SANDIA REPORT

SAND2015-7886
Unlimited Release
Printed September, 2015

ASC Trilab L2 Codesign Milestone 2015 - Sandia

C.R. Trott, S.D. Hammond, D. Dinge, P.T. Lin, C.T. Vaughan, J. Cook, H.C. Edwards, M. Rajan, and R. Hoekstra

Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185

Sandia National Laboratories

SAND2015-7886
Unlimited Release
Printed September, 2015

ASC Trilab L2 Codesign Milestone 2015 - Sandia

C.R. Trott, S.D. Hammond, D. Dinge, P.T. Lin, C.T. Vaughan,
J. Cook, H.C. Edwards, M. Rajan, and R. Hoekstra
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, 87185

For the FY15 ASC L2 Trilab Codesign milestone Sandia National Laboratories performed two main studies. The first study investigated three topics (performance, cross-platform portability and programmer productivity) when using OpenMP directives and the RAJA and Kokkos programming models available from LLNL and SNL respectively. The focus of this first study was the LULESH mini-application developed and maintained by LLNL. In the coming sections of the report the reader will find performance comparisons (and a demonstration of portability) for a variety of mini-application implementations produced during this study with varying levels of optimization. Of note is that the implementations utilized including optimizations across a number of programming models to help ensure claims that Kokkos can provide native-class application performance are valid. The second study performed during FY15 is a performance assessment of the MiniAero mini-application developed by Sandia. This mini-application was developed by the SIERRA Thermal-Fluid team at Sandia for the purposes of learning the Kokkos programming model and so is available in only a single implementation. For this report we studied its performance and scaling on a number of machines with the intent of providing insight into potential performance issues that may be experienced when similar algorithms are deployed on the forthcoming Trinity ASC ATS platform.

Acknowledgment

The results in this report represent significant efforts from many teams at Sandia including the Kokkos development team, the ASC Advanced Architecture Test Bed and management team and several LDRD projects. We are grateful for the support of all the teams at the laboratories.

Access to the LULESH RAJA implementations described in this report was made possible by Jeff Keasler, Rich Hornung. We have greatly benefited from conversations with them regarding C++ abstraction layers and from additional conversations with Ian Karlin regarding the structure of LULESH.

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.

Contents

List of Figures

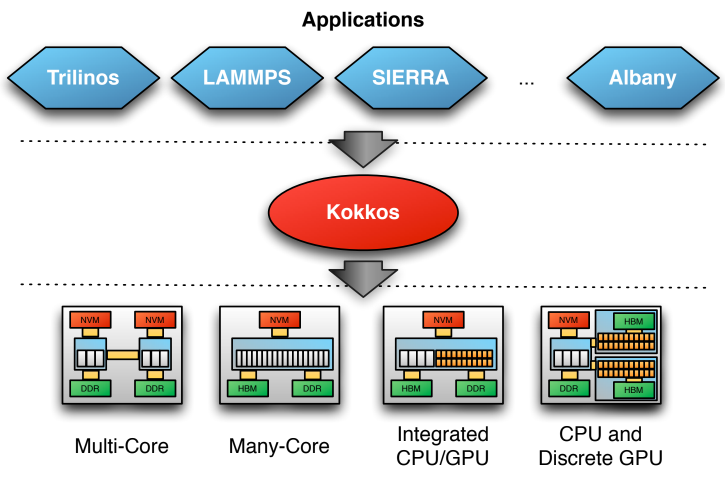
List of Tables

Chapter 1

Analysis of Performance, Portability and Programmer Productivity using Kokkos in the LULESH Benchmark

1.1 Kokkos Programming Model

The Kokkos programming model [5, 6, 4] is designed to act as an abstraction layer to address three important concerns for cross platform application development: (1) where computation is performed; (2), where data is allocated and, finally, (3) how data is accessed. By abstracting these concerns the mapping of each of these to modern high-performance computing architectures can be precisely remapped to provide strong performance. Thus the abstraction layers provide insulation for application developers from the details of rapidly changing hardware architectures.



1.1.1 Abstraction Concepts in the Kokkos Model

Kokkos provides six key abstractions which are designed to transparently manage thread parallel computations and their data access patterns. This capability enables users' thread parallel computations to be both portable and performant across diverse and heterogeneous manycore architectures such as multicore CPUs, the Intel Xeon Phi, and NVIDIA GPUs. The six key abstractions are as follows:

1. Users' express parallel computations with **parallel patterns**; e.g., for-each, reduce, scan, and directed acyclic graph (DAG) of tasks.

2. Parallel computations occur within **execution spaces** of a heterogeneous architecture; e.g., latency-optimized CPU cores and throughput optimized GPU cores.

3. Parallel computations are scheduled according to **execution policies**; e.g., statically scheduled range [0..N) and dynamically scheduled thread teams.

4. Data are allocated within **memory spaces** of a heterogeneous architecture; e.g., in CPU main memory and GPU memory.

5. Data are allocated through multidimensional arrays with **polymorphic layout** that specifies how an arrays multi-index domain space is mapped to an allocation within a memory space.

6. Arrays may be annotated with **access intent traits** such as "random acces" or "atomic access". Kokkos may use these traits to map array entry access to architecture-specific mechanisms such as GPU texture cache or atomic instructions.

The Kokkos library API has been designed and improved through many development iterations. Given an understanding of the Kokkos abstractions listed previously and an intermediate knowledge of C++ (2011 standard), this library API is concise and intuitive for users. Performance portability has been demonstrated across architectures using mini applications including MiniFE, MiniMD, MiniAero, and LULESH.

Kokkos is the main performance portable programing model in a number of projects at Sandia National Laboratories and continues to be a key component of the strategy for next-generation computing platforms. The three projects with the most components written to use Kokkos are the Trilinos solver library, the LAMMPS molecular dynamics simulation code and Albany, an open source multi-physics research code. We are also writing research prototypes for important linear algebra kernels, solvers, graph analytics problems and during the next year, engineering contact problems, finite element code and fluid-dynamics simulations.

In the ASC FY15 Kokkos formally became an independent open source project with development now publicly available on the github source code repository site (github.com/kokkos/kokkos). The project is available under a BSD license.

1.2 Advanced Architecture Test Beds and Benchmarking Resources

1.2.1 Shepard: Intel Haswell Architecture Testbed

Shepard contains the latest HPC variant of Intel's Haswell server class processor. Each socket has 16 dual-threaded SMT cores and 40MB of level-3 cache. The clock rate is 2.3GHz continuing the trend of server processor clocks decreasing over previous generations when new cores are added. Haswell processors introduce the AVX-2 instruction set from Intel which extends the floating point capabilities to include fused multiply-add instructions (effectively doubling peak floating point SIMD throughput), gather/scatter memory operations and vectorized integer operations which may be useful in the generation of address offsets. For Shepard all benchmark runs utilize Intel's 15.2 Composer-XE compiler environment.

1.2.2 Compton: Intel Sandy Bridge and Knights Corner Architecture Testbed

Compton is a well established test-bed at Sandia providing access to Intel's Xeon Phi Knights Corner many-core architecture. The cards feature 57-cores and 6GB of GDDR-5 system memory. Each core is clocked at 1.1GHz and provides a single 8 double-precision operand wide vector unit, delivering greater than 1TFLOP of peak computation performance. For this platform all benchmarks utilize the Intel 15.2 Composer-XE compiler environment.

1.2.3 Shannon: Intel Sandy Bridge and NVIDIA Kepler Testbed

The Shannon test bed provides a mixture of NVIDIA GPU hardware connected to Intel Sandy Bridge processors. GPUs provided including current best-of-class K80 HPC GPUs as well as older K40 and K20 GPUs. All benchmarks run using Shannon utilize the NVIDIA CUDA 7.5 SDK and the GNU 4.9.2 compiler.

1.2.4 White: IBM POWER8 and NVIDIA Kepler Testbed

White is the newest test bed machine provided in Sandia's Advanced Architecture test bed program and provides access to dual-socket, 10-core per socket IBM POWER8 processors with a single K40 GPU attached to each socket via PCIe. Nodes are interconnected with FDR InfiniBand which is also the first deploy of this technology from Mellanox at Sandia. For benchmark runs on this system we utilize the IBM XL 13.1.2 compiler and the GNU 4.9.2 compiler.

1.2.5 Hammer: ARM64 Testbed

Hammer is a 46-node installation of ARM64 processors designed by Applied Micro (APM). These single-socket processors provide 8-cores and fully comply with the ARMv8 ISA specification. Each core runs at 2.4GHz. Four channels of DDR3 memory are connected giving memory bandwidth which is similar to an Intel Sandy Bridge socket. For benchmark runs on these machines we use the GNU 4.9.2 compiler which was the first to provide optimization support for the ARM64 ISA.

1.3 Baseline and Non-Kokkos Implementations of LULESH

1.3.1 Baseline MPI-Only Implementation

The LULESH mini-application version 2.0 [1, 12] is provided by LLNL for codesign and benchmarking activities. The code comes with versions written using MPI, OpenMP, CUDA and OpenACC, the latter are written by NVIDIA. When downloaded, LULESH uses preprocessor directives to compile in OpenMP pragmas and headers as well as calls to the MPI runtime. For our purposes we have developed an MPI-only version of the code, which we may refer to as 'Serial' since it contains no on-node parallelism, by removing all OpenMP statements. The MPI-only version is regarded as our baseline application for developing peer implementations since it more closely represents the code with which many ASC application developers will be starting to parallelize on the Trinity and Sierra platforms.

1.3.2 Baseline OpenMP Implementation

As stated, the baseline versions of the code provided by LLNL include an OpenMP implementation which we call the "OpenMP Original" variant. This code was included in the CORAL procurement and has been run on a wide variety of platforms and compilers.

1.3.3 RAJA Implementations

The development RAJA code provided by LLNL during the L2 includes two variants of the LULESH miniapp. The first termed 'basic' utilizes RAJA for parallel-execution dispatch effectively replacing OpenMP parallel-for and reduction parameters with RAJA equivalents. The second version, which is designed to improve performance on some platforms, uses data structures known as Index Sets. These allow for contiguous ranges of data structures within the application to be represented specifically improving the level of vectorization which may be possible. We term this implementation 'RAJA-IndexSet'.

1.3.4 Minimal OpenMP Implementation

In the minimal OpenMP implementation, developed by Sandia, we utilize the Intel Analyzer and Inspector XE toolkits to add parallelism to the baseline MPI-only version of the code. These tools allow us to profile the hotspots of the miniapp and then, under guidance from the tool, gradually add parallel dispatch to expensive compute loops. Thread data race conditions can then be detected (which are a feature of the serial LULESH algorithm) and corrected. In this implementation we employ the use of atomic operations on the domain data-structure to ensure threads are not able to overwrite each others updates. The purpose of this implementation is to provide insight into a minimal version of the code which may be developed using semi-automated or tool-assisted processes.

