Performance Optimization and Auto-tuning

Samuel Williams[1]

Jonathan Carter[1], Richard Vuduc[3], Leonid Oliker[1], John Shalf[1],
Katherine Yelick[1,2], James Demmel[1,2]

[1]Lawrence Berkeley National Laboratory
[2]University of California Berkeley
[3]Georgia Institute of Technology

SWWilliams@lbl.gov

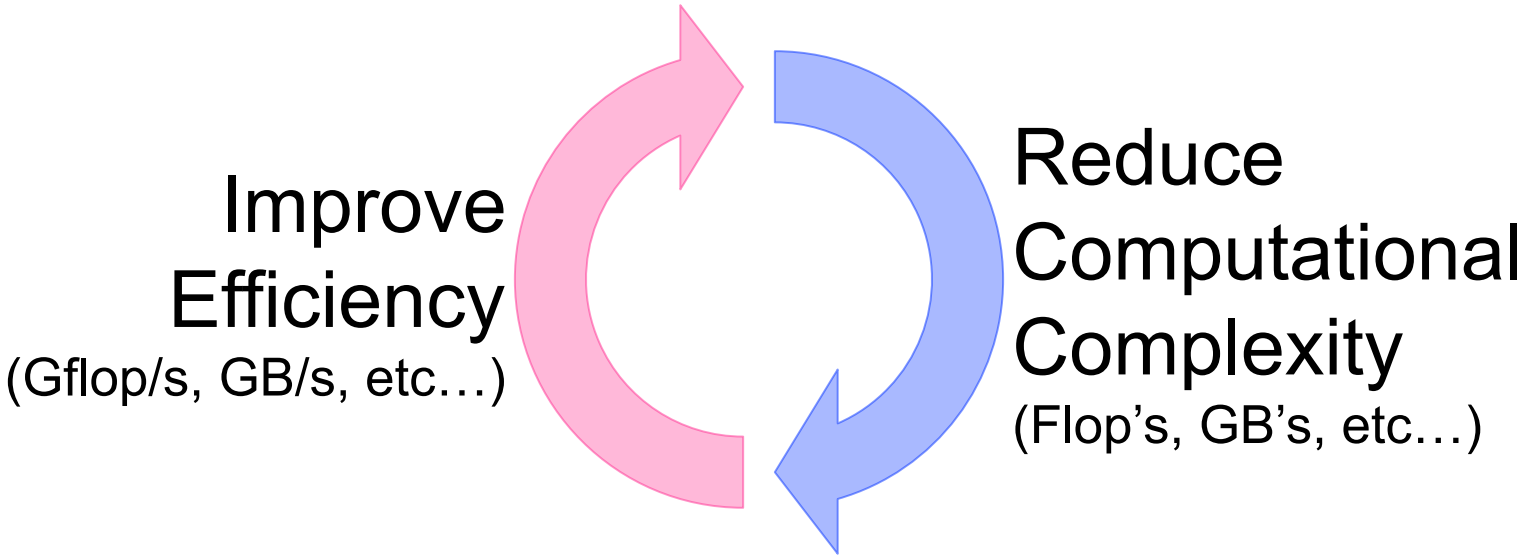
Outline

- Fundamentals
- Software Solutions to Challenges
  - Sequential Programming Model
  - Shared Memory World
  - Message Passing World
- Optimizing across an evolving hardware base
  - Automating Performance Tuning (auto-tuning)
  - Example 1: LBMHD
  - Example 2: SpMV
- Summary

Fundamentals

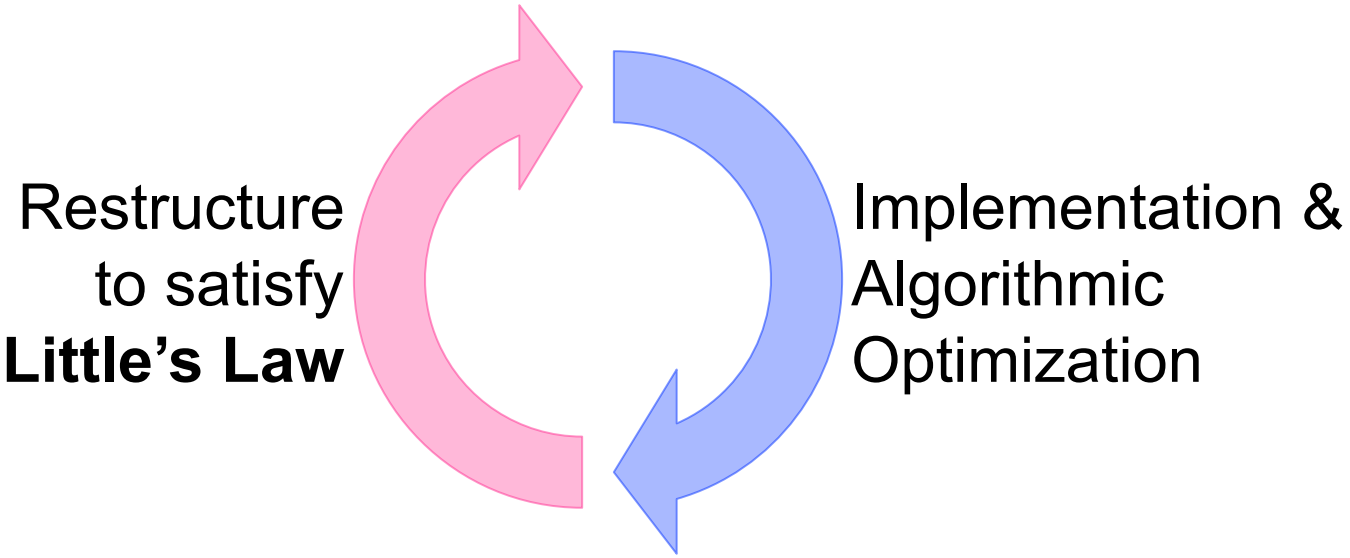
Performance Optimization: Contending Forces

- Contending forces of Efficiency and Computational Complexity
- We improve time to solution by improving throughput (efficiency) and reducing computational complexity



Performance Optimization: Contending Forces

- Contending forces of Efficiency and Computational Complexity
- We improve time to solution by improving throughput (efficiency) and reducing computational complexity



- In practice, we're willing to sacrifice one in order to improve the time to solution.

Basic Efficiency Quantities

- At all levels of the system (register files through networks), there are Three Fundamental (efficiency-oriented) Quantities:
  - **Latency** every operation requires time to execute (i.e. instruction, memory or network latency)
  - **Bandwidth** # of (parallel) operations completed per cycle (i.e. #FPUs, DRAM, Network, etc…)
  - **Concurrency** Total # of operations in flight

Little's Law

- Little's Law relates these three:

  Concurrency = Latency * Bandwidth

  - or -

  Effective Throughput = Expressed Concurrency / Latency

- This concurrency must be filled with parallel operations
- Can't exceed peak throughput with superfluous concurrency. (each channel has a maximum throughput)

Computational Complexity Quantities

- Complexity often expressed in terms of
  - **#Floating-point operations** (FLOPs)
  - **#Bytes** from (registers, cache, DRAM, network)

- Just as channels have throughput limits, kernels and algorithms can have lower bounds to complexity (traffic).

Architects, Mathematicians, Programmers

- Architects invent paradigms to improve (peak) throughput (efficiency?) and facilitate(?) Little's Law.
- Mathematicians invent new algorithms to improve performance by reducing (bottleneck) complexity or traffic
- As programmers, we must restructure algorithms and implementations to these new features.
- Often boils down to several key challenges:
  - Management of **data/task locality**
  - Management of **data dependencies**
  - Management of **communication**
  - Management of **variable and dynamic parallelism**

Software Solutions to Challenges

Challenges: Sequential

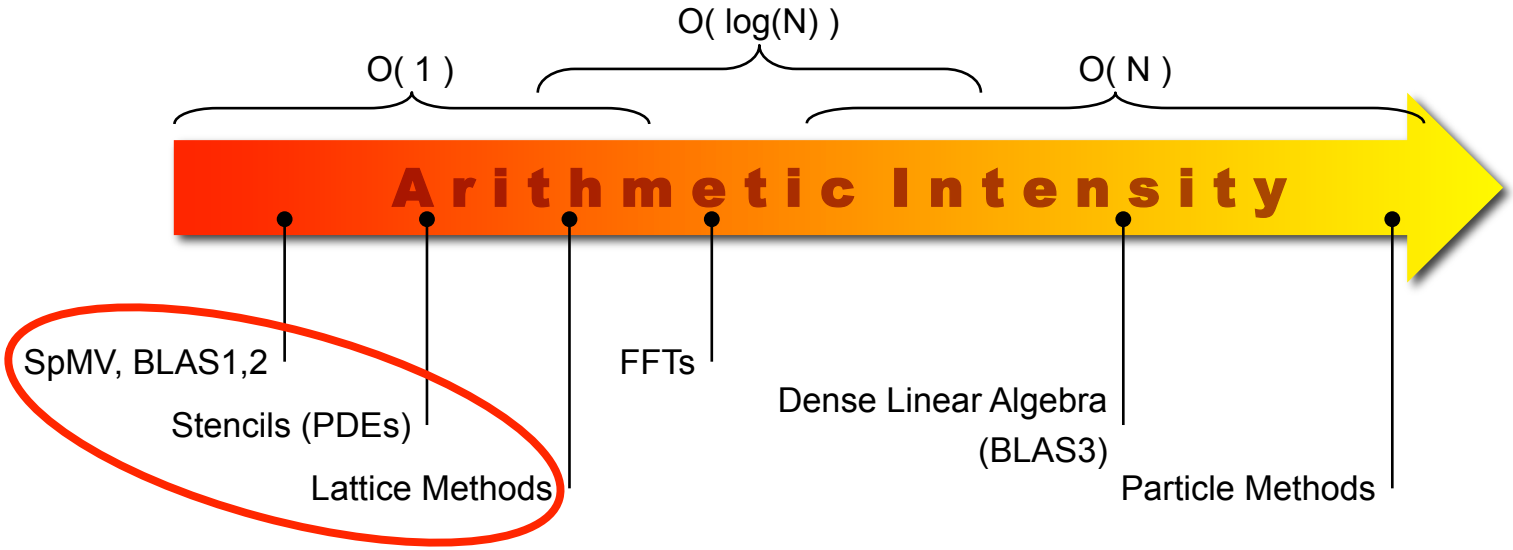
- *Even with only one thread, there is parallelism due to pipelining and SIMD*

- Data Dependencies:
  - HW (despite any out of order execution) manages data dependencies from ILP
  - user/compiler manages those from DLP (SIMDize only if possible)
- Data Locality:
  - compilers do a pretty good job of register file locality
  - For consumer apps, caches hide the complexity of attaining good on-chip locality fairly well
  - However, for performance-critical HPC apps, working sets can be so large and unpredictable, caches do poorly. When coupled with finite memory bandwidth, performance can suffer.

    ➔ cache block (reorder loops) or change data structures/types to improve arithmetic intensity.
- Communication (limited to processor-DRAM):
  - modern architectures predominately used HW stream prefetching to hide latency.

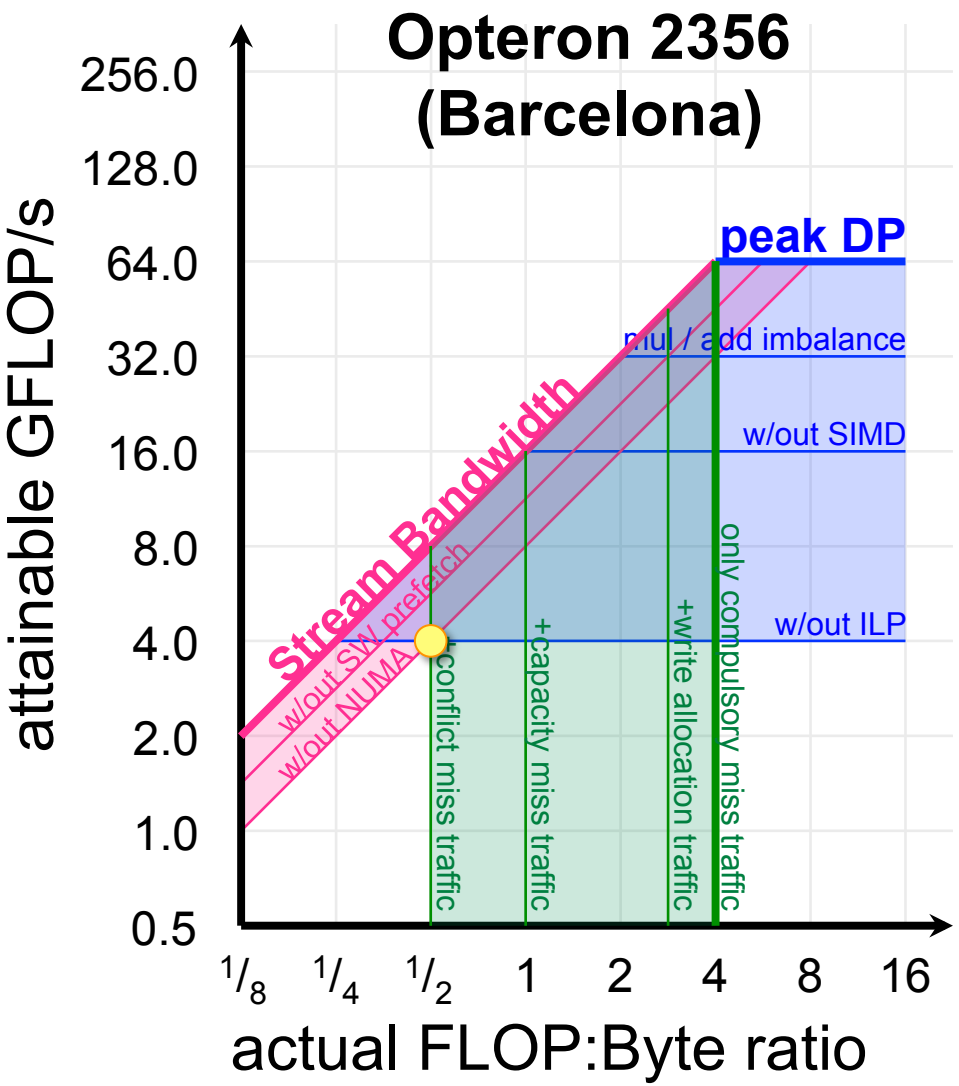
    ➔ structure memory access patterns into N unit stride streams
- Variable/Dynamic Parallelism:
  - OOO processors can mitigate the complexity of variable parallelism (ILP/DLP) within the instruction set so long as it occur within a ~few dozen instruction window

Arithmetic Intensity



- **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.

Roofline Model



- Visualizes how bandwidth, compute, locality, and optimization bound performance.
- Based on application characteristics, one can infer what needs to be done to improve performance.

Challenges: Shared Memory

- *Multiples threads in a shared memory environment*

- Data Dependencies:
  - no inherent support for managing data dependencies.

    ➔ Bulk Synchronous (barriers), locks/semaphores, atomics
- Data Locality:
  - Must manage NUMA and NUCA as well as on-/off-chip locality

    ➔ Use Linux affinity routines (parallelize accordingly),

    cache block (reorder loops), or change data structures/types.
- Communication:
  - Message aggregation, concurrency, throttling etc…
  - Ideally, HW/SW (cache coherency/GASNet) runtime should manage this:

    ➔ Use collectives and/or memory copies in UPC
- Variable/Dynamic Parallelism
  - variable TLP is a huge challenge

    ➔ Task queues (?)

Challenges: Message Passing

- *Multiples Processes in a message passing environment*

- Data Dependencies:
  - no inherent support for managing data dependencies.
  - user must express all data dependencies via send/recv/wait
- Data Locality:
  - No shared memory, so all communication is via MPI

    ➔ **User manages partitioning/replication of data at process level**
- Communication:
  - Message aggregation, concurrency throttling etc…
  - ideally, MPI should manage this

    ➔ **Use collectives, larger messages, limit the number of send/recv's at a time.**
- Variable/Dynamic Parallelism
  - no good solution for variable TLP

Coping with Diversity of Hardware

- There are dozens of processor and machine architectures in use today.
- The best implementation of an algorithm is dependent on:
  - machine
  - data set
  - concurrency
  - machine load
  - …
- Hand optimizing each architecture/dataset combination is not feasible

Auto-tuning

Auto-tuning

- Automatic Performance Tuning (Auto-tuning) is an empirical feedback driven technique designed to automate performance engineering.

- Our auto-tuning approach finds a good performance solution by a combination of heuristics and exhaustive search
  - Perl script generates many possible kernels
  - (Generate SIMD optimized kernels)
  - Auto-tuning benchmark examines kernels and reports back with the best one for the current architecture/dataset/compiler/…
  - Performance depends on the optimizations generated
  - Heuristics are often desirable when the search space isn't tractable

- Proven value in Dense Linear Algebra(ATLAS), Spectral (FFTW,SPIRAL), and Sparse Methods(OSKI)

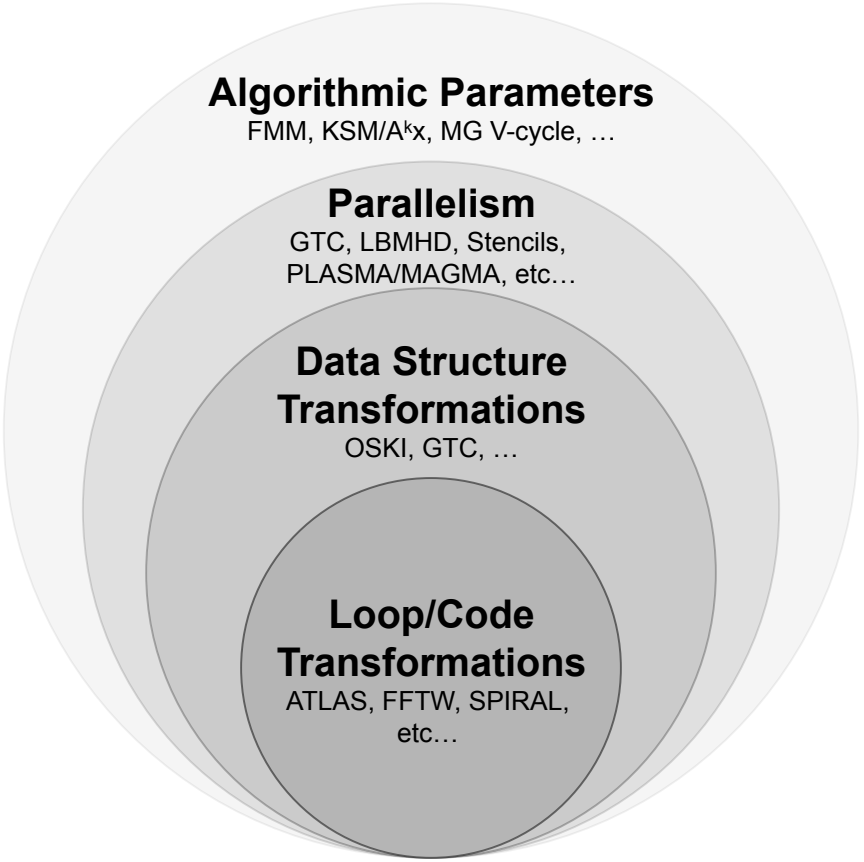
Auto-tuning

- Provides **performance portability** across the existing breadth and evolution of microprocessors
- One time up front productivity cost is amortized by the number of machines its used on

- Auto-tuning does not invent new optimizations
- **Auto-tuning automates the code generation and exploration of the optimization and parameter space**
- Two components:
  - parameterized code generator (we wrote ours in Perl)
  - Auto-tuning exploration benchmark
    (combination of heuristics and exhaustive search)
- Can be extended with ISA specific optimizations (e.g. DMA, SIMD)

Advancing the State of the Art in Optimizations

- Over the last 15 years, the set of optimizations available to auto-tuning has grown immensely.



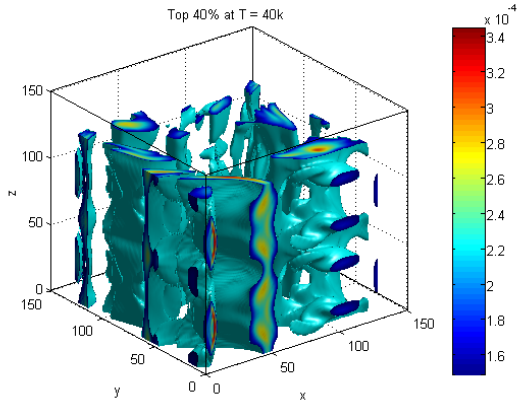
Lattice Boltzmann Magnetohydrodynamics (LBMHD)

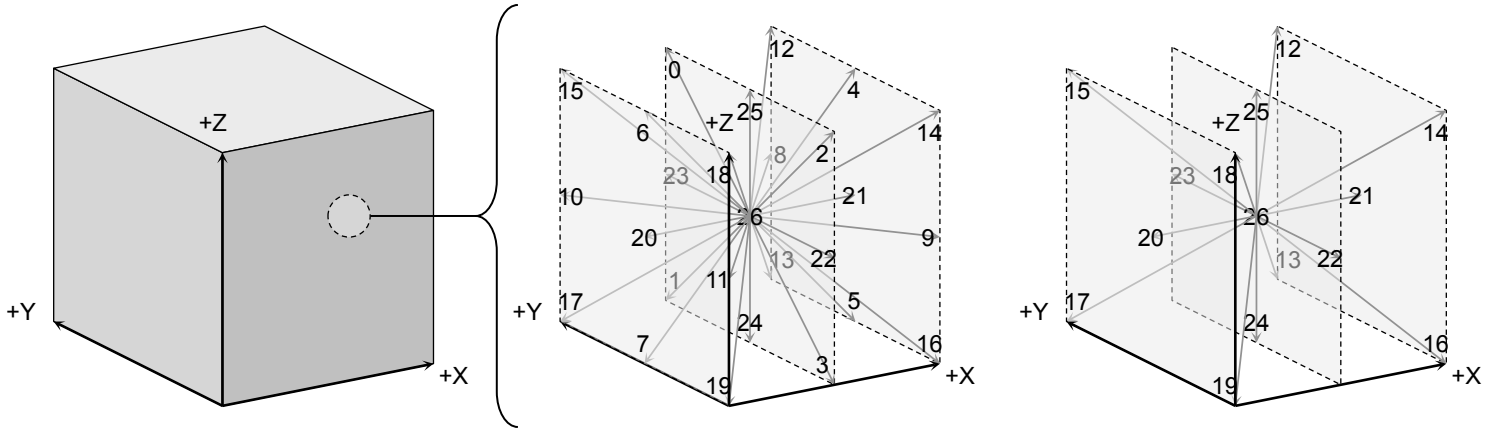
Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, Katherine Yelick, "Extracting Ultra-Scale Lattice Boltzmann Performance via Hierarchical and Distributed Auto-Tuning", Supercomputing (SC), 2011.

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, Applications Track***

LBMHD

- Lattice Boltzmann Magnetohydrodynamics (CFD+Maxwell's Equations)
- Plasma turbulence simulation via Lattice Boltzmann Method for simulating astrophysical phenomena and fusion devices
- Three macroscopic quantities:
  - Density
  - Momentum (vector)
  - Magnetic Field (vector)
- Two distributions:
  - momentum distribution (27 scalar components)
  - magnetic distribution (15 Cartesian vector components)





LBMHD

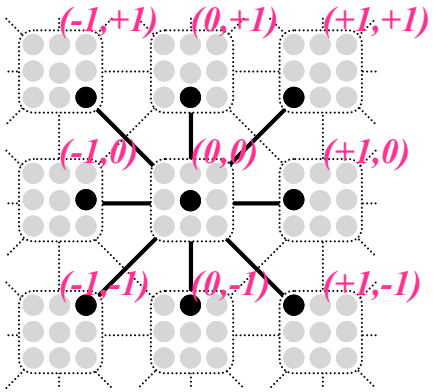
- Code Structure
  - time evolution through a series of *collision( )* and *stream( )* functions
- *stream( )*
  - performs a ghost zone exchange of data to facilitate distributed memory implementations as well as boundary conditions
  - should constitute 10% of the runtime
- *collision( )*'s Arithmetic 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 Cray XT4/XE6.

- 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

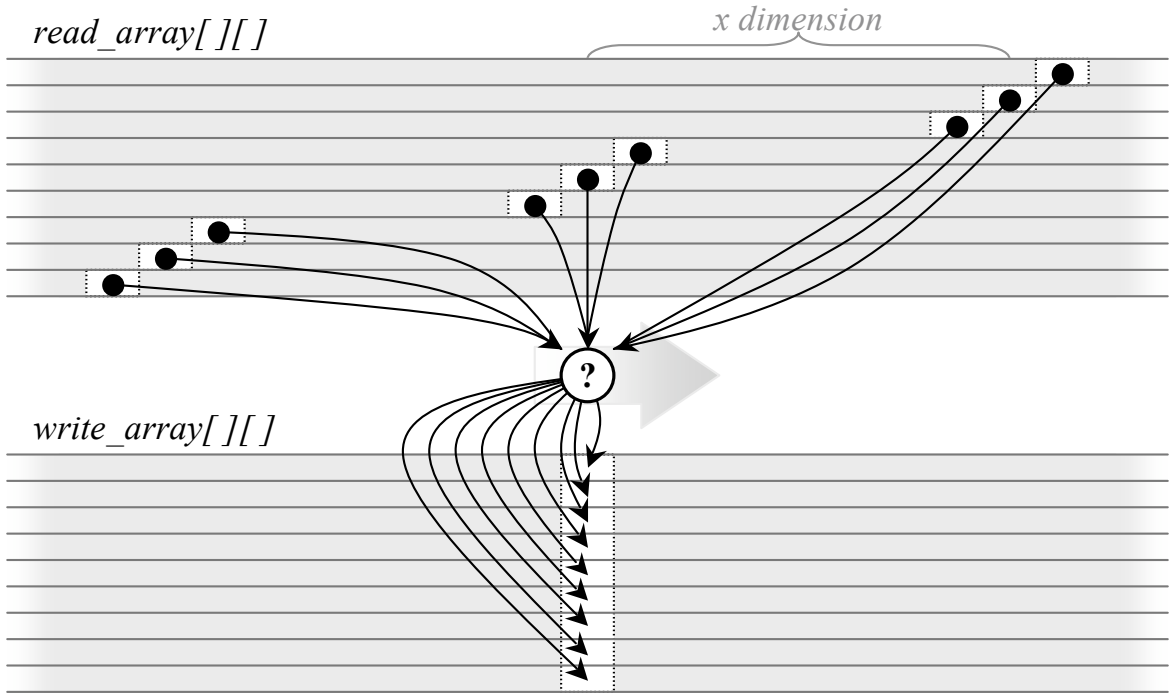
- periodic boundary conditions

LBMHD Stencil

- Simplified example reading from 9 arrays and writing to 9 arrays
- Actual LBMHD reads 73, writes 79 arrays



(a)



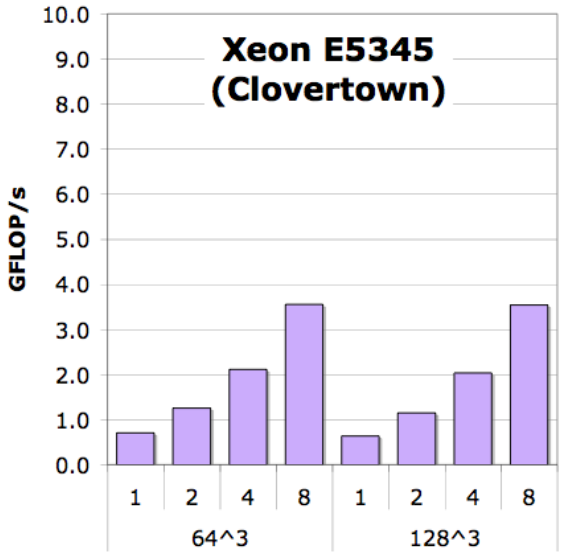
(b)

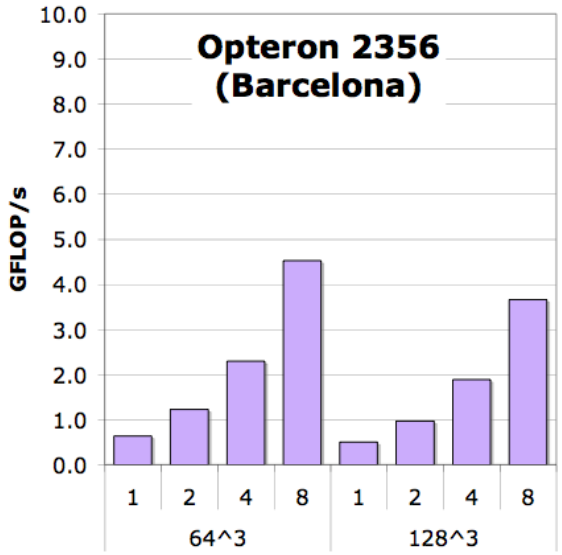
Auto-tuning LBMHD on Multicore SMPs

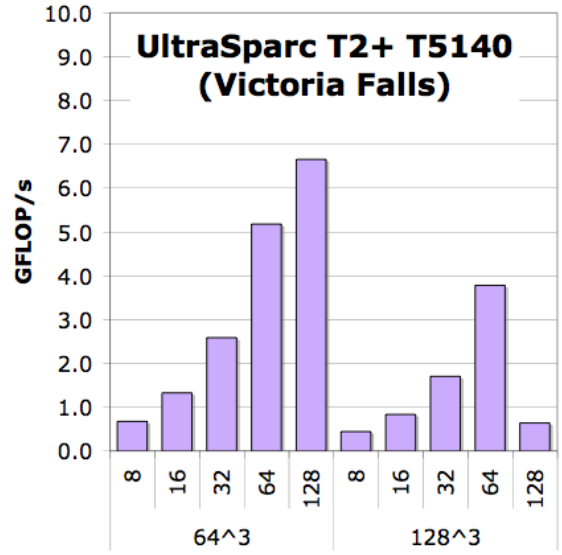
Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, Katherine Yelick, "Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms", International Parallel & Distributed Processing Symposium (IPDPS), 2008.

