Unstructured-grid Algorithms for a Many-core Landscape

Aaron Walden and Eric Nielsen
NASA Langley Research Center

Mohammad Zubair and Jason Orender
Old Dominion University

Justin Luitjens
NVIDIA Corporation

John Wohlbier
DoD HPCMP PETTT
Engility Corporation

John Linford, Izaak Beekman, Sam Khuvis, and Sameer S. Shende
ParaTools, Inc.

The authors would like to acknowledge the computational resources and PETTT software support from the DoD High Performance Computing Modernization office under Contract No. GS04T09DBC0017. Support from the NASA Langley Research Center Comprehensive Digital Transformation initiative and the Revolutionary Computational Aerosciences sub-project within the NASA Aeronautics Research Mission Directorate is also acknowledged. The authors also wish to acknowledge Dr. Larry Davis of the DoD HPCMP Office for hardware resources used in this study. Some resources supporting this work were provided by the NASA High-End Computing (HEC) Program through the NASA Advanced Supercomputing (NAS) Division at Ames Research Center. This research also used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

Motivation: Future HPC Landscape

- Exascale power requirements constrain processor operating voltage and frequency, favoring performance increases through concurrency
- DoE Exascale Workshop predicts 1024 cores/node, already at 260 (TaihuLight)
- Upcoming ORNL Summit: 6 NVIDIA Volta V100/node (6 × 5,120 “CUDA cores”)
- Current multi-core compute nodes ~50-way concurrency
- Increasing core counts and proliferation of high bandwidth/low capacity memory as well as exponential increase in flops vs bandwidth emphasize the need to minimize memory footprint and communication (data movement)
- Simulation of time-dependent problems calls for strong scaling
- Dense domain-decomposed MPI may be rendered inadequate by the coming paradigm shift

Conventional FUN3D

- FUN3D is a NASA Langley unstructured-grid CFD solver: uses implicit time integration on mixed-element unstructured grids
- Elements (tet/prism/pyramid/hex) have different compute costs
- FUN3D employs a coarse-grained domain-decomposed dense MPI model (1 rank per processing element) in which each rank independently processes a grid partition
- Experience shows partition quality shrinks with $n_{grid\_points} / n_{ranks}$ ratio, causing load imbalance by element type for heterogeneous workloads
- As $n_{partitions}$ increases, so does the $surface/volume$ ratio of each partition and thus the number of halo exchanges, which we suspect as the prime limiter of scalability
- Thus there is a clear motivation to reduce the number of MPI ranks over which a grid is decomposed, especially for many-core systems

Compute "Node" Architectures

	HWL	BWL	SKY	KNL	P100	V100
Architecture	Haswell	Broadwell	Skylake	Knights Landing	Pascal	Volta
Model	E5-2699v3	E5-2680v4	Gold 6148	KNL 7230	P100 PCIe	V100 SXM
NUMA x Cores x Threads	4 x 9 x 2	2 x 14 x 2	2 x 20 x 2	1 x 64 x 4	1 x 56 x 32	1 x 80 x 32
Clock Speed, GHz	2.3	2.4	2.4	1.3	1.303	1.53
Vector Length, DP	4	4	8	8	--	--
Memory, GB	116	128	192	16/90	16	16
Memory Bandwidth, GB/s	106	117	163	450/80	720	900
Peak GFLOPS, DP	530	431	1229	2662	4670	7834
MSRP, US$	4115	3490	6156	1992	5500	8000
TDP, Watts	290	240	300	215	250	300

- Many-core (KNL, GPUs) will always run in shared memory
- Xeon (BWL, SKY) will not (for node-level studies), serving as a benchmark of current practice

Outline

1. **Shared-memory Node-level Programming Models**
    - LHS Matrix Assembly
    - Linear Solver
2. **Node-level Performance**
3. **Fortran Considerations**
4. **Strong Scaling Performance**
    - Hybrid Shared-mem MPI+OpenMP vs. Pure MPI
    - Multi-GPU Systems

LHS Matrix Assembly:

Compressible Viscous Flux Jacobians

LHS: Algorithm

```
Initialize $A_{DIAG} \leftarrow 0$ and $A_{OFF} \leftarrow 0$

for each cell $\in$ $Grid$ do
  for each node $\in$ $cell$ do // loop 1
    // compute cell avgs, set local arrays
  end for
  for each face $\in$ $cell$ do  // loop 2
    // linearize cell gradients
  end for
  for each edge $\in$ $cell$ do  // loop 3
    // compute edge contribution to Jacobian
    for each node $\in$ $cell$ do
      // compute gradients at dual face
    end for
  end for
  for each node $\in$ $cell$ do // loop 4
    // assemble 17 contributions to Jacobian
  end for
end for
```

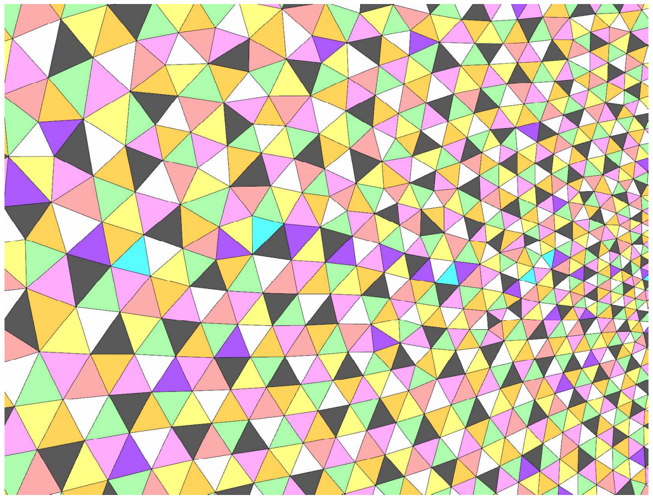
A_{DIAG}	: Diagonal block matrix
A_{OFF}	: Off-diagonal block-sparse matrix

- The computation can be parallelized over the number of cells, however, **atomic updates are required** to avoid race conditions when writing to A_{DIAG} and A_{OFF}
- Challenges:
  - Irregular memory access pattern
  - Algebraic complexities (dependency chains) related to the underlying physics limit vectorization
  - A large number of temporary variables results in cache and register pressure

Langley Research Center

Race Condition Avoidance

To avoid race conditions during matrix updates, we must serialize the processing of elements that share nodes. For GPUs, we use the hardware-supported `atomicAdd()` with virtually no loss of performance. For Xeon/Xeon Phi, we attempted the following strategies:


Example greedily colored grid.

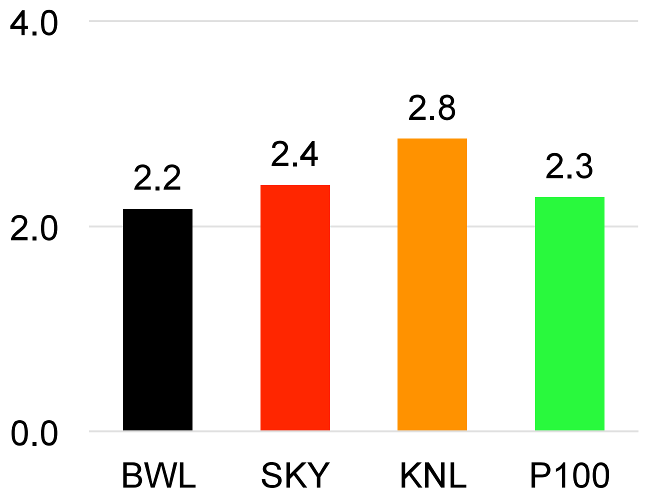
- Atomics: We use OpenMP atomics to protect all matrix updates. This may be optimized so that only nodes shared by threads are protected. 200+% performance penalty.
- Coloring: Using a greedy algorithm, we organize cells into groups which do not share nodes. This generally requires 12-15 color groups. The more scattered memory access pattern incurs a 30-60% performance penalty.