1.3.5 Optimized OpenMP Implementation

The optimized OpenMP implementation of LULESH uses experiences from Sandia's Application Performance Team which have been gathered over four years of porting codes to Xeon, Xeon-Phi and POWER processors/co-processors. The changes implemented in the code include:

- **Extensive Use of the `restrict` and `const` Keywords**: which enable the compiler to more accurately reason about variable/memory use. The effect is to provide more efficient instruction generation which often includes higher level of vectorization.
- **Efficient Reductions**: reductions are re-implemented to utilize OpenMP-reduction constructs. Our experience has shown that the compiler generates more efficient code when these are used.
- **Thread Memory Buffer Hoisting**: in the original OpenMP implementation provided by LLNL buffers are repeatedly allocated and then freed. In the optimized variant these are hoisted out of parallel loops and replaced by schemes which keep the buffer live allowing it to grow when needed. The effect is to significantly reduce calls to memory allocation/deallocation routines which are typically synchronization points in threaded sections.
- **Use of OpenMP 4.0 SIMD Pragmas**: we make extensive use of SIMD pragmas where the executing code can be shown to be safe. In our analysis of the baseline LULESH OpenMP implementation from LLNL many compilers are unable to verify that loop dependency is safe, where these can be safely reasoned about and we expect performance improvements we can force vector instruction to be generated.

1.4 Kokkos Implementations of LULESH

1.4.1 Minimal-CPU Kokkos Implementation

The minimal-CPU Kokkos variant is roughly equivalent in its approach to the Minimal OpenMP variant developed by Sandia. It modifies the serial version of LULESH to use Kokkos:: parallel_for and Kokkos::parallel_reduce to parallelize loops. Examples of loop transformation are shown below.

Listing 1.1. Example Original For Loop

```
for (Index_t i = 0; i < numElem; ++i) {
  ...
}
```

Listing 1.2. Example For Loop Rewritten into Kokkos

```
Kokkos::parallel_for ( numElem, [&] (const Index_t& i) {
  ...
});
```

Additionally, atomic operations are employed to resolve write conflicts. In contrast to the recommended way of using Kokkos for new applications we utilize *capture by reference* semantics to create C++11 Lambdas. This allows the data structures to remain unchanged including the use of the std :: vector container.

Listing 1.3. Example Domain Update in Original Code

```
    for (Index_t lnode = 0; lnode < 8; ++lnode) {
      Index_t gnode = elemToNode[lnode];
      domain.fx(gnode) += fx_local[lnode];
      domain.fy(gnode) += fy_local[lnode];
      domain.fz(gnode) += fz_local[lnode];
    }
```

Listing 1.4. Domain Update using Kokkos Atomics Operations

```
    for (Index_t lnode = 0; lnode < 8; ++lnode) {
      Index_t gnode = elemToNode[lnode];
      Kokkos::atomic_add(&domain.fx(gnode), fx_local[lnode]);
      Kokkos::atomic_add(&domain.fy(gnode), fy_local[lnode]);
      Kokkos::atomic_add(&domain.fz(gnode), fz_local[lnode]);
    }
```

Finally, we modified the error handling code from the original serial algorithm in LULESH to use parallel reductions when error cases are identified. This eliminates the need to call MPI process abort functions directly from loops which prevents vectorization and execution on GPUs.

The changes described allow the serial code to now run in thread-parallel execution on CPU architectures like traditional Xeon processors, POWER8 processors and Intel's Xeon Phi many-core accelerator. It represents a low optimization state for the type of code we expect to run on the Trinity Phase II platform.

1.4.2 Minimal-GPU Kokkos Implementation

In order to execute the code on GPUs more changes are required. In particular it is necessary to capture variables by copy, so that a self contained functor can be given to the GPU to execute there. A consequence is that class instances captured by the lambda function must have their member functions marked appropriately with const. In LULESH this entails the Domain class and its data accessor functions. Furthermore those accessor functions must be marked with the KOKKOS_INLINE_FUNCTION macro which allows for the correct mark-up needed to compile code for NVIDIA's CUDA or the Kalmar compiler for AMD Fusion APU devices. An example of this change is:

Listing 1.5. Original Domain Definition Code from LULESH

```
class Domain {
  ...
  Real_t &x(const Index_t idx) { return m_x[idx]; }
};
```

Listing 1.6. Modified Domain Definition allowing execution on the GPU

```
class Domain {
  ...
  KOKKOS_INLINE_FUNCTION Real_t &x(const Index_t idx) const { return m_x[idx]; }
};
```

Another consequence of capturing by copy is that std::vector containers can no longer be used. The effect of continuing std::vector usage while capturing by copy would be that at each kernel launch all data allocations referenced inside the loop body would be copied into new allocations. Further, std::vector is not supported by the NVIDIA CUDA or the AMD Kalmar Fusion APU backends. In order to address this difficulty, Kokkos provides a more or less "drop-in" replacement for std::vector containers by using the Kokkos::vector class. It has a similar API to std::vector but utilizes Kokkos View semantics to ensure assignments are shallow copies.

Listing 1.7. Original Domain Definition in LULESH

```
class Domain {
  ...
  std::vector<Real_t> m_x;
};
```

Listing 1.8. Kokkos Domain Definition using Kokkos Vector Replacements

```
class Domain {
  ...
  Kokkos::vector<Real_t> m_x;
};
```

We also replace raw memory allocations with Kokkos equivalents to ensure memory allocations can be correctly mapped to GPU backends.

1.4.3 Kokkos Optimization Level 1

The first set of optimization addresses excessive data reallocation. LULESH uses large temporary allocations which are allocated and then deallocated for subsets of kernels. Instead of allocating and reallocating the actual memory, we added a large buffer allocation and a function which creates views into those buffer allocations.

Listing 1.9. Original LULESH Element Allocation

```
//replace
Real_t *fx_elem = Allocate<Real_t>(numElem8);
Real_t *fy_elem = Allocate<Real_t>(numElem8);
Real_t *fz_elem = Allocate<Real_t>(numElem8);
...
Release(&fz_elem);
Release(&fy_elem);
Release(&fx_elem);
```

Listing 1.10. Modified LULESH Element Allocation

```
ResizeBuffer((numElem8*sizeof(Real_t)+4096)*3);
Real_t *fx_elem = AllocateFromBuffer<Real_t>(numElem8);
Real_t *fy_elem = AllocateFromBuffer<Real_t>(numElem8);
Real_t *fz_elem = AllocateFromBuffer<Real_t>(numElem8);
```

Next we utilized NVIDIA's visual profiler to determine the biggest bottlenecks in GPU performance, since the relative performance of the CPU and GPU execution was poor compared to the relative hardware capabilities. The biggest issue highlighted by the profiler is the low occupancy achieved by a number of kernels. Much of the register pressure (i.e. the size of the state of a given thread) comes from temporary static sized arrays. The most expensive kernel calls is a sub function which calculates values for three static sized arrays corresponding to three spacial dimensions. Since the calculations are independent one can break that down into three successive calculations for a single dimension, and thus reuse the temporary static sized arrays.

Listing 1.11. Original Hourglass Calculation Sequence in LULESH

```
...
xd1[..] = ..;
yd1[..] = ..;
...
CalcElemFBHourglassForce(xd1, yd1, zd1, hourgam, coefficient,
                         hgfx, hgfy, hgfz);
```

Listing 1.12. Modified Hourglass Calculation Sequence in LULESH

```
...
xd1[..] = ..;
...
CalcElemFBHourglassForce(xd1, hourgam, coefficient, hgfx);
...
yd1[..] = ..;
...
CalcElemFBHourglassForce(yd1, hourgam, coefficient, hgfy);
...
```

This represents a commonly useful technique, that in cases where register pressure is high, it is better to isolate the calculations than interleaving them, even though the latter can provide more instruction parallelism.

1.4.4 Kokkos Optimization Level 2

The second level of optimization addresses data layout and access properties, as well as introduces hierarchical parallelism where appropriate. A defining feature of Kokkos are memory layouts for multi-dimensional views. In LULESH a number of linearized 2D allocations are used, which in the previous variants were simple pointers with hard coded indexing. In this version those pointers are replaced by actual 2D views for which in most cases the default layout for each architecture is the correct one. Furthermore there are a number of data streams for which gather operations are performed. Replacing those with Kokkos views using the Kokkos::RandomAccess memory trait can improve performance on GPUs.

Listing 1.13. Kokkos Random Access Memory Traits

```
Kokkos::View<const Real_t*, Kokkos::MemoryTraits<Kokkos::RandomAccess> > m_c_x ;
...
KOKKOS_INLINE_FUNCTION Real_t c_x(const Index_t idx) const { return m_c_x[idx]; }
```

Some of the kernels expose a natural hierarchical parallelism, *i.e.* they have nested loops suited for parallelization. Using Kokkos nested parallel calls can help improve performance on GPUs by exposing more parallelism and thus increasing the occupancy. In particular a technique was employed where the inner most loop is mapped to a ThreadVectorRange policy and the outer loop is split between a team-level parallel loop and a thread level parallel loop. Effectively we create artificial work sets for teams. This approach can help with cache locality by making threads in the same team collaborate on cache usage vs. competing for scarce resources. In particular kernels which show a gather behavior with reuse of data, tend to benefit from using hiearchical parallelism.

Listing 1.14. Original Hourglass Kernel-B Calculation in Kokkos

```
Kokkos::parallel_for("CalcFBHourglass_B", numNode, KOKKOS_LAMBDA(const int gnode) {
  Real_t fx_tmp = Real_t(0.0);
  for (Index_t i = 0; i < count; ++i)
    fx_tmp += fx_elem[cornerList[i]];
  domain.fx(gnode) += fx_tmp;
```

Listing 1.15. Modified Hourglass Kernel-B Calculation using Hierarchical Parallelism

```
Kokkos::parallel_for ("CalcFBHourglass_B",
                        Kokkos::TeamPolicy<>((numNode+127)/128,team_size,2),
  KOKKOS_LAMBDA (const typename Kokkos::TeamPolicy<>::member_type& team) {
  const Index_t gnode_begin = team.league_rank()*128;
  const Index_t gnode_end =
      (gnode_begin + 128<numNode)?gnode_begin + 128:numNode;
  Kokkos::parallel_for(Kokkos::TeamThreadRange(team,gnode_begin,gnode_end),
    [&] (const Index_t& gnode) {
    reduce_double3 f_tmp;
    Kokkos::parallel_reduce(Kokkos::ThreadVectorRange(team,count),
      [&](const Index_t& i,double3& tmp) {
      tmp.x += fx_elem[cornerList[i]] ;
    },f_tmp);
    Kokkos::single(Kokkos::PerThread(team), [&](){ domain.fx(gnode) += f_tmp.x ;});
  });
```

1.4.5 Kokkos Optimization Level 3

The LULESH mini-application consists of many successive loops which go over the same range and are trivially data parallel (i.e. there is no reduction, scatter or gather operation. Those loops can be merged which results in better cache reuse and reduced memory traffic. For parallel execution it also better amortizes the start up costs of parallel regions.

