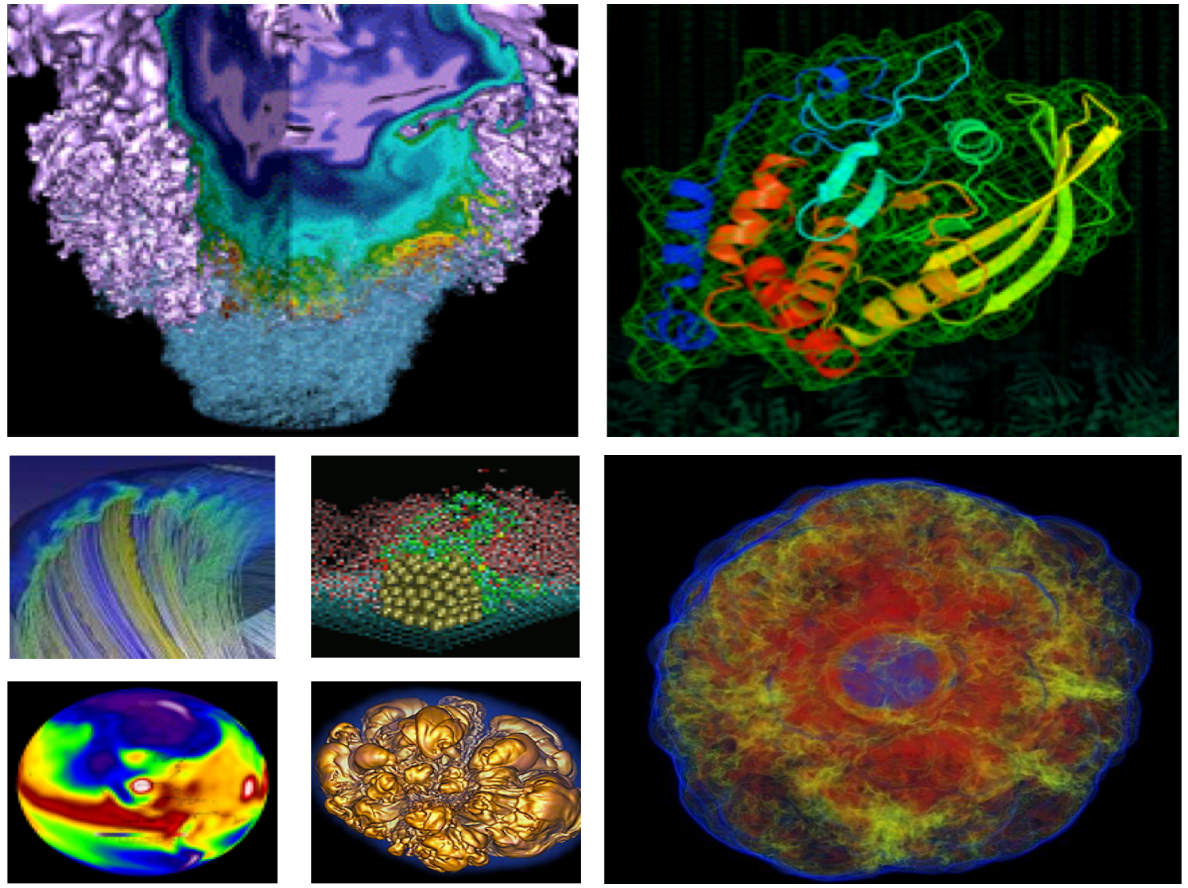
Introduction to Directive Based Programming on GPU



Helen He

Feb 28, 2020

NERSC 40 YEARS at the FOREFRONT 1974-2014

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory

Acknowledgement

- **Used materials and examples from:**
  - Jeff Larkin, Eric Wright: Slides and code examples from NVidia OpenACC training materials.
  - Tim Mattson, Simon McIntosh-Smith: SC19 Programming your GPUs with OpenMP tutorial
  - Michael Klemm and Bronis de Supinski: What's new in OpenMP 5.0
  - Oscar Hernandez, et al.: ECP 2020 OpenMP 5.0/5.1 tutorial
  - Chris Daley: slides contents, OpenMP and OpenACC performance results
  - Nvidia OpenACC Boot Camp slides.
  - Many others: for OpenMP updates and performance results presented at the DOE ECP 2020 OpenMP BoF, etc.
- **Thank you all!**

CPU *vs.* GPUs

- CPUs generally have a small number of very fast physical cores.
- GPUs have thousands of simple cores able to achieve high performance in aggregate.
- Mostly CPU and GPU do not share memory. Data move between CPU and GPU is expensive. Keep data on GPU as long as possible.
- We need to keep GPU busy, and only offload computational intensive kernels to GPU.

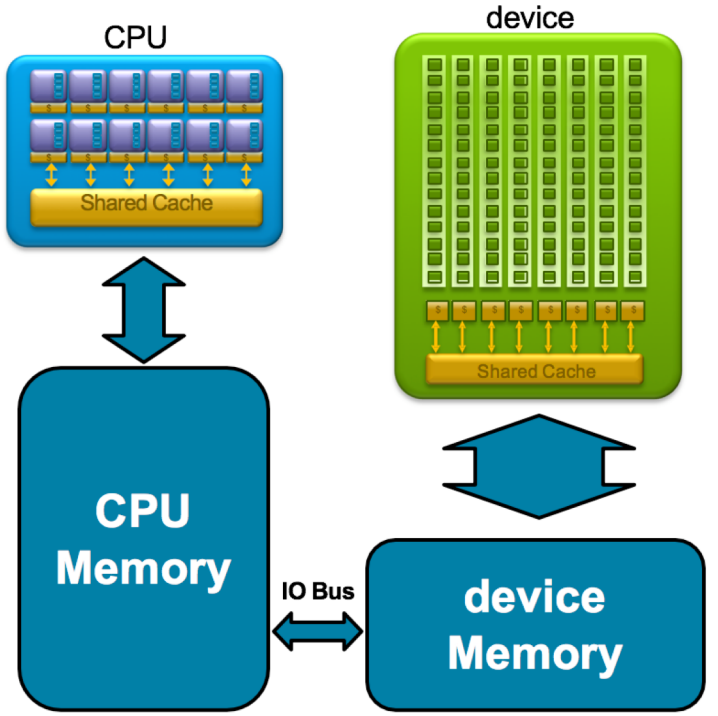


Image from NVidia

Sample OpenMP and OpenACC Codes

```
#define N 128
double x[N*N];
int i, j, k;
for (k=0; k<N*N; ++k) x[k] = k;

#pragma omp target
#pragma omp teams distribute
for (i=0; i<N; ++i) {
#pragma omp parallel for simd
  for (j=0; j<N; ++j) {
    x[j+N*i] *= 2.0;
  }
}
```

```
#define N 128
double x[N*N];
int i, j, k;
for (k=0; k<N*N; ++k) x[k] = k;

#pragma acc parallel
#pragma acc gang worker
for (i=0; i<N; ++i) {
#pragma acc vector
  for (j=0; j<N; ++j) {
    x[j+N*i] *= 2.0;
  }
}
```

directives

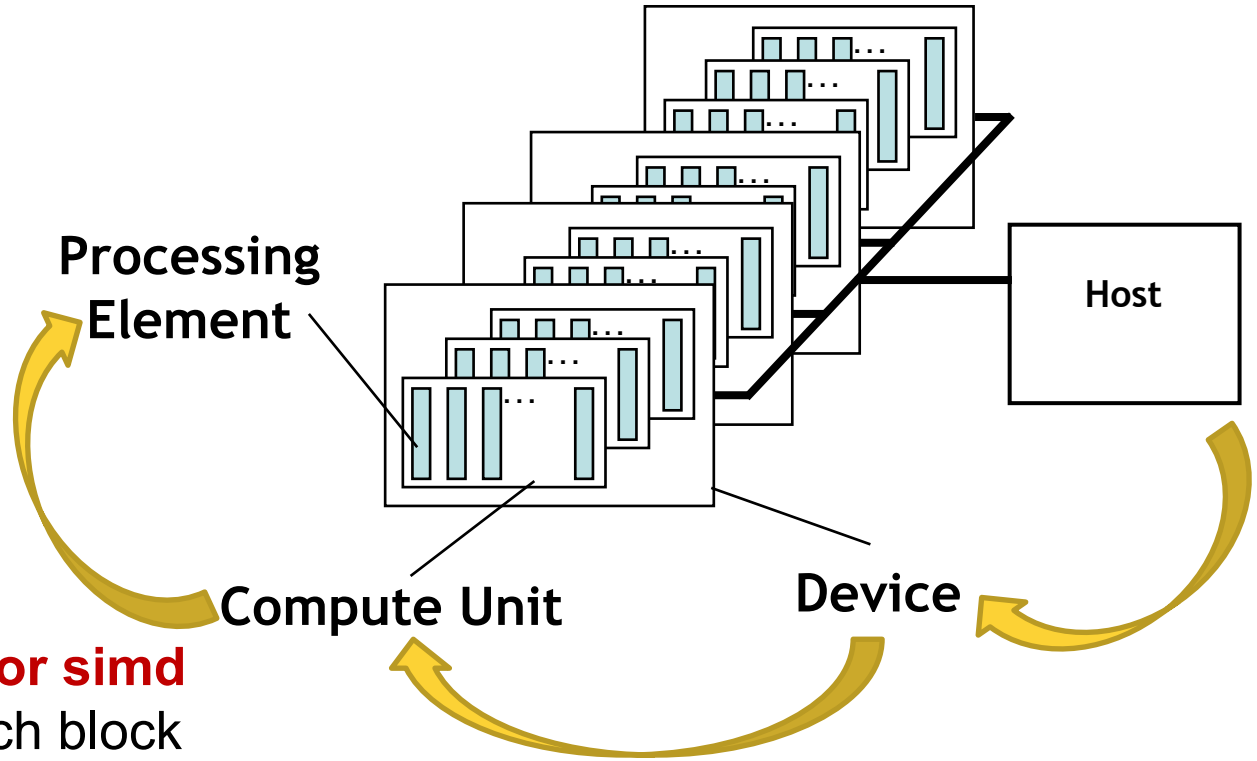
Advantages of Directive Based Parallelism

- **Incremental parallel programming**
  - Find hotspot, parallelize, check correctness, repeat
- **Single source code for sequential and parallel programs**
  - Use compiler flag to enable or disable
  - No major overwrite of the serial code
- **Works for both CPU and GPU**
- **Low learning curve, familiar C/C++/Fortran program environment**
  - Do not need to worry about lower level hardware details
- **Simple programming model than lower level programming models**
- **Portable implementation:**
  - different architectures, different compilers handle the hardware differences

Device Execution Model

- **Device: An implementation-defined logical execution unit.**
- **Can have a single host and one or more target devices (accelerators).**
- **Host and Device have separate data environment (except with managed memory or unified shared memory).**
- **The execution model is host-centric**
  - Host creates/destroys data environment on the device(s)
  - Host maps data to the device data environment.
  - Host then offloads accelerator regions to the device for execution
  - Host updates the data between the host and the device.
  - Host destroys data environment on device.

Host/Device Platform Model and OpenMP



Target construct to get onto a device

Parallel for simd to run each block of loop iterations on the processing elements

Teams construct to create a league of teams with one team of threads on each compute unit.

