Dense Matrix-Vector Multiplication

- SGEMV / DGEMV in BLAS
- Memory-bound performance

Dense Matrix
Sparse Matrix-Vector Multiplication

- One multiply-add per nonzero entry
- Some reuse of vector data

Sparse Matrix
Performance

- Performance is *Memory-Bound*
  - 5-20 GFLOP/s is typical

- Low Arithmetic Intensity
  - 2 flops: 8+ bytes (*float*)
  - 2 flops: 16+ bytes (*double*)

- Primary objective
  - Use memory bandwidth efficiently
Performance

- **Tesla C2050 floating point performance**
  - Single 1,030 GFLOP/s (peak)
  - Double 515 GFLOP/s (peak)

- **Tesla C2050 memory bandwidth**
  - 144 GB/s (peak)

- **Intensity Threshold**
  - 7.14 FLOP : Byte (single)
  - 3.57 FLOP : Byte (double)
Performance

- **Tesla C2050 threshold**
  - 7.14 FLOP : Byte (single)
  - 3.57 FLOP : Byte (double)

- **Dense Matrix-Vector Multiplication Intensity**
  - 0.25 - 0.50 FLOP : Byte (single)
  - 0.12 - 0.25 FLOP : Byte (double)

- **Sparse Matrix-Vector Multiplication Intensity**
  - 0.12 - 0.50 FLOP : Byte (single)
  - 0.08 - 0.25 FLOP : Byte (double)
Performance Considerations

- Use memory bandwidth efficiently
  - Memory coalescing

- Expose lots of fine-grained parallelism
  - Need thousands of threads

- Find opportunities for reuse
  - Make use of caching
Memory Coalescing

Recall: *blocks* divided into physical *warps*
Memory Coalescing

- Fully Coalesced Memory Access
Memory Coalescing

Partially Coalesced Memory Access

gap
Memory Coalescing

- Misaligned memory access
Memory Coalescing

- Uncoalesced Memory Access
  - Separated by 32+ words
Memory Coalescing (SAXPY)

- Example: SAXPY with stride
  - Fully Coalesced when stride is 1

```c
for (int i = 0; i < N; i++)
    y[stride * i] += a * x[stride * i];
```
Memory Coalescing (SAXPY)
Memory Coalescing (SAXPY)

Example: SAXPY with offset
- Aligned when offset is 0

```c
for (int i = 0; i < N; i++)
    y[i + offset] += a * x[i + offset];
```
Memory Alignment (SAXPY)

Efficiency vs. Aligned Offset

- Tesla C2050
- GeForce GTX 280
- GeForce 8800 GTS
Types of Memory Access

- Reading matrix structure
  - Determined by matrix format
  - Most bandwidth consumption

- Reading source vector (x)
  - Determined by matrix structure
  - Little control over access pattern
  - Potential reuse

- Writing destination vector (y)
  - Little bandwidth consumption
Compressed Sparse Row (CSR)

- Rows laid out in sequence
- Inconvenient for fine-grained parallelism
void csr_spmv(int num_rows,
               int * row_offsets,
               int * column_indices,
               float * values,
               float * x,
               float * y)
{
    for (int row = 0; row < num_rows; row++)
    {
        int row_begin = row_offsets[row];
        int row_end = row_offsets[row + 1];

        float sum = 0;

        for (int offset = row_begin; offset < row_end; offset++)
            sum += values[offset] * x[column_indices[offset]];

        y[row] = sum;
    }
}
CSR (scalar) kernel

```c
__global__
void csr_spmv(int num_rows,
               int * row_offsets,
               int * column_indices,
               float * values,
               float * x,
               float * y)
{
    int row = blockDim.x * blockIdx.x + threadIdx.x;

    if (row < num_rows)
    {
        int row_begin = row_offsets[row];
        int row_end = row_offsets[row + 1];

        float sum = 0;

        for (int offset = row_begin; offset < row_end; offset++)
            sum += values[offset] * x[column_indices[offset]];

        y[row] = sum;
    }
}
```
CSR (scalar) kernel

- One thread per row
  - Poor memory coalescing
  - Unaligned memory access
CSR (vector) kernel

- One SIMD vector or *warp* per row
  - Partial memory coalescing
  - Unaligned memory access
CSR (vector) kernel

- Reduce partial sums in shared memory
  - Example: warp of 8 threads

Shared memory
ELL kernel

Full coalescing

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5

column-major ordering
ELL kernel

Pad columns for alignment
DIA kernel

- Same as ELL

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5

padding
**COO kernel**

- One thread per nonzero element
- Fully coalesced
COO kernel

- Store $i$ and $A(i,j) \times x(j)$ in shared memory
- Compute row sums using \textit{segmented} reduction
Memory Coalescing Summary

- **Full Coalescing**
  - DIA, ELL, and COO

- **Partial Coalescing**
  - CSR (vector)
  - Efficiency depends on row length

- **Little Coalescing**
  - CSR (scalar)
Performance Considerations

- Use memory bandwidth efficiently
  - Memory coalescing

- Expose lots of fine-grained parallelism
  - Need thousands of threads

- Find opportunities for reuse
  - Make use of caching
Exposing Parallelism

- **DIA, ELL & CSR (scalar)**
  - One thread per row

- **CSR (vector)**
  - One warp per row

- **COO**
  - One thread per nonzero
Exposing Parallelism

All matrices have 4M nonzeros
Exposing Parallelism

GPU: GeForce GTX 285
Exposing Parallelism

- One thread per row
  - ELL, DIA, and CSR (scalar) kernel
  - Generally good enough (>20K rows is common)

- One warp per row
  - CSR (vector) kernel
  - Fewer rows is sufficient (>256)

- One thread per entry
  - COO kernel
  - Insensitive to matrix shape
Performance Considerations

- Use memory bandwidth efficiently
  - Memory coalescing

- Expose lots of fine-grained parallelism
  - Need thousands of threads

- Find opportunities for reuse
  - Make use of caching
Caching

Repeated accesses to source vector

- Used 3 times
- Used 1 time
- Used 2 times
Caching

Effectiveness depends on matrix structure

- Used 2 times
- Never used
- Used 1 time
Caching

- Fermi architecture has L1 cache
  - No effort needed

- Earlier architectures have texture cache
  - Often worth ~30% improvement

- Software-managed cache
  - Preload into shared memory

- Effectiveness depends on matrix structure
Other Techniques

- Block Formats
  - Reduce index overhead

- Multiple Vectors
  - Reuse matrix data
Performance Considerations

- Use memory bandwidth efficiently
  - Memory coalescing

- Expose lots of fine-grained parallelism
  - Need thousands of threads

- Find opportunities for reuse
  - Make use of caching
References

*Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors*
Nathan Bell and Michael Garland
Proceedings of Supercomputing ’09

*Efficient Sparse Matrix-Vector Multiplication on CUDA*
Nathan Bell and Michael Garland

*Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs*
Jee Whan Choi, Amik Singh and Richard W. Vuduc
Proceedings of Principles and Practice of Parallel Programming (PPoPP) 2010