Tuning Sparse Matrix Vector Multiplication for multi-core processors

Sam Williams

samw@cs.berkeley.edu

Other Contributors

- Rich Vuduc
- Lenny Oliker
- John Shalf
- Kathy Yelick
- Jim Demmel

Outline

- Introduction
- Machines / Matrices
- Initial performance
- Tuning
- Optimized Performance

Introduction

- Evaluate $y=Ax$, where x & y are dense vectors, and A is a sparse matrix.
- Sparse implies most elements are zero, and thus do not need to be stored.
- Storing just the nonzeros requires their value and meta data containing their coordinate.

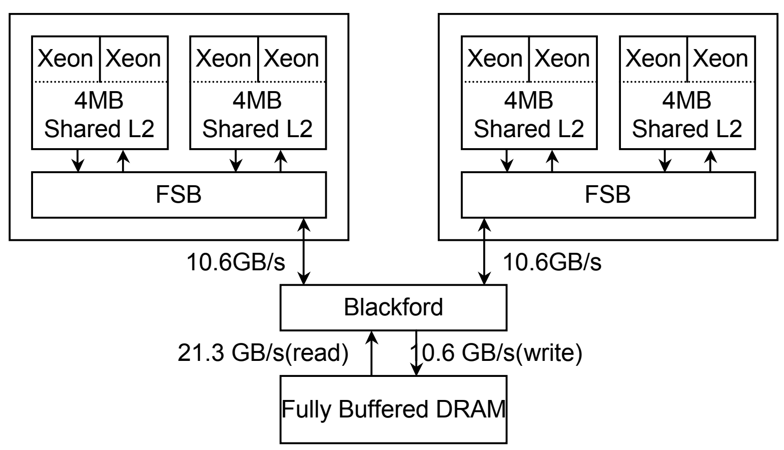
Multi-core trends

- It will be far easier to scale peak gflop/s (via multi-core) than peak GB/s (more pins/higher frequency)
- With sufficient number of cores, any low computational intensity kernel should be memory bound.
- Thus, the problems with the smallest footprint should run the fastest.
- Thus, tuning via heuristics, instead of search becomes more tractable

Which multi-core processors?

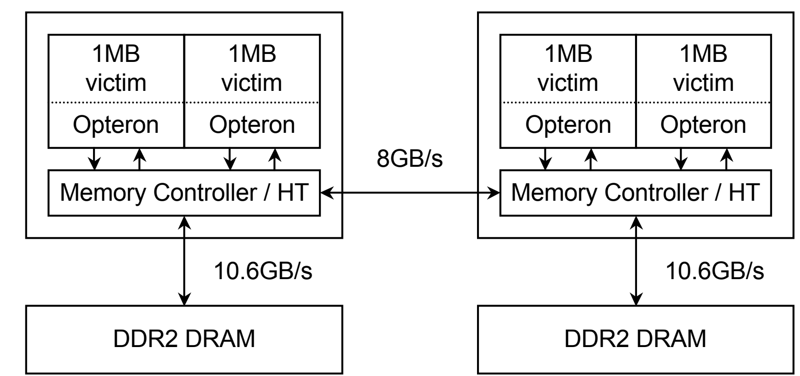
Intel Xeon(Clovertown)

- pseudo quad-core / socket
- 4-issue, out of order, super-scalar
- Fully pumped SSE
- 2.33GHz
- 21GB/s, 74.6 GFlop/s



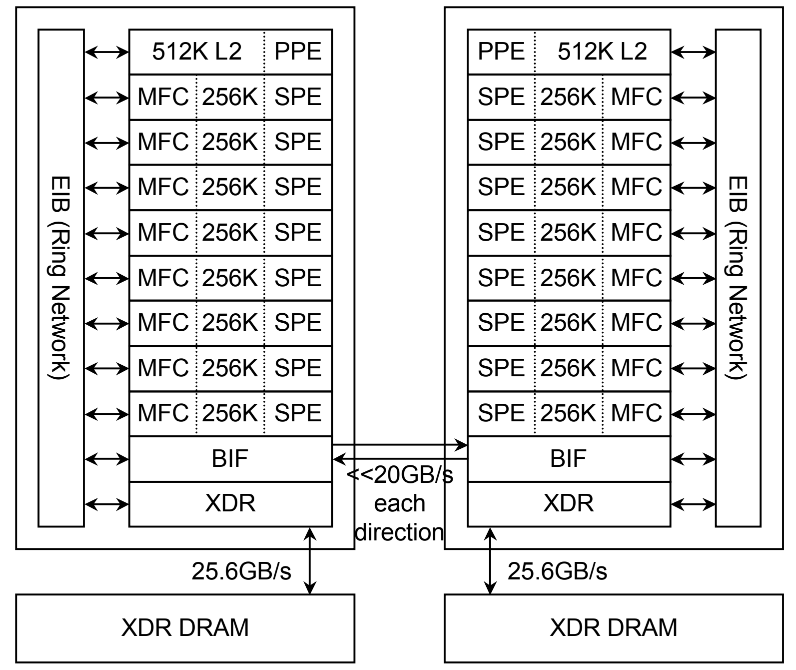
AMD Opteron

- Dual core / socket
- 3-issue, out of order, super-scalar
- Half pumped SSE
- 2.2GHz
- 21GB/s, 17.6 GFlop/s
- Strong NUMA issues



