An Auto-Tuning Framework for Parallel Multicore Stencil Computations

Shoaib Kamil[†‡], Cy Chan[†⋆], Leonid Oliker[†], John Shalf[†], Samuel Williams[†]
[†]*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

Although stencil auto-tuning has shown tremendous potential in effectively utilizing architectural resources, it has hitherto been limited to single kernel instantiations; in addition, the large variety of kernels used in practice makes this computation pattern difficult to assemble into a library. This work presents a stencil auto-tuning framework that significantly advances programmer productivity by automatically *converting a straightforward Fortran 95 stencil expression to tuned implementations in Fortran, C, or CUDA, thus allowing performance portability across diverse computer architectures, including the AMD Barcelona, Intel Nehalem, Sun Victoria Falls, and the latest NVIDIA GPUs. Results show that our generalized methodology delivers significant performance gains of up to* $22\times$ *speedup over the reference serial implementation. 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.*

1 Introduction

Petascale systems are becoming available to the computational science community with an increasing diversity of architectural models. These high-end systems, like all computing platforms, will increasingly rely on software-controlled on-chip parallelism to manage the trade-offs between performance, energy-efficiency and reliability [1]. This results in a daunting problem for performance-oriented programmers. Scientific progress will be substantially slowed without productive programming models and tools that allow programmers to efficiently utilize massive on-chip concurrency. The challenge of our decade is to create new programming models and tools that enable concise program expression while exposing fine-grained, explicit parallelism to the hardware across a diversity of chip multiprocessors (CMPs). Exotic programming models and domain-specific languages have been proposed to meet this challenge but run counter to the desire to preserve the enormous investment in existing software infrastructure.

This work presents a novel approach for addressing these conflicting requirements for stencil-based computations using a generalized auto-tuning framework. Our framework takes as input a straightforward Fortran 95 stencil expression and automatically generates tuned implementations in Fortran, C, or CUDA, thus providing performance portability across diverse architectures that range from conventional multicore processors to some of the latest graphics processing units (GPUs). This approach enables a viable migration path from existing applications to codes that pro-

vide scalable intra-socket parallelism across a diversity of emerging chip multiprocessors — preserving portability and allowing for productive code design and evolution.

Our work addresses the performance and portability of stencil (nearest-neighbor) computations, a class of algorithms at the heart of many calculations involving structured (rectangular) grids, including both implicit and explicit partial differential equation (PDE) solvers. These solvers constitute a large fraction of scientific applications in such diverse areas as heat diffusion, climate science, electromagnetics, and fluid dynamics. Previous efforts have successfully developed stencil-specific auto-tuners [4, 14, 21], which search over a set of optimizations and their parameters to minimize runtime and provide performance portability across a variety of architectural designs. Unfortunately, the stencil auto-tuning work to date has been limited to static kernel instantiations with pre-specified data structures. As such, they are not suitable for encapsulation into libraries usable by most real-world applications.

The focus of our study is to examine the potential of a generalized stencil auto-parallelization and auto-tuning framework, which can effectively optimize a broad range of stencil computations with varying data structures, dimensionalities, and topologies. Our novel methodology first builds an abstract representation from a straightforward user-provided Fortran stencil problem specification. We then use this intermediate representation to explore numerous auto-tuning transformations. Finally, our infrastructure is capable of generating a variety of shared-memory parallel (SMP) backend code instantiations, allowing optimized performance across a broad range of architectures.

To demonstrate the flexibility of our framework, we examine three stencil computations with a variety of different computational characteristics, arising from the 3D Laplacian, Divergence, and Gradient differential operators. Auto-parallelized and auto-tuned performance is then shown on several leading multicore platforms, including the AMD Barcelona, Sun Victoria Falls, NVIDIA GTX280, and the recently released Intel Nehalem. Results show that our generalized methodology can deliver significant performance gains of up to $22\times$ speedup compared with the reference serial version, while allowing portability across diverse CMP 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 code that required several months of tedious work to produce.

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

2 Related Work

Auto-tuning has been applied to a number of scientific kernels, most successfully to dense and sparse linear algebra. ATLAS [24] is a system that implements BLAS (basic linear algebra subroutines) and some LAPACK [15] kernels using compile-time auto-tuning. Similarly, OSKI [23] applies auto-tuning techniques to sparse linear algebra kernels, using a combination of compile-time and run-time tuning. FFTW [8] is a similar system for producing auto-tuned efficient signal processing kernels.

Unlike the systems above, SPIRAL [19] is a recent auto-tuning framework that implements a compiler for a specific class of kernels, producing high-performance tuned signal processing kernels. While previous auto-tuners relied on simple string manipulation,

SPIRAL's designers defined an algebra suitable for describing a class of kernels, and built a tuning system for that class. Our work's goal is to create a similar system for stencil auto-tuning.

Optimizing stencil calculations have primarily focused on on tiling optimizations [16, 20, 21] that attempt to exploit locality by performing operations on cache-sized blocks of data before moving on to the next block. A study of stencil optimization [3] on (single-core) cache-based platforms found that tiling optimizations were primarily effective when the problem size exceeded the on-chip cache's ability to exploit temporal recurrences. Previous work in stencil auto-tuning for multicore and GPUs [3, 4] demonstrated the potential for greatly speeding up stencil kernels, but concentrated on a single kernel (the 7-pt Laplacian in 3D). Because the tuning system was hand-coded for that particular kernel, it cannot easily be ported to other stencils instantiations. In addition, it does not allow easy composition of different optimizations, or the integration of different search strategies such as hill-climbing.