We merged kernels in two functions: EvalEOSForElems and CalcEnergyForElems resulting in another significant performance improvement on GPUs.

1.5 LULESH on AMD Fusion APUs

For the last months Sandia National Laboratories has been collaborating with AMD on several topics. These include: (1) an AMD Fusion APU backend for Kokkos and, (2), deployment of the HSA runtime on Sandia's Cooper Fusion Architecture test bed. The Kokkos backend utilizes AMD's Kalmar compiler which is currently under active development. The Kalmar compiler provides a parallel programming environment based on Microsoft's C++AMP. While the latter is not yet completely finished (a number of features are not implemented yet, with an expected completion time around the SC15 conference), all capabilities necessary to run the Minimal-GPU variant and the optimization level 1 variant of LULESH are available. That experiment was successfully performed on an AMD test platform with Kaveri APUs. We do not report performance results here, since the software stack is still far away from a release and the hardware is a consumer grade product with little double precision compute capabilities. Hence the performance is currently not representative of what we believe HPC variants of AMD's APUs will deliver in the future.

1.6 Code Development and ATS Platforms

The porting of full ASC applications to future ATS platforms will no doubt be a time consuming process, not least because the algorithms used in many existing production settings are well optimized for serial environments. We note that our research activities using the Mantevo mini-app suite have shown that data races between threads are difficult to debug and optimize as code porting activities take place.

In Figure 1.1 we show a possible mapping of our various implementations to a timeline for future ATS platforms. Our conjecture is that initial ports of applications to machines will likely not be well optimized but will be performed to get code onto the platform and utilize the available hardware. Performance will likely not be a first priority in these ports. Over time, as programmer expertise grows and the associated debugging and profiling infrastructure is improved, levels of performance will increase. We separate out the continued use of directives from C++ abstractions as the path for Trilinos and several important code bases at Sandia is slowly converging to the use of Kokkos throughout the software stack. Our inclusion of RAJA is to show how we believe these ports of LULESH will compare noting that the optimized GPU variant has not yet been developed by LLNL. The eventual aim of the Kokkos project is to see widespread adoption of many of the core concepts introduced in the C++20 Language Specification and beyond but we recognized that continued development to support specializations for HPC (for instance the inclusion of checkpoints or other reliability infrastructure) may be required in the future (see the dashed outlines for possible future version of the Kokkos programming model).

Our approach takes three main phases: (1) the initial porting of parallel execution dispatch to future hardware, we term these ports "minimal" in that they achieve only the basics neccessary to execute on Trinity Phase-II and Sierra; (2) the second, longer, phase from 2016 to 2018 requires modification of the application data structures. This is likely to be a significant undertaking requiring considerable development but will yield greater performance on GPU-based systems such as Sierra. Finally, (3), from 2018 onwards,

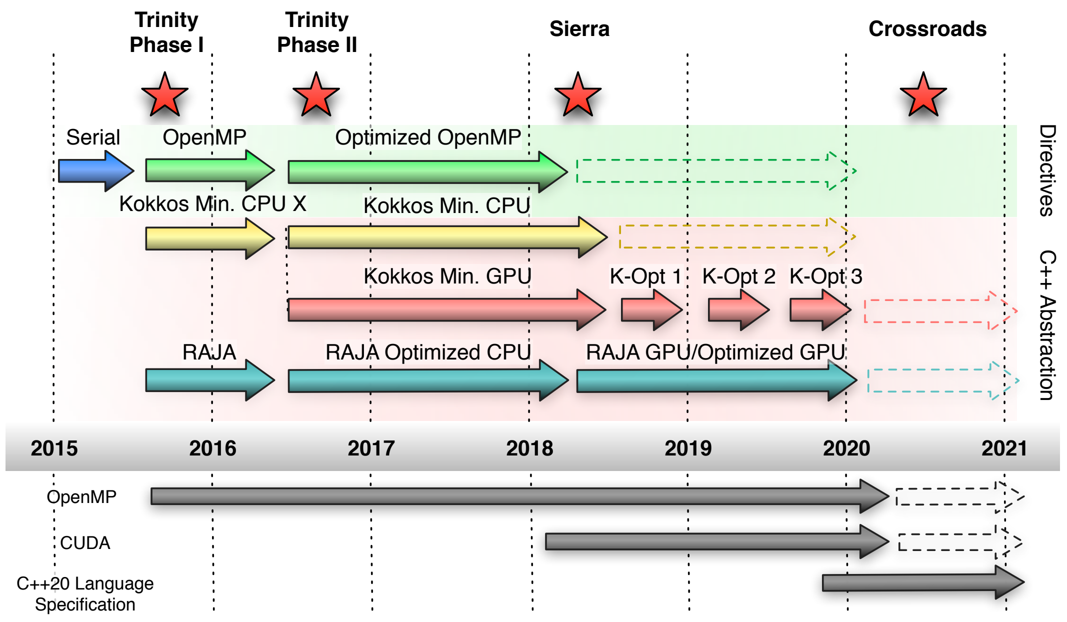


Figure 1.1. Code Development and Benchmark Versions mapped to ATS Platforms

application developers will slowly begin to optimize the applications that have been ported to next generation architectures delivering much greater levels of performance.

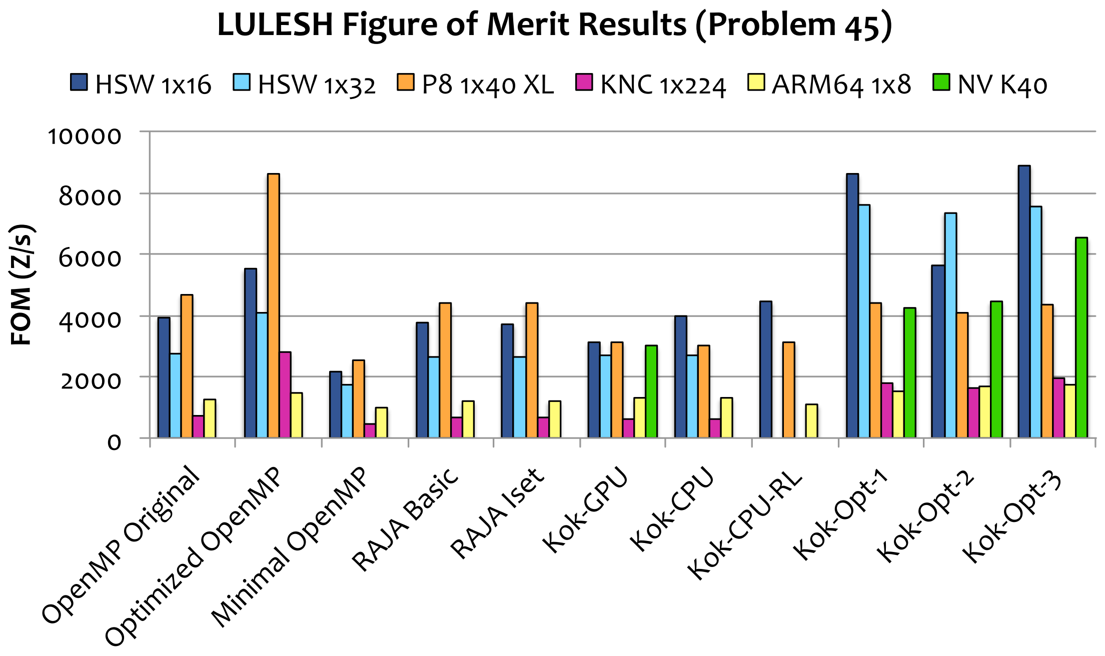
In the latter sections of this document, we compare the performance and associated changes in application code (which we term "programmer productivity") for the implementation variants show in Figure 1.1. This provides a possible overview of the tradeoffs which our code groups may make in deciding how quickly to port and optimize codes in the future. Put differently, what levels of performance we might be able to expect from our codes at what point in time and for what level of programmer effort.

1.7 Performance Portability for LULESH Implementations

1.7.1 Environment Configuration

For each of the runtimes gathered we utilized our experiences from Sandia's ASC architecture testbed benchmarking to configure each run for optimized MPI process placement and thread affinity. Unless otherwise stated we force nodes to be partitioned evenly by MPI ranks and request thread affinities be applied that match each partition. On all platforms except the POWER8 we use default OpenMP runtime settings which typically, in our experience, provide the highest performance. For the POWER8 systems we explicitly tell the OpenMP runtime that binding must be applied as this is not the default when using the IBM-XL runtime.

The MPI and compiler environments deployed on the test bed are custom build by Sandia's research team using the latest versions of NUMA libraries, the Hardware-Locality library [2] and OpenMPI [9]. Over time we have shown that these configurations provide high performance for the systems being used in this study.



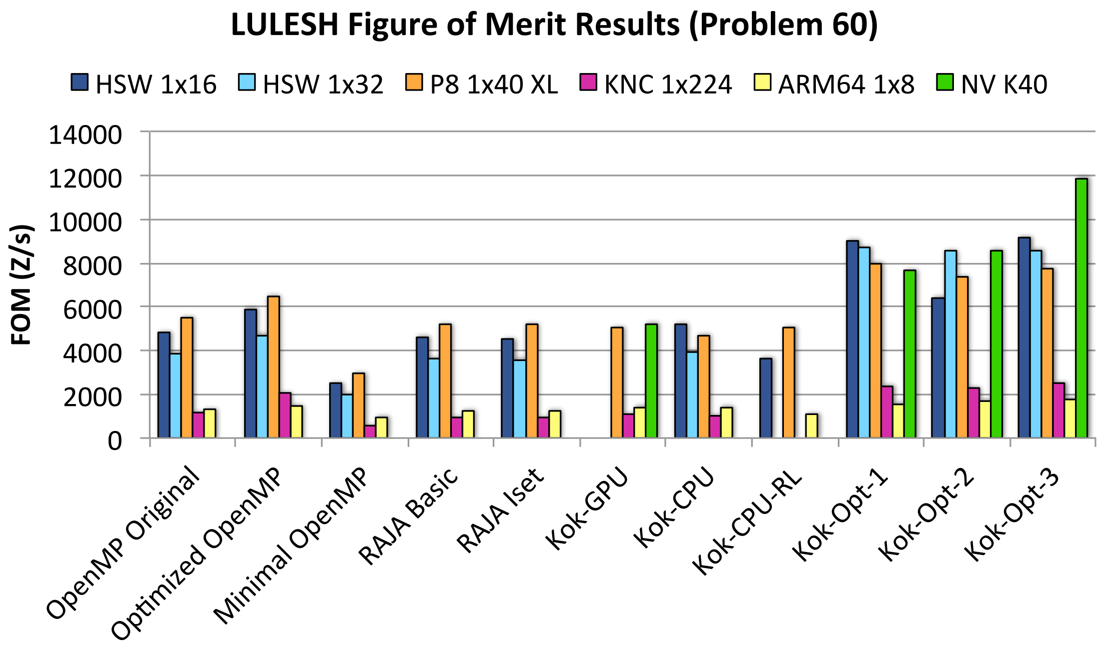
1.7.2 Performance Comparison

For the performance comparison we report the harmonic mean of LLNL-coded Figure of Merit (FOM) from a minimum of 10 runs. Variations are recorded and show typically differences of 1-3% between runs on the same machine. The simulations are run to completion, that means no iteration count limit was given. We ran three different sizes (45, 60, 90) based on guidance from LLNL Lulesh developers. The associated memory footprint is 100 MB, 220 MB and 700 MB respectively.

