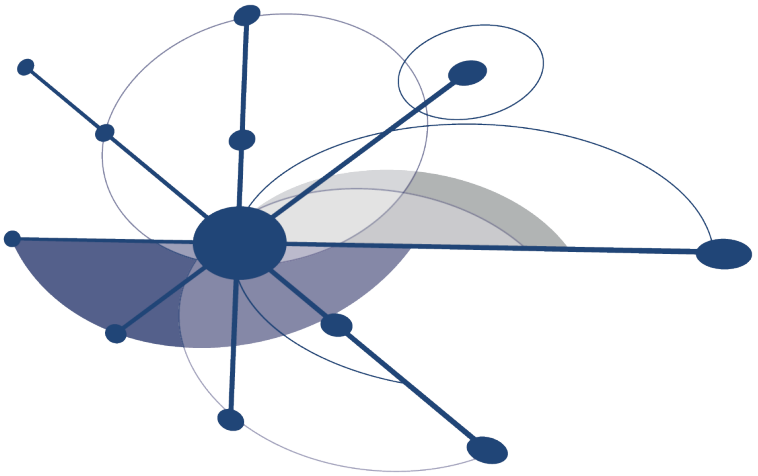
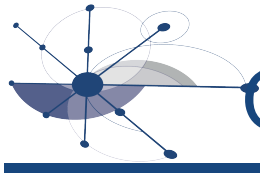


Center for Scalable Application Development Software: Libraries and Compilers

Kathy Yelick (U.C. Berkeley)

SciDAC
Scientific Discovery through Advanced Computing

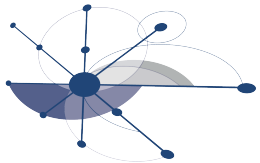


Overview of Compiler and Library Research

Compilers
Rice & Berkeley

Communication Libraries
Argonne & Berkeley

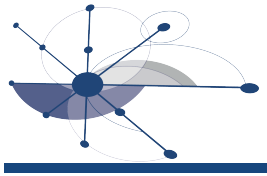
Numerical Libraries
Tennesse & Berkeley



Redundancy Elimination in Loops

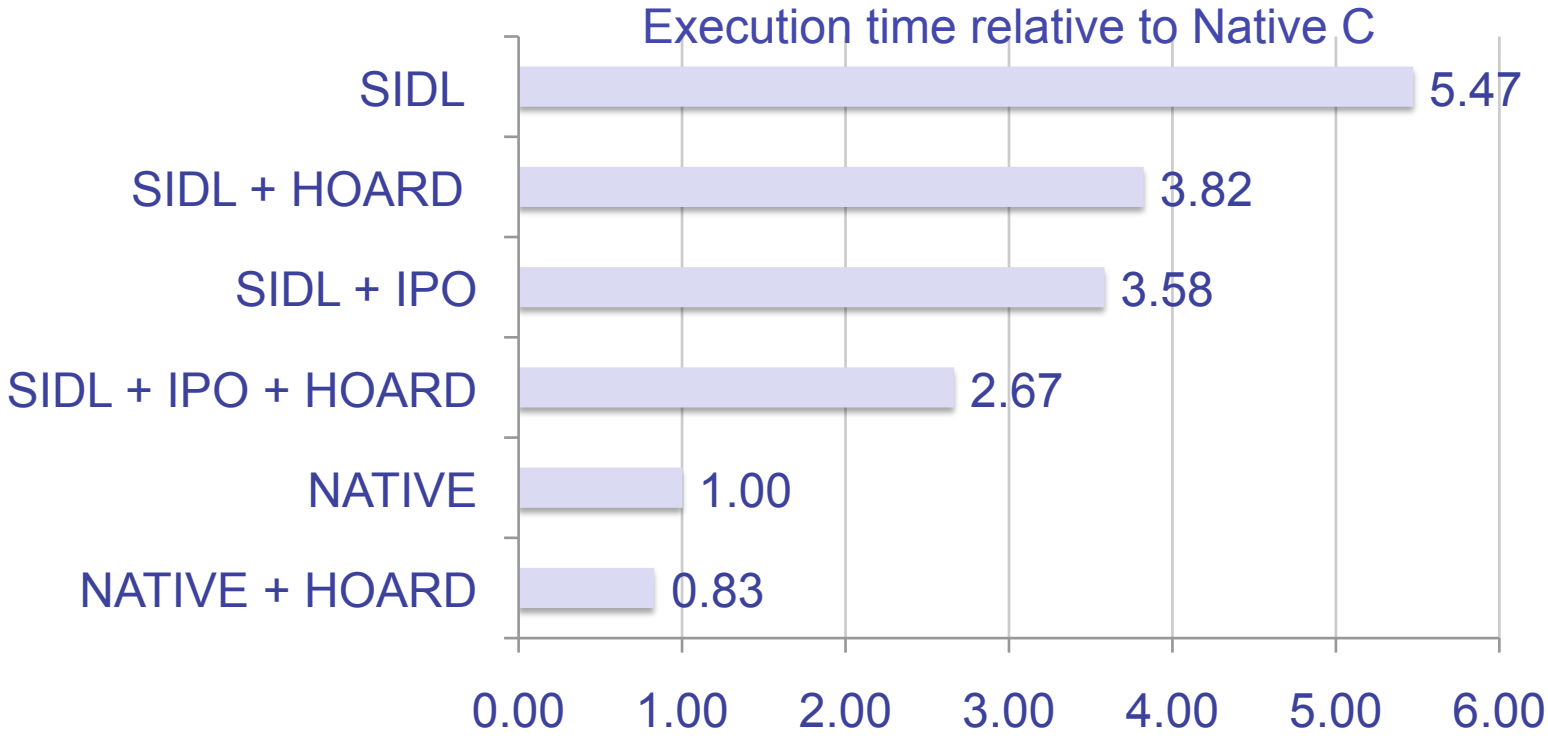
- Redundancy elimination: re-use previously computed expressions
- Previous techniques
  - value numbering: detects general expressions within a single iteration
  - scalar replacement: detects inter-iteration redundancies involving only array references
- Miss opportunities to improve stencil operations: scientific computations, plus signal, and image processing
  - “a+c+d+b” and “d+a+c+b+e” contain redundant subexpression
  - need to change code shape (using associativity and commutativity)
- Approach: construct a graph representation, such that finding the maximal cliques corresponds to redundant subexpressions
- Prototyped in Open64 compiler as part of the loop nest optimizer
- More redundancies eliminated with combined technique
  - ~50% performance improvement on a POP kernel and multigrid stencil

Cooper, Eckhardt, Kennedy, “Redundancy Elimination Revisited,” PACT, 2008.



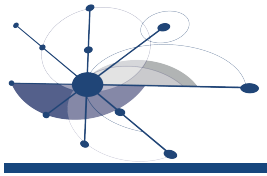
Exploring Optimization of Components

Offline optimization of fine-grained TSTT mesh operations



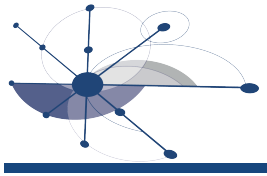
SIDL: Generic code using SIDL Arrays
HOARD: Optimized memory allocator
IPO: LLVM interprocedural optimization (whole program)
NATIVE: Native C code using pointers and C arrays

TSTT Mesh Interface: http://tetra.mech.ubc.ca/ANSLab/publications/TSTT-asm06.pdf



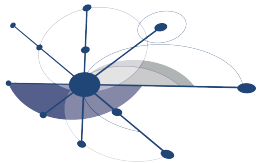
The Case for Dynamic Compilation

- Static compilation challenges
  - software: modular designs, abstract interfaces, de-coupled implementations, dynamic dispatch, dynamic linking/loading
  - hardware: difficult to model, unpredictable latencies, backwards compatibility precludes specialization
- Dynamic compilation opportunities
  - cross-library interprocedural optimization (true whole program)
  - program specialization based on input data
  - machine dependent optimizations for the underlying architecture
  - profile-guided optimization
    - branch straightening for hot paths
    - inlining of indirect function calls
    - insertion/removal of software prefetch instructions
    - tuning of cache-aware memory allocators



Approach

- Fully exploit offline opportunities
  - aggressive interprocedural optimization statically at link time
  - static analysis to detect potential runtime opportunities
  - classic profile-feedback optimization between executions
- Minimize online cost
  - lightweight profiling
    - hardware performance counter sampling (using HPCToolkit infrastructure)
  - multiple cores
    - runtime analysis and optimization in parallel with program execution
- Leverage strengths of multiple program representations
  - optimized native code: provides efficient baseline performance
  - higher-level IR: enables powerful dynamic recompilation (LLVM)
- Selectively optimize
  - focus optimization only on promising regions of code

