Cache Oblivious Matrix Operations Using Peano Curves

Michael Bader and Christian Mayer

Institut für Informatik, Technische Universität München, Germany

Abstract. Algorithms are called *cache oblivious*, if they are designed to benefit from any kind of cache hierarchy—regardless of its size or number of cache levels. In linear algebra computations, block recursive approaches are a common approach that, by construction, lead to inherently local data access pattern, and thus to an overall good cache performance[3].

In this article, we present block recursive approaches that use an element ordering based on a Peano space filling curve to store the matrix elements. We present algorithms for matrix multiplication and LU decomposition, which are able to minimize the number of cache misses on any cache level.

1 A block recursive scheme for matrix multiplication

Consider the multiplication of two 3×3-matrices, such as given in equation (1), where the indices of the matrix elements indicate the order in which the elements are stored in memory.

$$\underbrace{\begin{pmatrix} a_0 & a_5 & a_6 \\ a_1 & a_4 & a_7 \\ a_2 & a_3 & a_8 \end{pmatrix}}_{=:A} \underbrace{\begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix}}_{=:B} = \underbrace{\begin{pmatrix} c_0 & c_5 & c_6 \\ c_1 & c_4 & c_7 \\ c_2 & c_3 & c_8 \end{pmatrix}}_{=:C} . \qquad (1)$$

The scheme is similar to a column-major ordering, however, the order of the even-numbered columns have been inverted, which leads to a meandering scheme, which is also equivalent to the basic pattern of a Peano space filling curve. Now, if we examine the operations to compute the elements c_r of the result matrix, we note that the operations can be executed in a very convenient order – from each operation to the next, an element is either reused or one of its direct neighbours in memory is accessed:

$$\begin{tikzcd}
c_0 \mathrel{+}= a_0b_0 \arrow[d] & c_0 \mathrel{+}= a_6b_2 \arrow[r] & c_5 \mathrel{+}= a_5b_4 \arrow[d] & c_6 \mathrel{+}= a_0b_6 \arrow[r] & c_6 \mathrel{+}= a_6b_8 \arrow[d] \\
c_1 \mathrel{+}= a_1b_0 \arrow[d] & c_1 \mathrel{+}= a_7b_2 \arrow[u] & c_4 \mathrel{+}= a_4b_4 \arrow[d] & c_7 \mathrel{+}= a_1b_6 \arrow[u] & c_7 \mathrel{+}= a_7b_8 \arrow[d] \\
c_2 \mathrel{+}= a_2b_0 \arrow[d] & c_2 \mathrel{+}= a_8b_2 \arrow[u] & c_3 \mathrel{+}= a_3b_4 \arrow[d] & c_8 \mathrel{+}= a_2b_6 \arrow[u] & c_8 \mathrel{+}= a_8b_8 \\
c_2 \mathrel{+}= a_3b_1 \arrow[d] & c_3 \mathrel{+}= a_8b_3 \arrow[u] & c_3 \mathrel{+}= a_2b_5 \arrow[d] & c_8 \mathrel{+}= a_3b_7 \arrow[u] & \\
c_1 \mathrel{+}= a_4b_1 \arrow[d] & c_4 \mathrel{+}= a_7b_3 \arrow[u] & c_4 \mathrel{+}= a_1b_5 \arrow[d] & c_7 \mathrel{+}= a_4b_7 \arrow[u] & \\
c_0 \mathrel{+}= a_5b_1 \arrow[r] & c_5 \mathrel{+}= a_6b_3 \arrow[u] & c_5 \mathrel{+}= a_0b_5 \arrow[r] & c_6 \mathrel{+}= a_5b_7 \arrow[u] &
\end{tikzcd} \tag{2}$$

An algorithmic scheme with this spatial locality property can be obtained for any matrices of odd dimensions, as long as we adopt a meandering numbering scheme. However, cache efficiency requires *temporal* locality, as well, in the sense that matrix elements are reused within short time intervals, and will therefore not be removed from the cache by other data. To achieve temporal locality, we combine the scheme with a block recursive approach. Consequently, the element numbering is then also defined by a block recursive meandering scheme.

