Sparse Linear Algebra on GPUs

Hartwig Anzt
hanzt@icl.utk.edu
http://www.icl.utk.edu/~hanzt/

Innovative Computing Lab
Electrical Engineering and Computer Science
University of Tennessee
Recall from last lecture

Properties of GPUs:
- Many lightweight cores (high degree of parallelism)
- Extremely high computing power (FLOP/s)
- SIMD execution mode (all threads of a warp follow the same execution path)
- High memory bandwidth
- No sophisticated memory hierarchy, small caches

Why?
- Originally designed for graphics use
- No dependencies when updating pixels on a screen
- Many pixels need to be updated at the same time

How is the data accessed for general purpose computing?
- On CPUs: sophisticated memory hierarchy (L1, L2, L3...)
- On GPU: fast switching between contexts to hide latencies: execute operation on data that is present
Recall from last lecture

Suitable for
• Compute-intensive operations
• SIMD operations (no branching)
• Data-independent operations
• Uniform memory access

High speedups over CPU code possible for
• Monte-Carlo simulations
• Molecular Dynamics code
• Statistics
• Dense Linear Algebra
•Stencil operations
• Machine learning

Performance

Algorithm complexity

• Regular data structures
• Explicit time stepping
• Simple update rules

• Unstructured mesh
• Implicit solvers
• Adaptive alg.
Recall from last lecture

C Program
 Sequential
 Execution

Serial code

Parallel kernel
Kernel0<<<0>>>()

Device
Grid 0
Block (0, 0) Block (1, 0) Block (2, 0) Block (3, 0) Block (4, 0) Block (5, 0)
Block (0, 1) Block (1, 1) Block (2, 1) Block (3, 1) Block (4, 1) Block (5, 1)
Block (0, 2) Block (1, 2) Block (2, 2) Block (3, 2) Block (4, 2) Block (5, 2)

send data to the GPU

Serial code

Parallel kernel
Kernel1<<<0>>>()

Device
Grid 1
Block (0, 0) Block (1, 0) Block (2, 0) Block (3, 0) Block (4, 0) Block (5, 0)
Block (0, 1) Block (1, 1) Block (2, 1) Block (3, 1) Block (4, 1) Block (5, 1)
Block (0, 2) Block (1, 2) Block (2, 2) Block (3, 2) Block (4, 2) Block (5, 2)

receive results from the GPU

GPU memory
(GDDR5)

PCI bus

CPU memory
(DDG3)
Recall from last lecture:

- CPU Program Execution:
  - Sequential code
  - Parallel kernel: Kernel0<<>>()

- Device:
  - Grid 0:
    - Block (0, 0)
    - Block (1, 0)
    - Block (2, 0)
    - Block (0, 1)
    - Block (1, 1)
    - Block (2, 1)

- Host
  - Send data to the GPU
  - Receive results from the GPU

- PCI bus:
  - ~ 16 GB/s (PCI gen. 3.0)
  - ~ 32 GB/s (PCI gen. 4.0)

- GPU memory (GDDR5):
  - ~ 720 GB/s (P100)

- CPU memory (DDG3):
  - ~ 68 GB/s (Haswell)

- C Program
  - Sequential Execution
  - Serial code
  - Parallel kernel
    - Kernel1<<>>()
Recall from last lecture

C Program
Sequential
Execution

<table>
<thead>
<tr>
<th>Serial code</th>
<th>Host</th>
<th>Device</th>
<th>Grid 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Block (0, 0)</td>
<td>Block (0, 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Block (1, 0)</td>
<td>Block (1, 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Block (2, 0)</td>
<td>Block (2, 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel kernel</th>
<th>Host</th>
<th>Device</th>
<th>Grid 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel0&lt;&lt;&lt;&gt;&gt;&gt;()</td>
<td></td>
<td>Block (0, 0)</td>
<td>Block (1, 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Block (0, 1)</td>
<td>Block (1, 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Block (0, 2)</td>
<td>Block (1, 2)</td>
</tr>
</tbody>
</table>

GPU memory
(GDDR5)

~ 720 GB/s
(P100)

PCI bus

+ NVLink ?

