SANDIA REPORT

SAND2016-10327
Unlimited Release
Printed October 2016

Performance Portability of the Aeras Atmosphere Model to Next Generation Architectures using Kokkos

Jerry Watkins, Irina Tezaur

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2016-10327
Unlimited Release
Printed October 2016

Performance Portability of the Aeras Atmosphere Model to Next Generation Architectures using Kokkos

Jerry Watkins
Extreme Scale Data Science & Analytics Department
Sandia National Laboratories
P.O. Box 969, MS 9159
Livermore, CA 94551-9159

Irina Tezaur
Extreme Scale Data Science & Analytics Department
Sandia National Laboratories
P.O. Box 969, MS 9159
Livermore, CA 94551-9159

Abstract

The subject of this report is the performance portability of the Aeras global atmosphere dynamical core (implemented within the Albany multi-physics code) to new and emerging architecture machines using the Kokkos library and programming model. We describe the process of refactoring the finite element assembly process for the 3D hydrostatic model in Aeras and highlight common issues associated with development on GPU architectures. After giving detailed build and execute instructions for Aeras with MPI, OpenMP and CUDA on the Shannon cluster at Sandia National Laboratories and the Titan supercomputer at Oak Ridge National Laboratory, we evaluate the performance of the code on a canonical test case known as the baroclinic instability problem. We show a speedup of up to 4 times on 8 OpenMP threads, but we were unable to achieve a speedup on the GPU due to memory constraints. We conclude by providing methods for improving the performance of the code for future optimization.

Acknowledgment

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This work would not have been possible without the aid of a number of Sandians. We are grateful for the help of the Kokkos team including Mark Hoemmen, who helped with a number of issues with the Kokkos implementation, as well as Simon Hammond and Eric Phipps, who helped resolve compilation and execution issues on Shannon and Titan, respectively. We would like to thank all those involved with Albany and Aeras for providing valuable feedback including Irina Demeshko who started the Kokkos implementation in Albany and provided insight and comments throughout the refactoring process. Special thanks to Oksana Guba for providing the 3D Hydrostatic test cases and Peter Bosler for providing the numerical results. Lastly, we would like to thank Andy Salinger, Kyungjoo Kim and Mauro Perego for the Dynamic Rank View refactoring of Albany and Intrepid2. The results from GPU architectures would not have been complete without their work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The effects of climate change have pushed the need for more accurate and reliable global simulations at very high resolutions. Higher resolutions require more computational power, and large, high performance computing clusters are needed to obtain results within a reasonable amount of time. As high performance computing architectures become increasingly more heterogeneous, climate modeling tools must also adapt and be more efficient in taking advantage of potential performance capabilities.

This report, which goes hand-in-hand with [7], describes our work involving the integration of the Kokkos [3] library into a next generation global atmosphere model called Aeras for performance-portability to next generation architectures, e.g., GPUs, Intel Xeon Phis, multi-core CPUs, etc. Attention is restricted to porting only the finite element assembly in Aeras, as this operation requires most of the computation effort in global atmosphere models.

The Aeras global atmosphere model was created as a part of a three-year project funded by Sandia National Laboratories' Laboratory-Directed Research and Development (LDRD) program. Implemented within the Albany [6] open-source[1] C++ finite element multi-physics code written using the Trilinos [4] libraries, Aeras consists of a suite of atmosphere models: a shallow water model, an *x-z* Hydrostatic model, and a 3D Hydrostatic model. For a detailed description of Aeras, the reader is referred to [8, 7].

Aeras features two next-generation capabilities which are considered most relevant to a global atmosphere model: performance portability and embedded uncertainty quantification (UQ). It is the former of these capabilities that is the subject of this report. We focus on the 3D Hydrostatic model in Aeras, as performance portability of the shallow water model was the subject of earlier work by Demeshko *et al.* [1]. We achieve performance portability of Aeras (and, more broadly, Albany) using the open-source Kokkos [3] library and programming model developed at Sandia National Laboratories. The main benefit of Kokkos is that it allows for a single implementation of a computational kernel to run efficiently on hardware with drastically different memory models. This is enabled by data abstractions that adjust at compile time the memory layout of basic data structures to allow the transparent utilization of special hardware load and store operations.

The remainder of this report is organized as follows. In Chapter 2, we describe our refactor of the Aeras 3D Hydrostatic model using Kokkos. Chapters 3 and 4 gives detailed instructions for

[1] Albany is available on github: `https://github.com/gahansen/Albany`.

building and running Albany on two platforms, both of which contain GPUs: the Shannon cluster at Sandia National Laboratories, and the Titan supercomputer at Oak Ridge National Laboratory. In Chapter 5, we evaluate the performance of the Aeras model on these platforms with three `KokkosNodes`: Serial, OpenMP and CUDA. Attention is focused on a canonical 3D Hydrostatic test case known as the baroclinic instability problem [5]. Our results as well as future research directions are discussed in Chapter 7.

Chapter 2

Refactor of Aeras 3D Hydrostatic Model

This section describes the process of refactoring the Aeras code to work on next generation architectures via Kokkos when solving the 3D hydrostatic equations on the sphere. Similar to the Euler/Navier-Stokes, the 3D hydrostatic equations are a set of conservation laws for mass, momentum and energy [9]. These equations can be formulated into a primitive form to solve for density, velocity and temperature directly.

The equations are discretized using the spectral element method and the solution is advanced in time using a Runge-Kutta, explicit time integration. The global mesh is constructed using quadrilateral shell elements on a spherical shell domain. The mass matrix of the system is diagonal and trivial to invert when a Gauss-Lobatto points are used and the matrix is easily inverted. A finite difference technique is used in a hybrid vertical coordinate system and hyperviscosity is used to stabilize the system.

The finite element assembly process requires the computation of a Jacobian matrix (Jac) and a residual vector (Res). In Albany, the algorithm for the assembly is split into three sections:

1. Gather Jac/Res from a global structure and store it into a local structure
2. Compute Jac/Res
3. Scatter Jac/Res from a local structure to a global structure

The assembly is split into a number of evaluators. These are listed below:

Function Evaluators:

1. ComputeBasisFunctions
2. Pressure
3. Velocity
4. PiVel
5. SPressureResid
6. VirtualT

7. Density
8. Omega
9. SurfaceGeopotential
10. GeoPotential
11. EtaDotPi
12. TemperatureResid
13. KineticEnergy
14. VelResid
15. VorticityLevels

Interpolation Routines:

1. DOFDivInterpolationLevels
2. DOFGradInterpolationLevels
3. DOFVecInterpolationLevels
4. DOFInterpolationLevels
5. DOFInterpolation
6. DOFDInterpolationLevels

Gather/Scatter:

1. GatherSolution
2. ScatterHydrostatic
3. ComputeAndScatterJacobian

Computing and scattering the Jacobian only occurs once during preprocessing while the residual is gathered, computed and scattered at each iteration.

Parallelizing the assembly process among multiple CPU cores can be achieved by distributing a mesh among MPI ranks so that each rank can compute a certain chunk of cells. Each chunk can further be parallelized by utilizing more cores or dedicated cards (e.g. GPUs). The following sections describe how the latter speedup is achieved though Kokkos. A number of resources were used to refactor Aeras using the Kokkos library [10, 2]. It's highly recommended to go through these resources for a more in depth understanding of Kokkos.

