



Solving PDEs with PGI CUDA Fortran

Part 1: Introduction to NVIDIA hardware and CUDA architecture

Ladislav Hanyk
Charles University in Prague, Faculty of Mathematics and Physics
Czech Republic

Outline

Multiprocessors and compute capability.
Floating-point arithmetic, Gflops.
CUDA programming model: threads, blocks and grids, warps and kernels.
Memory hierarchy.
Compute-capability limits. memory coalescing.
A kernel source code.

Accelerators

coprocessors for offloading compute-intensive processes

GPUs (graphics processing units)

- coprocessors specialized to accelerate graphics (esp., games) but evolved recently to serve for general-purpose (**GP**) GPU computing
- massively parallel: collect many (hundreds) processors (**cores**)
- appropriate algorithms may get speedups of 10x-100x, but redesign of applications is necessary



Accelerators

NVIDIA

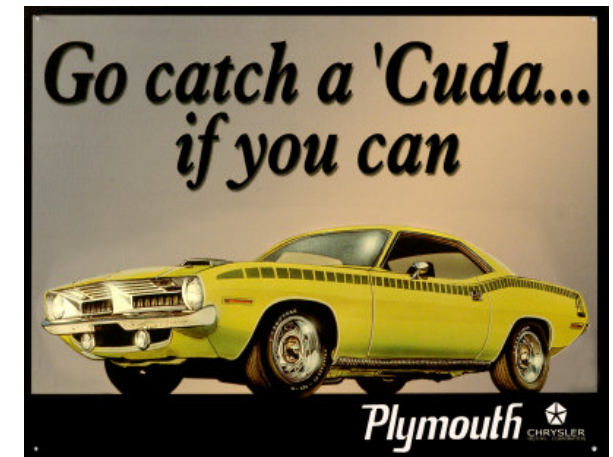
- CUDA** – the most popular GP GPU parallel programming model today
- from notebooks and personal desktops to high performance computing (HPC)
 - a **host** (CPU) offloads a suitable part of a process (a **kernel**) to the **device** (GPU)
 - the device with many cores runs the kernel concurrently by many subprocesses (**threads**)
 - two-level hardware parallelism on a device:
 - SIMD** (single-instruction multiple-data) and **MIMD** (multiple-instruction multiple-data)
 - a programming model reflects the hardware parallelism by grouping the threads into **blocks** and **grids**

nvcc and **CUDA API** (Application Programming Interface)

- C/C++ based proprietary compiler and library provided by NVIDIA
- many third-party tools on top of nvcc...

Portland Group Inc. (**PGI**): a Fortran compiler with CUDA extensions

- a high-level programming model that interoperates with highly-tuned low-level kernels: **CUDA Fortran**
- directive-based programming: **PGI Accelerator** (a software model for coding hardware accelerators)
- access to optimized GPU libraries



NVIDIA GPU generations and compute capability

G80 (since 2006): compute capability 1.0, **1.1**
features (1.1): 8 cores/multiprocessor, single-precision (SP) real arithmetic
models: GeForce 9800, Quadro FX 5600, Tesla C870, D870, S870

GT200 (since 2008): compute capability 1.2, **1.3**
features (1.3): double-precision (DP)
models: GeForce GTX 295, Quadro FX 5800, Tesla C1060, S1070

Fermi/GF100/GT300 (since 2010): compute capability **2.0**, 2.1
features (2.0): 32 cores/multiprocessor, faster DP, hardware cache
models: GeForce GTX 580, Quadro 6000, Tesla C2050, S2070

Product families: **GeForce** for games and PC graphics, **Quadro** for professional graphics, **Tesla** for HPC



A first view of NVIDIA hardware – Fermi (CC 2.0)

a **device**: a) 1–16 streaming multiprocessors (SMs)

b) **device memory** of about GB size, L2 cache of 768 KB

a **multiprocessor**:

a) 32 thread processors (CUDA cores) for integer and SP/DP real,
4 SP special function units (SFUs)

b) registers: 128 KB,
L1 cache + shared memory: 64 KB,
constant cache: 8 KB,
texture cache: 6–8 KB

