An Example of Porting PETSc Applications to Heterogeneous Platforms with OpenACC

Pi-Yueh Chuang
The George Washington University

Fernanda S. Foertter
Oak Ridge National Laboratory

OAK RIDGE INSTITUTE FOR SCIENCE AND EDUCATION
Managed by ORAU for DOE

Goal

- Develop an OpenACC example for Titan user
    - Step-by-step
    - A more realistic example of MPI + OpenACC
- Provide a reference to non-Titan users
    - How to accelerate PETSc applications in an easy way
    - Exploit the full power of heterogeneous platforms

Background: PETSc

- PETSc -- Portable, Extensible Toolkit for Scientific Computation
  - Argonne National Laboratory
  - MPI or MPI+CUDA/OpenCL/OpenMP
  - Dense/sparse linear algebra
- Large-scale parallel scientific programs in an easy way
  - Strong programming skills **(x)**
  - HPC knowledge **(x)**
  - Deriving a linear system Ax=b from physic problem **(o)**

Background: PETSc -- typical use case

In **`main()`**:

1. User-defined functions
   - Prepare a linear system (Ax=b).
2. PETSc function -- **`KSPSetup()`**
   - PETSc sets up the solver.
3. PETSc function -- **`KSPSolve()`**
   - PETSc solves the linear system.

Why this example matters

- Accelerating PETSc applications with GPUs
  - Accelerating user-defined portion **(x)**
  - Accelerating PETSc library itself **(o)**
    - A black box
    - Complicated source code
- The GPU-version of PETSc (MPI+CUDA/OpenCL)
  - Only exists in develop branch, not in official release -- unstable
  - Never worked on Titan at the time of this project

Problem and solver settings in this example

- 3D Poisson problem
    - $\nabla^2 u(x, y, z) = -12\pi^2 \cos(2\pi x) \cos(2\pi y) \cos(2\pi z)$
    - Unknowns: 27M
    - A performance bottleneck in computational fluid dynamics
- Linear solver settings
    - Conjugate-gradient method
    - Non-smoothed aggregation algebraic multigrid preconditioner
        - V cycle
        - Smoother: block Jacobi + local Jacobi