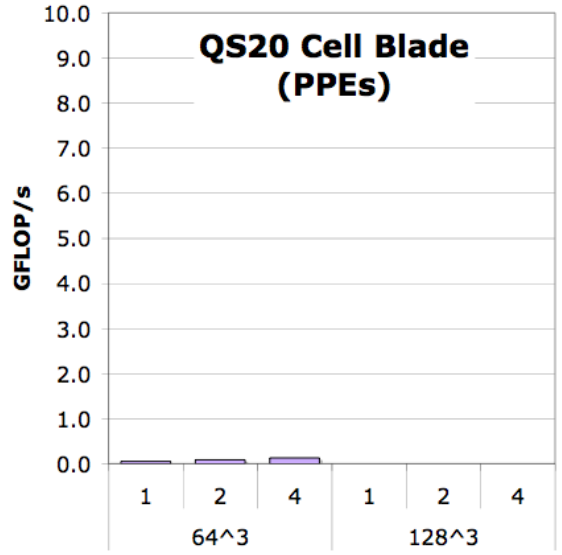
LBMHD Performance

(reference implementation)









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

Reference+NUMA

*collision() only

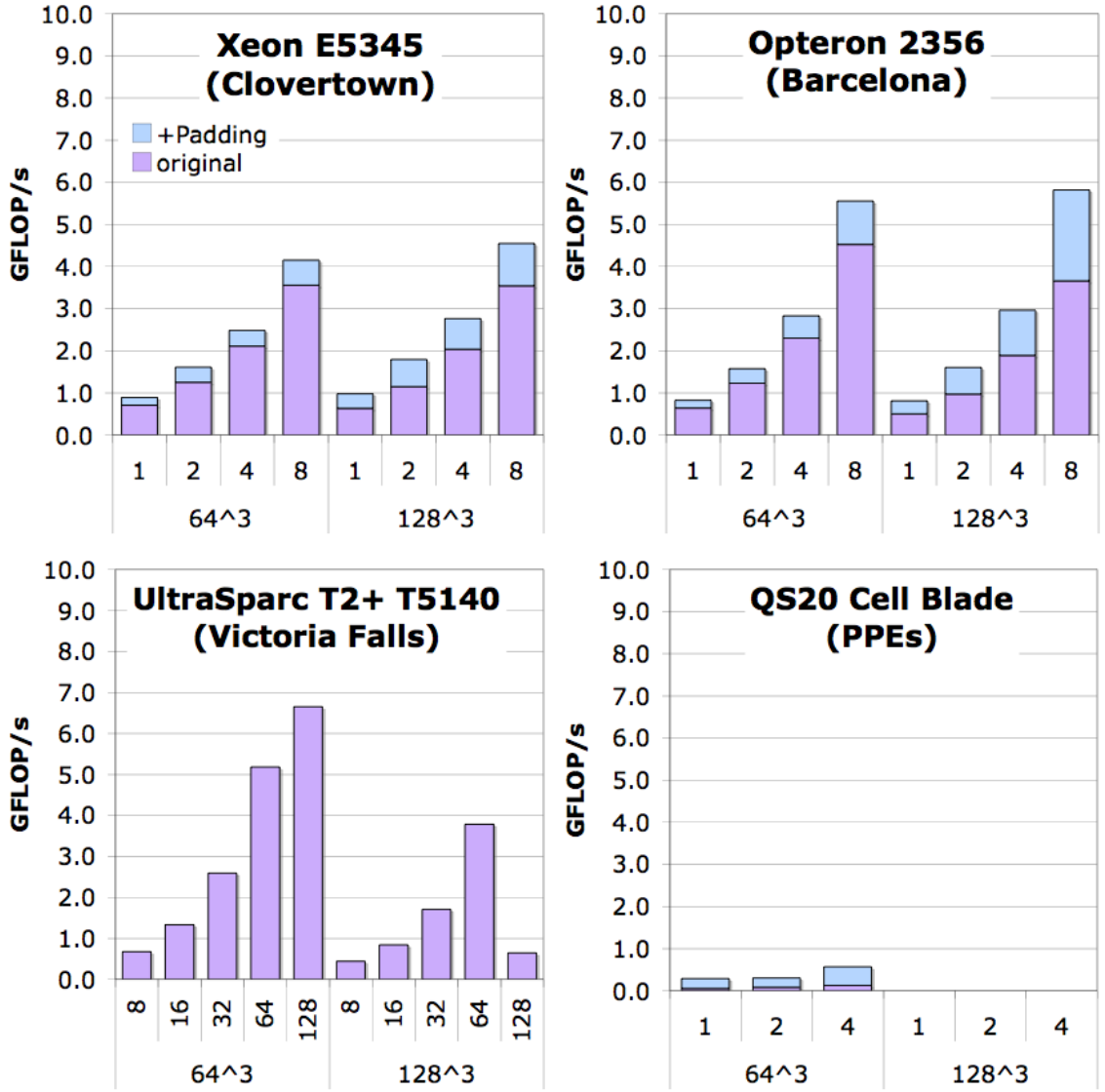
Lattice-Aware Padding

- 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 (±x,±y,±z) they are uniformly distributed throughout the sets of the cache
- Maximizes cache utilization and minimizes conflict misses.

LBMHD Performance

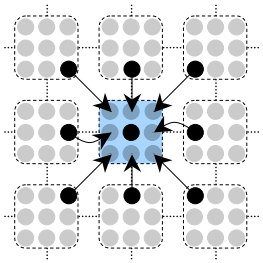
(lattice-aware array padding)

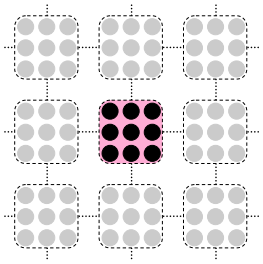


- LBMHD touches >150 arrays.
- Most caches have limited associativity
- Conflict misses are likely
- Apply **heuristic** to pad arrays

Vectorization

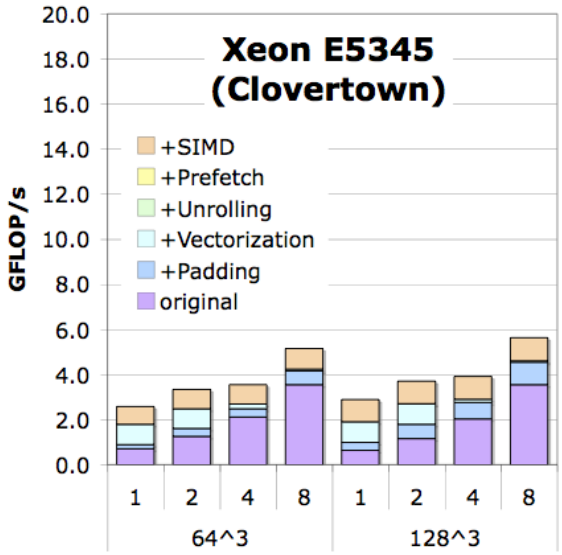
- Two phases with a lattice method's collision() operator:
  - reconstruction of macroscopic variables
  - updating discretized velocities
- Normally this is done one point at a time.
- Change to do a vector's worth at a time (loop interchange + tuning)

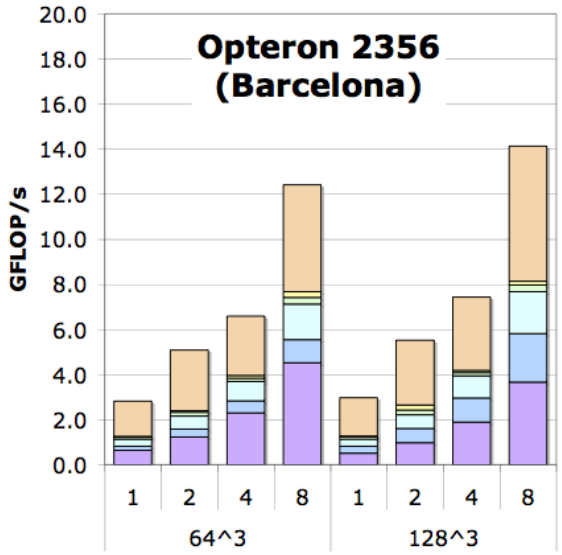


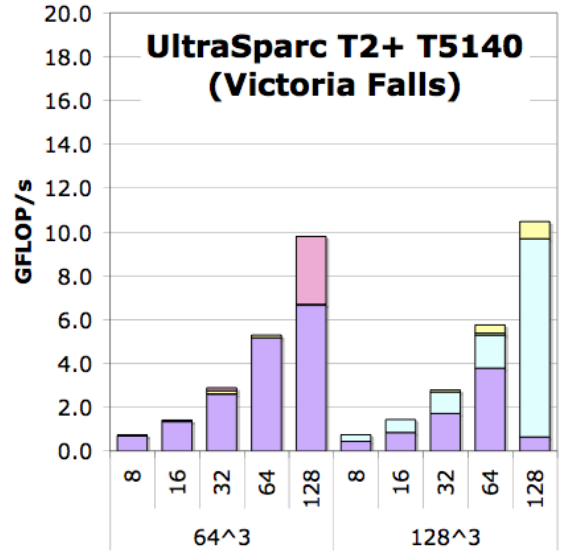


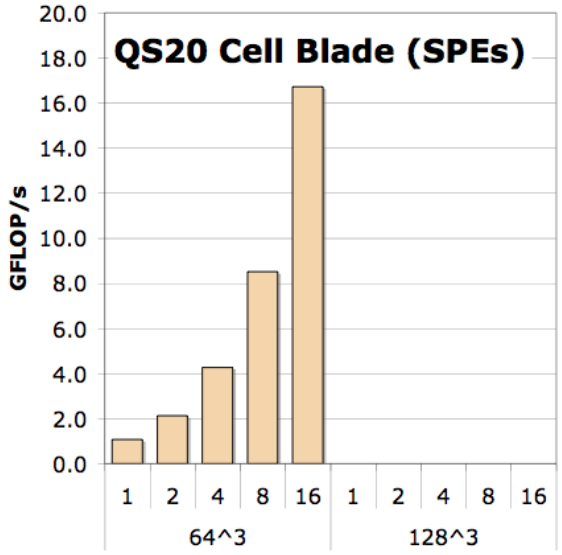
LBMHD Performance

(architecture specific optimizations)

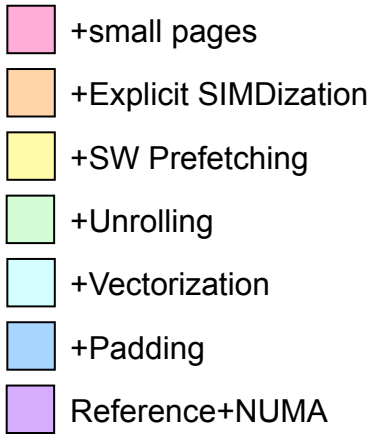








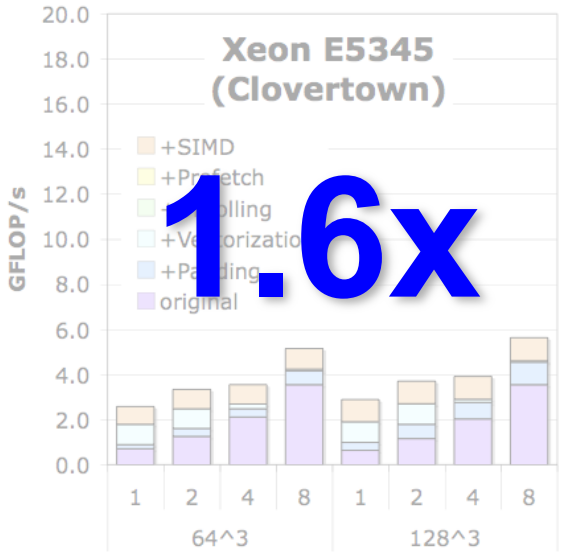
- Add unrolling and reordering of inner loop
- Additionally, it exploits SIMD where the compiler doesn't
- Include a SPE/Local Store optimized version

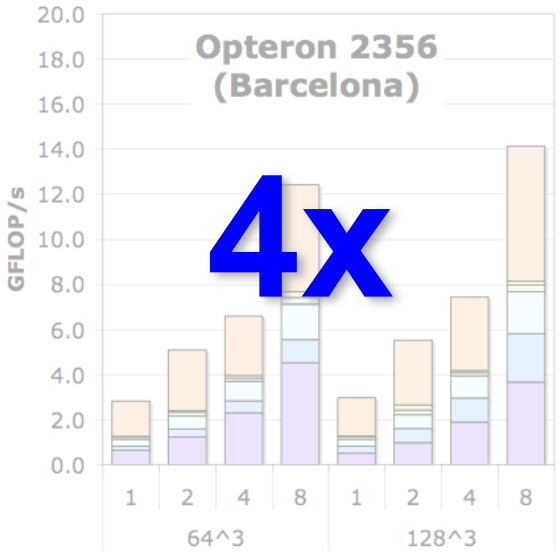


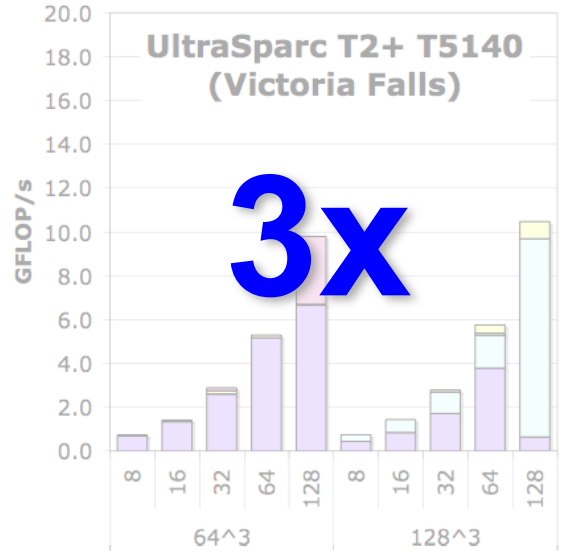
*collision() only

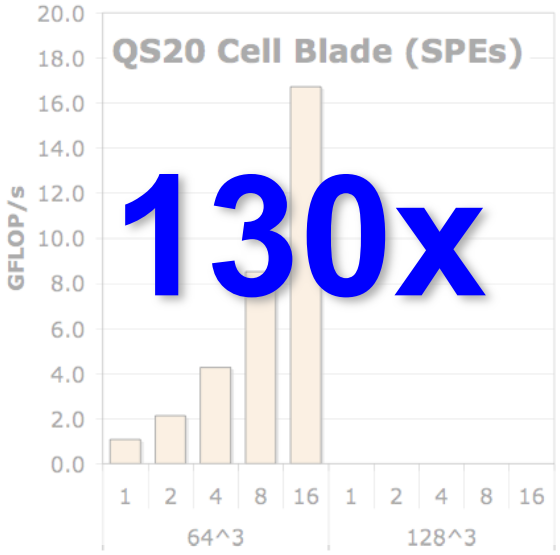
LBMHD Performance

(architecture specific optimizations)









- 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
Reference+NUMA

*collision() only

Limitations

- 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

Auto-tuning LBMHD on Multicore MPPs

Samuel Williams, Jonathan Carter, Leonid Oliker, John Shalf, Katherine Yelick, "Extracting Ultra-Scale Lattice Boltzmann Performance via Hierarchical and Distributed Auto-Tuning", Supercomputing (SC), 2011.

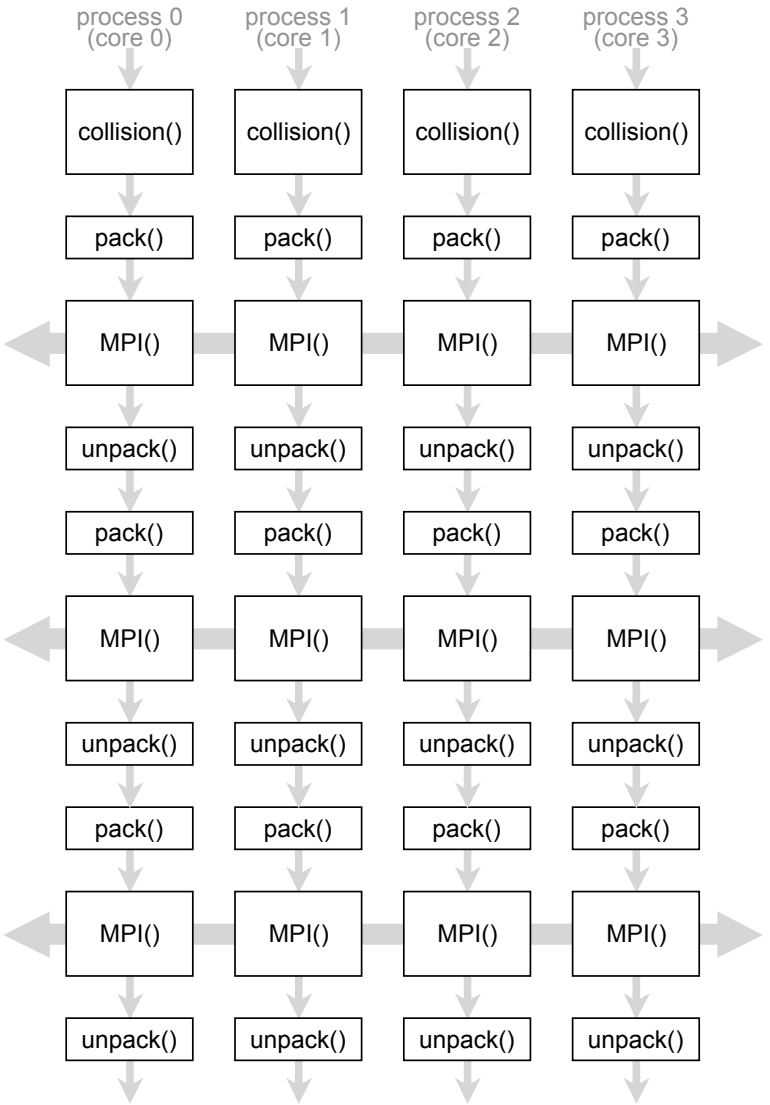
MPI+Pthreads and MPI+OpenMP Implementations

Explored performance on 3 ultrascale machines using 2048 nodes on each and running a 1GB, 4GB, and if possible 16GB(per node) problem size.

•IBM Blue Gene/P at Argonne (Intrepid)	8,192 cores
•Cray XT4 at NERSC (Franklin)	8,192 cores
•Cray XE6 at NERSC (Hopper)	49,152 cores

Flat MPI

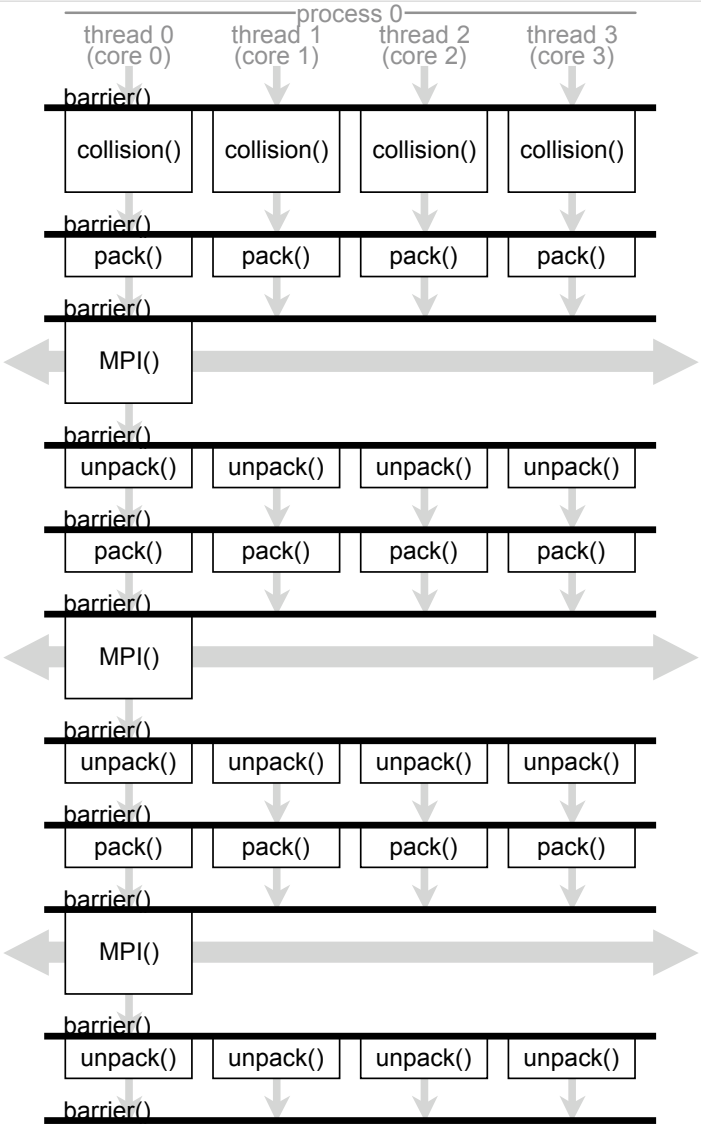
- In the flat MPI world, there is one process per core, and only one thread per process
- All communication is through MPI



Hybrid MPI + Pthreads/OpenMP

- As multicore processors already provide cache coherency for free, we can exploit it to **reduce MPI overhead and traffic**.
- We examine using pthreads and OpenMP for threading (other possibilities exist)
- For correctness in pthreads, we are required to include a intra-process (thread) barrier between function calls for correctness.

  (we wrote our own)
- Implicitly, OpenMP will barrier via the #pragma
- We can choose any balance between processes/ node and threads/process
- In both Pthreads and OpenMP, only thread 0 performs MPI calls



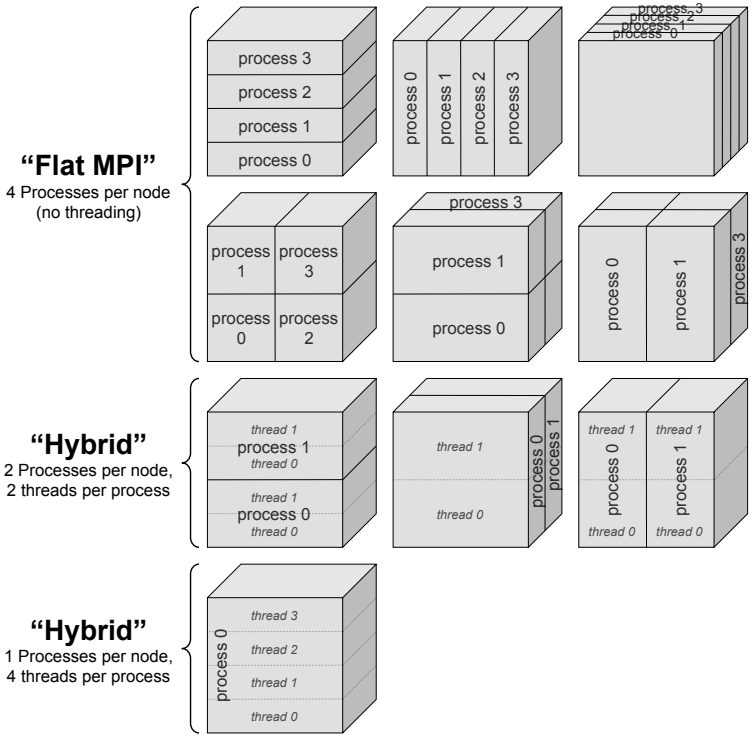
The Distributed Auto-tuning Problem

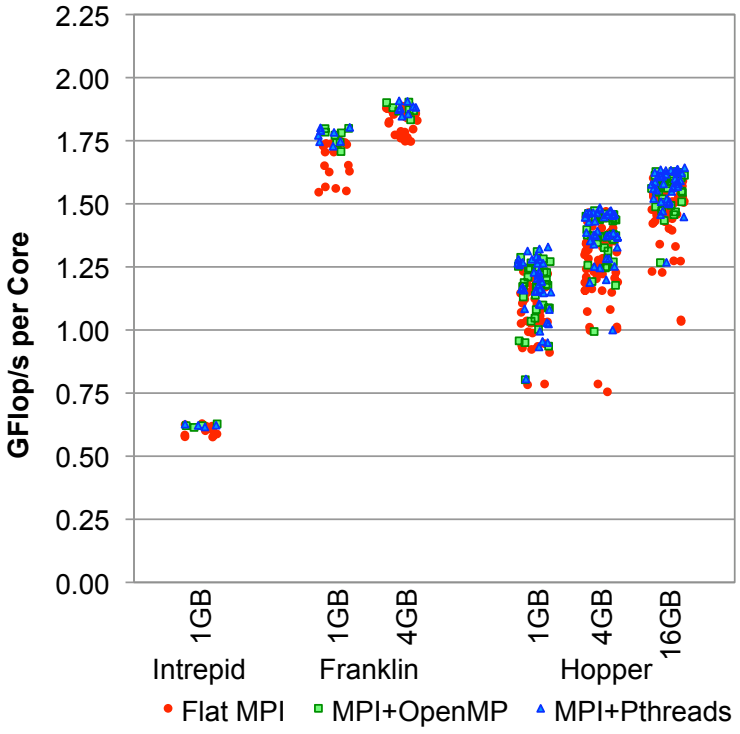
- We believe that even for relatively large problems, auto-tuning only the local computation (e.g. IPDPS'08) will deliver sub-optimal MPI performance.
- Want to explore MPI/Hybrid decomposition as well
- We have a combinatoric explosion in the search space coupled with a large problem size (number of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

- To remedy this, we employ a greedy search approach that
  - determines the best single core implementation (on a single node) ~ IPDPS work
  - explores the best parallel MPI decomposition among nodes and the best on-node programming model (8-64 nodes)
  - Evaluates performance at scale (2048 nodes = 49,152 cores)

Stage 2

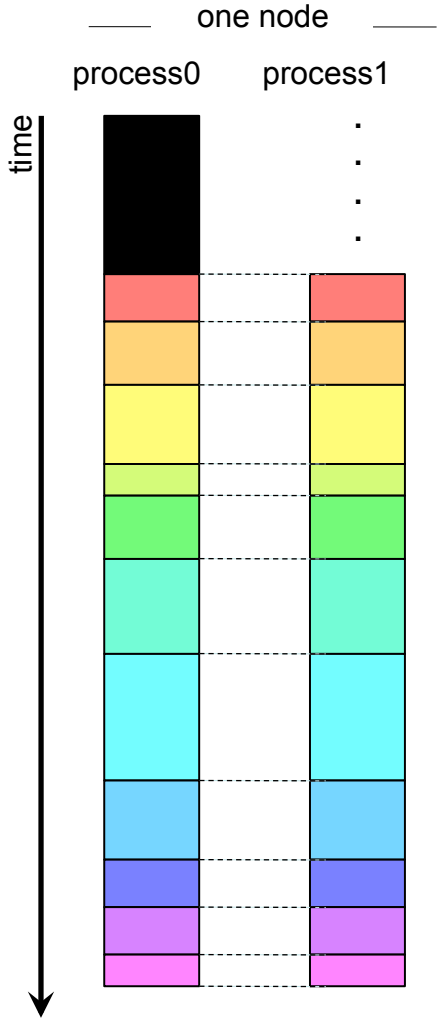
- In stage 2, we prune the MPI space.
- Given a fixed memory footprint per node, explore the different ways of partitioning it among processes and threads.





Stage 2

- Hybrid Auto-tuning requires we mimic the SPMD environment
- 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 (a configuration with 100's of iterations)



Results

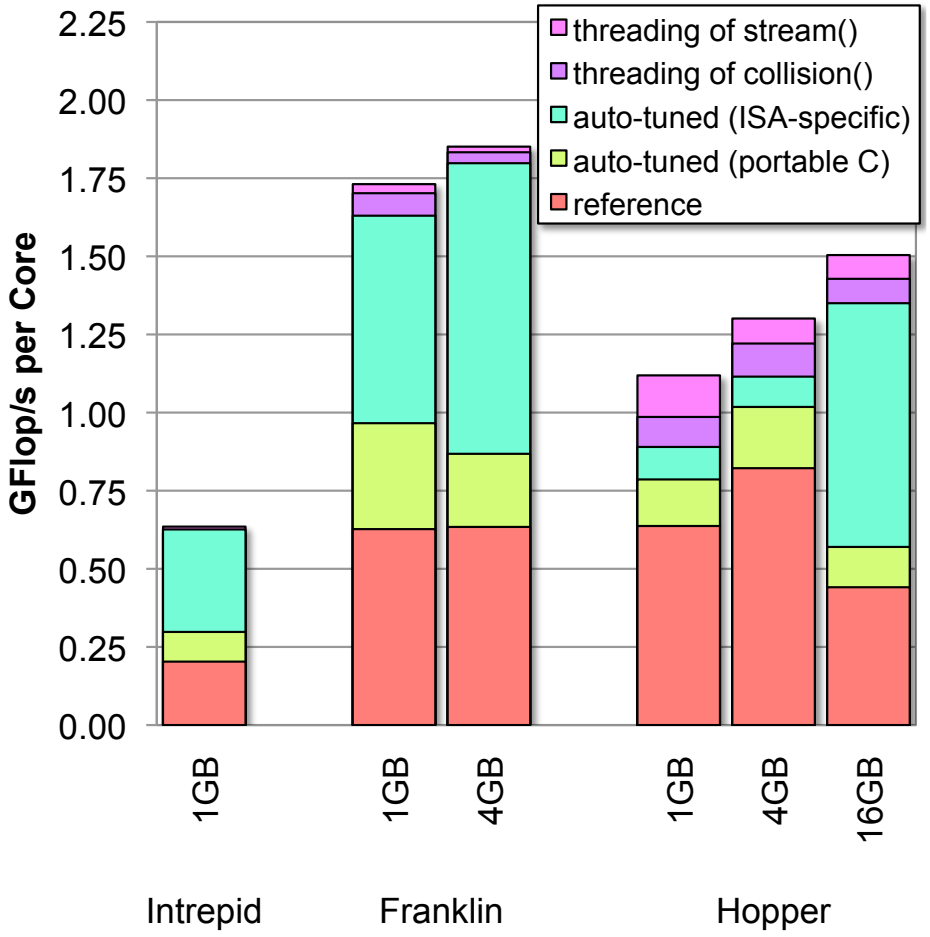
Performance Results

(using 2048 nodes on each machine)

- We present the best data for progressively more aggressive auto-tuning efforts
- Remember, Hopper has 6x as many cores per node as Intrepid or Franklin. So performance per node is far greater.

- auto-tuning can improve performance
- ISA-specific optimizations (e.g. SIMD intrinsics) help more
- Overall, we see speedups of up to 3.4x

- As problem size increased, so to does performance. However, the value of threading is diminished.



