Efficient SpMV on GPUs

Gaurish Telang

SUNY Stony Brook

December 4th,2012

Table Of Contents

The CUDA model

Host and Device

- **Host**

- **Device**

The CUDA model: Introduction

- An application consists of a *sequential* CPU program that may execute parallel programs known as kernels on a GPU.
- A kernel is a computation that is executed by a potentially large number (1000's) of parallel threads.

The CUDA model: Introduction

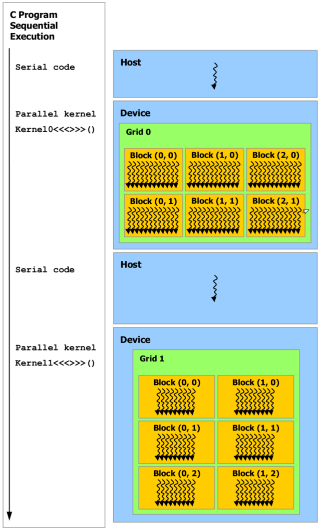
- Each thread runs the *same* scalar sequential program.
- Threads are organized into a grid of *thread-blocks*
- Threads of a **given** block can co-operate amongst themselves using
  - Barrier Synchronization.
  - Atomic operations
  - Per-block shared memory space *private* to each block.
- Each block can execute in *any order* relative to other blocks.
- Threads in different blocks *cannot* co-operate.

The CUDA model: Introduction

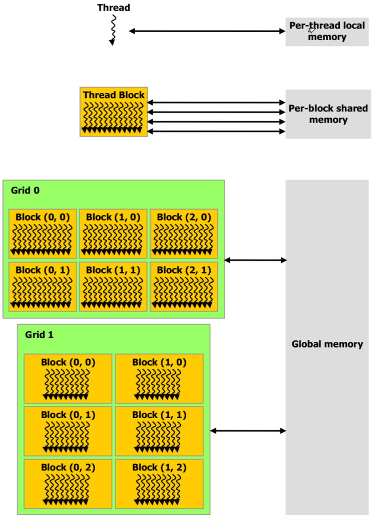
- Each thread runs the *same* scalar sequential program.
- Threads are organized into a grid of *thread-blocks*
- Threads of a **given** block can co-operate amongst themselves using
  - Barrier Synchronization.
  - Atomic operations
  - Per-block shared memory space *private* to each block.
- Each block can execute in *any order* relative to other blocks.
- Threads in different blocks *cannot* co-operate.

The CUDA model: Introduction

Program Execution

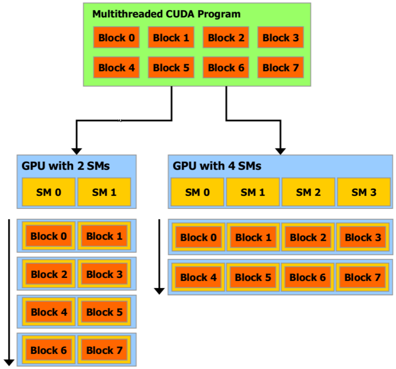


Memory Hierarchy



The CUDA model: Scalability

- Hardware is free to assign blocks to any processor at any time.
  - A kernel scales across any number of parallel processors.



The CUDA C/C++ API

Allocating and Releasing Device Memory

- **Memory Allocation**

```
int* ptr = 0    ;
int  N    = 5000;
int  num_bytes = N * sizeof( int );
cudaMalloc( &ptr, num_bytes);
```

