



# Resource-Efficient, Hierarchical Auto-tuning of a Hybrid Lattice Boltzmann Computation on the Cray XT4

**Samuel Williams**, Jonathan Carter,  
Leonid Oliker, John Shalf, Katherine Yelick

Lawrence Berkeley National Laboratory (LBNL)  
National Energy Research Scientific Computing Center (NERSC)

*SWWilliams@lbl.gov*



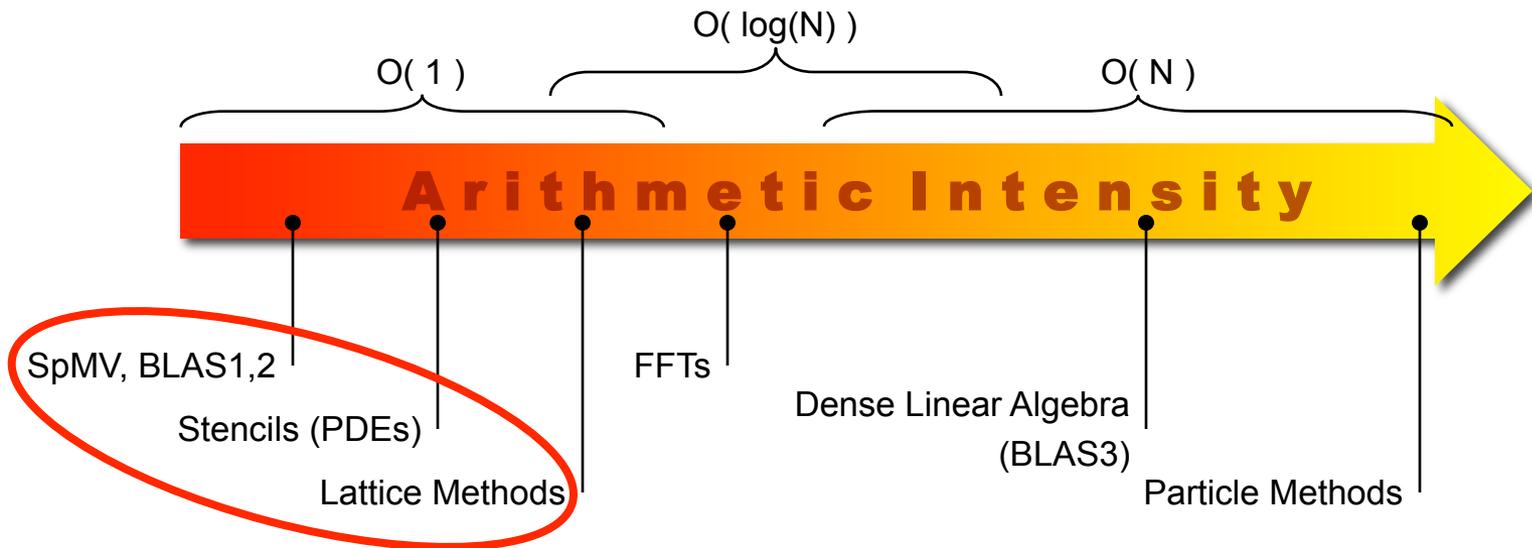
# Outline

F U T U R E   T E C H N O L O G I E S   G R O U P

1. Background
2. LBMHD
3. Previous work: Auto-tuning LMBHD on Multicore SMPs
4. Hybrid MPI-Pthreads implementations
5. Distributed, Hybrid LBMHD Auto-tuning
6. Results
7. Summary



# Background



- ❖ **True Arithmetic Intensity (AI) ~ Total Flops / Total DRAM Bytes**
- ❖ Some HPC kernels have an arithmetic intensity that scales with problem size (increased temporal locality), but remains constant on others
- ❖ Arithmetic intensity is ultimately limited by compulsory traffic
- ❖ Arithmetic intensity is diminished by conflict or capacity misses.



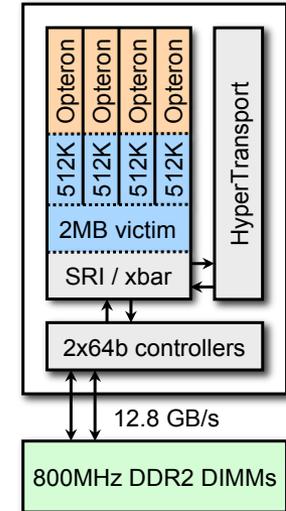
# Kernel Arithmetic Intensity and Architecture

FUTURE TECHNOLOGIES GROUP

- ❖ For a given architecture, one may calculate its flop:byte ratio.
- ❖ For a 2.3GHz Quad Core Opteron (like in the XT4),
  - 1 SIMD add + 1 SIMD multiply per cycle per core
  - 12.8GB/s of DRAM bandwidth
  - =  $36.8 / 12.8 \sim 2.9$  flops per byte

- ❖ When a kernel's arithmetic intensity is substantially less than the architecture's flop:byte ratio, transferring data will take longer than computing on it  
→ **memory-bound**

