A Generalized Framework for Auto-tuning Stencil Computations

Shoaib Kamil[1,3], Cy Chan[4], **Samuel Williams**[1], Leonid Oliker[1], John Shalf[1,2], Mark Howison[3], E. Wes Bethel[1], Prabhat[1]

[1]Lawrence Berkeley National Laboratory (LBNL)
[2]National Energy Research Scientific Computing Center (NERSC)
[3]EECS Department, University of California, Berkeley (UCB)
[4]CSAIL, Massachusetts Institute of Technology (MIT)

SAKamil@lbl.gov

The Challenge: Productive Implementation of an Auto-tuner

Conventional Optimization

- Take one kernel/application
  - Perform some analysis of it
  - Research the literature for appropriate optimizations
  - Implement a couple of them by hand optimizing for one target machine.
  - Iterate a couple of times.

- Result:

  improve performance for **one** kernel on **one** computer.

Conventional Auto-tuning

- Automate the code generation and tuning process.
  - Perform some analysis of the kernel
  - Research the literature for appropriate optimizations
  - implement a code generator and search benchmark
  - explore optimization space
  - report best implementation/parameters

- Result:

  significantly improve performance for **one** kernel on **any** computer.

  i.e. provides ***performance portability***

- Downside:
  - autotuner creation time is substantial
  - must reinvent the wheel for every kernel

Generalized Frameworks for Auto-tuning

- Integrate some of the code transformation features of a compiler with the domain-specific optimization knowledge of an auto-tuner
  - parse high-level source
  - apply transformations allowed by the domain, but not necessarily safe based on language semantics alone
  - generate code + auto-tuning benchmark
  - explore optimization space
  - report best implementation/parameters

- Result:

  significantly improve performance for **any** kernel on **any** computer for a domain or motif.

  i.e. ***performance portability without sacrificing productivity***

Outline

1. Stencils
2. Machines
3. Framework
4. Results
5. Conclusions

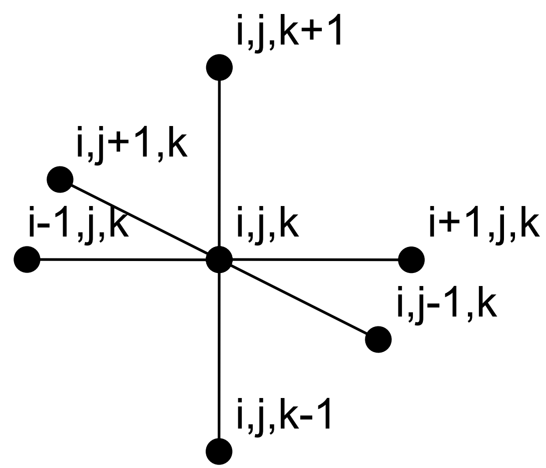
Benchmark Stencils

- Laplacian
- Divergence
- Gradient
- Bilateral Filtering

What's a stencil ?

- Nearest neighbor computations on structured grids (1D…ND array)

- stencils from PDEs are often a weighted linear combination of neighboring values
- cases where weights vary in space/time
- stencil can also result in a table lookup
- stencils can be nonlinear operators

