CUDA na Twoim biurku

czyli słów kilka na temat Compute Unified Device Architecture

Zbigniew Koza

Instytut Fizyki Teoretycznej
Uniwersytet Wrocławski

Wrocław, 3 lipca 2009

Spis treści

1. Wstęp: GPU a CPU
   - CPU
   - GPU

Spis treści

1. Wstęp: GPU a CPU
   - CPU
   - GPU
2. CUDA
   - Co to jest CUDA?
   - Architektura procesora GT 200
   - CUDA API

Spis treści

1. Wstęp: GPU a CPU
   - CPU
   - GPU
2. CUDA
   - Co to jest CUDA?
   - Architektura procesora GT 200
   - CUDA API
3. Przyszłość GPU

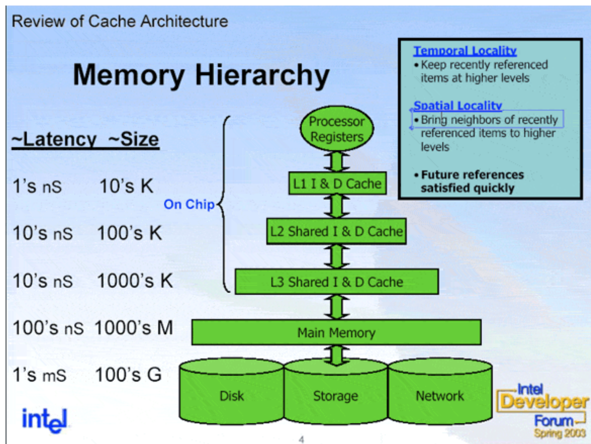
CPU („jednostka arytmetyczno-logiczna”)

- CPU to procesor sekwencyjny ogólnego przeznaczenia
  - Optymalizowany do przetwarzania strumienia poleceń na heterogenicznych danych
  - Swobodny dostęp do pamięci operacyjnej
  - Stosunkowo skomplikowane sterowanie
  - Oryginalnie projektowany jako procesor typu SISD (single instruction, single data)

CPU – coraz prędzej. . .

- Ok. 2003 r. osiągnięto taktowanie rzędu 3 GHz i tak już zostało. . .
- Taktowanie rzędu 3-4 GHz to obecnie fundamentalna bariera technologiczna

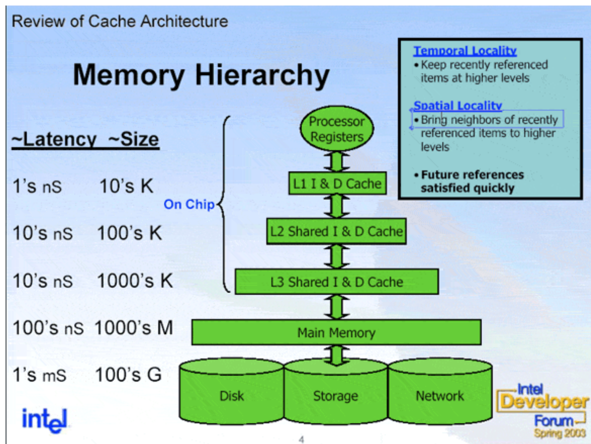
Wąskie gardło: dostęp do danych (*latency*)



Źródło: tomshardware.com/Intel

- *Latency* ≈ czas między zleceniem a pierwszym jego efektem
- Efekt można zmniejszyć, pobierając dane z sąsiednich lokalizacji

Antidotum sprzętowe: pamięć podręczna L1, L2, L3



Źródło: tomshardware.com/Intel

- Procesor Intela Montecito (dual core Itanium 2) to ponad 1.7×10^9 tranzystorów; 1.5×10^9 z nich to 24 MB pamięci podręcznej L3

CPU – coraz szerzej. . .

- Zamiast zwiększać częstotliwość zegara, w coraz większym stopniu „urównolegla” się CPU:
  - Architektura superskalarna
  - Instrukcje typu SIMD (single instruction, multiple data)
  - Hyperthreading
  - Wielordzeniowość
  - Wieloprocesorowość
  - Klastry
  - . . .

CPU – coraz trudniej. . .

Postępujące „urównoleglanie” CPU wymaga
zmiany paradygmatu programowania

CPU – coraz trudniej. . .

Postępujące „urównoleglanie” CPU wymaga
zmiany paradygmatu programowania

Ten nowy paradygmat jeszcze nie istnieje. . .

GPU („koprocesor graficzny")

- GPU to wyspecjalizowany akcelerator sprzętowy zaprojektowany do przetwarzania geometrii (np. miliony trójkątów) i tekstur (mapy bitowe) w barwne piksele na ekranie (frame buffer)
- GPU stosuje te same operacje matematyczne (np. macierze obrotów i translacji) na ogromnej liczbie podobnych elementów
- Te operacje wykonywane są zwykle na *liczbach zmiennoprzecinkowych pojedynczej precyzji*

GPU – procesor strumieniowy

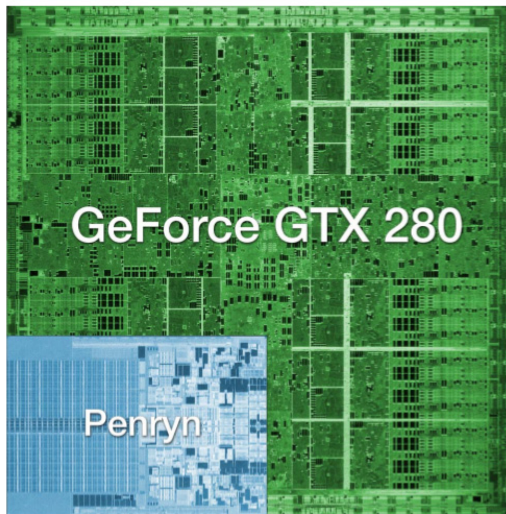
- GPU są projektowane do jednoczesnego wykonywania tych samych instrukcji na tysiącach niemal niezależnych wierzchołkach trójkątów i elementach tekstur
- Na wejściu jest strumień wierzchołków i strumień tekseli (teksel = texture element), na wyjściu – strumień pikseli.
- Tekstury mają zwykle geometrię 2D, stąd GPU posiadają specjalne układy do „geometrycznej” optymalizacji wczytywania strumienia wejściowego (specjalna pamięć podręczna)
- GPU są optymalizowane do wykonywania dodawania, odejmowania i mnożenia danych zmiennopozycyjnych pojedynczej precyzji; słabiej radzą sobie z dzieleniem.
- GPU są więc przykładem procesorów SIMD (Single Instruction, Multiple Data)

GPU – procesor „menyrdzeniowy”

Terminologia:

- CPU: „multicore processor”. „Multi” $\lesssim 8$
- GPU: „a highly parallel, multithreaded, manycore processor”. „Many” $\gtrsim 100$

Który większy?

