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

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

Content

Two cases of code balance analysis (and data access optimization):

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

Matrix-vector multiply

A 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: $\mathbf{C} = \mathbf{C} + \mathbf{A} * \mathbf{B}$
such that each value in vector **C** is calculated as

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

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 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

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

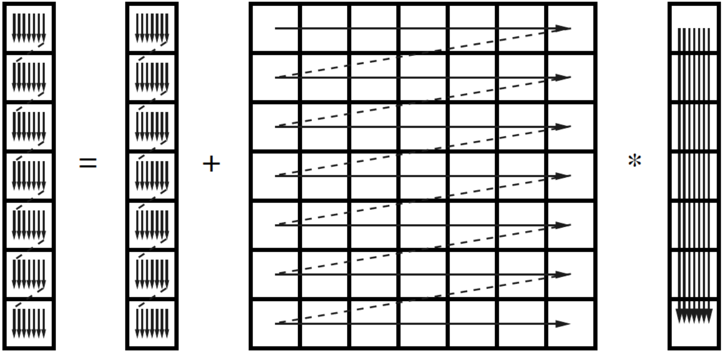

Figure 3.11: Unoptimized $N \times N$ dense matrix vector multiply. The RHS vector is loaded N times.

How to reduce memory traffic for small-cache case?

m-way unroll and jam:

```
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
- Total 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*

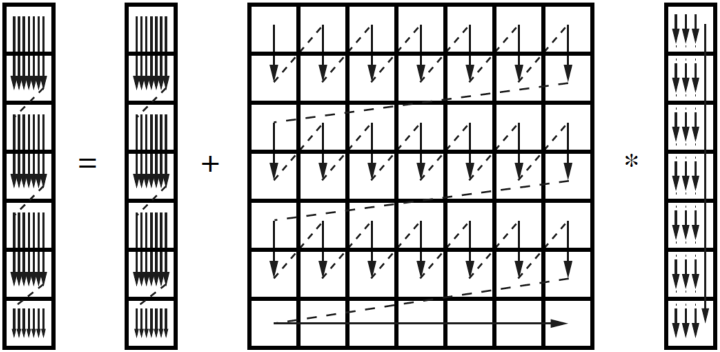

Figure 3.12: Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

Improved code balance

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

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

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 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

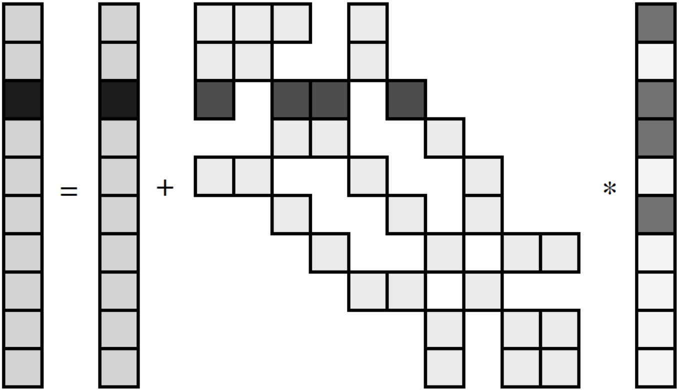

Figure 3.15: Sparse matrix-vector multiply. Dark elements visualize entries involved in updating a single LHS element. Unless the sparse matrix rows have no gaps between the first and last nonzero elements, some indirect addressing of the RHS vector is inevitable.

Basic idea for saving storage and computation

- Store only the nonzero values of **A**
  - 2D-array format can no longer be used, requires an efficient storage format
- 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

Compressed row storage (CRS) format

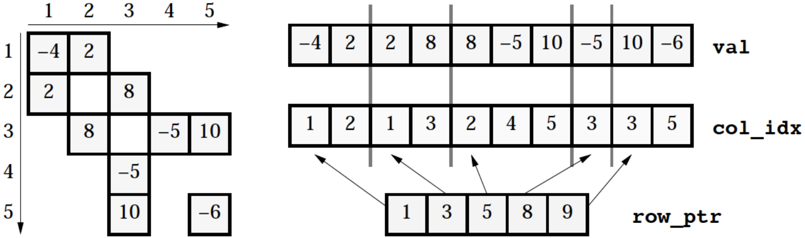

Figure 3.16: CRS sparse matrix storage format.

Three arrarys:

- 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 the all nonzero values
- 1D array `row_ptr`, of length $N+1$, contains the indices at which new rows start in array `val`

Implementation of matrix-vector multiply using CRS format

```
for (i=0; i<N; i++) {
  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
- Access to array `B` is indirect (via `col_idx`) and can be irregular

Code balance analysis of matrix-vector multiply with CRS

Best-case scenario (entire `B` array is cached, needing only N loads), each entry in `row_ptr` and `col_idx` is half a word:

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

Worst-case scenario (`B[col_idx[j]]` needs to be loaded from memory every single time, and only one value is used per cacheline):

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

Main ideas for improvement

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