cI0SKI: An OpenCL Spmv Framework on GPU Platforms

Bor-Yiing Su, subrian@eecs.berkeley.edu

Parallel Computing Lab,
University of California, Berkeley
Outline

- Introduction
  - OpenCL
  - OSKI and pOSKI
  - clOSKI
- Sparse Matrix Format
- Ultimate Hybrid (UlHy) Sparse Matrix Format
- clOSKI Framework
- Experimental Results
- Conclusion
OpenCL – Open Computing Language
Open standard for portable programming of heterogeneous platforms (CPUs, GPUs, and other processors)

CPU:
- Multiple cores driving performance increases
- Multi-processor programming – e.g. OpenMP

GPU:
- Increasingly general data-parallel computing
- Graphics APIs and Shading Languages

OpenCL: Heterogeneous Computing

Slide Borrowed from Tim Mattson
OpenCL Working Group

Slide Borrowed from Tim Mattson
OpenCL Programming Model on GPUs

- Data Parallel
  - Exactly the same as the CUDA programming model
    - OpenCL: Each work group is independently scheduled on available computation units
    - CUDA: Each thread block is independently scheduled on available computation units
    - OpenCL: Each work item within a work group is mapped to an execution unit on the computation unit
    - CUDA: Each thread within a thread block is mapped to a SIMD lane

Picture Borrowed from Tim Mattson
OSKI

- The Optimized Sparse Kernel Interface (OSKI) Library is a collection of low-level C primitives that provide automatically tuned computational kernels on sparse matrices, for use in solver libraries and applications.
- Explored Design Space: Register blocking, loop unrolling
- Contributor: Rich Vuduc, James Demmel, and Kethy Yelick and others
- Targeting platform: Single core CPU

pOSKI

- The parallel OSKI
- Explored Design Space: Register blocking, loop unrolling, SIMD, multi-thread, ...
- Contributor: Jong-Ho Byun, Richard Lin, Kathy Yelick, James Demmel, and others
- Targeting platform: Multi-core CPU
cIOSKI

- The OpenCL version of OSKI
- Targeting platforms: Many-core GPUs, such as Nvidia GPUs, ATI GPUs, Intel GPUs, mobile GPUs
  - As long as the GPU vendors release their OpenCL implementations
- Current progress: Now it only supports sparse matrix vector multiplication
Outline

- Introduction
- Sparse Matrix Format
  - Sparse Matrix Vector Multiplication
  - Diagonal formats
  - Flat formats
  - Block formats
- Ultimate Hybrid (UlHy) Sparse Matrix Format
- cIOSKI Framework
- Experimental Results
- Conclusion
In a sparse matrix with dimension $n \times m$, $k$ non-zero entries

- Total flops: $2k$
- Least memory footprint: $k$ floats from the matrix + $m$ floats from the vector
  - In many cases row/column indexes of the non-zero entries are required
  - Flop/byte ratio: $2k/((k+m) \times (4 \text{ bytes per float})) < 0.5$
- Memory bounded computation
- Optimizing the sparse matrix representation is the key

![Sparse Matrix Representation](image)
DIA Format

- Extract diagonals form the sparse matrix
  - Memory footprint of a $m \times m$ matrix with $d$ diagonals: $Ω(k)$ storage
    - Offset: $d$ integers
    - Data: $d \times m$ floats
  - Parallelization strategy: use one work item to process one matrix row
- Pros
  - Does not need any column/row indices for the non-zeros
  - Single-stride access on the matrix data
  - Single-stride access on the multiplied vector
- Cons
  - Should be restricted to dense diagonals

Data = 
\[
\begin{bmatrix}
3 & 7 & 0 & 0 \\
0 & 4 & 8 & 0 \\
1 & 0 & 5 & 9 \\
0 & 2 & 0 & 6 \\
\end{bmatrix}
\]

Offset = [-2, 0, 1]

Data = [0 0 1 2, 3 4 5 6, 7 8 9 0]

Work items
BDIA Format

- Extract banded diagonals form the sparse matrix
  - Memory footprint of a $m \times m$ matrix with $b$ bands, $d$ diagonals: $\Omega(k)$ storage
    - Offset: $b$ integers
    - Band pointer: $b+1$ integers
    - Data: $d \times m$ floats
  - Parallelization strategy: use one work item to process one matrix row
- Pros
  - Does not need any column/row indices for the non-zeros
  - Single-stride access on the matrix data
  - Single-stride access on the multiplied vector
  - Can use local memory to cache vector segments used in a band
- Cons
  - Should be restricted to dense banded diagonals

Data = 
```
[0 0 1 2,
 3 4 5 6,
 7 8 9 0]
```
Offset = [-2, 0]
Band_ptr = [0, 1, 3]
ELLPack Format

