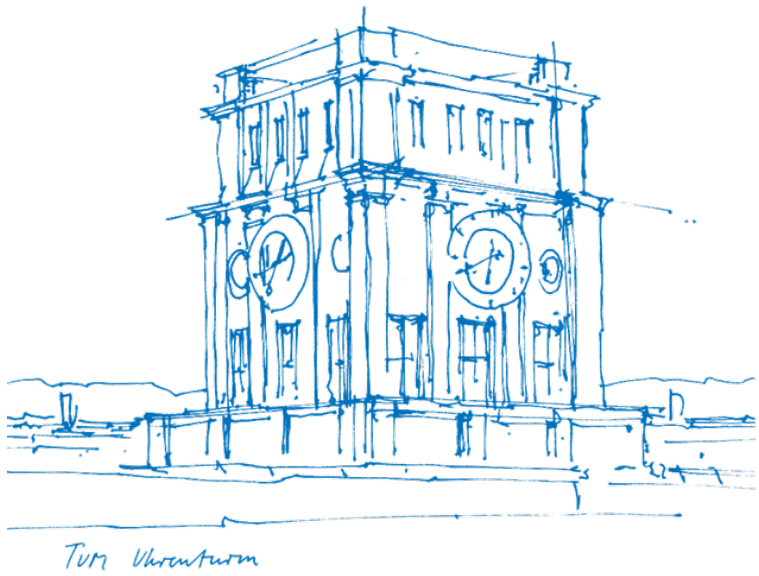
Dense Linear Algebra

HPC - Algorithms and Applications

Alexander Pöppl
Technical University of Munich
Chair of Scientific Computing

23.11.2016



Last Tutorial

Coalesced Access

- Hardware: warp dispatcher limits thread concurrency
- Warp index computation

Sparse Matrix Data Structures

- Few non-zeros in the system matrix
- Different ways to represent sparse data in matrix
- Possibly problematic in terms of memory accesses

CSR kernel - scalar

- One row per thread
- Uncoalesced memory access
- Non-uniform matrices

CSR kernel - vectorized

- One row per warp
- Coalesced access to inner loop arrays?

Table of Contents

Assignment H3.2a - Vectorized CSR (call)

Kernel call

```
//#threads = #rows * #threads per row (= N * WARP_SIZE)
dim3 grid((N * WARP_SIZE - 1)/TILE_SIZE + 1, 1, 1);
dim3 block(TILE_SIZE, 1, 1);

k_csr2_mat_vec_mm <<< grid, block >>> (...);
```

Note: grid size is > 0 if $N = 0$.

Assignment H3.2a - Vectorized CSR (setup)

Kernel body

```
__global__ void csr_matvec_v(ptr, indices, data, x, y) {
  __shared__ float vals[TILE_SIZE];

  int thread_id = TILE_SIZE * blockIdx.x + threadIdx.x;
  int warp_id = thread_id / 32;
  int lane = thread_id & (32 - 1);
  int row = warp_id;

  if (row < num_rows) {
    int row_start = ptr[row];
    int row_end = ptr[row + 1];

    //(cont.)
```

Assignment H3.2a - Vectorized CSR (loop)

```
//(cont.)

// compute running sum per thread
vals[threadIdx.x] = 0;

for (int jj = row_start + lane; jj < row_end; jj += 32) {
  vals[threadIdx.x] += data[jj] * x[indices[jj]];
}

//(cont.)
```

Assignment H3.2a - Vectorized CSR (reduction)

```
    //(cont.)

    // parallel reduction in shared memory
    for (int d = WARP_SIZE >> 1; d >= 1; d >>= 1) {
      if (lane < d) vals[threadIdx.x] += vals[threadIdx.x + d];
    }

    // first thread in a warp writes the result
    if (lane == 0) {
      y[row] += vals[threadIdx.x];
    }
  }
}
```

Assignment H3.2a - Vectorized CSR 2 (call)

Alternative kernel call

```
//#threads = #rows * #threads per row (= N * WARP_SIZE)
dim3 grid((N - 1)/TILE_SIZE + 1, 1, 1);
dim3 block(WARP_SIZE, TILE_SIZE, 1);

k_csr2_mat_vec_mm <<< grid, block >>> (...);
```

Note: grid size is > 0 if $N = 0$.