Standard workflow

1). Profiling with **Score-P**

2). Identifying the most expensive kernels

3). Inserting basic OpenACC directives

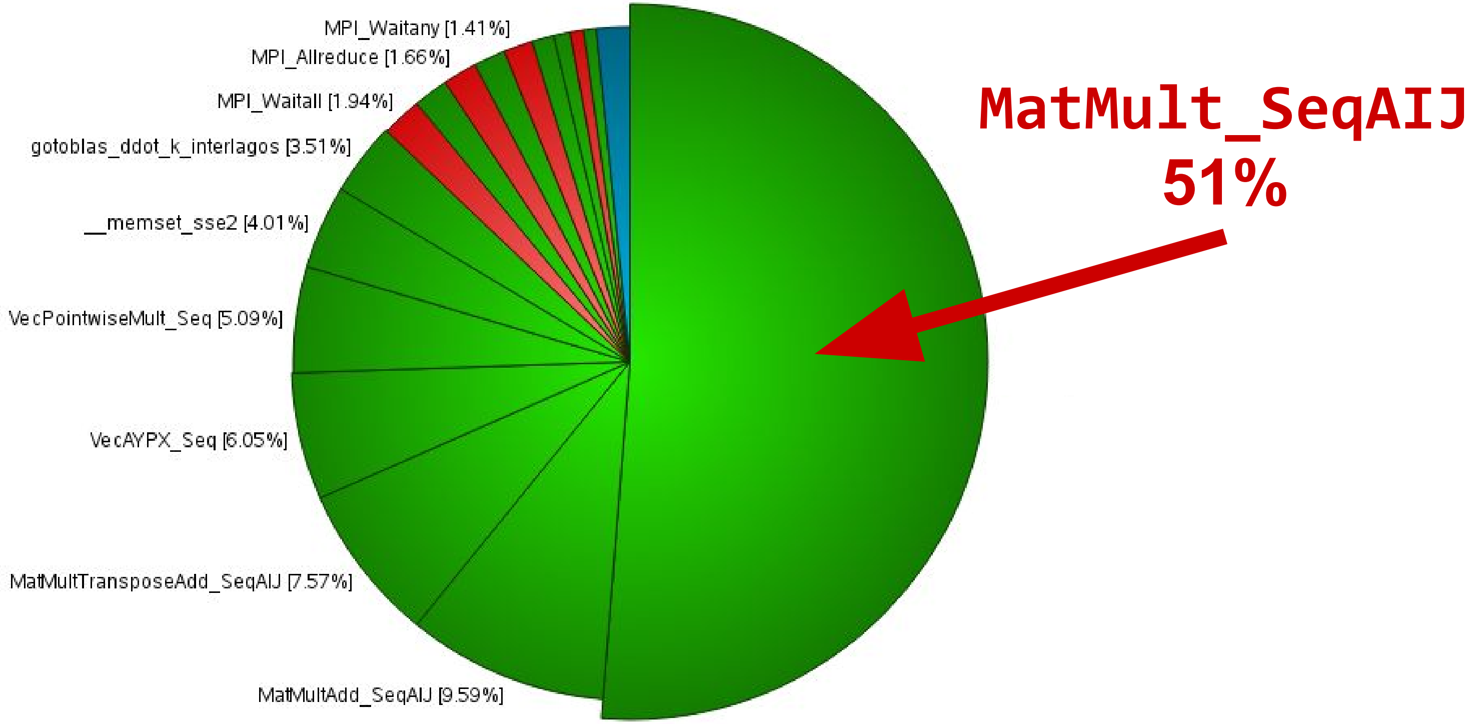
4). Profiling **NVProf** to show data transfer latency

5). Tuning/modifying the program to hide more latency

6). Repeating 4) and 5) until satisfactory

All profilings are done with a single computing node.
(16 CPU cores for CPU kernel; 16 CPU core + 1 K20x GPU for OpenaCC kernels.)

ScoreP profiling -- **KSPSolve** scope



MatMult_SeqAIJ

- Basically a sequential SpMVM (sparse matrix-vector multiplication)

```
  for (i=0; i<m; i++) {
    n             = ii[i+1] - ii[i];
    aj            = a->j + ii[i];
    aa            = a->a + ii[i];
    sum           = 0.0;
    PetscSparseDensePlusDot(sum,x,aa,aj,n);
    y[i] = sum;
  }
```

a macro

```
for (_i=0; _i<n; _i++) sum += aa[_i] * x[aj[_i]];
```

`MatMult_SeqAIJ` -- OpenACC strategy

For our Poisson matrix:

- **`m`** <= 27M / # of MPI processes
- **`n`** <= 7

For other matrices automatically created by PETSc for multigrid preconditioners, we can only guess.

- Outer loop ➞ threads/vectors; inner loop ➞ sequential
  - Heavier tasks per thread, if **`n`** is not small enough
  - Maximize the utilization of a GPU if **`m`** is large enough