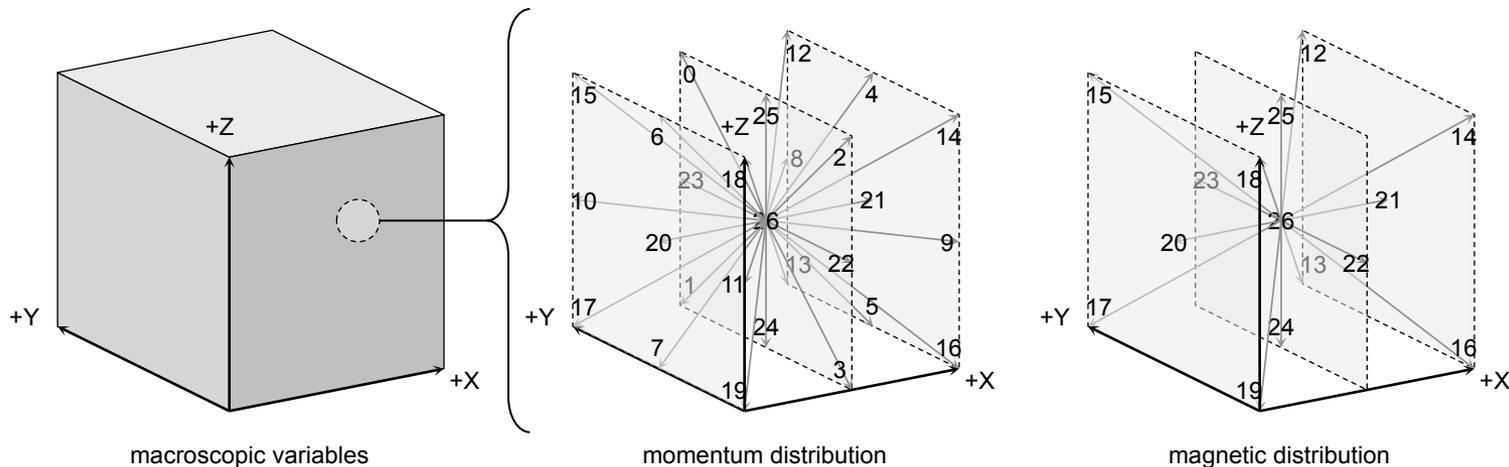
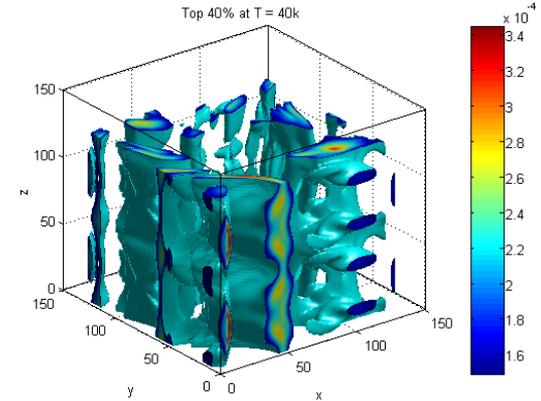
- ❖ When a kernel's arithmetic intensity is substantially greater than the architecture's flop:byte ratio, computation will take longer than data transfers  
→ **compute-bound**





# LBMHD

- ❖ Plasma turbulence simulation via Lattice Boltzmann Method
- ❖ Two distributions:
  - momentum distribution (27 scalar components)
  - magnetic distribution (15 vector components)
- ❖ Three macroscopic quantities:
  - Density
  - Momentum (vector)
  - Magnetic Field (vector)





- ❖ Code Structure
  - time evolution through a series of *collision()* and *stream()* functions
- ❖ When parallelized, *stream()* should constitute 10% of the runtime.
- ❖ *collision()*'s Arithmetic Intensity:
  - Must read 73 doubles, and update 79 doubles per lattice update (1216 bytes)
  - Requires about 1300 floating point operations per lattice update
  - **Just over 1.0 flops/byte (ideal architecture)**
  - Suggests LBMHD is **memory-bound** on the XT4.
- ❖ Structure-of-arrays layout (component's are separated) ensures that cache capacity requirements are independent of problem size
- ❖ However, TLB capacity requirement increases to >150 entries
- ❖ periodic boundary conditions



# Previous Work: Auto-tuning LBMHD on Multicore SMPs

Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, Katherine Yelick,  
"Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms",  
International Parallel & Distributed Processing Symposium (IPDPS), 2008.

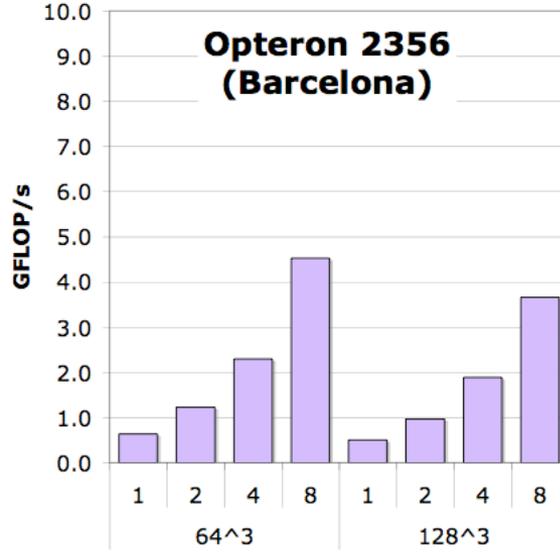
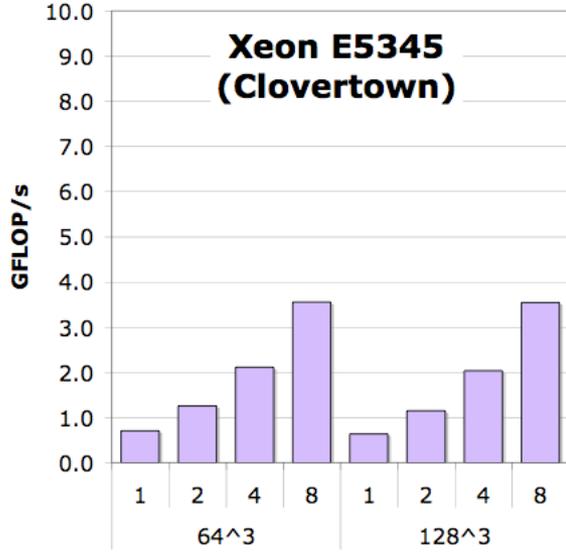
**Best Paper, Application Track**



