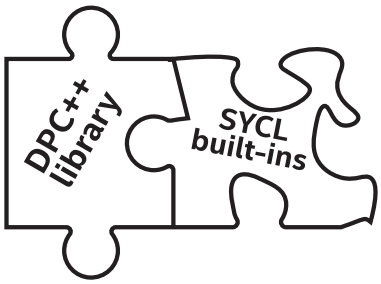
CHAPTER 18

Libraries



We have spent the entire book promoting the art of *writing our own code*. Now we finally acknowledge that some great programmers have already written code that we can just use. Libraries are the best way to get our work done. This is not a case of being lazy—it is a case of having better things to do than reinvent the work of others. This is a puzzle piece worth having.

The open source DPC++ project includes some libraries. These libraries can help us continue to use libstdc++, libc++, and MSVC library functions even within our kernel code. The libraries are included as part of DPC++ and the oneAPI products from Intel. These libraries are not tied to the DPC++ compiler so they can be used with any SYCL compiler.

The DPC++ library provides an alternative for programmers who create heterogeneous applications and solutions. Its APIs are based on familiar standards—C++ STL, Parallel STL (PSTL), and SYCL—to provide high-productivity APIs to programmers. This can minimize programming effort across CPUs, GPUs, and FPGAs while leading to high-performance parallel applications that are portable.

J. Reinders et al., *Data Parallel C++*, https://doi.org/10.1007/978-1-4842-5574-2_18

The SYCL standard defines a rich set of built-in functions that provide functionality, for host and device code, worth considering as well. DPC++ and many SYCL implementations implement key math built-ins with math libraries.

The libraries and built-ins discussed within this chapter are compiler agnostic. In other words, they are equally applicable to DPC++ compilers or SYCL compilers. The `fpga_device_policy` class is a DPC++ feature for FPGA support.

Since there is overlap in naming and functionality, this chapter will start with a brief introduction to the SYCL built-in functions.

Built-In Functions

DPC++ provides a rich set of SYCL built-in functions with respect to various data types. These built-in functions are available in the `sycl` namespace on host and device with low-, medium-, and high-precision support for the target devices based on compiler options, for example, the `-mfma`, `-ffast-math`, and `-ffp-contract=fast` provided by the DPC++ compiler. These built-in functions on host and device can be classified as in the following:

- Floating-point math functions: asin, `acos`, `log, sqrt`, `floor`, etc. listed in Figure 18-2.
- Integer functions: `abs`, `max, min`, etc. listed in Figure 18-3.
- Common functions: `clamp`, `smoothstep`, etc. listed in Figure 18-4.
- Geometric functions: `cross`, `dot`, `distance`, etc. listed in Figure 18-5.
- Relational functions: `isequal`, `isless`, `isfinite`, etc. listed in Figure 18-6.

If a function is provided by the C++ std library, as listed in Figure 18-8, as well as a SYCL built-in function, then DPC++ programmers are allowed to use either. Figure 18-1 demonstrates the C++ `std::log` function and SYCL built-in `sycl::log` function for host and device, and both functions produce the same numeric results. In the example, the built-in relational function `sycl::isequal` is used to compare the results of `std:log` and `sycl:log`.

```
constexpr int size = 9;
std::array<double, size> A;
std::array<double, size> B;

bool pass = true;

for (int i = 0; i < size; ++i) { A[i] = i; B[i] = i; }

queue Q;
range sz{size};

buffer<double> bufA(A);
buffer<double> bufB(B);
buffer<bool>   bufP(&pass, 1);

Q.submit([&](handler &h) {
  accessor accA{ bufA, h};
  accessor accB{ bufB, h};
  accessor accP{ bufP, h};

  h.parallel_for(size, [=](id<1> idx) {
    accA[idx] = std::log(accA[idx]);
    accB[idx] = sycl::log(accB[idx]);
    if (!sycl::isequal( accA[idx], accB[idx]) ) {
      accP[0] = false;
    }
  });
});
```

Figure 18-1. *Using* `std::log` *and* `sycl::log`

In addition to the data types supported in SYCL, the DPC++ device library provides support for `std:complex` as a data type and the corresponding math functions defined in the C++ std library.

Use the `sycl::` Prefix with Built-In Functions

The SYCL built-in functions should be invoked with an explicit `sycl::` prepended to the name. With the current SYCL specification, calling just `sqrt()` is not guaranteed to invoke the SYCL built-in on all implementations even if "`using namespace sycl;`" has been used.

SYCL built-in functions should always be invoked with an explicit `sycl::` in front of the built-in name. Failure to follow this advice may result in strange and non-portable results.

