EECS
Electrical Engineering and Computer Sciences

BERKELEY PAR LAB

PARALLEL COMPUTING LABORATORY

Auto-tuning Stencil Codes for Cache-Based Multicore Platforms

Kaushik Datta

Dissertation Talk

December 4, 2009

Motivation

- Multicore revolution has produced wide variety of architectures
- Compilers alone fail to fully exploit multicore resources
- Hand-tuning has become infeasible
- *We need a better solution!*

Contributions

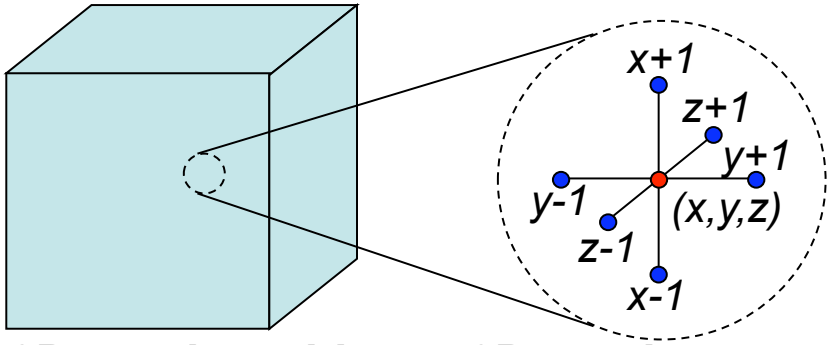
- We have created an automatic stencil tuner (auto-tuner) that achieves up to 5.4x speedups over naïvely threaded stencil code
- We have developed an “Optimized Stream” benchmark for determining a system’s highest attainable memory bandwidth
- We have bound stencil performance using the Roofline Model and in-cache performance

Outline

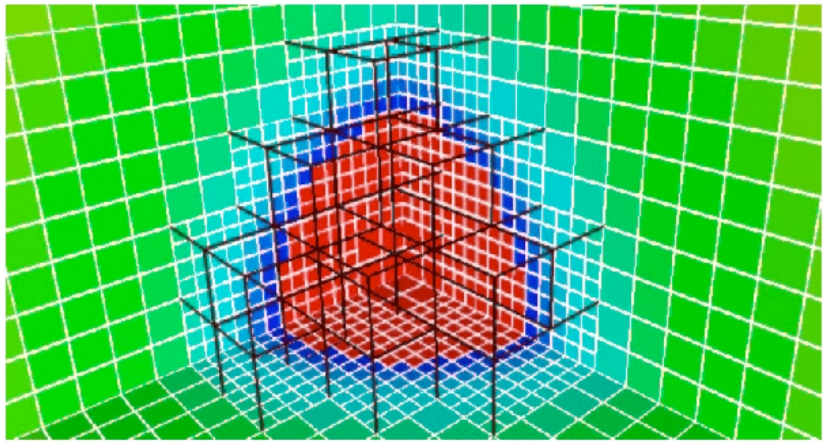
- Stencil Code Overview
- Cache-based Architectures
- Auto-tuning Description
- Stencil Auto-tuning Results

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
- Also used in areas like image processing and geometric modeling
- This talk will focus on three stencil kernels:
  - 3D 7-point stencil
  - 3D 27-point stencil
  - 3D Helmholtz kernel



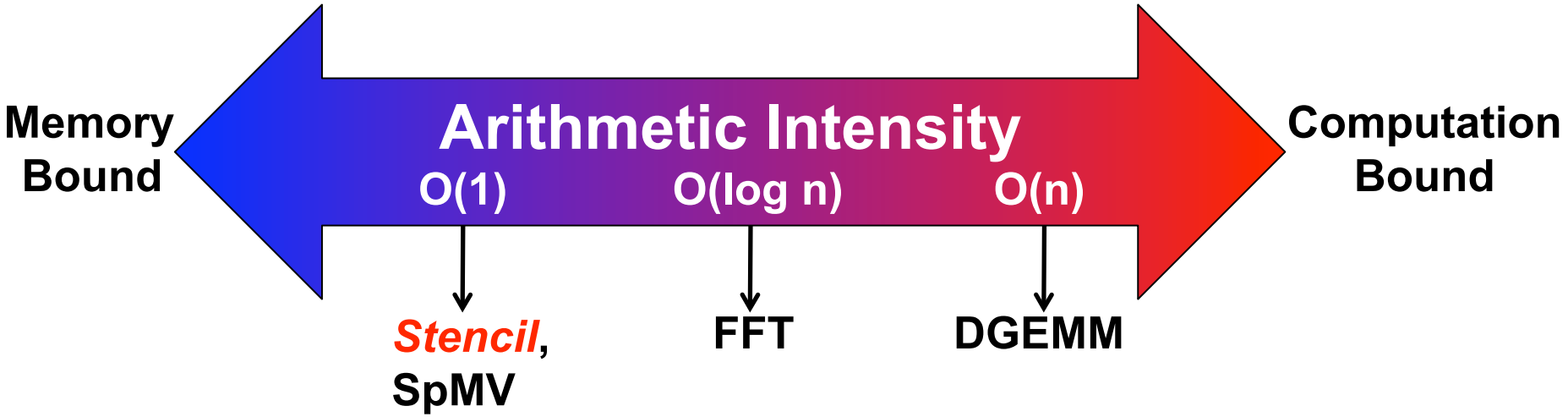
3D regular grid **3D 7-point stencil**



Adaptive Mesh Refinement (AMR)

Arithmetic Intensity
(Ratio of flops to DRAM bytes)

- AI is rough indicator of whether kernel is memory or compute-bound
- Counting *only* compulsory misses:

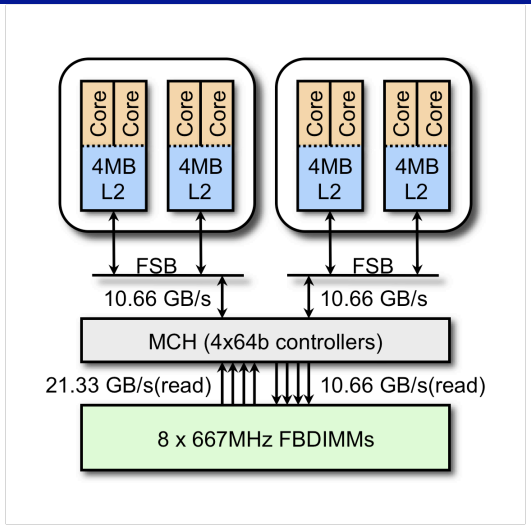


- Stencil codes usually (but not always) bandwidth-bound
  - Long unit-stride memory accesses
  - Little reuse of each grid point
  - Few flops per grid point
- Actual AI values are typically lower (due to other types of cache misses)

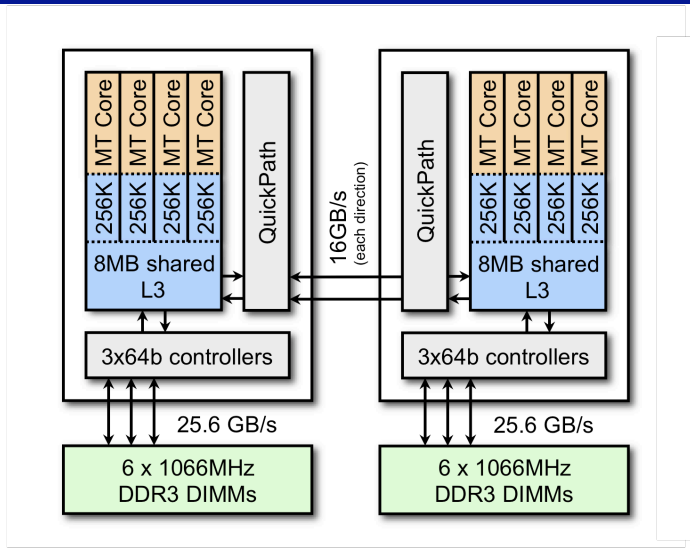
Outline

- Stencil Code Overview
- Cache-based Architectures
- Auto-tuning Description
- Stencil Auto-tuning Results