The performance data shows relatively consistent performance across all ”basic” implementations. The only outlier is the ”Minimal OpenMP” variant, which performs at about 50% of the other variants. The optimized Kokkos variants consistently outperform the basic variants by 60-90%. It is noteworthy that already at the first optimization level most of the performance benefit for CPU like architectures is achieved, while GPU performance significantly increases with the using more aggressive optimization.

1.8 LULESH Binary Sizes

In Figure 1.2 we show the binary sizes associated with the Kokkos, RAJA and pure OpenMP programming models. We breakdown the binary size associated with each model by the number of instructions generated during compile and the binary space allocated for initialized and uninitialized static data structures. We note that pure OpenMP binaries are the smallest in terms of both instructions and associated data allocations. Kokkos binaries have additional instructions and data allocations required for the processing of Kokkos View structures and runtime handling. Both initialized and uninitialized data structures, which make up a significant proportion of the final binary size in each implementation, are roughly static across variants representing what we believe to be a static overhead. We attribute some of the increase in instruction bytes to the routines associated with the Kokkos runtime however, the additional use of templates and different optimizations to the binary are very likely to factor into this observation also.



1.9 LULESH Compile Time

The heavy of C++ abstraction layers creates the potential for increased compile times due to additional parsing, processing and optimization steps required to generate a binary. Figure 1.3 shows the compile times using the Intel 15.2.164 compiler on the Sandia Shepard platform. These times utilize the platform specific options to the compiler supplied by the RAJA and Kokkos build system. In assessing the compiler output we are able to relate increased compile times with additional processing of the compiler inlining features. In particular we see RAJA provides higher than default maximum inline thresholds which when removed drop compile and link times to be close to the OpenMP implementations. Our conclusion is that the addition of C++ layers is more likely to require heavy inlining support (which is seen in the higher Kokkos compile times where default thresholds are used) and therefore their use in larger production settings is likely to result in longer compile and link times.

1.10 Programmer Productivity Metrics for LULESH Implementations

1.10.1 Calculating Programmer Productivity

The Software Engineering community as a whole continues to research into the measurement of programmer productivity and code complexity. Although a number of approaches have been proposed [11, 3] there is no consistent approach to measurement. In order to provide a consistent measurement we have developed a simple approach which is outlined as follows:

1. **Remove Programmer Comments**: comments in code vary considerably across implementations as each programmer describes implementation specifics in a variety of ways. These are stripped out to ensure analysis of differences does attribute any inherent code difference to these comments.

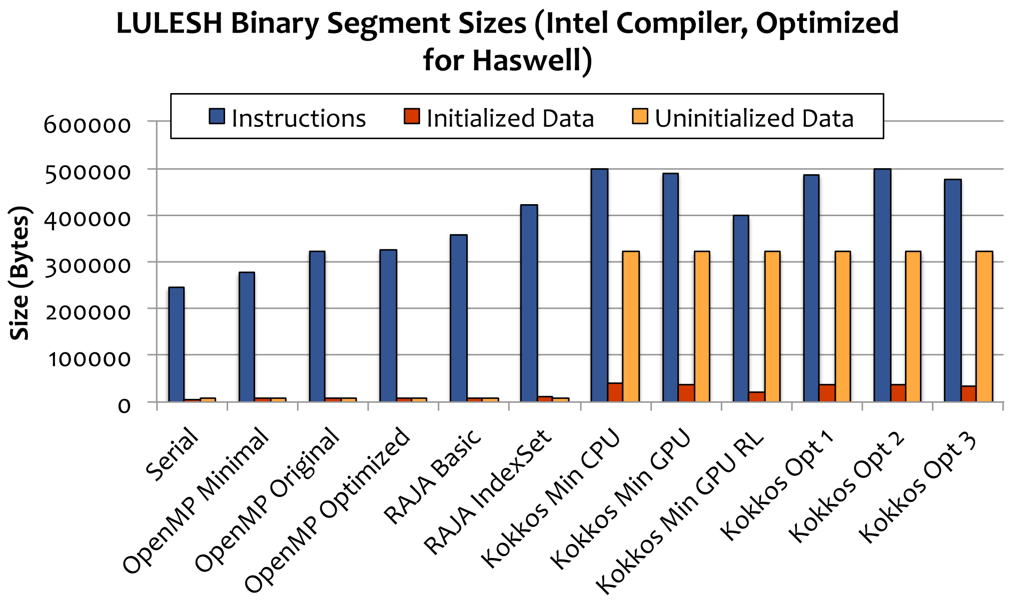


Figure 1.2. LULESH Binary Sizes by Programming Model (compiled using the Intel 15.2 compiler with optimization for the Haswell platform specified)

2. **Format using the `clang-format` utility**: we provide a consistent formatting of source code using the LLVM formatting utilities set to the 'Google' coding style as this allows for longer code lines. The automation of code formatting means we apply a consistent scheme to all files. In some cases minor hand tweaks to the formatting need to be applied for small differences; we have attempted to perform these using the same approach across all implementations to provide fair analysis.

3. **Code Change Sites using Apple FileMerge**: each LULESH implementation is compared for code site changes using Apple's FileMerge developer utility. The baseline source code is our MPI-only variant of LULESH. This tool aggregates multiple sequential different lines to provide an accurate heuristic for the number of places in the code where developer modifications have been made.

4. **Source code lines changed using `diff`**: finally, we utilize the UNIX diff utility to count lines in the implementations which would have to be added or removed when compared to the baseline MPI-only variant.

We note that this approach does not always provide an exact metric but our experience, which has included hand tweaking **all** implementations to reduce errors, the results are representative of the amount of effort our research team has had to apply in developing each code.

1.10.2 Sites of Changes in Code for LULESH Implementations

LULESH has 37 loops which are parallelized in the Minimal-CPU variant. The initial parallelization effort is very similar across OpenMP, RAJA and Kokkos with OpenMP being the least invasive. Modifying code to execute on the GPU with Kokkos doubles the number of change sites from about 125 to 250. But most of those changes are technical in nature requiring the addition of correct mark-up of member functions, and the replacement of std :: vector with Kokkos::vector. While tedious we do not expect that these changes will be as complicated as identifying parallelism and solving thread-safety issues. The non-expert variant

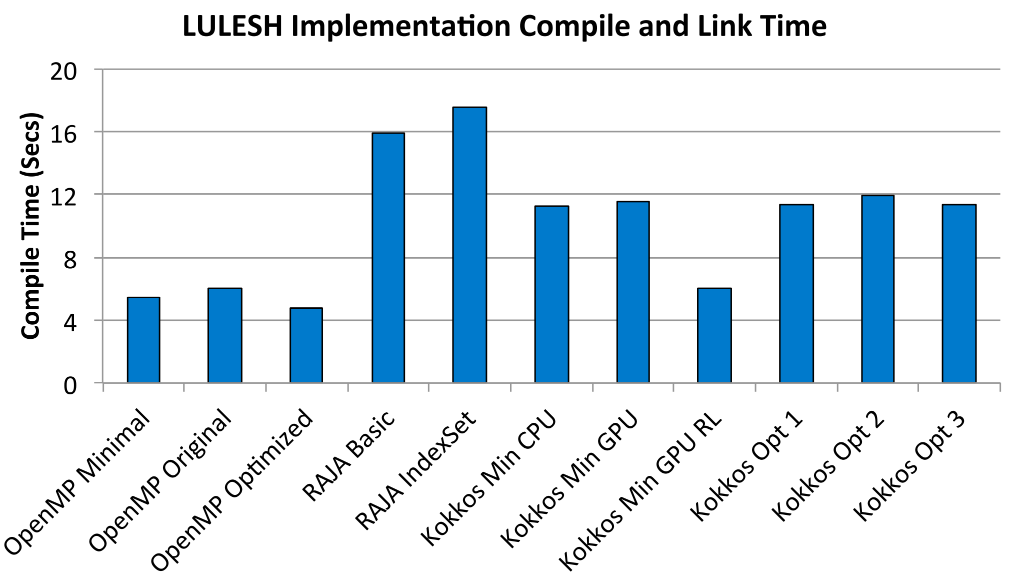


Figure 1.3. LULESH Compile Times by Programming Model (compiled using the Intel 15.2 compiler with optimizations for the Shepard Haswell platform)

follows a lot of the coding recommendations of Kokkos and thus is much closer to the minimal-GPU than the minimal-CPU variant. The reason that the Minimal-CPU variant has less changes than the RAJA variant is that it achieves thread-safety through use of atomic operations instead of changing the algorithm.

In Figure 1.4 we present the sites at which changes to the source code have been made. OpenMP provides the lowest number of sites because it does not require closing brackets to be added to loops as both RAJA and Kokkos need for lambda functions to parse correctly. Ensuring execution on the GPU requires an additional 20 - 30 code sites to be modified. The lowest number of modification sites in the C++ abstraction models (RAJA and Kokkos) occurs with the Kokkos capture-by-reference implementation (Kokkos-CPU-RL) because it requires no modification to the application data structures. We note that the effect of this is considerably poorer application performance.

1.10.3 Source Code Changes for LULESH Implementations

The number of source code lines added, removed and final total is presented in Figure 1.5. The Minimal OpenMP implementation requires the fewest changes but provides poor application performance. Some variation exists between the OpenMP, RAJA and Kokkos implementations because the default time stepping calculation in RAJA is poorly framed for parallel execution and so is modified in these implementations to execute more efficiently.

The conclusion comparing directives to the C++ abstraction layers is that approximately similar lines of code must be modified between the implementations which acceptable levels of performance are desired. There is a clear correlation between improving performance and higher levels of SLOC modification.