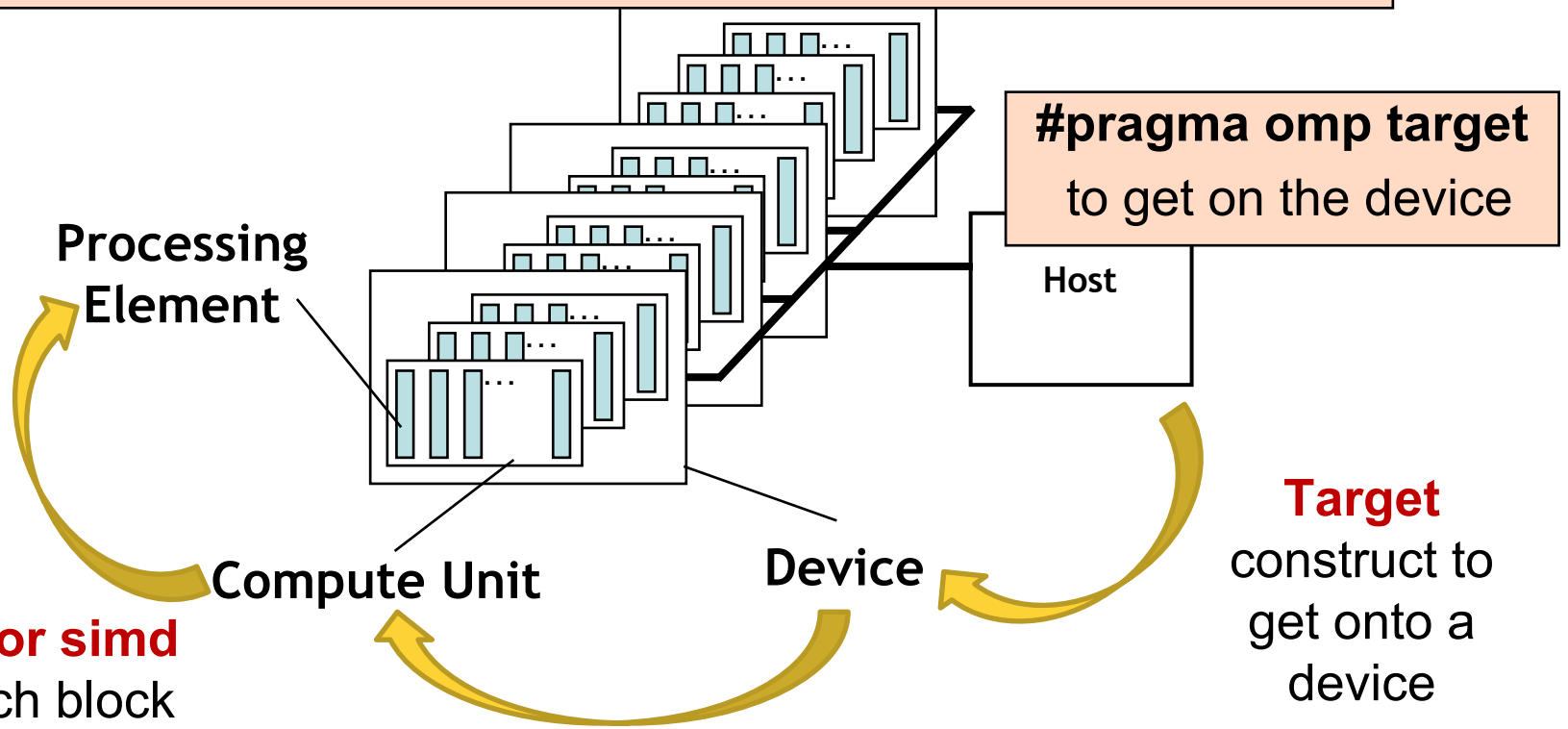
No synchronization among teams

Distribute construct to assign blocks of loop iterations to teams.

Courtesy of Tim Mattson and Simon McIntosh-Smith

Host/Device Platform Model and OpenMP

Typical usage ... let the compiler do what's best for the device:



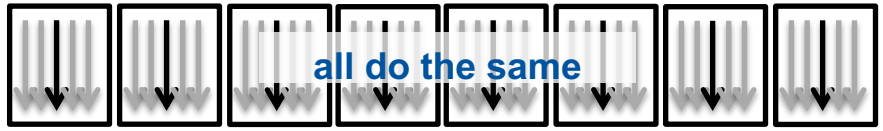
#pragma omp teams distribute parallel for simd
to assign work to the device processing elements

Distribute construct to assign blocks of loop iterations to teams.

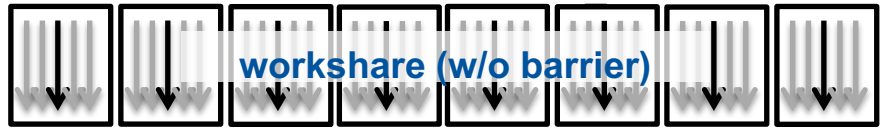
Courtesy of Tim Mattson and Simon McIntosh-Smith

Multi-level Device Parallelism

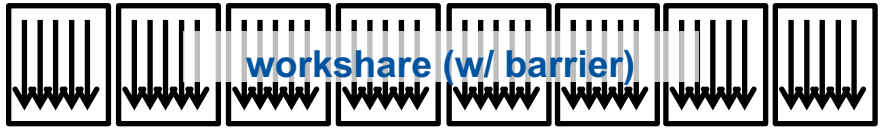
```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
```




```
#pragma omp distribute
  for (int i = 0; i < n; i += num_blocks){
```



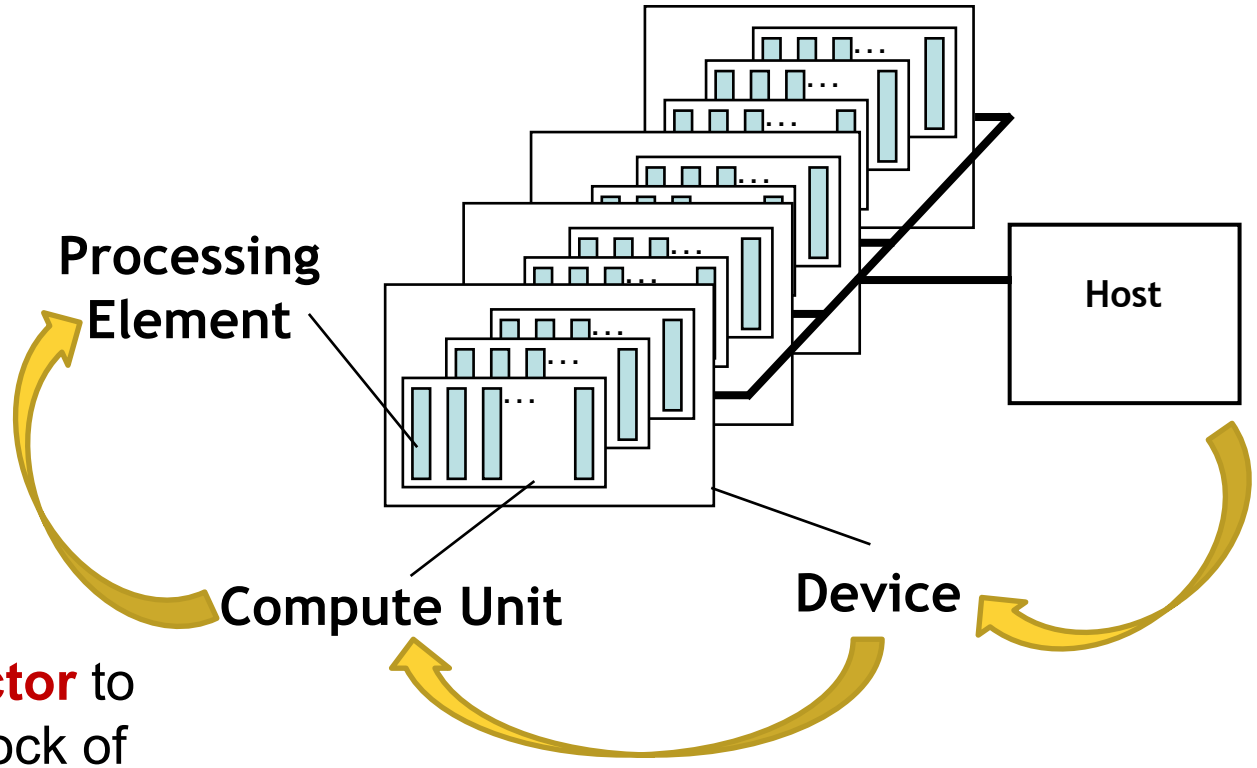

```
#pragma omp parallel for simd
    for (int j = i; j < i + num_blocks; j++) {
```




```
        y[j] = a*x[j] + y[j];
} } } }
```

Courtesy of Michael Klemm

Host/Device Platform Model and OpenACC



Parallel construct to get onto a device

Worker Vector to run each block of loop iterations on the processing elements

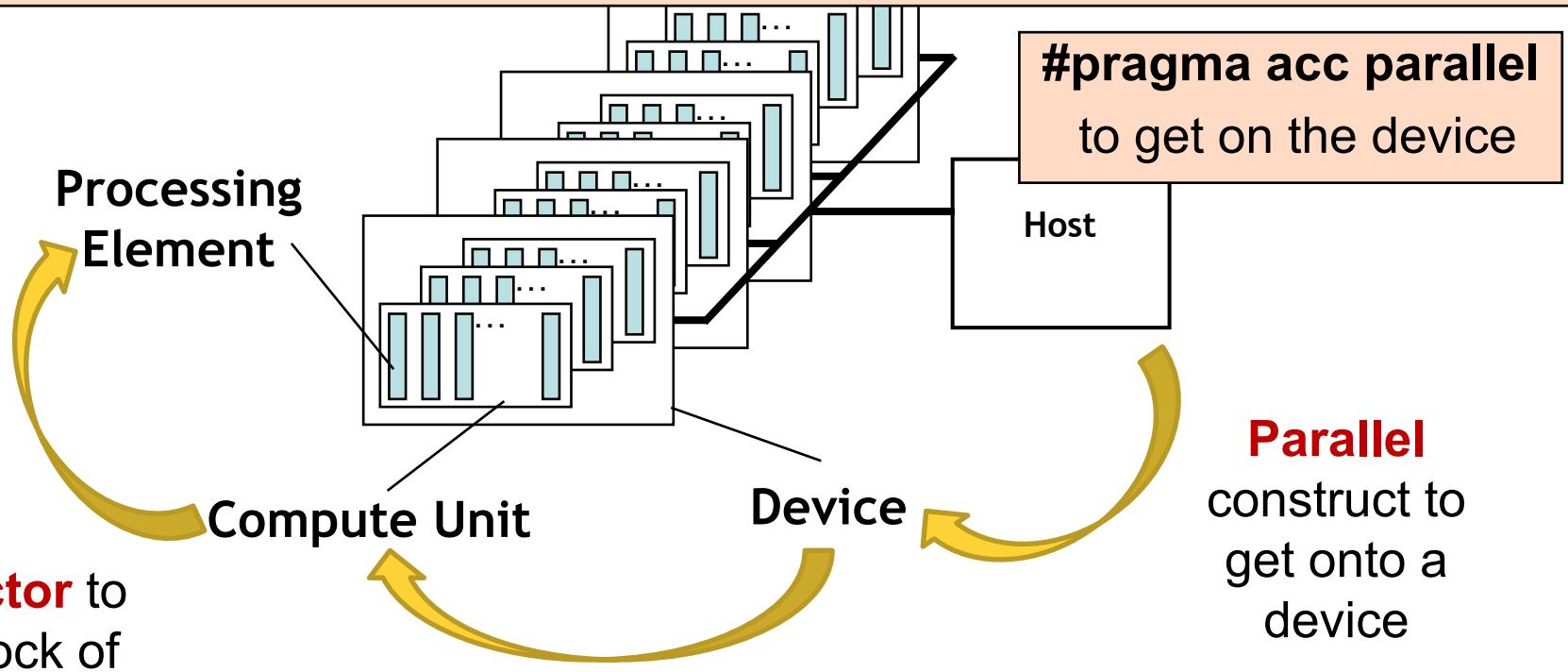
Gang directive to create a league of gangs with one gang of threads on each compute unit, and assign blocks of loop iterations to gangs.