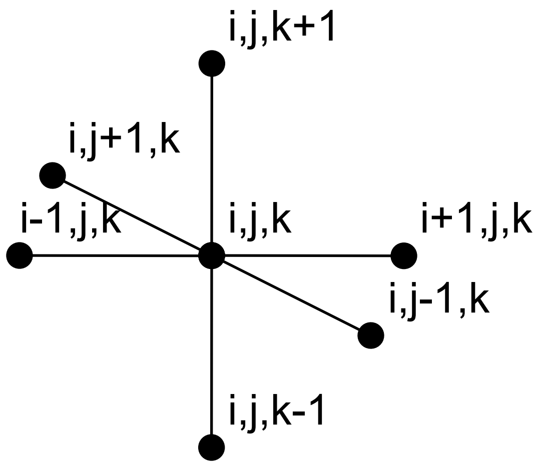


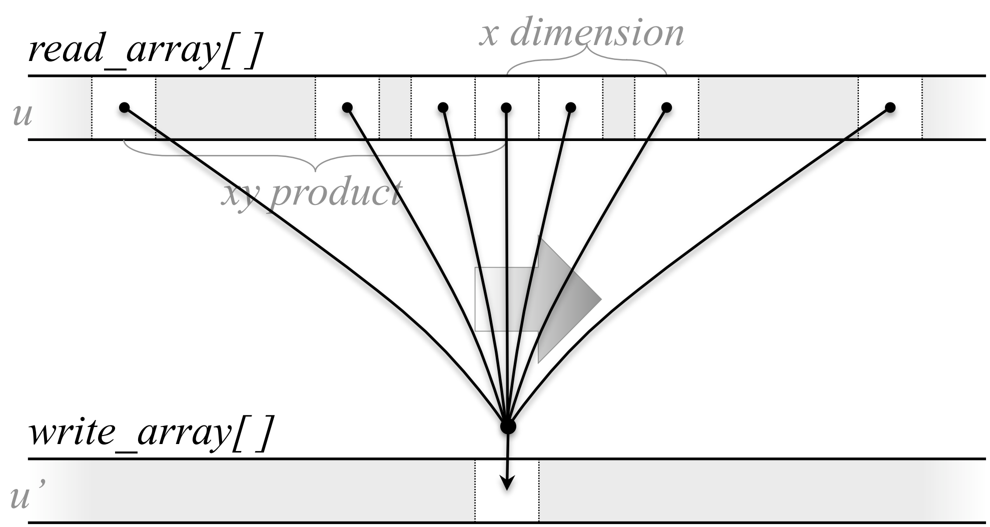
- caveat: We only examine implementations like Jacobi's Method *(i.e. separate read and write arrays)*

Laplacian Differential Operator

- 7-point stencil on scalar grid, produces a scalar grid

- Substantial reuse (+high working set size)
- **Memory-intensive** kernel
- Elimination of capacity misses may improve performance by **66%**

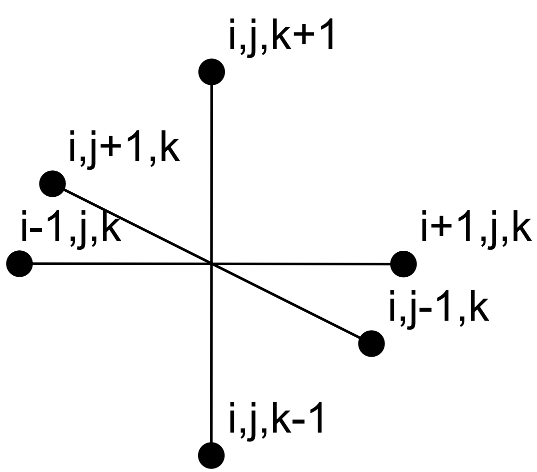


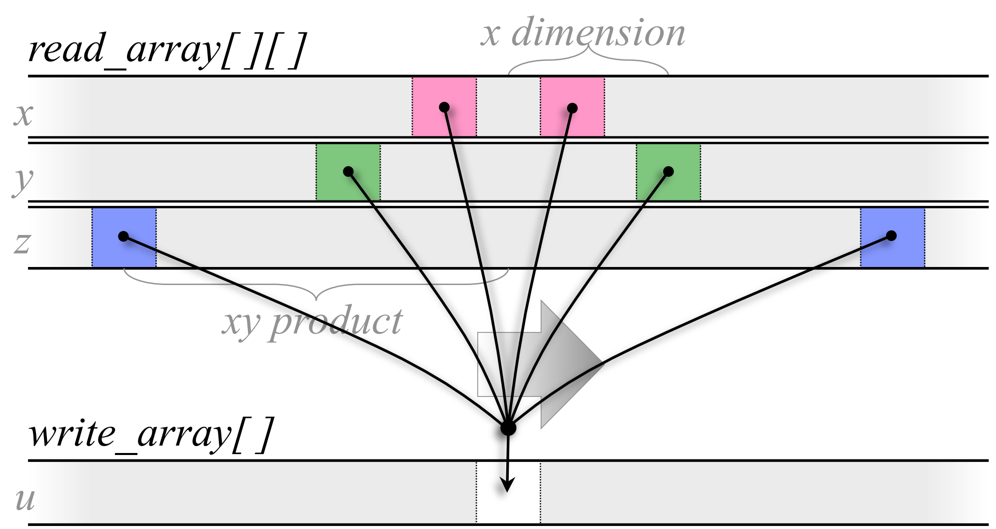


Divergence Differential Operator

- 6-point stencil on a vector grid, produces a scalar grid

- Low reuse per component.
- Only z-component demands a large working set
- **Memory-intensive** kernel
- Elimination of capacity misses may improve performance by **40%**



