OpenMP Advanced Overview SIMD and Target Offload

Doug Jacobsen, Intel Corporation
John Pennycook, Intel Corporation
Tim Mattson, Intel Corporation
Alejandro Duran, Intel Corporation

Topics

- We Will Cover:
  - Accelerated OpenMP* basics review
  - Vectorization and OpenMP* SIMD
  - OpenMP* Device Usage (target)
- We will NOT Cover:
  - Detailed OpenMP* basics usage
  - OpenMP* Tasking
  - Affinity
  - NUMA effects
- Reference material in backup:
  - Interface descriptions for all described constructs
  - Brief Introduction to OpenMP* 5.0 Changes

OpenMP* Review

OpenMP* Basics Review

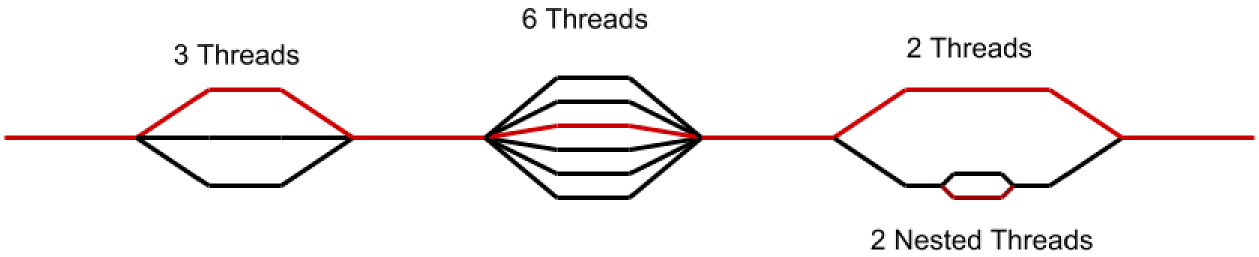
OpenMP*:

- Application Program Interface (API) focused on the expression of parallelism within an application
- Provides methods for creating multi-threaded, heterogeneous device, and vectorized applications
- Contains standardized directives and library functions
- Supports C, C++, and Fortran
- Was established in 1997

OpenMP* Threads

Threads are the original and most basic concept within OpenMP*.

A master thread spawns a team of threads when it encounters a parallel region



Threads can distribute work among themselves to accomplish the same task faster, or can work on independent tasks in parallel.

OpenMP* Directive Overview

Directive	Description
#pragma omp parallel !$omp parallel	Create threads in a parallel region
#pragma omp for !$omp do	Distribute iterations of a loop among threads
#pragma omp for reduction(+:sum) !$omp do reduction(+:sum)	Distribute iterations of a loop among threads, and reduce the thread private ‘sum’ after the loop is complete.
#pragma omp single !$omp single	Execute code with a single thread
#pragma omp master !$omp master	Execute code only on the master thread
#pragma omp barrier !$omp barrier	Wait for all threads in current team before proceeding

OpenMP* Data Sharing Attributes

Data Sharing Level	Description
shared	Variables are shared among all members. If one thread updates the value, all members see the new value.
private	Variables are private among members. If one member updates the value, no other member sees that update.
firstprivate	Same as private, but all members are initialized with the value before the scope changed.
lastprivate	Same as private, but the logically last iteration of a loop is used to set the value of the variable after the region ends.

OpenMP*: Beyond Threading

Modern computational architectures include everything from traditional CPUs, to Vector Processing Units, to Graphics Processing Units, and beyond.

OpenMP* is constantly evolving to help developers properly utilize the features of their underlying hardware.

The remainder of this talk will focus on more advanced topics from OpenMP*. These topics include:

- SIMD (Vectorization)
- Target Teams (Heterogeneous Devices)

Introduction to SIMD

Introduction to SIMD

Scalar Code

- Executes code one element at a time.
- Not all scalar instructions have a vector equivalent (e.g. cpuid)

Vector Code

- Executes code multiple elements at a time.
- Single Instruction Multiple Data.
- Not all vector instructions have a scalar equivalent (e.g. shuffle)

