

Optimizing 27-point Stencil on Multicore

Kaushik Datta, Samuel Williams, Vasily Volkov,
Jonathan Carter, Leonid Oliker, John Shalf, and
Katherine Yelick

CRD/NERSC, Berkeley Lab

EECS, University of California, Berkeley

JTCarter@lbl.gov

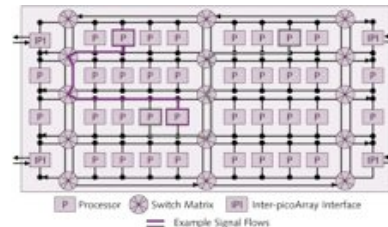
iWAPT 2009

October 1-2 2009

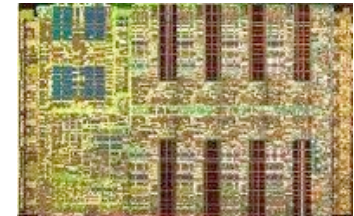
Expanding Set of Manycore Architectures

- **Potential to deliver most performance for space and power for HPC**
- **Server and PC commodity**
 - Intel and AMD x86, Sun UltraSparc
- **Graphics Processors & Gaming**
 - NVIDIA GTX280, STI Cell
- **Embedded**
 - Intel Atom, ARM (cell phone, etc.)

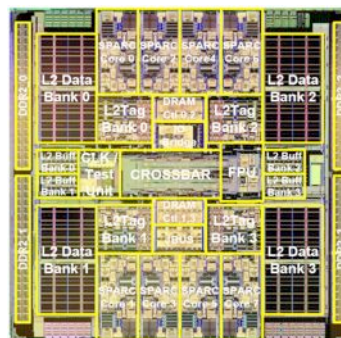
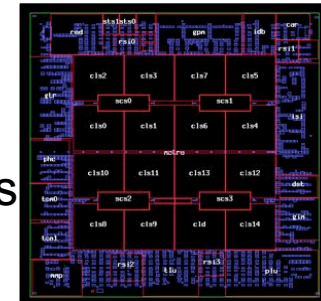
Picochip DSP
1 GPP core
248 ASPs



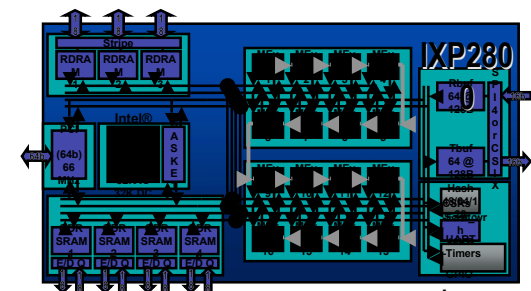
STI Cell
8 ASPs, 1GPP



Cisco CRS-1
188 Tensilica GPPs



Sun Niagara
8 GPP cores (32 threads)

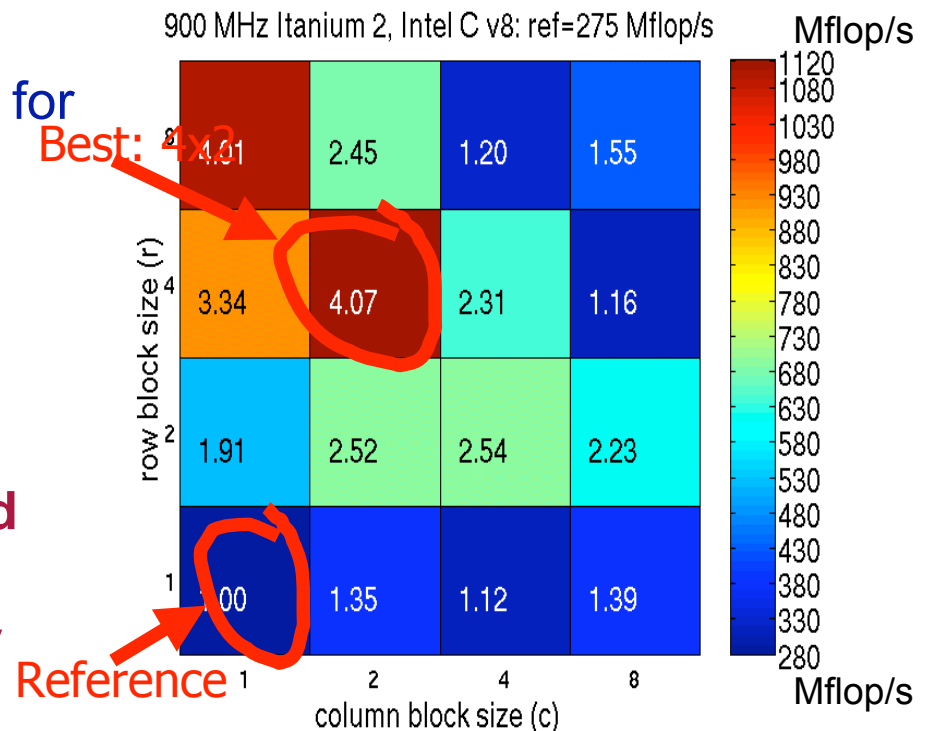


Intel Network Processor
1 GPP Core
16 ASPs (128 threads)

Auto-tuning

- **Problem: want to obtain and compare best potential performance of diverse architectures, avoiding**
 - Non-portable code
 - Labor-intensive user optimizations for each specific architecture
- **A Solution: Auto-tuning**
 - Automate search across a complex optimization space
 - Achieve performance far beyond current compilers
 - Achieve performance portability for diverse architectures

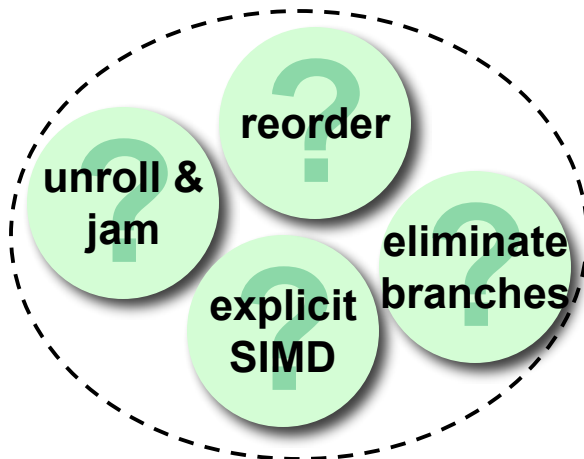
For finite element problem (BCSR)
[Im, Yelick, Vuduc, 2005]



Optimization Categorization

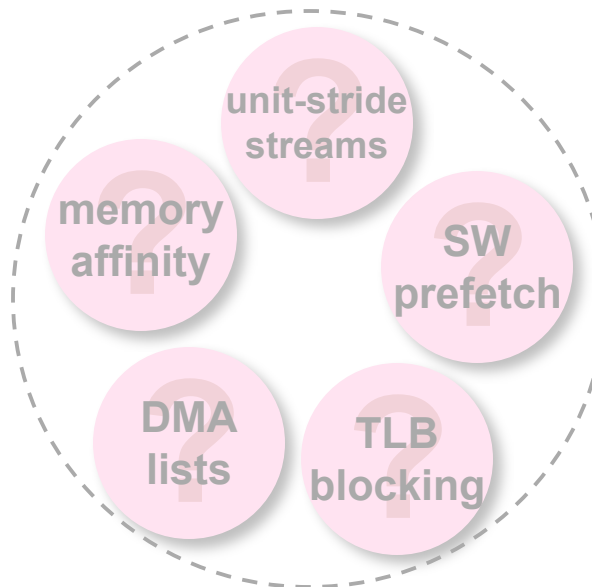
Maximizing In-core Performance

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



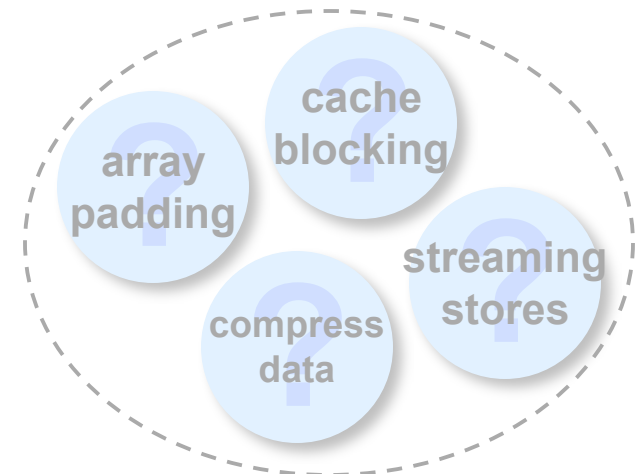
Maximizing Memory Bandwidth

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



Minimizing Memory Traffic

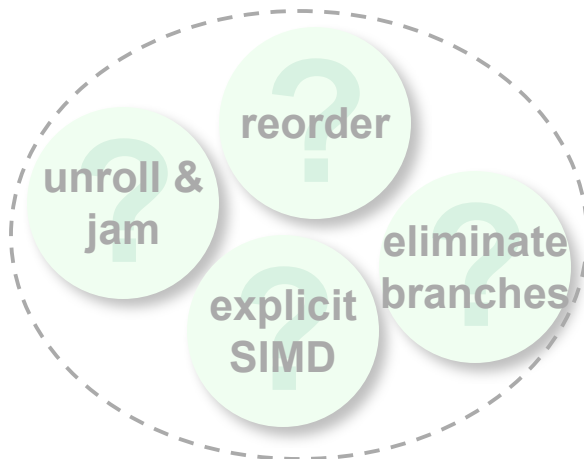
- Eliminate:
- Capacity misses
 - Conflict misses
 - Compulsory misses
 - Write allocate behavior



Optimization Categorization

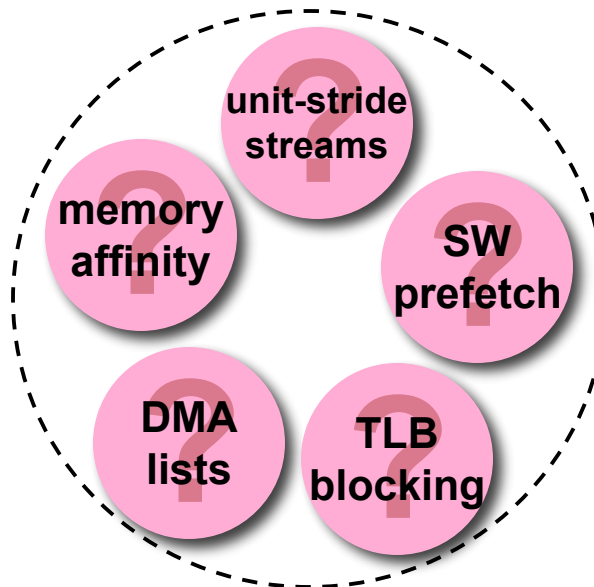
Maximizing In-core Performance

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



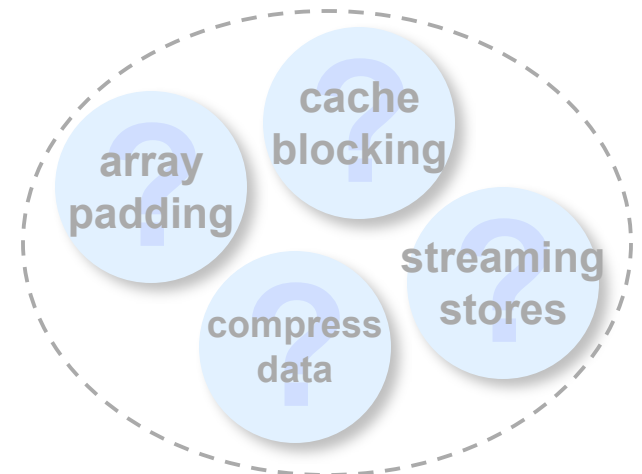
Maximizing Memory Bandwidth

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



Minimizing Memory Traffic

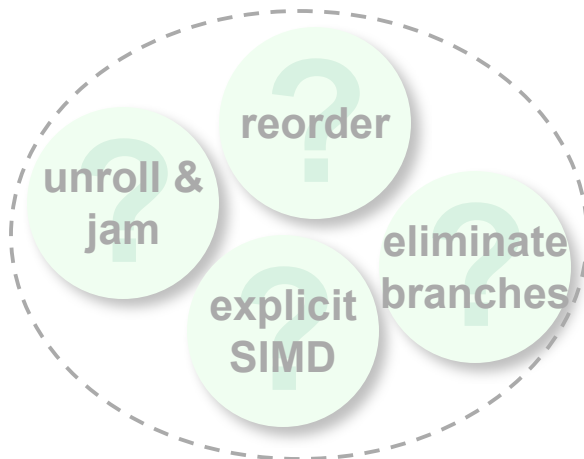
- Eliminate:
- Capacity misses
 - Conflict misses
 - Compulsory misses
 - Write allocate behavior