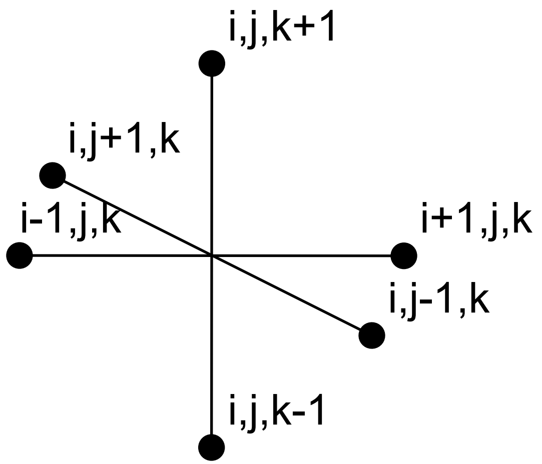


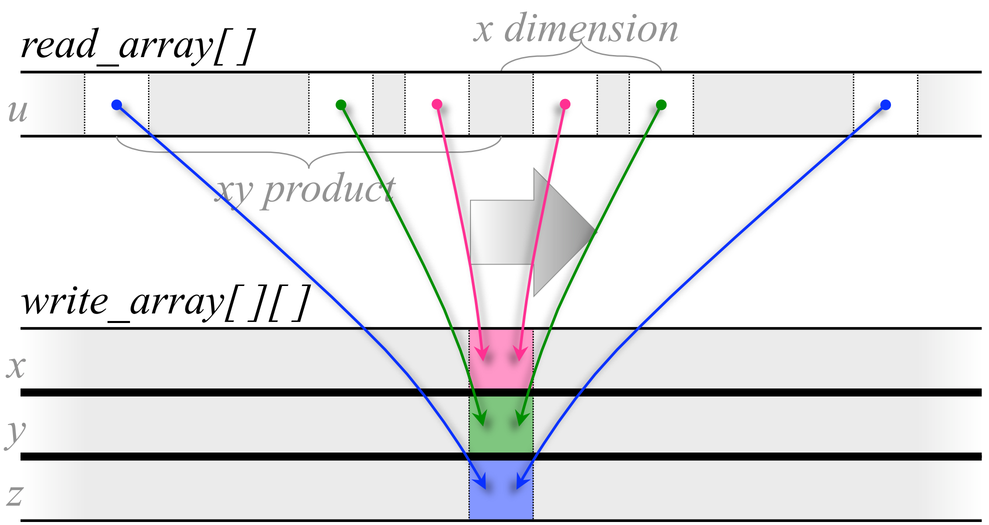
Gradient Differential Operator

FUTURE TECHNOLOGIES GROUP

- 6-point stencil on a scalar grid, produces a vector grid

- High reuse (like laplacian)
- High working set size
- three write streams (+ write allocation streams) = 7 total streams
- **Memory-intensive** kernel
- Elimination of capacity misses may improve performance by **30%**





3D Bilateral Filtering

- Extracted from a medical imaging application (MRI processing)
- Normal Gaussian stencils **smooth images**, but **destroy sharp edges**.
- This kernel performs **anistropic** filtering thus preserving edges.
- We may scale the size of the stencil (radius=3,5)
  - 7^3-pt or 11^3-pt stencils.
  - apply to dataset of 192 x 256x256 slices
  - originally 8-bit grayscale voxels, but processed as **32-bit floats**

3D Bilateral Filtering
(pseudo code)

- Each point in the stencil mandates a **voxel-dependent indirection**, and each stencil also requires one **divide**.

```
for all points (xyz) in x,y,z{
  voxelSum  = 0
  weightSum = 0
  srcVoxel = src[xyz]
  for all neighbors (ijk) within radius of xyz{
    neighborVoxel  = src[ijk]
    neighborWeight = table2[ijk]*table1[neighborVoxel-srcVoxel]
    voxelSum +=neighborWeight*neighborVoxel
    weightSum+=neighborWeight
  }
  dstVoxel = voxelSum/weightSum
}
```

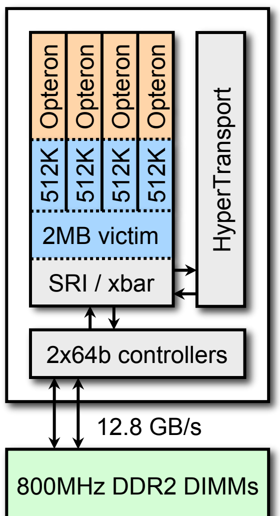
- Large radii results in extremely **compute-intensive** kernels with large working sets

Benchmark Machines

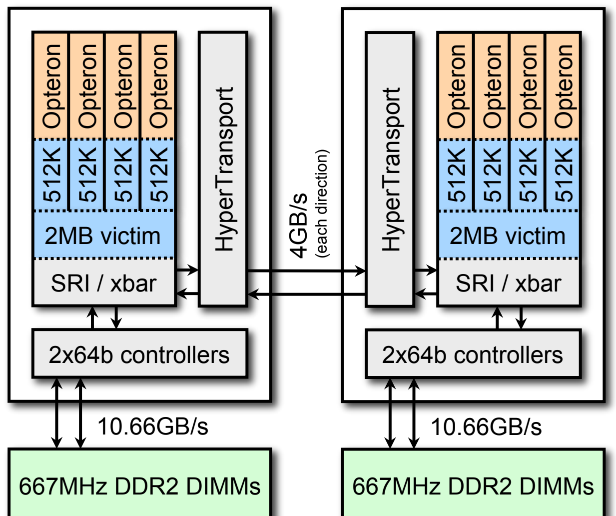
Multicore SMPs

- Experiments only explored parallelism within an SMP
- We use a Sun X2200 M2 as a proxy for the XT5 (e.g. Jaguar)
- We use a Nehalem machine as a proxy for possible future Cray machines.
- Barcelona/Nehalem are **NUMA**

