Lecture 10 - GP-GPU and High Performances Computing

Sparse Methods

Previously

Key concepts from previous lectures:

- Dense matrix operations and memory access patterns
- Parallel reduction techniques
- Memory coalescing and warp-level operations
- Performance optimization strategies

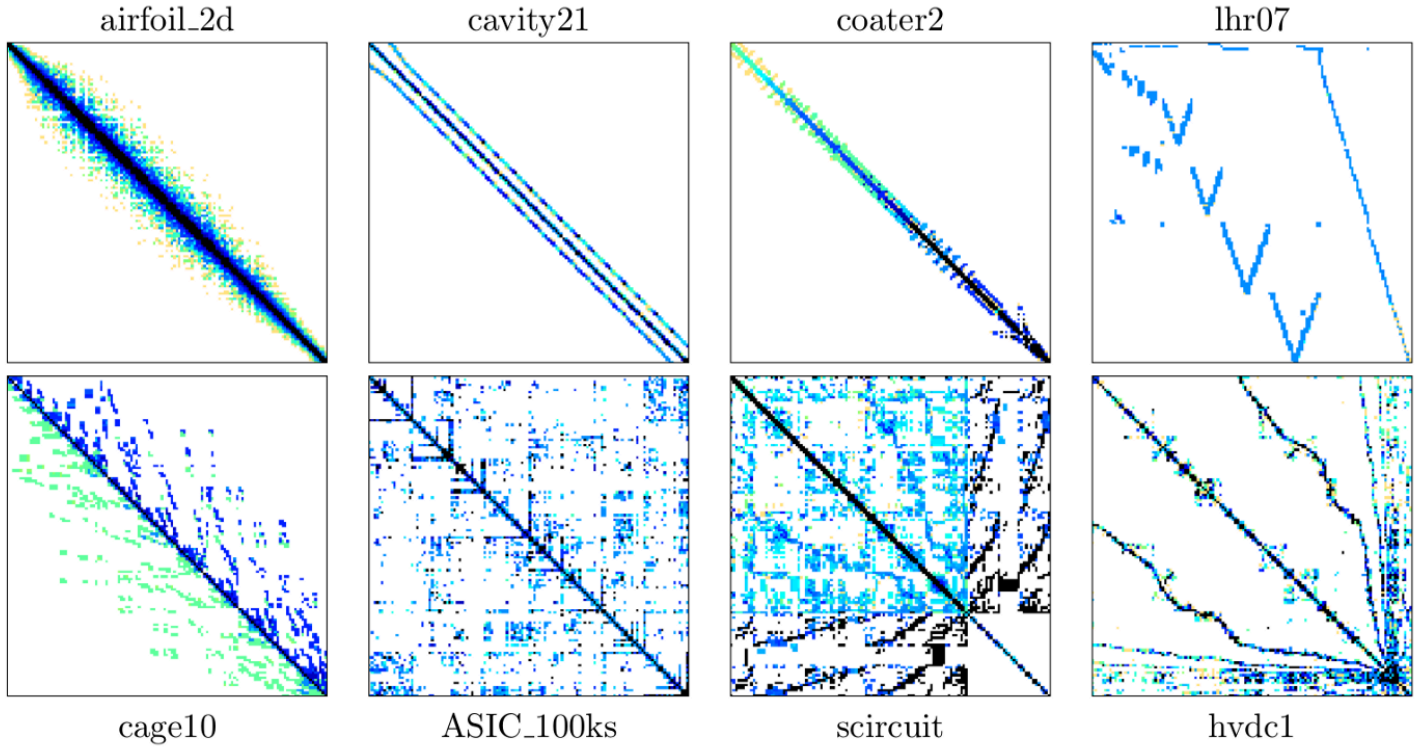
Sparse Matrix Computation

Objectives

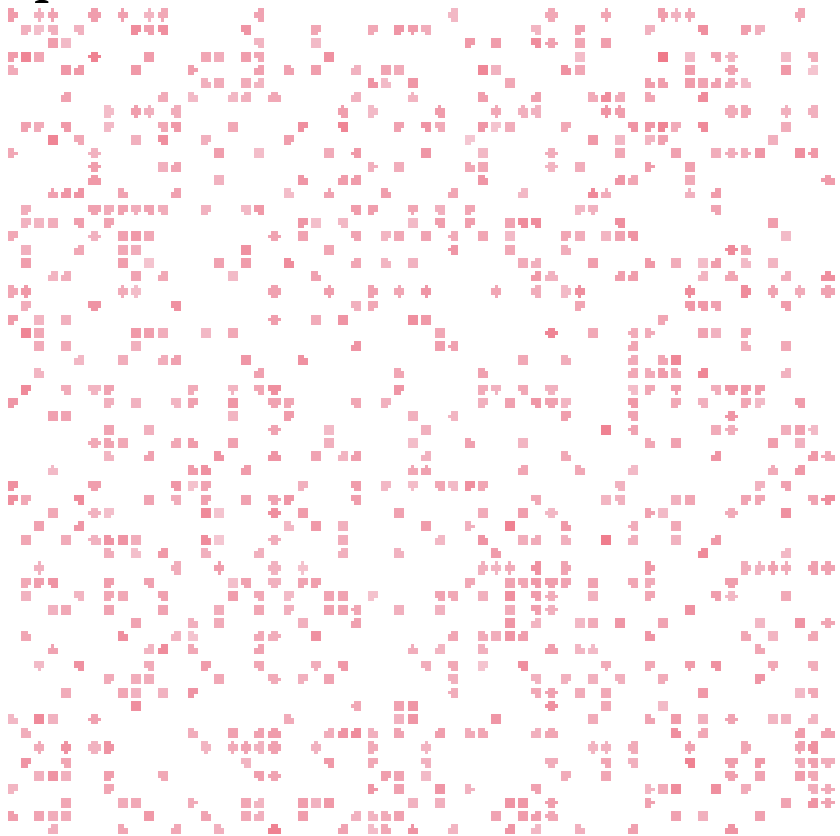
The key techniques for compacting input data in parallel sparse methods for reduced consumption of memory bandwidth:

- Better utilization of on-chip memory
- Fewer bytes transferred to on-chip memory
- Retaining regularity

Sparse Data Examples



Sparse Data



Many real-world inputs are sparse/non-uniform:

- Signal samples
- Mesh models
- Transportation networks
- Communication networks
- etc.

Sparse Matrix

- Many real-world systems are sparse in nature
- **Solving sparse linear systems**:
  - Solving these systems requires inversion of the coefficient matrix
  - Traditional inversion algorithms such as Gaussian elimination can create too many "fill-in" elements and explode the size of the matrix
  - Iterative Conjugate Gradient solvers based on sparse matrix-vector multiplication are preferred
- Solution of PDE systems can be formulated into linear operations using sparse matrix-vector multiplication

Challenges

Compared to dense matrix multiplication, SpMV:

- **Irregular/unstructured** - Non-zero pattern is unpredictable
- **Little input data reuse** - Each element accessed once
- **Limited compiler optimization** - Compiler can’t predict access patterns

Key to maximal performance:

- **Maximize regularity** - Reduce thread divergence and load imbalance
- **Maximize DRAM burst utilization** - Arrange data for coalesced access
- **Minimize memory traffic** - Use compact storage formats

A Simple Parallel SpMV

Row	Col 1	Col 2	Col 3	Col 4	Col 5	Thread
Row 0	**1**	0	0	**1**	0	Thread 0
Row 1	**3**	**2**	0	**3**	0	Thread 1
Row 2	**6**	0	**8**	**9**	**2**	Thread 2
Row 3	0	0	**5**	**9**	0	Thread 3
Row 4	0	0	0	0	**25**	Thread 4

One thread per row: The simplest parallelization strategy

Each thread computes: $y[i] = \sum_{j} A[i,j] \cdot x[j]$ for its assigned row

Storage (CSR Format)

To simplify the storage we use the following data structures:

AA - Array of non-zero values (row-major order):

```
AA[12] = {1.0, 1.0, 3.0, 2.0, 3.0, 6.0,
          8.0, 9.0, 2.0, 5.0, 9.0, 25.0}
```

JA - Column indices corresponding to values in AA (0-based):

```
JA[12] = {0, 3, 0, 1, 3, 0, 2, 3, 4, 2, 3, 4}
```

IA - Row pointers (IAi points to start of row i in AA/JA, 0-based):

```
IA[6] = {0, 2, 5, 9, 11, 12}  // IA[i+1] - IA[i] = nnz in row i
```

CSR: Brief

Compressed Sparse Row (CSR) Format:

Structure:

- AA, JA: nnz elements each
- IA: $n + 1$ elements (row pointers)
- $IA[j + 1] - IA[j]$ = nnz in row j

Access:

- ✅ Fast row access (sequential)
- ❌ Slow column access (search)

Properties:

- No structure assumptions
- Storage: $2 \cdot nnz + n + 1$ words
- Works for any sparsity pattern
- **Alternative**: CSC for column ops

CSR Kernel in CUDA

```
// Each thread processes one row
int row = blockDim.x * blockIdx.x + threadIdx.x;
if (row < num_rows)
{
  float dot = 0;
  int row_start = IA[row];       // Start index in AA/JA arrays
  int row_stop  = IA[row+1];     // End index (exclusive)
  for (int jj = row_start; jj < row_stop; jj++)
    dot += AA[jj] * x[JA[jj]];   // AA: values, JA: column indices
  y[row] = dot;
}
```

Challenges with Standard CSR Kernel

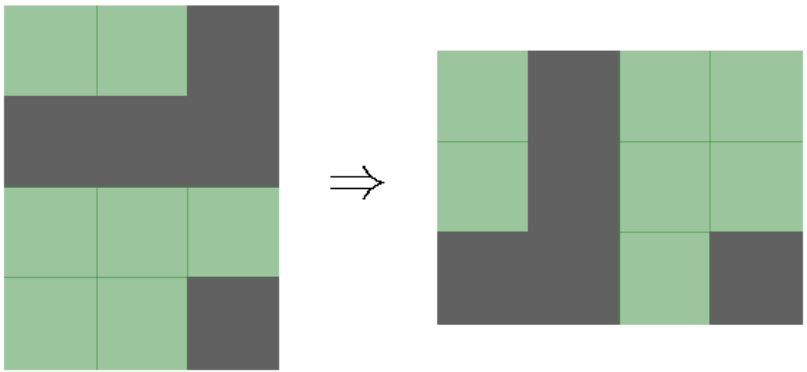
Execution divergence: rows are varying by lengths

→ Within each warp time execution will have a different work load

Memory divergence: uncoalesced accesses

→ Adjacent threads access non-adjacent memory locations

Regularizing Sparse Matrix Vector (ELLPACK)



ELLPACK Format Strategy:

- **Pad all rows** to the same length (dense row length)
- **Column-major layout** for coalesced memory access
- Both AA (values) and JA (column indices) are padded and transposed
- **Trade-off**: Regular structure vs. wasted computation on padding

Inefficiency: If a few rows are much longer than average, padding overhead becomes significant

ELLPack Kernel

```
// Each thread processes one row (column-major layout)
int row = blockIdx.x * blockDim.x + threadIdx.x;
if (row < num_rows) {
  float dot = 0;
  for (int i = 0; i < num_elem; i++) {
    // Column-major: row + i*num_rows gives element at (row, i)
    dot += data[row+i*num_rows] * x[col_index[row+i*num_rows]];
  }
  y[row] = dot;
}
```

ELLPACK Challenges

Advantages:

- Every thread handles one row (balanced workload)
- No row-pointer needed (fixed stride access)
- Regular structure → no divergence within warp
- Column-major layout → coalesced memory access

Disadvantages:

- **Significant overhead for unbalanced problems**
- Padding zeros waste computation cycles
- Memory overhead if max row length >> average row length

Coordinate Storage (COO)

Problem with ELL:

- ELL can cause excessive padding when a small number of rows have many more non-zero elements than average
- This wastes memory and computation on padded zeros

Solution:

- Coordinate format (COO) stores exceptional entries separately
- COO stores a list of $(row, column, value)$ tuples
- Each non-zero is represented independently
- **Advantage**: No padding overhead, perfect for irregular patterns
- **Disadvantage**: Requires atomic operations for accumulation (multiple threads may write to same row)
- COO storage is efficient for very sparse matrices or as a complement to ELL

COO for Maximal Parallelism

List row, column and value for every non-zero entry:

AA - Array of non-zero values:

```
AA[12] = {1.0, 1.0, 3.0, 2.0, 3.0, 6.0
          8.0, 9.0, 2.0, 5.0, 9.0, 25.
```

JA - Column indices (0-based):

```
JA[12] = {0, 3, 0, 1, 3, 0, 2, 3, 4, 2
```

IR - Row indices (0-based):

```
IR[12] = {0, 0, 1, 1, 1, 2, 2, 2, 2, 3
```

Parallelism:

- Each thread is assigned one non-zero entry
- Each thread computes $A[i, j] \times x[j]$ product
- Products are accumulated using atomic operations (or segmented reduction)
- **Insensitive to row length distribution** - perfect load balance

COO Kernel

```
// Each thread processes one non-zero element
int element = blockIdx.x * blockDim.x + threadIdx.x;

if (element < nnz)
  atomicAdd(y + IR[element], AA[element] * x[JA[element]]);
```

⚠️ **Atomic operations required!**

Multiple threads may write to the same row → need atomicAdd for correctness

Memory footprint: $nnz \cdot (val) + 2 \cdot nnz \cdot (int)$

- One value + one row index + one column index per non-zero

Hybrid Approach (ELL/COO)

Best of both worlds:

- **ELL** handles typical entries (most rows)
  - Regular structure → no divergence
  - Column-major layout → coalesced memory access
  - Direct assignment (no atomics needed)
- **COO** handles exceptional entries (overflow rows)
  - Removes padding overhead from ELL
  - Handles rows with many non-zeros efficiently
  - Requires atomic operations for accumulation

Strategy: Set ELL width to average row length, put overflow in COO

Hybrid Kernel

```
// Process ELL part (one thread per row)
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < n_rows) {
  int row = idx;
  data_type dot = 0;
  for (int element = 0; element < elements_in_rows; element++) {
    int element_offset = row + element * n_rows;
    dot += ell_data[element_offset] * x[ell_col_ids[element_offset]];
  }
  y[row] = dot;  // Direct assignment (no atomic needed)
}

// Process COO part (parallel over all elements)
for (int element = idx; element < n_elements;
      element += blockDim.x * gridDim.x) {
  data_type dot = coo_data[element] * x[col_ids[element]];
  atomicAdd(y + row_ids[element], dot);  // Atomic needed (multiple threads per row)
}
```

Storage Requirements

M=rows, N=cols, K=max row nnz, S=sparsity, nnz=M×N×S

Format	Storage (words)	Notes
Dense	M×N	Full matrix
CSR	2×nnz + M + 1	Values + indices + pointers
ELL	2×M×K	Padded to max row
COO	3×nnz	Row + col + value
HYB	> 3×nnz, < 2×M×K	Depends on threshold

Format Comparison

CSR

- ✅ Variable row lengths
- ✅ Compact storage
- ❌ Thread divergence
- ❌ Uncoalesced access

Use when: General-purpose, row-oriented

ELL

- ✅ No divergence
- ✅ Coalesced access
- ❌ Padding overhead
- ❌ Wasted computation

Use when: Uniform row lengths

COO

- ✅ Perfect load balance
- ✅ No padding
- ❌ Atomic operations
- ❌ Higher storage cost

Use when: Very sparse, irregular

HYB

- ✅ Best of ELL + COO
- ✅ Handles outliers
- ❌ More complex
- ❌ Tuning required

Use when: Mixed patterns

Choosing the Right Format

Decision Tree:

```
Uniform structure?
├─ YES → max ≈ avg? → ELL
│        └─ NO → HYB
└─ NO → very sparse? → COO
         └─ NO → CSR
```

Considerations:

- Memory constrained? → CSR
- Column ops? → CSC
- Unpredictable? → HYB
- Using library? → cuSPARSE

cuSPARSE Library

Features:

- Auto format selection/conversion
- Optimized kernels (CSR, CSC, COO, ELL, HYB)
- Multiple data types
- cuBLAS integration

Use when:

- Production code
- Multiple matrix types
- Auto optimization needed

Customize when:

- Specific access patterns
- Fine-grained control needed
- Research/learning
- Extreme performance

Example:

```
cusparseSpMV(handle, op, alpha,
             matA, vecX, beta, vecY, ...);
```

Conclusion

Conclusion

Formats:

- **CSR**: Variable rows, compact
- **ELL**: Regular, padding overhead
- **COO**: Perfect balance, atomics
- **HYB**: Best of both

Practice:

- Use cuSPARSE for production
- Understand formats for debugging
- Consider hardware characteristics
- Memory-bound operations

Performance Considerations:

- Storage requirements depend on sparsity pattern
- Trade-off between regularity (ELL) and efficiency (CSR/COO)
- Hardware characteristics influence format choice
- Achieving high compute-to-memory ratio is challenging for sparse operations