No synchronization among gangs

Courtesy of Tim Mattson and Simon McIntosh-Smith

Host/Device Platform Model and OpenACC

Typical usage ... let the compiler do what's best for the device: use "acc loop"



Parallel construct to get onto a device

Worker Vector to run each block of loop iterations on the processing elements

Gang directive to create a league of teams with one team of threads

#pragma acc loop

to assign work to the device processing elements

Courtesy of Tim Mattson and Simon McIntosh-Smith

Sample OpenACC Codes

```
#pragma acc parallel
{
    for (i =0; i<n; i++) {
        c[i] = a[i] + b[i];
        …
    }
}
```

Without #pragma acc loop, the loop is not distributed, and all threads will execute the entire loop redundantly

```
#pragma acc parallel
{
  #pragma acc loop
      for (i =0; i<n; i++) {
          c[i] = a[i] + b[i];
          …
      }
}
```

```
#pragma acc parallel
{
  #pragma acc loop gang worker vector
      for (i =0; i<n; i++) {
          c[i] = a[i] + b[i];
          …
      }
}
```

These two are equivalent. With #pragma acc loop, it will choose the best gang/worker/vector values and parallelize the loop

The loop Directive/Construct

- OpenACC: "#pragma acc loop" lets the compiler to decide what's best gang, worker, vector values to use to parallelize.
- OpenMP 5.0: the loop construct asserts to the compiler that the iterations of a loop are free of dependencies and may be run concurrently in any order.
- It lets the OpenMP implementation to choose the right parallelization scheme.
- Each iteration execute exactly once.

```
#pragma omp target map(to:x[0:n]) map(tofrom:y)
{
   #pragma omp loop
   for (int i = 0; i < n; ++i){
      y[i] = a*x[i] + y[i];
   }
}
```

OpenACC and OpenMP Syntax

C/C++:

OpenACC: #pragma acc directive clauses
<code>

OpenMP: #pragma omp directive clauses
<code>

Fortran:

OpenACC: !$acc directive clauses
<code>
!$acc end directive

OpenMP: !$omp directive clauses
<code>
!$omp end directive

- A ***pragma*** in C/C++ or !$ in Fortran gives instructions to the compiler on how to compile the code. !$acc end or !$omp end sometimes is optional (depends on what the directive is).
- "***acc" or "omp"*** informs the compiler that this is an OpenACC or OpenMP directive.
- Directives are ignored by a compiler that does not understand a particular pragma (such as when the compiler flag is not turned on to enable OpenACC or OpenMP support).
- ***Clauses*** are specifiers or additions to directives.

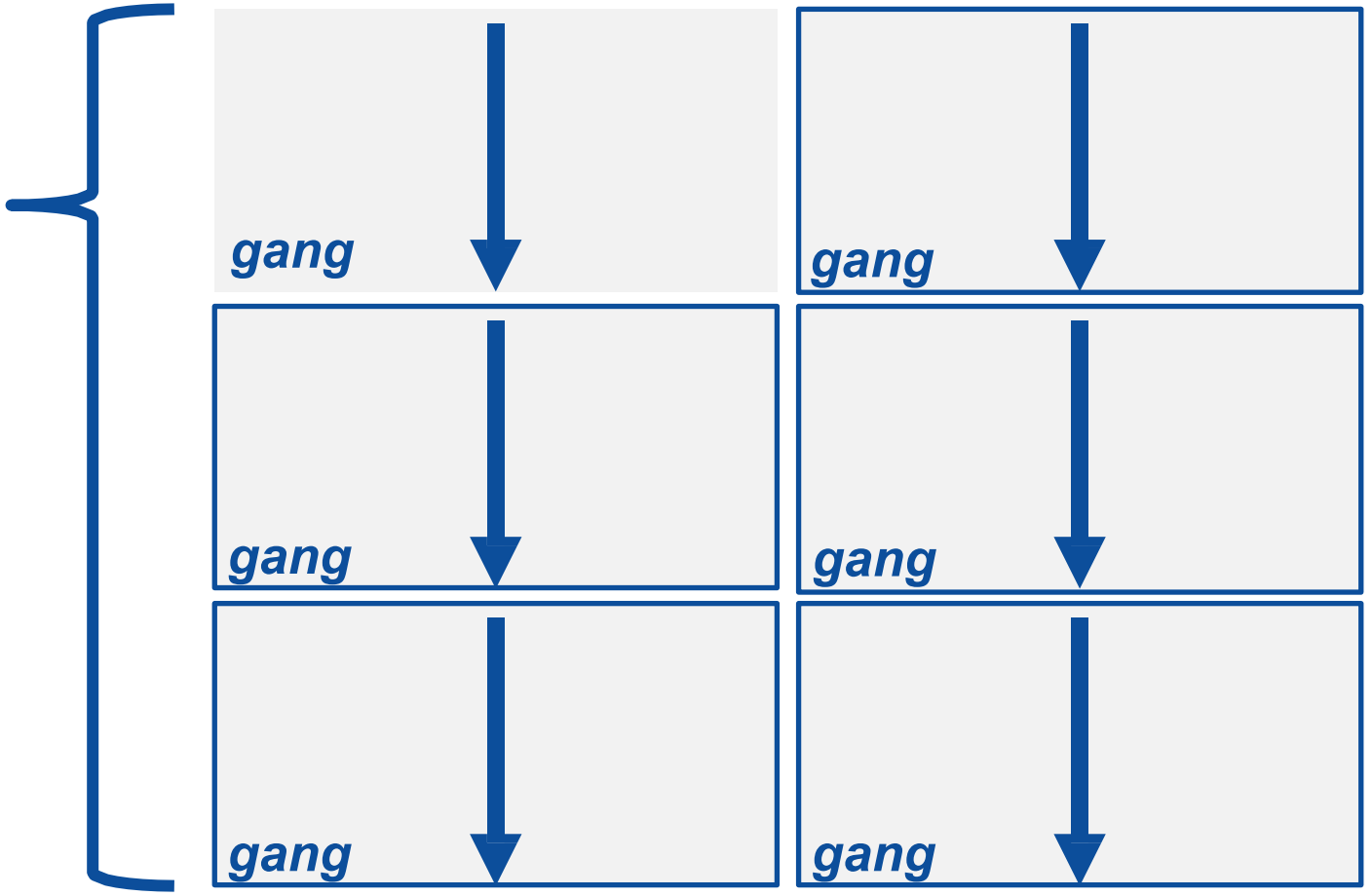
OPENACC parallel Directive

Expressing parallelism

```
#pragma acc parallel
{
```

When encountering the ***parallel*** directive, the compiler will generate *1 or more parallel* ***gangs***, which execute redundantly.

```
}
```



Slide courtesy of Jeff Larkin, Eric Wright, NVidia

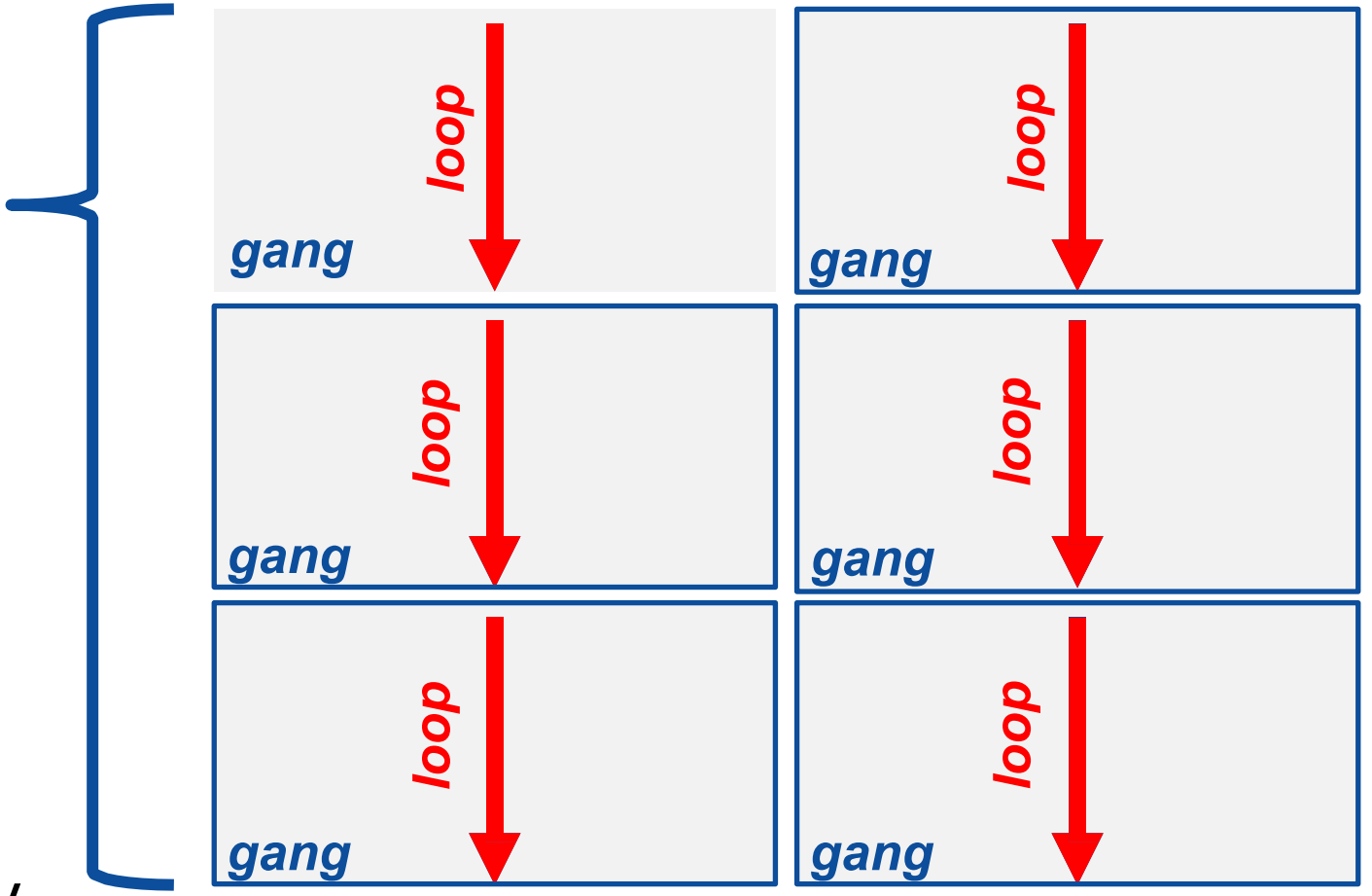
OPENACC parallel Directive

Expressing parallelism

```
#pragma acc parallel
{
    for(int i =0; i <N; i++)
    {
        // Do Something
    }
}
```

loop

This loop will be executed redundantly on each gang



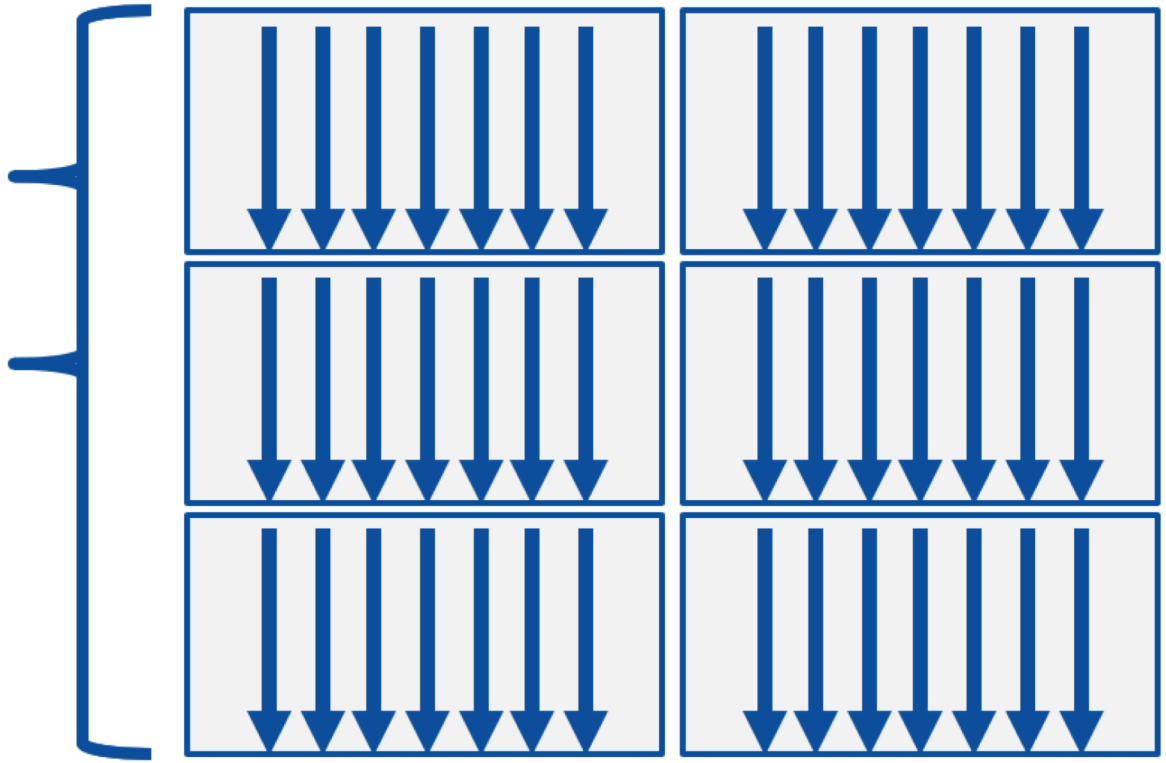
Slide courtesy of Jeff Larkin, Eric Wright, NVidia