2.1 Parallelizing over cells

A significant amount of speedup can be achieved in each evaluator by parallelizing each operation over cells. Most of the evaluators can be parallelized by using `Kokkos::parallel_for`. The gather and scatter operation are parallelized by using `Kokkos::atomic_fetch_add`. This section focuses on the `Kokkos::parallel_for` operator. Consider the following evaluator,

```
template<typename EvalT, typename Traits>
void XZHydrostatic_Pressure<EvalT, Traits>::
evaluateFields(typename Traits::EvalData workset)
{
  for (int cell=0; cell < workset.numCells; ++cell) {
    for (int node=0; node < numNodes; ++node) {
      for (int level=0; level < numLevels; ++level) {
        Pressure(cell,node,level) = A(level)*P0 + B(level)*Ps(cell,node);
      }
    }
  }
}
```

In this case, pressure is being computed in each cell, node and vertical level on the mesh. This type of operation is embarrassingly parallel and each computation of pressure could theoretically occur at the same time on every point in the mesh. This loop can be parallelized rather easily by using a Kokkos `parallel_for`. The transformation is shown below,

```
template<typename EvalT, typename Traits>
void XZHydrostatic_Pressure<EvalT, Traits>::
evaluateFields(typename Traits::EvalData workset)
{
  Kokkos::parallel_for(XZHydrostatic_Pressure_Policy(0,workset.numCells),
    *this);
}
```

The Range Policy, `XZHydrostatic_Pressure_Policy`, is defined in the class definition,

```
typedef Kokkos::RangePolicy<ExecutionSpace, XZHydrostatic_Pressure_Tag>
  XZHydrostatic_Pressure_Policy;
```

An execution space and a tag are passed into the Range Policy as template parameters. The execution space, `ExecutionSpace`, is needed to ensure that the operation is executed on the Phalanx device, `PHX::Device`. The tag, `XZHydrostatic_Pressure_Tag`, is used to point to a specific Kokkos operator. The template parameters and the Kokkos operator are also defined in the class definition,

```
typedef Kokkos::View<int***, PHX::Device>::execution_space ExecutionSpace;
struct XZHydrostatic_Pressure_Tag{};

KOKKOS_INLINE_FUNCTION
```

```
void operator() (const XZHydrostatic_Pressure_Tag& tag, const int& i) const;
```

Note that the operator is defined as an inline function. The operator is shown below,

```
template<typename EvalT, typename Traits>
KOKKOS_INLINE_FUNCTION
void XZHydrostatic_Pressure<EvalT, Traits>::
operator() (const XZHydrostatic_Pressure_Tag& tag, const int& cell) const{
  for (int node=0; node < numNodes; ++node) {
    for (int level=0; level < numLevels; ++level) {
      Pressure(cell,node,level) = A(level)*P0 + B(level)*Ps(cell,node);
    }
  }
}
```

This operator is often referred to as a kernel. The kernel can now be used to run on a variety of different Kokkos devices (e.g. OpenMP, CUDA) and is ready to be used on any future Kokkos devices under development. In this case, the computation is only parallelized over cells but further parallelization is available. A simple way to parallelize over all indices will be made available in future releases of Kokkos.

2.2 Development for Kokkos CUDA

Kokkos kernels offer a lot of flexibility when compiling for different devices. Optimal data layouts are constucted at compile time for each kernel but a programmer **must** utilize Kokkos Views. By using Kokkos Views, a programmer can ensure that all memory is allocated on the device. This also limits the amount of automatic memory transfer needed by CUDA UVM. Without the use of Views, kernels which may perform well in OpenMP may not work well in CUDA. In the worst case, a kernel may not even compile for CUDA. Some common issues are listed below.

2.2.1 Maintaining memory within class scope

Memory that is accessed in a Kokkos kernel must be available within the scope of the class. Consider the following modified Kokkos kernel,

```
template<typename EvalT, typename Traits>
KOKKOS_INLINE_FUNCTION
void XZHydrostatic_Pressure<EvalT, Traits>::
operator() (const XZHydrostatic_Pressure_Tag& tag, const int& cell) const{
  for (int node=0; node < numNodes; ++node) {
    for (int level=0; level < numLevels; ++level) {
      Pressure(cell,node,level) = A(level)*E.p0() + B(level)*Ps(cell,node);
    }
```

```
  }
}
```

Notice that the constant reference pressure, `p0`, is being accessed from a different class, `E`, via a function call. This may work for a kernel that is run on the host (i.e. OpenMP) but it will not work for a kernel that is run on a device (i.e. CUDA). This variable needs to be brought into the scope of the class before the Kokkos kernel is called.

2.2.2 Dynamic memory allocation

Dynamic memory allocation within a CUDA kernel is typically avoided due to loss in performance. A Kokkos kernel will still allow dynamic memory allocation on the host execution space but not on the CUDA execution space. There are multiple ways of avoiding dynamic memory allocation,

1. **Allocate dynamic memory outside the kernel:**
   This may be very simple but it may lead to data corruption. To avoid data corruption, a large enough array must be allocated so that different threads are not writing into the same memory address. This could lead to large dynamic memory allocations and should generally be avoided.

2. **Allocate static memory inside the kernel:**
   Sometimes the size of the array is already known at compile time. In this case, static memory allocation can both improve performance and be used on the fly inside the kernel. Even if the size of an array is not known, it's possible to set an upper bound on the possible memory usage and throw an error if this upper bound is reached.

3. **Compile kernels for all possible cases:**
   Sometimes the size of the array is known for a fixed number of cases. In this case, a kernel can be compiled with static memory allocation for each case. This may lead to long compile times or large programs if there are too many cases.

4. **Split the kernel into multiple kernels:**
   Often times the best option is to write multiple kernels instead of trying to do too much work inside a single kernel.

Each method may be valid depending on the kernel. To further illustrate this point, consider the following example,

```
template<typename EvalT, typename Traits>
KOKKOS_INLINE_FUNCTION
void XZHydrostatic_Pressure<EvalT, Traits>::
operator() (const XZHydrostatic_Pressure_Tag& tag, const int& cell) const{
  Kokkos::View<double*,PHX::Device> A("A", numLevels);
  for (int level=0; level < numLevels; ++level) {
    A(level) = 0.5*(a(level) + a(level+1));
  }
```

```
  for (int node=0; node < numNodes; ++node) {
    for (int level=0; level < numLevels; ++level) {
      Pressure(cell,node,level) = A(level)*P0 + B(level)*Ps(cell,node);
    }
  }
}
```

In this case, a `Kokkos::View` is being used to allocate dynamic memory inside the kernel. This will work on the host execution space but not on the CUDA execution space.

Allocate dynamic memory outside the kernel:
If the dynamic memory is allocated outside the kernel, data corruption will occur because each thread will try to fill the array for each cell. One possible solution is to allocate sufficient dynamic memory to avoid data corruption,

```
template<typename EvalT, typename Traits>
void XZHydrostatic_Pressure<EvalT, Traits>::
evaluateFields(typename Traits::EvalData workset)
{
  A = Kokkos::View<double**,PHX::Device>("A", workset.numCells, numLevels);
  Kokkos::parallel_for(XZHydrostatic_Pressure_Policy(0,workset.numCells),
    *this);
}
```

Note that `A` must be defined in the class definition. Now each thread will be able to write to a separate memory address,

```
template<typename EvalT, typename Traits>
KOKKOS_INLINE_FUNCTION
void XZHydrostatic_Pressure<EvalT, Traits>::
operator() (const XZHydrostatic_Pressure_Tag& tag, const int& cell) const{
  for (int level=0; level < numLevels; ++level) {
    A(cell,level) = 0.5*(a(level) + a(level+1));
  }

  for (int node=0; node < numNodes; ++node) {
    for (int level=0; level < numLevels; ++level) {
      Pressure(cell,node,level) = A(cell,level)*P0 + B(level)*Ps(cell,node);
    }
  }
}
```

If the number of cells is too large, this may lead to a memory constraint.

Allocate static memory inside the kernel:
Another option is to allocate sufficient static memory inside the kernel,

```
template<typename EvalT, typename Traits>
KOKKOS_INLINE_FUNCTION
void XZHydrostatic_Pressure<EvalT, Traits>::
operator() (const XZHydrostatic_Pressure_Tag& tag, const int& cell) const{
  if (numLevels > 100)
    Kokkos::abort("Error in XZHydrostatic_Pressure Kokkos kernel: numLevels > 100!")

  double A[100];
  for (int level=0; level < numLevels; ++level) {
    A[level] = 0.5*(a(level) + a(level+1));
  }

  for (int node=0; node < numNodes; ++node) {
    for (int level=0; level < numLevels; ++level) {
      Pressure(cell,node,level) = A[level]*P0 + B(level)*Ps(cell,node);
    }
  }
}
```

This assumes that the number of levels will never reach a value above 100. Also, if the number of levels is much less than 100, a lot of unused memory is being allocated on the stack.

