Directive-based Programming for Highly-scalable Nodes

Doug Miles
Michael Wolfe

PGI Compilers & Tools
NVIDIA

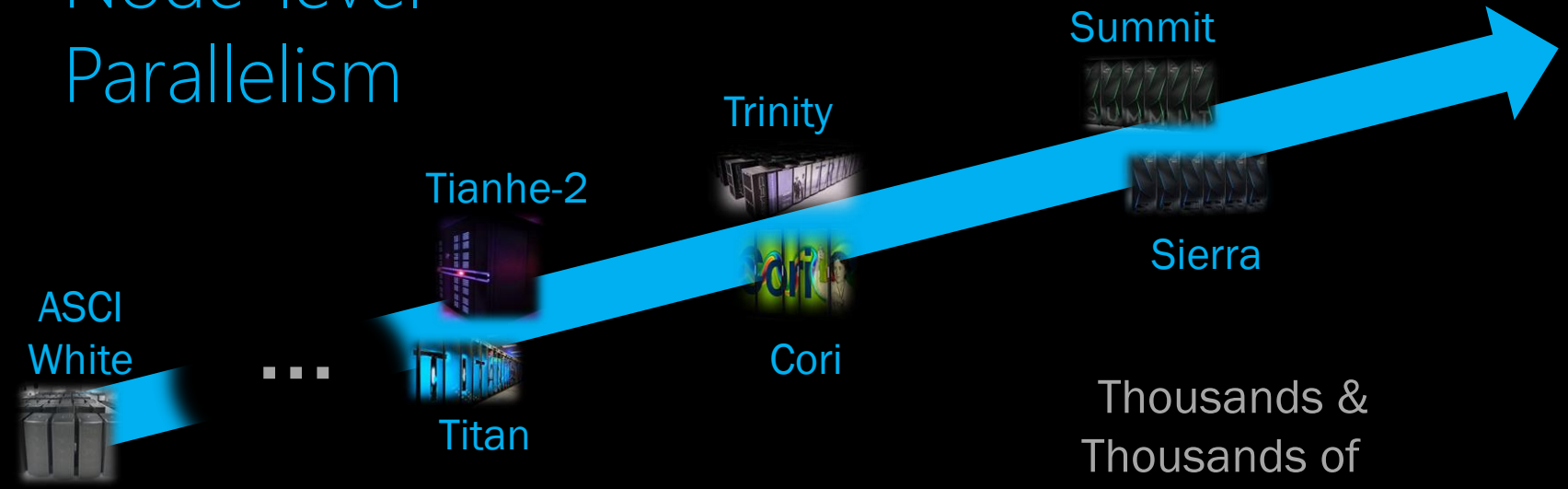
Cray User Group Meeting
May 2016

PGI®

Talk Outline

- Increasingly Parallel Nodes
- Exposing Parallelism - the Role of Directives
- Expressing Parallelism in OpenMP and OpenACC
- Using High-bandwidth Memory
- Directives and Scalability

Node-level Parallelism

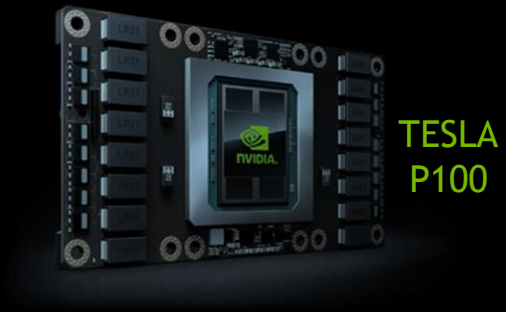


PGI®

The Latest HPC Processors

22 Cores
88 FP64 lanes
176 FP32 lanes

72 Cores
1152 FP64 lanes
2304 FP32 lanes



56 SMs
1792 FP64 lanes
3584 FP32 lanes

PGI®

Exposing parallelism in your code.
Fill the Lanes!