OPENACC PARALLEL DIRECTIVE

Expressing parallelism

```
#pragma acc parallel
{

    #pragma acc loop
    for(int i = 0; i < N; i++)
    {
        // Do Something
    }

}
```

The ***loop*** directive informs the compiler which loops to parallelize.

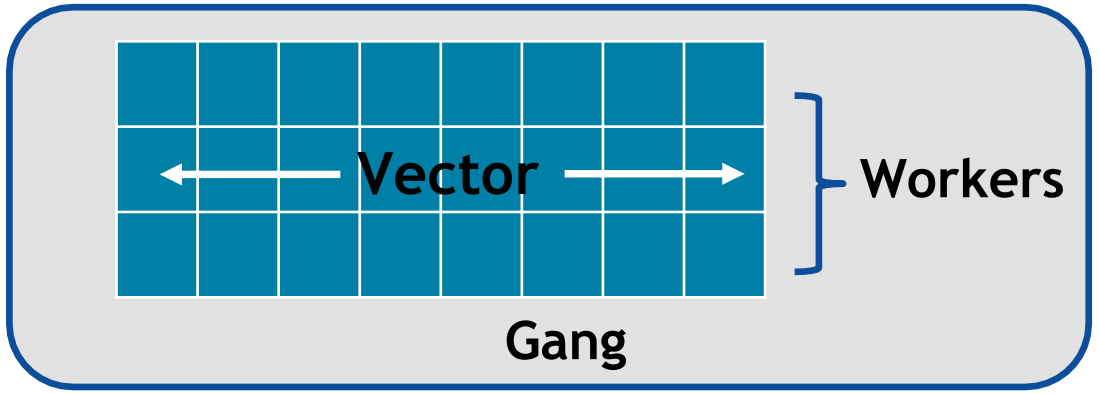


OpenACC

Slide courtesy of Jeff Larkin, Eric Wright, NVidia

GANG WORKER VECTOR

- Gang / Worker / Vector defines the various levels of parallelism we can achieve with OpenACC
- This parallelism is most useful when parallelizing multi-dimensional loop nests
- OpenACC allows us to define a generic Gang / Worker / Vector model that will be applicable to a variety of hardware, but we fill focus a little bit on a GPU specific implementation




```
#pragma acc parallel loop gang worker
for( i =0; i <N; i++ )
  #pragma acc loop vector
  for( j =0; j <M; j++ )
    <loop code >
```

Slide courtesy of Jeff Larkin, Eric Wright, NVidia

Use combined Directives

- **Each compiler supports different levels of parallelism**
  - LLVM/clang 10, AMD, CCE9, IBM, PGI: teams, parallel
  - (Planned) LLVM/Clang 11, Intel: teams, parallel, simd
  - CCE8: teams, parallel or teams, simd
- **Caveats:**
  - *Real applications* will have algorithms that are structured such that they can't immediately use the combined construct.
  - It may also make collapse hard to do
  - Performance can be achieved without combined directives, but likely won't be portable

```
#pragma omp target teams distribute parallel for simd
  for (int i = 0; i < n; i++) {
     F(i) = G(i);
  }
```

Hardware and Software Mapping

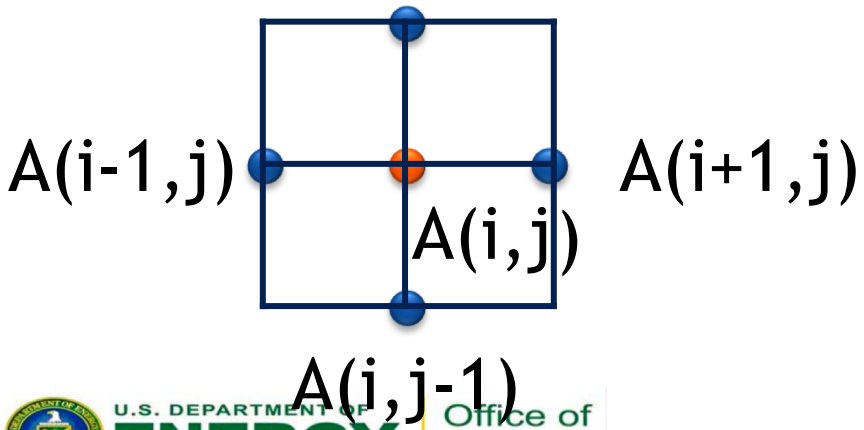
GPU Hardware	OpenACC	OpenMP	CUDA	OpenCL
Thread Group	Gang	Team	Thread Block	Work Group
EU Thread	Worker	Thread	Worker Thread	Wave Front
SIMD Lane	Vector	SIMD	Warp	Thread

- For OpenACC, recommend to use “acc loop” to let the compiler choose number of gangs, workers, and vectors.
- If you choose manually, recommend to use gang for most outer loop, and vector for most inner loop. Also to use the vector length a multiple of 32, which is the warp size.
- For OpenMP, recommend to use the combined contructs syntax and let the compiler choose number of teams, threads, etc.

EXAMPLE: Solve Laplace Equation

- Example: Solve Laplace equation in 2D: $\nabla^2 f(x, y) = 0$
- Use Jacobi solver to iteratively update the value (e.g. Temperature) at each point from the average of neighboring points, until it converges
- Common, useful algorithm

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$



Laplace Equation: C Code

```
while ( err > tol && iter < iter_max ) {
  err=0.0;


  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }


  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

Slide and example code courtesy of Jeff Larkin, Eric Wright, NVidia

reduction Clause

- Common situation: combine values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed
- We can use "reduction" clause in both OpenMP and OpenACC

```
double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
 for (i=0;i< MAX; i++) {
        ave + = A[i];
 }
 ave = ave/MAX;
```

- **Syntax: Reduction (operator : list).**
- **Reduces list of variables into one, using operator.**
- **Reduced variables must be shared variables.**
- **Allowed Operators:**
  - **Arithmic: + - * / # add, substract, multiply, divide**
  - **Fortran intrinsic: max min**
  - **Bitwise: & | ^ # and, or, xor**
  - **Logical: && || # and, or**

collapse clause

FORTRAN example:

```
!$acc loop collapse (2)
   do i = 1, 1000
      do j = 1, 100
        a(i,j) = b(i,j) + c(i,j)
      enddo
   enddo
!$acc end loop
```

FORTRAN example:

```
!$omp do collapse(2)
   do i = 1, 1000
      do j = 1, 100
        a(i,j) = b(i,j) + c(i,j)
      enddo
   enddo
!$omp end do
```

- **collapse *(n)* collapses the *n* nested loops into 1 large loop, then schedule work for each thread accordingly.**

OpenACC Parallel

```
while ( err > tol && iter < iter_max ) {
  err=0.0;

#pragma acc parallel loop reduction(max:err)collapse(2)
for( int j = 1; j < n-1; j++) {
   for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }

#pragma acc parallel loop collapse(2)
for( int j = 1; j < n-1; j++) {
   for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
      }
  }
  iter++;
}
```

Example code courtesy of Jeff Larkin, Eric Wright, NVidia

CUDA Managed Memory (Unified Shared Memory)

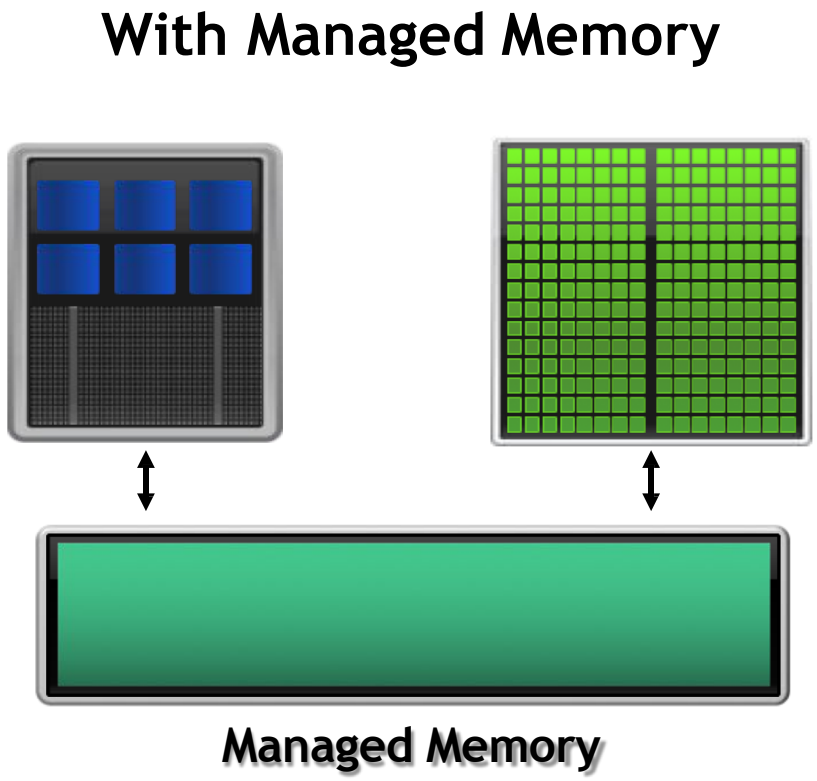


Image from NVidia

- Single address space over CPU and GPU memories
- Compiler manages data migration between CPU and GPU memories - no need to explicitly copy data
- Usually slower than explicitly memory management
- OpenACC: Currently only available from PGI on NVIDIA GPUs. Enabled via a compiler flag
- OpenMP 4.5: allocate use cudaMallocManaged
  OpenMP 5.0: language feature

```
#pragma omp requires unified_shared_memory
for (k=0; k < NTIMES; k++) {
#pragma omp target teams distribute parallel for simd
  for (j=0; j < N; j++) {
    a[j] = b[j] + scalar * c[j];
  }
}
```

OpenACC PGI, Use Managed Memory Example

% pgcc -fast -ta=tesla:cc70 -Minfo=accel laplace2d.c jacobi.c

```
PGC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index for symbol - A
(laplace2d.c: 37)
calcNext:
     37, Generating Tesla code
         38, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:error)
         41, #pragma acc loop vector(128) /* threadIdx.x */
     38, Accelerator restriction: size of the GPU copy of Anew,A is unknown
     41, Loop is parallelizable
PGC-F-0704-Compilation aborted due to previous errors. (laplace2d.c)
PGC/x86-64 Linux 19.10-0: compilation aborted
```

Without managed memory, compilation fails due to the size of GPU copy is unknown (the parallel code has no explicit data copy)

% pgcc -fast -ta=tesla:cc70,managed -Minfo=accel laplace2d.c jacobi.c

