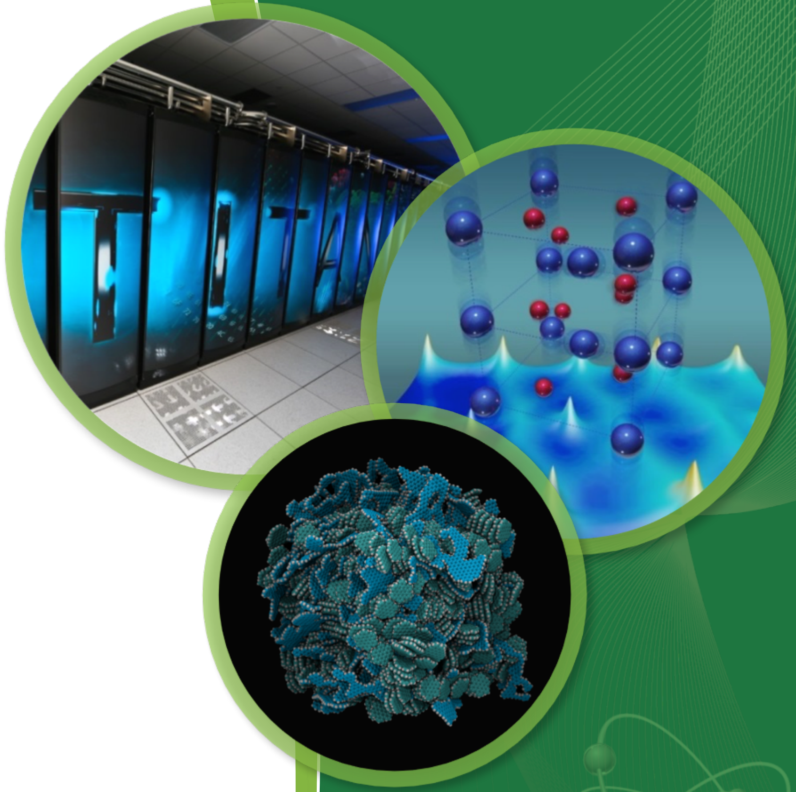
Towards Achieving Performance Portability Using Directives for Accelerators

M. Graham Lopez, Veronica Vergara Larrea, Wayne Joubert, Oscar Hernandez, Azzam Haidar, Stanimire Tomov, Jack Dongarra



WACCPD 2016

Salt Lake City, 14 Nov 2016

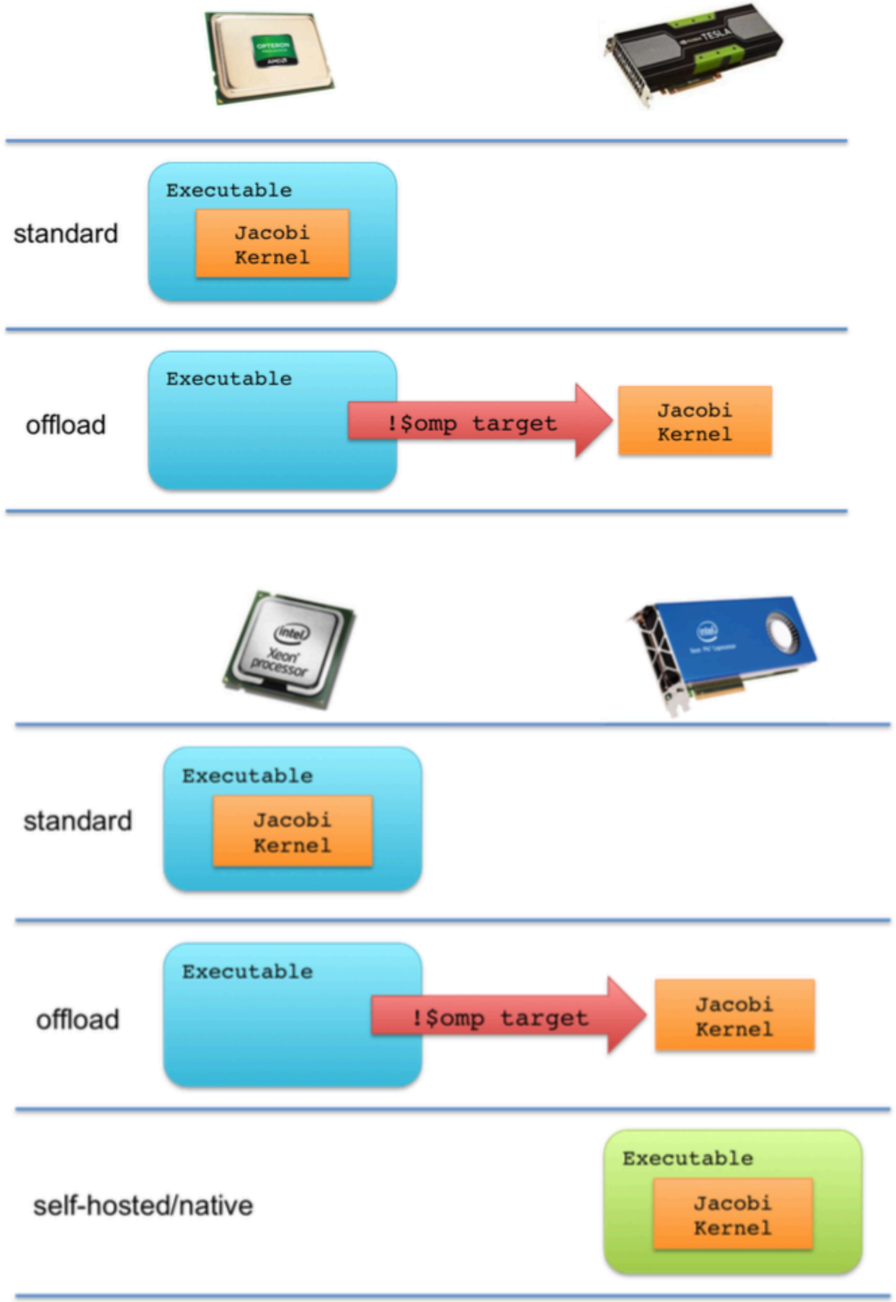
ORNL is managed by UT-Battelle
for the US Department of Energy

