Sparse Matrix-Vector Multiplication (Sparsity, Bebop)

María Jesús Garzarán
CS 498: Compiler Optimizations, Fall 2006

(based on the slides from Markus Pueschel, CMU)

University of Illinois at Urbana-Champaign

MMM versus MVM

- MMM
  - BLAS3
  - $O(n^2)$ data, $O(n^3)$ computation, implies $O(n)$ reuse per data.
- MVM: y=Ax
  - BLAS2
  - $O(n^2)$ data, $O(n^2)$ computation
  - Optimizations that are still useful:
    - Cache blocking?
    - Register blocking?
    - Unrolling?
    - Scalar replacent?
    - Add/mult interleaving, skewing?

MMM versus MVM: Performance

- Performance for 2000 x 2000 matrices
- Best code out of ATLAS, vendor lib. , Goto's assemby codes

Processor and compiler	Clock (MHz)	Data cache sizes	DGEMV (MFLOPS)	DGEMM (MFLOPS)
Sun UltraSPARC IIi Sun C v6.0	333	L1: 16 KB L2: 2 MB	58	425
Intel Pentium III Mobile (Coppermine) Intel C v6.0	800	L1: 16 KB L2: 256 KB	147	590
IBM Power4 IBM xlc v6	1300	L1: 64 KB L2: 1.5 MB L3: 32 MB	915	3500
Intel Itanium 2 Intel C v7.0	900	L1: 16 KB L2: 256 KB L3: 3 MB	1330	3500

Source: Eun-JIN Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: "An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Computing App. 18 (1), pp. 135 -158, 2004

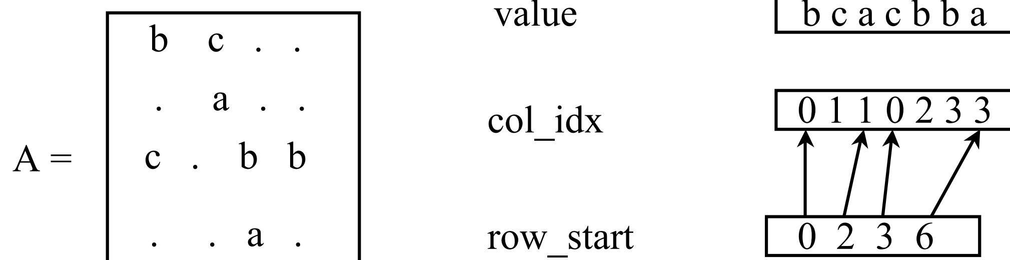
Sparse MVM

- y = Ax, A sparse but known
- Important routine in:
  - Finite element methods
  - PDE solvers
  - Physical/chemical simulations (e.g. fluid dynamics)
  - Linear programming
  - Scheduling
  - Signal processing (e.g. filters)
  - …
- In these applications, y = Ax is performed many times.
  - Justifies one-time tuning effort

Storage of Sparse Matrices

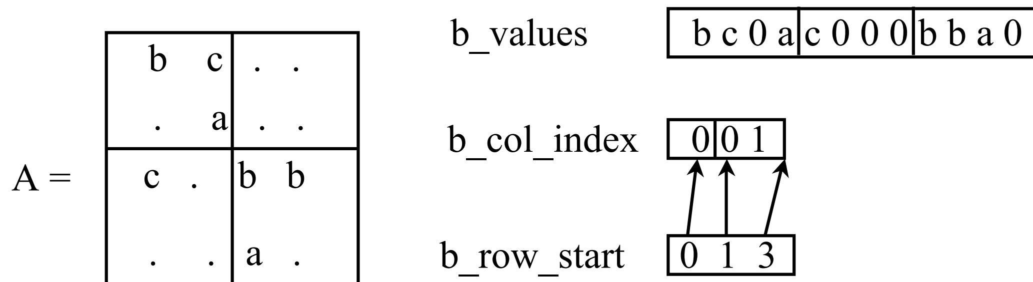
- Standard storage (as 2-D array) inefficient
  – Many zeros are stored
- Several sparse format are available.
  – Compressed Sparse Row Format
  – Blocked CSR (BCSR) Format

Compressed Sparse Row (CSR)



