "e" or "E", then optional plus or minus sign, and then a decimal exponent.
- Hexadecimal floating-point constant - bit-accurate representation of a particular floating-point number. It starts with "0x" or "0X" prefix, followed by a zero or a one, a period, and then the remainder of the mantissa in hexadecimal form, with digits from 0-9, a-f, or A-F. The start of the exponent is denoted by a "p" or "P", which is then followed by an optional plus or minus sign and then digits from 0 to 9, representing decimal value of the exponent. The exponent is never optional for hexadecimal floating-point literals.

The default type for floating-point literals is `float`. Floating-point literals can be specified by adding one of the following suffixes:

Operators

| Suffix | Type |
|---|---|
| `f16` or `F16` | `float16` |
| `f` or `F` | `float` |
| `d` or `D` | `double` |

For example:

`float` type floating point literals

```
float16 two_f16   = 2.0f16;                // 2.0
float16 pi_f16    = 0x1.92p+1f16;          // 3.1406
float16 neg_f16   = -65520.f16;            // -Inf
float   two_f     = 0x1p+1;                // 2.0
float   pi_f      = 0x1.921fb6p+1;         // 3.14159274
float   neg_f     = -0x1.ffep+11;          // -4095.0
double  two_d     = 2.0d;                  // 2.0
double  pi_d      = 0x1.921fb54442d18p+1d; // 3.1415926535897931
double  neg_d     = -0.3333333333333333d;  // -1/3
```

Also, "Fortran double" format is accepted - a scientific notation with a literal "d" or "D" used instead of "e". This notation yields a double precision floating point literal:

```
double d1 = 1.234d+3;  // 1234.0d
double d2 = 1.234e+3d; // 1234.0d
```

### String Literals

String constants in `ispc` are denoted by an opening double quote " followed by any character other than a newline, up to a closing double quote. Within the string, a number of special escape sequences can be used to specify special characters. These sequences all start with an initial \ and are listed below:

Escape sequences in strings

| | |
|---|---|
| \\ | backslash: \ |
| \" | double quotation mark: " |
| \' | single quotation mark: ' |
| \a | bell (alert) |
| \b | backspace character |
| \f | formfeed character |
| \n | newline |
| \r | carriage return |
| \t | horizontal tab |
| \v | vertical tab |
| \ followed by one or more digits from 0-8 | ASCII character in octal notation |
| \x, followed by one or more digits from 0-9, a-f, A-F | ASCII character in hexadecimal notation |

`ispc` doesn't support a string data type; string constants can be passed as the first argument to the `print()` statement, however. `ispc` also doesn't support character constants.

The following identifiers are reserved as language keywords: `bool`, `break`, `case`, `cdo`, `cfor`, `char`, `cif`, `cwhile`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `export`, `extern`, `false`, `float`, `float16`, `for`, `foreach`, `foreach_active`, `foreach_tiled`, `foreach_unique`, `goto`, `if`, `in`, `inline`, `noinline`, `int`, `int8`, `int16`, `int32`, `int64`, `launch`, `NULL`, `print`, `return`, `signed`, `sizeof`, `soa`, `static`, `struct`, `switch`, `sync`, `task`, `true`, `typedef`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `uniform`, `union`, `unsigned`, `varying`, `__vectorcall`, `void`, `volatile`, `while`.

`ispc` defines the following operators and punctuation:

Operators

| Symbols | Use |
|---|---|
| = | Assignment |
| +, -, *, /, % | Arithmetic operators |
| &, \|, ^, !, ~, &&, \|\|, <<, >> | Logical and bitwise operators |
| ++, -- | Pre/post increment/decrement |
| <, <=, >, >=, ==, != | Relational operators |
| *=, /=, +=, -=, <<=, >>=, &=, \|= | Compound assignment operators |
| ?, : | Selection operators |
| ; | Statement separator |
| , | Expression separator |
| . | Member access |

A number of tokens are used for grouping in `ispc`:

Grouping Tokens

| | |
|---|---|
| (, ) | Parenthesization of expressions, function calls, delimiting specifiers for control flow constructs. |
| [, ] | Array and short-vector indexing |
| {, } | Compound statements |

## Types

### Basic Types and Type Qualifiers

ispc is a statically-typed language. It supports a variety of core basic types:

- void: "empty" type representing no value.
- bool: boolean value; may be assigned true, false, or the value of a boolean expression.
- int8: 8-bit signed integer.
- unsigned int8: 8-bit unsigned integer; may also be specified as uint8.
- int16: 16-bit signed integer.
- unsigned int16: 16-bit unsigned integer; may also be specified as uint16.
- int: 32-bit signed integer; may also be specified as int32.
- unsigned int: 32-bit unsigned integer; may also be specified as unsigned int32, uint32 or uint.
- int64: 64-bit signed integer.
- unsigned int64: 64-bit unsigned integer; may also be specified as uint64.
- float16: 16-bit floating point value
- float: 32-bit floating point value
- double: 64-bit double-precision floating point value.

There are also a few built-in types related to pointers and memory:

- size_t: the maximum size of any object (structure or array)
- ptrdiff_t: an integer type large enough to represent the difference between two pointers
- intptr_t: signed integer type that is large enough to represent a pointer value
- uintptr_t: unsigned integer type large enough to represent a pointer

Implicit type conversion between values of different types is done automatically by the ispc compiler. Thus, a value of float type can be assigned to a variable of int type directly. In binary arithmetic expressions with mixed types, types are promoted to the "more general" of the two types, with the following precedence:

```
double > uint64 > int64 > float > uint32 > int32 >
    float16 > uint16 > int16 > uint8 > int8 > bool
```

In other words, adding an int64 to a double causes the int64 to be converted to a double, the addition to be performed, and a double value to be returned. If a different conversion behavior is desired, then explicit type-casts can be used, where the destination type is provided in parenthesis around the expression:

```
double foo = 1. / 3.;
int bar = (float)bar + (float)bar;  // 32-bit float addition
```

If a bool is converted to an integer numeric type (int, int64, etc.), then the result is a non-zero value if the bool has the value true and has the value zero otherwise. A bool with value true is not guaranteed to be one if converted to an integer numeric type.

Variables can be declared with the const qualifier, which prohibits their modification.

```
const float PI = 3.1415926535;
```

As in C, the extern qualifier can be used to declare a function or global variable defined in another source file, and the static qualifier can be used to define a variable or function that is only visible in the current scope. The values of static variables declared in functions are preserved across function calls.

### "uniform" and "varying" Qualifiers

If a variable has a uniform qualifier, then there is only a single instance of that variable shared by all program instances in a gang. (In other words, it necessarily has the same value across all of the program instances.) In addition to requiring less storage than varying values, uniform variables lead to a number of performance advantages when they are applicable (see Uniform Control Flow, for example.) Varying variables may be qualified with varying, though doing so has no effect, as varying is the default.

There are two exceptions for this rule described in Pointer Types and Type Casting sections.

uniform variables can be modified as the program executes, but only in ways that preserve the property that they have a single value for the entire gang. Thus, it's legal to add two uniform variables together and assign the result to a uniform variable, but assigning a non-uniform (i.e., varying) value to a uniform variable is a compile-time error.

uniform variables implicitly type-convert to varying types as required:

```
uniform int x = ...;
int y = ...;
int z = x * y;  // x is converted to varying for the multiply
```

Arrays themselves aren't uniform or varying, but the elements that they store are:

```
float foo[10];
uniform float bar[10];
```

The first declaration corresponds to 10 gang-wide `float` values in memory, while the second declaration corresponds to 10 `float` values.

### Defining New Names For Types

The `typedef` keyword can be used to name types:

```
typedef int64 BigInt;
typedef float Float3[3];
```

Following C's syntax, the code above defines `BigInt` to have `int64` type and `Float3` to have `float[3]` type.

Also as in C, `typedef` doesn't create a new type: it just provides an alternative name for an existing type. Thus, in the above example, it is legal to pass a value with `float[3]` type to a function that has been declared to take a `Float3` parameter.

### Pointer Types

It is possible to have pointers to data in memory; pointer arithmetic, changing values in memory with pointers, and so forth is supported as in C. As with other basic types, pointers can be both `uniform` and `varying`.

** Like other types in `ispc`, pointers are `varying` by default, if an explicit `uniform` qualifier isn't provided. However, the default variability of the pointed-to type is uniform. ** This rule will be illustrated and explained in examples below.

For example, the `ptr` variable in the code below is a varying pointer to `uniform float` values. Each program instance has a separate pointer value and the assignment to `*ptr` generally represents a scatter to memory.

```
uniform float a[] = ...;
int index = ...;
float * ptr = &a[index];
*ptr = 1;
```

A `uniform` pointer can be declared with an appropriately-placed qualifier:

```
float f = 0;
varying float * uniform pf = &f;  // uniform pointer to a varying float
*pf = 1;
```

The placement of the `uniform` qualifier to declare a `uniform` pointer may be initially surprising, but it matches the form of how, for example, a pointer that is itself `const` (as opposed to pointing to a `const` type) is declared in C. (Reading the declaration from right to left gives its meaning: a uniform pointer to a float that is varying.)

A subtlety comes in in cases like the where a uniform pointer points to a varying datatype. In this case, each program instance accesses a distinct location in memory (because the underlying varying datatype is itself laid out with a separate location in memory for each program instance.)

```
float a;
varying float * uniform pa = &a;
*pa = programIndex;  // same as (a = programIndex)
```

Also as in C, arrays are silently converted into pointers:

```
float a[10] = { ... };
varying float * uniform pa = a;     // pointer to first element of a
varying float * uniform pb = a + 5; // pointer to 5th element of a
```

Any pointer type can be explicitly typecast to another pointer type, as long as the source type isn't a `varying` pointer when the destination type is a `uniform` pointer.

```
float *pa = ...;
int *pb = (int *)pa;  // legal, but beware
```

Like other types, `uniform` pointers can be typecast to be `varying` pointers, however.

Any pointer type can be assigned to a `void` pointer without a type cast:

```
float foo(void *);
int *bar = ...;
foo(bar);
```

There is a special NULL value that corresponds to a NULL pointer. As a special case, the integer value zero can be implicitly converted to a NULL pointer and pointers are implicitly converted to boolean values in conditional expressions.

```
void foo(float *ptr) {
    if (ptr != 0) { // or, (ptr != NULL), or just (ptr)
       ...
```

It is legal to explicitly type-cast a pointer type to an integer type and back from an integer type to a pointer type. Note that this conversion isn't performed implicitly, for example for function calls.

### Function Pointer Types

Pointers to functions can also be taken and used as in C and C++. The syntax for declaring function pointer types is the same as in those languages; it's generally easiest to use a `typedef` to help:

```
int inc(int v) { return v+1; }
int dec(int v) { return v-1; }

typedef int (*FPType)(int);
FPType fptr = inc;  // vs. int (*fptr)(int) = inc;
```

Given a function pointer, the function it points to can be called:

```
int x = fptr(1);
```

It's not necessary to take the address of a function to assign it to a function pointer or to dereference it to call the function.

As with pointers to data in `ispc`, function pointers can be either `uniform` or `varying`. A call through a `uniform` causes all of the running program instances in the gang to call into the target function; the implications of a call through a `varying` function pointer are discussed in the section Gang Convergence Guarantees.

### Reference Types

`ispc` also provides reference types (like C++ references) that can be used for passing values to functions by reference, allowing functions can return multiple results or modify existing variables.

```
void increment(float &f) {
    ++f;
}
```

As in C++, once a reference is bound to a variable, it can't be rebound to a different variable:

```
float a = ..., b = ...;
float &r = a;  // makes r refer to a
r = b;  // assigns b to a, doesn't make r refer to b
```

An important limitation with references in `ispc` is that references can't be bound to varying lvalues; doing so causes a compile-time error to be issued. This situation is illustrated in the following code, where `vptr` is a `varying` pointer type (in other words, there each program instance in the gang has its own unique pointer value)

```
uniform float * uniform uptr = ...;
float &ra = *uptr;  // ok
uniform float * varying vptr = ...;
float &rb = *vptr;  // ERROR: *ptr is a varying lvalue type
```

(The rationale for this limitation is that references must be represented as either a uniform pointer or a varying pointer internally. While choosing a varying pointer would provide maximum flexibility and eliminate this restriction, it would reduce performance in the common case where a uniform pointer is all that's needed. As a work-around, a varying pointer can be used in cases where a varying lvalue reference would be desired.)

### Enumeration Types

It is possible to define user-defined enumeration types in `ispc` with the `enum` keyword, which is followed by an optional enumeration type name and then a brace-delimited list of enumerators with optional values:

```
enum Color { RED, GREEN, BLUE };
enum Flags {
    UNINITIALIZED = 0,
    INITIALIZED = 2,
    CACHED = 4
};
```

Each `enum` declaration defines a new type; an attempt to implicitly convert between enumerations of different types gives a compile-time error, but enumerations of different types can be explicitly cast to one other.

```
Color c = (Color)CACHED;
```

Enumerators are implicitly converted to integer types, however, so they can be directly passed to routines that take integer parameters and can be used in expressions including integers, for example. However, the integer result of such an expression must be explicitly cast back to the enumerant type if it to be assigned to a variable with the enumerant type.

```
Color c = RED;
int nextColor = c+1;
c = (Color)nextColor;
```

In this particular case, the explicit cast could be avoided using an increment operator.

```
Color c = RED;
++c;  // c == GREEN now
```

### Short Vector Types

`ispc` supports a parameterized type to define short vectors. These short vectors can only be used with basic types like `float` and `int`; they can't be applied to arrays or structures. Note: `ispc` does *not* use these short vectors to facilitate program vectorization; they are purely a syntactic convenience. Using them or writing the corresponding code without them shouldn't lead to any noticeable performance differences between the two approaches.

Syntax similar to C++ templates is used to declare these types:

```
float<3> foo;   // vector of three floats
double<6> bar;
```

The length of these vectors can be arbitrarily long, though the expected usage model is relatively short vectors.

