

Exceptional service in the national interest



Sandia
National
Laboratories

SAND2018-12623PE

Exploring the use of Kokkos in HOMME to achieve performance on multiple architectures

L.Bertagna, M.Deakin, O.Guba, D.Sunderland,
A.Bradley, I.Tezaur, M.Taylor, A.Salinger

Sandia National Laboratories, Albuquerque, NM

November 6th, 2018

SAND 2018-0000

atories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, a Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract

1 Coding in the spirit of HPC architectures

2 The CMDV project

3 From HOMME to HOMMEXX

4 Results

The spring of HPC architectures



- In the last decade, we have seen radically different architectures arise in production clusters (CPU, MIC, GPU).
- More architectures are coming to new clusters soon or may do so in the future: ARM, P9/P10, FPGA, CSA,...
- Each architecture has peculiar characteristics that must be considered in order to efficiently use it.
- By far, most radical differences are between GPU and non-GPU: threads organization, memory access, vectorization, ...
- Two important keywords:
 - performance: achieve a throughput that is reasonably close to the maximum possibly achievable (due to machine or algorithmic limits);
 - portability: have a code whose performance on different architectures is equally good.

The maintainability challenge



Challenge: how to get performance portable code while separating science and HPC concerns?

Three main strategies to separate scientist and developer concerns, and achieve performance-portable code:

- **compiler directives:** hint/force compiler on how to optimize (OpenACC, OpenMP)
- **GP performance libraries:** delegate arch-specific optimizations to libraries written in the native language (Kokkos, Raja, OCCA)
- **DS languages/libraries:** add source-to-source compilation step, where arch-specific choices are made to generate an optimal source code (Stella, Claw, GridTools).

The CMDV project



CMDV: **C**limate **M**odel **D**evelopment and **V**alidation. Goals:

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

The CMDV project

CMDV: Climate Model Development and Validation. Goals:

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

Software modernization:

- **Task:** study the feasibility of using Kokkos (a GP 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), 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

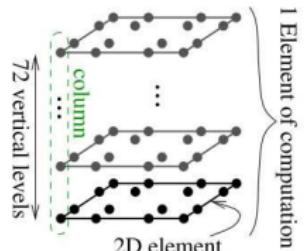
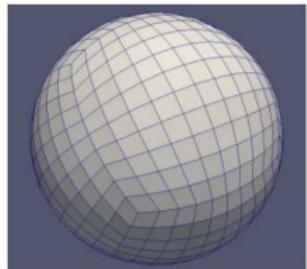


- Developed at Sandia National Labs, written in C++ (with C++11 required).
- Provides templated constructs for on-node parallel execution: execution space (host vs device), execution policy (range vs team), parallel operation (for, scan, reduce).
- Provides template abstraction for a multidimensional 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>.

The HOMME dycore

HOMME:High-Order Methods Modeling Environment

- 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.
- 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, $N \sim 10-40$).

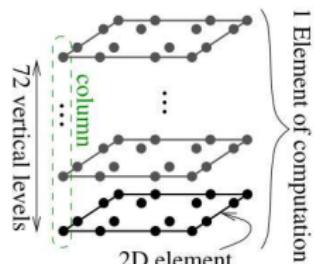


- Incremental conversion of original Fortran code to C++.
- Heavily tested (~85% of kernels are individually tested).
- Bit-for-bit agreement with original implementation.
- Minimization of architecture-specific code.
- Primary design goals:
 - expose parallelism,
 - maximize vectorization,
 - minimize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

- 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(\times # 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.



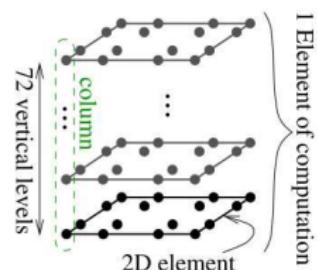
HOMMEXX design: exposing vectorization

- 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.

⇒ Vectoriation over vertical levels (and data laid out accordingly in memory).



