# A Generalized Framework for Auto-tuning Stencil Computations

Shoaib Kamil[†‡], Cy Chan[†⋆], Samuel Williams[†], Leonid Oliker[†], John Shalf[†]
Mark Howison[†], E. Wes Bethel[†], Prabhat[†]
[†]*CRD/NERSC, Lawrence Berkeley National Laboratory Berkeley, CA 94720*
[‡]*EECS Department, University of California at Berkeley, Berkeley, CA 94720*
[⋆] *CSAIL, Massachusetts Institute of Technology, Cambridge, MA 02139*

**ABSTRACT:** This work introduces a generalized framework for automatically tuning stencil computations to achieve superior performance on a broad range of multicore architectures. Stencil (nearest-neighbor) based kernels constitute the core of many important scientific applications involving block-structured grids. Auto-tuning systems search over optimization strategies to find the combination of tunable parameters that maximizes computational efficiency for a given algorithmic kernel. Although the auto-tuning strategy has been successfully applied to libraries, generalized stencil kernels are not amenable to packaging as libraries. Studied kernels in this work include both memory-bound kernels as well as a computation-bound bilateral filtering kernel. We introduce a generalized stencil auto-tuning framework that takes a straightforward Fortran expression of a stencil kernel and automatically generates tuned implementations of the kernel in C or Fortran to achieve performance portability across diverse computer architectures.

**KEYWORDS:** Multicore, XT4, Auto-tuning, Auto-parallelization

## 1 Introduction

As we enter the era of billion transistor chips, the computing industry has moved away from exponential scaling of clock frequency towards chip multiprocessors (CMPs) in order to better manage trade-offs among performance, energy efficiency, and reliability; however, The immaturity of the CMP approach has resulted in an ever-changing landscape of architectural features and memory hierarchy designs. Thus performance programmers are faced with enormous challenges in productively designing applications that effectively leverage the underlying computational resources, while still allowing for performance portability across the myriad of current and future CMP instantiations. Scientific progress will be substantially slowed without advanced programming models and tools that allow programmers to utilize massive on-chip concurrency.

The challenge of this decade for scientific computing is to create new programming models and tools that enable concise expression of fine-grained, explicit parallelism that is portable across a diversity of chip multiprocessor (CMP) instantiations. In this work, we present a novel approach to address these conflicting requirements for stencil-based computations using a generalized auto-tuning framework. Our framework takes a straightforward Fortran 95 stencil expression and automatically generates tuned implementations of the stencil in C or Fortran (we use the former in this work). This approach provides performance portability across diverse computer architectures and enables a viable migration path from existing application codes to codes that provide scalable intra-socket parallelism across a diversity of emerging chip multiprocessors – preserving portability and allowing for productive code design and evolution.

To demonstrate the flexibility and generality of our framework, we examine several stencil computations with a variety of different computational characteristics, arising from PDE solvers in climate science, image processing, and structural mechanics. Auto-parallelized and auto-tuned performance is then shown on three leading multicore platforms – the AMD Budapest processor in the form of the Cray XT4, an AMD Barcelona (a reasonable proxy for the XT5), and the Intel Xeon (Nehalem). Results show that our generalized methodology can deliver significant performance gains of up to $22\times$ speedup compared with the default serial version, while allowing portability across diverse architectural technologies. Furthermore, while our framework only required a few minutes of human effort to instrument each stencil code, the resulting code achieved performance comparable to previous hand-optimized auto-tuned code that required several months of tedious work to produce.

Overall we demonstrate that such domain-specific auto-tuners hold enormous promise for architectural efficiency, programmer productivity, performance portability, and algorithmic adaptability on existing and emerging multicore systems.

## 2 Stencil Computations

This study presents an autotuning framework for stencil computations on regular grids, which often arise from iterative finite-difference techniques sweeping over a spatial grid. At each point, a nearest-neighbor computation (a *stencil*) is performed: the point is updated with weighted contributions from a subset of points nearby in both time and space. Applications which use stencil computations include PDE solvers and adaptive mesh refinement [1].

The grid data structures are typically much larger than the data caches of current-generation microprocessors, and, along with the fact that many stencil computations use a relatively small number of neighbors at each point, such computations often produce large amounts of memory traffic and relatively little computation. As a result, performance is often bound by memory traffic. Previous work has investigated tiling optimizations [2, 10, 13, 14] which attempt to reorganize stencil calculations to exploit locality and reduce capacity misses; such optimizations are effective when the problem size is larger than the cache's ability to exploit temporal locality. Multicore stencil optimizations using hand-written kernel-specific tuners has shown the potential for tuning stencil calculations on modern microprocessors [3].

Although these recent studies have successfully shown auto-tuning's ability to achieve performance portability across the breadth of existing multicore processors, they have been constrained to a single stencil instantiation, thus failing to provide broad applicability to general stencil kernels due to the immense effort required to hand-write auto-tuners. In this work, we rectify this limitation by evolving the auto-tuning methodology into a generalized code generation framework, allowing significant flexibility compared to previous approaches that use prepackaged sets of limited functionality library routines. Our approach complements existing compiler technology. The framework leverages domain-specific knowledge of the application to enable both code and memory layout transformations that would otherwise be difficult to implement in a compiler. Compilers have a difficult time analyzing code to prove that code transformations are safe, and even more difficulty transforming data layout in memory. The framework sidesteps the complex task of analysis and presents a very simple, uniform, and familiar interface for expressing stencil kernels as a conventional Fortran expression.