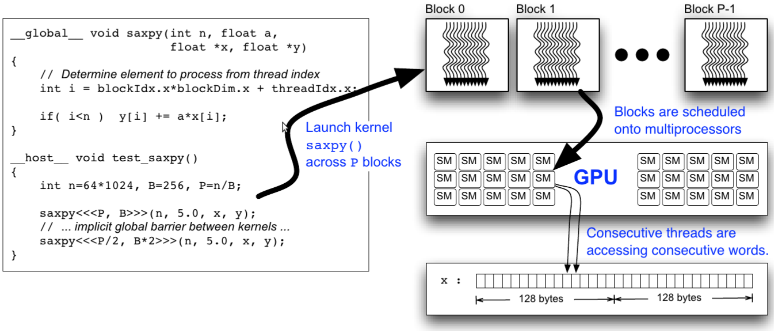
- **Memory transfer**

```
int* h = new int[5000];//Fill the h array as needed
cudaMemcpy(d , h , num_bytes , cudaMemcpyHostToDevice);
// Do Computation on the device
// ...
cudaMemcpy(h , d , num_bytes , cudaMemcpyDeviceToHost);
```

- **Releasing Device Memory**

```
cudaFree(d);
```

Example CUDA Code: Simple SAXPY

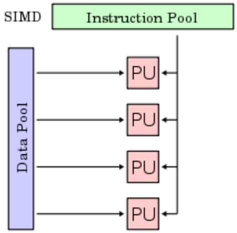


```
__global__ void saxpy(int n, float a,
                      float *x, float *y)
{
    // Determine element to process from thread index
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    if( i<n )  y[i] += a*x[i];
}

__host__ void test_saxpy()
{
    int n=64*1024, B=256, P=n/B;

    saxpy<<<P, B>>>(n, 5.0, x, y);
    // ... implicit global barrier between kernels ...
    saxpy<<<P/2, B*2>>>(n, 5.0, x, y);
}
```

Note: Calls to kernels are asynchronous by default.

The CUDA model: SIMT nature

- Thread creation, scheduling and management is performed entirely in hardware.
- To manage this large population of threads efficiently, the GPU employs a *SIMT* (Single Instruction Multiple Thread) architecture.
- Threads of a block are executed in *groups* of 32 called warps



The CUDA model: Warps

- A warp executes a *single* instruction at a time across **all** its threads.
- Threads of a warp are free to follow their own execution path and all such *execution divergence* is handled automatically in hardware.
- However it is **substantially** more efficient for threads to follow the *same* execution path for the bulk of the computation.

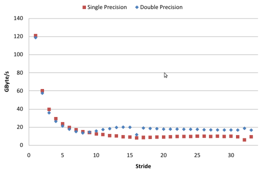
The CUDA model: Warps

- Threads of a warp are free to use *arbitrary* addresses when accessing off-chip memory with load/store operations.
- Accessing scattered locations results in *memory-divergence* and requires the processor to perform **one** memory transaction per-thread.

The CUDA model: Warps

- Non-contiguous access to memory is *detrimental* to memory bandwidth efficiency and to memory-bound computations.

```
__global__
void saxpy_with_stride(int n, float a, float * x, float * y, int stride)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n)
        y[i * stride] = a * x[i * stride] + y[i * stride];
}
```



Relationship between stride and memory bandwidth in the saxpy and daxpy kernels.

Sparse Matrix Formats

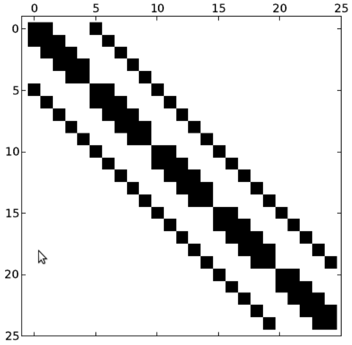
Sparse Matrix Formats

- **Main Idea**: Encode the **non-zero** entries along with the **position** information.
- The encoding's *efficiency* depends on the *sparsity pattern.*
- Examples:
  - Diagonal (DIA)
  - ELLPACK (ELL)
  - Coordinate (COO)
  - Compressed Sparse Row (CSR)
  - Hybrid (ELL+COO)
  - Packet (PKT)

  and others (including hybrids) . . .

