

# CUDA GPU Development

Principles, Techniques and Tools

by Clifford Wolf, April 2013

# Outline

- Overall Work-flow
- GPU Kernel Threads Hierarchy
- Fermi GPU Architecture
- Memory Hierarchy
- Delay Hiding and Occupancy
- Synchronization
- CUDA Streams
- Amdahl's and Little's Law
- Development Tools
- CUDA-enabled Libraries and Compiler extensions
- CUDA beyond Fermi (Kepler, Maxwell, Volta)
- Recommended Reading

# Overall Work-flow

- Create functional C/C++ Implementation
- Optimize and Profile C/C++
  - Identify critical parts
  - Check if GPUs can be used for critical parts
  - Devise a plan for GPU implementation
- Implement for GPU
  - Often GPU version is slower at first
  - Profile and optimize GPU implementation
- Repeat: Profile and move stuff to GPU

# Running GPU Kernels – Big Picture

- Prepare input data
- Copy input data from CPU to GPU
- Launch Kernel(s) on GPU
- Wait for Kernels to finish
- Copy output data from GPU to CPU
- Process output data
- Optional: Use asynchronous API to overlap GPU tasks with CPU or other GPU tasks.

# Kernel Threads Hierarchy

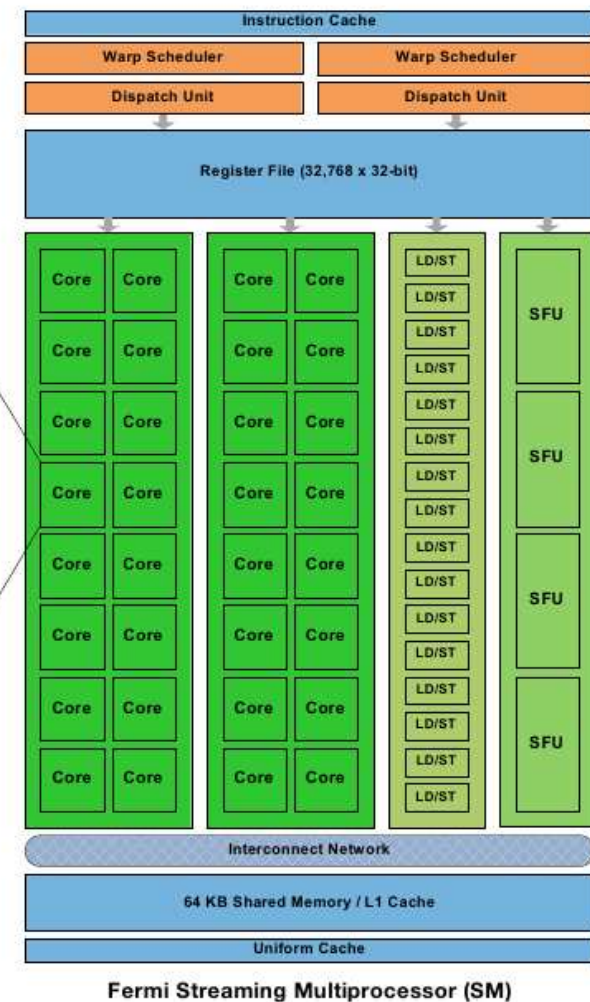
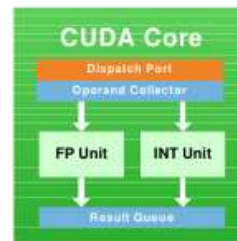
- Kernel Launch
  - $N$  Thread Blocks (Grid)
  - $M$  Threads per Block
  - Total number of threads:  $N * M$
- All threads in a block run on the same SM.
- Blocks may be processed sequentially or in parallel.
- Only threads in the same block can communicate.

# More on Kernel Threads

- Threads only use their thread- and block-id to determine their individual tasks.
  - Use data divergence, not code divergence!
- Coordinates and Dimensions for Grids and Blocks are 3-dimensional
  - Eases mapping to problem in some cases
- 32 consecutive threads in a block belong to the same *warp*.
  - Code divergence within a warp divides instruction throughput!

# Ingredients to a Streaming Multiprocessor (SM)

- Warp scheduler and instruction dispatcher
- 32k-words Register File
- 32x Integer and Floating-Point Core
  - 32-bit ISA with 64-bit support
- 16x Load/Store Unit
- 4x Special Functions Units
- 64kB Shared Memory / L1 Cache
  - 32-bit alignment
  - 32 independent Banks
- Additional Constants and Texture Memories



