Efficient and Scalable Sparse Matrix-Vector Multiplications on Cache-Based GPUs

István Reguly, Endre László, Mike Giles
Oxford e-Research Centre

Sparse Linear Algebra Solvers for High Performance Computing Workshop
University of Warwick, 2013 July 8.-9.
**Sparse Matrix-Vector multiplication (spMV)**

- Most widely used format is CSR
- Conversion to other formats is expensive and degrades flexibility, isn’t much faster
- Bandwidth-limited operation
- Doesn’t strong-scale very well over MPI
- Sensitivity to the size of the matrix
  - In AMG we have increasingly smaller matrices, but still want to strong scale
NVIDIA Fermi/Kepler

- Introduction of caches on GPUs:
  - 16k/48k L1 (per SM) and 768k L2 on Fermi
  - 48k read-only texture cache (per SMX) and 1.5M L2 on Kepler

- Global fetch operations load 128 byte cache lines

- 384 lines in L1 cache (vs. ~1536 threads)

- Cache reuse is important, trashing is expensive

- Throughput, not latency oriented machines
Matrix structure vs. GPU parameters

- Number of rows and nonzeros
  - Average number of nonzeros per row
- Std. deviation of the number of nonzeros per row
- “Structuredness”
  - Similarity of column indices in consecutive rows

- Number of threads in a block
  - Occupancy: ability to overlap memory transactions with computations
- Number of blocks
  - GPU likes a lot of work
- Warps and coalescing
  - Affects achieved bandwidth
Experiments

• Tuning the spMV operation on the GPU
• General purpose, compared against CUSPARSE 5.5
• 49 Florida matrices
  – Mostly from PDEs, but some others (road networks, circuit simulations, web-search graphs, etc.)
• Single K20c
Compressed Sparse Row

Row pointers: 0 3 7 10 14

Column indices: 0 1 5 1 3 6 7 0 1 3 3 4 5 7

Values: 0.3 1.4 0.5 1.1 3.7 0.6 7.1 1.0 0.1 3.2 8.3 1.4 4.5 2.7
Naïve implementation

```
int i = blockIdx.x*blockSize + threadIdx.x;
floa t rowSum = 0;
int rowPtr = rowPtrs[i];
for (int j = 0; j<rowPtrs[i+1]-rowPtr; j+=1) {
    rowSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
}
y[i] = rowSum;
```

Coalescing/Caching
Bad for long rows, good for short ones
Thread cooperation

coop = 4

- Coalesced access
- Good caching
- Warp divergence
__global__ void csrmv(float *values, int *rowPtrs,
                    int *colIdxs, float *x, float *y,
                    int dimRow, int repeat, int coop) {
    int i = (repeat*blockIdx.x*blockDim.x + threadIdx.x)/coop;
    int coopIdx = threadIdx.x%coop;
    int tid = threadIdx.x;
    extern __shared__ volatile float sdata[];
    for (int r = 0; r<repeat; r++) {
        float localSum = 0;
        if (i<dimRow) {
            // do multiplication
            int rowPtr = rowPtrs[i];
            for (int j = coopIdx; j<rowPtrs[i+1]-rowPtr; j+=coop) {
                localSum += values[rowPtr+j] * x[colIdxs[rowPtr+j]];
            }
            // do reduction in shared mem
            sdata[tid] = localSum;
            for(unsigned int s=coop/2; s>0; s>>=1) {
                if (coopIdx < s) sdata[tid] += sdata[tid + s];
            }
            if (coopIdx == 0) y[i] = sdata[tid];
            i += blockDim.x/coop;
        }
    }
}

Parameters:
- coop
- repeat
- blockSize
Number of cooperating threads

<table>
<thead>
<tr>
<th>Number of cooperating threads</th>
<th>Run time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>atmosmodd (2.63)</td>
</tr>
<tr>
<td>2</td>
<td>crankseg_2 (14.89)</td>
</tr>
<tr>
<td>4</td>
<td>shallow_water1 (2.00)</td>
</tr>
<tr>
<td>8</td>
<td>webbase–1M (1.76)</td>
</tr>
<tr>
<td>16</td>
<td>cant (8.01)</td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>
The fixed rule

- Optimal parameters depend on matrix
- Auto-tuning, exhaustive search...
- General-purpose library
  - “No” overhead is tolerated
  - Quick, heuristic decision based on information:
    - Number of rows/columns
    - Number of nonzeros
The fixed rule

• blockSize = 128
• coop next bigger power of two to:
  \[ \sqrt{\frac{\text{num\_nonzeros}}{\text{num\_rows}}} \times 1.6 \]
• repeat and the number of blocks: aim for at least 5000 blocks
Instruction throughput

12/17/20 GFlops single, 11/14/16 GFlops double
Bandwidth estimation

- Minimal, effective bandwidth
- On caching architectures “effective” bandwidth is not the worst case
  - 128 byte cache line
- Number of cache lines touched by blocks
  - No sharing between blocks
  - Infinite cache size
Achieved bandwidth

