Sandia National Laboratories

Exceptional service in the national interest

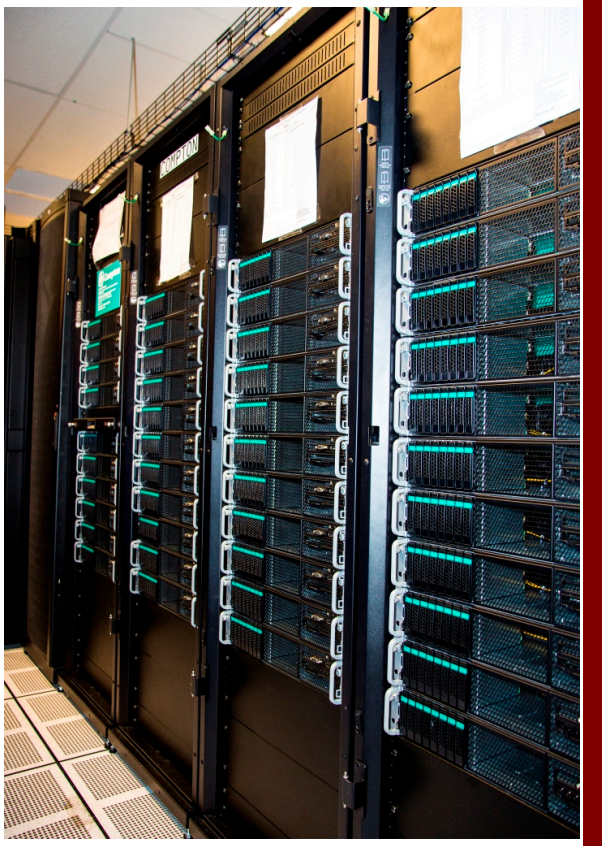
CCR
Center for Computing Research

Kokkos: Performance Portability *and* Productivity for C++ Applications

2016 Berkeley C++ Summit
October 17, 2016

H. Carter Edwards
Christian Trott

SAND2016-10230 PE
Unclassified Unlimited Release (UUR)



Advanced Simulation & Computing

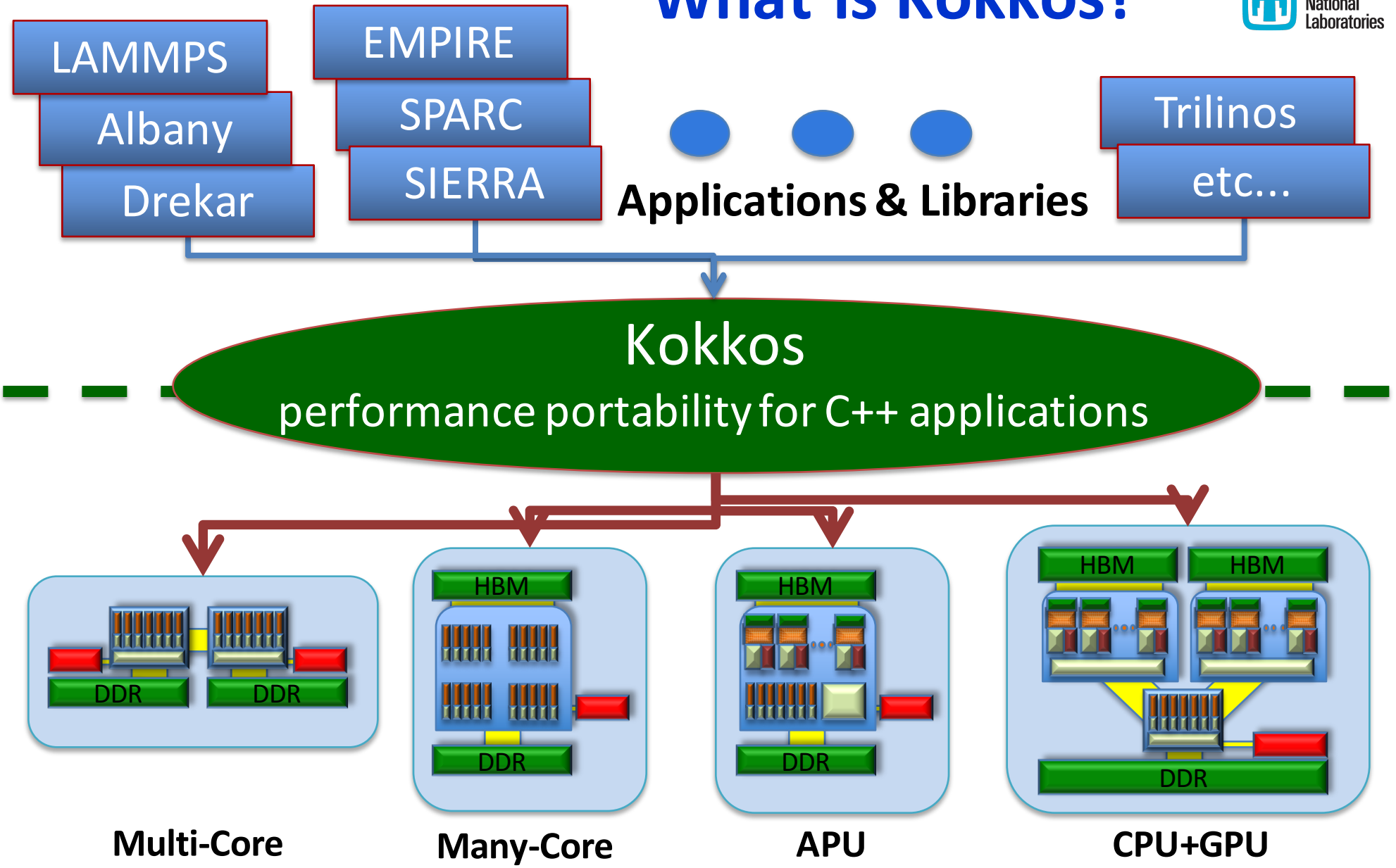
U.S. Department of Energy

National Nuclear Security Administration

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. SAND NO. 2011-XXXXP

What is Kokkos?

Sandia National Laboratories



What is *Kokkos*?

Sandia National Laboratories

- **ΚΌΚΚΟΣ** (Greek, not an acronym)
  - Translation: "granule" or "grain" ; *like grains of sand on a beach*
- **Performance Portable Thread-Parallel Programming Model**
  - E.g., "X" in "MPI+X" ; **not** a distributed-memory programming model
  - Application identifies its parallelizable grains of computations *and* data
  - Kokkos maps those computations onto cores *and* that data onto memory
- **Fully Performance Portable C++11 Library Implementation**
  - ***Not* a language extension (e.g., OpenMP, OpenACC, OpenCL, …)**
  - **Production** – open source at https://github.com/kokkos/kokkos
  - ✓ **Compilers: GNU, LLVM, Intel, NVIDIA, IBM XL, Cray**
  - ✓ **Multicore CPU** - including NUMA architectural concerns
  - ✓ **Intel Xeon Phi (KNC)** – toward DOE's Trinity (ATS-1) supercomputer
  - ✓ **NVIDIA GPU (Kepler)** – toward DOE's Sierra (ATS-2) supercomputer
  - ✓ **IBM Power 8** – toward DOE's Sierra (ATS-2) supercomputer
  - ○ **AMD Fusion** – prototype back-end via collaboration with AMD using HCC

Abstractions: Patterns, Policies, and Spaces

Sandia National Laboratories

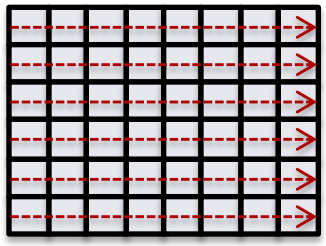
- Parallel Pattern of user's computations
  - parallel_for, parallel_reduce, parallel_scan, task-graph, ... *(extensible)*
- Execution Policy tells *how* user computation will be executed
  - Static scheduling, dynamic scheduling, thread-teams, ... *(extensible)*
- Execution Space tells *where* user computations will execute
  - Which cores, numa region, GPU, ... *(extensible)*
- Memory Space tells *where* user data resides
  - Host memory, GPU memory, high bandwidth memory, ... *(extensible)*
- Layout (policy) tells *how* user data is laid out in memory
  - Row-major, column-major, array-of-struct, struct-of-array ... *(extensible)*
- Differentiating: Layout and Memory Space
  - Versus other programming models (OpenMP, OpenACC, ...)
  - Critical for performance portability ...