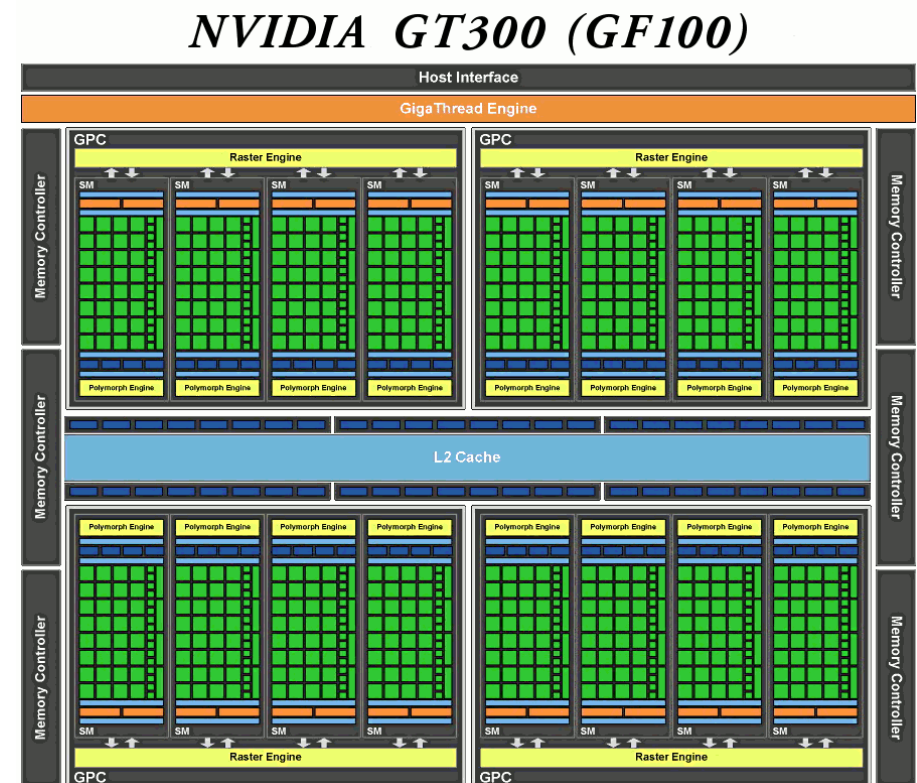
c) 2 instruction (warp) schedulers

one device: up to 16 SMs, i.e., $16 \times 32 = 512$ CUDA cores

one graphics card: up to 2 devices

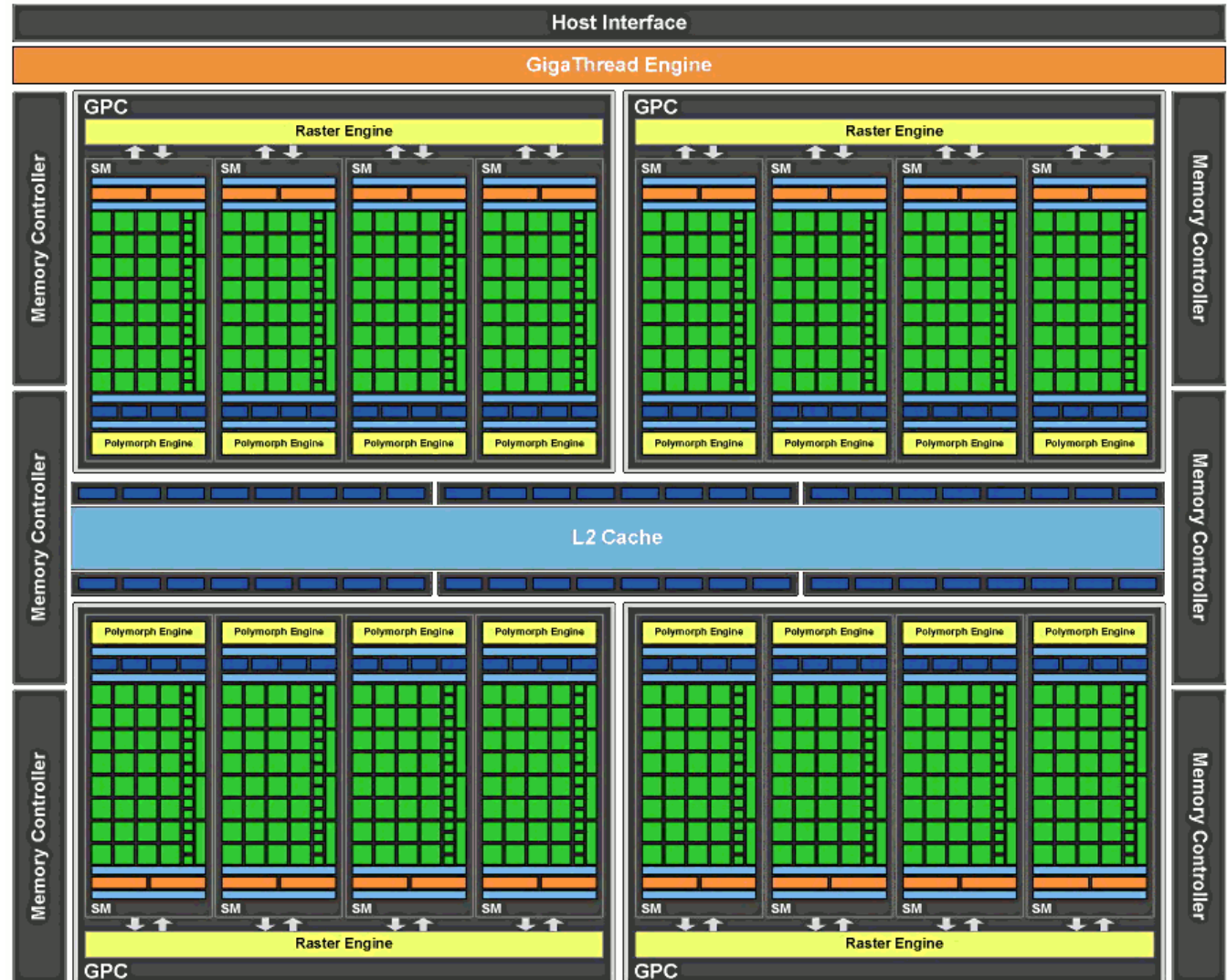
one motherboard: up to 2 graphics cards

a rack solution: 4 devices per module



A first view of NVIDIA hardware – Fermi (CC 2.0)

NVIDIA GT300 (GF100)



Comparison with multicore-CPU terminology

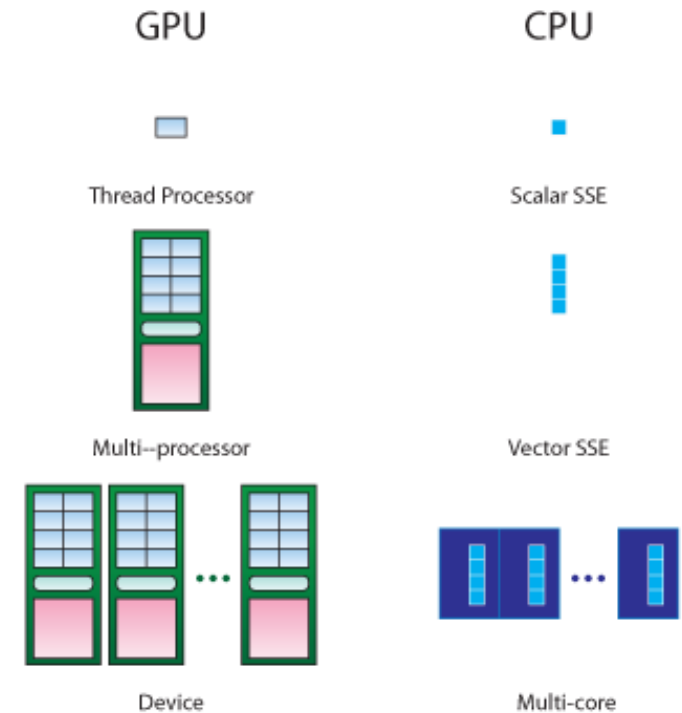
NVIDIA terms parallel-computing terms

a device ~ a multicore processor
with each core able to run independent to another
(MIMD parallelism)

a multiprocessor ~ a (vector) core
with the ability to switch
among several (vector) instruction streams
(interleaved multithreading)

CUDA cores ~ scalar units
executing concurrently a vector instruction stream
(SIMD parallelism)

see Wolfe (2010) about Intel Knights Ferry versus Fermi



Other compute capabilities

CC 1.3

a multiprocessor: **8 CUDA cores** for integer and SP real, **1 DP** real unit, 2 SP SFUs, 1 instruction scheduler
64 KB registers/SM, 16 KB smem/SM, 8 KB cmem cache/SM, 6–8 KB texture cache
according to NVIDIA documentation no L1 & L2 cache, but there is some (e.g., Volkov 2008)
devices with up to **30 SMs**, i.e., $30 \times 8 = 240$ CUDA cores/device