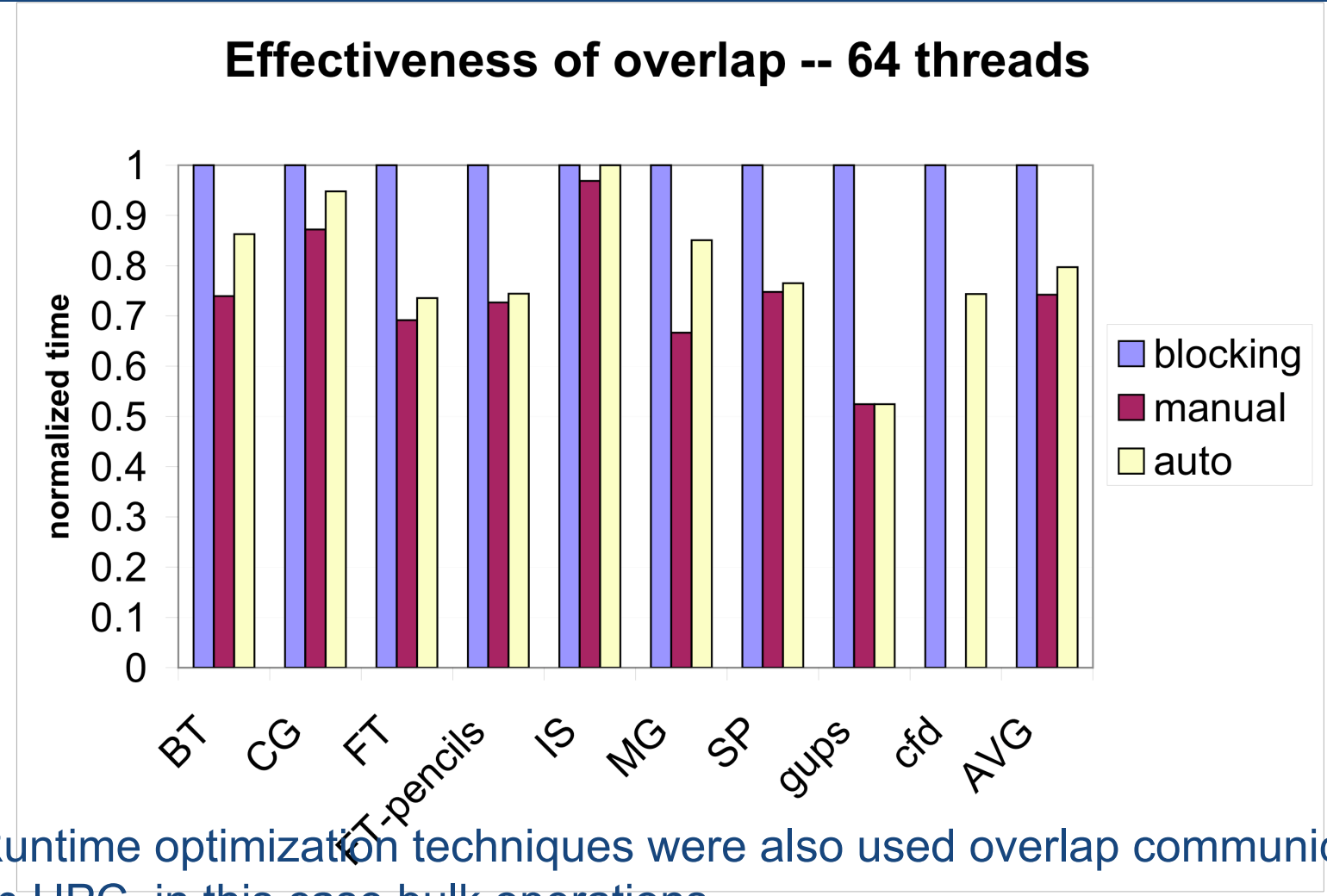


Toward Dynamic Optimization

- Working with ORNL to explore similar optimization of a true CCA mesh benchmark
- Integrating HPCToolkit measurement infrastructure with dynamic compilation framework
- Beginning experimentation with inter-component dynamic optimization of CCA applications using LLVM

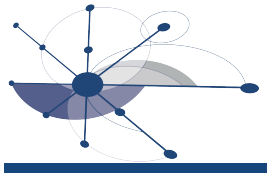
Work is being presented at the Spring 2009 CCA Forum Meeting, April 23-24, 2009

Dynamic Optimizations in PGAS Languages



- Runtime optimization techniques were also used overlap communication in UPC, in this case bulk operations
- Dynamically checks dependences to ensure correct semantics

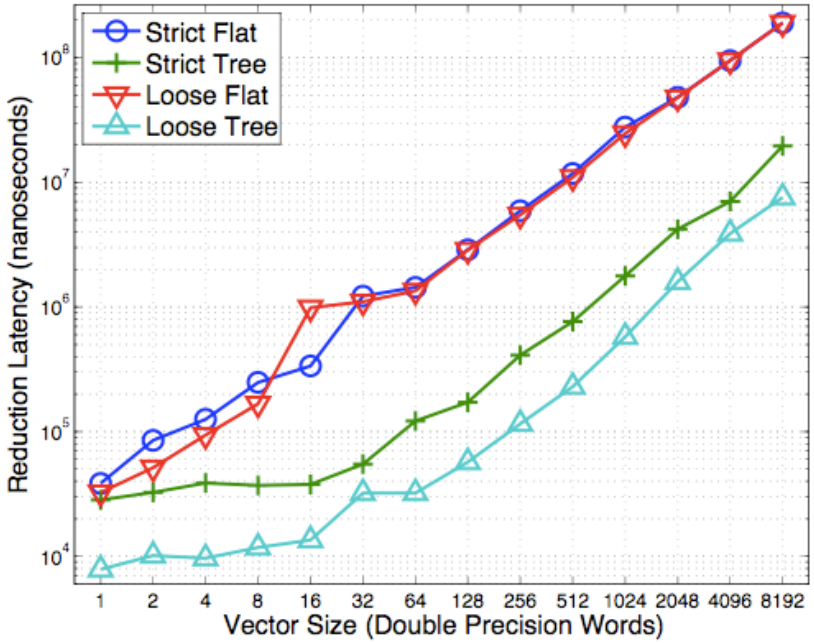
Chen, Iancu, Yelick, ICS 2008.



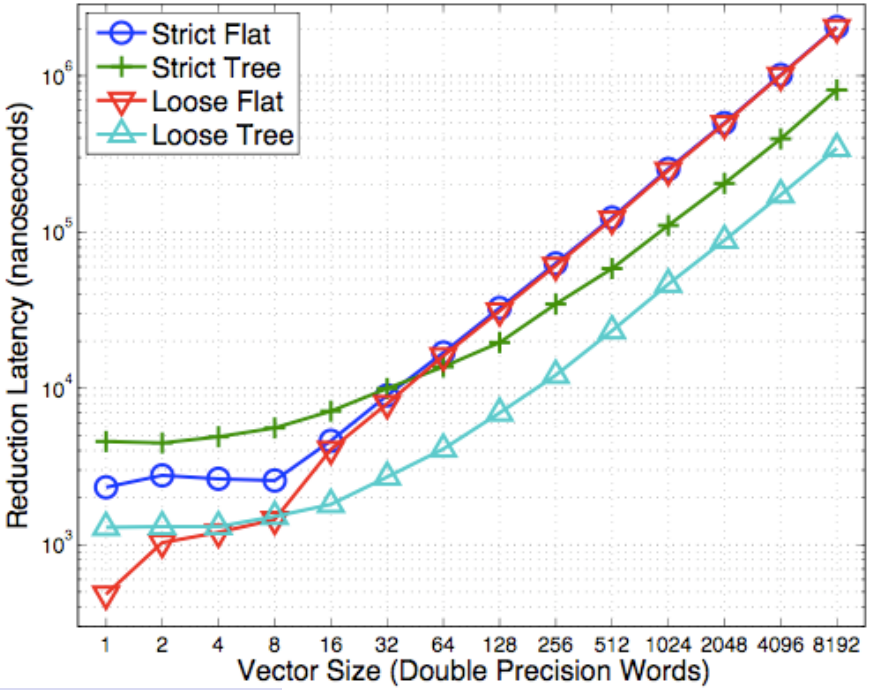
PGAS Collectives

- Use one-sided communication to optimize UPC collectives
- Developing automatic tuning to select optimizations
  - Tree shapes, overlap techniques, synchronization protocols, etc.
- These results are on reduction operations for multicore

Sun Victoria Fall (256 threads) Reductions



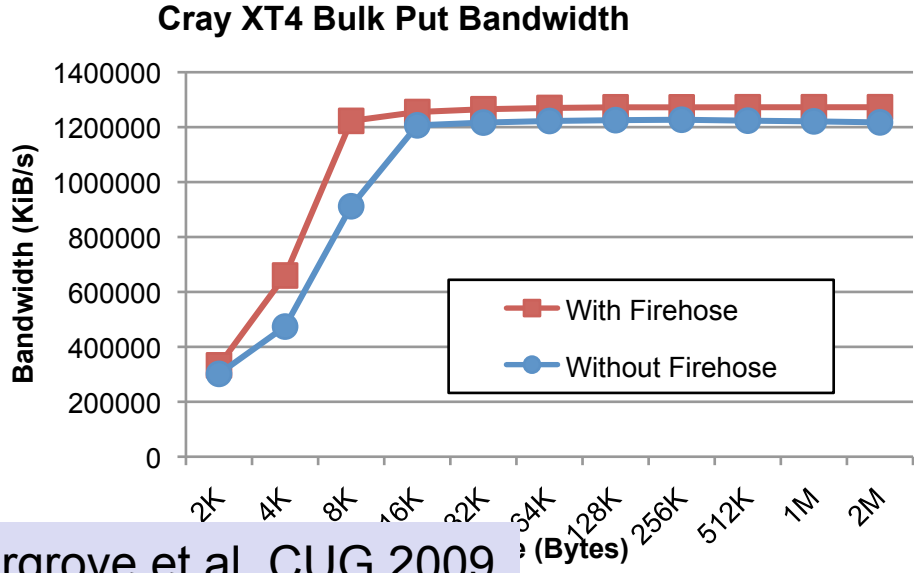
AMD Opteron (32 threads) Reductions

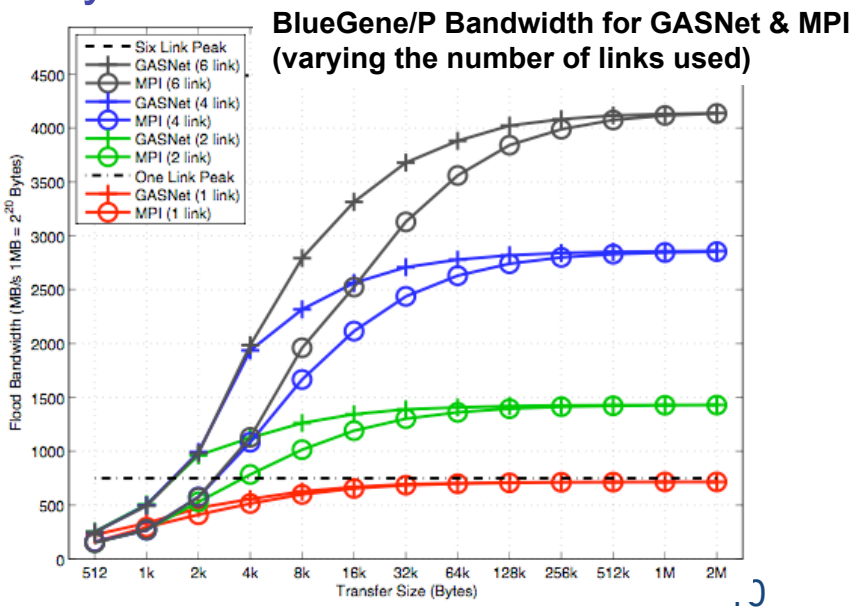


Nishtala & Yelick, HotPar 2009

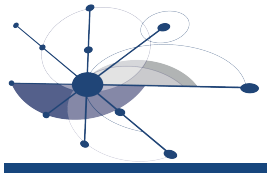
PGAS Communication Runtime Work

- PGAS languages use a one-sided communication model
- GASNet is widely used as a communication layer (joint funding)
  - Berkeley UPC compiler, Berkeley Titanium compiler
  - Cray UPC and CAF compilers for XT4 and XT5
  - Intrepid UPC (gcc-upc); Cray Chapel compiler; Rice CAF compiler
- Released in November, CDs distributed at SC08
  - Improvements in Portals performance (Cray XT) with “firehose”
  - New implementation for BG/P DCMF layer with help from ANL