### 2.1 Benchmark Kernels

To show the broad utility of our framework, we select three conceptually easy-to-understand, yet deceptively difficult to optimize stencil kernels arising from the application of the finite difference method to the Laplacian ($u_{next} \leftarrow \nabla^2 u$), Divergence ($u \leftarrow \nabla \cdot \mathbf{F}$), Gradient ($\mathbf{F} \leftarrow \nabla u$), In addition, we include a bilateral filtering algorithm with a stencil structure that is similar to other stencils where weights must be computed per-neighbor. The first three operators are implemented using the nearest-neighbor central-difference method on a 3D rectahedral block-structured grid using Jacobi's method (out-of-place), and benchmarked on a $256 \times 256 \times 256$ grid. The bilateral filtering operator has a variable radius operator that ranges from 1–11 grid points and is benchmarked on a $256 \times 256 \times 192$ grid, representing 192 slices of 2D imaging data. Note that although the code generator has no restrictions on data structure, for brevity, we only explore the use of the structure of arrays form for vector fields. Except for the bilateral filter, the kernels studied here have such low arithmetic intensity that they are expected to be memory-bandwidth bound, and thus deliver performance approximately equal to the product of their arithmetic intensity (AI) with the system stream bandwidth. The bilateral filter kernel exhibits AI that increases as stencil's radius increases.

*Laplacian:* The Laplacian stencil ($u_{next} \leftarrow \nabla^2 u$) reads 7 values from one scalar array, performs an 8-flop linear combination, and writes the result to another scalar array. With sufficient cache capacity, each subsequent stencil requires only a single element to be read from DRAM, due to reuse. Each stencil also requires a single write; if there is insufficient cache capacity, the kernel can benefit from cache blocking to eliminate capacity misses. A properly blocked kernel will incur 24 bytes of memory traffic, while capacity misses can increase the traffic to 40 bytes. Therefore, tuning block sizes can increase Laplacian performance by 66%.

*Divergence:* The Divergence stencil ($u \leftarrow \nabla \cdot \mathbf{F}$), is dramatically different as it must read only two elements from each of three different components of the vector field, perform an 8-flop linear combination, and write the result to a scalar grid. There is little reuse of data. Moreover, only accesses to the z-component will be challenged by capacity misses. As such, we expect the 48 read bytes requested from the cache to be easily reduced to 32 bytes, and with the appropriate cache blocking, as few as 24. As with the Laplacian, 16 bytes of traffic will be generated per stencil to update the destination array. Thus we expect auto-tuning to provide a 20% boost on machines with caches too small to fit the entire problem.

*Gradient:* In many ways, the Gradient stencil ($\mathbf{F} \leftarrow \nabla u$) is the opposite of the Divergence stencil. It reads 6 values from a scalar array, and performs three 2-flop stencils, writing those results to the three components of the output vector grid. Thus, naively one expects 24 bytes of read memory traffic (capacity misses), and another 48

| System<br>Chip Architecture | Cray XT4<br>AMD Budapest | Sun x2200<br>AMD Barcelona | Xeon X5550<br>Intel Nehalem |
|---|---|---|---|
| Type | superscalar<br>out-of-order | superscalar<br>out-of-order | superscalar<br>out-of-order |
| Clock (GHz) | 2.3 | 2.3 | 2.66 |
| DP GFlop/s per Core | 9.2 | 9.2 | 10.66 |
| Sockets per SMP node | 1 | 2 | 2 |
| Cores per Socket | 4 | 4 | 4 |
| Threads per Socket | 4 | 4 | 8 |
| DRAM Capacity | 8GB | 16GB | 12GB |
| DRAM Pin Bandwidth (GB/s) | 12.8 | 21.33 | 51.2 |
| DP GFlop/s | 36.8 | 73.6 | 85.33 |
| DP Flop:Byte Ratio | 2.88 | 3.45 | 1.66 |
| Compiler | gcc 4.3.2 | gcc 4.2.1 | gcc 4.3.2 |

**Table 1. Architectural summary of evaluated platforms.**

bytes of write traffic — an arithmetic intensity of 0.083. Cache blocking should asymptotically reduce the read traffic down to 8 bytes, but will do nothing for for the traffic resulting from writes. As such, auto-tuning can only improve the arithmetic intensity to 0.107.

*Bilateral Filtering:* Bilateral filtering, as defined by Tomasi [16], aims to perform anisotropic image smoothing using a low-cost, non-iterative formulation. The idea is to smooth images by computing the influence of nearby points in a way that removes noise "within regions," and that does not have the undesirable property of smoothing edge features. This formulation uses a straightforward, tunable estimate for region boundaries: a Gaussian-weighted difference in signal, or photometric space. The idea is that where a sharp edge exists, there will be a large difference in signal. That estimate is combined with a traditional Gaussian-weighted distance function to lessen the contribution from pixels distant in both geometric and signal space. In the bilateral filtering kernel, the output at each image pixel $d(i)$ is the weighted average of the influence of nearby image pixels $\bar{i}$ from the source image $s$ at location $i$. The "influence" is computed as the product of a geometric spatial component $g(i,\bar{i})$ and signal difference $c(i,\bar{i})$. While it is possible to precompute the portions of $k(i)$ contributed by $g(i,\bar{i})$, which depend only on the 3D Gaussian probability distribution function, the set of contributions from $c(i,\bar{i})$ are not known *a priori* as they depend upon the actual set of photometric differences observed across the neighborhood of $c(i,\bar{i})$ and will vary depending upon the source image contents and target location $i$. Arithmetic intensity varies based on filter size; for larger filters, the code becomes more and more computationally-bound. Note that because this filter is in 3D because it operates on a series of image, each representing data from a different layer (as is often found in MRI scans). The filter stencil operates across layers, allowing for better diagnostic imaging in medical applications.

