CUDA Kernels for SpMV

Oliver Meister

January 09th 2013

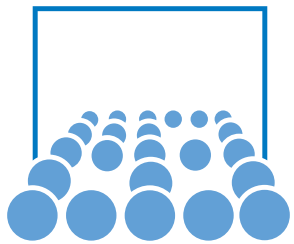

Table of Contents

Last Tutorial

Optimization of the SWE code

- Performance analysis
- Using shared memory and parallel reduction for speedup
- Further optimization technique: kernel fusion

Further extensions to SWE

- Hybrid parallelization
- ASAGI, OpenGL visualization

Assignment 1

Task: Coalesced memory access

- Switch indices:

```
l_cellIndexI = blockDim.y * blockIdx.x + threadIdx.y + 1;
l_cellIndexJ = blockDim.x * blockIdx.y + threadIdx.x + 1;
```

- Adapt block size:

```
dim3 dimBlock(TILE_SIZE, TILE_SIZE);
dim3 dimGrid(nx/TILE_SIZE,ny/TILE_SIZE);

dim3 dimRightBlock(TILE_SIZE, 1);
dim3 dimRightGrid(1, ny/TILE_SIZE);

dim3 dimTopBlock(1, TILE_SIZE);
dim3 dimTopGrid(nx/TILE_SIZE, 1);
```

Assignment 2

Task: Access state vector via shared memory

- Load state vector for each cell (hu, hv and b likewise)

```
s_h[threadIdx.y + 1][threadIdx.x + 1] =
  i_h[l_rightCellPosition];

if (threadIdx.x == 0) {
  l_leftCellPosition = computeOneDPositionKernel(
    l_cellIndexI, l_cellIndexJ-1, i_nY+2);
  s_h[threadIdx.y + 1][0] = i_h[l_leftCellPosition];
}

if (threadIdx.y == 0) {
  l_leftCellPosition = computeOneDPositionKernel(
    l_cellIndexI-1, l_cellIndexJ, i_nY+2);
  s_h[0][threadIdx.x + 1] = i_h[l_leftCellPosition];
}
```

Assignment 2

Task: Access state vector via shared memory

- Compute horizontal net updates

```
fWaveComputeNetUpdates(
  9.81,
  s_h   [threadIdx.y]     [threadIdx.x + 1],
  s_h   [threadIdx.y + 1] [threadIdx.x + 1],
  s_hu  [threadIdx.y]     [threadIdx.x + 1],
  s_hu  [threadIdx.y + 1] [threadIdx.x + 1],
  s_b   [threadIdx.y]     [threadIdx.x + 1],
  s_b   [threadIdx.y + 1] [threadIdx.x + 1],
  l_netUpdates
);
```

Assignment 2

Task: Access state vector via shared memory

- Compute vertical net updates

```
fWaveComputeNetUpdates(
  9.81,
  s_h   [threadIdx.y + 1] [threadIdx.x],
  s_h   [threadIdx.y + 1] [threadIdx.x + 1],
  s_hv  [threadIdx.y + 1] [threadIdx.x],
  s_hv  [threadIdx.y + 1] [threadIdx.x + 1],
  s_b   [threadIdx.y + 1] [threadIdx.x],
  s_b   [threadIdx.y + 1] [threadIdx.x + 1],
  l_netUpdates
);
```

Compressed Sparse Row (CSR)

CSR matrix-vector multiplication:

```
const int N;       // number of matrix rows
const int K;       // number of nonzero matrix entries
float a[K];        // array of nonzero matrix entries
float j[K];        // array of column indices
float start[N+1]; // array of row start indices
float x[N];        // input vector x
float y[N];        // result vector y

for(int i = 0; i < N; i++) {
  y[i] = 0;

  for(k = start[i]; k < start[i + 1]; k++) {
    y[i] += a[k] * x[j[k]];
  }
}
```

Compressed Sparse Row (CSR) Kernel 1

First straightforward approach: each thread does a row $\times$ vector multiplication

```
__global__ void csr_matvec_s(ptr, indices, data, x, y) {
  int row = blockDim.x * blockIdx.x + threadIdx.x ;
  if (row < num_rows) {
    float dot = 0;
    int row_start = ptr[row];
    int row_end = ptr[row + 1];

    for (int jj = row_start; jj < row_end; jj++) {
      dot += data[jj] * x[indices[jj]];
    }

    y[row] += dot;
  }
}
```

Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:

- contiguous, fully compressed storage of column and value vectors

Example data:

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

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

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

Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:

- contiguous, fully compressed storage of column and value vectors
- `x` is accessed randomly

Example data:

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

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

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

Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:

- contiguous, fully compressed storage of column and value vectors
- `x` is accessed randomly
- **non-coalesced** memory access to `indices` and `data`, **coalesced** access to `y`

Example data:

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

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

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

Compressed Sparse Row (CSR) Kernel 1 (cont.)