Langley Research Center

LHS: General Optimizations

Using conventional FUN3D Fortran (or direct CUDA C++ port) as baseline:

- Factor algebra to avoid recomputation
- Store computed addresses in lookup table
- Hard-code loop extents
- Prefetch data to reduce memory latency



Speedup over baseline.

Langley Research Center

LHS: Further GPU Optimizations

```
Initialize $A_{DIAG} \leftarrow 0$ and $A_{OFF} \leftarrow 0$

for each cell $\in$ Grid do
  for each node $\in$ cell do
    // compute cell averages, set local arrays
  end for
  for each face $\in$ cell do
    // linearize cell gradients
  end for
  for each edge $\in$ cell do
    // compute edge contributions to jacobia
    for each node $\in$ cell do
      // compute gradients at dual face
    end for
  end for
  for each node $\in$ cell do
    // assemble 17 contributions to Jacobian
  end for
end for
```

Parallelize across gridDim.x * blockDim.y threads

Parallelize using blockDim.x threads

Flatten nested loops and parallelize using blockDim.x threads

Parallelize using blockDim.x threads

- Increases number of active threads and improves thread utilization
- Coalesce memory access pattern
- Reduces register and shared memory pressure, increasing occupancy
- Enable reduction in inner loops using shared memory
- Auto-tuning used to choose `blockDim.x` and `blockDim.y`
- Further 2× speedup

Multicolor Point-Implicit Linear Solver

Solver: Algorithm

x : Solution vector (initialize to zero)
L_D^k : Lower triangular of A_{DIAG}^k
U_D^k : Upper triangular of A_{DIAG}^k

```
for $i$ = 1 to number_of_sweeps do
  for $k$ = 1 to number_of_colors do
    // Compute halo values
    Compute $q^k \leftarrow b^k - A_{OFF}^k x$
    Solve for $y^k$ in $L_D^k y^k = q^k$
    Solve for $x^k$ in $U_D^k x^k = y^k$
    // Non-blocking MPI send/rec halo
    // Compute interior values
    Compute $q^k \leftarrow b^k - A_{OFF}^k x$
    Solve for $y^k$ in $L_D^k y^k = q^k$
    Solve for $x^k$ in $U_D^k x^k = y^k$
    // MPI_Waitall
  end for
end for
```

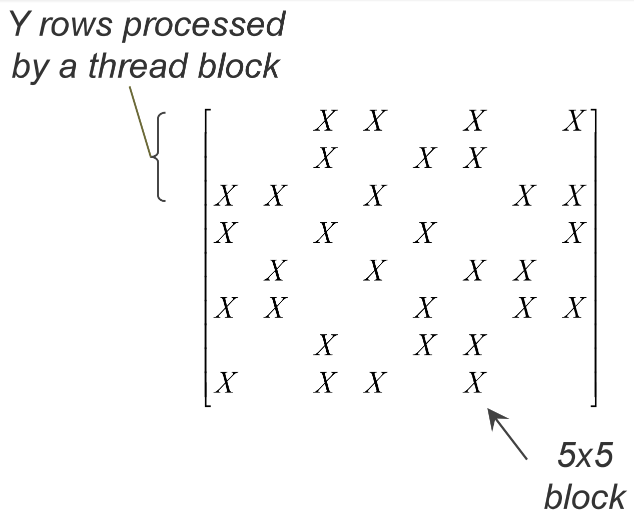
FUN3D uses a series of multicolor point-implicit sweeps to form an approximate solution to **Ax = b**

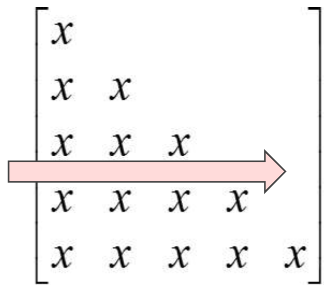
- Color by rows which share no adjacent unknowns
- Re-order matrix rows by color contiguously in memory in block CSR format
- The algorithm requires a block-sparse matrix-vector product and forward-backward substitutions
- Halo rows ordered and processed first, then call non-blocking MPI send/receive for halos
- Computation of interior values proceeds as halos are exchanged
- Blocking MPI_Waitall follows interior computation
- Strong scaling heavily dependent upon interior computation effectively hiding comm. latency

Langley Research Center

Solver: CUDA Implementation

- CUDA sparse libraries exist, but determined to be inadequate, developed and optimized a custom solver
- Process rows of one color with $nRowsInColor/Y$ thread blocks
- Process a 5x5 block with the first 25 threads of warp (*X* threads)
- Aggregate partial sums of matrix-vector product using shuffles
- Store all intermediate results and diagonal block in shared memory
- Auto-tune block sizes (*X*, *Y*) and launch bounds
- The columns of the lower triangular factor of A_{DIAG} are processed from left to right using a single warp
- The amount of parallelism available to the warp decreases as we move from left to right
- Shuffle instruction broadcasts values from the previous column
- Upper triangular portion processed in a similar fashion