Exposing parallelism: OpenACC KERNELS as a porting tool

```
!$acc kernels
do j = 1, m
  do i = 1, n
       a(j,i) = b(j,i)*alpha +
                c(i,j)*beta
  enddo
enddo

...

!$acc end kernels
```

```
% pgf90 a.f90 -ta=multicore -c -Minfo
sub:
  10, Loop is parallelizable
      Generating Multicore code
      10, !$acc loop gang
  11, Loop is parallelizable
```

CPU

```
% pgf90 a.f90 -ta=tesla -c -Minfo
sub:
   9, Generating present(a(:,:),b(:,:),c(:,:))
  10, Loop is parallelizable
  11, Loop is parallelizable
      Accelerator kernel generated
      Generating Tesla code
      10, !$acc loop gang, vector(4)
      11, !$acc loop gang, vector(32)
```

GPU

PGI®

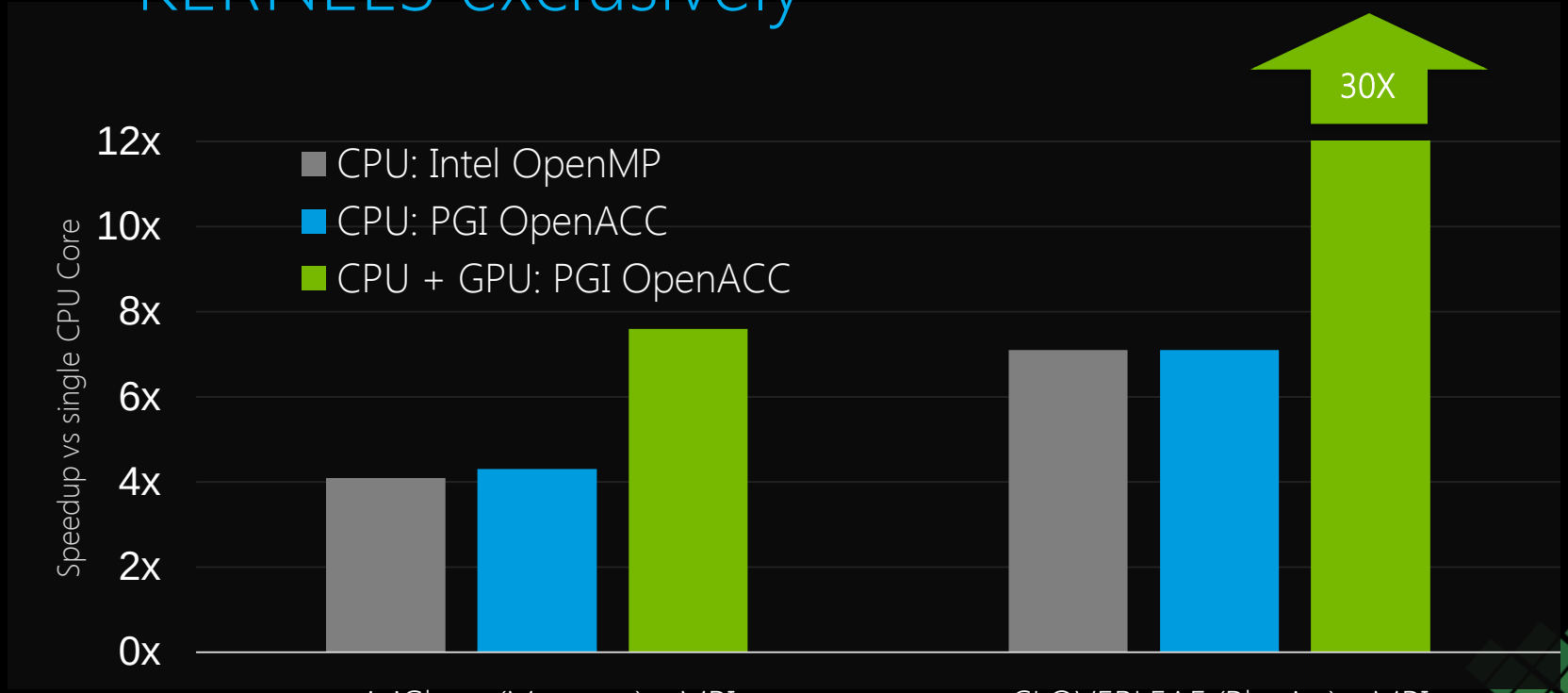
OpenACC KERNELS as a construct

```
% pgfortran –fast –acc –Minfo -c advec_mom_kernel.f90
advec_mom_kernel:

    ...

    167, Loop is parallelizable
    169, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        167, !$acc loop gang, vector(4) ! blockidx%y
                                        ! threadidx%y
        169, !$acc loop gang, vector(32) ! blockidx%x
                                         !threadidx%x
    ...
```

```
 96 !$ACC KERNELS
    ...
167 !$ACC LOOP INDEPENDENT
168    DO k=y_min,y_max+1
169 !$ACC LOOP INDEPENDENT PRIVATE(upwind,downwind,donor,dif,sigma,
                                  width,limiter,vdiffuw,vdiffdw,auw,adw,wind)
170      DO j=x_min-1,x_max+1
171        IF(node_flux(j,k).LT.0.0)THEN
172          upwind=j+2
173          donor=j+1
174          downwind=j
175          dif=donor
176        ELSE
177          upwind=j-1
178          donor=j
179          downwind=j+1
180          dif=upwind
181        ENDIF
182        sigma=ABS(node_flux(j,k))/(node_mass_pre(donor,k))
183        width=celldx(j)
184        vdiffuw=vel1(donor,k)-vel1(upwind,k)
185        vdiffdw=vel1(downwind,k)-vel1(donor,k)
186        limiter=0.0
187        IF(vdiffuw*vdiffdw.GT.0.0)THEN
188          auw=ABS(vdiffuw)
189          adw=ABS(vdiffdw)
190          wind=1.0_8
191          IF(vdiffdw.LE.0.0) wind=-1.0_8
192          limiter=wind*MIN(width*((2.0_8-sigma)*adw/width+(1.0_8+sigma)*
                              auw/celldx(dif))/6.0_8,auw,adw)
193        ENDIF
194        advec_vel(j,k)=vel1(donor,k)+(1.0-sigma)*limiter
195        mom_flux(j,k)=advec_vel(j,k)*node_flux(j,k)
196      ENDDO
197    ENDDO
     ...
```