Compiler optimizations for stencils concentrate on the polyhedral model [2] for improving performance by altering traversals to minimize (modeled) memory overhead. Auto-parallelization of stencil kernels is also possible using the polyhedral model [6]. Future work will combine/compare the polyhedral model with our auto-tuning system to explore the tradeoffs of simple models and comprehensive auto-tuning. Some planned optimizations (particularly those that alter the data structures of the grid) are currently not handled by the polyhedral model.

The goal of our framework is not just to automatically generate parallel stencil codes and tune the associated parallelization parameters, but also to tune the parallelization parameters in tandem with lower-level serial optimizations. Doing so will find the globally optimal combination for a given parallel machine.

Previous work also includes the ParAgent [18] tool, which uses static analysis to help minimize communication between compute nodes in a distributed memory system. In contrast, our framework addresses both locality and parallelization parameters on a shared memory system. Like our framework, the PLuTo system [6] strives to simultaneously optimize parameters for both data locality and parallelization. Their approach determines a parameterization through the minimization of a unified cost function that incorporates aspects of both intra-tile locality and inter-tile communication. Our work differs from these methods primarily in that we leverage auto-tuning to find optimal solutions, which may provide better results in cases where machine characteristics are difficult to model effectively using static analysis.

This work presents a novel advancement in auto-tuning stencil kernels by building a framework that incorporates experience gained from building kernel-specific tuners. In particular, the framework has the applicability of a general domain-specific compiler like SPIRAL, while supporting multiple backend architectures. In addition, modularity allows the framework to, in the future, support data structure transformations and additional front- and backends using a simple plugin architecture. A preliminary overview of our methodology was presented at a recent Cray User's Group Workshop [13]; our current work extends this framework for a wider spectrum of optimizations and architectures including Victoria Falls and GPUs, as well as incorporating performance model expectations and analysis.

3 Stencil & Architectures

Stencil computations on regular grids are at the core of a wide range of scientific codes. These applications are often implemented using iterative finite-difference techniques that

Stencil	Cache Refs. (doubles)	Flops per Stencil	Compulsory Read Traffic	Writeback Traffic	Write Allocate Traffic	Capacity Miss Traffic	Naïve Arith. Intensity	Tuned Arith. Intensity	Expected Auto-tuning Benefit
Laplacian	8	8	8 Bytes	8 Bytes	8 Bytes	16 Bytes	0.20	0.33	1.66×
Divergence	7	8	24 Bytes	8 Bytes	8 Bytes	16 Bytes	0.14	0.20	1.40×
Gradient	9	6	8 Bytes	24 Bytes	24 Bytes	16 Bytes	0.08	0.11	1.28×

Table 1. Average stencil characteristics. Arithmetic Intensity is defined as the Total Flops / Total bytes. Capacity misses represent a reasonable estimate for cache-based superscalar processors. Auto-tuning benefit is a reasonable estimate based on the improvement in arithmetic intensity assuming a memory bound kernel without conflict misses.

sweep over a spatial grid, performing nearest neighbor computations called *stencils*. In a stencil operation, each point in a multidimensional grid is updated with weighted contributions from a subset of its neighbors within a fixed distance in both time and space, locally solving a discretized version of the PDE for that data element. These operations are then used to build solvers that range from simple Jacobi iterations to complex multigrid and adaptive mesh refinement methods.

Stencil calculations perform repeated sweeps through data structures that are typically much larger than the data caches of modern microprocessors. As a result, these computations generally produce a high memory traffic for relatively little computation, causing performance to be bound by memory throughput rather than floating-point operations. Reorganizing these stencil calculations to take full advantage of memory hierarchies has therefore been the subject of much investigation over the years.

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 and accommodates new architecture-specific languages such as CUDA. Implementation of these kernels using existing languages and compilers destroys domain-specific knowledge. As such, compilers have difficulty proving that code transformations are safe, and even more difficulty transforming data layout in memory. The framework side-steps the complex task of analysis and presents a simple, uniform, and familiar interface for expressing stencil kernels as a conventional Fortran expression — while presenting a proof-of-concept for other potential classes of domain-specific generalized auto-tuners.

3.1 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 ($u \leftarrow \nabla \cdot \mathbf{F}$) and Gradient ($\mathbf{F} \leftarrow \nabla u$) differential operators. Details of these kernels are shown in Figure 1 and Table 1. All three operators are implemented using central-difference on a 3D rectahedral block-structured grid via Jacobi's method (out-of-place), and benchmarked on