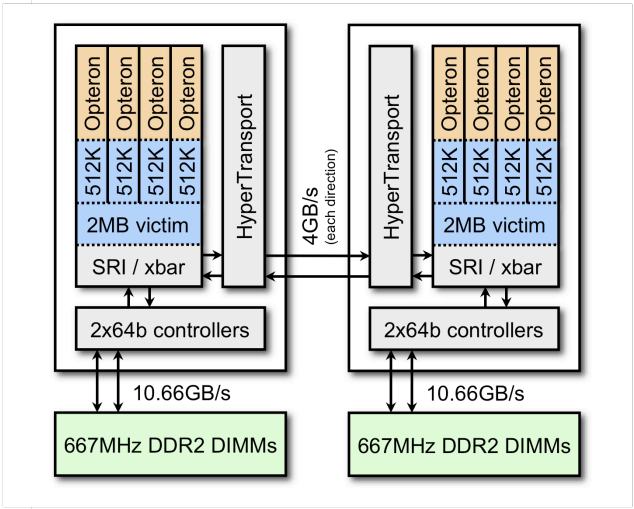
Cache-Based Architectures



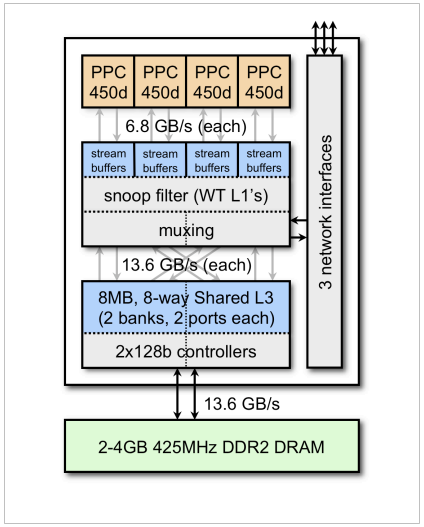
Intel Clovertown



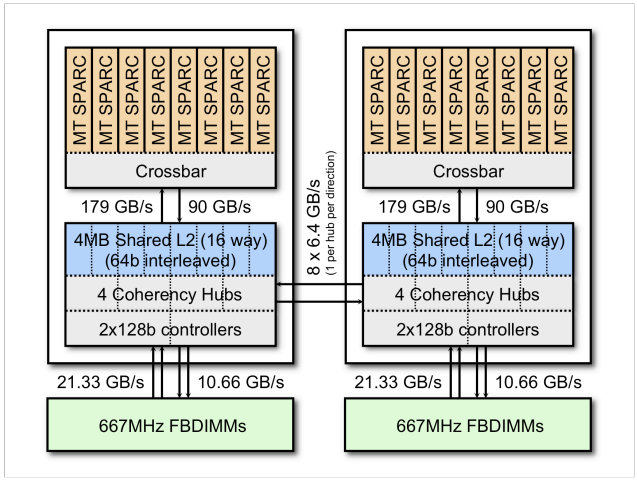
Intel Nehalem



AMD Barcelona

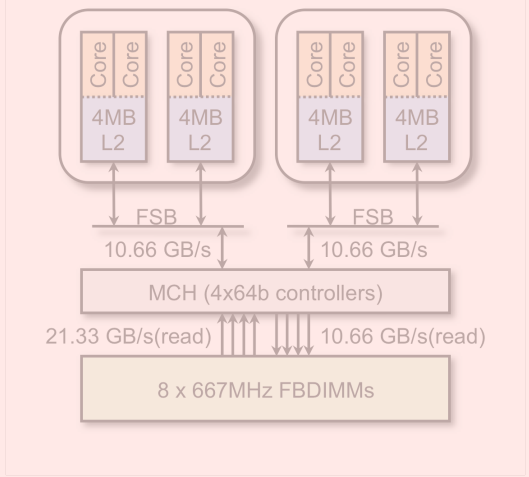


IBM Blue Gene/P

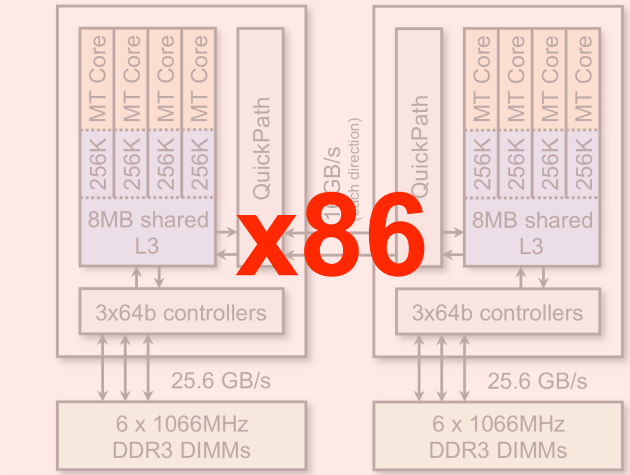


Sun Niagara2

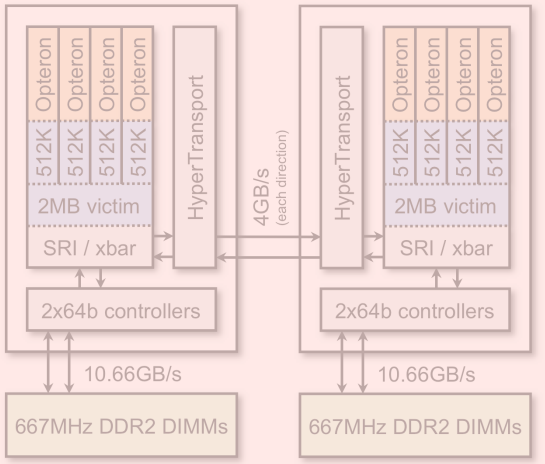
Cache-Based Architectures



Intel Clovertown

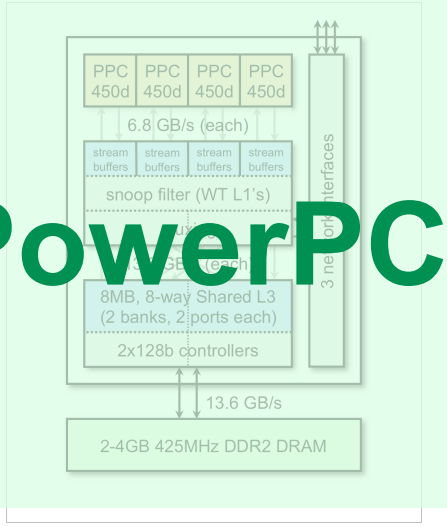


Intel Nehalem



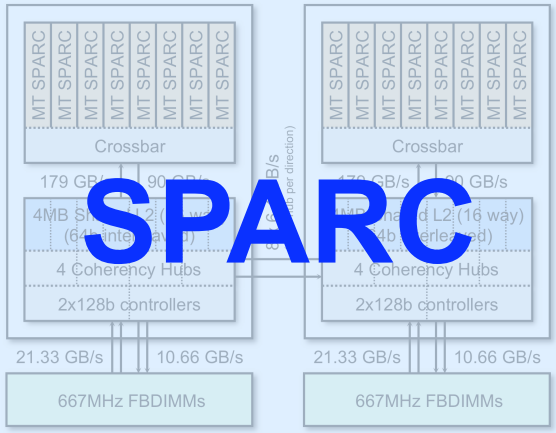
AMD Barcelona

PowerPC



IBM Blue Gene/P

SPARC



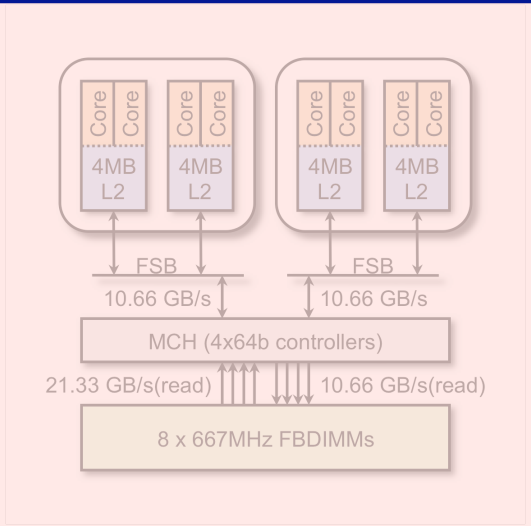
Sun Niagara2

ISA

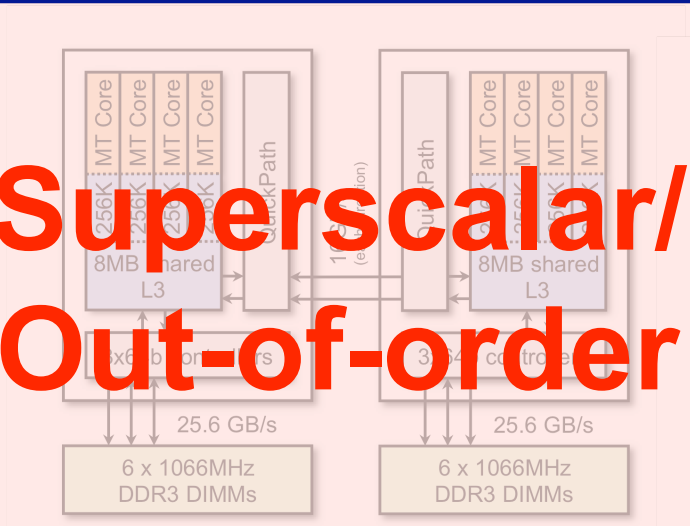
Cache-Based Architectures

EECS Electrical Engineering and Computer Sciences

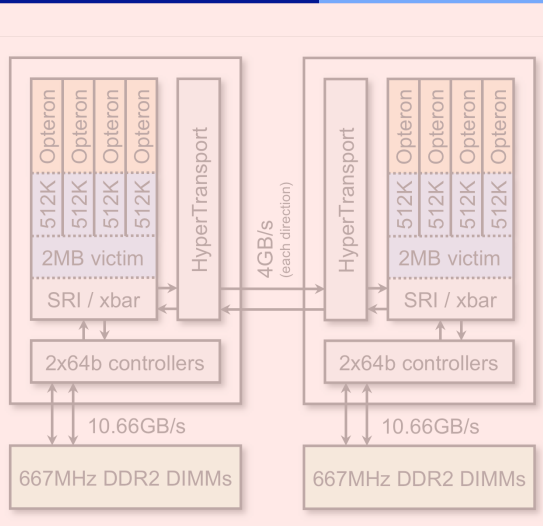
BERKELEY PAR LAB



Intel Clovertown

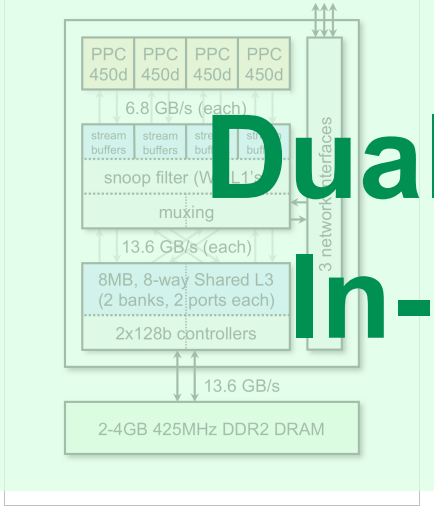


Intel Nehalem

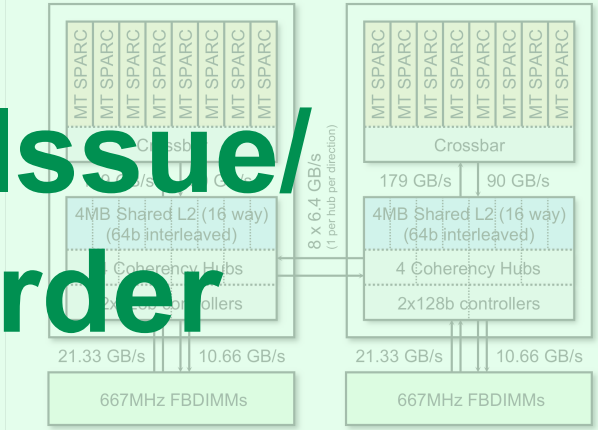


AMD Barcelona

Superscalar/ Out-of-order



IBM Blue Gene/P

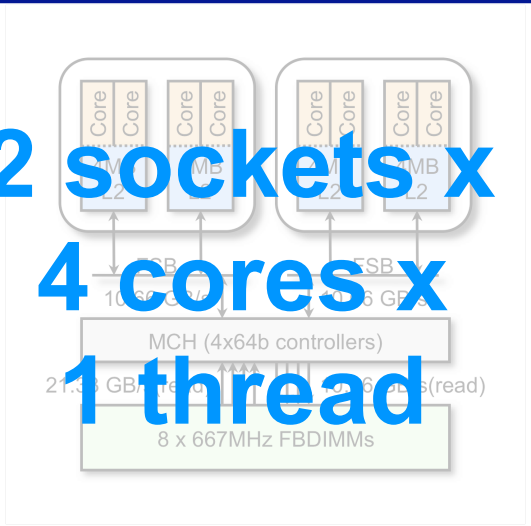


Sun Niagara2

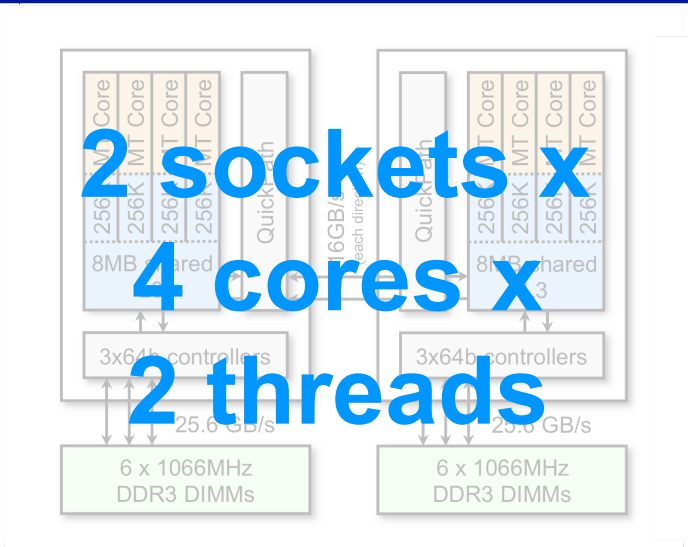
Dual Issue/ In-order

Core Type

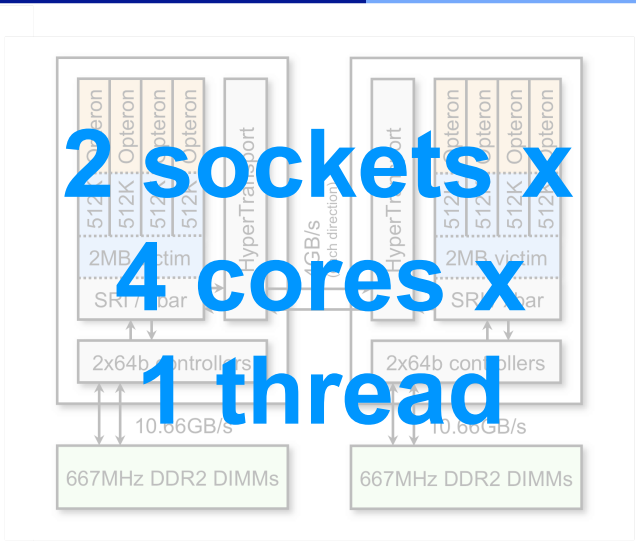
Cache-Based Architectures



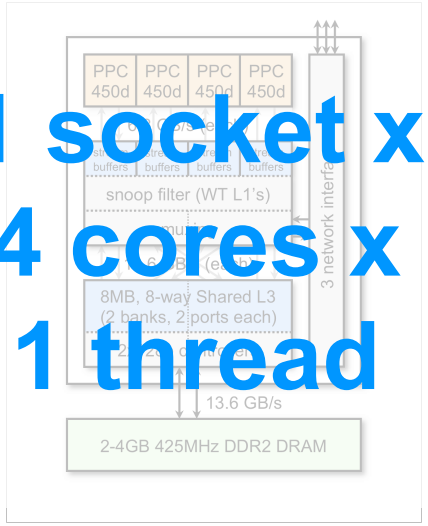
Intel Clovertown



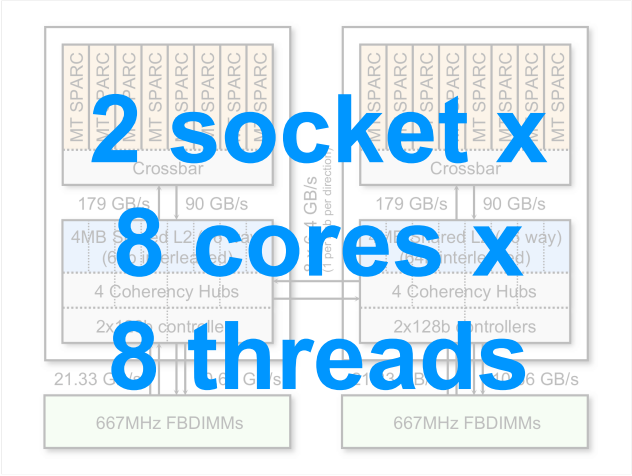
Intel Nehalem



AMD Barcelona



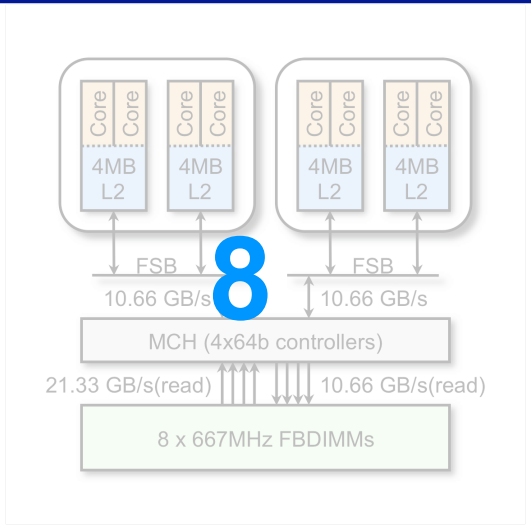
IBM Blue Gene/P



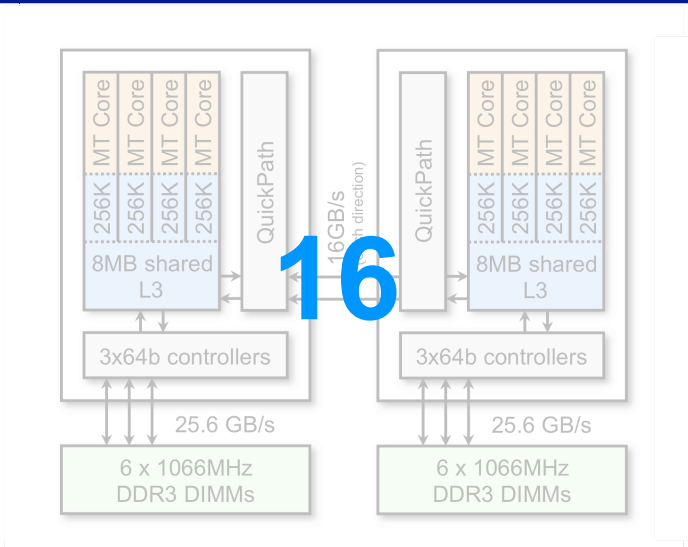
Sun Niagara2

Socket/Core/ Thread Count

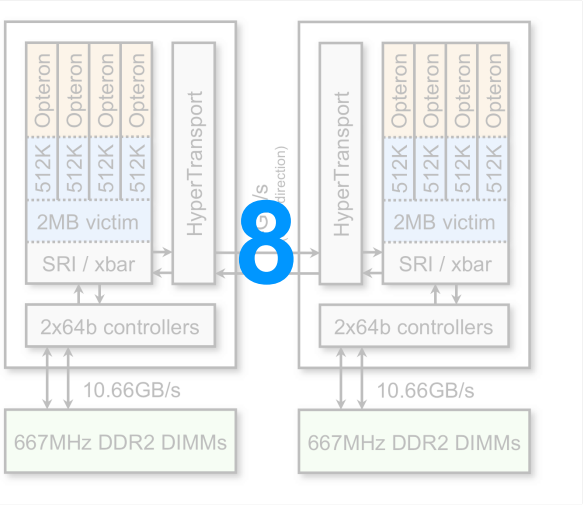
Cache-Based Architectures



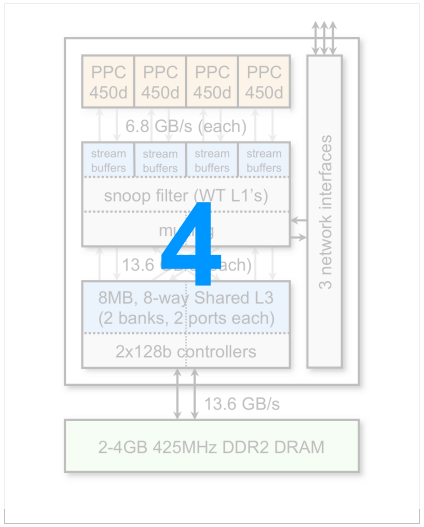
Intel Clovertown



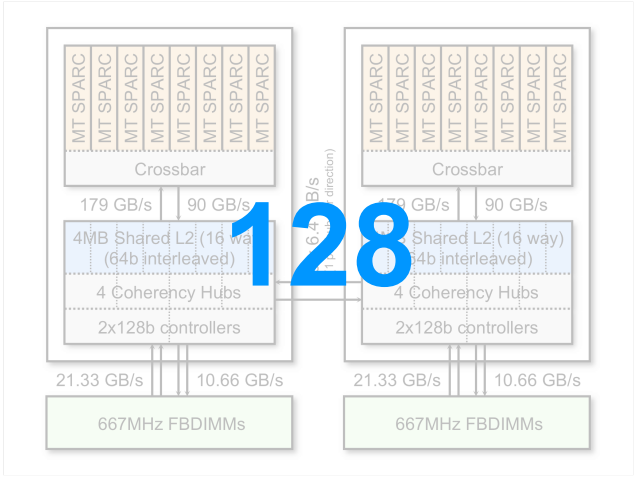
Intel Nehalem



AMD Barcelona



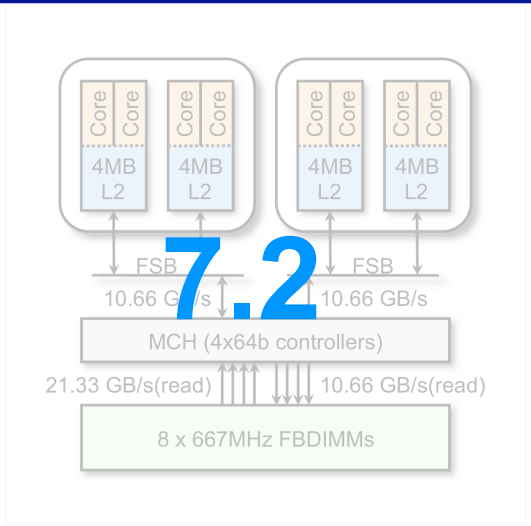
IBM Blue Gene/P



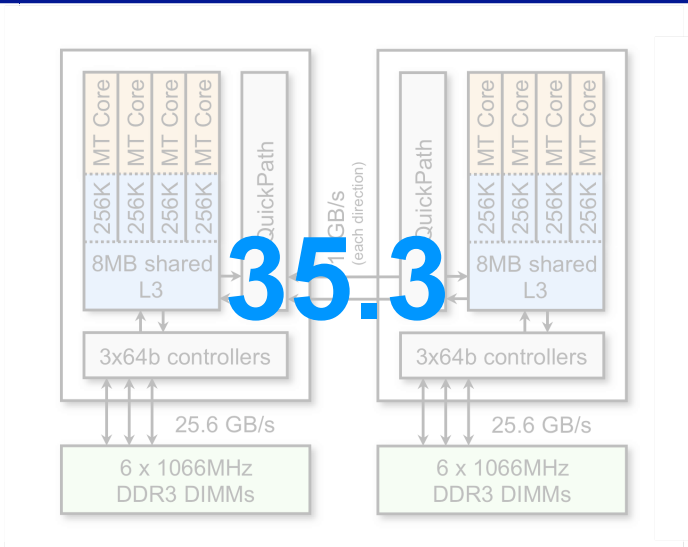
Sun Niagara2

Total HW Thread Count

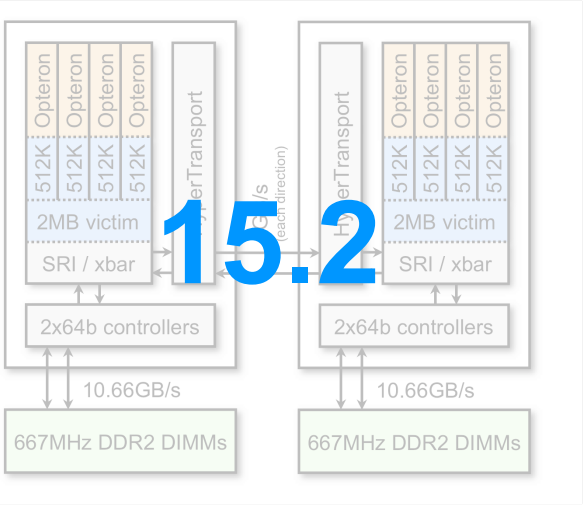
Cache-Based Architectures



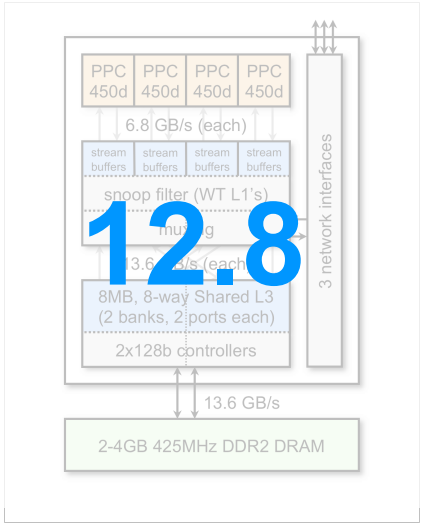
Intel Clovertown



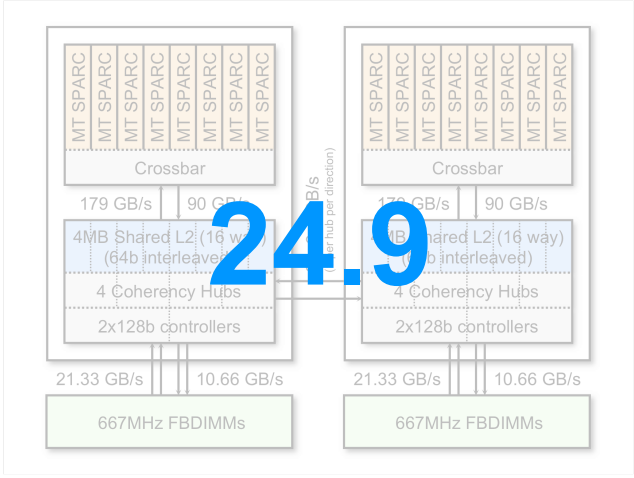
Intel Nehalem



AMD Barcelona



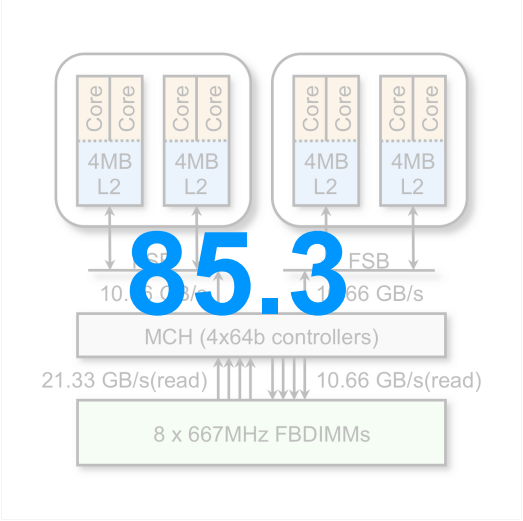
IBM Blue Gene/P



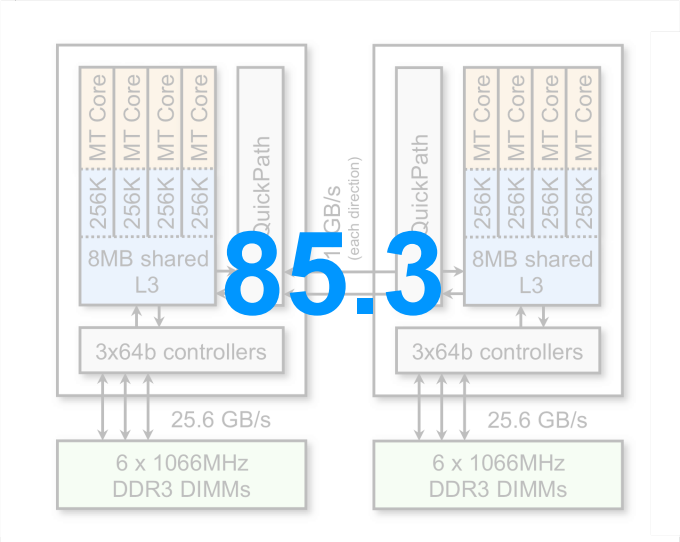
Sun Niagara2

Stream Copy Bandwidth (GB/s)

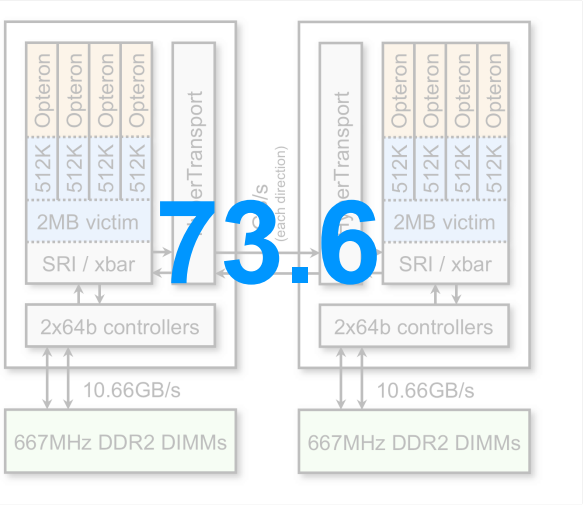
Cache-Based Architectures



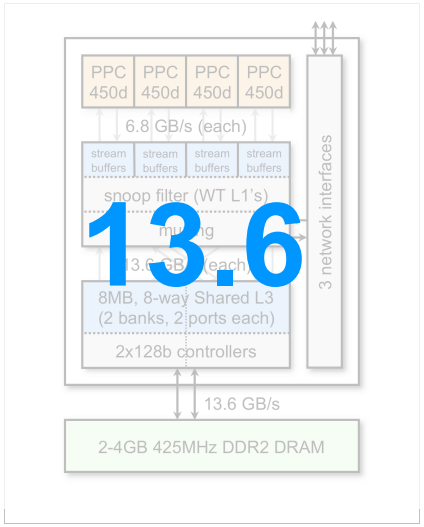
Intel Clovertown



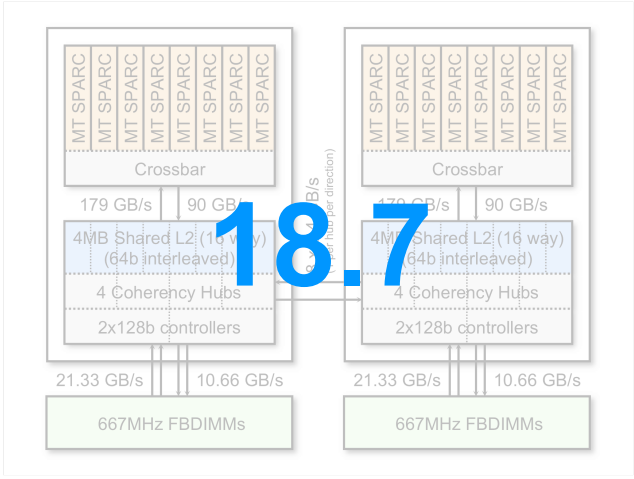
Intel Nehalem



AMD Barcelona



IBM Blue Gene/P



Sun Niagara2

Peak DP Computation Rate (GFlop/s)

