PPaDS MMXXI

Matúš Jókay, Château de la Mûre
matus.jokay@stuba.sk

uim.fei.stuba.sk/predmet/i-ppds
konzultácie elektronicky
mail, discord

Načo GPU?

- Dlhý čas boli grafické karty bez možnosti vykonávania všeobecných výpočtov (GPGPU)
- Mali fixovanú funkcionalitu (OpenGL, DirectX)

Načo GPU?

- Dlhý čas boli grafické karty bez možnosti vykonávania všeobecných výpočtov (GPGPU)
- Mali fixovanú funkcionalitu (OpenGL, DirectX)

- Čo v prípade prevodu obrázka z RGB do šedej škály?

Načo GPU?

- Čo v prípade prevodu obrázka z RGB do šedej škály? Napr. 1920x1080

Načo GPU?

- Čo v prípade prevodu obrázka z RGB do šedej škály? Napr. 1920x1080

- CPU musí vykonávať konverziu sekvenčne (bod po bode); a teda čas výpočtu závisí od rozmerov obrázka

Načo GPU?

- Čo v prípade prevodu obrázka z RGB do šedej škály? Napr. 1920x1080

- CPU musí vykonávať konverziu sekvenčne (bod po bode); a teda čas výpočtu závisí od rozmerov obrázka
- V našom príklade treba 2 073 600 výpočtov

Načo GPU?

- Čo v prípade prevodu obrázka z RGB do šedej škály? Napr. 1920x1080 = 2 073 600

- Vykonáva sa tá istá operácia, ale nad rôznymi údajmi!

Načo GPU?

- Čo v prípade prevodu obrázka z RGB do šedej škály? Napr. 1920x1080 = 2 073 600

- Vykonáva sa tá istá operácia, ale nad rôznymi údajmi! SIMD!!!

Načo GPU?

- Čo v prípade prevodu obrázka z RGB do šedej škály? Napr. 1920x1080 = 2 073 600

- Vykonáva sa tá istá operácia, ale nad rôznymi údajmi! SIMD!!!
- Ideálne pre GPU; na GPU vieme spustiť niekoľko miliónov vlákien súčasne (tzv. workers)

Načo GPU?

- Čo v prípade prevodu obrázka z RGB do šedej škály? Napr. 1920x1080 = 2 073 600

- Vykonáva sa tá istá operácia, ale nad rôznymi údajmi! SIMD!!!
- Ideálne pre GPU; na GPU vieme spustiť niekoľko miliónov vlákien súčasne (tzv. workers) (t.j. pri rôznych rozmeroch obrázkov bude výpočet vždy v O(1)!!!)

CUDA

CUDA – zdroje

- https://www.tutorialspoint.com/cuda

- https://developer.nvidia.com/cuda-zone

CUDA

- **C**ompute **U**nified **D**evice **A**rchitecture

CUDA

- **C**ompute **U**nified **D**evice **A**rchitecture

- Rozšírenie C-čka
- API model určený na paralelné výpočty

CUDA

- **C**ompute **U**nified **D**evice **A**rchitecture

- Rozšírenie C-čka
- API model určený na paralelné výpočty

- Vytvorený firmou Nvidia

CPU a Moorov zákon

CPU a Moorov zákon

- Moorov zákon
  - zložitosť integrovaných obvodov sa zdvojnásobuje každých 18 mesiacov, pričom cena ostáva konštantná

CPU a Moorov zákon

- Moorov zákon
  - zložitosť integrovaných obvodov sa zdvojnásobuje každých 18 mesiacov, pričom cena ostáva konštantná
  - 1965

CPU a Moorov zákon

- Moorov zákon
  - zložitosť integrovaných obvodov sa zdvojnásobuje každých 18 mesiacov, pričom cena ostáva konštantná
  - 1965
- Platil 40 rokov…

CPU a Moorov zákon

- Moorov zákon
  - zložitosť integrovaných obvodov sa zdvojnásobuje každých 18 mesiacov, pričom cena ostáva konštantná
  - 1965
- Platil 40 rokov…
  - Zvyšovanie výkonu sa v súčasnosti deje nie miniaturizáciou