## 3 Experimental Platforms

To evaluate our stencil auto-tuning framework, we examine a range of leading multicore designs: Cray XT4/AMD Budapest, a Sun X2200 M2/AMD Barcelona, and a Supermicro Xeon X5550 (Nehalem). A summary of key architectural features of the evaluated systems appears in Table 1.

**Cray XT4** The core building block of the Cray XT4 system is the single-socket AMD Opteron (Budapest) multicore SMP. Each Opteron core on the XT4 runs at 2.3 GHz and contains a 64KB L1 cache and a 512KB L2 victim cache. In addition, each chip instantiates a 2MB L3 quasi-victim cache shared among all four cores. All core prefetched data is placed in the L1 cache of the requesting core, whereas all DRAM prefetched data is placed into a dedicated buffer. The Opteron includes two DDR2-800 memory controllers delivering up to 12.8 GB/s of DRAM bandwidth.

**Sun X2200 M2** The Opteron 2356 (Barcelona) is AMD's flagship quad-core processor offering. Unlike Budapest, one can build multi-socket SMPs from Barcelona. Aside from the multisocket support (an additional HyperTransport link providing cache coherency and NUMA access to a second socket's DRAM) the Opteron cores are identical to those in the XT4. Each Barcelona socket in this X2200 machine is connected to its own bank of DDR2-667 DIMMs. The lower DIMM speed only delivers up to 10.66GB/s per socket, or an aggregate 21.33GB/s for the dual-socket system. Our dual-socket X2200 machine functions as a proxy for the XT5 system.

**Intel Nehalem:** The recently released Nehalem is the successor to the Intel "Core" architecture and represents a dramatic departure from Intel's previous multiprocessor designs. Whereas the Clovertown and prior architectures used a shared frontside bus (FSB) to connect all processors to a common northbridge chip which in turn provided access to memory or other peripherals, the Nehalem adopts a modern multisocket architecture similar to the Opteron: memory controllers have been integrated on-chip, requiring an inter-chip network. The resultant QuickPath Interconnect (QPI) is similar to HyperTransport in that it must handle access to remote memory controllers, coherency, and access to I/O.

Two other architectural innovations were incorporated into Nehalem: two-way simultaneous multithreading (SMT), and TurboMode, which allows a single core to operate faster than the set clock rate under certain workloads. On our machine, TurboMode is disable due to its inconsistent timing behavior. The system in this study is a dual socket 2.66 GHz Xeon X5550 (Gainestown) with two thread contexts per core and four cores per socket; a total of 16 hardware thread contexts in this dual-socket machine. Each socket integrates three DDR3 memory controllers operating at 1066MHz providing up to 25.6GB/s of DRAM bandwidth to each socket.

## 4 Transformation Framework Overview

Stencil applications use a wide variety of data structures in their implementations, representing grids of multiple dimensionalities and topologies. Furthermore, the details of the underlying stencil applications call for a myriad of numerical kernel operations. Thus, building a static auto-tuning library in the spirit of ATLAS [19] or OSKI [18] (dense and sparse matrix algebra) to implement the many different stencil kernels is infeasible; as it would require a unique auto-tuner for each stencil instantiation. Although previous work [3, 9] has shown the effectiveness of auto-tuning for stencil operations, the tuning systems themselves were rather primitive, consisting of simple Perl scripts that generate optimized versions for a single kernel. Additionally, the scripts are unable to test all combinations of possible optimizations due to their lack of semantic stencil knowledge.

This work presents a proof-of-concept of a generalized auto-tuning approach, which uses a domain-specific transformation and code-generation framework combined with a fully-automated search to replace stencil kernels with their optimized versions. The interaction with the application program begins with simple annotation of the loops targeted for optimization. The search system then extracts each designated loop and builds a test harness for that particular kernel instantiation. Next, the search system uses the transformation and generation framework to apply our suite of auto-tuning optimizations, running the test harness for each candidate implementation to determine its optimal performance. After the search is complete, the optimized implementation is built into an application-specific library that is called in place of the original.

### 4.1 Parsing

The front-end to the tranformation engine parses a description of the stencil in a domain-specific language. For simplicity, we use a subset of Fortran 95, since many stencil applications are already written in some flavor of Fortran. Due to the modularity of the transformation engine, a variety of front-end implementations are possible. The result of parsing in our preliminary implementation is an *Abstract Syntax Tree* (AST) representation of the stencil, on which subsequent transformations are performed.

Currently, several restrictions exist in the domain of parsable and auto-tunable kernels handled by our framework. In particular, the system requires Jacobi kernels (no overwriting in-place) with perfectly-nested loops in a rectangular domain. In addition, array indexing is fairly simple: only additive (or subtractive) offsets are currently allowed. As the auto-tuning framework matures, we plan on relaxing these restrictions. However, many important existing stencil computations require little or no modification to meet our current requirements.

### 4.2 Transformation and Code Generation Overview

The largest component of the auto-tuning framework is the transformation engine and the backend code generation engine. The transformation engine operates on the AST, transforming the abstract structure of the program while preserving the original intent of the programmer. Serial backend targets generate portable C and Fortran code, while parallel targets include `pthreads` C code designed to run on a variety of cache-based multicore processor SMPs.

Once the intermediate form is created from the front-end description, it is manipulated by the transformation engine across our spectrum of auto-tuned optimizations. The intermediate form and transformations are expressed in portable Lisp code using the portable and lightweight ECL compiler [4], making it simple to interface with the parsing front-ends (written in Flex and YACC) and preserving portability across a wide variety of architectures.

