

# Porting OpenACC 2.0 to OpenMP 4.0: Key Similarities and Differences

Oscar Hernandez

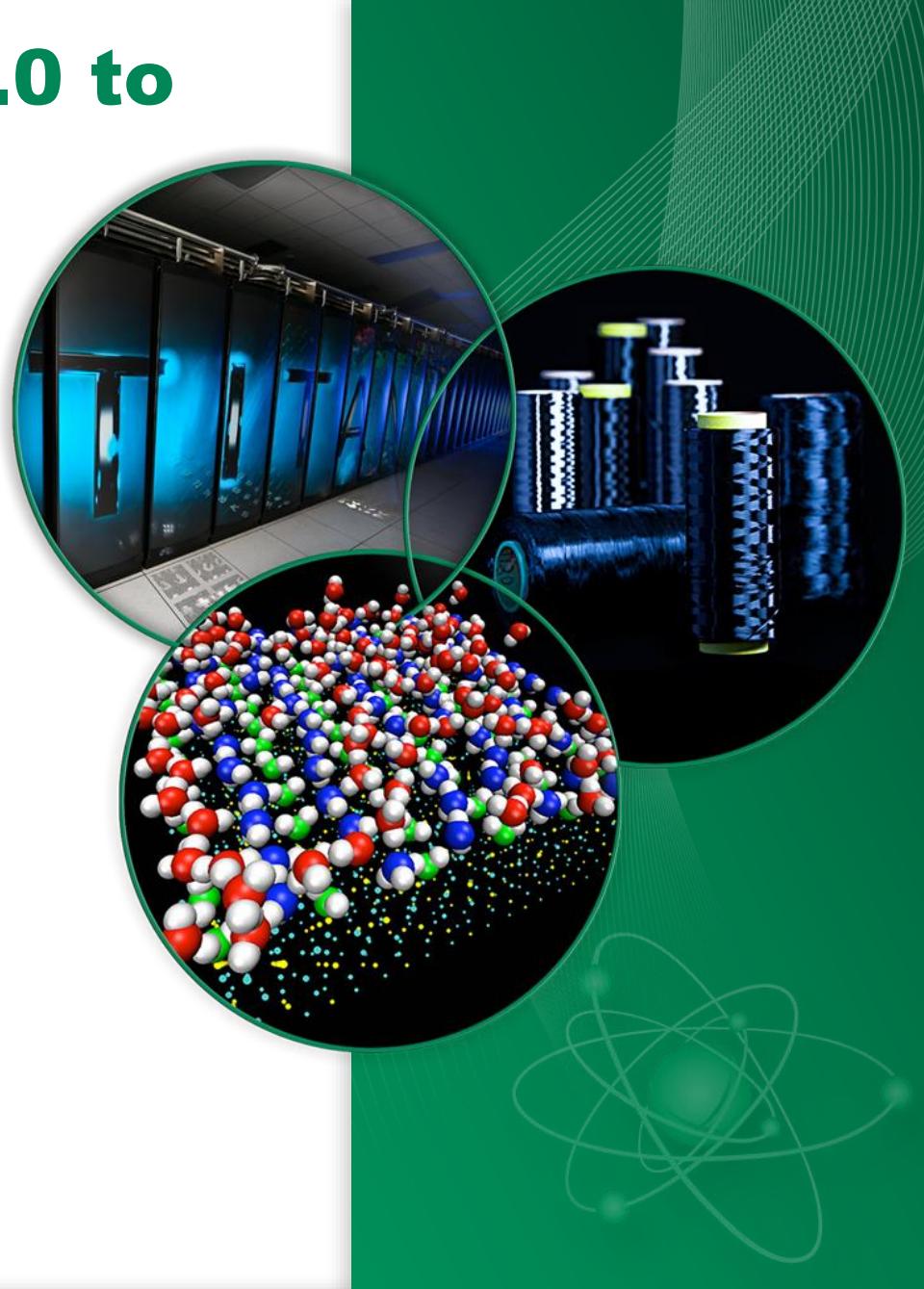
Wei Ding

Wayne Joubert

David Bernholdt

Markus Eisenbach

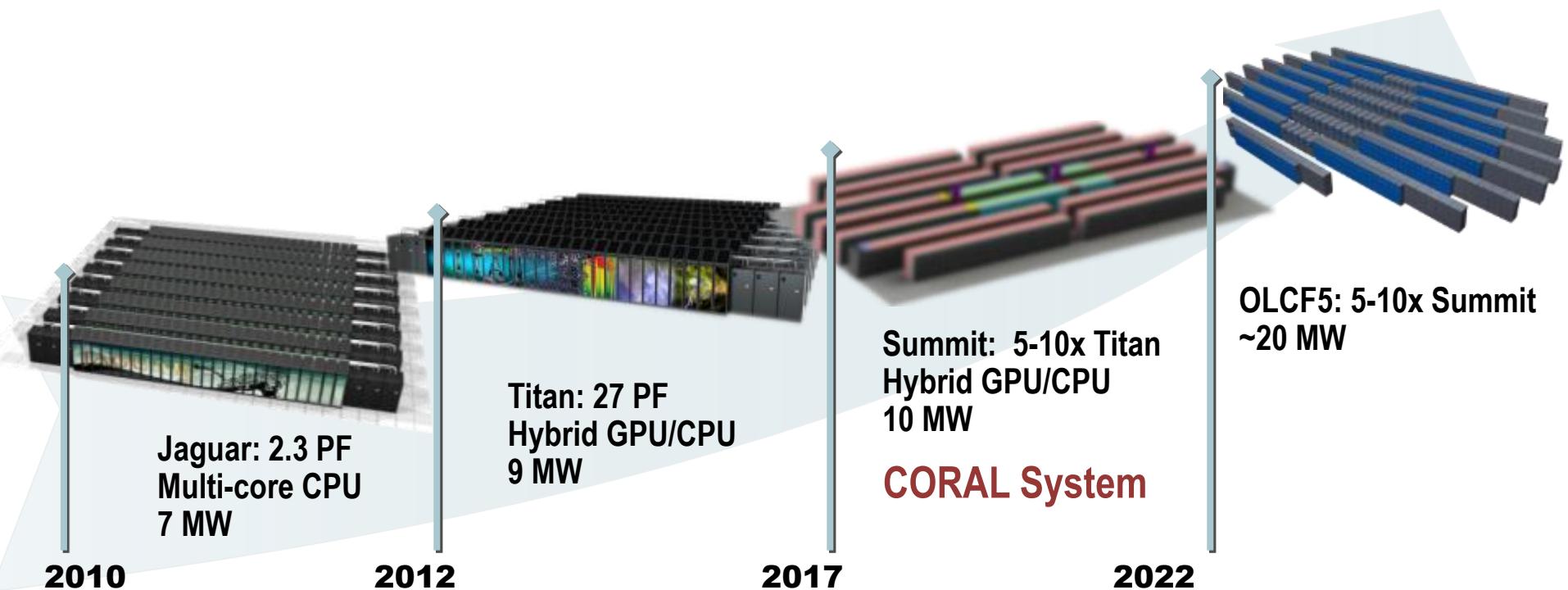
Christos Kartsaklis



# Roadmap to Exascale (ORNL)

Since clock-rate scaling ended in 2003, HPC performance has been achieved through **increased parallelism**. Jaguar scaled to 300,000 CPU cores.

