Introduction to GPU computing with CUDA

Pierre Kestener

CEA-Saclay, DSM, France
Maison de la Simulation

Formation Doctorants, June 22-24, 2016

Motivations

- The aims of this course
  - Motivation for GPU programing / computing
  - Acquire a pratical knowledge of CUDA programming
  - Overview of CUDA programing model, hands-on exercices
  - Introduction to GPU performance analysis tools / profiling / tracing
  - Short introduction to OpenACC / OpenMP for GPU with CLANG
  - Provide code walkthrough
  - Some practical items: coupling CUDA / MPI, using autotools or cmake build system, ...

Schedule: CUDA - GPU Programing - Saclay, June 2016

- **Wednesday, 22th**:
  - **Short historical view of GPU computing**, from graphics to HPC
  - **Main differences between CPU and GPU**
  - **CUDA software abstraction / programming model:** basics, development environment
  - **Hands-on / code experience:** SAXPY, Thrust, memory handling pitfalls
- **Thursday, 23rd**:
  - **long hands-on** about heat equation solver (learn more about gpu memory handling, 2D/3D, single/double, OpenGL/CUDA interop, ...).
  - **Advanced memory usage examples:** reduction, transpose, asynchronous memory transfert, ...
  - **Performance analysis tools:** `nvvp`, CUPTI/PAPI/TAU for GPU
  - **Overlapping computations / memory transfert:** use CUDA Streams,
- **Friday, 24th**:
  - **OpenACC:** use the PGI compiler,
  - **OpenMP for GPU:** use the CLANG compiler,
  - **CUDA/Fortran:** use the PGI compiler or wrap CUDA/C
  - **CUDA/Python:** use *cython* to wrap existing CUDA code,
  - **CUDA/MPI:**, GPUDirect, development with autotools,

CUDA training @ Saclay: Monday 7th, Dec 2015

- Architectures, parallelism and Moore's law
- A short historical overview
  - from vector processors to GPU
  - transfer graphics functionalities CPU/GPU - hardware/software
- Introduction to *GPU Computing*,
  - **hardware architecture**
  - **CUDA** programming model, workflow, execution model, CUDA language
  - **development tools** (compiler, debug, etc...)
  - ***parallel programming patterns*** : reduction, ...
- Web bibliography
  - many links to on-line resources (courses, source codes, articles, ...)

Ways to accelerate (scientific) applications

- Software developer's **tradeoffs**

Ways to Accelerate Applications

Applications

Libraries
e.g. FFTW -> cuFFT

"A la OpenMP"
Directives
OpenACC, HMPP, ...

Programming Languages (CUDA, ..)
OpenCL, ...

High Level Languages (Matlab, ..)

Easiest Approach
High Level approach, closest to application

Maximum Performance
High level languages, close to hardware

No Need for Programming Expertise

New ways: high-level libraries *à la TBB* (Thrust, kokkos, ...)
reference: Axel Koehler, NVIDIA

Ways to accelerate (scientific) applications

High

Development Productivity

Sequential GPLs and high-level DSLs

High-level parallel languages

Libraries, frameworks

Low-level languages or APIs, fully explicit parallelism control

Low

Scalability

High

reference: Boyana Norris (MCSD, Argonne National Lab),

State of programming models and code transformations on heterogeneous platforms

Supercomputers architectures - TOP500

A Supercomputer is designed to be at bleeding edge of current technology.
Leading technology paths (to exascale) using TOP500 ranks (Nov. 2014)

- **Multicore:** Maintain complex cores, and replicate (x86, SPARC, Power7) (#4)
- **Manycore/Embedded:** Use many simpler, low power cores from embedded (IBM BlueGene) (#3, 5, 8 and 9)
- **GPU/MIC/Accelerator:** Use highly specialized processors from gaming/graphics market space (Nvidia GPU, Intel XeonPhi (MIC)), (# 1, 2, 6, 7 and 10)

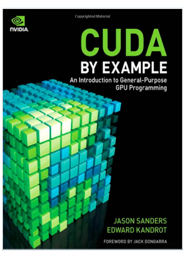
Nodes are complex, heterogenous, ...

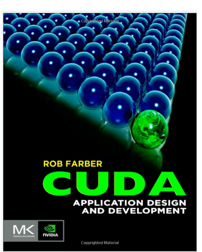
CUDA Recommended reading

- NVIDIA toolkit 7.5 documentation (pdf and html): cuda-c-programming-guide
- NVIDIA PTX (Parallel Thread Excution) documentation : parallel-thread-execution
- Udacity, on-line course cs344 https://www.udacity.com/course/cs344 (Introduction to Parallel Programming)

CUDA Recommended reading

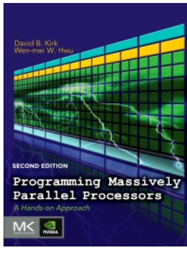
- book ***CUDA by example***, 2010, by J. Sanders and E. Kandrot from NVIDIA
- book ***CUDA Application Design and Development***, 2011, by R. Farber

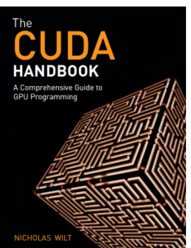




CUDA Recommended reading

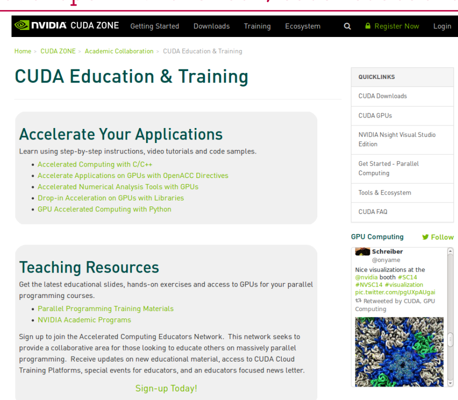
- NVIDIA toolkit 7.5 documentation (pdf and html): cuda-c-programming-guide
- book ***Programming Massively Parallel Processors: a hands-on approach***, 2nd ed., 2012, by D. Kirk and W.-M. Hwu
- book ***The CUDA Handbook***, 2013, by N. Wilt





CUDA On-line materials

Learning and teaching GPU programing:
https://developer.nvidia.com/cuda-education-training



CUDA On-line materials

- GPU Computing webinars: http://www.gputechconf.com/resources/gtc-express-webinar-program
- GPU Technology Conference resources : select for example *GTC* event for year *2012*, you'll get material about:
  - Algorithms and numerical techniques
  - Astrophysics, Image processing, Computer Vision, Bioinformatics, Climate, Cloud, CFD, Data Mining
  - Development tools, libraries
- http://www.moderngpu.com/: a very insightful presentation
- List / comparison of Nvidia GPU: http://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

Summary

1. GPU computing: Architectures, Parallelism and Moore law
   - Why multi-core ?
   - Understanding hardware, better at optimization
   - What's a thread ?
   - History
2. CUDA Programing model
   - Why shoud we use GPU ?
   - Hardware architecture / programming model
   - CUDA : optimisation / perf measurement / analysis tools
   - GPU computing : perspectives / Install CUDA
3. Other languages: CUDA/Fortran, PyCuda, CUDA/MPI
4. Extra slides

Parallel computing - Rendering - GPU

- before the 1990's, **parallel computers** were **rare** and available for only the most critical problems
- **Toy Story (1995)** : **first completely computer-generated feature-length film**, processed on a "renderfarm" consisting of 117 Sun(™) SPARCstation(™) @100MHz workstations. Computing the 114000 frames (77 minutes) required 800000 computer hours. Each frame consists in 300 MBytes of data (one hard-disk in 1995).
- Computer animation (and rendering, i.e. the process of generating an image from a model or scene description) is where **parallel computing / HPC** and **graphics computing / GPU** meets.
- **Software for off-line rendering** : e.g. RenderMan (`http://renderman.pixar.com`) by Pixar, from modelling to rendering
- **Hardware rendering** : OpenGL low-level API used in real-time rendering (i.e. done in dedicated hardware like **GPU**), gaming industry

Parallel computing - Rendering - GPU

State of the art in computer generated feature-length movie: Big Hero 6

http://www.tomshardware.fr/articles/big-hero-6-technologie,1-54655.html

- 55000 CPU-core
- 180 days ⇒ **200 Millions computing hours**
- **It's roughly half of CURIE resources during 6 months !! (CURIE is one the most powerfull supercomputer in Europe.)**

Actualités & Articles | Comparatifs | Marques | Forum | Autres sites

tom's HARDWARE

THÈMES : Processeurs | Cartes graphiques | SSD | Android | Consoles | Entreprise

Tom's Hardware > Serveurs > Actualité Serveurs

55 000 coeurs pour illuminer Big Hero 6 de Disney

Par Matthieu Lamelot 25 OCTOBRE 2014 09:06 - Source: Engadget | 0 COMMENTAIRE

THÈMES : Serveurs + Film + Disney + 3d +

La technologie de Big Hero 6

Lorsqu'on voit un quidam employer la merveille de technologie qu'est son smartphone pour jouer au solitaire, on peut se demander à quoi toute l'informatique rime. Heureusement, il y a des usages nettement plus cohérents avec la puissance disponible

Parallel computing / Parallel programming

- **Parallel computing** relies on exploitable **concurrency**
- *Concurrency exists in a computational problem when the problem can be decomposed into subproblems that can safely execute at the same time,*
  Mattson et al,
  in `Patters for parallel programming`
- concurrency: property of a system in which several computations are executing simultaneously
- **How to structure code to expose and exploit concurrency ?**

Moore's law - *the free lunch is over...*

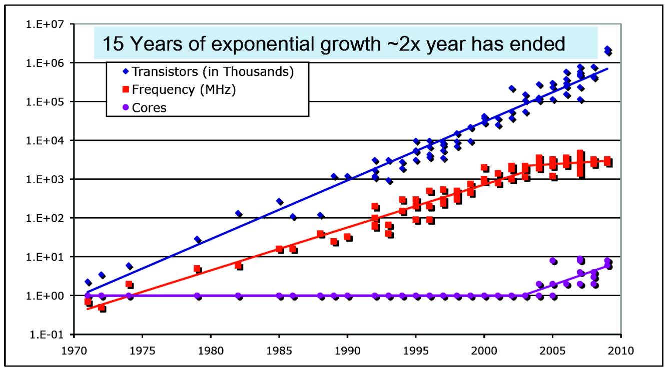
The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years

1.E+07
1.E+06
1.E+05
1.E+04
1.E+03
1.E+02
1.E+01
1.E+00
1.E-01
1970 1975 1980 1985 1990 1995 2000 2005 2010
Transistors (in Thousands)
Mercredi 5 Janvier 2011

Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç

Moore's law - *the free lunch is over...*

The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years



Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç

Moore's law - *the free lunch is over...*

Moore's Law continues with

- **technology scaling** (32 nm in 2010, 22 nm in 2011),
- improving transistor performance to increase frequency,
- increasing transistor integration capacity to realize complex architectures,
- reducing energy consumed per logic operation to keep power dissipation within limit.

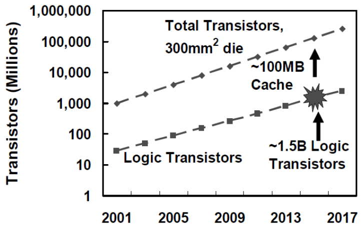


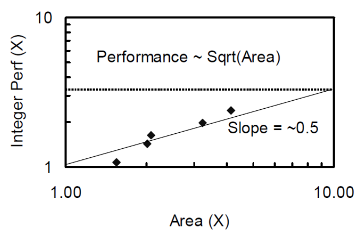
Figure 1: Transistor integration capacity

Shekhar Borkar, *Thousand Core Chips - A Technology Perspective*, in Intel Corp, Microprocessor Technology Lab, 2007, p. 1-4

Moore's law - Towards multi-core architectures

Pollack's rule - Wide adoption of multi-core architectures

- if you **double the logic** in a processor core, then it delivers **only 40% more performance**
- A multi-core microarchitecture has potential to provide near linear performance improvement with complexity and power.
- For example, **two smaller processor cores, instead of a large monolithic processor core, can potentially provide 70-80% more performance, as compared to only 40% from a large monolithic core**



Shekhar Borkar, *Thousand Core Chips - A Technology Perspective*, in Intel Corp, Microprocessor

Moore's law - Towards multi-core architectures

(Heterogeneous) **Multi-core processors have several other benefits**:

- each processor core could be individually turned on or off, thereby **saving power**;
- each processor core can be run at its own optimized supply voltage and frequency;
- easier to load balance among processor cores to distribute heat across the die;
- can potentially produce lower die temperatures improving reliability and leakage.

Moore's law - Towards multi-core architectures

- More transistors $\nleftrightarrow$ more computing power !
- **More transistors ? What's the purpose ?** How to use them **efficiently** ?
- **Improve single-core CPU performances:**
  - ☹ keep frequency increasing (watch electric power !)
  - ☺ keep transistor density increasing (more and more difficult) : 32 nm in 2010
- **Utilize efficiently transistors on chip**
  - ☹ instruction-level parallelism (out-of-order execution, etc...)
  - ☺ data-level parallelism (SIMD, vector units) : SSE, Cell Spe, GPU !
  - ☺ thread-level parallelism: hardware-*multi-threading*, multi-core, many-core ...

http://www.ugrad.cs.ubc.ca/~cs448b/2010-1/lecture/2010-09-09-ugrad.pdf

Moore's law - Towards multi-core architectures

- More transistors $\not\leftrightarrow$ more computing power !
- **More transistors ? What's the purpose ?** How to use them **efficiently** ?
- **re-think or re-design** algorithms to exploit **parallelism** (*multithreads*, multicores, ...) and make them **scalable** (whatever the number of cores) !
- **modern GPU have massively *multi-threads* architectures** (up to 48 active *threads* per core in Fermi)

http://www.ugrad.cs.ubc.ca/~cs448b/2010-1/lecture/2010-09-09-ugrad.pdf

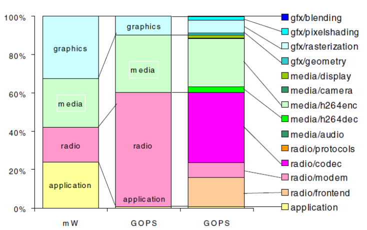
Multi-core - a technology perspective

- not only in supercomputers
- almost every PC in 2010 has a multi-core CPU
- multiple small cores at a lower clock frequency are more power efficient
  corollary : the parallel version of a code can be much more energy-efficient than its sequential version running at the same frequency
- Even smartphones processors become multi-core: multiple small cores at a lower frequency are more efficient than single core (increase battery life-time, ...)
- Modern smartphones perform nearly 100GOPS within a power budget of only 1W !

http://www.date-conference.com/proceedings/PAPERS/2009/DATE09/PDFFILES/

S. Borkar and A. Chien, The Future of Microprocessors, ACM Communications

Multi-core parallelism in smartphones

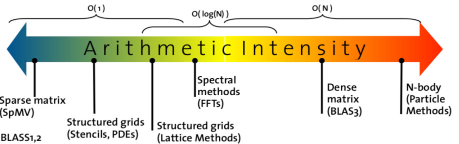


- maximum workload for a 3G smartphone is about **100 GOPS**
- **Application processing** : user interface, address books, diaries, sms, java, internet browsing, email, gaming, document and spreadsheet editing, photo handling
- **Radio processing** : demodulation, decoding, protocol, ...
- **Multi-media processing**
- The challenge is to provide 100GOPS within a 1W power budget
- The solution has to be **multicore**, as **multiple small cores at a lower clock frequency are more power efficient**

http://www.date-conference.com/proceedings/PAPERS/2009/DATE09/PDFFILES/

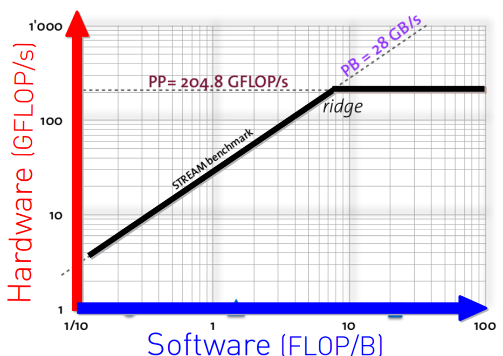
Roofline model

- an increasingly large diversity of architectures
- software challenges to use new architectures:
  - refactoring (when ? where ?); avoid sub-optimal use of hardware/software
  - optimization strategies
- **Roofline model:** a simple way of characterizing hardware performances
  Each algorihm implementation is characterized by
  - **arithmetic intensity (FLOPS/Bytes):** number of FLOP per bytes read/write from external memory
  - **effective memory bandwidth (GB/s):** data moved to/from external RAM



reference:

Roofline model



- it visually relates hardware with software
- Performance = min (Peak Bandwidth * Arith Intensity, Peak Flops)

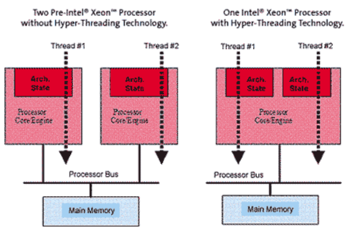
reference:

http://www.nvidiacodesignlab.ethz.ch/news/CoDesignLabWorkshop2013_Rossinelli_Roofline.pdf

What is a *thread* anyway ?

- ***Thread*** **- hardware / software definition(s)**:
  - execution unit - fil d'exécution
  - instruction unit - fil d'instruction
  - processing unit - unité de traitement
  - lightweight process - processus léger (jargon Unix) : in software, switching between threads does not involve changing the memory context.
- confusion: ***multi-threading ≠ multi-cores***
- **multi-threading on a single core** = **time-division multiplexing** i.e. the processor switches between different threads. Multiple thread are not actually running simultaneously, just interleaved execution
- **SMT (Simultaneous Multi-Threading)** : a single core able to process multiple instructions from multiple threads in one cycle. This is called Hyper-Threading by Intel. The two (or more) harware thread share acces to cache memory or virtual memory (TLB) leading potentially to contention.
- **CMT (Chip-level Multi-Threading)** : integrates two or more processors into one chip, each executing threads independently; also called multi-processing

What is a *thread* anyway ?



- **SMT (Simultaneous Multi-Threading)** : One physical processor for multipe logical processors.
  - Each logical processor maintains a complete set of the architecture state (general- purpose registers, control registers, ...)
  - Logical processors share nearly all other resources, such as caches, execution units, branch predictors, control logic, and buses

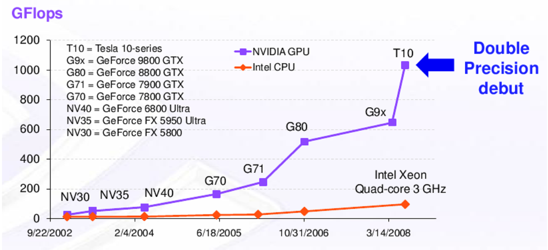
  Costs of a 2-way Hyper-Threading : chip area (+5%), performance (+15 to 30%)
- **CMT (Chip-level Multi-Threading)** :

What is a *thread* anyway ?