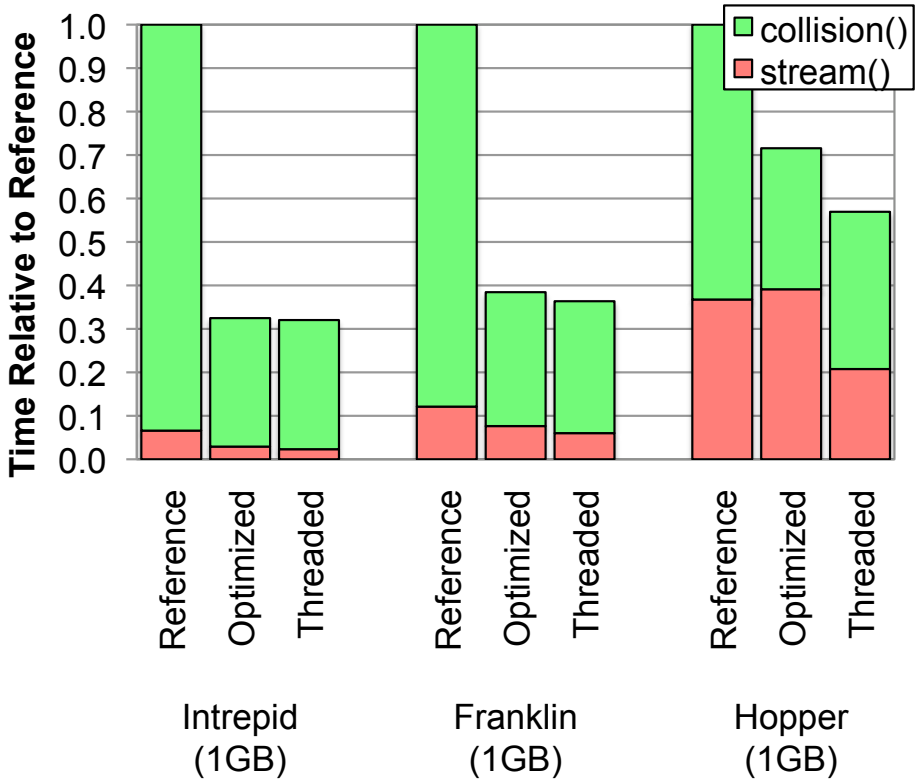
Performance Results

(using 2048 nodes on each machine)

- We present the best data for progressively more aggressive auto-tuning efforts
- Remember, Hopper has 6x as many cores per node as Intrepid or Franklin. So performance per node is far greater.

- auto-tuning can improve performance
- ISA-specific optimizations (e.g. SIMD intrinsics) help more
- As problem size increased, so to does performance. However, the value of threading is diminished.

- ***For small problems, MPI time can dominate runtime on Hopper***
- ***Threading mitigates this***



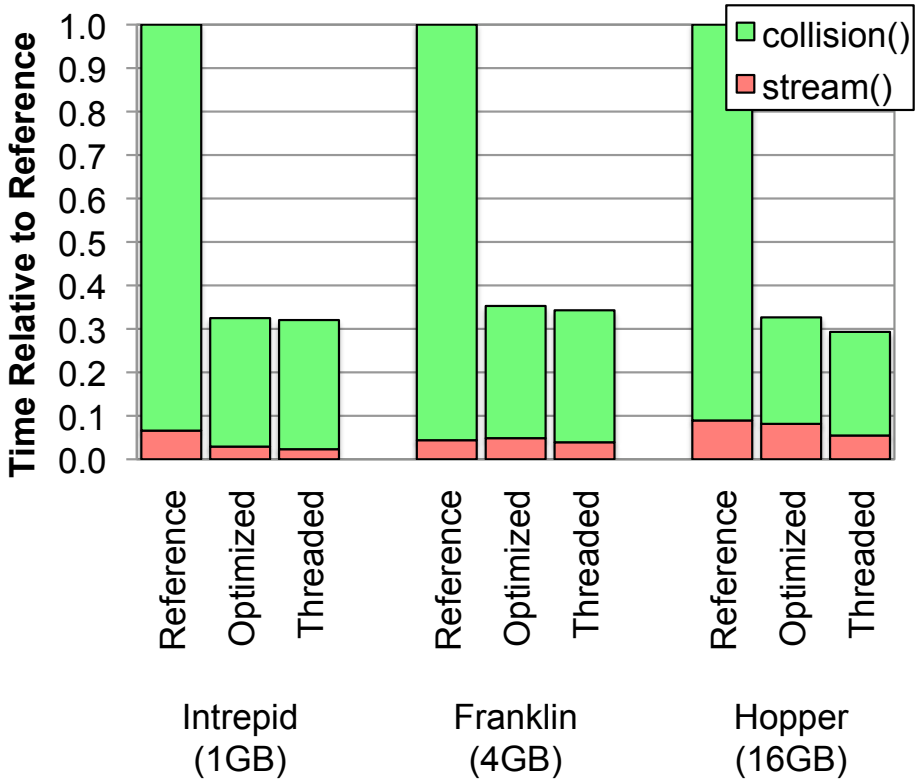
Performance Results

(using 2048 nodes on each machine)

- We present the best data for progressively more aggressive auto-tuning efforts
- Remember, Hopper has 6x as many cores per node as Intrepid or Franklin. So performance per node is far greater.

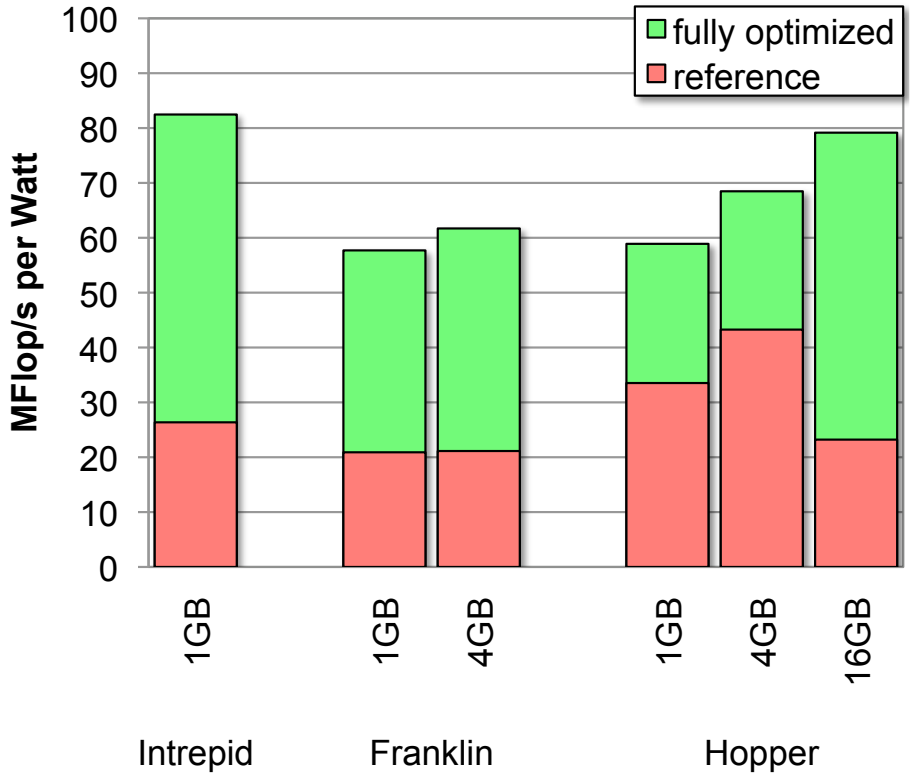
- auto-tuning can improve performance
- ISA-specific optimizations (e.g. SIMD intrinsics) help more
- As problem size increased, so to does performance. However, the value of threading is diminished.

- **For large problems, MPI time remains a small fraction of overall time**



Energy Results
(using 2048 nodes on each machine)

- Ultimately, energy is becoming the great equalizer among machines.
- Hoper has 6x the cores, but burns 15x the power of Intrepid.
- To visualize this, we explore energy efficiency (Mflop/s per Watt)
- Clearly, despite the performance differences, energy efficiency is remarkably similar.



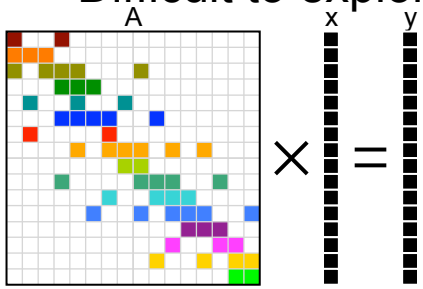
Sparse Matrix Vector Multiplication (SpMV)

Auto-tuning Sparse Matrix-Vector Multiplication (SpMV)

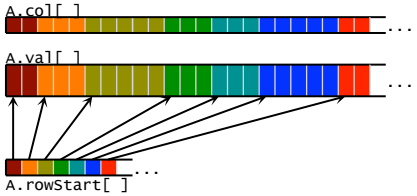
Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, James Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms", Supercomputing (SC), 2007.

Sparse Matrix Vector Multiplication

- What's a Sparse Matrix ?
  - Most entries are 0.0
  - Performance advantage in only storing/operating on the nonzeros
  - Requires significant meta data to record the matrix structure
- What's SpMV ?
  - Evaluate y=Ax
  - A is a sparse matrix, x & y are dense vectors
- Challenges
  - **Very memory-intensive (often <0.166 flops/byte)**
  - Difficult to exploit ILP (bad for pipelined or superscalar),
  - Difficult to exploit DLP (bad for SIMD)



(a)
algebra conceptualization



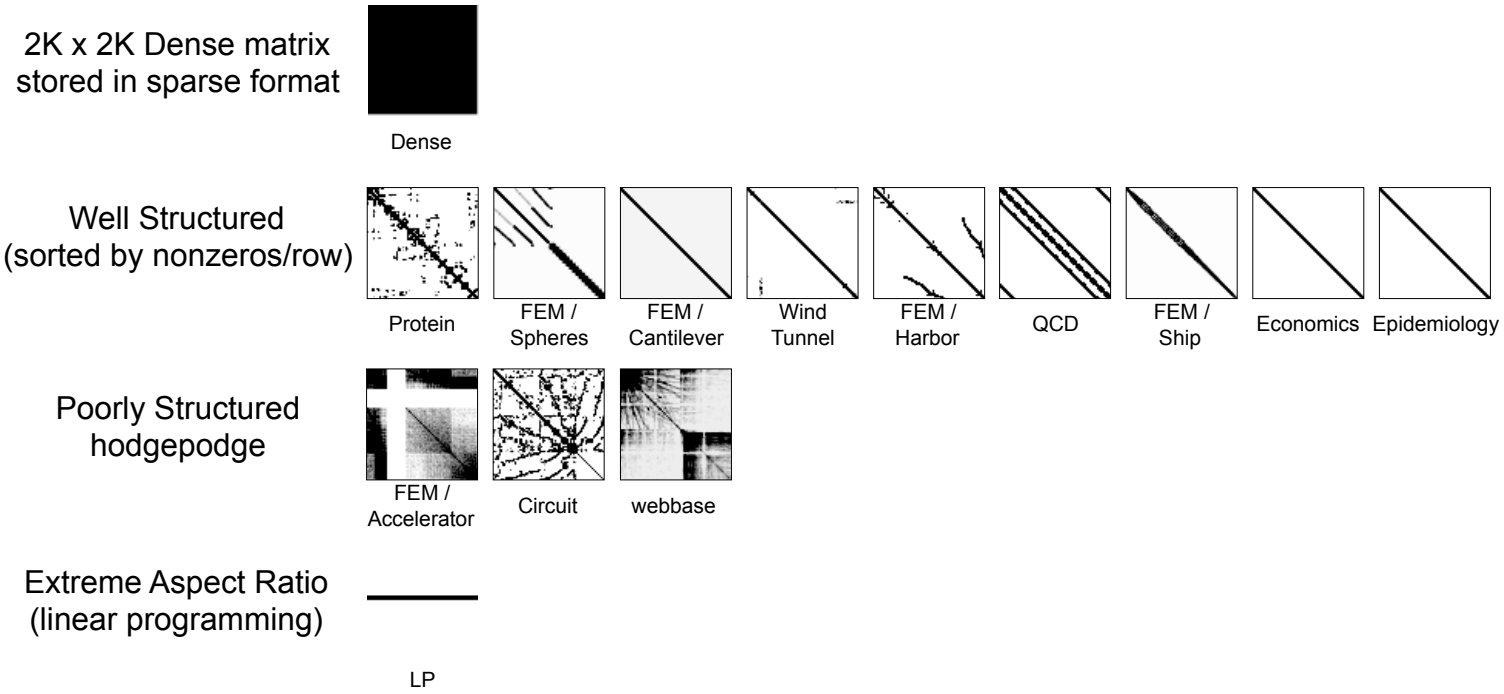
(b)
CSR data structure

```
for (r=0; r<A.rows; r++) {
  double y0 = 0.0;
  for (i=A.rowStart[r]; i<A.rowStart[r+1]; i++){
    y0 += A.val[i] * x[A.col[i]];
  }
  y[r] = y0;
}
```

(c)
CSR reference code

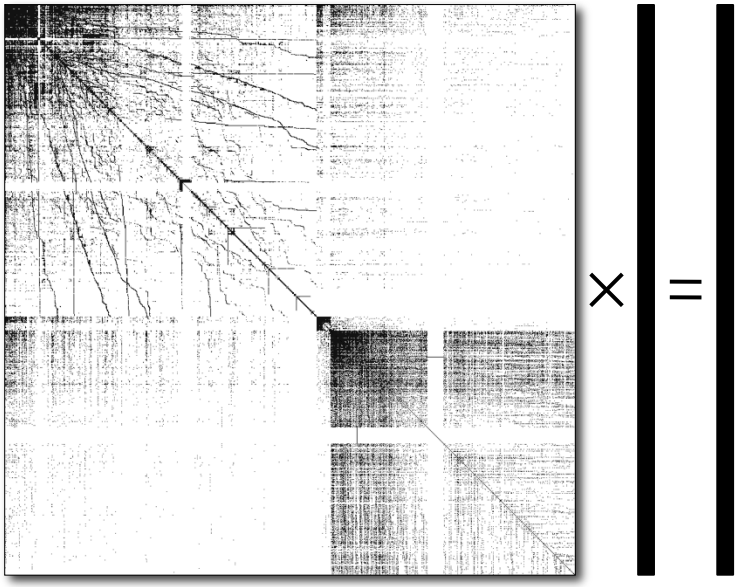
The Dataset (matrices)

- Unlike DGEMV, performance is dictated by sparsity
- Suite of 14 matrices
- All bigger than the caches of our SMPs
- We'll also include a median performance number



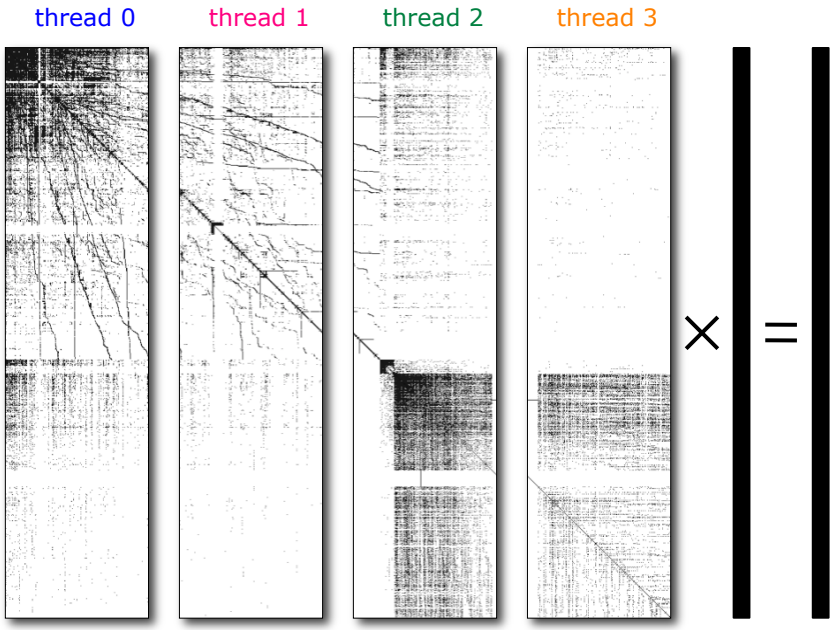
SpMV Parallelization

- How do we parallelize a matrix-vector multiplication ?



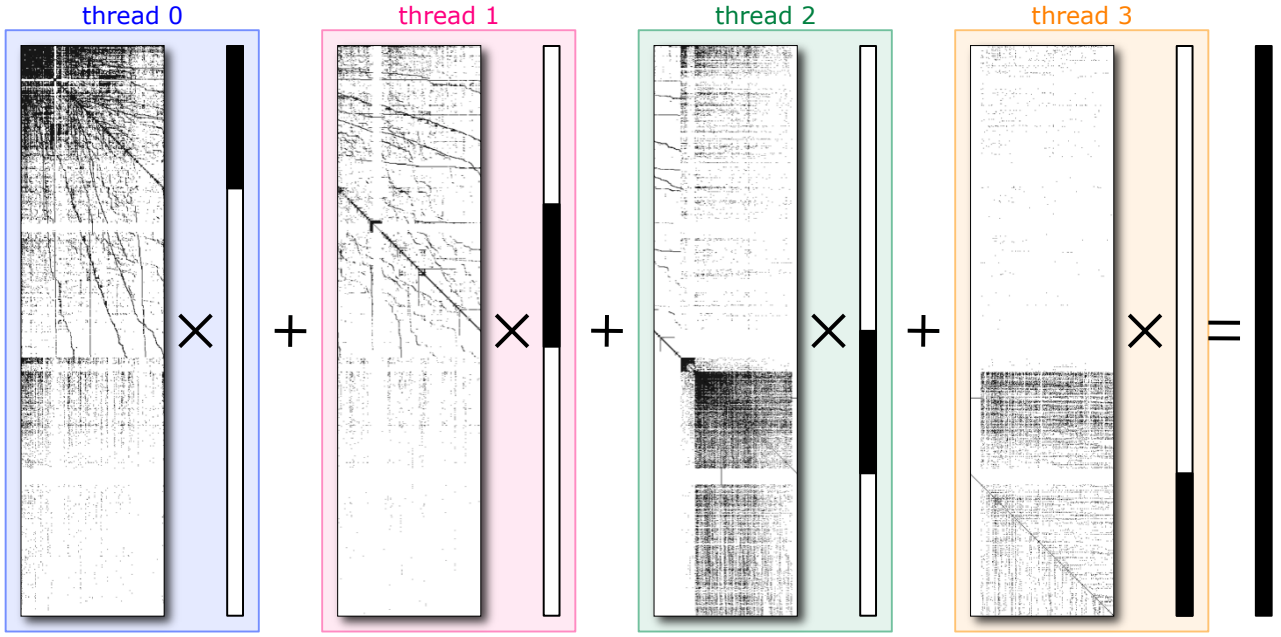
SpMV Parallelization

- How do we parallelize a matrix-vector multiplication ?
- We could parallelize by columns (sparse matrix time dense sub vector) and in the worst case simplify the random access challenge but:
  - each thread would need to store a temporary partial sum
  - and we would need to perform a reduction (inter-thread data dependency)



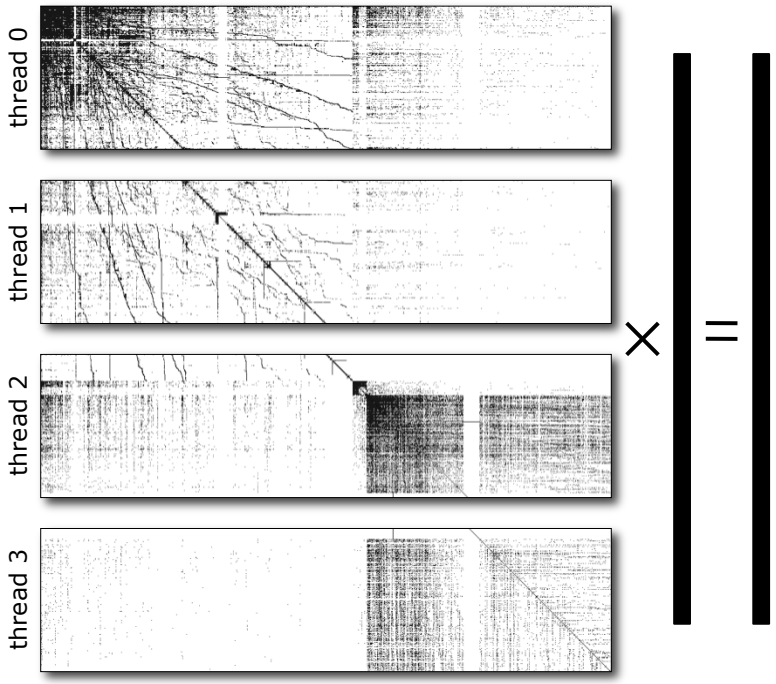
SpMV Parallelization

- How do we parallelize a matrix-vector multiplication ?
- We could parallelize by columns (sparse matrix time dense sub vector) and in the worst case simplify the random access challenge but:
  - each thread would need to store a temporary partial sum
  - and we would need to perform a reduction (inter-thread data dependency)

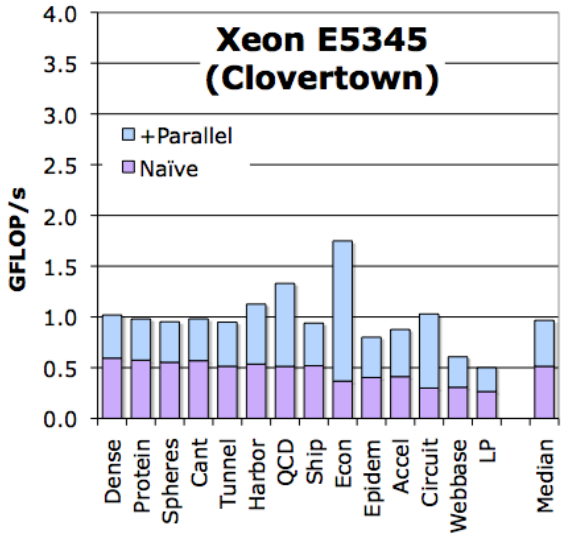


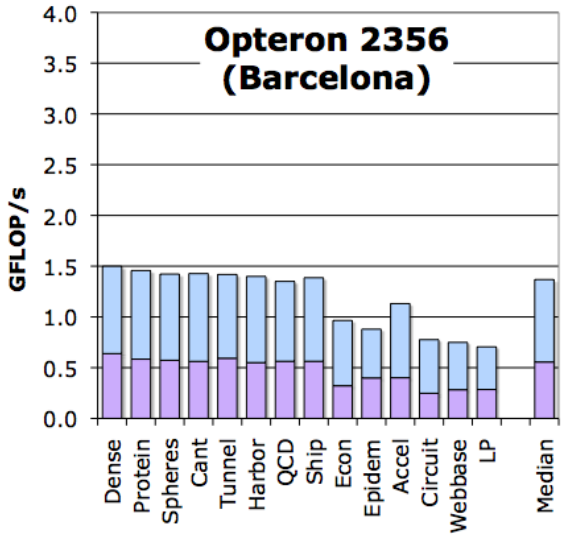
SpMV Parallelization

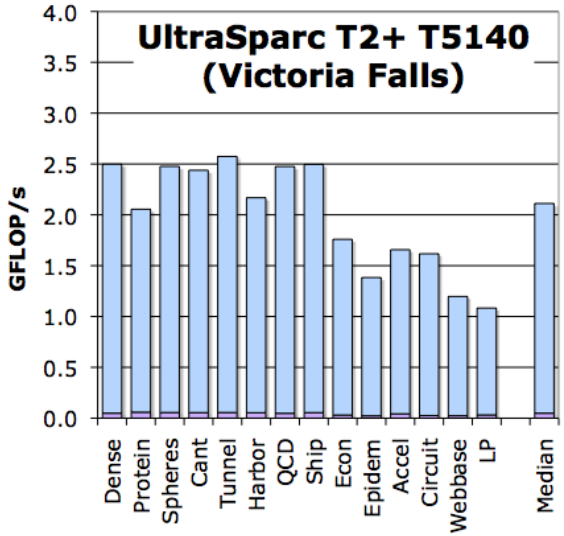
- How do we parallelize a matrix-vector multiplication ?
- Alternately, we could parallelize by rows.
- Clearly, there are now no inter-thread data dependencies, but, in the worst case, one must still deal with challenging random access

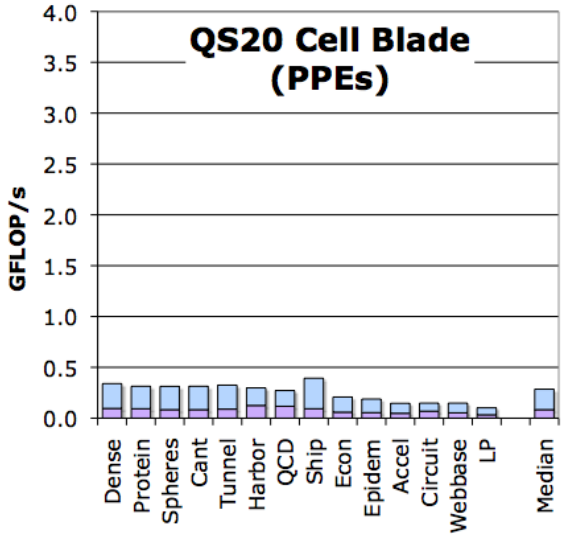


SpMV Performance
(simple parallelization)









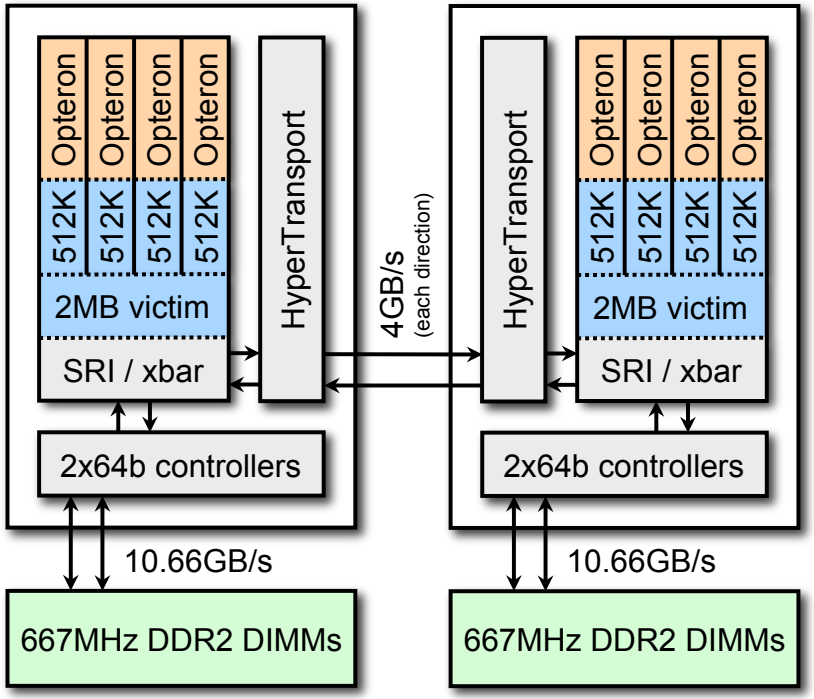
- Out-of-the box SpMV performance on a suite of 14 matrices
- Simplest solution = parallelization by rows (solves data dependency challenge)
- **Scalability isn't great**
- **Is this performance good?**

Naïve Pthreads

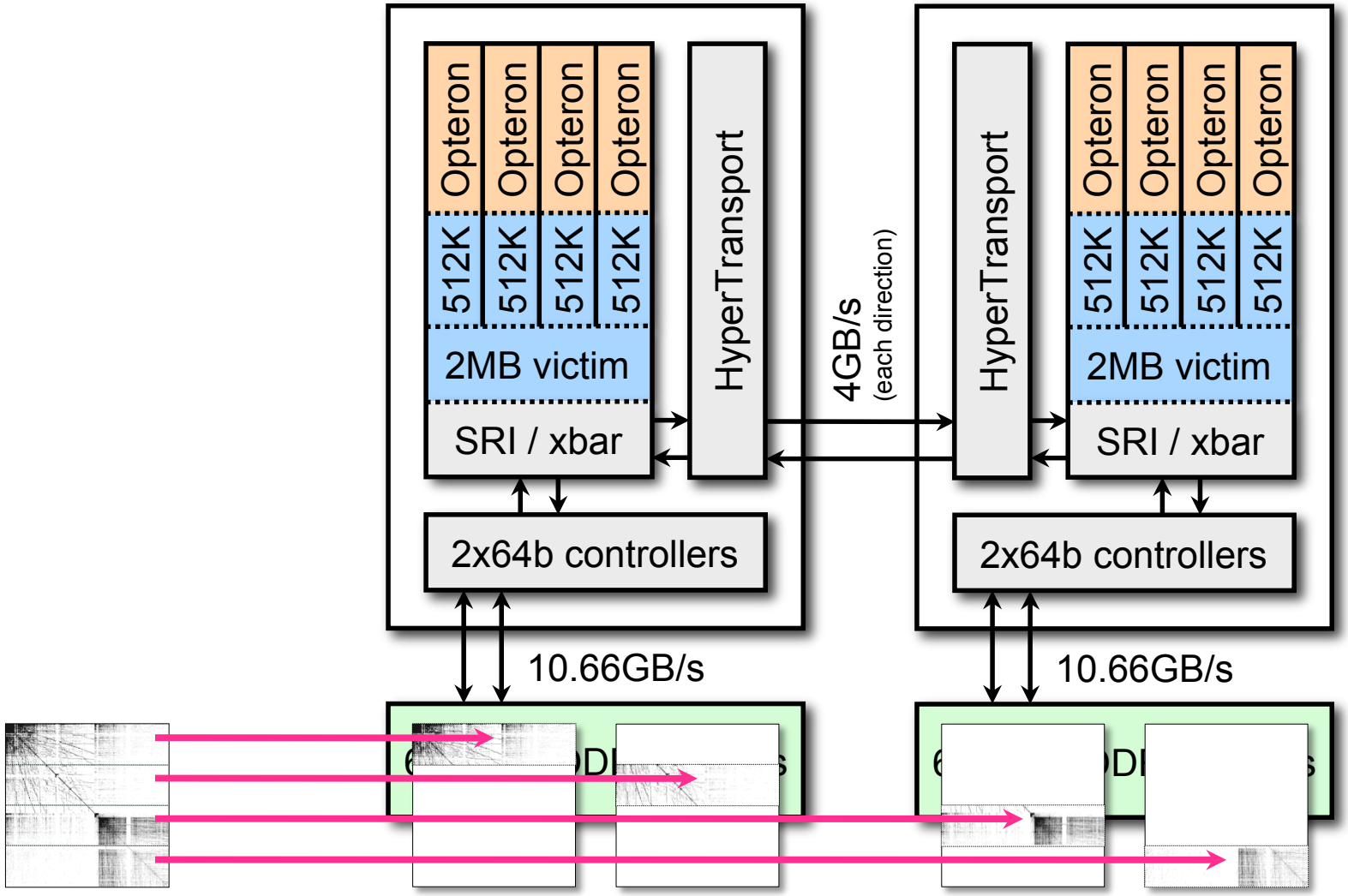
Naïve

NUMA
(Data Locality for Matrices)