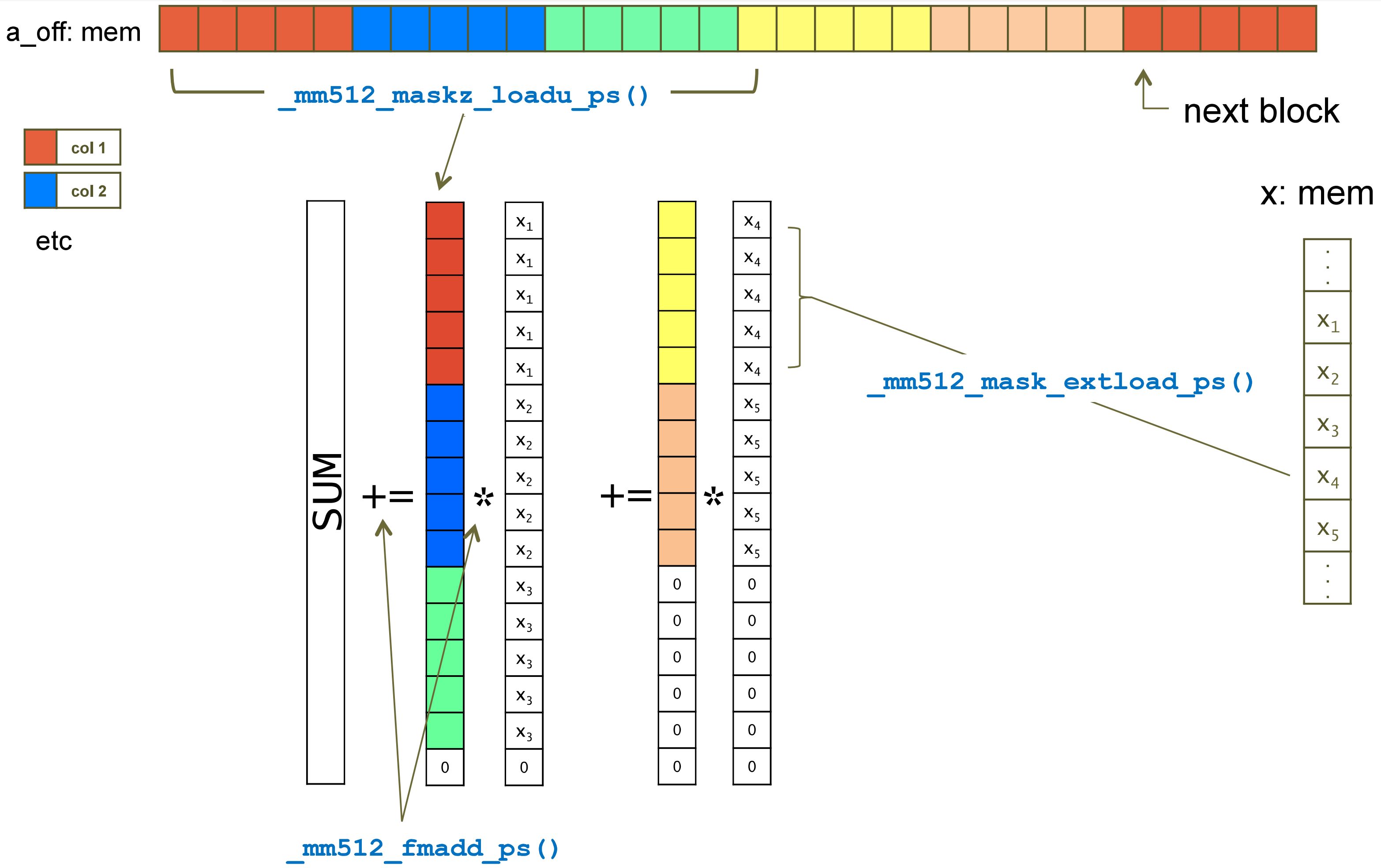




Solver: AVX-512 Implementation

- For KNL (and now SKY), solver is amenable to writing in AVX-512 vector intrinsics
- Vector intrinsics are essentially assembly that controls AVX-512 vector unit (SIMD operations)
- Bypass compiler vectorization
- Instead of thread blocks, work directly with 512-bit registers

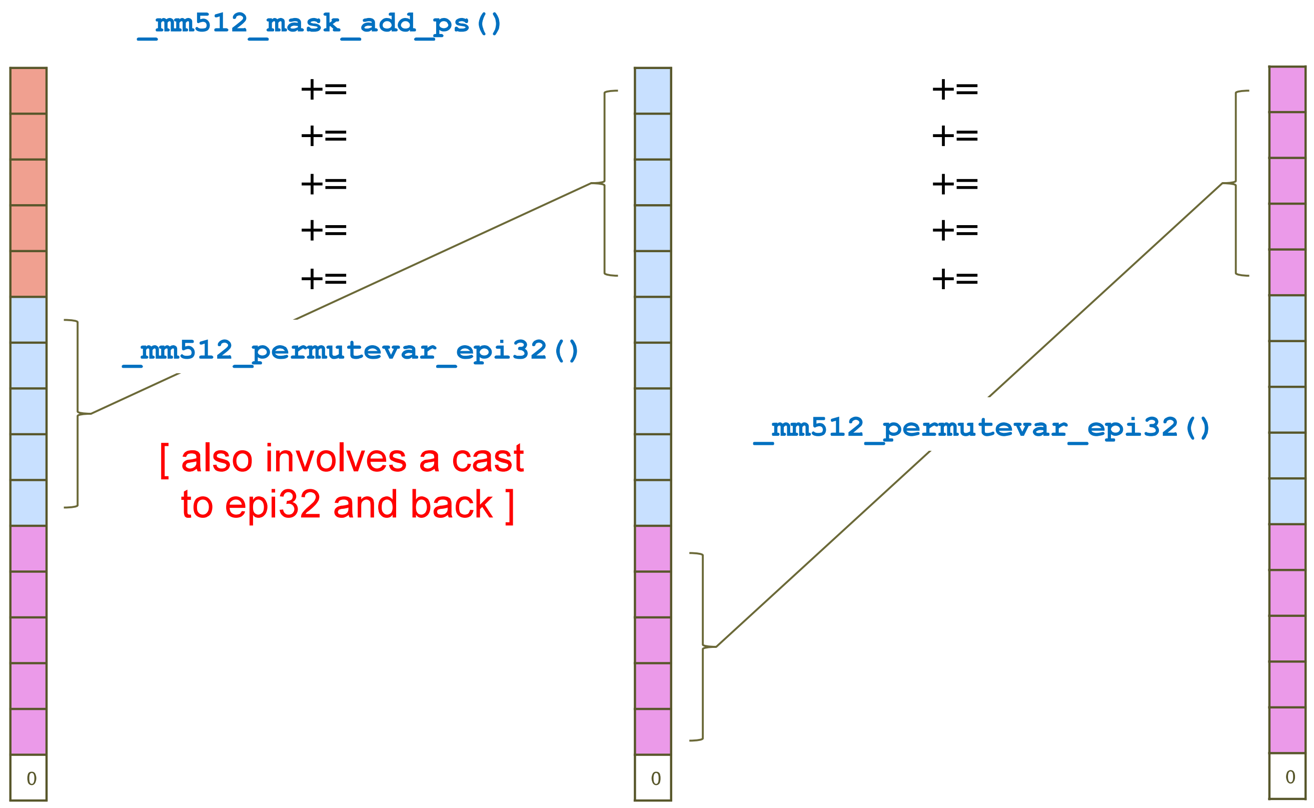
Solver: AVX-512 Implementation



Langley Research Center

Solver: AVX-512 Implementation

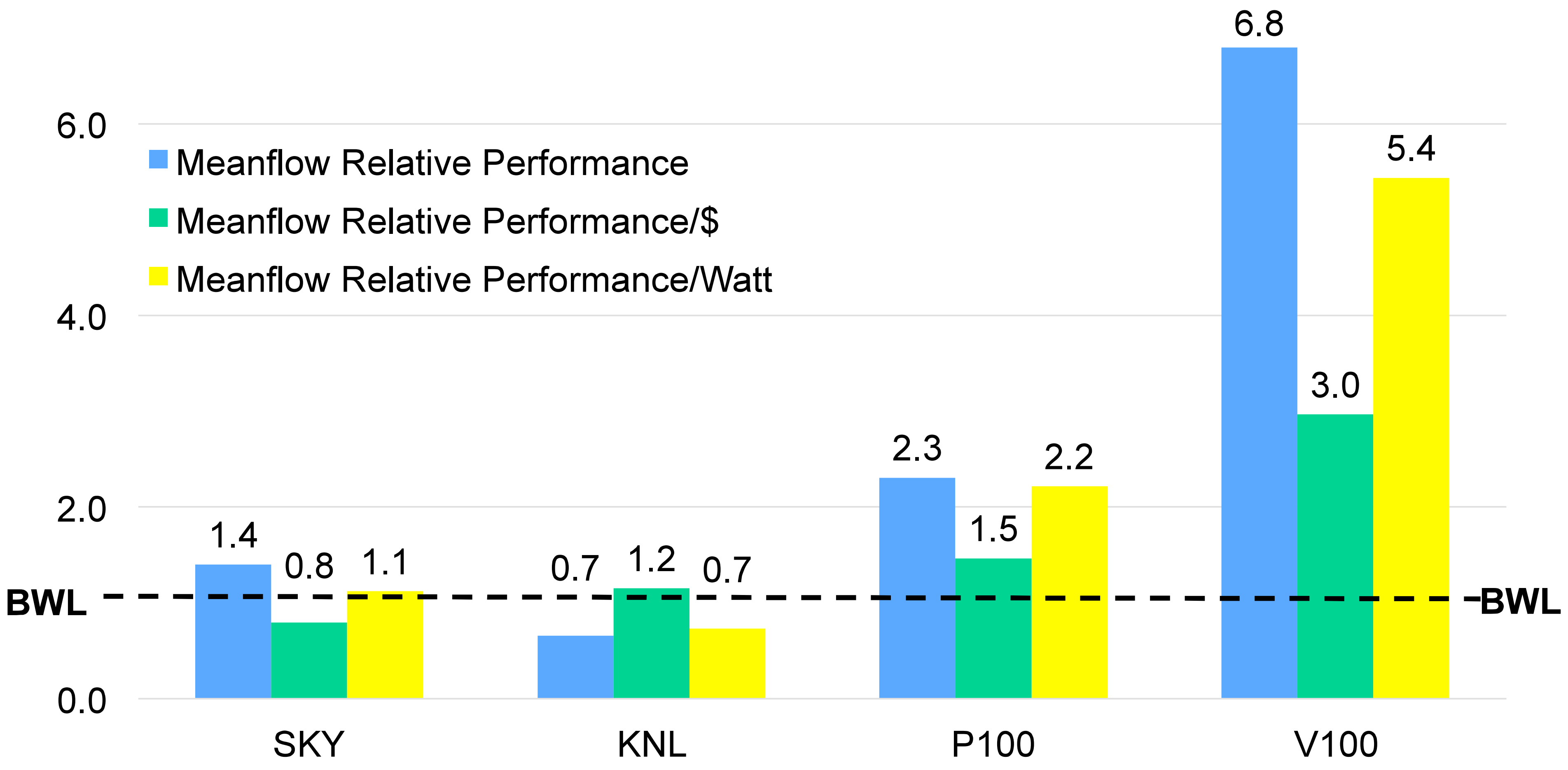
Add partial sum columns together using permutes to obtain final matvec result
(colors now only indicate a separation between 3 partial sums)