Because optimizations are expressed as transformations on the AST, it is possible to combine them in ways that would otherwise be difficult using simple string substitution. For example, it is straightforward to apply register blocking either before or after cache-blocking the loop, allowing for a comprehensive exploration of optimization configurations.

In the rest of this section, we discuss serial transformations and code generation; auto-parallelization and parallel-specific transformations and generators are explored in Section 5.

| Category | Optimization Parameter | Name | Parameter Tuning Range |
|---|---|---|---|
| Data Allocation | NUMA Aware | | ✓ |
| Domain Decomposition | Core Block Size | *CX* | NX |
| | | *CY* | {8...NY} |
| | | *CZ* | {128...NZ} |
| | Thread Block Size | *TX* | CX |
| | | *TY* | CY |
| | | *TZ* | CZ |
| | Chunk Size | | $\{1 \ldots \frac{NX \times NY \times NZ}{CX \times CY \times CZ \times NThreads}\}$ |
| Low Level | Array Indexing | | ✓ |
| | Register Block Size | *RX* | {1...8} |
| | | *RY* | {1...2} |
| | | *RZ* | {1...2} |

**Table 2. Attempted optimizations and the associated parameter spaces explored by the auto-tuner for a $256^3$ stencil problem ($NX, NY, NZ = 256$). All numbers are in terms of doubles.**

## 4.3 General Optimization Transformations

Several common optimizations have been implemented in the framework as AST transformations, including loop unrolling/register blocking (to improve innermost loop efficiency), cache blocking (to expose temporal locality and increase cache reuse), and arithmetic simplification/constant propagation. These optimizations are implemented to take advantage of the specific domain of interest: Jacobi-like stencil kernels of arbitrary dimensionality. Future transformations will include those shown in previous work [3]. In particular, we will strive for better utilization of SIMD instructions and common subexpression elimination (to improve arithmetic efficiency), use of cache bypass (to eliminate unnecessary cache fills), and use of explicit software prefetching (to reduce wasted cache bandwidth). Additionally, future work will support aggressive memory and code structure transformations.

Although the current set of optimizations may overlap existing compiler optimizations, future speedup approaches such as memory structure transformations will be beyond the scope of compilers — since they are specific to stencil-based computations. Additionally, the fact that our framework's transformations yield code that outperforms compiler-only optimized versions highlights that compiler algorithms cannot always prove that these (safe) optimizations are allowed; the domain-specific knowledge embodied in the auto-tuner can apply these optimizations without needing to understand general (non-stencil) code.

## 5 Parallelization and Auto-Tuning

Given the stencil transformation framework, we now present parallelization code generation and our auto-tuning methodology. The shared-memory parallel code generators leverage the serial code generation routines to produce the version run by each individual thread. Because the mechanisms for parallelization are specific to each architecture, both the strategy engines and the code generators must be tailored to the desired targets. For the cache-based architectures, we chose `pthreads` for lightweight parallelization.

Note that for serial programs, a separate code generation phase logically follows immediately after loop transformations have been applied to the AST. For parallel programs, these conceptually disparate steps become an integrated process. Since the parallelization strategy influences code structure, the AST must be modified to reflect the chosen strategy. Additionally, since the valid parameter space of many serial optimizations depends on platform-specific parallelization parameters, it is necessary to search both spaces of optimizations simultaneously rather than sequentially.

### 5.1 Auto-Tuning Strategy Engines

The auto-tuning framework examines the set of valid independent transformations and measures the performance of each potential optimization combination within a defined platform-specific search space. The explored parameter space is enumerated by platform specific *strategy engines*, which are designed to examine the region of the full auto-tuning search space that best utilizes the underlying architecture's cache capacity. For example, cache blocking in the unit stride dimension might be practical on some

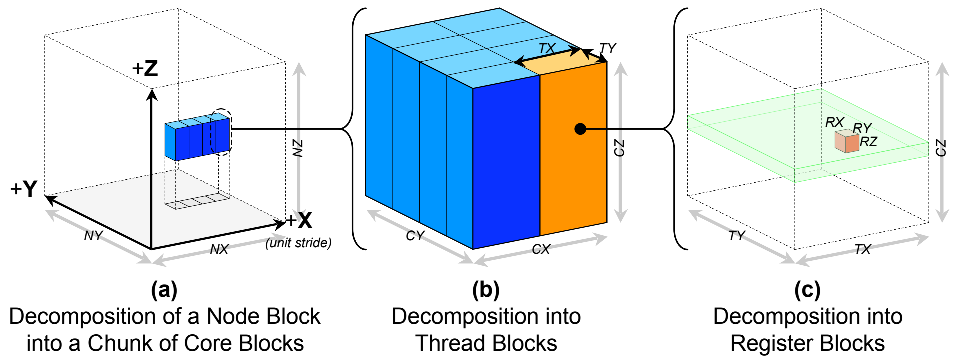
+Z
+Y
+X
(unit stride)
NY
NX
NZ
TX
TY
CY
CX
CZ
RX
RY
RZ
TY
TX
CZ
(a)
Decomposition of a Node Block into a Chunk of Core Blocks
(b)
Decomposition into Thread Blocks
(c)
Decomposition into Register Blocks

**Figure 1. Four-level problem decomposition: In (a), a *node block* (the full grid) is broken into smaller *chunks*. All the *core blocks* in a chunk are processed by the same subset of threads. One core block from the chunk in (a) is magnified in (b). A properly sized core block can avoid capacity misses in the last level cache. A single *thread block* from the core block is magnified in (c). A thread block should exploit common resources among threads. Finally, the magnified thread block in (c) is decomposed into *register blocks*, which exploit data level parallelism.**

architectures, but on AMD's Opteron or Intel's Nehalem, the reliance on hardware prefetchers makes such a transformation detrimental [2]. Table 2 shows the attempted optimizations and associated parameter search space explored by our auto-tuner for each of the platform targets. Future work will include more intelligent search mechanisms such as hill-climbing or machine learning techniques [6].

