# Implementing Sparse Matrix Vector Product in OpenCL

Anton Lokhmotov, ARM

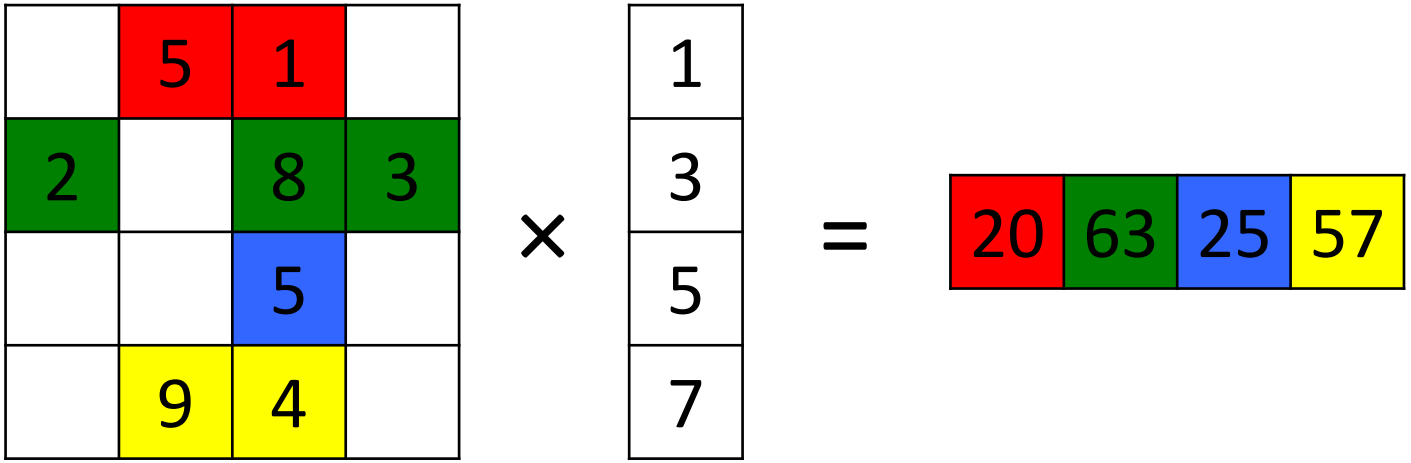
(OpenCL tutorial, HiPEAC'11)

# Sparse Matrices

- In a sparse matrix, the majority of elements are zero
- Specialised formats reduce storage and computation requirements
- Must store non-zero elements
- Must be able to infer element coordinates
- Many formats have been proposed
  - For specific matrix structures (e.g. diagonal)
  - For specific architectures (e.g. GPUs)

# Sparse Matrix Vector Product

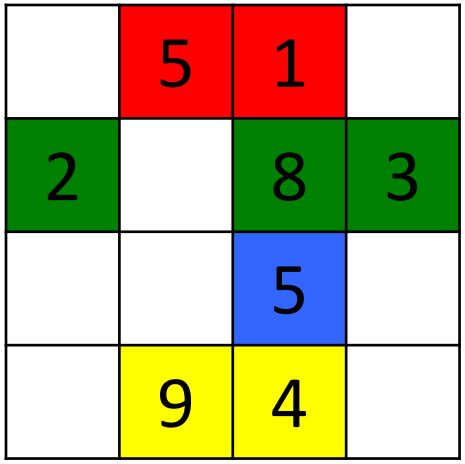
- Sparse linear systems can be solved using iterative methods
- Sparse matrix vector product (perhaps the most important sparse linear algebra kernel?)
- Multiplies a sparse matrix by a dense vector



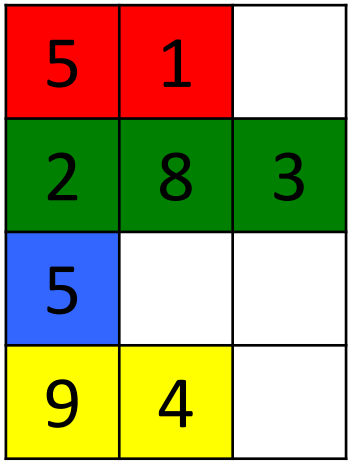
# Representation defines algorithm

- Nathan Bell and Michael Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors". In *SC'09.*
- Jee W. Choi, Amik Singh, and Richard W. Vuduc. "Model-driven autotuning of sparse matrix-vector multiply on GPUs". In *PPoPP'10.*
- Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. "Automatically tuning sparse matrix-vector multiplication for GPU architectures". In *HiPEAC'10.*

