Lawrence Berkeley National Laboratory
LBL Publications

Title

Auto-Tuning the 27-point Stencil for Multicore

Permalink

https://escholarship.org/uc/item/4wb5w6xq

Authors

Williams, SW
Datta, K
Volkov, V
et al.

Publication Date

2009-10-01

eScholarship.org Powered by the California Digital Library
University of California

Auto-tuning the 27-point Stencil for Multicore

Kaushik Datta[2], Samuel Williams[1], Vasily Volkov[2], Jonathan Carter[1], Leonid Oliker[1], John Shalf[1], and Katherine Yelick[1]

[1] CRD/NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
[2] Computer Science Division, University of California at Berkeley, Berkeley, CA 94720, USA

Abstract. This study focuses on the key numerical technique of stencil computations, used in many different scientific disciplines, and illustrates how auto-tuning can be used to produce very efficient implementations across a diverse set of current multicore architectures.

1 Introduction

The recent transformation from an environment where gains in computational performance came from increasing clock frequency to an environment where gains are realized through ever increasing numbers of modest-performing cores has profoundly changed the landscape of scientific application programming. A major problem facing application programmers is the diversity of multicore architectures that are now emerging. From relatively complex out-of-order CPUs with complex cache structures to relatively simple cores that support hardware multithreading, designing optimal code for these different platforms represents a serious challenge. An emerging solution to this problem is auto-tuning: the automatic generation of many versions of a code kernel that incorporate various tuning strategies, and the benchmarking of these to select the best performing version. Often a key parameter is associated with each tuning strategy (e.g. the amount of loop unrolling or the cache blocking factor), so these parameters must be explored in addition to the layering of the basic strategies themselves.

In Section 2, we give an overview of the stencil studied, followed by a review of the multicore architectures that form our testbed in Section 3. Then, in Sections 4, 5 and 6, we discuss the characteristics of the 27-point stencil, our applied optimizations, and the parameter search respectively. Finally, we present performance results and conclusions in Sections 7 and 8.

2 Stencil Overview

Partial differential equation (PDE) solvers are employed by a large fraction of scientific applications in such diverse areas as heat diffusion, electromagnetics, and fluid dynamics. These applications are often implemented using iterative finite-difference techniques that sweep over a spatial grid, performing nearest neighbor computations called *stencils*. In a stencil operation, each point in a

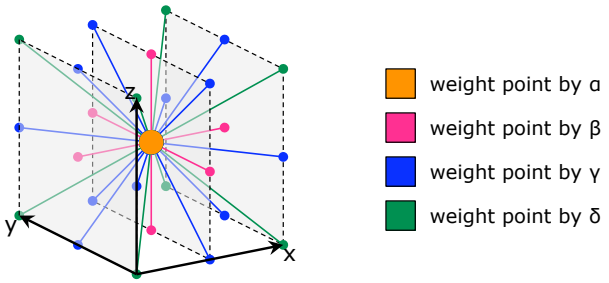


Fig. 1. Visualization of the 27-point stencil used in this work. Note: color represents the weighting factor for each point in the linear combination stencils.

multidimensional grid is updated with weighted contributions from a subset of its neighbors in both time and space — thereby representing the coefficients 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 [2].

Stencil calculations perform global sweeps through data structures that are typically much larger than the capacity of the available data caches. In addition, the amount of data reuse within a sweep is limited to the number of points in a stencil — often less than 27. As a result, these computations generally achieve a low fraction of processor peak performance as these kernels are typically bandwidth-limited. By no means does this imply there is no potential for optimization. In fact, reorganizing these stencil calculations to take full advantage of memory hierarchies has been the subject of much investigation over the years. These have principally focused on tiling optimizations [9–11] that attempt to exploit locality by performing operations on cache-sized blocks of data before moving on to the next block — a means of eliminating capacity misses. A study of stencil optimization [6] 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. A more recent study of lattice-Boltzmann methods [14] employed auto-tuners to explore a variety of effective strategies for refactoring lattice-based problems for multicore processing platforms. That study expanded on prior work by utilizing a more compute-intensive stencil, identifying the TLB as the performance bottleneck, developing new optimization techniques and applying them to a broader selection of processing platforms.

In this paper, we build on our prior work [4] and explore the optimizations and evaluate the performance of each sweep of a Jacobi (out-of-place) iterative method using a 3D 27-point stencil. As shown in Figure 1, the stencil includes all the points within a 3×3×3 cube surrounding the center grid point. Being symmetric, it only uses four different weights for these points– one each for the center point, the six face points, the twelve edge points, and the eight corner points.

Since we are performing a Jacobi iteration, we keep two separate double-precision (DP) 3D arrays — one that is exclusively read from and a second that is only written to. This means that the stencil computation at each grid point is *independent* of every other point. As a result, there are no dependencies between these calculations, and they can be computed in any order. We take advantage of this fact in our code.

In general, Jacobi iterations converge more slowly and require more memory than Gauss-Seidel (in-place) iterations, which only require a single array that is both read from and written to. However, the dependencies involved in Gauss-Seidel stencil sweeps significantly complicate performance optimization. This topic, while important, will be left as future research.