If a built-in function name conflicts with a non-templated function in our application, in many implementations (including DPC++), our function will prevail, thanks to C++ overload resolution rules that prefer a non-templated function over a templated one. However, if our code has a function name that is the same as a built-in name, the most portable thing to do is either avoid `using namespace sycl;` or make sure no actual conflict happens. Otherwise, some SYCL compilers will refuse to compile the code due to an unresolvable conflict within their implementation. Such a conflict will not be silent. Therefore, if our code compiles today, we can safely ignore the possibility of future problems.

Tf **acos**(Tf x)			Arc cosine
Tf **acosh**(Tf x)			Inverse hyperbolic cosine
Tf **acospi**(Tf x)			acos(x) / π
Tf **asin**(Tf x)			Arc sine
Tf **asinh**(Tf x)			Inverse hyperbolic sine
Tf **asinpi**(Tf x)			asin(x) / π
Tf **atan**(Tf y_over_x)			Arc tangent
Tf **atan2**(Tf y,Tf x)			Arc tangent of y / x
Tf **atanh**(Tf x)			Hyperbolic arc tangent
Tf **atanpi**(Tf x)			atan(x) / π
Tf **atan2pi**(Tf x,Tf y)			atan2(y,x) / π
Tf **cbrt**(Tf x)			Cube root
Tf **ceil**(Tf x)			Round to integer toward + infinity
Tf **copysign**(Tf x,Tf y)			x with sign changed to sign of y
Tf **cos**(Tf x)		H	Cosine
Tf **cosh**(Tf x)	N		Hyperbolic cosine
Tf **cospi**(Tf x)			cos(π x)
Tf **divide**(Tf x,Tf y)	*N		x / y
Tf **erfc**(Tf x)			Complementary error function
Tf **erf**(Tf x)			Calculates error function
Tf **exp**(Tf x)	N	H	Exponential base e
Tf **exp2**(Tf x)	N	H	Exponential base 2
Tf **exp10**(Tf x)	N	H	Exponential base 10
Tf **expm1**(Tf x)	N	H	ex -1.0
Tf **fabs**(Tf x)			Absolute value
Tf **fdim**(Tf x,Tf y)			Positive difference between x and y
Tf **floor**(Tf x)			Round to integer toward infinity
Tf **fma**(Tf a,Tf b,Tf c)			Multiply and add,then round
Tf **fmax**(Tf x,Tf y)			Return y if x < y,otherwise it returns x
Tf **fmin**(Tf x,Tf y)			
Tf **fmod**(Tf x,Tf y)			Modulus. Returns x – y * trunc(x/y)
floatn **fract**(floatn x, intn *iptr) float **fract**(float x, int *iptr)			Fractional value in x
doublen **frexp**(doublen x, intn *exp) double **frexp**(double x, int *exp)			Extract mantissa and exponent
Tf **hypot**(Tf x,Tf y)			Square root of x2 + y2
int **logb**(float x) intn **ilogb**(Tf x) int **logb**(double x) intn logb(doublen x)			Return exponent as an integer value

Tf **ldexp**(Tf x,Ti k) floatn **ldexp**(floatn x, int k) doublen **ldexp**(doublen x , int k)			x * 2^n
Tf **lgamma**(Tf x) Tf **lgamma_r**(Tf x,Ti*signp)			Log gamma function.
Tf **log**(Tf)	N	H	Natural logarithm
Tf **log2**(Tf)	N	H	Base 2 logarithm
Tf **log10**(Tf)	N	H	Base 10 logarithm
Tf **log1p**(Tf x)			ln(1.0 + x)
Tf **logb**(Tf x)			Exponent of x
Tf **mad**(Tf a,Tf b,Tf c)			Approximates a * b + c
Tf **maxmag**(Tf x,Tf y)			Maximum magnitude of x and y
Tf **minmag**(Tf x,Tf y)			Minimum magnitude of x and y
Tf **modf**(Tf x,Tf *iptr)			Decompose floating-point number
floatn **nan**(uintn nancode) float **nan**(unsigned int nancode) doublen **nan**(ulonglongn nancode) doublen **nan**(longlong int nancode)			Quiet NaN(Return is scalar when nancode is scalar)
Tf **nextafter**(Tf x,Tf y)			Next representable floating-point value after x in the direction of y
Tf **pow**(Tf x,Tf y)			Compute x to the power of y
Tf **pown**(Tf x,Ti y)			Compute x y,where y is an integer
Tf **powr**(Tf x,Tf y)	N		Compute x y,where x is >= 0
Tf **recip**(Tf x)	*N	H	1 / x
Tf **remainder**(Tf x,Tf y)			Floating point remainder
Tf **remquo**(Tf x,Tf y,Ti *q)			Remainder and quotient
Tf **rint**(Tf)			Round to nearest even integer
Tf **rootn**(Tf x,Ti y)			Compute x to the power of 1/y
Tf **round**(Tf x)			Integral value nearest to x rounding
Tf **rsqrt**(Tf)	N	H	Inverse square root
Tf **sin**(Tf)	N	H	Sine
Tf **sincos**(Tf x,Tf *cosval)			Sine and cosine of x
Tf **sinh**(Tf x)			Hyperbolic sine
Tf **sinpi**(Tf x)			sin(π x)
Tf **sqrt**(Tf x)	N	H	Square root
Tf **tan**(Tf x)	N	H	Tangent
Tf **tanh**(Tf x)			Hyperbolic tangent
Tf **tanpi**(Tf x)			tan(π x)
Tf **tgamma**(Tf x)			Gamma function
Tf trunc(Tf x)			Round to integer toward zero