Optimization Categorization

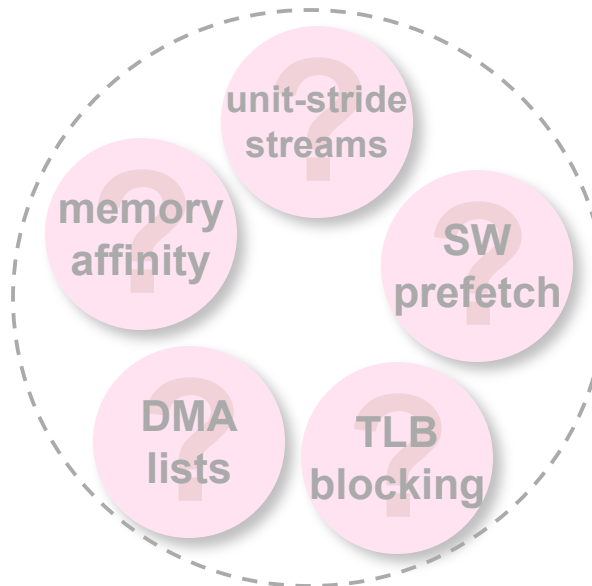
Maximizing In-core Performance

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



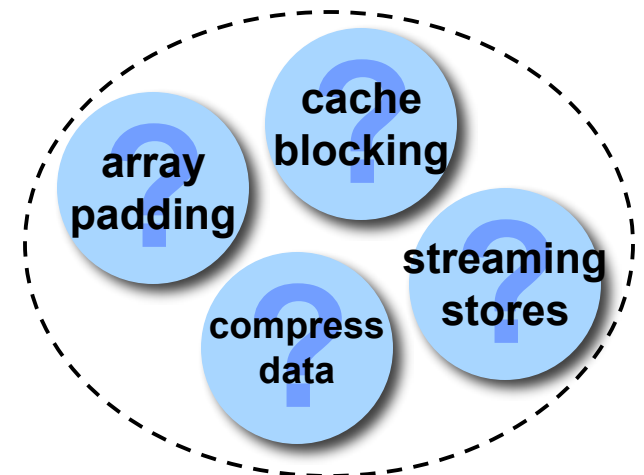
Maximizing Memory Bandwidth

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



Minimizing Memory Traffic

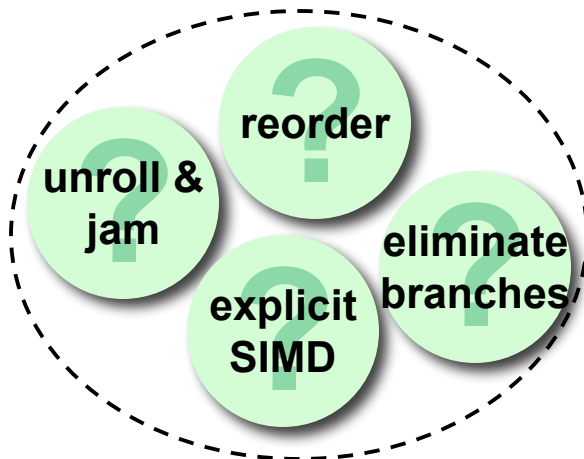
- Eliminate:
- Capacity misses
 - Conflict misses
 - Compulsory misses
 - Write allocate behavior



Optimization Categorization

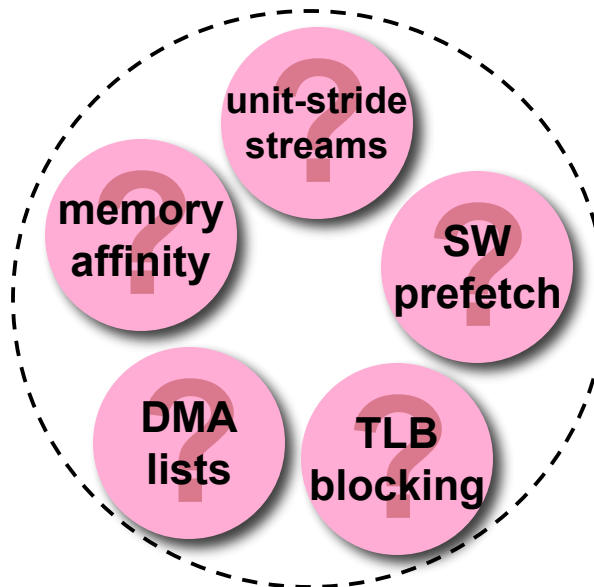
Maximizing In-core Performance

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



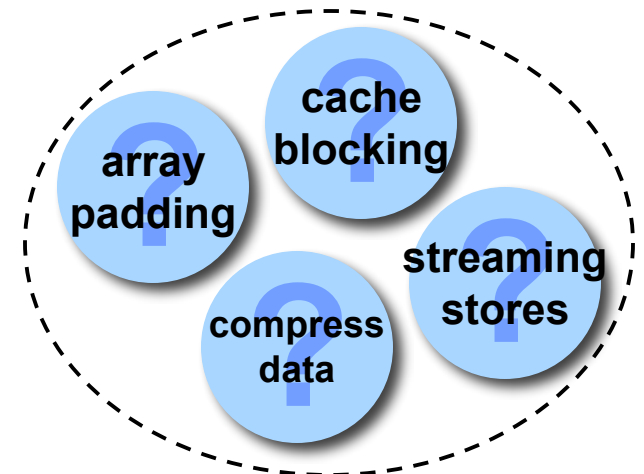
Maximizing Memory Bandwidth

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



Minimizing Memory Traffic

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



Optimization Categorization

Maximizing In-core Performance

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

Maximizing Memory Bandwidth

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

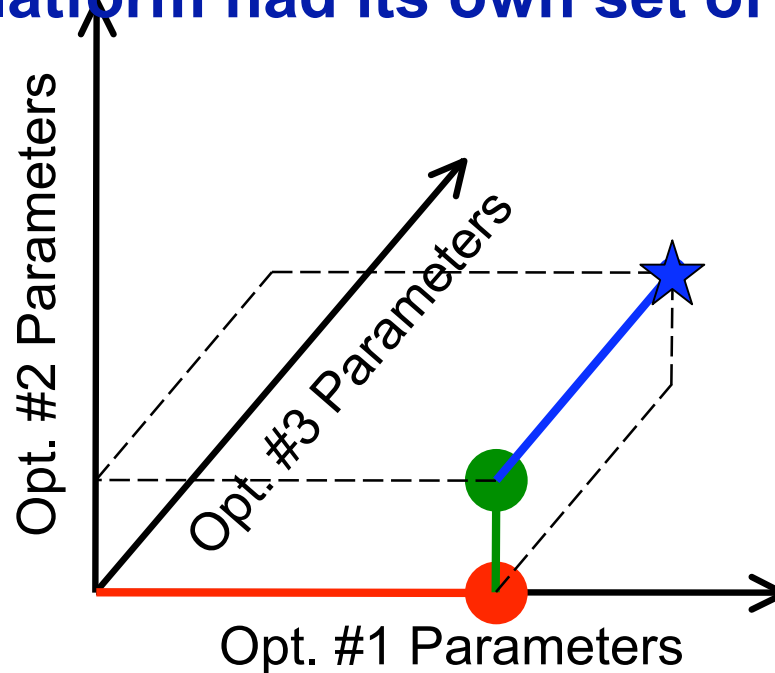
Minimizing Memory Traffic

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

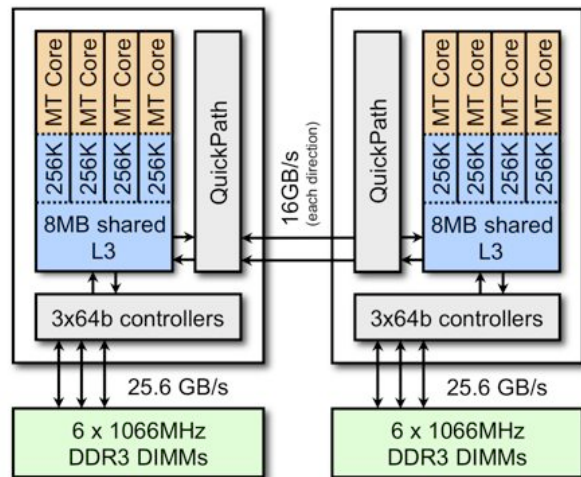


Traversing the Parameter Space

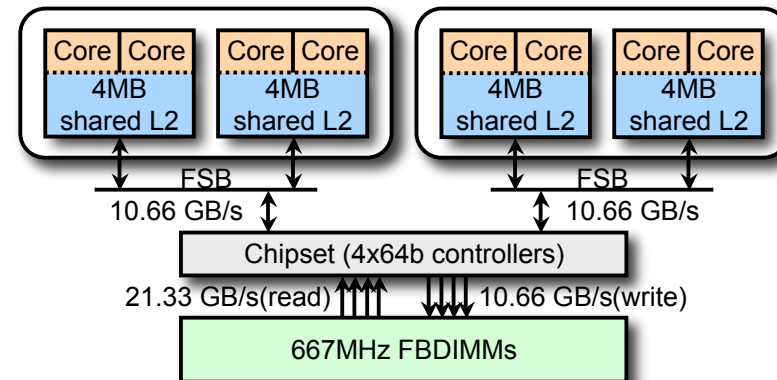
- Exhaustive search of these complex layered optimizations is *impossible*
- To make problem tractable, we:
 - order the optimizations
 - applied them consecutively
- Every platform had its own set of best parameters



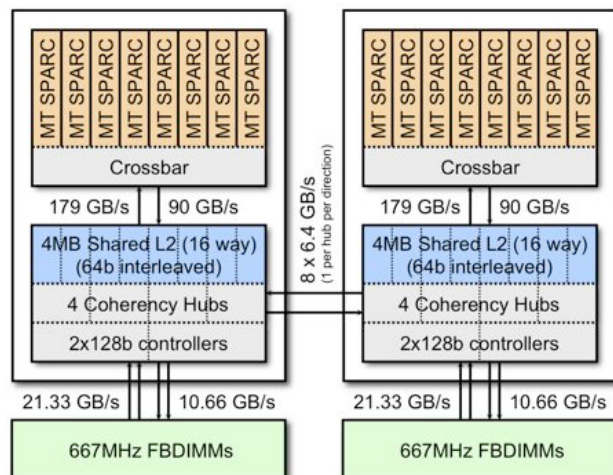
Multicore Architectures



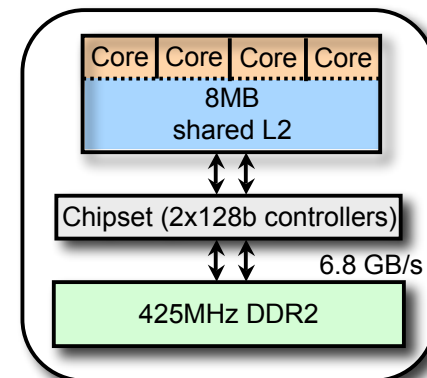
Intel Nehalem (Gainestown)