```
 calcNext:
     37, Generating Tesla code
         38, #pragma acc loop gang /* blockIdx.x */
             Generating reduction(max:error)
         41, #pragma acc loop vector(128) /* threadIdx.x */
     37, Generating implicit copyin(A[:]) [if not already present]
         Generating implicit copy(error) [if not already present]
         Generating implicit copyout(Anew[:]) [if not already present]
     41, Loop is parallelizable
```

Total 1.102 sec

GPU activities:

Time(%)	Time	Calls	Avg	Min	Max	Name
57.88%	622.48ms	1000	622.48us	542.20us	60.732ms	calcNext_37_gpu
41.11%	442.20ms	1000	442.20us	430.40us	454.84us	swap_53_gpu
0.74%.	7.9702ms	1000	7.9700us	7.4880us	9.5360us	calcNext_37_gpu__red
0.14%	1.5076ms	1000	1.5070us	1.4400us	3.6480us	[CUDA memcpy DtoH]
0.13%	1.3761ms	1000	1.3760us	1.3120us	1.9840us	[CUDA memset]

Almost no memory copy cost, with managed memory

Data Clauses

Multi-dimensional Array shaping

OpenACC		OpenMP
`copy(array[0:N][0:M])`	C/C++	`map(tofrom:array[0:N][0:M])`
`copy(array(1:N,1:M))`	Fortran	`map(tofrom:copy(array(1:N,1:M))`

C/C++: starting_index : length
Fortran: starting_index : ending_index

- **Data clauses allow the programmer to tell the compiler which data to move an when between the host and device.**
- **Data clauses may be added to kernels or parallel regions, but also data, enter data, and exit data.**

OpenACC and OpenMP data Clauses

OpenACC	OpenMP	Meaning
copy (var)	map (tofrom: var)	Allocate on GPU, copy from host to GPU when enter and copy from GPU to host when exit
copyin (var)	map (to: var)	Allocate on GPU, copy from host to GPU when enter
copyout (var)	map (from: var)	Allocate on GPU, copy from GPU to host when exit
create (var)	map (alloc: var)	Allocate on GPU but does not copy
delete (var)	map (delete: var)	Delete from GPU
present (var)	present (var)	Data is present on GPU

OpenACC Parallel, Basic Data

```
#pragma acc parallel loop copy(A[:m*n],Anew[:m*n]) reduction(max:err) collapse(2)
for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

        Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                             A[j-1][i] + A[j+1][i]);
        err = max(err, abs(Anew[j][i] - A[j][i]));
      }
    }

  #pragma acc parallel loop collapse(2)
    for( int j = 1; j < n-1; j++) {
       for( int i = 1; i < m-1; i++ )
      {
        A[j][i] = Anew[j][i];
      }
    }
```

Example code courtesy of Jeff Larkin, Eric Wright, NVidia

OpenACC Parallel, Basic Data

```
% pgcc -acc -fast -ta=tesla:cc70 -Minfo=accel,opt laplace2d.c jacobi.c
calcNext:
    37, Generating copy(A[:n*m]) [if not already present]
        Generating Tesla code
        38, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
        40,   /* blockIdx.x threadIdx.x collapsed */
            Generating reduction(max:error)
    37, Generating implicit copy(error) [if not already present]
        Generating copy(Anew[:n*m]) [if not already present]
swap:
    52, Generating copy(Anew[:n*m],A[:n*m]) [if not already present]
        Generating Tesla code
        53, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
        55,   /* blockIdx.x threadIdx.x collapsed */
jacobi.c:
```

Data copy at each loop.
98% time spent in data move

Total: 206.875 sec

Time(%)	Time	Calls	Avg	Min.	Max	Name
52.23%	46.1974s	33000	1.3999ms	1.2480us	5.9465ms	[CUDA memcpy HtoD]
46.60%	41.2126s	37000	1.1139ms	1.8240us	1.5382ms	[CUDA memcpy DtoH]
0.65%	572.06ms	1000	572.06us	556.35us	590.68us	calcNext_36_gpu
0.51%	452.57ms	1000	452.57us	444.99us	464.09us	swap_52_gpu
0.01%	7.8517ms	1000	7.8510us	7.4550us	9.2160us	calcNext_44_gpu__red

Structured Data Directive

OpenACC

```
#pragma acc data map(to:A, B) map(from: C)
{
    #pragma acc parallel
        {do lots of stuff with A, B and C}

    {do something on the host}

    #pragma acc parallel
        {do lots of stuff with A, B, and C}
}
```

OpenMP

```
#pragma omp target data map(to:A, B) map(from: C)
{
    #pragma omp target
        {do lots of stuff with A, B and C}

    {do something on the host}

    #pragma omp target
        {do lots of stuff with A, B, and C}
}
```

OpenACC Parallel, Structured Data

```
#pragma acc data copyin(A[:n*m]) create(Anew[:n*m])
while ( err > tol && iter < iter_max ) {
  err=0.0;

 #pragma acc parallel loop copy(A[:n*m],Anew[:n*m]) reduction(max:err) collapse(2)
   for( int j = 1; j < n-1; j++) {
     for(int i = 1; i < m-1; i++) {


       Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                            A[j-1][i] + A[j+1][i]);
       err = max(err, abs(Anew[j][i] - A[j][i]));
     }
   }

 #pragma acc parallel loop collapse(2)
   for( int j = 1; j < n-1; j++) {
     #pragma acc loop
     for( int i = 1; i < m-1; i++ ) {
       A[j][i] = Anew[j][i];
     }
   }
   iter++;
}
```

Example code courtesy of Jeff Larkin, Eric Wright, NVidia

OpenACC Parallel, Structured Data

```
% pgcc -acc -fast -ta=tesla:cc70 -Minfo=accel,opt laplace2d.c jacobi.c
laplace2d.c:
calcNext:
     37, Generating copy(A[:n*m]) [if not already present]
         Generating Tesla code
         38, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         40,   /* blockIdx.x threadIdx.x collapsed */
             Generating reduction(max:error)
     37, Generating implicit copy(error) [if not already present]
         Generating copy(Anew[:n*m]) [if not already present]
swap:
     52, Generating copy(Anew[:n*m],A[:n*m]) [if not already present]
         Generating Tesla code
         53, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         55,   /* blockIdx.x threadIdx.x collapsed */
jacobi.c:
main:
     59, Generating create(Anew[:m*n]) [if not already present]
         Generating copyin(A[:m*n]) [if not already present]
```

Now < 2% time spent in data move

Total 1.071 sec

Time(%)	Time	Calls	Avg	Min	Max	Name
44.06%	369.08ms	1000	369.08us	367.32us	372.12us	calcNext_37_gpu
43.01%	360.23ms	1000	360.23us	358.40us	362.04us	swap_52_gpu
11.29%	94.543ms	1000	94.542us	89.503us	102.72us	calcNext_37_gpu__red
1.30%	10.877ms	8	1.3596ms	1.3534ms	1.3708ms	[CUDA memcpy HtoD]
0.18%	1.5005ms	1000	1.5000us	1.4080us	2.8160us	[CUDA memcpy DtoH]
0.16%	1.3699ms	1000	1.3690us	1.3110us	1.9840us	[CUDA memset]

OpenMP Target, Fortran, Structured Data

```
!$omp target data map(to:A) map(alloc:Anew)
do while ( error .gt. tol .and. iter .lt. iter_max )
 !$omp target teams distribute parallel do simd collapse(2) map(to:A) map(from:Anew)
map(tofrom:error) reduction(max:error)
      do j=1,m-2
        do i=1,n-2
          Anew(i,j) = 0.25_fp_kind * ( A(i+1,j  ) + A(i-1,j  ) + &
                                       A(i  ,j-1) + A(i  ,j+1) )
          error = max( error, abs(Anew(i,j)-A(i,j)) )
        end do
      end do
!$omp end target teams distribute parallel do simd

!$omp target teams distribute parallel do simd collapse(2) map(to:Anew) map(from:A)
      do j=1,m-2
        do i=1,n-2
          A(i,j) = Anew(i,j)
        end do
      end do
  !$omp end target teams distribute parallel do

  iter = iter + 1
end do
!$omp end target data
```

Total 0.782 sec. Use CCE9

Time(%)	Time	Calls	Avg	Min	Max	Name
60.41%	595.91ms	1000	595.91us	586.91us	607.67us	jacobi_$ck_L47_1
36.49%	359.97ms	1000	359.97us	359.07us	361.24us	jacobi_$ck_L48_4
2.94%	29.016ms	1002	28.958us	1.3120us	27.652ms	[CUDA memcpy HtoD]
0.15%	1.4800ms	1000	1.4800us	1.4390us	1.8880us	[CUDA memcpy DtoH]

Unstructured Data Directive

OpenACC

```
void init_array(int *A, int N) {
  for (int i = 0; i < N; ++i)
    A[i] = i;
 #pragma acc enter data copyin (A[0:N])
}

int main(void) {
  int N = 1024;
  int *A = malloc(sizeof(int) * N);
  init_array(A, N);

  #pragma acc loop
  for (int i = 0; i < N; ++i)
    A[i] = A[i] * A[i];

 #pragma acc exit data copyout (A[0:N])
}
```

OpenMP

```
void init_array(int *A, int N) {
  for (int i = 0; i < N; ++i)
    A[i] = i;
 #pragma omp target enter data map(to: A[0:N])
}

int main(void) {
  int N = 1024;
  int *A = malloc(sizeof(int) * N);
  init_array(A, N);

  #pragma omp target teams distribute parallel for simd
  for (int i = 0; i < N; ++i)
    A[i] = A[i] * A[i];

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

Structured vs. Unstructured Data Directive

- **Structured**
  - Has explicit start and end points
  - Within a single function
  - Memory exist within the data region

- **Unstructured**
  - Can have multiple start and end points
  - Can branch across multiple functions
  - Memory exists until explicitly deallocated

OpenACC Parallel, Unstructured Data

```
void initialize( …)
{…
  #pragma acc enter data copyin(A[:m*n],Anew[:m*n])
}
while ( err > tol && iter < iter_max ) {
  err=0.0;
#pragma acc parallel loop copy(A[:n*m],Anew[:n*m]) reduction(max:err) collapse(2)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      err = max(err, abs(Anew[j][i] - A[j][i]));
    }
  }
  ..
  iter++;
}
void deallocate(..)
{#pragma acc exit data delete(A,Anew)
 ..}
```

Example code courtesy of Jeff Larkin, Eric Wright, NVidia

OpenACC Parallel, Unstructured Data

% pgcc -acc -fast -ta=tesla:cc70 -Minfo=accel,opt laplace2d.c jacobi.c

```
laplace2d.c:
initialize:
     33, Generating enter data copyin(Anew[:m*n],A[:m*n])
calcNext:
     39, Generating copy(A[:n*m]) [if not already present]
         Generating Tesla code
         40, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         42,   /* blockIdx.x threadIdx.x collapsed */
             Generating reduction(max:error)
     39, Generating implicit copy(error) [if not already present]
         Generating copy(Anew[:n*m]) [if not already present]
swap:
     54, Generating copy(Anew[:n*m],A[:n*m]) [if not already present]
         Generating Tesla code
         55, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         57,   /* blockIdx.x threadIdx.x collapsed */
deallocate:
     66, Generating exit data delete(Anew[:1],A[:1])
jacobi.c:
```