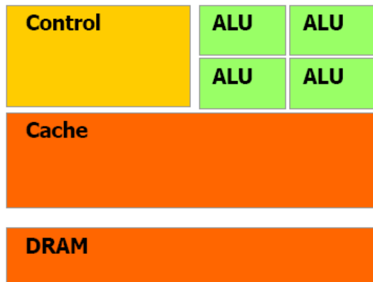


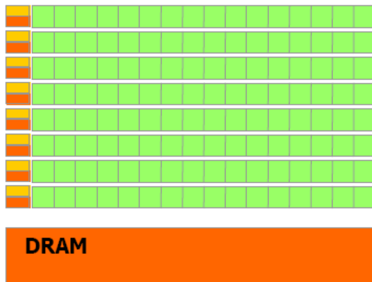
- GTX 280:
  1.4×10^9 tranzystorów,
  65 nm,
  240 rdzeni,
  pobór mocy: 160-330 W.
  1 TFLPOS (fp32)
- Intel Penryn:
  0.4×10^9 tranzystorów,
  technologia 45 nm,
  2 rdzenie,
  pobór mocy: 4-70 W.
  $\approx$ 0.01 TFLOPS (fp64)

Źródło: anandtech.com

GPU – numerkowy potwór

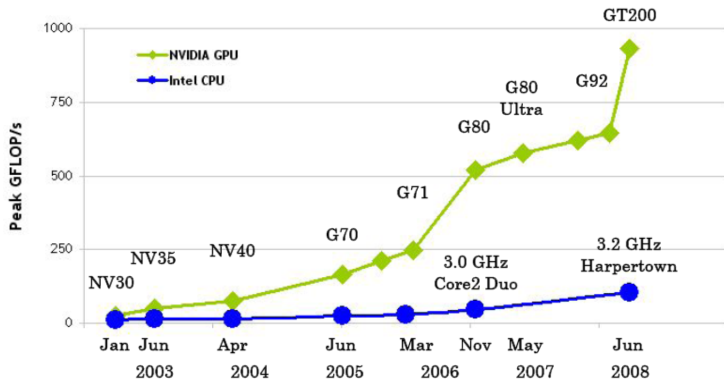
- CPU: większość tranzystorów tworzy pamięć podręczną
- GPU: pamięć podręczna mniejsza; więcej tranzystorów wykonuje obliczenia





Źródło: NVIDIA CUDA Programming Guide

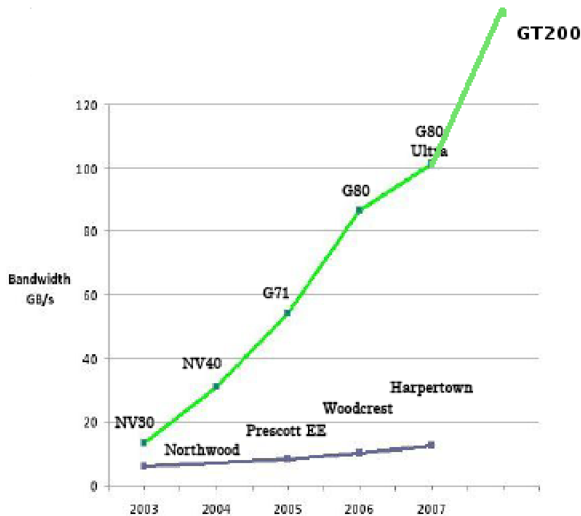
GPU – superkomputer za 300 zł



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

Źródło: NVIDIA CUDA Programming Guide

Magiczne słowo – bandwidth



Źródło: NVIDIA CUDA Programming Guide

GPU są naprawdę szybkie!

Performance (gflops) Development Time (hours)

	Performance (gflops)	Development Time (hours)
Matlab	0.3	0.5
C/SSE	9.0	10.0
PS3	110.0	30.0
GT200	330.0	10.0

3D Filterbank Convolution

Nicolas Pinto, James DiCarlo, David Cox (MIT, Harvard)

Źródło: Nicolas Pinto, MIT

GPU są szybkie nie tylko w testach!

- Istnieje wiele przykładów przyspieszenia obliczeń naukowo-inżynierskich na GPU o czynnik rzędu 10-100 względem CPU
- Takie przyspieszenie to prawdziwa rewolucja
  - Obliczenia trwające rok → tydzień
  - Obliczenia trwające tydzień → godziny
- Nowe obszary zastosowań (np. diagnostyka medyczna)

GPU = śmierć CPU?

- Obliczenia na GPU zamiast CPU mają sens tylko w wąskiej klasie problemów
- Bardzo wąskie gardło: komunikacja CPU↔GPU
- Mniej wąskie gardło: komunikacja rdzeni GPU↔DRAM GPU
- GPU wymaga bardzo specyficznych technik programistycznych

GPU = śmierć CPU?

- Obliczenia na GPU zamiast CPU mają sens tylko w wąskiej klasie problemów
- Bardzo wąskie gardło: komunikacja CPU↔GPU
- Mniej wąskie gardło: komunikacja rdzeni GPU↔DRAM GPU
- GPU wymaga bardzo specyficznych technik programistycznych
- Wniosek: GPU nigdy nie zastąpi CPU

GPU = śmierć CPU?

- Obliczenia na GPU zamiast CPU mają sens tylko w wąskiej klasie problemów
- Bardzo wąskie gardło: komunikacja CPU↔GPU
- Mniej wąskie gardło: komunikacja rdzeni GPU↔DRAM GPU
- GPU wymaga bardzo specyficznych technik programistycznych
- Wniosek: GPU nigdy nie zastąpi CPU
  - ale może w jakimś stopniu zostać zintegrowany z CPU jak niegdyś koprocesor matematyczny...
  - albo może sam upodobnić się do CPU ⇒ cGPU

GPU = śmierć CPU?

- Obliczenia na GPU zamiast CPU mają sens tylko w wąskiej klasie problemów
- Bardzo wąskie gardło: komunikacja CPU↔GPU
- Mniej wąskie gardło: komunikacja rdzeni GPU↔DRAM GPU
- GPU wymaga bardzo specyficznych technik programistycznych
- Wniosek: GPU nigdy nie zastąpi CPU
  - ale może w jakimś stopniu zostać zintegrowany z CPU jak niegdyś koprocesor matematyczny...
  - albo może sam upodobnić się do CPU ⇒ cGPU
- Nigdy nie mów nigdy...