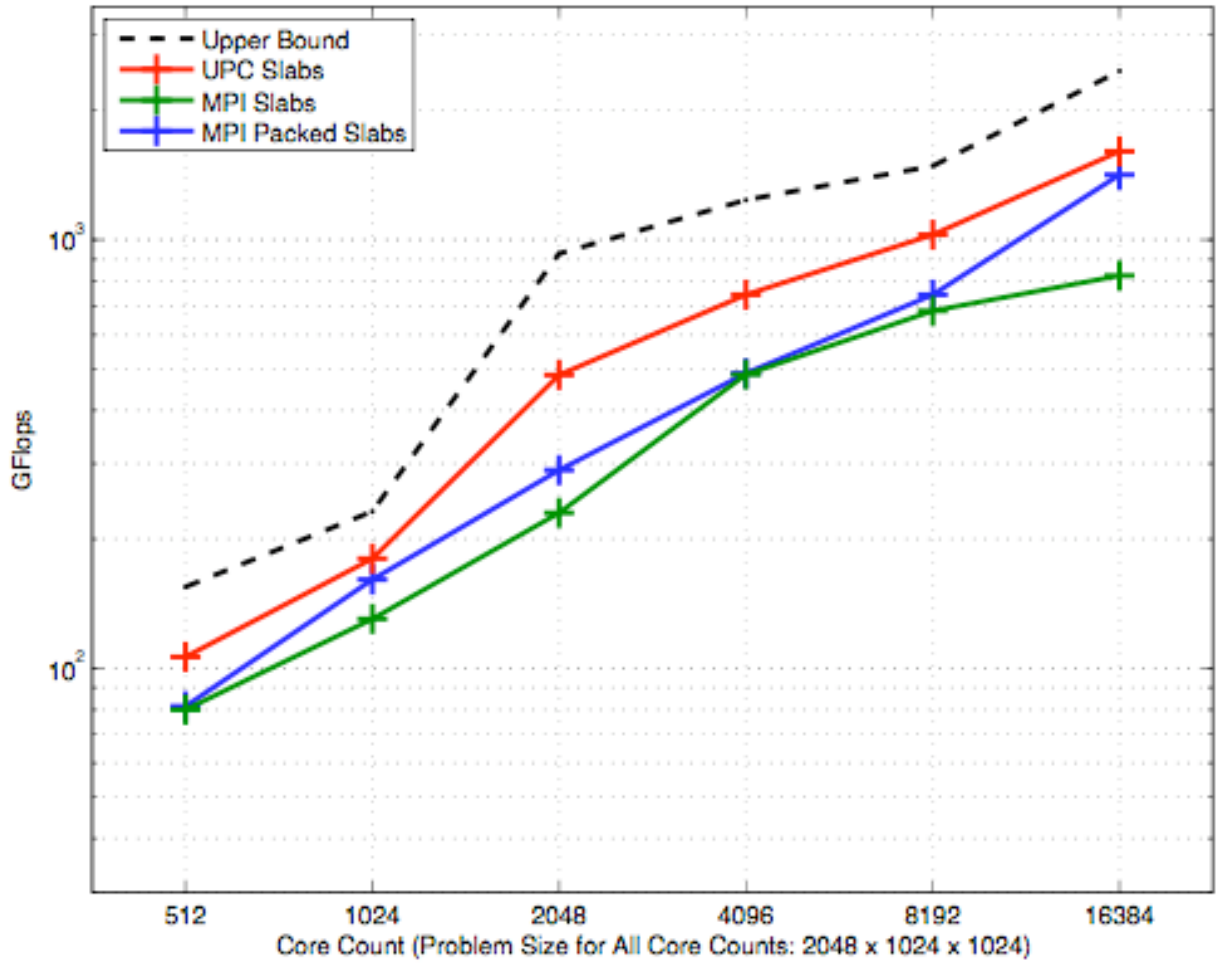




Hargrove et al, CUG 2009



3D FFT Performance on BG/P



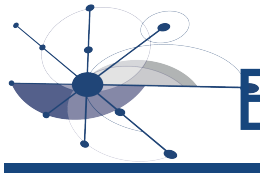
Upper bound is based on a performance model of torus topology and bandwidth

Packed slabs is a bulk-synchronous algorithm that minimizes the number of messages

Slabs overlaps communication with computation by sending data as soon as it is ready

- Strong scaling: shows good performance up to 16K cores

Nishtala et al, IPDPS 2009

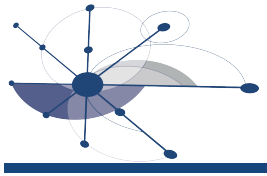


Experiments Comparing MPI and UPC

- UPC work motivated MPICH work
- Table of timings extracted from various experiments with NPB-FT
  - UPC code from Berkeley and MPI code by ANL
- On 512 processes of BG/P

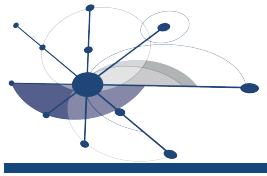
Naïve UPC (not done)	
tuned UPC 61.67	MPI isend/irecv/wait No int: 83.43; Int: 60.32
UPC alltoall 82.64	MPI alltoall 72.68

  - Fastest (by a teeny bit) is MPI isend/irecv *with interrupts on*
- On 1024 processes of BG/P
  - Tuned Berkeley UPC is slightly faster, but MPI all2all is close.



UPC and MPI

- Asynchronous progress in the communication engine is what matters for performance in this particular example
  - Not so much the one-sidedness of UPC put
  - Not so much the fact that UPC
- Non-blocking collective operations in MPI are needed
  - Being worked on now by the MPI-3 Forum.
- But there is no reason that MPI and UPC need to compete
  - MPI + UPC is an important hybrid programming model
  - An alternative path to effective use of multicore
    - Saves memory within a node compared to all MPI: sharing rather than replication
    - Sometimes faster
  - UPC offers locality control for multisocket SMP nodes unlike OpenMP
  - Working on this model is ongoing



Numerical Libraries

Linear Algebra for Multicore

Multicore is a disruptive technology for software

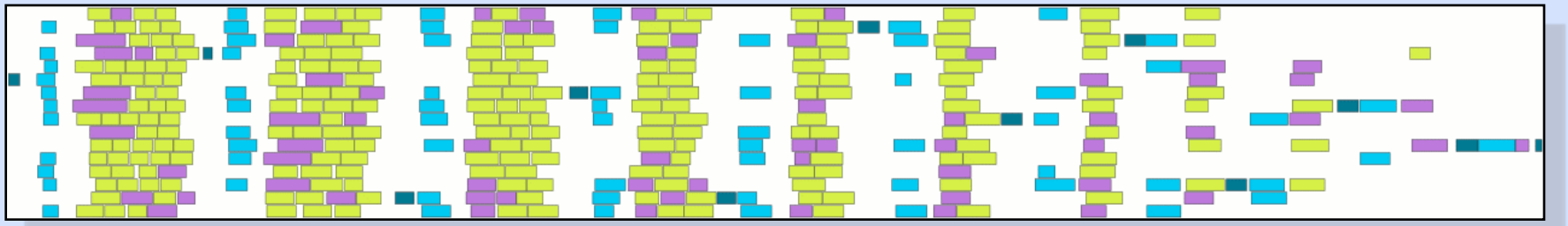
- Must rethink and rewrite applications, algorithms and software
  - as before with cluster computing and message passing
- Numerical libraries, e.g. LAPACK and ScLAPACK, need to change

A Motivating Example: Cholesky

Existing software based on BLAS uses fork-join parallelism

- causes stalls on multicore systems

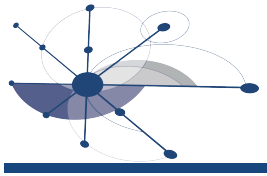
Nested fork-join parallelism (e.g., Cilk, TBB)



PLASMA: Parallel Linear Algebra s/w for Multicore

- Objectives
  - parallel performance
    - high utilization of each core
    - scaling to large numbers of cores
  - any memory model
    - shared memory: symmetric or non-symmetric
    - distributed memory
    - GPUs
- Solution properties
  - asychronicity: avoid fork-join (bulk synchronous design)
  - dynamic scheduling: out-of-order execution
  - fine granularity: independent block operations
  - locality of reference: store data using block data layout

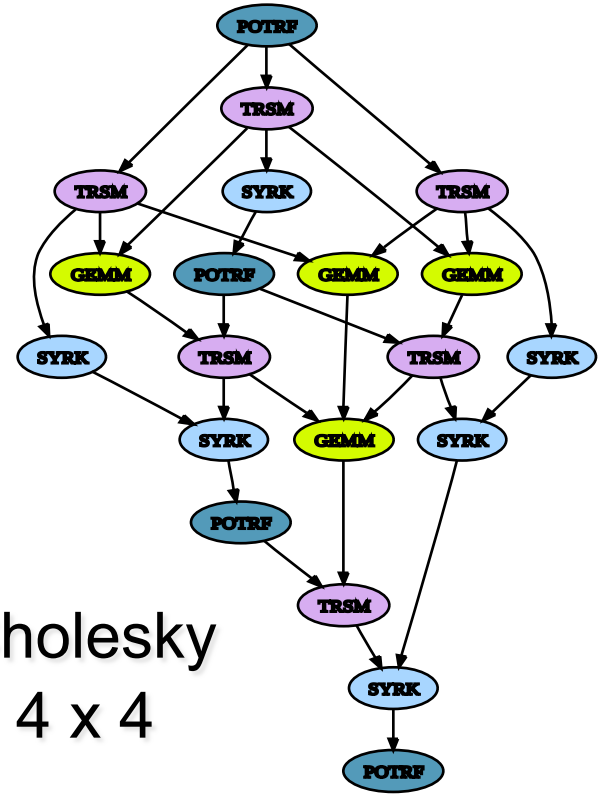
A community effort led by Tennessee and Berkeley
(similar to LAPACK/ScaLAPACK)



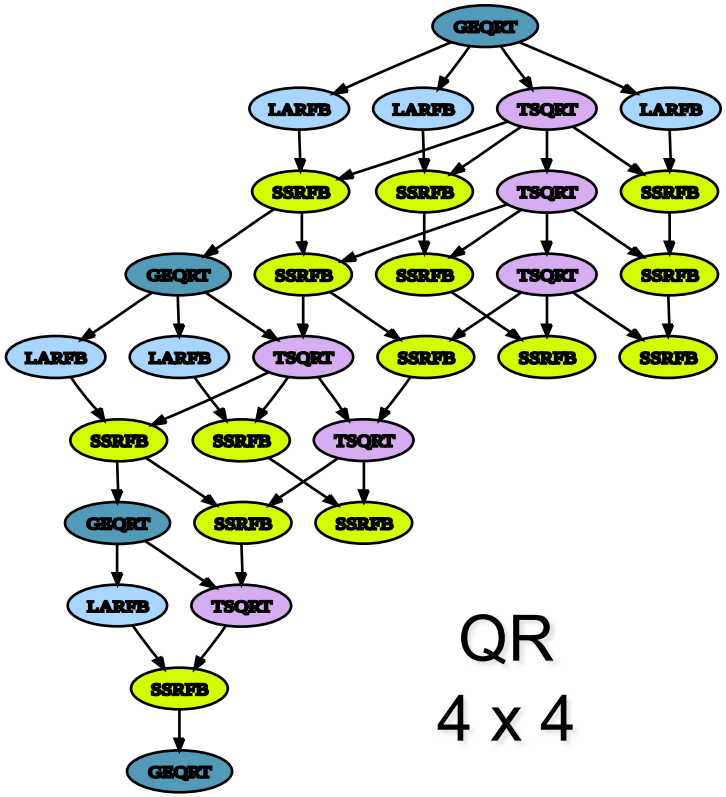
PLASMA Methodology

Computations as DAGs

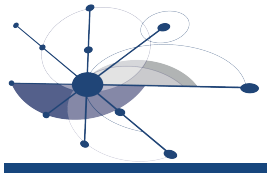
Reorganize algorithms and software to work on tiles that are scheduled based on the directed acyclic graph of the computation



