RWTH AACHEN UNIVERSITY

GPGPU
Tuning Concepts

CUDA as an example

Sandra Wienke, M.Sc.
wienke@rz.rwth-aachen.de

PPCES 2012

Links

- **GPU Technology Conference (GTC) On-Demand:**

  http://www.gputechconf.com/gtcnew/on-demand-GTC.php

  - Paulius Micikevicius: Fundamental Performance Optimizations for GPUs
  - Vasily Volkov: Better Performance on Lower Occupancy
  - Peng Wang: Analysis Driven Optimization with CUDA

Agenda

- **Kernel optimizations**
  - Control flow
  - Launch configuration
  - Global memory throughput
  - Shared memory
  - Misc
- **GPU-CPU interaction**
  - Minimal data transfer
  - Pinned memory
  - Streams & async memcopies
- **Summary**

Kernel optimizations: Control flow

- **Threads execute as groups of 32 threads (warps)**
  - Threads in warp share same program counter

→ **SIMT architecture**

- **Diverging threads within a warp**
  - Threads within a single warp take different paths
  - Different execution paths within warp are serialized

```
if (threadIdx.x > 2)
{...} else {...}
```

→ **Same code path for threads within warp**

  - Threads execute synchronously

→ **Granularity of branches: multiple of warp size**

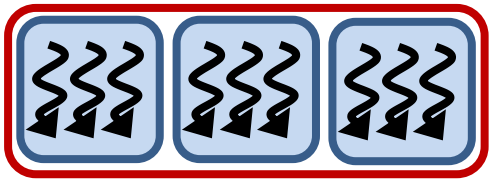
```
if (threadIdx.x / WARP_SIZE > 2) {...} else {...}
```

Different code for different warps: no impact on performance

Agenda

- **Kernel optimizations**
  - Control flow
  - Launch configuration
  - Global memory throughput
  - Shared memory
  - Misc
- **GPU-CPU interaction**
  - Minimal data transfer
  - Pinned memory
  - Streams & async memcopies
- **Conclusion**

Kernel optimizations: Launch configuration



- **Kernel executes grid of blocks of threads**

```
myKernel<<<blocksPerGrid, threadsPerblock>>>(…)
```

→ Launch configuration

- **How many threads/blocks to launch?**
- **Hardware operation**
  - Instructions are issued in order
  - Thread execution stalls when one of the operands isn't ready
  - Latency is hidden by switching threads
    - GMEM latency: 400-800 cycles
    - Arithmetic latency: 18-22 cycles

→ Need enough threads to hide latency

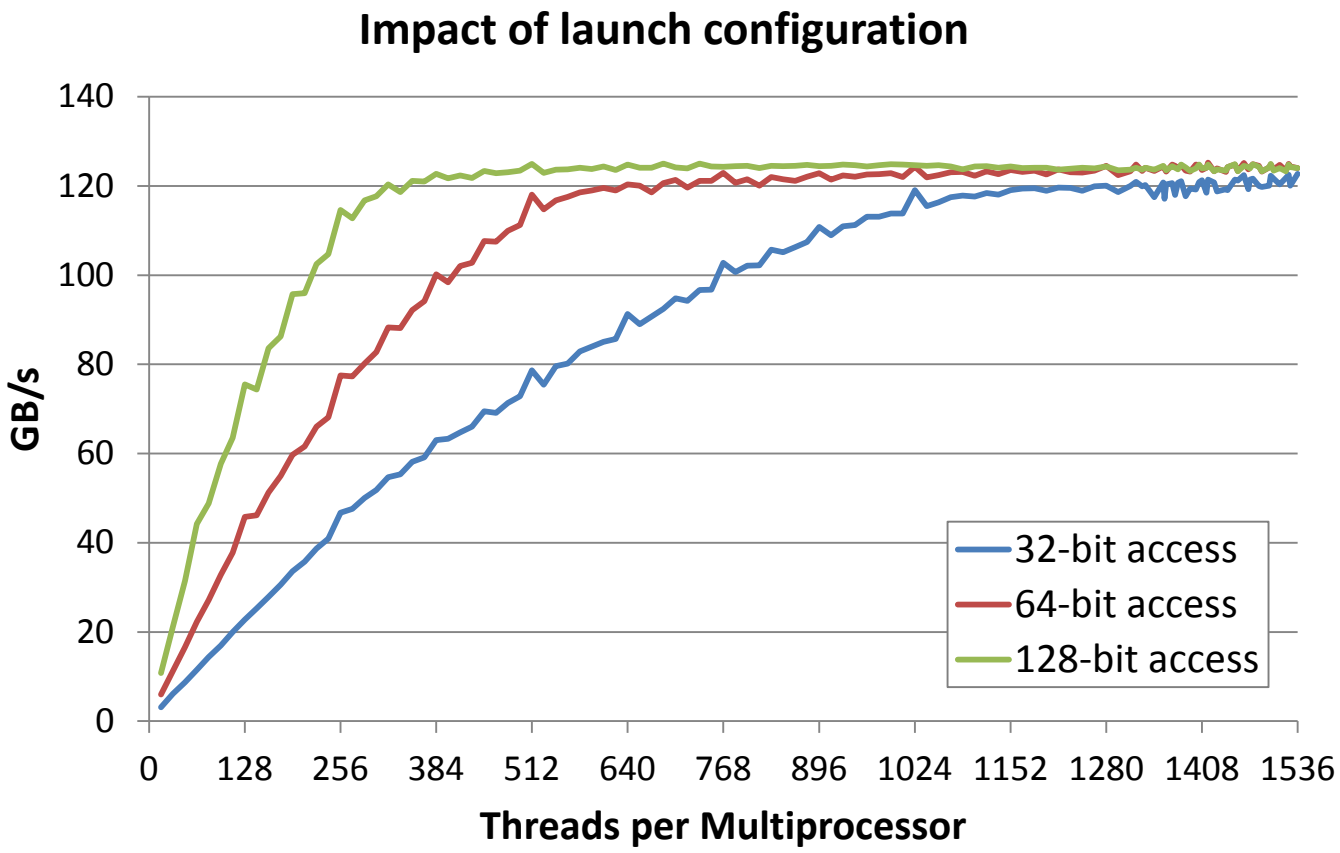
Kernel optimizations: Launch configuration