Total 0.863 sec

Time(%)	Time	Calls	Avg	Min	Max.	Name
43.52%	369.11ms	1000	369.11us	367.64us	372.70us	calcNext_39_gpu
42.47%	360.21ms	1000	360.21us	358.78us	362.14us	swap_54_gpu
11.12%	94.307ms	1000	94.307us	89.791us	103.17us	calcNext_39_gpu__red
2.56%	21.678ms	16	1.3549ms	1.3535ms	1.3627ms	[CUDA memcpy HtoD]
0.18%	1.4986ms	1000	1.4980us	1.4080us	2.8480us	[CUDA memcpy DtoH]
0.16%	1.3682ms	1000	1.3680us	1.3110us	1.9840us	[CUDA memset]

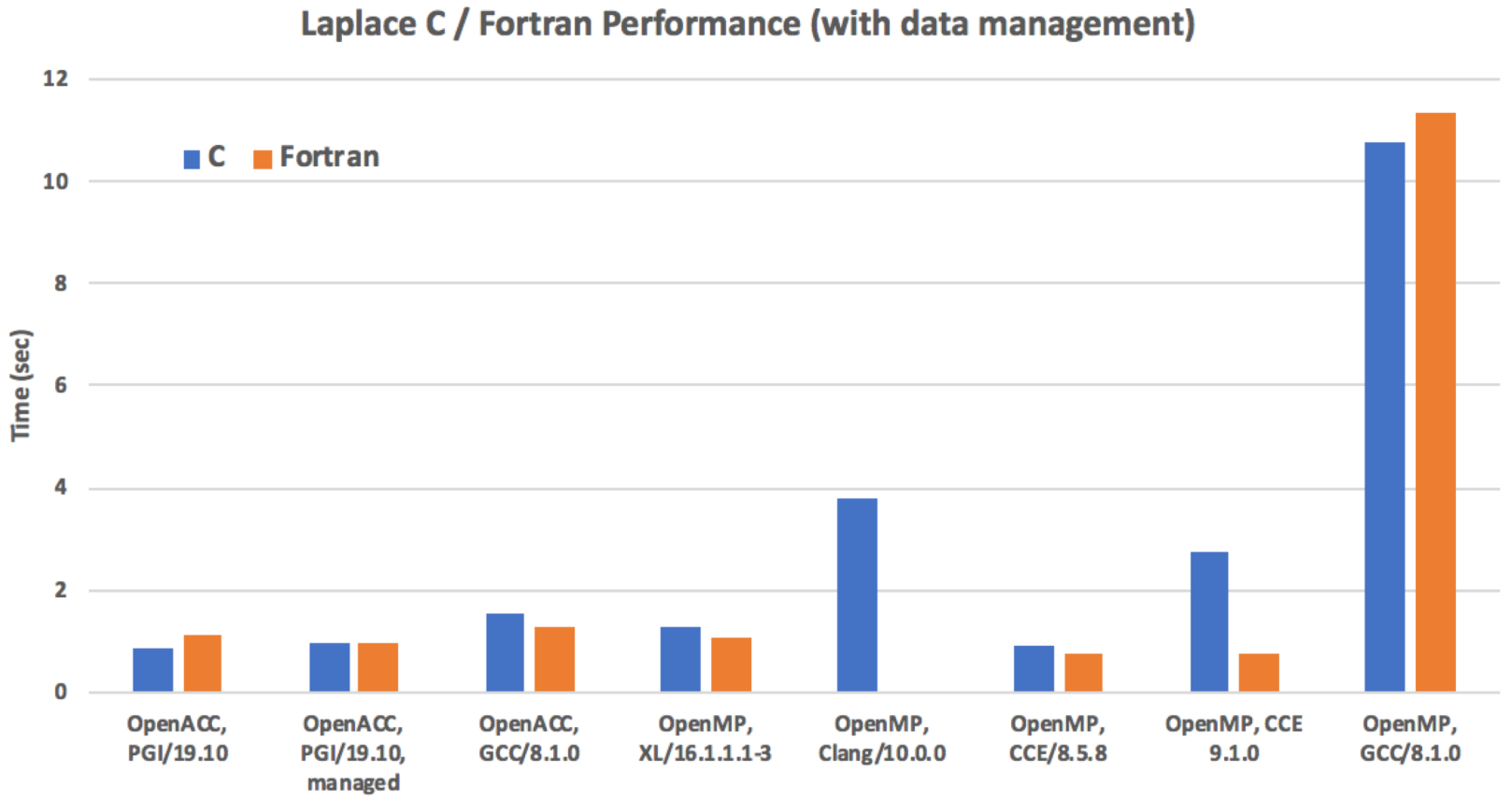
OpenMP Target, Fortran, Unstructured Data

```
!$omp enter data map(to:A) map(alloc:Anew)
 do while ( error .gt. tol .and. iter .lt. iter_max )

 !$omp target teams distribute parallel do simd collapse(2) map(to:A) map(from:Anew)
map(tofrom:error) reduction(max:error)
      do j=1,m-2
         do i=1,n-2
          Anew(i,j) = 0.25_fp_kind * ( A(i+1,j  ) + A(i-1,j  ) + &
                                       A(i  ,j-1) + A(i  ,j+1) )
          error = max( error, abs(Anew(i,j)-A(i,j)) )
        end do
      end do
!$omp end target teams distribute parallel do simd

!$omp target teams distribute parallel do simd collapse(2) map(to:Anew) map(from:A)
      do j=1,m-2
        do i=1,n-2
          A(i,j) = Anew(i,j)
        end do
      end do
!$omp end target teams distribute parallel do simd

!$omp exit data map(from:A) map(delete:Anew)
```

Total 0.777 sec. Use CCE9

Time(%)	Time	Calls	Avg	Min	Max	Name
58.85%	595.69ms	1000	595.69us	586.23us	607.67us	jacobi_$ck_L46_1
35.53%	359.68ms	1000	359.68us	357.88us	361.08us	jacobi_$ck_L47_4
2.86%	28.984ms	1002	28.925us	1.3120us	27.601ms	[CUDA memcpy HtoD]
2.75%	27.839ms	1001	27.811us	1.4400us	26.336ms	[CUDA memcpy DtoH]

Laplace Performance on Cori GPU (and Summit)



- Code optimized with explicit data management, but no further tuning.
- OpenMP XL is from Summit
- OpenMP CCE 9 Fortran, CCE 8 C/Fortran, and OpenACC PGI C are among the best.
- OpenACC GCC8 Fortran and OpenMP XL C are closer up.

OpenACC kernels vs. parallel

```
#pragma acc kernels
   for( int j=1; j<n-1; j++) {
       for(int i=1; i< m-1; i++)  {
        Anew[j][i] = …(A[j][i+1]) …
     }
  }
```

```
#pragma acc parallel
    #pragma acc loop
    for( int j=1; j<n-1; j++) {
        #pragma acc loop
        for(int i=1; i< m-1; i++){
           Anew[j][i] = …(A[j][i+1]) …
        }
    }
```

- **The kernels directive instructs the compiler to search for parallel loops in the code**
- **The compiler will analyze the loops and parallelize those it finds safe and profitable to do so. Correctness is guaranteed.**
- **The kernels directive can be applied to regions containing multiple loop nests. The compiler will attempt to parallelize all loops within the kernels region. Each loop can be parallelized/optimized in a different way**

Laplace, OpenACC Kernels. Not Managed, Runtime Error

```
% pgcc -acc -fast -ta=tesla:cc70 -Minfo=accel,opt laplace2d.c jacobi.c -o jacobi_kernels
  37, Accelerator restriction: size of the GPU copy of Anew,A is unknown
      Loop carried dependence of Anew-> prevents parallelization
      Loop carried dependence of Anew-> prevents vectorization
      Loop carried backward dependence of Anew-> prevents vectorization
      Generating implicit copyout(Anew[:]) [if not already present]
      Generating implicit copyin(A[:]) [if not already present]
  39, Loop is parallelizable
      Generating Tesla code
      37, #pragma acc loop seq
      39, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      43, Generating implicit reduction(max:error)
```

Runtime error:

call to cuMemcpyDtoHAsync returned error 700: Illegal address during kernel execution

call to cuMemFreeHost returned error 700: Illegal address during kernel execution

Laplace OpenACC Kernels, Managed

```
% pgcc -acc -fast -ta=tesla:cc70,managed -Minfo=accel,opt laplace2d.c jacobi.c -o jacobi_kernels_managed
    36, Generating implicit copyin(A[:]) [if not already present]
        Generating implicit copyout(Anew[:]) [if not already present]
    37, Loop carried dependence of Anew-> prevents parallelization
    Loop carried dependence of Anew-> prevents vectorization
    Loop carried backward dependence of Anew-> prevents vectorization
    39, Loop is parallelizable
    Generating Tesla code
    37, #pragma acc loop seq
    39, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    43, Generating implicit reduction(max:error)
```

Slower than the unmanaged version of unstructured data with 0.863 sec

Total: 1.68 sec

```
 Time(%)      Time     Calls       Avg       Min       Max  Nam
58.34%  975.07ms      1000 975.07us 847.70us  95.594ms calcNext_39_gpu
41.37%  691.33ms      1000 691.33us  682.30us 702.04us  swap_54_gpu
 0.13%  2.1075ms      1000 2.1070us  2.0470us 12.448us  calcNext_43_gpu__red
 0.08%  1.4173ms      1000 1.4170us  1.3760us 2.5280us  [CUDA memcpy DtoH]
 0.08%  1.2969ms      1000 1.2960us  1.2480us 1.9520us  [CUDA memcpy HtoD]
```

update Directive

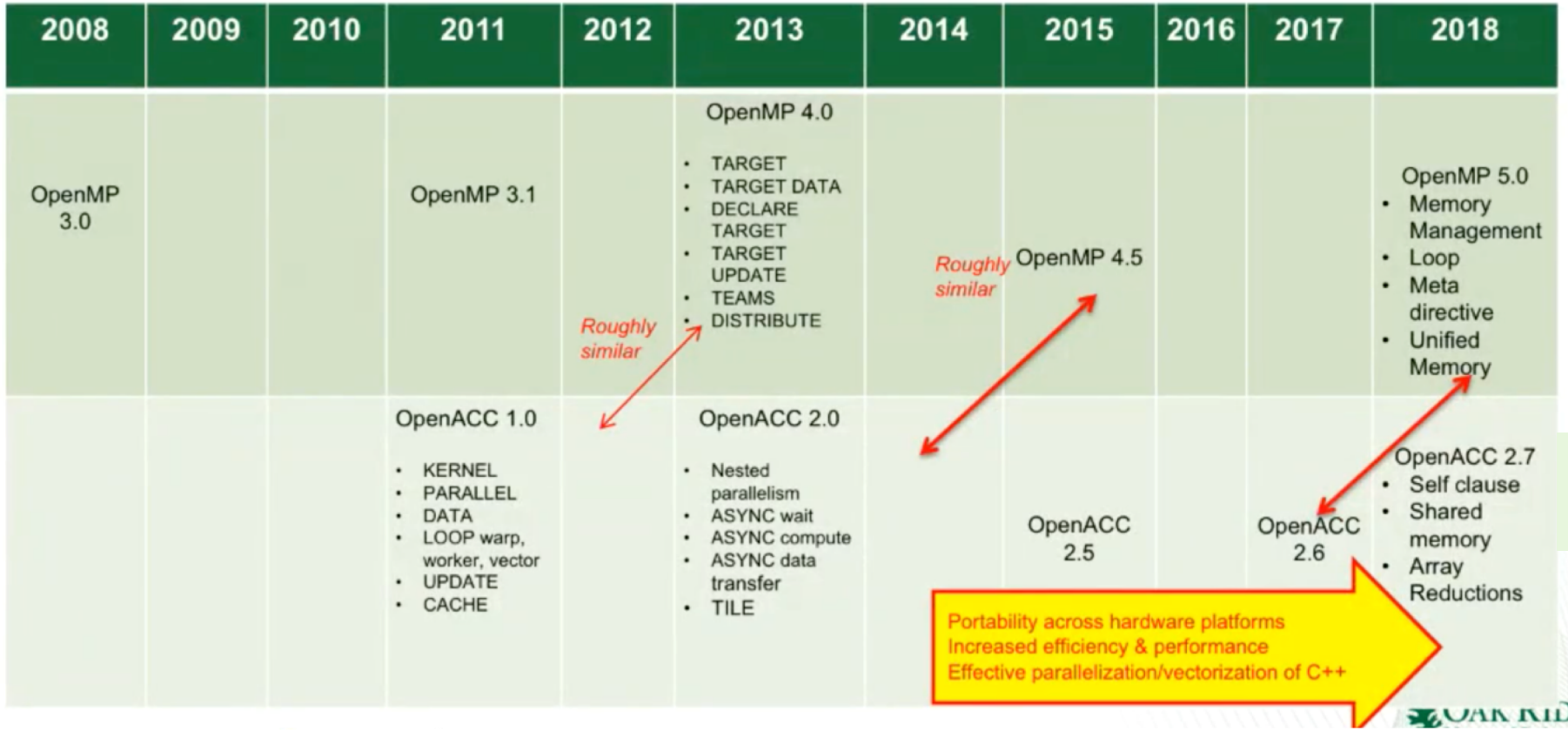
- **You can update data between host and device memory.**
- **Useful when you want to synchronize data in the middle of a data region**

```
#pragma acc data map(to: A,B) map(from: C)
{
    #pragma acc parallel
        {do lots of stuff with A, B and C}

    #pragma acc update self(A)

    host_do_something_with(A)

    #pragma scc update device(A)

    #pragma acc parallel
        {do lots of stuff with A, B, and C}
}
```

```
#pragma omp target data map(to: A,B) map(from: C)
{
    #pragma omp target
        {do lots of stuff with A, B and C}

    #pragma omp target update from(A)

    host_do_something_with(A)

    #pragma omp target update to(A)

    #pragma omp target
        {do lots of stuff with A, B, and C}
}
```