~ 16 GB/s (PCI gen. 3.0)
~ 32 GB/s (PCI gen. 4.0)

CPU memory
(DDG3)

~ 68 GB/s
(Haswell)
Recall from last lecture

Keep data & computations on the GPU as much as possible!
<table>
<thead>
<tr>
<th>Tesla Products</th>
<th>Tesla K40</th>
<th>Tesla M40</th>
<th>Tesla P100</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>GK110 (Kepler)</td>
<td>GM200 (Maxwell)</td>
<td>GP100 (Pascal)</td>
</tr>
<tr>
<td>SMs</td>
<td>15</td>
<td>24</td>
<td>56</td>
</tr>
<tr>
<td>TPCs</td>
<td>15</td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td>FP32 CUDA Cores / SM</td>
<td>192</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>FP32 CUDA Cores / GPU</td>
<td>2880</td>
<td>3072</td>
<td>3584</td>
</tr>
<tr>
<td>FP64 CUDA Cores / SM</td>
<td>64</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>FP64 CUDA Cores / GPU</td>
<td>960</td>
<td>96</td>
<td>1792</td>
</tr>
<tr>
<td>Base Clock</td>
<td>745 MHz</td>
<td>948 MHz</td>
<td>1328 MHz</td>
</tr>
<tr>
<td>GPU Boost Clock</td>
<td>810/875 MHz</td>
<td>1114 MHz</td>
<td>1480 MHz</td>
</tr>
<tr>
<td>Peak FP32 GFLOPs¹</td>
<td>5040</td>
<td>6840</td>
<td>10600</td>
</tr>
<tr>
<td>Peak FP64 GFLOPs¹</td>
<td>1680</td>
<td>210</td>
<td>5300</td>
</tr>
<tr>
<td>Texture Units</td>
<td>240</td>
<td>192</td>
<td>224</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>384-bit GDDR5</td>
<td>384-bit GDDR5</td>
<td>4096-bit HBM2</td>
</tr>
<tr>
<td>Memory Size</td>
<td>Up to 12 GB</td>
<td>Up to 24 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>1536 KB</td>
<td>3072 KB</td>
<td>4096 KB</td>
</tr>
<tr>
<td>Register File Size / SM</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>Register File Size / GPU</td>
<td>3840 KB</td>
<td>6144 KB</td>
<td>14336 KB</td>
</tr>
<tr>
<td>TDP</td>
<td>235 Watts</td>
<td>250 Watts</td>
<td>300 Watts</td>
</tr>
<tr>
<td>Transistors</td>
<td>7.1 billion</td>
<td>8 billion</td>
<td>15.3 billion</td>
</tr>
<tr>
<td>GPU Die Size</td>
<td>551 mm²</td>
<td>601 mm²</td>
<td>610 mm²</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>28-nm</td>
<td>28-nm</td>
<td>16-nm FinFET</td>
</tr>
</tbody>
</table>
NVIDIA P100
**NVIDIA P100**

**Unified memory**
- Simplifies life for programmer: no host/device memory handling.
- Memory size limited by device memory.
- Performance?

**NVLink**
- Enhance inter-device bandwidth
- CPU-GPU or GPU-GPU
- 80 GB/s per lane, 4 lanes per device
Programming GPUs

CUDA:
• C extension to program an NVIDIA GPUs
• GPU- kernel is executed on the GPU multi-processors
• targeting highly-parallel algorithms

Abstraction layer:
• hierarchy of thread-blocks
• shared memory for communication between threads in the same block
• barrier synchronization
• scales up to hundreds of cores/thousands of threads

Kernel execution:
• kernel written as program for one thread
• all threads execute that code
• threads are gathered in thread blocks (up to 3D)
• every thread block is assigned to a multiprocessor (SM)
• **32 thread=1warp are executed in parallel**
• **1 warp always reads coalescent data from main memory**
• many thread-blocks can be active
• a kernel is executed by a (up to 3D) grid of thread blocks
• # of thread blocks executed in parallel depends on resources

```c
__global__ void kernel(int* a)
{
    a[blockIdx.x*blockDim.x + threadIdx.x]=0;
}

int main() {
    
    dim3 block(4);
    dim3 grid(n/block.x);
    kernel<<<grid,block>>>(d_a);
    
    return 0;
}
```
## Programming GPUs

<table>
<thead>
<tr>
<th>GPU</th>
<th>Kepler GK110</th>
<th>Maxwell GM200</th>
<th>Pascal GP100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>3.5</td>
<td>5.2</td>
<td>6.0</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps / Multiprocessor</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads / Multiprocessor</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks / Multiprocessor</td>
<td>16</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max 32-bit Registers / SM</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Block</td>
<td>65536</td>
<td>32768</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Thread</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Max Thread Block Size</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory Size / SM</td>
<td>16 KB/32 KB/48 KB</td>
<td>96 KB</td>
<td>64 KB</td>
</tr>
</tbody>
</table>
Programming GPUs

