Research Paper Presentation #1
~
Michael M. Folkerts
CSE 262
Spring 2011
JW Choi, A Sing, RW Vuduc

Model-Driven Autotuning of Sparse Matrix-Vector Multiply on GPUs

Proc 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP 2010), pp 115-126
Introduction

- Motivation for GPU study
  - SpMV multiplication is memory bandwidth intensive
  - GPUs have significantly higher memory bandwidth than CPUs

- Obvious Issues
  - SpMV is irregular in memory access patterns
  - SpMV requires extra (memory) arrays to track indices
Compressed Sparse Row (CSR)
- \( m \times n \) sparse matrix with \( k \) non-zero entries
- Stored in 3 arrays:
  - \( \text{val}[k] \) and \( \text{col\_ind}[k] \) store and index non-zeros
  - \( \text{row\_ptr}[m+1] \) store starting pointer to each row

Blocked CSR (BCSR)
- Sub-divide matrix into \( r \times c \) blocks
  - Pad with zeros as needed
  - Reduces column index storage by at least \( 1/(r^*c) \)
  - Only one index per block
BCSR
BCSR SpMV Implementation

Extends 'state of the art' CSR SpMV [3]

- Thread handles row of matrix sub-blocks

- Short Vector Packing
  - Load sub-block as 'float4'

- Row Alignment
  - Pad short sub-block rows with zeros

**Algorithm 1: 2×2 BCSR kernel to compute \( y \leftarrow y + A \cdot x \)**

**Input:** \( m \times n \) matrix \( A \), stored in BCSR \((r \times c)\) format as \( (bval, col_bind, row_bptr) \);
  - vectors \( x \) as \( x[1 \ldots n] \), \( y \) as \( y[1 \ldots m] \)

**Output:** Modifies \( y \)
  - Let \( TB \) = thread block size (1-D)
  - Let \( tid \) = local thread ID
  - Initialize \( sdata[TB][2] \)

1. for each block row, \( I \) do
   2. \( \text{row.start} = \text{row.bptr}[I] \)
   3. \( \text{row.end} = \text{row.bptr}[I + 1] \)
   4. for \( k = \text{row.bptr}[I]; k < \text{row.bptr}[I + 1]; k = k + TB \) do
      5. for each thread do
         6. \( j_0 = \text{col.bind}[k + \text{tid}] \)
         7. \( \text{float4} \ \text{tmp} = bval[k + \text{tid}] \)
         8. \( sdata[\text{tid}][0] += \text{tmp.x} \times x[j_0] + \text{tmp.y} \times x[j_0 + 1] \)
         9. \( sdata[\text{tid}][1] += \text{tmp.z} \times x[j_0] + \text{tmp.w} \times x[j_0 + 1] \)

10. parallel reduction in shared memory to \( sdata[0][0] \)
11. parallel reduction in shared memory to \( sdata[0][1] \)
12. \( Y[I \times 2] += sdata[0][0] \)
13. \( Y[I \times 2 + 1] += sdata[0][1] \)
BCSR SpMV Results

![Graph showing BCSR SpMV Results](image)
ELLPACK and BELLPACK

- **ELLPACK**
  - m x n sparse matrix w/ k non-zero entries
  - Stored in 2 arrays:
    - V[m][L] and J[m][L] store and index non-zeros
    - L is maximum number of non-zeros in a row

- **Blocked ELLPACK (BELLPACK)**
  - Sub-divide matrix into r x c blocks
    - Pad with zeros as needed
  - Resort rows - descending order of number of blocks
  - Group block rows in sets of size R
BELLPACK Summary