- (academic ?) Implicit Multi-thread processor : Dynamically generate threads from single-threaded programs and execute such speculative threads concurrent with the lead thread. Multiscalar, dynamic multithreading, speculative multithreaded, ...
- **Wikipedia [Multi-threading]:**
  (...) multithreading aims to increase utilization of a **single core** by leveraging **thread-level** as well as **instruction-level parallelism**.
- *multi-threading* aim: improve instruction throughput of the processor.
- example : hide memory access latency by thread-context switching (**key feature in modern GPU high performances**)

From multi-core CPU to manycore GPU

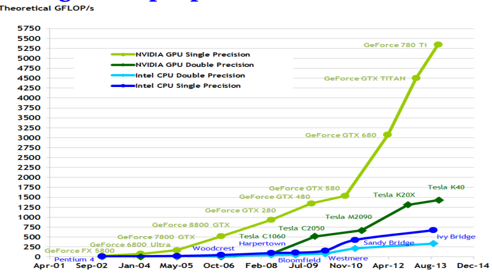
Why is there a large performance gap between manycore GPUs and general purpose multicore CPU ?



- **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

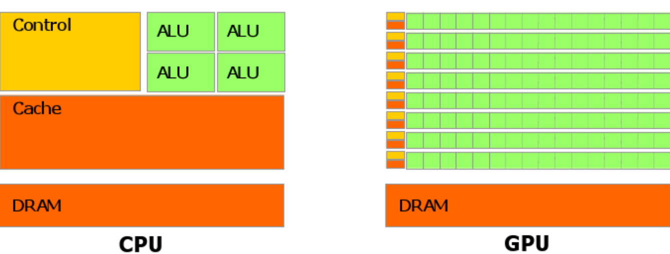
Why is there a large performance gap between manycore GPUs and general purpose multicore CPU ?



- **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

Why is there a large performance gap between manycore GPUs and general purpose multicore CPU ?



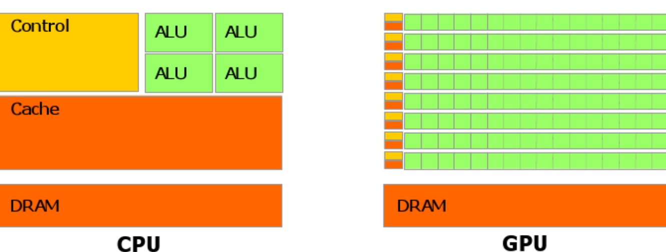
- **GPU** design goal : maximize throughput of **all threads**
  - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency => skip the big caches
  - share control logic across many threads

ref: Jared Hoberock, Stanford, cs193,

http://code.google.com/p/stanford-cs193g-sp2010/

From multi-core CPU to manycore GPU

Why is there a large performance gap between manycore GPUs and general purpose multicore CPU ?

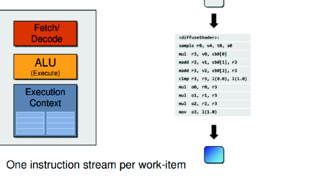


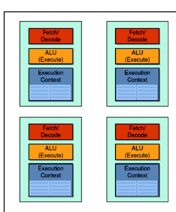
- fast growing game industry made a tremendous economic pressure to design architectures optimized for maximum chip area / power budget per floating point operations.
- GPU takes advantage of a **large number of execution threads** to find work to do when other threads are waiting for long-latency memory accesses, thus **minimizing the control logic** required for each execution thread.

From multi-core CPU to manycore GPU

Why is there a large performance gap between manycore GPUs and general purpose multicore CPU ?

- **CPU Conventional Core** **Quad**





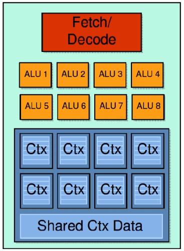
- GPU: much more area dedicated to floating point computations
- GPUs are numeric computing engines that will not perform well on some tasks for which CPU are optimized. Need to take advantage of both !

reference: D. Kirk and W.W. Hwu, *Programming massively parallel processors*, Morgan and Kaufmann eds.

J. Dongarra, CEA-EDF-INRIA summer school, 2011

From multi-core CPU to manycore GPU

Why is there a large performance gap between manycore GPUs and general purpose multicore CPU ?



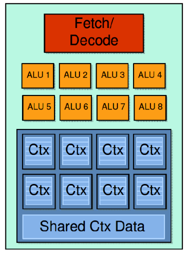
- SIMD
- GPU: Amortize cost / complexity of managing an instruction stream across many ALUs.

reference: D. Kirk and W.W. Hwu, *Programming massively parallel processors*, Morgan and Kaufmann eds.

J. Dongarra, CEA-EDF-INRIA summer school, 2011

From multi-core CPU to manycore GPU

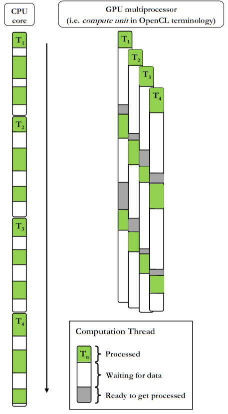
Why is there a large performance gap between manycore GPUs and general purpose multicore CPU ?



- CPU: task parallelism, threads managed explicitly
- GPU: data parallelism, threads managed/scheduled by hardware: (**context switch almost free**), threads are grouped into *warps*), need lots of warps to hide memory latecy by computation

Other practical issues to explain GPU rise

Hiding memory latency



reference: D. Kirk and W.W. Hwu, *Programming massively parallel processors*, Morgan and Kaufmann eds.

Other practical issues to explain GPU rise

- **cost of software development is best justified by a very large customer population:**
  - Traditional parallel computing system used to have negligible market (e.g. CRAY vector processor in the 80s).
  - large market (game industry) made GPU economically attractive
- Ease of accessibility of parallel computing systems
  - before 2006, parallel software ran only on data-center / clusters
  - **According to Kirk and Hwu**, NIH refused to fund parallel programing projects for some time because they felt the impact of parallel software would be limited (no huge cluster-based machines in clinical settings). Today, the possibility to have small GPU-based equipment re-enable parallel software research for such medical application.

reference: D. Kirk and W.W. Hwu, *Programming massively parallel processors*, Morgan and Kaufmann eds.

Other practical issues to explain GPU rise

- before 2006, OpenGL® / Direct3D® API were the only way to program GPUs - this is called legacy GPGPU (General Purpose GPU) computing and required highly specific programming skills
  see the GPGPU tutorial
  http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html
- everything changed in 2007, with the release of CUDA (new hardware, new programming model, etc...)

reference: D. Kirk and W.W. Hwu, *Programming massively parallel processors*, Morgan and Kaufmann eds.

Fundamentals of Parallel Computing

- **Parallel computing requires that**
  - The problem can be decomposed into sub-problems that can be safely solved at the same time
  - The programmer structures the code and data to solve these sub-problems concurrently
- **The goals of parallel computing are**
  - To solve problems in less time, and/or
  - To solve bigger problems, and/or
  - To achieve better solutions

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.
reference :
http://courses.engr.illinois.edu/ece498/al/Syllabus.html

Parallel programming / computational thinking

Ideal parallel programmer skills:

- **Computer architecture:**
  - Memory organization,
  - caching and locality,
  - memory bandwidth,
  - SIMT (single instruction multiple thread) versus SPMD (single program multiple-data versus SIMD,
  - floating point accuracy
- **Programming models and compilers:**
  - parallel execution models,
  - types of available memories,
  - array data layout,
  - loop transformation
- **Algorithm techniques:**
  - tiling, cutoff, binning
- **Domain knowledge:**
  - numerical methods, models accuracy required

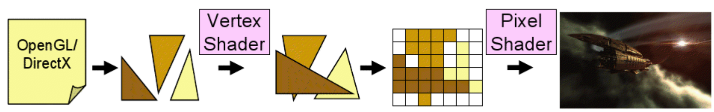
reference: Kirk, Hwu, *Programming massivelly parallel processors*, chapter 10

From vector processors to GPU

http://www.irisa.fr/archi09/defour.pdf

- Vector processor : implements an instruction set containing instructions that operate on 1D arrays of data called vectors
- most super-computer in the 80s / 90s were vector processor
- CRAY-1: 1976, 80MHz, 64-bit/data, 24-bit/adress, vector register file, 160 MIPS, 250 MFLOPS, 8MB RAM, 5.5 tonnes, ~ 200-kW (cooling included)
- modern/current scalar processors have vector instructions set (SIMD) : SSE, Altivec (using 128 bits register).
- **vector processors fall (end of 80s - beginning of 90s) :** CMOS technology; rising of PC mass market; single chip microprocessor vs difficult to fit a vector processor on a single chip; cache size increase in scalar processors; programming vector procesor required assembly language skills.

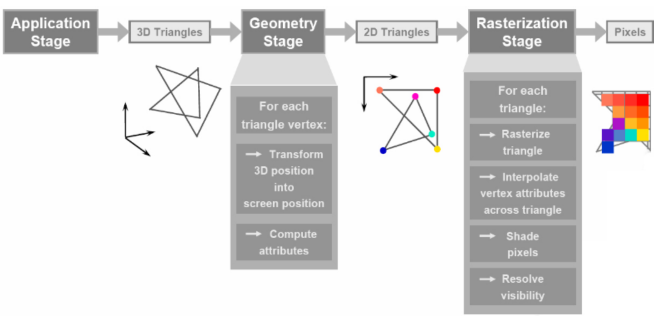
GPU evolution: architecture and programming



- basics of **graphics *pipeline***
- basics of ***shaders*** (programmable functionalities in the graphics *pipeline*)
- Overview of GPU architectures
- **Legacy GPGPU** (before CUDA, ~ 2004)

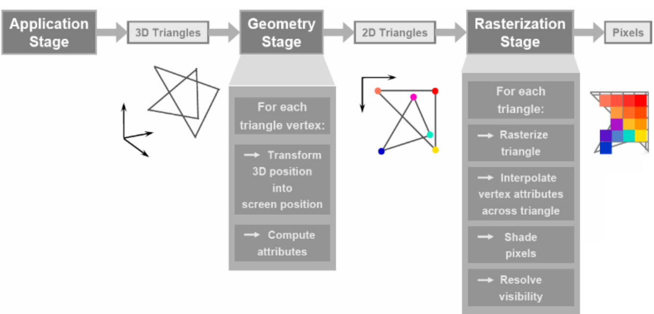
The Evolution of GPUs for General Purpose Computing,
by Ian Buck (CUDA nventor)
http://www.nvidia.com/content/GTC-2010/pdfs/2275_GTC2010.pdf See also
ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

Overview of GPU architectures: from 1995 to 1999



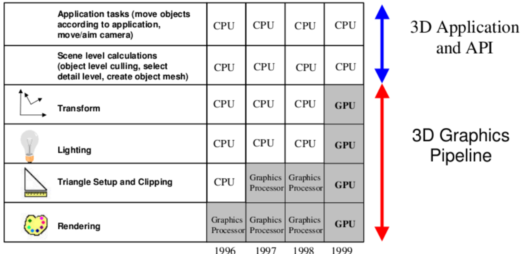
- *graphics pipeline*: conceptual model of stages that graphics data is sent through (**software** or **hardware**)
- GPU main function : **off-load graphics task from CPU to GPU**
- GPU: dedicated hardware for specialized tasks : **GPU hardware designed to optimize highly repetitive tasks:** e.g. texturing (bi-, tri-, quadri-linear filtering), rastering (pixel interpolation), z-cull (remove hidden surface), etc...

Overview of GPU architectures: from 1995 to 1999



- *graphics pipeline*: conceptual model of stages that graphics data is sent through (**software** or **hardware**)
- from 1995 : 3D rendering, rasterization (vector image into pixel image), more complex rendering

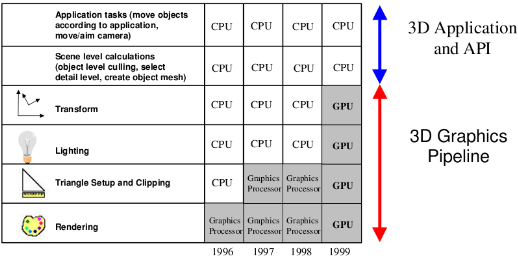
A short historical overview : 1995 to 1999



http://www.nvidia.com/object/Technical_Brief_TandL.html
http://www.cs.unr.edu/~fredh/papers/thesis/023-crow/GPUFinal.pdf

- design specification of graphics API (API should be cross-vendor, cross-paltform, provide hardware abstraction) : OpenGL, DirectX
- 3D graphics boom :
  - Motion picture industry : *Toy Story I* (1995, Pixar) is the first full-length computer-generated feature film (*off-line* rendering with RenderMan)
  - mass-market game industry: Quake, Doom, etc...; fligh simulators

A short historical overview : 1995 to 1999



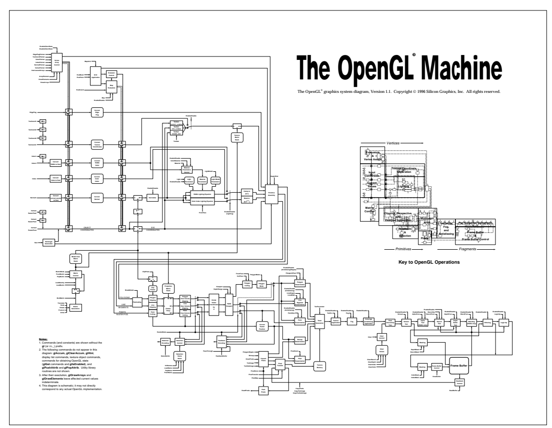
	1996	1997	1998	1999
Application tasks (move objects according to application, move/aim camera)	CPU	CPU	CPU	CPU
Scene level calculations (object level culling, select detail level, create object mesh)	CPU	CPU	CPU	CPU
Transform	CPU	CPU	CPU	GPU
Lighting	CPU	CPU	CPU	GPU
Triangle Setup and Clipping	CPU	Graphics Processor	Graphics Processor	GPU
Rendering	Graphics Processor	Graphics Processor	Graphics Processor	GPU

http://www.nvidia.com/object/Technical_Brief_TandL.html
http://www.cs.unr.edu/~fredh/papers/thesis/023-crow/GPUFinal.pdf

- 1999 : transfert transformation operations to GPU (4x4 matrix multiplication) and lightning
- 1999 : Nvidia introduces the first consumer-level GPU with the entire graphics pipeline in hardware.
- **fixed function pipeline** (developper limited to a fixed set of features)

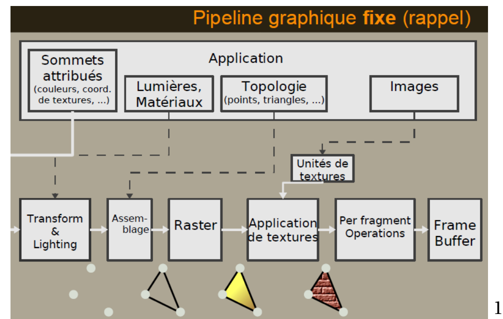
OpenGL StateMachine

http://www.opengl.org/documentation/specs/version1.1/state.pdf



A short historical overview : 2000 and after

- **"All processors aspire to be general-purpose."**
  – Tim Van Hook, Graphics Hardware 2001
- 2001 : Nvidia GeForce3
- **programmable graphics pipeline** : *pixel* and *vertex-shaders* writen in **low-level language**, pros : high flexibility and allow hardware to follow rapid evolution of graphics standards, *ease* of development

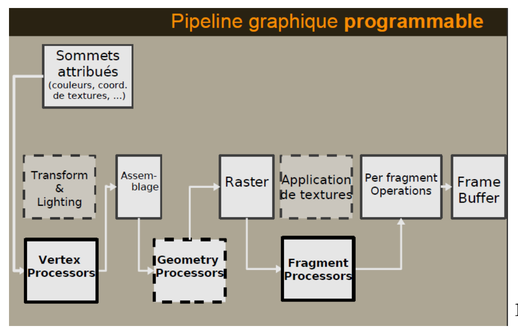


[1]

[1] www.irit.fr/~Vincent.Forest/teaching/SeminaireGPU.pdf

A short historical overview : 2000 and after

- **"All processors aspire to be general-purpose."**
  – Tim Van Hook, Graphics Hardware 2001
- 2001 : Nvidia GeForce3
- **programmable graphics pipeline** : *pixel* and *vertex-shaders* writen in **low-level language**, pros : high flexibility and allow hardware to follow rapid evolution of graphics standards, *ease* of development



[1]

[1] www.irit.fr/~Vincent.Forest/teaching/SeminaireGPU.pdf

A short historical overview : 2000 and after

- **"All processors aspire to be general-purpose."**
  – Tim Van Hook, Graphics Hardware 2001
- 2001 : Nvidia GeForce3
- **programmable graphics pipeline** : *pixel* and *vertex-shaders* writen in **low-level language**, pros : high flexibility and allow hardware to follow rapid evolution of graphics standards, *ease* of development
- GPU : *Vertex Processors* (MIMD), *Fragment Processors* (SIMD), 32 bits float
- introduce **high-level languages** : Cg (Nvidia, compatibility with OpenGL/DirectX), HLSL (Microsoft, compatibility with DirectX only)
- hardware abstraction, computer programmer only needs to know little about hardware

A short historical overview : 2000 and after

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://perso.ens-lyon.fr/sylvain.collange/talks/raim11_scollange.pdf

A short historical overview : 2000 and after

Floating point computations capability implemented in GPU hardware

- **Why are my floating results different on GPU from CPU ?**
  - cannot expect always the same results from different hardware
  - **one algorithm**, but 2 different software implementations for 2 different hardware CPU and GPU
  - **Hardware differences:**
    - CPU floating point unit (FPU) might use x87 instructions (with 80 bits precision used internally); SSE operations use 32 or 64 bits values.
    - Compiler dependency: values kept in register ? or spilled to external memory ?
  - **Fused Multiply-Add from IEEE754-2008**
    - $a * b + c$ in a single rounding step; better precision
    - implemented on all GPU from CUDA hardware ≥ 2.0 but not all CPU (AMD/Bulldozer/2011, Intel/Haswell/2013,...)
    - use `nvcc` flags `-fmad=false` to tell compiler not to use FMAD instructions
    - see code snipnet in hands-on
- CUDA doc : Floating_Point_on_NVIDIA_GPU_White_Paper.pdf

A short historical overview : 2000 and after

Floating point computations capability implemented in GPU hardware

- **Why are my floating results different on GPU from CPU ?**
  - **parallel computations rearrange operations**; associativity $((a+b)+c \neq a+(b+c)) \Rightarrow$ different the results; see reduction example
  - **Different ? By how much ?** Can be very hard to distinguish differences coming from parallel computation from a genuine bug in the algorithm. When comparing a reference CPU algorithm with the GPU ported algorithm, **observing differences do not necessarily imply there is bug in your GPU code !!**
  - Take care that single precision uses 23 bits for mantissa ($2^{-23} \sim 10^{-7}$) and double precision 52 bits ($2^{-52} \sim 10^{-16}$): **CPU/GPU differences will accumulate much faster in single precision**
  - **Allways a good idea to use double precision for test/debug**, even if target precision is single precision (e.g use a `typedef real_t`)
- CUDA doc : Floating_Point_on_NVIDIA_GPU_White_Paper.pdf

From legacy GPGPU to CUDA : unified architecture (2007)

Nvidia GeForce 7800

Host / FW / VTF
Cull / Clip / Setup
Z-Cull
Shader Instruction Dispatch
L2 Tex
Fragment Crossbar
Memory Partition
Memory Partition
Memory Partition
Memory Partition
DRAM(s)
DRAM(s)
DRAM(s)
DRAM(s)

From legacy GPGPU to CUDA : unified architecture (2007)

Nvidia GeForce 8000

Legacy GPU computing - before CUDA

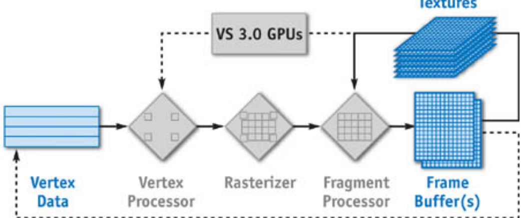
Legacy GPGPU :
General Purpose computations on GPU

- Legacy GPGPU: twist Graphics APIs to perform general purpose computing task
- GPU were designed for computer graphics (output streams of colored pixels from input streams of vertices, texels)
- There is a need for translating computational concepts into GPU programming model
- GPGPU dedicated tutorial for OpenGL (2005)
  http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html
  have also a look at
  chapiter 31 in GPUGems2, *Mapping Computational concepts to GPUs*,
  http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter31.html

Legacy GPU computing - before CUDA

Legacy GPGPU :
General Purpose computations on GPU

- CPU - GPU analogies
- GPGPU concept 1 : arrays = texture
  - CPU memory can be read / written anywhere in a program
  - Vextex programs are not allowed to randomly index into input vertices.
  - GPU texture (memory buffer) are read-only but random acces allowed !!
    These are output of vextex/fragment processor.
  - render-to-texture (instead of display), very recent feature (~ 2003)



Legacy GPU computing - before CUDA

Legacy GPGPU :
General Purpose computations on GPU

- CPU - GPU analogies
- GPGPU concept 1 : arrays = texture
  - create a frame buffer object for off-screen rendering; random access not allowed
  - memory : CPU array → GPU GL_TEXTURE_2D / (read-only or write-only), bind a texture to a FBO
  - CPU array indexes (integer) → GPU texture coordinates (float $\in [0,1]$)
  - data type : float → GL_LUMINANCE or GL_RGBA
  - array size : power of 2 (?)
  - scatter operations ($a[i] = x$) need to be converted into gather ($x = a[i]$)

Legacy GPU computing - before CUDA

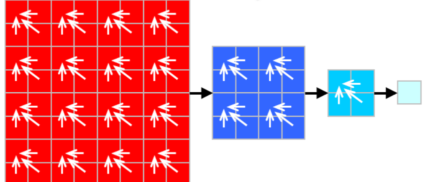
Legacy GPGPU :
General Purpose computations on GPU

- GPGPU concept 2 : CPU programs, loops = GPU fragment shaders (a fragment code on a stream of vertices)
- GPGPU concept 3 : CPU computing = GPU drawing : The vertex processors transform the geometry and the rasterizer determine which pixels in the output buffer it cover and generate a fragment for each one.