- On NUMA architectures, all large arrays should be partitioned either
  - explicitly (multiple malloc()'s + affinity)
  - implicitly (parallelize initialization and rely on first touch)
- You cannot partition on granularities less than the page size
  - 512 elements on x86
  - 2M elements on Niagara
- For SpMV, partition the matrix and perform multiple malloc()'s
- Pin submatrices so they are co-located with the cores tasked to process them



NUMA
(Data Locality for Matrices)

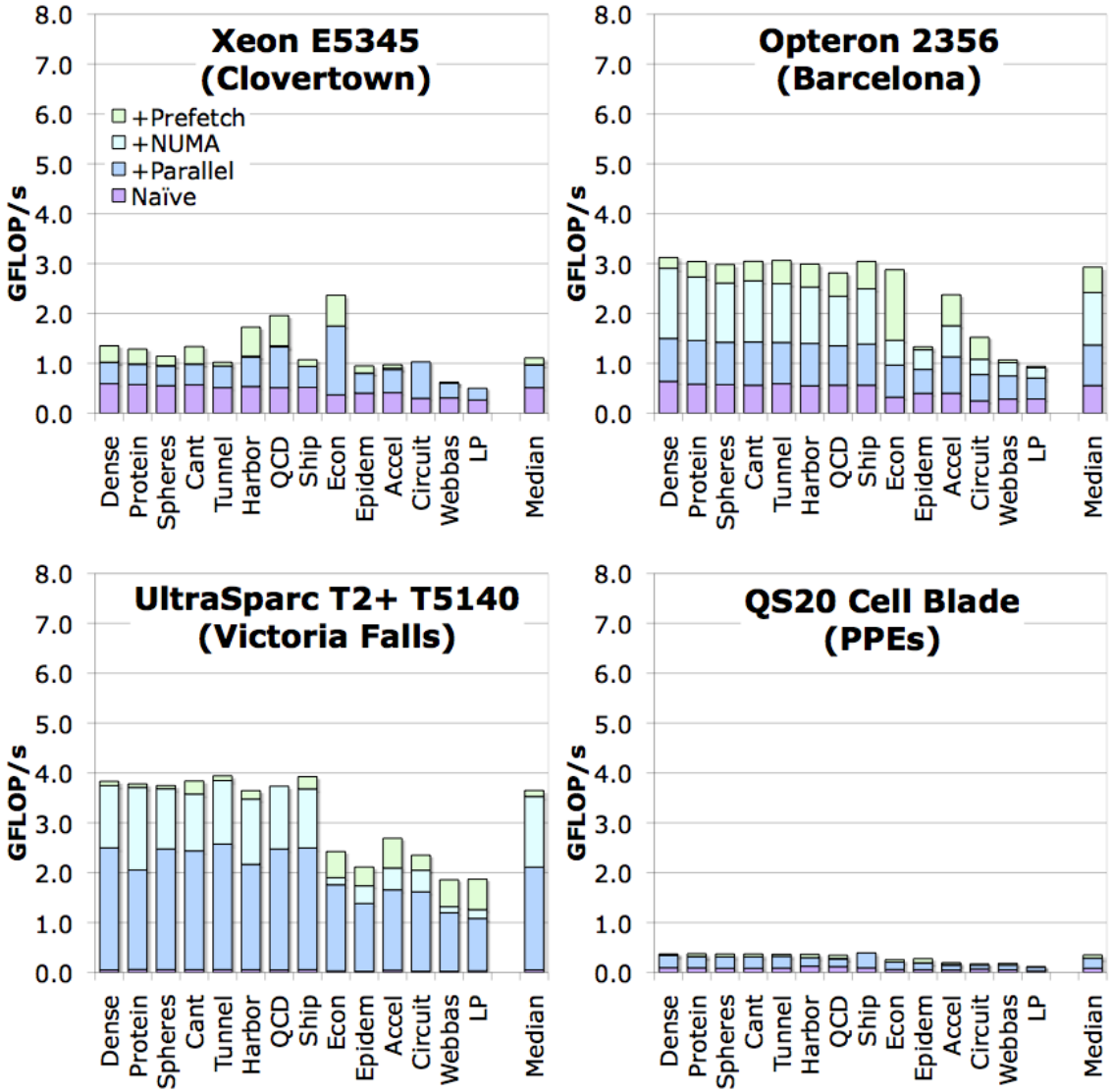


Prefetch for SpMV

- SW prefetch injects more MLP into the memory subsystem.
- (attempts to supplement HW prefetchers in their attempt to satisfy **Little's Law**)
- Can try to prefetch the
  - values
  - indices
  - source vector
  - *or any combination thereof*
- In general, should only insert one prefetch per cache line (works best on unrolled code)

```
for(all rows){
  y0 = 0.0;
  y1 = 0.0;
  y2 = 0.0;
  y3 = 0.0;
  for(all tiles in this row){
    PREFETCH(V+i+PFDistance);
    y0+=V[i  ]*X[C[i]]
    y1+=V[i+1]*X[C[i]]
    y2+=V[i+2]*X[C[i]]
    y3+=V[i+3]*X[C[i]]
  }
  y[r+0] = y0;
  y[r+1] = y1;
  y[r+2] = y2;
  y[r+3] = y3;
}
```

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

ILP/DLP vs Bandwidth

- In the multicore era, which is the bigger issue?
  - a lack of ILP/DLP (a major advantage of BCSR)
  - insufficient memory bandwidth per core

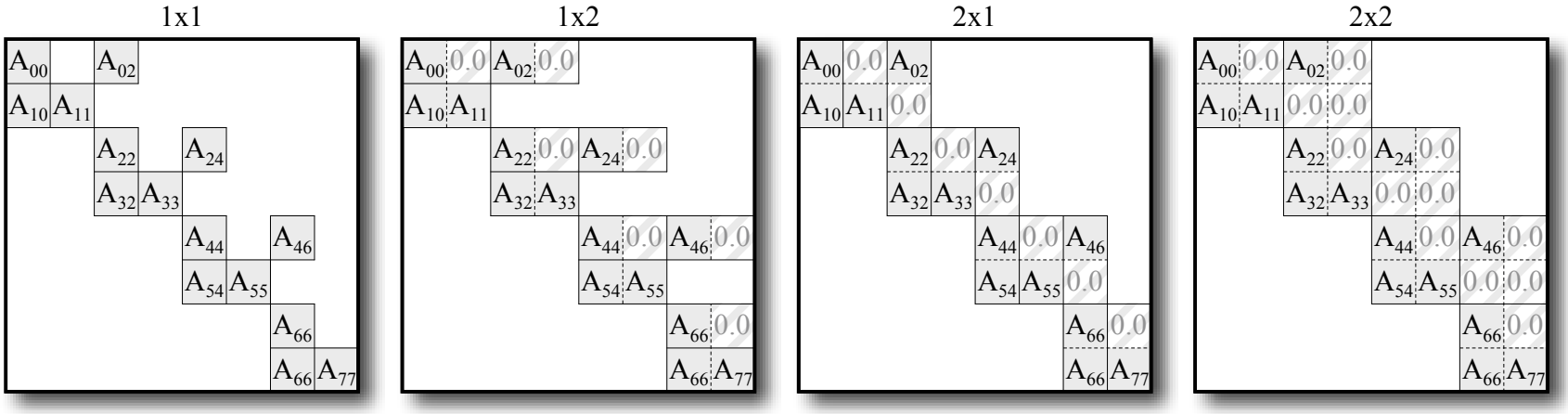
- There are many architectures that when running low arithmetic intensity kernels, there is so little available memory bandwidth per core that you won't notice a complete lack of ILP

- Perhaps we should concentrate on **minimizing memory traffic** rather than maximizing ILP/DLP

- Rather than benchmarking every combination, just
  Select the register blocking that minimizes the matrix foot print.

Matrix Compression Strategies

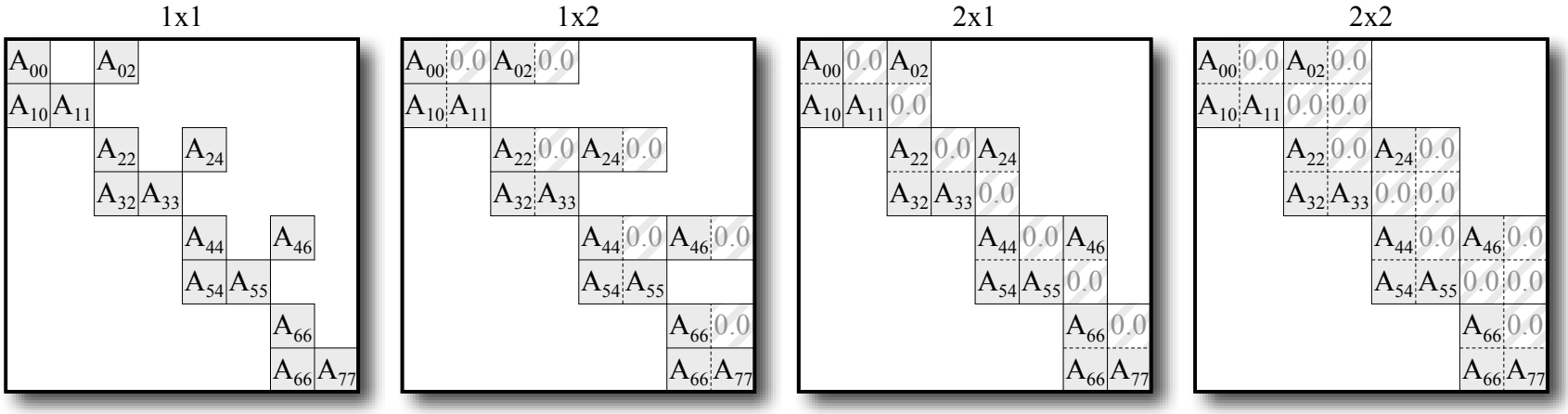
- Register blocking creates small dense tiles
  - better ILP/DLP
  - reduced overhead per nonzero



- Let each thread select a unique register blocking
- In this work,
  - we only considered power-of-two register blocks
  - select the register blocking that minimizes memory traffic

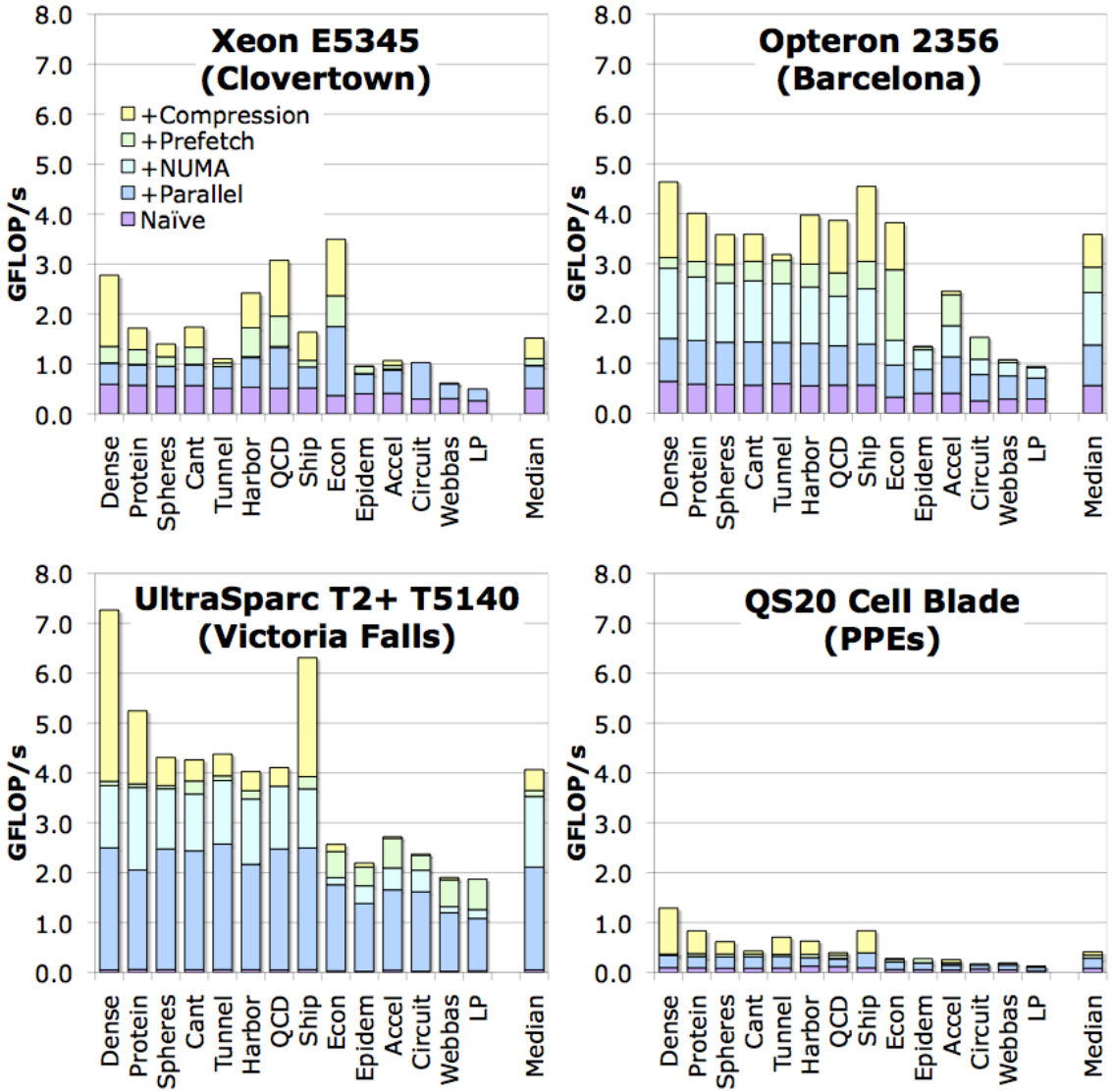
Matrix Compression Strategies

- Where possible we may encode indices with less than 32 bits
- We may also select different matrix formats



- In this work,
  - we considered 16-bit and 32-bit indices (relative to thread's start)
  - we explored BCSR/BCOO (GCSR in book chapter)

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)

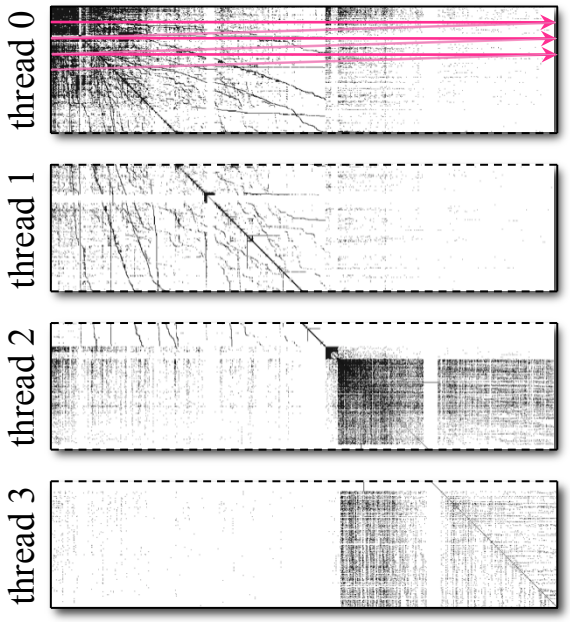
Cache blocking for SpMV

(Data Locality for Vectors)

- Cache-blocking sparse matrices is very different than cache-blocking dense matrices.
- Rather than changing loop bounds, store entire submatrices contiguously.
- The columns spanned by each cache block are selected so that all submatrices place the same pressure on the cache

  i.e. touch the same number of unique source vector cache lines

- TLB blocking is a similar concept but instead of on 8 byte granularities, it uses 4KB granularities

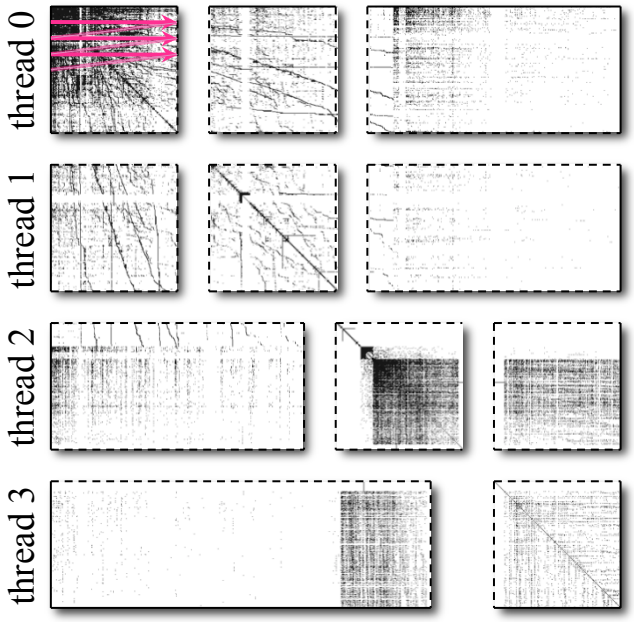


Cache blocking for SpMV
(Data Locality for Vectors)

- Cache-blocking sparse matrices is very different than cache-blocking dense matrices.
- Rather than changing loop bounds, store entire submatrices contiguously.
- The columns spanned by each cache block are selected so that all submatrices place the same pressure on the cache

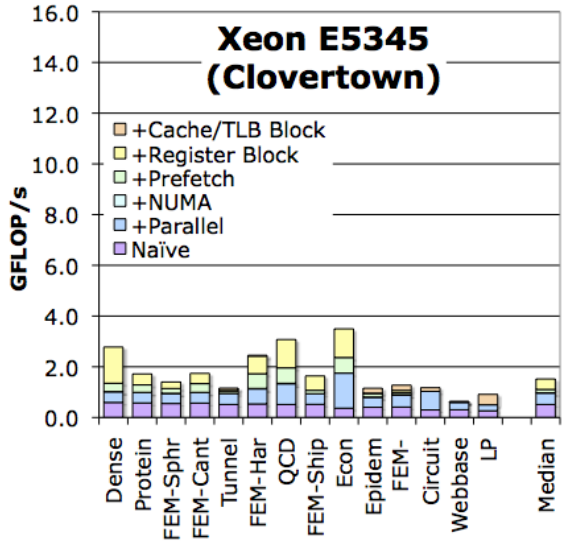
  i.e. touch the same number of unique source vector cache lines

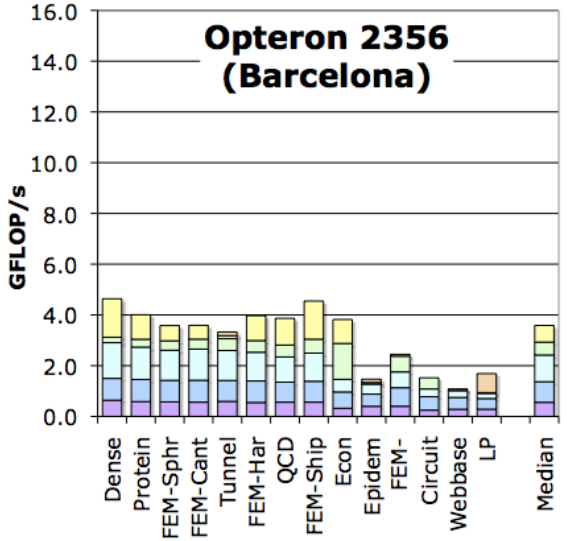
- TLB blocking is a similar concept but instead of on 64 byte granularities, it uses 4KB granularities

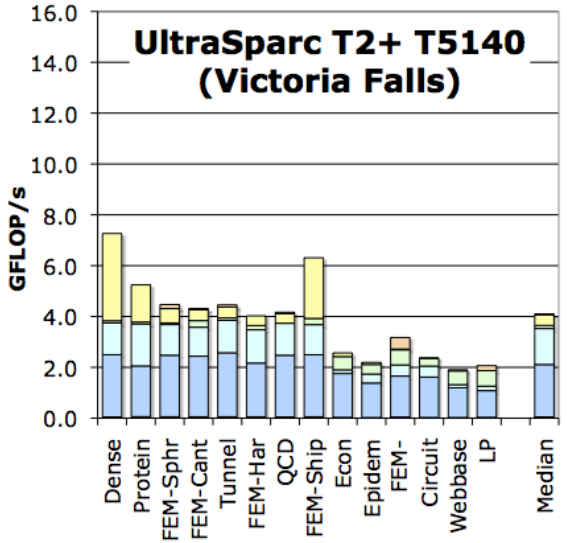


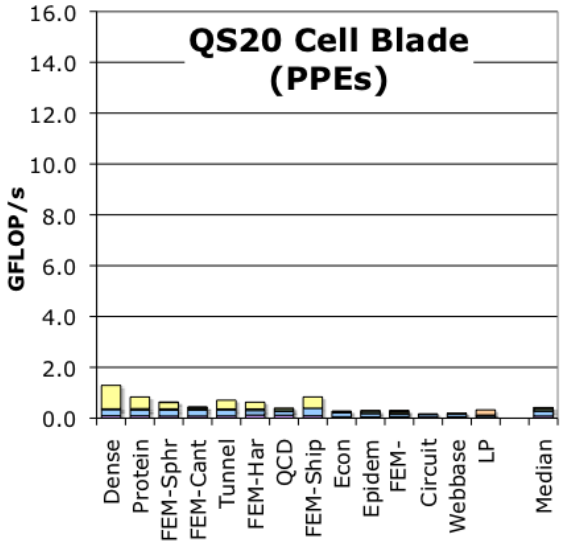
Auto-tuned SpMV Performance

(cache and TLB blocking)







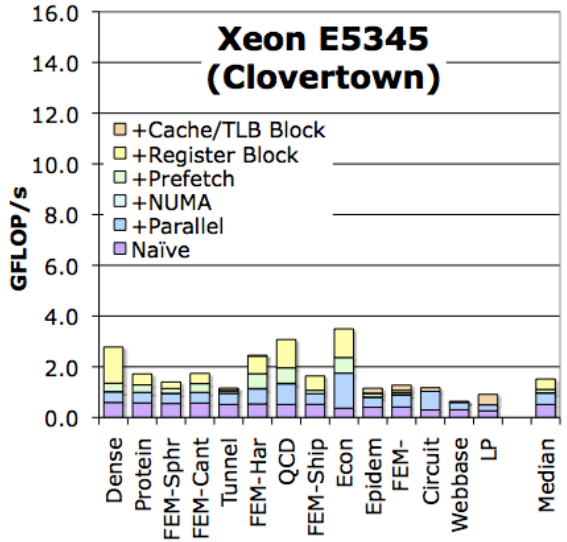


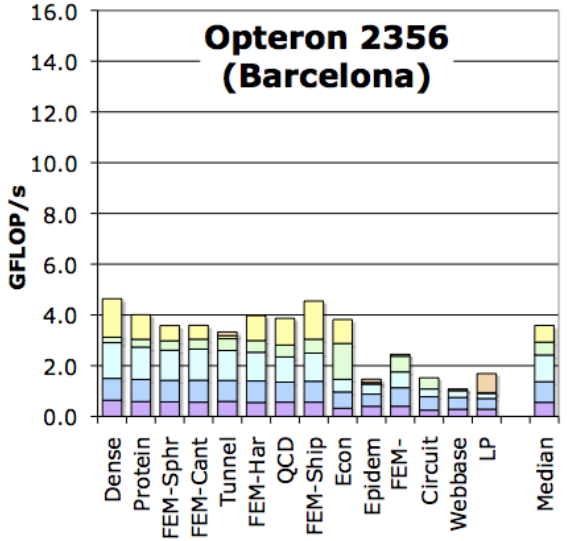
- Fully auto-tuned SpMV performance across the suite of matrices
- Why do some optimizations work better on some architectures?
- **matrices with naturally small working sets**
- **architectures with giant caches**

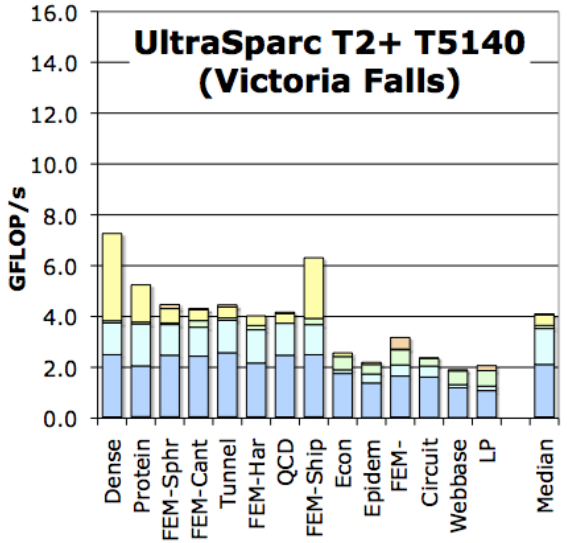
+Cache/LS/TLB Blocking
+Matrix Compression
+SW Prefetching
+NUMA/Affinity
Naïve Pthreads
Naïve

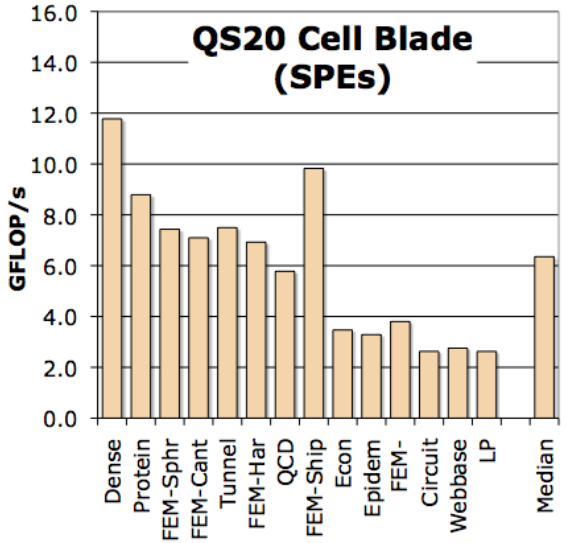
Auto-tuned SpMV Performance

(architecture specific optimizations)

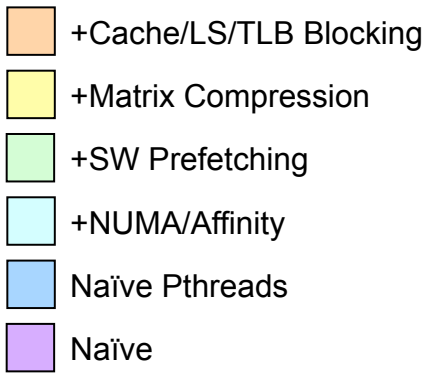




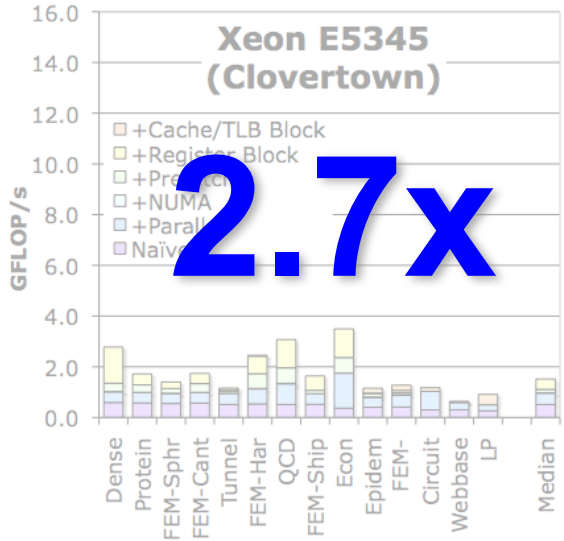


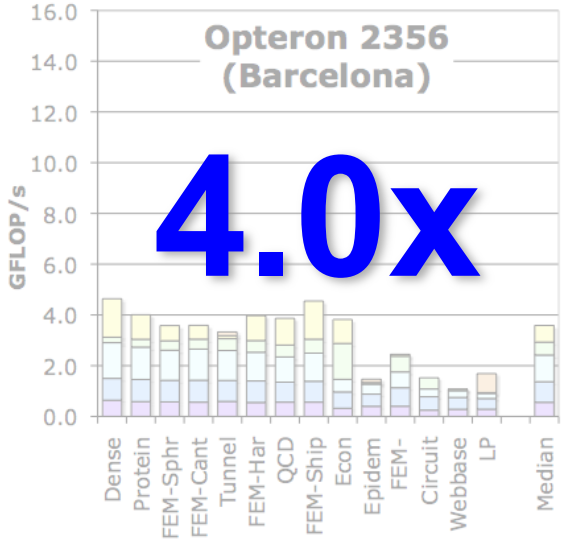


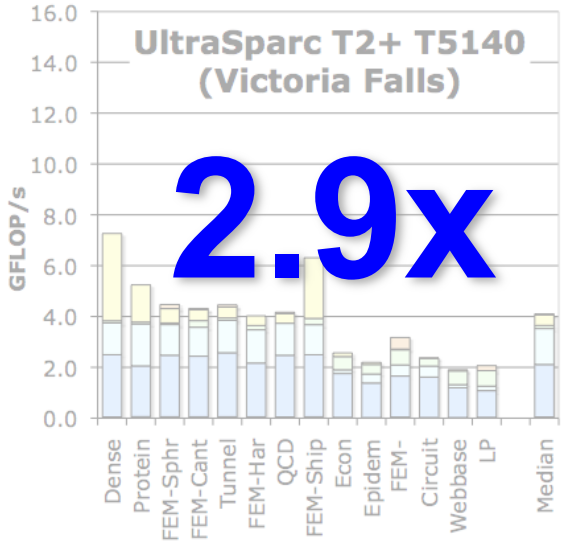
- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?