Titan and beyond deliver **hierarchical parallelism** with very powerful nodes. MPI plus thread level parallelism through **OpenACC or OpenMP** plus vectors



From Sierra and Summit: Scaling to New Heights with OpenPower by David E. Bernholdt, Terri Quinn

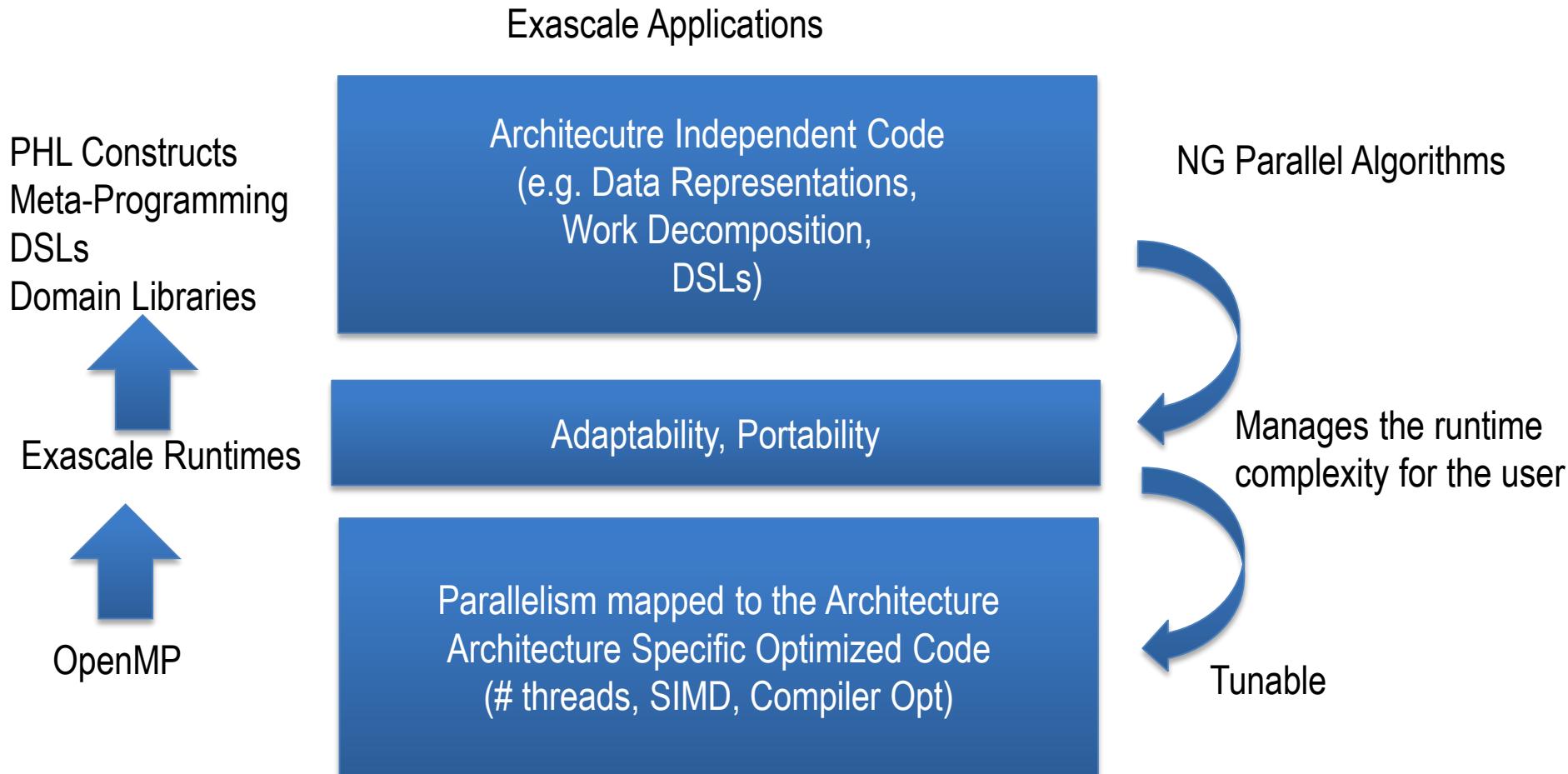
# Challenges (CORAL)

