Specialized Sparse Matrix Formats and SpMV Kernel Tuning for GPUs

Alexander Monakov, amonakov@ispras.ru

Institute for System Programming of Russian Academy of Sciences

May 16, 2012

Sparse Matrices

- Most elements are zero
  $\rightarrow$ Need specialized formats: store only non-zero values and positions
- Appear in many scientific codes, most notably CFD
- Usually need to optimize only matrix-vector multiplication:

$$y = Ax : y_i = \sum_{j:A_{ij} \neq 0} A_{ij} x_j$$

- Only two flops per non-zero element
  $\rightarrow$ Usually limited by memory bandwidth
- We consider matrices with unknown complex sparsity patterns
  $\rightarrow$ Otherwise specialized methods can do better

Problem Statement

Develop sparse matrix storage formats with these requirements:

- Optimize only matrix-vector multiplication on GPU
  → When multiplication by transpose is required, store the transpose explicitly
- Reasonably versatile: matrices for arbitrary CFD meshes, preconditioners
  → Not purely diagonal, or with fixed number of non-zeros per row
  → Not only for blocked sparse matrices
- Not relying on ability to reorder the matrix
- For matrices that are reused many times:
  → May need complex conversion
  → May rely on run-time auto-tuning

General Considerations on Optimizing SpMV for GPUs

Primary limiting factor: global memory bandwidth

- Matrix data: streaming, too large for caches
- Source vector: non-streaming random access, but usually good locality
- Destination vector: need coalesced updates

Need to design the format accordingly:

- Facilitate efficient implementation:
  - Minimize synchronization
  - Maximize coalescing
- Reduce memory footprint
  $\rightarrow$ But make space-saving tricks GPU-friendly
- Facilitate coalesced updates to the destination vector

Common Formats: CSR

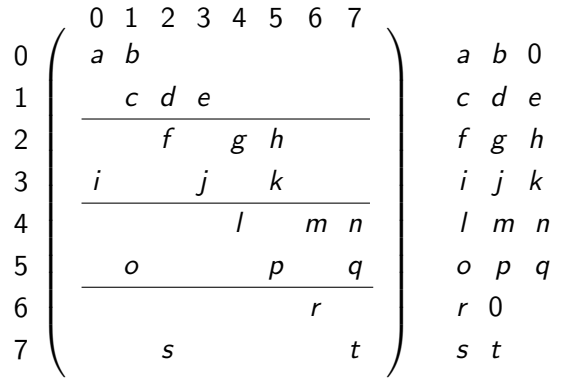
$$
\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\left(\begin{array}{cccccccc}
a & b & & & & & & \\
 & c & d & e & & & & \\
 & & f & & g & h & & \\
i & & & j & & k & & \\
 & & & & l & & m & n \\
 & o & & & & p & & q \\
 & & & & & & r & \\
 & & s & & & & & t
\end{array}\right)
\end{array}
$$

rowptr	0		2			5			8			11			14			17	18		20
value	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*n*	*o*	*p*	*q*	*r*	*s*	*t*	
column	0	1	1	2	3	2	4	5	0	3	5	4	6	7	1	5	7	6	2	7	

Common Formats: ELLPACK

$$
\begin{array}{c}
 \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7
\end{array}
\begin{array}{c}
\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\left(\begin{array}{cccccccc}
a & b & & & & & & \\
 & c & d & e & & & & \\
 & & f & & g & h & & \\
i & & & j & & k & & \\
 & & & & l & & m & n \\
 & o & & & & p & & q \\
 & & & & & & r & \\
 & & s & & & & & t
\end{array}\right)
\end{array}
\qquad
\left(\begin{array}{ccc}
a & b & 0 \\
c & d & e \\
f & g & h \\
i & j & k \\
l & m & n \\
o & p & q \\
r & 0 & 0 \\
s & t & 0
\end{array}\right)
\qquad
\left(\begin{array}{ccc}
0 & 1 & * \\
1 & 2 & 3 \\
2 & 4 & 5 \\
0 & 3 & 5 \\
4 & 6 & 7 \\
1 & 5 & 7 \\
6 & * & * \\
2 & 7 & *
\end{array}\right)
$$