Legacy GPU computing - before CUDA

Legacy GPGPU : General Purpose computations on GPU

- Reduction example : compute Max of items in a 2D array



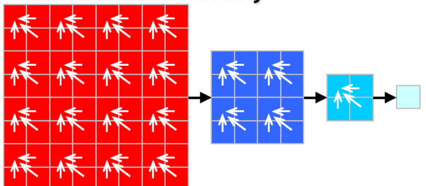
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter31.html

- Implementation: use a pair of buffers and iterate a 2-steps operation of rendering to / reading from buffer by reducing the output size by 2 until we have only a single element containing the max. We need $O(log(n))$ passes.
- other example code : sum of arrays (hands-on; see sub dir `gpgpu_legacy/goeddeke_gpgpu`) saxpy_cg.cpp

Legacy GPU computing - before CUDA

Legacy GPGPU :
General Purpose computations on GPU

- Reduction example : compute Max of items in a 2D array

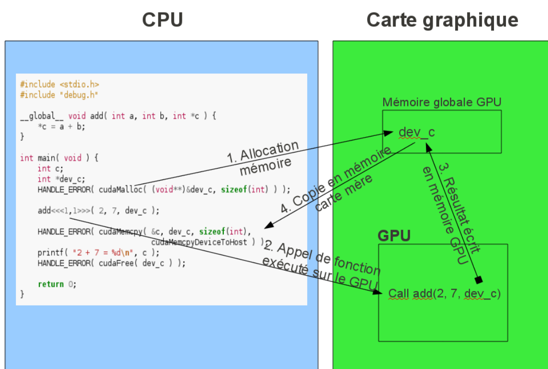


- Algorithm and data layout are **tightly coupled** !

GPU computing - CUDA

- 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
- new hardware architecture generation: CUDA (**Compute Unified Device Architecture**)
- new programming model: CUDA + development tools (toolkit, compiler, SDK, librairies like cuFFT), a *C-like* programming language

CUDA - langage C-like - HelloWorld



CUDA - langage C-like - HelloWorld

Host

```
void *gpuAllocation;
cudaMalloc( &gpuAllocation, bytes );

cudaMemcpy( gpuAllocation, source, bytes,
   cudaMemcpyHostToDevice);

kernel<<< gridDim, ctaDim >>>(
  gpuAllocation,
  otherParameters
);

cudaMemcpy( result, gpuAllocation, bytes,
   cudaMemcpyDeviceToHost);
```

GPU

data transfer to GPU

GPU Kernel Launch

Grid of cooperative thread arrays

Cooperative Thread Array

barrier

divergent branch

barrier

data transfer to Host

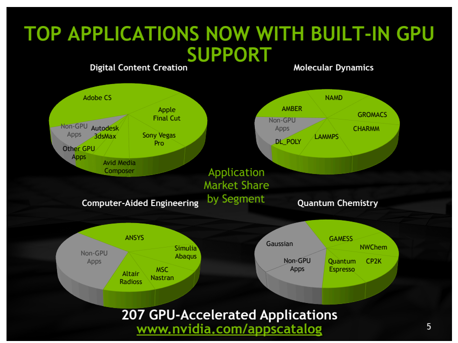
GPU computing - CUDA

Applications examples:

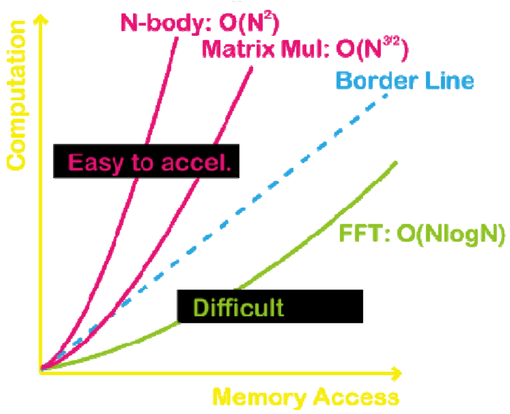
- meteo : Numerical weather prediction, http://www.mmm.ucar.edu/wrf/WG2/GPU/
- CFD: http://www.nvidia.com/object/computational_fluid_dynamics.html
- Molecular Dynamics / Quantum chemistry: http://www.nvidia.com/object/molecular_dynamics.html
- Bioinformatics http://www.nvidia.com/object/tesla_bio_workbench.html
- financial computing: http://people.maths.ox.ac.uk/gilesm/hpc/slides.html
- Signal/Image processing, Computer vision, etc
- too many by now.. see CUDA ZONE
- Books: GPU Computing GEMS, Emerald Edition,

GPU computing - CUDA

Conference GTC (GPU Technology Conference, since 2010) : several hundred's applications examples
http://on-demand-gtc.gputechconf.com/gtcnew/on-demand-gtc.php
Have a look at the CUDA *show case*:
http://www.nvidia.com/object/cuda_apps_flash_new.html



What kinds of computation map well to GPUs ?



Nukada *et al.*, SC08

http://www.nvidia.com/content/GTC-2010/pdfs/2084_GTC2010.pdf

What kinds of computation map well to GPUs ?

Algorithm Examples

Performance Per mm²

CPU

GPUs

Algorithm Regularity

Dense Linear Algebra

Image/video processing

Sort, Computational geometry, finance

- Sort, computational geometry, finance
  - Modest control flow
  - ...rse/Irregular data structures
  - Irre... ...ar communication between elements
- CP... ...tory
  - Ge... ...l purpose features vital for software efficiency
  - ...ency sensitive applications

All dates, figures and product plans are preliminary and are subject to change without notice. Copyright © Intel Corporation 2006

Nukada *et al.*, SC08

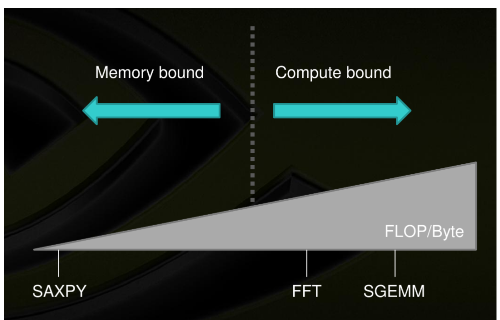
http://www.nvidia.com/content/GTC-2010/pdfs/2084_GTC2010.pdf

What kinds of computation map well to GPUs ?

1030 GFLOP/s
144 GB/s
GPU
DRAM
Tesla C2050

- **System FLOP/Byte** is quite high for GPU, significantly higher than multi-core CPU
- Actual FLOP/Byte is of course **algorithm depend**

What kinds of computation map well to GPUs ?



- **Memory-bound algorithm:** need special care about external memory handling (coalescence, SAO-SOA, etc...), see CUDA SDA reduction example
- **Compute-bound algorithm:** ideal for GPU data-parallel architecture

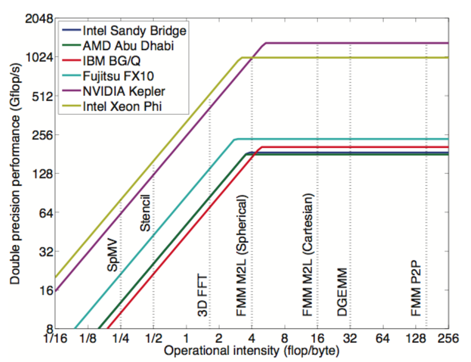
What kinds of computation map well to GPUs ?

Kernel	FLOP/Byte**
Vector Addition	1 : 12
SAXPY	2 : 12
Ternary Transformation	5 : 20
Sum	1 : 4
Max Index	1 : 12

** excludes indexing overhead

Kernel	FLOP/Byte
GeForce GTX 280	~7.0 : 1
GeForce GTX 480	~7.6 : 1
Tesla C870	~6.7 : 1
Tesla C1060	~9.1 : 1
Tesla C2050	~7.1 : 1

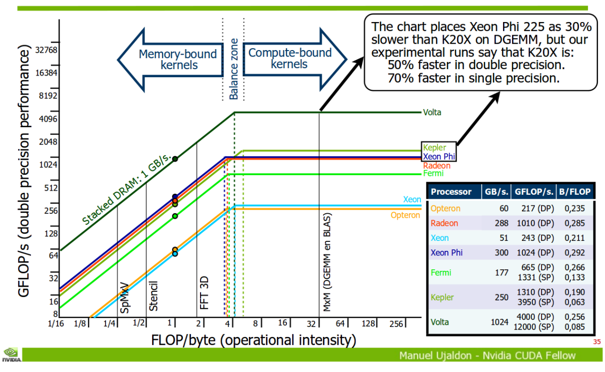
About roofline model of current (future ?) hardware



- Understand inherent hardware limitations
- Show priority of optimization

references:
http://lorenabarba.com/news/fast-multipole-method-in-the-exascale-era/

About roofline model of current (future ?) hardware



Processor	GB/s.	GFLOP/s.	B/FLOP
Opteron	60	217 (DP)	0,235
Radeon	288	1010 (DP)	0,285
Xeon	51	243 (DP)	0,211
Xeon Phi	300	1024 (DP)	0,292
Fermi	177	665 (DP) 1331 (SP)	0,266 0,133
Kepler	250	1310 (DP) 3950 (SP)	0,190 0,063
Volta	1024	4000 (DP) 12000 (SP)	0,256 0,085

- Understand inherent hardware limitations
- Show priority of optimization

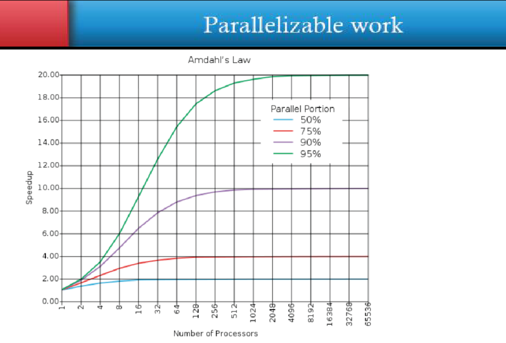
references:
http://lorenabarba.com/news/fast-multipole-method-in-the-exascale-era/

http://icpp2013.ens-lyon.fr/GPUs-ICPP.pdf

Is parallelisation worth it ?

$$\textbf{Amdahl's law: } R = \frac{1}{(1-p)+p/N}$$

- p: fraction of work that can be parallelized
- $1-p$: fraction of sequential work
- R: prediction maximum speed-up using N parallel processors



embedded systems...

- Embedded systems and HPC both run after lower power and better performance
- OpenGL ES (Open Graphics Library for Embedded System)
  - OpenGL ES 1.1.X (2004-2008): *fixed function hardware*, chosen as 3D API 3D in Symbian , Android, iPhone SDK, ...
  - OpenGL ES 2.0.X (2007-...): *programmable hardware*, specification of a high-level *shading language*, used in new iPod Touch, Nokia N900, ...
  - OpenGL ES 3.0.X (2012-...): fill the perf and feature gap between embedded graphics mobile platform and desktop/game console; 1 billion GPU by 2014

embedded systems...

- Embedded systems and HPC both run after lower power and better performance
- OpenGL ES (Open Graphics Library for Embedded System)
- new *system-on-chip* (CPU+GPU on a single chip), tablet-PC market, *entertainment center*, ...
  http://tegradeveloper.nvidia.com
- More ARM-based embedded systems, tools for better performance on Android/ARM:
  https://developer.nvidia.com/content/native-android-development-tutorial

FPGA : reconfigurable computing

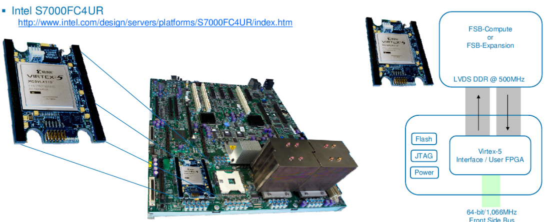
- **FPGA** : **programmable** integrated circuits, firmware (hardware with software flexibility), array of configurable logic block (encode a n-input boolean function) + routing tracks
- since ~ 1985
- low-level HDL (*Hardware description language*) e.g. VHDL (ADA-like), allow to design a functionality at clock cycle level
- IP (*Intellectual Properties*) library
- more and more complex : integration of communication block (Ethernet, Gigabit Tranceiver, etc) and computing blocks (PowerPC CPU, bloc DSP, small floating-point ALU (2003))
- *reconfigurable computing* : can we use FPGA general purpose computing ?
- since 2004-2005, emergence of high-level languages (C-like) + small board (with PCI-express form factor) + driver
- high-level language as always allows abstraction hiding low-level design flow (logic synthesis into RTL, placement, routing, ...) focus on scientific algorithm

FPGA : reconfigurable computing

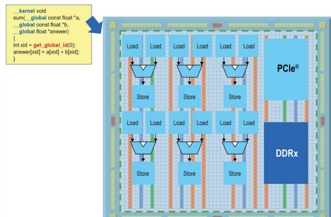
- **FPGA** : **programmable** integrated circuits, firmware (hardware with software flexibility), array of configurable logic block (encode a n-input boolean function) + routing tracks
- since ~ 1985
- low-level HDL (*Hardware description language*) e.g. VHDL (ADA-like), allow to design a functionality at clock cycle level
- IP (*Intellectual Properties*) library
- more and more complex : integration of communication block (Ethernet, Gigabit Tranceiver, etc) and computing blocks (PowerPC CPU, bloc DSP, small floating-point ALU (2003))
- *reconfigurable computing* : can we use FPGA general purpose computing ?
- Appealling but design too complex (to much low-level, i.e. hardware knowledge required to make efficient use of them), FPGA are expensive...

FPGA : reconfigurable computing / hardware accelerator

- See http://www.nallatech.com/



- OpenCL for Altera FPGAs available (Nov 2012)



- OpenCL for FPGA webcast

FPGA : reconfigurable computing - Links

Some links :

- bibliographic reference : Reconfigurable Computing, *Accelerating Computation with FPGA*, by M. Gokhale and P.S. Graham, Springer, 2005.
- wikipedia Reconfigurable computing
- workshop HPRCTA10
- OpenCL for FPGA OpenCL for FPGA webcast

- FPGA seem less appealing after the rise of `GPU Computing` and possible merging CPU/GPU architectures and developpement tools

credits:

http://fpgablog.com/posts/stratix-front-side-bus/

XtremeData XD2000i In-Socket Accelerator (2008)

FPGA : reconfigurable computing - Applications

Applications :

- computing applications : SDR (*Software Defined Radio*, telecommunication, radio-astrophysics, militaire)
- applications : high throuput networking / storage : low latency (finance, *trading*), compression, ...
- some vendors : Nallatech, DRC Computer, XtremeData
- commercial high-level tools (*C-to-RTL*) : Mitrion-C, ImpulseC, ...

Cons / difficulties :

- FPGA are expensive (≠ GPU, gaming market)
- Design tools complexity for software programmers
- Manufacturers involve in HPC ?

FPGA-based heterogenous computing in 2015

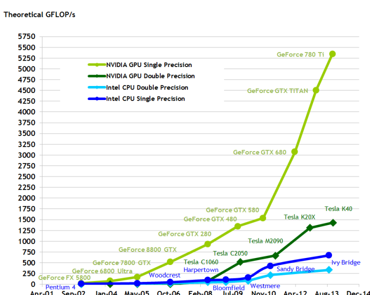
- Reconfigurable computing reloaded ?
- Internet of Things (IoT) / Cloud services / Data Center / Big Data / Deep learning applications drives the trend
- Intel bought Altera in June 2015
- IBM is at the heart of OpenPOWER foundation to develop software/accelerated solutions through partnership, e.g. : with NVIDIA for GPU (e.g. future OLCF/SUMMIT supercomputer) or with XILINX for FPGA

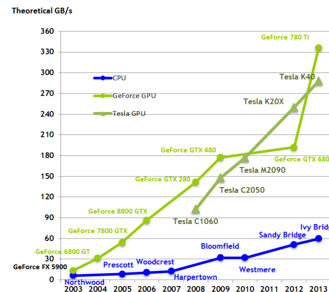
Summary

1. GPU computing: Architectures, Parallelism and Moore law
   - Why multi-core ?
   - Understanding hardware, better at optimization
   - What's a thread ?
   - History
2. CUDA Programing model
   - Why shoud we use GPU ?
   - Hardware architecture / programming model
   - CUDA : optimisation / perf measurement / analysis tools
   - GPU computing : perspectives / Install CUDA
3. Other languages: CUDA/Fortran, PyCuda, CUDA/MPI
4. Extra slides

Why should we use GPU ?

- Brut force (high peak GFLOPS rate)
- lower GFLOPS cost
- massively *multi-thread* architecture





Why should we use GPU ?