Cholesky
4 x 4

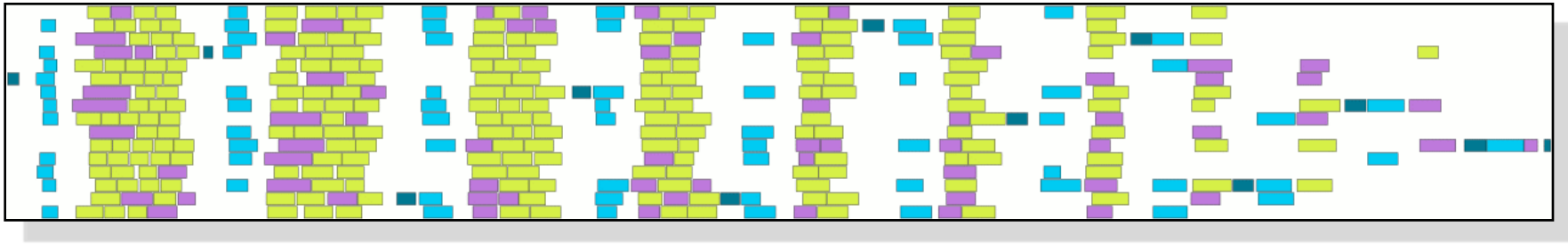


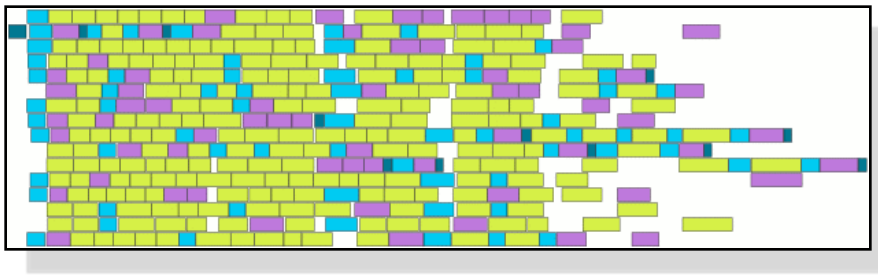
QR
4 x 4



Cholesky using PLASMA

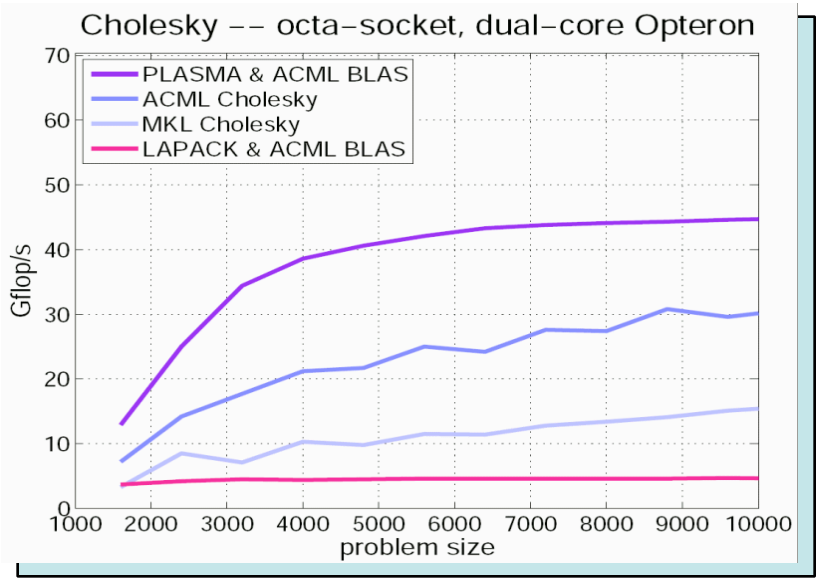
Nested fork-join parallelism (e.g., Cilk, TBB)

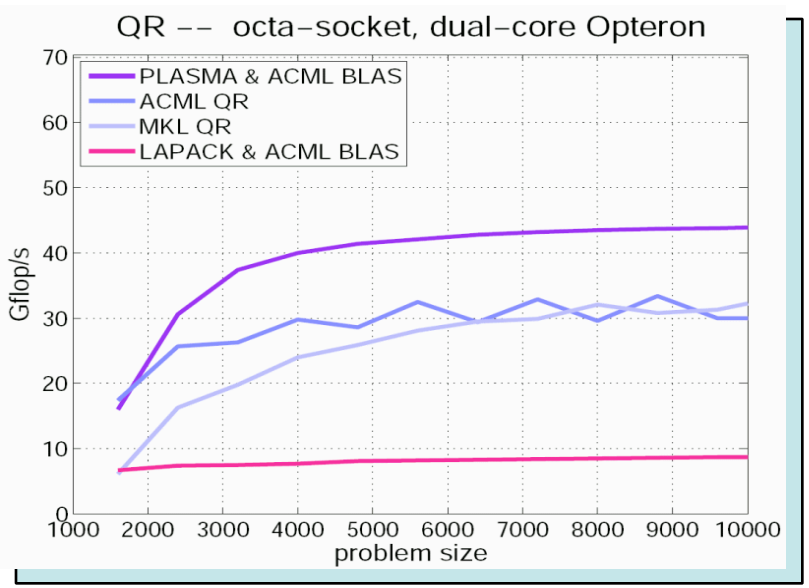


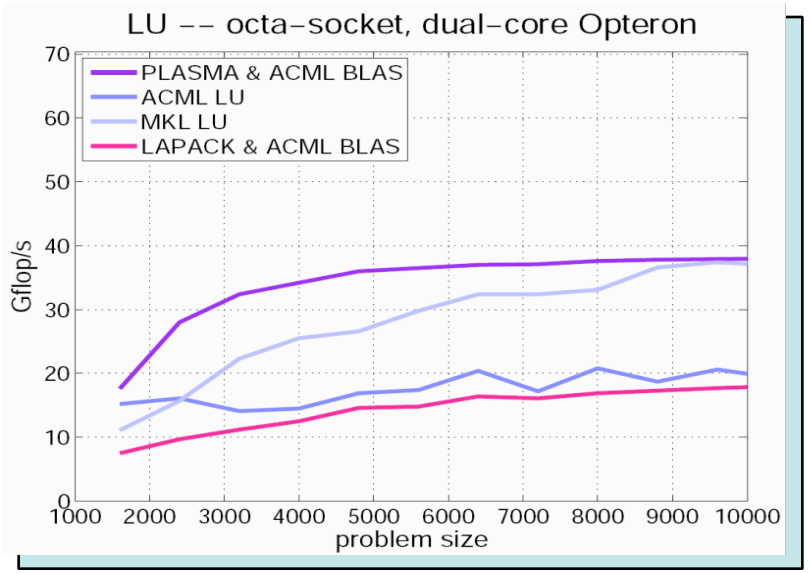


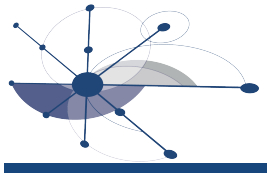
PLASMA
Arbitrary DAG
Fully dynamic scheduling

PLASMA Provides Highest Performance



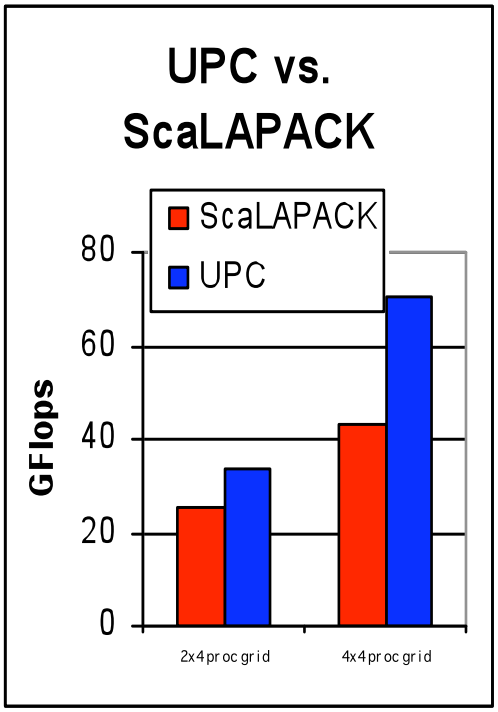


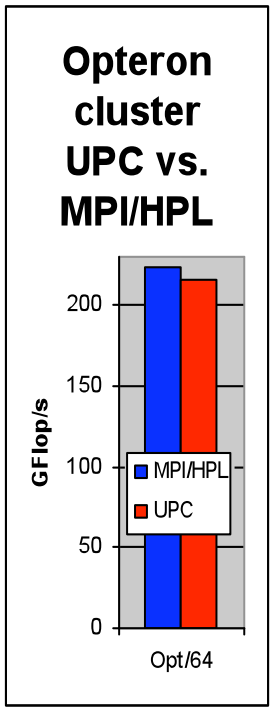


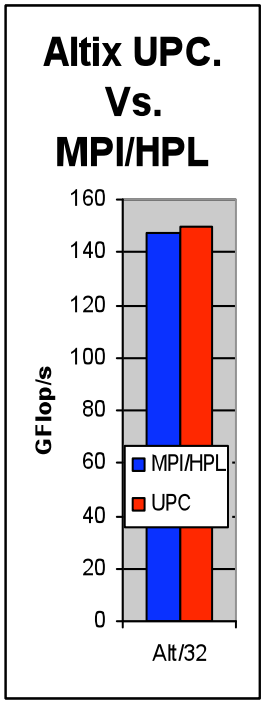


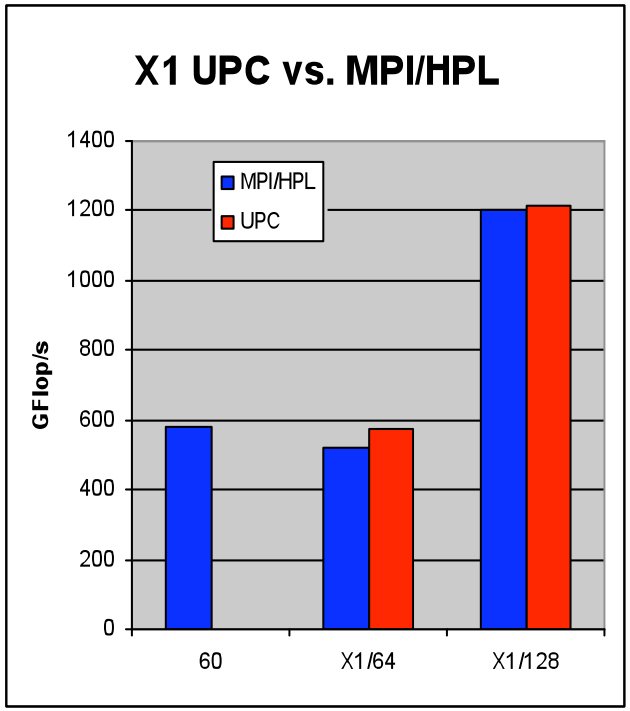
DAG Scheduling of LU in UPC + Multithreading

