Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures

Kaushik Datta[*†], Mark Murphy[†], Vasily Volkov[†], Samuel Williams[*†], Jonathan Carter[*],
Leonid Oliker[*†], David Patterson[*†], John Shalf[*], and Katherine Yelick[*†]

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

Abstract

Understanding the most efficient design and utilization of emerging multicore systems is one of the most challenging questions faced by the mainstream and scientific computing industries in several decades. Our work explores multicore stencil (nearest-neighbor) computations — a class of algorithms at the heart of many structured grid codes, including PDE solvers. We develop a number of effective optimization strategies, and build an auto-tuning environment that searches over our optimizations and their parameters to minimize runtime, while maximizing performance portability. To evaluate the effectiveness of these strategies we explore the broadest set of multicore architectures in the current HPC literature, including the Intel Clovertown, AMD Barcelona, Sun Victoria Falls, IBM QS22 PowerXCell 8i, and NVIDIA GTX280. Overall, our auto-tuning optimization methodology results in the fastest multicore stencil performance to date. Finally, we present several key insights into the architectural trade-offs of emerging multicore designs and their implications on scientific algorithm development.

1. Introduction

The computing industry has recently moved away from exponential scaling of clock frequency toward chip multiprocessors (CMPs) in order to better manage trade-offs among performance, energy efficiency, and reliability [1]. Because this design approach is relatively immature, there is a vast diversity of available CMP architectures. System designers and programmers are confronted with a confusing variety of architectural features, such as multicore, SIMD, simultaneous multithreading, core heterogeneity, and unconventional memory hierarchies, often combined in novel arrangements. Given the current flux in CMP design, it is unclear which architectural philosophy is best suited for a given class of algorithms. Likewise, this architectural diversity leads to uncertainty on how to refactor existing algorithms and tune them to take the maximum advantage of existing and emerging platforms. Understanding the most efficient design and utilization of these increasingly parallel multicore systems is one of the most challenging questions faced by the computing industry since it began.

This work presents a comprehensive set of multicore optimizations for *stencil* (nearest-neigbor) computations — a class of algorithms at the heart of most calculations involving structured (rectangular) grids, including both implicit and explicit partial differential equation (PDE) solvers. Our work explores the relatively simple 3D heat equation, which can be used as a proxy for more complex stencil calculations. In addition to their importance in scientific calculations, stencils are interesting as an architectural evaluation benchmark because they have abundant parallelism and low computational intensity, offering a mixture of opportunities for on-chip parallelism and challenges for associated memory systems.

Our optimizations include NUMA affinity, array padding, core/register blocking, prefetching, and SIMDization — as well as novel stencil algorithmic transformations that leverage multicore resources: thread blocking and circular queues. Since there are complex and unpredictable interactions between our optimizations and the underlying architectures, we develop an *auto-tuning* environment for stencil codes that searches over a set of optimizations and their parameters to minimize runtime and provide performance portability across the breadth of existing and future architectures. We believe such application-specific auto-tuners are the most practical near-term approach for obtaining high performance on multicore systems.

To evaluate the effectiveness of our optimization strategies we explore the broadest set of multicore architectures in the current HPC literature, including the out-of-order cache-based microprocessor designs of the dual-socket×quad-core AMD Barcelona and the dual-socket×quad-core Intel Clovertown, the heterogeneous local-store based architecture of the dual-socket×eight-core fast double precision STI Cell QS22 PowerXCell 8i Blade, as well as one of the first scientific studies of the hardware-multithreaded dual-socket×eight-core×eight-thread Sun Victoria Falls machine. Additionally, we present results on the single-socket×240-core multithreaded streaming NVIDIA GeForce GTX280 general purpose graphics processing unit (GPGPU).

This suite of architectures allows us to compare the mainstream multicore approach of replicating conventional cores that emphasize serial performance (Barcelona and

Clovertown) against a more aggressive manycore strategy that employs large numbers of simple cores to improve power efficiency and performance (GTX280, Cell, and Victoria Falls). It also enables us to compare traditional cache-based memory hierarchies (Clovertown, Barcelona, and Victoria Falls) against chips employing novel software controlled memory hierarchies (GTX280 and Cell). Studying this diverse set of CMP platforms allows us to gain valuable insight into the tradeoffs of emerging multicore architectures in the context of scientific algorithms.

Results show that chips employing large numbers of simpler cores offer substantial performance and power efficiency advantages over more complex serial-performance oriented cores. We also show that the more aggressive software-controlled memories of the GTX280 and Cell offer additional raw performance, performance productivity (tuning time) and power efficiency benefits. However, if the GTX280 is used as an accelerator offload engine for applications that run primarily on the host processor, the combination of limited PCIe bandwidth coupled with low reuse within GPU device memory will severely impair the potential performance benefits. Overall results demonstrate that auto-tuning is critically important for extracting maximum performance on such a diverse range of architectures. Notably, our optimized stencil is 1.5×–5.6× faster than the naïve parallel implementation, with a median speedup of 4.1× on cache-based architectures — resulting in the fastest multicore stencil implementation published to date.

2. Stencil Overview

Partial differential equation (PDE) solvers constitute 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 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 [4]. A conceptual representation of a generic stencil computation and its resultant memory access pattern is shown in Figures 1(a—b).

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 is limited to the number of points in a stencil, which is typically small. As a result, these computations generally achieve a low fraction of theoretical peak performance, since data from main memory cannot be transferred fast enough to avoid stalling the computational units on modern microprocessors. 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 [5]–[7]

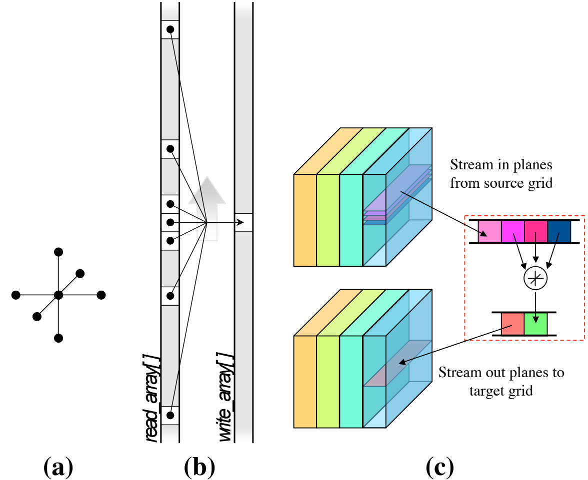

Figure 1. Stencil visualization: (a) Conceptualization of stencil in 3D space. (b) Mapping of stencil from 3D space onto linear array space. (c) Circular queue optimization: planes are streamed into a queue containing the current time step, processed, written to out queue, and streamed back.

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 [8] 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 [9] employed auto-tuners to explore a variety of effective strategies for refactoring lattice-based problems for multicore processing platforms. This study expands on prior work by developing new optimization techniques and applying them to a broader selection of processing platforms, while incorporating GPU-specific strategies.