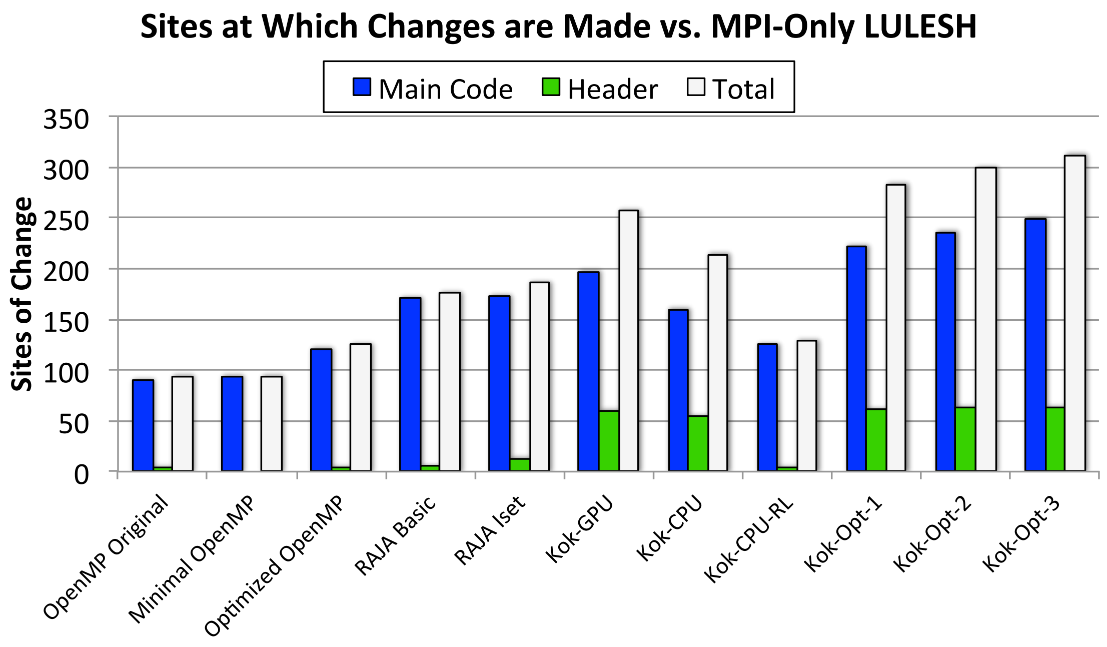


Figure 1.4. Sites in the LULESH source code at which lines of code are changed (multiple lines of code may be changed at each site)

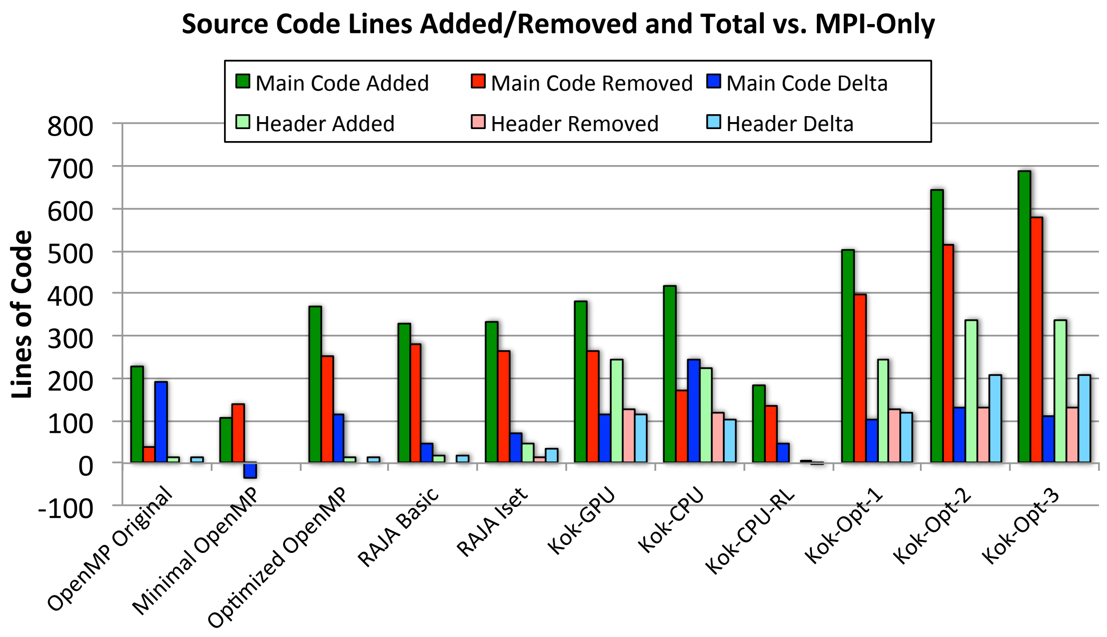


Figure 1.5. Source Code lines added, removed and total for LULESH implementations

Chapter 2

Analysis of Performance and Portability of the MiniAero Mini-Application

2.1 Analysis of MiniAero

MiniAero is a mini-application for the simulation of compressible flow computational fluid dynamics (CFD) problems [7, 8]. It is available through the Mantevo project suite of mini-applications [10]. MiniAero solves the compressible Navier-Stokes equations and employs a cell-centered unstructured mesh finite volume method (first- or second-order accuracy) with explicit Runge-Kutta time integration. An internal mesh generator produces a structured mesh of hexahedral elements, however, unstructured mesh data structures are employed. MiniAero uses MPI for inter-node communication and Kokkos for intra-node parallelism. A layer of ghost cells is communicated across MPI process boundaries. The cell-centered finite volume method involves a loop over all the faces. The flux is calculated across the faces and summed into the two adjacent cells. This will lead to thread conflicts, and MiniAero has two approaches for resolving the thread conflicts: a "gather-sum" approach and an atomic fetch and add approach. The gather-sum approach is a two stage process. For the first stage a loop is performed over the faces. The fluxes are computed then stored at the faces. The second stage is a loop over all the elements, summing the fluxes on the six faces. Since the FY2014 ASC co-design milestone review, there have been various extensions to MiniAero. From the aspects of the physical model and discretization, they include the addition of:

- viscous terms to the Euler equations to solve the Navier-Stokes equations
- second-order discretization for the convective fluxes.

The second-order discretization for the convective fluxes was an important addition as full application codes are second order or higher. This also required the addition of flux limiters. All the results presented employ the second-order discretization. A first order discretization is too simple to be representative of a full application code. There were also several important bug fixes and improvements:

- A key bug was fixed that involved incorrect handling of the ghost element values.
- A memory scaling bottleneck during the mesh generation was removed. As the focus of MiniAero is on the solver performance, and the mesh generation is a preprocessing step, the improved algorithm exploited the fact that the internal mesh generator employed a structured hexahedral mesh. The six neighboring processes are known in advance so the inter-process communication is limited to those six neighboring processes. This replaced an all-to-all communication for discovery of neighboring processes.
- The original restriction of powers-of-two process decomposition has been removed. The improved algorithm that allows non-powers-of-two decomposition also improved load balance.

2.1.1 Performance Analysis of MiniAero

In the ASC L2 Trilab codesign milestone for FY14, MiniAero was analyzed for on-node performance using several ASC platforms including Sandy Bridge, BlueGene/Q, the Intel Xeon Phi Knights Corner accelerator, AMD's Interlagos processor and NVIDIA's K20 Kepler GPU for the first order method. In the following sections we show updated scaling analysis on newer platforms for the second order method and show multi-node analysis. The single node comparisons include a comparison of the effect of the choice of number of MPI processes and OpenMP threads for both a Haswell and POWER8 compute node. The test case employed for the following performance analysis is a 30° ramp in a Mach 2.5 inviscid flow with second-order method. Note that the problem size varied with the different studies.

On-Node Performance Comparison

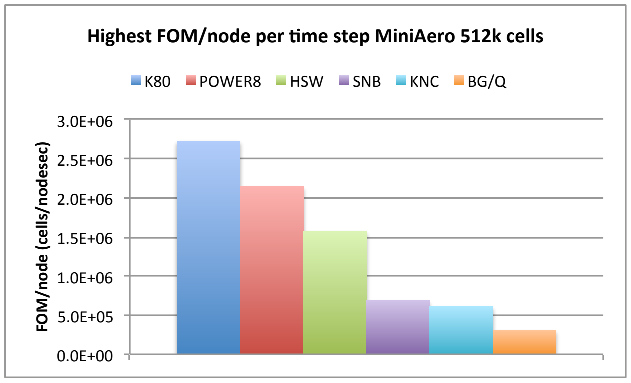


Figure 2.1. MiniAero highest performance (cell updates per node per second per time step) for 128x64x64 (516,000) cells for: Shannon K80 (1 out of the 2 GPUs), White dual-socket POWER8, Shepard dual-socket Haswell, TLCC2 dual-socket Sandy Bridge, Compton single KNC card (1 MPI process with 224 OpenMP threads, BG/Q compute node (1 MPI process with 64 OpenMP threads)

Figure 2.1 compares the performance for the architectures described above (some descriptions repeated from Section 1.2 for convenience). The performance is given in terms of the number of cell updates per compute node per second per time step. The problem size is 128x64x64 (516,000) cells and is dictated by the requirement to fit in the 6GB memory of the KNC.

- Shannon K80 (1 out of the 2 GPUs); atomics gives higher performance than gather-sum
- White dual-socket POWER8; 20 MPI processes each with 8 OpenMP threads was optimal, i.e. 1 MPI process with 8 OpenMP threads per core; gather-sum gives higher performance than atomics
- Shepard dual-socket Haswell; 2 MPI processes each with 32 OpenMP threads was optimal (used both SMT threads on a core); gather-sum gives slightly higher performance than atomics
- TLCC2 dual-socket Sandy Bridge (2.6GHz Intel Xeon E5-2670); MPI-only with 16 MPI processes was optimal; built with Intel 14.0.4.211 compiler; gather-sum gives higher performance than atomics
- Compton single KNC (1.1GHz 57-core); 1 MPI process with 224 OpenMP threads; atomics gives higher performance than gather-sum

- BG/Q compute node (single socket with 16 1.6GHz A2 cores for compute, each supporting 4 hardware threads); 1 MPI process with 64 OpenMP threads; built with GNU 4.7.2 compiler; gather-sum gives higher performance than atomics

As expected, the K80, POWER8 and Haswells give considerably higher performance than the three other architectures (other architectures are older architectures, e.g. Sandy Bridge has half the number of cores as Haswell and KNC has shortcomings which are hoped to be remedied by KNL, while a BG/Q socket uses substantially less power than the others).

Figure 2.2 compares the performance on a single dual-socket Haswell compute node on Shepard for the permutations of MPI processes and OpenMP threads using hyperthreading, from 64 MPI processes (MPI-only) to a single MPI process with 64 OpenMP threads. The mesh used for this study is 1024x128x128 (16.8 million) cells and is a factor of 32 times larger than the mesh used for the study in the previous figure (Figure 2.1) and is intended to have a memory usage closer to that used for a full application. For the 64 MPI process case, the memory usage is about 90GB, while for the 64 OpenMP thread case the memory usage is about 40GB. For this test case, the use of hyperthreading increases the performance. The atomics approach (presented in the figure) gives slightly higher performance than the gather-sum approach. The performance for the different permutations of MPI processes and OpenMP threads are effectively the same with the exception of the case with 64 OpenMP threads. This is due to having a single MPI process with threads on two sockets, i.e. two NUMA regions. Although the use of threads may not have a performance advantage over MPI-only for this test case, it definitely has a memory advantage as the MPI-only case requires a factor of 2.25 of the memory required by the case with 64 threads.

