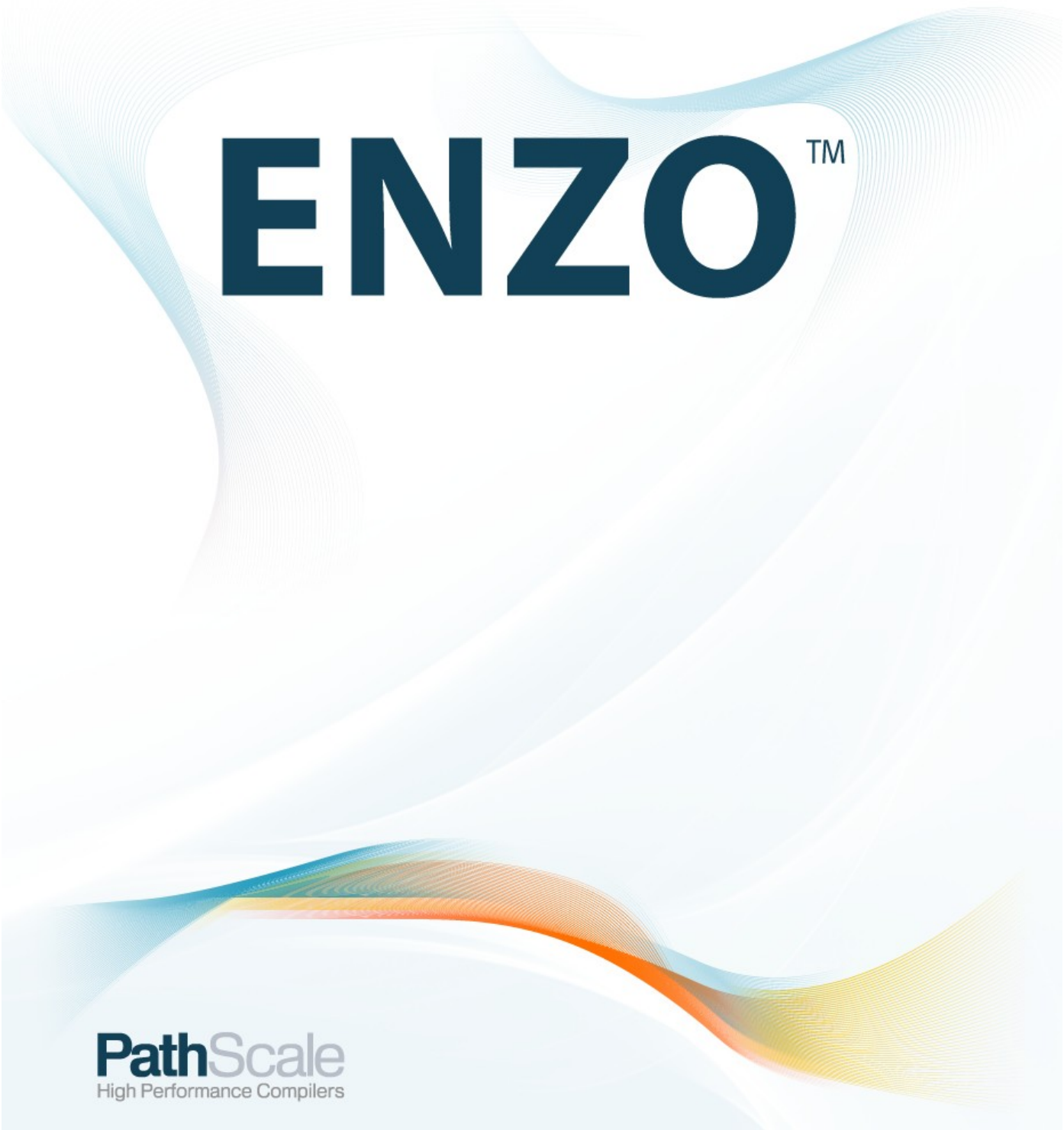
PathScale
High Performance Compilers

ENZO™



NVIDIA® is a registered trademark of the NVIDIA Corporation. PathScale® is a registered trademark of Pathscale Inc.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

PathScale ENZO

User Guide Version 1.1

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Table of Contents

1 Introduction....5
1.1 PathScale ENZO™ Overview....5
1.2 ENZO™ Runtime Overview....7
1.3 PathScale ENZO™ Code Generation....7
1.4 Scope of this Document....7
2 OpenACC....7
2.1 The OpenACC accelerated regions....7
3 OpenMP....9
4 Remote Procedure Call and Data management....10
5 ENZO™ Memory Model....11
6 OpenACC Directives....11
6.1 Introduction....11
6.2 Syntax of the OpenACC directives....12
6.3 Directives for offloading code regions to HWA....13
6.3.1 parallel Directive....13
6.3.2 kernels Directive....15
6.4 Data Allocation and Transfer....17
6.4.1 data Directive....17
6.4.2 enter data / exit data directives....18
6.4.3 declare directive....20
6.4.4 update directive....21
6.4.5 Subarrays in OpenACC....22
6.5 Loop directive....24
6.6 Asynchronous execution....25
6.7 Procedure calls....26
6.8 Nested parallelism....27
6.9 Atomic operations....28
7 OpenMP directives....28
7.1 Introduction....28
7.2 Syntax of OpenMP directives....29
7.3 Parallel directive....30
7.4 Worksharing constructs....31
7.4.1 Loop construct....31
7.4.2 Sections construct....32
7.4.3 Single construct....33
7.4.4 Workshare construct....33
7.5 Device constructs....33
7.5.1 Target directive....34
7.5.2 Teams directive....34
7.5.3 Distribute directive....35
7.6 Data allocation and transfers....35
7.6.1 Target data directive....35
7.6.2 Target update directive....36
7.6.3 Declare target directive....37
7.6.4 Threadprivate directive....38
7.6.5 OpenMP array subscripts....38
7.7 Tasking constructs....39
7.7.1 task construct....39
7.7.2 taskyield directive....40

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

7.7.3 taskwait directive....40
7.7.4 taskgroup directive....41
7.8 Synchronization....41
7.8.1 master construct....41
7.8.2 flush construct....42
7.8.3 critical construct....42
7.8.4 barrier construct....42
7.8.5 atomic construct....43
7.8.6 ordered construct....43
7.9 SIMD....43
7.9.1 simd directive....43
7.9.2 declare simd directive....44
7.10 Cancellation....45
8 Supported Languages....46
9 Compiling OpenACC and OpenMP Applications....46
9.1 Overview....46
9.2 Common Command Line Parameters....47
10 Running OpenACC and OpenMP Applications....47
10.1 Launching the Application....47
11 Improved code generation and performance....47
11.1 Loops performance optimizations....48
11.1.1 OpenACC kernels region....48
11.1.1.1 OpenACC independent clause....48
11.1.1.2 OpenACC seq clause....48
11.1.2 reduction....49
11.1.3 OpenACC tile clause....51
11.1.4 Levels of parallelism....52
11.1.4.1 OpenACC levels of parallelism....52
11.1.4.2 OpenMP levels of parallelism....53
11.1.4.3 Levels of parallelism: examples....53
11.1.5 Collapse clause....54
11.2 OpenACC Cache management....55
12 Environment variables....56
12.1 OpenACC environment variables....56
12.2 OpenMP environment variables....56
13 Runtime API....56
13.1 OpenACC Runtime API....56
13.1 OpenMP Runtime API....57
14 ENZO™ Supported HWA....57
14.1.1 Hardware Accelerators....57

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

1 Introduction

The PathScale ENZO™ Suite brings the power of the OpenACC directives together with direct code generation for NVIDIA Tesla™ GPU. This approach leverages the strength of the GPU as a hardware accelerator (HWAs) to replace traditional SIMD computing units.

PathScale ENZO using OpenACC directives allow the programmer to write hardware independent applications where hardware specific codes are dissociated from the legacy code as additional software plug-ins.

1.1 PathScale ENZO Overview

PathScale ENZO currently supports OpenACC and OpenMP directives for Fortran, C and C++ which in combination with the ENZO runtime allows seamless execution of heterogeneous applications.

To accelerate the execution of your application with ENZO, the first step is to identify the regions of the application source code which is suitable for the HWA target. Those will then become either *parallel* or *kernels* regions or (see Section 2.1) using the OpenACC directives. The hardware-accelerated versions of the regions are defined in their specific language i.e. Fortran and using the OpenACC programming model.

The OpenACC/OpenMP annotated source code is parsed by the PathScale Fortran, C and C++ frontends to translate the OpenACC/OpenMP directives into calls to the ENZO runtime API. The ENZO runtime API is in charge of managing the concurrent execution of the host and device code regions.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Figure 1 - PathScale ENZO Compilation Process