You can use `typedef` to create types that don't carry around the brackets around the vector length:

```
typedef float<3> float3;
```

`ispc` doesn't support templates in general. In particular, not only must the vector length be a compile-time constant, but it's also not possible to write functions that are parameterized by vector length.

```
uniform int i = foo();
// ERROR: length must be compile-time constant
float<i> vec;
// ERROR: can't write functions parameterized by vector length
float<N> func(float<N> val);
```

Arithmetic on these short vector types works as one would expect; the operation is applied component-wise to the values in the vector. Here is a short example:

```
float<3> func(float<3> a, float<3> b) {
    a += b;    // add individual elements of a and b
    a *= 2.;   // multiply all elements of a by 2
    bool<3> test = a < b;  // component-wise comparison
    return test ? a : b;   // return each minimum component
}
```

As shown by the above code, scalar types automatically convert to corresponding vector types when used in vector expressions. In this example, the constant `2.` above is converted to a three-vector of 2s for the multiply in the second line of the function implementation.

Type conversion between other short vector types also works as one would expect, though the two vector types must have the same length:

```
float<3> foo = ...;
int<3> bar = foo;    // ok, cast elements to ints
int<4> bat = foo;    // ERROR: different vector lengths
float<4> bing = foo; // ERROR: different vector lengths
```

For convenience, short vectors can be initialized with a list of individual element values:

```
float x = ..., y = ..., z = ...;
float<3> pos = { x, y, z };
```

There are two mechanisms to access the individual elements of these short vector data types. The first is with the array indexing operator:

```
float<4> foo;
for (uniform int i = 0; i < 4; ++i)
    foo[i] = i;
```

`ispc` also provides a specialized mechanism for naming and accessing the first few elements of short vectors based on an overloading of the structure member access operator. The syntax is similar to that used in HLSL, for example.

```
float<3> position;
position.x = ...;
position.y = ...;
position.z = ...;
```

More specifically, the first element of any short vector type can be accessed with `.x` or `.r`, the second with `.y` or `.g`, the third with `.z` or `.b`, and the fourth with `.w` or `.a`. Just like using the array indexing operator with an index that is greater than the vector size, accessing an element that is beyond the vector's size is undefined behavior and may cause your program to crash.

It is also possible to construct new short vectors from other short vector values using this syntax, extended for "swizzling". For example,

```
float<3> position = ...;
float<3> new_pos = position.zyx;  // reverse order of components
float<2> pos_2d = position.xy;
```

Though a single element can be assigned to, as in the examples above, it is not currently possible to use swizzles on the left-hand side of assignment expressions:

```
int8<2> foo = ...;
int8<2> bar = ...;
foo.yz = bar;   // Error: can't assign to left-hand side of expression
```

### Array Types

Arrays of any type can be declared just as in C and C++:

```
float a[10]; // array of 10 varying floats
uniform int * varying b[20]; // array of 20 varying pointers to uniform int
```

Multidimensional arrays can be specified as arrays of arrays; the following declares an array of 5 arrays of 15 floats.

```
uniform float a[5][15];
```

The size of arrays must be a compile-time constant, though array size can be determined from array initializer lists; see the following section, Declarations and Initializers, for details. One exception to this is that functions can be declared to take "unsized arrays" as parameters:

```
void foo(float array[], int length);
```

Finally, the name of an array will be automatically implicitly converted to a uniform pointer to the array type if needed:

```
uniform int a[10];
int * uniform ap = a;
```

### Struct Types

Aggregate data structures can be built using `struct`.

```
struct Foo {
    float time;
    int flags[10];
};
```

As in C++, after a `struct` is declared, an instance can be created using the `struct`'s name:

```
Foo f;
```

Alternatively, `struct` can be used before the structure name:

```
struct Foo f;
```

Members in a structure declaration may each have `uniform` or `varying` qualifiers, or may have no rate qualifier, in which case their variability is initially "unbound".

```
struct Bar {
    uniform int a;
    varying int b;
    int c;
};
```

In the declaration above, the variability of `c` is unbound. The variability of struct members that are unbound is resolved when a struct is defined; if the `struct` is `uniform`, then unbound members are `uniform`, and if the `struct` is `varying`, then unbound members are varying.

```
Bar vb;
uniform Bar ub;
```

Here, `b` is a `varying Bar` (since `varying` is the default variability). If `Bar` is defined as above, then `vb.a` is still a `uniform int`, since its variability was bound in the original declaration of the `Bar` type. Similarly, `vb.b` is `varying`. The variability of `vb.c` is `varying`, since `vb` is `varying`.

(Similarly, `ub.a` is `uniform`, `ub.b` is `varying`, and `ub.c` is `uniform`.)

In most cases, it's worthwhile to declare `struct` members with unbound variability so that all have the same variability for both `uniform` and `varying` structs. In particular, if a `struct` has a member with bound `uniform` type, it's not possible to index into an array of the struct type with a `varying` index. Consider the following example:

```
struct Foo { uniform int a; };
uniform Foo f[...] = ...;
int index = ...;
Foo fv = f[index];  // ERROR
```

Here, the `Foo` type has a member with bound `uniform` variability. Because `index` has a different value for each program instance in the above code, the value of `f[index]` needs to be able to store a different value of `Foo::a` for each program instance. However, a `varying Foo` still has only a single `a` member, since `a` was declared with `uniform` variability in the declaration of `Foo`. Therefore, the indexing operation in the last line results in an error.

### Operators Overloading

ISPC has limited support for overloaded operators for `struct` types. Only binary operators are supported currently, namely they are: `*, /, %, +, -, >> and <<`. Operators overloading support is similar to the one in C++ language. To overload an operator for `struct S`, you need to declare and implement a function using keyword `operator`, which accepts two parameters of type `struct S` or `struct S&` and returns either of these types. For example:

```
struct S { float re, im;};
struct S operator*(struct S a, struct S b) {
    struct S result;
    result.re = a.re * b.re - a.im * b.im;
    result.im = a.re * b.im + a.im * b.re;
    return result;
}

void foo(struct S a, struct S b) {
    struct S mul = a*b;
    print("a.re:   %\na.im:   %\n", a.re, a.im);
    print("b.re:   %\nb.im:   %\n", b.re, b.im);
    print("mul.re: %\nmul.im: %\n", mul.re, mul.im);
}
```

### Structure of Array Types

If data can be laid out in memory so that the executing program instances access it via loads and stores of contiguous sections of memory, overall performance can be improved noticably. One way to improve this memory access coherence is to lay out structures in "structure of arrays" (SOA) format in memory; the benefits from SOA layout are discussed in more detail in the Use "Structure of Arrays" Layout When Possible section in the ispc Performance Guide.

`ispc` provides two key language-level capabilities for laying out and accessing data in SOA format:

- An `soa` keyword that transforms a regular `struct` into an SOA version of the struct.
- Array indexing syntax for SOA arrays that transparently handles SOA indexing.

As an example, consider a simple struct declaration:

```
struct Point { float x, y, z; };
```

With the `soa` rate qualifier, an array of SOA variants of this structure can be declared:

```
soa<8> Point pts[...];
```

The in-memory layout of the `Point` instances has had the SOA transformation applied, such that there are 8 `x` values in memory followed by 8 `y` values, and so forth. Here is the effective declaration of `soa<8> Point`:

```
struct { uniform float x[8], y[8], z[8]; };
```

Given an array of SOA data, array indexing (and pointer arithmetic) is done so that the appropriate values from the SOA array are accessed. For example, given:

```
soa<8> Point pts[...];
uniform float x = pts[10].x;
```

The generated code effectively accesses the second 8-wide SOA structure and then loads the third `x` value from it. In general, one can write the same code to access arrays of SOA elements as one would write to access them in AOS layout.

Note that it directly follows from SOA layout that the layout of a single element of the array isn't contiguous in memory--`pts[1].x` and `pts[1].y` are separated by 7 `float` values in the above example.

There are a few limitations to the current implementation of SOA types in `ispc`; these may be relaxed in future releases:

- It's illegal to typecast to `soa` data to `void` pointers.
- Reference types are illegal in SOA structures
- All members of SOA structures must have no rate qualifiers--specifically, it's illegal to have an explicitly-qualified `uniform` or `varying` member of a structure that has `soa` applied to it.

### Declarations and Initializers

Variables are declared and assigned just as in C:

```
float foo = 0, bar[5];
float bat = func(foo);
```

More complex declarations are also possible:

```
void (*fptr_array[16])(int, int);
```

Here, `fptr_array` is an array of 16 pointers to functions that have `void` return type and take two `int` parameters.

If a variable is declared without an initializer expression, then its value is undefined until a value is assigned to it. Reading an undefined variable is undefined behavior.

Any variable that is declared at file scope (i.e. outside a function) is a global variable. If a global variable is qualified with the `static` keyword, then its only visible within the compilation unit in which it was defined. As in C/C++, a variable with a `static` qualifier inside a functions maintains its value across function invocations.

As in C++, variables don't need to be declared at the start of a basic block:

```
int foo = ...;
if (foo < 2) { ... }
int bar = ...;
```

Variables can also be declared in `for` statement initializers:

```
for (int i = 0; ...)
```

Varying variables can be initialized with individual element values in braces. The number of values has to be equal to the target width. So, static varying initialization is not portable across targets with different widths unless guarded with `#if TARGET_WIDTH`:

```
#if TARGET_WIDTH == 4
    varying int bar = { 1, 2, 3, 4 };
#elif TARGET_WIDTH == 8
    varying int bar = { 1, 2, 3, 4, 5, 6, 7, 8 };
#elif TARGET_WIDTH == 16
    ...
#endif
```

Arrays can be initialized with individual element values in braces:

```
int bar[2][4] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } };
```

An array with an initializer expression can be declared with some or all of its dimensions unspecified. In this case, the "shape" of the initializer expression is used to determine the array dimensions:

```
// This corresponds to bar[2][4], due to the initializer expression
int bar[][] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } };
```

Structures can also be initialized by providing element values in braces:

```
struct Color { float r, g, b; };
....
Color d = { 0.5, .75, 1.0 }; // r = 0.5, ...
```

Arrays of structures and arrays inside structures can be initialized with the expected syntax:

```
struct Foo { int x; float bar[3]; };
Foo fa[2] = { { 1, { 2, 3, 4 } }, { 10, { 20, 30, 40 } } };
// now, fa[1].bar[2] == 40, and so forth
```

### Expressions

All of the operators from C that you'd expect for writing expressions are present. Rather than enumerating all of them, here is a short summary of the range of them available in action.

```
unsigned int i = 0x1234feed;
unsigned int j = (i << 3) ^ ~(i - 3);
i += j / 6;
float f = 1.234e+23;
float g = j * f / (2.f * i);
double h = (g < 2) ? f : g/5;
```

Structure member access and array indexing also work as in C.

```
struct Foo { float f[5]; int i; };
Foo foo = { { 1,2,3,4,5 }, 2 };
return foo.f[4] - foo.i;
```

The address-of operator, pointer dereference operator, and pointer member operator also work as expected.

```
struct Foo { float a, b, c; };
Foo f;
Foo * uniform fp = &f;
(*fp).a = 0;
fp->b = 1;
```

As in C and C++, evaluation of the `||` and `&&` logical operators as well as the selection operator `? :` is "short-circuited"; the right hand side won't be evaluated if the value from the left-hand side determines the logical operator's value. For example, in the following code, `array[index]` won't be evaluated for values of `index` that are greater than or equal to `NUM_ITEMS`.

```
if (index < NUM_ITEMS && array[index] > 0) {
    // ...
}
```

Short-circuiting may impose some overhead in the generated code; for cases where short-circuiting is undesirable due to performance impact, see the section Logical and Selection Operations, which introduces helper functions in the standard library that provide these operations without short-circuiting.

### Dynamic Memory Allocation

`ispc` programs can dynamically allocate (and free) memory, using syntax based on C++'s `new` and `delete` operators:

```
int count = ...;
int *ptr = new int[count];
// use ptr...
delete[] ptr;
```

In the above code, each program instance allocates its own `count` sized array of `uniform int` values, uses that memory, and then deallocates that memory. Uses of `new` and `delete` in `ispc` programs are implemented as calls to C library's aligned memory allocation routines, which are platform dependent (`posix_memalign()` and `free()` on Linux* and macOS* and `_aligned_malloc()` and `_aligned_free()` on Windows*). So it's advised to pair ISPC's `new` and `delete` with each other, but not with C/C++ memory management functions.

Note that the rules for `uniform` and `varying` for `new` are analogous to the corresponding rules for pointers (as described in Pointer Types). Specifically, if a specific rate qualifier isn't provided with the `new` expression,

then the default is that a "varying" `new` is performed, where each program instance performs a unique allocation. The allocated type, in turn, is by default `uniform`.

After a pointer has been deleted, it is illegal to access the memory it points to. However, that deletion happens on a per-program-instance basis. In other words, consider the following code:

```
int *ptr = new int[count];
// use ptr
if (count > 1000)
    delete[] ptr;
// ...
```

Here, the program instances where `count` is greater than 1000 have deleted the dynamically allocated memory pointed to by `ptr`, but the other program instances have not. As such, it's illegal for the former set of program instances to access `*ptr`, but it's perfectly fine for the latter set to continue to use the memory `ptr` points to. Note that it is illegal to delete a pointer value returned by `new` more than one time.

Sometimes, it's useful to be able to do a single allocation for the entire gang of program instances. A `new` statement can be qualified with `uniform` to indicate a single memory allocation:

```
float * uniform ptr = uniform new float[10];
```

While a regular call to `new` returns a `varying` pointer (i.e. a distinct pointer to separately-allocated memory for each program instance), a `uniform new` performs a single allocation and returns a `uniform` pointer. Recall that with a `uniform new`, the default variability of the allocated type is `varying`, so the above code is allocating an array of ten `varying float` values.

When using `uniform new`, it's important to be aware of a subtlety; if the returned pointer is stored in a varying pointer variable (as may be appropriate and useful for the particular program being written), then the varying pointer may inadvertently be passed to a subsequent `delete` statement, which is an error: effectively

```
varying float * ptr = uniform new float[10];
// use ptr...
delete ptr;  // ERROR: varying pointer is deleted
```

In this case, `ptr` will be deleted multiple times, once for each executing program instance, which is an error (unless it happens that only a single program instance is active in the above code.)

When using `new` statements, it's important to make an appropriate choice of `uniform` or `varying`, for both the `new` operator itself as well as the type of data being allocated, based on the program's needs. Consider the following four memory allocations:

```
uniform float * uniform p1 = uniform new uniform float[10];
float * uniform p2 = uniform new float[10];
float * p3 = new float[10];
varying float * p4 = new varying float[10];
```

Assuming that a `float` is 4 bytes in memory and if the gang size is 8 program instances, then the first allocation represents a single allocation of 10 `uniform float` values (40 bytes), the second is a single allocation of 10 `varying float` values (8*4*10 = 320 bytes), the third is 8 allocations of 10 `uniform float` values (8 allocations of 40 bytes each), and the last performs 8 allocations of 320 bytes each.