- **Hiding arithmetic latency**
  - Need ~18 warps (576 threads) per Fermi MP (192 threads per GT200 MP)
  - Or, independent instructions from the same warp

- **Maximizing global memory throughput**
  - Gmem throughput depends on the access pattern, and word size
  - Need enough memory transactions in flight to saturate the bus
    - Independent loads and stores from the same thread
    - Loads and stores from different threads
    - Larger word sizes can also help

Maximizing global memory throughput

- Example program: increment an array of ~67M elements

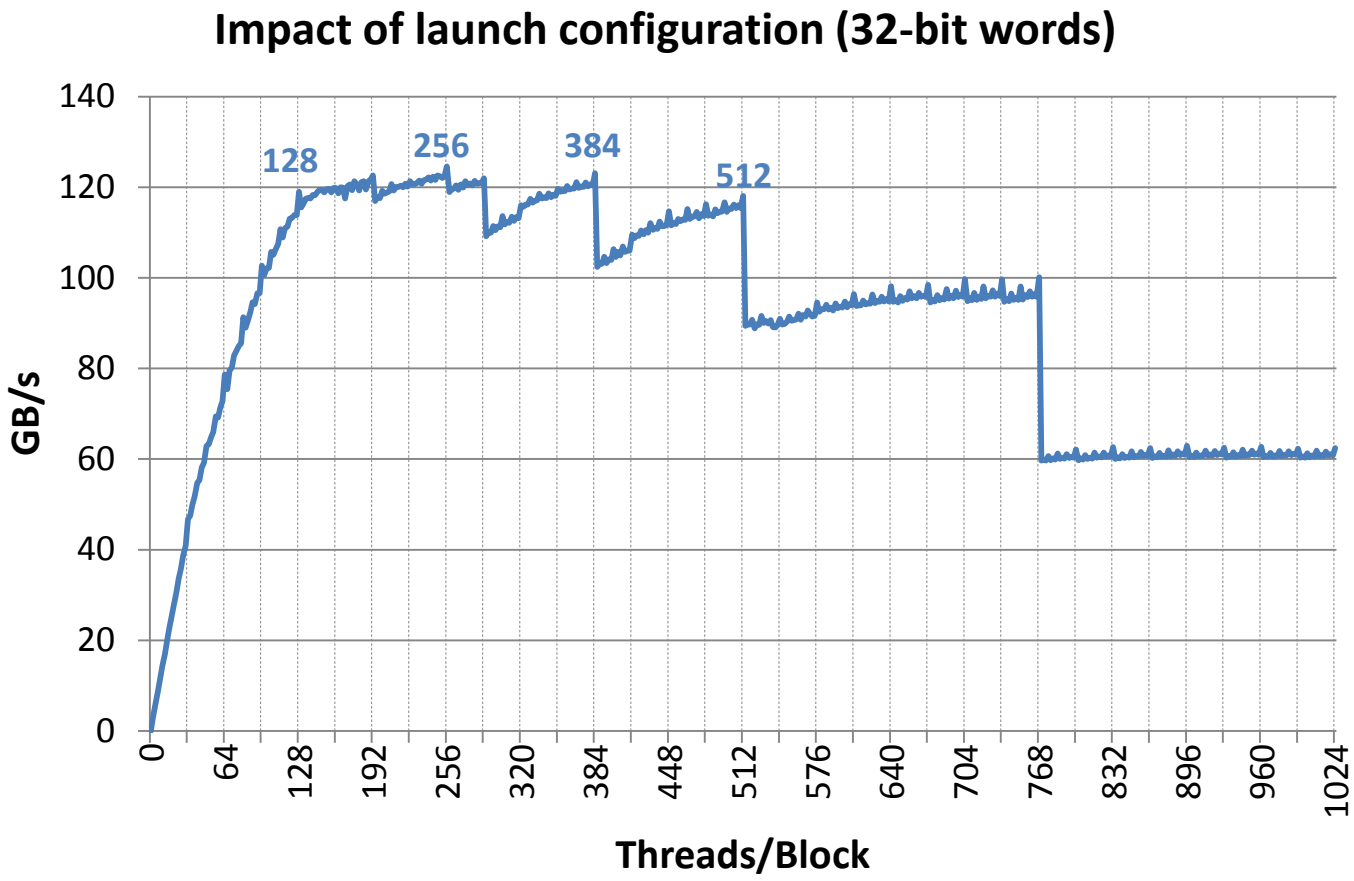


- NVIDIA Tesla C2050 (Fermi)
- ECC off
- Bandwidth: 144 GB/s

Kernel optimizations: Launch configuration

Maximizing global memory throughput

- Example program: increment an array of ~67M elements



- NVIDIA Tesla C2050 (Fermi)
- ECC off
- Bandwidth: 144 GB/s

Kernel optimizations: Launch configuration

- **Threads per block: Multiple of warp size (32)**
  - Threads are always spanned in warps onto resources
  - Not used threads are marked as inactive
  - Starting point: 128-256 threads/block
  - Too little threads/block: prevent achieving good occupancy
  - Too many threads/block: less flexible (resources)
- **Blocks per grid heuristics**
  - #blocks > #MPs
    - MPs have at least one block to execute
  - #blocks / #MPs > 2
    - MP can concurrently execute up to 8 blocks
    - Blocks that aren't waiting in barrier keep hardware busy
    - Subject to resource availability (registers, smem)
  - #blocks > 100 to scale to future devices

