Practical programming in CUDA

Massimiliano Piscozzi

Università degli Studi di Milano

June 2008

Outline

Introduction

Optimization strategies

Parallel operations

References

Outline

Introduction

Optimization strategies

Parallel operations

References

Parallelism?

1. **Functional parallelism**
   "Different part of data are processed concurrently by separate functional sections on different computational units"
   - No explicit association between kernels and multiprocessors
   - No simulataneous execution of kernels
   - *Big-kernel* approach = waste of shared resources & need of blocks synchronization mechanism
2. **Data parallelism**
   "Data are processed in parallel by distributing elements across different processing units, all of which perform *more or less* the same algorithmic function"
   - *Streams* and *kernels* approach
     - *Stream* = (large) set of homogeneous data
     - *Kernel* = function transforming one or more input streams into one or more output streams
   - Kernels launch as synchronization point
   - *More* is better than *less* . . .

Parallelism?

1. **Functional parallelism**
   "Different part of data are processed concurrently by separate functional sections on different computational units"
   - No explicit association between kernels and multiprocessors
   - No simulataneous execution of kernels
   - *Big-kernel* approach = waste of shared resources & need of blocks synchronization mechanism
2. **Data parallelism**
   "Data are processed in parallel by distributing elements across different processing units, all of which perform *more or less* the same algorithmic function"
   - *Streams* and *kernels* approach
     - *Stream* = (large) set of homogeneous data
     - *Kernel* = function transforming one or more input streams into one or more output streams
   - Kernels launch as synchronization point
   - *More* is better than *less* ...

Parallelism? (cont'd)

- The importance of optimization
  1. High-level programming language
  2. Low-level access to hardware

- Parallelism in CUDA
  1. Blocks scheduling mechanism (implicit)
  2. Thread warps, SIMD groups (explicit)
  3. Shared memory access (explicit)

Parallelism? (cont'd)

- The importance of optimization
    1. High-level programming language
    2. Low-level access to hardware
- Parallelism in CUDA
    1. Blocks scheduling mechanism (implicit)
    2. Thread warps, SIMD groups (explicit)
    3. Shared memory access (explicit)

Parallelism? (cont'd)

- The importance of optimization
  1. High-level programming language
  2. Low-level access to hardware
- Parallelism in CUDA
  1. Blocks scheduling mechanism (implicit)
  2. Thread warps, SIMD groups (explicit)
  3. Shared memory access (explicit)

Parallelism? (cont'd)

- The importance of optimization
  1. High-level programming language
  2. Low-level access to hardware

- Parallelism in CUDA
  1. Blocks scheduling mechanism (implicit)
  2. Thread warps, SIMD groups (explicit)
  3. Shared memory access (explicit)

Parallelism? (cont'd)

- The importance of optimization
  1. High-level programming language
  2. Low-level access to hardware
- Parallelism in CUDA
  1. Blocks scheduling mechanism (implicit)
  2. Thread warps, SIMD groups (explicit)
  3. Shared memory access (explicit)

Outline

Introduction

Optimization strategies

Parallel operations

References

Optimization strategies

- Essential ingredients to write efficient CUDA kernel...

  1. No-branching code
  2. Data Read/Write optimization
     - Prefetching memory
     - Memory coalescing
  3. Bank conflicts resolution

- ...and a *good* parallel algorithm!!
  - PRAM CRCW (Parallel Random Access Machine - Concurrent Read Concurrent Write) model
  - No message passing model

Optimization strategies

- Essential ingredients to write efficient CUDA kernel. . .
  1. No-branching code
  2. Data Read/Write optimization
     - Prefetching memory
     - Memory coalescing
  3. Bank conflicts resolution
- . . . and a *good* parallel algorithm!!
  - PRAM CRCW (Parallel Random Access Machine - Concurrent Read Concurrent Write) model
  - No message passing model

Optimization strategies

- Essential ingredients to write efficient CUDA kernel...
  1. No-branching code
  2. Data Read/Write optimization
     - Prefetching memory
     - Memory coalescing
  3. Bank conflicts resolution
- ...and a *good* parallel algorithm!!
  - PRAM CRCW (Parallel Random Access Machine - Concurrent Read Concurrent Write) model
  - No message passing model