[Scalar] 1 elem at a time

```
addss xmm1, xmm2
```

[SSE] 4 elems at a time

```
addps xmm1, xmm2
```

[AVX] 8 elems at a time

```
vaddps ymm1, ymm2, ymm3
```

[MIC / AVX-512] 16 elems at a time

```
vaddps zmm1, zmm2, zmm3
```

Introduction to SIMD

Hardware (Colors represent SIMD lanes)

3 1 2 4 + 2 3 5 5 = 5 4 7 9

Software (Vectorization)

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

```
int i = 0;
for (; i < size; i += 4) {
    for( int j = 0; j < 4; j++)
        C[i+j] = A[i+j] + B[i+j];
}
for (; i < size; i++) {
    C[i] = A[i] + B[i];
}
```

Utilizing SIMD Instructions

Utilizing SIMD Instructions

SIMD Libraries

- Use a library that is already SIMD-optimized (e.g. Intel® Math Kernel Library)

Implicit (Auto) Vectorization

- Use a compiler that recognises vectorization opportunities (e.g. Intel® Composer XE)
- Possibly annotate with vendor specific pragmas (i.e. #pragma ivdep)

Explicit (Manual) Vectorization

- Express vectorization opportunities to a compiler (e.g. OpenMP* 4.0)
- Write intrinsics code (or inline assembly!)

Utilizing SIMD Instructions – Auto-vectorization

Very powerful, but a compiler cannot make unsafe assumptions.

```
int* g_size;
void not_vectorizable
(float* a, float* b, float* c, int* ind)
{
	for (int i = 0; i < *g_size; i++)
	{
		int j = ind[i];
		c[j] += a[i] + b[i];
	}
}
```

- Unsafe Assumptions:
  - a, b and c point to different arrays.
  - Value of global g_size is constant.
  - ind[i] is a one-to-one mapping.

Utilizing SIMD Instructions – Auto-vectorization

Very powerful, but a compiler cannot make unsafe assumptions.

```
int* g_size;
void vectorizable
(float* restrict a, float* restrict b, float* restrict c, int* ind)
{
      int size = *g_size;
      #pragma ivdep
      for (int i = 0; i < size; i++)
      {
            int j = ind[i];
            c[j] += a[i] + b[i];
      }
}
```

- Safe Assumptions:
  - a, b and c point to different arrays. (restrict)
  - Value of global g_size is constant. (pointer dereference outside loop)
  - ind[i] is a one-to-one mapping. (#pragma ivdep)

Utilizing SIMD Instructions – Compiler Pragmas

```
float sum = 0.0f;
float *p = a;
int step = 4;

#pragma omp simd reduction(+:sum) linear(p:step)
for (int i = 0; i < N; ++i) {
        sum += *p;
        p += step;
}
```

- The same constructs can have different meaning from each other:
  - The two += operators have a different purpose.
  - The variables sum and p relate differently to the iteration space.
- The compiler has to generate different code.
- OpenMP* 4.0 pragmas allow programmers to express this.

Implicit vs Explicit Vectorization

Implicit

- Automatic dependency analysis.
  (e.g. recognises SIMD reductions)
- Recognizes idioms with data dependencies.
  (e.g. array[i++] = x; -> vcompress)
- Non-inline functions will be scalarized.
- Limited support for outer-loop vectorization (only with –O3).

Explicit

- No dependency analysis.
  (e.g. SIMD reductions must be declared)
- Recognizes idioms without data dependencies.
- Non-inline functions can be vectorized.
- Outer loops can be vectorized.

OpenMP* SIMD Example: STREAM Triad

Let's say we want to measure achievable bandwidth.

In order to achieve the highest bandwidth we can, we need to move a maximum number of bytes per cycle.

The STREAM Triad benchmark is typically used to measure this, it is a simple vector multiply and add of the form:

```
for ( int i = 0; i < N; i++ ) {
   a[i] = b[i] * scalar + c[i];
}
```

OpenMP* SIMD Example: STREAM Triad

OpenMP 4.5

```
#pragma omp parallel for private(j)
for (i = 0; i < N; i+=16) {
   #pragma vector nontemporal(a)
   #pragma omp simd aligned(a,b,c)
   for(j = 0; j < 16; j++) {
      a[i+j] = b[i+j] * scalar + c[i+j];
   }
}
```