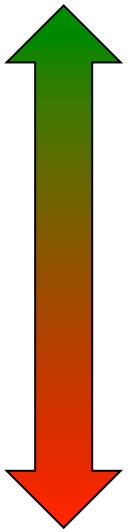
Outline

- Stencil Code Overview
- Cache-based Architectures
- Auto-tuning Description
- Stencil Auto-tuning Results

General Compiler Deficiencies

- Historically, compilers have had problems with domain-specific transformations:
  - Register allocation (explicit temps)
  - Loop unrolling
  - Software pipelining
  - Tiling
  - SIMDization
  - Common subexpression elimination
  - Data structure transformations
  - Algorithmic transformations

} Domain-specific



- Compilers typically use heuristics (not actual runs) to determine the best code for a platform
  - Difficult to generate optimal code across many diverse multicore architectures

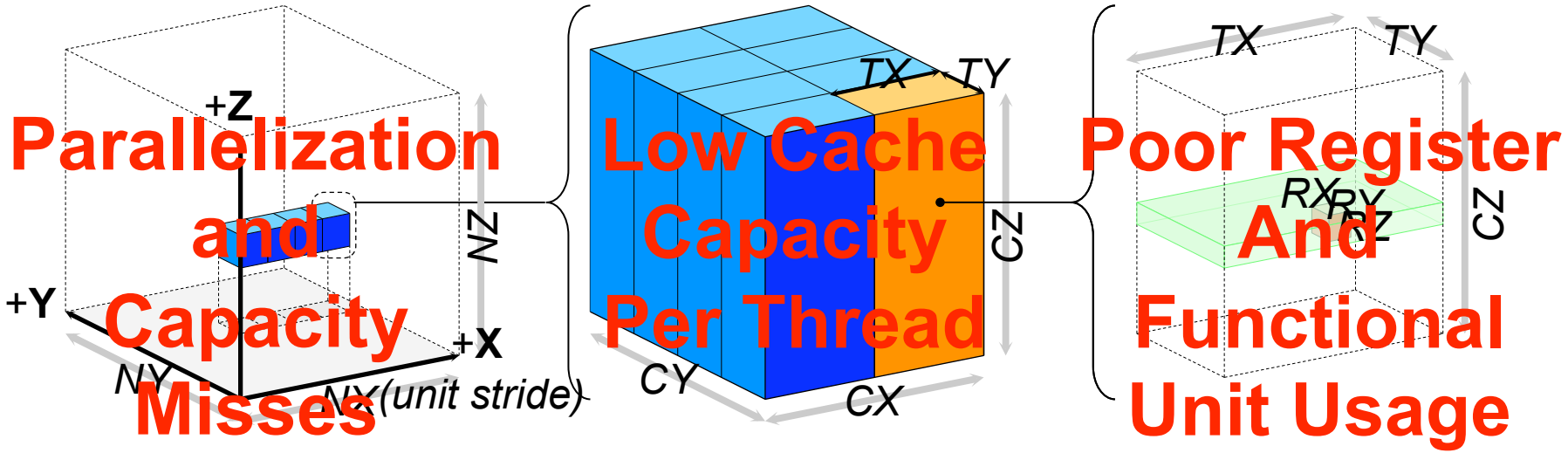
Rise of Automatic Tuning

- Auto-tuning became popular because:
  - Domain-specific transformations could be included
  - Runs experiments instead of heuristics
  - Diversity of systems (and now increasing core counts) made *performance portability* vital
- Auto-tuning is:
  - Portable (to an extent)
  - Scalable
  - Productive (if tuning for multiple architectures)
  - Applicable to many metrics of merit (e.g. performance, power efficiency)
- We let the machine search the parameter space intelligently to find a (near-)optimal configuration
- Serial processor success stories: FFTW, Spiral, Atlas, OSKI, others…

Outline

- Stencil Code Overview
- Cache-based Architectures
- Auto-tuning Description
  - Identify motif-specific optimizations
  - Generate code variants based on these optimizations
  - Traverse parameter space for best configuration
- Stencil Auto-tuning Results

Problem Decomposition
(across an SMP)



Core Blocking

- Allows for domain decomposition and cache blocking

Thread Blocking

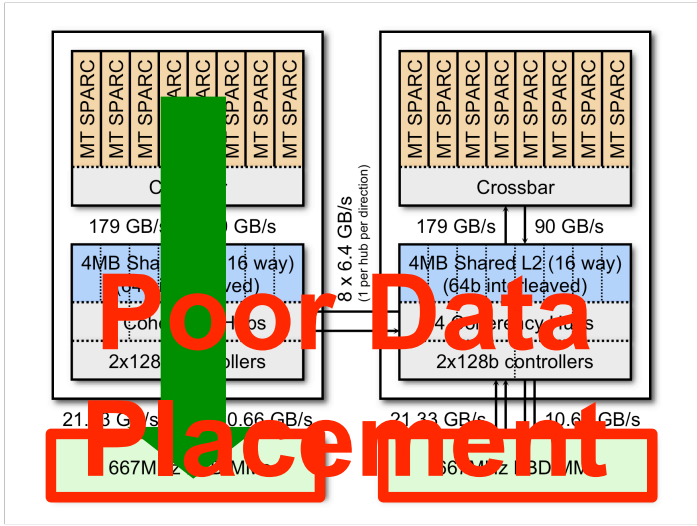
- Exploit caches shared among threads within a core

Register Blocking

- Loop unrolling in any of the three dimensions
- Makes DLP/ILP explicit

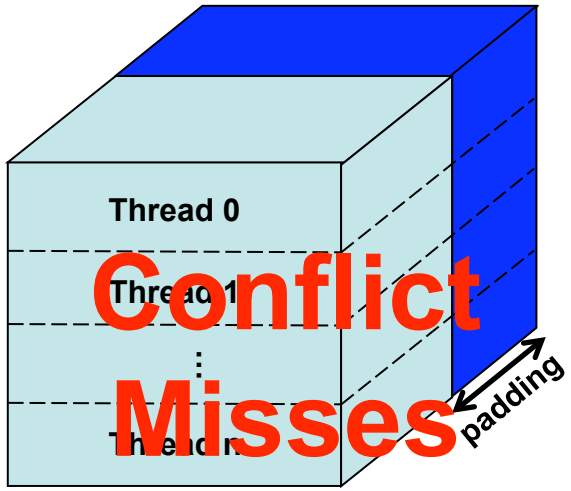
- This decomposition is universal across *all* examined architectures
- Decomposition does **not** change data structure
- Need to choose best block sizes for each hierarchy level

Data Allocation



NUMA-Aware Allocation

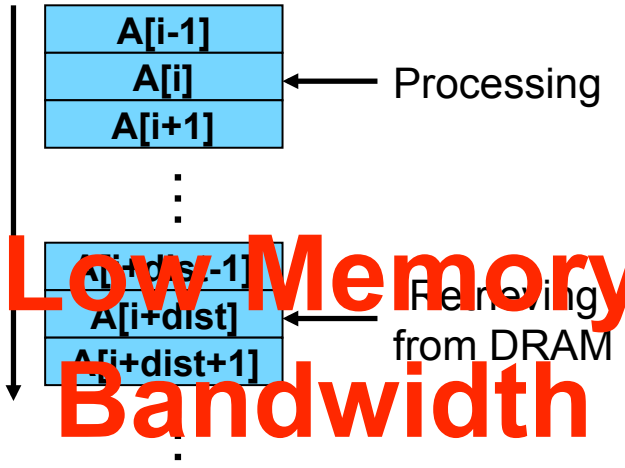
- Ensures that the data is co-located on same socket as the threads processing it



Array Padding

- Alters the data placement so as to minimize conflict misses
- Tunable parameter