Compile kernels for all possible cases:
A set of kernels can be precompiled if there are only a few cases which are going to be used in a specific application. For example, if the set consists of `numLevels={10,15,30,60}`, a Kokkos `parallel_for` can be used for each case,

```
template<typename EvalT, typename Traits>
void XZHydrostatic_Pressure<EvalT, Traits>::
evaluateFields(typename Traits::EvalData workset)
{
  // numLevels must be known at compile time in order to allocate array in kernel
  switch (numLevels) {
    case 10: {
      Kokkos::parallel_for(XZHydrostatic_Pressure_Policy<10>(0,workset.numCells),
        *this);
      break;
    }
    case 15: {
      Kokkos::parallel_for(XZHydrostatic_Pressure_Policy<15>(0,workset.numCells),
        *this);
      break;
    }
    case 30: {
```

```
      Kokkos::parallel_for(XZHydrostatic_Pressure_Policy<30>(0,workset.numCells),
        *this);
      break;
    }
    case 60: {
      Kokkos::parallel_for(XZHydrostatic_Pressure_Policy<60>(0,workset.numCells),
        *this);
      break;
    }
    default: {
      TEUCHOS_TEST_FOR_EXCEPTION(true, std::logic_error,
        "XZHydrostatic_Pressure Kokkos kernel not constructed for this case"
        << std::endl);
      break;
    }
  }
}
```

In this case, the Range Policy is templated,

```
template<int numLvls>
using XZHydrostatic_Pressure_Policy = Kokkos::RangePolicy
  <ExecutionSpace, XZHydrostatic_Pressure_Tag<numLvls>>;
```

The tag and Kokkos operator are also templated,

```
typedef Kokkos::View<int***, PHX::Device>::execution_space ExecutionSpace;

template<int numLvls>
struct XZHydrostatic_Pressure_Tag{};

template<int numLvls>
KOKKOS_INLINE_FUNCTION
void operator() (const XZHydrostatic_Pressure_Tag<numLvls>& tag, const int& i) const;
```

Now the Kokkos kernel can use the value `numLvls` to allocate a static memory at compile time,

```
template<typename EvalT, typename Traits>
template<int numLvls>
KOKKOS_INLINE_FUNCTION
void XZHydrostatic_Pressure<EvalT, Traits>::
operator() (const XZHydrostatic_Pressure_Tag<num_Lvls>& tag, const int& cell) const{
  double A[numLvls];
  for (int level=0; level < numLvls; ++level) {
    A[level] = 0.5*(a(level) + a(level+1));
  }

  for (int node=0; node < numNodes; ++node) {
    for (int level=0; level < numLvls; ++level) {
```

```
      Pressure(cell,node,level) = A[level]*P0 + B(level)*Ps(cell,node);
    }
  }
}
```

Each kernel will be compiled so this method may lead to longer compile times or a larger executable.

Split the kernel into multiple kernels:
In this case, the last option is the best option. The array `A` need not be computed multiple times since it has no dependency on different cells. In the code, `A` is precomputed and used wherever it's needed. This is a very simple example and the last option may not always be available.

It is important to note that although static memory allocation is faster on both CPUs and GPUs, the GPU has limited register memory and register "spilling" can occur if the static memory allocation is too large. This can lead to a significant degradation in performance so it is something to keep in mind when writing kernels.

Chapter 3

Build Intructions

These build instructions are for compiling Albany on the Shannon cluster at Sandia National Laboratories and the Titan[1] supercomputer at Oak Ridge National Laboratory.

3.1 Shannon

In order to clone remote repositories from github, set the following proxy settings:

```
export https_proxy="https://wwwproxy.sandia.gov:80"
export http_proxy="http://wwwproxy.sandia.gov:80"
```

The Trilinos and Albany can be downloaded from github as follows:

```
git clone https://github.com/trilinos/Trilinos.git
git clone https://github.com/gahansen/Albany.git
```

3.1.1 Building Albany master branch with Kokkos::Serial

Load the following modules:

```
module purge
module load openmpi/1.10.1/gcc/5.1.0/cuda/7.5.7 cmake/2.8.11.2
```

Make a build directory for Trilinos and copy over the configuration file provided in `Albany/doc/`:

```
cd Trilinos/
mkdir build-kokkos-serial
cd build-kokkos-serial/
cp ../../Albany/doc/do-cmake-trilinos-shannon-serial ./
```

Before building Trilinos, log into a batch node:

```
salloc -N1 -p pbatch
```

[1] `https://www.olcf.ornl.gov/titan/`.

Build Trilinos by using the configuration file provided:

```
source do-cmake-trilinos-shannon-serial
make -j 16
make install
```

For debugging, use `make -j 16 2>&1 | tee output.txt` to output the results of `make` into a file.

Make a build directory for Albany and copy over the configuration file provided in `Albany/doc/`:

```
cd ../../Albany/
mkdir build-kokkos-serial/
cd build-kokkos-serial/
cp ../doc/do-cmake-albany-shannon ./
```

Make sure to modify the file to point to your Trilinos build directory and build:

```
source do-cmake-albany-shannon
make -j 16
```

Once building is complete, exit the batch node:

```
exit
```

The same instructions can be used to build the Albany master branch with Kokkos::OpenMP except that the configuration file `Albany/doc/do-cmake-trilinos-shannon-openmp` needs to be used instead of `Albany/doc/do-cmake-trilinos-shannon-serial`.

3.1.2 Building Albany DynRankViewIntrepid2Refactor branch with Kokkos::Cuda

First checkout the Trilinos develop branch in the Trilinos directory:

```
git checkout develop
```

Checkout the Albany DynRankViewIntrepid2Refactor branch in the Albany directory:

```
git checkout DynRankViewIntrepid2Refactor
```

Load the following modules:

```
module purge
module load openmpi/1.10.1/gnu/4.7.2/cuda/7.5.7 cmake/2.8.11.2
module load nvcc-wrapper/gnu
```

Set the following compiler and GPU settings. Make sure to modify the `OMPI_CXX` path to point to your `nvcc_wrapper` file:

```
export NVCC_WRAPPER_DEFAULT_COMPILER=mpicc
export OMPI_CXX=/home/jwatkin/Trilinos/packages ...
                /kokkos/config/nvcc_wrapper
export CUDA_MANAGED_FORCE_DEVICE_ALLOC=1
export CUDA_LAUNCH_BLOCKING=1
```

Make a build directory for Trilinos and copy over the configuration file: `Albany/doc/do-cmake-trilinos-shannon-cuda`. Make sure to modify the `OMPI_CXX` path to point to your `nvcc_wrapper` file. Log into a batch node:

```
salloc -N1 -p pbatch
```

Build Trilinos by using the configuration file provided:

```
source do-cmake-trilinos-shannon-cuda
make -j 16
make install
```

Make a build directory for Albany and copy over the configuration file: `Albany/doc/do-cmake-albany-shannon`. Make sure to modify the file to point to your Trilinos build directory and build:

```
source do-cmake-albany-shannon
make -j 16
```

Once building is complete, exit the batch node:

```
exit
```

3.2 Titan

Download Trilinos and Albany from github:

```
git clone https://github.com/trilinos/Trilinos.git
git clone https://github.com/gahansen/Albany.git
```

3.2.1 Building Albany master branch with Kokkos::Serial

Load the following modules:

```
module swap PrgEnv-pgi PrgEnv-gnu
module load cray-netcdf-hdf5parallel cmake3/3.2.3 boost
```

Make a build directory for Trilinos and copy over the configuration file: `Albany/doc/do-cmake-trilinos-titan-serial`.

Build Trilinos:

```
source do-cmake-trilinos-titan-serial
make -j 8
make install
```

Make a build directory for Albany and copy over the configuration file: `Albany/doc/do-cmake-albany-titan`.

Make sure to modify the file to point to your Trilinos build directory and build:

```
source do-cmake-albany-titan
make -j 8
```

The same instructions can be used to build the Albany master branch with Kokkos::OpenMP except that the configuration file `Albany/doc/do-cmake-trilinos-titan-openmp` needs to be used instead of `Albany/doc/do-cmake-trilinos-titan-serial`.

If a linking error occurs after `make`, some modifications need to be made to the link line:

1. `cd src/`
2. Add " /usr/lib64/libxml*.a /usr/lib64/libpciaccess.a -static-libstdc++ -static-libgcc" to the end of the following file: `CMakeFiles/Albany.dir/link.txt`
3. Copy the linking script from `CMakeFiles/Albany.dir/link.txt` and paste into the terminal.
4. Repeat this process for every Albany executable.

