Introduction to High Performance Computing for Scientists and Engineers

Chapter 3: Data Access Optimization

Slides by Mike Heath
Performance & Bandwidth Trends

- Gap between processor performance and memory bandwidth is growing, especially for multicore processors.
Balance between data access and processing speeds of machine is expressed by ratio $B_m = \frac{W}{F}$, where $W$ and $F$ are measured in words and floating-point operations, respectively, per unit time.

<table>
<thead>
<tr>
<th>data path</th>
<th>balance [W/F]</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache</td>
<td>0.5–1.0</td>
</tr>
<tr>
<td>machine (memory)</td>
<td>0.03–0.5</td>
</tr>
<tr>
<td>interconnect (high speed)</td>
<td>0.001–0.02</td>
</tr>
<tr>
<td>interconnect (GBit ethernet)</td>
<td>0.0001–0.0007</td>
</tr>
<tr>
<td>disk (or disk subsystem)</td>
<td>0.0001–0.01</td>
</tr>
</tbody>
</table>

Similarly, balance between loads/stores and flops executed by program is given by $B_c = \frac{W}{F}$, where $W$ and $F$ are measured in words and floating-point operations, respectively.
Performance Model

• Ratio of machine balance to code balance gives crude performance model for expected fraction of peak performance, \( \min(1, \frac{B_m}{B_c}) \)

• For example, vector triad executes three loads, one store, and two flops per iteration, so \( B_c = \frac{W}{F} = \frac{3+1}{2} = 2 \), and thus expected fraction of peak on processor with \( B_m = 0.1 \) is 0.05, or 5%

\[
\text{do } i=1,N \\
   A(i) = B(i) + C(i) \ast D(i) \\
\text{enddo}
\]

• Reciprocal of code balance, \( \frac{1}{B_c} = \frac{F}{W} \), called computational intensity of code, provides measure of potential data reuse
STREAM Benchmarks

- STREAM benchmarks are simple loop kernels commonly used to characterize memory performance

<table>
<thead>
<tr>
<th>type</th>
<th>kernel</th>
<th>DP words</th>
<th>flops</th>
<th>$B_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY</td>
<td>$A(\cdot) = B(\cdot)$</td>
<td>2 (3)</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>SCALE</td>
<td>$A(\cdot) = s \times B(\cdot)$</td>
<td>2 (3)</td>
<td>1</td>
<td>2.0 (3.0)</td>
</tr>
<tr>
<td>ADD</td>
<td>$A(\cdot) = B(\cdot) + C(\cdot)$</td>
<td>3 (4)</td>
<td>1</td>
<td>3.0 (4.0)</td>
</tr>
<tr>
<td>TRIAD</td>
<td>$A(\cdot) = B(\cdot) + s \times C(\cdot)$</td>
<td>3 (4)</td>
<td>2</td>
<td>1.5 (2.0)</td>
</tr>
</tbody>
</table>

- Unfortunately, even these simple loops often fail to attain substantial fraction of peak performance

- STREAM benchmarks generally provide upper bound on performance expected of more realistic codes
Multidimensional arrays, especially two-dimensional matrices, are extremely common in scientific codes.

Machine memory layout is inherently one-dimensional, divided into cache lines.

Mapping of multidimensional arrays to one-dimensional memory, as well as order in which array entries are accessed, dramatically affect cache behavior of array-based programs.

For example, *strided* access to one-dimensional array (accessing every $k$th entry rather than consecutive entries) reduces spatial locality and effective utilization of memory bandwidth.

Different programming languages have different conventions for storing multidimensional arrays.
C and its variants store arrays in row major order, i.e., last subscript varies most rapidly.
Column Major Order

* Fortran stores arrays in column major order, i.e., first subscript varies most rapidly.
Strided Memory Access

Because of different array storage orders, similar codes in different languages may access memory with different strides.

Stride-N access

```plaintext
1  do i=1,N
2     do j=1,N
3      A(i,j) = i*j
4  enddo
5  enddo
```

Stride-1 access

```plaintext
for(i=0; i<N; ++i) {
    for(j=0; j<N; ++j) {
        a[i][j] = i*j;
    }
}
```

To optimize memory access, inner loop variable indexing multidimensional array should be chosen to ensure stride-one access (first index in Fortran, last index in C).
Case Study: Diffusion Equation

- Diffusion equation given by \( \frac{\partial \Phi}{\partial t} = \Delta \Phi \)

- Jacobi method for solving finite difference discretization

\[
\frac{\delta \Phi(x_i, y_i)}{\delta t} = \frac{\Phi(x_{i+1}, y_i) + \Phi(x_{i-1}, y_i) - 2\Phi(x_i, y_i)}{(\delta x)^2} \\
+ \frac{\Phi(x_i, y_{i-1}) + \Phi(x_i, y_{i+1}) - 2\Phi(x_i, y_i)}{(\delta y)^2}
\]

- Sweep through two-dimensional grid in some order, updating solution at each grid point by contributions from four neighboring grid points