Paralelizácia CPU

Paralelizácia CPU

- Kroky CPU potrebné k vykonaniu inštrukcie

Paralelizácia CPU

- Kroky CPU potrebné k vykonaniu inštrukcie
  - Instruction Fetch

Paralelizácia CPU

- Kroky CPU potrebné k vykonaniu inštrukcie
  - Instruction Fetch
  - Instruction Decode

Paralelizácia CPU

- Kroky CPU potrebné k vykonaniu inštrukcie
  - Instruction Fetch
  - Instruction Decode
  - Memory Access (operand(s), result)

Paralelizácia CPU

- Kroky CPU potrebné k vykonaniu inštrukcie
  - Instruction Fetch
  - Instruction Decode
  - Memory Access (operand(s), result)
  - Instruction Execute

Paralelizácia CPU

- Kroky CPU potrebné k vykonaniu inštrukcie
  - Instruction Fetch
  - Instruction Decode
  - Memory Access (operand(s), result)
  - Instruction Execute
  - Register Write-Back

Paralelizácia CPU

- Kroky CPU potrebné k vykonaniu inštrukcie
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - Memory Access (operand(s), result) (Mem)
  - Instruction Execute (Ex)
  - Register Write-Back (WB)

Paralelizácia CPU

- Kroky CPU potrebné k vykonaniu inštrukcie
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - Memory Access (operand(s), result) (Mem)
  - Instruction Execute (Ex)
  - Register Write-Back (WB)

- Základná 5-fázová RISC architektúra

Paralelizácia CPU – Pipelining

Paralelizácia CPU – Pipelining

- Paralelizácia na úrovni inštrukcií vykonávaných v CPU

Paralelizácia CPU – Pipelining

- Paralelizácia na úrovni inštrukcií vykonávaných v CPU
- Pipeline

Paralelizácia CPU – Pipelining

- Paralelizácia na úrovni inštrukcií vykonávaných v CPU
- Pipeline – počet inštrukcií, ktoré vie CPU obslúžiť behom jedného taktu

Paralelizácia CPU – Pipelining

- Paralelizácia na úrovni inštrukcií vykonávaných v CPU
- Pipeline – počet inštrukcií, ktoré vie CPU obslúžiť behom jedného taktu
- CPU bez pipeline obsluhuje každú inštrukciu v krokoch uvedených vyššie (t.j. sekvenčne!)

Paralelizácia CPU – Pipelining

- Paralelizácia na úrovni inštrukcií vykonávaných v CPU
- Pipeline – počet inštrukcií, ktoré vie CPU obslúžiť behom jedného taktu
- CPU bez pipeline obsluhuje každú inštrukciu v krokoch uvedených vyššie (t.j. sekvenčne!)
- Takže na vykonanie jednej inštrukcie potrebuje CPU približne 5 taktov

Paralelizácia CPU – Pipelining

- Počas piatich fáz spracovania inštrukcie niektoré časti CPU musia vždy čakať, kým sa nedokončí celé spracovanie

Paralelizácia CPU – Pipelining

- Počas piatich fáz spracovania inštrukcie niektoré časti CPU musia vždy čakať, kým sa nedokončí celé spracovanie
- Tu je priestor na súčasné “rozpracovanie” viacerých inštrukcií naraz

Paralelizácia CPU – Pipelining

- Počas piatich fáz spracovania inštrukcie niektoré časti CPU musia vždy čakať, kým sa nedokončí celé spracovanie
- Tu je priestor na súčasné “rozpracovanie” viacerých inštrukcií naraz

- Tento prístup sa nazýva ILP – Instruction Level Parallelism

Paralelizácia CPU – Pipelining

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Paralelizácia CPU – Pipelining

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

- Spracovanie prvej inštrukcii trvá 5 taktov
- Dokončenie spracovania každej z nasledujúcich sa deje v každom takte

Paralelizácia CPU – Pipelining

- Takéto vykonávanie ale nie je vždy úplne správne…

Paralelizácia CPU – Pipelining

- Takéto vykonávanie ale nie je vždy úplne správne…

- Uvažujme, že CPU vykoná iba nasledovné 2 inštrukcie:
  - I1 – ADD 1 to R5
  - I2 – COPY R5 to R6