3.2.2 Building Albany DynRankViewIntrepid2Refactor branch with Kokkos::Cuda

First checkout the Trilinos develop branch in the Trilinos directory:

```
git checkout develop
```

Checkout the Albany DynRankViewIntrepid2Refactor branch in the Albany directory:

```
git checkout DynRankViewIntrepid2Refactor
```

Load the following modules:

```
module swap PrgEnv-pgi PrgEnv-gnu
module load cray-netcdf-hdf5parallel cmake3/3.2.3 boost
module load cudatoolkit
```

Set the following compiler and GPU settings. Make sure to modify the `NVCC_WRAPPER_PATH` path to point to your `nvcc_wrapper` file:

```
NVCC_WRAPPER_PATH=$HOME/Trilinos/packages/kokkos/config/nvcc_wrapper
export NVCC_WRAPPER_DEFAULT_COMPILER=CC
export CUDA_MANAGED_FORCE_DEVICE_ALLOC=1
export CUDA_LAUNCH_BLOCKING=1
```

Make a build directory for Trilinos and copy over the configuration file: `Albany/doc/do-cmake-trilinos-titan-cuda`. Make sure to modify the `NVCC_WRAPPER_PATH` path to point to your `nvcc_wrapper` file.

Build Trilinos:

```
source do-cmake-trilinos-titan-cuda
make -j 8
make install
```

Make a build directory for Albany and copy over the configuration file: `Albany/doc/do-cmake-albany-titan`.

Make sure to modify the file to point to your Trilinos build directory and build:

```
source do-cmake-albany-titan
make -j 8
```

You might receive the following linking error:

```
/usr/bin/ld: cannot find -lcusparse
/usr/bin/ld: cannot find -lcudart
/usr/bin/ld: cannot find -lcublas
/usr/bin/ld: cannot find -lcufft
/usr/bin/ld: cannot find -lcupti
/usr/bin/ld: cannot find -lcudart
/usr/bin/ld: cannot find -lcuda
collect2: error: ld returned 1 exit status
```

If this linking error does occur after `make`, some modifications need to be made to the link line:

1. `cd src/`
2. Remove all instances of "`-Wl,-Bdynamic`", "`-Wl,-Bdynamic`", "`-lcusparse`", "`-lcudart`", "`-lcublas`", "`-lcufft`", "`-lcupti`", "`-lcudart`" and "`-lcuda`" from the following file: `CMakeFiles/Albany.dir/link.txt`
3. Add " `-Wl,-Bdynamic -lcusparse -lcudart -lcublas -lcufft -lcupti -lcudart -lcuda`" to the end of the file.

4. Copy the linking script from `CMakeFiles/Albany.dir/link.txt` and paste into the terminal.

5. Repeat this process for every Albany executable.

Chapter 4

Execution Intructions

These execution instructions are for running Aeras on the Shannon and Titan clusters. Batch scripts are used to submit jobs to a queue manager. The shell script will run when resources become available.

4.1 Shannon

Shannon uses Slurm as its resource manager and job scheduler. A set of nodes can be obtained and commands can be entered interactively by using `salloc`. A batch script can also be used to run jobs by using the command `sbatch [BatchScriptFile]`. To view information about jobs, use the command `squeue`. Here are some example batch script files for using different devices on Shannon.

4.1.1 MPI with Kokkos::Serial

```
#!/bin/bash -login

#SBATCH --job-name=MPIjob
#SBATCH --output=MPIjob.%j.out     # %j is the job number
#SBATCH --error=MPIjob.%j.err
#SBATCH --partition=fatk20x        # Shannon partition
#SBATCH --nodes=8                  # Number of nodes
#SBATCH --ntasks=128               # Number of MPI ranks
#SBATCH --ntasks-per-node=16       # Number of MPI ranks per node
#SBATCH --cpus-per-task=1          # Number of cores per MPI rank
#SBATCH --exclusive                # No other jobs can run on this node

# Load modules
module purge
module load openmpi/1.10.1/gcc/5.1.0/cuda/7.5.7 cmake/2.8.11.2

# Run MPI job
mpirun -n 128 [AlbanyExecutable] [InputFile]
```

4.1.2 MPI+OpenMP with Kokkos::OpenMP

```
#!/bin/bash -login

#SBATCH --job-name=MPIOMPjob
#SBATCH --output=MPIOMPjob.%j.out    # %j is the job number
#SBATCH --error=MPIOMPjob.%j.err
#SBATCH --partition=fatk20x          # Shannon partition
#SBATCH --nodes=8                    # Number of nodes
#SBATCH --ntasks=16                  # Number of MPI ranks
#SBATCH --ntasks-per-node=2          # Number of MPI ranks per node
#SBATCH --cpus-per-task=8            # Number of cores per MPI rank
#SBATCH --exclusive                  # No other jobs can run on this node

# Load modules
module purge
module load openmpi/1.10.1/gcc/5.1.0/cuda/7.5.7 cmake/2.8.11.2

# OpenMP environment variables
export OMP_DISPLAY_ENV=TRUE    # Displays the map of openmp threads to CPU cores
export OMP_PROC_BIND=TRUE      # Ensures that threads remain within a processor
export OMP_NUM_THREADS=8       # Number of OpenMP threads per MPI rank

# Run MPIOMP job
mpirun -n 16 --map-by ppr:1:socket:pe=8 [AlbanyExecutable] [InputFile]

# --map-by ppr:1:socket:pe=8 : maps each MPI rank to 1 CPU socket and
#                              uses 8 processing elements (openmp threads)
```

4.1.3 MPI+GPU with Kokkos::Cuda

```
#!/bin/bash -login

#SBATCH --job-name=MPIGPUjob
#SBATCH --output=MPIGPUjob.%j.out    # %j is the job number
#SBATCH --error=MPIGPUjob.%j.err
#SBATCH --partition=fatk20x          # Shannon partition
#SBATCH --nodes=8                    # Number of nodes
#SBATCH --ntasks=32                  # Number of MPI ranks
#SBATCH --ntasks-per-node=4          # Number of MPI ranks per node
#SBATCH --cpus-per-task=1            # Number of cores per MPI rank
#SBATCH --exclusive                  # No other jobs can run on this node

# Load modules
module purge
module load openmpi/1.10.1/gnu/4.7.2/cuda/7.5.7 cmake/2.8.11.2
module load nvcc-wrapper/gnu
```

```
# GPU environment variables (for CUDA UVM)
export CUDA_MANAGED_FORCE_DEVICE_ALLOC=1
export CUDA_LAUNCH_BLOCKING=1

# Run MPIGPU job
mpirun -n 32 [AlbanyExecutable] [InputFile] --kokkos-ndevice=4

# --kokkos-ndevice=4 : sets the number of GPUs used per node
```

For more information about sbatch and its different options, go to `http://slurm.schedmd.com/sbatch.html`.

4.2 Titan

Titan uses a different resource manager and job scheduler. A set of nodes can be obtained and commands can be entered interactively by using `qsub`. A batch script can also be used to run jobs by using the command `qsub [BatchScriptFile]`. To view information about your jobs, use the command `showq | grep [UserName]`. Here are some example batch script files for using different devices on Titan.

4.2.1 MPI with Kokkos::Serial

```
#!/bin/bash
#PBS -A [ProjectAllocation]
#PBS -N MPIjob                  # Job Name
#PBS -o MPIjob.$PBS_JOBID.out   # $PBS_JOBID is the job number
#PBS -j oe                      # stdout and stderr to the file above
#PBS -m e                       # Send email when job is complete
#PBS -M [EmailAddress]
#PBS -l nodes=64                # Number of nodes
#PBS -l walltime=00:10:00       # Maximum wall-clock time
                                # Format: [Hours]:[Minutes]:[Seconds]

# Load modules
source $MODULESHOME/init/bash
module swap PrgEnv-pgi PrgEnv-gnu
module load cray-netcdf-hdf5parallel cmake3/3.2.3 boost

# Change to working directory
cd $MEMBERWORK/[ProjectAllocation]/

# Make a directory for run and change to that directory
mkdir MPIjob_$PBS_JOBID
```

```
cd MPIjob_$PBS_JOBID

# Copy executable file, input file and mesh files to current directory
cp [AlbanyExecutable] ./
cp [InputFile] ./
cp [MeshFiles] ./

# Run MPI job
aprun -n 1024 [AlbanyExecutable] [InputFile]
```