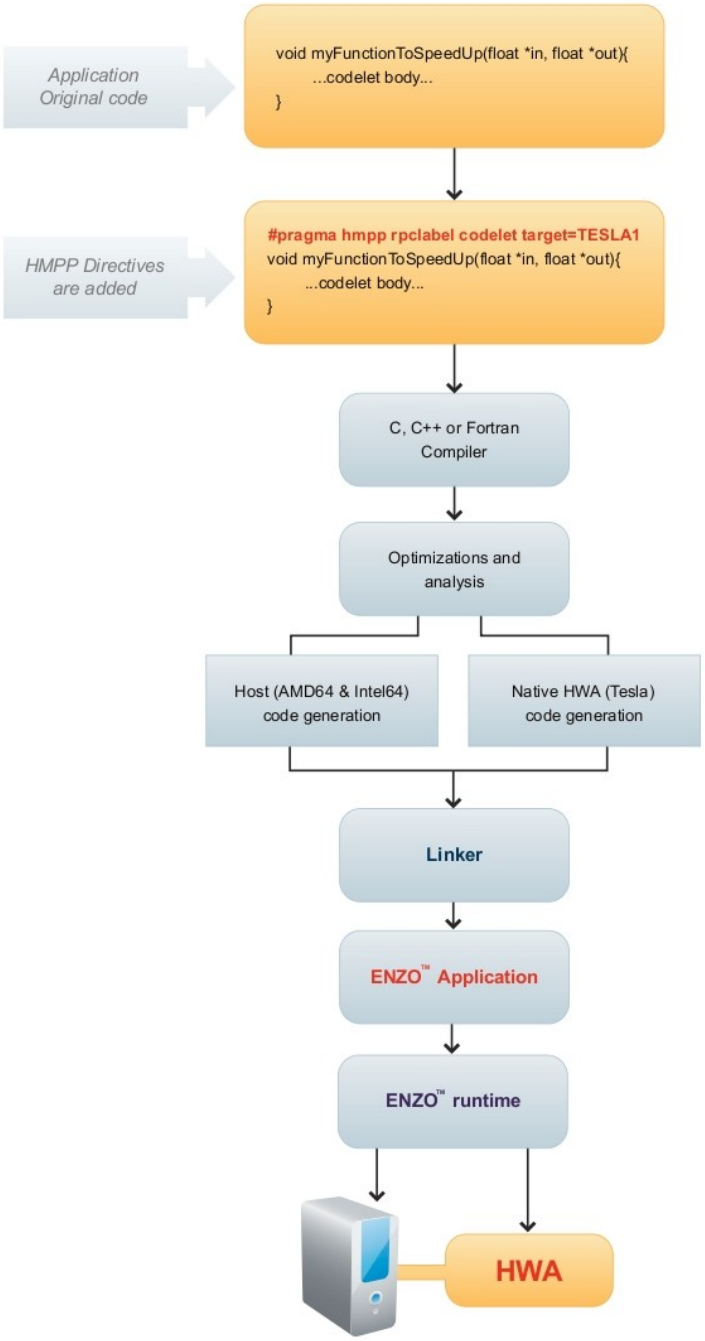


Figure 1 shows the compilation and code generation flow of an ENZO application using HMPP accelerator directives. The path is identical to ENZO OpenACC and OpenMP application. The unified flow for both the native code and HWA code until the final stage when the targets are lowered down to native heterogeneous assembly.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

1.2 ENZO Runtime Overview

The ENZO runtime API is in charge of the execution of the remote procedure calls to the HWA. Linked to the application, this library allocates memory and initializes the HWA in order to allow the execution of the kernels. It relays communications between the host and the HWA and manages the asynchronous execution of kernels and data transfers.

1.3 PathScale ENZO Code Generation

PathScale ENZO does direct to native HWA instruction code generation to maximize performance and optimization for your ENZO applications. The entire process is a complete unified solution using no source-to-source conversion and PathScale ENZO runtime handles the entire offloading process including execution and data management.

1.4 Scope of this Document

This manual covers the PathScale ENZO runtime, OpenACC and OpenMP directives.

For documentation on the compiler CLI interface and installation instructions please reference PathScale ENZO CLI Guide and PathScale ENZO Installation notes.

2 OpenACC

OpenACC is based on the concept of source code directives that specify which parts of the original program must/may be targeted to execution on the HWA . The ENZO runtime API library is in charge of calling the remote procedure (RPCs) as well as managing resources. OpenACC supports two styles of acceleratable code region declaration — OpenMP-style *parallel* regions directly instructing compiler on how the kernels should be generated from the original code — and more relaxed *kernels* regions marking specific code blocks as potentially interesting for offloading. OpenACC directives have the facility of defining data regions allowing the programmer to share data between distinct kernels.

Please note that while PathScale ENZO does semantic checking on the directives it does not guarantee all errors of incorrect usage will be reported.

2.1 The OpenACC accelerated regions

OpenACC defines two kinds of regions targeted for acceleration: *parallel region* and *kernels region*.
Parallel region defines a single kernel. For the *kernels region* compiler analyses loop nests and extracts as many kernels as needed.
OpenACC provides means to mark loops targeted for parallel execution by a *loop* directive.

OpenACC supports three levels of parallelism — gangs, workers and vectors. (See section 3.8 — Levels of parallelism).

All code in a *parallel region* that is not inside a *parallel loop* with gang-level parallelism, will be executed redundantly by all gangs.

There is no mechanisms for barriers inside a *parallel region,* but implicit syncronization is assumed on exit from it, unless it is explictly marked as async.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

ENZO Runtime protocol supports data exchange between accelerated and hosts regions.

Regions marked as *parallel* or *kernels must* the following properties:

- Branches to and from these regions are not allowed
- Program must not depend on the order of evaulation of clauses and/or produced side-effects

These properties ensure that a codelet RPC can be remotely executed by a HWA. This RPC and its associated data transfers can be asynchronous.

By default, all the parameters are uploaded to the HWA just before the RPC and downloaded just after its execution has completed. Below is a C code example of a correct *parallel* and *kernels* regions defintion:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Parallel region definition

```
static void foo(int n, float v1[n], float v2[n], float v3[n]) {
  int i;
  #pragma acc parallel
  {
    #pragma acc loop
    for (i = 0 ; i < n ; i++) {
      v1[i] = v2[i] + v3[i];
    }
  }
} // TBD - does not really work unless code is inlined to a place where arrays are
defined (parallel_region.c)
```

Kernels region definition

```
static void foo(int n, float v1[n], float v2[n], float v3[n]) {
  int i;
  #pragma acc kernels
  {
    #pragma acc loop
    for (i = 0 ; i < n ; i++) {
      v1[i] = v2[i] + v3[i];
    }
  }
} // TBD - does not really work unless code is inlined to a place where arrays are
defined (parallel_region.c)
```

3 OpenMP

OpenMP provides a set of source code directives that describe how the source code should be parallelized and/or offloaded to HWA device. Unlike OpenACC, OpenMP approach is not limited to host-device model, but also covers multi-core parallelization tasks, although we will concentrate on host-device model in the current document. Two main groups of OpenMP directives are parallelization/offloading directives and data management directives.

OMP *parallel* directive is used to mark the start of parallel region. On a *parallel* region start a team of threads is allocated for its execution. By default, code in a region is executed redundantly by all threads, but it is possible to use worksharing constructs to distribute individual statements, loop iterations or code region sections between different threads.

Second level of thread grouping may be achieved by using of *teams* directive. The *teams* directive creates a league of thread teams, with master thread of each team starting to execute the corresponding region.

Simd directive may be used to parallelize loop iterations across multiple vector lanes.

It is allowed to have nested parallel regions — if a *parallel* directive is met inside a parallel region, then a new thread team is created for each thread executing the outer region.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

When a thread is allocated for code execution, it is associated with implicit task. It is also possible to create explicit tasks using *task* directive. Explicit task is assigned to some thread of the current team. Its actual execution may be suspended until there is a thread available to execute it. Explicit tasks may be useful for specificitation of asynchrounous computation trees — explicit tasks are launched asynchronously and inter-task dependencies may be specified using *depend* clause of *task* directive.

Target directive is used to mark code region for offloading. If there is an available device capable of execution of the region, then code is offloaded to the device and a new device thread is allocated to start its execution.

OpenMP offloading

```
int i;
// offload code to HWA
#pragma omp target
// Initialize parallel execution mode - allocate threads
 #pragma omp parallel
{
  // distribute loop iterations across allocated threads
   #pragma omp for
  for (int i = 0 ; i < n ; i++) {
    v1[i] = v2[i] + v3[i];
  }
}
```

4 Remote Procedure Call and Data management

OpenACC and OpenMP allow both explicit and implicit data management.
By default, when an offloaded code *region* is defined, all input parameters for the region are uploaded to the HWA just before the RPC. The output parameters are downloaded back to the host memory once the region has sucessfully completed the execution.

User may control which arrays should be transferred back and forth upon entering and exiting the region using *copyin/copyout* clauses. Furthermore, array image may be allocated on the device using OpenACC *create*/OpenMP map(alloc:) clause and later — inside parallel region — manually synchronized with host using OpenACC *update*/OpenMP *target update* clause.

Default behaviour, when data is transferred between host and device on every RPC call can cause severe performance penalties, if a non-scalar result of kernel execution is used by a subsequent kernel. In that case we would have to do two excessive memory transfer from device to host (output) and from host to device (input).
In order to avoid it, in OpenACC/OpenMP it is possible to define a device data region using *data/target data* directive that may span along multiple offloaded *regions* and allocate data arrays on the device to be used by several kernels.

In the case of a synchronous (default) or asynchronous codelet RPC, when an error occurs ENZO will call abort(), report an error and exit.

Asynchronous data transfer or asynchronous codelet execution are hardware accelerator dependant.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

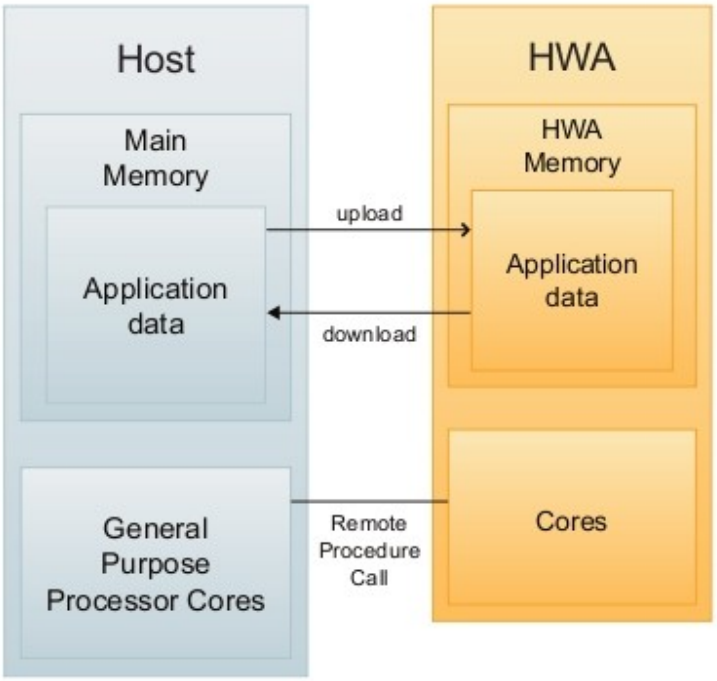
Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

5 ENZO Memory Model

OpenACC/OpenMP support systems with shared memory between device and host and systems with distinct memories. We will assume distinct memory model for the rest of the document.