→ Launch MANY threads to keep GPU busy

Agenda

- **Kernel optimizations**
  - Control flow
  - Launch configuration
  - Global memory throughput
  - Shared memory
  - Misc
- **GPU-CPU interaction**
  - Minimal data transfer
  - Pinned memory
  - Streams & async memcopies
- **Summary**

Background information

- **NVIDIA‘s compute capability (cc)**
  - Describes architecture and features of GPU
  - E.g. number of cores or registers
  - E.g. double precision computations (only since cc 1.3)
  - E.g. management of memory accesses (per 16/32 threads)

- **Examples**
  - GT200, Tesla C1060: cc 1.3
  - Fermi C2050, Quadro 6000: cc 2.0

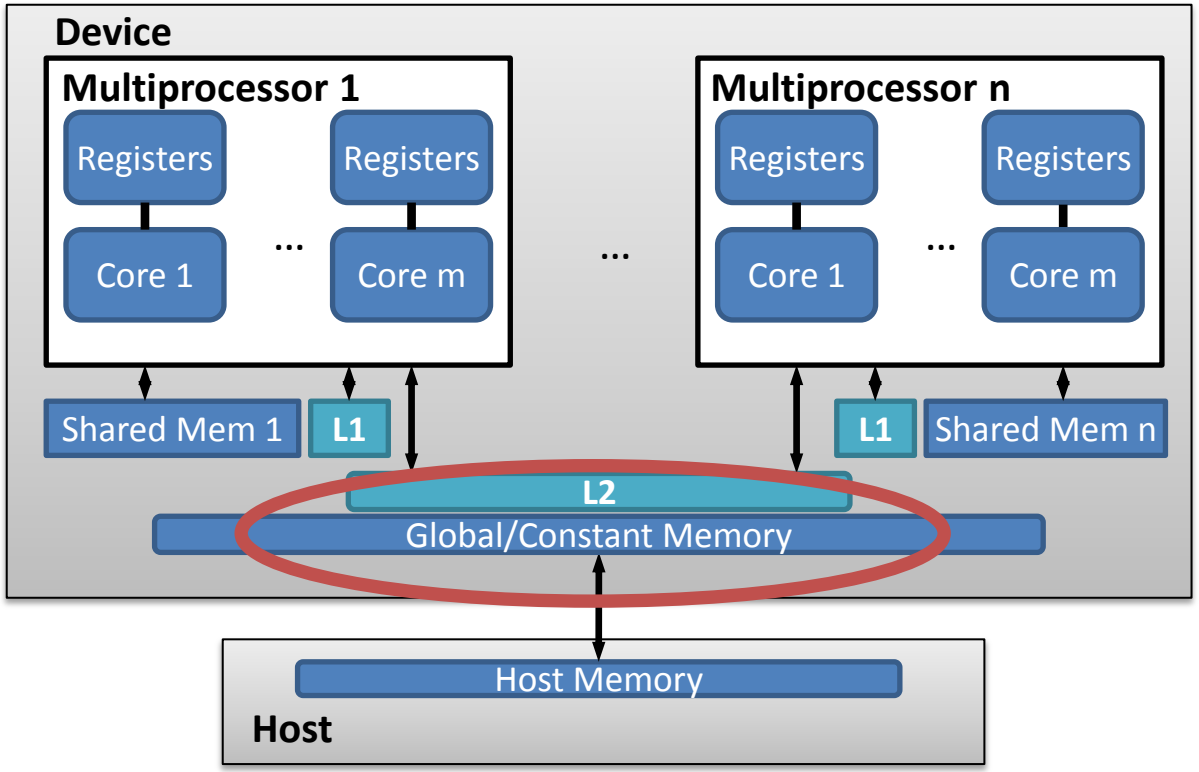
Review: Memory hierarchy

- **Local memory/ Registers**
- **Shared memory/ L1**
  - Very low latency

    (~100x than gmem)
  - Bandwidth (aggregate):

    1+ TB/s
- **L2**
- **Global memory**
  - High latency

    (400-800 cycles)
  - Bandwidth: 144 GB/s



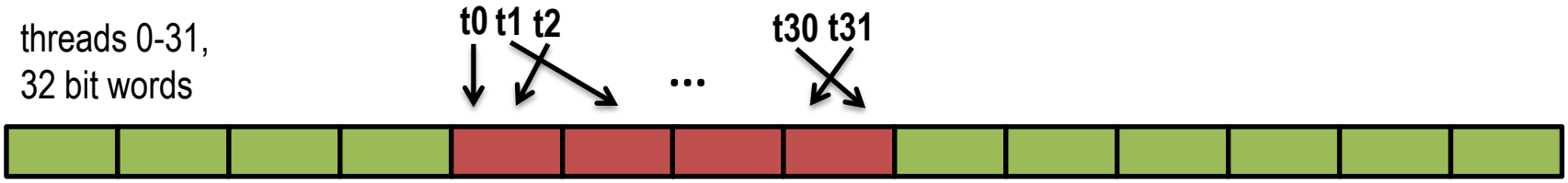
Kernel optimizations: gmem throughput

- **Loads**
  - Caching
    - Default mode
    - Attempts to hit in L1, then L2, then gmem
    - Load granularity is 128-byte line
  - Non-caching
    - Compiler-option (CUDA: -Xptxas-dlcm=cg)
    - Attempts to hit in L2, then gmem
      - Do not hit in L1, invalidate the line if it‘s in L1 already
    - Load granularity is 32-bytes
- **Stores**
  - Invalidate L1, write-back for L2