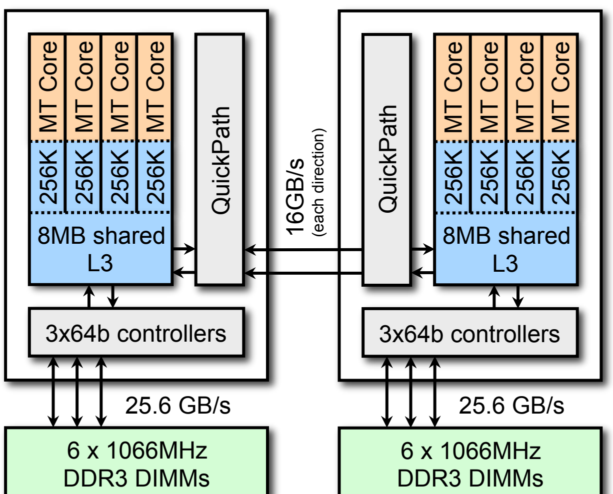
AMD Budapest (XT4)



AMD Barcelona (X2200 M2)



Intel Nehalem (X5550)



Generalized Framework for Auto-tuning Stencils

Copy and Paste auto-tuning

Overview

Given a F95 implementation of an application:

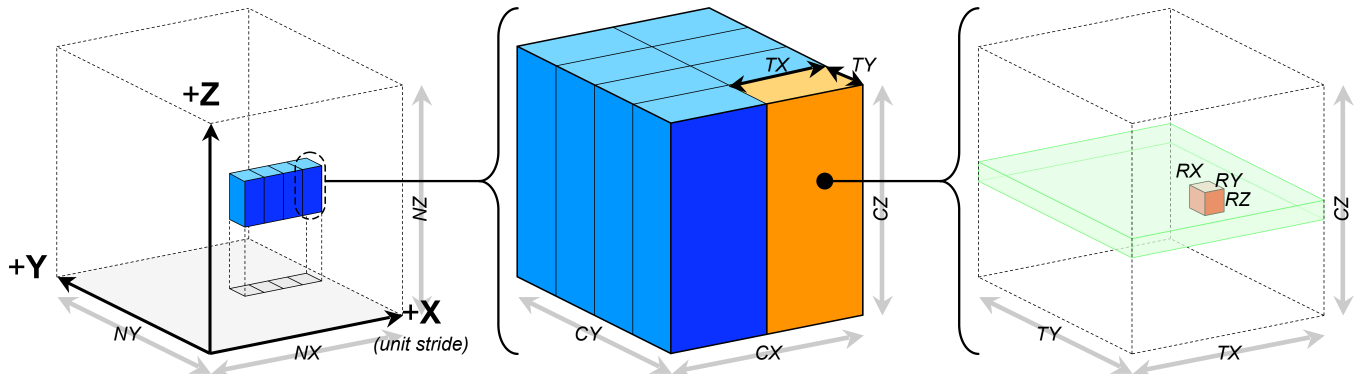
1. Programmer annotates target stencil loop nests
2. Auto-tuning System:
   - converts FORTRAN implementation into internal representation (AST)
   - builds a test harness
   - Strategy Engine iterates on:
     - apply optimization to internal representation
     - backend generation of optimized C code
     - compile C code
     - benchmark C code
   - using best implementation, automatically produces a library for that kernel/machine combination
3. Programmer then updates application to call optimized library routine

Strategy Engine:
Auto-parallelization

- The strategy engines can auto-parallelize cache blocks among hardware thread contexts.
- We use a single-program, multiple-data (SPMD) model implemented with POSIX Threads (Pthreads).
- All threads are created at the beginning of the application.
- We also produce an initialization routine that exploits the first touch policy to ensure proper NUMA-aware allocation.

