Squeezing GPU performance

GPGPU 2015: High Performance Computing with CUDA
University of Cape Town (South Africa), April, 20th-24th, 2015

Manuel Ujaldón
Associate Professor @ Univ. of Malaga (Spain)
Conjoint Senior Lecturer @ Univ. of Newcastle (Australia)
CUDA Fellow @ Nvidia

NVIDIA
GPU peak performance vs. CPU

Peak Double Precision FLOPS  Peak Memory Bandwidth

GPU 6x faster on “double”:
- GPU: 3000 GFLOPS
- CPU: 500 GFLOPS

GPU 6x more bandwidth:
- 7 GHz x 48 bytes = 336 GB/s.
- 2 GHz x 32 bytes = 64 GB/s.
Let’s make a Malaga - Madrid travel (500 km)

**Effective time using the train:**
- Preliminaries: 3 minutes.
- Travel: 2 hours, 30 minutes.
- Closing: 2 minutes.
- TOTAL: 2 hours, 35 minutes.

**Effective time using the plane:**
- Preliminaries: 90 minutes.
- Travel: 50 minutes.
- Closing: 30 minutes.
- TOTAL: 2 hours, 50 minutes (and you are away from downtown!)
The real speed of my car

- **Maximum:**
  - 250 km/h.

- **Average on a 10 years use:**
  - 50 km/h.

- So I regularly use my car at 20% of peak performance. Should I be disappointed?
Instructions for the game available on the web site: http://cms.ac.uma.es/kepler

Welcome!

This practical tutorial provides you an easy way to interact with features of the GPUs based on the Kepler architecture, like:

- Ways to deploy massive parallelism via blocks, kernels and streams for Hyper-Q.
- Computing power (GFLOPS) and data bandwidth (GB/s).
- Latency for operands (int/float/double…) and operators (add/mul/div…).

You face different challenges here as CUDA programmer:

- Analyze performance for all the features described above, binding results to the GPU hardware for a better knowledge of its architecture.
- Investigate ways and mechanisms to get closer to the theoretical peak performance of a GPU.
- See how GFLOPS and bandwidth behave and are related to each other using the roofline model.

Enjoy the hands-on and... good luck with CUDA!
The simplest possible parallel program:

- Loops are parallelizable.
- Workload is known at compile-time.

```
for (i=0; i<N; i++)
  for (j=0; j<max(ElementsOnRow[i]); j++)
    convolution (i, j);
```

The simplest impossible parallel program:

- Workload is unknown at compile-time.
- The challenge is data partitioning.

```
for (i=0; i<N; i++)
  for (j=0; j<ElementsOnRow[i]; j++)
    convolution (i, j);
```

Poor solution #1: Oversubscription.
Poor solution #2: Serialization.
How you represent a sparse matrix in a Compressed Column Format

**Example for a 5x5 matrix:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>61</td>
<td>287</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>33</td>
<td>4 21</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>52</td>
<td>5 42</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Row indices**

**colptr**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

number of elements on each column (accumulated)

**value**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>75</td>
<td>52</td>
<td>61</td>
<td>11</td>
</tr>
<tr>
<td>33</td>
<td>42</td>
<td>21</td>
<td>52</td>
<td>11</td>
</tr>
<tr>
<td>4 21</td>
<td>33</td>
<td>42</td>
<td>21</td>
<td>87</td>
</tr>
<tr>
<td>0 + 3 + 1 + 2 + 1 + 2</td>
<td>3 4 6 7 9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

as traversed vertically

**rowidx**

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

horiz. position for each value

**Given the data structure, this is how you traverse matrix:**

```java
for (i=0; i<N; i++)
    for (j=colptr[i]; j<colptr[i+1]; j++)
        value[j] += value[j];
```
A challenge for CUDA programmers around the world: Performed on 8 countries so far

What the program does: Iterate in parallel on each element of a sparse matrix compressed by columns.

The sparse matrix may have $N=100$ or $N=200$ columns, each with a different number of nonzero elements. “numops” operations are performed on each element:

```plaintext
for (i=0; i<N; i++)
  for (j=colptr[i]; j<colptr[i+1]; j++)
    for (k=0; k<numops; k++)
      value[j] += value[j];
```

All loops are fully parallel.

Workload is unknown at compile-time.

The challenge is data partitioning:

- Deploy streams, kernels, blocks and threads wisely.
## Input sparse matrices
(taken from the Matrix Market collection)

<table>
<thead>
<tr>
<th>Application area</th>
<th>Matrix rows</th>
<th>Matrix columns</th>
<th>Nozeros</th>
<th>Workload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Economics</td>
<td>300</td>
<td>100</td>
<td>22,000</td>
<td>Base</td>
</tr>
<tr>
<td>Demography</td>
<td>6,000</td>
<td>100</td>
<td>440,000</td>
<td>20 x Base</td>
</tr>
<tr>
<td>Oceanography</td>
<td>24,000</td>
<td>100</td>
<td>1,760,000</td>
<td>160 x Base</td>
</tr>
<tr>
<td>Quantum physics</td>
<td>96,000</td>
<td>100</td>
<td>7,040,000</td>
<td>2560 x Base</td>
</tr>
<tr>
<td>Linear algebra</td>
<td>200</td>
<td>200</td>
<td>27,000</td>
<td>Base</td>
</tr>
<tr>
<td>Image processing</td>
<td>4,000</td>
<td>200</td>
<td>540,000</td>
<td>20 x Base</td>
</tr>
<tr>
<td>Astrophysics</td>
<td>32,000</td>
<td>200</td>
<td>4,320,000</td>
<td>160 x Base</td>
</tr>
<tr>
<td>Biochemistry</td>
<td>512,000</td>
<td>200</td>
<td>69,120,000</td>
<td>2560 x Base</td>
</tr>
</tbody>
</table>
You can try different operands and operators