[also involves a cast
to epi32 and back]

Langley Research Center

Relative (to BWL MPI) Performance Results



Fortran Concerns

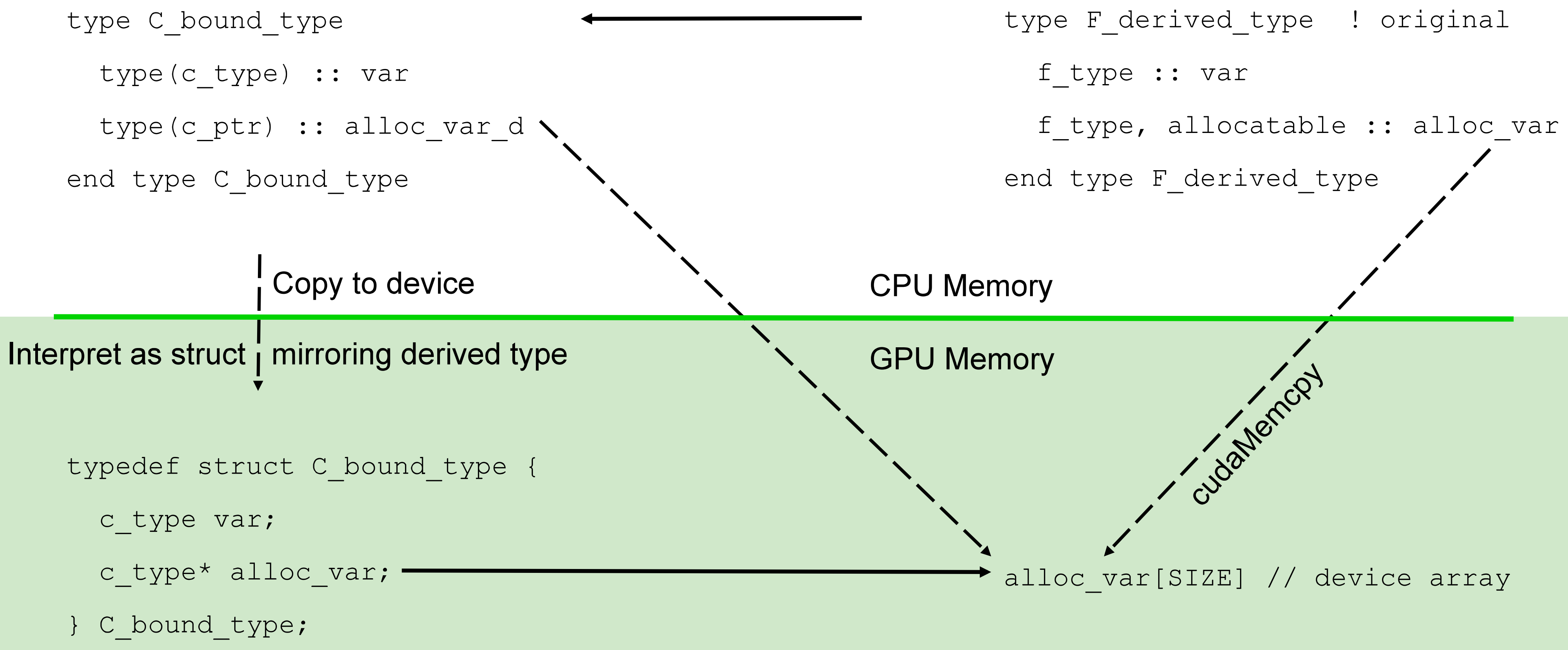
Results indicate that optimal many-core performance requires low-level programming in C/C++ (CUDA, AVX-512), this brings Fortran/C interoperability concerns:

- Passing derived types with allocatables (and arrays of such types)
- Fortran compiler inconsistencies (C_SIZEOF)
- Device pointer arithmetic (IBM XL Fortran compiler only?)
- SummitDev Spectrum MPI issues:
  - Support for accepting device ptr arguments from Fortran
  - Array of statuses
  - Request size in bytes (differs F2C in Spectrum MPI)

Langley Research Center

Fortran Concerns: Derived Types

Mirror original derived type with C-bound derived type



```
type C_bound_type
  type(c_type) :: var
  type(c_ptr) :: alloc_var_d
end type C_bound_type
```

```
type F_derived_type   ! original
  f_type :: var
  f_type, allocatable :: alloc_var
end type F_derived_type
```

```
typedef struct C_bound_type {
  c_type var;
  c_type* alloc_var;
} C_bound_type;
```

```
alloc_var[SIZE] // device array
```

We can use this method to access arrays of derived types with allocatables on the device

Strong Scaling Studies

Hybrid MPI+OpenMP

vs.

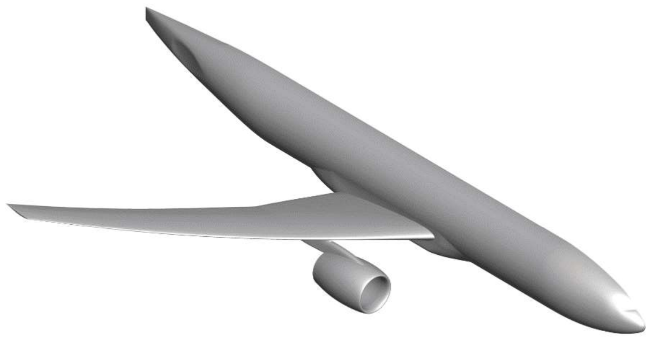
Pure MPI

(Xeon/Xeon Phi)

Langley Research Center

Input Grids

	A	B	C	D
Points	60,701,918	208,849,719	1,651,089,924	13,157,364,372
Tetrahedra	222,081,338	737,451,314	5,902,801,476	47,241,557,592
Pyramids	249,807	797,741	4,786,446	28,718,676
Prisms	44,585,182	163,786,283	1,310,290,264	10,482,322,112
File Size, GB	6.1	20.9	292.4	2,334.8



- 60M to 13.2B grid points, up to 57B elements
- Grids C and D derived from uniform grid refinement of Grid B
- All cases RANS with 1-equation turbulence model

Wing-body grids developed at NASA Langley for the Sixth AIAA Drag Prediction Workshop

