CUDA ADVANCED
Contents

▶ Data Transfer Optimization

▶ Kernel Optimization
  – Thread ID & Warps
  – Impact of Branches
  – Global Memory Accesses
  – Constant Memory
  – Shared Memory
  – Matrix Transpose Example
  – Minimize Global Memory Accesses
  – Increasing GPU Efficiency
DATA TRANSFER OPTIMIZATION
DATA TRANSFERS

- Most of the time the main issue for performance
  - need to minimize their impact on the application

- Common sense:
  - transfer only useful data
  - batch small transfers into larger ones:
    • Reduce data transfer initialization cost
    • Best PCIe BW utilization = total latency lowered
DATA TRANSFERS

- Port on GPU non compute intensive parts of the code if it can avoid transfer

Functionalities
- Use pinned memory
- Overlap data transfers & kernel execution (streams)
  - Memory copies host to device of a memory block of 64 KB or less are asynchronous
Pinned Memory

When data transfer from **pageable host memory** to device memory is invoked, the driver:
- allocate a temporary page-locked host array (probably just once)
- copy the host data to the pinned array
- transfer data from pinned array to device memory

```c
A = malloc(n * sizeof(int))
...
free(A)
```
Pinned Memory

When data transfer from **pinned memory** to device memory is invoked
  - allocate a temporary page-locked host array
  - copy the host data to the pinned array
  - transfer data from pinned array to device memory

Use carefully
Thread ID & Warps

31-10-2016
Thread ID

The ID of a thread within a block is always:

\[
tid = threadIdx.x + threadIdx.y \times (blockDim.x) + threadIdx.z \times (blockDim.x \times blockDim.y)
\]

- \(threadIdx.x, y, z\): index of the thread in its x, y, z dimension
- \(blockDim.x, y, z\): size of the block in its x, y, z dimension

Example:
- Considering a block with: \(blockDim.x = 8, blockDim.y = 6, blockDim.z = 4\)
- the thread with: \(threadIdx.x = 1, threadIdx.y = 2, threadIdx.z = 3\)
- has the ID: \(1 + 2 \times (8) + 3 \times (6 \times 8) = 161\)
WARP

The warp is the base unit of execution on the GPU multiprocessor:
- warps are subdivision of a block of threads
- streaming multiprocessors have warp schedulers
  - Each threads of a warp launch 1 instruction by cycle

All aspects of kernel performance are driven by warps:
- memory access
  - global, constant and shared
- branch issues
- instruction level parallelism (ILP)
- occupancy

A warp is composed of 32 threads of consecutive increasing ID
- the first warp starts with the thread of ID 0
- a warp scheduler launches 32 instructions by cycle (one by thread)
WARP: Example 1

- Warps inside thread blocks of size 32x4:
  - 4 warps by block

```c
dim3 dimBlock;
dimBlock.x = 32;
dimBlock.y = 4;
```
**WARP: Example 2**

- Warps inside thread blocks of size 16x4:
  - 2 warps by block

```
dim3 dimBlock;
dimBlock.x = 16;
dimBlock.y = 4;
```
Impact of Branches
BRANCHES

- Branches: if, while, ...

- Branches are not an issue if all threads within a same warp take the same branch:

- Having threads within a same warp doing different things leads to serialization!
  - Worth scenario: all threads in the warp take different branches
Branches: Example 1

Thread blocks of size 32x4:
- 4 warps by block

```c
__global__ void myKernel(...){ //no serialization
    int id = threadIdx.x + threadIdx.y * blockDim.x
            + threadIdx.z * blockDim.x * blockDim.y;

    if(threadIdx.y % 2 == 0){
        do 1
    } else{
        do 2
    }
}
```

```c
__global__ void myKernel(...){ //serialization
    int id = threadIdx.x + threadIdx.y * blockDim.x
            + threadIdx.z * blockDim.x * blockDim.y;

    if(threadIdx.x < 16){
        do 1
    } else{
        do 2
    }
}
```
Branches: Example 2

- Thread blocks of size 8x4:
  - 4 warps by block

```
//serialization
__global__ void myKernel(...){
  int id = threadIdx.x + threadIdx.y * blockDim.x + threadIdx.z * blockDim.x * blockDim.y;
  if(threadIdx.y % 2 == 0){
    do 1
  }else{
    do 2
  }
}

//no serialization
__global__ void myKernel(...){
  int id = threadIdx.x + threadIdx.y * blockDim.x + threadIdx.z * blockDim.x * blockDim.y;
  if(threadIdx.y < 2){
    do 1
  }else{
    do 2
  }
}
```
Global Memory Accesses

31-10-2016
Global Memory Accesses

- Global memory is accessed via 32 or 128 bytes memory transactions

- These memory transactions must be naturally aligned:
  - Only the 32 or 128 byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions

- When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of its threads into one or more of these memory transactions:
  - depending on:
    • the size of the word accessed by each thread
    • the distribution of the memory addresses across the threads
Global Memory Accesses

- Global memory is cache in L2
- Each multiprocessor has a L1
- Cache: data will be cached in L1. This policy depends on:
  - architecture (2.x, 3.x, ...)
  - compilation options
  - (see documentation)
**Copy Kernel: Mis-Aligned Accesses Effect**

```c
__global__ void transpose_global(int offset, const int* __restrict__ in, int* out){
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    out[idx + offset] = in[idx + offset];
}
```

<table>
<thead>
<tr>
<th>offset (1= 4 Bytes)</th>
<th>time in µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>1078.656</strong></td>
</tr>
<tr>
<td>1</td>
<td>1174.720</td>
</tr>
<tr>
<td>2</td>
<td>1171.520</td>
</tr>
<tr>
<td>3</td>
<td>1175.456</td>
</tr>
<tr>
<td>4</td>
<td>1174.336</td>
</tr>
<tr>
<td>5</td>
<td>1174.560</td>
</tr>
<tr>
<td>6</td>
<td>1172.992</td>
</tr>
<tr>
<td>7</td>
<td>1176.320</td>
</tr>
<tr>
<td>8</td>
<td>1164.416</td>
</tr>
<tr>
<td>9</td>
<td>1185.824</td>
</tr>
<tr>
<td>10</td>
<td>1187.744</td>
</tr>
<tr>
<td>11</td>
<td>1185.696</td>
</tr>
<tr>
<td>12</td>
<td>1186.144</td>
</tr>
<tr>
<td>13</td>
<td>1185.440</td>
</tr>
<tr>
<td>14</td>
<td>1185.696</td>
</tr>
<tr>
<td>15</td>
<td>1187.360</td>
</tr>
<tr>
<td>16</td>
<td>1165.792</td>
</tr>
<tr>
<td>17</td>
<td>1184.992</td>
</tr>
<tr>
<td>18</td>
<td>1185.280</td>
</tr>
<tr>
<td>19</td>
<td>1184.416</td>
</tr>
<tr>
<td>20</td>
<td>1183.264</td>
</tr>
<tr>
<td>21</td>
<td>1184.704</td>
</tr>
<tr>
<td>22</td>
<td>1185.024</td>
</tr>
<tr>
<td>23</td>
<td>1186.400</td>
</tr>
<tr>
<td>24</td>
<td>1167.808</td>
</tr>
<tr>
<td>25</td>
<td>1186.688</td>
</tr>
<tr>
<td>26</td>
<td>1187.200</td>
</tr>
<tr>
<td>27</td>
<td>1184.768</td>
</tr>
<tr>
<td>28</td>
<td>1186.976</td>
</tr>
<tr>
<td>29</td>
<td>1187.840</td>
</tr>
<tr>
<td>30</td>
<td>1189.472</td>
</tr>
<tr>
<td>31</td>
<td>1186.080</td>
</tr>
<tr>
<td>32</td>
<td><strong>1068.224</strong></td>
</tr>
</tbody>
</table>
Copy Kernel: Stride Effect

```c
__global__ void transpose_global(int offset, const int* __restrict__ in, int *out) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    out[idx * stride] = in[idx * stride];
}
```

<table>
<thead>
<tr>
<th>stride</th>
<th>time in µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1091.648</td>
</tr>
<tr>
<td>2</td>
<td>2694.624</td>
</tr>
<tr>
<td>3</td>
<td>3974.112</td>
</tr>
<tr>
<td>4</td>
<td>5305.536</td>
</tr>
<tr>
<td>5</td>
<td>6723.072</td>
</tr>
<tr>
<td>6</td>
<td>8128.608</td>
</tr>
<tr>
<td>7</td>
<td>9554.144</td>
</tr>
<tr>
<td>8</td>
<td>10944.320</td>
</tr>
<tr>
<td>9</td>
<td>11256.000</td>
</tr>
<tr>
<td>10</td>
<td>11547.104</td>
</tr>
<tr>
<td>11</td>
<td>11819.072</td>
</tr>
<tr>
<td>12</td>
<td>12184.064</td>
</tr>
<tr>
<td>13</td>
<td>12530.112</td>
</tr>
<tr>
<td>14</td>
<td>12919.040</td>
</tr>
<tr>
<td>15</td>
<td>13300.896</td>
</tr>
<tr>
<td>16</td>
<td>13354.432</td>
</tr>
<tr>
<td>17</td>
<td>14070.464</td>
</tr>
<tr>
<td>18</td>
<td>14800.096</td>
</tr>
<tr>
<td>19</td>
<td>16281.440</td>
</tr>
<tr>
<td>20</td>
<td>17069.568</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stride</th>
<th>time in µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>17948.545</td>
</tr>
<tr>
<td>22</td>
<td>18972.928</td>
</tr>
<tr>
<td>23</td>
<td>19875.199</td>
</tr>
<tr>
<td>24</td>
<td>20824.256</td>
</tr>
<tr>
<td>25</td>
<td>21818.207</td>
</tr>
<tr>
<td>26</td>
<td>22839.297</td>
</tr>
<tr>
<td>27</td>
<td>23967.936</td>
</tr>
<tr>
<td>28</td>
<td>24991.039</td>
</tr>
<tr>
<td>29</td>
<td>26094.176</td>
</tr>
<tr>
<td>30</td>
<td>27074.623</td>
</tr>
<tr>
<td>31</td>
<td>28111.424</td>
</tr>
<tr>
<td>32</td>
<td>28460.607</td>
</tr>
<tr>
<td>33</td>
<td>29365.248</td>
</tr>
<tr>
<td>34</td>
<td>29742.432</td>
</tr>
<tr>
<td>35</td>
<td>29933.119</td>
</tr>
<tr>
<td>36</td>
<td>30344.928</td>
</tr>
<tr>
<td>37</td>
<td>30212.352</td>
</tr>
<tr>
<td>38</td>
<td>30526.912</td>
</tr>
<tr>
<td>39</td>
<td>30723.584</td>
</tr>
</tbody>
</table>
Constant Memory