Intel Clovertown



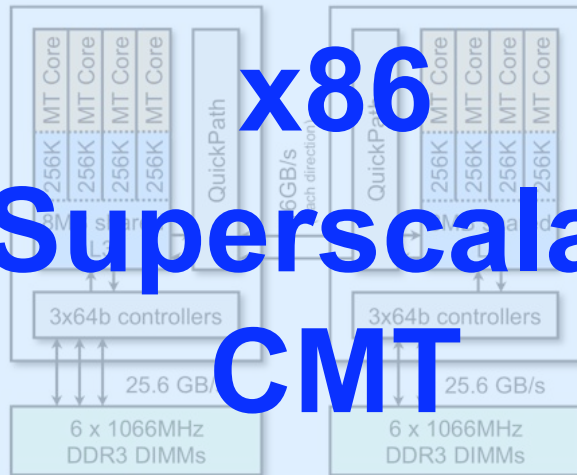
Sun Niagara2 (Victoria Falls)



IBM PPC 450
(BG/P)

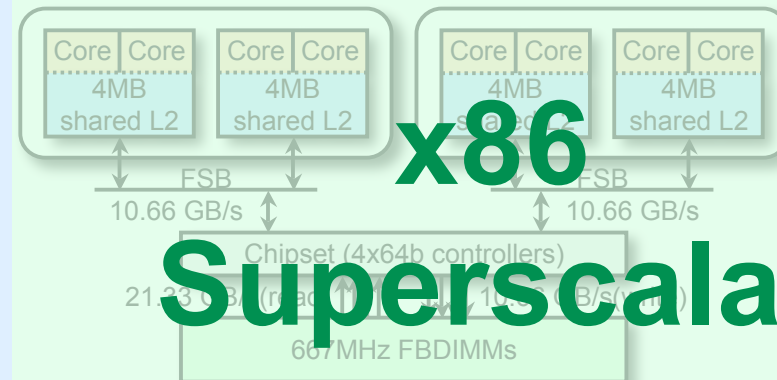
Multicore Architectures

**x86
Superscalar/
CMT**



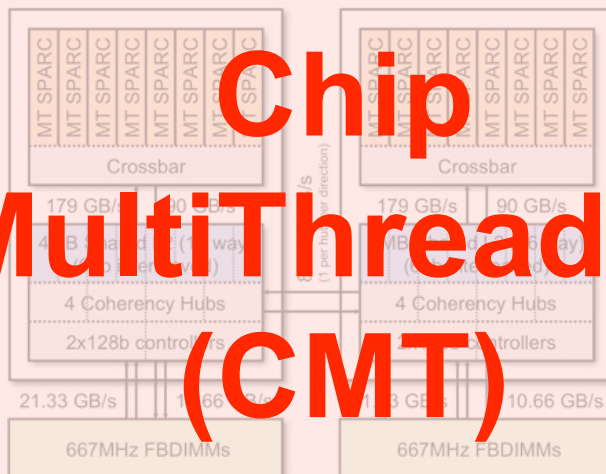
Intel Nehalem (Gainestown)

**x86
Superscalar**



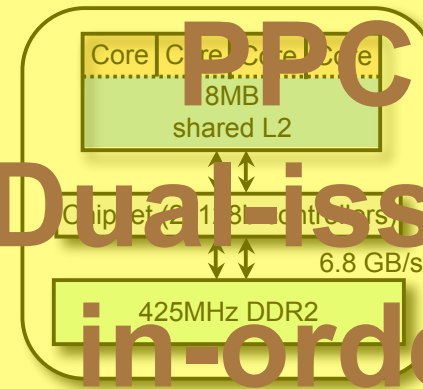
Intel Clovertown

**Chip
MultiThreaded
(CMT)**



Sun Niagara2 (Victoria Falls)

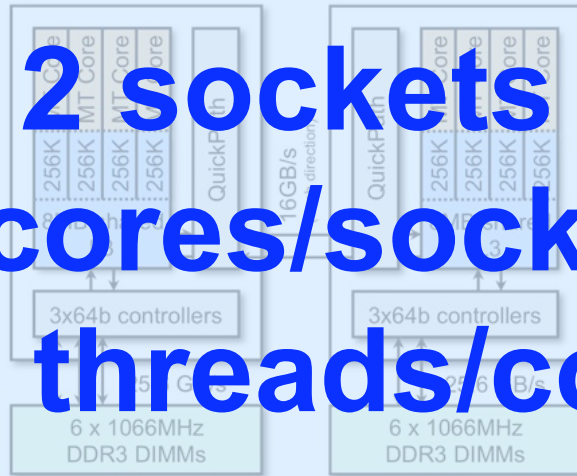
**PPC
Dual-issue
in-order**



IBM PPC 450
(BG/P)

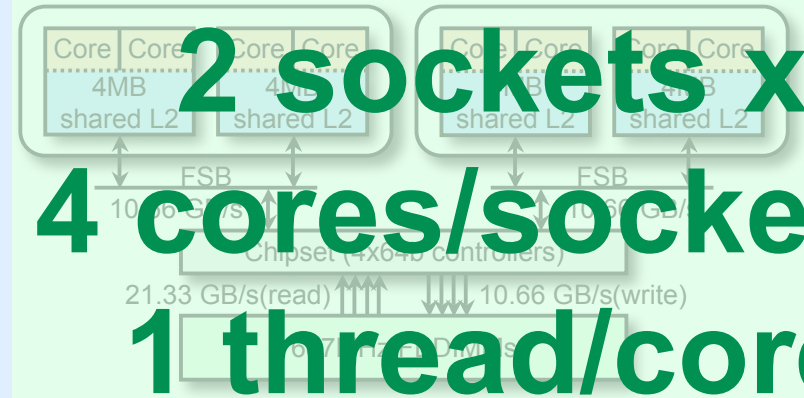
Multicore Architectures

**2 sockets x
4 cores/socket x
2 threads/core**



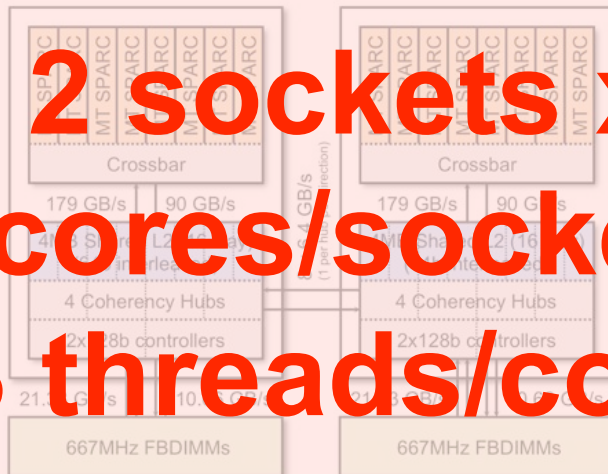
Intel Nehalem (Gainestown)

**2 sockets x
4 cores/socket x
1 thread/core**



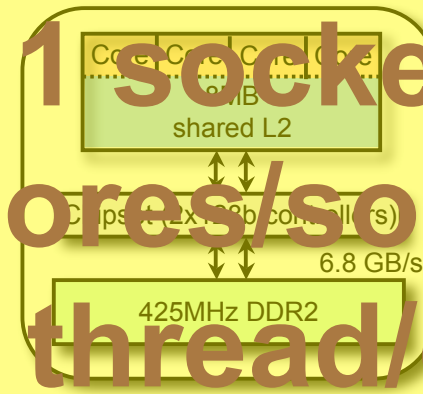
Intel Clovertown

**2 sockets x
8 cores/socket x
8 threads/core**



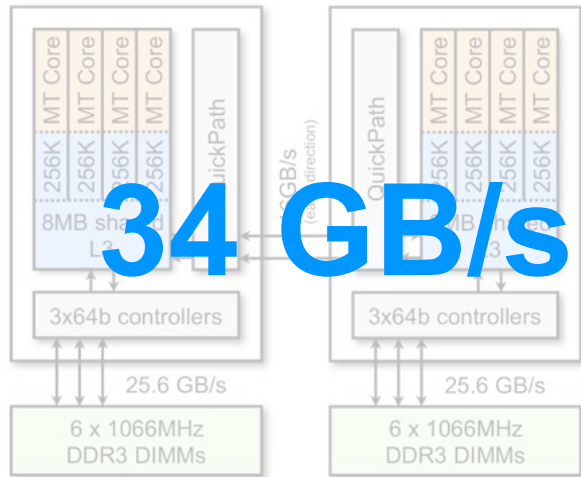
Sun Niagara2 (Victoria Falls)

**1 socket x
4 cores/socket x
1 thread/core**

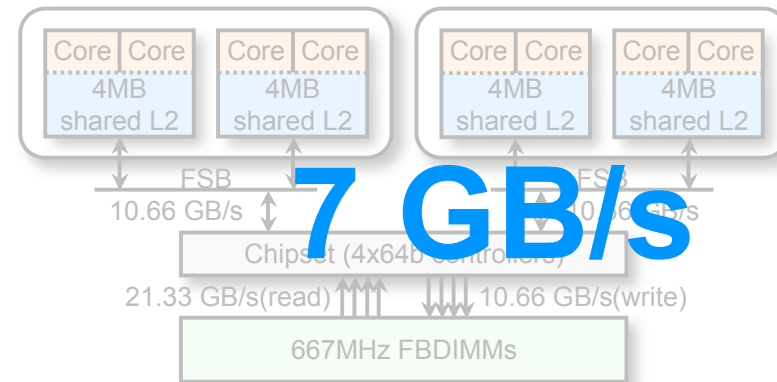


IBM PPC 450
(BG/P)

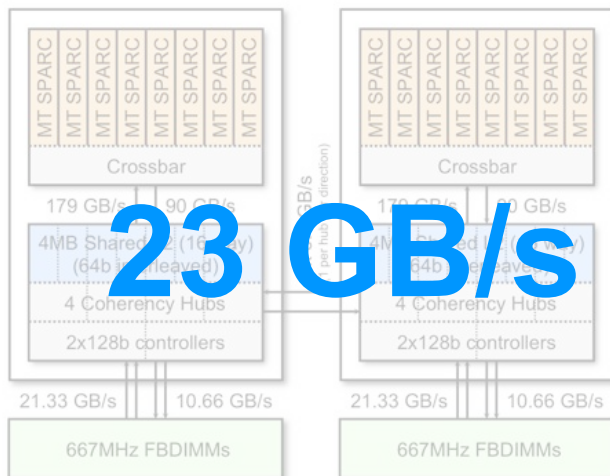
Multicore Architectures



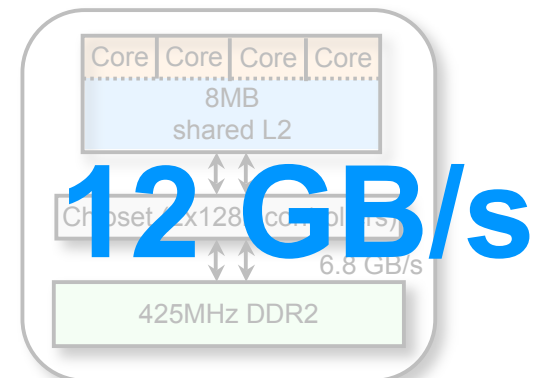
Intel Nehalem (Gainestown)



Intel Clovertown



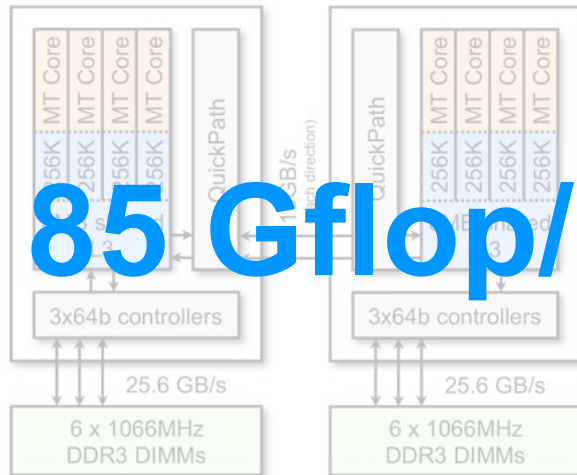
Sun Niagara2 (Victoria Falls)