Krótka historia GPU

- Pierwsze GPU – procesory strumieniowe całkowicie „zaprogramowane” w fabryce
- 2003 – GeForce FX
  - obsługa liczb zmiennopozycyjnych
  - ponad tysiąc instrukcji
  - wada: dostęp wyłącznie poprzez OpenGL/DirectX etc.
- 2003/2004 – Brook
  - Brook to kompilator i nowy język programowania oparty na C („C with streams”)
  - → Nowa gałąź informatyki: GPGPU (*General-purpose computing on graphics processing units*)
  - Problem 1: brook korzysta z OpenGL/DirectX/... → narzut
  - Problem 2: zwykła zmiana sterownika może „rozłożyć” program
- 2006 – GeForce 8 z technologią **CU**DA. **U**nifikacja programowalnych jednostek (*vertex* i *pixel shaders*) → uniwersalne multiprocesory.

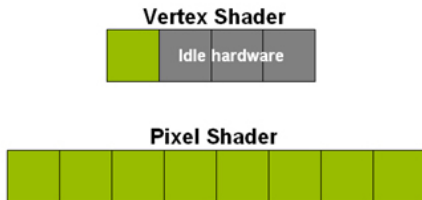
Wielka unifikacja

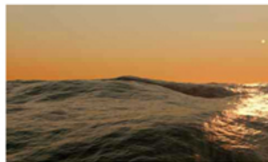
- Pierwsze programowalne GPU miały dwie niezależne programowalne „jednostki cieniujące" – *vertex shader* i *pixel shader*, rozszerzone później o *geometry shader*
- *Vertex shader* – przetwarzanie danych o wierzchołkach (m.in. położenie, wektor normalny do powierzchni, rodzaj powierzchni) i rzutowanie na płaszczyznę ekranu
- *Geometry shader* – zmiana struktury siatki wierzchołków
- *Pixel shader* – wypełnianie wielokątów kolorem (interpolacja!) lub teksturą oraz wyznaczanie parametrów koloru na podstawie danych o oświetleniu, materiale, przezroczystości. . .
- Współczesne karty graficzne – zunifikowane, uniwersalne jednostki cieniujące. *Vertex*, *geometry* & *pixel shader* w jednym układzie.

Po co wielka unifikacja?

Why unify?

Heavy Geometry
Workload Perf = 4



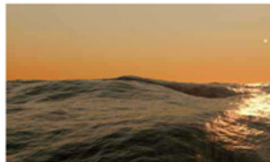


Heavy Pixel
Workload Perf = 8

Po co wielka unifikacja?

Why unify?

Heavy Geometry
Workload Perf =12



Heavy Pixel
Workload Perf = 12

Część 2

1. Wstęp: GPU a CPU
   - CPU
   - GPU
2. CUDA
   - Co to jest CUDA?
   - Architektura procesora GT 200
   - CUDA API
3. Przyszłość GPU

CUDA

- CUDA = Compute Unified Device Architecture

CUDA

- CUDA = Compute Unified Device Architecture
- CUDA = uniwersalna architektura sprzętowo-programistyczna masowo równoległych, masowo wielordzeniowych procesorów (graficznych) firmy NVidia

CUDA

- CUDA = Compute Unified Device Architecture
- CUDA = uniwersalna architektura sprzętowo-programistyczna masowo równoległych, masowo wielordzeniowych procesorów (graficznych) firmy NVidia
- CUDA = specyfikacja abstrakcyjnego procesora równoległego (niekoniecznie graficznego)

CUDA

- CUDA = Compute Unified Device Architecture
- CUDA = uniwersalna architektura sprzętowo-programistyczna masowo równoległych, masowo wielordzeniowych procesorów (graficznych) firmy NVidia
- CUDA = specyfikacja abstrakcyjnego procesora równoległego (niekoniecznie graficznego)
- CUDA = abstrakcyjne API programowania procesorów zgodnych z CUDA izolujące programistów od „metalu”, tj. sprzętu

CUDA

- CUDA = Compute Unified Device Architecture
- CUDA = uniwersalna architektura sprzętowo-programistyczna masowo równoległych, masowo wielordzeniowych procesorów (graficznych) firmy NVidia
- CUDA = specyfikacja abstrakcyjnego procesora równoległego (niekoniecznie graficznego)
- CUDA = abstrakcyjne API programowania procesorów zgodnych z CUDA izolujące programistów od „metalu", tj. sprzętu
- CUDA = wieloplatformowe (Windows, Linux, MacOS), bezpłatne, uniwersalne środowisko programowania (SDK) procesorów firmy NVidia (kompilator, emulator, debuger, profiler, biblioteki) oparte na języku wysokiego poziomu („*C for CUDA*")

Zalety CUDA dla programisty C/C++

- Język wysokiego poziomu (minimalne rozszerzenie C/C++) + dokumentacja + SDK

Zalety CUDA dla programisty C/C++

- Język wysokiego poziomu (minimalne rozszerzenie C/C++) + dokumentacja + SDK
- Abstrakcja sprzętu – programy skompilowane dziś będą *optymalnie* działać na sprzęcie, który pojawi się za kilka lat; są niewrażliwe na zmianę sterowników etc.

Zalety CUDA dla programisty C/C++

- Język wysokiego poziomu (minimalne rozszerzenie C/C++) + dokumentacja + SDK
- Abstrakcja sprzętu – programy skompilowane dziś będą *optymalnie* działać na sprzęcie, który pojawi się za kilka lat; są niewrażliwe na zmianę sterowników etc.
- Automatyzacja zarządzania wątkami (których są tysiące!)

Zalety CUDA dla programisty C/C++

- Język wysokiego poziomu (minimalne rozszerzenie C/C++) + dokumentacja + SDK
- Abstrakcja sprzętu – programy skompilowane dziś będą *optymalnie* działać na sprzęcie, który pojawi się za kilka lat; są niewrażliwe na zmianę sterowników etc.
- Automatyzacja zarządzania wątkami (których są tysiące!)
- Niemal liniowa skalowalność – te same programy można uruchamiać na różnych procesorach lub ich „klastrach" (uwaga: obsługa klastrów GPU wymaga wielowątkowości kodu CPU)
- CUDA nie wymaga znajomości OpenGL/DirectX/etc. – jest stosunkowo łatwa do opanowania dla „numeryków"

Zalety CUDA dla programisty C/C++

- Język wysokiego poziomu (minimalne rozszerzenie C/C++) + dokumentacja + SDK
- Abstrakcja sprzętu – programy skompilowane dziś będą *optymalnie* działać na sprzęcie, który pojawi się za kilka lat; są niewrażliwe na zmianę sterowników etc.
- Automatyzacja zarządzania wątkami (których są tysiące!)
- Niemal liniowa skalowalność – te same programy można uruchamiać na różnych procesorach lub ich „klastrach" (uwaga: obsługa klastrów GPU wymaga wielowątkowości kodu CPU)
- CUDA nie wymaga znajomości OpenGL/DirectX/etc. – jest stosunkowo łatwa do opanowania dla „numeryków"
- CUDA umożliwia bezpośredni rendering (OpenGL lub DirectX)