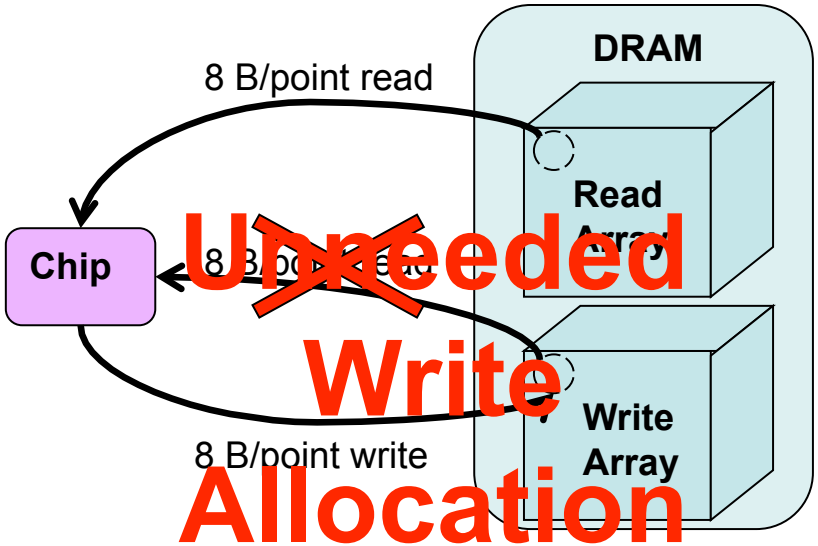
Bandwidth Optimizations



Low Memory Bandwidth

Software Prefetching

- Helps mask memory latency by adjusting look-ahead distance
- Can also tune number of software prefetch requests

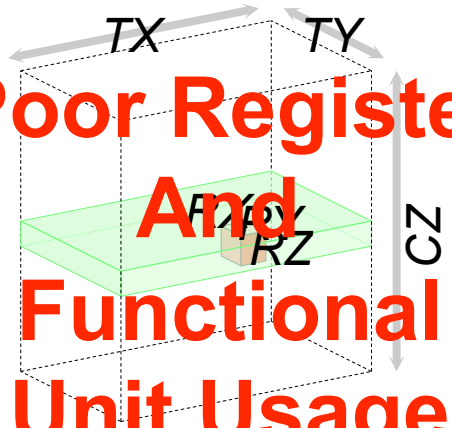


Unneeded Write Allocation

Cache Bypass

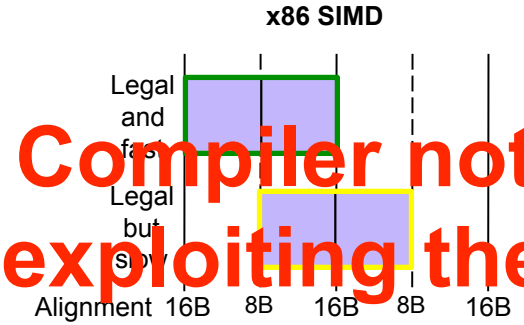
- Eliminates cache line fills on a write miss
- Reduces memory traffic by 50% on write misses!
- Only available on x86 machines

In-Core Optimizations



Poor Register And Functional Unit Usage

Register Blocking



Compiler not exploiting the ISA

Explicit SIMDization

- Single instruction processes multiple data items
- Non-portable code

```
c = a+b;
d = a+b;
e = c+d;
```

```
c = a+b;
e = c+c;
```

Unneeded flops are being performed

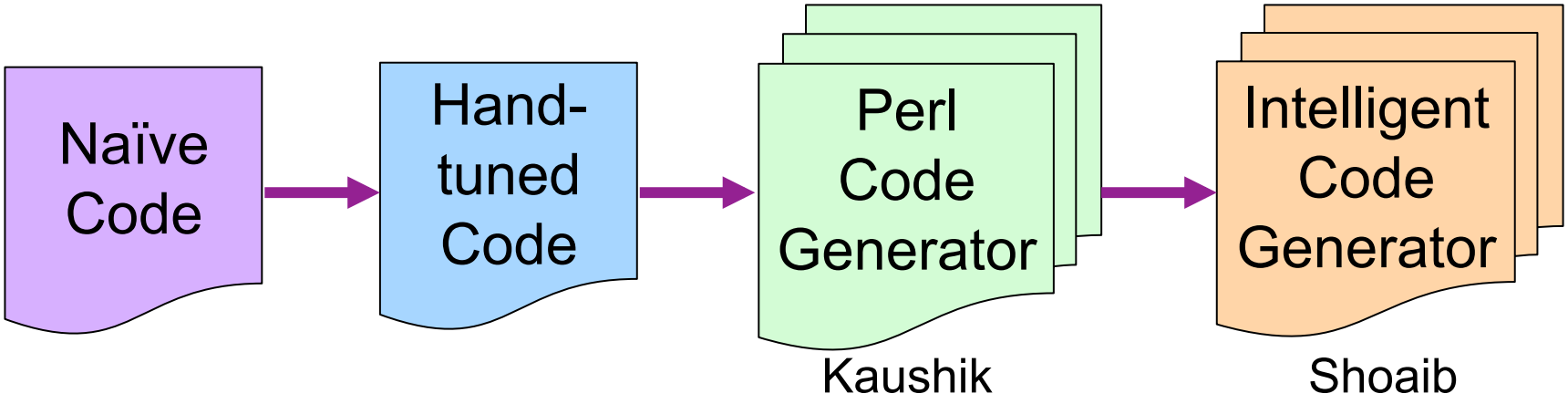
Common Subexpression Elimination

- Reduces flops by removing redundant expressions
- `icc` and `gcc` often fail to do this

Outline

- Stencil Code Overview
- Cache-based Architectures
- Auto-tuning Description
  - Identify motif-specific optimizations
  - Generate code variants based on these optimizations
  - Traverse parameter space for best configuration
- Stencil Auto-tuning Results

Stencil Code Evolution



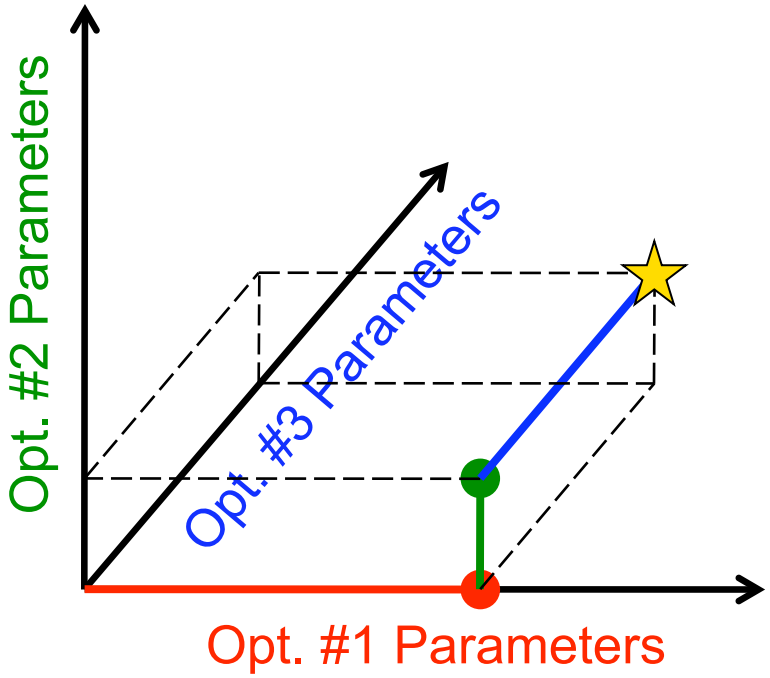
- Hand-tuned code only performs well on a single platform
- Perl code generator can produce many different code variants for performance portability
- Intelligent code generator can take pseudo-code and specified set of transformations to produce code variants
  - Type of domain-specific compiler

Outline

- Stencil Code Overview
- Cache-based Architectures
- Auto-tuning Description
  - Identify motif-specific optimizations
  - Generate code variants based on these optimizations
  - Traverse parameter space for best configuration
- Stencil Auto-tuning Results

Traversing the Parameter Space

- We introduced 9 different optimizations, each of which has its own set of parameters
- Exhaustive search is *impossible*
- To make problem tractable, we:
  - Used expert knowledge to order the optimizations
  - Applied them consecutively
- Every platform had its own set of best parameters

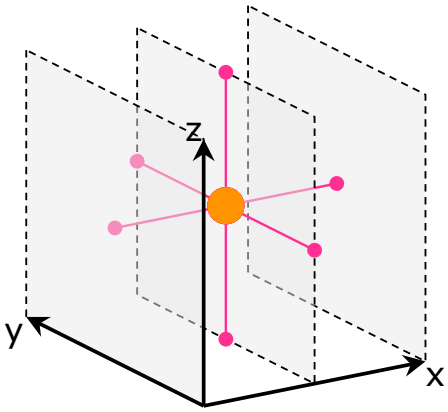


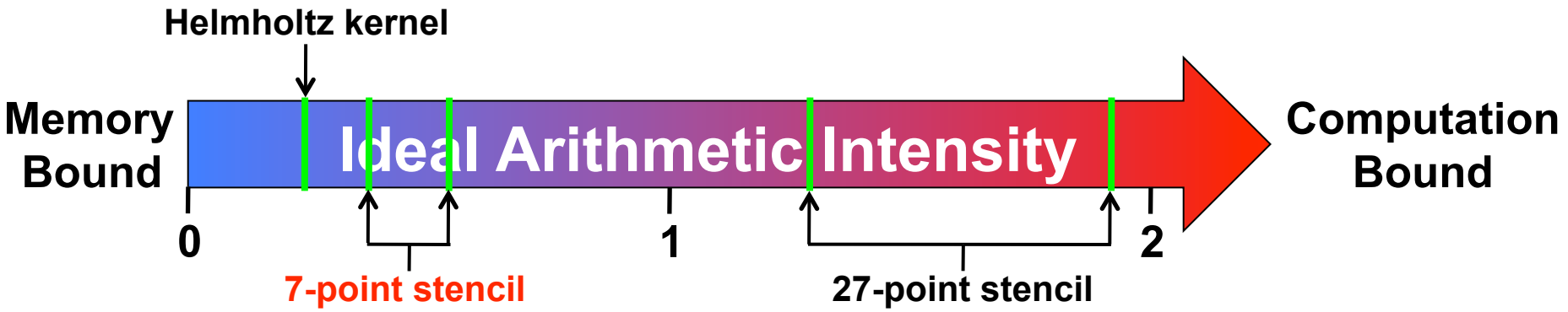
Outline

- Stencil Code Overview
- Cache-based Architectures
- Auto-tuning Description
- Stencil Auto-tuning Results
  - 3D 7-Point Stencil (Memory-Intensive Kernel)
  - 3D 27-Point Stencil (Compute-Intensive Kernel)
  - 3D Helmholtz Kernel

3D 7-Point Stencil Problem

- The 3D 7-point stencil performs:
  - 8 flops per point
  - 16 or 24 Bytes of memory traffic per point
- AI is either 0.33 or 0.5 (w/ cache bypass)
- This kernel should be memory-bound on most architectures:

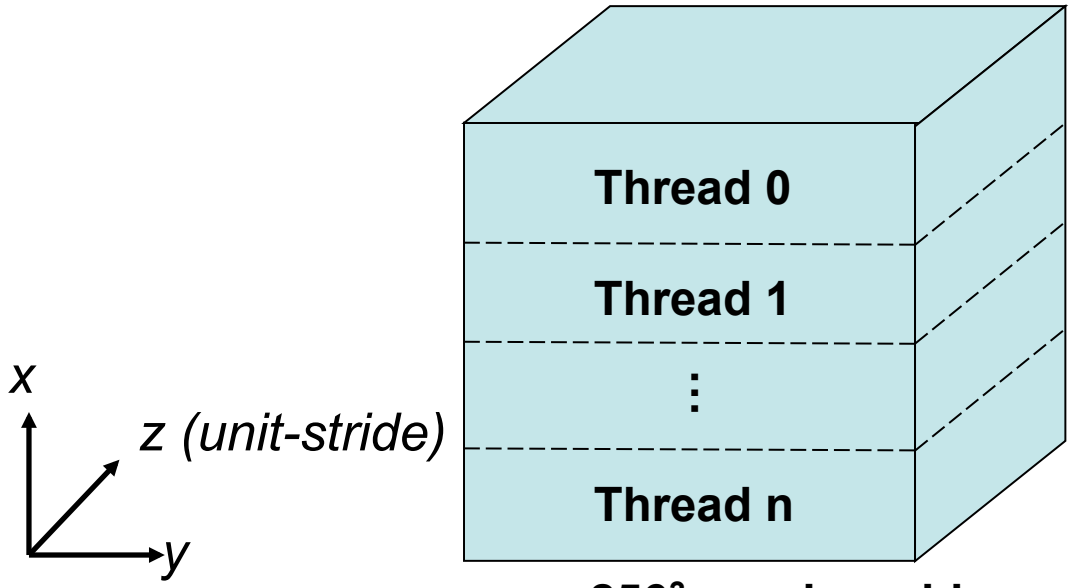




- We will perform a single out-of-place sweep of this stencil over a 256^3 grid

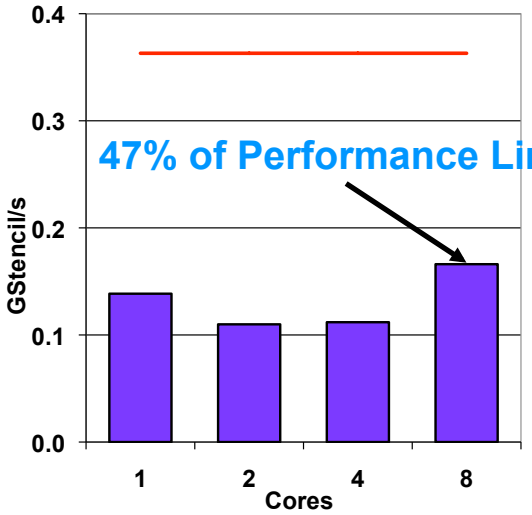
Naïve Stencil Code

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

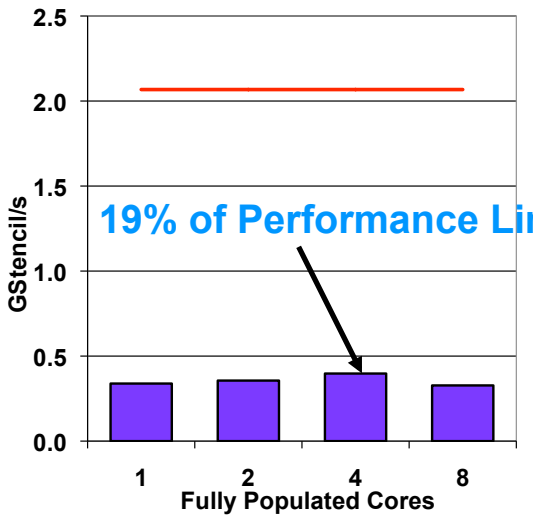


256^3 regular grid

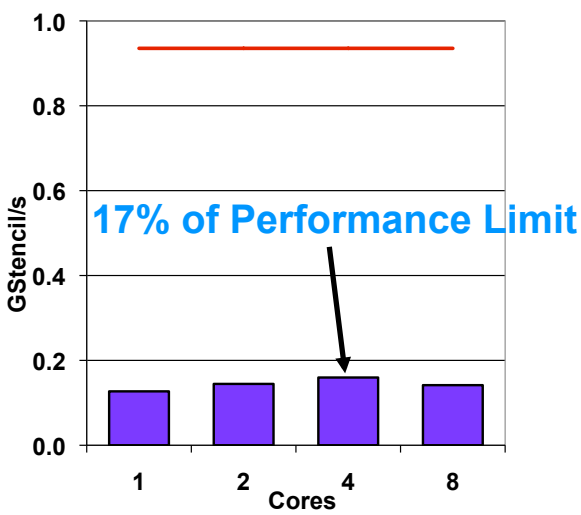
Naïve Performance
(3D 7-Point Stencil)