Directives for Performance Portability

- These are the expected solutions for CORAL
- Some questions remain:
  - What are the performance tradeoffs for using different “modes” in OpenMP?
  - Where are we -- what is the current* status of ”production ready” performance portability?

OpenMP 4.x – offloading

- There are (at least) now two models offered by the OpenMP standard:
  - "shared memory" (traditional 3.x)
  - "offload" (4.x support for discrete accelerators)
- How does a programmer pick and write performance portable code across both architectures?
  - shared memory is simpler (Intel)
  - are there tradeoffs for "offloading", even to self?



OpenMP 4.x – offloading

- Different capabilities on different architectures
- Best of both worlds? ”shared + target”

App. Kernel Version	Abbrev.	Executed On	Offloading To
shared memory	SM	CPU	n/a
		Xeon-Phi	n/a
shared+target	SM+t	CPU	CPU
		CPU	Xeon Phi
		CPU	GPU
		Xeon Phi	Xeon Phi
accelerator	accel	CPU	CPU
		CPU	Xeon Phi
		CPU	GPU
		Xeon Phi	Xeon Phi

OpenMP styles

```
do while (k.le.maxit .and. error.gt. tol)
 error = 0.0
 do j=1,m  !---Copy solution
  do i=1,n
   uold(i,j) = u(i,j)
  enddo
 enddo
 do j = 2,m-1 !---Update interior
  do i = 2,n-1
   r = &
      (ax*(uold(i-1,j) + uold(i+1,j)) &
     + ay*(uold(i,j-1) + uold(i,j+1)) &
     - f(i,j))*brecip + uold(i,j)
   u(i,j) = uold(i,j) - omega * r
   error = error + r*r
  enddo
 enddo
 k = k + 1
 error = sqrt(error)/dble(n*m)
enddo
```

serial

```
do while (k.le.maxit .and. error.gt. tol)
 error = 0.0
!$omp parallel
!$omp do
 do j=1,m  !---Copy solution
  do i=1,n
   uold(i,j) = u(i,j)
  enddo
 enddo
!$omp do private(r) reduction(+:error)
 do j = 2,m-1 !---Update interior
  do i = 2,n-1
   r = &
      (ax*(uold(i-1,j) + uold(i+1,j)) &
     + ay*(uold(i,j-1) + uold(i,j+1)) &
     - f(i,j))*binv + uold(i,j)
   u(i,j) = uold(i,j) - omega * r
   error = error + r*r
  enddo
 enddo
!$omp enddo nowait
!$omp end parallel
 k = k + 1
 error = sqrt(error)/dble(n*m)
enddo
```

omp 3.x / “shared”