Feature	Summit	Titan
Application Performance	5-10x Titan	Baseline
Number of Nodes	~3,400	18,688
Node performance	> 40 TF	1.4 TF
Memory per Node	>512 GB (HBM + DDR4)	38GB (GDDR5+DDR3)
NVRAM per Node	800 GB	0
Node Interconnect	NVLink (5-12x PCIe 3)	PCIe 2
System Interconnect (node injection bandwidth)	Dual Rail EDR-IB (23 GB/s)	Gemini (6.4 GB/s)
Interconnect Topology	Non-blocking Fat Tree	3D Torus
Processors	IBM POWER9 NVIDIA Volta™	AMD Opteron™ NVIDIA Kepler™
File System	120 PB, 1 TB/s, GFS™	32 PB, 1 TB/s, Lustre®
Peak power consumption	10 MW	9 MW

# Programming Models Challenges

- Parallelism, Heterogeneity, Memory Hierarchies, Resilience, Power
- Performance Portability
  - Collaboration between industry and research institutions
    - ...but mostly industry (because they built the hardware)
- Maintenance
  - Maintaining a network stack is time consuming and expensive
  - Industry have resources and strategic interest for this
- Extendibility
  - MPI+X+Y OR OMP + X' (e.g. PGAS, Exascale runtimes)
  - Exascale programming environment is emerging

# Exascale Challenges

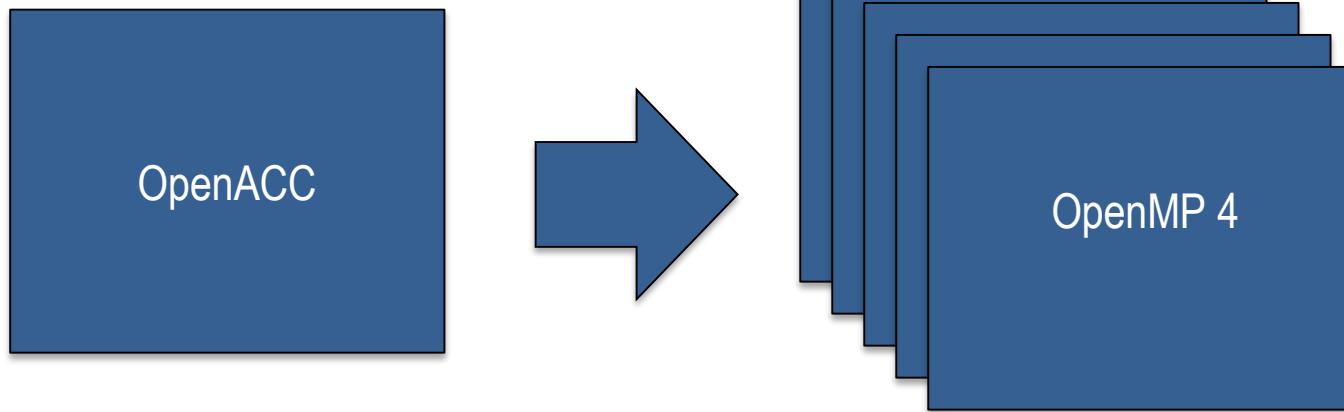


# Directive-based programming

- A key strategy for portable programming of accelerators
- OpenACC was designed to program Titan
  - Several current implementations of OpenACC (PGI, Cray, GCC, Pathscale and several research compilers)
- Growing support for OpenMP 4.
  - Implementations are starting to emerge: Cray, Intel, LLVM, Pathscale, GCC
- Community of users based on OpenACC
  - Largely because of mature implementations of OpenACC
- Plan to port benchmarks to OpenMP 4.0 understand differences and similarities
  - There are some technical challenges that OpenMP 4.1 implementers are solving
    - E.g. Simulating SIMD on GPUs
    - Performance portability of codes

# Converting OpenACC to OpenMP 4

- Main difference is in the way we express the directives
  - Descriptive vs Prescriptive



- Many constructs have 1-1 mappings
- Others constructs can be lowered from OpenACC to OpenMP
- Some constructs are present in one but not the other
- At some points there are subtle differences, e.g., OpenACC allows the compiler more discretion regarding how loops are mapped to hierarchical parallelism. OpenMP 4 the user has the ultimate decision.