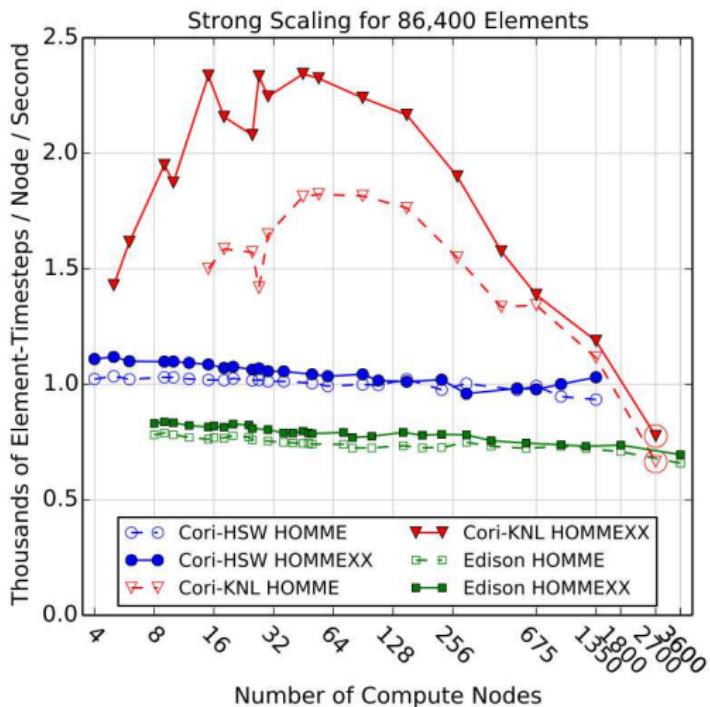
Results: tested architectures



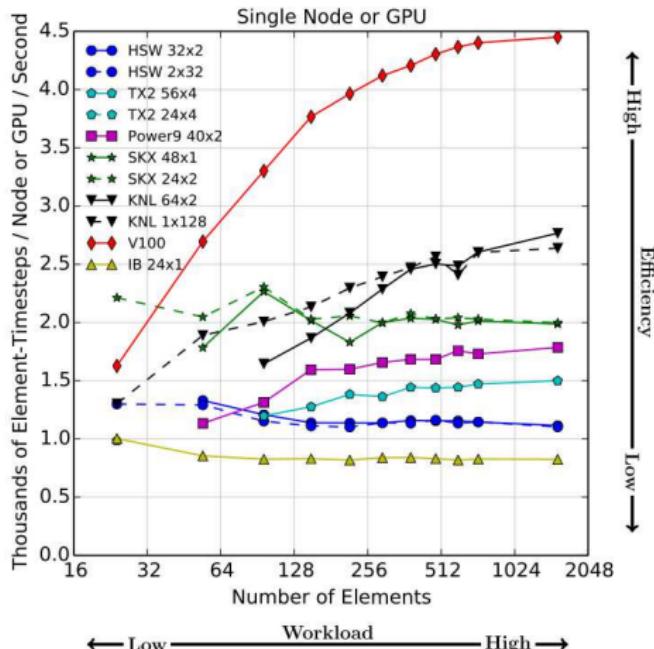
- (IB) **Intel Ivy Bridge**: 2 sockets/node, 12 cores/socket, 2 threads/core, DDR3
- (HSW) **Intel Haswell**: 2 sockets/node, 16 cores/socket, 2 threads/core, DDR4
- (KNL) **Intel Xeon Phi**: 68 cores/node, 4 threads/core, HBM+DDR4
- (SKX) **Intel Skylake**: 2 sockets/node, 24 cores/socket, 2 threads/core, DDR4
- (P9) **IBM Power9**: 2 sockets/node, 10 cores/socket, 4 threads/core, DDR4
- (TX2) **Cavium ThunderX2**: 2 sockets/node, 28 cores/socket, 4 threads/core, DDR4
- (V100) **NVidia Volta**: 2 sockets/node, 2 GPUs/socket, 2560 DP cores/GPU

Note: IB, HSW and KNL tested at large scale, SKX, P9, TX2, V100 only available on testbeds.