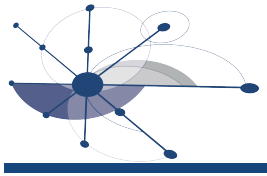
- UPC uses a static threads (SPMD) programming model
  - Multithreading used to mask latency and to mask dependence delays
  - Three levels of threads:
    - UPC threads (data layout, each runs an event scheduling loop)
    - Multithreaded BLAS (boost efficiency)
    - User level (non-preemptive) threads with explicit yield
  - New problem in distributed memory: allocator deadlock





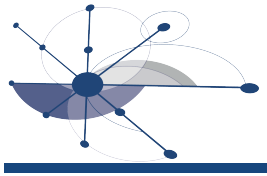






Leveraging Mixed Precision

- Why use single precision as part of the computation? Speed!
  - higher parallelism within vector units
    - 4 ops/cycle (usually) instead of 2 ops/cycle
  - reduced data motion
    - 32-bit vs. 64-bit data
  - higher locality in cache
    - more data items in cache
- Approach
  - compute a 32-bit result
  - calculate a correction for 32-bit results using 64-bit operations
  - update of 32-bit results with the correction using high precision



Mixed-Precision Iterative Refinement

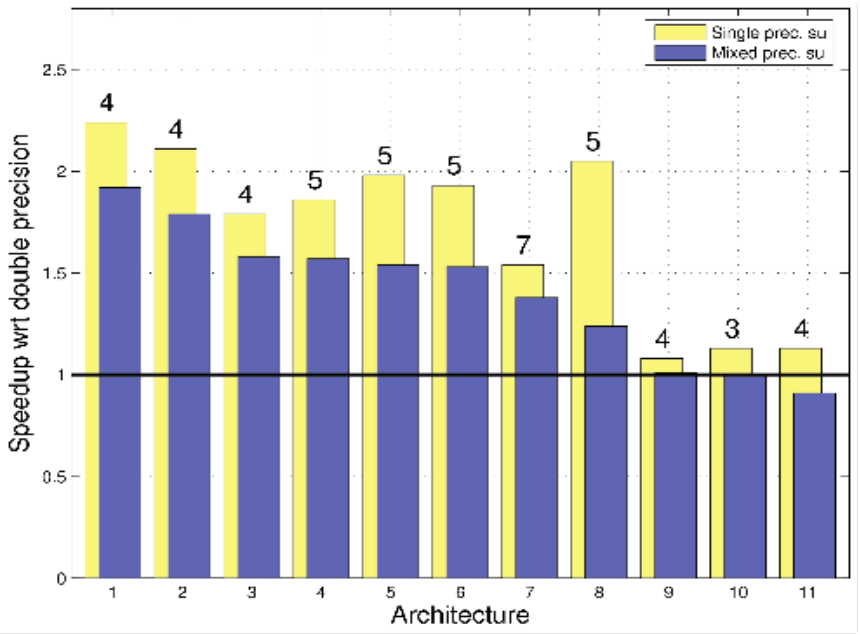
- Iterative refinement for dense systems, $Ax = b$, can work this way:

```
L U = lu(A)                                   O(n^3)
x = L\(U\b)                                   O(n^2)
r = b - Ax                                    O(n^2)
WHILE || r || not small enough
    z = L\(U\r)                               O(n^2)
    x = x + z                                 O(n^1)
    r = b - Ax                                O(n^2)
END
```

- Wilkinson, Moler, Stewart, & Higham provide error bound for SP floating point results when using DP floating point
- Using this, we can compute the result to 64-bit precision

Results for Mixed Precision

Iterative Refinement for Dense $Ax = b$

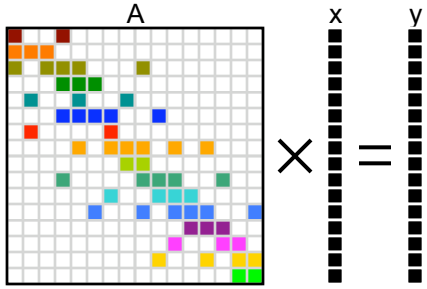


	Architecture (BLAS)
1	Intel Pentium III Coppermine (Goto)
2	Intel Pentium III Katmai (Goto)
3	Sun UltraSPARC IIe (Sunperf)
4	Intel Pentium IV Prescott (Goto)
5	Intel Pentium IV-M Northwood (Goto)
6	AMD Opteron (Goto)
7	Cray X1 (libsci)
8	IBM Power PC G5 (2.7 GHz) (VecLib)
9	Compaq Alpha EV6 (CXML)
10	IBM SP Power3 (ESSL)
11	SGI Octane (ATLAS)

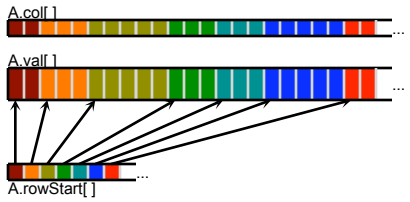
Architecture (BLAS-MPI)	# procs	n	DP Solve /SP Solve	DP Solve /Iter Ref	# iter
AMD Opteron (Goto – OpenMPI MX)	32	22627	1.85	1.79	6
AMD Opteron (Goto – OpenMPI MX)	64	32000	1.90	1.83	6

Autotuning Sparse Matrix Vector Multiply

- Sparse Matrix-Vectory Multiply (SpMV)
  - Evaluate y=Ax
  - A is a sparse matrix, x & y are dense vectors
- Challenges
  - **Very low arithmetic intensity (often <0.166 flops/byte)**
  - Difficult to exploit ILP(bad for superscalar),
  - Difficult to exploit DLP(bad for SIMD)
- Optimizations for Multicore by Williams et al, SC07
  - Supported in part by PERI