### 5.2 Multicore Parallelization for Cache-Based Systems

Following the effective blocking strategy presented in previous studies [3], we decompose the problem space into *core blocks*. The size of these core blocks is then tuned to avoid capacity misses in the last level cache. Each core block is further divided into *thread blocks* such that threads that share a common cache can cooperate on a core block. Though our code generator is capable of utilizing variable thread blocks, we set the size of the thread blocks equal to the size of the core blocks to help reduce the size of the auto-tuning search space. The threads of a thread block are then assigned *chunks* of contiguous core blocks in a round robin fashion until the entire problem space has been accounted for. Finally each thread's stencil loop is *register blocked* to best utilize registers and functional units. The auto-tuner tunes core block size, thread block size, chunk size, and register block size. Further, NUMA-aware memory allocation is implemented by pinning threads to the hardware and taking advantage of first-touch page mapping policy during data initialization. A visualization of the stencil domain decomposition reproduced from [3] is shown in Figure 1 .

## 6 Performance Evaluation

In this section, we examine the performance quality of our auto-parallelizing and auto-tuning framework across the four evaluated architectural platforms. We begin by examining the impact of our framework on each of the three differential operator kernels in Figures 2–5, showing performance of: the original serial kernel (gray), auto-parallelization (blue), auto-parallelization with NUMA-aware initialization (purple), and auto-tuning (red). Overall, results are ordered such that threads first exploit multithreading within a core (e.g. Nehalem), then multiple cores on a socket, and finally multiple sockets (e.g. Barcelona and Nehalem).

We next quantify the impact of our auto-parallelization by comparing results with straightforward OpenMP instrumentation of the baseline stencil kernels (shown as yellow diamonds in Figures 2–4). Finally, Figure 6 presents a cross-architectural comparison in terms of raw performance.

### 6.1 Laplacian Performance

Laplacian kernel performance results are shown in Figure 2. Auto-parallelization (blue) has differing impact depending on the underlying hardware's characteristics. Although the AMD Budapest (XT4) sees modest benefits from auto-parallelization, the AMD Barcelona sees little benefit, likely due to the fact that a single core can nearly saturate an entire socket's available bandwidth — it has the same computational capability per socket as the XT4, but 17% less bandwidth. On dual-socket SMPs such as the Nehalem and the Sun X2200 (and by extension the Cray XT5), NUMA optimization (purple) is essential for effectively utilizing both sockets, achieving roughly a doubling

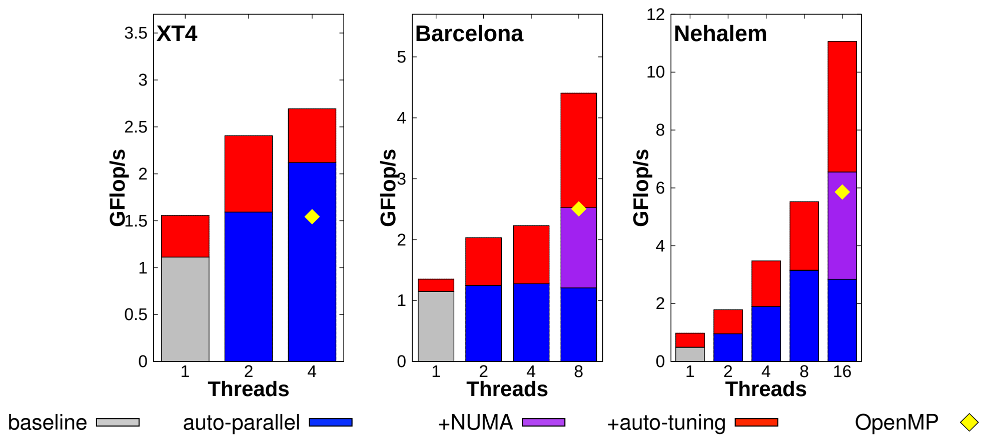

**Figure 2.** **Laplacian performance as a function of auto-parallelization and auto-tuning. For comparison, the yellow diamond shows performance achieved using the original stencil kernel with OpenMP pragmas and NUMA-aware initialization.**

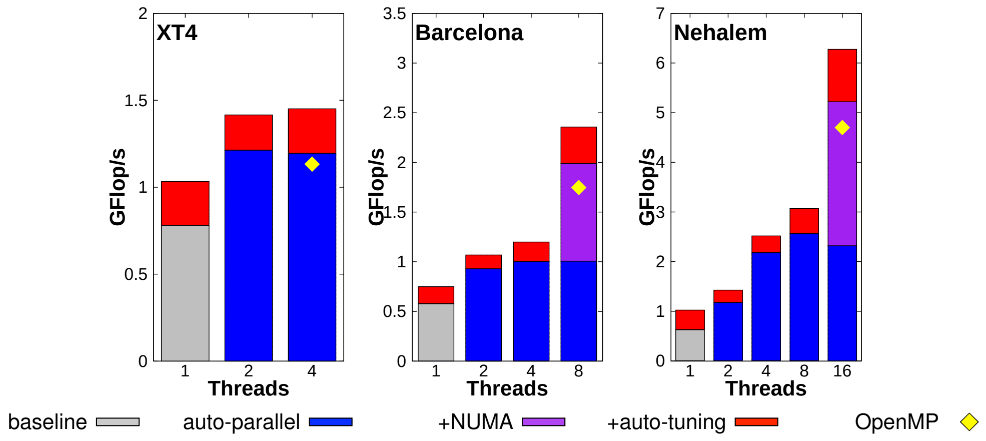

**Figure 3.** **Divergence performance as a function of auto-parallelization and auto-tuning. For comparison, the yellow diamond shows performance achieved using the original stencil kernel with OpenMP pragmas and NUMA-aware initialization.**

in performance. For the XT4, NUMA optimization yields no benefit, since there is a single socket.