- compare CPU/GPU : much more computing-dedicated transistors and less control logic on GPU

CPUs: Memory Bandwidth Bottleneck

NVIDIA

240 cores

Thread Manager

Core 1

Core 2

L2 Cache

64 bit

FSB

12.8 GB/sec

102 GB/sec

GDDR3

512 bit

Main Memory

Main Memory

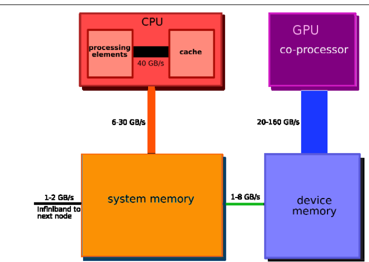
8x faster interface

Why should we use GPU ?

- **accessibility : the programming model is *simple* compared to legacy GPGPU** :
  - creation of a **structured** set of *threads*
  - **CRCW** memory model, **explicit** memory managing
  - C-like language (C + extensions)
  - graphics API (OpenGL) interoperability; (ease development of a GUI without the need to readback data on CPU memory)
- **CUDA : Compute Unified Device Architecture**
  Name of the architecture (hardware) and the programming model (software)

CPU/GPU : memory bandwidth

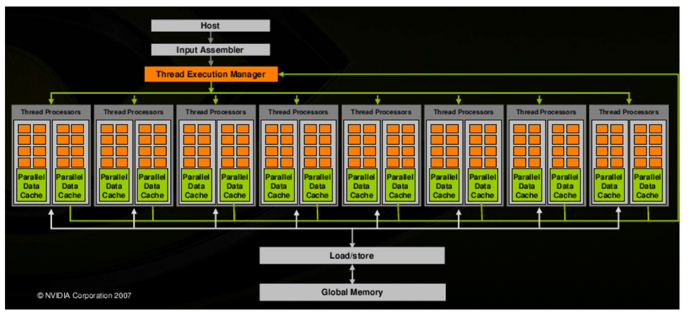
- **CPU-GPU link**, Pci-express bus x16, Gen2 :
  $BP = 16 * 2 * 250$MBytes/s = **8 GBytes/s**
- **CPU-local** : DDR memory ($f = 266$MHz)
  BP= $266 * 10^6$*4 (transferts/cycle) * 8 (bytes) = **8.5 GBytes/s**
- **GPU-local (carte GTX280)** : 512-bit bus, DDR@$f = 1100$MHz,
  BP= $2 * 1100 * 10^6$(transferts/s) * 512/8 (bytes/transfert)= **140 GBytes/s**



Hardware architecture: Nvidia CUDA : G80 to Fermi

CUDA course :

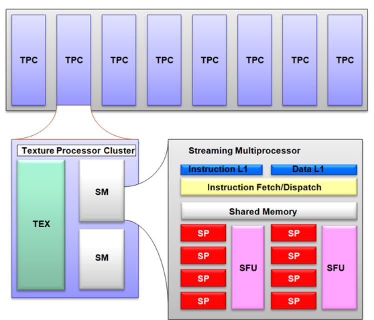
http://courses.engr.illinois.edu/ece498al/Archive/Spring2007/Syllabus.html



- 8 TPC (*Thread Processor Cluster*) - 2 SM (*Streaming Multiprocessor*) - 8 SP (*Streaming Processor*, SIMD)
- 8 TPC × 2 SM × 8 SP-cores = 128 cores

Hardware architecture: Nvidia CUDA : G80 to Fermi

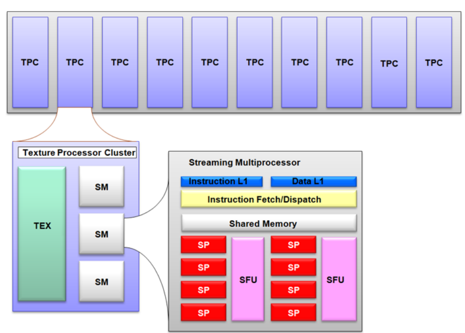
CUDA : G80 (end 2006), hardware capability 1.0



- 8 TPC (*Thread Processor Cluster*) - 2 SM (*Streaming Multiprocessor*) - 8 SP (*Streaming Processor*, SIMD)
- 8 TPC × 2 SM × 8 SP-cores = 128 cores

Hardware architecture: Nvidia CUDA : G80 to Fermi

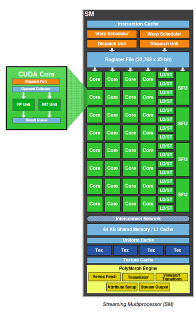
CUDA : GT200 (mid 2008), hardware capability 1.3



- 10 TPC - 3 SM - 8 SP-cores
- 10 TPC × 3 SM × 8 SP-cores = 240 cores

Hardware architecture: Nvidia CUDA : G80 to Fermi

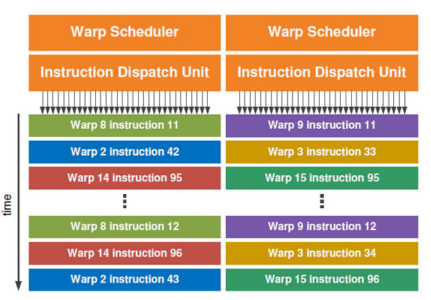
CUDA : Fermi (spring 2010), hardware capability 2.0



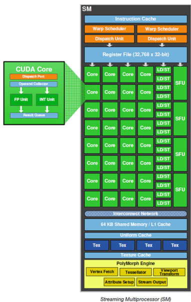
Streaming Multiprocessor (SM)

- **Streaming Multiprocessor** (32 cores), hardware ctrl, queuing system
- **GPU = scalable array of SM** (up to 16 on Fermi)
- **warp: vector of 32 threads**, executes the same instruction in lock-step
- cache L1/L2
- **throughput limiters**: finite limit on warp count, on register file, on shared memory, etc...

Hardware architecture: Nvidia CUDA : G80 to Fermi



Dual warp scheduling



Streaming Multiprocessor (SM)

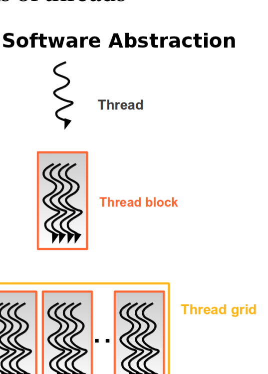
- **Streaming Multiprocessor** (32 cores), hardware ctrl, queuing system
- **GPU = scalable array of SM** (up to 16 on Fermi)
- **warp: vector of 32 threads**, executes the same instruction in lock-step
- cache L1/L2
- **throughput limiters**: finite limit on warp count, on register file, on shared memory, etc...

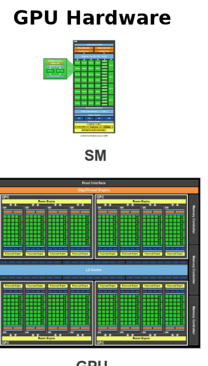
Architectures comparison: G80 - GT200 - Fermi

GPU	G80	GT200	GF100
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

CUDA - connecting program and execution model

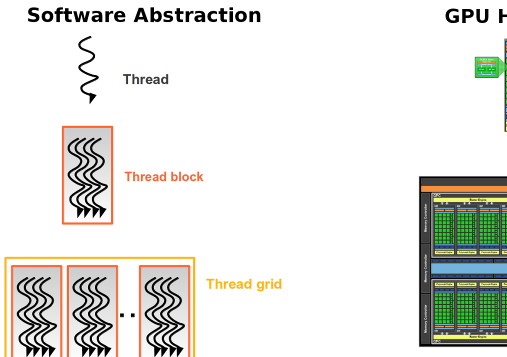
- **Need a programing model to efficiently use such hardware**; also provide **scalability**
- Provide a simple way of partitioning a computation into fixed-size blocks of threads





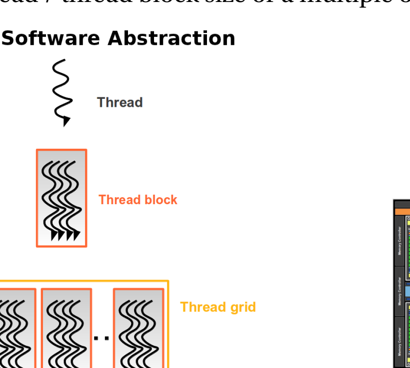
CUDA - connecting program and execution model

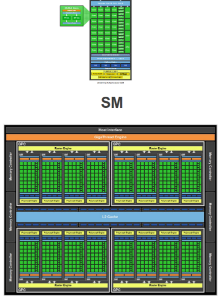
- **Total number of threads** must/need be quite larger than number of cores
- **Thread block** : **logical array of threads**, large number to hide latency
- **Thread block size** : control by program, specify at runtime, better be a multiple of warp size (i.e. 32)



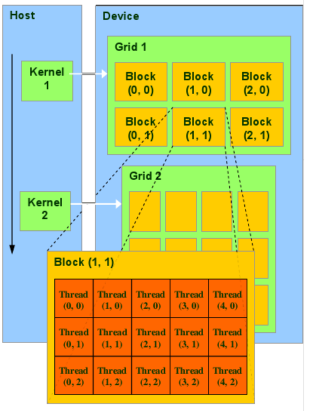
CUDA - connecting program and execution model

- **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



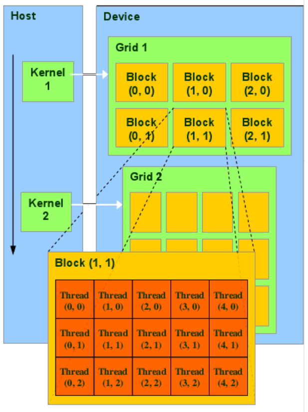


CUDA : programming model (PTX)



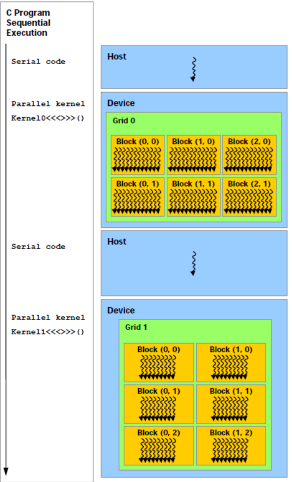
- a block of threads is a **CTA (Cooperative Thread Array)**
- **Threads are indexed inside a block; use that index to map memory**
- write a program once for a *thread*
- run this program on multiple *threads*
- **block** is a logical array of threads indexed with *threadIdx* (**built-in variable**)
- **grid** is a logical array of blocks indexed with *blockIdx* (**built-in variable**)
- Read chapter 1, 2, 3 of ptx_isa_4.3.pdf

CUDA : programming model (PTX)



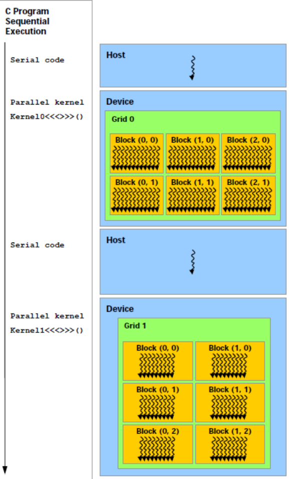
- **threads** from a given block can:
  - synchronize execution
  - exchange data using a shared memory space
- **blocks** are independent, no cross-block synchronisation, **execution in undefined order**

CUDA : programming model (PTX)



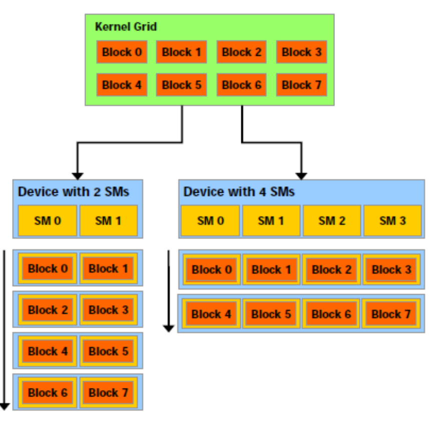
- heterogeneous systems : CPU and GPU have separated memory spaces (*host* and *device*)
- the programmer focuses on code parallelization (algorithm level) not on how he was to schedule blocks of threads on multiprocessors.
- PTX (Parallel Thread Execution); name of the **virtual machine (and instruction set, assembly language)** that exposes the GPU as a data-parallel computing device

CUDA : programming model (PTX)



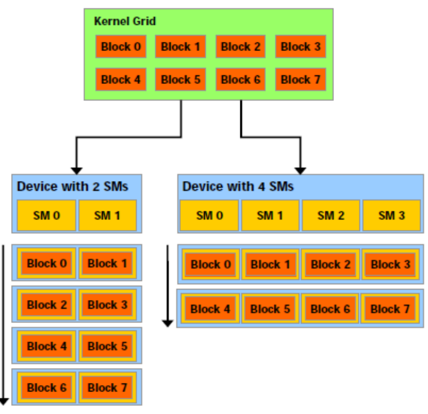
- Why do we divide the block into block ?
- When are blocks run ?
- In what order do they run ?
- How can threads **cooperate** ? Limitations ?

CUDA : Programming model



- each *block of threads* can be scheduled on any of the available SM (can not rely on a specific order to design algoritms)
  - concurrently (on different multiprocessors)
  - sequentially (on the same multiprocessor)
- independent execution of blocks of threads gives **scalability** of the programming model.

CUDA : Programming model



- **Why blocks ?**
  - Distribute work among SMs, load balance, keep them all busy
- independent execution of blocks of threads gives **scalability** of the programming model.

CUDA : Programming model

Connection: Hardware ↔ Programming Model

Axis 0

Axis 1

(Work) Group

Grid

(Kernel: Function on Grid)

(Work) Item

Software representation

Hardware

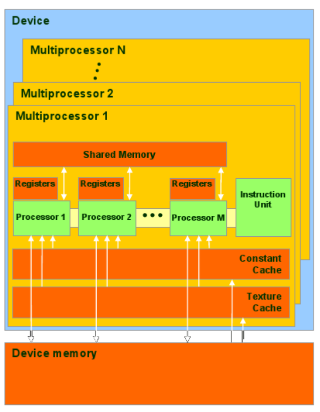
credits: slides by Andreas Klöckner (author of PyCUDA / PyOpenCL)

CUDA : Programming model

Connection: Hardware ↔ Programming Model

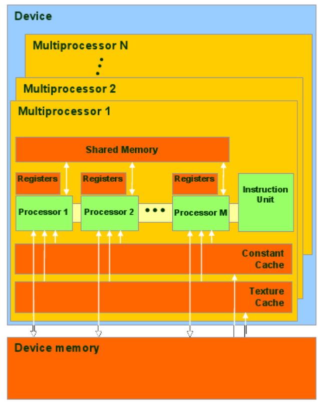
credits: slides by Andreas Klöckner (author of PyCUDA / PyOpenCL)

CUDA memory hierarchy : G80/GT200



- **CPU and GPU have physically separated memory spaces**
  - data transfert to/from GPU are explicit and controlled by CPU : GPU can't initiate transfers, acces disk, ...
  - dedicated memory management for allocation, free, transfers, ...
- **pointers are only addresses...**
  - Can't tell from the pointer value wether the address is CPU or GPU memory space !! (changed in cuda 4 with UVA)
  - dereferencing a CPU pointer inside a routine running on GPU ⇒ **CRASH !**
- **Unified Virtual Memory**: you can delegated actual memory transfert to the system

CUDA memory hierarchy : G80/GT200



- ***on-chip memory*** :
  - **shared** : RW, very fast (if no bank conflicts), 16kB/multiprocessor
  - **register** : RW, very fast, 8-16kB/multiprocessor
- ***off-chip memory*** :
  - **global** : RW, up to GBytes, slow (~ 100 clock cycles)
  - **constant** : RO, 64kB/chip, __*const*__ declared variable, very fast (1-4 cycles), cached
  - **texture** : RO, located in global memory, cached
  - **local** : RW, slow, use controlled by compiler, used if no more registers

GPU memory model summary for legacy/modern GPGPU

Limited memory access during computation

- **register** (per fragment/thread)
  - read/write
- **local memory** (shared among threads)
  - Does not exist in general (legacy GPGPU)
  - CUDA allows access to shared memory between threads
- **Global memory** (historical)
  - read-only during computation
  - write-only at the end of computation (precomputd address)
- **Global memory** (new, i.e. CUDA)
  - allows general scatter/gather (read/write)
  - take care: no collision rules, need atomic operations

CUDA global/external memory : allocation/release

- **Host (CPU) manages device (GPU) memory:**
  - header `cuda_runtime.h` for run time API
  - `cudaError_t cudaMalloc(void ** pointer, size_t nbytes)`
  - `cudaError_t cudaMemset(void * pointer, intvalue, size_tcount)`
  - `cudaError_t cudaFree(void* pointer)`
- **example use:**

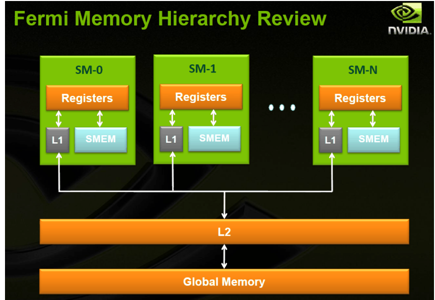
```
 int n = 1024;
int nbytes = 1024*sizeof(int);
int* d_a = 0;
cudaMalloc( (void**)&d_a, nbytes);
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

GPU memory : data transfer

- `cudaError_t cudaMemcpy(void *dst, void *src, size_t nbytes, enumcudaMemcpyKinddirection)`
  - header `cuda_runtime_api.h` for run time API
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn't start copying until previous CUDA calls complete
- `enum cudaMemcpyKind`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

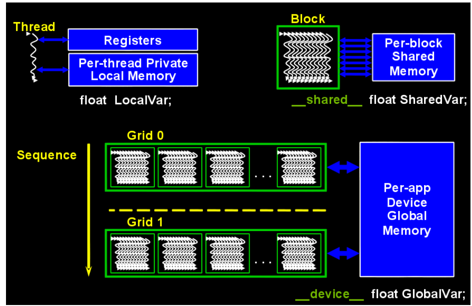
CUDA memory hierarchy: software/hardware

- **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



CUDA C/C++ (software point of view)

- **CUDA C/C++**
  - **Large subset of C/C++ language**
  - CPU and GPU code in the same file; preprocessor to filter GPU specific code from CPU 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
    - **kernel**: routine executed on GPU

CUDA : C-language extensions and run-time API

- Function and type qualifiers

```
__global__ void KernelFunc(...); // kernel callable from host
__device__ void DeviceFunc(...); // function callable on device
__device__ int  GlobalVar;       // variable in device memory
__shared__ int  SharedVar;       // shared in PDC by thread block
__host__   void HostFunc(...);   // function callable on host
```

- built-in variables : *threadIdx* and *blockDim*, *blockIdx* and *gridDim* (*read-only registers*)
- kernel function launch syntax

```
KernelFunc<<<500, 128>>>(...); // launch 500 blocks w/ 128 threads each
```

  «< .. »> is used to set grid and block sizes (can also set shared mem size per block)
- synchronisation *threads* inside bloc