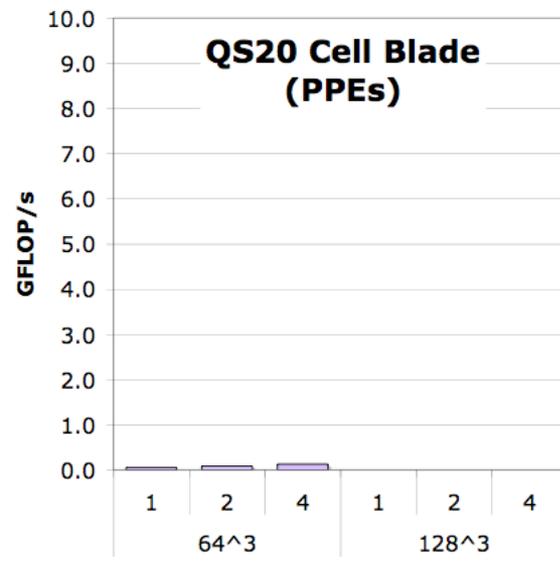
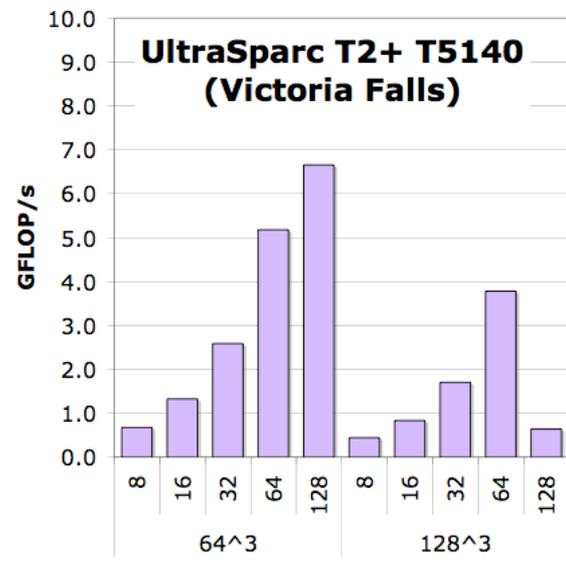
# LBMHD Performance

(reference implementation)

FUTURE TECHNOLOGIES GROUP



- ❖ Generally, scalability looks good
- ❖ Scalability is good
- ❖ but is performance good?



Naïve+NUMA



# Lattice-Aware Padding

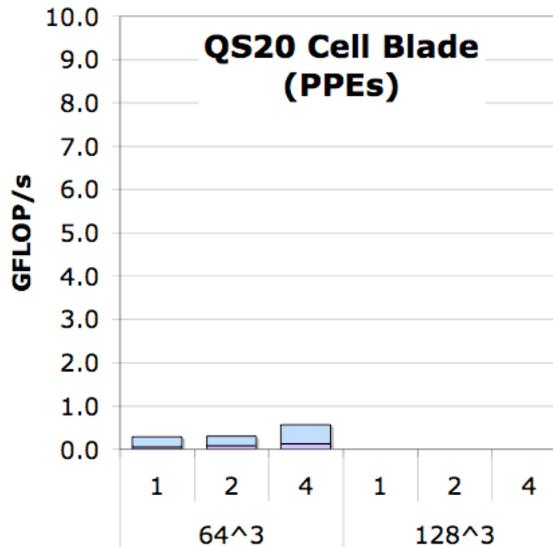
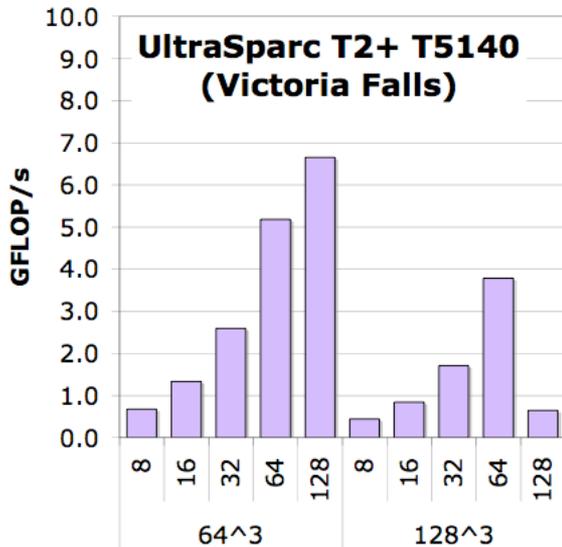
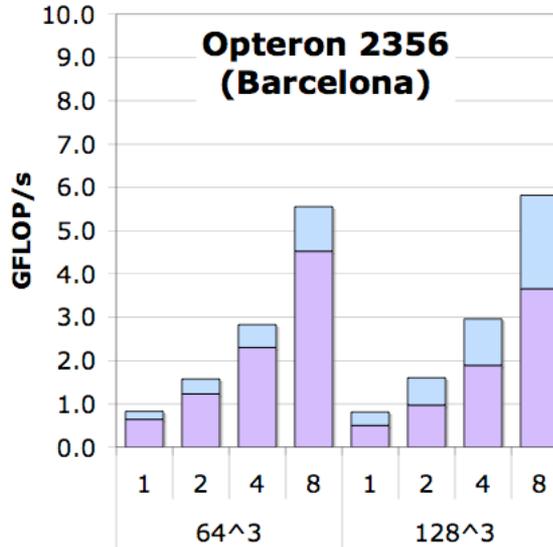
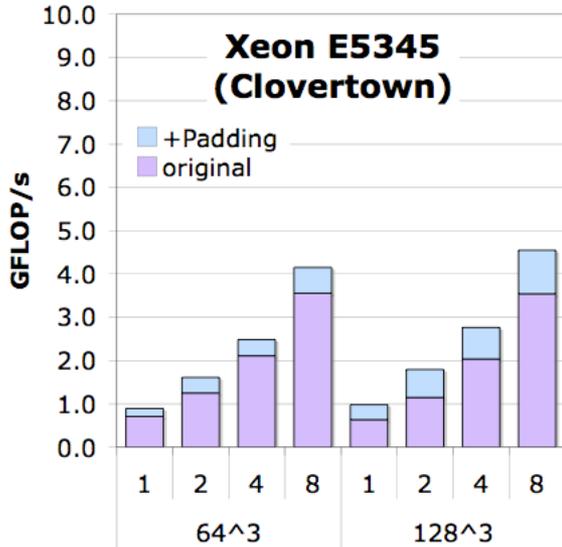
F U T U R E   T E C H N O L O G I E S   G R O U P

- ❖ For a given lattice update, the requisite velocities can be mapped to a relatively narrow range of cache sets (lines).
- ❖ As one streams through the grid, one cannot fully exploit the capacity of the cache as conflict misses evict entire lines.
- ❖ In an structure-of-arrays format, pad each component such that when referenced with the relevant offsets ( $\pm x, \pm y, \pm z$ ) they are uniformly distributed throughout the sets of the cache
- ❖ Maximizes cache utilization and minimizes conflict misses.

