

Auto-tuning Performance on Multicore Computers

Samuel Williams

David Patterson, advisor and chair

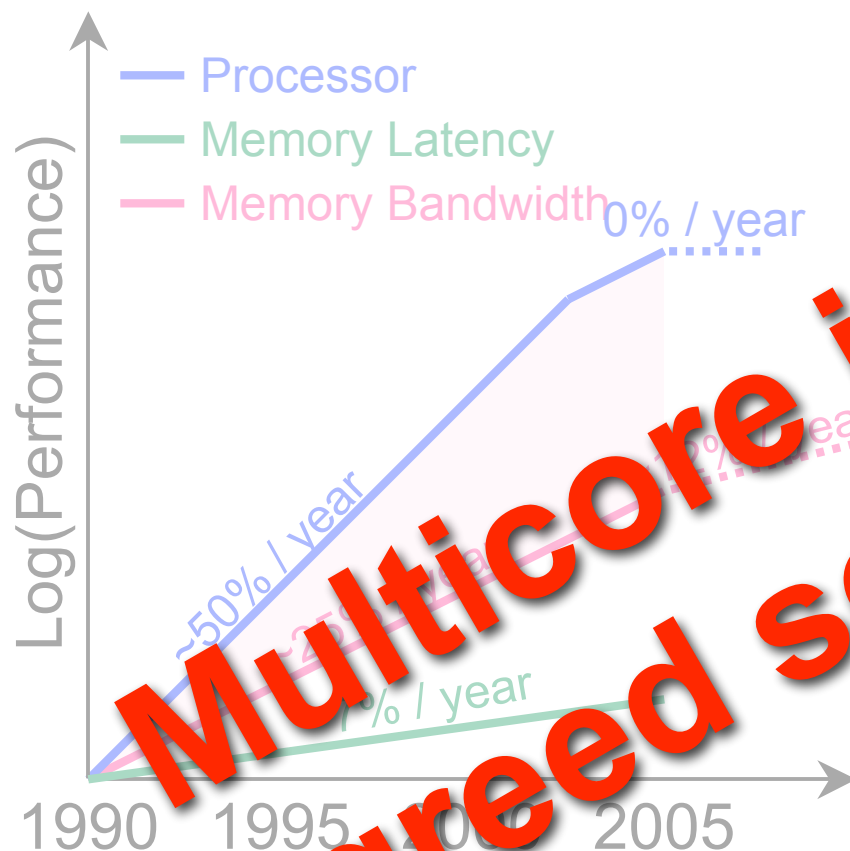
Kathy Yelick

Sara McMains

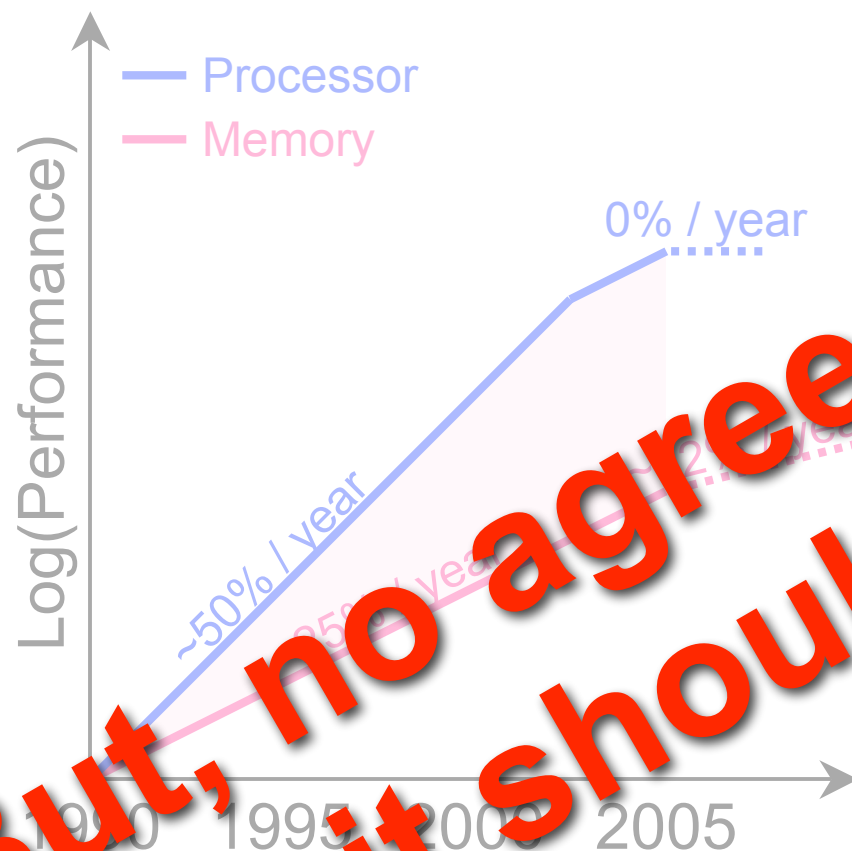
Ph.D. Dissertation Talk

samw@cs.berkeley.edu

Multicore Processors

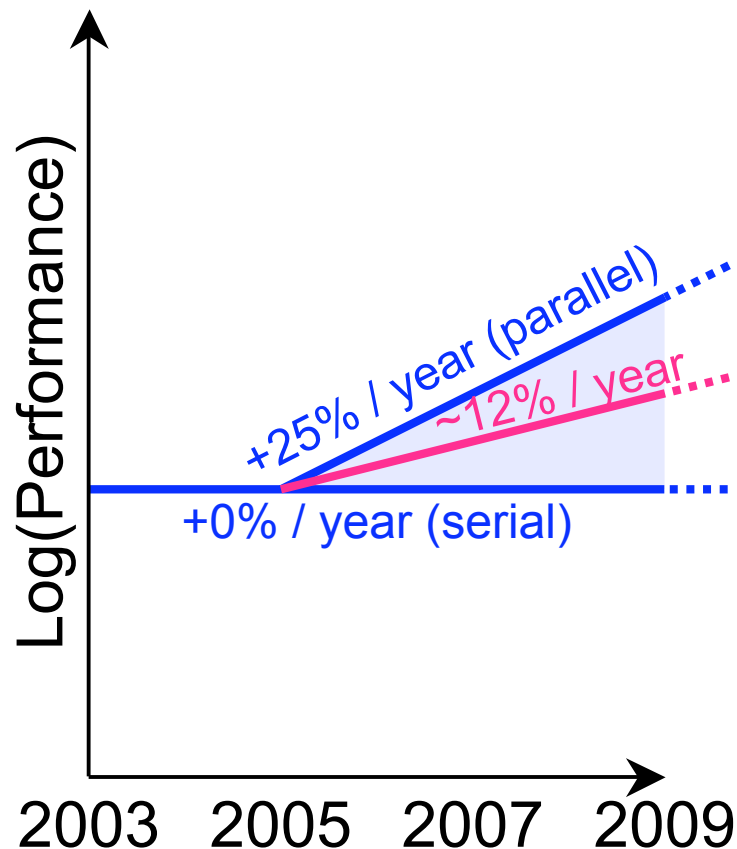


- ❖ Single thread performance scaled at 50% per year
- ❖ Bandwidth increases much more slowly, but we could add additional bits or channels
- ❖ Lack of diversity in architecture = lack of individual tuning
- ❖ Power wall has capped single thread performance

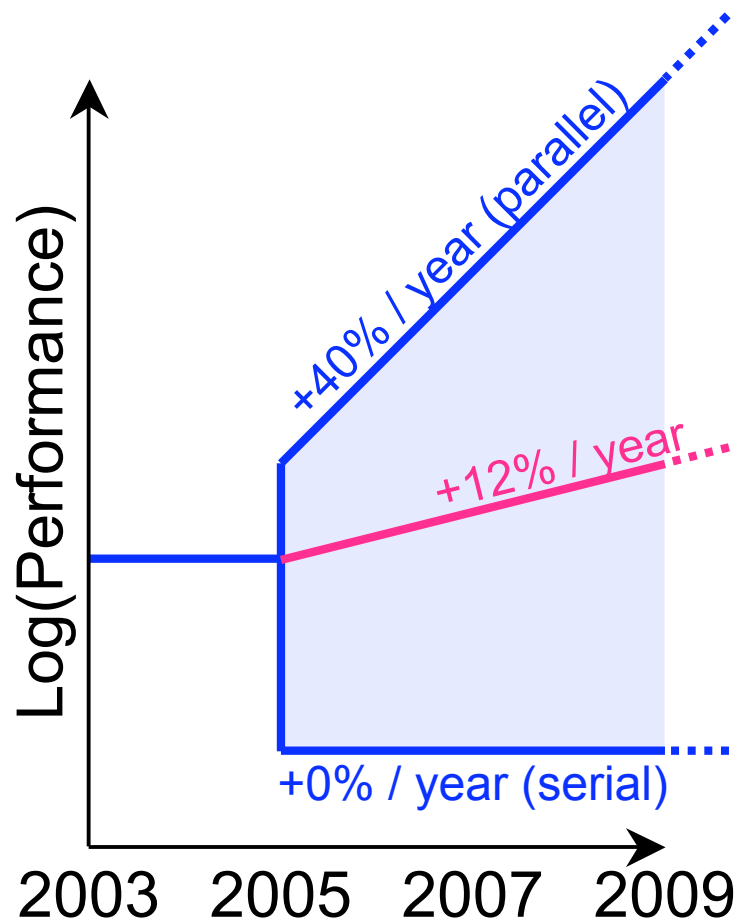


- ❖ Single thread performance scaled at 50% per year
- ❖ Bandwidth increases much more slowly, but we could add additional bits or channels
- ❖ Lack of diversity in architecture = lack of individual tuning
- ❖ Power wall has capped single thread performance

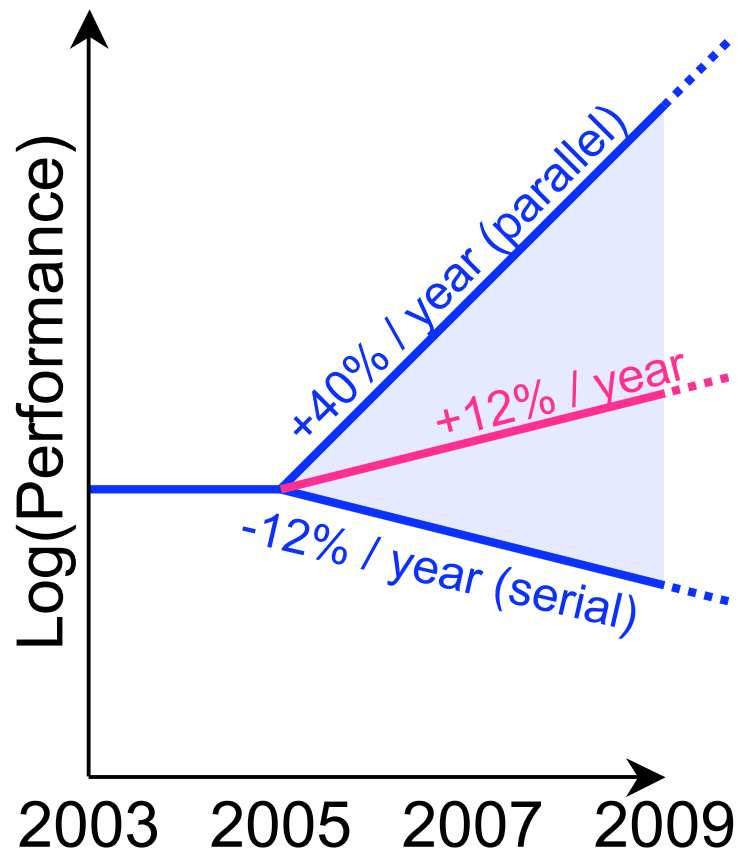
But, no agreement on what it should look like



- ❖ Take existing power limited superscalar processors, and add cores.
- ❖ Serial code shows no improvement
- ❖ Parallel performance might improve at 25% per year.
= improvement in power efficiency from process tech.
- ❖ DRAM bandwidth is currently only improving by ~12% per year.
- ❖ Intel / AMD model.



- ❖ Radically different approach
- ❖ Give up superscalar OOO cores in favor of small, efficient, in-order cores
- ❖ Huge, abrupt, shocking drop in single thread performance
- ❖ May eliminate power wall, allowing many cores, and performance to scale at up to 40% per year
- ❖ Troubling, the number of DRAM channels many need to double every three years
- ❖ Niagara, Cell, GPUs



- ❖ Another option would be to reduce per core performance (and power) every year
- ❖ Graceful degradation in serial performance
- ❖ Bandwidth is still an issue

- ❖ There are still many other architectural knobs
 - Superscalar? Dual issue? VLIW ?
 - Pipeline depth
 - SIMD width ?
 - Multithreading (vertical, simultaneous)?
 - Shared caches vs. Private Caches ?
 - FMA vs. MUL+ADD vs. MUL or ADD
 - Clustering of cores
 - Crossbars vs. Rings vs. Meshes
- ❖ Currently, no consensus on optimal configuration
- ❖ As a result, there is a plethora of multicore architectures

Computational Motifs

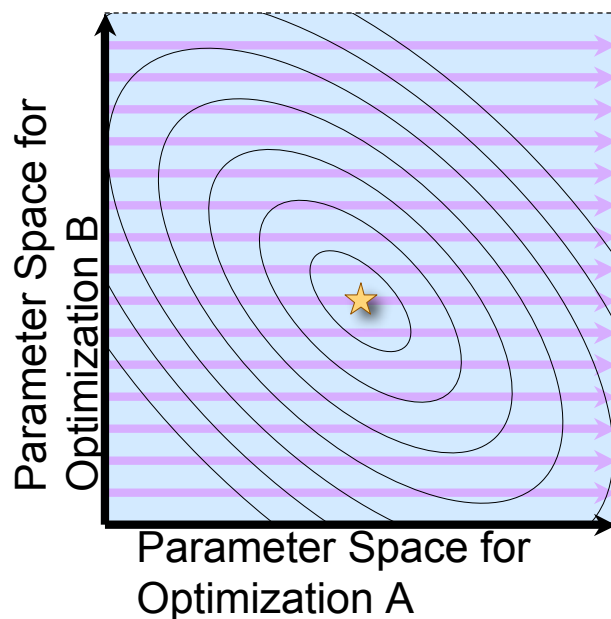
- ❖ Evolved from the Phil Colella's *Seven Dwarfs* of Parallel Computing
- ❖ Numerical Methods common throughout scientific computing
 - Dense and Sparse Linear Algebra
 - Computations on Structured and Unstructured Grids
 - Spectral Methods
 - N-Body Particle Methods
 - Monte Carlo
- ❖ Within each dwarf, there are a number of computational *kernel*s
- ❖ The Berkeley View, and subsequently Par Lab expanded these to many other domains in computing (embedded, SPEC, DB, Games)
 - Graph Algorithms
 - Combinational Logic
 - Finite State Machines
 - etc...
- ❖ rechristened *Computational Motifs*
- ❖ Each could be black-boxed into libraries or frameworks by domain experts.
- ❖ **But how do we get good performance given the diversity of architecture?**

Auto-tuning

- ❖ Given a huge diversity of processor architectures, an code hand-optimized for one architecture will likely deliver poor performance on another. Moreover, code optimized for one input data set may deliver poor performance on another.
- ❖ We want a single code base that delivers performance portability across the breadth of architectures today and into the future.
- ❖ Auto-tuners are composed of two principal components:
 - A code generator based on high-level functionality rather than parsing C
 - And the auto-tuner proper that searches for the optimal parameters for each optimization.
- ❖ *Auto-tuner's don't invent or discover optimizations, they search through the parameter space for a variety of known optimizations.*
- ❖ Proven value in Dense Linear Algebra(ATLAS), Spectral(FFTW,SPIRAL), and Sparse Methods(OSKI)

- ❖ The code generator produces many code variants for some numerical kernel using known optimizations and transformations.
- ❖ For example,
 - cache blocking adds several parameterized loop nests.
 - Prefetching adds parameterized intrinsics to the code
 - Loop unrolling and reordering explicitly unrolls the code. Each unrolling is a unique code variant.
- ❖ Kernels can have dozens of different optimizations, some of which can produce hundreds of code variants.
- ❖ *The code generators used in this work are kernel-specific, and were written in Perl.*

- ❖ In this work, we use two search techniques:
- ❖ **Exhaustive** - examine every combination of parameter for every optimization. (often intractable)
- ❖ **Heuristics** - use knowledge of architecture or algorithm to restrict the search space.



Auto-tuning Performance on Multicore Computers

Overview

Multicore SMPs

The Roofline Model

Auto-tuning LBMHD

Auto-tuning SpMV

Summary

Future Work

- ❖ Introduced the Roofline Model
- ❖ Extended auto-tuning to the structured grid motif (specifically LBMHD)
- ❖ Extended auto-tuning to multicore
 - Fundamentally different from running auto-tuned serial code on multicore SMPs.
 - Apply the concept to LBMHD and SpMV.
- ❖ Analyzed the breadth of multicore architectures in the context of auto-tuned SpMV and LBMHD
- ❖ Discussed future directions in auto-tuning

Multicore SMPs

Overview

Chapter 3

Multicore SMPs

The Roofline Model

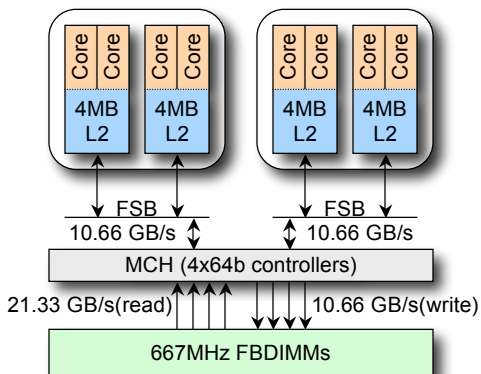
Auto-tuning LBMHD

Auto-tuning SpMV

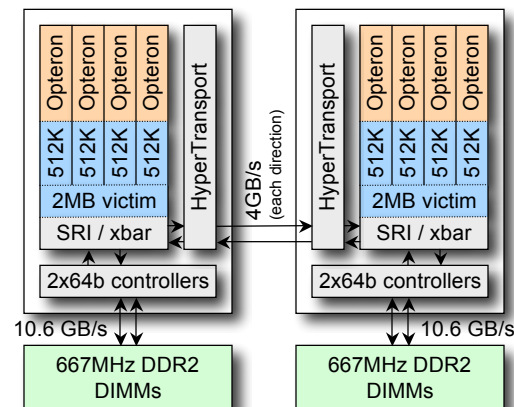
Summary

Future Work

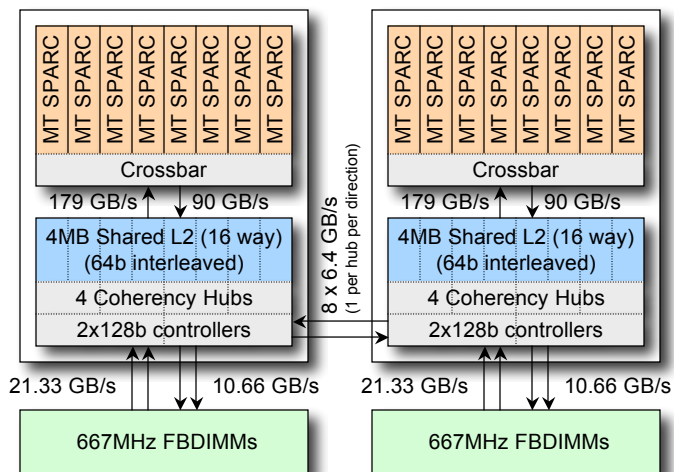
Intel Xeon E5345
(Clovertown)



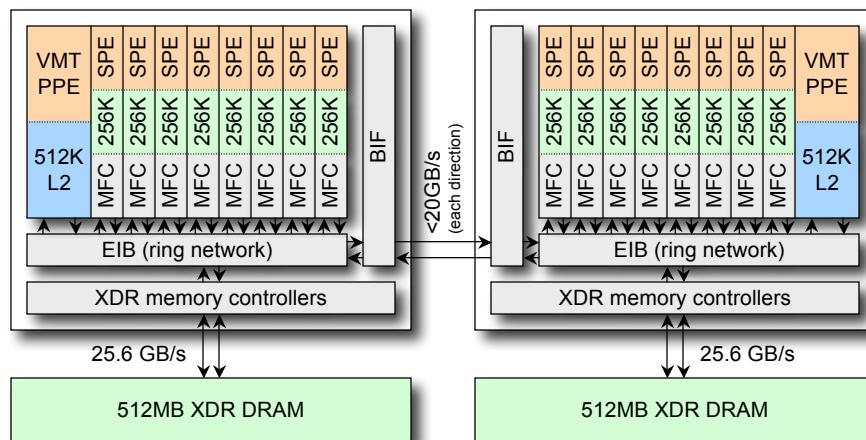
AMD Opteron 2356
(Barcelona)



Sun UltraSPARC T2+ T5140
(Victoria Falls)



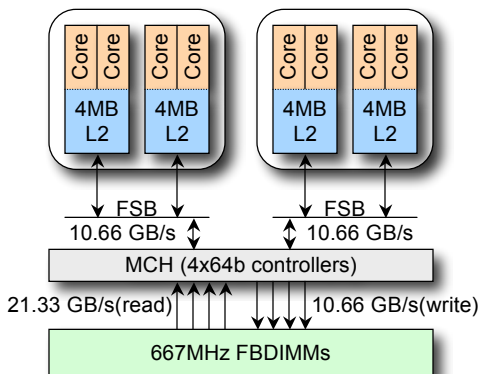
IBM QS20
(Cell Blade)



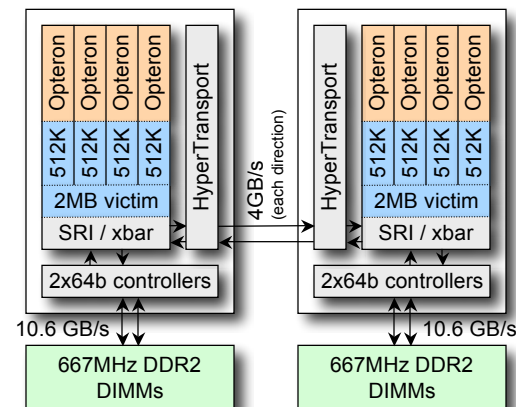
Multicore SMP Systems

(Conventional cache-based memory hierarchy)

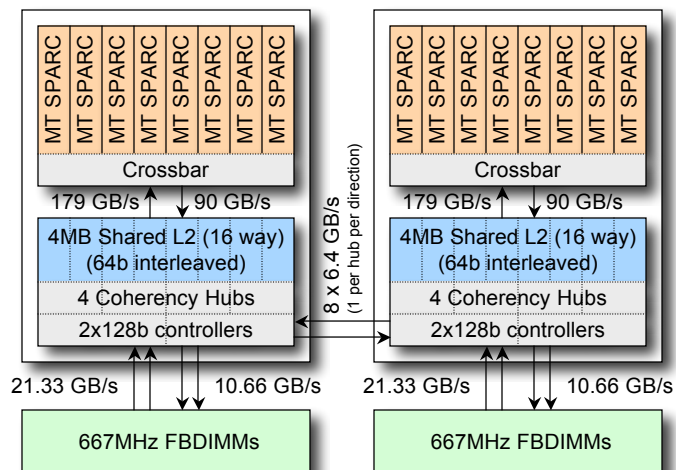
Intel Xeon E5345
(Clovertown)



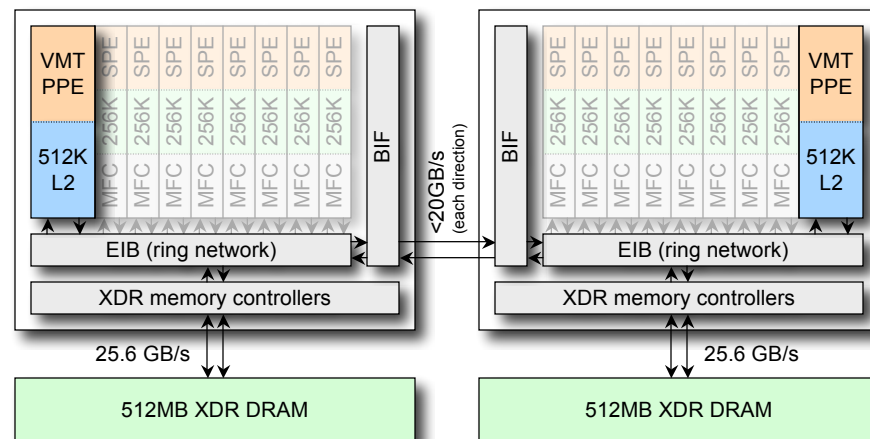
AMD Opteron 2356
(Barcelona)



Sun UltraSPARC T2+ T5140
(Victoria Falls)



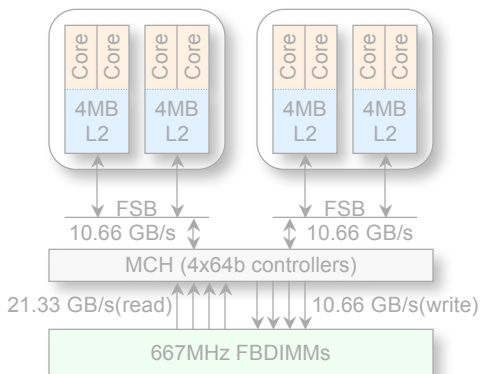
IBM QS20
(Cell Blade)