In this work, we examine performance of the explicit 3D heat equation, naïvely expressed as triply nested loops ijk over:

$$\begin{aligned} B[i,j,k] &= C_0 A[i,j,k] + C_1(\\ &+ A[i-1,j,k] + A[i,j-1,k] + A[i,j,k-1] \\ &+ A[i+1,j,k] + A[i,j+1,k] + A[i,j,k+1]) \end{aligned}$$

This seven-point stencil performs a single Jacobi (out-of-place) iteration; thus reads and writes occur in two distinct arrays. For each grid point, this stencil will execute 8 floating point operations and transfer either 24 Bytes (for write-allocate architectures) or 16 Bytes (otherwise). Architectures with flop:byte ratios less than this stencil's 0.33 or 0.5 flops per byte are likely to be compute bound.

3. Experimental Testbed

A summary of key architectural features of the evaluated systems appear in Table I. The sustained system power data was obtained using an in-line digital power

Core Architecture	Intel Core2	AMD Barcelona	Sun Niagara2	STI Cell eDP SPE	NVIDIA GT200 SM
Type	super scalar out of order	super scalar out of order	MT dual issue†	SIMD dual issue	MT SIMD
Process	65nm	65nm	65nm	65nm	65nm
Clock (GHz)	2.66	2.30	1.16	3.20	1.3
DP GFlop/s	10.7	9.2	1.16	12.8	2.6
Local-Store	—	—	—	256KB	16KB**
L1 Data Cache	32KB	64KB	8KB	—	—
private L2 cache	—	512KB	—	—	—

System	Xeon E5355 (Clovertown)	Opteron 2356 (Barcelona)	UltraSparc T5140 T2+ (Victoria Falls)	QS22 PowerXCell 8i (Cell Blade)	GeForce GTX280
Heterogeneous	no	no	no	multicore	multichip
# Sockets	2	2	2	2	1
Cores per Socket	4	4	8	8(+1)	30 (×8)
shared L2/L3 cache	4×4MB (shared by 2)	2×2MB (shared by 4)	2×4MB (shared by 8)	—	—
DP GFlop/s	85.3	73.6	18.7	204.8	78
primary memory parallelism paradigm	HW prefetch	HW prefetch	Multithreading	DMA	Multithreading with coalescing
DRAM Bandwidth (GB/s)	21.33(read) 10.66(write)	21.33	42.66(read) 21.33(write)	51.2	141 (device) 4 (PCIe)
DP Flop:Byte Ratio	2.66	3.45	0.29	4.00	0.55
DRAM Capacity	16GB	16GB	32GB	32GB	1GB (device) 4GB (host)
System Power (Watts)§	330	350	610	270‡	450 (236)⋆
Chip Power (Watts)¶	2×120	2×95	2×84	2×90	165
Threading	Pthreads	Pthreads	Pthreads	libspe2.1	CUDA 2.0
Compiler	icc 10.0	gcc 4.1.2	gcc 4.0.4	xlc 8.2	nvcc 0.2.1221

Table 1. Architectural summary of evaluated platforms. †Each of the two thread groups may issue up to one instruction. **16 KB local-store shared by all concurrent *CUDA thread blocks* on the SM. ‡Cell Bladecenter power running Linpack averaged per blade. (www.green500.org) §All system power is measured with a digital power meter while under a full computational load. ¶Chip power is based on the maximum Thermal Design Power (TDP) from the manufacturer's datasheets. ⋆GTX280 system power shown for the entire system under load (450W) and GTX280 card itself (236W).

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.

3.1. Intel Xeon E5355 (Clovertown)

Clovertown is Intel's first foray into the quad-core arena. Reminiscent of Intel's original dual-core designs, two dual-core Xeon chips are paired onto a multi-chip module (MCM). Each core is based on Intel's Core2 microarchitecture, runs at 2.66 GHz, can fetch and decode four instructions per cycle, execute 6 micro-ops per cycle, and fully support 128b SSE, for peak double-precision performance of 10.66 GFlop/s per core.

Each Clovertown core includes a 32KB L1 cache, and each chip (two cores) has a shared 4MB L2 cache. Each socket has access to a 333MHz quad-pumped front side bus (FSB), delivering a raw bandwidth of 10.66 GB/s. Our study evaluates the Sun Fire X4150 dual-socket platform, which contains two MCMs with dual independent busses. The chipset provides the interface to four fully buffered DDR2-667 DRAM channels that can deliver an aggregate read memory bandwidth of 21.33 GB/s, with a DRAM capacity of 16GB. The full system has 16MB of L2 cache and an impressive 85.3 GFlop/s peak performance.

3.2. AMD Opteron 2356 (Barcelona)

The Opteron 2356 (Barcelona) is AMD's newest quad-core processor offering. Each core operates at 2.3 GHz, can fetch and decode four x86 instructions per cycle, execute 6 micro-ops per cycle and fully support 128b SSE instructions, for peak double-precision performance of 9.2 GFlop/s per core or 36.8 GFlop/s per socket.

Each Opteron core contains a 64KB L1 cache, and a 512MB L2 victim cache. In addition, each chip instantiates a 2MB L3 victim cache shared among all four cores. All core prefetched data is placed in the L1 cache of the requesting core, whereas all DRAM prefetched data is placed

*. Node power under a computational load can differ dramatically from both idle power and from the manufacturer's peak power specifications.

into the L3. Each socket includes two DDR2-667 memory controllers and a single cache-coherent HyperTransport (HT) link to access the other socket's cache and memory; thus delivering 10.66 GB/s per socket, for an aggregate NUMA (non-uniform memory access) memory bandwidth of 21.33 GB/s for the quad-core, dual-socket Sun X2200 M2 system examined in our study. The DRAM capacity of the tested configuration is 16 GB.

3.3. Sun UltraSparc T2+ (Victoria Falls)

The Sun "UltraSparc T2 Plus", a dual-socket $\times$ 8-core SMP referred to as Victoria Falls, presents an interesting departure from mainstream multicore chip design. Rather than depending on four-way superscalar execution, each of the 16 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. Each core may issue up to one instruction per thread group assuming there is no resource conflict. The CMT approach is designed to tolerate instruction, cache, and DRAM latency through fine-grained multithreading.

Victoria Falls instantiates only one floating-point unit (FPU) per core (shared among 8 threads). Our study examines the Sun UltraSparc T5140 with two T2 processors operating at 1.16 GHz, with a per-core and per-socket peak performance of 1.16 GFlop/s and 9.33 GFlop/s, respectively (no fused-multiply add (FMA) functionality). Each core has access to a private 8KB write-through L1 cache, but is connected to a shared 4MB L2 cache via a 149 GB/s(read) on-chip crossbar switch. Each of the two sockets is fed by two dual channel 667 MHz FBDIMM memory controllers that deliver an aggregate bandwidth of 32 GB/s (21.33 GB/s for reads, and 10.66 GB/s for writes) to each L2 (32 GB DRAM capacity). Victoria Falls has no hardware prefetching and software prefetching only places data in the L2. Multithreading may hide instruction and cache latency, but may not fully hide DRAM latency.