In the current version of ENZO, the memory address managed at the host level and at the HWA level are different (see Figure 3). The “Application” and the ENZO runtime API have their own private memory. ENZO deals with this in a transparent way for the user. ENZO can be seen as programming glue between target-specific programming environments and general purpose programming.

Figure 3 - ENZO memory model



6 OpenACC Directives

6.1 Introduction

The OpenACC directives may be seen as “meta-information” added in the application source code. They are safe meta-information i.e. they do not change the original code. They address the remote execution (RPC) of code regions as well as the transfers of data to/from the HWA memory.

The simplest use case of OpenACC directives is to mark a loop for parallel execution on the device.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Example of a simple OpenACC parallel loop declaration

```
# pragma acc parallel loop
for (i = 0 ; i < n ; i++) {
  v1[i] = v2[i] + v3[i];
}
```

The table below introduces the OpenACC directives. OpenACC directives address different needs: some of them are dedicated to declarations and others are dedicated to the management of the execution.

Table 1- OpenACC Directives

	Control flow instructions	Directives for data management
Declarations	• parallel • kernels • loop • routine	• data • declare
Operational Directives	• atomic • wait	• enter data • exit data • update • cache

6.2 Syntax of the OpenACC directives

In order to simplify the notations, regular expressions will be used to describe the syntax of the OpenACC directives. Below is a short summary of the main notations used.

- "?" The question mark indicates there is no preceding item or one preceding item.
- "*" The asterisk indicates there are zero or more the preceding items.
- "+" The plus sign indicates that there is one or more the preceding items.

Furthermore, to keep the notation as simple as possible, we separately present the notation used in stand-alone codelet context of the one used with group of codelets. The main difference between the two syntaxes lies in an additional label dedicated to the management of the groups.

We also introduced a color convention for the description of syntax directives:

- Reserved OpenACC keywords are in **blue**
- Elements of grammar which can be declared as OpenACC keywords are in **red**
- Code which is meant to be empahsized is in **bold black**
- Highlighted code is in **magenta**

The general syntax of the OpenACC directives is:

- For C and C++

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma acc directive_type [, directive_clauses]*
```

- For Fortran 95, 2003 and 2008

```
!$acc directive_type [, directive_clauses]*
```

Where:

- directive_type: is the type of the directive;
- directive_clauses: designates some parameters associated to the directive_type. These parameters may be of different kinds and specify either some arguments given to the directive either a mode of execution (asynchronous versus synchronous for example);

Furthermore, the clauses may accept arguments. Typically these arguments are some integer values specifying number of parallel execution lanes, etc. or identifiers.

OpenACC directive examples:

```
#pragma acc parallel num_gangs(100) vector_length(8) async
#pragma acc loop reduction(*:x)
#pragma acc update host(y) wait
```

It is also allowed to combine *parallel* and *kernels* directives with *loop* directive, which has a semantics of consecutive specification of two directives:

```
#pragma acc parallel loop
#pragma acc kernels loop
```

In the remainder of this document, most examples of directives will be given in C. Fortran directives only differ by their prefix.

6.3 Directives for offloading code regions to HWA

6.3.1 parallel Directive

A parallel directive marks the start of core region dedicated to be run as a kernel on HWA.

The syntax of the directive is:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma acc parallel        [async[(handle-id)]?]
                            [wait[(handle-id-list)]?]
                            [num_gangs(gangs-count)]
                            [num_workers(workers-count)]
                            [vector_length(vec-length)]
                            [device_type(device-name-list)]*
                            [if(condition)]
                            [reduction(operator:var-list)]*
                            [copy(var-list)]*
                            [copyin(var-list)]*
                            [copyout(var-list)]*
                            [create(var-list)]*
                            [present(var-list)]*
                            [present_or_copy(var-list)]*
                            [present_or_copyin(var-list)]*
                            [present_or_copyout(var-list)]*
                            [present_or_create(var-list)]*
                            [deviceptr(var-list)]*
                            [private(var-list)]*
                            [firstprivate(var-list)]*
                            [default(none)]
```

Only the async, wait, num_gangs, num_workers, and vector_length clauses may follow a device_type clause.

Where:

- *async* clause specifies that the *parallel region* will be executed asynchronously with the following code. Optional handle-id argument may be used to associate region execution with synchronous execution queue
- *wait* clause forces runtime system to delay launch of region execution until all currently started operations (or enqued to execution queues with handle-ids) finish their execution
- *num_gangs* specifies number of parallel gangs allocated for execution of region
- *num_workers* specifies number of workers for each gang
- *vector_length* specifies number of vector lanes in a vector
- *device_type* specifies device name, typically this clause is followed by a list of clauses that tune execution mode for that device type. Commonly used device types are «nvidia», «radeon» and «xeonphi»
- *if* clause is used to specify condition, based on which either host- or device- version of region is executed
- *copyin* clause declares that value of one or more variable should be copied from host to device upon region enter
- *reduction* clause specifies reduction operator and the set of variables. Reduction result is avaiable on region exit. See section 7 of the current document for more information on this clause.
- *copyout* clause declares that value of one or more variable should be copied back from device to host upon region exit
- *copy* clause is a composition of *copyin* and *copyout* clauses
- *create* clause is used to allocate memory on device, no data transfer between host and device memory is assumed
- *present* clause is used to tell the implementation that corresponding data fragment is already available on accelerator memory due to previous data transfers
- *present_or_** clause group combines semantics of *present* and *copy** clauses. If correponding data fragment is present on accelerator memory, then no further action is required, otherwise behaviour is as for corresponding *copy* clause

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

- *deviceptr* marks pointer variables as pointers to device memory
- *private* clause instructs implementation to create a private copy of each variable on the list for each of parallel gangs allocated for region execution
- *firstprivate* clause behaves similar to *private* clause, additionaly each private variable is initialized with a value of this variable on host
- *default(none)* clause prohibits implicit inference of data attributes for any variable used in region

Parallel directive may also be combined with loop directive, if the code region of interest consists of a single loop nest.

Parallel region with device-specific tuning

```
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm], float *outv){
  int i, j;
  #pragma acc parallel loop collapse(2) \
                       device_type(nvidia) num_gangs(256) num_workers(64) \
                       device_type(radeon) num_gangs(128) num_workers(32)
  for (i = 0 ; i < sm ; i++) {
    for (j = 0 ; j < sn ; j++) {
      outv[i] += inv[j] * inm[i][ j];
    }
  }
} // TBD arrays in header
```

The code sample above gives an example of *parallel* region with combined loop directive and device-specific gridification tuning.

Note: *device_type* clause is not supported in the current release of PathScale ENZO compiler, it will supported in future releases.

6.3.2 kernels Directive

The kernels directive allows to mark code region as potentially interesting for offloading. Compiler analyses code region and decides on how many kernels to generate (typically — one for each top-level loop), how to gridify threads executing loop iterations (how many parallel gangs, workers in a gang and vector lanes in a vector to allocate). Automated offloading of *kernels* region does not assume that no hints or instructions to compiler can be made inside *kernels* region; quite the contrary most of OpenACC constructs allowed for *paralell* region are also allowed for *kernels* region.

The syntax of the directive is:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma acc kernels         [async[(handle-id)]?]
                            [wait[(handle-id-list)]?]
                            [device_type(device-name-list)]*
                            [if(condition)]
                            [copy(var-list)]*
                            [copyin(var-list)]*
                            [copyout(var-list)]*
                            [create(var-list)]*
                            [present(var-list)]*
                            [present_or_copy(var-list)]*
                            [present_or_copyin(var-list)]*
                            [present_or_copyout(var-list)]*
                            [present_or_create(var-list)]*
                            [deviceptr(var-list)]*
                            [default(none)]
```

Set of clauses allowed for *kernels* directive is a subset of clauses allowed for *parallel* construct and their semantics is effectively the same.

Source code below shows example of kernel region with multiple loop nests:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Kernels region with multiple loop nests

```
static void matvec(int sn, int sm,
                   float inv[sm], float inm[sn][sm], float *outv){
  int i, j;
  #pragma acc kernels
  {
    // distinct kernel will be most likely generated for this loop nest
    for (i = 0 ; i < sm ; i++) {
      for (j = 0; j < sn; j++ ) {
        inm[i][j] = - inm[j][i];
      }
    }
    // distinct kernel will be most likely generated for this loop nest
    for (i = 0 ; i < sm ; i++) {
      for (j = 0 ; j < sn ; j++) {
        outv[i] += inv[j] * inm[i][j];
      }
    }
  }
} // TBD: re-check
```

6.4 Data Allocation and Transfer

6.4.1 data Directive

Data directive defines a static lifetime region for variables allocated in HWA memory. It can be used to allocate memory objects in the device memory used by several kernels. Data directive clauses may be used to instruct implementation on how the actual data should be transferred between host and device memory.

The syntax is:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma acc data
                    [if(condition)]
                    [copy(var-list) ]*
                    [copyin(var-list) ]*
                    [copyout(var-list) ]*
                    [create(var-list) ]*
                    [present(var-list) ]*
                    [present_or_copy(var-list) ]*
                    [present_or_copyin(var-list) ]*
                    [present_or_copyout(var-list) ]*
                    [present_or_create(var-list) ]*
                    [deviceptr(var-list) ]*
```

Semanicst of all of the *data* directive clauses are described in *parallel* region section of the current document.

Below is the example of static data region usage. Lifetime of arrays A and B on the device spans along the whole data block. Array A is copied to the device memory from host memory upon region enter and used by two kernels. Array B is copied from device memory to host memory upon region exit.

```
#pragma acc data copyin(A) copyout(B)
{
    #pragma acc parallel loop
    for (int i = 0; i < n; i++) {
       A[i] = A[i] * i + C[i]; // copyin property derived for C array automatically
    }
    #pragma acc parallel loop
    for (int i = 0; i < n; i++) {
       B[i] = A[i] - i;
    }
}
```

6.4.2 enter data / exit data directives

enter data and *exit data* clauses may be used to create dynamic data lifetime regions.

enter data clause is used to allocate data blocks in device memory (and copy actual data from host, if required).