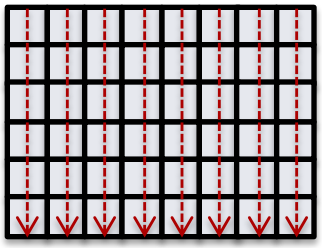
Layout Abstraction: Multidimensional Array

Sandia National Laboratories

- **Classical (50 years!) data pattern for science & engineering codes**
  - Computer languages hard-wire multidimensional array layout mapping
  - Problem: different architectures *require* different layouts for performance
  - ➢ **Leads to architecture-specific versions of code to obtain performance**
  - E.g., "Array of Structure" ↔ "Structure of Array" redesigns

e.g., "row-major" CPU caching





e.g., "column-major" GPU coalescing

- **Kokkos *separates* layout from user's computational code**
  - *Choose* layout for architecture-specific memory access pattern
    - ➢ **Without modifying user's computational code**
  - **Polymorphic** layout via C++ template meta-programming *(extensible)*
    - ➢ **e.g., Hierarchical Tiling layout** (array of structure of array)
- **Bonus: easy/transparent use of special data access hardware**
  - Atomic operations, GPU texture cache, … *(extensible)*

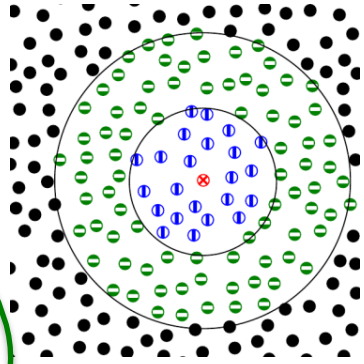
Performance Impact of Data Layout

- **Molecular dynamics computational kernel in miniMD**
- **Simple Lennard Jones force model:**

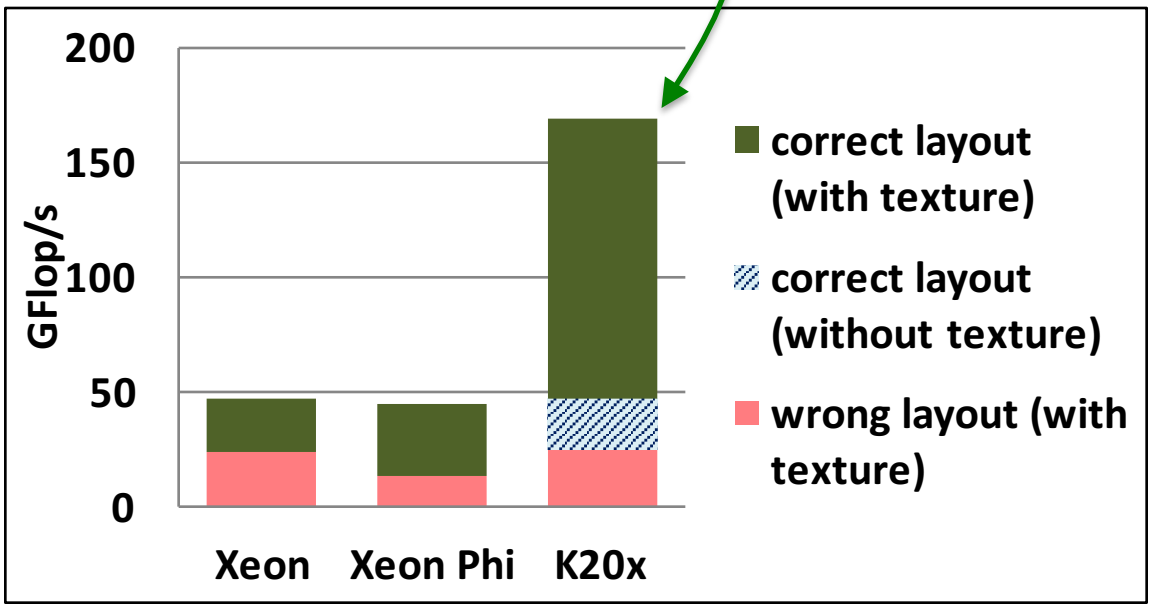
$$F_i = \sum_{j,\, r_{ij} < r_{cut}} 6\left[\left(\frac{\sigma}{r_{ij}}\right)^{7} - 2\left(\frac{\sigma}{r_{ij}}\right)^{13}\right]$$

- **Atom neighbor list to avoid N^2 computations**

```
pos_i = pos(i);
for( jj = 0; jj < num_neighbors(i); jj++) {
  j = neighbors(i,jj);
  r_ij = pos(i,0..2) - pos(j,0..2); // random read 3 floats
  if (|r_ij| < r_cut) f_i += 6*e*((s/r_ij)^7 - 2*(s/r_ij)^13)
}
f(i) = f_i;
```



- **Test Problem**
  - 864k atoms, ~77 neighbors
  - 2D neighbor array
  - Different layouts CPU vs GPU
  - Random read 'pos' through GPU texture cache
- **Large performance loss with wrong data layout**

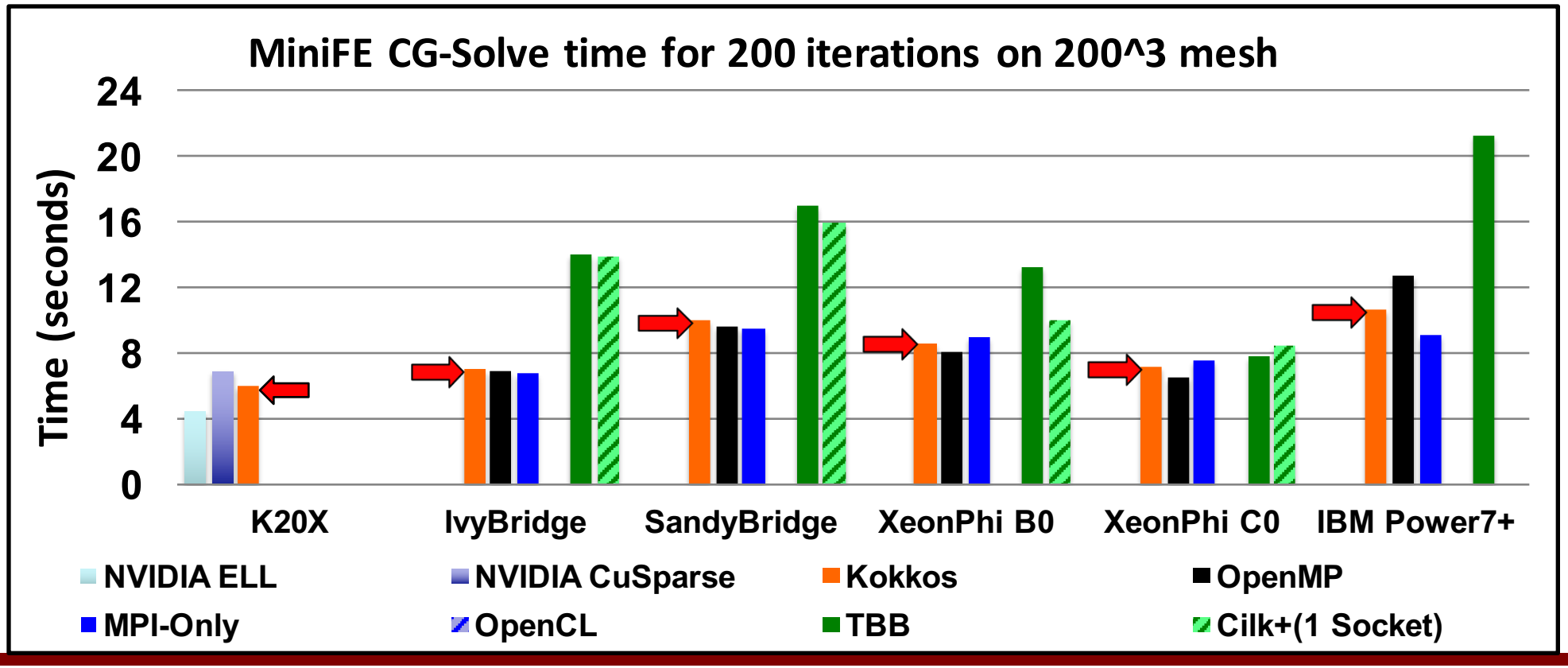


Performance Overhead?

Sandia National Laboratories

Kokkos is competitive with other programming models

- Regularly performance-test mini-applications on Sandia's next generation platform (NGP) test beds
- MiniFE: finite element linear system iterative solver mini-app
  - Compare to versions with architecture-specialized programming models



6

Performance Portability & Future Proofing

Sandia National Laboratories

Integrated mapping of users' parallel computations *and* data through abstractions of patterns, policies, spaces, *and* layout.

- **Versus other thread parallel programming models (mechanisms)**
  - OpenMP, OpenACC, OpenCL, … have parallel execution
  - OpenMP 4 finally has execution spaces; when memory spaces ??
  - ➢ **All of these neglect data layout mapping**
    - Requiring significant code refactoring to change data access patterns
    - Cannot provide *performance* portability
  - ➢ **All require language and compiler changes for extension**
- **Kokkos extensibility "future proofing" for evolving architectures**
  - Library extensions, not compiler extensions
  - E.g., Intel KNL high bandwidth memory ← just another memory space
- **Productivity versus other programming models?**

Patterns, Policies, and C++11 Lambdas

Sandia National Laboratories

- Pattern composed with policy drives the computational body

```
for ( int i = 0 ; i < N ; ++i ) { /* body */ }
```

pattern — policy — body

```
parallel_for ( N , [=]( int i ) { /* body */ } );
```

C++11 lambda

- C++11 lambda implements computational body
  - C++ compiler creates a *closure* for you: function body + captured data
    - Old school: tedium of writing a C++ class with operator()(int i)
  - Kokkos executes your closure according to pattern and policy
- C++17 lambda within a class member function: [=,*this]
  - Fixed defect in C++11: previously no way to capture *this *by value*
- Data parallel patterns: for, reduce, scan
- Execution policies: range and hierarchical thread team
- Illustrate with the following examples…

Example: Sparse Matrix-Vector Multiply (SPMV)

Sandia National Laboratories

- **Baseline serial version**

```
for ( int i = 0 ; i < nrow ; ++i ) {
  for ( int j = irow[i] ; j < irow[i+1] ; ++j )
    y[i] += A[j] * x[ jcol[j] ];
}
```

- **Simple Kokkos parallel version**

```
parallel_for( nrow , KOKKOS_LAMBDA( int i ) {
  for ( int j = irow[i] ; j < irow[i+1] ; ++j )
    y[i] += A[j] * x[ jcol[j] ];
});
```