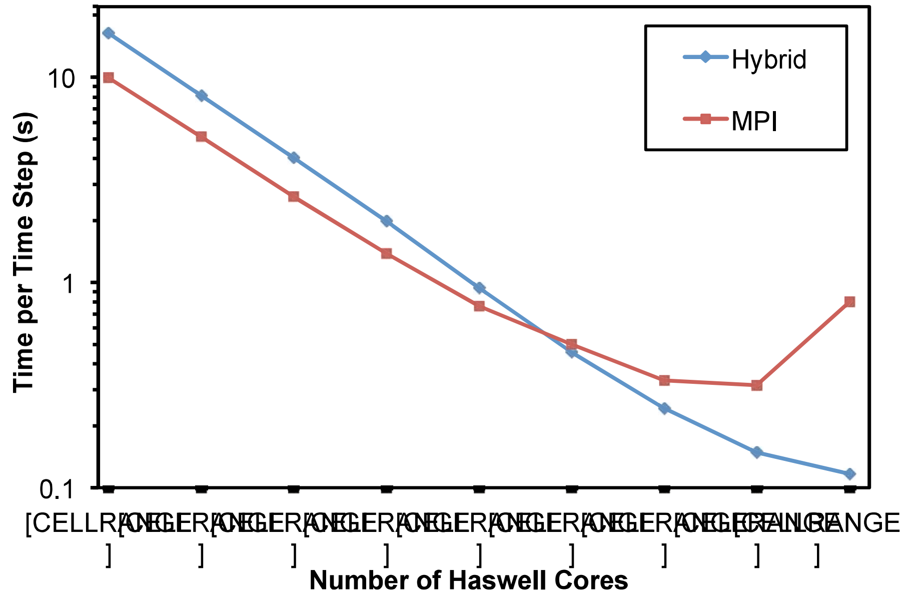
HPC Systems

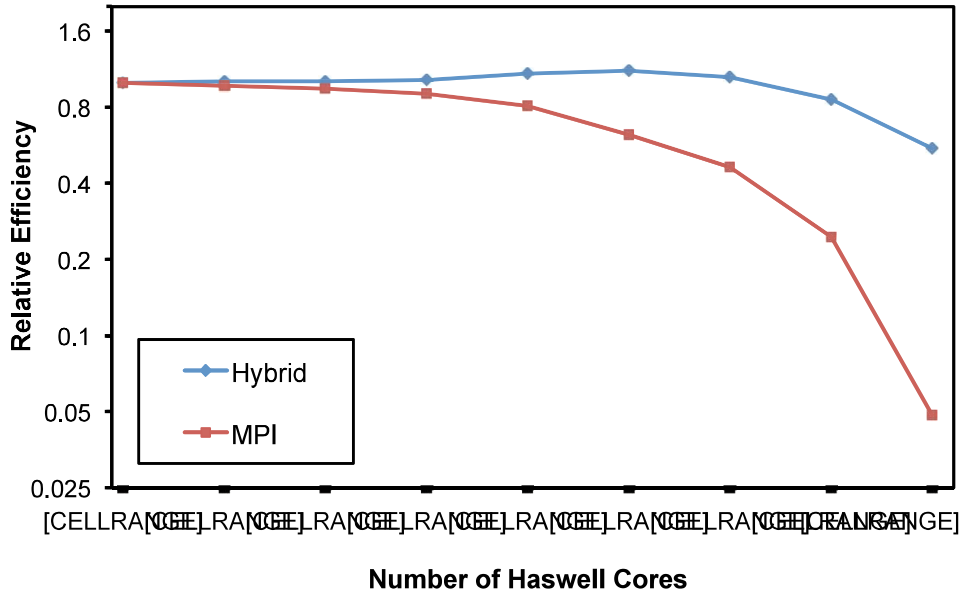
	Electra	Onyx	Topaz
System Type	SGI ICE X	Cray XC40/50	SGI ICE X
Processor	SKY	KNL	HWL
Nodes (of Processor)	1,152	544	3,456
NUMA × Cores × Threads	2×20×2	1×64×4	4×9×2
Cores (of Processor)	46,080	34,816	124,416
High-bandwidth Memory	0	16 GB	0
Interconnect	4× InfiniBand EDR	Cray Aries	4× InfiniBand FDR
Topology	Hypercube	Dragonfly	Hypercube
MPI	mpt.2.17r4	cray-mpich/7.6.2	mpt/2.13-1128
Compiler	Intel 18	Intel 17	Intel 16
Grids	B,C,D	A,B	A,C,D

- Hybrid uses 1 MPI rank per NUMA domain and one thread per physical processing element, pure MPI uses one rank per physical processing element
- Efficiency is calculated relative to an arbitrary core count's timing: $scaling{\downarrow}actual\ /\ scaling{\downarrow}perfect$

Langley Research Center

Grid A (60M) Results on Topaz (HWL)

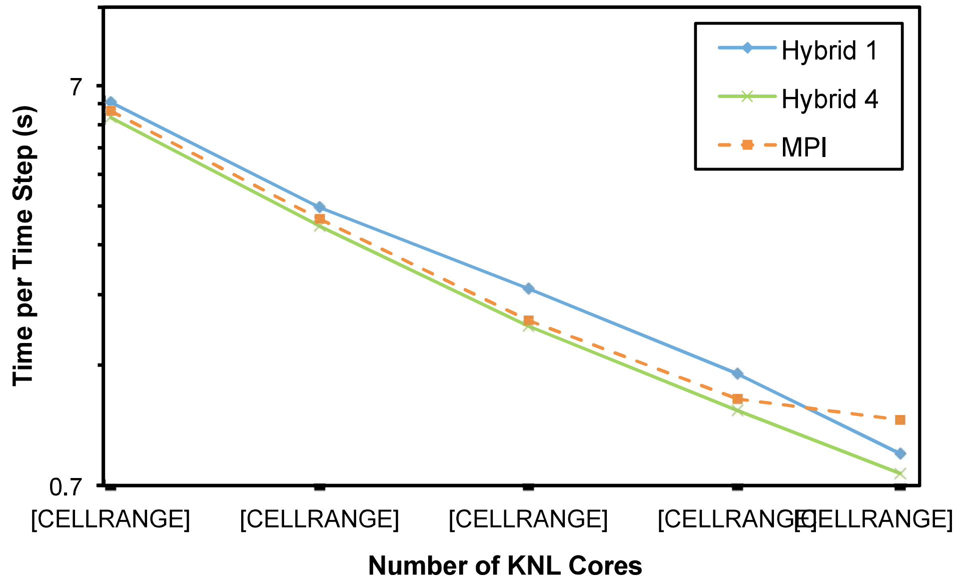


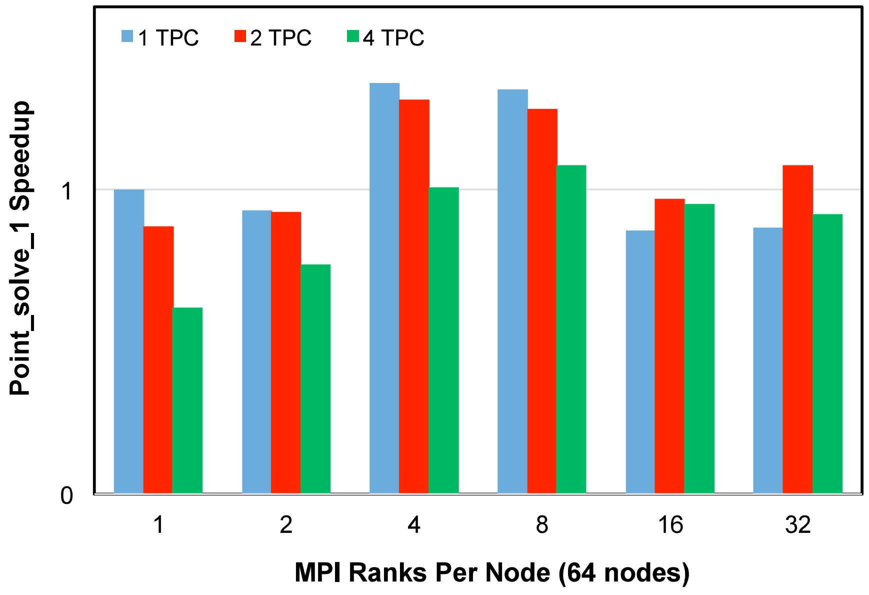


- MPI initially 1.6× faster than hybrid
- Crossover point at 6,912 cores (~8,700 grid points/core) where Hybrid faster
- Hybrid solver starts slower but faster by 6,912: need for threaded comm?

