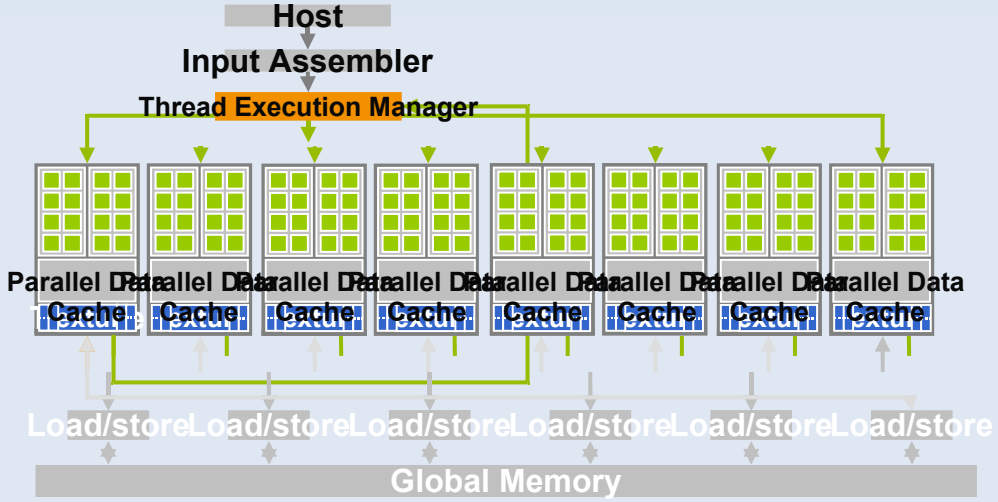
Kurs:

Procesory graficzne w obliczeniach równoległych (CUDA)

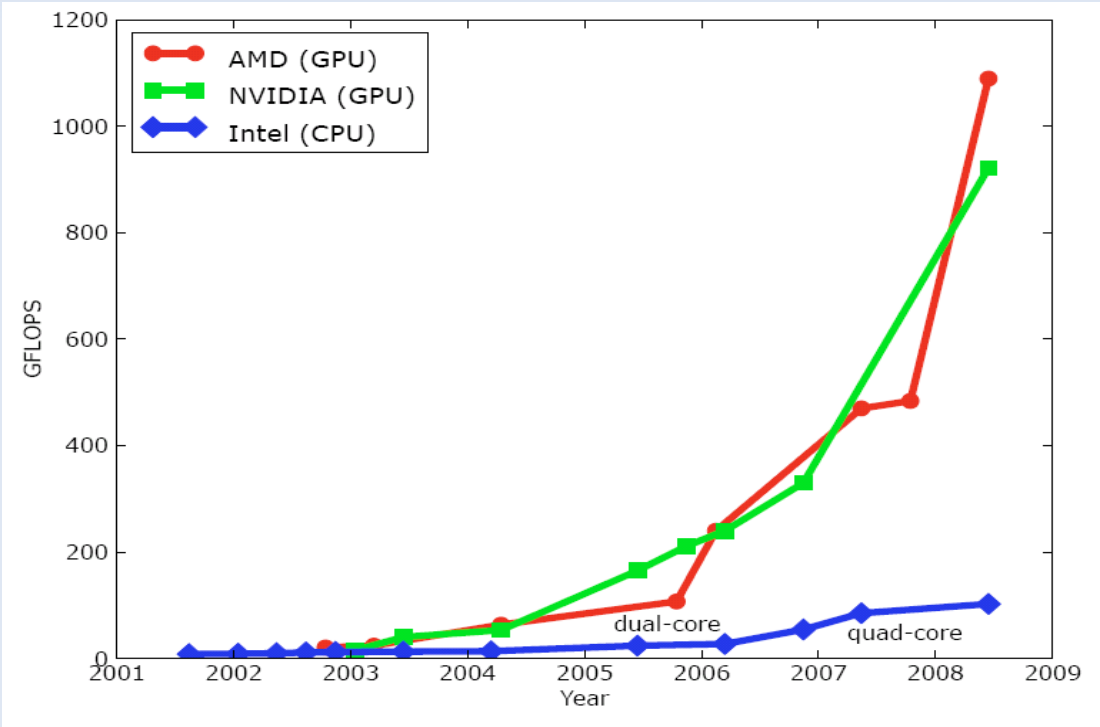


Obrazki na slajdach jeśli nie zaznaczone inaczej: © Kirk, Hwu ”Programming massively Parallel Processors”, Elsevier 2010

Dlaczego programujemy GPU ?

