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

Chi-Keung Luk, Ryan Newton, William Hasenplaugh, Mark Hampton, Geoff Lowney
Intel Corporation
Hudson, MA 01749
chi-keung.luk@intel.com

ABSTRACT

In the era of multicores, many applications that tend to require substantial compute power and data crunching (aka *Throughput Computing Applications*) can now be run on desktop PCs. However, to achieve the best possible performance, applications need to be written in a way that exploits both *parallelism* and *cache locality*. In this paper, we propose one such approach for x86-based architectures. Our approach uses *cache-oblivious techniques* to divide a large problem into smaller subproblems which are mapped to different cores or threads. We then use the compiler to exploit *SIMD parallelism* within each subproblem. Finally, we use *autotuning* to pick the best parameter values throughout the optimization process. We have implemented our approach with the *Intel®Compiler* and the newly developed *Intel®Software Autotuning Tool*. Experimental results collected on a dual-socket quad-core Nehalem show that our approach achieves an average speedup of almost 20x over the best serial cases for an important set of computational kernels.

Keywords

Multicore, Throughput Computing, Cache-Oblivious Algorithms, Parallelization, Simdization, Vectorization, Autotuning.

1. INTRODUCTION

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 over 100 GFLOPS[1] and transfer 32GB of data per second between the CPU and memory. As a result, many traditional supercomputer applications like scientific computing, server applications like databases, and emerging applications like image and video processing can now be deployed on desktops. We collectively define these applications as *Throughput Computing Applications*.

Nevertheless, harnessing the raw compute power of desktops has become a very significant challenge to software developers. Limited by the 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[2]: there are two sockets, each containing a chip with four cores; each core supports simultaneous multithreading with two hardware threads (T0 and T1); each core has two SIMD[3] 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: the two sockets share the main memory; the four cores in each socket share an L3 cache; each core has a separate instruction cache (I$) and data cache (D$) and a unified L2 cache. The challenge faced by software developers is how to exploit both *parallelism* and *data locality* for a given application.

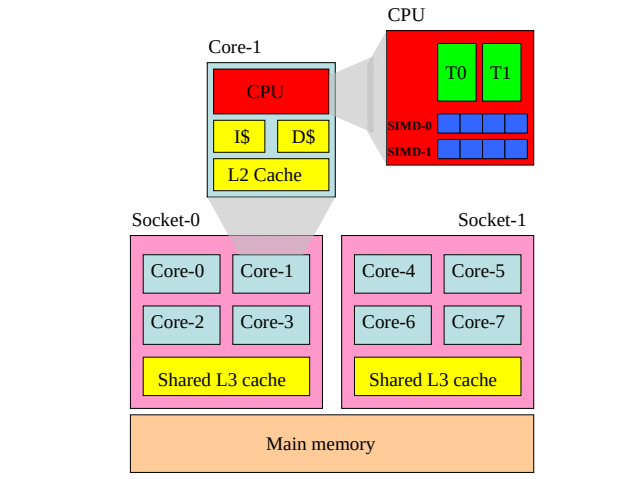


Figure 1: Blocked diagram of a dual-socket Intel Nehalem

In this work, we advocate an approach to throughput computing that optimizes both parallelism and locality in a single framework. We decide to focus on x86-based multicore desktops, since they are the most common compute platforms these days. We will first describe our approach and then present three case studies. Then we will show some experimental evidence that our approach is effective. Finally, we relate our work to others' and conclude.

2. OUR APPROACH

2.1 Overview

[1]GFLOPS stands for Giga Floating point Operations Per Second.

[2]Not shown in the figure is instruction-level parallelism (ILP) which is largely exploited by hardware on the x86 architecture.

[3]SIMD stands for Single Instruction Multiple Data.

To appear in 2011 Jan/Feb *IEEE Software: Special Issue on Software for the Multiprocessor Desktop*

Matrix-multiplication problem:

$$C = A \times B$$

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

Strassen algorithm:

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

P_1 to P_7 can be computed in parallel

$C_{11} = P_1 + P_4 - P_5 + P_7$
$C_{12} = P_3 + P_5$
$C_{21} = P_2 + P_4$
$C_{22} = P_1 - P_2 + P_3 + P_6$

C_{11} to C_{22} can be computed in parallel once P_1 to P_7 are available

Figure 2: Strassen's matrix multiplication algorithm

We observe on x86 multicores that both the parallel processing units and caches are organized hierarchically as shown in Figure 1. Therefore, the *divide-and-conquer* paradigm fits very well to this architecture. In particular, we advocate using a class of techniques called *cache-oblivious algorithms* [10, 17, 20] to exploit *thread-level parallelism*. To exploit *SIMD* parallelism, we use *compiler-based simdization* [4] instead of hand-coded simdization. Finally, during this whole process, various program parameters must be tuned to achieve good performance. We rely on *autotuning* techniques [2, 6, 9, 12, 15, 16, 23, 24, 25] to tune these parameters.