Assignment H3.2a - Vectorized CSR 2 (setup)

Alternative kernel body

```
__global__ void csr_matvec_v(ptr, indices, data, x, y) {
  __shared__ float vals[TILE_SIZE][WARP_SIZE];

  int warp_id = TILE_SIZE * blockIdx.x + threadIdx.y;
  int lane = threadIdx.x;
  int row = warp_id;

  if (row < num_rows) {
    int row_start = ptr[row];
    int row_end = ptr[row + 1];

    //(cont.)
```

Assignment H3.2a - Vectorized CSR 2 (loop)

```
//(cont.)

// compute running sum per thread
vals[threadIdx.y][lane] = 0;

for (int jj = row_start + lane; jj < row_end; jj += 32) {
  vals[threadIdx.y][lane] += data[jj] * x[indices[jj]];
}

//(cont.)
```

Assignment H3.2a - Vectorized CSR 2 (reduction)

```
    //(cont.)

    // parallel reduction in shared memory
    for (int d = WARP_SIZE >> 1; d >= 1; d >>= 1) {
      if (lane < d) vals[threadIdx.y][lane] +=
        vals[threadIdx.y][lane + d];
    }

    // first thread in a warp writes the result
    if (lane == 0) {
      y[row] += vals[threadIdx.y][0];
    }
  }
}
```

Assignment H3.2a - Vectorized CSR kernel

CMake

- Experimental CMake support for this exercise
- CMake can create Projects for different platforms
  - Makefiles
  - Visual studio projects
- To use it:
  - Windows: Use cmake GUI
  - Linux/OSX: in Project folder:

```
mkdir build; cd build; cmake ..; make
```

Assignment H3.2b - Comparison of both kernels

Measurements (GT-650M):

Matrix	CSR	Vectorized CSR
my.mtx	0.032s	0.032s
usroads.mtx	0.381s	0.580s
flickr.mtx	5.196s	5.065s

Assignment H3.2b - Comparison of both kernels

Measurements (Tesla M2090):

Matrix	CSR	Vectorized CSR
my.mtx	5.467s	5.861s
usroads.mtx	5.707s	5.769s
flickr.mtx	11.089s	11.079s

Vectorized CSR kernel - analysis

Observations:

- contiguous, fully compressed storage of `indices` and `data`

Example data:

`ptr` `[0 2 4 7 9]`

Access pattern to `indices` and `data` by row / warp ID (0-3):

`jj = row_start + lane` `[0 0 1 1 2 2 2 3 3]`

Vectorized CSR kernel - analysis

Observations:

- contiguous, fully compressed storage of `indices` and `data`
- `x` is accessed randomly

Example data:

`ptr` `[0 2 4 7 9]`

Access pattern to `indices` and `data` by row / warp ID (0-3):

`jj = row_start + lane` `[0 0 1 1 2 2 2 3 3]`

Vectorized CSR kernel - analysis

Observations:

- contiguous, fully compressed storage of `indices` and `data`
- `x` is accessed randomly
- **partially coalesced** memory access to `indices`, `data` and `vals`

Example data:

`ptr` `[0 2 4 7 9]`

Access pattern to `indices` and `data` by row / warp ID (0-3):

`jj = row_start + lane` `[0 0 1 1 2 2 2 3 3]`

Vectorized CSR kernel - analysis

Observations:

- contiguous, fully compressed storage of `indices` and `data`
- `x` is accessed randomly
- **partially coalesced** memory access to `indices`, `data` and `vals`
- Technically, memory access to `y` is **uncoalesced**, but also rare.

Example data:

`ptr` `[0 2 4 7 9]`

Access pattern to `indices` and `data` by row / warp ID (0-3):

`jj = row_start + lane` `[0 0 1 1 2 2 2 3 3]`

Vectorized CSR kernel - analysis

Observations:

- contiguous, fully compressed storage of `indices` and `data`
- `x` is accessed randomly
- **partially coalesced** memory access to `indices`, `data` and `vals`
- Technically, memory access to `y` is **uncoalesced**, but also rare.
- Non-uniform distribution of non-zeros is handled to some degree

Example data:

`ptr` `[0 2 4 7 9]`

Access pattern to `indices` and `data` by row / warp ID (0-3):

`jj = row_start + lane` `[0 0 1 1 2 2 2 3 3]`

Vectorized CSR kernel - analysis

Observations:

- contiguous, fully compressed storage of `indices` and `data`
- `x` is accessed randomly
- **partially coalesced** memory access to `indices`, `data` and `vals`
- Technically, memory access to `y` is **uncoalesced**, but also rare.
- Non-uniform distribution of non-zeros is handled to some degree
- What about diagonal matrices?