31-10-2016
Constant Memory

- The constant memory space resides in device memory
  - size 64kB

- Each multiprocessor has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space

- When threads within a warp perform a request to the constant memory:
  - the request is split into as many separate requests as there are different memory addresses
  - the resulting requests are then serviced:
    • at the throughput of the constant cache in case of a cache hit
    • at the throughput of device memory otherwise
Example

Warp 0:
- threads access the same memory address: served in 1 request

Warp 2:
- threads access to 2 different memory addresses: served in 2 requests
Shared Memory

31-10-2016
Shared Memory

- The shared memory is a memory on a multiprocessor (on-chip)
  - small: usually 48 KB
- This memory is shared between threads of a same thread block
  - if thread 1 from block 3 writes in shared memory:
    • all threads from block 3 can access the new data
    • threads from other blocks can not

- The shared memory:
  - is faster than the global memory
    • registers: ~ few cycles
    • shared memory: ~ dozens of cycles
    • global memory: ~ hundreds cycles
  - can be seen as a cache driven by the user

- Can be declared statically inside the kernel or dynamically at the kernel call.
Banks

To achieve high bandwidth, **shared memory is divided into** equally-sized memory modules, called **banks**

Any memory read or write request made of \( n \) addresses that fall in \( n \) distinct memory banks can be serviced simultaneously
- 1 transaction

Otherwise:
- serialized accesses: separation in bank conflict free groups
- except if threads access the same address in a bank => broadcast

Memory requests are made by warp, so bank conflicts can appears only inside a warp.

For all CUDA architecture \( \geq 2.x \), the number of shared memory banks is 32
Banks

```c
__global__ void myKernel_1D(...){
  __shared__ int myshared[1024];
  myshared[threadIdx.x] = threadIdx.x + 10;
}
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>...</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>...</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>...</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1002</td>
<td>1003</td>
<td>1004</td>
<td>1005</td>
<td>...</td>
<td>1031</td>
<td>1032</td>
<td>1033</td>
<td></td>
</tr>
</tbody>
</table>
Banks

__global__ void myKernel_1D(...)
{
__shared__ int myshared[1024+2];
myshared[threadIdx.x + 2] = threadIdx.x;
}

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>27</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>...</td>
<td>59</td>
<td>60</td>
</tr>
<tr>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>...</td>
<td>91</td>
<td>92</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1022</td>
<td>1023</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Conflict & Conflict-Free Accesses

- All threads within the warp access different banks:
  
  - regular accesses: conflict free

<table>
<thead>
<tr>
<th>banks</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>...</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>threads</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

- permutation:

<table>
<thead>
<tr>
<th>banks</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>...</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>threads</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>
Conflict & Conflict-Free Accesses

- Threads within the warp access several times the same bank:
  - same addresses accessed by bank: conflict free (broadcast)

```c
__global__ void myKernel_1D(...){
  __shared__ int myshared[1024];
  int temp = myshared[threadIdx.x / 2];
}
```

<table>
<thead>
<tr>
<th>banks</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>...</th>
<th>15</th>
<th>...</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>...</td>
<td>15</td>
<td>...</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>threads</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>...</td>
<td>15</td>
<td>...</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>
Conflict & Conflict-Free Accesses

Threads within the warp access several times a same bank:

- different addresses accessed by bank: conflict