PGI

The OpenACC version of CloverLeaf uses KERNELS exclusively



PGI

359.miniGhost: CPU: Intel Xeon E5-2698 v3, 2 sockets, 32-cores total, GPU: Tesla K80 (single GPU)
CLOVERLEAF: CPU: Dual socket Intel Xeon CPU E5-2690 v2, 20 cores total, GPU: Tesla K80 both GPUs

OpenACC KERNELS as a construct …

```
% pgfortran –fast –acc –Minfo -c advec_mom_kernel.f90
advec_mom_kernel:

    ...

    167, Loop is parallelizable
    169, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
        167, !$acc loop gang, vector(4) ! blockidx%y
                                        ! threadidx%y
        169, !$acc loop gang, vector(32) ! blockidx%x
                                         !threadidx%x
    ...
```

```
 96 !$ACC KERNELS
    ...
167 !$ACC LOOP INDEPENDENT
168    DO k=y_min,y_max+1
169 !$ACC LOOP INDEPENDENT PRIVATE(upwind,downwind,donor,dif,sigma,
                                  width,limiter,vdiffuw,vdiffdw,auw,adw,wind)
170      DO j=x_min-1,x_max+1
171        IF(node_flux(j,k).LT.0.0)THEN
172          upwind=j+2
173          donor=j+1
174          downwind=j
175          dif=donor
176        ELSE
177          upwind=j-1
178          donor=j
179          downwind=j+1
180          dif=upwind
181        ENDIF
182        sigma=ABS(node_flux(j,k))/(node_mass_pre(donor,k))
183        width=celldx(j)
184        vdiffuw=vel1(donor,k)-vel1(upwind,k)
185        vdiffdw=vel1(downwind,k)-vel1(donor,k)
186        limiter=0.0
187        IF(vdiffuw*vdiffdw.GT.0.0)THEN
188          auw=ABS(vdiffuw)
189          adw=ABS(vdiffdw)
190          wind=1.0_8
191          IF(vdiffdw.LE.0.0) wind=-1.0_8
192          limiter=wind*MIN(width*((2.0_8-sigma)*adw/width+(1.0_8+sigma)*
                              auw/celldx(dif))/6.0_8,auw,adw)
193        ENDIF
194        advec_vel(j,k)=vel1(donor,k)+(1.0-sigma)*limiter
195        mom_flux(j,k)=advec_vel(j,k)*node_flux(j,k)
196      ENDDO
197    ENDDO
     ...
```