2.2 Cache-Oblivious Techniques

A cache-oblivious algorithm is one that is designed to maximize data reuse in caches. Unlike cache blocking, it does not have the cache size as an explicit parameter (hence "cache oblivious"). 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 and 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.

An example cache-oblivious algorithm is the Strassen matrix multiplication algorithm [22], as illustrated in Figure 2. 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: P_1 to P_7 could be computed in parallel as well as C_{11} to C_{22}.

There are *optimal* cache-oblivious algorithms that are proved to have asymptotically minimum number of cache misses [10]. Nevertheless, since our goal is to use cache-oblivious algorithms to improve performance instead of as an algorithm analysis tool, we do *not* restrict ourselves to optimal cache-oblivious algorithms. In particular, in deciding when to stop subdividing a problem, we use *au-*

(a) Original loop in C

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

(b) Execution without simdization

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]

(c) Execution with simdization

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-1] = A[N-4..N-1] * B[N-4..N-1]

(d) Auto-simdization

```
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
        ...
    }
}
```

(e) Programmer-directed simdization

```
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];
}
```

(f) Simdization with array notation

```
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];
}
```

Figure 3: An Example of Simdization with the Intel®Compiler

totuning to determine the basecase sizes for different architectures and problems.

2.3 Compiler-based Simdization

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

We believe that most developers should use the compiler to simdize instead of doing it by hand. We focus on using the Intel®Compiler (ICC) since it is widely regarded as having the best simdization support among all x86 compilers. Figures 3 (d)-(f) illustrate three methods to simdize using ICC.

The first method is *Auto-Simdization*, shown in Figure 3(d), where the simdization step is done entirely by the compiler. Since the compiler cannot 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*, as shown in Figure 3(e). The programmer uses the ICC pragma `simd` to communicate to the compiler that it is safe and beneficial to simdize the loop.

The last method is to use *Array Notation* [14], a new feature introduced in ICC v12. We rewrite the loop in array notation as shown in Figure 3(f). 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 is not possible, they could go for programmer-directed simdization. In cases where the program structure is too complicated for programmer-directed simdization, they can use array notation. We expect that with such rich support of simdization in the compiler, developers should rarely need manual simdization.

2.4 Autotuning

Autotuning [2, 6, 9, 12, 15, 16, 23, 24, 25] is an approach for producing efficient and portable codes. It works by generating many different variants of the same code and then empirically finding the best performing variant on the target machine. In our approach, there are a number of parameters that could be tuned via autotuning, including:

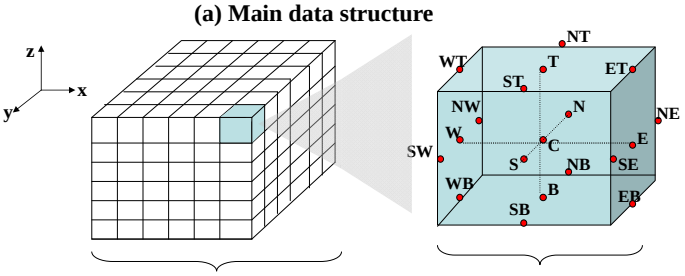
Basecase size in a cache-oblivious algorithm: We want the basecase to be small enough to fit in the cache, while at the same time big enough that the overhead of parallelization does not overwhelm the benefit. Analytically finding the right basecase size is difficult if not impossible.

Degree of parallelism: In some situations, using fewer software threads than the number of hardware threads available may 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 vs. just a subset of them achieve similar performance, we may 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 is 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. We will show a concrete example of such tuning in the case study in Section 3.2.

Scheduling policy and granularity: Threading APIs such as TBB [21], OpenMP [7], and Cilk [5] support a number of scheduling policies for users to choose, including static scheduling, dynamic scheduling, and combinations thereof. Also, the granularity of scheduling (i.e. 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.

