IN3200/IN4200: Chapter 3
Data access optimization
(Part 3)

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Overview

Goals of Chapter 3:

- be able to reason about data access and performance using machine/code balance analysis
- be able to recognize and apply data access optimizations
  - loop unrolling
  - loop unroll-and-jam
  - loop blocking
  - loop fusion
  - re-organization of data structure

Today's lecture

Three cases of code balance analysis and data access optimization:

- Dense matrix-vector multiply (repetition)
- Matrix transpose
- Sparse matrix-vector multiply

Mathematical definition of matrix-vector multiply

Square matrix A: N rows and N columns of numerical values
Vector B: N numerical values
Vector C: N numerical values

Mathematical definition of matrix-vector multiply: $\mathrm{C} = \mathrm{C} + \mathrm{A} * \mathrm{B}$ such that each value in vector C is calculated as

$$C_i = C_i + \sum_{0 \le j < N} A_{i,j} * B_j \qquad 0 \le i < N$$

Dense matrix-vector multiply (repetition)

Here, we consider the case of A being a "dense" matrix: all its $N \times N$ numerical values are nonzero.

Storage on a computer:

- Dense square matrix A as a 2D array, N rows and N columns, row-major storage (in C language)
- Vectors B and C each as a 1D array of length N

Each value $A_{i,j}$ is accessed as A[i][j]

Straightforward implementation & balance analysis

```
for (i=0; i<N; i++) {
  double tmp = C[i];
  for (j=0; j<N; j++)
    tmp = tmp + A[i][j]*B[j];
  C[i] = tmp;
}
```

- Total number of floating-point (FP) operations: $2N^2$
- Memory traffic: N^2 loads for 2D array `A`, N loads & N stores for 1D array `C`
- How many loads are associated with 1D array `B`?
  - Small cache $\rightarrow$ array `B` is loaded N times $\rightarrow$ N^2 memory loads
  - Large cache $\rightarrow$ array `B` is loaded only once $\rightarrow$ N memory loads

Code balance for the small-cache case:

$$\frac{N^2 + N^2 + 2N}{2N^2} = 1 + \frac{1}{N}$$

Illustration of array B being loaded N times

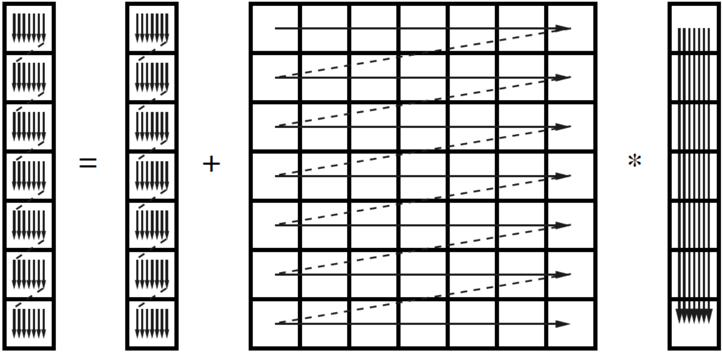

Figure 3.11: Unoptimized $N \times N$ dense matrix vector multiply. The RHS vector is loaded N times.

How to reduce memory traffic for small-cache case?

***m*-way unroll-and-jam:**

- Unroll the outer loop m times
- Fuse the m inner loops

```
for (i=0; i<N; i+=m) {
  for (j=0; j<N; j++) {
    C[i+0] += A[i+0][j]*B[j];
    C[i+1] += A[i+1][j]*B[j];
    // ...
    C[i+m-1] += A[i+m-1][j]*B[j];
  }
}
// remainder code in case (N%m)>0 ....
```

- m-fold reuse of each `B[j]` from register
- Number of memory loads for array `B`: N^2/m (for small-cache case)
- Size of m shouldn't be too large, to avoid too high *register pressure*

Illustration of the effect of unrolling

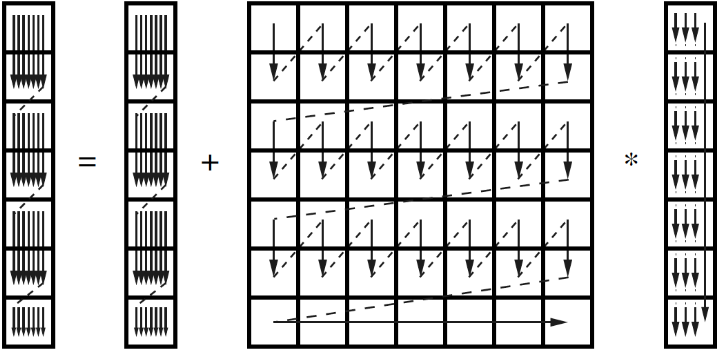

Figure 3.12: Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

Improved code balance

For the small-cache case, unroll-and-jam will result in the following improved code balance:

$$\frac{N^2 + \frac{N^2}{m} + 2N}{2N^2} = \frac{1}{2} + \frac{1}{2m} + \frac{1}{N}$$

Matrix transpose

The *transpose* of an N-by-N matrix B is another N-by-N matrix $A = B^T$ such that $A_{i,j} = B_{j,i}$.

