



Automatic Performance Tuning

Samuel Williams¹, Kaushik Datta^{1,2}, Shoaib Kamil^{1,2}

Vasily Volkov^{1,2}, Cy Chan³, Archana Ganapathi², Leonid Oliker^{1,2},
John Shalf¹, Jonathan Carter¹, Katherine Yelick^{1,2}

¹Lawrence Berkeley National Laboratory

²University of California, Berkeley

³Massachusetts Institute of Technology

SWWilliams@lbl.gov

kdatta@eecs.berkeley.edu

skamil@eecs.berkeley.edu



Motivation

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

- ❖ There are many kernels/methods whose performance is key to good application performance.
- ❖ By optimizing performance we may achieve the same throughput for substantially reduced cost (time, hardware, energy)
- ❖ The fastest **code** may vary with architecture or input.
- ❖ The fastest **data structure** may vary with architecture or input.
- ❖ The fastest **algorithm** may vary with architecture or input.
- ❖ Architectures continue to rapidly evolve and diversify
(*hand optimizing for one machine solves yesterday's challenge*)
- ❖ We believe a solution that provides **performance portability** is well worth an up front productivity cost.



Automatic Performance Tuning

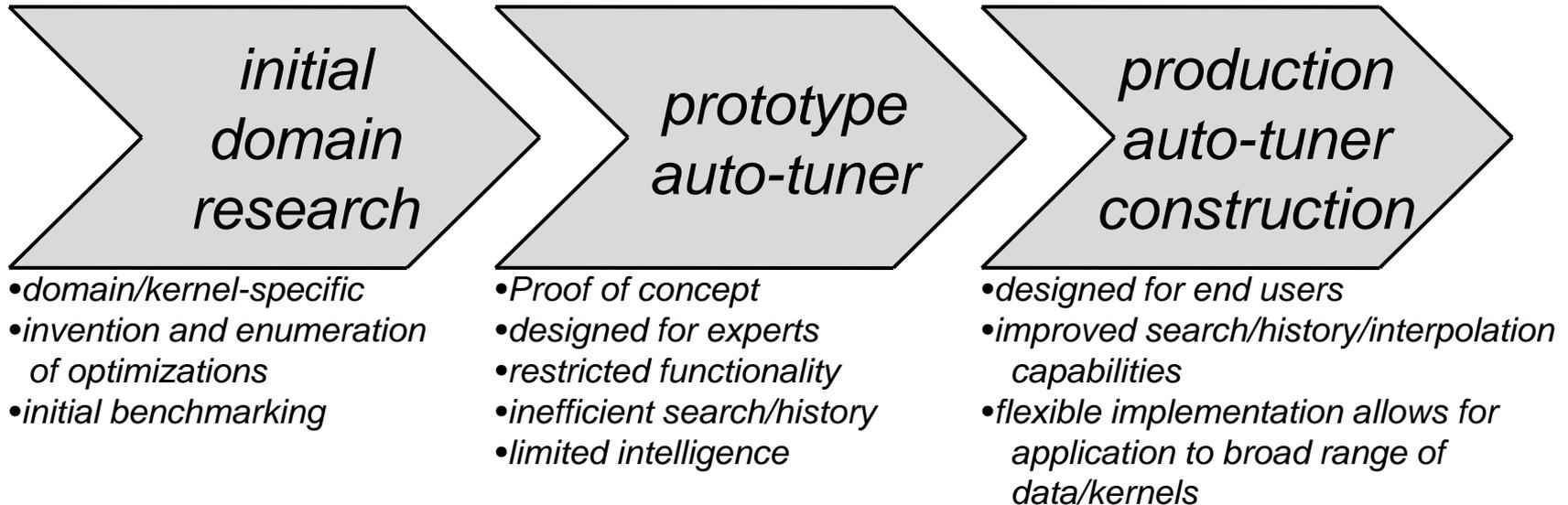
F U T U R E T E C H N O L O G I E S G R O U P

- ❖ ***auto-tuning*** is built on the premise that if one can test every possible implementation of a program, one can find the fastest.

- ❖ Key steps:
 - enumeration of an optimization space
 - generation of test cases from that space
 - exploration of those test cases (results in database/history)

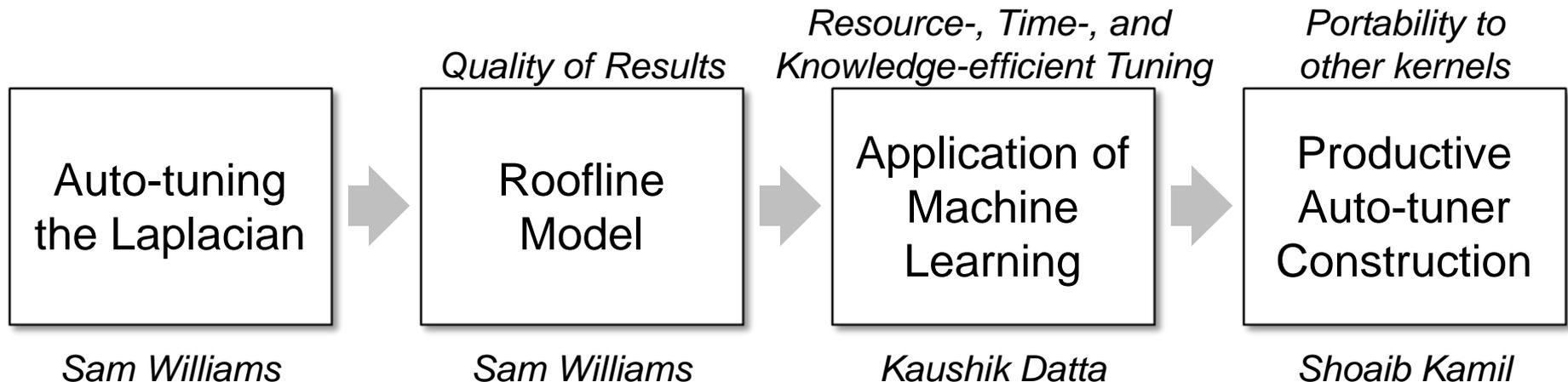
- ❖ Additional Components:
 - interpolation of results to select best implementation for runtime problem
 - assessment of the quality of results.
 - packaging the auto-tuner with an interface the end programmer can productively exploit and/or modify

- ❖ Production Quality Auto-tuners are built after substantial initial development work



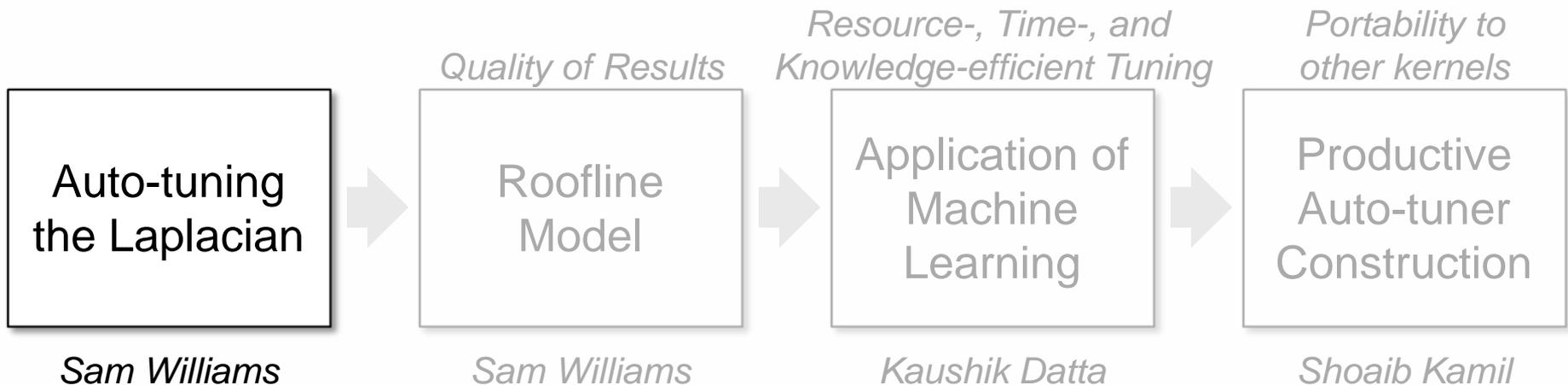
- ❖ Although this talk focuses on the initial research and construction of prototype auto-tuners for one particular domain, we've conducted research into other domains (noted later)

- ❖ This talk is composed of four mini talks (by three speakers) discussing results to date as well as future research directions
- ❖ For conciseness, we'll focus on:
 - one particular domain: finite difference methods for PDEs.
 - only cache-based computers



I

Auto-tuning the Laplacian Operator





The Laplacian Operator (Finite Difference Method)



The Heat Equation Stencil

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

- ❖ Consider Poisson's Equation (common PDE)
 - $\nabla^2 u(x,y,z) = f(x,y,z)$ ∇^2 is the Laplacian differential operator
 - Solving this equation via the finite difference method (GS, GSRB, Jacobi) devolves into performing iterative **stencil** sweeps over a 3D regular volume.
 - **stencils** are a linear combination a point's nearest neighbors.
 - many possible derivations with different finite difference representations.

- ❖ We will apply auto-tuning to improve the performance of the heat equation: $\nabla^2 u(x,y,z,t) = \frac{\partial}{\partial t} u(x,y,z,t)$
- ❖ We will restrict ourselves to
 - optimizing the per sweep performance on a 256^3 problem
 - single node performance (MPI extension is trivial)
 - Jacobi's (iterative) method (even/odd grids for even/odd time steps)
 - Employ a **computational collective** model
(all threads collective process one problem)

7-point Stencil

- ❖ Simplest derivation of the Laplacian operator results in a 7-point stencil

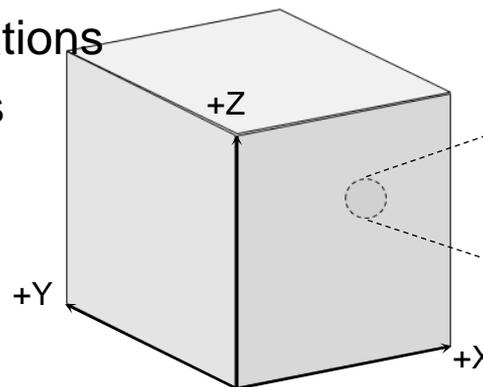
for all x, y, z :

$$u(x, y, z, t+1) = \alpha u(x, y, z, t) + \beta (u(x, y, z-1, t) + u(x, y-1, z, t) + u(x-1, y, z, t) + u(x+1, y, z, t) + u(x, y+1, z, t) + u(x, y, z+1, t))$$

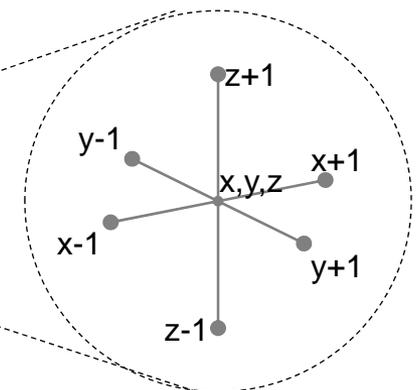
- ❖ Clearly each stencil performs:

- 8 floating-point operations
- 8 memory references

all but 2 should be filtered by an ideal cache



PDE grid



stencil for heat equation PDE



Multicore SMPs of Interest

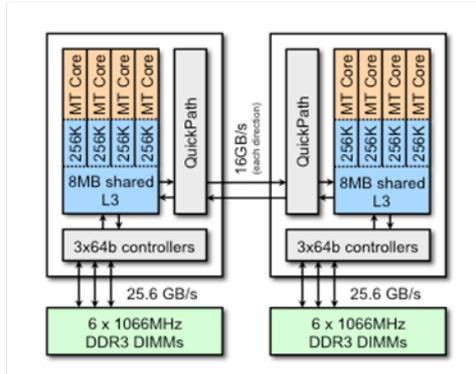
(used throughout the rest of the talk)

Multicore SMPs Used

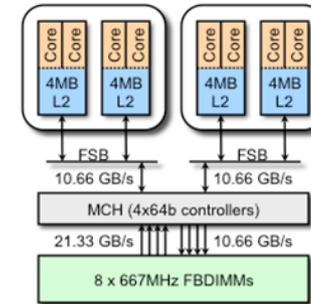
(peak performance)

FUTURE TECHNOLOGIES GROUP

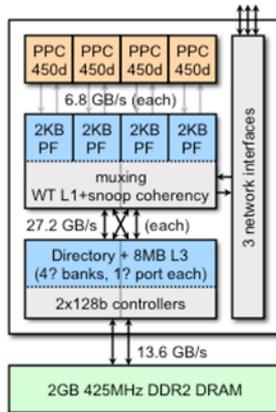
Intel Xeon X5550 (Nehalem)



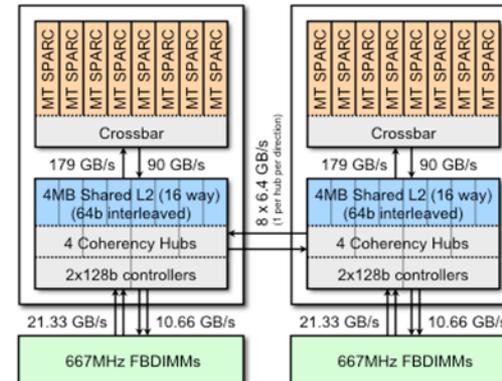
Intel Xeon X5355 (Clovertown)



Blue Gene/P



Sun T2+ T5140 (Victoria Falls)

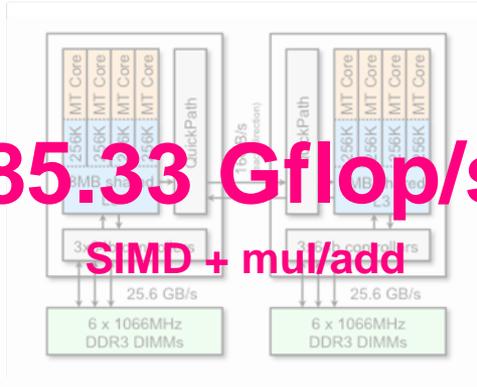


Multicore SMPs Used

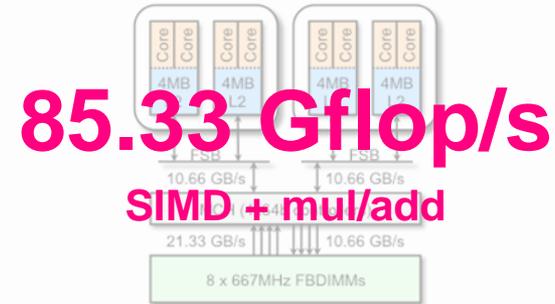
(peak performance)

FUTURE TECHNOLOGIES GROUP

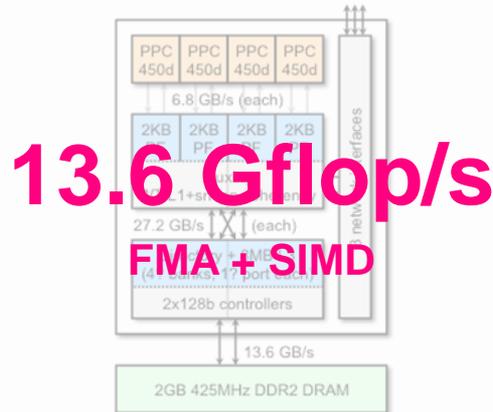
Intel Xeon X5550 (Nehalem)



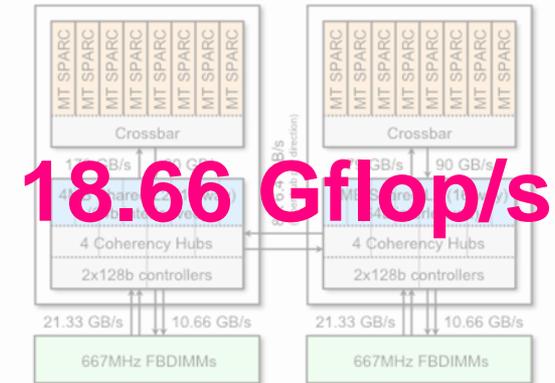
Intel Xeon X5355 (Clovertown)



Blue Gene/P



Sun T2+ T5140 (Victoria Falls)

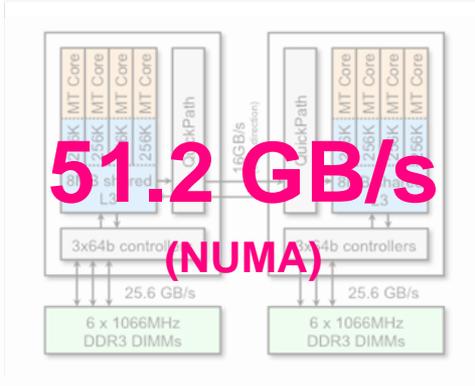


Multicore SMPs Used

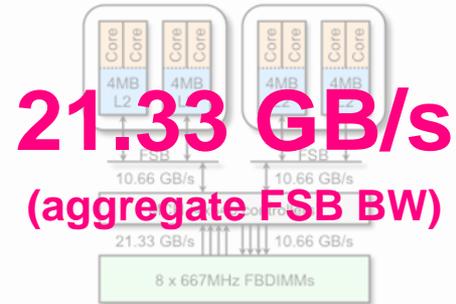
(pin bandwidth)

FUTURE TECHNOLOGIES GROUP

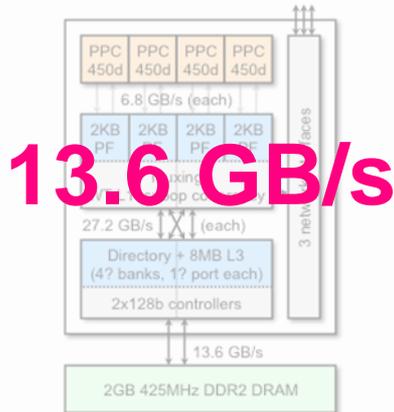
Intel Xeon X5550 (Nehalem)



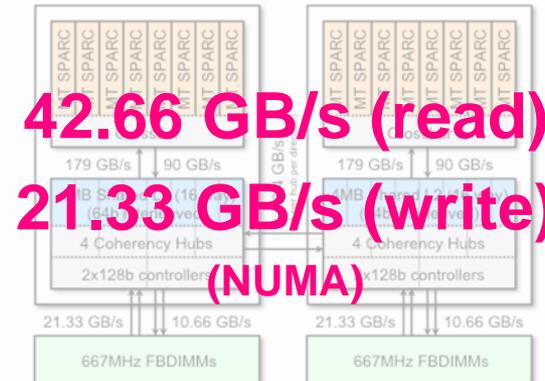
Intel Xeon X5355 (Clovertown)



Blue Gene/P



Sun T2+ T5140 (Victoria Falls)



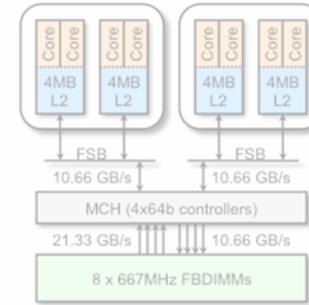
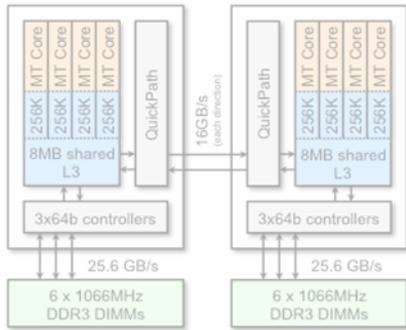
Multicore SMPs Used

(in-order cores = sw/compiler must pick up the slack)

FUTURE TECHNOLOGIES GROUP

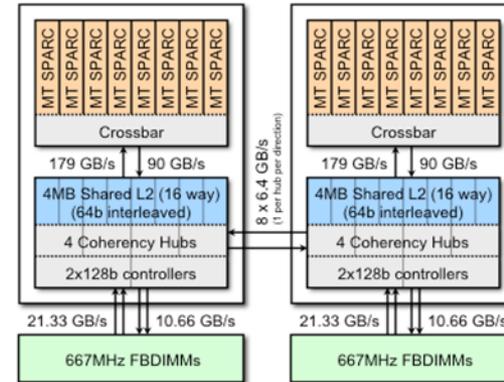
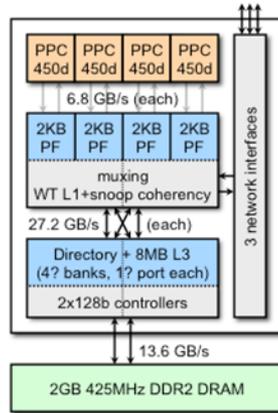
Intel Xeon X5550 (Nehalem)

Intel Xeon X5355 (Clovertown)



Blue Gene/P

Sun T2+ T5140 (Victoria Falls)

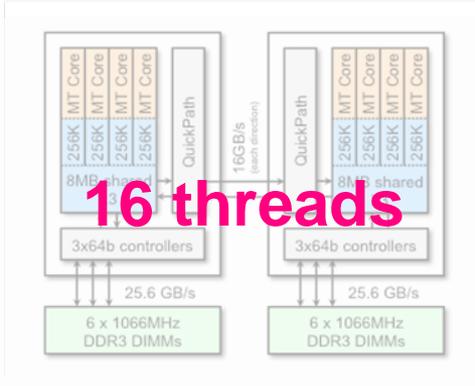


Multicore SMPs Used

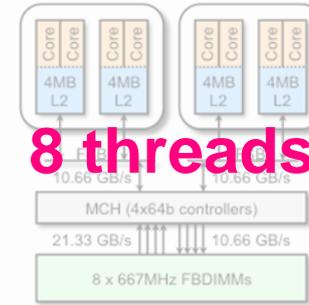
(thread-level parallelism)

FUTURE TECHNOLOGIES GROUP

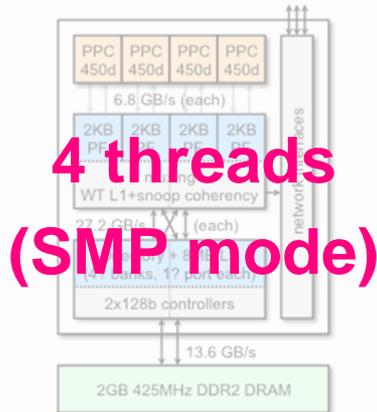
Intel Xeon X5550 (Nehalem)



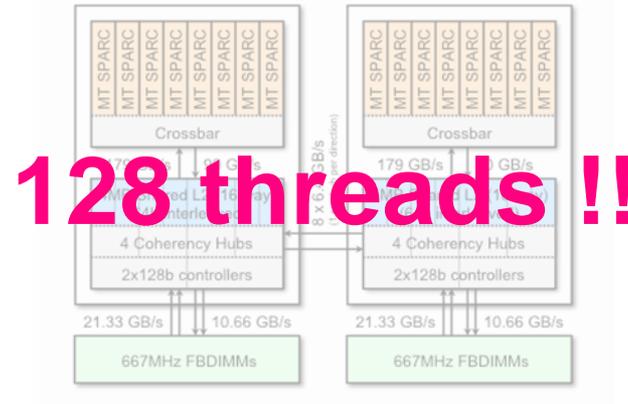
Intel Xeon E5345 (Clovertown)



Blue Gene/P



Sun T2+ T5140 (Victoria Falls)





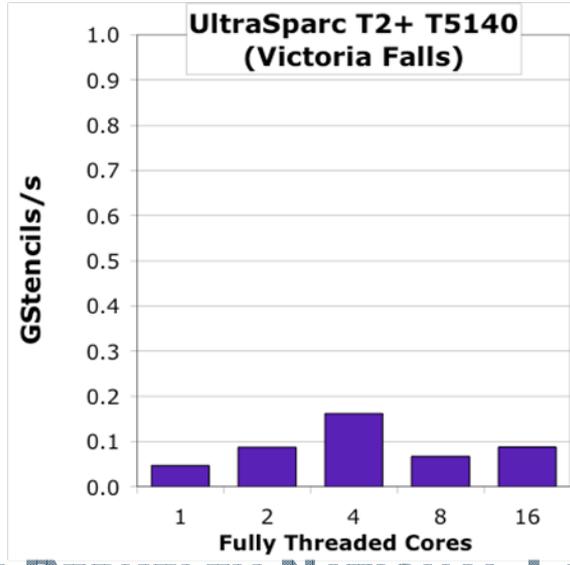
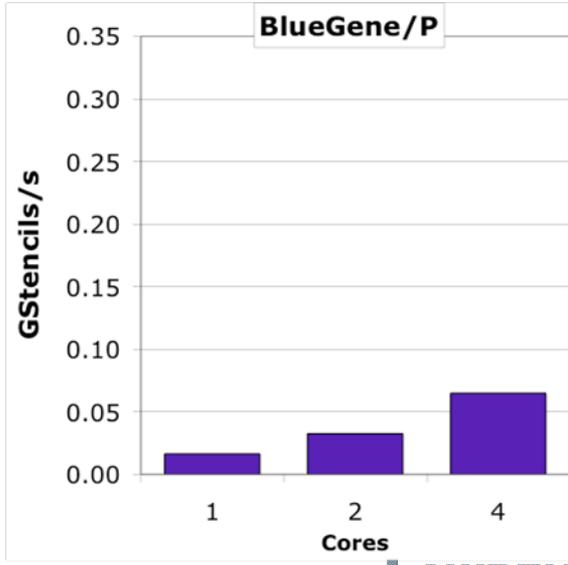
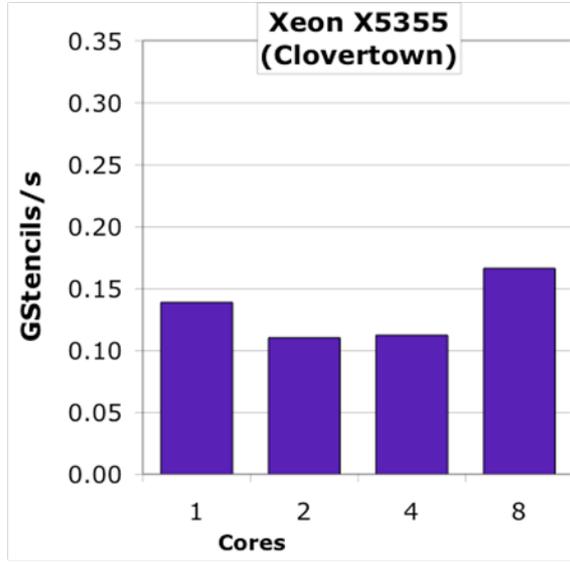
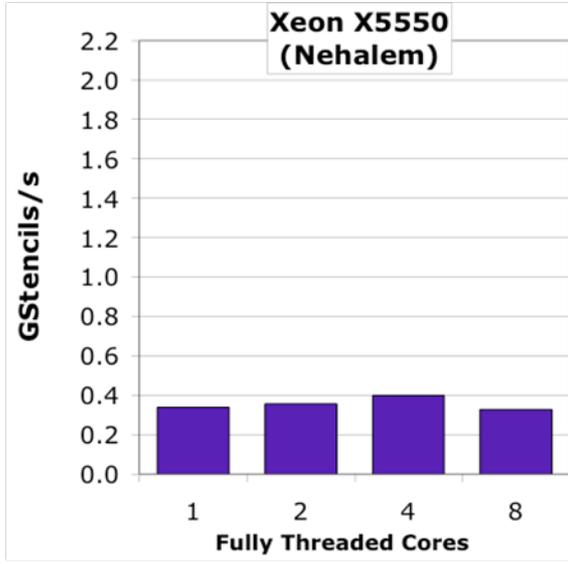
Auto-tuning the Heat Equation



Stencil Performance

(reference code)

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



- ❖ NOTE (for 7-point):
1 GStencil/s = 8 Gflop/s
- ❖ No scalability
- ❖ Poor performance
- ❖ VF performance peaks using 1/2 of one socket

 Reference Implementation



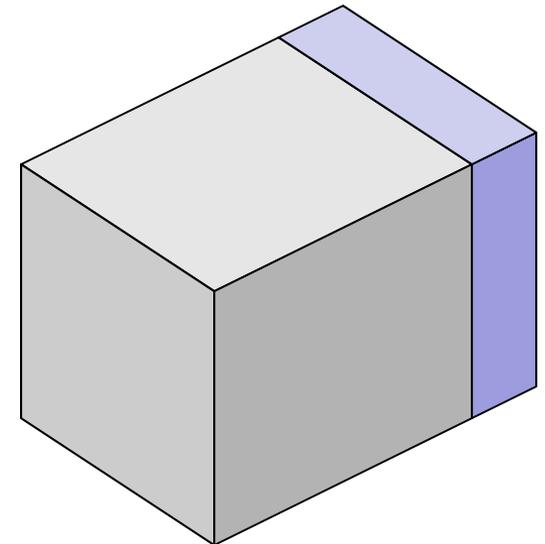
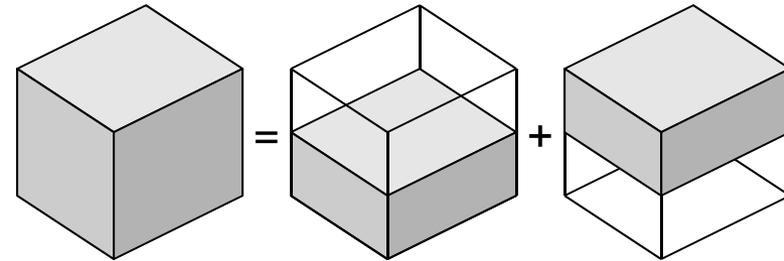
Greedy Search