It has the following syntax:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma acc enter data
                      [if(condition)]
                      [async[(handle-id)]?]
                       [wait[(handle-id-list)]?]
                      [copyin(var-list) ]*
                      [create(var-list) ]*
                      [present_or_copyin(var-list) ]*
                      [present_or_create(var-list) ]*
                      [deviceptr(var-list) ]*
```

Exit data clause can be used to deallocate data and (optionally) to move data from device memory to host.

It has the following syntax:

```
#pragma acc exit data
                      [if(condition)]
                      [async[(handle-id)]?]
                       [wait[(handle-id-list)]?]
                      [copyout(var-list) ]*
                      [delete(var-list) ]*
```

Please note new *delete* clause that should be used for deallocation of data in device memory.

Example below sketches possible scenario for *enter data* and *exit data* directvies usage. Application allocates array in device memory before starting compute portion of code, this array is then used by multiple kernels during code exeuction and deallocated when it is not needed anymore.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#define N 1024
int A[N];

void app_init() {
    #pragma acc enter data create(A)
}

void app_shutdown() {
    #pragma acc exit data delete(A)
}

...

void func() {
    #pragma acc parallel loop
    for (int i = 0; i < N; i++)
       A[i] = A[i] * 2;
}

void main() {
 app_init();
 func();
 ...
 app_shutdown();

}  // TBD: dynamic_data_lifetime.c
```

6.4.3 declare directive

Declare directive creates static data lifetime region very much the same as *data* directive, but the rules for region bounds inference are different. If *declare* directive is met in a function scope or subroutine scope, data lifetime region spans along corresponding function or subroutine. If *declare* directive is met in a global scope, then data lifetime region is from program execution start to program execution end.

Declare directive has the following syntax:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma acc declare
                    [copy(var-list) ]*
                    [copyin(var-list) ]*
                    [copyout(var-list) ]*
                    [create(var-list) ]*
                    [present(var-list) ]*
                    [present_or_copy(var-list) ]*
                    [present_or_copyin(var-list) ]*
                    [present_or_copyout(var-list) ]*
                    [present_or_create(var-list) ]*
                    [deviceptr(var-list) ]*
                    [device_resident(var-list) ]*
                    [link(var-list) ]*
```

device_resident clause may be used to specify that named variables on the list should be allocated in device memory only

6.4.4 update directive

When using a HWA, an important bottleneck is often the data transfer between the HWA memory and the host memory. To limit the communication overhead, the programmer can try to overlap data transfers with computations by using the asynchronous property of the HWA.
Update directive is used to transfer partial or complete values of variables between host and device memory.

It has the following syntax:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma acc update    [if(condition)]
                      [async[(handle-id)]?]
                      [wait[(handle-id-list)]?]
                      [self(var-list)]*
                      [host(var-list)]*
                      [device(var-list)]*
                      [device_type(device-name-list)]*
```

Where:

- *device* clause specifies that values of device version of variables on the list should be updated to correspond to host values
- *self* clause is a synonym for *device* clause
- *host* clause specifies that values of host versions of variables on the list should be updated to correspond to device values

6.4.5 Subarrays in OpenACC

A subarray is a selected portion of an array. It designates a set of elements from an array.

The subarrays can be used in order to optimize data transfers between the host and the HWA in some cases where it is not necessary to transfer the whole array.

Subarrays may serve as an argument for most of data-related clauses.

Syntax of subarray specification is different for C/C++ and Fortran languages.

In C/C++ it takes the following form:

```
array_name([start-index?:length?])+
```

Where *start-index* is a start index of array subscript and *length* is a length of array subscript along corresponding dimension.

If *start-index* is skipped, it is assumed to be 0. If *length* is skipped, then array size along corresponding dimension must be deducible from array declaration.

In Fortran it takes the following form:

```
array_name(low-bound?:high-bound])+
```

Array subscript usage example:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
int A[10][20][20];

int func() {
  int x;
  #pragma acc parallel loop copyin(A[0:1][0:20][0:20]) reduction(+:x) collapse(2)
  for (int i = 0; i < 20; i++) {
    for (int j = 0; j < 20; j++) {
      x = A[0][i][j] * A[1][i][j];
    }
  }
  return x;
} // TODO: does not work because of collapse
```

Important subarray restriction is that it should occupy contiguous chunk of memory with the exception of dynamically-allocated array dimenstion is C:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
void func() {
  int A[10][20][20];
  int x;
  // inorrect - copied data is not contiguous!!!
  #pragma acc parallel loop copyin(A[0:1][0:10][0:10]) reduction(+:x) collapse(2)
  for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
      x = A[0][i][j] * A[1][i][j];
    }
  }
}
```

6.5 Loop directive

OpenACC directive applies to a loop immediately following (or a loop nest) it and instructs compiler how to parallelize it.

One can use gang, worker and vector clauses to describe how loop (or loop nest) iteration should be mapped to different levels of parallelism.

The syntax of loop directive is as follows:

```
#pragma acc loop
                    [collapse(num)]
                    [gang(gang-arg-list) ]
                    [worker(num)]
                    [vector(num)]
                    [seq]
                    [auto]
                    [tile(tile-spec)]
                    [device_type(device-name-list)]
                    [independent]
                    [private(var-list)]
                    [reduction(operator:var-list)]*
```

Private clause specifies that a private copy of each specified variable should be created for each thread that executes loop iteration.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma acc parallel loop
for (i=0; i < n; i++) {
  int tmp = i * 10;
  #pragma acc loop private(tmp)
  for (j=0; j < n; j++) {
     tmp = tmp * a[i][j]; // private copy of tmp variable is created
     b[i][j] += tmp;
  }
} // TBD: strange warning (private.c)
```

Most of the specified loop clauses are used to tune loop implementation performance and are described in section 11. of current document.

6.6 Asynchronous execution

The main intent of asynchronous execution model in OpenACC is support for overlapping of computation and data transfers, but also for overlapping of host and device code.

Async clause may be specified for *parallel* and *kernels regions, enter data, exit data* and *update* directives.

If a handle-id is specified as *async* clause argument then corresponding kernel execution or data transfer is associated with a stream of actions. All actions within a stream are executed sequentially in a FIFO manner, but may overlap with actions from other streams.

Explicit synchronization with asynchronously lanuched action may be achieved by means of *wait* clause (applied to all directives mentioned above) or *wait* directive.

Wait clause without specific handle-id forces current thread to wait for completion of all actions asynchronously launched by this thread, whereas *wait* clause with a list of handle-ids waits only for completion of all currently scheduled actions in corresponding streams.

Wait directive has the following syntax

```
#pragma acc wait [,async(handle-ids)]
```

Its semantics is close to the semantic of *wait* clause; please refer to OpenACC 2.0 specification for the exact semantics.

The example below illustrates use of overlapping of data transfers and computations. We divide input arrays on a number of chunks and associate asynchronous stream with each chunk. Suppose that HWA device cannot execute more than one kernel at a time, then we will effectively overlap computation for chunk **i** with output data transfer for chunk **i - 1** and input data transfer for chunk **i + 1.**

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

async clause usage

```
void func(int *A, int *B, int size) {

  # pragma acc data create(A[0:size], B[0:size])
  for (int i = 0; i < size / CHUNKSIZE; i++) {
    # pragma acc update device(B[CHUNKSIZE * i:CHUNKSIZE]) async(i)
    # pragma acc update device(A[CHUNKSIZE * i:CHUNKSIZE]) async(i)
    # pragma acc parallel loop async(i)
    for (int j = CHUNKSIZE * i ; j < CHUNKSIZE * (i + 1); j++) {
      A[j] = B[j] + A[j];
    }
    # pragma acc update host (A[CHUNKSIZE * i:CHUNKSIZE]) async(i)
  }
} ++
```

OpenACC async/wait clauses may be also used for construction of tree-like dependencies between tasks.

6.7 Procedure calls

Procedure may be marked for execution on the HWA using *routine* directive — in that case both host and device versions of code are generated.

When *routine* directive is used in a file containing procedure call, it is used by implementation in order to determine procedure call attributes.

Intended level of parallelism should be specified for routine, which can be one of: gang, worker, vector or sequential.

Calls to procedure from a parallelization context of the same or lower level are prohibited, i.e. call to gang-parallel routine may not be made from a loop explicitly marked as gang-paralell:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Invalid routine call

```
#pragma acc routine gang
void func(int *A, int *B, int size) {
  #pragma acc loop
  for (int j = 0; j < size; j++) {
     A[j] = B[j] + A[j] * 2;
  }
}
void caller(int **A, int **B, int size) {
  #pragma acc parallel loop gang
  for (int i = 0; i < size; i++) {
      func(A[i], B[i], size);
  }
}
```

The example above may be fixed by changing routine parallelization level from *gang* to *worker, vector* or seq.

Valid routine call example

```
#pragma acc routine seq
int scmul(int A[2], int B[2]) {
  int result = 0;
  for (int i = 0; i < size; i++) { // each device thread executes the whole loop
    result += A[i] * B[i];
  }
  return result;
}

void caller(int ***A, int ***B, int **C, int size) {
  #pragma acc parallel loop gang
  for (int i = 0; i < size; i++) {
    #pragma acc parallel loop vector
    for (int j = 0; j < size; j++) {
       C[i][j] = scmul(A[i][j], B[i][j]);
    }
  }
}
```

Note: device routines are not supported in the current release of PathScale ENZO solution, it will supported in future releases.

6.8 Nested parallelism

OpenACC specification allows nested accelerated regions *(kernels* or *parallel).* This syntax construct represents ability of some HWA devices to dynamically generate and launch kernels.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Nested parallelism is not fully supported in the current release of PathScale ENZO solution, but partial support exists.

6.9 Atomic operations

OpenACC supports atomic access to variables via *atomic* directive.

It may be applied either to a single statement or a pair of statements of special kind.

Atomic directive have the following syntax:

```
#pragma acc atomic  [atomic-clause]
```

Where *atomic-clause* is one of the following list: read, write, *update, capture.*

Below is the example of atomic directive use:

```
#pragma acc parallel loop
for (int i = 0; i < n; i++)
  #pragma acc atomic update
    a++;
}
```

Please refer to OpenACC specification for the complete list of restrictions for atomic statement blocks and exact semantics of clauses.

Note: atomic operations are not supported in the current release of PathScale ENZO solution, it will supported in future releases.

7 OpenMP directives

7.1 Introduction

The OpenMP directives may be seen as “meta-information” added in the application source code. They are safe meta-information i.e. they do not change the original code. They address parallelization of code regions, offloading code regions to accelerator devices and transfers of data to/from the HWA memory. We will be concentrated on host-device OpenMP model in this document.

Below is the simple example of vector multiplication offloaded to HWA:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma omp target map(to: v1, v2) map(from: v3)
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  v3[i] = v1[i] * v2[i]
}
```

First OpenMP directive marks code for offloading and defines copy-in/copy-out properties for arrays.
Arrays **v1** and **v2** will be copied from host memory to device memory upon region start and array **v3** will be copied from device to host upon region finish.