Note in particular that varying allocations of varying data types are rarely desirable in practice. In that case, each program instance is performing a separate allocation of `varying float` memory. In this case, it's likely that the program instances will only access a single element of each `varying float`, which is wasteful. (This in turn is partially why the allocated type is uniform by default with both pointers and `new` statements.)

Although `ispc` doesn't support constructors or destructors like C++, it is possible to provide initializer values with `new` statements:

```
struct Point { float x, y, z; };
Point *pptr = new Point(10, 20, 30);
```

Here for example, the "x" element of the returned `Point` is initialized to have the value 10 and so forth. In general, the rules for how initializer values provided in `new` statements are used to initialize complex data types follow the same rules as initializers for variables described in Declarations and Initializers.

### Type Casting

C-style type casting expression works as in C language with an exception that unbound type is not treated as `varying` by default.

When typecasting to some type `T` without specifying a variability, the variability is derived from the type of expression being casted. I.e. the expression `(int) E` has the same variability as original expression `E`. This feature may lead to confusion when the resulting expression is used as a function argument. Consider an example:

```
float bar(uniform float f);
float bar(varying float f);
float foo(uniform int b) {
   return bar((float)b);
}
```

This code will yield the following warning suggesting to use fully qualified type in this case.

```
Warning: Typecasting to type "/*unbound*/ float" (variability not specified)
       from "uniform" type "uniform int32" results in "uniform" variability.
       In the context of function argument it may lead to unexpected behavior.
       Casting to "uniform float" is recommended.
```

### Control Flow

`ispc` supports most of C's control flow constructs, including `if`, `switch`, `for`, `while`, `do`. It has limited support for `goto`, detailed below. It also supports variants of C's control flow constructs that provide hints about the expected runtime coherence of the control flow at that statement. It also provides parallel looping constructs, `foreach` and `foreach_tiled`, all of which will be detailed in this section.

### Conditional Statements: "if"

The `if` statement behaves precisely as in C; the code in the "true" block only executes if the condition evaluates to `true`, and if an optional `else` clause is provided, the code in the "else" block only executes if the condition is false.

```
float x = ..., y = ...;
if (x < 0.)
    y = -y;
else
    x *= 2.;
```

### Conditional Statements: "switch"

The `switch` conditional statement is also available, again with the same behavior as in C; the expression used in the `switch` must be of integer type (but it can be uniform or varying). As in C, if there is no `break` statement at the end of the code for a given case, execution "falls through" to the following case. These features are demonstrated in the code below.

```
int x = ...;
switch (x) {
case 0:
case 1:
    foo(x);
    /* fall through */
case 5:
    x = 0;
    break;
default:
    x *= x;
}
```

### Iteration Statements

In addition to the standard iteration statements `for`, `while`, and `do`, inherited from C/C++, `ispc` provides a number of additional specialized ways to iterate over data.

### Basic Iteration Statements: "for", "while", and "do"

`ispc` supports `for`, `while`, and `do` loops, with the same specification as in C. As in C++, variables can be declared in the `for` statement itself:

```
for (uniform int i = 0; i < 10; ++i) {
  // loop body
}
// i is now no longer in scope
```

You can use `break` and `continue` statements in `for`, `while`, and `do` loops; `break` breaks out of the current enclosing loop, while `continue` has the effect of skipping the remainder of the loop body and jumping to the loop step.

Note that all of these looping constructs have the effect of executing independently for each of the program instances in a gang; for example, if one of them executes a `continue` statement, other program instances executing code in the loop body that didn't execute the `continue` will be unaffected by it.

### Iteration over active program instances: "foreach_active"

The `foreach_active` construct specifies a loop that serializes over the active program instances: the loop body executes once for each active program instance, and with only that program instance executing.

As an example of the use of this construct, consider an application where each program instance independently computes an offset into a shared array that is being updated:

```
uniform float array[...] = { ... };
int index = ...;
++array[index];
```

If more than one active program instance computes the same value for `index`, the above code has undefined behavior (see the section Data Races Within a Gang for details.) The increment of `array[index]` could instead be written inside a `foreach_active` statement:

```
foreach_active (index) {
    ++array[index];
}
```

The variable name provided in parenthesis after the `foreach_active` keyword (here, `index`), causes a `const uniform int64` local variable of that name to be declared, where the variable takes the `programIndex` value of the program instance executing at each loop iteration.

In the code above, because only one program instance is executing at a time when the loop body executes, the update to `array` is well-defined. Note that for this particular example, the "local atomic" operations in the standard library could be used instead to safely update `array`. However, local atomics functions aren't always available or appropriate for more complex cases.)

`continue` statements may be used inside `foreach_active` loops, though `break` and `return` are prohibited. The order in which the active program instances are processed in the loop is not defined.

See the Using "foreach active" Effectively Section in the `ispc` Performance Guide for more details about `foreach_active`.

### Iteration over unique elements: "foreach_unique"

It can be useful to iterate over the elements of a varying variable, processing the subsets of them that have the same value together. For example, consider a varying variable `x` that has the values `{1, 2, 2, 1, 1, 0, 0, 0}`, where the program is running on a target with a gang size of 8 program instances. Here, `x` has three unique values across the program instances: 0, 1, and 2.

The `foreach_unique` looping construct allows us to iterate over these unique values. In the code below, the `foreach_unique` loop body executes once for each of the three unique values, with execution mask set to match the program instances where the varying value matches the current unique value being processed.

```
int x = ...; // assume {1, 2, 2, 1, 1, 0, 0, 0}
foreach_unique (val in x) {
    extern void func(uniform int v);
    func(val);
}
```

In the above, `func()` will be called three times, once with value 0, once with value 1, and once with value 2. When it is called for value 0, only the last three program instances will be executing, and so forth. The order in which the loop executes for the unique values isn't defined.

The varying expression that provides the values to be iterated over is only evaluated once, and it must be of an atomic type (`float`, `int`, etc.), an `enum` type, or a pointer type. The iteration variable `val` is a variable of `const uniform` type of the iteration type; it can't be modified within the loop. Finally, `break` and `return` statements are illegal within the loop body, but `continue` statements are allowed.

### Parallel Iteration Statements: "foreach" and "foreach_tiled"

The `foreach` and `foreach_tiled` constructs specify loops over a possibly multi-dimensional domain of integer ranges. Their role goes beyond "syntactic sugar"; they provide one of the two key ways of expressing parallel computation in `ispc`.

In general, a `foreach` or `foreach_tiled` statement takes one or more dimension specifiers separated by commas, where each dimension is specified by `identifier = start ... end`, where `start` is a signed integer value less than or equal to `end`, specifying iteration over all integer values from `start` up to and including `end-1`. An arbitrary number of iteration dimensions may be specified, with each one spanning a different range of values. Within the `foreach` loop, the given identifiers are available as `const varying int32` variables. The execution mask starts out "all on" at the start of each `foreach` loop iteration, but may be changed by control flow constructs within the loop.

It is illegal to have a `break` statement or a `return` statement within a `foreach` loop; a compile-time error will be issued in this case. (It is legal to have a `break` in a regular `for` loop that's nested inside a `foreach` loop.) `continue` statements are legal in `foreach` loops; they have the same effect as in regular `for` loops: a program instances that executes a `continue` statement effectively skips over the rest of the loop body for the current iteration.

It is also currently illegal to have nested `foreach` statements; this limitation will be removed in a future release of `ispc`.

As a specific example, consider the following `foreach` statement:

```
foreach (j = 0 ... height, i = 0 ... width) {
    // loop body--process data element (i,j)
}
```

It specifies a loop over a 2D domain, where the `j` variable goes from 0 to `height-1` and `i` goes from 0 to `width-1`. Within the loop, the variables `i` and `j` are available and initialized accordingly.

`foreach` loops actually cause the given iteration domain to be automatically mapped to the program instances in the gang, so that all of the data can be processed, in gang-sized chunks. As a specific example, consider a simple `foreach` loop like the following, on a target where the gang size is 8:

```
foreach (i = 0 ... 16) {
    // perform computation on element i
}
```

One possible valid execution path of this loop would be for the program counter the step through the statements of this loop just `16/8==2` times; the first time through, with the `varying int32` variable `i` having the values (0,1,2,3,4,5,6,7) over the program instances, and the second time through, having the values (8,9,10,11,12,13,14,15), thus mapping the available program instances to all of the data by the end of the loop's execution.

In general, however, you shouldn't make any assumptions about the order in which elements of the iteration domain will be processed by a `foreach` loop. For example, the following code exhibits undefined behavior:

```
uniform float a[10][100];
foreach (i = 0 ... 10, j = 0 ... 100) {
    if (i == 0)
        a[i][j] = j;
    else
        // Error: can't assume that a[i-1][j] has been set yet
        a[i][j] = a[i-1][j];
```

The `foreach` statement generally subdivides the iteration domain by selecting sets of contiguous elements in the inner-most dimension of the iteration domain. This decomposition approach generally leads to coherent memory reads and writes, but may lead to worse control flow coherence than other decompositions.

Therefore, `foreach_tiled` decomposes the iteration domain in a way that tries to map locations in the domain to program instances in a way that is compact across all of the dimensions. For example, on a target with an 8-wide gang size, the following `foreach_tiled` statement might process the iteration domain in chunks of 2 elements in `j` and 4 elements in `i` each time. (The trade-offs between these two constructs are discussed in more detail in the ispc Performance Guide.)

```
foreach_tiled (j = 0 ... height, i = 0 ... width) {
    // loop body--process data element (i,j)
}
```

### Parallel Iteration with "programIndex" and "programCount"

In addition to `foreach` and `foreach_tiled`, `ispc` provides a lower-level mechanism for mapping SPMD program instances to data to operate on via the built-in `programIndex` and `programCount` variables.

`programIndex` gives the index of the SIMD-lane being used for running each program instance. (In other words, it's a varying integer value that has value zero for the first program instance, and so forth.) The `programCount` builtin gives the total number of instances in the gang. Together, these can be used to uniquely map executing program instances to input data. [4]

[4] `programIndex` is analogous to `get_global_id()` in OpenCL* and `threadIdx` in CUDA*.

As a specific example, consider an `ispc` function that needs to perform some computation on an array of data.

```
for (uniform int i = 0; i < count; i += programCount) {
    float d = data[i + programIndex];
    float r = ....
    result[i + programIndex] = r;
}
```

Here, we've written a loop that explicitly loops over the data in chunks of `programCount` elements. In each loop iteration, the running program instances effectively collude amongst themselves using `programIndex` to determine which elements to work on in a way that ensures that all of the data elements will be processed. In this particular case, a `foreach` loop would be preferable, as `foreach` naturally handles the case where

`programCount` doesn't evenly divide the number of elements to be processed, while the loop above assumes that case implicitly.

### Unstructured Control Flow: "goto"

`goto` statements are allowed in `ispc` programs under limited circumstances; specifically, only when the compiler can determine that if any program instance executes a `goto` statement, then all of the program instances will be running at that statement, such that all will follow the `goto`.

Put another way: it's illegal for there to be "varying" control flow statements in scopes that enclose a `goto` statement. An error is issued if a `goto` is used in this situation.

The syntax for adding labels to `ispc` programs and jumping to them with `goto` is the same as in C. The following code shows a `goto` based equivalent of a `for` loop where the induction variable `i` goes from zero to ten.

```
  uniform int i = 0;
check:
  if (i > 10)
      goto done;
  // loop body
  ++i;
  goto check;
done:
  // ...
```

### "Coherent" Control Flow Statements: "cif" and Friends

`ispc` provides variants of all of the standard control flow constructs that allow you to supply a hint that control flow is expected to be coherent at a particular point in the program's execution. These mechanisms provide the compiler a hint that it's worth emitting extra code to check to see if the control flow is in fact coherent at run-time, in which case a simpler code path can often be executed.

The first of these statements is `cif`, indicating an `if` statement that is expected to be coherent. The usage of `cif` in code is just the same as `if`:

```
cif (x < y) {
    ...
} else {
    ...
}
```

`cif` provides a hint to the compiler that you expect that most of the executing SPMD programs will all have the same result for the `if` condition.

Along similar lines, `cfor`, `cdo`, and `cwhile` check to see if all program instances are running at the start of each loop iteration; if so, they can run a specialized code path that has been optimized for the "all on" execution mask case.

### Functions and Function Calls

Like C, functions must be declared in `ispc` before they are called, though a forward declaration can be used before the actual function definition. Also like C, arrays are passed to functions by reference. Recursive function calls are legal:

```
int gcd(int a, int b) {
    if (a == 0)
        return b;
    else
        return gcd(b%a, a);
}
```

Functions can be declared with a number of qualifiers that affect their visibility and capabilities. As in C/C++, functions have global visibility by default. If a function is declared with a `static` qualifier, then it is only visible in the file in which it was declared.

```
static float lerp(float t, float a, float b) {
    return (1.-t)*a + t*b;
}
```

Any function that can be launched with the `launch` construct in `ispc` must have a `task` qualifier; see Task Parallelism: "launch" and "sync" Statements for more discussion of launching tasks in `ispc`.

A function can also be given the `unmasked` qualifier; this qualifier indicates that all program instances should be made active at the start of the function execution (or, equivalently, that the current execution mask shouldn't be passed to the function from the function call site.) If it is known that a function will always be called when all program instances are executing, adding this qualifier can slightly improve performance. See the Section Re-establishing The Execution Mask for more discussion of `unmasked` program code.

Functions that are intended to be called from C/C++ application code must have the `export` qualifier. This causes them to have regular C linkage and to have their declarations included in header files, if the `ispc` compiler is directed to generated a C/C++ header file for the file it compiled.

```
export uniform float inc(uniform float v) {
    return v+1;
}
```

Finally, any function defined with an `inline` qualifier will always be inlined by `ispc`; `inline` is not a hint, but forces inlining. The compiler will opportunistically inline short functions depending on their complexity, but any function that should always be inlined should have the `inline` qualifier. Similarly, any function defined with a `noinline` qualifier will never be inlined by `ispc`. `noinline` and `inline` cannot be used on the same function.