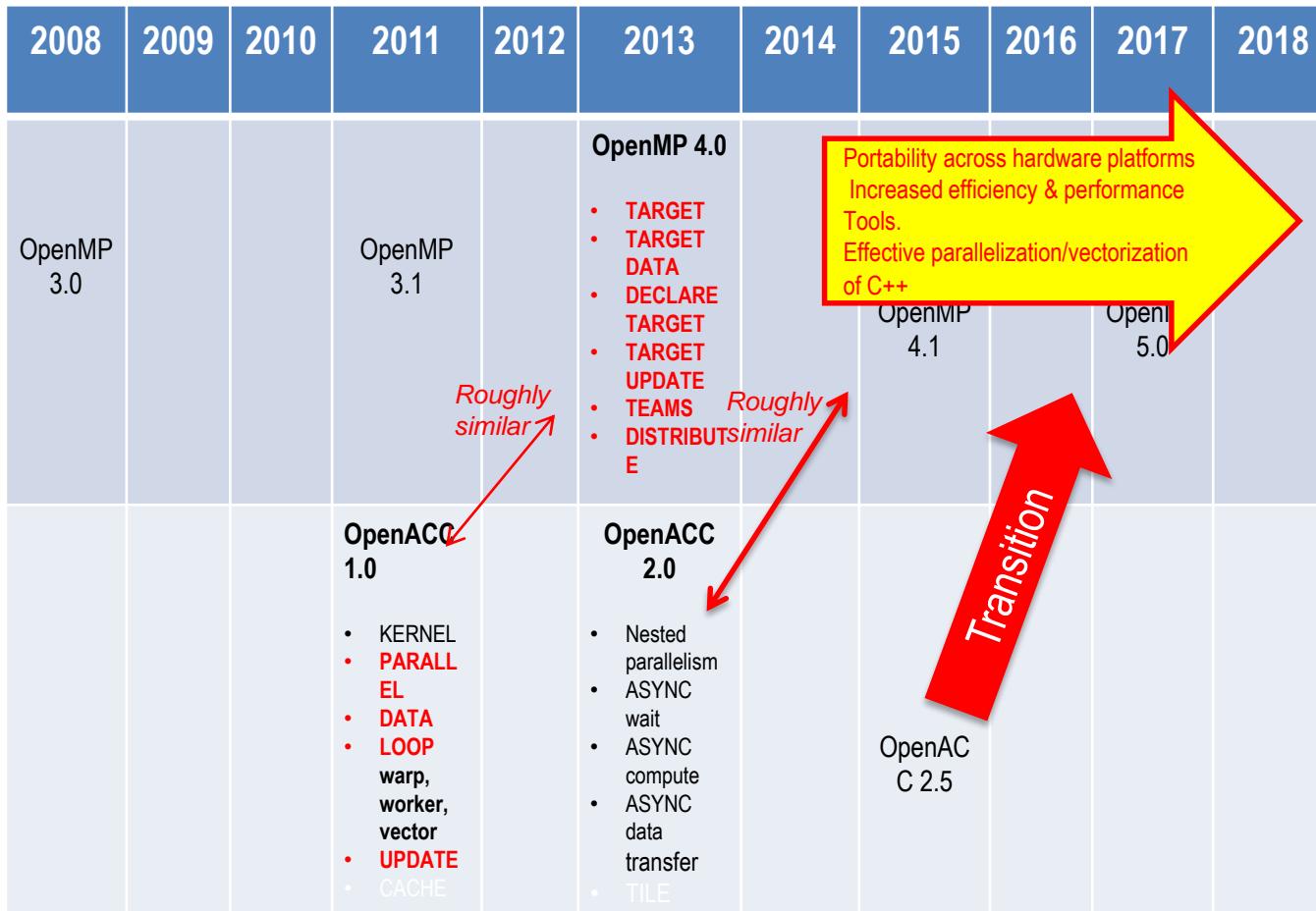
# OpenACC 2.0 vs OpenMP 4

- OpenACC 2.0 features evolving with new features which may impact OpenMP 4.1 or 5.
- OpenACC interoperability with OpenMP is important incrementally port codes to OpenMP 4
  - Experiment with OpenACC features not available in OpenMP
- Current application investments in OpenACC porting are preserved when porting to OpenMP

OpenACC 2.0	OpenMP 4.0
parallel	target
parallel/gang/workers/vector	target teams/parallel/simd
data	target data
parallel loop	teams/distribute/parallel for
update	target update
cache	Implicit compiler opt.
wait	OpenMP 4.1 draft
declare	declare target
data enter/exit	OpenMP 4.1 draft
routine	declare target
async /wait	OpenMP 4.1 draft
device type	OpenMP 4.1 draft
tile	
host data	OpenMP 4.1 draft

# OpenMP and OpenACC progress

- OpenACC innovation continues, OpenMP adopts relevant features



*Programming models must evolve before they can stabilize*

10/1/2015

# Steps for the translation

- Normalize OpenACC supported directives in OpenMP
- Data Directives
- Parallel directives
- Accelerator Subroutines
- Convert runtime APIs

# Converting OpenACC 2 to OpenMP 4: Procedure (1)

1. The user must modify any OpenACC constructs for which no equivalent counterpart exists in OpenMP:
  - Explicit data regions
  - **kernel**s directive
  - **device\_type** clause // no problem for OpenMP 4.1
  - **host\_data** and **link** clauses // no problem for OpenMP 4.1
  - **cache** directive
  - Complex use of asynchronous streams

