Performance portability for next generation HPC architectures in E3SM via the Kokkos programming model


Sandia National Laboratories, Albuquerque, NM

July 2, 2018
To answer important question on climate we need finer resolutions.

Upcoming computing facilities will bring us close to or above Exascale capabilities, on different architectures.
1 The E3SM and CMDV projects

2 Kokkos and HOMME

3 From HOMME to HOMMEXX

4 Results
What is E3SM?

- DOE effort for a high resolution earth model.
- Branched from Community Earth System Model (CESM) in 2014.
- Modular library, with several components: atmosphere dynamics/physics, land, land-ice, ocean, sea-ice, biogeochemistry, ...
- All component can run with variable-resolution, unstructured grids.
- Mostly written in Fortran 90.
- Broad variety of time and space scales.
- 2018: E3SM version 1 is released in April.
Project goal is to improve

- trustworthiness of the model for decision support,
- code agility for adapting to exascale architectures,
- productivity through leveraging of cutting-edge computational science.

Coding challenge: have a single code base, performant on a variety of architectures, and capable of rapidly adapting to new ones.

Task: study the feasibility of using Kokkos (a library for on-node parallelism, more on it later) to achieve a single code base which is performant on a variety of architectures (CPU, MIC, GPU).

Path: convert a component of E3SM, namely the atmosphere component HOMME (more on that later), to C++, using Kokkos.

Metrics: correctness (bit-for-bit with original HOMME), and performance (on par with original HOMME on CPU/MIC).
Project goal is to improve

- trustworthiness of the model for decision support,
- code agility for adapting to exascale architectures,
- productivity through leveraging of cutting-edge computational science.

Coding challenge: have a single code base, performant on a variety of architectures, and capable of rapidly adapting to new ones.

- **Task**: study the feasibility of using Kokkos (a library for on-node parallelism, more on it later) to achieve a single code base which is performant on a variety of architectures (CPU, MIC, GPU).
- **Path**: convert a component of E3SM, namely the atmosphere component HOMME (more on that later), to C++, using Kokkos.
- **Metrics**: correctness (bit-for-bit with original HOMME), and performance (on par with original HOMME on CPU/MIC).
The Kokkos library

- Developed at Sandia National Labs, written in C++ (with C++11 required).
- Provides templated constructs for on-node parallel execution: execution space (host vs device), execution policy (range vs team), parallel operation (for, scan, reduce).
- Provides template abstraction for a multidimensional array: data type, memory space (host, device, UVM), layout (left, right, ...), memory access/handling (atomic, unmanaged, ...).
- Supports several back-ends: Serial, OpenMP, Pthreads, Cuda, ....
- Available at http://github.com/kokkos/kokkos.
High-Order Methods Modeling Environment (HOMME)

- Component of E3SM (and CESM) for dynamics and transport in the atmosphere.
- Accounts for 20-25% of total run time of typical fully-coupled simulation.
- Highly optimized for MPI and OpenMP parallelism.
- Horizontal (2D) and vertical (1D) differential operators are decoupled.
- Spectral Element Method (SEM) in the horizontal direction.
- Eulerian or Lagrangian schemes for vertical operators.
- Solves for 4 prognostic variables (2 horizontal velocities, temperature, pressure), and the transport of $N$ tracers (usually, $N \sim 10-40$).
From HOMME to HOMMEXX

- Incremental conversion of original Fortran code to C++.
- Heavily tested (∼85% of kernels are individually tested).
- Bit-for-bit agreement with original implementation.
- Minimization of architecture-specific code.
- Primary design goals:
  - expose parallelism,
  - maximize vectorization,
  - minimize memory movement.
From HOMME to HOMMEXX

- Incremental conversion of original Fortran code to C++.
- Heavily tested (≈85% of kernels are individually tested).
- Bit-for-bit agreement with original implementation.
- Minimization of architecture-specific code.
- Primary design goals:
  - expose parallelism,
  - maximize vectorization,
  - minimize memory movement.
HOMMEXX design: exposing parallelism