Steps toward final OpenACC kernels

Step 1. inserting basic OpenACC directives

Step 2. uploading required data to GPU only once

Step 3. hiding latency with concurrent GPU/CPU executions

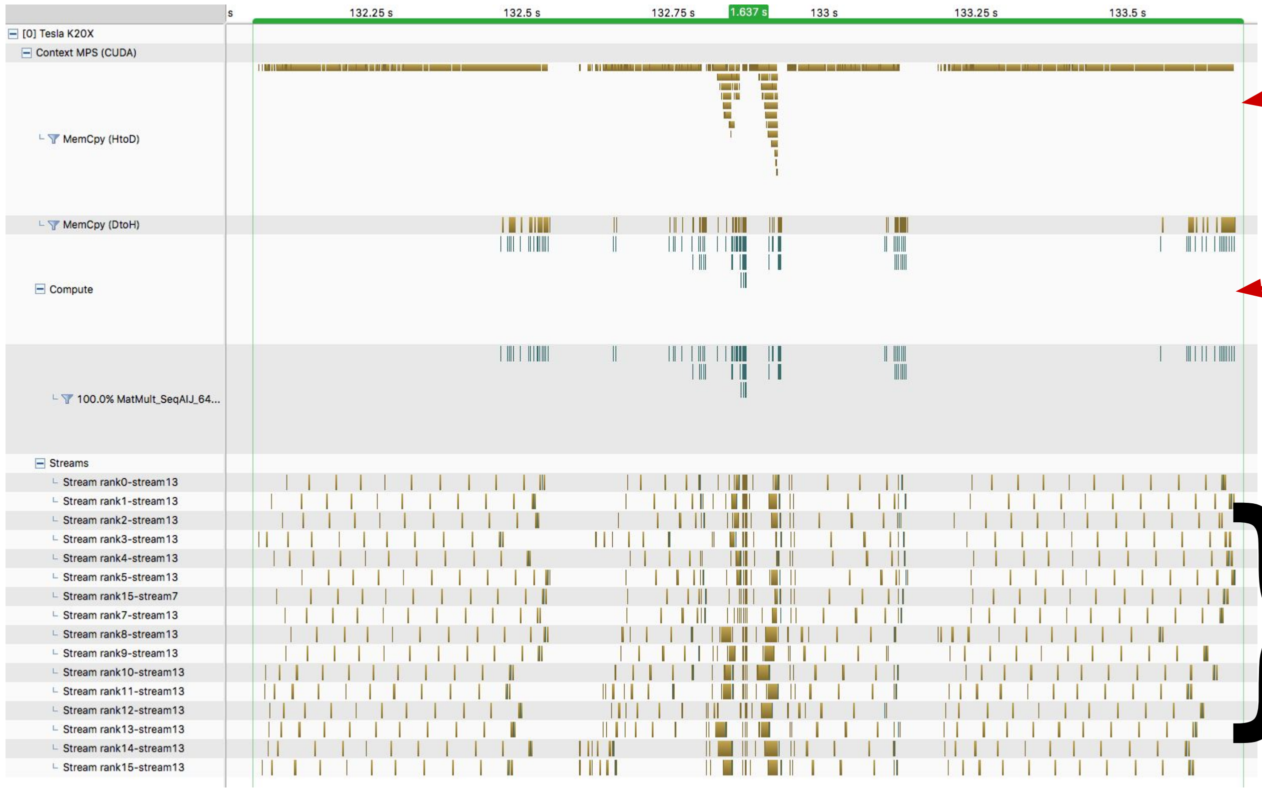
Step 4. hiding more latency with a block algorithm

Steps toward final OpenACC kernels: Step 1

- `MatMult_SeqAIJ:` **2 new lines** of directives ➡ **0.4x** speedup

```
# pragma acc kernels loop independent gang vector(32) \
  copyin(ii[:m+1] , cols[:a->nz], data[:a->nz], x[:xSize]) \
  copyout(y[:m])
for (i=0; i<m; i++) {
  n           = ii[i+1] - ii[i];
  aj          = ~~a->j~~ cols + ii[i];
  aa          = ~~a->a~~ data + ii[i];
  sum         = 0.0;
  # pragma acc loop seq reduction(+:sum)
  PetscSparseDensePlusDot(sum,x,aa,aj,n);
  y[i] = sum;
}
```

Steps toward final OpenACC kernels: Step 1



MemCpy
H ➡ D

Kernel execution

$\left.\right\}$ one MPI process

$\Downarrow$

one CUDA stream.