— Rewrite these in terms of features that map to OpenMP 4

# Translating OpenACC 2 to OpenMP 4: Procedure (2)

## 2. Translate data regions:

- `acc data` → `omp target data`
- `acc declare data` → `omp target declare data`
- `create(...)` → `map(alloca:...)`
- `pcopy(...)` → `map(tofrom:...)`
- `pcopyin(...)` → `map(to:...)`
- `pcopyout(...)` → `map(from:...)`
- `scalars` need to be converted to `firstprivate`
- use of `async(...)` can be replaced by use of OpenMP CPU threads or tasks to handle transfers
- if multiple devices used, replace calls to `acc_set_device_num(...)` with `device(...)` clause

Note: OpenACC 2.5 draft – copy = pcopy

# Translating OpenACC 2 to OpenMP 4: Procedure (3)

## 3. Translate data update operations:

- **acc update** → **omp target update**
- **host(...)** → **from(...)**
- **self(...)** → **from(:...)**
- **device(...)** → **to(...)**
- use of **async (...)** can be replaced by use of OpenMP CPU threads or tasks to handle transfers

# Translating OpenACC 2 to OpenMP 4: Procedure (4)

4. Translate accelerator parallel regions: generally,

- `acc parallel` → `omp target teams`
- `acc loop gang` → `omp distribute`
- `acc loop worker` → `omp parallel for [simd]`
- `acc loop vector` → `omp simd`
- `acc loop independent` -> `teams |parallel for | simd or any permutation of the combined directive`
- `gang(...)` → `dist_schedule(...)`
- `num_gangs(...)` → `num_teams(...)`
- `num_workers(...)` → `thread_limit(...)`
- `vector_length(...)` → `safelen(...)`
- use of `async(...)` can be replaced by use of OpenMP CPU threads or tasks to handle device execution

# Translating OpenACC 2 to OpenMP 4: Procedure (5)

## 5. Adjust function attribute specifiers:

- `acc routine` → `omp declare target / end declare target`
- OpenACC `gang`, `worker`, `vector`, `seq` clauses have no exact counterpart in OpenMP

# Example: Stencil (Version 1)

## OpenACC

```
• void cpu_stencil(float c0,float c1, float *A0,float * Anext,const int nx, const int ny, const int nz)
• {
•     int i, j, k;
• #pragma acc kernels pcopyin(A0[0:nx*ny*nz]),
• pcopyout(Anext[0:nx*ny*nz])
• {
• #pragma acc loop independent
•     for(i=1;i<nx-1;i++)
•     {
• #pragma acc loop independent
•     for(j=1;j<ny-1;j++)
•     {
• #pragma acc loop independent
•     for(k=1;k<nz-1;k++)
•     {
•         Anext[Index3D (nx, ny, i, j, k)] =
•             (A0[Index3D (nx, ny, i, j, k + 1)] +
•              A0[Index3D (nx, ny, i, j, k - 1)] +
•              A0[Index3D (nx, ny, i, j + 1, k)] +
•              A0[Index3D (nx, ny, i, j - 1, k)] +
•              A0[Index3D (nx, ny, i + 1, j, k)] +
•              A0[Index3D (nx, ny, i - 1, j, k)])*c1
•             - A0[Index3D (nx, ny, i, j, k)]*c0;
•     }
•     }
• }
• }
```

## OpenMP 4.0

```
• void cpu_stencil(float c0,float c1, float *A0,float * Anext,const int nx, const int ny, const int nz)
• {
•     int i, j, k;
•     int size=nx*ny*nz;
• #pragma omp target map(alloc:A0[0:size], Anext[0:size])
• #pragma omp parallel for collapse(2)
•     for(k=1;k<nz-1;k++)
•     {
•         for(j=1;j<ny-1;j++)
•         {
• #pragma omp simd
•             for(i=1;i<nx-1;i++)
•             {
•                 Anext[Index3D (nx, ny, i, j, k)] =
•                     (A0[Index3D (nx, ny, i, j, k + 1)] +
•                      A0[Index3D (nx, ny, i, j, k - 1)] +
•                      A0[Index3D (nx, ny, i, j + 1, k)] +
•                      A0[Index3D (nx, ny, i, j - 1, k)] +
•                      A0[Index3D (nx, ny, i + 1, j, k)] +
•                      A0[Index3D (nx, ny, i - 1, j, k)])*c1
•                     - A0[Index3D (nx, ny, i, j, k)]*c0;
•             }
•         }
•     }
• }
```



# Example: Stencil (Version 2)

## OpenACC

```
void cpu_stencil(float c0,float c1, float *A0,float * Anext,const int nx, const int ny,
const int nz)
{
    int i, j, k;
#pragma acc kernels pcopyin(A0[0:nx*ny*nz]),
pcopyout(Anext[0:nx*ny*nz])
{
#pragma acc loop independent
    for(i=1;i<nx-1;i++)
    {
#pragma acc loop independent
        for(j=1;j<ny-1;j++)
        {
#pragma acc loop independent
            for(k=1;k<nz-1;k++)
            {
                Anext[Index3D (nx, ny, i, j, k)] =
                (A0[Index3D (nx, ny, i, j, k + 1)] +
                A0[Index3D (nx, ny, i, j, k - 1)] +
                A0[Index3D (nx, ny, i, j + 1, k)] +
                A0[Index3D (nx, ny, i, j - 1, k)] +
                A0[Index3D (nx, ny, i + 1, j, k)] +
                A0[Index3D (nx, ny, i - 1, j, k)])*c1
                - A0[Index3D (nx, ny, i, j, k)]*c0;
            }
        }
    }
}
```