Zalety CUDA dla programisty C/C++

- Język wysokiego poziomu (minimalne rozszerzenie C/C++) + dokumentacja + SDK
- Abstrakcja sprzętu – programy skompilowane dziś będą *optymalnie* działać na sprzęcie, który pojawi się za kilka lat; są niewrażliwe na zmianę sterowników etc.
- Automatyzacja zarządzania wątkami (których są tysiące!)
- Niemal liniowa skalowalność – te same programy można uruchamiać na różnych procesorach lub ich „klastrach" (uwaga: obsługa klastrów GPU wymaga wielowątkowości kodu CPU)
- CUDA nie wymaga znajomości OpenGL/DirectX/etc. – jest stosunkowo łatwa do opanowania dla „numeryków"
- CUDA umożliwia bezpośredni rendering (OpenGL lub DirectX)
- CUDA udostępnia ogólnie znany model pamięci (jak w języku C) i standardowe operacje matematyczne (+, −, ∗, /, sin, sqrt, exp,. . .)

Zalety CUDA dla podatnika/płatnika

Źródło: Estonia Donates

- 4 podwójne karty GTX295 GPU (8 procesorów GT 200)
- 8 TFLOPS mocy obliczeniowej (1560 procesorów strumieniowych)
- 2 zasilacze 850 W ;-)
- cena kart graficznych: 7 000 – 8 000 zł

Zalety CUDA dla podatnika/płatnika

- SGI Altix 3700 (centra superkomputerowe we Wrocławiu, Gdańsku, Poznaniu, Krakowie)
- Rok 2007: „Najnowocześniejszy komputer w Polsce"
- 0.768 TFLOPS mocy obliczeniowej (128 procesorów IA-64)
- zasilanie – ?? kW
- cena: milion zł

Źródło: task.gda.pl

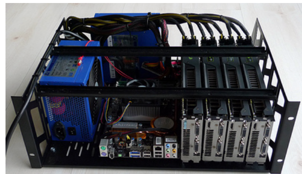
Zalety CUDA dla podatnika/płatnika

CPU: SGI Altix 3700

- „Najnowocześniejszy komputer w Polsce (2007)” (Wrocław, Gdańsk, Poznań, Kraków)
- 128 × Itanium IA-64
- 0.768 TFLOPS fp2/fp64
- zasilanie – ?? kW
- cena: milion zł

GPU: PC + 4 × GTX 295

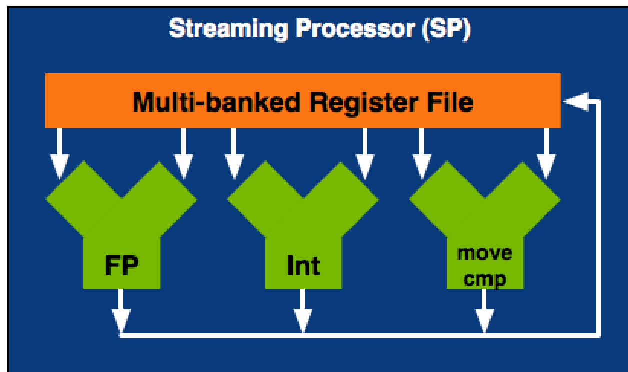
- Komputer osobisty
- 8 × GT 200 (1560 procesorów strumieniowych)
- 8 TFLOPS fp32, 0.6 TFLOPS (fp64)
- zasilanie: 2 × 850 W
- cena: 7 000 – 8 000 zł + dobry PC + zasilacz



Część 2

1. Wstęp: GPU a CPU
   - CPU
   - GPU
2. CUDA
   - Co to jest CUDA?
   - Architektura procesora GT 200
   - CUDA API
3. Przyszłość GPU

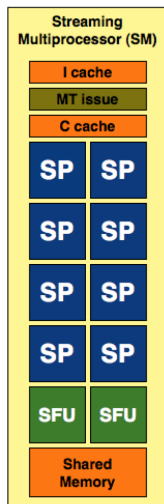
Procesor strumieniowy (SP)



Źródło: anandtech.com

- Streaming processor (SP, *scalar processor*) jest dość prymitywny
- Przetwarza jeden wątek programu
- Ale to z takich „zer” tworzą się miliony („teraFLOPS-y”)...

Multiprocesor strumieniowy (SM)



Źródło: anandtech.com

- 8 równoległych procesorów strumieniowych (SP)
- 2 jednostki dla funkcji specjalnych (SFU), np. sin, cos,. . .
- 1 jednostka fp64 (brak na rysunku)
- jednostka sterująca (MT Issue, *multithreaded instruction fetch and issue*)
- pamięć podręczna instrukcji (I Cache)
- pamięć podręczna danych (Constant Cache), dla SP tylko do odczytu
- pamięć współdzielona (*shared memory*, 16 KB)
- 16 384 rejestrów

Multiprocesor strumieniowy (SM)

- Wątki są wykonywane jednocześnie w pojedynczych SP (ten sam kod!)
- Wątki są automatycznie grupowane w wiązki przetwarzane w tym samym SM, tzw. bloki (*blocks*)
- Kolejność i miejsce przetwarzania różnych bloków jest nieokreślona
- Wątki z różnych bloków nie mogą wymieniać (efektywnie) informacji

Zapamiętaj!

Rozłączność bloków gwarantuje skalowalność!

Multiprocesor strumieniowy (SM)

- Wątki w tym samym bloku mogą korzystać z programowalnych pamięci Constant Cache i **pamięci dzielonej** równie szybkich jak rejestry procesora

Zapamiętaj!

Efektywne posługiwanie się różnego rodzajami pamięci (szczególnie pamięci dzielonej) to podstawowa umiejętność programisty CUDA!