Second OpenMP directive is a composition of *parallel* and *for* directives. *Parallel* directive creates a team of device threads and starts parallel execution of code region and *for* directive instructs OpenMP runtime to distribute loop iterations between device threads.

7.2 Syntax of OpenMP directives

The general syntax of the OpenMP directives is:

- For C and C++

```
#pragma omp directive_type [, directive_clauses]*
```

- For Fortran 95, 2003 and 2008

```
!$omp directive_type [, directive_clauses]*
```

Where:

- directive_type: is the type of the directive;
- directive_clauses: designates some parameters associated to the directive_type. These parameters may be of different kinds and specify either some arguments given to the directive either a mode of execution (asynchronous versus synchronous for example);

Furthermore, the clauses may accept arguments. Typically these arguments are some integer values specifying number of parallel execution lanes, etc. or identifiers.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

OpenMP directive examples:

```
#pragma omp parallel
#pragma omp for reduction(*:x)
#pragma omp target update to(y[0:10])
```

It is also possible to create certain combinations of OpenMP directives as a shortcut for specifying them separately:

```
#pragma omp parallel for simd
```

Please refer to OpenMP specification for the complete list of allowed directive combinations.

In the remainder of this document, most examples of directives will be given in C. Fortran directives only differ by their prefix.

7.3 Parallel directive

Parallel directive starts parallel execution of a code region. Team of threads is created for its execution, with current thread becoming a *master* for the team.

The syntax of *parallel* directive is as follows:

```
#pragma omp parallel        [if(condition)]
                            [num_threads(threads-count)]
                            [proc_bind(master | close | thread)]
                            [reduction(operator:var-list)]*
                            [copyin(var-list)]*
                            [private(var-list)]*
                            [firstprivate(var-list)]*
                            [shared(var-list)]*
                            [default(shared | none)]
```

Where:

- *num_threads* specifies number of threads allocated for execution of region
- *if* clause specifies condition based on which runtime either starts parallel execution of the region or fallbacks to serial execution.
- *procbind* clause controls OpenMP thread affinity policy
- *private* clause instructs implementation to create a private copy of each variable on the list for each of threads allocated for region execution
- *firstprivate* clause behaves similar to private clause, additionaly each private variable is initialized with a value of the original variable
- *copyin* clause semantics is similar to firstprivate, although it can be used if variable was already defined as *threadprivate*

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

- *shared* clause declares one or more variable to be shared between team threads
- default() clause defines default data-sharing attributes of variables used in the region
  default(none) prohibits implicit inference of data attributes for any variable used in region
  default(shared) sets *shared* data attribute for variable referenced inside region
- *reduction* clause specifies reduction operator and the set of variables. Reduction result is avaiable on region exit.

```
#pragma omp parallel num_threads(4)
{
  // each thread executes parallel region statements
  printf("Hello from thread %d\n", omp_get_thread_num());
}
```

7.4 Worksharing constructs

Worksharing constructs may be used to explicitly distribute work items (such as indvidual statements or loop iterations) inside parallel region across team threads. By default, implicit barrier exists in the end of workshare region, but it is possible to cancel it using *nowait* clause.

7.4.1 Loop construct

Loop construct can be used to distribute loop (or loop nest) iterations across team threads.

Unlike OpenACC, there is a explicit barrier in the end of loop construct, unless nowait clause is specified.

The syntax of loop construct is as follows:

```
#pragma omp for             [private(var-list)]*
                            [firstprivate(var-list)]*
                            [lastprivate(var-list)]*
                            [reduction(operator:var-list)]*
                            [schedule(kind:chunk-size)]
                            [collapse(num)]
                            [ordered]
                            [nowait]
```

Where:

- *private* clause instructs implementation to create a private copy of each variable on the list for each of threads allocated for region execution
- *firstprivate* clause behaves similar to private clause, additionaly each private variable is initialized with a value of the original variable
- *lastprivate* clause behaves similar to private clause, additionaly each original variable is updated to have a value of private variable corresonding ot last loop iteration upon loop exit
- *reduction* clause specifies reduction operator and the set of variables. Reduction result is avaiable on region exit.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

- *schedule* clause specifies how loop iterations should be divided into chunks, which are distributed between threads. S*chedule kind* may be one of: *static, dynamic, guided* and *auto*
- *collapse* may be used to associate more than one loop nest with an OpenMP loop construct
- *ordered* clause may be used to force in-order execution of loop iterations
- *nowait* clause may be used to cancel implicit barrier in the end of loop construct

Source code loop to which the *loop* directive is applied should be of canonical form. In case of C/C++ it means that it should be *for* loop with index variable initializer, termination condition and increment statement of simple kinds. Please refer to OpenMP specification for the exact definition of canonical loop.

```
#pragma omp parallel for
for (int i = 0; i < n; i++) { // canonical OpenMP loop
  ..
}

#pragma omp parallel for
for (i = 0, j = 0; i + j < n; i++, j += i) { // non-canonical OpenMP loop
  ..
}
```

7.4.2 Sections construct

Sections directive is used to mark a structured block with several structural *section* sub-blocks.

Its syntax is as follows:

```
#pragma omp sections        [private(var-list)]*
                            [firstprivate(var-list)]*
                            [lastprivate(var-list)]*
                            [reduction(operator:var-list)]*
                            nowait
```

Individual sections are distributed between threads and may be executed in parallel, if implementation decides to do that.

Below is the example of sections construct usage:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
int i;

#pragma omp parallel sections private(i)
{   // individual sections may be executed concurrently
    #pragma omp section
    {
      for (i = 0; i < n ; i++) {
        c[i] = a[i] - b[i];
      }
    }
    #pragma omp section
    {
      for (i = 0; i < n ; i++) {
        d[i] = a[i] + b[i];
      }
    }
}
```

7.4.3 Single construct

Single directive marks a code region that must be executed by only a single thread of the current thread team.

Below is the example of *single* construct usage:

```
#pragma omp parallel
{
   foo(); // called by all threads

   #pragma omp single
    printf("Foo stage completed");
   // implicit barrier at that point

   bar(); // called by all threads
}
```

7.4.4 Workshare construct

Workshare construct allows distribution of units of work such as scalar assignments, array assignments, etc. across team threads. This construct is Fortran-specific and not covered in the current document. Please refer to OpenMP specification for details.

7.5 Device constructs

Current section covers OpenMP constructs related to HWA offloading, but not related to HWA data management and transfers.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

7.5.1 Target directive

Target directive offloads region to HWA:

```
#pragma omp target          [device(num)]
                            [map(map-type:var-list)]*
                            [if(condition)]
```

Where:

- *device* specifies id of device targeted for region execution
- *map* clause controls data transfers between host and device memory
- *if* clause is used to specify condition, based on which either host- or device- version of region is executed

Example below shows code region targeted for HWA offloading. Mapping attributes are explictily specified for the region: v1 and v2 are copied-in upon region start and v3 is copied-out upon region end. Region is offloaded to HWA only if number of loop iterations is larger than threshold 100.
Note that, unlike OpenACC, implicit barrier is inserted in the end of parallel loops, so the code below is correct.

```
#pragma omp target map(to: v1[:n], v2[:n]) map(from: v3[:n]) if (n > 100)
{
  #pragma omp parallel for
  for (int i = 0; i < n; i++) {
    v1[i] = v1[i] * v2[i]
  } // implicit barrier
  #pragma omp parallel for
  for (int i = 0; i < n; i++) {
    v3[i] = v1[i] + v2[i]
  }
}
```

7.5.2 Teams directive

Teams directive introduces second level of thread grouping — team leagues. Master thread of each team executes the region:

```
#pragma omp teams           [num_teams(num)]
                            [thread_limit(num)]
                            [reduction(operator:var-list)]*
                            [private(var-list)]*
                            [firstprivate(var-list)]*
                            [shared(var-list)]*
                            [default(shared | none)]
```

Where:

- *num_teams* specifies number of teams in a league
- *thread_limit* — maximal number of threads that runtime may allocate for a team

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

- *private* clause instructs implementation to create a private copy of each variable on the list for each of teams allocated for region execution
- *firstprivate* clause behaves similar to private clause, additionaly each private variable is initialized with a value of the original variable
- reduction clause specifies reduction operator and the set of variables. Reduction result is avaiable on region exit.
- *shared* clause declares one or more variable to be shared between teams
- default() clause defines default data-sharing attributes of variables used in the region

7.5.3 Distribute directive

Distribute directive is used to distribute loop iterations between thread teams within a league:

```
#pragma omp distribute      [collapse(num)]
                            [private(var-list)]*
                            [firstprivate(var-list)]*
                            [dist_schedule(static, [chunk-size])]
```

Where:

- *collapse(n)* clause specifies that *distribute* directive to a loop nest of n loops
- *private* clause declares variable on the list to be private for each thread team
- *firstprivate* clause inherits semantics of *private* clause and also initializes private variables with the value of original variable
- *dist_schedule* defines distribution of consecutive iteration chunks across thread teams.
  Only the static distribution is allowed in current version of OMP standard.

7.6 Data allocation and transfers

7.6.1 Target data directive

Target data directive creates host data environment, allocates variables in host memory and defines host-device data transfer policies.

Its syntax is as follows:

```
#pragma omp target data     [device(num)]
                            [map(map-type:var-list)]*
                            [if(condition)]
```

It has the same the set of clauses as *target* directive, with same semantics. It is useful to create data regions that span across multiple target regions, to avoid excessive device-host-device data transfers.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma omp target data map(to:A[:n])
{
    #pragma omp target map(to:B[:n])
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
       A[i] = A[i] * i + B[i];
    }
    // A is resident in device memory
    #pragma omp target map(from:C[:n])
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
       C[i] = A[i] - i;
    }
}
```

7.6.2 Target update directive

Target data directive may be used to synchronize values of device and host versions of variable.

