Solving the heat equation with CUDA

Oliver Meister
December 3rd\textsuperscript{th} 2014
Last Tutorial

Coalesced Access

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

PageRank algorithm

- Few non-zeros in the system matrix
- Requires a sparse matrix - dense vector product

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

**Heat Equation**
- Problem
- Numerical solution

**ELLPACK Kernel**
- CUDA parallelization
- Performance analysis

**CUDA accelerated libraries**
- Recap: PageRank algorithm
- cuBLAS
- cuSPARSE
Assignment H3.2a - Vectorized CSR (call)

Kernel call

```c
// #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

```c
__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.)
    }
}
```
// (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

```c
__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

Some remarks:

- No need for `__syncthreads()`, only the threads of a warp have to be synchronized.
- Careful with concurrent access in the reduction.
- MAC-Cluster: Use `srun ./sparse`, otherwise the code will be executed on the login node and not the nvd node
- CUDA 3.0 (Kepler): We could actually implement the kernel without shared memory, using shuffle instructions (`__shfl_down`, `__shfl_up`, `__shfl_xor`, `__shfl`)
Assignment H3.2b - Comparison of both kernels

Measurements (NVS 5200M):

<table>
<thead>
<tr>
<th>Matrix</th>
<th>CSR</th>
<th>Vectorized CSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>my.mtx</td>
<td>0.061s</td>
<td>0.053s</td>
</tr>
<tr>
<td>usroads.mtx</td>
<td>0.228s</td>
<td>0.415s</td>
</tr>
<tr>
<td>flickr.mtx</td>
<td>6.492s</td>
<td>5.772s</td>
</tr>
</tbody>
</table>
## Assignment H3.2b - Comparison of both kernels

Measurements (Tesla M2090):

<table>
<thead>
<tr>
<th>Matrix</th>
<th>CSR</th>
<th>Vectorized CSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>my.mtx</td>
<td>5.467s</td>
<td>5.861s</td>
</tr>
<tr>
<td>usroads.mtx</td>
<td>5.707s</td>
<td>5.769s</td>
</tr>
<tr>
<td>flickr.mtx</td>
<td>11.089s</td>
<td>11.079s</td>
</tr>
</tbody>
</table>
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:

\[
\text{ptr} = [0 \ 2 \ 4 \ 7 \ 9]
\]

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

\[
jj = \text{row}_\text{start} + \text{lane} = [0 \ 0 \ 1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3]\n\]
Vectorized CSR kernel - analysis

Observations:
- contiguous, fully compressed storage of indices and data
- $x$ is accessed randomly

Example data:

ptr \[\begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}\]

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

\[jj = \text{row\_start} + \text{lane}\]
\[\begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix}\]
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

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

$jj = \text{row}\_\text{start} + \text{lane}$
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:

$$\text{ptr} \quad [0 \ 2 \ 4 \ 7 \ 9]$$

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

$$\text{j}j = \text{row}_\text{start} + \text{lane} \quad [0 \ 0 \ 1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3]$$
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:

\[
\text{ptr} = [0 \ 2 \ 4 \ 7 \ 9]
\]

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

\[
jj = \text{row}_\text{start} + \text{lane} = [0 \ 0 \ 1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3]
\]
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 $\begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$

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

$jj = \text{row}_\text{start} + \text{lane}$ $\begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix}$
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)

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

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)

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

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:

\[
\frac{1}{\triangle t} (q_i^{(n+1)} - q_i^{(n)}) = c \frac{1}{(\triangle x)^2} (q_{i+1}^{(n)} - 2q_i^{(n)} + q_{i-1}^{(n)})
\]

\[ q_i^{(n+1)} = q_i^{(n)} + c \frac{\triangle t}{(\triangle x)^2} (q_{i+1}^{(n)} - 2q_i^{(n)} + q_{i-1}^{(n)}) \]

Boundary condition: \( q_0^{(n)} := 0 \) and \( q_{s+1}^{(n)} := 0 \).
Heat Equation (1D)

\[ q^{(n+1)}_i = q^{(n)}_i + c \frac{\Delta t}{(\Delta x)^2} (q^{(n)}_{i+1} - 2q^{(n)}_i + q^{(n)}_{i-1}) \]

In matrix-vector notation this can be written as:

\[ q^{(n+1)} = q^{(n)} + c \frac{\Delta t}{(\Delta x)^2} A q^{(n)} \]

where \( A \) looks like this:

\[
\begin{pmatrix}
-2 & 1 & & \\
1 & -2 & 1 & \\
1 & -2 & 1 & \ldots \\
1 & -2 & 1 & \\
& & \vdots & \\
& & & \ddots
\end{pmatrix}
\]

\( \Rightarrow A \) is \textit{sparse}, with at most 3 non-zero entries per row.
Heat Equation (1D)