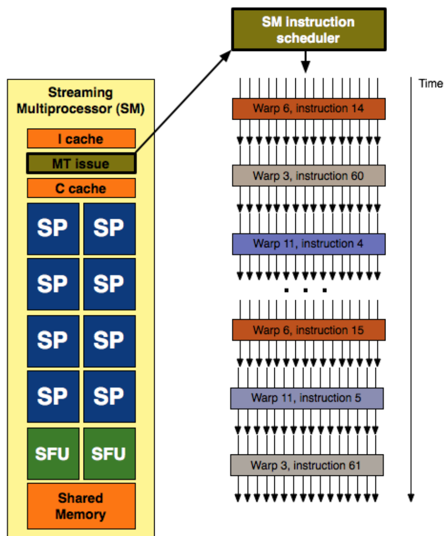
Multiprocesor strumieniowy (SM)

- Liczba wątków aktywnych w multiprocesorze (maks. 1024) znacznie przekracza liczbę fizycznie dostępnych SP (większość „śpi", ale zajmuje rejestry multiprocesora)
- Wiązka wątków fizycznie obsługiwanych w danej chwili (32) to warp (wiązka)
- Muliprocesor może jednocześnie obsługiwać do 8 bloków i do 32 warpów; maksymalna wielkość bloku: 512 wątków.

Zapamiętaj!

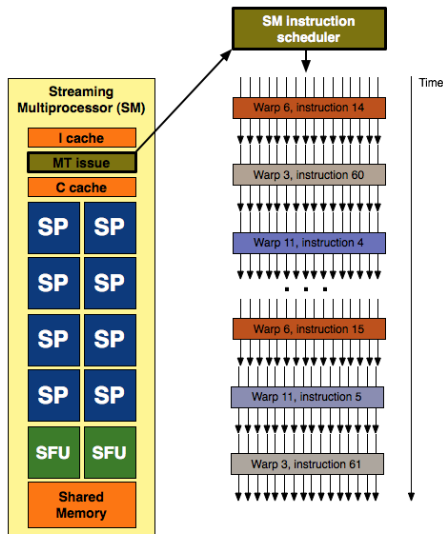
Nadwyżka liczby wątków w bloku w stosunku do liczby SP w SM maskuje latencję w dostępie do danych globalnych. Cena: rejestry multiprocesora.

Multiprocesor strumieniowy (SM)



- Wiązka (warp) to 32 wątki przetwarzane w danym cyklu przez SM (2 takty zegara MT Issue = 4 takty zegara SP; kontroler „widzi' 16 procesorów SP, tzw. *half-warp*)
- Kontroler SM wybiera do wykonania kolejno różne wiązki (karuzela)
- Wszystkie 32 wątki w wiązce wykonują tę samą instrukcję

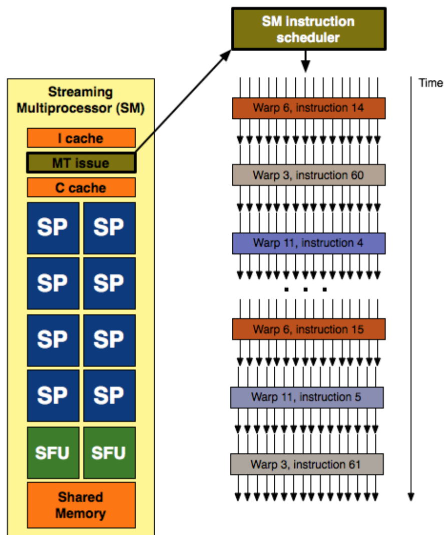
Multiprocesor strumieniowy (SM)



Jeżeli program zawiera `if`/`else` `goto`...

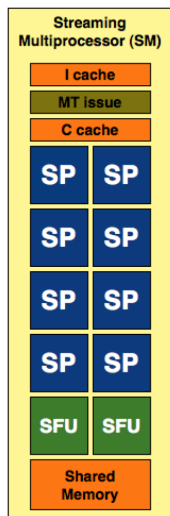
- Wątki w różnych wiązkach mogą swobodnie wykonywać różne gałęzie danego programu
- Jeżeli różne wątki danej wiązki zechcą wykonać różne instrukcje, instrukcje wszystkich gałęzi zostaną kolejno wykonane przez wszystkie wątki, ale część z nich będzie maskowana („unieważniana")

Multiprocesor strumieniowy (SM)



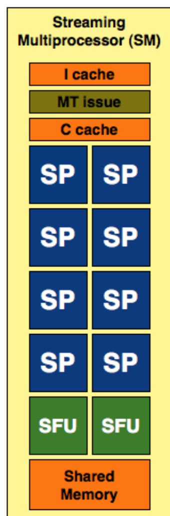
- Wiązka (warp) to 32 wątki
- Każdy SM może łącznie przetwarzać do 32 wiązek (1024 wątków)
- GT 200 może więc jednocześnie przetwarzać 1024*30 = 30720 wątków

Multiprocesor strumieniowy (SM)



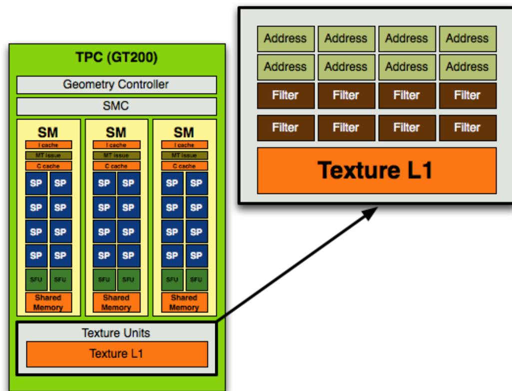
- Komunikacja z pamięcią karty: magistrala 512 bitowa (16 floatów na 30 multiprocesorów i 240 SP)
- Wąskie gardło: latencja DRAM (400-600 cykli zegara!)
- Optymalizacja: „burst mode" – jednoczesne wczytywanie danych z ciągłego obszaru pamięci (np. 16 floatów)
- Optymalizacja: dużo liczyć, by zamaskować latencję DRAM
- Optymalizacja: właściwe „data alignment"
- Optymalizacja: „coalesced data transfer" (złączony transfer danych)

Multiprocesor strumieniowy (SM)



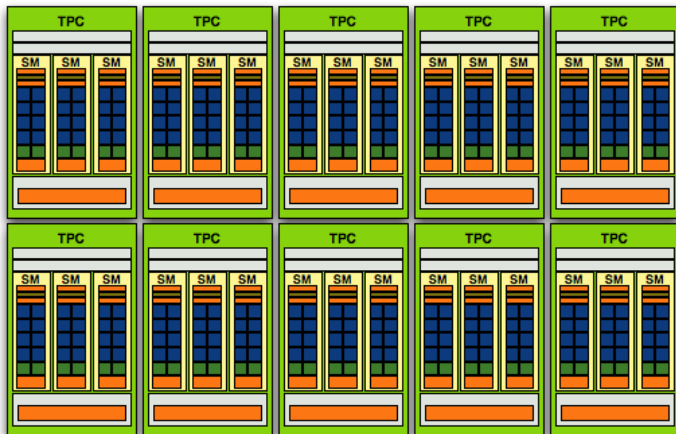
- Aż dziesięć slajdów poświęciłem multiprocesorowi...
- Bo to fundament architektury CUDA