- Push all the non-zeros towards left
  - Memory footprint of a mxn matrix with ell width e: \( \Omega(2k) \) storage
    - Column indices: e*m integers
    - Data: e*m floats
  - Parallelization strategy: use one work item to process one matrix row
- Pros
  - Single-stride access on the matrix data and column indices
- Cons
  - Random access on the multiplied vector
  - Should be restricted to matrices with uniform number of non-zeros per row

Data =
\[
\begin{bmatrix}
3 & 7 & 0 & 0 \\
0 & 4 & 8 & 0 \\
1 & 0 & 5 & 9 \\
0 & 2 & 0 & 6
\end{bmatrix}
\]

Col =
\[
\begin{bmatrix}
[3 7 0, \\
4 8 0, \\
1 5 9, \\
2 6 0]
\end{bmatrix}
\]

Work items
Sliced ELL Format

- Using different ell width for different slices
  - Memory footprint of a \( mxn \) matrix with \( s \) slices: \( \Omega(2k+(s+1)) \) storage
    - Slice pointers: \( s+1 \) integers
    - Column indices: \( \Omega(k) \) integers
    - Data: \( \Omega(k) \) floats
  - Parallelization strategy: use one work item to process one matrix row
- Pros
  - Single-stride access on the matrix data and column indices
  - Might have less zero paddings compared against ELL
- Cons
  - Random access on the multiplied vector
  - Non-zeros per row should be uniform within a slice

\[
\begin{array}{c}
\text{Slice 1} \\
\begin{bmatrix}
3 & 7 & 0 & 0 \\
0 & 4 & 8 & 0 \\
1 & 0 & 5 & 9 \\
0 & 2 & 0 & 6
\end{bmatrix}
\end{array}
\begin{array}{c}
\text{Slice 2} \\
\begin{bmatrix}
3 & 7 & 0 & 0 \\
0 & 4 & 8 & 0 \\
1 & 0 & 5 & 9 \\
0 & 2 & 0 & 6
\end{bmatrix}
\end{array}
\]

\[
\begin{array}{c}
\text{Slice_ptr} = [0, 4, 10] \\
\text{Data} = [3 7, 4 8, 1 5 9, 2 6 0] \\
\text{Col} = [0 1, 1 2, 0 2 3, 2 4 0]
\end{array}
\]

Work items
**CSR Format**

- Compressed sparse row format
  - Memory footprint of a $m \times n$ matrix: $2k + (m+1)$ storage
    - Row pointers: $m+1$ integers
    - Column indices: $k$ integers
    - Data: $k$ floats
  - Parallelization strategy: use multiple work items to process one row
- Pros
  - No zero paddings
- Cons
  - Random access on the multiplied vector
  - Might have unaligned access on the matrix
  - Might have load balance problem

Data =

```
3 7 0 0
0 4 8 0
1 0 5 9
0 2 0 6
```

Col =

```
[3 7, 4 8, 1 5 9, 2 6]
[0 1, 1 2, 0 2 3, 2 4]
```
**COO Format**

- **Coordinate format**
  - Memory footprint of a mxn matrix: 3k storage
    - Row indices: k integers
    - Column indices: k integers
    - Data: k floats
  - Parallelization strategy: flatten the matrix, and perform segmented reduction
- **Pros**
  - No zero paddings
  - Introduce steady performance regardless of the matrix structures
- **Cons**
  - Random access on the multiplied vector
  - Has the worst storage requirement

\[
\begin{bmatrix}
3 & 7 & 0 & 0 \\
0 & 4 & 8 & 0 \\
1 & 0 & 5 & 9 \\
0 & 2 & 0 & 6
\end{bmatrix}
\]

Row = [0 0 1 1 2 2 2 3 3]
Col = [0 1 1 2 0 2 3 1 3]
Data = [3 7 4 8 1 5 9 2 6]
Blocked ELL Format

- Extract dense blocks and represent using ELL format
  - Memory footprint of a $mxn$ matrix with block size $c$: $\Omega(k + k/c)$ storage
    - Column indices: $\Omega(k/c)$ integers
    - Data: $\Omega(k)$ floats
  - Block size: 1x4, 2x4, 4x4, 8x4, 1x8, 2x8, 4x8, 8x8
    - Using the vector format float4 to store the blocks
      - In OpenCL API, the texture memory interface is aligned with float4
      - For Nvidia GPUs, memory bandwidth of float4 and float arrays are similar
      - For ATI GPUs, memory bandwidth of float4 arrays is higher than float arrays
  - Parallelization strategy: use one work item to process one block row
  - Pros
    - Single-stride access on the matrix data
    - Can cache the multiplied vector into register as long as the block has height > 1
  - Cons
    - Should be restricted on matrices with uniform number of blocks per row