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

- ❖ Resulting Optimization space will be so large that exhaustive search (even knowing the problem size) is intractable.
- ❖ Greedy search orders the optimizations (with expert architecture knowledge) and then searches them sequentially.
- ❖ Essentially transforms complexity from an $O(N^D)$ into a $O(N \times D)$ problem
- ❖ Fast, but not guaranteed to be optimal

- ❖ NUMA SMPs are common.
- ❖ It is essential the auto-tuner control data allocation to ensure maximum bandwidth
- ❖ one malloc() with parallel initialization (exploit first touch)

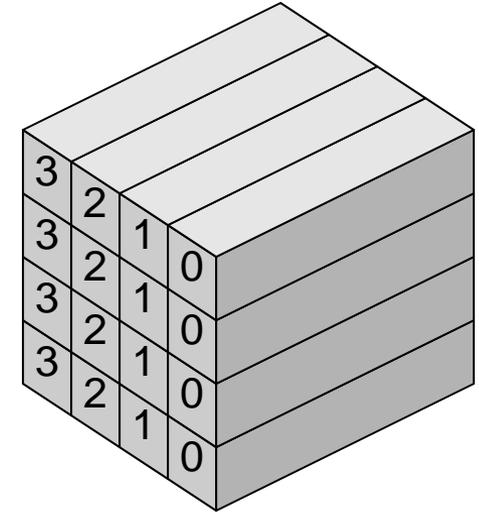
- ❖ Caches have limited associativity
- ❖ Also requires auto-tuner to control data structure allocation
- ❖ **avoids cache conflict misses**
- ❖ Explore paddings up to 32 in unit stride



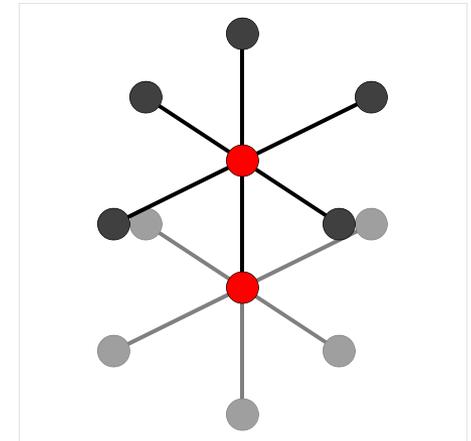
Code Transformations (1)

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

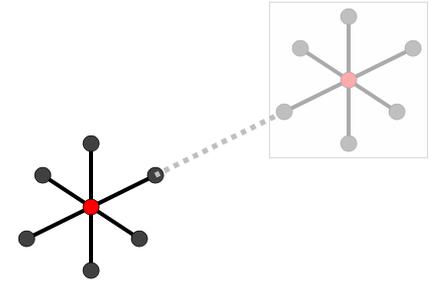
- ❖ Naïve parallelization gives each thread Z/P
- ❖ Cache blocking gives each thread a number of consecutive $(CX \times CY \times CZ)$ blocks.
- ❖ **avoids LLC capacity misses**
- ❖ Explore all possible power of 2 blockings



- ❖ Register Blocking attempts to create locality in the register file/L1 cache
- ❖ Many possible blockings $(RX \times RY \times RZ)$
- ❖ **avoids RF/L1 capacity misses**
- ❖ Explore all possible power of 2 blockings



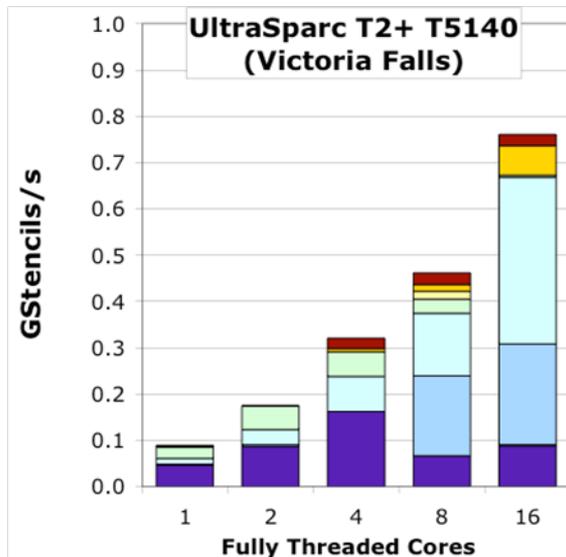
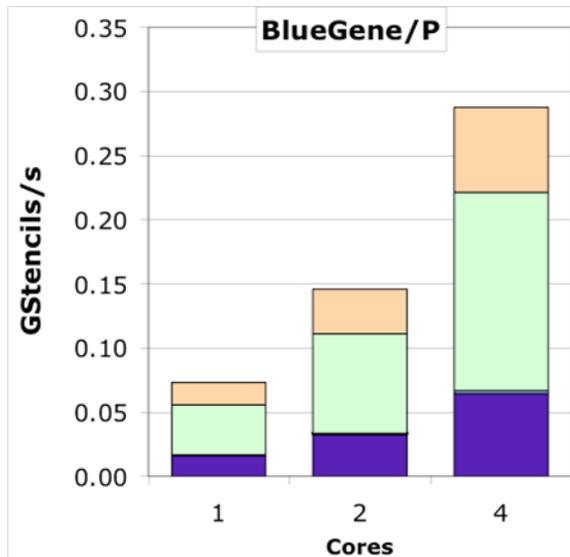
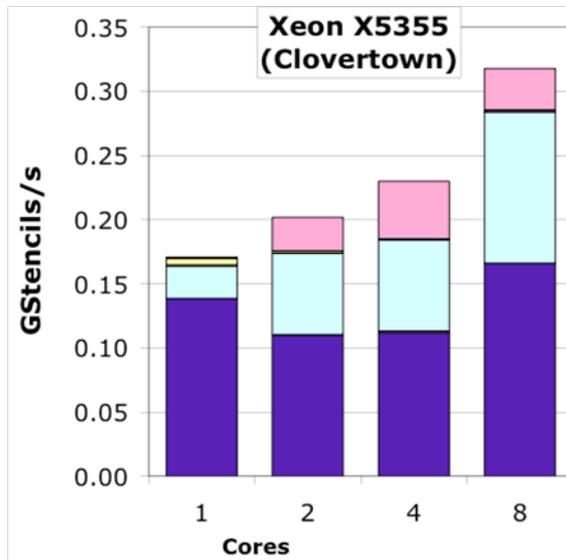
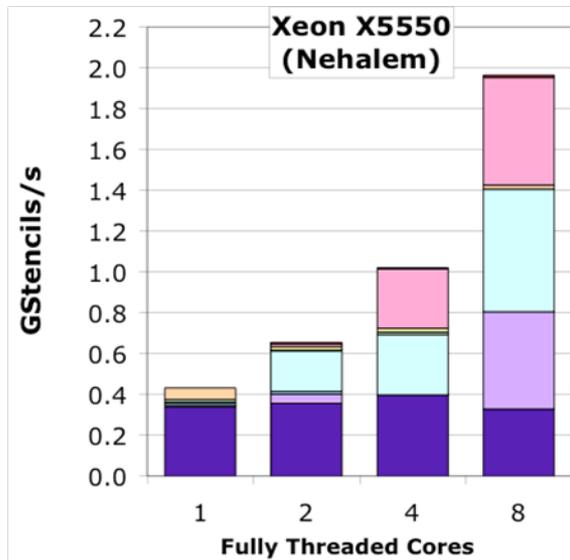
- ❖ Prefetching initiates now the loads of points that will eventually be needed
- ❖ Explore a number of prefetch distances
- ❖ **hides memory latency**
- ❖ Thread blocking adds a second level of parallelization for SMT architectures.
- ❖ **inter-thread locality via shared L1's**
- ❖ Explicit SIMDization attempts to compensate for a compiler's inability to generate good SIMD code.
- ❖ Cache bypass exploits instructions designed to eliminate write allocate traffic



Auto-tuned Stencil Performance

(full tuning, greedy search)

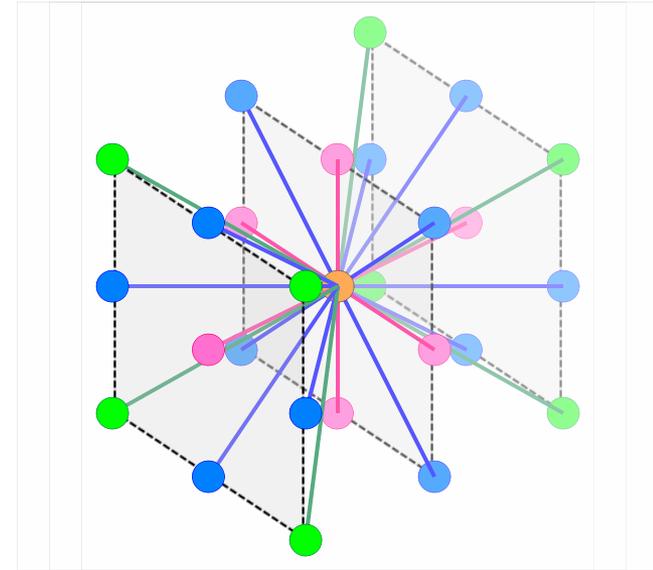
FUTURE TECHNOLOGIES GROUP



- ❖ Dramatically better performance and scalability
- ❖ Clearly cache blocking (capacity misses) were critical to performance despite these machines having 8-16MB of cache
- ❖ XLC utterly failed to register block (unroll&jam)
- ❖ Array padding was critical on VF (conflict misses)

- +2nd pass thru greedy search
- +Cache bypass
- +Explicit SIMDization
- +Thread Blocking
- +SW Prefetching
- +Register Blocking
- +Cache Blocking
- +Padding
- +NUMA
- Reference Implementation

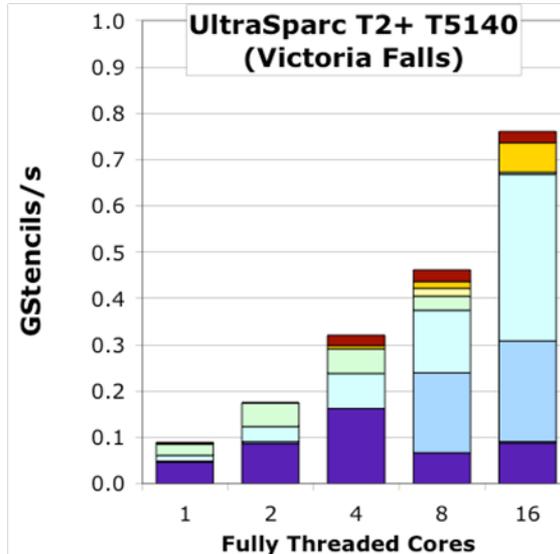
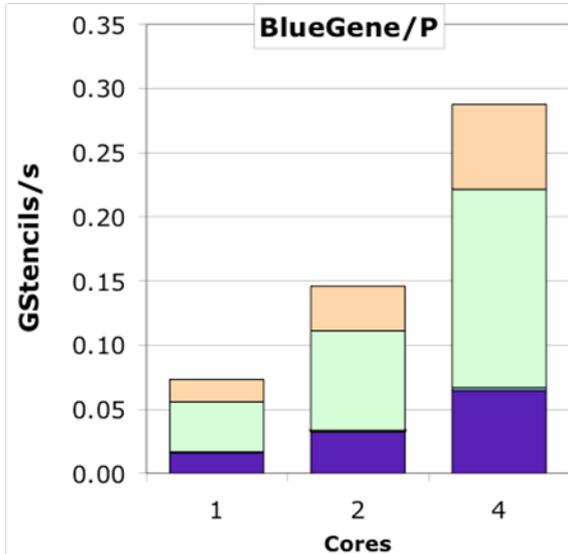
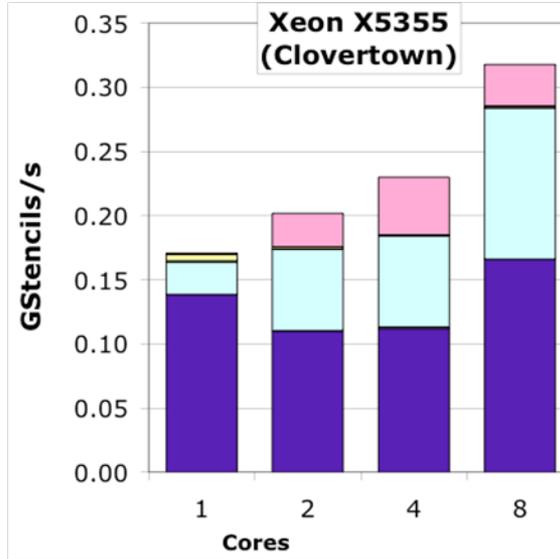
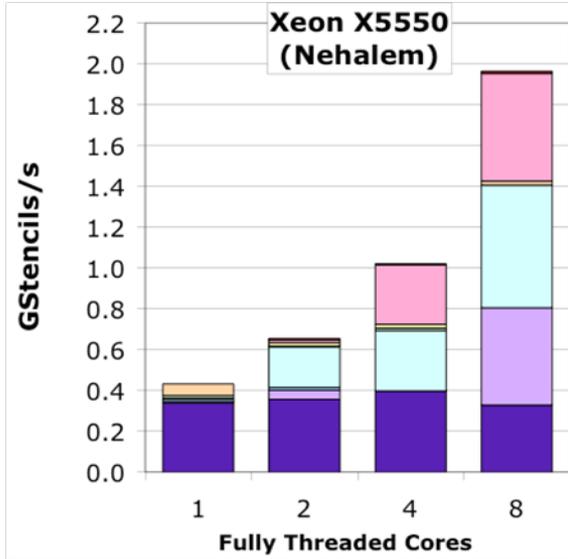
- ❖ Explore alternate finite difference method derivations.
- ❖ Subtly performance (stencils/s) can go down (more flops per stencil) but both performance (gflop/s) and time to solution can improve (fewer sweeps to converge)
- ❖ Two approaches:
 - 27-point stencil (~30 flops per stencil)
 - 27-point stencil with inter-stencil common subexpression elimination (~20+ flops/stencil)



Auto-tuned 7-point Stencil

(full tuning, greedy search)

FUTURE TECHNOLOGIES GROUP



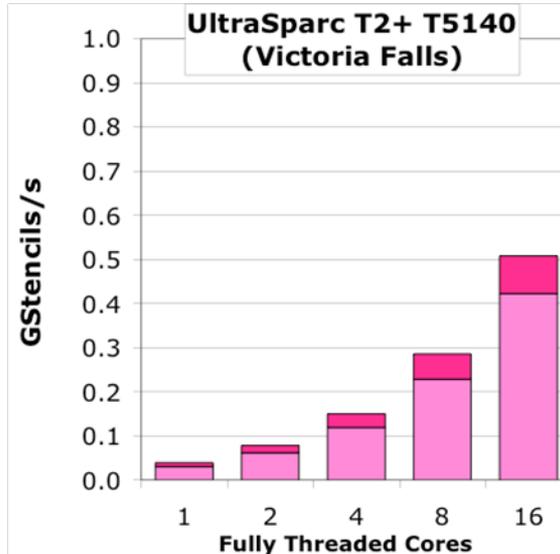
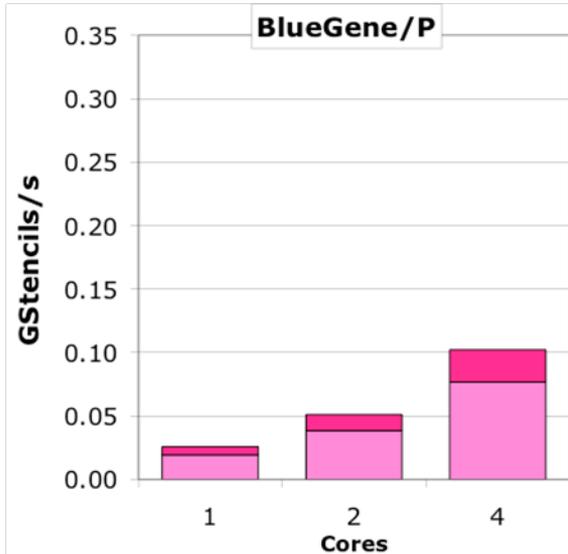
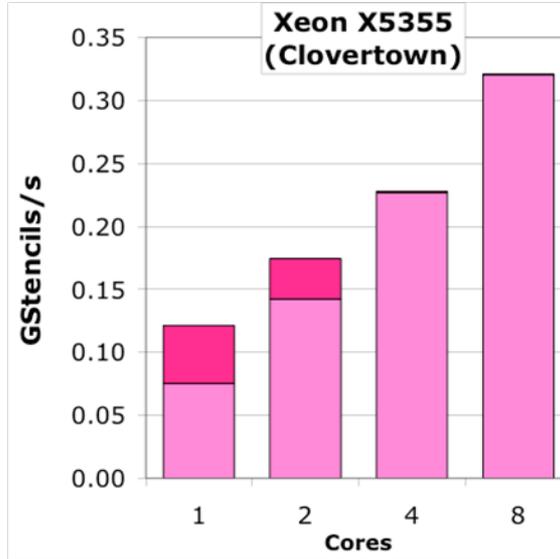
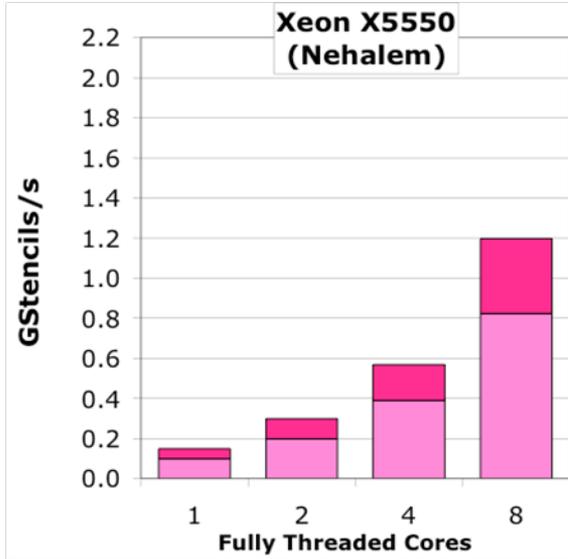
- ❖ When embedded in an iterative solver, 27-point should require fewer iterations to converge
- ❖ However, performance (stencils/second) is lower, so one must tune for the right balance of performance per iteration and number of iterations
- ❖ Observe Clovertown performance didn't change (bandwidth is so poor it's the bottleneck in either case)

 +Common Subexpression Elimination
 +Cache bypass

Auto-tuned 27-point Stencil

(full tuning, greedy search)

FUTURE TECHNOLOGIES GROUP



- ❖ When embedded in an iterative solver, 27-point should require fewer iterations to converge
- ❖ However, performance (stencils/second) is lower, so one must tune for the right balance of performance per iteration and number of iterations
- ❖ Observe Clovertown performance didn't change (bandwidth is so poor it's the bottleneck in either case)

 +Common Subexpression Elimination
 +Cache bypass



Summary of Stencil Auto-tuning Efforts

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

- ❖ We've shown that auto-tuning can provide **performance portability** across a wide range of cache-based processor architectures
 - **6x** on the most advanced architecture (Nehalem)
 - **8x** on the most parallel architecture (Victoria Falls)

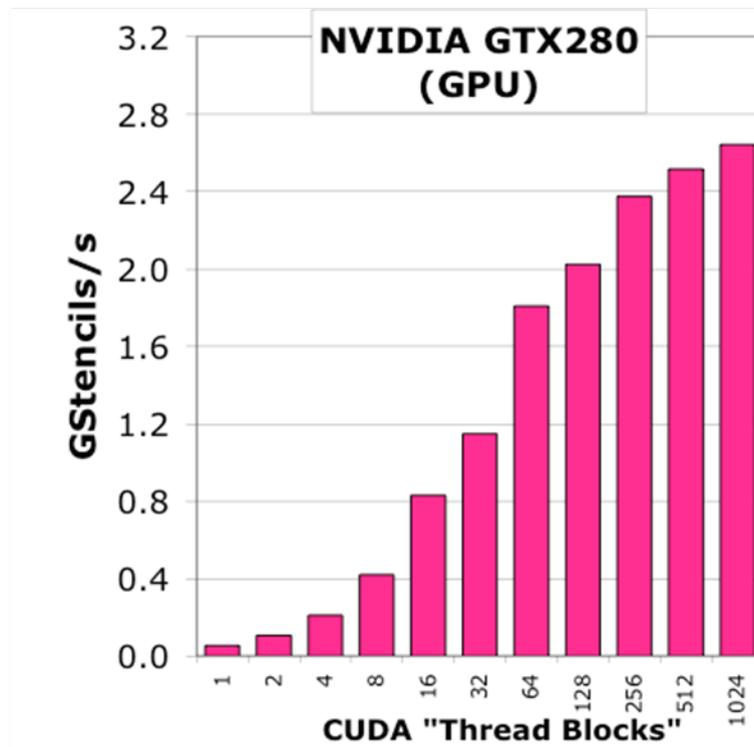
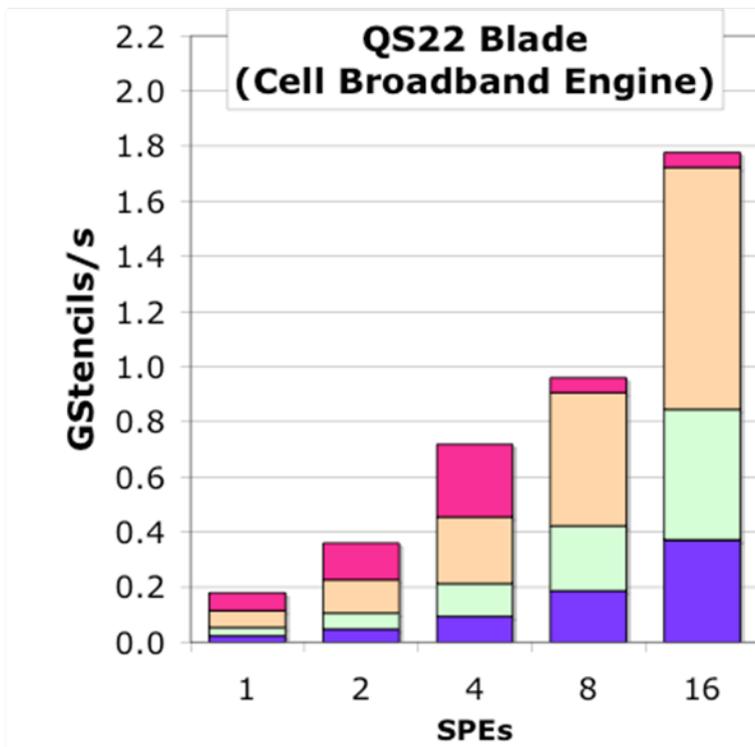
- ❖ Clearly, optimizations that control data structure, modify code, and change algorithms (different stencils) are required to attain the best performance.

- ❖ This work was submitted as:
 - *K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, K. Yelick, "Auto-tuning the 27-point Stencil for Multicore", The Fourth International Workshop on Automatic Performance Tuning (iWAPT 2009), Tokyo, Japan, October 1-2, 2009.*
 - *K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, K. Yelick, "Auto-tuning Stencil Computations on Diverse Multicore Architectures", Parallel and Distributed Computing, ISBN 978-3-902613-45-5, IN-TECH Publishers.*



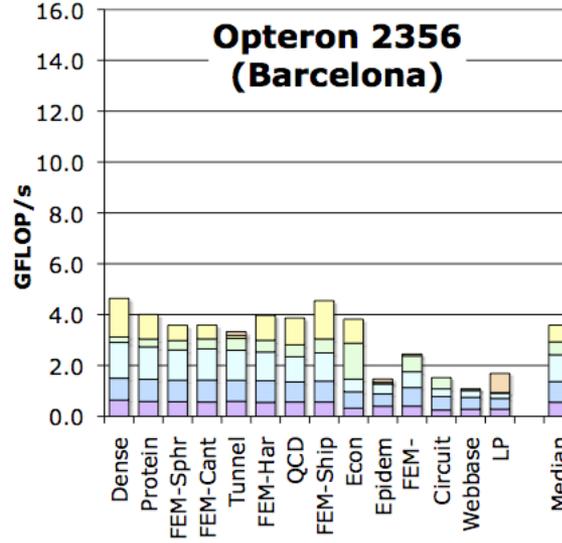
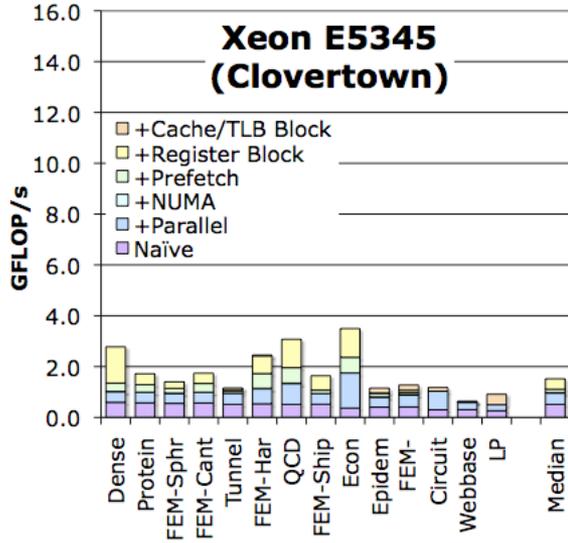
Additional Auto-tuning Work

- ❖ All the stencil work was also implemented via an auto-tuner on a QS22 CBE, and a direct optimized implementation on a GTX280.
- ❖ Clearly, the new (DDR2) Cell is bandwidth-starved

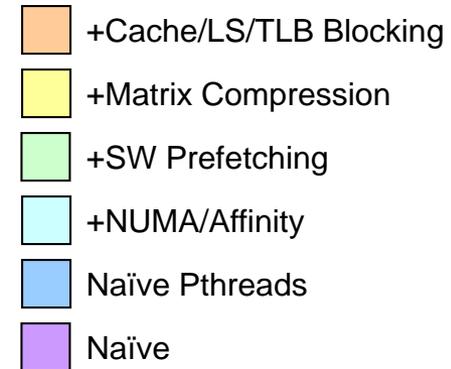
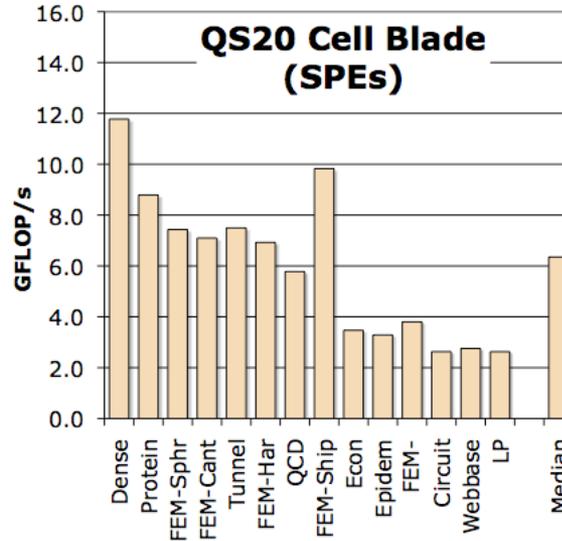
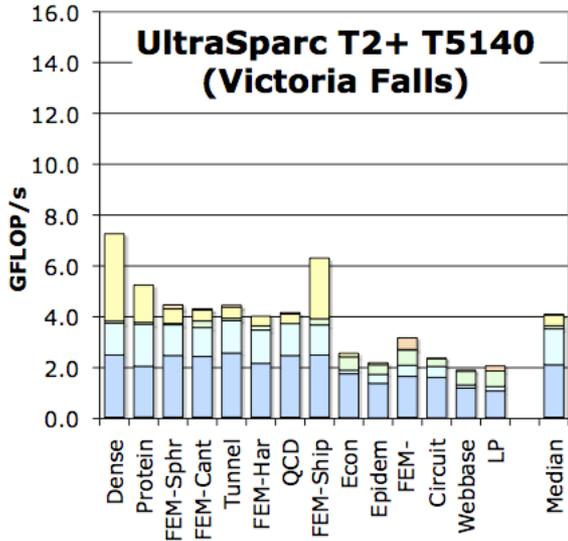


Auto-tuner for SpMV

FUTURE TECHNOLOGIES GROUP

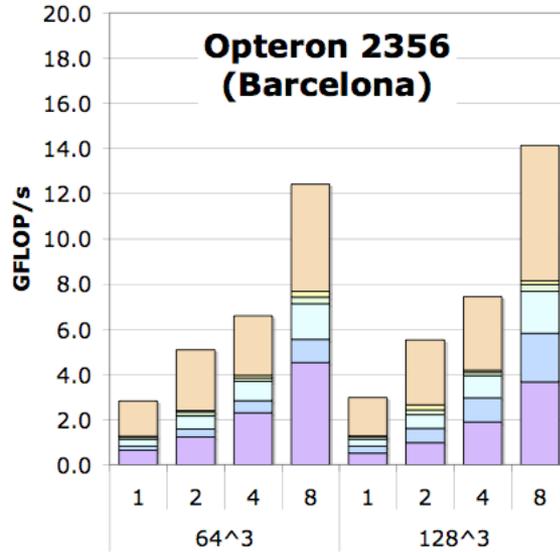
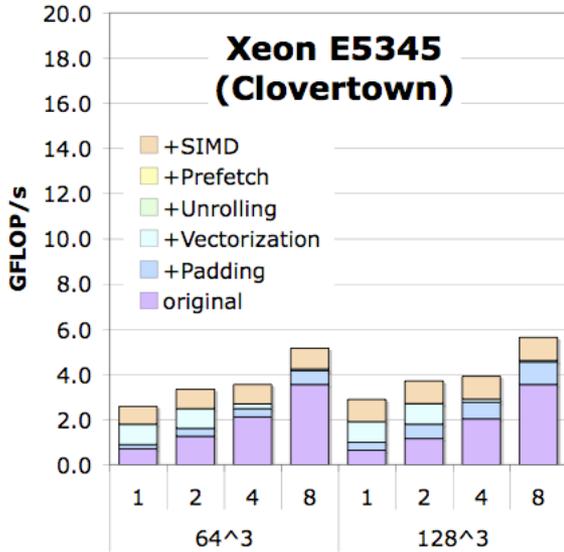


❖ We've also implemented a sparse matrix-vector multiply auto-tuner on multicore and Cell.