Klaster procesorów strumieniowych (TPC)



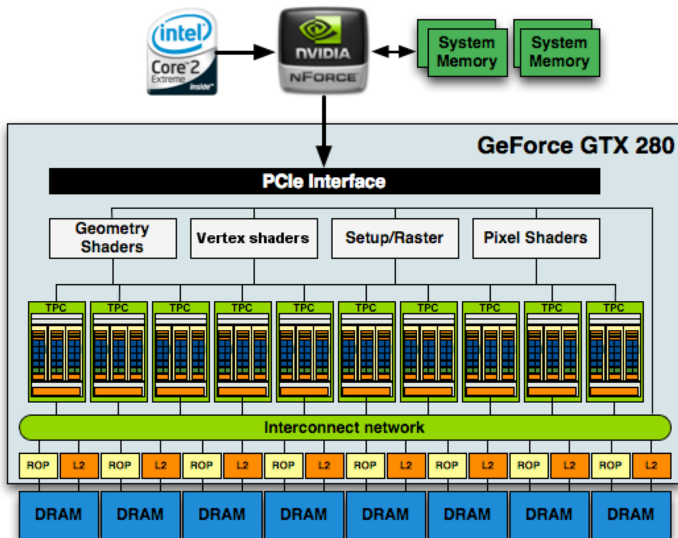
- Multiprocesory łączy się w klastry (*Texture/Processor Cluster*)
- Buforowana pamięć tekstur – ulubione źródło danych CUDA-programisty

Macierz procesorów strumieniowych (SPA)



Triumf modularyzacji: 10 TPC = 30 SM = 240 SP + 60 SFU

GT 200 w pełnej okazałości



Porównanie GT 80 i GT 200

	8800 GTX	GTX 280	postęp
rdzenie SP	128	240	88%
tekstury	64t/clk	80t/clk	25%
ROP	12p/clk	32p/clk	167%
precyzja	fp32	fp32, fp64	
GFLOPS	518	933	80%
przepustowość FB	86 GB/s	142 GB/s	65%
przepustowość PCI-E	6,4 GB/s	12,8 GB/s	100%

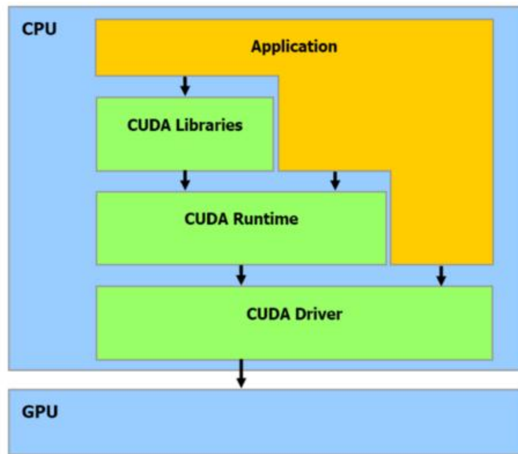
- Wąskie gardło: przepustowość złącza PCI-Express (łączy CPU z GPU)

1. Wstęp: GPU a CPU
- CPU
- GPU

2. CUDA
- Co to jest CUDA?
- Architektura procesora GT 200
- CUDA API

3. Przyszłość GPU

CUDA API



Źródło: tomshardware.com

- CUDA Driver: niskopoziomowy
- CUDA Runtime: bliższy języka C
- Biblioteki:
  - CUBLAS (Basic Linear Algebra)
  - CUFFT (Fast Fourier Transform)

Słowniczek

- host: CPU (dokładniej: wątek na CPU)
- device: GPU (każdy GPU jest kontrolowany przez innego hosta)
- kernel: program uruchomiony na GPU
- thread: wątek
- warp: pakiet 32 wątków przetwarzanych w tym samym multiprocesorze w tym samym cyklu zegara
- block: grupa wątków załadowanych do rejestrów tego samego multiprocesora, komunikująca się ze sobą poprzez pamięć dzieloną i podlegająca synchronizacji; maksymalny rozmiar: 512 wątków; jeden multiprocesor może przetwarzać kilka bloków naraz (skalowalność!).
- grid: niemal dowolnie duża grupa bloków przetwarzanych w różnych multiprocesorach
- wywołanie asynchroniczne: wywołanie funkcji (na GPU) i natychmiastowa kontynuacja obliczeń na CPU

Kernel

- Kernel to program wykonywany na GPU asynchronicznie
- W kodzie programu (C/C++) wyróżniony słowem kluczowym `__global__`
- Nie może zawierać rekurencji/zmiennych statycznych
- Nie powinien „konsumować" wielu zmiennych lokalnych (rejestry!)
- Powinien zawierać możliwie dużo instrukcji matematycznych (bo właśnie to GPU robi najlepiej)
- Nie może zwrócić do hosta wartości

Wątek

- Wątek na GPU jest „lekki" – łatwo go utworzyć i zniszczyć
- Wątki na GPU łatwo zsynchronizować
- Stosując odpowiedni styl programowania, można uniknąć blokady wątków („deadlock")
- Wątki tworzy się wokół danych a nie algorytmów
- Wątki można w programie zidentyfikować poprzez wartości 3 nowych „parametrów kluczowych":
  - `threadIdx` (numer/położenie wątku w bloku)
  - `blockIdx` (numer/położenie bloku w siatce)
  - `blockDim` (rozmiar bloku)
- Powyższe „parametry" mogą mieć trzy składowe (x, y i z)
- Na ich podstawie zwykle każdemu wątkowi jednoznacznie przyporządkowuje się w programie elementy tablic (1D, 2D lub 3D), na których wątek operuje

Grid, block

- Blok to grupa wątków, które mogą wymieniać informację za pośrednictwem szybkiej pamięci dzielonej (*shared memory*) i podlegające synchronizacji
- Bloków powinno być duuuużo
- Siatka (*grid*) to sposób podziału wątków na bloki. Dzięki niej programy są niezależne od sprzętu i łatwo skalowalne (programista nie musi wiedzieć, ile bloków na raz może przetwarzać GPU)
- Rozmiar siatki uwarunkowany jest zwykle rozmiarem danych do przetworzenia, rozmiar bloku – charakterystyką sprzętu (niestety!)