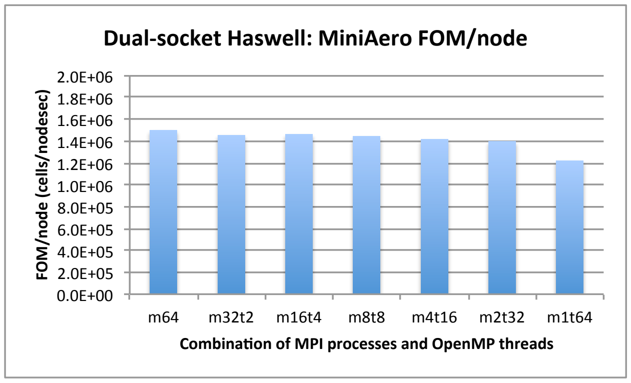


Figure 2.2. MiniAero performance for 1024x128x128 (16.8 million) cells for dual-socket Haswell comparing the peformance for the different permutations of MPI processes and threads.

Figure 2.3 compares the performance on a single dual-socket POWER8 compute node on White for the permutations of MPI processes and OpenMP threads using hyperthreading, from 160 MPI processes (MPI-only) to a single MPI process with 160 OpenMP threads. The mesh used is 1024x128x128 (16.8 million) cells and is the same as that used for the Haswell study in the previous figure (Figure 2.2). The gather-sum approach (presented in the figure) provides higher performance than the atomics approach. For this test case, the performance for the permutations of MPI processes and OpenMP threads are effectively the same with the exception of the MPI-only case (160 MPI processes) and the two cases where an MPI process has threads that span more than one NUMA region (the case with 2 MPI processes each with 80 OpenMP threads and 1 MPI process with 160 OpenMP threads).

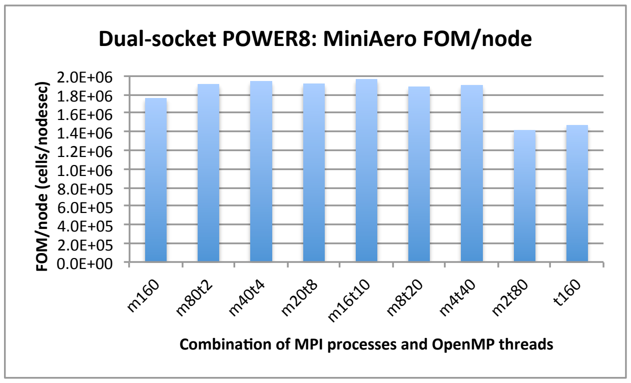


Figure 2.3. MiniAero performance (cell updates per node per second per time step) for 1024x128x128 (16.8 million) cells for dual-socket POWER8 comparing the performance for the different permutations of MPI processes and threads.

Strong Scaling

Strong scaling studies of the optimized MiniAero code were performed on two platforms, Mutrino and Volta. Results for a 1024x64x64 (4.2 million) cell mesh are shown in Figures 2.4–2.5. The figures plots performance per node against number of nodes. Performance is given in cell updates per second. The same metric will be used for all following plots. A combination of MPI processes and OpenMP threads is used in a fashion so that 32 (i.e. all) cores per node are used on Mutrino and 8 cores per socket are utilized on Volta. As seen in the figures, as the number of nodes increases, the application scales the same regardless of the how the ranks and threads are implemented. The only exception is the use case of 1 MPI rank with 8 threads per socket on Volta (i.e. 1 MPI rank with 16 threads, 8 on each socket), which is about 20% slower due to a single MPI rank spanning more than one NUMA region. This is consistent with the study in Figure 2.2 where the performance is similar as long as each socket has at least one MPI process (the performance decreases for the case for one MPI process for two sockets, with OpenMP threads on both sockets). In Figure 2.4 the most threads used is 16 OpenMP threads for one MPI process, but all 16 OpenMP threads are on the same socket.

Weak scaling

Weak scaling studies are done on several platforms, although in some of these studies, we use a smaller problem size due to memory constraints. Figure 2.6 presents a weak scaling study with 1024x64x64 (4.2 million) cell mesh per compute node for Mutrino. Figures 2.7–2.9 present weak scaling studies for Shannon K80, Compton KNC and Vulcan BG/Q platforms, respectively. These studies employ 128x64x64 (516,000) cells per GPU, KNC or BG/Q compute node. The problem size was dictated by the requirement to fit in the 6GB memory of the KNC. For each KNC have a single MPI process with 224 OpenMP threads. For a BG/Q compute node, all 64 hardware threads are employed, either with 64 MPI processes or 64 OpenMP threads. The largest BG/Q runs are on 4096 nodes with 2 billion degrees of freedom. For the MPI-only case, using 64 MPI processes per compute node gives a total of 262,144 MPI processes. The performance of the atomics version of MiniAero is considerably worse than the gather-sum version on BG/Q, but atomics are slightly better than gather-sum on the KNC, and atomics are significantly better than gather-sum on the GPU. From Figure 2.9 one can see that MPI+OpenMP threads gives higher performance than MPI-only for the larger numbers of nodes.

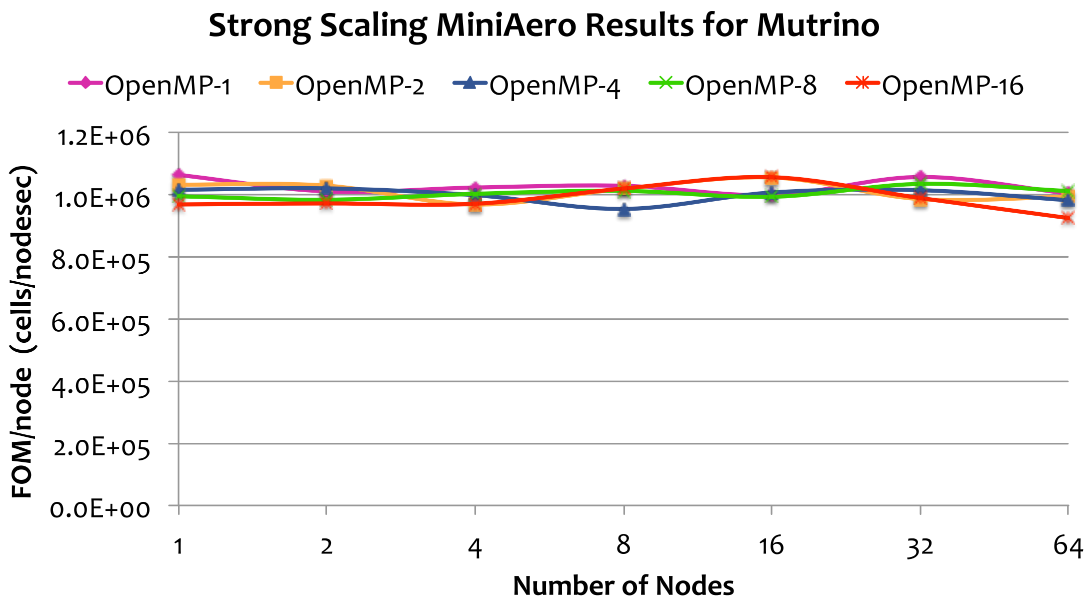


Figure 2.4. MiniAero strong scaling on Mutrino with 1024x64x64 (4.2 million) cell mesh.

Figure 2.10 presents the weak scaling study for the average memory usage per compute node corresponding to Figure 2.9. As expected, gather-sum has a larger memory requirement than atomics because of the need to store fluxes at the faces. Also as expected, MPI+OpenMP has a lower memory requirement than MPI-only because the latter has more replication of data for ghosting. Note that as the problem size is scaled up, the average memory usage per compute node is increasing faster for the MPI-only case while the memory usage for the MPI+OpenMP threads case stays mostly flat. For the MPI-only case at 4096 compute nodes, 262,144 MPI processes are used. We have previously observed large memory increases for other miniapps as well as application codes at very large numbers of MPI processes. Clearly the use of MPI and threads provides a memory advantage over MPI-only. Combining the use of atomic operations with threading achieves a memory footprint reduction of more than 1/3 for large node counts.

2.1.2 MiniAero binary size

Figure 2.11 presents the miniAero binary size for Haswell. As with the LULESH binary size presentation, we breakdown the binary size by the number of instructions generated during compile and the binary space allocated for initialized and uninitialized static data structures.

2.1.3 Vectorization and Instruction Analysis of MiniAero

The ATS-1 Trinity Phase-I and Phase-II deployments will both bring new processors and instruction set architectures to DOE computing facilities. In Phase-I Intel Xeon Haswell processors will be used which will provide AVX2 instructions that maintain the 256-bit width of AVX found in TLCC-2 machines but augment the capabilities with vectorized integer operations, gather/scatter instructions and fused-multiply-add for floating point. The availability of gather/scatter memory access instructions has the potential to improve vectorization opportunities as operands can now be more easily gather sparsely in memory. In the Phase-II

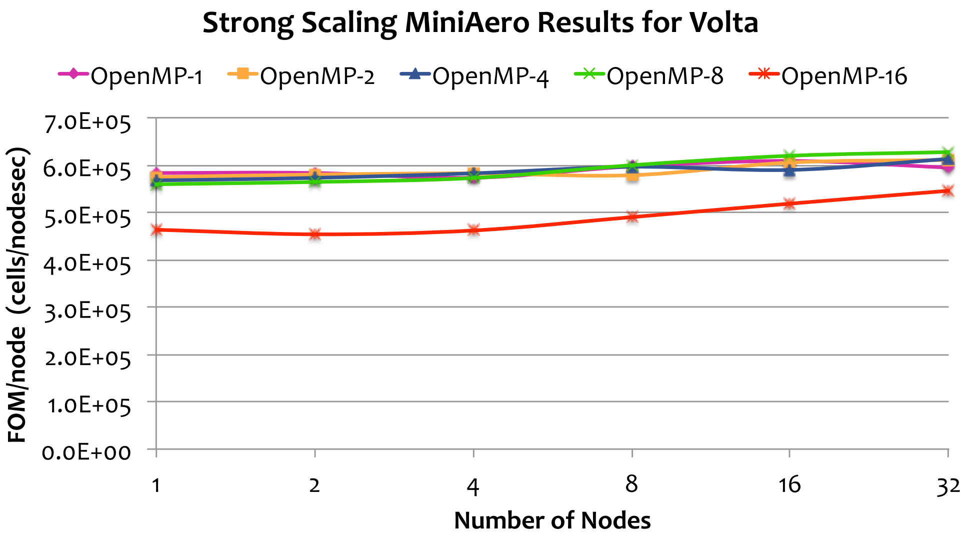


Figure 2.5. MiniAero strong scaling on Volta with 1024x64x64 (4.2 million) cell mesh.

deployment of Trinity application developers will need to make their code work on the Intel Knights Landing Xeon Phi processor (KNL). The KNL will offer wider (512-bit) vector operations and additional capabilities in the form of lane-masking, gather/scatter conflict detection and others.

In Figure 2.12 we present a breakdown of the dynamic instructions executed by MiniAero on Sandy Bridge, Haswell and Knights Landing emulation environments. The instructions recorded relate only to the use of the vector-units (so this is a subset of the total instructions executed but includes all floating point operations). The Figure shows that approximately 80% of all vector instructions executed on Sandy Bridge and Haswell architectures and 75% of instructions on Knights Landing are executed over a scalar operand.