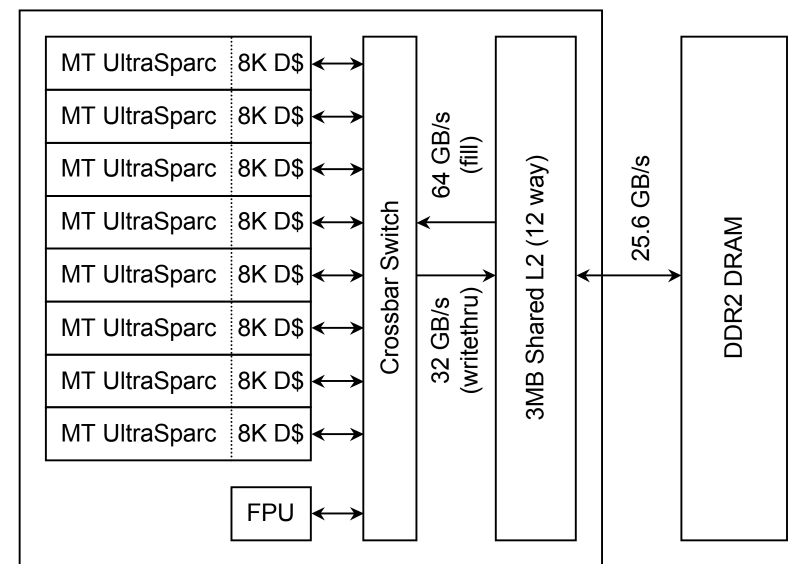
IBM Cell Blade

- Eight SPEs / socket
- Dual-issue, in-order, VLIW like
- SIMD only ISA
- Disjoint Local Store address space + DMA
- Weak DP FPU
- 51.2GB/s, 29.2GFlop/s,
- Strong NUMA issues



Sun Niagara

- Eight core
- Each core is 4 way multithreaded
- Single-issue in-order
- Shared, very slow FPU
- 1.0GHz
- 25.6GB/s, 0.1GFlop/s, (8GIPS)



64b integers on Niagara?

- To provide an interesting benchmark, Niagara was run using 64b integer arithmetic.
- This makes the peak “Gflop/s” in the ballpark of the other architectures.
- Downside: integer multiplies are not pipelined, and require 10 cycles. Increased register pressure
- Perhaps a rough approximation to Niagara2 performance and scalability

- Cell’s double precision isn’t poor enough to necessitate this work around.

Niagara2

- 1.0 -> 1.4GHz
- Pipeline redesign
- 2 thread groups/core (2x the ALUs, 2x the threads)
- FPU per core
- FBDIMM (42.6GB/s read, 21.3 write)

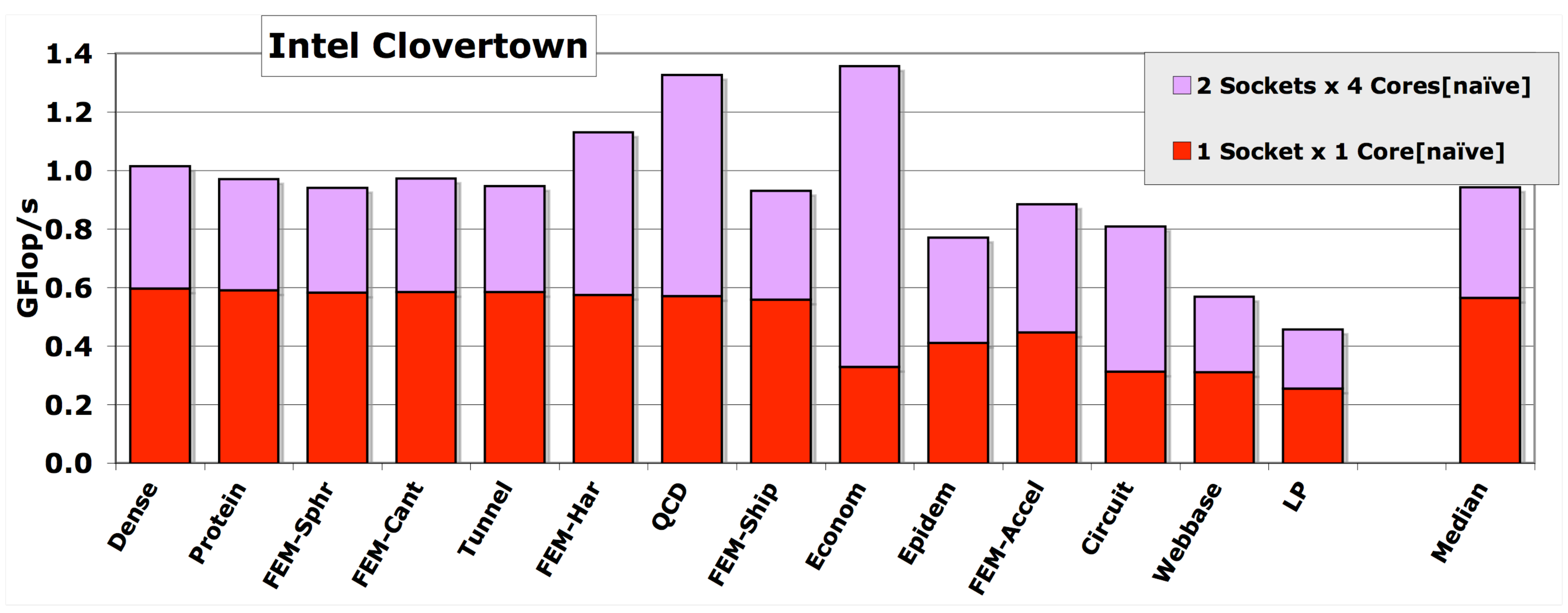
- Sun's Claims:
  - Increasing threads per core from 4 to 8 to deliver up to 64 simultaneous threads in a single Niagara 2 processor, resulting in at least 2x throughput of the current UltraSPARC T1 processor-- all within the same power and thermal envelope
  - The integration of one Floating Point Unit per core, rather than one per processor, to deliver 10X higher throughput on applications with high floating point content such as scientific, technical, simulation, and modeling programs

Which matrices?