(a) Construction.
### BELLPACK Implementation

<table>
<thead>
<tr>
<th>Algorithm 2: $2 \times 2$ BELLPACK kernel to compute $y \leftarrow y + A \cdot x$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> array of arrays <code>Ad.values</code> for values and <code>Ad.index</code> for indices, array <code>Ad.numBlocks</code> for block sizes, vector $x$ as $x[1 \ldots n]$, and blocking sizes $r$ and $c$</td>
</tr>
<tr>
<td><strong>Output:</strong> Modifies $y$</td>
</tr>
<tr>
<td>Let $bid =$ block ID</td>
</tr>
<tr>
<td>Let $tid =$ local thread ID</td>
</tr>
<tr>
<td>Let $TB =$ thread block size (1-D)</td>
</tr>
<tr>
<td>1. if $tid == 0$ then</td>
</tr>
<tr>
<td>2. $numBlocks = Ad.numBlocks[bid]$</td>
</tr>
<tr>
<td>3. $A.values = Ad.values[bid]$</td>
</tr>
<tr>
<td>4. $A.index = Ad.index[bid]$</td>
</tr>
<tr>
<td>5. for $i = 0$ to $numBlocks$ do</td>
</tr>
<tr>
<td>6. $col.Index = A.index[TB \times i + tid] \times c$</td>
</tr>
<tr>
<td>7. $xval = x[col.Index]$</td>
</tr>
<tr>
<td>8. $rs.1 += A.values[TB \times 4 \times i + tid] \times xval$</td>
</tr>
<tr>
<td>9. $rs.2 += A.values[TB \times 4 \times i + TB \times 2 + tid] \times xval$</td>
</tr>
<tr>
<td>10. $xval = X[col.Index++]$</td>
</tr>
<tr>
<td>11. $rs.1 += A.values[TB \times 4 \times i + TB \times 1 + tid] \times xval$</td>
</tr>
<tr>
<td>12. $rs.2 += A.values[TB \times 4 \times i + TB \times 3 + tid] \times xval$</td>
</tr>
<tr>
<td>13. $y[TB \times bid \times 2 + tid \times 2] = rs.1$</td>
</tr>
<tr>
<td>14. $y[TB \times bid \times 2 + tid \times 2 + 1] = rs.2$</td>
</tr>
</tbody>
</table>

- Each thread handles row in $R \times L$ sub-matrix (stored as column)
- $X$ is stored as a texture and is therefore cached on chip
BELLPACK Detail
Auto-tuning Model

- **Generic GPU Model**
  - For a workload $L$ (thread blocks): $M$ (# of SMs) process $T$ (thread blocks) for $I$ (iterations)
    - $I = \text{ceil}(L / (T * M))$
  - Time for each iteration (assume homogeneity):
    
    $t_i(k) = \sigma_i + \alpha_i \times (k - 1), \quad (3)$

    
    \[
    t = \tau_1 + (I - 2) \times \tau + \tau_I \quad (4)
    \]
    \[
    \tau_1 = \sigma_1 + \alpha \times (T - 1) \quad (5)
    \]
    \[
    \tau = \sigma + \alpha \times (T - 1) \quad (6)
    \]
    \[
    \tau_I = \sigma + \alpha \times \left[ \frac{(L \mod (T \times M)) - 1}{M} \right] \quad (7)
    \]
Thread Occupancy

- Computing $T_{\text{max}}$ (threads blocks per SM)
  - Computed using hardware specific properties
    - Register Memory
    - Shared Memory
    - Max # thread blocks/SM
    - ...
  - Exactly like 'occupancyCalculator.xls' provided by NVIDIA
GPU Model

\[ L = I \text{ iterations} \]

\[ \sigma(N_2) \quad \alpha(N_2) \times (T-1) \]

\[ \text{second iteration} \]

\[ \text{third iteration} \]

\[ \text{last iteration} \]

\[ \text{first iteration} \]

\[ \sigma_{l,ovhd} \quad \gamma \times N_2 \quad \sigma_1(N_2) \quad \alpha(N_2) \times (T-1) \quad \tau_1(N_2) \quad \tau(N_2) \quad \alpha(N_2) \times \left[ (L \mod (T \times M))^{-1}/M \right] \]

\[ \text{Number of Thread Blocks} \]