- **"nrow" implies a *Range* execution policy**
  - Call body with i = [0..nrow), call in parallel with no ordering guarantees
  - Call body in the *default* execution space
- **`KOKKOS_LAMBDA` for GPU/CUDA portability**
  - CPU : `#define KOKKOS_LAMBDA [=] /* nothing */`
  - GPU : `#define KOKKOS_LAMBDA [=] __host__ __device__`
  - GPU requires CUDA 8 and lambda capture-by-value: `[=]`, `[=,*this]`

Example: Dot-product and Prefix-Sum

Sandia National Laboratories

- **Baseline serial versions, is the pattern obvious?**

```
double result = 0 ;
for ( int i = 0 ; i < N ; ++i ) { result += x[i] * y[i]; }

y[i] = 0 ;
for ( int i = 0 ; i < N ; ++i ) { y[i+i] = y[i] + x[i]; }
```

- **Simple Kokkos parallel versions**

```
parallel_reduce( N, KOKKOS_LAMBDA( int i, double & tmp ){
  tmp += x[i] * y[i] ;
}, result );

y[i] = 0 ;
parallel_scan( N, KOKKOS_LAMBDA( int i, int & tmp, bool final ){
  tmp += x[i];
  if ( final ) y[i+1] = tmp ;
});
```

- **Kokkos manages for you:**
  - Thread local temporary variables
  - Inter-thread synchronizations and reductions of thread local temporaries

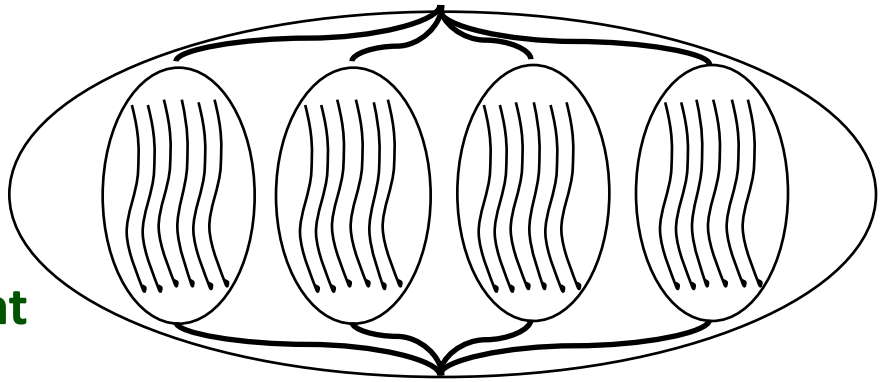
Example: Sparse Matrix-Vector Multiply (SPMV)

Sandia National Laboratories

- **Explicit Range execution policy version**

```
parallel_for( RangePolicy<Space>(0,nrow), KOKKOS_LAMBDA(int i){
  for ( int j = irow[i] ; j < irow[i+1] ; ++j )
    y[i] += A[j] * x[ jcol[j] ];
});
```

- **Is [0 .. nrow) enough parallelism?**
  - With O(1000)s GPU threads? That nested loop could also be parallel …
- **Hierarchical Thread Team execution policy**
  - `TeamPolicy<Space>(LeagueSize,TeamSize)`
  - OpenMP : league of teams of threads
  - CUDA : grid of blocks of threads
  - Threads within a team are concurrent
  - Teams within a league are not concurrent



Example: Sparse Matrix-Vector Multiply (SPMV)

Sandia National Laboratories

```
parallel_for( TeamPolicy<Space>(nrow,AUTO),
  KOKKOS_LAMBDA( TeamPolicy<Space>::member_type member ) {
    const int i = member.league_rank();
    double result = 0 ;
    parallel_reduce(
      TeamThreadRange(member,irow[i],irow[i+1]),
      [&]( int j , double & tmp) { tmp += A[j] * x[jcol[j]];},
      result );
    if ( member.team_rank() == 0 ) y[i] = result ;
});
```

- **Outer level of parallel pattern + execution policy**
  - TeamPolicy requires closure (lambda) with 'member_type' argument
    - *member* is a handle for *thread* within a *team* within a *league*
  - Requires KOKKOS_LAMBDA macro (CPU➔GPU)
- **Inner level of parallel pattern + execution policy**
  - TeamThreadRange identifies *member* threads that participate
  - Ordinary C++11 lambda may be used (without KOKKOS_LAMBDA)

Data Placement and Layout: Views

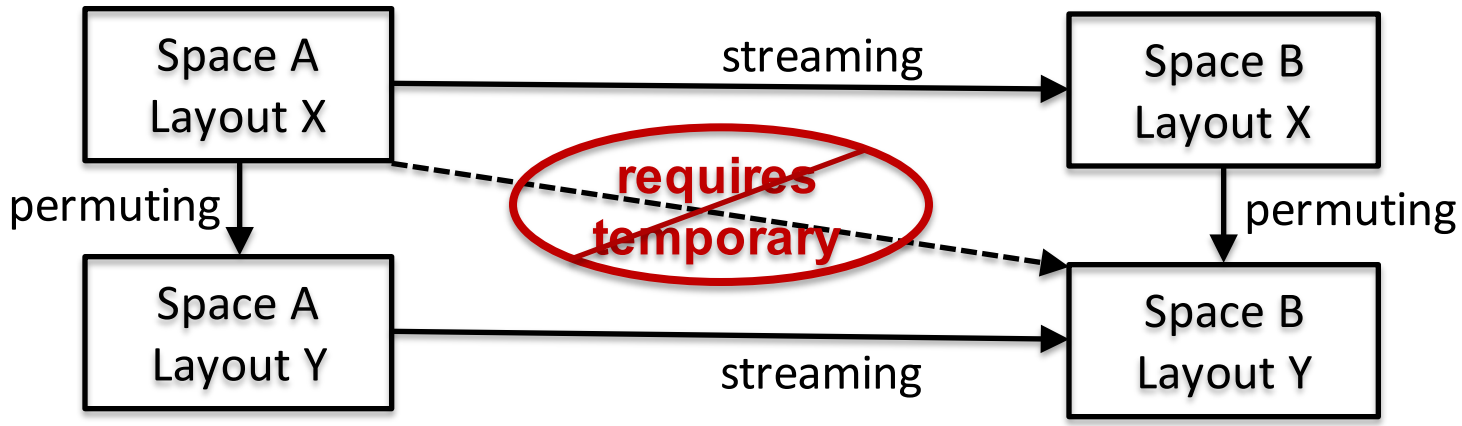
Sandia National Laboratories

- **View< double**[3][8] , $Space_{opt}$ > a("a",N,M);**
  - Allocate array data in a memory Space with dimensions [N][M][3][8]
  - *View* semantics analogous to C++11 std::shared_ptr
- **a(i,j,k,l) : User's access to array datum**
  - Multi-index mapping according to layout
  - "Space" accessibility enforced; e.g., GPU code cannot access CPU memory
  - Optional array bounds checking of indices for debugging
- **View< ArrayType , $Layout_{opt}$, $Space_{opt}$, $Attributes_{opt}$ >**
  - Explicitly declare array *layout* instead of letting Kokkos choose
  - Access intent *attributes*; e.g., atomic, random access (GPU texture cache)
- **Array subview of array view**
  - b = subview(a , *{10,100} , {200,300} , 2 , 3*); // ranges and indices
  - View of same data, with the appropriate layout and multi-index map
- ***True* multidimensional array functionality on-track for C++20**

View's Shared Ownership Semantics

Sandia National Laboratories

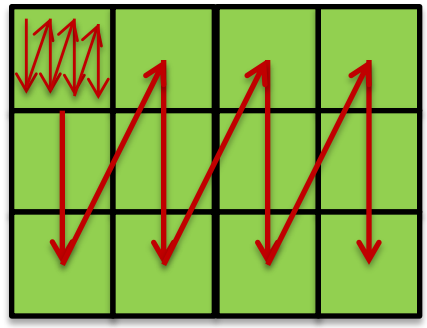
- **View< double**[3][8] , Space$_{opt}$ > b = a ;**
  - *Shallow* copy: 'a' and 'b' are *pointers* to the same allocated array
  - Analogous to C++11 std::shared_ptr
  - Last view of an allocated array deletes that array – *reference counting*
  - Subviews share ownership (reference count) with original allocation
- **deep_copy(destination_view , source_view)**
  - Copy *contents* of allocated array; all or subview portion
  - Element-by-element copy when incompatible layout; e.g., transposing
  - Efficient streaming copy when compatible layout, across memory spaces
  - Kokkos policy: never hide an expensive deep copy operation



Polymorphic Multidimensional Array Layout

Sandia National Laboratories

- **Layout mapping : a(i,j,k,l) → memory location**
  - **Layout is polymorphic, defined at compile time**
  - **Kokkos chooses default array layout appropriate for "Space"**
  - **E.g., row-major, column-major, Morton ordering, dimension padding, ...**
- **User can specify Layout : View< ArrayType, *Layout*, Space >**
  - **Override Kokkos' default choice for layout**
  - **Why? For compatibility with legacy code, algorithmic performance tuning, ...**
- **Customizable Layout : Example Tiling**
  - **View<double**,Tile<8,8>,Space> m("matrix",N,N);**
  - **Tiling layout transparent to user code : m(i,j) unchanged**
  - **Layout-aware algorithm extracts tile subview**



Managing Memory Access Pattern:

Compose Parallel Execution ○ Array Layout

- **Map Parallel Execution**
  - Maps calls to function(iw) onto threads
  - GPU: iw = threadIdx + blockDim * blockIds
  - CPU: iw ∈[begin,end)$_{Th}$; contiguous partitions among threads
- **Choose Multidimensional Array Layout**
  - Leading dimension is parallel work dimension
    - Leading multi-index is 'iw' : a(iw , j, k, l)
  - Choose appropriate array layout for space's architecture
    - E.g., AoS for CPU and SoA for GPU
- **Fine-tune Array Layout**
  - E.g., padding dimensions for cache line alignment