- Grafika komputerowa
- Inne kosztowne obliczenia o podobnym charaktererze użycia pamieci

- Rosnąca moc i przepustowość GPU względem CPU:



Dlaczego CUDA ?

- Technologia CUDA NVIDIA najdłużej na rynku
- Karty NVIDIA szeroko rozpowszechnione,

  tania moc obliczeniowa
- Konkurencja: OpenCL uniwersalny, także dla kart ATI lub innych procesorów, drivery od niedawna, model podobny
- W praktyce musimy znać konkretną architekturę i jej ograniczenia aby pisać efektywnie (nie ma uniwersalnych rozwiązań)
-

Programowanie GPU : CUDA

- Historia i rozwój kart graficznych
- Biblioteka CUDA i proste przykłady programów
- Jak programować masywnie równoległe procesory i osiągnąć: wydajność, skalowalność (także dla przyszłych GPU) i przejrzysty kod który można dalej rozwijać
- Wiedza na temat wzorców programowania równoległego, cech architektury, ograniczeń, API, narzędzi i technik

Wymagania

- Bardzo dobra znajomość języka C
- Praktyka programowania w języku C
- Zapał do niskopoziomowego programowania

Wymagania

- Bardzo dobra znajomość języka C
- Praktyka programowania w języku C
- Zapał do niskopoziomowego programowania

Dobrze widziane:

- Grafika komputerowa
- Programowanie vertex/fragment shaderów
- Obliczenia równoległe np. co to jest deadlock ?
- Architektura procesorów

WWW, Literatura

- Moodle: http://kno.ii.uni.wroc.pl klucz kursu '7cuda7'
- WWW: NVIDIA developer, CUDA Zone, CUDA Showcase (linki na strone kursu)
- Dawid B.Kirk, Wen-mie W. Hwu - ”Programming Massively Parallel Processors”, Elsevier 2010.
- Nvidia,*CUDA Programming Guide: /usr/local/cuda/doc*
- Nvidia,*CUDA Reference Guide,.. (katalog jak wyżej)*

Wykład i pracownia

- Wykład 15 godzin, głównie początek semestru
- 10 pracowni z krótszymi zadaniami
- 5 tygodni na większy projekt końcowy

Pracownie:

- Linux (A.Łukaszewski)
- Windows (Ł.Piwowar)