## OpenMP 4.0

```
void cpu_stencil(float c0,float c1, float *A0,float * Anext,const int nx, const int ny,
const int nz)
{
    int i, j, k;
    int size=nx*ny*nz;
#pragma omp target map(alloc:A0[0:size], Anext[0:size])
#pragma omp parallel for collapse(2)
    for(k=1;k<nz-1;k++)
    {
        for(j=1;j<ny-1;j++)
        {
#pragma parallel for
            for(i=1;i<nx-1;i++)
            {
                Anext[Index3D (nx, ny, i, j, k)] =
                (A0[Index3D (nx, ny, i, j, k + 1)] +
                A0[Index3D (nx, ny, i, j, k - 1)] +
                A0[Index3D (nx, ny, i, j + 1, k)] +
                A0[Index3D (nx, ny, i, j - 1, k)] +
                A0[Index3D (nx, ny, i + 1, j, k)] +
                A0[Index3D (nx, ny, i - 1, j, k)])*c1
                - A0[Index3D (nx, ny, i, j, k)]*c0;
            }
        }
    }
}
```

# Example: Stencil (Version 3)

## OpenACC

```
• void cpu_stencil(float c0,float c1, float *A0,float * Anext,const int nx, const int ny, const int nz)
• {
•     int i, j, k;
• #pragma acc kernels pcopyin(A0[0:nx*ny*nz]),
• pcopyout(Anext[0:nx*ny*nz])
• {
• #pragma acc loop independent
•     for(i=1;i<nx-1;i++)
•     {
• #pragma acc loop independent
•     for(j=1;j<ny-1;j++)
•     {
• #pragma acc loop independent
•     for(k=1;k<nz-1;k++)
•     {
•         Anext[Index3D (nx, ny, i, j, k)] =
•             (A0[Index3D (nx, ny, i, j, k + 1)] +
•              A0[Index3D (nx, ny, i, j, k - 1)] +
•              A0[Index3D (nx, ny, i, j + 1, k)] +
•              A0[Index3D (nx, ny, i, j - 1, k)] +
•              A0[Index3D (nx, ny, i + 1, j, k)] +
•              A0[Index3D (nx, ny, i - 1, j, k)])*c1
•             - A0[Index3D (nx, ny, i, j, k)]*c0;
•     }
•   }
• }
```

## OpenMP 4.0

```
• void cpu_stencil(float c0,float c1, float *A0,float * Anext,const int nx, const int ny, const int nz)
• {
•     int i, j, k;
•     int size=nx*ny*nz;
• #pragma omp target map(alloc:A0[0:size], Anext[0:size])
• #pragma omp teams distribute parallel for collapse(2)
•     for(k=1;k<nz-1;k++)
•     {
•         for(j=1;j<ny-1;j++)
•         {
• #pragma omp simd
•             for(i=1;i<nx-1;i++)
•             {
•                 Anext[Index3D (nx, ny, i, j, k)] =
•                     (A0[Index3D (nx, ny, i, j, k + 1)] +
•                      A0[Index3D (nx, ny, i, j, k - 1)] +
•                      A0[Index3D (nx, ny, i, j + 1, k)] +
•                      A0[Index3D (nx, ny, i, j - 1, k)] +
•                      A0[Index3D (nx, ny, i + 1, j, k)] +
•                      A0[Index3D (nx, ny, i - 1, j, k)])*c1
•                     - A0[Index3D (nx, ny, i, j, k)]*c0;
•             }
•         }
•     }
```



# Example: Stencil (Version 4)

## OpenACC

```
void cpu_stencil(float c0,float c1, float *A0,float * Anext,const int nx, const int ny,
const int nz)
{
    int i, j, k;
#pragma acc kernels pcopyin(A0[0:nx*ny*nz]),
pcopyout(Anext[0:nx*ny*nz])
{
#pragma acc loop independent
    for(i=1;i<nx-1;i++)
    {
#pragma acc loop independent
        for(j=1;j<ny-1;j++)
        {
#pragma acc loop independent
            for(k=1;k<nz-1;k++)
            {
                Anext[Index3D (nx, ny, i, j, k)] =
                    (A0[Index3D (nx, ny, i, j, k + 1)] +
                     A0[Index3D (nx, ny, i, j, k - 1)] +
                     A0[Index3D (nx, ny, i, j + 1, k)] +
                     A0[Index3D (nx, ny, i, j - 1, k)] +
                     A0[Index3D (nx, ny, i + 1, j, k)] +
                     A0[Index3D (nx, ny, i - 1, j, k)])*c1
                    - A0[Index3D (nx, ny, i, j, k)]*c0;
            }
        }
    }
}
```