CC 2.1

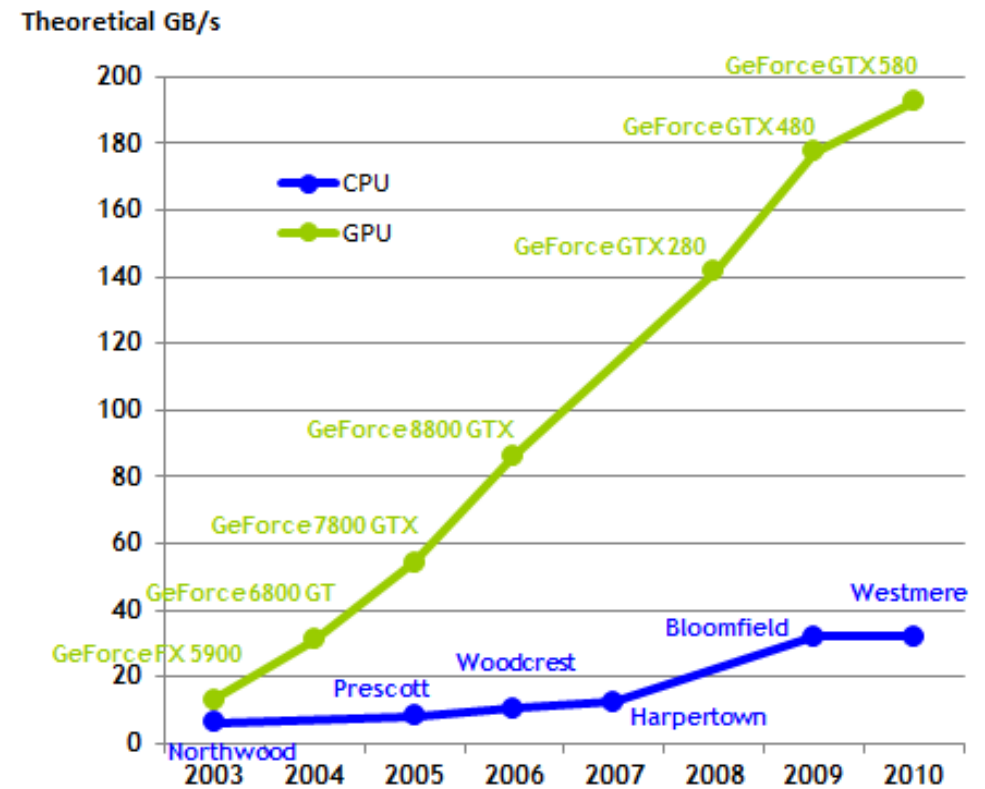
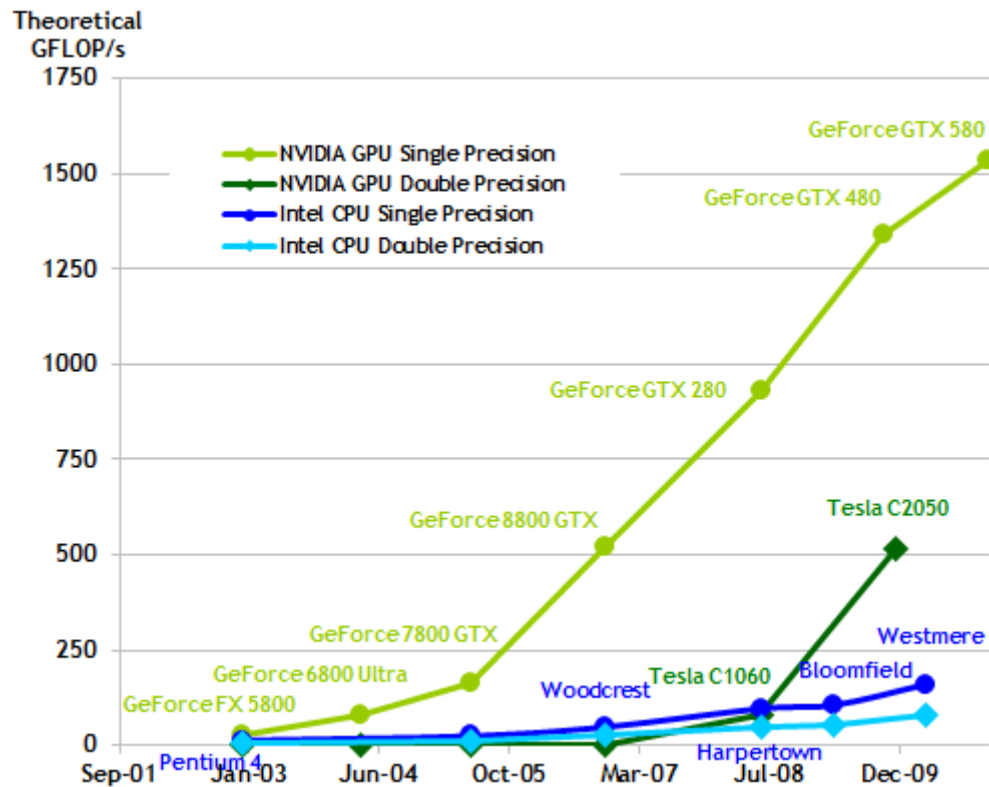
a multiprocessor: **48 CUDA cores**, **4 DP** instructions per clock cycle, 8 SP SFUs, 2 instruction schedulers
on-chip memory and L2 cache same as CC 2.0