4.2.2 MPI+OpenMP with Kokkos::OpenMP

```
#!/bin/bash
#PBS -A [ProjectAllocation]
#PBS -N MPIOMPjob                  # Job Name
#PBS -o MPIOMPjob.$PBS_JOBID.out   # $PBS_JOBID is the job number
#PBS -j oe                         # stdout and stderr to the file above
#PBS -m e                          # Send email when job is complete
#PBS -M [EmailAddress]
#PBS -l nodes=64                   # Number of nodes
#PBS -l walltime=00:10:00          # Maximum wall-clock time
                                   # Format: [Hours]:[Minutes]:[Seconds]

# Load modules
source $MODULESHOME/init/bash
module swap PrgEnv-pgi PrgEnv-gnu
module load cray-netcdf-hdf5parallel cmake3/3.2.3 boost

# OpenMP environment variables
export OMP_NUM_THREADS=8       # Number of OpenMP threads per MPI rank

# Change to working directory
cd $MEMBERWORK/[ProjectAllocation]/

# Make a directory for run and change to that directory
mkdir MPIOMPjob_$PBS_JOBID
cd MPIOMPjob_$PBS_JOBID

# Copy executable file, input file and mesh files to current directory
cp [AlbanyExecutable] ./
cp [InputFile] ./
cp [MeshFiles] ./

# Run MPIOMP job
aprun -n128 -N2 -d8 -cc 0-7:8-15 [AlbanyExecutable] [InputFile]

# -N  : Number of MPI ranks per node
```

```
# -d  : Number of cores per MPI rank
# -cc : List of CPU cores per MPI rank per node
#       Binds each thread to a specific core
```

4.2.3 MPI+GPU with Kokkos::Cuda

```
#!/bin/bash
#PBS -A [ProjectAllocation]
#PBS -N MPIGPUjob                  # Job Name
#PBS -o MPIGPUjob.$PBS_JOBID.out   # $PBS_JOBID is the job number
#PBS -j oe                         # stdout and stderr to the file above
#PBS -m e                          # Send email when job is complete
#PBS -M [EmailAddress]
#PBS -l nodes=64                   # Number of nodes
#PBS -l walltime=00:10:00          # Maximum wall-clock time
                                   # Format: [Hours]:[Minutes]:[Seconds]

# Load modules
source $MODULESHOME/init/bash
module swap PrgEnv-pgi PrgEnv-gnu
module load cray-netcdf-hdf5parallel cmake3/3.2.3 boost
module load cudatoolkit

# GPU environment variables (for CUDA UVM)
export CUDA_MANAGED_FORCE_DEVICE_ALLOC=1
export CUDA_LAUNCH_BLOCKING=1

# Change to working directory
cd $MEMBERWORK/[ProjectAllocation]/

# Make a directory for run and change to that directory
mkdir MPIGPUjob_$PBS_JOBID
cd MPIGPUjob_$PBS_JOBID

# Copy executable file, input file and mesh files to current directory
cp [AlbanyExecutable] ./
cp [InputFile] ./
cp [MeshFiles] ./

# Run MPIGPU job
aprun -n64 -N1 [AlbanyExecutable] [InputFile]

# -N  : Number of MPI ranks per node
```

For more information about running jobs on Titan, go to `https://www.olcf.ornl.gov/support/system-user-guides/titan-user-guide/#273`.

Chapter 5

Verification

In this section, the baroclinic instability test case [5] is simulated in order to verify the implementation of the 3D Hydrostatic equations in Aeras. Table 5.1 shows the three cubed-sphere mesh resolutions used in these simulations along with other notable parameters. Bicubic shell quadrilat-

Mesh	Resolution	# Elements	Fixed dt	Hyperviscosity Tau
uniform_30	1.0°	5400	30	5.00e15
uniform_60	0.5°	21,600	10	1.09e14
uniform_120	0.25°	86,400	5	1.18e13

Table 5.1. Cubed-sphere mesh resolutions considered for Aeras 3D Hydrostatic performance results

eral spectral elements were used for this test case for total of 16 nodes per element and 10 levels were used in the vertical direction. Each simulation is advanced in time using an explicit 4 stage, 3rd order Runge-Kutta time-stepping scheme. The results of the simulation are plotted in Figure 5.1 after a total physical time of 9 days. These plots are qualitatively similar to results obtained using the Higher-Order Methods Modeling Environment (HOMME) CAM-SE[1] dynamical core. For a more detailed discussion of the verification of the 3D Hydrostatic model in Aeras, including quantitative verification results, the reader is referred to the Aeras project final report [7].

[1] Community Atmosphere Model – Spectral Element.

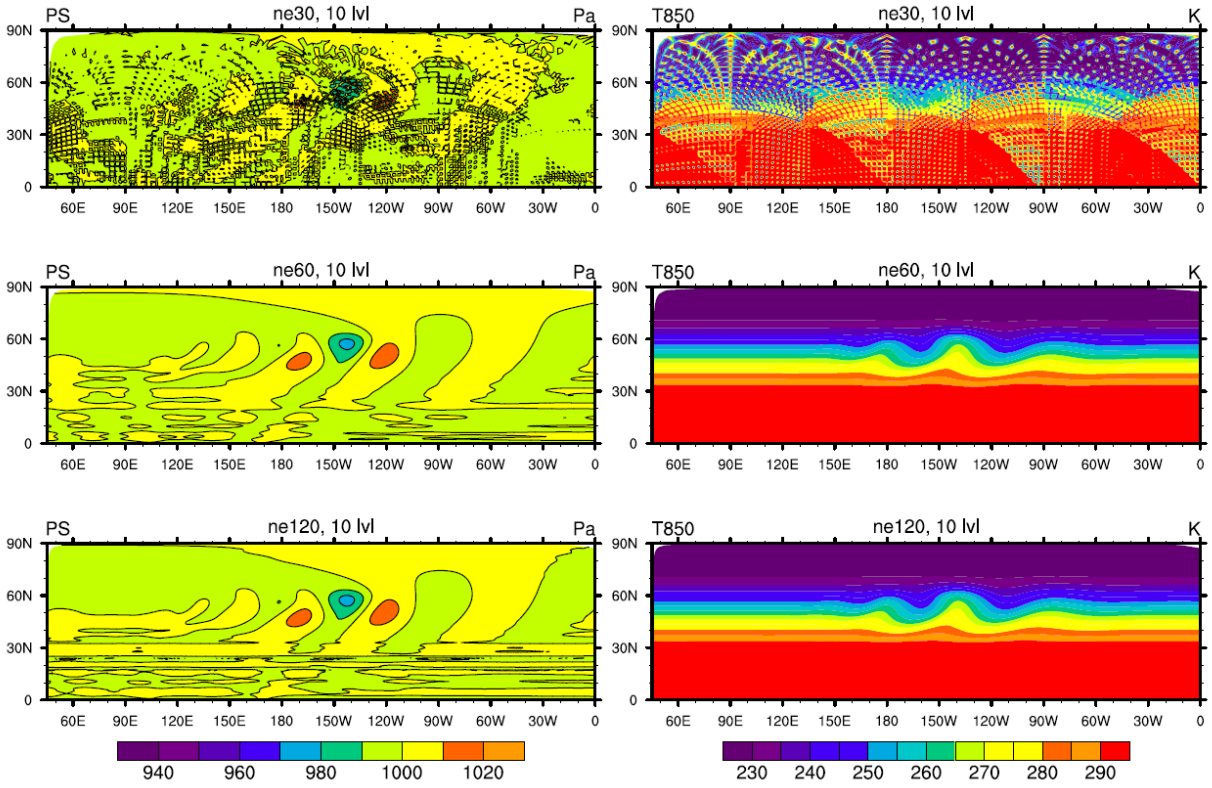


Figure 5.1. Results of baroclinic wave instability for three meshes after 9 days.

Chapter 6

Performance Analysis

In this section, the computational performance of the Kokkos implementation for the Aeras 3D Hydrostatic equations is characterized by using OpenMP and CUDA. We focus our performance analysis on the baroclinic instability test case for the meshes in Table 5.1. The same input settings were used as before except each simulation is only advanced 100 iterations. The wall-clock time of each simulation is computed by subtracting the setup time from the total wall-clock time of the simulation.

The Sandia supercomputing cluster called Shannon contains 32 nodes each with two 8-core 2.60GHz Intel Xeon E5-2670 processors and either 2 K80, 2 K20x, 4 K20x or 4 K40m NVIDIA GPUs. In contrast, the Oak Ridge Titan cluster contains 18,688 physical compute nodes each with a 16-core 2.2GHz AMD Opteron 6274 processor shown in Figure 6.1 and an NVIDIA K20X GPU. Table 6.2 show more specific details about the architectures.