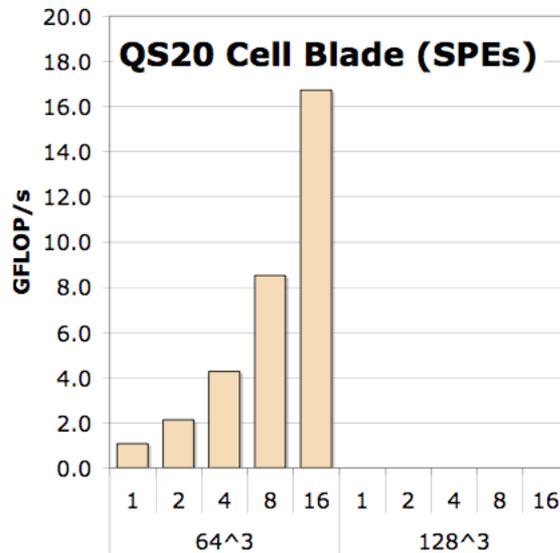
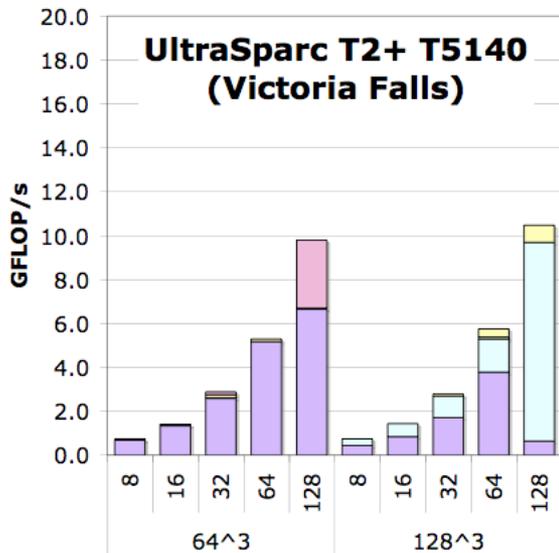


LBMHD Auto-tuner

FUTURE TECHNOLOGIES GROUP



- ❖ and another for a key kernel extracted from a lattice-boltzmann magneto-hydrodynamics code (LBMHD)
- ❖ Essentially a variant on stencils/structured grids



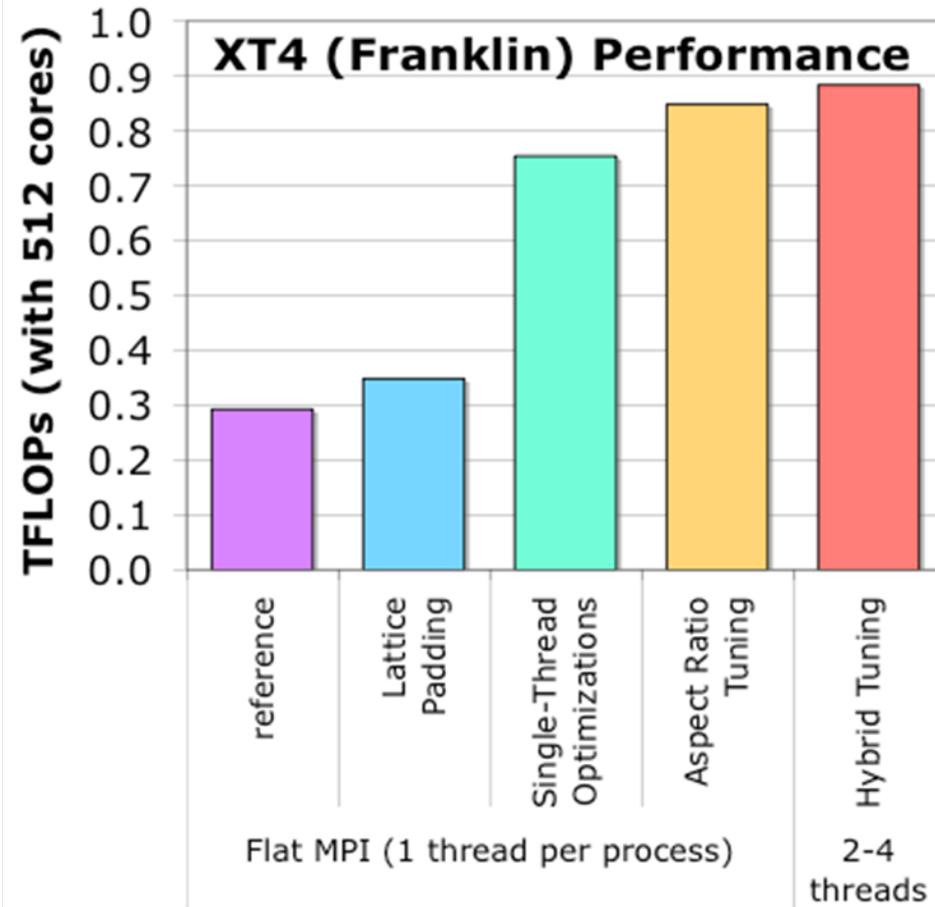
- +small pages
- +Explicit SIMDization
- +SW Prefetching
- +Unrolling
- +Vectorization
- +Padding
- Naïve+NUMA



Auto-tuning Distributed Applications

FUTURE TECHNOLOGIES GROUP

- ❖ Additionally, we extended auto-tuning to the distributed implementation of LBMHD
- ❖ Implemented an auto-tuned hybrid version where we tune for the optimal balance between threads and processes.
- ❖ **+600 Gflop/s** performance boost at 128 nodes
- ❖ Note each of the last 3 bars may have unique MPI decompositions as well as VL/unroll/DLP
- ❖ Observe that for this large problem, auto-tuning flat MPI delivered significant boosts (2.5x)
- ❖ However, expanding auto-tuning to include the domain decomposition and balance between threads and processes provided an extra 17%
- ❖ 2 processes with 2 threads was best





Particle Methods PIC, FMM, etc..

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

- ❖ As Kamesh discussed yesterday, we've optimized a **particle-in-cell (PIC)**, called GTC for multicore and are considering when/where/how auto-tuning is needed
- ❖ Similarly, we're (led by Aparna Chandramowliswaran) conducting the initial research into optimizing the **Fast Multipole Method (FMM)** for multicore.



Limitations of Auto-tuning



Limitations

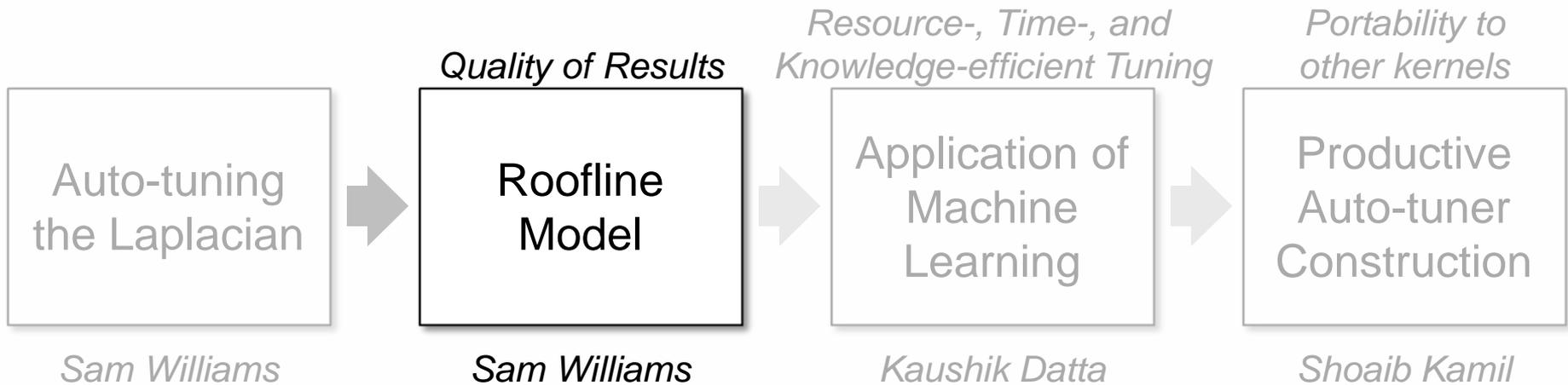
F U T U R E T E C H N O L O G I E S G R O U P

- ❖ Although we showed auto-tuning dramatically improved performance there are some critical limitations to our approach:
 - There are currently no absolutes when it comes to performance (only relative statements of better)
 - The designer must possess PhD-level architectural- and domain-knowledge to both construct the code generator and search components.
 - Even with a greedy search algorithm the tuning time/resources/foreknowledge can be prohibitively high.
 - The auto-tuner is specific to one kernel. Any changes to that kernel (changing 7pt to 27pt) requires constructing an entirely new auto-tuner.

- ❖ **Our latest research is focused on these problems and the results to date will be discussed parts II, III, and IV.**

II

Quantitative Assessment of Results





Motivation for Performance Assessment

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

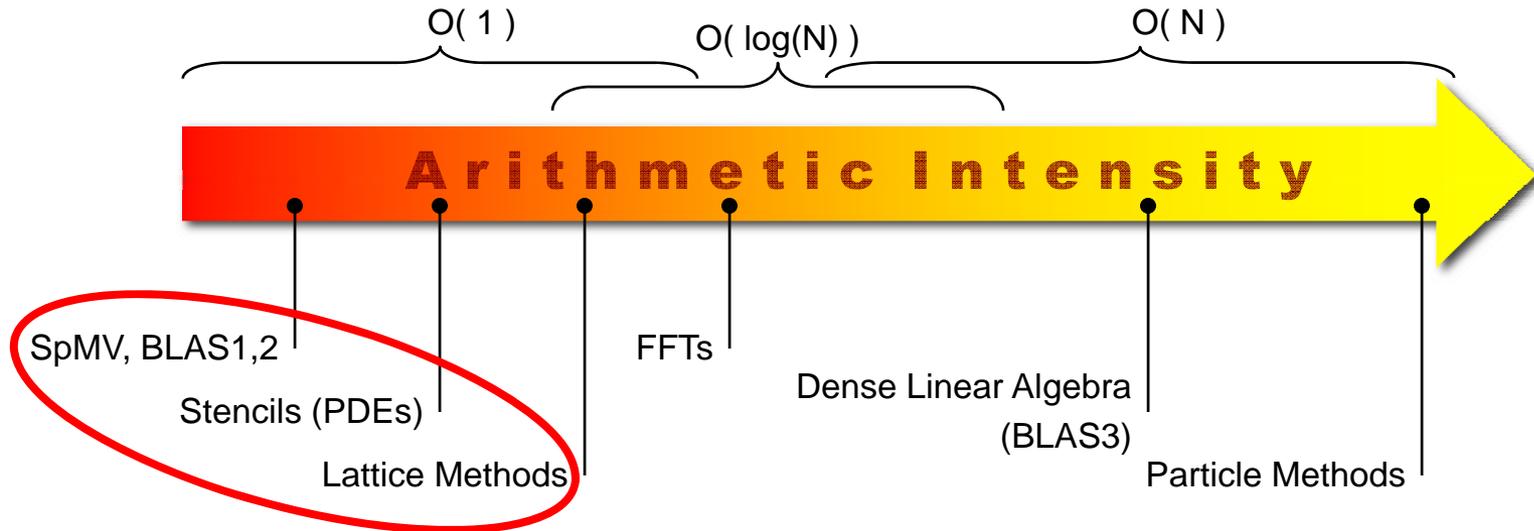
- ❖ We want to say more than simply performance improved by 147%
- ❖ We want to be able to say that no matter how much additional work is poured into the tuner, performance can only increase by 5%.

- ❖ With this information:
 - we can abort tuning early (when we reach a certain level of quality)
 - in conjunction with performance counters, dynamically select the optimization order of the greedy search.
 - pinpoint performance bottlenecks to motivate new optimizations
 - pinpoint performance bottlenecks so as to drive future procurements
 - provide feedback to architects and computational scientists as to how to design the next generation of architectures or algorithms.

- ❖ We believe that this information can be easily visualized with a model we call the **Roofline Model**



Key Concepts...



- ❖ **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.



Categorization of Software Optimizations



Optimization Categorization

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

**Maximizing (*attained*)
In-core Performance**

**Maximizing (*attained*)
Memory Bandwidth**

**Minimizing (*total*)
Memory Traffic**



Optimization Categorization

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

Maximizing In-core Performance

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

Maximizing Memory Bandwidth

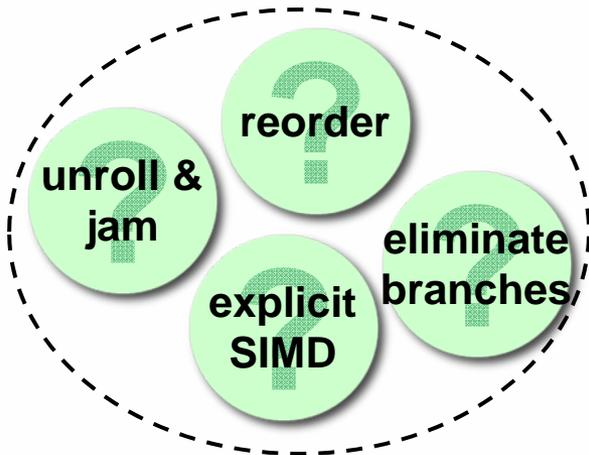
Minimizing Memory Traffic

Maximizing In-core Performance

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

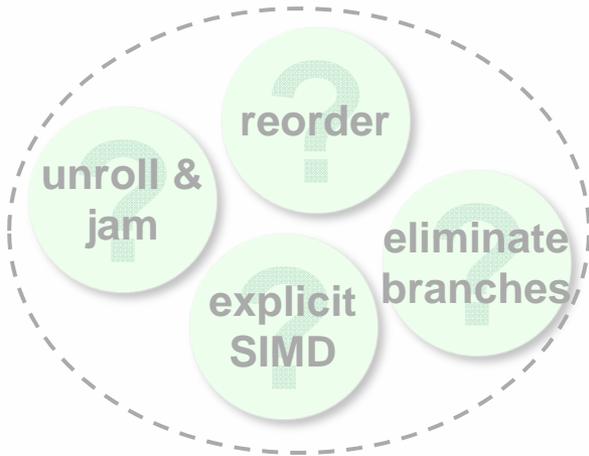
Maximizing Memory Bandwidth

Minimizing Memory Traffic



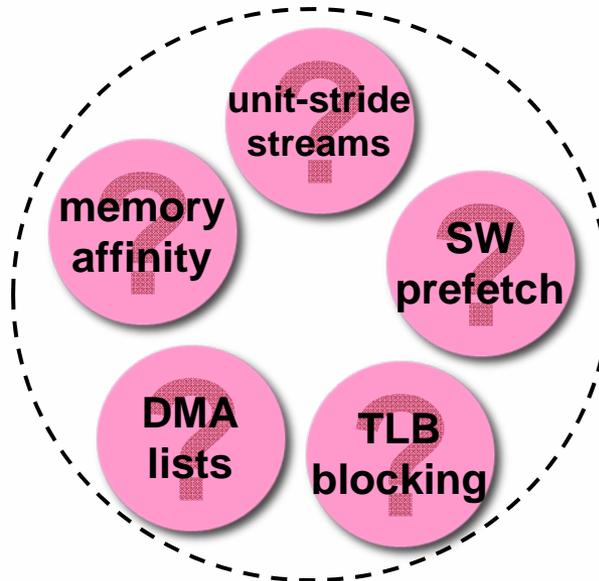
Maximizing In-core Performance

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



Maximizing Memory Bandwidth

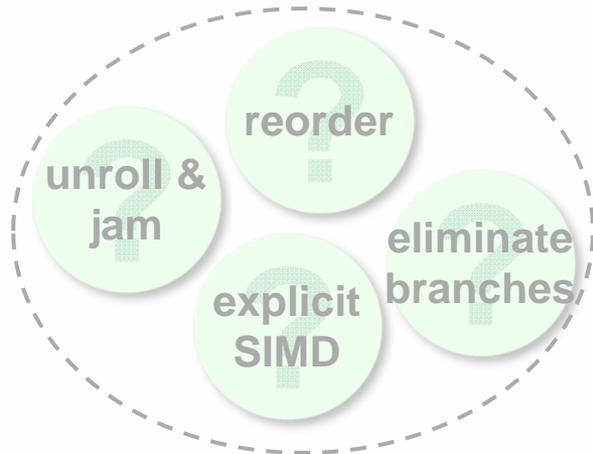
- Exploit NUMA
- Hide memory latency
- Satisfy Little's Law



Minimizing Memory Traffic

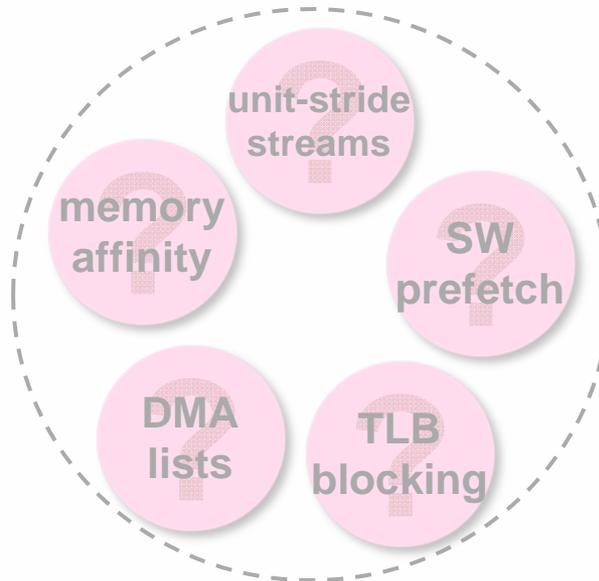
Maximizing In-core Performance

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



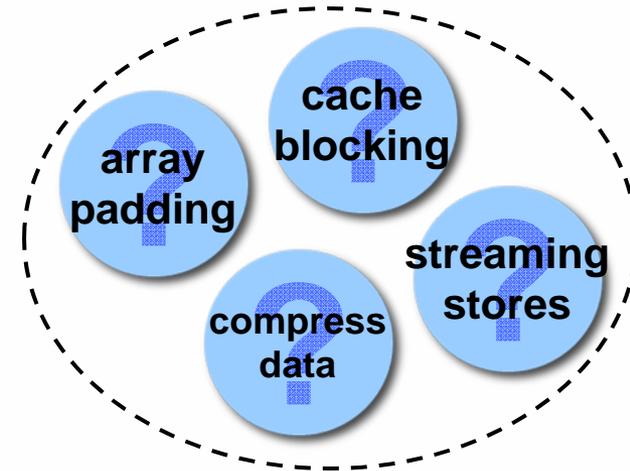
Maximizing Memory Bandwidth

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



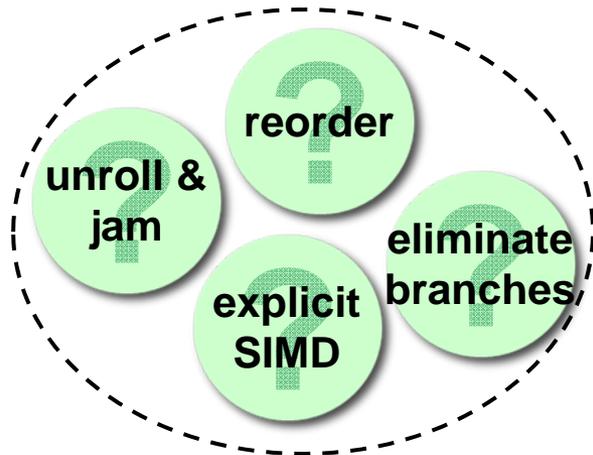
Minimizing Memory Traffic

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



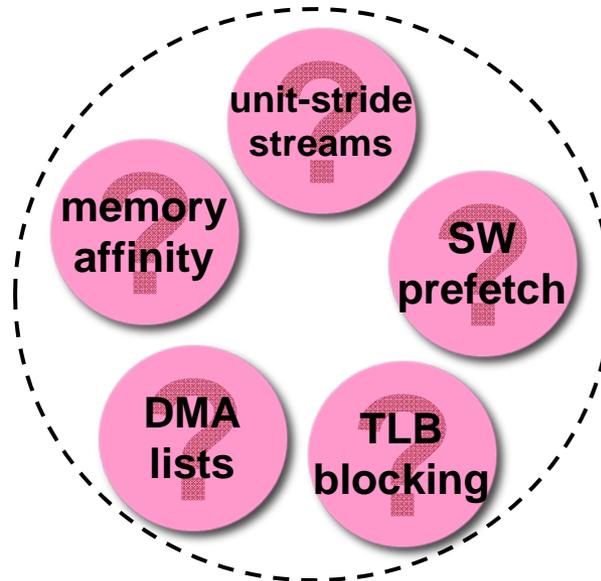
Maximizing In-core Performance

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



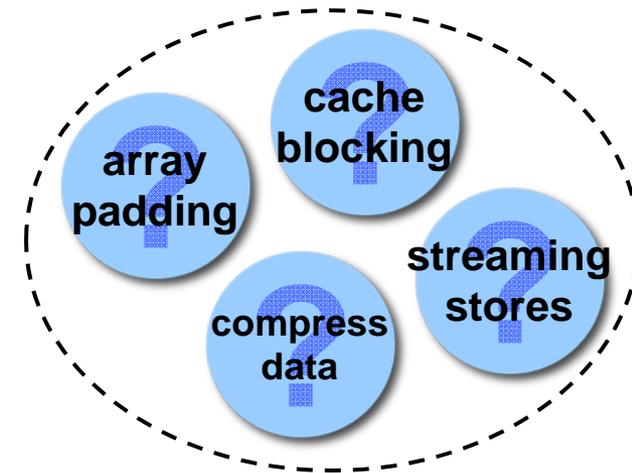
Maximizing Memory Bandwidth

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



Minimizing Memory Traffic

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





Introduction to the Roofline Model



Roofline Model

Basic Concept

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

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

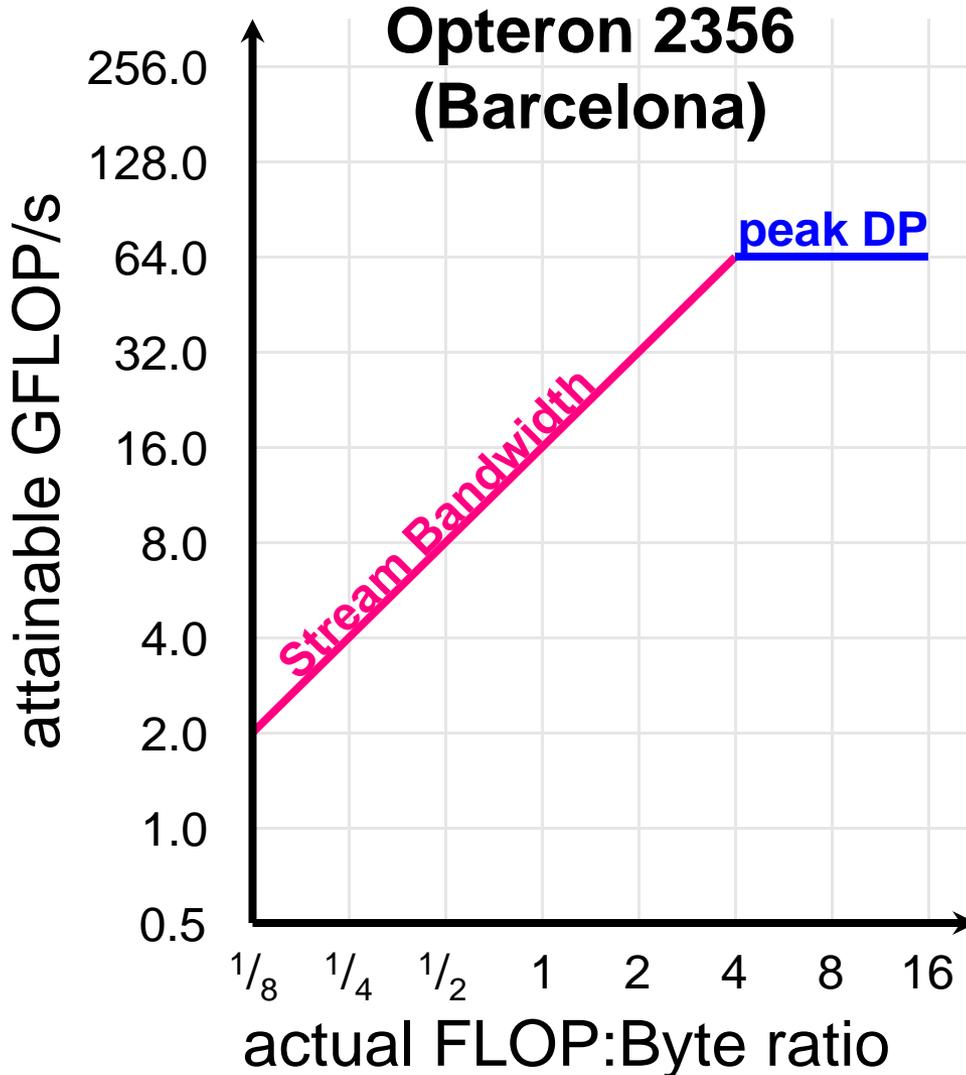
$$\text{Attainable Performance}_{ij} = \min \left\{ \begin{array}{l} \text{FLOP/s with Optimizations}_{1-i} \\ \text{AI} * \text{Bandwidth with Optimizations}_{1-j} \end{array} \right.$$