OpenMP styles

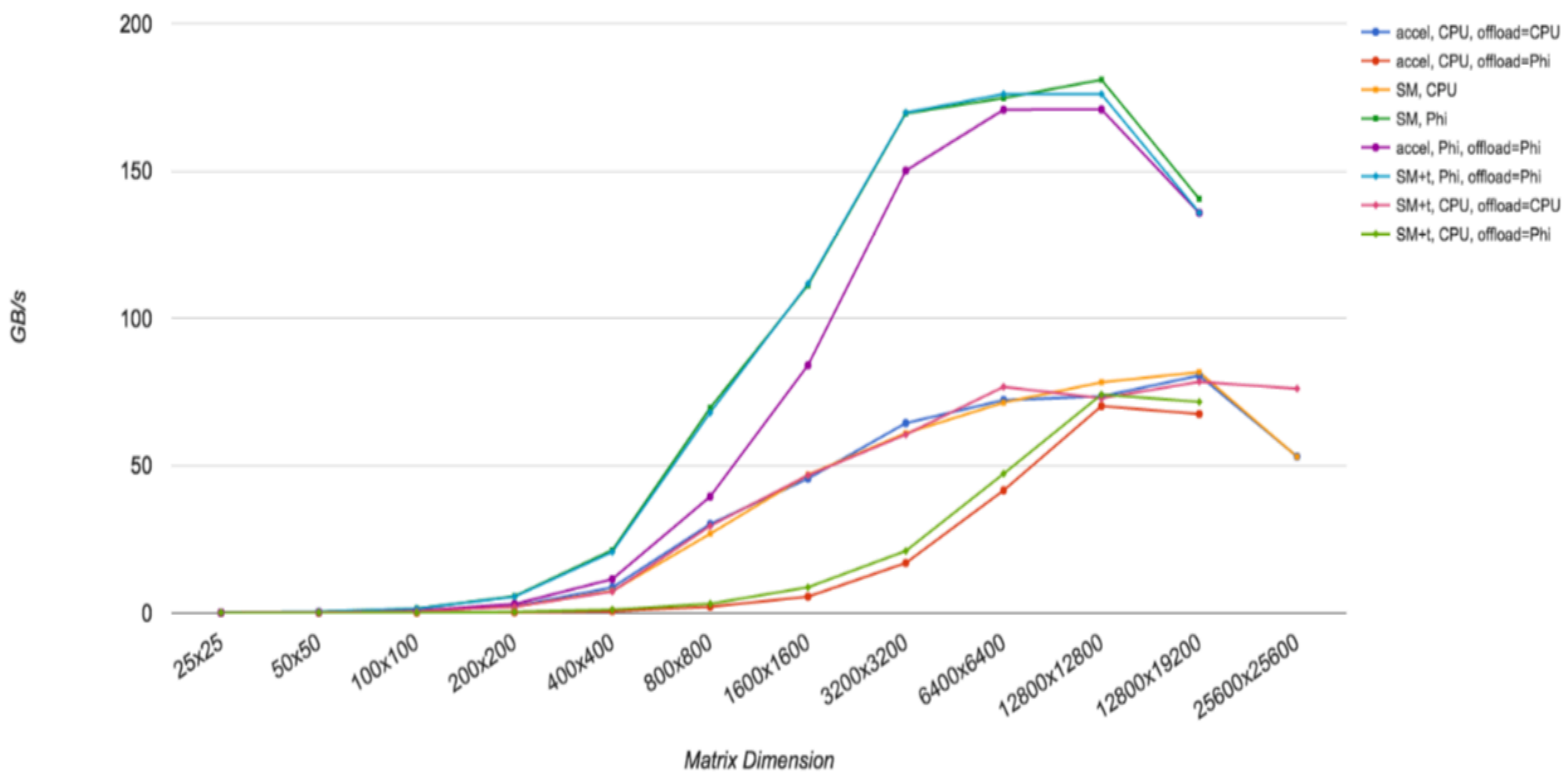
```
!$omp target data map(to:f) map(tofrom:u)
!$omp+ map(alloc:uold)
do while (k.le.maxit .and. error.gt. tol)
 error = 0.0
!$omp target
!$omp parallel
!$omp do
 do j=1,m  !---Copy solution
  do i=1,n
   uold(i,j) = u(i,j)
  enddo
 enddo
!$omp do private(r) reduction(+:error)
 do j = 2,m-1 !---Update interior
  do i = 2,n-1
   r = &
      (ax*(uold(i-1,j) + uold(i+1,j)) &
     + ay*(uold(i,j-1) + uold(i,j+1)) &
     - f(i,j))*binv + uold(i,j)
   u(i,j) = uold(i,j) - omega * r
   error = error + r*r
  enddo
 enddo
!$omp enddo nowait
!$omp end parallel
!$omp end target
 k = k + 1
 error = sqrt(error)/dble(n*m)
enddo
!$omp end target data
```

"shared + target"

```
!$omp target data map(to:f) map(tofrom:u)
!$omp+ map(alloc:uold)
do while (k.le.maxit .and. error.gt. tol)
 error = 0.0
!$omp target
!$omp teams distribute parallel do
 do j=1,m  !---Copy solution
  do i=1,n
   uold(i,j) = u(i,j)
  enddo
 enddo
!$omp end teams distribute parallel do
!$omp end target
!$omp target
!$omp teams distribute parallel
!$omp+ do reduction(+:error)
 do j = 2,m-1 !---Update interior
!$omp simd private(r) reduction(+:error)
  do i = 2,n-1
   r = &
      (ax*(uold(i-1,j) + uold(i+1,j)) &
     + ay*(uold(i,j-1) + uold(i,j+1)) &
     - f(i,j))*binv + uold(i,j)
   u(i,j) = uold(i,j) - omega * r
   error = error + r*r
  enddo
 enddo
!$omp end teams distribute parallel do
!$omp end target
 k = k + 1
 error = sqrt(error)/dble(n*m)
enddo
!$omp end target data
```

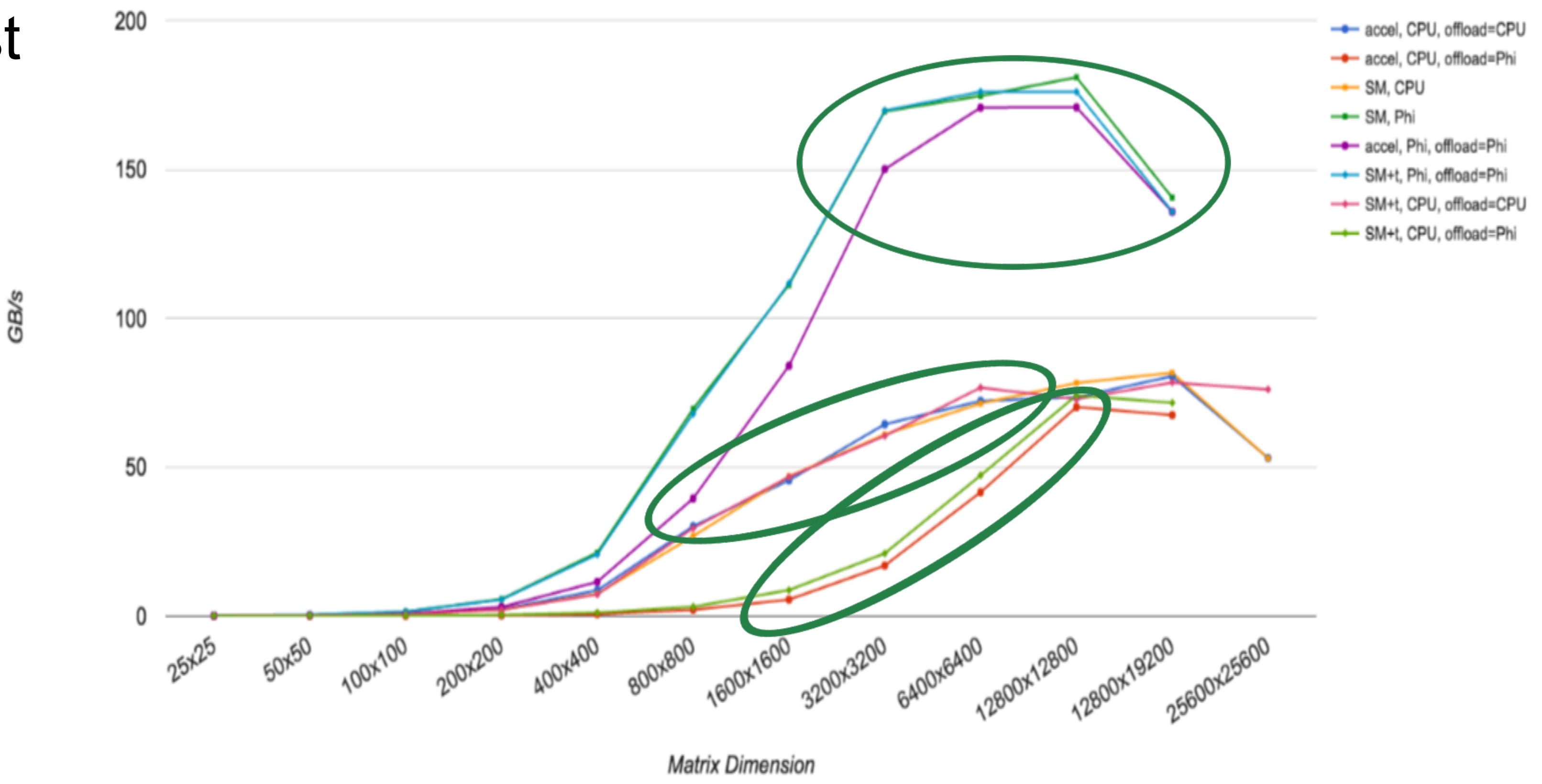
omp 4.x "offload"

Intel Phi (Beacon)

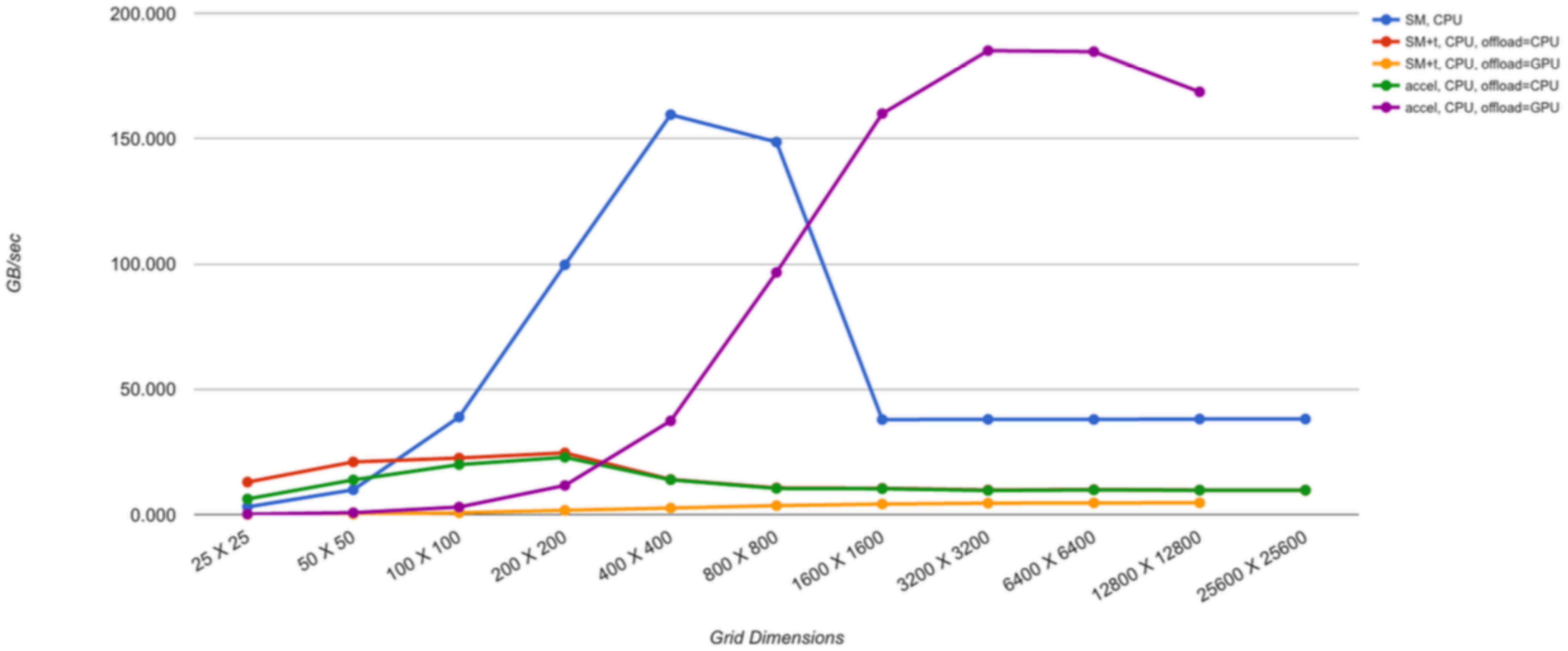


Intel Phi (Beacon)

1. Native Phi always best
2. No overhead for 4.x on CPU only
3. Both modes of offload CPU→Phi do poorly



GPU (Chester)



200.000
150.000
100.000
50.000
0.000
GB/sec
25 X 25
50 X 50
100 X 100
200 X 200
400 X 400
800 X 800
1600 X 1600
3200 X 3200
6400 X 6400
12800 X 12800
25600 X 25600
Grid Dimensions
SM, CPU
SM+t, CPU, offload=CPU
SM+t, CPU, offload=GPU
accel, CPU, offload=CPU
accel, CPU, offload=GPU

Intermediate observations

- The Intel compiler provided good performance using the self-offload approach that was nearly as effective as native OpenMP 3.1. This is an existence proof that the approach can in principle work.
- The Cray compiler on Chester, a GPU-based system, generated good code using standard approaches but did not perform well using self-offload.
- To satisfy portability, a standards-based approach seems most reasonable. OpenMP's directives-based approach is one of a handful of current candidates
- OpenMP can, at least in principle, enable some performance portability across the two architectural "swim-lanes"
- More work to do: POWER 8+, self-offloading, more GA implementations, multiple accelerators

OAK RIDGE National Laboratory

HACK-mk – CORAL benchmark

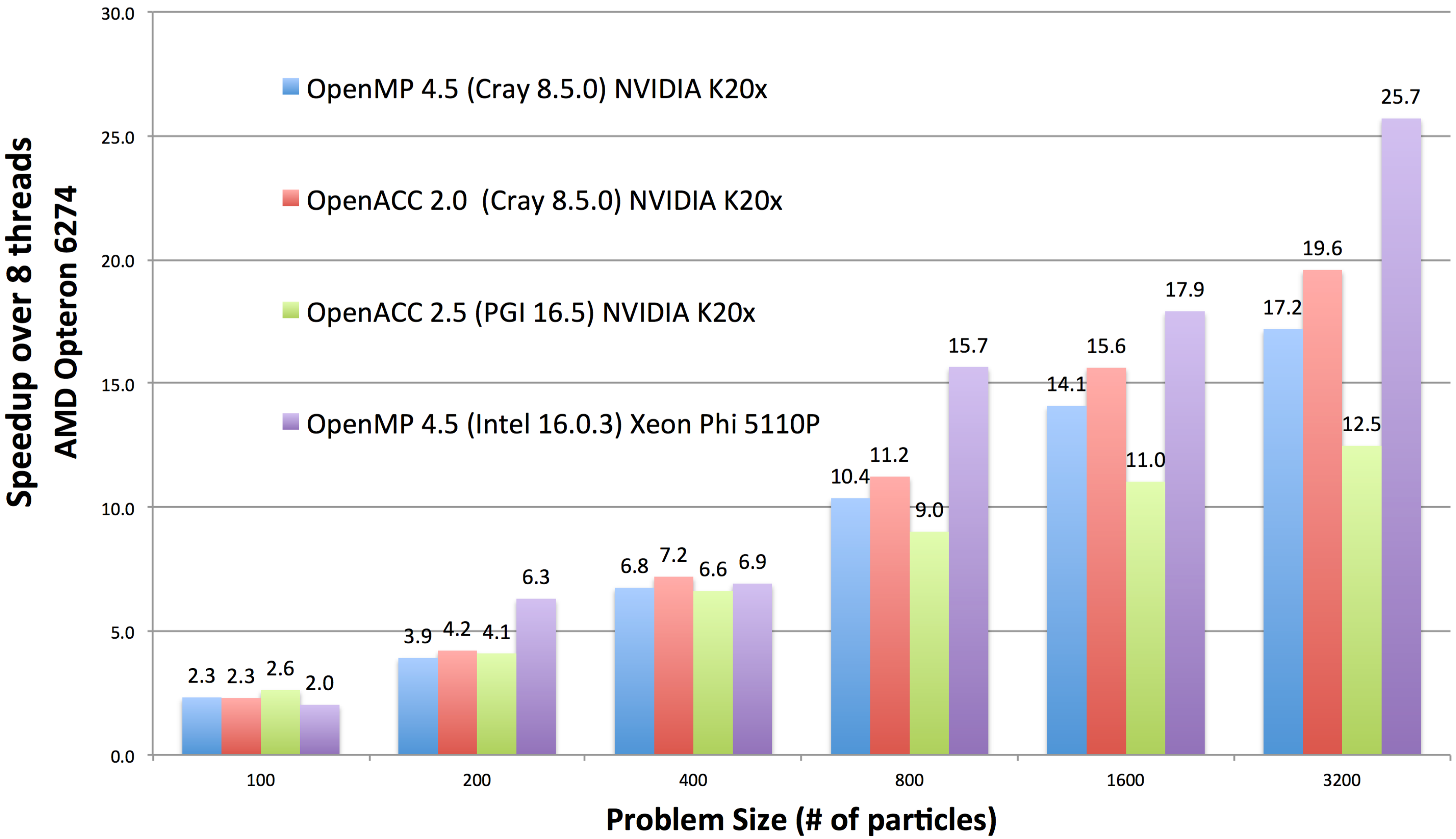
```
#pragma omp declare target
void
Step10(int count1, float xxi, float yyi,
       float zzi, float fsrrmax2, float mp_rsm2,
       float *xx1, float *yy1, float *zz1,
       float *mass1, float *dxi, float *dyi,
       float *dzi );
#pragma omp end declare target
int main() {
...
  #pragma omp target teams private(dx1, dy1, dz1)&
  #pragma omp map(to: xx[0:n],yy[0:n],zz[0:n])
  #pragma omp distribute parallel for
  for (i = 0; i < count; ++i) {
    Step10(n, xx[i], yy[i], zz[i], fsrrmax2,
           mp_rsm2, xx, yy, zz, mass, &dx1,
           &dy1, &dz1 );

    vx1[i] = vx1[i] + dx1 * fcoeff;
    vy1[i] = vy1[i] + dy1 * fcoeff;
    vz1[i] = vz1[i] + dz1 * fcoeff;
  }
...
}
void
Step10(...) {
...
  #pragma omp simd private(dxc, dyc, dzc, r2, m, f) &
  #pragma omp reduction(+:xi,yi,zi)
  for (j = 0; j < count1; j++) {
    dxc = xx1[j] - xxi;
    dyc = yy1[j] - yyi;
    dzc = zz1[j] - zzi;
    r2 = dxc * dxc + dyc * dyc + dzc * dzc;
    m = (r2 < fsrrmax2) ? mass1[j] : 0.0f;
    f =  powf(r2 + mp_rsm2, -1.5) -
         (ma0 + r2*(ma1 + r2*(ma2 +
          r2*(ma3 + r2*(ma4 + r2*ma5)))));
    f = (r2 > 0.0f) ? m * f : 0.0f;
    xi = xi + f * dxc;
    yi = yi + f * dyc;
    zi = zi + f * dzc;
  }
...
}
```

```
int main() {
...
  #pragma acc parallel private(dx1,dy1,dz1) &
  #pragma acc copy(vx1,vy1,vz1) &
  #pragma acc copyin(xx[0:n],yy[0:n],zz[0:n])
  #pragma acc loop gang
  for (i = 0; i < count; ++i) {
  ...
    #pragma acc loop vector &
    #pragma acc private(dxc, dyc, dzc, r2, m, f) &
    #pragma acc reduction(+:xi,yi,zi)
    for (j = 0; j < n; j++) {
        dxc = xx[j] - xx[i];
        dyc = yy[j] - yy[i];
        dzc = zz[j] - zz[i];
        r2 = dxc * dxc + dyc * dyc + dzc * dzc;
        m = (r2 < fsrrmax2) ? mass[j] : 0.0f;
        f = powf(r2 + mp_rsm2, -1.5) -
            (ma0 + r2*(ma1 + r2*(ma2 +
             r2*(ma3 + r2*(ma4 + r2*ma5)))));
        f = (r2 > 0.0f) ? m * f : 0.0f;
        xi = xi + f * dxc;
        yi = yi + f * dyc;
        zi = zi + f * dzc;
    }
    dx1 = xi;
    dy1 = yi;
    dz1 = zi;
    vx1[i] = vx1[i] + dx1 * fcoeff;
    vy1[i] = vy1[i] + dy1 * fcoeff;
    vz1[i] = vz1[i] + dz1 * fcoeff;
  }
...
}
```