For float*n*, double*n*, and int*n*: *n* is 2, 3, 4, 8, or 16.
Tf (genfloat in the spec) is type float, float*n*, double, or double*n*.
sTf (sgenfloat in the spec) is type float or double.
Ti (genint in the spec) is type int or int*n*.
N indicates that native variants are available.
**N* indicates availability only in native forms.
H indicates availability in half-precision for devices with fp16 extension available.

Figure 18-2. *Built-in math functions*

Tui **abs**(Ti x)	| x |
Tui **abs_**diff(Ti x,Ti y)	| x – y | without modulo overflow
Ti **add_sat** (Ti x,Ti y)	x + y and saturates the result
Ti **hadd**(Ti x,Ti y)	(x + y) >> 1 without mod. overflow
Ti **rhadd**(Ti x,Ti y)	(x + y + 1) >> 1
Ti **clamp**(Ti x,Ti min,Ti max)	min(max(x,min),max)
Ti **clamp**(Ti x,Tsi min,Tsi max)	min(max(x,min),max)
Ti **clz**(Ti x)	number of leading zero-bits in x; special case for x == 0: returns the size in bits of the type of x, or the component type of x if x is a vector type.
Ti **ctz**(Ti x)	Same as clz() but for *trailing* zero-bits
Ti **mad_hi**(Ti a,Ti b,Ti c)	mul_hi(a,b) + c
Ti **mad_sat** (Ti a,Ti b,Ti c)	a * b + c and saturates the result
Ti **max**(Ti x,Ti y)	y if x < y,otherwise it returns x
Ti **max**(Ti x,Tsi y)	y if x < y,otherwise it returns x
Ti **min**(Ti x,Ti y)	y if y < x,otherwise it returns x
Ti **min**(Ti x,Tsi y)	y if y < x,otherwise it returns x
Ti **mul_hi**(Ti x,Ti y)	high half of the product of x and y
Ti **popcount** (Ti x)	Number of non-zero bits in x
Ti **rotate**(Ti v,Ti i)	result[indx] = v[indx] << i[indx]
Ti **sub_sat**(Ti x,Ti y)	x - y and saturates the result
T popcount(T x)	Number of non-zero bits in x
shortn **upsample**(charn hi,ucharn lo)	result[i]=((short)hi[i]<< 8)|lo[i]
ushortn **upsample**(ucharn hi,ucharn lo)	result[i]=((ushort)hi[i]<< 8)|lo[i]
intn **upsample**(shortn hi,ushortn lo)	result[i]=((int)hi[i]<< 16)|lo[i]
uintn **upsample**(ushortn hi,ushortn lo)	result[i]=((uint)hi[i]<< 16)|lo[i]
longlongn **upsample**(intn hi,uintn lo)	result[i]=((long)hi[i]<< 32)|lo[i]
ulonglongn **upsample**(uintn hi,uintn lo)	result[i]=((ulong)hi[i]<< 32)|lo[i]
intn **mad24**(intn x,intn y,intn z)	Multiply 24-bit integer values x,y,add 32-bit int. result to 32-bit integer z
uintn **mad24**(uintn x,uintn y,uintn z)	Multiply 24-bit integer values x,y,add 32-bit int. result to 32-bit integer z
intn **mul24**(intn x,intn y)	Multiply 24-bit integer values x and y
uintn **mul24**(uintn x,uintn y)	Multiply 24-bit integer values x and y

T (gentype in the spec) is all integer and float types.
Ti (geninteger in spec) is all signed and unsigned integer types.
Tsi (sgeninteger in the spec) is all scalar integer types.
Tui (ugeninteger in the spec) is all unsigned integer types.