Examining the impact of auto-tuning (red) on the Laplacian kernel shows little improvement on the serial version, as a single core may have full use of the socket's cache and DRAM bandwidth resources. However, after auto-parallelization, auto-tuning becomes a critical component that can substantially reduce capacity misses and thereby dramatically improve arithmetic intensity. Thus, the XT4 and Barcelona systems delivered $1.3\times$ and $1.7\times$ improvement respectively at the highest concurrency. The Nehalem machine also demonstrates a $1.7\times$ performance improvement.

Overall for the Laplacian computation, auto-parallelization coupled with auto-tuning improves performance by $2.4\times$ for the XT4, $3.8\times$ for Barcelona, and $22\times$ for Nehalem.

## 6.2 Divergence Performance

Figure 3 presents Divergence kernel performance. Results show auto-parallelization and NUMA optimization benefits similar to the Laplacian results.

Examining the impact of auto-tuning, we see somewhat lower benefits than in the Laplacian case, with speedups of $1.2\times$ all three systems.

Overall for the Divergence computation, auto-parallelization coupled with auto-tuning improves performance by $2.2\times$ on the XT4, $4.1\times$ on the Barcelona, and $10\times$ on Nehalem.

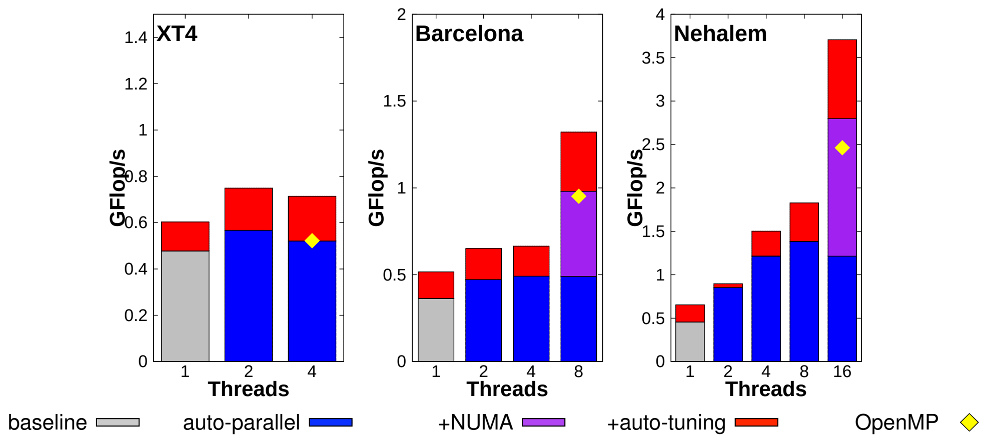

**Figure 4.** **Gradient performance as a function of auto-parallelization and auto-tuning. For comparison, the yellow diamond shows performance achieved using the original stencil kernel with OpenMP pragmas and NUMA-aware initialization.**

## 6.3 Gradient Performance

Figure 4 presents Gradient stencil results. Recall that of our three differential operator kernels, Gradient has the most demanding memory bandwidth requirements and significant reuse of the read arrays. Note that the caches require write-allocate with write-back to the memory. As such, on a write-miss, the entire cache line will be filled. Subsequently, the cache line will be evicted and written back to DRAM. Thus, for each write of a component to the output grid, 16 bytes of memory traffic will be generated. Future work will explore cache-bypass to eliminate this fill traffic. When examining the benefit of auto-parallelization, we once again see that the Opteron is heavily memory bound with one core, but still see a substantial improvement over the baseline due to more efficient use of the available memory bandwidth.

With respect to auto-tuning, we expect a higher performance impact than the Divergence kernel, given Gradient's higher reuse pattern. The Gradient kernel presents more opportunities for data reuse, so auto-tuning is expected to deliver a more substantial boost than for the Laplacian. This theory is borne out by the demonstrated improvements of $34\%$ on Barcelona and $32\%$ on Nehalem. Interestingly, the XT4 is unable to increase performance going from 2 to 4 cores, probably due to this kernel's high write bandwidth requirements. Auto-tuning increases performance by $31\%$ on the XT4 at 2 cores.

Overall, auto-parallelization coupled with auto-tuning improves performance of the Gradient kernel by $1.6\times$ on the XT4, $3.6\times$ on Barcelona, and $8.1\times$ on Nehalem.

## 6.4 Bilateral Performance

The bilateral filtering stencil is somewhat different than the previous three kernels, in that it exposes much more computation relative to memory traffic. Depending on the value of the filter radius, the kernel is generally compute-bound. Figure 5 shows the performance of the vis kernel with a radius of 3 (top) and 5 (bottom). Interestingly, Nehalem delivered lower performance on the higher arithmetic intensity (radius=5) kernel. This is due to the reduced efficacy of register blocking at the higher radius size, perhaps due to register pressure. We also observe that the hand-optimized pthreads version delivered performance better than our productive auto-parallelization-alone performance, but lower than our auto-parallelized and auto-tuned results. In all cases, performance scaled well with the number of cores — indicative that bandwidth did not significantly impede performance.