Nowe typy danych

- float1, float2, float3, float4
- char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4
- double2
- etc.
- dim3

Sa to typy wektorowe o składowych x (y, z, w).
Przykład:

```
float4 z;
float p = z.w + z.x;
```

Wywołanie kernela

- Kernel wywoływany jest asynchronicznie z hosta
- Specjalna składnia:

```
nazwa_funkcji<<<dimGrid, dimBlock>>>(param1, param2,...);
```

Przykład:

```
float A[N][N];
float B[N][N];
float C[N][N];
...
dim3 dimBlock(16, 16);
dim3 dimGrid( (N + dimBlock.x - 1) / dimBlock.x,
              (N + dimBlock.y - 1) / dimBlock.y );
...
matAdd<<<dimGrid, dimBlock>>>(A, B, C);
```

Przykład kernela

```
__global__
void matAdd(float A[N][N], float B[N][N], float C[N][N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < N)
     C[i][j] = A[i][j] + B[i][j];
}
```

- 256 wątków na blok o strukturze 2D (16*16)
- `blockDim` podaje rozmiar bloku
- `blockIdx` podaje pozycję bloku na siatce
- `threadIdx` podaje położenie wątku w bloku

Synchronizacja

```
__syncthreads()
```

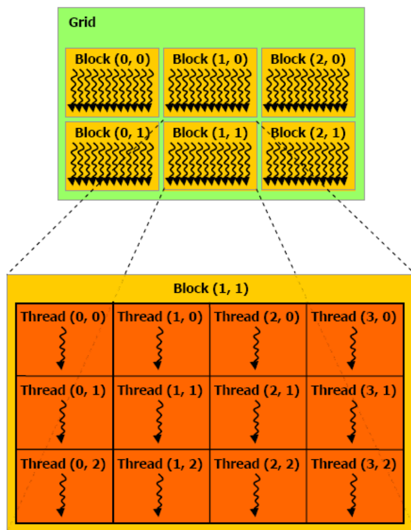
- Funkcja `__syncthreads()` synchronizuje wszystkie wątki *bloku*
- Synchronizacja polega na wstrzymaniu wątków bloku aż wszystkie zechcą wykonywać tę samą instrukcję `__syncthreads()`
- W normalnych warunkach synchronizacja jest „tania”
- Unikaj synchronizacji w blokach `if...else`, pętlach o nieokreślonej liczbie kroków etc.

Funkcje atomowe

```
// Thread 0 of each block signals that it is done
unsigned int value = atomicInc(&count, gridDim.x);
```

- Funkcje atomowe umożliwiają bezpieczny, serializowany dostęp wielu wątków bloku do tego samego fragmentu pamięci globalnej/dzielonej.
- Nie ma żadnej gwarancji co do kolejności, w jakiej poszczególne wątki wykonają operację atomową.

Ułożenie bloków w siatce

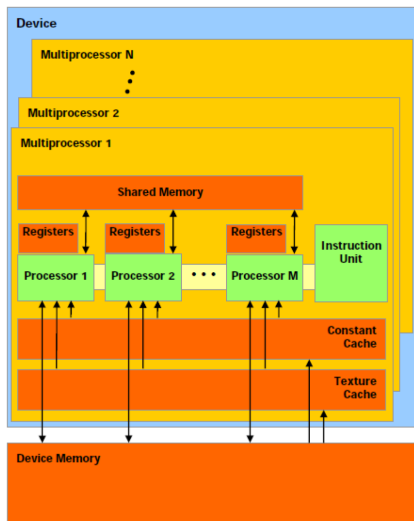


- Siatki i bloki mogą mieć geometrię (1D, 2D, 3D)
- Dobór właściwej geometrii może przyspieszyć dostęp do danych (poprzez optymalizację cache-ów)

Optymalizacja

- CUDA SDK zawiera profiler
- Zalecenia programistyczne
  - Organizuj program wokół przepływu danych, a nie wokół algorytmów
  - Optymalizuj wykorzystanie różnych rodzajów pamięci, szczególnie buforowanych na poziomie L1/L2 (shared memory, constant memory, rejestry, bufor tekstur)
  - Twórz bloki o dużej liczbie wątków (tu trzeba eksperymentować)
  - Twórz kernele o dużej pracochłonnosci obliczeniowej i niewielkim zapotrzebowaniu na rejestry

Klucz do sukcesu: wykorzystanie pamięci podręcznych



- Rejestry (16 384 na multiprocesor)
- Shared memory – ultraszybka, całkowicie programowalna, dostęp swobodny, mała (16 KB w 16 bankach)
- Constant memory – ultraszybka, tylko do odczytu (8KB cache / MP, łącznie 64 KB)
- Texture memory – buforowany magazyn gigabajtów danych (cache rzędu 6-8 KB / MP)

Compute capability

- *Compute capability* to specyfikacja gwarantowanych możliwości sprzętu.
- Obecnie: *Compute capability* 1.0, 1.1, 1.2 i 1.3.
- Nowe generacje sprzętu – coraz większa *Compute capability*
- Kompatybilność wsteczna
- Przykład: tylko urządzenia z *Compute capability* 1.3 mogą przetwarzać liczby w podwójnej precyzji
- *Compute capability* 1.3: GeForce GTX 260, 280, 285, 295; Tesla S1070, C1060, Quadro Plex 2200 D2, FX 5800, FX 4800
- *Compute capability* 1.3 stanieje, gdy pojawi sie *Compute capability* 1.4 ;-)

FMAD (Fused multiply and add)

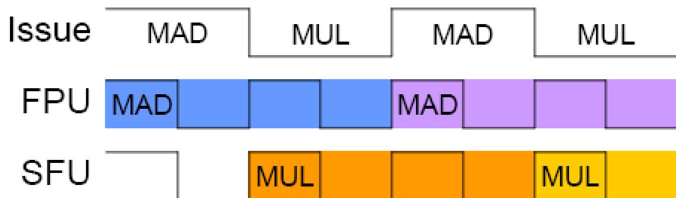
- FMAD to operacja polegająca na pomnożeniu zawartości dwóch rejestrów i dodaniu wyniku do trzeciego rejestru:

$$a \leftarrow a + b \times c$$

- „*Fused*" oznacza, że FMAD traktowana jest jak jedna operacja – tylko jedno zaokrąglenie wyniku!
- FMAD przyspiesza i poprawia dokładność obliczeń:
  - Iloczynów skalarnych
  - Iloczynów macierzy
  - Wartości wielomianów
- FMAD występuje w standardzie IEEE 754-2008 i języku C99
- FMAD zaimplementowano sprzętowo m.in. w procesorach POWER, SPARC, Itanium, GT 200

Podwójne instrukcje (Dual issue)

- Jednostki obliczeniowe (SP, SFU, FP64) są taktowane 2 razy szybciej niż jednostka sterująca (*MT Issue*)
- Najszybsze operacje obliczeniowe trwają 2 takty zegara MT Issue = 4 takty zegara SP (bo 4*8 wątków = warp).
- SFU mogą działać niezależnie od SP
- ⇒ multiprocesor może jednocześnie przetwarzać „2 potoki” instrukcji