```
__syncthreads(); // barrier synchronization within kernel
```

- *libc*-like routine (e.g.: memory allocation, CPU/GPU data transfer, ...)

CUDA Code walkthrough

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

CPU program

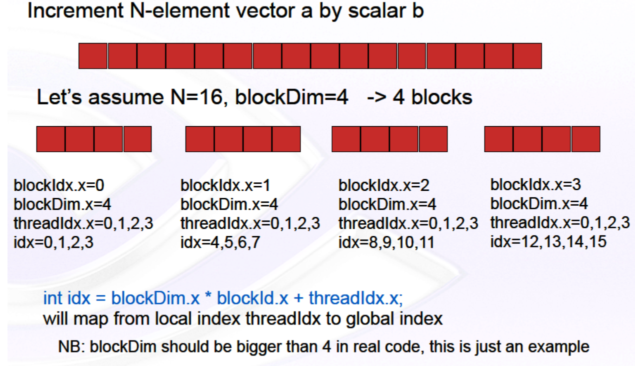
```
void increment_cpu(float *a, float b, int N)
{
      for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}


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**



CUDA Code walkthrough

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

```
/*
 * nvcc -m64 -gencode arch=compute_20,code=sm_20 --ptxas-options -v
 * -o scalarAdd scalarAdd.cu
 */
#include <stdio.h>

/**
 * a simple CUDA kernel
 *
 * \param[inout]  a input integer pointer
 * \param[in]     b input integer
 * \param[in]     n input array size
 */
__global__ void add( int *a, int b, int n ) {

  int idx = threadIdx.x + blockIdx.x*blockDim.x;

  if (idx<n)
    a[idx] = a[idx] + b;
}
```

CUDA Code walkthrough

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

```
/*
 * main
 */
int main( void ) {
  // array size
  int N = 16;

  // host variables
  int *a;  int  b;

  // device variables
  int *dev_a;

  // CPU memory allocation
  a = (int *) malloc(N*sizeof(int));
  b = N;
  // CPU memory initialization
  for (int i=0; i<N; i++) a[i]=i;
```

```
// GPU device memory allocation
cudaMalloc( (void**)&dev_a,
            N*sizeof(int) ) ;

// GPU device memory initialization
cudaMemcpy( dev_a, a, N*sizeof(int),
            cudaMemcpyHostToDevice ) ;

// perform computation on GPU
int nbThreads = 8;
dim3 blockSize(nbThreads,1,1);
dim3 gridSize((N+1)/nbThreads,1,1);
add<<<gridSize,blockSize>>>( dev_a, b, N );

// get back computation result
// into host CPU memory
cudaMemcpy( a, dev_a, N*sizeof(int),
            cudaMemcpyDeviceToHost ) ;
```

CUDA Code walkthrough

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

```
  // do something !

  // de-allocate CPU host memory
  free(a);

  // de-allocate GPU device memory
  cudaFree( dev_a ) ;

  cudaDeviceSynchronize();
  cudaDeviceReset();

  return 0;
}
```

CUDA Code walkthrough

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

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 Code walkthrough

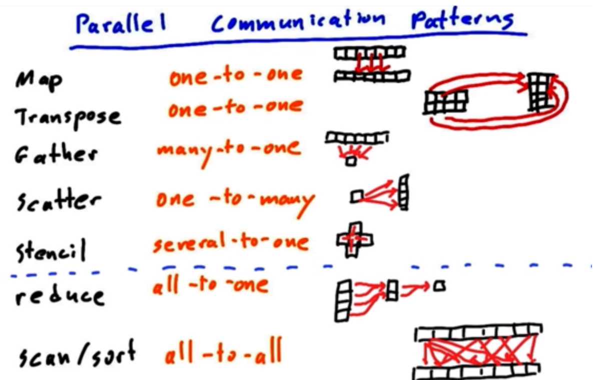
CPU program

```
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 N)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if( idx < N)
            a[idx] = a[idx] + b;
}

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

CUDA - Parallel communication patterns

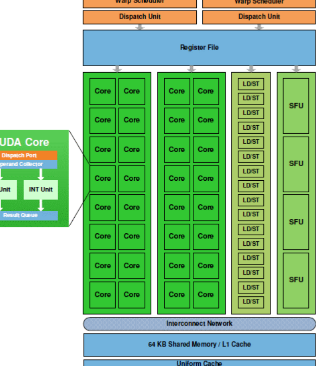


source: J. Owens, D. Luebke, Udacity cs344, lesson 2

CUDA : Execution model- notion of *warp*

/usr/local/cuda-5.5/doc/pdf/ptx_isa_3.2.pdf (240 pages)

- **GPU execution requires the *kernel code* + grid,block geometry**
  - Concurrent kernel execution (hardware $\geq$ 2.0)
- **multiprocessor control unit** creates, manages, organize thread (*scheduling*); threads are grouped into **warp** (group of 32 threads with consecutive indexes)
  $\Rightarrow$ **hardware resources sharing !**

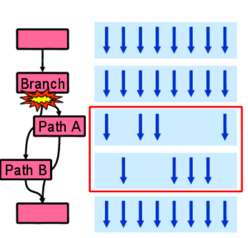


Fermi Streaming Multiprocessor (SM)

CUDA : Execution model- notion of *warp*

- ***branch divergence***: A warp executes one common instruction at a time. If threads of a warp diverge via a data-dependent conditional branch (if-then-else with condition on threadIdx e.g.), the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Performance drops. See reduction code example.

  - GPU uses SIMD pipeline to save area on control logic.
    - Group scalar threads into warps
  - *Branch divergence* occurs when threads inside warps branches to different execution paths.



credits : W. Fung, Dynamic warp formation and scheduling for efficient GPU flow control

CUDA : Execution model- notion of *warp*

- Execution context (PC, registers, ...) for each warp is maintained on-chip during the entire lifetime of the warp !
- **GPU *threads* GPU are very lighweight** (creation and context switching are almost free, i.e. only take a few cycles).
- Read CUDA_C_Programming_Guide.pdf, chapter 4
- The number of blocks and warps that can reside and be processed together on the multiprocessor for a given kernel depends on the amount of registers and shared memory used by the kernel and the amount of registers and shared memory available on the multiprocessor. (Cuda Programming guide, section 4.2)

CUDA : SIMT Multithread Execution

	Compute Capability						
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**
Maximum dimensionality of grid of thread blocks	2				3		
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1	
Maximum y- or z-dimension of a grid of thread blocks	65535						
Maximum dimensionality of thread block	3						
Maximum x- or y-dimension of a block	512				1024		
Maximum z-dimension of a block	64						
Maximum number of threads per block	512				1024		
Warp size	32						
Maximum number of resident blocks per multiprocessor	8					16	
Maximum number of resident warps per multiprocessor	24		32		48	64	
Maximum number of resident threads per multiprocessor	768		1024		1536	2048	
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K	

CUDA_C_Programming_Guide.pdf, appendix G

For a given kernel (given number of register per thread, given shared memory per block), this table helps understanding which hardware resources will be exhausted first when changing run-time parameters (grid size, block size).

CUDA : SIMT Multithread Execution

Based on Brent Oster's slides (NVIDIA)

NVIDIA Parallel Execution Model

NVIDIA

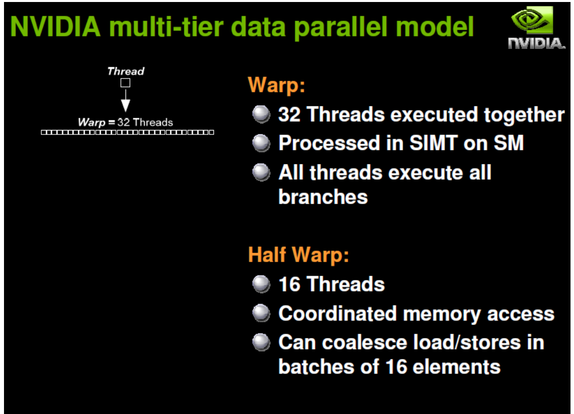
Thread

Thread:

- Runs a kernel program and performs the computation for 1 data item.
- Thread Index is a built-in variable
- Has a set of registers containing it's program context

CUDA : SIMT Multithread Execution

Based on Brent Oster's slides (NVIDIA)



CUDA : SIMT Multithread Execution

Based on Brent Oster's slides (NVIDIA)

NVIDIA multi-tier data parallel model

NVIDIA

Thread

Warp = 32 Threads

Block of Threads

Block:

- 1 or more warps running on the same SM
- Different warps can take different branches
- Can synchronize all warps within a block
- Have common shared memory for extremely fast data sharing

CUDA : SIMT Multithread Execution

Based on Brent Oster's slides (NVIDIA)

SIMT Multithreaded Execution

NVIDIA

Single-Instruction Multi-Thread instruction scheduler

time

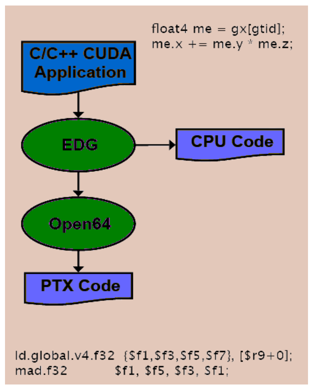
warp 8 instruction 11
warp 1 instruction 42
warp 3 instruction 95
⋮
warp 8 instruction 12
warp 3 instruction 96

SP SP SP SP SP SP SP SP

- SIMT: Single-Instruction Multi-Thread
- Warp: the set of 32 parallel threads that execute a SIMT instruction
- Hardware implements zero-overhead warp and thread scheduling
- Deeply pipelined to hide memory and instruction latency
- SIMT warp diverges and converges when threads branch independently
- Best efficiency and performance when threads of a warp execute together

CUDA : compilation workflow

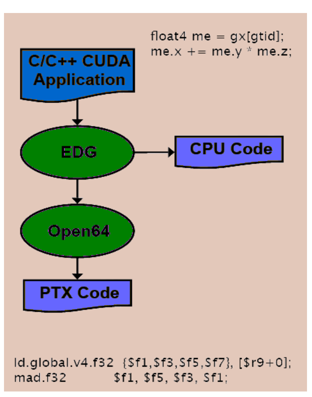
http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf



- **NVCC : *compiler driver*** : call behind nvopencc (open64), gcc/g++, ...)
- **PTX : *Parallel Thread eXecution***
- PTX defines an **ISA (*Instruction Set Architecture*)** and a low-level **virtual machine** providing hardware abstraction (portability across GPU hardware evolution, GPU generations, scalability accross GPU sizes and number of SM)

CUDA : compilation workflow

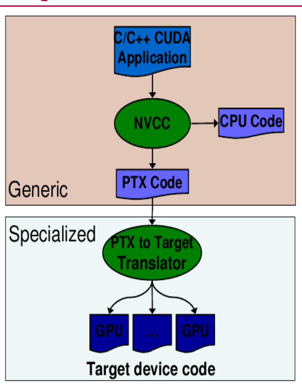
http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf



- **NVCC :** ***compiler driver*** : call behind nvopencc (open64), gcc/g++, ...)
- **PTX :** ***Parallel Thread eXecution***
- High-level language compilers (e.g. nvcc) generate PTX instructions, which in a second stage, are optimized and translated into native hardware instructions (depending hardware capability).
- Possibility to define other high-level languages to target the same ISA (e.g. CUDA-Fortran)

CUDA : compilation workflow

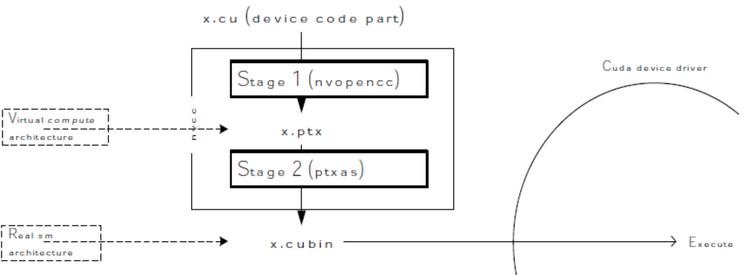
http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf



- second stage uses the **ptxas tool: PTX assembly to cubin** (low-level machine instruction)
- graphics driver can also convert PTX into CUBIN (***Just-In-Time* optimisation**) and issue a PCI-express upload to GPU.

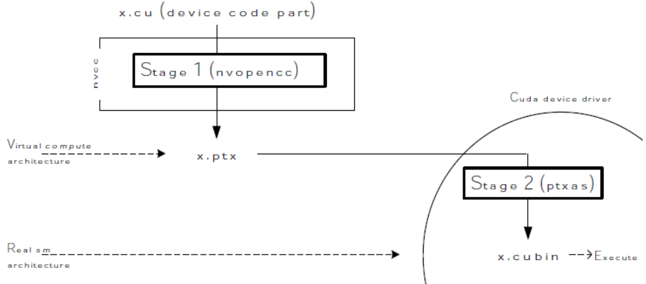
CUDA : compilation workflow

nvcc documentation : CUDA_Compiler_Driver_NVCC.pdf



CUDA : compilation workflow

nvcc documentation : CUDA_Compiler_Driver_NVCC.pdf



CUDA : C-language extensions and run-time API

- Function and type qualifiers

```
__global__ void KernelFunc(...); // kernel callable from host
__device__ void DeviceFunc(...); // function callable on device
__device__ int  GlobalVar;       // variable in device memory
__shared__ int  SharedVar;       // shared in PDC by thread block
__host__   void HostFunc(...);   // function callable on host
```

- built-in variables : *threadIdx* and *blockDim*, *blockIdx* and *gridDim* (*read-only registers*)
- kernel function launch syntax

```
KernelFunc<<<500, 128>>>(...); // launch 500 blocks w/ 128 threads each
```

«< .. »> is used to set grid and block sizes (can also set shared mem size per block)

- synchronisation *threads* inside bloc

```
__syncthreads(); // barrier synchronization within kernel
```

- *libc*-like routine (e.g.: memory allocation, CPU/GPU data transfer, ...)

CUDA : C-Programming and run-time API

- specific data types, (vector_types.h) : example *dim3*, design for memory alignment

		Memory	Scope	Lifetime
`__shared__`	`int SharedVar;`	shared	thread block	thread block
`__device__`	`int GlobalVar;`	global	grid	application
`__constant__`	`int ConstantVar;`	constant	grid	application

- in a *CUDA kernel*, automatic variables (i.e. without any type qualifiers) generally reside in a register. In some case, the compiler can choose to place them in local memory (external RAM, take care of performance drop, might need to rewrite the alogrithm to fit in register)
- size of an array placed in shared memory can be either set explicitely or only at launch time, in that case use declaration:
  `extern __shared__ float shared[];`

CUDA : C-Programming and run-time API

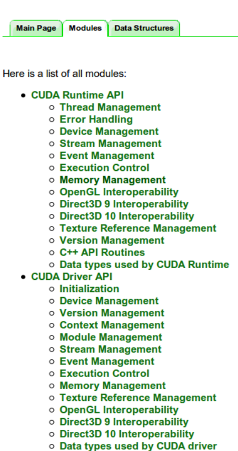
- specific data types, (vector_types.h) : example *dim3*, design for memory alignment

		Memory	Scope	Lifetime
`__shared__`	`int SharedVar;`	shared	thread block	thread block
`__device__`	`int GlobalVar;`	global	grid	application
`__constant__`	`int ConstantVar;`	constant	grid	application

- a `__constant__` **declared variable** is placed in the constant memory (external DRAM but cached, so very fast, few clock cycles);
- has the lifetime of the application (no need to re-init between different kernel calls);
- ~~has **static storage** (take care if you want to use it in multiple compilation unit; you can't use the `extern` keyword);~~
- ~~if you want to use the same constant variable in multiple compilation unit (different .cu files), you need to init constant memory in each of them, i.e. call `cudaMemcpyToSymbol`~~ since CUDA 5.0, separation compilation feature, `extern` allowed

CUDA : C-Programming and run-time API

`file:///usr/local/cuda-5.5/doc/html/cuda-runtime-api/index.html`



- **Runtime API** : high-level, build on top of the driver API (init, context and module management are implicit, code is concise), prefix `cuda` *device emulation*
- **Driver API** : low-level, better level of control, harder to program/debug, optimisation PTX JIT (Just-In-Time), code lengthy, prefix `cu`
- **GPU Context ~ CPU process**
- **GPU Module ~ CPU dynamic library**
- Driver and Run-time API can be used/linked together

CUDA Run-time API: code walkthrough

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 Run-time API: code walkthrough

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 Run-time API: code walkthrough

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 N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if( idx < N)
        a[idx] = a[idx] + b;
}

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

CUDA Run-time API: code walkthrough

- example : sum of vectors

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

- invocation

```
void main() {
  // allocate device (GPU) memory
  float* d_A, d_B, d_C;
  cudaMalloc( (void**) &d_A, N * sizeof(float));
  cudaMalloc( (void**) &d_B, N * sizeof(float));
  cudaMalloc( (void**) &d_C, N * sizeof(float));

  // copy host memory to device
  cudaMemcpy( d_A, h_A, N * sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy( d_B, h_B, N * sizeof(float),cudaMemcpyHostToDevice);

  // Execute on N/256 blocks of 256 threads each
  vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

CUDA Optimisations

A CUDA program should take into account the following constraints

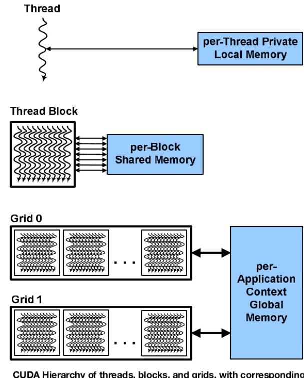
- **Coalescent acces to global memory** : *threads* with consecutive indexes should access consecutive memory addresses for good alignment
- **Use shared memory** (high bandwidth, low latency)
- **Efficient use of parallelism**
  Keep GPU busy as long as possible
  Try to have a high computing ops / memory acces
  Make a good use of thread hierarchy
- **Try to avoid shared memory bank conflicts**

Parallel primitives

parallel-prefix-sum ("scan"), parallel sort and parallel reduction

- Thrust : http://code.google.com/p/thrust
- cudpp : http://gpgpu.org/developer/cudpp
- comparison Thrust/CUDPP : http://code.google.com/p/thrust/wiki/ThrustAndCUDPP
- cub : http://nvlabs.github.io/cub/ (new, 2013)
- reduction example reduction in CUDA SDK, slides by Mark Harris
- **advanced reference:** http://nvlabs.github.io/moderngpu/ by Sean Baxter, Nvidia

CUDA: summary



CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid.

A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-Block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in Global Memory space after kernel-wide global synchronization.