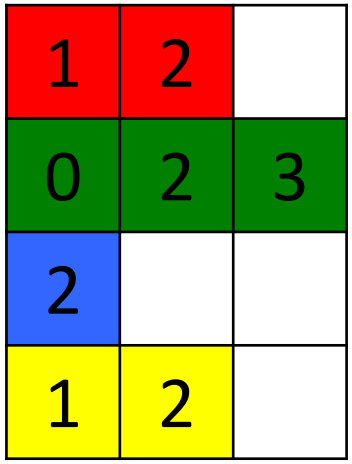
# ELLPACK/ITPACK (ELL)




values




col_idx




- Let K be the maximum number of NZ per row
- values[] stores K (NZ + Z) values for each row
- col_idx[] stores K column indices for each row
- values[] and col_idx[] are in column-major order

ELL is good for GPUs but potentially wasteful

# ELL: format has two parameters

```
const sampler_t _sampler = \
  CLK_ADDRESS_NONE | CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE;

__kernel void mvm_0 (__global float * values, __global int * col_idx,
  __read_only image2d_t vec_in, __global float * vec_out,
  const int num_rows, const int nonzeros_per_row)
{
  const int __idx = get_global_id(0);
  if (__idx >= num_rows)
    return;

  float sum = 0.0f;
  int _row = __idx;
  for (int i = 0; i <= (nonzeros_per_row - 1); ++i)
  {
    int _col = col_idx[((i * num_rows) + _row)];
    float _val = values[((i * num_rows) + _row)];
    sum += _val * read_imagef(vec_in, _sampler, (int2)_col).x;
  }
  vec_out[_row] += sum;
}
```

# ELL: matrix stored as two arrays

```
const sampler_t _sampler = \
  CLK_ADDRESS_NONE | CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE;

__kernel void mvm_0 (__global float * values, __global int * col_idx,
  __read_only image2d_t vec_in, __global float * vec_out,
  const int num_rows, const int nonzeros_per_row)
{
  const int __idx = get_global_id(0);
  if (__idx >= num_rows)
    return;

  float sum = 0.0f;
  int _row = __idx;
  for (int i = 0; i <= (nonzeros_per_row - 1); ++i)
  {
    int _col = col_idx[((i * num_rows) + _row)];
    float _val = values[((i * num_rows) + _row)];
    sum += _val * read_imagef(vec_in, _sampler, (int2)_col).x;
  }
  vec_out[_row] += sum;
}
```

# ELL: one work-item per row

```
const sampler_t _sampler = \
  CLK_ADDRESS_NONE | CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE;

__kernel void mvm_0 (__global float * values, __global int * col_idx,
  __read_only image2d_t vec_in, __global float * vec_out,
  const int num_rows, const int nonzeros_per_row)
{
  const int __idx = get_global_id(0);
  if (__idx >= num_rows)
    return;

  float sum = 0.0f;
  int _row = __idx;
  for (int i = 0; i <= (nonzeros_per_row - 1); ++i)
  {
    int _col = col_idx[((i * num_rows) + _row)];
    float _val = values[((i * num_rows) + _row)];
    sum += _val * read_imagef(vec_in, _sampler, (int2)_col).x;
  }
  vec_out[_row] += sum;
}
```

# ELL: accumulation into register

```
const sampler_t _sampler = \
  CLK_ADDRESS_NONE | CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE;

__kernel void mvm_0 (__global float * values, __global int * col_idx,
  __read_only image2d_t vec_in, __global float * vec_out,
  const int num_rows, const int nonzeros_per_row)
{
  const int __idx = get_global_id(0);
  if (__idx >= num_rows)
    return;

  float sum = 0.0f;
  int _row = __idx;
  for (int i = 0; i <= (nonzeros_per_row - 1); ++i)
  {
    int _col = col_idx[((i * num_rows) + _row)];
    float _val = values[((i * num_rows) + _row)];
    sum += _val * read_imagef(vec_in, _sampler, (int2)_col).x;
  }
  vec_out[_row] += sum;
}
```