Diagonal Format: Introduction

- Appropriate when non-zero values are restricted to a small number of matrix diagonals,
- Efficiently encodes matrices arising from application of stencils to regular grids.



Diagonal Format: An Example

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\mathtt{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \mathtt{offsets} = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

Arrays `data` and `offsets` comprise the DIA representation of `A`.

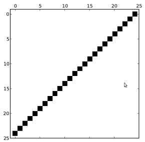
Diagonal Format: + and -'s

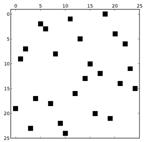
Advantages:

- Row and column indices of each non-zero are defined *implicitly*. This *reduces* memory footprint and data-transfer during an SpMV operation.
- Good for stencils applied to *regular* grids.

Disadvantages:

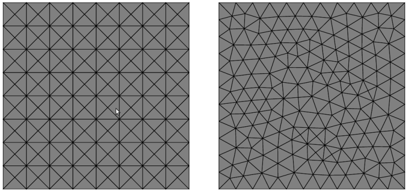
- Allocates storage of values 'outside' the matrix and *explicitly* stores zero values along the diagonal.
- Not good for matrices with *arbitrary* sparsity patterns.





ELLPACK format: Introduction

- Appropriate for an M-by-N matrix with a *maximum* of K non-zeros per row where K is *small*.
- Efficiently encodes sparse matrices which have *roughly the same* number of non-zeros per row.



ELLPACK Format: An Example

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\mathtt{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \mathtt{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 6 & 4 & * \end{bmatrix}$$

Arrays **data** and **indices** comprise the ELL representation of **A**.

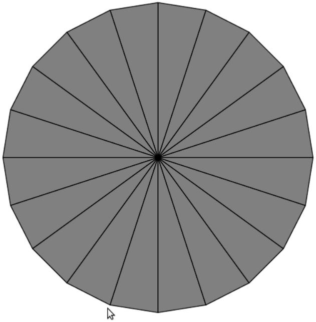
ELLPACK Format: + and -'s

Advantages:

- More general than DIA
- Good for matrices obtained from *semi-structured* and well-behaved *unstructured* meshes .

Disadvantages:

- For unstructured meshes, the *ratio* between **max** number of non-zeros per row and the **average** might be arbitrarily large.
- Hence a *vast* majority of the entries in the data and indices array will be *wasted*.



COO format

- Consists of **three** separate arrays:
  - **row** row-indices
  - **col** column-indices
  - **data** non-zero values
- Required storage *always proportional* to the number of non-zeros. unlike DIA and ELL
- Row-array is kept *sorted* ensuring entries with the same row-index are stored contiguously

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\begin{aligned} \mathtt{row} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix} \\ \mathtt{col} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \mathtt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

CSR format

- A popular and *general purpose* sparse matrix representation
- Can be viewed as a *natural extension* of the (sorted) COO representation with a simple **compression** scheme applied to the (oft-repeated) row indices
- Consists of **three** separate arrays:
    - **ptr** new-row start pointers
    - **indices** column-indices
    - **data** non-zero values
- Required storage *always proportional* to the number of non-zeros. like the COO

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\begin{aligned} \mathtt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \mathtt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \mathtt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

CSR format

- A popular and *general purpose* sparse matrix representation
- Can be viewed as a *natural extension* of the (sorted) COO representation with a simple **compression** scheme applied to the (oft-repeated) row indices
- Consists of **three** separate arrays:
  - **ptr** new-row start pointers
  - **indices** column-indices
  - **data** non-zero values
- Required storage *always proportional* to the number of non-zeros. like the COO

$$\mathtt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$\begin{aligned} \mathtt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \mathtt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \mathtt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