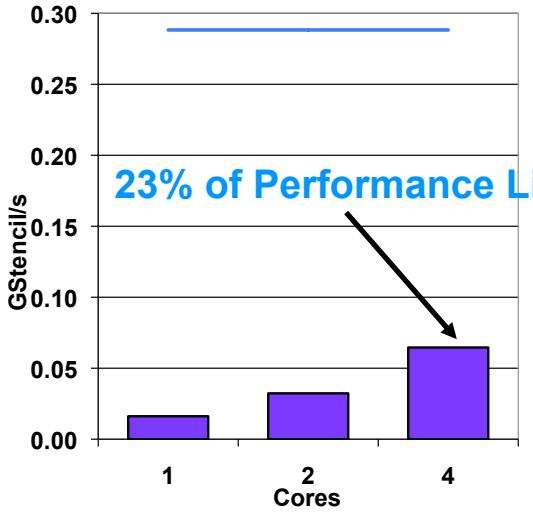
Intel Clovertown



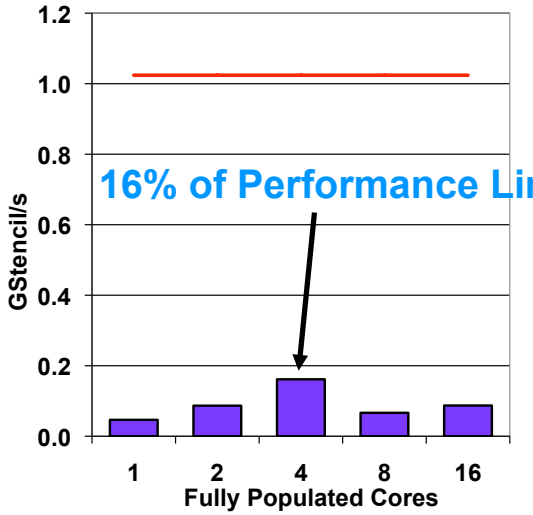
Intel Nehalem



AMD Barcelona

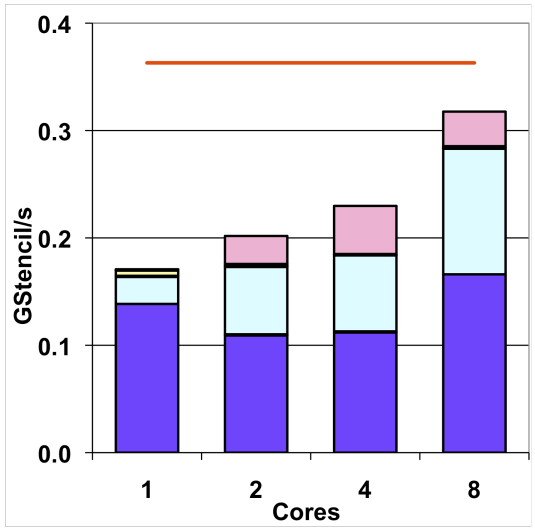


IBM Blue Gene/P

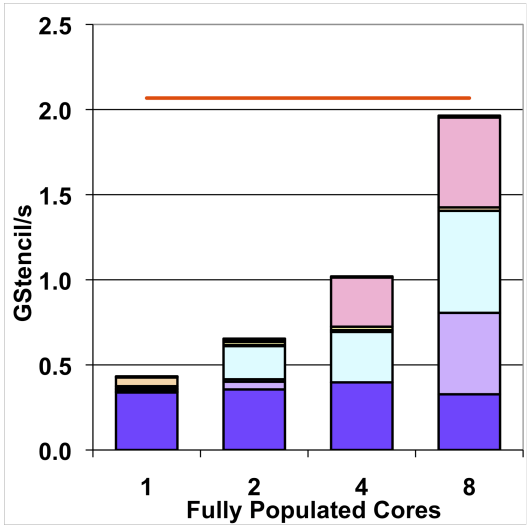


Sun Niagara2

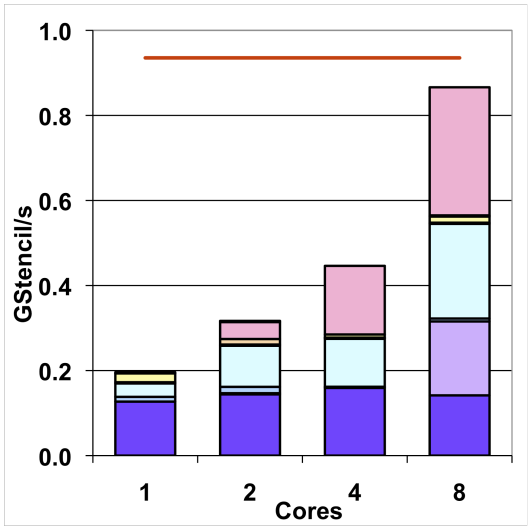
Auto-tuned Performance
(3D 7-Point Stencil)



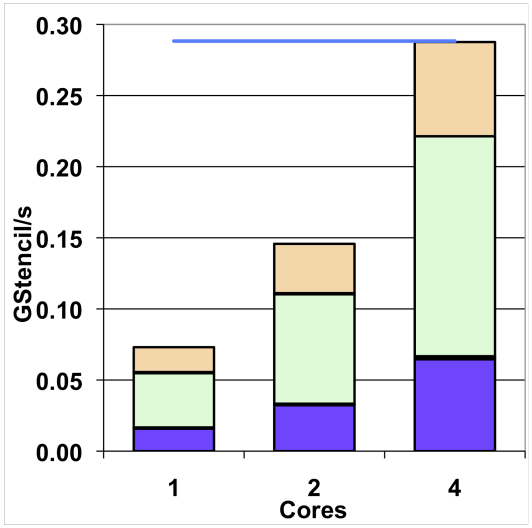
Intel Clovertown



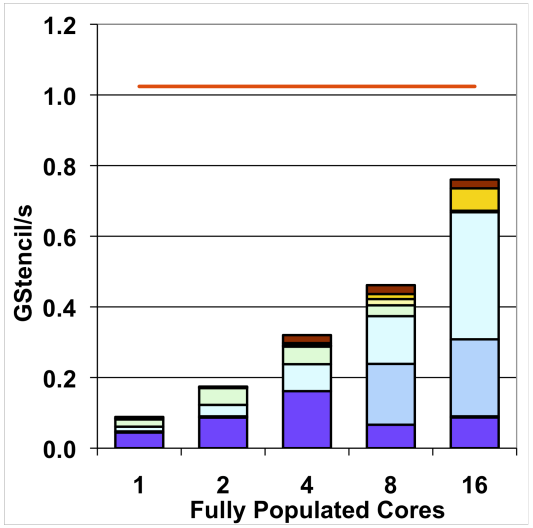
Intel Nehalem



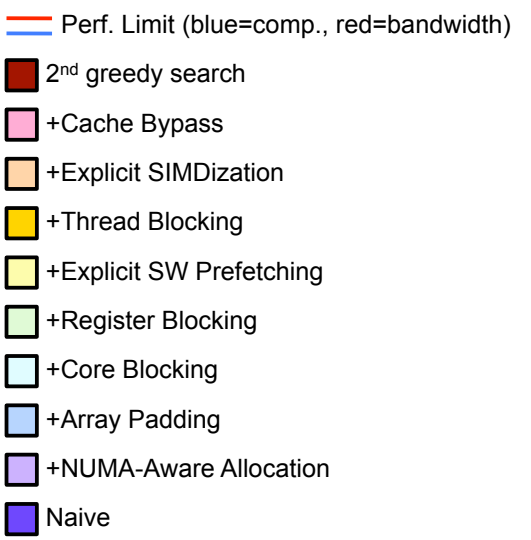
AMD Barcelona



IBM Blue Gene/P



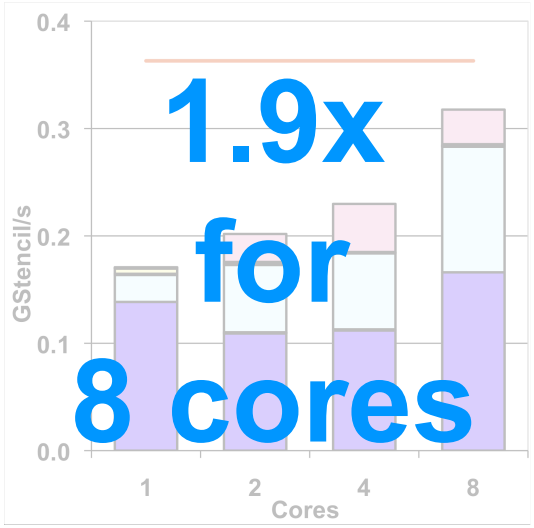
Sun Niagara2



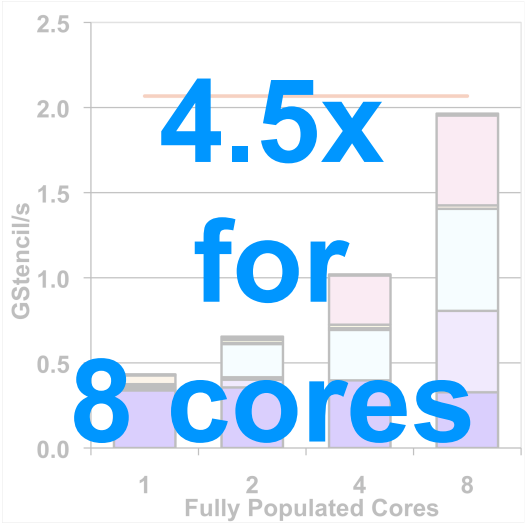
EECS
Electrical Engineering and Computer Sciences

Scalability?
(3D 7-Point Stencil)

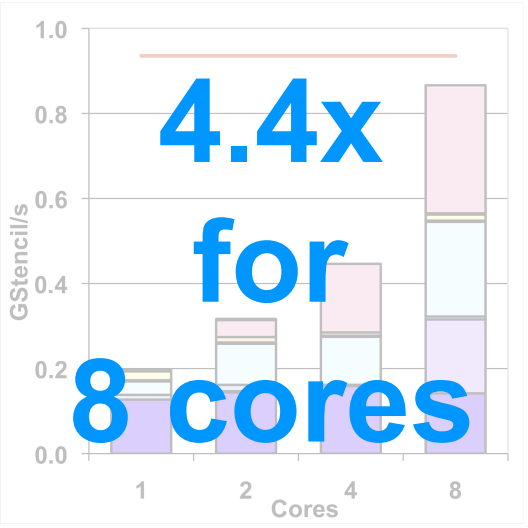
BERKELEY PAR LAB



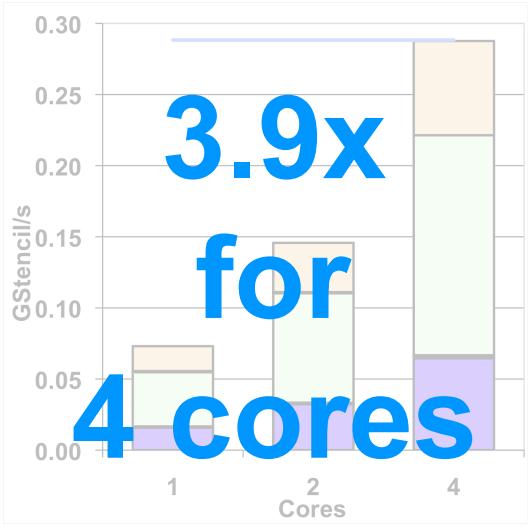
Intel Clovertown



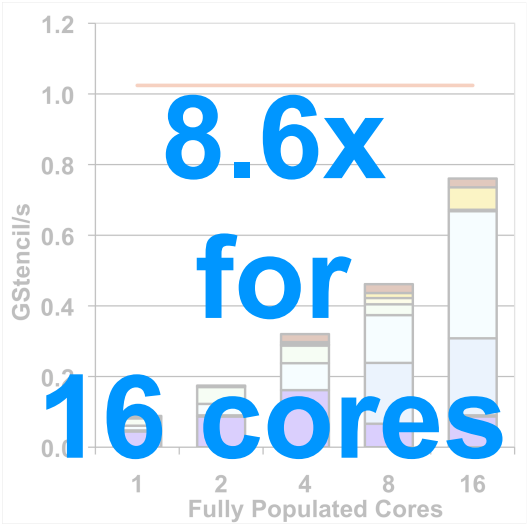
Intel Nehalem



AMD Barcelona



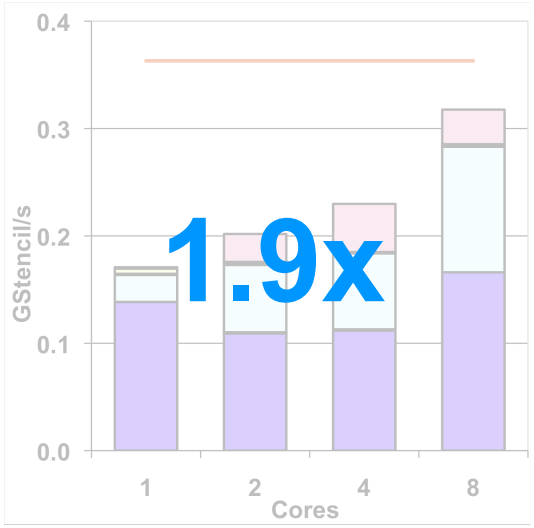
IBM Blue Gene/P



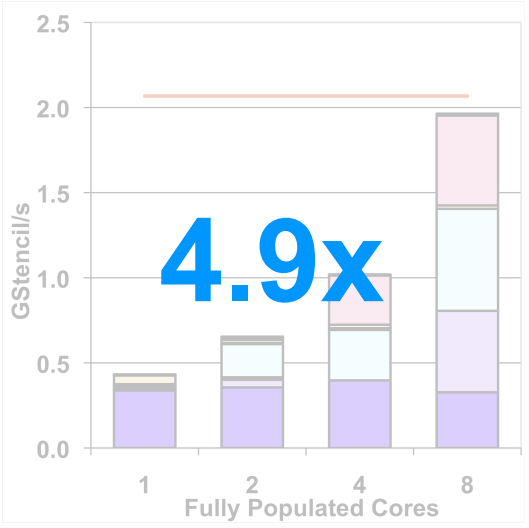
Sun Niagara2

Parallel Scaling Speedup Over Single Core Performance

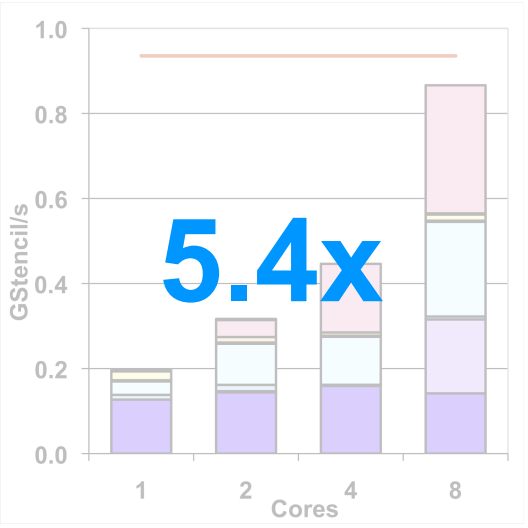
How much improvement is there?
(3D 7-Point Stencil)