OpenMP 5.0

```
#pragma omp parallel for private(j)
for (i = 0; i < N; i+=16) {
   #pragma omp simd aligned(a,b,c) nontemporal(a)
   for(j = 0; j < 16; j++) {
      a[i+j] = b[i+j] * scalar + c[i+j];
   }
}
```

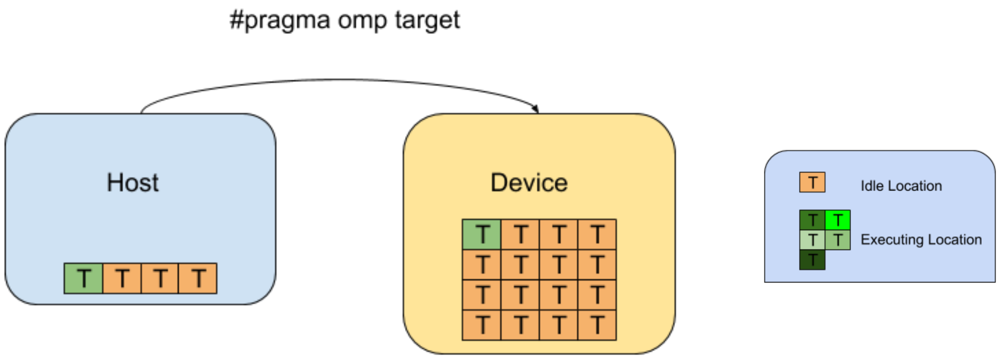
In this example, the 16 is tuned for single precision on AVX512

OpenMP* Target

OpenMP* Target

Introduced in OpenMP* 4.0

The **target** construct is used to create a *target task* that should be offloaded to a device.



We'll carry the STREAM triad example with us to show device execution.

OpenMP* Target Example: STREAM Triad

Always Move Everything

```
#pragma omp target map(tofrom:a[0:N], b[0:N], c[0:N], scalar)
for (i = 0, i < N, i++){
    a[i] = b[i] * scalar + c[i];
}
```

Only Move What's Needed

```
#pragma omp target map(to:b[0:N], c[0:N], scalar) map(tofrom:a[0:N])
for (i = 0, i < N, i++){
    a[i] = b[i] * scalar + c[i];
}
```

OpenMP* Target Example: STREAM Triad

Structured Data Management

```
#pragma omp target data map(to:b[0:N],c[0:N],scalar) \
                        map(from:a[0:N])
{
  #pragma omp target
  for (i = 0, i < N, i++){
    a[i] = b[i] * scalar + c[i];
  }
}
```

Unstructured Data Management

```
#pragma omp target enter data map(to:a[0:N], b[0:N], \
                                  c[0:N],scalar)

#pragma omp target
for (i = 0, i < N, i++){
    a[i] = b[i] * scalar + c[i];
}

#pragma omp target exit data map(from:a[0:N])
```

OpenMP* Target Example: STREAM Triad

```
#pragma omp target enter data \
            map(to:a[0:N],b[0:N],c[0:N])

#pragma omp target
for (i = 0, i < N, i++){
   a[i] = b[i] * scalar + c[i];
}

#pragma omp target exit data map(from:a[0:N])
```

At the beginning of the target region, b, c, and scalar will be copied onto the device and after the target region is complete, a will be copied back to the host. The target region will be executed on the device.

However….

All of the examples so far have a **serial** execution**!**

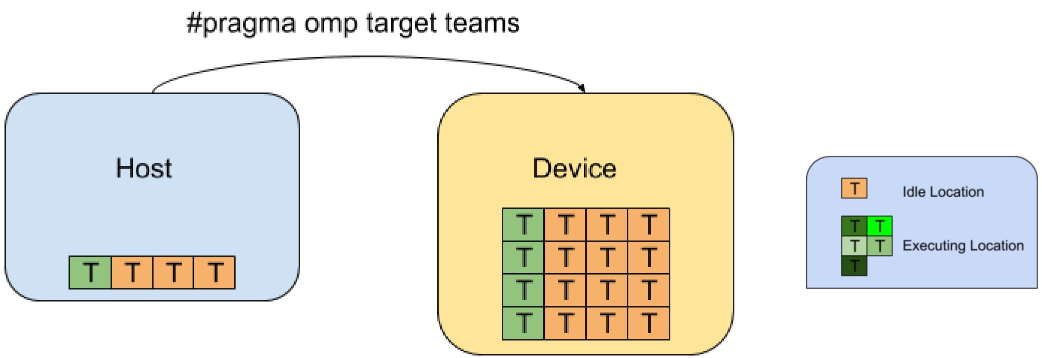
The target construct only creates a target task which is executed by a single thread!

