Introduction to CUDA

Overview

- HW computational power
- Graphics API vs. CUDA
- CUDA glossary
- Memory model, HW implementation, execution
- Performance guidelines
- CUDA compiler
- C/C++ Language extensions
- Limitations & Advantages

Computational power

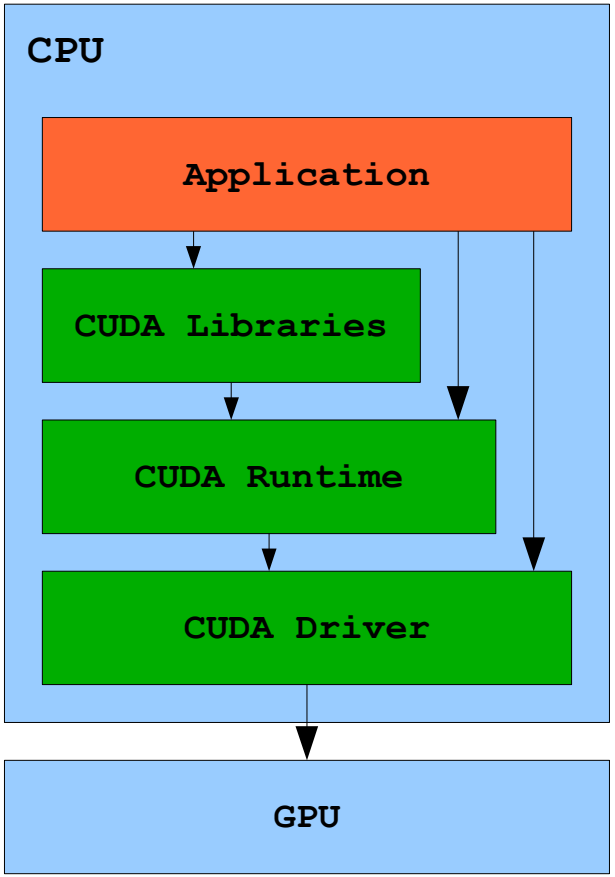
- FLOPS – # float operations per second

HW	Intel Core 2 Extreme QX6850	IBM CELL/BE	Nvidia GeForce 8800 GTS
Clock Speed (GHz)	3.00	3.20	1.2
Processors	4	8	96
Bits per register	128	128	32
32bit floats per register	4	4	1
Parallel operations per processor	1	1/2*	1/3*
Paralel 32bit floats	**16**	**32**/64*	**96**/288*
GFLOPS	**48.0**	**102.4**/204.8*	**115.2**/345.6**

- * perform a load, store, shuffle, channel or branch operation in parallel with a computation.
- ** parallel MAD (a=b*c + d) and ADD (a=a+b)
- 8800 ULTRA 1.5 GHz 128 processors = 192.0 / 576.0** GFLOPS

Compute **U**nified **D**evice **A**rchitecture

- HW & SW architecture
  - No need of graphics API
  - HW
    - GeForce 8 series (G8x)
    - Quadro FX 5600/4600
  - SW
    - CUDA Driver
    - CUDA Runtime
    - CUDA Libraries
    - CUDA Compiler (nvcc)



Graphics API vs. CUDA

- Graphics API
  - API
  - Program
    - External
    - Language
      - GLSL,Cg,HLSL
      - C/C++ like
    - Types (function-like)
      - Geometry Shader
      - Vertex Shader
      - Fragment Shader
- CUDA
  - API
  - Program
    - Internal
    - Language
      - C/C++ extension
    - Types (program-like)
      - Kernel