Figure 18-3. *Built-in integer functions*

Tf **clamp**(Tf x,Tf minval,Tf maxval) floatn **clamp** (floatn x,float minval,float maxval) doublen **clamp** (doublen x,double minval,doublen maxval)	Clamp x to range given by minval,maxval
Tf **degrees**(Tf radians)	radians to degrees
Tf **max**(Tf x,Tf y) Tff **max**(Tff x,float y) Tfd **max**(Tfd x,double y)	Max of x and y
Tf **min**(Tf x,Tf y) Tff **min**(Tff x,float y) Tfd **min**(Tfd x,double y)	Min of x and y
Tf **mix**(Tf x,Tf y,Tf a) Tff **mix**(Tff x,Tff y,float a) Tfd **mix**(Tfd x,Tfd y,double a)	Linear blend of x and y
Tf **radians**(Tf degrees)	degrees to radians
Tf **step**(Tf edge,Tf x) Tff **step**(float edge,Tff x) Tfd **step**(double edge,Tfd x)	0.0 if x < edge,else 1.0
Tf **smoothstep** (Tf edge0,Tf edge1,Tf x) Tff **smoothstep**(float edge0,float edge1,Tff x); Tfd **smoothstep**(double edge0,double edge1,Tfd x)	Step and interpolate
Tf **sign**(Tf x)	Sign of x

For float*n* and double*n*: *n* is 2, 3, 4, 8, or 16.
Tf (genfloat in the spec) is float, float*n*, double, or double*n* types.
Tff (genfloatf in the spec) is float or float*n* types.
Tfd (genfloatd in the spec) is double or double*n* types.

Figure 18-4. *Built-in common functions*

T*34* cross (T*34* p0,T*34* p1)	Cross product
T distance(T*n* p0,T*n* p1)	Vector distance
T dot(T*n* p0,T*n* p1)	Dot product
T length(T*n* p)	Vector length
T*n* normalize(T*n* p)	Normal vector length 1
float fast_distance(float*n* p0,float*n* p1)	Vector distance
float fast_length(float*n* p)	Vector length
float*n* fast_normalize(float*n* p)	Normal vector length 1

For T*n*: *n* is 2, 3, 4, 8, or 16.
T*34* means either T3 or T4.
T is type float, float*n*, double, or double*n*, consistently applied per function.

Figure 18-5. *Built-in geometric functions*

Functions **F**(…) can be: isequal isnotequal isgreater isgreaterequal isless islessequal islessgreater isordered isunordered	int ***F***(float x,float y) int*n* ***F***(floatn x,floatn y) long long ***F***(double x,double y) long long*n* ***F***(doublen x,doublen y)
Functions **F**(…) can be: isfinite isinf isnan isnormal signbit	int ***F***(float) int*n* ***F***(floatn) long long ***F***(double) long long*n* ***F***(doublen)
int **any**(Ti x)	1 if MSB in component of x is set; else 0
int **all**(Ti x)	1 if MSB in all components of x are set else 0
T **bitselect** (T a,T b,T c)	Each bit of result is corresponding bit of a if corresponding bit of c is 0
T **select** (T a,T b,Ti c)	For each component of a vector type,result[i] = if MSB of c[i] is set ? b[i] : a[i] For scalar type,result = c ? b : a

For int*n* and long*n*: n is 2, 3, 4, 8, or 16.
T (gentype in the spec) is all signed , unsigned, float, double, scalar and vector types.
Ti (geninteger in spec) is signed and unsigned integer types.
Tui (ugeninteger in the spec) is all unsigned integer types.

Figure 18-6. *Built-in relational functions*

DPC++ Library

The DPC++ library consists of the following components:

- A set of tested C++ standard APIs—we simply need to include the corresponding C++ standard header files and use the `std` namespace.
- Parallel STL that includes corresponding header files. We simply use `#include <dpstd/...>` to include them. The DPC++ library uses namespace `dpstd` for the extended API classes and functions.

Standard C++ APIs in DPC++

The DPC++ library contains a set of tested standard C++ APIs. The basic functionality for a number of C++ standard APIs has been developed so that these APIs can be employed in device kernels similar to how they are employed in code for a typical C++ host application. Figure 18-7 shows an example of how to use `std::swap` in device code.

```
class KernelSwap;
std::array <int,2> arr{8,9};
buffer<int> buf{arr};

{
  host_accessor host_A(buf);
  std::cout << "Before: " << host_A[0] << ", " << host_A[1] << "\n";
}  // End scope of host_A so that upcoming kernel can operate on buf

queue Q;
Q.submit([&](handler &h) {
  accessor A{buf, h};
  h.single_task([=]() {
  // Call std::swap!
  std::swap(A[0], A[1]);
  });
});

host_accessor host_B(buf);
std::cout << "After:  " << host_B[0] << ", " << host_B[1] << "\n";
```

Figure 18-7. *Using `std::swap` in device code*

We can use the following command to build and run the program (assuming it resides in the `stdswap.cpp` file):

```
dpcpp -std=c++17 stdswap.cpp -o stdswap.exe
./stdswap.exe
```

The printed result is:

```
8, 9
9, 8
```

Figure 18-8 lists C++ standard APIs with "*Y*" to indicate those that have been tested for use in DPC++ kernels for CPU, GPU, *and* FPGA devices, at the time of this writing. A blank indicates incomplete coverage (not all three device types) at the time of publication for this book. A table is also included as part of the online DPC++ language reference guide and will be updated over time—the library support in DPC++ will continue to expand its support.