### Function Overloading

Functions can be overloaded by parameter type. Given multiple definitions of a function, `ispc` uses the following model to choose the best function: each conversion of two types has its cost. `ispc` tries to find conversion with the smallest cost. When `ispc` can't find any conversion it means that this function is not suitable. Then `ispc` sums costs for all arguments and chooses the function with the smallest final cost. If the chosen function has some arguments which costs are bigger than their costs in other function this treats as ambiguous. Costs of type conversions placed from small to big:

1. Parameter types match exactly.
2. Function parameter type is reference and parameters match when any reference-type parameter are considered equivalent to their underlying type.
3. Function parameter type is const-reference and parameters match when any reference-type parameter are considered equivalent to their underlying type ignoring const attributes.
4. Parameters match exactly, except constant attributes. [NO CONSTANT ATTRIBUTES LATER]
5. Parameters match exactly, except reference attributes. [NO REFERENCES ATTRIBUTES LATER]
6. Parameters match with only type conversions that don't risk losing any information (for example, converting an int16 value to an int32 parameter value.)
7. Parameters match with only promotions from uniform to varying types.
8. Parameters match using arbitrary type conversion, without changing variability from uniform to varying (e.g., int to float, float to int.)
9. Parameters match with widening and promotions from uniform to varying types. (combination of "6" and "7")
10. Parameters match using arbitrary type conversion, including also changing variability from uniform to varying.

- If function parameter type is reference and neither "2" nor "3" aren't suitable, function is not suitable
- If "10" isn't suitable, function is not suitable

### Re-establishing The Execution Mask

As discussed in Functions and Function Calls, a function that is declared with an `unmasked` qualifier starts execution with all program instances running, regardless of the execution mask at the site of the function call. A block of statements can also be enclosed with `unmasked` to have the same effect within a function:

```
int a = ..., b = ...;
if (a < b) {
    // only program instances where a < b are executing here
    unmasked {
        // now all program instances are executing
    }
    // and again only the a < b instances
}
```

`unmasked` can be useful in cases where the programmer wants to "change the axis of parallelism" or use nested parallelism, as shown in the following code:

```
uniform WorkItem items[...] = ...;
foreach (itemNum = 0 ... numItems) {
    // do computation on items[itemNum] to determine if it needs
    // further processing...
    if (/* itemNum needs processing */) {
        foreach_active (i) {
            unmasked {
                uniform int uItemNum = extract(itemNum, i);
                // apply entire gang of program instances to uItemNum
            }
        }
    }
}
```

The general idea is that we are first using SPMD parallelism to determine which of the items requires further processing, checking a gang's worth of them concurrently inside the `foreach` loop. Assuming that only a

subset of them needs further processing, would be wasteful to do this work within the `foreach` loop in the same program instance that made the initial determination of whether more work as needed; in this case, all of the program instances corresponding to items that didn't need further processing would be inactive, with corresponding unused computational capability in the system.

In the above code, this issue is avoided by working on each of the items requiring more processing in turn with `foreach_active` and then using `unmasked` to re-establish execution of all of the program instances. The entire gang can in turn be applied to the computation to be done for each `items[itemNum]`.

The `unmasked` statement should be used with care; it can lead to a number of surprising cases of undefined program behavior. For example, consider the following code:

```
void func(float);
float a = ...;
float b;
if (a < 0) {
    b = 0;
    unmasked {
        if (b == 0)
            func(a);
    }
}
```

The variable `a` is initialized to some value and `b` is declared but not initialized, and thus has an undefined value. Within the `if` test, we have assigned zero to `b`, though only for the program instances currently executing--i.e. those where `a < 0`. After re-establishing the executing mask with `unmasked`, we then compare `b` to zero--this comparison is well-defined (and "true") for the program instances where `a < 0`, but it is undefined for any program instances where that isn't the case, since the value of `b` is undefined for those program instances. Similar surprising cases can arise when writing to `varying` variables within `unmasked` code.

As a general rule, code within an `unmasked` block, or a function with the `unmasked` qualifier should use great care when accessing `varying` variables that were declared in an outer scope.

### Task Parallel Execution

In addition to the facilities for using SPMD for parallelism across the SIMD lanes of one processing core, `ispc` also provides facilities for parallel execution across multiple cores though an asynchronous function call mechanism via the `launch` keyword. A function called with `launch` executes as an asynchronous task, often on another core in the system.

### Task Parallelism: "launch" and "sync" Statements

One option for combining task-parallelism with `ispc` is to just use regular task parallelism in the C/C++ application code (be it through Intel® Thread Building Blocks, OpenMP or another task system), and for tasks to use `ispc` for SPMD parallelism across the vector lanes as appropriate. Alternatively, `ispc` also has support for launching tasks from `ispc` code. (Check the `examples/mandelbrot_tasks` example to see how it is used.)

Any function that is launched as a task must be declared with the `task` qualifier:

```
task void func(uniform float a[], uniform int index) {
    ...
    a[index] = ....
}
```

Tasks must return `void`; a compile time error is issued if a non-`void` task is defined.

Given a task declaration, a task can be launched with `launch`:

```
uniform float a[...] = ...;
launch func(a, 1);
```

Program execution continues asynchronously after a `launch` statement in a function; thus, a function shouldn't access values written by a task it has launched within the function without synchronization. A function can use a `sync` statement to wait for all launched tasks to finish:

```
launch func(a, 1);
sync;
// now safe to use computed values in a[]...
```

Alternatively, any function that launches tasks has an automatically-added implicit `sync` statement before it returns, so that functions that call a function that launches tasks don't have to worry about outstanding asynchronous computation from that function.

The task generated by a `launch` statement is a single gang's worth of work. The same program instances are respectively active and inactive at the start of the task as were active and inactive when their `launch`

statement executed. To make all program instances in the launched gang be active, the `unmasked` construct can be used (see Re-establishing The Execution Mask.)

There are two ways to write code that launches a group multiple tasks. First, one task can be launched at a time, with parameters passed to the task to help it determine what part of the overall computation it's responsible for:

```
for (uniform int i = 0; i < 100; ++i)
    launch func(a, i);
```

This code launches 100 tasks, each of which presumably does some computation that is keyed off of given the value `i`. In general, one should launch many more tasks than there are processors in the system to ensure good load-balancing, but not so many that the overhead of scheduling and running tasks dominates the computation.

Alternatively, a number of tasks may be launched from a single `launch` statement. We might instead write the above example with a single `launch` like this:

```
launch[100] func2(a);
```

Where an integer value (not necessarily a compile-time constant) is provided to the `launch` keyword in square brackets; this number of tasks will be enqueued to be run asynchronously. Within each of the tasks, two special built-in variables are available--`taskIndex`, and `taskCount`. The first, `taskIndex`, ranges from zero to one minus the number of tasks provided to `launch`, and `taskCount` equals the number of launched tasks. Thus, in this example we might use `taskIndex` in the implementation of `func2` to determine which array element to process.

```
task void func2(uniform float a[]) {
    ...
    a[taskIndex] = ...
}
```

Inside functions with the `task` qualifier, two additional built-in variables are provided in addition to `taskIndex` and `taskCount`: `threadIndex` and `threadCount`. `threadCount` gives the total number of hardware threads that have been launched by the task system. `threadIndex` provides an index between zero and `threadCount-1` that gives a unique index that corresponds to the hardware thread that is executing the current task. The `threadIndex` can be used for accessing data that is private to the current thread and thus doesn't require synchronization to access under parallel execution.

The tasking system also supports multi-dimensional partitioning (currently up to three dimensions). To launch a 3D grid of tasks, for example with `N0`, `N1` and `N2` tasks in x-, y- and z-dimension respectively

```
float data[N2][N1][N0]
task void foo_task()
{
   data[taskIndex2][taskIndex1][threadIndex0] = taskIndex;
}
```

we use the following `launch` expressions:

```
launch [N2][N1][N0] foo_task()
```

or

```
launch [N0,N1,N2] foo_task()
```

Value of `taskIndex` is equal to `taskIndex0 + taskCount0*(taskIndex1 + taskCount1*taskIndex2)` and it ranges from `0` to `taskCount-1`, where `taskCount = taskCount0*taskCount1*taskCount2`. If `N1` or/and `N2` are not specified in the `launch` expression, a value of `1` is assumed. Finally, for an one-dimensional grid of tasks, `taskIndex` is equivalent to `taskIndex0` and `taskCount` is equivalent to `taskCount0`.

### Task Parallelism: Runtime Requirements

If you use the task launch feature in `ispc`, you must provide C/C++ implementations of three specific functions that manage launching and synchronizing parallel tasks; these functions must be linked into your executable. Although these functions may be implemented in any language, they must have "C" linkage (i.e. their prototypes must be declared inside an `extern "C"` block if they are defined in C++.)

By using user-supplied versions of these functions, `ispc` programs can easily interoperate with software systems that have existing task systems for managing parallelism. If you're using `ispc` with a system that isn't otherwise multi-threaded and don't want to write custom implementations of them, you can use the implementations of these functions provided in the `examples/common/tasksys.cpp` file in the `ispc` distributions.

If you are implementing your own task system, the remainder of this section discusses the requirements for these calls. You will also likely want to review the example task systems in

examples/common/tasksys.cpp for reference. If you are not implementing your own task system, you can skip reading the remainder of this section.

Here are the declarations of the three functions that must be provided to manage tasks in ispc:

```
void *ISPCAlloc(void **handlePtr, int64_t size, int32_t alignment);
void ISPCLaunch(void **handlePtr, void *f, void *data, int count0, int count1, in
void ISPCSync(void *handle);
```

All three of these functions take an opaque handle (or a pointer to an opaque handle) as their first parameter. This handle allows the task system runtime to distinguish between calls to these functions from different functions in ispc code. In this way, the task system implementation can efficiently wait for completion on just the tasks launched from a single function.

The first time one of ISPCLaunch() or ISPCAlloc() is called in an ispc function, the void * pointed to by the handlePtr parameter will be NULL. The implementations of these function should then initialize *handlePtr to a unique handle value of some sort. (For example, it might allocate a small structure to record which tasks were launched by the current function.) In subsequent calls to these functions in the emitted ispc code, the same value for handlePtr will be passed in, such that loading from *handlePtr will retrieve the value stored in the first call.

At function exit (or at an explicit sync statement), a call to ISPCSync() will be generated if *handlePtr is non-NULL. Therefore, the handle value is passed directly to ISPCSync(), rather than a pointer to it, as in the other functions.

The ISPCAlloc() function is used to allocate small blocks of memory to store parameters passed to tasks. It should return a pointer to memory with the given size and alignment. Note that there is no explicit ISPCFree() call; instead, all memory allocated within an ispc function should be freed when ISPCSync() is called.

ISPCLaunch() is called to launch one or more asynchronous tasks. Each launch statement in ispc code causes a call to ISPCLaunch() to be emitted in the generated code. The three parameters after the handle pointer to the function are relatively straightforward; the void *f parameter holds a pointer to a function to call to run the work for this task, data holds a pointer to data to pass to this function, and count0, count1 and count2 are the number of instances of this function to enqueue for asynchronous execution. (In other words, count0, count1 and count2 correspond to the value n0, n1 and n2 in a multiple-task launch statement like launch[n2][n1][n0] or launch [n0,n1,n2] respectively.)

The signature of the provided function pointer f is

```
void (*TaskFuncPtr)(void *data, int threadIndex, int threadCount,
                    int taskIndex, int taskCount,
                    int taskIndex0, int taskIndex1, int taskIndex2,
                    int taskCount0, int taskCount1, int taskCount2);
```

When this function pointer is called by one of the hardware threads managed by the task system, the data pointer passed to ISPCLaunch() should be passed to it for its first parameter; threadCount gives the total number of hardware threads that have been spawned to run tasks and threadIndex should be an integer index between zero and threadCount uniquely identifying the hardware thread that is running the task. (These values can be used to index into thread-local storage.)

The value of taskCount should be the total number of tasks launched in the launch statement (it must be equal to taskCount0*taskCount1*taskCount2) that caused the call to ISPCLaunch() and each of the calls to this function should be given a unique value of taskIndex, taskIndex0, taskIndex1 and taskIndex2 between zero and taskCount, taskCount0, taskCount1 and taskCount2 respectively, with taskIndex = taskIndex0 + taskCount0*(taskIndex1 + taskCount1*taskIndex2), to distinguish which of the instances of the set of launched tasks is running.

### LLVM Intrinsic Functions

ispc has an experimental feature to call LLVM intrinsics directly from ispc source code. It's strongly discouraged to use this feature in production code, unless the consequences are well understood. Specifically:

- Availability and naming of LLVM intrinsics depend on the specific LLVM version used for ispc build and may change without notice.
- Only basic verification of availability of target-specific intrinsics on the target CPU is performed. The attempt of using not supported intrinsics may lead to compiler crash.

Using LLVM intrinsics is encouraged for experiments and may be useful in the following cases:

- If ispc fails to generate specific instruction, which is necessary for better performance.
- If there's no higher level primitives (in standard library or language itself) for some of instructions. For example, this might be the case with new ISA extensions.

If you found the case where the use of LLVM intrinsics is beneficial in your code, please let us know by opening an issue in ispc bug tracker.

To use this feature, --enable-llvm-intrinsics switch must be passed to ispc. The syntax is similar to a normal function call, but the name must start with @ symbol. For example:

```
transpose = @llvm.matrix.transpose.v8f32.i32.i32(matrix, row, column);
```

To detect if this feature is enabled during compile time, check if `ISPC_LLVM_INTRINSICS_ENABLED` macro is defined.

## The ISPC Standard Library

`ispc` has a standard library that is automatically available when compiling `ispc` programs. (To disable the standard library, pass the `--nostdlib` command-line flag to the compiler.)

### Basic Operations On Data

### Logical and Selection Operations

Recall from Expressions that `ispc` short-circuits the evaluation of logical and selection operators: given an expression like `(index < count && array[index] == 0)`, then `array[index] == 0` is only evaluated if `index < count` is true. This property is useful for writing expressions like the preceding one, where the second expression may not be safe to evaluate in some cases.

This short-circuiting can impose overhead in the generated code; additional operations are required to test the first value and to conditionally jump over the code that evaluates the second value. The `ispc` compiler does try to mitigate this cost by detecting cases where it is both safe and inexpensive to evaluate both expressions, and skips short-circuiting in the generated code in this case (without there being any programmer-visible change in program behavior.)