Although the simpler 7-point 3D stencil is fairly common, there are many instances where larger stencils with more neighboring points are required. For instance, the NAS Parallel MG (Multigrid) benchmark utilizes a 27-point stencil to calculate the Laplace operator for a finite volume method [1]. Broadly speaking, the 27-point 3D stencil can be a good proxy for many compute-intensive stencil kernels, which is why it was chosen for our study. For example, consider T. Kim's work on optimizing a fluid simulation code [8]. By using a Mehrstellen scheme [3] to generate a 19-point stencil (where δ equals 0) instead of the typical 7-point stencil, he was able to reach the desired error reduction in 34% fewer stencil iterations. For this study, we do not go into any of the numerical properties of stencils; we merely study and optimize their performance across different multicore architectures. As an added benefit, this analysis also helps to expose many interesting features of current multicore architectures.

3 Experimental Testbed

Table 1 details the core, socket, system and programming of the four cache-based computers used in this work. These include two generations of Intel quad-core superscalar processors (Clovertown and Nehalem) representing similar core architectures but dramatically different integration approaches. Nehalem has replaced Clovertown's front side bus (FSB)-external memory controller hub (MCH) architecture with integrated memory controllers and quick-path for remote socket communication and coherency. At the other end of the spectrum is IBM's Blue Gene/P (BGP) quad-core processor. BGP is a single-socket, dual-issue in-order architecture providing substantially lower throughput and bandwidth while dramatically reducing power. Finally, we included Sun's dual-socket, 8-core Niagara architecture (Victoria Falls). Although its peak bandwidth is similar to Clovertown and Nehalem, its peak flop rate is more inline with BGP. Rather than depending on superscalar execution or hardware prefetchers, each of the eight strictly in-order cores supports two groups of four hardware thread contexts (referred to as Chip MultiThreading or CMT) — providing a total of 64 simultaneous hardware threads per socket. The CMT approach is designed to tolerate instruction, cache, and DRAM latency through fine-grained multi-threading.

Core Architecture	Intel Nehalem	Intel Core2	IBM PowerPC 450	Sun Niagara2
Type	superscalar ooo[†]	superscalar ooo	dual issue in-order	dual issue in-order
Threads/Core	2	1	1	8
Clock (GHz)	2.66	2.66	0.85	1.16
DP GFlop/s	10.7	10.7	3.4	1.16
L1 Data Cache	32KB	32KB	32KB	8KB
Private L2 Data Cache	256KB	—	—	—

Socket Architecture	Xeon X5550 Nehalem	Xeon E5355 Clovertown	Blue Gene/P Compute Chip	UltraSparc T5140 T2+ Victoria Falls
Cores per Socket	4	4 (MCM)	4	8
shared L2/L3 $	8MB	2×4MB (shared by 2)	8MB	4MB
primary memory parallelism paradigm	HW prefetch	HW prefetch	HW prefetch	MT

System Architecture	Xeon X5550 Nehalem	Xeon E5355 Clovertown	Blue Gene/P Compute Node	UltraSparc T5140 T2+ Victoria Falls
Sockets per SMP	2	2	1	2
DP GFlop/s	85.3	85.3	13.6	18.7
DRAM BW (GB/s)	51.2	21.33(read) 10.66(write)	13.6	42.66(read) 21.33(write)
DP Flop:Byte Ratio	1.66	2.66	1.00	0.29
DRAM Capacity (GB)	12	16	2	32
DRAM Type	DDR3-1066	FBDIMM-667	DDR2-425	FBDIMM-667
System Power (W) [§]	375	530	31[‡]	610
Threading	Pthreads	Pthreads	Pthreads	Pthreads
Compiler	`icc` 10.0	`icc` 10.0	xlc 9.0	`gcc` 4.0.4

Table 1. Architectural summary of evaluated platforms. [§]All system power is measured with a digital power meter while under a full computational load. [‡]Power running Linpack averaged per blade. (www.top500.org) [†]out-of-order (ooo).

4 Stencil Characteristics

The left panel of Figure 2 shows the naïve 27-point stencil code after it has been loop unrolled once. The number of reads (from the memory hierarchy) per stencil is 27, but the number of writes is only one. However, when one considers adjacent stencils, we observe substantial reuse. Thus, to attain good performance, a cache (if present) must filter the requests and present only the two compulsory (in 3C's parlance) requests per stencil to DRAM [5]. There are two compulsory requests

Stencil	Flops per Stencil	Naïve Cache refs	Arithmetic Intensity		Potential Benefit from Auto-tuning
			Naïve	Tuned	
27-pt	30	28	0.75	1.25 (1.88)	1.65× (2.5×)
27-pt CSE	18	10	0.45	0.75 (1.13)	

Table 2. Average stencil characteristics. *Arithmetic Intensity* is defined as the Total Flops / Total bytes. We assumes 8 bytes each for compulsory read and write, 8 bytes write-allocate traffic, and 16 bytes for capacity misses. The numbers in parentheses assume cache bypass. The rightmost column, *Potential Benefit from Auto-tuning*, is computed by dividing the tuned arithmetic intensity by the naïve arithmetic intensity. If memory bandwidth is the bottleneck, this is the largest speedup we should expect to see.

per stencil because every point in the grid must be read once and written once. One should be mindful that many caches are *write allocate*. That is, on a write miss, they first load the target line into the cache. Such an approach implies that writes generate twice the memory traffic as reads even if those addresses are written but never read. The two most common approaches to avoiding this superfluous memory traffic are *write through* caches or cache bypass stores.