PGI

… and as a path to standard languages

Fortran 2015 DO CONCURRENT

- \+ True Parallel Loops
- \+ Loop-scope shared/private data
- − No support for reductions
- − No support for data regions

```
168     DO CONCURRENT k=y_min,y_max+1
169       DO CONCURRENT j=x_min-1,x_max+1 LOCAL(upwind,downwind,donor,dif,
                                                sigma,width,limiter,vdiffuw,
                                                vdiffdw,auw,adw,wind)
171         IF(node_flux(j,k).LT.0.0)THEN
172           upwind=j+2
173           donor=j+1
174           downwind=j
175           dif=donor
176         ELSE
177           upwind=j-1
178           donor=j
179           downwind=j+1
180           dif=upwind
181         ENDIF
182         sigma=ABS(node_flux(j,k))/(node_mass_pre(donor,k))
183         width=celldx(j)
184         vdiffuw=vel1(donor,k)-vel1(upwind,k)
185         vdiffdw=vel1(downwind,k)-vel1(donor,k)
186         limiter=0.0
187         IF(vdiffuw*vdiffdw.GT.0.0)THEN
188           auw=ABS(vdiffuw)
189           adw=ABS(vdiffdw)
190           wind=1.0_8
191           IF(vdiffdw.LE.0.0) wind=-1.0_8
192           limiter=wind*MIN(width*((2.0_8-sigma)*adw/width+(1.0_8+sigma)*
                               auw/celldx(dif))/6.0_8,auw,adw)
193         ENDIF
194         advec_vel(j,k)=vel1(donor,k)+(1.0-sigma)*limiter
195         mom_flux(j,k)=advec_vel(j,k)*node_flux(j,k)
196       ENDDO
197     ENDDO
        ...
```

PGI

Expressing parallelism in OpenMP and OpenACC. Fill the Lanes!

Prescribing vs Describing*

OpenMP

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

#pragma omp parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
#pragma omp simd
        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]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

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

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

CPU

OpenACC

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

#pragma acc parallel loop reduction(max:error)
    for( int j = 1; j < n-1; j++) {
#pragma acc loop reduction(max:error)
        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]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc parallel loop
    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];
        }
    }

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

CPU

PGI

* “Performance Portability Through Descriptive Parallelism”, Jeff Larkin, NVIDIA, DOE Centers of Excellence Performance Portability Meeting, April 29-21, 2016

Prescribing vs Describing*

OpenMP

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

#pragma omp target teams distribute parallel for \
  reduction(max:error) collapse(2) schedule(static,1)
    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]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma omp target teams distribute parallel for \
  collapse(2) schedule(static,1)
    for( int j = 1; j < n-1; j++) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