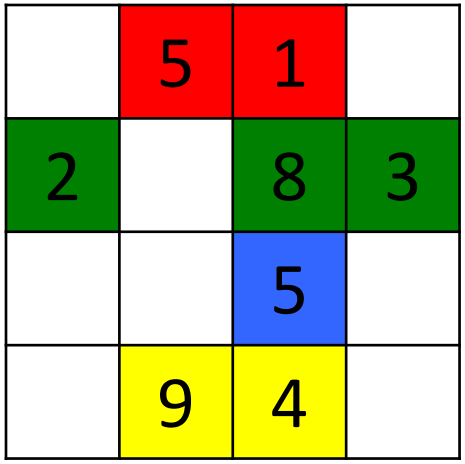
# ELL: “coalesced” memory accesses

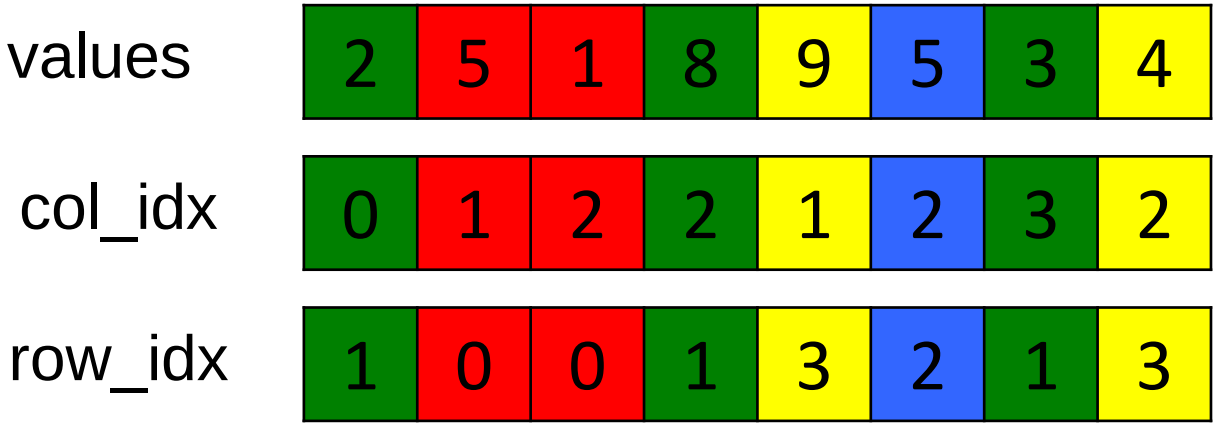
```
const sampler_t _sampler = \
  CLK_ADDRESS_NONE | CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE;

__kernel void mvm_0 (__global float * values, __global int * col_idx,
  __read_only image2d_t vec_in, __global float * vec_out,
  const int num_rows, const int nonzeros_per_row)
{
  const int __idx = get_global_id(0);
  if (__idx >= num_rows)
    return;

  float sum = 0.0f;
  int _row = __idx;
  for (int i = 0; i <= (nonzeros_per_row - 1); ++i)
  {
    int _col = col_idx[((i * num_rows) + _row)];
    float _val = values[((i * num_rows) + _row)];
    sum += _val * read_imagef(vec_in, _sampler, (int2)_col).x;
  }
  vec_out[_row] += sum;
}
```

# ELL: input vector stored as image

```
const sampler_t _sampler = \
  CLK_ADDRESS_NONE | CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE;

__kernel void mvm_0 (__global float * values, __global int * col_idx,
  __read_only image2d_t vec_in, __global float * vec_out,
  const int num_rows, const int nonzeros_per_row)
{
  const int __idx = get_global_id(0);
  if (__idx >= num_rows)
    return;

  float sum = 0.0f;
  int _row = __idx;
  for (int i = 0; i <= (nonzeros_per_row - 1); ++i)
  {
    int _col = col_idx[((i * num_rows) + _row)];
    float _val = values[((i * num_rows) + _row)];
    sum += _val * read_imagef(vec_in, _sampler, (int2)_col).x;
  }
  vec_out[_row] += sum;
}
```

# Coordinate (COO)




- values[] stores the value for each NZ
- col_idx[] stores the column index for each NZ
- row_idx[] stores the row index for each NZ

COO is simple but often unsuitable for computation (elements can be in random order)

# COO: format has one parameter

```
__kernel void mvm_1 (__global float * values,
  __global int * row_idx, __global int * col_idx,
  __global float * vec_in, __global float * vec_out,
  const int num_nonzeros)
{
  const int __idx = get_global_id(0);
  if (__idx >= num_nonzeros)
    return;

  int i = __idx;
  int _row = row_idx[i];
  int _col = col_idx[i];
  float _val = values[i];
  my_atomic_add(&vec_out[_row], _val * vec_in[_col]);
}
```

# COO: matrix stored as three arrays

```
__kernel void mvm_1 (__global float * values,
  __global int * row_idx, __global int * col_idx,
  __global float * vec_in, __global float * vec_out,
  const int num_nonzeros)
{
  const int __idx = get_global_id(0);
  if (__idx >= num_nonzeros)
    return;

  int i = __idx;
  int _row = row_idx[i];
  int _col = col_idx[i];
  float _val = values[i];
  my_atomic_add(&vec_out[_row], _val * vec_in[_col]);
}
```