- ❖ 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

FUTURE TECHNOLOGIES GROUP



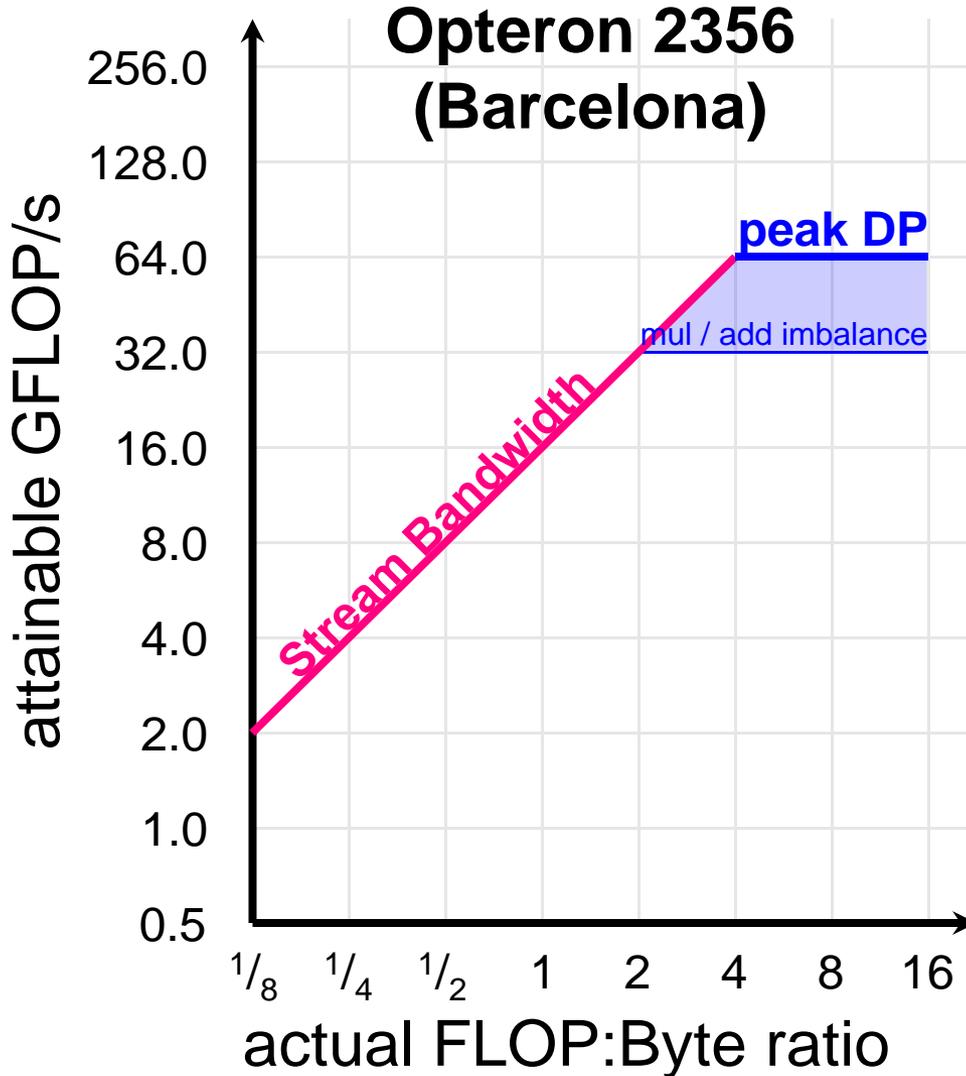
- ❖ Plot on log-log scale
- ❖ Given AI, we can easily bound performance
- ❖ But architectures are much more complicated
- ❖ We will bound performance as we eliminate specific forms of in-core parallelism



Roofline Model

computational ceilings

FUTURE TECHNOLOGIES GROUP

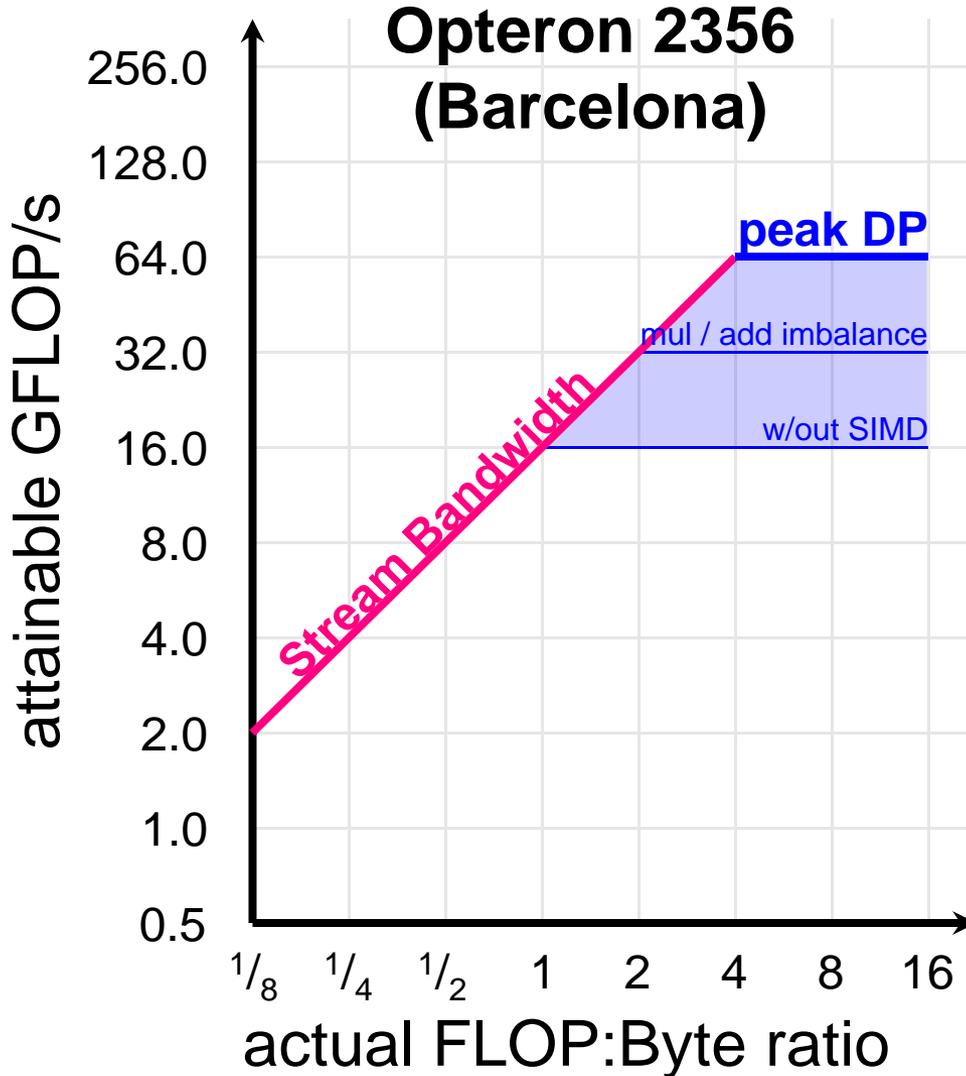


- ❖ Opterons have dedicated multipliers and adders.
- ❖ If the code is dominated by adds, then attainable performance is half of peak.
- ❖ We call these **Ceilings**
- ❖ They act like constraints on performance

Roofline Model

computational ceilings

FUTURE TECHNOLOGIES GROUP

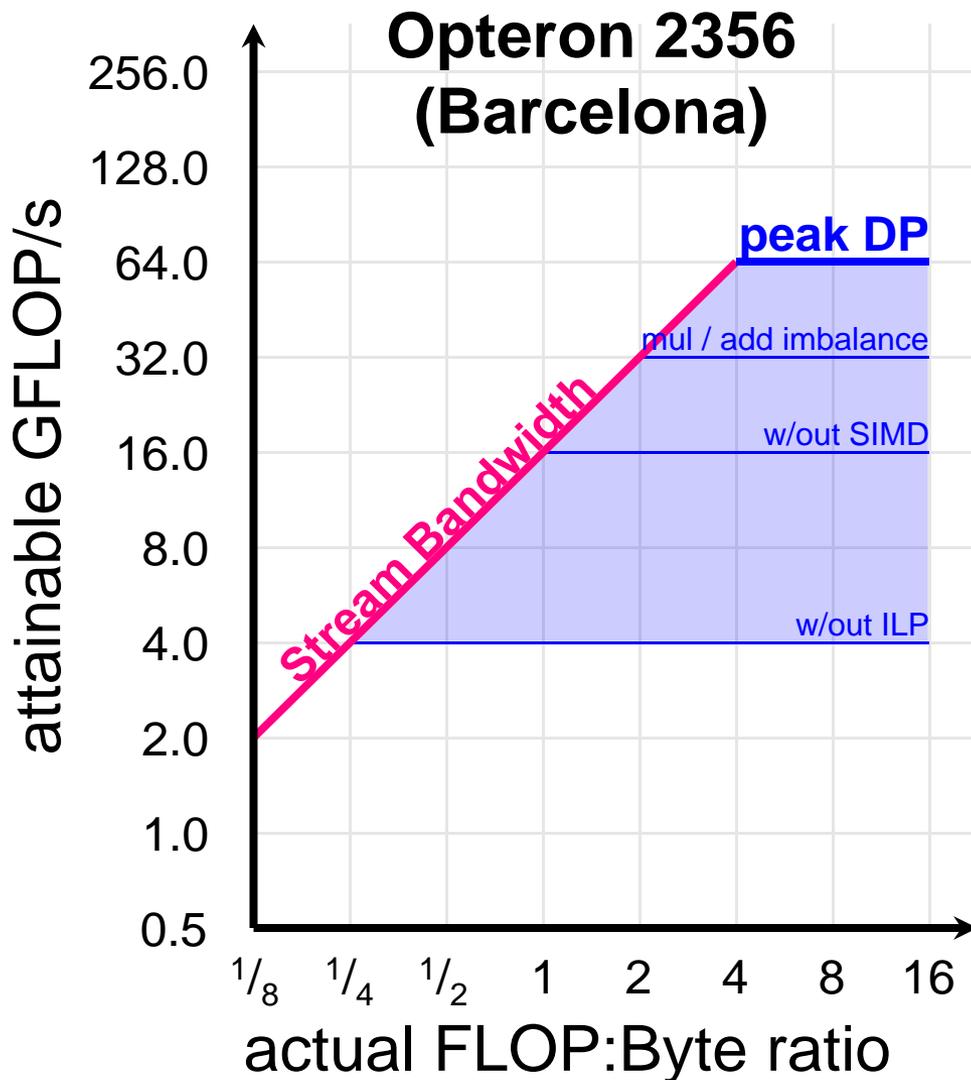


- ❖ Opterons have 128-bit datapaths.
- ❖ If instructions aren't SIMDized, attainable performance will be halved

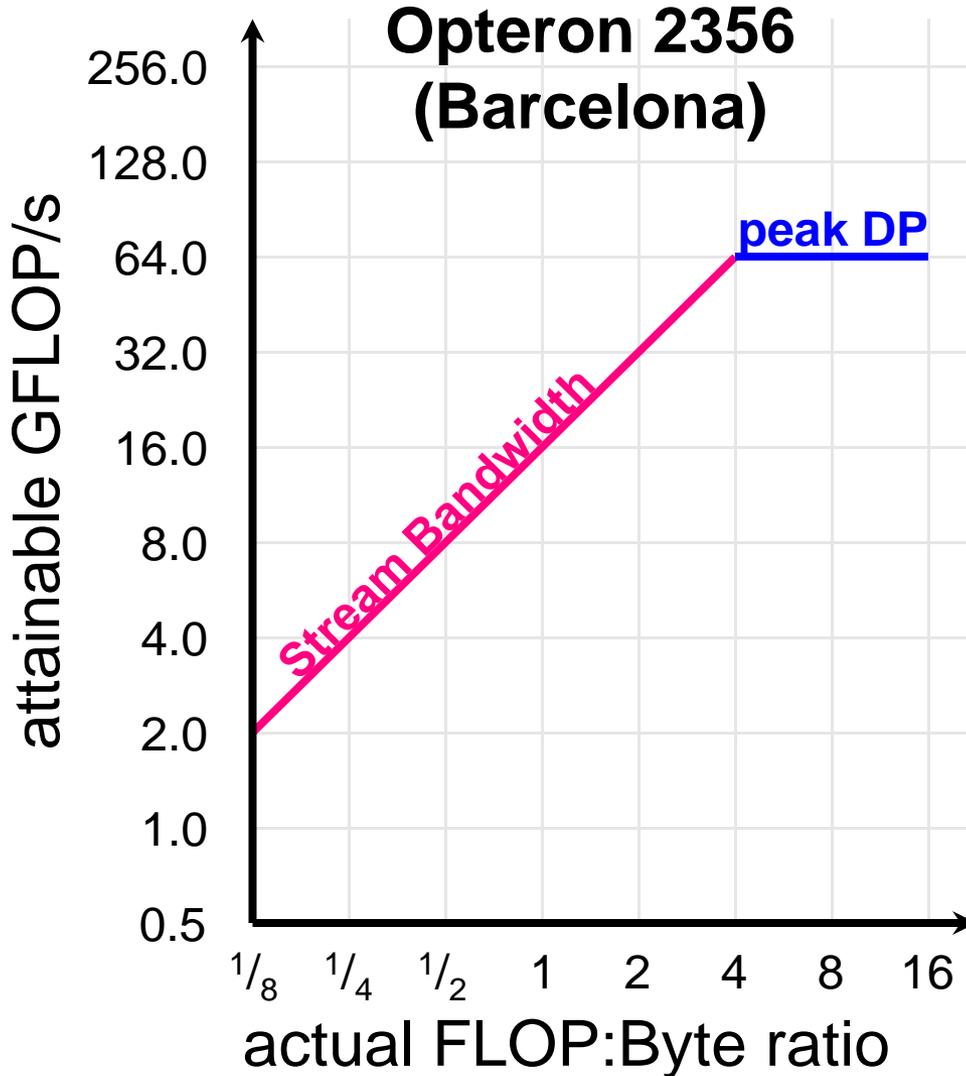
Roofline Model

computational ceilings

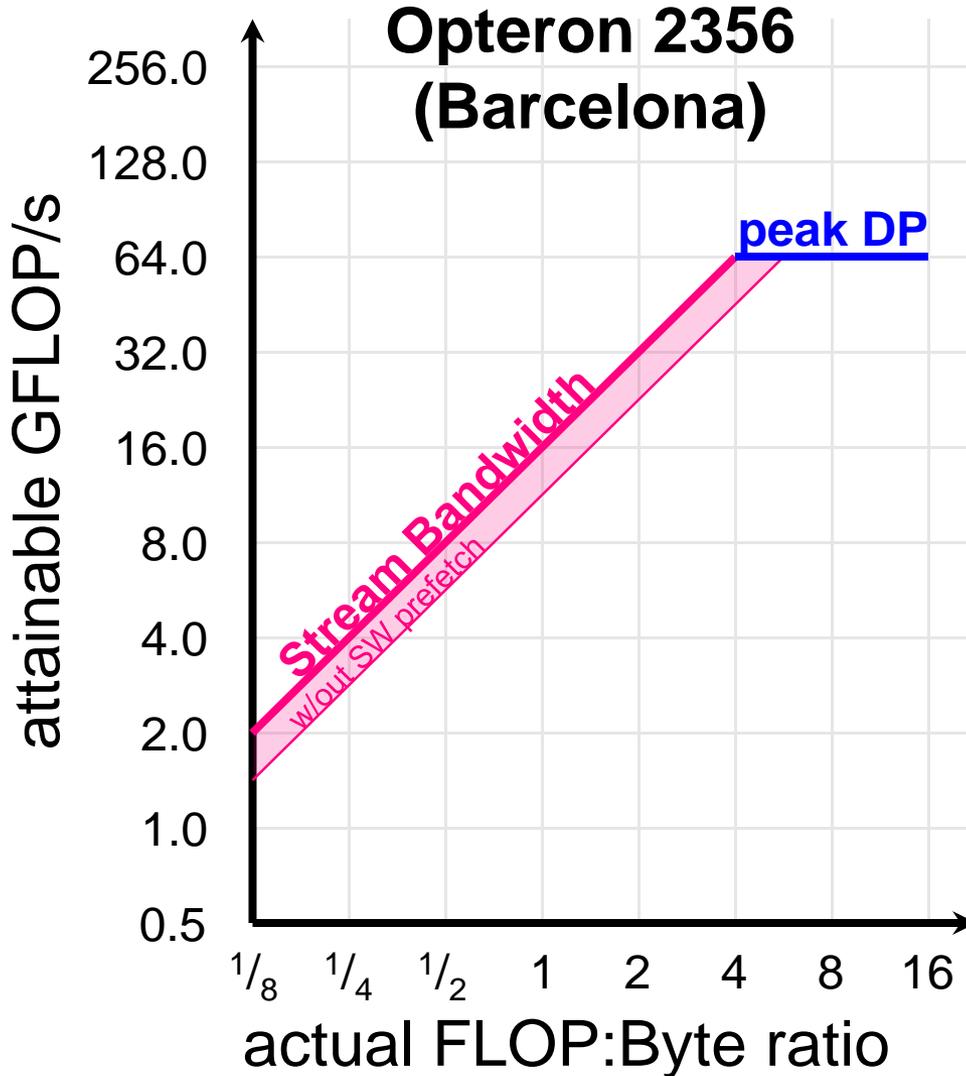
FUTURE TECHNOLOGIES GROUP



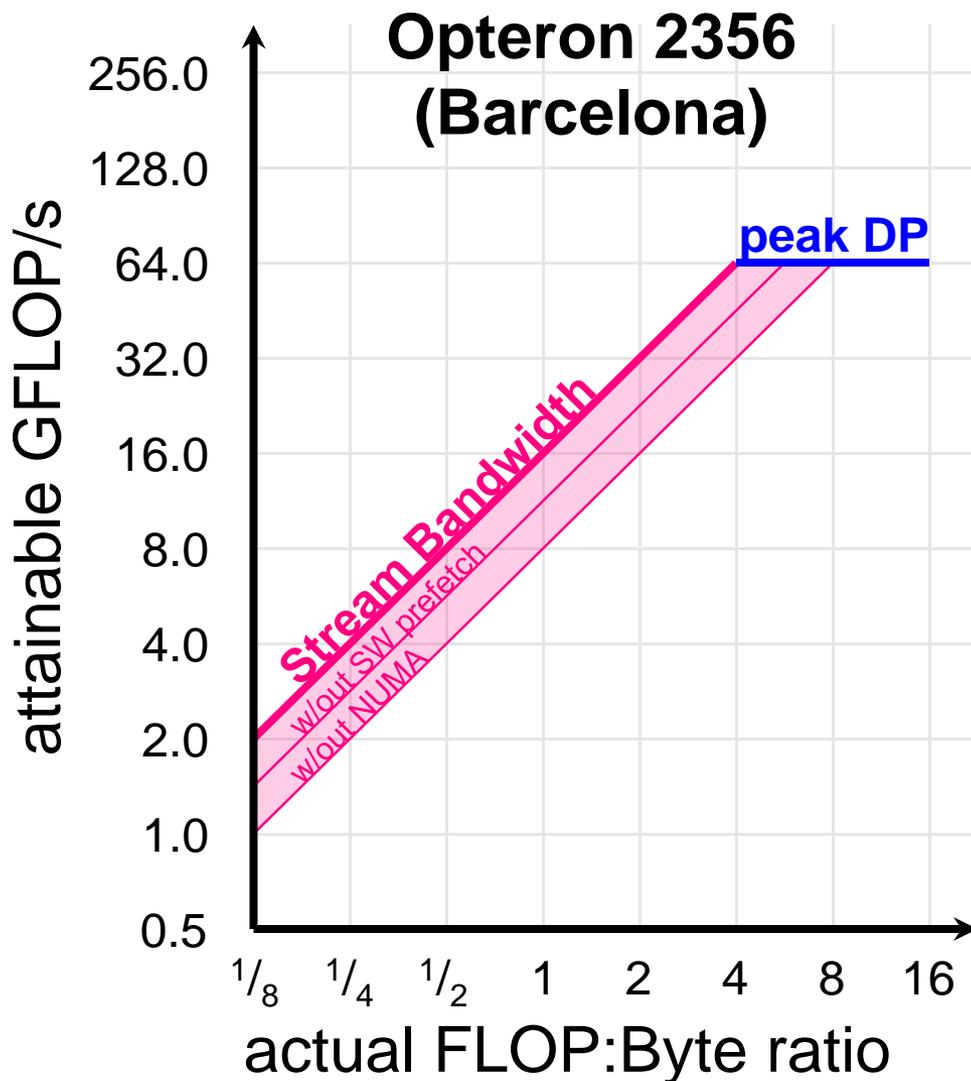
- ❖ On Opterons, floating-point instructions have a 4 cycle latency.
- ❖ If we don't express 4-way ILP, performance will drop by as much as 4x



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



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

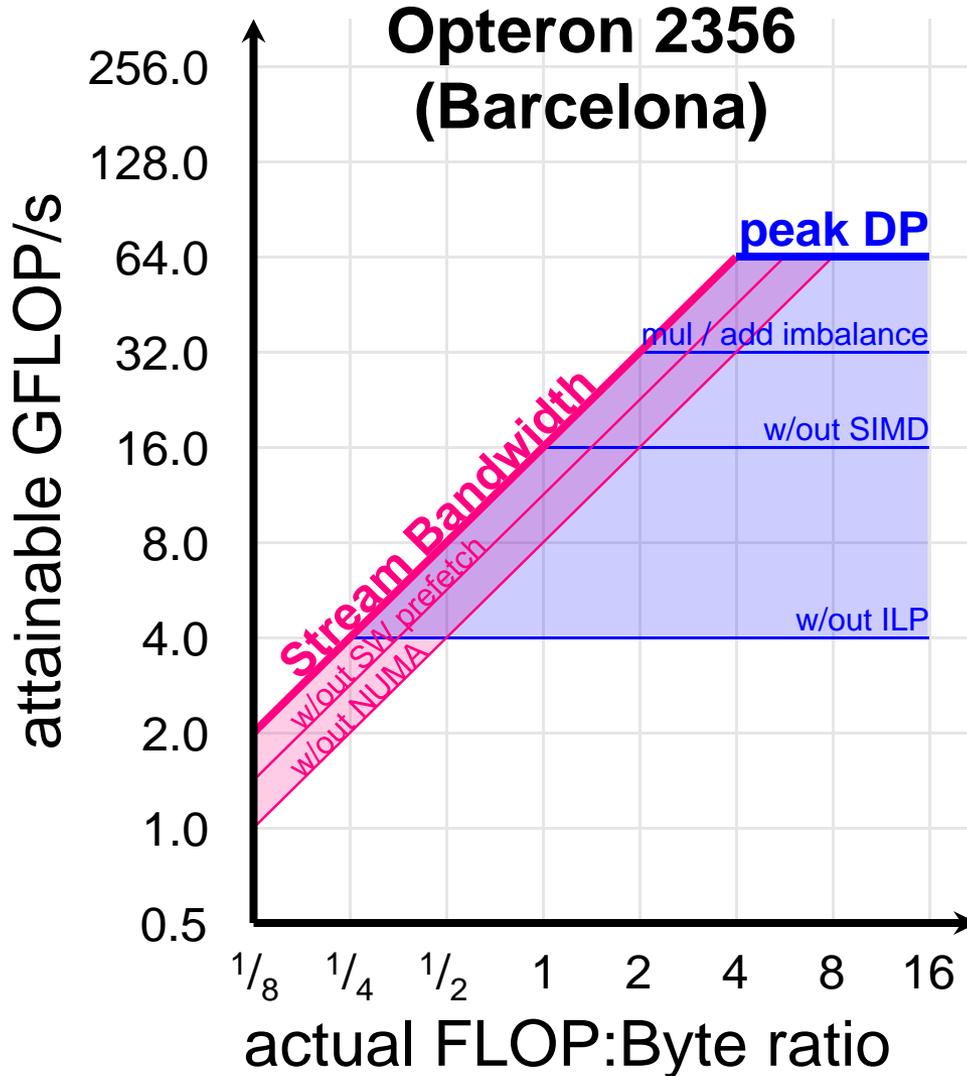


- ❖ Opterons are NUMA
- ❖ As such memory traffic must be correctly balanced among the two sockets to achieve good Stream bandwidth.
- ❖ We could continue this by examining strided or random memory access patterns

Roofline Model

computation + communication ceilings

FUTURE TECHNOLOGIES GROUP

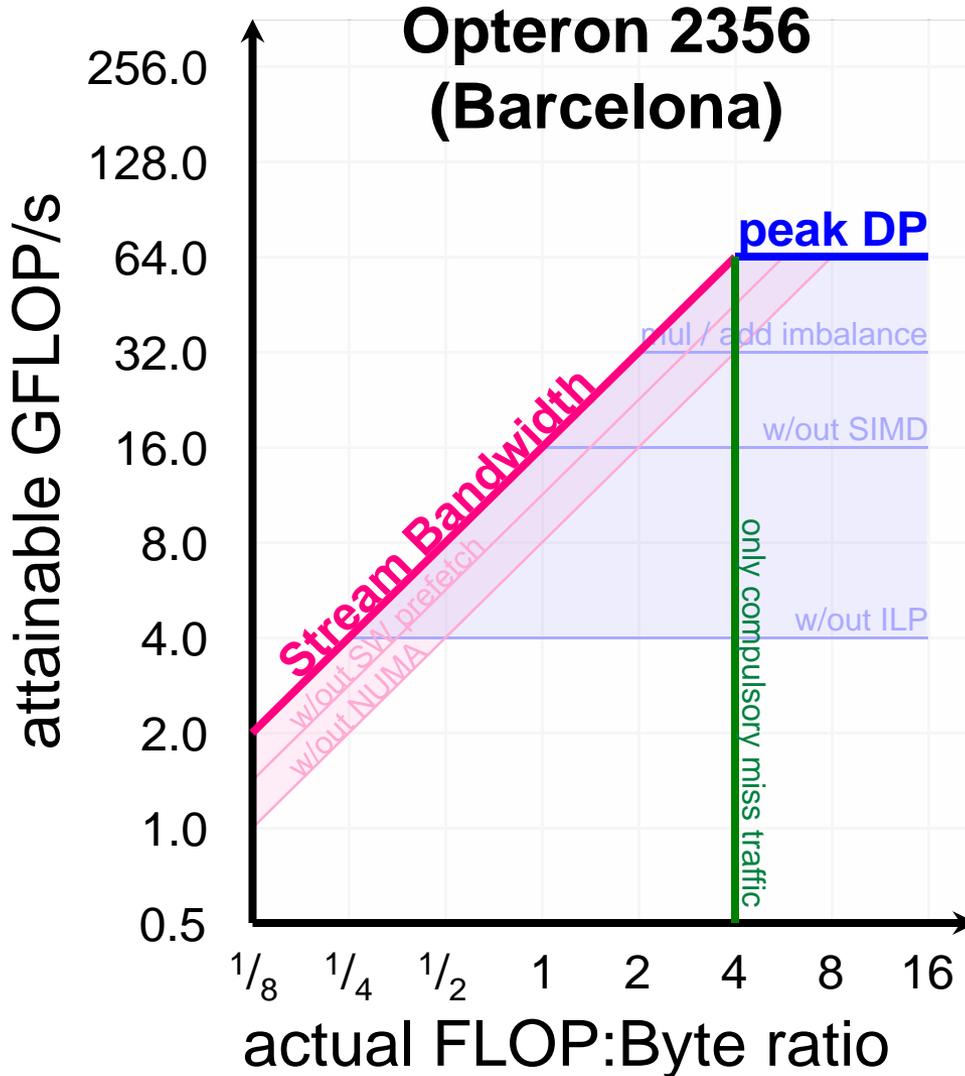


- ❖ We may bound performance based on the combination of expressed in-core parallelism and attained bandwidth.