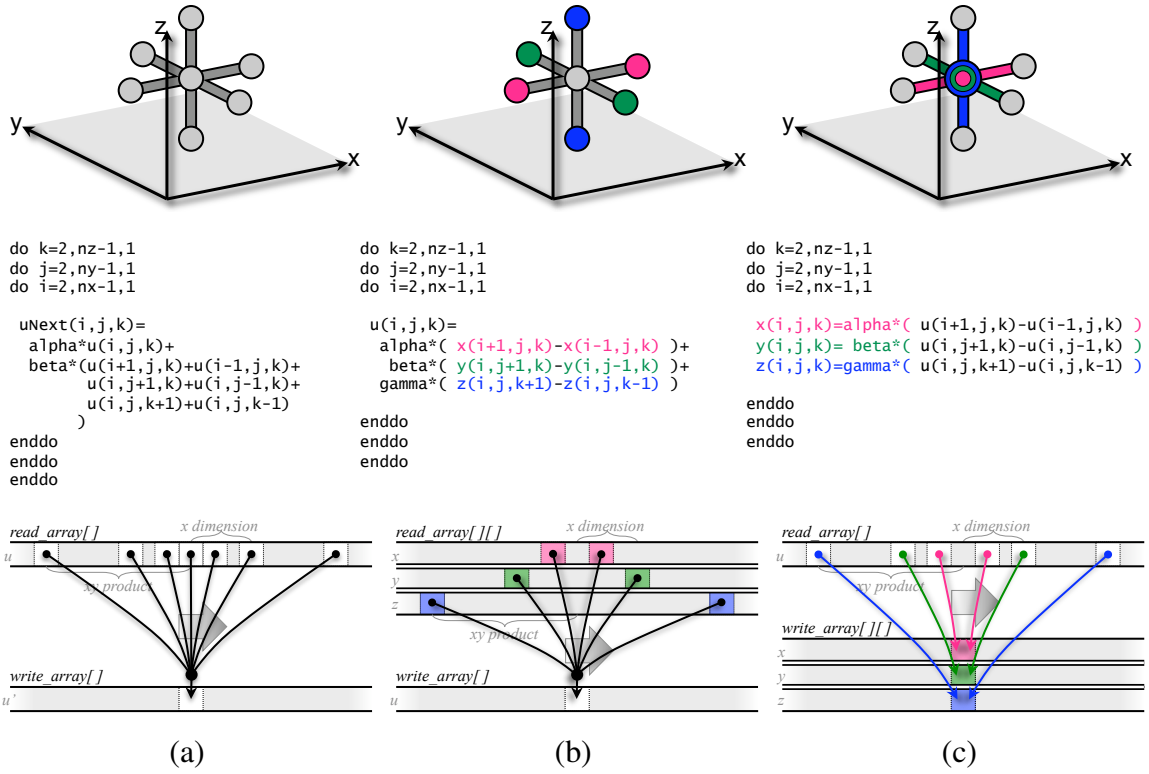

Figure 1. (a) Laplacian, (b) Divergence, and (c) Gradient stencils. Top: 3D visualization of the nearest neighbor stencil operator. Middle: code as passed to the parser. Bottom: memory access pattern as the stencil sweeps from left to right. Note: the color represents cartesian component of the vector fields (scalar fields are gray).

a 256^3 grid. Note that although the code generator has no restrictions on data structure, for brevity, we only explore the use of structure of arrays for vector fields. As described below, these kernels 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 — defined as the ratio of arithmetic operations to memory traffic — and the system stream bandwidth.

Table 1 presents the characteristics of the three stencil operators and sets performance expectations. Similar to the 3C's cache model [12], we break memory traffic into compulsory read, write back, write allocate, and capacity misses. A naïve implementation will produce memory traffic equal to the sum of these components, and will therefore result in the shown arithmetic intensity ($\frac{totalflops}{totalbytes}$), ranging from 0.20–0.08. The auto-tuning effort explored in this work attempts to improve performance by eliminating capacity misses; thus it is possible to bound the resultant arithmetic intensity based only on compulsory read, write back, and write allocate memory traffic. For the three examined kernels, capacity misses account for dramatically different fractions of the total memory traffic. Thus, we can also bound the resultant potential performance boost from auto-tuning per kernel — 1.66×, 1.40×, and 1.28× for the Laplacian, Divergence, and Gradient respectively. Moreover, note that the kernel's auto-tuned arithmetic intensity will vary substantially from each other, ranging from 0.33–0.11. As such, performance is expected to vary proportionally, as predicted in by the Roofline model [25].

Core Architecture	AMD Barcelona	Intel Nehalem	Sun Niagara2	NVIDIA GT200 SM
Type	superscalar out of order	superscalar out of order	HW multithread dual issue	HW multithread SIMD
Clock (GHz)	2.30	2.66	1.16	1.3
DP GFlop/s	9.2	10.7	1.16	2.6
Local-Store	—	—	—	16KB**
L1 Data Cache	64KB	32KB	8KB	—
private L2 cache	512KB	256KB	—	—

System	Opteron 2356 (Barcelona)	Xeon X5550 (Gainestown)	UltraSparc T5140 (Victoria Falls)	GeForce GTX280
# Sockets	2	2	2	1
Cores per Socket	4	4	8	30
Threads per Socket‡	4	8	64	240
primary memory parallelism paradigm	HW prefetch	HW prefetch	Multithreading	Multithreading with coalescing
DRAM Capacity	2×2MB 16GB	2×8MB 12GB	2×4MB 32GB	1GB (device) 4GB (host)
DRAM Pin Bandwidth (GB/s)	21.33	51.2	42.66(read) 21.33(write)	141 (device) 4 (PCIe)
DP GFlop/s	73.6	85.3	18.7	78
DP Flop:Byte Ratio	3.45	1.66	0.29	0.55
Threading	Pthreads	Pthreads	Pthreads	CUDA 2.0
Compiler	gcc 4.1.2	gcc 4.3.2	gcc 4.2.0	nvcc 0.2.1221

Table 2. Architectural summary of evaluated platforms. †Each of 2 thread groups may issue up to 1 instruction. ‡A *CUDA thread block* is considered 1 thread. **16 KB local-store shared by all concurrent *CUDA thread blocks* on the SM.

