GPU Hardware Performance

Jiří Filipovič

Fall 2015

Atomic operations

- performs read-modify-write operations on shared or global memory
- no interference with other threads
- for 32-bit and 64-bit integers (c. c. ≥ 1.2), float addition (c. c. ≥ 2.0)
- using global memory for c. c. ≥ 1.1 and shared memory for c. c. ≥ 1.2
- arithmetic (Add, Sub, Exch, Min, Max, Inc, Dec, CAS) a bitwise (And, Or, Xor) operations

Warp Voting

All threads in one warp evaluate the same condition and perform its comparison. Available in c. c. ≥ 1.2.

```
int __all(int predicate);
```

Result is non-zero iff the predicate is non-zero for all the threads in the warp.

```
int __any(int predicate);
```

Result is non-zero iff the predicate is non-zero for at least one thread in the warp.

```
unsigned int __ballot(int predicate);
```

Contains voting bit mask of individual threads.

Shuffle Functions

Threads within a warp can efficiently communicate using warp shuffle functions (from c.c. ≥ 3.0).

```
float __shfl(float var, int srcLane, int width=warpSize);
```

Copy value from `srcLane`.

```
float __shfl_up(float var, unsigned int delta, int width=warpSize)
```

Copy value from threads with lower ID relative to caller. Analogically `__shfl_down`.

```
float __shfl_xor(float var, int laneMask, int width=warpSize);
```

Copy from a thread based on bitwise XOR of own ID and `laneMask`.

Parameter `width` defines the number of participating threads. It must be power of two, indexing starts at 0.

Synchronization of Memory Operations

Compiler can optimize operations on shared/global memory (intermediate results may be kept in registers) and can reorder them

- if we need to ensure that the data are visible for others, we use `__threadfence()` or `__threadfence_block()`
- if a variable is declared as `volatile`, all load/store operations are implemented in shared/global memory
  - very important if we assume implicit warp synchronization

Global Synchronization using Atomic Operations

Alternative implementation of a vector reduction

- each block sums elements in its part of a vector
- barrier (weak global barrier)
- one block sums results of all the blocks