GPU

OpenACC

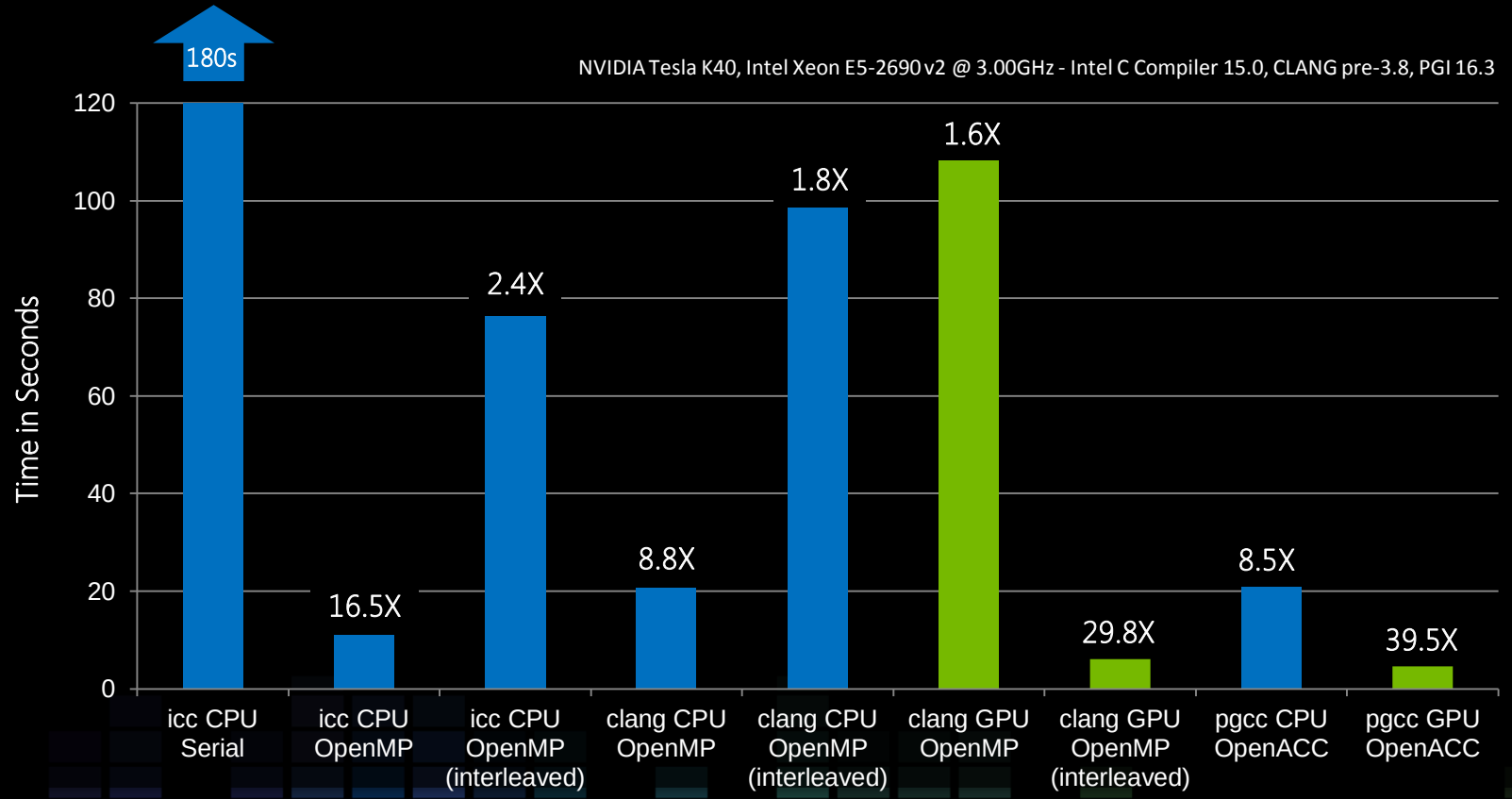
```
while ( error > tol && iter < iter_max )
{
    error = 0.0;

#pragma acc parallel loop reduction(max:error)
    for( int j = 1; j < n-1; j++) {
#pragma acc loop reduction(max:error)
        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]);
            error = fmax( error, fabs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc parallel loop
    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];
        }
    }

    if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

GPU

PGI

* “Performance Portability Through Descriptive Parallelism”, Jeff Larkin, NVIDIA, DOE Centers of Excellence Performance Portability Meeting, April 29-21, 2016

Performance Impact of Explicit OpenMP Scheduling*

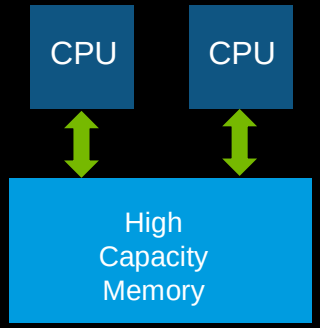


PGI®

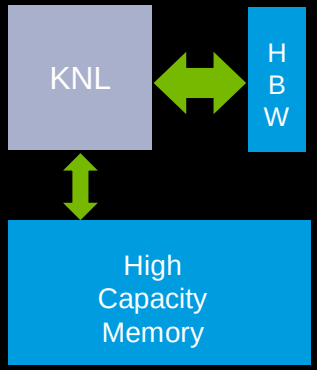
* "Performance Portability Through Descriptive Parallelism", Jeff Larkin, NVIDIA, DOE Centers of Excellence Performance Portability Meeting, April 29-21, 2016

Using High-bandwidth Memory

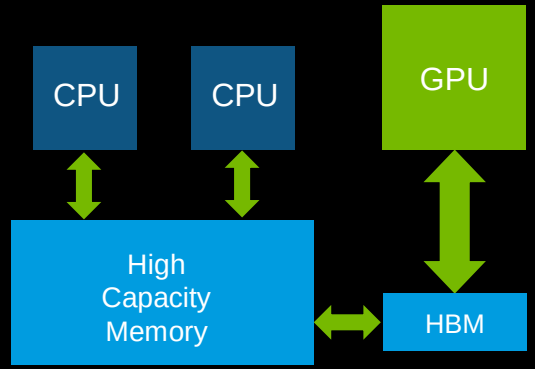
Using High-bandwidth Memory




