Program Generation for the All-Pairs Shortest Path Problem

Sung-Chul Han, Franz Franchetti, and Markus Püschel
Electrical and Computer Engineering, Carnegie Mellon University Pittsburgh, PA 15213
{sungh, franzf, pueschel}@ece.cmu.edu

ABSTRACT

A recent trend in computing are domain-specific program generators, designed to alleviate the effort of porting and re-optimizing libraries for fast-changing and increasingly complex computing platforms. Examples include ATLAS, SPIRAL, and the codelet generator in FFTW. Each of these generators produces highly optimized source code directly from a problem specification. In this paper, we extend this list by a program generator for the well-known Floyd-Warshall (FW) algorithm that solves the all-pairs shortest path problem, which is important in a wide range of engineering applications.

As the first contribution, we derive variants of the FW algorithm that make it possible to apply many of the optimization techniques developed for matrix-matrix multiplication. The second contribution is the actual program generator, which uses tiling, loop unrolling, and SIMD vectorization combined with a hill climbing search to produce the best code (float or integer) for a given platform.

Using the program generator, we demonstrate a speed-up over a straightforward single-precision implementation of up to a factor of 1.3 on Pentium 4 and 1.8 on Athlon 64. Use of 4-way vectorization further improves the performance by another factor of up to 5.7 on Pentium 4 and 3.0 on Athlon 64. For data type short integers, 8-way vectorization provides a speed-up of up to 4.6 on Pentium 4 and 5.0 on Athlon 64 over the best scalar code.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*Path and circuit problems*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.2 [**Programming Techniques**]: Automatic Programming

General Terms

Algorithms, Design, Performance

Keywords

Floyd-Warshall algorithm, tiling, blocking, empirical search, SIMD vectorization

PACT'06, September 16–20, 2006, Seattle, Washington, USA.

1. INTRODUCTION

The theoretical peak performance of off-the-shelf computing platforms continues to reliably increase exponentially following Moore's Law. However, a considerable fraction of this performance increase is not obtained by boosting the CPU frequency, but by microarchitectural innovations such as deep memory hierarchies, special instruction sets, and, more recently, multiple CPU cores. The resulting complexity makes it extremely difficult for software developers to write code that reaches the optimal possible performance, even for single-processor machines. Compilers can mask part of the platform's complexity, but are general-purpose tools and cannot be expected to perform optimizations relevant to every specific problem domain.

Program generators and adaptive libraries. This problem is particularly noticeable in the domain of numerical software. To solve it, a recent research trend has been the development of domain-specific program generators and self-adaptive libraries for well-understood numerical functionality. There are several prominent examples. ATLAS is a program generator for basic linear algebra subroutines (BLAS) such as matrix-matrix multiplication (MMM). For MMM, ATLAS generates implementations with different tilings (or blockings) and degrees of unrolling. It uses empirical search to find the best match to the given computer's memory hierarchy [19, 5]. A model-based, deterministic version of ATLAS was introduced in [20]. Other examples include FFTW, a combination of program generator and adaptive library for the discrete Fourier transform [8, 9]; SPIRAL, a program generator for the entire domain of linear transforms [17]; Bebop, an automatic tuning framework for sparse linear algebra problems [12, 5]; FLAME, a tuning framework for higher level linear algebra problems [11, 3]; and TCE, a program generator for tensor computations used in quantum chemistry [1].

The all-pairs shortest path problem. The focus of this paper is the all-pairs shortest path problem (APSP), which finds the length of the shortest path for all source-destination pairs in a (positively) weighted graph. It belongs to the most fundamental problems in graph theory. For example, it is well known that almost all dynamic programming problems can be equivalently viewed as problems seeking the shortest path in a directed graph [6]. Accordingly, there are many practical applications of shortest path algorithms in a broad range of engineering fields, such as geographical information systems, VLSI circuit routing, intelligent transportation systems (ITS), communication networks, and robotics. For example, computer network rout-

ing protocols such as the widely used Open Shortest Path First (OSPF) protocol make use of these algorithms to minimize the network traffic and transmission time [15].

The APSP is solved by the well-known Floyd-Warshall (FW) algorithm [2], which computes the solution inplace from the weight matrix of the graph using a triple loop similar to MMM, but involving only additions and minimum operations, and with dependencies that restrict the ordering of the three loops (the k-loop has to be the outermost one). To optimize cache performance, [18] introduced a tiled version of the FW algorithm. Further improvements were made in [16], which showed that the tiling can be done recursively up to some chosen base case and combined with a Z-Morton data layout to increase performance. The degree of tiling and the base case size were found by search using an adaptive library framework. However, [18, 16] focused on the exploitation of cache locality, and did not consider broader ranges of techniques that are commonly used for MMM. Further, vectorization for the latest generation of SIMD architectures was not considered but is crucial to achieve the best possible performance.

There are also a number of problems that are closely related to APSP, but with fundamentally different approaches to their solutions. Updating an APSP solution when the edge weights change dynamically is the subject of [4]. An efficient algorithm for the point-to-point shortest path problem using Dijkstra-based search and landmarks to provide lower bounds for the distance is presented in [10]. Finally, there is also work on computing approximations of the shortest paths to reduce computation time [7].

Contribution of this paper. The goal of this work was to develop a thorough automatic tuning framework that produces very fast implementations of the FW algorithm. In doing so, we could improve the previous performance considerably. To achieve this, we built a program generator, in design similar to ATLAS, that searches over different degrees of tiling and unrolling to produce the fastest code for a given platform. Unlike ATLAS, our generator considers two levels of tiling and, more importantly, can generate scalar and SIMD vector code (we focused on Intel's 4-way float and 8-way short integer instructions).

Further, to apply the above techniques to the extent possible, we introduced one crucial algorithmic innovation: for suitably chosen subproblems we were able to remove the loop order restrictions and thus match the MMM implementation more closely and further improve performance.

Comparing our generated scalar code to [16] on a Pentium 4 for data type float, we could improve performance from 8% to 32% and gained another speed-up of between 3.5 times and 6.2 times using 4-way vectorization. We also show experiments with data type 16-bit short integer, where 8-way vectorization yields a speed-up factor of 2.6 to 5.8 over scalar integer code. The experiments also show that, for vector code in particular, tiling and unrolling is crucial to achieve the best performance. The results obtained on an Athlon 64 show similar trends, but the 8-way vectorization yields higher improvement than on a Pentium 4.

Organization of this paper. In Section 2 we explain the standard FW algorithm and derive parameterized variants through tiling and unrolling. These constitute the implementation space we search for the fastest on a given platform. Section 3 explains how to vectorize all FW algorithms derived previously for SIMD short vector architectures. Additional details are provided for Intel's SSE architecture. The code generator and the search strategy for the fastest implementation are described in Section 4. Section 5 shows experimental results obtained with code generated and optimized for a Pentium 4 and an Athlon 64 using both float and short integer data type and corresponding 4-way and 8-way vector code, respectively. Finally, we offer conclusions in Section 6.