IBM PPC 450
(BG/P)

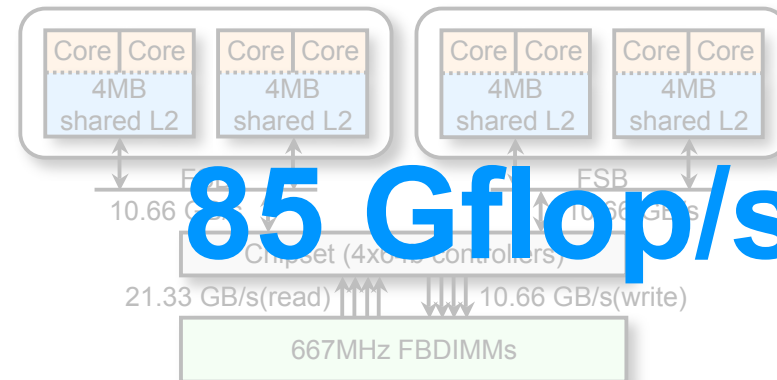
Multicore Architectures

85 Gflop/s



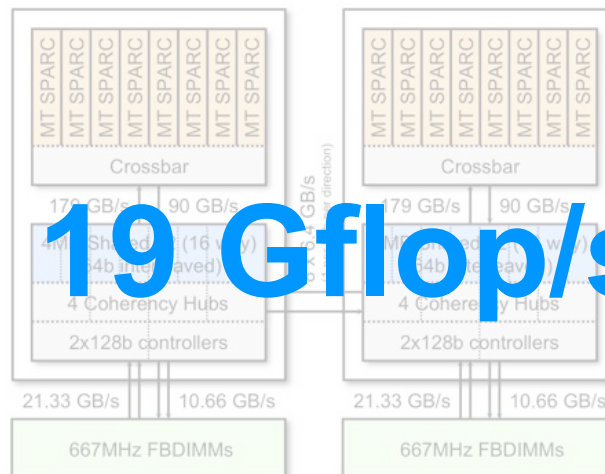
Intel Nehalem (Gainestown)

85 Gflop/s



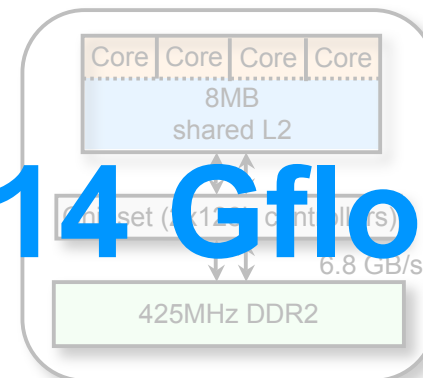
Intel Clovertown

19 Gflop/s



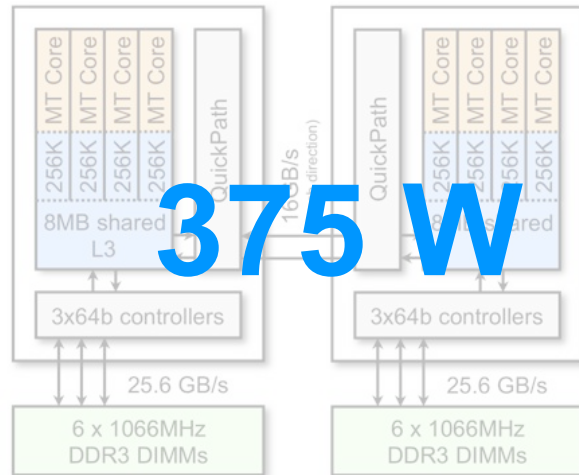
Sun Niagara2 (Victoria Falls)

14 Gflop/s



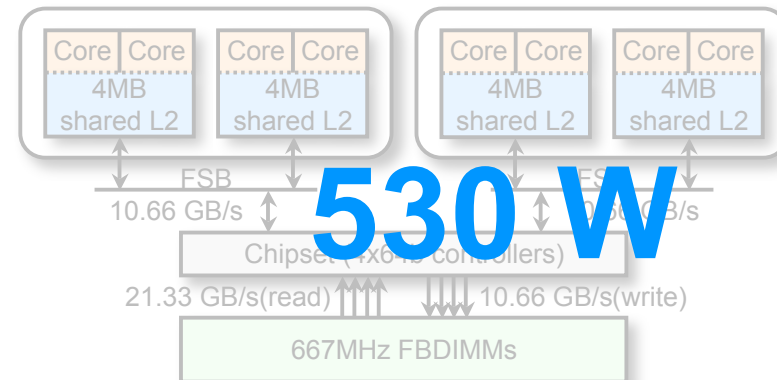
IBM PPC 450
(BG/P)

Multicore Architectures



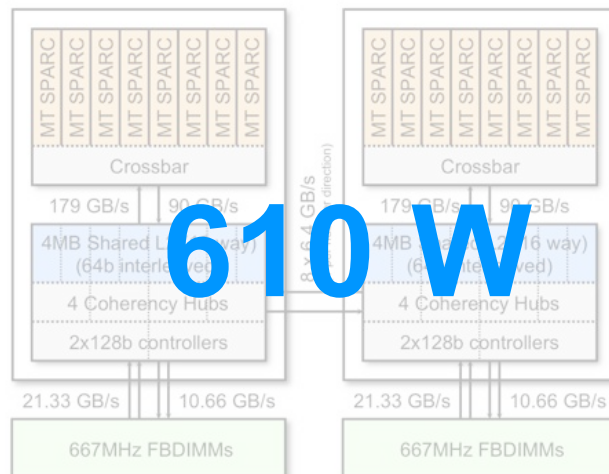
375 W

Intel Nehalem (Gainestown)



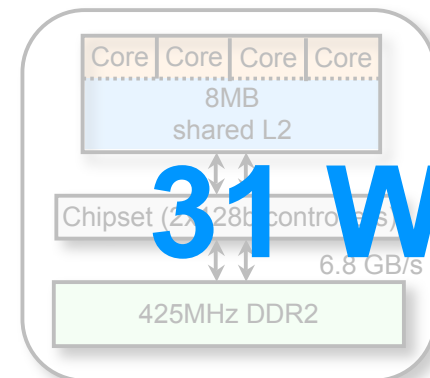
530 W

Intel Clovertown



610 W

Sun Niagara2 (Victoria Falls)

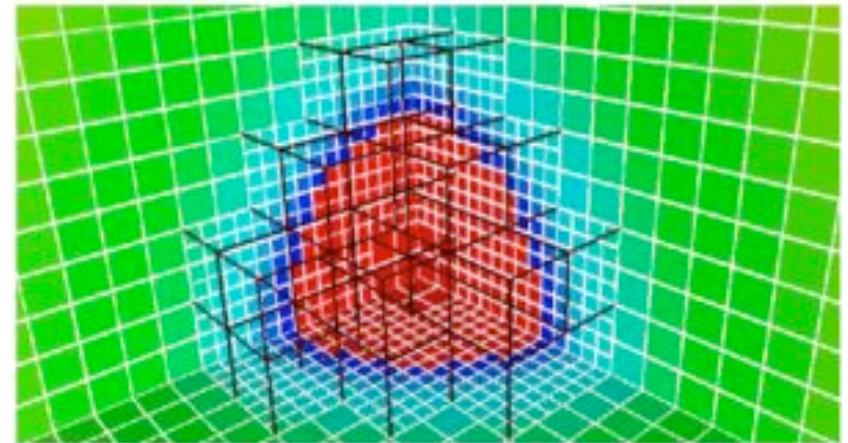
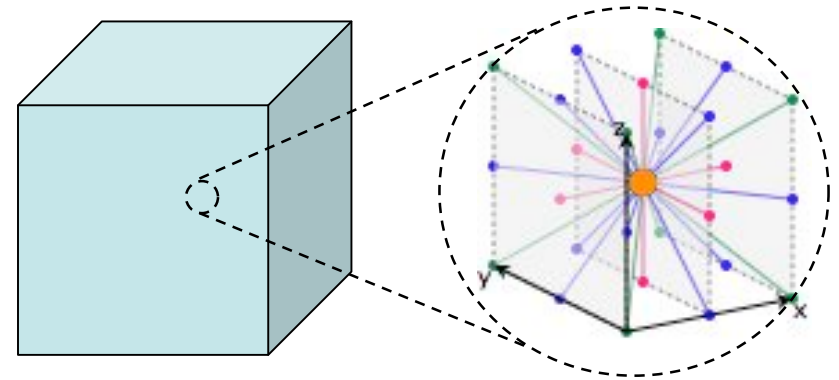


31 W

IBM PPC 450
(BG/P)

Stencil Code Overview

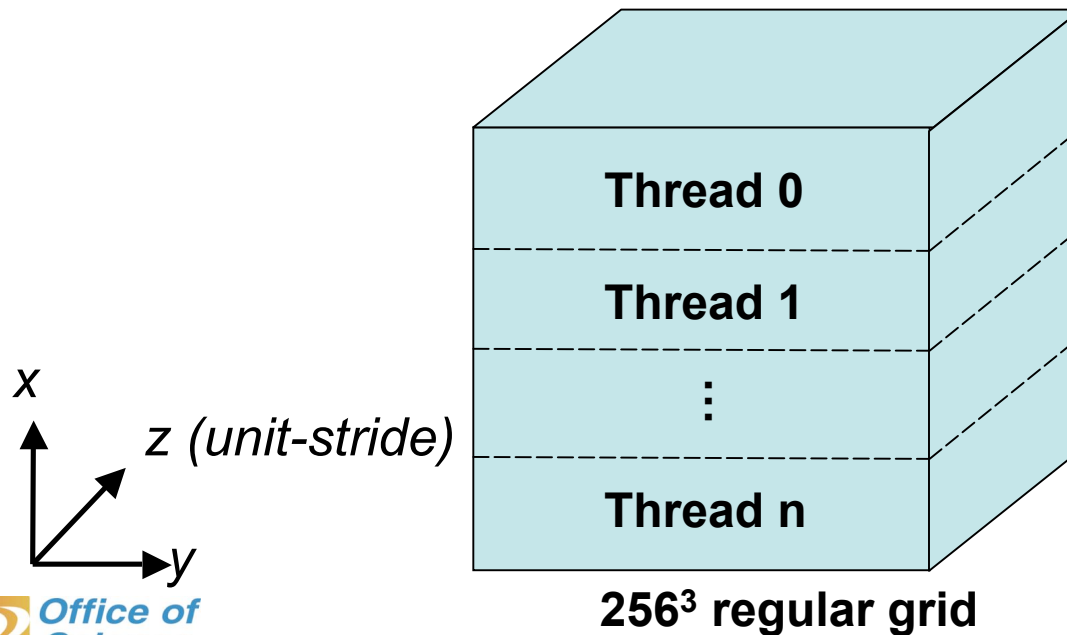
- For a given point, a *stencil* is a fixed subset of nearest neighbors
- A *stencil code* updates every point in a regular grid by “applying a stencil”
- Used in iterative PDE solvers like Jacobi, Multigrid, and AMR
- Focus on a out-of-place 3D 27-point stencil sweeping over a 256^3 grid
 - Problem size > Cache size
- Stencil codes characteristics
 - Long unit-stride memory accesses
 - Some reuse of each grid point
 - 30 flops per grid point
- Arithmetic Intensity 0.75-1.88