```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma omp parallel for …
    for( int j = 1; j < n-1; j++) {
#pragma omp simd
        for( int i = 1; i < m-1; i++ ) {
        ...
        }
...
```



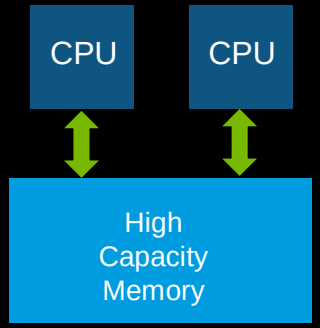

```
double *A = hbw_malloc (sizeof(double)*n*m)
double *Anew = hbw_malloc (sizeof(double)*n*m)
...
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma omp parallel for …
    for( int j = 1; j < n-1; j++) {
#pragma omp simd
        for( int i = 1; i < m-1; i++ ) {
        ...
        }
...
```



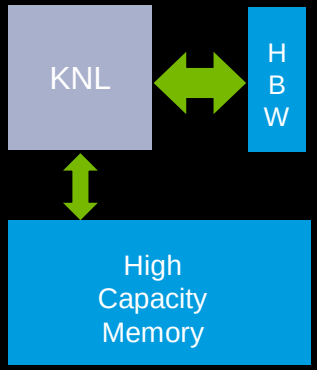

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma acc parallel loop …
    for( int j = 1; j < n-1; j++) {
#pragma acc loop
        for( int i = 1; i < m-1; i++ ) {
        ...
        }
...
```

PGI®

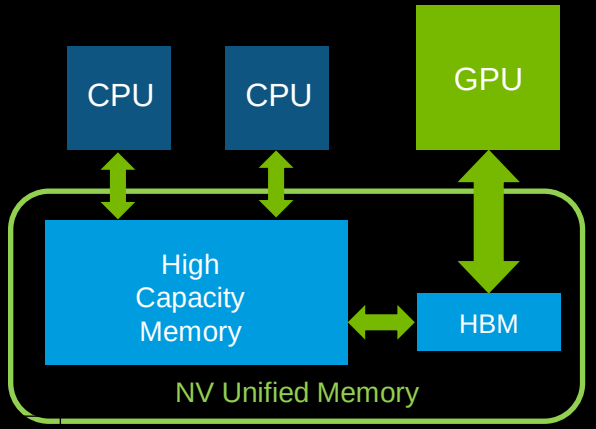
Using High-bandwidth Memory




```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma omp parallel for …
    for( int j = 1; j < n-1; j++) {
#pragma omp simd
        for( int i = 1; i < m-1; i++ ) {
        ...
        }
...
```



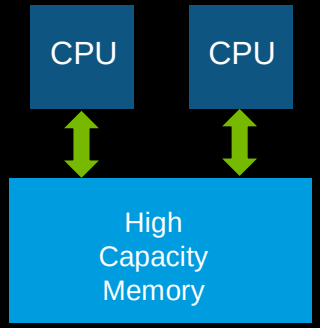

```
double *A = hbw_malloc (sizeof(double)*n*m)
double *Anew = hbw_malloc (sizeof(double)*n*m)
...
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma omp parallel for …
    for( int j = 1; j < n-1; j++) {
#pragma omp simd
        for( int i = 1; i < m-1; i++ ) {
        ...
        }
...
```