Fixed rule: 101/70/98 GB/s single, 82/57/80 GB/s double
Optimised: 119/83/115 GB/s single, 97/68/94 GB/s double
Dynamic run-time tuning

- As seen, there is quite a gap between the fixed rule and the optimum
- No time for offline optimisations
- Iterative algorithms
  - Conjugate Gradient
  - Newton-Raphson
- If the matrix structure is the same
  - Tune these parameters
  - Correct output in reasonable time
Tuning over 49 matrices

CUSPARSE 52%
Improvement over CUSPARSE

Fixed rule: 1.38* in single and 1.32* in double precision
Tuned: 1.88* in single and 1.72* in double precision
Strong-scaling spMV

- Research for NVAMG scaling (as an NVIDIA intern)
- 6 2D/3D ANSYS matrices (4*4 blocks, u,v,w,p)
  - Dumped at different levels from NVAMG
  - 1 nonzero = 16 doubles
- Scaling 1->8 GPUs: timing experiments run on NVIDIA’s Professional Services Group (PSG) cluster

- Factors
  - Redundant compute (multiple halo-rings)
  - Method of memory transfer of halo data
  - 2 or 4 GPU nodes

- How far can we push strong scaling?
  - 80 Jacobi iterations
Single GPU spMV performance

1 GPU: number of nonzeros vs. 80 Jacobi iterations

- Execution time (s)
- Number nonzeros

- Matrix 1
- Matrix 2
- Matrix 3 (2D)
- Matrix 4
- Matrix 5
- Matrix 6
Distributed spMV

\[ y = Ax \]

Halo information (of vector \( x \)) is required to carry out computations

Can do latency hiding:
- Asynchronously send boundary information
- Do computation on the interior
- Wait for halo to arrive
- Do computation on the boundary
Issues with multi-GPU

• As matrices get smaller, ratio of boundary and halo over interior increases
• GPUs are good at **throughput**, not latency
  – Significant overhead of getting the data to/from the GPU
  – GPU -> CPU -> MPI -> CPU -> GPU
• At very small sizes, launch latency is significant
  – Operations over the boundary are usually small
  – “Wasted” CPU time spent launching a GPU kernel
spMV scaling

Speedup over 1 GPU with 1 ring 2–8 GPUs

Number nonzeros

Speedup of 1 ring distributed over 1 GPU

- 2 GPUs
- 4 GPUs
- 6 GPUs
- 8 GPUs
Halo size

Ratio of halo nodes to owned nodes

Number of total nodes

2 GPUs
4 GPUs
6 GPUs
8 GPUs
Redundant compute

• We are concerned about latency
• Create overlapping partitions (multiple halo-rings), which requires rows of the matrix to be present on the “other side”
• Send more data, less frequently

1 ring, 2 ring, 4 ring
Redundant compute

Speedup over 1 GPU with 4 ring 2–8 GPUs

Number nonzeros

Speedup of 4 ring distributed over 1 GPU

2 GPUs
4 GPUs
6 GPUs
8 GPUs
4-ring over 1-ring

Speedup of 4 ring vs. 1 ring with 2–8 GPUs

Speedup of 4 ring over 1 ring

Number nonzeros

- 2 GPUs
- 4 GPUs
- 6 GPUs
- 8 GPUs

- single GPU flatline
- 4-ring > 1GPU
- 4 ring < 1GPU
Level 0 – 200k rows, 1.2M nonzeros

- Compute interior
- 1-ring halo 3.94 ms
- 160MB interior 12KB halo
- Gather halo
- Compute boundary
- Upload halo and scatter
- Update x

Level 6 – 2.5k rows, 30k nonzeros

- Compute interior
- 1-ring halo 730 us
- 4MB interior 0.5KB halo
- Compute owned + 2 rings
- Compute boundary + 3 rings
- Compute owned + 0 rings

- 4-ring halo 300 us
- 4MB interior 2.5KB halo
Zero-copy transfers

• Host memory mapped into GPU memory space
  – GPU kernels can directly write to pinned memory on the CPU
• Eliminates the need for an explicit cudaMemcpy
• Boundary gather, halo scatter kernels
Zero-copy transfers

Speedup of 1–2–4 ring host-copy vs. zero-copy with 2–6 GPUs

More useful with no redundant compute

Bad at a high number of vertices (caching?)
2 or 4 GPU nodes

Speedup of 1–2–4 ring 2 GPU nodes vs 4 GPU nodes with 2–6 GPUs

Number nonzeros

Speedup of 4 GPU node over 2 GPU node over 2 GPU node

Graph showing the speedup of different configurations of GPU nodes with varying number of nonzeros. The x-axis represents the number of nonzeros, and the y-axis represents the speedup of the 4 GPU node over the 2 GPU node.
Conclusions

• Getting good spMV performance on the GPU is not easy
  – Several parameters, can be tuned to great effect
  – But can be generalised and put in a library

• The bottleneck when scaling is latency
  – On multiple levels: MPI, CPU-GPU, GPU
  – Redundant compute helps more as the problem gets smaller
  – Other tricks to improve scalability