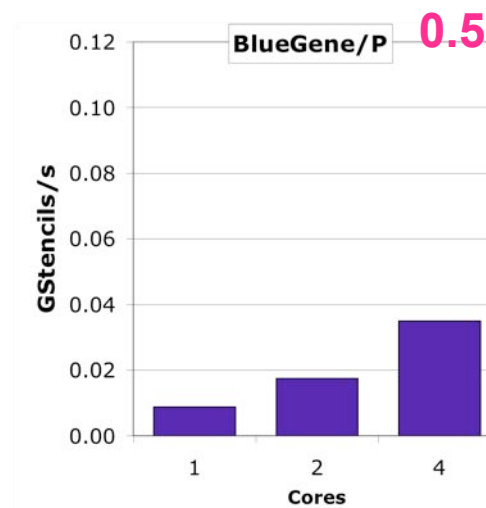
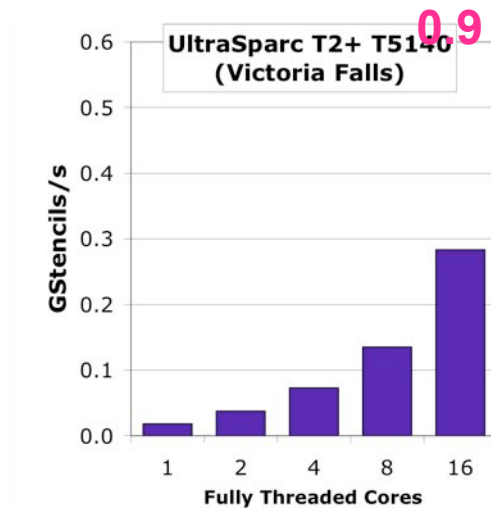
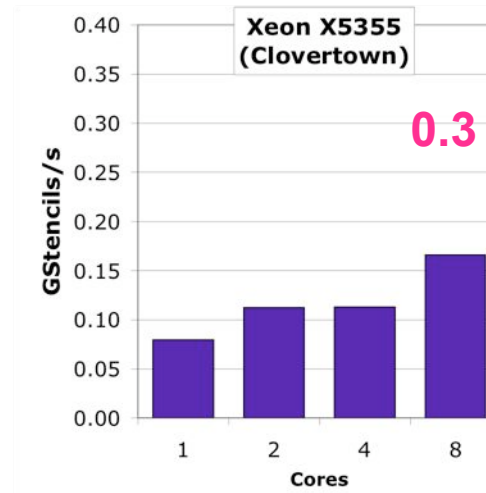
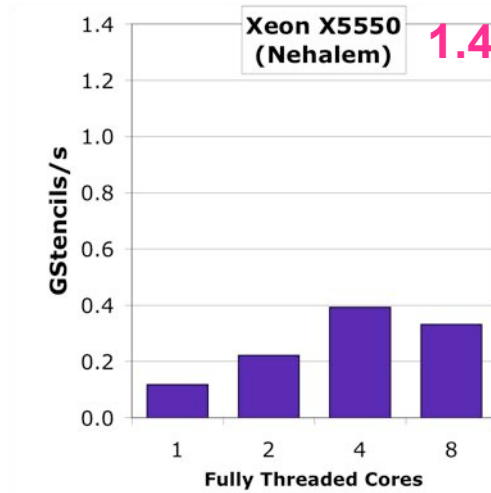
Adaptive Mesh Refinement (AMR)

Naïve Stencil Code

- We wish to exploit multicore resources
- Simple parallel stencil code:
 - Use pthreads
 - Parallelize in least contiguous grid dimension
 - Thread affinity for scaling: multithreading, then multicore, then multisocket

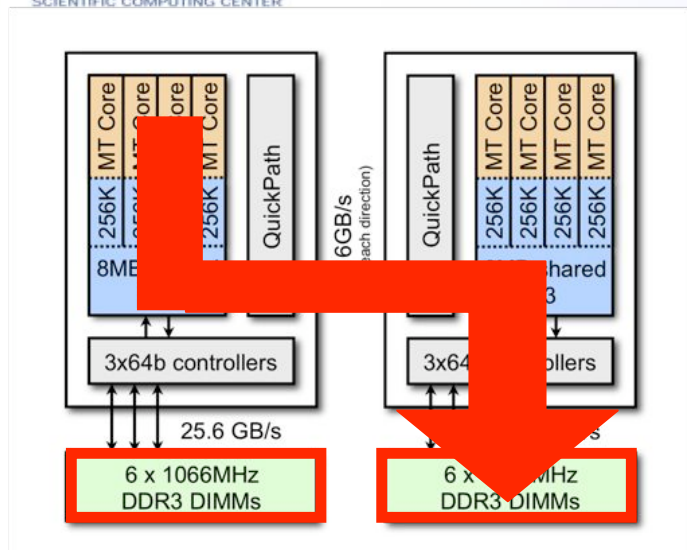


Naïve Performance

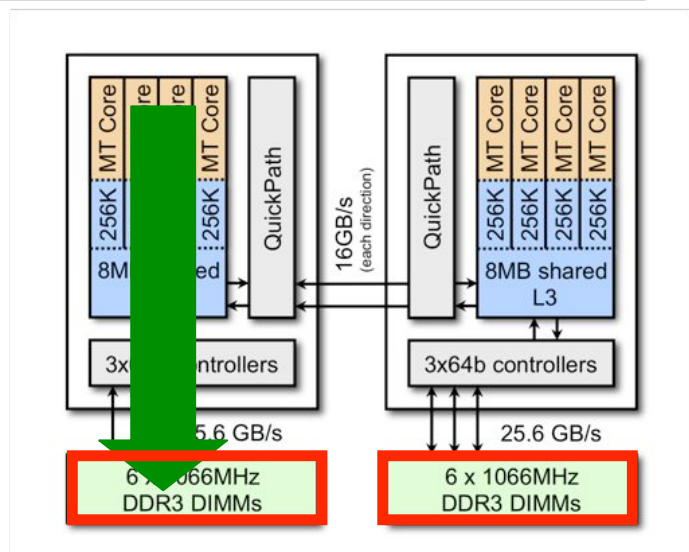


- Compiler delivers poor performance
 - icc for Intel
 - gcc for VF
 - xlc for BG/P
- No parallel scaling for two architectures
- Low performance as compared with stream bandwidth prediction
 - Reasonably high AI means that other bottlenecks likely exist

NUMA Optimization

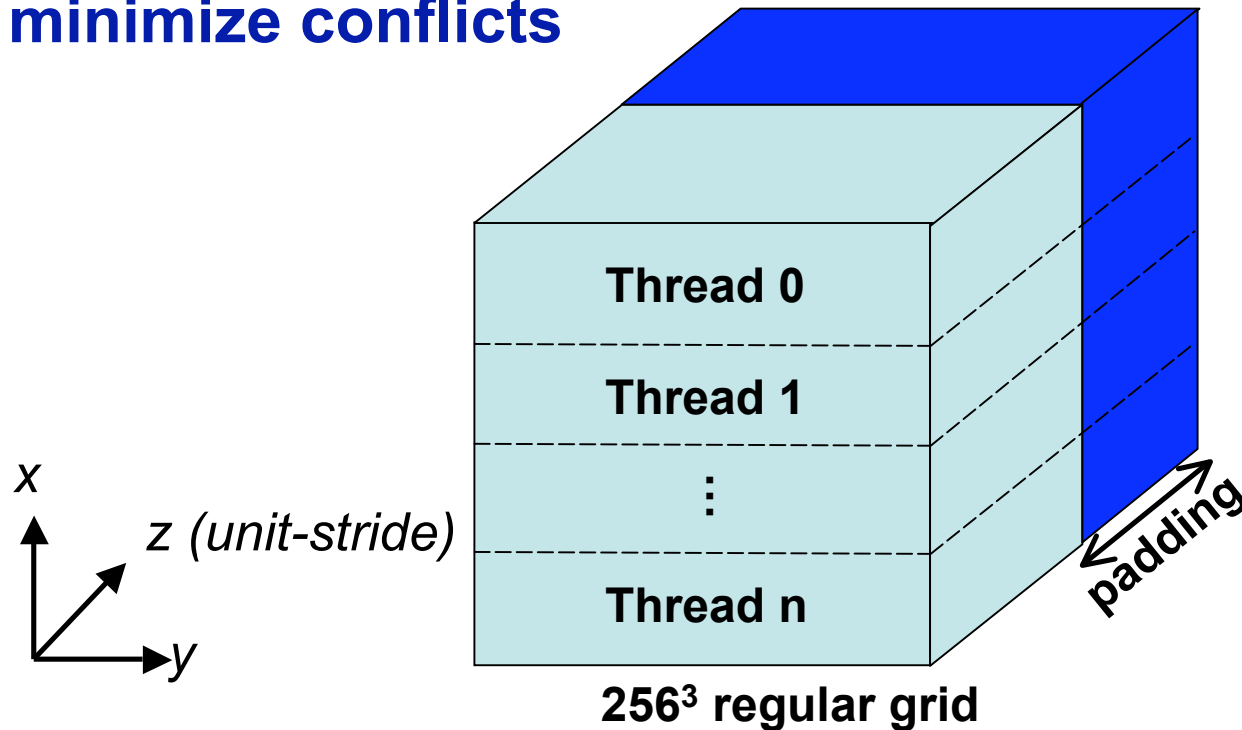


- ❖ All DRAMs are highlighted in red
- ❖ Co-located data on same socket as thread processing it

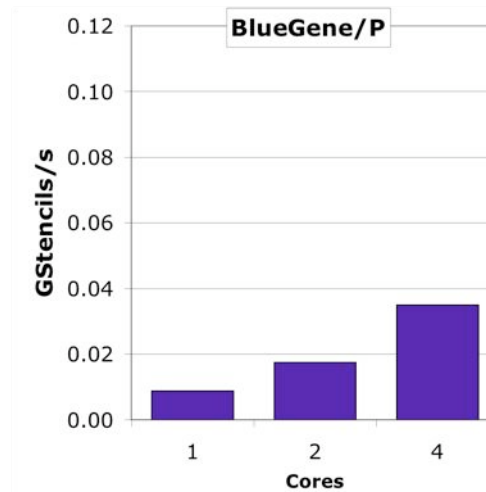
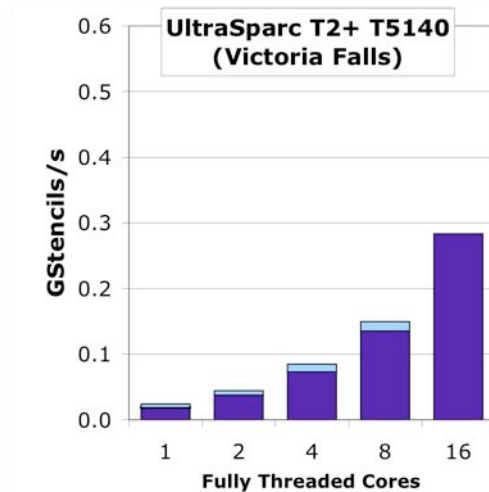
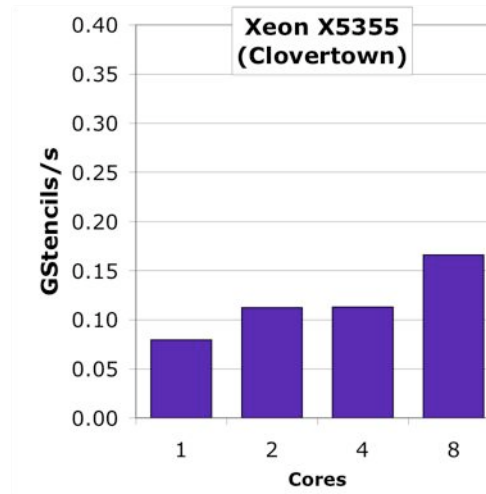
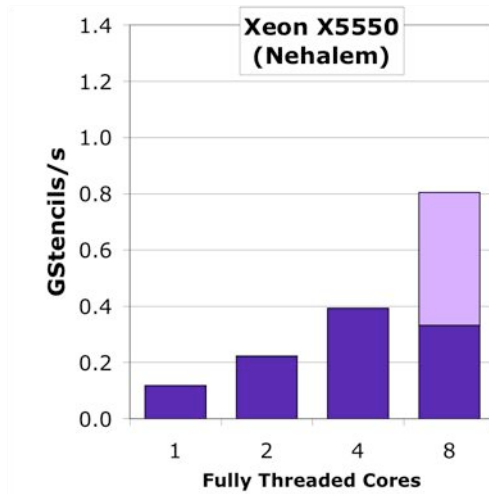