Example data:

`ptr` `[0 2 4 7 9]`

Access pattern to `indices` and `data` by row / warp ID (0-3):

`jj = row_start + lane` `[0 0 1 1 2 2 2 3 3]`

CSR - Drawbacks

Drawbacks:

- Both scalar and vectorized kernel have problems with locally deformed matrices.
  Example: n non-zeros in row 0, 0 non-zeros in row 1
- Vectorized kernel: useful only if many non-zeros per row exist.

Ideally: number of non-zeros per row constant (1 for scalar kernel, $k \cdot 32$ for vectorized kernel).

Heat Equation (1D)

$$q_t(x,t) = c \cdot q_{xx}(x,t) \qquad \text{for } x \in (0,1)$$
$$q(x,t) = 0 \qquad \text{for } x \in \{0,1\}$$

where:

- $c > 0$: heat conductivity
- $x \in [0,1], t \in \mathbb{R}_{\geq 0}$: space and time variables
- $q : [0,1] \times \mathbb{R}_{\geq 0} \to \mathbb{R}$: temperature (function over space and time)

Heat Equation (1D)

$$q_t(x,t) = c \cdot q_{xx}(x,t) \qquad \text{for } x \in (0,1)$$
$$q(x,t) = 0 \qquad \text{for } x \in \{0,1\}$$

where:

- $c > 0$: heat conductivity
- $x \in [0,1], t \in \mathbb{R}_{\geq 0}$: space and time variables
- $q : [0,1] \times \mathbb{R}_{\geq 0} \to \mathbb{R}$: temperature (function over space and time)

Finite difference discretization: $s \in \mathbb{N}$ unknowns on Cartesian grid:

$$\frac{1}{\triangle t}(\mathbf{q}_i^{(n+1)} - \mathbf{q}_i^{(n)}) = c\frac{1}{(\triangle x)^2}(\mathbf{q}_{i+1}^{(n)} - 2\mathbf{q}_i^{(n)} + \mathbf{q}_{i-1}^{(n)})$$

$$\mathbf{q}_i^{(n+1)} = \mathbf{q}_i^{(n)} + c\frac{\triangle t}{(\triangle x)^2}(\mathbf{q}_{i+1}^{(n)} - 2\mathbf{q}_i^{(n)} + \mathbf{q}_{i-1}^{(n)})$$

Boundary condition: $\mathbf{q}_0^{(n)} := 0$ and $\mathbf{q}_{s+1}^{(n)} := 0$.

Heat Equation (1D)

$$\mathbf{q}_i^{(n+1)} = \mathbf{q}_i^{(n)} + c\frac{\triangle t}{(\triangle x)^2}(\mathbf{q}_{i+1}^{(n)} - 2\mathbf{q}_i^{(n)} + \mathbf{q}_{i-1}^{(n)})$$

In matrix-vector notation this can be written as:

$$\mathbf{q}^{(n+1)} = \mathbf{q}^{(n)} + c\frac{\triangle t}{(\triangle x)^2}\mathbf{A}\,\mathbf{q}^{(n)}$$

where **A** looks like this:

$$\begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \dots \\ & & 1 & -2 & 1 & \\ & & \vdots & & & \ddots \end{pmatrix}$$

$\Rightarrow$ **A** is *sparse*, with at most 3 non-zero entries per row.

Heat Equation (1D)

Explicit Euler

Input: $\mathbf{x}^{(0)}, \mathbf{A}, c > 0, \epsilon > 0, \triangle x > 0, \triangle t > 0$
Output: $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(N)}$ with $\mathbf{A}\,\mathbf{x}^{(N)} \approx 0$ for $N \in \mathbb{N}$

$n \leftarrow 0$;
repeat
  $\mathbf{y}^{(n+1)} \leftarrow \mathbf{A}\,\mathbf{x}^{(n)}$;
  $\mathbf{x}^{(n+1)} \leftarrow \mathbf{x}^{(n)} + c\frac{\triangle t}{(\triangle x)^2}\mathbf{y}^{(n+1)}$;
  $n \leftarrow n + 1$;
until $||\mathbf{y}^{(n)}|| < \epsilon$;