Gflops by NVIDIA GPUs and Intel CPUs

Giga=10⁹, flops = flop/s = floating-point operations/s

(theoretical) Gflops = processor_clock_in_MHz * CUDA_cores * operations_per_clock / 1000



Gflops by NVIDIA GPUs and Intel CPUs

Top CC 2.0 products (June 2011)

CC	name	CUDA cores	dmem	SP Gflops	DP Gflops	power
2.0	Tesla S2050	4 x 14 x 32 = 1792	12 GB	4122	2061	900 W
2.0	Tesla M2090	16 x 32 = 512	6 GB	1331	665	? W
2.0	Tesla C2070	14 x 32 = 448	6 GB	1030	515	247 W
2.0	GeForce GTX 590	2 x 16 x 32 = 1024	3 GB	2488	1244	365 W
2.0	GeForce GTX 580	16 x 32 = 512	1.5 GB	1581	790	244 W

Top CC 1.3 products

1.3	Tesla S1070	4 x 30 x 8 = 960	16 GB	2765	346	700 W
1.3	Tesla C1060	30 x 8 = 240	4 GB	622	78	188 W
1.3	GeForce GTX 295	2 x 30 x 8 = 480	1.8 GB	1788	224	289 W

GPUs and CPUs for ~ USD 300

2.0	GeForce GTX 470	14 x 32 = 448	1.3 GB	1089	1/2 of SP	215 W
1.3	GeForce GTX 260	27 x 8 = 216	0.9 GB	912 (715)	1/8 of SP	182+ W
	Intel Core i7 950 (Nehalem Bloomfield)	4 cores		49	1/2 of SP	130 W

This notebook

2.1	GeForce GT 425M	2 x 48 = 96	1.0 GB	215	1/12 of SP	
	Intel Core i7 740QM (Nehalem mobile)	4 cores		28	1/2 of SP	45 W

(theoretical) Gflops = processor_clock_in_MHz * CUDA_cores * operations_per_clock / 1000

operations_per_clock = 2 (FMA) on CC 1.x, 2 on CC 2.x, possibly 3 (FMA+SF) on Tesla, 4 on Intel Nehalem

FMA = fused multiply-add, fma(x,y,z)=x*y+z, SF = special function

Gflops by NVIDIA GPUs and Intel CPUs

Throughput of native arithmetic instructions per multiprocessor
(operations per clock cycle per multiprocessor)

	integer +	integer *,FMA	SP +,*,FMA	DP +,*,FMA	SP SF (frcp, log2f, exp2f, sinf, cosf)
1.x	8	multiple	8	1	2
2.0	32	16	32	16	4
2.1	48	16	48	4 (slow!)	8

FMA = fused multiply-add, $\text{fma}(x,y,z)=x*y+z$

SF = special function

SP = (4B) single-precision real

DP = (8B) double-precision real

(NVIDIA CUDA C Programming Guide, Chap. 5)

Gflops/W

CC 2.0	4-7 Gflops/W
GeForce CC 1.3	3-6 Gflops/W
Intel i7 950	0.4 Gflops/W



CUDA software architecture

CUDA (Compute Unified Device Architecture): a general purpose parallel computing architecture

hardware: multiprocessor, cores, memory

software: a programming model

C/C++ compiler **nvcc**

CUDA API (Application Programming Interface) library

more CUDA tools by NVIDIA:

CUDA Toolkit with nvcc, CUDA debugger, Visual Profiler

GPU-accelerated numerical libraries: CUBLAS, CUSPARSE, CUFFT, CURAND

Computing SDK (Software Development Kit) code samples

more languages by third parties:

OpenCL (Khronos), **Brook** (Stanford University) – based on C language

Microsoft DirectCompute – a part of DirectX

PGI compiler suite (Portland Group) – **PGI CUDA Fortran**, **PGI CUDA C/C++**, **PGI Accelerator**

Jacket (AccelerEyes) – platform for Matlab

and many others



THE PORTLAND GROUP

CUDA programming model

in hardware: a device with multiprocessors (MIMD parallelism)
 a multiprocessor with CUDA cores (SIMD parallelism)

in software: a **grid** of blocks
 a **block** of **threads**

- blocks correspond to multiprocessors, a grid to a device
- a thread is executed by a CUDA core
- all threads of a block are executed by CUDA cores of a single multiprocessor
- threads of different blocks can be executed by different multiprocessors, each independent of another ("MIMD")

More about grids and blocks

- grids and blocks are effectively 1D, 2D or 3D indexed arrays of threads
- blocks are limited in size (~1024 threads), to fit well into 32 cores of a multiprocessor
- a grid size is effectively unlimited (~ 2^{48} ~ 10^{14} blocks)
- an optimal block size should be chosen carefully in order to reach a high **multiprocessor occupancy** (i.e., a number of threads resident in a multiprocessor)
- a grid size is chosen to meet a problem size with a given block size, $\text{block size} * \text{grid size} = \text{problem size}$
- a device with more multiprocessors can process a large grid faster



Software



Thread



Thread Block



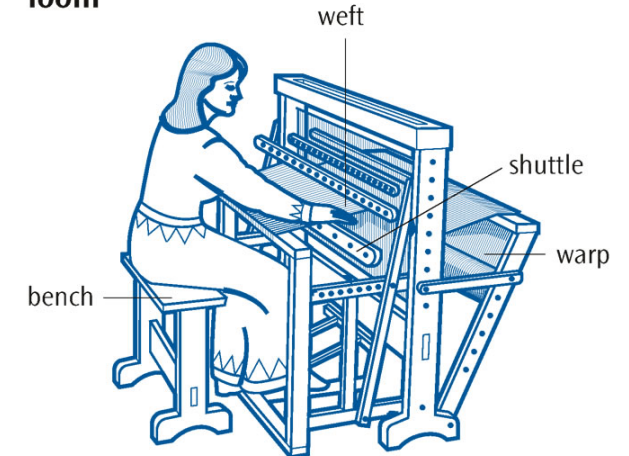
Thread Grid

CUDA programming model

Moreover, there are **warps**:

- groups (vectors) of 32 consecutive threads of a block that are executed in parallel in hardware ("SIMD", in CUDA rather **SIMT**: single-instruction multiple-threads)
- warps in a block are executed concurrently, but one at a time ("interleaved multithreading"), they are switched by **warp schedulers**
- threads in a warp are free to branch and execute independently, but a performance of a warp would be reduced (**divergent warps**)
- threads in a warp can benefit from access patterns to device memory that can be merged into one transaction (**memory coalescing**), e.g., addressing consecutive elements of a properly aligned array

loom



Kernel

- a procedure launched from the host and executed on the device
- a source code is written as for a single thread and executed by all threads
- the kernel executes asynchronously, i.e., the host process continues concurrently
- the host and the device are synchronized implicitly at the point of host-device memory transfer, or explicitly by a synchronization routine
- some devices are capable of execution concurrent with memory transfers
- the total number of threads, i.e., grid and block sizes, is set dynamically at the time of kernel launch