Observations:

- contiguous, fully compressed storage of column and value vectors
- `x` is accessed randomly
- **non-coalesced** memory access to `indices` and `data`, **coalesced** access to `y`
- non-uniform distribution of nonzeros may lead to serialization

Example data:

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

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

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

Compressed Sparse Row (CSR) Kernel 2

Idea: each **warp** does a row $\times$ vector multiplication.

Requires the following steps:

- For each block: allocate a shared array `vals[]` for the results
- For each thread in a warp: find warp id (`row`) and warp-local thread-id (`lane`)
- For each thread in a warp: do CSR loop for row `row` with offset `lane` and stride `WARP_SIZE`
- For each warp: reduce `vals[]` to entry `y[row]`

Compressed Sparse Row (CSR) Kernel 2 - setup

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

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

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

    //(cont.)
```

Compressed Sparse Row (CSR) Kernel 2 - loop

```
//(cont.)

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

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

//(cont.)
```

Compressed Sparse Row (CSR) Kernel 2 - reduce

```
    //(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];
    }
  }
}
```

Compressed Sparse Row (CSR) Kernel 2 (cont.)

Observations:

- contiguous, fully compressed storage of column and value vectors

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]`

Compressed Sparse Row (CSR) Kernel 2 (cont.)

Observations:

- contiguous, fully compressed storage of column and value vectors
- `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]`

Compressed Sparse Row (CSR) Kernel 2 (cont.)

Observations:

- contiguous, fully compressed storage of column and value vectors
- `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]`

Compressed Sparse Row (CSR) Kernel 2 (cont.)

Observations:

- contiguous, fully compressed storage of column and value vectors
- `x` is accessed randomly
- **partially coalesced** memory access to `indices`, `data` and `vals`
- non-coalesced (but also rare) memory access to `y`

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]`

Compressed Sparse Row (CSR) Kernel 2 (cont.)

Observations:

- contiguous, fully compressed storage of column and value vectors
- `x` is accessed randomly
- **partially coalesced** memory access to `indices`, `data` and `vals`
- non-coalesced (but also rare) memory access to `y`
- non-uniform distribution of nonzeros 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]`

Compressed Sparse Row (CSR) Kernel 2 (cont.)

Observations:

- contiguous, fully compressed storage of column and value vectors
- `x` is accessed randomly
- **partially coalesced** memory access to `indices`, `data` and `vals`
- non-coalesced (but also rare) memory access to `y`
- non-uniform distribution of nonzeros 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]`

Assignment: PageRank

PageRank algorithm:

Let $\mathbf{B} \in \mathbb{R}^{n \times n}$ be non-negative, $\alpha \in (0, 1)$. We assume **B** is left stochastic - otherwise apply step a of the algorithm.
Goal: Find $\mathbf{x} \in [0, 1]^n$ with $\mathbf{x} = \mathbf{Bx}$.

a if **B** is not left stochastic: divide each entry in **B** by its column sum. If a 0-column exists, abort. Otherwise set **B** to the resulting matrix.

b initialize solution vector: $\mathbf{x} \leftarrow \frac{1}{n}\mathbf{e}$, where $\mathbf{e} = (1, 1, 1, ...)^T$

c multiply matrix with vector: $\mathbf{y} \leftarrow \mathbf{Bx}$

d regularize: $\mathbf{x} \leftarrow \alpha\mathbf{y} + (1 - \alpha)\frac{1}{n}\mathbf{e}$

e while error criterion is not fulfilled, back to step c

Assignment: PageRank

Assignments:

1. load M (matrix market format) matrix from disk (e.g. *flickr.mtx*)
   `http://www.cise.ufl.edu/research/sparse/MM/Gleich/flickr.tar.gz`
2. implement scalar CSR kernel and try it on a small matrix first (*my.mtx*) next, try with bigger matrices
3. implement vectorized CSR kernel, compare performance results for different matrices. What do you observe?

Literature

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