For cases where the compiler can't detect this case but the programmer wants to avoid short-circuiting behavior, the standard library provides a few helper functions. First, `and()` and `or()` provide non-short-circuiting logical AND and OR operations.

```
bool and(bool a, bool b)
bool or(bool a, bool b)
uniform bool and(uniform bool a, uniform bool b)
uniform bool or(uniform bool a, uniform bool b)
```

And there are three variants of `select()` that select between two values based on a boolean condition. If condition `cond` is true, `t` is selected, otherwise `f`. These are the variants of `select()` for the `int8` type:

```
int8 select(bool cond, int8 t, int8 f)
int8 select(uniform bool cond, int8 t, int8 f)
uniform int8 select(uniform bool cond, uniform int8 t, uniform int8 f)
```

There are also variants for `int16`, `int32`, `int64`, `float`, `float16` and `double` types.

### Bit Operations

The various variants of `popcnt()` return the population count--the number of bits set in the given value.

```
uniform int popcnt(uniform int v)
int popcnt(int v)
uniform int popcnt(bool v)
```

A few functions determine how many leading bits in the given value are zero and how many of the trailing bits are zero; there are also `unsigned` variants of these functions and variants that take `int64` and `unsigned int64` types.

```
int32 count_leading_zeros(int32 v)
uniform int32 count_leading_zeros(uniform int32 v)
int32 count_trailing_zeros(int32 v)
uniform int32 count_trailing_zeros(uniform int32 v)
```

Sometimes it's useful to convert a `bool` value to an integer using sign extension so that the integer's bits are all on if the `bool` has the value `true` (rather than just having the value one). The `sign_extend()` functions provide this functionality:

```
int sign_extend(bool value)
uniform int sign_extend(uniform bool value)
```

Also it is possible to convert a `bool` varying value to an integer using `packmask` function.

```
uniform int packmask(bool value)
```

The `intbits()`, `float16bits()`, `floatbits()` and `doublebits()` functions can be used to implement low-level floating-point bit twiddling. For example, `intbits()` returns an `unsigned int` that is a bit-for-bit copy of the given `float` value. (Note: it is **not** the same as `(int)a`, but corresponds to something like `*((int *)&a)` in C.

```
float16 float16bits(unsigned int16 a);
uniform float16 float16bits(uniform unsigned int16 a);
float floatbits(unsigned int a);
uniform float floatbits(uniform unsigned int a);
double doublebits(unsigned int64 a);
uniform double doublebits(uniform unsigned int64 a);
unsigned int16 intbits(float16 a);
uniform unsigned int16 intbits(uniform float16 a);
unsigned int intbits(float a);
uniform unsigned int intbits(uniform float a);
unsigned int64 intbits(double a);
uniform unsigned int64 intbits(uniform double a);
```

The `intbits()`, `float16bits()`, `floatbits()` and `doublebits()` functions have no cost at runtime; they just let the compiler know how to interpret the bits of the given value. They make it possible to efficiently write functions that take advantage of the low-level bit representation of floating-point values.

For example, the `abs()` function in the standard library is implemented as follows:

```
float abs(float a) {
    unsigned int i = intbits(a);
    i &= 0x7fffffff;
    return floatbits(i);
}
```

This code directly clears the high order bit to ensure that the given floating-point value is positive. This compiles down to a single `andps` instruction when used with an Intel® SSE target, for example.

### Math Functions

The math functions in the standard library provide a relatively standard range of mathematical functionality.

A number of different implementations of the transcendental math functions are available; the math library to use can be selected with the `--math-lib=` command line argument. The following values can be provided for this argument.

- `default`: `ispc`'s default built-in math functions. These have reasonably high precision. (e.g. `sin` has a maximum absolute error of approximately 1.45e-6 over the range -10pi to 10pi.)
- `fast`: more efficient but lower accuracy versions of the default `ispc` implementations.
- `svml`: use Intel "Short Vector Math Library". This is a proprietary library shipped as part of Intel® oneAPI DPC++/C++ Compiler (`icx/icpx`) and Intel® oneAPI C++ Compiler Classic (`icc/icpc`). Use either of them to link your final executable so that the appropriate libraries are linked.
- `system`: use the system's math library. On many systems, these functions are more accurate than both of `ispc`'s implementations. Using these functions may be quite inefficient; the system math functions only compute one result at a time (i.e. they aren't vectorized), so `ispc` has to call them once per active program instance. (This is not the case for the other three options.)

### Basic Math Functions

In addition to an absolute value call, `abs()`, `signbits()` extracts the sign bit of the given value, returning `0x80000000` if the sign bit is on (i.e. the value is negative) and zero if it is off.

```
float16 abs(float a)
uniform float16 abs(uniform float a)
float abs(float a)
uniform float abs(uniform float a)
double abs(double a)
uniform double abs(uniform double a)
int8 abs(int8 a)
uniform int8 abs(uniform int8 a)
int16 abs(int16 a)
uniform int16 abs(uniform int16 a)
int abs(int a)
uniform int abs(uniform int a)
int64 abs(int64 a)
uniform int64 abs(uniform int64 a)
unsigned int16 signbits(float16 x)
uniform unsigned int16 signbits(uniform float16 x)
unsigned int signbits(float x)
uniform unsigned int signbits(uniform float x)
unsigned int64 signbits(double x)
uniform unsigned int64 signbits(uniform double x)
```

Standard rounding functions are provided for `float16`, `float` and `double` types. (On machines that support Intel®SSE or Intel® AVX, these functions all map to variants of the `roundss` and `roundps` instructions,

respectively.)

```
float round(float x)
uniform float round(uniform float x)
float floor(float x)
uniform float floor(uniform float x)
float ceil(float x)
uniform float ceil(uniform float x)
float trunc(float x)
uniform float trunc(uniform float x)
```

`rcp()` computes an approximation to `1/v`. The amount of error is different on different architectures.

```
float rcp(float v)
uniform float rcp(uniform float v)
```

ispc also provides a version of `rcp()` for float with less precision which doesn't use Newton-Raphson.

```
float rcp_fast(float v)
uniform float rcp_fast(uniform float v)
```

A standard set of minimum and maximum functions is available for all ispc standard types. These functions also map to corresponding intrinsic functions.

```
float min(float a, float b)
uniform float min(uniform float a, uniform float b)
float max(float a, float b)
uniform float max(uniform float a, uniform float b)
unsigned int min(unsigned int a, unsigned int b)
uniform unsigned int min(uniform unsigned int a,
                         uniform unsigned int b)
unsigned int max(unsigned int a, unsigned int b)
uniform unsigned int max(uniform unsigned int a,
                         uniform unsigned int b)
```

The `clamp()` functions clamp the provided value to the given range. (Their implementations are based on `min()` and `max()` and are thus quite efficient.)

```
float clamp(float v, float low, float high)
uniform float clamp(uniform float v, uniform float low,
                    uniform float high)
unsigned int clamp(unsigned int v, unsigned int low,
                   unsigned int high)
uniform unsigned int clamp(uniform unsigned int v,
                           uniform unsigned int low,
                           uniform unsigned int high)
```

The `isnan()` functions test whether the given value is a floating-point "not a number" value:

```
bool isnan(float16 v)
uniform bool isnan(uniform float16 v)
bool isnan(float v)
uniform bool isnan(uniform float v)
bool isnan(double v)
uniform bool isnan(uniform double v)
```

A number of functions are also available for performing operations on 8- and 16-bit quantities; these map to specialized instructions that perform these operations on targets that support them. `avg_up()` computes the average of the two values, rounding up if their average is halfway between two integers (i.e., it computes `(a+b+1)/2`).

```
int8 avg_up(int8 a, int8 b)
unsigned int8 avg_up(unsigned int8 a, unsigned int8 b)
int16 avg_up(int16 a, int16 b)
unsigned int16 avg_up(unsigned int16 a, unsigned int16 b)
```

`avg_down()` computes the average of the two values, rounding down (i.e., it computes `(a+b)/2`).

```
int8 avg_down(int8 a, int8 b)
unsigned int8 avg_down(unsigned int8 a, unsigned int8 b)
int16 avg_down(int16 a, int16 b)
unsigned int16 avg_down(unsigned int16 a, unsigned int16 b)
```

### Transcendental Functions

The square root of a given value can be computed with `sqrt()`, which maps to hardware square root intrinsics when available. An approximate reciprocal square root, `1/sqrt(v)` is computed by `rsqrt()`. Like `rcp()`, the error from this call is different on different architectures.

```
float sqrt(float v)
uniform float sqrt(uniform float v)
float rsqrt(float v)
uniform float rsqrt(uniform float v)
```

ispc also provides a version of `rsqrt()` for float with less precision which doesn't use Newton-Raphson.

```
float rsqrt_fast(float v)
uniform float rsqrt_fast(uniform float v)
```

`ispc` provides a standard variety of calls for trigonometric functions:

```
float sin(float x)
uniform float sin(uniform float x)
float cos(float x)
uniform float cos(uniform float x)
float tan(float x)
uniform float tan(uniform float x)
```

The corresponding inverse functions are also available:

```
float asin(float x)
uniform float asin(uniform float x)
float acos(float x)
uniform float acos(uniform float x)
float atan(float x)
uniform float atan(uniform float x)
float atan2(float y, float x)
uniform float atan2(uniform float y, uniform float x)
```

If both sine and cosine are needed, then the `sincos()` call computes both more efficiently than two calls to the respective individual functions:

```
void sincos(float x, varying float * uniform s, varying float * uniform c)
void sincos(uniform float x, uniform float * uniform s,
            uniform float * uniform c)
```

The usual exponential and logarithmic functions are provided.

```
float exp(float x)
uniform float exp(uniform float x)
float log(float x)
uniform float log(uniform float x)
float pow(float a, float b)
uniform float pow(uniform float a, uniform float b)
```

A few functions that end up doing low-level manipulation of the floating-point representation in memory are available. As in the standard math library, `ldexp()` multiplies the value `x` by `2^n`, and `frexp()` directly returns the normalized mantissa and returns the normalized exponent as a power of two in the `pw2` parameter.

```
float ldexp(float x, int n)
uniform float ldexp(uniform float x, uniform int n)
float frexp(float x, varying int * uniform pw2)
uniform float frexp(uniform float x,
                    uniform int * uniform pw2)
```

All transcendental functions are provided for `float16`, `float` and `double` types.

### Saturating Arithmetic

A saturation (no overflow possible) addition, substraction, multiplication and division of all integer types are provided by the `ispc` standard library.

```
int8 saturating_add(uniform int8 a, uniform int8 b)
int8 saturating_add(varying int8 a, varying int8 b)
unsigned int8 saturating_add(uniform unsigned int8 a, uniform unsigned int8 b)
unsigned int8 saturating_add(varying unsigned int8 a, varying unsigned int8 b)
```

```
int8 saturating_sub(uniform int8 a, uniform int8 b)
int8 saturating_sub(varying int8 a, varying int8 b)
unsigned int8 saturating_sub(uniform unsigned int8 a, uniform unsigned int8 b)
unsigned int8 saturating_sub(varying unsigned int8 a, varying unsigned int8 b)

int8 saturating_mul(uniform int8 a, uniform int8 b)
int8 saturating_mul(varying int8 a, varying int8 b)
unsigned int8 saturating_mul(uniform unsigned int8 a, uniform unsigned int8 b)
unsigned int8 saturating_mul(varying unsigned int8 a, varying unsigned int8 b)

int8 saturating_div(uniform int8 a, uniform int8 b)
int8 saturating_div(varying int8 a, varying int8 b)
unsigned int8 saturating_div(uniform unsigned int8 a, uniform unsigned int8 b)
unsigned int8 saturating_div(varying unsigned int8 a,varying unsigned int8 b)
```

In addition to the `int8` variants of saturating arithmetic functions listed above, there are versions that supports `int16`, `int32` and `int64` values as well.

### Pseudo-Random Numbers

A simple random number generator is provided by the `ispc` standard library. State for the RNG is maintained in an instance of the `RNGState` structure, which is seeded with `seed_rng()`.

```
struct RNGState;
void seed_rng(varying RNGState * uniform state, varying int seed)
void seed_rng(uniform RNGState * uniform state, uniform int seed)
```

Note that if the same `varying` seed value is used for all of the program instances (e.g. `RNGState state; seed_rng(&state, 1);`), then all of the program instances in the gang will see the same sequence of pseudo-random numbers. If this behavior isn't desred, you may want to add the `programIndex` value to the provided seed or otherwise ensure that the seed has a unique value for each program instance.

After the RNG is seeded, the `random()` function can be used to get a pseudo-random `unsigned int32` value and the `frandom()` function can be used to get a pseudo-random `float` value.

```
unsigned int32 random(varying RNGState * uniform state)
float frandom(varying RNGState * uniform state)
uniform unsigned int32 random(RNGState * uniform state)
uniform float frandom(uniform RNGState * uniform state)
```

### Random Numbers

Some recent CPUs (including those based on the Intel® Ivy Bridge micro-architecture), provide support for generating true random numbers. A few standard library functions make this functionality available:

```
bool rdrand(uniform int32 * uniform ptr)
bool rdrand(varying int32 * uniform ptr)
bool rdrand(uniform int32 * varying ptr)
```

If the processor doesn't have sufficient entropy to generate a random number, then this function fails and returns `false`. Otherwise, if the processor is successful, the random value is stored in the given pointer and `true` is returned. Therefore, this function should generally be used as follows, called repeatedly until it is successful:

```
int r;
while (rdrand(&r) == false)
    ; // empty loop body
```

In addition to the `int32` variants of `rdrand()` listed above, there are versions that return `int16`, `float`, and `int64` values as well.

Note that when compiling to targets older than `avx2`, the `rdrand()` functions always return `false`.

### Output Functions

`ispc` has a simple `print` statement for printing values during program execution. In the following short `ispc` program, there are three uses of the `print` statement:

```
export void foo(uniform float f[4], uniform int i) {
    float x = f[programIndex];
    print("i = %, x = %\n", i, x);
    if (x < 2) {
        ++x;
        print("added to x = %\n", x);
    }
```

```
    print("last print of x = %\n", x);
}
```

There are a few things to note. First, the function is called `print`, not `printf` (unlike C). Second, the formatting string passed to this function only uses a single percent sign to denote where the corresponding value should be printed. You don't need to match the types of formatting operators with the types being passed. However, you can't currently use the rich data formatting options that `printf` provides (e.g. constructs like `%.10f`.).