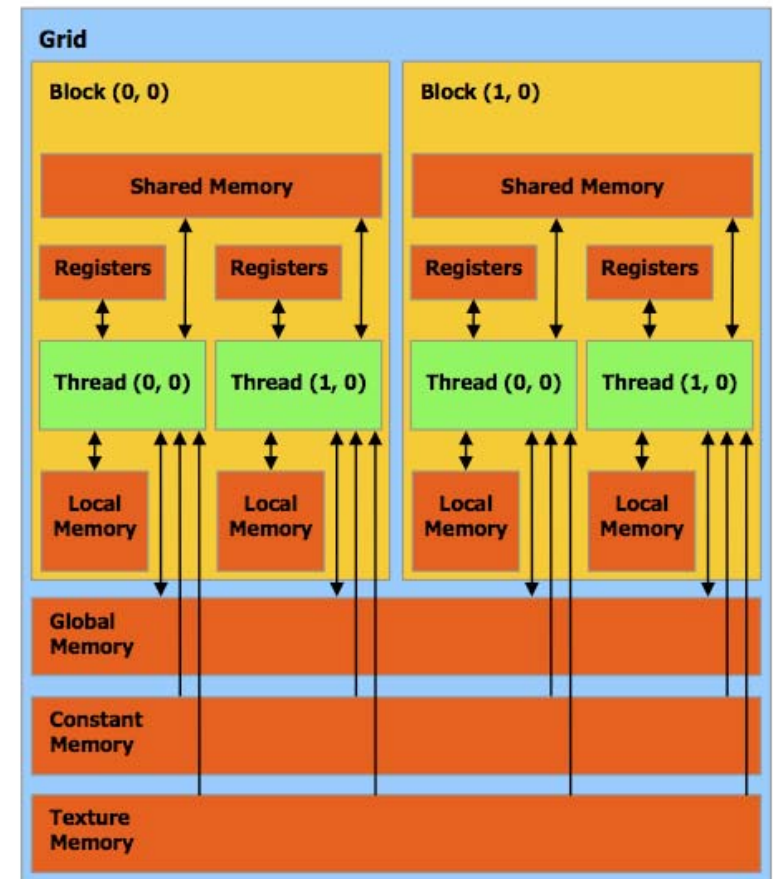
GPU memory hierarchy – Fermi (CC 2.0)

On device...

- device memory (**dmem**) used for
 - global memory: public data, shared by threads
 - local memory (**lmem**): private data, local in threads, did not fit into registers
 - constant memory: data initialized by the host, read-only in the device
 - texture memory: data initialized by the host, read-only in the device
- L2 cache for faster access to device memory, shared by all multiprocessors

On each multiprocessor („on-chip“)...

- registers: local data, also used internally by the compiler
- L1 cache: for faster access to device memory, shared by all CUDA cores in a multiprocessor
- shared memory (**smem**), shared by all CUDA cores in a multiprocessor („software-managed cache“)
- available configurations: 16 KB L1 cache + 48 KB smem or 48 KB L1 cache + 16 KB smem
- 8 KB constant cache: for faster reading from 64 KB constant memory (**cmem**) residing in dmem
- 6–8 KB texture cache: for faster reading from texture memory residing in dmem, optimized for 2D arrays



GPU memory hierarchy – Fermi (CC 2.0)

About the latency (Volkov 2008)...

- registers: no (read) latency
- smem (i.e., L1 cache): units or tens of clock cycles
- dmem: hundreds of clock cycles

and the memory bandwidth...

- transfers in dmem: from tens to above 100 GB/s
- host-dmem transfers: 6 GB/s (PCI Express 2.0) or less

On the host side...

- host memory can be allocated as **pinned** (page-locked):
 - pinned host-dmem transfers are faster by tens of % up to two times
 - the page-locked memory may not be available

```
Clock Rate:          1215 MHz
Initialization time: 337853 microseconds
Current free memory: 1260118016
Upload time (4MB):   863 microseconds ( 718 ms pinned)
Download time:       798 microseconds ( 649 ms pinned)
Upload bandwidth:    4860 MB/sec (5841 MB/sec pinned)
Download bandwidth: 5256 MB/sec (6462 MB/sec pinned)
```


CUDA compute capability limits

(NVIDIA CUDA C Programming Guide, App. F, also CUDA_Occupancy_Calculator.xls)

Grid and block related limits

- 1.x max block dimensions: 512-512-64, but total size: 512 threads/block
max grid dimensions: 65535-65535-1 (max 2D grids)
- 2.x max block dimensions: **1024**-1024-64, but total size: 1024 threads/block
max grid dimensions: **65535**-65535-65535 (3D grids)
- 1.0, 1.1 max 24 resident warps/SM, i.e., max 768 threads/SM
- 1.2, 1.3 max 32 resident warps/SM, i.e., max 1024 threads/SM
- 2.0, 2.1 max 48 resident warps/SM, i.e., max **1536** threads/SM
- all max 8 resident blocks/SM
warp size: 32 threads/warp



Memory related limits

	registers	lmem	smem	cmem	cmem cache	texture cache
1.0, 1.1	32 KB/SM	16 KB/thread	16 KB/SM	64 KB/device	8 KB/SM	6-8 KB/2 SMs
1.2, 1.3	64 KB/SM	16 KB/thread	16 KB/SM	64 KB/device	8 KB/SM	6-8 KB/3 SMs
2.0, 2.1	128 KB/SM	512 KB/thread	16-48 KB/SM	64 KB/device	8 KB/SM	6-8 KB/SM