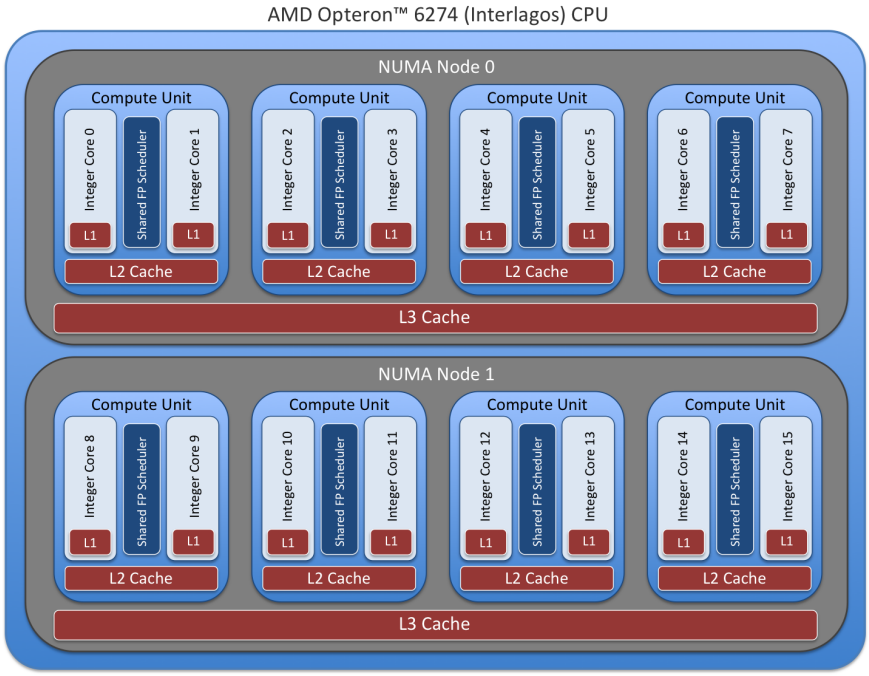


Figure 6.1. Titan 16-core AMD Opteron 6274 CPU configuration

	Shannon	Titan
Nodes	32	18688
CPU	Intel Xeon E5-2670	AMD Opteron 6274
GPU	Varies per node	1 K20x
Memory/node	128 GB	32 GB
Interconnect	QDR IB	Gemini
OS	RedHat 6.2	Cray Linux
Compiler	gcc 4.7.2	gcc 4.9.3
MPI	openmpi 1.10.1	cray-mpich 7.4.0
NVCC	7.5.7	7.5.18

Table 6.1. Supercomputing Architectures

6.1 Multicore Analysis

A multicore analysis is performed on the Titan 16-core AMD CPU architecture shown in Figure 6.1 in order to determine the best MPI+OpenMP rank/thread to core map for the 3D Hydrostatic problem. The CPU has a unique cache layout and computing scheme with various levels of cache and a shared floating point scheduler between each pair of cores. This could help or hinder performance depending on the mapping used by MPI and OpenMP.

Three maps are tested with varying levels of MPI ranks and OpenMP threads. The first named "Close" uses 1 MPI rank and 2 OpenMP threads per compute unit in order maintain a level of shared memory in L2 cache. The second named "Spread" uses 2 MPI ranks and 1 OpenMP thread per rank per compute unit in order to minimize the interaction between threads on a single compute unit. The third named "Control" uses 1 MPI rank and 1 OpenMP thread per compute unit in order to eliminate the interaction between ranks and threads on a single compute unit. The wall-clock time for all cases are shown in Table 6.2 for a distribution across two nodes (2 CPUs or 36 cores).

	Close			Spread			Control		
MPI ranks	4	8	16	4	8	16	2	4	8
OpenMP threads	8	4	2	8	4	2	8	4	2
Wall-clock Time (s)	60.7	52.0	47.2	84.8	54.2	49.1	113.1	79.7	71.9

Table 6.2. Multicore analysis of two 16-core AMD Opteron 6274 CPUs with varying MPI ranks and OpenMP threads

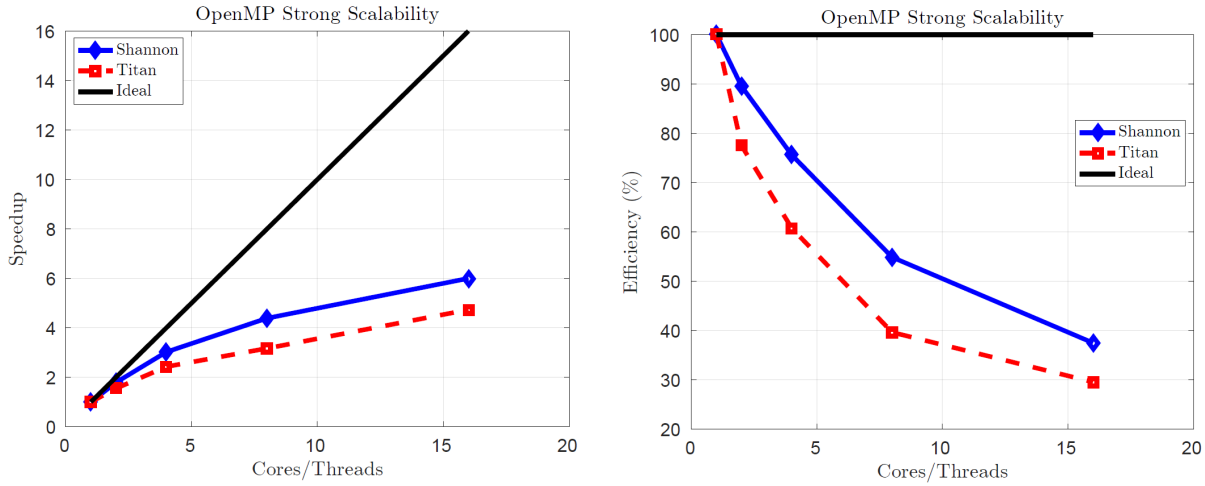
In this case, the "Close" map produced the fastest simulation times showing that it is better to maintain threads within a close proximity. Since the "Close" map produced faster times than the control, it is also safe to assume that it is better to utilize all cores even when a pair of cores share a floating point scheduler. The results also show that more MPI ranks and less OpenMP threads produce faster simulation times. This means that the MPI implementation is more efficient when compared to the Kokkos OpenMP implementation. This will be explained further in the next section.

6.2 Strong Scalability

Two strong scalability studies are performed for OpenMP, MPI and MPI+OpenMP in order to determine two crucial properties. First, a strong scalability study of OpenMP will show how well OpenMP scales when using more threads. Second, a strong scalability study of MPI and MPI+OpenMP will show whether an MPI+OpenMP architecture can be faster than pure MPI (i.e. 1 MPI rank per core). A strong scalability study for MPI+GPU was not performed because memory constraints on a single GPU.

6.2.1 OpenMP

Figure 6.2 shows the speedup and efficiency of OpenMP for up to 16 cores on both the Shannon and Titan clusters. Speedup is calculated by dividing the wall-clock time of the single core simulation and dividing it by the wall-clock time of the subsequent multithreaded simulations. Efficiency is calculated by dividing the ideal speedup by the actual speedup (i.e. 16 cores should produce a speedup of 16 but the results show a much lower speedup leading to a lower percentage). In this particular case the Shannon Intel Xeon CPU outperformed the Titan AMD Opteron CPU. The results show a very poor OpenMP scaling with 16 cores dropping to around 30-40% efficiency. For the remainder of the OpenMP simulations, the number of threads will be fixed to 8. This should lead to a 40-50% efficiency depending on the cluster.