Array Padding Optimization

- Conflict misses may occur on low-associativity caches
- Each array was padded by a tuned amount to minimize conflicts

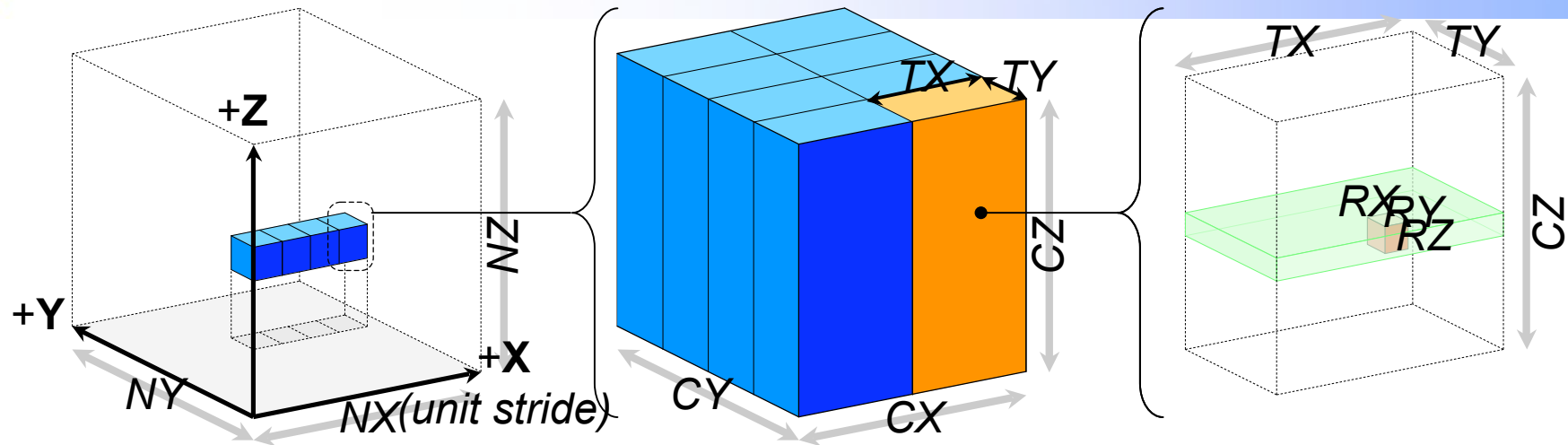


Performance



-  + Array Padding
-  + NUMA
-  Naive

Problem Decomposition



Decomposition of the Grid into a Chunk of Core Blocks

- Large chunks enable efficient NUMA Allocation
- Small chunks exploit LLC shared caches

Decomposition into Thread Blocks

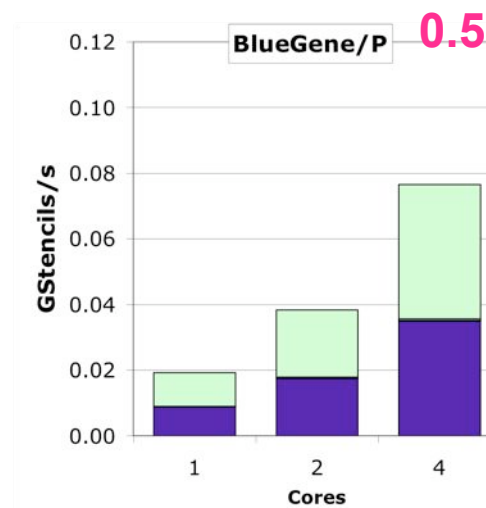
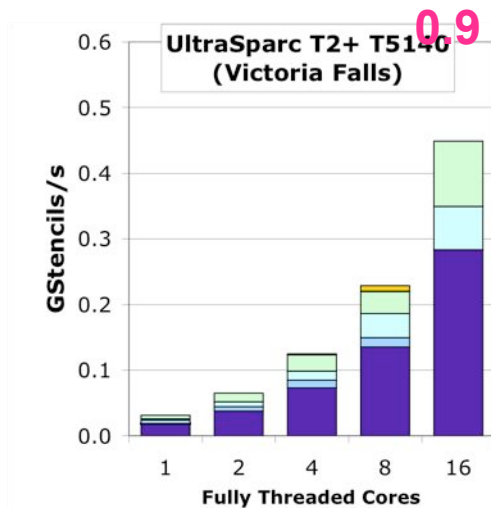
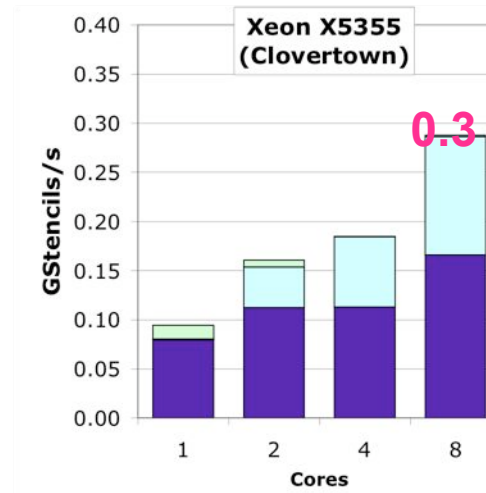
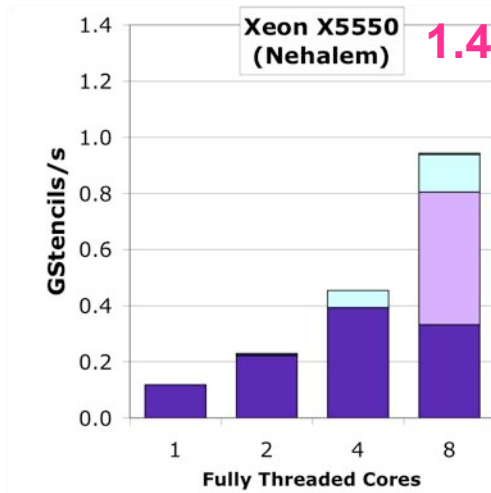
- Exploit caches shared among threads within a core

Decomposition into Register Blocks

- Make DLP/ILP explicit
- Make register reuse explicit

- This decomposition is universal across *all* examined architectures
- Decomposition does **not** change data structure
- Need to choose best block sizes for each hierarchy level

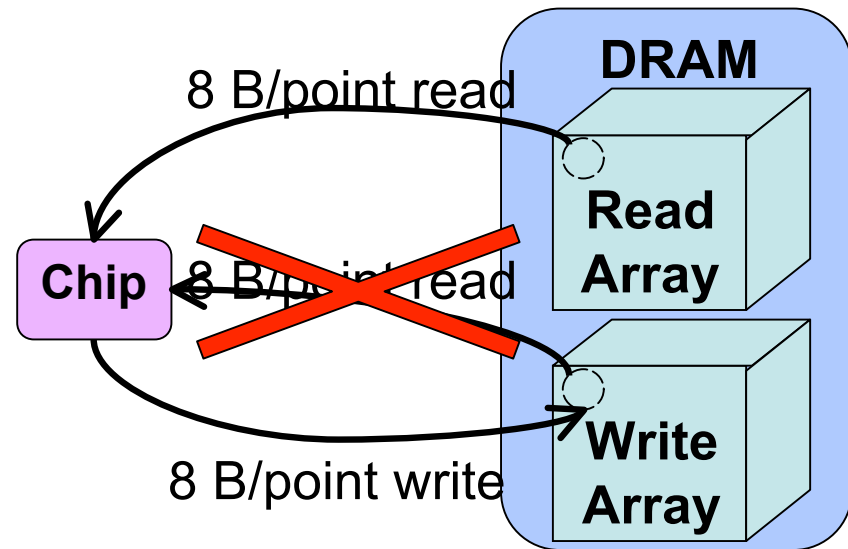
Performance



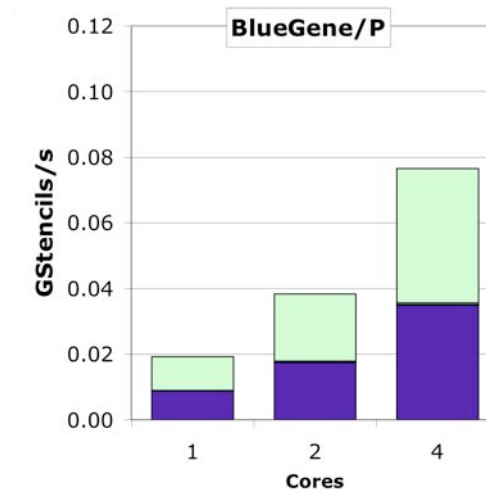
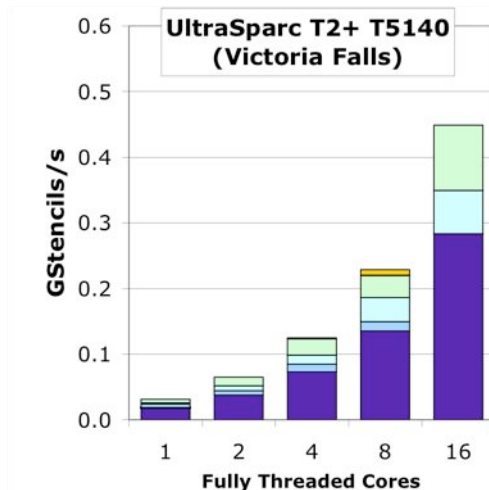
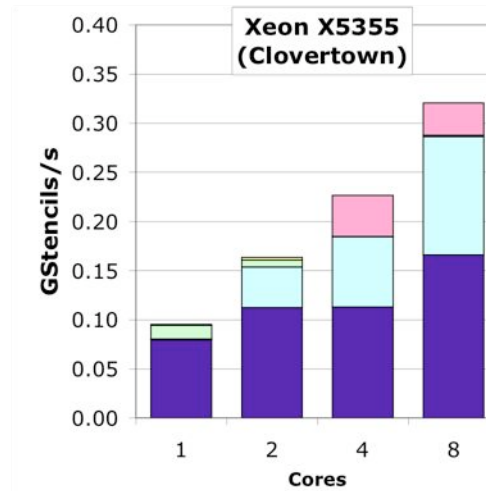
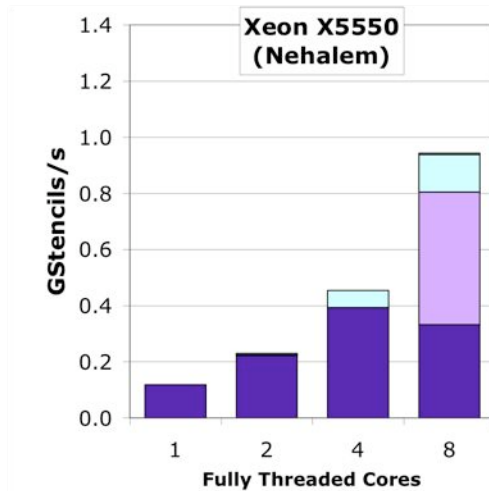
- + Thread Blocking
- + Register Blocking
- + Core Blocking
- + Array Padding
- + NUMA
- Naive

ISA Specific Optimizations

- Software prefetch
- Explicit SIMD
 - PPC SIMD loads do not improve performance due to unaligned data
- Cache Bypass
 - Initial values in write array not used
 - Eliminate write array cache fills with intrinsics
 - Reduces memory traffic from 24 B/point to 16 B/point



