CUDA
Lecture 3

Manfred Liebmann
Technische Universität München
Chair of Optimal Control
Center for Mathematical Sciences, M17
manfred.liebmann@tum.de

December 21, 2015

Sparse Matrix-Vector Multiplication Kernel

Let $A \in \mathbb{R}^{N \times N}$ be a matrix in compressed row storage format and $u, b \in \mathbb{R}^N$.

- **CRS Sparse Matrix-Vector Multiplication Kernel**
  - Schedule a thread for every sparse scalar product!
  - Thread i calculates $u_i = \sum_{j=1}^{N} A_{ij} b_j$

Looks nice! Performs very badly!

Problems and Solutions

- **Non-coalesced memory access!**
  - Rearrange CRS data structure for coalesced access
  - Interleave the sparse matrix rows for at least 16 consecutive rows
  - Holes in the data structure: Not critical! Typical 5-10% increase in storage

- **Random access to b vector!**
  - Use the texture unit of the GPU for random access to b vector
  - Texturing is optimized for spacial locality: Small read-only cache

GPU-Accelerated Sparse Matrix-Vector Multiplication

An efficient sparse matrix-vector multiplication is key to the PCG-AMG solver performance.

4	0	−2	0	0	0	−1	0
−1	4	0	0	0	0	0	−1
0	0	3	−1	0	−1	0	0
0	0	0	4	0	−1	0	0
−1	0	0	0	2	0	−1	0
0	0	−1	0	0	4	0	0
−1	0	0	0	0	3	−1	0
0	0	−1	0	0	−1	0	4

Figure 1: A sample matrix with the rows colored in different hues.

Compressed Row Storage Data Format (CRS)

A flexible data format for sparse matrices.

3	3	3	2	3	2	3	3
0	3	6	9	11	14	16	19

1	3	7	1	2	8	3	4	6	4	6	1	5	7	3	6	1	6	7	3	6	8
4	−2	−1	−1	4	−1	3	−1	−1	4	−1	−1	2	−1	−1	4	−1	3	−1	−1	−1	4

Figure 2: CRS data structure for the sample matrix with the count and displacement vector on top and the column indices and matrix entries below.

Interleaved Compressed Row Storage Data Format (ICRS)

Coalesced memory access patterns on GPUs are required to achieve high performance.

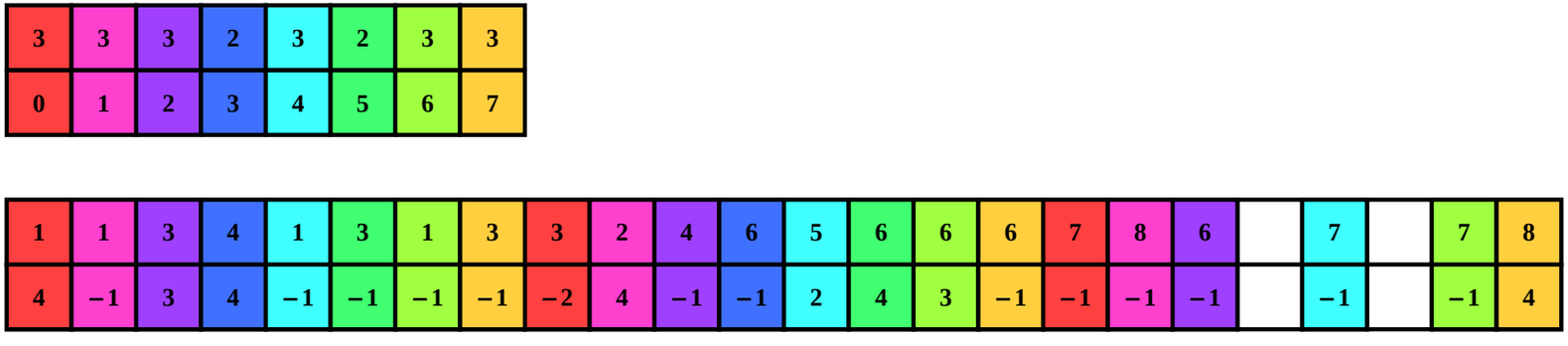

Figure 3: ICRS data structure for the sample matrix with the count and displacement vector on top and the interleaved column indices and matrix entries below. The eight interleaved matrix rows create holes in the data structure represented by the white boxes.

CUDA Code Sample for Sparse Matrix-Vector Multiplication

```
#define L 256
struct linear_operator_params {
    const int *cnt;       //ICRS count vector
    const int *dsp;       //ICRS displacement vector
    const int *col;       //ICRS column indices
    const double *ele;    //ICRS matrix entries
    const double *u;      //Input vector
    double *v;            //Output vector
    int n;                //Matrix dimension
};
texture<int2> tex_u;

void _device_linear_operator(int *cnt, int *dsp, int *col, double *ele,
    int m, int n, double *u, double *v)
{
    cudaBindTexture(0, tex_u, (int2*)u, sizeof(double) * m); //Bind the texture

    struct linear_operator_params parms;
    parms.cnt = cnt; parms.dsp = dsp; parms.col = col; parms.ele = ele;
    parms.u = u; parms.v = v; parms.n = n;
    __device_linear_operator<<< (n + N - 1)/N, N >>>(parms); //GPU kernel launch

    cudaUnbindTexture(tex_u);                                //Unbind the texture
}
```

```
__global__ void __device_linear_operator(struct linear_operator_params parms)
{
    unsigned int j = N * blockIdx.x + threadIdx.x;
    if(j < parms.n)
    {
        unsigned int blkStart = parms.dsp[j];
        unsigned int blkStop = blkStart + L * parms.cnt[j];
        double s = 0.0;
        for(unsigned int i = blkStart; i < blkStop; i += L)
        {
            unsigned int q = parms.col[i];          //Load column index
            double a = parms.ele[i];                //Load matrix entry
            int2 c = tex1Dfetch(tex_u, q);          //Load vector entry using texture mapping
            double b = __hiloint2double(c.y, c.x);  //Convert texture entries to double number
            s += a * b;                             //Calculate the sparse scalar product
        }
        parms.v[j] = s;                             //Store the sparse scalar product
    }
}
```