In the DPC++ library, some C++ `std` functions are implemented based on their corresponding built-in functions on the device to achieve the same level of performance as the SYCL versions of these functions.

C++ standard API	libstdc++	libc++	MSVS
std::acos	Y		
std::acosh	Y		
std::add_const	Y	Y	Y
std::add_cv	Y	Y	Y
std::add_volatile	Y	Y	Y
std::alignment_of	Y	Y	Y
std::array	Y	Y	Y
std::asin	Y		
std::is_fundamental	Y	Y	Y
std::is_literal_type	Y	Y	Y
std::is_member_pointer	Y	Y	Y
std::is_move_assignable	Y	Y	Y
std::is_move_constructible	Y	Y	Y
std::is_object	Y	Y	Y
std::is_pod	Y	Y	Y
std::is_reference	Y	Y	Y
std::is_fundamental	Y	Y	Y
std::is_literal_type	Y	Y	Y
std::is_member_pointer	Y	Y	Y
std::is_move_assignable	Y	Y	Y
std::is_move_constructible	Y	Y	Y
std::is_object	Y	Y	Y
std::is_pod	Y	Y	Y
std::asinh	Y		
std::assert	Y		Y
std::atan	Y		
std::atan2	Y		
std::atanh	Y		
std::binary_negate	Y	Y	Y
std::binary_search	Y	Y	Y
std::bit_and	Y	Y	Y
std::bit_not	Y	Y	Y
std::bit_or	Y	Y	Y
std::bit_xor	Y	Y	Y
std::cbrt	Y		
std::common_type	Y	Y	Y
std::complex	Y		
std::conditional	Y	Y	Y
std::cos	Y		
std::cosh	Y		
std::decay	Y	Y	Y
std::declval	Y	Y	Y
std::divides	Y	Y	Y
std::enable_if	Y	Y	Y
std::equal_range	Y	Y	Y
std::equal_to	Y	Y	Y
std::erf	Y		
std::erfc	Y		

std::exp	Y		
std::exp2	Y		
std::expm1	Y		
std::extent	Y	Y	Y
std::fdim	Y		
std::fmod	Y		
std::forward	Y	Y	Y
std::frexp	Y		
std::greater	Y	Y	Y
std::greater_equal	Y	Y	Y
std::hypot	Y		
std::ilogb	Y		
std::initializer_list	Y	Y	Y
std::integral_constant	Y	Y	Y
std::is_arithmetic	Y	Y	Y
std::is_assignable	Y	Y	Y
std::is_base_of	Y	Y	Y
std::is_base_of_union	Y	Y	Y
std::is_compound	Y	Y	Y
std::is_const	Y	Y	Y
std::is_constructible	Y	Y	Y
std::is_convertible	Y	Y	Y
std::is_copy_assignable	Y	Y	Y
std::is_copy_constructible	Y	Y	Y
std::is_default_constructible	Y	Y	Y
std::is_destructible	Y	Y	Y
std::is_empty	Y	Y	Y
std::is_same	Y	Y	Y
std::is_scalar	Y	Y	Y
std::is_signed	Y	Y	Y
std::is_standard_layout	Y	Y	Y
std::is_trivial	Y	Y	Y
std::is_trivially_assignable	Y	Y	Y
std::is_trivially_constructible	Y	Y	Y
std::is_trivially_copyable	Y	Y	Y
std::is_unsigned	Y	Y	Y
std::is_volatile	Y	Y	Y
std::ldexp	Y		
std::less	Y	Y	Y
std::less_equal	Y	Y	Y
std::lgamma	Y		
std::log	Y		
std::log10	Y		
std::log1p	Y		
std::log2	Y		
std::logb	Y		
std::logical_and	Y	Y	Y
std::logical_not	Y	Y	Y
std::logical_or	Y	Y	Y
std::lower_bound	Y	Y	Y
std::minus	Y	Y	Y
std::modf	Y		
std::modulus	Y	Y	Y
std::move	Y	Y	Y

Figure 18-8. *Library support with CPU/GPU/FPGA coverage (at time of book publication)*

std::move_if_noexcept	Y	Y	Y
std::multiplies	Y	Y	Y
std::negate	Y	Y	Y
std::nextafter	Y		
std::not_equal_to	Y	Y	Y
std::not1/2	Y	Y	Y
std::numeric_limits	Y	Y	Y
std::pair	Y	Y	Y
std::plus	Y	Y	Y
std::pow	Y		
std::rank	Y	Y	Y
std::ratio	Y	Y	Y
std::ref/cref	Y	Y	Y

std::reference_wrapper	Y	Y	Y
std::remainder	Y		
std::remove_all_extents	Y	Y	Y
std::remove_const	Y	Y	Y
std::remove_cv	Y	Y	Y
std::remove_extent	Y	Y	Y
std::remove_volatile	Y	Y	Y
std::remquo	Y		
std::sin	Y		
std::sinh	Y		
std::sqrt	Y		
std::swap	Y	Y	Y

Figure 18.8. (*continued*)

The tested standard C++ APIs are supported in `libstdc++` (GNU) with `gcc` 7.4.0 and `libc++` (LLVM) with `clang` 10.0 and MSVC Standard C++ Library with Microsoft Visual Studio 2017 for the host CPU as well.