Interestingly, this kernel does not benefit from NUMA optimizations. In addition, the scaling is almost perfect, due to the bottleneck being computation. Auto-tuning yields large improvements by eliminating capacity misses as well as ensuring high inner-loop performance through register blocking. Thus auto-tuning increases performance $2\times$ on the XT4 , $2.23\times$ on the Barcelona and $3.6\times$ on Nehalem for a radius of 3, and $2\times$ on the XT4, Barcelona, and Nehalem for radius 5.

Overall, the bilateral filter kernel at radius 3 is improved relative to the serial version by $4.8\times$ on the XT4, $14.8\times$ for Barcelona and $20.7\times$ for Nehalem; for radius 5, the improvement is $4\times$ for the XT4, $11.6\times$ for Barcelona, and $10.2\times$ for Nehalem.

## 6.5 Comparison to OpenMP

The auto-parallelization scheme specifies a straightforward domain decomposition over threads in the least unit-stride dimension, with no core, thread, or register blocking. To examine the quality of the framework's auto-

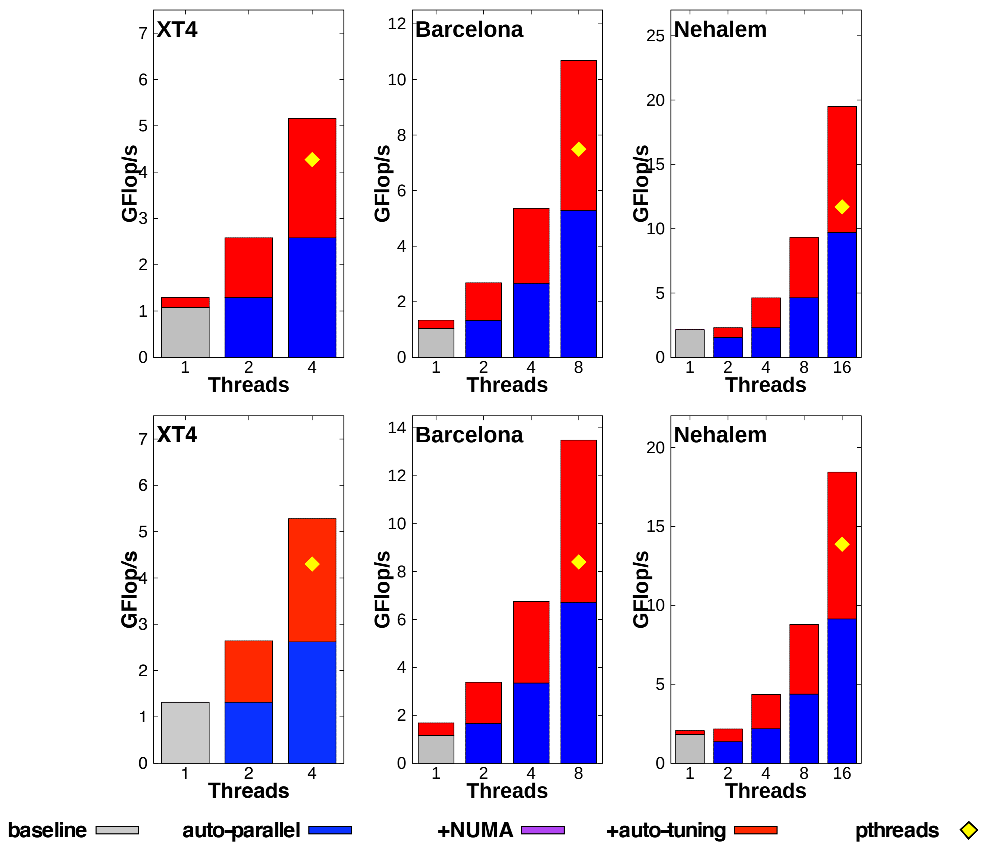

**Figure 5. Bilateral filtering performance as a function of auto-parallelization and auto-tuning, for radius 3 (top) and radius 5 (bottom). For comparison, the yellow diamond shows performance achieved using a handwritten pthreads filtering implementation with NUMA-aware initialization.**

parallelization capabilities, we compare performance with a parallelized version using OpenMP [5] pragmas, which ensures proper NUMA memory decomposition via the first-touch pinning policy. Results, represented as yellow diamonds in Figures 2–4, show that performance is well correlated with our framework's NUMA-aware auto-parallelization. However, we note that for the Laplacian kernel on the XT4, our auto-parallelization technique outperforms OpenMP by 25%. Furthermore, our auto-tuning approach dramatically improves performance over the OpenMP-only versions.

## 6.6 Programmer Productivity Benefits

We now compare our framework's performance in context of programming productivity. Our previous work [3] presented the results of Laplacian kernel optimization using a hand-written auto-tuning code generator, which required months of Perl script implementation, and was inherently limited to a single kernel instantiation. In contrast, utilizing our framework across a broad range of possible stencils only requires a few minutes to annotate a given kernel region, and pass it through our auto-parallelization and auto-tuning infrastructure; thus tremendously improving productively as well as kernel extensibility.

Currently our framework does not implement several hand-tuned optimizations [3], including SIMDization, padding, or the employment of employ cache bypass (*movntpd*). However, comparing results over the same set of optimizations, we find that our framework attains excellent performance that is comparable to the hand-written version. We obtain near identical results the evaluated platforms. Future work will continue incorporating additional optimization schemes into our automated framework.