(a) Main data structure



Grid of cells — 1 cell (with 19 velocity fields)

```
typedef enum {
   C=0, N, S, E, W, T, B,
   NE, NW, SE, SW, NT, NB,
   ST, SB, ET, EB, WT, WB,
   FLAGS, N_CELL_ENTRIES
} CELL_ENTRIES;

typedef LBM_Grid[(PADDING + SIZE_Z * SIZE_Y * SIZE_X) * N_CELL_ENTRIES];
```

Grid of cells defined as a C array

(b) Original sweeping code

```
void LBM_performStreamCollide(LBM_Grid* src, LBM_Grid* dst) {
  for (z=0; z<SIZE_Z; z++)
    for (y=0; y<SIZE_Y; y++)
      for (x=0, x<SIZE_X; x++) {
        // Compute dst[]'s as functions of src[]'s
      }
}
void main() {
  LBM_Grid* srcGrid, *dstGrid;
  srcGrid = AllocateGrid();
  dstGrid = AllocateGrid();
  …
  for (int t=1; t<= nTimeSteps; t++) {
    LBM_performStreamCollide(*srcGrid, *dstGrid);
    LBM_swapGrids(&srcGrid, &dstGrid);
  }
}
```

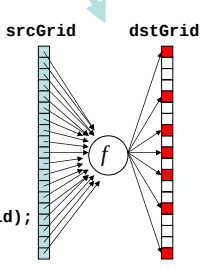


Figure 4: Lattice Boltzman Method (LBM) in SPEC'06

To perform autotuning, we have developed the *Intel®Software Autotuning Tool (ISAT)* which is able to tune the parameters men-

```
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,(2, 8, 1))
variable(dx_threshold, (2, 128, 1))
variable(dy_threshold, (2, 128, 1))
variable(dz_threshold, (2, 128, 1))
variable(dt_threshold, (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, (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: LBM code optimized by our approach

tioned above and other program parameters. With ISAT, the programmer adds tuning directives to a program (in the form of pragmas) to specify where in 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). We will show an example program which uses ISAT in Section 3.

Lastly, we need to emphasize that our approach is *not* about auto-parallelization. It is about how developers could write efficient parallel programs using high-level programming techniques. Also, in our approach, the role of autotuning is auxiliary as it is used to improve the effectiveness of the first two steps via parameter searching. In contrast, other researchers have been looking at using autotuning in a more proactive way such as trying different combinations of code transformations [2, 12], which is outside the scope of our approach.

3. PUTTING IT ALL TOGETHER

In this section, we discuss three case studies that use the approach described in the previous section. The first one, Lattice Boltzmann Method, uses the common stencil computational pattern. The second one, binary-tree search, models query searching operations in a database. The third one performs sorting. In all cases, we use the Cilk [5] and simdization support in ICC.

3.1 Stencil Computation: Lattice Boltzman Method (LBM)

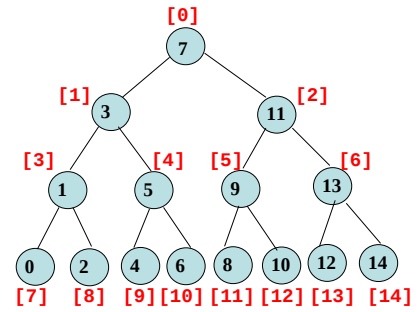
Stencil computation is an important class of computational pattern commonly used in scientific computing, image processing, and geometric modeling. A *stencil* defines the computation of an element in an n-dimensional spatial grid at time t as a function of neighboring grid elements at time $t-1, \ldots, t-k$ [11].

The particular stencil problem we study is the Lattice Boltzman Method (LBM) benchmark drawn from the SPEC CPU2006 Suite [13]. It performs numerical simulation in computational fluid dynamics in the 3D space. The main data structure used is the 3D grid of *cells* shown in Figure 4(a). The original stencil code performs a sweep through the grid at each time step. Figure 4(b) shows an abstract version of this sweeping code. Two grids `srcGrid` and `dstGrid` are used throughout the computation and they are 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 finally writes 19 floating-point values to `dstGrid`. This translates to a FLOP/Byte ratio of 1.8 FLOP/Byte, suggesting that performance of this function (which accounts for 95% of the total runtime of LBM) 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 a number of `isat` pragmas for the sake of autotuning, which we will explain later. Finally, we replace the for-each-time-step loop in the original `main()` by a call to `CO()`, which implements the cache-oblivious stencil algorithm proposed by Frigo and Strumpen [11].

Function `CO()` recursively divides the 4D iteration space (x, y, z and $time$) into smaller and smaller subproblems until the base-case criteria is met. Data-independent subproblems are executed in parallel using `cilk_spawn()` and `cilk_sync()`. The function

(a) Breadth-first layout in memory



(b) The corresponding query search code