Roofline Model

locality walls

FUTURE TECHNOLOGIES GROUP



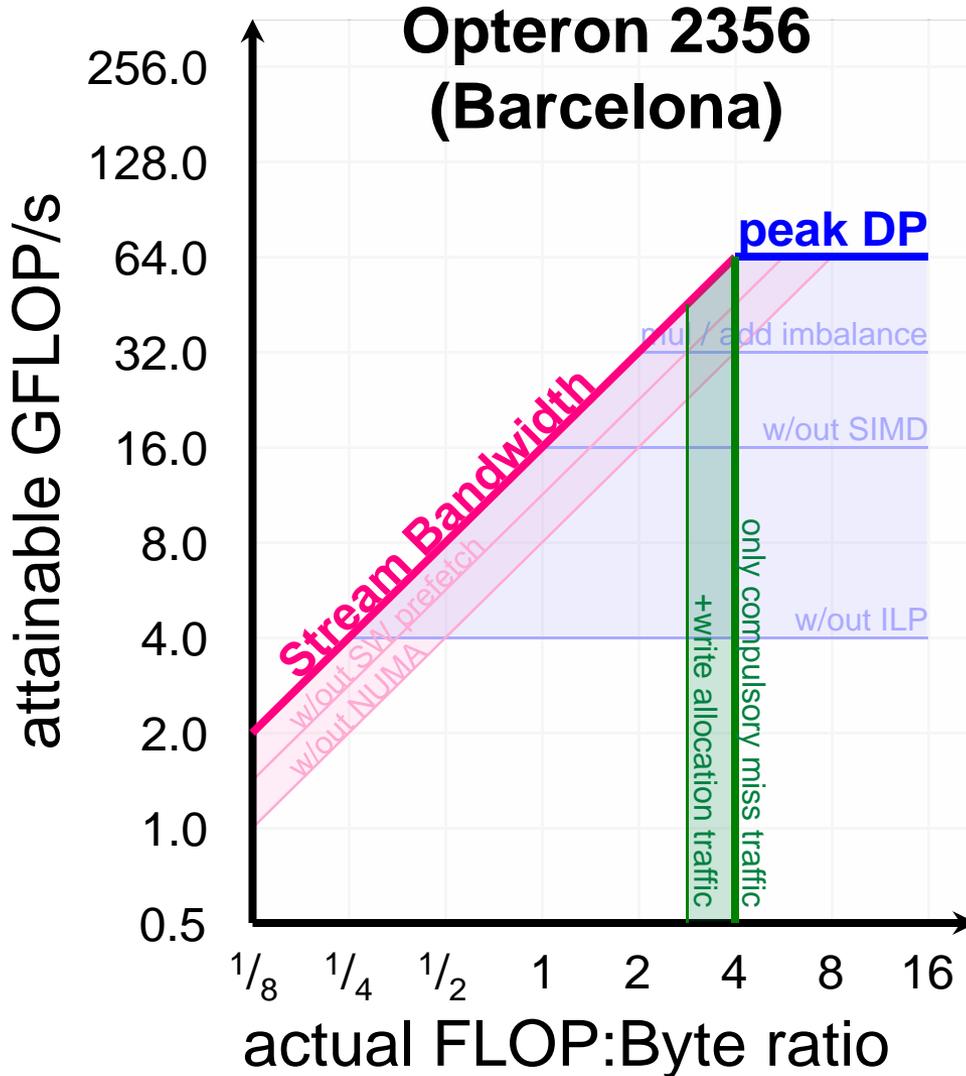
- ❖ Remember, memory traffic includes more than just compulsory misses.
- ❖ As such, actual arithmetic intensity may be substantially lower.
- ❖ Walls are unique to the architecture-kernel combination

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

Roofline Model

locality walls

FUTURE TECHNOLOGIES GROUP



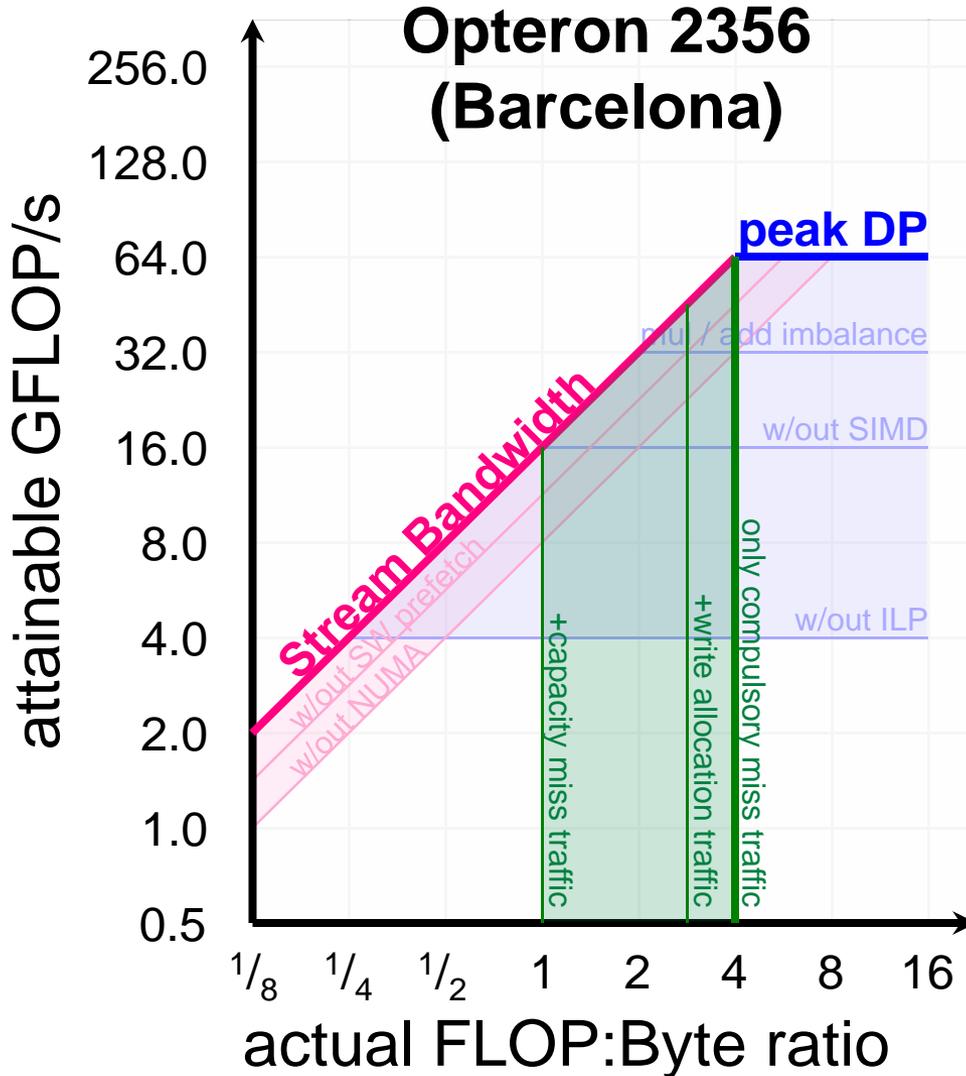
- ❖ Remember, memory traffic includes more than just compulsory misses.
- ❖ As such, actual arithmetic intensity may be substantially lower.
- ❖ Walls are unique to the architecture-kernel combination

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

Roofline Model

locality walls

FUTURE TECHNOLOGIES GROUP



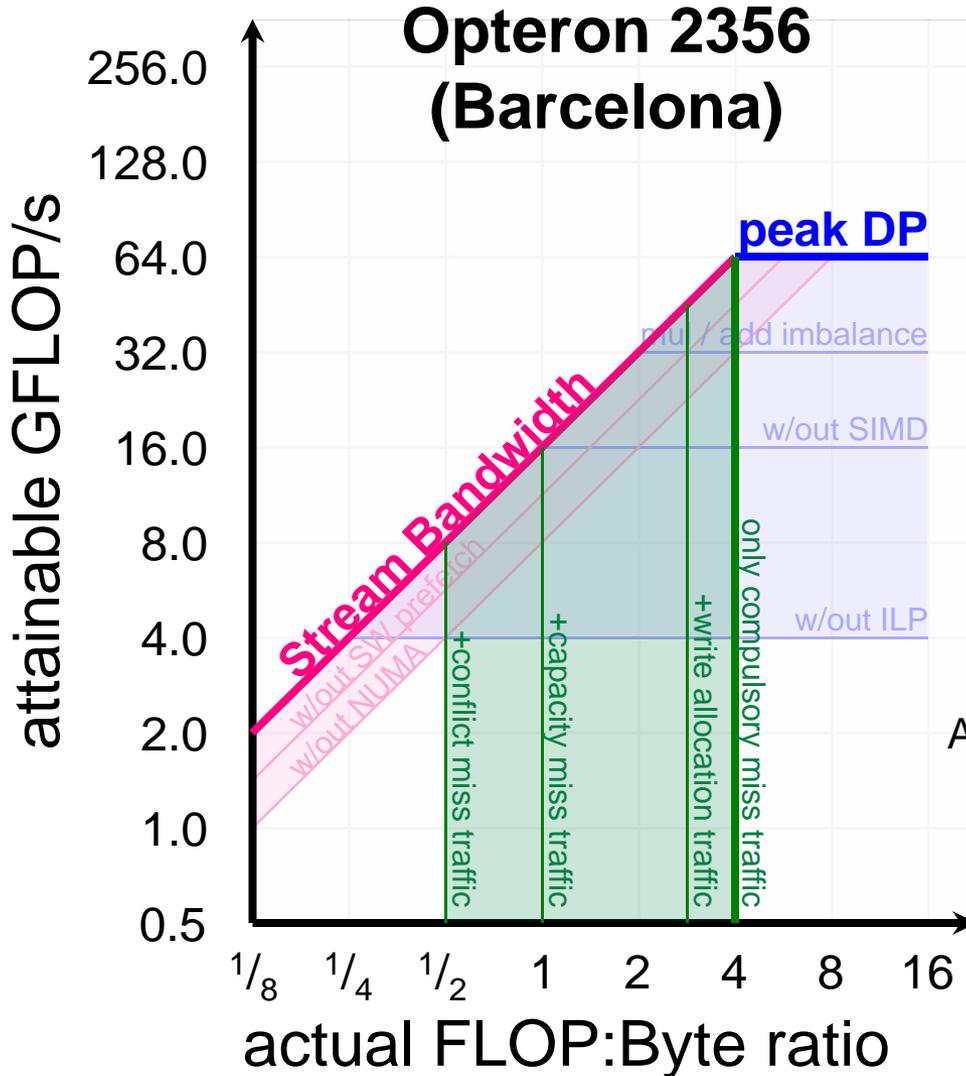
- ❖ Remember, memory traffic includes more than just compulsory misses.
- ❖ As such, actual arithmetic intensity may be substantially lower.
- ❖ Walls are unique to the architecture-kernel combination

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

Roofline Model

locality walls

FUTURE TECHNOLOGIES GROUP



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

- ❖ 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



Ceiling-Optimization Interplay

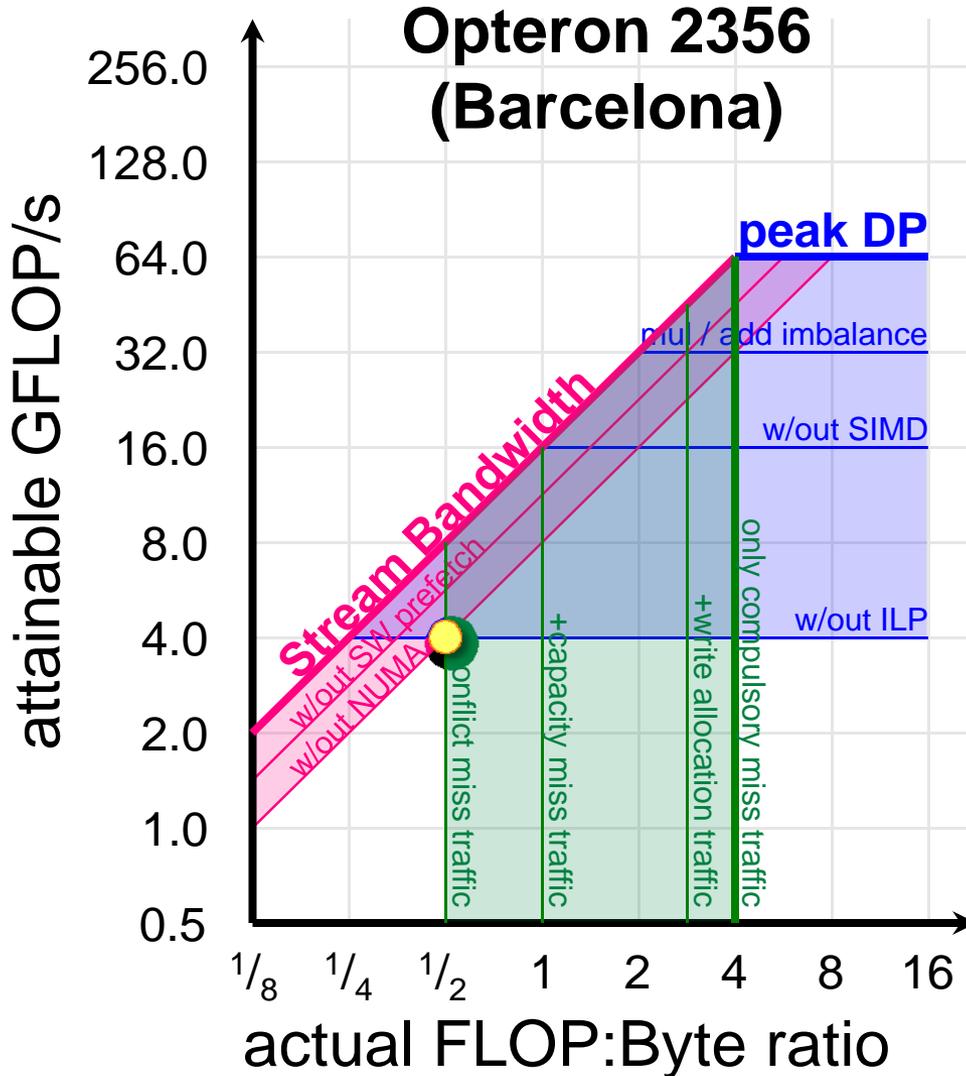
F U T U R E T E C H N O L O G I E S G R O U P

- ❖ Ceilings act to constrain performance coordinates
 - ceilings limit performance
 - walls limit AI
- ❖ Optimizations target specific ceilings and remove them as (potential) constraints to performance.

Roofline Model

locality walls

FUTURE TECHNOLOGIES GROUP

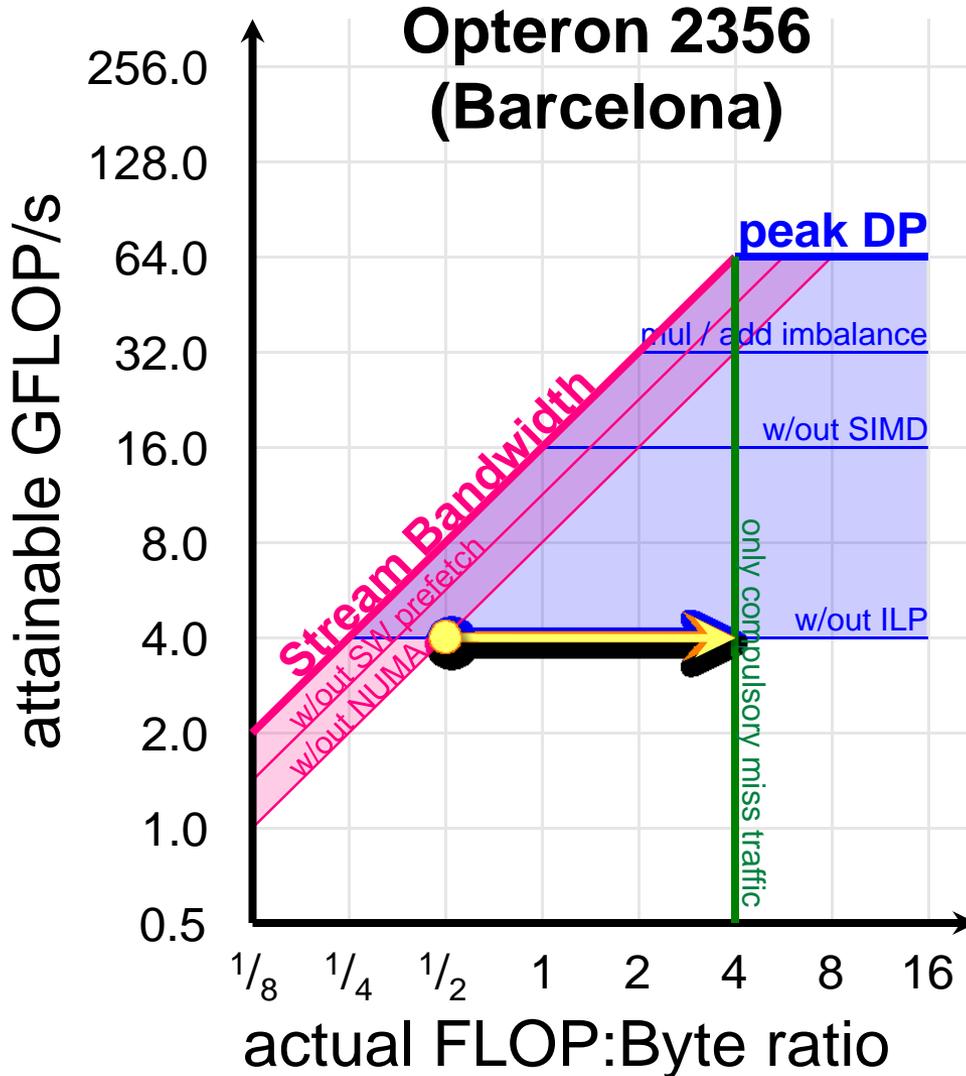


- ❖ Optimizations remove these walls and ceilings which act to constrain performance.

Roofline Model

locality walls

FUTURE TECHNOLOGIES GROUP

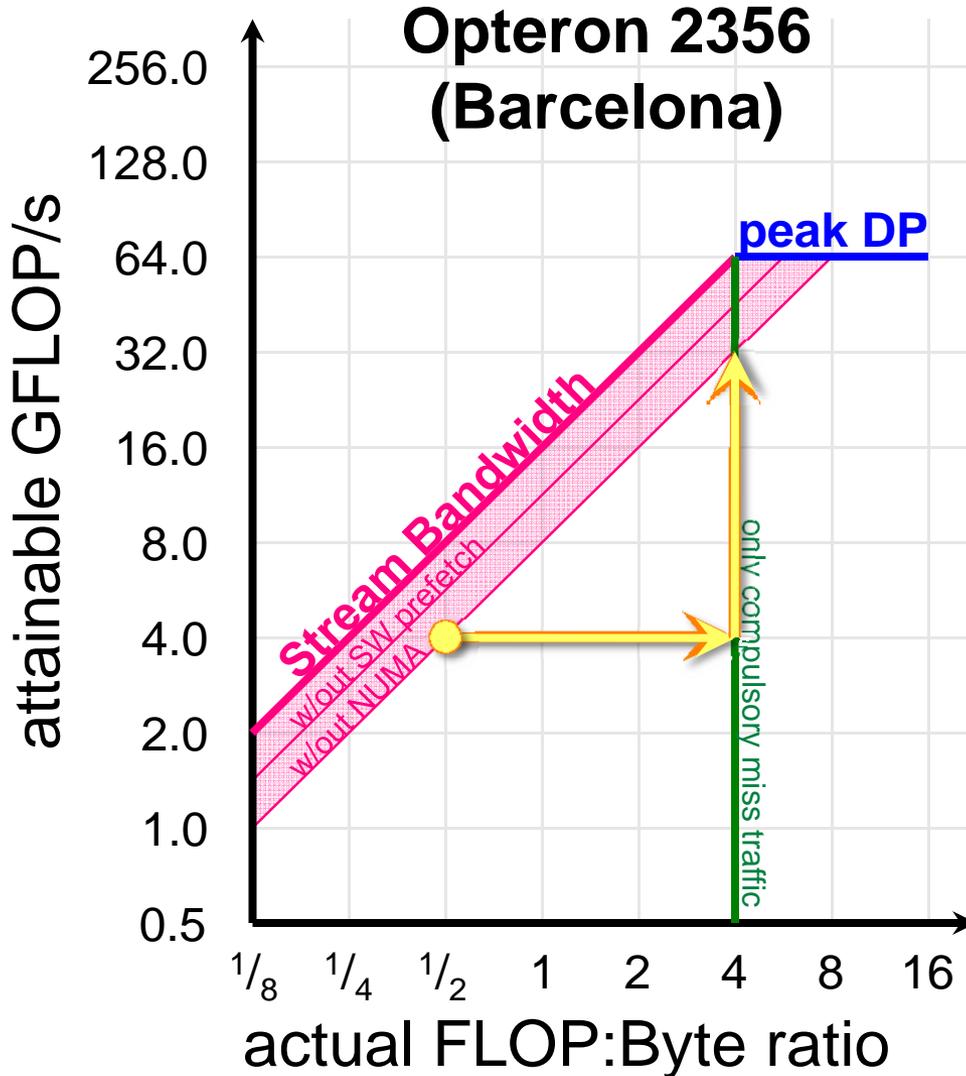


- ❖ Optimizations remove these walls and ceilings which act to constrain performance.

Roofline Model

locality walls

FUTURE TECHNOLOGIES GROUP

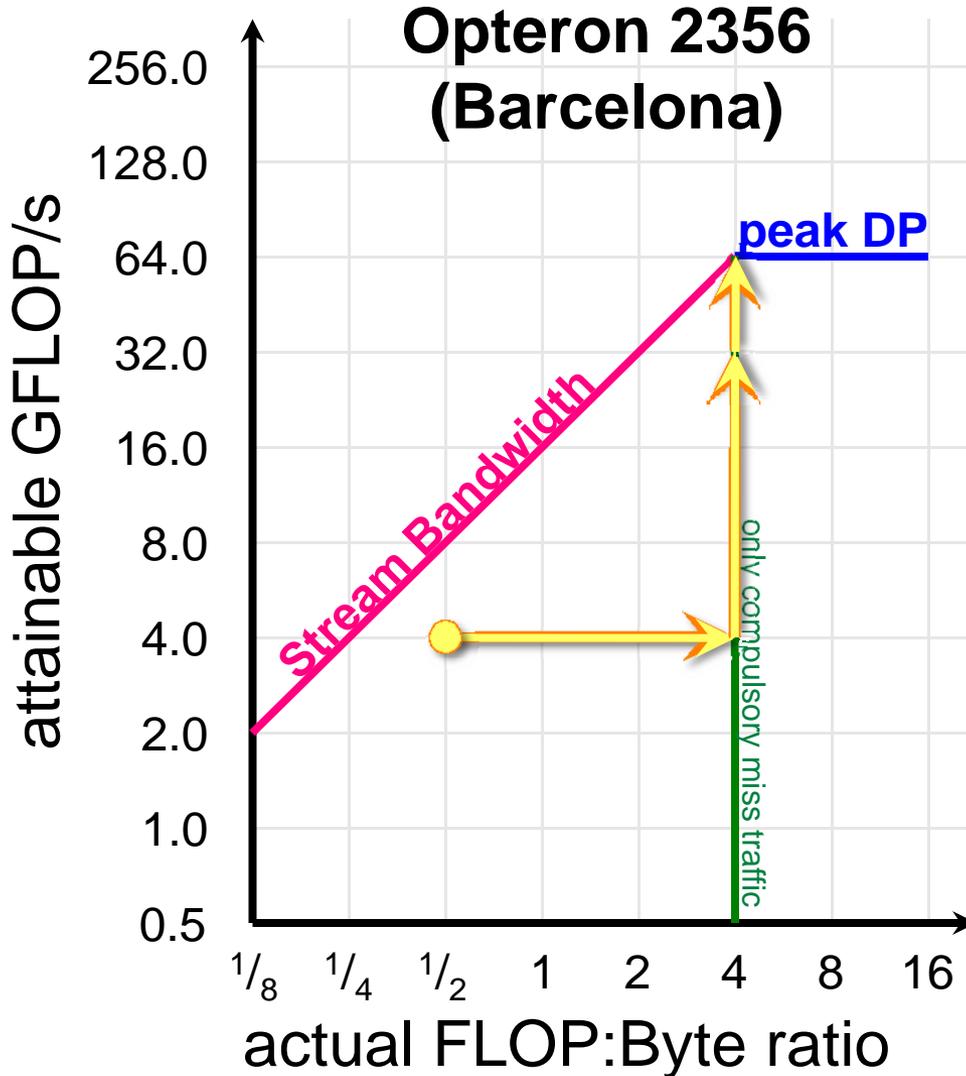


- ❖ Optimizations remove these walls and ceilings which act to constrain performance.

Roofline Model

locality walls

FUTURE TECHNOLOGIES GROUP



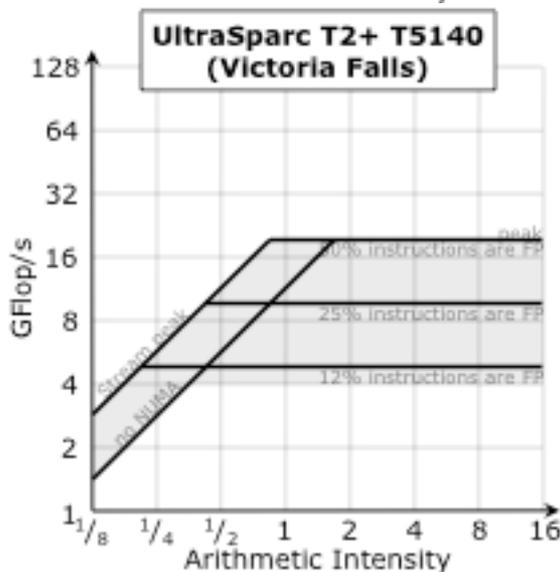
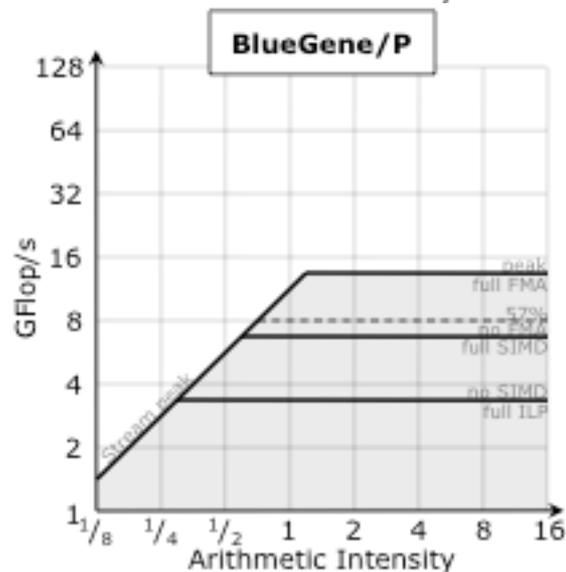
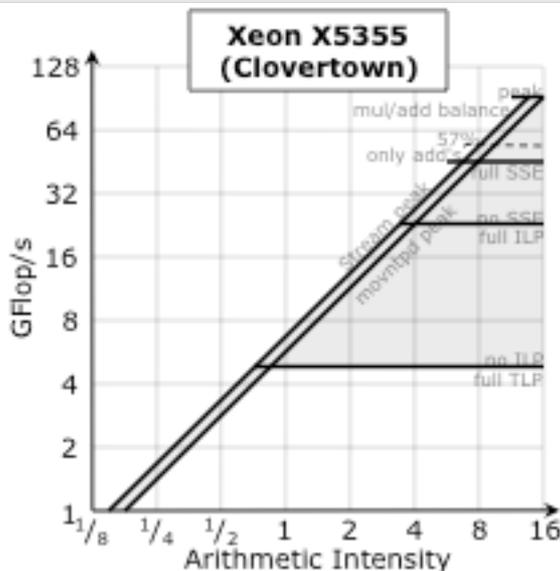
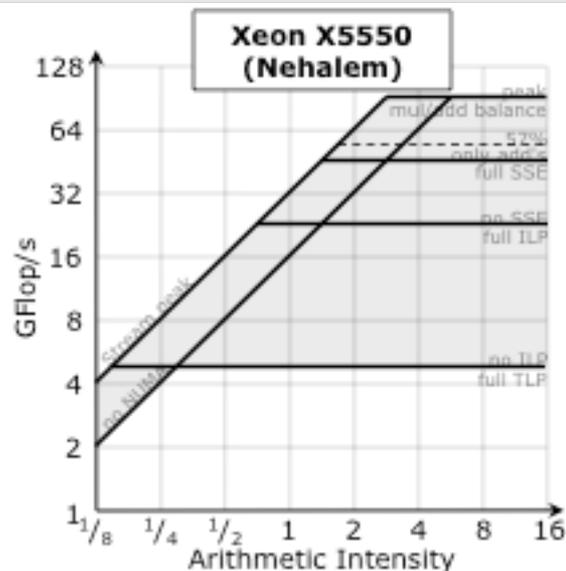
- ❖ Optimizations remove these walls and ceilings which act to constrain performance.



Application of the Roofline Model to the 7-point Stencil

Roofline Model

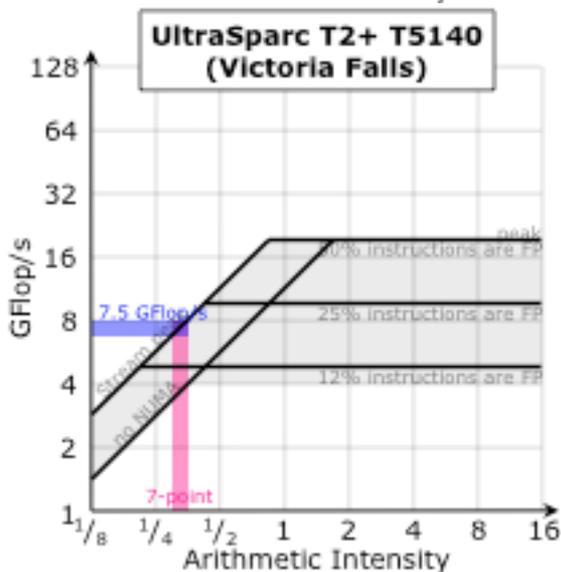
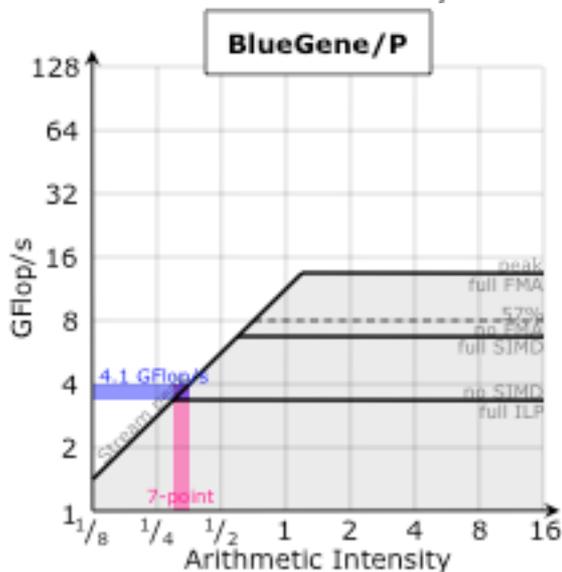
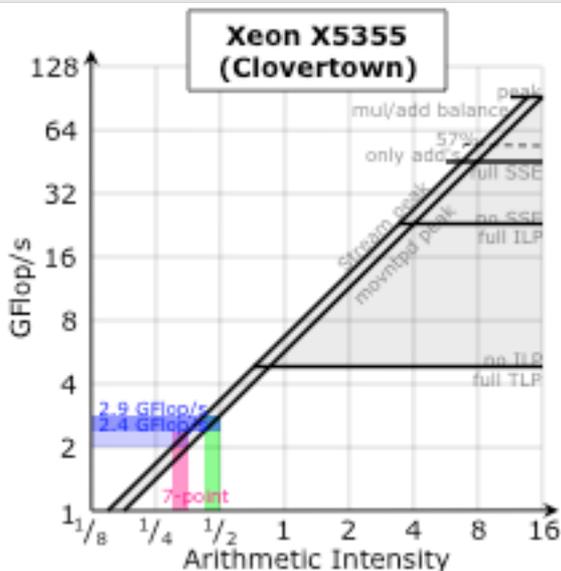
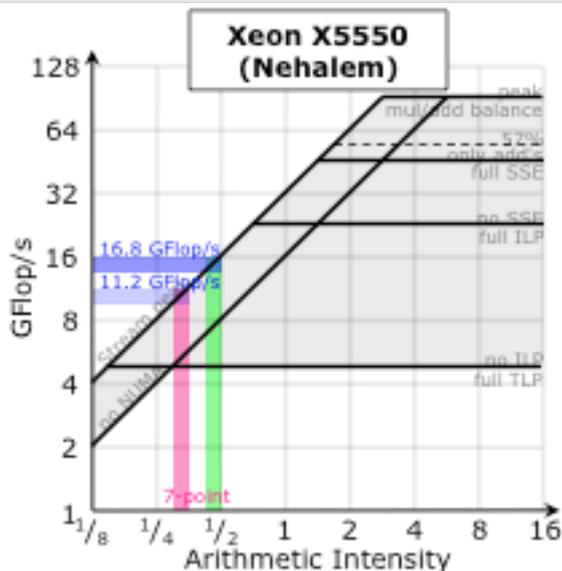
FUTURE TECHNOLOGIES GROUP



- ❖ DRAM-FP roofline models for the architectures of interest.
- ❖ NOTE: as VF is an in-order dual-issue CMT architecture, its ceilings are floating-point mix.