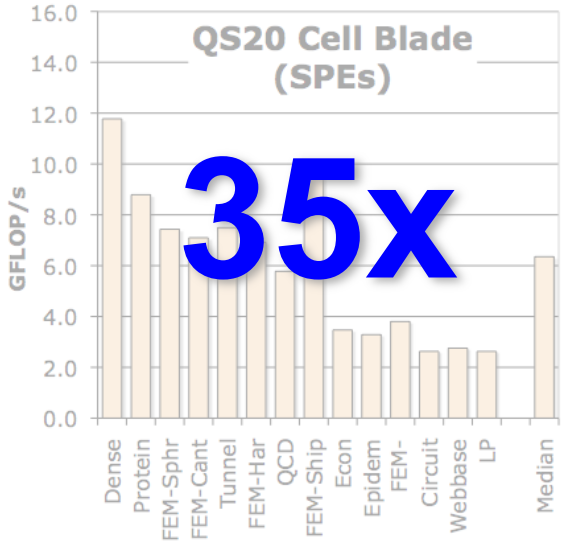


Auto-tuned SpMV Performance
(max speedup)

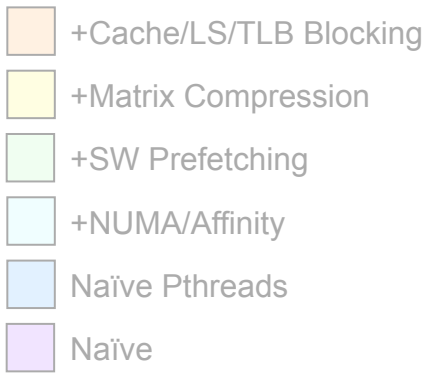








- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?



Summary

Summary

- There is a continual struggle between computer architects, mathematicians, and computer scientists.
  - architects will increase peak performance
  - architects may attempt to facilitate satisfying Little's Law
  - mathematicians create new, more efficient algorithms

- In order to minimize time to solution, we often must simultaneously satisfy Little's Law and minimize computation/communication.
  - Even if we satisfy little's Law, applications may be severely bottlenecked by computation/communication

- Perennially, we must manage:
  - **data/task locality**
  - **data dependencies**
  - **communication**
  - **variable and dynamic parallelism**

Summary (2)

- When optimizing code, the ideal solution for one machine is often found to be deficient on another.
- To that end, we are faced with the prospect of optimizing key computations for every architecture-input combination.

- Automatic Performance Tuning (**auto-tuning**) has been shown to mitigate these challenges by parameterizing some of the optimizations.
- Unfortunately, the more diverse the architectures the more we must rely on radically different implementations and algorithms to improve time to solution.

Questions?

Acknowledgments

Research supported by DOE Office of Science under contract number DE-AC02-05CH11231. All XT4/XE6 simulations were performed on the XT4 (Franklin) and XE6 (Hopper) at the National Energy Research Scientific Computing Center (NERSC). This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. 2005. George Vahala and his research group provided the original (FORTRAN) version of the LBMHD code. Jonathan Carter provided the optimized (FORTRAN) implementation that served as a basis for the LBMHD portion of this talk.

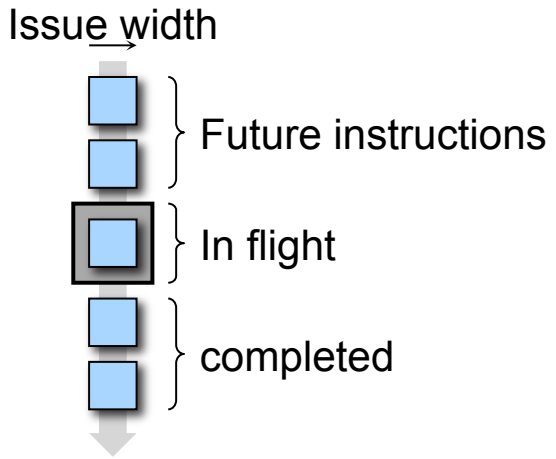
BACKUP SLIDES

Evolution of Computer Architecture and Little's Law

Yesterday's Constraint:
Instruction Latency & Parallelism

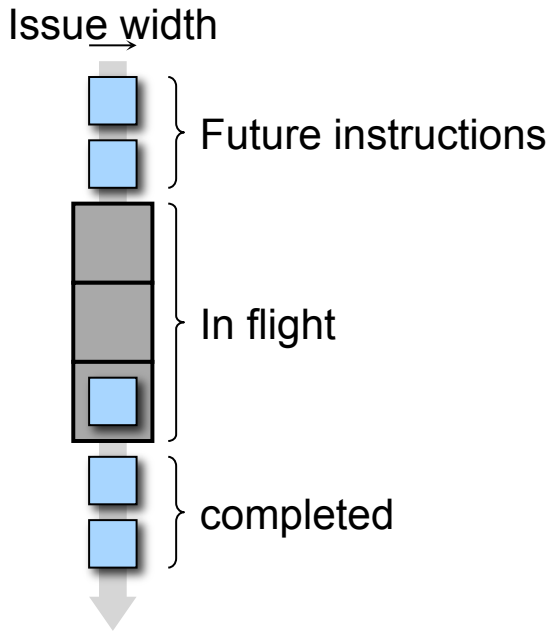
Single-issue, non-pipelined

- Consider a single issue, non-pipelined processor
- Little's Law
  - Bandwidth = issue width = 1
  - Latency = 1
  - Concurrency = 1
- Very easy to get good performance even if all instructions are dependent



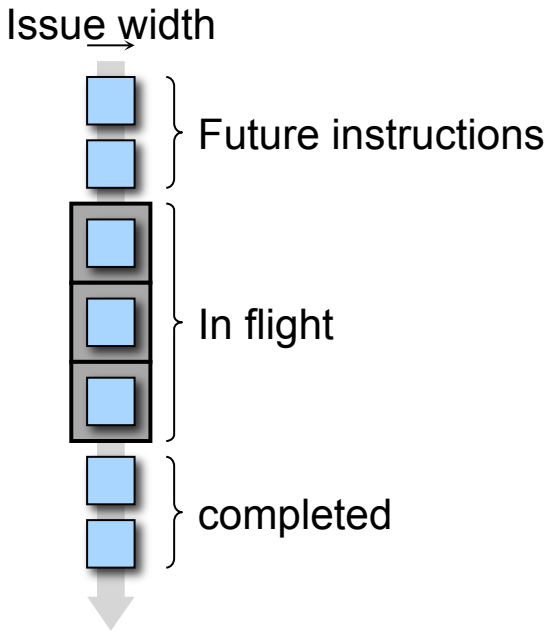
Pipelined

- By pipelining, we can increase the processor frequency.
- However, we must ensure the pipeline remains filled to achieve better performance.
- Little's Law
  - Bandwidth = issue width = 1
  - Latency = 3
  - Concurrency = 3
- Performance may drop to 1/3 of peak



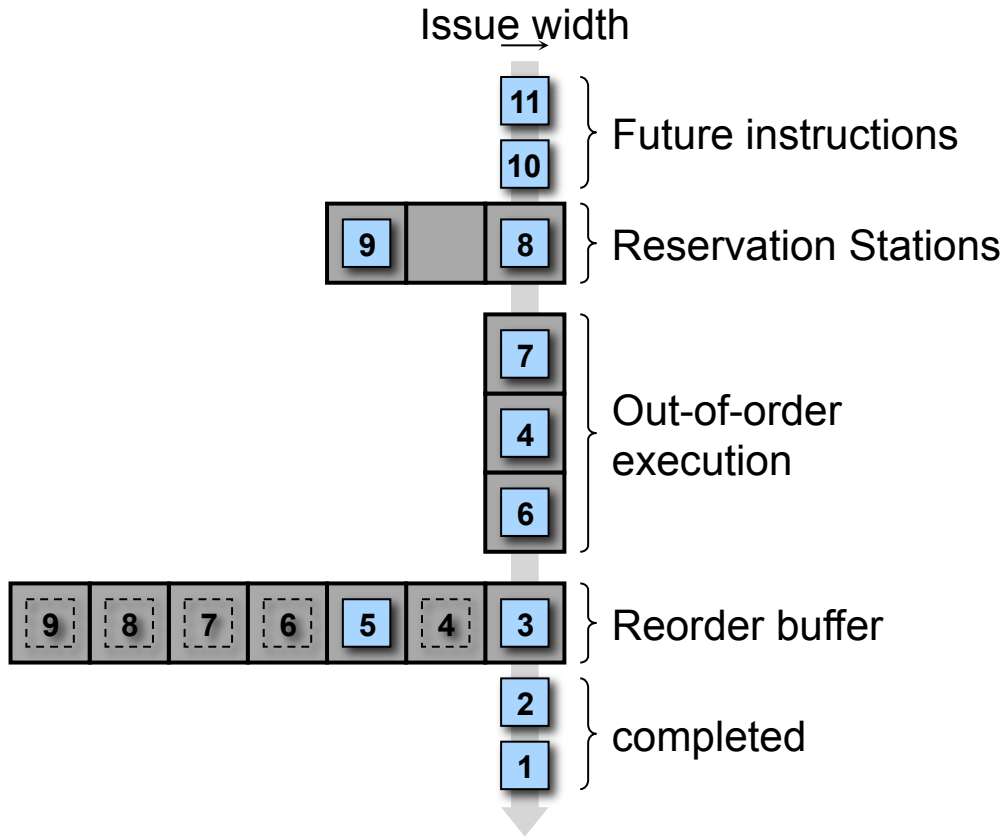
Pipelined
(w/unrolling, reordering)

- There may be inherent and untapped parallelism in the code
- Compilers/programmers must find parallelism, and unroll/ reorder the code to keep the pipeline full



Out-of-order

- Alternately, the hardware can try to find instruction level parallelism (ILP)
- Instructions are:
  - Queued up
  - Executed out-of-order
  - Reordered
  - Committed in-order
- Useful when parallelism or latency cannot be determined at compile time.

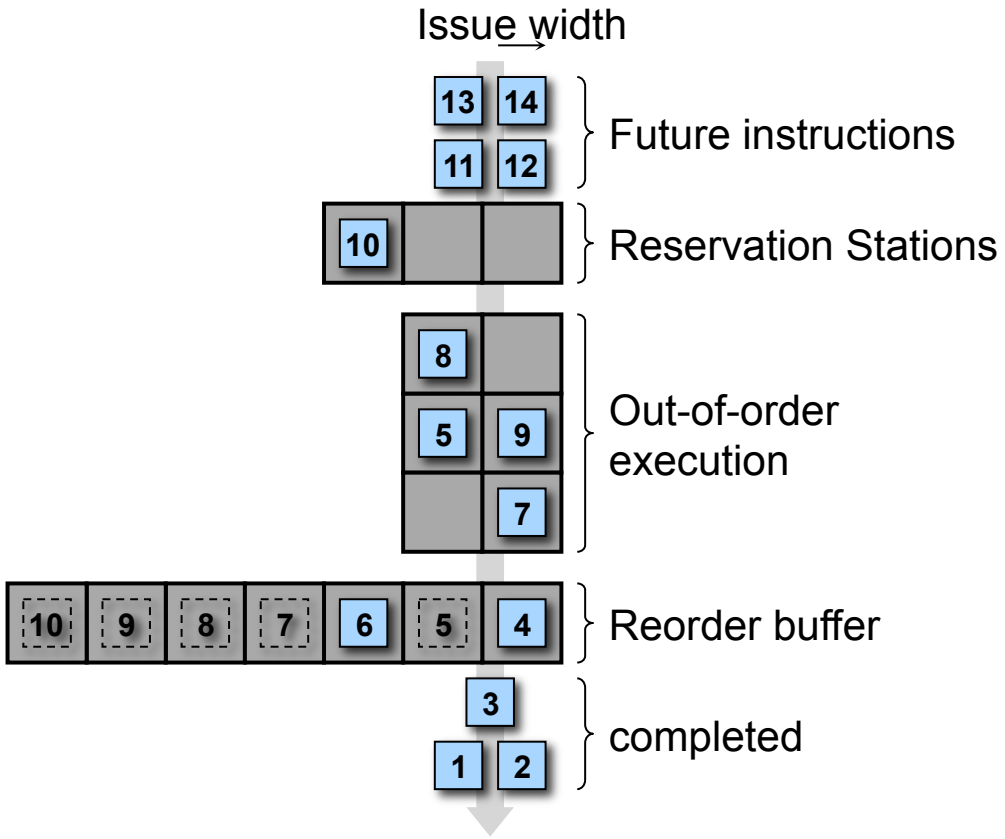


Superscalar

- Increase throughput, by executing multiple instructions in parallel
- Usually separate pipelines for different instruction types:

  FP, integer, memory
- Significantly complicates

  out-of-order execution



Instruction-Level Parallelism

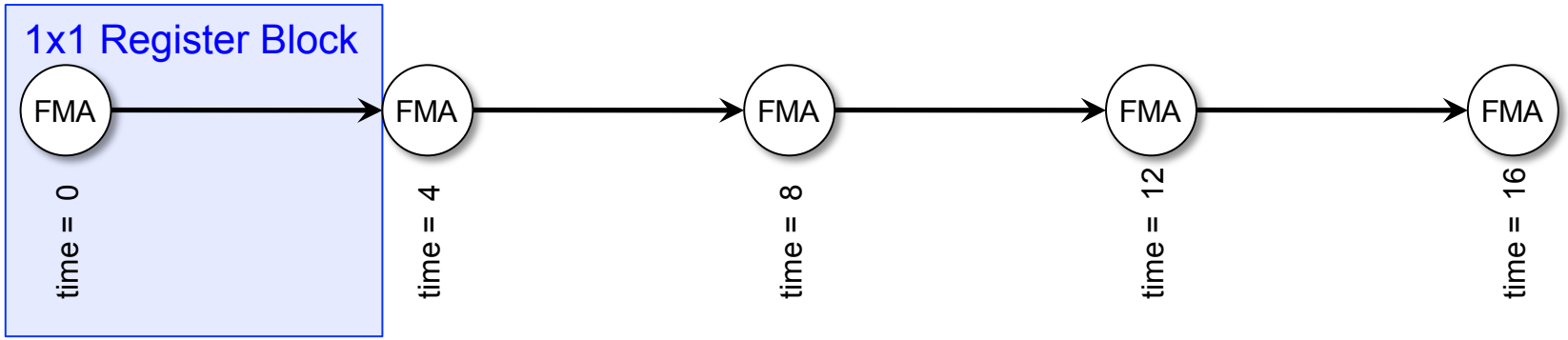
- On modern pipelined architectures, operations (like floating-point addition) have a latency of 4-6 cycles (until the result is ready).
- However, independent adds can be pipelined one after another.
- Although this increases the peak flop rate,
  - one can only achieve peak flops on the condition that on any given cycle the program has >4 independent adds ready to execute.
  - **failing to do so will result in a >4x drop in performance.**
- The problem is exacerbated by superscalar or VLIW architectures like POWER or Itanium.

- **One must often reorganize kernels to express more instruction-level parallelism**

ILP Example (1x1 BCSR)

- Consider the core of SpMV
- No ILP in the inner loop
- OOO can't accelerate serial FMAs

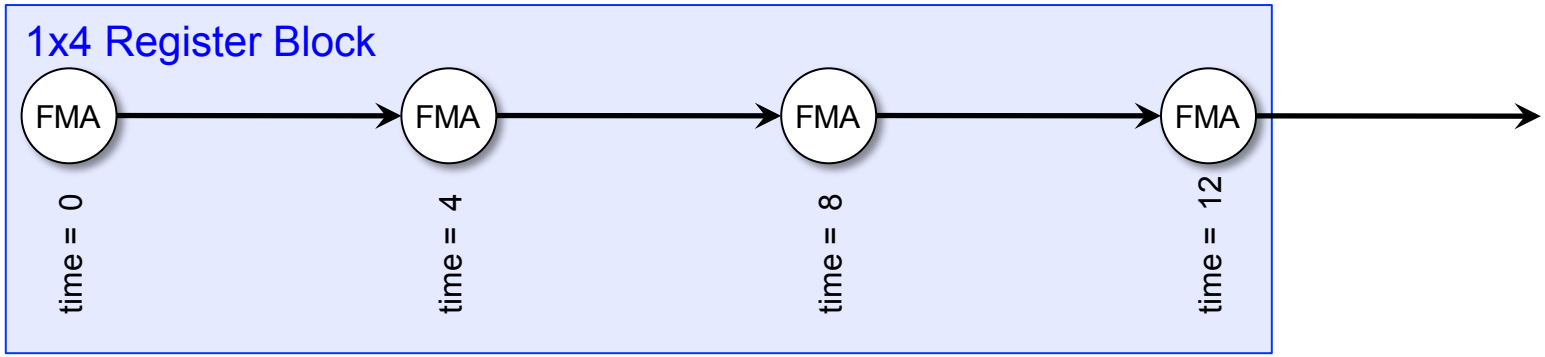
```
for(all rows){
  y0 = 0.0;
  for(all tiles in this row){
    y0+=V[i]*X[C[i]]
  }
  y[r] = y0;
}
```



ILP Example (1x4 BCSR)

- What about 1x4 BCSR ?
- Still no ILP in the inner loop
- FMAs are still dependent on each other

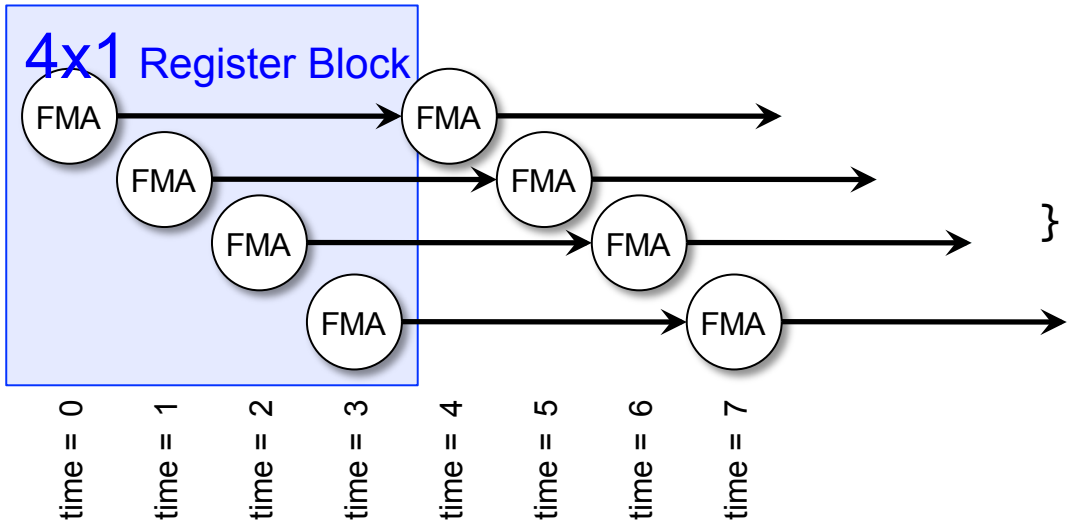
```
for(all rows){
  y0 = 0.0;
  for(all tiles in this row){
    y0+=V[i  ]*X[C[i]  ]
    y0+=V[i+1]*X[C[i]+1]
    y0+=V[i+2]*X[C[i]+2]
    y0+=V[i+3]*X[C[i]+3]
  }
  y[r] = y0;
}
```



ILP Example (4x1 BCSR)

- What about 4x1 BCSR ?
- Updating 4 different rows
- The 4 FMAs are independent
- Thus they can be pipelined.

```
for(all rows){
  y0 = 0.0;y1 = 0.0;
  y2 = 0.0;y3 = 0.0;
  for(all tiles in this row){
    y0+=V[i  ]*X[C[i]]
    y1+=V[i+1]*X[C[i]]
    y2+=V[i+2]*X[C[i]]
    y3+=V[i+3]*X[C[i]]
  }
  y[r+0] = y0; y[r+1] = y1;
  y[r+2] = y2; y[r+3] = y3;
}
```



SIMD

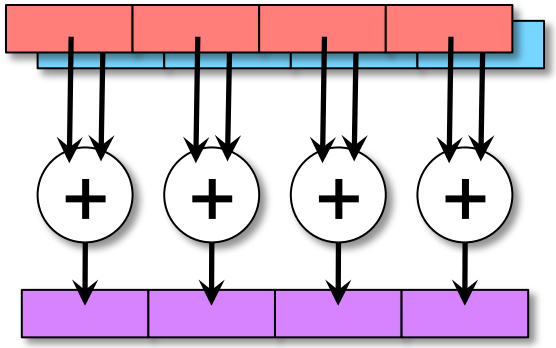
- Many codes perform the same operations on different pieces of data (Data level parallelism = DLP)
- SIMD : **S**ingle **I**nstruction **M**ultiple **D**ata

- Register sizes are increased.
- Instead of each register being a 64b FP #, each register holds 2 or 4 FP#'s

- **Much more efficient solution than superscalar on data parallel codes**

Data-level Parallelism

- DLP = apply the same operation to multiple independent operands.



- Today, rather than relying on superscalar issue, many architectures have adopted SIMD as an efficient means of boosting peak performance. (SSE, Double Hummer, AltiVec, Cell, GPUs, etc…)
- Typically these instructions operate on four single precision (or two double precision) numbers at a time.
- However, some are more GPUs(32), Larrabee(16), and AVX(8)
- **Failing to use these instructions may cause a 2-32x drop in performance**
- **Unfortunately, most compilers utterly fail to generate these instructions.**

Memory-Level Parallelism (1)

- Although caches may filter many memory requests, in HPC many memory references will still go all the way to DRAM.
- Memory latency (as measured in core cycles) grew by an order of magnitude in the 90's
- **Today, the latency of a memory operation can exceed 200 cycles (1 double every 80ns is unacceptably slow).**
- Like ILP, we wish to pipeline requests to DRAM
- Several solutions exist today
  - HW stream prefetchers
  - HW Multithreading (e.g. hyperthreading)
  - SW line prefetch
  - DMA

Memory-Level Parallelism (2)

- HW stream prefetchers are by far the easiest to implement and exploit.
- They detect a series of consecutive cache misses and speculate that the next addresses in the series will be needed. They then prefetch that data into the cache or a dedicated buffer.
- To effectively exploit a HW prefetcher, ensure your array references accesses 100's of consecutive addresses.

- **e.g. read `A[i]…A[i+255]` without any jumps or discontinuities**

- This force limits the effectiveness (shape) of the cache blocking you implemented in HW1 as you accessed:

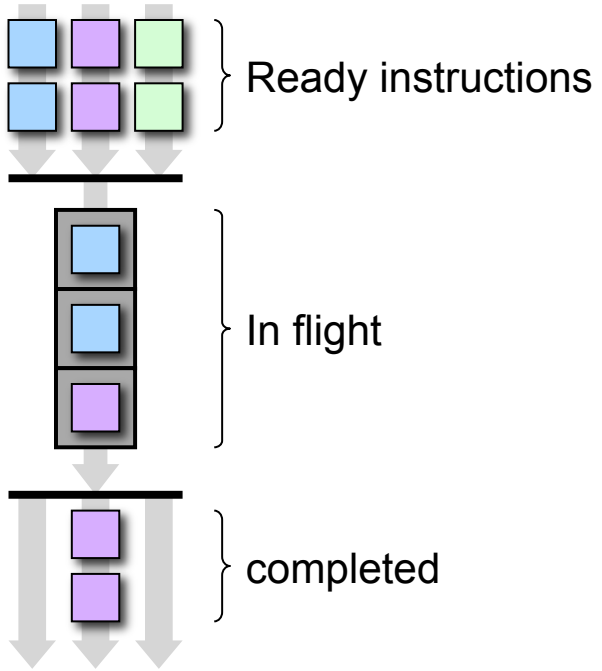
```
A[(j+0)*N+i]…A[(j+0)*N+i+B], jump
A[(j+1)*N+i]…A[(j+1)*N+i+B], jump
A[(j+2)*N+i]…A[(j+2)*N+i+B], jump
…
```

Multithreaded

- Superscalars fail when there is no ILP or DLP
- However, there are many codes with
  thread-level parallelism (TLP)
- Consider architectures that are virtualized to appear as N cores.
- In reality, there is one core maintaining multiple contexts and dynamically switching between them
- There are 3 main types of multithread architectures:
  - **Coarse-grained multithreading (CGMT)**
  - **Fine-grained multithreading (FGMT) ,** aka Vertical Multithreading
  - **Simultaneous multithreading (SMT)**

Coarse-grained Multithreading

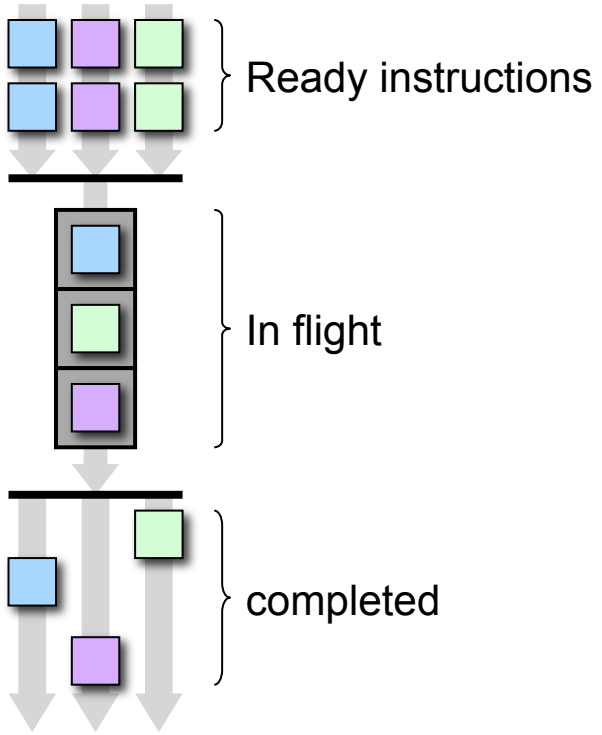
- Maintain multiple contexts
- On a long latency instruction:
  - dispatch instruction
  - Switch to a ready thread
  - Hide latency with multiple ready threads
  - Eventually switch back to original



Fine-grained Multithreading

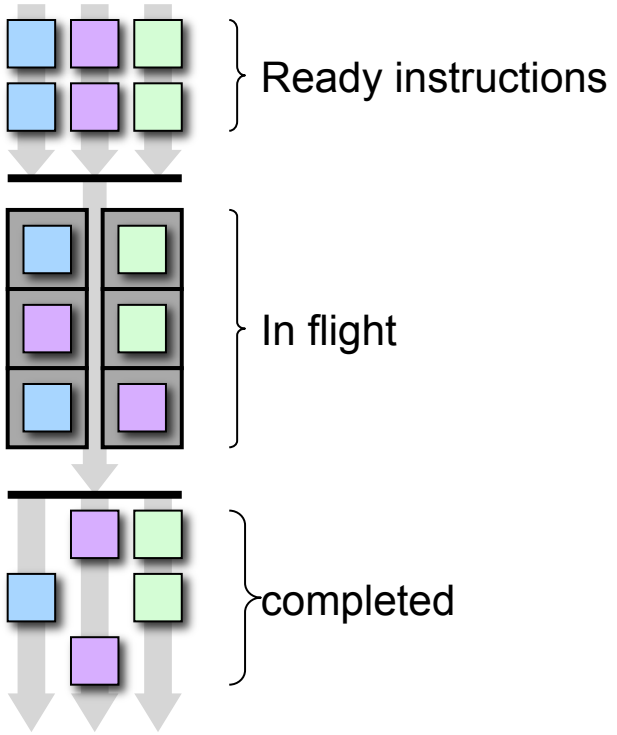
- Maintain multiple contexts
- On every cycle choose a ready thread
- May now satisfy Little's Law through multithreading:

  threads ~ latency * bandwidth



Simultaneous Multithreading

- Maintain multiple contexts
- On every cycle choose as many ready instructions from the thread pool as possible
- Can be applied to both in-order and out-of-order architectures

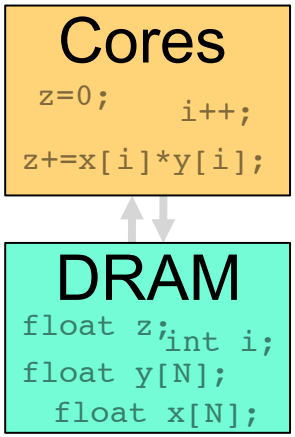


Today's Constraint:
The Memory Wall

Abstract Machine Model

(as seen in programming model)

- In the abstract, processor architectures appear to have just memory, and functional units.
- On early HPC machines, reading from memory required just one cycle.



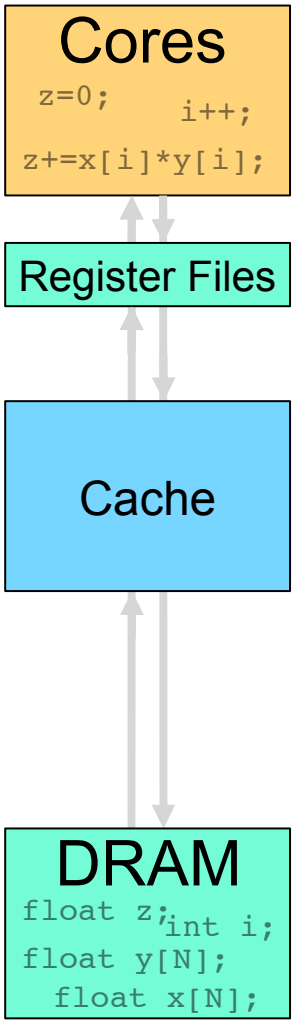
Abstract Machine Model

(as seen in programming model)

- In the abstract, processor architectures appear to have just memory, and functional units.
- On early HPC machines, reading from memory required just one cycle.

- Unfortunately, as processors developed, DRAM latencies (in terms of core cycles) dramatically increased.
- Eventually a small memory (the register file) was added so that one could hide this latency (by keeping data in the RF)
- The programming model and compiler evolved to hide the fact the management of data locality in the RF.

- Unfortunately, today, latency to DRAM can be 1000x that to the register file.
- As the RF is too small for today's problems, architects inserted another memory (cache) between the register file and the DRAM.
- Data is transparently copied into the cache for future reference.
- This memory is entirely invisible to the programmer and the compiler, but still has latency 10x higher than the register file.

