Caches

Making them work for you
Optimizations for the Memory Hierarchy

❖ Write code that has locality!
  ▪ **Spatial**: access data contiguously
  ▪ **Temporal**: make sure access to the same data is not too far apart in time

❖ How can you achieve locality?
  ▪ Adjust memory accesses in *code* (software) to improve miss rate (MR)
    • Requires knowledge of *both* how caches work as well as your system’s parameters
  ▪ Proper choice of algorithm
  ▪ Loop transformations
Example: Matrix Multiplication

\[ C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \]
Matrices in Memory

- How do cache blocks fit into this scheme?
  - Row major matrix in memory:

COLUMN of matrix (blue) is spread among cache blocks shown in red
Naïve Matrix Multiply

```c
# move along rows of A
for (i = 0; i < n; i++)
    # move along columns of B
    for (j = 0; j < n; j++)
        # EACH k loop reads row of A, col of B
        # Also read & write c(i,j) n times
        for (k = 0; k < n; k++)
            c[i*n+j] += a[i*n+k] * b[k*n+j];
```

\[ C(i,j) = C(i,j) + A(i,:) \times B(:,j) \]
Cache Miss Analysis (Naïve)

- Scenario Parameters:
  - Square matrix \((n \times n)\), elements are doubles
  - Cache block size \(K = 64\) \(B = 8\) doubles
  - Cache size \(C \ll n\) (much smaller than \(n\))

- Each iteration:
  \[
  \frac{n}{8} + n = \frac{9n}{8} \text{ misses}
  \]

Ignoring matrix \(C\)
Cache Miss Analysis (Naïve)

❖ Scenario Parameters:
   - Square matrix \((n \times n)\), elements are doubles
   - Cache block size \(K = 64\) \(B = 8\) doubles
   - Cache size \(C \ll n\) (much smaller than \(n\))

❖ Each iteration:
   - \(\frac{n}{8} + n = \frac{9n}{8}\) misses
   - Afterwards in cache: (schematic)
Cache Miss Analysis (Naïve)

- Scenario Parameters:
  - Square matrix \((n \times n)\), elements are doubles
  - Cache block size \(K = 64\) B = 8 doubles
  - Cache size \(C \ll n\) (much smaller than \(n\))

- Each iteration:
  \[
  \frac{n}{8} + n = \frac{9n}{8} \text{ misses}
  \]

- Total misses:
  \[
  \frac{9n}{8} \times n^2 = \frac{9}{8} n^3
  \]

Once per element
Linear Algebra to the Rescue (1)

❖ Can get the same result of a matrix multiplication by splitting the matrices into smaller submatrices (matrix “blocks”)

❖ For example, multiply two 4×4 matrices:

\[
A = \begin{bmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix} = \begin{bmatrix}
  A_{11} & A_{12} \\
  A_{21} & A_{22}
\end{bmatrix}, \text{ with } B \text{ defined similarly.}
\]

\[
AB = \begin{bmatrix}
  (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\
  (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22})
\end{bmatrix}
\]

This is extra (non-testable) material
Linear Algebra to the Rescue (2)

Matrices of size $n \times n$, split into 4 blocks of size $r$ ($n=4r$)

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k}B_{k2}$$

- Multiplication operates on small “block” matrices
  - Choose size so that they fit in the cache!
  - This technique called “cache blocking”
**Blocked Matrix Multiply**

- Blocked version of the naïve algorithm:

```c
// move by r x r BLOCKS now
for (i = 0; i < n; i += r)
    for (j = 0; j < n; j += r)
        for (k = 0; k < n; k += r)
            # block matrix multiplication
            for (ib = i; ib < i+r; ib++)
                for (jb = j; jb < j+r; jb++)
                    for (kb = k; kb < k+r; kb++)
                        c[ib*n+jb] += a[ib*n+kb]*b[kb*n+jb];
```

- $r =$ block matrix size (assume $r$ divides $n$ evenly)
Cache Miss Analysis (Blocked)

- **Scenario Parameters:**
  - Cache block size $K = 64$ B = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks $(r \times r)$ fit into cache: $3r^2 < C$

- **Each block iteration:**
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$

- Ignoring matrix $C$
Cache Miss Analysis (Blocked)

- Scenario Parameters:
  - Cache block size $K = 64$ B = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks $(r \times r)$ fit into cache: $3r^2 < C$

  $r^2$ elements per block, 8 per cache block

- Each block iteration:
  - $r^2 / 8$ misses per block
  - $2n/r \times r^2 / 8 = nr/4$

  $n/r$ blocks in row and column

- Afterwards in cache (schematic)
Cache Miss Analysis (Blocked)

❖ Scenario Parameters:
  - Cache block size $K = 64$ B = 8 doubles
  - Cache size $C \ll n$ (much smaller than $n$)
  - Three blocks $\square (r \times r)$ fit into cache: $3r^2 < C$

$r^2$ elements per block, 8 per cache block

❖ Each block iteration:
  - $r^2/8$ misses per block
  - $2n/r \times r^2/8 = nr/4$

$n/r$ blocks in row and column

❖ Total misses: $nr/4 \times (n/r)^2 = n^3/(4r)$
Cache-Friendly Code

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique

- All systems favor “cache-friendly code”
  - Getting absolute optimum performance is very platform specific
    - Cache size, cache block size, associativity, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
    - Focus on inner loop code