- HOMME has 3 layers of nested for loops: element(× # variables), GLL points, vertical levels.
- Elements and levels independently processed through majority of code.
- 2D differential operators couple GLL points.
- Kokkos supports up to 3 levels of hierarchical parallelism:
  - team level: a parallel region over the number of teams (of threads)
  - thread level: a parallel region over the number of threads in a team
  - vector level: a parallel region over the number of vector lanes of a thread.
- Hierarchical parallelism allows to expose maximum parallelism with minimal index bookkeeping.
A simple nested loop:

```c
for (int i=0; i<dim0; ++i) {
    for (int j=0; j<dim1; ++j) {
        for (int k=0; k<dim2; ++k) {
            // do some work on i, j, k
        }
    }
}
```

Expose parallelism by flattening:

```c
for (int idx=0; idx<dim0*dim1*dim2; ++idx) {
    int i = idx / (dim1*dim2);
    int j = idx / dim2;
    int k = idx % dim2;
    // do some work on i, j, k
}
```

Embarassingly parallel.
A simple nested loop:

```c
for (int i=0; i<dim0; ++i) {
    for (int j=0; j<dim1; ++j) {
        for (int k=0; k<dim2; ++k) {
            // do some work on i, j, k
        }
    }
}
```

Expose parallelism by flattening:

```c
for (int idx=0; idx<dim0*dim1*dim2; ++idx) {
    int i = idx / (dim1*dim2);
    int j = idx / dim2;
    int k = idx % dim2;
    // do some work on i, j, k
}
```
A simple nested loop:

```c
for (int i=0; i<dim0; ++i) {
    for (int j=0; j<dim1; ++j) {
        for (int k=0; k<dim2; ++k) {
            // do some work on i, j, k
        }
    }
}
```

Expose parallelism by flattening:

```c
for (int idx=0; idx<dim0*dim1*dim2; ++idx) {
    int i = idx / (dim1*dim2);
    int j = idx / dim2;
    int k = idx % dim2;
    // do some work on i, j, k
}
```

Embarassingly parallel.
HOMMEXX design: exposing parallelism

A more complex scenario

```c
for (int ie=0; ie<num_elements; ++ie) {
    for (int idx=0; idx<NP*NP; ++idx) {
        int i = idx / NP; int j = idx % NP;
        double v0 = v(ie,0,i,j);  double v1 = v(ie,1,i,j);
        buf(0,i,j) = (J(0,0,i,j)*v0 + J(1,0,i,j)*v1)*metdet(i,j);
        buf(1,i,j) = (J(0,1,i,j)*v0 + J(1,1,i,j)*v1)*metdet(i,j);
    }
    for (int idx=0; idx<NP*NP; ++idx) {
        int i = idx / NP; int j = idx % NP;
        double dudx = 0.0, dvdy = 0.0;
        for (int k = 0; k < NP; ++k) {
            dudx += D(j,k) * buf(0,i,k);
            dvdy += D(i,k) * buf(1,k,j);
        }
        div(ie,i,j) = (dudx+dvdy) / (metdet(i,j)*rearth);
    }
    ...
}
```

July 2, 2018
HOMMEXX design: exposing parallelism

A more complex scenario

```c
for (int ie=0; ie<num_elements; ++ie) {
    for (int idx=0; idx<NP*NP; ++idx) {
        int i = idx / NP; int j = idx % NP;
        double v0 = v(ie,0,i,j); double v1 = v(ie,1,i,j);
        buf(0,i,j) = (J(0,0,i,j)*v0 + J(1,0,i,j)*v1)*metdet(i,j);
        buf(1,i,j) = (J(0,1,i,j)*v0 + J(1,1,i,j)*v1)*metdet(i,j);
    }
}
```

```c
for (int idx=0; idx<NP*NP; ++idx) {
    int i = idx / NP; int j = idx % NP;
    double dudx = 0.0, dvdy = 0.0;
    for (int k = 0; k < NP; ++k) {
        dudx += D(j,k) * buf(0,i,k);
        dvdy += D(i,k) * buf(1,k,j);
    }
    div(ie,i,j) = (dudx+dvdy) / (metdet(i,j)*rearth);
}
```