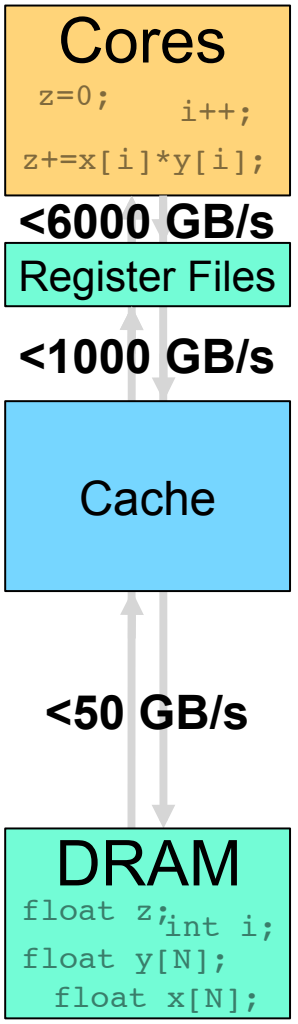


Abstract Machine Model

(as seen in programming model)

- Not only are the differences in latencies substantial, so to are the bandwidths
- Once a link has been saturated (Little's law is satisfied), it acts as a bottleneck against increased performance.

- The only solution is to **reduce the volume of traffic** across that link



Impact on Little's Law ?

- Today, utilizing the **full DRAM bandwidth** and **minimizing memory traffic** are paramount.

- DRAM latency can exceed **1000 cpu cycles**.
- Impact on Little's Law (200ns * 20GB/s): **4KB of data in flight**

- **How did architects solve this?**

out-of-order ?

- Out-of-order machines can only scale to ~100 instructions in flight of which only **~40 can be loads**
- This is ~10% of what Little's Law requires

- Out-of-order execution can now only hide cache latency, not DRAM latency

Software prefetch ?

- A software prefetch is an instruction similar to a load
- However,
  - It does not write to a register
  - Its execution is decoupled from the core
  - It is designed to bring data into the cache before it is needed
  - It must be scheduled by software early enough to hide DRAM latency
- Limited applicability
  - work best on patterns for which many addresses are known well in advance.
  - must be inserted by hand with the distance tuned for

Hardware Stream Prefetchers ?

- Hardware examines the cache miss pattern
- Detects unit-stride (now strided) miss patterns, and begins to prefetch data before the next miss occurs.

- Summary
  - Only works for simple memory access patterns
  - Can be tripped up if there are too many streams (>8)
  - Cannot prefetch beyond TLB page boundaries

Local Store + DMA

- Create an on-chip memory(**Local Store**) disjoint from the cache/ TLB-heirarchy
- Use DMA to transfer data from DRAM to local store
  - Basic operation: specify a long contiguous transfer (**unit stride**)
  - Can be extended to specifying a list of transfers (**random access**)
  - DMA is decoupled from execution (poll to check for completion)
- Allows one to efficiently satisfy the concurrency from Little's Law:
  - Concurrency = #DMAs * DMA length
  - #DMAs * DMA length
  - #DMAs * DMA length
- **Requires major software effort**

Multithreading

- Another approach to satisfying Little's law
- But more threads/core -> less cache(& associativity) per thread

- Allow one cache miss per thread
  - #threads * 1 cacheline vs. Latency * Bandwidth
  - #threads = Latency*Bandwidth / cacheline ~ 64

- 64 threads/core is unrealistically high

Roofline Model

Roofline Model

Basic Concept

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

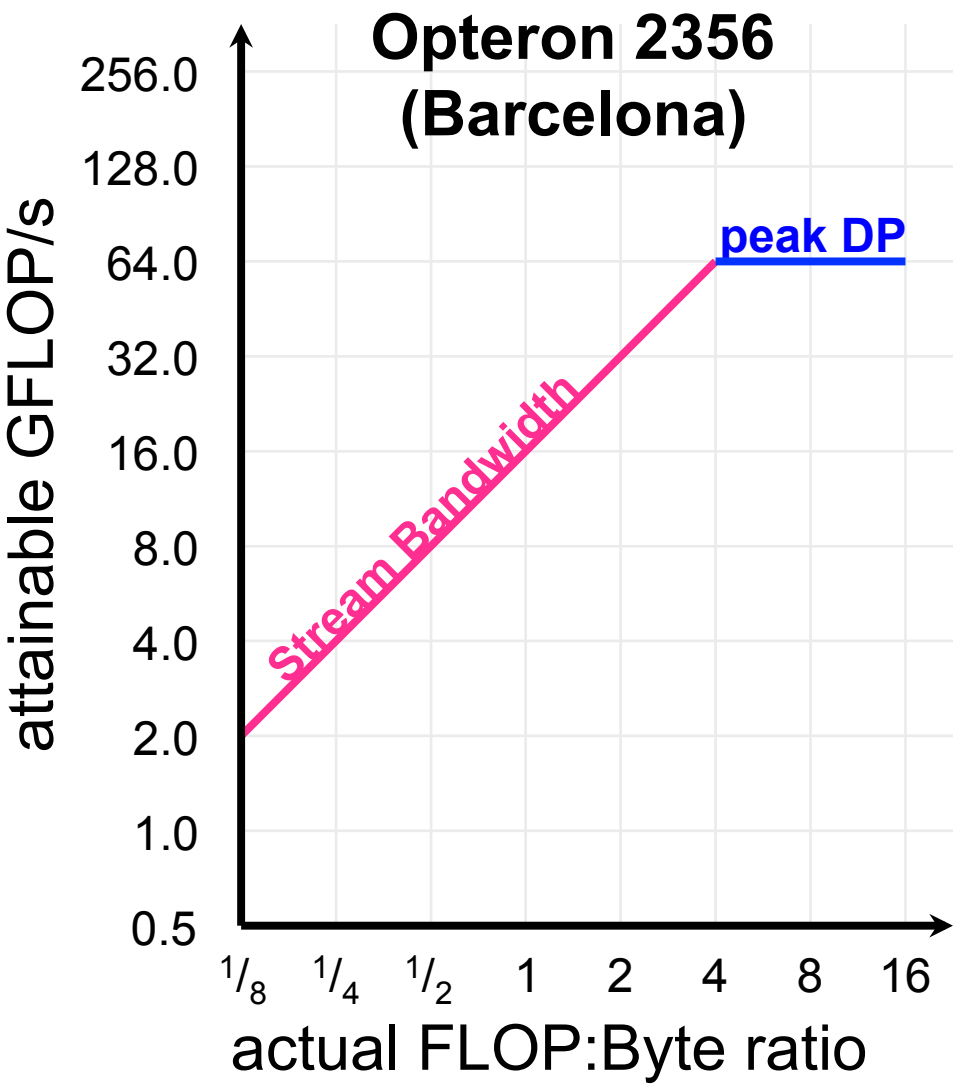
$$\text{Attainable Performance}_{ij} = \min \begin{cases} \text{FLOP/s with Optimizations}_{1\text{-}i} \\ \text{AI} * \text{Bandwidth with Optimizations}_{1\text{-}j} \end{cases}$$

- where *optimization i* can be SIMDize, or unroll, or SW prefetch, …
- 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.