OpenACC and OpenMP Progress and Timeline

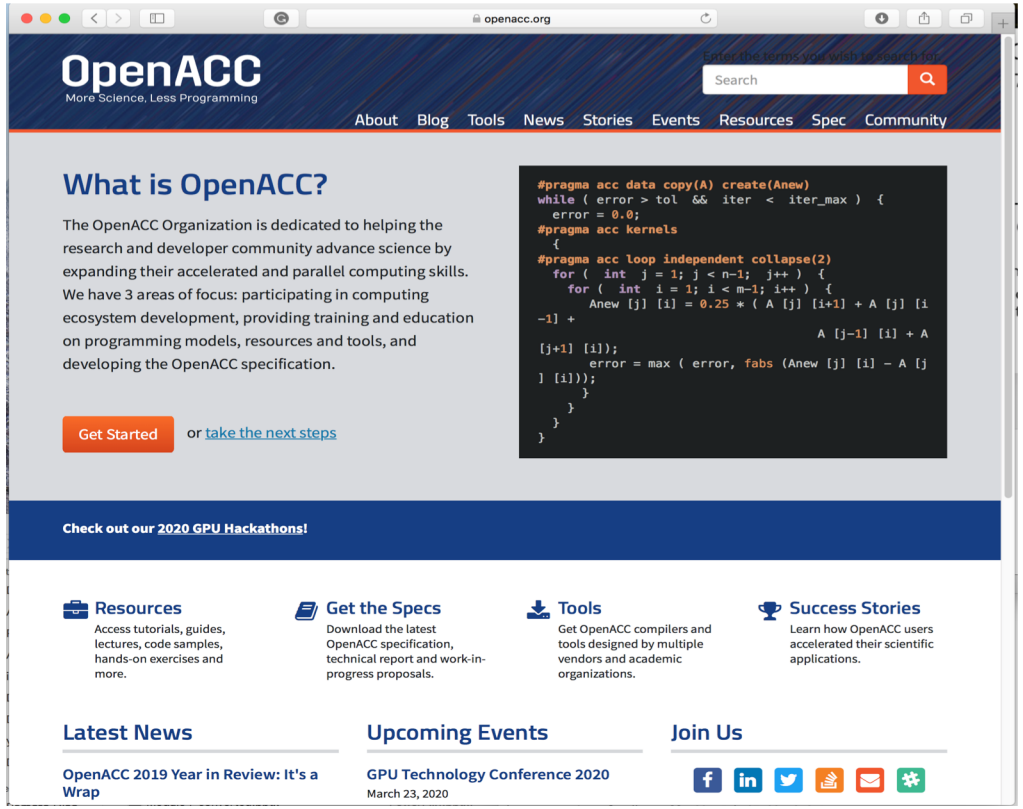
- **OpenACC started initially to focus on accelerator performance and quicker specification turnaround**
- **OpenMP target offload adopts important features from OpenACC**



OpenACC 3.0 (minor updates) released in Nov 2019

Table courtesy of Oscar Hernandez

OpenACC Resources and Compilers



Commercial Compilers

CRAY THE SUPERCOMPUTER COMPANY

Contact Cray Inc for more information.

PGI Community Edition

Annual license. Free download.

国家超级计算无锡中心 National Supercomputing Center in Wuxi

Contact National Supercomputing Center in Wuxi for more information.

Open Source Compilers

GCC 9
Includes initial support for OpenACC 2.6

Academic Compilers

RIKEN 筑波大学 University of Tsukuba

Omni compiler project, RIKEN/University of Tsukuba

OAK RIDGE National Laboratory

OpenARC, Oak Ridge National Laboratory

UNIVERSITY OF HOUSTON Stony Brook University

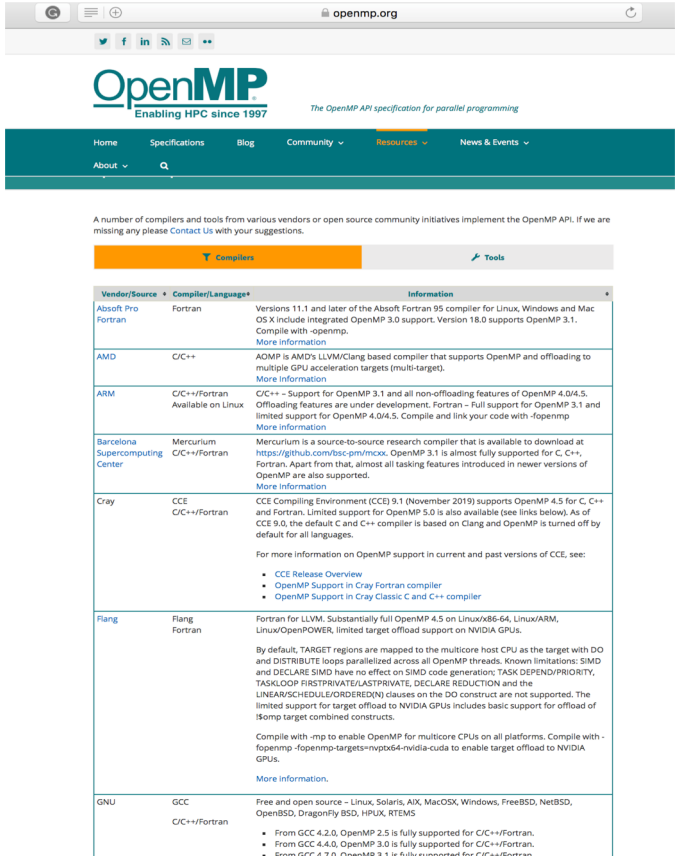
OpenUH, University of Houston, Stony Brook University

Lawrence Livermore National Laboratory
UNIVERSITY OF DELAWARE

ROSEACC, LLNL/University of Delaware

Available on Perlmutter:
-- PGI, GCC, Cray (deprecated since CCE9)
Available on other DOE systems:
-- PGI, GCC, Cray (deprecated)

OpenMP Resources and Compilers



Available on Perlmutter:
-- PGI, Cray, Clang, GCC, Flang
Available on other DOE systems:
-- Cray, Clang, GCC, PGI, Flang, IBM, Intel, AMD

The complete list is available at:
https://www.openmp.org/resources/openmp-compilers-tools/

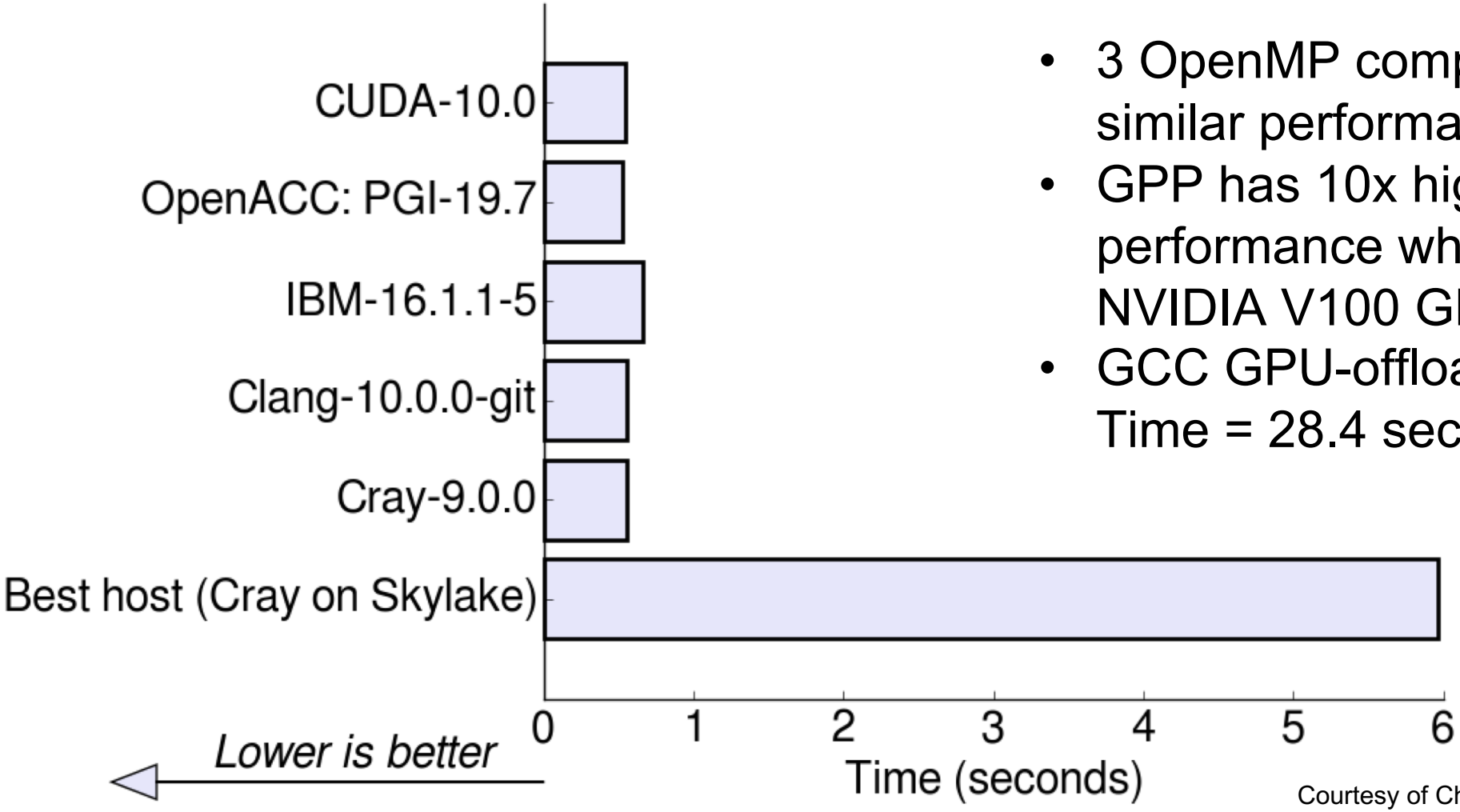
From DOE ECP OpenMP BoF

- **Critical for application developers to consider portable (and even better performance portable) solutions which can target different platforms across vendors.**
- **OpenMP is an open standard supported by nearly every vendor. Multiple Compilers will support a common set of OpenMP directives on GPUs**
  - LLVM/Clang 10
  - AMD (mostly tracks LLVM)
  - Cray (CCE 10)
  - IBM (XL V16.1.6)
  - Intel (Approximately 2021 timeframe),
  - Nvidia/PGI (Early 2021 for a production release through the NERSC/PGI NRE)