3.2 Experimental Platforms

To evaluate our stencil auto-tuning framework, we examine a broad range of leading multicore designs: AMD Barcelona, Intel Nehalem, Sun Victoria Falls, and NVIDIA GTX 280. A summary of key architectural features of the evaluated systems appears in Table 2; space limitations restrict detailed descriptions of the systems. Observe, as all architectures have Flop:DRAM byte ratios significantly greater than the arithmetic intensities mentioned in Section 3.1, we expect all architectures to be memory bound. Note that the sustained system power data was obtained using an in-line digital power meter while the node was under a full computational load, while chip and GPU card power is based on the maximum Thermal Design Power (TDP), extrapolated from manufacturer's datasheets. Although the node architectures are diverse, most accurately represent building-blocks of current and future ultra-scale supercomputing systems.

4 Auto-tuning Framework

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 [24] or OSKI [23] to implement the many different stencil kernels is infeasible.

This work presents a proof-of-concept of a

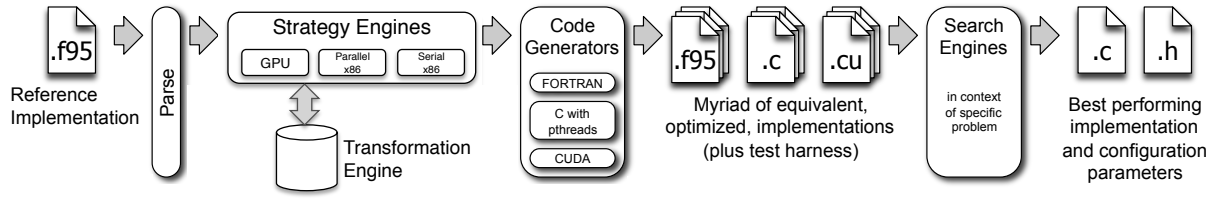

Figure 2. Stencil auto-tuning framework flow. Readable domain-specific code is parsed into an abstract representation, transformations are applied, code is generated using specific target backends, and the optimal auto-tuned implementation is determined via search.

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. The overall flow through the auto-tuning system is shown in Figure 2.

4.1 Front-End 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.

4.2 Stencil Kernel Breadth

Currently, the auto-tuning system handles a specific class of stencil kernels of certain dimensionality and code structure. In particular, the auto-tuning system assumes a 2D or 3D rectahedral grid, and a stencil based on arithmetic operations and table lookups (array accesses). Future work will further extend the generality to allow grids of arbitrary dimensionality. Although this proof-of-concept framework does auto-tune serial kernels with imperfect loop nests, the parallel tuning relies on perfect nesting in order to determine legal domain decompositions and NUMA (non-uniform memory access) page mapping initialization — future framework extensions will incorporate imperfectly nested loops. Additionally, we currently treat boundary calculations as a separate stencil, although future versions may integrate stencils with overlapping traversals into a single stencil. Overall, our auto-tuning system can target and accelerate a large group of stencil kernels currently in use, while active research continues to extend the generality of the framework.

5 Optimization & Codegen

The heart of the auto-tuning framework is the transformation engine and the back-end code generation for both serial and parallel implementations. The transformation engine is in many respects similar to a source-to-source translator, but it exploits domain-specific knowledge of the problem space to implement transformations that would otherwise

be difficult to implement as a fully generalized loop optimization within a conventional compiler. 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 nodes as well as CUDA versions specifically for the massively parallel NVIDIA GPGPUs.

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 [5], making it simple to interface with the parsing front-ends (written in Flex and YACC) and preserving portability across a wide variety of architectures. Potential future alternatives include implemention of affine scaling transformations or more complex AST representations, such as the one used by LLVM [17], or more sophisticated transformation engines such as the one provided by the Sketch [22] compiler.

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 5.2.

5.1 Serial Optimizations

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 [4]:better utilization of SIMD instructions and common subexpression elimination (to improve arithmetic efficiency), cache bypass (to eliminate cache fills), and explicit software prefetching. Additionally, future work will support aggressive memory and code structure transformations.

We also note that, although the current set of optimizations may seem identical to existing compiler optimizations, future strategies such as memory structure transformations will be beyond the scope of compilers, since such optimizations are specific to stencil-based computations. Additionally, the fact that our framework's transformations yield code that outperforms compiler-only optimized versions shows compiler algorithms cannot always prove that these (safe) optimizations are allowed. Thus, a domain-specific code generator run by the user has the freedom to implement transformations that a compiler may not.

5.2 Parallelization & Code Generation

Given the stencil transformation framework, we now present parallelization optimizations, as well as cache- and GPU-specific optimizations. The shared-memory parallel code generators leverage the serial code generation routines to produce the version run by each individual thread. Because the parallelization mechanisms are specific to each architecture, both the strategy engines and code generators must be tailored to the desired targets. For the cache-based systems (Intel, AMD, Sun) we chose `pthreads` for lightweight parallelization; on the NVIDIA GPU, the only parallelization option is *CUDA thread blocks* that execute in a SPMD (single program multiple data) fashion.