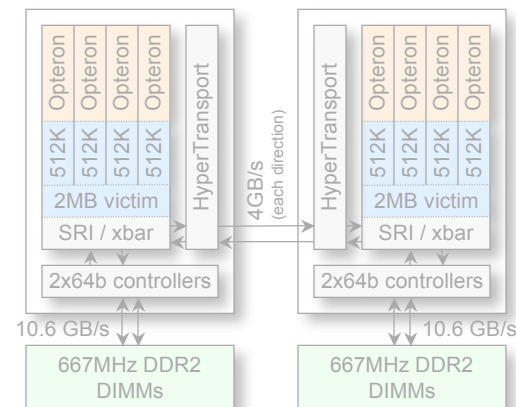
Multicore SMP Systems

(local store-based memory hierarchy)

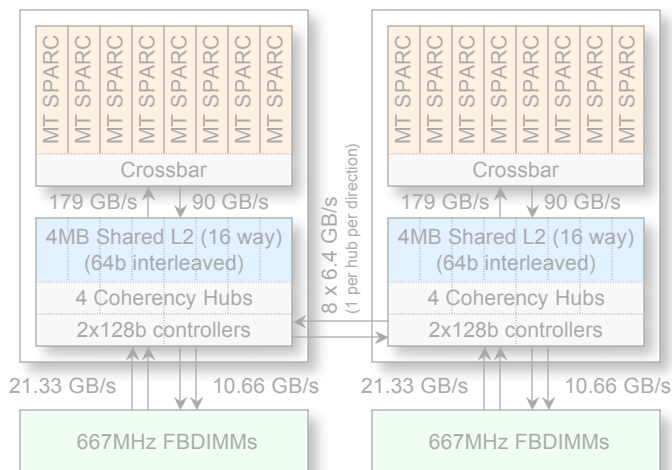
Intel Xeon E5345
(Clovertown)



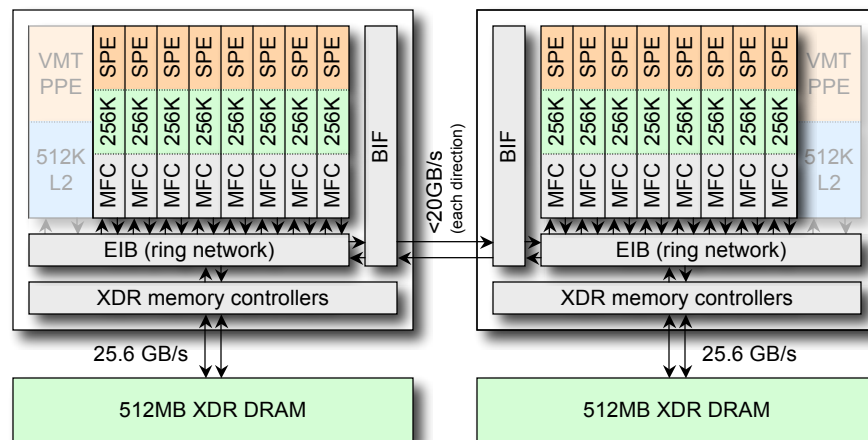
AMD Opteron 2356
(Barcelona)



Sun UltraSPARC T2+ T5140
(Victoria Falls)



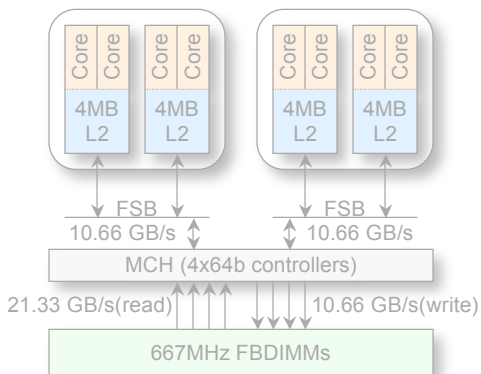
IBM QS20
(Cell Blade)



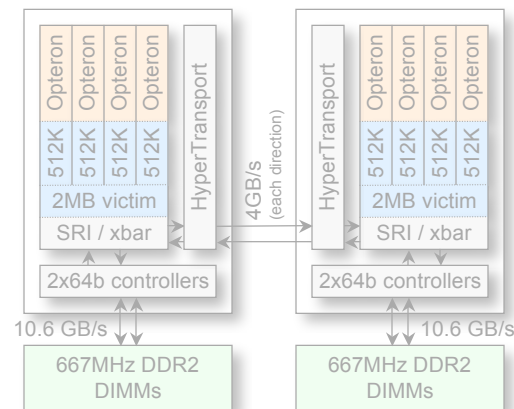
Multicore SMP Systems

(CMT = Chip Multithreading)

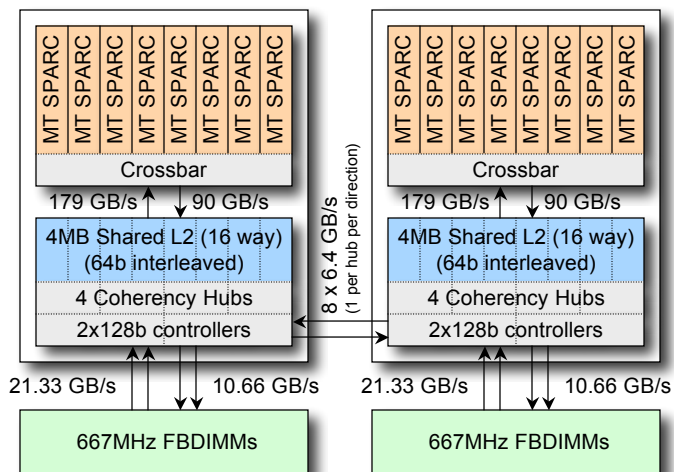
Intel Xeon E5345
(Clovertown)



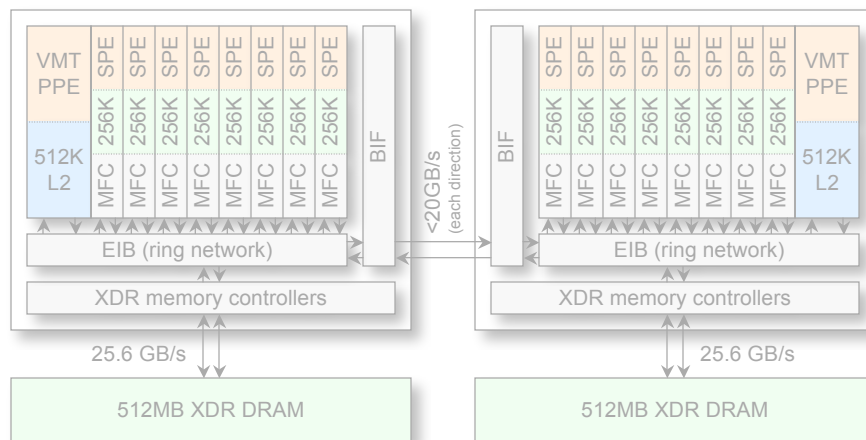
AMD Opteron 2356
(Barcelona)



Sun UltraSPARC T2+ T5140
(Victoria Falls)



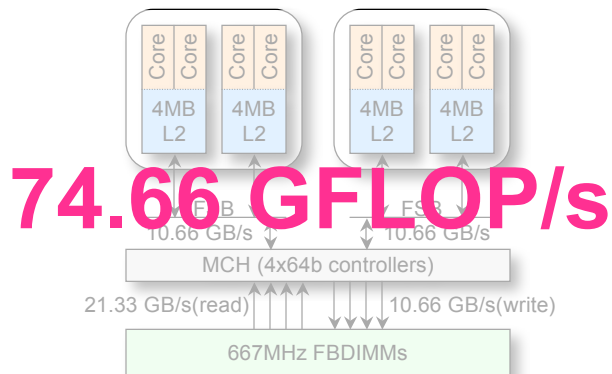
IBM QS20
(Cell Blade)



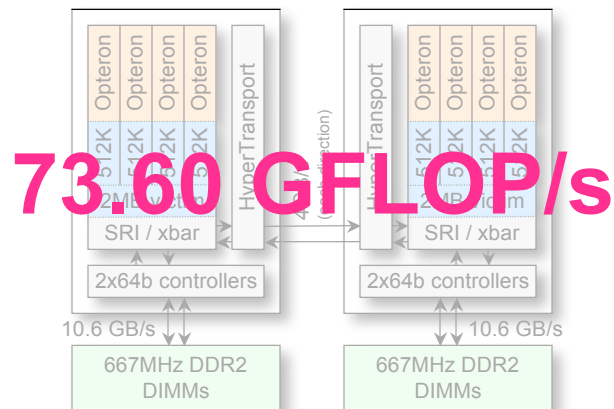
Multicore SMP Systems

(peak double-precision FLOP rates)

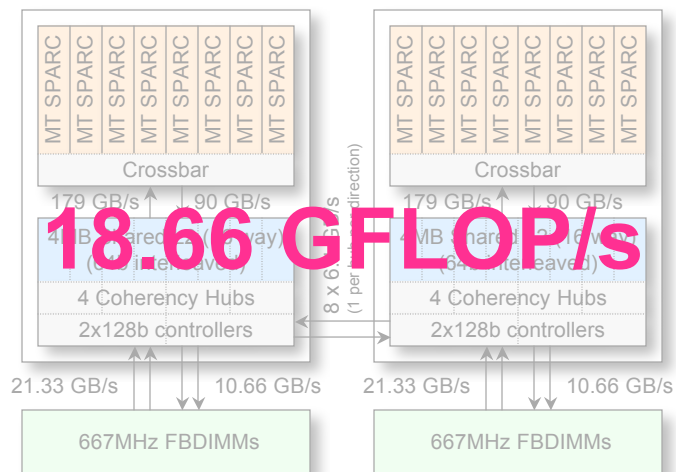
Intel Xeon E5345
(Clovertown)



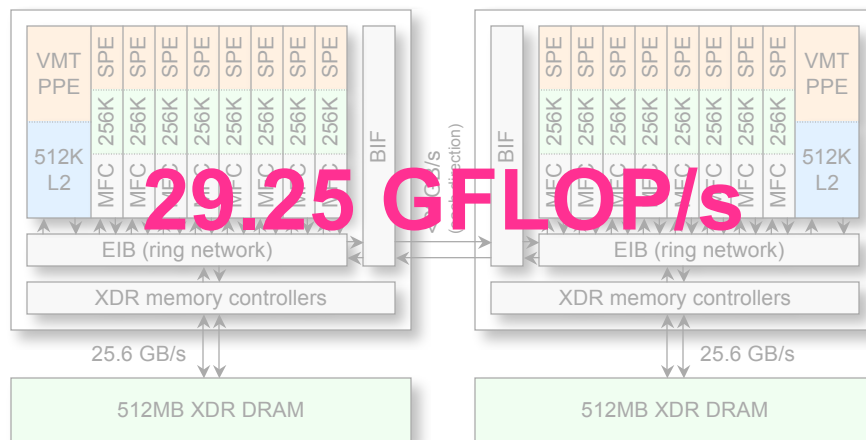
AMD Opteron 2356
(Barcelona)



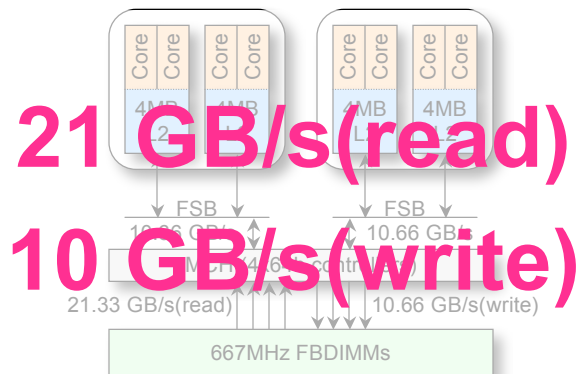
Sun UltraSPARC T2+ T5140
(Victoria Falls)



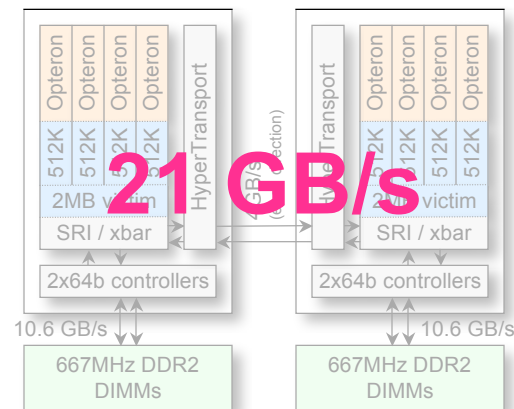
IBM QS20
(Cell Blade)



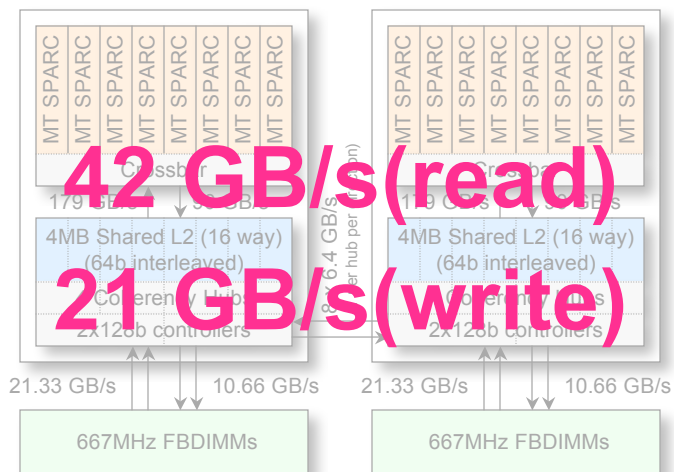
Intel Xeon E5345
(Clovertown)



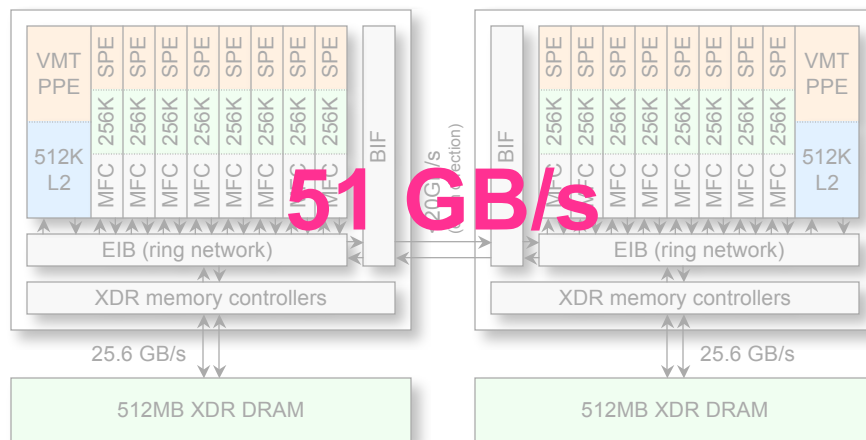
AMD Opteron 2356
(Barcelona)



Sun UltraSPARC T2+ T5140
(Victoria Falls)

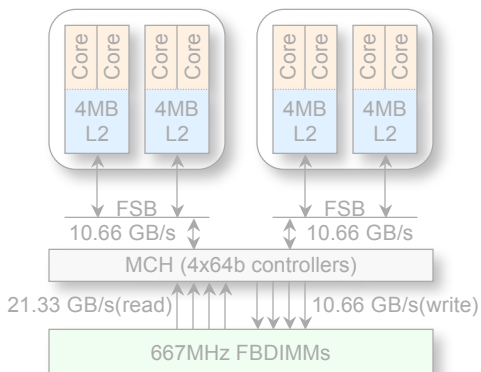


IBM QS20
(Cell Blade)

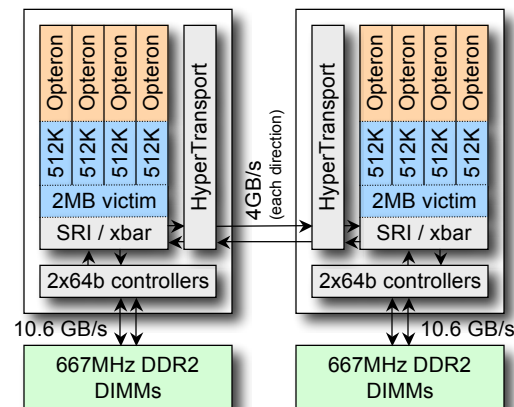


Multicore SMP Systems (Non-Uniform Memory Access)

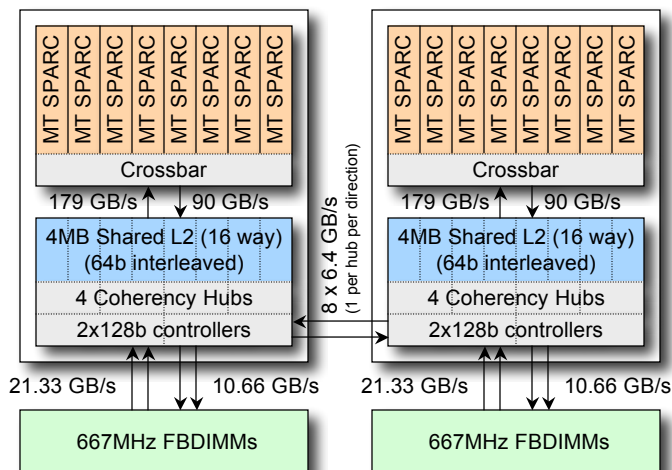
Intel Xeon E5345
(Clovertown)



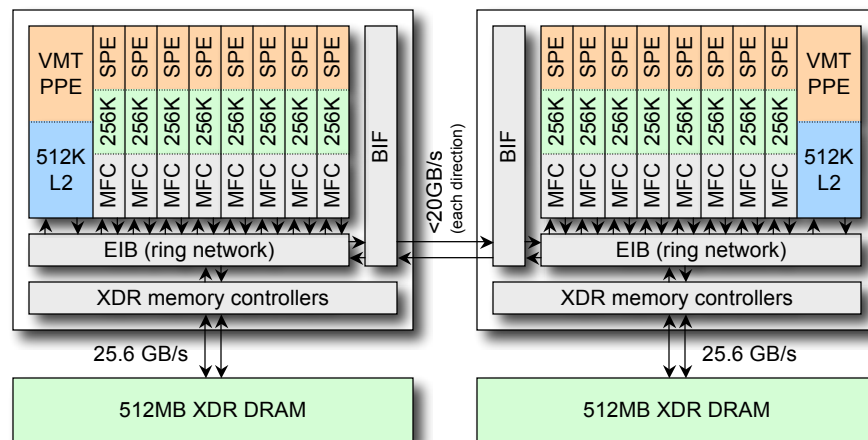
AMD Opteron 2356
(Barcelona)



Sun UltraSPARC T2+ T5140
(Victoria Falls)



IBM QS20
(Cell Blade)



Roofline Model

Overview

Multicore SMPs

The Roofline Model

Auto-tuning LBMHD

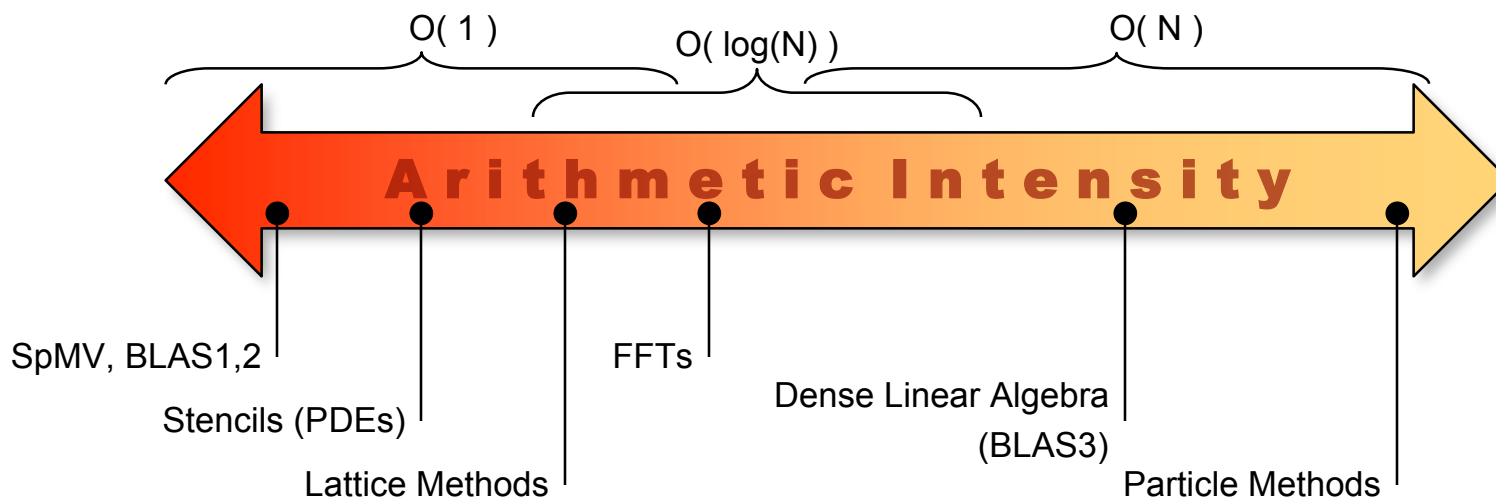
Auto-tuning SpMV

Summary

Future Work

Chapter 4

- ❖ Total bytes to/from DRAM
- ❖ Can categorize into:
 - Compulsory misses
 - Capacity misses
 - Conflict misses
 - Write allocations
 - ...
- ❖ Oblivious of lack of sub-cache line spatial locality



- ❖ For purposes of this talk, we'll deal with floating-point kernels
- ❖ **Arithmetic Intensity $\sim$ Total FLOPs / Total DRAM Bytes**
- ❖ Includes cache effects
- ❖ Many interesting problems have constant AI (w.r.t. problem size)
 - Bad given slowly increasing DRAM bandwidth
 - Bandwidth and Traffic are key optimizations

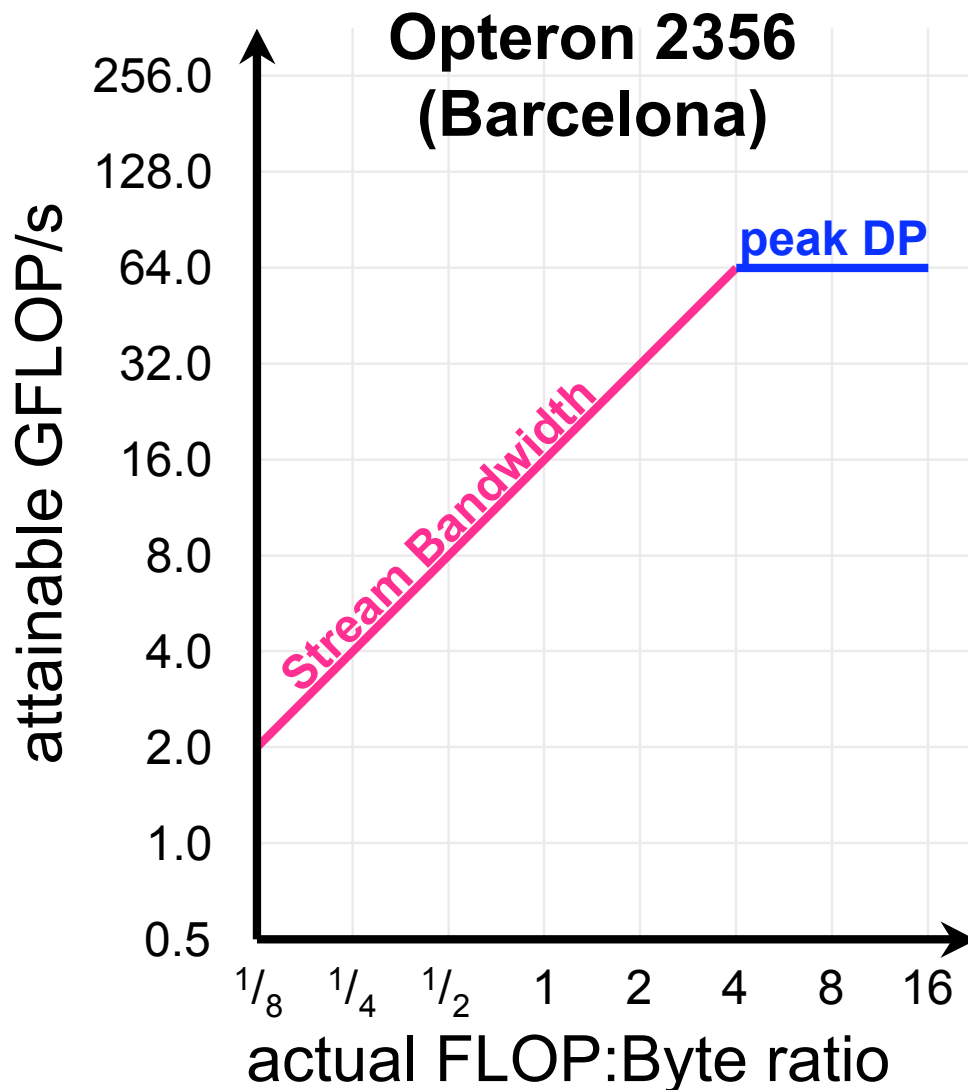
- ❖ Synthesize communication, computation, and locality into a single visually-intuitive performance figure using bound and bottleneck analysis.

$$\text{Attainable Performance}_{ij} = \min \begin{cases} \text{FLOP/s with Optimizations}_{1-i} \\ \text{AI} * \text{Bandwidth with Optimizations}_{1-j} \end{cases}$$

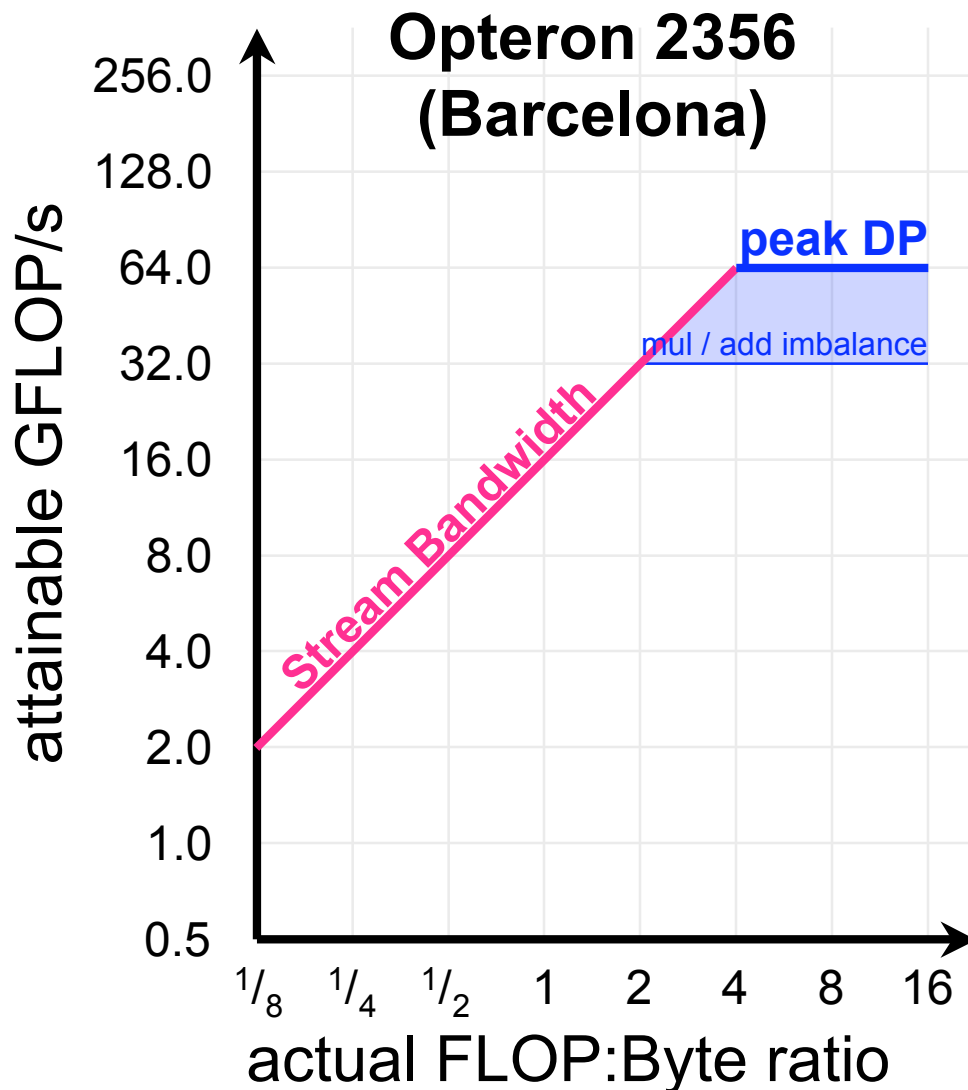
- ❖ Given a kernel's arithmetic intensity (based on DRAM traffic after being filtered by the cache), programmers can inspect the figure, and bound performance.
- ❖ Moreover, provides insights as to which optimizations will potentially be beneficial.