```c
__global__ void myKernel_1D(...)
{
    __shared__ int myshared[32*2];
    int temp = myshared[threadIdx.x * 2];  //2-way conflict
}
```

---

**Banks**

**Data**

**Threads**

---
Shared Memory and Architecture

2.x (Fermi):
- Shared memory has **32 banks** that are organized such that successive **32-bit words** map to successive banks
- Each bank has a bandwidth of **32 bits per two clock cycles**

3.x (Kepler):
- Shared memory has **32 banks** with two user-selectable addressing modes: 4 or 8 bytes bank width (32-bits or 64-bits modes)
- Each bank has a bandwidth of **64 bits per clock cycle**

5.x (Maxwell):
- Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks
- Each bank has a bandwidth of **32 bits per clock cycle**
Matrix Transpose Example

31-10-2016
Matrix Transpose: Global Memory Only

__global__ void transpose_global(int size, int *in, int *out) {
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int idx = tx + blockIdx.x * blockDim.x;
    int idy = ty + blockIdx.y * blockDim.y;
    out[idy + idx * size] = in[idx + idy * size];
}

- **Block size 32x32**

- 3009.024 µs for a 4096 * 4096 matrix on a K80

- Writes in global memory are non-coalesced

```c
__global__ void transpose_global(int size, int *in, int *out) {
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int idx = tx + blockIdx.x * blockDim.x;
    int idy = ty + blockIdx.y * blockDim.y;
    out[idy + idx * size] = in[idx + idy * size];
}
```
Matrix Transpose: Via Shared Memory

__global__ void transpose_shared(int size, int *in, int *out){
    __shared__ int shmem[32][32];
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int idx_in = tx + blockIdx.x * blockDim.x;
    int idy_in = ty + blockIdx.y * blockDim.y;

    int idx_out = tx + blockIdx.y * blockDim.y;
    int idy_out = ty + blockIdx.x * blockDim.x;

    shmem[ty][tx] = in[idx_in + idy_in * size];
    __syncthreads();
    out[idx_out + idy_out * size] = shmem[tx][ty];
}

- **2393.984 µs** for a 4096 * 4096 matrix on a K80
- but bank conflicts when reading shared memory
__global__ void transpose_shared(int size, int *in, int *out) {
    __shared__ int shmem[32][32];
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int idx_in = tx + blockIdx.x * blockDim.x;
    int idy_in = ty + blockIdx.y * blockDim.y;
    int idx_out = tx + blockIdx.y * blockDim.y;
    int idy_out = ty + blockIdx.x * blockDim.x;
    shmem[ty][tx] = in[idx_in + idy_in * size];
    __syncthreads();
    out[idx_out + idy_out * size] = shmem[tx][ty];
}
Solving Bank Conflicts

```c
__global__ void transpose_shared(int size, int *in, int *out) {

    __shared__ int shmem[32][33];

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int idx_in = tx + blockIdx.x * blockDim.x;
    int idy_in = ty + blockIdx.y * blockDim.y;

    int idx_out = tx + blockIdx.y * blockDim.y;
    int idy_out = ty + blockIdx.x * blockDim.x;

    shmem[ty][tx] = in[idx_in + idy_in * size];

    __syncthreads();

    out[idx_out + idy_out * size] = shmem[tx][ty];
}
```

**1317.024 µs** for a 4096 * 4096 matrix on a K80
## Performance Summary

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time in µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>1108.928</td>
</tr>
<tr>
<td>Transposition in global</td>
<td>3009.024</td>
</tr>
<tr>
<td>Transposition in shared</td>
<td>2393.984</td>
</tr>
<tr>
<td>Transposition in shared without bank conflicts</td>
<td>1317.024</td>
</tr>
</tbody>
</table>
Minimize Global Memory Accesses
Minimize Global Memory Accesses

- Device memory is the slowest memory on the GPU

- Reducing global memory accesses inside a kernel implies better performances

- Be careful with data reuse:
  - on Kepler global memory accesses are by default cached only in L2 (not L1)

- Keep data in:
  - register: if the current thread will reuse the data
  - shared memory: if another thread from the same block will also use the data
    - as in the transpose example
Software Pipelining Example

Each thread \((i,j)\) performs
\[
\begin{align*}
\text{out}(i,j,k) &= \text{in}(i,j,k) + \text{in}(i,j,k+1) - \text{in}(i,j,k-1) \\
&\quad + \text{in}(i,j,k+2) - \text{in}(i,j,k-2) \\
&\quad + \text{in}(i,j,k+3) - \text{in}(i,j,k-3) \\
&\quad + \text{in}(i,j,k+4) - \text{in}(i,j,k-4)
\end{align*}
\]

__global__ void myKernel_ref(int size_x, int size_y, int size_z, int* in, int* out){

    int idx = threadIdx.x + blockIdx.x * blockDim.x; //i
    int idy = threadIdx.y + blockIdx.y * blockDim.y; //j
    int idz = 0; //k

    int temp;

    for(idz = 4; idz < (size_z-4); idz++){
        temp = in[idx + idy *size_x + (idz+0) * size_x * size_y] \\
            + in[idx + idy *size_x + (idz+1) * size_x * size_y] - in[idx + idy *size_x + (idz-1) * size_x * size_y] \\
            + in[idx + idy *size_x + (idz+2) * size_x * size_y] - in[idx + idy *size_x + (idz-2) * size_x * size_y] \\
            + in[idx + idy *size_x + (idz+3) * size_x * size_y] - in[idx + idy *size_x + (idz-3) * size_x * size_y] \\
            + in[idx + idy *size_x + (idz+4) * size_x * size_y] - in[idx + idy *size_x + (idz-4) * size_x * size_y];

        if( (idx<size_x) && (idy<size_y) ){
            out[idx + idy *size_x + idz * size_x * size_y] = temp;
        }
    }
}

9 reads in the \(in\) array by iteration for a \((i,j)\) thread

\[\Rightarrow (size_z - 8) \times 9 \] read by threads
Software Pipelining Example

- For a same (i,j) thread, successive computations of k accesses common data

- \( in(i,j,k+4) \) for the \( k^{th} \) iteration is the same memory access as:
  - \( in(i,j,k+3) \) for the \( (k+1) \) iteration
  - \( in(i,j,k+2) \) for the \( (k+2) \) iteration
  - ...  
  - \( in(i,j,k-4) \) for the \( (k+8) \) iteration

- Two consecutive k iterations share 7 data

- Keep data in register to avoid to re-read a data from global memory
Software Pipelining Example