Basic Sliced ELLPACK



sliceptr	0						6						12							18			22
value	*a*	*c*	*b*	*d*	0	*e*	*f*	*i*	*g*	*j*	*h*	*k*	*l*	*o*	*m*	*p*	*n*	*q*	*r*	*s*	0	*t*	
column	0	1	1	2	∗	3	2	0	4	3	5	5	4	1	6	5	7	7	6	2	∗	7	

Sliced ELLPACK Analysis

A hybrid between CSR and ELLPACK:

- Exactly like CSR for slice size $= 1$
- Exactly like ELLPACK for infinite slice size

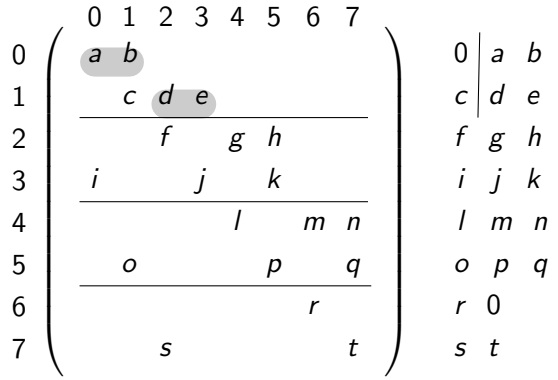
Allows for run-time auto-tuning:

- Slice height
- Threads for slice
- May use variable-height slices

Can do even better:

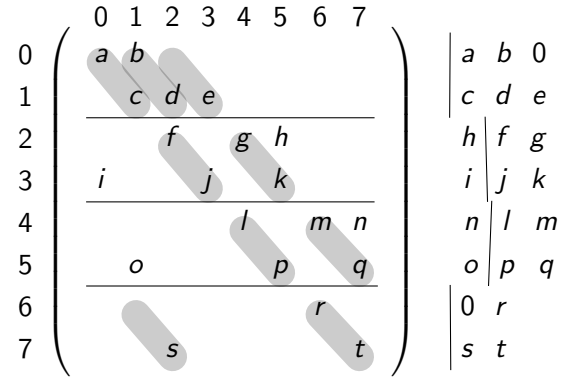
- Compact encoding for 1×2, 1×4 dense sub-blocks
- Compact encoding for diagonal regions in slices

Sliced ELLPACK with Short Blocks



sliceptr	(0, 0)						(6, 4)							(12, 10)							(18, 16)	(22, 20)
value	0	*c*	*a*	*b*	*d*	*e*	*f*	*i*	*g*	*j*	*h*	*k*	*l*	*o*	*m*	*p*	*n*	*q*	*r*	*s*	0	*t*
column	∗	1	0		2		2	0	4	3	5	5	4	1	6	5	7	7	6	2	∗	7

Sliced ELLPACK with Diagonals



sliceptr	(0, 0, 0)							(6, 0, 3)							(12, 2, 5)							(18, 4, 7)		(22, 4, 9)
value	*a*	*c*	*b*	*d*	0	*e*	*h*	*i*	*f*	*j*	*g*	*k*	*n*	*o*	*l*	*p*	*m*	*q*	0	*s*	*r*	*t*		
column							5	0					7	1										
diagoffs	0			1			2				0		2				0		2		−5		0	