- Requires two copies of solution array, as solution values cannot be overwritten until sweep is complete
Case Study: Diffusion Equation
Case Study: Diffusion Equation

```fortran
double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
integer :: t0,t1

! Depending on cache line size, problem dimensions, and order of
! traversal, neighboring points may still be in cache from previous
! access
```
Case Study: Diffusion Equation

- Performance graph shows decline in performance when problem size exceeds cache size and code becomes memory bound.
Case Study: Matrix Transpose

* Calculating transpose of dense matrix, $A = B^T$, involves no arithmetic operations, but illustrates performance issues in accessing memory

* Access to either $A$ or $B$ must be strided

1. do $i=1,N$
2.   do $j=1,N$
3.       $A(i,j) = B(j,i)$
4. enddo
5. enddo

* Strided write more expensive than strided read because of write allocate

* Moving index $i$ to inner loop changes access from strided writes to strided reads (“flipped”)
Case Study: Matrix Transpose

- If both matrices fit in cache \((2N^2 \leq C)\), code should run at full cache speed despite strided access.

- If matrices are too large to fit in cache, but one row or column fits in cache \((N L_c \leq C)\), then spatial locality may still allow performance at near full cache speed.

- If matrices are so large that one row or column does not fit in cache \((N L_c > C)\), then spatial locality is lost and performance drops.

- TLB (translation lookaside buffer) misses can also dramatically affect performance.

- TLB caches mapping between logical and physical memory pages.
Case Study: Matrix Transpose
Case Study: Matrix Transpose

- If dimension of matrix $N$ happens to match cache line size, then strided access can cause cache thrashing.
- Padding array can eliminate this effect.
Algorithm Classification

- Algorithms can be classified according to ratio of number of arithmetic operations to number of data items involved

- For example,
  
  - vector addition: $O(N)$ arithmetic operations, $O(N)$ data
  
  - matrix-vector multiplication: $O(N^2)$ operations, $O(N^2)$ data
  
  - matrix-matrix multiplication: $O(N^3)$ operations, $O(N^2)$ data

- Opportunities for reusing data already in cache are obviously greater when number of operations greatly exceeds number of data items
When number of operations and number of data items are both proportional to problem size, opportunities for data reuse are limited and performance is generally memory bound.

Although loops are not nested, multiple loops can potentially combined to reduce number of loads, as in *loop fusion*

```plaintext
1  do i=1,N
2    A(i) = B(i) + C(i)
3  enddo
4  do i=1,N
5    Z(i) = B(i) + E(i)
6  enddo

! optimized
! save a load for B(i)
A(i) = B(i) + C(i)
Z(i) = B(i) + E(i)
```

Compilers can often apply this optimization.
\( O(N^2) / O(N^2) \)

- This type of algorithm usually involves nested loops with two levels
- Consider code for matrix-vector multiplication

```
1  do i=1,N
2      tmp = C(i)
3      do j=1,N
4          tmp = tmp + A(j,i) * B(j)
5      enddo
6  C(i) = tmp
7  enddo
```

- Matrix A is loaded once, but vector B is loaded N times, once for each iteration of outer loop
- We can fuse N inner loops by *loop unrolling*, traversing outer loop with stride \( m \) and replicating inner loop \( m \) times
- This technique is called *unroll and jam*
Examples: Unroll and Jam

* Matrix-vector multiply

```plaintext
! remainder loop ignored
do i=1,N,m
  do j=1,N
    C(i) = C(i) + A(j,i) * B(j)
    C(i+1) = C(i+1) + A(j,i+1) * B(j)
    ! m times
  ...
  C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
  enddo
endo
```

* Matrix transpose

```plaintext
do j=1,N,m
  do i=1,N
    A(i,j) = B(j,i)
    A(i,j+1) = B(j+1,i)
  ...
  A(i,j+m-1) = B(j+m-1,i)
  enddo
endo
```
Example: Loop Blocking

- Loop blocking can achieve optimal cache line use
- Does not reduce loads or stores, but increases cache hit ratio
- Example: matrix transpose

```plaintext
  do jj=1,N,b
      jstart=jj; jend=jj+b-1
      do ii=1,N,b
          istart=ii; iend=ii+b-1
          do j=jstart,jend,m
              do i=istart,iend
                  a(i,j) = b(j,i)
                  a(i,j+1) = b(j+1,i)
                  ...
                  a(i,j+m-1) = b(j+m-1,i)
              enddo
          enddo
      enddo
  enddo
```
\(O(N^3)/O(N^2)\)

- When number of arithmetic operations exceeds number of data items by factor that grows with problem size, opportunities for reuse of data are greatly enhanced
- This type of algorithm usually involves nested loops with three levels, such as matrix-matrix multiplication
- Carefully chosen blocking and unrolling can often make code cache bound rather than memory bound
- Many vendors provide highly optimized libraries of routines for common operations of linear algebra involving dense vectors and matrices, such as BLAS (Basic Linear Algebra Subprograms), LAPACK, etc.