Matrix Transposition

Monday, March 13, 2017 12:15 PM

Problem Statement

Given a 2-dimensional matrix M, compute M^T.

Simple Solution: Iteratively Reducing Mirror

Swap the corresponding elements of each outer row and column. Reduce the problem to ignore the outer shell and repeat on the new smaller matrix. If the edge length of the matrix is 0, the problem has been solved.

Define $n \times n$ matrix M	$n = 3$ $M = \begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix}$
Define M_h and M_v as the sets of elements along the top and left of M, respectivly. Define N as the submatrix without those elements.	$M = \begin{bmatrix} \overbrace{M_h\{a1}^{M_v} & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix}$ $N = \begin{bmatrix} b2 & c2 \\ b3 & c3 \end{bmatrix}$
Swap every element in M_h with is corresponding element in M_v	$M = \begin{bmatrix} a1 & a2 & a3 \\ b1 & b2 & c2 \\ c1 & b3 & c3 \end{bmatrix}$
If n is not zero for N, solve N	$N = \begin{bmatrix} b2 & c2 \\ b3 & c3 \end{bmatrix}$ $N = \begin{bmatrix} b2 & b3 \\ c2 & c3 \end{bmatrix}$ $O = [c3]$ $O = [c3]$ $P = [\,]$ $M = \begin{bmatrix} a1 & a2 & a3 \\ b1 & b2 & b3 \\ c1 & c2 & c3 \end{bmatrix}$

Time Complexity:

The algorithm does n-1 swaps at each level, where n is the side length of M.
This leads to the recurrence $T(n) = T(n-1) + n - 1$, T(0)=0.
Solving this recurrence leads to $T(n) = 0 + n(n-1) = n^2 - n$
Therefore $T(n) \in O(n^2)$

Locality

The algorithm above works well when the matrix is small, but doesn't perform well with larger sizes of n due to poor temporal and spatial locality during the swap procedure. We can alleviate this by noticing that we can perform the same type of operation on groups of elements, not just on the elements themselves. We can therefore subdivide the matrix into arbitrarily small sub-matrices with better locality, solve them, and then merge with excellent temporal locality by using buffered copying of the sub-matrices. If we subdivide until 3 of our sub problems fit in the cache, we can do the swap without incurring the overhead of caching a page into memory more than once for each sub problem.

Define an $n \times n$ matrix $\mathbb{M}$, where each element is a sub-matrix *that is $\leq \frac{1}{3}$ the size of the cache*	$\mathbb{M} = \begin{bmatrix} \mathbb{A}1 & \mathbb{A}2 \\ \mathbb{B}1 & \mathbb{B}2 \end{bmatrix}$
Using the algorithm above, transpose each element of $\mathbb{M}$	$\begin{bmatrix} \mathbb{A}1^T & \mathbb{A}2^T \\ \mathbb{B}1^T & \mathbb{B}2^T \end{bmatrix}$
Using the algorithm above, transpose $\mathbb{M}$	$\begin{bmatrix} \mathbb{A}1^T & \mathbb{B}1^T \\ \mathbb{A}2^T & \mathbb{B}2^T \end{bmatrix}$ $[\mathbb{B}2^T]$ $[\,]$

Note that this method can be used instead of the simple algorithm on a matrix to solve it, though it takes longer to partition the elements.

Parallelization

Parallelizing this algorithm isn't particularly tricky. For both algorithms, solving the subproblem is independent of doing the required step, meaning that the solve step of each algorithm can be parallelized across any number of processors without synchronization. The first algorithms merge step is similarly independent, and is technically non-existent. The second algorithm however requires that all subproblems be solved before the merge step occurs, or at least requires each pair of subproblems to be solved before swapping. Depending on our hardware, we have a couple of choices.

Most likely, we can give some information to our kernel and have it write the required values to an output buffer on the device. we can then read this buffer back out once all the kernels have finished. If we're on a device with limited memory, we further chunk the problems into groups of problems that can be done on the hardware, then do the merge step during the read for each problem set by swapping which host buffers receive the data.