2. FLOYD-WARSHALL ALGORITHMS

In this section we formally introduce the all-pairs shortest path problem for a weighted graph and the original Floyd-Warshall (FW) algorithm for its solution. Then we derive different parameterized variants of the algorithm through unrolling and tiling (or blocking). The FW algorithm has a structure similar to a standard dense matrix-matrix multiplication (MMM); thus, the derived blocked variants are similar to blocked MMM algorithms (for example those used in ATLAS). However, there are also important differences due to dependencies in the FW algorithm. As a consequence, only some of the MMM optimizations and code transformations are applicable. These differences also impact the final performance one can expect and we discuss them as we go along.

As in ATLAS, the rationale for suitable parameterization of the algorithms is to connect them with a search (explained in Section 4) that finds the best match for the computing platform's memory hierarchy.

We focus on standard scalar implementations in this section. The corresponding vectorized algorithms, i.e., optimized for short-vector SIMD instruction sets, are shown in Section 3.

In the following, we denote matrices with A, B, C, matrix elements with $A[i][j]$, and submatrices Matlab-style with $A[i_1 : i_2][j_1 : j_2]$.

All-pairs shortest paths problem (APSP). Let $G = (V, E, w)$ be a given weighted graph with vertices $v \in V = \{1, \ldots, N\}$, edges $(i, j) \in E \subset V \times V$, and the positive weight function $w : E \to \mathbb{R}^+$; $w(i, j)$ is the "cost" of the edge (i, j). We want to compute the minimum distance

$$\mathrm{dist}_G(i, j) = \min_{P \in \mathrm{paths}(i,j)} \sum_{(u,v) \in P} w(u, v)$$

of all pairs of vertices i and j in the graph G.

For the actual computation, we assume that the graph G is given by the $N \times N$ *cost matrix*

$$C \quad \text{with} \quad C[i][j] = \begin{cases} 0 & \text{if } i = j \\ w(i, j) & \text{if } (i, j) \in E \\ \infty & \text{else} \end{cases}.$$

Then we want to compute the $N \times N$ matrix

$$C' \quad \text{with} \quad C'[i][j] = \mathrm{dist}_G(i, j).$$

The FW algorithms considered in this paper compute the solution inplace, i.e., C' overwrites C.

FW algorithm. The FW algorithm [2] solves the APSP using three nested loops as shown in Fig. 1.

The outermost k-loop updates the cost matrix C in each of its N iterations. At the beginning of the kth iteration the cost matrix C contains the cost of the shortest path for all pairs (i, j) over all paths that only contain intermediate nodes $1, \ldots, k - 1$. Now, for each pair (i, j), these costs

```
// standard FW algorithm
function FW(C, N)
  for k=1:N
    for i=1:N
      for j=1:N
        C[i][j] = min(C[i][j], C[i][k]+C[k][j]);
```

Figure 1: Standard FW algorithm.

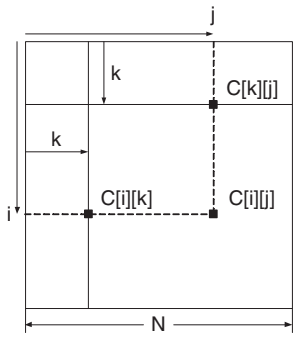


Figure 2: Access Pattern of the FW algorithm.

are compared (in a double loop) to the costs of paths via k and the better ones are stored to $C[i][j]$. Fig. 2 visualizes the corresponding access pattern of the cost matrix C in relation to the three loop variables k, i, and j.

The algorithm requires exactly N^3 additions and N^3 minimum operations. The structure of the code is very similar to a standard MMM. However, there is one important difference. In MMM the loops can be permuted into any order. In the FW algorithm, the k-loop has to be outermost due to dependencies in the computation; the order of the i- and j-loop can be exchanged [18]. For MMM, it is known that choosing the i- or j-loop as the outermost one is far superior; this is one of the reasons why the performance of MMM is not achievable with the FW algorithm. However, as we show below, it is possible to remove the dependency for subproblems, thus enabling an improved order of access.

Tiling the FW algorithm. Similar to MMM, the FW algorithms can be improved for execution on a memory hierarchy by tiling as was shown in [16, 18]. We are closely following the approach taken by ATLAS and considering three different classes of algorithms. Each is parameterized by degrees of unrolling or by tile sizes: an iterative FW variant (FWI) partially tiled and unrolled, a singly tiled FW variant (FWT), and a doubly tiled FW variant (FWD) to improve cache locality. Fig. 3 shows the different algorithms, their recursive structure, and their parameters: U_* and U'_* are unrolling/tiling parameters, L_* are cache tiling parameters. FWT is used as subroutine in FWD and FWI is used as subroutine in FWT. Note that FWI and FWT have "abc variants," which are only used as subroutines. In these versions, the dependencies are removed (and with them the need to have the k-loop as outermost) and MMM-style full tiling becomes possible. The details on the three algorithms including correctness proofs are provided next.

2.1 Iterative FW algorithm: FWI

Before we introduce the iterative FW algorithm (FWI), we need to generalize the FW algorithm in Fig. 1. Namely, we replace the one matrix C in FW by three matrices A, B, and C, which may be different or not. The result is FWgen as shown in Fig. 4. A contains the distances from i to k, B the distances from k to j and C the shortest distances from i to

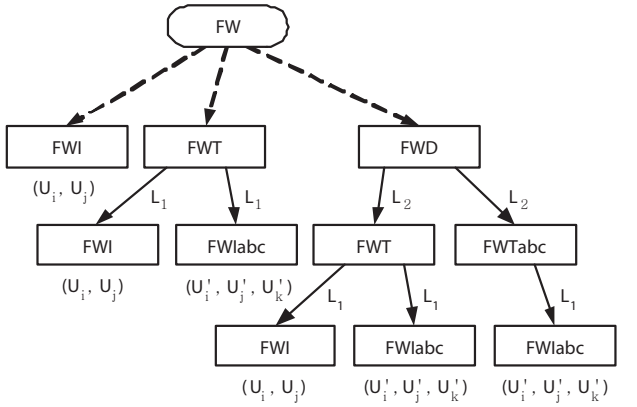


Figure 3: Parameterized FW algorithms considered in this paper. The algorithms arise from different levels of tiling and unrolling.

```
// generalized FW algorithm
function FWgen(A, B, C, N)
  for k=1:N
    for i=1:N
      for j=1:N
        C[i][j] = min(C[i][j], A[i][k]+B[k][j]);
```

Figure 4: Generalized FW algorithm.

j, which will be updated. Note that if $A = B = C$, FWgen reduces to the standard FW algorithm. In the following we will always work with the generalized version.

Tiling and unrolling: FWI. As the first optimization we tile and unroll FWgen similar to MMM. However, because of the mentioned dependencies, a full tiling is not possible, i.e., only the i- and j-loop can be tiled. We perform this tiling and unroll the two innermost loops to obtain the FWI shown in Fig. 5. FWI is parameterized by the tiling/unrolling factors U_i and U_j. Note that the FWI in [16] is equivalent to our FWgen.