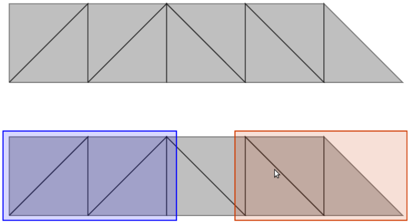
HYB ($=$ ELL $+$ COO) format

- ELL format is suited for *vector machines*, but its efficiency rapidly degrades when the number of non-zeros per matrix varies a lot.
- *Storage efficiency* of COO is invariant to the distribution of non-zeros per row. This invariance **extends** to the cost of a SpMV product using COO.
- Split $A = B + C$ where
  - B contains the *typical* number of non-zeros per row and is stored in ELL.
  - C contains the remaining entries of the *exceptional* rows and is stored in COO

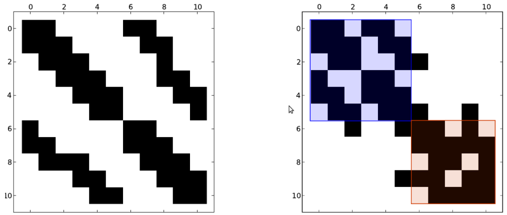
Packet(PKT) format

- Tailored for *symmetric* mesh-based matrices such as those in FEM.
- Idea
  - Reorder the non-zeros in the matrix such that non-zeros are 'concentrated' near the main diagonal.
  - Decompose this reordered matrix into a given number of fixed size partitions
  - Each partition is then stored in a packet data-structure. and each packet is assigned to a thread-block.

PKT format



A sample triangle mesh with 11 vertices partitioned into two sets.



Reordering rows and columns by partition concentrates nonzeros into diagonal submatrices.

PKT format

$$\texttt{pkt0} = \begin{bmatrix} (0,0) & (0,1) & (0,3) & (0,4) & * & * & * & * \\ (1,0) & (1,1) & (1,2) & (1,4) & (1,5) & * & * & * \\ (2,1) & (2,2) & (2,5) & (4,0) & (4,1) & (4,3) & (4,4) & (4,5) \\ (3,0) & (3,3) & (3,4) & (5,1) & (5,2) & (5,4) & (5,5) & * \end{bmatrix}$$

$$\texttt{pkt1} = \begin{bmatrix} (0,0) & (0,1) & (0,3) & (4,1) & (4,2) & (4,3) & (4,4) \\ (1,0) & (1,1) & (1,2) & (1,3) & (1,4) & * & * \\ (2,1) & (2,2) & (2,4) & * & * & * & * \\ (3,0) & (3,1) & (3,3) & (3,4) & * & * & * \end{bmatrix}$$

Packet storage for the two diagonal submatrices in Figure 13. The first submatrix has base index $(0,0)$ while the second submatrix begins at $(6,6)$. Rows of the `pkt0` and `pkt1` correspond to individual thread lanes.

Sparse-Matrix Vector Multiplication

Sparse Matrix Vector multiplication

- We consider the operation $y \leftarrow Ax + y$
- **Central idea**
  Minimize the number of *independent* paths of execution and use the *coalescing* of threads whenever possible.

SpMV for the DIA format

```
__global__ void
spmv_dia_kernel(const int num_rows,
                const int num_cols,
                const int num_diags,
                const int   * offsets,
                const float * data,
                const float * x,
                      float * y)
{
    int row = blockDim.x * blockIdx.x + threadIdx.x;

    if(row < num_rows){
        float dot = 0;

        for(int n = 0; n < num_diags; n++){
            int   col = row + offsets[n];
            float val = data[num_rows * n + row];

            if(col >= 0 && col < num_cols)
                dot += val * x[col];
        }

        y[row] += dot;
    }
}
```

$$
\begin{array}{l|cccccccccccc|}
\text{data} & * & * & 5 & 6 & 1 & 2 & 3 & 4 & 7 & 8 & 9 & * \\
\\
\text{Iteration 0} & & & 2 & 3 & & & & & & & & \\
\text{Iteration 1} & & & & & 0 & 1 & 2 & 3 & & & & \\
\text{Iteration 2} & & & & & & & & & 0 & 1 & 2 &
\end{array}
$$