```c
__global__ void myKernel_pipeline(int size_x, int size_y, int size_z, const int* __restrict in, int* __restrict__ out) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;    int idy = threadIdx.y + blockIdx.y * blockDim.y;   int idz = 0;

    int in_m4 = 0;
    int in_m3 = in[idx + idy *size_x + 0 * size_x * size_y]; //in[idx + idy *size_x + (idz-4) * size_x * size_y]; with idz = 4
    int in_m2 = in[idx + idy *size_x + 1 * size_x * size_y]; //in[idx + idy *size_x + (idz-3) * size_x * size_y]; with idz = 4
    ...
    int in_p3 = in[idx + idy *size_x + 6 * size_x * size_y]; //in[idx + idy *size_x + (idz+2) * size_x * size_y]; with idz = 4
    int in_p4 = in[idx + idy *size_x + 7 * size_x * size_y]; //in[idx + idy *size_x + (idz+3) * size_x * size_y]; with idz = 4

    for(idz = 4; idz < (size_z-4); idz++) {
        in_m4 = in_m3;
        in_m3 = in_m2;
        in_m2 = in_m1;
        ...
        in_p3 = in_p4;
        in_p4 = in[idx + idy *size_x + (idz+4) * size_x * size_y];

        int temp = in_cu + in_p1 - in_m1 + in_p2 - in_m2 + in_p3 - in_m3 + in_p4 - in_m4;

        if( (idx<size_x) && (idy<size_y) ) {
            out[idx + idy *size_x + idz * size_x * size_y] = temp;
        }
    }
}
```
### Software Pipelining Example

<table>
<thead>
<tr>
<th>size_z</th>
<th>1st version 9*(size_z-8)</th>
<th>2nd version size_z</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>18</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>108</td>
<td>20</td>
</tr>
<tr>
<td>100</td>
<td>828</td>
<td>100</td>
</tr>
</tbody>
</table>

- On a 512*512*512 cube (on Tesla K80):
  - 1st version: **21471.871** µs
  - 2nd version: **8530.880** µs

- This optimization increases pressure on registers and may degrade performance in peculiar cases (more register spilling, less occupancy)
Application bottlenecks

- **Compute bound**: all computation units used
  - INT/SP computations
  - DP computations
  - SFU computations

- **Memory bound**: computation units are waiting for operand data from load/store units
  - Latency bound
  - Bandwidth bound
Definitions

- Latency: time required to perform an operation

- Throughput: how many operations complete per cycle

- Pizza delivery analogy:
  - Latency: Pizza delivery takes 30 minutes to go from pizza shop to your house
  - Throughput: Pizza delivery can’t deliver more than 5 pizzas at a time
Latencies

- Global / local memory: ~200-400 clock cycles of memory latency (3.x)
- Registers: zero extra clock cycles/instruction

  - Issue:
    - Delays (11 cycles) may occur due to register read-after-write dependency. No dependent operation can start for this time.