OpenMP* Teams and Distribute

OpenMP* Teams

To utilize multiple threads on a device, we need to first use the **teams** construct.

A teams construct creates a league of teams. Each team consists of some number of threads, and to begin with the master thread of each team executes the code in the teams region.



OpenMP* Teams Example: STREAM Triad

```
#pragma target teams

for (i = 0, i < N, i++){
    a[i] = b[i] * scalar + c[i];
}
```

Now we have multiple threads executing on the device, however they are all performing the **same** work.

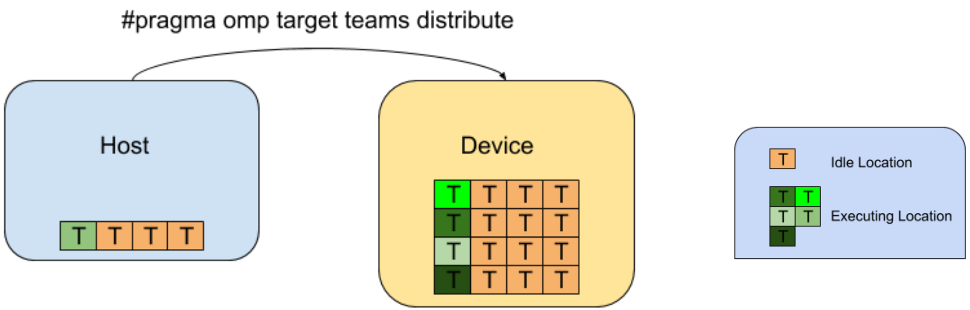
Additionally, only the master thread of each team is doing anything.

OpenMP* Distribute

Right now we have:

- Multiple thread teams working on a device
- All thread teams performing the same work
- Only the master thread of each team executing anything

When we encounter a loop, we can spread the iterations among teams using the **distribute** construct.



OpenMP* Distribute Example: STREAM Triad

```
#pragma target teams distribute

for (i = 0, i < N, i++){
   a[i] = b[i] * scalar + c[i];
}
```

At this point, we have teams performing independent iterations of this loop, however each team is only executing serially.

In order to work in parallel within a team we thread as we normally would with the basic OpenMP usage.

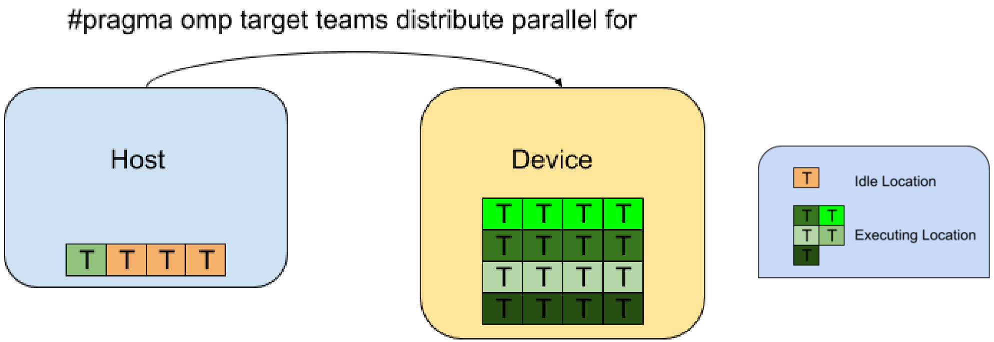
However, there are composite constructs that can help.

OpenMP* Distribute Full Example: STREAM Triad

```
#pragma target teams distribute parallel for

for (i = 0, i < N, i++){
    a[i] = b[i] * scalar + c[i];
}
```

Now, we have teams splitting the work into large chunks on the device, while threads within the team will further split the work into smaller chunks and execute in parallel.



Legal Disclaimers

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. § For more information go to www.intel.com/benchmarks.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at www.intel.com.