SpMV for the ELL format

```
__global__ void
spmv_ell_kernel(const int num_rows,
                const int num_cols,
                const int num_cols_per_row,
                const int   * indices,
                const float * data,
                const float * x,
                      float * y)
{
    int row = blockDim.x * blockIdx.x + threadIdx.x;

    if(row < num_rows){
        float dot = 0;

        for(int n = 0; n < num_cols_per_row; n++){
            int   col = indices[num_rows * n + row];
            float val = data[num_rows * n + row];

            if(val != 0)
                dot += val * x[col];
        }

        y[row] += dot;
    }
}
```

$$\begin{array}{rl} \text{data} & \begin{bmatrix} 1 & 2 & 5 & 6 & 7 & 8 & 3 & 4 & * & * & 9 & * \end{bmatrix} \\ \text{indices} & \begin{bmatrix} 0 & 1 & 0 & 6 & 1 & 2 & 2 & 4 & * & * & 3 & * \end{bmatrix} \end{array}$$

$$\begin{array}{rl} \text{Iteration 0} & \begin{bmatrix} 0 & 1 & 2 & 3 & & & & & & & & \end{bmatrix} \\ \text{Iteration 1} & \begin{bmatrix} & & & & 0 & 1 & 2 & 3 & & & & \end{bmatrix} \\ \text{Iteration 2} & \begin{bmatrix} & & & & & & & & 0 & 1 & 2 & 3 \end{bmatrix} \end{array}$$

SpMV for the CSR format(first-version)

```
__host__ void
spmv_csr_serial(const int num_rows,
                const int   * ptr,
                const int   * indices,
                const float * data,
                const float * x,
                      float * y)
{

    for(int row = 0; i < num_rows; i++){
        float dot = 0;

        int row_start = ptr[row];
        int row_end   = ptr[row+1];

        for (int jj = row_start; jj < row_end; jj++)
            dot += data[jj] * x[indices[jj]];

        y[row] += dot;
    }
}
```

$$\begin{array}{rl}
\text{indices} & \begin{bmatrix}0&1&1&2&0&2&3&1&3\end{bmatrix}\\
\text{data} & \begin{bmatrix}1&7&2&8&5&3&9&6&4\end{bmatrix}\\
\\
\text{Iteration 0} & \begin{bmatrix}0& &1& &2& & &3& \end{bmatrix}\\
\text{Iteration 1} & \begin{bmatrix} &0& &1& &2& & &3\end{bmatrix}\\
\text{Iteration 2} & \begin{bmatrix} & & & & & &2& & \end{bmatrix}
\end{array}$$

SpMV for the CSR format(second-version)

```
__global__ void
spmv_csr_vector_kernel(const int num_rows,
                       const int   * ptr,
                       const int   * indices,
                       const float * data,
                       const float * x,
                             float * y)
{
    __shared__ float vals[];

    int thread_id = blockDim.x * blockIdx.x + threadIdx.x;  // global thread index
    int warp_id   = thread_id / 32;                         // global warp index
    int lane      = thread_id & (32 - 1);                   // thread index within the warp

    // one warp per row
    int row   = warp_id;

    if (row < num_rows){
        int row_start = ptr[row];
        int row_end   = ptr[row+1];

        // compute running sum per thread
        vals[threadIdx.x] = 0;
        for(int jj = row_start + lane; jj < row_end; jj += 32)
            vals[threadIdx.x] += data[jj] * x[indices[jj]];

        // parallel reduction in shared memory
        if (lane < 16) vals[threadIdx.x] += vals[threadIdx.x + 16];
        if (lane <  8) vals[threadIdx.x] += vals[threadIdx.x +  8];
        if (lane <  4) vals[threadIdx.x] += vals[threadIdx.x +  4];
        if (lane <  2) vals[threadIdx.x] += vals[threadIdx.x +  2];
        if (lane <  1) vals[threadIdx.x] += vals[threadIdx.x +  1];

        // first thread writes the result
        if (lane == 0)
            y[row] += vals[threadIdx.x];
    }
}
```