Performance

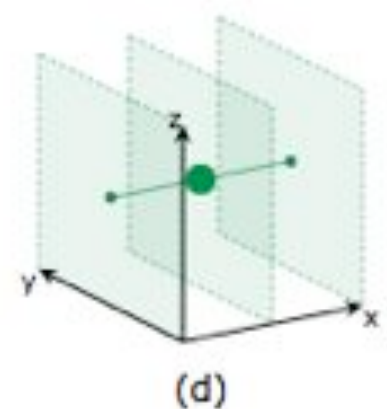
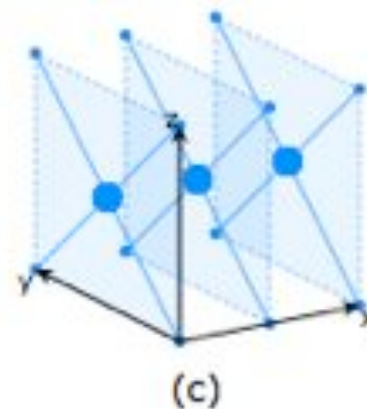
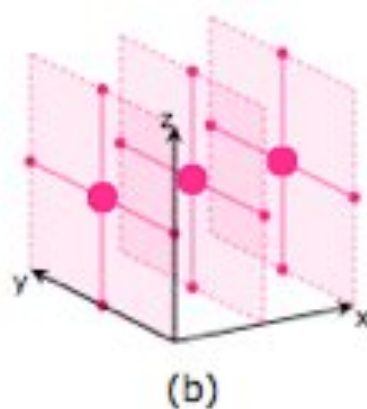
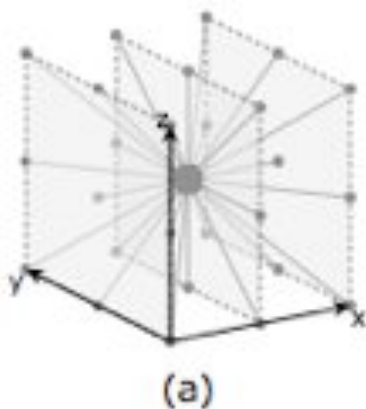


• Optimizations effect architectures in different ways

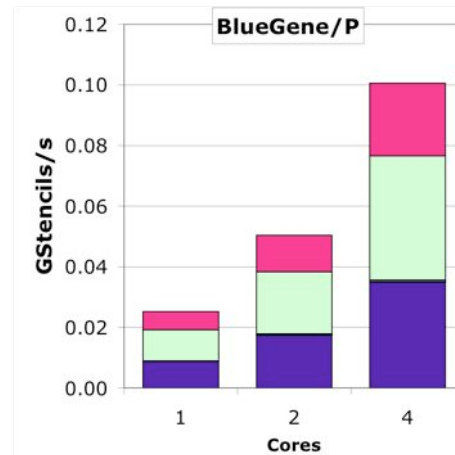
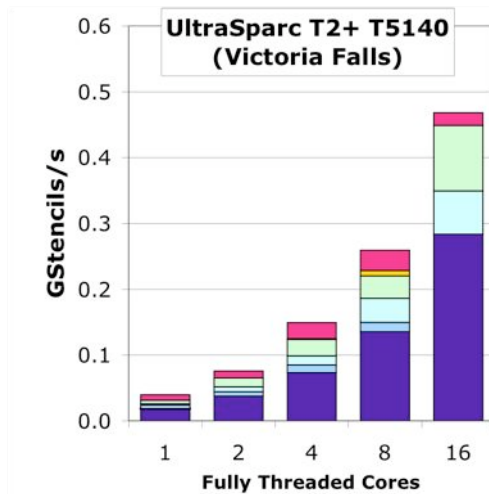
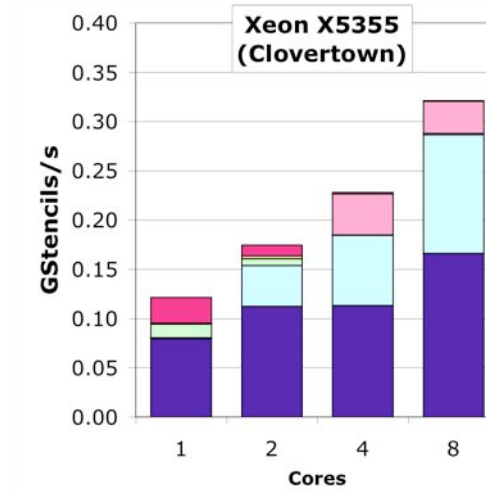
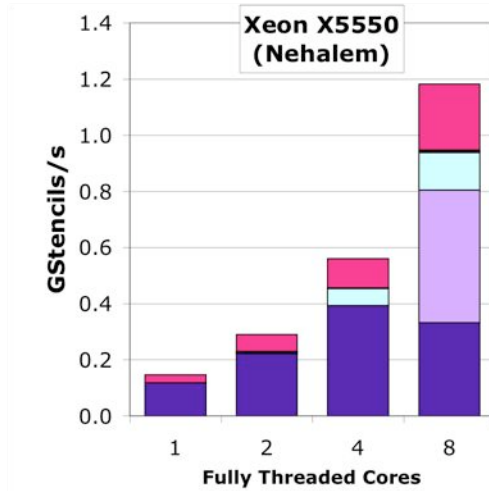
- + Cache Bypass
- + SIMD
- + Software Prefetch
- + Thread Blocking
- + Register Blocking
- + Core Blocking
- + Array Padding
- + NUMA
- Naive

Common Subexpression Elimination Optimization

- Common computation exists between different stencil updates
- Compiler does not recognize this
- Reduce number of flops from 30 to 18



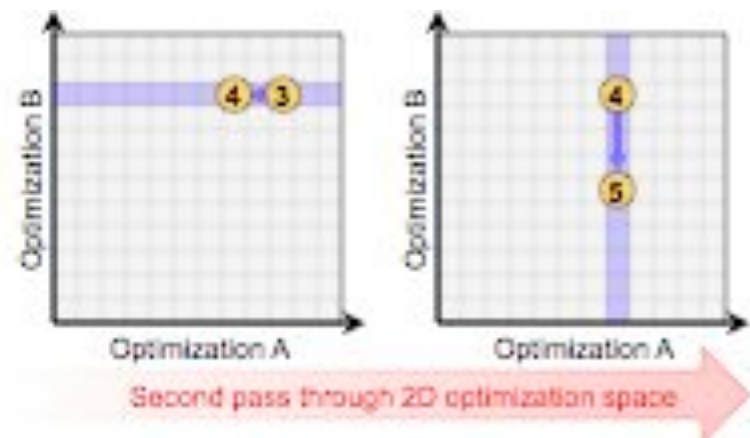
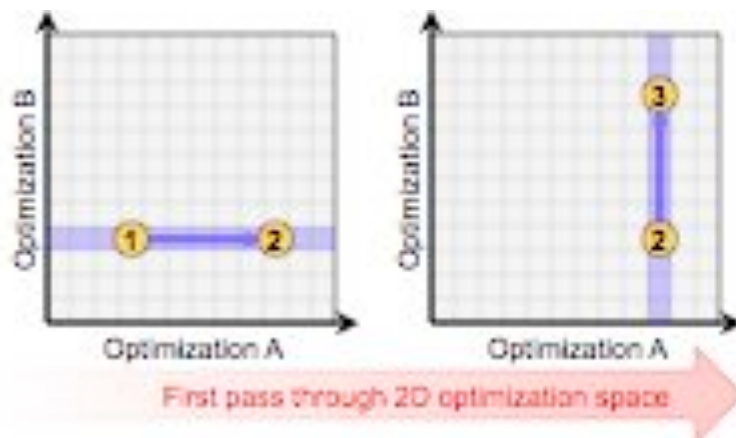
CSE Version Performance



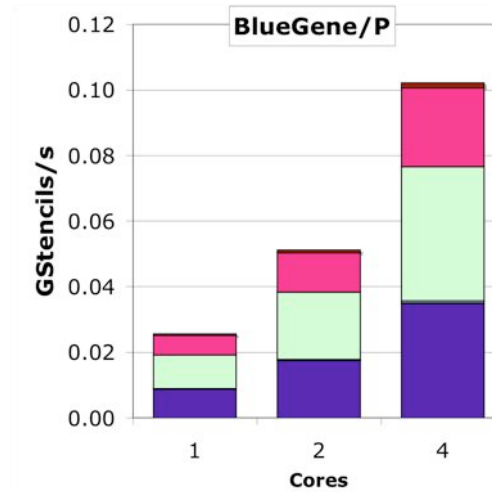
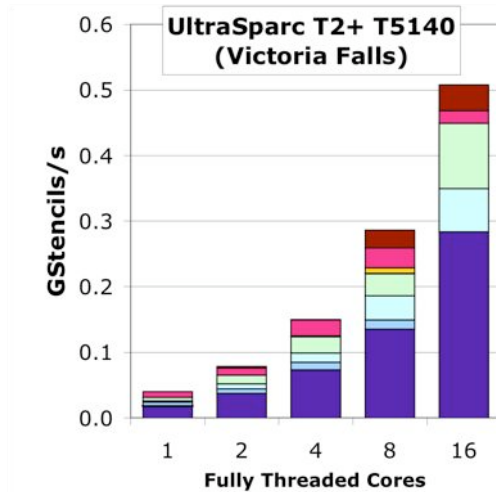
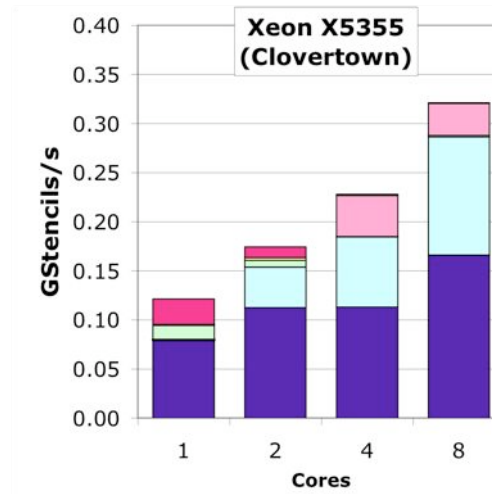
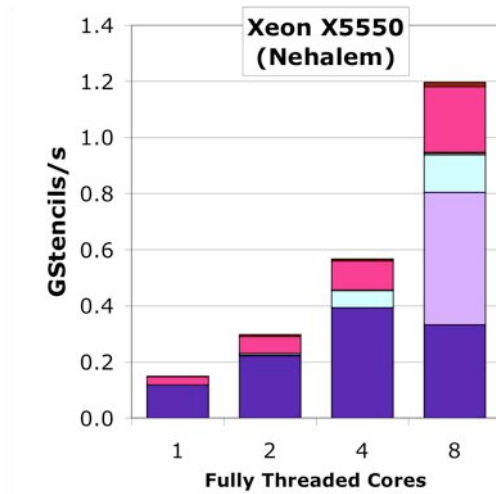
- + CSE
- + Cache Bypass
- + SIMD
- + Software Prefetch
- + Thread Blocking
- + Register Blocking
- + Core Blocking
- + Array Padding
- + NUMA
- Naive

Is Performance Acceptable?