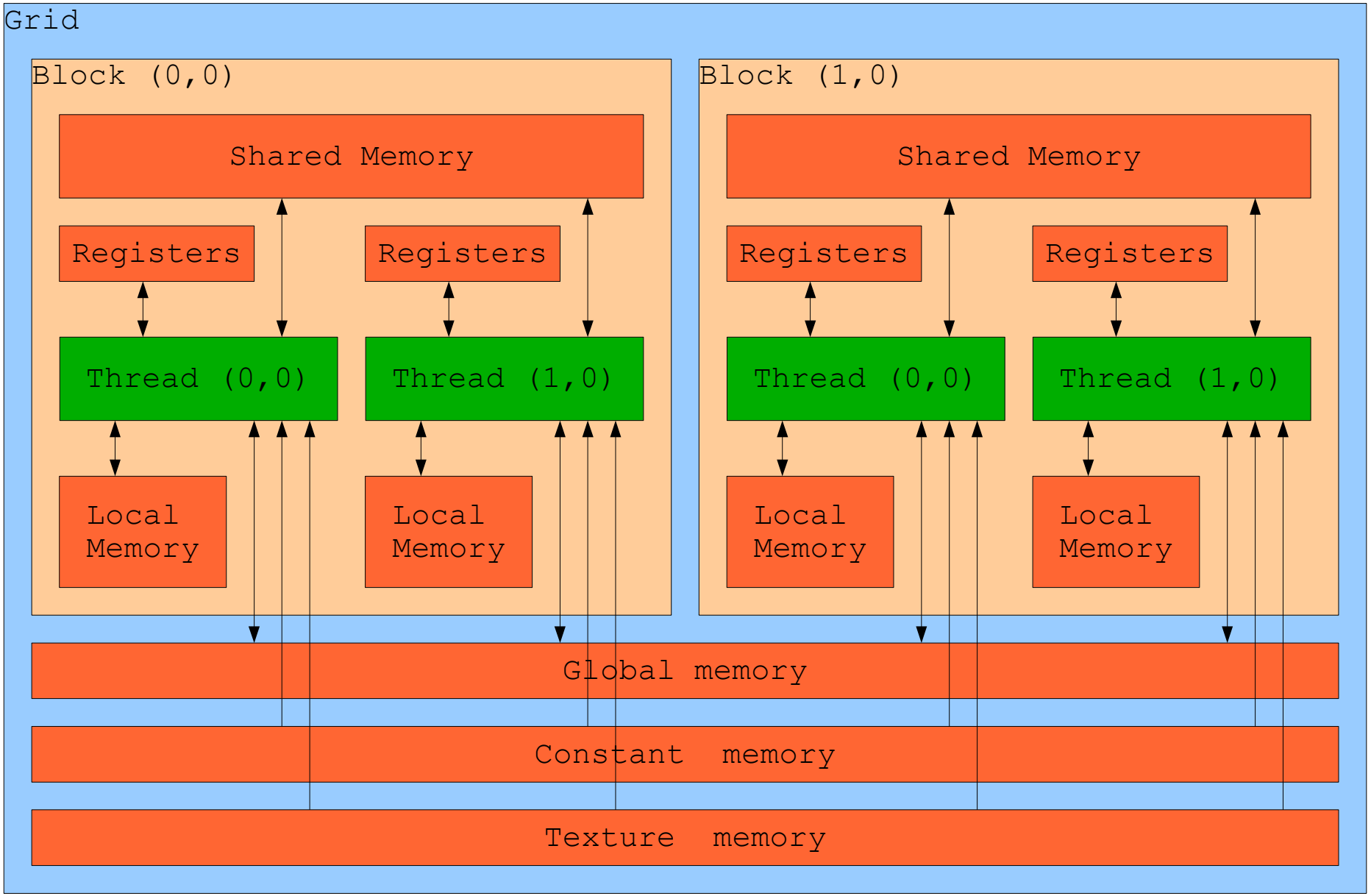
Graphics API vs. CUDA

- Graphics API
  - Fragment shader limited to outputting 32 floats at a pre-specified location
  - store data in textures (packing long arrays into 2D textures, extra addressing math)
- CUDA
  - scattered writes (unlimited # of stores to any address)
  - load from any address
  - shared fast memory (currently 16KB per multiprocessor) accessible in parallel by blocks of threads

CUDA glossary

- *Thread*
  - Running computation on a single scalar processor
- *Kernel*
  - Program running on device
- *Thread Block*
  - 3D block of cooperative threads, that share fast memory
- *Grid of Thread Blocks*
  - 2D grid of ***Thread Blocks*** of same dimensionality executing the same ***Kernel***
  - No communication between Thread Blocks !