Roofline Model

FUTURE TECHNOLOGIES GROUP



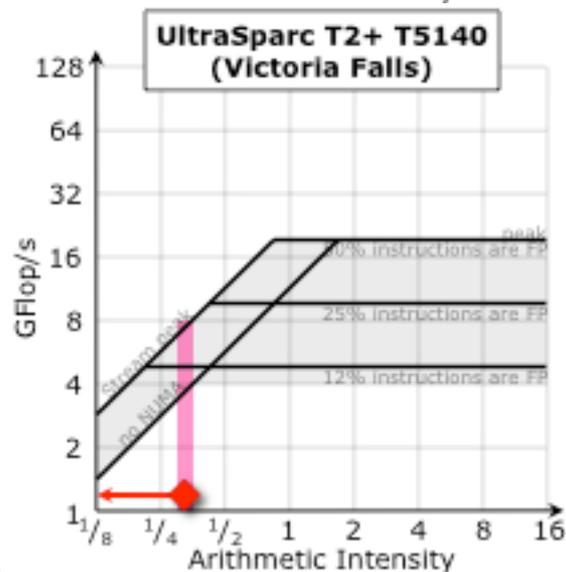
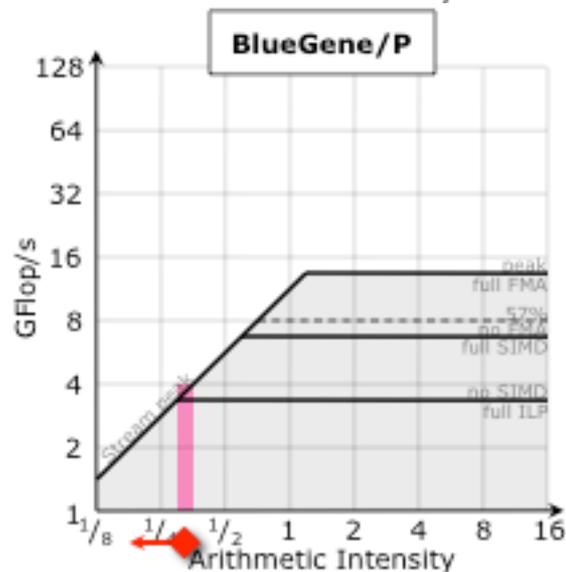
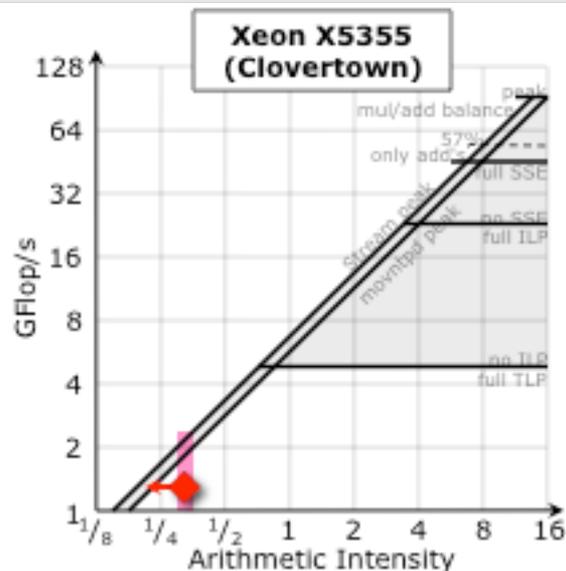
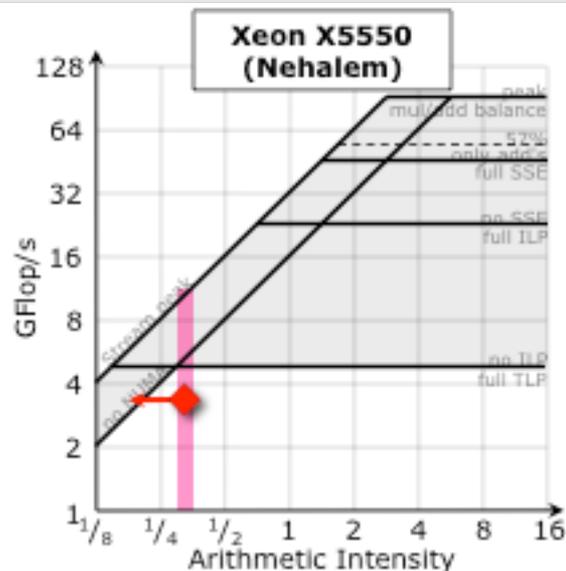
- ❖ Overlay of range in arithmetic intensity and corresponding **performance bound** (in GFlop/s)
- ❖ Clearly, performance will be heavily memory bound on some architectures, but will require varying degrees of in-core optimizations.
- ❖ Arithmetic intensity is the ideal compulsory limit for either case
- ❖ capacity misses, conflict misses, padding, prefetching will further decrease it.

❖ NOTE:

red = write
 allocate
 green = cache bypass

Roofline Model

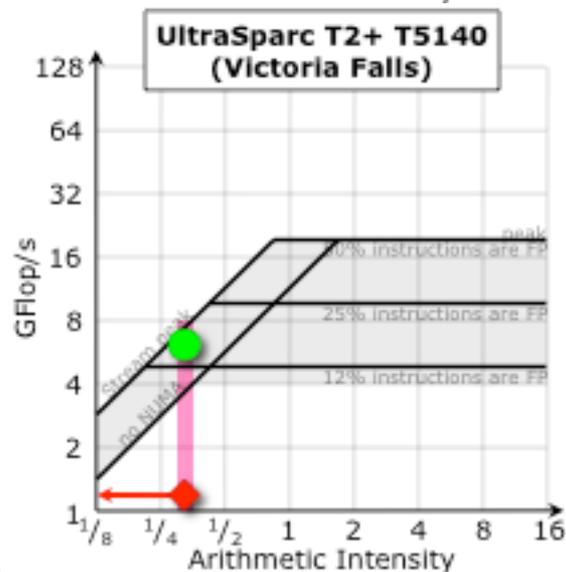
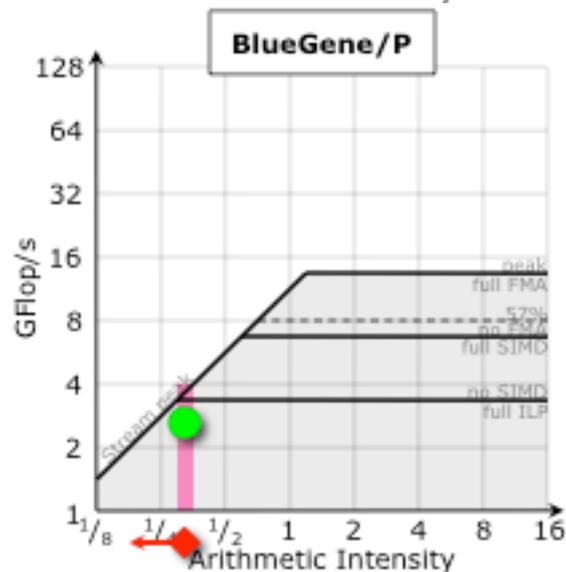
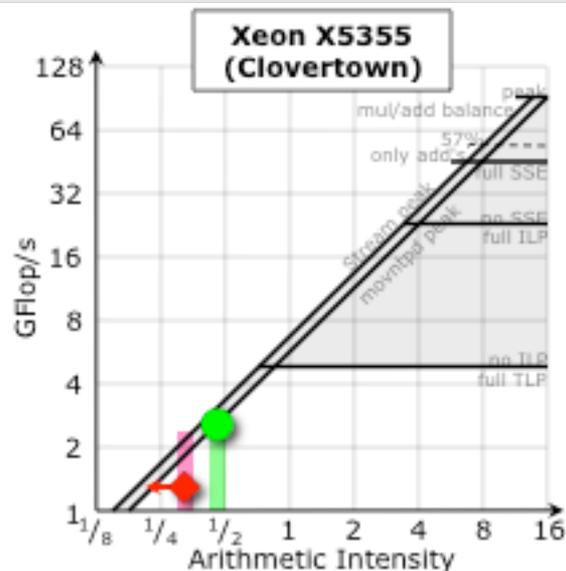
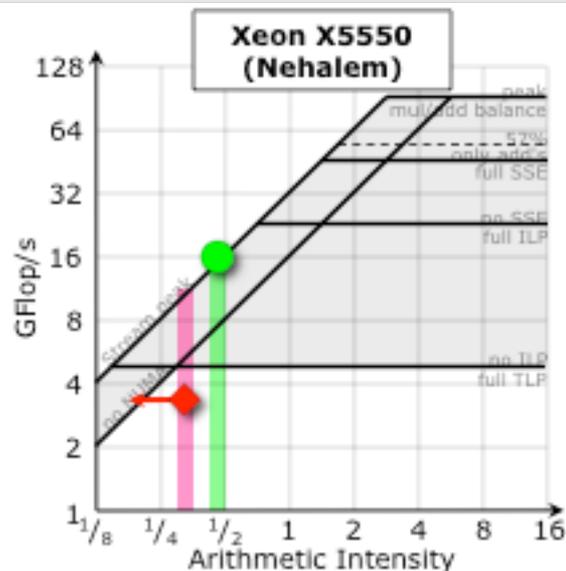
FUTURE TECHNOLOGIES GROUP



- ❖ Reference performance.
- ❖ Note, the x-coordinate is not well defined without accurate performance counters and is shown based on an ideal write-allocate cache.

Roofline Model

FUTURE TECHNOLOGIES GROUP



- ❖ Auto-tuned performance is shown in green.
- ❖ x-coordinate should be much more accurate but still an upper bound.
- ❖ **Clearly, performance is close to the roofline limit**



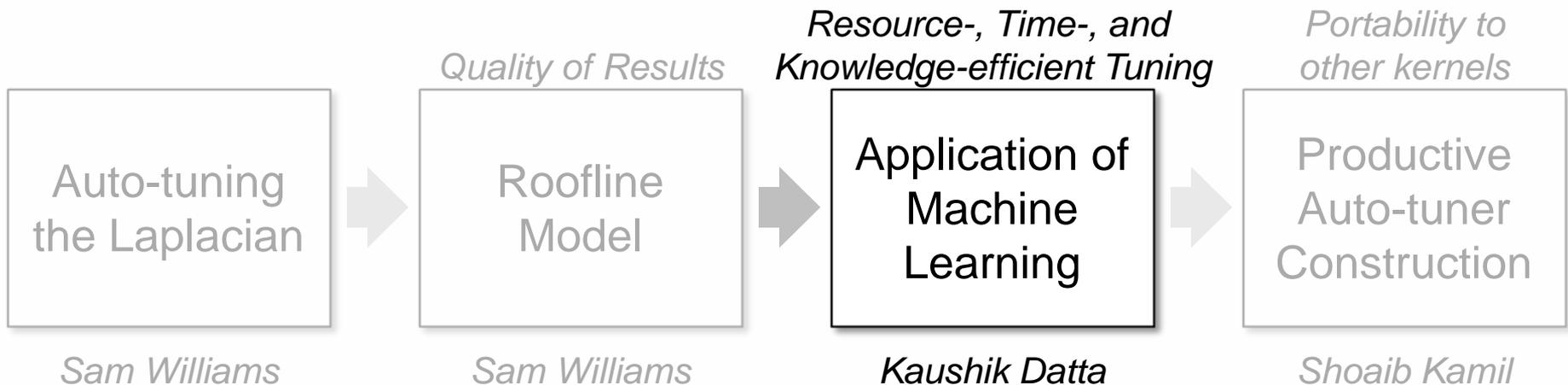
Roofline Summary

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

- ❖ Roofline clearly enumerates the performance bound.
- ❖ We observe that after auto-tuning, performance is very close to the bound
- ❖ The roofline model could be extended to the 27-point stencil, but one should implement a hierarchical model as cache bandwidth can also be a bottleneck.
- ❖ Unfortunately the biggest limitation of the model is its reliance on accurately knowing/measuring arithmetic intensity.
- ❖ Without accurate performance counters, this is extremely difficult.

III

Accelerating Tuning via Machine Learning



- ❖ For the 7-point stencil:

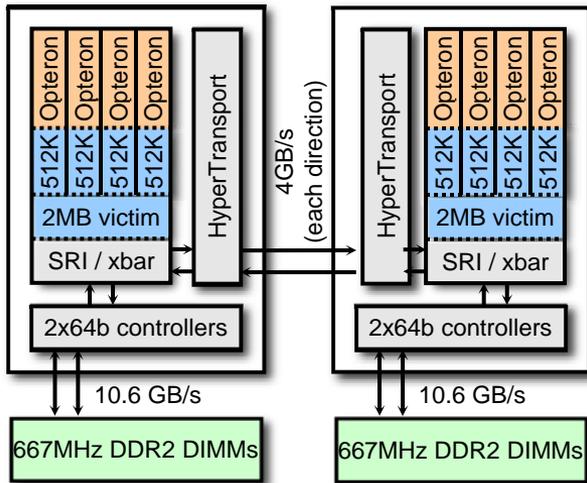
Optimization	Parameters	Total Configurations
Thread Count	1	4
Domain Decomposition	4	36
Software Prefetching	2	18
Padding	1	32
Inner Loop	8	480
Total	16	4×10^7

- ❖ There are more than *40 million* different configurations for this simple stencil auto-tuner
- ❖ This problem continues to get worse when:
 - More cores are added to a die
 - More kernels are combined to form an application
- ❖ Our previous method for traversing the search space required significant expert knowledge
- ❖ *How can non-experts search this space efficiently and effectively?*

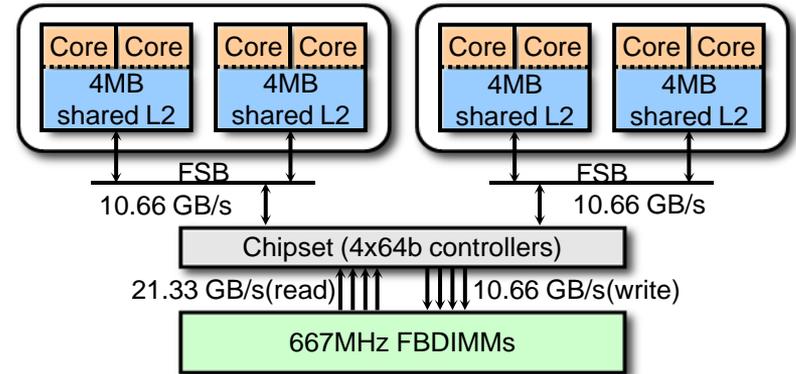


- ❖ We propose that machine learning:
 - Efficiently traverses auto-tuning's vast parameter space
 - Produces faster search to high-quality solution
 - Requires little domain knowledge
 - Allows us to handle problems with even larger search spaces

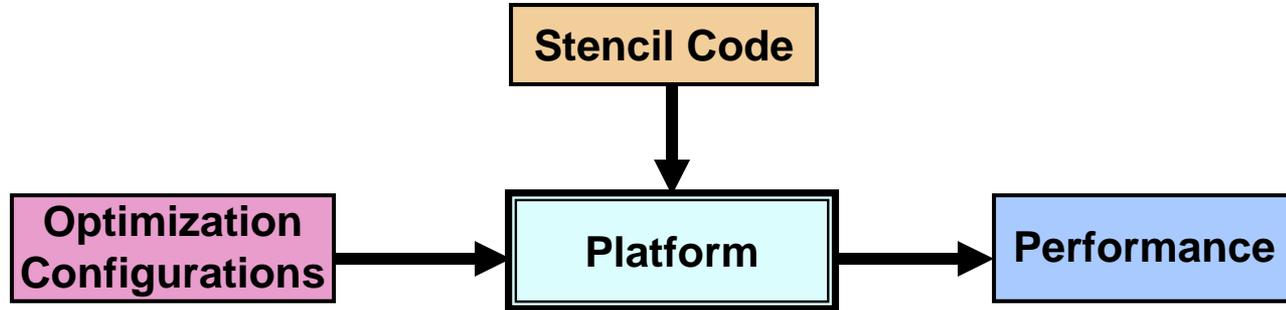
AMD Barcelona



Intel Clovertown



- Cache-based, x86 Architectures
- 2 sockets x 4 cores/socket x 1 HW thread/core
- gcc on Barcelona, icc on Clovertown
- PAPI for performance counter data



Goal: Find best value for configuration parameters to optimize performance

1. Sample 1500 data points from configuration space
2. Run stencil with chosen configurations
3. Identify relationship between optimization configurations and performance
4. Manipulate this relationship to find the best performing configuration

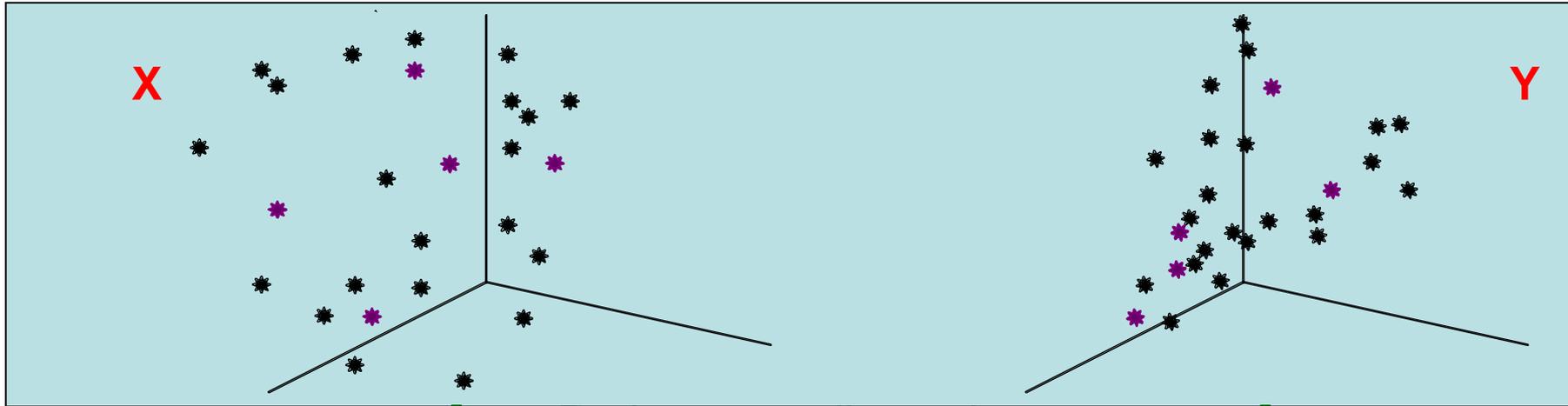
Finding Correlations

FUTURE TECHNOLOGIES GROUP

Configuration features

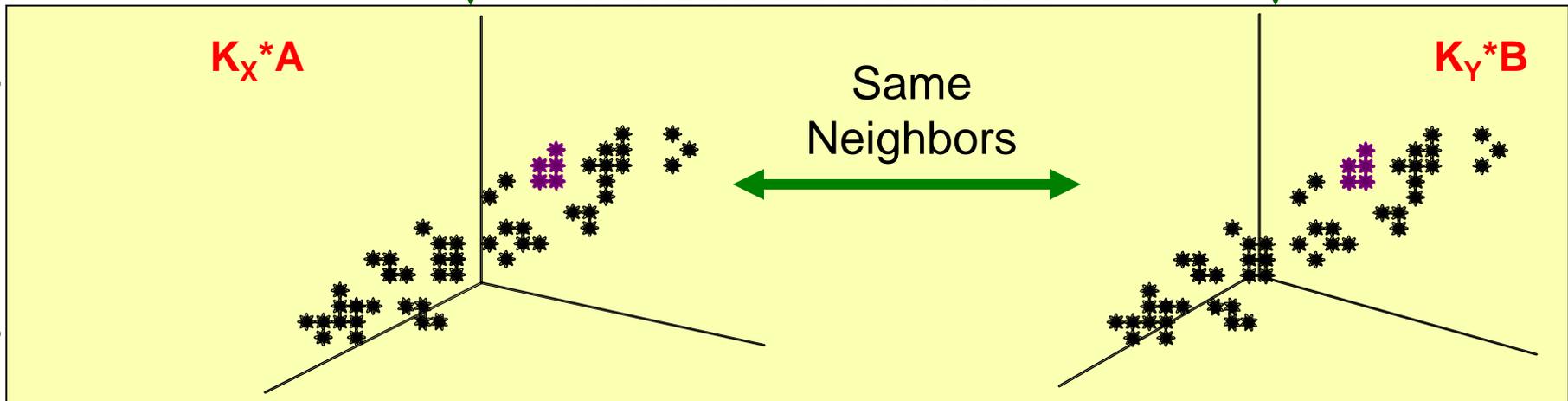
Performance Features

Raw Data Space



Project onto dimensions of maximum correlation

Projected Data Space





3 Challenges in using ML

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

1. How to represent configuration parameters and performance metrics as feature vectors
2. How to compare similarity of two feature vectors
3. How to leverage existing ML techniques to find optimal configuration



Configuration Parameters

FUTURE TECHNOLOGIES GROUP

Optimization	Parameter		Parameter Range	Number of Configurations
	Type	Name		
Thread Count	Number of Threads	<i>NThreads</i>	{2 ⁰ ...2 ³ }	4
Domain Decomposition	Block Size	<i>CX</i>	{2 ⁷ ... <i>NX</i> }	36
		<i>CY</i>	{2 ¹ ... <i>NY</i> }	
		<i>CZ</i>	<i>NZ</i>	
	Chunk Size		{1... $\frac{NX \times NY \times NZ}{CX \times CY \times CZ \times NThreads}$ }	
Software Prefetching	Prefetching Type		{register block, plane, pencil}	3
	Prefetching Distance		{0, 2 ⁵ ...2 ⁹ }	6
Padding	Padding Size		{0...31}	32
Inner Loop Optimizations	Register Block Size	<i>RX</i>	{2 ⁰ ...2 ¹ }	10
		<i>RY</i>	{2 ⁰ ...2 ¹ }	
		<i>RZ</i>	{2 ⁰ ...2 ³ }	
	Statement Type		{complete, individual}	2
	Read From Type		{array, variable}	2
	Pointer Type		{fixed, moving}	2
	Neighbor Index Type		{register block, plane, pencil}	3
	FMA-like Instructions		{yes, no}	2

Feature Vector~~x~~

Threads	Block Size CX	Block Size CY	Block Size CZ	Padding Size	Prefetch type	Prefetch distance	Statement Type
4	32	128	256	32	Plane	64	individual



Performance Metrics

FUTURE TECHNOLOGIES GROUP

Counter	Description
PAPI_TOT_CYC	Cycles per thread per job
PAPI_L1_DCM	L1 data cache misses per thread
PAPI_L2_DCM	L2 data cache misses per thread
PAPI_TLB_DCM	TLB misses per thread
PAPI_CA_SHR	Accesses to shared cache lines
PAPI_CA_CLN	Accesses to clean cache lines
PAPI_CA_ITV	Cache interventions
Power meter	Watts consumed

(total cycles * # of flops) / (clk rate * # of watts)

Feature Vector **Y**

Total Cycles	L1_DCM	L2_DCM	TLB_DCM	CA_SHR	CA_CLN	CA_ITV	Energy Efficiency
1.9E7	2.4E5	1.5E5	1.2E4	1.2E5	1.4E4	1.2E3	2.3E4



3 Challenges in using ML

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

1. How to represent configuration parameters and performance metrics as feature vectors
2. How to compare similarity of two feature vectors
3. How to leverage existing ML techniques to find optimal configuration

Kernel functions

FUTURE TECHNOLOGIES GROUP

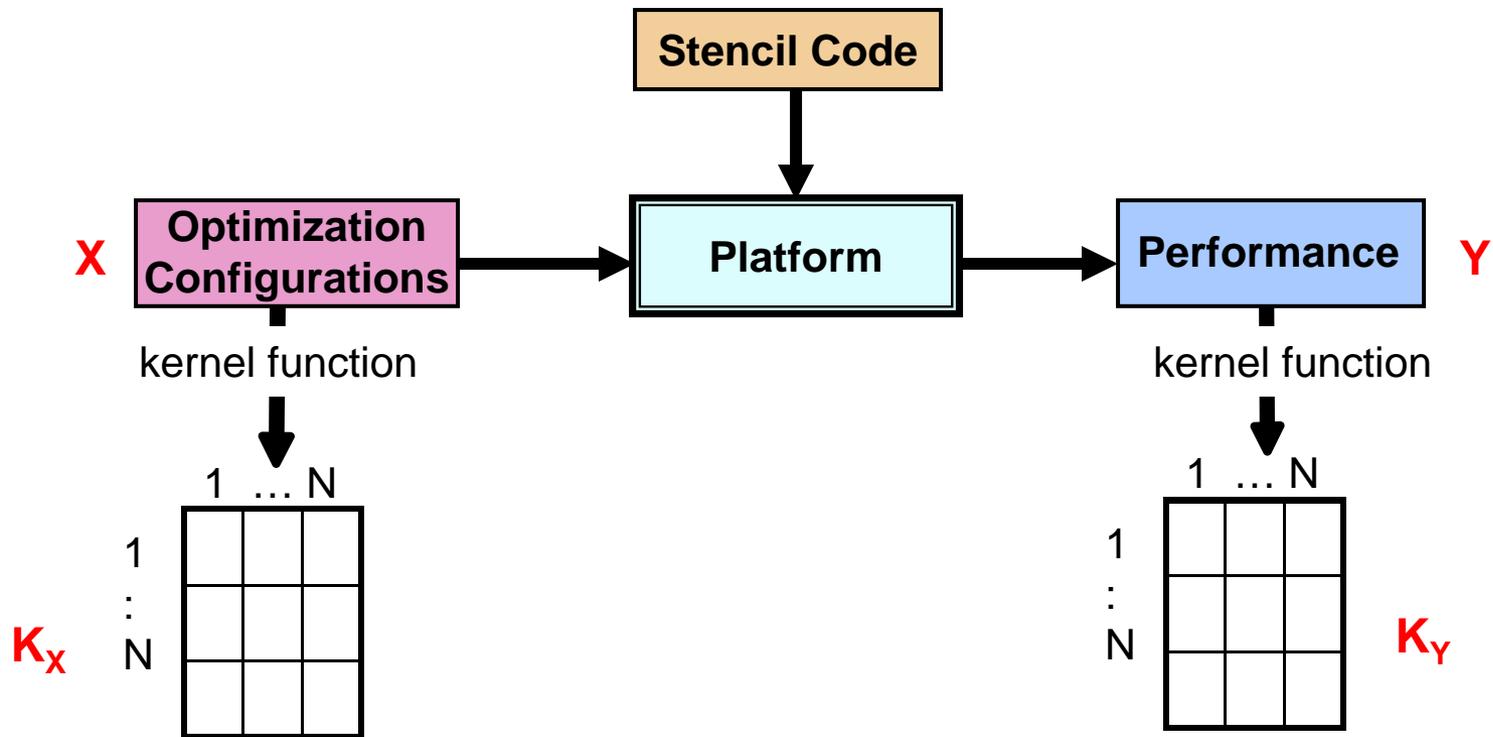
Threads	Block Size CX	Block Size CY	Block Size CZ	Padding Size	Prefetch type	Prefetch distance	Statement Type
4	32	128	256	32	Plane	64	Individual
2	64	128	256	32	Plane	64	Individual
2	64	128	256	32	Pencil	64	complete

