A STUDY OF ARCHITECTURAL OPTIMIZATION METHODS IN BIOINFORMATICS APPLICATIONS

G. Tan
L. Xu
Z. Dai
S. Feng
N. Sun

KEY LABORATORY OF COMPUTER SYSTEM AND ARCHITECTURE, INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES, BEIJING, CHINA
(TGM@NCIC.AC.CN)

Abstract

Studies in the optimization of sequence alignment have been carried out in bioinformatics. In this paper, we have focused on two aspects: memory usage and execution time. Our study suggests that cache memory does not have a significant effect on system performance. Our attention then turns to optimize Smith–Waterman's algorithm. Two instruction level methods have been proposed and 2–8 fold speed improvements have been observed after the optimization has been implemented. Further improvements on system performance have been achieved by overlapping computation with system I/O usage.

Key words: bioinformatics, sequence alignment, cache memory, instruction level parallelism, I/O overlapping

The International Journal of High Performance Computing Applications, Volume 21, No. 2, Summer 2007, pp. 000–000
DOI: 10.1177/1094342006078175

Figures 1–3 appear in color online: http://hpc.sagepub.com

1 Introduction

The peak performance of modern computers evolves rapidly with Moore's law. It is, however, increasingly hard to harness the peak performance because the computer system is becoming more complex. As a consequence, good implementations get less peak performance than before, because the systems are more complex. Many solutions in the field of numerical computation were formulated in order to minimize this performance gap. A number of high performance library routines are carefully constructed and hand-tuned. Many optimization methods used by these libraries are based on computer architecture, as the performance of most numerical kernels is often determined by the number of operations required to handle a cache miss, which may be 10–100 times more expensive than a multiplication operation. More generally, the performance of numerical code depends crucially on the use of the platform's memory hierarchy, register sets and instruction sets. For example, the optimization methods for matrix–matrix multiplication include decreasing the number of operations with divide and conquer algorithms, using blocked algorithms on cache memory, and utilizing special instruction sets. Because hand-tuned approaches are time consuming and require extensive knowledge of the algorithm and architecture, many automatic optimization tools have been developed. One important example is ATLAS (Demmel et al. 2005), a linear algebra library generator. Some more recent tools are FFTW (Frigo and Johnson 2005), SPIRAL (Puschel et al. 2005), SPARSITY (Im, Yelick and Vuduc 2004) and UHFFT (Mirkovie and Johnsson 2001). However, one should be reminded that many automatic optimization tools were developed with reference to newly formulated optimization methods, so the study of optimization methodology is the basis of these automatic solutions.

Bioinformatics applications are attracting great attention from the high performance computing research community. Sequence alignment (Gusfield 2001), a commonly used process of scanning gene and protein sequence databases, exerts great pressure on the processing capability of current computer systems (Camp, Cofer, and Gomperts 1998). The need for a more efficient process comes from the exponential growth of the biosequence banks: every year their sizes increase by a factor of 1.5 to 2. The scanning operation tries to find similarities between a particular query sequence and all the sequences in the bank. This operation allows biologists to identify sequences sharing common subsequences. From a biological point of view, it leads to the identification of similar functionality. The complexity of the comparison algorithms is quadratic with respect to the length of the sequences. One frequently used approach to speed up this time-consuming operation is to introduce heuristics in search algo-

rithms such as BLAST (Altschul et al. 1990) and FASTA (Galison 2001) at the cost of the quality of results. The importance of the sequence alignment algorithms, which have been the most important kernel of many sequence analysis tools, is analogous to the matrix multiplication algorithms in numerical computation. So it is necessary to study how to efficiently implement them on high performance platforms.

However, while bioinformatics has attracted much attention recently, with a few exceptions little has been done to optimize their performance. For example, as far as we know, the studies of the memory system behavior of bioinformatics applications have not been published before. Bader et al. (n.d.) are developing BioPerf, a suite of representative applications assembled from the computational biology community. While they also use approaches similar to ours to analyze the performance of these applications, the main motivation of their work is to develop a benchmark in bioinformatics to evaluate and optimize high-end computer system architectures.

The rest of this paper is organized as follows: The next section studies the memory system performance of some important bioinformatics programs. Section 3 proposes two optimization methods for the alignment algorithm kernel, and an optimization scheme for a more complex alignment program. Section 4 concludes the paper.

2 Memory System Performance

Memory wall[1] (Henessy and Patterson 1999) has been widely researched because of the high cost of memory accesses and cache misses. Papers studying memory system performance have focused on SPEC benchmarks and, more recently, on commercial workloads or desktop applications[2] (Barroso and Gharachorloo 1998; Lee et al. 1998; Xu et al. 2004). For example, Xu et al. (2004) studied the execution characteristics of multimedia applications on several Linux and Windows platforms. They found that most current multimedia programs will continue to be CPU-bound rather than memory-bound.

In order to study high performance algorithms in bioinformatics, we selected several popular applications to investigate their memory system performance by comparison with that of SPEC2000.[2]

We used hardware performance counters as well as execution-driven simulation in this study. Oprofile[3] is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. PAPI (Performance Application Programming Interface)[4] specifies a standard API for accessing hardware performance counters available on most modern microprocessors. Both tools provide useful information for alleviating commonly occurring bottlenecks in high performance computing. We used these tools to profile program memory behavior. In order to study cache behavior under different configurations, we also performed extensive execution-driven simulation experiments using SandFox[5], a Vmips[6] based executive-driven simulator developed by a group at our institute.

The application set is comprised of six sequence alignment programs: Swat (Smith and Waterman 1981), Blastall (Altschul et al. 1990), Fasta (Galison 2001), MegaBlast (Zhang et al. 2000), Phrap[7], ClustalW (Thompson, Higgins, and Gibson 1994). Swat is a sequence alignment program using the standard Smith–Waterman's dynamic programming model. Fasta, Blastall and MegaBlast are widely used heuristic local sequence similarity search tools.[8] Phrap is a program for assembling DNA sequence data. ClustalW is used to construct evolutionary relationships among divergent sequences by producing biologically meaningful multiple sequence alignments.

We chose the SPEC2000 benchmark to represent traditional applications. The input files were provided by the benchmark suites in *reference* directions. The SPEC2000 and six application programs were compiled statically with an optimization level *O2*.

2.1 Memory References Per Instruction

DNA/protein databases tend to be large in size, therefore processing these data sets might place a high demand on the memory system. In order to compare the memory requirement of bioinformatics applications with that of traditional applications, our first experiment was to measure the number of data memory references per instruction for these programs. A higher value indicates that a program generates more memory traffic, and hence places a higher demand on the memory system.