Explicit Euler

Input: \( x^{(0)}, A, c > 0, \epsilon > 0, \Delta x > 0, \Delta t > 0 \)

Output: \( x^{(0)}, \ldots, x^{(N)} \) with \( A x^{(N)} \approx 0 \) for \( N \in \mathbb{N} \)

1. \( n \leftarrow 0; \)
2. repeat
3. \hspace{1em} \( y^{(n+1)} \leftarrow A x^{(n)}; \)
4. \hspace{1em} \( x^{(n+1)} \leftarrow x^{(n)} + c \frac{\Delta t}{(\Delta x)^2} y^{(n+1)}; \)
5. \hspace{1em} \( n \leftarrow n + 1; \)
6. until \( \|y^{(n)}\| < \epsilon; \)
Heat Equation (1D)

Explicit Euler

**Input:** $x^{(0)}, A, c > 0, \epsilon > 0, \triangle x > 0, \triangle t > 0$

**Output:** $x^{(0)}, \ldots, x^{(N)}$ with $A x^{(N)} \approx 0$ for $N \in \mathbb{N}$

$n \leftarrow 0;$

repeat

\[
\begin{align*}
y^{(n+1)} & \leftarrow A \cdot x^{(n)}; \\
x^{(n+1)} & \leftarrow x^{(n)} + c \frac{\triangle t}{(\triangle x)^2} y^{(n+1)}; \\
n & \leftarrow n + 1;
\end{align*}
\]

until $\|y^{(n)}\| < \epsilon$;

In order to terminate, the algorithm must fulfill the following condition:

\[
c \frac{\triangle t}{(\triangle x)^2} \leq \frac{1}{2}. \Rightarrow \text{Choose } \triangle t = \frac{(\triangle x)^2}{2c}.
\]
Heat Equation (1D)

Explicit Euler

**Input:** $x^{(0)}, A, c > 0, \varepsilon > 0, \triangle x > 0$

**Output:** $x^{(0)}, \ldots, x^{(N)}$ with $A x^{(N)} \approx 0$ for $N \in \mathbb{N}$

\[
\triangle t \leftarrow \frac{(\triangle x)^2}{2c};
\]

\[
n \leftarrow 0;
\]

**repeat**

\[
y^{(n+1)} \leftarrow A x^{(n)};
\]

\[
x^{(n+1)} \leftarrow x^{(n)} + c \frac{\triangle t}{(\triangle x)^2} y^{(n+1)};
\]

\[
n \leftarrow n + 1;
\]

**until** $||y^{(n)}|| < \varepsilon$;

In order to terminate, the algorithm must fulfill the following condition:

\[
c \frac{\triangle t}{(\triangle x)^2} \leq \frac{1}{2}. \Rightarrow \text{Choose } \triangle t = \frac{(\triangle x)^2}{2c}.
\]
ELLPACK

ELLPACK matrix-vector multiplication in C/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]];
}
```

Oliver Meister: Solving the heat equation with CUDA
Tutorial Parallel Programming and High Performance Computing, December 3rd\textsuperscript{th} 2014
__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.

```c
__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 $\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 2 & 2 & 3 & * & * & 3 & * \end{bmatrix}$
data $\begin{bmatrix} 1 & 2 & 5 & 6 & 7 & 8 & 3 & 4 & * & * & 9 & * \end{bmatrix}$

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

n = 0 $\begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}$
n = 1 $\begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}$
n = 2 $\begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}$
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:

<table>
<thead>
<tr>
<th>indices</th>
<th>0 1 0 1 1 2 2 3 * * 3 *</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>1 2 5 6 7 8 3 4 * * 9 *</td>
</tr>
</tbody>
</table>

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:

```c
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:

```c
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 ↔ 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 $\text{sum}$ 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, ...
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: \( s = \|x\| \)

- `cublas<T>dot(handle, n, x, incx, y, incy, &s)`
  computes a dot product: \( s = x \cdot y \)

- `cublas<T>axpy(handle, n, &alpha, x, incx, y, incy)`
  computes a weighted sum: \( y = \alpha x + y \)

where \( <T> \) is a template for different data types:

<table>
<thead>
<tr>
<th>character</th>
<th>data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>single</td>
</tr>
<tr>
<td>D</td>
<td>double</td>
</tr>
<tr>
<td>C</td>
<td>single complex</td>
</tr>
<tr>
<td>Z</td>
<td>double complex</td>
</tr>
</tbody>
</table>
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


- 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 $y = \beta y + \alpha op(A)x$

- `cusparse<T>hybmv(handle, op, alpha, &descrA, hybA, x, &beta, y)`
  Computes a hybrid matrix-vector product $y = \beta y + \alpha op(A)x$
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

→ Homework!