```plaintext
x = a + b; // takes ~11 cycles to execute
y = a + c; // independent, can start anytime (stall)
z = x + d; //dependent, must wait for completion
```
Latencies

- Global / local memory: ~200-400 clock cycles of memory latency (3.x)

- Registers: zero extra clock cycles/instruction

  - Issue:
    - Delays (11 cycles) may occur due to register read-after-write dependency. No dependent operation can start for this time.

  - Solution:
    - Hide it by overlapping with other operations.

\[
x = a + b; \quad // \text{takes } \sim 11 \text{ cycles to execute}\\
y = a + c; \quad // \text{independent, can start anytime}\\
// \text{add other independent operations}\\
z = x + d; \quad //\text{dependent, must wait for completion}
\]
Adding Other Operations

Adding other operations can be done with:

- **Thread level parallelism (TLP)**
  
<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
<th>thread 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = x + c</td>
<td>y = y + c</td>
<td>z = z + c</td>
<td>w = w + c</td>
</tr>
<tr>
<td>x = x + b</td>
<td>y = y + b</td>
<td>z = z + b</td>
<td>w = w + b</td>
</tr>
<tr>
<td>x = x + a</td>
<td>y = y + a</td>
<td>z = z + a</td>
<td>w = w + a</td>
</tr>
</tbody>
</table>

- **Instruction level parallelism (ILP)**
  
<table>
<thead>
<tr>
<th>thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>w = w + b</td>
</tr>
<tr>
<td>z = z + b</td>
</tr>
<tr>
<td>y = y + b</td>
</tr>
<tr>
<td>x = x + b</td>
</tr>
</tbody>
</table>

Hiding Latency = Maintaining Throughput
**Thread Level Parallelism**

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
<th>thread 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = x + c</td>
<td>y = y + c</td>
<td>z = z + c</td>
<td>w = w + c</td>
</tr>
<tr>
<td>x = x + b</td>
<td>y = y + b</td>
<td>z = z + b</td>
<td>w = w + b</td>
</tr>
<tr>
<td>x = x + a</td>
<td>y = y + a</td>
<td>z = z + a</td>
<td>w = w + a</td>
</tr>
</tbody>
</table>

4 independent operations
What is occupancy?

- **Scope:**
  - Software: kernel (not the whole program)
  - Hardware: streaming multiprocessor (independent of the amount of SM available)

- **Measure:**

  \[
  \text{occupancy} = \frac{\text{#active threads}}{\text{#maximum supported threads}}
  \]

Why should I maximize it?

- If enough transactions in flight, latency is covered through warp context switching.
Occupancy

- Potential occupancy limiters:
  - Hardware resources are allocated for an entire block
  - Utilizing too many resources per thread may limit the occupancy

- Because of this, programmers need to choose with care:
  - The size of **thread blocks**
  - The right amount of mixed resources per thread: **registers** & **shared memory**

- Occupancy of a same kernel may vary on different architecture:
  - due to differences between streaming multiprocessors:
    - maximum active threads supported: 1536 on 2.x, 2048 on 3.x
    - 32 bit-registers: 64K on 3.5, 128K on 3.7
    - shared memory size : 48KB on 3.0, 112KB on 3.7
Potential Occupancy Limiters: Block Size

Example on 3.5 architecture:

- Each SM can have up to:
  - maximum 16 active blocks

\[
\text{occupancy} = \frac{\text{#active threads}}{\text{maximum supported threads}}
\]

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Active Threads (if 16 active blocks)</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>32 \times 16 = 512</td>
<td>512/2048 = 0.25</td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
<td>0.5</td>
</tr>
<tr>
<td>128</td>
<td>2048</td>
<td>1</td>
</tr>
<tr>
<td>192</td>
<td>3072</td>
<td>&gt;1</td>
</tr>
<tr>
<td>256</td>
<td>4096</td>
<td>&gt;1</td>
</tr>
</tbody>
</table>
Potential Occupancy Limiters: Block Size

Example on 3.5 architecture:

- Each SM can have up to:
  - maximum 2048 active threads

\[
\text{occupancy} = \frac{\#\text{active threads}}{\#\text{maximum supported threads}}
\]

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Active Threads (without active block limit)</th>
<th>Number of Block</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>2048</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>2048</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>128</td>
<td>2048</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>192</td>
<td>1920 (192 * 11 (blocks) &gt; 2048)</td>
<td>10</td>
<td>0.9375</td>
</tr>
<tr>
<td>256</td>
<td>2048</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
## Potential Occupancy Limiters: Block Size

### Example on 3.5 architecture:

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Active Threads (if 16 active blocks)</th>
<th>Active Threads (without active block limit)</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>512</td>
<td>2048</td>
<td>0.25</td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
<td>2048</td>
<td>0.5</td>
</tr>
<tr>
<td>128</td>
<td>2048</td>
<td>2048</td>
<td>1</td>
</tr>
<tr>
<td>192</td>
<td>3072</td>
<td><strong>1920</strong></td>
<td>0.9375</td>
</tr>
<tr>
<td>256</td>
<td>4096</td>
<td>2048</td>
<td>1</td>
</tr>
</tbody>
</table>
### Potential Occupancy Limiters: Block Size

**Example on 3.5 architecture:**
- maximum 1024 threads in a block

<table>
<thead>
<tr>
<th>block size</th>
<th>occupancy</th>
<th>block size</th>
<th>occupancy</th>
<th>block size</th>
<th>occupancy</th>
<th>block size</th>
<th>occupancy</th>
<th>block size</th>
<th>occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.25</td>
<td>288</td>
<td>0.98</td>
<td>544</td>
<td>0.80</td>
<td>800</td>
<td>0.78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>0.5</td>
<td>320</td>
<td>0.94</td>
<td>576</td>
<td>0.84</td>
<td>832</td>
<td>0.81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>0.75</td>
<td>352</td>
<td>0.86</td>
<td>608</td>
<td>0.89</td>
<td>864</td>
<td>0.84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>384</td>
<td>0.94</td>
<td>640</td>
<td>0.94</td>
<td>896</td>
<td>0.88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>0.94</td>
<td>416</td>
<td>0.81</td>
<td>672</td>
<td>0.98</td>
<td>928</td>
<td>0.91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>0.94</td>
<td>448</td>
<td>0.88</td>
<td>704</td>
<td>0.69</td>
<td>960</td>
<td>0.94</td>
<td></td>
<td></td>
</tr>
<tr>
<td>224</td>
<td>0.98</td>
<td>480</td>
<td>0.94</td>
<td>736</td>
<td>0.72</td>
<td>992</td>
<td>0.97</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>512</td>
<td>1</td>
<td>768</td>
<td>0.75</td>
<td>1024</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Potential Occupancy Limiters: Registers & Shared Memory

- Registers & shared memory usage given with:
  - nvcc --ptxas-options=-v / -Xptxas=-v

