Review

- Iterative Solutions to linear equations
  - Jacobi
  - Gauss-Seidel
- Sparse matrices
  - Red-Black labeling
- Conjugate Gradient method
- Sparse matrix representations
  - Coordinate form
  - Compressed Sparse Row (CSR method)

2. Compressed Sparse row format

Here too we have 3 arrays, VAL, I and J

- The non zero elements in VAL are ordered by rows
- We first store all non zero values from row 0, then all non zero values from row 1, etc
CSCE 5160 Parallel Processing

3. Diagonal Storage format

Here we will assume that any row has no more than d element around the diagonal

We store all non zero elements in n*d matrix

-- only non zero elements from each row are stored

The OFFSET array (d*1) stores the offsets for the diagonal elements

Offset -3 shows that the first non zero values in VAL are from a diagonal which is 3 positions to the left of the principal diagonal

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>17</td>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>-</td>
</tr>
</tbody>
</table>

OFFSET

-3 -1 0 1

CSCE5160 April 15, 2019

4. Ellpack-Itpack format

This is similar to Diagonal Storage format but more general

Good if no row has more than m non zero elements

and the average number of non-zero elements per row is close to m

We use two n*m matrices VAL and J

VAL stores all non zero elements of the original matrix

J indicates the column position for the non zero values

-1 in J matrix indicates the end of non-zero elements in that row

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>17</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>-1</td>
</tr>
</tbody>
</table>

Note: the trade-off we are looking at is: the memory needed for storing the matrix versus the amount of time needed to access the elements

CSCE5160 April 15, 2019
We have a block around the principle diagonal (with max of 3 elements per row) and we also have two other blocks (each with a principle diagonal only).

For this partitioning, we can use the Diagonal storage form.

Rows of VAL are distributed using stripping -- there is no need to distribute OFFSET since the location of the diagonal elements is fixed and known to all processors.

Now let us see if and what type of communication is needed between processors.

Note each processor will also receive “stripped” elements of the vector b in A*b

The computations along the principle diagonal, all values are local.
Consider this example (12x12 matrix and 12 vector elements) and 4 processors.

No communication for main diagonal (vector elements needed are in the same processor).

May need communication for off-diagonal elements (vector elements needed may be in neighboring processors).

The outer diagonal elements may need communication (vector elements needed are at farther away processors).
CSCE 5160 Parallel Processing

To compute the necessary multiplications and additions, each processor needs one vector element from its neighbor above and one vector element from its neighbor below. This communication costs 2 * (t_s + t_w) assuming one word communication.

Now we need to consider the computations involving the "outer" diagonal elements — the blocks above and below the principle diagonal blocks.

We need to get the vector elements to the processors containing these sparse matrix elements.

The communication cost for these steps depends on how far the outer diagonal elements are, and the value of p in relationship to this value.

Let us assume that we will use (n^{1/2}) as the distance — because of 2-D shape of grid points — one n^{1/2} above and one n^{1/2} below.

Let us estimate communication cost using this value.

This depends on how many processors we have:

If n/p is greater than n^{1/2} or p < n^{1/2}:

You communicate values with your two neighbors and you need to exchange n^{1/2} elements with your neighbor.

Communication cost = 2 * (t_s + (n^{1/2}) * t_w)

Total cost = 5 * t_c * (n/p) + 2 * (t_s + t_w) + 2 * (t_s + (n^{1/2}) * t_w)

If n/p is less than n^{1/2} or p > n^{1/2}:

Processor p_j needs vector elements from processor j + or - p/ (n^{1/2})

Thus each processor must exchange all its n/p vector elements with processors located at a distance of p/ (n^{1/2}).

This is like a shift (not circular) operation exchanging data among processors. Actually two shift operations, one in each direction.

Communication cost = 2^n (t_c + (n/p)^n t_e)

Total cost = 5 * t_c * (n/p) + 2 * (t_s + t_e) + 2^n (t_c + (n/p)^n t_e)
Can we do any better? Consider an idea something like checkerboard partitioning with a twist

Consider assigning grid points to processors.