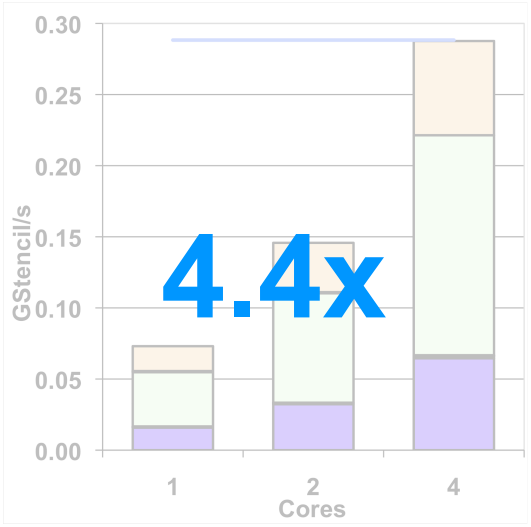
Intel Clovertown



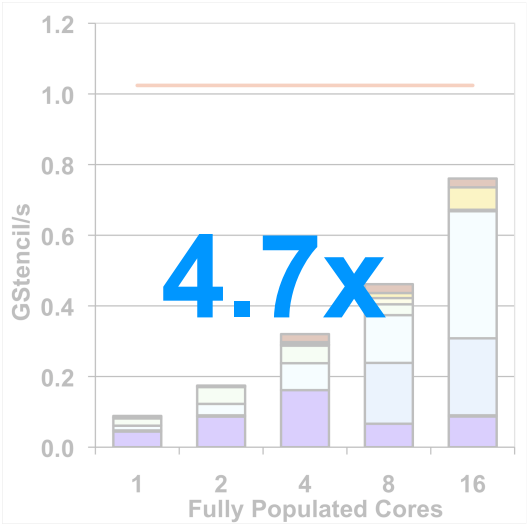
Intel Nehalem



AMD Barcelona



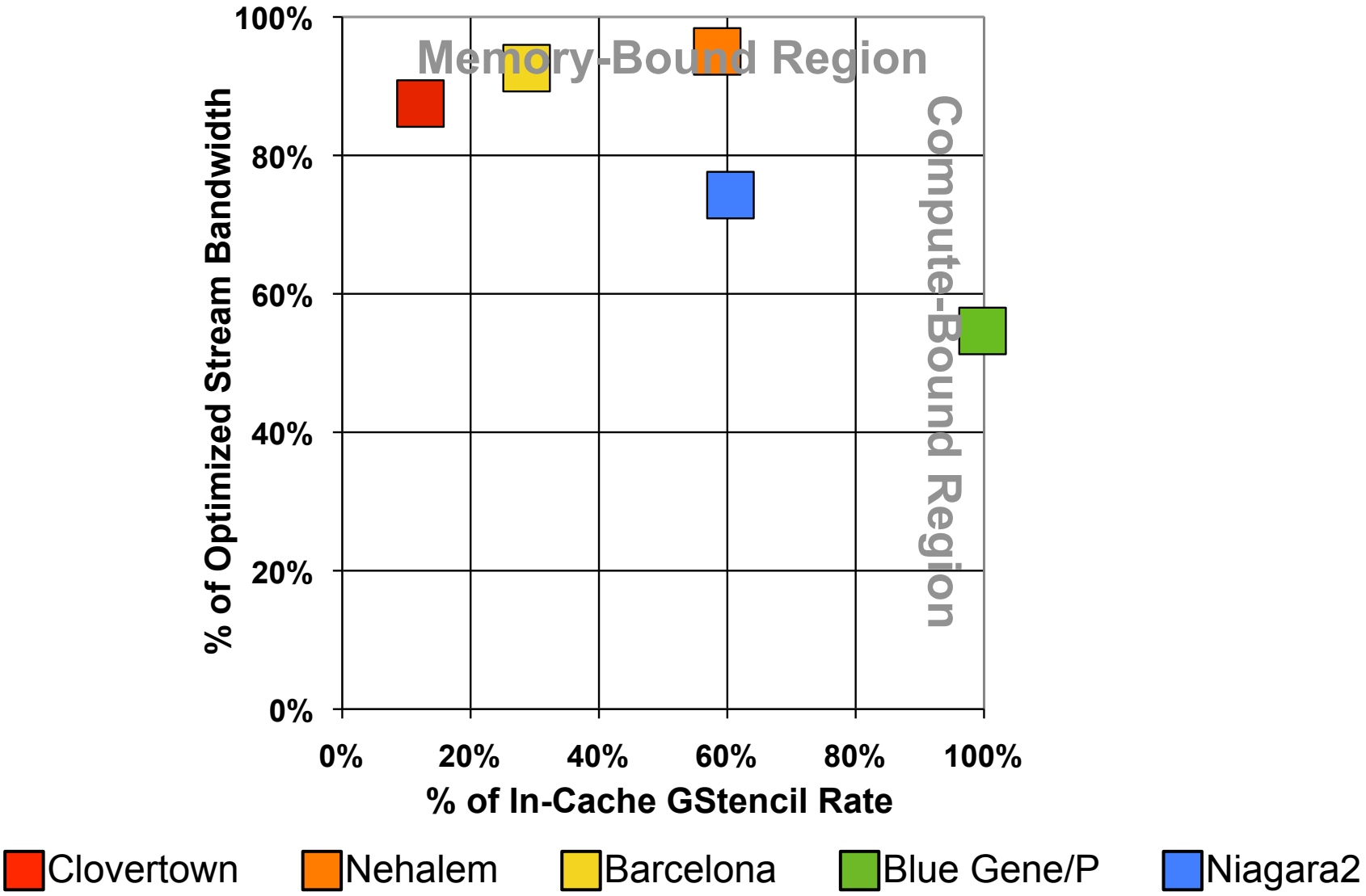
IBM Blue Gene/P



Sun Niagara2

Tuning Speedup Over Best Naïve Performance

How well can we do?
(3D 7-Point Stencil)

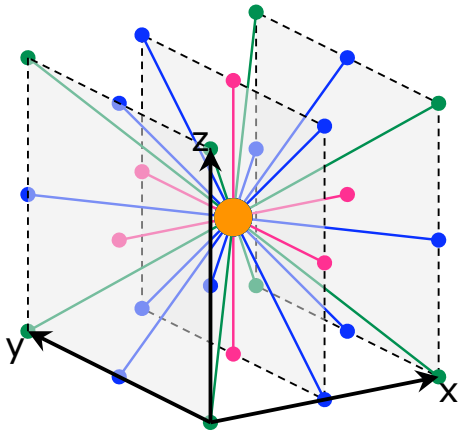


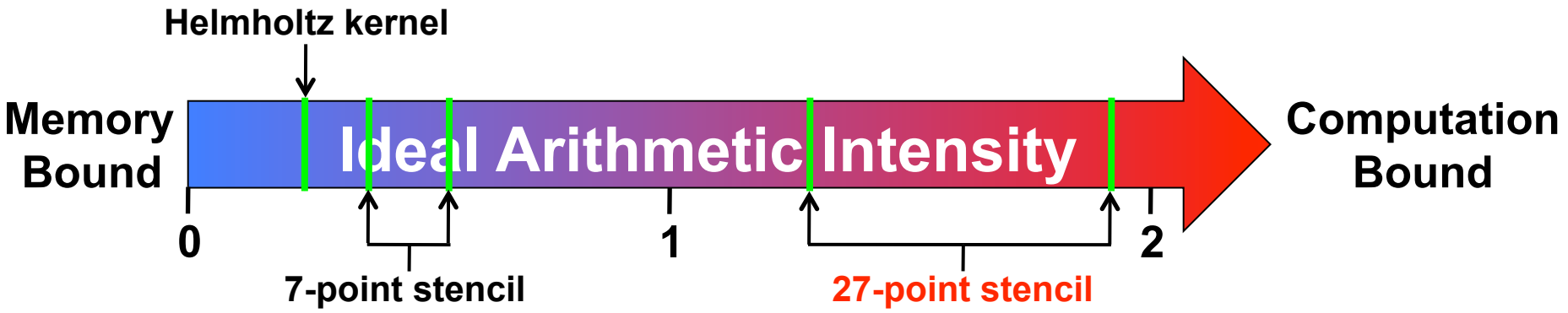
Outline

- Stencil Code Overview
- Cache-based Architectures
- Auto-tuning Description
- Stencil Auto-tuning Results
  - 3D 7-Point Stencil (Memory-Intensive Kernel)
  - 3D 27-Point Stencil (Compute-Intensive Kernel)
  - 3D Helmholtz Kernel

3D 27-Point Stencil Problem

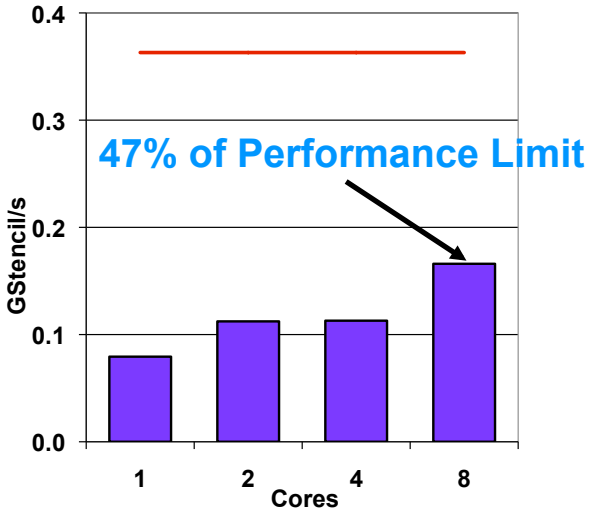
- The 3D 27-point stencil performs:
  - 30 flops per point
  - 16 or 24 Bytes of memory traffic per point
- AI is either 1.25 or 1.88 (w/ cache bypass)
- CSE can reduce the flops/point
- This kernel should be compute-bound on most architectures:



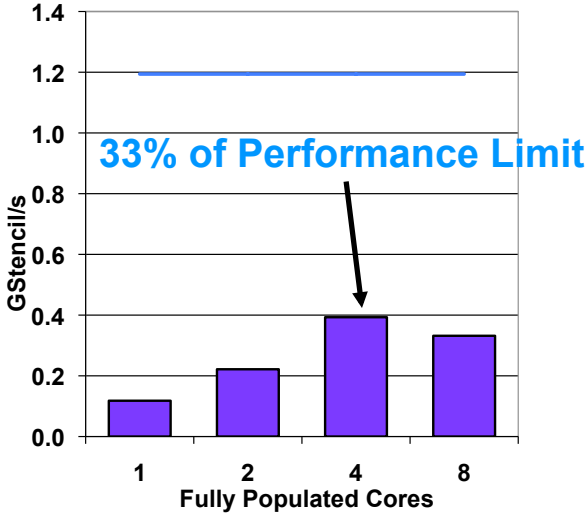


- We will perform a single out-of-place sweep of this stencil over a 256^3 grid

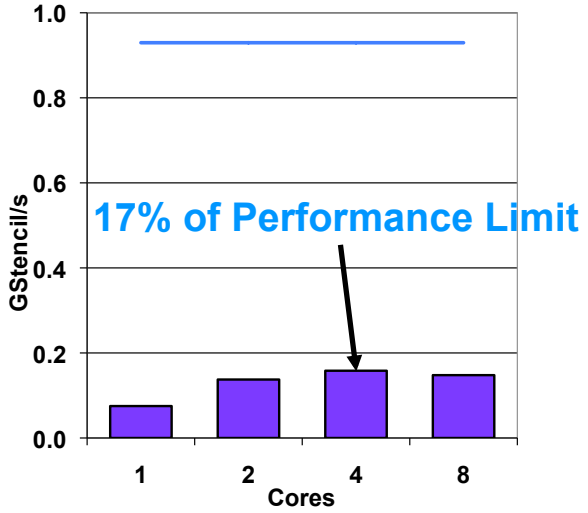
Naïve Performance
(3D 27-Point Stencil)



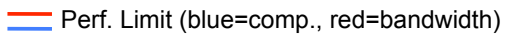
Intel Clovertown

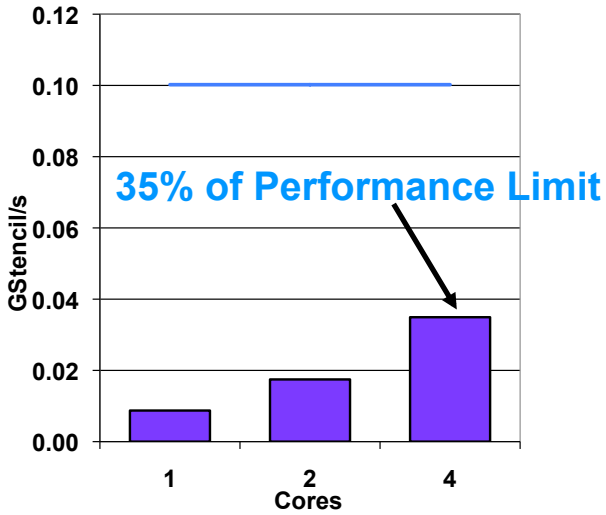


Intel Nehalem

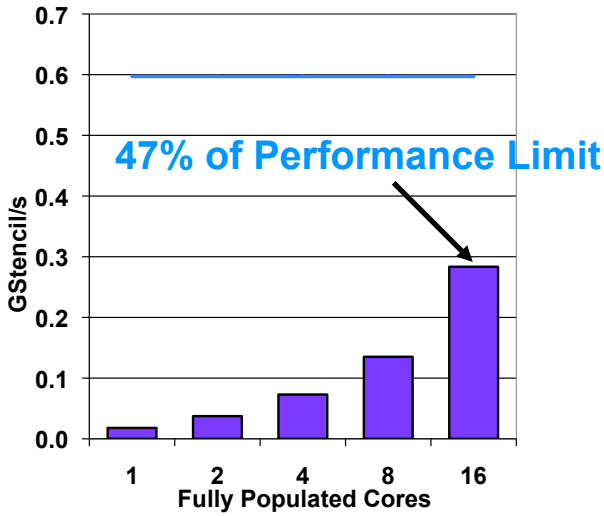


AMD Barcelona



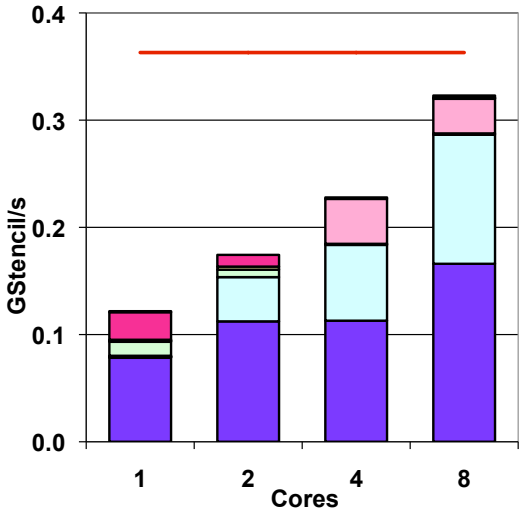


IBM Blue Gene/P

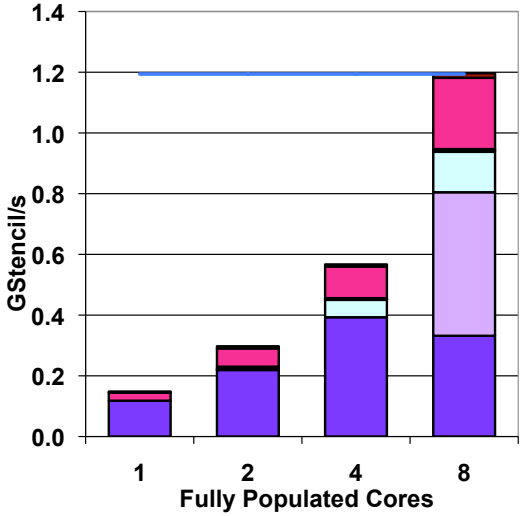


Sun Niagara2

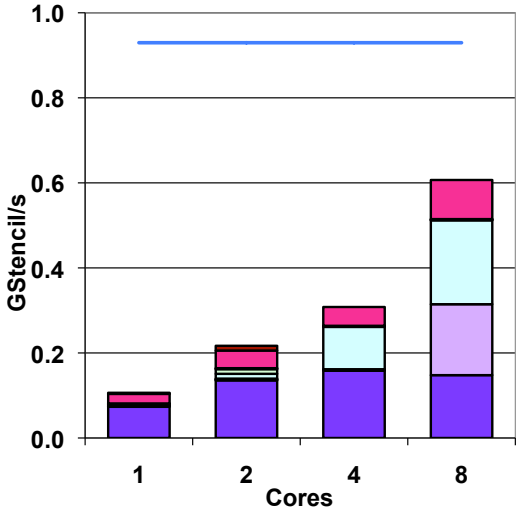
Auto-tuned Performance
(3D 27-Point Stencil)