Constructing a Roofline Model

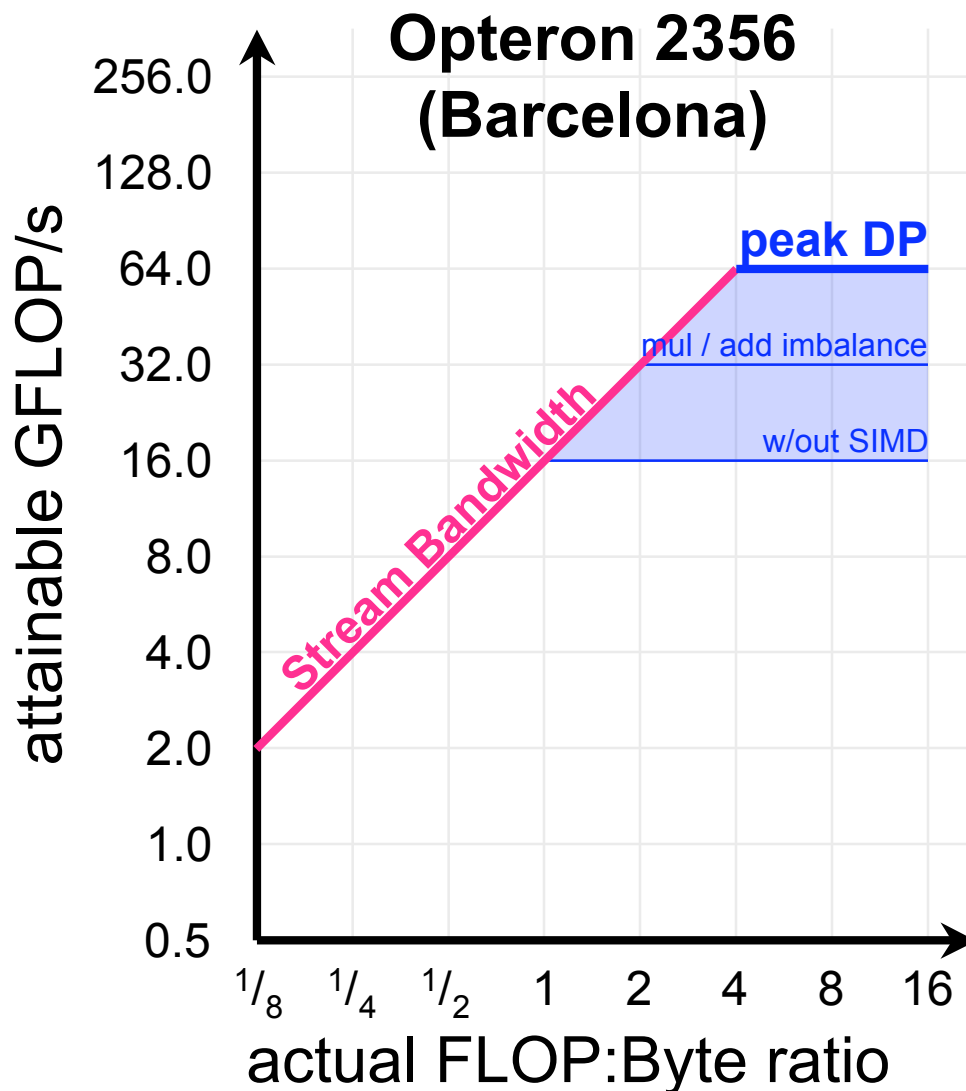
(computational ceilings)



- ❖ Plot on log-log scale
- ❖ Given AI, we can easily bound performance
- ❖ But architectures are much more complicated
- ❖ We will bound performance as we eliminate specific forms of in-core parallelism



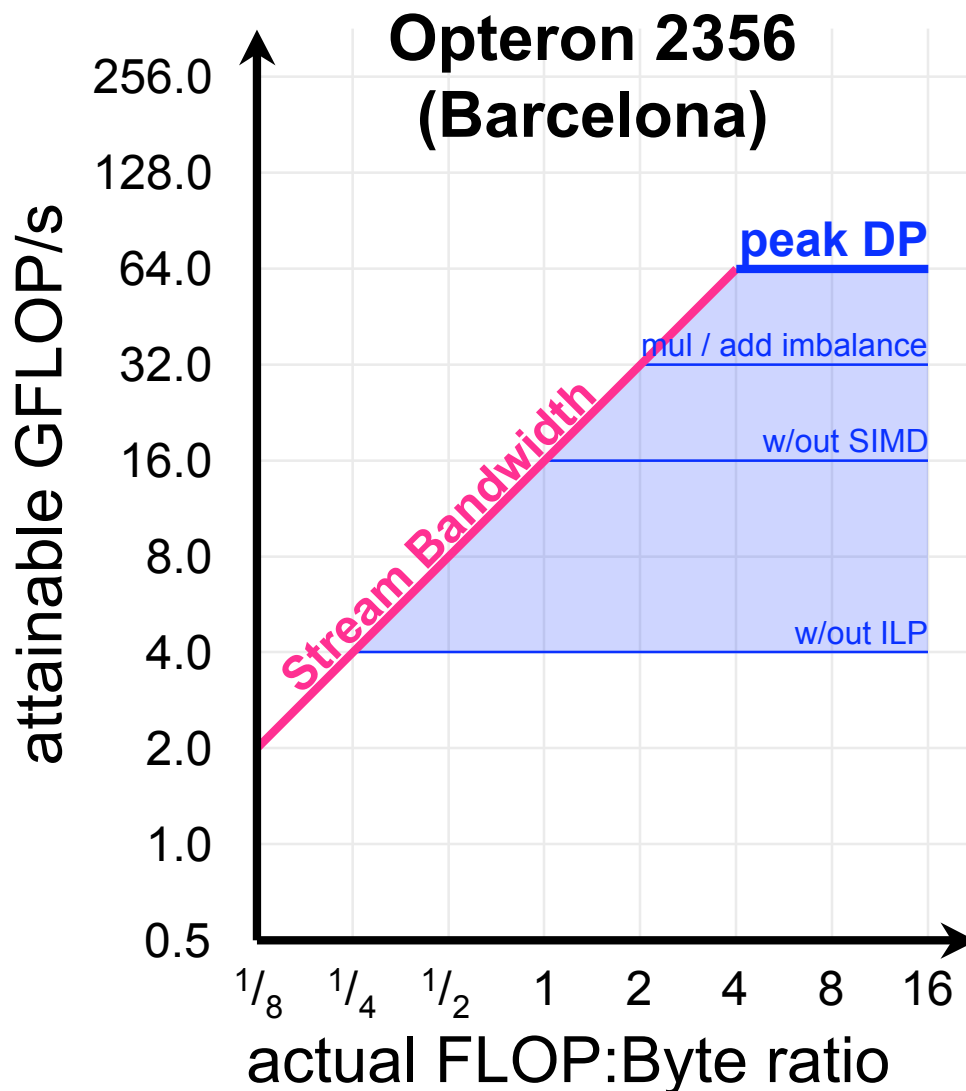
- ❖ Opterons have dedicated multipliers and adders.
- ❖ If the code is dominated by adds, then attainable performance is half of peak.
- ❖ We call these **Ceilings**
- ❖ They act like constraints on performance



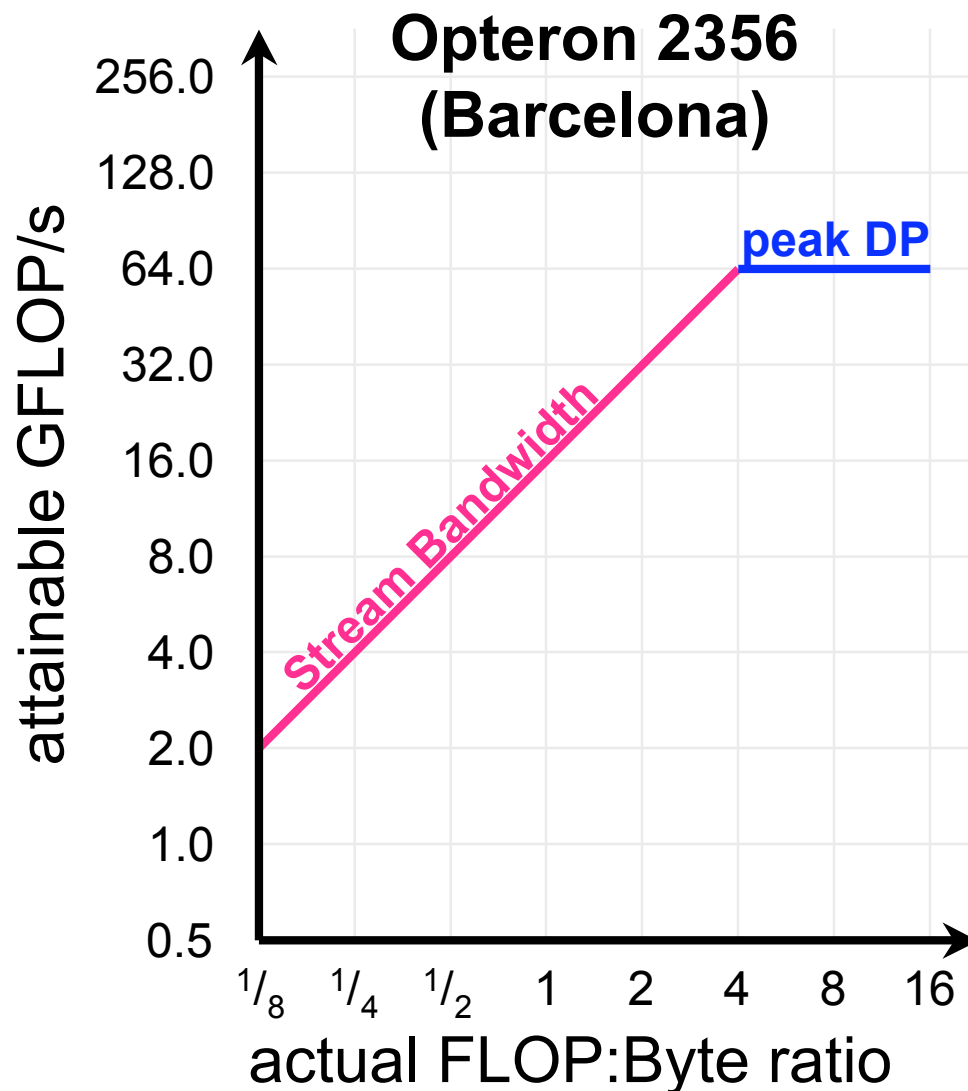
- ❖ Opterons have 128-bit datapaths.
- ❖ If instructions aren't SIMDized, attainable performance will be halved

Constructing a Roofline Model

(computational ceilings)



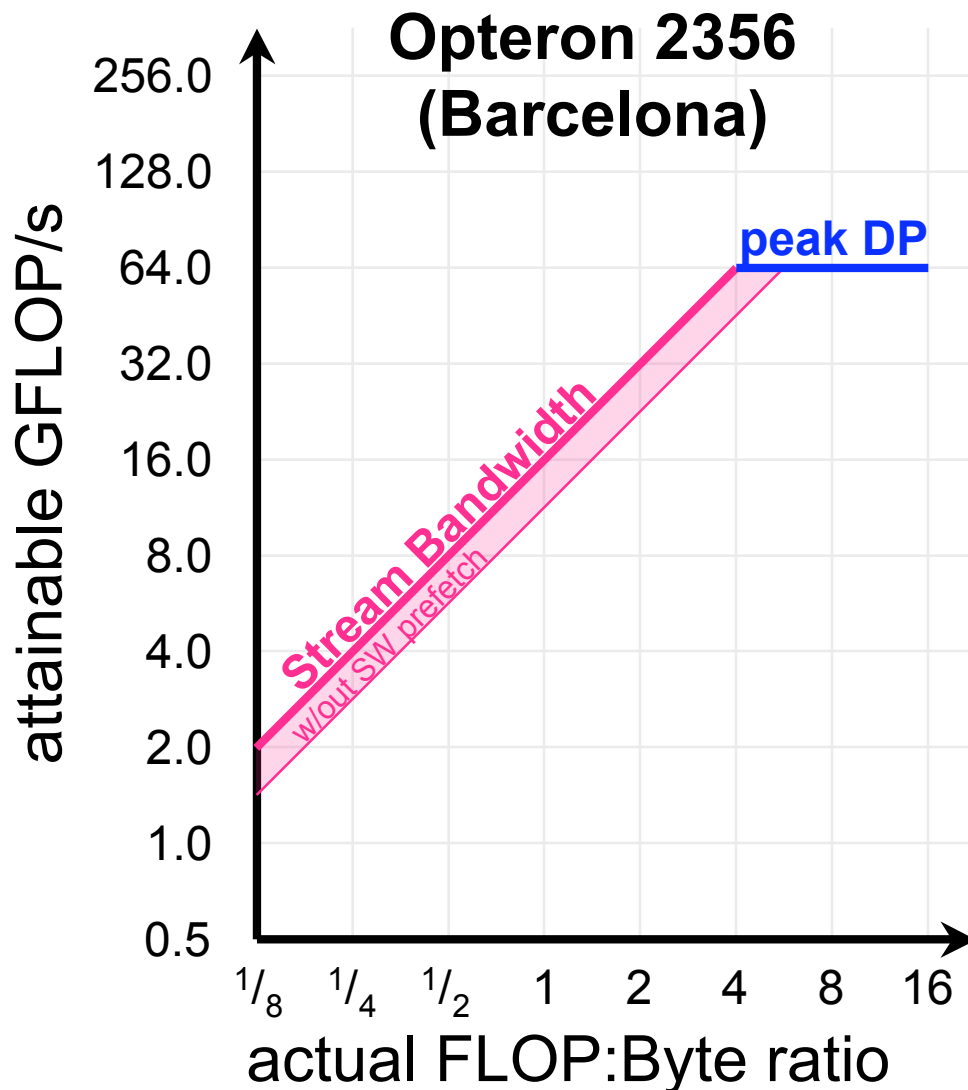
- ❖ On Opterons, floating-point instructions have a 4 cycle latency.
- ❖ If we don't express 4-way ILP, performance will drop by as much as 4x



- ❖ We can perform a similar exercise taking away parallelism from the memory subsystem

Constructing a Roofline Model

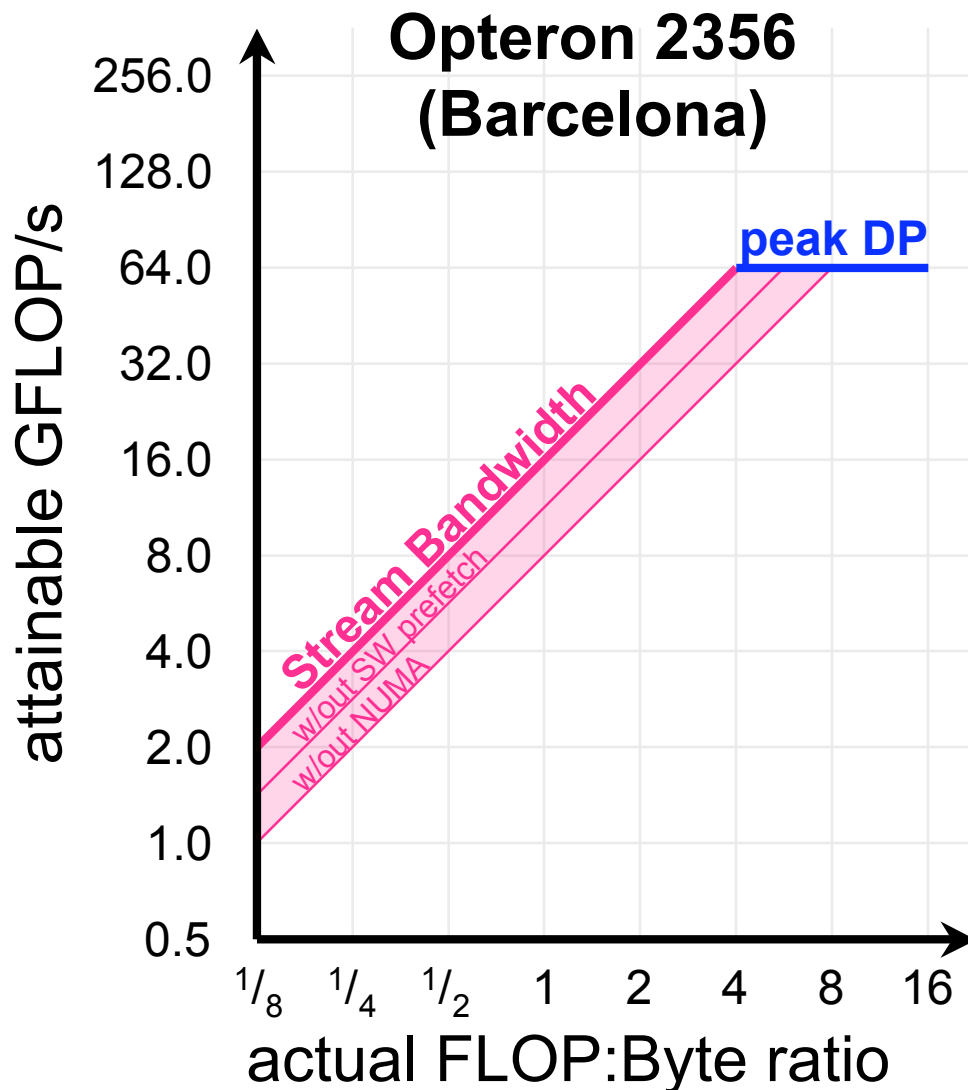
(communication ceilings)



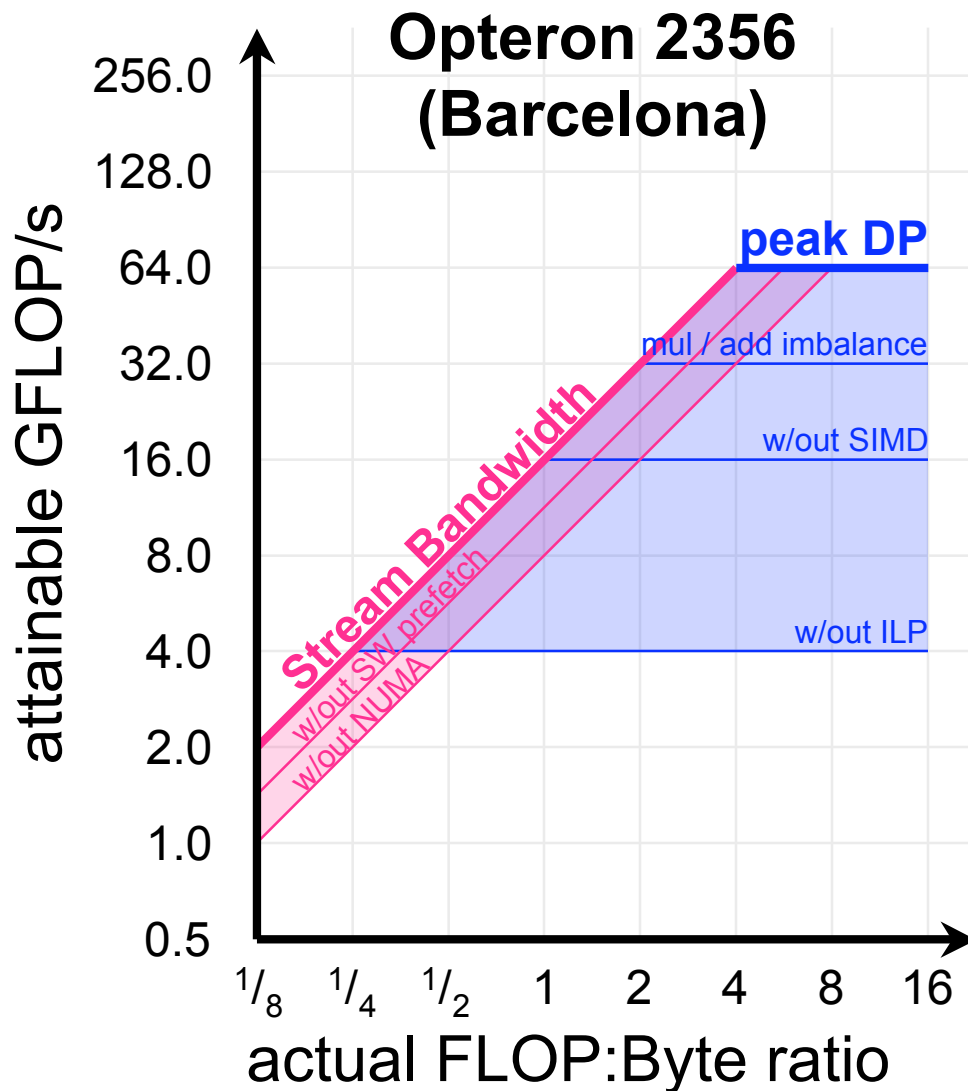
- ❖ Explicit software prefetch instructions are required to achieve peak bandwidth

