Further Details on Dense Linear Algebra in CUDA

Oliver Meister

November 13th 2012
Last Tutorial

CUDA Architecture

- thread hierarchy: threads, warps, blocks, grids
- memory hierarchy: local memory, shared memory, global memory, host memory
- GPU: CUDA cores and load/store units with registers, SM with warp scheduler and L1 cache/shared memory, GPU with L2 Cache, giga thread engine and global memory
Last Tutorial

CUDA API

- host code – executed on CPU
  - memory operations
  - grid and block size definition
  - kernel calls
- device code – executed on GPU
  - execution of lightweight kernels
  - memory-based hierarchical communication between kernels
Dense Matrix Multiplication

Simple Implementation

- max. size of 16

Separation into blocks

- arbitrary matrix size
- still access to global memory only

Using Tiles

- load data into fast shared memory
- computation on shared tile
Assignment 1 - Make it run

- Trouble with our machine/CUDA? Contact or visit me
- My office: LRZ Room E.2.040
  - Enter LRZ main entrance, go down stairs on the left
  - Follow signs to LRZ Building 2 through hallway with black walls and colored windows.
  - Go up to second floor and to the end of the corridor.
  - I’m in the third-last office on the left.
- Contact me before visiting if you want to be sure I’m here
Everybody got that right :)  

Some minor issues I encountered:  

- $\text{ceil}(n/\text{TILE\_SIZE})$  
- \text{TILE\_SIZE} too big? Edit \text{cuda\_mmult\_kernels.h} if you want to change the value  
- \text{cuda\_Memcpy} is always called from host code in order to copy between host memory (RAM) and device memory (global memory), NEVER in device code.
__global__ void matrixMultKernel(float* Ad, float* Bd, float* Cd, int n) {
    __shared__ float Ads[TILE_SIZE][TILE_SIZE];
    __shared__ float Bds[TILE_SIZE][TILE_SIZE];
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int i = blockIdx.x * TILE_SIZE + tx;
    int k = blockIdx.y * TILE_SIZE + ty;
    /* (cont.) */
Assignment 4 - Tiled Matrix Multiplication

/* (cont.) */
for(int m=0; m < n/TILE_SIZE; m++) {
    Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
    Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
    __syncthreads();
    /* perform matrix multiplication on shared tiles */
    for(int j=0; j<TILE_SIZE; j++)
        Celem += Ads[tx][j]*Bds[j][ty];
    __syncthreads();
}
Cd[i*n+k] += Celem;
Assignment 4 - Tiled Matrix Multiplication

/* (cont.) */

for (int m=0; m < n/TILE_SIZE; m++) {
    Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
    Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
    __syncthreads();
    /* perform matrix multiplication on shared tiles */
    for (int j=0; j<TILE_SIZE; j++)
        Celem += Ads[tx][j]*Bds[j][ty];
    __syncthreads();
}
Cd[i*n+k] += Celem;

Let's see what happens.
Assignment 4 - Tiled Matrix Multiplication

/* (cont.) */
for(int m=0; m < n/TILE_SIZE; m++) {
    Ads[tx][ty] = Ad[i*n + m*TILE_SIZE+ty];
    Bds[tx][ty] = Bd[(m*TILE_SIZE+tx)*n + k];
    __syncthreads();
    /* perform matrix multiplication on shared tiles */
    for(int j=0; j<TILE_SIZE; j++)
        Celem += Ads[tx][j]*Bds[j][ty];
    __syncthreads();
}
Cd[i*n+k] += Celem;

Let's see what happens. ⇒ Better, but not great.
### Assignment 3c, 4e - Performance measurements

<table>
<thead>
<tr>
<th>Matrix size n</th>
<th>MFlop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td></td>
</tr>
<tr>
<td>16384</td>
<td></td>
</tr>
</tbody>
</table>

Floating point performance by kernel (tile size):

- cpu
- global (4)
- global (8)
- global (16)
- global (32)
- tiled (4)
- tiled (8)
- tiled (16)
- tiled (32)
Assignment 3c, 4e - Performance measurements

Floating point performance by kernel(tile size)