Special case: FWIabc. To solve the APSP for an $N \times N$ cost matrix C, FWI is called as FWI(C, C, C, N). However, we will use FWI below as a subroutine with possibly distinct inputs A, B, C. If the matrices A, B, C are known to be mutually distinct, then the dependencies are removed and we can 1) reorder the loops to make the k-loop innermost; and 2) introduce full tiling. We call the resulting routine FWIabc. It is shown in Fig. 6 and is parameterized by the tiling/unrolling factors U'_i, U'_j, U'_k.

Note that the outermost loop in the inner loop nest of FWIabc is the k'-loop for register blocking and instruction level parallelism as in ATLAS.

```
// iterative FW algorithm (FWI)
// tiling factors Ui and Uj
function FWI(A, B, C, N)
  for k=1:1:N
  for i=1:Ui:N
  for j=1:Uj:N
    // loops below are completely unrolled
    for i'=i:1:i+Ui-1
    for j'=j:1:j+Uj-1
      C[i'][j'] = min(C[i'][j'], A[i'][k]+B[k][j']);
```

Figure 5: FWI parameterized by U_i, U_j.

```
// FWI for 3 distinct matrices (FWIabc)
// tiling factors Uk, Ui, and Uj
function FWIabc(A, B, C, N)
  for i=1:Ui':N
  for j=1:Uj':N
  for k=1:Uk':N
    // loops below are completely unrolled
    for k'=k:1:k+Uk'-1
    for i'=i:1:i+Ui'-1
    for j'=j:1:j+Uj'-1
      C[i'][j'] = min(C[i'][j'], A[i'][k']+B[k'][j']);
```

Figure 6: FWIabc parameterized by U_i', U_j', U_k'.

```
// tiled FW algorithm (FWT)
// tile size: L1 x L1
function FWT(A, B, C, N, L1)
  // A_ij: L1 x L1 submatrix (i,j) of A, i.e.,
  //      A[(i-1)*L1+1:i*L1][(j-1)*L1+1:j*L1];
  M = N/L1;
  for k=1:1:M
    // phase 1
    FWI(A_kk, B_kk, C_kk, L1);
    // phase 2
    for j=1:1:M, j!=k
      FWI(A_kk,B_kj,C_kj, L1);
    // phase 3
    for i=1:1:M, i!=k
      FWI(A_ik, B_kk, C_ik, L1);
    // phase 4
    for i=1:1:M, i!=k
    for j=1:1:M, j!=k
      FWIabc(A_ik,B_kj,C_ij, L1);
```

Figure 7: FWT parameterized by the tile size L_1 and the parameters of FWI and FWIabc.

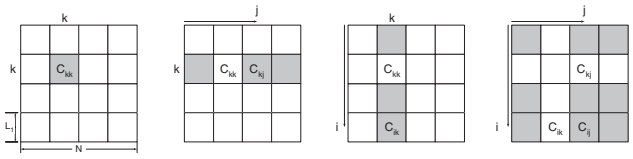
2.2 Tiled FW algorithm: FWT

To enhance cache performance, cache tiling of the FW algorithm was introduced in [16, 18]. Following this idea we introduce FWT, a singly cache tiled version of the FW algorithm. The tiles, in turn, are handled by FWI and FWIabc introduced before. FWT takes the tile size L_1 in addition to the parameters of FWI and FWIabc.

Phases in FWT. The FWT computes the result looping over the tiles in a specific order obeying the dependencies of the algorithm. As shown in Fig. 7, the outermost loop in FWT iterates from 1 to $M = N/L_1$, which is the number of tiles. If FWT is chosen as FW implementation, it is called as FWT(C, C, C, N), i.e., with the same input tiles.

Within FWT, the argument tile of size N is divided into tiles of size L_1. For a fixed k, all of the tiles are updated using four distinct phases, visualized in Fig. 8. At the kth iteration, in phase 1, FWI is called to update the diagonal tile $C_{k,k}$ itself. Phase 2 updates all tiles $C_{k,j}$ in the same row as the diagonal tile and phase 3 updates the respective column of tiles $C_{i,k}$ using FWI. Note that in the phases 1, 2, and 3, the subroutine FWI is invoked as FWI(C, C, C, L_1), FWI(A, C, C, L_1), and FWI(C, B, C, L_1), respectively. Since two or more of the argument tiles are the same, the loop order restriction should be observed, which implies that FWIabc cannot be used in place of FWI.

The major part of the computation is done in phase 4 by updating all of the remaining tiles $C_{i,j}$. It is a crucial insight that in phase 4 the 3 tiles considered in each step are always known to be distinct; thus, FWIabc with its improved tiling structure (Fig. 6) can be used to improve performance. Within each FWT, FWIabc is called $M(M-1)^2 = O(M^3)$ times while FWI is only called $M(2M-1) = O(M^2)$ times.



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4

Figure 8: Visualization of the 4 phases in FWT.

This implies that the FW performance can approach that of MMM for large M.

Special case: FWTabc. Analogous to FWIabc, we consider and also define FWTabc as a version of the FWT that can be used when the input matrices are mutually distinct. Compared to FWT, all calls to FWI are replaced by calls to FWIabc. FWTabc is used as subroutine in FWD that is introduced next.

2.3 Doubly tiled FW algorithm: FWD

We implement another level of tiling in the doubly tiled FW implementation (FWD). FWD requires two blocking parameters L_1 and L_2 and calls FWT and FWTabc as child routines. Although any number of tilings is possible, we limited it to two levels since the incremental improvement with more blocking levels turned out to be insignificant. This second level of tiling enables locality for two levels of caches. Note that ATLAS only tiles for one level of cache.

2.4 Correctness of FWI, FWT, and FWD

We will use the following notation in this section. Given two matrices A and A' of the same size, $A = A'$ signifies that A and A' reside in the same memory location. On the other hand, $A \neq A'$ means that A and A' reside in different memory locations with no overlapping region. Further, given two functions F and F', $F() = F'()$ means that $F()$ and $F'()$ yield the same result for all possible choices of input.

THEOREM 1. *FWI(A, B, C, N) = FWgen(A, B, C, N).*

Proof. We will show a slightly stronger statement, namely that in FWgen (see Fig. 4) the state of C at the end of each k-iteration does not depend on the order in which the pairs (i, j) were processed. In particular, this enables the tiling of the i- and j-loop in FWI (see Fig. 5). We have to consider four cases.

Case 1 ($A = B = C$): Assume the outer loop variable k is fixed. The update in FWgen can be written in this case as

$$C[i][j] \leftarrow \min(C[i][j], C[i][k] + C[k][j]). \qquad (1)$$

First, we assert that for $i = k$ and $j = k$, $C[i][j]$ does not change. (These are all elements in the same row or column as $C[k][k]$, i.e., those on the cross in Fig. 2.) Namely, in these cases,

$$C[i][k] \leftarrow \min(C[i][k], C[i][k] + C[k][k]) = C[i][k],$$
$$C[k][j] \leftarrow \min(C[k][k], C[k][k] + C[k][j]) = C[k][j].$$

Now, consider the remaining elements $C[i][j], i \neq k, j \neq k$. From (1) and the above, we can see that the new value of $C[i][j]$ is solely determined by the value of $C[i][j], C[i][k]$, and $C[k][j]$ at the $(k-1)$th iteration. This implies that there is no dependency on the processing order of i and j as desired.

Case 2 ($A = C \neq B$): The update operation in FWgen can be written as

$$C[i][j] \leftarrow \min(C[i][j], C[i][k] + B[k][j]). \quad (2)$$

Note that B is never updated. Further, for $j = k$,

$$C[i][k] \leftarrow \min(C[i][k], C[i][k] + B[k][k]) = C[i][k],$$

i.e., $C[i][k]$ is unchanged. Since $C[i][j]$ depends only on the the values of $C[i][j]$ and $C[i][k]$ at the previous iteration and the constant $B[k][j]$, there is again no dependency on the processing order of i and j.

Case 3 ($B = C \neq A$): The proof is the same as case 2 except that A is constant and that $C[k][j]$ does not change at the kth iteration.

Case 4 ($A \neq C$ and $B \neq C$): With constant A and B, $C[i][j]$ at each iteration is not affected by the processing order of i and j.

THEOREM 2. *FWIabc(A, B, C, N) = FWgen(A, B, C, N) if $A \neq C$ and $B \neq C$.*

Proof. FWgen(A, B, C, N) does not modify A and B. C is modified by FWgen as follows

$$C[i][j] \leftarrow \min\Big(C[i][j], \min_{k\in\{1,\ldots,N\}}(A[i][k] + B[k][j])\Big).$$

Since the minimum operator is associative and commutative with respect to k, the result does not depend on the processing order of i, j and k. In particular, the order in FWIabc can be chosen.

THEOREM 3. *FWT(C, C, C, N) = FW(C, N).*

Proof. In FWT, there are two subroutines: FWI and FWIabc. FWI can be replaced by FWgen by Theorem 1. FWIabc can also be replaced by FWgen by Theorem 2. The replacements yield the original blocked FW algorithm introduced in [18] as "BlockedAllPairs," which was proved to be equal to FW. This yields the result.

THEOREM 4. *FWD(C, C, C, N) = FW(C, N).*

We only sketch the proof as it does not introduce any new ideas, but would be lengthy to carry out in detail. FWD calls two subroutines: FWT and FWTabc. Having the same tiling structure as FWT, FWD(C, C, C, N) is equivalent to FWT(C, C, C, N) if the following two conditions hold

1. FWT(A, B, C, N) = FWgen(A, B, C, N), and
2. FWTabc(A, B, C, N) = FWgen(A, B, C, N) for $A \neq B \neq C \neq A$.

Condition 1 holds for $A = B = C$ by Theorem 3. The cases $B = C \neq A$ and $A = C \neq B$ can be shown by extending the proof in [16] for the recursive FW algorithm (FWR). (They showed that FWR(A, B, C) and FWI(A, B, C) give the same result for a tiling factor of 2, where their FWI is the same as our FWgen.) Condition 2 follows from the proof of Theorem 2, which shows that the result is independent of the processing order of i, j, and k.

3. SIMD VECTORIZATION

Recent generations of general-purpose microprocessors introduced vector SIMD (single instruction, multiple data) instructions that operate on short vectors (length 2 to 16) of floating-point or integer data types. For example, Intel's newest Pentium 4 features the third generation of streaming SIMD extensions called SSE3.

In this section we explain how we vectorize the FW algorithms introduced in Section 2 and thus extend our code generator to produce SIMD vector code. Then we provide further details for the actual implementation on Intel's SSE architecture.

Vectorization of FW algorithms. All computation of the FW algorithms introduced in Section 2 is done in the two iterative leaf routines FWI and FWIabc (see Figs. 5 and 6). The regular structure of both routines allows us to vectorize the innermost loop in these routines and thus the entire computation in all FW algorithms. All our matrices are stored linearly in memory using the row-major storage scheme (also called C storage scheme). This means that $C[i][j]$ and $C[i][j+1]$ are adjacent in memory.

We vectorize FWI and FWIabc for arbitrary vector length, denoted by ν. To express the vectorized implementation, we introduce three generic ν-way vector instructions. Later in this section we explain how to implement these operations using 4-way and 8-way SSE instructions.

- `v_add(a,b)`: elementwise addition of vectors `a` and `b`.
- `v_min(a,b)`: elementwise minimum of vectors `a` and `b`.
- `v_dup(a)`: creates a length-ν vector that contains the value of the scalar `a` in all vector elements.

SIMD vector code for FWI and FWIabc is obtained by standard loop vectorization as briefly explained next.

The first code in Fig. 9 shows the innermost loop of FWI and FWIabc. We require that $\nu | U_j$. We see that the j'-loop accesses contiguous data $B[k][j']$ and $C[i][j']$ for all values of i and k and that $A[i][k]$ is independent of j'. Instead of unrolling the j' loop as in FWI and FWIabc, we first vectorize the loop and then unroll it.

To obtain the second code in Fig. 9, we tile the j'-loop into a j_0-loop with U_j/ν iterations and a j_1-loop with ν iterations.

The actual vectorization is done by executing the ν iterations of the j_1-loop in parallel using ν-way vector instructions; the result is the third code in Fig. 9. Fig. 10 shows the tile elements that are involved in a `v_add()` operation. The remaining outer j_0-loop will be unrolled.

SIMD vector architectures are most (sometimes only) efficient if contiguous, aligned vectors are loaded and stored. From the row-major format follows that the vectors $B[k][j_0 : j_0 + \nu - 1]$ and $C[i][j_0 : j_0 + \nu - 1]$ are contiguous in memory. To guarantee alignment, we place our matrices such that $A[0][0]$, $B[0][0]$, and $C[0][0]$ are stored at a correctly aligned address, allowing us to use efficient vector memory access. Loading the vector of identical elements $[A[i,k], \ldots, A[i,k]]$ requires a scalar load and a duplication of the constant into all vector elements using `v_dup()`.

Using the vector minimum operation requires a vector store for *each* result vector $C[i][j_0 : j_0 + \nu - 1]$, even though typically only few changes of $C[i][j]$ throughout the algorithm occur. This increases memory traffic considerably but

```
// original unrolled loop
for j'=j:1:j+Uj-1
   C[i][j'] = min(C[i][j'], A[i][k]+B[k][j']);

// tiled loop
// j0 loop will be unrolled
for j0=j:v:j+Uj-1
   // j1 loop will be done in parallel
   // using v-way vector instructions
   for j1=0:1:v-1
      C[i][j0+j1] =
         min(C[i][j0+j1], A[i][k] + B[k][j0+j1]);

// vectorized loop
// this loop will be unrolled
for j0=j:v:j+Uj-1
   // this implements the j1-loop
   // using v-way vector instructions
   C[i][j0:j0+v-1] =
      v_min(
         C[i][j0:j0+v-1],
         v_add(
            v_dup(A[i][k]), B[k][j0:j0+v-1]));
```

Figure 9: Vectorization of the innermost loops in FWI and FWIabc. From top to bottom: innermost loop in FWI; the same loop tiled with tile size ν; the tiled loop vectorized.

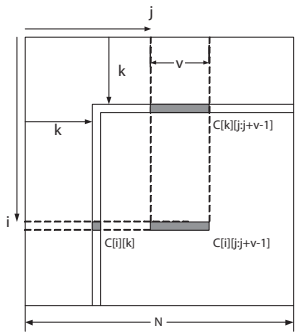


Figure 10: Access pattern of the vectorized version of FWI/FWIabc.

is compensated by the efficient vector minimum in the SSE implementation.

The code for the fully vectorized FWI implementations (FWIV and FWIVabc) can be obtained as shown in Fig. 9.

Intel SSE. We refer to the SSE instruction set family (SSE/2/3) as SSE and generate 4-way 32-bit single-precision floating-point and 8-way 16-bit integer FWIV/FWIVabc implementations. In all implementations we use Intel's proprietary C extension implementing intrinsic functions and vector data types to avoid assembly coding [13].

SSE requires data to be 16-byte aligned and accessed in 16-byte units (one ν-way vector) to use the very efficient vector load and store instruction `movaps` or `movdqa` for 4-way float or 8-way integer, respectively. We store our matrices at addresses that are multiples of 16 bytes to enable vector memory access.

The elementwise vector addition `v_add()` is implemented by `addps` in the 4-way float case and by `paddw` in the 8-way integer case.

The elementwise vector minimum `v_min()` is implemented by `minps` in the 4-way case and by `pminsw` in the 8-way case to perform 4 or 8 minimum operations in parallel by one instruction without the necessity of additional compare or jump instructions.

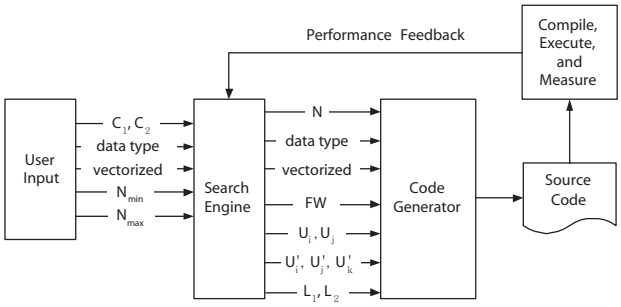


Figure 11: Architecture of the APSP program generator and optimizer.

Parameter	Description
Problem size and data type	
N	input size
data type	float or short integer
vectorized	yes or no
FW algorithm	
FW	FW algorithm (FWI, FWT or FWD)
(U_i, U_j)	Unrolling factors for FWI kernel
(U_i', U_j', U_k')	Unrolling factors for FWIabc kernel
L_1	level-1 tile size for FWT and FWD
L_2	level-2 tile size for FWD

Table 1: Input parameters to the code generator.

`v_dup()` is implemented by first loading the scalar into the lowest vector element using `movss` in 4-way mode. In 8-way mode the 16-bit integer is loaded into a standard integer register using `mov` and then copied to the vector register using `movd`. Then a vector broadcast is used to copy vector element 1 to all other vector elements. It is `shufps` for 4-way float. For 8-way short integer mode we use the instructions `punpcklwd` to copy the scalar from element 1 to elements 1 and 2 and then `punpckldq` to copy this pair to the remaining three pairs of the 8-way vector.

4. AUTOMATIC TUNING

In Sections 2 and 3 we introduced parameterized scalar and vector variants of the FW algorithms (FWI, FWT and FWD) obtained through tiling and unrolling. To obtain an optimized implementation we now follow the approach of ATLAS: using automatic program generation coupled with search to find the best algorithm and parameter choice for the given platform.

In the following we overview the program generator and explain the search strategy to find the best implementation.

4.1 Overview

Fig. 11 shows an overview of our APSP program generation and optimization approach. The diagram is similar to the diagram of ATLAS in [20].

At the heart of the optimization is a code generator, which takes as input the problem size N, the data type, and a set of parameters that uniquely specify an FW implementation from Section 2 or 3 (see Table 1). The output is the corresponding implementation.

The program generator is wrapped into a feedback loop that is controlled by a search engine. The search engine gen-

Parameter	Description
$C_1,\ C_2$	level-1 and level-2 cache size
data type	float or short int
vectorized	yes or no
$N_{\min}$	minimum input size
$N_{\max}$	maximum input size

Table 2: User-specified parameters.

erates different input configurations for the program generator to find the best implementation according to a search strategy (explained below).

The search engine takes several user-specified parameters as input (see Table 2). These parameters dictate the desired range of problem sizes N (for simplicity we restrict ourselves to two-power sizes), the desired data type (which is passed along to the program generator), and the sizes of level-1 and level-2 cache. The latter are used to compute initial block sizes for FWT and FWD in the search.

4.2 Search Strategy

In this section we describe the search strategy to find optimal parameters for our parameterized FW variants. Due to the large parameter space an exhaustive search is not practical, and we use a combination of exhaustive and hill climbing schemes. In essence, for each FW algorithm we first find a "reasonable" choice of parameters, then we further refine the parameters using hill climbing. This strategy is different from the orthogonal line search used in ATLAS.

The outline of the search for each problem size N with $N_{\min} \le N \le N_{\max}$ is as follows and will be explained in detail below.

1. Initial guess of unrolling parameters:
   (a) Find best (U_i, U_j) for FWI with $N = 64$.
   (b) Find best (U'_i, U'_j, U'_k) for FWIabc with $N = 64$.
2. Optimize FWI:
   (a) Set (U_i, U_j) as found in step 1(a).
   (b) Refine (U_i, U_j) by hill climbing.
3. Optimize FWT:
   (a) Set $(U_i, U_j, U'_i, U'_j, U'_k)$ as found in step 1(b).
   (b) Set L_1 to an analytical estimate.
   (c) Refine $(L_1, U_i, U_j, U'_i, U'_j, U'_k)$ by hill climbing.
4. Optimize FWD:
   (a) Set $(U_i, U_j, U'_i, U'_j, U'_k)$ as found in step 1(b).
   (b) Set (L_1, L_2) to analytical estimates.
   (c) Refine $(L_1, L_2, U_i, U_j, U'_i, U'_j, U'_k)$ by hill climbing.

The same search strategy is used for scalar and vector code. The search was designed with the underlying assumption that the best unrolling parameters for different FW algorithms and different input sizes would be similar but not necessarily the same. We provide further details.

Step 1: Initial guess for unrolling parameters. To find the best unrolling parameters for FWI, we choose a problem size of $N = 64$. Then, we consider all unrolling parameters (U_i, U_j) with $1 \le U_i \le 16$ and $1 \le U_j \le 32$ (two-powers only) for scalar code and restrict to $\nu \le U_j \le 32$ for vector code.

For FWIabc, we choose again a problem size of $N = 64$, and set $U'_k = 1$. Then, we search exhaustively by considering all unrolling parameters (U'_i, U'_j) with $1 \le U'_i \le 16$ and $1 \le U'_j \le 64$ (two-powers only) for scalar code and restrict to $\nu \le U'_j \le 64$ for vector code. Finally, with the best (U'_i, U'_j) found, we consider all possible choices of U'_k with $1 \le U'_k \le 32$ (two-powers only).

Although this search is much faster than a fully exhaustive search, a pairwise exhaustive search for (U_i, U_j) or (U'_i, U'_j) still generates a large number of cases to measure. Nevertheless, this step does not consume much time since the problem size is very small.

From this point on, searches are done by hill climbing, where the performances with a unit-step change in all directions are measured, and the best direction is chosen as the next state. The search is terminated when all of the neighbors have lower performance than the current state, i.e., upon reaching a local maximum. Thus a good choice of an initial value is crucial for the quality of the search result.

By starting from the best unrolling factors found at step 1, the hill climbing search at the steps 2, 3, and 4 completes in a reasonable time even for large input matrices.

Step 2: Optimization for FWI. The unrolling parameters (U_i, U_j) found in step 1 are further optimized by hill climbing.

Step 3: Optimization for FWT. There are six parameters $(L_1, U_i, U_j, U'_i, U'_j, U'_k)$ to be optimized in this step. The unrolling parameters found at step 1(b) are used as the initial state.

The initial state for the tile size L_1 is determined using a simple analytical model in the spirit of [20] and explained next. Inspecting FWT in Fig. 7 (see also the visualization Fig. 8) shows that at most three tiles are concurrently used as working set. By requiring that they fit into level-2 cache, the tile size should satisfy

$$3L_1^2 \le C_2, \tag{3}$$

where C_2 is the level-2 cache size (measured in the size of the chosen data type). Thus, we set

$$L_1 = \left\lfloor \sqrt{C_2/3} \right\rfloor. \tag{4}$$

Note that we choose level-2 cache, since on modern platforms it is almost as fast as the level-1 cache. Thus, if only one level of blocking is chosen, it should be for the level-2 cache.

Step 4: Optimization for FWD. Since there are two blocking parameters L_1 and L_2, we use the above model for both levels of cache:

$$L_1 = \left\lfloor \sqrt{C_1/3} \right\rfloor \quad \text{and} \quad L_2 = \left\lfloor \sqrt{C_2/3} \right\rfloor, \tag{5}$$

where C_1 is the level-1 data cache size (measured in the size of the chosen data type).

After that we use a hill climbing search to further optimize all parameters.

5. EXPERIMENTAL RESULTS

In this section we present performance results obtained with our generated code. We first explain the experimental

setup. Then we show and discuss performance plots for the generated scalar and vector code of data types float and short integer.

Platform. The experiments were conducted on two platforms: 1) Pentium 4, 3.6 GHz (model number 560), with 16 KB L1 cache, 1 MB L2 cache, and 1 GB main memory, and 2) Athlon 64, 2.4 GHz (model 4000+), with 64 KB L1 cache, 1 MB L2 cache, and 1 GB main memory. The operating system is SuSE Linux 9.3. We used the GNU C compiler 3.3.5 (gcc) with flags "-O3 -march=pentium4" for scalar code, and the Intel C++ compiler 9.0 (icc) with the flag "-O2" for vector code. We performed a small search to make sure we use the best compiler and compiler flags.

We used both single precision float and short integer (16-bit). In both cases, we considered corresponding short vector data types provided by Intel's SSE: 4-way for float and 8-way for short integer. Note that double precision should be unnecessary for practically all application since the FW algorithms are numerically very stable (since they involve only additions). For short integers, overflow may occur depending on the weights and the graph size and structure.

Implementation of the minimum operation. Half of all operations in the FW algorithm are minimum operations, which can be expensive. The vectorized code uses `v_min` implemented as explained in Section 3. For scalar minimum operations, we tried to make the compiler use predicated move instructions `cmov`, but only succeeded by resorting to inline assembly. Due to the side effects of inline assembly on register allocation and instruction scheduling, the straightforward implementation was ultimately the fastest:

```
#define MIN(a, b) ((a) < (b) ?  (a) :  (b))
```

Cost matrix generation. As input to all FW algorithms, we generated bidirectional directed graphs with random weights from 1 to 10 using the graph generation package provided by [14]. The number of nodes N in the graphs was constrained to be a power of two in the range $64 \leq N \leq 4096$, and the number of edges in the graph was set to approximately $N^2/3$. We also experimented with two extreme edge densities, a fully connected graph and a ring, but the influence on the performance was not noticeable. This behavior was also mentioned in [18].

We compute the performance numbers as $(2N^3)$/runtime, i.e., we count both additions and minimum operations as 1.

5.1 Benchmarks on Pentium 4

Scalar code. In the first experiment, we compared our best generated scalar code for data type float against a standard triple-loop implementation (Fig. 1) and against the fastest implementation we found in the literature [16]. The authors kindly provided us with their code. Their algorithm first permutes the input matrix into the Z-Morton data layout and then uses a recursive multilevel tiling algorithm until a base case size is reached, at which point their FWI (our FWgen) is used. Search is used to find the best base case size with a tiling factor of two (the same in every level). No unrolling is performed and the loop exchange based on FWIabc is not performed. We call their implementation ZM-FWR.

Fig. 12(a) shows the performance achieved in each case. Our generated code is separated into the three discussed FW variants (FWI, FWT, FWD). The best of those run between 21% and 29% faster than the standard implementation and between 8% and 32% faster than ZM-FWR. Further, our best code reaches 28% of the scalar peak performance. Matrix-matrix multiplication using ATLAS' search reaches about 70% [20]. The loss is due to the k-loop being the outermost loop and due to the minimum operations, which produce branches.

Fig. 12(a) also shows that FWT and FWD exhibit similar performance while both are superior to FWI. The sudden drop for FWT at $N = 4096$ is because the hill climbing search fell into a local maximum. By manually tuning the parameters one can remove that behavior.

Vector code (float). Fig. 12(b) compares the performance of our best generated vector implementations for data type float with our best scalar implementation. The vector code is up to about 5.7 times faster, exceeding the vector length 4 and reaching up to 5.3 GFLOPS. We see superlinear speed-up because we are not close to the machine's peak performance and so other factors can contribute significantly. For example, vectorization reduces the code size and enables larger blocks to be unrolled. The performance of the vector code using FWT and FWD is roughly equal. FWI performs poorly for $N > 256$ (the L2-cache boundary), which shows that tiling is mandatory for good vector code performance. Using compiler vectorization (icc) in tandem with our program generator yields only marginal improvements of up to 20%.

Vector code (short integer). The performance with data type short integer is shown in Fig. 12(c), which exhibits essentially the same behavior as Fig. 12(b) for float. The vector code reaches 9.2 GIPS, which is about 4.1 times faster than the best scalar code. As in the case of data type float, FWT and FWD show similar performance while FWI performs poorly beyond the L2-cache boundary. The compiler vectorization (using icc) showed no improvement over our best scalar code.

Parameters found. For FWD and float, the parameters found are shown in Table 4.

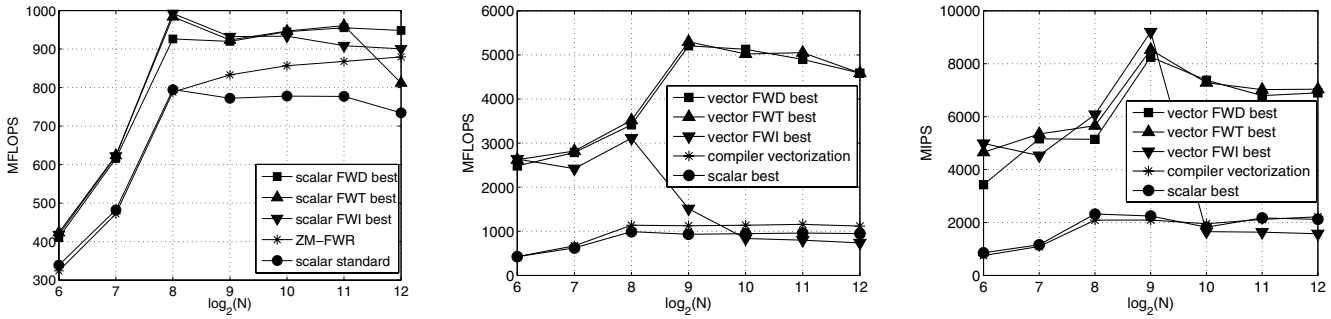
5.2 Benchmarks on Athlon 64

Scalar code. Fig. 13(a) compares the performance of the different scalar FW algorithms. The best scalar code runs between 44% and 78% faster than the standard implementation and between 13% and 50% faster than ZM-FWR. It seems that the Athlon benefits more from unrolling, since it has a 64 KB traditional L1 instruction cache whereas the Pentium only has a 12 KB micro ops trace cache. Table 4 shows that larger unrolling factors where found on the Athlon.

Fig. 13(a) also shows that for larger problem sizes the performance of FWT and FWD decreases below FWI. This is different to the Pentium and may be due to the Athlon's different cache structure.

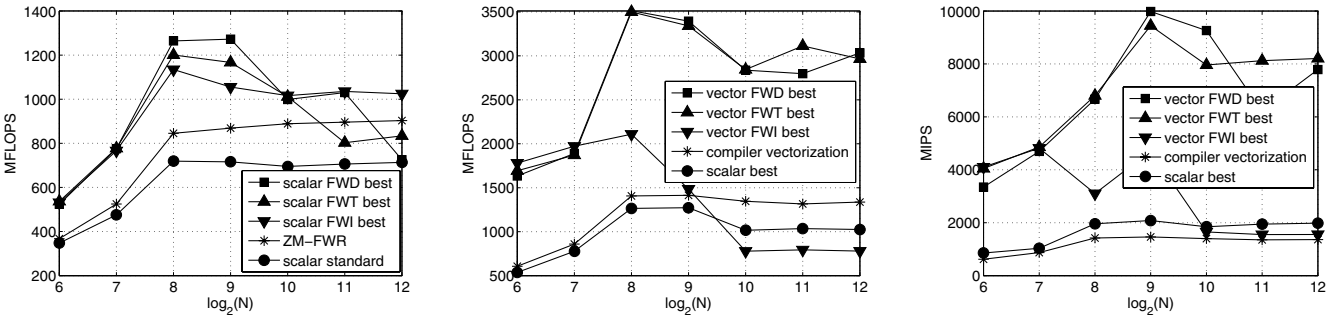
Vector code (float). Fig. 13(b) compares the different vectorized FW algorithms. The best generated vector code is up to 3 times faster than the best scalar code with data type float, reaching up to 3.5 GFLOPS. Similarly to the performance characteristics on the Pentium, FWI's performance starts to deteriorate for $N > 256$ while the tiled versions (FWT and FWD) further gain performance. The improvement using compiler vectorization (icc) was only up to 30%.

Vector code (short integer). The performance with data type short integer is shown in Fig. 13(c). We see a



(a) Scalar code (float) (b) Vector code (float) (c) Vector code (short integer)

Figure 12: Performance comparison on Pentium 4.



(a) Scalar code (float) (b) Vector code (float) (c) Vector code (short integer)

Figure 13: Performance comparison on Athlon 64.

	float [MFLOPS]		*short integer [MIPS]*	
	Pentium 4	Athlon 64	Pentium 4	Athlon 64
Standard	795	719	1443	925
Scalar best	992	1273	2322	2082
Vector best	5298	3513	9200	9983

Table 3: Best obtained performance.

similar behavior as for data type float. The vector code reaches close to 10 GIPS for a speed-up of up to 5 times over scalar code.

Parameters found. For FWD and float, the parameters found are shown in Table 4.

5.3 Additional Experiments

The best achieved performance from the above experiments on both platforms is summarized in Table 3.

Below we discuss in greater detail the relative performance of our generated scalar and vector implementations on Pentium 4 and Athlon 64 across both data types, investigate the benefit of FWIabc (Fig. 14), and investigate the benefits of adaptation (Fig. 15). Note that the scales in the plots differ.

Scalar code vs. standard implementation. Fig. 14(a) compares our best scalar code to the standard implementation. The speed-up is higher on Athlon (40–80% for float and around 100% for short integer) than on Pentium, and higher for short integer than for float.

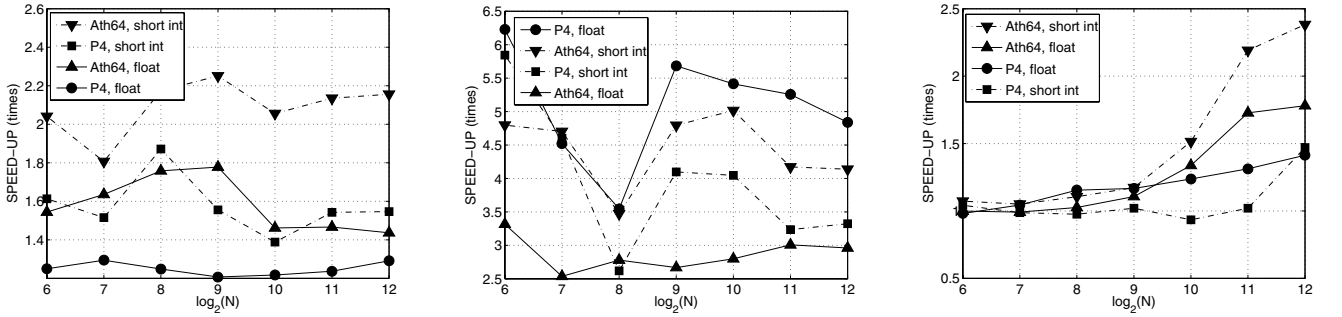
As an additional experiment (not shown) we modified ZM-FWR to use the data type short integer; this resulted in a 70% lower performance than the standard implementation on Pentium 4, and a 30% lower performance on Athlon 64.

Vector code vs. scalar code. Fig. 14(b) compares the performance of our best vector implementations with our best scalar implementations. For 4-way float the speed-up is around 2.5–3 on Athlon and around 5 for Pentium. The latter compensates for the poor scalar performance. For 8-way short integer the speed-up is 4–5 on Athlon and 3–4 on Pentium.

Effect of the optimization using FWIabc. We evaluate the benefit of the subroutine FWIabc. Fig. 14(c) shows the performance gain of the best generated vector code over the best generated vector code without using FWIabc. It shows considerable gains for sizes outside the L2 cache: up to a factor of more than 2 for Athlon and short integer. The scalar code did not benefit significantly from FWIabc.

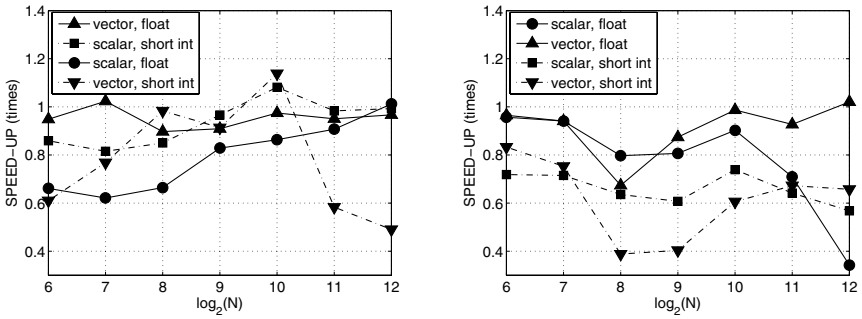
Effect of adaptation. To show the need for platform adaptation, we performed crosstiming experiments. First, we measure the best code generated for the Athlon on the Pentium and record the speed-up to the best performance on the Pentium 4. Fig. 15(a) summarizes the results. As expected the speed-up is generally smaller than 1, i.e., a slow-down. Up to 40% can be lost through porting. For float vector code, the loss is marginal.

Fig. 15(b) shows the same experiment with the roles reversed. The result are comparable.



(a) Best scalar vs. standard scalar (b) Best vector vs. best scalar (c) Gain through FWIabc (vector code)

Figure 14: Speed-up on Pentium 4 and Athlon 64 for both float (solid) and short integer (dashed).



(a) Pentium 4 (b) Athlon 64

Figure 15: Crosstiming between Pentium 4 and Athlon 64 for both float (solid) and short integer (dashed). The best generated implementation for one platform is benchmarked on the other platform.

6. CONCLUSIONS

We believe that domain-specific program or library generators are the future, at least for well-understood numerical kernel functionality. The science of building these generators is still in a very early stage, as this is a rather recent trend and only few examples exist so far. To further advance the area it is thus necessary to explore program generation for other domains and the present paper is a contribution in this direction.

On the algorithmic side, the main contribution of the paper is the subroutine FWIabc, which does not incur the dependencies of FW and can hence be structured like a matrix-matrix multiplication with the k-loop as the innermost loop. This in turn enabled us to follow closely the approach of ATLAS, but to go beyond it through two levels of cache blocking and SIMD vectorization.

The experimental results offer some surprises, at least to the authors, which is typical for the domain of performance optimization and one of the main reasons why empirical search has become a popular optimization strategy. Understanding and modeling the performance behavior or the parameters found would be worthwhile. Finally, it is worth pointing out the considerable, sometimes even superlinear, speed-up that we obtained by vectorization. Vectorization is often neglected in work on performance optimization but is mandatory if highest performance is desired.

7. ACKNOWLEDGEMENTS

This work was supported by NSF through awards 0234293 and 0325687 and by DARPA under DOI grant number NBCH-1050009. Franz Franchetti was supported by the Austrian Science Fund FWF's Erwin Schroedinger Fellowship J2322.

8. REFERENCES

[1] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishanmoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[2] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, Upper Saddle River, 1992.

[3] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. Quintana-Orti, and R. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[4] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. In *STOC '03: Proceedings of the 35th ACM Symposium on Theory of*

	scalar					*vector*				
N	U_i, U_j	U'_i, U'_j, U'_k	L_1, L_2	FLOPS	Runtime [s]	$U_i, \frac{U_j}{\nu}$	$U'_i, \frac{U'_j}{\nu}, U'_k$	L_1, L_2	FLOPS	Runtime [s]
Pentium-4 float										
64	16,1	8,1,1	16,32	410	0.00128	4,2	2,2,16	16,32	2482	0.000211
128	16,1	16,1,1	32,64	615	0.00681	8,4	2,2,16	32,64	2787	0.00150
256	8,1	16,1,1	32,64	926	0.0362	8,4	2,2,16	32,128	3413	0.00983
512	4,2	8,1,8	32,256	919	0.292	2,4	2,2,8	32,256	5206	0.0516
1024	4,2	8,1,8	32,512	945	2.27	8,4	4,1,16	32,64	5125	0.419
2048	4,1	8,1,8	32,1024	955	17.98	4,4	2,4,16	32,64	4897	3.51
4096	8,4	8,1,8	32,1024	948	144.92	1,4	2,4,32	32,2048	4586	29.97
Athlon64 float										
64	16,8	16,1,16	16,32	523	0.00100	4,2	4,1,16	16,32	1546	0.000339
128	16,8	8,1,8	32,64	775	0.00541	4,4	4,1,32	32,64	1811	0.00232
256	16,4	8,1,32	32,128	1264	0.0265	8,2	4,1,32	32,64	3341	0.0100
512	16,8	8,1,16	16,256	1272	0.211	16,4	4,1,32	32,256	3255	0.0824
1024	16,8	16,2,16	16,64	998	2.15	16,4	4,1,32	32,64	2837	0.757
2048	8,16	4,4,32	64,512	1030	16.68	4,4	4,1,32	32,128	2797	6.14
4096	8,32	16,4,16	128,512	725	189.46	8,4	2,4,32	32,2048	3034	45.30

Table 4: Parameters for FWD found by the search.

Computing, pages 159–166, New York, NY, USA, 2003. ACM Press.

[5] J. Demmel, J. J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):342–357, 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[6] S. E. Dreyfus and A. M. Law. *The Art and Theory of Dynamic Programming*. Academic Press, New York, NY, 1977.

[7] M. Elkin. Computing almost shortest paths. *ACM Transactions on Algorithms*, 1(2):283–323, 2005.

[8] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 169–180, 1999.

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

[10] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA '05: Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, Philadelphia, PA, USA, 2005.

[11] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. Flame: Formal linear algebra methods environment. *TOMS*, 27(4):422–455, December 2001.

[12] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int'l Journal of High Performance Computing Applications*, 18(1), 2004.

[13] Intel Corporation. Intel C++ compiler documentation, 2005. Document number 304968-005.

[14] R. Johnsonbaugh and M. Kalin. A graph generation package. www.depaul.edu/∼rjohnson/source.

[15] J. Moy. *OSPF version 2*. RFC 2328, April 1998.

[16] J.-S. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[17] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[18] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *J. Experimental Algorithms*, 8:2.2, 2003.

[19] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000.

[20] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS. *Proceedings of the IEEE*, 93(2):358–386, 2005. Special issue on "Program Generation, Optimization, and Adaptation".