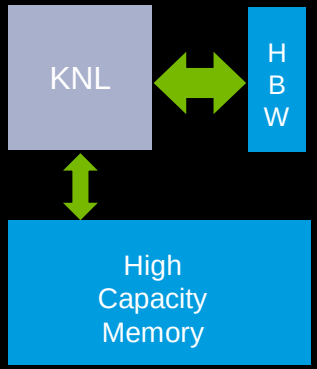

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma acc parallel loop …
    for( int j = 1; j < n-1; j++) {
#pragma acc loop
        for( int i = 1; i < m-1; i++ ) {
        ...
        }
...
```

PGI

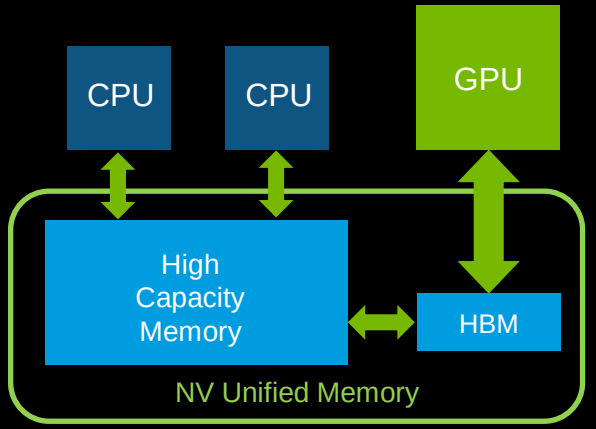
Using High-bandwidth Memory



OpenACC?



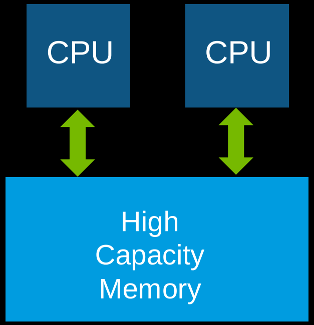
OpenACC?



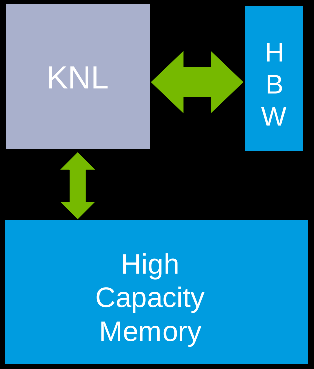

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma acc parallel loop …
    for( int j = 1; j < n-1; j++) {
#pragma acc loop
        for( int i = 1; i < m-1; i++ ) {
        ...
        }
...
```

PGI

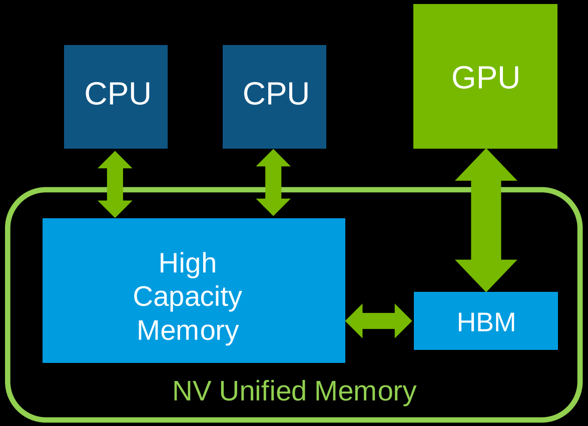
Using High-bandwidth Memory




```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma acc parallel loop …
    for( int j = 1; j < n-1; j++) {
#pragma acc loop
        for( int i = 1; i < m-1; i++ ) {
        ...
        }
...
```

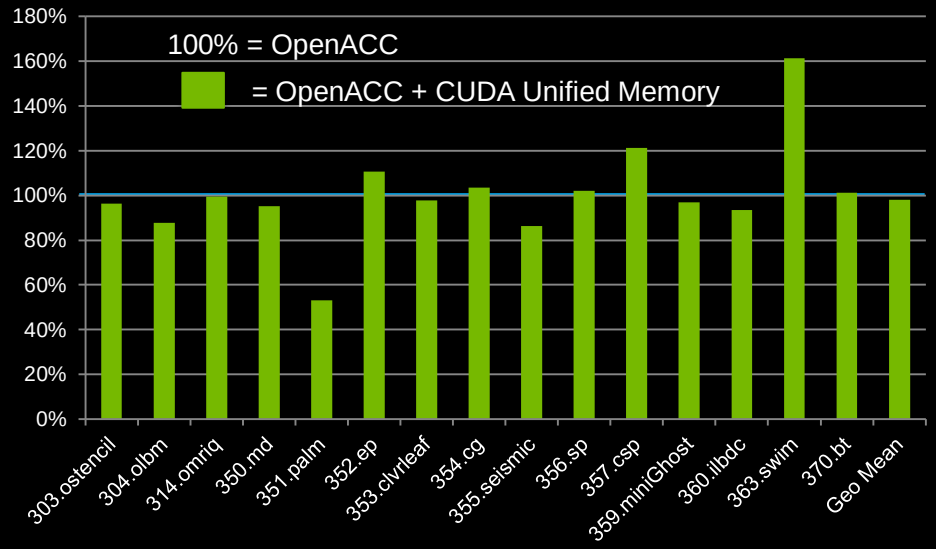



```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma acc parallel loop …
    for( int j = 1; j < n-1; j++) {
#pragma acc loop
        for( int i = 1; i < m-1; i++ ) {
        ...
        }
...
```




```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma acc parallel loop …
    for( int j = 1; j < n-1; j++) {
#pragma acc loop
        for( int i = 1; i < m-1; i++ ) {
        ...
        }
...
```