Thread Safety and Atomic Operations

Sandia National Laboratories

- Some algorithms have inherent thread safety challenges
  - Histogram summing into buckets
  - Finite element assembly of linear system coefficients
  - ➢ Scatter-add pattern : `A[ index[i] ] += f( x[i] , y[i] , … );`
- Strategies for thread safety
  - *Coloring* (partitioning) of work into disjoint subsets avoids conflicts
    - Serial execution across subsets, parallel execution within a subset
    - Performance concerns: reduced parallelism and coloring algorithm overhead
  - *Atomic* operations serializes conflicts
    - Special hardware for "+=" of numeric types, perhaps reduced performance
    - Simpler to use than coloring, no loss of parallelism
- Atomics, C++11, and Kokkos
  - C++11 has "hard wired" atomic types with atomic operations
  - Kokkos provides atomic operations on ordinary types
  - C++20 atomic operations for non-atomic types is "in the works"

Other Features (new or in-development)

Sandia National Laboratories

- **Back-ends for new & changing node architectures**
  - AMD Fusion with new open source HCC compiler
  - Intel KNL heterogeneous memory (high bandwidth memory)
  - NVIDIA GPU register shuffle for intra- thread team collectives
- **Patterns, policies, spaces, layout**
  - Dynamic scheduling (work stealing) execution policies
  - Multidimensional range policies (parallel “loop collapse”)
  - Dynamically resizable arrays - thread-scalable within parallel operations
  - Directed acyclic graph (DAG) of “fine grain” tasks execution pattern/policy
  - Tiling layout mapping
- **Portable embedded performance instrumentation**
  - Selective instrumentation of individual parallel dispatch

Application Developer Productivity
Case Study: FY15 ASC Co-design L2 Milestone

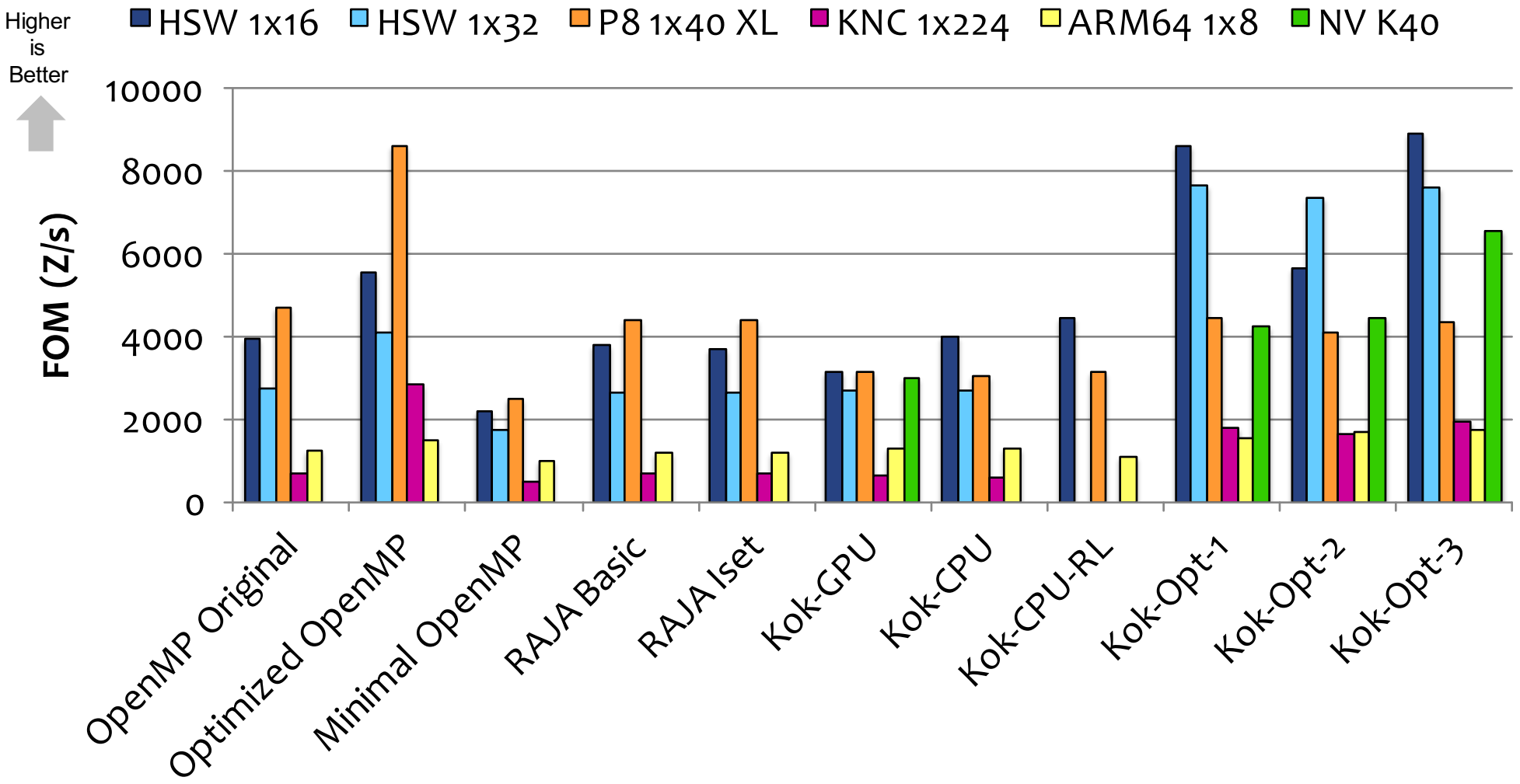
Sandia National Laboratories

- C. Trott, S. Hammond, C. Vaughan, D. Dinge, P. Lin, J. Cook, D. Pase
- **Port LLNL's LULESH mini-application to Kokkos,** part of milestone
  - Starting point is serial version of LULESH
  - Comparison of performance with other programming models
  - Comparison of programmer effort with other programming models
  - Programming models: OpenMP, Kokkos, and LLNL's RAJA
  - Several porting & optimization phases: minimal to extensive
- **LULESH-on-Kokkos porting optimization phases**
  - CPU-RL : basic port using lambdas
  - GPU : make class member functions 'const' to call from within GPU kernel
  - Opt-v1 : eliminate buffer reallocation and reduce register pressure
  - Opt-v2 : use Kokkos Views and TeamPolicy
  - Opt-v3 : fuse computational kernels

Performance Portability Metrics

Sandia National Laboratories

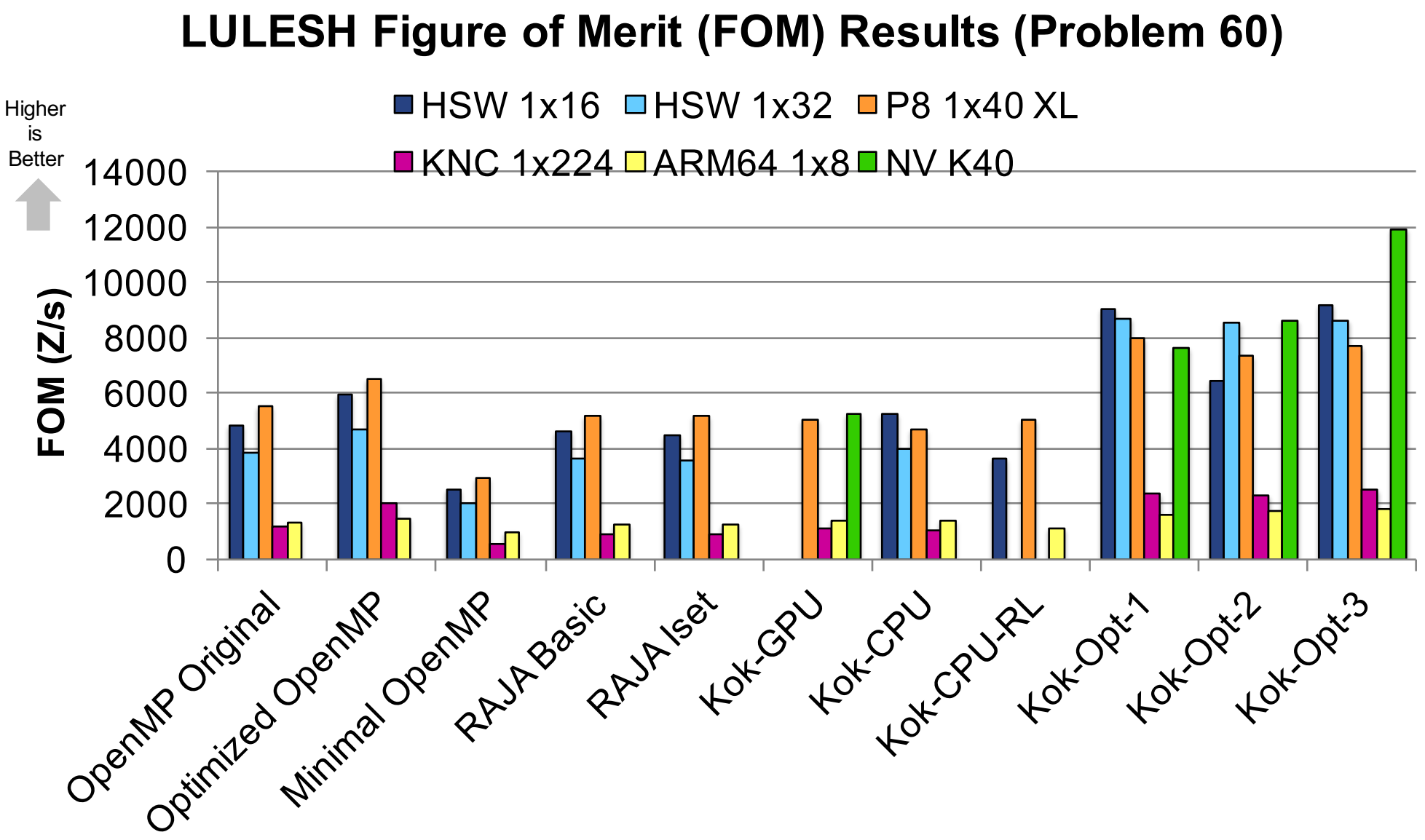
LULESH Figure of Merit (FOM) Results (Problem 45)