# CUDA Kernel Memory Hierarchy

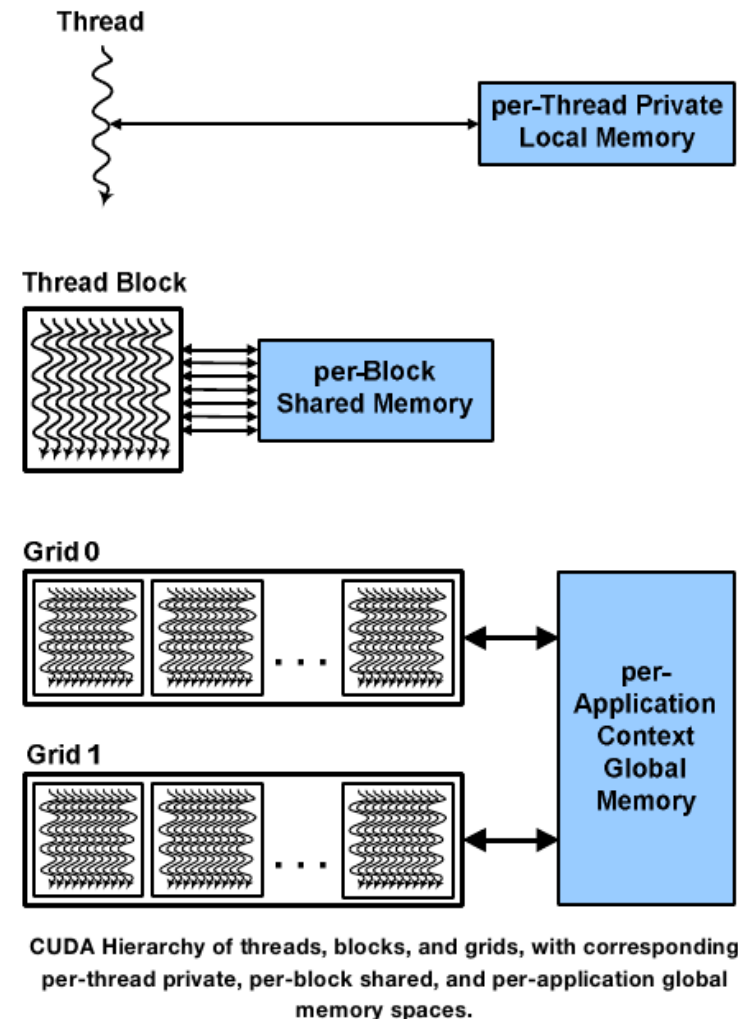
- Thread-Local Memory (Register File)
  - Can only be accessed by the Thread
  - Support for Arrays, etc.
- Shared Memory
  - Can be accessed by all Threads in a Block
  - Shares Resources with L1 Cache
- Global Memory
  - Can be accessed by all Threads
  - L2 Cache shared with memcpy engine
- Unified Address Space for all Memories



# Relation of Thread- and Memory-Hierarchy

## Notes:

- Many blocks can run on the same SM, if the SM has enough local and shared memory for all blocks.
- Shared memory has 32 banks, each 32 bit wide.
- Global memory is organized in cache lines of 128 bytes.
- Use thread- and block-ids to align memory accesses.



# Comparison CPU vs. GPU

- On a CPU:
  - Creating threads is expensive
  - Switching between threads is expensive
  - Having much more threads than cores has a negative impact on performance
  - Threads can do very different things
- On a GPU:
  - Creating many threads at once is cheap
  - Switching between threads is for free
  - Best performance with a couple of 100 threads per SM
  - Threads should not diverge much, at least within a warp.

# Latency Hiding

- Instruction delay: 18-22 clock cycles
- GMEM access delay: 400-800 clock cycles
- Hide delays by having many independently runnable instructions at all times.
  - By using threads (500-1500 per SM)
  - By using independent instructions in a thread
- GPUs are pretty slow!  
The trick is to be slow massively parallel!

# Occupancy

- The ratio of number of threads on a SM and the max. number of threads possible per SM is called *Occupancy*.
- Keeping occupancy high is a simple way of improving latency hiding.
- Limiting factors on occupancy:
  - Number of register per Thread
  - Shared registers per Thread block
  - Max. Number of Thread blocks per SM
  - Many more (see *NVIDIA Occupancy Calculator*)
- But: Occupancy is only a crude rule of thumb!  
(see *Better Performance at Lower Occupancy*, Vasily Volkov, GTC 2010)

# Synchronization

- Between Thread blocks:
  - Using atomic global memory access
  - Warning: Slow!  
Consolidate Data in shared memory first and keep rate of atomic operations low.
- Between Threads in a block:
  - Using `__syncthreads()`
  - Using atomic memory access (shared or global)
  - Also use this within a warp!

# CUDA Streams

- CUDA knows three kinds of operation:
  - Memcpy H2D, Memcpy D2H, Kernel Launch
- Fermi can execute in parallel:
  - 1 Memcpy H2D and 1 Memcpy D2H
  - Up to 16 Kernels
- Operations must be committed to different streams to be executable in parallel.
- One blocking operation of a kind blocks all operations of the same kind committed later.

# Various Additional Hardware Features

- Various atomic functions
- Warp vote functions
- Constant and Texture Memory
- Surface Functions (Array accessors)
- Synchronization with count / and / or
- Various special floating point functions
- Option to turn off L1 cache

Before we continue...

Questions?



# Amdahl's Law

- The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.  
(wikipedia)
- Examples:
  - Seq. fraction: 5%  $\rightarrow$  max. speedup: 20x
  - Seq. fraction: 50%  $\rightarrow$  max. speedup: 2x

# Little's Law

- The long-term average number of customers in a stable system  $L$  is equal to the long-term average effective arrival rate,  $\lambda$ , multiplied by the (Palm-)average time a customer spends in the system,  $W$ ; or expressed algebraically:  $L = \lambda W$ . (wikipedia)
- Example:
  - $W = 20$  cycles latency / ALU operation
  - $\lambda = 384$  processor cores
  - $L = \lambda W = 20 * 384 = 7680$  independent ops required at all times for effective latency hiding

# Development Tools

- Compiler itself is a command line tool: nvcc
  - Input file is combined C/C++ and CUDA code
  - Using GCC or VC++ for host code
- IDE with extensive CUDA support:
  - NVIDIA Nsight
  - Versions for Eclipse or Visual Studio
  - Includes tools for debugging and profiling
- Stand-alone profiler: nvprof, nvvp

# CUDA-enabled Libraries

- Don't re-invent the wheel: use CUDA-enabled Libraries!
- Often the performance bottleneck is not an application-specific algorithm. Use a library if possible and fast enough.
- For common math problems (incl. in CUDA Toolkit):
  - CUBLAS (dense linear algebra)
  - CUSPARSE (sparse linear algebra)
  - CUFFT (FFT)
- For common algorithmic problems (incl. In CUDA Toolkit):
  - Thrust (STL-style C++ template library)
- High-quality 3rd Party libraries are available but very expensive.

# Compiler extensions for CUDA

- Nvcc basically is a C/C++ compiler extension
  - Device code and kernel code in same file
  - <<<...>>>-Syntax for Kernel Launches
- OpenACC
  - Using pragmas, similar to OpenMP
  - Much more abstraction than with CUDA
  - Support for different Accelerators  
(NVIDIA, AMD and Intel as of Nov. 2012)

# OpenCL vs. CUDA

- OpenCL is similar to OpenGL shader API.
- Disadvantages of OpenCL compared to CUDA:
  - No pre-compiled device code
  - No support for C++ data types or templates
  - No support for newest CUDA features
- Advantages of OpenCL compared to CUDA:
  - Vendor-independent standard
  - Supports non-GPU accelerators
  - Will catch up with architecture eventually (see OpenGL in mid 2000's)

# CUDA Roadmap beyond Fermi

- Kepler (available since 2012):
  - Dynamic Parallelism, HyperQ
  - Warp shuffle functions, Funnel shift, ...
- Maxwell (announced for 2014):
  - 64-bit ARM Core (Project Denver)
  - Unified Memory Architecture (Host ↔ Device)
- Volta (expected for 2016):
  - Stacked DRAM (1 TB/s GMEM bandwidth)

# Theoretical Performance

- Upper Bound on Fermi (GTX580):
  - Global Memory bandwidth: 192 GB/s
  - Floating point performance: 1581 GFLOS
- Upper Bound on Kepler (GTX680):
  - Global Memory bandwidth: 192 GB/s
  - Floating point performance: 3090 GFLOS

Source: Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs, Junjie Lai, Andre Seznec, CGO 2013



# Should I port my algorithm to CUDA?

- Yes, if the following statements are correct:
  - It has many independent operations that could be parallelized.
  - It is not limited by initial data load (can be assumed if harder than  $O(n)$ ).
  - It contains uniform pattern of instructions.
  - It can be implemented without I/O during phases of computation.
  - There is no existing CUDA-aware library that does the job.

# Recommended Reading

- Included with the CUDA Toolkit:
  - CUDA C Best Practices Guide
  - CUDA C Programming Guide
  - CUDA Compiler Driver NVCC
- External Resources:
  - NVIDIA Fermi and Kepler Whitepapers
  - CUDA By Example (J. Sanders and E. Kandrot)
  - <http://en.wikipedia.org/wiki/CUDA>

# Recommended Watching

- Videos from conferences and webinars:
  - GTC On-Demand
  - NVIDIA GPU Computing Webinars
- Especially Recommended:
  - Introduction To GPU Computing & CUDA (Sarah Tariq)
  - CUDA Concurrency & Streams (Steven Rennich)
  - Better Performance at Lower Occupancy (Vasily Volkov)
  - Fundamental Performance Optimizations for GPUs (Paulius Micikevicius)
  - GPU Performance Analysis and Optimization (Paulius Micikevicius)
  - Analysis-Driven Performance Optimization (Paulius Micikevicius)

Questions?