Strategy Engine:
Auto-tuning Optimizations

- Strategy Engine explores a number of auto-tuning optimizations:
  - loop unrolling/register blocking
  - cache blocking
  - constant propagation / common subexpression elimination



(a) Decomposition of a Node Block into a Chunk of Core Blocks

(b) Decomposition into Thread Blocks

(c) Decomposition into Register Blocks

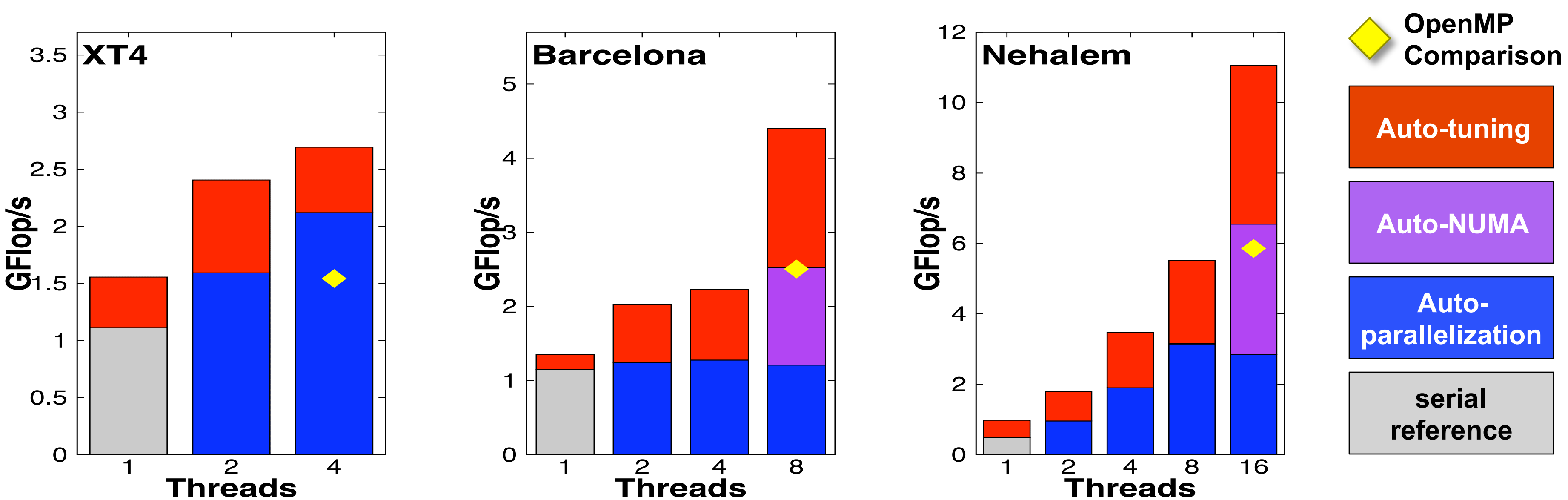
- Future Work:
  - cache bypass (e.g. *movntpd*)
  - software prefetching
  - SIMD intrinsics
  - data structure transformations

Experimental Results

NOTE: threads are ordered to exploit:
multiple threads within a core (Nehalem only),
then multicore,
then multiple sockets (Barcelona/Nehalem)