Name	Dense	Protein	FEM / Spheres	FEM / Cantilever	Wind Tunnel	FEM / Harbor	QCD	FEM / Ship	Economics	Epidemiology	FEM / Accelerator	Circuit	webbase	LP
Spyplot														
Rows	2K	36K	83K	62K	218K	47K	49K	141K	207K	526K	121K	171K	1M	4K
Columns	2K	36K	83K	62K	218K	47K	49K	141K	207K	526K	121K	171K	1M	1.1M
Nonzeros (per row)	4.0M (2K)	4.3M (119)	6.0M (72)	4.0M (65)	11.6M (53)	2.37M (50)	1.90M (39)	3.98M (28)	1.27M (6)	2.1M (4)	2.62M (22)	959K (6)	3.1M (3)	11.3M (2825)
Description	Dense matrix in sparse format	Protein data bank 1HYS	FEM concentric spheres	FEM cantilever	Pressurized wind tunnel	3D CFD of Charleston harbor	Quark propagators (QCD/LGT)	FEM Ship section/detail	Macroeconomic model	2D Markov model of epidemic	Accelerator cavity design	Motorola circuit simulation	Web connectivity matrix	Railways set cover Constraint matrix

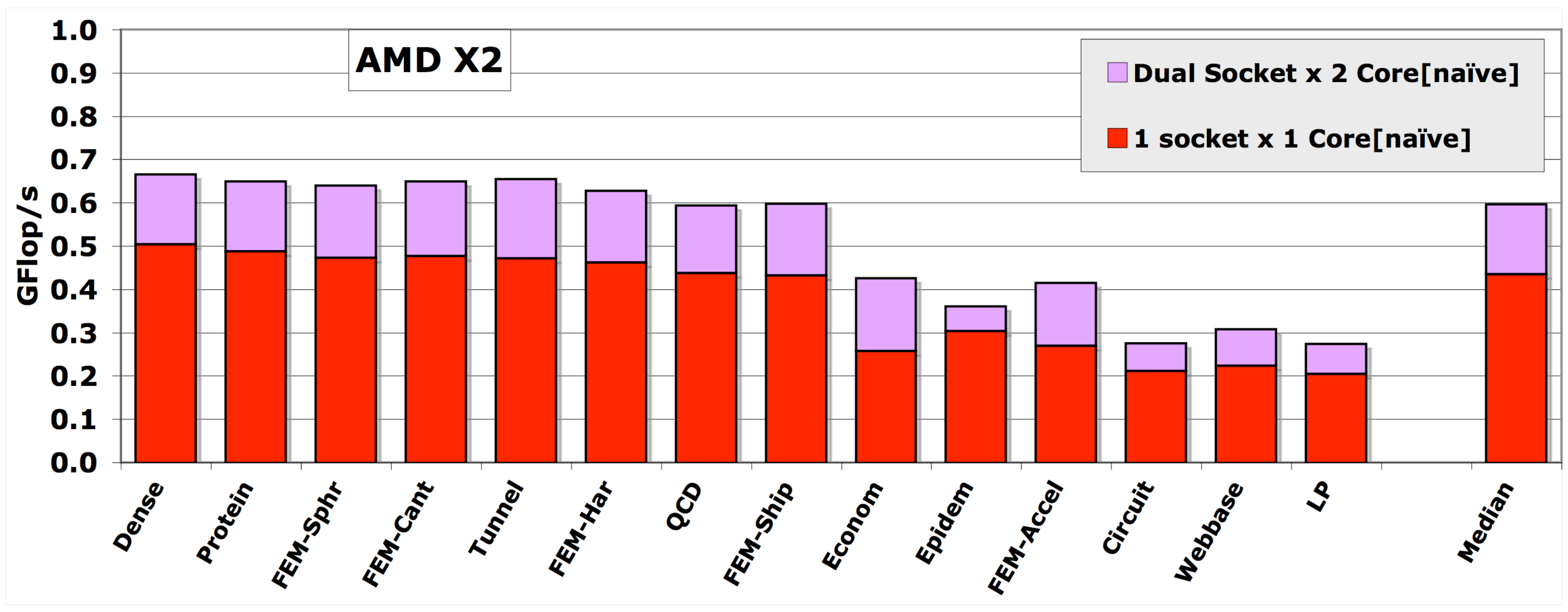
Un-tuned Serial & Parallel Performance

Intel Clovertown



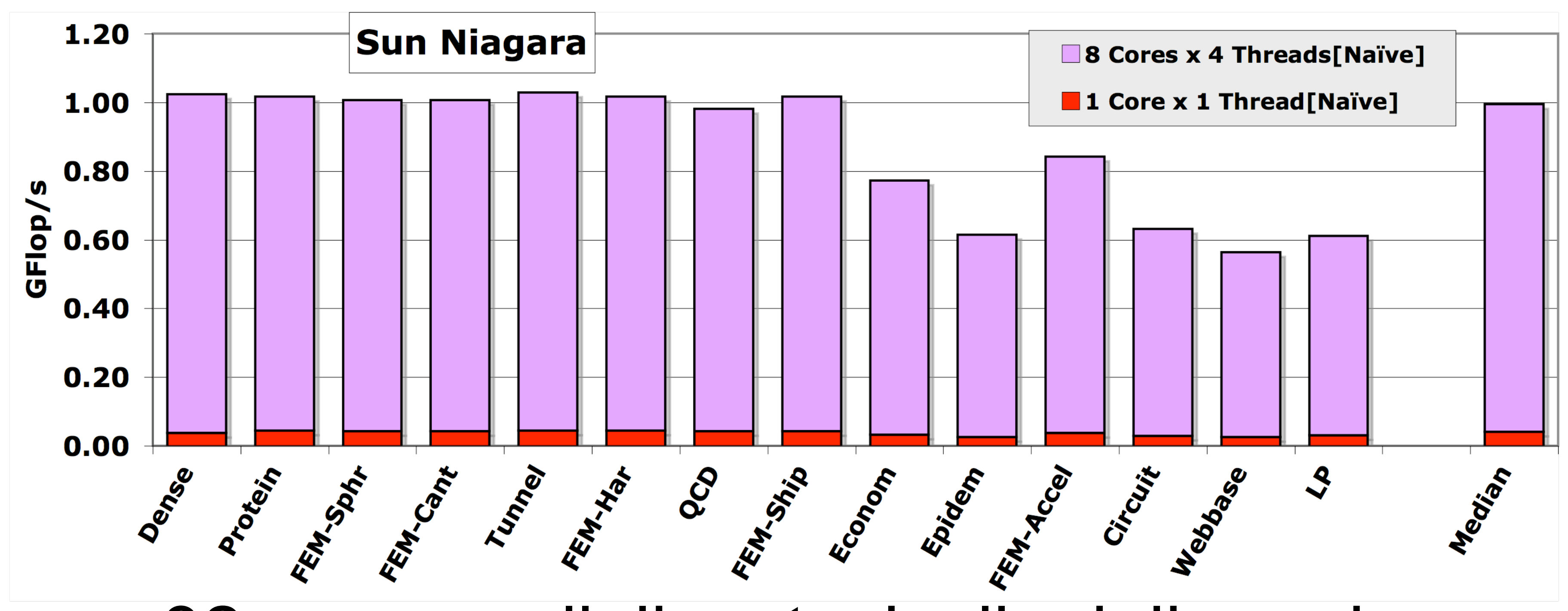
- 8 way parallelism typically delivered only 66% improvement

AMD Opteron



- 4 way parallelism typically delivered only 40% improvement in performance

Sun Niagara



- 32 way parallelism typically delivered 23x performance improvement

Tuning

OSKI & PETSc

- OSKI is a serial auto-tuning library for sparse matrix operations
- Much of the OSKI tuning space is included here.
- For parallelism, it can be included in the PETSc parallel library using a shared memory version of MPICH
- We include these 2 points as a baseline for the x86 machines.

Exploit NUMA

- Partition the matrix into disjoint sub-matrices (thread blocks)
- Use NUMA facilities to pin both thread block and process

- x86 Linux sched_setaffinity()
  (libnuma ran slow)
- Niagara Solaris processor_bind()
- Cell libnuma - numa_set_membind()
  - numa_bind()

Processor ID to Physical ID

- The mapping of linux/solaris processor ID to physical core/thread ID was unique to each machine
- Important if you want to share cache

  or accurately benchmark single socket/core

- Opteron
- Clovertown
- Niagara

Machine					
Opteron				Socket (bit 1)	Core (bit 0)
Clovertown			Core in socket (bits 2:1)		Socket (bit 0)
Niagara		Socket (bit 6)	Core within a socket (bits 5:3)	Thread within a core (bits 2:0)	

Fast Barriers

- Pthread barrier wasn’t available on x86 machines
- Emulate it with mutexes & broadcasts (sun’s example)
