GPU Computing - CUDA

A short overview of hardware and programing model

Pierre Kestener[1]

[1]CEA Saclay, DSM, Maison de la Simulation

Saclay, June 12, 2012

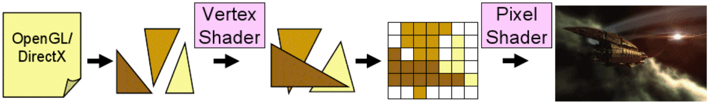
Atelier AO and GPU

Content

- **Short historical view**
- **Main differences between CPU and GPU**
- **CUDA Hardware : differences with CPU**
- **CUDA software abstraction / programing model**
  - **SIMT - Single Instruction Multiple Thread**
  - **Memory hierarchy**
- **OpenCL ??**

Summary

GPU evolution: before 2006, i.e. CUDA



- GPU == dedicated hardware for **graphics *pipeline***
- GPU main function : **off-load graphics task from CPU to GPU**
- GPU: **dedicated hardware for specialized tasks**
- **"All processors aspire to be general-purpose."**
  – Tim Van Hook, Graphics Hardware 2001
- 2000's : ***shaders*** (programmable functionalities in the graphics *pipeline*) : low-level vendor-dependent assembly, high-level Cg, HLSL, etc...
- **Legacy GPGPU** (before CUDA, ~ 2004), premises of GPU computing

The Evolution of GPUs for General Purpose Computing,
par Ian Buck
http://www.nvidia.com/content/GTC-2010/pdfs/2275_GTC2010.pdf

Floating-point computation capabilities in GPU ?

Floating point computations capability implemented in GPU hardware

- **IEEE754 standard** written in mid-80s
- Intel 80387 : first floating-point coprocessor IEEE754-compatible
- Value = $(-1)^S \times M \times 2^E$, denormalized, infinity, NaN; rounding algorithms quite complex to handle/implement
- FP16 in 2000
- **FP32 in 2003-2004** : simplified IEEE754 standard, float point rounding are complex and costly in terms of transistors count,
- **CUDA 2007** : rounding computation fully implemented for + and * in 2007, denormalised number not completed implemented
- **CUDA Fermi : 2010** : 4 mandatory IEEE rounding modes; Subnormals at full-speed (Nvidia GF100)
- links:
  http://homepages.dcc.ufmg.br/~sylvain.collange/talks/raim11_scollange.pdf

GPU computing - CUDA hardware - 2006

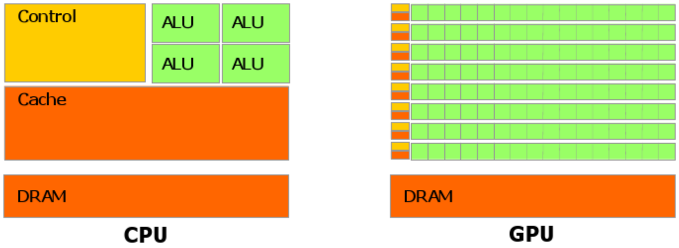
CUDA : Compute Unified Device Architecture

- Nvidia Geforce8800, 2006, introduce a **unified architecture** (only one type of *shader processor*)
- first generation with hardware features designed with GPGPU in mind: almost full support of IEEE 754 standard for single precision floating point, random read/write in external RAM, memory cache controlled by software
- **CUDA ==**
  new hardware architecture +
  new programming model/software abstraction (a *C-like* programming language + development tools : compiler, SDK, librairies like cuFFT)

Summary

From multi-core CPU to manycore GPU

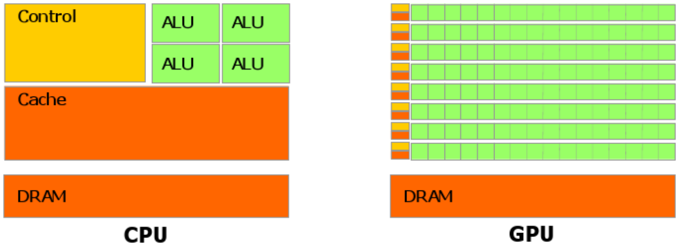
Architecture design differences between manycore GPUs and general purpose multicore CPU ?