Heat Equation (1D)

Explicit Euler

Input: $\mathbf{x}^{(0)}, \mathbf{A}, c > 0, \epsilon > 0, \triangle x > 0, \triangle t > 0$
Output: $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(N)}$ with $\mathbf{A}\,\mathbf{x}^{(N)} \approx 0$ for $N \in \mathbb{N}$

$n \leftarrow 0$;
repeat
  $\mathbf{y}^{(n+1)} \leftarrow \mathbf{A}\,\mathbf{x}^{(n)}$;
  $\mathbf{x}^{(n+1)} \leftarrow \mathbf{x}^{(n)} + c\frac{\triangle t}{(\triangle x)^2}\mathbf{y}^{(n+1)}$;
  $n \leftarrow n + 1$;
until $||\mathbf{y}^{(n)}|| < \epsilon$;

In order to terminate, the algorithm must fulfill the following condition:
$c\frac{\triangle t}{(\triangle x)^2} \leq \frac{1}{2}. \Rightarrow$

Heat Equation (1D)

Explicit Euler

Input: $\mathbf{x}^{(0)}, \mathbf{A}, c > 0, \epsilon > 0, \triangle x > 0$
Output: $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(N)}$ with $\mathbf{A}\,\mathbf{x}^{(N)} \approx 0$ for $N \in \mathbb{N}$

$\triangle t \leftarrow \frac{(\triangle x)^2}{2c}$;
$n \leftarrow 0$;
repeat
  $\mathbf{y}^{(n+1)} \leftarrow \mathbf{A}\,\mathbf{x}^{(n)}$;
  $\mathbf{x}^{(n+1)} \leftarrow \mathbf{x}^{(n)} + c\frac{\triangle t}{(\triangle x)^2}\mathbf{y}^{(n+1)}$;
  $n \leftarrow n + 1$;
until $||\mathbf{y}^{(n)}|| < \epsilon$;

In order to terminate, the algorithm must fulfill the following condition:
$c\frac{\triangle t}{(\triangle x)^2} \leq \frac{1}{2}$. $\Rightarrow$Choose $\triangle t = \frac{(\triangle x)^2}{2c}$.

ELLPACK

ELLPACK matrix-vector multiplication in C/C++:

```
const int N;          // number of matrix rows
const int k_max;      // max. number of nonzero columns per row
float a[N][k_max];    // array of nonzero column entries
int j[N][k_max];      // array of nonzero column indices
float x[N];           // input vector x
float y[N];           // result vector y

for(i = 0; i < N; i++) {
  y[i] = 0;
  for(k = 0; k < k_max; k++)
    y[i] += a[i][k] * x[j[i][k]];
  }
}
```

Heat Equation with cuBLAS and ELLPACK

```
__global__ void ell(float* indices, float* data,
                    float* x, float* y) {
  int row = blockDim.x * blockIdx.x + threadIdx.x;
  if (row < num_rows) {
    float dot = 0;
    for (int k = 0; k < k_max; k++) {
      int j = /*TODO*/
      float val = /*TODO*/
      if (val != 0) /*TODO*/
    }
    y[row] += dot;
  }
}
```

Task: Complete the given ELLPACK kernel and take care that access to the arrays `indices` and `data` is coalesced. You may assume that both arrays are ordered respectively.

ELLPACK (ELL) Kernel

Straightforward approach: each thread multiplies one row with the vector.

```
__global__ void ell(float* indices, float* data,
                    float* x, float* y) {
  int row = blockDim.x * blockIdx.x + threadIdx.x;
  if (row < num_rows) {
    float dot = 0;
    for (int k = 0; k < k_max; k++) {
      int j = indices[num_rows * k + row];
      float val = data[num_rows * k + row];
      if (val != 0) dot += val * x[j];
    }
    y[row] += dot;
  }
}
```

ELLPACK (ELL) Kernel

Observations:

- contiguous, partially compressed storage of `indices` and `data`

Example data:

```
indices          [0 1 0 1 1 2 2 3 * * 3 *]
data             [1 2 5 6 7 8 3 4 * * 9 *]
```

Access pattern to `indices` and `data` by row / thread ID (0-3):

```
n = 0            [0 1 2 3                 ]
n = 1            [       0 1 2 3          ]
n = 2            [             0 1 2 3]
```

ELLPACK (ELL) Kernel

Observations:

- contiguous, partially compressed storage of `indices` and `data`
- `x` is accessed randomly

Example data:

```
indices          [0 1 0 1 1 2 2 3 * * 3 *]
data             [1 2 5 6 7 8 3 4 * * 9 *]
```

Access pattern to `indices` and `data` by row / thread ID (0-3):

```
n = 0            [0 1 2 3                 ]
n = 1            [       0 1 2 3          ]
n = 2            [             0 1 2 3]
```

ELLPACK (ELL) Kernel

Observations:

- contiguous, partially compressed storage of `indices` and `data`
- `x` is accessed randomly
- **coalesced** memory access to `indices`, `data` and `y`

Example data:

```
indices          [0 1 0 1 1 2 2 3 * * 3 *]
data             [1 2 5 6 7 8 3 4 * * 9 *]
```

Access pattern to `indices` and `data` by row / thread ID (0-3):

```
n = 0            [0 1 2 3                 ]
n = 1            [       0 1 2 3          ]
n = 2            [             0 1 2 3]
```

ELLPACK (ELL) Kernel

Observations:

- contiguous, partially compressed storage of `indices` and `data`
- `x` is accessed randomly
- **coalesced** memory access to `indices`, `data` and `y`
- non-uniform distribution of non-zeros may degenerate data structure to dense matrix $\rightarrow$ Solution: hybrid formats

Example data:

```
indices          [0 1 0 1 1 2 2 3 * * 3 *]
data             [1 2 5 6 7 8 3 4 * * 9 *]
```

Access pattern to `indices` and `data` by row / thread ID (0-3):

```
n = 0            [0 1 2 3                 ]
n = 1            [       0 1 2 3          ]
n = 2            [             0 1 2 3]
```

PageRank main loop

Let's take a look at the main loop of the PageRank algorithm:

```
for (i = 1; err > EPS; i++) {
  // copy data to GPU, compute y += Bx, copy back to CPU
  CSRmatvecmult(ptr, J, Val, N, nnz, x, y, bVectorizedCSR);
  err = 0.0; sum = 0.0;
  for (j = 0; j < N; ++j) {
    newX = alpha * y[j] + (1.0 - alpha) * 1.0/N;
    err += fabs(x[j] - newX);
    x[j] = newX; y[j] = 0;
    sum += x[j];
  }
}
```

Is this efficient?

PageRank main loop

Let's take a look at the main loop of the PageRank algorithm:

```
for (i = 1; err > EPS; i++) {
  // copy data to GPU, compute y += Bx, copy back to CPU
  CSRmatvecmult(ptr, J, Val, N, nnz, x, y, bVectorizedCSR);
  err = 0.0; sum = 0.0;
  for (j = 0; j < N; ++j) {
    newX = alpha * y[j] + (1.0 - alpha) * 1.0/N;
    err += fabs(x[j] - newX);
    x[j] = newX; y[j] = 0;
    sum += x[j];
  }
}
```

Is this efficient? No, unnecessary CPU $\leftrightarrow$ GPU data transfer

PageRank main loop

Observations:

- Each time `CSRmatvecmult` is called, all data is copied back and forth between GPU and CPU.
- `alpha, err` should be available on the CPU in each iteration
- `x` and `sum` must be copied to the CPU for output only
- Everything else can remain on the GPU

Two solutions for this problem:

- Write (yet another) two kernels for the j-loop + error reduction and for the sum
- Better: Use a library, only simple linear algebra is required here

cuBLAS

https://developer.nvidia.com/cublas

- Dense linear algebra library by Nvidia
- Data types: single, double, single complex, double complex
- Helper functions: Create, SetVector, SetMatrix, ...
- Level 1: Vector operations: amax, asum, dot, axpy, nrm2, rot, ...
- Level 2: Matrix-vector operations: gemv, gbmv, symv ...
- Level 3: Matrix-matrix operations: gemm, symm, ...

cuBLAS: Helper functions

Helper functions:

- `cublasCreate(handle)` creates a cuBLAS context
- `cublasDestroy(handle)` destroys a cuBLAS context
- `cublasSetVector(n, elemSize, x, incx, y, incy)` copies **x** from host memory to **y** in device memory
- `cublasGetVector(n, elemSize, x, incx, y, incy)` copies **x** from device memory to **y** in host memory

cuBLAS: Dense Vectors

Level 1: Dense Vector operations

- `cublas<T>nrm2(handle, n, x, incx, &s)`
  computes a norm: $s = ||\mathbf{x}||$
- `cublas<T>dot(handle, n, x, incx, y, incy, &s)`
  computes a dot product: $s = \mathbf{x} \cdot \mathbf{y}$
- `cublas<T>axpy(handle, n, &alpha, x, incx, y, incy)`
  computes a weighted sum: $\mathbf{y} = \alpha\,\mathbf{x} + \mathbf{y}$

where `<T>` is a template for different data types:

character	data type
S	single
D	double
C	single complex
Z	double complex

Heat Equation with cuBLAS and ELLPACK

```
//choose something bigger than epsilon initially
err = 2.0f * epsilon;
for (i = 0; err > epsilon; ++i) {
  ELL_kernel(N, num_cols_per_row, indices_d, data_d, x_d, y_d);
  /** TODO: err = || y_d || **/
  /** TODO: x_d = x_d + dt / (dx * dx) * c * y_d **/
}
/** TODO: x = x_d **/
```

Task: Complete the given main loop of a heat equation solver using cuBLAS Level 1 instructions

Required cuBLAS instructions:

```
cublasGetVector(n, elemSize, x, incx, y, incy)
cublas<T>nrm2(handle, n, x, incx, &s)
cublas<T>axpy(handle, n, &alpha, x, incx, y, incy)
```

Heat Equation with cuBLAS and ELLPACK

```
//choose something bigger than epsilon initially
err = 2.0f * epsilon;
for (i = 0; err > epsilon; ++i) {
  ELL_kernel(N, num_cols_per_row, indices_d, data_d, x_d, y_d)
  cublasSnrm2(cublasHandle, N, y_d, 1, &err);

  float alpha = dt/(dx * dx) * c;
  cublasSaxpy(cublasHandle, N, &alpha, y_d, 1, x_d, 1);
}
cublasGetVector(N, sizeof(float), x_d, 1, x, 1)
```

cuSPARSE

https://developer.nvidia.com/cusparse.

- Sparse linear algebra library by Nvidia
- Data types: single, double, single complex, double complex
- Sparse formats: COO, CSR, HYB (ELL + COO), BSR, ...
- Helper functions: Create, CreateHybMat, ...
- Level 1: Dense and sparse vectors: axpy, gather, scatter, ...
- Level 2: Sparse matrices and dense vectors: csrmv, hybmv, bsrmv, ...
- Level 3: Sparse and dense matrices: csrmm, ...
- Preconditioners, converters, ...

cuSPARSE: Helper and conversion functions

Helper and conversion functions:

- `cusparseCreate(handle)` creates a cuSPARSE context
- `cusparseDestroy(handle)` destroys a cuSPARSE context
- `cusparseCreateMatDescr(descrA)`
  creates a matrix descriptor, defines matrix type and index base
- `cusparseCreateHybMat(hybA)`
  creates a matrix in hybrid ELLPACK + COO format
- `cusparse<T>csr2hyb(handle, m, n, descrA, csrValA, csrRowPtrA, csrColIndA, hybA, userEllWidth, partitionType)`
  converts a CSR matrix to hybrid format. For the heat equation:
  - `userEllWidth = 3`
  - `partitionType = CUSPARSE_HYB_PARTITION_MAX`.

cuSPARSE: Sparse Matrices and Dense Vectors

Level 2: Sparse Matrix and Dense Vector operations

- `cusparse<T>csrmv(handle, op, m, n, nnz, alpha, &descrA, csrValA, csrRowPtrA, csrColIndA, x, &beta, y)`
  Computes a CSR matrix-vector product $\mathbf{y} = \beta\,\mathbf{y} + \alpha\,op(\mathbf{A})\,\mathbf{x}$
- `cusparse<T>hybmv(handle, op, alpha, &descrA, hybA, x, &beta, y)`
  Computes a hybrid matrix-vector product $\mathbf{y} = \beta\,\mathbf{y} + \alpha\,op(\mathbf{A})\,\mathbf{x}$

Heat Equation with cuBLAS and cuSPARSE

Task:

- A sparse matrix in CSR format exists for the heat equation
- Convert CSR to HYB format (ELLPACK $>$ CSR for the heat equation)
- Call cuSPARSE for matrix-vector multiplication in the time step loop

$\rightarrow$ Homework!