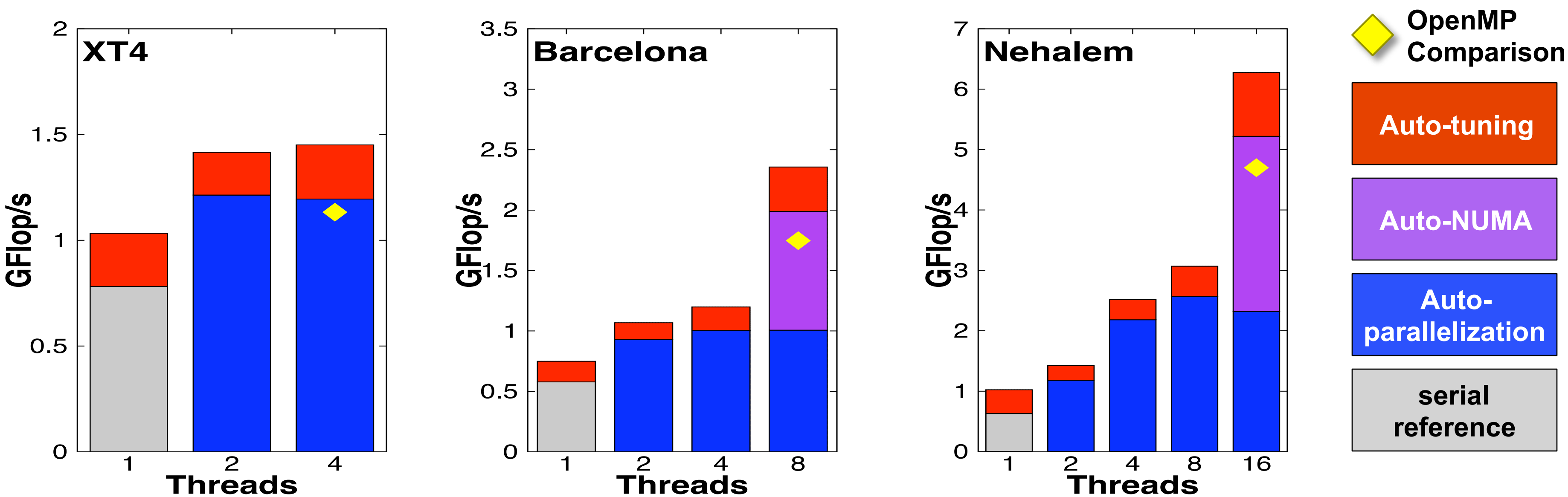
Laplacian Performance

- On the memory-bound architecture (Barcelona), auto-parallelization doesn't make a difference.
- Auto-tuning enables scalability.
- Barcelona is bandwidth-proportionally faster than the XT4.
- Nehalem is ~2.5x faster than Barcelona, and 4x faster than the XT4
- Auto-parallelization plus tuning significantly outperforms OpenMP.



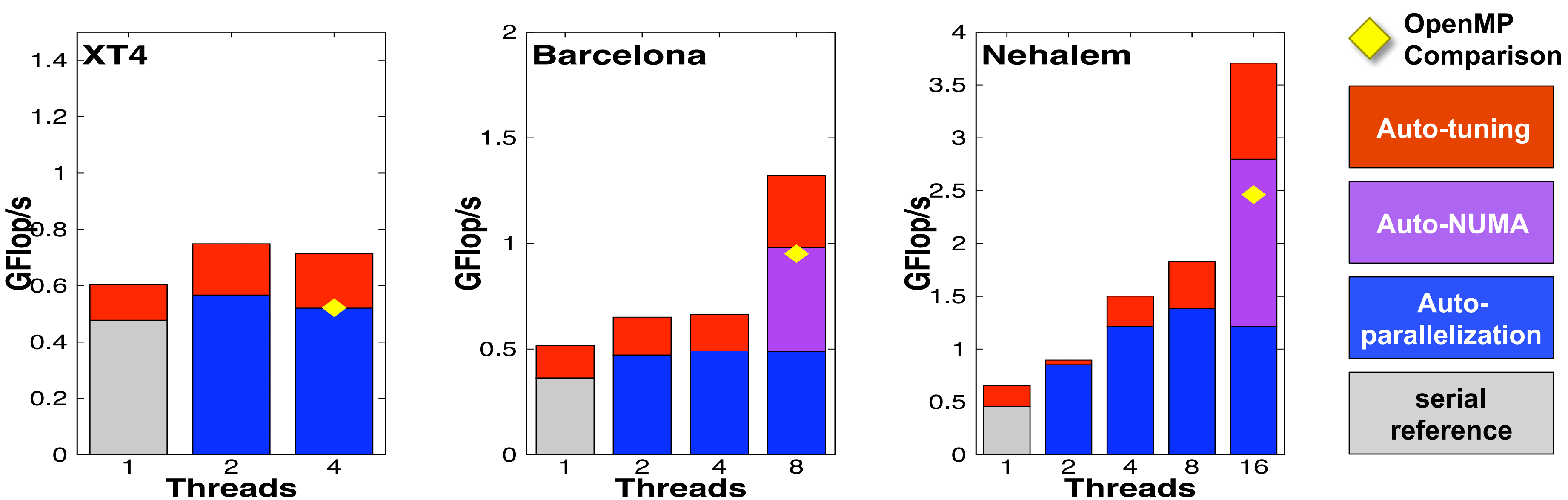
Divergence Performance

- **No changes to the framework were required (just drop in F95 code)**
- As there was less reuse in the Divergence than in Laplacian, there are fewer capacity misses.
- So auto-tuning has less to improve upon
- Nehalem is ~2.5x faster than Barcelona