2 An Element Numbering Based on a Peano Space Filling Curve

Figure 1 illustrates the recursive scheme used to linearise the matrix elements in memory. It is based on a so-called *iteration* of a Peano curve. Four different block numbering patterns marked as P, Q, R, and S are combined in a way to ensure a contiguous numbering of the matrix elements – direct neighbours in memory will always be direct neighbours in the matrix as well.

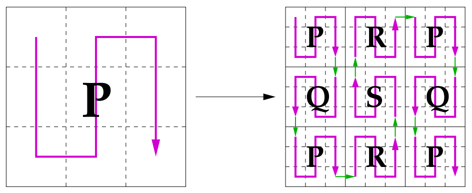

Fig. 1. Recursive construction of a Peano curve (first two iterations)

3 A Block Recursive Scheme for Matrix Multiplication

Equation (3) shows the blockwise multiplication of matrices stored according to the proposed numbering scheme. Each matrix block is named with respect to its numbering scheme and indexed with the name of the global matrix and the position within the storage scheme:

$$\underbrace{\begin{pmatrix} P_{A0} & R_{A5} & P_{A6} \\ Q_{A1} & S_{A4} & Q_{A7} \\ P_{A2} & R_{A3} & P_{A8} \end{pmatrix}}_{=:\,A} \underbrace{\begin{pmatrix} P_{B0} & R_{B5} & P_{B6} \\ Q_{B1} & S_{B4} & Q_{B7} \\ P_{B2} & R_{B3} & P_{B8} \end{pmatrix}}_{=:\,B} = \underbrace{\begin{pmatrix} P_{C0} & R_{C5} & P_{C6} \\ Q_{C1} & S_{C4} & Q_{C7} \\ P_{C2} & R_{C3} & P_{C8} \end{pmatrix}}_{=:\,C} . \tag{3}$$

Analoguous to the 3×3 multiplication in equation (1), we obtain an execution order for the individual block operations. The first operations are

$$P_{C0} \mathrel{+}= P_{A0}P_{B0} \quad \rightarrow \quad Q_{C1} \mathrel{+}= Q_{A1}P_{B0} \quad \rightarrow \quad P_{C2} \mathrel{+}= P_{A2}P_{B0} \quad \rightarrow \quad \ldots$$

If we only consider the ordering scheme of the matrix blocks, we obtain eight different types of block multiplications:

$$\begin{array}{llll} P \mathrel{+}= PP & Q \mathrel{+}= QP & R \mathrel{+}= PR & S \mathrel{+}= QR \\ P \mathrel{+}= RQ & Q \mathrel{+}= SQ & R \mathrel{+}= RS & S \mathrel{+}= SS\,. \end{array} \tag{4}$$

All eight types of block multiplication lead to multiplication schemes similar to that given in equation (2), and generate inherently local execution orders. Thus, we obtain a closed system of eight block multiplication schemes which can be implemented by a respective system of nested recursive procedures. The resulting algorithm has several interesting properties concerning cache efficiency:

1. the number of cache misses on an ideal cache can be shown to be asymptotically minimal [1];
2. on any level of recursion, after a matrix block has been used, either itself will be directly reused or one of its direct neighbours in space will be used; as a result:
3. precise knowledge for prefetching is available.

4 A Block Recursive Scheme for LU Decomposition

Based on the presented numbering scheme, we can also try to set up a block recursive algorithm for LU decomposition. Hence, consider the following decomposition of block matrices:

$$\underbrace{\begin{pmatrix} \lfloor\underline{P_{L0}} & 0 & 0 \\ Q_{L1} & \lfloor\underline{S_{L4}} & 0 \\ P_{L2} & R_{L3} & \lfloor\underline{P_{L8}} \end{pmatrix}}_{=:\,L} \underbrace{\begin{pmatrix} \overline{P_{U0}}\rceil & R_{U5} & P_{U6} \\ 0 & \overline{S_{U4}}\rceil & Q_{U7} \\ 0 & 0 & \overline{P_{U8}}\rceil \end{pmatrix}}_{=:\,U} = \underbrace{\begin{pmatrix} P_{A0} & R_{A5} & P_{A6} \\ Q_{A1} & S_{A4} & Q_{A7} \\ P_{A2} & R_{A3} & P_{A8} \end{pmatrix}}_{=:\,A} . \tag{5}$$