Kernel optimizations: gmem throughput

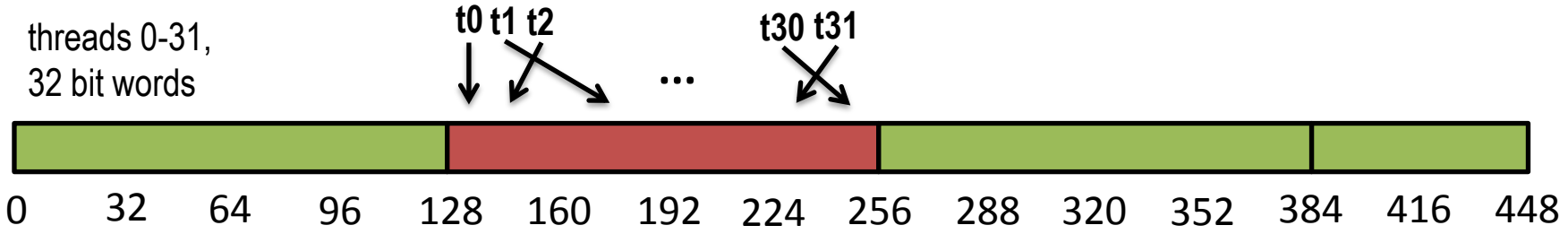
- **Gmem access per warp (32 threads)**
  - cc 1.x: access per half-warp (16 threads)

- **Load operation from gmem**
  - Threads in a warp provide memory addresses
  - Determine which lines/segments are needed
  - Request the needed lines/segments

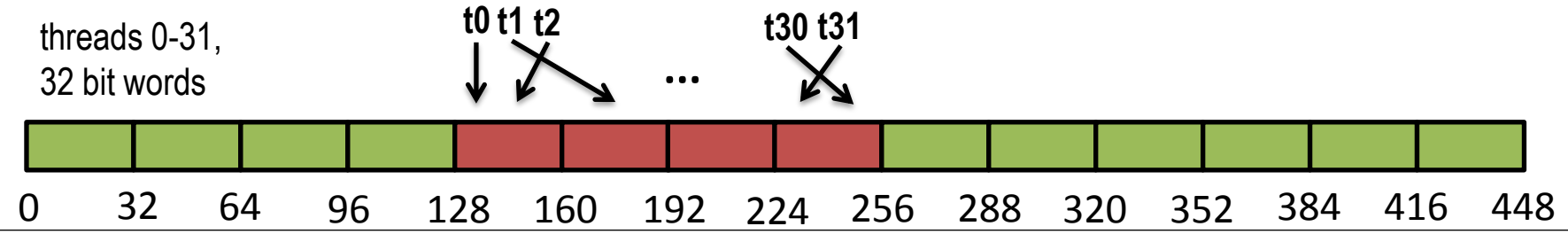


Kernel optimizations: gmem throughput

- **Example 1**
  - Warp requests 32 aligned, permuted 4-byte words → 128 bytes needed
- **Caching load**
  - Addresses fall within 1 cache-line:
    128 bytes move across the bus on a miss → **100 %** bus utilization

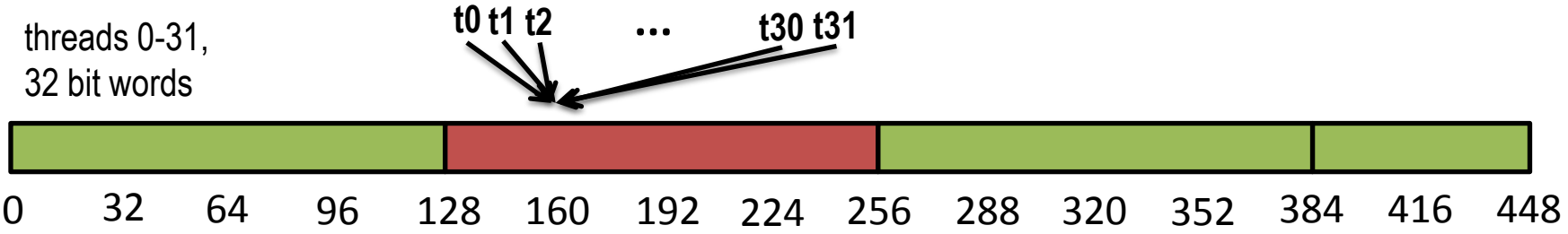


- **Non-caching load**
  - Addresses fall within 4 segments
    128 bytes move across the bus on a miss → **100 %** bus utilization

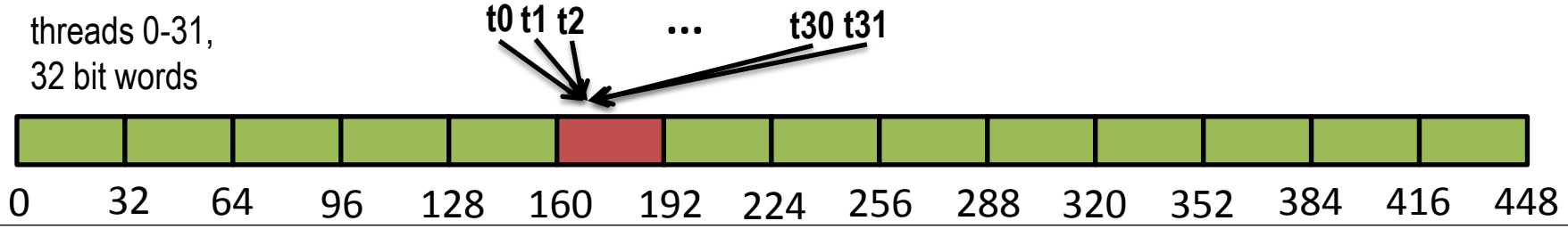


Kernel optimizations: gmem throughput

- **Example 2 (*broadcast*)**
  - All threads in a warp request the same 4-byte word → 4 bytes needed
- **Caching load**
  - Addresses fall within 1 cache-line:
    128 bytes move across the bus on a miss → **3.125 %** bus utilization