Historia GPU: fixed piepline

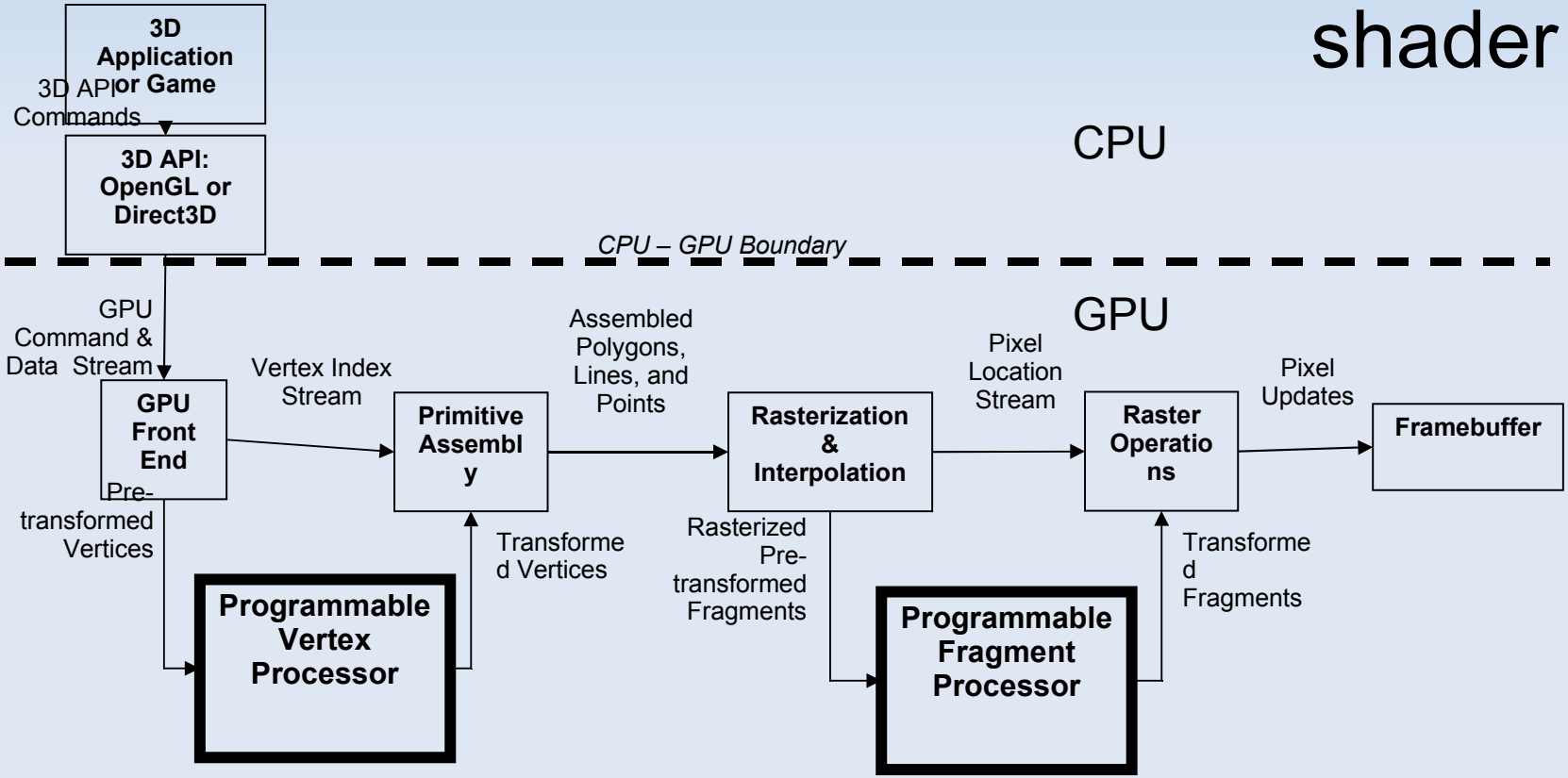
- 1986 SGI Iris pierwsza karta graficzna z akceleracja 3D :

  standardowy potok przetwarzania grafiki

  trójkąty->transformacje,obcinanie,oświetlenie->
  ->rasteryzacja,interpolacja,teksturowanie->2d

Historia GPU: programowalny potok graficzny

- 2001 Nvidia GeForce3 (NV20), karty grafiki z programowalnym potokiem: vertex/fragment shader