- **Different goals produce different designs:**
  - **CPU** must be good at everything, parallel or not
  - **GPU** assumes work load is highly parallel

From multi-core CPU to manycore GPU

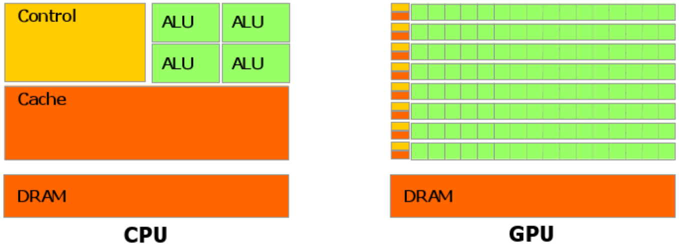
Architecture design differences between manycore GPUs and general purpose multicore CPU ?



- **CPU** design goal : optimize architecture for sequential code performance : minimize latency experienced by **1 thread**
  - sophisticated (i.e. large chip area) control logic for instruction-level parallelism (branch prediction, out-of-order instruction, etc...)
  - CPU have large cache memory to reduce the instruction and data access latency

From multi-core CPU to manycore GPU

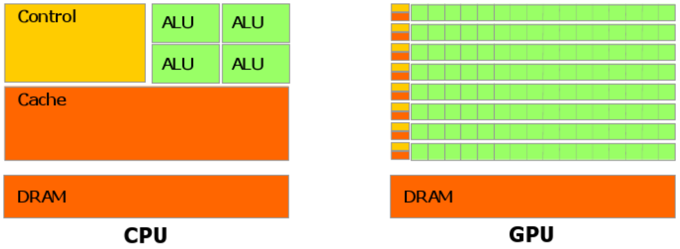
Architecture design differences between manycore GPUs and general purpose multicore CPU ?



- **GPU** design goal : maximize throughput of **all threads**
  - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency => skip the big caches
  - share control logic across many threads

From multi-core CPU to manycore GPU

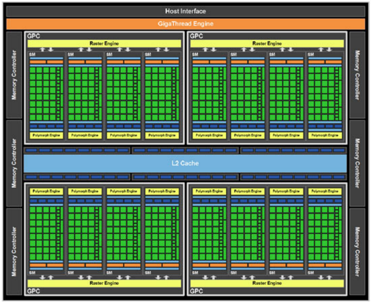
Architecture design differences between manycore GPUs and general purpose multicore CPU ?

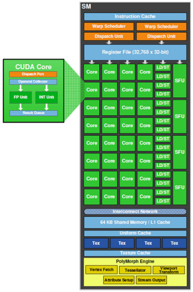


- GPU takes advantage of a **large number of execution threads** to find work to do when other threads are waiting for long-latency memory accesses, thus **minimizing the control logic** required for each execution thread.

Nvidia Fermi hardware (2010)

- **Streaming Multiprocessor** (32 cores), hardware control, queuing system
- **GPU = scalable array of SM** (up to 16 on Fermi)
- **warp: vector of 32 threads**, executes the same instruction in lock-step
- **throughput limiters**: finite limit on warp count, on register file, on shared memory, etc...

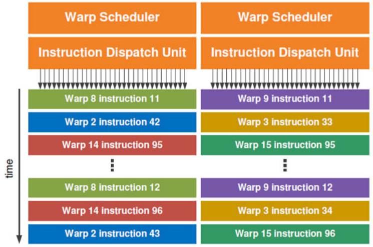




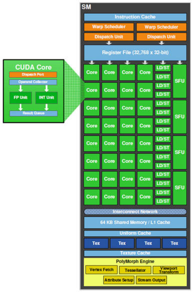
Streaming Multiprocessor (SM)

Nvidia Fermi hardware (2010)

- **Streaming Multiprocessor** (32 cores), hardware control, queuing system
- **GPU = scalable array of SM** (up to 16 on Fermi)
- **warp: vector of 32 threads**, executes the same instruction in lock-step
- **throughput limiters**: finite limit on warp count, on register file, on shared memory, etc...