## OpenMP 4.0

```
void cpu_stencil(float c0,float c1, float *A0,float * Anext,const int nx, const int ny,
const int nz)
{
    int i, j, k;
    int size=nx*ny*nz;
#pragma omp target map(alloc:A0[0:size], Anext[0:size])
#pragma omp teams distribute parallel for simd collapse(3)
    for(k=1;k<nz-1;k++)
    {
        for(j=1;j<ny-1;j++)
        {
            for(i=1;i<nx-1;i++)
            {
                Anext[Index3D (nx, ny, i, j, k)] =
                    (A0[Index3D (nx, ny, i, j, k + 1)] +
                     A0[Index3D (nx, ny, i, j, k - 1)] +
                     A0[Index3D (nx, ny, i, j + 1, k)] +
                     A0[Index3D (nx, ny, i, j - 1, k)] +
                     A0[Index3D (nx, ny, i + 1, j, k)] +
                     A0[Index3D (nx, ny, i - 1, j, k)])*c1
                    - A0[Index3D (nx, ny, i, j, k)]*c0;
            }
        }
    }
}
```

# Example: Euler kernel

## OpenACC

```
typedef double DATATYPE;
#define N 64 // dimension size of array
#define ITERATIONS 10 // number of iterations

#pragma acc routine seq
DATATYPE f(DATATYPE t,DATATYPE y)
{
    return y + 2*t*exp(2*t);
}
void run(DATATYPE* y, int iterations)
{
    #pragma acc data pcopy(y[0:N])
    for (int t=0; t<iterations; ++t)
    {
        #pragma acc parallel loop
        for (int i=0; i<N; ++i) {
            y[i] = y[i] + f(t,y[i]);
        }
        #pragma acc update host(y[0:N])
        print(y);
    }
}
```

## OpenMP 4.0

```
typedef double DATATYPE;
#define N 64 // dimension size of array
#define ITERATIONS 10 // number of iterations

#pragma omp declare target
DATATYPE f(DATATYPE t,DATATYPE y)
{
    return y + 2*t*exp(2*t);
}
#pragma omp end declare target

void run(DATATYPE* y, int iterations)
{
    #pragma omp target data map(tofrom:y[0:N])
    for (int t=0; t<iterations; ++t)
    {
        #pragma omp target teams distribute parallel for simd
        for (int i=0; i<N; ++i) {
            y[i] = y[i] + f(t,y[i]);
        }
        #pragma omp target update from(y[0:N])
        print(y);
    }
}
```

# Example: Convolution Kernel

OpenACC

```
void run(DATATYPE* a, DATATYPE *b, int iterations)
{
    #pragma acc data pcopy(b[0:N]) pcreate(a[0:N])
    for (int iteration=0; iteration<iterations; ++iteration)
    {
        #pragma acc parallel loop
        for (int i=1; i<N-1; ++i) {
            a[i] = b[i] + b[i-1] + b[i+1];
        }

        #pragma acc parallel loop
        for (int i=1; i<N-1; ++i) {
            b[i] = a[i];
        }

        if (iteration%10 == 0) {
            #pragma acc update host(b[0:N])
            for (int i=0; i<N; ++i) {
                printf("%d ", b[i]);
            }
            printf("\n");
        }
    }
}
```

OpenMP 4.0

```
void run(DATATYPE* a, DATATYPE *b, int iterations)
{
    #pragma omp target data map(tofrom:b[0:N]) map(alloc:a[0:N])
    for (int iteration=0; iteration<iterations; ++iteration)
    {
        #pragma omp target teams distribute parallel for simd
        for (int i=1; i<N-1; ++i) {
            a[i] = b[i] + b[i-1] + b[i+1];
        }

        #pragma omp target teams distribute parallel for simd
        for (int i=1; i<N-1; ++i) {
            b[i] = a[i];
        }

        if (iteration%10 == 0) {
            #pragma omp target update from(b[0:N])
            for (int i=0; i<N; ++i) {
                printf("%d ", b[i]);
            }
            printf("\n");
        }
    }
}
```

# Methodology

- Successfully ported several benchmarks using this methodology
  - From OpenACC to OpenMP 4.0
- Functional correct results
- Focus on performance and portability
  - Compiler implementations
    - OpenMP 4.1 leave a lot of freedom to compilers
    - [implementation default values] can we assume smart compilers?
  - GPUs more sensible to teams and # of threads
  - Xeon Phis sensible to SIMD [safelen]
    - Loop interchanges are needed to exploit more SIMD
  - Macros may be needed to tune code

# What is missing in OpenMP 4.1

- Deep copy
  - Problem may go away but is important for data locality or privatization.
- Support for different types of memory? Memory pools...
- How to specify different memory hierarchies in the programming model
  - Shared memory within GPU, etc
- Support for C++ for Accelerators
  - Are directives the right approach for C++?
    - Templates
    - Inheritance
    - STLs
- Tools APIs