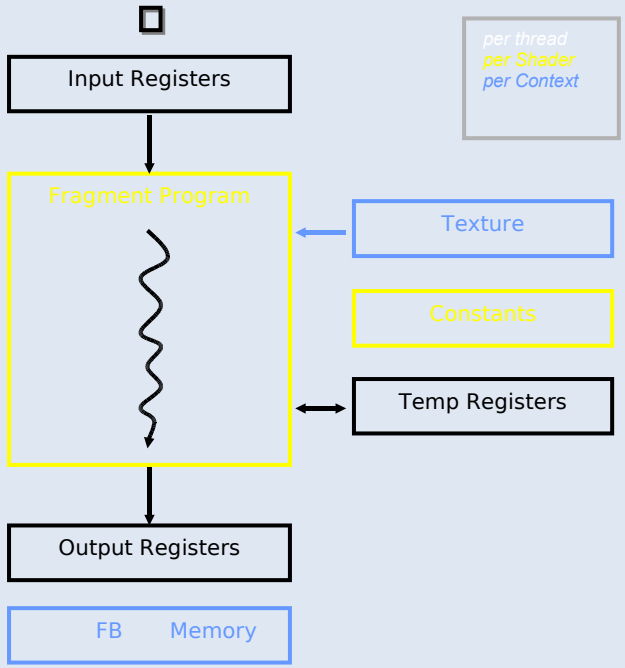
Historia GPU: GPGPU

- GPGPU - General Processing on GPU
- Wykorzystanie programowalnych vertex/fragment shaderów w potoku graficznym do obliczeń równoległych innych niż potok 3D
- Kodowanie wejścia. Wyniki: framebufer/ekran
- Mało naturalne i z dużą ilością ograniczeń ale dzięki wzrastającej mocy kart (liczne procesory) wiele rozwiązań szybszych niż na CPU
- Na początku grafika komp. np. mapy fotonów

Ograniczenia GPGPU

- Graphics API
  - Skrajne przypadki API
- Tryby adresowania
  - Ograniczony wymiar/rozmiar tekstur
- Możliwości shaderów
  - Ograniczone wyjście (multiple render targets)
- Ograniczone instrukcje
  - Brak obliczeń całkowitych i logicznych
- Ograniczone możliwości komunikacji
  - Pomiędzy pikselami