- **Non-caching load**
  - Addresses fall within 1 segment
    32 bytes move across the bus on a miss → **12.5 %** bus utilization



Kernel optimizations: gmem throughput

Conclusion

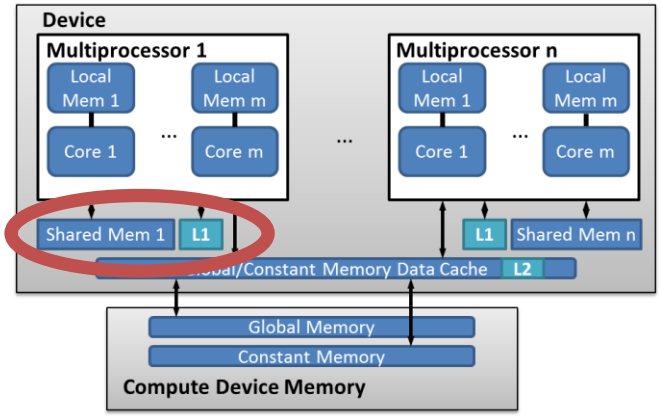
- **Strive for perfect coalescing**
  - Warp should access within contiguous region
  - Align starting address (may require padding)
- **Have enough concurrent accesses to saturate the bus**
  - Process several elements per thread
  - Launch enough threads to cover access latency
- **Try L1 and caching configurations (cc 2.0)**
  - Caching vs non-caching loads (compiler option)
  - 16KB vs 48KB L1

Agenda

- **Kernel optimizations**
  - Control flow
  - Launch configuration
  - Global memory throughput
  - Shared memory
  - Misc
- **GPU-CPU interaction**
  - Minimal data transfer
  - Pinned memory
  - Streams & async memcopies
- **Summary**

Kernel optimizations: Shared memory

- **Smem per MP (10s of KB)**
  - Inter-thread communication within a block
  - Need synchronization to avoid RAW / WAR / WAW hazards

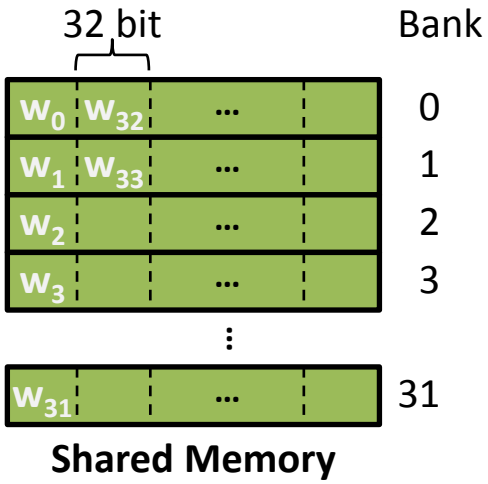


- **Low-latency, high-throughput memory**

  → Cache data in smem to reduce gmem accesses

Kernel optimizations: Shared memory

- **Smem divided into equally sized modules**
- **32 banks (16 banks on cc 1.x)**
  - Each bank: 32-bit (4B) wide
- **Successive 32-bit words go to different banks**
- **Access per 32-threads (16 on cc 1.x)**



- **Bank bandwidth: 32 bits per 2 clock cycles**
- **Banks can be accessed simultaneously**

  if memory loads/stores of n addresses spans n distinct memory banks

- **But: bank conflicts possible**

Kernel optimizations: Shared memory

Bank conflict

- N threads (of 32) access different 32-bit words in the same bank

→ N accesses are serialized

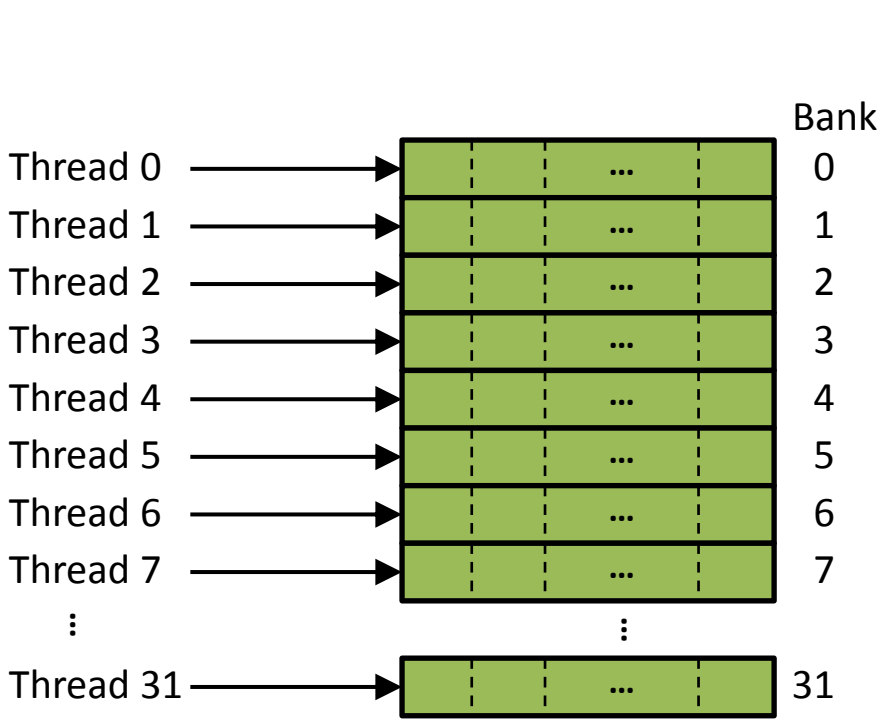
→ Bandwidth decreases by factor: #separate mem requests