- Acceptable performance with low number of threads

- Pthread barrier on Solaris doesn’t scale well & emulation is even slower

- Implement lock free barrier (thread 0 sets others free)
- Similar version on Cell (PPE sets SPEs free)

Cell Local Store Blocking

- Heuristic Approach
- Applied individually to each thread block
- Allocate ~half the local store to caching 128byte lines of the source vector and the other half for the destination
- Thus partitions a thread block into multiple blocked rows
- Each is in turn blocked by marking the unique cache lines touched expressed in stanzas
- Create a DMA list, and compress the column indices to be cache block relative.
- Limited by maximum stanza size and max number of stanzas

Cache Blocking

- Local Store blocking can be extended to caches, but you don’t get an explicit list or compressed indices
- Different than standard cache blocking for SPMV (lines spanned vs touched)
- Beneficial on some matrices

TLB Blocking

- Heuristic Approach
- Extend cache lines to pages
- Let each cache block only touch TLBSize pages ~ 31 on opteron
- Unnecessary on Cell or Niagara

Register Blocking and Format Selection

- Heuristic Approach
- Applied individually to each cache block
- re-block the entire cache block into 8x8
- Examine all 8x8 tiles and compute how many smaller power of 2 tiles they would break down into.
- e.g. how many 1x1, 2x1, 4x4, 2x8, etc…
- Combine with BCOO and BCSR to select the format x r x c that minimizes the cache block size

- Cell only used 2x1 and larger BCOO

Index Size Selection

- It is possible (always for Cell) that only 16b are required to store the column indices of a cache block
- The high bits are encoded in the coordinates of the block or the DMA list

Architecture Specific Kernels

- All optimizations to this point have been common (perhaps bounded by configurations) to all architectures.
- The kernels which process the resultant sub-matrices can be individually optimized for each architecture

SIMDization

- Cell and x86 support explicit SIMD via intrinsics
- Cell showed a significant speedup
- Opteron was no faster
- Clovertown was no faster (if the correct compiler options were used)

Loop Optimizations

- Few optimizations since the end of one row is the beginning of the next.
- Few tweaks to loop variables
- Possible to software pipeline the kernel
- Possible to implement a branchless version (Cell BCOO worked)