CUDA on GeForce

a GeForce GPU is usually attached to a display and serves the graphical user interface of an operating system

the GUI is stalled during a kernel run, the display is updated between the kernel runs

there is a **runtime limit** for a single kernel on a GPU with a display attached:

Linux: ~ 8 s

Microsoft Windows XP: ~ 5 s

Microsoft Windows Vista, Windows 7: ~ 2 s

after that, the process calling the kernel is cancelled or the OS crash occurs

Linux: a window manager can be stopped (Ubuntu: `service gdm stop`), the system can then be accessed remotely and there is no timeout

Windows Vista and Windows 7 can disable or extend the limit via registry editing or merging registry entries by the .reg scripts:

to disable Timeout Detection and Recovery (TDR)...

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GraphicsDrivers]
"TdrLevel"=dword:00000000
```

to extend the 2-s limit to 60 s...

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GraphicsDrivers]
"TdrDelay"=dword:00000060
```

see `CUDA_Toolkit_Release_Notes.txt` or

http://www.microsoft.com/whdc/device/display/wddm_timeout.mspx
(Timeout Detection and Recovery of GPUs through WDDM)

Finally, a first example: addition of a 1D array and a scalar, $a(:)=a(:)+z$

a CPU version ... pgfortran -fast t1c.f90

```

MODULE mConst
INTEGER,PARAMETER :: DP=4,NMAX=4096*256
END MODULE

MODULE mProc
USE mConst
IMPLICIT NONE

CONTAINS

SUBROUTINE Assign(a,z)
REAL(DP) :: a(:)
REAL(DP) :: z
INTEGER :: j
do j=1,size(a)
  a(j)=a(j)+z
enddo
END SUBROUTINE

END MODULE

PROGRAM Template_1_CPU
USE mConst
USE mProc
IMPLICIT NONE
REAL(DP) :: a(NMAX),z

a=0.
z=1.
call Assign(a,z)

print *,a(1),a(NMAX),sum(a)

END PROGRAM

```

a GPU version ... pgfortran -fast -Mcuda t1g.f90

```

MODULE mConst
USE cudafor
INTEGER,PARAMETER :: DP=4,NG=4096,NB=256,NMAX=NG*NB
TYPE(dim3),PARAMETER :: grid=dim3(NG,1,1),block=dim3(NB,1,1)
END MODULE

MODULE mProc
USE mConst
IMPLICIT NONE

CONTAINS

ATTRIBUTES(GLOBAL) SUBROUTINE Assign(a,z)
REAL(DP) :: a(:) ! DEVICE attribute by default
REAL(DP),VALUE :: z
INTEGER :: j
  j=threadidx%x+NB*(blockidx%x-1)
  a(j)=a(j)+z
END SUBROUTINE

END MODULE

PROGRAM Template_1_GPU
USE mConst
USE mProc
IMPLICIT NONE
REAL(DP) :: a(NMAX),z
REAL(DP),DEVICE :: ad(NMAX)

ad=0.
z=1.
call Assign<<<grid,block>>>(ad,z)
a=ad
print *,a(1),a(NMAX),sum(a)

END PROGRAM

```

A first example: addition of a 1D array and a scalar, $a(:)=a(:)+z$

Differences between CPU and GPU versions:

- initialization: cudafor module, grid and block shape and size
- a kernel: global attribute, attributes of arguments, outer loops replaced by thread indexing
- a kernel call: allocation of device data, host-device data transfers, executable configuration

Examples in CUDA Fortran SDK folder

bandwidthTest

goal: speed of CPU-GPU and GPU-GPU data transfers



Links and references

NVIDIA hardware

<http://www.nvidia.com/tesla> etc.

http://en.wikipedia.org/wiki/Nvidia_Tesla etc.

NVIDIA GPU Computing Documentation

NVIDIA CUDA C Programming Guide (esp., Chap. 4 & 5 & App. A & F)

<http://developer.nvidia.com/nvidia-gpu-computing-documentation>

PGI resources

Articles, PGInsider newsletters, White papers and specifications, Technical papers and presentations

<http://www.pgroup.com/resources/articles.htm>

Volkov V., Demmel J. W., Benchmarking GPUs to tune dense linear algebra, 2008

<http://www.cs.berkeley.edu/~volkov/>

Wolfe M., Compilers and More: Knights Ferry versus Fermi, 2010

http://www.hpcwire.com/hpcwire/2010-08-05/compilers_and_more_knights_ferry_versus_fermi.html