Table 2 illustrates the dramatic difference in the per stencil averages for the number of loads and floating-point operations, both for the basic stencil as well as the highly optimized common subexpression elimination (CSE) version of the stencil. Although an ideal cache would distill these loads and stores into 8 bytes of compulsory DRAM read traffic and 8 bytes of compulsory DRAM write traffic, caches are typically not write through, infinite or fully associative, and naïve codes are not cache blocked. As such, we expect an additional 8 bytes of DRAM write allocate traffic, and another 16 bytes of capacity miss traffic (based on the caches found in superscalar processors and the reuse pattern of this stencil) — a 2.5× increase in memory traffic. Auto-tuners for structured grids will actively or passively attempt to elicit better cache behavior and less memory traffic on the belief that reducing memory traffic and exposed latency will improve performance. If the auto-tuner can eliminate all cache misses, we can improve performance by 1.65×, but if the auto-tuner also eliminates all write allocate traffic, then it may improve performance by 2.5×.

5 Stencil Optimizations

Compilers utterly fail to achieve satisfactory stencil code performance because implementations optimal for one microarchitecture may deliver suboptimal performance on another. Moreover, their ability to infer legal domain-specific transformations, given the freedoms of the C language, is limited. To improve upon this, there are a number of optimizations that can be performed at the source level to increase performance, including: NUMA-aware data allocation, array padding, multilevel blocking (shown in Figure 3), loop unrolling and reordering,

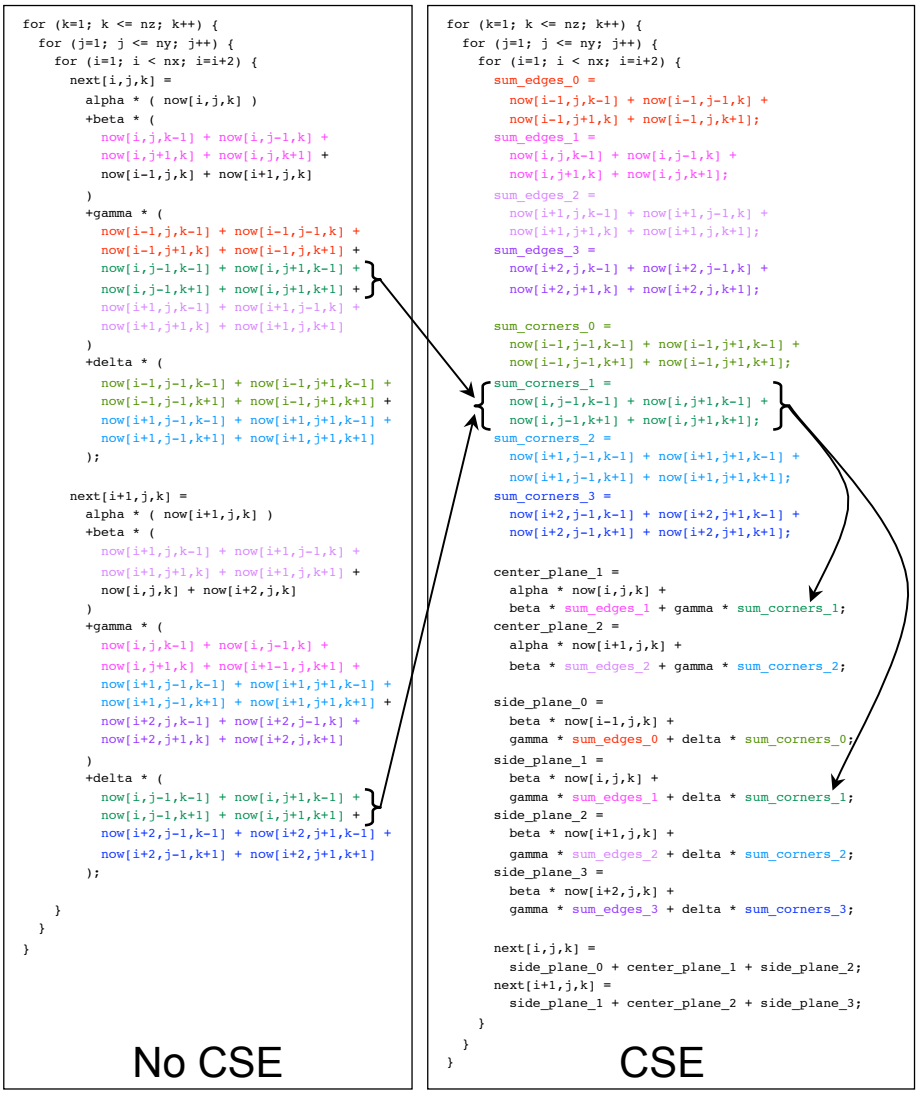


Fig. 2. Pseudo-code for one grid sweep using a 27-point stencil. Both panels have code that has been loop unrolled once in the unit-stride (x) dimension. However, the left panel does not exploit common subexpression elimination (CSE), while the right panel does. For instance, the value of the variable `sum_corners_1` is computed twice in the left panel, but only once in the right panel. The variables `sum_edges_*` in the right panel are graphically displayed in Figure 4(b), while the variables `sum_corners_*` are shown in Figure 4(c). In this example, the left panel performs 30 flops/point, while the right panel performs 29 flops/point; however, with more loop unrollings, the CSE code will approach 18 flops/point.