$$\begin{array}{l}
\texttt{indices} \quad [0\ 1\ 1\ 2\ 0\ 2\ 3\ 1\ 3] \\
\texttt{data} \quad [1\ 7\ 2\ 8\ 5\ 3\ 9\ 6\ 4]
\end{array}$$

$$\begin{array}{lccccccccc}
\text{Warp } 0 & [0 & 0 & & & & & & &] \\
\text{Warp } 1 & [& & 1 & 1 & & & & &] \\
\text{Warp } 2 & [& & & & 2 & 2 & 2 & &] \\
\text{Warp } 3 & [& & & & & & & 3 & 3]
\end{array}$$

SpMV for the PKT format

```
__device__ void
process_packet(const int work_per_thread,
               const int   * index_packet,
                     float * data_packet,
               const float * x_local,
                     float * y_local)
{
    for(int i = 0; i < work_per_thread; i++){
        // offset into packet arrays
        int pos = blockDim.x * i + threadIdx.x;

        //row and column indices are packed in one 32-bit word
        int packed_index = index_array[pos];

        // row and column stored in upper and lower half-words
        int row = packed_index >> 16;
        int col = packed_index & 0xFFFF;

        float val = data_array[pos];

        y_local[row] += val * x_local[col];
    }
}
```

A block of threads computes the matrix-vector product for a single packet. Here
y_local reside in shared memory for low-latency access.

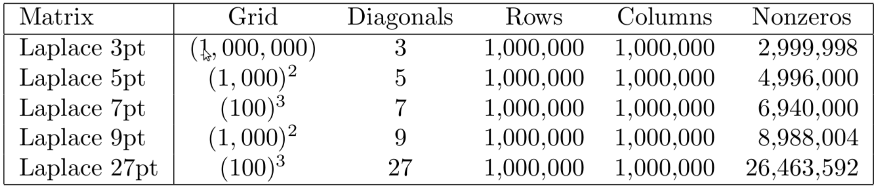
Performance

Performance

- Different classes of matrices highlight the strength and weaknesses of the sparse formats and their associated SpMV algorithms.
- Performance data are measured in terms of
  - Speed of execution (GFLOP/s)
  - Memory bandwidth utilization (GBytes/s)
- Measurements do *not* include time spent *transferring* data between the CPU and the GPU since we are trying to measure just the SpMV performance.
- The reported figures represent an *average* (arithmetic-mean) of **500** SpMV operations.