That is if each processor is responsible for say n/p grid points, then we will allocate the appropriate rows corresponding to these grid points to processors -- along with appropriate vector elements.

An example: we have only 4 processors
we will assign 4 neighboring grid points to each processor

```
  0 1 2 3
  4 5 6 7
  8 9 10 11
 12 13 14 15
```

P0: 0, 1, 4, 5
P1: 2, 3, 6, 7
P2: 8, 9, 12, 13
P3: 10, 11, 14, 15
Each processor needs to communicate with at most 4 others to obtain “boundary” vector elements for the boundary grids points.

In general each processor has \((n/p)^{1/2}\) boundary points (number of data items needed).

To compute boundary point we need communication with 4 neighbors.

Total communication cost = \(4*(t_a + (n/p)^{1/2} * t_w)\)

Total cost = \(5*t_c *(n/p) + 4*(t_s + (n/p)^{1/2} * t_w)\)

If \(p > (n^{1/2})\), the new method is better than row striping.

What if the sparse matrix is not in tri-diagonal form

Even if we have unstructured, in most scientific computations, we can assume that on average each row of the matrix has a small number of non zero elements.

We can then use Ellpack-Itpack representation of the matrix.

How do we distribute such representation?

Row striping of VAL and J structures.
Row striping of the vector also.

Let us discuss the communication needed.

Since the matrix unstructured, the elements in any row may be randomly distributed, a processor may need any random vector element.

So we need all-to-all broadcast to distributed the vector elements.
Each processor has \(n/p\) vector elements.
CSCE 5160 Parallel Processing

Assuming hypercube: the broadcast takes \( t_s \log p + t_w n \)

If we have an average of \( m \) non zero elements in each row, each processor performs \( m \times (n/p) \) multiplications and additions.

total time = \( t_c \times m \times (n/p) + t_s \log p + t_w n \)

Looking at the last term, the parallel run time = \( O(n!) \)

What will be complexity of the sequential implementation.

A matrix by vector product takes \( O(n^2) \). But if have an average of \( m \) nonzero elements in a row – and we can store the nonzero elements using one of the sparse matrix representations

the sequential time is given by \( O(n^2 m) = O(n) \) if \( m \) is much smaller than \( n \)

NO SPEEDUP

CSCE 5160 Parallel Processing

Consider a different partitioning. How about using checker-board (2-D) partitioning of the unstructured sparse matrix (either Ellpack-Itpack or some other format).

Each checkerboard will consist of \( (n/p^{1/2}) \times (n/p^{1/2}) \) sub-matrices.

However, the vector is assigned only to the last column \( (p^{1/2}) \) of processors uniformly say using row stripping - each processor gets \( (n/p^{1/2}) \) vector elements

So, only this last column of processors need to send their data to all processors.

How can we achieve this communication efficiently? – pipelining?
CSCE 5160 Parallel Processing

What is the communication cost?

Read discussion from the supplemental material posted

There are other ways of distributing matrices to processors discussed

Introduction to GPU Architecture

GPUs rely on SIMD model of computation
   All threads execute the same instruction in lock-step fashion
   Each thread operates on a different set of elements based on its index

So, the key to programming GPU’s is to understand how each thread can compute
the index it operates on.

This depends on how many dimensions there are in the data
Depends on how threads are organized

Threads are grouped into blocks
   (possibly 16 or 32 or 64 threads per block)

Blocks are grouped into Grids
   (can be 8, 16 or 32 blocks per grid)

Grids can be organized in 3D grids
   Each thread, block can be 1, 2 3 Dimensional
Think of **Device Memory** (we will also refer to it as **Global Memory**) as a RAM for your GPU.