Since the parallelization strategy influences code structure, the AST — which represents code run on each individual thread — must be modified to reflect the chosen parallelization strategy. The parallel code generators make the necessary modifications to the AST before passing it to the serial code generator.

5.2.1 Multicore-specific Optimizations and Code Generation

Following the effective blocking strategy presented in previous studies [4], we decompose the problem space into *core blocks*, as shown in Figure 3. The size of these core blocks can be tuned to avoid capacity misses in the last level cache. Each core block is further divided into *thread blocks* such that threads sharing a common cache can cooperate on a core block. Though our code generator is capable of using variable-sized 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 core block size, thread block size, chunk size, and register block size are all tunable by the framework.

The code generator creates a new set of loops for each thread to iterate over its assigned set of thread blocks. Register blocking is accomplished through strip mining and loop unrolling via the serial code generator.

NUMA-aware memory allocation is implemented by pinning threads to the hardware and taking advantage of first-touch page mapping policy during data initialization. The code generator analyzes the decomposition and has the appropriate processor touch the memory during initialization.

5.2.2 CUDA-specific Optimizations and Code Generation

CUDA programming is oriented around *CUDA thread blocks*, which differ from the thread blocks used in the previous section. CUDA thread blocks are vector elements mapped to the scalar cores (lanes) of a streaming multiprocessor. The vector conceptualization facilitates debugging of performance issues on GPUs. Moreover, CUDA thread blocks are analogous to threads running SIMD code on superscalar processors. Thus, parallelization on the GTX280 is a straightforward SPMD domain decomposition among CUDA thread blocks; within each CUDA thread block, work is parallelized in a SIMD manner.

To effectively exploit cache-based systems, code optimizations attempt to employ unit-stride memory access patterns and maintain small cache working sets through cache blocking — thereby leveraging spatial and temporal locality. In contrast, the GPGPU model forces programmers to write a program for each *CUDA thread*. Thus, spatial locality may only be achieved by ensuring that memory accesses of adjacent threads (in a CUDA thread block) reference contiguous segments of memory to exploit hardware coalescing. Consequently, our GPU implementation ensures spatial locality for each stencil point by tasking adjacent threads of a CUDA thread block to perform stencil operations on adjacent grid locations. Some performance will be lost as not all coalesced memory references are aligned to 128-byte boundaries.

The CUDA code generator is capable of exploring the myriad different ways of dividing the problem among CUDA thread blocks, as well as tuning both the number of threads in a CUDA thread block and the access pattern of the threads. For example, in a single time step, a CUDA thread block of 256 CUDA threads may access a tile of 32 x 4 x 2 contiguous data elements; the thread block would then iterate

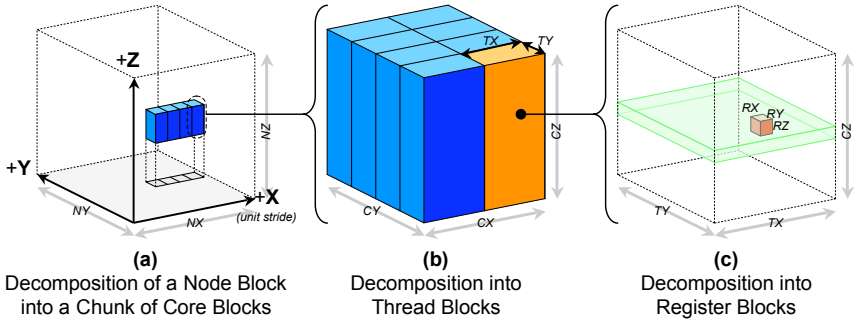

Figure 3. Four-level problem decomposition: In (a), a *node block* (the full grid) is broken into smaller *chunks*. All *core blocks* in a chunk are processed by the same subset of threads. One core block from the chunk in (a) is magnified in (b). A single *thread block* from the core block in (b) is then 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.

this tile shape over its assigned core block. In many ways, this exploration is analogous to register blocking within each core block on cache-based architectures.

Our code generator currently only supports the use of global device memory, and so does not take advantage of the low-latency local-store style shared memory present on the GPU. As such, the generated code does not take advantage of temporal locality of the memory accesses. Future work will incorporate support for exploitation of CUDA shared memory.

6 Auto-Tuning Strategy Engine

In this section, we describe how the auto-tuner searches the enormous parameter space of serial and parallel optimizations described in previous sections. Because the combined parameter space of the preceding optimizations is so large, it is clearly infeasible to try all possible strategies. In order to reduce the number of code instantiations the auto-tuner must compile and evaluate, we used strategy engines to enumerate an appropriate subset of the parameter space for each platform.

The strategy engines enumerate only those parameter combinations (strategies) in the sub-region of the full search space that best utilize the underlying architecture. For example, cache blocking in the unit stride dimension could be practical on the Victoria Falls architecture, while on Barcelona or Nehalem, the presence of hardware prefetchers makes such a transformation non-beneficial [3].