## 6.7 Architectural Performance Comparison

Figure 6 shows a comparative summary of the fully tuned performance on each architecture for each kernel. For the three differential operator kernels, we observe that the 2P Barcelona is slightly less than twice as fast as fast as the 1P Budapest (XT4). Such a situation on these memory-intensive kernels likely arises from the 66% difference in

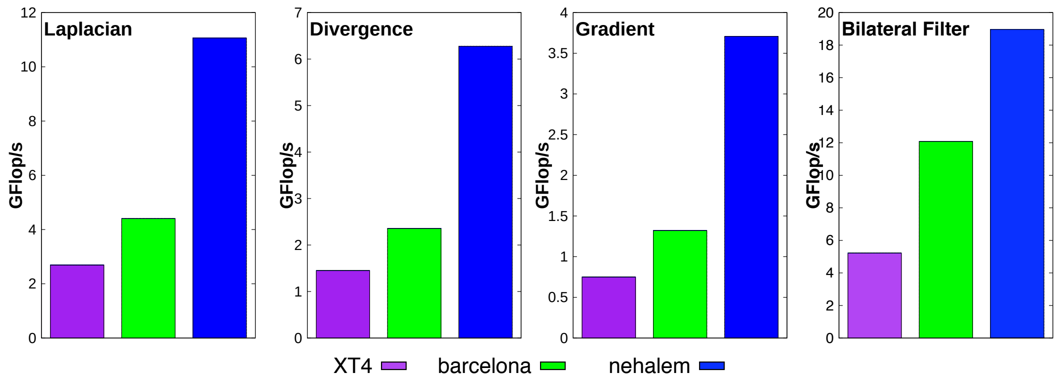

**Figure 6. Performance summary. For the bilateral filter kernel, the chart shows average performance across the two radius values.**

aggregate DRAM bandwidth (12.8GB/s vs. 21.3GB/s). Clearly, Nehalem is substantially faster than either AMD-based system — clearly an advantage of 50% more DRAM channels running at up to 60% higher frequency.

The bilateral filtering kernel shows that the 2P Barcelona can acheive over twice the performance of the XT4 on this computation-bound kernel, even though the computational capabilities are only double. The source of this discrepancy is under investigation. As expected, Nehalem performs quite a bit better than either AMD processor, due to both its higher bandwidth capabilities as well as its computational ability.

## 7 Summary and Conclusions

Modern [erformance programmers are faced with the enormous challenge of productively designing applications that leverage the computational resources of leading multicore designs, while attaining performance portability across the myriad of current and future CMP instantiations. In this work, we introduce a generalized framework for stencil auto-tuning that takes the first steps towards making complex chip multiprocessors accessible to domain-scientists, in a productive and performance portable fashion — demonstrating gains of up to 22× speedup compared with the default serial version.

Overall we make a number of important contributions that include the (*i*) introduction of a high performance, multi-target framework for auto-parallelizing and auto-tuning multidimensional stencil loops; (*ii*) presentation of a novel tool chain based on an abstract syntax tree (AST) for processing, transforming, and generating stencil loops; (*iii*) description of an automated parallelization process for targeting multidimensional stencil codes; (*iv*) achievement of excellent performance on our evaluation suite using three important stencil access patterns as well as a visualization kernel that implements bilateral filtering; and (*v*) demonstration that automated frameworks such as these can enable greater programmer productivity by reducing the need for individual, hand-coded auto-tuners.

The modular architecture of our framework enables it to be extended through the development of additional parser, strategy engine, and code generator modules. Future work will concentrate on extending the scope of optimizations as outlined in Section 6.6, including cache bypass, padding, and prefetching. We also plan to expand our framework to broaden the range of allowable stencil computation classes, including in-place and multigrid methods. Finally, we plan to demonstrate our framework's applicability by investigating its impact on realistic, large-scale scientific applications.

## Acknowledgments

This work was supported by Microsoft (Award #024263), Intel (Award #024894), by matching funding by U.C. Discovery (Award #DIG07-10227), and by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231. All XT4 runs were conducted on the XT4 (Franklin) at NERSC.

## References

[1] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[2] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures. In *SC '08: Proceedings of*

*the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[4] Embeddable Common Lisp. `http://ecls.sourceforge.net/`.

[5] O. A. S. for Parallel Programming. `http://openmp.org`.

[6] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A case for machine learning to optimize multicore performance. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*, March 2009.

[7] GreenFlash. `http://www.lbl.gov/CS/html/greenflash.html`.

[8] R. Heikes and D. Randall. Numerical integration of the shallow-water equations of a twisted icosahedral grid. part i: basic design and results of tests. *Mon. Wea. Rev.*, 123:1862–1880, 1995.

[9] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *ACM SIGPLAN Workshop Memory Systems Performance and Correctness*, San Jose, CA, 2006.

[10] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2001.

[11] LLVM Homepage. `http://llvm.org/`.

[12] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `http://www.cs.virginia.edu/stream/`.

[13] G. Rivera and C. Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of SC'00*, Dallas, TX, November 2000. Supercomputing 2000.

[14] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. *International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

[15] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Sketching stencils. In *Proc. International Conference on Programming Languages Design and Implementation (PLDI)*, June 2007.

[16] C. Tomasi and R. Manduchi. Bilateral Filtering for Gray and Color Images. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, page 839, Washington, DC, USA, 1998. IEEE Computer Society.

[17] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[18] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005, J. of Physics: Conference Series*. Institute of Physics Publishing, June 2005.

[19] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimization of Software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.