Langley Research Center

Grid A (60M) Results on Onyx (KNL)

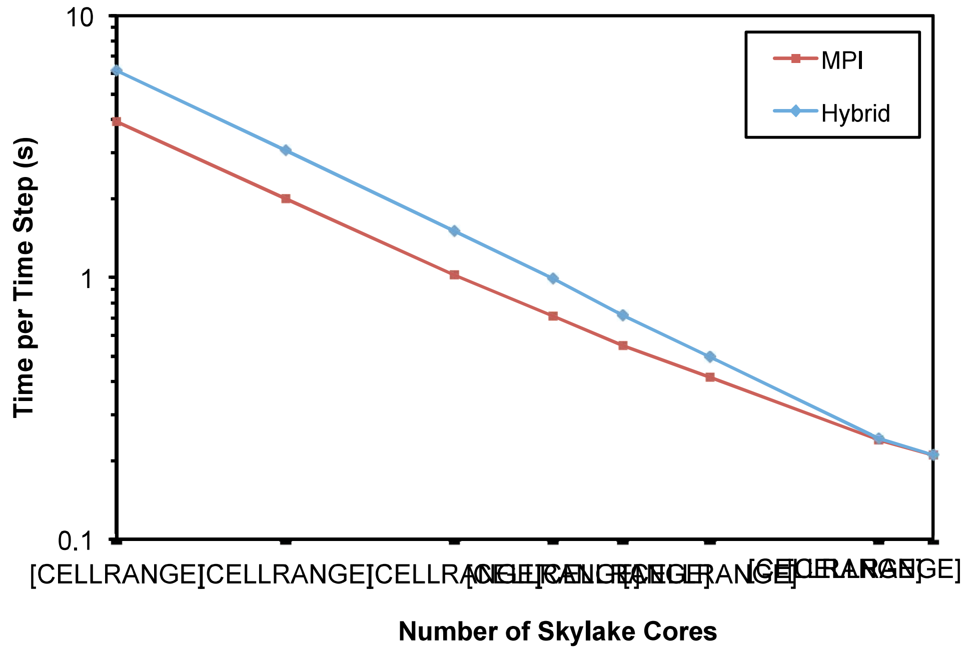


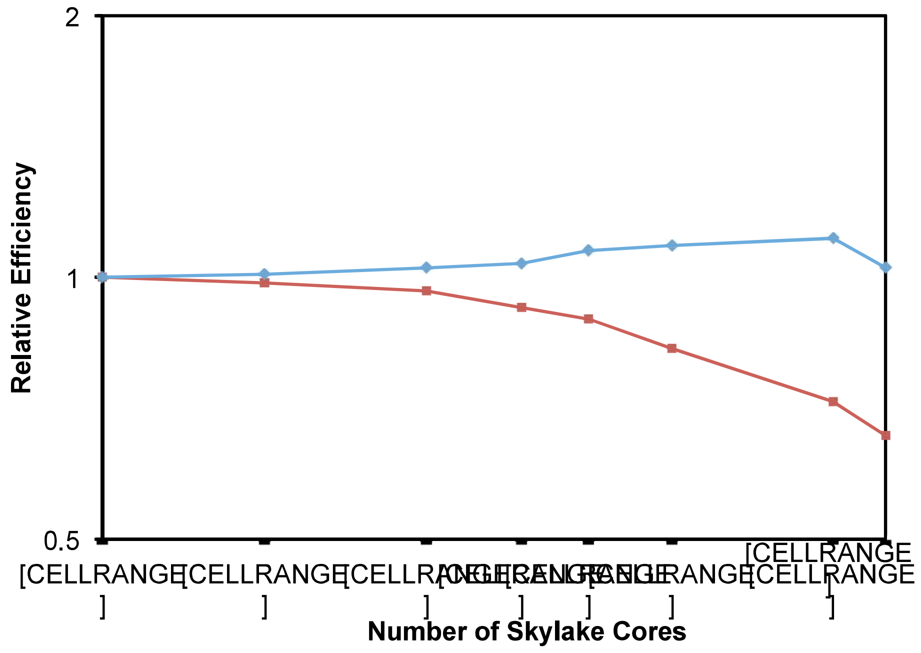


- Hybrid faster than MPI even with coloring to avoid data races
- Hybrid 20-30% benefit using 4 ranks/node over 1, why?
- Hyperthreading causes slowdown in solver only with off-node comm
- Hypothesis: MPI implementation is oversubscribing calling core during overlapped comm