X1
X2
X3

- ❖ Are X1 and X2 more similar than X2 and X3?
- ❖ Euclidian distance does not suffice
 - Non-numeric data
- ❖ Custom measure of similarity
 - Numeric Columns: Gaussian Kernel

$$K(x_i, x_j) = \exp(-\|x_i - x_j\|^2 / \tau)$$
 - Non-numeric Columns:

$$K(x_i, x_j) = \{1 \text{ if } x_i = x_j, 0 \text{ if } x_i \neq x_j\}$$
- ❖ Similarity(X1, X2) = average(K(X1_i, X2_i))

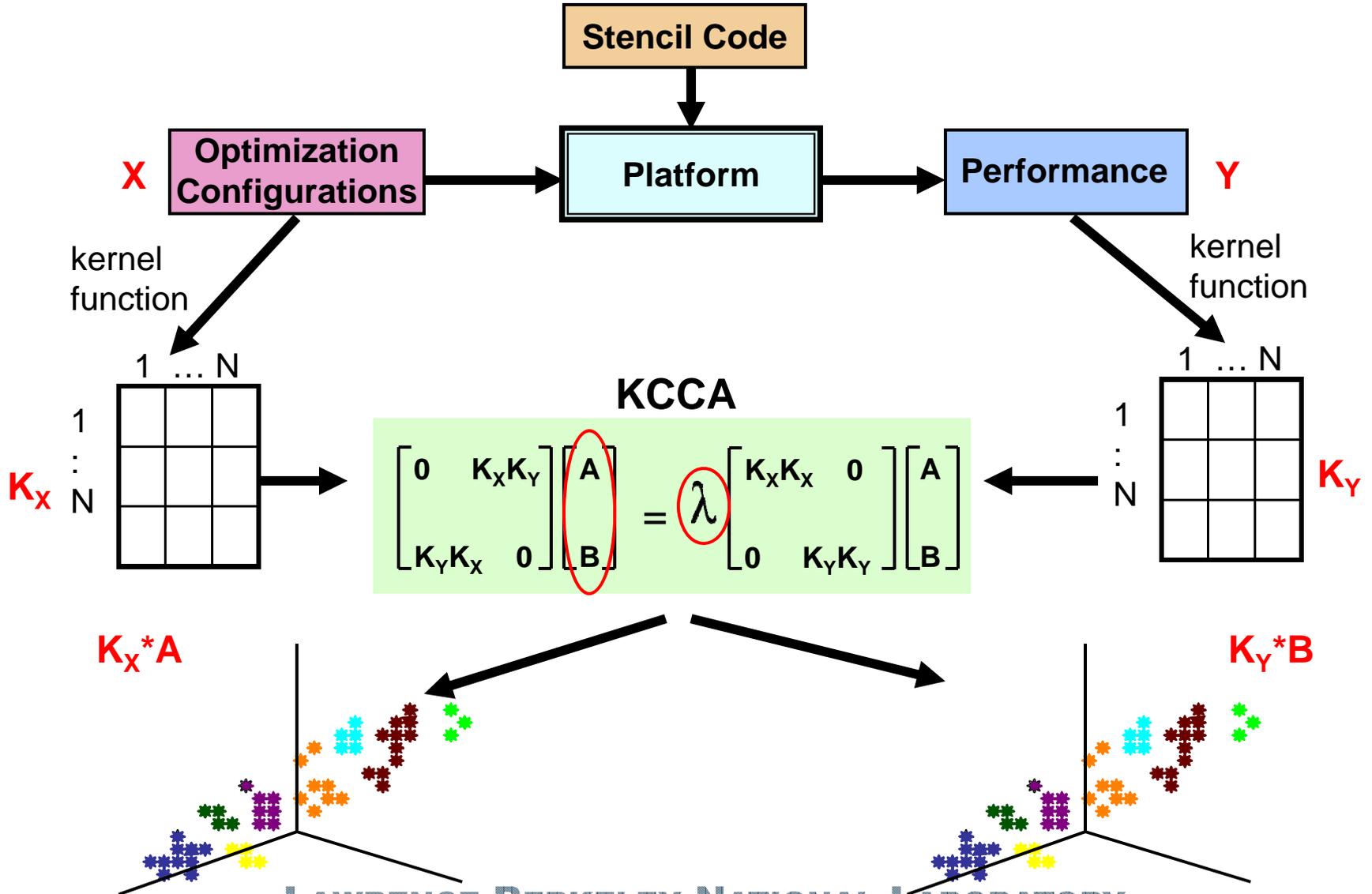




3 Challenges in using ML

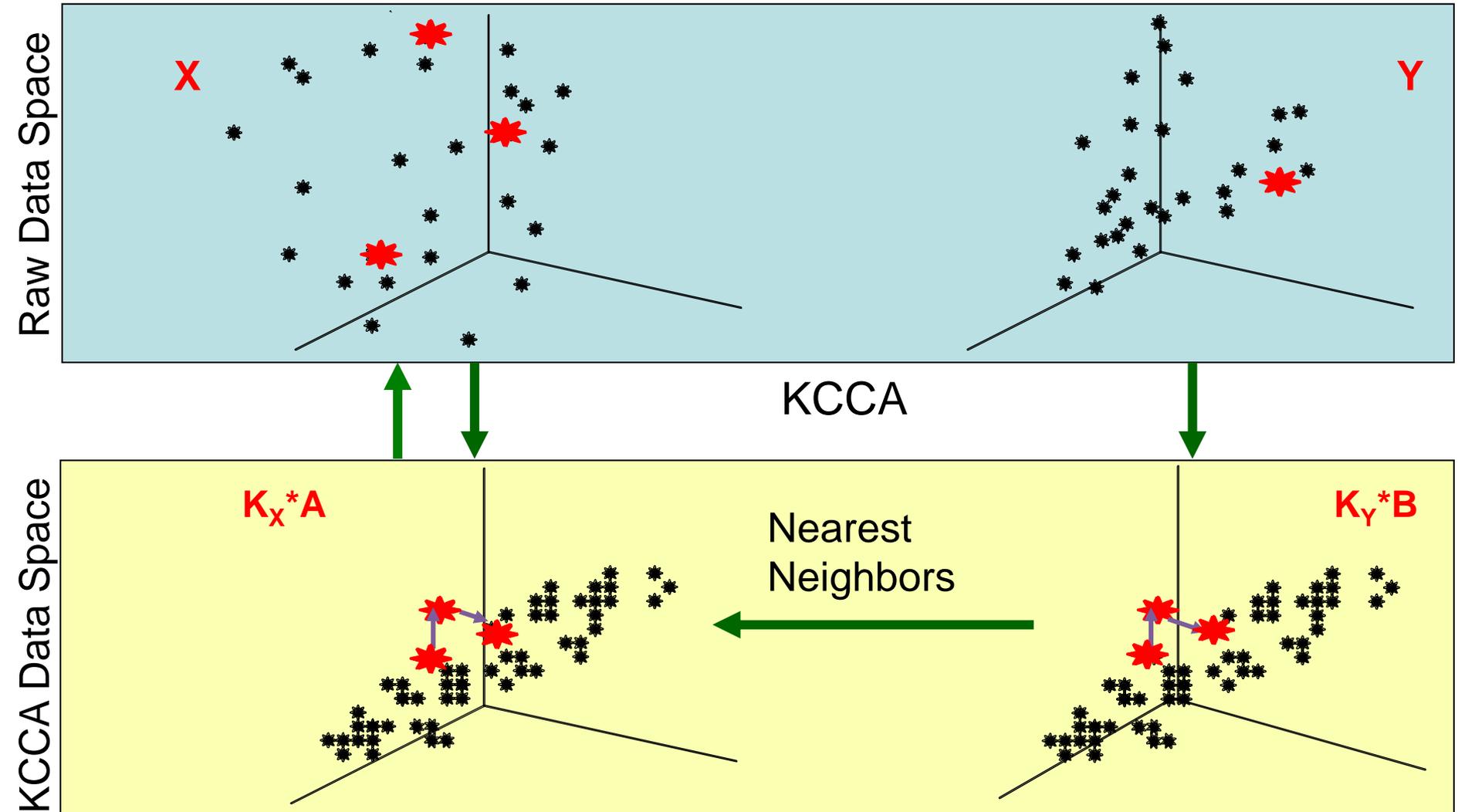
F U T U R E T E C H N O L O G I E S G R O U P

1. How to represent configuration parameters and performance metrics as feature vectors
2. How to compare similarity of two feature vectors
3. How to leverage existing ML techniques to find optimal configuration

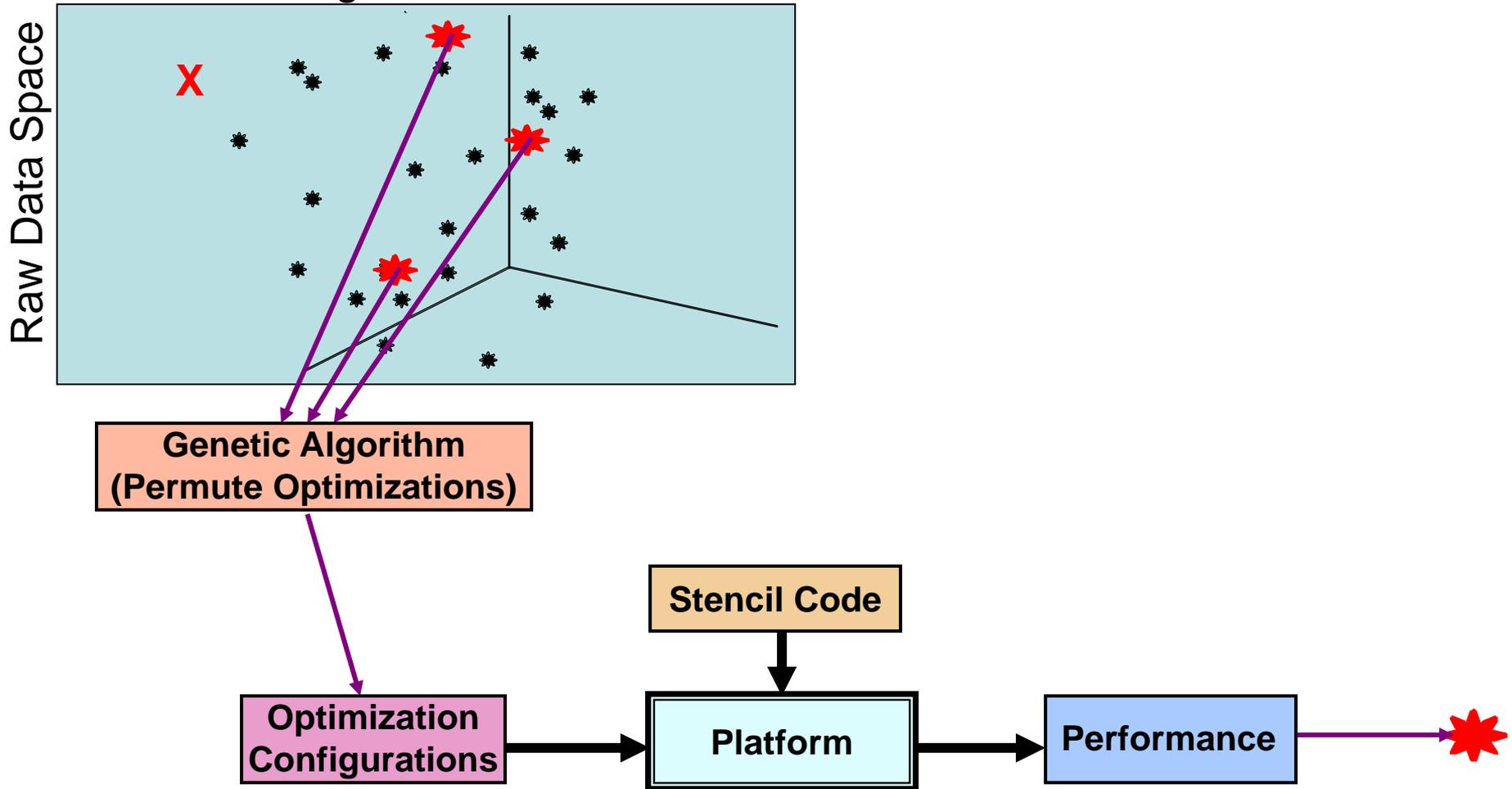


Configuration features

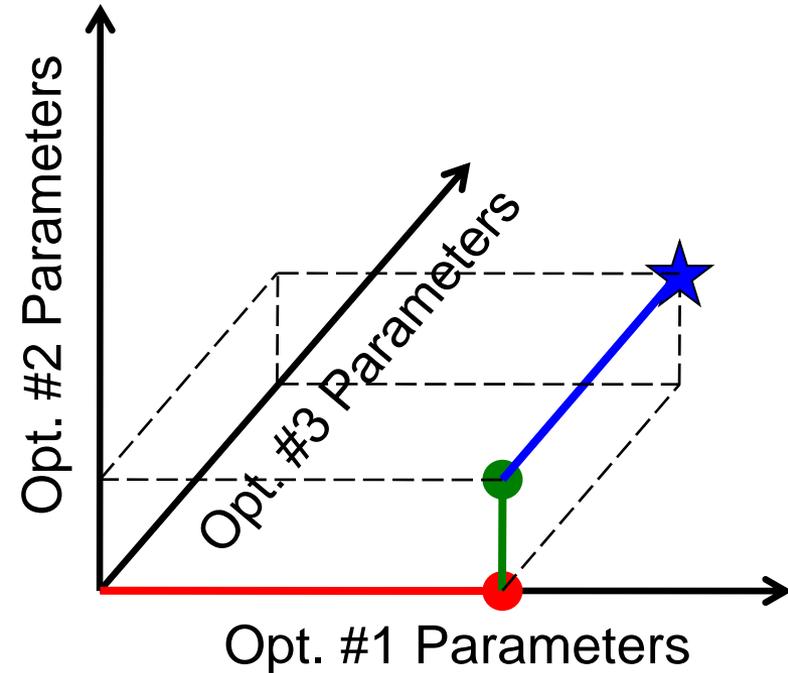
Performance Metrics



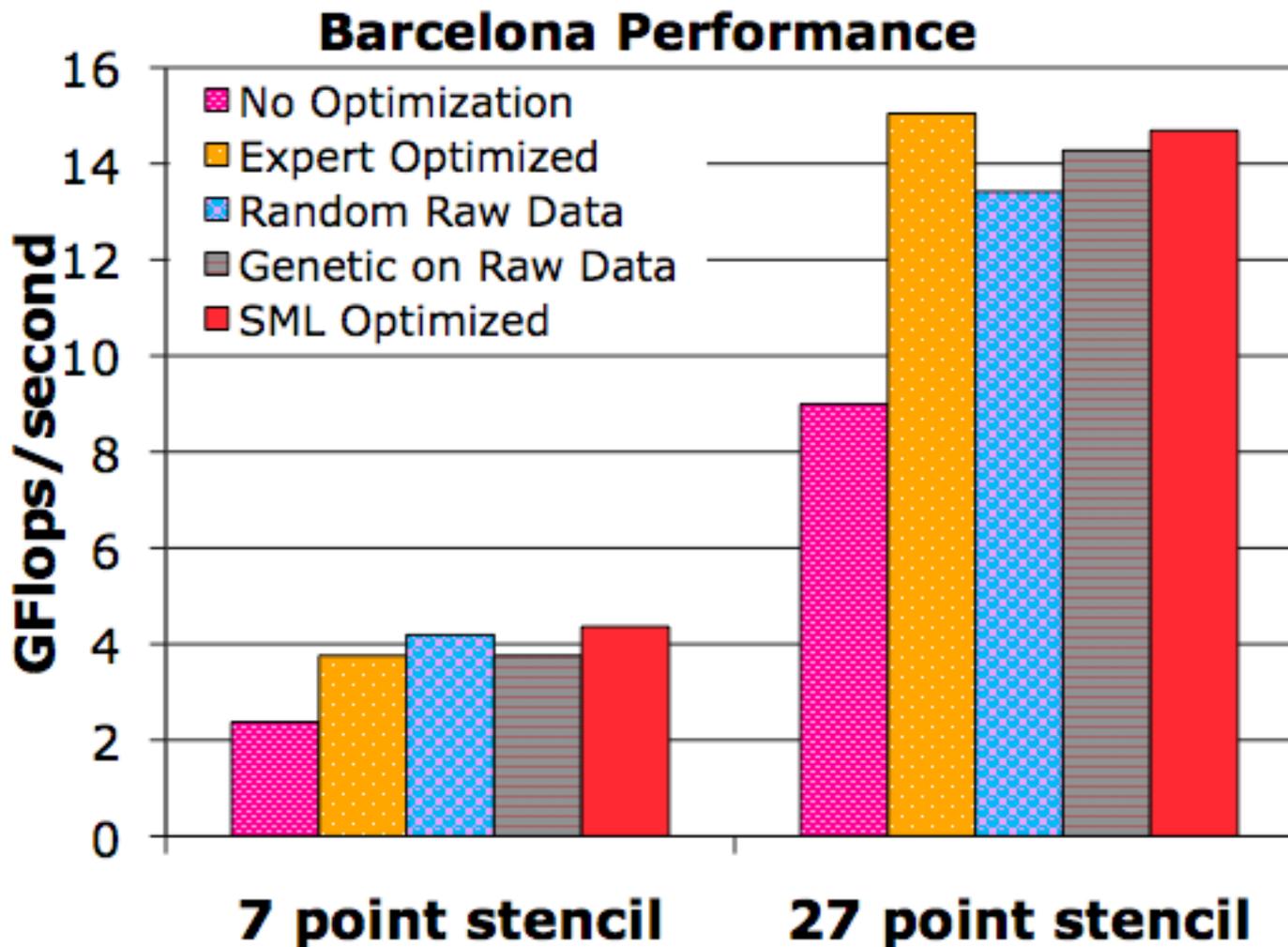
Configuration features

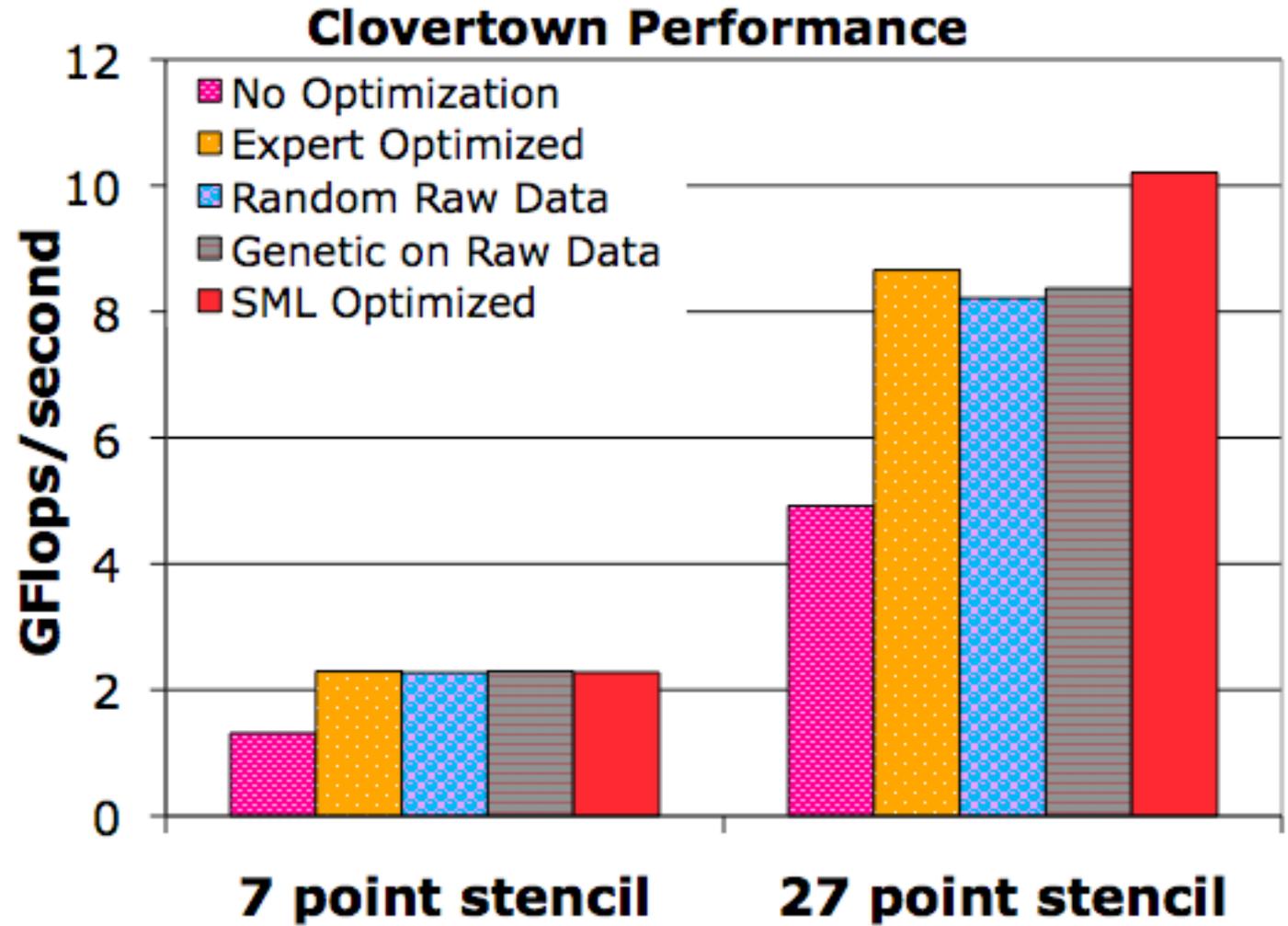


- ❖ “*No Optimization*”: running naïve code
- ❖ “*Expert Optimized*”:
 - ordered the optimizations
 - applied them consecutively
 - this was explained previously



- ❖ “*Random Raw Data*”: best performing point in raw data
- ❖ “*Genetic on Raw Data*”: permute configs for top three best performing points in raw data







Auto-tuning Time

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

- ❖ Exhaustive Search:
 - 4×10^7 configs x 0.08 s/trial x 5 trials > **180 days**
- ❖ Expert Optimized:
 - Ordering optimizations: **Cannot be quantified**
 - Applying optimizations consecutively: 570 configs x 0.08 s/trial x 5 trials = **3.8 min**
- ❖ Our Technique:
 - Training data: 1500 configs x 0.08 s/trial x 5 trials = 10 minutes
 - Training time using KCCA: 90 minutes
 - Genetic algorithm: 243 configs x 0.08 s/trial x 5 trials = 1.6 min
 - Total Time < **2 hrs**
- ❖ Our performance results are up to 18% faster than expert-optimized!



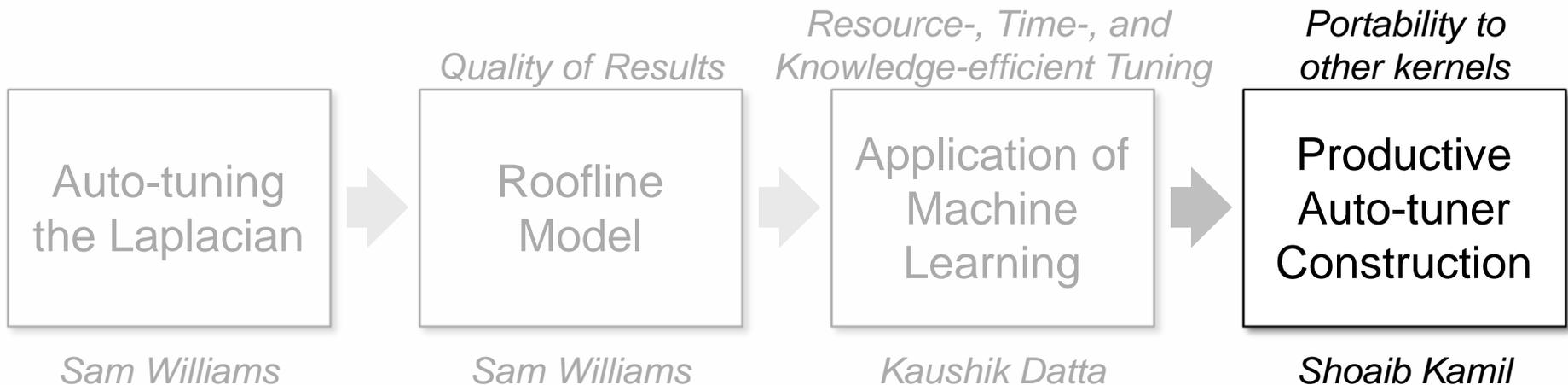
Machine Learning Future Work

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

- ❖ Reduce the model training time (currently performed on a laptop)
- ❖ Try other architectures
- ❖ Try other motifs (e.g. sparse matrices)
- ❖ Expand the search space:
 - Tuning optimization parameters AND compiler flags
 - Tuning for multiple metrics of merit
 - Tuning for composition of multiple kernels
- ❖ ML:
 - KCCA + kernel regression + gradient descent to find optimal configurations

IV

Productive Construction of Domain-Specific Auto-tuners





Problem

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

- ❖ So far have built *kernel-specific* auto-tuners
- ❖ For each kernel, requires PhD-level architectural- and domain-knowledge
- ❖ For each kernel, requires *months or years* of development effort

- ❖ Can we productively build auto-tuners for a *class* of kernels: provide performance portability across architectures *and* kernels

- ❖ Unlike dense linear algebra, many different variants of stencils
 - variants in data structure
 - variants in grid topology
 - etc

- ❖ Better solution: represent stencils in an abstract representation
 - based on the application's implementation
 - basis for *code generation* and *transformation*
 - express optimizations as code transformations



Proof-of-Concept Auto-tuner

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

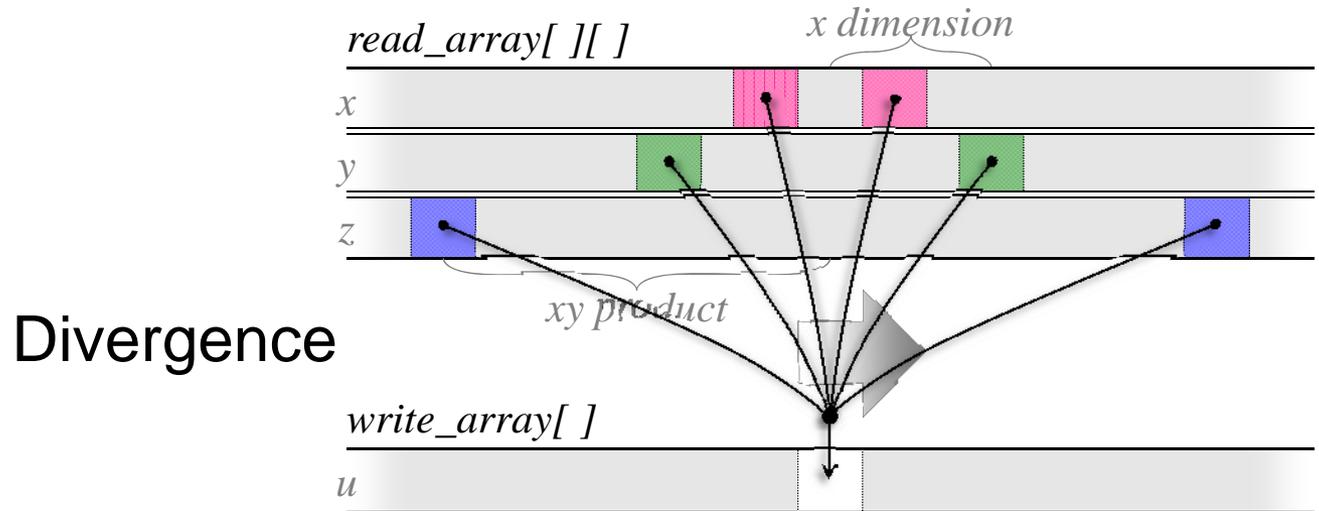
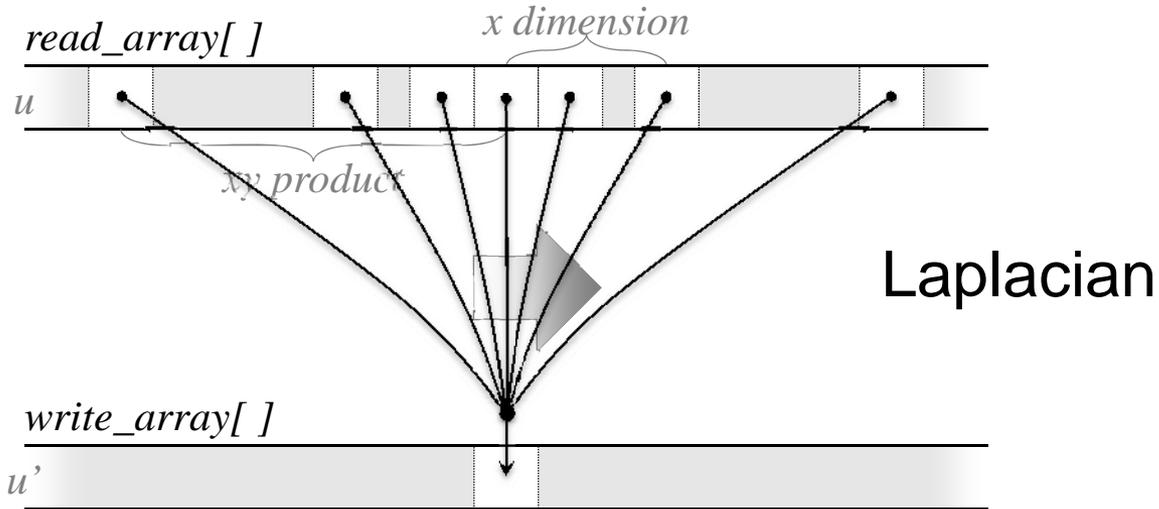
- ❖ Built a proof-of-concept auto-tuner for stencil kernels