Roofline Model

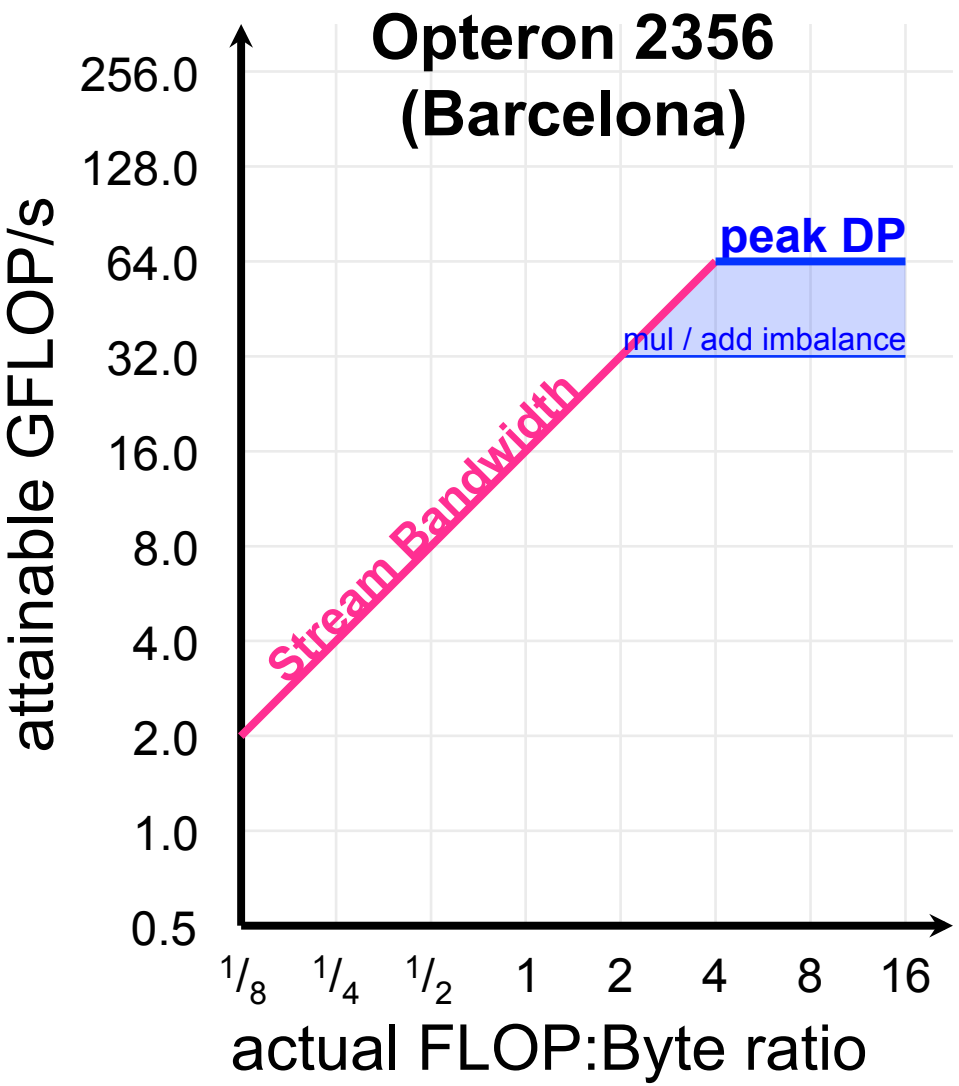
Basic Concept



- 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

Roofline Model

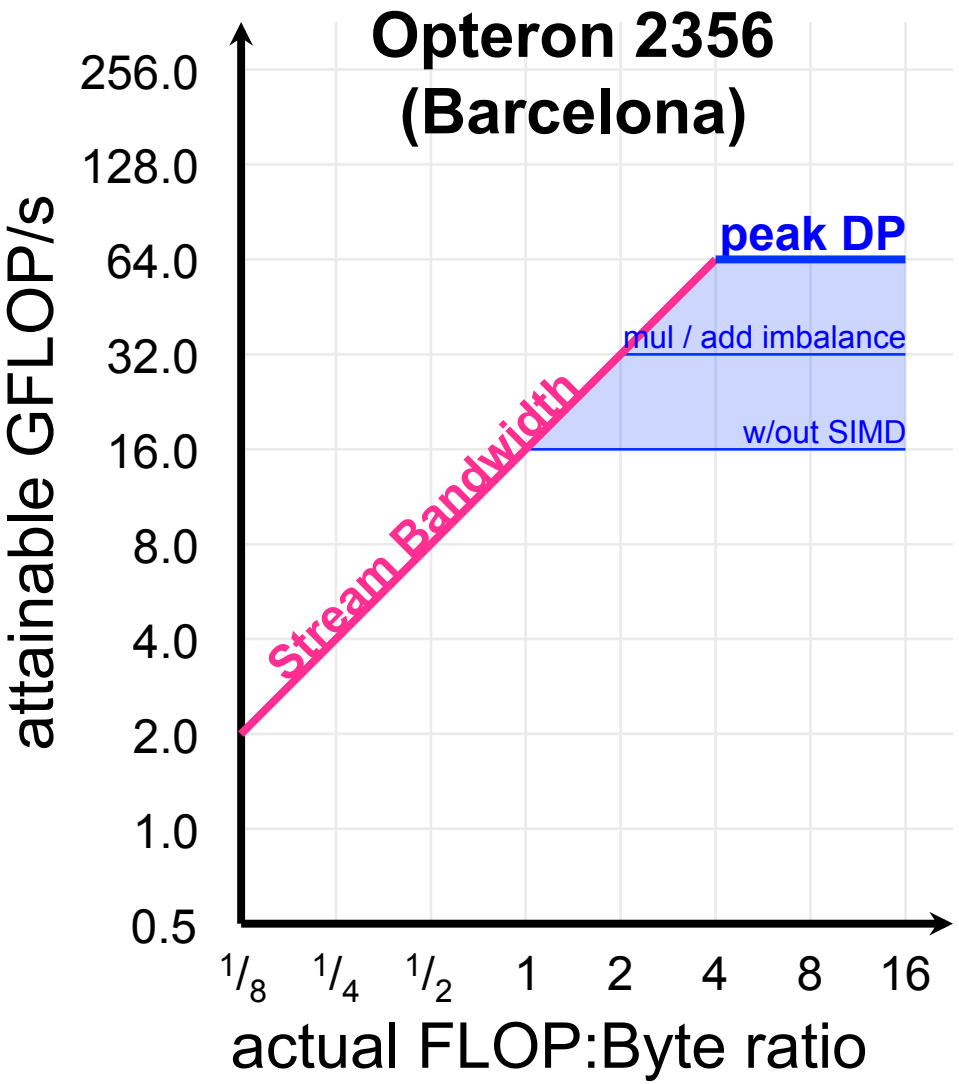
computational ceilings



- 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

Roofline Model

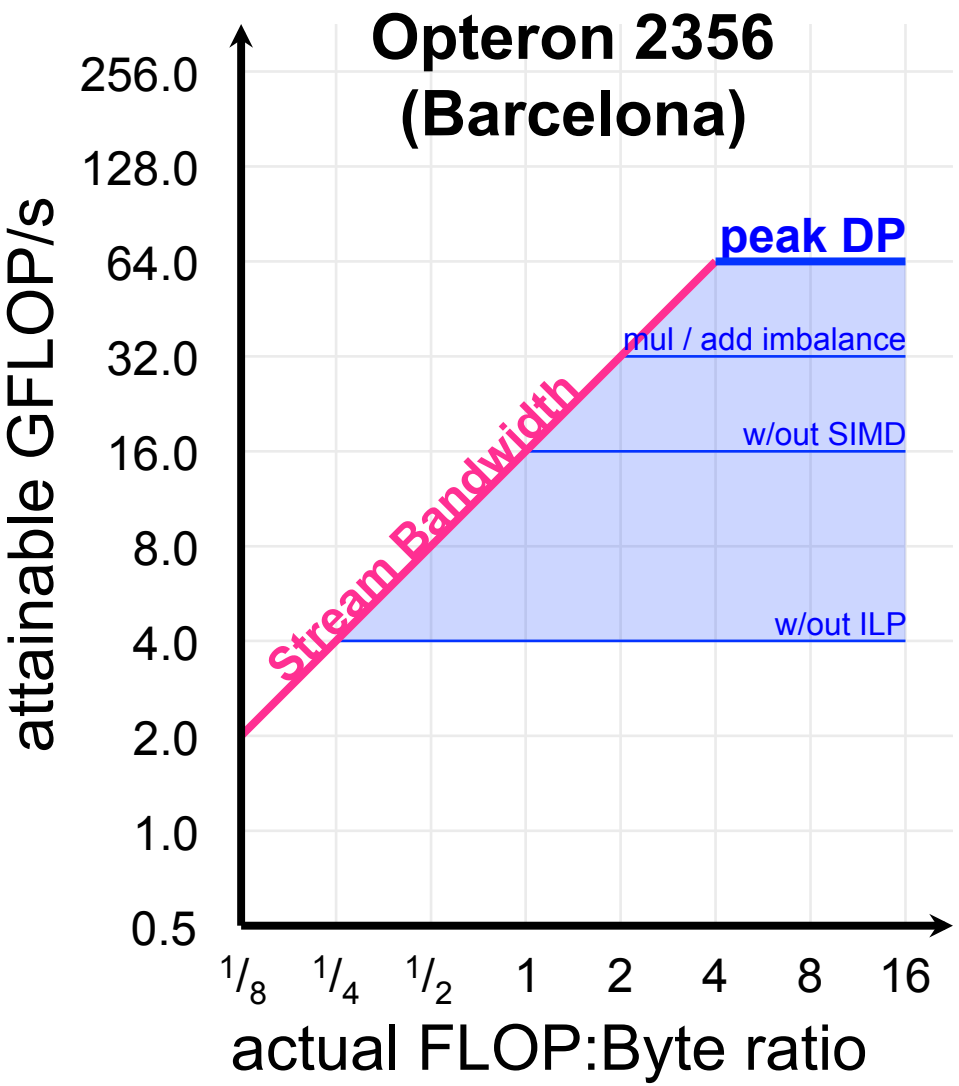
computational ceilings



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

Roofline Model

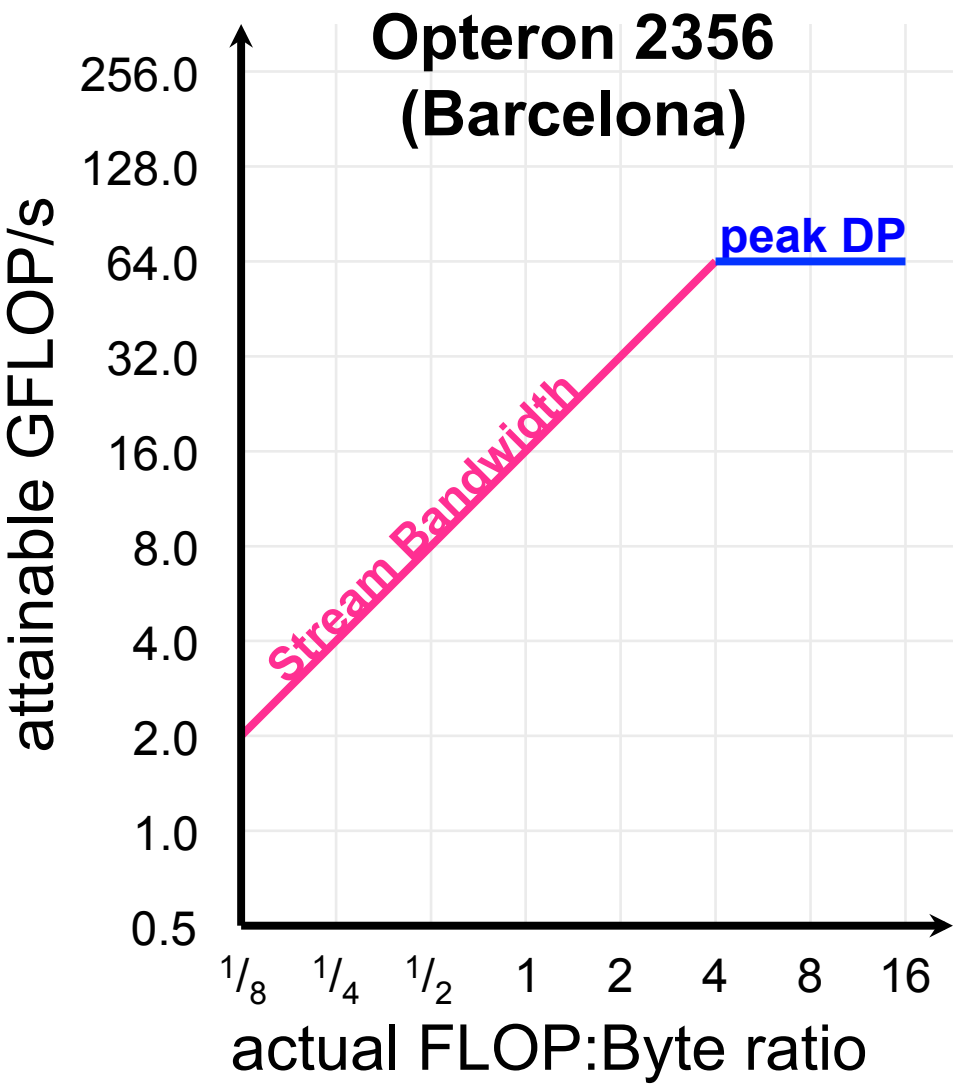
computational ceilings



- 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

Roofline Model

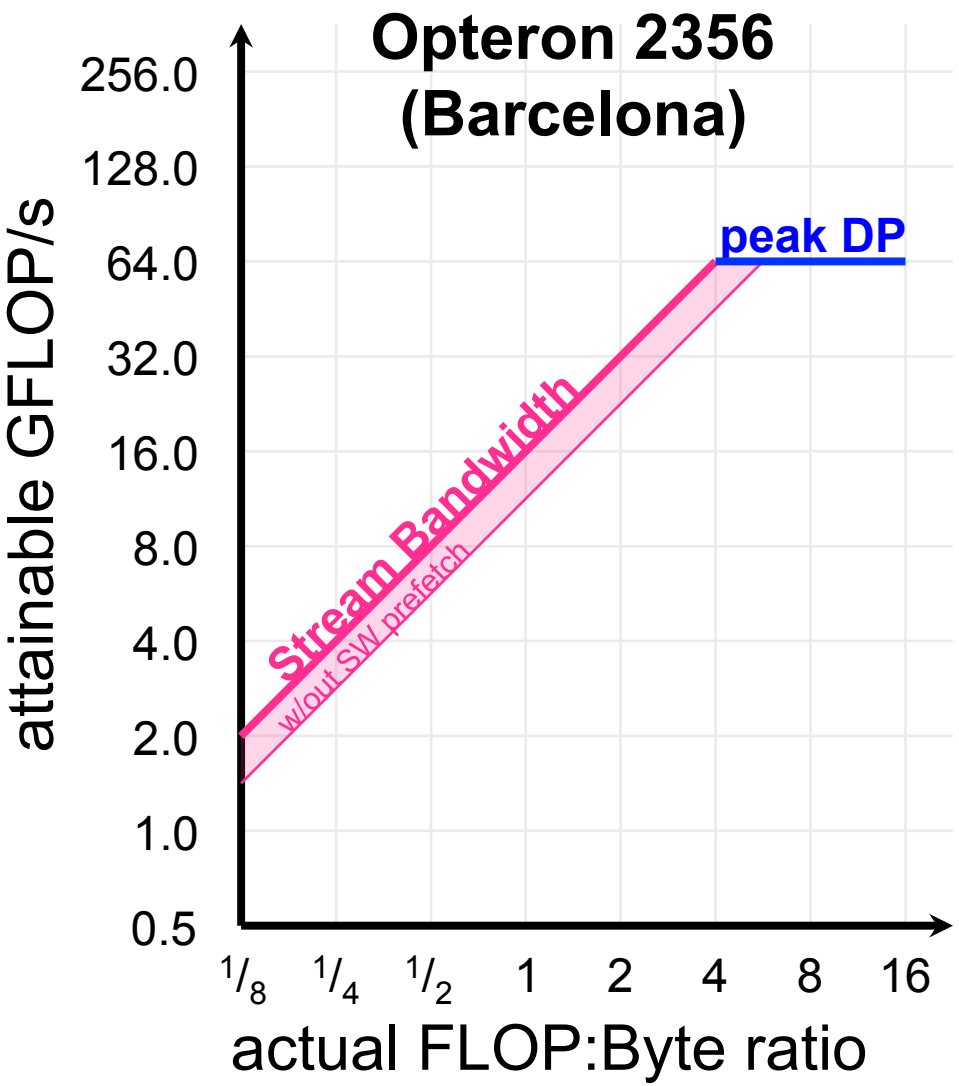
communication ceilings



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

Roofline Model

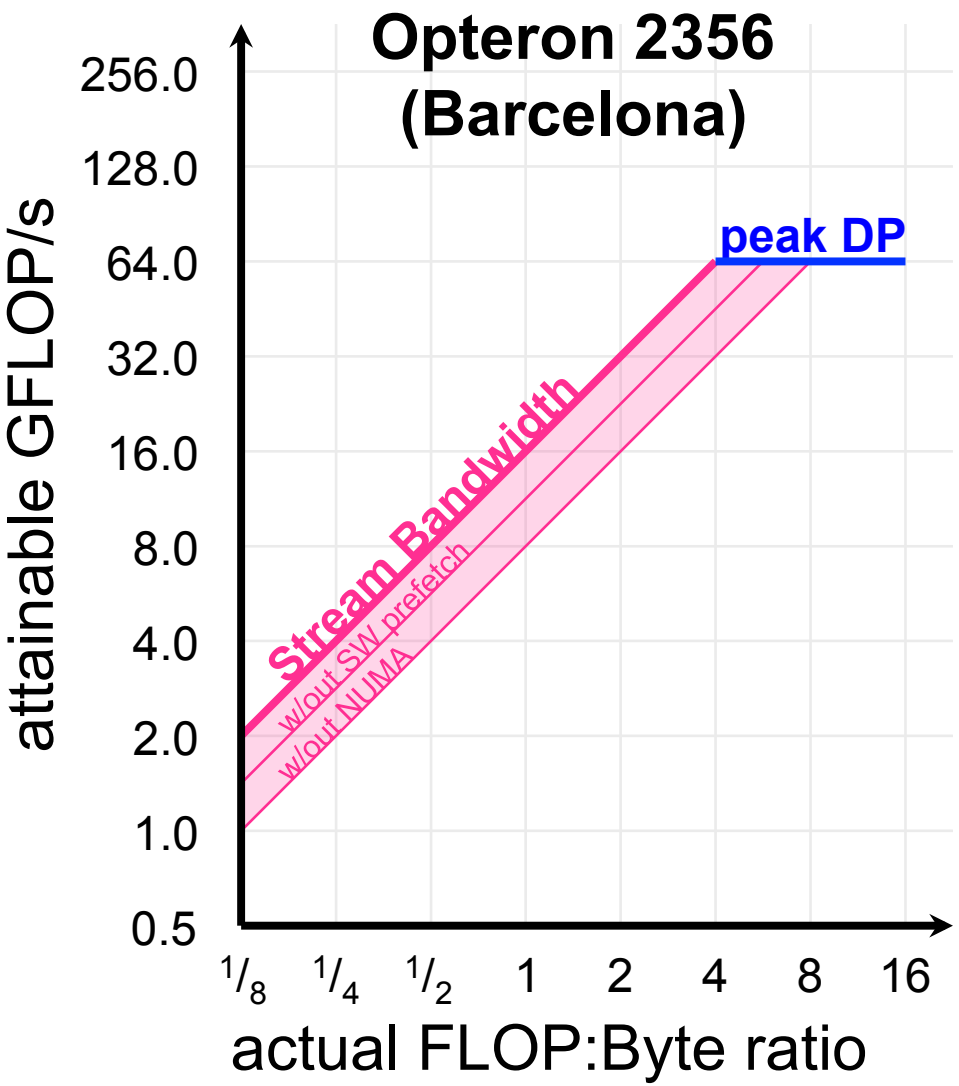
communication ceilings



- Explicit software prefetch instructions are required to achieve peak bandwidth

Roofline Model

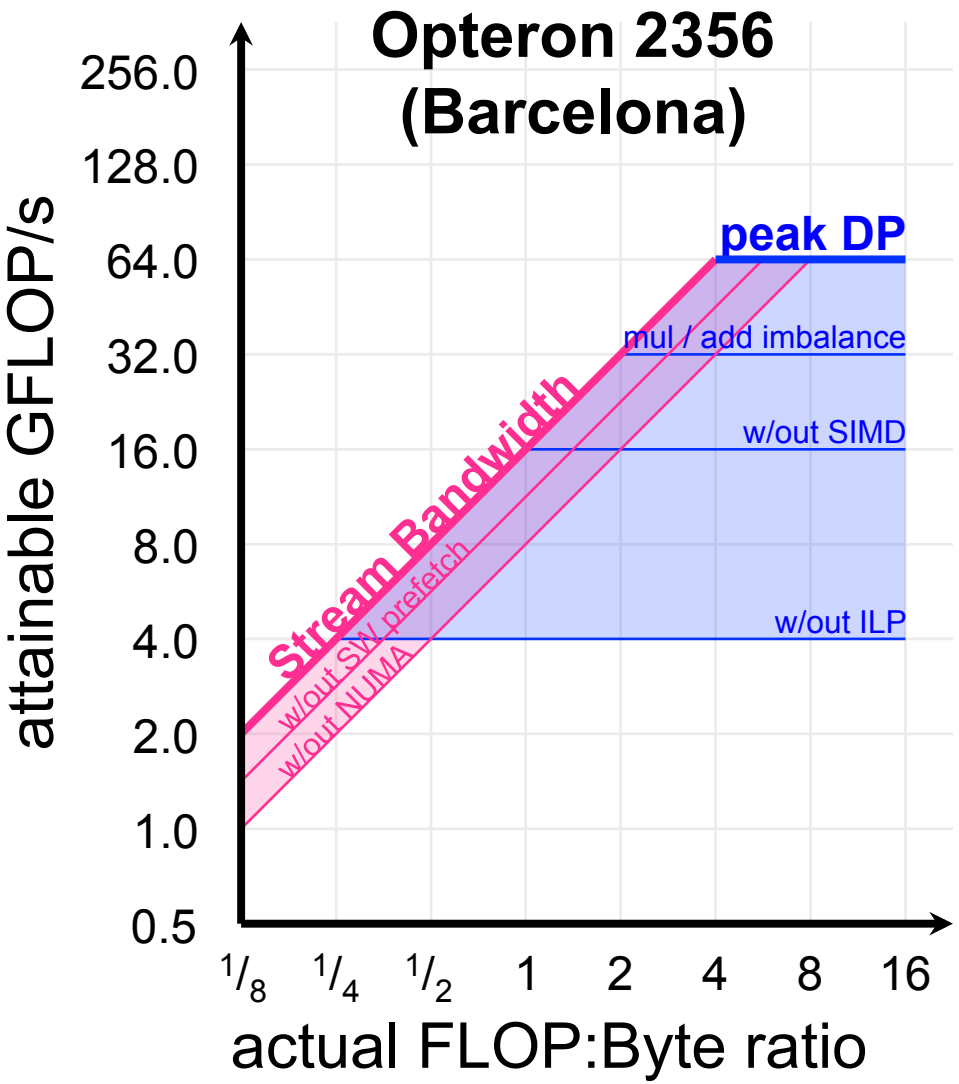
communication ceilings



- 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

Roofline Model

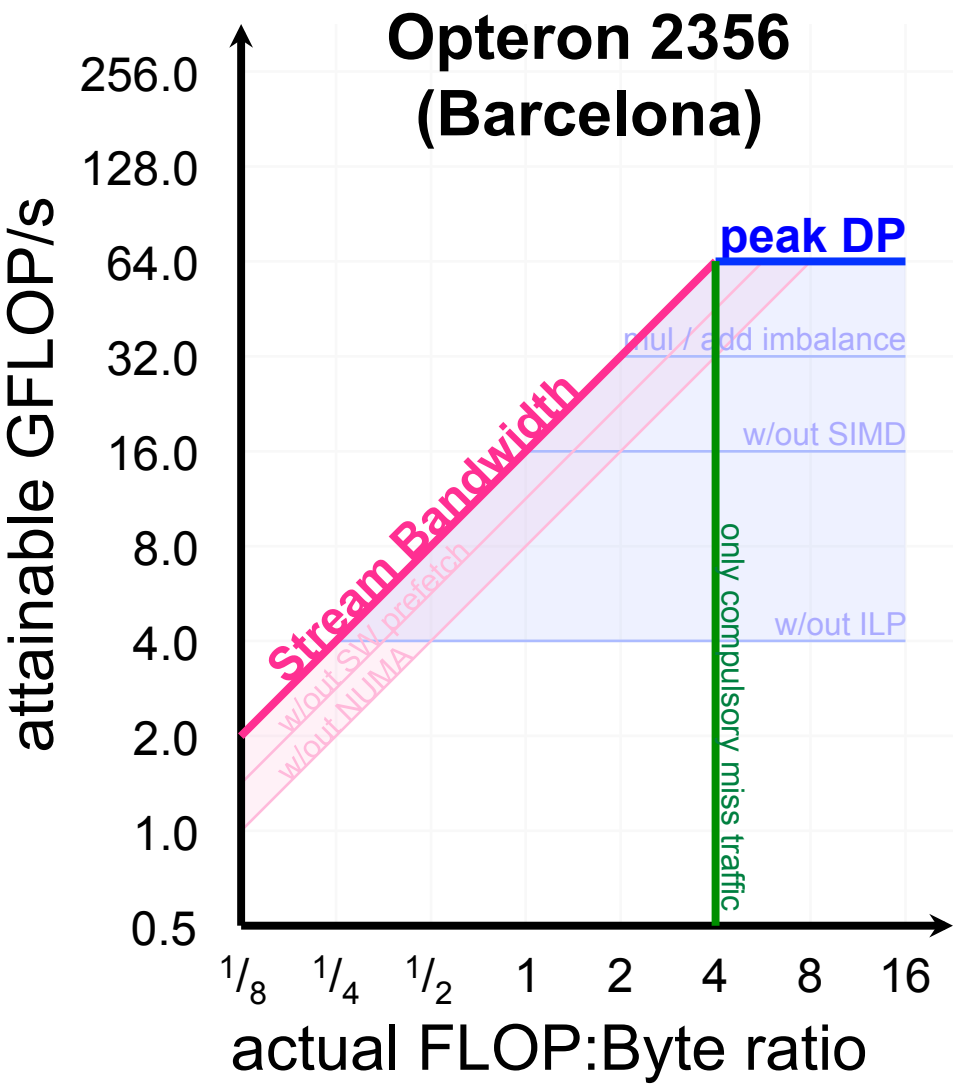
computation + communication ceilings



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

Roofline Model

locality walls

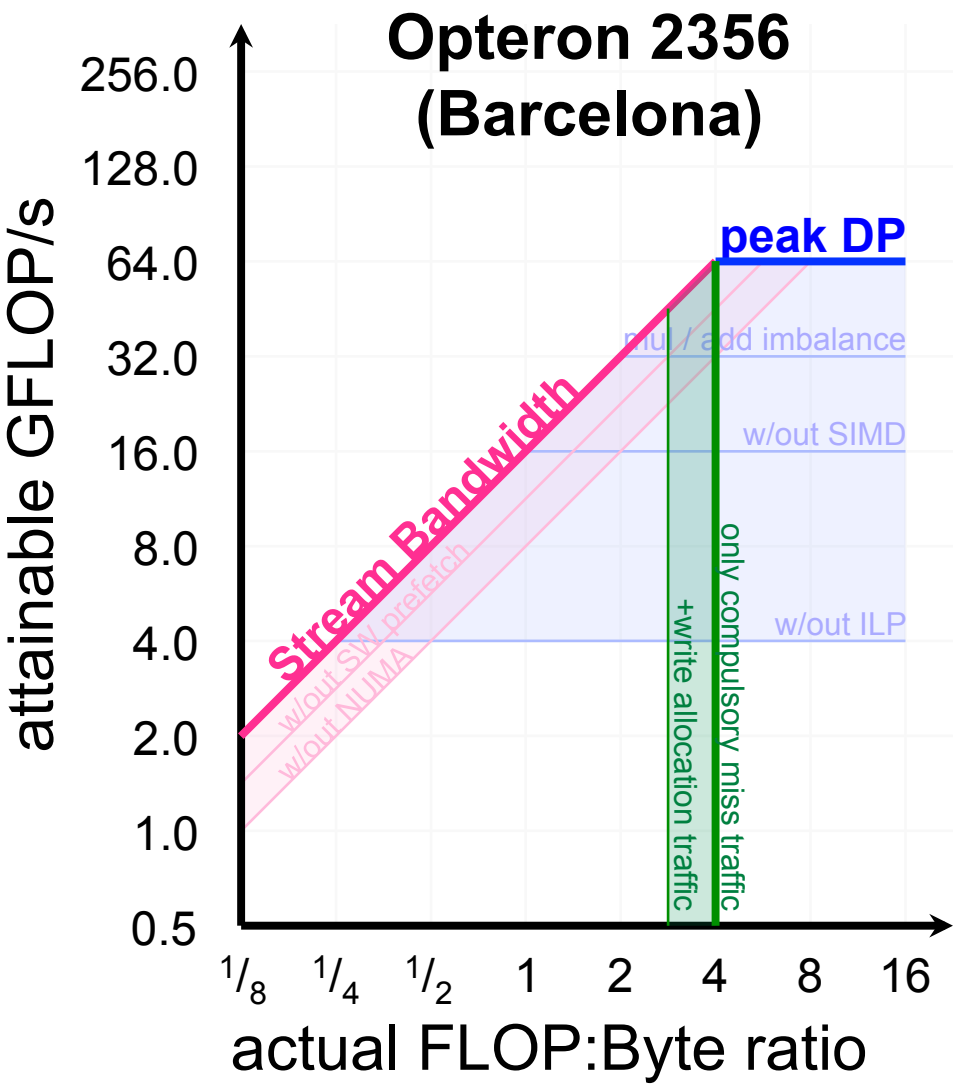


- Remember, memory traffic includes more than just compulsory misses.
- As such, actual arithmetic intensity may be substantially lower.
- Walls are unique to the architecture-kernel combination

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

Roofline Model

locality walls

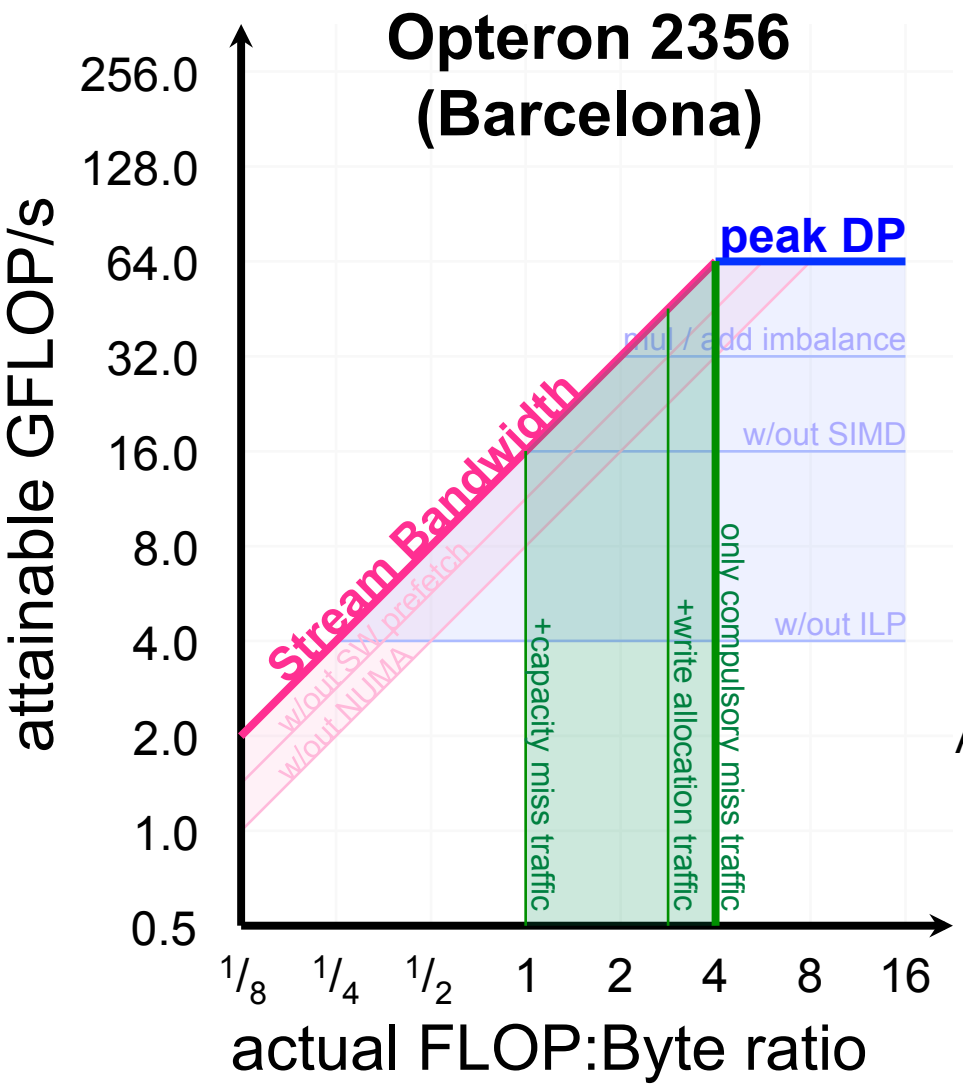


- Remember, memory traffic includes more than just compulsory misses.
- As such, actual arithmetic intensity may be substantially lower.
- Walls are unique to the architecture-kernel combination

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

Roofline Model

locality walls

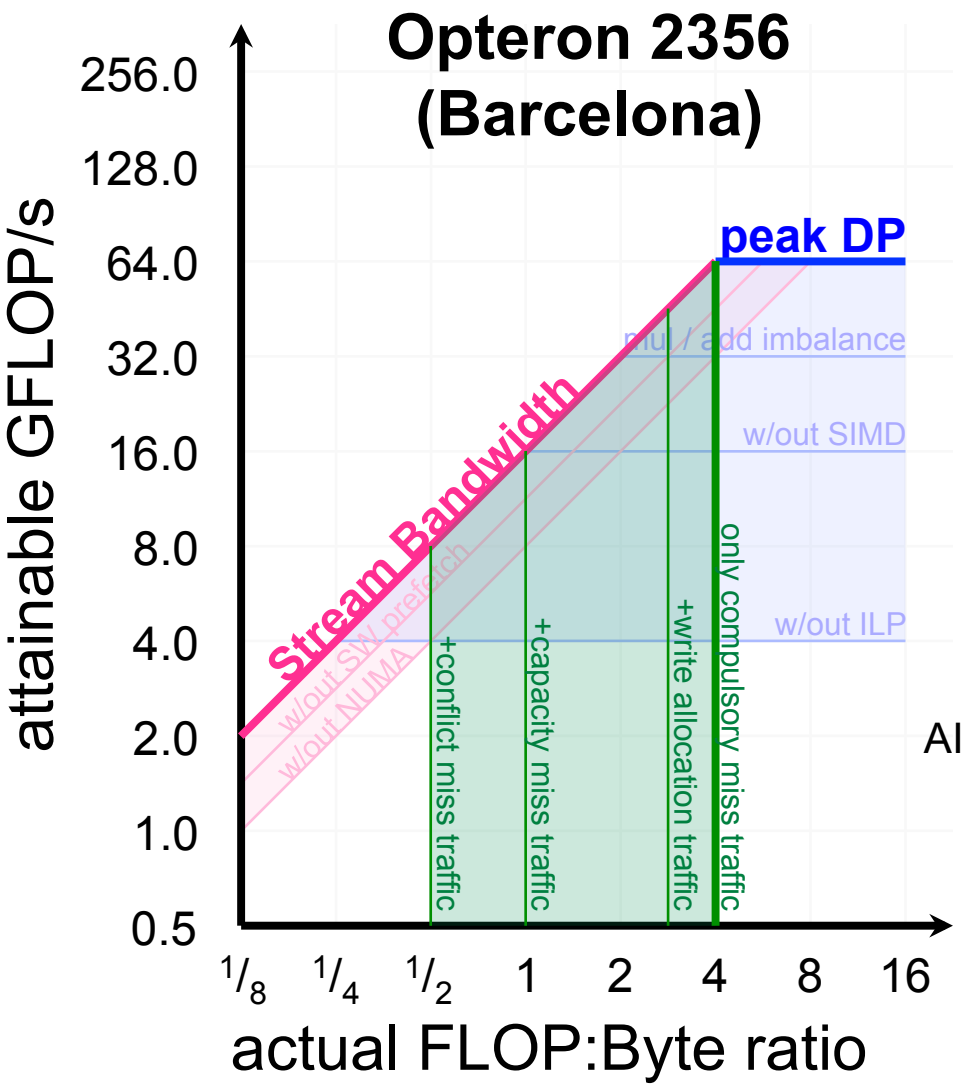


- Remember, memory traffic includes more than just compulsory misses.
- As such, actual arithmetic intensity may be substantially lower.
- Walls are unique to the architecture-kernel combination

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

Roofline Model

locality walls



- 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

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

Optimization Categorization

Maximizing *(attained)* **In-core Performance**	**Maximizing** *(attained)* **Memory Bandwidth**	**Minimizing** *(total)* **Memory Traffic**

Optimization Categorization

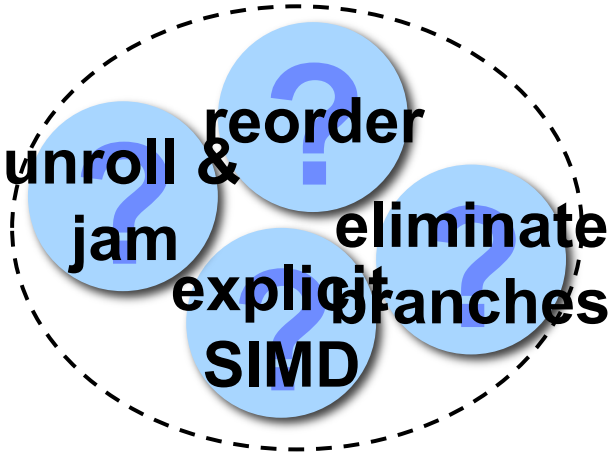
Maximizing In-core Performance	Maximizing Memory Bandwidth	Minimizing Memory Traffic

- Exploit in-core parallelism (ILP, DLP, etc…)
- Good (enough) floating-point balance

Optimization Categorization

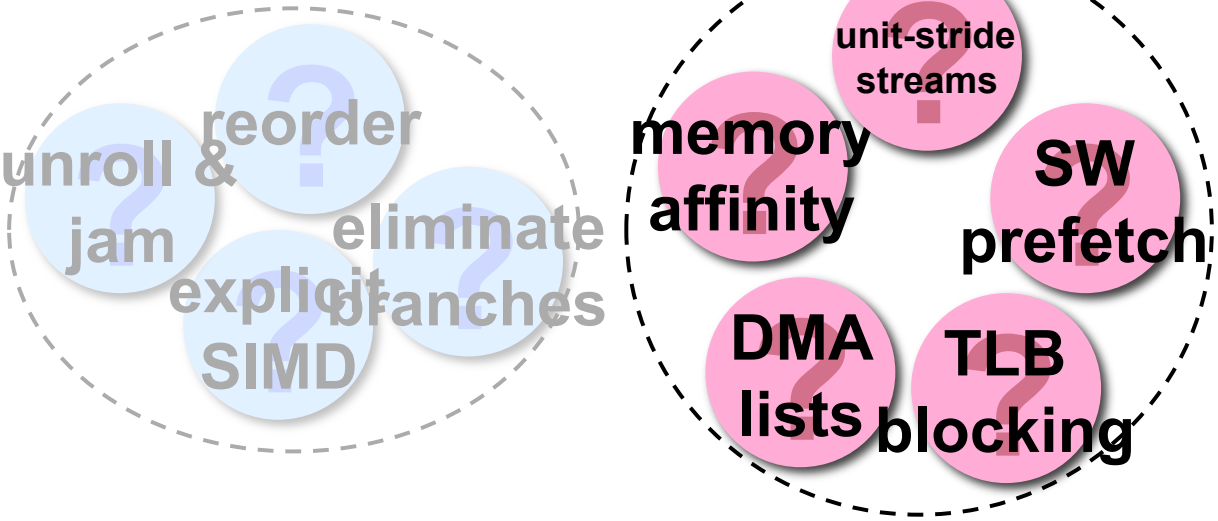
Maximizing In-core Performance | Maximizing Memory Bandwidth | Minimizing Memory Traffic

- **Exploit in-core parallelism (ILP, DLP, etc…)**
- **Good (enough) floating-point balance**



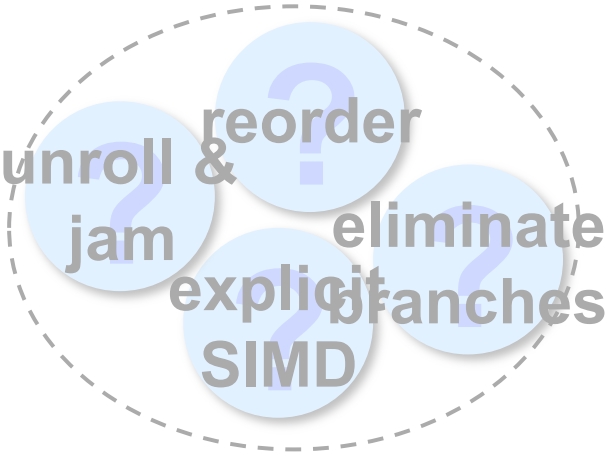
Optimization Categorization

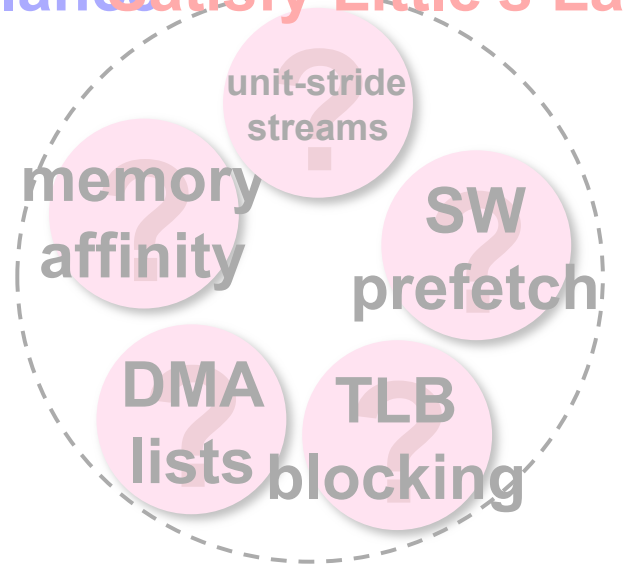
Maximizing In-core Performance	Maximizing Memory Bandwidth	Minimizing Memory Traffic
•Exploit in-core parallelism (ILP, DLP, etc…)	•Exploit NUMA	
	•Hide memory latency	
•Good (enough) floating-point balance	•Satisfy Little's Law	

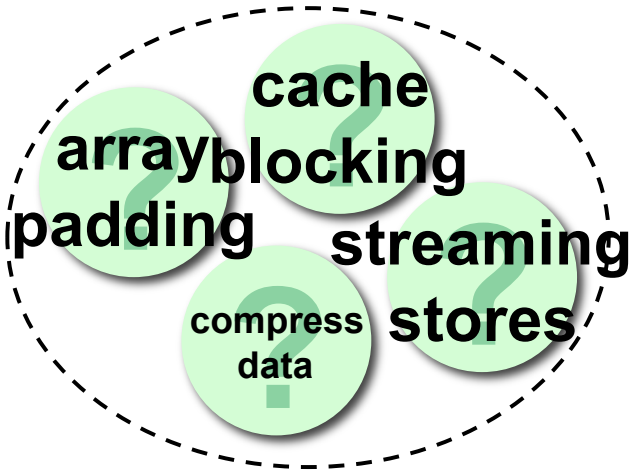


Optimization Categorization

Maximizing In-core Performance	Maximizing Memory Bandwidth	Minimizing Memory Traffic
•Exploit in-core parallelism (ILP, DLP, etc…)	•Exploit NUMA	Eliminate:
	•Hide memory latency	•Capacity misses
•Good (enough) floating-point balance	•Satisfy Little's Law	•Conflict misses
		•Compulsory misses
		•Write allocate behavi

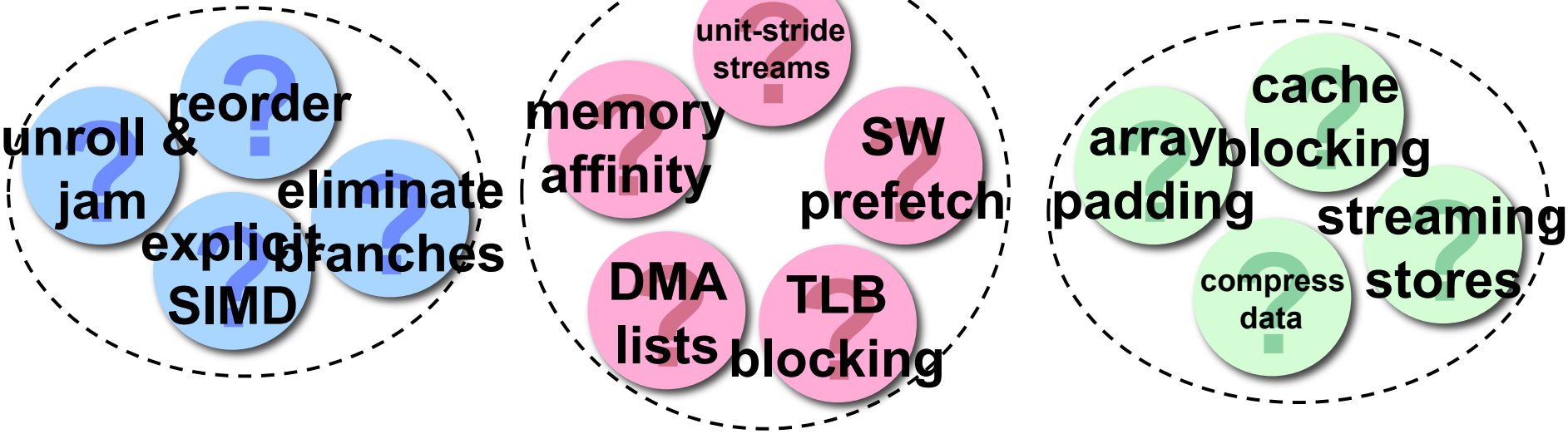






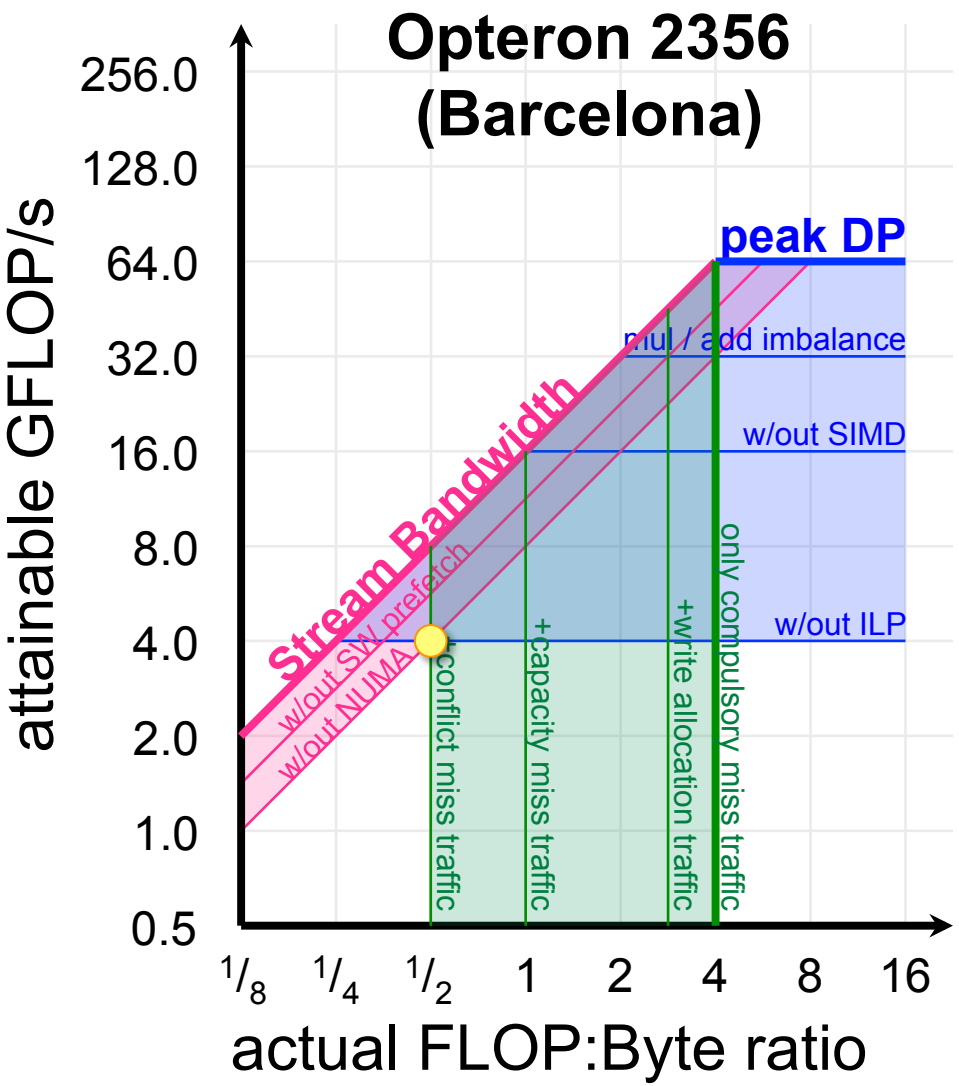
Optimization Categorization

Maximizing In-core Performance	Maximizing Memory Bandwidth	Minimizing Memory Traffic
•Exploit in-core parallelism (ILP, DLP, etc…)	•Exploit NUMA	Eliminate:
	•Hide memory latency	•Capacity misses
•Good (enough) floating-point balance	•Satisfy Little's Law	•Conflict misses
		•Compulsory misses
		•Write allocate behavi



Roofline Model

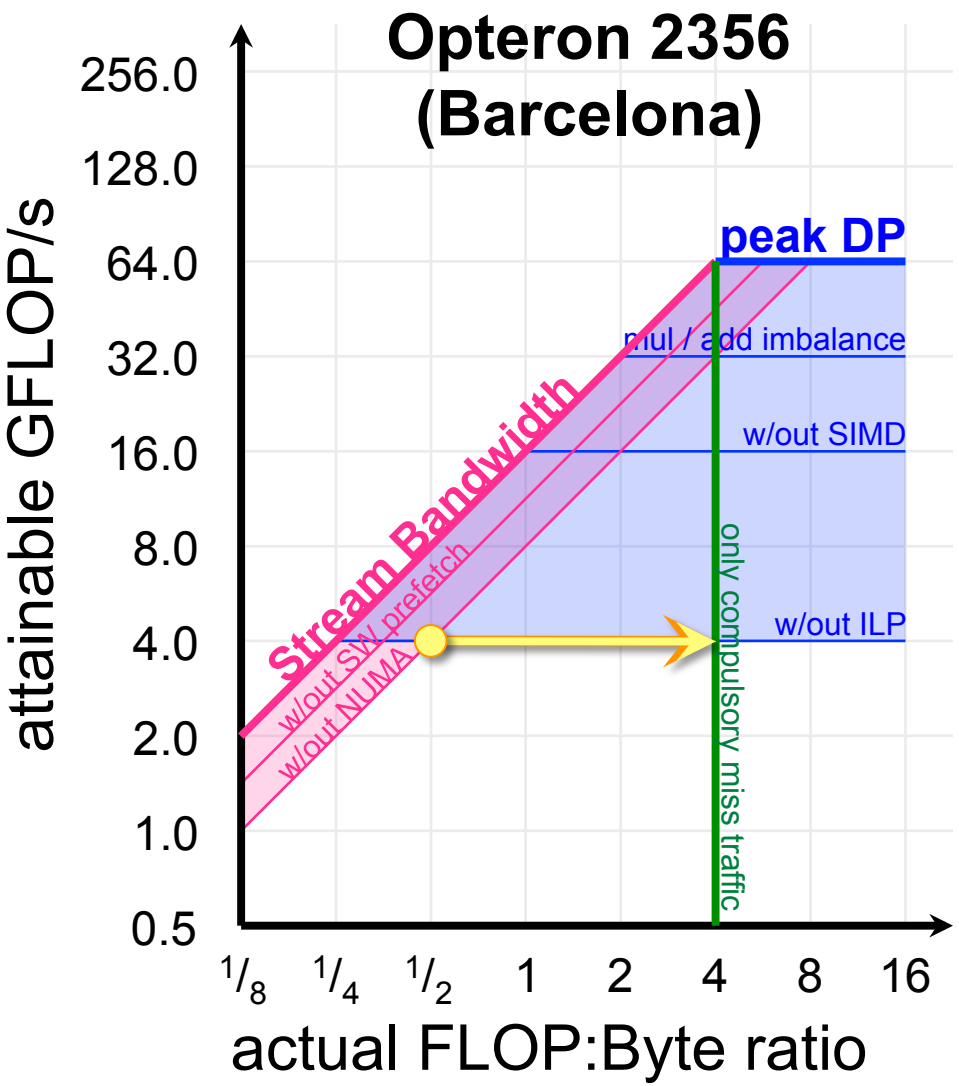
locality walls



- Optimizations remove these walls and ceilings which act to constrain performance.