Results: strong scaling at large scale



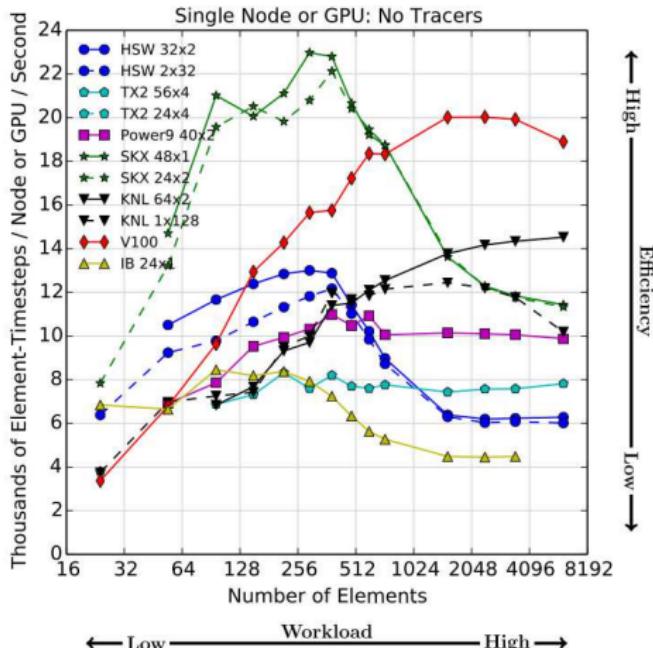
Results: single node performance (40 tracers)



Power consumption
(at high workload):

- IB: 260W
- HSW: 360W
- KNL: 260W
- SKX: 330W
- P9: 360W
- TX2: 300W (?)
- V100: 200W

Results: single node performance (no tracers)



Power consumption (at high workload):

- IB: 260W
- HSW: 360W
- KNL: 260W
- SKX: 330W
- P9: 360W
- TX2: 300W (?)
- V100: 200W

Conclusions



- With Kokkos, HOMMEXX can run on multiple architectures with a (mostly) single implementation.
- HOMMEXX slightly faster than HOMME on CPU/MIC ($\sim 1.1\times$ on HSW, and up to $1.4\times$ on KNL).
- Reasonable performance on GPUs. Need to test performance with NVLink 2.0.
- C++ and Kokkos is a *viable* path to achieve a performance portable code; in particular:
 - + relies on state-of-the art on-node parallelism library;
 - + benefits from rich language features and libraries in C++;
 - has a syntax that can be overwhelming for new developers;
 - is not the code climate scientists are *used* to see.

Thank you!

Questions?

The maintainability challenge

All approaches have some good and bad aspects

Directives:

- Pros: can get a working version quickly, limited amount of lines to add.
- Cons: it's a different language from the native one, and may lead to code duplication.

General Purpose Performance Libraries:

- Pros: hide some aspect of performance optimization while leaving enough room for ad-hoc tuning
- Cons: requires some effort to identify the best optimization choices for the particular problem.

Domain Specific Languages/Libraries:

- Pros: hides virtually all the HPC choices, leaving a friendly looking source code for scientists.
- Cons: limits the ability to add ad-hoc optimization choices.

A simple nested loop:

```
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  
        }  
    }  
}
```

A simple nested loop:

```
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:

```
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:

```
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:

```
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: divergence on the sphere

```
for (int ie=0; ie<num_elements; ++ie) {
    some_array_type buf(2, NP, NP);
    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);
    }
}
```

HOMMEXX design: exposing parallelism

A more complex scenario: divergence on the sphere

```
for (int ie=0; ie<num_elements; ++ie) { ← || over # teams
    some_array_type buf(2, NP, NP);
    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);
    }
}
```

HOMMEXX design: exposing parallelism

A more complex scenario: divergence on the sphere

```
for (int ie=0; ie<num_elements; ++ie) { ← || over # teams
    some_array_type buf(2, NP, NP);
    for (int idx=0; idx<NP*NP; ++idx) { ←———— || over # threads
        int i = idx / NP; int j = idx % NP;           in a team
        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) { ←———— || over # threads
        int i = idx / NP; int j = idx % NP;           in a team
        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);
    }
}
```

HOMMEXX design: exposing parallelism

A more complex scenario: divergence on the sphere

```
for (int ie=0; ie<num_elements; ++ie) { ← || over # teams
    some_array_type buf(2, NP, NP);
    for (int idx=0; idx<NP*NP; ++idx) { ← || over # threads
        int i = idx / NP; int j = idx % NP;           in a team
        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
    for (int idx=0; idx<NP*NP; ++idx) { ← || over # threads
        int i = idx / NP; int j = idx % NP;           in a team
        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);
    }
}
```

Annotations:

- Blue arrows point to the first two nested loops, indicating parallel execution over teams.
- Green arrows point to the innermost loop and the assignment statements within it, indicating parallel execution within a team.
- A red box highlights the assignment statement `buf(0, i, j) = ...`, and a red arrow points to the corresponding statement `buf(1, i, j) = ...` below it, with the text "shared within team" written in red.