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++)
    A[j][i] = B[i][j];
```

It is assumed that A and B are 2D arrays in row-major storage.

(Note: The matrix transpose example in the textbook (Section 3.4) is programmed in Fortran and assumes column-major storage!)

In the above code, values are loaded from `B` in the order

$$\mathtt{B[i][j]}, \quad \mathtt{B[i+1][j]}, \quad \mathtt{B[i+2][j]}, \quad \ldots$$

These large jumps in memory can lead to poor cache line utilization.

Loop unrolling applied to matrix transpose

m-way unroll-and-jam:

- Unroll the outer loop m times
- Fuse the m inner loops

```
for (j=0; j<N; j+=m) {
  for (i=0; i<N; i++) {
    A[j+0][i] = B[i][j+0];
    A[j+1][i] = B[i][j+1];
    // ...
    A[j+m-1][i] = B[i][j+m-1];
  }
}
```

Illustration of unrolled matrix transpose

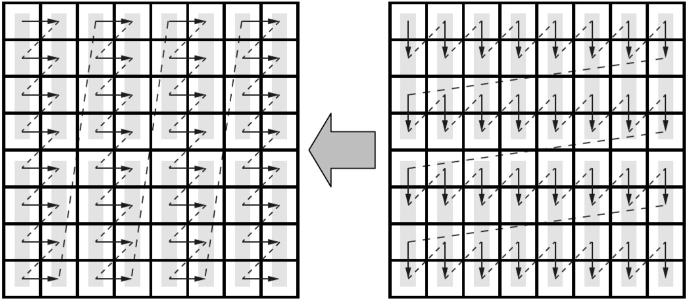

Figure 3.13: Two-way unrolled "flipped" matrix transpose (i.e., with strided load in the original version).

Loop blocking + unrolling applied to matrix transpose

```
for (jj=0; jj<N; jj+=b) {
  jstart = jj; jstop = jj+b-1;
  for (ii=0; ii<N; ii+=b) {
    istart = ii; istop = ii+b-1;

    for (j=jstart; j<=jstop; j+=m) {
      for (i=istart; i<=istop; i++) {
        A[j+0][i] = B[i][j+0];
        A[j+1][i] = B[i][j+1];
        // ...
        A[j+m-1][i] = B[i][j+m-1];
      }
    }
  }
}
```

Blocking improves locality for accessing `B`.

Loop blocking + unrolling applied to matrix transpose

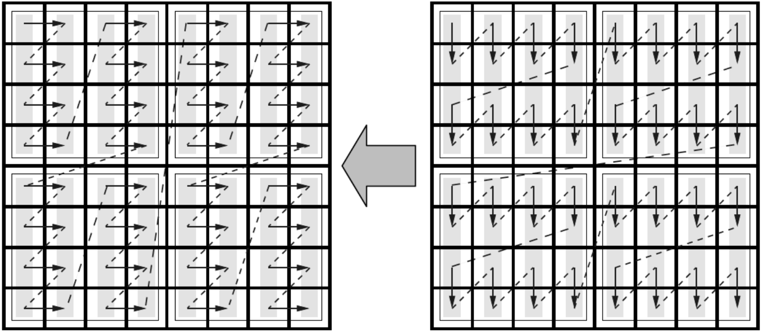

Figure 3.14: 4×4 blocked and two-way unrolled "flipped" matrix transpose.

B A

Sparse matrix

When most of the numerical values of matrix A are zero, it is called a *sparse* matrix.

- It will be a waste of float-point operations if we still use the straightforward (dense matrix-vector multiply) implementation
- It will also be a waste of storage if we store a sparse matrix as a 2D array

Illustration of sparse matrix-vector multiply

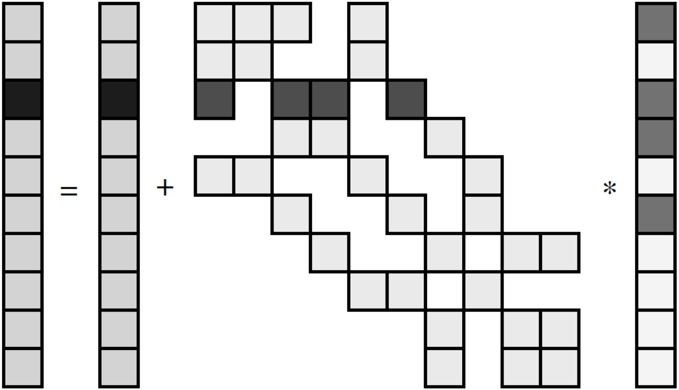

Figure 3.15: Sparse matrix-vector multiply. Dark elements visualize entries involved in updating a single LHS element. Unless the sparse matrix rows have no gaps between the first and last nonzero elements, some indirect addressing of the RHS vector is inevitable.

Basic idea for saving storage and computation

- Store only the nonzero values of A
  - 2D-array format can no longer be used
  - many sparse storage formats are possible
  - all sparse formats must somehow store the row i and column j of every *nonzero* value $A_{i,j}$
- Avoid multiplications with zero
  - If $N_{\mathrm{nz}}(\ll N^2)$ denotes the number of nonzero values in a sparse matrix A, then we only need $2N_{\mathrm{nz}}$ floating-point operations (instead of $2N^2$ FP) for a sparse matrix-vector multiply

Coordinate storage (COO) format

Three 1D arrays of length N_{nz}:

- `val`, stores all the nonzero values of the sparse matrix
- `row_idx`, stores the row positions of the nonzero values
- `col_idx`, stores the column positions of the nonzero values

