A Design Methodology for Domain-Optimized Power-Efficient Supercomputing

Marghoob Mohiyuddin[†⋆], Mark Murphy[⋆], Leonid Oliker[†], John Shalf[†], John Wawrzynek[⋆], Samuel Williams[†]

[†]*CRD/NERSC, Lawrence Berkeley National Laboratory Berkeley, CA 94720*
[⋆]*EECS Department, University of California at Berkeley, Berkeley, CA 94720*

ABSTRACT

As power has become the pre-eminent design constraint for future HPC systems, computational efficiency is being emphasized over simply peak performance. Recently, static benchmark codes have been used to find a power efficient architecture. Unfortunately, because compilers generate sub-optimal code, benchmark performance can be a poor indicator of the performance potential of architecture design points. Therefore, we present hardware/software co-tuning as a novel approach for system design, in which traditional architecture space exploration is tightly coupled with software auto-tuning for delivering substantial improvements in area and power efficiency. We demonstrate the proposed methodology by exploring the parameter space of a Tensilica-based multiprocessor running three of the most heavily used kernels in scientific computing, each with widely varying micro-architectural requirements: sparse matrix vector multiplication, stencil-based computations, and general matrix-matrix multiplication. Results demonstrate that co-tuning significantly improves hardware area and energy efficiency – a key driver for next generation of HPC system design.

1. INTRODUCTION

Energy efficiency is rapidly becoming the primary concern of all large-scale scientific computing facilities. According to power consumption data collected by the Top500 list [24], high-performance computing (HPC) systems draw on the order of 2–5 Megawatts (MW) of power to reach a petaflop of peak performance. Furthermore, current projections suggest that emerging multi-petaflop systems are expected to draw as much as 15 MW of power including cooling. Extrapolating the current trends, the Department of Energy (DOE) E3 [21] report predicts an exascale system would require 130 MW. At these levels, the cost of electricity will dwarf the procurement cost of the hardware systems; unless the energy efficiency of future large-scale systems increases dramatically, HPC will face a crisis in which the cost of running large scale systems is impractically high.

Our approach in this paper is inspired by embedded system design methodologies, which routinely employ specialized processors to meet demanding cost and power efficiency requirements. Leveraging design tools from embedded systems can dramatically reduce time-to-solution as well as non-recurring engineering (NRE) design and implementation cost of architecturally specialized systems. Building a System-on-Chip (SoC) from pre-verified parameterized core designs in the embedded space, such as the Tensilica approach, enables fully programmable solutions that offer more tractable design and verification costs compared to a full-custom logic design. For this reason, we use the Stanford Smart Memories [12], which is based on Tensilica cores, as the target architecture in this work. Given that the cost of powering HPC systems will soon dwarf design and procurement costs, energy efficiency will justify a larger investment in the original system design — thus necessitating approaches that can significantly decrease energy consumption.

General-purpose commodity microprocessors, which form the building blocks of most massively parallel systems, are grossly energy inefficient because they have been optimized for serial performance. This energy inefficiency has not been a concern for small-scale systems where the power budget is typically sufficient. However, energy efficiency becomes a concern for large-scale HPC systems, where a even few megawatts of power savings can make a dramatic difference in operating costs or even feasability. From the perspective of an application, energy efficiency is obtained by tailoring the code to the target machine, whereas from the perspective of a machine, energy efficiency comes by tailoring the machine to the target applications. Naturally, tailoring both the hardware and software to each other is expected to achieve better energy efficiency — this is the approach taken in this work.

The novelty of our proposed methodology, illustrated in Figure 1, is to incorporate extensive software tuning into an iterative process for system design. Due to the increasing diversity of target architectures, software auto-tuning is becoming the de-facto optimization technique to tailor applications to target machines. Hardware design space exploration is routinely performed to determine the right hardware design parameters for the target applications. Our co-tuning strategy integrates the two paradigms of hardware and software design exploration; we employ automatically tuned software to maximize the utilization of each potential architectural design point. The auto-tuning methodology achieves performance by searching over a large space of software implementations of an algorithm to find the best mapping to a microarchitecture [6]. Though our proposed approach may seem intuitive, this work is the first to quantify the potential benefits of co-tuning.

We demonstrate the effectiveness of our methodology using the sophisticated Stanford Smart Memories [12] simulator on three of the most heavily used kernels in scientific computing: sparse matrix vector multiplication (SpMV), stencil-based computation, and

SC09 November 14-20, 2009, Portland, Oregon, USA

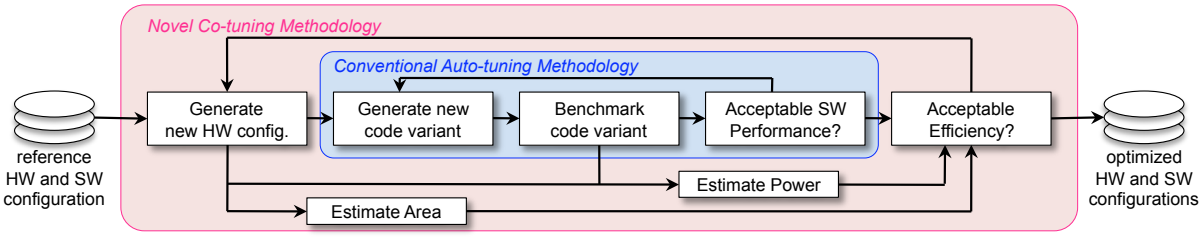


Figure 1: Our proposed approach for hardware/software co-tuning. In essence we have embedded a conventional auto-tuning framework within our novel-cotuning framework. As efficiency rather than peak performance is our metric of interest, we use models in conjunction with performance counters to to estimate area and power efficiency. The result is both a hardware configuration and a software implementation.

general matrix-matrix multiplication (SGEMM). Our experiments examine co-tuning advantages on isolated kernels, as well as multi-kernel application experiments. Overall results demonstrate that our co-tuning strategy can yield significant improvements in performance, power efficiency, and area efficiency when compared with the traditional approaches. We conclude that co-tuning can have a tremendous impact on future HPC system design.

2. RELATED WORK

Software auto-tuning is an emerging field, with optimization packages for several key scientific computing kernels, including FFTW [7], SPIRAL [16], OSKI [26], and ATLAS [28]. The success of the these examples has demonstrated that software auto-tuning is a practical method for portable high-performance scientific libraries. In addition to specific kernel optimization, recent work on the ROSE compiler project [19] enables auto-tuning individual loops to improve overall code performance. Work on the POET language [31] allows developers to define domain-specific optimizations for an application, thus simplifying the process of auto-tuning design.

Current work in system design treats hardware design space exploration (DSE) and software tuning separately. Approaches to DSE [10, 11, 14, 15] tune hardware parameters for benchmarks [2, 4, 30], with little [10] or no software tuning [11, 14, 15]. Intuitively, coupling DSE with software tuning should improve overall performance metrics — however, we are the first to study the benefits of such an approach. In a recent study closest to our work [20], DSE is performed for horizontally partitioned cache architectures and it is shown that including compiler (which targets a specific hardware configuration) in the DSE loop gives better results when compared to the traditional approach. However, we extend this idea to include both the compiler and the software auto-tuner in the DSE loop — the compiler incorporates hardware configuration specific knowledge, whereas the auto-tuner incorporates application/kernel specific knowledge to improve performance.

Finally we note that our study utilizes the Tensilica processor-based architecture, due to the significant power efficiency benefits offered by embedded cores. The practicality of HPC systems built using power-efficient embedded cores is borne out by the existence of IBM BlueGene/P [22] (using IBM PowerPC 450 cores), SiCortex [17] (using MIPS cores), and the Anton [9] molecular dynamics (MD) supercomputer (using Tensilica LX cores).

3. EVALUATED NUMERICAL KERNELS

Many high-performance computing applications spend a high fraction of their running time in a few, relatively small kernels. For purposes of the design methodology presented in this work, we examine three key kernels from scientific computing: matrix-matrix multiplication (SGEMM), PDEs on structured grids as implemented by a 7-point stencil (Stencil), and sparse-matrix vector multiplication (SpMV). All examined kernels are single-precision implementations due to the constraints of the Tensilica simulator (described in Section 4), and will be extended to double precision as the Tensilica platform evolves. In general, our design methodology could easily be applied to applications dominated by other kernel classes or in any other precision or data type.

In this section we provide the fundamental details of these kernels as well as the auto-tuning methodologies used to optimize their performance. Table 1 quantifies some key computational characteristics, including the total number of floating-point operations, the arithmetic intensity in flop:DRAM bytes, and the cache or local store capacity required to attain said arithmetic intensity. Superficially, SpMV and stencil have arithmetic intensities that are constant with respect to problem size whereas SGEMM has an arithmetic intensity that scales with cache block size and is limited only by cache capacity and computational capabilities. As such, one naturally expects SGEMM performance to exceed stencil, and stencil to exceed SpMV. One should also observe that cache-based and local store-based implementations will place different demands on their respective on-chip memories. We will now provide the details on these three kernels.

3.1 Dense Matrix Matrix Multiplication (SGEMM)

SGEMM (Single-precision General Matrix-Matrix Multiplication) is a critical dense linear algebra kernel. As a fundamental BLAS-3 routine, an efficient SGEMM implementation is crucial for efficient implementation of many linear algebra codes. For our purposes, the SGEMM kernel performs single-precision matrix-matrix multiplication with increment on square N×N matrices ($C \leftarrow A \cdot B + C$). Such a kernel is easily ammenable to cache blocking in which the matrix is decomposed into B×B cache blocks. As a result, SGEMM can attain a high computational intensity, and highly-tuned implementations usually achieve close to peak machine performance. Note that extensive optimization is necessary, as a naïve version sub-optimally exploits cache and register resources.

For this work we implement an SGEMM auto-tuner capable of exploiting local store based architectures and using a subset of the optimizations presented in previous studies [28]. However, to maximize performance we utilize the well-known ATLAS [28] code generator for the innermost kernel codes. We implemented blocking for cache and register file utilization, as well as loop unroll and jam. For cache-based architectures, we store matrices A and B in block-major format, and dynamically transpose A to enable unit-stride access.

Our auto-tuner implements a greedy algorithm and thus operates in two phases. It first determines the register blocking and

	SGEMM	Stencil	SpMV
FLOPs	$2{\cdot}N^3$	$8{\cdot}N^3$	$2{\cdot}NNZ$
flop:byte ratio	$< \frac{B}{6}$	<1.0	<0.5
Requisite LS/Cache	$12{\cdot}B^2$ (CC) $20{\cdot}B^2$ (LS)	$8{\cdot}XY$ (CC) $24{\cdot}XY$ (LS)	$<4{\cdot}N$

Table 1: Computational characteristics of three evaluated kernels (in single-precision). Cache/local store (LS) capacities are measured in bytes.

Architecture	cores per chip	Power per chip	Sustained MFlop/s per chip			Sustained PFlop/s with 10MW of chip power		
			DGEMM	Stencil	SpMV	DGEMM	Stencil	SpMV
Opteron	4	95W	32000	3580	1980	3.37	0.38	0.21
Blue Gene/P	4	16W	10200	520	590	6.38	0.33	0.37

Table 2: Performance of the double-precision implementations of our three key kernels on petascale computers. Note, power requirements are for the chip only assuming perfect scaling.

loop unrolling to optimize single-core performance and then determines the cache-blocking parameters for best multi-core performance. Due to the time constraints of the software-based simulation framework, we limit our dataset to matrices of dimension 512×512.

3.2 Stencil Arising from the Heat Equation PDE

A frequent approach to solving partial differential equations (PDE) is the iterative, explicit finite-difference method. Typically, it sweeps through the discretized space, usually out-of-cache, performing a linear combination of each point's nearest-neighbors — a *stencil*. Stencils can be used to implement a variety of PDE solvers, ranging from simple Jacobi iterations, to complex multi-grid and adaptive mesh refinement methods [3]. In this work, we examine performance of Jacobi's method to the single-precision 7-point 3D heat equation, naïvely expressed as triply nested loops ijk over:
$B[i,j,k] = C_0 \cdot A[i,j,k] + C_1 \cdot (A[i+1,j,k] + A[i-1,j,k] + A[i,j+1,k] + A[i,j-1,k] + A[i,j,k+1] + A[i,j,k-1])$.

The auto-tunner used in this work implements a subset of those described in previous investigations [5], which had proven to be extremely effective over a wide range of multi-core architectures. The work here focuses exclusively on optimizations that are relevant to the architectures within our design space: register blocking, array padding, and cache/local store blocking, including an implementation of the circular queue DMA blocking algorithm. We now briefly describe the implemented optimizations. Interested readers should refer to the prior work for more details [5].

Stencil register blocking consists of an unroll-and-jam in the X (unit-stride) and Y dimensions. This enables re-use of data in the register file, and decreases loop overheads. The best unrolling factors balance an increase in register-pressure against decreased L1 data cache bandwidth. Array padding consists of adding a small number of dummy cells at the end of each pencil (1D row of the 3D stencil arrays), and perturbs the aliasing pattern of the arrays in set-associative caches to decrease cache conflict misses. Such an optimization avoids the need for designs with highly associative caches. The cache-blocking optimization is an implementation of the Rivera-Tseng blocking algorithm [18]. We tile in the X (unit stride) and Y dimensions, and perform a loop-interchange to bring the Z-dimension (least unit stride) loop inside of the tiled loop nests, exploiting re-use between vertically adjacent planes. For cacheless, local store-based targets with software-managed DMA, the circular-queue technique of local store management [5] is used to implement the Rivera Tiling and schedule the DMAs to overlap memory accesses with computation (see Figure 2).

Our approach to auto-tuning the stencil code is designed to balance coverage of the search space against the amount of simulation time required. To that end, we implement a greedy algorithm that starts with the "innermost" optimizations and works its way outward. Thus, it begins by tuning the register-block size, then tunes for the optimal array padding, and tunes for the optimal cache-block size last.

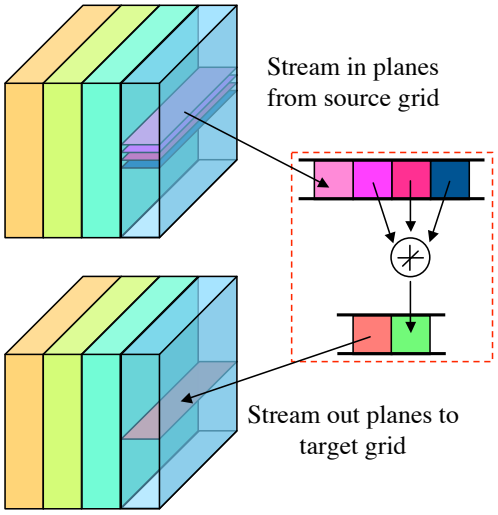


Figure 2: Visualization of stencil circular queue optimization for local store systems [5].

3.3 Sparse Matrix-Vector Multiplication (SpMV)

SpMV dominates the performance of diverse applications in scientific and engineering computing, economic modeling, information retrieval, among others. Reference SpMV versions perform very poorly, running at 10% or less of machine peak on single-core cache-based microprocessor-based systems [25]. In this work, we use the single-precision SpMV operation ($y \leftarrow Ax$) in which a sparse matrix A is multiplied by a dense source vector x. This produces the dense destination vector y. A is sparse; most of its entries are zero, and are neither stored in memory or used in the computation. Typically, A is represented in a compressed sparse row (CSR) data structure. SpMV has a low arithmetic intensity as each $A_{i,j}$ is used exactly once per SpMV to perform one multiply-accumulate operation. Moreover, to reference $A_{i,j}$, CSR demands an additional 4 bytes of meta data for every nonzero.

Our SpMV auto-tuning optimization approach utilizes previously established techniques [25, 29], which we describe only briefly. SpMV performance suffers primarily from large instruction and storage overheads for sparse matrix data structures and from irregular memory access patterns. Optimizations focus on selecting a compact data structure to represent the matrix and code transformations that exploit both the structure of the sparse matrix and the

spyplot	Name	Dimensions	Nonzeros (nnz/row)	Description
	Dense	2K x 2K	4.0M (2K)	Dense matrix in sparse format
	FEM / Spheres	83K x 83K	6.0M (72)	FEM concentric spheres
	FEM / Cantilever	62K x 62K	4.0M (65)	FEM cantilever
	Wind Tunnel	218K x 218K	11.6M (53)	Pressurized wind tunnel
	QCD	49K x 49K	1.90M (39)	Quark propagators (QCD/LGT)
	FEM/Ship	141K x 141K	3.98M (28)	FEM Ship section/detail
	Epidemiology	526K x 526K	2.1M (4)	2D Markov model of epidemic
	Circuit	171K x 171K	959K (6)	Motorola circuit simulation

Figure 3: Overview of matrices used for SpMV evaluation, representing a variety of computational structures.

underlying machine architecture.

This work considers thread, cache, and register blocking, and software prefetching. On local store-based architectures, cache blocking is called local store blocking, and prefetching becomes DMA. Thread blocking is slightly distinguished from parallelization in that the matrix is partitioned into equal-sized sub-matrices that can be individually allocated, padded, and optimized. Cache blocking exploits re-use of the source vector by tiling accesses to columns of the matrix. The tiling transformation must be implemented on local store architectures with software-managed DMA to guarantee correctness rather than to improve performance. Our software prefetching and DMA optimizations load only the nonzero values and column index arrays. For local store-based architectures, we must explicitly load all the referenced source-vector elements, as well as the matrix data structure.

Our SpMV auto-tuner is a port of the cache-based and Cell/DMA code described in [29]. However, in this work, as we are tuning hardware to maximize efficiency, we cannot make the assumption that our target architecture (see Section 4) will be heavily memory-bound. Thus, we employ a more complex heuristic [25] that attempts to balance the superfluous memory traffic associated with filling zeros when register blocking with a register block's inherently higher raw flop rate.

SpMV performance is heavily dependent upon the nonzero structure (sparsity pattern) of the sparse matrix. Thus, it is necessary to evaluate SpMV implementations against matrices drawn from realistic applications. We conducted experiments on seven sparse matrices from a wide variety of actual applications, including finite element method-based modeling, circuit simulation, and linear programming. An overview of their salient characteristics appears in Figure 3. We also evaluate a dense matrix stored in sparse format as a performance upper-bound. These matrices cover a range of properties relevant to SpMV performance: matrix dimension, non-zeros per row, the existence of dense block substructures, and degree of non-zero concentration near the diagonal. Previous work presented an overview of their salient characteristics [29]. Due to space limitations, we present data only on the median performance.

3.4 Kernel Performance on Petscale Systems

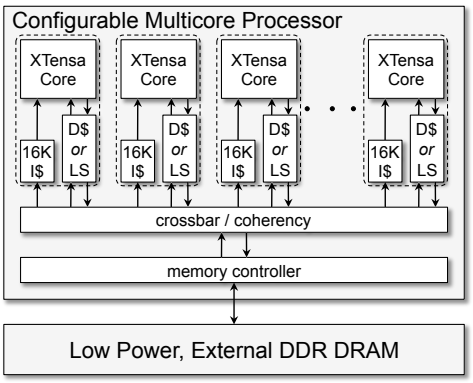
To provide context, Table 2 presents the performance of the double-precision implementations of DGEMM, the 7-pt stencil, and SpMV on two petascale-class architectures: the XT4 and BlueGene/P. The XT4 is built from commodity general-purpose quad-core Opteron processors, while BlueGene/P is built from customized quad-core embedded PPC450 processors. Chip performance is shown for each of the studied kernels. Additionally, as a point of reference, we extrapolate the maximum attainable PFlop/s performance (assuming perfect scaling) given a 10MW *chip* power budget (clearly the overall system power will exceed the chip power requirements). Note that the chip power requirements differ by about a factor of six with the more power hungry chip delivering superior per node performance. However, in a power-conscious or constrained world, the BlueGene/P system would deliver about twice the performance on a fixed 10MW chip power budget. It is, therefore, clear that power efficiency will have tremendous impact on the attainable performance of next-generation supercomputing facilities. Our co-tuning work, as detailed in the remaining of this study, brings forth a methodology that can significantly improve a system's power (or area) efficiency.

4. EXPERIMENTAL PLATFORM AND DESIGN SPACE EXPLORATION

Our study is heavily geared towards producing area- and power-efficient designs. As such, we embrace SiCortex's and IBM's decision to utilize embedded processors for HPC systems. In this work, we used the Tensilica XTensa core primarily due to its flexibility, ease of system integration, configurability of micro-architectural resources, and of course, XTensa's target market is energy-efficient embedded systems.

As a testbed for the evaluation of the myriad of different hardware configurations, our work utilizes the Smart Memories [12] (SM) reconfigurable simulator developed at Stanford for general-purpose multi-core research. We present and exploit only the aspects of the architecture relevant to our design space, and refer the reader to cited works for other details. Figure 4 generalizes our multicore architecture. The SM simulator was designed to simulate a wide variety of on-chip memory hierarchies for multi-core processor, and utilizes cycle-accurate simulator of the Tensilica XTensa processor for performance modeling of the individual cores. The goal of SM is functional emulation and accurate performance estimation, subject to the throughput and latency specifications of the system configuration. Our power and area estimation methodology is presented in Section 5. In this work, we use the configurability of the simulator to explore an enumerated design space. Since the experiments are conducted in a software simulation environment, we have pruned this design space to reduce the amount of compute time needed. Future work will explore the space faster by using FPGA-based hardware emulation [27].

Previous studies of numerical algorithms have shown that cache hierarchy, memory system, and on-chip computational resources are crucial system design parameters for HPC architectures. Figure 4 enumerates our hardware design space. The core architecture is a fixed 500MHz single-issue, in-order Tensilica XTensa core with a private 16KB instruction cache. The 500MHz rate is useful, as it allows the Tensilica toolchain to provide us with accurate



(a)

Component		Parameters	Explored Configs
cores	**in-order XTensa core**	Issue width	**single-issue**
		Frequency	**500 MHz**
		Number of Cores	1, 4, **16**
		Inst. Cache (per core)	**16 KB**
Memory Hierarchy	**Coherent Data Caches**	Capacity (per core)	16, 32, 64, **128 KB**
		Associativity	**4 way**
		Line size	**64 Bytes**
	Local Store	Capacity (per core)	16, 32, 64, 128 KB
	External DRAM	Bandwidth	0.8, 1.6, **3.2 GB/s**
		Latency	**100 core cycles**

(b)

Figure 4: Left: Restricted SmartMemories architecture for some number of cores. Each core has a private instruction cache and either a private data cache or a private local store. Right: Hardware parameters explored in co-tuning architectural-space exploration. The parameters corresponding to baseline (untuned) hardware configuration are in boldface. Note that data cache and local store designs are mutually exclusive.

power and area projections. We vary the number of cores from one to 16 in powers of four. The memory hierarchy is divided into two parts: on-chip memories, and off-chip memory. On-chip memories are either a private coherent caches (CC) per core or a private disjoint local stores (LS) per core. We fixed cache associatvity as 4-way and line size is 64 bytes. For caches and local stores we explore four different capacities. All cores use the same design — there is no heterogeneity. Off-chip memory is abstracted as a uniform memory access DRAM running at one of three different possible bandwidths.

5. EVALUATION METRICS

Our area of focus is parallelized scientific applications running on large-scale, energy-efficient, high-performance systems consisting of tens of thousands, if not millions, of individual processing elements. Obtaining enough power for such systems can obviously be an impediment to their adoption. Thus, achieving high performance when designing such machines is less dependent on maximizing each node's performance, but rather on maximizing each node's power efficiency. Moreover, large silicon surface area can be expensive both from a fabrication cost, and also in its impact on mean time between failures (MTBF). Thus, our design methodology focuses on two key optimization metrics: *power efficiency* — the ratio of attained MFlop/s per chip to chip power, and *area efficiency* — the ratio of attained MFlop/s per chip to chip area.

Given these metrics, one can impose either a per-node or per-supercomputer chip power (or area) budget and estimate the resultant attainable performance: minimum of area efficiency $\times$ chip area budget and power efficiency $\times$ chip power budget. Systems with limited power budgets should be selected based on power efficiency, whereas systems with limited silicon budgets should be selected based on area efficiency. Identifying the trade-off between the two allows a designer to balance the system acquisition costs, system power requirements, and system up time. It is important to note that further gains in power efficiency can be realized by optimizing all the system components (in addition to chip power) — this will be focus of future investigations.

5.1 Modeling Chip Power

The power estimation is based on the model used in [10] by weighting a number of key architectural events counted by the software simulator with appropriately modeled or derived energies weighted by the total execution time. Energy for events originating in the cores are derived using the energy estimates from the Tensilica tools. The effect of clock gating is taken into account by a reduced power consumption when the core is stalled (assumed to be 10% of peak power). The dynamic energy for the caches and local stores is modeled on a per transaction basis using a CACTI 5 [23] model. The external DRAM energy is modeled using the current profiles from the Micron datasheets [13] for a 256 MB DDR2-400 memory module. Given the low power nature of the Tensilica cores, DRAM power is substantial. On-chip network energy is calculated based on the total on-chip network traffic and using the scaled energy numbers from [10]. Finally, leakage power is assumed to be 15% of peak power for any configuration. Although, every software implementation for a given hardware design will yield different power estimates, this model allows us to explore variations in the constants without having to resimulate the design.

5.2 Modeling Chip Area

The area of a given processor configuration is an important metric due to its effect on the end cost of fabricating and packaging the circuit. To this end, we model the hardware configuration area within the design space, assuming 65nm node technology for the core chip. Core area estimates are provided by the Tensilica toolchain, while CACTI 5 [23] was used to model cache or local store area. To mitigate the effect of area on yield, we assume sparing is used for increasing yield — one spare core is assumed for chips with up to eight cores, and two spare cores are assumed for chips with 16 cores. Each Tenscilica cores is extremely small when

compared with modern high-performance microprocessors — less than 0.5 mm^2. As such, we expect such a sparing strategy to have very high die yields and a yield percentage that is effectively independent of the chip area. In essence, the resultant yield-adjusted chip costs should be roughly linear with core chip area. We assume that the on-chip network and clocking add another 20% to the core chip area. Finally, we assume that the memory interface adds a constant 35 mm^2 to the chip area regardless of the frequency we clock the DIMMs at—we assume that DRAM area costs 3× less when compared to core chip area. For comparison, each core is less than 0.5 mm^2, and each 128 KB cache is less than 1 mm^2. As such, there is a clear economy of scale by incorporating many cores.

6. EXPERIMENTAL RESULTS

Before delving into the experimental results, we briefly reiterate our novel co-tuning approach shown in Figure 1: for each of the three kernels, and for each processor configuration in our search space, the kernel is auto-tuned on that hardware configuration to find the software implementation that maximizes performance. Therefore, given a hardware configuration, we always report data corresponding to the best performance achieved by the auto-tuner on it. Note that while our auto-tuners heuristically prune their own search spaces of tuned software implementations, we explore the hardware configuration space exhaustively by running the auto-tuners on all the configurations within our design space. An application of co-tuning for a real system, however, would use a more efficient search strategy to explore a much larger hardware design space. For the purpose of this paper, though, exhaustive search suffices as the hardware design space is small and serves well to illustrate the effectiveness of co-tuning.

We now quantify the effectiveness of our co-tuning methodology for a variety of hardware and software optimization strategies on each of the three numerical kernels. We commence with a study of the relationship between architectural configuration and attained performance. Next, we measure the potential improvements in kernel power and area efficiency using our co-tuned design space exploration. Finally, we analyze the benefit of co-tuning for applications dominated by a varying mix of these three kernels.

6.1 Performance of Design Parameters

We now explore the per kernel performance response to changes in both software optimization and processor configuration. Doing so provides insights into the inherent hardware requirements and attainable efficiencies for each kernel as well as quantifying the importance of the instantiation of the auto-tuning component within the co-tuner. Not only do these simulation results, shown in Figure 5, follow our intuitions regarding the expected performance of our kernels under varying architectural conditions, they also serve to validate the underlying simulation environment.

Auto-tuning.

Figures 5(a–c) show the performance benefit of auto-tuning for SpMV, Stencil, and SGEMM using a fixed memory bandwidth of 1.6 GB/s and either a 64 KB data cache or a 64 KB local store for 1, 4, or 16 cores. The stacked bars indicate the improvement of our software optimizations.

Observe that due to SpMV's constant and low arithmetic intensity, with enough cores, SpMV performance plateaus for both the tuned and untuned code versions using either caches or local stores. In effect, the changes in processor configuration transitioned its behavior from compute-bound to memory-bound. Note that for smaller concurrencies, the untuned DMA-based implementations outperform the cache coherent versions by a factor of 2×. Such a situation arises because the DMA version utilizes block transfers, which represent a means of more easily satisfying Little's Law [1] and mandates a reuse-friendy blocked implementation for correctness. Nevertheless, the SpMV auto-tuner provides significant benefit even on bandwidth-limited configurations. This class of SpMV auto-tuner attempts to both minimize memory traffic and express more instruction-level-parallelism. The results reaffirm the significant impact of auto-tuning shown previously shown on numerous multicore architectures [5, 6, 29].

The stencil code is both moderately more arithmetically intense than SpMV, and also contains more regularity in its memory access pattern. Figure 5b demonstrates that, relative to SpMV, the higher arithmetic intensity forestalls the advent of a memory-bound processor configuration. Thus, as applications shift their focus from SpMV to stencil-like kernels, they may readily exploit more cores. Most interesting, the quad-core local store version attains the performance comparable to the 16-core cache-based implementation. In effect, DMA transfers eliminate superfluous write allocate traffic and express more memory-level parallelism. The incorporation of effective prefetching into the cache-based stencil auto-tuner might mitigate the latter. Finally, the tuned local store stencil code can utilize a large portion (∼75%) of memory system with four cores; hence the performance improvement is limited to about 30% when quadrupling the number of cores to 16. With 16-cores, even the untuned DMA-based code is nearly able to saturate the memory system.

SGEMM has a high arithmetic intensity arithmetic hierarchally limited by the register and cache capacities. Thus, it alone among our kernels is capable of exploiting increasing numbers of cores (and cache capacities). Figure 5c shows the performance of SGEMM scales linearly with the number of cores, for any processor configuration both with and without auto-tuning. This does not imply all configurations delivered the same performance or benefits. To the contrary, auto-tuning was essential on cache-based architectures; improving performance by 64×, but only provided a moderate speedup on the already well-blocked local store implementations. Moreover, the local store configuration consistently outperformed the cache configurations. The naïve code incurs significant cache conflict misses for large matrices, especially when the matrix dimensions are powers of 2 — the common case in our experiments. Furthermore, the latency penalty of a cache miss is high due to the absence of an L2 cache in our configurations. Due to SGEMM's hierarchal arithmetic intensity, the effects of inner-loop code generation and blocking for data reuse are extremely important. In contrast to the stencil and SpMV codes, even our most highly optimized SGEMM implementations are not significantly limited by memory bandwidth.

On-chip Memory Capacity.

Figures 5(d–f) quantify performance response to changes in core count and per-core memory capacity for the auto-tuned codes. We show both the cache-based and DMA-based codes for each of 1, 4, and 16 cores. Although the cache-based configurations can be more sensitive to cache size compared with the local store versions — since it is harder to control blocking and data movement via scalar loads and stores — performance is relatively insensitive to cache and local-memory sizes. SpMV performance hardly changes at all, as the smallest cache size is enough to exploit re-use of the two vectors. The cache-based stencil code sees about 60% performance improvement as the cache size increase from 16 KB to 64 KB. However, the explicitly-blocked, DMA-based stencil code can exploit nearly all temporal locality using the smallest local memory. SGEMM on 16-core systems benefits from increased cache and lo-

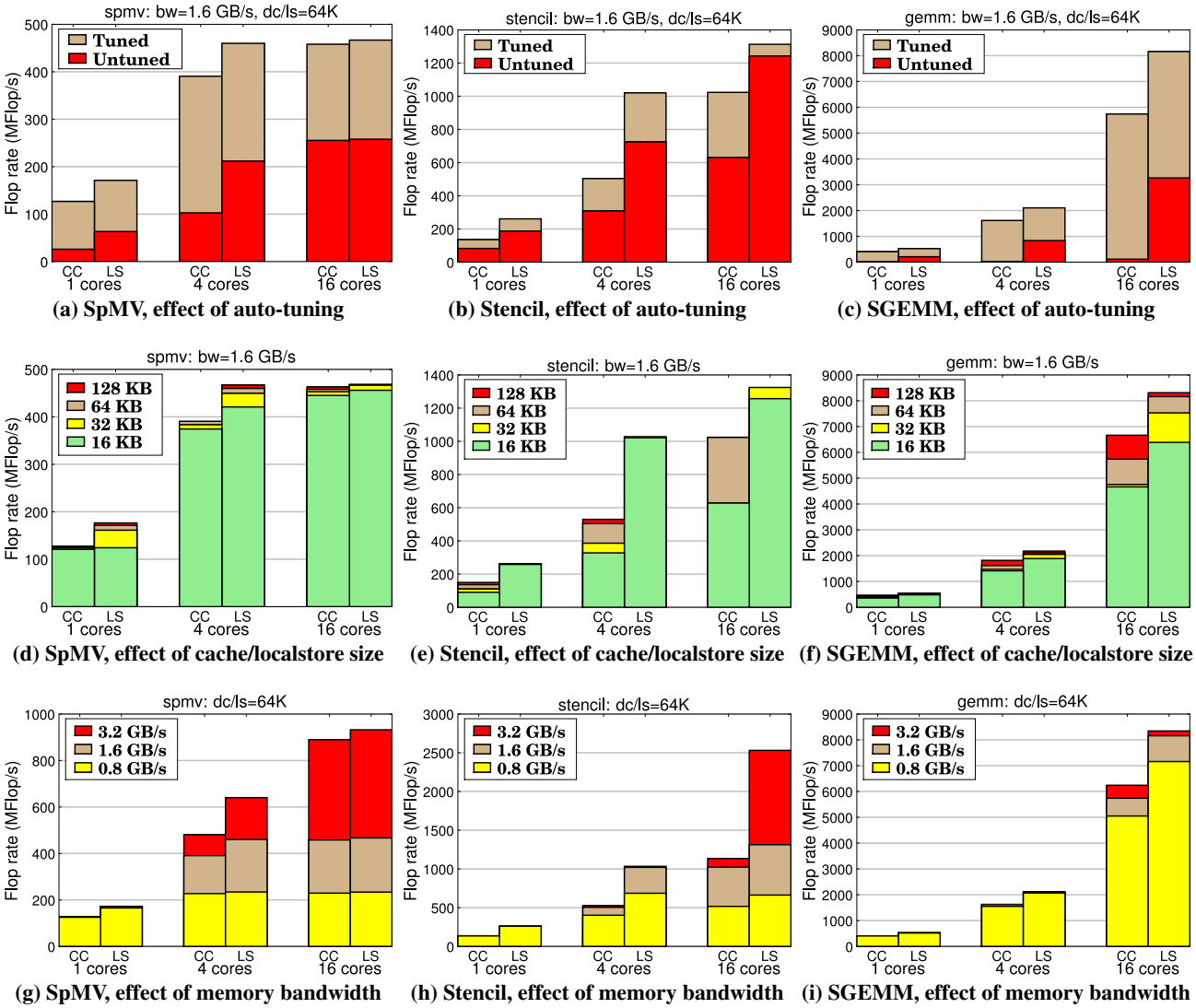


Figure 5: The interplay between processor configuration and auto-tuning for the SpMV, Stencil, and SGEMM kernels. Note: 'LS' indicates DMA-managed local store architectures, and 'CC' indicates coherent-cache systems.

cal memory sizes due to memory bandwidth contention; the larger caches enable larger block sizes and reduce pressure on memory bandwidth — *i.e.* higher arithmetic intensity.

Memory Bandwidth.

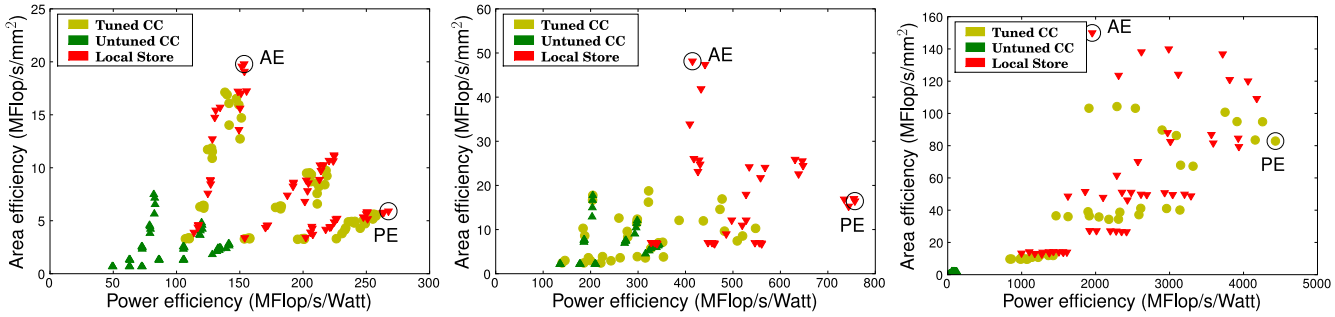
Figures 5(g–i) show performance as the processors' memory bandwidth is changed. Clearly, for SpMV and stencil, increasing the number of cores is only viable when the memory bandwidth is similarly increased since they are ultimately memory limited. This effect is less pronounced for Stencil due to its higher arithmetic intensity. SGEMM, on the other hand, only begins to show the limitations of memory bandwidth with 16 cores.

6.2 Tuning for Power and Area Efficiency

Having established raw performance characteristics, we now examine the power and area efficiency of our methodology. Figure 6 plots these efficiency metrics (as defined in Section 5) for our three test kernels. Each point in the scatter plot represents a unique processor configuration, with yellow circles, green triangles, and red triangles corresponding to auto-tuned cache, untuned cache, and (either auto-tuned or untuned) local store versions respectively. Additionally, a circle is drawn to highlight the configurations with the best power or area efficiencies.

These figures serve to demonstrate the extreme variation in both efficiency metrics spanned by the design points within our configuration search space. Figure 6a shows that a poor choice of hardware can result in as much as a 3× degradation in power efficiency for SpMV (MFlops/s per Watt), whether software tuning is employed or not. Figure 6b shows that for stencil, the difference is nearly 8×. For SGEMM in Figure 6c, this difference is nearly two orders of magnitude! Since the operational cost and performance ceiling of future HPC facilities are limited by the power consumption of compute resources, these results quantify the potential impact of an energy-efficient design and hold the promise of reducing petascale system power by several megawatts.

We now measure the potential effectiveness of our combined hardware/software tuning methodology. Performance is explored



(a) SpMV area vs. power efficiency **(b) Stencil area vs. power efficiency** **(c) SGEMM area vs. power efficiency**

Figure 6: Area efficiency vs. power efficiency for each of the three Kernels. 'AE' and 'PE' denote the most area- and power-efficient configurations respectively.

in the context of four configurations: untuned software on the fastest processor configuration, auto-tuned software on the fastest processor configuration, tuned hardware running untuned (fixed) software, and co-tuned hardware/software. This serves to differentiate the efficiency gains from tuning software and hardware individually from the efficiency gains of co-tuning.

Out-of-the box: Untuned Software on the Fastest Processor Configuration.

Our lowest baseline comparison is the conventional wisdom strategy of choosing a system design by using the most powerful hardware configuration. We do not tune the software running on these processors. The most powerful hardware configuration within our search space is a coherent-cache chip multi-processor with 16 cores, 128 KB of L1 data cache per core, and 3.2 GB/s of main memory bandwidth. While local store architectures generally provide better performance, it is impossible to produce un-tuned codes to utilize them. As this comparison represents putting essentially no effort into the system design, it is highly unlikely to be a viable power- or area-efficient solution. Rather, we present it as a point of comparison to illustrate how much efficiency our coupled hardware/software design space exploration provides. Table 3 presents an overview of the optimal power and area efficiency data for each optimization strategy (including the improvement impact of co-tuning), starting with this baseline configuration, shown in the fourth column. Observe that our co-tuning methodology would deliver 3.2–80× better power and area efficiencies for our evaluated kernels.

Auto-Tuned Software on the Fastest Processor Configuration.

In order to differentiate the efficiency that the current state-of-the-art provides, we present the result of auto-tuned software on the fastest hardware. This combination is analogous to building a system from high-performance commodity cores and utilizing auto-tuning for software optimization — an increasingly common solution. The hardware provides as much of each architectural resource as our design space allows, but without having been specifically tailored to any specific kernel. The auto-tuned kernels exploit those resource to maximize performance. The fifth column (auto-tuned SW, fastest HW) of Table 3 shows the optimal power and area efficiency produced by this approach.

Since the hardware configuration is the same as the untuned SW on fastest HW, the efficiency gains correspond roughly to improvements in attained floating-point performance through auto-tuning. These ratios are different for power and area efficiency, since power depends on the activities of the various architectural resources, while area depends only on their physical quantity. Comparing the fourth and fifth columns of Table 3 shows that SGEMM's auto-tuning achieves an impressive 54× and 53× improvement in power and area efficiency (respectively), due to the enormous performance impact of auto-tuning on this kernel. For Stencil and SpMV, the improvement is not as spectacular, but nonetheless substantial: Auto-tuning improves Stencil's power efficiency by 1.5× and its area efficiency by 1.2×, while SpMV benefits by 1.8× in power efficiency, and 2.2× in area efficiency. These results reiterate the conclusions of prior works [5, 6, 29] that auto-tuned codes can outperform compiler-only optimizations by a wide margin.

Untuned software, Tuned Hardware.

We now examine the effect of hardware design space exploration alone, without the benefit of software auto-tuning. Note that we omit local store-based configurations in the hardware tuning design space for this case. This is because our so-called "untuned" kernels on local store-based configurations are cognizant of both the local store capacity, as well as the locality inherent in the algorithms. A truly untuned (architecturally and algorithmically agnostic) local store code would doubtlessly achieve significantly lower performance. The green triangles in the scatter plots in Figure 6 represent the range of efficiencies that hardware-only tuning achieves, while the sixth column (untuned SW, tuned HW) in Table 3 shows the efficiencies of the Pareto-optimal hardware configurations.

Looking at the sixth column of Table 3 shows simply tuning over the hardware space improves both power and area efficiency for all our kernels. SpMV, stencil and SGEMM achieve power efficiency improvements of approximately 1.7×, and area efficiency gains of 1.3×, 1.4×, and 1.1× (respectively), when compared the untuned SW/HW. Note that since we are optimizing for power and area efficiency, the attained floating-point performance is lower compared with the untuned SW/HW case. Examining the fifth and sixth columns of Table 3 shows that even after searching over the hardware design space, we can still under-perform even on the most powerful hardware configuration, if auto-tuning software is not employed. This difference is quite dramatic for SGEMM (30× and 47× lower power and area efficiency) because it benefits tremendously from tuning, especially for the specific matrix dimensions used in our experiments.

Hardware/Software Co-Tuning.

Our hardware/software co-tuning methodology performs soft-

Design Objective	Co-Tuned Kernel	Metric	Untuned SW Fastest HW	Auto-Tuned SW Fastest HW	Untuned SW Tuned HW*	Co-Tuned SW/HW
Power Efficiency	SpMV	MFlop/s	397	895	127	229
		Power (W)	4.8	6.0	0.9	0.9
		Power Efficiency	82.4	150.2	141.6	267.5
		Co-Tuning Advantage	**3.2x**	**1.7x**	**1.9x**	—
	Stencil	MFlop/s	906	1139	262	686
		Power (W)	4.5	3.5	0.8	0.9
		Power Efficiency	203.2	321.9	344.5	756.9
		Co-Tuning Advantage	**3.7x**	**2.4x**	**2.2x**	—
	SGEMM	MFlop/s	132	7079	122	5823
		Power (W)	1.9	1.9	1.0	1.3
		Power Efficiency	68.7	3750.5	124.7	4431.4
		Co-Tuning Advantage	**65x**	**1.2x**	**36x**	—

Design Objective	Co-Tuned Kernel	Metric	Untuned SW Fastest HW	Auto-Tuned SW Fastest HW	Untuned SW Tuned HW*	Co-Tuned SW/HW
Area Efficiency	SpMV	MFlop/s	397	895	390	897
		Area (mm^2)	70.3	70.3	52.0	45.3
		Area Efficiency	5.8	12.7	7.5	19.8
		Co-Tuning Advantage	**3.5x**	**1.6x**	**2.6x**	—
	Stencil	MFlop/s	906	1139	923	2502
		Area (mm^2)	70.3	70.3	52.0	52.0
		Area Efficiency	12.9	16.2	17.9	48.1
		Co-Tuning Advantage	**3.7x**	**3x**	**2.7x**	—
	SGEMM	MFlop/s	132	7079	110	8173
		Area (mm^2)	70.3	70.3	52.0	55.0
		Area Efficiency	1.9	100.7	2.1	149.9
		Co-Tuning Advantage	**80x**	**1.5x**	**70x**	—

Table 3: Summary of optimal power-efficiency (in MFlops/s/Watt) and area-efficiency (in MFlops/s/mm^2) data (and relative improvement of co-tuning) for each optimization configuration.* The hardware configuration space for 'Untuned SW, Tuned HW' only includes coherent-cache based configurations.

ware auto-tuning for each potential point in the hardware design space, and thus represents a more complete coverage of the overall system design space than the other examined approaches. Each hardware design point is evaluated with a more complete picture of its potential for performance and efficiency. The last column in Table 3 shows the power and area efficiency of the Pareto-optimal configurations using the co-tuning methodology. Results show that this approach yields significant improvements in power and area efficiency when compared to the three previously discussed configurations (as shown in the parenthesized values of each column).

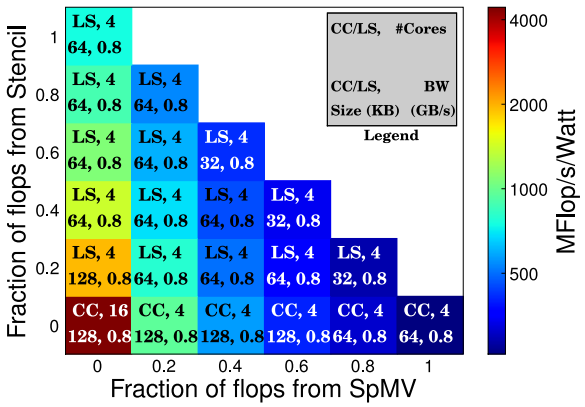
It is interesting to compare co-tuning (seventh column) with only software-based (fifth column) or hardware-based (sixth column) tuning. Given the tremendous benefits of software tuning, it is not surprising to see that co-tuning outperforms the hardware-only tuning approach. This difference is particularly dramatic for SGEMM — 36× and 70× in power and area efficiency respectively — where the untuned code performs quite poorly. Additionally, co-tuning gains for stencil and SpMV range from 1.9×–2.7×. Comparison of the co-tuning versus the software-only tuning approach shows that even after fully optimizing the code on the fastest hardware, there is still significant room for efficiency improvements. Notably, the power efficiency gains of co-tuning for SpMV, SGEMM, and stencil are 1.7×, 2.4×, and 1.2× respectively, whereas, the area efficiency improvements are 1.6×, 3×, and 1.5× respectively. These are the first results to quantify the intuitive notion that ignoring either hardware or software optimization while performing system design will naturally lead to suboptimal solutions. Finally, we also note that each individual kernel can have different optimal hardware configurations which is evident by the different areas for the best area efficiency configurations.

6.3 Co-Tuning for Multiple-Kernel Applications

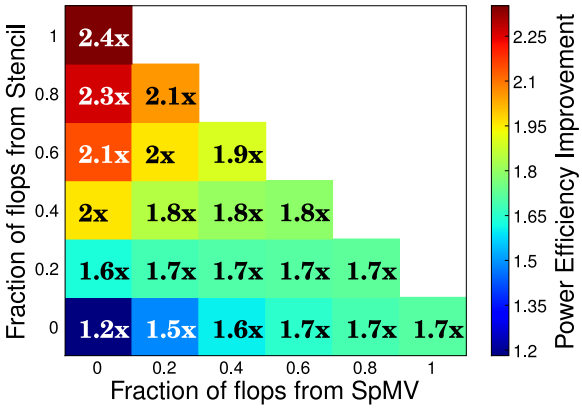
No one individual kernel can give a complete picture of an entire application's performance on a given system. Realistic large-scale scientific applications consist of multiple subsystems that solve different parts of the overall problem. We therefore approximate the effect of co-tuning on a multi-kernel application by combining the results from Section 6.2.

We thus construct co-tuning results for the set of kernels by taking a weighted mean of their tuned performance data on each hardware configuration. Given that each individual kernel contributes some fraction of the floating-point operations for an entire application, we sum these kernel contributions via a weighted harmonic mean. This basic strategy assumes that interaction between kernels does not have significant impact on whole-application performance. Although this is clearly a simplifying assumption, it is nonetheless an important first step toward quantifying the potential impact of the co-tuning methodology to full-scale applications. Expanding this approach will be the focus of future work.

Figure 7a plots the power efficiency of the best co-tuned hardware configuration relative to the fractions of floating-point operations contributed by the three kernels, where the individual contributions always sum to one. We present these data by defining the x-axis and y-axis of Figure 7a as the fractional contribution of



(a) Co-tuned weighted power efficiency showing optimal configuration: CC/LS, core count, CC/LS size (KB), memory BW (GB/s).



(b) Weighted improvements in power efficiency of co-tuning versus untuned-hardware with tuned-software approach.

Figure 7: Co-tuning for multiple kernels, using a 3D graph with the fractional contribution of SPMV on the x-axis, stencil on the y-axis, and SGEMM on the implicit z-axis. Each square in (a) depicts the HW parameters of the corresponding square in (b). The sum of the three kernels' flops contributions (x-,y-, and z-axis) always adds up to one.

Stencil and SpMV to an application's computation, while the remaining fractional portion represents the contribution of SGEMM (on the implicit z-axis). Therefore, the lower-left corner of the plot represents optimized hardware configurations for applications consisting entirely of SGEMM-like dense linear algebra algorithms. Similarly, the lower-right corner represents applications consisting entirely of SpMV-like sparse linear algebra, while and the upper left corner is stencil-based grid computations. As the most power efficient configuration differs for each mix of kernels, we annotate Figure 7a with the parameters of the best configuration: CC/LS, core-count (1-16), CC/LS size (16K-128K), and memory bandwidth (0.8-3.2 GB/s). Results show the variety of co-tuning architectural solutions and corresponding power efficiencies based on a given application's underlying characteristics.

Figure 7b plots the power efficiency improvements of the co-tuned systems for each kernel mix compared with the untuned-software tuned-hardware approach (described in Section 6.2). This approach most closely resembles prior work in automated system design, which have hitherto not included extensive software optimization. Recall, that we only consider coherent-cache based configurations for untuned-software base case (see Section 6.2). Results show that co-tuning results in power efficiency gains ranging between 1.2×–2.4× depending on each kernels contributions. A similar analysis for area efficiency (not shown), demonstrates improvements varying from 1.6× to 3× (as seen in Table 3). Overall this approach points to the potential of applying our co-tuning methodology to more complex, multi-algorithmic applications.

7. CONCLUSIONS AND FUTURE WORK

Power efficiency is rapidly becoming the primary concern for HPC system design. Conventionally designed ultra-scale platforms constructed with the conventional-wisdom approach based on using commodity server-oriented processors, will draw tens to hundreds of Megawatts — making the cost of powering these machines impractically high. Therefore, it is critical to develop design tools and technologies that improve the power efficiencies of future high-end systems.

We have proposed a novel co-tuning methodology — traditional architecture space exploration is tightly coupled with software auto-tuning — for high-performance system design, and demonstrated that it provides substantial efficiency benefits by customizing the system's architecture to software and software to the system's architecture. Our study applies this approach to a multi-core processor design with three heavily used kernels from scientific applications spanning a wide variety of computational characteristics. Based on the optimization results for the individual kernels, we demonstrate power and area efficiency gains of 1.2–2.4× and 1.5–3× respectively, due to co-tuning — when compared to using auto-tuned software on the fastest, embedded processor configuration. Additionally, we show that these improvements can also be attained in multi-kernel application environments. As highlighted in Table 2, this increased efficiency can translate into hundreds of Teraflops, if not Petaflops, of additional performance for next-generation power-constrained HPC systems.

Building platforms from pre-verified parameterized core designs in the embedded space enables programmability and accelerated system design compared to a full-custom logic design, while providing higher efficiencies than general purpose processors tailored for serial performance. Furthermore, our hardware/software co-tuning methodology is a tool for assisting and automating the optimization of programmable HPC systems for energy efficiency. Tools for automatic design space exploration in the context of ad-hoc architectures do not exist, and the design space is intractably large. However, basing architectures on programmable multi-core processors constrains the design space, making the search space tractable and verification costs reasonable — as the same core can be replicated millions of times.

Future work will examine more complex architectural designs that can potentially improve power efficiency such as VLIW, SIMD, vector, streaming and hardware multi-threading. Additionally, we plan to explore more algorithmic techniques as well as their interactions and potential cross-kernel optimizations. We also note that the search space in our study is primarily limited by the software-based architectural simulators. Each auto-tuning search produces hundreds of individual kernel implementations that must be executed on all 72 hardware configurations explored in our study—requiring thousands of CPU-hours of simulation time. Our ongoing work is therefore leveraging RAMP [27] to optimize architectural configurations by using FPGA-based hardware emulation to accelerate the exploration process. Finally, we plan to incorporate

intelligent pruning approaches [8] into our methodology to accelerate searching of the co-tuning design space.

Our proposed co-tuning strategy offers a promising trade-off between the additional design cost of architectural customization and the portability and programmability of off-the-shelf microprocessors. Moreover, existing toolchains of companies like Tensilica enable a large space of hardware configurations, and the evolving maturity of auto-tuners for scientific kernels provides the ability to extract near-peak performance from these designs. Overall, this approach can provide a quantum leap in hardware utilization and energy efficiency, the primary metrics driving the design of the next-generation HPC systems.

8. ACKNOWLEDGMENTS

The authors kindly thank the Stanford Smart Memories team for their insightful comments and generous access to their simulation resources. All authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. Authors from the University of California Berkeley were supported by Microsoft (Award #024263), Intel (Award #024894), and by matching funding by U.C. Discovery (Award #DIG07-10227). In addition, we are grateful to Forschungszentrum Jülich for access to their Bluegene/P.

9. REFERENCES

[1] D. Bailey. Little's Law and High-Performance Computing, 1997.

[2] D. Bailey, E. Barszcz, J. Barton, et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA, 1994.

[3] M. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, 53:484–512, 1984.

[4] S. P. E. Corporation. SPEC Benchmarks, 2009. `http://www.spec.org/benchmarks.html`.

[5] K. Datta, M. Murphy, V. Volkov, et al. 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*, 2008.

[6] J. Demmel, J. Dongarra, V. Eijkhout, et al. Self Adapting Linear Algebra Algorithms and Software. In *Proc. of the IEEE: Special Issue on Program Generation, Optimization, and Adaptation*, February 2005.

[7] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. In *Proc. of the IEEE*, 2004.

[8] M. Gries and K. Keutzer. *Building ASIPs: The Mescal Methodology*. Springer, 2005.

[9] J. P. Grossman, C. Young, J. A. Bank, et al. Simulation and embedded software development for anton, a parallel machine with heterogeneous multicore asics. In C. H. Gebotys and G. Martin, editors, *CODES+ISSS*, pages 125–130. ACM, 2008.

[10] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing Memory Systems for Chip Multiprocessors. In *Proc. of the International Symposium on Computer Architecture*, 2007.

[11] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP design space exploration subject to physical constraints. In *High-Performance Computer Architecture, 2006*, pages 17–28, Feb. 2006.

[12] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proc. of the International Symposium on Computer Architecture*, pages 161–171, 2000.

[13] Micron Inc. Calculating Memory System Power for DDR2, June 2006. `http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf`.

[14] M. Monchiero, R. Canal, and A. González. Design space exploration for multicore architectures: a power/performance/thermal view. In *ICS '06: Proceedings of the International Conference on Supercomputing*, pages 177–186, New York, NY, USA, 2006. ACM.

[15] E. Musoll and M. Nemirovsky. Design Space Exploration of High-Performance Parallel Architectures. In *Journal of Integrated Circuits and Systems*, 2008.

[16] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. In *Journal of High Performance Computing Applications*, 2004.

[17] M. Reilly, L. C. Stewart, J. Leonard, and D. Gingold. SiCortex Technical Summary, 2009.

[18] G. Rivera and C. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of SC'00*, Dallas, TX, November 2000.

[19] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, Aug. 2003.

[20] A. Shrivastava, I. Issenin, and N. Dutt. A Compiler-in-the-Loop Framework to Explore Horizontally Partitioned Cache Architectures. In *Proc. of ASPDAC 2008*, pages 328–333, March 2008.

[21] H. Simon, R. Stevens, T. Zacharia, et al. Modeling and Simulation at the Exascale for Energy and the Environment (E3). Technical report, DOE ASCR Program Technical Report, 2008.

[22] B. G. Team. Overview of the IBM Blue Gene/P Project. *IBM Journal of Research and Development*, 52(1-2), 2008.

[23] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, 2008.

[24] Top500.org. TOP500 List, June 2008. `http://www.top500.org`.

[25] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, December 2003.

[26] R. Vuduc, J. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, June 2005.

[27] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: A Research Accelerator for Multiple Processors. *IEEE Micro*, 27(2), 2007.

[28] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1):3–25, 2001.

[29] S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *Proc. of Supercomputing*, 2007.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and

Methodological Considerations. In *Proc. of ISCA*, 1995.

[31] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized Optimizations for Empirical Tuning. In *Workshop on Performance Optimization for High-Performance Languages and Libraries*, 2007.