On Linux, GNU `libstdc++` is the default C++ standard library for the DPC++ compiler, so no compilation or linking option is required. If we want to use `libc++`, use the compile options `-stdlib=libc++ -nostdinc++` to leverage `libc++` and to not include C++ std headers from the system. The DPC++ compiler has been verified using `libc++` in DPC++ kernels on Linux, but the DPC++ runtime needs to be rebuilt with `libc++` instead of `libstdc++`. Details are in https://intel.github.io/llvm-docs/GetStartedGuide.html#build-dpc-toolchain-with-libc-library. Because of these extra steps, `libc++` is not the recommended C++ standard library for us to use in general.

On FreeBSD, `libc++` is the default standard library, and the `-stdlib=libc++` option is not required. More details are in https://libcxx.llvm.org/docs/UsingLibcxx.html. On Windows, only the MSVC C++ library can be used.

To achieve cross-architecture portability, if a std function is not marked with "Y" in Figure 18-8, we need to keep portability in mind when we write device functions!

DPC++ Parallel STL

Parallel STL is an implementation of the C++ standard library algorithms with support for execution policies, as specified in the ISO/IEC 14882:2017 standard, commonly called C++17. The existing implementation also supports the unsequenced execution policy specified in Parallelism TS version 2 and proposed for the next version of the C++ standard in the C++ working group paper P1001R1.

When using algorithms and execution policies, specify the namespace `std::execution` if there is no vendor-specific implementation of the C++17 standard library or `pstl::execution` otherwise.

For any of the implemented algorithms, we can pass one of the values `seq`, `unseq`, `par`, or `par_unseq` as the first parameter in a call to the algorithm to specify the desired execution policy. The policies have the following meanings:

Execution Policy	Meaning
`seq`	Sequential execution.
`unseq`	Unsequenced SIMD execution. This policy requires that all functions provided are safe to execute in SIMD.
`par`	Parallel execution by multiple threads.
`par_unseq`	Combined effect of `unseq` and `par`.

Parallel STL for DPC++ is extended with support for DPC++ devices using special execution policies. The DPC++ execution policy specifies where and how a Parallel STL algorithm runs. It inherits a standard C++ execution policy, encapsulates a SYCL device or queue, and allows us to set an optional kernel name. DPC++ execution policies can be used with all standard C++ algorithms that support execution policies according to the C++17 standard.

DPC++ Execution Policy

Currently, only the parallel unsequenced policy (`par_unseq`) is supported by the DPC++ library. In order to use the DPC++ execution policy, there are three steps:

1. Add `#include <dpstd/execution>` into our code.
2. Create a policy object by providing a standard policy type, a class type for a unique kernel name as a template argument (optional), and one of the following constructor arguments:
   - A SYCL queue
   - A SYCL device
   - A SYCL device selector
   - An existing policy object with a different kernel name
3. Pass the created policy object to a Parallel STL algorithm.

A `dpstd::execution::default_policy` object is a predefined `device_policy` created with a default kernel name and default queue. This can be used to create custom policy objects or passed directly when invoking an algorithm if the default choices are sufficient.

Figure 18-9 shows examples that assume use of the `using namespace dpstd::execution;` directive when referring to `policy` classes and functions.

```
auto policy_b =
  device_policy<parallel_unsequenced_policy, class PolicyB>
    {sycl::device{sycl::gpu_selector{}}};
std::for_each(policy_b, …);

auto policy_c =
  device_policy<parallel_unsequenced_policy, class PolicyC>
    {sycl::default_selector{}};
std::for_each(policy_c, …);

auto policy_d = make_device_policy<class PolicyD>(default_policy);
std::for_each(policy_d, …);

auto policy_e = make_device_policy<class PolicyE>(sycl::queue{});
std::for_each(policy_e, …);
```

Figure 18-9. *Creating execution policies*

FPGA Execution Policy

The `fpga_device_policy` class is a DPC++ policy tailored to achieve better performance of parallel algorithms on FPGA hardware devices. Use the policy when running the application on FPGA hardware or an FPGA emulation device:

1. Define the `_PSTL_FPGA_DEVICE` macro to run on FPGA devices and additionally `_PSTL_FPGA_EMU` to run on an FPGA emulation device.
2. Add #include <dpstd/execution> to our code.
3. Create a policy object by providing a class type for a unique kernel name and an unroll factor (see Chapter 17) as template arguments (both optional) and one of the following constructor arguments:
   - A SYCL queue constructed for the FPGA selector (the behavior is undefined with any other device type)
   - An existing FPGA policy object with a different kernel name and/or unroll factor
4. Pass the created policy object to a Parallel STL algorithm.