Matrix size n

MFlop/s

cpu

global (4)

global (8)

tiled (4)

global (16)

global (32)

tiled (8)

tiled (16)

tiled (32)
Updated Performance Estimate

- m-loop: each thread loads one matrix element from global memory
- j-loop: shared memory $\rightarrow$ no further loads in TILE_SIZE iterations
- we have reduced the memory transfer from global memory to $1/TILE\_SIZE$ of the original code
- for $TILE\_SIZE = 32$: new performance limit at 128 GFlop/s
  $\rightarrow$ we’ve eliminated a major bottleneck, but apparently hit another . . .
## CUDA Profiler

### Counters (Selection)

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occupancy</td>
<td>number of active warps / max. number of active warps</td>
</tr>
<tr>
<td>Branch</td>
<td>number of branches, that might split a warp</td>
</tr>
<tr>
<td>Warp serialize</td>
<td>Serialization of a warp due to branch / memory access</td>
</tr>
<tr>
<td>gld / gst un-coalesced</td>
<td>global memory operations which serialize the warps</td>
</tr>
<tr>
<td>Instruction throughput</td>
<td>achieved instruction rate compared to the peak instruction rate</td>
</tr>
</tbody>
</table>
Global Memory Architecture

- DRAM (i.e., global memory) built, such that multiple contiguous memory slots are read together (compare: cache lines)

Warp Serialize

- access to global memory requires serialization
- each thread gets its own piece of memory and not consecutive entries

leads to uncoalesced memory access, since a single warp (max. 16 threads) can only execute a single instruction per cycle. When access to global memory is not coalesced, it is serialized.
Coalesced Memory Access

- necessary for maximal bandwidth
- neighboring threads in a warp should access neighboring memory addresses
- not all threads have to participate
- have a valid starting address
- have the right order
- strides are allowed but the speed is reduced significantly
- have to have the right order
- no double accesses

exact compute pattern is depending on the compute capability
Compute Capability

Different Compute Capabilities of devices (1.x to 2.x)

- different technical specifications (caches, number of SMs ...)
- different treatment of
  - global memory
  - shared memory
### Aligned and non-sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threads:</th>
<th>0</th>
<th>...</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compute capability:</th>
<th>1.0 and 1.1</th>
<th>1.2 and 1.3</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Uncached</td>
<td>Cached</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 128</td>
<td>1 x 64B at 128</td>
<td>1 x 128B at 128</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 192</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Misaligned and sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute capability:</td>
<td>1.0 and 1.1</td>
<td>1.2 and 1.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 x 32B at 128</td>
<td>1 x 128B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 192</td>
<td>1 x 32B at 256</td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Coalesced Access in Matrix Multiplication

Global memory access:

\[
\begin{align*}
\text{Ads}[tx][ty] &= \text{Ad}[ i \times n + m \times \text{TILE\_SIZE} + ty]; \\
\text{Bds}[tx][ty] &= \text{Bd}[ (m \times \text{TILE\_SIZE} + tx) \times n + k];
\end{align*}
\]

- row computation: \( i = \text{blockIdx}.x \times \text{TILE\_SIZE} + tx \)
- column computation: \( k = \text{blockIdx}.y \times \text{TILE\_SIZE} + ty \)

⇒ stride-1 access w.r.t. \( ty \), stride-\( n \) access w.r.t. \( tx \)
Coalesced Access in Matrix Multiplication

Shared memory access:

```c
for(int j=0; j<TILE_SIZE; j++)
    Celem += Ads[tx][j]*Bds[j][ty];
```

- As in C, matrix layout is row-wise
- stride-TILE_SIZE access to Ads by tx
- stride-1 access to Bds by ty
Question: How are threads ordered in a block?

Combination of threads into warps:

- **1D thread block:** thread 0, ..., 31 into warp 0; thread 32, ..., 63 into warp 1; etc.
- **2D thread block:** x-dimension is "faster-running"; e.g., dimBlock(8,8,1), i.e., 64 threads (2 warps)
  - then threads (0,0), ..., (7,3) are in warp 0
  - and threads (0,4), ..., (7,7) are in warp 1
- **3D thread block:** x, then y, then z
Question: How are threads ordered in a block?