Steps toward final OpenACC kernels: Step 2

- Upload required data to GPU only once
  - For multigrid preconditioners, we don’t know and can’t control what are passed to **`MatMult_SeqAIJ`**.
- Let PETSc controls what to upload to and keep on GPU.
  - Allocating and uploading only when necessary
  - Data will be changed on host ➞ GPU counterpart will, too
  - Data on host will be destroyed ➞ GPU counterpart will, too.

Steps toward final OpenACC kernels: Step 2

- `MatMult_SeqAIJ`
  - only data passed into this function should be uploaded

```
# pragma acc enter data copyin( \
        ii[:m+1], cols[:a->nz], data[:a->nz], x[:xSize])

# pragma acc kernels loop independent gang vector(32) \
  present(ii[:m+1], cols[:a->nz], data[:a->nz], x[:xSize]) \
  copyout(y[:m])
for( … ) { … }

# pragma acc exit data delete(x[:xSize])
```

Steps toward final OpenACC kernels: Step 2

- **`MatAssemblyEnd_SeqAIJ`**
  - the final function called by PETSc when anything in a matrix changed

```
present[0] = acc_is_present(ai, <size>);
present[1] = acc_is_present(aj, <size>);
present[2] = acc_is_present(aa, <size>);

# pragma acc exit data delete(aj[:<length>]) if(present[1])
# pragma acc exit data delete(aa[:<length>]) if(present[2])
/* Original MatAssemblyEnd_SeqAIJ code */
# pragma acc update device(ai[:<length>]) if(present[0])
# pragma acc enter data copyin(aj[:<length>]) if(present[1])
# pragma acc enter data copyin(aa[:<length>]) if(present[2])
```

Steps toward final OpenACC kernels: Step 2

- **`MatDestroy_SeqAIJ`**
  - the final function called by PETSc when destroying a matrix

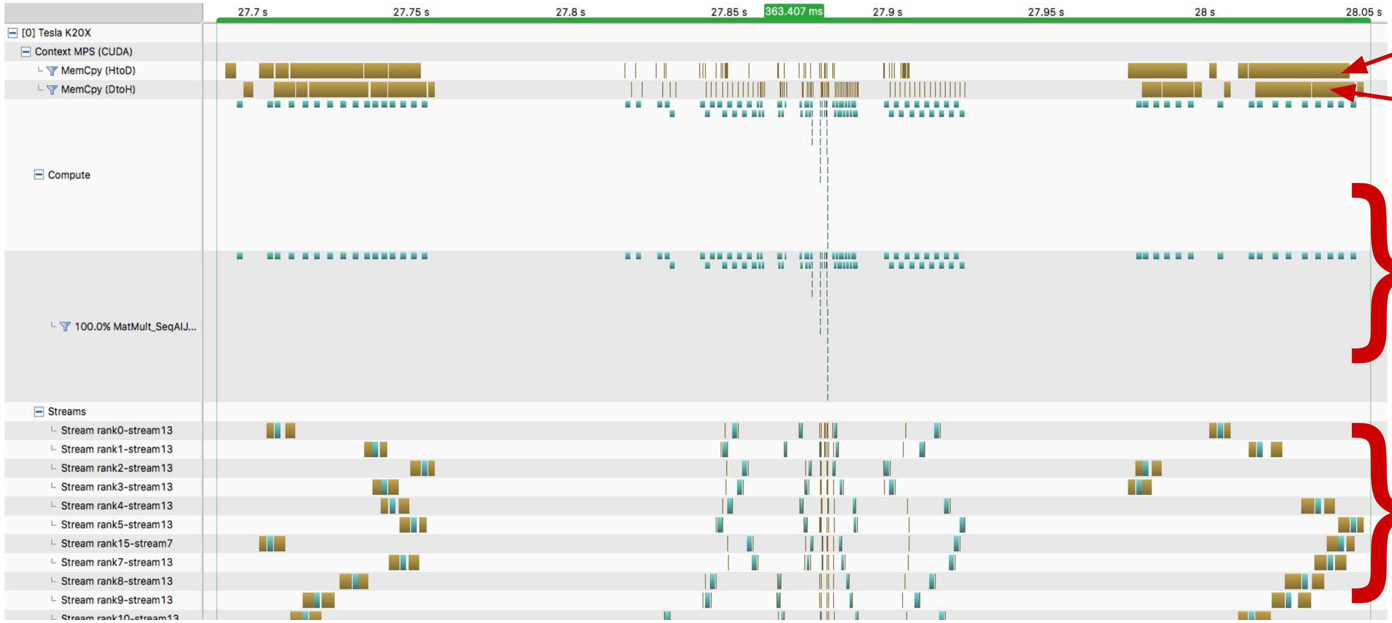
```
present[0] = acc_is_present(ai, <size>);
present[1] = acc_is_present(aj, <size>);
present[2] = acc_is_present(aa, <size>);

# pragma acc exit data delete(ai[:<length>]) if(present[0])
# pragma acc exit data delete(aj[:<length>]) if(present[1])
# pragma acc exit data delete(aa[:<length>]) if(present[2])

/* Original MatDestroy_SeqAIJ code */
```