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>2.x</th>
<th>3.0</th>
<th>3.2</th>
<th>3.5</th>
<th>3.7</th>
<th>5.0</th>
<th>5.2</th>
<th>5.3</th>
<th>6.0</th>
<th>6.1</th>
<th>6.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of resident grids per device (Concurrent Kernel Execution)</td>
<td>16</td>
<td>4</td>
<td>32</td>
<td></td>
<td></td>
<td>16</td>
<td></td>
<td>128</td>
<td></td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x-dimension of a grid of thread blocks</td>
<td>65535</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum y- or z-dimension of a grid of thread blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>1536</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>32 K</td>
<td>64 K</td>
<td>128 K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread block</td>
<td>32 K</td>
<td>64 K</td>
<td>32 K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td>63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GPU kernel execution

Thread localication via unique IDs

threadIdx.x = 0 ... 11
threadIdx.y = 0 ... 1
blockDim.x = 12
blockDim.y = 2

blockIdx.x = 0 ... 2
blockIdx.y = 0 ... 1
gridDim.x = 3
gridDim.y = 2
GPU kernel execution

gridDim.x = 3
gridDim.y = 2

blockDim.x = 4
blockDim.y = 3

blockIdx.x = 0 ... 2
blockIdx.y = 0 ... 1

threadIdx.x = 0 ... 3
threadIdx.y = 0 ... 2
CUDA thread arrangement: Hands-on exercise

See kernel01.cu and kernel02.cu for more examples.

```c
int main() {
    ...
    int n = 12;
    dim3 block(4);
    dim3 grid(n/block.x);
    kernel<<<grid,block>>>(d_a);
    ...
}

__global__ void kernel(int* a){
    a[blockIdx.x*blockDim.x + threadIdx.x] = 7;
}

__global__ void kernel(int* a){
    a[blockIdx.x*blockDim.x + threadIdx.x] = blockIdx.x;
}

__global__ void kernel(int* a){
    a[blockIdx.x*blockDim.x + threadIdx.x] = threadIdx.x;
}
```
CUDA thread arrangement: Hands-on exercise

int main() {
    ...
    int n = 12;
    dim3 block(4);
    dim3 grid(n/block.x);
    kernel<<<grid,block>>>(d_a);
    ...
}

__global__ void kernel(int* a){
    a[blockIdx.x*blockDim.x + threadIdx.x] = 7;
}

__global__ void kernel(int* a){
    a[blockIdx.x*blockDim.x + threadIdx.x] = blockIdx.x;
}

__global__ void kernel(int* a){
    a[blockIdx.x*blockDim.x + threadIdx.x] = threadIdx.x;
}

Output: 77777777777777...

Output: 0000111122223333...

Output: 012301230123...

See kernel01.cu and kernel02.cu for more examples.
CUDA thread arrangement: Hands-on exercise

1D data of size N: linear arrangement of threads

- \( \text{dim3 block}(b,1,1), \text{grid}(g,1,1) \) with \( b \cdot g = N \)
- \( \text{idx} = \text{blockId.x} \cdot \text{blockDim.x} + \text{threadId.x} \)
- \( \text{kernel}<<<\text{grid}, \text{block}>>>(...) \) or \( \text{kernel}<<<\text{g}, \text{b}>>>(...) \)
CUDA thread arrangement: Hands-on exercise

1D data of size N: linear arrangement of threads
- dim3 block(b,1,1), grid(g,1,1) with b*g=N
- \(\text{id}x = \text{blockId.x*blockDim.x} + \text{threadId.x}\)
- kernel<<<grid, block>>>(...) or kernel<<<g, b>>>(...)