(a)
algebra conceptualization



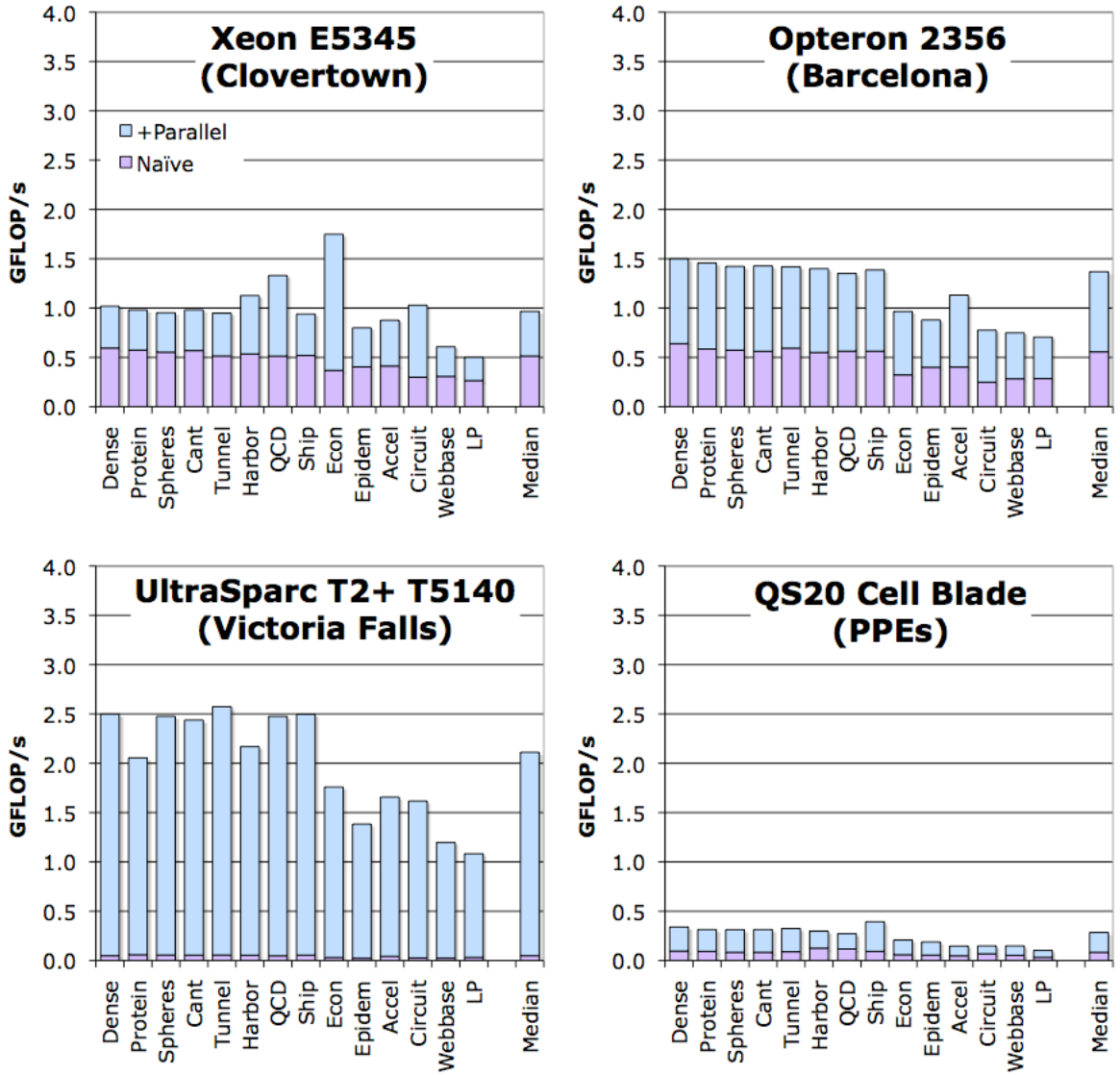
(b)
CSR data structure

```
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

SpMV Performance

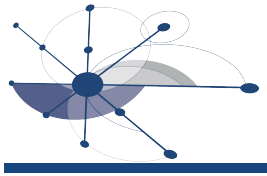
(simple parallelization)



- Out-of-the box SpMV performance on a suite of 14 matrices
- Scalability isn't great
- Is this performance good?

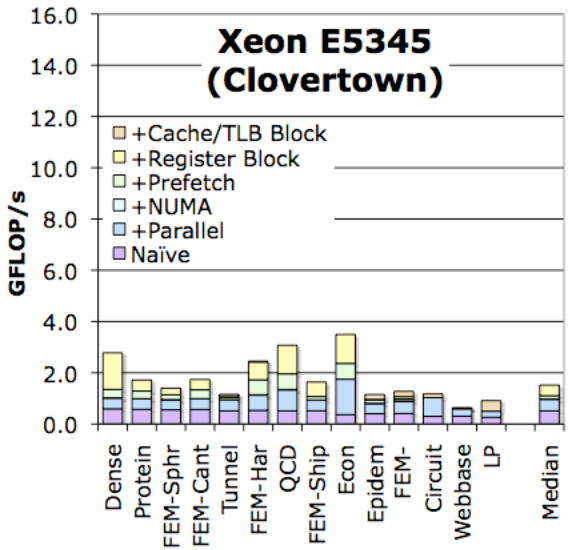
Naïve Pthreads

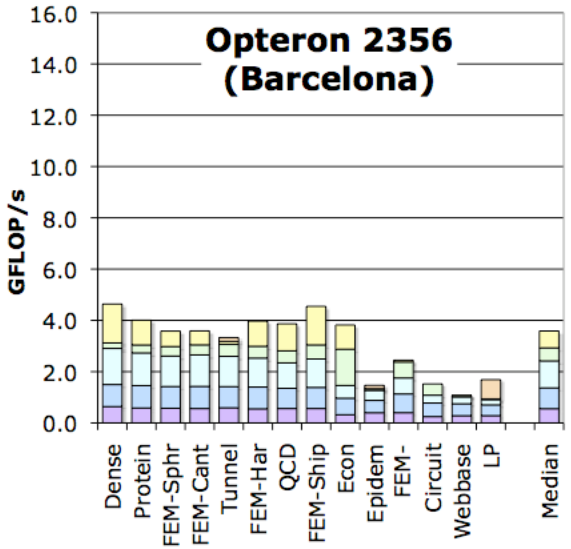
Naïve

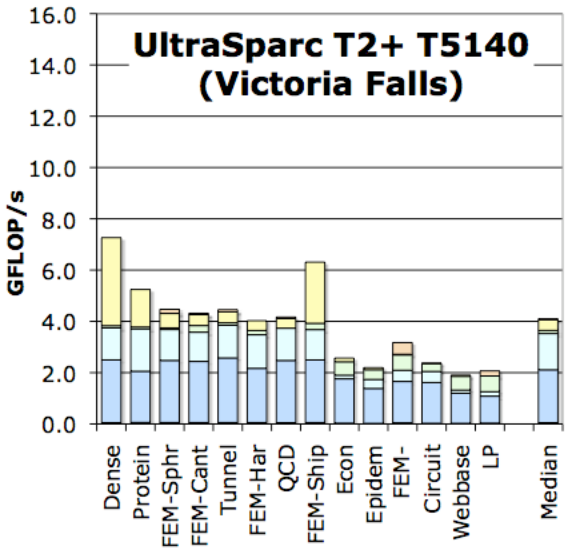


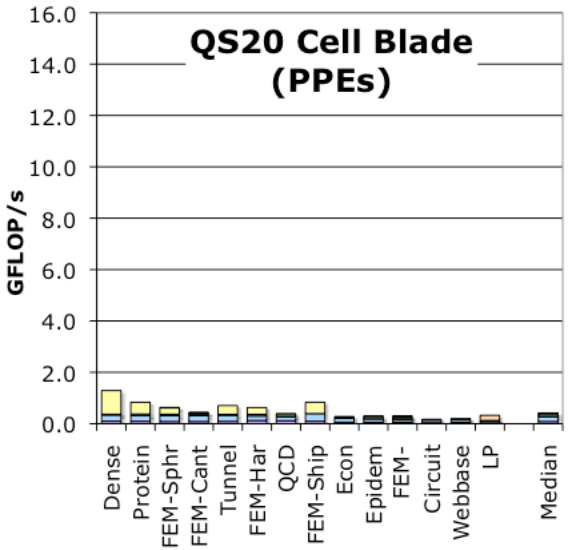
Auto-tuned SpMV Performance

(portable C)

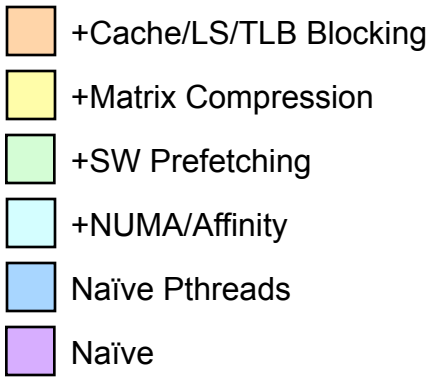


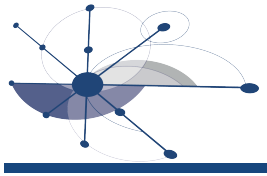






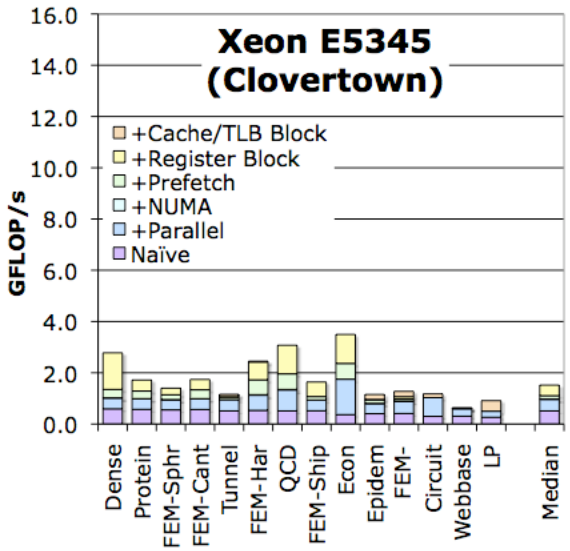
- Fully auto-tuned SpMV performance across the suite of matrices
- Why do some optimizations work better on some architectures?

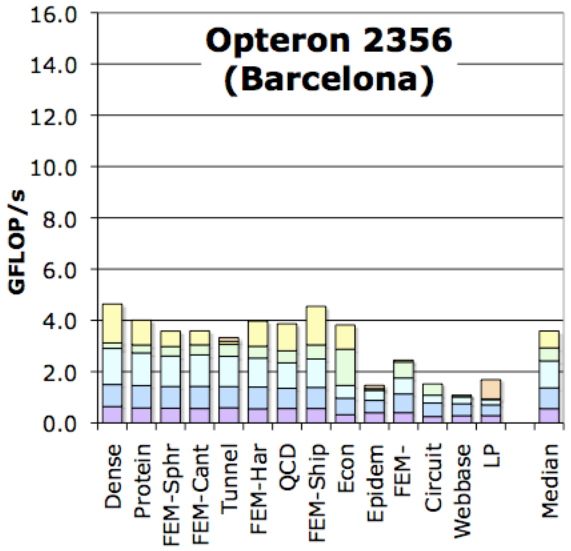


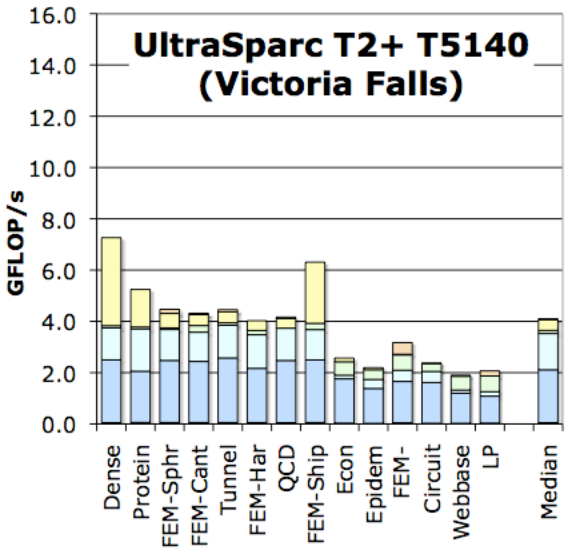


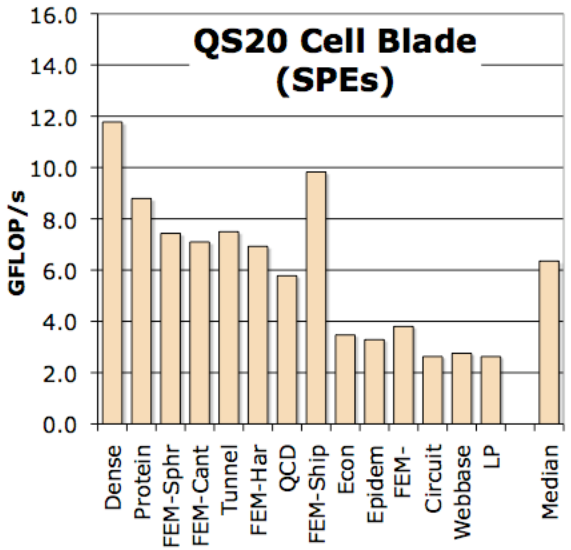
Auto-tuned SpMV Performance

(architecture specific optimizations)

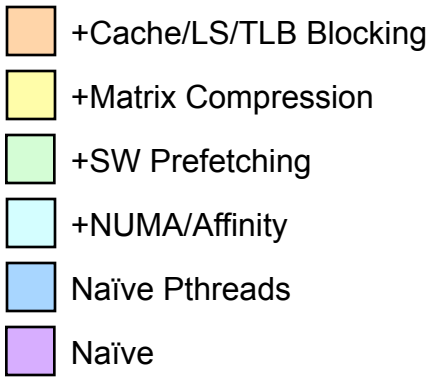




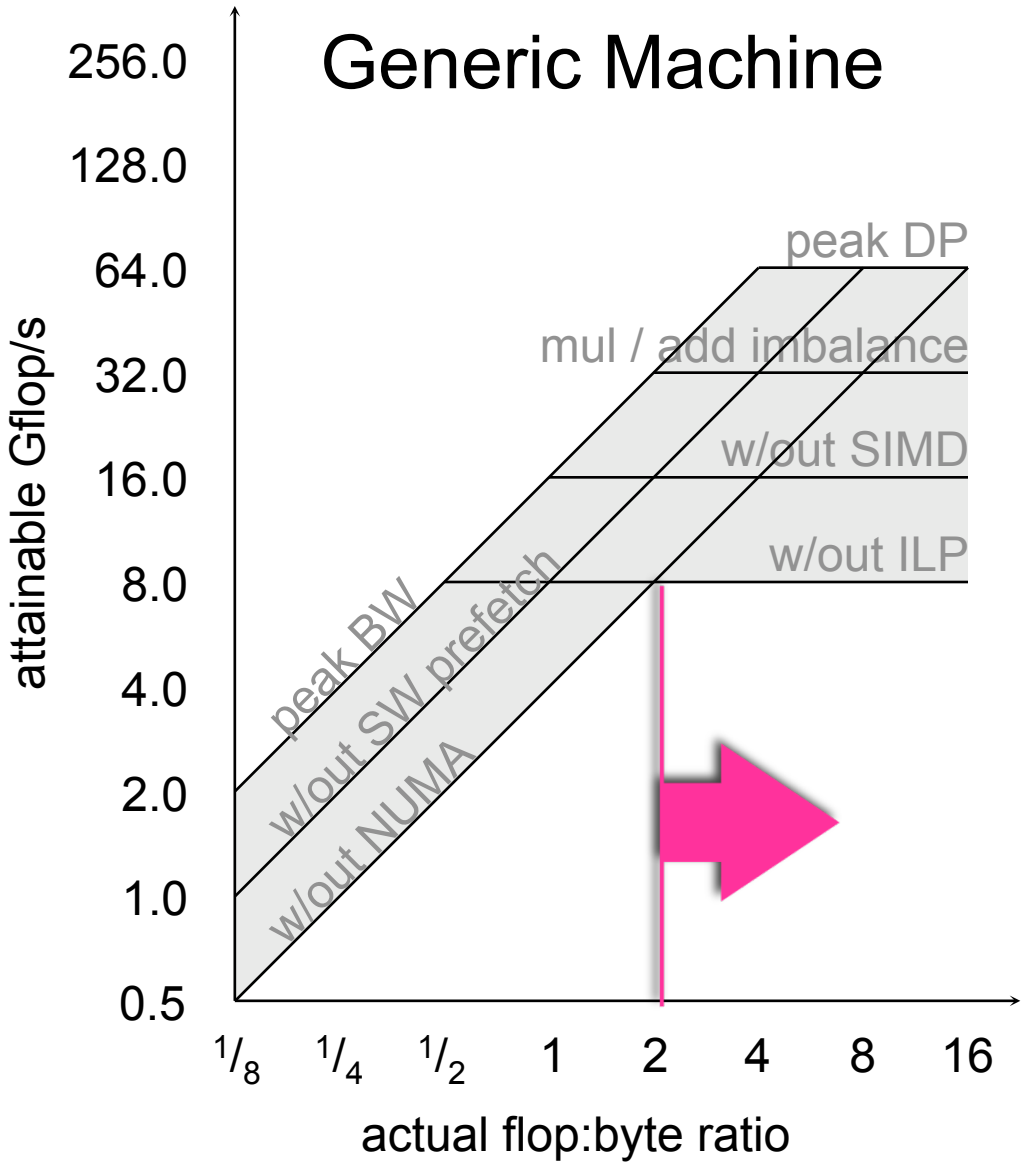




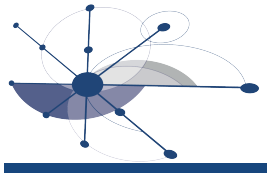
- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?



The Roofline Performance Model