Steps toward final OpenACC kernels: Step 2

- Result: **17 new lines** of code ➡ **1.34x** speedup



MemCpy H ➡ D

MemCpy D ➡ H

Kernel execution

Concurrent MPI processes (1 process = 1 CUDA stream)

Steps toward final OpenACC kernels: Step 3

- `MatMult_SeqAIJ`
  - Overlapping CPU/GPU tasks
  - Result: **25 additional new lines** ➡ **1.34x** speedup

```
# pragma acc enter data copyin( … ) async

PetscInt offset = 0;
while((! acc_async_test_all()) && (offset < m)) { …; offset++; }

# pragma acc kernels … copyout(y[offset:remain])
for (i=offset; i<m; i++) { … }

# pragma acc exit data delete( … ) async
```

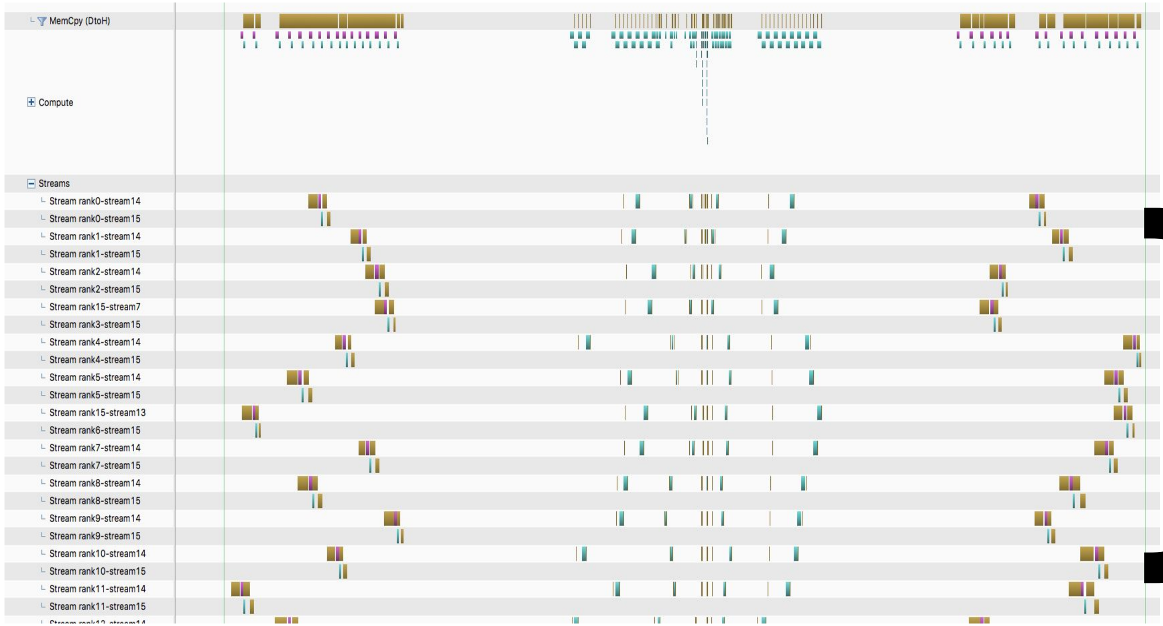
Steps toward final OpenACC kernels: Step 4