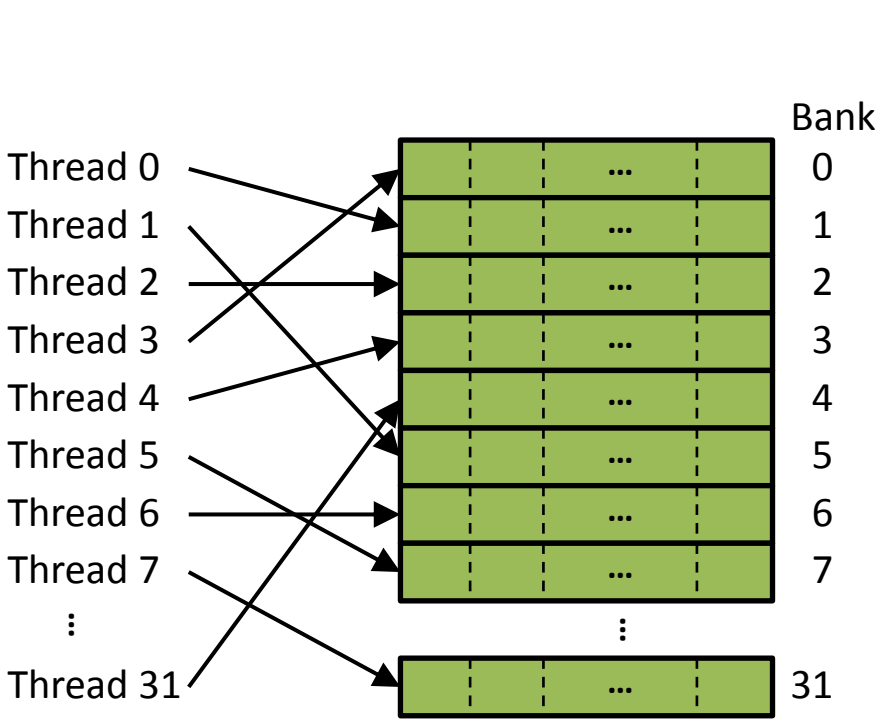
Conflict-free

- Multicast: N threads access the same word in one fetch (since cc 2.0)
  - Could be different bytes within the same word
  - Read: Broadcast to requesting threads
  - Write: Only by one undefined thread