Optimization strategies

- Essential ingredients to write efficient CUDA kernel...
  1. No-branching code
  2. Data Read/Write optimization
     - Prefetching memory
     - Memory coalescing
  3. Bank conflicts resolution
- ...and a *good* parallel algorithm!!
  - PRAM CRCW (Parallel Random Access Machine - Concurrent Read Concurrent Write) model
  - No message passing model

Optimization strategies

- Essential ingredients to write efficient CUDA kernel. . .
  1. No-branching code
  2. Data Read/Write optimization
     - Prefetching memory
     - Memory coalescing
  3. Bank conflicts resolution
- . . . and a *good* parallel algorithm!!
  - PRAM CRCW (Parallel Random Access Machine - Concurrent Read Concurrent Write) model
  - No message passing model

No-branching code

No-branching code = from SPMD to SIMD

- Branching inside a *warp* = serialization
  - No divergence if branch granularity is a whole multiple of *warp* size
- Use a multiple of 32 threads per block
  - Prefer *data padding* than *special cases*

- Low-control flow overhead
  - Small loops unrolled

- **Think in parallel!**
  - Do not rely on any ordering between warps (use __syncthreads())
  - __syncthreads() in a branch = deadlock

No-branching code

No-branching code = from SPMD to SIMD

- Branching inside a *warp* = serialization
  - No divergence if branch granularity is a whole multiple of *warp* size
- Use a multiple of 32 threads per block
  - Prefer *data padding* than *special cases*

- Low-control flow overhead
  - Small loops unrolled

- **Think in parallel!**
  - Do not rely on any ordering between warps (use __syncthreads())
  - __syncthreads() in a branch = deadlock

No-branching code

No-branching code = from SPMD to SIMD

- Branching inside a *warp* = serialization
  - No divergence if branch granularity is a whole multiple of *warp* size
- Use a multiple of 32 threads per block
  - Prefer *data padding* than *special cases*

- Low-control flow overhead
  - Small loops unrolled

- **Think in parallel!**
  - Do not rely on any ordering between warps (use __syncthreads())
  - __syncthreads() in a branch = deadlock

No-branching code

No-branching code = from SPMD to SIMD

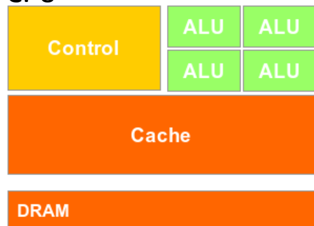
- Branching inside a *warp* = serialization
  - No divergence if branch granularity is a whole multiple of *warp* size
- Use a multiple of 32 threads per block
  - Prefer *data padding* than *special cases*

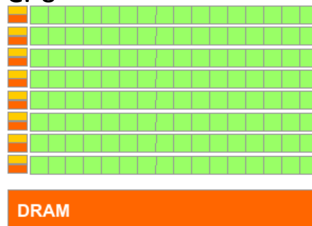
- Low-control flow overhead
  - Small loops unrolled

- **Think in parallel!**
  - Do not rely on any ordering between warps (use __syncthreads())
  - __syncthreads() in a branch = deadlock

Prefetching memory

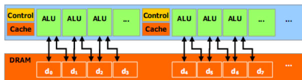
- Maximize arithmetic intensity (many calculations per memory access)
- Latency hiding
  - Help the (young) compiler do a better job
    - Memory instruction followed by independent ALU instructions (if possible)
  - Sometimes it's better to recompute than to cache
  - Exploit block scheduling mechanism = use more than 16 blocks
  - Use prefetching strategy (manual caching)

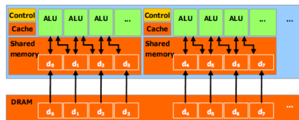




Prefetching strategy

- **Prefetching strategy**
  1. Use threads to cooperatively move data from *device memory* to *shared memory*
  2. Barrier synchronization
  3. Use threads to process data
  4. . . .
  5. Barrier synchronization
  6. Use threads to cooperatively write results to *device memory*