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
  float* result) {
  float partialSum = calculatePartialSum(array, N);
  if (threadIdx.x == 0) {
    result[blockIdx.x] = partialSum;
    __threadfence();
    unsigned int value = atomicInc(&count, gridDim.x);
    isLastBlockDone = (value == (gridDim.x - 1));
  }
  __syncthreads();
  if (isLastBlockDone) {
    float totalSum = calculateTotalSum(result);
    if (threadIdx.x == 0) {
      result[0] = totalSum;
      count = 0;
    }
  }
}
```

Global Memory Access Optimization

Performance of global memory becomes a bottleneck easily

- global memory bandwdith is low relatively to arithmetic performance of GPU (G200 $\geq$ 24 FLOPS/float, G100 $\geq$ 30, GK110 $\geq$ 62, GM200 $\geq$ 73)
- 400–600 cycles latency

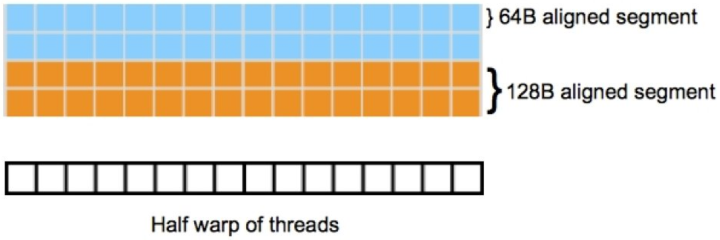
The throughput can be significantly worse with bad parallel access pattern

- the memory has to be accessed *coalesced*
- use of just certain subset of memory regions should be avoided (*partition camping*)

Coalesced Memory Access (C. C. < 2.0)

GPU memory needs to be accessed in larger blocks for efficiency

- global memory is split into 64 B segments
- two of these segments are aggregated into 128 B segments



Coalesced Memory Access (C. C. < 2.0)

A half of a warp can transfer data using single transaction or one to two transactions when transactions when transferring a 128 B word

- it is necessary to use large words
- one memory transaction can transfer 32 B, 64 B, or 128 B words
- GPUs with c. c. ≤ 1.2
  - the accessed block has to begin at an address dividable by $16\times$ data size
  - *k*-th thread has to access *k*-th block element
  - some threads needn't participate
- if these rules are not obeyed, each element is retrieved using a separate memory transaction

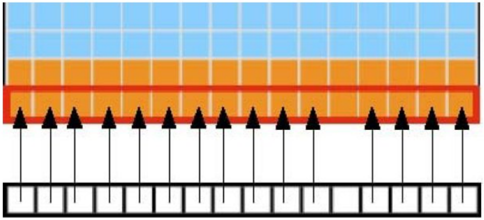
Coalesced Memory Access (C. C. < 2.0)

GPUs with c. c. ≥ 1.2 are less restrictive

- each transfer is split into 32 B, 64 B, or 128 B transactions in a way to serve all requests with the least number of transactions
- order of threads can be arbitrarily permuted w.r.t. transferred elements

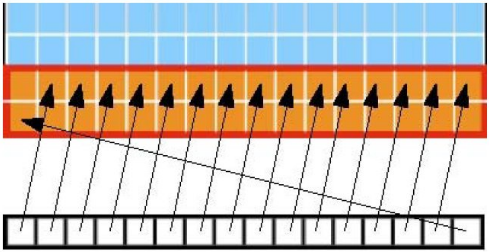
Coalesced Memory Access (C. C. $<$ 2.0)

Threads are aligned, element block is contiguous, order is not permuted – coalesced access on all GPUs



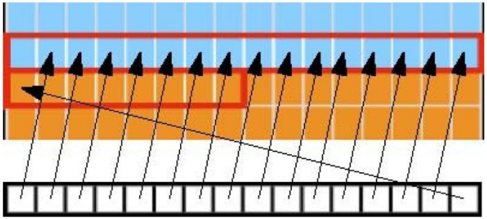
Unaligned Memory Access (C. C. < 2.0)

Threads **are not** aligned, contiguous elements accessed, order is not permuted – one transaction on GPUs with c. c. ≥ 1.2



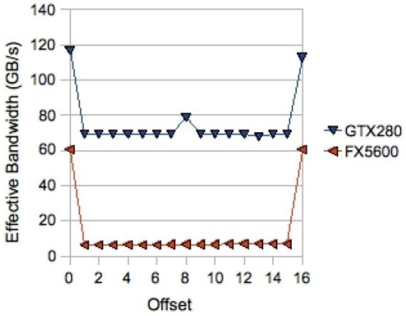
Unaligned Memory Access (C. C. $<$ 2.0)

Similar case may result in a need for two transactions



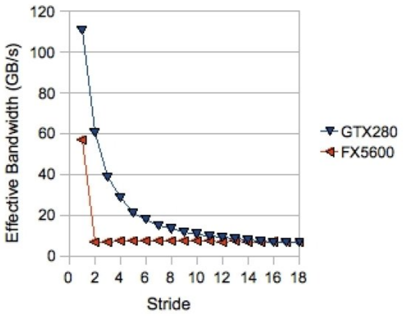
Unaligned Memory Access Performance (C. C. < 2.0)

Older GPUs perform smallest possible transfer (32 B) for each element, thus reducing performance to 1/8
Newer GPUs perform (c. c. ≥ 1.2) two transfers



Interleaved Memory Access Performance (C. C. < 2.0)

The bigger the spaces between elements, the bigger performance drop on GPUs with c. c. ≥ 1.2 – the effect is rather dramatic



Global Memory Access with Fermi (C. C. $= 2.x$)

Fermi has L1 and L2 cache

- L1: 256 B per row, 16 kB or 48 kB per multiprocesor in total
- L2: 32 B per row, 768 kB on GPU in total

What are the advantages?

- more efficient programs with unpredictable data locality
- more efficient when shared memory is not used from some reason
- unaligned access – no slowdown in principle
- interleaved access – data needs to be used before it is flushed from the cache, otherwise the same or bigger problem as with c. c. < 2.0 (L1 cache may be turned of to avoid overfetching)