Data = \[0 1 2 3, 8 9 a b, 4 5 6 7, c d e f, g h i j, o p q r, k l m n, s t u v\]

Col = \[0 0 , 4 4\]
Using different blocked ell width for different slices

- Memory footprint of a mxn matrix with block size c and s slices: $\Omega(k + k/c)$ storage
  - Slice pointers: $s+1$ integers
  - Column indices: $\Omega(k/c)$ integers
  - Data: $\Omega(k)$ floats
- Block size: 1x4, 2x4, 4x4, 8x4, 1x8, 2x8, 4x8, 8x8
- Parallelization strategy: use one work item to process one block row
- Pros
  - Single-stride access on the matrix data
  - Can cache the multiplied vector into register as long as the block has height > 1
  - Might have less zero paddings compare against BELL
- Cons
  - Should be restricted on matrices with uniform number of blocks per row
  - Non-zeros per row should be uniform within a slice

Data = [0 1 2 3, 8 9 a b, 4 5 6 7, c d e f, g h i j, o p q r, k l m n, s t u v]
Col = [0 0 , 4 4]
Slice_ptr = [0 2 4]
Blocked CSR Format

- Extract dense blocks and represent using CSR format
  - Memory footprint of a $mxn$ matrix with block $wxh=c$: $\Omega(k + k/c) + m/h+1$ storage
    - Row pointers: $m/h + 1$ integers
    - Column indices: $\Omega(k/c)$ integers
    - Data: $\Omega(k)$ floats
  - Block size: 1x4, 2x4, 4x4, 8x4, 1x8, 2x8, 4x8, 8x8
  - Parallelization strategy: use multiple work items to process one block row
  - Pros
    - Can cache the multiplied vector into register as long as the block has height $> 1$
  - Cons
    - Might have unaligned access on the matrix
    - Might have load balance problem

Data = [0 1 2 3, g h i j; 4 5 6 7, k l m n; 8 9 a b, o p q r, 4 5 6 7, k l m n; c d e f, s t u v]

Col  = [0 4 ; 0 4]
Row_ptr = [0 2 4]
Outline

- Introduction
- Sparse Matrix Format
- Ultimate Hybrid (UlHy) Sparse Matrix Format
- cIOSKI Framework
- Experimental Results
- Conclusion
Matrix Splitting Ideas

- Splitting the matrix \( A \) into \( p \) submatrices: 
  \[ A = A_1 + A_2 + \ldots + A_p \]

- Matrix splitting efforts
  - **OSKI:**
    - Splitting the matrix \( A \) into 2 to 4 submatrices, different submatrix can have different block size
  - **Cusp:** a library for sparse linear algebra and graph computations on CUDA
    - Splitting the matrix \( A \) into 2 submatrices, one using ELL representation, remaining using COO format
Ultimate Hybrid (UlHy) Matrix Format

- Every specialized region on the sparse matrix deserve its own specialized representation
- Hybrid everything together
- Partition the original sparse matrix into many submatrices
  - Diagonal matrices
    - BDIA
    - DIA
  - Blocked matrices
    - Slice Blocked ELL (SBELL)
    - Blocked ELL (BELL)
    - Blocked CSR (BCSR)
  - Flat matrices
    - Sliced ELL (SELL)
    - ELL
    - CSR
    - COO
Design Space of the UIHy Format

- Extraction priority of the 9 different formats
- Diagonal based formats
  - How to choose between BDIA and DIA?
  - How to find the definition of “dense diagonals”?
- Block based formats
  - How to choose between BELL and SBELL?
  - When should we choose BCSR instead?
  - What is the preferred block size?
  - How to find the definition of “dense blocks”?
- Flat formats
  - How to choose between SELL and ELL?
  - How to choose between CSR and COO?
Outline

- Introduction
- Sparse Matrix Format
- Ultimate Hybrid (UlHy) Sparse Matrix Format
- cIOSKI Framework
- Experimental Results
- Conclusion
Two stages
- Offline Benchmarking
  - Understand the performance of each different formats under different settings
- Online Decision Making
  - Analyze the input sparse matrix at runtime
  - Trade-off among all the formats, and come up with a recommended hybrid representation across all available formats
Offline Benchmarking

- **Overhead**
  - Major overhead: kernel launch overhead
  - Minor overhead: wasted memory bandwidth (need to read and write the result vector multiple times)
  - Measured by launch empty kernels with different work group numbers and different work group sizes

- **Diagonal performance**
  - Measured on banded matrices with different matrix dimensions and band widths

- **Blocked performance**
  - Measured on banded blocked matrices with different matrix dimensions, different block numbers per row, and different block sizes