Software Prefetching / DMA

- On Cell, all data is loaded via DMA
- It is double buffered only for nonzeros

- On the x86 machines, a hardare prefetcher is supposed to cover the unit-stride streaming access
- We found that explicit NTA prefetches deliver a performance boost

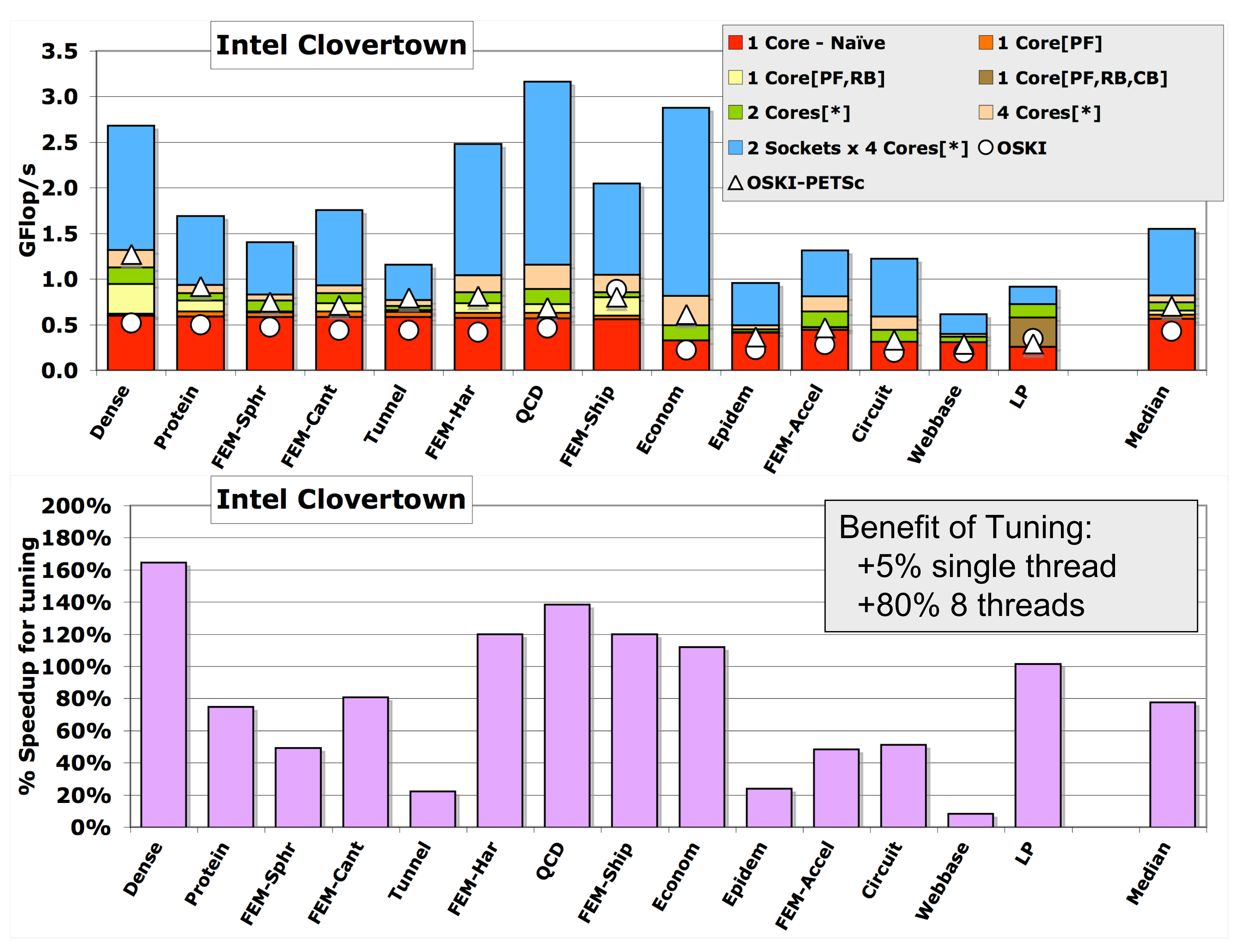
- Niagara(any version) cannot satisfy Little's law with only multi-threading
- Prefetches would be useful if performance weren't limited by other problems