Constructing a Roofline Model

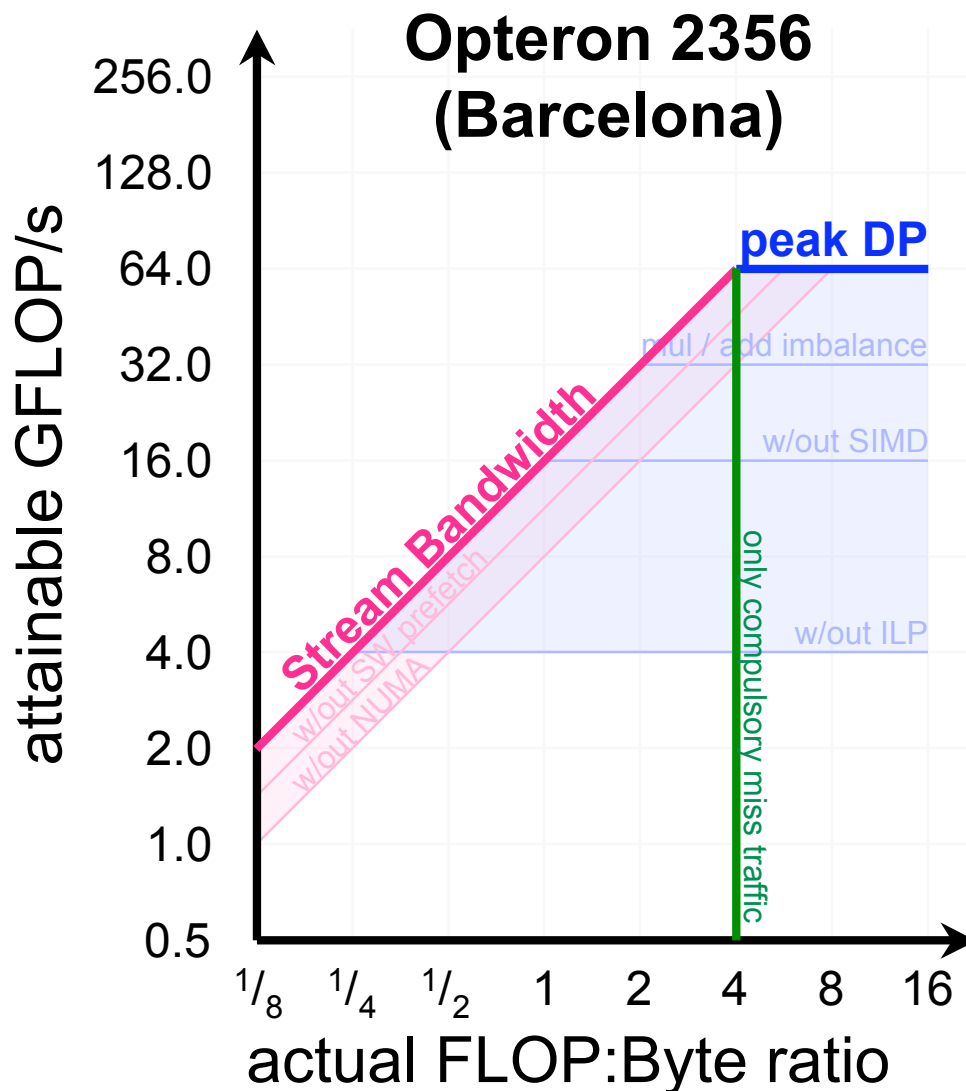
(communication ceilings)



- ❖ Opterons are NUMA
- ❖ As such memory traffic must be correctly balanced among the two sockets to achieve good Stream bandwidth.
- ❖ We could continue this by examining strided or random memory access patterns

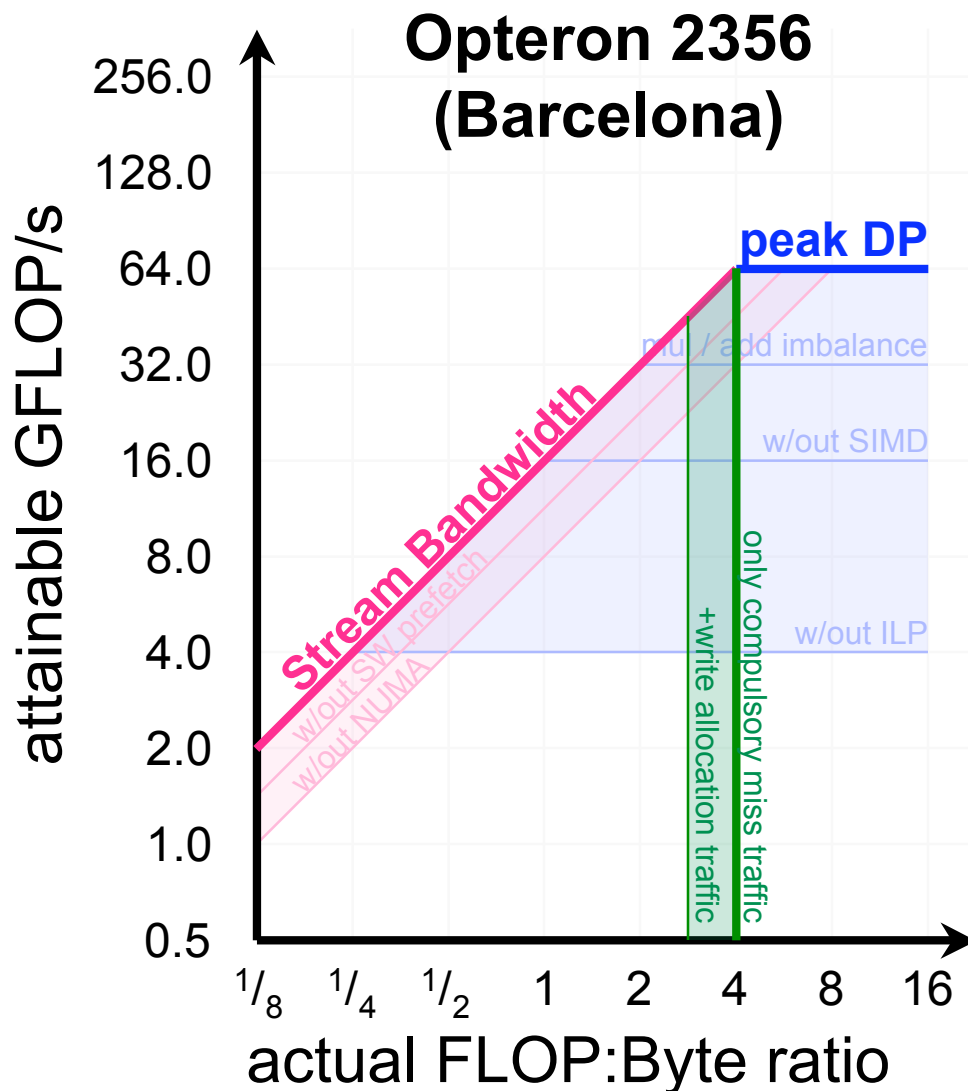


- ❖ We may bound performance based on the combination of expressed in-core parallelism and attained bandwidth.



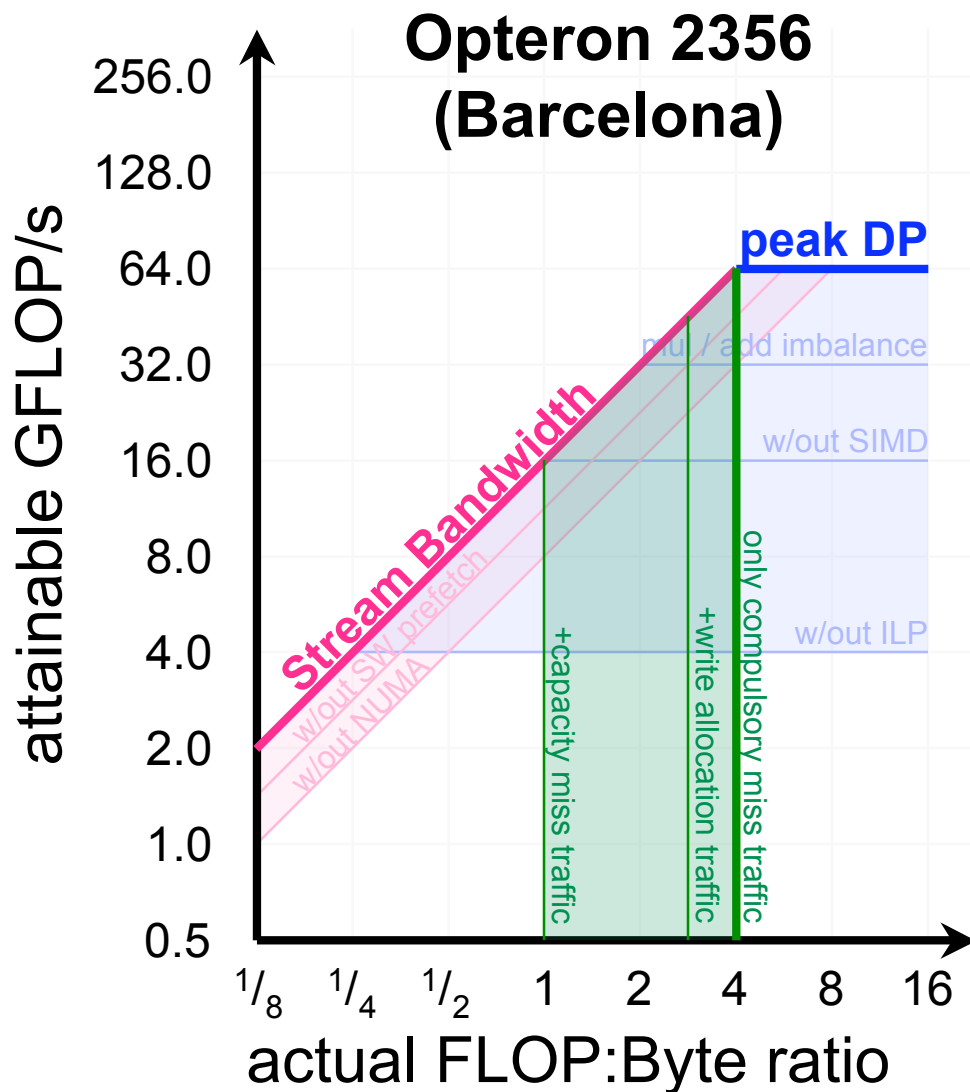
- ❖ Remember, memory traffic includes more than just compulsory misses.
- ❖ As such, actual arithmetic intensity may be substantially lower.
- ❖ Walls are unique to the architecture-kernel combination

$$AI = \frac{\text{FLOPs}}{\text{Compulsory Misses}}$$



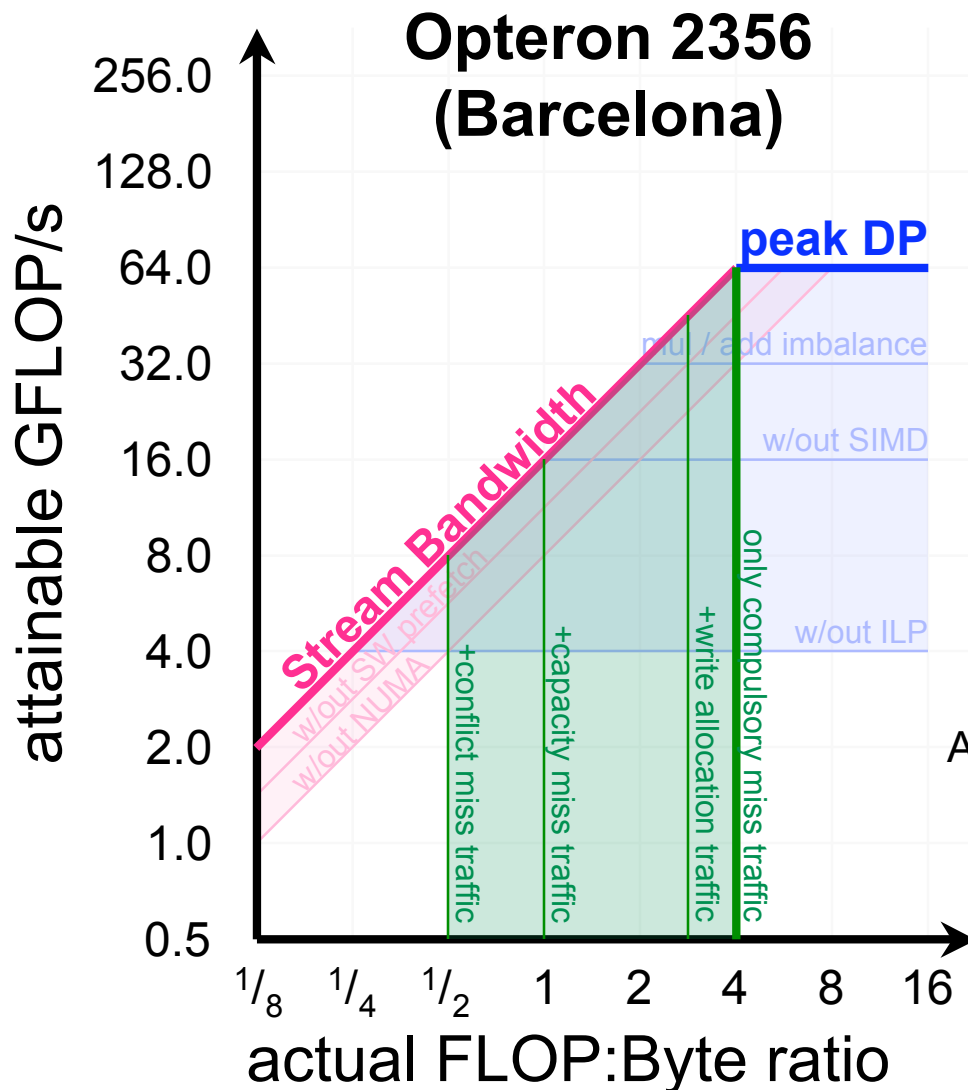
- ❖ Remember, memory traffic includes more than just compulsory misses.
- ❖ As such, actual arithmetic intensity may be substantially lower.
- ❖ Walls are unique to the architecture-kernel combination

$$AI = \frac{\text{FLOPs}}{\text{Allocations} + \text{Compulsory Misses}}$$



- ❖ Remember, memory traffic includes more than just compulsory misses.
- ❖ As such, actual arithmetic intensity may be substantially lower.
- ❖ Walls are unique to the architecture-kernel combination

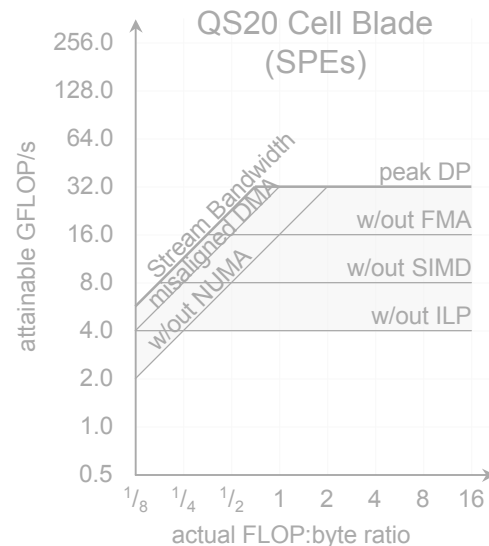
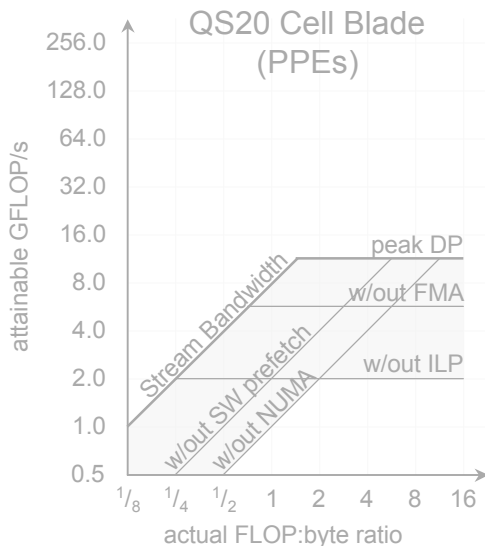
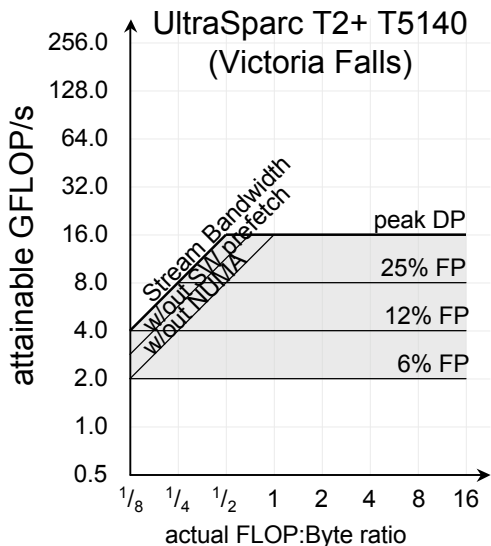
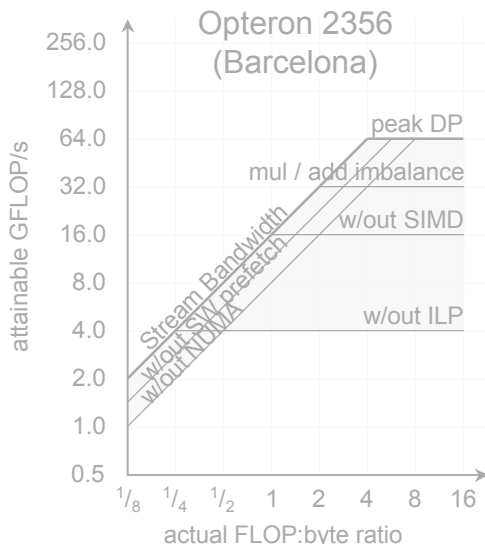
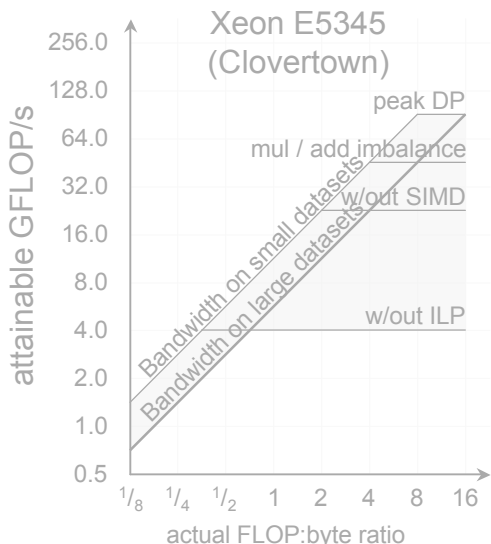
$$AI = \frac{\text{FLOPs}}{\text{Capacity} + \text{Allocations} + \text{Compulsory}}$$



- ❖ Remember, memory traffic includes more than just compulsory misses.
- ❖ As such, actual arithmetic intensity may be substantially lower.
- ❖ Walls are unique to the architecture-kernel combination

$$AI = \frac{\text{FLOPs}}{\text{Conflict} + \text{Capacity} + \text{Allocations} + \text{Compulsory}}$$

- ❖ Note, the multithreaded Niagara is limited by the instruction mix rather than a lack of expressed in-core parallelism
- ❖ Clearly some architectures are more dependent on bandwidth optimizations while others on in-core optimizations.



Auto-tuning

Lattice-Boltzmann

Magnetohydrodynamics (LBMHD)

Chapter 6

Overview

Multicore SMPs

The Roofline Model

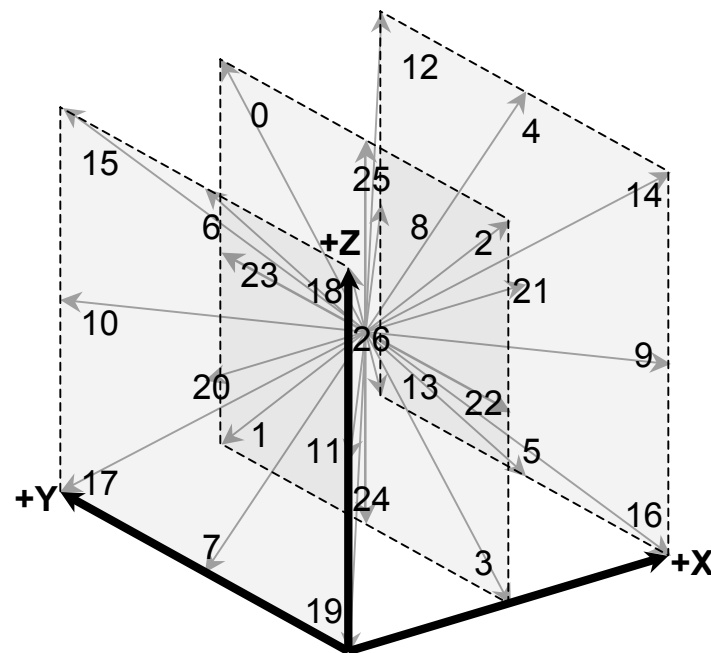
Auto-tuning LBMHD

Auto-tuning SpMV

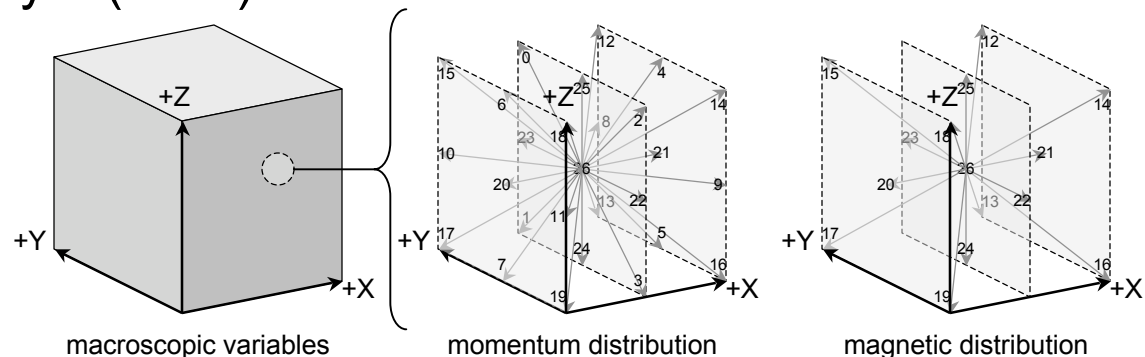
Summary

Future Work

- ❖ Structured grid code, with a series of time steps
- ❖ Popular in CFD
- ❖ Allows for complex boundary conditions
- ❖ No temporal locality between points in space within one time step
- ❖ Higher dimensional phase space
 - Simplified kinetic model that maintains the macroscopic quantities
 - **Distribution functions** (e.g. 5-27 velocities per point in space) are used to reconstruct macroscopic quantities
 - Significant Memory capacity requirements

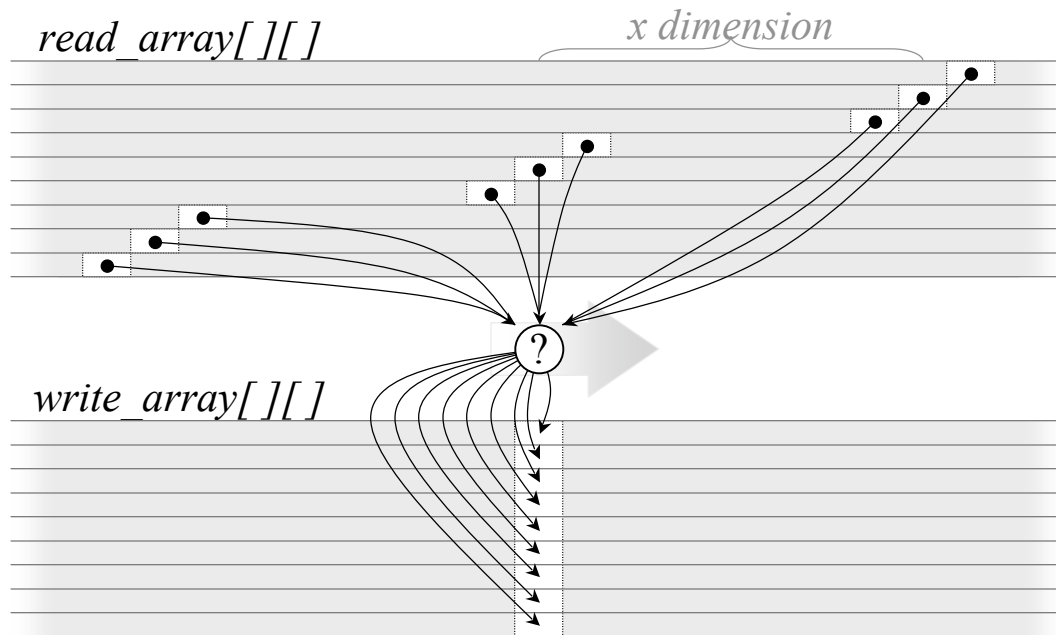
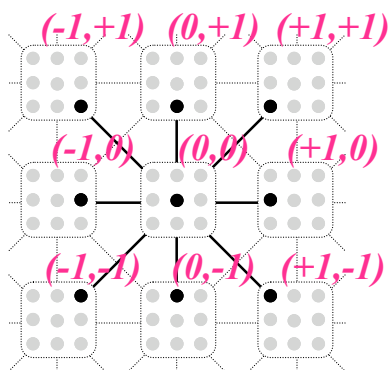


- ❖ Plasma turbulence simulation
- ❖ Two distributions:
 - momentum distribution (27 scalar components)
 - magnetic distribution (15 vector components)
- ❖ Three macroscopic quantities:
 - Density
 - Momentum (vector)
 - Magnetic Field (vector)
- ❖ Must read 73 doubles, and update 79 doubles per point in space
- ❖ Requires about 1300 floating point operations per point in space
- ❖ Just over 1.0 FLOPs/byte (ideal)