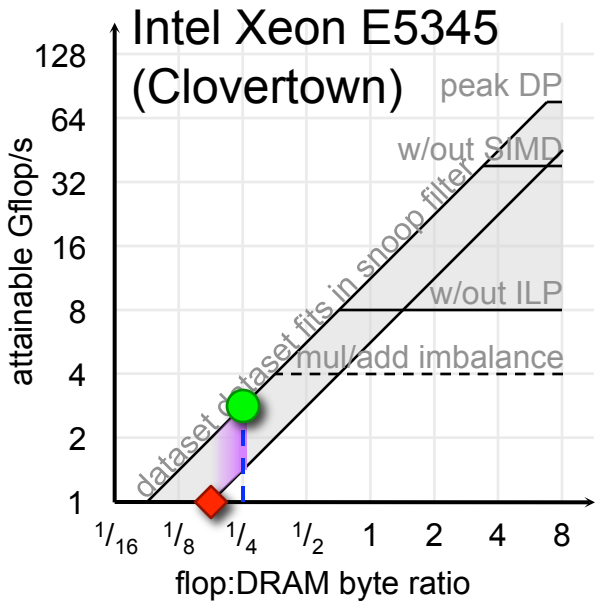


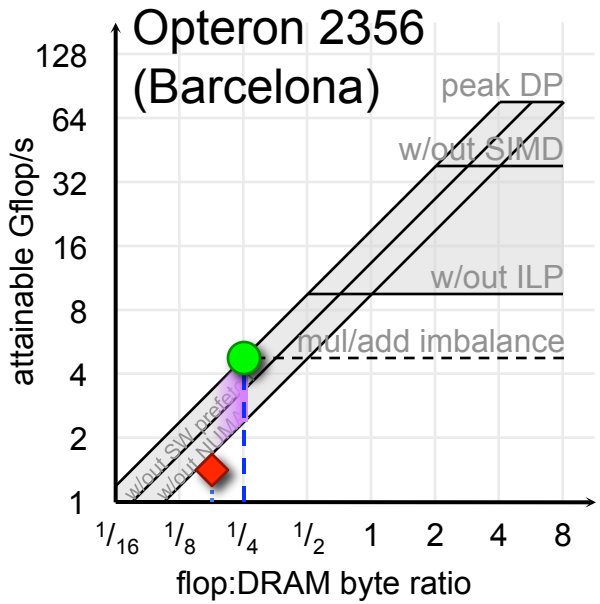
- Locations of posts in the building are determined by algorithmic intensity
- Will vary across algorithms and with bandwidth-reducing optimizations, such as better cache re-use (tiling), compression techniques

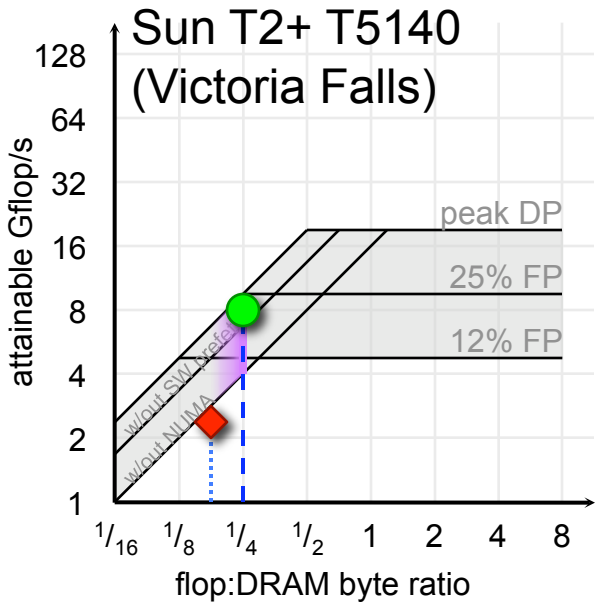


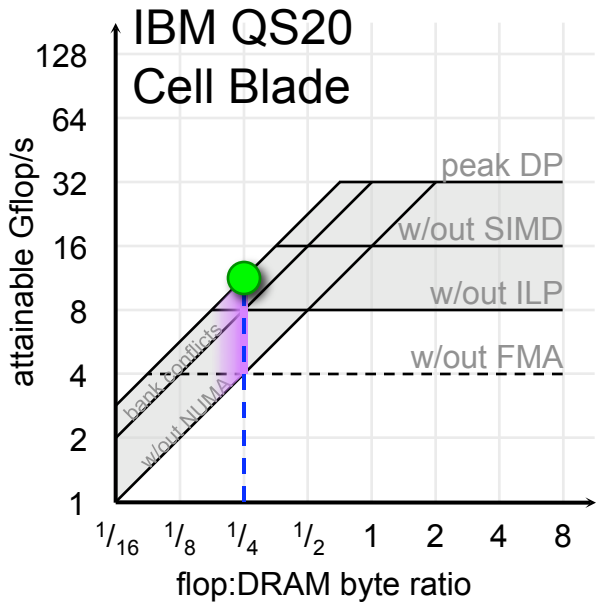
Roofline model for SpMV

(matrix compression)

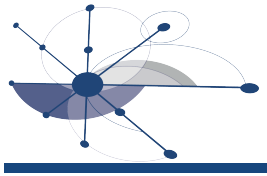






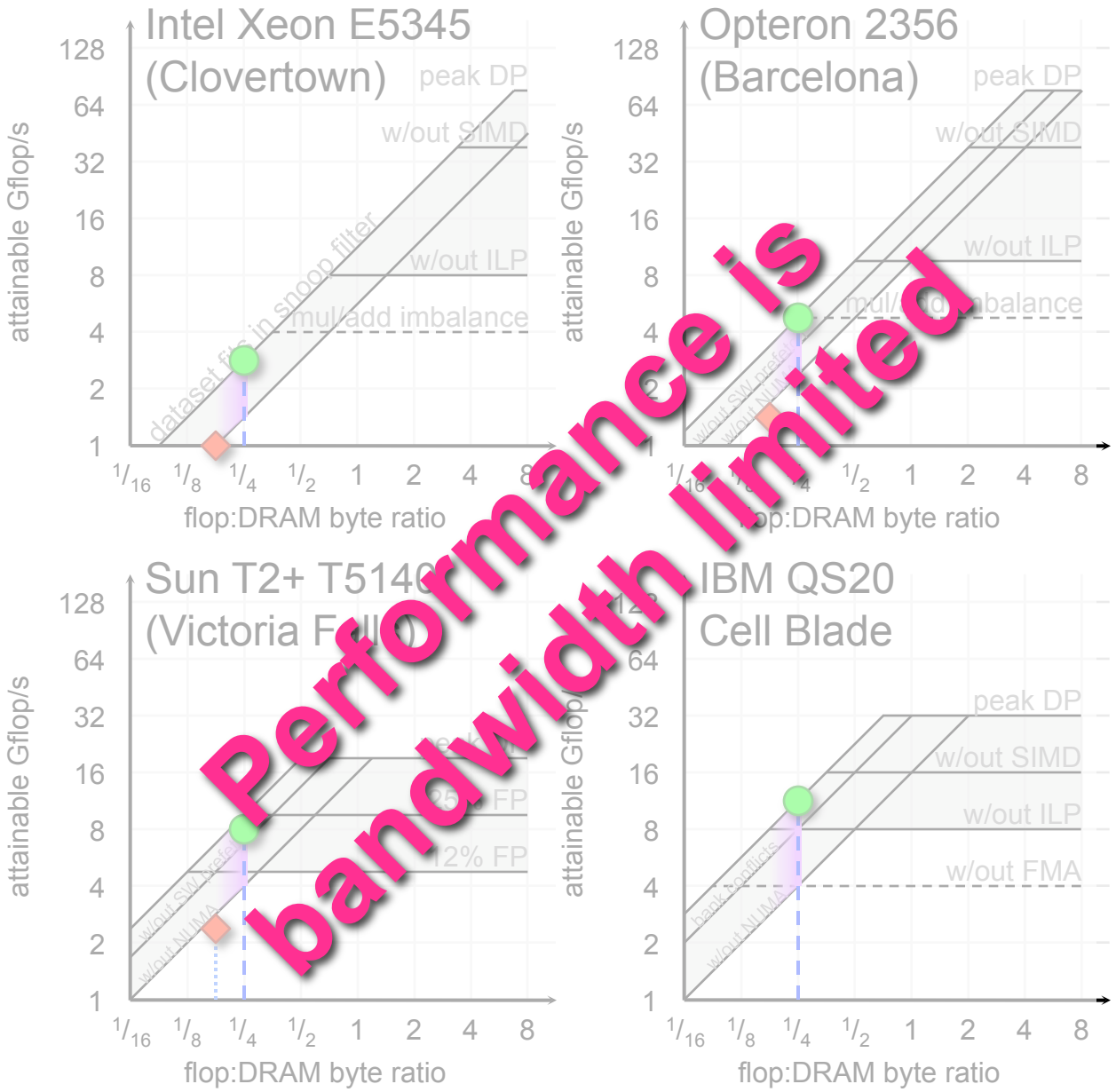


- Inherent FMA
- Register blocking improves ILP, DLP, flop:byte ratio, and FP% of instructions



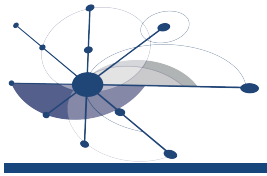
Roofline model for SpMV

(matrix compression)



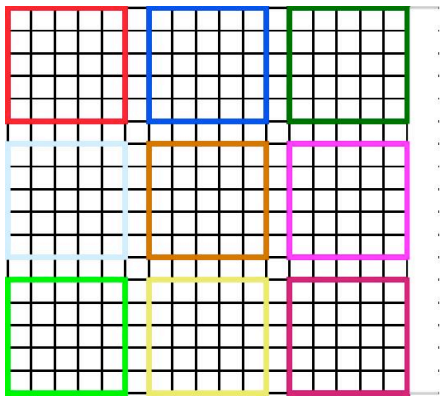
Performance is bandwidth limited

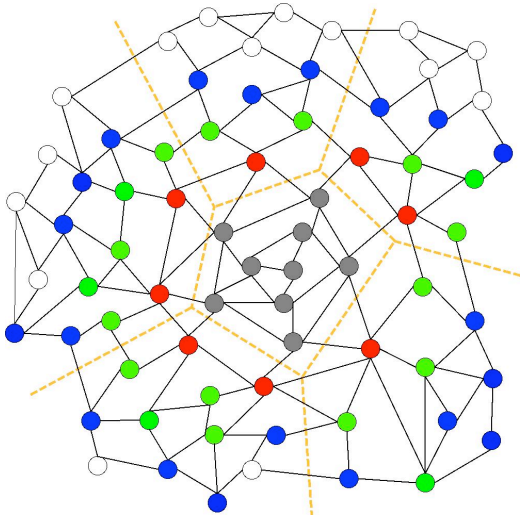
- SpMV *should* run close to memory bandwidth
  - Time to read matrix is major cost
- Can we do better?
- Can we compute A^k*x with one read of A?
- If so, this would
  - Reduce # messages
  - Reduce memory bandwidth



Avoiding Communication in Iterative Solvers

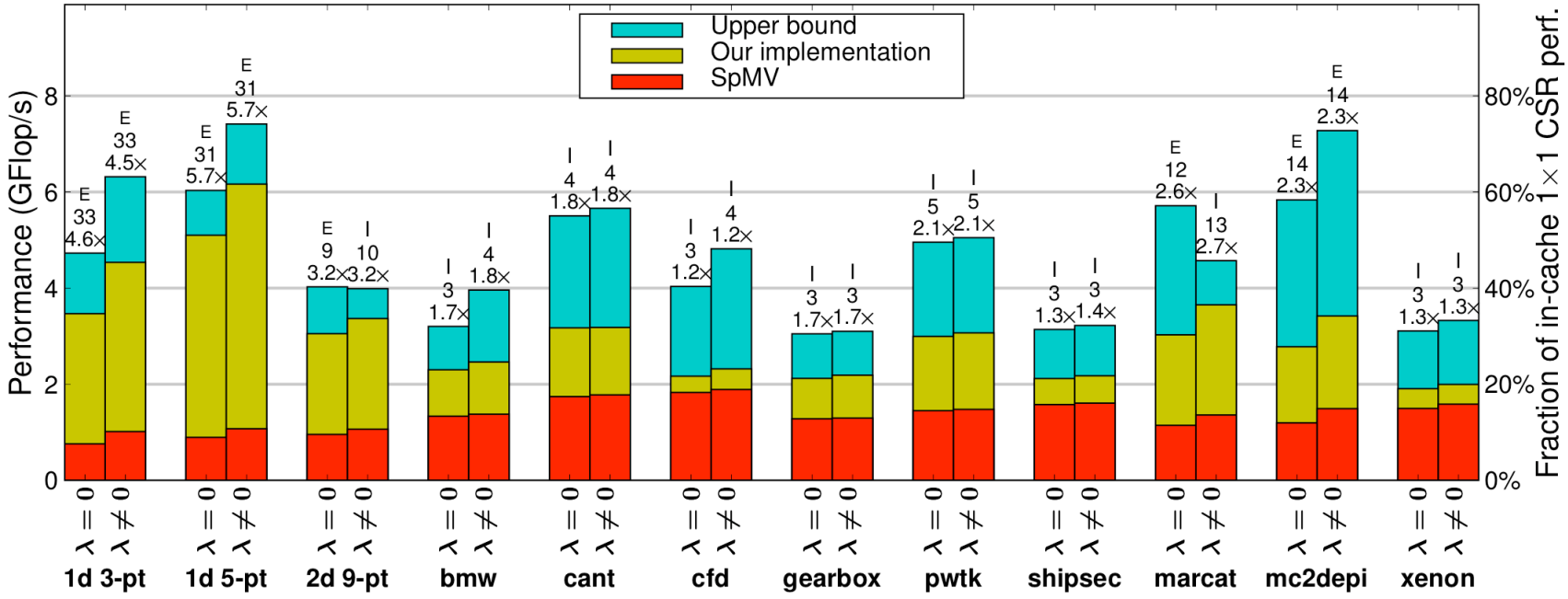
- Consider Sparse Iterative Methods for *Ax=b*
  - Use Krylov Subspace Methods: GMRES, CG
  - Can we lower the communication costs?