- **Flat performance**
  - Measured on banded matrices with different matrix dimensions and different band widths
Extraction priority decided by the best flop each category can achieve given the matrix dimension and the average non-zeros per row.
Online Decision Making

- DIA vs BDIA
  - Extract diagonals when the expected performance is better than any flat formats
  - Extract BDIA only if the performance is better than DIA
- SBELL vs BELL vs BCSR
  - Count the number of dense blocks of each different block size, choose the block size with the highest expected performance
  - Compute the zero paddings each format is needed, and choose the one that delivers the highest performance
- SELL vs ELL
  - Compute the zero paddings of each format, and choose the better one
- CSR vs COO
  - CSR has load balancing problem, but COO does not
    - Need to compute whether the work load is distributed in a balanced way
Outline

- Introduction
- Sparse Matrix Format
- Ultimate Hybrid (UlHy) Sparse Matrix Format
- cIOSKI Framework
- Experimental Results
- Conclusion
Experiment Settings

- Sparse matrix benchmarks
  - Using the 14 matrices from Sam Williams’ 2007 SC paper
- Execution platform
  - Nvidia GTX 480
- Comparison
  - Compare against the cusp implementation
  - The source code has CUDA implementation of DIA, ELL, CSR, COO, and ELL+COO formats
Matrix: Dense

- Blue bars: cusp implementations
- Brown bars: flat, flat+block, flat + dia hybrid
- Green bar: flat + block + dia hybrid
- Best representation from cIOSKI: bcsr

<table>
<thead>
<tr>
<th>spyplot</th>
<th>Name</th>
<th>Dimensions</th>
<th>Nonzeros (nnz/row)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dense</td>
<td>2K x 2K</td>
<td>4.0M (2K)</td>
<td>Dense matrix in sparse format</td>
</tr>
</tbody>
</table>
Matrix: Protein

- Best representation from clOSKI: bell+ell+coo
- Slightly worse than the cusp CSR format
  - Too much kernel launch overhead
Matrix: Spheres

- Best representation from cIOSKI: blockflat sbell+ell+coo
  - Representation from cIOSKI all: dia+bell+ell+coo
  - Too much kernel launch overhead

FEM / Spheres 83K x 83K 6.0M (72) FEM concentric spheres

![GLOPS Chart]

<table>
<thead>
<tr>
<th>Format</th>
<th>GLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIA</td>
<td>0</td>
</tr>
<tr>
<td>ELL</td>
<td>20</td>
</tr>
<tr>
<td>CSR</td>
<td>15</td>
</tr>
<tr>
<td>COO</td>
<td>10</td>
</tr>
<tr>
<td>ELL+COO</td>
<td>25</td>
</tr>
<tr>
<td>Best flat only</td>
<td>22</td>
</tr>
<tr>
<td>blockflat</td>
<td>30</td>
</tr>
<tr>
<td>diagflat</td>
<td>25</td>
</tr>
<tr>
<td>all</td>
<td>30</td>
</tr>
</tbody>
</table>
Matrix: Cantilever

- Best representation from ciOSKI: dia+ell
Matrix: Wind Tunnel

- Best representation from clOSKI: bdia + sell
Matrix: Harbor

- Best representation from clOSKI: ell + coo
Matrix: Ship

- Best representation from cI0SKI: bell + ell + coo
Matrix: Economics

- Best representation from cIOSKI: ell + coo
Best representation from cIOSKI: ell
Matrix: Accelerator

- Best representation from clOSKI: sell
- Best representation from cIOSKI: ell+coo
Matrix: Webbase

- Best representation from cIOSKI: ell+coo
Matrix: LP

- Best representation from cIOSKI: bcsr + csr
  - Slightly worse than the CSR from cusp
    - The CSR implementation of cusp is faster than my OpenCL implementation
      - OpenCL kernels seems to be slower than the same CUDA kernels
Outline

- Introduction
- Sparse Matrix Format
- Ultimate Hybrid (UlHy) Sparse Matrix Format
- cIOSKI Framework
- Experimental Results
- Conclusion
Conclusion

- This project is the first existing OpenCL implementation that covers 9 different sparse matrix formats.
- This project proposes the ultimate hybrid representation that tries to represent each specialized region of the sparse matrix using the best matching format.
- The cIOSKI framework can dynamically decide the best representation of a sparse matrix at runtime based on the offline benchmarking information.
- On the 14 matrices, the cIOSKI framework achieves 74% better GFLOPS performance compared against the cusp ELL+COO format, and 27% better GFLOPS compared against the best cusp implementation.
- The kernel launch overhead degrades the performance, the framework can be expected to deliver even better performance on larger sparse matrices.
Thank You