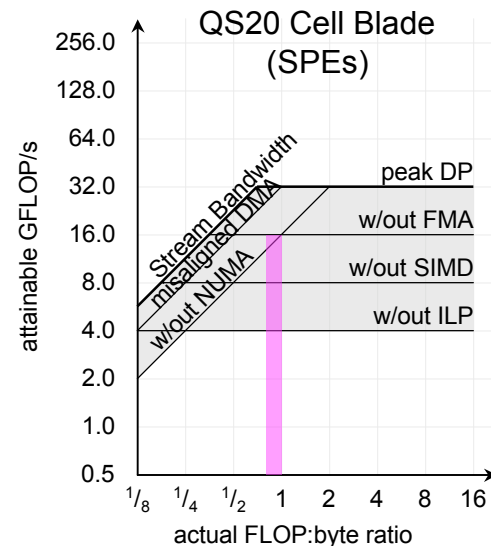
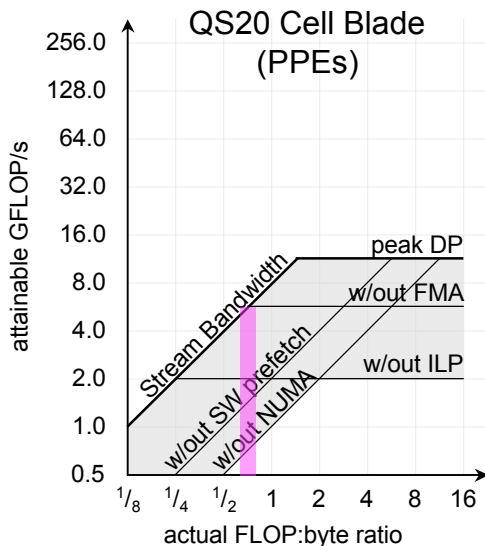
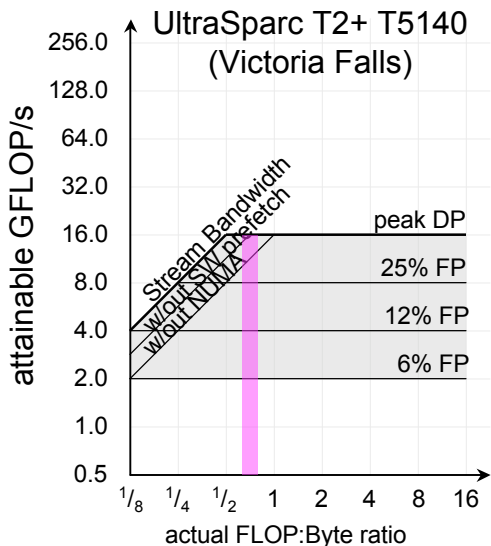
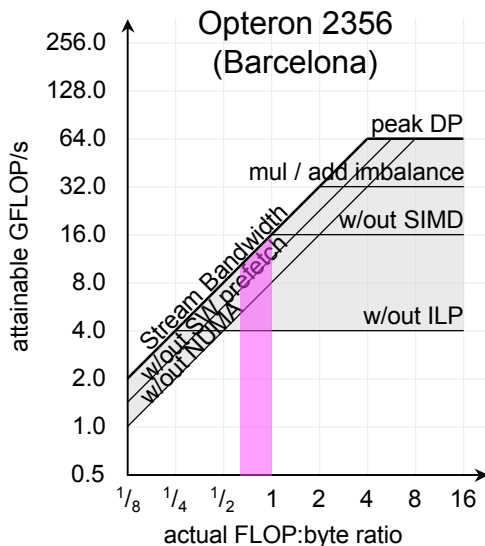
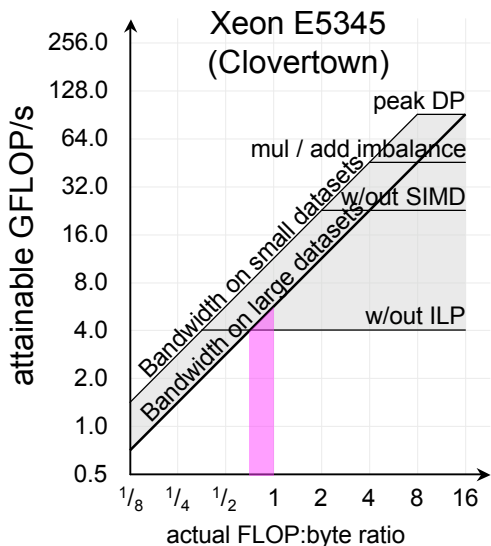


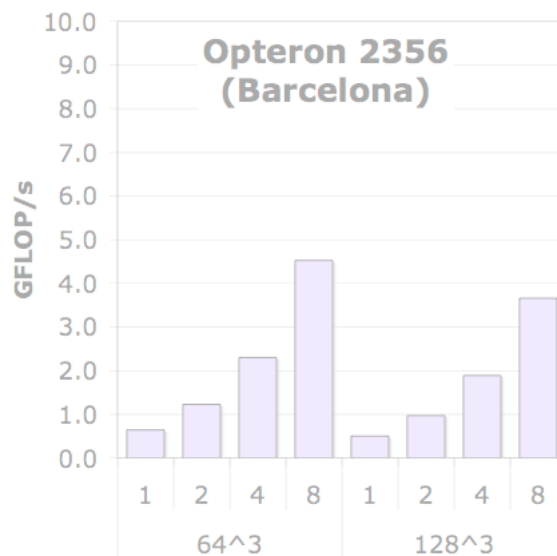
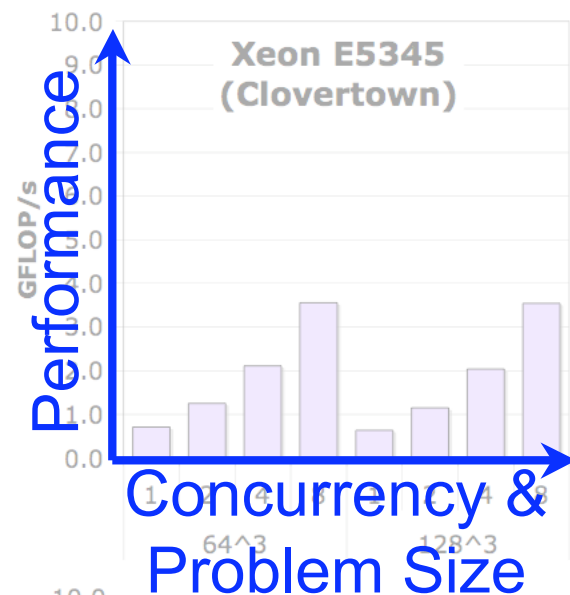
- ❖ Data Structure choices:
 - **Array of Structures**: no spatial locality, strided access
 - **Structure of Arrays**: huge number of memory streams per thread, but guarantees spatial locality, unit-stride, and vectorizes well
- ❖ Parallelization
 - Fortran version used MPI to communicate between nodes.
= bad match for multicore
 - The version in this work uses pthreads for multicore
(this thesis is not about innovation in the threading model or programming language)
 - MPI is not used when auto-tuning
- ❖ Two problem sizes:
 - 64^3 (~330MB)
 - 128^3 (~2.5GB)

- ❖ Consider a simple D2Q9 lattice method using SOA
- ❖ There are 9 read arrays, and 9 write arrays, but all accesses are unit stride
- ❖ LBMHD has 73 read and 79 write streams per thread

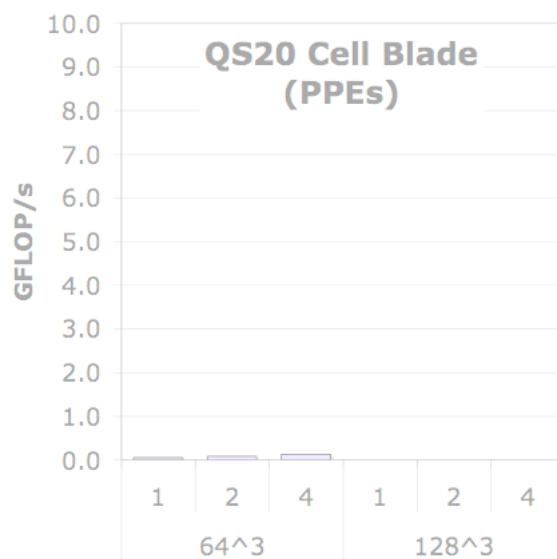
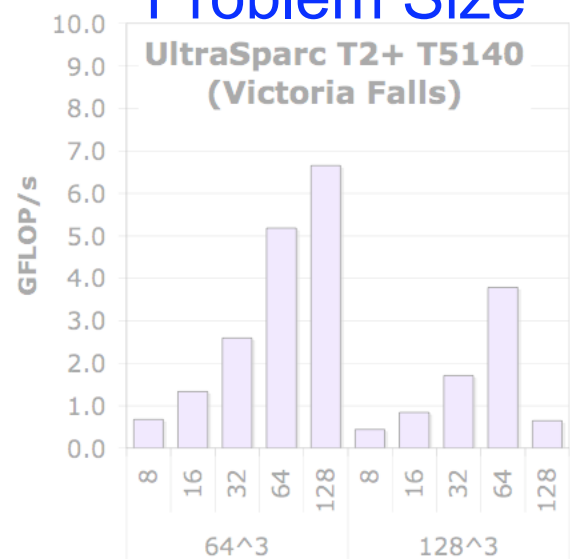


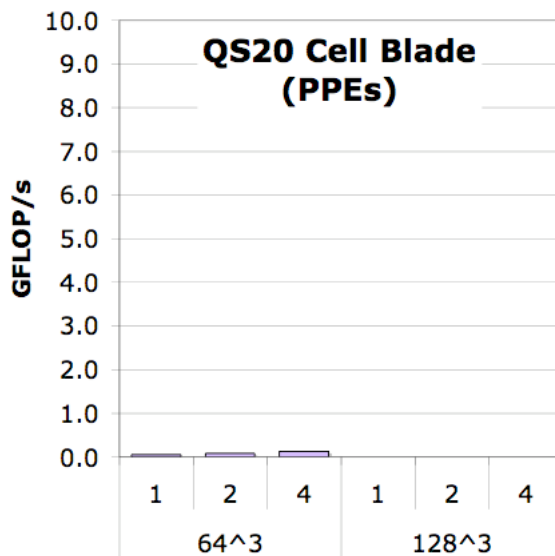
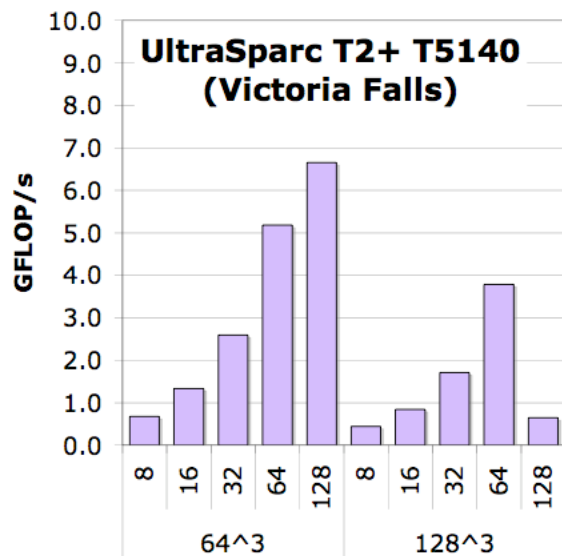
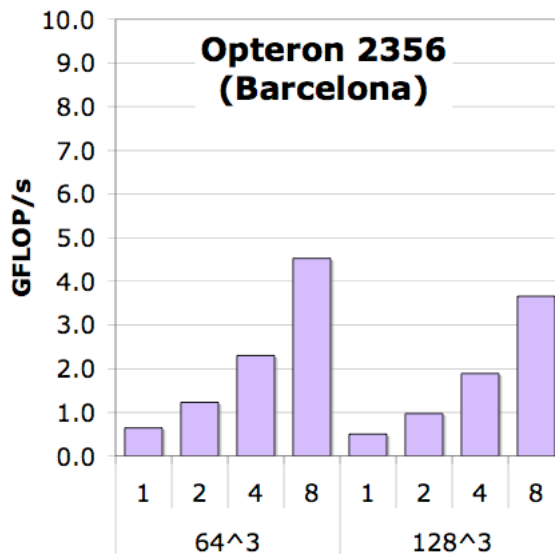
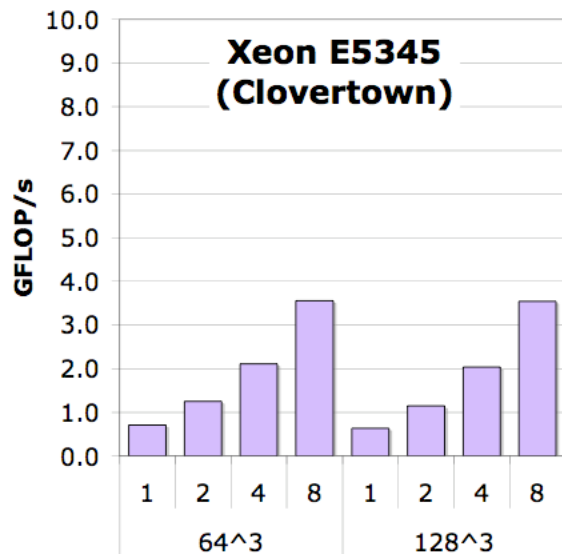
- ❖ LBMHD has a AI of **0.7** on write allocate architectures, and **1.0** on those with cache bypass or no write allocate.
- ❖ MUL / ADD imbalance
- ❖ Some architectures will be bandwidth-bound, while others compute bound.





- ❖ Standard cache-based implementation can be easily parallelized with pthreads.
- ❖ NUMA is implicitly exploited
- ❖ Although scalability looks good, is performance ?



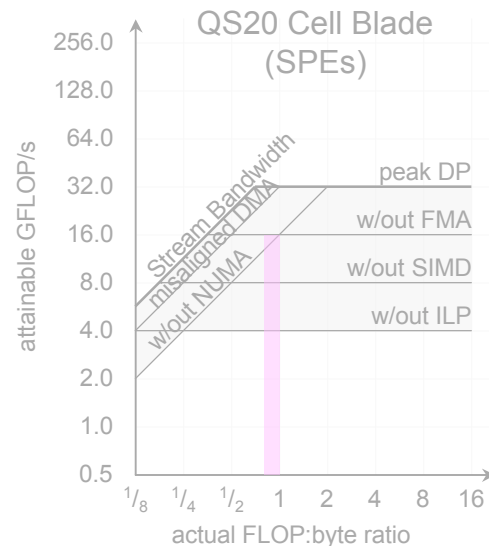
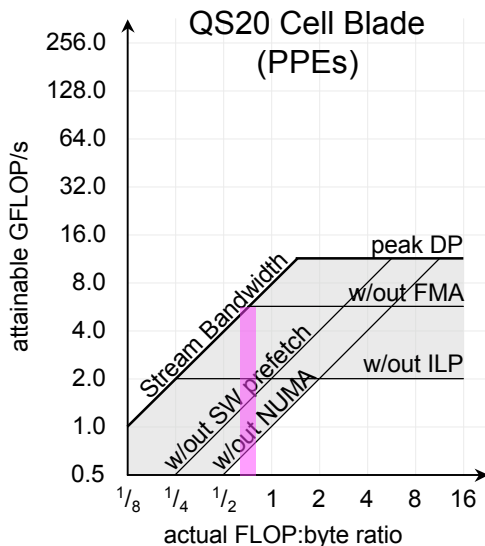
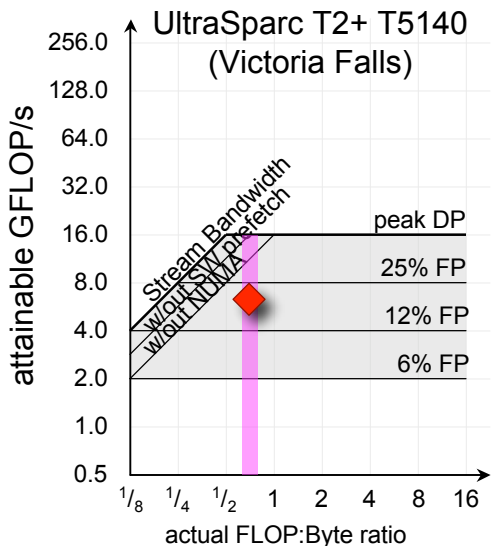
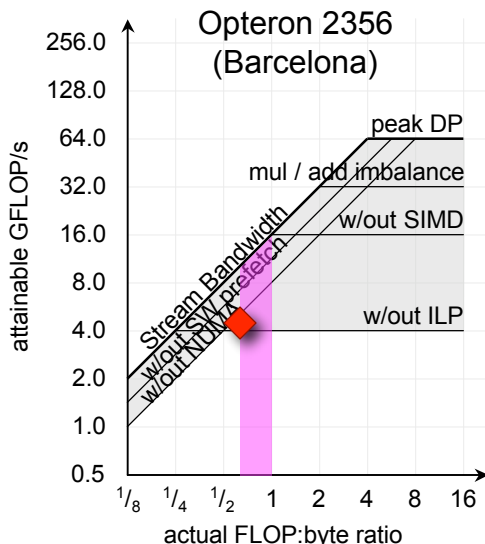
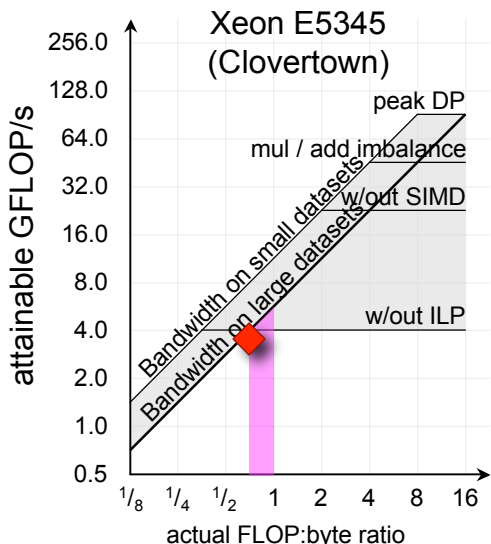


- ❖ Standard cache-based implementation can be easily parallelized with pthreads.
- ❖ NUMA is implicitly exploited
- ❖ Although scalability looks good, is performance ?

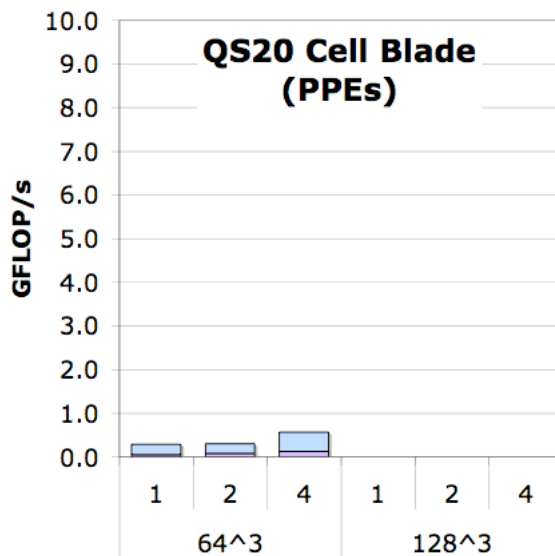
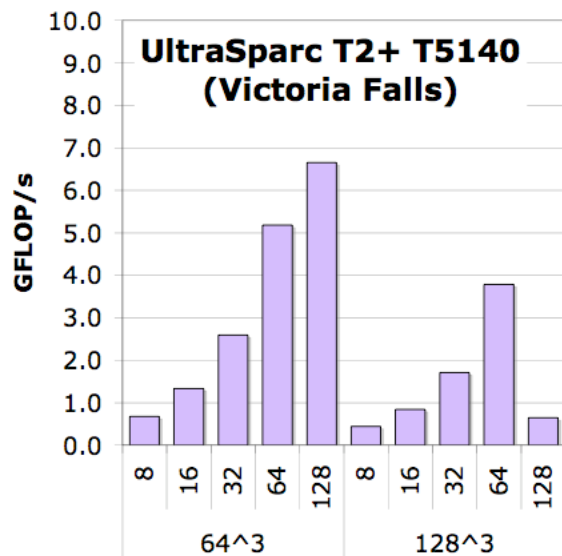
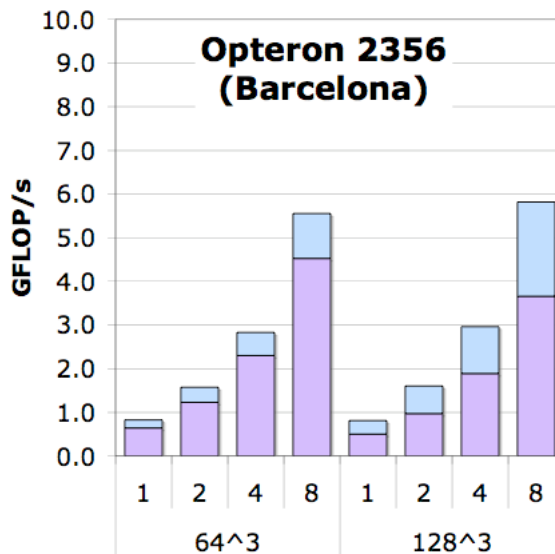
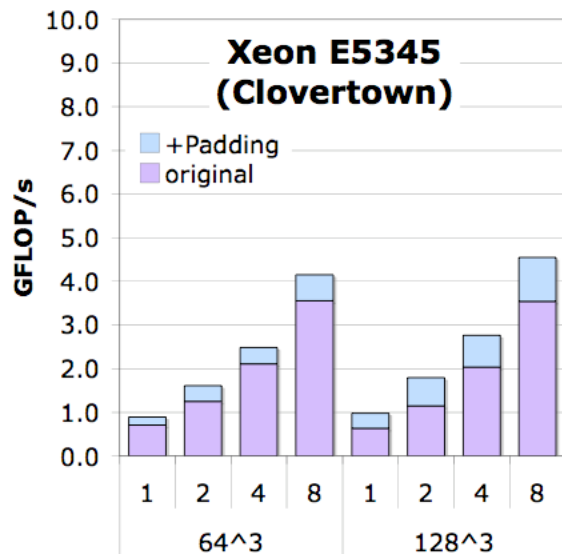
LBMHD Performance

(reference implementation)

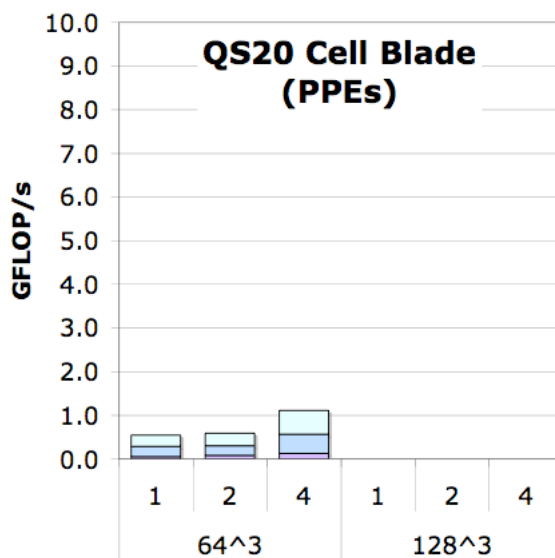
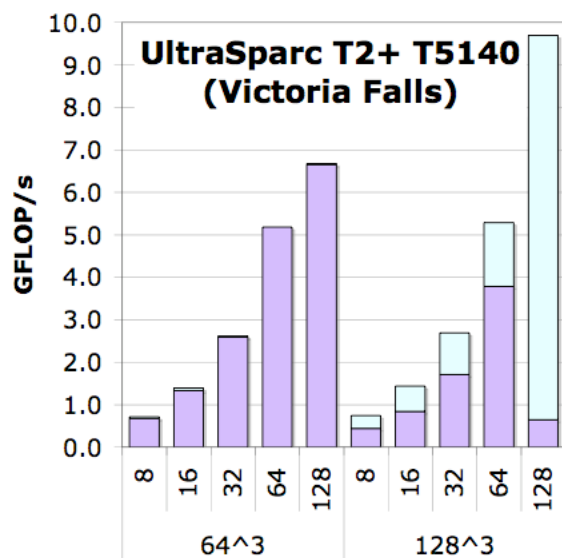
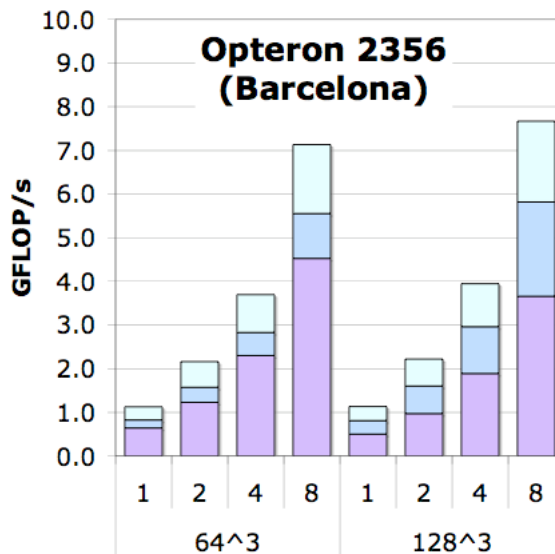
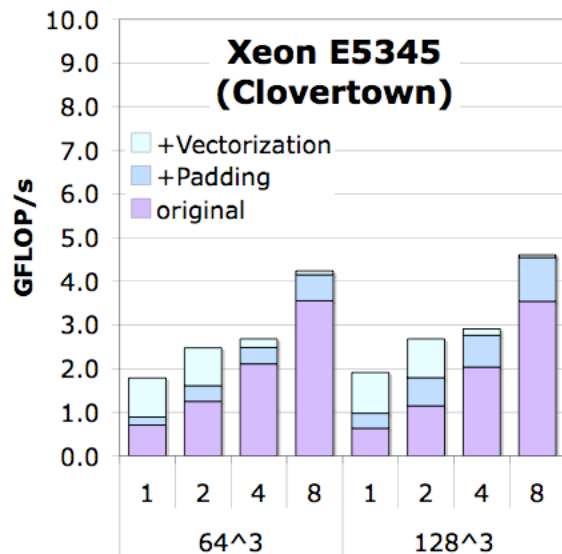
- ❖ Superscalar performance is surprisingly good given the complexity of the memory access pattern.
- ❖ Cell PPE performance is abysmal.