If this function is called with the array of floats (0,1,2,3) passed in for the `f` parameter and the value 10 for the `i` parameter, it generates the following output on a four-wide compilation target:

```
i = 10, x = [0.000000,1.000000,2.000000,3.000000]
added to x = [1.000000,2.000000,((2.000000)),((3.000000))]
last print of x = [1.000000,2.000000,2.000000,3.000000]
```

When a varying variable is printed, the values for program instances that aren't currently executing are printed inside double parenthesis, indicating inactive program instances. The elements for inactive program instances may have garbage values, though in some circumstances it can be useful to see their values.

### Assertions

The `ispc` standard library includes a mechanism for adding `assert()` statements to `ispc` program code. Like `assert()` in C, the `assert()` function takes a single boolean expression as an argument. If the expression evaluates to false at runtime, then a diagnostic error message printed and the `abort()` function is called.

When called with a `varying` quantity, an assertion triggers if the expression evaluates to false for any of the executing program instances at the point where it is called. Thus, given code like:

```
int x = programIndex - 2;  // (-2, -1, 0, ... )
if (x > 0)
    assert(x > 0);
```

The `assert()` statement will not trigger, since the condition isn't true for any of the executing program instances at that point. (If this `assert()` statement was outside of this `if`, then it would of course trigger.)

To disable all of the assertions in a file that is being compiled (e.g., for an optimized release build), use the `--opt=disable-assertions` command-line argument.

### Compiler Optimization Hints

The `ispc` standard library includes a mechanism for adding `assume()` statements to `ispc` program code. The `assume()` function takes a single uniform boolean expression as an argument. This expression is assumed to be `true` and this information will be used for optimization when possible.

The condition used in `assume()` statement will not be code generated and does not imply runtime checks. It will solely be used as optimization hint, if compiler is able to use this information.

Below are some basic examples of this functionality.

```
inline uniform int bar1(uniform int a, uniform int b) {
    if (a < b)
        return 2;
    return 5;
}
uniform int foo1(uniform int a, uniform int b) {
    assume(a < b);
    return bar1(a, b);
}
```

The `assume()` hint allows the compiler to resolve `a < b` during compile time in `bar1()` and return 2 thus removing the additional branch.

```
inline void bar2(uniform int * uniform a) {
    if (a != NULL) {
        a[2] = 9;
    }
 }
 void foo2(uniform int a[]) {
    assume(a != NULL);
    bar2(a);
 }
```

The `assume()` hint allows the compiler to remove `a != NULL` during compile time in `bar2()` thus removing the additional check.

```
int foo3(uniform int a[], uniform int count) {
    int ret = 0;
    assume(count % programCount == 0);
    foreach (i = 0 ... count) {
        ret += a[i];
    }
    return ret;
}
```

The `assume()` hint informs the compiler `count` is a multiple of `programCount` during compile time. This results in removal of remainder loop usually required for `foreach`.

```
typedef float<TARGET_WIDTH> AlignedFloat;
unmasked void foo4(uniform float Result[], const uniform float Source1[], const u
{
     assume(((uniform uint64)((void*)Source1) & (32 * TARGET_WIDTH)-1) == 0);
     assume(((uniform uint64)((void*)Result) & (32 * TARGET_WIDTH)-1) == 0);
     uniform AlignedFloat S1;
     S1[programIndex] = Source1[programIndex];
     const uniform AlignedFloat R = S1;
     Result[programIndex] = R[programIndex];
}
```

The `assume()` hint informs the compiler that memory locations used by loads and stores are aligned. This results in aligned instructions instead of unaligned instructions.

### Cross-Program Instance Operations

`ispc` programs are often used to express independently-executing programs performing computation on separate data elements. (i.e. pure data-parallelism). However, it's often the case where it's useful for the program instances to be able to cooperate in computing results. The cross-lane operations described in this section provide primitives for communication between the running program instances in the gang.

The `lanemask()` function returns an integer that encodes which of the current SPMD program instances are currently executing. The i'th bit is set if the i'th program instance lane is currently active.

```
uniform int lanemask()
```

To broadcast a value from one program instance to all of the others, a `broadcast()` function is available. It broadcasts the value of the `value` parameter for the program instance given by `index` to all of the running program instances.

```
int8 broadcast(int8 value, uniform int index)
int16 broadcast(int16 value, uniform int index)
int32 broadcast(int32 value, uniform int index)
int64 broadcast(int64 value, uniform int index)
float16 broadcast(float16 value, uniform int index)
float broadcast(float value, uniform int index)
double broadcast(double value, uniform int index)
```

The `rotate()` function allows each program instance to find the value of the given value that their neighbor `offset` steps away has. For example, on an 8-wide target, if `value` has the value (1, 2, 3, 4, 5, 6, 7, 8) across the gang of running program instances, then `rotate(value, -1)` causes the first program instance to get the value 8, the second program instance to get the value 1, the third 2, and so forth. The provided offset value can be positive or negative, and may be greater than the size of the gang (it is masked to ensure valid offsets).

```
int8 rotate(int8 value, uniform int offset)
int16 rotate(int16 value, uniform int offset)
int32 rotate(int32 value, uniform int offset)
int64 rotate(int64 value, uniform int offset)
float16 rotate(float16 value, uniform int offset)
float rotate(float value, uniform int offset)
double rotate(double value, uniform int offset)
```

The `shift()` function allows each program instance to find the value of the given value that their neighbor `offset` steps away has. This is similar to `rotate()` with the exception that values are not circularly shifted. Instead, zeroes are shifted in where appropriate.

```
int8 shift(int8 value, uniform int offset)
int16 shift(int16 value, uniform int offset)
int32 shift(int32 value, uniform int offset)
int64 shift(int64 value, uniform int offset)
```

```
float16 shift(float16 value, uniform int offset)
float shift(float value, uniform int offset)
double shift(double value, uniform int offset)
```

Finally, the `shuffle()` functions allow two variants of fully general shuffling of values among the program instances. For the first version, each program instance's value of permutation gives the program instance from which to get the value of `value`. The provided values for `permutation` must all be between 0 and the gang size.

```
int8 shuffle(int8 value, int permutation)
int16 shuffle(int16 value, int permutation)
int32 shuffle(int32 value, int permutation)
int64 shuffle(int64 value, int permutation)
float16 shuffle(float16 value, int permutation)
float shuffle(float value, int permutation)
double shuffle(double value, int permutation)
```

The second variant of `shuffle()` permutes over the extended vector that is the concatenation of the two provided values. In other words, a value of 0 in an element of `permutation` corresponds to the first element of `value0`, the value of two times the gang size, minus one corresponds to the last element of `value1`, etc.)

```
int8 shuffle(int8 value0, int8 value1, int permutation)
int16 shuffle(int16 value0, int16 value1, int permutation)
int32 shuffle(int32 value0, int32 value1, int permutation)
int64 shuffle(int64 value0, int64 value1, int permutation)
float16 shuffle(float16 value0, float16 value1, int permutation)
float shuffle(float value0, float value1, int permutation)
double shuffle(double value0, double value1, int permutation)
```

Finally, there are primitive operations that extract and set values in the SIMD lanes. You can implement all of the broadcast, rotate, shift, and shuffle operations described above in this section from these routines, though in general, not as efficiently. These routines are useful for implementing other reductions and cross-lane communication that isn't included in the above, though. Given a `varying` value, `extract()` returns the i'th element of it as a single `uniform` value. .

```
uniform bool extract(bool x, uniform int i)
uniform int8 extract(int8 x, uniform int i)
uniform int16 extract(int16 x, uniform int i)
uniform int32 extract(int32 x, uniform int i)
uniform int64 extract(int64 x, uniform int i)
uniform float16 extract(float16 x, uniform int i)
uniform float extract(float x, uniform int i)
uniform double extract(double x, uniform int i)
```

Similarly, `insert` returns a new value where the `i` th element of `x` has been replaced with the value `v`

```
bool insert(bool x, uniform int i, uniform bool v)
int8 insert(int8 x, uniform int i, uniform int8 v)
int16 insert(int16 x, uniform int i, uniform int16 v)
int32 insert(int32 x, uniform int i, uniform int32 v)
int64 insert(int64 x, uniform int i, uniform int64 v)
float16 insert(float16 x, uniform int i, uniform float16 v)
float insert(float x, uniform int i, uniform float v)
double insert(double x, uniform int i, uniform double v)
```

### Reductions

A number of routines are available to evaluate conditions across the running program instances. For example, `any()` returns `true` if the given value `v` is `true` for any of the SPMD program instances currently running, `all()` returns `true` if it true for all of them, and `none()` returns `true` if `v` is always `false`.

```
uniform bool any(bool v)
uniform bool all(bool v)
uniform bool none(bool v)
```

You can also compute a variety of reductions across the program instances. For example, the values of the given value in each of the active program instances are added together by the `reduce_add()` function.

```
uniform int16 reduce_add(int8 x)
uniform unsigned int16 reduce_add(unsigned int8 x)
uniform int32 reduce_add(int16 x)
uniform unsigned int32 reduce_add(unsigned int16 x)
uniform int64 reduce_add(int32 x)
uniform unsigned int64 reduce_add(unsigned int32 x)
```

```
uniform int64 reduce_add(int64 x)
uniform unsigned int64 reduce_add(unsigned int64 x)

uniform float16 reduce_add(float16 x)
uniform float reduce_add(float x)
uniform double reduce_add(double x)
```

You can also use functions to compute the minimum value of the given value across all of the currently-executing program instances.

```
uniform int32 reduce_min(int32 a)
uniform unsigned int32 reduce_min(unsigned int32 a)
uniform int64 reduce_min(int64 a)
uniform unsigned int64 reduce_min(unsigned int64 a)

uniform float16 reduce_min(float16 a)
uniform float reduce_min(float a)
uniform double reduce_min(double a)
```

Equivalent functions are available to comptue the maximum of the given varying variable over the active program instances.

```
uniform int32 reduce_max(int32 a)
uniform unsigned int32 reduce_max(unsigned int32 a)
uniform int64 reduce_max(int64 a)
uniform unsigned int64 reduce_max(unsigned int64 a)

uniform float16 reduce_max(float16 a)
uniform float reduce_max(float a)
uniform double reduce_max(double a)
```

Finally, you can check to see if a particular value has the same value in all of the currently-running program instances:

```
uniform bool reduce_equal(int32 v)
uniform bool reduce_equal(unsigned int32 v)
uniform bool reduce_equal(int64 v)
uniform bool reduce_equal(unsigned int64 v)

uniform bool reduce_equal(float16 v)
uniform bool reduce_equal(float v)
uniform bool reduce_equal(double)
```

There are also variants of these functions that return the value as a `uniform` in the case where the values are all the same. (There is discussion of an application of this variant to improve memory access performance in the Performance Guide.

```
uniform bool reduce_equal(int32 v, uniform int32 * uniform sameval)
uniform bool reduce_equal(unsigned int32 v,
                          uniform unsigned int32 * uniform sameval)
uniform bool reduce_equal(int64 v, uniform int64 * uniform sameval)
uniform bool reduce_equal(unsigned int64 v,
                          uniform unsigned int64 * uniform sameval)

uniform bool reduce_equal(float16 v, uniform float16 * uniform sameval)
uniform bool reduce_equal(float v, uniform float * uniform sameval)
uniform bool reduce_equal(double, uniform double * uniform sameval)
```

If called when none of the program instances are running, `reduce_equal()` will return `false`.

There are also a number of functions to compute "scan"s of values across the program instances. For example, the `exclusive_scan_add()` function computes, for each program instance, the sum of the given value over all of the preceding program instances. (The scans currently available in `ispc` are all so-called "exclusive" scans, meaning that the value computed for a given element does not include the value provided for that element.) In C code, an exclusive add scan over an array might be implemented as:

```
void scan_add(int *in_array, int *result_array, int count) {
    result_array[0] = 0;
    for (int i = 1; i < count; ++i)
        result_array[i] = result_array[i-1] + in_array[i-1];
}
```

`ispc` provides the following scan functions--addition, bitwise-and, and bitwise-or are available:

```
int32 exclusive_scan_add(int32 v)
unsigned int32 exclusive_scan_add(unsigned int32 v)
float16 exclusive_scan_add(float16 v)
float exclusive_scan_add(float v)
int64 exclusive_scan_add(int64 v)
unsigned int64 exclusive_scan_add(unsigned int64 v)
double exclusive_scan_add(double v)
int32 exclusive_scan_and(int32 v)
unsigned int32 exclusive_scan_and(unsigned int32 v)
int64 exclusive_scan_and(int64 v)
unsigned int64 exclusive_scan_and(unsigned int64 v)
int32 exclusive_scan_or(int32 v)
unsigned int32 exclusive_scan_or(unsigned int32 v)
int64 exclusive_scan_or(int64 v)
unsigned int64 exclusive_scan_or(unsigned int64 v)
```

The returned value for the first program instance will be `0` for `exclusive_scan_add` and `exclusive_scan_or`, and have all bits set to `1` for `exclusive_scan_and`.

The use of exclusive scan to generate variable amounts of output from program instances into a compact output buffer is discussed in the FAQ.

### Stack Memory Allocation

The `ispc` standard library includes an `alloca()` function that can be used to allocate memory on the stack.

```
void * uniform alloca(uniform size_t size);
```

The `alloca()` function allocates `size` bytes of space in the stack frame of the caller. This temporary space is automatically freed when the function that called `alloca()` returns to its caller.

### Data Movement

### Setting and Copying Values In Memory

There are a few functions for copying blocks of memory and initializing values in memory. Along the lines of the equivalently-named routines in the C Standard libary, `memcpy` copies a given number of bytes starting from a source location in memory to a destination locaiton, where the two regions of memory are guaranteed by the caller to be non-overlapping. Alternatively, `memmove` can be used to copy data if the buffers may overlap.

```
void memcpy(void * uniform dst, void * uniform src, uniform int32 count)
void memmove(void * uniform dst, void * uniform src, uniform int32 count)
void memcpy(void * varying dst, void * varying src, int32 count)
void memmove(void * varying dst, void * varying src, int32 count)
```

Note that there are variants of these functions that take both `uniform` and `varying` pointers. Also note that `sizeof(float)` and `sizeof(uniform float)` return different values, so programmers should take care when calculating `count`.