- `MatMult_SeqAIJ`
  - Block algorithm & increase concurrency

```
/* the same as in previous step (pragma acc & while loop). */
for(PetscInt b=0; b<bN; b++) {
  # pragma acc … copyout(y[offset:bSize]) async(b+1)
  for (i=offset; i<(offset+bSize); i++) { … }
  offset += bSize;
}
/* handle remaining rows */
# pragma acc wait
# pragma acc exit data …
```

Steps toward final OpenACC kernels: Step 4

- Result: **35 new lines** of code ➡ **1.34x**

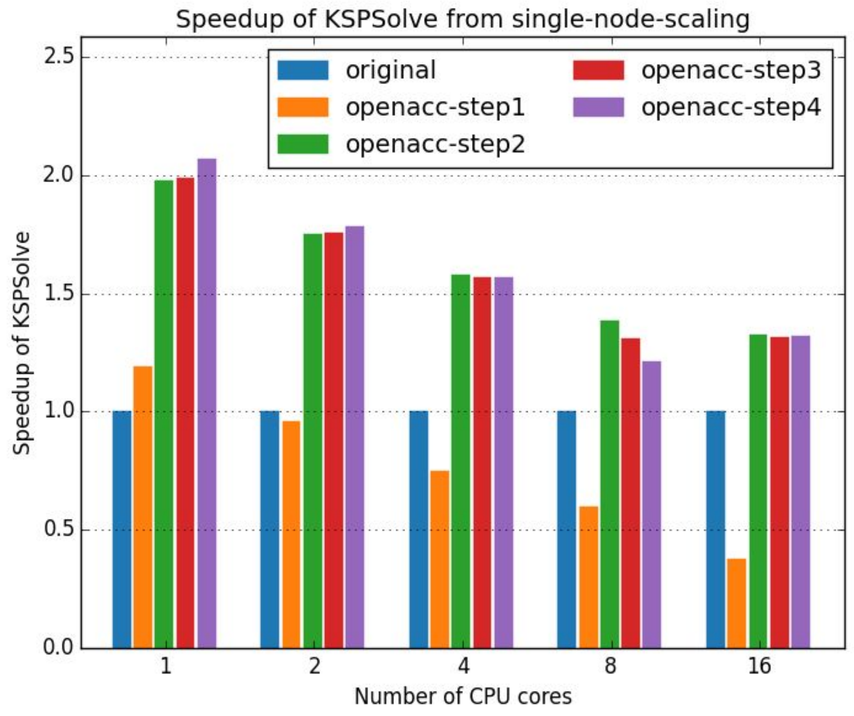


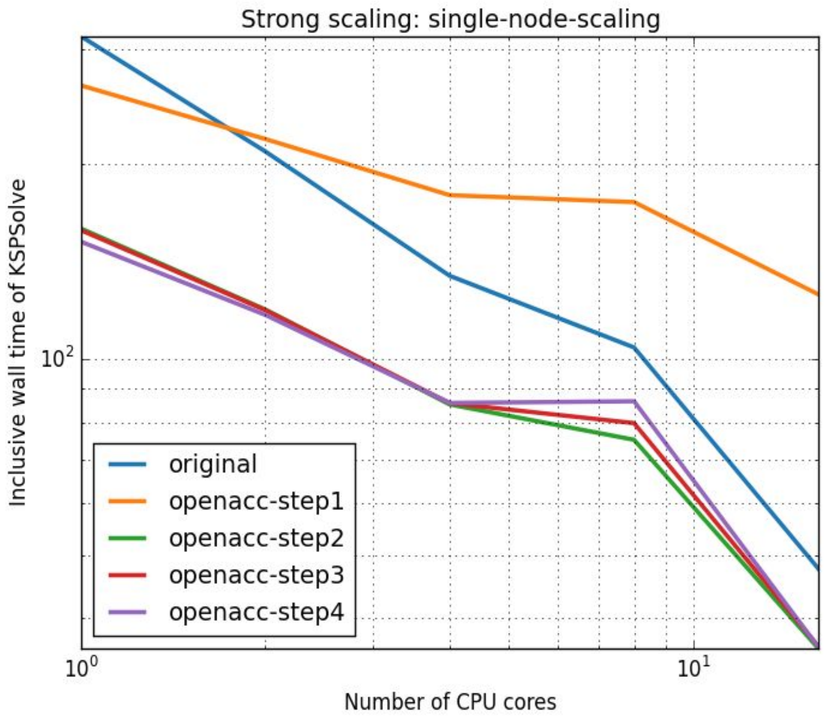
$\Bigg\}$

Now we have more than one streams on each MPI process.

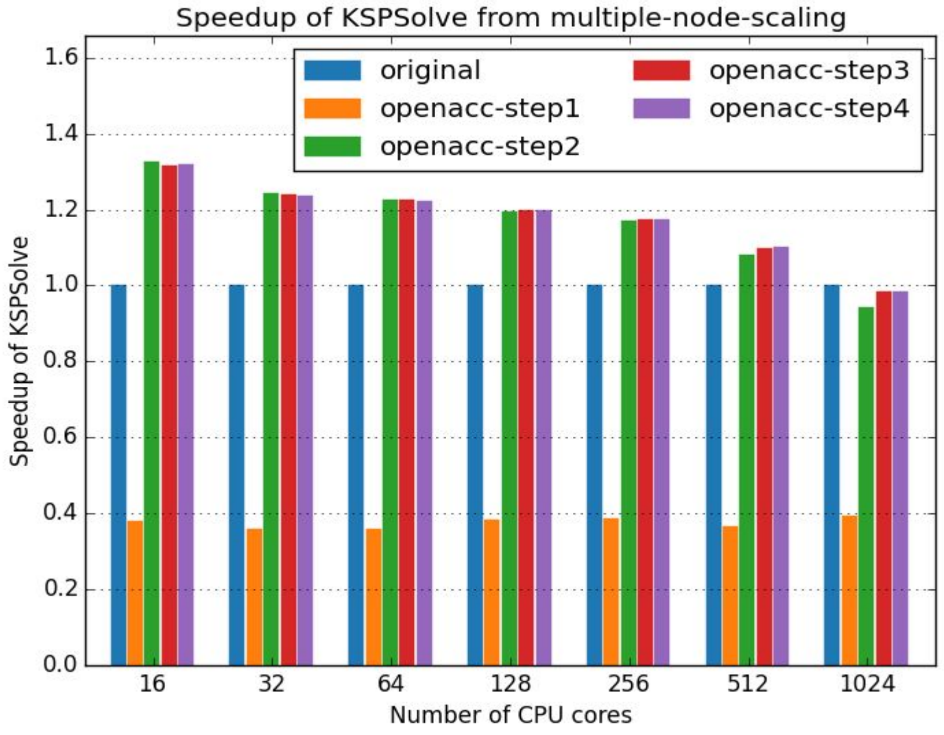
No obvious benefit.