Code Generation

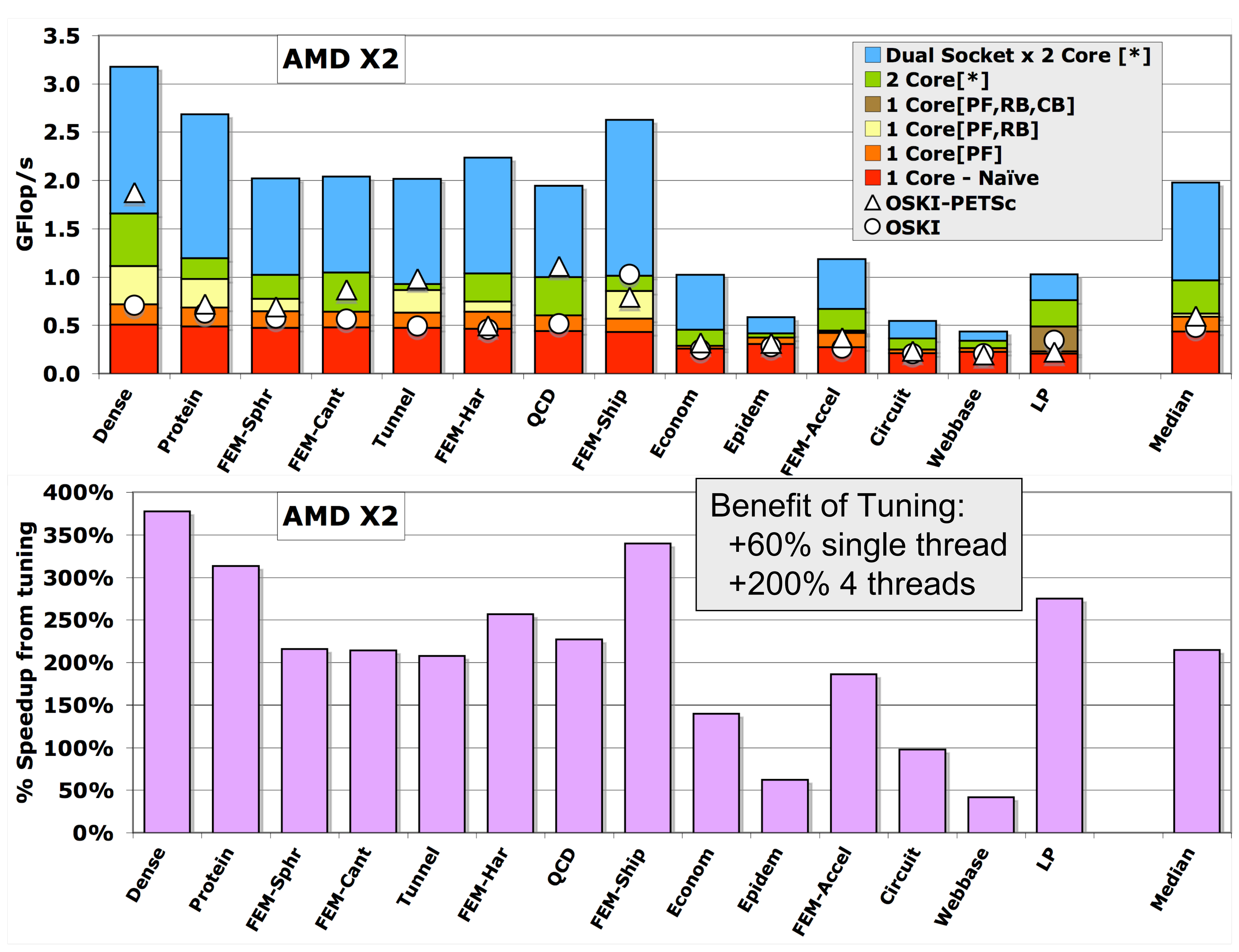
- Write a Perl script to generate all kernel variants.
- For generic C, x86/Niagara used the same generator
- Separate generator for SSE
- Separate generator for Cell's SIMD

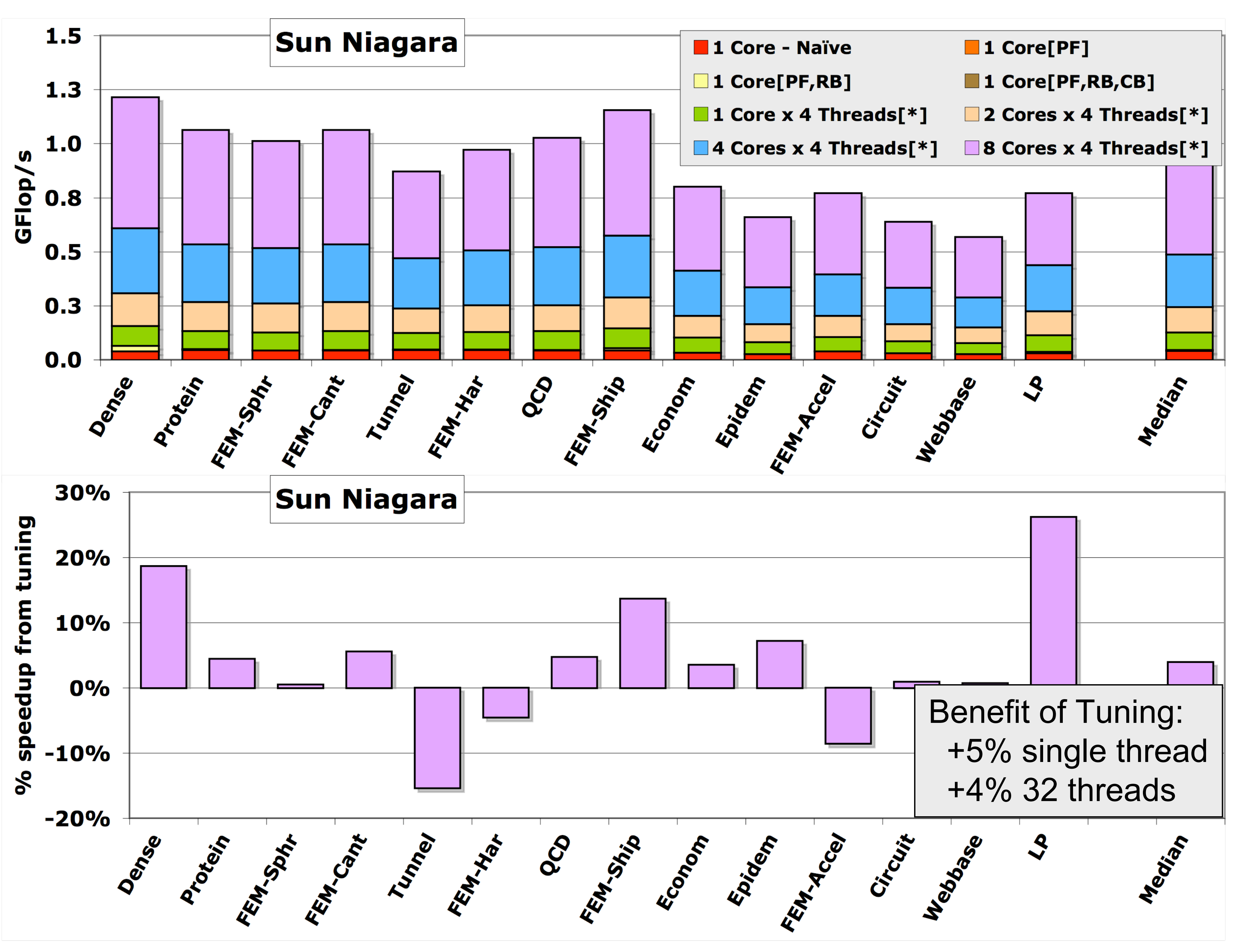
- Produce a configuration file for each architecture that limits the optimizations that can be made in the data structure, and their requisite kernels