# Summary

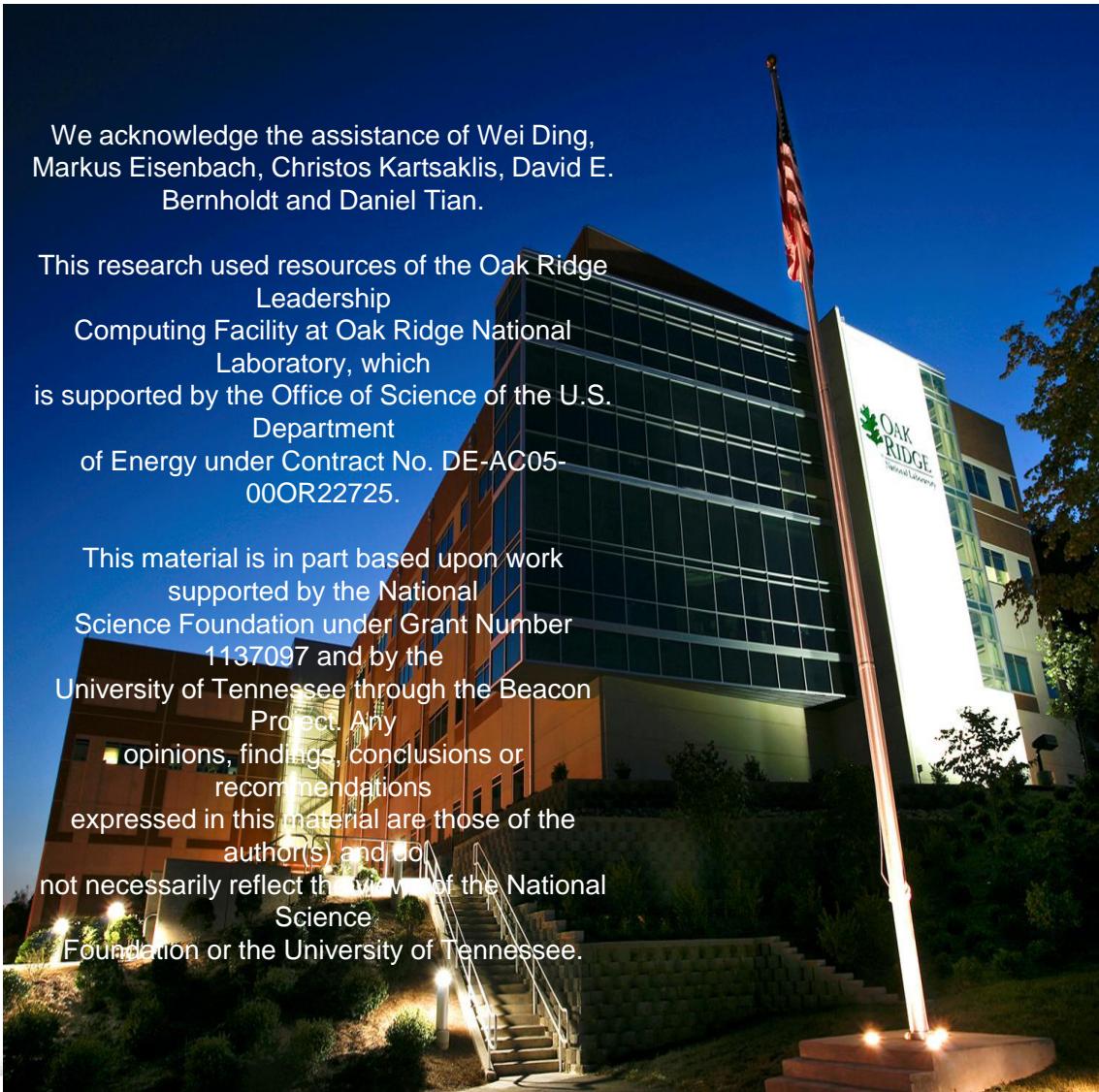
- OpenMP 4 and OpenACC support for accelerator devices is similar in many respects
- Converting code from OpenACC to OpenMP 4 is straight forward
- Programmer needs to be aware of optimization choices
  - MACROS to generate code for different architectures
    - collapse, num\_teams, thread\_limit, etc. or develop smart algorithms in the application that can select them at runtime.
  - Challenges: maturity of compilers
- Application developers should be aware of similarities and differences in order to develop future-proofed code that can run well under either API while they transition to OpenMP.

# Questions?

We acknowledge the assistance of Wei Ding,  
Markus Eisenbach, Christos Kartsaklis, David E.  
Bernholdt and Daniel Tian.

This research used resources of the Oak Ridge  
Leadership  
Computing Facility at Oak Ridge National  
Laboratory, which  
is supported by the Office of Science of the U.S.  
Department  
of Energy under Contract No. DE-AC05-  
00OR22725.

This material is in part based upon work  
supported by the National  
Science Foundation under Grant Number  
1137097 and by the  
University of Tennessee through the Beacon  
Project. Any  
opinions, findings, conclusions or  
recommendations  
expressed in this material are those of the  
author(s) and do  
not necessarily reflect the views of the National  
Science  
Foundation or the University of Tennessee.



# OpenACC Example: Gang, Workers and Vector in a Grid

Directives used to map parallel loops to hierarchical parallelism on accelerator

Execution configuration determined by the compiler or configured manually

```
#pragma acc kernels
{
#pragma acc loop independent
    for (int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j){
            for (int k = 0; k < n; ++k){
                B[i][j*k%n] = A[i][j*k%n];
            }
        }
    }
}

#pragma acc loop gang(NB) worker(NT)
    for (int i = 0; i < n; ++i){
        #pragma acc loop vector(NI)
            for (int j = 0; j < m; ++j){
                B[i][j] = i * j * A[i][j];
            }
    }
}
```