GPUs have many **Streaming Multiprocessors (SMs)**:
- Each SM has multiple processors but only one instruction unit.
- Groups of processors must run the exact same set of instructions at any given time with in a single SM.
When a kernel (the thing you define in .cu files) is called, the task is divided up into threads

- Each thread handles a small portion of the given task

The threads are divided into a Grid of Blocks

- Both Grids and Blocks are 3 dimensional
  - e.g.
    - \texttt{dim3 dimBlock(8, 8, 8)};
    - \texttt{dim3 dimGrid(100, 100, 1)};
    - \texttt{Kernel<<<dimGrid, dimBlock>>>(…)};

- However, we’ll often only work with 1 dimensional grids and blocks
  - e.g. \texttt{Kernel<<<block\_count, block\_size>>>(…)};

Maximum number of threads per block is usually restricted (512 or 1024 max) depending on the machine

Maximum number of blocks per grid is usually 65535

- If you go over either of these numbers your GPU will just give up or output garbage data
- Much of GPU programming is dealing with this kind of hardware limitations!
- This limitation also means that your Kernel must compensate for the fact that you may not have enough threads to individually allocate to your data points
Each block is assigned to an SM (streaming multiprocessor)

Inside the SM, the block is divided into Warps of threads

- Warps consist of 32 threads
- All 32 threads MUST run the exact same set of instructions at the same time
- Due to the fact that there is only one instruction unit
- Warps are run concurrently in an SM
- If your Kernel tries to have threads do different things in a single warp (using if statements for example), the two tasks will be run sequentially
- Called **Warp Divergence** (NOT GOOD)

---

We need to define number of blocks per grid, number of threads per block.

Simple example: Add elements of two vectors

```c
for (i =0; i<N; i++) {C[i] = A[i]+B[i];}
```

```c
int main()
{
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

```c
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Each thread performs one addition (one iteration)
The index used by the thread is equal to the thread ID. (Id can be obtained using threadIdx or threadIdy (if a block is defined with 2 dimensions)

The main is invoking N threads, one block per grid and N threads per block

Compile using `nvcc filename.cu --o outputfile`
CSCE 5160 Parallel Processing

Consider addition elements of two matrices

\[
\text{for} \ (i = 0; i < N; i++) \ \{ \\
\text{for} \ (j = 0; j < N; j++) \ C[i][j] = A[i][j] + B[i][j]; \}
\]

// Kernel definition

```c
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
```

int main()
{
...
// Kernel invocation with one block of N * N * 1 threads
int numBlocks = 1;
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
...
}

Here also we are using one block but the number of threads per block are N*N → 2D block

In this case, each thread will be identified by x and y IDs.

CSCE 5160 Parallel Processing

Same example but different thread configuration

```c
int main()
{
...
// Kernel invocation
dim3 threadsPerBlock(16, 16);
dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
...
}
```

// Kernel definition

```c
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N) C[i][j] = A[i][j] + B[i][j];
}
```

int main()
{
...
// Kernel invocation with one block of N * N * 1 threads
int numBlocks = 1;
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
...
}

Number of threads per block is limited (512 or 1024)
Here we limit to 256 but use more blocks
Thread configuration should take advantage of how GPU memory is organized

Threads in a block share memory (and registers)
Blocks share local memory
All blocks have access to global memory

We need to send the data from CPU memory to GPU memory

GPU (Device) memory is accessible to both CPU and GPU

So, we need to first allocate space in Device memory for kernel data
Then move the data from CPU memory to GPU/Device memory
And when kernel completes, move the data from Device memory to CPU memory

Consider our vectorADD example:

```c
for (i=0; i<N; i++) C[i] = A[i]+B[i];

int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors
    ...
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    
    // Vector addition
    ...
    
    // Copy vectors from device memory to host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_B, d_B, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost);

    // Call kernel
    callKernel<<<...>>>(d_A, d_B, d_C);
}```
CSCE 5160 Parallel Processing

Continuing with Vector Add example (still part of main running on CPU)

// invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
// Free host memory
... Sample programs on arch are at /usr/local/cuda-10.1/samples/
Test at least the vectorAdd application