Its syntax is as follows:

```
#pragma omp target update  [device(num)]
                           [to(var-list)]*
                           [from(var-list)]*
                           [if(condition)]
```

Where:

- *device* specifies id of device targeted for region execution
- *to* clause specifies a list of variables that should be transferred from host to device
- *from* clause specifies a list of variables that should be transferred from deivce to host
- *if* clause is used to specify condition, based on which either host- or device- version of region is executed

Example below shows how *target update* may be used to update existing device image of host array fragment:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma omp target data map(to:A[:n]) map(to:C[:n])
{
    // First target section
    #pragma omp target
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
       A[i] = A[i] * i + C[i];
    }
    // Update host version of C
    update_array_range_with_new_values(C, 0, n / 2);
    // Update low half of device version of C
    #pragma omp target update to(C[:n/2])
    // Second target section
    #pragma omp target map(from:B[:n])
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
       B[i] = A[i] - C[i];
    }
}
```

7.6.3 Declare target directive

Declare target directive may be used to create device variables directly in device memory and declare functions as callable on device.

ts syntax is as follows:

```
#pragma omp declare target
  …
  < variable and function declarations >
  …
#pragma omp end declare target
```

Example below shows how *declare target* directive can be used:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma omp declare target
int norm(int x[2]) { // Function defined and called on device
   return x[0] * x[0] + x[1] * x[1];
}
#pragma omp end declare target

...
{
  #pragma omp target
  #pragma omp parallel for
  for (int i = 0; i < n; i++) {
    nrm[i] = norm(x[i]);
  }
}
```

7.6.4 Threadprivate directive

Threadprivate directive is used to declare variables as private for each OMP thread.

Its syntax is as follows:

```
#pragma omp threadprivate(var-list)
```

7.6.5 OpenMP array subscripts

OpenMP data allocation and transfer directives allow references to array fragments.
Fortran source language has embedded array subscripting. Most common generic C/C++ OpenMP single dimension array subscript has a form of

[**lower-bound : length**]

Length may be omitted if array size is known. If lower-bound is omitted, its value is implied to be zero.
For multi-dimensional C/C++ its only allowed to specify subscript for the first dimension.

Example below shows an example of array subscript usage. By using subscript we copy only low halves of arrays to device, instead of copying full arrays.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
int A[20];
int B[20];

int func() {
  int x;
  #pragma omp target map(to:A[5:3]) map(to:B[5:3])
  #pragma omp parallel for reduction(+:x)
  for (int i = 5; i < 8; i++) {
    x += A[i] * B[i];
  }
  return x;
}
```

7.7 Tasking constructs

7.7.1 task construct

When an OpenMP thread is allocated for parallel region execution, implicit task is associated with it.
When an application logic requires creation of asyncrhonous computations and specifying dependencies between them, it is useful to employ OpenMP explicit tasks.

The syntax of task directive is as follows:

```
#pragma omp task            [if(cond)]
                            [final(cond)]
                            [untied]
                            [default(shared | none)]
                            [mergeable]
                            [private(var-list)]*
                            [firstprivate(var-list)]*
                            [shared(var-list)]*
                            [depend([in | out | inout], var-list)]*
```

Where:

- *if* clause specifies condition based on which either a new asynchronous task will be lauched, or current thread will synchronously execute code in the task region.
- *final* clause specifies condition for making the task final. Final task lauches all sub-tasks synchronously.
- *untied* clause marks the task is untied — which means that task may be resumed by a thread different from the initial one
- *default* clause sets default data sharing attribute for task region variables
- *private, shared, firstprivate* clauses set different data sharing attributes
- *mergeable* clause denotes that a task if it is to be executed synchronously by runtime may be merged in the enclosing task context — i.e. data enviroment may be inherited instead of creating a new one
- *depend* clause defines input and output dependencies for task, associated with source code variables

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

In the example below we launch three tasks asynchronously. First two tasks use HWA device to compute value of *a* and *b.* Third task will only be lauched when both *a* and *b* compuation is finished.

```
  #pragma omp task depend(out: a)
  {
      #pragma omp target
      #pragma omp parallel for reduction(+:a)
      for (int i = 0 ; i < n; i++) {
        a += somefunc1(i);
      }
  }

  #pragma omp task depend(out: b)
  {
      #pragma omp target
      #pragma omp parallel for reduction(+:b)
      for (int i = 0 ; i < n; i++) {
        b += somefunc2(i);
      }
  }

  #pragma omp task depend(in: a, b)
  {
      c = a + b;
      printf("Asynchronous computation finished with result: %d\n");
  }
// TBD: fails when "omp target" removed
```

7.7.2 taskyield directive

Taskyield directive specifies that current task may be suspended, and other task assigned to the current thread instead.

```
#pragma omp taskyield
```

7.7.3 taskwait directive

Taskwait directive specifies that current task should be suspended until all sub-tasks execution is completed.

```
#pragma omp taskwait
```

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
for (int i = 0; i < n; i++) {
  #pragma omp task
  somefunc(i);
}

// wait for all somefunc() calls to complete
#pragma omp taskwait
```

7.7.4 taskgroup directive

Taskwait directive specifies that current task should be suspended until all execution of all sub-tasks and their descendant tasks is completed.

```
#pragma omp taskgroup
```

7.8 Synchronization

7.8.1 master construct

Master directive is applied to a structued block, that must be executed only by master thread of the current team. No barrier on the beginning or in the end of the block is assumed.

Syntax of this directive is as follows:

```
#pragma omp master
```

Master construct may be used for asynchronous jobs, such as logging:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < n; i++) {
    #pragma omp for
    for (int j = 0; j < m; j++) {
      C[i][j] = A[i][j] * B[i][j];
    }
    #pragma omp master
    logger.log("Iteration %d finished", i);
  }
}
```

7.8.2 flush construct

OpenMP flush directive is used to flush all cached version of shared variables for a current thread to shared memory. It also invalidates all variable images privately cached by the thread.

```
#pragma omp flush(var-list)
```

7.8.3 critical construct

Critical directive applied to a structural block creates critical section. Exclusive access is guaranteed within thread contention group, which is defined as a group of threads allocated for current parallel region execution.

```
#pragma omp critical [name]?
```

Optional name may be specified for the *critical* construct. Semantics of name assigment assumes that no two critical sections with same name may be executed concurrently.

7.8.4 barrier construct

Barrier directive may be used to enforce synchronization for threads within a team.

```
#pragma omp barrier
```

All threads must execute barrier before any of them may continue execution beyond barrier. *Barrier* directive semantics assumes implicit *flush.*

Below is the example of barrier usage:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma omp parallel shared(A, B) private(id, i) num_threads(N)
{
   id = omp_get_thread_num();
   A[id] = somefunc(id);
   // wait for A computation to finish
   #pragma omp barrier
   #pragma omp for
   for (i = 0; i < N; i++) {
     B[i] = somefunc2(i, A);
   }
}
```

7.8.5 atomic construct

Atomic construct allows to express atomic reads, writes and updates of shared variable values.

Atomic directive syntax is as follows:

```
#pragma omp atomic [ read | write | update | capture ] [seq_cst]?
```

It may be followed by expression of certain kind or structured block (in case when *capture* clause is specified).

Please refer to OpenMP documentation for the details fo *atomic* directive semantics and usage.

7.8.6 ordered construct

Ordered construct may be used to specify that a structured block inside a region must be executed in-order by concurrently executing loop iterations, as if the whole loop was not parallelized.

```
#pragma omp ordered
```

It is only allowed to have a single *ordered* block within a parallel loop.

7.9 SIMD

7.9.1 simd directive

Simd directive is used to specify that loop iterations may be executed concurrently by means of SIMD instruction of target processor.

Its syntax is as follows:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma omp simd          [safelen(num)]
                          [linear(var-list[:num])]*
                          [aligned(var-list[:num])]*
                          [private(var-list)]*
                          [lastprivate(var-list)]*
                          [reduction(operator:var-list)]*
                          [collapse(num)]
```

Where:

- *safelen* clause specifies maximal distance between loop iterations that may be safely executed in parallel
- *linear* clause creates private variables with value for each iteration **i** being value of original variable plus **i** times *num.*
- *aligned* clause is used to specify array alignment
- *private* clause declares variable on the list to be private for each simd lane
- *firstprivate* clause inherits semantics of *private* clause and also initializes private variables with the value of original variable
- *reduction* clause specifies reduction operator and variables on which reduction must be performed.
- *collapse(n)* clause specifies that *simd* directive applies to a loop nest of n loops

Below is the example of *simd* directive usage:

```
#pragma omp target
#pragma omp parallel
// distribute loop iterations between simd lanes of single
// device thread
#pragma omp simd safelen(4)
for (int i = 4; i < n; i++) {
  v3[i] = v1[i] * v2[i] + v3[i - 4];
}
```

Note: *safelen* clause is not supported in the current release of PathScale ENZO solution, it will supported in future releases.

7.9.2 declare simd directive

Declare simd directive is used to declare functions and procedures as callable from device simd execution context.

Its syntax is as follows:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma omp declare simd  [simdlen(num)]
                          [linear(arg-list[:num])]*
                          [aligned(arg-list[:num])]*
                          [uniform(arg-list)]*
                          [inbranch]
                          [notinbranch]
```

Where:

- *simdlen* clause specifies simd vector length for the function
- *linear* clause creates private variables with value for each iteration **i** being value of original argument plus **i** times *num.*
- *aligned* clause is used to specify array alignment
- *uniform* clause specifies that values of arguments on the list are invariant for all simd lanes of enclosing call context
- *firstprivate* clause inherits semantics of *private* clause and also initializes private variables with the value of original variable
- *inbranch* clause specifies that function will always be called from a conditional statement inside *simd* loop
- *notinbranch* clause specifies that function will never be called from a conditional statement inside *simd* loop

7.10 Cancellation

Cancel directive may be used to abort execution of parallel region or group of tasks.