Roofline Model

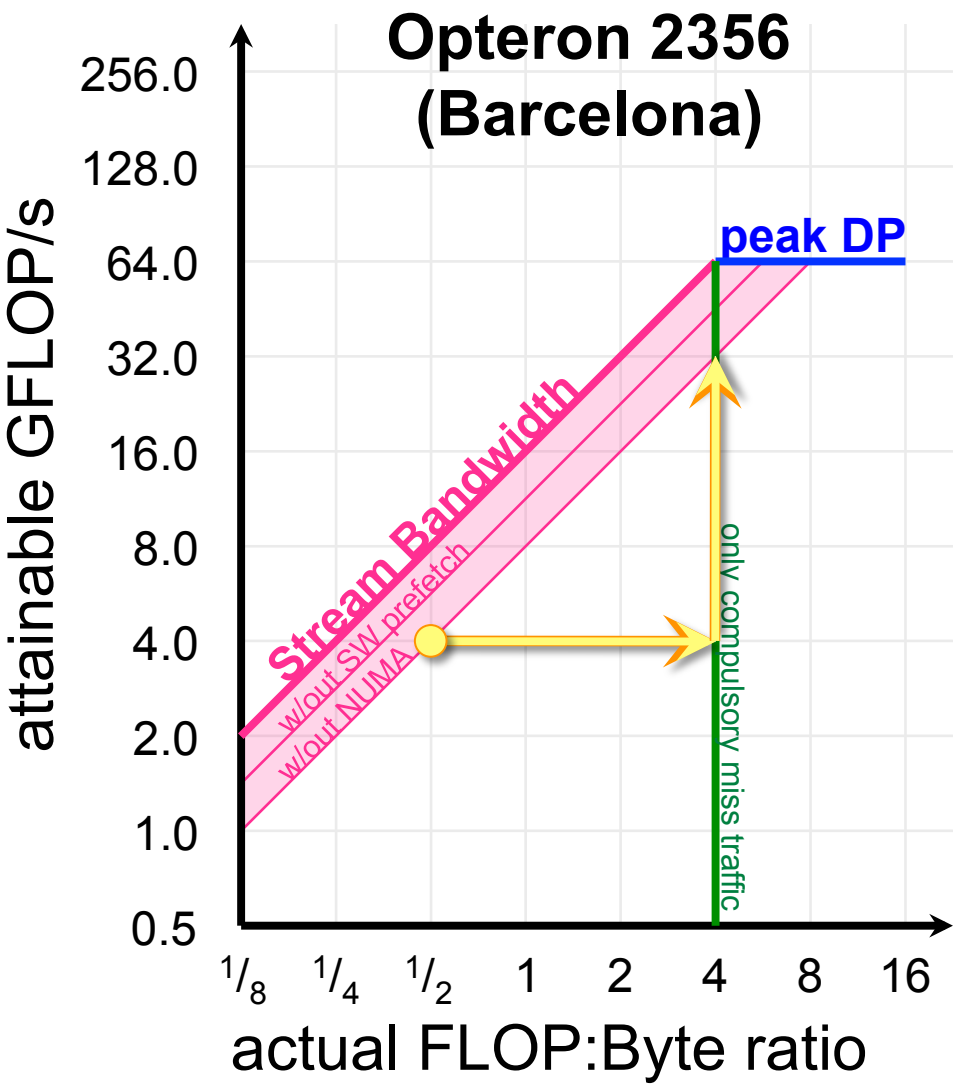
locality walls



- Optimizations remove these walls and ceilings which act to constrain performance.

Roofline Model

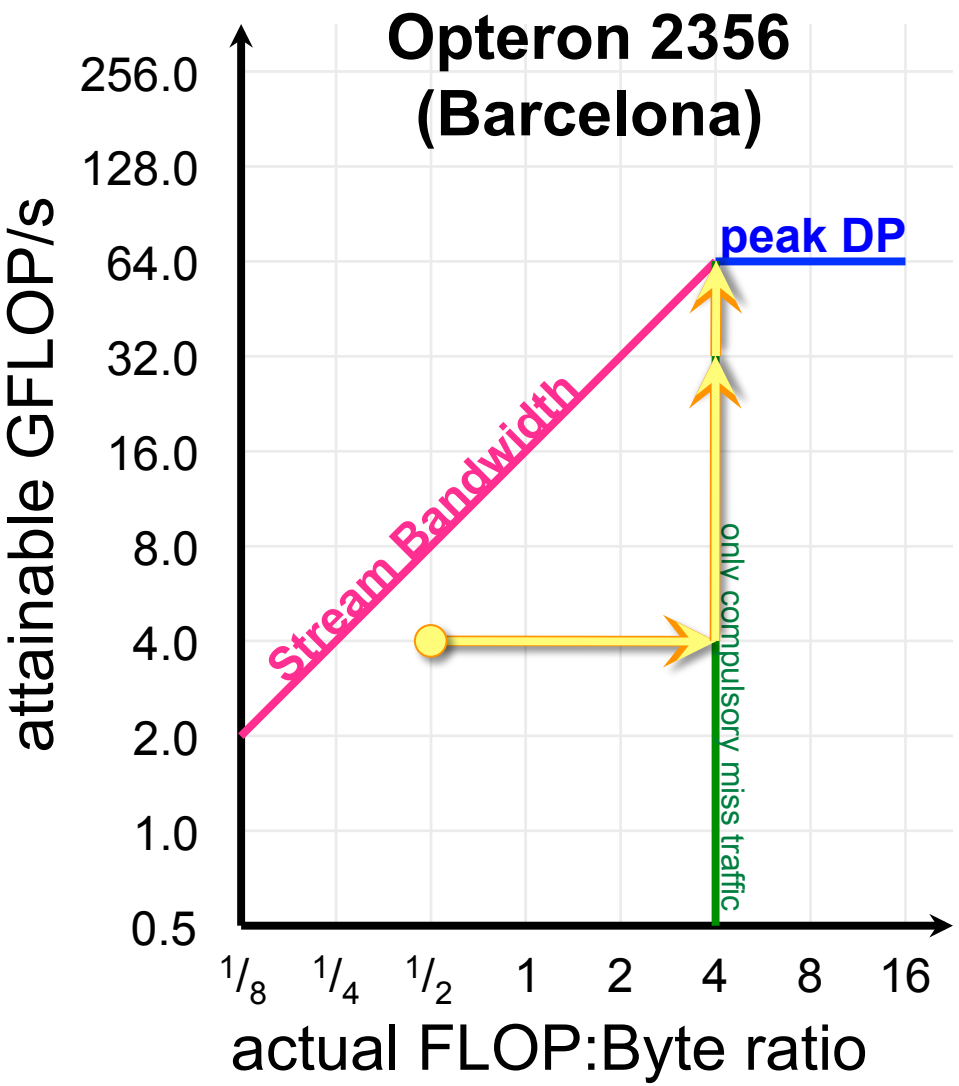
locality walls



- Optimizations remove these walls and ceilings which act to constrain performance.

Roofline Model

locality walls



- Optimizations remove these walls and ceilings which act to constrain performance.

Optimization Categorization

Maximizing In-core Performance

- Exploit in-core parallelism (ILP, DLP, etc…)
- Good (enough) floating-point balance

Maximizing Memory Bandwidth

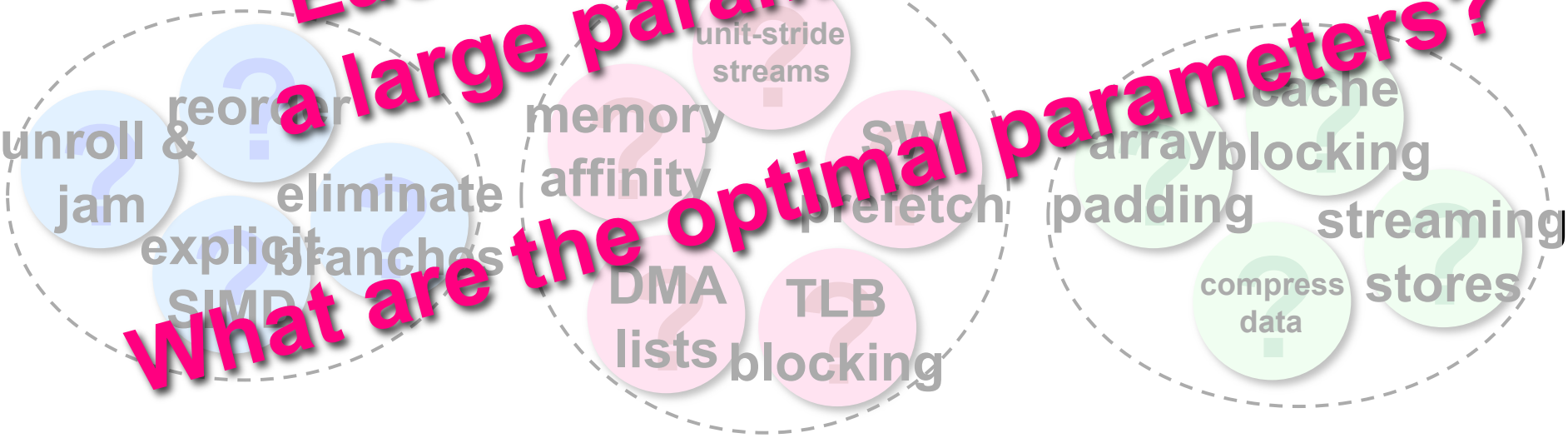
- Exploit NUMA
- Hide memory latency
- Satisfy Little's Law

Minimizing Memory Traffic

- Eliminate:
  - Capacity misses
  - Conflict misses
  - Compulsory misses
- Write allocate behavi

Each optimization has a large parameter space

What are the optimal parameters?



Auto-tuning?

- Provides **performance portability** across the existing breadth and evolution of microprocessors
- One time up front productivity cost is amortized by the number of machines its used on

- Auto-tuning does not invent new optimizations
- **Auto-tuning automates the code generation and exploration of the optimization and parameter space**
- Two components:
  - parameterized code generator (we wrote ours in Perl)
  - Auto-tuning exploration benchmark
    (combination of heuristics and exhaustive search)
- Can be extended with ISA specific optimizations (e.g. DMA, SIMD)

Multicore:
Architectures & Challenges

Options:

- Moore's law continues to double the transistors, what do we do with them ?
  - More out-of-order (prohibited by complexity, performance, power)
  - More threading (asymptotic performance)
  - More DLP/SIMD (limited applications, compilers?)
  - Bigger caches (doesn't address compulsory misses, asymptotic perf.)
  - **Place a SMP on a chip = 'multicore'**

What are SMPs? What is multicore ? What are multicore SMPs ?

- **SMP = shared memory parallel**
- In the past, it meant multiple chips (typically < 32) could address any location in a large shared memory through a network or bus

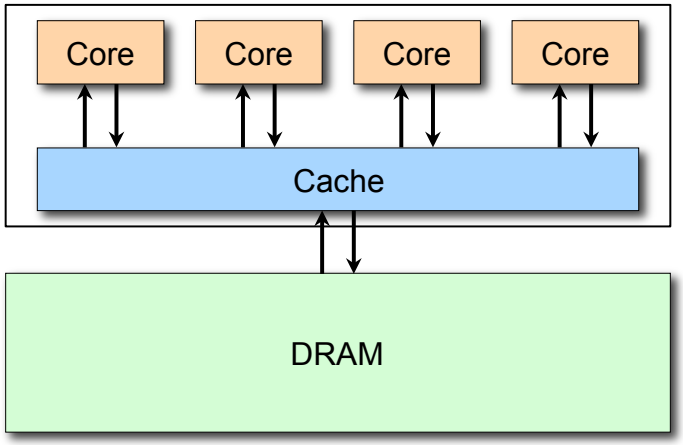
- Today, multiple cores are integrated on the same chip
- Almost universally this is done in a SMP fashion
- For “convince”, programming multicore SMPs is indistinguishable from programming multi-socket SMPs. (easy transition)

- Multiple cores can share:
  - memory controllers
  - caches
  - occasionally FPUs

- **Although there was a graceful transition from multiple sockets to multiple cores from the point of view of correctness, achieving good performance can be incredibly challenging.**

Multicore & SMP Comparison

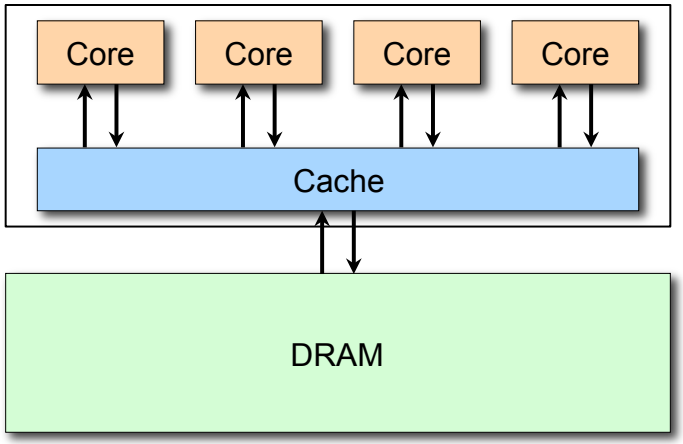
- Advances in Moore's Law allows for increased integration on-chip.
- Nevertheless, the basic architecture and programming model remained the same:
- Physically partitioned, logically shared caches and DRAM



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

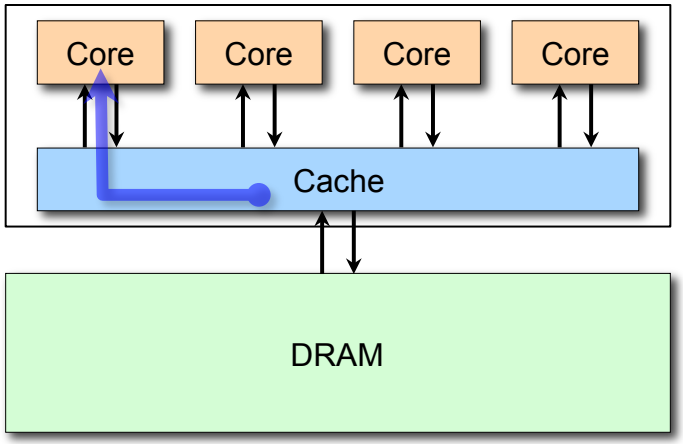
- UCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

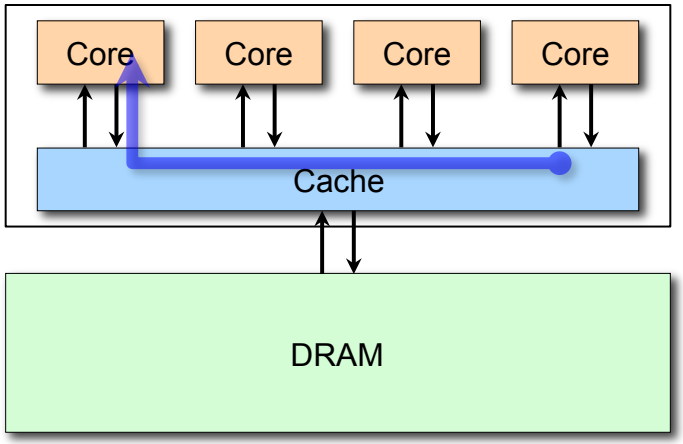
- UCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

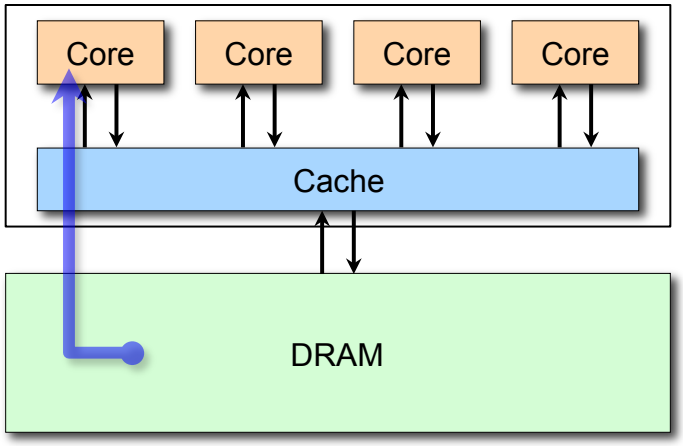
- UCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

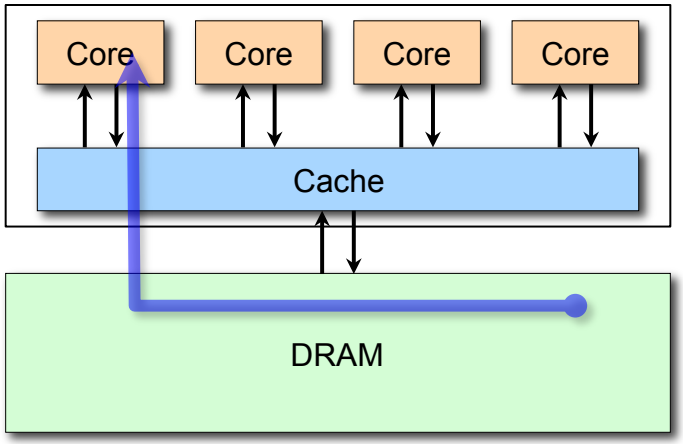
- UCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

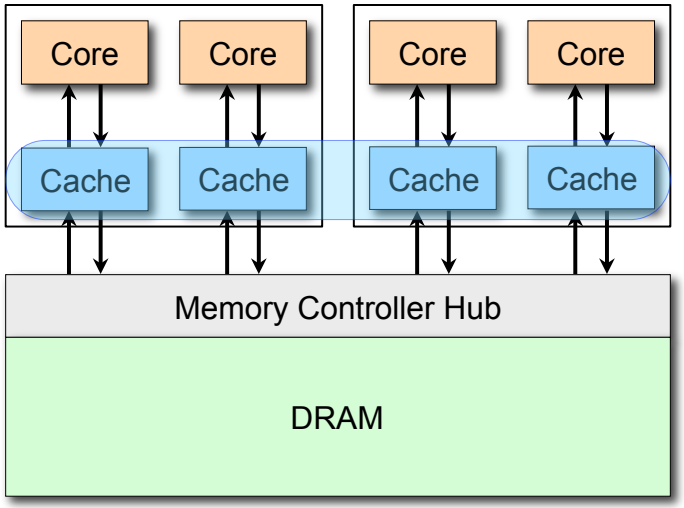
- UCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

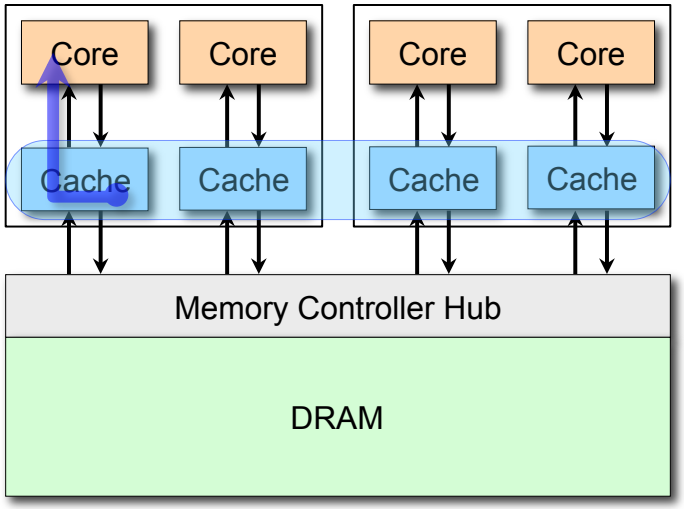
- NUCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

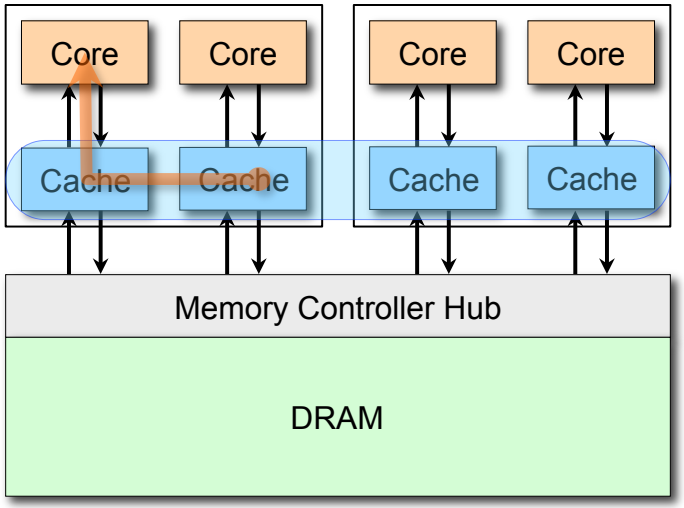
- NUCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

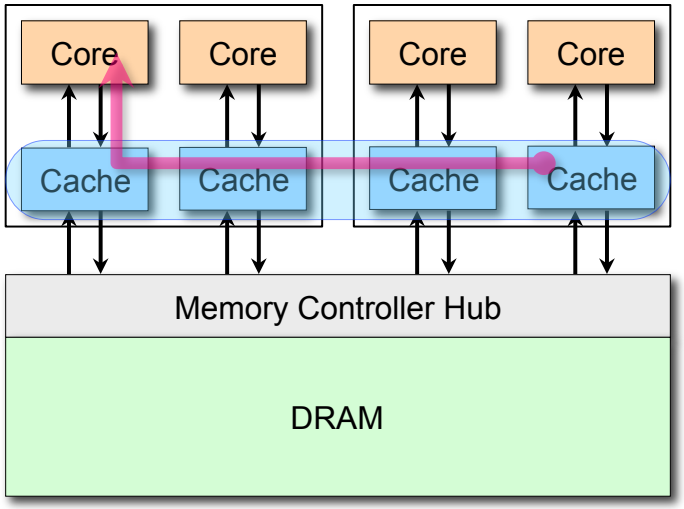
- NUCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

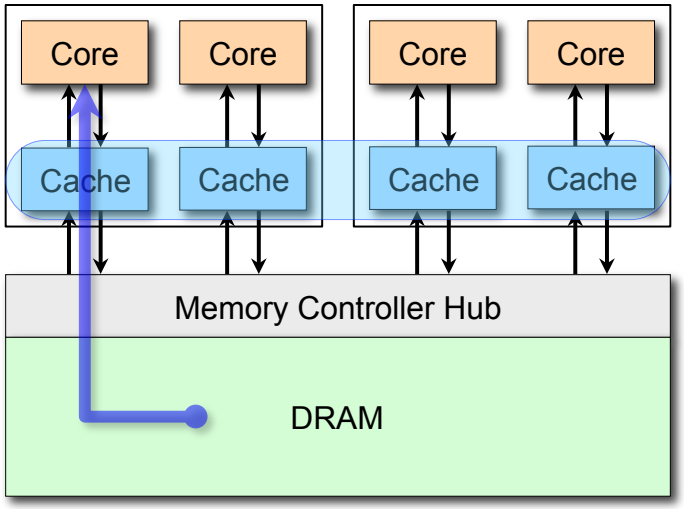
- NUCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

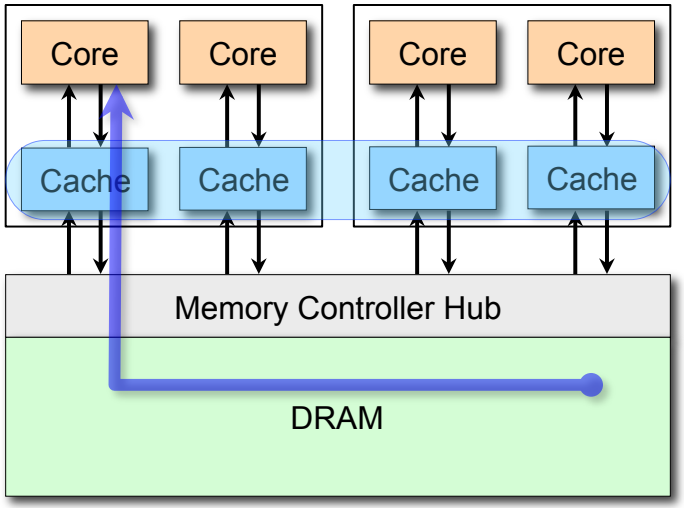
- NUCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

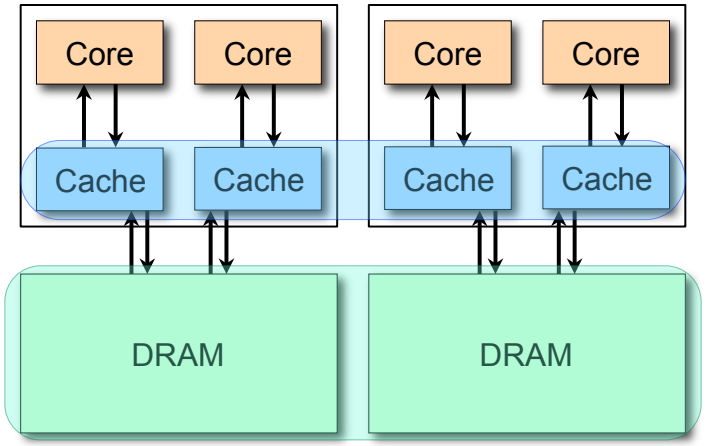
- NUCA & UMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

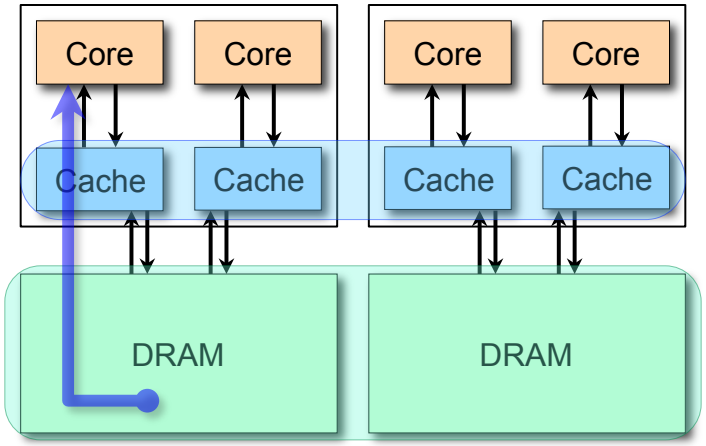
- NUCA & NUMA architecture:



NUMA vs NUCA

- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

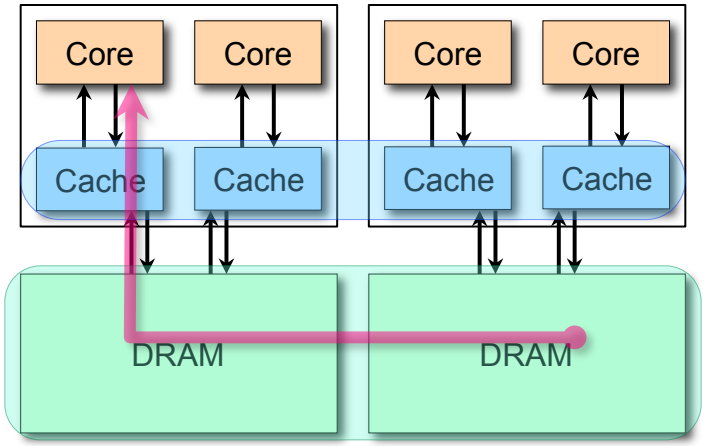
- NUCA & NUMA architecture:



NUMA vs NUCA

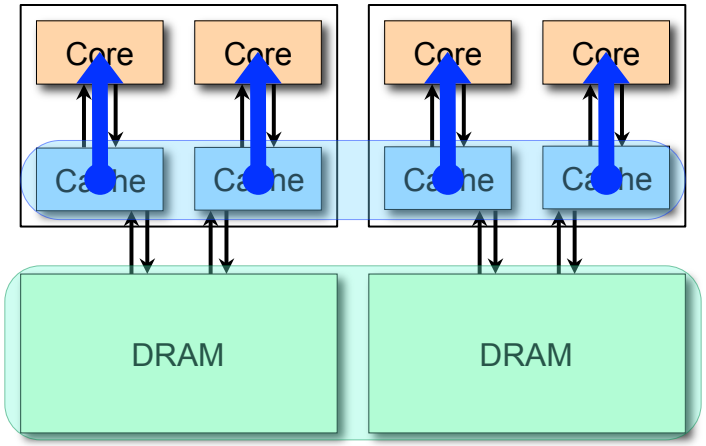
- When physically partitioned, cache or memory access is non uniform (latency and bandwidth to memory/cache addresses varies)

- NUCA & NUMA architecture:



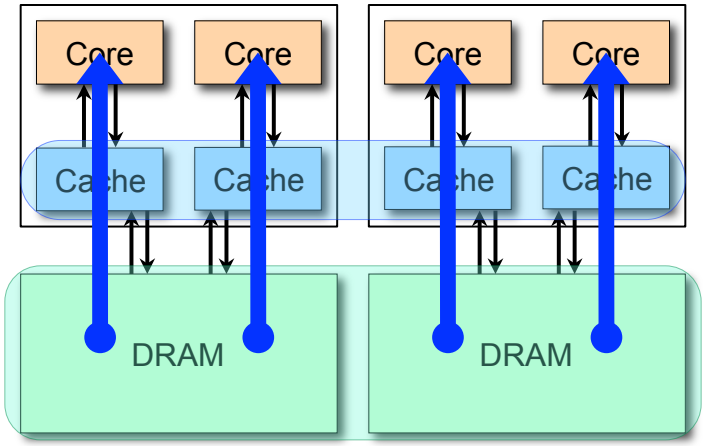
NUMA vs NUCA

- Proper cache locality



NUMA vs NUCA

- Proper DRAM locality



Multicore and Little's Law?

- Like MT, for codes with enough TLP, multicore helps in satisfying Little's Law
- Combination of multicore and multithreading also works

Concurrency per chip =

Concurrency per thread * threads per core * cores per chip =

latency * bandwidth

Best Architecture?

- Short answer: **there's not one**

- Architectures have diversified into different markets (different balance between design options)
- Architectures are constrained by a company's manpower, money, target price, volume, as well as a new **Power Wall**

- As a result, architectures are becoming simpler:
  - shallower pipelines (hard to increase frequency)
  - narrower superscalar or in-order

- But there are
  - more cores (6 minimum)
  - more threads per core (2-4)