The respective width of the instructions executed, 128-, 256- and 512-bit, is shown in green, yellow and red respectively. For KNL roughly 10% of instructions utilize the new SIMD vector units.

Given that the majority of floating point compute capability is being directed toward SIMD vector units in future designs the inability of application code to vectorize indicates a potential concern for future runtime performance. This highlights a recurring theme from analysis of our mini-applications and larger production codes – vectorization levels are often very low. Our runtime analysis, from which these results are derived, allows us to inspect application functions and even basic blocks by SIMD instruction execution allowing us to locate the most frequently executed functons which do not have high levels of vectorization. The approach used in this study is being actively developed and enhanced to support application code teams in preparation for the Trinity Phase-II deployment in 2016.

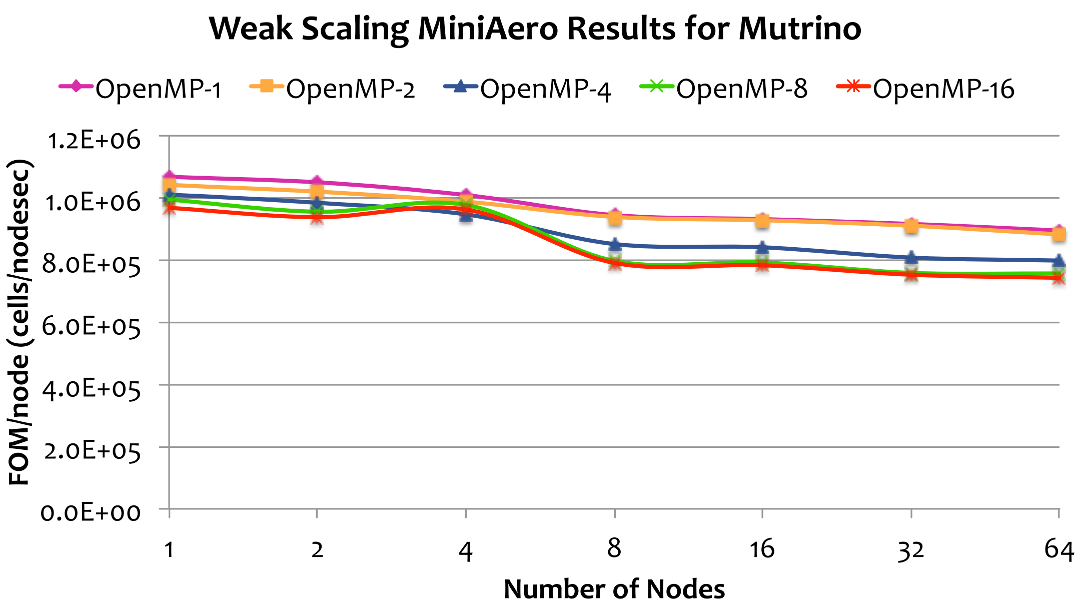


Figure 2.6. MiniAero weak scaling on Mutrino with 1024x64x64 (4.2 million) cell mesh per compute node.

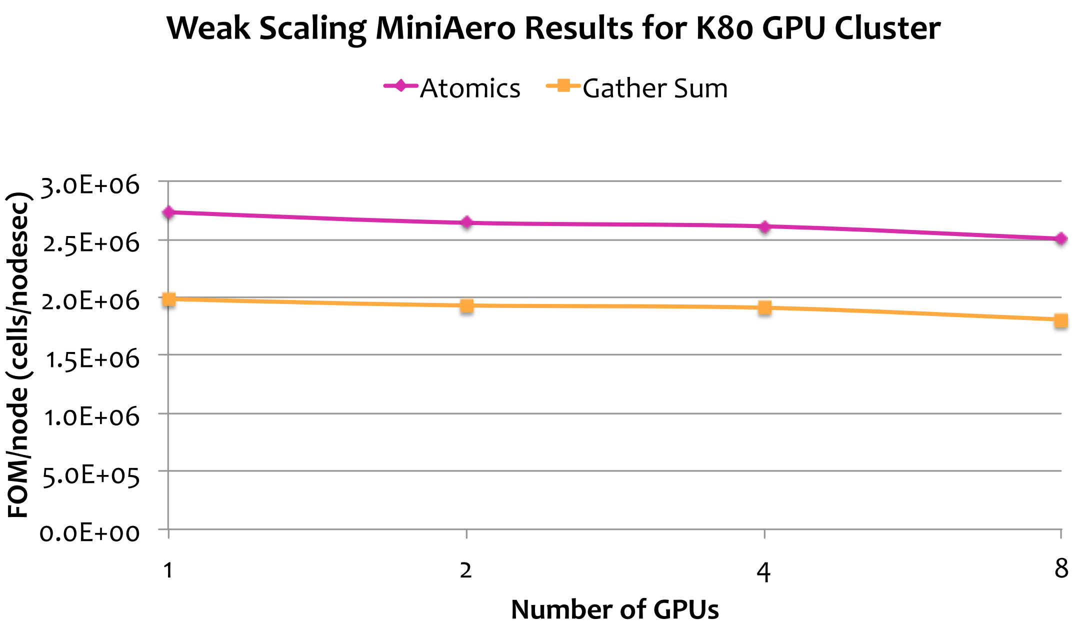


Figure 2.7. MiniAero weak scaling on Shannon K80 platform. 128x64x64 (516,000) cells per GPU (half of the K80)).

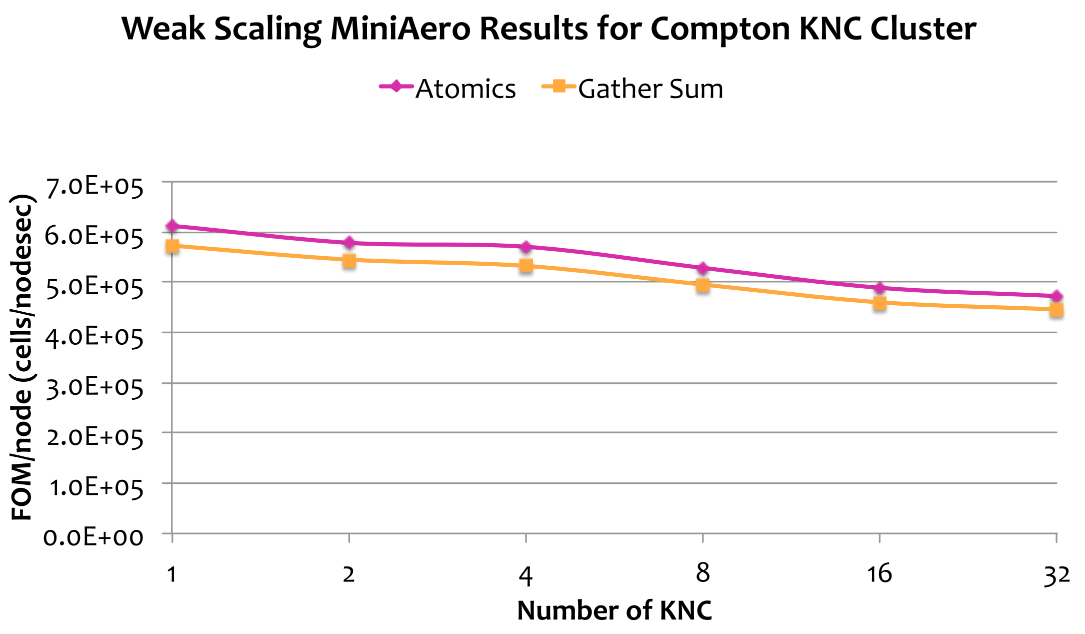


Figure 2.8. MiniAero weak scaling on Compton KNC platform (using both KNC cards per compute node). 128x64x64 cells (516,000) per KNC card (has 6GB RAM). One MPI process per KNC with 224 OpenMP threads.

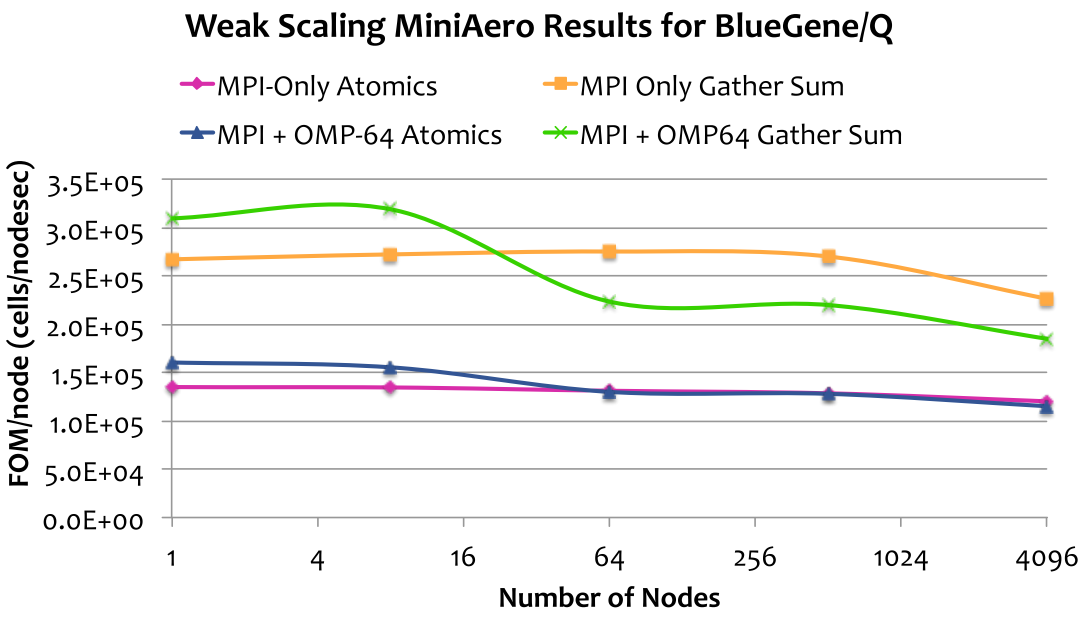


Figure 2.9. MiniAero weak scaling on LLNL Vulcan BG/Q platform. 128x64x64 (516,000) cells per compute node. For MPI-only case, used 64 MPI processes per compute node. For MPI+OpenMP case, use 1 MPI process with 64 OpenMP threads per compute node. 262,144 MPI processes for the MPI-only case on 4096 compute nodes.