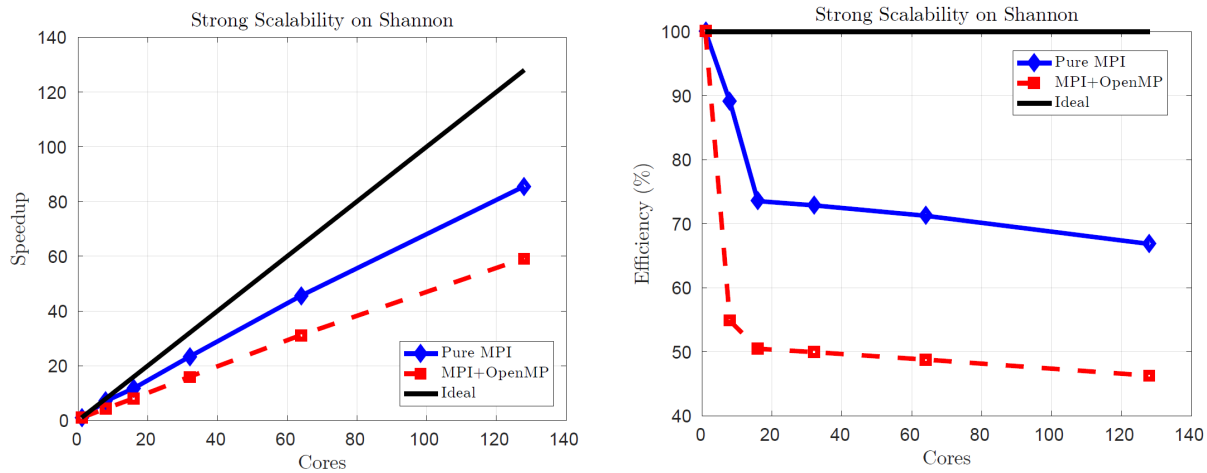
(a) Speedup

(b) Efficiency (%)

Figure 6.2. OpenMP strong scalability for Aeras 3D Hydrostatic baroclinic instability problem, uniform_30 mesh

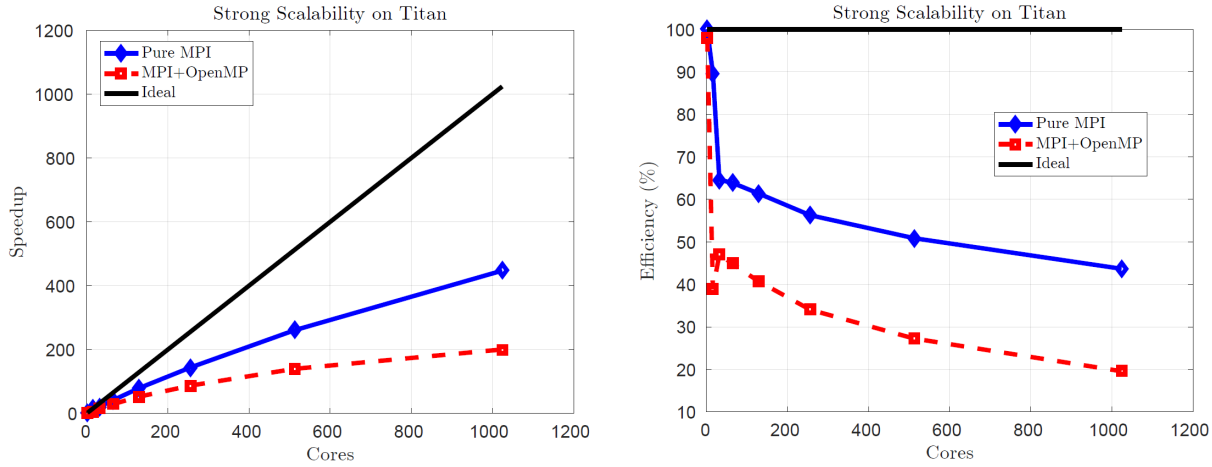
6.2.2 MPI vs. MPI+OpenMP

Figure 6.3 shows speedup and efficiency of for MPI and MPI+OpenMP for up to 128 cores on the Shannon cluster. Each node uses 16 MPI ranks or 2 MPI ranks + 8 OpenMP threads per rank. The results show that the pure MPI implementation scales much better than the Kokkos MPI+OpenMP implementation. Similar results are shown in Figure 6.4 for up to 1028 cores on the Titan cluster.



(a) Speedup **(b)** Efficiency (%)

Figure 6.3. MPI and MPI+OpenMP strong scalability study on Shannon for Aeras 3D Hydrostatic baroclinic instability problem, uniform_30 mesh



(a) Speedup **(b)** Efficiency (%)

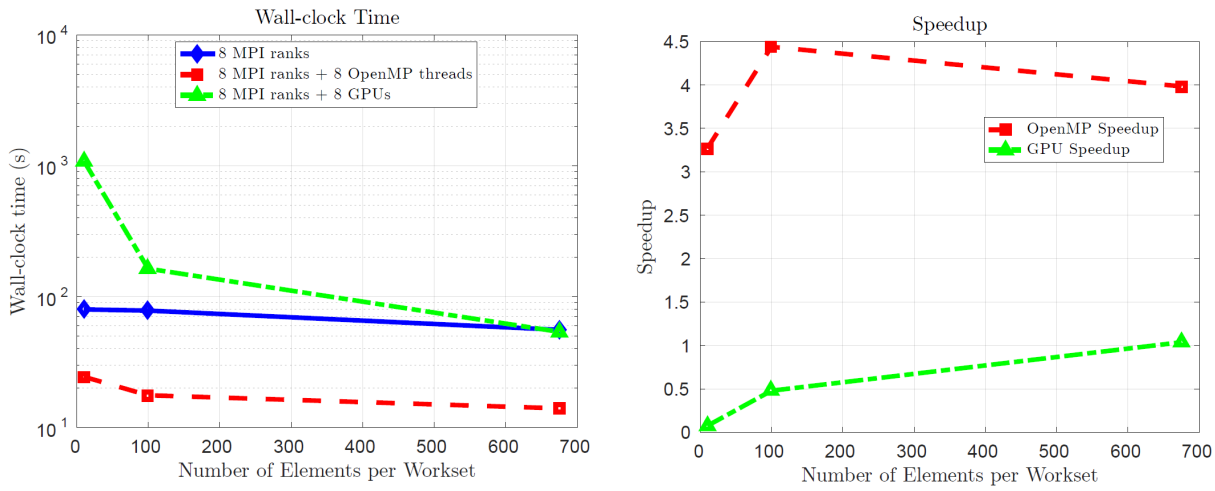
Figure 6.4. MPI and MPI+OpenMP strong scalability study on Titan for Aeras 3D Hydrostatic baroclinic instability problem, uniform_30 mesh

6.3 Workset and Weak Scalability

A workset and weak scalability study is performed on Shannon and Titan, respectively, in order to determine how well MPI+OpenMP and MPI+GPU perform with increasing problem size. The workset study uses a fixed architecture to simulate the 3D Hydrostatic baroclinic instability problem on the uniform_30 mesh with different workset sizes. A workset is defined as a set of elements which are computed on the device (e.g. CPU, GPU), and can be thought of as a threading index. Each workset is computed one at a time on a device in order to reduce the total memory on the device. The weak scalability study solves the same problem on three meshes for a workset of 1. The amount of processing power is quadrupled for each subsequent mesh since each mesh has 4 times the number of elements than the previous. These studies are performed in order to determine how well each architecture scales and to show how much speedup is acheived on each device.

6.3.1 Workset Scalability

Figure 6.5 shows the wall-clock time and OpenMP and GPU speedup over MPI for the uniform_30 mesh for different workset sizes on the Shannon cluster. The simulations using OpenMP (8 threads) are up to 4 times faster than the single-CPU simulations. The speedup in the GPU is much smaller because of the small workset size. Larger workset sizes could not be used because of memory limitations on the GPU.



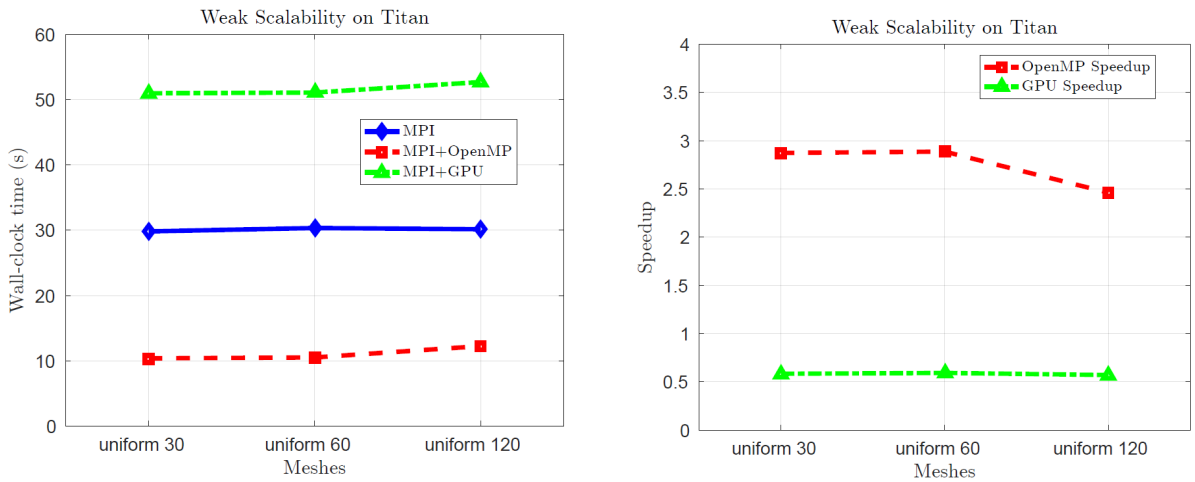
(a) Wall-clock time (s)