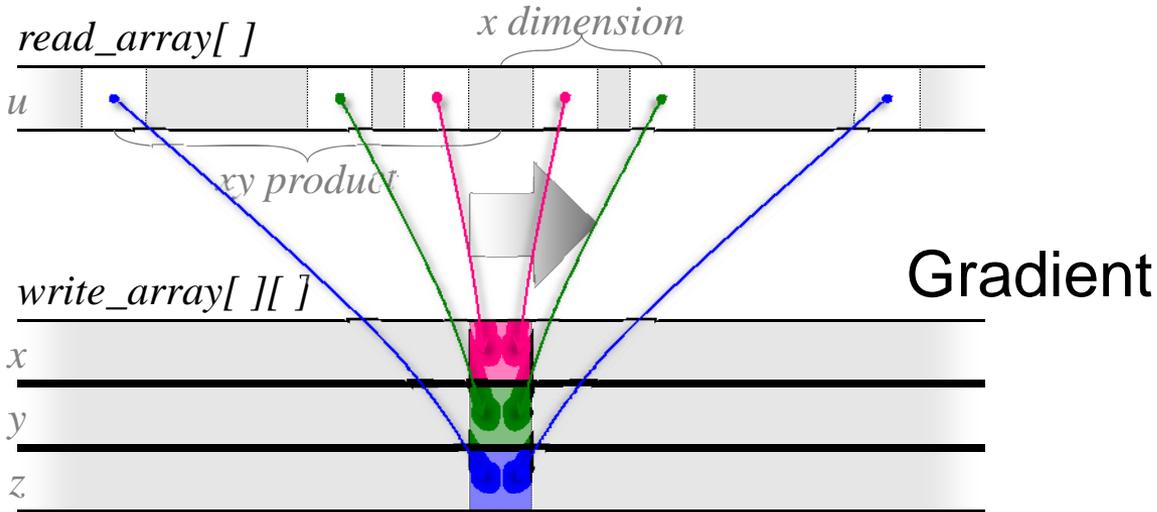
Given a Fortran application with stencil kernels:

1. Programmer annotates stencil loops
2. Auto-tuning system automatically
 - converts stencils into internal representation
 - generate candidate versions of stencil & test harness
 - discover best implementation of each stencil
 - produce library of best implementations
3. Programmer updates application to call optimized library

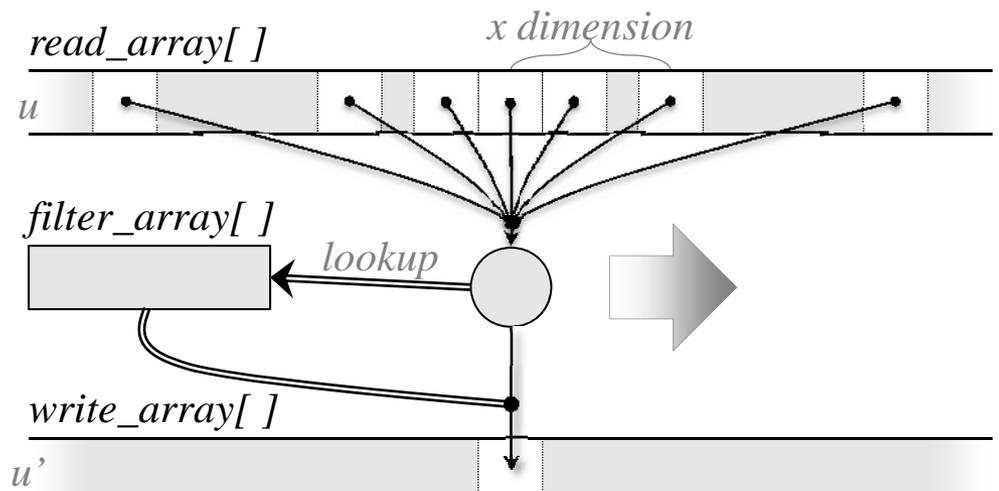
Benchmark Stencils

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





Bilateral Filter





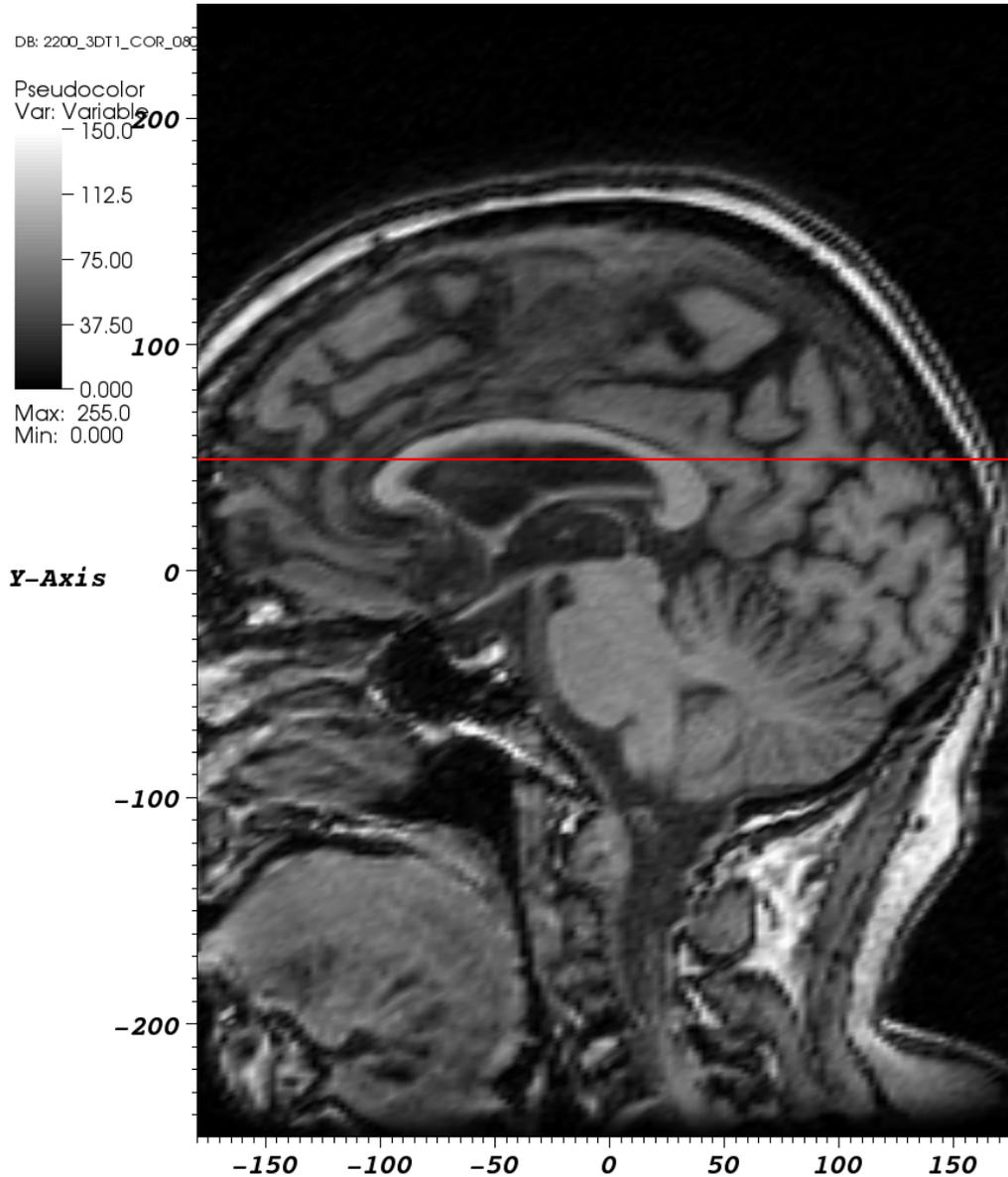
3D Bilateral Filtering

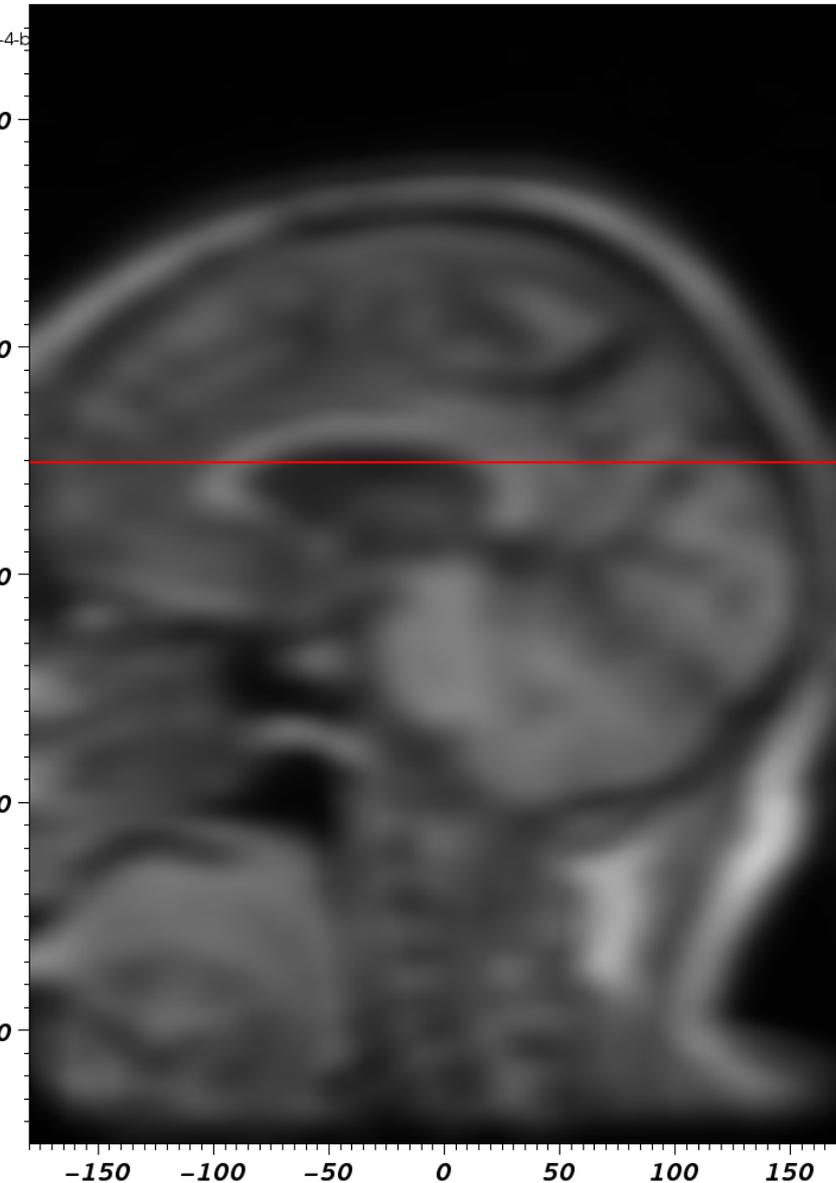
F U T U R E T E C H N O L O G I E S G R O U P

- ❖ MRI images need edge-preserving smoothing to remove artifacts from instruments
- ❖ Normal Gaussian filters **smooth images**, but **destroy sharp edges**.
- ❖ This kernel performs **anisotropic** filtering thus preserving edges.
- ❖ For each neighboring voxel, must lookup filter value based on photometric difference between center point and neighbor

Studied Kernels

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





DB: smooth-bilat-4-25-1

Pseudocolor

Var: Variable

200

112.5

75.00

37.50

100

0.000

Max: 235.0

Min: 0.000

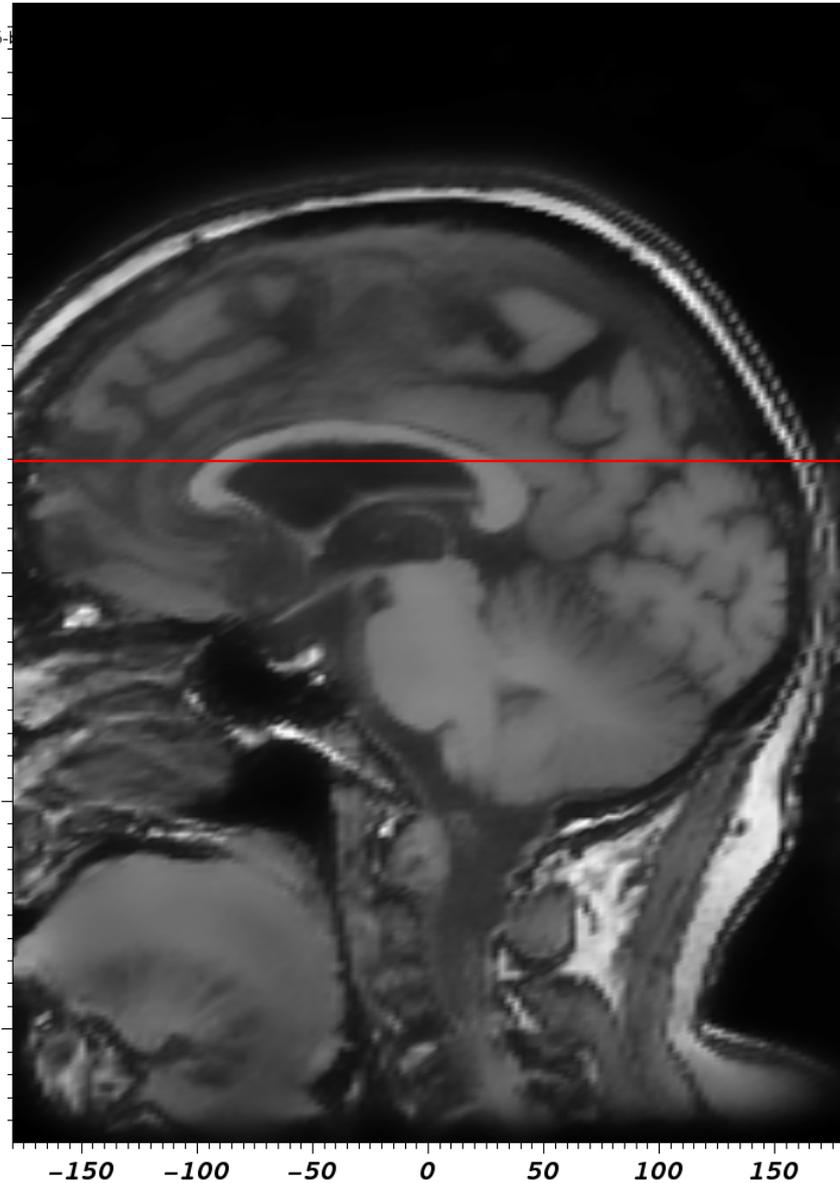
0

Y-Axis

0

-100

-200



serial
reference

Original code in Fortran

Auto-
parallelization

Auto-parallelized using the stencil framework (no tuning)

Auto-NUMA

Auto-parallelized **plus NUMA optimization**

Auto-tuning

Auto-tuned and auto-parallelized using the stencil framework

STREAM
Predicted

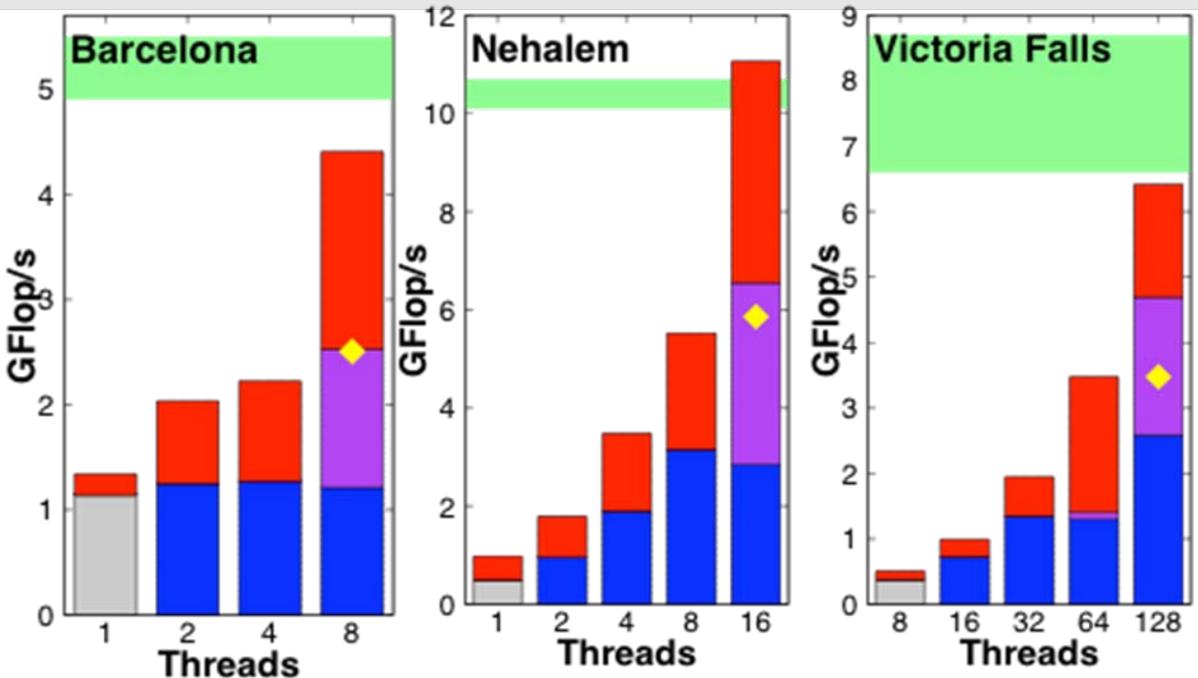
Memory-bound **performance predicted using OpenMP STREAM** benchmark

 OpenMP
Comparison

Performance of a NUMA-aware auto-parallelized with **OpenMP version of the original code**

Laplacian Results

FUTURE TECHNOLOGIES GROUP

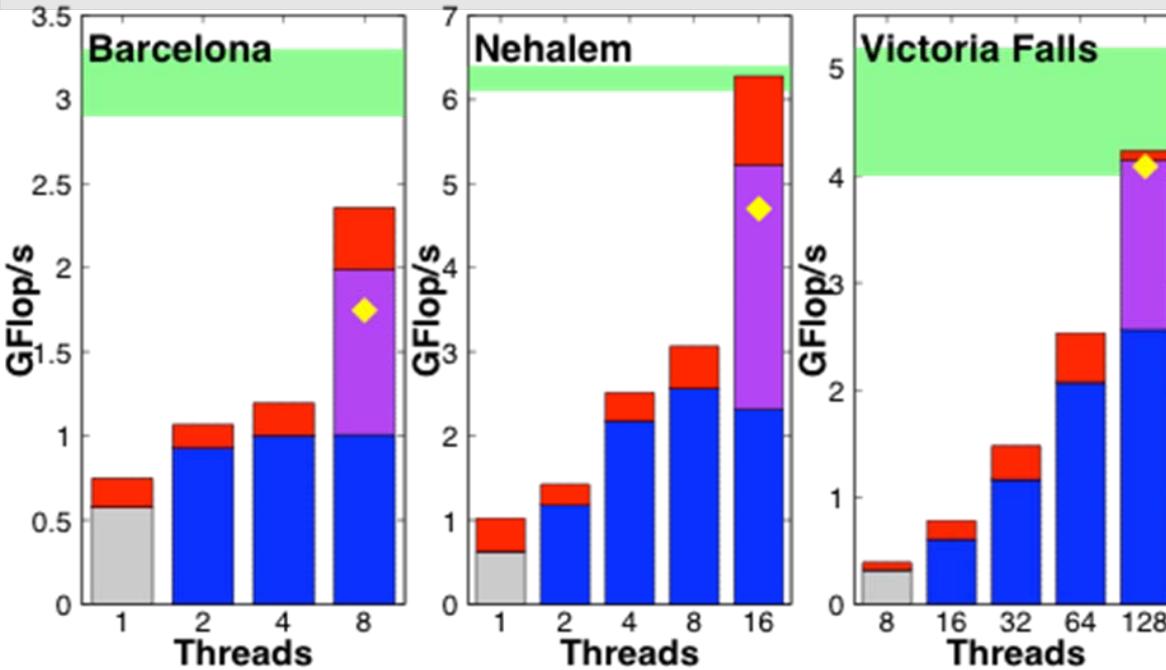


- Auto-tuning
- OpenMP Comparison
- Auto-NumA
- STREAM Predicted
- Auto-parallelization
- serial reference

- Auto-parallelization by itself does not scale well on CPUs
 - requires NUMA-aware alloc to get decent performance
 - our auto-parallelizer gets equal or better performance than OpenMP
- Overall speedups of up to 22x on Nehalem (vs. serial reference)

Divergence Results

FUTURE TECHNOLOGIES GROUP



Auto-tuning

OpenMP Comparison

Auto-NUMA

STREAM Predicted

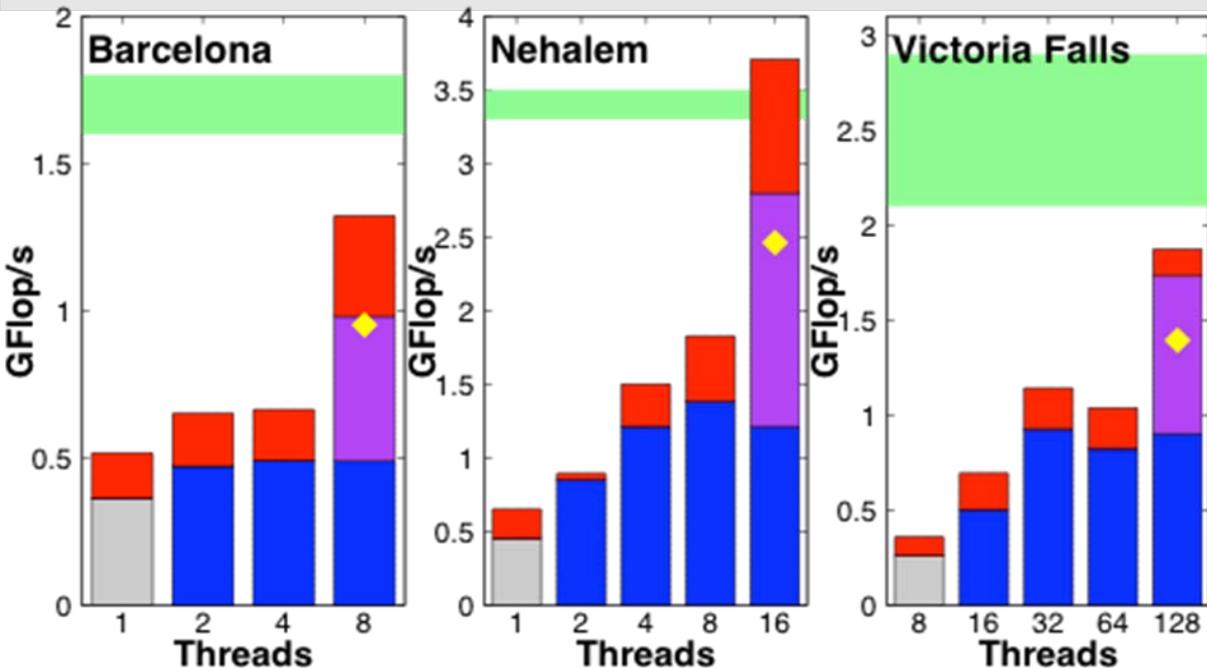
Auto-parallelization

serial reference

- Less benefit from auto-tuning on cache-based architectures here
 - As we expect based on arithmetic intensity
- Overall speedups of up to 13x on Victoria Falls

Gradient Results

FUTURE TECHNOLOGIES GROUP



Auto-tuning



OpenMP Comparison

Auto-Numa

STREAM Predicted

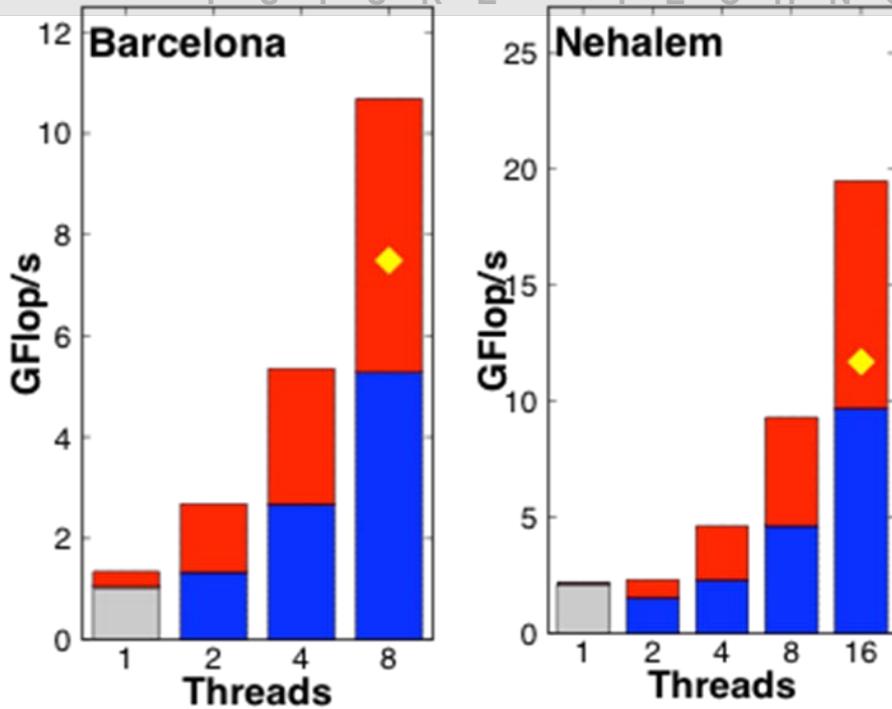
Auto-parallelization

serial reference

- Heavily memory-bound, so architectures with high memory BW get higher performance
- Overall speedups of up to 8.1x on Nehalem

Bilateral Filter Results (r=3)

FUTURE TECHNOLOGIES GROUP



- Auto-tuning
- Auto-NUMA
- Auto-parallelization
- serial reference



OpenMP Comparison

- Heavily compute-bound, plus lookup for filter weights
 - Most of auto-tuning benefit comes from better innermost-loop
- Overall speedups of 14.8x for Barcelona, 20.7x for Nehalem
- Near linear speedup as cores increase



Productivity Results

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

- ❖ Building this auto-tuning system
 - similar in terms of time & size (LoC) to single-kernel tuners
 - but embeds knowledge gained from single-kernel tuners into an auto-tuning system

- ❖ Running the auto-tuning system
 - few seconds to generate candidate versions
 - few hours to find best implementation
 - with better search (as covered in Kaushik's talk) this could be reduced dramatically
 - simple enough for application developers to use: no specialized understanding of the architecture required



Summary

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

- ❖ Proof-of-concept shows we can build an auto-tuner for a class of stencil kernels
- ❖ Performance is excellent: equal to single-kernel auto-tuning with the same optimizations
- ❖ Research goal: now that proof-of-concept shows it is possible
 - build a usable auto-tuning system for stencils
 - add in optimizations from single-kernel auto-tuners
 - make sure components can be re-used for other auto-tuning systems or compilers
- ❖ Huge step forward for auto-tuning: users no longer need highly-specialized knowledge to auto-tune their stencil kernels



Auto-tuning Summary

- ❖ Auto-tuning has been shown to benefit a large number of scientific and high performance codes across a large number of domains
- ❖ On-going research is allowing us to expand into other domains and apply auto-tuning to novel architectures (e.g. Cell)

- ❖ Recent research progress has allowed us to
 - quantify our auto-tuning success
 - accelerate/simplify the tuning search process
 - provide a means of productive auto-tuner construction for the structured grid domain

- ❖ Requested Feedback: future research directions?
 - **outside scientific computing what are the kernels/problems of interest ?**



Acknowledgements

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

❖ Research supported by:

- DOE Office of Science under contract number DE-AC02-05CH11231
- Microsoft and Intel funding (Award #20080469)
- NSF contract CNS-0325873
- Sun Microsystems - Niagara2 / Victoria Falls donations
- Georgia Tech - access to QS22 Cell blades
- Nvidia – GTX280 donations



Questions ?



BACKUP SLIDES