Algorithmic cache management

Consider a computer with unlimited memory and a cache of size $H$

- we can design algorithms by manually managing cache transfers
- simple metrics:
  - amount of data moved from memory to cache (bandwidth cost)
  - number of synchronous memory-to-cache transfers (latency cost)
- generally, efficient algorithms in this model try to select blocks of computation that minimize the surface-to-volume ratio
  - i.e., do as much computation with the cache-resident data as possible
  - in other words, exploit temporal and spatial locality
Cache-efficient matrix multiplication
Consider multiplication of \( n \times n \) matrices \( C = A \cdot B \)

For \( i \in [1, n/s], j \in [1, n/t], k \in [1, n/v] \), define blocks \( C[i, j], A[i, k], B[k, j] \) with dimensions \( s \times t, s \times v, \) and \( v \times t, \) respectively

\[
\begin{align*}
\text{for } (i = 1 \text{ to } n/s) \\
\quad \text{for } (j = 1 \text{ to } n/t) \\
\quad \quad \text{initialize } C[i, j] = 0 \text{ in cache} \\
\quad \quad \text{for } (k = 1 \text{ to } n/v) \\
\quad \quad \quad \text{load } A[i, k] \text{ into cache} \\
\quad \quad \quad \text{load } B[k, j] \text{ into cache} \\
\quad \quad \quad C[i, j] = C[i, j] + A[i, k] \times B[k, j] \\
\quad \quad \text{end} \\
\quad \text{write } C[i, j] \text{ to memory} \\
\text{end} \\
\text{end}
\end{align*}
\]

Q: What restriction must we impose to insure \( A[i, k], B[k, j] \) and \( C[i, j] \) fit in cache simultaneously?
A: \( st + sv + vt \leq H \)
Memory-bandwidth analysis of matrix multiplication

So we have the constraint, \( st + sv + vt \leq H \)
- there are a total of \( \frac{n}{s} \cdot \frac{n}{t} \cdot \frac{n}{v} \) inner loop iterations
- Q: what is the asymptotic memory latency cost of the algorithm
- A: the number of inner loop iterations, \( \frac{n^3}{stv} \)
- since each block of \( C \) stays resident in the innermost loop, we write each element of \( C \) to memory only once
- we read each block \( s \times v \) block of \( A \) and \( v \times t \) block of \( B \) in each innermost loop
- Q: how many times do we read each element of \( A \) and \( B \)?
- A: \( \frac{n}{t} \) and \( \frac{n}{s} \), respectively
- therefore, the bandwidth cost is
  \[ Q = n^2 + (\frac{n}{s} + \frac{n}{t})n^2 = n^2 + \frac{n^3}{s} + \frac{n^3}{t} \]
- if we pick \( s = t = v = \sqrt{H/3} \), we satisfy the constraint and obtain \( Q \approx 2\frac{n^3}{\sqrt{H/3}} \), with \( \frac{n^3}{H^{3/2}} \) memory latency cost
- if we pick \( s = t = \sqrt{H - 2\sqrt{H}} \) and \( v = 1 \), we obtain \( Q \approx 2\frac{n^3}{\sqrt{H}} \) with \( \frac{n^3}{H} \) memory latency cost
Memory-bandwidth cost of LU decomposition

For most dense linear algebra problems, achieving good bandwidth cost is strictly easier in the sequential case than in the parallel case

- example: non-pivoted LU factorization
- we can use the same recursive algorithm, two recursive calls, $O(1)$ matrix multiplications
- $T(n, H) = 2T(n/2) + O(\nu \cdot n^3 / \sqrt{H})$ where $\nu$ is inverse of memory bandwidth
- cost decreases geometrically by factor of 4 with each level, we can stop at base case dimension $n_0 = \sqrt{H}$ and compute LU sequentially
- memory latency cost is just $O(n^3 / H^{3/2} \cdot \nu)$, same as matrix multiplication
- Q: given memory bandwidth cost $O(n^3 / \sqrt{H} \cdot \nu)$, why is it not possible to have less than a $\Theta(n^3 / H^{3/2})$ memory latency cost?
- A: we cannot transfer messages larger than the cache size $H$
Memory-bandwidth cost of eigenvalue decompositions

The symmetric matrix eigenvalue problem (nearly same as nonsymmetric SVD) provides a nice example of where memory-bandwidth requires extra consideration with respect to distributed memory bandwidth cost

- probably the last dense numerical linear algebra problem we study in this course
- given a symmetric matrix $A$, we would like to compute its eigenvalues
- stable algorithms work by first reducing $A$ to tridiagonal form, then using the MRRR algorithm
- the reduction to tridiagonal form dominates the cost
- needs to be done via two-sided orthogonalization to preserve eigenvalues $T = Q^T AQ$
Direct tridiagonalization