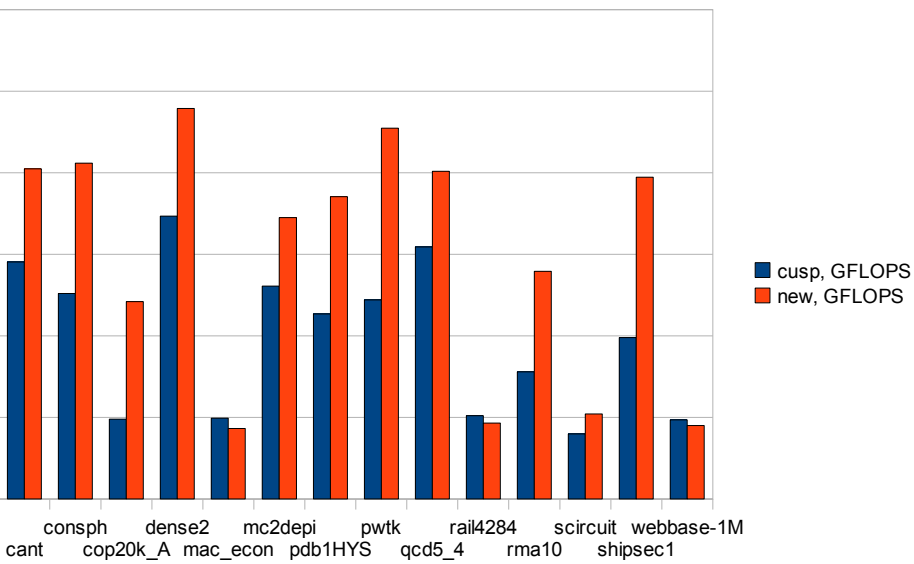
Advanced Formats Analysis

Conversion time — memory footprint tradeoff

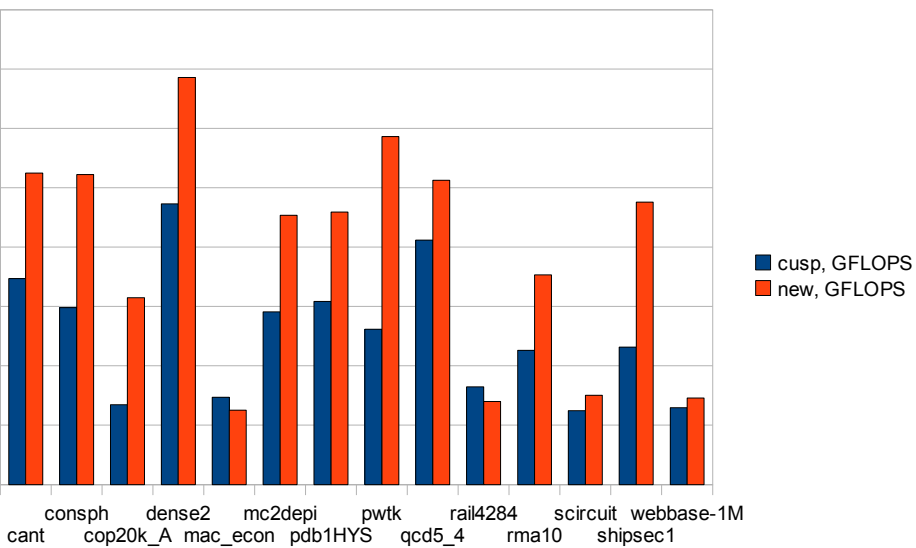
- Require additional pass over matrix data to choose blocks/diagonals
- $\approx$ 1 GB/s conversion speed for non-diagonal variant
- $\approx$ 0.2 GB/s for diagonal variant
- If the matrix is reused, cost is amortized

Auto-tuning will have to try multiple formats: can do that in a long-running CFD simulation

Performance Results: Single Precision (Tesla M2090)



Performance Results: Double Precision (Tesla M2090)



Questions?

Thanks!

Memory Footprint Stats

Testcase	SpMV bytes, CSR	SpMV bytes, Proposed
cant.mtx	32M	24M
consph.mtx	49M	43M
cop20k	22M	22M
dense2.mtx	32M	24M
mac	12M	39M
mc2depi.mtx	23M	13M
pdb1HYS.mtx	35M	33M
pwtk.mtx	95M	76M
qcd5	15M	13M
rail4284.mtx	90M	140M
rma10.mtx	19M	24M
scircuit.mtx	9M	23M
shipsec1.mtx	64M	56M
webbase-1M.mtx	36M	64M