Dual warp scheduling



Streaming Multiprocessor (SM)

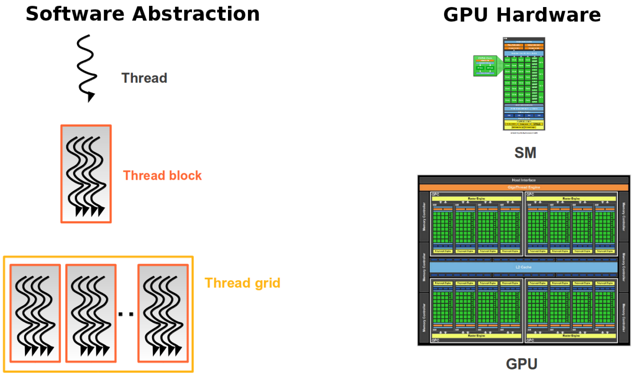
Nvidia Fermi hardware (2010)

CUDA Hardware (HW) key concepts

- Hardware thread management
  - HW thread launch and monitoring
  - HW thread switching
  - up to 10 000's lightweight threads
- SIMT execution model
- Multiple memory scopes
  - Per-thread private memory : (register)
  - Per-thread-block shared memory
  - Global memory
- Using threads to hide memory latency
- Coarse grained thread synchronization

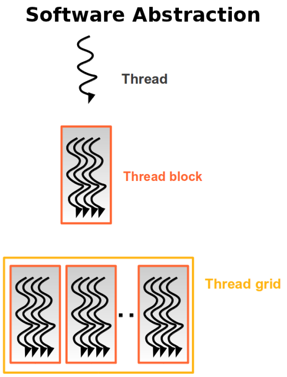
CUDA - connecting program and execution model

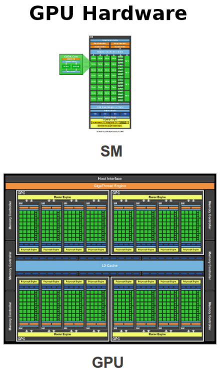
- **Need a programing model to efficiently use such hardware**; also provide **scalability**
- Provide a simple way of partitioning a computation into fixed-size blocks of threads



CUDA - connecting program and execution model

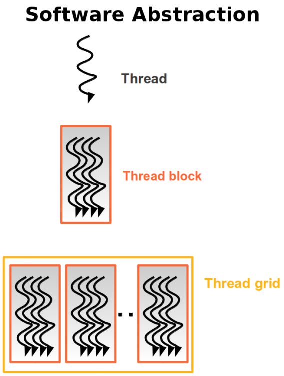
- **Total number of threads** must/need be quite larger than number of cores
- **Thread block** : **logical array of threads**, large number to hide latency
- **Thread block size** : control by program, specify at runtime, better be a multiple of warp size (i.e. 32)

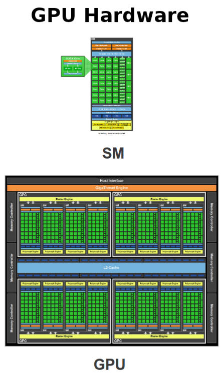




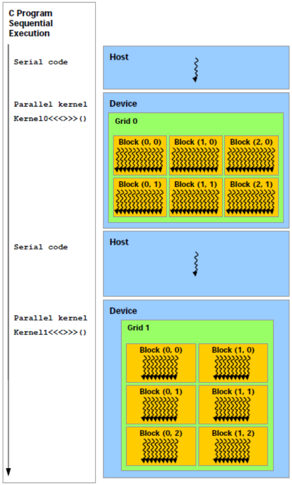
CUDA - connecting program and execution model

- **Must give the GPU enought work to do !** : if not enough thread blocks, some SM will remain idle
- **Thread grid** : **logical array of thread blocks** distribute work among SM, several blocks / SM
- **Thread grid** : chosen by program at runtime, can be the total number of thread / thread block size or a multiple of # SM