Results by Dennis Dinge, Christian Trott and Si Hammond

Performance Portability Metrics

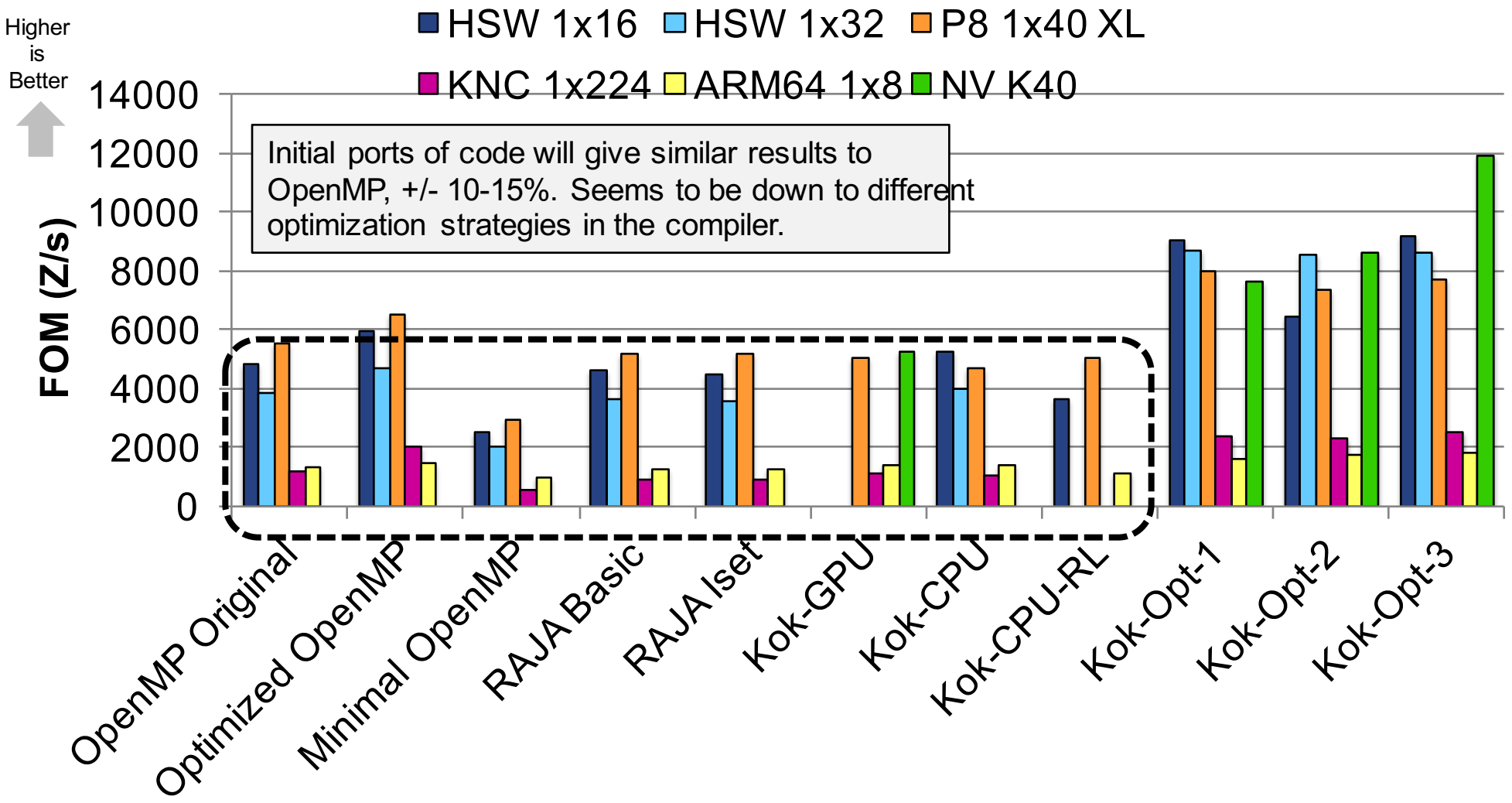
Sandia National Laboratories



Performance Portability Metrics

Sandia National Laboratories

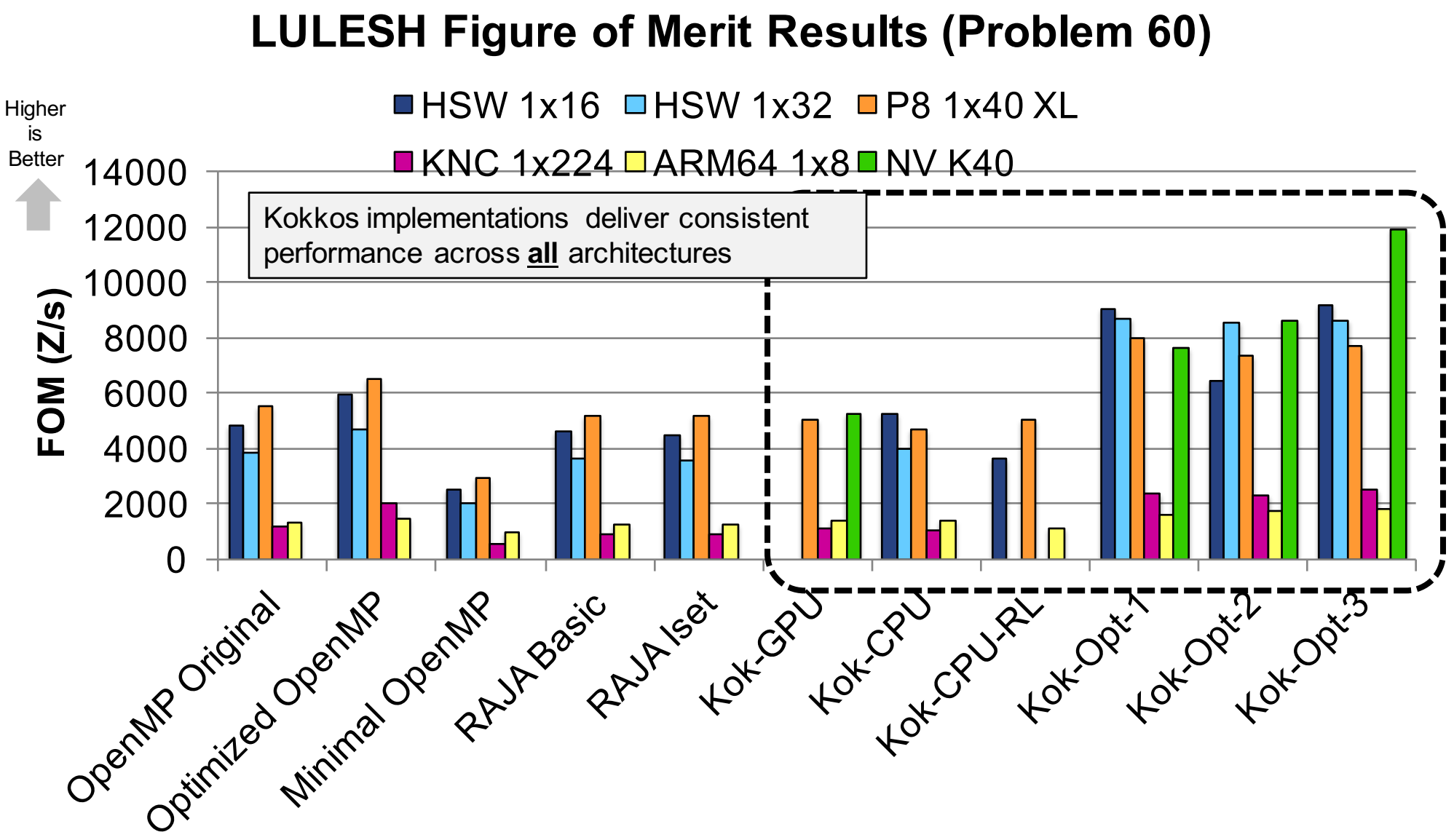
LULESH Figure of Merit (FOM) Results (Problem 60)