We can perform two-sided orthogonalization via Householder QR

- compute Householder vector to eliminate \( n - 2 \) lower entries of first column
- \( Q_1^T A = (I - 2uu^T)A \) does not affect top row, so we can perform \( Q_1^T AQ_1 \)
- applying \( Q_1^T \) from the left is independent across columns
- applying \( Q_1 \) from the right is independent across rows
- this means we need to compute \( Q_1^T AQ_1 \) fully, before we can compute the Householder vector of the next column
- for designing a 2D algorithm, we can keep \( A \) in place and broadcast the vectors, for \( O(n^2/\sqrt{P}) \) communication
- but if the matrix blocks do not fit in cache \( (n^2/P \geq H) \), we will have \( O(n^3/P) \) memory bandwidth cost (no reuse), rather than \( O(n^3/(P\sqrt{H})) \)
Full-to-band reduction

We can alleviate the problem by reducing to a banded matrix first

- compute rectangular QR of $n - b \times b$ lower left minor (submatrix)
- $Q_1^T A = (I - 2uu^T)A$ reduces first $b$ columns to bandwidth $2b$ and does not affect top $b$ rows, so we can perform $Q_1^T AQ_1$
- now we can perform the trailing matrix update by matrix multiplication with rectangular matrices of dimensions $(n - b) \times b$
- Q: what is the minimal $b$ we would want to pick to get $\sqrt{H}$ reuse of trailing matrix entries, and consequently $O(n^3/(P\sqrt{H}))$ memory bandwidth cost?
- A: $b = \sqrt{H}$
- it then remains to reduce the banded matrix to tridiagonal form, which can be done via bulge chasing [Lang 1993]
Symmetric band reduction (bulge chasing)
Ideal cache model

A more accurate model is to consider a cache line size $L$ in addition to the cache size $H$

- each memory-to-cache transfer has size $L$
- new unified metric: cache misses (number of cache lines transferred)
- the bandwidth cost is the number of cache misses multiplied by $L$
- the (old) latency cost (number of transfers) is disregarded
- assume ‘tall’ cache, $L \leq \sqrt{H}$ (more convenient, $H = \Omega(L^2)$)
- we can now consider different caching protocols
- an ideal cache model corresponds to the assumption that the protocol always makes the best decision
- this ideal cache model is in a sense equivalent to a manually orchestrated cache protocol
- arbitrary manual orchestration can be achieved with an LRU (least-recently-used protocol)
Matrix transposition in the ideal cache model

Matrix multiplication bandwidth cost with a tall cache is not affected by $L$
- if we read square blocks into cache they have dimension $\Theta(L)$
- if we compute outer products, just need to transpose $B$ initially
- $n \times n$ matrix transposition becomes non-trivial
  - when $L = 1$ (original model), there is no notion of how a matrix is laid out in memory
  - for general $L$, we should read $\sqrt{H} \times \sqrt{H}$ blocks into cache, transpose them, then write them to memory to get linear bandwidth cost $O(n^2)$
- matrix transposition is a very useful subroutine when we need to ensure contiguous access to cache lines
Cache obliviousness

Introduced by Frigo, Leiserson, Prokop, Ramachadran (original paper worth reading)

- basic idea: algorithms should not be parameterized by architectural parameters
- good ideas in computer science are most often good abstractions
- designing an algorithm obliviously of cache size makes it portable and efficient for all levels of a cache hierarchy
- cache oblivious algorithms are stated without explicit control of data movement
- their communication cost is derived by assuming an ideal cache model
- ideal caches can be simulated by an LRU cache protocol for most (regular) algorithms
Cache oblivious matrix transposition

Given $m \times n$ matrix $A$, compute $B = A^T$

- if $m \leq n$ subdivide $A = [A_1 \ A_2]$ and $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$ and compute recursively, $B_1 = A_1^T$, $B_2 = A_2^T$

- if $m > n$ subdivide $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$ and $B = [B_1 \ B_2]$ and compute recursively, $B_1 = A_1^T$, $B_2 = A_2^T$

obtains linear bandwidth cost $T(mn) = 2T(mn/2)$, $T(1) = O(1)$, so $T(mn) = O(mn)$
Cache oblivious matrix multiplication

Given $m \times k$ matrix $A$ and $k \times n$ matrix $B$, compute $m \times n$ matrix $C = AB$

