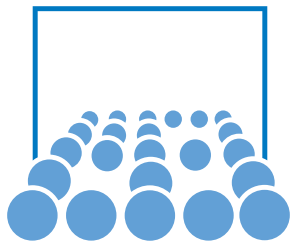
Solving the heat equation with CUDA

Oliver Meister

January 09th 2013



Last Tutorial

CSR kernel - scalar

- One row per thread
- No coalesced memory access
- Non-uniform matrices

CSR kernel - vectorized

- One row per warp
- Works well with large number ($\geq$ 16) of nonzeros per row

Table of Contents

Assignment 2 + 3

Task: Kernel call

```
if (!bVectorized) {
  //#threads = #rows (= N)
  dim3 grid((N + TILE_SIZE - 1)/TILE_SIZE, 1, 1);
  dim3 block(TILE_SIZE, 1, 1);

  k_csr_mat_vec_mm <<< grid, block >>> (...);
} else {
  //#threads = #rows * #threads per row (= N * WARP_SIZE)
  dim3 grid((N * WARP_SIZE + TILE_SIZE - 1)/TILE_SIZE, 1, 1);
  dim3 block(TILE_SIZE, 1, 1);

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

Heat Equation (1D)

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

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)

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

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 over $s \in \mathbb{N}$ unknowns on cartesian grid:

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

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

As boundary condition, we set $\mathbf{q}_0^{(n)} := \mathbf{q}_1^{(n)}$ and $\mathbf{q}_{s+1}^{(n)} := \mathbf{q}_s^{(n)}$.

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:

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

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

The algorithm terminates if $c\frac{\triangle t}{(\triangle x)^2} \leq \frac{1}{2}$. Why?

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

The algorithm terminates if $c\frac{\triangle t}{(\triangle x)^2} \leq \frac{1}{2}$. Why?

$\Rightarrow$ Choose $\triangle t = \frac{(\triangle x)^2}{2c}$.

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

The algorithm terminates if $c\frac{\triangle t}{(\triangle x)^2} \leq \frac{1}{2}$. Why?

$\Rightarrow$ Choose $\triangle t = \frac{(\triangle x)^2}{2c}$.

ELLPACK

ELLPACK matrix-vector multiplication:

```
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
float 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]];
  }
}
```

ELLPACK (ELL) Kernel

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

```
__global__ void ell_matvec(indices, data, x, 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 col = indices[num_rows * k + row];
      float val = data[num_rows * k + row];

      if (val != 0) dot += val * x[col];
    }
    y[row] += dot;
  }
}
```

ELLPACK (ELL) Kernel

Observations:

- contiguous, partially compressed storage of column and value vectors
- `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 column and value vectors
- `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 column and value vectors
- `x` is accessed randomly
- **coalesced** memory access to `indices`, `data` and `y`
- non-uniform distribution of nonzeros may degenerate data structure to dense matrix

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 column and value vectors
- `x` is accessed randomly
- **coalesced** memory access to `indices`, `data` and `y`
- non-uniform distribution of nonzeros 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]
```

cuBLAS

https://developer.nvidia.com/cublas

- Dense linear algebra library by Nvidia
- Data types: single, double, single complex, double complex
- 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, ...

Heat Equation with cuBLAS and ELLPACK

```
int* indices = (int*)calloc(N*num_cols_per_row, sizeof(int));
float* data = (float*)calloc(N*num_cols_per_row, sizeof(float));

// fill matrix with stencil [1 -2 1]
for (i = 1; i < N-1; i++) {
  data[N * 0 + i] = 1;      indices[N * 0 + i] = i-1;
  data[N * 1 + i] = -2;     indices[N * 1 + i] = i;
  data[N * 2 + i] = 1;      indices[N * 2 + i] = i+1;
}

// first and last line (Outflow condition)
data  [N * 1 + 0] = -1;     indices[N * 1 + 0] = 0;
data  [N * 2 + 0] = 1;      indices[N * 2 + 0] = 1;

data  [N * 0 + N - 1] = 1;  indices[N * 0 + N - 1] = N-2;
data  [N * 1 + N - 1] = -1; indices[N * 1 + N - 1] = N-1;
```

Heat Equation with cuBLAS and ELLPACK

```
//(...) Copy indices, data, x, y to device memory

//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);

  alpha = dt/(dx * dx) * c;
  cublasSaxpy(cublasHandle, N, &alpha, y_d, 1, x_d, 1);
}

cudaMemcpy(x, x_d, N * sizeof(float), cudaMemcpyDeviceToHost);
```

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, ...
- 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, ...

Heat Equation with cuBLAS and cuSPARSE

Steps:

- Create sparse matrix in CSR format
- Convert to HYB format using:

```
cusparseCreateHybMat(&hyb_d);
cusparseScsr2hyb(cusparseHandle, N, N, descr,
  data_d, start_d, indices_d,
  hyb_d, 0, CUSPARSE_HYB_PARTITION_MAX);
```

- Iterate over time steps and call `cusparseShybmv` for matrix-vector multiplication

Heat Equation with cuBLAS and cuSPARSE

```
//choose something bigger than epsilon initially
err = 2.0f * epsilon;

for (i = 0; err > epsilon; ++i) {
  alpha = 1.0f; beta = 0.0f;
  cusparseShybmv(cusparseHandle,
    CUSPARSE_OPERATION_NON_TRANSPOSE,
    &alpha, descr, hyb_d, x_d, &beta, y_d);

  cublasSnrm2(cublasHandle, N, y_d, 1, &err);

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

Assignment

1. Read the paper by Bell and Garland: click me
2. Try to answer the following questions:
   - Why does HYB usually perform best in Nvidia's examples? What are its drawbacks?
   - For which problem types and sizes can you expect good performance from using CUDA for Sparse LA?
   - In general, is it safe to assume that sparse linear systems can be solved faster on GPUs than on CPUs?

Literature

Nathan Bell and Michael Garland
Efficient Sparse Matrix-Vector Multiplication on CUDA.
2008.