Intel, the Intel logo, Look Inside, Xeon, Xeon Phi, VTune, are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

intel®

OpenMP* References

Explicit Vectorization – OpenMP* SIMD Loops

#pragma omp simd / !$omp simd => for/do loop is a SIMD loop.

Clause	Description
private(list)	Listed variables are scoped as private
lastprivate(list)	Listed variables are scoped as lastprivate
reduction(reduction-identifier:list)	Reduce listed variables according to reduction-identifier
collapse(n)	Associate loop construct with n following nested loops
safelen(n)	Maximum allowed ‘distance’ between SIMD lanes in a single execution
simdlen(n)	Preferred SIMD width
aligned(list)	Hint to the run-time to use aligned loads and stores on listed variables
linear(list)	Described later

Explicit Vectorization – OpenMP* 5.0 SIMD Loops

#pragma omp simd / !$omp simd => for/do loop is a SIMD loop.

Clause	Description
nontemporal(list)	Hint to the runtime to use accesses with no temporal locality, i.e. there is no need to cache the data
order(concurrent)	Hint to the run-time that iterations of the loop can be executed in any order, including concurrently

Explicit Vectorization – OpenMP* SIMD Functions

#pragma omp declare simd / !$omp declare simd => function called from a SIMD loop.

Clause	Description
simdlen(n)	Preferred SIMD width is n
aligned(list)	Listed variables should use aligned accesses
uniform(list)	Listed variables have an invariant value for all concurrent calls to the SIMD function
inbranch	Function is always called from within a branch
notinbranch	Function is never called from within a branch
linear(list)	Described later

OpenMP* Linear Clause

The linear clause provides the compiler additional information about variables.

linear(linear-list:step)

Linear List Options	Description
variable val(variable)	Variable listed have linear values with a distance between consecutive SIMD lanes of step.
ref(variable)	Variable listed have linear memory addresses with a distance between consecutive SIMD lanes of step.
uval(variable)	Variable listed have linear values with a distance between consecutive SIMD lanes of step. Variable uses the same storage location across all SIMD lanes, and the logically last iteration is stored out.

OpenMP* Target Construct

#pragma omp target / !$omp target
!$omp end target

=> Create a *target task* task, and execute on a device

Clause	Description
if(scalar-expression)	If scalar-expression is false, device used is host
device(integer-expression)	Controls the device used, as described in data management constructs
private(list)	Listed variables are scoped as private
firstprivate(list)	Listed variables are scoped as firstprivate
map([[map-type-modifier[,]]map-type:]list)	Described later
is_device_ptr(list)	Listed variables are already device pointers (allocated before the target construct)
defaultmap(tofrom:scalar)	If specified, scalars are by default mapped using tofrom. If not specified, nothing is mapped by default.
depend(dependence-type:list)	Follows task dependencies, as previously described.
nowait	If present, execution of the target task may be deferred. If not, the task is an undeferred task.

OpenMP* 5.0 Target Construct

#pragma omp target / !$omp target
!$omp end target

=> Create a *target task* task, and execute on a device

Clause	Description
in_reduction(reduction-identifier:list)	The generated target task will participate in a previously defined task_reduction
defaultmap(implicit-behavior:[variable-category])	Implicit behavior can be one of alloc, to, from, tofrom, firstprivate, none, or default and variable category can be one of scalar, aggregate, pointer, or allocatable (in fortran)
allocate([allocator:]list)	Allocates listed variables using the specified allocator (or a default device allocator)
uses_allocators(allocator[(allocator-traits-arrays)], …)	Each allocator listed will be made available in the target region.
device([device-modifier:]integer-expression)	Modifier can be ancestor, or device_num. Used to determine which device will be selected. ancestor:1 executes on the parent device of the target region.

OpenMP* Target Data Mapping

map([[map-type-modifier[,]]map-type:]list)

Clause	Valid for construct	Description
to	target enter data, target data, target	Copies data from listed variables into device data region
from	target exit data, target data, target	Copies data from listed variables out of the device data region
tofrom	target data, target	Copies data into and out of the device data region for listed variables
alloc	target enter data, target data, target	Allocates memory for listed variables in the device data region
release	target exit data	Releases device pointers for listed variables in the device data region
delete	target exit data	Frees memory for listed variables in the device data region

Structured data regions trigger events upon entry and exit.