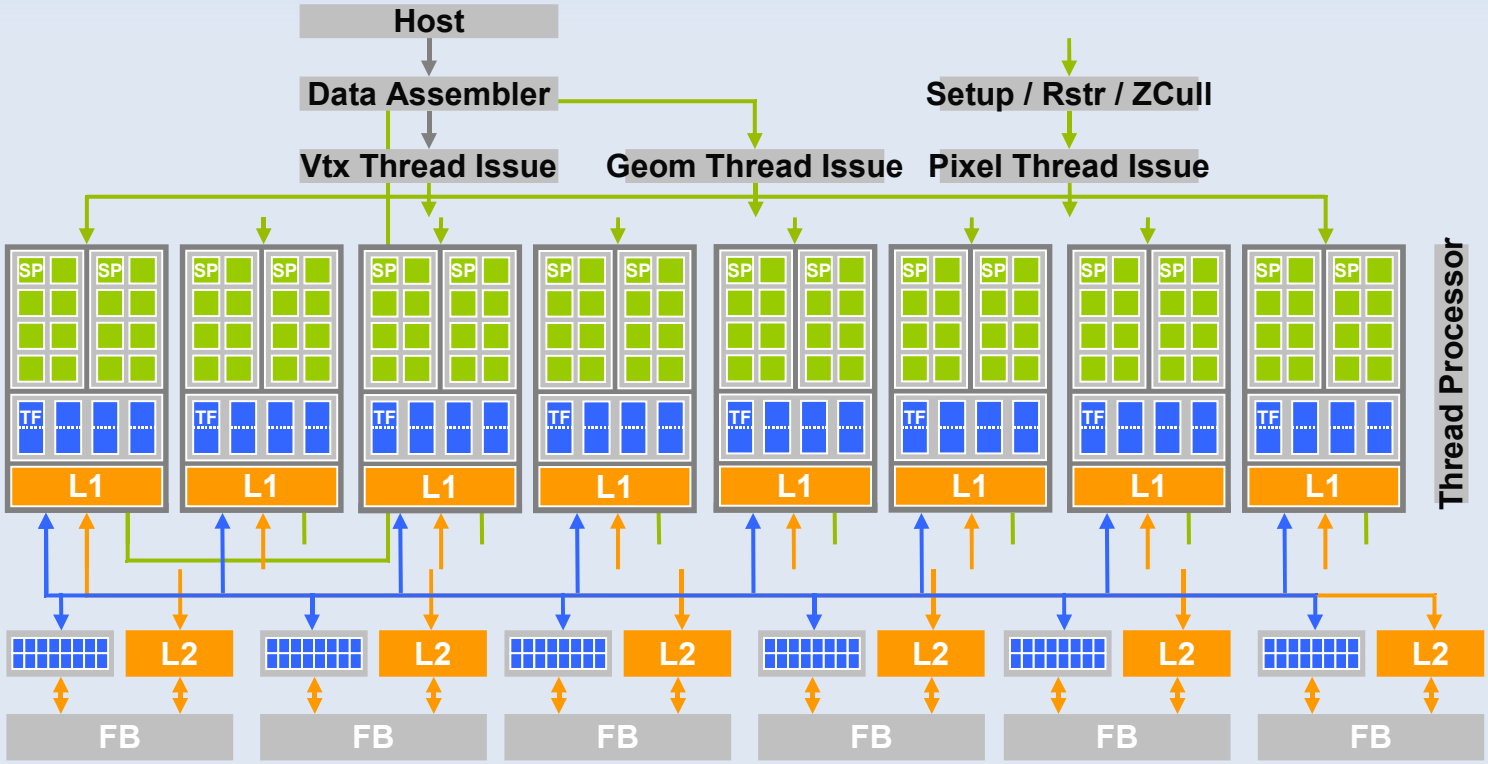
Więcej np.: gpgpu.org



Historia GPU: zunifikowany potok

- 2006/2007 Nvidia GeForce 8xxx (NV80):

  CUDA (Common Unified Device Architecture) zunifikowane procesory do różnych shaderów



Historia GPU: CUDA

Dwa znaczenia:

- Architektura karty (CUDA - Common Unified Device Architecture)
- Nazwa bibilioteki do programowania równoległego kart graficznych

- Używając OpenGL i CUDA trzeba przełączać kontekst, można używać kilku kart GPU

CUDA: urządzenia i wątki

- Urządzenie: compute device
  - Dla CPU/host jest to koprocesor
  - Własna pamięć DRAM (device memory)
  - Wykonuje wiele wątków równolegle (threads in parallel)
  - Typowo to GPU ale może też być innym urządzeniem
- Równoległe fragmenty aplikacji są wyrażone w postaci jąder (device kernels) uruchamianych równolegle
- Różnice między wątkami na GPU i CPU
  - Wątki GPU ekstremalnie lekkie
    - Bardzo mały narzut przy tworzeniu
  - GPU potrzebuje tysięcy wątków dla pełnej efektywności
    - Wielordzeniowym CPU wystarczy kilka

Instalacja biblioteki CUDA

- Google: nvidia cuda download
- Obecnie wersja 3.2 (IX.2010), 3.1(VI.2010),...
- Trzy rzeczy dopasowane do wersji systemu (!):
  - Driver karty graficznej
  - Biblioteka CUDA
  - SDK z przykładami
- SDK na pracowni w /opt/NVIDIA_...
- Kompilacja SDK w podkatalogu C/ ”make”

Instalacja biblioteki CUDA

- Biblioteka CUDA: kompilator nvcc, debugger, biblioteki dynamiczne, dokumentacja
- Ustawiamy ścieżki do kompilatora (linux):

  export PATH=$PATH:/usr/local/cuda/bin
- Ścieżki do bibiliotek (linux):

  do LD_LIBRARY_PATH /usr/local/cuda/lib
- Skopiować jeden przykład i zmienić makefile

  tak aby skompilować u siebie

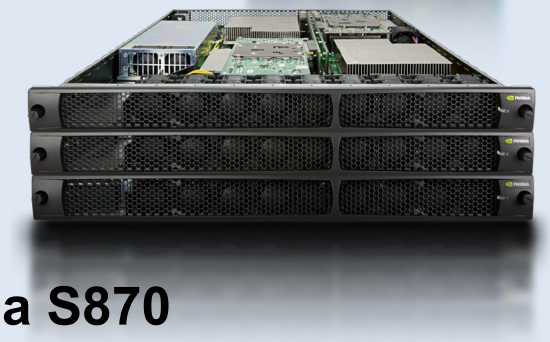
Hardware

- NVIDIA GF od generacji 8xxx tzn.:
  8200, 8800,... 9600,... 270, 275,... 3xx, 4xx,...
  (także karty wbudowane w płyte główną)
- Pracownia 7: komputery z kartami GTX275:
  30multiprocesorów=240rdzeni, 896MB pamięci
- Inne rozwiązanie: emulacja
  po zainstalowaniu SDK : ”make emu=1”