Źródło: beyond3d.com

- Niestety, jednostki FP64 nie mogą pracować w trybie *dual issue*

Giga FLOPS-y

- FLOPS (*Floating point operation per second*) – liczba operacji zmiennoprzecinkowych na sekundę
- W trybie *dual-issue* każdy SP może wykonać 1 FMAD, a SFU – 1 MUL = dodawanie i dwa mnożenia = 3 operacje arytmetyczne (fp32)
- W trybie fp64 pojedynczy multiprocesor może wykonać 1 FMAD = 2 operacje arytmetyczne w podwójnej precyzji
- Liczba FLOPS-ów fp32 = częstot. zegara $\times$ liczba rdzeni SP $\times$ 3
- Liczba FLOPS-ów fp64 = częstot. zegara $\times$ liczba jednostek fp64 $\times$ 2

Ge Force GTX 280

1296 MHz $\times$ 240 $\times 3 =$ 933 GFLOPS (fp32)
1296 MHz $\times$ 30 $\times 2 =$ 77.8 GFLOPS (fp64)

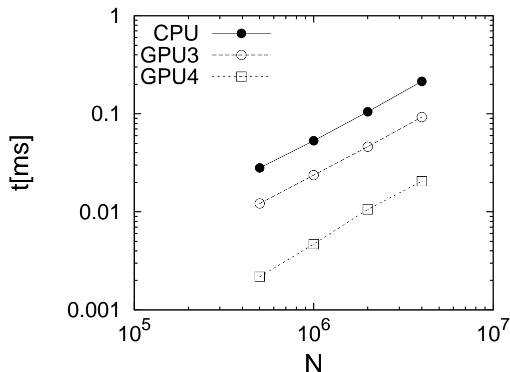
Lepiej mnożyć niż dzielić

Wydajność instrukcji (w jednym multiprocesorze)

`float a, x, y; int n, m; double A, X, Y`

Instrukcja	liczba instrukcji w cyklu zegara
$x + y$, $x * y$, a += x*y (FMAD)	8
$1/x$	2
x/y	0,88
$1/\sqrt{x}$	2
$\sqrt{x}$	1
$__logf(x)$	2
__sinf(x), __cosf(x), __expf(x)	1
sinf(x), cosf(x), expf(x)	< 1, dużo rejstrów
$n + m$, $n \times m$	8
n/m, $n\%m$	bardzo powoli
min, max, ==, operatory bitowe	8
X + Y, X * Y, A += X * Y	1 (?)

Nasz pierwszy test – problem klasy $O(N)$

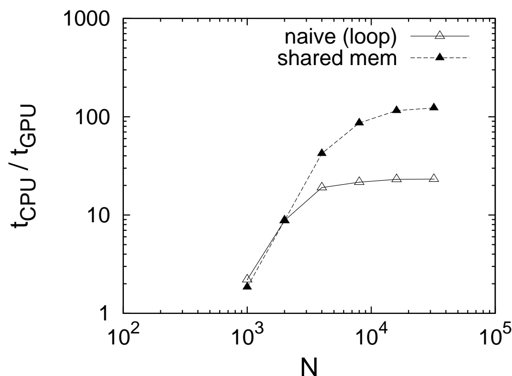


Źródło: Maciej Matyka

- $N \sim$ liczba przetwarzanych danych
- $t =$ czas wykonania zadania
- CPU – wiadomo
- GPU3 – zwykłe przepisanie CPU na GPU
- GPU4 – reorganizacja dostępu do globalnej pamięci karty

Nasz drugi test – problem klasy $O(N^2)$

Rola pamięci dzielonej



Źródło: Maciej Matyka

- *naive*:
  zwykła pętla jak dla CPU
- *shared mem*:
  kod z optymalnym wykorzystaniem pamięci dzielonej

Część 3

1. Wstęp: GPU a CPU
   - CPU
   - GPU
2. CUDA
   - Co to jest CUDA?
   - Architektura procesora GT 200
   - CUDA API
3. Przyszłość GPU

Futurologia – GT 300

Podobno GT 300 (jesień 2009) ma mieć następujące cechy:

- Rdzenie skalarne (SP) mają działać w trybie MIMD (*Multiple Instruction Multiple Data*) zamiast SIMD (*Single Instruction Multiple Data*)
- Dlatego mają bardziej przypominać rdzenie współczesnych CPU
- Żegnaj GPU – witaj cGPU! (cGPU = GPU bliższy CPU)

Futurologia – GT 300

Podobno GT 300 (jesień 2009) ma mieć następujące cechy:

- Klastry mają zawierać 4 multprocesory, czyli 32 (zamiast 24) rdzeni SP
- Klastrów ma być 16 – to daje $16 \times 32 = 512$ rdzeni!
- 512-bitowy interfejs pamięci RAM typu GDDR5 (teraz: GDDR3) z przepustowością ok. 256 GB/s (teraz: 142 GB/s)
- Moc obliczeniowa – ok. 2–3 TFLOPS
- Umożliwienie komunikacji między rdzeniami na poziomie klastra (a nie tylko multiprocesora) ułatwi zastosowania numeryczne
- Nawet 6-krotna poprawa wydajności w arytmetyce podwójnej precyzji
- Technologia 40 nm
- CUDA 3.0

Pierwsze proroctwo Sweeney'a

*„**DirectX 10 is the last DirectX graphics API that is truly relevant to developers.***
In the future, developers will tend to write their own renderers that will use both the CPU and the GPU – using graphics processor programming language rather than DirectX."

Tim Sweeney

Drugie proroctwo Sweeney'a

„It is hard to say at what point we are going to see graphics hardware being able to understand C++ code. [...] Then, [...] you can [...] recompile the kernel for a GPU and actually run the Linux kernel off the GPU – running entirely by itself. ***Then, the boundary between the CPU and the GPU will become just a matter of performance trade-offs.****"*

Tim Sweeney

Linki

- CUDA Zone (NVidia)

Literatura

- NVIDIA CUDA Programming Guide 2.2
- F. Abi-Chahla, Nvidia's CUDA: The End of the CPU?
- D. Kirk, W. Hwu, CUDA Textbook
- Materiały dydaktyczne MIT
- A. L. Shimpi, D. Wilson, NVIDIA's 1.4 Billion Transistor GPU: GT200 Arrives as the GeForce GTX 280 & 260
- R. Farber, CUDA, Supercomputing for the Masses
- David Kanter, NVIDIA's GT200: Inside a Parallel Processor
- NVIDIA GT200 GPU and Architecture Analysis
- J. Piekarski, CUDA się zdarzają, czyli programowanie GPGPU (Software Developer's Journal 5/2009)