Tuned Performance

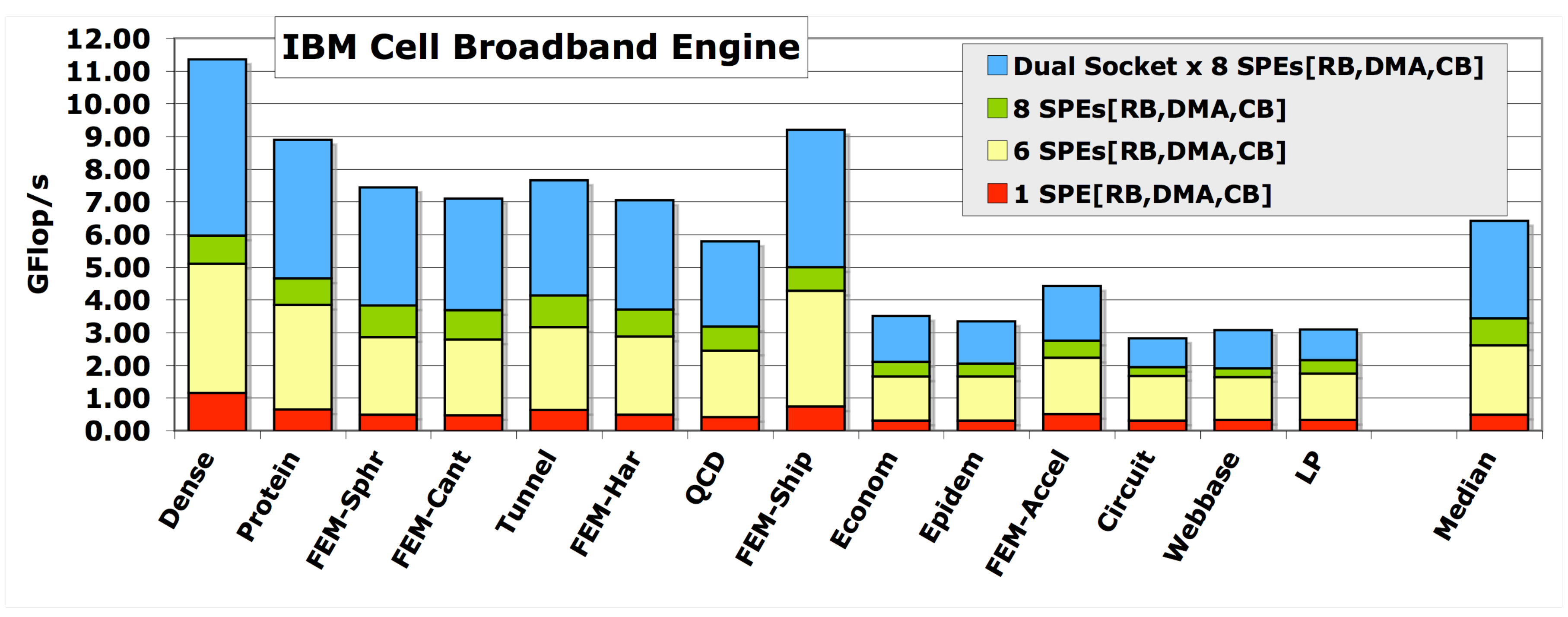


Intel Clovertown
GFlop/s
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1 Core - Naïve
1 Core[PF]
1 Core[PF,RB]
1 Core[PF,RB,CB]
2 Cores[*]
4 Cores[*]
2 Sockets x 4 Cores[*]
OSKI
OSKI-PETSc
Dense
Protein
FEM-Sphr
FEM-Cant
Tunnel
FEM-Har
QCD
FEM-Ship
Econom
Epidem
FEM-Accel
Circuit
Webbase
LP
Median
Intel Clovertown
% Speedup for tuning
200%
180%
160%
140%
120%
100%
80%
60%
40%
20%
0%
Benefit of Tuning:
+5% single thread
+80% 8 threads
Dense
Protein
FEM-Sphr
FEM-Cant
Tunnel
FEM-Har
QCD
FEM-Ship
Econom
Epidem
FEM-Accel
Circuit
Webbase
LP
Median


AMD X2
GFlop/s
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Dual Socket x 2 Core [*]
2 Core[*]
1 Core[PF,RB,CB]
1 Core[PF,RB]
1 Core[PF]
1 Core - Naïve
OSKI-PETSc
OSKI
Dense
Protein
FEM-Sphr
FEM-Cant
Tunnel
FEM-Har
QCD
FEM-Ship
Econom
Epidem
FEM-Accel
Circuit
Webbase
LP
Median
AMD X2
% Speedup from tuning
400%
350%
300%
250%
200%
150%
100%
50%
0%
Benefit of Tuning:
+60% single thread
+200% 4 threads
Dense
Protein
FEM-Sphr
FEM-Cant
Tunnel
FEM-Har
QCD
FEM-Ship
Econom
Epidem
FEM-Accel
Circuit
Webbase
LP
Median