```
for (int k=0; k<Nnz; k++)
  C[row_idx[k]] = C[row_idx[k]] + val[k]*B[col_idx[k]];
```

- Single loop over all nonzeros (of length N_{nz})
- Accesses to arrays `val`, `row_idx` and `col_idx` are with stride one (good spatial locality)
- Accesses to arrays `B` and `C` are indirect (via `row_idx` and `col_idx`) and can be completely irregular (spatial and temporal locality depends on row and column positions)

Code balance analysis of matrix-vector multiply with COO

```
for (int k=0; k<Nnz; k++)
  C[row_idx[k]] = C[row_idx[k]] + val[k]*B[col_idx[k]];
```

Assume that each entry in `row_idx` and `col_idx` is half a word.

Best-case scenario: entire `B` and `C` arrays are cached, needing in total only $2N$ loads and N stores:

$$\frac{N_{\mathrm{nz}}(1+0.5+0.5)+N+2N}{2N_{\mathrm{nz}}} = 1 + \frac{3}{2}\frac{N}{N_{\mathrm{nz}}}$$

Worst-case scenario: `B[col_idx[k]]` and `C[row_idx[k]]` need to be loaded from and stored to memory every single time, and only one value is used per cacheline:

$$\frac{N_{\mathrm{nz}}(1+0.5+0.5)+3N_{\mathrm{nz}}\frac{\text{cacheline size}}{\text{word size}}}{2N_{\mathrm{nz}}} = 1 + \frac{3}{2}\frac{\text{cacheline size}}{\text{word size}}$$

Compressed row storage (CRS) format

Idea: *grouping nonzeros by rows* and thus fewer accesses to array `C`

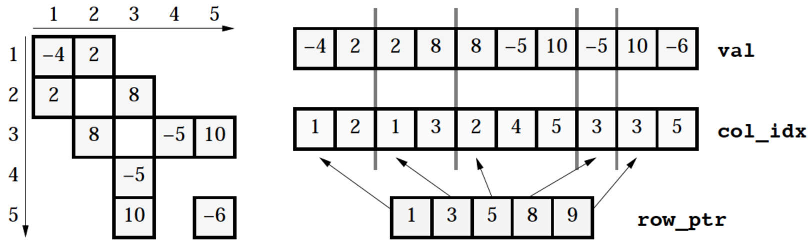

Figure 3.16: CRS sparse matrix storage format.

Note: In the above figure from Chapter 3, arrays `row_ptr` and `col_idx` contain 1-based indices due to Fortran programming.

Three arrays:

- 1D array `val`, of length N_{nz}, stores all the nonzero values of the sparse matrix
- 1D array `col_idx` , of length N_{nz}, records the original column positions of all the nonzero values
- 1D array `row_ptr`, of length $N+1$, contains the indices at which

Implementation of matrix-vector multiply using CRS format

```
for (i=0; i<N; i++) {
  double tmp = C[i];
  for (j=row_ptr[i]; j<row_ptr[i+1]; j++)
    tmp = tmp + val[j]*B[col_idx[j]];
  C[i] = tmp;
}
```

- There is a long outer loop (of length N)
- The inner loop can be very short
- Access to array `C` will be well optimized by compiler
- Access to array `val` is with stride one (perfect situation)
- Access to array `B` is indirect (via `col_idx`) and can be irregular

Code balance analysis of matrix-vector multiply with CRS

Assume that each entry in `row_ptr` and `col_idx` is half a word.

Best-case scenario: entire `B` array is cached, needing only N loads:

$$B_c = \frac{N_{\mathrm{nz}}(1+0.5) + 0.5N + N + 2N}{2N_{\mathrm{nz}}} = \frac{3}{4} + \frac{7}{4}\frac{N}{N_{\mathrm{nz}}}$$

Note: In Chapter 3.6.1, page 87, the estimated code balance $B_c = \frac{5}{4}$ is not optimal.

Exercise: What is the code balance for the *worst-case scenario*, where `B[col_idx[j]]` must be loaded from memory every single time, and only one value is used per cacheline?

Other ideas

- Continue using CRS format, but with suitable permutations (to reduce the actual memory traffic associated with array B)
- Use the JDS format (which targets vector processors) with further optimization (see Sections 3.6.1 & 3.6.2)