3.4. IBM QS22 PowerXCell 8i Blade

The Sony Toshiba IBM (STI) Cell processor adopts a heterogeneous approach to multicore, with one conventional processor core (Power Processing Element / PPE) to handle OS and control functions, combined with up to eight simpler SIMD cores (Synergistic Processing Elements / SPEs) for the computationally intensive work [2], [11]. The SPEs differ considerably from conventional core architectures due to their use of a disjoint software controlled local memory instead of the conventional hardware-managed cache hierarchy employed by the PPE. Rather than using prefetch to hide latency, the SPEs have efficient software-controlled DMA engines which decouple transfers between DRAM and the 256KB local-store from execution. This approach allows potentially more efficient use of available memory bandwidth, but increases the complexity of the programming model.

The QS22 PowerXCell 8i blade uses the enhanced double-precision implementation of the Cell processor used in the LANL Roadrunner system, where each SPE is a dual issue SIMD architecture that includes a fully pipelined double precision FPU. The enhanced SPEs can now execute two double precision FMAs per cycle, for a peak of 12.8 GFlop/s per SPE. The QS22 blade used in this study is comprised of two sockets with eight SPEs each (204.8 GFlop/s double-precision peak). Each socket has a four channel DDR2-800 memory controller delivering 25.6 GB/s, with a DRAM capacity of 16 GB per socket (32 GB total). The Cell blade connects the chips via a separate coherent interface delivering up to 20 GB/s, resulting in NUMA characteristics (like Barcelona and Victoria Falls).

3.5. NVIDIA GeForce GTX280

The recently released NVIDIA GT200 GPGPU architecture is designed primarily for high-performance 3D graphics rendering, and is available only as discrete graphics units on PCI-Express cards. However, the inclusion of double precision datapaths makes it an interesting target for HPC applications. The C-like CUDA [3] programming language interface allows a significantly simpler and much more general-purpose programming paradigm than on previous GPGPU platforms.

The GeForce GTX280 evaluated in this work is a single-socket$\times$240-core multithreaded streaming processor (30 streaming multiprocessors, or SMs, comprising 8 scalar cores). Each SM may execute one double-precision FMA per cycle, for a peak double-precision throughput of 78 GFlop/s at 1.3GHz. This performance is only attainable if all threads remain converged in a SIMD fashion. Given our code structure, we find it most useful to conceptualize each multiprocessor as a 8-lane vector core. The 64 KB register file present on each streaming multiprocessor (16,384 32-bit registers) is partitioned among vector elements; vector lanes may only communicate via the 16 KB software managed local-store, synchronizing via a barrier intrinsic. The GT200 includes hardware multithreading support. Thus, local-store and register files are further partitioned between different vector thread computations executing on the same core. In accordance with the CUDA terminology, we refer to one such vector computation as a *CUDA thread block*. CUDA differs from the traditional vector model in that thread blocks are indexed multi-dimensionally, and CUDA vector programs are written in an SPMD manner. Each vector element corresponds to a *CUDA thread*.

The GTX280 architecture provides a Uniform Memory Access interface to 1100 MHz GDDR3 DRAM, with a phenomenal peak memory bandwidth of 140.8 GB/s. The extraordinarily high bandwidth can provide a significant performance advantage over commodity DDR based CPUs by sacrificing capacity. However, the GTX280 cannot directly access system (CPU) memory. As a result, problems that either exceed the 1 GB on-board memory capacity or cannot be run exclusively on the GTX280 coprocessor can suffer from costly data transfers between graphics DRAM and the host DRAM over the PCI-express (PCIe) x16 bus. Consequently, we present both the GTX280 results

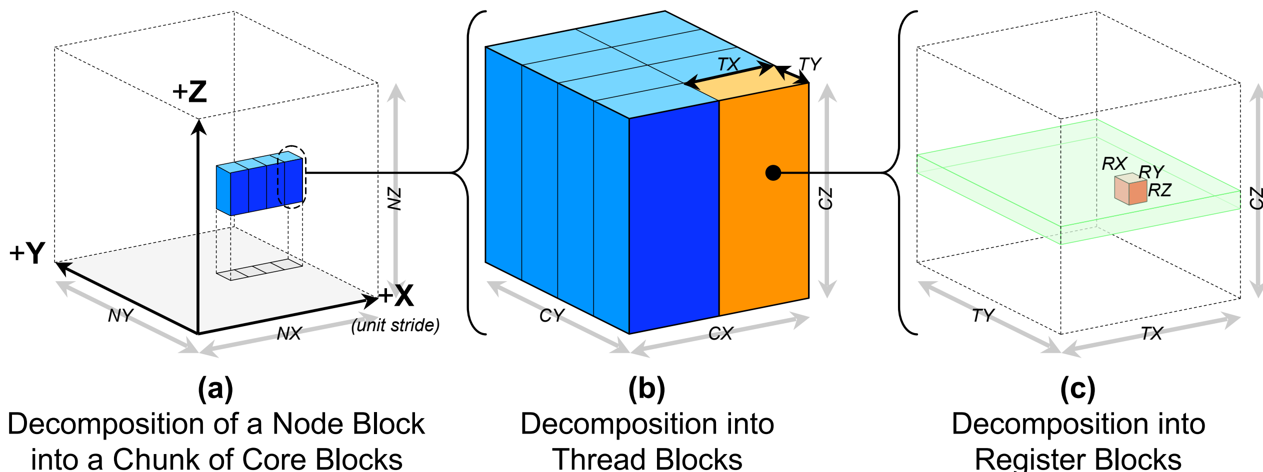

Figure 2. Four-level problem decomposition: In (a), a *node block* (the full grid) is broken into smaller *chunks*. All the *core blocks* in a chunk are processed by the same subset of threads. One core block from the chunk in (a) is magnified in (b). A properly sized core block should avoid capacity misses in the last level cache. 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.

unburdened by the host data transfers, to demonstrate the ultimate potential of the architecture, as well as performance handicapped by the data transfers.

4. Optimizations

To improve stencil performance across our suite of architectures, we examine a wide variety of optimizations, including: NUMA-aware allocation, array padding, multi-level blocking, loop unrolling and reordering, as well as prefetching for cache-based architectures and DMA for local-store based architectures. Additionally, we present two novel multicore-specific stencil optimizations: circular queue and thread blocking. These techniques, applied in the order most natural for each given architectures (generally ordered by their level of complexity), can roughly be divided into four categories: problem decomposition, data allocation, bandwidth optimizations, and in-core optimizations. In the subsequent subsections, we discuss these techniques as well as our overall auto-tuning strategy in detail. Any exceptions are further explained in Section 4-F. In addition, a summary of our optimizations and their associated parameters is shown in Table II.