What each thread does:
```c
int float double value[numelements];
for all elements assigned to each thread:
    for numops. to be done on each element
        value[i] *= value[i];
```

Changing the operator to lighter (addition) or heavier (division) will also have an impact depending on the time spent to perform each operation (its latency).
And you have to choose the winner parallelization strategy

Example strategy:
- We launch a CUDA kernel for each matrix column.
- Each kernel will have the lowest number of blocks.
- Each kernel will have the largest number of warps.
- Each thread will be as thin as possible (computes on a single elem.)

1: Thread-level parallelism (TLP)
2: Instruction-level parallelism (ILP)
3: Data parallelism (SIMD)
4: Vectorial (warp = 32)

Our code traverses the whole matrix, performing operations independently on each element.
The way we create streams. An example of 3 streams, each composed of 3 kernels

```c
__global__ kernel_A(pars) {body} // Same for B...Z
cudaStream_t stream_1, stream_2, stream_3;
...
cudaStreamCreateWithFlags(&stream_1, ...);
cudaStreamCreateWithFlags(&stream_2, ...);
cudaStreamCreateWithFlags(&stream_3, ...);
...
kernel_A <<< dimgridA, dimblockA, 0, stream_1 >>>> (pars);
kernel_B <<< dimgridB, dimblockB, 0, stream_1 >>>> (pars);
kernel_C <<< dimgridC, dimblockC, 0, stream_1 >>>> (pars);
...
kernel_P <<< dimgridP, dimblockP, 0, stream_2 >>>> (pars);
kernel_Q <<< dimgridQ, dimblockQ, 0, stream_2 >>>> (pars);
kernel_R <<< dimgridR, dimblockR, 0, stream_2 >>>> (pars);
...
kernel_X <<< dimgridX, dimblockX, 0, stream_3 >>>> (pars);
kernel_Y <<< dimgridY, dimblockY, 0, stream_3 >>>> (pars);
kernel_Z <<< dimgridZ, dimblockZ, 0, stream_3 >>>> (pars);
```
Top 10 optimizations performed by students

1. Increase the number of operations per element (1024).
2. Increase the sparse matrix size (up to 69M nonzeros).
3. Change the operator (add/sub/mul/div).
4. Change the operand (int/float/double).
5. Tune the CUDA block size (384 threads per block).
6. Group blocks in kernels and those in streams to express more parallelism.
7. Optimize memory access using shared memory and regs.
8. Guide the compiler via #pragma unroll directives.
9. Enable the fused multiply-add operator.
10. Use vector instructions to exploit (x,y,z,w) and (r,g,b,a).
### Performance attained on a GeForce GTX480 (peak performance 1330 GFLOPS on 32-bit)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Acceler.</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Departure point</td>
<td></td>
<td>0.0008 GFLOPS</td>
</tr>
<tr>
<td>1. Increase the number of operations per element (up to 1024)</td>
<td>250.00 x</td>
<td>0.20 GFLOPS</td>
</tr>
<tr>
<td>2. Use a bigger sparse matrix (up to 69,120,000 nonzeros)</td>
<td>116.35 x</td>
<td>23.27 GFLOPS</td>
</tr>
<tr>
<td>3. Choose the sum operator (add)</td>
<td>1.00 x</td>
<td>23.27 GFLOPS</td>
</tr>
<tr>
<td>4. Replace the double operand (64-bits) by float (32-bit)</td>
<td>1.89 x</td>
<td>44.00 GFLOPS</td>
</tr>
<tr>
<td>5. Tune the block size (384 threads)</td>
<td>1.00 x</td>
<td>44.00 GFLOPS</td>
</tr>
<tr>
<td>6. Group kernels in streams</td>
<td>1.00 x</td>
<td>44.00 GFLOPS</td>
</tr>
<tr>
<td>7. Optimize memory accesses using shared memory and registers</td>
<td>3.19 x</td>
<td>140.75 GFLOPS</td>
</tr>
<tr>
<td>8. Unroll the loop via a #pragma unroll compiler directive</td>
<td>4.07 x</td>
<td>573.95 GFLOPS</td>
</tr>
<tr>
<td>9. Enable the FMADD (fused multiply-add) operator</td>
<td>2.15 x</td>
<td>1236.58 GFLOPS</td>
</tr>
<tr>
<td>10. Enable vector processing on computational sentences (4 in 1)</td>
<td>1.00 x</td>
<td>1236.58 GFLOPS</td>
</tr>
<tr>
<td>1.2 Saturate the number of operations (up to 1M)</td>
<td>1.02 x</td>
<td>1260.00 GFLOPS</td>
</tr>
<tr>
<td>8.2 Saturate the loop unroll factor (until 4096)</td>
<td>1.01 x</td>
<td>1280.00 GFLOPS</td>
</tr>
<tr>
<td>2.2 Generate a huge matrix to exploit GPU scalability</td>
<td>1.02 x</td>
<td>1310.00 GFLOPS</td>
</tr>
<tr>
<td>2.3 Tune the matrix to match the structure of CUDA parallelism</td>
<td>1.01 x</td>
<td>1330.00 GFLOPS</td>
</tr>
</tbody>
</table>