Further, the strategy engines keep track of parameter interactions to ensure that only legal strategies are enumerated. For example, since the parallel decomposition changes the size and shape of the data block assigned to each thread, the space of legal serial optimization parameters dependends on the values of the parallel parameters. The strategy engines ensure all such constraints (in addition to other hardware restrictions such as maximum number of threads per processor) are satisfied during enumeration.

For each parameter combination enumerated by the strategy engine, the auto-tuner's search engine then directs the parallel and serial code generator components to produce the code instantiation corresponding to that strategy. The auto-tuner runs each instantiation and records the time taken on the target machine. After all enumerated strategies have been timed, the fastest parameter combination is reported to the user, who can then link the optimized version of the stencil into their existing code.

Table 3 in the Appendix shows the attempted optimizations and the associated parameter subspace explored by the strategy engines corresponding to each of our testedplatforms. While the search engine currently does a comprehensive search over the parameter subspace dictated by the strategy engine, future work will include more intelligent search mechanisms such as hill-climbing or machine learning techniques [9], where the search engine can use timing feedback to dynamically direct the search.

7 Performance Evaluation

In this section, we examine the performance quality and expectations of our auto-parallelizing and auto-tuning framework across the four evaluated architectural platforms. The impact of our framework on each of the three kernels is compared in Figure 4, showing performance of: the original serial kernel (gray), auto-parallelization (blue), auto-parallelization with NUMA-aware initialization (purple), and auto-tuning (red). The GTX280 reference performance (blue) is based on a straightforward implementation that maximizes CUDA thread parallelism. We do not consider the impact of host transfer overhead; previous work [4] examined this potentially significant bottleneck in detail. Overall, results are ordered such that threads first exploit multithreading within a core, then multiple cores on a socket, and finally multiple sockets. Thus, on Nehalem, the two thread case represents one fully-packed core; similarly, the GTX280 requires at least 30 *CUDA thread blocks* to utilize the 30 cores (streaming multiprocessors).

7.1 Auto-Parallelization Performance

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-parallelization capabilities, we compare performance with a parallelized version using OpenMP [7], which ensures proper NUMA memory decomposition via first-touch pinning policy. Results, shown as yellow diamonds in Figure 4, show that performance is well correlated with our framework's NUMA-aware auto-parallelization. Furthermore, our approach slightly improves Barcelona's performance, while Nehalem and Victoria Falls see up to a 17% and 25% speedup (respectively) compared to the OpenMP version, indicating the effectiveness of our auto-parallelization methodology even before auto-tuning.

7.2 Performance Expectations

When tuning any application, it is important to know when you have reached the architectural peak performance, and have little to gain from continued optimization. We make use of a simple empirical performance model to establish this point of diminishing returns and use it to evaluate how close our automated approach can come to the machine limits. We now examine achieved performance in the context of this simple model based on the hardware's characteristics. Assuming all kernels are memory bound and do not suffer from an abundance of capacity misses, we approximate the performance bound as the product of streaming bandwidth and each stencil's arithmetic intensity (0.33, 0.20 and 0.11 — as shown in Table 1). Using an optimized version of the OpenMP Stream benchmark, which we modified to reflect the number of read and write streams for each kernel, we obtain expected peak performance based on memory bandwidth for the CPUs. For the GPU, we use two versions of Stream: one that consists of exclusively read traffic, and another that is half read and half write.

Our model's expected performance range is represented as a green line (for the CPUs) and a green region (for the GPUs) in Figure 4. For

