Summary

- Algorithms and Data Structures need to take memory prefetch hardware into account
- This talk shows one example - Matrix-vector multiply
- As we'll show, the results can be dramatic
- Prefetch is designed to improve realized memory bandwidth. How important is that?

Stream Performance Estimate

- Easy estimate: 11 GB/s = 2 * 5.5 GB/Sec to L3, 5.5 GB/Sec to main memory
  - Minimum link speed is 5.5 GB/s each way, Stream adds both
  - Measured performance is 1.2 GB/s!
  - Why?
  - Time to move each cache line
- 5.5 GB/s ~ 8 bytes/cycle (memory bus bandwidth)
- ~60 cycles L2 miss (latency)
- 64 byte cache line = 8 cycles (bandwidth) + 60 cycles (latency) = 68 cycles or ~ 0.94 byte/cycle (read)
- Stream bytes read + bytes written / time, so stream estimate is 2 * 0.94 byte/cycle, or 1.3 GB/sec
- This is typical (if anything, better than many systems because L2 miss cost is low)
- (there’s more to this analysis, of course)
Example: Sparse Matrix-Vector Product

- Common operation for optimal (in floating-point operations) solution of linear systems
- Sample code (in C):
  for row=1,n
   \[ m = i[row] - i[row-1]; \]
   \[ \text{sum} = 0; \]
   for k=1,m
     \[ \text{sum} += *a++ * x[*j++]; \]
   \[ y[i] = \text{sum}; \]
- Data structures are \( a[\text{nnz}], j[\text{nnz}], i[n], x[n], y[n] \)

Simple Performance Analysis

- Memory motion:
  - \( \text{nnz} \) (\( \text{sizeof(double)} + \text{sizeof(int)} \)) + \( n \) (\( 2 \times \text{sizeof(double)} + \text{sizeof(int)} \))
  - Assume a perfect cache (never load same data twice; only compulsory loads)
- Computation
  - \( \text{nnz} \) multiply-add (MA)
  - Roughly 12 bytes per MA
  - Typical WS node can move 1-4 bytes/MA
  - Maximum performance is 8-33% of peak

Realistic Measures of Peak Performance

- Sparse Matrix Vector Product
  - One vector, matrix size, \( m = 90,708 \), nonzero entries \( nz = 5,047,120 \)

Comments

- Simple model based on memory performance gives good bounds on performance
  - Detailed prediction requires much more work; often not necessary or relevant to the algorithm designer
- Note that a key feature of the model is the use of measured sustained memory bandwidth
  - In many cases, achieved performance is close to that limit; advanced techniques, such as auto-tuners, cannot significantly boost performance
  - But the measured memory bandwidth is low relative to the raw hardware bandwidth…
Prefetch Engine on IBM Power Microprocessors

- Beginning with the Power 3 chip, IBM provided a hardware component called a prefetch engine to monitor cache misses, guess the data pattern (“data stream”) and prefetch data in anticipation of their use.
- Power 4, 5 and 6 microchips enhanced this functionality.

<table>
<thead>
<tr>
<th>Data Streams</th>
<th>L3 Cache (MB)</th>
<th>L3 Cache (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power 4</td>
<td>8</td>
<td>1.5</td>
</tr>
<tr>
<td>Power 5</td>
<td>8</td>
<td>1.875</td>
</tr>
<tr>
<td>Power 6</td>
<td>16</td>
<td>4</td>
</tr>
</tbody>
</table>

Data Stream and Cache Information

Inefficiency of CSR and BCSR formats

- The traditional CSR and Blocked CSR are hard to reorganize for data streams (esp > 2 streams) to enable prefetch, since the number of non-zero elements or blocks for every row may be different.
- Blocked CSR (BCSR) format can improve performance for some sparse matrices that are locally dense, even if a few zeros are added to the matrix.
  - If the matrix is too sparse (or structure requires too many added zeros), BCSR can hurt performance

Streamed Compressed Sparse Row (S-CSR) format

- S-CSR format partitions the sparse matrix into blocks along rows with size of bs. Zeros are added in to keep the number of elements the same in each row of a block. The column indices for ZEROS in each row are set to the index of the last non-zero element in the row. The first rows of all blocks are stored first, then second, third … and bs-th rows.
- For the sample matrix in the following Figure, NNZ = 29. Using a block size of bs = 4, it generates four equal length streams R, G, B and P. This new design only adds 7 zeros every 4 rows.