Blocked CSR (BCSR) Format

- Block A into r x c blocks
- Assume r=c=2



Direct Implementation y=Ax, A in CSR

```
void smvm_1x1( int m, const double* value, const int* col_icx, const inst*
    row_start, const double* x, double y)
{
    int i, jj;
    /* loop over rows */
    for (i=0; i<m; i++) {
        double y_i= y[I];
        /*loop over non-zero elements */
        for (jj = row_start[i]; jj<row_start[i+1];
            jj++, col_idx++, value++) {
             y_i += value[0]*x[col_idx[0]];
        }
        y[i] = y_i;
}
```

scalar replacement

indirect array accessing

Impact of Matrix-Sparsity on Performance

- Addresssing overhead (dense MVM vs. dense MVM in CSR):
  - ~2x slower (mflops, example only)
- Irregular structure
  - ~5x slower (mflops, example only) for "random" sparse matrices
- Fundamental difference between MVM and sparse MVM (SMVM)
  - Sparse MVM is input dependent (sparsity pattern of A)
  - Changing the order of computation (blocking) requires change of data structure (CSR)

Bebop/Sparsity: SMVM Optimizations

- Register blocking
- Cache blocking

Register Blocking

- Idea: divide SMVM y = Ax intro micro (dense) MVMs of matrix size r x x
  - Store A in r x c block CSR (r x c BCSR)

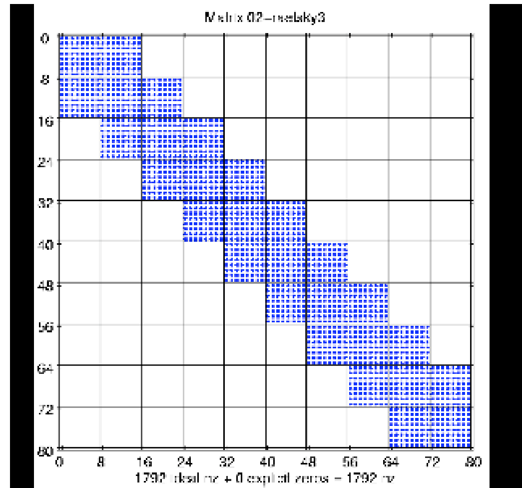
- Advantages:
  - Reuse of x and y (as for dense MVM)
  - Reduces index overhead
- Disadvantages:
  - Computational overhead (zeros added)
  - Storage overhead (for A)

Example: y=Ax in 2x2 BCSR

```
void smvm_2x2( int bm, const in *b_row_start, const int *b_col_idx, const double
    *b_value, const double *x, double *y)
    int I, jj;
    /* loop over block rows */
    for (i =0; i<bm; i++, y+=2){
            register double d0 = y[0];
            register double d1 = y[1];
    /*dense micro MVM */
            for (jj = b_row_start[i]; jj<b_row_start[i+1];
            j++, b_col_idx++, b_value += 2*2) {
                d0 += b_value[0] * x[b_col_idx[0]+0];
                d1 += b_value[2] * x[b_col_idx[0]+0];
                d0 += b_value[1] * x[b_col_idx[0]+1];
                d1 += b_value[3] * x[b_col_idx[0]+1];
            }
    y[0]=d0;
    y[1]=d1;
    }
```

Which Block Size (r x c) is Optimal?

- Example: ~20,000 x 20,000 matrix with perfect 8x8 block structure, 0,33% non-zero entries
- In this case, no overhead when blocked r x c, with r, c divides 8.

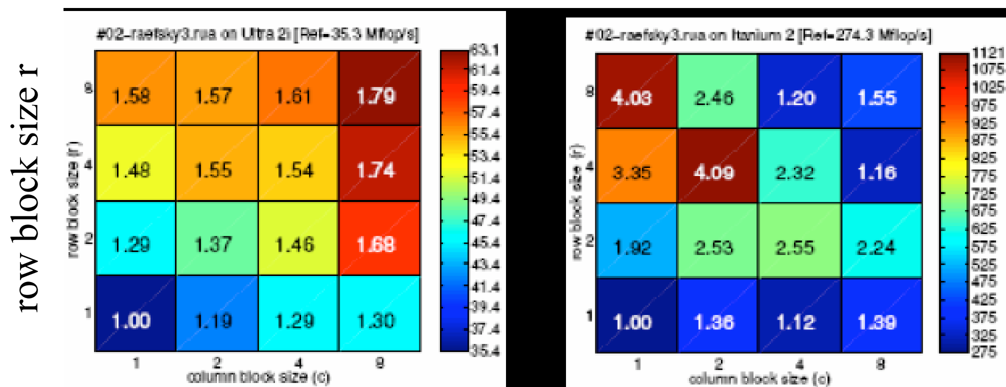


Source: Vuduc, LLNL

Speed-up through r x c Blocking

Ultra 2i

Itanium 2



col. Block size c

col. Block size c

- Machine dependent
- Hard to predict

Source: Eun-JIN Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: "An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Computing App. 18 (1), pp. 135 -158, 2004

