Locality of Reference

- Application data space is often too large to all fit into cache
- Key is locality of reference
- Types of locality
  - Register locality
  - Cache-temporal locality
  - Cache-spatial locality
- Attempt (in order of preference) to:
  - Keep all data in registers
  - Order algorithms to do all ops on a data element before it is overwritten in cache
  - Order algorithms such that successive ops are on physically contiguous data

Data Locality

- Locality of Reference
- Types of locality
  - Register locality
  - Cache-temporal locality
  - Cache-spatial locality
- Attempt (in order of preference) to:
  - Keep all data in registers
  - Order algorithms to do all ops on a data element before it is overwritten in cache
  - Order algorithms such that successive ops are on physically contiguous data

Single Processor Performance Enhancement

- Two fundamental issues
  - Adequate op-level parallelism (to keep superscalar, pipelined processor units busy)
  - Minimize memory access costs (locality in cache)
- Three useful techniques to improve locality of reference (loop transformations)
  - Loop permutation
  - Loop blocking (tiling)
  - Loop unrolling

Access Stride and Spatial Locality

- Access stride: separation between successively accessed memory locations
  - Unit access stride maximizes spatial locality (only one miss per cache line)
- 2-D arrays have different linearized representations in C and FORTRAN
  - FORTRAN’s column-major representation favors column-wise access of data
  - C’s representation favors row-wise access of data

Matrix-Vector Multiply (MVM)

```plaintext
do i = 1, N
  do j = 1, N
    y(i) = y(i) + A(i,j) * x(j)
  end do
end do
```

- Total no. of references = 4N^2
- Two questions:
  - Can we predict the miss ratio of different variations of this program for different cache models?
  - What transformations can we do to improve performance?

Cache Model

- Fully associative cache (so no conflict misses)
- LRU replacement algorithm (ensures temporal locality)
- Cache size
  - Large cache model: no capacity misses
  - Small cache model: miss depending on problem size
- Cache block size
  - 1 floating-point number
  - b floating-point numbers
MVM: Dot-Product Form (Scenario 1)
- Loop order: i-j
- Cache model
  - Small cache: assume cache can hold fewer than (2N+1) numbers
  - Cache block size: 1 floating-point number
- Cache misses
  - Matrix A: N² cold misses (cold misses)
  - Vector x: N cold misses + N(N-1) capacity misses
  - Vector y: N cold misses
- Miss ratio = (2N² + N)/4N² → 0.5

MVM: Dot-Product Form (Scenario 2)
- Cache model
  - Large cache: cache can hold more than (2N+1) numbers
  - Cache block size: 1 floating-point number
- Cache misses
  - Matrix A: N² cold misses
  - Vector x: N cold misses
  - Vector y: N cold misses
- Miss ratio = (N² + 2N)/4N² → 0.25

MVM: Dot-Product Form (Scenario 3)
- Cache block size: b floating-point numbers
- Matrix access order: row-major access (C)
- Cache misses (small cache)
  - Matrix A: N²/b cold misses
  - Vector x: N/b cold misses
  - Vector y: N/b cold misses
  - Miss ratio = (1/2 + 1/4N)(1/b) → 1/2b
- Cache misses (large cache)
  - Matrix A: N²/b cold misses
  - Vector x: N/b cold misses
  - Vector y: N/b cold misses
  - Miss ratio = (1/4 + 1/2N)(1/b) → 1/4b

MVM: SAXPY Form (Scenario 4)
- Cache block size: b floating-point numbers
- Matrix access order: row-major access (C)
- Cache misses (small cache)
  - Matrix A: N² cold misses
  - Vector x: N/b cold misses
  - Vector y: N/b cold misses
  - Miss ratio = 0.25(1 + 1/b) + 1/4Nb → 0.25(1 + 1/b)
- Cache misses (large cache)
  - Matrix A: N² cold misses
  - Vector x: N/b cold misses
  - Vector y: N/b cold misses
  - Miss ratio = 1/4 + 1/2Nb → 1/4

MVM: SAXPY Form
- do j = 1, N
  - do i = 1, N
    - y(i) = y(i) + A(i,j)*x(j)
  - end do
- Total no. of references = 4N²

Loop Blocking (Tiling)
- Loop blocking enhances temporal locality by ensuring that data remain in cache until all operations are performed on them
- Concept
  - Divide or tile the loop computation into chunks or blocks that fit into cache
  - Perform all operations on block before moving to the next block
- Procedure
  - Add outer loop(s) to tile inner loop computation
  - Select size of the block so that you have a large cache model (no capacity misses, temporal locality maximized)
MVM: Blocked

\[
\begin{align*}
\text{do } & bi = 1, N/B \\
\text{do } & bj = 1, N/B \\
\text{do } & i = (bi-1)*B+1, bi*B \\
\text{do } & j = (bj-1)*B+1, bj*B \\
y(i) & = y(i) + A(i, j)*x(j)
\end{align*}
\]

MVM: Blocked (Scenario 5)

- Cache size greater than \(2B + 1\) floating-point numbers
- Cache block size: \(b\) floating-point numbers
- Matrix access order: row-major access (C)
- Cache misses per block
  - Matrix A: \(B/b\) cold misses
  - Vector x: \(B/b\)
  - Vector y: \(B/b\)
  - Miss ratio: \((1/4 + 1/2B)*(1/b) \rightarrow 1/4b\)
- Lesson: proper loop blocking eliminates capacity misses. Performance essentially becomes independent of cache size (as long as cache size is > \(2B +1\) fp numbers)