OpenACC <=> OpenMP Conversion

acc parallel	omp [target] teams
acc loop independent	omp loop
acc loop gang	omp distribute order(concurrent)
acc loop worker	omp parallel for order(concurrent)
acc loop vector	omp simd order(concurrent)
acc parallel loop	omp [target] teams loop
acc copyin(), copyout(), copy()	omp map(to:), map(from:), map(tofrom:)
acc data, acc end data	omp target data, omp end target data
acc enter data,acc exit data	omp target enter data, omp target exit data
acc update host(), acc update device()	omp target update to(), omp target update from()

GPP mini-app from BerkeleyGW



- 3 OpenMP compilers obtain similar performance to CUDA
- GPP has 10x higher performance when using a NVIDIA V100 GPU
- GCC GPU-offload not shown Time = 28.4 seconds

Courtesy of Chris Daley, Rahul Gayatri

ACCEL Benchmarks 1.2 (offload)

Summit
OpenMP -- XL 16.1.0
OpenACC -- PGI 18.3

Titan
OpenMP - CCE 8.7.0
OpenACC - PGI 18.4

SPEC ACCEL 1.2 Benchmark	Summit	Titan
Stencil (C)	1.12	0.67
LBM (C)	1.77	1.95
MRI-Q (C)	0.9	0.92
MD (Fortran)	0.14	
EP (Fortran)	0.27	1.07
CLVRLEAF (C,Fortran)	0.88	
CG (C)	0.32	1.12
SEISMIC (C)	1.49	
SP (Fortran)	1.27	
SP (C)	0.21	0.54
MinGhost (C,Fortran)	2.18	
LBDC (Fortran)	2.45	
Swim (Fortran)	1.18	
BT (C)	0.12	0.7

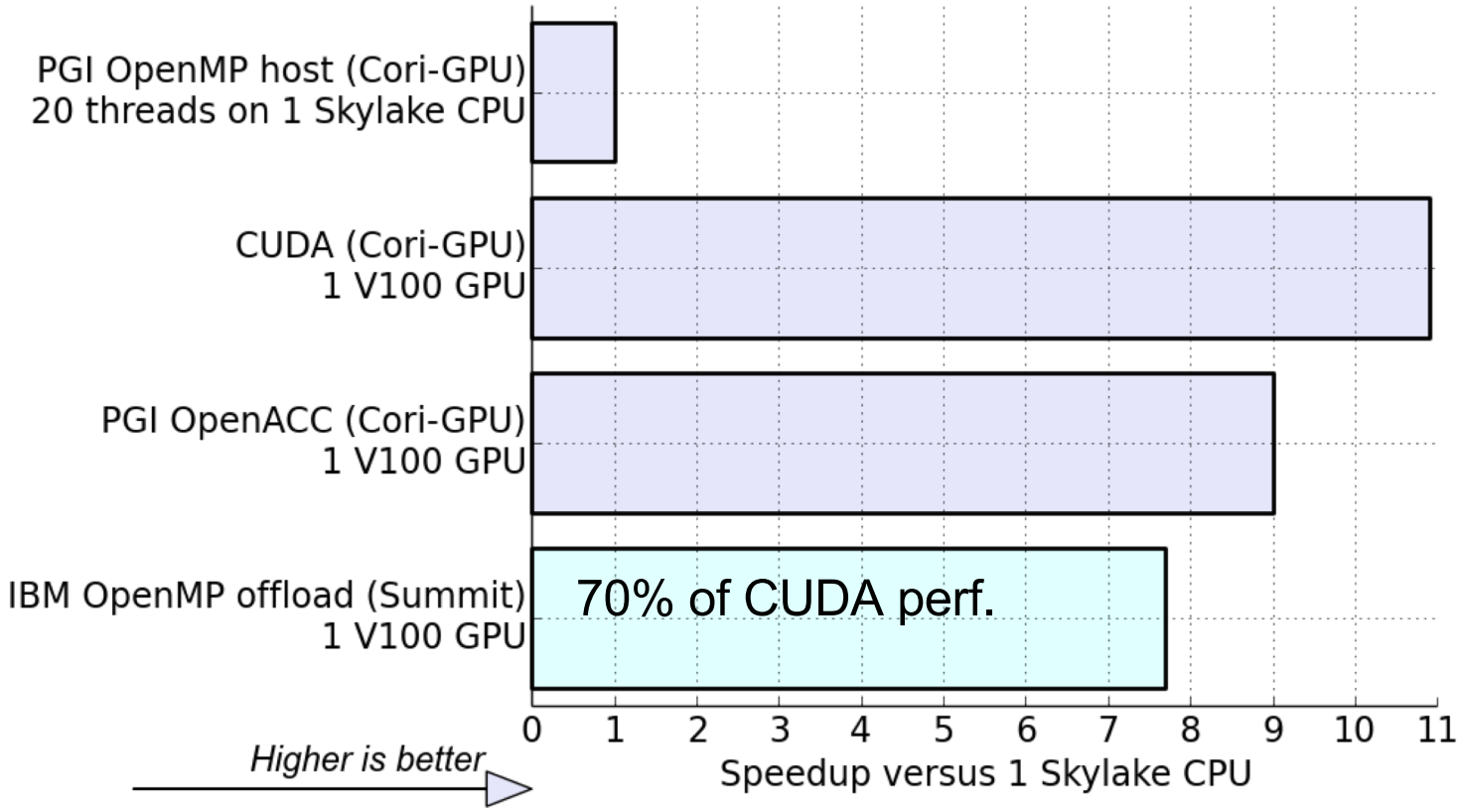
OpenMP is Better

OpenACC is Better

Update from Swen Boehm, Swaroop Pophale, Veronica G. Vergara Larrea, and Oscar Hernandez. "Evaluating Performance Portability of Accelerator Programming Models using SPEC ACCEL 1.2 Benchmarks" P3MA, ISC'2018

Snapshot of default compilers
September 2018

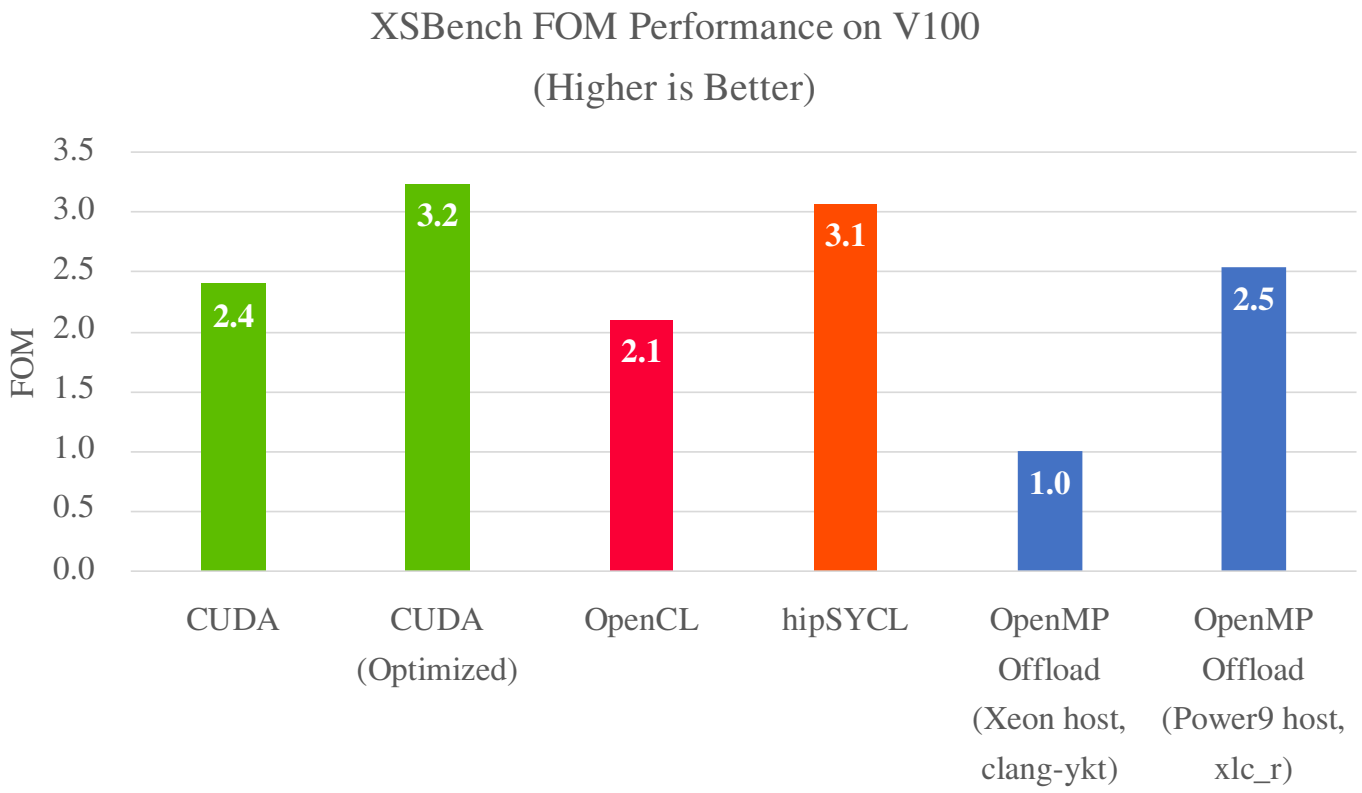
HPGMG-FV



- Initial OpenMP offload performance with IBM compiler is encouraging
- Easy to translate from CUDA launch configuration to OpenACC / OpenMP loops
- Several compiler bugs found with strict correctness tests

Courtesy of Chris Daley

Monte Carlo Particle Transport Case Study: XSBench



- OpenMP offloading relatively easy to implement
- Available compilers worked well – no bugs.
- IBM XL compiler generated fast code (better than naive CUDA implementation!)
- However, clang-ykt was 2.5x slower than IBM XL

Courtesy of John Tramm

Best Practices

- Profile the sequential code, choose hot spots, give enough work for GPU to do. Offload computational intensive work, such as large loops
- Use compiler hint, and remove loop carried dependencies to help parallelizing loops.
- Reorder loops or transpose arrays if needed to expose SIMD/SIMT in outer most loops.
- Keeping data resident on the device for the greatest possible time.
- Collapsing loops with the collapse clause, so there is a large enough iteration space to saturate the device.
- To utilize warps in OpenACC, always make sure vector length is a multiple of 32
- Try different compilers, such as PGI for OpenACC, XL/CCE/Clang/GCC for OpenMP.

Summary

- OpenACC has been used widely in many applications (since Titan). Mature implementation for Nvidia GPU.
- More OpenMP compilers available than OpenACC.
- OpenMP 5.0 compilers will have more mature support for accelerators, expect to see great improvements in quality and more devices to use.
- Performance with OpenMP 4.5 is on par or catching up with OpenACC and native CUDA.
- PGI OpenMP upcoming, leverage OpenACC implementation expertise.
- Transition from OpenACC to OpenMP is relatively straightforward.
- OpenMP 5.0 improvement provides more convenient porting from OpenACC to OpenMP.

Recommendation

- OpenACC is good for Nvidia GPUs, on Summit and Perlmutter, for example.
- OpenMP is good for all DOE Exascale computers, such as Perlmutter, ORNL Frontier, and ANL Aurora.
- NERSC users have used OpenMP for past systems, easier to use OpenMP for continuity. MPI + OpenMP is a successful programming model for Cori.
- Recommend to use OpenMP on Perlmutter, especially with PGI now on board (timing lines up with when Perlmutter is arriving).

Thank you.