```
int Keys[numNodes]; // keys organized as a binary tree
int Queries[numQueries]; // input queries
int Answers[numQueries]; // output if the query is found

void ParallelSearchForBreadthFirstLayout() {
    // Search the queries in parallel
    cilk_for (int q=0; q<numQueries; q++) {
        const int searchKey = Queries[q];

        // Look for searchKey in the binary tree
        for (int i=0; i<numNodes; ) {
            const int currKey = Key[i];

            if (searchKey == currKey) {
                Answers[q] = 1;
                break; // found
            }
            else if (searchKey < currKey)
                i = 2*i + 1;
            else
                i = 2*i + 2;
        }
    }
}
```

Figure 6: Packed binary tree laid out in memory in the breadth-first manner and the corresponding search code.

`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 $t \bmod 2$ and $(t+1) \bmod 2$, 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`, `end_scope`) and (`start_timing`, `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 within the scope (`S0`, `S1`) by measuring their performance impact on the region (`M0`, `M1`). The first argument of a `variable` clause 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 "`$`" symbol is pre-defined by ISAT. For instance, `$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`, `dt_threshold`) in the cache-oblivious algorithm.

3.2 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, as shown in Figure 6(a); the corresponding query search code is shown in Figure 6(b). We use `cilk_for`, which is similar to the parallel-for of OpenMP, to search for independent queries in parallel.

There are two optimization opportunties in Figure 6. First, as we get close to the bottom of the tree, the nodes accessed during the search for a *single* query would not be on the same cache lines and thereby causing many cache misses. Second, we have not 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* described by Bender *et.al.* [3]. Nevertheless, we find that the searching code for the VEB layout is not amenable to efficient simdization. So, we instead use a non-optimal cache-oblivious layout that enables simdization.

Figure 7(a) 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 7(b) shows the corresponding search code. We divide the input queries into a number of 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`, `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. Note that this is an example of tuning the distribution of work over thread-level, instruction-level, and SIMD-level parallelism.

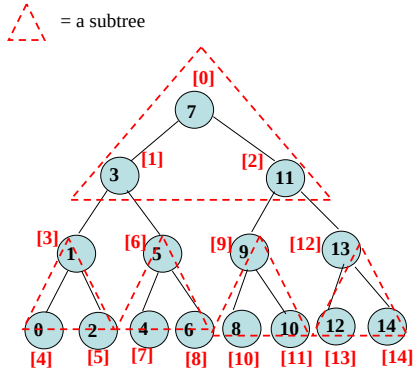
3.3 Sorting

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, *quicksort* separates elements into two categories before recursively 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-flow that is not 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 mergesort—a subset of the techniques described in [8], reimplemented using our synergetic approach.

1. **Merge sort** exposes task parallelism at a coarse granular-

(a) Cache-oblivious tree layout



(b) Parallelized and simdized query search code

```
#pragma isat tuning scope(start_scope, end_scope) measure(start_timing, end_timing)
                    variable(SUBTREE_HEIGHT, [4,6,8,12])
#pragma isat tuning scope(start_scope, end_scope) measure(start_timing, end_timing)
                    variable(BUNDLE_SIZE, (8,64,1)) variable(VLEN, (4,64,4)) search(dependent)
```

autotune SUBTREE_HEIGHT, BUNDLE_SIZE, and VLEN

```
void ParallelSearchForCacheOblivious() {
    int numNodesInSubTree = (1 << SUBTREE_HEIGHT) - 1;
    int bundleSize = BUNDLE_WIDTH * VLEN; int remainder = numQueries % bundleSize;
    int quotient = numQueries / bundleSize; int numBundles = ((remainder==0)? quotient : (quotient+1));

    cilk_for (int b=0; b < numBundles; b++) {
        int q_begin = b * bundleSize; int q_end = MIN(q_begin+bundleSize, numQueries);
        for (int q = q_begin; q < q_end; q += VLEN) {
            int searchKey[VLEN] = Queries[q:VLEN]; int* array[VLEN] = Keys;
            int subTreeIndexInLayout[VLEN] = 0; int localAnswers[VLEN] = 0;
            for (int hTreeLevel=0; hTreeLevel < HierTreeHeight; ++hTreeLevel) {
                int i[VLEN] = 0;
                for (int levelWithSubTree = 0; levelWithSubTree < SUBTREE_HEIGHT; ++levelWithSubTree) {
                     int currKey[VLEN];
                     for (int k=0; k<VLEN; k++)
                         currKey[k] = (array[k])[i[k]];
                     bool eq[:] = (searchKey[:] == currKey[:]);
                     bool lt[:] = (searchKey[:] < currKey[:]);
                     localAnswers[:] = eq[:]? 1: localAnswers[:];
                     i[:] = localAnswers[:]? i[:]: ((lt[:])?  (2*i[:]+1): (2*i[:]+2));
                }
                int whichChild[VLEN] = i[:] - numNodesInSubTree;
                subTreeIndexInLayout[:] = localAnswers[:]? subTreeIndexInLayout[:] :
                                 (subTreeIndexInLayout[:]<<SUBTREE_HEIGHT + whichChild[:] + 1);
                array[:] = localAnswers[:]? array[:] :
                                 (Keys + subTreeIndexInLayout[:] * numNodesInSubTree);
            }
            Answers[q:VLEN] = localAnswers[:];
        }
    }
}
```

each thread processes a bundle of queries

Simdized with Array Notation;
each SIMD lane processes a query

Figure 7: Optimizing the search with cache-oblivious layout and array notation

ity. We augment the basic algorithm to make the merge-step (as well as the recursive step) parallel[4]. 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 quicksort) allowing independent, recursive processing of all elements under, and all elements 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 basecase size determined by auto-tuning.

2. **Bitonic merge networks** are used to expose ILP and enable SIMD when merging two sorted subsequences. A bitonic merge network of size 2^N has a $N-1$ stages, each stage comparing and swapping elements at decreasing distances. The number of comparisons in each stage is the same, but to simdize the computation, 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 minimum leading element is taken from the head of a sequence being merged. The chunk is mixed with left-overs 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 left-overs. Chunk size is auto-tuned.

3. **In-register sort** via **sorting network** (finest grain): This 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 the above algorithms, a number of implementation difficulties arise. The sorting networks described above rely heavily on *permuting* vectors. Permutation code is not currently amenable to automatic compiler-based simdization. However, array notation allows arbitrary permutations (automatically generating shuffle instructions for the target machine) if permutations are known at compile time.

Unfortunately, while arbitrarily-sized bitonic and odd-even networks mentioned above can be implemented by simple recursive functions, only when those functions are *executed* at a given size will the permutations become apparent. 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[5]. We use a complementary technique to auto-tuning that we call *lightweight code generation*. The idea is that whenever a computation kernel is needed at different sizes or configurations for auto-tuning or portability purposes, we write a very 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), very little work is required to build simple code generators in any high-level language (e.g. Python, Haskell, etc). In this case, we wrote 86 lines of non-comment, non-blank 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.

[4]This algorithm appears first in [1]

[5]Intel®Array Building Blocks is an appropriate framework for staged code generation in this example.

4. EVALUATION

We now report our experimental results. Table 1 shows our experimental setup. Table 2 shows the details of our benchmarks. They are important throughput computing kernels that are also used by other researchers [18, 19].

Figure 8 shows our overall performance results, where the benchmarks plus their average are on the x-axis and the performance in GFLOPS is on the y-axis (in log scale). The serial cases were compiled with the `-fast` option in ICC, which generally produces the best performing code. In each benchmark, we show four bars. The first one is simple loop-based parallelization in Cilk. The second bar is the case with cache-oblivious parallelization. The third bar has both cache-oblivious parallelization and compiler-based simdization. Finally, the last bar uses cache-oblivious parallelization, simdization, and autotuning together.

As shown in Figure 8, simple loop-based parallelization achieves a 4.8x speedup on average, which is not bad given an 8-core machine. Nevertheless, cache-oblivious techniques improve the average speedup to 10.7x, more than doubling the performance. Note that this apparently superlinear speedup is a result of improved cache locality. Their impacts are particularly large in `LBM`, `Search`, and `MatrixMultiply`. Adding simdization improves the average speedup to 17.3x, especially helping `MatrixMultiply` 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 GFLOPS is 15.6, which is still far below the machine's peak GFLOPS of 145.3, indicating that we are largely limited by the memory latency.

Architecture	Intel®Nehalem
Core Clock	2.27 GHz
Number of Cores	8 cores (on two sockets)
Memory size	12 GB
Memory Bandwidth	22.6 GB/s
Compiler	ICC v12, "-fast" option
OS	64-bit CentOS v4

Table 1: Experimental Setup

Benchmark	Description	Problem Size
3dfd [19]	3D finite difference computation	x=1000, y=1000, z=1000, t=20
Bilateral [18]	Bilateral image filtering	8K x 8K pixels
LBM [13]	Lattice Boltzman Method	The reference input
MatrixMultiply [19]	Dense matrix multiplication	Dimensions = 4kx4k
Search [18]	Searching a binary tree	24-level tree, 4M queries
Sort [18]	Sorting	16M elements

Table 2: Benchmarks

To get an idea of how well our results compared against highly-tuned codes, Figure 9 compares the performance of single-precision matrix multiplication between our approach and the Intel®Math Kernel Library (MKL v.11), while Figure 10 compares sorting performance between our approach and the Intel®Integrated Performance Primitives (IPP v.7) It is quite encouraging that our high-level approach achieves comparable or better performance than highly-tuned library codes.