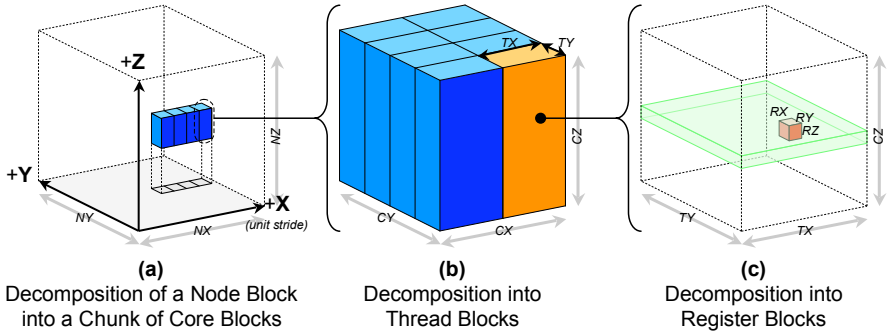


Fig. 3. Four-level problem decomposition: In (a), a *node block* (the full grid) is broken into smaller *chunks*. One core block from the chunk in (a) is magnified in (b). A single thread block from the core block in (b) is then magnified in (c) and decomposed into register blocks.

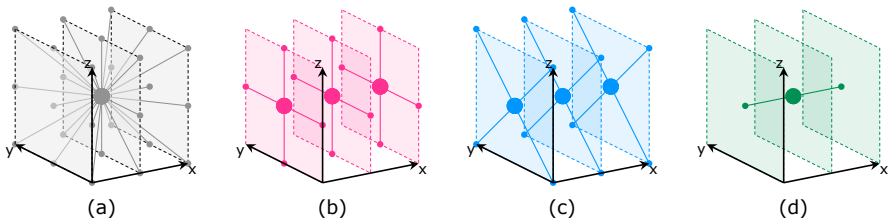


Fig. 4. Visualization of common subexpression elimination. (a) Reference 27-point stencil. (b)-(d) decomposition into 7 simpler stencils. As one loops through x, 2 of the stencils from both (b) and (c) will be reused for $x + 1$.

as well as prefetching for cache-based architectures. These well known optimizations were detailed in our previous work [4]. Remember, Jacobi's method is both out-of-place and easily parallelized (theoretically, stencils may be executed in any order). This greatly facilitates our parallelization efforts as threads must only synchronize via a barrier after executing all their assigned blocks. In this paper, we detail a new common subexpression elimination optimization.

Common subexpression elimination (CSE) involves identifying and eliminating common expressions across several stencils. This type of optimization can be considered to be an algorithmic transformation because of two reasons. First, the flop count is being reduced, and second, the flops actually being performed may be performed in a different order than our original implementation. Due to the non-associativity of floating point operations, this may well produce results that are not bit-wise equivalent to those from the original implementation.

Category	Optimization Parameter	Name	parameter tuning range x86	BG/P	VF
Data Allocation	NUMA Aware		✓	N/A	✓
	Pad by a maximum of:		32	32	32
	Pad to a multiple of:		1	1	1
Domain Decomp	Core Block Size	CX	NX	NX	{8...NX}
		CY	{4...NY}	{4...NY}	{4...NY}
		CZ	{4...NZ}	{4...NZ}	{4...NZ}
	Thread Block Size	TX	CX	CX	{8...CX}
		TY	CY	CY	{8...CY}
	Chunk Size		$\{1...\frac{NX \times NY \times NZ}{CX \times CY \times CZ \times NThreads}\}$		
Low Level	Register Block Size	RX	{1...8}	{1...8}	{1...8}
		RY	{1...4}	{1...4}	{1...4}
		RZ	{1...4}	{1...4}	{1...4}
	(explicitly SIMDized)		✓	✓	N/A
	Prefetching Distance		{0...64}	{0...64}	{0...64}
	Cache Bypass		✓	—	N/A
Tuning	Search Strategy		Iterative Greedy		
	Data-aware		✓	✓	✓

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.

Consider Figure 4. If one were to perform the reference stencil for successive points in x, we perform 30 flops per stencil. However, as we loop through x, we may dynamically create several temporaries (unweighted reductions) — Figure 4(b) and (c). For stencils at x and $x + 1$, there is substantial reuse of these temporaries. On fused multipy-add (FMA)-based architectures, we may implement the stencil by creating these temporaries and performing a linear combination using three temporaries from Figure 4(b), three from Figure 4(c) and the stencil shown in Figure 4(d). On the x86 architectures, we create a second group of temporaries by weighting the first set, the pseudo-code for which is shown in the right panel of Figure 2. With enough loop unrollings in the inner loop, the CSE code has a lower bound of 18 flops/point. Disappointingly, neither the `gcc` nor `icc` compilers were able to apply this optimization automatically.

6 Auto-Tuning Methodology

Thus far, we have described our applied optimizations in general terms. In order to take full advantage of the optimizations mentioned in Section 5, we developed an auto-tuning environment [4] similar to that exemplified by libraries like ATLAS [13] and OSKI [12]. To that end, we first wrote a Perl code generator that produces multithreaded C code variants encompassing our stencil optimizations.

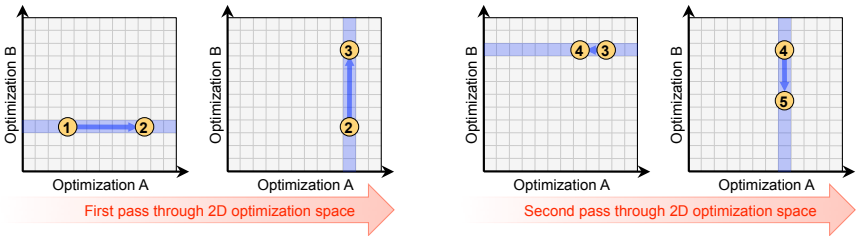


Fig. 5. Visualization of the iterative greedy search algorithm.

This approach allows us to evaluate a large optimization space while preserving performance portability across significantly varying architectural configurations.

The parameter space for each optimization individually, shown in Table 3, is certainly tractable — but the parameter space generated by combining these optimizations results in a combinatorial explosion. Moreover, these optimizations are not independent of one another; they can often interact in subtle ways that vary from platform to platform. Hence, the second component of our auto-tuner is the search strategy used to find a high-performing parameter configuration.

To find the best configuration parameters, we employed an iterative "greedy" search. First, we fixed the order of optimizations. Generally, they were ordered by their level of complexity, but there was some expert knowledge employed as well. This ordering is shown in the legend of Figure 6; the relevant optimizations were applied in order from bottom to top. Within each individual optimization, we performed an exhaustive search to find the best performing parameter(s). These values were then fixed and used for all later optimizations. We consider this to be an iterative greedy search. If all applied optimizations were independent of one another, this search method would find the global performance maxima. However, due to subtle interactions between certain optimizations, this usually won't be the case. Nonetheless, we expect that it will find a good-performing set of parameters after doing a full sweep through all applicable optimizations.

In order to judge the quality of the final configuration parameters, two metrics can be used. The more useful metric is the Roofline model [15], which provides an upper bound on kernel performance based on bandwidth and computation limits. If our fully tuned implementation approaches this bound, then further tuning will not be productive. However, the Roofline model is outside the scope of this paper. Instead, we can gain some intuition on the quality of our final parameters by doing a second pass through our greedy iterative search. This is represented by the topmost color in the legends of Figure 6. If this second pass improves performance substantially, then our initial greedy search obviously wasn't effective.

Figure 5 visualizes the iterative greedy search algorithm for a domain where there are only two optimizations: A and B. As such, the optimization space is only two dimensional. Given an initial guess at the ideal optimization parameters (point #1), we search over all possible values of optimization A while keeping

optimization B fixed. The best performing configuration along this line search then becomes point #2. Starting from point #2, we then search all possible values of optimization B while keeping optimization A fixed. This produces an even better performing configuration — point #3. At this point we've completed one pass through the greedy algorithm. We make the algorithm iterative by repeating the procedure, but starting at point #3. This results in successively better points #4 and #5.

In reality, we are dealing with many more than just two optimizations, and thus a significantly higher dimensional search space. This is the same parameter search technique that we employed in our earlier work for tuning the 3D 7-point stencil [4], however here we employ two passes across the search space.

6.1 Architecture Specific Exceptions

Due to limited potential benefit and architectural characteristics, not all architectures implement all optimizations or explore the same parameter spaces. Table 3 details the range of values for each optimization parameter by architecture. In this section, we explain the reasoning behind these exceptions to the full auto-tuning methodology. To make the auto-tuning search space tractable, we typically explored parameters in powers of two.

Both the x86 and BG/P architectures rely on hardware stream prefetching as their primary means for hiding memory latency. As previous work [7] has shown that short stanza lengths severely impair memory bandwidth, we prohibit core blocking in the unit stride (X) dimension, so $CX = NX$. Thus, we expect the hardware stream prefetchers to remain engaged and effective.

Although the standard code generator produces portable C code, compilers often fail to effectively SIMDize the resultant code. As such, we created several instruction set architecture (ISA) specific variants that produce *explicitly SIMDized* code for x86 and Blue Gene/P using intrinsics. For x86, the option of using a non-temporal store *movntpd* to bypass the cache was also incorporated.

Victoria Falls is also a cache-coherent architecture, but its multithreading approach to hiding memory latency is very different than out-of-order execution coupled with hardware prefetching. As such, we allow core blocking in the unit stride dimension. Moreover, we allow each core block to contain either 1 or 8 thread blocks. In essence, this allows us to conceptualize Victoria Falls as either a 128 core machine or a 16 core machine with 8 threads per core.

