Generating and Automatically Tuning OpenCL Code for Sparse Linear Algebra

Dominik Grewe    Anton Lokhmotov

Media Processing Division
ARM

School of Informatics
University of Edinburgh

December 13, 2010
Introduction

• Sparse matrices are matrices where the majority of elements are zero

• Sparse linear algebra used in many applications
  • Iterative solvers of linear systems
  • Eigenvalue problems

• Many different formats have been proposed
  • For specific types of matrices (e.g. diagonal matrices)
  • For specific architectures (e.g. GPUs)

• Not obvious which format is best suited for particular architecture
  • Scalar vs. vector architectures
  • Not feasible to implement each format for all architectures
Proposal

We have developed a framework consisting of:

1. A high-level representation for describing sparse matrix formats
   - Independent of architecture and thus portable
   - Compiler is responsible for system-specific optimizations
Proposal

We have developed a framework consisting of:

1. A high-level representation for describing sparse matrix formats
   - Independent of architecture and thus portable
   - Compiler is responsible for system-specific optimizations

2. A compiler generates efficient low-level code from the high-level representation
   - Sparse Matrix-Vector Multiplication (SpMV)
   - Targetting GPUs (OpenCL and CUDA back-end)
Proposal

We have developed a framework consisting of:

1. A high-level representation for describing sparse matrix formats
   - Independent of architecture and thus portable
   - Compiler is responsible for system-specific optimizations

2. A compiler generates efficient low-level code from the high-level representation
   - Sparse Matrix-Vector Multiplication (SpMV)
   - Targeting GPUs (OpenCL and CUDA back-end)

3. Automatic tuning to further improve performance
   - Explore configuration space
Coordinate (COO) format

- Each non-zero element and its coordinates are stored explicitly
- Elements can be stored in random order
- No structure and therefore often unsuitable for computation
# parameter description
parameters:
  num_rows
  num_cols
  num_nonzeros

# data storage description
data:
  values [num_nonzeros] : DATA_TYPE
  colIdx [num_nonzeros] : INDEX_TYPE
  rowIdx [num_nonzeros] : INDEX_TYPE
# access description
access:
    for i in [ 0 : num_nonzeros-1 ]
    {
        _col = colIdx[i]
        _row = rowIdx[i]

        _val = values[i]
    }
ELLPACK/ITPACK (ELL) format

- All rows are stored with fixed width
- Row coordinate implicit in position of elements in arrays
- Elements from adjacent rows are stored in adjacent memory cells
ELLPACK/ITPACK (ELL) format

values colIdx

values
colIdx

Dominik Grewe, Anton Lokhmotov

www.inf.ed.ac.uk
Representation of ELL Format

# parameter description
parameters:
    num_rows
    num_cols
    width      # elements per row

# data storage description
data:
    values [width * num_rows] : DATA_TYPE
    colIdx [width * num_rows] : INDEX_TYPE
Representation of ELL Format

```python
# access description
access:
    for _row in [ 0 : num_rows-1 ]
    {
        for i in [ 0 : width-1 ]
        {
            _col = colIdx[i*num_rows + _row]
            _val = values[i*num_rows + _row]
        }
    }
```
Hybrid (HYB) format by Bell and Garland (NVIDIA)

- Majority of matrix stored in ELL, remaining elements in COO
- Reduces number of explicitly stored zeros
- Useful when there are a few rows with many non-zero elements
Representation of HYB Format

# parameter description
parameters:
    num_rows
    num_cols
    ell_width
    coo_nonzeros

# data storage description
data:
    ell_values [ell_width * num_rows] : DATA_TYPE
    ell_colIdx [ell_width * num_rows] : INDEX_TYPE
    coo_values [coo_nonzeros] : DATA_TYPE
    coo_rowIdx [coo_nonzeros] : INDEX_TYPE
    coo_colIdx [coo_nonzeros] : INDEX_TYPE
Representation of HYB Format

access:

for _row in [ 0 : num_rows-1 ]
{
    for i in [ 0 : ell_width-1 ]
    {
        _col = colIdx[i*num_rows + _row]
        _val = values[i*num_rows + _row]
    }
}

for i in [ 0 : coo_nonzero-1 ]
{
    _col = colIdx[i]
    _row = rowIdx[i]
    _val = values[i]
}
SpMV Code Generation

- Compute dot-product of each matrix row with input vector
- Requires reduction in each row
Compilation Strategies

Formats with efficient row access:

- **Single** work-item per row ("1IPR")
  - Requires no synchronization

- **Multiple** work-items per row ("nIPR")
  - Synchronization across work-items required
Compilation Strategies

Formats with efficient row access:

- *Single* work-item per row (“1IPR”)
  - Requires no synchronization
- *Multiple* work-items per row (“nIPR”)
  - Synchronization across work-items required

Formats without efficient row access:

- Single work-item for entire matrix (“SEQ”)
  - No synchronization, but very slow
- Single work-item per element (“ATOMIC”)
  - Use atomic functions to synchronize work-items assigned to same row
Exploiting data reuse

- Storing the input vector as an image
  - Input vector is reused across rows
  - Accesses are essentially random
  - Use caching of texture memory
Exploiting data reuse

- Storing the input vector as an image
  - Input vector is reused across rows
  - Accesses are essentially random
  - Use caching of texture memory

- Eliminating loads from the input vector
  - Check if matrix element is 0 before loading vector element
  - Reuse vector value for columns in blocked formats
Exploiting data reuse