To initialize values in memory, the `memset` routine can be used. (It also behaves like the function of the same name in the C Standard Library.) It sets the given number of bytes of memory starting at the given location to the value provided.

```
void memset(void * uniform ptr, uniform int8 val, uniform int32 count)
void memset(void * varying ptr, int8 val, int32 count)
```

There are also variants of all of these functions that take 64-bit values for the number of bytes of memory to operate on:

```
void memcpy64(void * uniform dst, void * uniform src, uniform int64 count)
void memcpy64(void * varying dst, void * varying src, int64 count)
void memmove64(void * uniform dst, void * uniform src, uniform int64 count)
void memmove64(void * varying dst, void * varying src, int64 count)
void memset64(void * uniform ptr, uniform int8 val, uniform int64 count)
void memset64(void * varying ptr, int8 val, int64 count)
```

### Packed Load and Store Operations

The standard library also offers routines for writing out and reading in values from linear memory locations for the active program instances. The `packed_load_active()` functions load consecutive values starting at the given location, loading one consecutive value for each currently-executing program instance and storing it into that program instance's `val` variable. They return the total number of values loaded.

```
uniform int packed_load_active(uniform int * uniform base,
                               varying int * uniform val)
```

```
uniform int packed_load_active(uniform unsigned int * uniform base,
                               varying unsigned int * uniform val)
```

Similarly, the `packed_store_active()` functions store the `val` values for each program instances that executed the `packed_store_active()` call, storing the results consecutively starting at the given location. They return the total number of values stored.

```
uniform int packed_store_active(uniform int * uniform base,
                                int val)
uniform int packed_store_active(uniform unsigned int * uniform base,
                                unsigned int val)
```

There are also `packed_store_active2()` functions with exactly the same signatures and the same semantic except that they may write one extra element to the output array (but still returning the same value as `packed_store_active()`). These functions suggest different branch free implementation on most of supported targets, which usually (but not always) performs better than `packed_store_active()`. It's advised to test function performance on user's scenarios on particular target hardware before using it.

As an example of how these functions can be used, the following code shows the use of `packed_store_active()`.

```
uniform int negative_indices(uniform float a[], uniform int length,
                             uniform int indices[]) {
    uniform int numNeg = 0;
    foreach (i = 0 ... length) {
        if (a[i] < 0.)
            numNeg += packed_store_active(&indices[numNeg], i);
    }
    return numNeg;
}
```

The function takes an array of floating point values `a`, with length given by the `length` parameter. This function also takes an output array, `indices`, which is assumed to be at least as long as `length`. It then loops over all of the elements of `a` and, for each element that is less than zero, stores that element's offset into the `indices` array. It returns the total number of negative values. For example, given an input array `a[8] = { 10, -20, 30, -40, -50, -60, 70, 80 }`, it returns a count of four negative values, and initializes the first four elements of `indices[]` to the values `{ 1, 3, 4, 5 }` corresponding to the array indices where `a[i]` was less than zero.

### Streaming Load and Store Operations

The standard library offers routines for streaming load and streaming store operations. The implementation serves as both a streaming as well as a non-temporal operation. There are separate routines to be used depending on whether loading from and storing to a uniform variable or a varying variable.

The different available variants of streaming store are given below.

For storing to array from varying variable:

```
void streaming_store(uniform unsigned int8 a[], unsigned int8 vals)
void streaming_store(uniform int8 a[], int8 vals)
void streaming_store(uniform unsigned int16 a[], unsigned int16 vals)
void streaming_store(uniform int16 a[], int16 vals)
void streaming_store(uniform unsigned int a[], unsigned int vals)
void streaming_store(uniform int a[], int vals)
void streaming_store(uniform unsigned int64 a[], unsigned int64 vals)
void streaming_store(uniform int64 a[], int64 vals)
void streaming_store(uniform float16 a[], float16 vals)
void streaming_store(uniform float a[], float vals)
void streaming_store(uniform double a[], double vals)
```

For storing to array from uniform variable:

```
void streaming_store(uniform unsigned int8 a[], uniform unsigned int8 vals)
void streaming_store(uniform int8 a[], uniform int8 vals)
void streaming_store(uniform unsigned int16 a[], uniform unsigned int16 vals)
void streaming_store(uniform int16 a[], uniform int16 vals)
void streaming_store(uniform unsigned int a[], uniform unsigned int vals)
void streaming_store(uniform int a[], uniform int vals)
void streaming_store(uniform unsigned int64 a[], uniform unsigned int64 vals)
void streaming_store(uniform int64 a[], uniform int64 vals)
void streaming_store(uniform float16 a[], uniform float16 vals)
void streaming_store(uniform float a[], uniform float vals)
void streaming_store(uniform double a[], uniform double vals)
```

The different available variants of streaming load are given below.

For loading as varying from array:

```
varying unsigned int8 streaming_load(uniform unsigned int8 a[])
varying int8 streaming_load(uniform int8 a[])
varying unsigned int16 streaming_load(uniform unsigned int16 a[])
varying int16 streaming_load(uniform int16 a[])
varying unsigned int streaming_load(uniform unsigned int a[])
varying int streaming_load(uniform int a[])
varying unsigned int64 streaming_load(uniform unsigned int64 a[])
varying int64 streaming_load(uniform int64 a[])
varying float16 streaming_load(uniform float16 a[])
varying float streaming_load(uniform float a[])
varying double streaming_load(uniform double a[])
```

For loading as uniform from array:

```
uniform unsigned int8 streaming_load_uniform(uniform unsigned int8 a[])
uniform int8 streaming_load_uniform(uniform int8 a[])
uniform unsigned int16 streaming_load_uniform(uniform unsigned int16 a[])
uniform int16 streaming_load_uniform(uniform int16 a[])
uniform unsigned int streaming_load_uniform(uniform unsigned int a[])
uniform int streaming_load_uniform(uniform int a[])
uniform unsigned int64 streaming_load_uniform(uniform unsigned int64 a[])
uniform int64 streaming_load_uniform(uniform int64 a[])
uniform float16 streaming_load_uniform(uniform float16 a[])
uniform float streaming_load_uniform(uniform float a[])
uniform double streaming_load_uniform(uniform double a[])
```

### Data Conversions

### Converting Between Array-of-Structures and Structure-of-Arrays Layout

Applications often lay data out in memory in "array of structures" form. Though convenient in C/C++ code, this layout can make `ispc` programs less efficient than they would be if the data was laid out in "structure of arrays" form. (See the section Use "Structure of Arrays" Layout When Possible in the performance guide for extended discussion of this topic.)

The standard library does provide a few functions that efficiently convert between these two formats, for cases where it's not possible to change the application to use "structure of arrays layout". Consider an array of 3D (x,y,z) position data laid out in a C array like:

```
// C++ code
float pos[] = { x0, y0, z0, x1, y1, z1, x2, ... };
```

In an `ispc` program, we might want to load a set of (x,y,z) values and do a computation based on them. The natural expression of this:

```
extern uniform float pos[];
uniform int base = ...;
float x = pos[base + 3 * programIndex];     // x = { x0 x1 x2 ... }
float y = pos[base + 1 + 3 * programIndex]; // y = { y0 y1 y2 ... }
float z = pos[base + 2 + 3 * programIndex]; // z = { z0 z1 z2 ... }
```

leads to irregular memory accesses and reduced performance. Alternatively, the `aos_to_soa3()` standard library function could be used:

```
extern uniform float pos[];
uniform int base = ...;
float x, y, z;
aos_to_soa3(&pos[base], &x, &y, &z);
```

This routine loads three times the gang size values from the given array starting at the given offset, returning three `varying` results. There are `int32`, `int64`, `float` and `double` variants of this function:

```
void aos_to_soa3(uniform float a[], varying float * uniform v0,
                 varying float * uniform v1, varying float * uniform v2)
void aos_to_soa3(uniform int32 a[], varying int32 * uniform v0,
                 varying int32 * uniform v1, varying int32 * uniform v2)
void aos_to_soa3(uniform double a[], varying double * uniform v0,
                 varying double * uniform v1, varying double * uniform v2)
void aos_to_soa3(uniform int64 a[], varying int64 * uniform v0,
                 varying int64 * uniform v1, varying int64 * uniform v2)
```

After computation is done, corresponding functions convert back from the SoA values in `ispc varying` variables and write the values back to the given array, starting at the given offset.

```
extern uniform float pos[];
uniform int base = ...;
float x, y, z;
aos_to_soa3(&pos[base], &x, &y, &z);
// do computation with x, y, z
soa_to_aos3(x, y, z, &pos[base]);
```

```
void soa_to_aos3(float v0, float v1, float v2, uniform float a[])
void soa_to_aos3(int32 v0, int32 v1, int32 v2, uniform int32 a[])
void soa_to_aos3(double v0, double v1, double v2, uniform double a[])
void soa_to_aos3(int64 v0, int64 v1, int64 v2, uniform int64 a[])
```

Note that these functions do not take the current program execution mask into account; they unconditionally read and write three times the gang size. Hence, if the iteration count is not an integer multiple of the program count, `aos_to_soa3()` will read past the end of the input data and `soa_to_aos3()` will write past the end of the output data. To avoid memory corruption in this case, one of the following approaches can be taken:

- Ensure that the data buffers have a size that is a multiple of `programCount`, so that the read/write overflow does not cause memory corruption
- For the main loop, mask the iteration count to be a multiple of `programCount` and add a manual "remainder" loop (which will probably use a gather/scatter) for the remaining iterations

There are also variants of these functions that convert 4-wide values and 2-wide values between AoS and SoA layouts. In other words, `aos_to_soa4()` converts AoS data in memory laid out like `r0 g0 b0 a0 r1 g1 b1 a1 ...` to four `varying` variables with values `r0 r1...`, `g0 g1...`, `b0 b1...`, and `a0 a1...`, reading a total of four times the gang size values from the given array, starting at the given offset.

```
void aos_to_soa4(uniform float a[], varying float * uniform v0,
                 varying float * uniform v1, varying float * uniform v2,
                 varying float * uniform v3)
void aos_to_soa4(uniform int32 a[], varying int32 * uniform v0,
                 varying int32 * uniform v1, varying int32 * uniform v2,
                 varying int32 * uniform v3)
void soa_to_aos4(float v0, float v1, float v2, float v3, uniform float a[])
void soa_to_aos4(int32 v0, int32 v1, int32 v2, int32 v3, uniform int32 a[])
```

The following 2-wide variant of these functions are also supported.

```
void aos_to_soa2(uniform float a[], varying float * uniform v0,
                 varying float * uniform v1)
void aos_to_soa2(uniform int32 a[], varying int32 * uniform v0,
                 varying int32 * uniform v1)
void soa_to_aos2(float v0, float v1, uniform float a[])
void soa_to_aos2(int32 v0, int32 v1, uniform int32 a[])
```

### Conversions To and From Half-Precision Floats

There are functions to convert to and from the IEEE 16-bit floating-point format. Note that there is a `float16` data-type in `ispc`, which has full language and standard library support, but only on the targets with hardware support for this type. The following functions facilitate converting to and from half-format data in memory and are primarily targeted for the use on the targets without native support for `float16` in the hardware.

To use them, half-format data should be loaded into an `int16` and the `half_to_float()` function used to convert it to a 32-bit floating point value. To store a value to memory in half format, the `float_to_half()` function returns the 16 bits that are the closest match to the given `float`, in half format.

```
float half_to_float(unsigned int16 h)
uniform float half_to_float(uniform unsigned int16 h)
int16 float_to_half(float f)
uniform int16 float_to_half(uniform float f)
```

There are also faster versions of these functions that don't worry about handling floating point infinity, "not a number" and denormalized numbers correctly. These are faster than the above functions, but are less precise.

```
float half_to_float_fast(unsigned int16 h)
uniform float half_to_float_fast(uniform unsigned int16 h)
int16 float_to_half_fast(float f)
uniform int16 float_to_half_fast(uniform float f)
```

### Converting to sRGB8

The sRGB color space is used in many applications in graphics and imaging; see the Wikipedia page on sRGB for more information. The `ispc` standard library provides two functions for converting floating-point color values to 8-bit values in the sRGB space.

```
int float_to_srgb8(float v)
uniform int float_to_srgb8(uniform float v)
```

### Systems Programming Support

### Atomic Operations and Memory Fences

The standard set of atomic memory operations are provided by the standard library, including variants to handle both uniform and varying types as well as "local" and "global" atomics.

Local atomics provide atomic behavior across the program instances in a gang, but not across multiple gangs or memory operations in different hardware threads. To see why they are needed, consider a histogram calculation where each program instance in the gang computes which bucket a value lies in and then increments a corresponding counter. If the code is written like this:

```
uniform int count[N_BUCKETS] = ...;
float value = ...;
int bucket = clamp(value / N_BUCKETS, 0, N_BUCKETS);
++count[bucket];  // ERROR: undefined behavior if collisions
```

then the program's behavior is undefined: whenever multiple program instances have values that map to the same value of `bucket`, then the effect of the increment is undefined. (See the discussion in the Data Races Within a Gang section; in the case here, there isn't a sequence point between one program instance updating `count[bucket]` and the other program instance reading its value.)

The `atomic_add_local()` function can be used in this case; as a local atomic it is atomic across the gang of program instances, such that the expected result is computed.

```
...
int bucket = clamp(value / N_BUCKETS, 0, N_BUCKETS);
atomic_add_local(&count[bucket], 1);
```

It uses this variant of the 32-bit integer atomic add routine:

```
int32 atomic_add_local(uniform int32 * uniform ptr, int32 delta)
```

The semantics of this routine are typical for an atomic add function: the pointer here points to a single location in memory (the same one for all program instances), and for each executing program instance, the value stored in the location that `ptr` points to has that program instance's value "delta" added to it atomically, and the old value at that location is returned from the function.

One thing to note is that the type of the value being added to is a `uniform` integer, while the increment amount and the return value are `varying`. In other words, the semantics of this call are that each running program instance individually issues the atomic operation with its own `delta` value and gets the previous value back in return. The atomics for the running program instances may be issued in arbitrary order; it's not guaranteed that they will be issued in `programIndex` order, for example.

Global atomics are more powerful than local atomics; they are atomic across both the program instances in the gang as well as atomic across different gangs and different hardware threads. For example, for the global variant of the atomic used above,

```
int32 atomic_add_global(uniform int32 * uniform ptr, int32 delta)
```

if multiple processors simultaneously issue atomic adds to the same memory location, the adds will be serialized by the hardware so that the correct result is computed in the end.