```
$> nvcc -Xptxas -v cuda_code.cu
ptxas info : Compiling entry function 'cuda_kernel'
ptxas info : Used 4 registers, 60+56 bytes lmem, 44+40 bytes smem,
             20 bytes cmem[1], 12 bytes cmem[14]
```

- Registers and shared memory are limited by streaming multiprocessor:
  - using too much of these resources reduces the occupancy
Potential occupancy limiters: Registers

Example on 3.5 architecture:

- Each SM can have up to:
  - 64K registers / SM

- Example 1
  - Kernel uses 32 registers per thread
  - 64k/32 = 2048 threads ( >= 2048 threads thus occupancy = 1)

- Example 2
  - Kernel uses 64 registers per thread
  - 64k/64 = 1024 threads (occupancy = 1024/2048 = 0.5)

- Control register usage using nvcc flag: --maxregcount=N
  - registers are spilled in local memory
Potential occupancy limiters: Registers

Example on 3.5 architecture:

- Each SM can have up to:
  - 48k, 32k or 16k shared memory/L1 per SM

- Example 1, 48k shared memory
  - Kernel uses 24 bytes of shared memory per thread
  - $48k / 24 = 2048$ threads (occupancy = 1)

- Example 2, 16k shared memory
  - Kernel uses 24 bytes of shared memory per thread
  - $16k / 24 = 682$ threads (occupancy = 0.3333)

- Don’t use too much shared memory

- Choose L1/shared configuration appropriately
Occupancy

- But ... a better occupancy does not imply better performance:
  - fewer active threads computing in lower latency memory are usually faster

- Computing in register and/or shared memory instead of global memory

- Occupancy is good for final optimizations
  - try to save registers or shared memory to increase active threads on optimized kernels
    - computation reorganization
    - --maxregcount
    - different block sizes can use different amount of shared memory
      - for example 2D kernel using a stencil of 4 points
        - 16*8 block => \((16+2*4) \times (8+2*4) = 384\) values needed in shared memory => \(384/128 = 3\) values by threads
        - 32*4 block => \((32+2*4) \times (4+2*4) = 480\) values needed in shared memory => \(480/128 = 3.75\) values by threads
Occupancy Calculator

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below (or click here for help)

1. Select Compute Capability (aclk): 3.5

2. Enter your resource usage:
   - Threads Per Block
   - Registers Per Thread
   - Shared Memory Per Block (Bytes)

3. GPU Occupancy Data is displayed here and in the graphs:
   - Active Threads per Multiprocessor
   - Active Warps per Multiprocessor
   - Active Thread Blocks per Multiprocessor
   - Occupancy of each Multiprocessor

Physical Limits for GPU Compute Capability: 3.5
- Threads per Warp: 32
- Warps per Multiprocessor: 96
- Threads per Multiprocessor: 2048
- Thread Blocks per Multiprocessor: 96
- Total # of 32-bit registers per Multiprocessor: 8932
- Register allocation unit size: 256
- Register allocation granularity: 128
- Registers per Thread: 256
- Shared Memory per Multiprocessor (Bytes): 49152
- Shared Memory Allocation unit size: 256
- Warp allocation granularity: 4
- Maximum Thread Block Size: 1024

Allocated Resources

<table>
<thead>
<tr>
<th>Per Block</th>
<th>Unit Per SM</th>
<th>Allocatable Block Per</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp</td>
<td>Warp Limit SM due to warp warp</td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td>Registers per SM due to warp and warp block</td>
<td></td>
</tr>
<tr>
<td>Shared Memory (Bytes)</td>
<td>4096</td>
<td>4096</td>
</tr>
</tbody>
</table>

Maximum Thread Blocks Per Multiprocessor

- Limited by Max Warps per Multiprocessor: 96
- Limited by Registers per Multiprocessor: 96

Physical Max Warps/SM = SM (occupancy = 64 / 64 = WPI)

---

Click here for detailed instructions on how to use this occupancy calculator. For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphic. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

---

65  | 31-10-2016  | G-E Moulard  | © Atos - Confidential
BDS | HPC | CEPP | May not be reproduced, distributed or used in any manner without permission
Instruction Level Parallelism

31-10-2016
Instruction Level Parallelism

► Help to hide latencies and/or to saturate the computation units

► More important on architecture with warp scheduler able to launch 2 instructions / cycle

► Example on Kepler:
  – 192 units for 32-bit floating-point add, multiply, multiply-add
  – 4 warp schedulers / multiprocessor
    • 2 independent instructions / cycle

  – without ilp:
    • $4 \times 32 \times 1$ op/cycle = 128 < 192

  – with ilp:
    • $4 \times 32 \times 2$ ops/cycle = 256 > 192
Roofline Model

31-10-2016
Arithmetic Intensity

\[ AI = \frac{\text{number of arithmetic instructions}}{\text{size of global memory accesses}} \]