4.1. Problem Decomposition

Although our data structures are just two large 3D scalar arrays, we apply a four-level decomposition strategy across all architectures. This allows us to simultaneously implement parallelization, cache blocking, and register blocking, as visualized in Figure 2. First, a *node block* (the entire problem) of size $NX \times NY \times NZ$ is partitioned in all three dimensions into smaller *core blocks* of size $CX \times CY \times CZ$, where X is the unit stride dimension. This first step is designed to avoid last level cache capacity misses by effectively cache blocking the problem. Each core block is further partitioned into a series of *thread blocks* of size $TX \times TY \times CZ$. Core blocks and thread blocks are the same size in the Z (least unit stride) dimension, so when $TX = CX$ and $TY = CY$, there is only one thread per core block. This second decomposition is designed to exploit the common locality threads may have within a shared cache or local memory. Note our *thread block* is different than *a CUDA thread block*. Then, our third decomposition partitions each thread block into register blocks of size $RX \times RY \times RZ$. This allows us to take advantage of the data level parallelism provided by the available registers.

Core blocks are also grouped together into *chunks* of size *ChunkSize* which are assigned to an individual core. The number of threads in a core block ($Threads_{core}$) is simply $\frac{CX}{TX} \times \frac{CY}{TY}$, so we then assign these chunks to a group of $Threads_{core}$ threads in a round-robin fashion (similar to the *schedule* clause in OpenMP's *parallel for* directive). Note that all the core blocks in a chunk are processed by the same subset of threads. When *ChunkSize* = 1, spaced out core blocks may map to the same set in cache, causing conflict misses. However, we do gain a benefit from diminished NUMA effects. In contrast, when *ChunkSize* = max, contiguous core blocks are mapped to contiguous set addresses in a cache, reducing conflict misses. This comes at the price of magnified NUMA effects. We therefore tune *ChunkSize* to find the best tradeoff of these two competing effects. Thus, our fourth and final decomposition is from chunks to core blocks. In general, this decomposition scheme allows us to explain shared cache locality, cache blocking, register blocking,

Category	Optimization Parameter	Name	Clovertown	Barcelona	Victoria Falls	Cell Blade	GTX280
			parameter tuning range by architecture				
Data Allocation	NUMA Aware		N/A	✓	✓	✓	N/A
	Pad to a multiple of:		1	1	1	16	16
Domain Decomp	Core Block Size	*CX*	NX	NX	{8...NX}	{64...NX}	{16...32}
		CY	{8...NY}	{8...NY}	{8...NY}	{8...NY}	CX
		CZ	{128...NZ}	{128...NZ}	{128...NZ}	{128...NZ}	64
	Thread Block Size	*TX*	CX	CX	{8...CX}	CX	1
		TY	CY	CY	{8...CY}	CY	CY/4
	Chunk Size		$\{1\ldots\frac{NX\times NY\times NZ}{CX\times CY\times CZ\times NThreads}\}$				N/A
Low Level	Register Block Size	*RX*	{1...8}	{1...8}	{1...8}	2	TX
		RY	{1...2}	{1...2}	{1...2}	8	TY
		RZ	{1...2}	{1...2}	{1...2}	1	1
	(explicitly SIMDized)		✓	✓	N/A	✓	N/A
	Prefetching Distance		{0...64}	{0...64}	{0...64}	N/A	N/A
	DMA Size		N/A	N/A	N/A	CX×CY	N/A
	Cache Bypass		✓	✓	N/A	implicit	implicit
	Circular Queue		—	—	—	✓	✓

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

and NUMA-aware allocation within a single formalism.

4.2. Data Allocation

The source and destination grids are each individually allocated as one large array. Since the decomposition strategy has deterministically specified which thread will update each point, we wrote a parallel initialization routine to initialize the data. Thus, on non-uniform memory access (NUMA) systems that implement a "first touch" page mapping policy, data is correctly pinned to the socket tasked to update it. Without this *NUMA-aware* allocation, performance could easily be cut in half.

Some architectures have relatively low associativity shared caches, at least when compared to the product of threads and cache lines required by the stencil. On such machines, conflict misses can significantly impair performance. Moreover, some architectures prefer certain alignments for coalesced memory accesses; failing to do so can greatly reduce memory bandwidth. To avoid these pitfalls, we pad the unit-stride dimension ($NX \leftarrow NX + pad$).

4.3. Bandwidth Optimizations

The architectures used in this paper employ four principal mechanisms for hiding memory latency: hardware prefetching, software prefetching, DMA, and multithreading. The x86 architectures use hardware stream prefetchers that can recognize unit-stride and strided memory access patterns. When such a pattern is detected successive cache lines are prefetched without first being demand requested. Hardware prefetchers will not cross TLB boundaries (only 512 consecutive doubles) and can be easily halted by spurious memory requests. Both conditions may arise when $CX < NX$ — i.e. when core blocking results in stanza access patterns. Although this is not an issue on multithreaded architectures, they may not be able to completely cover all cache and memory latency. In contrast, software prefetching, which is available on all cache-based machines, does not suffer from either limitation. However, it can only express a cache line's worth of memory level parallelism. In addition, unlike a hardware prefetcher (where the prefetch distance is implemented in hardware), software prefetching must specify the appropriate distance to effectively hide memory latency. DMA is only implemented on Cell, but can easily express the stanza memory access patterns. DMA operations are decoupled from execution and are implemented as double buffered reads of core block planes.

So far we have discussed optimizations designed to hide memory latency and thus improve memory bandwidth, but we can extend this discussion to optimizations that minimize memory traffic. The circular queue implementation, visualized in Figure 1(c), is one such technique. This approach allocates a shadow copy of the planes of a core block in local memory or registers. The seven-point stencil requires three read planes to be allocated, which are then populated through loads or DMAs. However, it can often be beneficial to allocate an output plane and double buffer reads and writes as well. The advantage of the circular queue is the potential avoidance of lethal conflict misses. We currently explore this technique only on the local-store architectures but note that future work will extend this to the cache based architectures.

Another technique for reducing memory traffic is the cache bypass instruction. On write-allocate architectures, a write miss will necessitate the allocation of a cache line.

Before execution can proceed, the contents of the line are filled from main memory. In the case of stencil codes, this superfluous transfer is wasteful as the entire line will be completely overwritten. There are cache initialization and cache bypass instructions that we exploit to eliminate this unnecessary fill — in SSE this is *movntpd*. By exploiting this instruction, we may increase arithmetic intensity by 50%. If bandwidth bound, this can also increase performance by 50%. This benefit is implicit on the cache-less Cell and GT200 architectures.

4.4. In-core Optimizations

Although superficially simple, there are innumerable ways of optimizing the execution of a 7-point stencil. After tuning for bandwidth and memory traffic, it often helps to explore the space of inner loop transformations to find the fastest possible code. To this end, we wrote a code generator that could generate any unrolled, jammed and reordered version of the stencil. Register blocking is, in essence, unroll and jam in X, Y, or Z. This creates small $RX \times RY \times RZ$ blocks that sweep through each thread block. Larger register blocks have better surface-to-volume ratios and thus reduce the demands for L1 cache bandwidth. However, they may significantly increase register pressure as well.

Although the standard code generator produces portable C code, compilers often fail to effectively SIMDize the resultant code. As such, we created several ISA-specific variants that produce SIMD code for x86 and Cell. These versions will deliver much better in-core performance than a compiler. However, as one might expect, this may have a limited benefit on memory-intensive codes.

4.5. Auto-Tuning Methodology

Thus far, we have described hierarchical blocking, unrolling, reordering, and prefetching in general terms. Given the combinatoric complexity of the aforementioned optimizations coupled with the fact that these techniques interact in subtle ways, we develop an auto-tuning environment similar to that exemplified by libraries like ATLAS [12] and OSKI [13]. To that end, we first wrote a Perl code generator that produces multithreaded C code variants encompassing our stencil optimizations. This approach allows us to evaluate a large optimization space while preserving performance portability across significantly varying architectural configurations. The second component of an auto-tuner is the auto-tuning benchmark that searches the parameter space (shown in Table II) through a combination of explicit search for global maxima with heuristics for constraining the search space. At completion, the auto-tuner reports both peak performance and the optimal parameters.

4.6. Architecture Specific Exceptions

Due to limited potential benefit and architectural characteristics, not all architectures implement all optimizations or explore the same parameter spaces. Table II 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.

The x86 architectures like Clovertown and Barcelona rely on hardware stream prefetching as their primary means for hiding memory latency. As previous work [10] 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. Second, as these core architectures are not multithreaded, we saw no reason to attempt thread blocking. Thus, the thread blocking search space was restricted so that $TX = CX$, and $TY = CY$. Both x86 machines implement SSE2. Therefore, we implemented a special SSE SIMD code generator for the x86 ISA that would produce both explicit SSE SIMD intrinsics for computation as well as the option of using a non-temporal store *movntpd* to bypass the cache. On both machines, the threading model was Pthreads.

Although Victoria Falls is also a cache-coherent architecture, 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. In addition, there are no supported SIMD or cache bypass instrinsics, so only the portable pthreads C code was run.

Unlike the previous three machines, Cell uses a cacheless local-store architecture. Moreover, instead of prefetching or multithreading, DMA is the architectural paradigm utilized to express memory level parallelism and hide memory latency. This has a secondary advantage in that it also eliminates superfluous memory traffic from the cache line fill on a write miss. The Cell code generator produces both C and SIMDized code. However, our use of SDK 2.1 resulted in poor double precision code scheduling as the compiler was scheduling for a QS20 rather than a QS22. Unlike the cache-based architectures, we implement the dual circular queue approach on each SPE. Moreover, we double buffer both reads and writes. For optimal performance, DMA must be 128 byte (16 doubles) aligned. As such, we pad the unit stride (X) dimension of the problem so that $NX+2$ is a multiple of 16. For expediency, we also restrict the minimum unit stride core blocking dimension (CX) to be 64. The threading model was IBM's libspe.

The GT200 has architectural similarities to both Victoria Falls (multithreading) and Cell (local-store based). However, it differs from all other architectures in that the device DRAM is disjoint from the host DRAM. Unlike the other architectures, the restrictions of the CUDA programming model constrained the auto-tuner to a very limited number of cases. First, we only explore only two core block sizes: 32×32 and 16×16. We depend on CUDA to

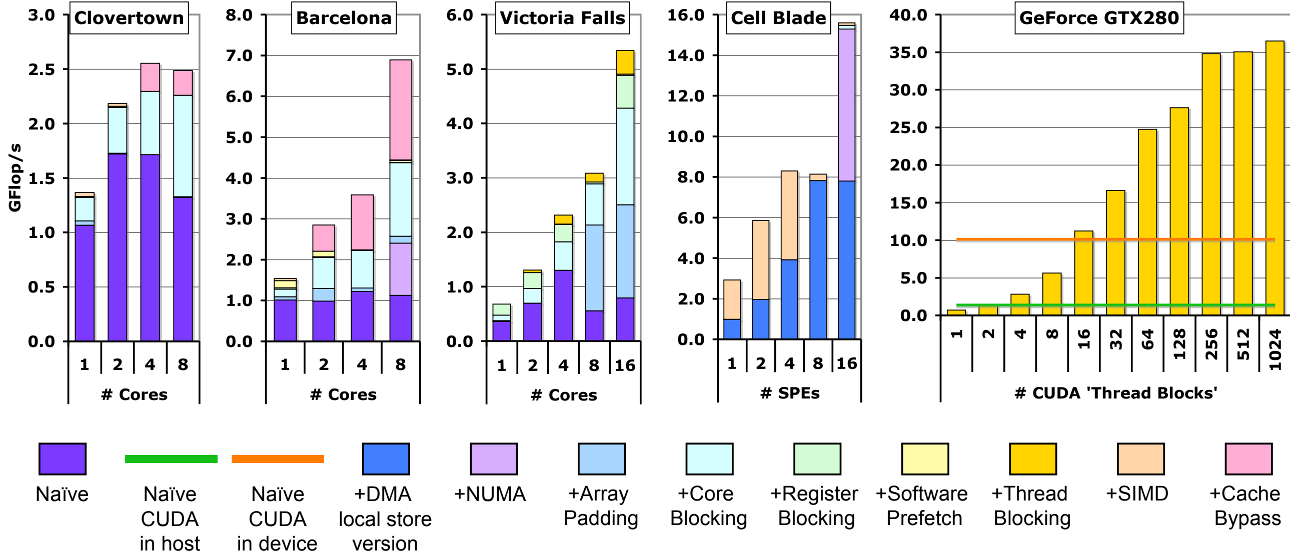

Figure 3. Optimized stencil performance results in double precision for Clovertown, Barcelona, Victoria Falls, a QS22 Cell Blade, and the GeForce GTX280. Note: naïve CUDA denotes the programming style NVIDIA recommends in tutorials. Host and device refer to CPU and GPU DRAM respectively.

implement the threading model and use thread blocking as part of the auto-tuning strategy. The thread blocks for the two core block sizes are restricted to $1{\times}8$ and $1{\times}4$ respectively. Since the GT200 contains no automatically-managed caches, we use the circular queue approach that was employed in the Cell stencil code. However, the register file is four times larger than the local memory, so we chose register blocks to be the size of thread blocks ($RX = TX, RY = TY, RZ = 1$) and chose to keep some of the planes in the register file rather than shared memory.

5. Performance Results and Analysis

To evaluate our optimization strategies and compare architectural features, we examine a 256^3 stencil calculation which, including ghost cells, requires a total of 262 MB of memory. Since scientific computing relies primarily on double precision, all of our computations are also performed in double precision across all architectures. In addition, to keep results both consistent and comparable, we exploit affinity routines to first utilize all the hardware thread contexts on a single core, then scale to all the cores on a socket, and finally use all the cores across all sockets. This approach prevents the benchmark code from exploiting a second socket's memory bandwidth until all the cores on a single socket are in use.

The stacked bar graphs in Figures 3 show individual platform performance as a function of core concurrency (using fully threaded cores). The stacked bars indicate the performance contribution from each of the relevant optimizations. All the attempted optimizations are listed in the legend below. On the Cell, only the SPEs are used, and on the GTX280 we plot performance as a function of the number of *CUDA thread blocks* per *CUDA grid*. However, neither the Cell SPEs nor the GTX280 can run our portable C code, so there is no truly naïve implementation for either platform. Instead, for Cell, a DMA and local-store implementation serves as the baseline. For the GTX280, there are two baselines, both of which use a programming style recommended in NVIDIA tutorials that we call *Naïve CUDA*. The lower green baseline represents the case where the entire grid must be transferred back and forth between host and device memory once per sweep (accelerator mode). In contrast, the upper red line is the ideal case when the grid may reside in device memory without any communication to host memory (stand-alone). A typical application will lie somewhere in between.

Figure 4 is a set of summary graphs that allows comparisons across all architectures. Figure 4(a) focuses on maximum performance, while Figure 4(b) examines per core scalability. Then, we examine each architecture's resource utilization in the 2D scatter plot of Figure 4(c). We computed the sustained percentage of the *attainable* memory bandwidth (A_{BW}) and attainable computational rate (A_{Flop}) for each architecture. The former fraction is calculated as the sustained stencil bandwidth divided by the OpenMP Stream [14] (copy) bandwidth. For Cell, we simply commented out the computation, but continued to execute the DMAs. Similarly, the attainable fraction of peak is the achieved stencil GFlop/s rate divided by the in-cache stencil performance derived by running a small problem that fits in the aggregate cache (or local-store). For Cell, we simply commented out the DMAs, but performed the stencils on data already in the local-store. Thus floating-point bound architectures will be near 100% A_{Flop} on the x-axis (GFlop/s), while memory bound platforms will approach 100% A_{BW} on the y-axis (GB/s).

These coordinates allow one to estimate how balanced or limited the architecture is. Architectures that depart from the upper or right edges of the figure fail to saturate one of

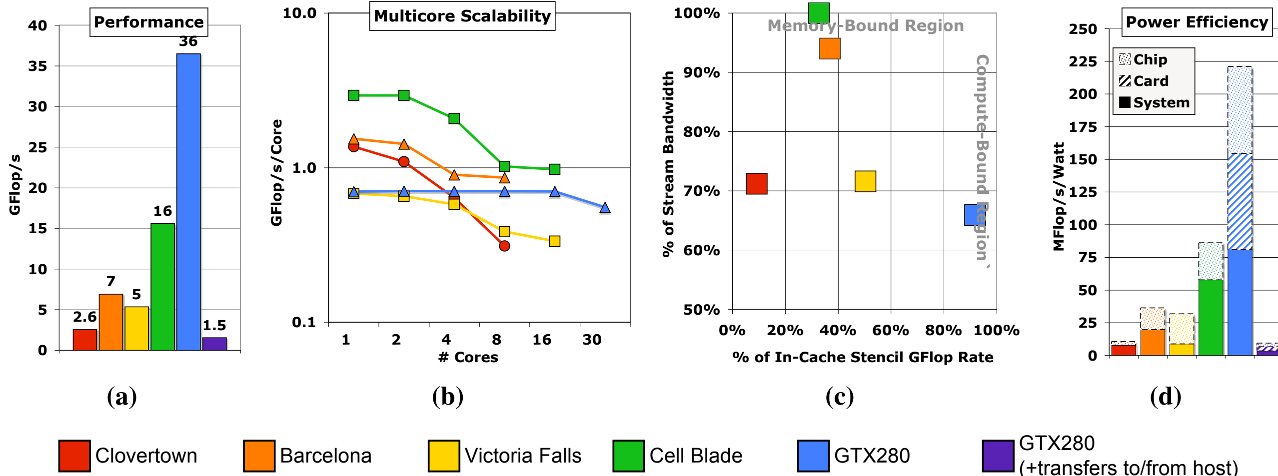

Figure 4. **Comparative architectural results for (a) aggregate double precision performance, (b) multicore scalability, (c) fraction of *attainable* computational and bandwidth performance, and (d) power efficiency (MFlop/s/Watt) based on system, GPU card, and chip power. GTX280-Host refers to performance with the PCIe host transfer overhead on each sweep. This performance is so poor it cannot be shown in (b) or (c).**

these key resources, while systems achieving near 100% for both metrics are well balanced for our studied stencil kernel. Note that attainable peak is a tighter performance bound than the traditionally used ratios of machine or algorithmic peak as it incorporates many microarchitectural and compiler limitations. Nevertheless, all three of these metrics have a potentially important role in understanding performance behavior. Finally, Figure 4(d) compares power efficiency across our architectural suite in terms of system-, card- and chip-power utilization.

5.1. Clovertown Performance

The Clovertown performance results are shown in the leftmost graph of Figure 3. Since the Clovertown cores have uniform memory access, the system is unaffected by NUMA optimizations. Notable performance benefits are seen from core blocking and cache bypass (1.7× and 1.1× speedups respectively at max concurrency). Additionally, for small numbers of cores Clovertown benefits from explicit SIMDization. Note that experiments on a smaller 128^3 calculation (not shown) saw little benefit from auto-tuning, as the entire working set easily fit within Clovertown's large 2MB per core L2 working set.

Clovertown's poor multicore scaling indicates that the system rapidly becomes memory bandwidth limited — utilizing approximately 4.5 GB/s after engaging only two of the cores, which is close to the practical limit of a single FSB [15]. The quad pumping of the dual FSB architecture has reduced data transfer cycles to the point where they are on parity with coherency cycles. Given the coherency protocol overhead, it is not too surprising that the performance does not improve between the four-core and eight-core experiment (when both FSBs are engaged), despite the doubling of the peak aggregate FSB bandwidth.

Overall, Clovertown's single-core performance of 1.4 GFlop/s grows by only 1.8× when using all eight cores, resulting in aggregate node performance of only 2.5 GFlop/s — about 2.7× slower than Barcelona. For this problem, the improved floating point performance of this architecture is wasted because of the sub-par FSB performance. We expect that Intel's forthcoming Nehalem, which eliminates the FSB in favor of dedicated on-chip memory controllers, will address many of these deficiencies.