Kernel optimizations: Shared memory

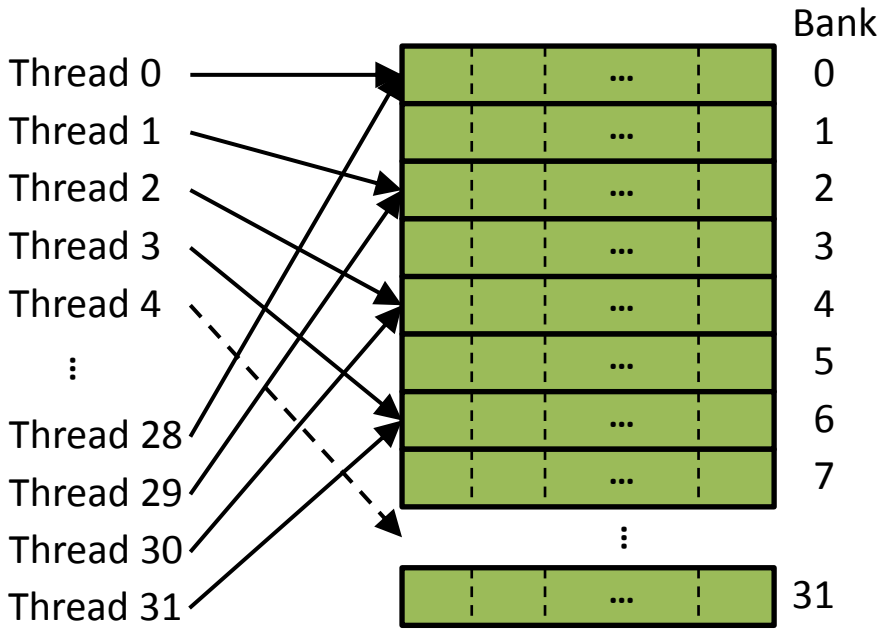
No bank conflicts examples



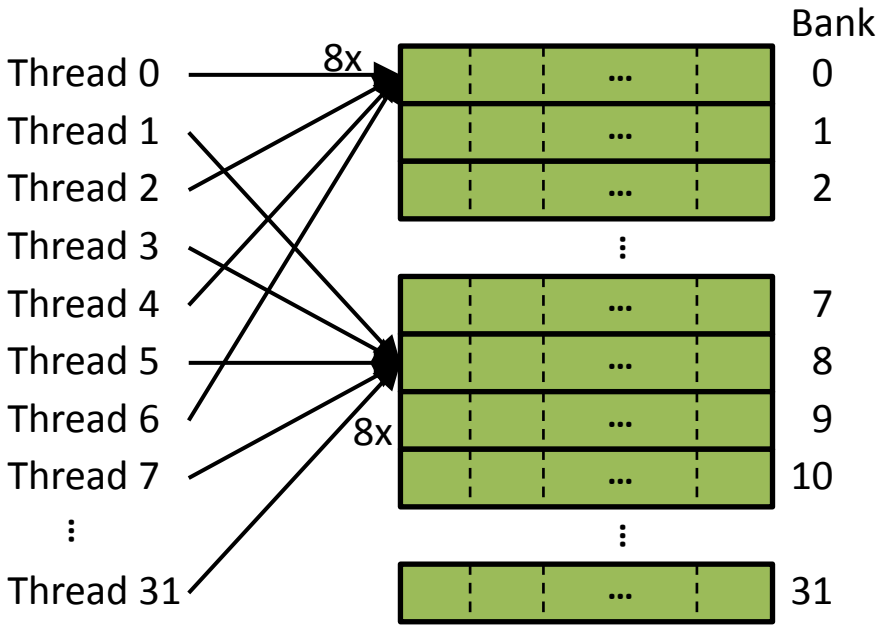


Kernel optimizations: Shared memory

2-way bank conflicts



8-way bank conflicts

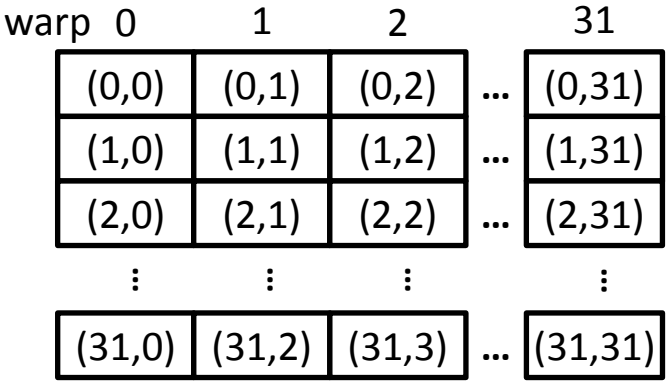


Kernel optimizations: Shared memory

Example

- 32x32 smem array
- Warp accesses a column

→ 32-way bank conflicts

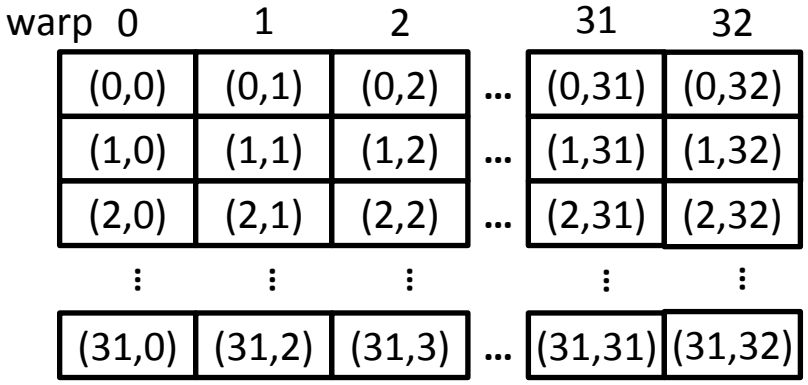


Bank 0, Bank 1, Bank 2, …, Bank 31

Solution: Column for padding

- 32x33 smem array
- Warp accesses a column

→ No bank conflicts



Agenda

- **Kernel optimizations**
  - Control flow
  - Launch configuration
  - Global memory throughput
  - Shared memory
  - Misc
- **GPU-CPU interaction**
  - Minimal data transfer
  - Pinned memory
  - Streams & async memcopies
- **Summary**

Kernel optimizations: Misc

- **Additional “memories”: texture and constant**
  - Read-only
  - Data resides in global memory
  - Read through different caches

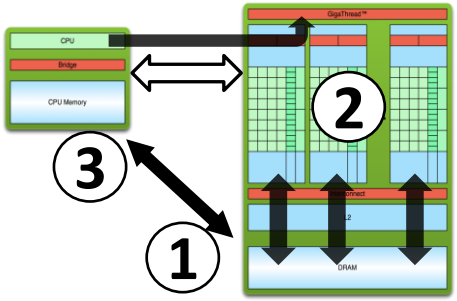
- **Runtime Math Library and Intrinsics**
  - func(): slow but high accuracy (e.g. sin(x), exp(x), pow(x,y))
  - __func(): fast but low accuracy (e.g. __sinf(x), __expf(x), __powf(x,y))

Agenda

- Kernel optimizations
  - Control flow
  - Launch configuration
  - Global memory throughput
  - Shared memory
  - Misc
- **GPU-CPU interaction**
  - Minimal data transfer
  - Pinned memory
  - Streams & async memcopies
- Summary

GPU-CPU interaction: Minimal data transfer

- **Data copies between host + device**
  - PCI Express x16 Gen2: 8 GB/s
  - GPU memory bandwidth (Fermi): 144 GB/s
  - Sync. memory copies block CPU thread



→ Minimize CPU/GPU idling + maximize PCIe throughput

- Copy as little data as possible
- Copy as rarely as possible
- Batch small transfers into one larger one
- Use pinned memory
- Overlap kernel and memcopies

Agenda

- Kernel optimizations
  - Control flow
  - Launch configuration
  - Global memory throughput
  - Shared memory
  - Misc
- **GPU-CPU interaction**
  - Minimal data transfer
  - Pinned memory
  - Streams & async memcopies
- Summary

GPU-CPU interaction: Pinned memory

- **Pinned (non-pageable) memory enables**
  - Faster PCIe copies
  - Memcopies asynchronous with CPU
  - Memcopies asynchronous with GPU
- **Implication**
  - Pinned memory is essentially removed from host virtual memory
- **Usage (CUDA)**
  - C: **`cudaHostAlloc / cudaFreeHost`** (instead of malloc/free)
  - Fortran: **`pinned`** variable type qualifier

Agenda

- Kernel optimizations
  - Control flow
  - Launch configuration
  - Global memory throughput
  - Shared memory
  - Misc
- **GPU-CPU interaction**
  - Minimal data transfer
  - Pinned memory
  - Streams & async memcopies
- Summary

GPU-CPU interaction: Streams & async memcopies

- **Default CUDA API**
  - Kernel launches are asynchronous with CPU
  - Memcopies (D2H, H2D) block CPU thread
  - CUDA calls are serialized by the driver
- **Streams and async functions provide**
  - Memcopies (D2H, H2D) asynchronous with CPU
  - Ability to concurrently execute a kernel and a memcopy
- **Stream = sequence of operations that execute in issue-order on GPU**
  - Operations from different streams can be interleaved
  - A kernel and memcopy from different streams can be overlapped

