Further Details on Dense Linear Algebra in CUDA

Oliver Meister

November 6th 2014
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
H1.1 - Make it run

- Trouble with the cluster/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
Remarks:

• $\text{ceil}(n/\text{TILE\_SIZE})$
  good idea, but what can we do better here?

• Where to change $\text{TILE\_SIZE}$? `cuda\_mmult\_kernels.h`
  Or: remove definition in the file and define macro in
  `Makefile` using `nvcc -DTILE\_SIZE=32 <...>

• `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];

    /* (cont.) */
int tx = threadIdx.x; int ty = threadIdx.y;
int i = blockIdx.x * TILE_SIZE + tx;
int k = blockIdx.y * TILE_SIZE + ty;

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;
}
H1.2b, H1.3b - Performance measurements

What are your results?
What are your results?

FP performance on NVS 5200M by kernel(tile size)

Matrix size n

MFlop/s

cpu
global (4)
global (8)
global (16)
global (32)
tiled (4)
tiled (8)
tiled (16)
tiled (32)
H1.2b, H1.3b - Performance measurements

What are your results?

FP performance on Tesla M2090 by kernel(tile size)

Matrix size n

MFlop/s

32  64  128  256  512  1024

Matrix size n

cpu    global (4)  global (8)  global (16)  global (32)  tiled (4)  tiled (8)  tiled (16)  tiled (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 M2090 (peak performance: 1331 GFlop/s, 177.0 GB/s).
- Find computational intensity of basic and tiled matrix multiplication (TILE_SIZE = 4, 8, 16, 32) and insert points with performance measurements.
Roofline model

Roofline: matrix multiplication on NVS 5200M
What did we miss?
Roofline model

Roofline: matrix multiplication on Tesla M2090

What did we miss? L1 Cache!
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 global kernel
- for TILE_SIZE = 32: new performance limit at 1331 GFlop/s
  $\rightarrow$ we’ve eliminated a major bottleneck, but apparently hit another...
CUDA Profiler

### Kernel analysis

<table>
<thead>
<tr>
<th>Compute</th>
<th>branches, warp serialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth</td>
<td>bandwidth limitations, load/store sizes</td>
</tr>
<tr>
<td>Latency</td>
<td>occupancy</td>
</tr>
</tbody>
</table>
Coalesced Memory Access

Global Memory Architecture

- DRAM (i.e., global memory): multiple contiguous memory slots are read simultaneously (compare: cache lines)

Warp Serialize

- access to global memory can be merged, if nearby memory locations are accessed concurrently
- all threads in a warp access memory concurrently – if data is located in consecutive memory: merge memory calls

A single (half-)warp can only execute a single instruction per cycle ⇒ Serialization, uncoalesced access.
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
- strides are allowed but the speed is reduced significantly
- have to be in 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

![Diagram of aligned and sequential addresses with compute capability and memory transactions]

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</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></td>
<td>1 x 64B at 128</td>
<td>1 x 64B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>
</tbody>
</table>
### 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>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<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>Cached</td>
<td></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>
</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></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 x 32B at 128</td>
<td>1 x 128B 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>1 x 128B at 256</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 192</td>
<td>1 x 32B at 256</td>
<td>1 x 128B at 256</td>
</tr>
<tr>
<td></td>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Global memory access:

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

- row computation: \( i = \text{blockIdx}.x * \text{TILE\_SIZE} + tx \)
- column computation: \( k = \text{blockIdx}.y * \text{TILE\_SIZE} + ty \)
⇒ 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];
```

- Matrix layout in CUDA is *row-wise* (same as C/C++)
- stride-TILE_SIZE access to $A_{ds}$ by $tx$
- stride-1 access to $B_{ds}$ by $ty$
Question: How are threads ordered in a block?

Combination of threads into warps:

1. 1D thread block: thread 0, ..., 31 into warp 0; thread 32, ..., 63 into warp 1; etc.

2. 2D thread block: x-dimension is “faster-running”; e.g.:
   - dimBlock(8,8), i.e., 64 threads (2 warps)
   - threads (0,0), (7,0), (0,3), (7,3) are in warp 0
   - threads (0,4), (7,4), (0,7), (7,7) are in warp 1

3. 3D thread block: x, then y, then z
Warps and Coalesced Access

Question: How are threads ordered in a block?

Combination of threads into warps:

- 1D thread block: thread 0, \ldots 31 into warp 0; thread 32, \ldots 63 into warp 1; etc.
- 2D thread block: x-dimension is “faster-running”; e.g.:
  - \texttt{dimBlock(8,8)}, i.e., 64 threads (2 warps)
  - threads (0,0), (7,0), (0,3), (7,3) are in warp 0
  - threads (0,4), (7,4), (0,7), (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?
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 $A_d$, $B_d$, $B_d$, stride-1 write access to $A_d$, $B_d$, $C_d$:

```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 (v2)

Let’s test the change!

<table>
<thead>
<tr>
<th>Matrix size n</th>
<th>FP performance on NVS 5200M by kernel (tile size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>cpu tiled (4)</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>
<tr>
<td>32768</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
</tr>
<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>
<tr>
<td>32768</td>
<td></td>
</tr>
</tbody>
</table>
Performance measurements (v2)

Let’s test the change!

FP performance on NVS 5200M by kernel(tile size)
Performance measurements (v2)

Let’s test the change!

FP performance on Tesla M2090 by kernel(tile size)
Roofline model

Task:

- Add a ceiling for coalescing to the roofline model (CUDA cc 1.2: $\leq 128$ B per fetch).
- Insert points with the performance measurements of the coalesced kernel.
Roofline model (v2)

Roofline: matrix multiplication on NVS 5200M

Oliver Meister: Further Details on Dense Linear Algebra in CUDA
Tutorial Parallel Programming and High Performance Computing, November 6th 2014
Roofline model (v2)

Roofline: matrix multiplication on Tesla M2090

Oliver Meister: Further Details on Dense Linear Algebra in CUDA
Tutorial Parallel Programming and High Performance Computing, November 6th 2014
Memory Latency for Tile Transfers

Recapitulate tiled matrix multiplication:

- tiles of $32 \times 32$ matrix elements
  $\rightarrow 32^2 = 1024$ threads per tile (also per thread block)
- thus: 32 warps (32 threads each) on 8 slots
- examine load operation for matrix tiles

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

- due to memory latency global memory access takes multiple cycles
- warp scheduler underloaded when threads read data
- meanwhile other warps could do some 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!