CUDA: summary

- Heterogeneous system architecture
- Use the right processor and memory for each task
- CPU excels at executing a few serial threads
  - Fast sequential execution
  - Low latency cached memory access
- GPU excels at executing many parallel threads
  - Scalable parallel execution
  - High bandwidth parallel memory access

CPU
Cache
Host Memory
Bridge
PCIe
GPU
SMem
SMem
SMem
Cache
Device Memory

CUDA: summary

```
kernel_func<<<nblk, nthread>>>(param, … );
```

Host Thread
Grid of Thread Blocks
CPU
Cache
Host Memory
Bridge
PCIe
GPU
SMs:
SMem
SMem
SMem
Cache
Device Memory

CUDA: summary

float myVar;
__shared__ float shVar;
__device__ float glVar;

Thread
Block
Registers
Per-thread Local Memory
Per-block Shared Memory
Per-app Device Global Memory
GPU
SMs:
SMem
SMem
SMem
CPU
Cache
Cache
Host Memory
Bridge
PCIe
Device Memory

CEA hybrid CPU/GPU Cluster, 2009 - Titane

Titane: ~ 100 CPU-TFLOPS + ~ 200 GPU-TFLOPS (Tesla S1070, hw 1.3)

CEA Hybrid Cluster

BULL

NVIDIA

1,000 TB HDD
LUSTRE file system

25TB RAM

3 x 42U rack for GPU
192 TFlops Peak

17 x 42U Bull NovaScale
103 TFlops Peak

Innovative GPU Platform
Allow specific applications to use multiple CPU-GPU cores and the entire system RAM

SMP Production Platform
- 3GB RAM per CPU
- For all industrial and research applications

Over 295 TFlops
#1 in Europe

42U rack with 16 pcs Tesla S1070
42U rack with 60 pcs Intel Nehalem CPU. 3GB RAM/core
HDD, network, service nodes
Infiniband DDR

Open Source system software

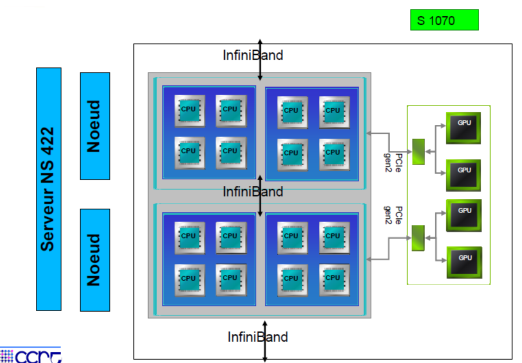
61

- `cudaGetDeviceCount(&count);` returns 2 because a Titane node only *sees* half a Tesla-S1070, i.e. 2 GPU devices.

http://www-ccrt.cea.fr/fr/moyen_de_calcul/titane.htm

CEA hybrid CPU/GPU Cluster, 2009 - Titane

Titane: ~ 100 CPU-TFLOPS + ~ 200 GPU-TFLOPS (Tesla S1070, hw 1.3)



- `cudaGetDeviceCount(&count);` returns 2 because a Titane node only *sees* half a Tesla-S1070, i.e. 2 GPU devices.

http://www-ccrt.cea.fr/fr/moyen_de_calcul/titane.htm

CUDA atomic operations

- What append when multiple threads in a warp try to modify data at the same address (global or shared memory) ?
- which thread performs the final write is **undefined** !
- examples algorithms: histogram or tree-building
- example atomic function: `int atomicAdd(int* address, int val);` : no other thread can access this address until the operation is complete.
- See CUDA programming guide, Section B.5 (Memory fence functions) to have another example of reduction implementation that uses atomic functions.

CUDA printf for debug

- `int printf(const char *format[, arg, ...]);`
- similar to the standard C-library (`stdio.h`)
- **only available for hardware 2.0 (Fermi)**
- Use with care ! Don't forget to reduce the block sizes or use *if* guards to authorize only certain threads to print !
- See CUDA programming guide, section B.14

CUDA: Miscellaneous development tools...

... for numerical applications

- cuFFT: CUDA-based FFT implementation,
- cuBLAS: CUDA-based Basic Linear Algebra by NVIDIA,
- culatools: GPU linear algebra package, $\sim cuBLAS$,
- MAGMA: Matrix algebra on GPU and multi-core (faster than cuBLAS),
- openCurrent: C++ library for solving PDE on GPU,
- cusp: sparse linear algebra on GPU,
- openNL: sparse linear algebra,
- libra: GPU SDK for Matlab,
- cudpp: data parallel primitive for GPU (see also Thrust)

http://www.nvidia.com/object/tesla_software.html

GPU computing challenges

- **Computations with no known scalable parallel algorithms**
  - Shortest path, Delaunay triangulation, ...
- **Data distributions that cause catastrophical load imbalance in parallel algorithms**
  - Free-form graphs, MRI spiral scan
- **Computations that do not have data reuse**
  - Matrix vector multiplication, ...
- **Algorithm optimizations that are hard and labor intensive**
  - Locality and regularization transformations

credits: slides by Wen-mei Hwu

CUDA hardware - Why no global synch between blocks ?

The global synchronization *myth*

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right ?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- Problem: GPU has limited resources
  - GPU has M multiprocessors
  - Each multiprocessor can support a limited # of blocks, b
  - If total # blocks $B > M * b$ ... DEADLOCK
- Also, GPUs rely on large amount of *independent* parallelism to cover memory latency
  - Global synchronization destroys independence

Why using GPU ?

- Solve problems faster
- Solve bigger problems
- Solve more problems

Optimizing GPU program

Principals of efficient GPU programming. What do we want ?

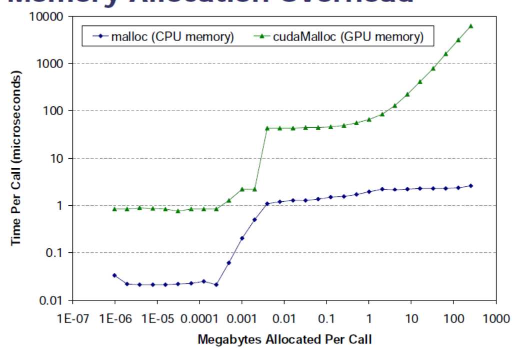
- decrease arithmetic intensity ? NO
- decrease time spent on memory operations ? YES
- coalescent global memory accesses ? YES
- do fewer memory operations per thread ? NO, not necessarily
- avoid thread divergence ? YES
- move all data to shared memory ? NO, only frequently accessed

reference: Udacity, cs344, lesson 5

CUDA : Memory allocation and transfer time

Based on K. Skadron slides (University of Virginia)

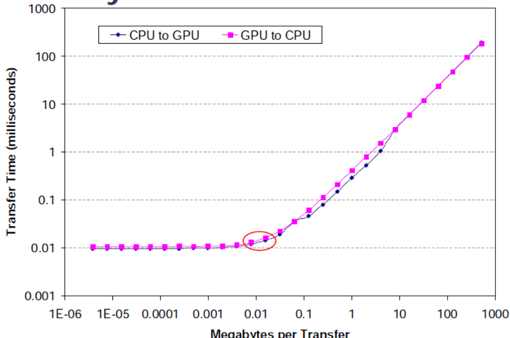
- GPU global/external memory allocation is costly



CUDA : Memory allocation and transfer time

Based on K. Skadron slides (University of Virginia)

- transfer time CPU ←→ GPU (think about tranfer overhead before off-loading computations to GPU)

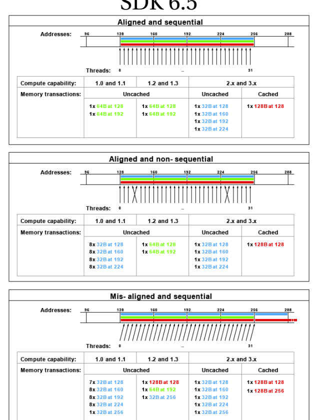


CUDA optimisation

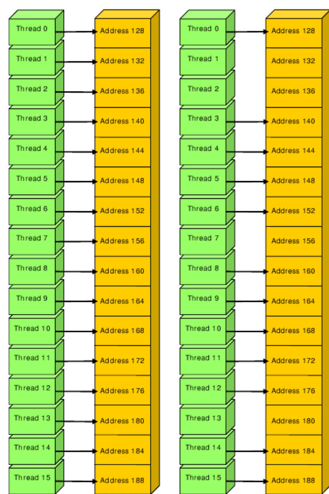
- slides from isc2009 : CUDA Optimisation
- slides `http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/GPU+CUDA.pdf` matrix transposition example code (illustrate CUDA concepts like coalesced memory R/W, shared memory, bank conflict ...) from slide 142.

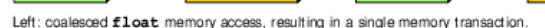
CUDA global memory R/W access: coalescence

Figure G1, /usr/local/cuda-6.5/doc/pdf/CUDA_C_Programming_Guide.pdf, SDK 6.5

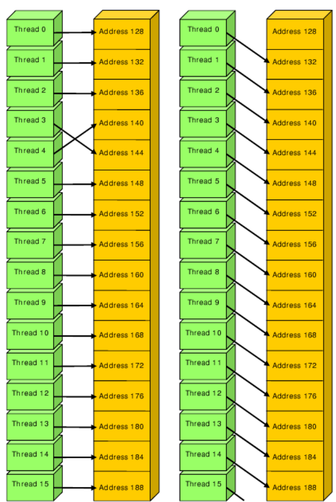


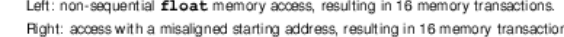
CUDA global memory R/W access: coalescence



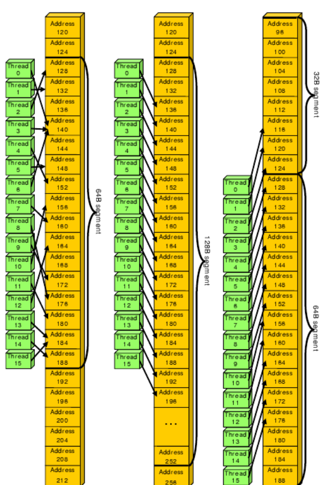

Left: coalesced **float** memory access, resulting in a single memory transaction.
Right: coalesced **float** memory access (divergent warp), resulting in a single memory transaction.

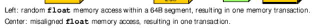
CUDA global memory R/W access: coalescence




Left: non-sequential **float** memory access, resulting in 16 memory transactions.
Right: access with a misaligned starting address, resulting in 16 memory transactions.

CUDA global memory R/W access: coalescence





Left: random **float** memory access within a 64B segment, resulting in one memory transaction.
Center: misaligned **float** memory access, resulting in one transaction.
Right: misaligned **float** memory access, resulting in two transactions.

CUDA optimisation : coalescence and SoA/AoS

- Array of Structures (AoS) ☹
  - 3 colors / pixel → alignment
  - complex access *pattern* to global memory
- Structure of Arrays (SoA) ☺
  - coalescence constraint by *design*
  - coalescence constraint still OK if you add more field/arrays; e.g. RGB → RGBA

```
struct Pixel {
    float r, g, b;
};
Pixel image_AoS[480][640];
```

```
struct Image {
    float R[480][640];
    float G[480][640];
    float B[480][640];
};
Image image_SoA;
```

http://perso.ens-lyon.fr/sylvain.collange/talks/calcul_gpu_scollange.pdf

CUDA : profiler

- Toolkit documentation: CUDA_Profiler_Users_Guide.pdf (for `nvvp` our command line usage)
- environment variables:
  - `CUDA_PROFILE` or `COMPUTE_PROFILE` : 1 for enable
  - `CUDA_PROFILE_LOG` or `COMPUTE_PROFILE_LOG` : results file name (default is launch directory/cuda_profile.log)
  - `CUDA_PROFILE_CONFIG` or `COMPUTE_PROFILE_CONFIG` : config file; list names of performance counters for logging information
- these variables are already set for hands-on tutorial.
- example of CUDA profile config file:

```
divergent_branch
warp_serialize
timestamp
gld_incoherent
gst_incoherent
```

CUDA : profiler / terminology

- gld_incoherent: Number of non-coalesced global memory loads
- gld_coherent: Number of coalesced global memory loads
- gst_incoherent: Number of non-coalesced global memory stores
- gst_coherent: Number of coalesced global memory stores
- local_load: Number of local memory loads
- local_store: Number of local memory stores
- branch: Number of branch events taken by threads
- divergent_branch: Number of divergent branches within a warp
- instructions: instruction count
- warp_serialize: Number of threads in a warp that serialize based on address conflicts to shared or constant memory
- cta_launched: executed thread blocks

CUDA optimisation : data prefetching

- data prefetching = **load data in advance** (e.g. for the next iteration in a loop) !!!
- When a memory access operation is executed, it does not block other operations following it as long as they don't use the data from the operation.
- `for` loop without data prefetching:

```
for (i = 0; i < N; i++) {
  sum += array[i];
}
```

- every addition waits for its data to be loaded from memory

CUDA optimisation : data prefetching

- data prefetching = **load data in advance** (e.g. for the next iteration in a loop) !!!
- When a memory access operation is executed, it does not block other operations following it as long as they don't use the data from the operation.
- `for` loop with data prefetching:

```
temp = array[0];
for (i = 0; i < N-1; i++) {
  temp2 = array[i+1];
  sum += temp;
  temp = temp2;
}
sum += temp;
```

- with data prefetching, inside the loop, memory load for iteration $i+1$ and the actual addition for iteration i are done in parallel
- data prefetching benefits: needs less warp to hide memory latency
- data prefetching cost for GPU: more registers, so less warps per SM

CUDA optimisation : APOD

Analyze - Parallelize - Optimize - Deploy

- See Udacity, cs344, lesson 5
- See also material by Julien Demouth (NVIDIA): GTC 2013, S3011, optimizing a sparse matrix-vector product
- See also material by Julien Demouth / Christoph Angerer (NVIDIA): GTC 2015, S5173, optimizing a sobel filter
  companion code:
  `https://github.com/chmaruni/nsight-gtc2015`

CUDA optimisation : APOD

- **Analyze:** Profile whole application
  - identify location where optimization can benefit ?
  - by how much ?
- **Parallelize:** Pick an approach, **an algorithm**
  - libraries
  - directives (OpenMP, OpenACC)
  - programming languages
- **Optimize:** use profile-driven optimization; use measurement on actual hardware
- **Deploy:**

Cuda Occupancy - modern GPU

A very pedagogical and insightful presentation from ModernGPU by Sean Baxter

- **Occupancy** is a measure of thread parallelism in a CUDA program; **max Nb of threads running / max Nb of threads possible**
- Each SM has limited number of
  - thread blocks
  - threads
  - total number of registers for all threads
  - amount of shared memory
- Use sdk example `deviceQuery` to know your hardware
- Use profiling tool `nvvp` to identify the limiting factor of a given cuda kernel
- **Higher occupancy doesn't necessary lead to higher performance !**

Cuda Occupancy - modern GPU

A very pedagogical and insightful presentation from ModernGPU by Sean Baxter

	sm_20	sm_30	sm_35
Max Threads (SM)	1536	2048	2048
Max CTAs (SM)	8	16	16
Shared Memory Capacity (SM)	48 KB	48 KB	48 KB
Register File Capacity (SM)	128 KB	256 KB	256 KB
Max Registers (Thread)	63	63	255

See also cuda c best practices guide

Cuda Occupancy - modern GPU

A very pedagogical and insightful presentation from ModernGPU by Sean Baxter

Simple heuristics:

- Total number of thread blocks (CTA):
  - Number of thread blocks ⇒ large enough so each SM has at least one block to process
  - Ideally each SM should have several thread blocks; some threads block may be stalled by a barrier (`__syncthreads`)
- Number of threads per blocks: suject to hardware resource limitations
  - should be a multiple of warpsize to avoid computation waste, better for memory coalescence
  - initial choice should be 128 or 256 threads per CTA
  - prefer a few smaller blocks than a large one, better for hiding global memory latency (more memory transaction in flight)

Cuda Occupancy - modern GPU

A very pedagogical and insightful presentation from ModernGPU by Sean Baxter

Others important features for reaching max performance:

- **global memory coalescence**
- **avoid branch divergence:** might need to restructure/refactor code; aloso branch diverge is a source of potential load imbalance (e.g. recursive call)
- **avoid too large kernels** (e.g. kernel which uses too much registers ⇒ register spilling ; solution could be to split kernel into 2 or more smlaller cuda kernels

Performance tuning thoughts

- **Threads are free**
  - Keep threads short and balanced
  - HW can (must) use LOTs of threads (several to 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 3.0/3.5 (Kepler)**

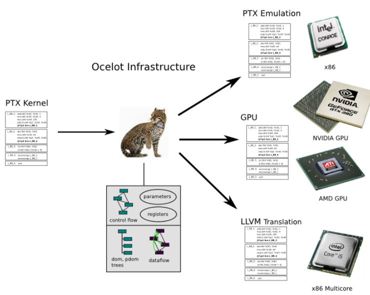
CUDA HW version	Features
1.0	basic CUDA execution model
1.3	double precision, improved memory accesses, atomics
2.0 (Fermi)	Caches (L1, L2), FMAD, 3D grids, ECC, P2P (unified address space), funtion pointers, recurs
3.5 (Kepler GK110) [1]	Dynamics parallelism, object linking, GPU Direct RemoteDMA, new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

More on PTX

- PTX is a virtual instruction set for a generic GPU
- gpuocelot is a dynamic compilation framework providing various backends for CUDA programs
- Ocelot currently allows CUDA programs to be executed on NVIDIA GPUs, AMD GPUs, and x86-CPUs at full speed without recompilation



More on PTX

- PTX is a virtual instruction set for a generic GPU
- emulator, profiling tools, dynamic instrumentation, runtime (mapping and thread scheduling), optimization
- Add GPU support in your favorite devel language: CUDA LLVM Compiler. Promising example: NumbaPro - integration python/Cuda using py-LLVM

Performance and Analysis tools for GPU computing

- Have a look at slides at SC11, by Allen Malony
- NVIDIA CUDA Profiling Tools Interface (CUPTI from CUDA toolkit) (e.g. used internally by `nvvp`)
- PAPI (Performance Application Programming Interface), a generic tool for accessing performance counter hardware found in most major microprocessors.
- PAPI CUDA component, use CUPTI -> GPU hardware performance counters through
- TAU: automatic source code instrumentation

PAPI for CUDA

- Installation on Ubuntu 14.04 (driver 340.29; toolkit 6.5-14)
- Make sure to have environment variable `LD_LIBRARY_PATH` set to have location of the the CUPTI library (from the NVIDIA driver)
- use PAPI release 5.4.0

PAPI for CUDA

- 1. **untar PAPI sources**
  2. **configure CUDA component:**
     ```
     cd src/component/cuda
     ./configure --with-cuda-dir=/usr/local/cuda-6.5
     --with-cupti-dir=/usr/local/cuda-6.5/extras/CUPTI
     ```
  3. **configure PAPI:**
     ```
     cd ../..
     ./configure --with-components=cuda
     --prefix=/opt/papi/5.4.0
     ```
  4. Use command line tool : `papi_native_avail` to get the list of PAPI native events for GPU
  5. have a look at test example in `src/components/cuda/tests`, build and run after choosing a valid event; e.g. for my desktop machine: `cuda:::GeForce_GTX_660:domain_d:warps_launched`

Perf and Analysis tools for GPU : PDT - TAU

- **Install PDT** (latest is 3.20, November 2013)
- configure line used on this school machines (Ubuntu 12.04):
  `./configure -GNU -prefix=/opt/pdt/3.20; make; make install`
- **Install TAU** (latest version is 2.24, November 2014) (don't use 2.22, Cupti support has bugs)
- **configure line used :**

```
./configure -prefix=/opt/tau/2.24 -cc=gcc -c++=g++
-fortran=gnu -pdt=/opt/pdt/3.20
-cuda=/usr/local/cuda-6.5
-cupti=/usr/local/cuda-6.5/extras/CUPTI
-papi=/opt/papi/5.4.0 -iowrapper -pdt_c++=g++
```

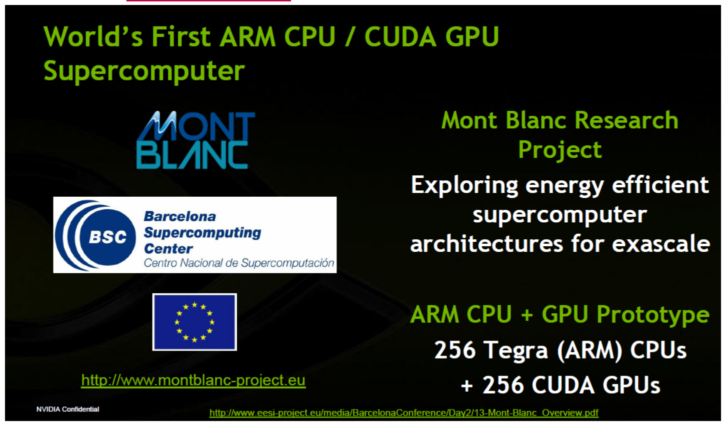
- Note that TAU can be configured multiple times to include different compilers. Here option `-fortran=pgi` will enable profiling/tracing in PGI/CUDA/fortran code.