Memory model



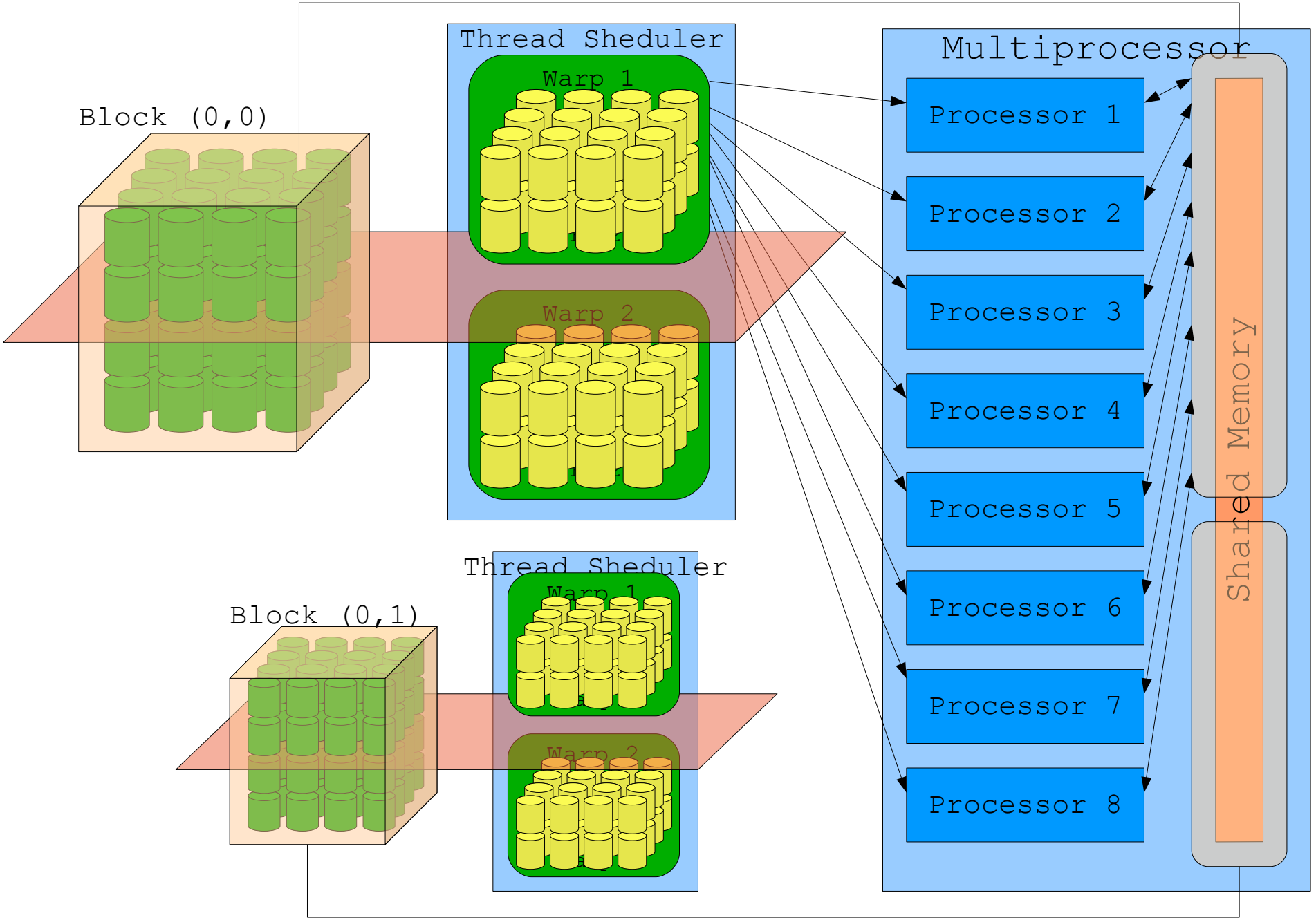
HW implementation

- Device
  - Set of Multiprocessors (GTS 12x, GTX 16x)
- Multiprocessor
  - Set of (scalar) processors (8x)
  - Shared memory (16 kB)
  - SIMD architecture
    - every processor executes the same instruction on different data
  - Processes one or more thread blocks
- Warp
  - SIMD group of threads (32) from the same block

Execution

- Grid of thread blocks -> one or more thread blocks on each multiprocessor
- One thread block on only one multiprocessor
  - efective usage of shared memory
- Block split into SIMD groups of threads (warps)
  - Equal number of threads (warp size) in each warp
- Each of the Block warps is executed by the Multiprocessor in a SIMD fashion
- Thread sheduler periodically switches between warps