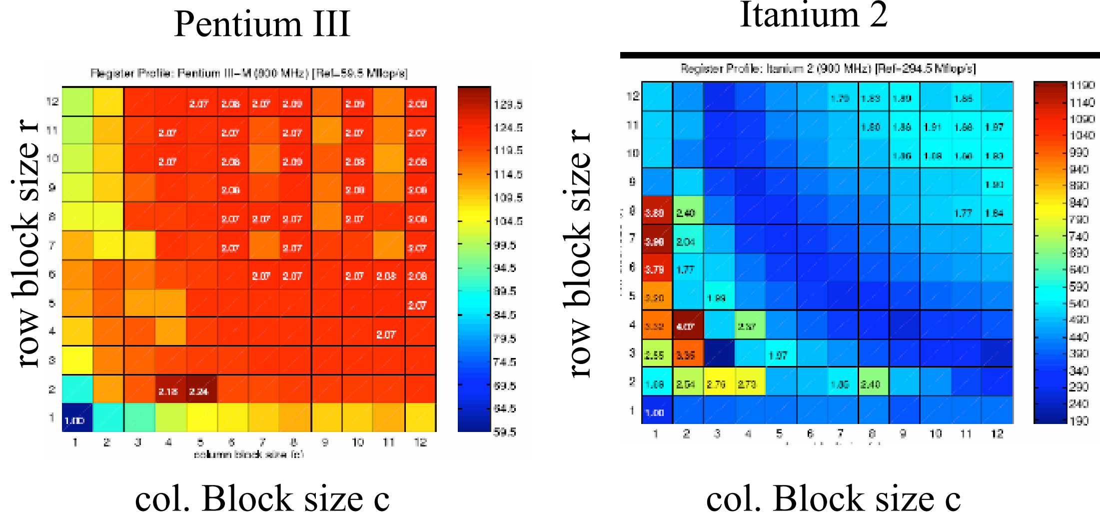
How to Find the Best Register Blocking for given A

- Best blocksize is hard to predict (see the previous slide)
- Searching over all r x c (within a range 1..12) is expensive
  - Conversion of A in CSR to BCSR roughly as expensive as 10 SMVMs
- Solution: Performance model for given A

Performance Model for given A

- Model for given A built from
  - Gain of blocking:
    $G_{r,c}$ = Performance r x c BCSR/performance CSR for dense MVM
    Machine depending, independent of matrix A
  - Computational overhead:
    $O_{r,c}$ = size of A in r x c BCSR/size of A in CSR
    machine independent, dependent of A
    computed by scanning only a fraction of the matrix
- Model: Performance gain from r x c blocking of A:
  $P_{r,c} = G_{r,c}/O_{r,c}$
- For given A, use this model to search over all r, c in {1..12}

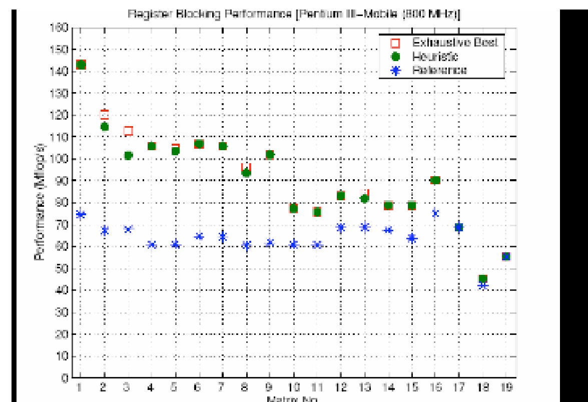
Gain from Blocking (Dense Matrix in BCSR)



Source: Eun-JIN Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: "An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Computing App. 18 (1), pp. 135 -158, 2004

Register Blocking: Experimental Results

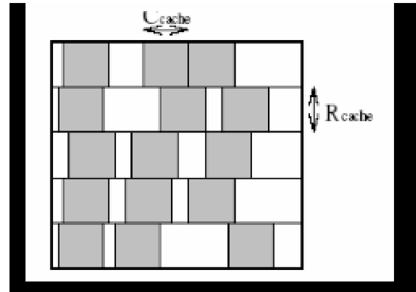
- Paper applies method to a large set of sparse matrices
- Performance gains between 1x (no gain) for very unstructured matrices and 4x



Source: Eun-JIN Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: "An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Computing App. 18 (1), pp. 135 -158, 2004

Cache Blocking

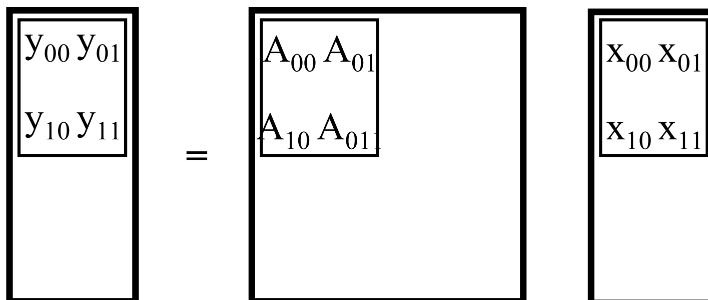
- Idea: divide sparse matrix into blocks of sparse matrices