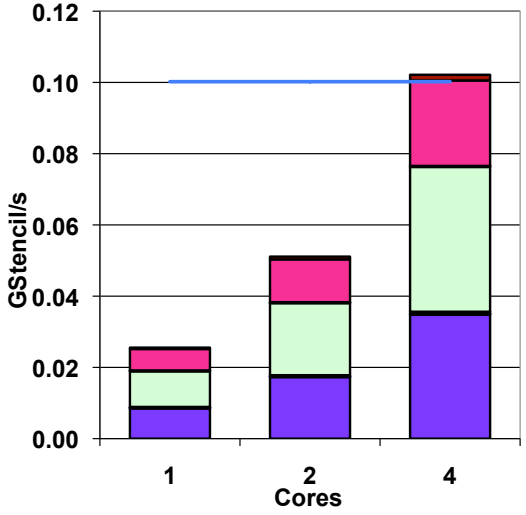
Intel Clovertown



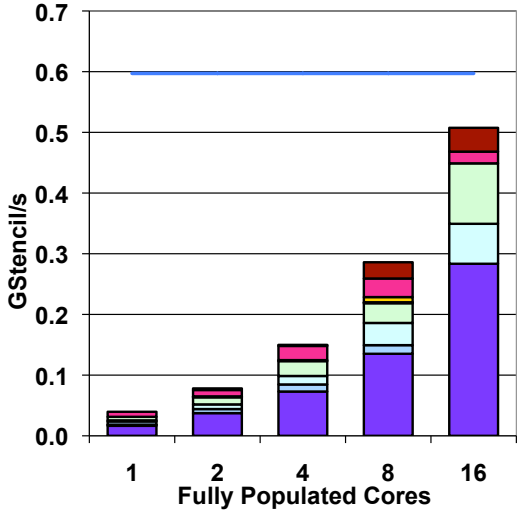
Intel Nehalem



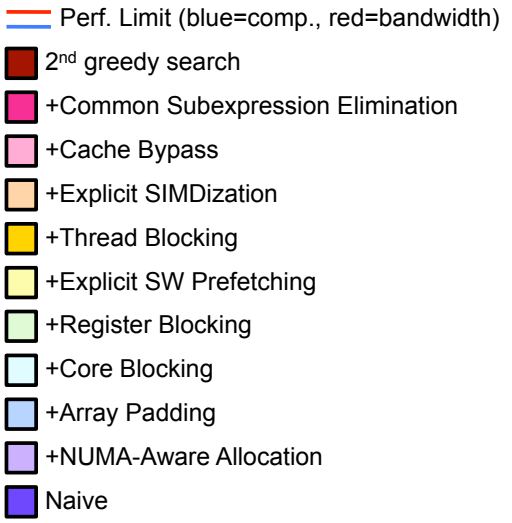
AMD Barcelona



IBM Blue Gene/P



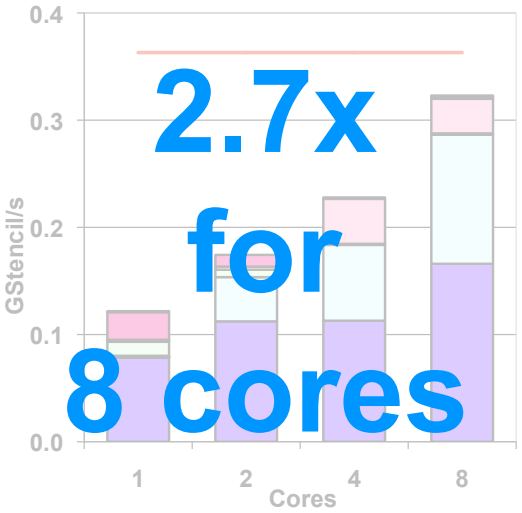
Sun Niagara2



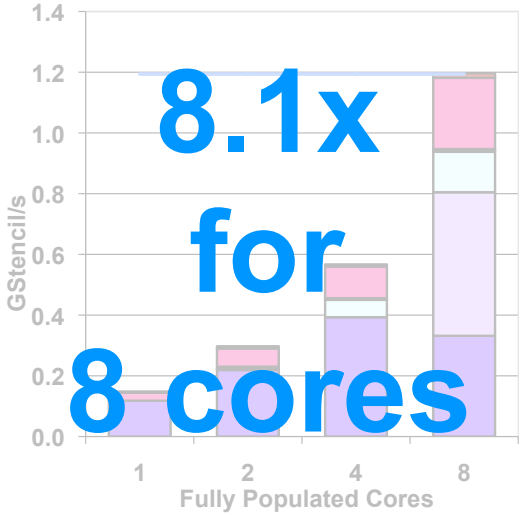
EECS
Electrical Engineering and Computer Sciences

Scalability?
(3D 27-Point Stencil)

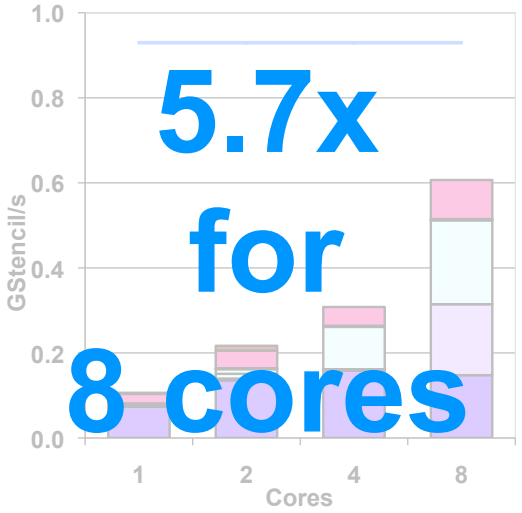
BERKELEY PAR LAB



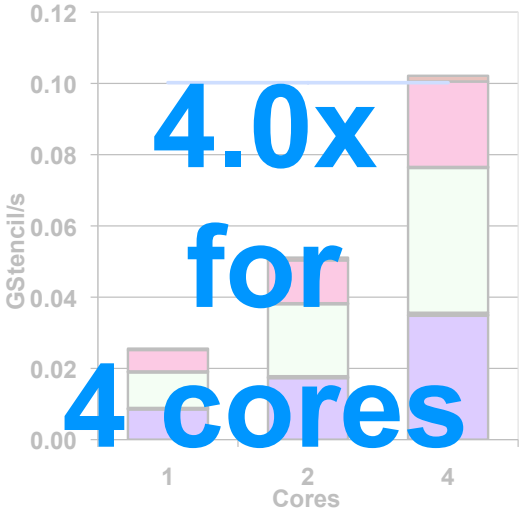
Intel Clovertown



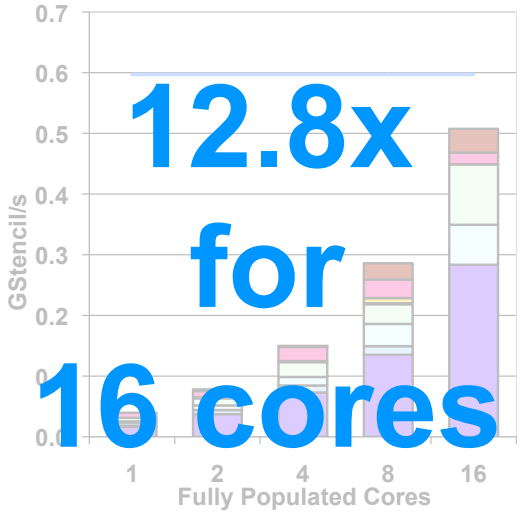
Intel Nehalem



AMD Barcelona



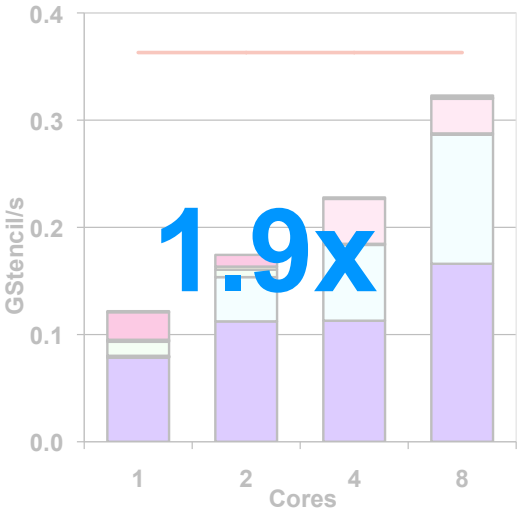
IBM Blue Gene/P



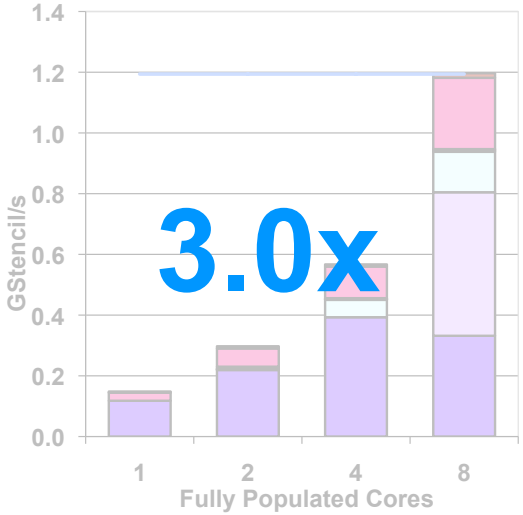
Sun Niagara2

Parallel Scaling Speedup Over Single Core Performance

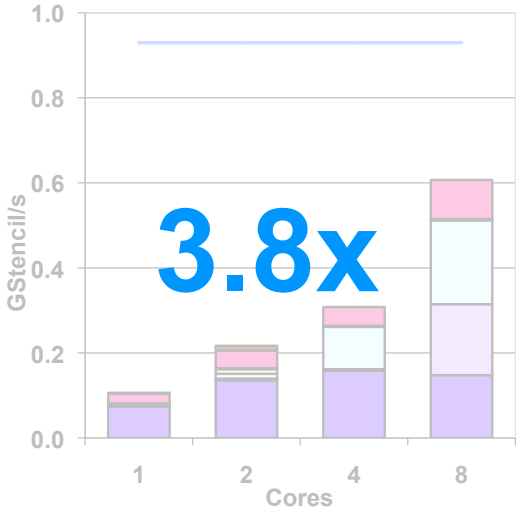
How much improvement is there?
(3D 27-Point Stencil)



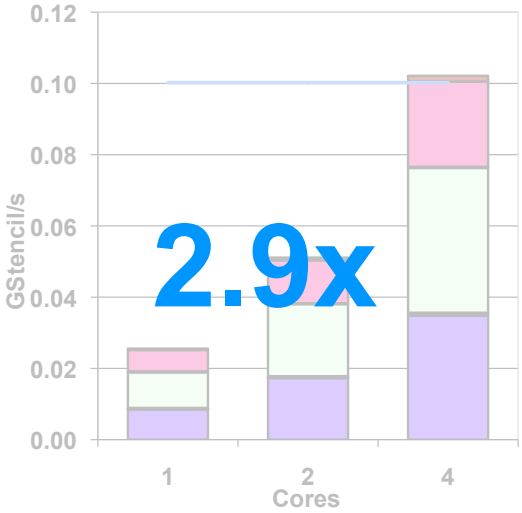
Intel Clovertown



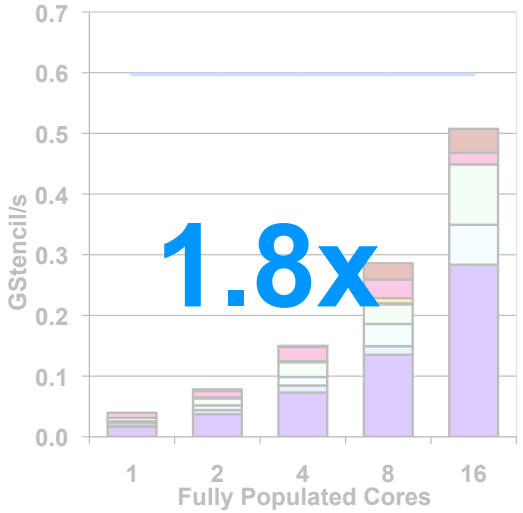
Intel Nehalem



AMD Barcelona



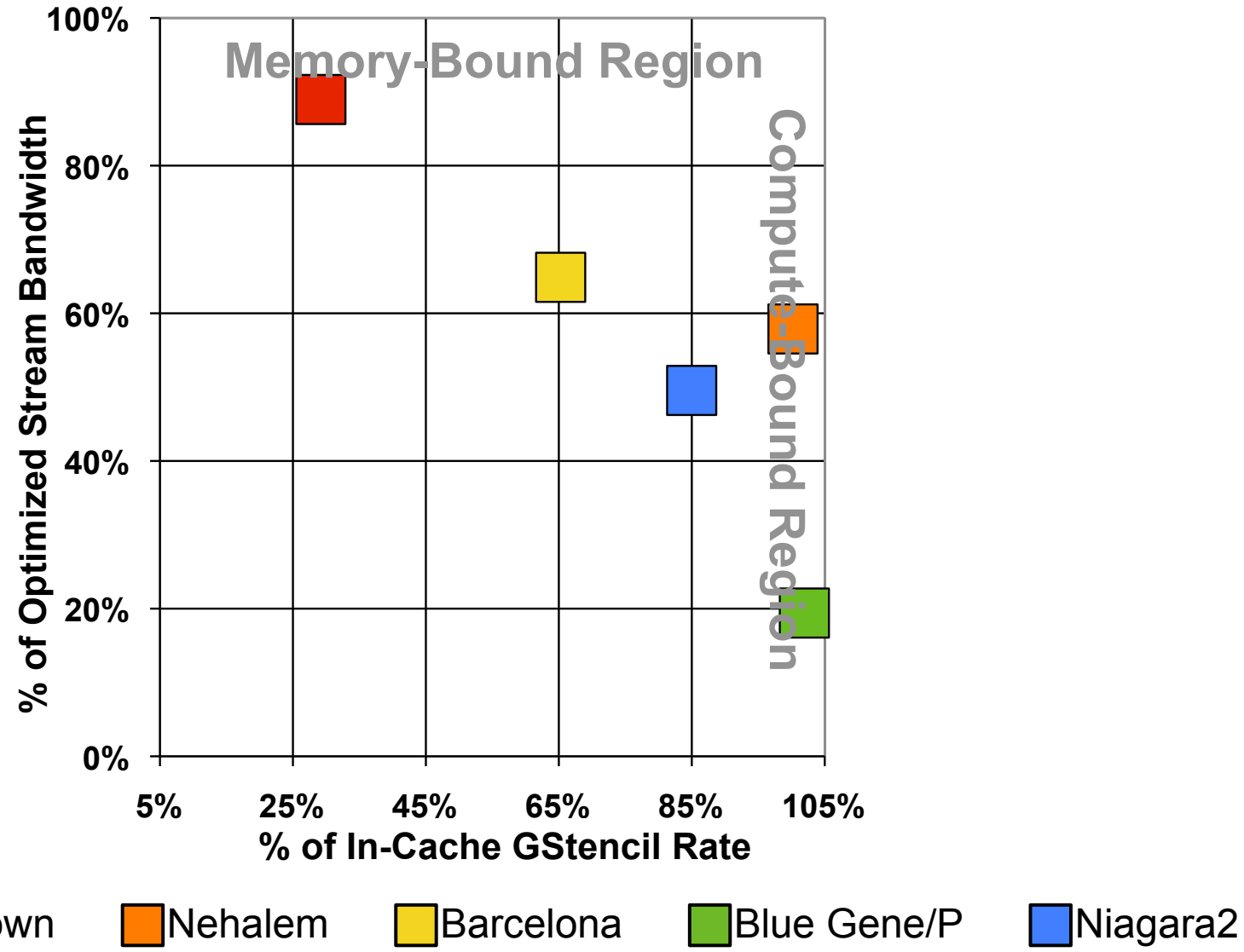
IBM Blue Gene/P



Sun Niagara2

Tuning Speedup Over Best Naïve Performance

How well can we do?
(3D 27-Point Stencil)