- Storing the input vector as an image
  - Input vector is reused across rows
  - Accesses are essentially random
  - Use caching of texture memory

- Eliminating loads from the input vector
  - Check if matrix element is 0 before loading vector element
  - Reuse vector value for columns in blocked formats

- Caching matrix data
  - Introduce \_\_reuse keyword for user to specify arrays that get reused
  - Store arrays that get reused as images to benefit from texture caches
Further Optimizations

- Optimizing the reduction phase
  - Parallel reduction
  - Avoid memory bank conflicts
  - No synchronization within “warps”
Further Optimizations

• Optimizing the reduction phase
  • Parallel reduction
  • Avoid memory bank conflicts
  • No synchronization within “warps”

• Loop unrolling
  • Unrolling small loop nests can improve performance
  • E.g. blocked formats
  • Currently always unrolls loop if iteration space is static, but can be made optional
Evaluation Methodology

- six state-of-the-art matrix formats
  - CSR, DIA, ELL and HYB from Bell and Garland [1] (SC 2009)
  - Blocked ELLPACK (BELL) from Choi et al. [2] (PPoPP 2010)
  - Sliced ELLPACK (SELL) from Monakov et al. [3] (HiPEAC 2010)

- 14 matrices taken from [1]

Experiments

- Generated vs hand-written code
- Generated scalar vs vector code

Experimental Setup

- NVIDIA Tesla S1070 w/ CUDA SDK 3.0/2.3
- ATI Radeon HD 5970 w/ Stream SDK v2.1
Results: Diagonal

![Graph showing performance comparison between hand-written, generated (fixed), and generated (tuned) methods for Dense and FEM/Cantilever models. The x-axis represents the type of model, and the y-axis represents GFLOP/s.]
Results: Compressed Sparse Row

CSR

Dense Protein FEM/Spheres FEM/Cantilever Wind Tunnel FEM/Ship QCD FEM/Ship Economics Epidemiology FEM/Ship LP

hand-written generated (fixed) generated (tuned)
Results: ELLPACK/ITPACK

ELL

hand-written  generated (fixed)  generated (tuned)

GFLOP/s

Dense  Protein  FEM/Spheres  FEM/Cantilever  Wind Tunnel  FEM/Harbor  QCD  FEM/Ship  Economics  Epidemiology  FEM/Accelerator  Circuit  Webbase  LP

Dominik Grewe, Anton Lokhmotov www.inf.ed.ac.uk
Results: Hybrid

The chart shows the performance of HYB systems across various domains, such as Dense, Protein, FEM/Spheres, FEM/Cantilever, Wind Tunnel, FEM/Ship, QCD, Economics, Epidemiology, FEM/Accelerator, Circuit, Webbase, and LP. The x-axis represents the different domains, and the y-axis represents the GFLOP/s performance. The bars are color-coded to indicate different types of generated models: hand-written, generated (fixed), and generated (tuned).
Results: Blocked ELLPACK

![Graph showing performance comparison of different models](image-url)
Results: sliced ELLPACK

![Graph showing performance comparison between different fields and their respective benchmarks (Dense, Protein, FEM/Spheres, FEM/Cantilever, Wind Tunnel, FEM/Ship, QCD, Economics, FEM/Ship, FEM/Ship, LP) with categories: hand-written, generated (fixed), generated (tuned).]
Results: Scalar vs Vector

CSR

GFLOP/s

Scalar (generated, tuned)

Vector (generated, tuned)

Dense  Protein  FEM/Spheres  FEM/Cantilever  Wind Tunnel  FEM/Ship  QCD  Economics  Epidemiology  FEM/Accelerator  Circuit  Webbase  LP
Summary

• A new high-level language for sparse matrix formats

• Compiler generating efficient SpMV code for GPUs from format description
  • Comparable performance to hand-written code

• Automatic tuning to find optimal configuration
  • Improves performance even further
Summary

• A new high-level language for sparse matrix formats
• Compiler generating efficient SpMV code for GPUs from format description
  • Comparable performance to hand-written code
• Automatic tuning to find optimal configuration
  • Improves performance even further

Future Work:

• Predict optimal format and configuration for matrix and architecture using machine learning
References


Example: Compressed Sparse Row (CSR) format

- Elements and their column index from same row are stored in contiguous chunks memory.
- Rows are identified by storing the indices of each row’s first element.
- Allows efficient access to individual rows.
Representation of CSR Format

# parameter description
parameters:
    num_rows
    num_cols
    num_nonzeros

# data storage description
data:
    values [num_nonzeros] : DATA_TYPE
    colIdx [num_nonzeros] : INDEX_TYPE
    rowPtr [num_rows + 1] : INDEX_TYPE
# access description
access:
    for _row in [ 0 : num_rows−1 ]
    {
        s = rowPtr[_row]
        e = rowPtr[_row + 1]
        for i in [ s : e−1 ]
        {
            _col = colIdx[i]
            _val = values[i]
        }
    }
Sliced ELLPACK (SELL) by Monakov et al.

- Divide matrix into slices and store each slice in ELL separately.
- Re-order rows to bring together rows of similar length.
- Fixed-height slices vs variable-height slices.
- Reduces number of explicitly stored zeros.
- Store elements in small blocks.
- Divide matrix into slices and store each slice separately in blocked version of ELL.
- Re-order rows to bring together rows of similar length.
- Particularly suitable for matrices where non-zero elements occur in small blocks.