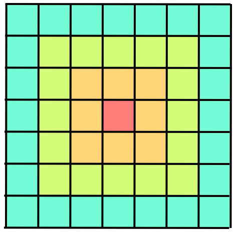
Gradient Performance

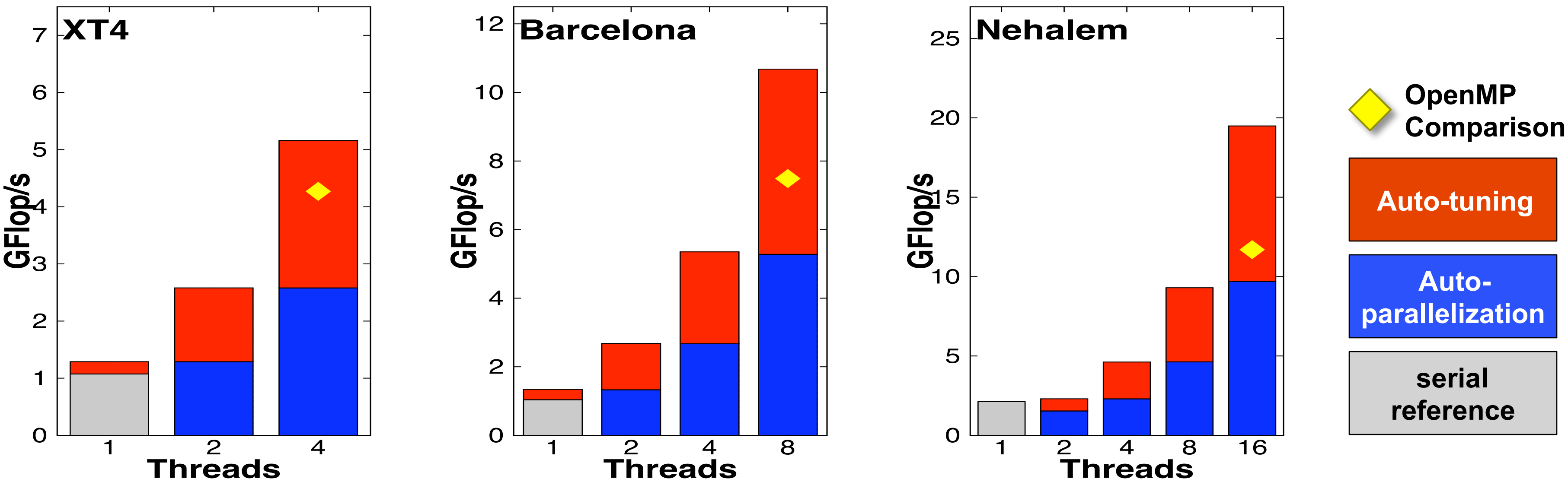
- **No changes to the framework were required (just drop in F95 code)**
- Gradient has moderate reuse, but a large number of output streams.
- Performance gains from auto-tuning are moderate (25-35%)
- Parallelization is only valuable in conjunction with auto-tuning



3D Bilateral Filter Performance
(radius=3)

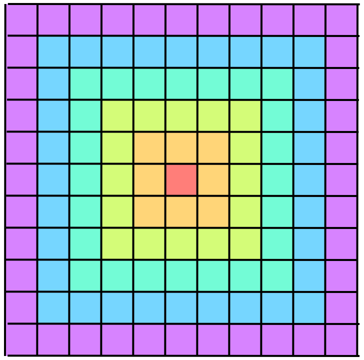
- **No changes to the framework were required (just drop in F95 code)**
- Essentially a 7x7x7 (343-pt) stencil
- Performance is much more closely tied to GHz instead of GB/s.
- Auto-parallelization yielded near perfect parallel efficiency wrt cores on Barcelona/Nehalem (Nehalem has HyperThreading)
- Auto-tuning significantly outperformed OpenMP (75% on Nehalem)