Hardware

GeForce 8800

Tesla D870

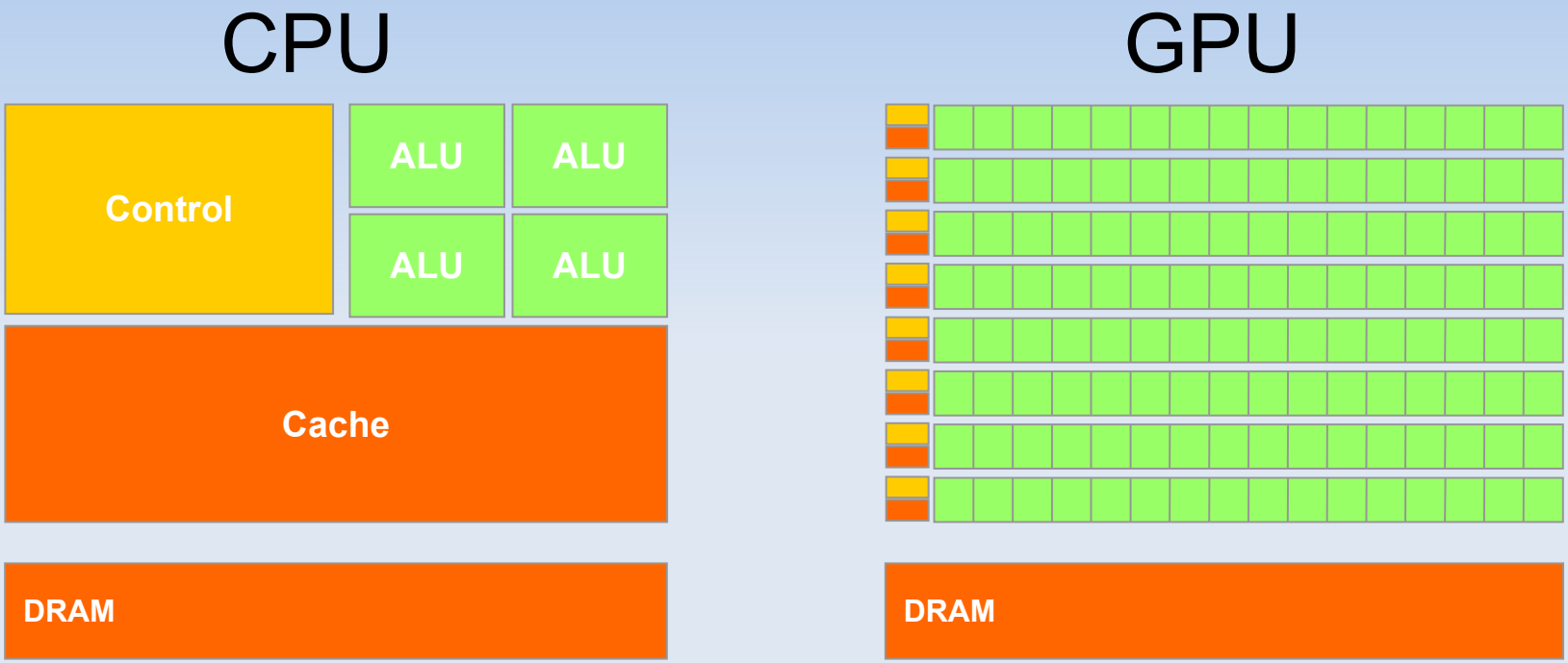


Tesla S870

Lab: GTX 275 (kwiecień 2009)