Structured Matrices

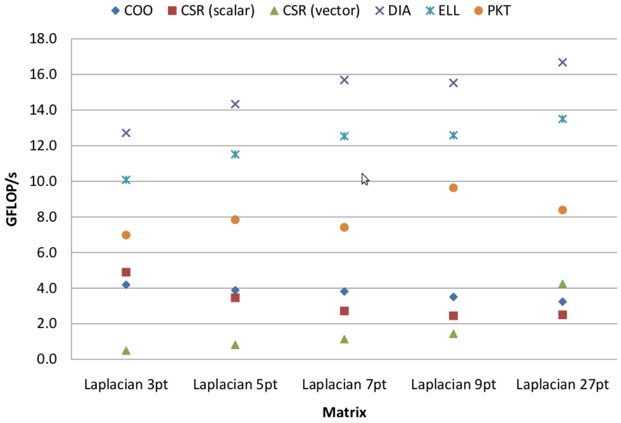
Performance: Structured Matrices



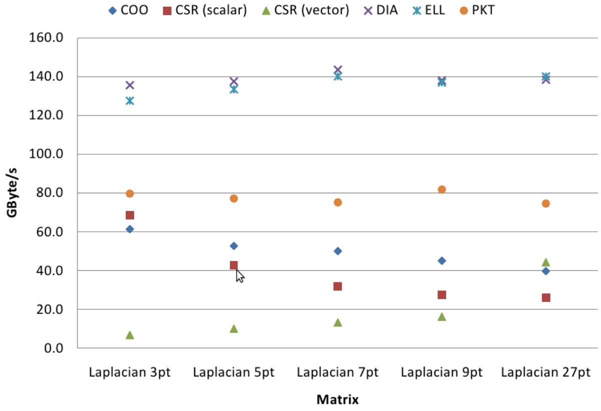
Matrix	Grid	Diagonals	Rows	Columns	Nonzeros
Laplace 3pt	$(1,000,000)$	3	1,000,000	1,000,000	2,999,998
Laplace 5pt	$(1,000)^2$	5	1,000,000	1,000,000	4,996,000
Laplace 7pt	$(100)^3$	7	1,000,000	1,000,000	6,940,000
Laplace 9pt	$(1,000)^2$	9	1,000,000	1,000,000	8,988,004
Laplace 27pt	$(100)^3$	27	1,000,000	1,000,000	26,463,592

Structured matrices used for performance testing.

Performance: Structured Matrices: Execution Speed

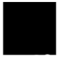
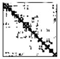
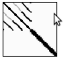
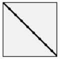
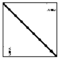
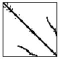
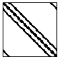


Performance: Structured Matrices: Memory Bandwidth

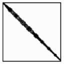
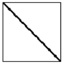
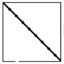
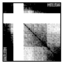
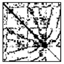
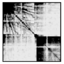


Unstructured Matrices

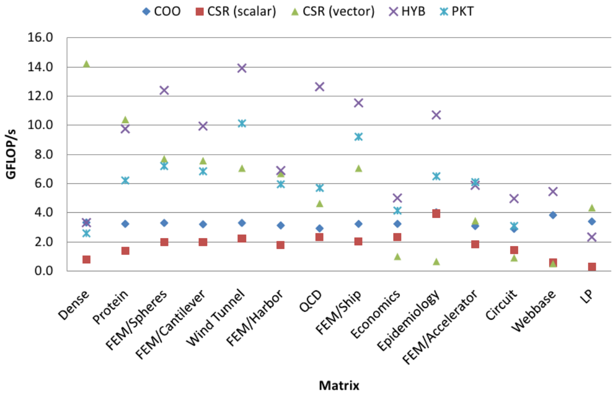
Performance: Unstructured Matrices: DataSet

spyplot	Name	Dimensions	Nonzeros (nnz/row)	Description
	Dense	2K x 2K	4.0M (2K)	Dense matrix in sparse format
	Protein	36K x 36K	4.3M (119)	Protein data bank 1HYS
	FEM / Spheres	83K x 83K	6.0M (72)	FEM concentric spheres
	FEM / Cantilever	62K x 62K	4.0M (65)	FEM cantilever
	Wind Tunnel	218K x 218K	11.6M (53)	Pressurized wind tunnel
	FEM / Harbor	47K x 47K	2.37M (50)	3D CFD of Charleston harbor
	QCD	49K x 49K	1.90M (39)	Quark propagators (QCD/LGT)

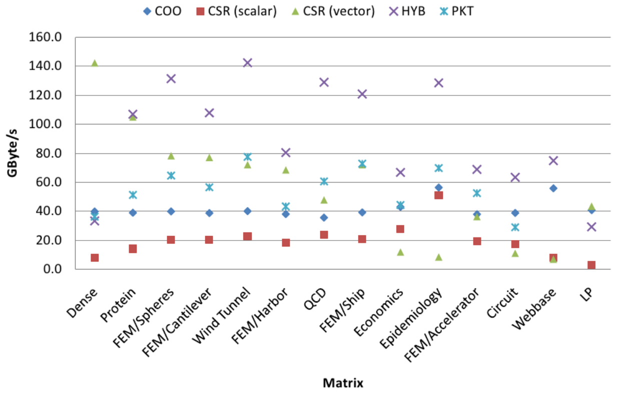
Performance: Unstructured Matrices: DataSet (ctd...)