The default constructor of `fpga_device_policy` creates an object with a SYCL queue constructed for `fpga_selector`, or for `fpga_emulator_selector` if `_PSTL_FPGA_EMU` is defined.

`dpstd::execution::fpga_policy` is a predefined object of the `fpga_device_policy` class created with a default kernel name and default unroll factor. Use it to create customized policy objects or pass it directly when invoking an algorithm.

Code in Figure 18-10 assumes `using namespace dpstd::execution;` for policies and `using namespace sycl;` for queues and device selectors.

Specifying an unroll factor for a policy enables loop unrolling in the implementation of algorithms. The default value is 1. To find out how to choose a better value, see Chapter 17.

```
auto fpga_policy_a = fpga_device_policy<class FPGAPolicyA>{};

auto fpga_policy_b = make_fpga_policy(queue{intel::fpga_selector{}});

constexpr auto unroll_factor = 8;
auto fpga_policy_c =
  make_fpga_policy<class FPGAPolicyC, unroll_factor>(fpga_policy);
```

Figure 18-10. *Using FPGA policy*

Using DPC++ Parallel STL

In order to use the DPC++ Parallel STL, we need to include Parallel STL header files by adding a subset of the following set of lines. These lines are dependent on the algorithms we intend to use:

- `#include <dpstd/algorithm>`
- `#include <dpstd/numeric>`
- `#include <dpstd/memory>`

`dpstd::begin` and `dpstd::end` are special helper functions that allow us to pass SYCL buffers to Parallel STL algorithms. These functions accept a SYCL buffer and return an object of an unspecified type that satisfies the following requirements:

- Is `CopyConstructible`, `CopyAssignable`, and comparable with operators `==` and `!=`.
- The following expressions are valid: `a + n`, `a - n`, and `a - b`, where `a` and `b` are objects of the type and `n` is an integer value.
- Has a `get_buffer` method with no arguments. The method returns the SYCL buffer passed to `dpstd::begin` and `dpstd::end` functions.

To use these helper functions, add `#include <dpstd/iterators>` to our code. See the code in Figures 18-11 and 18-12 using the `std::fill` function as examples that use the begin/end helpers.

```
#include <dpstd/execution>
#include <dpstd/algorithm>
#include <dpstd/iterators>

sycl::queue Q;
sycl::buffer<int> buf { 1000 };

auto buf_begin = dpstd::begin(buf);
auto buf_end   = dpstd::end(buf);

auto policy = dpstd::execution::make_device_policy<class fill>( Q );
std::fill(policy, buf_begin, buf_end, 42);
// each element of vec equals to 42
```

Figure 18-11. *Using* `std::fill`

REDUCE DATA COPYING BETWEEN THE HOST AND DEVICE

Parallel STL algorithms can be called with ordinary (host-side) iterators, as seen in the code example in Figure 18-11.

In this case, a temporary SYCL buffer is created, and the data is copied to this buffer. After processing of the temporary buffer on a device is complete, the data is copied back to the host. Working directly with existing SYCL buffers, where possible, is recommended to reduce data movement between the host and device and any unnecessary overhead of buffer creations and destructions.

```
#include <dpstd/execution>
#include <dpstd/algorithm>

std::vector<int> v( 1000000 );
std::fill(dpstd::execution::default_policy, v.begin(), v.end(), 42);
// each element of vec equals to 42
```

Figure 18-12. *Using* `std::fill` *with default policy*

Figure 18-13 shows an example which performs a binary search of the input sequence for each of the values in the search sequence provided. As the result of a search for the i^{th} element of the search sequence, a Boolean value indicating whether the search value was found in the input sequence is assigned to the i^{th} element of the result sequence. The algorithm returns an iterator that points to one past the last element of the result sequence that was assigned a result. The algorithm assumes that the input sequence has been sorted by the comparator provided. If no comparator is provided, then a function object that uses `operator<` to compare the elements will be used.

The complexity of the preceding description highlights that we should leverage library functions where possible, instead of writing our own implementations of similar algorithms which may take significant debugging and tuning time. Authors of the libraries that we can take advantage of are often experts in the internals of the device architectures to which they are coding, and may have access to information that we do not, so we should always leverage optimized libraries when they are available.