# LBMHD Performance

(lattice-aware array padding)

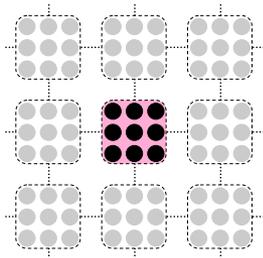
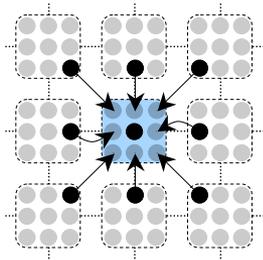
FUTURE TECHNOLOGIES GROUP



- ❖ LBMHD touches >150 arrays.
- ❖ Most caches have limited associativity
- ❖ Conflict misses are likely
- ❖ Apply **heuristic** to pad arrays

+Padding  
Naïve+NUMA

- ❖ Two phases with a lattice method's collision() operator:
  - reconstruction of macroscopic variables
  - updating discretized velocities
- ❖ Normally this is done one point at a time.
- ❖ Change to do a vector's worth at a time (loop interchange + tuning)

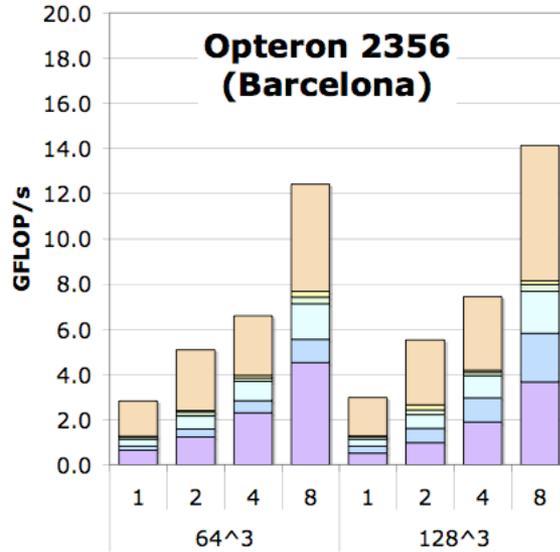
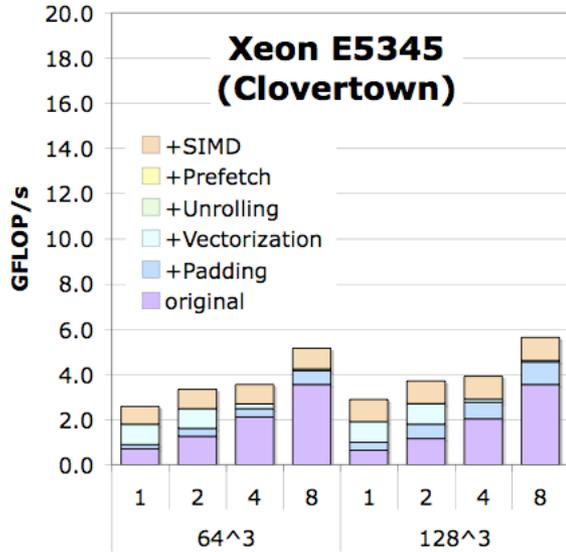




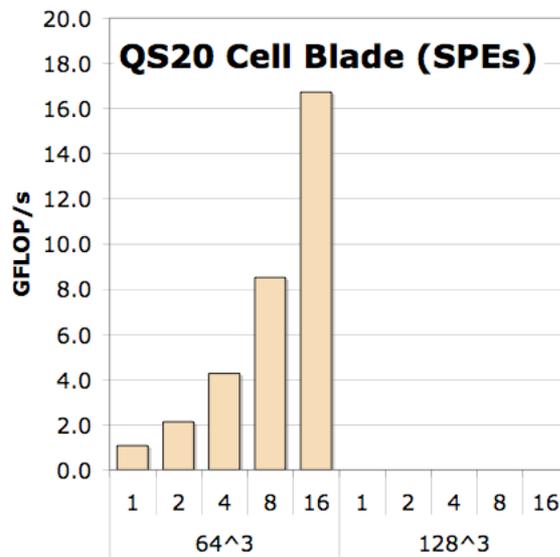
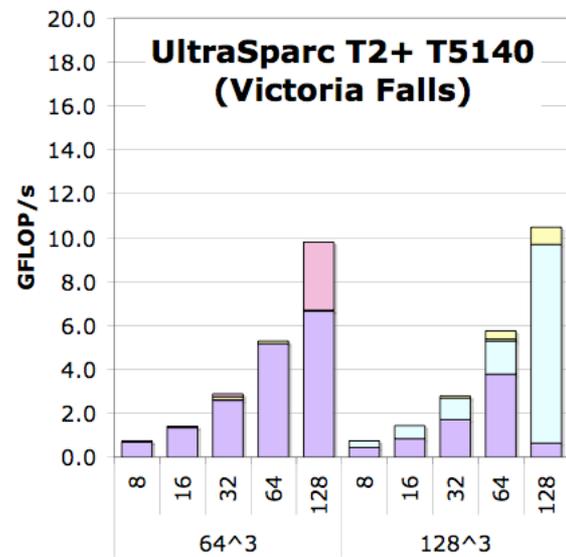
# LBMHD Performance

(architecture specific optimizations)

FUTURE TECHNOLOGIES GROUP



- ❖ Add unrolling and reordering of inner loop
- ❖ Additionally, it exploits SIMD where the compiler doesn't
- ❖ Include a SPE/Local Store optimized version



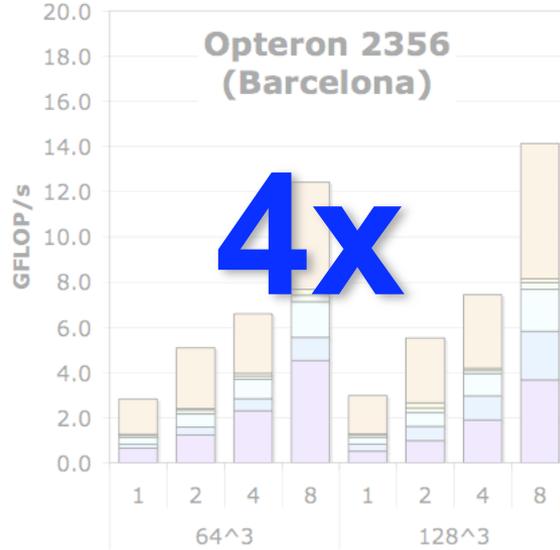
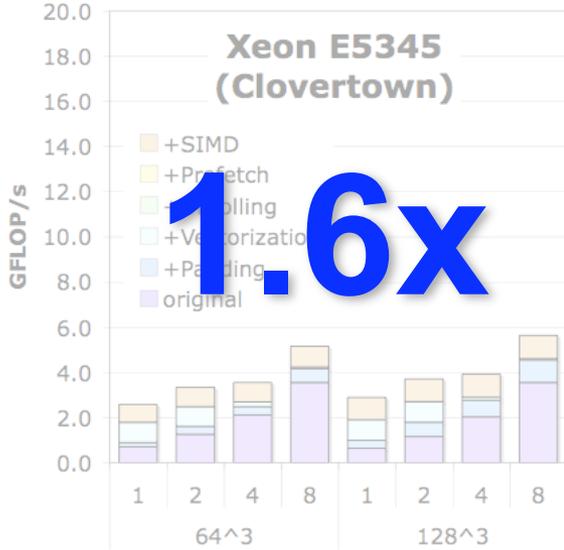
- +small pages
- +Explicit SIMDization
- +SW Prefetching
- +Unrolling
- +Vectorization
- +Padding
- Naive+NUMA



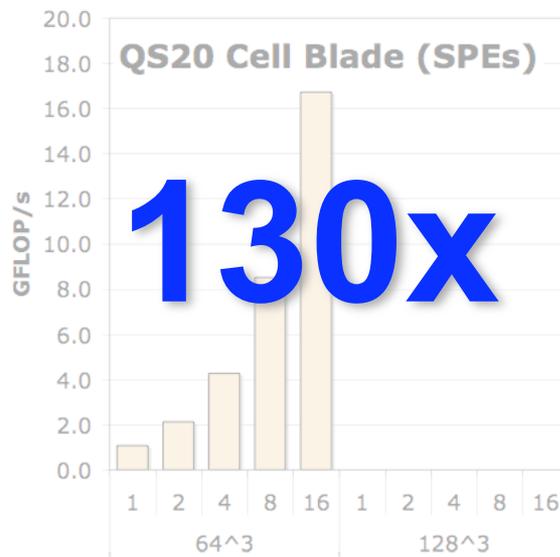
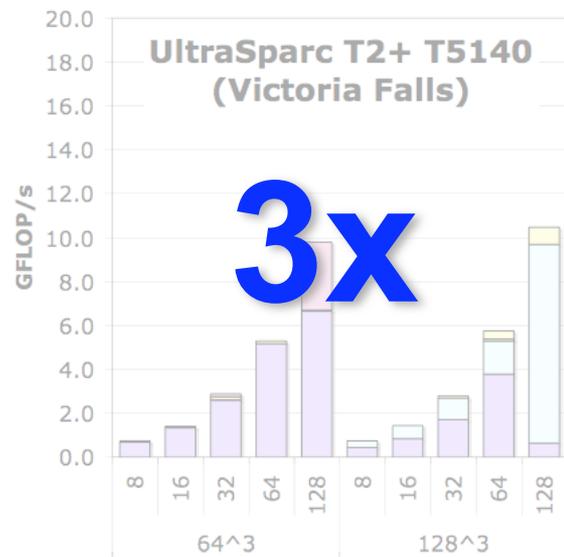
# LBMHD Performance

(architecture specific optimizations)

FUTURE TECHNOLOGIES GROUP



- ❖ Add unrolling and reordering of inner loop
- ❖ Additionally, it exploits SIMD where the compiler doesn't
- ❖ Include a SPE/Local Store optimized version



- +small pages
- +Explicit SIMDization
- +SW Prefetching
- +Unrolling
- +Vectorization
- +Padding
- Naïve+NUMA



# Limitations

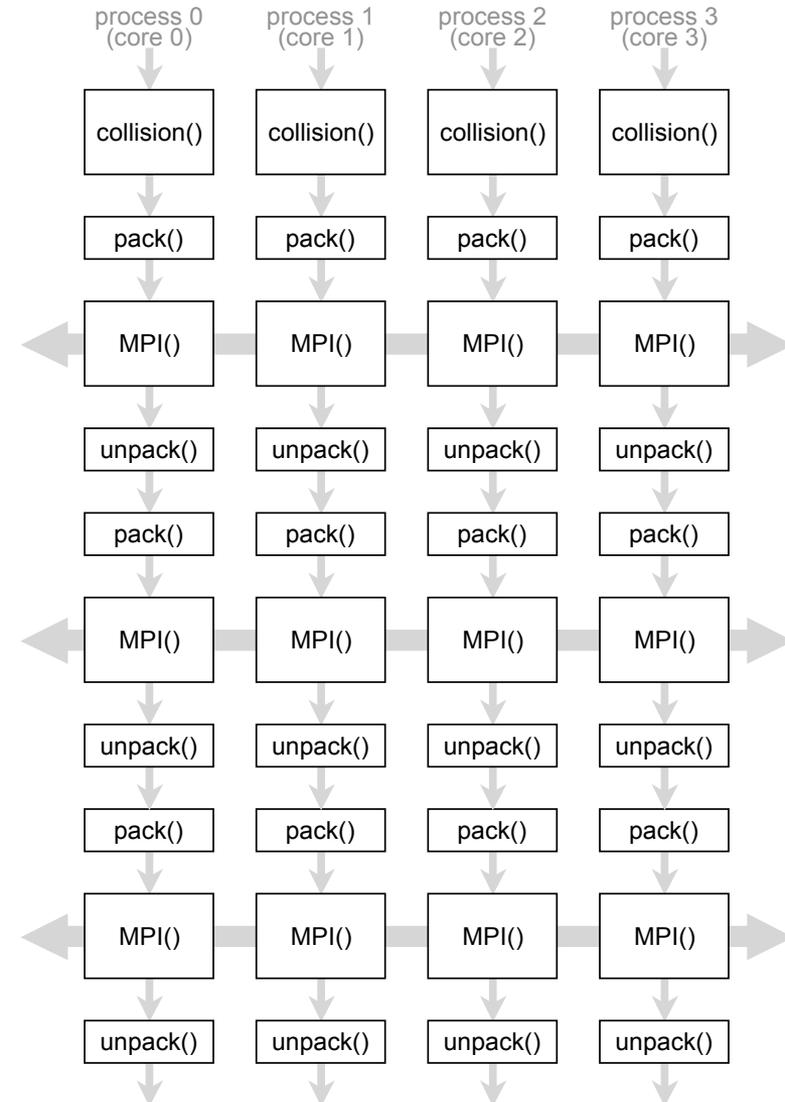
F U T U R E   T E C H N O L O G I E S   G R O U P

- ❖ Ignored MPP (distributed) world
- ❖ Kept problem size fixed and cubical
- ❖ When run with only 1 process per SMP, maximizing threads per process always looked best



# Hybrid MPI+Pthreads Implementations

- ❖ In the flat MPI world, there is one process per core, and only one thread per process
- ❖ All communication is through MPI







# Distributed, Hybrid Auto-tuning



# The Distributed Auto-tuning Problem

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ We believe that even for relatively large problems, only auto-tuning the local computation (e.g. IPDPS'08) will deliver sub-optimal MPI performance.
- ❖ We have a combinatoric explosion in the search space coupled with a large problem size (number of nodes)

at each concurrency:

for all aspect ratios

for all process/thread balances

for all thread grids

for all data structures

for all coding styles (reference, vectorized, vectorized+SIMDized)

for all prefetching

for all vector lengths

for all code unrollings/reorderings

benchmark



# Our Approach

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ We employ a resource-efficient 3-stage greedy algorithm that successively prunes the search space:

- for all data structures

- for all coding styles (reference, vectorized, vectorized+SIMDized)

- for all prefetching

- for all vector lengths

- for all code unrollings/reorderings

- benchmark

- at limited concurrency (single node):

- for all aspect ratios

- for all process/thread balances

- for all thread grids

- benchmark

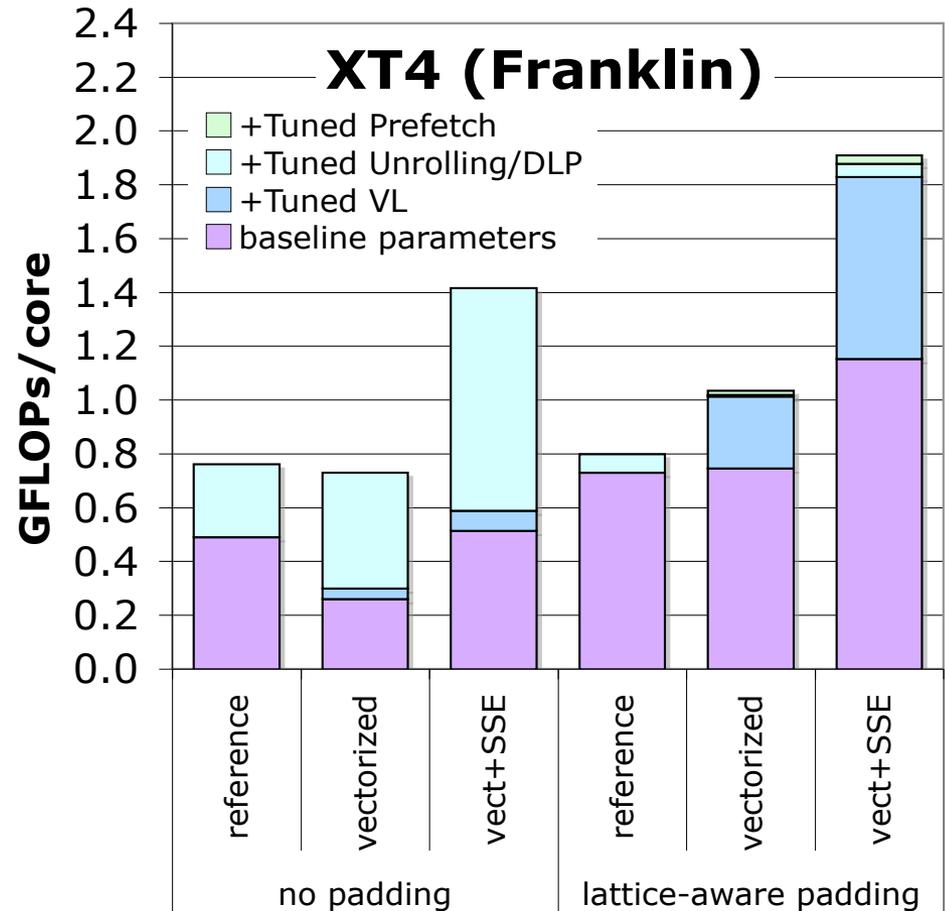
- at full concurrency:

- for all process/thread balances

- benchmark

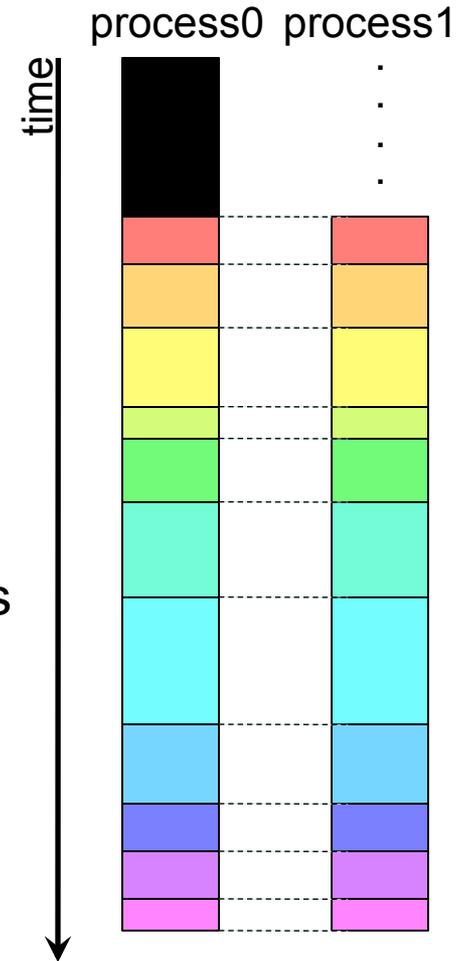
# Stage 1

- ❖ In stage 1, we prune the code generation space.
- ❖ We ran this as a  $128^3$  problem with 4 threads.
- ❖ As VL, unrolling, and reordering may be problem dependent, we only prune:
  - padding
  - coding style
  - prefetch distance
- ❖ We observe that vectorization with SIMDization, and a prefetch distance of 1 cache line worked best



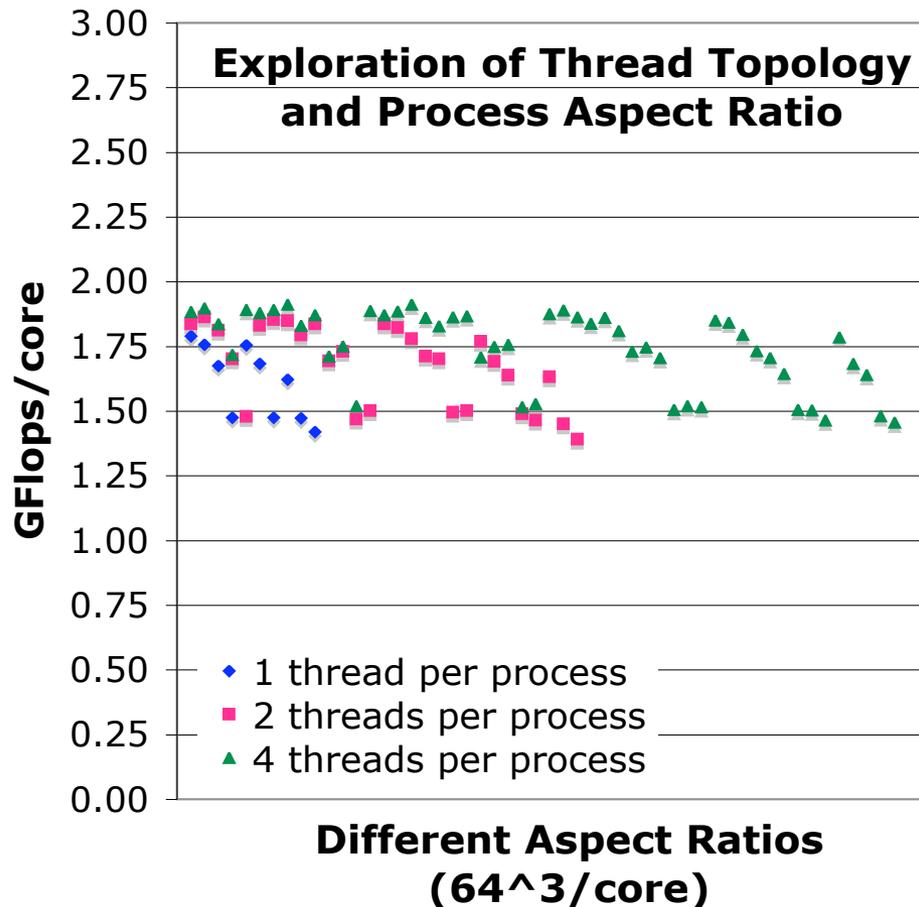
# Stage 2

- ❖ Suppose we wish to explore this color-coded optimization space.
- ❖ In the serial world (or fully threaded nodes), the tuning is easily run
- ❖ However, in the MPI or hybrid world a problem arises as processes are not guaranteed to be synchronized.
- ❖ As such, one process may execute some optimizations faster than others simply due to fortuitous scheduling with another processes' trials
- ❖ Solution: add an `MPI_barrier()` around each trial



# Stage 2 (continued)

- ❖ We create a database of optimal VL/unrolling/DLP parameters for each thread/process balance, thread grid, and aspect ratio



# Stage 3

- ❖ Given the data base from Stage 2,
- ❖ we run few large problem using the best known parameters/thread grid for different thread/process balances.
  
- ❖ We select the parameters based on minimizing
  - overall local time
  - *collision( )* time
  - local *stream( )* time



# Results

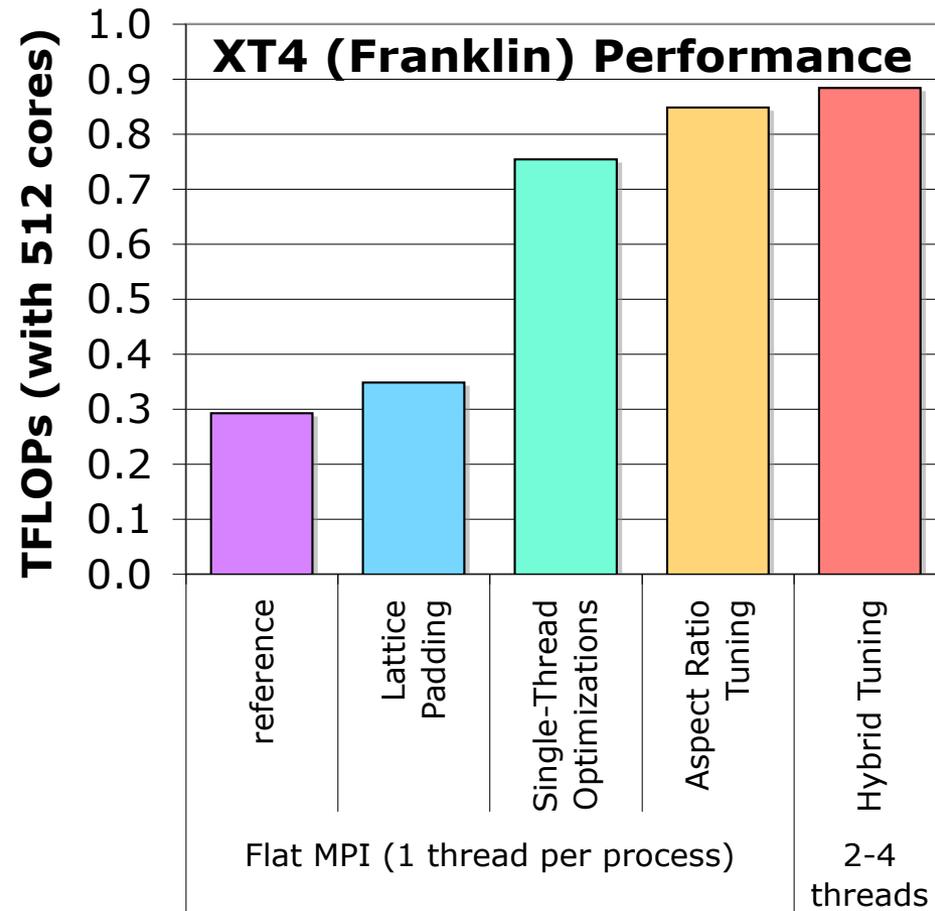


# XT4 Results

( $512^3$  problem on 512 cores)

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ Finally, we present the best data for progressively more aggressive auto-tuning efforts/
- ❖ Note each of the last 3 bars may have unique MPI decompositions as well as VL/unroll/DLP
- ❖ Observe that for this large problem, auto-tuning flat MPI delivered significant boosts (2.5x)
- ❖ However, expanding auto-tuning to include the domain decomposition and balance between threads and processes provided an extra 17%
- ❖ 2 processes with 2 threads was best

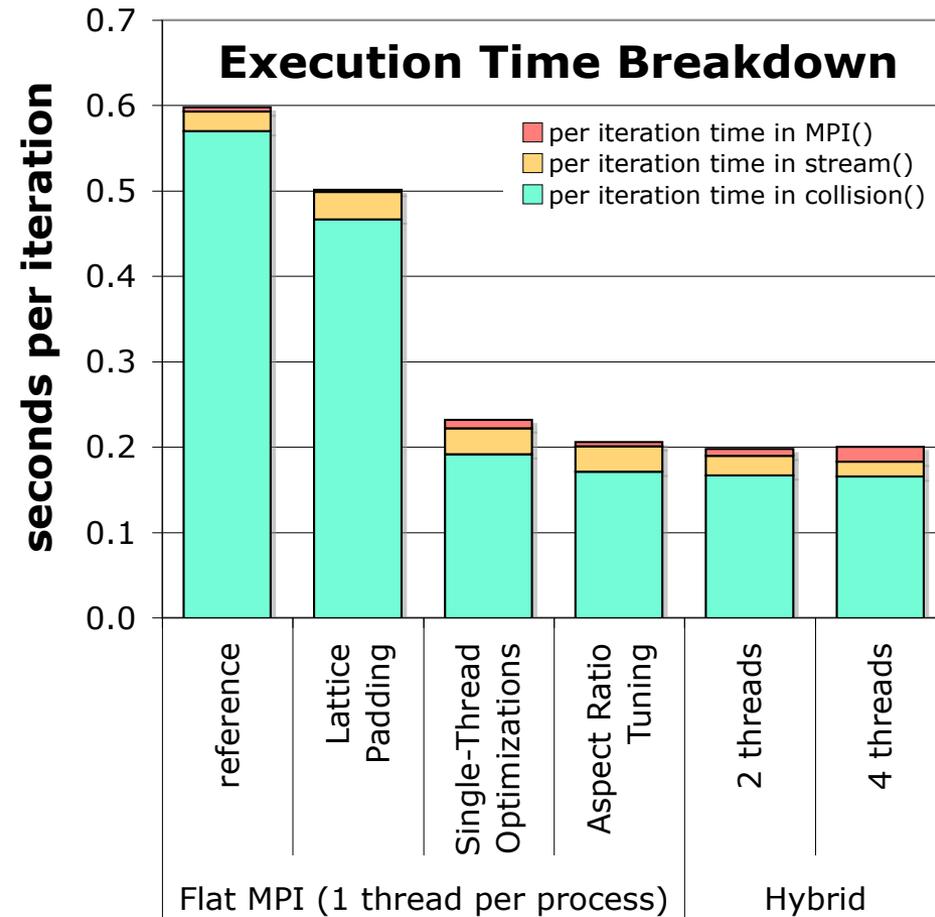


# XT4 Results

(512<sup>3</sup> problem on 512 cores)

F U T U R E T E C H N O L O G I E S G R O U P

- ❖ When examining the execution time breakdown, we see how the auto-tuner consistently favored faster *collision()* times.
- ❖ In the hybrid world we see a trade off between MPI time and stream() time
- ❖ As threads/process increase, we get less bandwidth, but less traffic.





# Summary



# Conclusions

F U T U R E   T E C H N O L O G I E S   G R O U P

- ❖ Multicore cognizant auto-tuning dramatically improves even flat MPI performance.
- ❖ Tuning the domain decomposition and hybrid implementations yielded almost an additional 20% performance boost.
- ❖ Although hybrid MPI promises improved performance through reduced communication, it is critical all components be thread-parallelized.
  
- ❖ Future work will move to other architectures (like the XT5), examine a range of problem sizes, and attempt to thread-parallelize the MPI.



# Acknowledgements

F U T U R E   T E C H N O L O G I E S   G R O U P

- ❖ Research supported by DOE Office of Science under contract number DE-AC02-05CH11231
- ❖ All XT4 simulations were performed on the XT4 (Franklin) at the National Energy Research Scientific Computing Center (NERSC)
- ❖ George Vahala and his research group provided the original (FORTRAN) version of the LBMHD code.



# Questions?