Perf and Analysis tools for GPU : PDT - TAU

- **Quick start Hands-On using benchmark SHOC** example Stencil2D
- use `module load tau_gpu` to setup TAU environment for GPU
- **Quick Start for profiling** (go into SHOC example exe):
  - `tau_exec -T serial -cuda ./Stencil2D`
  - `ls profile*; pprof | less; paraprof Stencil2D.ppk`
- **Quick start for tracing**
  - repeat above with `export TAU_TRACE=1`
  - execute `tau_multimerge`; this create tau.trc and tau.edf
  - convert tracing file into slog2 format: `tau2slog2 tau.trc tau.edf -o Stencil2D.slog2`
  - visualization: `jumpshot Stencil2D.slog2`
- CUPTI / performance counters: use command `tau_cupti_avail` to print available perf counters for your platform; then to use, for example
  `export TAU_METRICS=CUDA.GeForce_GT_430.domain_a.inst_issued2_0`
- Don't forget to look at slides from A. Malony at SC11.

Low-power supercomputing ? When ?

Waiting for Denver project: ARM 64bit-ISA + on-chip GPU (2013 ?)



source: Ian Buck, NVIDIA, SC11, Seattle, Nov 2011.

Low-power supercomputing ? When ?

Waiting for Denver project: ARM 64bit-ISA + on-chip GPU (2013 ?)

CUDA for ARM Development Kit

CUDA GPU   Tegra ARM CPU

Research development board
- Quad-core ARM based NVIDIA Tegra 3 processor
- NVIDIA CUDA GPU
- Gigabit Ethernet

CUDA software development kit

SECO Hardware Development Kit

Available: 1H 2012

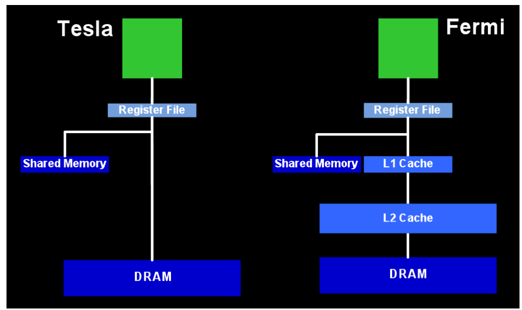
http://www.secoqseven.com/en/item/secocq7-mxm/

NVIDIA Confidential

http://www.hpcwire.com/hpcwire/2012-11-06/bsc_installs_new_arm-based_prototype.html

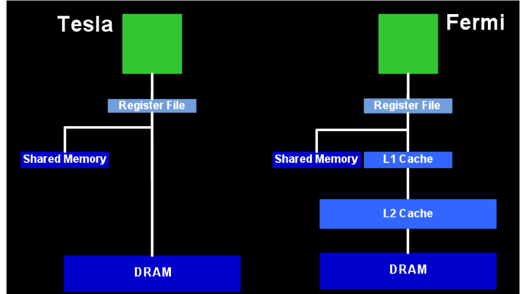
GPU computing - Fermi and L1/L2 cache

- global memory R/W are *pipelined*
- **global memory latency** : qq 100 cycles d'horloge
- **shared memory / L1 cache latency** : 10-20 cycles



GPU computing - Fermi and L1/L2 cache

- L1 cache: ressources shared with *shared memory* !
- L1 cache used for reading local variables, if not enough registers
- L2 cache: for global memory
- **gain:**
  - Caching captures **locality**, amplifies bandwidth, **reduces latency**
  - Caching aids **irregular or unpredictable accesses**
  - ⇒ better performances for algorithms with complex memory access *patterns*



Nvidia and OpenPOWER

POWER8 CAPI Coherence Attach Processor Interface

Virtual Addressing
- Accelerator can work with same memory addresses that the processors use
- Pointers de-referenced same as the host application
- Removes OS & device driver overhead

Hardware Managed Cache Coherence
- Enables the accelerator to participate in "Locks" as a normal thread Lowers Latency over IO communication model

POWER8
Coherence Bus
CAPP

PCI Gen 3
Transport for encapsulated messages

Custom Hardware Application
PSL
FPGA or ASIC

Customizable Hardware Application Accelerator
- Specific system SW, middleware, or user application
- Written to durable interface provided by PSL

Processor Service Layer (PSL)
- Present robust, durable interfaces to applications
- Offload complexity / content from CAPP

2013 International Business Machines Corporation

IBM

references:

http://openpowerfoundation.org/ https://www.olcf.ornl.gov/summit/nvidia-ws-2014/08-koehler-openpower.pdf

http://wccftech.com/ibm-power8-processor-architecture-detailed/

Nvidia and OpenPOWER

Power Systems IBM

Coherent Accelerator Processor Interface (CAPI) Overview

CAPP PCIe
POWER8 Processor
CAPI
FPGA
IBM Supplied POWER Service Layer
Function 0 Function 1 Function 2 Function n

Typical I/O Model Flow

DD Call → Copy or Pin Source Data → MMIO Notify Accelerator → Acceleration → Poll / Int Completion → Copy or Unpin Result Data → Ret. From DD Completion

Flow with a Coherent Model

Shared Mem. Notify Accelerator → Acceleration → Shared Memory Completion

Advantages of Coherent Attachment Over I/O Attachment

- Virtual Addressing & Data Caching
  - Shared Memory
  - Lower latency for highly referenced data
- Easier, Natural Programming Model
  - Traditional thread level programming
  - Long latency of I/O typically requires restructuring of application
- Enables Apps Not Possible on I/O
  - Pointer chasing, etc...

© 2014 IBM Corporation
IBM Confidential Until Announce on April 23, 2014 at 1 PM Eastern
1

references:
http://openpowerfoundation.org/ https://www.olcf.ornl.gov/summit/nvidia-ws-2014/08-koehler-openpower.pdf
http://wccftech.com/ibm-power8-processor-architecture-detailed/

Nvidia and OpenPOWER

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

Home Overview Call for Proposals Timeline Transitioning Media

SUMMIT
THE NEXT PEAK IN HPC

Summit is the next leap in leadership-class computing systems for open science. With Summit we will be able to address, with greater complexity and higher fidelity, questions concerning who we are, our place on earth, and in our universe.

Summit will deliver more than five times the computational performance of Titan's 18,688 nodes, using only approximately 3,400 nodes when it arrives in 2017. Like Titan, Summit will have a hybrid architecture, and each node will contain multiple IBM POWER9 CPUs and NVIDIA Volta GPUs all connected together with NVIDIA's high-speed NVLink. Each node will have over half a terabyte of coherent memory (high bandwidth memory + DDR4) addressable by all CPUs and GPUs plus 800GB of non-volatile RAM that can be used as a burst buffer or as extended memory. To provide a high rate of I/O throughput, the nodes will be connected in a non-blocking fat-tree using a dual-rail Mellanox EDR InfiniBand interconnect.

Upon completion, Summit will allow researchers in all fields of science unprecedented access to solving some of the world's most pressing challenges.

Summit Fact Sheet

TITAN VS SUMMIT

IBM Mellanox NVIDIA

Compute System Comparision

ATTRIBUTE	TITAN	SUMMIT
Compute Nodes	18,688	~3,400
Processor	(1) 16-core AMD Opteron per node	(Multiple) IBM POWER 9s per node
Accelerator	(1) NVIDIA Kepler K20x per node	(Multiple) NVIDIA Volta GPUs per node
Memory per node	32GB (DDR3)	>512GB (HBM+DDR4)
CPU-GPU Interconnect	PCI Gen2	NVLINK (5-12x PCIe3)
System Interconnect	Gemini	Dual Rail EDR-IB (23 GB/s)
Peak Power Consumption	9 MW	10 MW

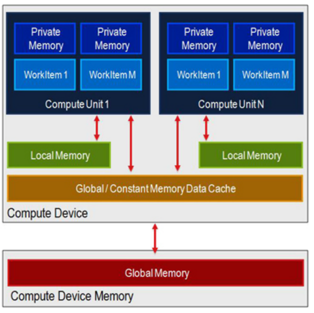
Summit FAQs

references:
http://openpowerfoundation.org/ https://www.olcf.ornl.gov/summit/
nvidia-ws-2014/08-koehler-openpower.pdf
http://wccftech.com/ibm-power8-processor-architecture-detailed/

GPU computing - CUDA

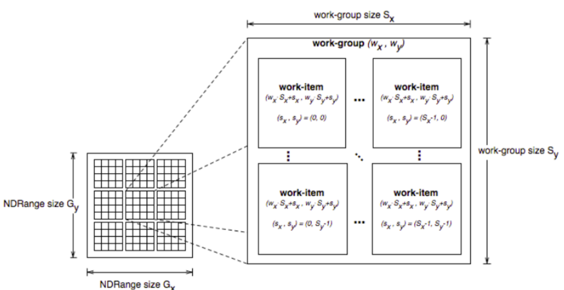
- Nvidia Performance Primitives
- NVIDIA NPP is a library of functions for performing CUDA accelerated processing
- The initial set of functionality in the library focuses on imaging and video processing

GPU computing - OpenCL



- OpenCL sur Wikipedia // Introduction OpenCL
- standard `http://www.khronos.org`, version 1.0 (12/2008)
- aim: programming model for GPU (Nvidia/ATI), multicore CPU and CELL: *Data and task parallel compute model*
- OpenCL programming model use most of the abstract concepts of CUDA

GPU computing - OpenCL



- OpenCL sur Wikipedia // Introduction OpenCL
- standard `http://www.khronos.org`, version 1.0 (12/2008)
- aim: programming model for GPU (Nvidia/ATI), multicore CPU and CELL: *Data and task parallel compute model*
- OpenCL programming model use most of the abstract concepts of CUDA

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)

- convert CUDA program to OpenCL :
  http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-ATI-Stream-v2.0-Beta.aspx#four
- CUDA used to be ahead OpenCL; CUDA adopted rapidly, people will start to move to OpenCL

Install CUDA / OpenCL

- **You must have a compatible with CUDA**. Personal example : GeForce GTX 560 Ti (hardware 2.0)
- **Comparison of Nvidia GPU features:**
  `http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units`
- **example almost up-to-date system:**
  - OS: 14.04 - x86_64 (October 2015)
  - CUDA version: 7.5.18
  - Since 2013, Nvidia driver+toolkit+sdk available as Deb or RPM package
- **Tips for Ubuntu >= 14.04**
  - `sudo dpkg -i cuda-repo-ubuntu1404_7.5-0_amd64.deb`; this will setup nvidia repository (write file `/etc/apt/sources.list.d/cuda.list`)
  - then you can install tools by `sudo apt-get install cuda`

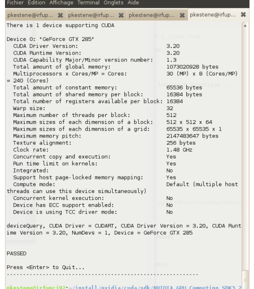
Install CUDA / OpenCL

- **Nvidia Linux kernel driver**: version 352.79
  - Deb package `nvidia-current`; builds kernel module `nvidia_current.ko`
  - Installs CUDA Driver API library `libcuda.so`, `libnvidia-opencl.so`, etc ...
  - misc: Nvidia codec libraries: `nvcuvid`, `nvcuvenc`
- **compiling toolchain (aka *toolkit*)**:
  - provides `nvcc` CUDA/C compiler
  - CUDA RunTime API library `libcudart.so`
  - Profiling tool: `nvvp` (can be used inside `nsight`)
  - IDE - custom version of Eclipse named `nsight`
  - Scientific libraries: cuBlas, cuFFT, cuRand, cuSparse, etc...
  - CUDA Documentation

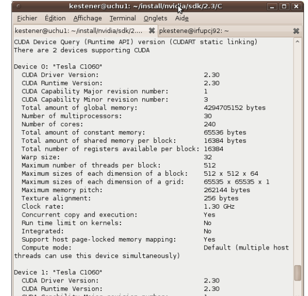
CUDA compatible GPU

- **Is the Nvidia graphics driver loaded ?** Have a look at proc filesystem: `/proc/driver/nvidia`
- Monitoring nvidia GPU; `nvidia-smi`
- What are the GPU features ? Run CUDA sample *deviceQuery*
- header `cuda_runtime_api.h` : cudaGetDeviceCount, cudaGetDeviceProperties
- What is the installed driver version ? `cat /proc/driver/nvidia/version`; `nvidia-smi`

GetForce GTX 285



Tesla C1060



SDK CUDA

- Most SDK samples follow roughly the same *template* (see project *template* in `0_Simple` subdir) :
  - `project_name.cu` : contains *main* and entry point to GPU computation (a call to a CUDA kernel)
  - `project_name_kernel.cu` : definition of some CUDA kernel
  - `project_name_gold.c` : native CPU version, for comparison or performance benchmark
- A fem important examples for *pedagogical reasons*
  - `transpose` : efficient use of device memory bandwidth, memory coalescence, shared memory, bank conflict
  - `reduction` : efficient use of device memory bandwidth
- One can mix CUDA code for device in a regular C/CPP source file, provided it is protected by macro `__CUDACC__`

Summary

1. GPU computing: Architectures, Parallelism and Moore law
   - Why multi-core ?
   - Understanding hardware, better at optimization
   - What's a thread ?
   - History
2. CUDA Programing model
   - Why shoud we use GPU ?
   - Hardware architecture / programming model
   - CUDA : optimisation / perf measurement / analysis tools
   - GPU computing : perspectives / Install CUDA
3. Other languages: CUDA/Fortran, PyCuda, CUDA/MPI
4. Extra slides

CUDA / Fortran: what should I do ?

- There are multiple ways to use CUDA in Fortran code
- 1. Use Fortran on host and CUDA/C on GPU device
    - a nice way to do that, it to use the `ISO_C_Bindings` module from Fortran2003
    - pros: multiple Fortran compilers can do that (gfortran >= 4.4, Intel, PGI, etc...)
    - cons: `ISO_C_Bindings` require some good Fortran knowledge (pointers, etc...); a lot of plumbery code !
    - http://sourceforge.net/projects/fortcuda/ : `iso_c_bindings` of CUDA runtime and driver API
    - example of use at http://www-irma.u-strasbg.fr/irmawiki/index.php/Call_CUDA_from_Fo
  2. Use Fortran for both CPU and GPU code: PGI compiler
  3. Use a compilation directive tool (PGI accelerator or CAPS HMPP)

CUDA / Fortran: what should I do ?

- There are multiple ways to use CUDA in Fortran code
- 1. Use Fortran on host and CUDA/C on GPU device
  2. Use Fortran for both CPU and GPU code: PGI compiler
     - Collaboration Nvidia/PGI
     - Most features have a 1-to-1 mapping to CUDA/C
     - Nice integration with Fortran arrays, greatly simplifies Host/Device memory transfert
     - Much easier and pleasant to use (regarding the CUDA runtime API)
     - can still use `iso_c_bindings` to interface your custom science/application C/C++ packages
  3. Use a compilation directive tool (PGI accelerator or CAPS HMPP)

CUDA / Fortran: what should I do ?

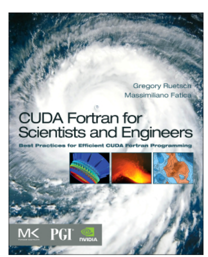
- There are multiple ways to use CUDA in Fortran code
- 1. Use Fortran on host and CUDA/C on GPU device
  2. Use Fortran for both CPU and GPU code: PGI compiler
  3. Use a compilation directive tool (PGI accelerator or CAPS HMPP)
     - *à la* OpenMP programing model
     - PGI accelerator (same compilation toolchain as for Cuda/Fortran above); can be used for C/C++ code also
     - CAPS HMPP (will be covered at the end of the week on Friday)

PGI CUDA/Fortran : mapping to CUDA/C

- Material: show slides from Justin Luitjens, at SC11 (tutorial M07) and S3050-Intro-to-CUDA-Fortran.pdf
- Show **CUDA/Fortran helloworld** with `ISO_C_Bindings`
- Show **CUDA/Fortran helloworld** with PGI's CUDA/Fortran
- Show SAXPY examples (with and without cuBLAS)

Hands-On sessions

- Use code from Ruetsch and Fatica's book, section 2.2 *Instruction, Bandwidth and Latency bound kernels*
- book ***CUDA Fortran for Scientists and Engineers*** by G. Ruetsch and M. Fatica, Morgan Kaufmann, 2013



PGI CUDA/Fortran : interesting links / on-line material

- Slides from Michael Wolfe (PGI) at SC11, Seattle
- Tutorial slides from conference GTC2013:
  - S3050: **Introduction to CUDA Fortran**
  - S3448: **CUDA Fortran 2013 Support for Kepler and CUDA 5**
- PGI compiler website and user's guide
- Cédric Castagnède slides on PGI CUDA/fortran development tools (in French).
- Ladislav Hanyk, GUCAS Summer School, Beijing, June-July 2011: Solving PDEs with PGI CUDA Fortran
- Oliver Mangold, HLRS Parallel Programming Workshop, some basic CUDA/C and CUDA/Fortran exercices

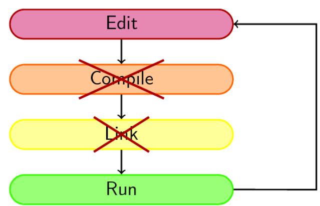
CUDA - Python bindings

PyCUDA

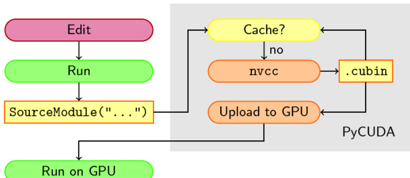
- CPU development tools for sequential program have a long history, provide high productivity / efficiency
- GPU development tools are in their infancy !
- PyCuda wraps the CUDA driver API into the Python language
- reference : Andeas Klöckner, http://mathema.tician.de/software/pycuda

PyCuda workflow

- **GPUs are everything that scripting languages are not**
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput



- **Python + CUDA = PyCUDA:**
  - manage resources automatically
  - provide a *glue-language* for lower-level building blocks
  - Very nice integration with `numpy` (a must in python scientific computing)



PyCUDA installation

- download the package from
  `http://pypi.python.org/pypi/pycuda`
- untar and configure (set the CUDA toolkit location) :
  `./configure.py -cuda-root=/usr/local/cuda32/ -cuda-enable-gl`
- build package:
  `make`
- install package system-wide: `sudo make install`
  This last step installs the package in `/usr/local/lib/python2.6/dist-packages/pycuda-0.94.2-py2.6-linux-x86_64.egg`

PyCuda - Code walkthrough

PyCUDA code `demo.py` from examples

Compute multiplication by a scalar, elementwise

- **Initialization and data array declaration:**

```
# acces to CUDA driver API
import pycuda.driver as cuda
# initialize, create context
import pycuda.autoinit
from pycuda.compiler import SourceModule

import numpy
a = numpy.random.randn(4,4).astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.size * a.dtype.itemsize)
cuda.memcpy_htod(a_gpu, a)
```

