Optimizing the Fast Fourier Transform over memory hierarchies for embedded digital systems: a fully in-cache algorithm

James E. Raynolds
College of Nanoscale Science and Engineering, University at Albany, State University of New York, Albany, NY 12203

Lenore R. Mullin
Computer Science Department, University at Albany, State University of New York, Albany, NY 12203

(Dated: May 28, 2004)

We present an algorithm for the Fast Fourier Transform (FFT) which maximizes the use of in-cache operations. Through repeated transpose and *reshape* operations, all computations can be carried out in cache for the situation in which the array size n is a power of two times the cache size c (also assumed to be a power of two). A formula for the number of transpose/reshape operations is given in terms of c and n. The example: $n = 32$, $c = 4$ is worked out in detail to demonstrate the general principles.

Our approach to maximizing in-cache operations is similar to the approach one might take in partitioning the data for a parallel implementation [1]. That is, the data is divided into blocks that fit into cache and as many cycles of the FFT are performed which can be accommodated in cache. Through a series of operations in which the data is transposed and then *reshaped* (composing indices prior to materialization of the transpose, better known as the *corner turn* [2]), we obtain an algorithm in which all operations are performed in cache. It is important to note that we want to *materialize* the array after it has been transposed and *reshaped* so that components have locality. The most difficult aspect of this optimization is determining *when* to materialize and when not to materialize. These are issues of cost functional analysis. That is to say, if the input vector fits in cache we'd compose the indices (i.e. we *don't* materialize the transpose). Materialization becomes even more important when cache misses propagate to include page faults. In this report we restrict our attention to the case in which the size of the array n is a power of 2 times the cache size c which, in turn is also a power of two. That is, we assume $n = 2^p * c$ for some integer p, and $c = 2^q$ for some integer q.

To illustrate the procedure, in the following example we choose the cache size, $c = 4$, and the input length, $n = 32$. We generate the vector of indices using the *iota* operator: $\iota\ (n)$:

$$\vec{v} \equiv \iota\ (n) =< 0\ 1\ 2\ \cdots\ 31\ > . \tag{1}$$

To conceptualize movement through the cache we abstract the indices as a two-dimensional matrix with rows of length equal to the cache size. The number of rows is then given by $r = n/c = 8$. Explicitly, we use the *reshape* operator $\hat{\rho}$ to define the matrix A (assuming row-major ordering) to be:

$$A \equiv < r\ c > \hat{\rho}\vec{v} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix} . \tag{2}$$

We say that the array A has shape $< r\ c >$, that is, A is an $r \times c$ matrix. This is more succinctly expressed using the *shape* operator ρ (not to be confused with the *reshape* operator $\hat{\rho}$) by the expression $\rho A =< r\ c >$. Two cycles of the FFT are next carried out on each of the rows. That is, $log_2(c)$ operations can be performed before a reorganization is needed. It is also important to note that based on the associativity of the cache, i.e. b-way versus direct, that using a radix b FFT versus radix 2 may be more appropriate [3]. That is, we require $2^p = (2^m)^l$ where $p = m + l$. For example, $2^6 = 64 = (2^3)^2 = 8^2 = (2^2)^3 = 4^3$. Hence, for a vector length 64, radix 2 may be substituted by radix b = 4, 8, or 64.

The well-known access patterns for the first $log_2(c)$ operations (i.e. 2 cycles of the FFT for this example) are indicated schematically in Fig. 1 where the vertical bars indicate the cache boundaries.

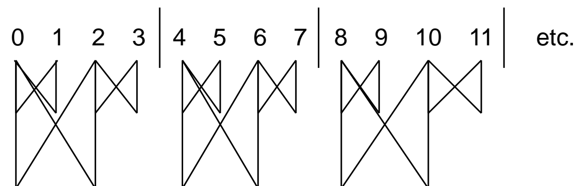

FIG. 1: Schematic illustration of some of the access patterns for first two cycles of the FFT (some patterns omitted for clarity).

To proceed further would require cache misses involving communication between rows. Therefore we transpose the array to give:

$$A^T = \begin{bmatrix} 0 & 4 & 8 & 12 & 16 & 20 & 24 & 28 \\ 1 & 5 & 9 & 13 & 17 & 21 & 25 & 29 \\ 2 & 6 & 10 & 14 & 18 & 22 & 26 & 30 \\ 3 & 7 & 11 & 15 & 19 & 23 & 27 & 31 \end{bmatrix}. \quad (3)$$

Now each row is of length $2*c$ and we want to redistribute the data by *reshaping* the array so that it once again has rows of length equal to the cache size. We write:

$$B \equiv < r\ c > \hat{\rho}\ (A^T) = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 16 & 20 & 24 & 28 \\ 1 & 5 & 9 & 13 \\ 17 & 21 & 25 & 29 \\ 2 & 6 & 10 & 14 \\ 18 & 22 & 26 & 30 \\ 3 & 7 & 11 & 15 \\ 19 & 23 & 27 & 31 \end{bmatrix}, \quad (4)$$

The *reshape* operator $\hat{\rho}$ is used to once again produce the matrix B whose rows are each of length equal to the cache size.