GPU-CPU interaction: Streams & async memcopies

RWTH AACHEN UNIVERSITY

Overlap kernel and memory copy requirements

- D2H or H2D memcopy from pinned memory
- Device with compute capability ≥ 1.1 (G84 and later)
- Kernel and memcopy in different, non-0 streams

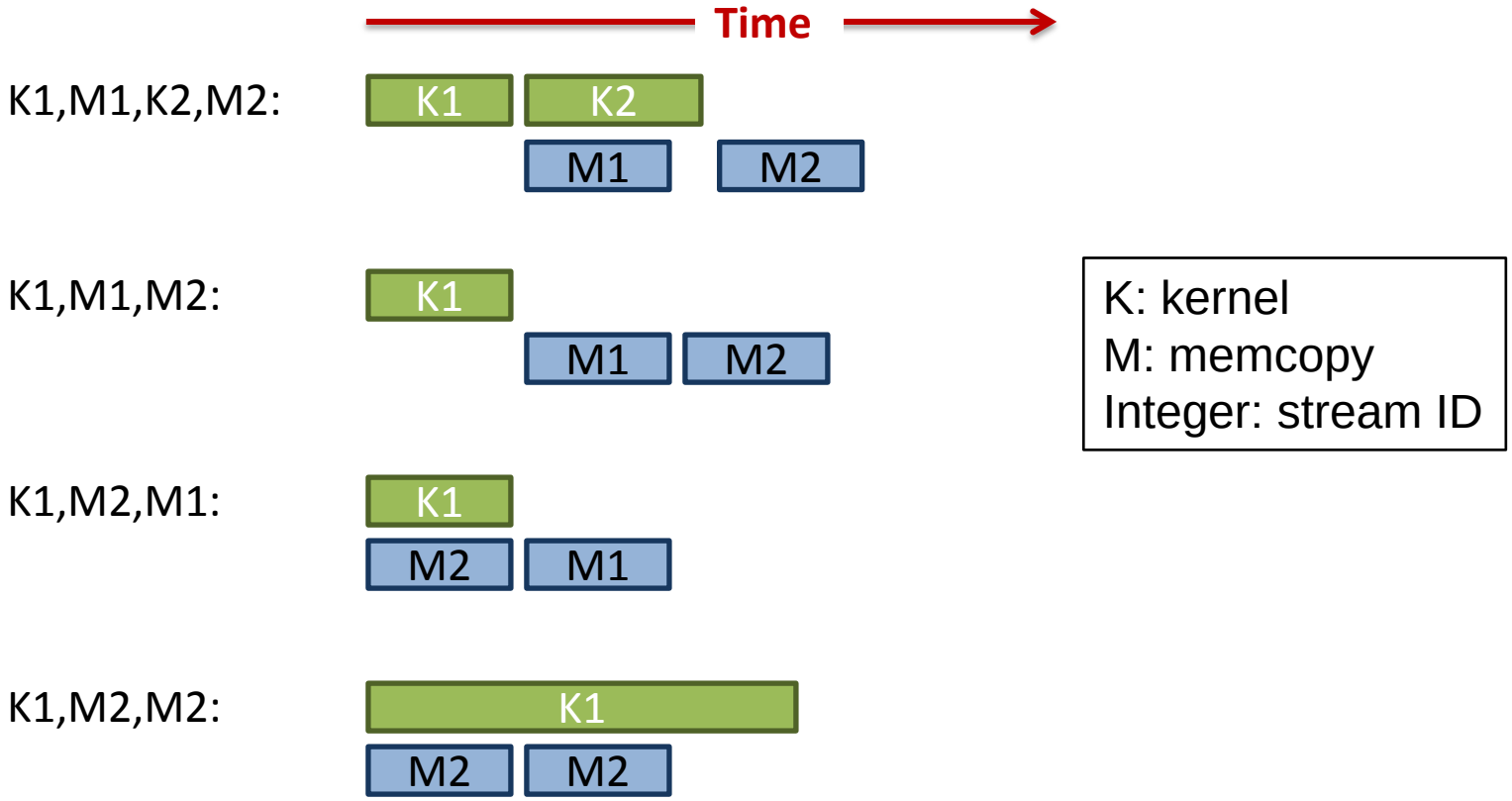
```
cudaStream_t      stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernel<<<grid, block, 0, stream2>>>(…);
```

potentially overlapped

GPU-CPU interaction: Streams & async memcopies

- **Call sequencing important for optimal overlap**
  - CUDA calls are dispatched to the HW in the sequence they were issued
- **A call is dispatched if both are true**
  - Resources are available
  - Preceding calls in the same stream have completed
- **Note that if a call blocks, it blocks all other calls of the same type behind it, even in other streams**
  - Type is one of {kernel, memcopy}

Stream Examples (current HW)



Agenda

- Kernel optimizations
  - Control flow
  - Launch configuration
  - Global memory throughput
  - Shared memory
  - Misc
- GPU-CPU interaction
  - Minimal data transfer
  - Pinned memory
  - Streams & async memcopies
- **Summary**

Summary

- **Avoid branch divergence in kernel**
- **Kernel launch configuration**
  - Launch enough threads per MP to hide latency
  - Launch enough thread blocks to load the GPU
- **Maximize global memory throughput**
  - GPU has lots of bandwidth, use it effectively
- **Use shared memory when applicable (over 1 TB/s bandwidth)**
- **GPU-CPU interaction**
  - Minimize CPU/GPU idling
  - Maximize PCIe throughput
- **Use analysis/ profiling when optimizing**