We ran the programs on an Opteron 1.6 GHz system. Using Oprofile and PAPI, we measured the total number of data memory references and the total number of instructions executed for each program. The results include instructions for both the user code and the OS. We noted that the SPEC2000 and bioinformatics programs used here spend more than 99 percent of their execution time in user mode. As a result, the OS effects can be ignored without affecting accuracy. Figure 1 indicates that while bioinformatics programs use large data banks, they generate only a slightly higher average number of memory references per instruction (0.224) than SPEC2000 (0.213). The reason for such similar behavior is that the workloads are computation intensive.

2.2 Data Cache Performance

An important question is: How well can the cache perform under different programs? To answer this question, we used Oprofile and PAPI to measure the cache performance on an Opteron system with 64 KB L1 data

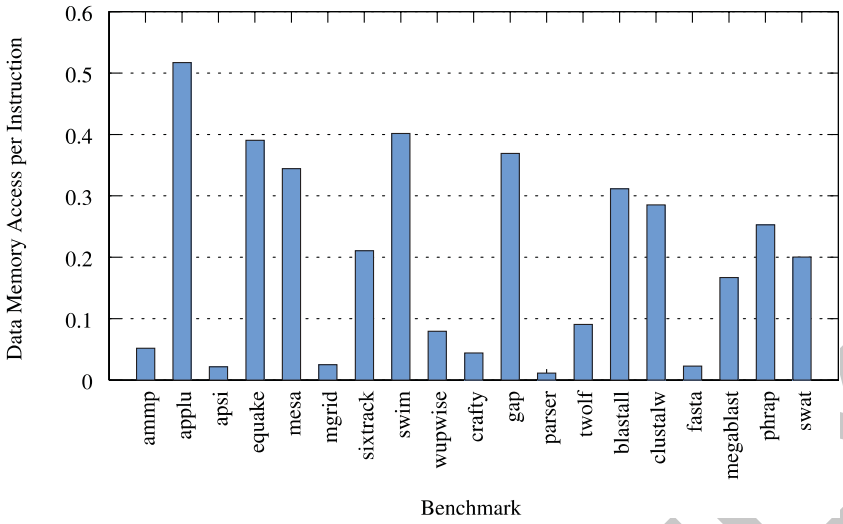

Fig. 1 Number of data memory references per instruction. The average number of memory references per instruction is 0.213 for SPEC2000 (dark bars) and 0.224 for bioinformatics programs.

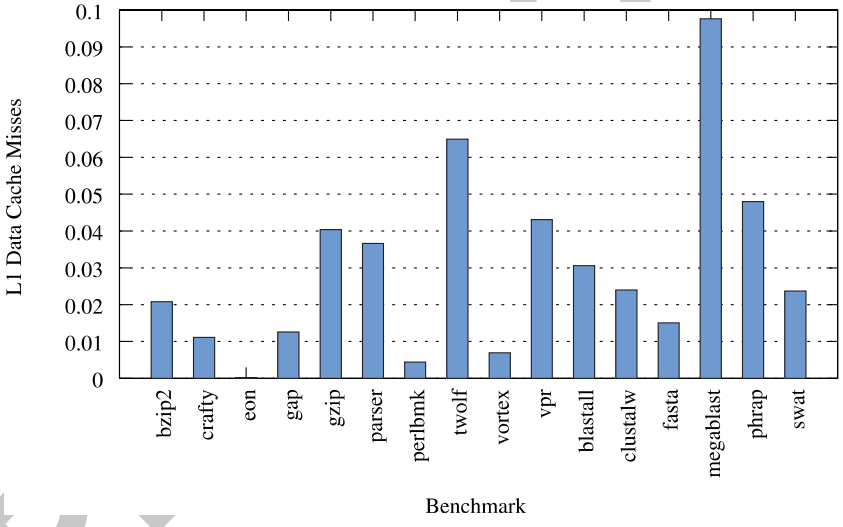

Fig. 2 L1 D-cache performance for SPECint2000 and bioinformatics programs.

cache. Figures 2 and 3 show the measured results for all programs. Figure 2 indicates that the average cache miss rate of the six bioinformatics programs is higher than that of SPECint2000, especially for MegaBlast. Figure 3 also indicates that the cache miss rates of bioinformatics are similar to those of SPECfp2000. Among the six bioinformatics programs, MegaBlast has the highest data cache miss rate. In Section 3, we discuss the reason in detail, and will propose an optimized algorithm to improve its memory and cache performance.

Profiling tools can only measure cache performance under a fixed configuration. To analyze the cache performance of these bioinformatics programs under different memory organizations, we used SandFox to simulate the behavior of these programs comprehensively. Figure 4 shows the simulation results for all the six bioinformatics

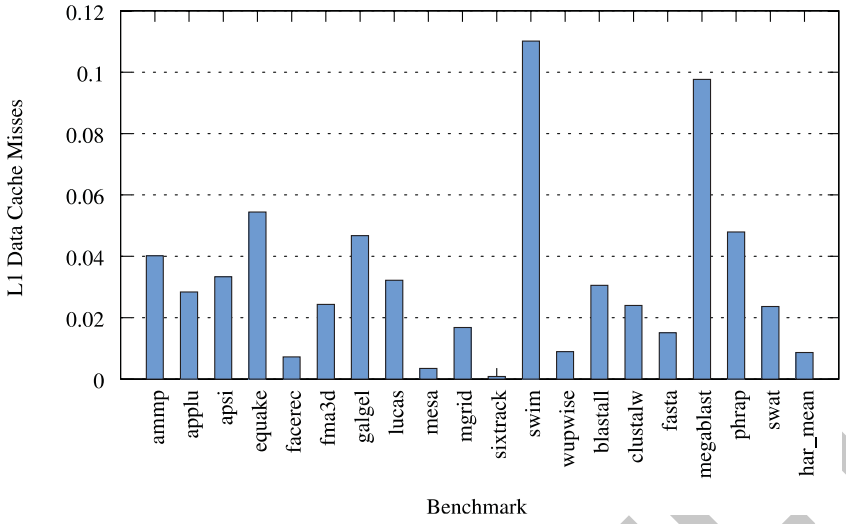

Fig. 3 L1 D-cache performance for SPECfp2000 and bioinformatics programs.