- **Kernel declaration and execution**
- **Retrieve result on CPU and compare**

PyCuda - Code walkthrough

PyCUDA code `demo.py` from examples

Compute multiplication by a scalar, elementwise

- **Initialization and data array declaration:**
- **Kernel declaration and execution**

```
# define CUDA kernel as a python string
# compile and load into GPU device
mod = SourceModule("""
    __global__ void doublify(float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
    """)
# get a handle
func = mod.get_function("doublify")
# launch GPU computation
func(a_gpu, block=(4,4,1))
```

- **Retrieve result on CPU and compare**

PyCuda - Code walkthrough

PyCUDA code `demo.py` from examples

Compute multiplication by a scalar, elementwise

- **Initialization and data array declaration:**
- **Kernel declaration and execution**
- **Retrieve result on CPU and compare**

```
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print "original array:"
print a
print "doubled with kernel:"
print a_doubled
```

PyCuda package: GPUArray

- data array abstraction : `gpuarray`
  import pycuda.gpuarray as gpuarray
- `gpuarray` is a `numpy.ndarray` work-alike that stores its data and performs its computations on the compute device
- hide host to device memory copy:
  a_gpu = gpuarray.to_gpu(numpy.random.randn(4,4).
  astype(numpy.float32))
- Some predefined CUDA kernel implemented as gpuarray instance methods : reductions (`min`,`max`,`sum`,...), elementwise operators (fabs, trigonometric, ...)
- pyfft package (execute FFT on GPU with python) :
  `http://pypi.python.org/pypi/pyfft`

Other ways to use python for GPU computing

- `cython`: a nice to wrap existing C/C++/Cuda code
  see code used in hands-on :
  `https://github.com/pkestene/npcuda-example`
- cudamat uses ctypes (a python module for foreign function library calls / basilcally C, no C++).
  `cudamat` only supports `float32` for now.
- CUDA SciKit (uses PyCUDA to provide a Numpy compatible interface).
- TO CHECK: `https://github.com/ashwinsrnth/petsc-pycuda`

PyCuda - Real life example

Lattice Boltzmann simulations : **Sailfish**
http://sailfish.us.edu.pl/
TP:

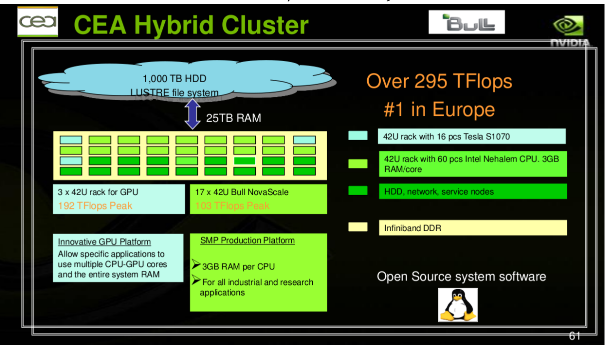
- Have a look at examples in pycuda distribution
- Grab the CUDA kernel code for solving the heat equation, and implement a pycuda version + a Matplotlib GUI
- Play with sailfish examples
- Implement a pyCUDA version of the CPU version rayleighbenard.py (simple python LBM solver)
- Have a look at reikna (this is a pyCUDA/pyOpenCL interface for computing FFT on GPU among others);
- future ?: copperhead : a data-parallel subset of Python, code dynamically compiled for target platform

CUDA -MPI coupling/integration

- **CUDA** and **MPI** are almost **orthogonal**: good !
- MPI: distributed parallel processing
- CUDA: co-processor, CPU accelerator
- **NVIDIA System Management Interface** (from Nvidia driver):
  - `nvidia-smi -s` : show the current rules for COMPUTE applications
  - `nvidia-smi -q` : query information (utilization rates, memory usage, etc...)
- nvidia-smi compute modes (on Tesla hardware running linux)
  - `Default`: Multiple host threads can use the device at the same time
  - `Exclusive`: Only one host thread can use the device at any given time.
  - `Prohibited`: No host thread can use the device.
  - When using the `Default` mode (this a system admin choice), the programmer **NEEDS** to call `cudaSetDevice(devId)` to explicit which GPU device is associated to which MPI process
  - example GPU device choice rule: `cudaGetDeviceCount(&count); int devId = myRank % count;`
- 225_GTC2010.pdf slides

CEA hybrid CPU/GPU Cluster, 2009 - Titane

Titane: ~ 100 CPU-TFLOPS + ~ 200 GPU-TFLOPS (Tesla S1070, hw 1.3)

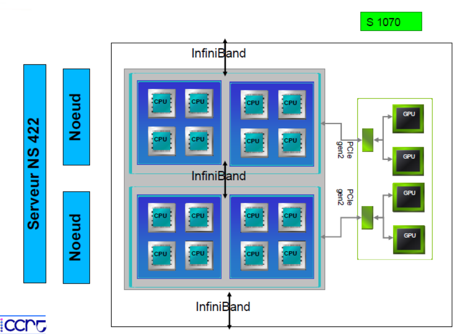


- `cudaGetDeviceCount(&count);` returns 2 because a Titane node only *sees* half a Tesla-S1070, i.e. 2 GPU devices.

http://www-ccrt.cea.fr/fr/moyen_de_calcul/titane.htm

CEA hybrid CPU/GPU Cluster, 2009 - Titane

Titane: ~ 100 CPU-TFLOPS + ~ 200 GPU-TFLOPS (Tesla S1070, hw 1.3)



- `cudaGetDeviceCount(&count);` returns 2 because a Titane node only *sees* half a Tesla-S1070, i.e. 2 GPU devices.

http://www-ccrt.cea.fr/fr/moyen_de_calcul/titane.htm

CUDA MPI: software development issues

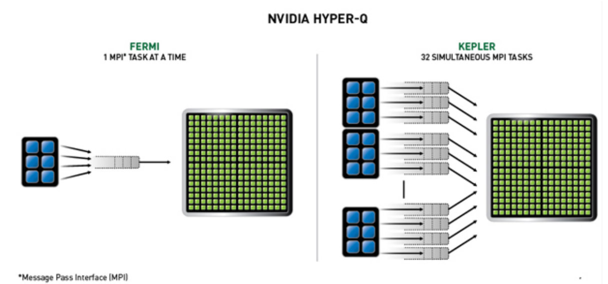
- easier to use the `nvcc` compiler driver than `mpicc` to integrate CUDA into a MPI program.
- Just need to pass the MPI `CFLAGS` and `LDFLAGS`
- **integration with a build system:** an example autotools-based project will be provided during hands-on session
- Example hands-on: try to parallelize the heat-solver using MPI over 2 nodes (only 1 border for MPI communications). Start with CPU version, and then the GPU version.
- potentially interesting tools:
  - Global Memory for Accelerator GMAC, is a user-level library that implements an Asymmetric Distributed Shared Memory model

MPI+CUDA strategies

- **One MPI process per GPU**: easy but *wastes* other CPU cores of the node
- **Several MPI processes, only one uses GPU**: load balancing problem (one MPI proc faster than the others)
- **Several MPI process share a GPU**: each MPI proc has its own GPU context, no com possible; divide memory resources, ...
- small reference: Parallel programming with CUDA and MPI

MPI+CUDA strategies

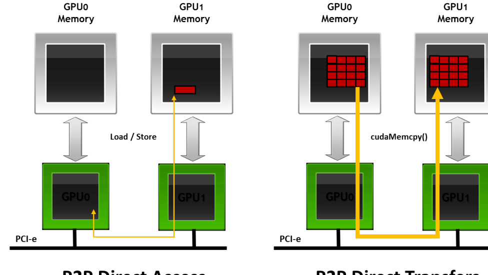
- with Kepler's Hyper-Q hardware feature, multiple MPI proc can send work to GPU through different queues see Nvidia HyperQ



- small reference: Parallel programming with CUDA and MPI

GPUDirect technology

- **Intra-node GPU-GPU communication**, GPUDirect P2P; see SDK

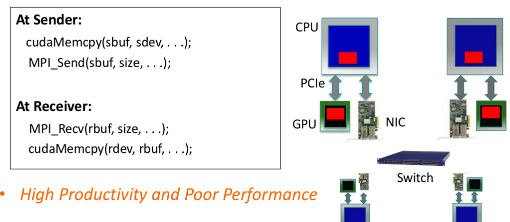


  example `simpleP2P`

- **Inter-node communication** (CPU-GPU memory transfert + MPI comm): see next slide

MPI+CUDA: inter-node communications

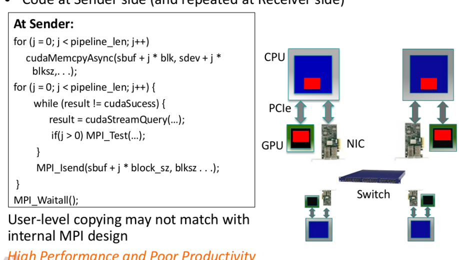
- **naïve approach:** use synchronous `cudaMemcpy` + `MPI_Send`/MPI_Recv it is OK for applications not MPI-bounded



- **advanced approach:** overlap CPU-GPU memory transfert with MPI communications; need to rewrite code to divide memory copies and MPI message into chunks and use `cudaMemcpyAsync` + `MPI_Isend`

MPI+CUDA: inter-node communications

- **naïve approach:** use synchronous `cudaMemcpy` + `MPI_Send`/`MPI_Recv` it is OK for applications not MPI-bounded
- **advanced approach:** overlap CPU-GPU memory transfert with MPI communications; need to rewrite code to divide memory copies and MPI message into chunks and use `cudaMemcpyAsync` + `MPI_Isend`

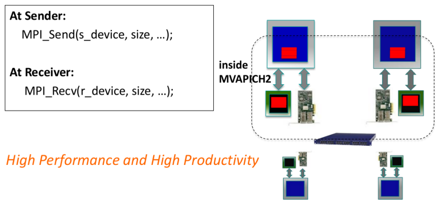


MPI+CUDA: inter-node communications

- **naïve approach:** use synchronous `cudaMemcpy` + `MPI_Send`/`MPI_Recv` it is OK for applications not MPI-bounded
- **advanced approach:** overlap CPU-GPU memory transfert with MPI communications; need to rewrite code to divide memory copies and MPI message into chunks and use `cudaMemcpyAsync` + `MPI_Isend`
- **Use latest devel MPI implementation like OpenMPI / MVAPICH2:** should do the CPU-GPU / MPI comm overlap for you !!! You could use a single/simple `MPI_Send` with reference to the local and the remote

Sample Code – MVAPICH2-GPU

- MVAPICH2-GPU: standard MPI interfaces used



GPU memory pointer !

MPI+CUDA: inter-node communications

- **naïve approach:** use synchronous `cudaMemcpy` + `MPI_Send`/MPI_Recv it is OK for applications not MPI-bounded
- **advanced approach:** overlap CPU-GPU memory transfert with MPI communications; need to rewrite code to divide memory copies and MPI message into chunks and use `cudaMemcpyAsync` + `MPI_Isend`
  reference: ofa_mar12-accelerator.pdf

CUDA / MPI / GPUDirect online material

- slides by S. von Alfthan, Parallel programming with CUDA and MPI (from another PRACE training school)
- DirectGPU_CSCS_alam.pdf
- ofa_mar12-accelerator.pdf

Summary

1. GPU computing: Architectures, Parallelism and Moore law
   - Why multi-core ?
   - Understanding hardware, better at optimization
   - What's a thread ?
   - History
2. CUDA Programing model
   - Why shoud we use GPU ?
   - Hardware architecture / programming model
   - CUDA : optimisation / perf measurement / analysis tools
   - GPU computing : perspectives / Install CUDA
3. Other languages: CUDA/Fortran, PyCuda, CUDA/MPI
4. Extra slides

Parallel patterns for OpenAcc / OpenMP on accelerators

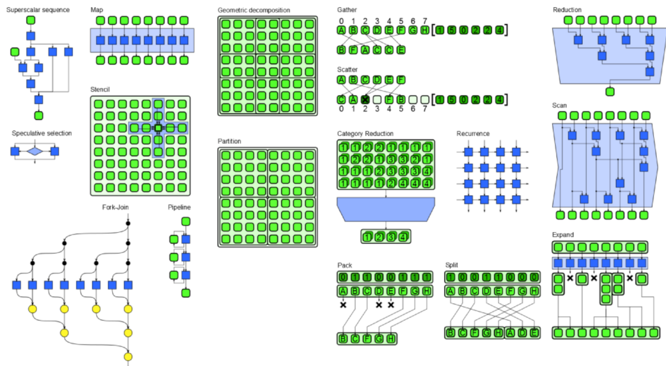
- **pattern : a basic structural entity of an algorithm**
- book
  Structured Parallel Programming: Patterns for Efficient Computation
- implementation: Intel TBB, and many others
- OpenMP/OpenAcc for GPU/XeonPhi: pattern-based comparison: map, stencil, reduce, scan, fork-join, superscalar sequence, parallel update

reference:

A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing

Parallel patterns for OpenAcc / OpenMP on accelerators

Parallel Patterns: Overview



reference: Structured Parallel Programming with Patterns, SC13 tutorial, by M. Hebenstreilt, J. reinders, A. Robison, M. McCool

Future of accelerator programming

- **passive libraries:** a collection of subroutines
- **active libraires:** take an active role in compilation (specialize algorithms, tune themselves for target architecture).

Library	CUDA	OpenCL	Other	Type
Thrust	X		OMP, TBB	header
Bolt		X	TBB, DX11	link
VexCL	X	X		header
Boost.Compute		X		header
C++ AMP		X	DX11	compiler
SyCL		X		compiler
ViennaCL	X	X	OMP	header
SkePU	X	X	OMP, seq	header
SkelCL		X		link
HPL		X		link
CLOGS		X		link
ArrayFire	X	X		link
CLOGS		X		link
hemi	X			header
MTL4	X			header
Kokkos	X		OMP, PTH	link
Aura	X	X		header

reference:

The Future of Accelerator Programming in C++, S. Schaetz, May 2014

Future of accelerator programming

- **Coordination:**
  - concurrency: asynchronicity, data dependency graph
  - memory management (explicit ? implicit ?) / memory layout (SoA, AoS, unordered map, Morton-index, ...)
- **Computation:**
  - parallel primitives
  - custom accelerator functions
  - numerical analysis
  - performance portability
  - kernel-space exploration / tuning

reference:
The Future of Accelerator Programming in C++, S. Schaetz, May 2014

Future of accelerator programming: Kokkos among other

Kokkos' Layered Libraries

- **Core**
  - Multidimensional arrays and subarrays in **memory spaces**
  - parallel_for, parallel_reduce, parallel_scan on **execution spaces**
  - Atomic operations: compare-and-swap, add, bitwise-or, bitwise-and
- **Containers**
  - UnorderedMap – fast lookup and **thread scalable insert / delete**
  - Vector – subset of std::vector functionality to ease porting
  - Compress Row Storage (CRS) graph
  - Host mirrored & synchronized device resident arrays
- **Sparse Linear Algebra**
  - Sparse matrices and linear algebra operations
  - Wrappers for vendors' libraries
  - Portability layer for Trilinos manycore solvers

4

reference: slides by Edwards, Trott, Sunderland (SANDIA) at GTC2014

Kokkos, a Manycore Device Performance Portability Library for C++ HPC Applications

Future of accelerator programming: Kokkos among other

- kokkos multi-dimensional array
  map multi-index $(i, j, k, ...) \Longleftrightarrow$ memory location in a **memory space** [2]
- Kokkos will choose a default memory layout adapted to the target device
- Decouple logical index $(i, j, k, ...)$ from actual memory layout

[2] In the same line of idea, see chapter 28 of book *High Performance Parallelism Pearls*, Morton order improve performance

Future of accelerator programming: Kokkos among other

MiniMD used to bench thread-scalable algorithm before integrating them in LAMMPS (2014)

MiniMD Performance

Lennard Jones force model using atom neighbor list

- **Solve Newton's equations for *N* particles**
- **Simple Lennard Jones force model:** $F_i = \sum_{j,\, r_{ij} < r_{cut}} 6\varepsilon \left[\left(\frac{\varsigma}{r_{ij}}\right)^7 - 2\left(\frac{\varsigma}{r_{ij}}\right)^{13} \right]$
- **Use atom neighbor list to avoid N^2 computations**

```
pos_i = pos(i);
for( jj = 0; jj < num_neighbors(i); jj++) {
  j = neighbors(i,jj);
  r_ij = pos_i - pos(j); //random read 3 floats
  if ( |r_ij| < r_cut )
    f_i += 6*e*( (s/r_ij)^7 - 2*(s/r_ij)^13 )
}
f(i) = f_i;
```

- **Moderately compute bound computational kernel**
- **On average 77 neighbors with 55 inside of the cutoff radius**

17

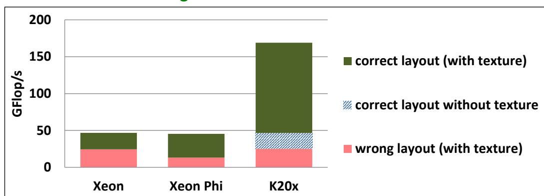
Future of accelerator programming: Kokkos among other

MiniMD used to bench thread-scalable algorithm before integrating them in LAMMPS (2014)

MiniMD Performance

Lennard Jones (LJ) force model using atom neighbor list

- Test Problem (#Atoms = 864k, ~77 neighbors/atom)
  - Neighbor list array with correct vs. wrong layout
    - Different layout between CPU and GPU
  - Random read of neighbor coordinate via GPU texture fetch

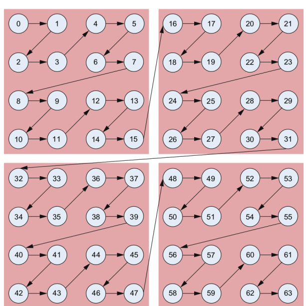


- Large loss in performance with wrong layout
  - Even when using GPU texture fetch
  - Kokkos, by default, selects the correct layout

Complex memory layout for performance

chapter 28 of book ***High Performance Parallelism Pearls***, Morton order improve performance, by **Kerry Evans (INTEL)**; mesure on Xeon and XeonPhi

transpose, **dense matrix multiplication** on Xeon and XeonPhi



dim	naïve 32 thr	Morton 32 thr	speedup
256	0.1	0.188	0.53
512	0.05	0.169	0.29
1024	0.62	0.366	1.7
2048	2.89	0.871	3.3
4096	211	7.92	26.6
8192	1850	60.2	30.7
16384	13695	473	29.0
32768	Too long	3989	--

dim	naïve 244 thr	Morton 244 thr	speedup
256	0.02	0.003	6.67
512	0.26	0.008	32.5
1024	1.78	0.046	38.7
2048	12.87	0.4	32.18
4096	105.5	2.9	36.38
8192	105.5	23	36.74
16384	6597	181	36.4
32768	Too long	1468	--

CUDA software developpement tools

- **Latest news (Nov. 2015) for heterogenous computing**
  - AMD announces a new C++ compiler / Linux Driver and new Heterogeneous-computing Interface for Programmers (HiP): a tool to convert CUDA runtime API into C++ (that can be compiled for AMD GPUs)
    https://community.amd.com/community/amd-business/blog/2015/11/16/a-defining-mom
  - Google is preparing an open-source GPGPU compiler on top of CLANG
    http://llvm.org/devmtg/2015-10/slides/Wu-OptimizingLLVMforGPGPU.pdf