Streamed Blocked Compressed Sparse Row (S-BCSR) format

- When the matrix is locally dense and can be blocked efficiently with a few ZEROS added in, we can restore the blocked matrix using the similar idea as S-CSR format. The first rows of all blocks are stored first, then second, third … and last rows. Using 4x4 block for example, it will generate R, G, B and P four equal length streams. We call this the Streamed Blocked Compressed Row storage format (S-BCSR).
### Codes for CSR and S-CSR-2 formats

**CSR**
```c
void CSR(double *v, double *x, double *z, int *ii, int *idx, int NROW) {
    int   i, j, n, *idxp;    double sum1, *v1, xb;
    #pragma omp parallel for 
    private(i,j,n, sum1, v1, idxp, xb) 
    schedule(static)
    for (i=0; i<NROW; i++) {      sum1 = 0.0;      n = ii[i] - ii[0]; v1 = v+n;       idxp = idx+n;      for (j=ii[i]; j<ii[i+1]; j++) {         xb = *(x + (*idxp++));         sum1 += (*v1++)*xb;      }      z[i] = sum1;    }
}
```

**S-CSR-2**
```c
void S_CSR_2(double *v, double *x, double *z, int *ii, int *idx, int NROW) {   int rbs = 2;   int MROW =NROW/rbs, rrows = NROW%rbs, rslast = rbs;   if(rrows > 0) {MROW++;  rslast = rrows;}   double *v1, *v2;   int    *ix1, *ix2;
    int   i, j, k, nl, nm, ilen;    double sum1, sum2;    ilen  = ii[MROW]  - ii[0];    ix1 = idx;       v1 = v;    ix2 = ix1+ilen;  v2 = v1+ilen;    int MNR = MROW-1;    double *v10,  *v20;    int    *iix1, *iix2;
    #pragma omp parallel for 
    private(i,j,nm, sum1,sum2,v10,v20,iix1,iix2) 
    schedule(static)
    for (i=0; i<MNR; i++) {       sum1 = sum2 = 0.0;      nm = ii[i] - ii[0];     // two streams      v10 = v1  + nm;   v20 = v2  + nm;      iix1= ix1 + nm;   iix2 = ix2 + nm;      for (j=ii[i]; j<ii[i+1]; j++) {         sum1 += *(v10++)*x[*iix1++];         sum2 += *(v20++)*x[*iix2++];      }//j      z[rbs*i  ] = sum1;     z[rbs*i+1] = sum2;    }//i
    i = MNR;    if (rrows == 1 ) {       sum1 = 0.0;       nm = ii[i] - ii[0];       v10 = v1  + nm;       iix1= ix1 + nm;       for (j=ii[i]; j<ii[i+1]; j++)                sum1 += *(v10++) * x[*iix1++];       z[rbs*i  ] = sum1;    }
}
```

### Codes for BCSR-4 and S-BCSR-4 formats

**BCSR-4**
```c
void BCSR_4(double *v, double *x, double *z, int *ii, int *idx, int MROW) {      int   i, j, n, *idxp;    double x1,x2,x3,x4,x5, sum1,sum2,sum3,sum4,sum5;    double *xb, *v0;    #pragma omp parallel for 
    private(i, j, n, sum1, sum2, sum3, sum4, sum5, v0, xb, idxp)  
    schedule(static)
    for (i=0; i<MROW; i++) {      n  = ii[i] - ii[0];      v0 = v+16*n;       idxp = idx+n;      sum1 = sum2 = sum3 = sum4 = 0.0;      for (j=ii[i]; j<ii[i+1]; j++) {        xb = x + 4*(*idxp++);        x1 = xb[0]; x2 = xb[1]; x3 = xb[2]; x4 = xb[3];        sum1 += v0[ 0] *x1  + v0[ 1] *x2 + v0[ 2] *x3  + v0[ 3] *x4;        sum2 += v0[ 4] *x1  + v0[ 5] *x2 + v0[ 6] *x3  + v0[ 7] *x4;        sum3 += v0[ 8] *x1  + v0[ 9] *x2 + v0[10]*x3  + v0[11]*x4;        sum4 += v0[12]*x1 + v0[13]*x2  +v0[14]*x3   + v0[15]*x4;        v0 += 16;      }      z[4*i  ] = sum1;        z[4*i+1] = sum2;      z[4*i+2] = sum3;      z[4*i+3] = sum4;    } }
```

**S-BCSR-4**
```c
void S_BCSR_4(double *v, double *x, double *z, int *ii, int *idx, int MROW) {    int    i, j, n, len, *idxp;    double x1,x2,x3,x4,x5, sum1,sum2,sum3,sum4;    double *xb, *v0, *v1, *v2, *v3, *v4, *v10, *v20, *v30, *v40;    len = (ii[MROW] - ii[0])*4;    v1 = v;   v2 = v+len;  v3 = v+len*2;    v4 = v+len*3;    #pragma omp parallel for 
    private(i,j,n,sum1,sum2,sum3,sum4,v10,v20,v30,v40,xb,idxp) 
    schedule(static)    for (i=0; i<MROW; i++) {       n  = ii[i] - ii[0];       v10 = v1+4*n;     v20 = v2+4*n;      v30 = v3+4*n;     v40 = v4+4*n;      idxp = idx+n;      sum1 = sum2 = sum3 = sum4 = 0.0;      for (j=ii[i]; j<ii[i+1]; j++) {        xb = x + 4*(*idxp++);        x1 = xb[0]; x2 = xb[1]; x3 = xb[2]; x4 = xb[3];        sum1 += v10[0]*x1 + v10[1]*x2 + v10[2]*x3  + v10[3]*x4;        sum2 += v20[0]*x1 + v20[1]*x2 + v20[2]*x3  + v20[3]*x4;        sum3 += v30[0]*x1 + v30[1]*x2 + v30[2]*x3  + v30[3]*x4;        sum4 += v40[0]*x1 + v40[1]*x2 + v40[2]*x3   + v40[3]*x4;        v10  += 4; v20 += 4; v30 += 4; v40 += 4;      }      z[4*i  ] = sum1;       z[4*i+1] = sum2;      z[4*i+2] = sum3;     z[4*i+3] = sum4;    } }
```