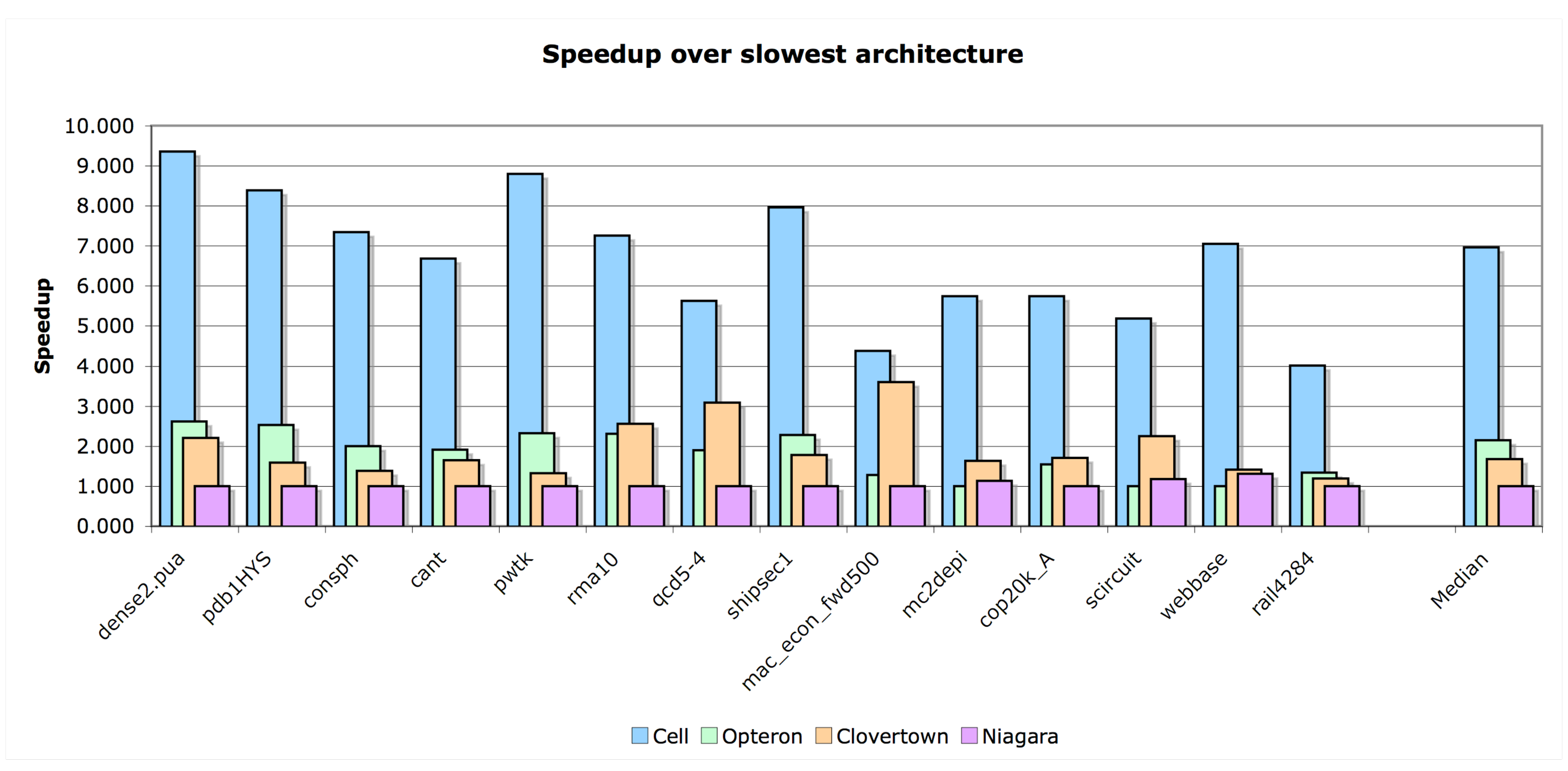

Sun Niagara
GFlop/s
1.5
1.3
1.0
0.8
0.5
0.3
0.0
1 Core - Naïve
1 Core[PF]
1 Core[PF,RB]
1 Core[PF,RB,CB]
1 Core x 4 Threads[*]
2 Cores x 4 Threads[*]
4 Cores x 4 Threads[*]
8 Cores x 4 Threads[*]
Dense
Protein
FEM-Sphr
FEM-Cant
Tunnel
FEM-Har
QCD
FEM-Ship
Econom
Epidem
FEM-Accel
Circuit
Webbase
LP
Median
Sun Niagara
% speedup from tuning
30%
20%
10%
0%
-10%
-20%
Benefit of Tuning:
+5% single thread
+4% 32 threads
Dense
Protein
FEM-Sphr
FEM-Cant
Tunnel
FEM-Har
QCD
FEM-Ship
Econom
Epidem
FEM-Accel
Circuit
Webbase
LP
Median

IBM Cell Blade



- Simpler & less efficient implementation
  - Only BCOO (branchless)
  - 2x1 and greater register blocking (no 1 x anything)
- Performance tracks the DMA flop:byte ratio

Relative Performance



- Cell is bandwidth bound
- Niagara is clearly not
- Noticeable Clovertown cache effect for Harbor, QCD, & Econ

Comments

- complex cores (superscalar, out of order, hardware stream prefetch, giant caches, …) saw the largest benefit from tuning.
- First generation Niagara saw relatively little benefit
- Single thread performance was as good or better performance than OSKI
- Parallel Performance was significantly better than PETSc+OSKI
- Benchmark took 20mins, comparable exhaustive search required 20hrs

Questions?