Access Stride and Spatial Locality

- Access stride: separation between successively accessed memory locations
  - Unit access stride maximizes spatial locality (only one miss per cache line)
- 2-D arrays have different linearized representations in C and FORTRAN
  - FORTRAN’s column-major representation favors column-wise access of data
  - C’s representation favors row-wise access of data

Access Stride Analysis of MVM

<table>
<thead>
<tr>
<th></th>
<th>Access stride for arrays</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dot-product form</td>
<td></td>
</tr>
<tr>
<td>do i =</td>
<td>1, N</td>
<td>C</td>
</tr>
<tr>
<td>do j =</td>
<td>1, N</td>
<td>N</td>
</tr>
<tr>
<td>y(i)</td>
<td>= y(i) + A(i, j)*x(j)</td>
<td></td>
</tr>
<tr>
<td>end do</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SAXPY form</td>
<td></td>
</tr>
<tr>
<td>do j =</td>
<td>1, N</td>
<td>C</td>
</tr>
<tr>
<td>do i =</td>
<td>1, N</td>
<td>N</td>
</tr>
<tr>
<td>y(i)</td>
<td>= y(i) + A(i, j)*x(j)</td>
<td></td>
</tr>
<tr>
<td>end do</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Matrix-Matrix Multiply (MMM)

for \(i = 0; i < N1; i++\) {
    for \(j = 0; j < N2; j++\) {
        for \(k = 0; k < N3; k++\) {
            \[C[i][j] = C[i][j] + A[k][i]*B[k][j];\]
        }
    }
}

Access strides (C compiler: row-major) + Performance (MFLOPS)

<table>
<thead>
<tr>
<th></th>
<th>ijk</th>
<th>jik</th>
<th>kij</th>
<th>jki</th>
<th>jki</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(i,j)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>A(k,i)</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>B(k,j)</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P4 1.6</td>
<td>48</td>
<td>45</td>
<td>77</td>
<td>77</td>
<td>8</td>
</tr>
<tr>
<td>P3 800</td>
<td>45</td>
<td>48</td>
<td>55</td>
<td>37</td>
<td>10</td>
</tr>
<tr>
<td>suraj</td>
<td>3.6</td>
<td>3.6</td>
<td>4.4</td>
<td>4.3</td>
<td>4.0</td>
</tr>
</tbody>
</table>

Loop Permutation

- Changing the order of nested loops can improve spatial locality
  - Choose the loop ordering that minimizes miss ratio, or
  - Choose the loop ordering that minimizes array access strides
- Loop permutation issues
  - Validity: Loop reordering should not change the meaning of the code
  - Performance: Does the reordering improve performance?
  - Array bounds: What should be the new array bounds?
  - Procedure: Is there a mechanical way to perform loop permutation?

These are of primary concern in compiler design. We will discuss some of these issues when we cover dependences, from an application software developer’s perspective.
Loop Unrolling

- Reduce number of iterations of loop but add statement(s) to loop body to do work of missing iterations
- Increase amount of operation-level parallelism in loop body
- Outer-loop unrolling changes the order of access of memory elements and could reduce number of memory accesses and cache misses

```
do i = 1, 2n
do j = 1, m
    loop-body(i,j)
end do
```

```
do i = 1, 2n, 2
    do j = 1, m
        loop-body(i,j)
        loop-body(i+1, j)
    end do
```

MVM: Dot-Product 4-Outer Unrolled Form
/* Assume \( n \) is a multiple of 4 */
for \( i = 0; i < n; i+=4 \) {
    for \( j = 0; j < n; j+=4 \) {
        \( y[i] = y[i] + A[i][j]*x[j]; \)
        \( y[i+1] = y[i+1] + A[i+1][j]*x[j]; \)
        \( y[i+2] = y[i+2] + A[i+2][j]*x[j]; \)
        \( y[i+3] = y[i+3] + A[i+3][j]*x[j]; \)
    }
}

Performance (MFLOPS)
<table>
<thead>
<tr>
<th></th>
<th>32 x 32</th>
<th>1024 x 1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4 1.6</td>
<td>23.0</td>
<td>8.5</td>
</tr>
<tr>
<td>P3 800</td>
<td>73.1</td>
<td>38.0</td>
</tr>
<tr>
<td>suraj</td>
<td>6.6</td>
<td>39.2</td>
</tr>
</tbody>
</table>

MVM: SAXPY 4-Outer Unrolled Form
/* Assume \( n \) is a multiple of 4 */
for \( i = 0; i < n; i+=4 \) {
    for \( j = 0; j < n; j+=4 \) {
        \( y[i] = y[i] + A[i][j]*x[j]; \)
        \( + A[i][j+1]*x[j+1] \)
        \( + A[i][j+2]*x[j+2] \)
        \( + A[i][j+3]*x[j+3]; \)
    }
}

Performance (MFLOPS)
<table>
<thead>
<tr>
<th></th>
<th>32 x 32</th>
<th>1024 x 1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4 1.6</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>P3 800</td>
<td>20.5</td>
<td>9.8</td>
</tr>
<tr>
<td>suraj</td>
<td>3.5</td>
<td>8.6</td>
</tr>
</tbody>
</table>

Summary

- Loop permutation
  - Enhances spatial locality
  - C and FORTRAN compilers have different access patterns for matrices resulting in different performances
  - Performance increases when inner loop index has the minimum access stride
- Loop blocking
  - Enhances temporal locality
  - Ensure that \( 2B \) is less than cache size
- Loop unrolling
  - Enhances superscalar, pipelined operations