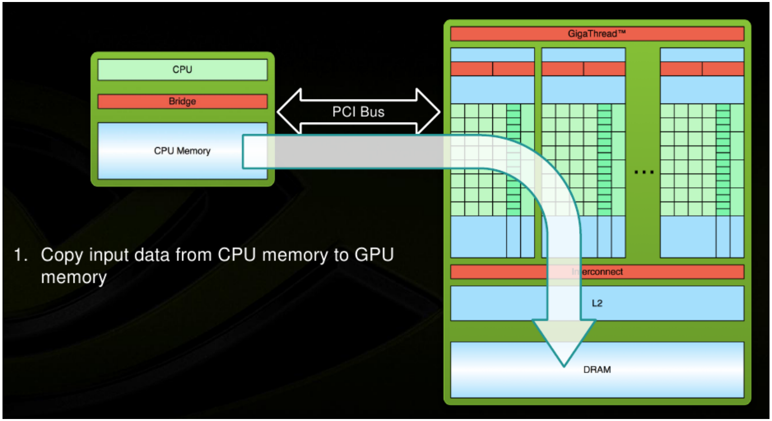
CUDA : heterogeneous programming



- heterogeneous systems : CPU and GPU have separated memory spaces (*host* and *device*)
- CPU code and GPU code can be in the same program / file (pre-processing tool will perform separation)
- the programmer focuses on code parallelization (algorithm level) not on how he was to schedule blocks of threads on multiprocessors.

CUDA : heterogeneous programming

Current GPU execution flow

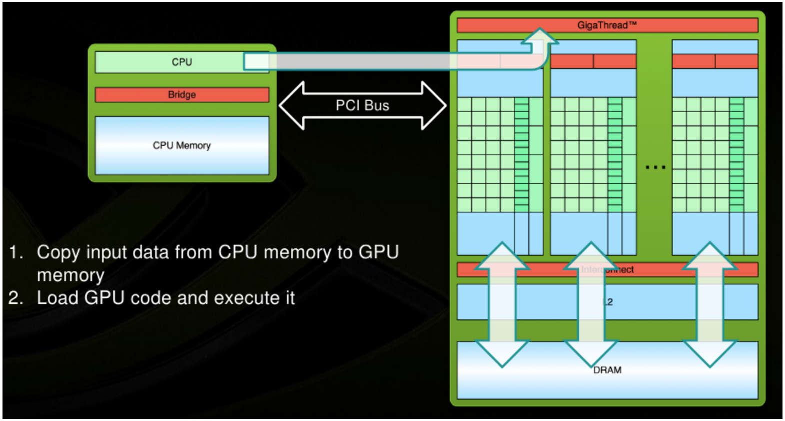


1. Copy input data from CPU memory to GPU memory

reference: Introduction to CUDA/C, GTC 2012

CUDA : heterogeneous programming

Current GPU execution flow

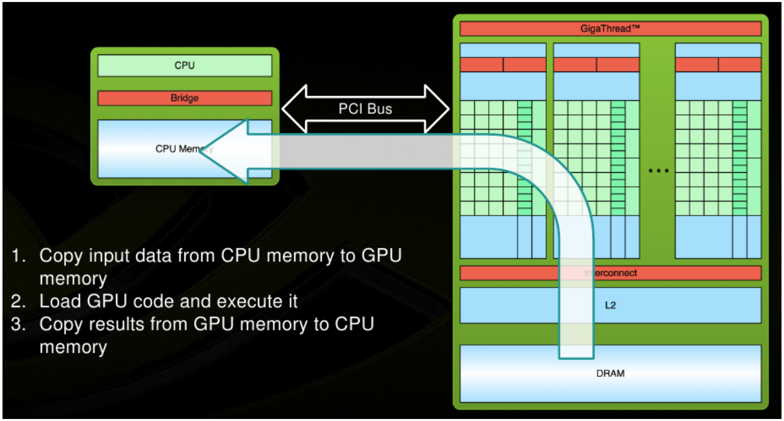


1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it

reference: Introduction to CUDA/C, GTC 2012

CUDA : heterogeneous programming

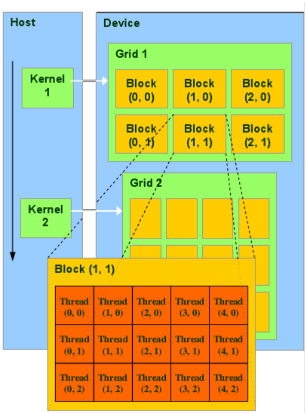
Current GPU execution flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it
3. Copy results from GPU memory to CPU memory

reference: Introduction to CUDA/C, GTC 2012

CUDA : programming model (PTX)