2D data of size NxM: rectangular arrangement of threads
- dim3 block=(b,c,1), grid=(G,H,1) with b*G=N and c*H=M
- \(\text{id}x = \text{blockId.x*blockDim.x} + \text{threadId.x}\) (global 2D)
- \(\text{idy} = \text{blockId.y*blockDim.y} + \text{threadId.y}\) (global 2D)
- \(\text{id} = \text{idy*N} + \text{id}x\) (global 1D)
- kernel<<<grid, block>>>(...)
GPU kernel performance: **SAXPY**  \( Y = a \cdot X + Y \)

**CPU (sequential)**

```c
void saxpy_serial(int n, float a, float *x, float *y){
    for (int i = 0; i < n; i++)    y[i] = a*x[i] + y[i];
}

saxpy_serial(n, 2.0, x, y);
```

**GPU (parallel)**

```c
__global__ void saxpy_parallel(int n, float a, float *x, float *y){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```
**GPU kernel performance: SAXPY** \( Y = a \times X + Y \)

**CPU (sequential)**

```c
void saxpy_serial(int n, float a, float *x, float *y){
    for (int i = 0; i < n; i++)    y[i] = a*x[i] + y[i];
}

saxpy_serial(n, 2.0, x, y);
```

**GPU (parallel)**

```c
__global__ void saxpy_parallel(int n, float a, float *x, float *y){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int nbblocks = (n + 255) / 256;
saxpy_parallel<<<nbblocks, 256>>>(n, 2.0, x, y);
```

- High parallelism.
- No dependencies.
- No communication.
- Every thread handles one vector entry.
- Coalescent memory access.
- No branching.
- Few operations on data!!!
GPU kernel performance: \textbf{SAXPY} \quad Y = a \times X + Y

Theoretical performance bounds

Compute bound?  
Memory-bound?

Arithmetic intensity: \( f = \frac{2}{3} \)  
Data format: IEEE Single Precision ( \( d_{SP} = 4 \text{ byte/vector entry} \) )

\begin{center}
\textbf{My GPU model: GT 750 M}
\begin{itemize}
  \item \( R_{\text{max}}: \) 1040 GFLOP/s
  \item \( L_{\text{max}}: \) 80 GB/s
\end{itemize}
\end{center}
GPU kernel performance: **SAXPY**  \[ Y = a^*X + Y \]

**Theoretical performance bounds**

- Compute bound?
- Memory-bound?

**Arithmetic intensity:** \( f = \frac{2}{3} \)

**Data format:** IEEE Single Precision  \( (d_{SP} = 4 \text{ byte/vector entry}) \)

**Memory bound:** \( \text{Perf} = \frac{L_{\text{max}}}{d} \times f \)

\[ (80 \text{ GB/s}) / (4 \text{ byte}) = 20 \times 10^9 \text{ vector-entries/s} \times 2/3 = 13.3 \text{ GFLOP/s} \]

**Experimental results even worse...**
GPU programming example

- 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array
GPU programming example

- 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array

- initial approach
  - 7 kernels, each adding one element to the sum
  - data always read from main memory

```c
__global__ void kernel_add(int n, int offset, int *a, int *b)
{
    int i = blockDim.x*blockIdx.x+threadIdx.x;
    int j = i + offset;
    if( j>-1 && j<n ){
        b[i]+=a[j];
    }
}
...
int nblocks = (n + 255) / 256;
kernel_add<<<nblocks, 256>>>(n, -3, a, b);
kernel_add<<<nblocks, 256>>>(n, -2, a, b);
kernel_add<<<nblocks, 256>>>(n, -1, a, b);
kernel_add<<<nblocks, 256>>>(n,  -0, a, b);
kernel_add<<<nblocks, 256>>>(n,   1, a, b);
kernel_add<<<nblocks, 256>>>(n,   2, a, b);
kernel_add<<<nblocks, 256>>>(n,   3, a, b);
```
GPU programming example

• 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array

• initial approach
  • 7 kernels, each adding one element to the sum
  • data always read from main memory

• better approach
  • merge the 7 kernels into one