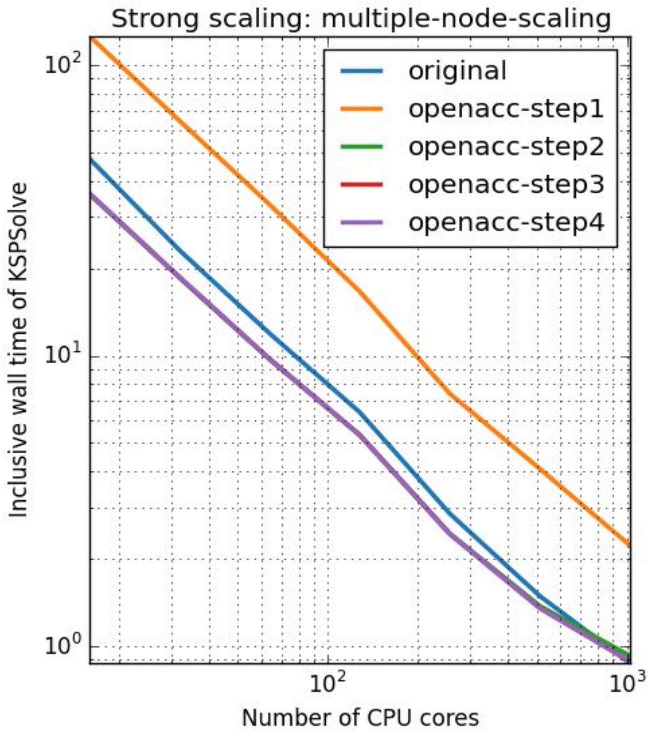
Speedups and strong scaling -- single node





Speedups and strong scaling -- multiple nodes





User experience and Conclusion

- The experience matches our expectation
  - lite code modification ➡ “**not outstanding but acceptable**” speedup.
- For a well design legacy MPI code
  - Users should be able to identify the bottleneck sequential kernel. Applying OpenACC to such a code won’t involve MPI issues.
  - Block algorithms may not be necessary, especially when many processes sharing one single GPU.

User experience and Conclusion

- Beginners' impression about OpenACC may cause wrong estimation on coding effort required.
  - Lite coding effort and no need of HPC experience may not be true
- For example, if using local SOR, instead of local Jacobi, as our local smoother, the bottleneck kernel function become **`MatSOR_SeqAIJ()`**
  - Data dependencies between iterations in nested loops
  - Require major modifications and algorithm re-design in the code
  - Require knowledge of parallel algorithms

Thank you!
Q & A

A frequently asked question

- Q: Is it necessary to port PETSc application to heterogeneous platforms?
    - Many supercomputers have more powerful CPU than Titan does. Users may get better speedups by simply running their legacy CPU codes on those supercomputers.
- A: Most researchers/scientists cannot access those supercomputers. Instead, many of them can only use university computing facilities or in-house Beowulf clusters, which may also have GPU cards installed.

Background: target readers

- Physics **(o)** and numerical methods **(?)**
- HPC or parallel programming trainings **(o)**
- Experience in real-world HPC programming **(x)**
  - Legacy codes developed by previous group members long time ago
- Willing to put much effort to modify their legacy code **(x)**
  - Project budgets or timelines don’t allow them to do so

Background: PETSc -- typical use case

In **`main()`**:

1. User-defined functions
   - Prepare a linear system (Ax=b).
2. PETSc function -- **`KSPSetup()`**
   - PETSc analyzes sets up the solver.
3. PETSc function -- **`KSPSolve()`**
   - PETSc solves the linear system.

The only part controlled by normal PETSc users.

Black boxes to normal PETSc users.

`MatMult_SeqAIJ` -- OpenACC strategies

Option 1. outer loop ➞ blocks/gangs; inner loop ➞ threads/vectors

- Simple tasks per thread
- **`n`** may be smaller than 32 for some sparse matrices

Option 2. outer loop ➞ threads/vectors; inner loop ➞ sequential

- Heavier tasks per thread, if **`n`** is not small enough
- Maximize the utilization of a GPU, if **`m`** is large enough