# COO: one work-item per NZ

```
__kernel void mvm_1 (__global float * values,
  __global int * row_idx, __global int * col_idx,
  __global float * vec_in, __global float * vec_out,
  const int num_nonzeros)
{
  const int __idx = get_global_id(0);
  if (__idx >= num_nonzeros)
    return;

  int i = __idx;
  int _row = row_idx[i];
  int _col = col_idx[i];
  float _val = values[i];
  my_atomic_add(&vec_out[_row], _val * vec_in[_col]);
}
```

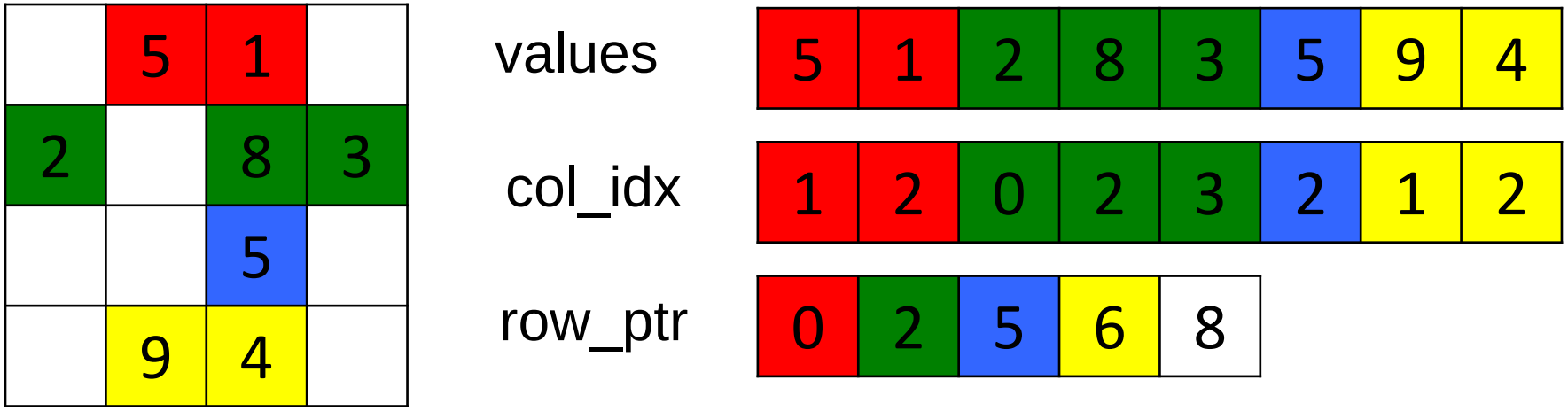
# COO: need float atomic add

```
__kernel void mvm_1 (__global float * values,
  __global int * row_idx, __global int * col_idx,
  __global float * vec_in, __global float * vec_out,
  const int num_nonzeros)
{
  const int __idx = get_global_id(0);
  if (__idx >= num_nonzeros)
    return;

  int i = __idx;
  int _row = row_idx[i];
  int _col = col_idx[i];
  float _val = values[i];
  my_atomic_add(&vec_out[_row], _val * vec_in[_col]);
}
```

# COO: enable int32 extension

```
#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable

void my_atomic_add(__global float * loc, const float f)
{
  float old = *loc;
  float sum = old + f;
  while(atomic_cmpxchg(
       (__global int*)loc, *((int*)&old), *((int*)&sum)
       ) != *((int*)&old))
  {
    old = *loc;
    sum = old + f;
  }
}
```

# Compressed Sparse Row (CSR)




- values[] stores the value for each NZ
- col_idx[] stores the column index for each NZ
- row_ptr[] stores the pointer (integer index) into values[] (and col_idx[]) for each row

CSR provides efficient access to individual rows

# CSR: format has one parameter

```
__kernel void mvm_0 (__global float4 * values,
  __global int4 * col_idx, __global int * row_ptr,
  __global float * vec_in, __global float * vec_out,
  const int num_rows, __local float4 * red_mem)
{
  const int __items_per_row = 8;
  const int __idx = get_global_id(0) / __items_per_row;
  if (__idx >= num_rows) return;

  float4 sum = 0.0f;
  int _row = __idx;
  int s = row_ptr[_row];
  int e = row_ptr[_row + 1];
  for (int i = s + get_local_id(0) % __items_per_row*4;
       i <= (e - 1); i += __items_per_row*4)
  {
    int4 _col = col_idx[(i / 4)];
    float4 _val = values[(i / 4)];
    sum += _val *
  (float4)(vec_in[_col.x], vec_in[_col.y], vec_in[_col.z], vec_in[_col.w]);
  }

  // accumulation followed by reduction...
}
```