### Some Sparse Matrices used in the tests

- We used matrices from the University of Florida collection
- All the codes were compiled with `xlc_r –O3 –qstrict –q64 -qarch=auto`. 64KB page size was set for text and data on Power5 and Power6 chips.
- Performance measured is that average of three runs after a “cold start” run

### Performance Ratio compared to CSR format

- S-CSR format is better than CSR format for all (on Power 5 and 6) or Most (on Power 4) matrices
- S-BCSR format is better than BCSR format for all (on Power 6) or Most (on Power 4 and 5) matrices
- Blocked format performance from $\frac{1}{3}$ to 3x CSR.
S-CSR formats with two and four streams
- S-BCSR-4 is generally better than S-BCSR-2 on Power 6.
- On Power 4 and 5, these two are mixed.
- S-CSR-4 format can achieve over 2x performance improvement of CSR.

MPI with 4 nodes
- Parallel tests with MPI using 4 nodes on P5 and P6. At most 50% improvement achieved.
- Probably due to communication overhead (these are small matrices)

SMP with 4 threads
- SMP with 4 threads also tested on P5 and P6. Typical performance boost of 1.5-2x over CSR
- Shows prefetch works with multiple threads (more tests needed)

Comparison of S-BCSR-4 format to BCSR-4 format
- The matrices are chosen with large data size (> 32 MB) and the performance of BCSR format is close to or better than CSR format.
- Performance Improvement of S-BCSR-4 format compared to BCSR-4 format:
  - P4: 20 -60%, P5: 30 -45%, P6: 75 -108%
Streamed format on Intel processors

- The tests were also run on abe.ncsa.uiuc.edu (Xeon/Clovertown, 2.33 GHz, 2x4 MB L2 cache)
- S-CSR-2 and/or S-CSR-4 format can result in better performance than CSR format for many matrices.
- S-BCSR-4 format is better than BCSR-4 format for all the matrices except for "bcsstm27", which is small and fits in cache. For most matrices, S-BCSR format provides a 10 - 20% of performance improvement.

Summary and Future Work

- The streamed CSR and BCSR storage formats can significantly improve the performance of SpMV for a variety of matrices on IBM processors. Over 100% performance improvement can be achieved.
  - Simulation results for POWER7 also are good
  - The new formats also show the benefits on Intel processors.
  - We will compare the new format with other auto-tuning packages, such as Berkeley-OSKI, and with other approaches to improve performance within rows (such as sorted CSR)
    - New formats will be provided within PETSc
    - Initial results: S-CSR provides more performance than OSKI
    - Alignment and SIMD instructions will also be considered in the new formats.
  - Perhaps most important – extension to other matrix-vector operations used in preconditioners

Yet Another Complication

- How many loads and stores from memory are required by $a = b$
- Natural answer is
  - One load (b), one store (a)
- For cache-based systems, the answer is
  - Two loads: cacheline containing b and cacheline containing a
  - One store: cacheline containing a
- Recall the 2 read for one write path to L1 cache
- For main memory, max rate is then $2/3 \times 5.5\text{GB}/\text{sec}$, or 3.7 GB/s
  - Still almost double the best, hand-tuned stream kernel

Peak and Observed Memory Performance

- This diagram leaves off the most important information - latency between components.
- Stated values are never observed by the programmer.
Sparse Matrix and Vector Multiply (SpMV)

- SpMV is a key operation
  - in iterative methods
  - in explicit methods, where the operator can be written as a matrix-vector product.
- Much effort has been focused on developing effective data structures for this problem including the recent application of auto-tuning approaches to optimize aspects of the data structure.
- As we’ve seen, a major impediment to improving performance of this operation is that it is memory-bandwidth intensive.

Dense Matrix Vector Multiply

- N×N Dense Matrix-Vector multiply is also tested. N is up to 10,000.
  - Two block sizes 4x4 and 5x5 are employed.
  - The tests clearly show better performance with S-BCSR format.

Dense Matrix Vector Multiply (cont’d)

- When N is larger and the data is out of L3 cache, S-BCSR format is steadily better than the original BCSR format.

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Power 4</th>
<th>Power 5</th>
<th>Power 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4x4</td>
<td>25 – 35</td>
<td>45 – 50</td>
<td>70 – 90</td>
</tr>
<tr>
<td>5x5</td>
<td>20 – 60</td>
<td>100 – 150</td>
<td>60 – 90</td>
</tr>
</tbody>
</table>

Streamed formats on Intel processors (cont’d)

- The results of Dense Matrix-Vector Multiply show that streamed BCSR format can improve 10-25% compared to the traditional BCSR format.