Memory coalescing

- If per-thread memory accesses form a contiguous range of adresses, accesses will be **coalesced** into a single access
  - Coalesced
    - fewer memory accesses
    - bigger data transfers (32-bit, 64-bit, 128-bit instructions)
  - Non coalesced
    - serialization of memory operations
- Memory alignment
  - Explicit alignment of custom types (__align keyword)
  - Prefer **Structure of Arrays** (SoA) than **Array of Structures** (AoS)
- Use **prefetching strategy** to do **coalesced** read/write …
- …use threads cooperation to **permute** data in **shared memory**

Memory coalescing

- If per-thread memory accesses form a contiguous range of adresses, accesses will be **coalesced** into a single access
  - Coalesced
    - fewer memory accesses
    - bigger data transfers (32-bit, 64-bit, 128-bit instructions)
  - Non coalesced
    - serialization of memory operations
- Memory alignment
  - Explicit alignment of custom types (__align keyword)
  - Prefer **Structure of Arrays** (SoA) than **Array of Structures** (AoS)
- Use **prefetching strategy** to do **coalesced** read/write ...
- ... use threads cooperation to **permute** data in **shared memory**

Memory coalescing

- If per-thread memory accesses form a contiguous range of adresses, accesses will be **coalesced** into a single access
  - Coalesced
    - fewer memory accesses
    - bigger data transfers (32-bit, 64-bit, 128-bit instructions)
  - Non coalesced
    - serialization of memory operations
- Memory alignment
  - Explicit alignment of custom types (__align keyword)
  - Prefer **Structure of Arrays** (SoA) than **Array of Structures** (AoS)
- Use **prefetching strategy** to do **coalesced** read/write . . .
- . . . use threads cooperation to **permute** data in **shared memory**

Shared memory & Bank conflicts

- Parallel shared memory access
  - Many threads access memory, memory is divides into **banks**
  - At every cycle: each bank can service one address
  - Successive 32-bit words = successive banks (16 banks)
- Bank conflicts
  - Conflicting accesses are serialized
  - Conflict = same bank! (not same address)
  - Conflicts can occur only inside a SIMD group
  - No conflict if ...
    - ...all threads access *different banks*
    - ...all threads access *the identical address* (broadcast, global data)
- Bank conflict resolution
  - Explicit *stride* based on *tid*
  - Use more shared memory

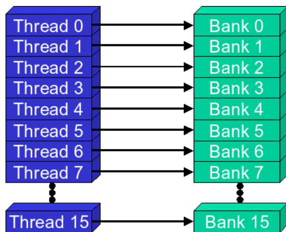
Shared memory & Bank conflicts

- Parallel shared memory access
  - Many threads access memory, memory is divides into **banks**
  - At every cycle: each bank can service one address
  - Successive 32-bit words = successive banks (16 banks)
- **Bank conflicts**
  - Conflicting accesses are serialized
  - Conflict = same bank! (not same address)
  - Conflicts can occur only inside a SIMD group
  - No conflict if ...
    - ...all threads access *different banks*
    - ...all threads access *the identical address* (broadcast, global data)
- Bank conflict resolution
  - Explicit *stride* based on *tid*
  - Use more shared memory

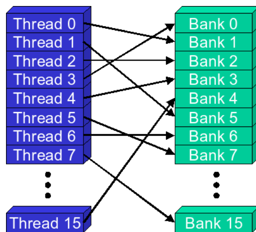
Shared memory & Bank conflicts

- Parallel shared memory access
  - Many threads access memory, memory is divides into **banks**
  - At every cycle: each bank can service one address
  - Successive 32-bit words = successive banks (16 banks)
- **Bank conflicts**
  - Conflicting accesses are serialized
  - Conflict = same bank! (not same address)
  - Conflicts can occur only inside a SIMD group
  - No conflict if . . .
    - . . . all threads access *different banks*
    - . . . all threads access *the identical address* (broadcast, global data)
- Bank conflict resolution
  - Explicit *stride* based on *tid*
  - Use more shared memory

Bank conflicts

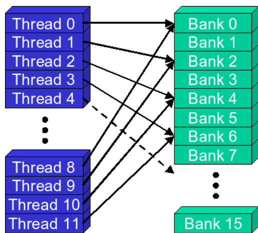


No conflicts

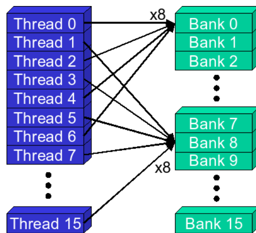


No conflicts

Bank conflicts (cont'd)



2-way conflicts



8-way conflicts

Outline

Introduction

Optimization strategies

Parallel operations

References

Data-Parallel building blocks

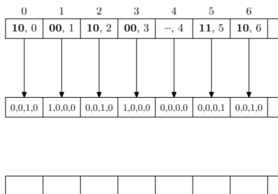
- Data-parallel operations
  - Stream: $S = [a_0, a_1, \ldots, a_{n-1}]$

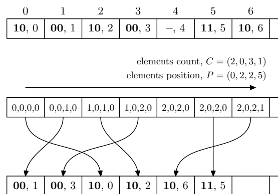
1. **Map**
   - Local function, f
   - $map(S) = [f(a_0), f(a_1), \ldots, f(a_{n-1})]$
   - Excellent data-parallelism, no threads communication
2. **Reduce**
   - Binary associative operator, $\otimes$
   - $reduce(S, \otimes) = a_0 \otimes a_1 \otimes \ldots \otimes a_{n-1}$
   - Pyramidal construction
   - $O(log_2 N)$ steps, $O(N)$ work
3. **Scan**
   - Binary associative operator, $\otimes$
   - Inclusive $scan(S, \otimes) = [a_0, (a_0 \otimes a_1), \ldots, (a_0 \otimes a_1 \otimes \ldots \otimes a_{n-1})]$
   - Exclusive $scan(S, \otimes) = [I, a_0, \ldots, (a_0 \otimes a_1 \otimes \ldots \otimes a_{n-2})]$
   - Common algorithmic pattern: the computation seems inherently sequential, but can be efficiently implemented in parallel
   - $O(log_2 N)$ steps, $O(N)$ work

The importance of scan

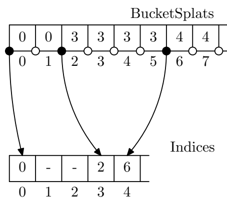
- M. Harris, S. Sengupta, J. Owens:
  Parallel Prefix Sum (Scan) in CUDA.
  GPU Gems 3, Hubert Nguyen, ed. Addison Wesley, August 2007
- Applications:
  - Sorting,
  - Stream compaction,
  - Building data structures (trees and summed-area tables)
- CUDPP: Cuda Data Parallel Primitives Library
  - http://www.gpgpu.org/developer/cudpp/

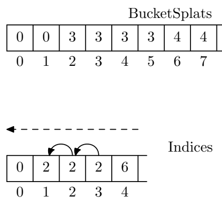
Example: radix-sort

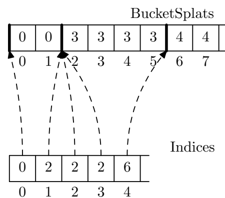




Example: pointers







Outline

Introduction

Optimization strategies

Parallel operations

References

Some references

- Pre-CUDA, but useful
  - A. Lefohn: *Glift: Generic Data Structures for Graphics Hardware*, PhD thesis, Computer Science, University of California, Davis, September 2006.
  - M. Kaas, A. Lefohn, J. D. Owens: *Interactive Depth of Field Using Simulated Diffusion*, Pixar Animation Studios, January 2006.
- CUDA references
  - NVIDIA website & CUDA forum
  - Google: *CUDA*, *G80* keywords

Not only CUDA

- CELL parallel architecture
  - IBM website
    - http://www.research.ibm.com/cell/
  - Cell Broadband Engine
    - http://cell.scei.co.jp/e_download.html
  - Multicore Programming Primer (MIT & Playstation3)
    - http://cag.csail.mit.edu/ps3/