In 5.0 requires(unified_shared_memory) can make map() optional

OpenMP* Target Structured Data Management

#pragma omp target data / !$omp target data
!$omp end target data

=> Create a data region for a device

Clause	Description
if(scalar-expression)	If scalar-expression is false, device used is host
device(integer-expression)	Integer expression for the device to use. Must be a non-negative integer less than the value of omp_get_num_devices()
map([[map-type-modifier[,]]map-type:]list)	Described previously
use_device_ptr(list)	Listed items are converted to device pointers in the device data environment
use_device_addr(list) (in 5.0)	Listed items have the address of the corresponding object in the device data environment.

OpenMP* Target Unstructured Data Management

#pragma omp enter target data / !$omp enter target data
#pragma omp exit target data / !$omp exit target data

=> Manage data associated with a device

Clause	Description
if(scalar-expression)	If scalar-expression is false, device used is host
device(integer-expression)	Integer expression for the device to use. Must be a non-negative integer less than the value of omp_get_num_devices() If not specified, device is determined from the default-device-var icv.
map([[map-type-modifier[,]]map-type:]list)	Described previously
depend(dependence-type:list)	Follows task dependencies, as previously described.
nowait	If present, execution of the target task may be deferred. If not, the task is an undeferred task.

OpenMP* Teams Construct

#pragma omp teams/ !$omp teams
!$omp end teams => Create a league of thread teams. The master thread of each team executes the region.

Clause	Description
num_teams(integer-expression)	Sets the maximum number of teams to create within a target task
thread_limit(integer_expression)	Sets the maximum number of threads to create within each team
default(shared | firstprivate | private | none)	Sets the default data sharing level
private(list)	Listed variables are scoped as private
firstprivate(list)	Listed variables are scoped as firstprivate
shared(list)	Listed variables are scoped as shared
reduction(reduction-identifier:list)	Perform a reduction on listed variables across teams using reduction-identifier.

OpenMP* Distribute Construct

#pragma omp distribute / !$omp distribute
!$omp end distribute => Distribute the iterations of one or more loops across the master threads of a league of teams

Clause	Description
private(list)	Listed variables are scoped as private
firstprivate(list)	Listed variables are scoped as firstprivate
lastprivate(list)	Listed variables are scoped as lastprivate
collapse(n)	Specifies how many loops are associated with a particular distribute construct
dist_schedule(kind[, chunk_size])	Kind must be *static*, chunk_size specifies the division of iterations. Chunks are assigned to teams in a round-robin fashion

OpenMP* Distribute Composite Constructs

In addition to using a vanilla *distribute* construct, there are also composite constructs that allow more parallelism.

Construct	Description
#pragma omp distribute simd !$omp [end] distribute simd	Distribute loop chunks among teams, and execute those iterations using SIMD instructions
#pragma omp distribute parallel for !$omp [end] distribute parallel do	Distribute loop chunks among teams, then distribute iterations within each chunk among threads within the team and execute in parallel
#pragma omp distribute parallel for simd !$omp [end] distribute parallel for simd	Distribute loop chunks among teams, then distribute iterations within each chunk among threads within the team and execute in parallel using SIMD instructions

OpenMP* 5.0

OpenMP* 5.0 adds several new features which can be useful in achieving a performance portable application. We won't go over these in detail, but some items to look into are:

Construct	Description
#pragma omp metadirective !$omp metadirective	Conditionally apply a directive from a set to a location.
#pragma omp declare variant !$omp declare variant	Define variants of a function to use on different architectures or devices.
#pragma omp requires !$omp requires	Specify that an implementation requires a specific hardware feature
#pragma omp teams loop !$omp teams loop	Specify a teams construct that contains a loop and nothing else Equivalent to #pragma omp teams distribute parallel for !$omp teams distribute parallel for
#pragma omp scan !$omp scan	Tells the compiler that the following statement is a scan that can be performed in parallel
Memory Management #pragma omp allocate !$omp allocate	More explicit control over memory spaces for host and device, and the ability to define and use custom memory allocators

OpenMP* 5.0 Specification is at: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf