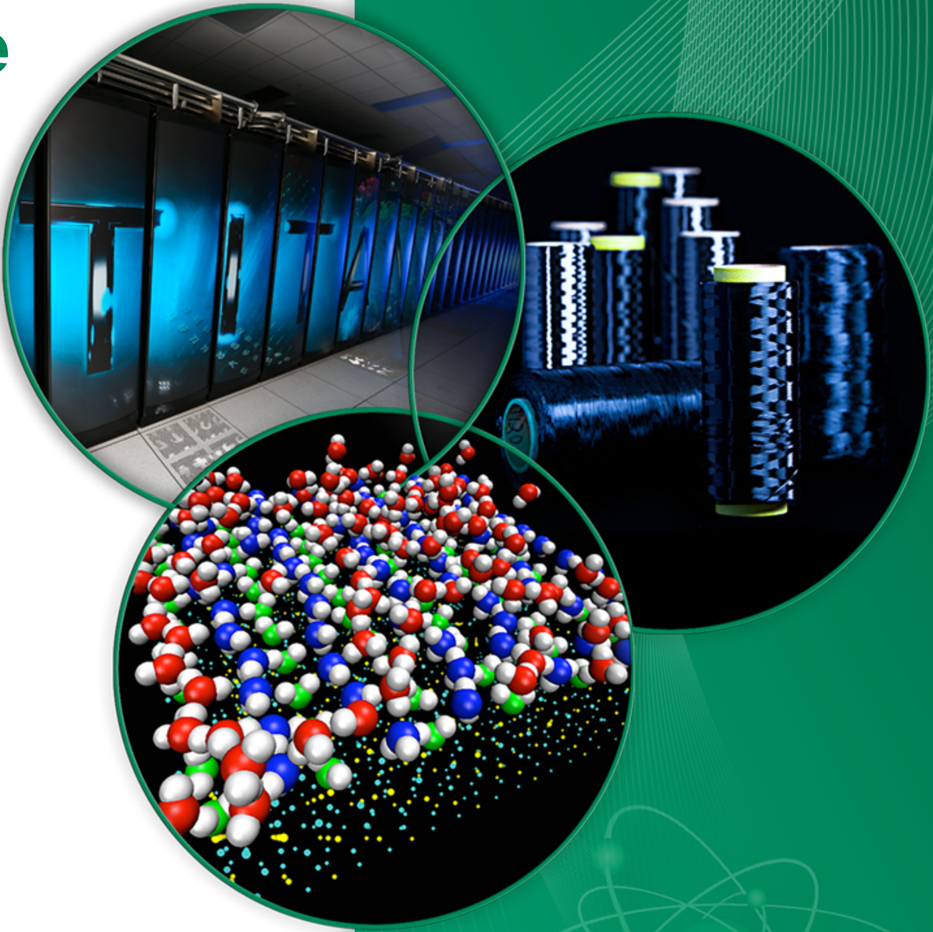
Early Experiences Writing Performance Portable OpenMP 4 Codes



Verónica G. Vergara Larrea

Wayne Joubert

M. Graham Lopez

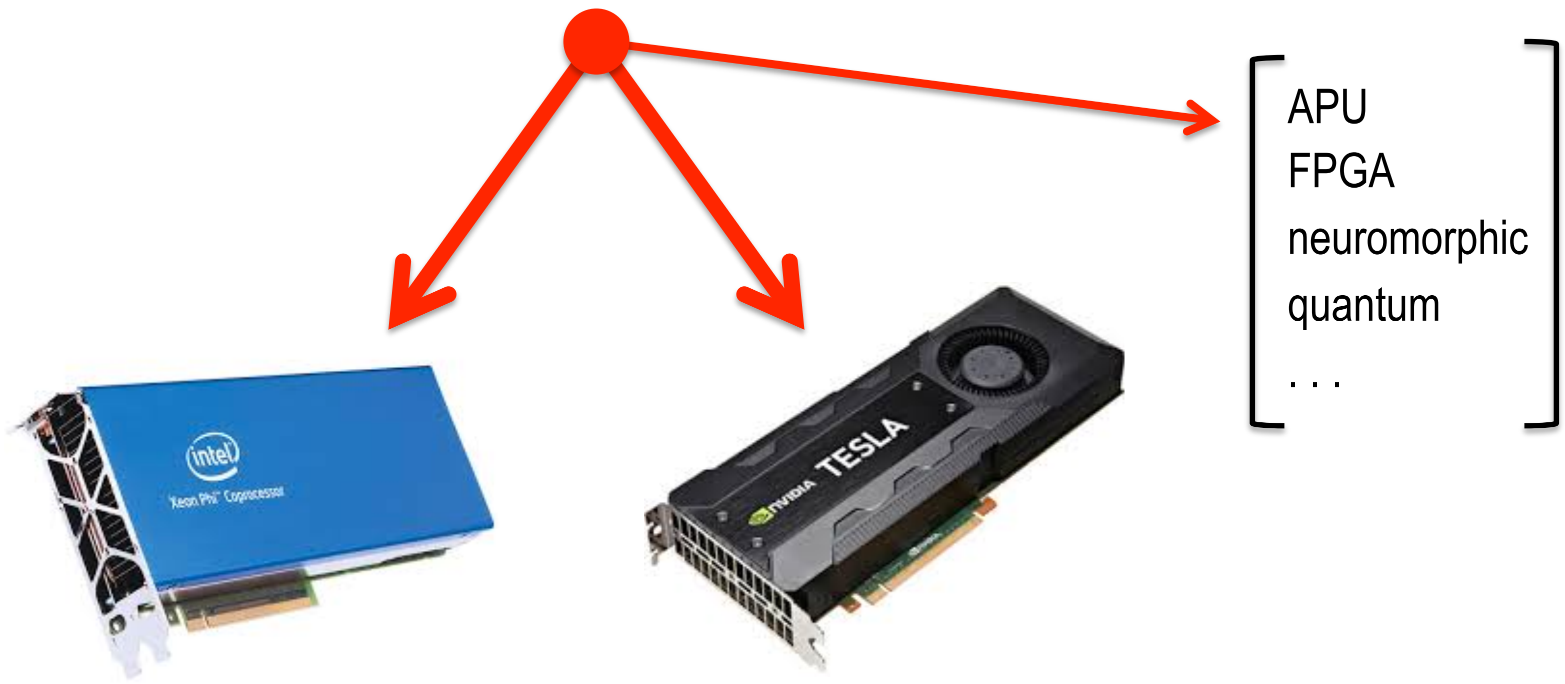
Oscar Hernandez

Oak Ridge National Laboratory

U.S. DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE U.S. DEPARTMENT OF ENERGY

Problem statement



All upcoming leadership-scale systems in the US are accelerated, of two different types:

– manycore self-hosted (Intel Phi)

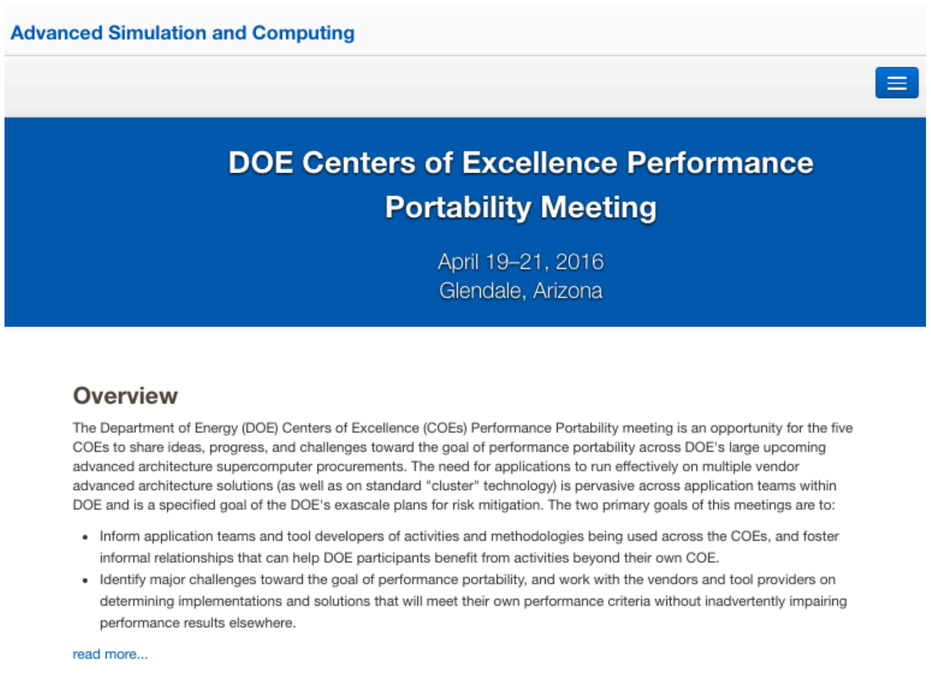
– CPU + discrete accelerator (NVIDIA GPU)

Many users run on both. How to program portably?

Performance portability requirements

- Based on the need, the US DOE has made it a priority that science applications be performance portable across multiple architectures
- This has led to several multi-lab meetings to discuss this issue, most recently the DOE Centers of Excellence workshop in Glendale, AZ, April 2016

U.S. DEPARTMENT OF ENERGY


Advanced Simulation and Computing

DOE Centers of Excellence Performance Portability Meeting

April 19–21, 2016
Glendale, Arizona

Overview

The Department of Energy (DOE) Centers of Excellence (COEs) Performance Portability meeting is an opportunity for the five COEs to share ideas, progress, and challenges toward the goal of performance portability across DOE's large upcoming advanced architecture supercomputer procurements. The need for applications to run effectively on multiple vendor advanced architecture solutions (as well as on standard "cluster" technology) is pervasive across application teams within DOE and is a specified goal of the DOE's exascale plans for risk mitigation. The two primary goals of this meetings are to:

- Inform application teams and tool developers of activities and methodologies being used across the COEs, and foster informal relationships that can help DOE participants benefit from activities beyond their own COE.
- Identify major challenges toward the goal of performance portability, and work with the vendors and tool providers on determining implementations and solutions that will meet their own performance criteria without inadvertently impairing performance results elsewhere.

read more...

Performance portability requirements

- Users at our center generally run their codes at other centers as well, on many different kinds of systems
- Furthermore, science application code bases are tending to become increasingly large and complex often with multiple physics, making it unwieldy to maintain alternative code branches for different systems
- Even within the same product family (e.g., Intel Phi Knights-X), architectures can be different enough to warrant different code tuning and possibly different code execution paths for performance
- The challenge is expected to increase as nodes become more complex with deepening memory hierarchies and complex NUMA domains

Performance portability approaches

- It is the belief of many that, at least for Fortran and C codes, the use of OpenMP compiler directives offers the most promising path forward for performance portability
- The broadly-supported OpenMP standard defines programming models to specify node-level parallelism
- Traditionally, OpenMP 3.1 directives can be used to specify parallelism for shared memory / self-hosted systems
- More recently, OpenMP 4.X directives provide data and computation offload syntax to support discrete accelerator devices

Performance portability approaches

- OpenMP thus offers (at least) two distinct programming models within the same standard: one that supports self-hosted devices, another supporting offload to discrete accelerators
- We want to address: How can an application developer use a single OpenMP-based programming style to support both architectures?
- In this study we examine how to use a single approach within the OpenMP standard to develop performance portable code across both broad classes of systems
- To test this approach, we use the Intel compiler for a representative Phi-based system (for Xeon multicore processor and for a Phi coprocessor) and also the Cray compiler for a GPU-based system

OpenMP 3.1 directives - review

We are primarily focused on parallelizing loops in the following fashion:

```
!$omp parallel !---parallel region
!$omp do !---parallelize loop iterations
do j=1,m
...
enddo
...
```

OpenMP 4.X offload directives - review

For OpenMP 4.X, we have more structure, including data and execution offload directives:

```
!---specify data transfer to/from device
!$omp target data map(...)
!---execute on device
!$omp target
!---specify thread teams
!---distribute loop iterations to teams,
!---threads and SIMD units
!$omp teams distribute parallel do simd
do j=1,m
...
enddo
```

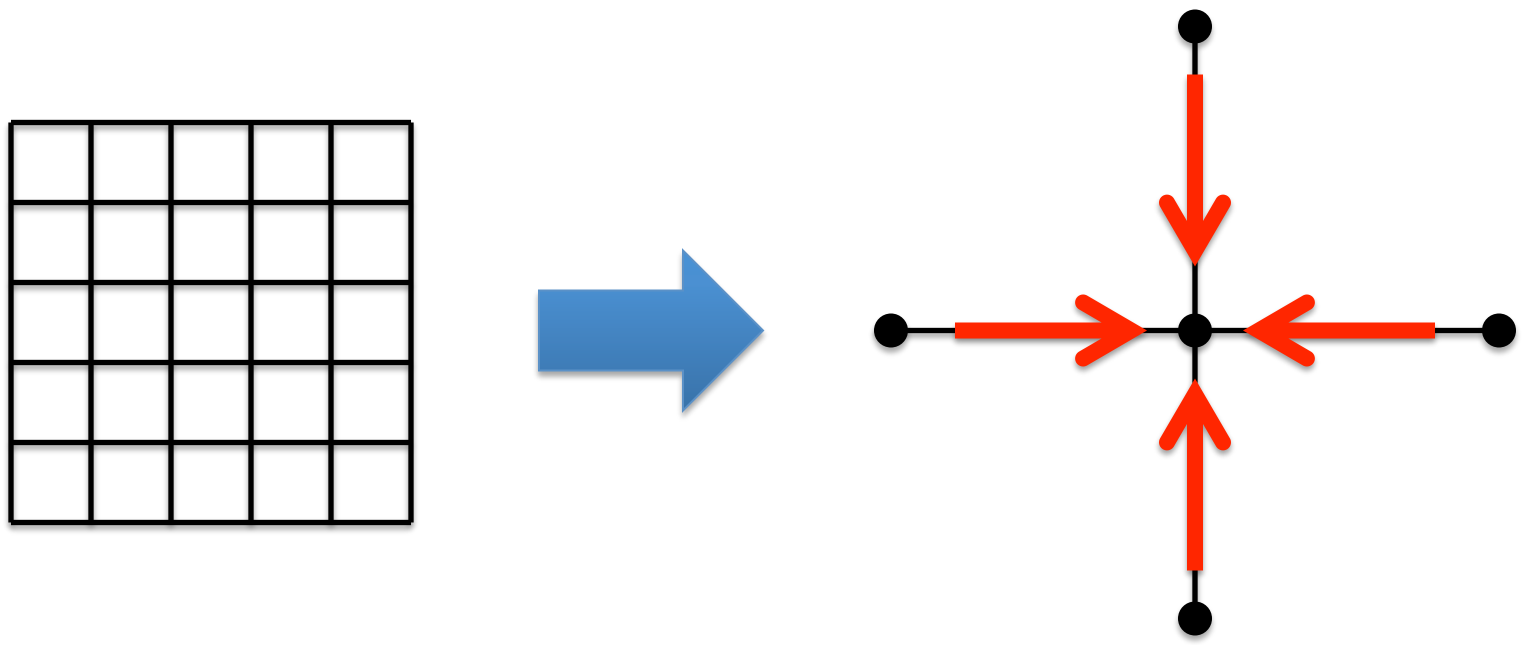
This is more verbose, but potentially more helpful to the compiler because it specifies more information, e.g., data motion

Fundamental approach

- To achieve performance portability in a single programming model, we select the OpenMP 4 offload approach
- To run code on either a self-hosted processor *or* a discrete accelerator device, we use an "*offload to self*" strategy, which is supported by these compilers via compilation options
- Our fundamental question is whether compilers will generate code using this technique that is (nearly) as efficient as using the best approach (whether shared or offload), for each respective hardware choice
- Also want to observe how each of these cases compares to "best performance attainable" for the algorithm and hardware

Test case: Jacobi iteration kernel

- A commonly studied and well-understood kernel
- Jacobi iterative solver for 2-D structured finite difference discretization of the Poisson equation
- 5-point constant coefficient stencil with nearest-neighbor updates



Jacobi kernel: serial case

```fortran
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
```

Jacobi kernel: OpenMP 3.1 directives

```
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
```

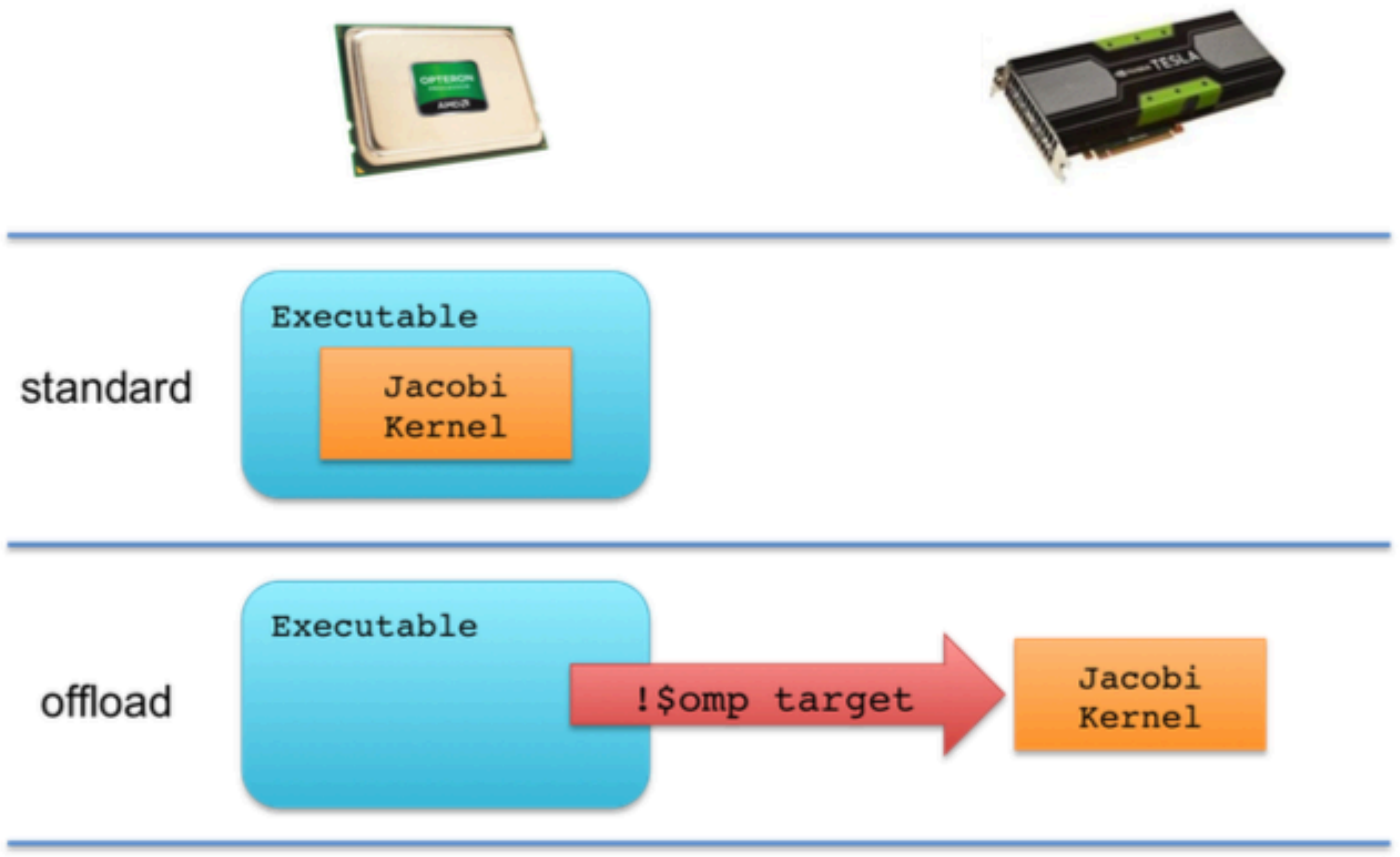
Jacobi kernel: OpenMP 4.X directives

```
!$omp target data map(to:f) map(tofrom:u)
!$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
```

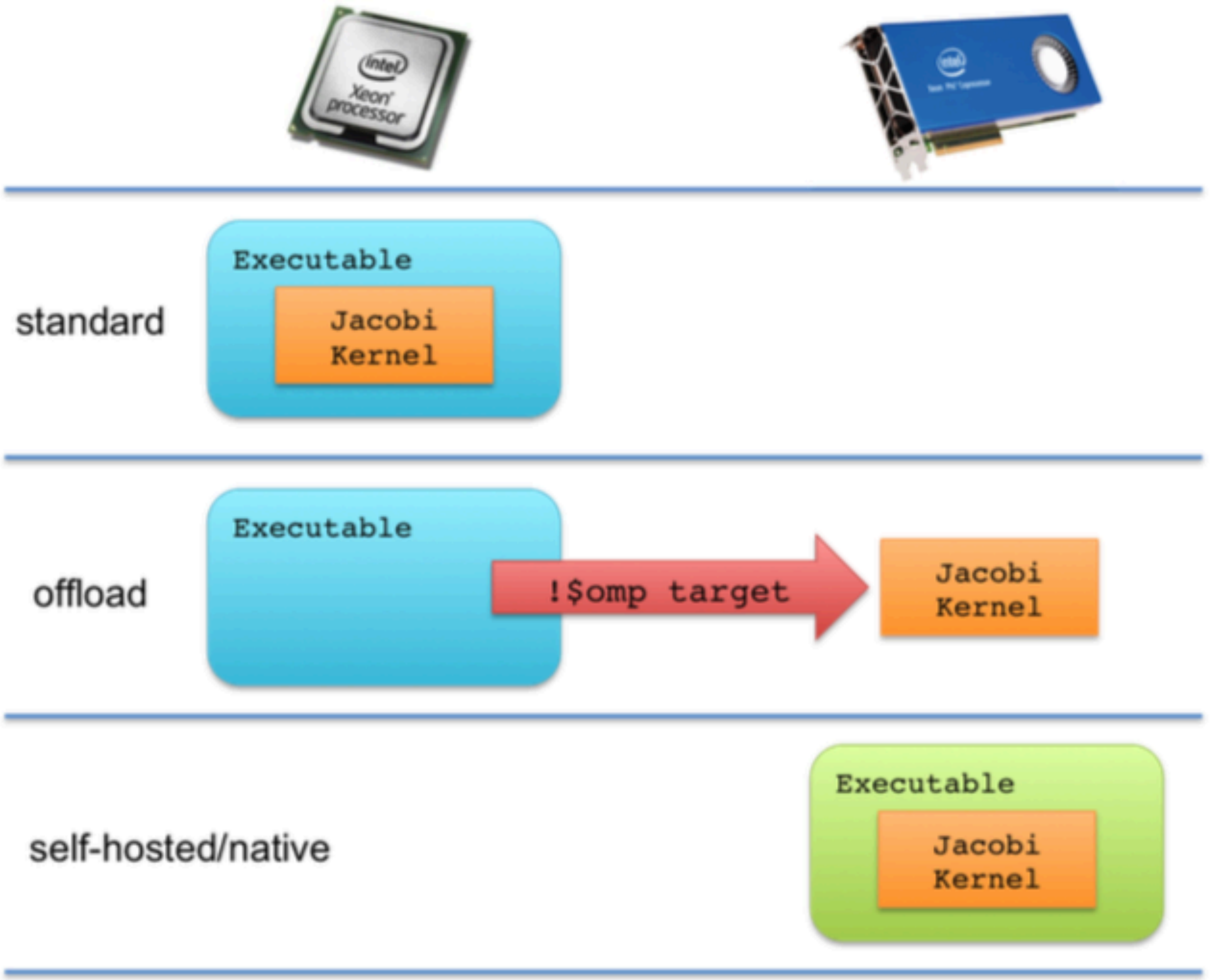
Jacobi kernel: OpenMP 3.1 + target

```
!$omp target data map(to:f) map(tofrom:u)
!$omp+ map(alloc:uold)
do while (k.le.maxit .and. error.gt. tol)
 error = 0.0
!$omp target
!$omp 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
```

Execution modes: CPU + GPU system



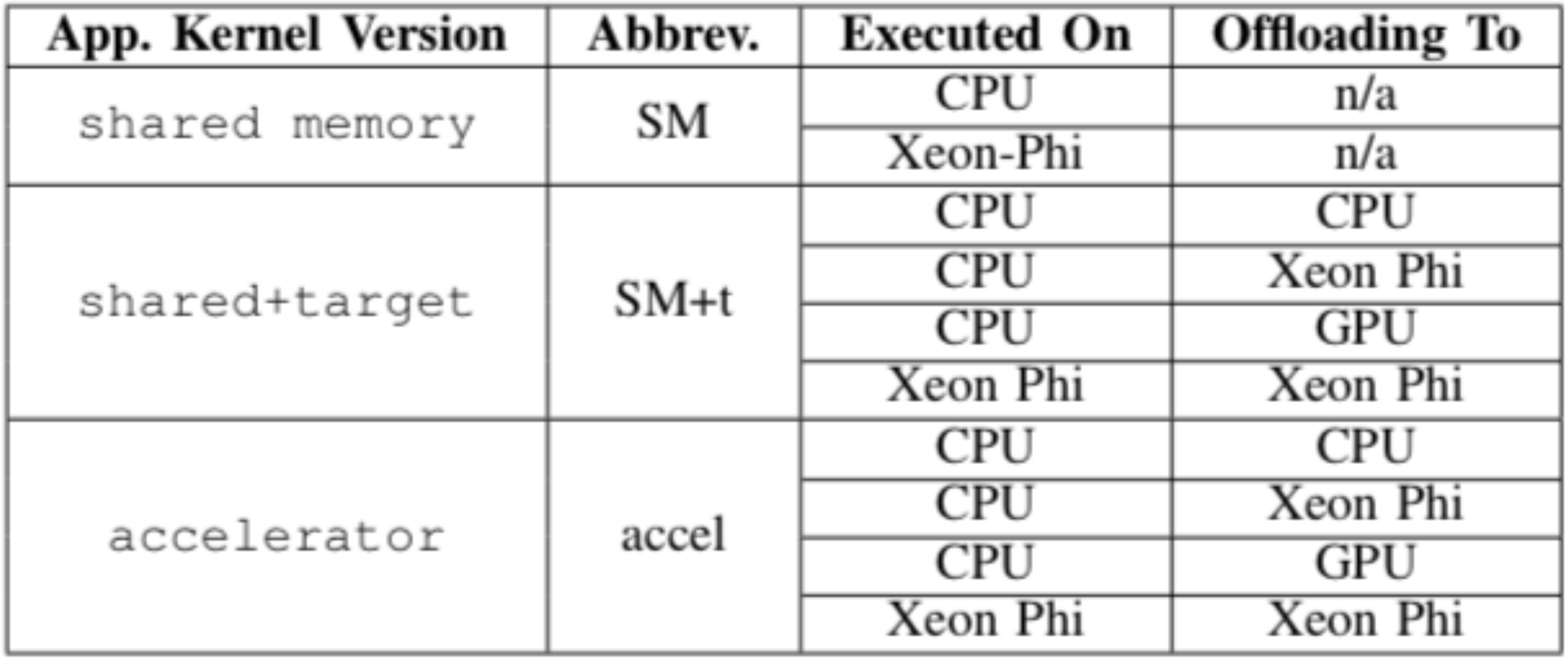
Execution modes: Knights Corner system



Systems used

- ORNL Chester system – Cray XK7 system with AMD Series 6200 Interlagos processors and NVIDIA K20X GPUs, architecturally identical to Titan but with slightly newer software stack
  - Cray compiler, CCE 8.4.5
  - Offload to host enabled by `module load craype-accel-host`
- UTK Beacon system – Cray CS-300-AC Linux cluster with Intel Xeon E5-2670 processors and Phi 5110P coprocessors
  - Intel compiler, 16.0.1 (XE compiler suite 2016.1.056)
  - Offload to host enabled by `ifort -qopenmp-offload=host`

Experiment set



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

Computational results

Codes are run on a single node of Beacon or Chester

Fixed iteration count of 100 iterations

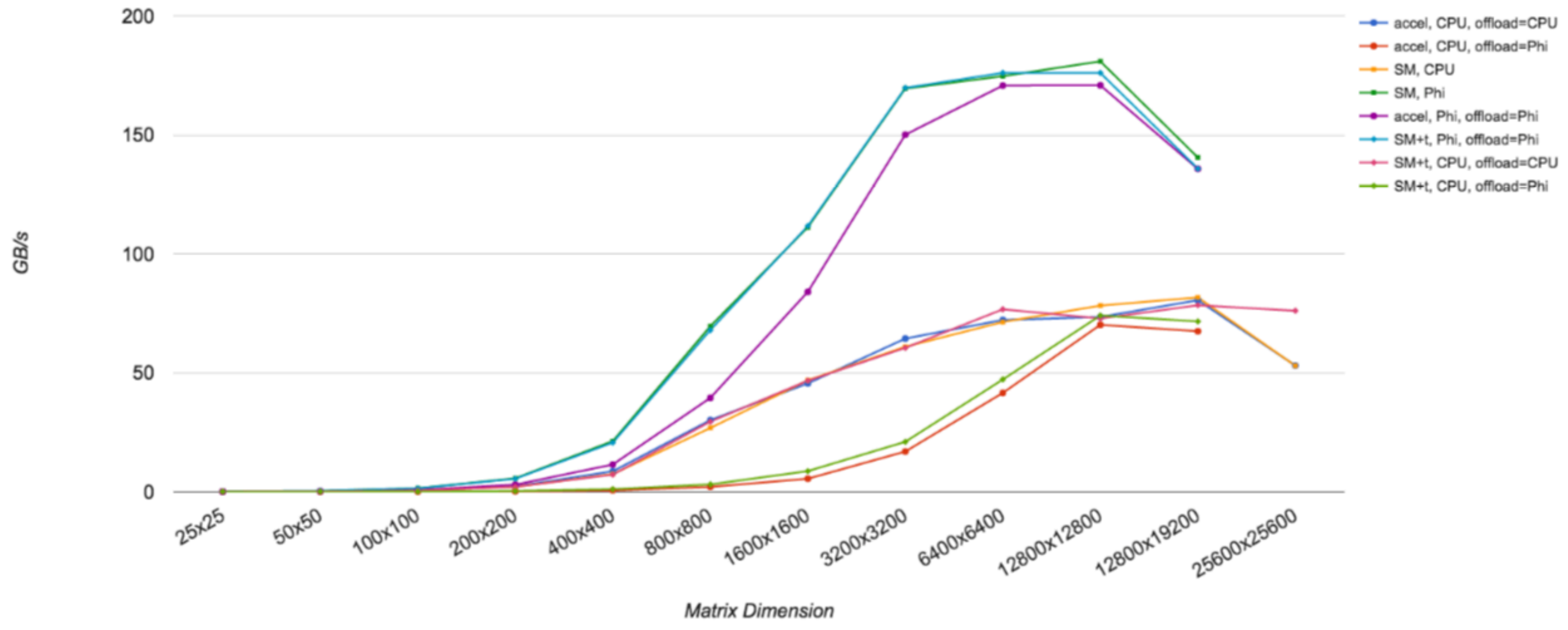
Run across different problem sizes, to the maximum size that fits in memory

Compare different execution modes

Note that the Jacobi kernel performance is fundamentally memory-bound

Thus, we will report performance in terms of GB/sec of memory bandwidth attained

Beacon results – Intel Phi



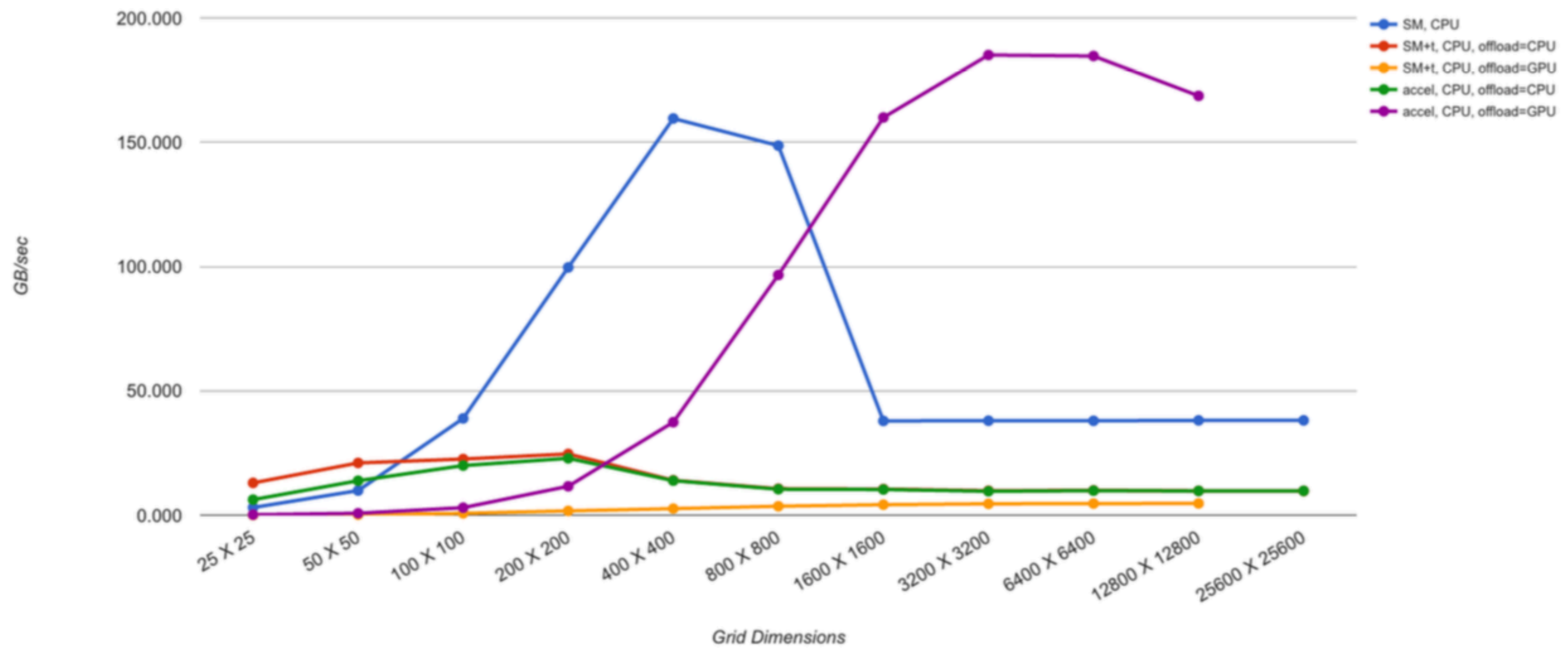
Compare to peak bandwidth: 102.4 GB/s (CPUs), 330 GB/s (Phi)

accel/phi/offload=phi and SM+t/phi/offload=phi perform similarly to SM/phi

accel/CPU/offload=CPU and SM+t/CPU/offload=CPU perform similarly to SM/CPU

accel/CPU/offload=phi and SM+t/CPU/offload=phi perform similarly

Chester results - GPU



Compare to peak bandwidth: 51.2 GB/s (CPU), 250 GB/s (GPU)

SM/CPU and accel/CPU/offload=GPU are high-performing

accel/CPU/offload=CPU performs poorly—compiler warns single-threaded code

SM+t/CPU/* performs poorly

Summary

- The Intel compiler on Beacon, a Knights Corner system, gave performance as we had hoped, providing good performance using the self-offload approach that was nearly as effective as native OpenMP 3.1. Thus is an existence proof that this approach to performance portability 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. The `craype-accel-host` option is not a widely documented feature and appears to be primarily intended for testing rather than production. Making this feature performant would help to enable this approach as a generally usable technique for performance portability

Discussion

- The end objective of this work is to seek a way to provide users with a means to express parallelism and data motion in their codes in a way that compilers can understand and use to generate performant code
- To satisfy portability, a standards-based approach seems most reasonable. OpenMP's directives-based approach is one of only a small number of current candidates
- This work shows how OpenMP can, at least in principle, enable true performance portability across the two major architectural classes of modern HPC systems, as well as standard multicore CPUs
- As OpenMP 4-capable compilers become mature, vendor support of this approach will help make this a viable strategy for performance portability

Future work

- This is a small study of very restricted scope. We would like to explore how well this approach applies to other kinds of code constructs prevalent in user codes
- Also, through our contact with code teams preparing for future systems, e.g., Summit, we wish to evaluate the effectiveness of this approach for those codes
- As OpenMP 4 support is improved for other compilers, we intend to evaluate those also
- Examine how to achieve performance portability with regard to other issues, e.g., memory hierarchies, NUMA issues and resilience

Questions?

Wayne Joubert, joubert@ornl.gov

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.