Results by Dennis Dinge, Christian Trott and Si Hammond

Performance Portability Metrics

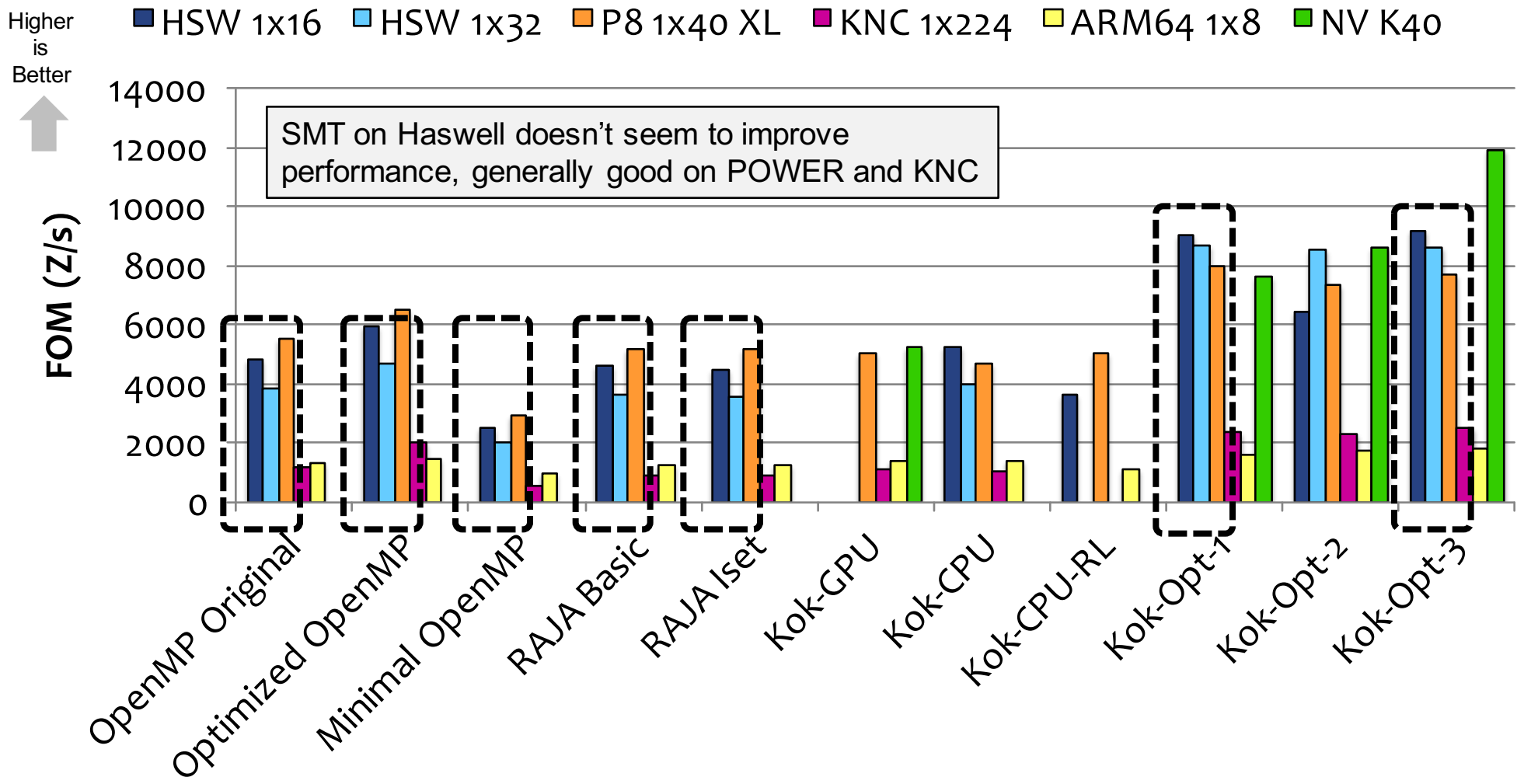
Sandia National Laboratories



Performance Portability Metrics

Sandia National Laboratories

LULESH Figure of Merit Results (Problem 60)



How do we calculate “productivity”?

Sandia National Laboratories

- With great difficulty – lots of discussion in the community about what this *really* means

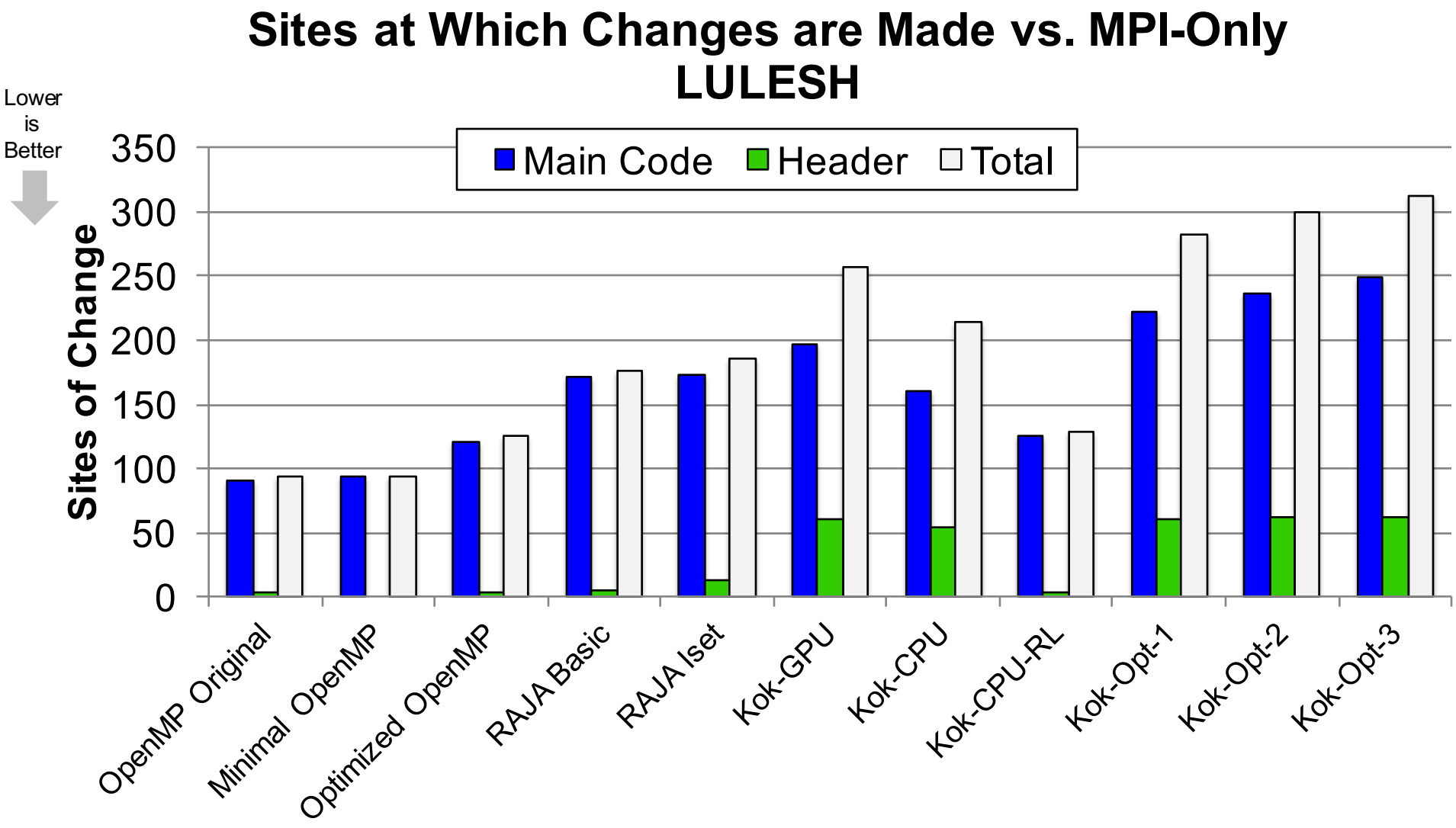
- Our approach:
  1. Remove all comments from the code
  2. Utilize the clang-format LLVM tool with “Google” code option
  3. Compare the number of sites using Apple’s FileMerge tool
  4. Compare the lines added/removed using diff –b –w <paths>

- Not perfect and we have hand modified code of *all* versions to bring the counts more into line (and to be fair wherever possible)
- Point is to show approximate level of programmer effort not be precisely quantitative because coding style largely down to individual