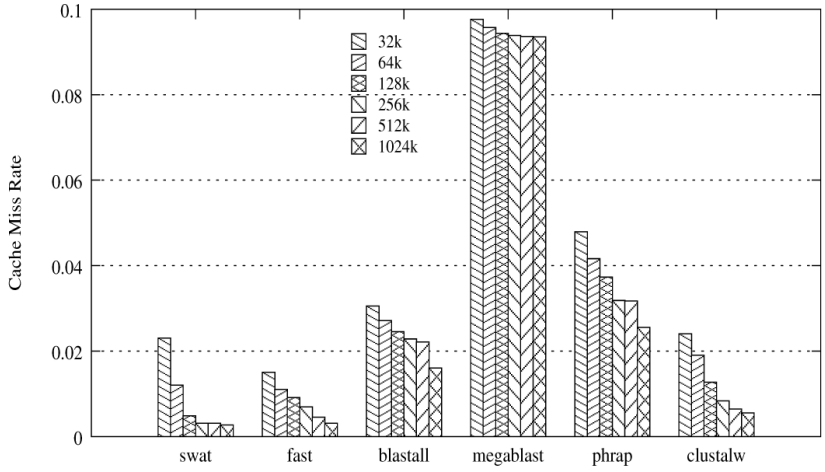

Fig. 4 L1 D-cache performance for bioinformatics programs with different cache sizes.

programs with different cache configurations. The sizes of the cache varied from 32 KB to 1 MB. From Figure 4, we noted that the cache performance of MegaBlast does not significantly improve with larger caches, while the other five programs are more sensitive to cache size increases.

We observed the impact of different cache line sizes and associativities. As shown in Figure 5, caches larger than 256 KB have little benefit. We also simulated the cache behavior with different line size and associativities while fixing the cache size at 256 KB. We observed that except for MegaBlast, the other five programs all show sensitivity to different cache configurations to some extent.

Our observations suggest that the six bioinformatics programs except MegaBlast are memory-bound rather than CPU-bound. Although the cache miss rate of the other five programs is reduced significantly by tuning the configuration at cache memory level, the rates are relatively small so that the overall performances are not

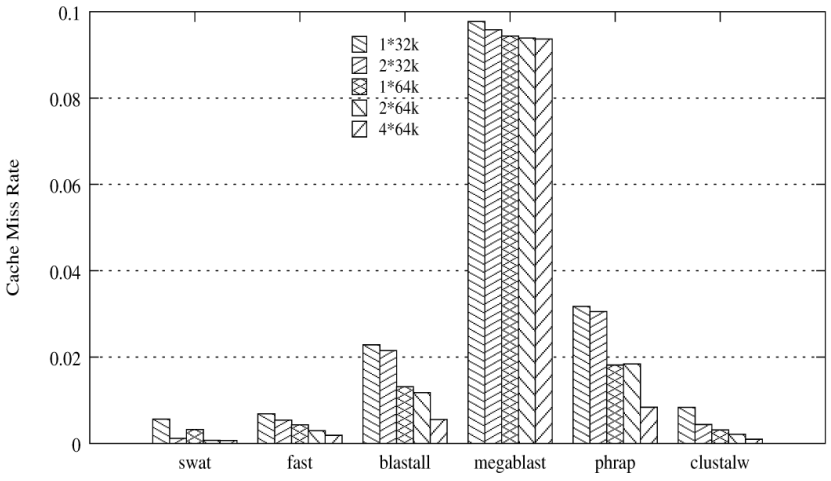

Fig. 5 L1 D-cache performance for bioinformatics programs with different cache line size and associativity (associativity ∗ line size). The size of cache is 256 KB.

expected to be improved. Therefore our focus is to utilize other architecture-based optimization algorithms. Because all the six programs of the bioinformatics application comprise a Smith–Waterman's dynamic programming kernel, we expect that the optimization in instruction level can improve the performance. We will look into reducing the time and space costs, particularly for MegaBlast, through analysis of computational behavior.

3 The Optimized Algorithms

3.1 Instruction Rearrangement in Smith–Waterman's Algorithm

As discussed in the previous section, the benefit of optimizing the cache behavior of Smith–Waterman's algorithm is expected to be limited. Bader performed a comprehensive test using IBM MAMBO simulator and implied that it is possible to optimize this algorithm at instruction levels such as branch mispredictions or special instruction sets.

Smith–Waterman's uses a dynamic programming algorithm to compute the optimal local alignment score. Given a query sequence A of length m, a database sequence B of length n, a substitution score matrix σ, a gap-open penalty q and a gap extension penalty r, the optimal local alignment score t can be computed by the following recursion relations:

$$e_{i,j} = \max\{e_{i,j-1}, h_{i,j-1} - q\} - r \quad (1)$$

$$f_{i,j} = \max\{f_{i-1,j}, h_{i-1,j} - q\} - r \quad (2)$$

$$h_{i,j} = \max\{h_{i-1,j-1} + \sigma(A[i], B[j]), e_{i,j}, f_{i,j}, 0\} \quad (3)$$

$$t = \max\{h_{i,j}\}$$

Here, $e_{i,j}$ and $f_{i,j}$ represent the maximum local alignment score involving the first i symbols of A and the first j symbols of B, and ending with a gap in sequence B or A, respectively. The overall maximum local alignment score involving the first i symbols of A and the first j symbols of B, is represented by $h_{i,j}$. Starting with $e_{i,j} = f_{i,j} = h_{i,j} = 0$ for all $i = 0$ or $j = 0$, the value of any cell cannot be computed before the value of all cells to the left and above it has been computed.

The Smith–Waterman's dynamic programming algorithm above can be easily implemented. The directly implemented program is called a naive program. As shown in equations 1, 2 and 3, the naive program needs a number of branch instructions, which result in high costs on current architecture. As long as h is less than the threshold $q + r$, e or f will remain at zero along a column or row in the dynamic programming matrix. So the branches used to choose whether e or f contributes to h occur in many missed predictions. Therefore, it is necessary to decrease branch operations and mispredicted branches.

In fact, equation 3 can be divided into two parts:

$$h'_{i,j} = \max\{h_{i-1,j-1} + \sigma(A[i], B[j]), e_{i,j}, 0\} \quad (4)$$

$$h_{i-1,j} = \max\{h'_{i-1,j}, f_{i-1,j}\} \quad (5)$$

Combining equation 2, we have

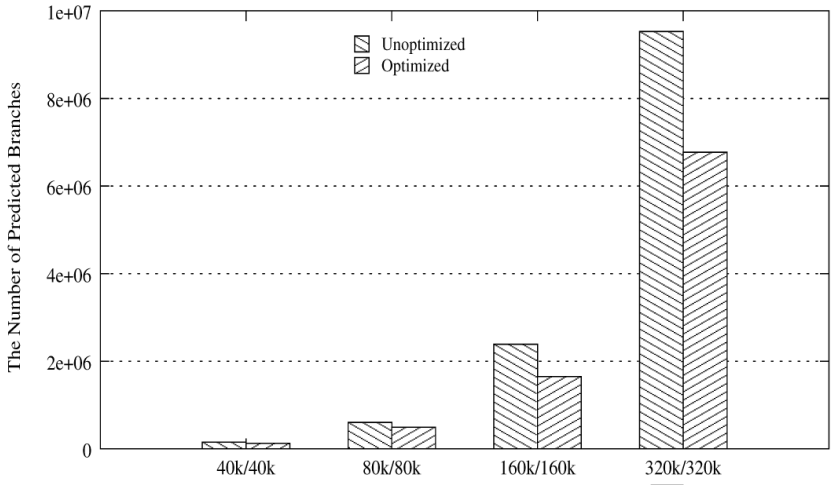

Fig. 6 Comparison of the number of predicted branches.

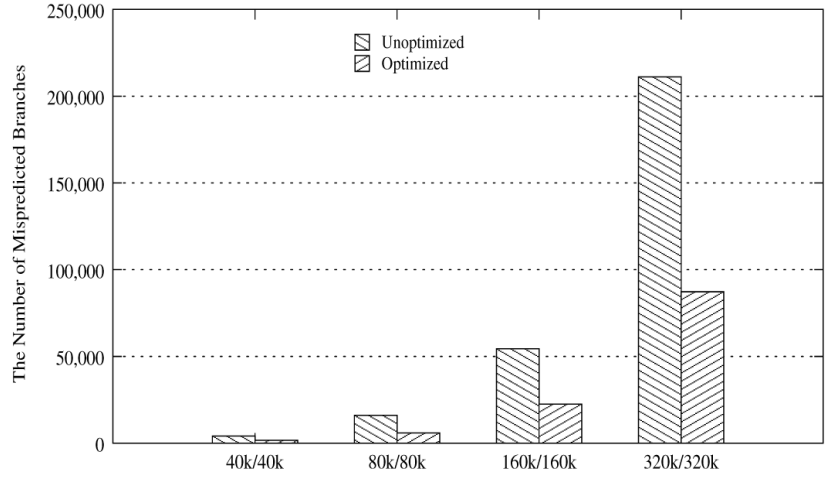

Fig. 7 Comparison of the number of mispredicted branches.

$$f_{i,j} = \max\{f_{i-1,j} - r, h'_{i-1,j} - q - r\}$$

For a fixed j, setting $p_i = f_{i,j} + i * r$ and $w_i = i * r$, then

$$p_i = \max\{p_{i-1}, h'_{i-1,j} - q - r + w_i\} \quad (6)$$

So setting

$$z_i = h'_{i-1,j} - q - r + w_i \quad (7)$$

we have

$$p_i = \max\{p_{i-1}, z_i\} \quad (8)$$

and

$$f_{i,j} = p_i - i * r \quad (9)$$

3.1.1 Reducing mispredicted branches Through the transformation of formulations above, the calculations of e and f are separated. The two calculation parts can be placed in two *while* loops until e or f equals zero. Once either of them is zero, it will be zero in the next align-

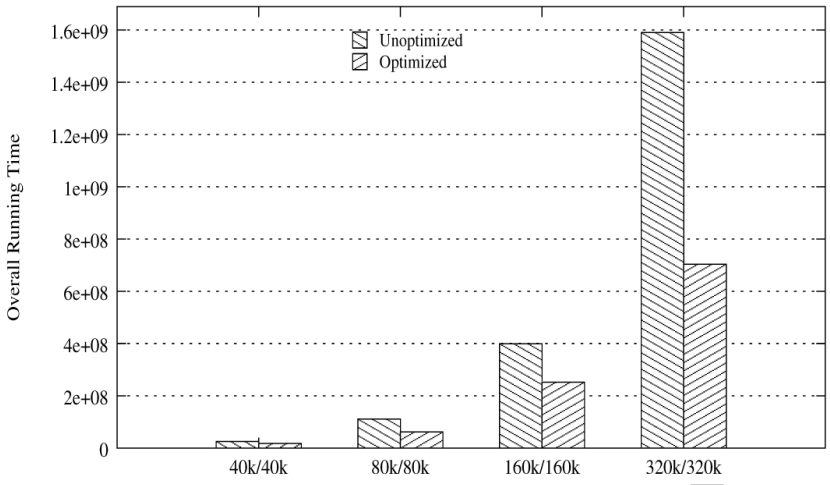

Fig. 8 Comparison of the overall running time.

ment step because of the gap penalty. Thus, the number of branches is greatly decreased. Accordingly, because the values of *e* or *f* keep still in a *while* loop, the predicted branches can promise correctness.

3.1.2 Using SSE2 instruction sets Microparallelism is an efficient method to speed up numerical kernels because there are many vector operations (Demmel et al. 2005; Frigo and Johnson 2005). Using the SIMD instruction set to improve Smith–Waterman's algorithm has been widely studied. Alpern, Carter, and Gatlin (1995) presented several ways to speed up the algorithm including a parallel implementation utilizing microparallelism by dividing 64-bit wide Z-buffer registers of the Intel Paragon i860 processors into four parts and achieved more than a 5-fold improvement over a conventional implementation. Wozniak (1996) presented a way to implement the algorithm using the Visual Instruction Set technology of Sun UltraSPARC microprocessors and obtained a speedup of about two relative to the same algorithm implemented with integer instructions on the same machine. Rognes and Seeberg (2000) presented an implementation of the optimized Smith–Waterman's algorithm for special cases, using Intel's MMX instruction set and represented a speedup of six with the SWAT-optimizations.

However, some of the more efficient methods listed above parallelize the computation along a diagonal. One of the frequently used technologies is wave-front parallel computation. One should note that the data in the dynamic programming matrix must be rearranged so that cache misses resulting from abnormality of input data along a diagonal can be avoided. In addition, although there is no dependence among cells in the diagonals, load unbalancing is a significant problem with this method because of the different lengths of the diagonals. In our method, the transformed dynamic programming formulations are used so that the SSE2 program computes along a matrix row instead of a diagonal.

In the SSE2 instruction set, the data length is 128 bits. We assume that the vectors $e_{(t,\ldots t+15),j}$, $h_{(t,\ldots t+15),j}$ and $f_{(t,\ldots t+15),j}$ need to be updated. Once equation 1 finishes calculating $e_{(t,\ldots t+15),j}$, vector f and h can be computed using equations 5 and 9, respectively. The order of calculation proceeds by matrix column. The cells in matrix h only depend on the cells to the left and above them, the cells in matrix e only depend on the cells to the left. Because the eight cells located in one column have no inter-dependence, it is easy to parallelize the computation of matrix e and h. By the prefix computing above, we eliminate some data dependence and can efficiently compute matrix f. The parallel algorithm is described as Figure 10.

Although the inter-dependence in calculating f is not eliminated completely, only two arithmetic operations and a few logic operations are needed in calculating the 16 cells of matrix f. In the innermost loop, there are only two logic operations and one arithmetic operation. When implementing this algorithm, we perform a loop unroll for the innermost loop.

3.2 High Spatial–Temporal Efficient MegaBlast

Although comprising the same dynamic programming computational kernel, MegaBlast has abnormally high

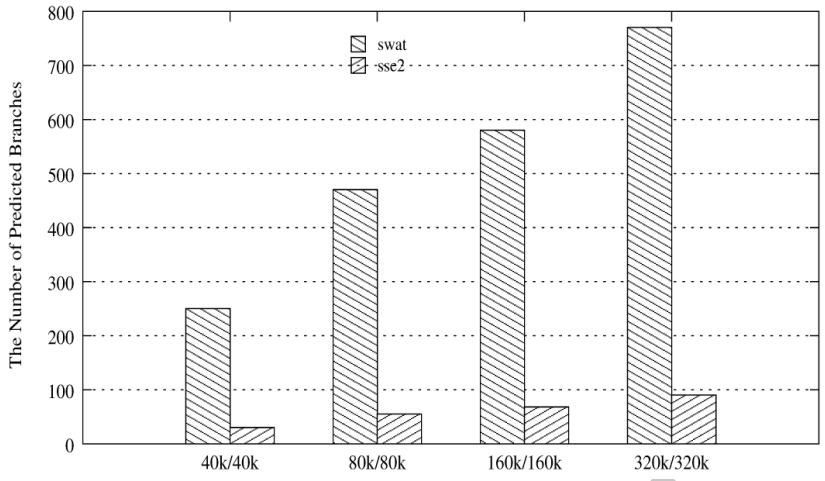

Fig. 9 Comparison of running times of the SSE2 program and the optimized serial algorithm in the previous section.

```
For (j = 1; j ≤ n; j++)
/* Calculating the j column score value */
   For(i = 1; i ≤ m/16;i++)
      /* Calculating a group composed of 16 cells.*/
      /*The matrix e, h, z, p, f and*/
      /*max16 are 16 dimensions vector. */
      Calculating vector e using equation 1;
      Calculating vector h' using equation 4;
      Calculating vector z using equation 7;
      Calculating vector p using equation 8;
      Calculating vector f using equation 9;
      Calculating vector h using equation 5;
      Updating the maximum score value vector max16;
   endfor
endfor
```

Fig. 10 Algorithm 1: The parallel SSE2 implementation for Smith–Waterman's algorithm.

cache miss rates compared with other programs, as shown in Figure 2. An analysis of its program flow is necessary in order to understand its computational behavior.

3.2.1 The original MegaBlast MegaBlast utilizes batch processing and a greedy algorithm to extend significantly similar segments. As a result, it is the fastest and highest-throughput program in the NCBI BLAST toolkits.[8] The basic flow of MegaBlast is the same as other programs in NCBI BLAST. The goal of the first step is to quickly locate ungapped similar regions in both the query sequence and the subject sequence. It builds a hash table based on the query sequence. Each entry in the hash table locates a word with length w in the query sequence. A hit is made with one or several successive pairs of similar words, and characterized by its position in each of the two sequences. All possible hits between the query and database (subject) are calculated in this fashion. Every hit that has been generated is then extended. If an extended segment pair's score is the same as or better than S (a parameter of the program), it is kept and called an *HSP* (highest scoring segment pair). The *HSP* with the highest score is called the *MSP* (maximal segment pair). In the end, the program outputs the *MSP*s with some *p-value* (an expectation value), which is the probability to get at least one score equal to or greater than that of the *MSP* by chance; i.e. the *p-value* is the probability that one or more *MSP* exists, obtained from the comparison of two random sequences (having same length and composition as the sequences of interest) whose score is equal to or greater than that of the *MSP* obtained with the actual sequences. Assume that for a set of query sequences $Q = \{q_1, q_2, \ldots, q_m\}$ and subject sequences $S = \{s_1, s_2, \ldots, s_n\}$, the length of word or hit is w. The MegaBlast algorithm is described in Figure 12.

There are two problems in the MegaBlast algorithm. First, because *HSP*s are selected from the alignment results, which are generated by aligning one sequence with all database sequences, all alignment results are kept in memory until the search for all subject sequences is finished. In the worst case, the memory cost is proportional to the product of the sizes of two sequence sets being compared.

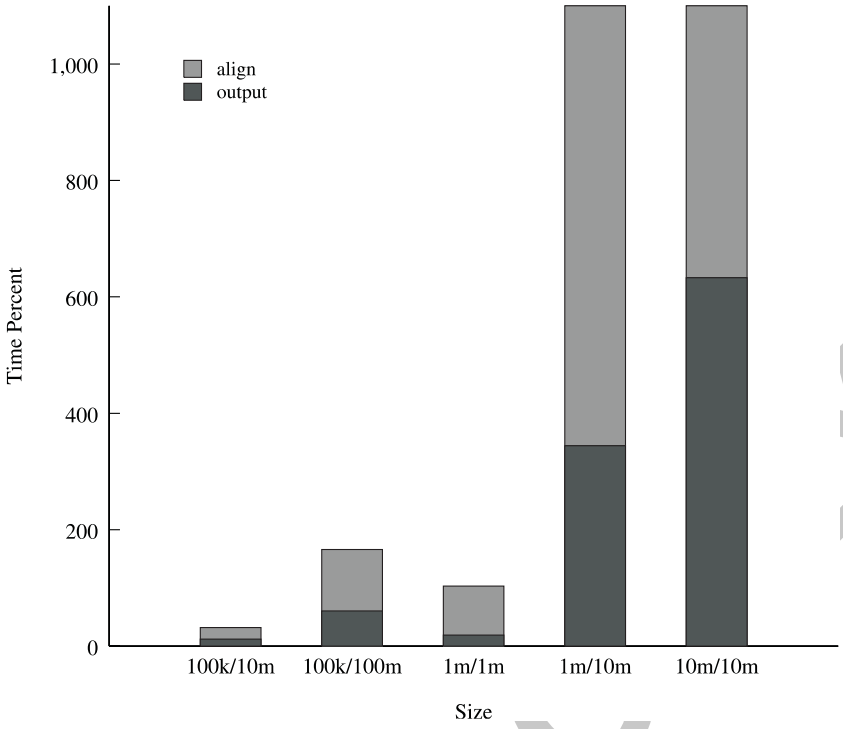

Fig. 11 The time distribution of the MegaBlast program. The sizes of query sequence files are 100 KB, 1 MB and 10 MB. The sizes of subject files are 1 MB, 10 MB and 100 MB. We measure the times for the alignment process and for outputing the results to disk file. It shows that the output time is the primary part of the overall time.

```
Buildhashtable(Q); /*build a hash table for all query
        sequences, which is considered as one sequence*/
for s_i in S /*all sequences in database*/
    Findseed(hashtable, s_i); /*find hits*/
    Extendfilterseed( ); /*Extending hits and get HSPs, */
    Sorthsp( ); /*insert HSPs into some priority queue*/
endfor
Output( ); /*output alignment results*/
```

Fig. 12 Algorithm 2: The original MegaBlast algorithm.

$$Memory_1 = O(c_1 * |Q| * |S|) \quad (10)$$

where $|Q|$ and $|S|$ are the sizes of two sets of sequences and c_1 is a relative parameter judging similarity between two sequences. We note that c_1 is larger when the similarity is higher. When the size of the sequences set is larger and the similarity is higher, the memory cost will increase. To illustrate the problem, we conducted an experiment with some randomly selected *mouse embryo* EST sequences. The results show that, when the size of the query sequence set is 1 MB and the size of the database sequence set is 100 MB, the memory requirement exceeded the available physical memory (2 GB),

The second problem with MegaBlast is that the program is both computational and I/O intensive. The CPU is idle when the alignment results are written to disk. Figure 11 shows the distribution of time between computation and I/O. The high requirement of memory and the high I/O load are the two main bottlenecks of MegaBlast, and we propose a solution to those bottlenecks.

3.2.2 The optimized MegaBlast We note that the query sequences and subject sequences are symmetric and exchanging query and database sequences can still generate correct alignment results, so we exchange the set of query sequences with the set of database sequences and build a hash table based on the database sequences. The optimized algorithm (ste_blast) is described in Figure 13.

The optimized algorithm places *Output* into the *for* loop, which actually eliminates the accumulation of memory requirements. The cost of memory is proportional to the size of subject sequences set, instead of the product of the sizes of the two sequence sets:

```
Buildhashtable(S) /* build a hash table for all database
      sequences,which is considered as one sequence */
for q_i in Q /*all query sequences*/
   Findseed(hashtable, q_i) /*find hits*/
   Extendfilterseed( ); /*Extending hits and get HSPs, */
   Sorthsp( ); /*insert HSPs into some priority queue*/
   Output( ); /*output the alignment of q_i */
endfor
```

Fig. 13 Algorithm 3: The optimized MegaBlast algorithm.

$$Memory_2 = O(c_2 * |S|) \quad (11)$$

where $|S|$ is the size of subject sequence set and c_2 is a similar parameter to c_1 in equation 10. The proportion of memory requirement is:

$$Memory_1/Memory_2 = O(c_1 * |Q|/c_2) \quad 0 < c_1/c_2 \leq 1 \quad (12)$$

When each alignment generates the same size of results, $c_1/c_2 = 1$ and the maximum of two algorithms is the same and the optimized algorithm achieves the best performance. When the size of one sequence's alignment results is far larger than the other sequence's, the memory requirement of optimized algorithm is determined by the maximum cost of memory and achieves the lowest performance gain.

The optimized MegaBlast program outputs alignment results as soon as one query sequence finishes aligning. The task of sending results to disk is managed by the I/O subsystem and the CPU continues aligning the next query sequences. Therefore the optimized algorithm overlaps computation with I/O, decreasing the overall execution time.

3.2.3 Experimental results We measured the performance of the optimized algorithm on a platform with a 1.6 GHz Opteron and 4.5 GB memory. As shown in Figures 14 and 15, the optimized MegaBlast greatly reduces the running time and memory usage.

For the alignment between a 10 MB query and a 10 MB database sequence set, when the sizes of two sequence sets are comparable, the optimized algorithm shows superiority. From equation 12, the value c_1/c_2 is less than 1. However, when the size of the subject sequence set is much larger than the size of the query sequence set, the hash table in the optimized algorithm is larger than the original MegaBlast algorithm. In the case of the 1 MB query sequence set and the 10 MB database sequence set, the original MegaBlast algorithm needs 129 MB memory and the optimized algorithm needs 389 MB for building hash table. Because the whole hash table is kept in memory, the value $Memory_1/Memory_2$ is not linearly proportional to the size of query sequence set. Another factor determining the value $Memory_1/Memory_2$ is the memory-access mode in the program. Both algorithms read their own subject sequence set through memory-mapped files. When the size of the subject sequence set is larger than the size of query sequence set, the original MegaBlast algorithm only allocates physical memory for part of the whole database of sequences, but the whole database of sequences are kept in physical memory as a hash table. Furthermore,

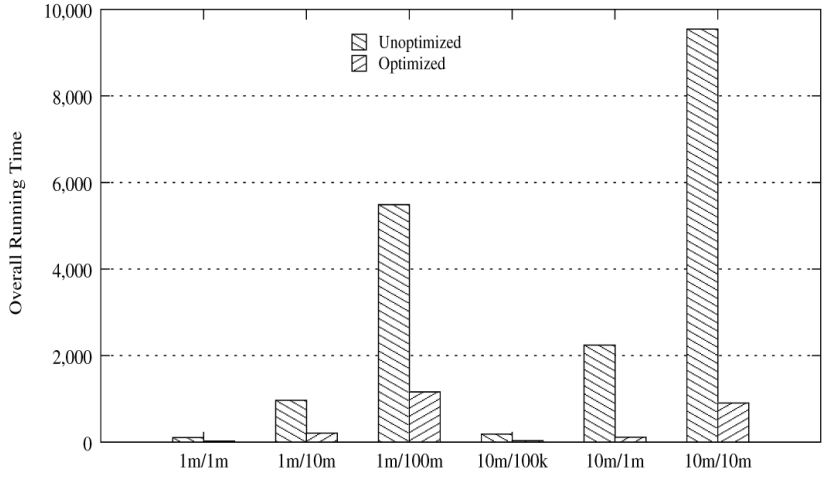

Fig. 14 Comparison of runtimes in seconds of two BLAST algorithms. Each right column represents the original MegaBlast algorithm. For 1 MB/1 MB, the optimized algorithm is 20 seconds and MegaBlast algorithm is 103 seconds. For 10 MB/100 KB, the time is 30 seconds and 180 seconds, respectively.