Langley Research Center

Grid B (208M) Results on Electra

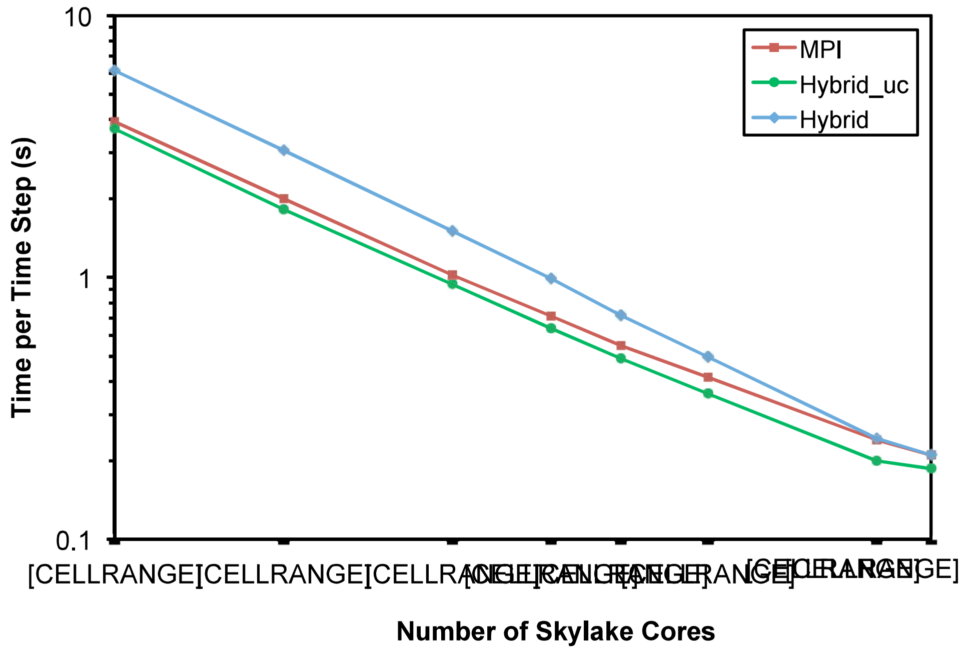


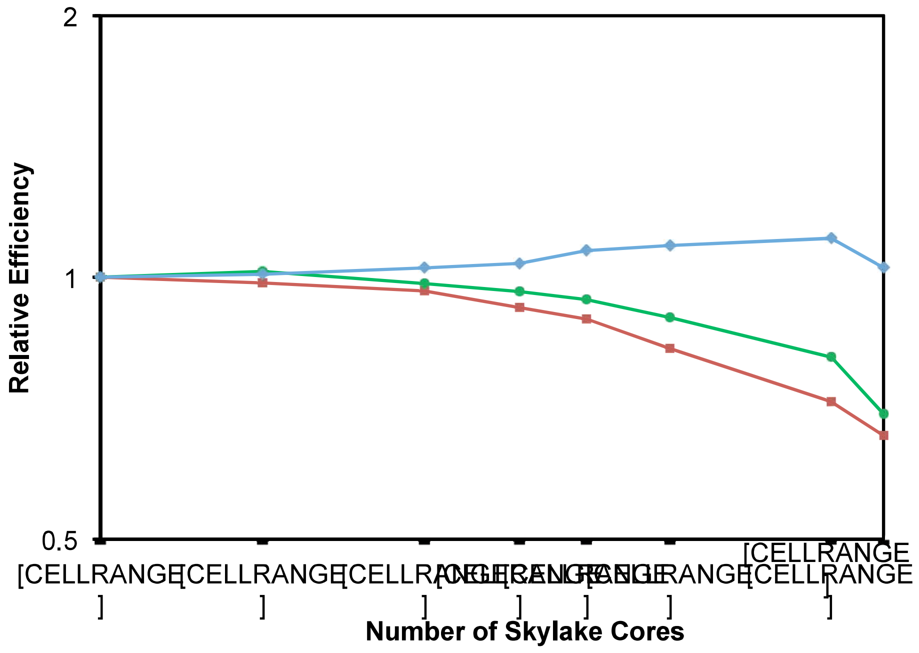


- Convergence at 40k, can hybrid ever run faster on Electra?
- Run Hybrid without coloring to test (incorrect physics)

Langley Research Center

Grid B (208M) Results on Electra

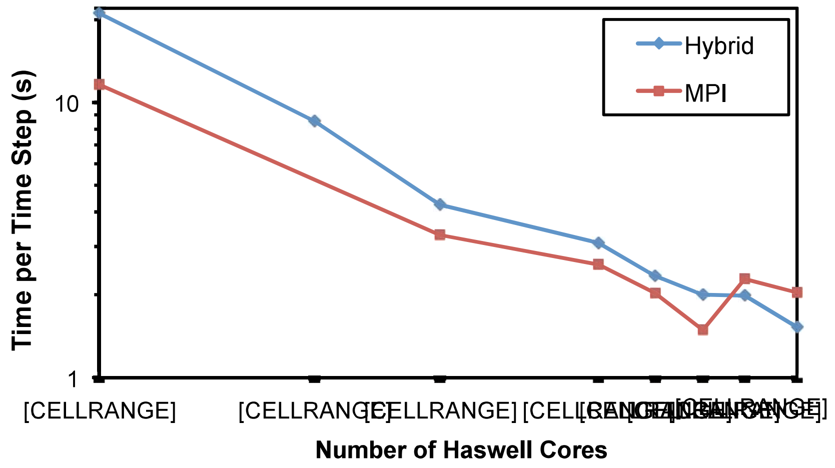


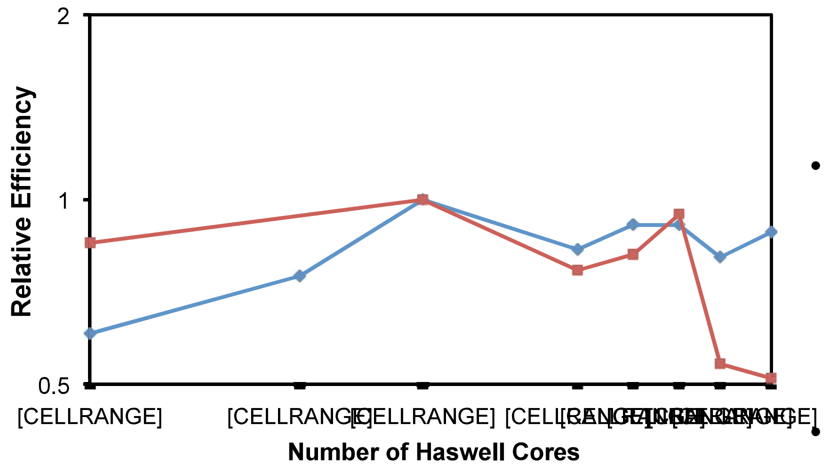


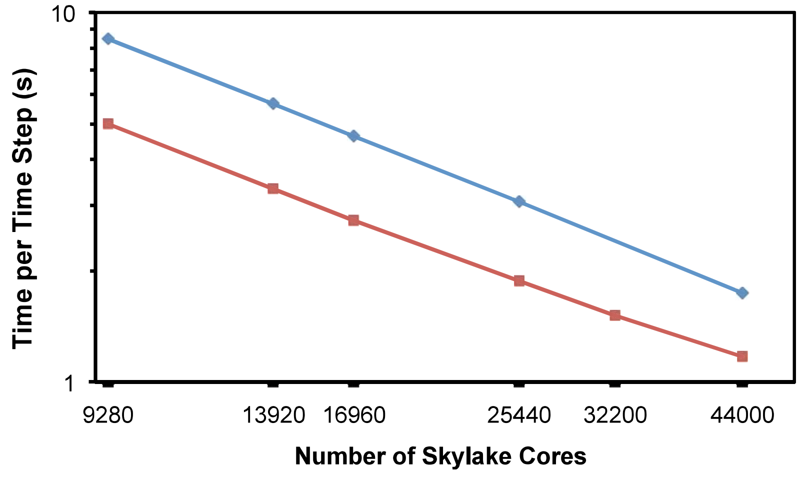
- Convergence at 40k, can hybrid ever run faster on Electra?
- Run Hybrid without coloring to test (incorrect physics), Hybrid_uc
- Answer: Yes, Hybrid_uc and Hybrid solvers 20% faster at 40k, coloring is the difference

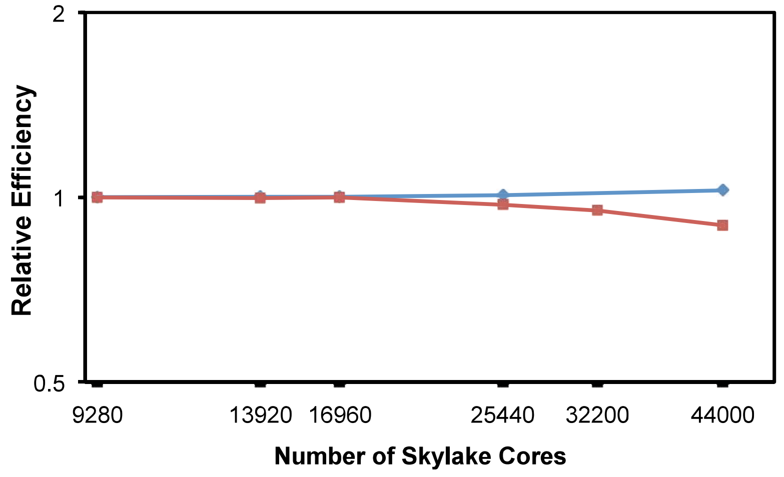
Langley Research Center

Grid C (1.65B) Results Topaz/Electra







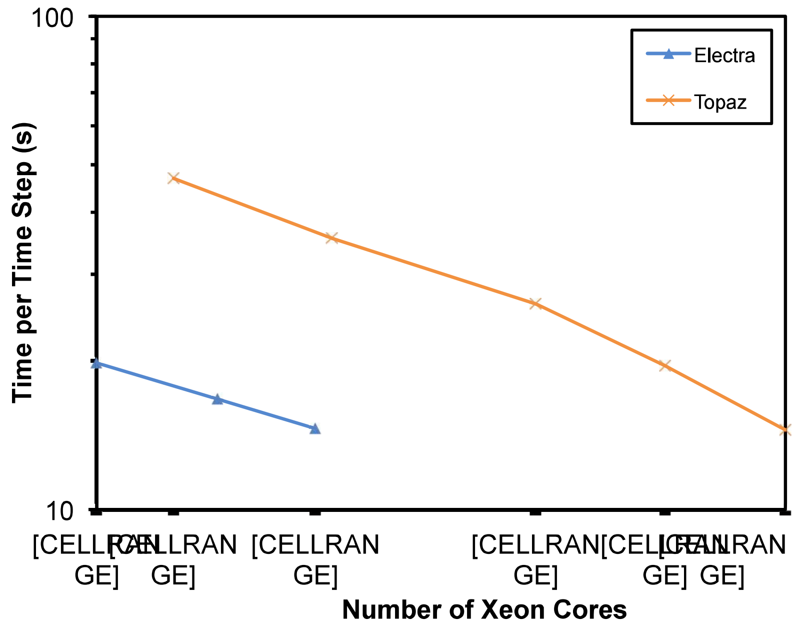


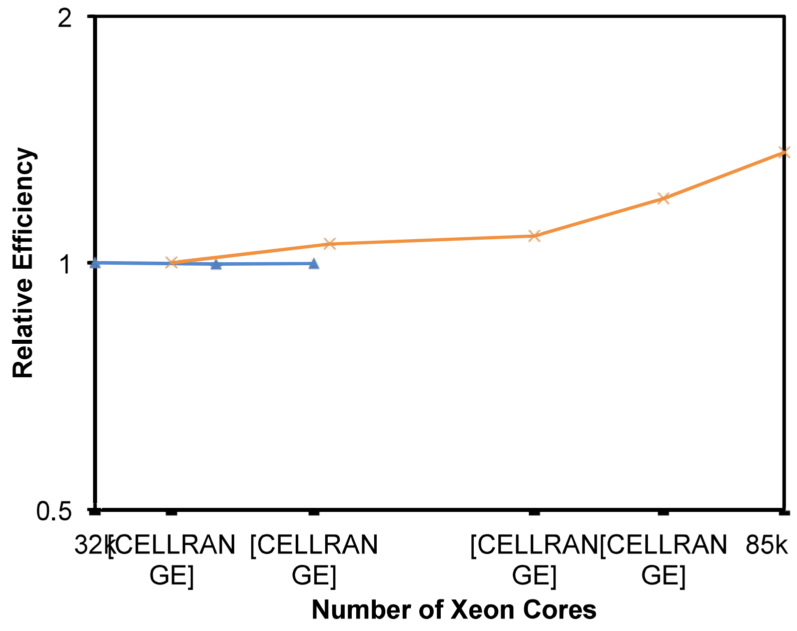
Top: Topaz

Bottom: Electra

- Up to 19k grid points/ node, too many for Hybrid to pass MPI, except for what looks like a degenerate case of MPI on Topaz
- Efficiency of scaling much improved at this level of work
- Topaz results more erratic
- Hybrid outscales MPI but does not **yet** catch up in absolute speed

Grid D (13.2B) Results Electra/Topaz

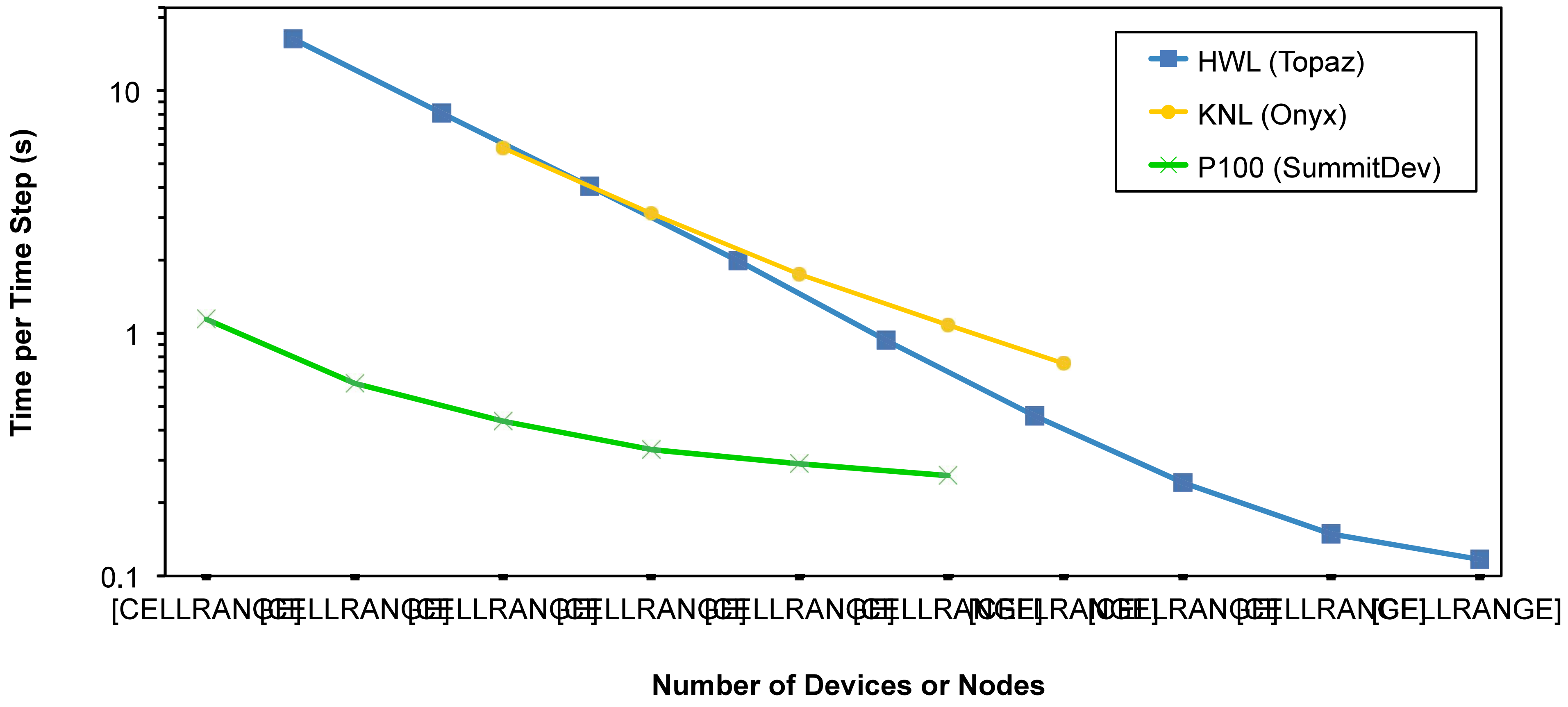




- Relative scaling of 100% or better continues as expected
- At this level of work (200× 60M), communication latency is easily hidden behind computation
- Topaz and Electra show the same linear (or better) trends in timing

Langley Research Center

60M Meanflow Strong Scaling Comparison



Conclusions and Future Work

Conclusions:

- KNL performance roughly equal to a 2-socket BWL, but more efficient
- GPUs’ hardware atomic support and flexible parallelism are extremely powerful
- GPUs’ fast solver outscaled by CPU/KNL, but same perf with many fewer nodes
- Many-core may require low-level programming (AVX-512, CUDA) for peak performance of complex code
- Translating a large code to CUDA requires a sizeable investment: what if equal effort were put into KNL-specific optimizations?
- Hybrid MPI+OpenMP has faster solver (ultimate scaling bottleneck) **at scale** despite slowdown caused by low ranks per node

Future Work:

- Solve the MPI + hyperthreads issue (aka low ranks per node issue)
- Explore alternatives to race condition avoidance (mimic MPI)
- Optimize vectorization/prefetching of key routines for AVX-512 devices
- CUDA turbulence models (coming soon to NVIDIA GTC ‘18 and ParCFD ‘18)

Appendix