	FEM/Ship	141K x 141K	3.98M (28)	FEM Ship section/detail
	Economics	207K x 207K	1.27M (6)	Macroeconomic model
	Epidemiology	526K x 526K	2.1M (4)	2D Markov model of epidemic
	FEM / Accelerator	121K x 121K	2.62M (22)	Accelerator cavity design
	Circuit	171K x 171K	959K (6)	Motorola circuit simulation
	webbase	1M x 1M	3.1M (3)	Web connectivity matrix
	LP	4K x 1.1M	11.3M (2825)	Railways set cover Constraint matrix

Performance: Unstructured Matrices: Speed Of Execution



Performace: Unstructured Matrices: Memory Bandwidth

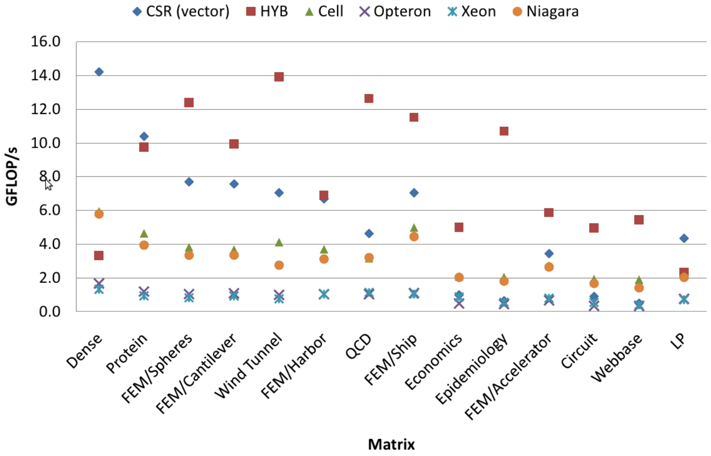


Performance: Comparisons With Multicore

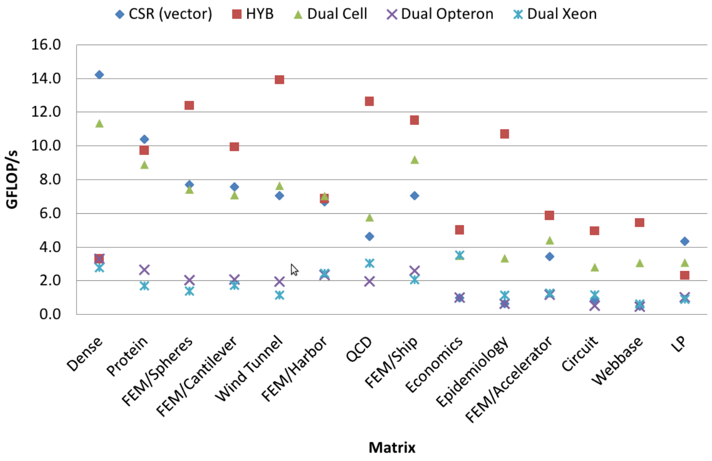
Platform Specifications

Name	Sockets	Cores	Clock (GHz)	Description
Cell	1	8 (SPEs)	3.2	IBM QS20 Cell Blade (half)
Opteron	1	2	2.2	AMD Opteron 2214
Xeon	1	4	2.3	Intel Clovertown
Niagara	1	8	1.4	Sun Niagara2
Dual Cell	2	16 (SPEs)	3.2	IBM QS20 Cell Blade (full)
Dual Opteron	2	4	2.2	2 x AMD Opteron 2214
Dual Xeon	2	8	2.3	2 x Intel Clovertown

Performance: Comparison With Single Socket



Performance: Comparison With Double Socket



$\boxed{\text{Questions?}}$