HACK-mk – CORAL benchmark

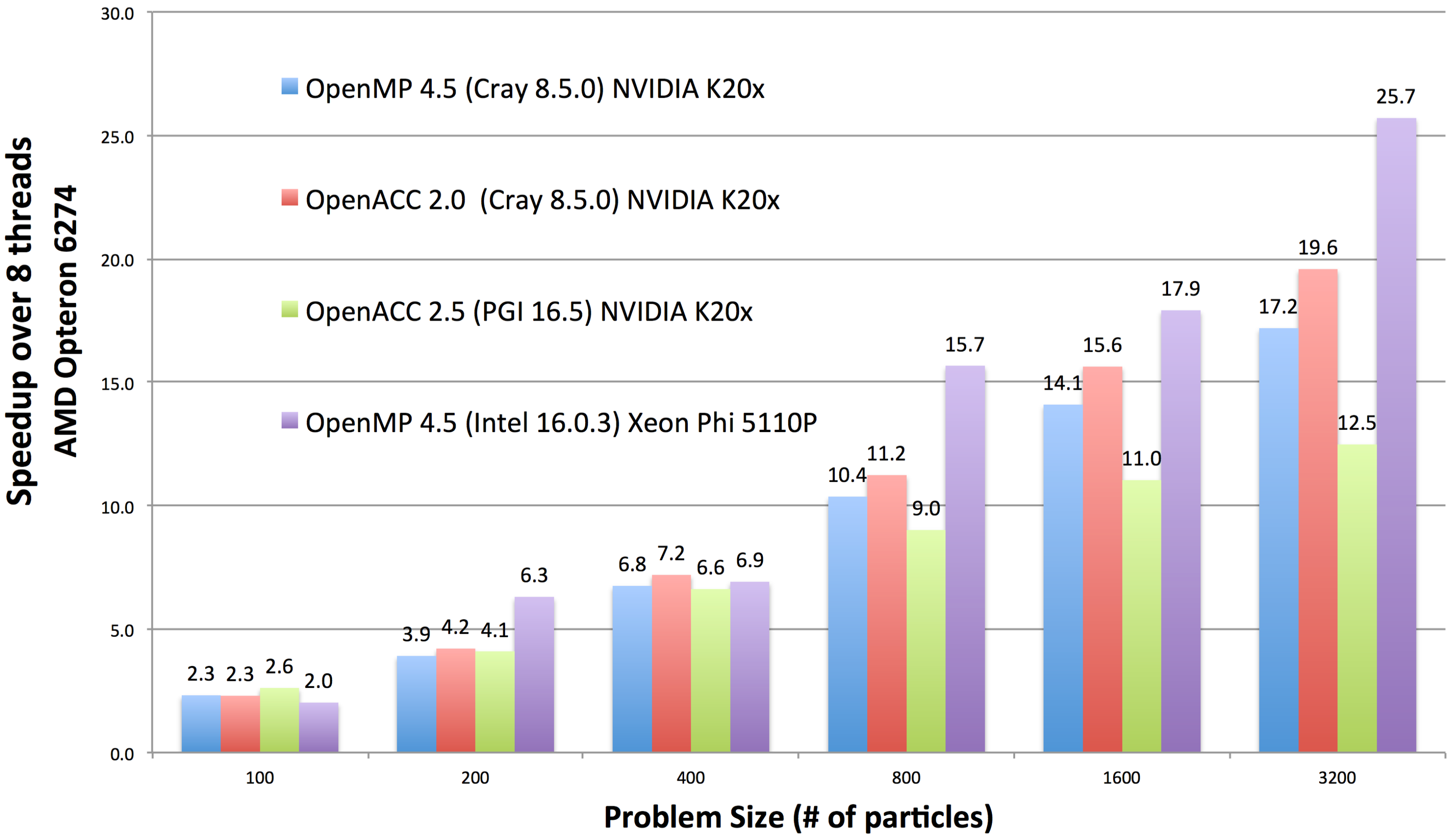
- Cray compiler OpenACC advantage for large sizes
  - forced manual inlining helped with loop scheduling and shared memory footprint
- Phi's long vector units help more at large problem sizes
- Host offloading (Cray OpenMP, PGI OpenACC)
  - all performed badly
  - poor SSE instruction support
  - poor vector support for intrinsics (powf)



OAK RIDGE National Laboratory

HACK-mk – CORAL benchmark

- Success of performance portability is **very** dependent on implementation
  - Programming model differences are not yet a deciding factor (for performance)
  - Different behaviors are still limiting PP (e.g. Intel ignores target directives in self-hosted mode)
  - SIMD directive is surprisingly helpful, across platforms and implementations



Even more “lessons learned”

DLA – DAXPY, DGEMV/N, DGEMV/T

- DAXPY, DGEMV/T: PP was generally good
  - OpenMP4/Phi, OpenMP3.1/Phi, OpenMP4/GPU, OpenACC/GPU
- DGEMV/N: mixed results
  - difficulty in optimizing non-stride-1 arithmetic
  - slightly more complex code logic

Jacobi

- Can achieve good performance across hardware, but best performance still uses different styles
  - self-hosted Phi; OMP 4.5 offloading GPU
- OMP 4.5 model is only currently PP option for “self-offloading”

Questions?