```c
__global__ void kernel_add2(int n, int *a, int *b)
{
    int i = blockDim.x*blockIdx.x+threadIdx.x;
    if( i<n ){
        if(i>2)   b[i]+=a[i-3];
        if(i>1)   b[i]+=a[i-2];
        if(i>0)   b[i]+=a[i-1];
        b[i]+=a[i];
        if(i<n-3)  b[i]+=a[i+3];
        if(i<n-2)  b[i]+=a[i+2];
        if(i<n-1)  b[i]+=a[i+1];
    }
}
...
int nblocks = (n + 255) / 256;
kernel_add2<<<nblocks, 256>>>(n, a, b);
```
GPU programming example

- 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array

- initial approach
  - 7 kernels, each adding one element to the sum
  - data always read from main memory

- better approach
  - merge the 7 kernels into one

- even better
  - one thread reads needed data into shared memory
  - every thread-block computes blockDim.x partial sums
  - data read from shared memory

• blockDim.x +6 values needed
• Who loads the "boundary" to shared memory?
• Load imbalance...
GPU programming example

• 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array

• initial approach
  - 7 kernels, each adding one element to the sum
  - data always read from main memory

• better approach
  - merge the 7 kernels into one

• even better
  - one thread reads needed data into shared memory
  - every thread-block computes blockDim.x partial sums
  - data read from shared memory

• even better
  - every thread reads one entry into shared memory
  - every thread-block computes blockDim.x-6 partial sums
  - data read from shared memory

• thread-blocks need to “overlap”
• more thread blocks needed
GPU programming example

- 1D-partial sum
  - add entries of the adjacent array entries (radius of 3) of a 1D array

- initial approach
  - 7 kernels, each adding one element to the sum
  - data always read from main memory

- better approach
  - merge the 7 kernels into one

- even better
  - one thread reads needed data into shared memory
  - every thread-block computes blockDim.x partial sums
  - data read from shared memory

- even better
  - every thread reads one entry into shared memory
  - every thread-block computes blockDim.x-6 partial sums
  - data read from shared memory

```c
__global__ void kernel4(int n, int *a, int *b)
{
  int i = (blockDim.x-6)*blockIdx.x+threadIdx.x-3;
  int idx = threadIdx.x;
  __shared__ int values[256];
  int sum = 0;
  values[idx] = (i>1 && i<n) ? a[i] : 0;
  if( idx>2 && idx<256-3 ){
    for( int j=-3; j<4; j++ ) sum += values[ idx+j ];
    b[i]=sum;
  }
}
```
Memory management
Memory management

Main memory access

• 32 load/store units handle the main memory access.
• Each memory transaction can read or write up to 128 byte of contiguous memory.
• Working with 32-bit data types, this means that each memory transaction can read or write up to 32 values, if these values fall within the same 128B segment of memory.
• Otherwise, part of the bandwidth is unexploited.
GPU programming example: GEMV

Input A, x, Output y := alpha*A*x
• Part of the BLAS functionality
• Key routine for many iterative solvers e.g. Krylov subspace solvers
• Then often with sparse matrices, (SpMV)

```c
__global__ void sgemv_rowmajor(int n, float alpha, float *m, float *x, float *y)
{
    int row = blockIdx.x*blockDim.x + threadIdx.x;
    float sum = 0.0;
    if (row < n)
    {
        for( int col=0; col<n; col++)
        {
            sum += m[row*n + col] * x[cl];
        }
        y[row] = alpha * sum;
    }
}

int nbblocks = (n + 255) / 256;
sgemv_rowmajor<<<nbblocks, 256>>>(n, 2.0, x, y);
```
GPU programming example: GEMV

Input $A$, $x$, Output $y := \alpha A x$

- Part of the BLAS functionality
- Key routine for many iterative solvers e.g. Krylov subspace solvers
- Then often with sparse matrices, (SpMV)

```c
__global__ void sgemv_colmajor(int n, float alpha, float *m, float *x, float *y) {
    int row = blockIdx.x*blockDim.x + threadIdx.x;
    float sum = 0.0;
    if (row < n) {
        for (int col=0; col<n; col++) {
            sum += m[row + n*col] * x[col];
        }
        y[row] = alpha * sum;
    }
}

int nblocks = (n + 255) / 256;
sgemv_colmajor<<<nblocks, 256>>>(n, 2.0, x, y);
```
GPU programming example: Sparse GEMV

Input A, x, Output y := alpha*A*x
- A contains only few nonzeros
- Storing all entries would result in large overhead (memory & computation)

- **Idea:** Store only nonzero elements \([nz]\) explicitly:
  - Need to also store location, e.g. COO format: \([\text{value rowindex colindex}]\)

Memory footprint of COO:
\[nz(\text{val}) + 2*nz(\text{int})\]
GPU programming example: Sparse GEMV

Input $A$, $x$, Output $y := \alpha A x$

• $A$ contains only few nonzeros
• Storing all entries would result in large overhead (memory & computation)

• **Idea:** Store only nonzero elements $[nz]$ explicitly:
  • Need to also store location, e.g. COO format: [value rowindex colindex]

• Popular is the CSR format

\[
A = \begin{pmatrix}
5.4 & 1.1 & 0 & 0 & 0 & 0 \\
0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\
0 & 0 & 1.1 & 0 & 0 & 0 \\
0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\
9.0 & 0 & 0 & 1.1 & 4.5 & 0 \\
1.1 & 0 & 2.9 & 3.7 & 0 & 1.1 \\
\end{pmatrix}
\]

Memory footprint of COO: $nz(val) + 2*nz(int)$
Memory footprint of CSR: $nz(val) + nz(int) + (n+1) \times int$

rowptr: (0 2 5 6 9 12 16) Points to the first element in row

colind: (0 1 1 3 5 2 2 4 5 0 3 4 0 2 3 5) Column-index

values: (5.4 1.1 6.3 7.7 8.8 1.1 2.9 3.7 2.9 9.0 1.1 4.5 1.1 2.9 3.7 1.1) Values
GPU programming example: Sparse GEMV

### CSR-GEMV kernel