(b) Speedup

Figure 6.5. OpenMP and Nvidia K80 GPU speedup over MPI as a function of the number of elements per workset for Aeras 3D Hydrostatic baroclinic instability on Shannon for the uniform_30 mesh

6.3.2 Weak Scalability

Figure 6.6 shows the wall-clock time and OpenMP/GPU speedup for the three mesh resolutions shown in Table 5.1 on the Titan cluster. In this case, weak scalability is analyzed by using 32 MPI ranks, 32 MPI ranks + 8 OpenMP threads and 32 MPI ranks + 32 GPUs for the uniform_30 mesh, 128 MPI ranks, 128 MPI ranks + 8 OpenMP threads and 128 MPI ranks + 128 GPUs for the uniform_60 mesh and 512 MPI ranks, 512 MPI ranks + 8 OpenMP threads and 512 MPI ranks + 512 GPUs for the uniform_120 mesh. The results show near perfect weak scaling. The OpenMP speedup (8 threads) is approximately 3 which is a bit less than the speedup of 4 on Shannon. The GPU slows the simulation to approximately half of the time it would take to run a simulation without the GPU. This is due to the small workset sizes on each GPU (approximately 168 elements per GPU). Larger workset sizes would be more efficient but could not be achieved because of memory limitations on the GPU.



(a) Wall-clock time (s)

(b) Speedup

Figure 6.6. OpenMP and Nvidia K20X GPU speedup over MPI for the Aeras 3D Hydrostatic baroclinic instability test case on Titan

Chapter 7

Discussion

In this project, the Aeras 3D Hydrostatic finite element code was successfully refactored for performance portability across multiple next generation architectures by utilizing the Kokkos software package and programming model. The process of refactoring, building, executing and analyzing the code on high performance computing architectures has been carefully documented and presented in an effort to promote the use of modern hardware such as GPUs. This project has reached a number milestones. First, common issues in CUDA refactoring have been identified and solutions have been proposed. This will be useful for future CUDA refactoring in Albany. Second, building and executing instructions for next generation architectures have been documented for both the Sandia GPU testbed cluster, Shannon, and the Oak Ridge supercomputer, Titan. These instructions should aid future users in being able to utilize the modern hardware available. Lastly, a performance analysis was executed on the Kokkos OpenMP/GPU implementation in order to set a baseline for future performance studies. These type of studies should be performed whenever a major milestone in performance optimization is reached in order to document hardware utilization and performance gains and losses.

The performance analysis identified several major areas of improvement. The most concerning is the large memory usage of the code. Large memory usage can significantly hinder the performance of next generation architectures such as GPU. Devices have limited memory and the only way to reduce the memory usage at runtime is distribute the work across multiple GPUs. Unfortunately, this also decreases the amount of work the GPU can perform which causes a drastic decrease in performance. Constructing a global Jacobian matrix usually causes large memory allocations but this remains to be investigated. Since the problem is unsteady and requires explicit time integration, the global Jacobian should be completely removed along with the gather/scatter operation. This should significantly improve memory usage and performance.

Another area of major concern is OpenMP strong scalability. Assigning more OpenMP threads to a parallel loop seems to significantly degrade performance. Efficiency dropped to around 30-40% for 16 threads. Determining the cause of this lose in efficiency requires a profiling the code with software such as TAU. This should show how well each core is performing. OpenMP strong scalability directly affects MPI+OpenMP strong scalability. The pure MPI implementation is faster when compared to the MPI + Kokkos OpenMP implementation with 8 threads. Less OpenMP threads would have improved performance. Both the OpenMP and GPU performance can be improved by parallelizing over more indices (e.g., the number of vertical levels). In this case, the parallelization only occurs over cells but there are many areas of the code that can be parallelized

over nodes and levels. The use of shared memory could also help in operations which can be formed into a matrix multiplication.

The weak scalability studies showed near perfect weak scaling for MPI, MPI+OpenMP and MPI+GPU. Adding 8 OpenMP threads to the MPI implementation led to a speedup of around 3 to 4. The GPU doubled the computation time of the MPI implementation because of the small workset sizes. The results show that larger workset sizes will lead to better performance but the amount of memory on the GPU needs to be reduced. The CUDA profiler called `nvprof` can be used to improve the performance on the GPU. The profiler shows that CUDA UVM was not a limiting issue in this case because the workset sizes were so small and memory transfer was minimal. Once larger workset are used, CUDA UVM will start becoming a more significant contributor to performance loss. In this case, it would be ideal to perform the entire iteration cycle on the GPU and use remote direct memory access (RDMA) to completely eliminate the memory transfer from host to device.

References

[1] I. Demeshko, W. Spotz, I. Tezaur, O. Guba, A. Salinger, R. Pawlowski, and M. Heroux. Towards performance-portability of the albany finite element analysis code using the kokkos library. *J. HPC Appl. (under review)*, 2016.

[2] H. Carter Edwards and Christian R. Trott. Kokkos tutorials. `https://github.com/kokkos/kokkos-tutorials`, 2016. Online; accessed 7-October-2016.

[3] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[4] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R.S. Tuminaro, J.M. Willenbring, A. Williams, and K.S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3), 2005.

[5] C. Jablonowski and D. L. Williamson. A baroclinic instability test case for atmospheric dynamical cores. *Q. J. R. Meteorol. Soc.*, 132, 2006. doi:10.1256/qj.06.n.

[6] A. Salinger, R. Bartlett, A. Bradley, Q. Chen, I. Demeshko, X. Gao, G. Hansen, A. Mota, R. Muller, E. Nielsen, J. Ostien, R. Pawlowski, M. Perego, E. Phipps, W. Sun, and I. Tezaur. Albany: Using agile components to develop a flexible, generic multiphysics analysis code. *Int. J. Multiscale Comput. Engng. (in press)*, 2016.

[7] W. Spotz, P. Bosler, S. Bova, I. Demeshko, J. Fike, O. Guba, J. Overfelt, A. Salinger, T. Smith, I. Tezaur, and J. Watkins. The aeras next generation global atmosphere model. Sandia National Laboratories Report, SAND No. 2016-XXXX, Sandia National Laboratories, Albuquerque, NM, 2016.

[8] W. Spotz, T. Smith, I. Demeshko, and J. Fike. Aeras: a next generation global atmosphere model. *Procedia Computer Science*, 51:2097–2106, 2015.

[9] M. A. Taylor. Conservation of mass and energy for the moist atmospheric primitive variables. In P. H. Lauritzen, C. Jablonowski, M. A. Taylor, and R. D. Nair, editors, *Numerical Techniques for Global Atmospheric Models*, chapter 12. Springer, 2012.

[10] Christian R. Trott, Mark Hoemmen, Simon D. Hammond, and H. Carter Edwards. *Kokkos: The Programming Guide*. Sandia National Laboratories, 2015. Online; accessed 7-October-2016.

DISTRIBUTION:

1	MS 1320	Bill Spotz, 01446
1	MS 1320	Tom Smith, 01446
1	MS 1318	Andy Salinger, 01442
1	MS 1321	Pete Bosler, 01446
1	MS 1322	Oksana Guba, 01441
1	MS 8343	Alejandro Mota, 9042
1	MS 8343	Coleman Alleman, 9012
1	MS 8954	Cosmin Safta, 9159
1	MS 1426	Dan Sunderland, 1318
1	MS 1442	Michael Deakin, 1320
1	MS 1442	Mauro Perego, 1320
1	MS 1426	Christian Trott, 1318
1	MS 1426	Carter Edwards, 1318
1	MS 1426	Mark Hoemmen, 1320
1	MS 1426	Siva Rajamanickam, 1320
1	MS 8954	Jerry McNeish, 9159
1	MS 8959	Gayle Thayer, 9158
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories