A Synergetic Approach to Throughput Computing on x86-Based Multicore Desktops

Chi-Keung Luk, Ryan Newton, William Hasenplaugh, Mark Hampton, and Geoff Lowney, Intel

// *To exploit the full performance potential of multicore desktops, the authors propose an approach that combines cache optimization, parallelization, simdization, and autotuning in a single framework.* //

ADVANCES IN SILICON AND PROCESSOR design technologies in the past few decades have brought enormous computing power to desktop PCs. For instance, a single-socket Intel Nehalem can compute more than 100 giga-floating-point-operations per second (Gflops) and transfer 32 Gbytes of data per second between the CPU and memory. Consequently, we can now deploy many traditional supercomputer applications such as scientific computing, server applications, and emerging applications such as image and video processing on desktops. We collectively define these applications as throughput computing applications.

Nevertheless, harnessing desktops' raw computing power has become a significant challenge to software developers. Limited by power consumption, recent desktops can no longer increase performance by increasing the clock frequency. Instead, they provide more parallel processing units on the same die. Figure 1 shows a block diagram of a dual-socket quad-core Nehalem. The system offers multiple levels of programmable parallelism: two sockets each contain a chip with four cores; each core supports simultaneous multithreading with two hardware threads (T0 and T1); and each core has two single instruction, multiple data (SIMD) units, each of which can execute four 32-bit (or two 64-bit) operations in parallel. These parallel processing units are built on top of a deep memory hierarchy. That is, the two sockets share the main memory, the four cores in each socket share an L3 cache, and each core has a separate instruction cache (I$) and data cache (D$) and a unified L2 cache. Software developers must determine how to exploit both parallelism and data locality for a given application.

We advocate a throughput computing approach that optimizes both parallelism and locality in a single framework. We've developed a synergetic approach to throughput computing for the x86-based multicores. We focus on x86-based multicore desktops because they're the most common computing platforms today. Overall, our approach achieves a nearly 20x speedup over the

0740-7459/11/$26.00 © 2011 IEEE

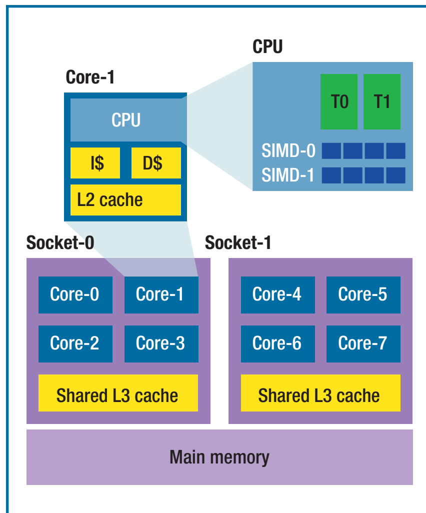


FIGURE 1. Blocked diagram of a dual-socket Intel Nehalem. The image doesn't show the instruction-level parallelism (ILP), which is largely exploited by hardware on the x86 architecture.

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}$$

$$\begin{matrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{matrix} = \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix} \times \begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix}$$

(a)

$$\begin{aligned}
P_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\
P_2 &= (A_{21} + A_{22}) \times B_{11} \\
P_3 &= A_{11} \times (B_{12} - B_{22}) \\
P_4 &= A_{22} \times (B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{12}) \times B_{22} \\
P_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\
C_{11} &= P_1 + P_4 - P_5 + P_7 \\
C_{12} &= P_3 + P_5 \\
C_{21} &= P_2 + P_4 \\
C_{22} &= P_1 - P_2 + P_3 + P_6
\end{aligned}$$

(b)

FIGURE 2. Strassen's matrix-multiplication algorithm. The Strassen algorithm recursively transforms (a) the original matrix multiplication problem into smaller subproblems. (b) P1 to P7 can be computed in parallel. C11 to C22 can be computed in parallel once P1 to P7 are available.

best serial case on a dual-socket quad-core Nehalem.

Our Approach

Both the parallel processing units and caches are organized hierarchically on x86 multicores (see Figure 1), so the divide-and-conquer paradigm fits well with this architecture. In particular, we advocate using cache-oblivious algorithms to exploit thread-level parallelism.[1–3] To exploit SIMD parallelism, we use compiler-based simdization (also known as short vectorization) instead of hand-coded simdization.[4] Finally, during this whole process, we rely on autotuning techniques to tune various program parameters to ensure they achieve good performance.[5,6]

Cache-Oblivious Techniques

A cache-oblivious algorithm is designed to maximize data reuse in caches. Unlike cache blocking, it doesn't have the cache size as an explicit parameter, so it could perform well across multiple cache levels in a memory hierarchy or on machines with different cache configurations. A cache-oblivious algorithm typically works by dividing the original problem into smaller subproblems until reaching a point where the data needed by the subproblem is small enough to fit in any reasonable cache. This stopping point is called the base case. When the subproblems are data independent of each other, we can compute them in parallel. Hence, we achieve both parallelism and data locality at the same time.

Figure 2 illustrates the Strassen matrix-multiplication algorithm, which is an example cache-oblivious algorithm.[7] The original matrix-multiplication problem is recursively transformed into multiplications and additions/subtractions of smaller matrices, which can eventually fit in the cache. Parallelism is naturally exploited; P1 to P7 could be computed in parallel as well as C11 to C22. There are optimal cache-oblivious algorithms that are proved to have an asymptotically minimum number of cache misses.[1] However, because our goal is to use cache-oblivious algorithms to improve performance instead of as an algorithm-analysis tool, we don't restrict ourselves to optimal cache-oblivious algorithms. In particular, in deciding when to stop subdividing a problem, we use autotuning to determine the base-case sizes for different architectures and problems.

Compiler-Based Simdization

Simdization is the software step that extracts parallelism from an application that can be exploited by the hardware SIMD units. Figure 3a shows a loop written in C. Figures 3b and 3c show the execution traces without and with simdization, respectively. By executing four multiplications in one CPU cycle, we can potentially achieve a 4x speed up in the simdized case.

We believe that most developers should use the compiler to simdize instead of doing it by hand. We use the Intel compiler (ICC) because it's widely regarded as having the best simdization support among all x86 compilers. Figures 3d through 3f illustrate three methods to simdize using ICC.

```
void Multiply(int N, float* A, float* B, float* C) {
    for (int i=0; i<N; i++)
        C[i] = A[i] * B[i];
}
```

(a)

Cycle	Instruction executed
0	C[0] = A[0] * B[0]
1	C[1] = A[1] * B[1]
2	C[2] = A[2] * B[2]
3	C[3] = A[3] * B[3]
.	.
.	.
.	.
N-1	C[N-1] = A[N-1] * B[N-1]

(b)

Cycle	Instruction executed
0	C[0..3] = A[0..3] * B[0..3]
1	C[4..7] = A[4..7] * B[4..7]
.	.
.	.
.	.
N/4-1	C[N-4..N-7] = A[N-4..N-7] * B[N-4..N-7]

(c)

```
void Multiply(int N, float* A, float* B, float* C) {
    // Compiler inserts following runtime check
    if (_OverlappedAddressRanges(N, A, B, C)) {
        // non-SIMDized version of the loop
        for (int i=0; i<N; i++)
            C[i] = A[i] * B[i];
    } else {
        // SIMDized version of the loop
        ...
    }
```

(d)

```
void Multiply(int N, float* A, float* B, float* C) {
    #pragma simd
    for (int i=0; i<N; i++)
        C[i] = A[i] * B[i];
}
```

(e)

```
void Multiply(int N, float* A, float* B, float* C) {
    // Rewrite the loop in array notation.
    // A[0:N] represents elements A[0] to A[N-1]
    C[0:N] = A[0:N] * B[0:N];
}
```

(f)

FIGURE 3. Simdization examples. (a) The original loop in C. (b) Execution trace without simdization and (c) with simdization. The Intel compiler (ICC) can simdize this loop with one of the following methods: (d) auto-simdization, (e) programmer-directed simdization, and (f) simdization with array notation.

The first method is auto-simdization (see Figure 3d), where the simdization step is done entirely by the compiler. Since the compiler can't statically determine the data dependencies among the three arrays, it generates two versions of the loop (one is simdized and one isn't) and inserts a check to select which version to use at runtime.

The second method is programmer-directed simdization (see Figure 3e). The programmer uses the ICC pragma `simd` to communicate to the compiler that it's safe and beneficial to simdize the loop.

The last method is array notation,[8] a new feature introduced in ICC v12. We rewrite the loop in an array notation, as shown in Figure 3f. In this notation, we apply operations to arrays instead of scalars. Hence, we no longer need the `for` loop to iterate over individual array elements.

In practice, developers should use auto-simdization whenever possible. When this isn't possible, they could use programmer-directed simdization. In cases where the program structure is too complicated for programmer-directed simdization, developers can use array notation. We expect that with such rich support of simdization in the compiler, developers should rarely need manual simdization.

Autotuning

Autotuning works by generating many

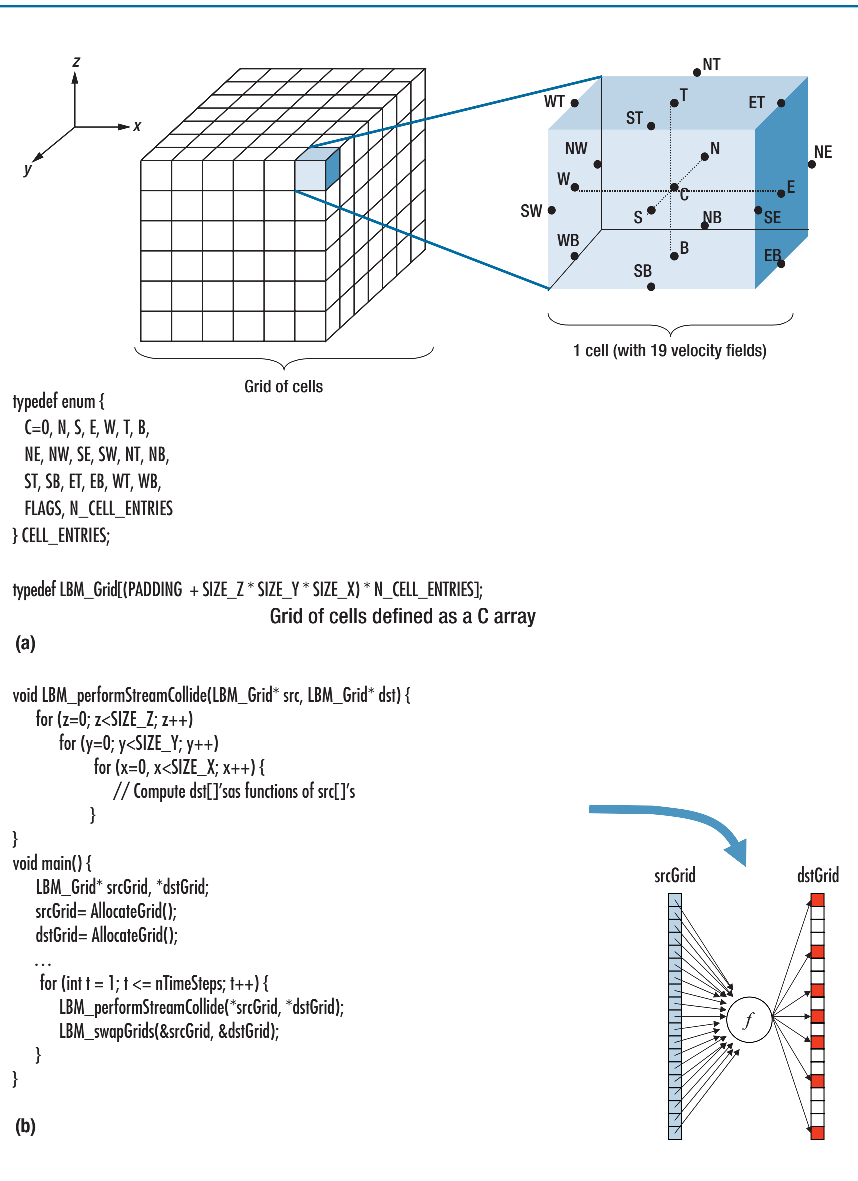


FIGURE 4. Lattice-Boltzman method (LBM) in the SPEC CPU2006 Suite. (a) A sweep through the grid is performed at each time step on (b) the original code.

different variants of the same code and then empirically finding the best-performing variant on the target machine. In our approach, several parameters could be tuned via autotuning:

- *Base-case size in a cache-oblivious algorithm.* We want the base case to be small enough to fit in the cache while at the same time being big enough that the parallelization overhead doesn't overwhelm the benefit. Analytically finding the right base-case size is difficult, if not impossible.
- *Degree of parallelism.* In some situations, using fewer software threads than the number of hardware threads available might result in better performance. This could happen in particular when two software threads are mapped to the same CPU core and hence contending for the same hardware resource. Also, if using all hardware threads versus just a subset of them achieves similar performance, we might want to use fewer threads to consume less energy.
- *Level of parallelism.* In some cases, a chunk of work is best parallelized by distributing it over multiple threads, exploiting thread-level parallelism. In other cases, it's best parallelized by mapping it to a single thread and exploiting SIMD and instruction-level parallelism (ILP) within the thread instead. This choice appears to be best made by autotuning as well.
- *Scheduling policy and granularity.* Threading APIs such as TBB,[9] OpenMP,[10] and Cilk[11] support numerous scheduling policies for users to choose, including static scheduling, dynamic scheduling, and combinations thereof. Also, the granularity of scheduling—for example, how big is the unit of scheduling?—is another parameter that the programmer can often specify via API. The optimal policy and granularity are likely to be problem and machine dependent and so are possibly best selected via autotuning.

To perform autotuning, we developed the Intel Software Autotuning Tool (ISAT), which can tune the parameters we've mentioned so far, as well as others. With ISAT, the programmer adds tuning directives to a program (in the form of *pragmas*) to specify where the program requires tuning, what parameters need to be tuned, and how. ISAT then automatically generates code variants according to this tuning specification, empirically determines the best value for each parameter, and finally produces the tuned version in source code form. Therefore, ISAT can be used on top of any compiler.

Finally, our approach isn't about auto-parallelization but about helping

```
LBM_Grid* Toggle[2];
void LBM_performStreamCollide_Vec(LBM_Grid* src, LBM_Grid* dst,
                    int x0, int x1, int y0, int y1, int z0, int z1) {
    for (z=z0; z<z1; z++)
      for (y=y0; y<y1; y++)
#pragma simd
        for (x=x0, x<x1; x++) {
            // Compute dst[]'s as functions of src[]'s
        }
}
void BaseCase(int t0, int t1, int x0, int dx0, int x1, int dx1,
                    int y0, int dy0, int y1, int dy1,
                    int z0, int dz0, int z1, int dz1) {
    LBM_Grid* src = Toggle[(t0+1) & 1];
    LBM_Grid* dst = Toggle[t0 & 1];
    for (int t=t0; t<t1; t++) {
        LBM_performStreamCollide_Vec(*src, *dst,
                                    x0, x1, y0, y1, z0, z1);
        src = Toggle[t & 1];
        dst = Toggle[(t+1) & 1];
        x0 += dx0; x1 += dx1;
        y0 += dy0; y1 += dy1;
        z0 += dz0; z1 += dz1;
    }
}
int NPIECES=2; int dx_threshold=32; int dy_threshold=2;
int dz_threshold=2; int dt_threshold=3;
#pragma isat tuning measure(start_timing, end_timing)
scope(start_scope, end_scope) variable(NPIECES, range(2, 8, 1))
variable(dx_threshold, range(2, 128, 1))
variable(dy_threshold, range(2, 128, 1))
variable(dz_threshold, range(2, 128, 1))
variable(dt_threshold, range(2, 128, 1))
#pragma isat marker start_scope
void CO(int t0, int t1, int x0, int dx0, int x1, int dx1,
          int y0, int dy0, int y1, int dy1,
          int z0, int dz0, int z1, int dz1) {
    int dt = t1-t0; int dx = x1-x0, int dy = y1-y0; int dz = z1-z0;
    if (dx >= dx_threshold && dx >= dy && dx >= dz &&
      dt >= 1 && dx >= 2 * dt * NPIECES) {
      int chunk = dx / NPIECES; int i;
      for (i=0; i<NPIECES-1; ++i)
        cilk_spawn CO(t0, t1, x0+i*chunk, 1, x0+(i+1)*chunk, -1,
                              y0, dy0, y1, dy1, z0, dz0, z1, dz1);
      cilk_spawn CO(t0, t1, x0+i*chunk, 1, x1, -1,
                            y0, dy0, y1, dy, z0, dz0, z1, dz1);
      cilk_sync(); …
    } else if (… /* Subdivide in y dimension? */)
    …
    } else if (… /* Subdivide in z dimension? */)
    …
    } else if (… /* Subdivide in t dimension? */)
    …
    } else /* call the basecase */
     BaseCase(t0, t1, x0, dx0, x1, dx1, y0, dy0, y1, dy1,
                z0, dz0, z1, dz1);
}
#pragma isat marker end_scope
void main() {
    LBM_Grid* srcGrid, *dstGrid;
    srcGrid = AllocateGrid(); dstGrid = AllocateGrid();
    Toggle[0] = srcGrid; Toggle[1] = dstGrid;
#pragma isat  tuning variable(nWorkers,
                                  range(1, $NUM_CPU_THREADS, 1))
measure(start_timing, end_timing)
    int nWorkers = GetNumHardwareThreads();
    InitCilk(nWorkers);
    …
#pragma isat marker start_timing
  CO(1, nTimeSteps, 0, 0, SIZE_X, 0, 0, 0, SIZE_Y, 0,
      0, 0, SIZE_Z, 0);
#pragma isat marker end_timing
  …
}
```

FIGURE 5. Lattice-Boltzman method (LBM) code optimized by our approach. The function `CO()` recursively divides the 4D iteration space (*x*, *y*, *z*, and time) into smaller subproblems until the base-case criteria is met.

developers write efficient parallel programs using high-level programming techniques. In our approach, the role of autotuning is auxiliary because it's used to improve the effectiveness of the first two steps via parameter searching. In contrast, other researchers have been looking at using autotuning more proactively, such as trying different combinations of code transformations,[5,12] which is outside the scope of our approach. (See the "Related Work in Throughput Computing" sidebar for more details.)

Putting It All Together

We use three case studies to illustrate our approach. First, the Lattice-Boltzmann method uses the common stencil computational pattern. Second, binary-tree search models query searching operations in a database. And the third performs sorting. In all cases, we use the Cilk[11] and simdization support in ICC.

Lattice-Boltzman Method

Stencil computation is an important computational pattern class commonly used in scientific computing, image

RELATED WORK IN THROUGHPUT COMPUTING

A recent study by Victor Lee and his colleagues compared the performance of several computing kernels on a CPU and GPU and found that the GPU is only 2.5x faster than the CPU on average.[1] Their work focuses on performance analysis and the architecture aspect. In contrast, we focus on the software aspect, advocating a high-level programming approach and tool-based optimization.

In the past, researchers have mostly studied cache-oblivious techniques for algorithmic analysis and serial processing.[2–4] Our work shows that cache-oblivious techniques can work well in practice on multicore processors.

Autotuning has also recently become a hot research topic.[5,6] In particular, one study showed that a pure autotuning-based approach can effectively optimize stencil computation.[7] Our approach differs from theirs by using cache-oblivious techniques instead of explicit blocking, although we still use autotuning to tune other parameters and the base case. Using this hybrid approach, we reduce the amount of tuning needed. In addition, our work covers both stencil computations and other domains such as sorting and searching.

References

1. V.W. Lee et al., "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA), ACM Press, 2010, pp. 451–460.
2. M. Frigo et al., "Cache Oblivious Algorithms," *Proc. 40th Ann. Symp. Foundations of Computer Science*, ACM Press, 1999, pp. 285–298.
3. M. Frigo and V. Strumpen, "Cache Oblivious Stencil Computations," *Proc. 2005 Int'l Conf. Supercomputing*, ACM Press, 2005, pp. 361–366.
4. P. Kumar, *Cache Oblivious Algorithms*, LNCS 2625, Springer, 2003, pp. 193–212.
5. D. Bailey et al., "PERI Auto-Tuning," *J. Physics: Conference Series* (SciDAC 2008), vol. 125, no. 1, 2008; www.mcs.anl.gov/uploads/cels/papers/P1517.pdf.
6. R.C. Whaley, A. Petitet, and J.J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing*, vol. 27, nos. 1–2, 2001, pp. 3–35.
7. J. Datta et al., "Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures," *Proc. 2008 ACM/IEEE Conf. Supercomputing*, ACM Press, 2008, article no. 4.

processing, and geometric modeling. A stencil defines the computation of an element in an n-dimensional spatial grid at time t as a function of neighboring grid elements at time $t - 1, \ldots, t - k$.[13]

The particular stencil problem we study is the Lattice-Boltzman method (LBM) benchmark drawn from the SPEC CPU2006 Suite.[14] It performs numerical simulation in computational fluid dynamics in the 3D space. For the main data structure, we used the 3D grid of cells shown in Figure 4a. The original stencil code performs a sweep through the grid at each time step. Figure 4b shows an abstract version of this sweeping code. Two grids srcGrid and dstGrid are used throughout the computation and swapped at the end of each sweep (by `LBM_swapGrids()`). During each sweep, the function `LBM_performStreamCollide()` reads 19 floating-point values from srcGrid, performs 268 floating-point operations, and writes 19 floating-point values to dstGrid. This translates to a ratio of 1.8 flops per byte (flops/byte), suggesting that this function's performance (which accounts for 95 percent of the LBM's total runtime) is limited by memory bandwidth.

Figure 5 sketches how we optimize LBM with our approach. In the new `main()`, we first initialize a two-element array `Toggle[]` to point to srcGrid and dstGrid. Our cache-oblivious code will access both grids via `Toggle[]`. Second, we explicitly set the number of Cilk worker threads used by calling `InitCilk(nWorkers)`. Third, we add several `ISAT` pragmas for the sake of autotuning. Finally, we replace the for-each time-step loop in the original `main()` with a call to `CO()`, which implements the Frigo and Strumpen cache-oblivious stencil algorithm.[13]

Function `CO()` recursively divides the 4D iteration space (x, y, z, and time) into smaller subproblems until the base-case criteria is met. Data-independent subproblems are executed in parallel using `cilk_spawn()` and `cilk_sync()`. The function `BaseCase()` takes the starting and ending points in the four dimensions as parameters. It iterates from time steps `t0` to `t1`. At each time step t, it determines the source and destination grids by indexing `Toggle[]` with `t mod 2` and `(t+1) mod 2`, respectively. It then invokes `LBM_performStreamCollide_Vec()` to sweep through the given ranges of x, y, and z. Note that `#pragma simd` is added to `LBM_performStreamCollide_Vec()` to simdize the x loop.

We add two types of ISAT pragmas to Figure 5. The first type is in the form of `#pragma isat marker ...` for marking a region in the program. In this example, we mark two regions: (`start_scope` and `end_scope`) and (`start_timing` and `end_timing`). The former region defines the lexical scope of the variables being tuned. The latter region defines the timing scope, where ISAT measures the performance of code variants. The second type of ISAT pragma marks tuning variables: `#pragma isat tuning measure(M0, M1) scope(S0, S1) variable(Var0, Range0) ... variable(VarN, RangeN)`. It instructs ISAT to tune the variables specified by the variable clauses

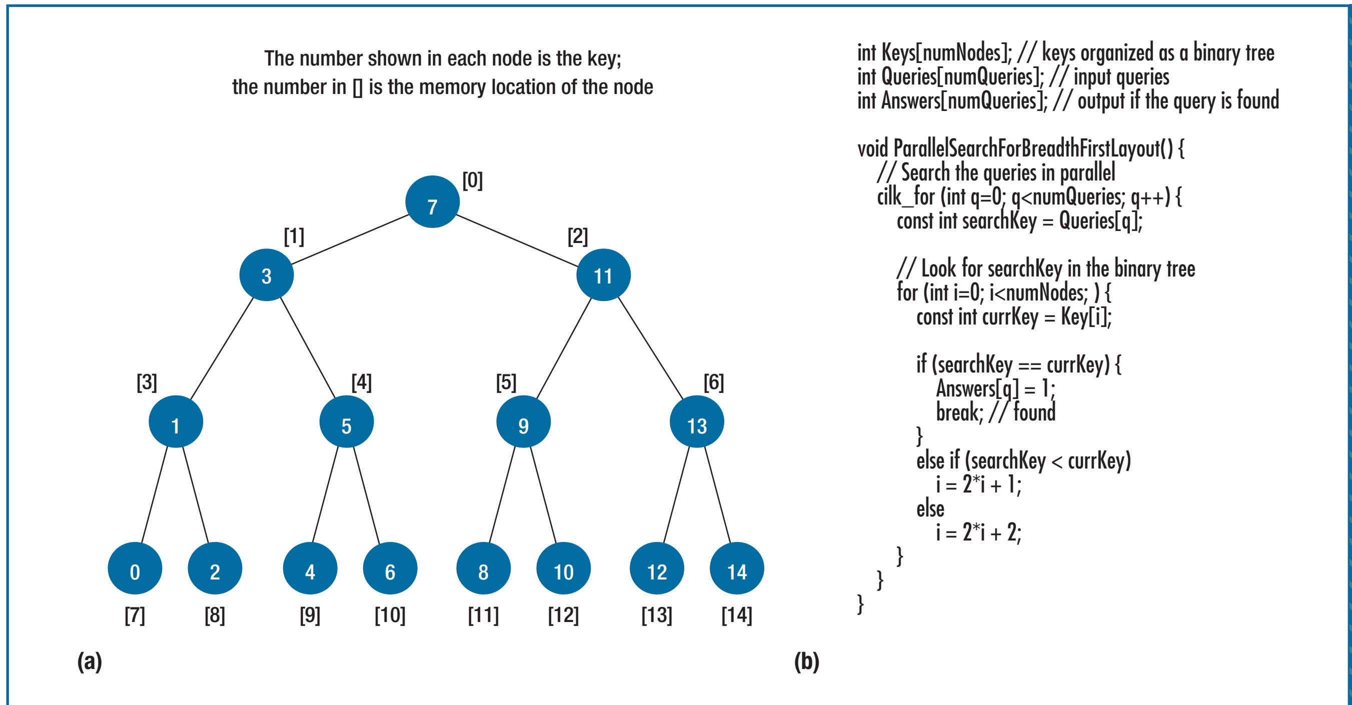


FIGURE 6. Packed binary tree. (a) The breadth-first layout in memory and (b) its corresponding query search code.

within the scope (S0, S1) by measuring their performance impact on the region (M0, M1). A variable clause's first argument is the variable being tuned, and the second argument is the range of values to be tried, which can be expressed in the form of (startValue, endValue, increment) or [startValue .. endValue]. A value started with the $ symbol is predefined by ISAT. For instance, $NUM_CPU_THREADS is the number of hardware threads available on the CPU. In this example, the parameters being tuned are the number of threads used by Cilk, and the five parameters (NPIECES, dx_threshold, dy_threshold, dz_threshold, and dt_threshold) in the cache-oblivious algorithm.

Binary-Tree Search

This case study is about searching for a query based on its key in a database organized as a packed binary tree. The tree is originally laid out in memory in a breadth-first manner (see Figure 6a). Figure 6b shows the corresponding query search code. We use cilk_for, which is similar to OpenMP's parallel-for, to search for independent queries in parallel.

Figure 6 highlights two optimization opportunities. First, as we get close to the bottom of the tree, the nodes accessed during the search for a single query won't be on the same cache lines and will therefore cause many cache misses. Second, we haven't taken advantage of the SIMD units. To reduce cache misses, we can layout the tree in a cache-oblivious way. The theoretically optimal method (in terms of cache misses) to do this is the Van Emde Boas (VEB) layout.[15] Nevertheless, we find that the searching code for the VEB layout isn't amenable to efficient simdization, so we instead use a nonoptimal cache-oblivious layout that enables simdization.

Figure 7a shows the new data layout, where we divide the original tree into multiple layers of subtrees of height SUBTREE_HEIGHT. Nodes in each subtree are laid out breadth first. This layout ensures that the nodes accessed during the search for a single query are always on the same or nearby cache lines, regardless of their tree levels. Figure 7b shows the corresponding search code. We divide the input queries into several bundles, each containing (BUNDLE_WIDTH * VLEN) queries. The cilk_for schedules bundles to threads. Each thread processes VLEN queries at a time until all queries in its bundle are done. We use array notation to map the VLEN queries to SIMD hardware. Finally, we use ISAT to tune the three parameters (SUBTREE_HEIGHT, BUNDLE_WIDTH, and VLEN). We tune SUBTREE_HEIGHT in one pragma and tune BUNDLE_WIDTH and VLEN together in another pragma because BUNDLE_WIDTH and VLEN are best searched dependently while SUBTREE_HEIGHT can be searched independently. This is an example of tuning the distribution of work over thread-level, instruction-level, and SIMD-level parallelism.

Sorting

Sorting an array is another problem amenable to a divide-and-conquer, cache-oblivious approach. For example, the merge-sort algorithm recursively sorts both halves of an array independently before recombining them. Likewise, quick sort separates elements into two categories before recursively

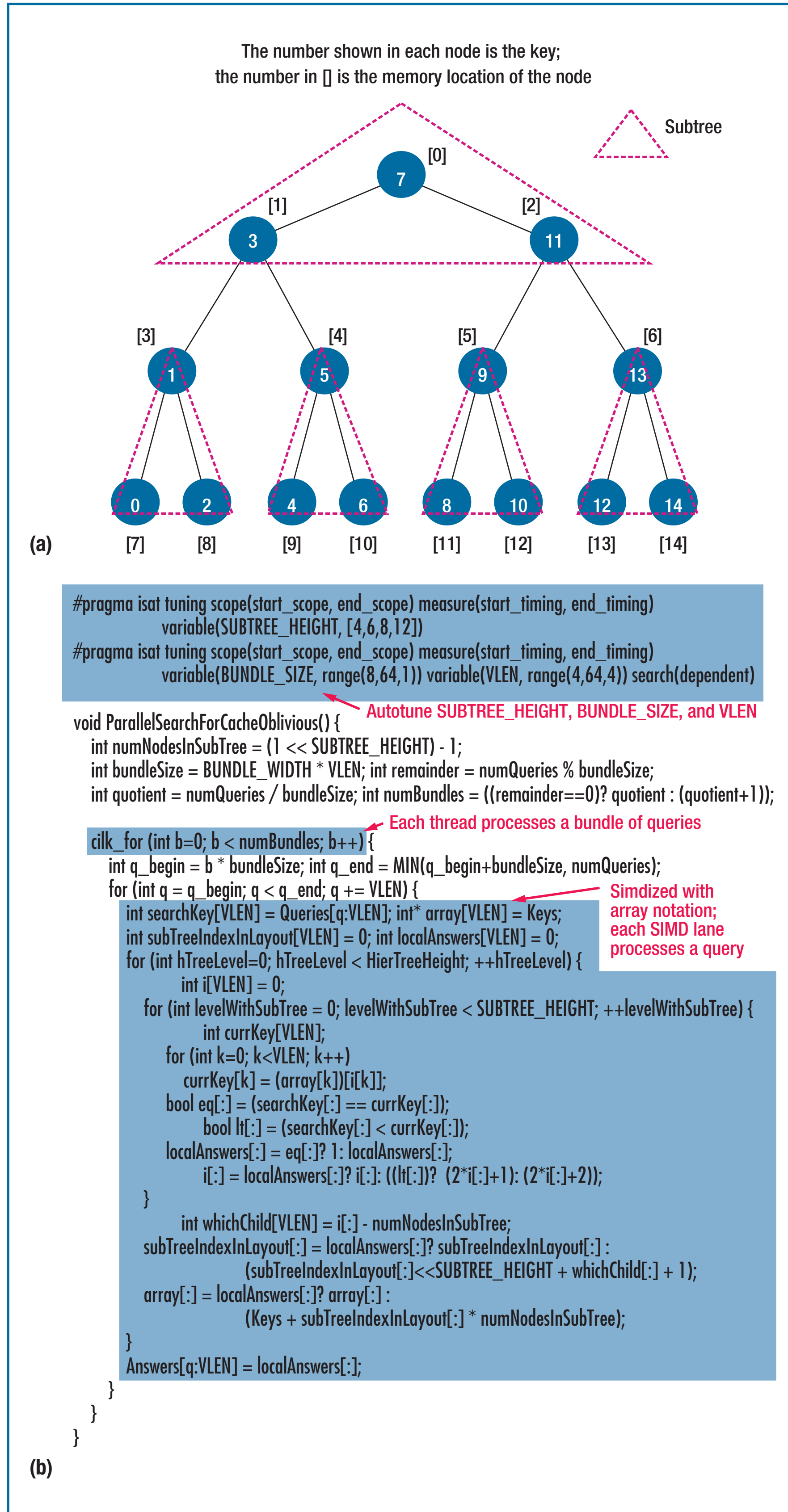


FIGURE 7. Optimizing the search with cache-oblivious layout and array notation. (a) We divided the cache-oblivious tree layout into multiple layers of subtrees. (b) The optimized version parallelizes the search with a cilk_for loop and then simdizes each loop iteration using an array notation.

processing them. In fact, independent portions of a sequence can be sorted using completely different algorithms, and the best-performing codes are an amalgam of distinct algorithms for different levels of the memory hierarchy. One reason to mix different algorithms is that traditional serial sorting algorithms have data-dependent control flows that aren't amenable to automatic simdization. A solution is to use sorting networks at smaller sizes to expose fine-grained parallelism. Our implementation uses two kinds of sorting networks together with a coarse-grained parallel merge sort—a subset of the techniques described in earlier work[16] that we've reimplemented using our synergetic approach.

Merge sort exposes task parallelism at a coarse granularity.[17] We augment the basic algorithm to make the merge step (as well as the recursive step) parallel. Before merging two sorted subsequences, we search for what will become the median element in the merged output. The median serves as a pivot (much like quick sort), allowing independent, recursive processing of all elements under and over the median. We use cilk_spawn to expose the task parallelism both in the downward sort phase and in the upward merging phase. The algorithm switches from parallel to serial at a base-case size determined by autotuning.

Bitonic merge networks expose ILP and enable SIMD when merging two sorted subsequences. A bitonic merge network of size $2N$ has $N - 1$ stages, each stage comparing and swapping elements at decreasing distances. The number of comparisons in each stage is the same, but to simdize the com-

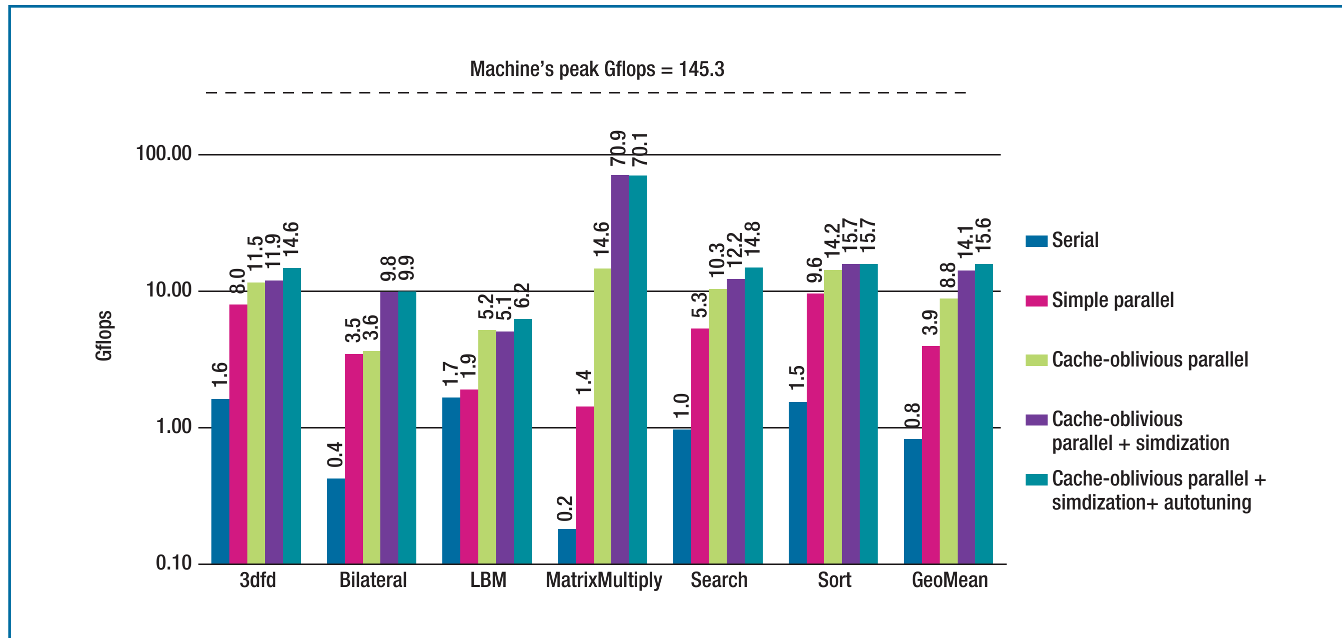


FIGURE 8. Performance results of our approach. The benchmarks plus their average are on the *x* axis, and the performance in gigaflops is on the *y* axis (in log scale).

putation, elements must be shuffled into position between stages at smaller comparison distances. Our merge sort (1.) invokes a bitonic merge network to consume CHUNKSIZE elements simultaneously. That is, at each step, the chunk (already internally in order) with a minimum leading element is taken from the head of a sequence being merged. The chunk is mixed with leftovers from the previous step by a bitonic merge network of size 2*CHUNKSIZE. The minimum half of the sorted result is output and the rest become new leftovers. Chunk size is autotuned.

The in-register sort via sorting network (finest grain) algorithm ensures that chunks are internally sorted. It treats CHUNKSIZE chunks to be sorted as the rows of a square matrix. The matrix is transposed (with shuffles), turning rows into columns, and then sorted with vector operations between rows. When the matrix is transposed a second time, the original rows are internally sorted. Any fixed sorting routine could be used; we choose a Batcher odd-even sort.

In trying to write a generic and portable version of these three algorithms, several implementation difficulties arise. The sorting networks we described rely heavily on permuting vectors. Permutation code isn't currently amenable to automatic compiler-based simdization, but array notation allows arbitrary permutations (automatically generating shuffle instructions for the target machine) if permutations are known at compile time. Unfortunately, arbitrarily sized bitonic and odd-even networks can only be implemented by recursive functions. In fact, because these kernels are at the heart of our computation, eliminating the recursive function calls is necessary for performance. Thus, staged code generation (or partial evaluation) is appropriate. (The Intel Array Building Blocks is an appropriate framework for staged code generation in this example.) We use a complementary technique to autotuning that we call lightweight code generation. Whenever a computation kernel is needed at different sizes or configurations for autotuning or portability purposes, we write a simple program generator (a script) to produce a large set of different kernels.

Code generation is usually thought of as relying on heavyweight infrastructure—for example, in the context of large, complex compilers. But we argue that for limited purposes (kernels), little work is required to build simple code generators in any high-level language (such as Python and Haskell). In this case, we wrote 86 lines of noncomment, nonblank Scheme code for manipulating permutations and another 135 lines of code that generate arbitrarily sized bitonic and odd-even kernel functions and output them to a .c file.

Evaluation

For our experiment, we used an Intel Nehalem, with eight cores (on two sockets), a 2.27-GHz core clock, and 12 Gbytes of memory. The architecture also used a 22.6 Gbyte/sec memory bandwidth; the 64-bit CentOS v4; and the ICC v12, -fast option complier. Table 1 shows the details of the benchmarks we used, which are important throughput computing kernels also used by other researchers.[18,19]

Figure 8 shows our overall performance results. We compiled the serial cases with the -fast option in ICC, which generally produces the best-performing

TABLE 1 Benchmarks.

Benchmark	Description	Problem size
3dfd[19]	3D finite difference computation	$x = 1{,}000$, $y = 1{,}000$, $z = 1{,}000$, $t = 20$
Bilateral[18]	Bilateral image filtering	8,000 × 8,000 pixels
LBM[14]	Lattice-Boltzman method	Reference input
Matrix Multiply[19]	Dense matrix multiplication	4,000 × 4,000 dimensions
Search[18]	Searching a binary tree	24-level tree, 4 million queries
Sort[18]	Sorting	16 million elements

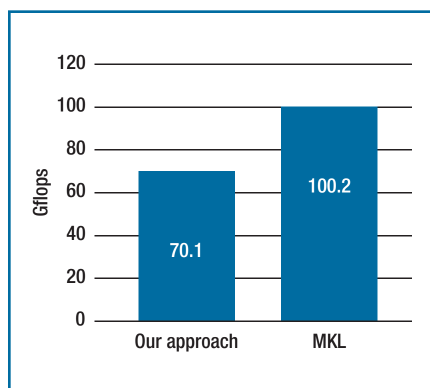


FIGURE 9. Performance of single-precision matrix multiplication. Our approach achieves performance comparable to the Intel Math Kernel Library (MKL v11).

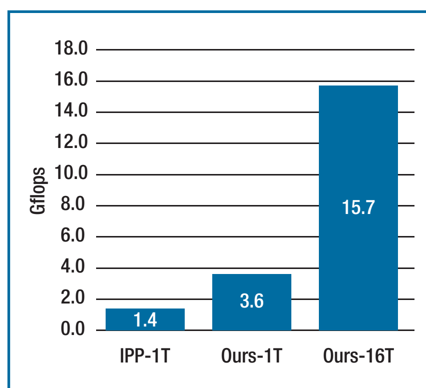


FIGURE 10. Sorting performance. IPP-1T stands for the Intel Integrated Performance Primitives v7 with one thread, Ours-1T stands for our approach with one thread, and Ours-16T shows results for our approach with 16 threads. IPP currently doesn't support multithreaded sorting.

code. For each benchmark, we show simple loop-based parallelization in Cilk; cache-oblivious parallelization; both cache-oblivious parallelization and compiler-based simdization; and cache-oblivious parallelization, simdization, and autotuning together.

As Figure 8 shows, simple loop-based parallelization achieves a 4.8x speedup on average, which isn't bad given an eight-core machine. Nevertheless, cache-oblivious techniques improve the average speedup to 10.7x, more than doubling the performance. This apparently superlinear speedup is a result of improved cache locality. Their impacts are particularly large in LBM, Search, and Matrix Multiply. Adding simdization improves the average speedup to 17.3x, especially helping Matrix Multiply and Bilateral. Finally, autotuning further improves performance of 3dfd, LBM, and Search. Overall, our approach achieves an average speedup of 19.1x over the best serial case or four times faster than simple parallelization. Nevertheless, Figure 8 also shows that our best average is 15.6 Gflops, which is still far below the machine's peak of 145.3 Gflops, indicating that we're largely limited by the memory latency.

To get an idea of how well our results compared against highly tuned codes, Figure 9 compares the performance of single-precision matrix multiplication with our approach and the Intel Math Kernel Library (MKL v11), while Figure 10 compares sorting performance with our approach and the Intel Integrated Performance Primitives (IPP v7). It's encouraging that our high-level approach achieves comparable or better performance than highly tuned library codes.

Figure 11 shows the LBM's performance with various optimization strategies. The serial case (first bar) achieves only 1.7 Gflops. Applying simple loop-based parallelization and simdization (the second bar) improves performance by only 17 percent because this application is limited by memory bandwidth. One optimization that is known to be effective to this application is the Array of Structures (AOS) to Structure of Arrays (SOA) transformation, which is the third bar. It results in 2.6x speedup over the serial case. Our approach (the fourth bar) achieves 3.8x speedup over serial without changing the data layout at all.

Figure 12 shows how the execution time of the Search benchmark changes as we vary the two parameters `VLEN` and `BUNDLE_WIDTH`. There are a number of local minimums, and the best configuration is `(VLEN=48, BUNDLE_WIDTH=32)`. This contrasts with the intuitive choice of `VLEN=4`, the number of SIMD lanes. Fortunately, autotuning enables us to pick this nonoblivious choice.

Finally, an interesting future work is to apply our approach to other architectures such as GPUs. The Intel compiler is available for purchase at http://software.intel.com/en-us/intel-compilers, and the In-

tel Software Autotuning Tool is freely available at http://software.intel.com/en-us/whatif.

Acknowledgments

We thank Charles Leiserson and the anonymous reviewers for their helpful feedbacks, Matteo Frigo and Yuxiong He for providing the initial implementations of 3dfd, and Mark Charney for letting us use the Intel Software Development Emulator tool.

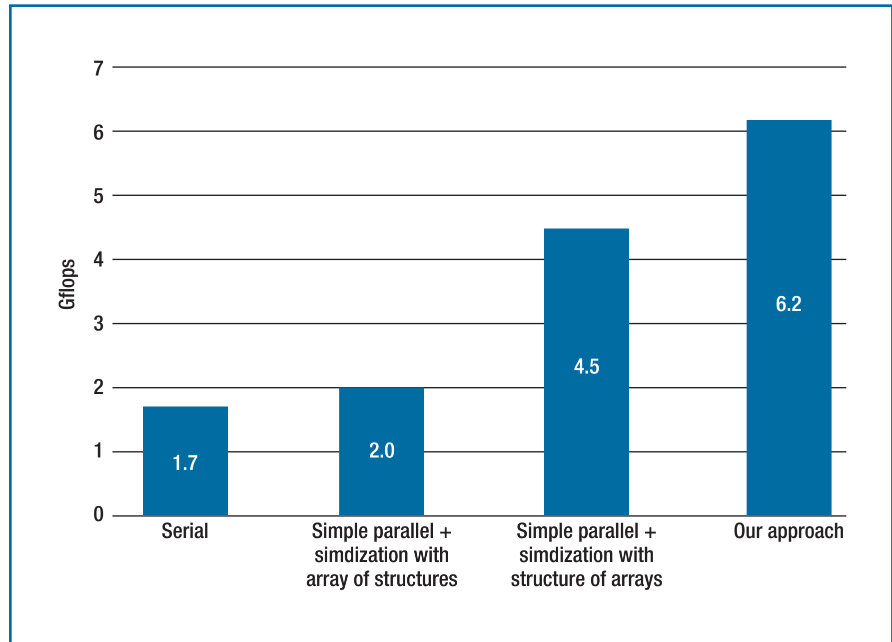


FIGURE 11. Performance of the Lattice-Boltzman method. The serial case (first bar) achieves only 1.7 Gflops, while our approach (the last bar) achieves 6.2 Gflops.

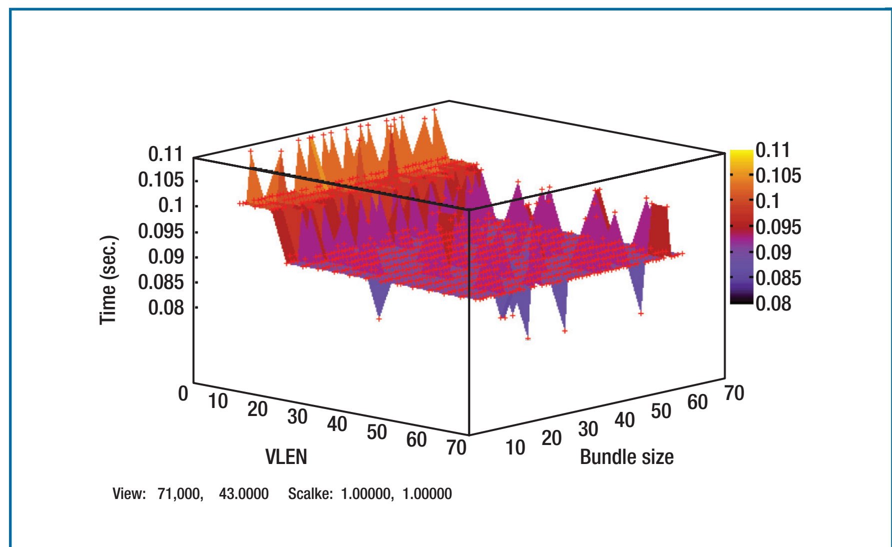
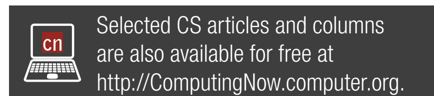


FIGURE 12. Performance impact of autotuning on the Search benchmark. The best configuration is **(VLEN=48, BUNDLE_WIDTH=32)**.

References

1. M. Frigo et al., "Cache Oblivious Algorithms," *Proc. 40th Ann. Symp. Foundations of Computer Science*, ACM Press, 1999, pp. 285–298.
2. P. Kumar, *Cache Oblivious Algorithms*, LNCS 2625, Springer, 2003, pp. 193–212.
3. H. Prokop, "Cache-Oblivious Algorithms," master's thesis, Laboratory for Computer Science, Massachusetts Inst. of Technology, June 1999.
4. A.J. Bik, *The Software Vectorization Handbook*, Intel Press, 2006.
5. D. Bailey et al., "PERI Auto-Tuning," *J. Physics: Conference Series* (SciDAC 2008), vol. 125, no. 1, 2008; www.mcs.anl.gov/uploads/cels/papers/P1517.pdf.
6. R.C. Whaley, A. Petitet, and J.J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing*, vol. 27, nos. 1–2, 2001, pp. 3–35.
7. V. Strassen, "Gaussian Elimination Is Not Optimal," *Numerical Mathematics*, vol. 13, 1969, pp. 354–356.
8. "Using Parallelism: (CEAN) C/C++ Extension for Array Notation," Intel, Mar. 2010.
9. J. Reinders, *Intel Threading Building Blocks*, O'Reilly, 2007.
10. R. Chandra et al., *Parallel Programming in OpenMP*, Morgan Kaufmann, 2001.
11. R.D. Blumofe et al., "Cilk: An Efficient Multithreaded Runtime System," *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP), ACM Press, 1995, pp. 207–216.
12. M. Hall et al, "Autotuning and Specialization: Speeding up Nek5000 with Compiler Technology," *Proc. Int'l Conf. Supercomputing*, ACM Press, 2010, pp. 253–262.
13. M. Frigo and V. Strumpen, "Cache Oblivious Stencil Computations," *Proc. 2005 Int'l Conf. Supercomputing*, ACM Press, 2005, pp. 361–366.
14. J.L. Henning, "SPEC CPU2006 Benchmark Descriptions," *Computer Architecture News*, vol. 34, no. 4, 2006, pp. 1–17.
15. M.A. Bender, E.D. Demaine, and M. Farach-Colton, "Cache-Oblivious B-trees," *Proc. 41st Ann. Symp. Foundations of Computer Science*, ACM Press, 2000, pp. 399–409.
16. J. Chhugani et al., "Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture," *Proc. VLDB Endowment*, vol. 1, no. 2, 2008, pp. 1313–1324.
17. S.G. Akl and N. Santoro, "Optimal Parallel Merging and Sorting without Memory Conflicts," *IEEE Trans. Computers*, vol. 36, no. 11, 1987, pp. 1367–1369.
18. V.W. Lee et al., "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *Proc. 37th Ann. Int'l Symp. Computer Architecture* (ISCA), ACM Press, 2010, pp. 451–460.
19. NVIDIA, "CUDA SDK," www.nvidia.com/object/cuda_get.html.



ABOUT THE AUTHORS

CHI-KEUNG LUK is a senior staff engineer at Intel. His research interests include parallel programming tools and techniques, compilers, and virtualization. Luk has a PhD in computer science from the University of Toronto. Contact him at chi-keung.luk@intel.com.

RYAN NEWTON is a software engineer at Intel. His research interests include parallel and distributed programming. Newton has a PhD in computer science from the Massachusetts Institute of Technology. Contact him at ryan.r.newton@intel.com.

WILLIAM HASENPLAUGH is a computer architect at Intel. His research interests include memory system design and performance optimization. Hasenplaugh has an MS in electrical engineering and optics from the University of Arizona. Contact him at william.c.hasenplaugh@intel.com.

MARK HAMPTON is a software engineer at Intel. His research interests include parallel programming and performance. Hampton has a PhD in computer science from the Massachusetts Institute of Technology. Contact him at mark.hampton@intel.com.

GEOFF LOWNEY is an Intel Fellow and CTO of the Developer Products Division at Intel. His research interests include compilers, programming tools, performance analysis, and parallel programming. Lowney has a PhD in computer science from Yale University. Contact him at geoff.lowney@intel.com.

IT Professional
TECHNOLOGY SOLUTIONS FOR THE ENTERPRISE
www.computer.org/itpro

IEEE Software

HOW TO REACH US

WRITERS

For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access www.computer.org/software/author.htm.

LETTERS TO THE EDITOR

Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

ON THE WEB

www.computer.org/software

SUBSCRIBE

www.computer.org/software/subscribe

SUBSCRIPTION CHANGE OF ADDRESS

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software.*

MEMBERSHIP CHANGE OF ADDRESS

Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

MISSING OR DAMAGED COPIES

If you are missing an issue or you received a damaged copy, contact help@computer.org.

REPRINTS OF ARTICLES

For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

REPRINT PERMISSION

To obtain permission to reprint an article, contact the Intellectual Property Rights Office at copyrights@ieee.org.