Here are the declarations of the `int32` variants of these functions. There are also `int64` equivalents as well as variants that take `unsigned int32` and `int64` values.

```
int32 atomic_add_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_subtract_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_min_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_max_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_and_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_or_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_xor_{local,global}(uniform int32 * uniform ptr, int32 value)
int32 atomic_swap_{local,global}(uniform int32 * uniform ptr, int32 value)
```

Support for `float` and `double` types is also available. For local atomics, all but the logical operations are available. (There are corresponding `double` variants of these, not listed here.)

```
float atomic_add_local(uniform float * uniform ptr, float value)
float atomic_subtract_local(uniform float * uniform ptr, float value)
float atomic_min_local(uniform float * uniform ptr, float value)
float atomic_max_local(uniform float * uniform ptr, float value)
float atomic_swap_local(uniform float * uniform ptr, float value)
```

For global atomics, only atomic swap is available for these types:

```
float atomic_swap_global(uniform float * uniform ptr, float value)
double atomic_swap_global(uniform double * uniform ptr, double value)
```

Finally, "swap" (but none of these other atomics) is available for pointer types:

```
void *atomic_swap_{local,global}(void * * uniform ptr, void * value)
```

There are also variants of the atomic that take `uniform` values for the operand and return a `uniform` result. These correspond to a single atomic operation being performed for the entire gang of program instances, rather than one per program instance.

```
uniform int32 atomic_add_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_subtract_{local,global}(uniform int32 * uniform ptr,
                                             uniform int32 value)
uniform int32 atomic_min_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_max_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_and_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_or_{local,global}(uniform int32 * uniform ptr,
                                       uniform int32 value)
uniform int32 atomic_xor_{local,global}(uniform int32 * uniform ptr,
                                        uniform int32 value)
uniform int32 atomic_swap_{local,global}(uniform int32 * uniform ptr,
                                         uniform int32 newval)
```

And similarly for pointers:

```
uniform void *atomic_swap_{local,global}(void * * uniform ptr,
                                         void *newval)
```

Be careful that you use the atomic function that you mean to; consider the following code:

```
extern uniform int32 counter;
int32 myCounter = atomic_add_global(&counter, 1);
```

One might write code like this with the intent that each running program instance increments the counter by one and gets the old value of the counter (for example, to store results into unique locations in an array). However, the above code calls the second variant of `atomic_add_global()`, which takes a `uniform int` value to add to the counter and only performs one atomic operation. The counter will be increased by just one, and all program instances will receive the same value back (thanks to the `uniform int32` return value being silently converted to a `varying int32`.) Writing the code this way, for example, will cause the desired atomic add function to be called.

```
extern uniform int32 counter;
int32 myCounter = atomic_add_global(&counter, (varying int32)1);
```

There is a third variant of each of these atomic functions that takes a `varying` pointer; this allows each program instance to issue an atomic operation to a possibly-different location in memory. (Of course, the proper result is still returned if some or all of them happen to point to the same location in memory!)

```
int32 atomic_add_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_subtract_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_min_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_max_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_and_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_or_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_xor_{local,global}(uniform int32 * varying ptr, int32 value)
int32 atomic_swap_{local,global}(uniform int32 * varying ptr, int32 value)
```

And:

```
void *atomic_swap_{local,global}(void * * ptr, void *value)
```

There are also atomic "compare and exchange" functions. Compare and exchange atomically compares the value in "val" to "compare"--if they match, it assigns "newval" to "val". In either case, the old value of "val" is returned. (As with the other atomic operations, there are also `unsigned` and 64-bit variants of this function. Furthermore, there are `float`, `double`, and `void *` variants as well.)

```
int32 atomic_compare_exchange_{local,global}(uniform int32 * uniform ptr,
                                             int32 compare, int32 newval)
uniform int32 atomic_compare_exchange_{local,global}(uniform int32 * uniform ptr,
                                uniform int32 compare, uniform int32 newval)
```

`ispc` also has a standard library routine that inserts a memory barrier into the code; it ensures that all memory reads and writes prior to be barrier complete before any reads or writes after the barrier are issued. See the Linux kernel documentation on memory barriers for an excellent writeup on the need for and the use of memory barriers in multi-threaded code.

```
void memory_barrier();
```

Note that this barrier is *not* needed for coordinating reads and writes among the program instances in a gang; it's only needed for coordinating between multiple hardware threads running on different cores. See the section Data Races Within a Gang for the guarantees provided about memory read/write ordering across a gang.

### Prefetches

The standard library has a variety of functions to prefetch data into the processor's cache. While modern CPUs have automatic prefetchers that do a reasonable job of prefetching data to the cache before its needed, high performance applications may find it helpful to prefetch data before it's needed.

For example, this code shows how to prefetch data to the processor's L1 cache while iterating over the items in an array.

```
uniform int32 array[...];
for (uniform int i = 0; i < count; ++i) {
    // do computation with array[i]
    prefetch_l1(&array[i+32]);
}
```

The standard library has routines to prefetch to the L1, L2, and L3 caches. It also has a variant, `prefetch_nt()`, that indicates that the value being prefetched isn't expected to be used more than once (so should be high priority to be evicted from the cache). Furthermore, it has versions of these functions that take both `uniform` and `varying` pointer types.

```
void prefetch_{l1,l2,l3,nt}(void * uniform ptr)
void prefetch_{l1,l2,l3,nt}(void * varying ptr)
```

The standard library also has routines to prefetch to the L1, L2, and L3 caches in anticipation of a write:

```
void prefetchw_{l1,l2,l3}(void * uniform ptr)
void prefetchw_{l1,l2,l3}(void * varying ptr)
```

### System Information

The value of a high-precision hardware clock counter is returned by the `clock()` routine; its value increments by one each processor cycle. Thus, taking the difference between the values returned by `clock()` at different points in program execution gives the number of cycles between those points in the program.

```
uniform int64 clock()
```

Note that `clock()` flushes the processor pipeline. It has an overhead of a hundred or so cycles, so for very fine-grained measurements, it may be worthwhile to measure the cost of calling `clock()` and subtracting that value from reported results.

A routine is also available to find the number of CPU cores available in the system:

```
uniform int num_cores()
```

This value can be useful for adapting the granularity of parallel task decomposition depending on the number of processors in the system.

## Interoperability with the Application

One of `ispc`'s key goals is to make it easy to interoperate between the C/C++ application code and parallel code written in `ispc`. This section describes the details of how this works and describes a number of the pitfalls.

### Interoperability Overview

As described in Compiling and Running a Simple ISPC Program it's relatively straightforward to call `ispc` code from C/C++. First, any `ispc` functions to be called should be defined with the `export` keyword:

```
export void foo(uniform float a[]) {
    ...
```

```
}
```

This function corresponds to the following C-callable function:

```
void foo(float a[]);
```

(Recall from the "uniform" and "varying" Qualifiers section that `uniform` types correspond to a single instances of the corresponding type in C/C++.)

In addition to variables passed from the application to `ispc` in the function call, you can also share global variables between the application and `ispc`. To do so, just declare the global variable as usual (in either `ispc` or application code), and add an `extern` declaration on the other side.

For example, given this `ispc` code:

```
// ispc code
uniform float foo;
extern uniform float bar[10];
```

And this C++ code:

```
// C++ code
extern "C" {
  extern float foo;
  float bar[10];
}
```

Both the `foo` and `bar` global variables can be accessed on each side. Note that the `extern "C"` declaration is necessary from C++, since `ispc` uses C linkage for functions and globals.

`ispc` code can also call back to C/C++. On the `ispc` side, any application functions to be called must be declared with the `extern "C"` qualifier.

```
extern "C" void foo(uniform float f, uniform float g);
```

Unlike in C++, `extern "C"` doesn't take braces to delineate multiple functions to be declared; thus, multiple C functions to be called from `ispc` must be declared as follows:

```
extern "C" void foo(uniform float f, uniform float g);
extern "C" uniform int bar(uniform int a);
```

It is illegal to overload functions declared with `extern "C"` linkage; `ispc` issues an error in this case.

Functions declared with `extern "C"` linkage can be made to follow `__vectorcall` calling convention on Windows by using `__vectorcall` qualifier.

```
extern "C" __vectorcall void foo(uniform float f, uniform float g);
```

`__vectorcall` can only be used for `extern "C"` function declarations and on Windows OS.

**Only a single function call is made back to C++ for the entire gang of running program instances**. Furthermore, function calls back to C/C++ are not made if none of the program instances want to make the call. For example, given code like:

```
uniform float foo = ...;
float x = ...;
if (x != 0)
    foo = appFunc(foo);
```

`appFunc()` will only be called if one or more of the running program instances evaluates `true` for `x != 0`. If the application code would like to determine which of the running program instances want to make the call, a mask representing the active SIMD lanes can be passed to the function.

```
extern "C" float appFunc(uniform float x,
                         uniform int activeLanes);
```

If the function is then called as:

```
...
x = appFunc(x, lanemask());
```

The `activeLanes` parameter will have the value one in the 0th bit if the first program instance is running at this point in the code, one in the first bit for the second instance, and so forth. (The `lanemask()` function is documented in Cross-Program Instance Operations.) Application code can thus be written as:

```
float appFunc(float x, int activeLanes) {
    for (int i = 0; i < programCount; ++i)
        if ((activeLanes & (1 << i)) != 0) {
            // do computation for i'th SIMD lane
        }
}
```

In some cases, it can be desirable to generate a single call for each executing program instance, rather than one call for a gang. For example, the code below shows how one might call an existing math library routine that takes a scalar parameter.

```
extern "C" uniform double erf(uniform double);
double v = ...;
double result;
foreach_active (instance) {
    uniform double r = erf(extract(v, instance));
    result = insert(result, instance, r);
}
```

This code calls `erf()` once for each active program instance, passing it the program instance's value of `v` and storing the result in the instance's `result` value.

### Data Layout

In general, `ispc` tries to ensure that `struct` types and other complex datatypes are laid out in the same way in memory as they are in C/C++. Matching structure layout is important for easy interoperability between C/C++ code and `ispc` code.

The main complexity in sharing data between `ispc` and C/C++ often comes from reconciling data structures between `ispc` code and application code; it can be useful to declare the shared structures in `ispc` code and then examine the generated header file (which will have the C/C++ equivalents of them.) For example, given a structure in `ispc`:

```
// ispc code
struct Node {
   int count;
   float pos[3];
};
```

If a `uniform Node` structure is used in the parameters to an `export` ed function, then the header file generated by the `ispc` compiler will have a declaration like:

```
// C/C++ code
struct Node {
   int count;
   float pos[3];
};
```

Because `varying` types have size that depends on the size of the gang of program instances, `ispc` has restrictrictions on using varying types in parameters to functions with the `export` qualifier. `ispc` prohibits parameters to exported functions to have varying type unless the parameter is of pointer type. (That is, `varying float` isn't allowed, but `varying float * uniform` (uniform pointer to varying float) is permitted.) Care must be taken by the programmer to ensure that the data being accessed through any pointers to varying data has the correct organization.

Similarly, `struct` types shared with the application can also have embedded pointers.

```
// C code
struct Foo {
    float *foo, *bar;
};
```

On the `ispc` side, the corresponding `struct` declaration is:

```
// ispc
struct Foo {
    float * uniform foo, * uniform bar;
};
```

If a pointer to a varying `struct` type appears in an exported function, the generated header file will have a definition like (for 8-wide SIMD):

```
// C/C++ code
struct Node {
  int count[8];
```

```
  float pos[3][8];
};
```

In the case of multiple target compilation, `ispc` will generate multiple header files and a "general" header file with definitions for multiple sizes. Any pointers to varyings in exported functions will be rewritten as `void *`. At runtime, the `ispc` dispatch mechanism will cast these pointers to the appropriate types. Programmers can provide C/C++ code with a mechanism to determine the gang width used at runtime by `ispc` by creating an exported function that simply returns the value of `programCount`. An example of such a function is provided in the file `examples/util/util.isph` included in the `ispc` distribution.

There is one subtlety related to data layout to be aware of: `ispc` stores `uniform` short-vector types in memory with their first element at the machine's natural vector alignment (i.e. 16 bytes for a target that is using Intel® SSE, and so forth.) This implies that these types will have different layout on different compilation targets. As such, applications should in general avoid accessing `uniform` short vector types from C/C++ application code if possible.

### Data Alignment and Aliasing

There are two important constraints that must be adhered to when passing pointers from the application to `ispc` programs.

The first is that it is required that it be valid to read memory at the first element of any array that is passed to `ispc`. In practice, this should just happen naturally, but it does mean that it is illegal to pass a `NULL` pointer as a parameter to a `ispc` function called from the application.

The second constraint is that pointers and references in `ispc` programs must not alias. The `ispc` compiler assumes that different pointers can't end up pointing to the same memory location, either due to having the same initial value, or through array indexing in the program as it executed.

This aliasing constraint also applies to `reference` parameters to functions. Given a function like:

```
void func(int &a, int &b) {
    a = 0;
    if (b == 0) { ... }
}
```

Then the same variable must not be passed to `func()`. This is another case of aliasing, and if the caller calls the function as `func(x, x)`, it's not guaranteed that the `if` test will evaluate to true, due to the compiler's requirement of no aliasing.

(In the future, `ispc` will have a mechanism to indicate that pointers may alias.)

### Restructuring Existing Programs to Use ISPC

`ispc` is designed to enable you to incorporate SPMD parallelism into existing code with minimal modification; features like the ability to share memory and data structures between C/C++ and `ispc` code and the ability to directly call back and forth between `ispc` and C/C++ are motivated by this. These features also make it easy to incrementally transform a program to use `ispc`; the most computationally-intensive localized parts of the computation can be transformed into `ispc` code while the remainder of the system is left as is.

For a given section of code to be transitioned to run in `ispc`, the next question is how to parallelize the computation. Generally, there will be obvious loops inside which a large amount of computation is done ("for each ray", "for each pixel", etc.) Mapping these to the SPMD computational style is often effective.

Carefully choose how to do the exact mapping of computation to SPMD program instances. This choice can impact the mix of gather/scatter memory access versus coherent memory access, for example. (See more on this topic in the ispc Performance Tuning Guide.) This decision can also impact the coherence of control flow across the running SPMD program instances, which can also have a significant effect on performance; in general, creating groups of work that will tend to do similar computation across the SPMD program instances improves performance.

## Notices & Disclaimers