5.2. Barcelona Performance

Figure 3 presents Opteron 2356 (Barcelona) results. Observe that the NUMA-aware version increases performance by 115% when all sockets are engaged; this highlights the potential importance of correctly mapping memory pages in systems with memory controllers on each socket. Additionally, the optimal (auto-tuned) core blocking resulted in an additional 70% improvement (similar to the Clovertown). The cache bypass (streaming store) intrinsic provides an additional improvement of 55% when using all eight cores — indicative of its importance only when the machine is memory bound. Using this optimization reduces memory traffic by 33% and thus changes the stencil kernel's flop:byte ratio from $\frac{1}{3}$ to $\frac{1}{2}$. This potential 50% improvement corresponds closely to the 55% observed improvement — confirming the memory bound nature of the stencil kernel on this machine.

Register blocking and software prefetching ostensibly had little performance effect on Barcelona; however, the auto-tuning methodology explores a large number of optimizations in the hope that they may be useful on a given architecture. As it is difficult to predict this beforehand, it is still important to try each relevant optimization.

The Opteron's per-core scalability can be seen in Figures 4(b). Overall, we see reasonably efficient scalability up to two cores, but then a fall off at four cores — indicative that the socket is only reaching a memory bound limit when all four cores are engaged. When the

second socket and its additional memory controllers are employed, near linear scaling is attained. Note, the X2200 M2 is not a split rail motherboard. As such, the lower northbridge frequency may reduce memory bandwidth, and thus performance by up to 20%.

5.3. Victoria Falls Performance

The Victoria Falls experiments in Figure 3 show several interesting trends. Using all sixteen cores, Victoria Falls sees a 6.1× performance benefit from array padding and core/register blocking, plus an additional 1.1× speedup from thread blocking to achieve an aggregate total performance of 5.3 GFlop/s. Therefore, the fully-optimized code generated by the auto-tuner was 6.7× faster than the naïve code. Victoria Falls is thus 2.7× faster than a fully-packed Clovertown system, but still 1.3× slower than Barcelona. The thread blocking optimization successfully boosted performance via better per-core cache behavior. However, the automated search to identify the best parameters was relatively lengthy, since the parameter space is larger than conventional threading optimizations.

5.4. Cell Performance

Looking at the Cell results in Figure 3, recall that generic microprocessor-targeted source code cannot be naïvely compiled and executed on the SPE's software controlled memory hierarchy. Therefore, we use a DMA local-store implementation as the baseline performance for our analysis. Our Cell-optimized version utilizes an auto-tuned circular queue algorithm (described in Section 4-F).

Examining Cell behavior reveals that the system is clearly computationally bound for the baseline stencil calculation when using one to four cores — as visualized in Figure 4(b). In this region, there is a significant performance advantage in using hand optimized SIMD code. However, at concurrencies greater than 8 cores, there is essentially no advantage — the machine is clearly bandwidth limited. The only pertinent optimization is optimal NUMA-aware data placement. Exhaustively searching for the optimal core blocking provided no appreciable speedup over a baseline heuristic. Although the resultant performance of 15.6 GFlop/s is a low fraction of performance when operating from the local-store, it achieves nearly 100% of the streaming memory bandwidth as evidenced in the scatter plot in Figure 4(c). Although this Cell blade does not provide a significant performance advantage over the previous Cell blade for memory intensive codes, it provides a tremendous productivity advantage by ensuring double precision performance is never the bottleneck — one only need focus on DMA and local-store blocking.

5.5. GTX280 Performance

Finally, we examine the new double-precision results of the NVIDIA GT200 (GeForce GTX280) shown in Figure 3. In this graph, we superimpose three sets of results. NVIDIA often recommends a style of CUDA programming where each *CUDA thread* within a *CUDA thread block* is responsible for a single calculation — a stencil for our code. We label this approach as naïve CUDA. As some applications may require the CPU to have frequent access to the entire problem, where others may be completely ported to a GPU, we further differentiate this category into two approaches: naïve CUDA in host, and naïve CUDA in device. The former presumes the entire problem must start and finish each time step in host (CPU) memory, while the latter allows the data to remain in device (GPU) memory. In either of these implementations the number of *CUDA thread blocks* is huge and all cores are used and balanced. Finally, we show our optimized implementation using 16×4 threads tasked with processing 16×16 blocks as a function of the number of *CUDA thread blocks*.

Note that GPGPU studies often do not address the performance overhead of CPU to GPU data transfer. For large-scale calculations, the actual performance impact will depend on the required frequency of GPU-host data transfers. Some numerical methods conduct only a single stencil sweep before other types of computation are performed, and will potentially suffer the roundtrip host latency between each iteration. However, there are important algorithmic techniques that require consecutive stencil sweeps — thereby amortizing the host data transfers. We therefore present both cases — the optimistic case, unburdened by the host transfers, and the pessimistic case that reflects the performance constraints of a hybrid programming model.

The naïve CUDA in host only affords us with about 1.4 GFlop/s. This is completely limited by a PCIe x16 sustained bandwidth of only 3.4GB/s. Clearly, for many applications such poor performance is unacceptable. We may optimize away the potentially superfluous PCIe transfers and only operate from device memory. Such an implementation delivers about 10.1 GFlop/s — a 3× speedup. Our optimized and tuned implementation selects the appropriate decomposition and number of threads. Unfortunately, the problem decomposes into a power of two number of *CUDA thread blocks* which we must run on 30 streaming multiprocessors. Clearly when the number of *CUDA thread blocks* is less than 30, there is a linear mapping without load imbalance. However, at 32 *CUDA thread blocks* the load imbalance is maximal (some cores are tasked with twice as many blocks as others). As concurrency increases load balance diminishes and performance saturates at a phenomenal 36.5 GFlop/s.

Figure 4(b) shows scalability as a function of the number of *CUDA thread blocks* from 1 to 16. Additionally, it shows performance when 1024 blocks are mapped to 30 streaming multiprocessors. Clearly, scalability is very good — this machine's phenomenal memory bandwidth is not a bottleneck. However, the scatter plot suggests the code is achieving nearly 100% of this algorithm's double precision peak flop rate while consuming better than 66% of its memory bandwidth. Clearly, if the number of double precision units per streaming multiprocessor were doubled, the GTX280 could not fully exploit it.

5.6. Architectural Comparison

Figure 4(a) compares raw performance across the evaluated architectures. For stencil problems where the overhead associated with copying the grid over PCIe can be amortized (or eliminated), the GTX280 delivers 36 GFlop/s, by far the best performance among the evaluated architectures — achieving 2.3×, 6.8×, 5.3×, and 14.3× speedups compared with Cell, Victoria Falls, Barcelona, and Clovertown respectively. However, for problems where this transfer cannot be eliminated, the GPU-CPU mixed implementation drops dramatically, achieving only 60% of Clovertown's relatively poor performance. In this scenario, Cell is the clear winner, delivering speedups of 6.1×, 2.3×, and 2.9× over the Clovertown, Barcelona, and Victoria Falls respectively.

Figure 4(b) allows us to compare the scalability of the various architectures. The poor scalability seen by the high flop:byte Cell and Barcelona is easily explained by their extremely high fractions of peak memory bandwidth seen in Figure 4(c). Similarly, the low flop:byte GTX280's near perfect scalability is well explained by its limited peak double precision performance. Unfortunately, neither Clovertown nor Victoria Falls' poor multicore scalability is well explained by either memory bandwidth or in-cache performance. Clovertown is likely unable to achieve sufficient memory bandwidth because cache coherence traffic consumes a substantial fraction of available FSB bandwidth. In addition, for both Clovertown and Victoria Falls, we do not include capacity or conflict misses when calculating bandwidth — unlike the local-store based architectures. As such, if either of those are high, then we are significantly underestimating bandwidth.

We highlight that across all three cache-based machines, the naïve implementation has shown both poor scalability and performance. In fact, for all three architectures, the naïve implementation is fastest when run at a lower concurrency than the maximum. This is an indication that even for this relatively simple computation, scientists cannot rely on compiler technology to effectively utilize the system's resources. However, once our auto-tuning methodology is employed, results show up to a dramatic 5.6× improvement, which was achieved on the Barcelona.

Finally, Figure 4(d) presents the stencil computational power efficiency (MFlop/s/Watt) of our studied systems (Table I) — one of the most crucial issues in large-scale computing today. The solid regions of the stacked-bar graph represent power efficiency based on measured total sustained system power, while the dashed region for the GTX280 is the power for the card only. Finally, the dotted region denotes power efficiency when only counting each chip's maximum TDP. This allows one to differentiate drastically different machine configurations and server expandability.

If (optimistically) no host transfer overhead is required, the GTX280-based system† is more power efficient in double precision than Cell, Barcelona, Victoria Falls, and Clovertown by an impressive 1.4×, 4.1×, 9.2×, and 10.5×, respectively. However, if (pessimistically) a CPU-GPU PCIe roundtrip is necessary for each stencil sweep, the GTX280 attains the worst power efficiency of the evaluated systems, whereas Cell's system power efficiency exceeds the GTX280 by almost 17×, and outperforms Barcelona, Victoria Falls, and Clovertown by 2.9×, 6.6×, and 7.5×.

While the Cell's and Opteron's DDR2 DRAM consume a relatively modest amount of power, the FBDIMMs used in the Clovertown and Victoria Falls systems are extremely power hungry and severely reduce the measured power efficiency of those systems. In fact, just the FBDIMMs used in Victoria Falls require a startling 200W; removing a rank or a switch to unbuffered DDR2 DIMMs might improve power efficiency by more than 16%.

†. GTX280 power consumption baseline includes total system power as well as the idle host CPU

6. Summary and Conclusions

This work examines optimization techniques for stencil computations on a wide variety of multicore architectures and demonstrates 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, made more challenging by the separate address spaces, software-managed memory local-stores, and NUMA features that appear in multicore systems today. Our work leverages 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.

The chip multiprocessors examined in our study span the spectrum of design trade-offs that range from replication of existing core technology (multicore) to employing large numbers of simpler cores (manycore) and novel memory hierarchies (streaming and local-store). For algorithms with sufficient parallelism, results show that employing a large number of simpler processors offers higher performance potential than small numbers of more complex processors optimized for serial performance. This is true both for peak performance and for performance per watt (power efficiency). We also see substantial benefit to novel strategies for hiding memory latency, such as using large numbers of threads (Victoria Falls and GTX280) and employing software controlled memories (Cell and GTX280). However, the software control of local-store architectures results in a difficult trade-off, since it gains performance and power efficiency at a significant cost to programming productivity.

Results also show that the new breed of GPGPU, exemplified by the NVIDIA GTX280, demonstrate substantial performance potential if used stand alone — achieving an impressive 36 GFlop/s in double precision for our stencil computation. The massive memory bandwidth available on this sytem is crucial in achieving this performance. However, the GTX280 designers traded memory capacity

in favor of bandwidth, potentially limiting the GPGPU's applicability in scientific applications. Additionally, when used as a coprocessor the performance advantage can be substantially constrained by Amdahl's law. The same limitation exists for *any* heterogeneous architecture that is programmed as an accelerator, but is exacerbated by the need to copy application data structures from host memory to accelerator memory (reminiscent of the lessons learned on the Thinking Machines CM5).

Comparing the cache-based systems, the recently-released Barcelona platform sustains higher performance and power efficiency than Clovertown or Victoria Falls for our stencil code. However, the highly multithreaded architecture of Victoria Falls allowed it to effectively tolerate memory transfer latency — thus requiring fewer optimizations, and consequently less programming overhead, to achieve high performance.

Now that power has become the primary impediment to future 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. We believe that our work represents a template of the kind of architectural evaluations that are necessary to gain insight into the tradeoffs of current and future multicore designs.

A disturbing aspect of the cache-based architectures' performance in our study is the complete lack of multicore scalability without auto-tuning — which may lead to a programmer's false impression that the architecture has approached a performance ceiling and holds little potential for further improvements. However, auto-tuning improves the relatively poor per-core speedups on Barcelona and Victoria Falls to near perfect scaling, resulting in a $5.6\times$ and $4.1\times$ speedup (respectively) over the original untuned parallel code. Although many of the techniques incorporated into our auto-tuner are ostensibly incorporated into compiler technology, computational scientists assuming that compilers will optimize performance of PDE solvers on multicores — even those as simple as 3D heat equations — will be greatly disappointed. In summary, these results highlight that auto-tuning is critically important for unlocking the performance potential across a diverse range of chip multiprocessors.

7. Acknowledgments

We would like to express our gratitude to IBM for access to their newest Cell blades, as well as Sun and NVIDIA for their machine donations. This work was supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231 and by NSF contract CNS-0325873.

References

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

[2] M. Gschwind, "Chip multiprocessing and the cell broadband engine," in *CF '06: Proceedings of the 3rd conference on Computing frontiers*, New York, NY, USA, 2006, pp. 1–8.

[3] *NVIDIA CUDA Programming Guide 1.1*, November 2007. [Online]. Available: http://www.nvidia.com/object/cuda_develop.html

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

[5] S. Sellappa and S. Chatterjee, "Cache-efficient multigrid algorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 115–133, 2004.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the Cell processor for scientific computing," in *Proceedings of the 3rd Conference on Computing Frontiers*, New York, NY, USA, 2006.

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

[13] 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.

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

[15] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. SC2007: High performance computing, networking, and storage conference*, 2007.