7 Results and Analysis

In our experiment, we apply a single out-of-place stencil sweep at a time to a 256^3 grid. The reference stencil code uses only two large flat 3D scalar arrays as data structures, and that is maintained through all subsequent tuning. We do increase the size of these arrays with an array padding optimization, but this does not introduce any new data structures nor change the array ordering. In addition, in order to get accurate measurements, we report the average of

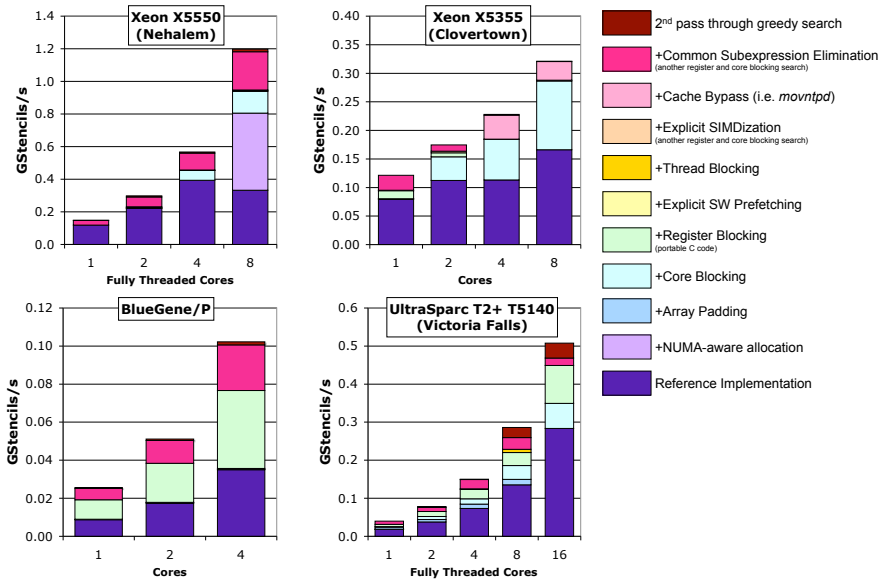


Fig. 6. Stencil performance. Before the *Common Subexpression Elimination* optimization is applied, "GStencil/s" can be converted to "GFlop/s" by multiplying by 30 flops/stencil. Note, explicitly SIMDized BG/P performance was slower than the scalar form both with and without CSE. As such, it is not shown.

at least 5 timings for each data point, and there was typically little variation among these readings.

Below we present and analyze the results from auto-tuning the stencil on each of the four architectures. To ensure fairness, across all architectures we ordered threads to first exploit all the threads on a core, then populate all cores within a socket, and finally use multiple sockets.

In all our figures, we present performance as GStencil/s (10^9 stencils per second) to allow a meaningful comparison between CSE and non-CSE kernels. In addition, in Figures 6 and 7, we stack optimization bars to represent the performance as the auto-tuning algorithm progresses through the greedy search (i.e. subsequent optimizations are built on best configuration of the previous optimizations).

7.1 Nehalem Performance

We see in Figure 6 that the performance of the reference implementation improves by 3.3× when scaling from 1 to 4 cores, but then drops slightly when we use all 8 cores across both sockets. This performance quirk is eliminated when we apply the NUMA-aware optimization.

There are several indicators that strongly suggest that it is compute-bound — core blocking shows only a small benefit, cache bypass doesn't show any benefit,

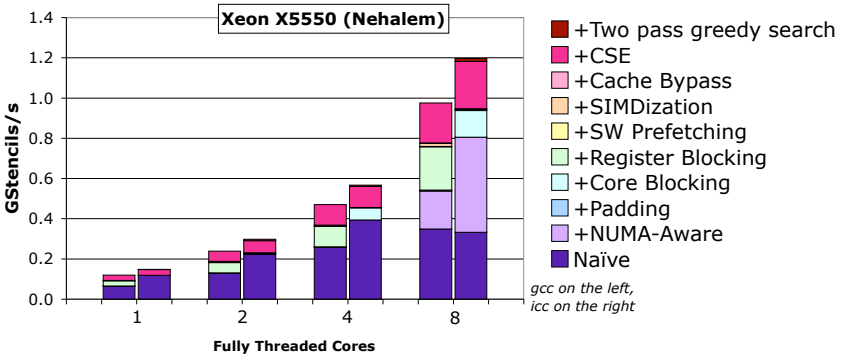


Fig. 7. A performance comparison between two compilers when auto-tuning the 27-point stencil on Nehalem. At each core count along the x-axis, the left stacked bar shows the performance of the `gcc` compiler, while the right stacked bar shows that of the `icc` compiler.

performance scales linearly with the number of cores, and the CSE optimization is successful across all core counts. Nonetheless, after full tuning, we now see a 3.6× speedup when using all 8 cores. Moreover, we also see parallel scaling of 8.1× when going from 1 to 8 cores — the ideal multicore scaling.

It is important to note that the choice of compiler plays a significant role in the effectiveness of certain optimizations as well as the final auto-tuning performance. For instance, Figure 7 shows the performance for both the `gcc` and `icc` compilers when tuning on Nehalem. There are several interesting trends that can be read from this graph. For instance, at every core count, the performance of `gcc` after applying register blocking is slightly below `icc`'s naïve (or naïve with NUMA-aware using all 8 cores) performance. Therefore, it is likely that the `gcc` compiler's unrolling facilities are not as good as `icc`'s. In addition, core blocking improves `icc` performance by at least 18% at the higher core counts, but it does not show any benefit for `gcc`. As Nehalem is nearly equally memory- and compute-bound, slightly inferior code generation capabilities will hide memory bottlenecks.

A notable deficiency in both compilers is that neither one was able to eliminate common subexpressions for the stencil. This was confirmed by examining the assembly code generated by both compilers; neither one reduced the flop count from 30 flops/point. As seen in Figure 2, the common subexpressions only arise when examining several adjacent stencil operations. Thus, a compiler must loop unroll before trying to identify common subexpressions. However, even when we *explicitly* loop unrolled the stencil code, neither compiler was able to exploit CSE. It is unclear whether the compiler lacked sufficient information to effect an algorithmic transformation or simply lacks the functionality. Whatever the reason for this, explicit CSE code improves performance by 25% for either compiler at maximum concurrency.

In general, `icc` did at least as well as `gcc` on the x86 architectures, so only `icc` results are shown for these platforms. However, the more bandwidth-limited the kernel, the less `icc`'s advantage.

7.2 Clovertown Performance

Unlike the Nehalem chip, the Clovertown is a Uniform Memory Access machine with an older front side bus architecture. This implies that the NUMA-aware optimization will not be useful and the 27-point stencil will likely be bandwidth-constrained.

As shown in Figure 6, memory bandwidth is clearly an issue at the higher core counts. When we run on 1 or 2 cores, cache bypass is not helpful, while the CSE optimization produces speedups of at least 30%, implying that the lower core counts are compute-bound. However, as we scale to 4 and 8 cores, we observe a transition to being memory-bound. The cache bypass instruction improves performance by at least 10%, while the effects of CSE are negligible. All in all, full tuning resulted in a 1.9× improvement using all 8 cores, as well as a 2.7× speedup when scaling from 1 to 8 cores.

7.3 Blue Gene/P Performance

Unlike the two previous architectures, the IBM Blue Gene/P implements the PowerPC ISA. In addition, the xlc compiler does not generate or support cache bypass at this time. As a result, the best arithmetic intensity we can achieve 1.25. The performance of Blue Gene/P seems to be compute-bound — as seen in Figure 6, memory optimizations like padding, core blocking, or software prefetching make no noticeable difference. The only optimizations that help performance are computation-related, like register blocking and CSE. Moreover, after full tuning, the 27-point stencil kernel shows perfect multicore scaling; going from 1 to all 4 cores results in a performance improvement of 3.9×.

It is important to note that while we did pass xlc the SIMDization flags when compiling the portable C code, we did not use any pragmas or functions like `#pragama unroll`, `#pragama align`, or `__alignx()` within the code. Interestingly, when we modified our stencil code generator to explicitly produce SIMD intrinsics, we observed a 10% decrease in performance of the CSE implementation. One should note that unlike x86, Blue Gene does not support an unaligned SIMD load. As such, to load a stream of consecutive elements where the first is not 16-byte aligned, one must perform permutations and asymptotically require two instructions for every two elements. Clearly this is no better than a scalar implementation of one load per element.

7.4 Victoria Falls Performance

Like the Blue Gene/P, the Victoria Falls does not exploit cache bypass. Moreover, it is a highly multi-threaded architecture with low-associativity caches.

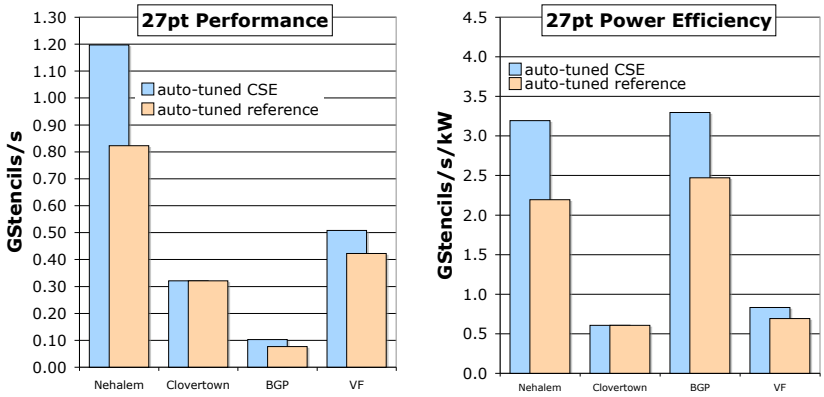


Fig. 8. A performance comparison for all architectures at maximum concurrency after full tuning. The graph displays performance for the auto-tuned stencil without common subexpression elimination (beige) and with it (blue).

To exploit these characteristics, we introduced the thread blocking optimization specifically for Victoria Falls. In the original implementation of the stencil code, each core block is processed by only one thread. When the code is thread blocked, threads are clustered into groups of 8; these groups work collectively on one core block at a time.

The reference implementation, shown in Figure 6, scales well. Nonetheless, auto-tuning was still able to achieve significantly better results. Many optimizations combined together to improve performance, including array padding, core blocking, common subexpression elimination, and a second sweep of the greedy algorithm. After full tuning, performance improved by 1.8× when using all 16 cores, and we also see parallel scaling of 13.1× when scaling to 16 cores. The fact that we almost achieve linear scaling strongly hints that it is compute-bound.

The Victoria Falls performance results are even more impressive considering that one must regiment 128 threads to perform one operation; this is 8 times as many as the Nehalem, 16 times more than Clovertown, and 32 times more than the Blue Gene/P.

7.5 Cross Platform Performance and Power Comparison

At ultra scale, power has become a severe impediment to increased performance. Thus, in this section not only do we normalize performance comparisons by looking at entire nodes rather than cores, we also normalize performance with power utilization. To that end, we use a power efficiency metric defined as the ratio of sustained performance to sustained system power — GStencil/s/kW. This is essentially the number of stencil operations one can perform per Joule of energy.

The evolution of x86 multicore chips from the Intel Clovertown, to the Intel Nehalem is an intriguing one. The Clovertown is a uniform memory access archi-

tecture that uses an older front-side bus architecture and supports only a single hardware thread per core. In terms of DRAM, it employs FBDIMMs running at a relatively slow 667 MHz. Consequently, it is not surprising to see in Figure 8 that the Clovertown is the slowest x86 architecture. In addition, due in part to the use of power-hungry FBDIMMs, it is also the least power efficient x86 platform (as evidenced in Figure 8). As previously mentioned, Intel's new Nehalem improves on previous x86 architectures in several ways. Notably, Nehalem features an integrated on-chip memory controller, the QuickPath inter-chip network, and simultaneous multithreading (SMT). It also uses three channels of DDR3 DIMMs running at 1066 MHz. On the compute-intensive CSE kernel, we still see a 3.7× improvement over Clovertown.

The IBM Blue Gene/P was designed for large-scale parallelism, and one consequence is that it is tailored for power efficiency rather than performance. This trend is starkly laid out in Figure 8. Despite Blue Gene/P delivering the lowest performance per SMP among all architectures, it still attained the *best* power efficiency. It should be noted that Blue Gene/P is two process technology generations behind Nehalem (90nm vs. 45nm).

Victoria Falls' chip multithreading (CMT) mandates one exploit 128-way parallelism. We see that Victoria Falls achieves better performance than either Clovertown or Blue Gene/P. However, in terms of power efficiency, it is second to last, better than only Clovertown. This should come as no surprise given they both use power-inefficient FBDIMMs.

8 Conclusions

In this work, we examined the application of auto-tuning to a 27-point stencil on a wide range of cache-based multicore architectures. The chip multiprocessors examined in our study lie at the extremes of a spectrum of design trade-offs that range from replication of existing core technology (multicore) to employing large numbers of simple multithreaded cores (CMT) to power-optimized designs. Results demonstrate that parallelism discovery is only a small part of the performance challenge. Of equal importance is selecting from various forms of hardware parallelism and enabling memory hierarchy optimizations.

Our work leverages the use of auto-tuners to enable portable, effective optimization across a broad variety of chip multiprocessor architectures, and successfully achieves the fastest multicore stencil performance to date. Clearly, auto-tuning was essential in providing substantial speedups regardless of whether the computational balance ultimately became memory or compute-bound; on the other hand, the reference implementation often showed poor (or even negative) scalability. Analysis shows that although every optimization was useful on at least one architecture (Figure 6), highlighting the importance of optimization within an auto-tuning framework, the portable C auto-tuner (without SIMDization and cache bypass) often delivered very good performance. This suggests one could forgo optimality for productivity without much loss.

Results show that Nehalem delivered the best performance of any of the systems, achieving more than a 6× speedup compared the previous generation Intel Clovertown — due, in-part, to the elimination of the front-side bus in favor of on-chip memory controllers. However, the low-power BG/P design offered one of the most attractive power efficiencies in our study, despite its poor single node performance; this highlights the importance of considering these design tradeoffs in an ultrascale, power intensive environment. Due to the complexity of reuse patterns endemic to stencil calculations coupled with relatively small per-thread cache capacities, Victoria Falls was perhaps the most difficult machine to optimize — it needed virtually every optimization. Through use of performance models like Roofline [15], future work will bound how much further tuning is required to fully exploit each of these architectures.

Now that power has become the primary impediment to future processor performance improvements, the definition of architectural efficiency is migrating from a notion of "sustained performance" towards a notion of "sustained performance per watt." Furthermore, the shift to multicore design reflects a more general trend in which software is increasingly responsible for performance as hardware becomes more diverse. As a result, architectural comparisons should combine performance, algorithmic variations, productivity (at least measured by code generation and optimization challenges), and power considerations.

9 Acknowledgments

We would like to express our gratitude to Sun for their machine donations. This work and its authors are supported by the Director, Office of Science, of the U.S. Department of Energy under contract number DE-AC02-05CH11231 and by NSF contract CNS-0325873. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. Finally, we express our gratitude to Microsoft, Intel, and U.C. Discovery for providing funding (under Awards #024263, #024894, and #DIG07-10227, respectively) and for the Nehalem computer used in this study.

References

1. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R.Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Advanced Supercomputing (NAS) Division, 1994.
2. M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.
3. L. Collatz. *The Numerical Treatment of Differential Equations*. Springer-Verlag, 1960.
4. 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.

5. M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.
6. 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.
7. S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *3rd Annual ACM SIGPLAN Workshop on Memory Systems Performance*, Chicago,IL, 2005.
8. T. Kim. Hardware-aware analysis and optimization of stable fluids. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 99–106, New York, NY, USA, 2008. ACM.
9. 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.
10. G. Rivera and C. Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of SC'00*, Dallas, TX, November 2000. Supercomputing 2000.
11. S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. *International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.
12. 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.
13. 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.
14. S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. In *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*, Miami, Florida, 2008.
15. 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.