# CSR: matrix stored as three arrays

```
__kernel void mvm_0 (__global float4 * values,
  __global int4 * col_idx, __global int * row_ptr,
  __global float * vec_in, __global float * vec_out,
  const int num_rows, __local float4 * red_mem)
{
  const int __items_per_row = 8;
  const int __idx = get_global_id(0) / __items_per_row;
  if (__idx >= num_rows) return;

  float4 sum = 0.0f;
  int _row = __idx;
  int s = row_ptr[_row];
  int e = row_ptr[_row + 1];
  for (int i = s + get_local_id(0) % __items_per_row*4;
       i <= (e - 1); i += __items_per_row*4)
  {
    int4 _col = col_idx[(i / 4)];
    float4 _val = values[(i / 4)];
    sum += _val *
  (float4)(vec_in[_col.x], vec_in[_col.y], vec_in[_col.z], vec_in[_col.w]);
  }

  // accumulation followed by reduction...
}
```

# CSR: eight work-items per row

```
__kernel void mvm_0 (__global float4 * values,
  __global int4 * col_idx, __global int * row_ptr,
  __global float * vec_in, __global float * vec_out,
  const int num_rows, __local float4 * red_mem)
{
  const int __items_per_row = 8;
  const int __idx = get_global_id(0) / __items_per_row;
  if (__idx >= num_rows) return;

  float4 sum = 0.0f;
  int _row = __idx;
  int s = row_ptr[_row];
  int e = row_ptr[_row + 1];
  for (int i = s + get_local_id(0) % __items_per_row*4;
       i <= (e - 1); i += __items_per_row*4)
  {
    int4 _col = col_idx[(i / 4)];
    float4 _val = values[(i / 4)];
    sum += _val *
  (float4)(vec_in[_col.x], vec_in[_col.y], vec_in[_col.z], vec_in[_col.w]);
  }

  // accumulation step followed by reduction step...
}
```

# CSR: four elements per work-item

```
__kernel void mvm_0 (__global float4 * values,
  __global int4 * col_idx, __global int * row_ptr,
  __global float * vec_in, __global float * vec_out,
  const int num_rows, __local float4 * red_mem)
{
  const int __items_per_row = 8;
  const int __idx = get_global_id(0) / __items_per_row;
  if (__idx >= num_rows) return;

  float4 sum = 0.0f;
  int _row = __idx;
  int s = row_ptr[_row];
  int e = row_ptr[_row + 1];
  for (int i = s + get_local_id(0) % __items_per_row*4;
       i <= (e - 1); i += __items_per_row*4)
  {
    int4 _col = col_idx[(i / 4)];
    float4 _val = values[(i / 4)];
    sum += _val *
  (float4)(vec_in[_col.x], vec_in[_col.y], vec_in[_col.z], vec_in[_col.w]);
  }

  // accumulation step followed by reduction step...

}
```

# CSR: fast reduction w/o barriers

```
__kernel void mvm_0 (__global float4 * values,
  __global int4 * col_idx, __global int * row_ptr,
  __global float * vec_in, __global float * vec_out,
  const int num_rows, __local float4 * red_mem)
{
  // reduction step follows accumulation step...

  const int local_id = get_local_id(0);
  const int local_row_id = local_id / __items_per_row;
  const int local_row_offset = local_id % __items_per_row;

  red_mem[local_id] = sum;
  red(red_mem + local_row_id * __items_per_row, local_row_offset);
  if (local_row_offset == 0) {
    float4 r = red_mem[local_row_id * __items_per_row];
    vec_out[_row] = r.x + r.y + r.z + r.w;
  }
}

// avoids bank conflicts in local memory, relies on lock-step execution
void red(__local float4 * red_mem, int idx)
{
  if (idx < 4) red_mem[idx] += red_mem[idx+4];
  if (idx < 2) red_mem[idx] += red_mem[idx+2];
  if (idx < 1) red_mem[idx] += red_mem[idx+1];
}
```

# Conclusion

- OpenCL programming is not for faint-hearted
- The more you optimise code for a particular device, the less performance-portable it gets
- Or even non-portable at all (if includes blatantly architecture-specific code)
- If software development is not painful enough for you, think about maintenance

# Acknowledgement


- No OpenCL code was written for this study…
- Dominik Grewe (University of Edinburgh) wrote an SpMV generator during his HiPEAC-sponsored internship at ARM in autumn 2010
- Performance of generated and automatically tuned code often exceeds that of hand-written code
- Many more interesting questions to answer…