- a block of threads is a **CTA (Cooperative Thread Array)**
- **Threads are indexed inside a block; use that index to map memory**
- write a program once for a *thread*
- run this program on multiple *threads*
- **block** is a logical array of threads indexed with *threadIdx* (**built-in variable**)
- **grid** is a logical array of blocks indexed with *blockIdx* (**built-in variable**)

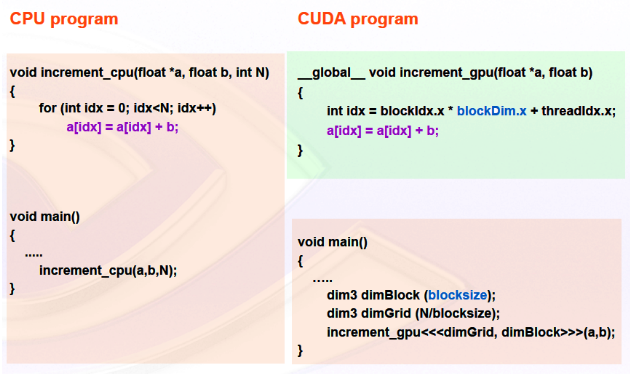
Summary

CUDA C/C++

- **CUDA C/C++**
  - Large subset of C/C++ language
  - CPU and GPU code in the same file; preprocessor to filter GPU specific code
  - Small set of extensions to enable heterogeneous programming: new keywords
  - **A runtime/driver API**
    - **Memory management:** cudaMalloc, cudaFree, ...
    - **Device management:** cudaChooseDevice, probe device properties (# SM, amount of memory , ...)
    - **Event management:** profiling, timing, ...
    - **Stream management:** overlapping CPU-GPU memory transfert with computations, ...
    - **...**
- **Terminology**
  - Host: CPU and its memory
  - Device: GPU and its memory

CUDA Code walkthrough

- **Data parallel model**
- **Use intrinsic variables `threadIdx` and `blockIdx` to create a mapping between threads and actual data**



CPU program

```
void increment_cpu(float *a, float b, int N)
{
      for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}


void main()
{
    .....
      increment_cpu(a,b,N);
}
```

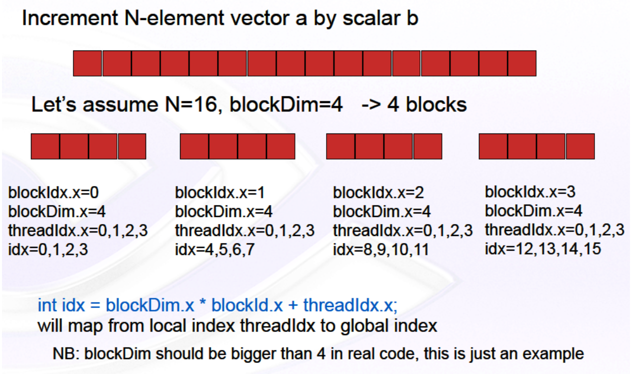
CUDA program

```
__global__ void increment_gpu(float *a, float b)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      a[idx] = a[idx] + b;
}


void main()
{
    .....
      dim3 dimBlock (blocksize);
      dim3 dimGrid (N/blocksize);
      increment_gpu<<<dimGrid, dimBlock>>>(a,b);
}
```

CUDA Code walkthrough

- **Data parallel model**
- **Use intrinsic variables `threadIdx` and `blockIdx` to create a mapping between threads and actual data**


Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

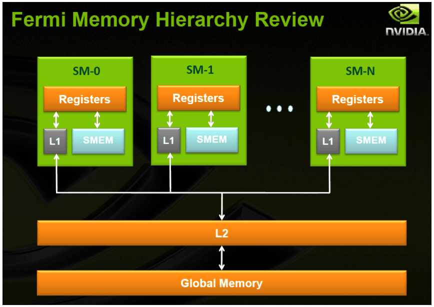
blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```
will map from local index threadIdx to global index

NB: blockDim should be bigger than 4 in real code, this is just an example

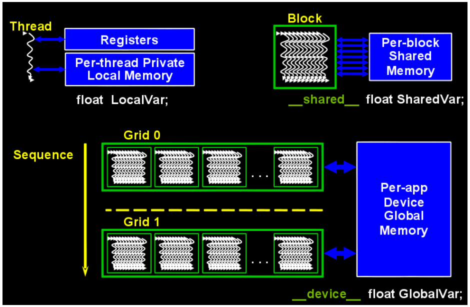
CUDA memory hierarchy: software/hardware

- **hardware (Fermi) memory hierarchy**
  - **on chip memory** : low latency, fine granularity, small amount
  - **off-chip memory** : high latency, coarse granularity (coalescence constraint, ...), large amount
  - shared memory: kind of cache, controlled by user, data reuse inside a thread block
  - need practice to understand how to optimise global memory bandwidth



CUDA memory hierarchy: software/hardware

- **software memory hierarchy**
  - **register** : for variables private to a thread
  - **shared** : for variables private to a thread block, public for all thread inside block
  - **global** : large input data buffer


Thread
Registers
Per-thread Private Local Memory
float LocalVar;
Block
Per-block Shared Memory
__shared__ float SharedVar;
Sequence
Grid 0
Grid 1
. . .
Per-app Device Global Memory
__device__ float GlobalVar;

Performance tuning thoughts

- **Threads are free**
  - Keep threads short and balanced
  - HW can (must) use LOTs of threads (10s thousands) to hide memory latency
  - HW launch ⇒ near zero overhead to create a thread
  - HW thread context switch ⇒ near zero overhead scheduling
- **Barriers are cheap**
  - single instruction: `_syncthreads();`
  - HW synchronization of thread blocks
- **Get data on GPU**, and let them there as long as possible
- **Expose parallelism**: give the GPU enough work to do
- **Focus an data reuse**: avoid memory bandwidth limitations

ref: M. Shebanon, NVIDIA

Other subjective thoughts

- tremendous rate of change in hardware from cuda 1.0 to 2.0 (Fermi) and coming 3.0 (Kepler)

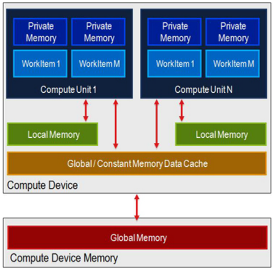
CUDA HW version	Features
1.0	basic CUDA execution model
1.3	double precision, improved memory accesses, atomics
2.0 (Fermi)	Caches (L1, L2), FMAD, 3D grids, ECC, P2P (unified address space), funtion pointers, recurs
3.5 (Kepler GK110) [1]	Dynamics parallelism, object linking, GPU Direct RemoteDMA, new instructions, read-only cache, Hyper-Q

- memory constraint like coalescence were very strong in cuda HW 1.0 ⇒ large perf drop in memory access pattern was not coaslescent
- **Obtaining functional CUDA code can be easy but optimisation might require good knowledge of hardware (just to fully understand profiling information)**