...
A more complex scenario

```c
for (int ie=0; ie<num_elements; ++ie) {
    for (int idx=0; idx<NP*NP; ++idx) {
        int i = idx / NP; int j = idx % NP;
        double v0 = v(ie,0,i,j); double v1 = v(ie,1,i,j);
        buf(0,i,j) = (J(0,0,i,j)*v0 + J(1,0,i,j)*v1)*metdet(i,j);
        buf(1,i,j) = (J(0,1,i,j)*v0 + J(1,1,i,j)*v1)*metdet(i,j);
    }
}
```

```c
for (int idx=0; idx<NP*NP; ++idx) {
    int i = idx / NP; int j = idx % NP;
    double dudx = 0.0, dvdy = 0.0;
    for (int k = 0; k < NP; ++k) {
        dudx += D(j,k) * buf(0,i,k);
        dvdy += D(i,k) * buf(1,k,j);
    }
    div(ie,i,j) = (dudx+dvdy) / (metdet(i,j)*rearth);
}
```
A more complex scenario

```c
for (int ie = 0; ie < num_elements; ++ie) {
    for (int idx = 0; idx < NP*NP; ++idx) {
        int i = idx / NP; int j = idx % NP;
        double v0 = v(ie, 0, i, j);
        double v1 = v(ie, 1, i, j);
        buf(0, i, j) = (J(0, 0, i, j)*v0 + J(1, 0, i, j)*v1)*metdet(i, j);
        buf(1, i, j) = (J(0, 1, i, j)*v0 + J(1, 1, i, j)*v1)*metdet(i, j);
    }

    // Team barrier
    shared within team
}
```

```c
for (int idx = 0; idx < NP*NP; ++idx) {
    int i = idx / NP; int j = idx % NP;
    double dudx = 0.0, dvdy = 0.0;
    for (int k = 0; k < NP; ++k) {
        dudx += D(j, k) * buf(0, i, k);
        dvdy += D(i, k) * buf(1, k, j);
    }
    div(ie, i, j) = (dudx+dvdy) / (metdet(i, j)*rearth);
}
...
Core data type is a packed (Vector) of N doubles.

On CPU, N varies: KNL/SKX N=8, HSW N=4.

On GPUs, N=1 (no SIMD, only SIMT).

Vectorization via call to compiler intrinsics.

Two natural choices for vectorization: GLL points and vertical levels. But:

- 2D differential operator much more frequent than 1D vertical integrals, and
- matching N with # vertical level feasible, while matching N with # of GLL point could become prohibitive.

⇒ Vectorization over vertical levels (and data laid out accordingly in memory).
Results: tested architectures

(IB) Intel Ivy Bridge: 2 sockets/node, 12 cores/socket, 2 threads/core, DDR3
(HSW) Intel Haswell: 2 sockets/node, 16 cores/socket, 2 threads/core, DDR4
(KNL) Intel Xeon Phi: 68 cores/node, 4 threads/core, HBM+DDR4
(SKX) Intel Skylake: 2 sockets/node, 24 cores/socket, 2 threads/core, DDR4
(P9) IBM Power9: 2 sockets/node, 10 cores/socket, 4 threads/core, DDR4
(P100) NVidia Pascal: 2 sockets/node, 2 GPUs/socket, 1792 DP cores/GPU
(V100) NVidia Volta: 2 sockets/node, 2 GPUs/socket, 2560 DP cores/GPU

Note: IB, HSW and KNL tested at large scale, SKX, P100, V100, P9 only available on testbeds.
Results: strong scaling at large scale
Results: single node performance (40 tracers)

Power consumption (at high workload):
- IB: 260W
- HSW: 360W
- KNL: 260W
- SKX: 330W
- P9: 360W (?)
- P100: 190W
- V100: 200W
Results: single node performance (no tracers)

![Graph showing performance vs. number of elements for different processors](image)

**Power consumption (at high workload):**
- IB: 260W
- HSW: 360W
- KNL: 260W
- SKX: 330W
- P9: 360W (?)
- P100: 190W
- V100: 200W

July 2, 2018
Conclusions

- With Kokkos, HOMMEXX can run on multiple architectures with a (mostly) single implementation.
- HOMMEXX slightly faster than HOMME on CPU/MIC (∼1.1× on HSW, and up to 1.4× on KNL).
- Reasonable performance on GPUs. Need to test performance with NVLink 2.0.
- Skylake-like architectures could become very interesting for E3SM.
- C++ and Kokkos is a viable path to achieve a performance portable code.