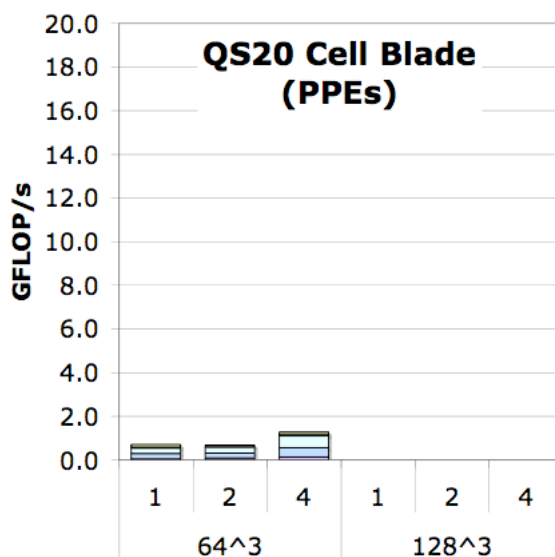
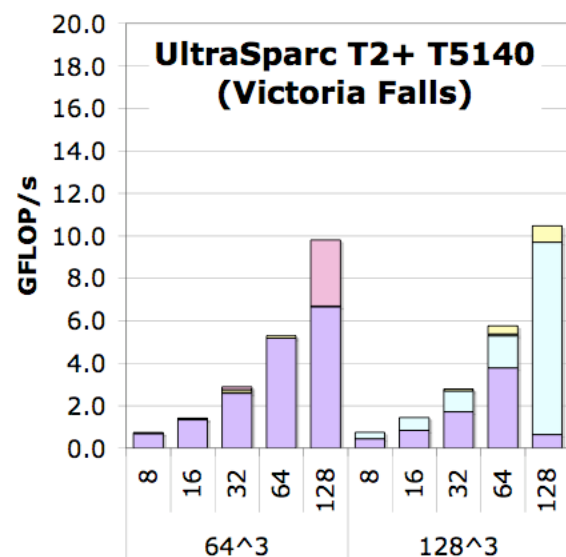
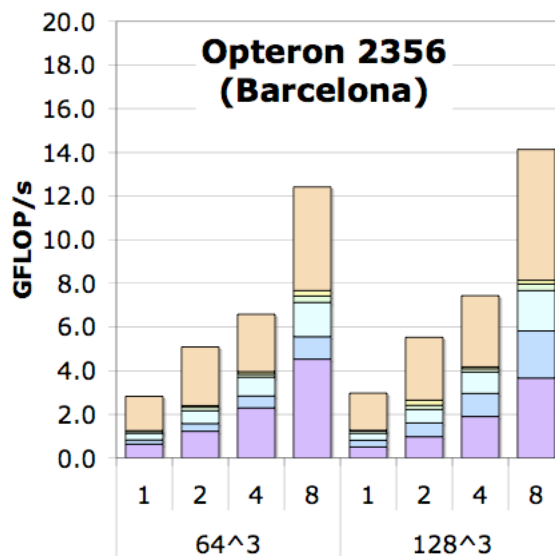
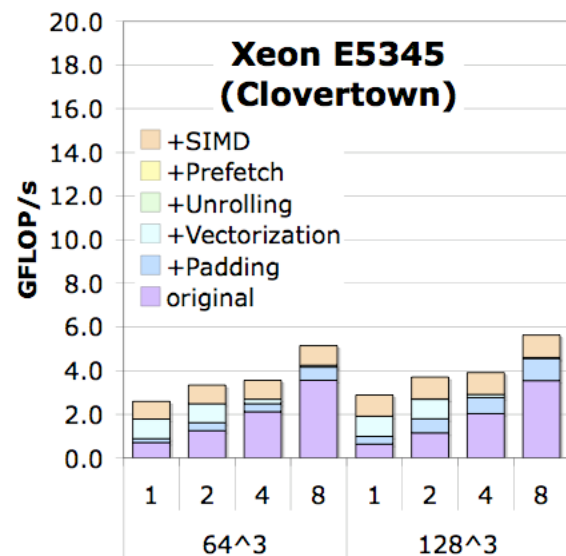
LBMHD Performance (array padding)



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



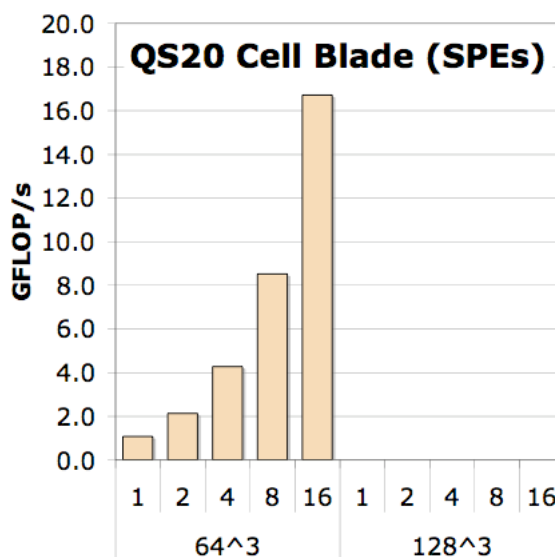
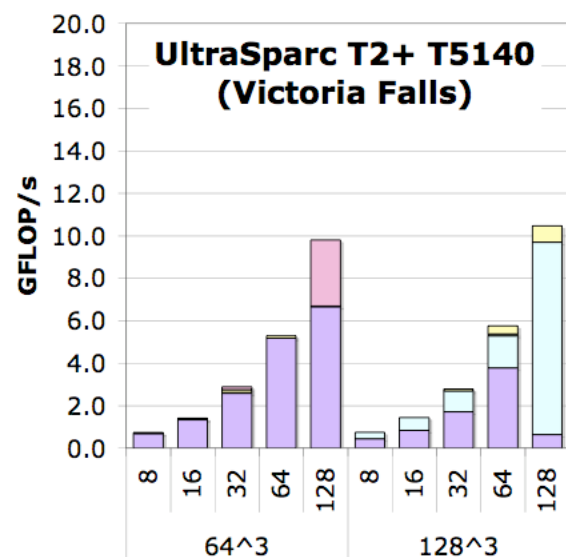
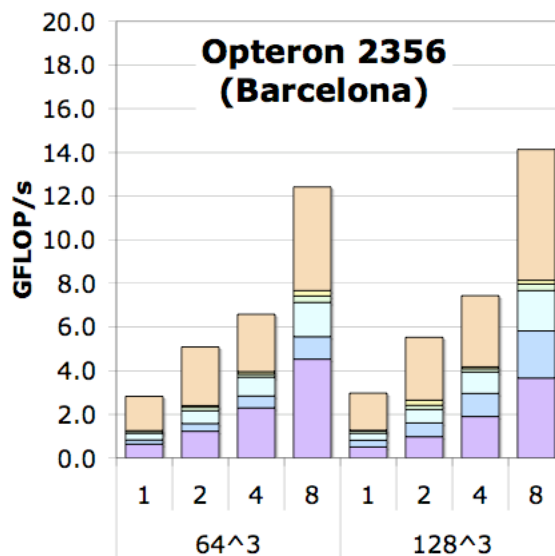
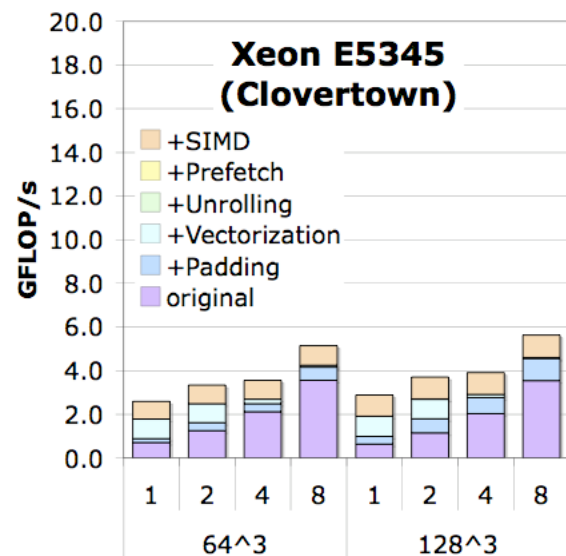
- ❖ LBMHD touches > 150 arrays.
- ❖ Most TLBs have << 128 entries.
- ❖ Vectorization technique creates a vector of points that are being updated.
- ❖ Loops are interchanged and strip mined
- ❖ **Exhaustively** search for the optimal “**vector length**” that balances page locality with L1 cache misses.



- ❖ **Heuristic-based** software prefetching
- ❖ **Exhaustive** search for low-level optimizations
- ❖ Loop unrolling/reordering
- ❖ SIMDization
- ❖ Cache Bypass increases arithmetic intensity by 50%
- ❖ Small TLB pages on VF

LBMHD Performance

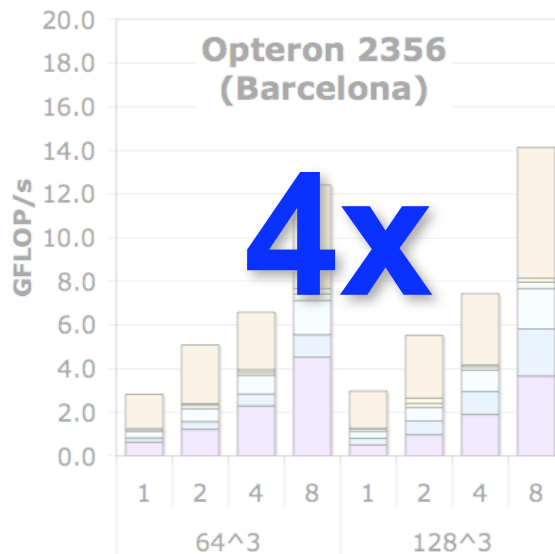
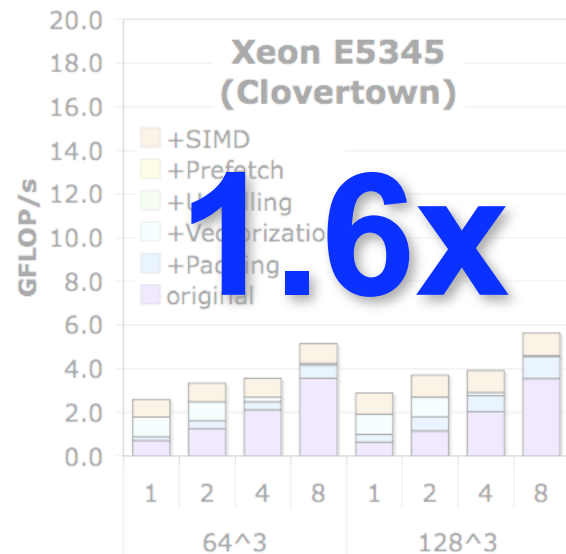
(Cell SPE implementation)



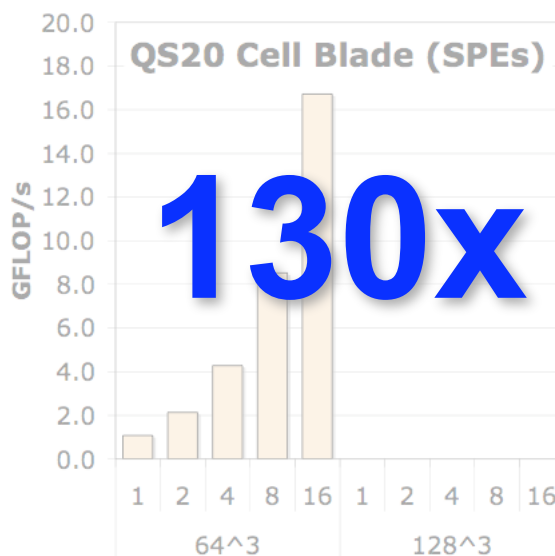
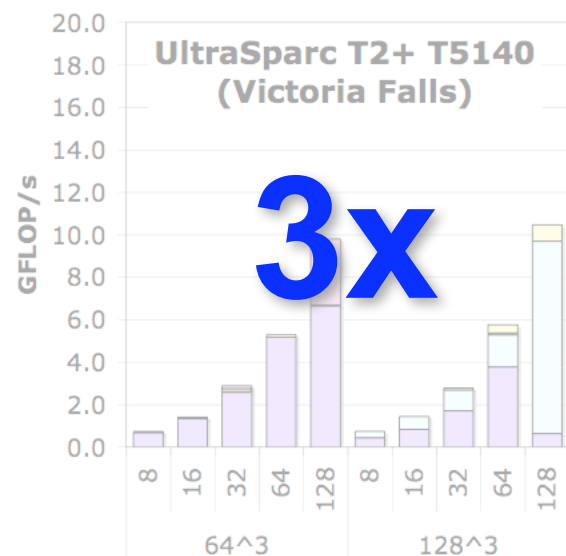
- ❖ We can write a local store implementation and run it on the Cell SPEs.
- ❖ Ultimately, Cell's weak DP hampers performance

LBMHD Performance

(speedup for largest problem)

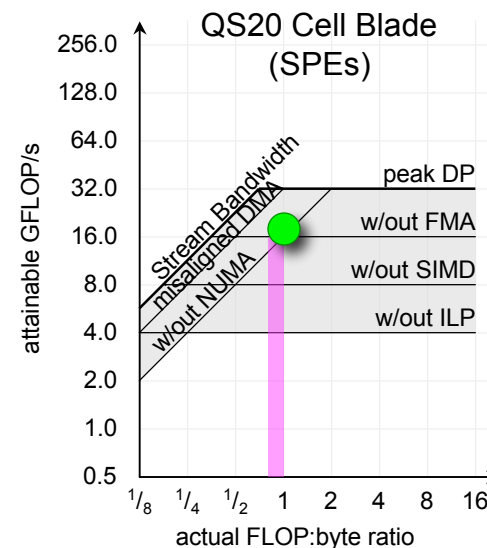
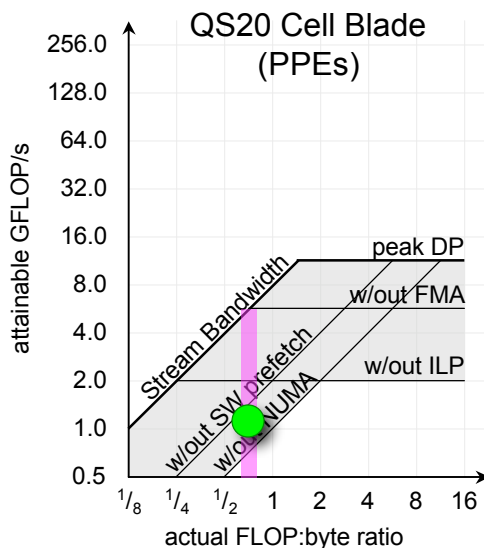
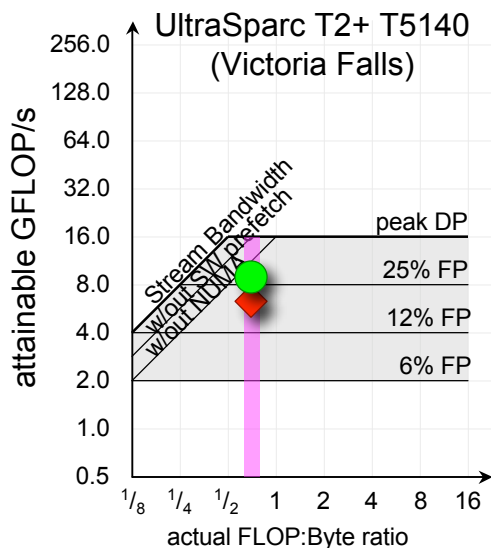
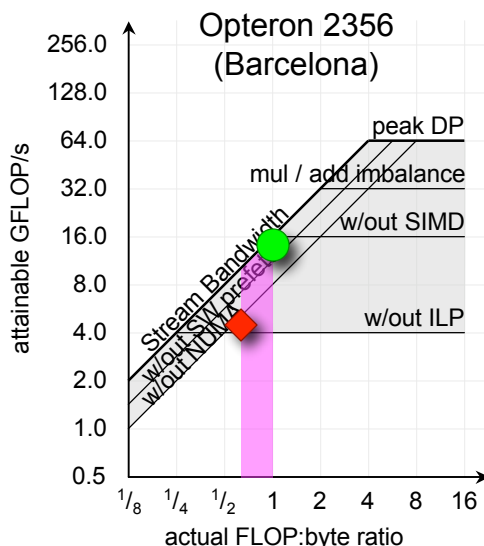
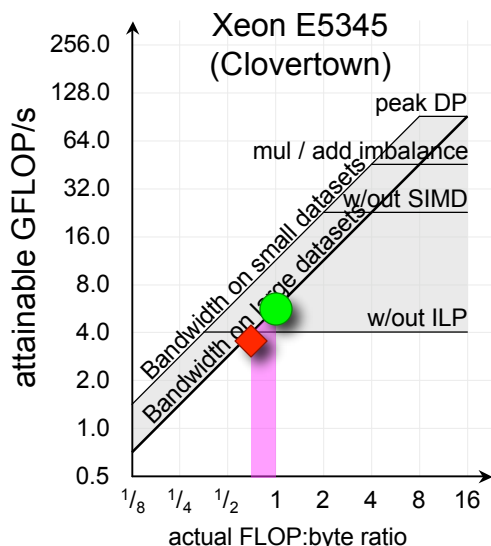


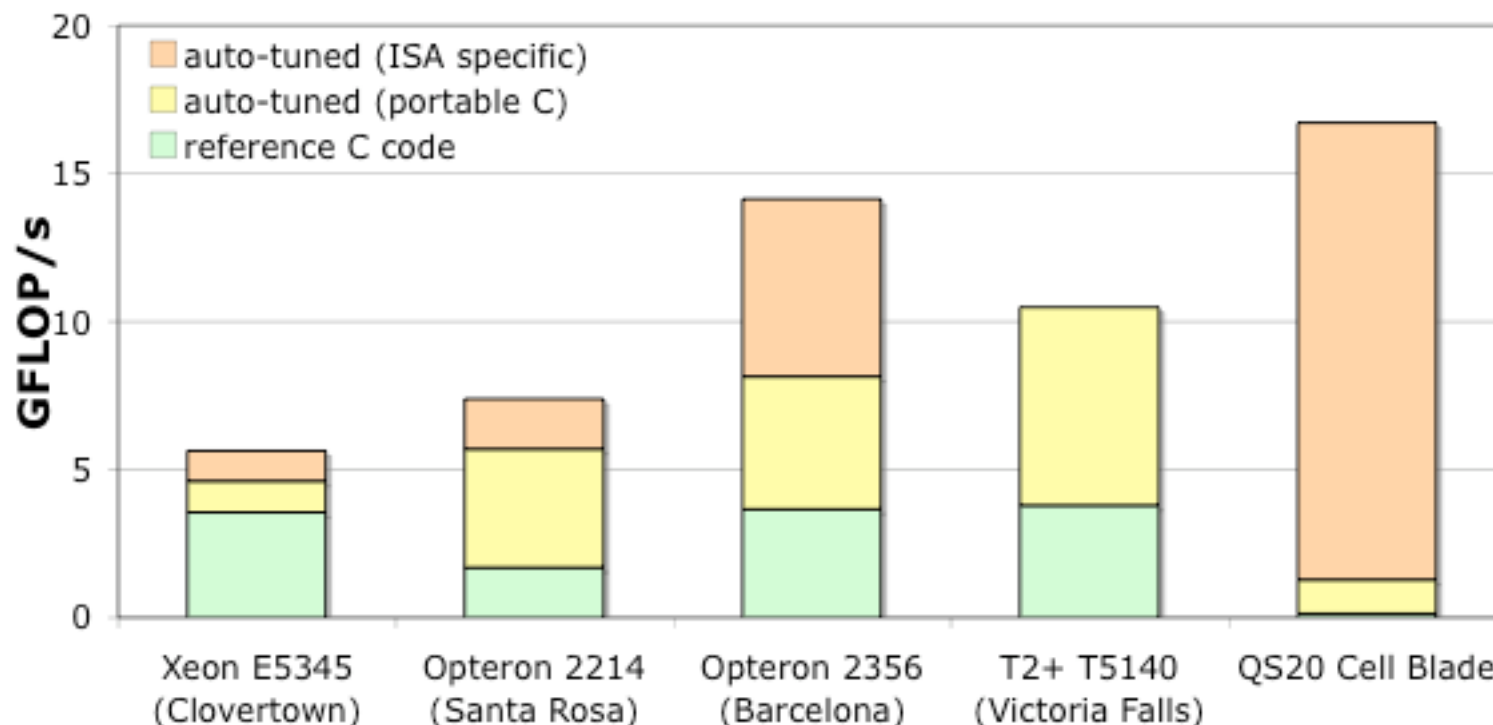
- ❖ We can write a local store implementation and run it on the Cell SPEs.
- ❖ Ultimately, Cell's weak DP hampers performance



LBMHD Performance (vs. Roofline)

- ❖ Most architectures reach their roofline bound performance
- ❖ Clovertown's snoop filter is ineffective.
- ❖ Niagara suffers from instruction mix issues
- ❖ Cell PPEs are latency limited
- ❖ Cell SPEs are compute-bound





- ❖ Reference code is clearly insufficient
- ❖ Portable C code is insufficient on Barcelona and Cell
- ❖ Cell gets all its performance from the SPEs
 - despite only 2x the area, and 2x the peak DP FLOPs

Auto-tuning

Sparse Matrix-Vector Multiplication

Overview

Multicore SMPs

The Roofline Model

Auto-tuning LBMHD

Auto-tuning SpMV

Summary

Future Work

Chapter 8

❖ Sparse Matrix

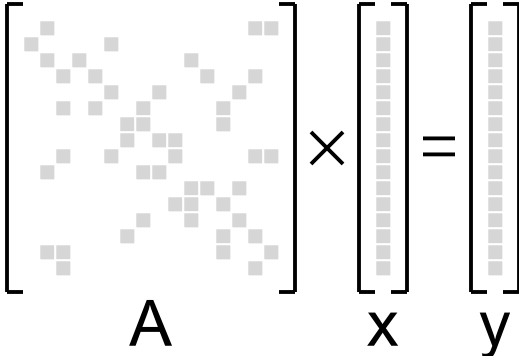
- Most entries are 0.0
- Performance advantage in only storing/operating on the nonzeros
- Requires significant meta data

❖ Evaluate: $y = Ax$

- A is a sparse matrix
- x & y are dense vectors

❖ Challenges

- Difficult to exploit ILP(bad for superscalar),
- Difficult to exploit DLP(bad for SIMD)
- Irregular memory access to source vector
- Difficult to load balance


$$\begin{bmatrix} \text{Sparse Matrix} \end{bmatrix} \times \begin{bmatrix} \text{Dense Vector } x \end{bmatrix} = \begin{bmatrix} \text{Dense Vector } y \end{bmatrix}$$

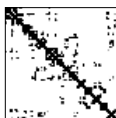
$A \quad x \quad y$

2K x 2K Dense matrix
stored in sparse format



Dense

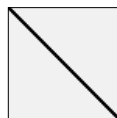
Well Structured
(sorted by nonzeros/row)



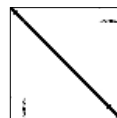
Protein



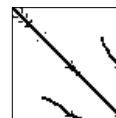
FEM /
Spheres



FEM /
Cantilever



Wind
Tunnel



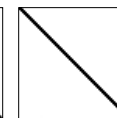
FEM /
Harbor



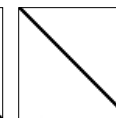
QCD



FEM /
Ship



Economics



Epidemiology

Poorly Structured
hodgepodge



FEM /
Accelerator



Circuit



webbase

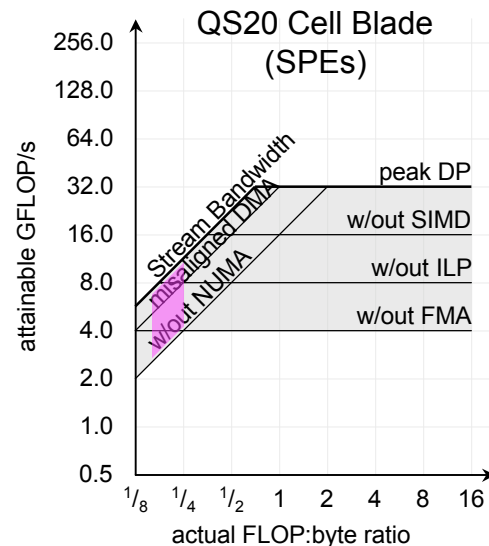
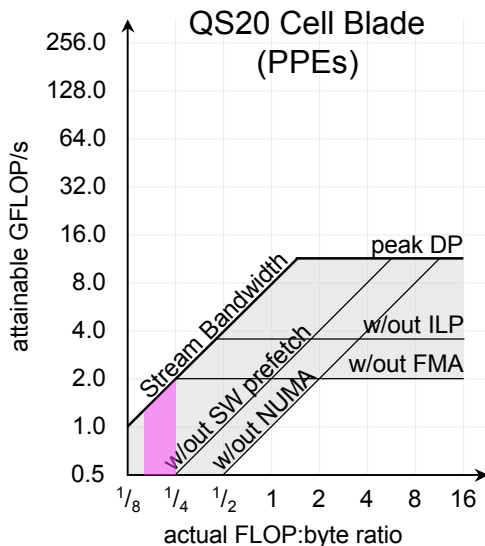
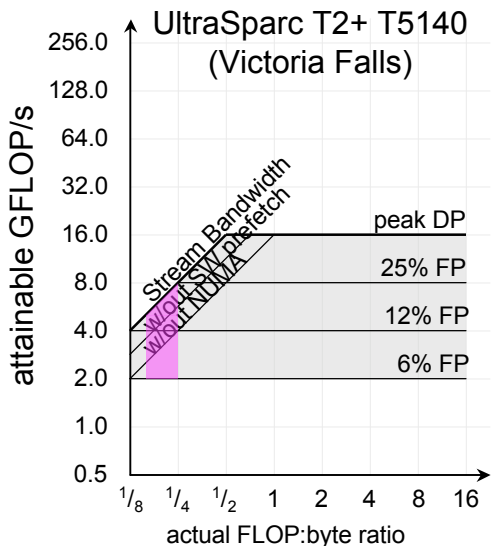
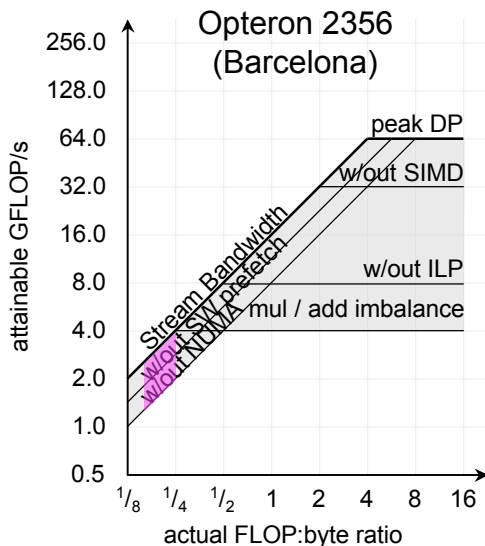
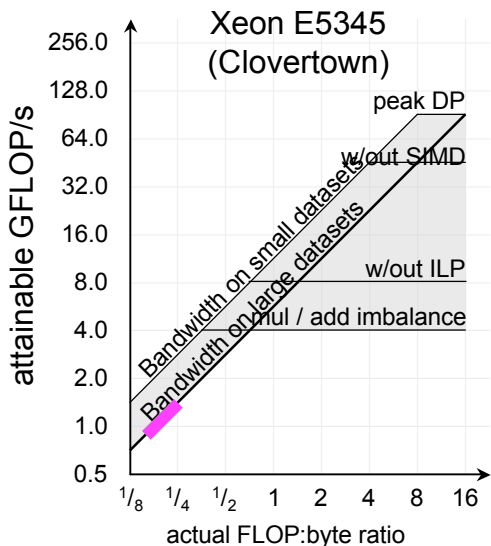
Extreme Aspect Ratio
(linear programming)



LP

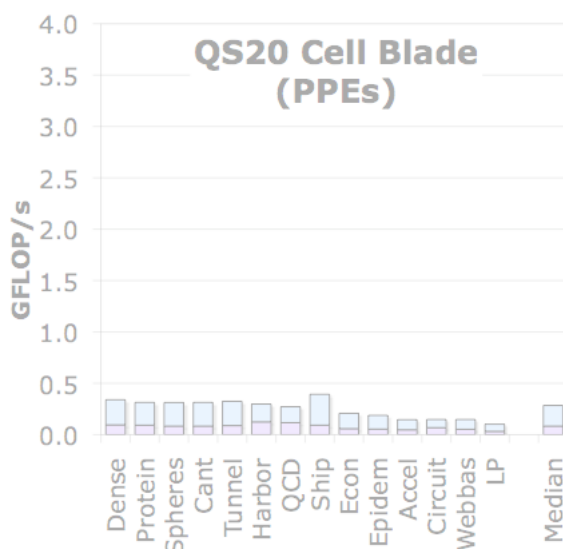
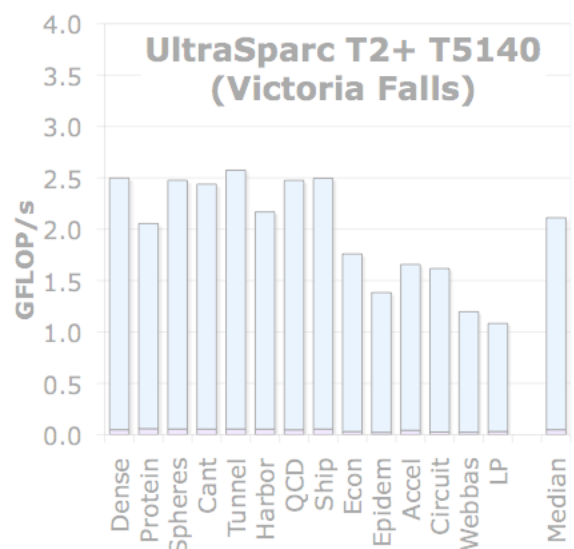
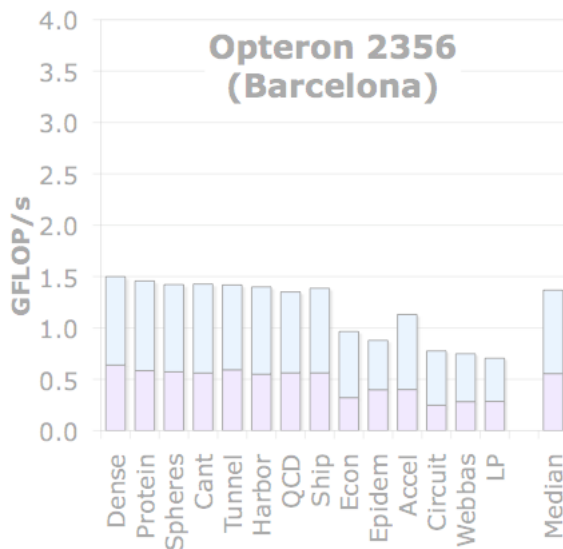
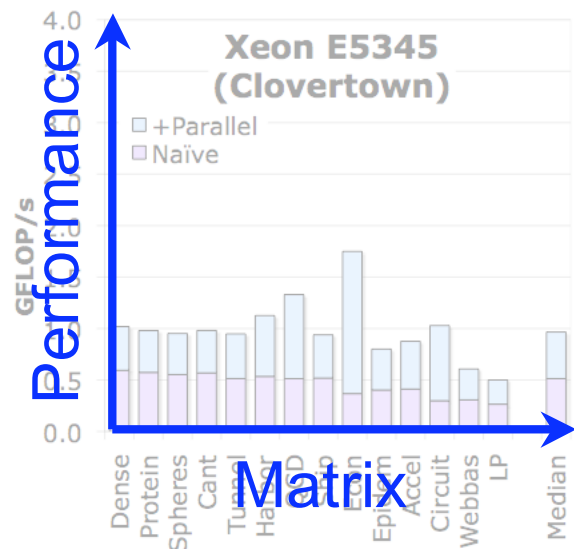
- ❖ Pruned original SPARSITY suite down to 14
- ❖ none should fit in cache
- ❖ Subdivided them into 4 categories
- ❖ Rank ranges from 2K to 1M

- ❖ Reference SpMV implementation has an AI of **0.166**, but can readily exploit FMA.
- ❖ The best we can hope for is an AI of **0.25** for non-symmetric matrices.
- ❖ All architectures are memory-bound, but some may need in-core optimizations



SpMV Performance

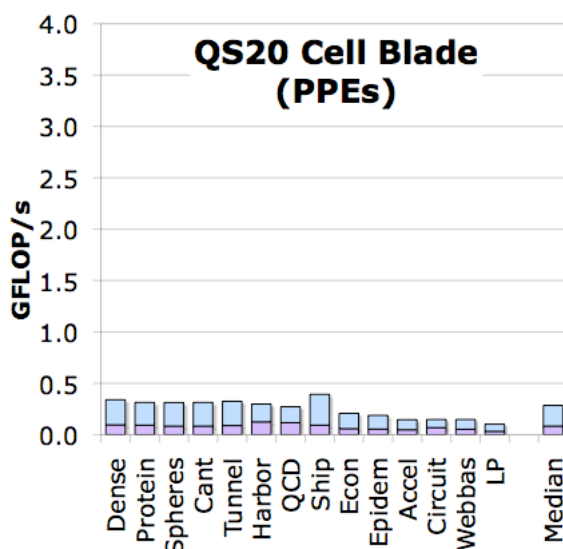
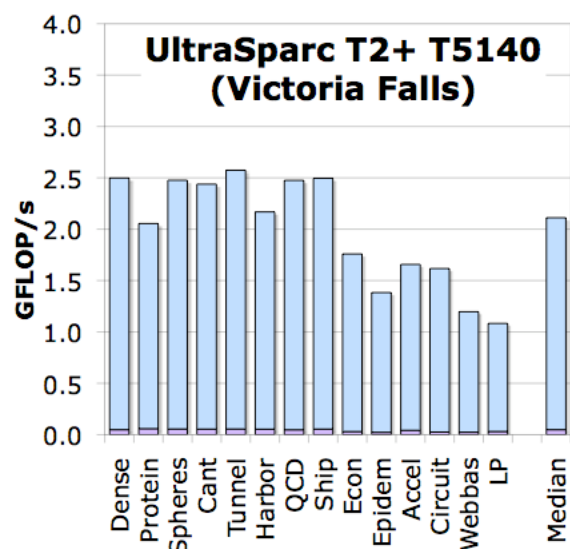
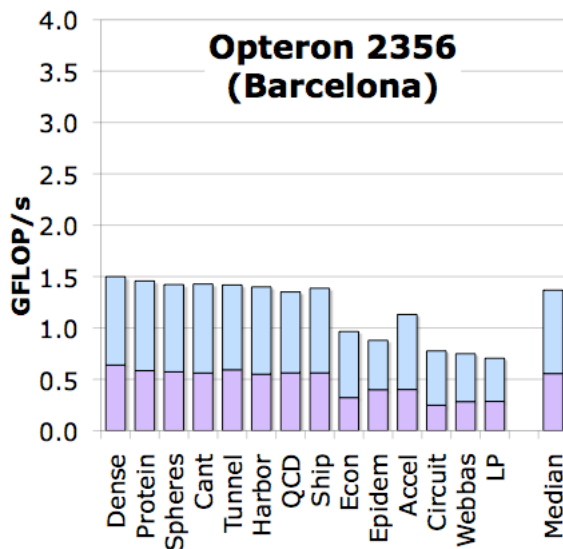
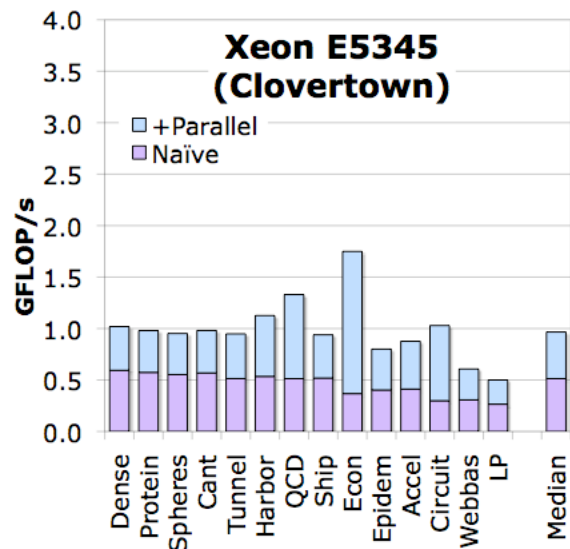
(Reference Implementation)



- ❖ Reference implementation is CSR.
- ❖ Simple parallelization by rows balancing nonzeros.
- ❖ No implicit NUMA exploitation
- ❖ Despite superscalar's use of 8 cores, they see little speedup.
- ❖ Niagara and PPEs show near linear speedups

SpMV Performance

(Reference Implementation)

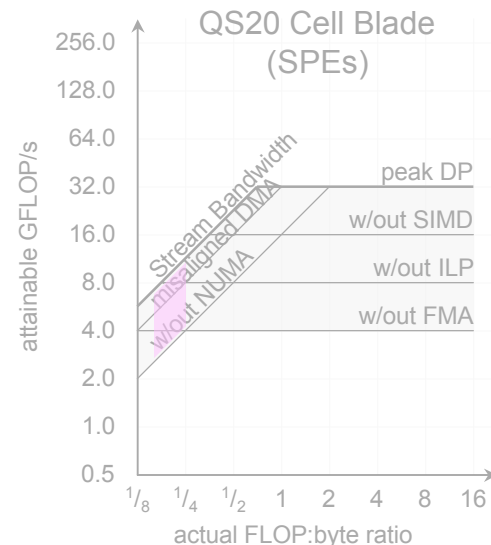
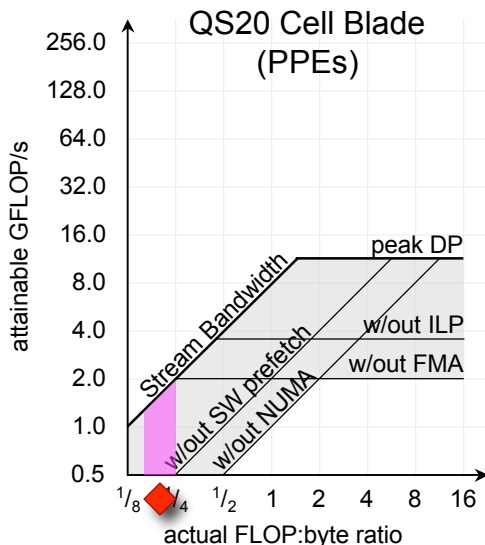
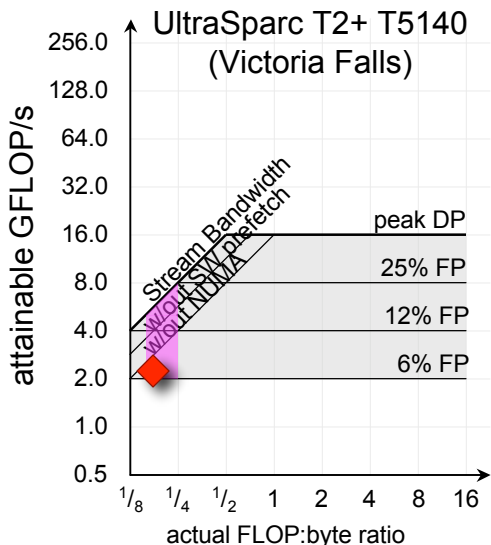
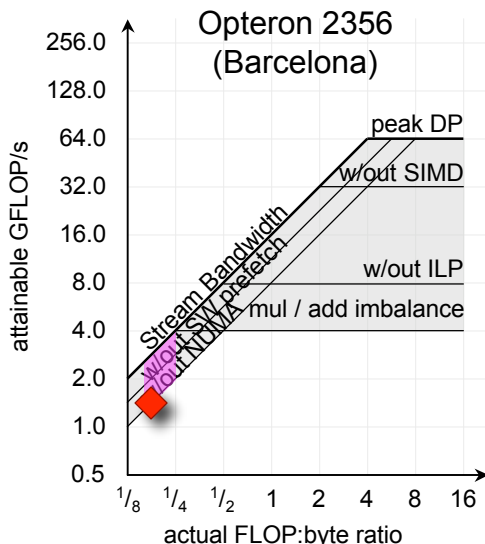
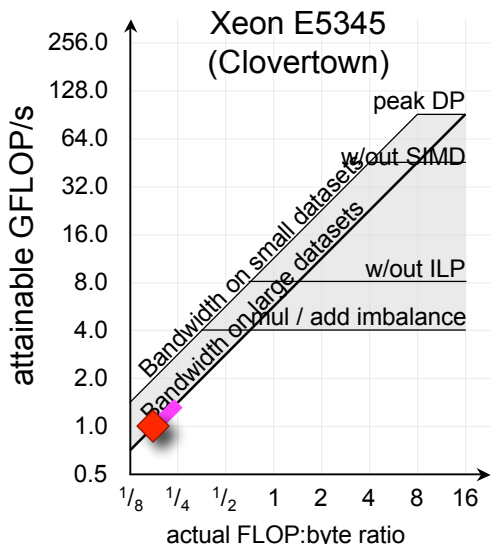


- ❖ Reference implementation is CSR.
- ❖ Simple parallelization by rows balancing nonzeros.
- ❖ No implicit NUMA exploitation
- ❖ Despite superscalar's use of 8 cores, they see little speedup.
- ❖ Niagara and PPEs show near linear speedups

SpMV Performance

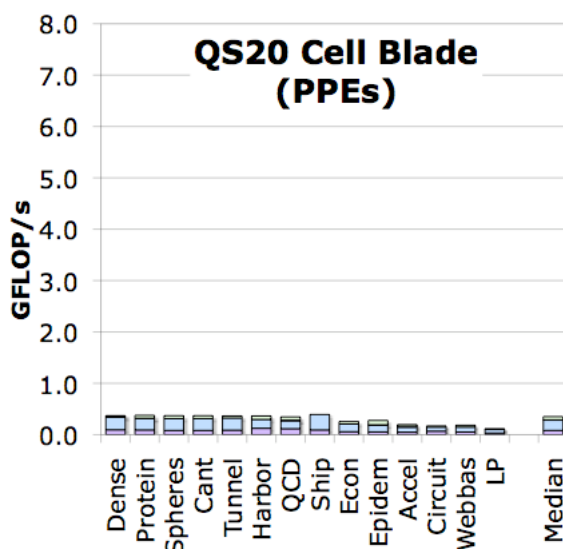
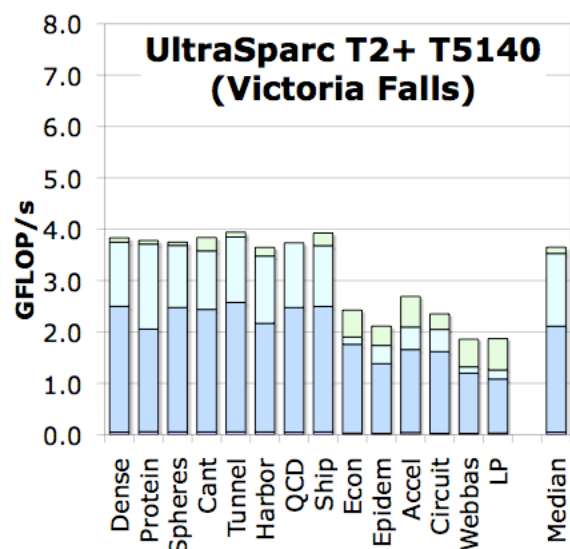
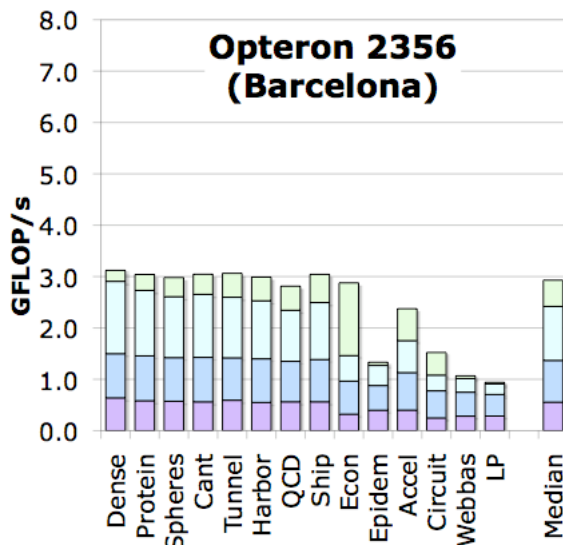
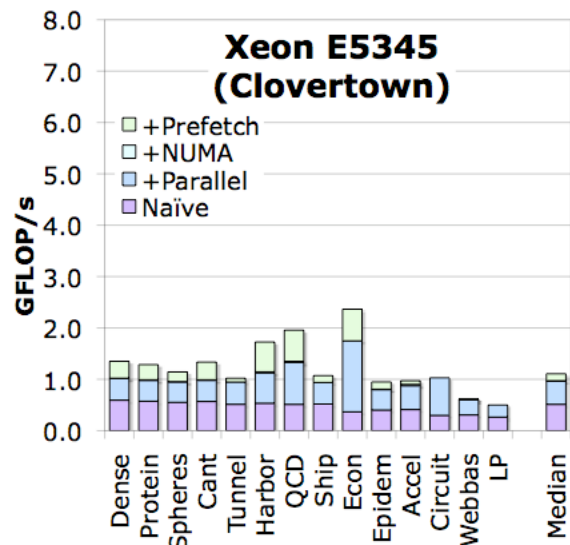
(Reference Implementation)

- ❖ Roofline for dense matrix in sparse format.
- ❖ Superscalars achieve bandwidth-limited performance
- ❖ Niagara comes very close to bandwidth limit.
- ❖ Clearly, NUMA and prefetching will be essential.



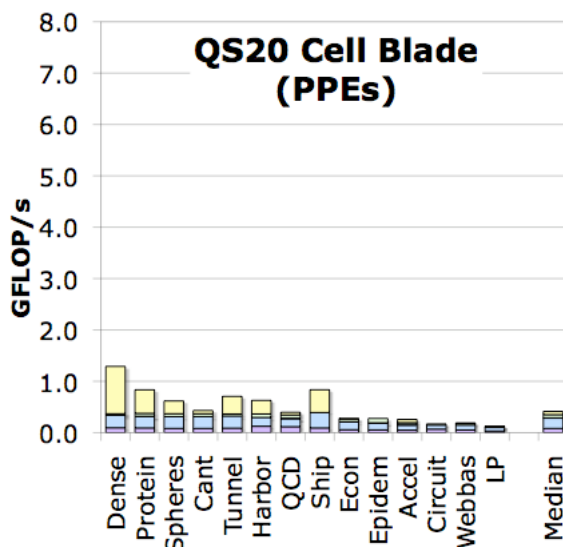
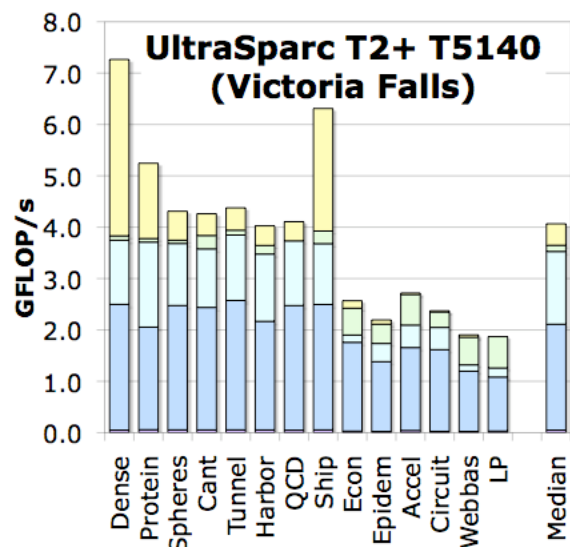
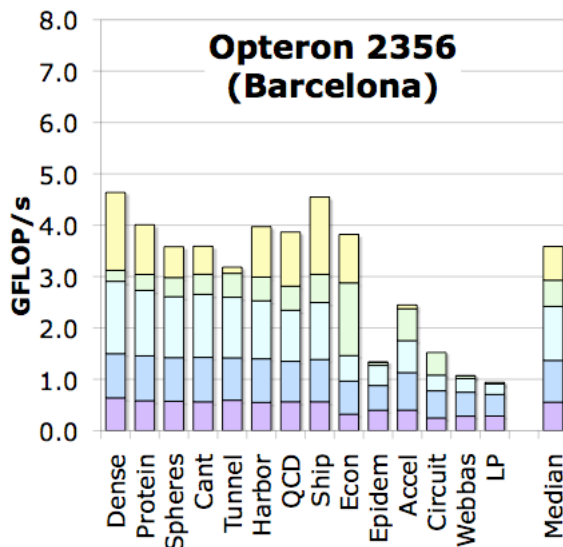
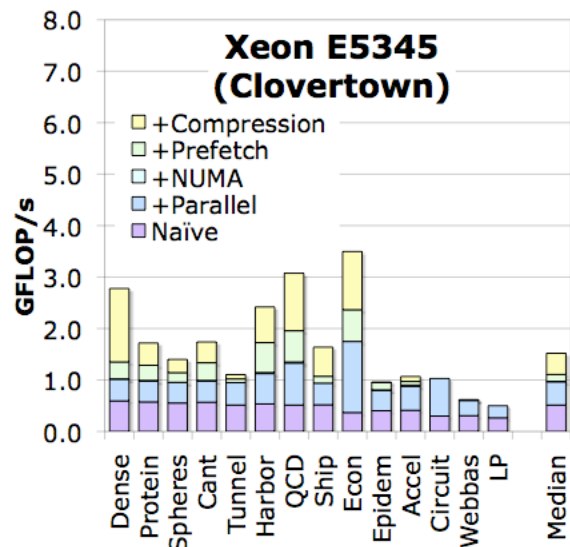
SpMV Performance

(NUMA and Software Prefetching)



- ❖ NUMA-aware allocation is essential on memory-bound NUMA SMPs.
- ❖ Explicit software prefetching can boost bandwidth and change cache replacement policies
- ❖ Cell PPEs are likely latency-limited.
- ❖ used **exhaustive** search

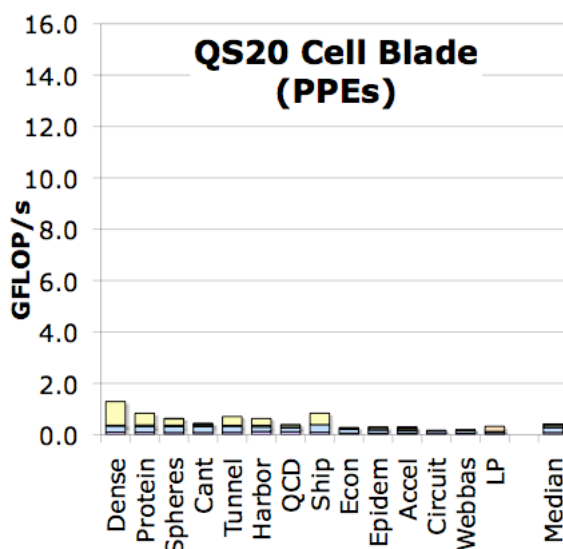
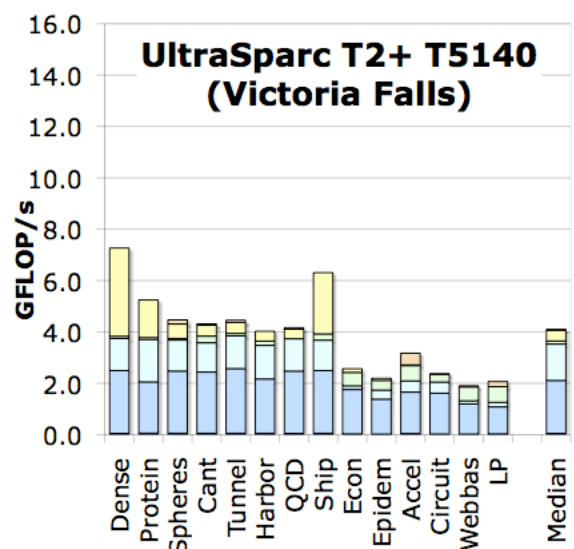
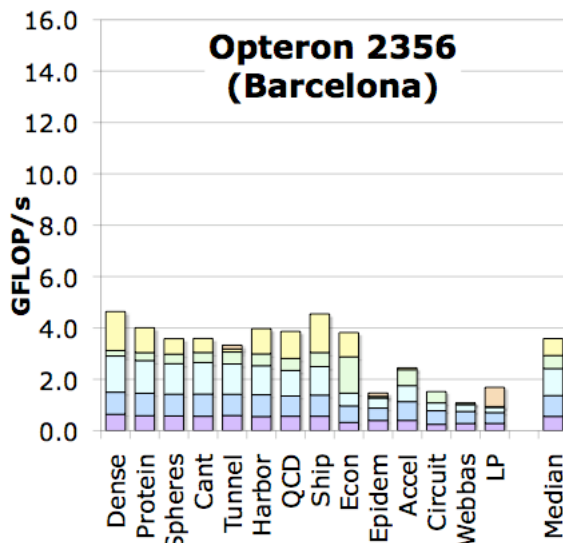
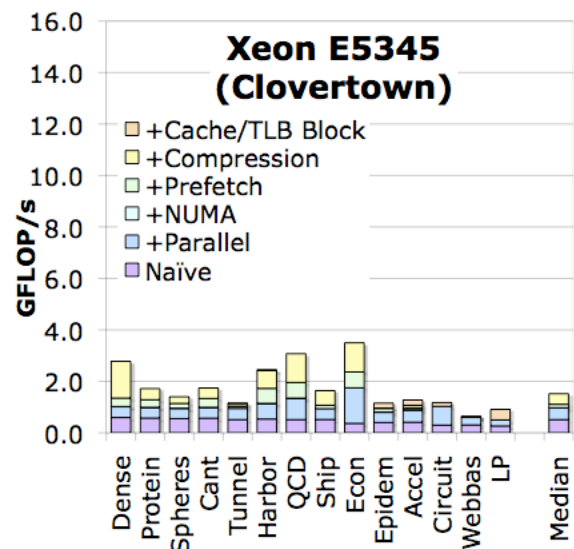
SpMV Performance (Matrix Compression)



- ❖ After maximizing memory bandwidth, the only hope is to minimize memory traffic.
- ❖ exploit:
 - register blocking
 - other formats
 - smaller indices
- ❖ Use a traffic minimization **heuristic** rather than search
- ❖ Benefit is clearly matrix-dependent.
- ❖ Register blocking enables efficient software prefetching (one per cache line)

SpMV Performance

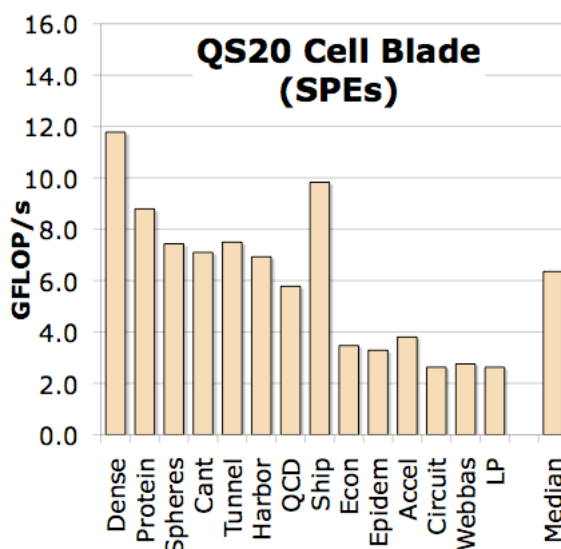
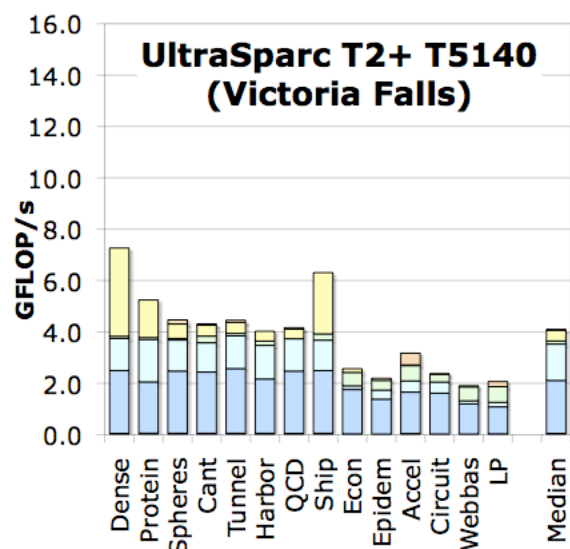
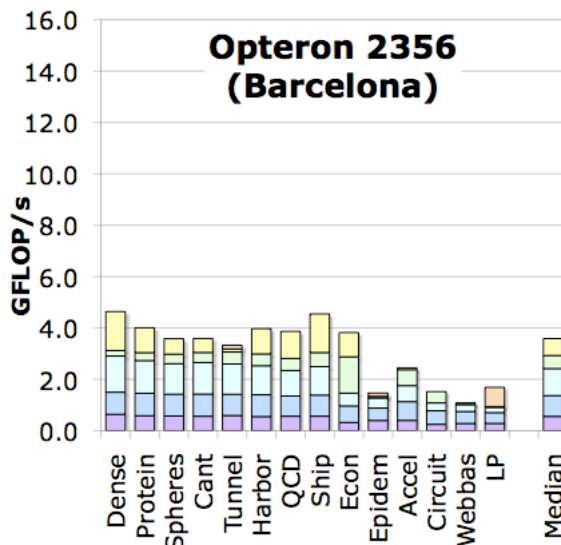
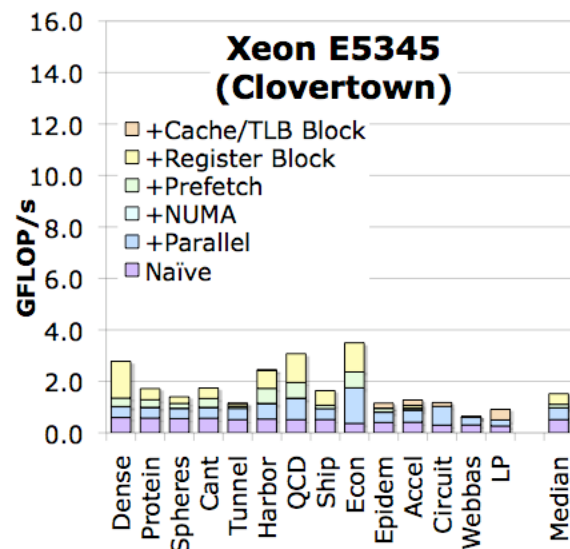
(Cache and TLB Blocking)



- ❖ Based on limited architectural knowledge, create **heuristic** to choose a good cache and TLB block size
- ❖ Hierarchically store the resultant blocked matrix.
- ❖ Benefit can be significant on the most challenging matrix

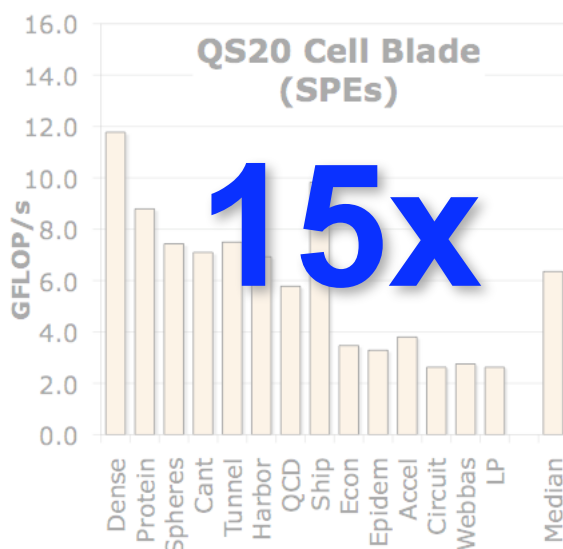
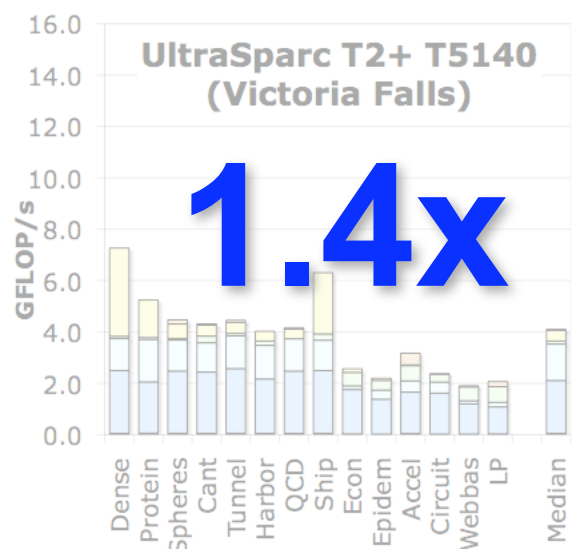
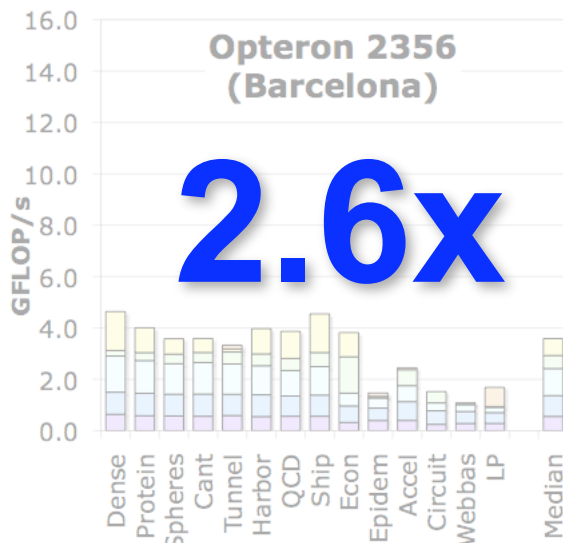
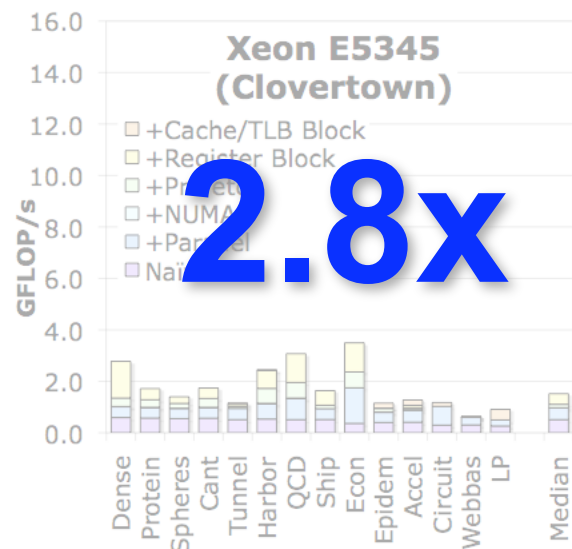
SpMV Performance

(Cell SPE implementation)



- ❖ Cache blocking can be easily transformed into local store blocking.
- ❖ With a few small tweaks for DMA, we can run a simplified version of the auto-tuner on Cell
 - BCOO only
 - 2x1 and larger
 - always blocks

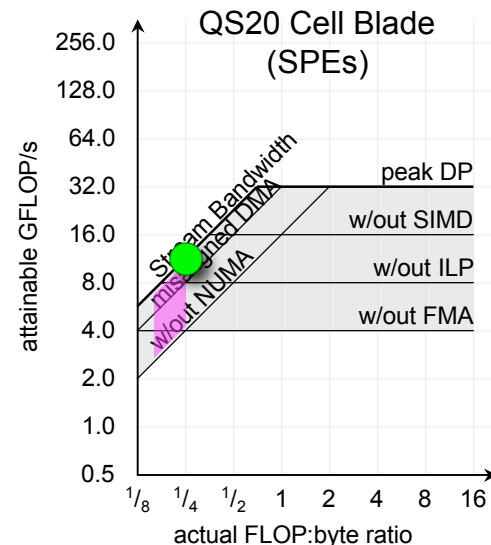
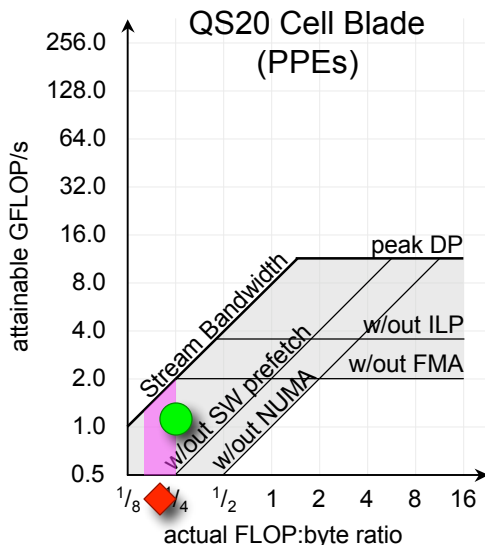
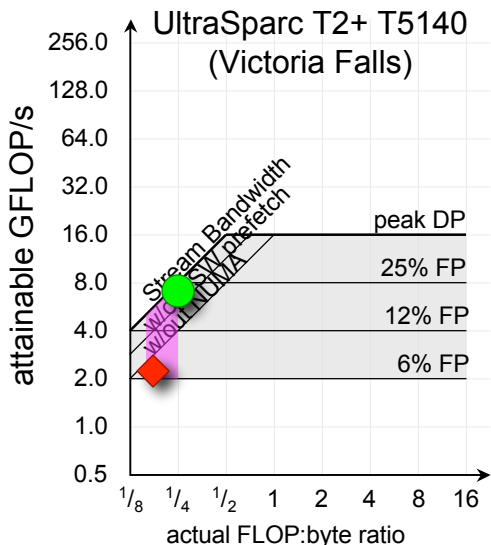
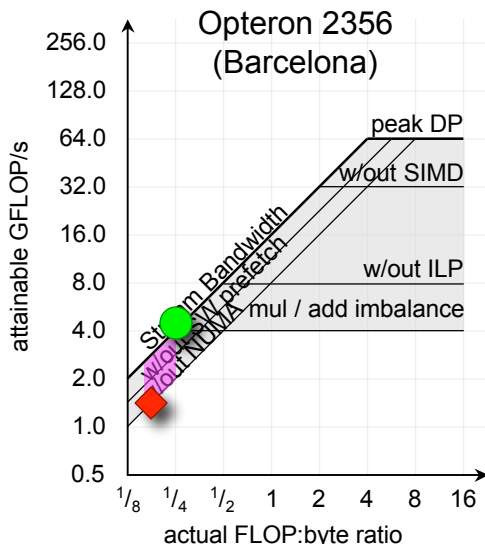
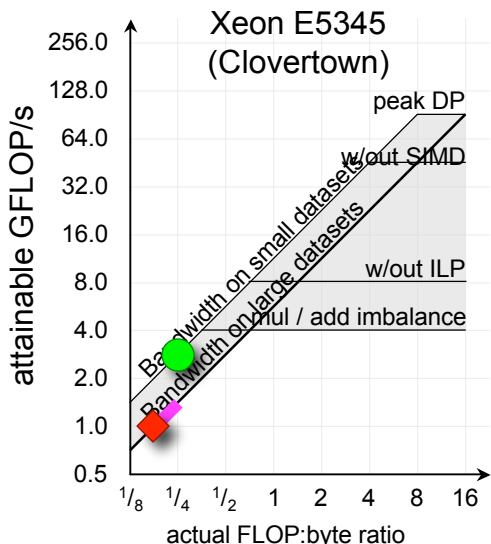
SpMV Performance (median speedup)



- ❖ Cache blocking can be easily transformed into local store blocking.
- ❖ With a few small tweaks for DMA, we can run a simplified version of the auto-tuner on Cell
 - BCOO only
 - 2x1 and larger
 - always blocks

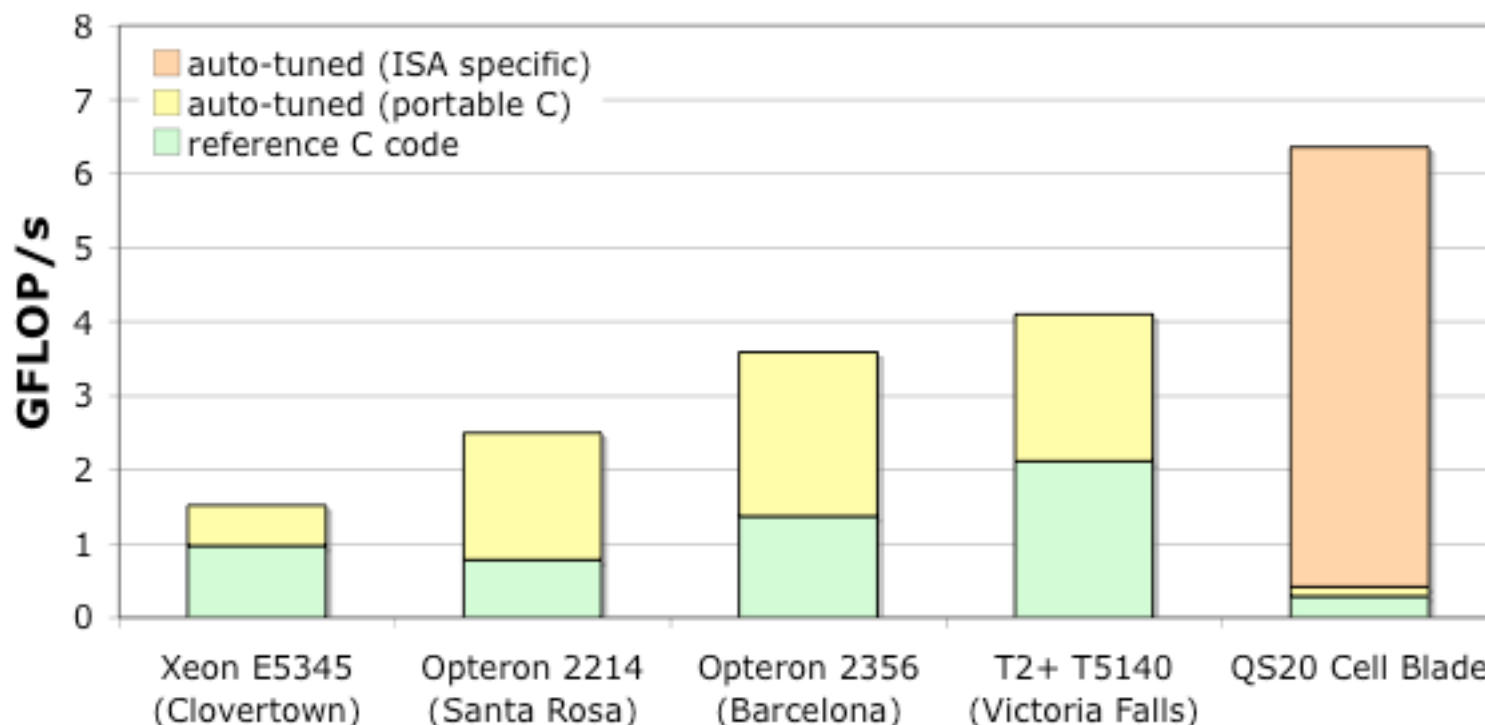
SpMV Performance (vs. Roofline)

- ❖ Roofline for **dense matrix in sparse format**.
- ❖ Compression improves AI
- ❖ Auto-tuning can allow us to slightly exceed Stream bandwidth (but not pin bandwidth)
- ❖ Cell PPEs perennially deliver poor performance



SpMV Performance

(summary)



- ❖ **Median SpMV performance**
- ❖ aside, unlike LBMHD, SSE was unnecessary to achieve performance
- ❖ Cell still requires a non-portable, ISA-specific implementation to achieve good performance.
- ❖ Novel SpMV implementations may require ISA-specific (SSE) code to achieve better performance.

Summary

Overview

Multicore SMPs

The Roofline Model

Auto-tuning LBMHD

Auto-tuning SpMV

Summary

Future Work

❖ **Introduced the Roofline Model**

- Apply bound and bottleneck analysis
- Performance and requisite optimizations are inferred visually

❖ **Extended auto-tuning to multicore**

- Fundamentally different from running auto-tuned serial code on multicore SMPs.
- Apply the concept to LBMHD and SpMV.

❖ **Auto-tuning LBMHD and SpMV**

- Multicore has had a transformative effect on auto-tuning.
(move from latency limited to bandwidth limited)
- Maximizing memory bandwidth and minimizing memory traffic is key.
- Compilers are reasonably effective at in-core optimizations,
but totally ineffective at cache and memory issues.
- Library or framework is a necessity in managing these issues.

❖ **Comments on architecture**

- Ultimately machines are bandwidth-limited without new algorithms
- Architectures with caches required significantly more tuning than the local store-based Cell

Future Directions in Auto-tuning

Overview

Multicore SMPs

The Roofline Model

Auto-tuning LBMHD

Auto-tuning SpMV

Summary

Future Work

Chapter 9

- ❖ Automatic generation of Roofline figures
 - Kernel oblivious
 - Select computational metric of interest
 - Select communication channel of interest
 - Designate common “optimizations”
 - Requires a benchmark

- ❖ Using performance counters to generate runtime Roofline figures
 - Given a real kernel, we wish to understand the bottlenecks to performance.
 - Much friendlier visualization of performance counter data

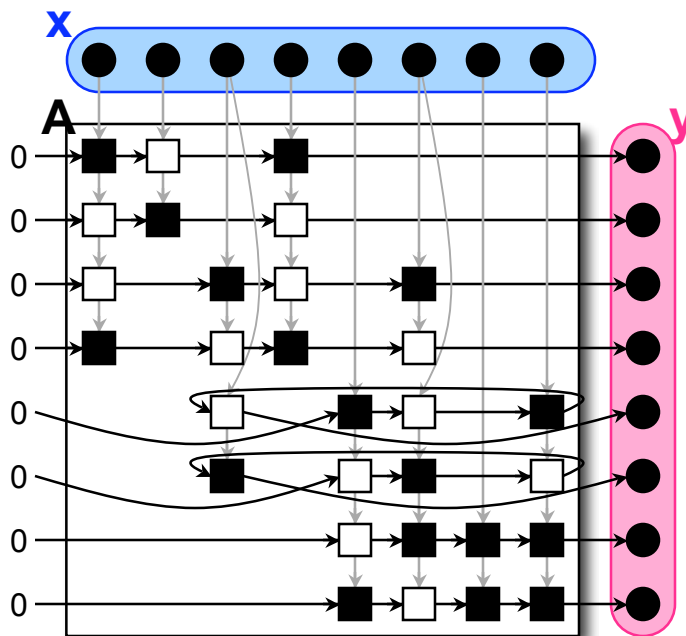
- ❖ Given the explosion in optimizations, exhaustive search is clearly not tractable.
- ❖ Moreover, heuristics require extensive architectural knowledge.
- ❖ In our SC08 work, we tried a greedy approach (one optimization at a time)
- ❖ We could make it iterative or, we could make it look like steepest descent (with some local search)

- ❖ We could certainly auto-tune other individual kernels in any motif, but this requires building a kernel-specific auto-tuner
 - ❖ However, we should strive for motif-wide auto-tuning.
 - ❖ Moreover, we want to decouple data type (e.g. double precision) from the parallelization structure.
-
1. A motif description or pattern language for each motif.
 - e.g. taxonomy of structured grids + code snippet for stencil
 - write auto-tuner parses these, and produces optimized code.
 2. A series of DAG rewrite rules for each motif.

Rules allow:

 - Insertion of additional nodes
 - Duplication of nodes
 - Reordering

- ❖ Consider SpMV
- ❖ In FP, each node in the DAG is a MAC
- ❖ DAG makes locality explicit (e.g. local store blocking)
- ❖ BCSR adds zeros to the DAG
- ❖ We can cut edges and reconnect them to enable parallelization.
- ❖ We can reorder operations
- ❖ Any other data type/node type conforming to these rules can reuse all our auto-tuning efforts



Acknowledgments

❖ **Berkeley ParLab**

- Thesis Committee: David Patterson, Kathy Yelick, Sara McMains
- BeBOP group: Jim Demmel, Kaushik Datta, Shoaib Kamil, Rich Vuduc, Rajesh Nishtala, etc...
- Rest of ParLab

❖ **Lawrence Berkeley National Laboratory**

- FTG group: Lenny Oliker, John Shalf, Jonathan Carter, ...

❖ **Hardware Donations and Remote Access**

- Sun Microsystems
- IBM
- AMD
- FZ Julich
- Intel

This research was supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231, by Microsoft and Intel funding through award #20080469, and by matching funding by U.C. Discovery through award #DIG07-10227.

Questions?