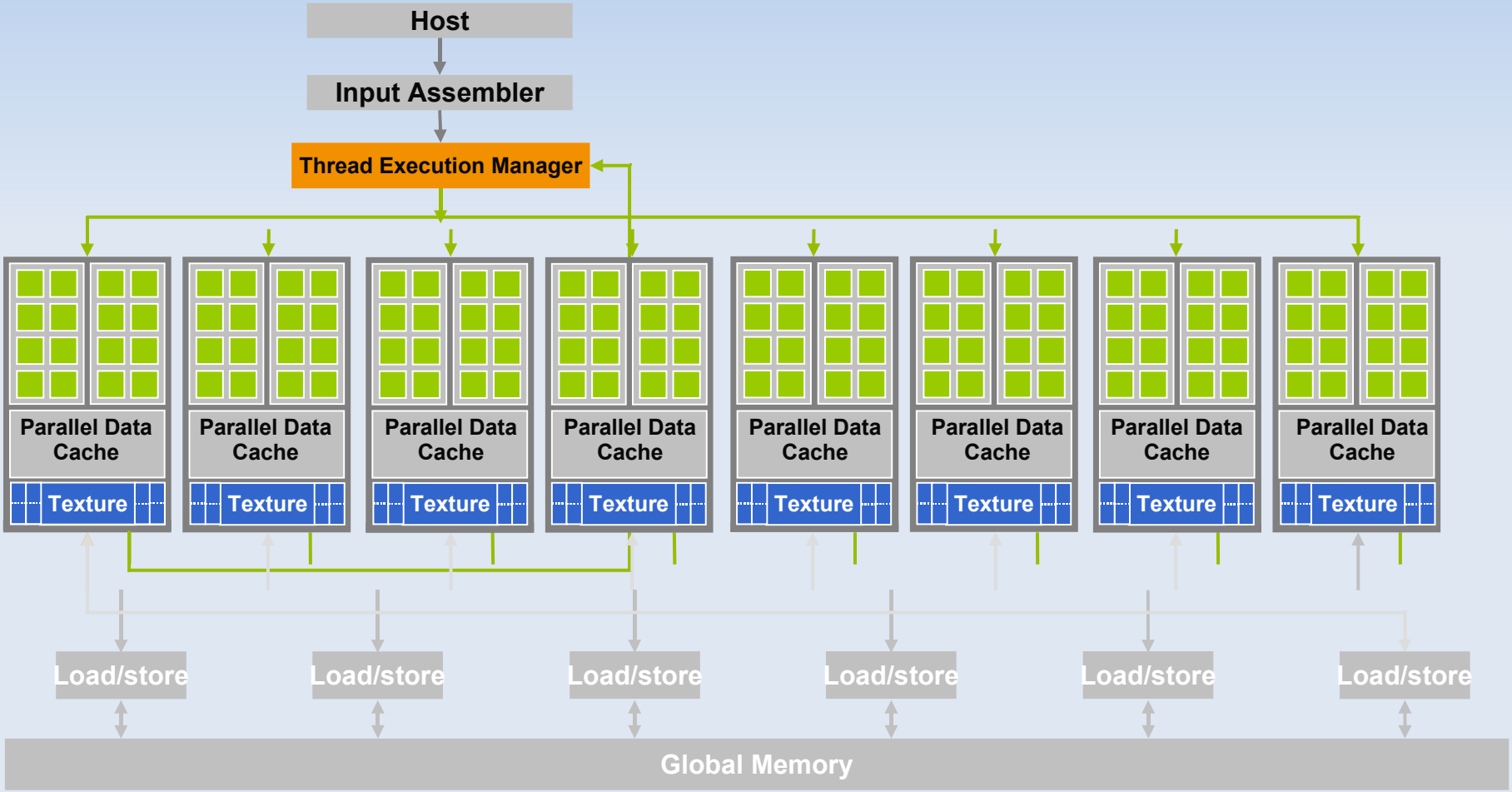
- 1 Teraflop mocy obliczeniowej, czyli 25-50 razy więcej niż współczesne CPU, szyna 150GB/s
- 240 rdzeni, potrzeba jeszcze więcej wątków
- Naiwne przeniesienie implementacji z CPU może być kilka razy szybsze lub też wolniejsze
- Zoptymalizowane implementacje 30-100 razy szybsze niż na CPU
- Nowsze: GTX 480 : 2x więcej rdzeni
- Nvidia Tesla – dedykowane karty do obliczeń

CPU vs. GPU



CPU: cache próbuje ”sprytnie” eliminować za nas opóźnienia, na GPU musimy o to zadbać sami, za to więcej jednostek obliczeniowych