Combination of threads into warps:

- **1D thread block**: thread 0, ..., 31 into warp 0; thread 32, ..., 63 into warp 1; etc.
- **2D thread block**: x-dimension is “faster-running”; e.g.:
  - \texttt{dimBlock}(8, 8, 1), i.e., 64 threads (2 warps)
  - then threads (0,0), ..., (7,3) are in warp 0
  - and threads (0,4), ..., (7,7) are in warp 1
- **3D thread block**: x, then y, then z
Coalesced Access in Matrix Multiplication

Task:

- Examine our tiled matrix multiplication, are global and shared memory access coalesced?
- If not, what has to be changed?
- Give an upper bound for the performance increase between uncoalesced an coalesced access
Matrix Multiplication with Tiling

Original algorithm:

```c
int i = blockIdx.x * TILE_SIZE + tx;
int k = blockIdx.y * TILE_SIZE + ty;
float Celem = 0;
for(int m=0; m < n/TILE_SIZE; m++) {
    Ads[tx][ty] = Ad[ i*n + m*TILE_SIZE+ty];
    Bds[tx][ty] = Bd[ (m*TILE_SIZE+tx)*n + k];
    _syncthreads();
    for(int j=0; j<TILE_SIZE; j++)
        Celem += Ads[tx][j]*Bds[j][ty];
    _syncthreads();
}
Cd[i*n+k] += Celem;
```
Matrix Multiplication with Coalesced Access

Switch \(x\) and \(y\) \(\Rightarrow\) stride-1 read access to \(A_d, B_d, B_{ds}\), stride-1 write access to \(A_{ds}, B_{ds}, C_d\):

```c
int i = blockIdx.y * TILE_SIZE + ty;
int k = blockIdx.x * TILE_SIZE + tx;
float Celem = 0;
for(int m=0; m < n/TILE_SIZE; m++) {
    Ads[ty][tx] = Ad[ i*n + m*TILE_SIZE+tx];
    Bds[ty][tx] = Bd[ (m*TILE_SIZE+ty)*n + k];
    __syncthreads();
    for(int j=0; j<TILE_SIZE; j++)
        Celem += Ads[ty][j]*Bds[j][tx];
    __syncthreads();
}
Cd[i*n+k] += Celem;
```
Performance measurements

Floating point performance by kernel (tile size)

Matrix size n

MFlop/s

cpu
tiled (4)
tiled (8)
tiled (16)
tiled (32)
clsced (4)
clsced (8)
clsced (16)
clsced (32)
Roofline model

A roofline model (Williams, Waterman, Patterson ’08) is a helpful tool to model performance issues for an HPC code. It relates floating point performance, memory performance and operational intensity (operations per memory fetch) in a 2D graph.

Task:

• Draw a roofline model for the NVidia Quadro NVS 290 (peak performance: 44.16 GFlop/s, 6.4 GB/sec).
• Include vertical lines for the basic and tiled matrix multiplication (TILE_SIZE = 16) and insert points with performance measurements.
• Add a ceiling for coalescing (CUDA cc 1.1: ≤ 64 B per fetch)
→ Is basic MM coalesced?
Roofline: matrix multiplication on NVS 5200M

→ Is basic MM coalesced? No, L1 Cache!
Memory Latency for Tile Transfers

Recapitulate tiled matrix multiplication:

- tiles of $16 \times 16$ matrix elements
  $\rightarrow 16^2 = 256$ threads per tile (also per thread block)
- thus: 8 warps (32 threads each)
- examine load operation for matrix tiles

  $\text{Ads}[ty][tx] = \text{Ad}[ i*n + m*\text{TILE\_SIZE}+tx] ;$
  $\text{Bds}[ty][tx] = \text{Bd}[ (m*\text{TILE\_SIZE}+ty)*n + k] ;$

  $\text{__syncthreads}() ;$

  $\rightarrow$ delay due to memory latency

- all threads in a warp wait for data to arrive
- but another warp can be scheduled to work
Tiled Matrix Multiplication with Prefetching

Include prefetching of blocks to reduce “idle” time for memory transfer:

1. load first tile into register(s)
2. copy register(s) to shared memory
3. barrier
4. load next tile into register(s)
5. compute current tile
6. barrier
7. proceed with 2 (if there are more tiles)
8. compute last tile

→ Homework!