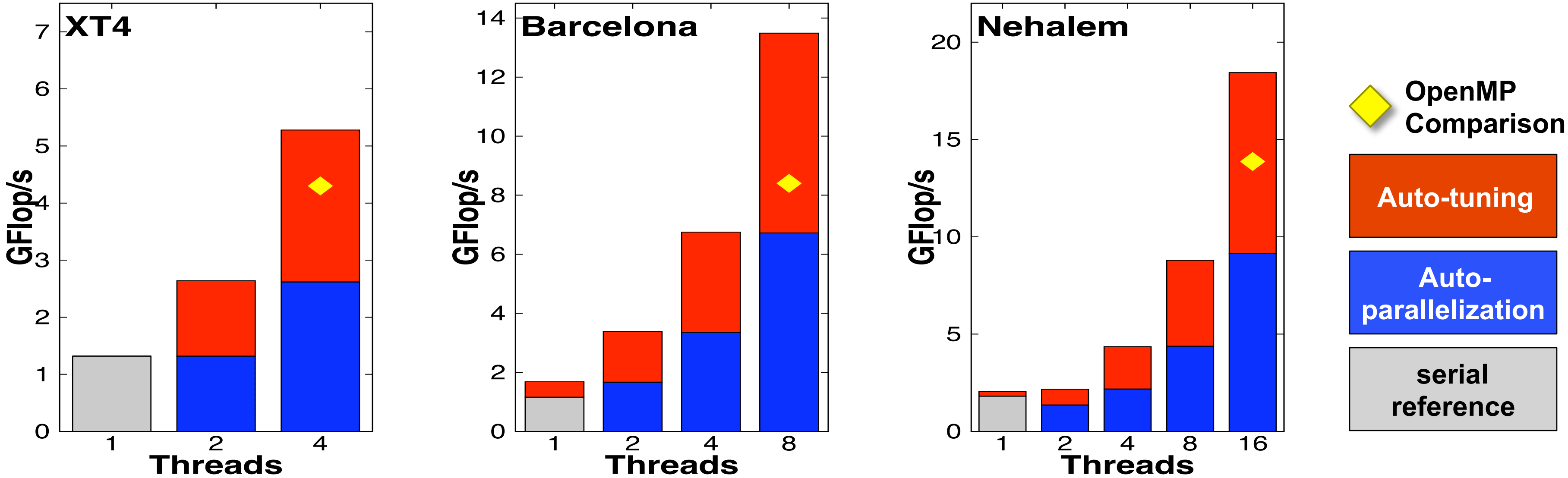




3D Bilateral Filter Performance
(radius=5)

- basically the same story as radius=3
- XT4/Nehalem delivered approximately same performance as they did with radius=3
- Barcelona delivered somewhat better performance.





Summary

Summary:
Framework for auto-tuning stencils

- **Dramatic step forward in auto-tuning technology**

- Although the framework required substantial up front work, it provides performance portability across the breadth of architectures **AND** stencil kernels.

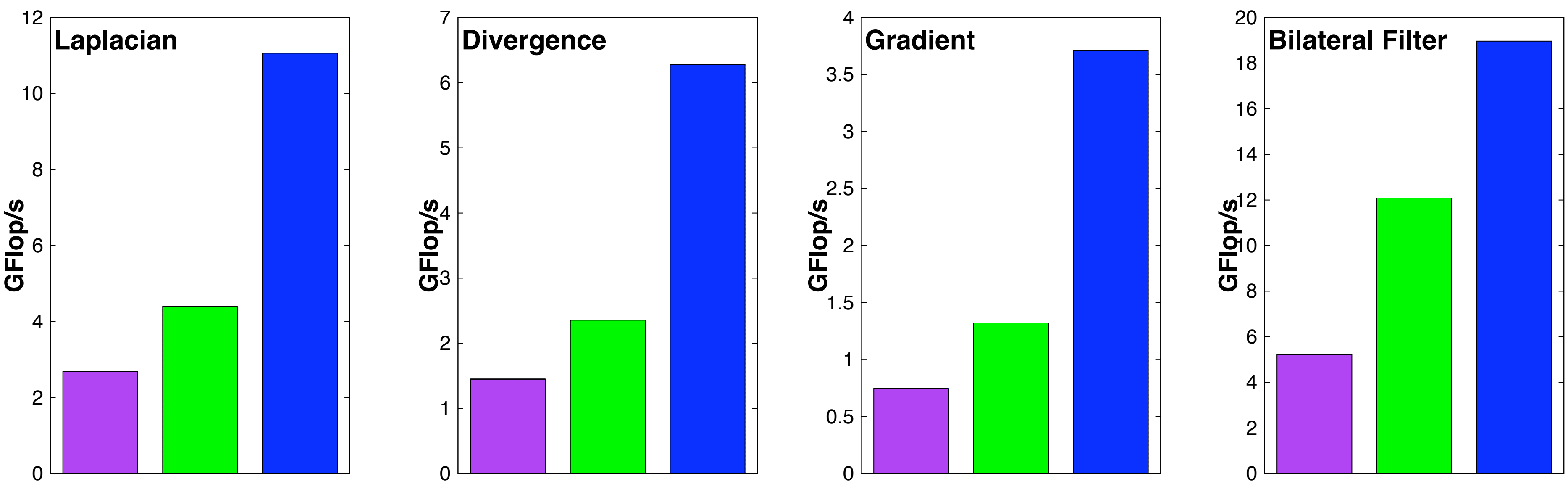
- Delivers very good performance, and well in excess of OpenMP.

- Future work will examine relevant optimizations
  - e.g. cache bypass would significantly improve gradient performance.

Summary:

Machine Comparison

- Barcelona delivers bandwidth-proportionally better performance on the memory-intensive differential operators.
- Surprisingly, Barcelona delivers ~2.5x better performance on the compute intensive bilateral filter.
- Nehalem clearly sustains dramatically better performance than either Opteron.
- Despite having a 15% faster clock, nehalem realizes a much better bilateral filter performance.



Acknowledgements

- Research supported by DOE Office of Science under contract number DE-AC02-05CH11231
- Microsoft (Award #024263)
- Intel (Award #024894)
- U.C. Discovery Matching Funds (Award #DIG07-10227)
- All XT4 simulations were performed on the XT4 (Franklin) at the National Energy Research Scientific Computing Center (NERSC)

Questions?