http://clang.llvm.org/docs/ClangFormat.html

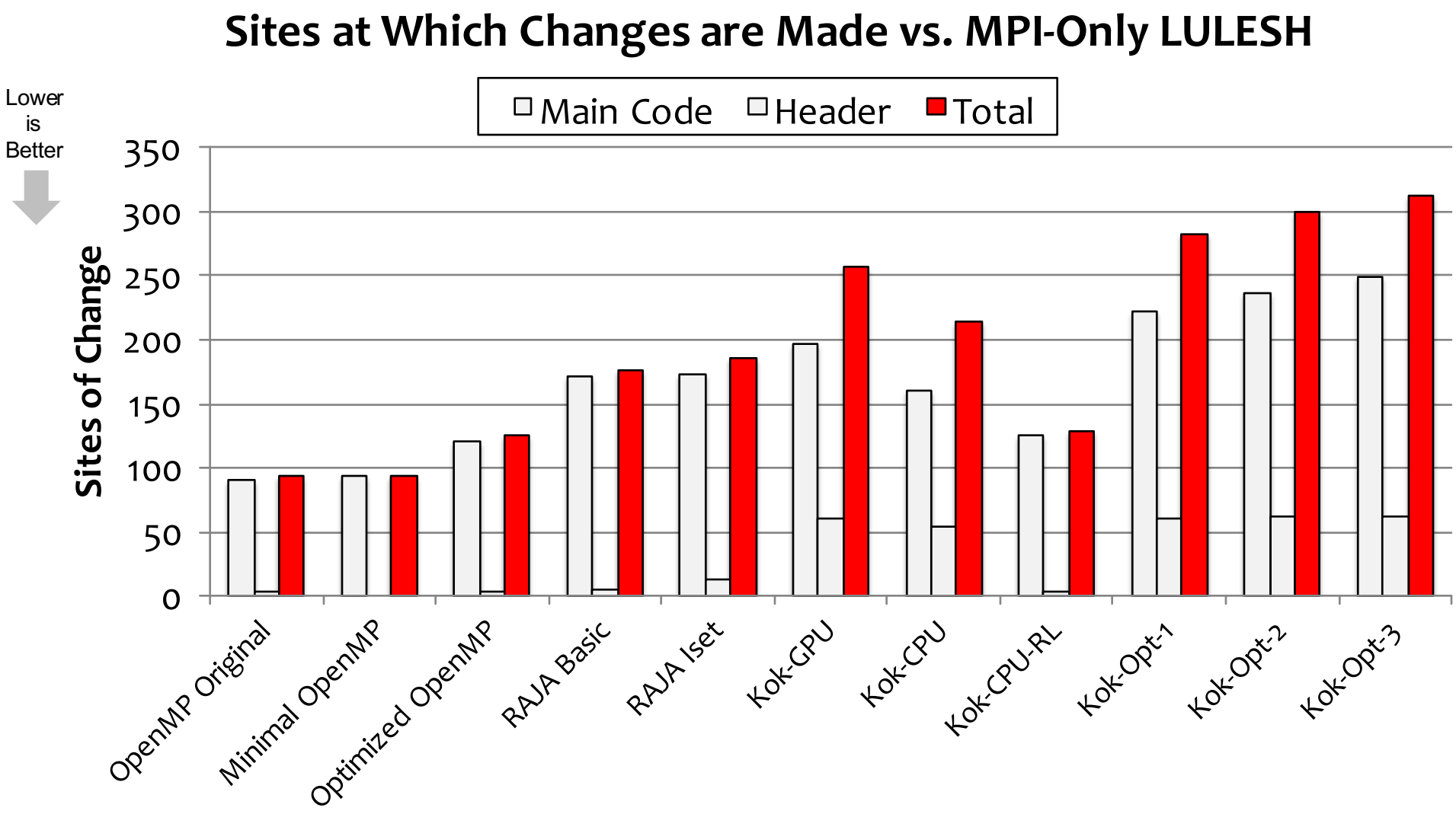
Code Sites at Which Changes are Made

Sandia National Laboratories



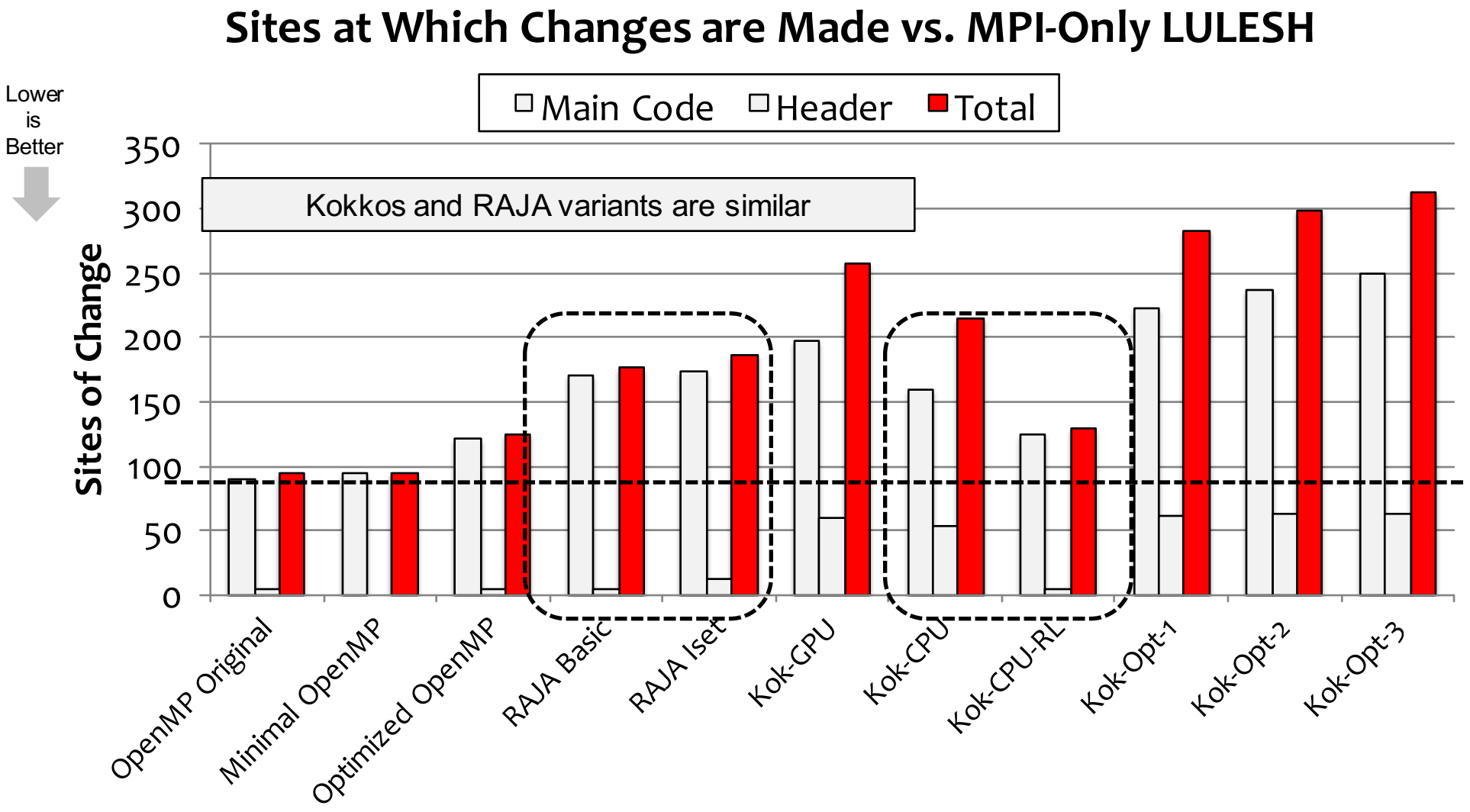
Code Sites at Which Changes are Made

Sandia National Laboratories



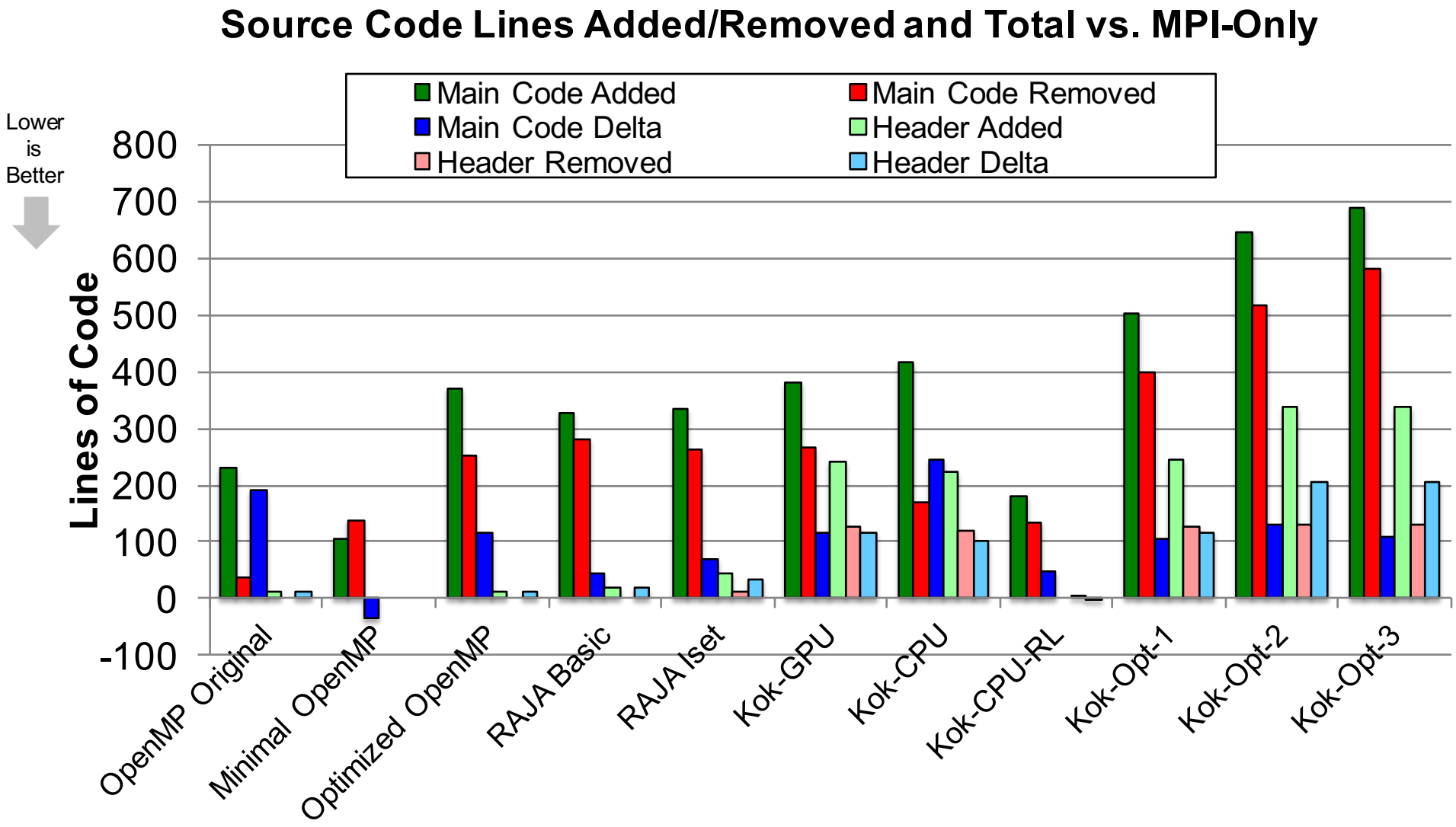
Code Sites at Which Changes are Made

Sandia National Laboratories



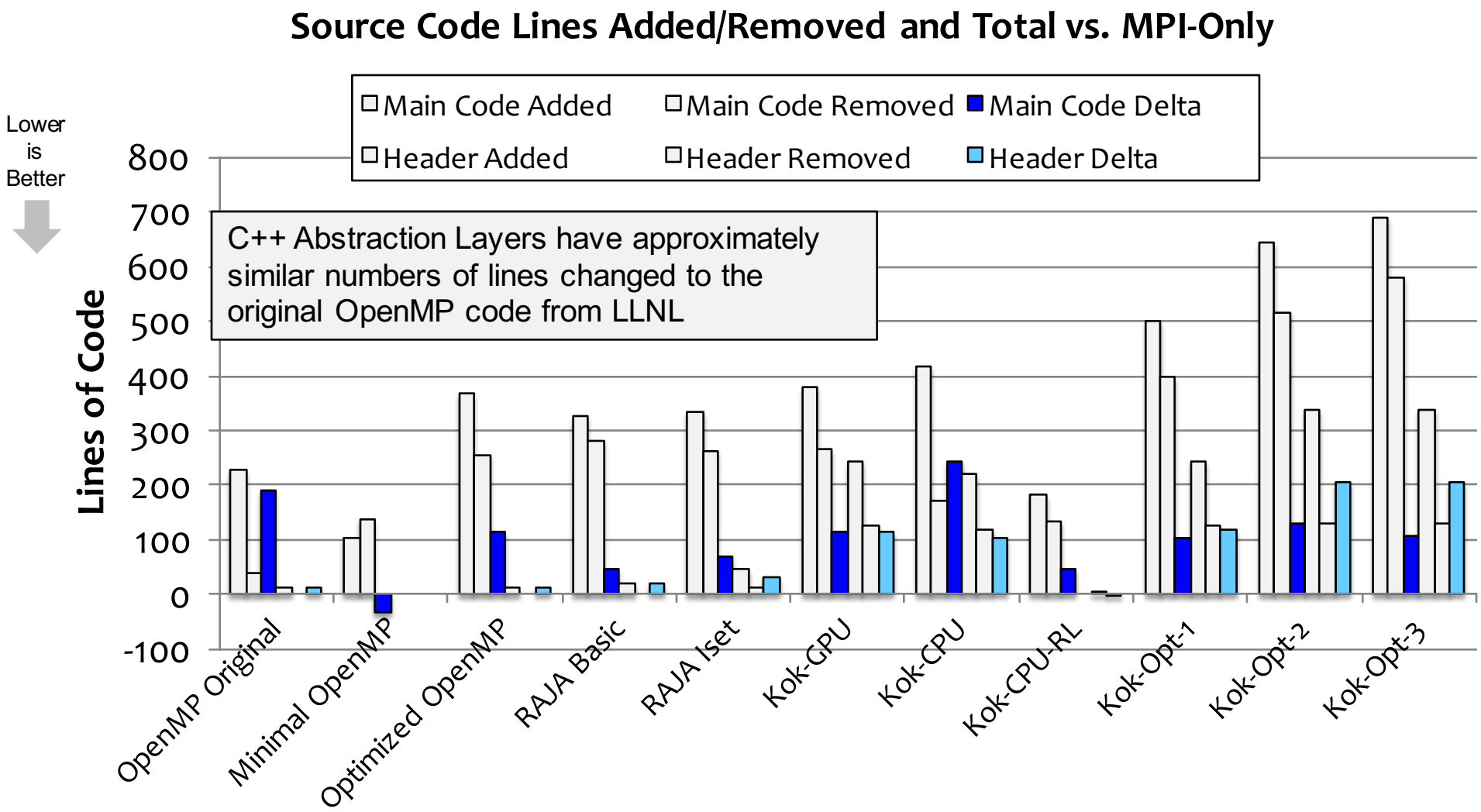
Source Code Line Changes

Sandia National Laboratories



Source Code Line Changes

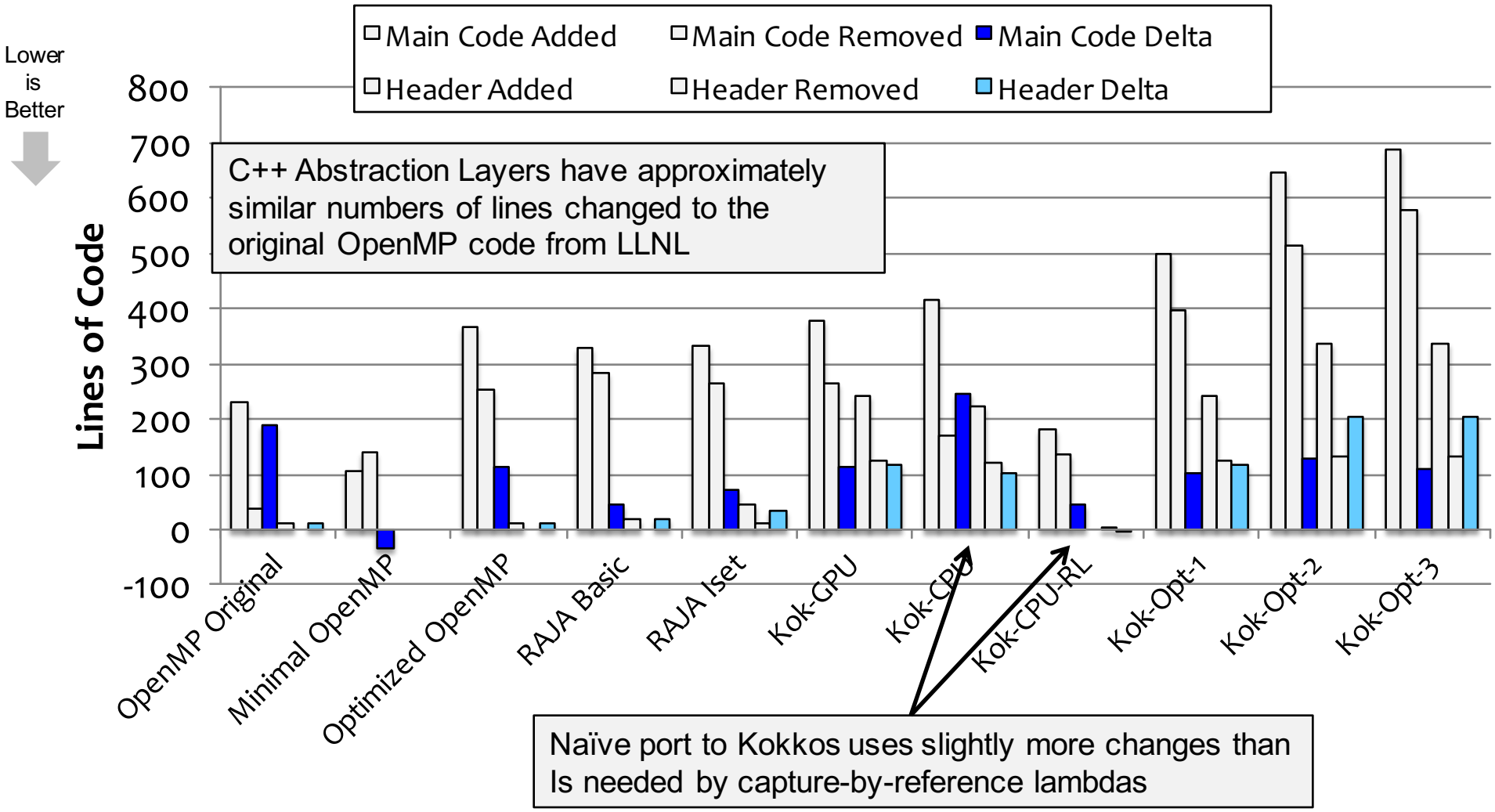
Sandia National Laboratories



Source Code Line Changes

Sandia National Laboratories

Source Code Lines Added/Removed and Total vs. MPI-Only



Programmer Development Time

Sandia National Laboratories

- Initial Kokkos-CPU port by new-to-code developer took a few months
  - No threading/OpenMP/Kokkos experience for code development
  - Lots of correctness and performance issues came up
  - Initial experience with programmer tools and profilers
- Kokkos optimized implementations
  - O(few weeks) of "Kokkos-expert" time
- OpenMP initial and optimized implementations
  - O(few days - week) of "OpenMP-expert" time written on a plane
- These are not significant amounts of FTE but the code is small in comparison to production settings (but code groups are larger and better resourced)
- Difficult (impossible?) to do a deep quantitative comparison

Conclusion / Takeaways

Sandia National Laboratories

- **Performance Portability, for C++ Applications**
  - Integrated mapping of applications' computations *and* data
    - Other programming models fail to map data and limit performance portability
  - Future proofing via designed-in extensibility and ongoing R&D
  - Production on Multicore CPU, Intel Xeon Phi, IBM Power 8, and NVIDIA GPU; *AMD Fusion in progress*
    - github.com/kokkos/kokkos
- **Application Developer Productivity, for C++ Applications**
  - C++11 lambda for simple conversion of 'for' loops to 'parallel_*pattern*'
  - Reduce and Scan inter-thread complexity managed by Kokkos
  - Hierarchical parallelism using nested patterns can increase parallelism
  - Case Study: no harder than OpenMP, optimization is easier
- **Goal: Future ISO/C++ Standard subsumes Kokkos abstractions**