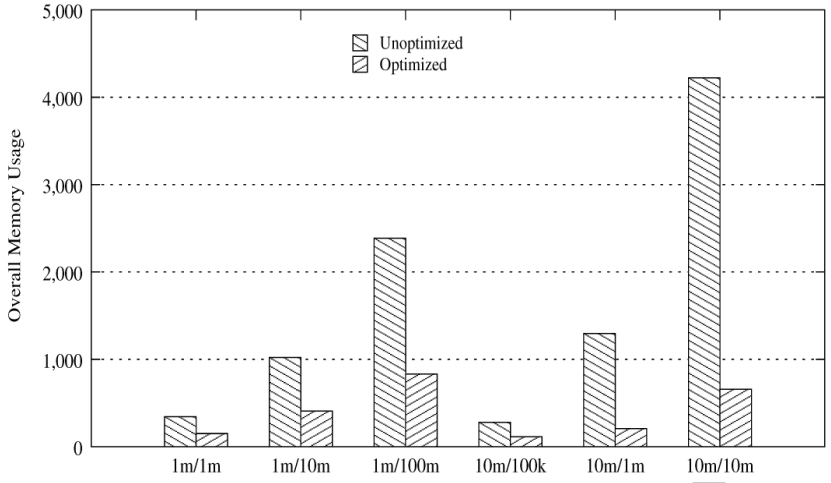

Fig. 15 Comparison of memory in MB of two BLAST algorithms. Each right column represents the original MegaBlast algorithm. For 1 MB/1 MB, the optimized algorithm is 143 MB and MegaBlast algorithm is 342 MB. For 10 MB/100 KB, the memory usage is 112 MB and 277 MB, respectively.

because building the hash table needs more time, these factors also make the running time of the optimized algorithm longer. For example, with the alignment of 1 MB query and 10 MB subject sets, the original MegaBlast algorithm only spends 0.05 seconds (where overall time is 962 seconds) and the optimized algorithm spends 88 seconds (where overall time is 204 seconds). So the optimized algorithm cannot achieve ideal performance.

Because MegaBlast consumes a large amount of memory, its cache miss rate is relatively high (See Figures 2 and 3). First, for the alignment results of one database sequence, the original MegaBlast algorithm sorts the whole two dimensional queue. Since the size of the queue is much greater than the size of the cache, there is a high cache miss rate in the original MegaBlast program. The optimized algorithm only needs to keep a one dimensional queue for an exchanged database sequence, which also improves spatial locality. As a result, the optimized algorithm reduces the cache capacity misses and conflict misses. Using Oprofile, we measured the cache performance for the two algorithms. Figure 16 shows that the cache miss rate of the optimized algorithm is greatly reduced.

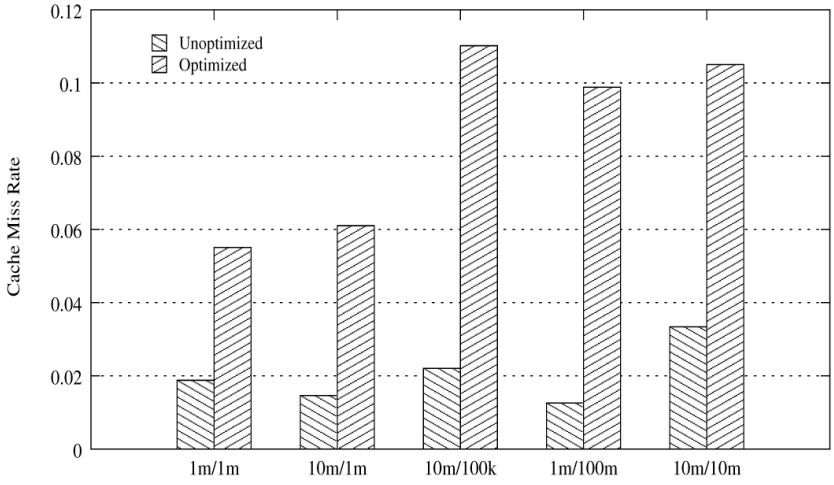

Fig. 16 Comparison of cache miss rates of two algorithms.

In addition, the original MegaBlast generates a large number of memory page faults because of its high requirement of memory. The optimized algorithm greatly reduces the requirement of memory, resulting in a much smaller number of page faults.

4 Conclusion

The motivation of our work presented in this paper is to exploit optimization methods in sequence alignment and its related applications, which are important emerging fields in high performance computing. In the traditional numerical mathematics field, many optimization techniques based on cache memory have been studied and proven effective. For example, Linpack[9] can achieve more than 90% peak performance on a single processor. However, many programs in bioinformatics such as sequence alignment and structure prediction are integer-based using abstract data and are irregular in structure. Besides, these programs are often both I/O and computation intensive.

The bioinformatics field is still in its infancy with changing problems, algorithms, applications, and even system architecture requirements. In this paper we first studied typical programs in sequence alignment to see whether traditional cache memory techniques were applicable. We found that since the most programs doing sequence alignment are both computation and I/O intensive and therefore the memory cache is not affected much. We proposed an optimization for programs using Smith–Waterman's dynamic programming algorithm

Specifically, our contributions are as follows:

1. **Memory performance analysis:** We performed a comprehensive study on the memory performance of several important bioinformatics programs. Our results show that the kernel algorithm itself is sensitive to cache memory hardware configuration, but cache memory performance is insignificant in the overall system performance because of the high cache hit rate. So, the algorithmic optimization should focus on other specific architecture features such as the optimization and parallelization of instruction levels. Another important observation is that MegaBlast is both CPU- and memory-bound.
2. **Optimizing for branch operation:** The naive implementation of Smith–Waterman's algorithm results in many branch operations. Our transformation in the algorithmic level improves the flow of instructions. The optimization technique reduces the number of both branches and mispredicted branches, halving the program execution time.
3. **Microparallelism with SSE2:** The transformation smoothes the dependency for computation along the row of the dynamic programming matrix. Utilizing the SSE2 instruction set, our optimized implementation achieves a speedup of eight. As of writing, our scheme is the fastest implementation on a general purpose single-CPU computer.
4. **Overlapping computation with I/O:** MegaBlast is a typical data- and computation-intensive program. By using a set of techniques that eliminate memory usage accumulation and hide the I/O cost by overlapping computation with I/O, we are able to significantly reduce the running time as well as the memory requirement.

A current important trend of high performance computing is to develop some efficient methods to improve the utilization of high-end computer systems. Our work is novel in the sense that it is the first effort to study architecture-based optimization methods for bioinformatics applications. Because of the infancy of the field, we only focus on one important problem: sequence alignment. As the field evolves, we expect to encompass more stable algorithms and applications. Thus, we may abstract some performance critical kernel programs and intend to develop a systematic method to optimize these kernels to be adaptive to different high performance computer architectures. On the other hand, when the performance critical kernels are standardized, it is easy to design a specific hardware accelerator to implement them as libraries.

Acknowledgments

The authors would like to thank the Advanced System Laboratory led by Prof. Mingyu Chen for providing SandFox simulator and Dr Jianwei Xu for helping us configuring the simulator, and thank Prof. Yimin Hu, Dr Yongbai Xu and all reviewers for their helpful comments and suggestions to improve this paper. This work is supported by the knowledge innovative project of the Chinese Academy of Sciences and National Natural Science Foundation of China.

Author Biographies

Guangming Tan is a Ph.D. student at the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include parallel algorithms, performance modeling and evaluation, and bioinformatics. As a visiting scholar, he studied parallel algorithm optimization on IBM Cyclops64 manycore architecture at the Computer Architecture and Parallel Systems Laboratory (CAPSL) at the University of Delaware from 2006 to 2007. He has published several papers from HICOMB'06, Euro-Par'06, SC'06 and SPAA'07, which are the top conferences in the parallel algorithm and high performance computing field.

Lin Xu is a Ph.D. student at the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include parallel algorithms, performance modeling and evaluation, and bioinformatics.

Zhenghua Dai received his Master's degree from the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include parallel algorithms, performance modeling and evaluation and bioinformatics.

Shengzhong Feng received his Ph.D. degree from the Beijing Institute of Technology. His main research interests include parallel algorithms, performance modeling and evaluation, and bioinformatics. He visited the University of Toronto from 2005 to 2007. He is the vice director of the National High Performance Computing Center, China.

Ninghui Sun received his Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences. His main research interests include computer architecture, operating systems, performance modeling and evaluation. He visited the University of Princeton from 1996 to 1997. He is the director of National Research Center for Intelligent Computing Systems. He has designed a series of Dawning supercomputers such as Dawning 1000/2000/3000/4000.

Notes

1 http://www.llnl.gov/CASC/sc2001_fliers/MemWall

2 http://www.spec2000.com

3 http://oprofile.sourceforge.net

4 http://icl.cs.utk.edu/papi

5 http://www.ncic.ac.cn/hpcog

6 http://www.dgate.org/vmips

7 http://www.phrap.org/phredphrapconsed.html

8 http://www.ncbi.nlm.nih.gov

9 http://www.netlib.org/linpack

References

Alpern, B., Carter, L., and Gatlin, K. S. (1995). Microparallelism and high performance protein matching, in *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, J. (1990). Basic local alignment search tool, *Journal of Molecular Biology*, **251**: 403–410.

Bader, D., Li, Y., Li, T., and Sachdeva, V. (n.d.). Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics, in *Proceedings of IEEE International Symposium on Workload Characterization*.

Barroso, L. and Gharachorloo, K. (1998). Memory system characterization of commercial workloads, in *25th International Symposium on Computer Architecture (ISCA'98)*, pp. 3–14.

Camp, N., Cofer, H., and Gomperts, R. (1998). High-throughput Blast, Technical Report, SGI White Paper.

Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R. C., and Yelick, K. (2005). Self adapting linear algebra algorithms and software, *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, **93**(2): 293–312.

Frigo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3, *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, **93**(2): 216–231.

Galison, F. (2001). The Fasta and Blast programs, http://bioweb.pasteur.fr/seqanal/blast/.

Gusfield, D. (2001). *Algorithms on strings, trees, and sequences: Computer science and computational biology*, Cambridge University Press.

Hennessy, J. L. and Patterson, D. A. (1999). *Computer architecture: A quantitative approach*, 2nd edn, San Francisco: Morgan Kaufmann.

Im, E. J., Yelick, K., and Vuduc, R. (2004). Sparsity: Optimization framework for sparse matrix kernels, *International Journal of High Performance Computing Applications*, **18**(1): 135–158.

Lee, D., Crowley, P., Baer, J., Anderson, T., and Bershad, B. (1998). Execution characteristics of desktop applications on Windows NT, in *25th Annual International Symposium on Computer Architecture (ISCA'98)*, pp. 27–38.

Mirkovie, D. and Johnsson, S. L. (2001). Automatic performance tuning for the UHFFT library, in *Proceedings of International Conference on Computational Science*, Lecture Notes in Computer Science, vol. 2073, pp. 71–80, Berlin: Springer.

Puschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. (2005). Spiral: Code generation for DSP transforms, in *Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, **93**(2): 232–275.

Rognes, T. and Seeberg, E. (2000). Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors, *Bioinformatics*, **16**(8): 699–706.

Smith, T. and Waterman, M. (1981). Identification of common molecular subsequences, *Journal of Molecular Biology*, **147**(1): 195–197.

Thompson, J. D., Higgins, D. G., and Gibson, T. J. (1994). Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice, *Nucleic Acids Research*, **22**: 4673–4680.

Wozniak, A. (1996). Using video-oriented instructions to speed up sequence comparison, *Bioinformatics*, **13**(2): 145–150.

Xu, Z., Sohum, S., Min, R., and Hu, Y. (2004). An analysis of cache performance of multimedia applications, *IEEE Transactions on Computers*, **53**(1): 23–38.

Zhang, Z., Schwartz, S., Wagner, L., and Miller, W. (2000). A greedy algorithm for aligning DNA sequence, *Journal of Computational Biology*, **7**(12): 203–214.