```c
__global__ void k1(float *a, float *b) {
    float x = a[ threadIdx.x ]; // read of 4 bytes
    float y = b[ threadIdx.x ]; // read of 4 bytes
    // 2 instructions and 1 write of 4 bytes
    a[ threadIdx.x ] = x * x + y;
}

__global__ void k2(float *a) {
    float x = threadIdx.x;
    // 12 instructions and 1 write of 4 bytes
    a[ threadIdx.x ] = x*x*x*x + 3.14 f*x*x*x - 5.3 f*x*x + 0.5 f*x;
}
```

- \( A1(k1) = 2/12 \ (3*4) = 0,1666 \)
- \( A1(k1) = 12/4 = 3 \)
Roofline Model


![Roofline Diagram](image)
Intra-Warp Operations
Intra-Warp Operations

- Since CC>3.x, there are operations of communications between threads of a same warp without using the shared memory.

- The communication is simultaneous for all the threads of an active warp.

- Operations:
  - shuffle: point to point data exchange
  - vote: reduction and broadcast
Shuffle Operations

- Work with \texttt{int}, \texttt{float} and half types.
  - Larger types require to split the instruction in several calls

\_shfl(var, laneId) : direct copy from the thread of \texttt{laneId} id

\_shfl\_up(var, shift) : relative copy from the thread of id \texttt{current threads} - \texttt{shift}
  - If \texttt{current threads} - \texttt{shift} < 0, the value is unchanged

\_shfl\_down(var, shift) : same but shift of +\texttt{shift}

\_shfl\_xor(var, mask) : copy from the thread of id \texttt{current threads XOR mask}

- An optional parameter \texttt{width} can be passed for all \_shfl functions and must by a power of 2 lower than the warp size (i.e. 2, 4, 8, 16 or 32).
  - Allow to do the \_shfl by sub-group of warp with logical id modulo \texttt{width}
Broadcast Example

```cpp
__global__ void bcast ( int arg ) {
    int laneId = threadIdx.x & 0x1f; // threadIdx.x % 32;
    int value;
    if (laneId == 0)
        value = arg; // Only for the thread of the 0 lane
    value = __shfl (value, 0); // retrieve the value of the 0 lane
```
Vote Instructions

▶ Work only with variables of int type and are only valid for active threads

▶ The result is the same for all the threads within a warp

```c
__all(val) : result ≠ 0 only if val ≠ 0 for all threads

__any(val) : result ≠ 0 only if val ≠ 0 for at least one thread

__ballot(val) : return an unsigned int in which the n-th bit is 1 only if val ≠ 0 for the n-th active thread
```
Thanks

For more information please contact:
Georges-Emmanuel Moulard
M+ 33 6 85529054
georges-emmanuel.moulard@atos.net

Atos, the Atos logo, Atos Consulting, Atos Worldgrid, Worldline, BlueKiwi, Canopy the Open Cloud Company, Yunano, Zero Email, Zero Email Certified and The Zero Email Company are registered trademarks of Atos. April 2015. © 2015 Atos. Confidential information owned by Atos, to be used by the recipient only. This document, or any part of it, may not be reproduced, copied, circulated and/or distributed nor quoted without prior written approval from Atos.

31-10-2016