Paralelizácia CPU – Pipelining

- CPU vykoná iba nasledovné 2 inštrukcie:
  - I1 – ADD 1 to R5
  - I2 – COPY R5 to R6

Paralelizácia CPU – Pipelining

- CPU vykoná iba nasledovné 2 inštrukcie:
  - I1 – ADD 1 to R5
  - I2 – COPY R5 to R6

- I1 začína v T1, v T5 sa aktualizuje hodnota R5

Paralelizácia CPU – Pipelining

- CPU vykoná iba nasledovné 2 inštrukcie:
  - I1 – ADD 1 to R5
  - I2 – COPY R5 to R6

- I1 začína v T1, v T5 sa aktualizuje hodnota R5
- I2 začína v T2, hodnota R5 sa číta v T3 (!!!), R6 sa aktualizuje v T6

Paralelizácia CPU – Pipelining

- CPU vykoná iba nasledovné 2 inštrukcie:
  - I1 – ADD 1 to R5
  - I2 – COPY R5 to R6

- I1 začína v T1, v T5 sa aktualizuje hodnota R5
- I2 začína v T2, hodnota R5 sa číta v T3 (!!!), R6 sa aktualizuje v T6

- Do R6 sa dostane nesprávna hodnota!

Paralelizácia CPU – Pipelining

- Takejto situácii sa hovorí “hazard”

Paralelizácia CPU – Pipelining

- Takejto situácii sa hovorí “hazard”; keďže sa jedná o údaje, tak je to údajový hazard

Paralelizácia CPU – Pipelining

- Takejto situácii sa hovorí “hazard”; keďže sa jedná o údaje, tak je to údajový hazard

- Rieši sa na úrovni kompilátora!!!

Paralelizácia CPU – Superscalar

Paralelizácia CPU – Superscalar

- Rozdiel medzi Pipelining a Superscalar

Paralelizácia CPU – Superscalar

- Rozdiel medzi Pipelining a Superscalar
  - Pri pipeline vždy iba jedna inštrukcia môže byť v jednom z piatich vyššie menovaných stavov

Paralelizácia CPU – Superscalar

- Rozdiel medzi Pipelining a Superscalar
  - Pri pipeline vždy iba jedna inštrukcia môže byť v jednom z piatich vyššie menovaných stavov
  - Pri superscalar architektúre môže byť viac inštrukcií v jednom z týchto stavov; superskalárna architektúra pridáva viacero vykonávacích jednotiek inštrukcií na čip

Paralelizácia CPU – Superscalar

- Rozdiel medzi Pipelining a Superscalar
  - Pri pipeline vždy iba jedna inštrukcia môže byť v jednom z piatich vyššie menovaných stavov
  - Pri superscalar architektúre môže byť viac inštrukcií v jednom z týchto stavov; superskalárna architektúra pridáva viacero vykonávacích jednotiek inštrukcií na čip

- Superscalar – aspoň 2 ALU!

Paralelizácia CPU – Superscalar

- Nejedná sa o paralelizáciu na úrovni vlákien či dokonca procesov!

Paralelizácia CPU – Superscalar

- Nejedná sa o paralelizáciu na úrovni vlákien či dokonca procesov!
- Aj keď nám CPU dokáže skutočne paralelne vykonávať inštrukcie, stále máme iba jeden register IP (Instruction Pointer)

Paralelizácia CPU – Superscalar

- Nejedná sa o paralelizáciu na úrovni vlákien či dokonca procesov!
- Aj keď nám CPU dokáže skutočne paralelne vykonávať inštrukcie, stále máme iba jeden register IP (Instruction Pointer)

- Takže sa jedná o za sebou idúce inštrukcie toho istého procesu (vlákna)

Vlastnosti CPU

- Vynikajúci výkon pre SISD

Vlastnosti CPU

- Vynikajúci výkon pre SISD (horší pre SIMD)

Vlastnosti CPU

- Vynikajúci výkon pre SISD (horší pre SIMD)
- