OPTIMIZING ALL-PAIRS SHORTEST-PATH ALGORITHM USING VECTOR INSTRUCTIONS

Sungchul Han and Sukchan Kang

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

In this paper, we present a vectorized version of the Floyd-Warshall shortest-path algorithm to improve the performance. The vectorized implementation utilizes the single instruction multiple data (SIMD) instructions available in Intel architectures. The experimental results show that the instruction-level parallelism obtained by simultaneously processing 8 nodes combined with unrolling yields between 2.3 and 5.2 times of speed-up over the existing blocking algorithms, which only utilize cache locality.

1. INTRODUCTION

1.1. Motivation

The rapid increase in the transistor count and the speed in the modern processors have resulted in a variety of state-of-the-art features in modern processors such as pipelining, simultaneous multithreading, vector instructions, and cache hierarchy. However, straight-forward implementations of many numerical algorithms cannot fully exploit these features. Thus, for the best result, the original algorithm has to be modified to utilize these features. The appropriate approach often depends on the specific algorithm to be optimized.

In this paper, we aim to develop a new implementation of the Floyd-Warshall algorithm by utilizing the vector instructions available in state-of-the-art architectures. The Floyd-Warshall algorithm is a representative shortest-path algorithm that is widely used in computer networks, computer aided design for integrate circuits, and many academical research problems. Since most shortest-path algorithms are similar in nature, the knowledge gained in this paper from the FW algorithm will be useful to find the set of optimization techniques that can be applied to other shortest-path algorithms in general.

1.2. Previous Work and State-of-the-Art

The all-pairs shortest-path problem is a well-known problem in graph theory in which we want to find the shortest path for all source-destination pairs within a given graph. If there are no negative-length cycles, the solution can be obtained by the Floyd-Warshall (FW) algorithm[1]. A number of papers discussed the optimization of the FW algorithms in terms of cache performance. Venkataraman et al.[3] presented a tiled implementation of the FW algorithm and reported a performance improvement of a factor of two against the straight-forward implementation. Further improvements were made by Joon-Sang Park et al.[2], who presented a recursive implementation with block data layout (BDL) that maps a tile of data, instead of a row, into a contiguous memory.

Both of these papers concentrated on the exploitation of data locality to improve the cache performance, but neither of these worked on the parallel execution of multiple instructions. The blocking techniques previously developed in these papers will be our starting point to make further improvements.

1.3. Overview

First, the performance of the blocked versions of the FW algorithms was analyzed for the test platform of our choice. These include the straight-forward iterative implementation (FWI), the recursive version (FWR), and the tiled version (FWT). Further experiments were conducted using unrolling, which is known to be a very effective technique for matrix-matrix multiplication and the fast Fourier transforms[4, 5]. Finally, we modified the algorithm to use vector instructions, specifically Intel single instruction multiple data extensions 2 (SSE2), which provides eight parallel arithmetic or logical operations on 16-bit integer data[6]. To find the best vectorized implementation, the performances of various unrolled versions were compared.

1.4. Organization of the Paper

The rest of the paper is organized as follows. Section 2 gives some backgrounds on the original FW algorithm and the blocked versions that are used to improve the cache performance. In Section 3, the setup for experiments will be explained with some preliminary results on the existing algorithms. In Section 4, the proposed vectorized FW algo-

The authors would like to thank Dr. Markus Püschel and Dr. Franz Franchetti for technical guidance and advice throughout the course 18799.

rithm is introduced. The experimental results of the proposed algorithm will be presented in Section 5. Finally, we offer conclusions in Section 6.

2. NECESSARY BACKGROUND

2.1. Floyd-Warshall Algorithm

The input of the FW algorithm is a $N \times N$ distance matrix D, in which the element $D(i, j)$ is initialized to the weight of the edge from node i to node j, or set to ∞ when there is no connection from i to j. The FW algorithm runs for N iterations and yields the output matrix D, where $D(i, j)$ now indicates the length of the shortest path from i to j. Thus, it is an in-place algorithm that overwrites the result of each iteration to the input matrix. If the reconstruction of the actual shortest path is desired, an additional output matrix V is also generated. The element $V(i, j)$ indicates the most recently added intermediate node between i and j. The pseudo-code for the straight-forward implementation is shown in Table 1.

```
function FW(D, V)
  for k=1:N
    for i=1:N
      for j=1:N
        sum = D[i][k]+D[k][j];
        if (sum<D[i][j])
          D[i][j] = sum;
          V[i][j] = k;
```

Table 1. Floyd-Warshall by definition

The operation of the straight-forward implementation (FW) can be intuitively understood. The initial content of the distance matrix correspond to the shortest paths for all pairs without allowing any intermediate nodes. In the first iteration ($k = 1$), node 1 is allowed as a potential intermediate node for all source-destination pairs. Accordingly, the direct path from i to j, for all i and j, is compared with the new path from i via 1 to j, and the shorter path is chosen. Then, the nodes $k = 2, \ldots, N$ are successively considered as a potential intermediate node. In the k^{th} iteration, the existing shortest path from i to j that does not contain node k is compared with the path from i via k to j. After N iterations, all possible intermediate nodes will have been considered, and the distance matrix will contain the lengths of the shortest paths. Whenever a new path containing the newly added intermediate node k is chosen, the node number k is stored at the corresponding position in the matrix V. The actual shortest paths can be reconstructed by recursively gathering the nodes recorded in the the matrix V. This matrix will be referred to as the "via" matrix in the rest of the paper.

2.2. Blocked Floyd-Warshall Algorithms

First, we define a generalized version of the FW algorithm called FWI as in Table 2. Unlike FW, FWI accepts three matrices A,B and C, and writes the result to the matrix A. Obviously, FWI(D,D,D,V) yields the same result as FW(D,V).

```
function FWI(A, B, C, V)
  for k=1:N
    for i=1:N
      for j=1:N
        sum = B[i][k]+C[k][j];
        if (sum<A[i][j])
          A[i][j] = sum;
          V[i][j] = k;
```

Table 2. Generalized Iterative FW Algorithm (FWI)

The triple-loop structure of the FWI seems very similar to matrix-matrix multiplication. In matrix-matrix multiplication, the order of i, j, and k does not affect the final result, and thus can be freely changed. In FW, on the other hand, k has to be the outermost loop to produce correct results while i and j can be done in any order. However, under certain conditions, the k-loop can be put inside the i-loop and j-loop, making blocking possible. By appropriate reordering of i, j, and k, FWI(A,B,C,V) with $N \times N$ matrices can be performed in a blocked manner, i.e., FWI(A,B,C,V) with $P \times P$ matrices are invoked $(N/P)^2$ times, where P is the subblock size after blocking. Therefore, it is possible to perform FWI recursively, and the recursive version will be referred to as FWR for the rest of the paper. The code for FWR is shown in Table 3. There is also a tiled version of FW, which is simply a recursion by only one level. This will be referred to as FWT. (In this section, only the final form of the blocked algorithm was introduced. Refer to [2] for a proof.)

```
function FWR(A, B, C, V)
  N : input size;
  P : subblock size;
  A[i,j] : PxP submatrix (i,j) of A, i.e.,
           A[(i-1)*P+1:i*P][(j-1)*P+1:j*P];
  M = N/P;
  if (N<=Base)
    FWI(A,B,C,V);
  else
    for k=1:M
      FWR(A[k,k],B[k,k],C[k,k],V[k,k]);
      for j=1:k, j!=k
        FWR(A[k,j],B[k,k],C[k,j],V[k,j]);
      for i=1:k, i!=k
        FWR(A[i,k],B[i,k],C[k,k],V[i,k]);
      for i=1:k, i!=k
        for j=1:k, j!=k
          FWR(A[i,j],B[i,k],C[k,j],V[i,j]);
```

Table 3. Blocked Recursive FW Algorithm (FWR)

When the via matrix is not included, the operations counts for all variants of the blocked versions are the same as that of the original FW, which is $2N^3$ integer additions, counting a comparison and a minimum operation as two operations. For the via matrix, however, it is not clear what should be the operations count. Basically, it involves one comparison, but if this can be shared with the minimum operation for the distance matrix as shown in Table 2, there is no arithmetic cost. If minimum operators are provided by the compiler,

we may need a separate comparison for the via matrix since the result of the implicit comparison in the minimum operator may not be accessible. This is true when we use the vector minimum operators. Furthermore, we had to use three logical operators (i.e., four integer operations in total) for the via matrix in efforts to reduce the branch instructions in the compiled codes. For fair comparison between conventional algorithms and the vectorized algorithms to follow, we assumed an operation count of $6N^3$ integer operations for any FW algorithm with the via matrix.

3. PRELIMINARY EXPERIMENTS

3.1. Setup of Experiments

For the experiments with the FW algorithms, a number of directed graphs with random weights from 1 to 10 have been generated using a graph generation package provided by R. Johnsonbaugh and M. Kalin[7]. The number of nodes N in the graphs was constrained to be power of two, and the number of edges in the graph was set to $N^2/3$.

The target system was a laptop with a 1600MHz Pentium-M processor equipped with 32KB I-cache, 32KB of D-cache, and 1MB of L2-cache. The main memory was 1GB. As a preliminary step for optimizing the FW algorithm, the optimal parameters for blocked algorithms were searched for by experiments. The optimum parameters found were as follows.

- For FWR, blocking factor of 2 and base size of 256
- For FWT, tile size of 256

The data type of the distance matrix D and the via matrix V was defined as 16-bit integers. This makes it possible to vectorize eight integer additions with the SSE2 128-bit registers. Also, since we assumed that each edge has an integer weight from 0 to 10, 16 bits are usually enough to represent the longest path for fairly large graphs.

The number of 16-bit words that can fit in the L2 cache is 512K words, and each FWI(A,B,C) involves three matrices. Since $\sqrt{512k/3} = 418$, the optimal base size (or tile size) of 256 roughly corresponds to the L2 cache. This indicates that the blocking should be performed for the L2 cache instead of the L1 cache since the miss penalty of L2 cache of Pentium-M processor is very small.

3.2. Performance of the Blocked Algorithms

With the optimal parameters, the performance of the blocked algorithms was compared against the straight-forward implementation FWI. As shown in Fig. 1 with the via matrix and in Fig. 2 without via matrix, the recursion-all-the-way strategy yields the poorest performance due to the excessive recursion overhead. On the other hand, regardless of the via matrix, the blocked algorithms FWT and FWR show similar performances as the simple FWI, and begin to outperform for fairly large input sizes ($N = 2^{10}$ or more). This is due to the small data width (2-byte) in comparison with large cache

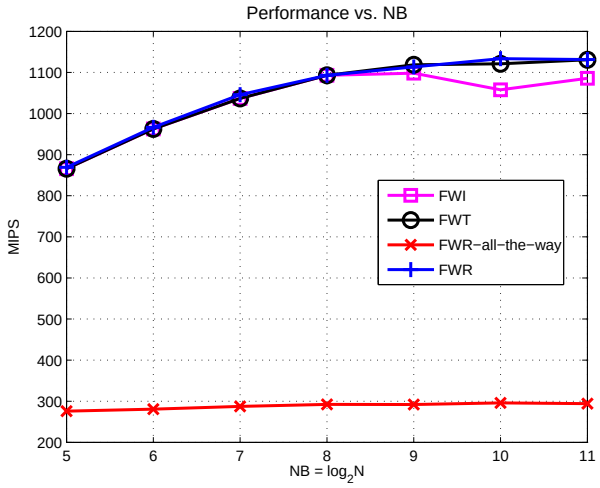

Fig. 1. Comparison of blocked algorithms (with via)

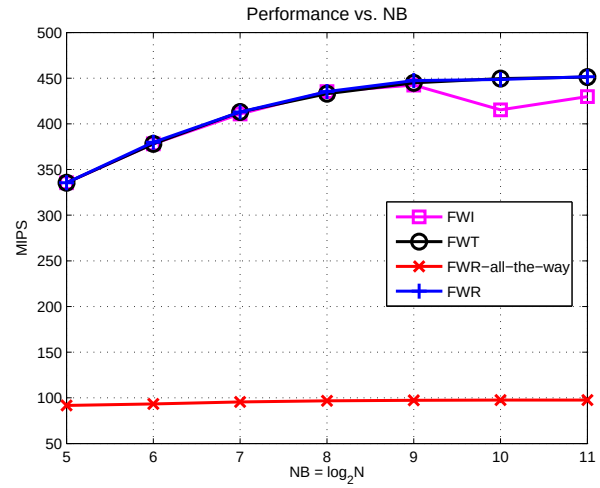

Fig. 2. Comparison of blocked algorithms (without via)

memory. We could not measure the performance for larger input sizes due to excessive runtime, but we could measure the performance with different data types. With 32-bit integers, the blocked algorithms were about 20% better than FWI. Therefore, for the input sizes considered in this paper, i.e., $N = 2^5, \ldots, 2^{11}$, with 2-byte integer types, there is little chance of improving the performance by solely blocking.

3.3. Effect of Unrolling

Initially, the performance of unrolling was measured for FWI, FWR, and FWT where the 2×2 and 4×4 micro-blocks were unrolled. However, the performance of any unrolled FW algorithm was only about 60% of their non-unrolled counterparts. We do not know exactly why unrolling has this adverse effect, but it seems to aggravate the branch penalty caused by the 'if' clause in the FW algorithm. In assembly level, there are conditional load statements (a.k.a.

predicated move mnemonics) that can implement the FW algorithms without incurring a branch. However, we could not consistently prevent the compiler from using branch instructions within the C++ language syntax.

4. PROPOSED OPTIMIZATION

For higher cache performance, we divided the input matrix into small tiles of appropriate size and performed each tile with vectorized FW routines. Then, unrolling was applied again to alleviate the loop overhead.

4.1. Vector Instructions

The Intel SSE2 instruction set allows the packing of eight 16-bit integers into one 128-bit register. The minimum operator for the distance matrix ($A, B,$ or C) could be simply implemented with a vector minimum intrinsic. However, the via matrix (V) was performed using 'compare', 'and', 'andnot', 'or' intrinsics with an aim to eliminate branch instructions. The main idea is that the conditional assignment $Y = (A < B)?C : D$ can be done by two instructions $M = A < B; Y = M \cdot C + \bar{M} \cdot D$, where M is a mask with all ones or all zeros depending upon the result of $A < B$, $\bar{M}$ is the one's complement of M, '$\cdot$' is a logical AND operator, and '$+$' is a logical OR operator.

4.2. Tiling

A modified FWI with 1x8 vector instructions (FWI-1x8) was designed, where the innermost loop variable j increments by 8. For each value of k and i, eight operations corresponding to eight j's are performed simultaneously. With FWI-1x8, the performance dependency on the tile size was also changed. To find out the optimal tiling parameters, the performance of one-level tiling was measured for various tile sizes. And then, the optimal tile size for two-level tiling was searched for with the inner tile size set to the previously found value. The result is as follows.

- Tile size 1: 256 (block size after first tiling)
- Tile size 2: 64 (block size after second tiling)

The two-level tiled FW that invokes FWI-1x8 for the base case will be denoted as FWT2-1x8.

4.3. Unrolling

To see the effect of unrolling for vectorized codes, the mid-loop variable i was unrolled. We designed three unrolled versions with the unrolling factor of 2, 4, and 8, which will be called FWT2-1x8-U2, FWT2-1x8-U4, and FWT2-1x8-U8, respectively. In addition, we also designed FWT2-1x16-U8, where the j variable is unrolled by a factor of 16 using two 1x8 vector instructions and the i variable is unrolled by a factor of 8 as before.

5. EXPERIMENTAL RESULTS

We used the gcc 3.3.4 compiler for all of the non-vectorized algorithms since it gave better performance. For vectorized codes, we used Intel icc 8.1 compiler for the same reason.

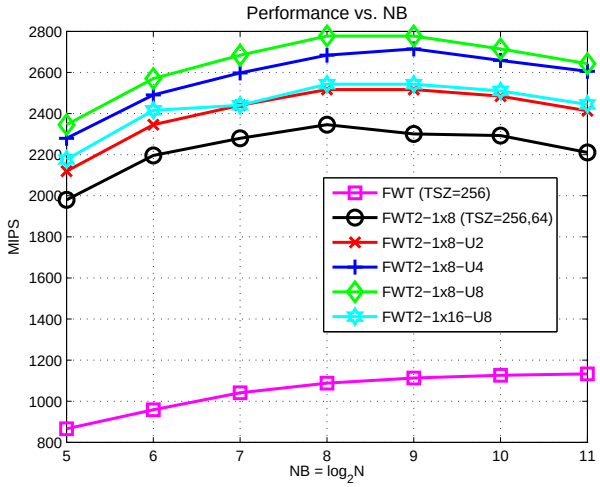

Fig. 3. Performance of vectorized algorithms (with via)

By experiment, the best optimization flags were chosen as follows.

- For gcc, "-O2 -march=pentium-m"
- For icc, "-O3"

The performance of the vectorized algorithms with the via matrix is shown in Fig. 3 along with the conventional blocking algorithm (FWT). It can be seen that between 95% and 130% of speed-up against FWT has been obtained with the non-unrolled version (FWT2-1x8), and between 133% and 170% of increase with the unrolled version (FWT2-1x8-U8). This graph also shows that the higher unrolling factor improves the performance except that the horizontally unrolled version (FWT2-1x16-U8) was only as good as the one unrolled by a factor of 2 (FWT2-1x8-U2).

Even greater improvements were observed for the case without the via matrix, as shown in Fig. 4. For the unrolled version (FWT-1x8), the increase was between 231% and 359%. Unlike the previous case with the via matrix, the best performance was observed from the most unrolled version (FWT2-1x16-U8), which gave an increase between 369% and 417%.

In the results shown in Fig. 3 and 4, the reasons for the improvements by the unrolled versions (FWT2-1x8) seem to be twofold: the parallel execution by vectorization and the elimination of branch instructions. Also, it was shown that, unlike the conventional algorithms, unrolling significantly improved the performance.

6. CONCLUSIONS

In this paper, we showed that the performance improvement for the FW algorithm by blocking was not noticeable for inputs sizes of up to 2^{11}. In addition, the results showed that unrolling did not improve performance for the conventional blocked FW algorithms. We presented a vectorized FW implementation that improved the performance by a factor of

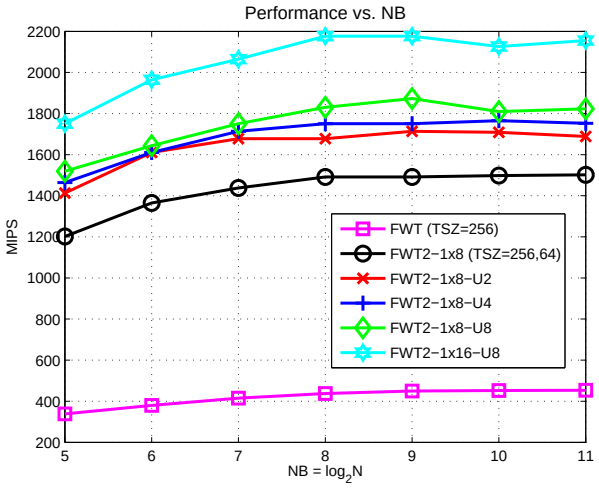

Fig. 4. Performance of vectorized algorithms (without via)

between 2.3 and 5.2 over the conventional blocked algorithms. It has been also observed that unrolling works effectively for vectorized versions. Since the results shown in this paper were acquired from Intel Pentium-M platform only, the experiments on other platforms seem to be an essential step to make this research a complete one. For another further step, we hope to extend the proposed techniques to other routing algorithms that are more popular in networking.

7. REFERENCES

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

[2] Joon-Sang Park, Michael Penner, and Viktor K. Prasanna, "Optimizing graph algorithms for improved cache performance", *IEEE Trans. Parallel and Distributed Systems*, vol. 15, pp. 769-782, Sep 2004.

[3] G. Venkataraman, S. Sahni, and S. Mukhopadhyana, "A blocked all-pairs shortest-paths algorithm", in *Proc. Scandinavian Workshop algorithms and Theory*, 2000.

[4] A. C. McKellar and E. G. Coffman, Jr., "Organizing matrices and matrix operations for paged memory systems", *Commun. ACM*, vol. 12, issue 3, pp. 153-165, 1969.

[5] M. Wolfe, "Iteration space tiling for memory hierarchies", in *Third SIAM Conference on Parallel Processing for Scientific Computing*, Dec 1987.

[6] Intel C++ Compiler User's Guides, `http://support.intel.com/support/performancetools/c/linux/v8/c_ug_lnx.pdf`.

[7] R. Johnsonbaugh and M. Kalin, a graph generation package, `http://condor.depaul.edu/~rjohnson/source/graph_ge.c`