Global Memory Access with "gaming" Kepler (C. C. = 3.0)

There is only L2 cache for read/write global memory access

- L2: 23 B per row, up to 1.5 GB per GPU
- L1: for local memory, 16 KB, 32 KB or 48 KB in total

Global Memory Access with full-featured Kepler and Maxwell (C. C. $\geq$ 3.5)

Read-only data cache

- shared with textures
- compiler tries to use, we can help with `__restrict__` and `__ldg()`
- slower than Fermi's L1

Maxwell does not have L1 cache for local memory

- inefficient for programs heavily using local memory

Partition camping

- relevant for c. c. 1.x (and AMD GPUs)
- processors based on G80 have 6 regions, G200 have 8 regions of global memory
- the memory is split into regions in 256 B chunks
- even access among the regions is needed for maximum performance
  - among individual blocks
  - block are usually run in order given by their position in the grid
- if only part of regions is used, the resulting condition is called *partition camping*
- generally not as critical as the coalesced access
- more tricky, problem size dependent, disguised from fine-grained perspective

HW Organization of Shared Memory

Shared memory is organized into memory banks, which can be accessed in parallel

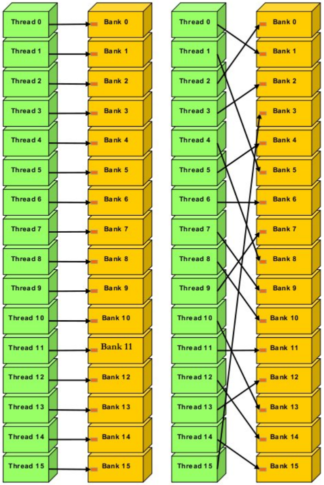
- c. c. 1.x 16 banks, c. c. $\geq$ 2.0 32 banks
- memory space mapped in an interleaved way with 32 b shift or 64 b shift (c.c. 3.x)
- to use full memory performance, we have to access data in different banks
- broadcast implemented – if all threads access the same data

Bank Conflict

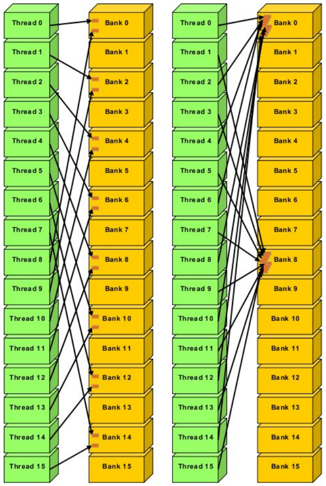
Bank conflict

- occurs when some threads in warp/half-warp access data in the same memory bank with several exceptions
    - threads access exactly the same data
    - threads access different half-words of 64 b word (c.c. 3.x)
- when occurs, memory access gets serialized
- performance drop is proportional to number of parallel operations that the memory has to perform to serve a request

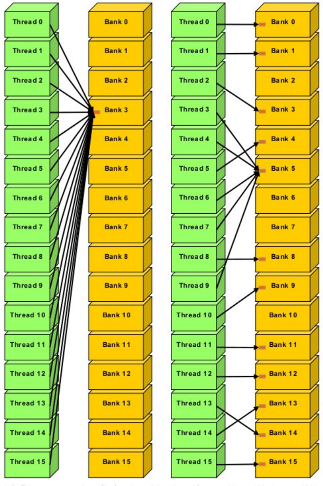
Access without Conflicts



n-Way Conflicts



Broadcast


Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
Bank 0
Bank 1
Bank 2
Bank 3
Bank 4
Bank 5
Bank 6
Bank 7
Bank 8
Bank 9
Bank 10
Bank 11
Bank 12
Bank 13
Bank 14
Bank 15

Access Patterns

Alignment is not needed, bank conflicts not generated

```
int x = s[threadIdx.x + offset];
```

Interleaving does not create conflicts if c is odd, for $c.c. \geq 3.0$ no conflict if $c = 2$ and 32 b numbers are accessed

```
int x = s[threadIdx.x * c];
```

Access to the same variable never generates conflicts on c. c. 2.x, while on 1.x only if thread count accessing the variable is multiple of 16

```
int x = s[threadIdx.x % c];
```

Other Memory Types

Transfers between host and GPU memory

- need to be minimized (often at cost of decreasing efficiency of computation on GPU)
- may be accelerated using page-locked memory
- it is more efficient to transfer large blocks at once
- computations and memory transfers should be overlapped

Texture memory

- designed to reduce number of transfers from the global memory
- works well for aligned access
- does not help if latency is the bottleneck
- may simplify addressing or add filtering

Other Memory Types

Constant memory

- as fast as registers if the same value is read
- performance decreases linearly with number of different values read

Registers

- read-after-write latency, hidden if at least 192 threads are running for c. c. 1.x or at least 768 threads are running for c. c. 2.x
- possible bank conflicts even in registers
  - compiler tries to avoid them
  - we can make life easier for the compiler if we set block size to multiple of 64