- A model (e.g. Roofline) could be used to predict best performance
- Use a two-pass greedy algorithm

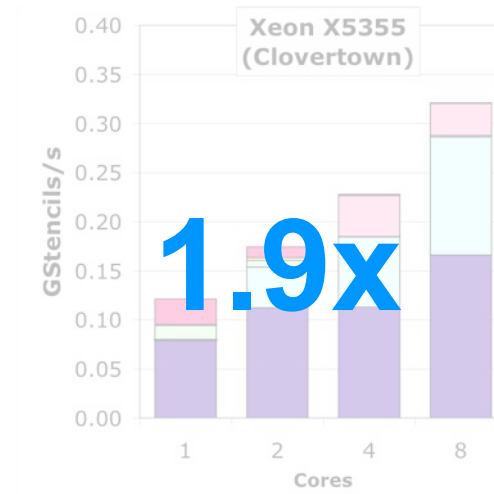
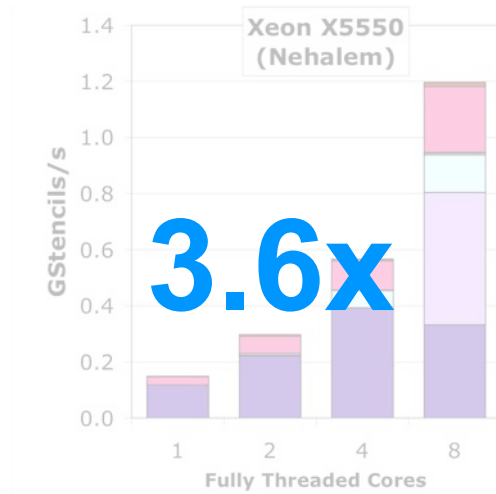


Second Pass Performance

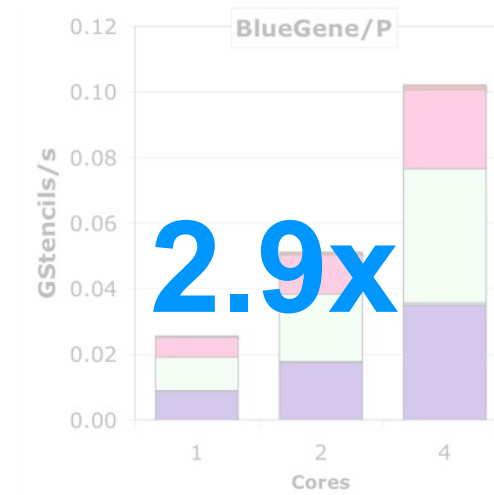
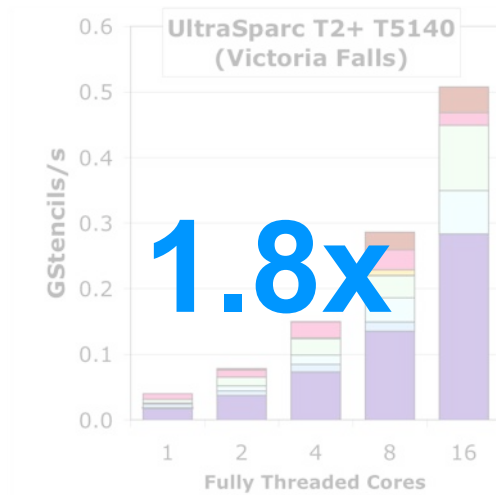


- + Second Pass
- + CSE
- + Cache Bypass
- + SIMD
- + Software Prefetch
- + Thread Blocking
- + Register Blocking
- + Core Blocking
- + Array Padding
- + NUMA
- Naive

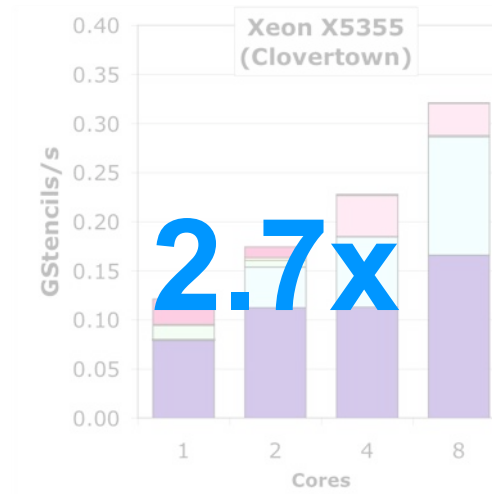
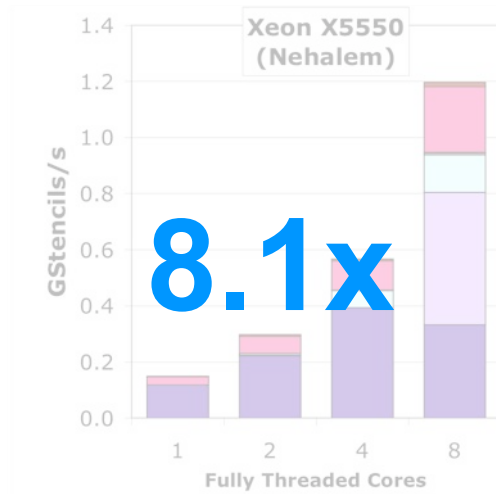
Tuning Speedup



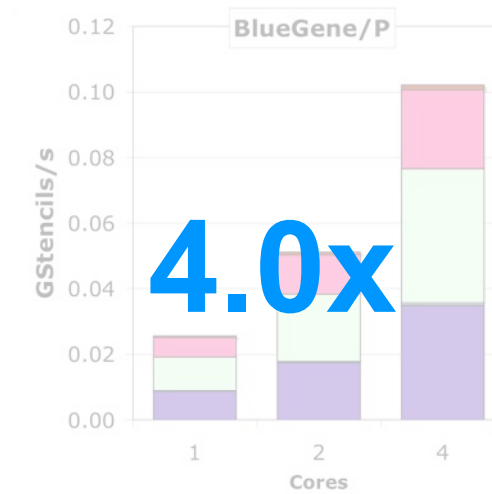
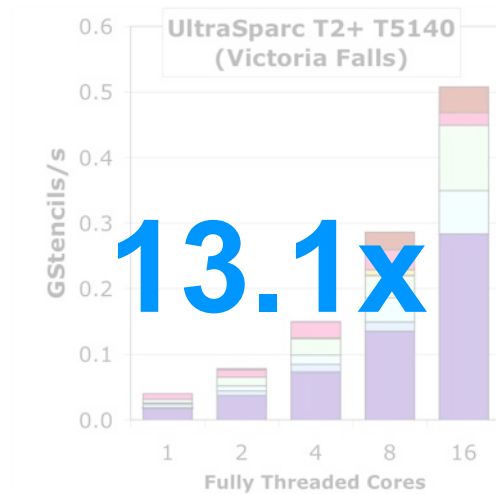
- Speedup at maximum concurrency



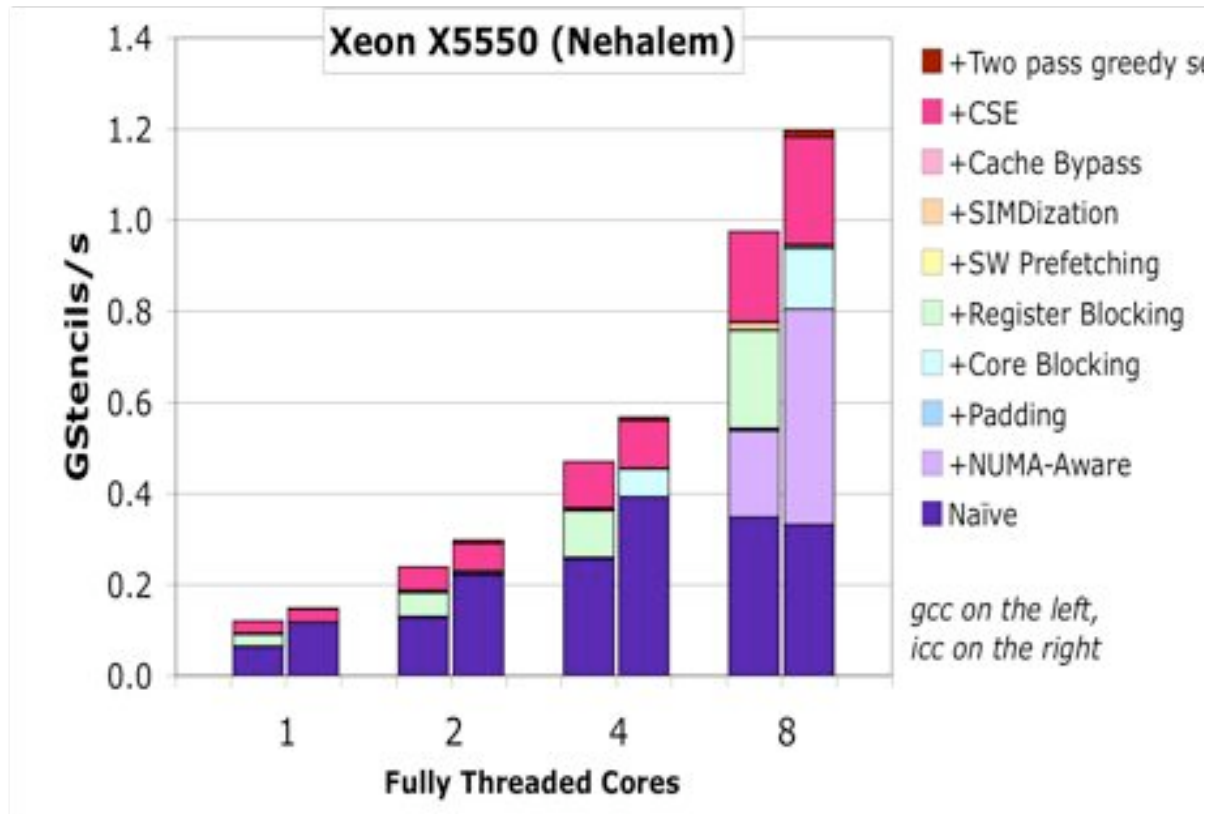
Parallel Speedup



- Speedup going from a single core to maximum concurrency
- All architectures now scale

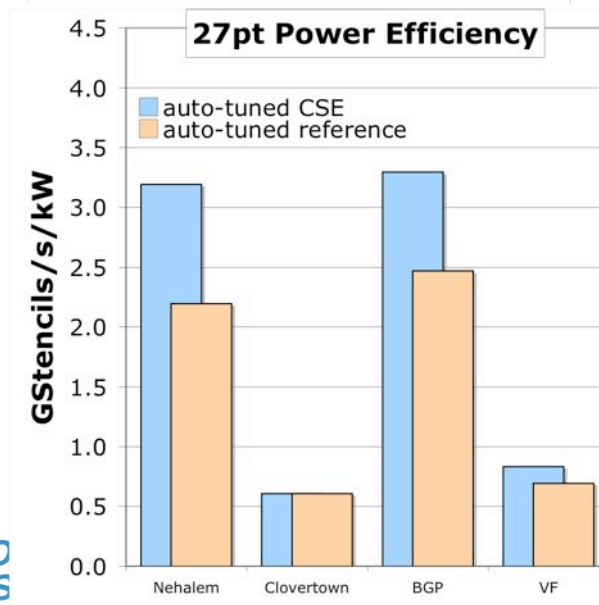
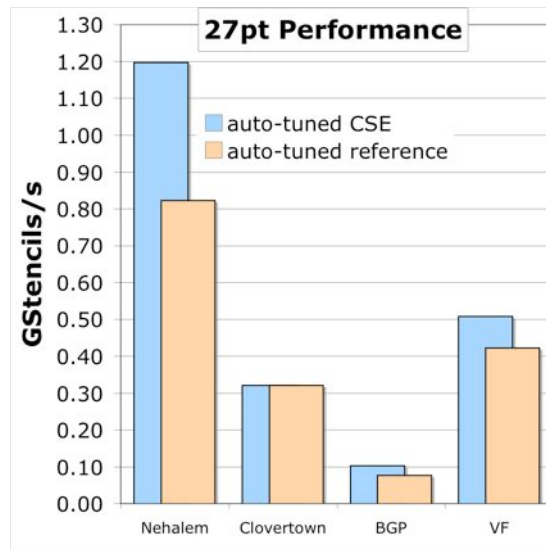


Effect of compilers



- **icc is consistently better than gcc**
- **For single socket gcc + register blocking has equivalent performance to icc**
- **Core blocking improves icc performance, but not gcc**
 - **Inferior code generation hides memory bottleneck?**

Performance Comparison



- Intel Nehalem best in absolute performance
- Normalize for low power, BG/P solution is much more attractive

Conclusions

- **Compiler alone achieves poor performance**
 - Low fraction possible performance
 - Often no parallel scaling
- **Autotuning is essential to achieving good performance**
 - 1.8x-3.6x speedups across diverse architectures
 - Automatic tuning is *necessary* for scalability
 - Most optimization with the same code base
- **Clovertown required SIMD (hampers productivity) for best performance**
- **When power consumption is taken into account, BG/P performs well**



Acknowledgements

- **UC Berkeley**
 - RADLab Cluster (Nehalem)
 - PSI cluster(Clovertown)
- **Sun Microsystems**
 - Niagara2 donations
- **ASCR Office in the DOE Office of Science**
 - contract DE-AC02-05CH11231