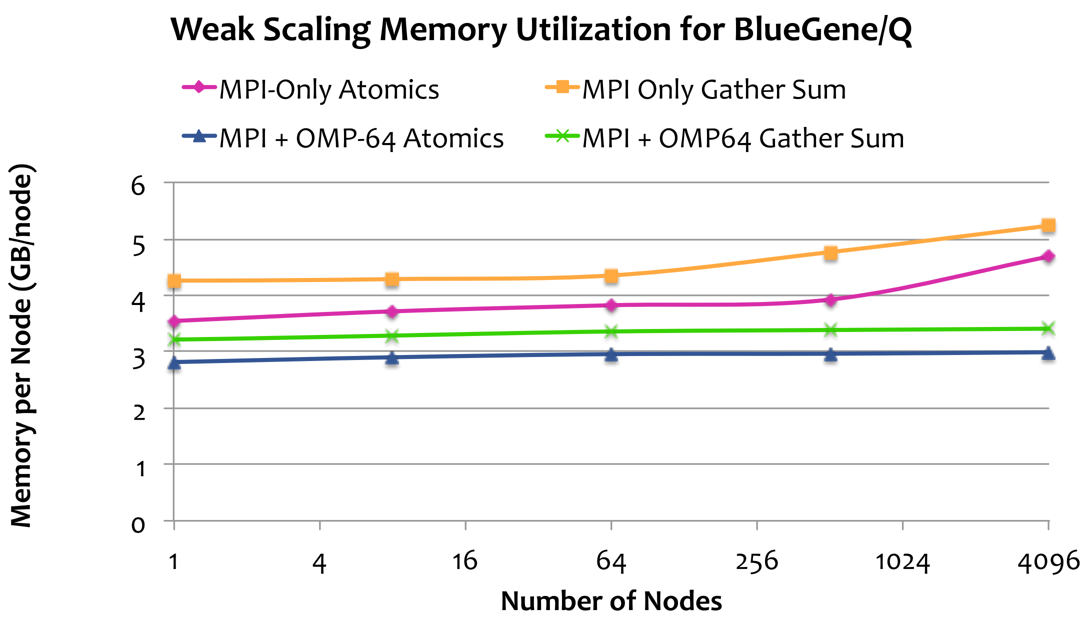


Figure 2.10. MiniAero weak scaling on LLNL Vulcan BG/Q platform memory usage. 128x64x64 (516,000) cells per compute node. For MPI-only case, used 64 MPI processes per compute node. For MPI+OpenMP case, use 1 MPI process with 64 OpenMP threads per compute node.

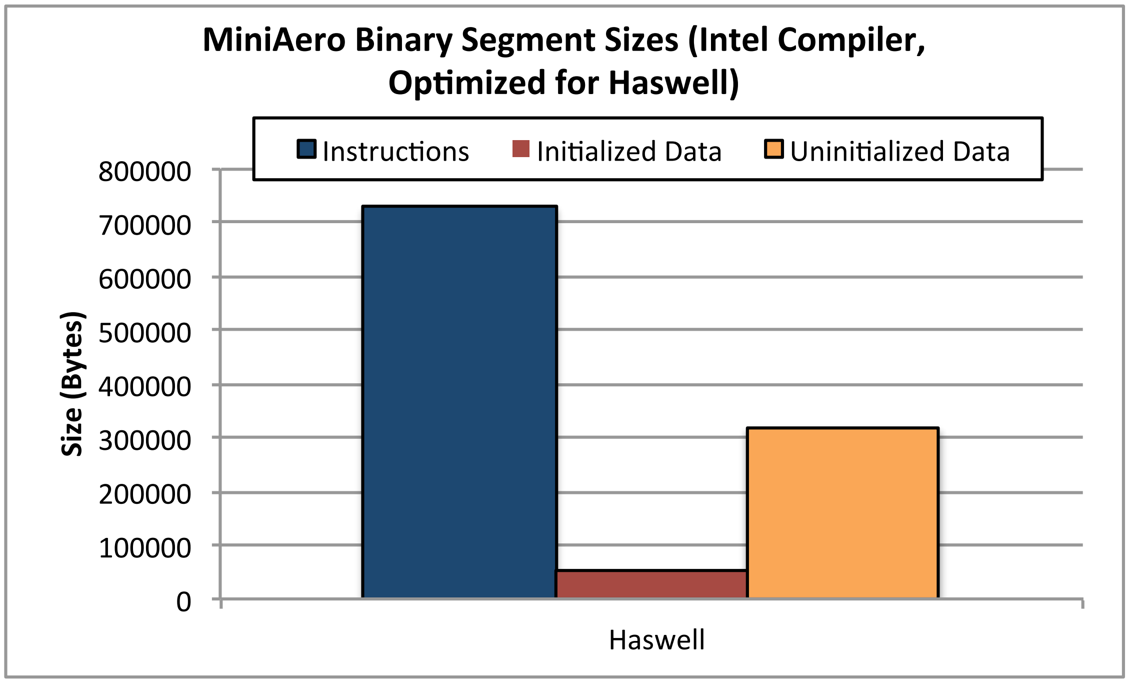


Figure 2.11. MiniAero binary size for Haswell (compiled using Intel 16.0.042 compiler).

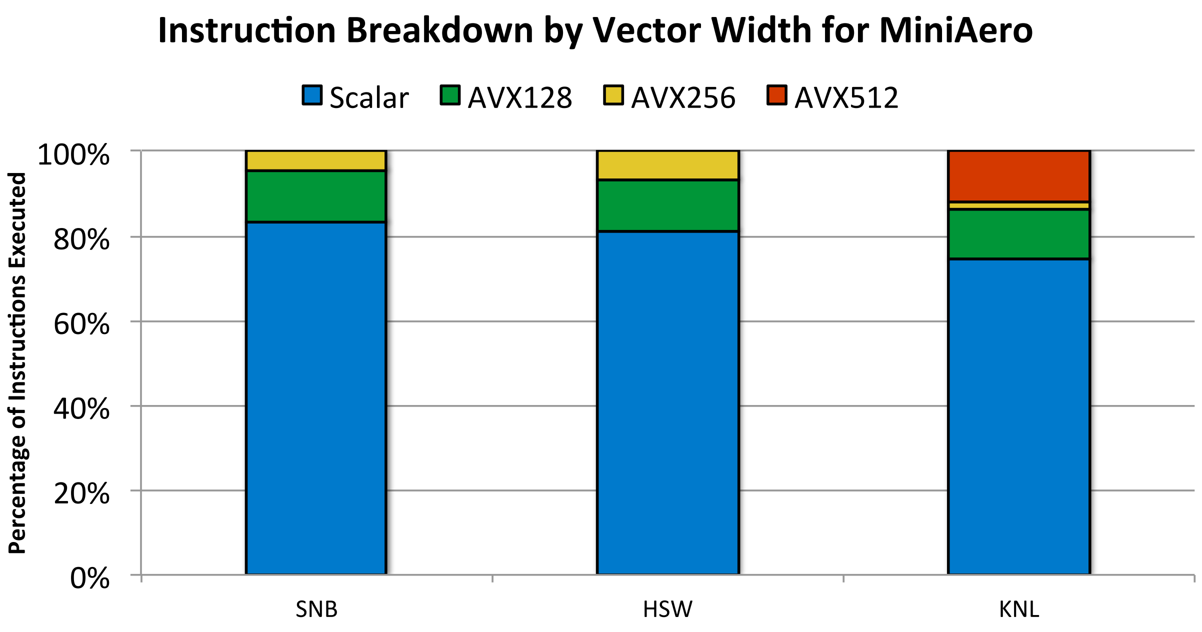


Figure 2.12. Instruction Analysis of MiniAero for Sandy Bridge (SNB), Haswell (HSW) and Knights Landing (KNL)

Chapter 3

Conclusions

The ASC L2 Trilab Milestone results shown in this report represent the first, albeit limited, exhaustive comparison of abstract C++ programming models to the use of directive-based solutions. For our C++ abstractions we have utilized both the RAJA and Kokkos models from Lawrence Livermore and Sandia National Laboratories respectively.

We have shown that the overhead to programmers within the context of the LULESH benchmark is approximately equal between the parallel dispatch mechanisms used. Data structure and access abstractions add overhead but result is significant increases in application performance. Our comparison of binary sizes and compile times shows that the C++ approaches result in larger bodies of instructions and variable compile times depanding on the level of inlining the compiler decides to introduce.

In the second half of study we demonstrated performance analysis of the MiniAero mini-application. The experience correlates with the LULESH benchmarking numbers – Kokkos is able to show strong application performance across a variety of advanced high performance computing solutions.

Our studies show that C++ abstractions, particularly Kokkos, are able to deliver strong portability between architectures, strong performance for each of the machines used, at similar levels of programmer overhead to community perceived “lightweight” solutions. We argue instead that the significant overhead is not down to the programming mechanisms used but rather the more challenging aspects of creating thread-scalable algorithms that can efficiently execute on machines with high degrees of parallelism. In our introduction we develop a “swim-lines” timeline for discussion purposes that shows how the experiences described in this report might be mapped into leading ASC platforms and production code teams. The challenges of the future ATS-class platforms are many, but a stepwise approach to porting our codes means we will continue to have a world-leading production code base as we adopt new classes of supercomputers.

References

[1] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[2] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186. IEEE, 2010.

[3] Edward T. Chen. Program Complexity and Programmer Productivity. *IEEE Transactions on Software Engineering*, (3):187–194, 1978.

[4] H Carter Edwards and Daniel Sunderland. Kokkos Array Performance-Portable Manycore Programming Model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10. ACM, 2012.

[5] H Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. Manycore Performance-Portability: Kokkos Multidimensional Array Library. *Scientific Programming*, 20(2):89–114, 2012.

[6] 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.

[7] Kenneth Franko. MiniAero: A performance portable mini-application for compressible fluid dynamics. In preparation.

[8] Kenneth J. Franko, Travis C. Fisher, Paul Lin, and Steven W. Bova. CFD for next generation hardware: Experiences with proxy applications. In *Proceedings of the 22nd AIAA Computational Fluid Dynamics Conference*, 2015. AIAA 2015–3053.

[9] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[10] M.A. Heroux, D.W. Doerfler, P.S. Crozier, J.M. Willenbring, H.C. Edwards, A. Williams, M. Rajan, E.R. Keiter, H.K. Thornquist, and R.W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574.

[11] Lorin Hochstein, Jeffrey Carver, Forrest Shull, Shadnaz Asgari, Victor Basili, Jeffrey K Hollingsworth, and Marvin V Zelkowitz. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 conference*, pages 35–35. IEEE, 2005.

[12] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

DISTRIBUTION:

1	MS 1322	K.F. Alvin, 01420
1	MS 1319	J.A. Ang, 01422
1	MS 1319	K.S. Hemmert, 01422
1	MS 1319	J.H. Laros, 01422
1	MS 1319	S.D. Hammond, 01422
1	MS 1319	R.J. Hoekstra, 01426
1	MS 1319	M.A. Heroux, 01426
1	MS 1319	C.R. Trott, 01426
1	MS 0845	S.A. Hutchinson, 01540
1	MS 0845	T.C. Fisher, 01541
1	MS 0845	M.W. Heinstein, 01542
1	MS 0845	K.H. Pierson, 01542
1	MS 0845	M.W. Glass, 01545
1	MS 0899	Technical Library, 9536 (electronic copy)

v1.37

Sandia National Laboratories