GP-GPU and High Performances Computing

Lecture 09 – Sparse methods

December 1, 2023

Previsouly

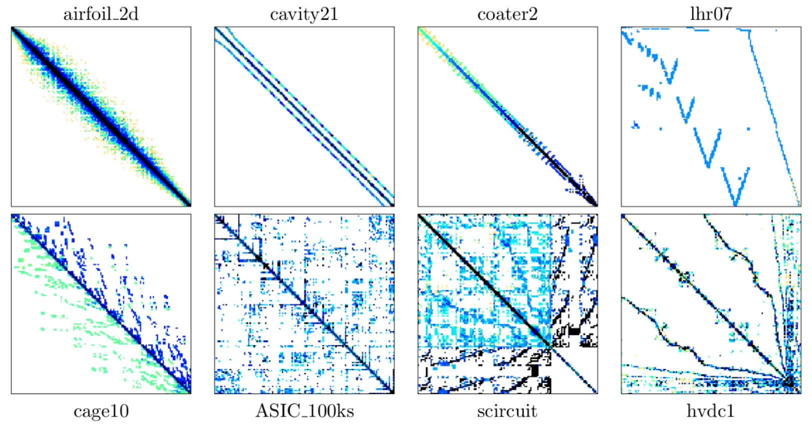
Sparse matrix computation

Objectives

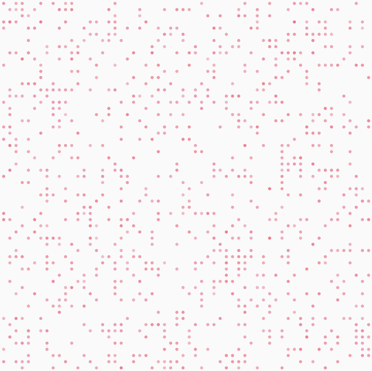
the key techniques for compacting input data in parallel sparse methods for reduced consumption of memory bandwidth

- 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 require 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 is preferred
- Solution of PDE systems can be formulated into linear operations using sparse matrix-vector multiplication

Challenges

Compared to dense matrix multiplication, SpMV

- Is Irregular/unstructured
- Has little input data reuse
- Benefits little from compiler transformation tools

Key to maximal performance

- Maximize regularity (by reducing divergence and load imbalance)
- Maximize DRAM burst utilization (layout arrangement)

A Simple Parallel SpMV

$$\begin{array}{l|ccccc|l}
\text{Row 0} & 1 & 0 & 0 & 1 & 0 & \text{Thread 0} \\
\text{Row 1} & 3 & 2 & 0 & 3 & 0 & \text{Thread 1} \\
\text{Row 2} & 6 & 0 & 8 & 9 & 2 & \text{Thread 2} \\
\text{Row 3} & 0 & 0 & 5 & 9 & 0 & \text{Thread 3} \\
\text{Row 4} & 0 & 0 & 0 & 0 & 25 & \text{Thread 4}
\end{array}$$

The simplest algorithm consists in associating one thread with one row of the input matrix

Storage

To simplify the storage we use the following data structures

```
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[12] = {1, 4, 1, 2, 4, 1, 3, 4, 5, 3, 4, 5}

IA[6] = { 1, 3, 6, 10, 12, 13}
```

CSR: brief

- The number of elements in AA and JA is nnz.
- The number of elements in IA is $n + 1$.
- $IA(j)$ point to the start of line j.
- There is no underlying structure in the matrix.
- Fast row access.
- Slow column access.
- Storage cost $2nnz + n + 1$ instead of n^2.
- No hypothesis on the density of the original matrix.
- Alternative : CSC

CSR kernel in Cuda

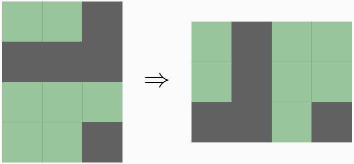
```
int row = blockDim.x * blockIdx.x + threadIdx.x
if ( row < num_rows )
{
  float dot = 0;
  int row_start = ptr[row];
  int row_stop  = ptr[row+1];
  for (int jj = row_start; jj > row_end; jj++)
  dot += data[jj] * x[indices[jj]];
  y[row] += dot;
}
```

Challenges with standard CSR kernel

- Execution divergence: rows are varying by lengths.
  ⇒ Within each wraps time execution will have a different work load.

- Memory divergence: uncoalesced accesses.
  ⇒ Adjacent threads access non-adjacent memory locations

Regularizing sparse matrix vector



- Pad all rows to the same length
- Inefficient if a few rows are much longer than others Transpose (Column Major) for DRAM efficiency
- Both AA and JA padded/transposed

ELLpack kernel

```
int row = blockIdx.x * blockDim.x + threadIdx.x;
if (row < num_rows) {
  float dot = 0;
  for (int i = 0; i < num_elem; i++) {
    dot += data[row+i*num_rows]*x[col_index[row+i*num_rows]];
    y[row] = dot;
  }
}
```

Challenges

- Every “thread” handles the computation of one sum in local memory.
- Balanced workload: add artificial zero elements, no row-pointer needed.
- Can result in significant overhead for unbalanced problems.

Coordinate storage

- ELL can cause excessive padding: this padding is caused by a small number of rows that possessed an excessive large number of non zero elements.
- Coordinated format (COO) to take away some elements of this rows.
- COO stores a list of (row, column, value) tuples.
- COO storage is efficient only for really sparse matrices.

COO for maximal parallelism

- list row, column and value for every non-zero entry

```
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[12] = {1, 4, 1, 2, 4, 1, 3, 4, 5, 3, 4, 5}
IA[12] = {1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5}
```

- Each thread is assigned a non-zero entry.
  - each thread computes an $A[i,j] \times x[j]$ product.
  - products can be sum with segmented reduction algorithm.
  - insensitive to row length distribution.

COO kernel

```
int element = blockIdx.x * blockDim.x + threadIdx.x;

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

To accumulate into output vecor, atomic operation are required!

- Memory footprint: $nz(val) + 2 * nz(int)$

hybrid approach

- ELL is used to handle typical entries.
- COO is used to handel exceptional entries, i.e., entries overflowing standard row size.

Hybrid kernel

```
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]];
  }
  atomicAdd (y + row, dot);
}

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);
}
```

Storage requirements

- M - number of rows in the matrix
- N - number of columns in the matrix
- K - number of nonzero entries in the densest row
- S - sparsity level [0 -1], 1 being fully-dense

Format	Storage Requirement (words)
Dense	MN
Compressed Sparse Row (CSR)	2MNS + M + 1
ELL	2MK
Coordinate (COO)	3MNS
Hybrid ELL / COO (HYB)	> 3MNS, < 2MK

Conclusion

Conclusion

- Sparse matrices are hard!
- There are a lot of ways to represent sparse matrices
- Different representations have different storage requirements
- The storage requirements depend differently on the sparsity pattern
- There is sometimes a need to safeguard against worst-case input
- There is often a trade-off between regularity and efficiency
- Some representations are better suited to certain hardware than others
- It can be difficult to achieve a high compute-to-global-memory-access ratio when it comes to sparse matrices