- if $k \geq m$ and $k \geq m$ subdivide $A = \begin{bmatrix} A_1 & A_2 \end{bmatrix}$ and $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$ and compute recursively, $\bar{C} = A_1B_1$, $\hat{C} = A_2B_2$, then $C = \bar{C} + \hat{C}$
- if $n > k$ and $n \geq m$ subdivide $C = \begin{bmatrix} C_1 & C_2 \end{bmatrix}$ and $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$ and compute recursively, $C_1 = AB_1$, $C_2 = AB_2$
- if $m > k$ and $m > n$ subdivide $C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix}$ and $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$ and compute recursively, $C_1 = A_1B$, $C_2 = A_2B$
Short pause
**DFT matrix**

*These notes are based on James Demmel's book, “Applied Numerical Linear Algebra”*

For any $n$, let $\omega_n = e^{-2\pi i / n}$, so $\omega_n^{n/2} = -1$ and $\omega_n^n = 1$, a DFT matrix of dimension $n$ is given by

$$D_n(j, k) = \omega_n^{jk}$$

for example

$$D_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}$$
DFT matrix

The matrix $A = \frac{1}{\sqrt{n}} D_n$ is symmetric and unitary $A = A^T = A^*$, $AA^{-1} = I$

$D_n^{-1}$ has the form $D_n^{-1}(j, k) = (1/n) \omega^{-jk}$, now $X = D_n D_n^{-1}$ has the form

$$X(j, k) = (1/n) \sum_{l=0}^{n-1} \omega^j_n \omega^{-lk}_n = (1/n) \sum_{l=0}^{n-1} \omega^{l(j-k)}_n$$

Clearly $X(j, j) = 1$, while $X(j, j + t) = (1/n) \sum_{l=0}^{n-1} (\omega^t_n)^l$ is a geometric sum for $t \neq 0$, so

$$X(j, j + t) = (1/n) \frac{1 - \omega^{nt}}{1 - \omega^t} = 0 \quad \text{since} \quad 1 - \omega^{nt} = 1 - (\omega^n)^t = 1 - 1^t = 0$$
Convolution

A convolution takes as input vectors $a$ and $b$ and computes vector $c$

$$\forall k \in [0, n - 1] \quad c(k) = \sum_{j=0}^{k} a(j)b(k - j)$$

- given coefficients of two polynomials of degree $n/2$ stored in $a$ and $b$, the convolution computes the coefficients $c$ of the product of the two polynomials
- naive evaluation costs $O(n^2)$ operations
- the convolution can also be interpreted as matrix-vector multiplication with a triangular Toeplitz matrix

$$[c(0) \ c(1) \ c(2) \ c(3)] = [a(0) \ a(1) \ a(2) \ a(3)] \cdot \begin{bmatrix} b(0) & b(1) & b(2) & b(3) \\ 0 & b(0) & b(1) & b(2) \\ 0 & 0 & b(0) & b(1) \\ 0 & 0 & 0 & b(0) \end{bmatrix}$$
Convolution via DFT

We can compute

$$\forall k \in [0, n - 1] \quad c(k) = \sum_{j=0}^{k} a(j)b(k - j)$$

via $c = D_{n}^{-1}[(D_{n}a) \odot (D_{n}b)]$ where $\odot$ is an elementwise product

$$z = v \odot w \rightarrow z(i) = v(i) \cdot w(i)$$

- we can find some intuition for this by thinking back to polynomial multiplication
- the DFT $D_{n}a$ evaluates a polynomial $f(x)$ at $x = \omega^{j}$ for $j \in [0, n - 1]$
- the elementwise product computes the values of the polynomial product at these points
- the inverse DFT $D_{n}^{-1}$ interpolates back from the points to get the coefficients of the polynomial product
Convolution via DFT

The polynomial interpretation is abstract, let's see what happens algebraically.

- First, let's write out the full expression in indexed form.

\[
c(k) = \sum_s D_n^{-1}(k, s) \left( \sum_j D_n(s, j) a(j) \right) \left( \sum_t D_n(s, t) b(t) \right)
\]

\[
= \sum_s \omega_n^{-ks} \left( \sum_j \omega_n^{sj} a(j) \right) \left( \sum_t \omega_n^{st} b(t) \right)
\]

- Now, let's rearrange the order of the summations to see what happens to every product of \( a \) and \( b \).

\[
c(k) = \sum_s \sum_j \sum_t \omega_n^{-ks} \omega_n^{sj} \omega_n^{st} a(j) b(t)
\]

\[
= \sum_s \sum_j \sum_t \omega_n^{(j+t-k)s} a(j) b(t)
\]

- We can observe that when \( j + t - k = 0 \) the products \( \omega_n^{(s+t-j)k} = 1 \), so the terms \( a(j)b(k-j) \) survive!

- For any \( u = j + t - k \neq 0 \), we observe \( \sum_s (\omega_n^u)^s = 0 \), as for \( D_n D_n^{-1} \)