Architektura CUDA

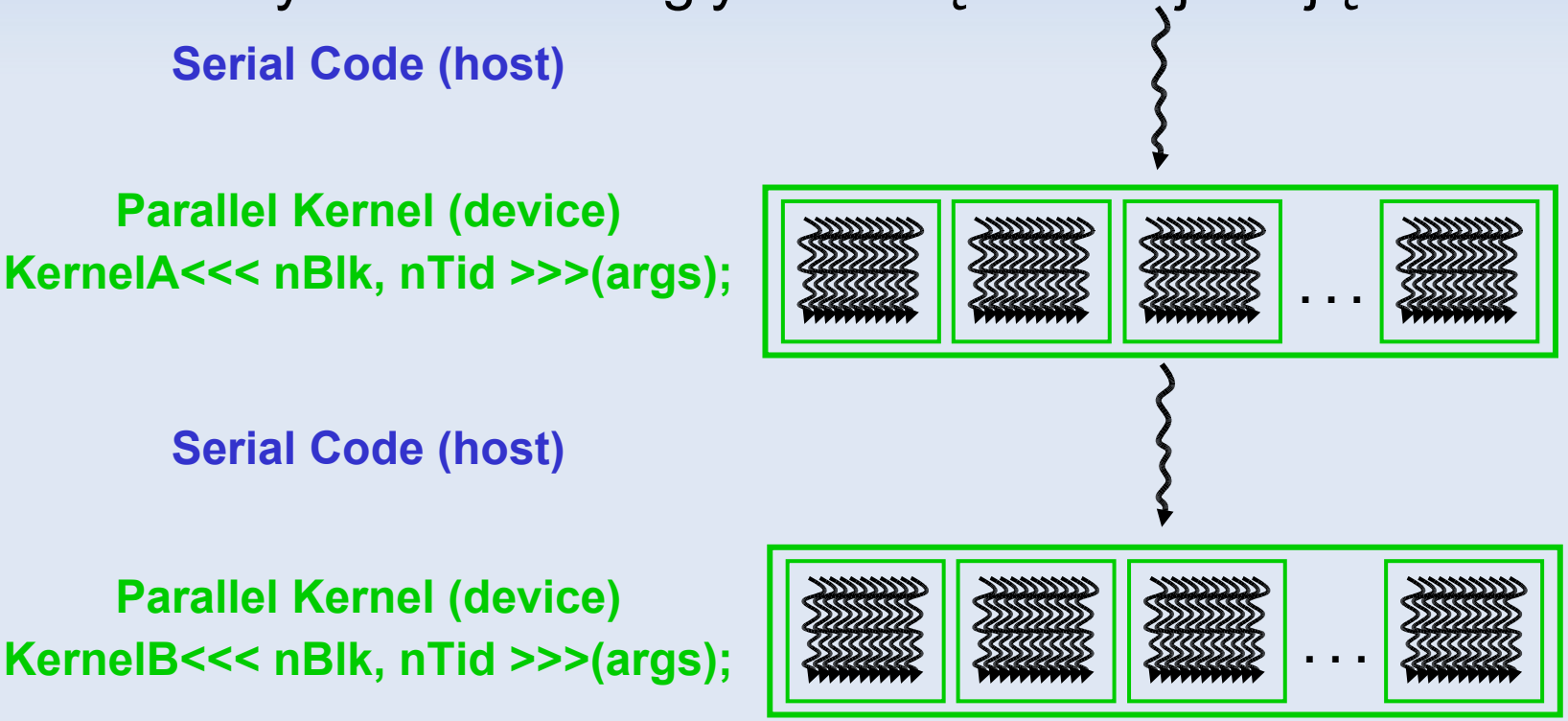


CUDA: model programowania

- Zintegrowany program w C dla CPU + urządzenia (GPU):

  kod sekwencyjny wykonywany na CPU

  kod masywnie równoległy na urządzeniu jako jądra SPMD



CUDA: pierwszy fragment

- CUDA program: rozszerzenie .cu, kompilator nvcc: głównie C z dodatkowymi rozszerzeniami

```
… // Cześć sekwencyjna na CPU
// setup execution parameters
dim3  grid( num_blocks, 1, 1);
dim3  threads( num_threads, 1, 1);
my_kernel<<<grid, threads>>>(param1, param2);
… // ciag dalszy na CPU
//-----------------------------------------
-
// kernel:  (plik.cu) dok wykonania na GPU
__global__ void my_kernel(int param1, char
*param2){
…
}
```