- Experiments:
  - Requires very large matrices (x and y do not fit in cache)
  - Speedup up to 80% only for few matrices, with 1x1 BSCR

Source: Eun-JIN Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: "An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Computing App. 18 (1), pp. 135 -158, 2004

Multiple Vector Optimization



OPTION 1

(1) $y_{00} = A_{00}\, x_{00} + A_{01}\, x_{10}$
(2) $y_{10} = A_{10}\, x_{00} + A_{11}\, x_{10}$

…

(nz +1) $y_{01} = A_{00}\, x_{01} + A_{01}\, x_{11}$
(nz + 2) $y_{11} = A_{10}\, x_{001} + A_{11}\, x_{11}$

OPTION 2

(1) $y_{00} = A_{00}\, x_{00} + A_{01}\, x_{10}$
(2) $y_{10} = A_{10}\, x_{00} + A_{11}\, x_{10}$
(3) $y_{01} = A_{00}\, x_{01} + A_{01}\, x_{11}$
(4) $y_{11} = A_{10}\, x_{01} + A_{11}\, x_{11}$

Multiple Vector Optimization

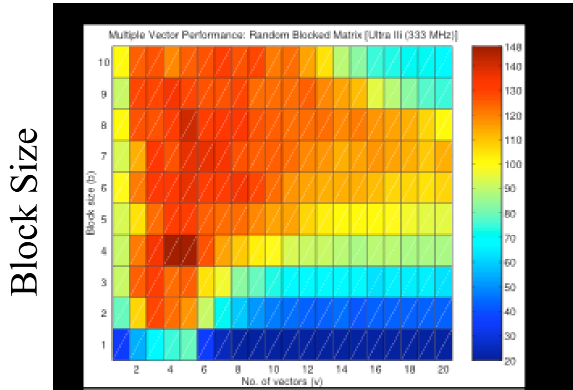
Ultra IIi

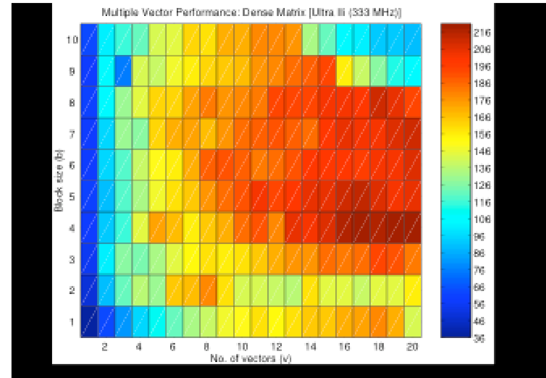
Random Blocked Matrix

Ultra IIi

Dense Matrix

Block Size





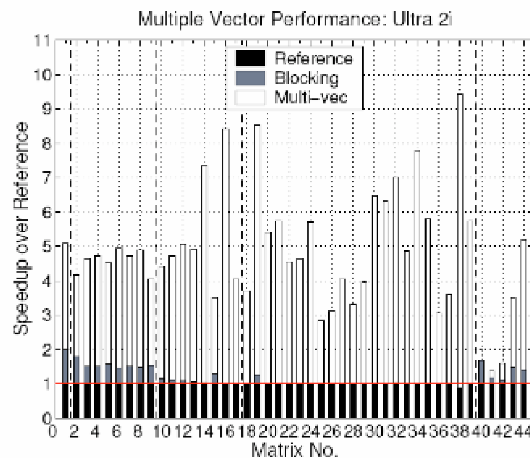
Number of Vectors

Number of Vectors

Source: Eun-JIN Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: "An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Computing App. 18 (1), pp. 135 -158, 2004

Multiple Vector Optimization

- Experiments: Up to 9x speedup for 9 vectors



Source: Eun-JIN Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: "An Optimization Framework for Sparse Matrix Kernels, Int'l Journal of High Performance Computing App. 18 (1), pp. 135 -158, 2004

Principles in Bebop/Sparsity

- Optimization for memory hierarchy = increasing locality
  - Blocking for registers (micro-MMMs) + change of data structure for A
  - Less important: blocking for cache
  - Optimizations are input dependent (on sparse structure of A)
- Fast basick blocks for small sizes (micro-MMM):
  - Loop unrolling (reduced loop overhead)
  - Some scalar replacement (enables better compiler optimizations)
- Search for the fastest over a relevant set of algorithm/implementation alternatives (=r,c)
  - Use of performance model (versus measuring runtime) to evaluate expected gain

red = different from ATLAS