Its syntax is as follows:

```
#pragma omp simd [ parallel | sections | for | taskgroup ] [if(cond)]?
```

First clause defines cancellation scope and the optional second clause — condition for cancellation to be fired.

Threads and tasks execution is not immediately aborted when cancel directive is encountered. There are several cancellation points — implicit and explicit barriers and cancel directives — at which cancellation of the parallel region or task group is being checked. It is also possible to define custom cancellation points using *cancellation point* directive:

```
#pragma omp cancellation point
```

Bellow is the example of *cancel* directive usage:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma omp parallel
{
  try {
    do_something_dangerous();
  }
  catch(...) {
    #pragma omp cancel parallel
  }

  // if exception occured, no thread
  // will start execution of the loop below
  #pragma omp for
  for (int i = 0; i < n; i++) {
    do_something_safe(i);
  }
} // TBD: aborts
```

8 Supported Languages

Following language limitations are listed in OpenACC documentation:

- A program may not branch into or out of an OpenACC parallel construct.
- In C and C++, function static variables are not supported in functions to which a routine directive applies.
- In Fortran, variables with the save attribute, either explicitly or implicitly, are not supported in subprograms to which a routine directive applies.

9 Compiling OpenACC and OpenMP Applications

PathScale ENZO provides developers with OpenACC/OpenMP standards compliant compilers in order to easily build ENZO applications. PathScale ENZO currently comes with OpenACC/OpenMP Fortran, C and C++ compilers.

9.1 Overview

In terms of use, PathScale ENZO works the same as the EKOPath compilers. However, the paths diverge at the final code generation phase.

Compiling an ENZO program is as simple as using the traditional EKOPath pathf90, pathcc or pathCC compiler drivers.

OpenACC support is enabled by -acc command line option. OpenMP support is enabled by -mp command line option.
Below are compiler invocation examples:

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
// compile C code with OpenACC support using NVidia Kepler as accelerator device
pathcc -acc -device=kepler test.c

// compile C++ code with OpenACC support using NVidia Kepler as accelerator device
pathCC -acc -device=kepler test.cpp

// compile Fortran code with OpenACC support using NVidia Kepler as accelerator device
pathf95 -acc -device=kepler test.f

// compile C code with OpenMP support using NVidia Kepler as accelerator  device
pathcc -mp -device=kepler test.c

// compile C++ code with OpenMP support using NVidia Kepler as accelerator  device
pathCC -mp -device=kepler test.cpp

// compile Fortran code with OpenMP support using NVidia Kepler as accelerator  device
pathf95 -mp -device=kepler test.f
```

Please reference the ENZO_cli_guide for the full list of supported targets.

9.2 Common Command Line Parameters

We strive to make the ENZO compiler as easy to use as possible, but for more details on compiler options please reference ENZO_cli_guide.pdf

10 Running OpenACC and OpenMP Applications

The execution of ENZO applications using the ENZO runtime library works just as regular applications.

10.1 Launching the Application

OpenACC and OpenMP programs using the ENZO runtime are launched just like regular programs

```
$ ./program
```

11 Improved code generation and performance

Most of the transformations described in this part apply on loops. A loop is a syntactic language construction expressing the repetition of some statements.

“for” loops in C language and “DO” loops in Fortran are supported.

To optimize the code generated by ENZO, two main types of directives are used:

- Some specifying loop properties;

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

- Others mentioning transformations to be applied on the loops.

Please note that ENZO does not check for all incorrect usages of the directives. Be aware that misuse of the OpenACC/OpenMP directives may lead to erroneous results.

11.1 Loops performance optimizations

The clauses and directives described in this part allow to specify some properties of loops and desired transformations that should be applied by compiler. These properties are then used by the code generator in order to optimize the generated code.

11.1.1 OpenACC kernels region

Unlike OpenMP/OpenACC *parallel* region, OpenACC kernels region serves not as a directive for compiler to parallelize certain code section, but as a hint that certain code section should be considered by the compiler for parallelization/offloading. In this section we consider two loop clauses, that may help to tune kernels region performance.

11.1.1.1 OpenACC independent clause

This clause has to be used when the code generator is not able to tell that loop iteration are data-independent.

It is typically used only in context of *kernels* region, since all explicitly marked *loops* in *parallel* region are implied to have data-independent iterations.

A parallel loop is declared using the following directive:

```
#pragma acc kernels loop independent
for (i = 0; i < n; i++)
    X[i] = Y[i] + Z[i];
```

11.1.1.2 OpenACC seq clause

A non-parallel loop (i.e. sequential) is declared using the following directive:

```
#pragma acc loop seq
```

The following example shows a loop nest where the use of the OpenACC directives allows guiding the code generation.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

explicit sequential clause usage

```
#pragma acc kernels loop seq
for (i=0; i < n; i++) { // execute iteration of this loop redundantly by all threads
#pragma acc loop independent
  for (j=0; j < n; j++) {
    D[i][j] = A[i][j] * E[3][j];
  }
}
```

This directive proves to be useful to control the gridification process of loops.

Note that this directive forces ENZO to consider the loop as sequential independently of any optimization analysis.

11.1.2 reduction

The *reduction* clause allows the user to indicate that one or several reductions are done in the loop or parallel region. Indeed, without this clause, the parallel execution of a loop with such an operation could lead to a wrong result.

The *reduction* clause has the following syntax:

```
reduction(operator:vars)
```

- operator: specifies a reduction operator (see Table 9)
- vars: names of a scalar variables referenced in the loop;

The table below summarize the list of allowed builtin reduction operators in the *reduction* clause for OpenACC and OpenMP.

Table 9 - List of reduction operators defined in OpenACC

C/C++ Operator	Fortran Operator	Initial value	Meaning
+	+	0	Addition
*	*	1	Multiplication
-	-	0	Substraction (OpenMP only)
min	min	least	Minimum
max	max	largest	Maximum
&&	.and.	1	Logical and
||	.or.	0	Logical or
^	ieor	0	bitwise exclusive or
|	ior	0	bitwise

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

			inclusive or
&	iand	~0	bitwise and
	.eqv.	.true.	eqvuivalence
	.neqv.	.false.	not-eqvuivalence

OpenACC reduction clause may be applied to parallel, kernels and loop directives.

OpenMP reduction clause may be applied to parallel, loop, sections, simd, teams directives.

OpenMP allows definition of custom reduction operators:

```
#pragma omp declare reduction(operator:typename-list:combiner) [initializer(expr)]
```

Below is the example of custom **merge** reduction operator declaration, that may be used to merge lists of integer numbers. **omp_out** and **omp_in** are special symbols, denoting variable used to cumulate reduction result and individual value used in cumulation.

```
#pragma omp declare reduction(merge:
                              std::list<int>:
                              omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()
                              )
```

OpenACC/OpenMP loop reduction

```
#pragma acc parallel
#pragma acc loop reduction(+:ssx,ssy)
  for ( i = 0; i < NK; i++)
  {
    if (qqprim2[i])
      {
        qq[qqprim[i]] += 1.0;
        ssx = ssx + qqprim3[i];
        ssy = ssy + qqprim4[i];
     }
  }
```

```
#pragma omp parallel
#pragma omp for reduction(+:ssx,ssy)
  for ( i = 0; i < NK; i++)
  {
    if (qqprim2[i])
      {
        qq[qqprim[i]] += 1.0;
        ssx = ssx + qqprim3[i];
        ssy = ssy + qqprim4[i];
     }
  }
```

Code fragment above illustrates the use of the OpenACC/OpenMP parallel directive with two addition (i.e. +) reduction operations.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Note that when reduction clause is specified for OpenACC *parallel* directive, a private copy of reduction variable is created for each gang. Reduction result is available on region exit.

When reduction clause is used with OpenACC *loop* directive, private copy is created for each thread implementing loop iterations. Result aviability depends on loop parallelization level and reduction variable properties. If loops is parallelized on level lower than *gang* and reduction variable is defined as gang-private, then reduction result is avaiable on loop exit, otherwise it is avialable on *parallel* or *kernels* region exit.

Thus, the following usage of *reduction* clause is incorrect:

Invalid usage of OpenACC parallel clause with reduction operations

```
#pragma acc parallel
{
  #pragma acc loop reduction(+:sum)
  for ( i = 0; i < NK; i++) {
     sum = a[i] + b[i]
  }
  #pragma acc loop
  for (i = 0; i < NK; i++) { // sum does not hold desired value at that point
     a[i] = a[i] / sum
  }
}
```

11.1.3 OpenACC tile clause

Tile clause can be used to instruct compiler to convert one more loops (tightly nested) into tiled form — each loop will be replaced with outer loop iterating across tiles and inner loop iterating across iterations in a tile.

This transformation allows to improve data access locality.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Tiled matrix multiplication

```
#pragma acc parallel loop
for (int i=0; i < n; i++) {
  #pragma acc loop tile(8,8)
  for (int j=0; j < n; j++) {
    for (int k=0; k < n; k++) {
       c[i][j] += a[i][k] * b[k][j];
    }
  }
}

// Compiler may transform it to something like the code below (assume n % 8 == 0)

#pragma acc parallel loop
for (int i=0; i < n; i++) {
  #pragma acc loop
  for (int t1=0; t1 < n / 8; t1++) {
    #pragma acc loop
    for (int t2=0; t2 < n / 8; t2++) {
      #pragma acc loop
      for (int j=0; j < 8; j++) {
        #pragma acc loop
        for (int k=0; k < 8; k++) {
          c[i][t1 * 8 + j] += a[i][t2 * 8 + k] * b[t2 * 8 + k][t1 * 8 +j];
        }
      }
    }
  }
}
```

Note: tiling is not supported in the current release of PathScale ENZO solution, it will supported in future releases.

11.1.4 Levels of parallelism

11.1.4.1 OpenACC levels of parallelism

OpenACC defines three levels of parallelism — gangs, workers and vector lanes.

Roughly speaking, they can be associated with CUDA thread_blocks, warps and threads.