Execution model

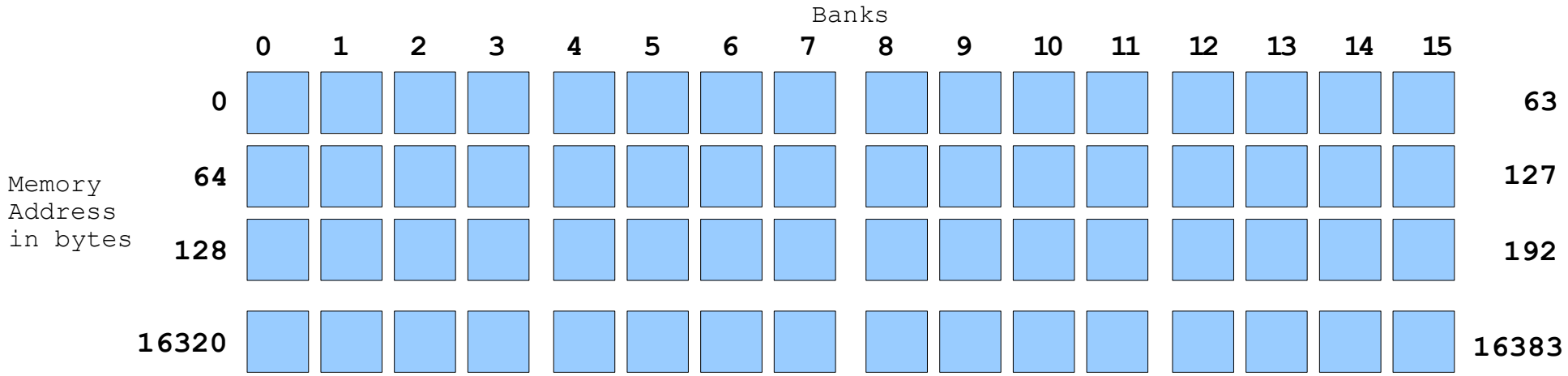


Execution model

- 1 x Multiprocessor (8 processors)
  - Thread Block = n-Threads (n <= 512)
    - Warp = 32 threads
      - 8 processors = 8 parallel threads
      - 1 x SIMD = 4 clock cycles (4 x 8 processors = whole Warp)
    - Next Warp(s)
  - Next Thread Block(s)
    - Only if enough registers & shared memory
- Other multiprocessors
  - Concurently processes their Blocks from Grid

Memory access

- 16 banks of shared memory
  - 32 bit words
- Bank conflict
  - Access of the same bank in the same clock cycle by threads from the same half-warp (16 threads)
  - Thread A accesses address N*4
  - Thread B accesses address (N+k*16)*4



Performance guidelines

- Threads per block
  - Multiple of 64
  - 64 minimum + multiple blocks per multiprocessor
  - 192-256 recomended
  - Use shared memory, avoid bank conflicts
- Number of blocks
  - 100+
  - 1000 to scale for future devices
  - 2+ per multiprocessor
    - Avoid wasting shared memory per block / registers per thread (8192 registers per multiprocessor)

CUDA Compiler

- C/C++ like source code
  - 'device' specific code
  - 'host' specific code
- NVCC compilation
  - Separate 'device' and 'host' specific code
  - Compile 'device' specific code to CUDA binary
  - Merge CUDA binary with 'host' code
  - Call standard C/C++ compiler and linker on the merged code

C/C++ Language 'extensions'

- Function type qualifiers
  - __device__ : device only
  - __global__ : exec on device, call from host
  - __host__ : exec on host, call from host
- Variable type qualifiers
  - __device__ : on the device
  - __constant__ : in constant memory
  - __shared__ : in shared memory of a thread block

C/C++ Language 'extensions'

- Build-in Vector types - {(u)inttype,float}{1,2,3,4}
- Build-in Variables
  - gridDim
    - dim3 - dimesion of grid
  - blockIdx
    - uint3 – block index within the grid
  - blockDim
    - dim3 – dimesion of the block
  - threadIdx
    - uint3 – thread index within the block

Execution configuration

- Function declartation
  - __global__ void Func(float parameter);
- Function execution
  - Func<<< Dg, Db, Ns >>>(parameter);
  - Dg : dimension and size of the grid
    - Dg.x*Dg.y -# block launched
  - Db : dimension and size of each block
    - Db.x*Db.y*Db.z -# threads per block
  - Ns : optional, # bytes of shared memory dynamicaly allocated

Memory management
Runtime API

- device memory
  - cudaMalloc(),cudaMallocPitch(),cudaFree()
- host memory
  - pageable
    - standard functions (malloc(),free(),...)
  - page-locked
    - cudaMallocHost(),cudaFreeHost()
- copying
  - cudaMemcpy()
    - cudaMemcpyKind
      - cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice

CUDA Libraries

- CUFFT
  - Fast fourier transform
    - 1D,2D,3D
    - Real -> Complex,Complex->Complex,Complex->Real
  - Emulation
- CUBLAS
  - Basic linear algebra subroutines
    - scalar,vector,matrix
  - Emulation

Current limitations

- SW limits
  - No streaming support
    - Memory copying is synchronous
- G8x
  - Double to float conversion
  - Only 24 bit integer multiply HW support
    - 32bit integer mul is compiled into multiple instructions

Advantages

- GPU & CPU can run in parallel
  - CUDA 1.0 Kernel invocation is asynchronous

Future

- HW
  - Double precision
  - Native 32-bit integer
  - More multiprocessors
- SW
  - CUDA 1.1
    - part of the display drivers
    - async execution
    - double pecision support

References

- NVidia CUDA Programming guide 1.0
  - http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf
- NVidia CUDA 1.0 FAQ
  - http://forums.nvidia.com/index.php?showtopic=36286
- Technical Brief NVIDIA GeForce 8800 GPU Architecture Overview
  - http://www.nvidia.com/object/IO_37100.html
- beta CUDA 1.1
  - http://forums.nvidia.com/index.php?showtopic=51026