OpenACC and CUDA Unified Memory on Kepler



OpenACC directive-based data movement
vs
OpenACC + CUDA UM on Haswell+Kepler

- Kepler: only dynamic memory allocation uses CUDA Unified Memory
  - Pascal: can place Global, Stack, Static data in Unified Memory
- Kepler: Much explicit data movement eliminated, simplifies initial porting
  - Pascal: OpenACC user-directed data movement becomes an optimization
- Broadening GPU application space, improving performance
  - Pascal: Demand-paging
  - Pascal: Oversubscription
  - Pascal: Reduced Synchronization

PGI®

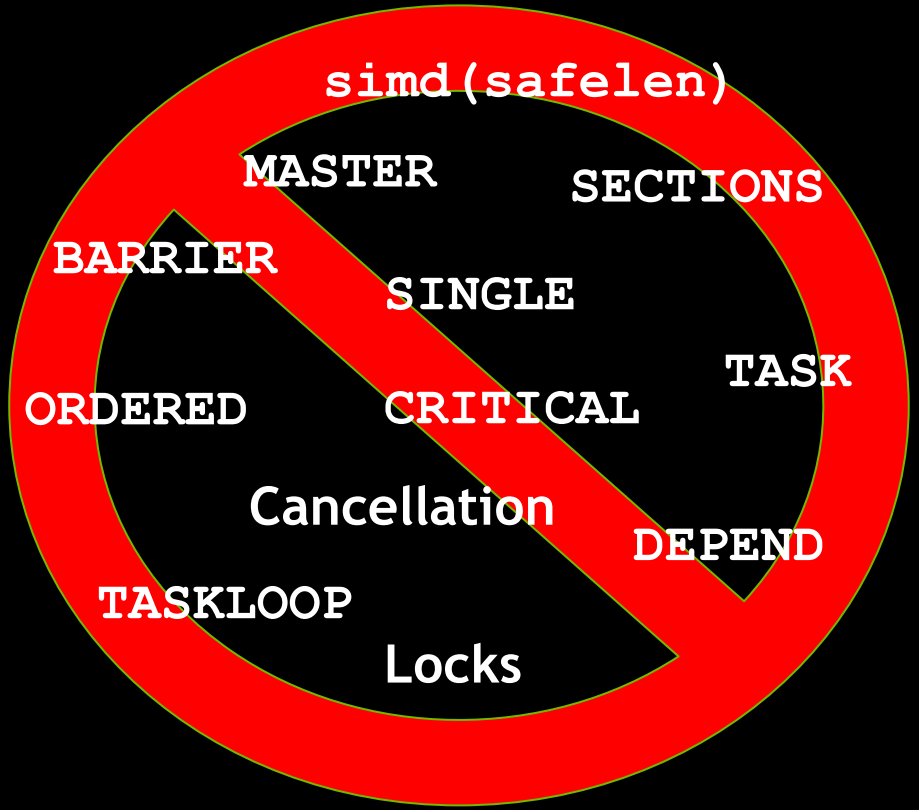
Directives and scalability – or not.

Synchronization in OpenMP and OpenACC

- Synchronization was *intentionally* minimized in the design of OpenACC: reductions & atomics only
- OpenMP is rich with synchronization features, most designed for modestly parallel systems
- NOTE: Fortran 2015 DO CONCURRENT and C++17 par_vec constructs offer little or no synchronization

Non-scalable OpenMP Features

Non-scalable OpenMP Features



PGI®

Concluding Thoughts

- Descriptive directives maximize productivity and performance portability
- We can (and should) use directives to manage high-bandwidth memory
- Legacy OpenMP synchronization features will inhibit on-node scalability – proceed with caution