When declaring *parallel* region number of gangs, number of workers in each gang and number of vector lanes in each vector may be specified using *num_gangs, num_workers* and *vector_length* clauses; for the *kernels* region compiler infers optimal parameters based on source code analysis.

num_gangs clause specify how many gangs will execute the region and thus will always affect the execution. *num_workers* and *vector_length* clauses control execution only for loops enabling worker- or vector-level parallelism (either explicitly forced by user or as a result of compiler auto-vectorization).

These execution parameters may be defined as device-specific using *device_type clause.*

By default compiler automatically distributes loop (or loop nest) itertations between gangs, workers and vector lanes.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Note that the optimal value of gridification parameters is dependent on the loop nest and on the targeted hardware.

gang, worker and *vector* clauses of *loop* directive may be used to explicitly distribute loop iterations across specific levels of parallelism.

If no explicit clauses are specified, then the compiler will make a decision according to what level of parallelism is available — for example if top level loop is marked with *gang* clause, then nested loop iterations will be distributed across workers and/or vector lanes.

In case of *parallel* region, implementation will used either explicitly specified or inferred dimension sizes for each level.
In case of k*ernels* region it is allowed to specify arguments for *gang, worker* and *vector* clauses. *Worker* and *vector clause* arguments specify number of workers in gang and vector length respectively. *Gang* clause arguments may be used to specify both gangs count (*num* argument) and chunks size (*static* argument*).*

11.1.4.2 OpenMP levels of parallelism

OpenMP defines three levels of parallelism — teams, threads and simd lanes.

Roughly speaking, they can be associated with CUDA thread_blocks, warps and threads.

Default OpenMP entity capable of code execution is a thread. Team of threads is allocated when parallel region is encountered, their number may be controlled by *num_threads* clause.

OpenMP *loop* construct is used to distribute loop iterations across individual threads.

In a host-device execution context (introduced by *target* directive) it is possible to create league of thread teams, by means of *teams* directive; *num_teams* clause is used to specify number of teams in the league.

OpenMP *distribute* construct is used to distribute loop iterations across teams in a league.

On the fine-grained level, it is possible to distribute loop iterations across lanes of SIMD instructions using *simd* directive.

11.1.4.3 Levels of parallelism: examples

OpenACC	OpenMP

`//Loops gridification will be chosen by implementation`
`#pragma acc parallel loop`
`for (i=0; i < n; i++) {`
`  #pragma acc loop`
`  for (j=0; j < n; j++) {`
`  ...`
`  }`
`}`
	`// Team-level gridification`
	`#pragma omp parallel for`
	`for (i=0; i < n; i++) {`
	`  // Team will be created`
	`  // for each thread`
	`  #pragma omp for`
	`  for (j=0; j < n; j++) {`
	`  ...`
	`  }`
	`}`

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
//Loop will be gridified with vector
//length of 8 and 8 workers in a gang
#pragma acc parallel loop worker vector
num_workers(8) vector_length(8)
for (i=0; i < n; i++) {
  ...
}
```

```
//Loop will be gridified with 8
threads //in a team and safe SIMD lenght
of 8
#pragma omp parallel for simd
num_threads(8) safelen(8)
for (i=0; i < n; i++) {
  ...
}
```

```
#pragma acc parallel num_gangs(256)
// distribute across gangs
#pragma acc loop gang
for (int i = 0; i < n; i++) {
  // distribure across workers and
     vectors
  #pragma acc loop worker vector
  for (int j = 0; j < n; j++) {
    // execute sequentially
    for (int k = 0; k < n; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

```
#pragma omp target
#pragma omp teams num_teams(256)
// distribute across gangs
#pragma omp distribute
for (int i = 0; i < n; i++) {
  // distribure across workers and
  // vectors
  #pragma omp parallel for simd
  for (int j = 0; j < n; j++) {
    // execute sequentially
    for (int k = 0; k < n; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

11.1.5 Collapse clause

Collapse clause can be used to apply loop directive to multiple tightly nested loops. Integer argument of the clause defines how many of tightly nested loops to collapse — i.e. set of their iterations will be linearized and distributed across specified levels of parallelism.

This clause may be used accompanying to *gang, worker* and *vector OpenACC* clauses and with *distribute, for, simd* OpenMP directives to apply parallelism level specification to multiple tightly nested loops— to hint compiler that a group of tightly nested loops should be scheduled on the same level(-s).

OpenACC	OpenMP

`#pragma acc kernels`
`// distribute iterations of "i" and "j" loops across 256 gangs`
`#pragma acc loop collapse(2) gang(256)`
`for (int i = 0; i < n; i++) {`
`  for (int j = 0; j < n; j++) {`
	`#pragma omp target`
	`#pragma omp teams num_teams(256)`
	`// distribute iterations of "i" and "j" loops across 256 teams`
	`#pragma omp distribute collapse(2)`
	`for (int i = 0; i < n; i++) {`

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
    // distribute iterations across
    // workers in each gang
    #pragma acc loop worker
    for (int k = 0; k < n; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

```
  for (int j = 0; j < n; j++) {
  // distribute iterations across
  // team threads
  #pragma omp parallel for
    for (int k = 0; k < n; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

```
// distribute all iterations of loop
nest across workers and vector lanes
// inside single gang (~ CUDA
threadblock)
#pragma acc parallel loop worker vector
#pragma acc collapse(3)
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

```
// distribute all iterations of loop
nest across threads and simd lanes
// inside single thread team (~ CUDA
threadblock)
#pragma omp target
#pragma omp parallel for simd
#pragma omp collapse(3)
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

11.2 OpenACC Cache management

OpenACC *cache* directive provides a way to specify a hint for code generator on which range of values should be stored in the top-level cache.

The syntax of *cache* directive is the following:

```
#pragma acc cache(var-list)
```

Cache directive is allowed only inside (in the beginning) of the loop. Only array variables or sub-array specifications may be specified as its arguments.

Below is the example of *cache* directive usage.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
#pragma acc parallel loop copy(B) copyin(A)
for (i=0; i < n; i++) {
#pragma acc cache(A[i * TILE_SIZE, TILE_SIZE], B[0:n])
  for (j=0; j < n; j++) {
    B[j] = A[i * TILE_SIZE + j] * B[j];
  }
}
```

Cache directive is not supported in the current release of PathScale ENZO compiler, but may be supported in future releases.

12 Environment variables

12.1 OpenACC environment variables

OpenACC specification defines two environment variables: ACC_DEVICE_TYPE and ACC_DEVICE_NUM.

ACC_DEVICE_TYPE environment variable defines a default device type for accelerated code offloading.

ADD_DEVICE_NUM defines number of devices of ACC_DEVICE_TYPE type to be used by runtime.

These environment variables are not supported in the current release of PathScale ENZO compiler, but may be supported in future releases.

12.2 OpenMP environment variables

OpenMP defines environemtnal variables as part of the standard and please refer to OpenMP standard for the exact list.

13 Runtime API

13.1 OpenACC Runtime API

OpenACC specification defines a number of runtime API functions.

First group of API functions is dedicated to device type identification and management

```
int acc_get_num_devices( acc_device_t ); // gets number of available device of specified type
void acc_set_device_type( acc_device_t ); // specifies which device type to use for offloading of kernels
acc_device_t acc_get_device_type( void ); // gets type of the device to which next kernel will be offloaded
void acc_set_device_num( int, acc_device_t ); // tells runtime which device to use
int acc_get_device_num( acc_device_t ); // gets which device will be used for next offloaded kernel
int acc_on_device( acc_device_t ); // tests if current execution thread is within device of specified type
```

Next group of two functions allow explicit control over device initialization and shutdown.

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

```
void acc_init( acc_device_t ); // initialize device of specified type for execution
void acc_shutdown( acc_device_t ); // shutdown device and free all acquired runtime resources
```

Asynchronous execution related API functions group introduce two functions of specific interest:

```
int acc_async_test( int ); // tests for completion of all actions associated with specified stream id
int acc_async_test_all( ) // tests for completion of all asynchronously launched actions
```

Also this group includes following function, which may (and it is recommended to) be expressed by means of OpenACC directives: *acc_wait, acc_wait_async, acc_wait_all, acc_wait_all _async.*

Last group is data management API functions. All functions on that list may (and it is recommended to) be expressed by means of OpenACC directives. This group includes following functions:
acc_malloc, acc_free, acc_copyin, acc_present_or_copyin, acc_create, acc_present_or_create, acc_copyout, acc_delete, acc_update_device, acc_update_self, acc_map_data, acc_unmap_data, acc_deviceptr, acc_hostptr, acc_is_present, acc_memcpy_to_device, acc_memcpy_from_device.

Besides API functions listed above OpenACC specification defines a set of target-specific API functions, which is not covered by the current document.

13.1 OpenMP Runtime API

OpenMP runtime defines large number of runtime API functions. Please refer to OpenMP standard for the exact list of API functions and their semantics.

14 ENZO Supported HWA

14.1.1 Hardware Accelerators

PathScale ENZO supports NVIDIA Tesla products from Kepler class or newer, AMD discrete GPU Hawaii class or newer using the open source AMDGPU driver 3.19+ and native execution on Cavium many-core ThunderX.

Glossary

Clause	OpenACC clause used as parameter of OpenACC directive
CUDA	Programming language for the NVIDIA CUDA compatible hardware
Device	A particular HWA device
Directive	OpenACC directive
Hardware Accelerators (HWA)	A device used to speedup segments of an application. Typical examples of a an HWA are : GPU, FPGA, or streaming units (SSE, ...).
ENZO runtime API	Runtime library linked with the ENZO program to manage the execution of the offloaded code region
ENZO runtime callbacks	API that provides the ENZO runtime with all the necessary

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

	services to execute a target codelet
main thread	Process that executes the original code
Remote Procedure Call (RPC)	a RPC denotes the remote execution of a codelet in a HWA

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016

Bibliography

The OpenACC Application Programming Interface, Version 2.0, 2013
The OpenMP Application Programming Interface, Version 4.0, 2013

This information is the property of PathScale Inc. and cannot be used, reproduced or transmitted without authorization .

Copyright © 2016 PathScale Inc. - DOC-ENZO-06-23-2016