At this point, two more cycles of the FFT are carried out on each of the rows as illustrated in Fig. 2.

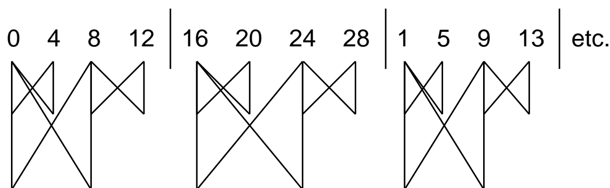

FIG. 2: Schematic illustration of some of the access patterns for next two cycles of the FFT after the transpose/reshape (some patterns omitted for clarity).

Again to proceed further would require cache misses so we transpose the array once again:

$$B^T = \begin{bmatrix} 0 & 16 & 1 & 17 & 2 & 18 & 3 & 19 \\ 4 & 20 & 5 & 21 & 6 & 22 & 7 & 23 \\ 8 & 24 & 9 & 25 & 10 & 26 & 11 & 27 \\ 12 & 28 & 13 & 29 & 14 & 30 & 15 & 31 \end{bmatrix}, \quad (5)$$

and as before we reshape this to give rows of length equal to the cache size:

$$C \equiv < r\ c > \hat{\rho}\ (B^T) = \begin{bmatrix} 0 & 16 & 1 & 17 \\ 2 & 18 & 3 & 19 \\ 4 & 20 & 5 & 21 \\ 6 & 22 & 7 & 23 \\ 8 & 24 & 9 & 25 \\ 10 & 26 & 11 & 27 \\ 12 & 28 & 13 & 29 \\ 14 & 30 & 15 & 31 \end{bmatrix}. \quad (6)$$

At this point the next and final cycle of the FFT is done. There is no need to transform the matrix back (actually moving the data) to yield the final form of the FFT because the composite index mapping from C to A is known.

This is an important point: the transformation from A to B and from B to C are operations in which data is actually moved. Thus in this example there are three materializations: A, B and C. In general for $n = 2^p * c$ (with $c = 2^q$), the number of materializations will be given by computing the fraction:

$$f = \frac{log_2(n)}{log_2(c)}. \quad (7)$$

If f is not an integer we round to the next higher integer. In the above example we have $f = log_2(32)/log_2(4) = 5/2 = 2.5$. Now rounding up to the next integer gives $f \rightarrow 3$, corresponding to the three materializations: A, B, and C required to complete the transform in cache.

Notice that we have been discussing a situation in which $n/c > c$. The case in which $n/c = c$ yields a square matrix A which need only be transposed once (without reshaping) to yield two materializations: A and A^T. For the case in which $n/c < c$ there are still only two materializations but now we must reshape after the transpose. Again we emphasize that the operation of transpose/reshape is composed to yield only one materialization.

Cache optimizations are traditionally done by compilers and are referred to as tiling operations [4, 5]. When access patterns are contiguous a single tiling is often adequate. However, for the FFT multiple tilings for a single cache level are required as illustrated above. It is also possible to obtain optimizations by blocking application code [6]. In this case compiler optimizations would be turned off [7]. Again, blocking usually occurs only once, not multiple times, for each cache level. Our research shows that, by using an algebraic specification of the problem and its mappings, we can describe how to deterministically build processor/memory hierarchy optimizations. Our methods will enable application programmers, who know the algorithm and target architectures, to design and build optimal, scalable, reconfigurable scientific software [8, 9]. We plan to introduce these *mechanizable* transformations into scientific libraries. The ideas

presented in this paper will be extended and a presentation of the algorithm in its complete generality will be presented in a forthcoming publication.

[1] H.Hunt, L. Mullin, and D. Rosenkrantz, in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications(PDPTA'99)* (Las Vegas, NV, 1999), pp. 1641–1647.
[2] L. M. R. Mullin, A Mathematics of Arrays, Ph.D. Thesis, Syracuse University, December 1988.
[3] L. Mullin and S. Small, Journal of Mathematical Modeling and Algorithms **1**, 193 (2002).
[4] K. S. Gatlin and L. Carter, in *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT '00)* (IEEE Computer Society Press, Philadelphia, PA, 2000), pp. 249–260.
[5] L. Carter, J. Ferrante, and S. Hummel, in *International Parallel Processing Symposium (9th IPPS'95)* (IEEE Computer Society Press, 1995).
[6] D. Takahashi, Lecture Notes in Computer Science **2110**, 551 (2001), ISSN 0302-9743.
[7] K. D. Cooper and L. Torczon, *Engineering a compiler* (Elsevier, 2003), pp. 386–391.
[8] L. Mullin, Digital Signal Processing (to appear).
[9] L. Mullin, E. Rutledge, and R. Bond, in *Proceedings of the High Performance Embedded Computing Workshop HPEC 2002* (MIT Lincoln Laboratory, Lexington, MA, 2002).