```c
for( row=0; row<n; row++ )
{
    sum = 0.0;
    for( j=rowptr[row]; j<rowptr[row+1]; j++)
        sum += values[ j ] * x[ colind[j] ];
    y[ row ] = alpha * sum;
}
```

- Small memory footprint
- Low computational cost
GPU programming example: Sparse GEMV

CSR-GEMV kernel

```
for( row=0; row<n; row++ )
{
    sum = 0.0;
    for( j=rowptr[row]; j<rowptr[row+1]; j++ )
        sum += values[ j ] * x[ colind[j] ];
    y[ row ] = alpha * sum;
}
```

- Small memory footprint
- Low computational cost
- Thread divergence
- Load imbalance
- Non-coalescent memory reads
GPU programming example: Sparse GEMV

Workaround: ELLPACK format

- Padding of the matrix with explicit zeros s.t. rows have same number of elements.
- No row-pointer necessary.

```c
for( row=0; row<n; row++ )
{
    sum = 0.0;
    for( j=0; j<max_nz_row; j++ )
        sum += values[ j ] * x[ colind[j] ];
    y[ row ] = alpha * sum;
}
```

- Coalescent memory access (if matrix is stored col-major).
- No thread divergence: all threads do same (number and) operations.
- Possibility to assign multiple threads to the same row (*if padded*):
  - Combine partial products with a reduction step in shared memory (SM) or registers
GPU programming example: Sparse GEMV

Workaround: ELLPACK format
- Padding of the matrix with explicit zeros s.t. rows have same number of elements.
- No row-pointer necessary.

```c
for( row=0; row<n; row++ )
{
    sum = 0.0;
    for( j=0; j<max_nz_row; j++)
        sum += values[ j ] * x[ colind[j] ];
    y[ row ] = alpha * sum;
}
```

- Coalescent memory access (if matrix is stored col-major).
- No thread divergence: all threads do same (number and) operations.
- Possibility to assign multiple threads to the same row (*if padded*):
  - Combine partial products with a reduction step in shared memory (SM) or registers

- Problem if one row is “dense”.
- Non-structured matrix pattern can result in significant overhead.
GPU programming example: Sparse GEMV

Workaround: Sliced ELLPACK (SELL / SELLP)
• Divide the matrix into blocks of rows.
• Blocksize can be adjusted to matrix.
• Each row-block is stored in ELLPACK format (col-major).
• The distinct blocks may have different elements in the row.
• Number of padding-elements in rows can be adjusted to be multiple of threads assigned to rows (SELL-P).
• Row-pointer points to first element in block.

• SELL-P allows to assign multiple threads to each row.
• Trade-off between CSR and ELLPACK

GPU programming example: Sparse GEMV

- 2nz+n
- Load imbalance
- Thread divergence
- Non-coalescent access

- 2*(max_nz_row*n)
- Coalescent access
- Potential overhead/breakdown
- Multiple threads per row

- Trade-off
- Coalescent access
- Multiple threads per row

GPU programming example: Sparse GEMV

SpMV (DP) on NVIDIA P100 (732 GB/s, 4.7 TFLOP/s)
Homework: Sparse GEMV

Write a CSR-SpMV GPU kernel

Document what you are doing!!!

1. Write your own CSR-SpMV GPU kernel.
2. Check correctness.
3. Measure the runtime and compare performance with a roofline model based on the memory bandwidth.
4. Extend the kernel to handle multiple vectors simultaneously (SpMM) (add a dimension to your thread-block, vector of dimension k*n).

You find the framework for the matrix I/O and the SpMV kernel in homework.zip on the webpage.

Project: Design your own SpMV

1. Make use of latest CUDA capabilities: atomic operations, shuffle ...
5. Test on selected Matrices and show better performance than CSR / ELL