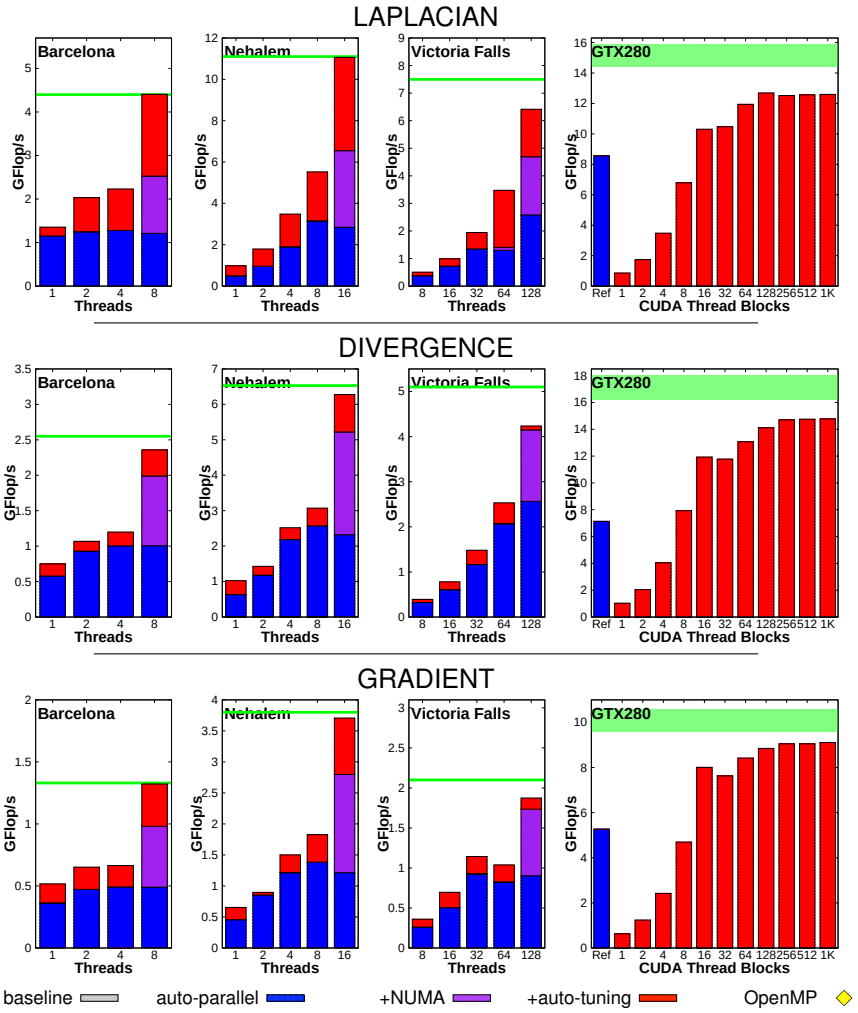

Figure 4. Laplacian (top row), Divergence (middle row), and Gradient (bottom row) performance as a function of auto-parallelization and auto-tuning — on the four evaluated platforms. Note: the green region marks the range in performance extrapolated from Stream bandwidth. For comparison, the yellow diamond shows performance achieved using the original stencil kernel with OpenMP pragmas and NUMA-aware initialization.

Barcelona and Nehalem, our optimized kernels obtain performance essentially equivalent to peak memory bandwidth. For Victoria Falls, the obtained bandwidth is around 20% less than peak for each of the kernels, because our framework does not currently implement software prefetching and array padding, which are critical for performance on this architecture. Finally, the GTX280 results were also below our performance model bound, likely due to no array padding [4]. Overall, our fully tuned performance closely matches our model's expectations, while highlighting areas which could benefit from additional optimizations.

7.3 Performance Portability

The auto-tuning framework takes a serial specification of the stencil kernel and achieve a substantial performance improvement, due to both auto-parallelization and auto-tuning. Overall, Barcelona and Nehalem see between $1.7\times$ to $4\times$ improvement for both the one and two socket cases over the conventional parallelized case, and up to 10 times improvement over the serial code. The results also show that

auto-tuning is essential on Victoria Falls, enabling much better scalability and increasing performance by $2.5\times$ and $1.4\times$ on 64 and 128 threads respectively in comparison to the conventional parallelized case, but a full $22\times$ improvement over an unparallelized example. Finally, auto-tuning on the GTX280 boosted performance by $1.5\times$ to $2\times$ across the full range of kernels — a substantial improvement over the baseline CUDA code, which is implicitly parallel. This clearly demonstrates the performance portability of this framework across the sample kernels.

Overall, we achieve substantial performance improvements across a diversity of architectures – from GPU's to multi-socket multicore x86 systems. The auto-tuner is able to achieve results that are extremely close to the architectural peak performance of the system, which is limited ultimately by memory bandwidth. This level of performance portability using a common specification of kernel requirements is unprecedented for stencil codes, and speaks to the robustness of the generalized framework.

7.4 Programmer Productivity Benefits

We now compare our framework's performance in context of programming productivity. Our previous work [4] 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 productivity as well as kernel extensibility.

Currently our framework does not implement several hand-tuned optimizations [4], 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 on the Barcelona and even higher results on the Victoria Falls platform (6 GFlop/s versus 5.3 GFlop/s). A significant disparity is seen on the GTX280, where previous hand-tuned Laplacian results attained 36 GFlop/s, compared with our framework's 13 GFlop/s. For the CUDA implementations, our automated version only utilizes optimizations and code structures applicable to general stencils, while the hand-tuned version explicitly discovered and exploit the temporal locality specific to the Laplacian kernel — thus maximizing performance, but limiting the method's applicability. Future work will continue incorporating additional optimization schemes into our automated framework.

7.5 Architectural Comparison

Figure 5 shows a comparative summary of the fully tuned performance on each architecture. The GTX280 consistently attains the highest performance, due to its massive parallelism at high clock rates, but transfer times from system DRAM to board memory through the PCI Express bus are not included and could significantly impact performance [4]. The recently-released Intel Nehalem system offers a substantial improvement over the previous generation Intel Clovertown by eliminating the front-side bus in favor of on-chip memory controllers. The Nehalem obtains the best overall performance of the cache-based systems, due to the combination of high memory bandwidth per socket and hardware multithreading to fully utilize the available bandwidth. Additionally, Victoria Falls obtains high performance, especially given its low clock speed, thanks to massive parallelism combined with an aggregate memory bandwidth of 64 GB/s.

Power efficiency, measured in (average stencil) MFlop/s/Watt, is also shown in Figure 5. For the GTX280 measurements we show the

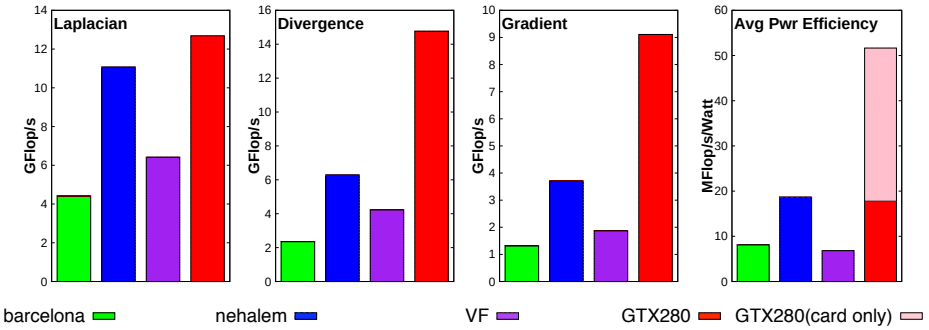

Figure 5. Peak performance and power efficiency after auto-tuning and parallelization. GTX280 power efficiency is shown based on system power as well as the card alone.

power efficiencies both with (red) and without the host system (pink). The GTX280 shows impressive gains over the cache-based architecture if considered as a standalone device, but if system power is included, the GTX280's advantage is diminished and the Nehalem becomes the most power efficient architecture evaluated in this study.

8 Summary and Conclusions

Performance programmers are faced with the enormous challenge of productively designing applications that effectively leverage the computational resources of leading multicore designs, while allowing for performance portability across the myriad of current and future CMP instantiations. In this work, we introduce a fully generalized framework for stencil auto-tuning that takes the first steps towards making complex chip multiprocessors and GPUs 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 multi-dimensional 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 on both cache-based multicore architectures as well as GPGPUs; (*iv*) achievement of excellent performance on our evaluation suite using three important stencil access patterns; (*v*) utilization of simple performance model that effectively predicts the expected performance range for a given kernel and architecture; and (*vi*) 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 (see Section 7.4), including cache bypass, padding, and prefetching. Additionally, we plan to extend the CUDA backend for a general local store implementation, thus leveraging temporal locality to improve performance and allowing extensibility to other local-store architectures such as the Cell processor. We also plan to expand our framework to broaden the range of allowable stencil computation classes (see Section 4.2), including in-place and multigrid methods. Finally, we plan to demonstrate our framework's applicability by investigating its impact on large-scale scientific applications, including a forthcoming optimization study of an icosahedral atmospheric climate simulation [10, 11].

References

[1] K. Asanovic, R. Bodik, B. Catanzaro, et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS, University of California, Berkeley, 2006.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04:Parallel Architectures and Compilation Techniques*, Washington, DC, 2004.

[3] 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.

[4] K. Datta, M. Murphy, V. Volkov, et al. Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures. In *Proceedings of SC '08*, Austin, Texas, 2008.

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

[6] B. et al. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, 2008.

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

[8] M. Frigo. A fast fourier transform compiler. pages 169–180, 1999.

[9] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A case for machine learning to optimize multicore performance. In *Workshop on Hot Topics in Parallelism*, March 2009.

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

[11] 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.

[12] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[13] S. Kamil, C. Chan, S. Williams, et al. A generalized framework for auto-tuning stencil computations. In *Cray User Group*, 2009.

[14] S. Kamil, K. Datta, S. Williams, et al. Implicit and explicit optimizations for stencil computations. In *Workshop Memory Systems Performance and Correctness*, San Jose, CA, 2006.

[15] LAPACK: Linear Algebra PACKage. `http://www.netlib.org/lapack/`.

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

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

[18] S. Mitra, S. C. Kothari, J. Cho, and A. Krishnaswamy. *ParAgent: A Domain-Specific Semi-automatic Parallelization Tool*, pages 141–148. Springer, 2000.

[19] M. Püschel, J. Moura, J. Johnson, et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

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

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

[22] A. Solar-Lezama, G. Arnold, Liviu, et al. Sketching stencils. In *International Conference on Programming Languages Design and Implementation (PLDI)*, June 2007.

[23] 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*, June 2005.

[24] 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.

[25] S. Williams, A. Watterman, and D. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the ACM*, April 2009.

Appendix

Category	Optimization Parameter	Name	Parameter Tuning Range by Architecture: Barcelona/Nehalem	Victoria Falls	GTX280
Data Allocation	NUMA Aware		✓	✓	N/A
Domain Decomposition	Core Block Size	*CX*	NX	{8...NX}	{16†..NX}
		CY	{8...NY}	{8...NY}	{16†..NY}
		CZ	{128...NZ}	{128...NZ}	{16†..NZ}
	Thread Block Size	*TX*	CX	CX	$\{1..\frac{CX}{16}\}^{\ddagger}$
		TY	CY	CY	$\{\frac{CY}{16}..CY\}^{\ddagger}$
		TZ	CZ	CZ	$\{\frac{CZ}{16}..CZ\}^{\ddagger}$
	Chunk Size		$\{1...\frac{NX \times NY \times NZ}{CX \times CY \times CZ \times NThreads}\}$		N/A
Low Level	Array Indexing		✓	✓	✓
	Register Block Size	*RX*	{1...8}	{1...8}	1
		RY	{1...2}	{1...2}	1*
		RZ	{1...2}	{1...2}	1*

Table 3. 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. † Actual values for minimum core block dimensions for GTX280 dependent on problem size. ‡ Thread block size constrained by a maximum of 256 threads in a CUDA thread block with at least 16 threads coalescing memory accesses in the unit-stride dimension. *The CUDA code generator is capable of register blocking the Y and Z dimensions, but due to a confirmed bug in the NVIDIA nvcc compiler, register blocking was not explored in our auto-tuned results.