Again, we obtain a set of block operations, which we can try to put into an execution order that preserves locality. However, in contrast to matrix multiplication, we now have to obey certain precedence rules. Unfortunately, these precedence rules deny us a scheme that is strictly memory local – however, we can still try to minimise the non-localities, which may lead us to the following scheme:

1) $\lfloor\underline{P_{L0}}\,\overline{P_{U0}}\rceil = P_{A0}$ – LU decomp.
2) $Q_{L1}\,\overline{P_{U0}}\rceil = Q_{A1}$ – solve for Q_{L1}
3) $P_{L2}\,\overline{P_{U0}}\rceil = P_{A2}$ – solve for P_{L2}
4) $\lfloor\underline{P_{L0}}\,P_{U6} = P_{A6}$ – solve for P_{U6}
5) $\lfloor\underline{P_{L0}}\,R_{U5} = R_{A5}$ – solve for R_{U5}
6) R_{A3} -= $P_{L2}R_{U5}$ – matr. mult.
7) S_{A4} -= $Q_{L1}R_{U5}$ – matr. mult.
8) $\lfloor\underline{S_{L4}}\,\overline{S_{U4}}\rceil = S_{A4}$ – LU decomp.
9) Q_{A7} -= $Q_{L1}P_{U6}$ – matr. mult.
10) P_{A8} -= $P_{L2}P_{U6}$ – matr. mult.
11) $R_{L3}\,\overline{S_{U4}}\rceil = R_{A3}$ – solve for R_{L3}
12) $\lfloor\underline{S_{L4}}\,Q_{U7} = Q_{A7}$ – solve for Q_{U7}
13) P_{A8} -= $R_{L3}Q_{U7}$ – matr. mult.
14) $\lfloor\underline{P_{L8}}\,\overline{P_{U8}}\rceil = P_{A8}$ – LU decomp.

We note that an additional LU decomposition scheme has to be derived for S-numbered blocks: $\lfloor\underline{S_{L4}}\,\overline{S_{U4}}\rceil = S_{A4}$. In addition, there are two further types of schemes to be derived (analogous to LU decomposition):

1. Solve a matrix equation such as $Q_L\,\overline{P}\rceil = Q_A$ for the matrix Q_L, where $\overline{P}\rceil$ is an already computed upper triangular matrix; in the same manner solve $P_L\,\overline{P}\rceil = P_A$, $R_L\,\overline{S}\rceil = R_A$, and $S_L\,\overline{S}\rceil = S_A$ for P_L, R_L, and S_L, respectively.
2. Solve a matrix equation such as $\lfloor\underline{P}\,P_U = P_A$ for the matrix P_U, where $\lfloor\underline{P}$ is an already computed lower triangular matrix; in the same manner solve $\lfloor\underline{P}\,R_U = R_A$, $\lfloor\underline{S}\,Q_U = Q_A$, and $\lfloor\underline{S}\,S_U = S_A$, for R_U, R_U, and R_U, respectively.

Finally, we again obtain a system of nested block recursive schemes to compute the LU decomposition of a matrix. With respect to locality properties, the scheme is not quite as nice as that for matrix multiplication. However, it still leads to a cache oblivious algorithm.

5 Performance, Conclusion

Implementations of the presented algorithmic schemes show that the cache oblivious approach is very successful regarding its core target: it leads to excellent cache hit rates, which are more than competitive with current BLAS/LAPACK implementations. However, the recursive approach does not combine well with processor features such as streaming SIMD extensions (SSE), and similar. Hence, on the last level of recursion, strongly processor specific implementations are required. At least until there is something like a *processor oblivios* approach for this kind of optimization.

References

1. Bader, M., Zenger, C.: Cache oblivious matrix multiplication using an element ordering based on a Peano curve. Linear Algebra and its Applications, submitted.
2. Bader, M., Zenger, C.: A Cache Oblivious Algorithm for Matrix Multiplication Based on Peano's Space Filling Curve. Proc. of the PPAM 2005, accepted.
3. Gustavson, F. G.: Recursion leads to automatic variable blocking for dense linear-algebra algorithms. IBM Journal of Research and Development 41 (6), 1999.