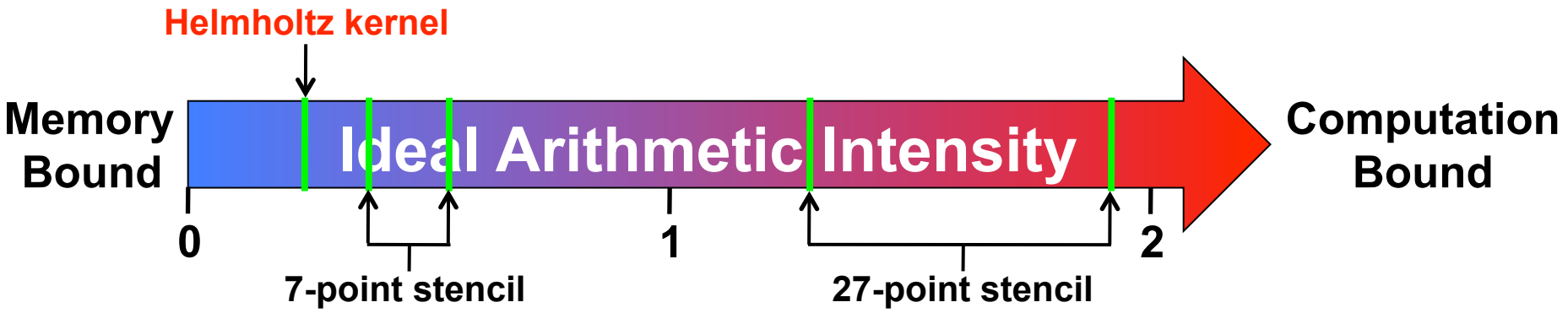


Outline

- Stencil Code Overview
- Cache-based Architectures
- Auto-tuning Description
- Stencil Auto-tuning Results
  - 3D 7-Point Stencil (Memory-Intensive Kernel)
  - 3D 27-Point Stencil (Compute-Intensive Kernel)
  - 3D Helmholtz Kernel

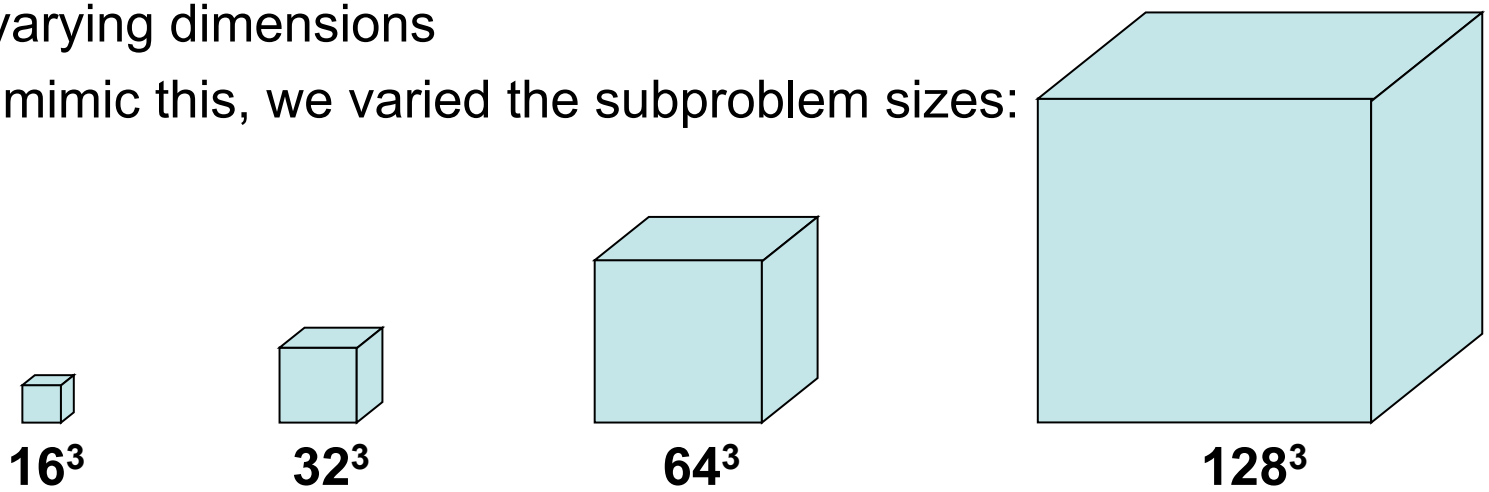
3D Helmholtz Kernel Problem

- The 3D Helmholtz kernel is very different from the previous kernels:
  - Gauss-Seidel Red-Black ordering
  - 25 flops per stencil
  - 7 arrays (6 are read only, 1 is read and write)
  - Many small subproblems- no longer one large problem
- Ideal AI is about 0.20
- This kernel should be memory-bound on most architectures:

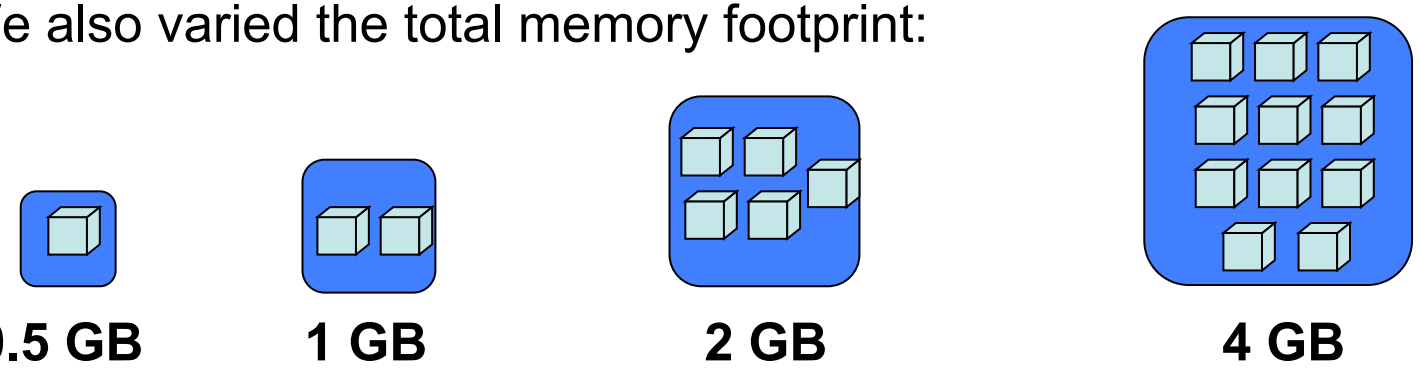


3D Helmholtz Kernel Problem

- Chombo (an AMR framework) deals with many small subproblems of varying dimensions
- To mimic this, we varied the subproblem sizes:

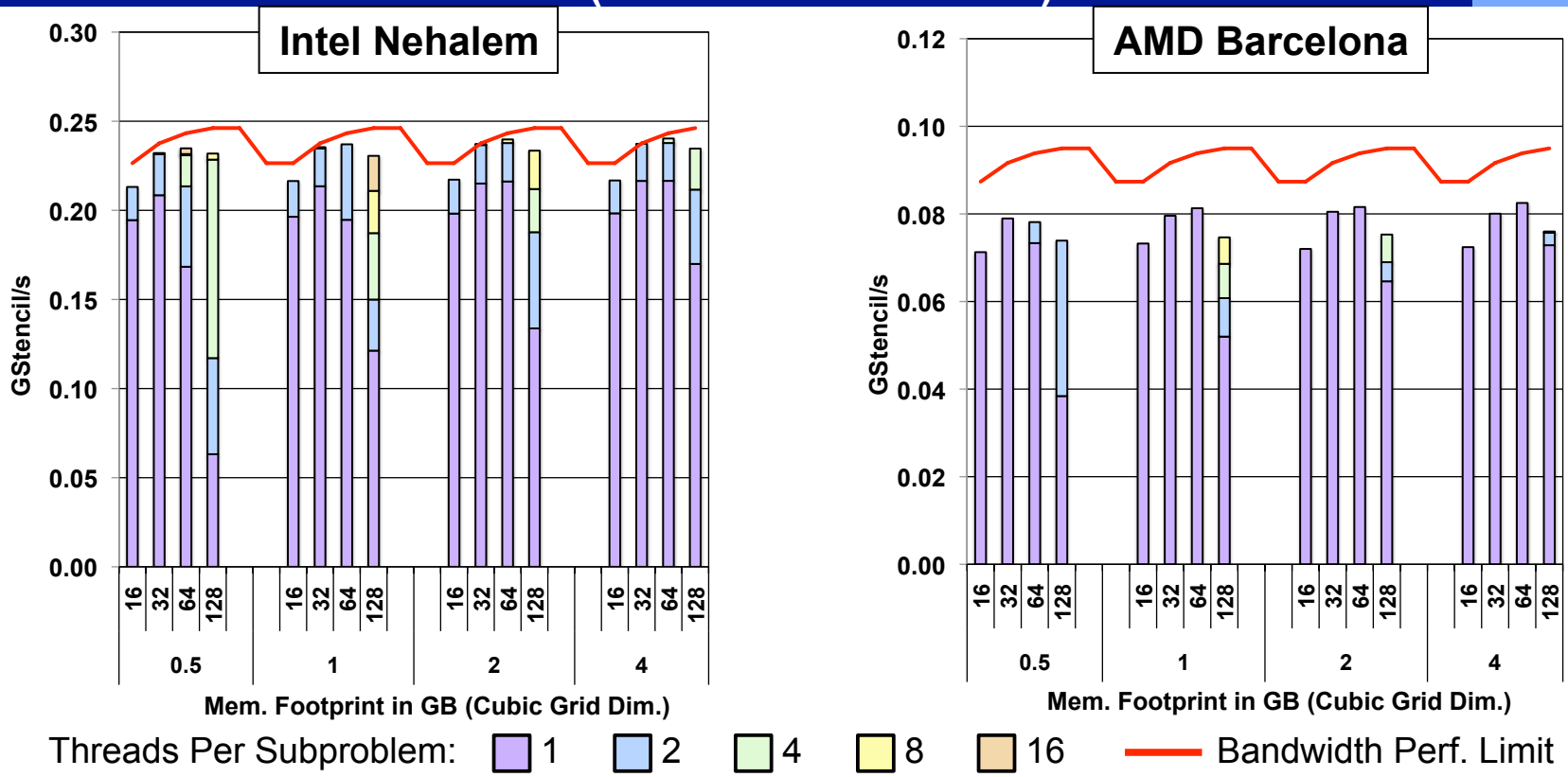


- We also varied the total memory footprint:



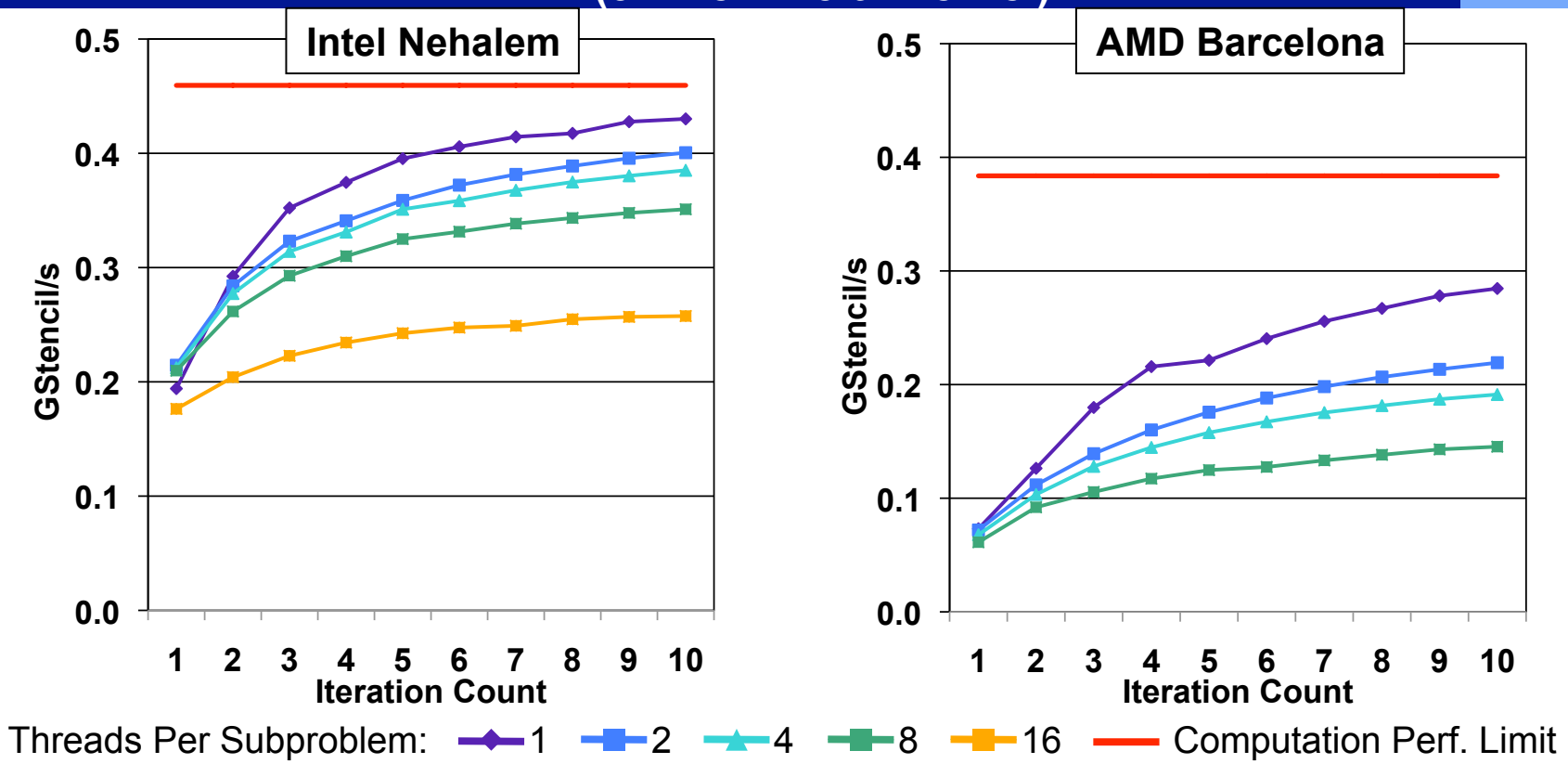
- We also introduced a new parameter- the number of *threads per subproblem*

Single Iteration
(3D Helmholtz Kernel)



- 1-2 threads per problem is optimal in cases where load balancing is not an issue
- If this trend continues, load balancing will be an even larger issue in the manycore era

Multiple Iterations
(3D Helmholtz Kernel)



- This is performance of 16^3 subproblems in a 0.5 GB memory footprint
- Performance gets worse with more threads per subproblem

Conclusions

- Compilers alone achieves poor performance
  - Typically achieve a low fraction of peak performance
  - Exhibit little parallel scaling
- Autotuning is essential to achieving good performance
  - 1.9x-5.4x speedups across diverse architectures
  - Automatic tuning is *necessary* for scalability
  - With few exceptions, the same code was used
- Ultimately, we are limited by the hardware
  - We can only do as well as Stream or in-core performance
  - The *memory wall* will continue to push stencil codes to be bandwidth-bound
- When dealing with many small subproblems, fewer threads per subproblem performs best
  - However, load balancing becomes a major issue
  - This is an even larger problem for the manycore era

Future Work

- Better Productivity:
  - Current Perl scripts are primitive
  - Need to develop an auto-tuning framework that has semantic knowledge of the stencil code (S. Kamil)
- Better Performance:
  - We currently do no data structure changes other than array padding
  - May be beneficial to store the grids in a recursive format using space-filling curves for better locality (S. Williams?)
- Better Search:
  - Our current search method does require expert knowledge to order the optimizations appropriately
  - Machine learning offers the opportunity for tuning with little domain knowledge and many more parameters (A. Ganapathi)

EECS
Electrical Engineering and Computer Sciences

Acknowledgements

BERKELEY PAR LAB

- Kathy and Jim for sure-handed guidance and knowledge during all these years
- Sam Williams for always being available to discuss research (and being an unofficial thesis reader)
- Rajesh Nishtala for being a great friend and officemate
- Jon Wilkening for being my outside thesis reader
- The Bebop group, including Shoaib Kamil, Karl Fuerlinger, and Mark Hoemmen
- The scientists at LBL, including Lenny Oliker, John Shalf, Jonathan Carter, Terry Ligocki, and Brain Van Straalen
- The members of the Parlab and Radlab, including Dave Patterson and Archana Ganapathi
- Many others that I don't have space to mention here…

- *I'll miss you all! Please contact me anytime.*

Supplemental Slides

Applications of this work

- Lawrence Berkeley Laboratory (LBL) is using stencil auto-tuning as a building block of its Green Flash supercomputer (Google: Green Flash LBL)
- Dr. Franz-Josef Pfreundt (head of IT at Fraunhofer-ITWM) used stencil tuning to improve the performance of oil exploration code

3D Helmholtz Kernel Problem