    - Latency of communication, i.e., reduce # messages by computing A^k*x with one read of remote x
    - Bandwidth to memory hierarchy, i.e., compute A
- Example: GMRES for *Ax=b* on "2D Mesh"
  - *x* lives on n-by-n mesh
  - Partitioned on $p^{1/2}$ -by- $p^{1/2}$ grid
  - *A* has "5 point stencil" (Laplacian)
- Much more complex in general
  - TSP algorithm to sort matrix
  - Minimize communication events



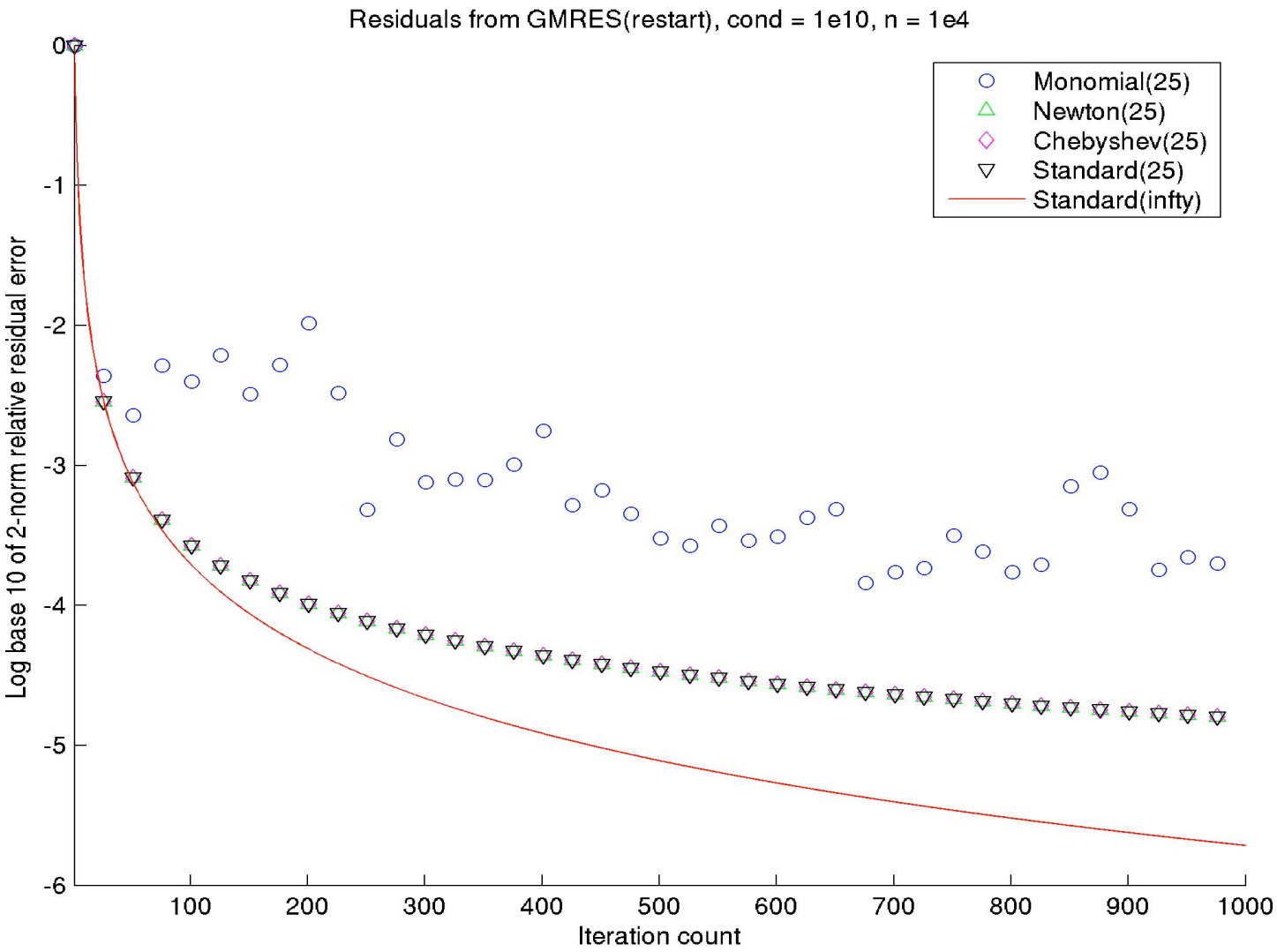


Avoiding Communication in Sparse Linear Algebra - Summary

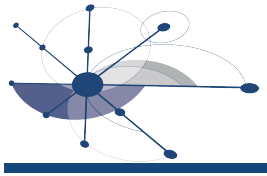
- Take *k* steps of Krylov subspace method
  - GMRES, CG, Lanczos, Arnoldi
  - Parallel implementation
    - Conventional: O(k log p) messages
    - **"New"**: O(log p) messages - optimal
  - Serial implementation
    - Conventional: O(k) moves of data from slow to fast memory
    - **"New"**: O(1) moves of data – optimal
- Performance of $A^k x$ operation relative to Ax and upper bound



But the Numerics have to Change!



- Collaboration with PERI and Tops SciDACS among others



Summary

- All aspects of the optimization space
  - Changing languages
  - Changing compilers
  - Changing architecture
    - or at least evaluating them and exploint all features
  - Changing algorithms

- Joint projects within this SciDAC effort and between others