The code example shown in Figure 18-13 demonstrates the three typical steps when using a DPC++ Parallel STL algorithm:

- Create DPC++ iterators.
- Create a named policy from an existing policy.
- Invoke the parallel algorithm.

The example in Figure 18-13 uses the `dpstd::binary_search` algorithm to perform binary search on a CPU, GPU, or FPGA, based on our device selection.

```
#include <dpstd/execution>
#include <dpstd/algorithm>
#include <dpstd/iterator>

buffer<uint64_t, 1> kB{ range<1>(10) };
buffer<uint64_t, 1> vB{ range<1>(5) };
buffer<uint64_t, 1> rB{ range<1>(5) };

accessor k{kB};
accessor v{vB};

// create dpc++ iterators
auto k_beg = dpstd::begin(kB);
auto k_end = dpstd::end(kB);
auto v_beg = dpstd::begin(vB);
auto v_end = dpstd::end(vB);
auto r_beg = dpstd::begin(rB);

// create named policy from existing one
auto policy = dpstd::execution::make_device_policy<class bSearch>
  (dpstd::execution::default_policy);

// call algorithm
dpstd::binary_search(policy, k_beg, k_end, v_beg, v_end, r_beg);

// check data
accessor r{rB};
if ((r[0] == false) && (r[1] == true) &&
    (r[2] == false) && (r[3] == true) && (r[4] == true)) {
  std::cout << "Passed.\nRun on "
            << policy.queue().get_device().get_info<info::device::name>()
            << "\n";
} else
  std::cout << "failed: values do not match.\n";
```

Figure 18-13. *Using* `binary_search`

Using Parallel STL with USM

The following examples describe two ways to use the Parallel STL algorithms in combination with USM:

- Through USM pointers
- Through USM allocators

If we have a USM allocation, we can pass the pointers to the start and (one past the) end of the allocation to a parallel algorithm. It is important to be sure that the execution policy and the allocation itself were created for the same queue or context, to avoid undefined behavior at runtime.

If the same allocation is to be processed by several algorithms, either use an in-order queue or explicitly wait for completion of each algorithm before using the same allocation in the next one (this is typical operation ordering when using USM). Also wait for completion before accessing the data on the host, as shown in Figure 18-14.

Alternatively, we can use `std::vector` with a USM allocator as shown in Figure 18-15.

```
#include <dpstd/execution>
#include <dpstd/algorithm>

sycl::queue q;
const int n = 10;
int* d_head = static_cast<int*>(
  sycl::malloc_device(n * sizeof(int),
                      q.get_device(),
                      q.get_context()));

std::fill(dpstd::execution::make_device_policy(q),
          d_head, d_head + n, 78);
q.wait();

sycl::free(d_head, q.get_context());
```

Figure 18-14. *Using Parallel STL with a USM pointer*

```
#include <dpstd/execution>
#include <dpstd/algorithm>

sycl::queue Q;
const int n = 10;
sycl::usm_allocator<int, sycl::usm::alloc::shared>
                          alloc(Q.get_context(), Q.get_device());
std::vector<int, decltype(alloc)> vec(n, alloc);

std::fill(dpstd::execution::make_device_policy(Q),
                            vec.begin(), vec.end(), 78);
Q.wait();
```

Figure 18-15. *Using Parallel STL with a USM allocator*

Error Handling with DPC++ Execution Policies

As detailed in Chapter 5, the DPC++ error handling model supports two types of errors. With *synchronous* errors, the runtime throws exceptions, while *asynchronous* errors are only processed in a user-supplied error handler at specified times during program execution.

For Parallel STL algorithms executed with DPC++ policies, handling of all errors, synchronous or asynchronous, is a responsibility of the caller. Specifically

- No exceptions are thrown explicitly by algorithms.
- Exceptions thrown by the runtime on the host CPU, including DPC++ synchronous exceptions, are passed through to the caller.
- DPC++ asynchronous errors are not handled by the Parallel STL, so must be handled (if any handling is desired) by the calling application.

To process DPC++ asynchronous errors, the queue associated with a DPC++ policy must be created with an error handler object. The predefined policy objects (`default_policy` and others) have no error handlers, so we should create our own policies if we need to process asynchronous errors.

Summary

The DPC++ library is a companion to the DPC++ compiler. It helps us with solutions for portions of our heterogeneous applications, using pre-built and tuned libraries for common functions and parallel patterns. The DPC++ library allows explicit use of the C++ STL API within kernels, it streamlines cross-architecture programming with Parallel STL algorithm extensions, and it increases the successful application of parallel

algorithms with custom iterators. In addition to support for familiar libraries (libstdc++, libc++, MSVS), DPC++ also provides full support for SYCL built-in functions. This chapter overviewed options for leveraging the work of others instead of having to write everything ourselves, and we should use that approach wherever practical to simplify application development and often to realize superior performance.