[1] as seen in slides CUDA 5 and Beyond from GTC2012

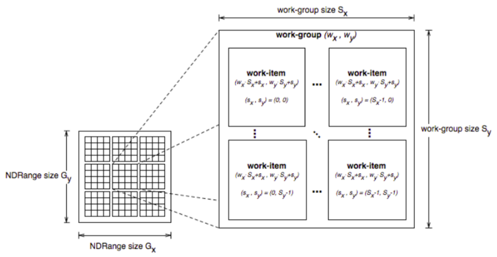
Summary

GPU computing - OpenCL



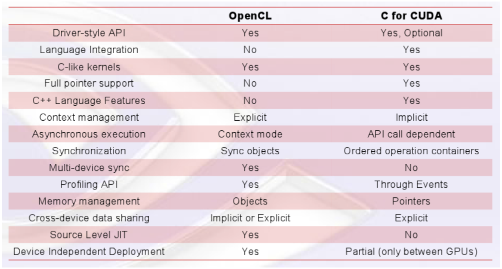
- OpenCL == Open **Computing Language**
- **standard** http://www.khronos.org, version 1.0 (12/2008)
- focus on **portability**: programming model for GPU (Nvidia/ATI), multicore CPU and other: *Data and task parallel compute model*
- **OpenCL programming model use most of the abstract concepts of CUDA**: grid of blocks of threads, memory hierarchy, ...

GPU computing - OpenCL


work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (S_x - 1, 0)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (0, S_y - 1)$
work-item
$(w_x \cdot S_x + s_x, w_y \cdot S_y + s_y)$
$(s_x, s_y) = (S_x - 1, S_y - 1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

- OpenCL language based on C99 with restrictions
- Some difficulties:
  - HW vendor must provide their OpenCL implementation; AMD is leading with OpenCL 1.2
  - multiple SDK with vendor specific addons ⇒ **breaks portability**
  - Both CUDA and OpenCL provide rather low-level API; but OpenCL's learning curve is steeper at first. Lots of low level code to handle different abstract concepts: plateform, device, command queue, ...; Need to probe hardware, ...

GPU computing - OpenCL



	OpenCL	C for CUDA
Driver-style API	Yes	Yes, Optional
Language Integration	No	Yes
C-like kernels	Yes	Yes
Full pointer support	No	Yes
C++ Language Features	No	Yes
Context management	Explicit	Implicit
Asynchronous execution	Context mode	API call dependent
Synchronization	Sync objects	Ordered operation containers
Multi-device sync	Yes	No
Profiling API	Yes	Through Events
Memory management	Objects	Pointers
Cross-device data sharing	Implicit or Explicit	Explicit
Source Level JIT	Yes	No
Device Independent Deployment	Yes	Partial (only between GPUs)

- OpenCL language based on C99 with restrictions
- Some difficulties:
  - **architecture-aware optimisation breaks portability:** NVIDIA and AMD/ATI hardware are different ⇒ require different optimisation strategy

GPU computing - OpenCL

- porting a CUDA program to OpenCL :
  http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-ATI-Stream-v2.0-Beta.aspx#four
- **tools to automate conversion CUDA/OpenCL:** SWAN, CU2CL (a CUDA to OpenCL source-to-source translator)
- **other tools:** MCUDA (a CUDA to OpenMP source-to-source translator)

GPU computing - OpenCL

Quick / partial comparison of CUDA / OpenCL; see
GPU software blog by Acceleyes

- **Performance**: both can fully utilize the hardware; but might be hardware dependant (across multiple CUDA HW version), algorithm dependent, etc ... Use benchmark SHOC to get an idea.
- **Portability**: CUDA is NVIDIA only (but new LLVM toolchain, also Ocelot provides a way from PTX to other backend targets like **x86-CPU** or **AMD-GPU**); OpenCL is an industry standard (run on AMD, NVIDIA, Intel CPU)
- **Community**: larger community for CUDA; HPC supercomputer with NVIDIA hardware; **Significantly larger number of existing applications in CUDA**
- **Third party libraries**: NVIDIA provides good starting points for FFT, BLAS or Sparse linear algebra which makes their toolkit appealing at first.
- **Other hardware**: some embedded device applications in OpenCL (e.g. Android OS); CUDA will probably be used for Tegra apps; CUDA toolkit for ARM (project MontBlanc)

CUDA - documentation and usefull links

- SuperComputing SC2011, slides: CUDA C BASICS, Performance Optimization (by P. Micikevicius)
- toolkit doc: PTX_ISA, CUDA_C_Programming_Guide, etc ...
- about coming new Kepler architecture: Kepler whitepaper