Figure 11 shows the performance of LBM with various optimization strategies. The serial case (first bar) achieves only 1.7

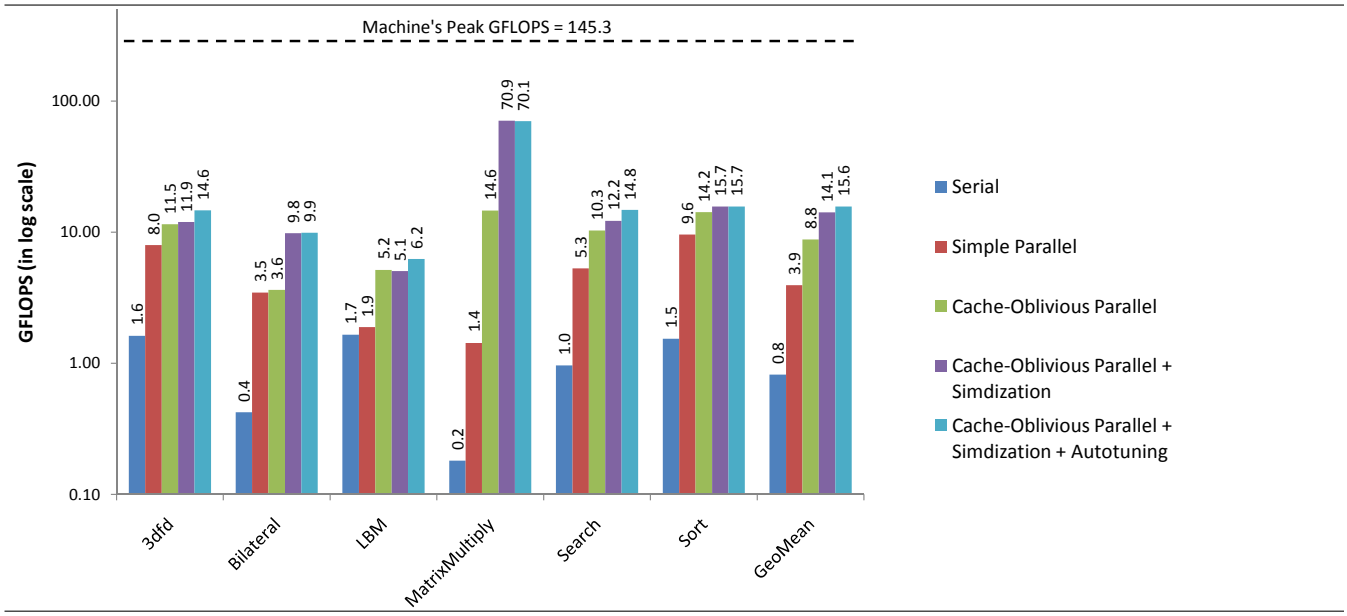


Figure 8: Performance results of our approach

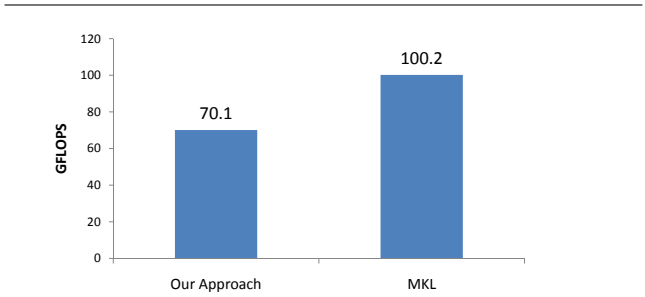


Figure 9: Performance of single-precision matrix multiplication.

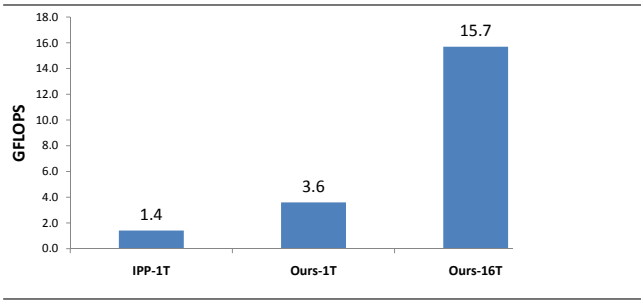


Figure 10: Performance of sorting (IPP-1T=Intel® Integrated Performance Primitives v7 with one thread, Ours-1T=our approach with one thread, Ours-16T=our approach with 16 threads. IPP currently doesn't support multithreaded sorting.).

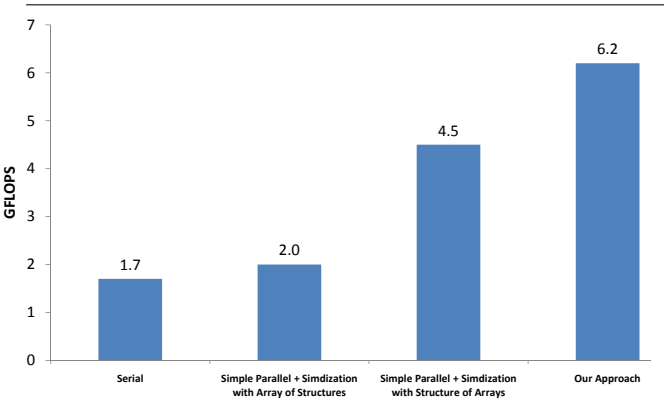


Figure 11: Performance of Lattice Boltzman Method.

GFLOPS. Applying simple loop-based parallelization and simdization (the second bar) improves performance by only 17%, as 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. Finally, 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 benchmark `Search` 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 non-oblivious choice.

5. RELATED WORK

A recent study by Lee *et. al.* [18] compared the performance of a number of computing kernels on CPU and GPU and found that the GPU is only 2.5x faster than the CPU on average. 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.

Cache-oblivious techniques were studied in the past mostly for algorithmic analysis and serial processing [10, 11, 17]. Our work shows that cache-oblivious techniques can also work well in practice on multicore processors. On the other hand, autotuning has recently become a hot research topic [2, 6, 9, 12, 15, 16, 23, 24,

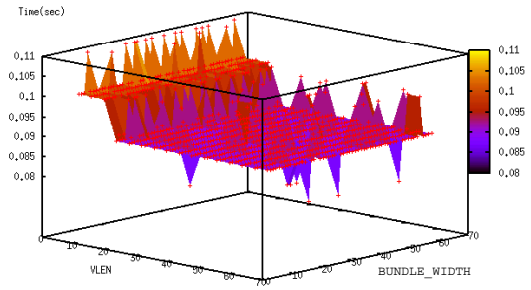


Figure 12: Performance impact of autotuning on benchmark `Search`.

25]. In particular, Datta *et. al.* [9] show that a pure autotuning-based approach can effectively optimize stencil computation. Our approach differs from theirs by using cache-oblivious techniques instead of explicit blocking, though we still use autotuning to tune other parameters and the basecase. By using this hybrid approach, we reduce the amount of tuning needed. In addition, our work covers not only stencil computations but also other domains like sorting and searching.

6. CONCLUSION

We have developed a synergetic approach to throughput computing for the x86-based multicores. Our approach uses cache-oblivious techniques to divide a large problem into subproblems that can be executed in parallel. A subproblem can then be simdized using the rich simdization support available in compilers like the Intel compiler. We also demonstrate the use of autotuning to guide the tuning steps throughout the process. Overall, our approach achieves a nearly 20x speedup over the best serial case on a dual-socket quad-core Nehalem.

Intel®Compiler can be purchased at `http://software.intel.com/en-us/intel-compilers`. Intel®Software Autotuning Tool is freely available at `http://software.intel.com/en-us/whatif`.

7. 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 allowing us to use his SDE tool.

8. REFERENCES

[1] AKL, S. G., AND SANTORO, N. Optimal parallel merging and sorting without memory conflicts. *IEEE Trans. Comput. 36*, 11 (1987), 1367–1369.

[2] BAILEY, D., CHAME, J., CHEN, C., DONGARRA, J., HALL, M., HOLLINGSWORTH, J., HOVLAND, P., MOORE, S., SEYMOUR, K., SHIN, J., TIWARI, A., WILLIAMS, S., AND YOU, H. PERI Auto-Tuning. *Journal of Physics: Conference Series (SciDAC 2008) 125*, 1 (2008).

[3] BENDER, M. A., DEMAINE, E. D., AND FARACH-COLTON, M. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science* (2000), pp. 399–409.

[4] BIK, A. J. *The Software Vectorization Handbook*. Intel Press, December 2006.

[5] BLUMOFE, R. D., JOERG, C., B. C, K., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (1995), pp. 207–216.

[6] BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. PLUTO: A Practical and Fully Automatic Polyhedral Program Optimization System. In *Proceedings of the ACM SIGPLAN 08 Conference on PLDI* (June 2008).

[7] CHANDRA, R., DAGUM, L., KOHR, D., MAYDAN, D., MCDONALD, J., AND MENON, R. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[8] CHHUGANI, J., NGUYEN, A. D., LEE, V. W., MACY, W., HAGOG, M., CHEN, Y.-K., BARANSI, A., KUMAR, S., AND DUBEY, P. Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endow. 1*, 2 (2008), 1313–1324.

[9] DATTA, K., MURPHY, M., VOLKOV, V., WILLIAMS, S., CARTER, J., OLIKER, L., PATTERSON, D., SHALF, J., AND YELICK, K. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (2008).

[10] FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. Cache oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (October 1999).

[11] FRIGO, M., AND STRUMPEN, V. Cache Oblivious Stencil Computations. In *Proceedings of the 2005 International Conference on Supercomputing* (2005).

[12] HALL, M., CHAME, J., CHEN, C., FISCHER, P., AND HOVLAND, P. Autotuning and Specialization: Speeding up Nek5000 with Compiler Technology. In *Proceedings of the Internation Conference on Supercomputing* (June 2010).

[13] HENNING, J. L. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News 34* (2006).

[14] INTEL. *Using Parallelism: (CEAN) C/C++ Extension for Array Notation*, Mar 2010.

[15] KAMIL, S., CHAN, C., OLIKER, L., SHALF, J., AND WILLIAMS, S. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)* (2010).

[16] KAMIL, S., CHAND, C., WILLIAMS, S., OLIKER, L., SHALF, J., HOWISON, M., BETHEL, E. W., AND PRABHAT. A Generalized Framework for Auto-tuning Stencil Computations. In *Proceedings of CUG: Cray User Group Conference* (2009).

[17] KUMAR, P. Cache oblivious algorithms. *LNCS 2625*, 193–212.

[18] LEE, V. W., KIM, C., CHHUGAN, J., DEISHER, M., KIM, D., NGUYEN, A. D., SATISH, N., SMELYANSKIY, M., CHENNUPATY, S., HAMMARLUND, P., SINGHAL, R., AND DUBEY, P. Debunking the 100X CPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proc. 2010 ISCA*.

[19] NVIDIA. *CUDA SDK*. http://www.nvidia.com/object/cuda_get.html.

[20] PROKOP, H. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.

[21] REINDERS, J. *Intel Threading Building Blocks*. O'Reilly, July 2007.

[22] STRASSEN, V. Gaussian elimination is not optimal. *Numer Math 13* (1969), 354–356.

[23] VUDUC, R., DEMMEL, J., AND YELICK, K. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005, Journal of Physics: Conference Series* (June 2005).

[24] WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing 27*, 1-2 (2001), 3–35.

[25] YI, Q., SEYMOUR, K., YOU, H., VUDUC, R., AND D. QUINLAN, B. . POET: parameterized optimizations for empirical tuning.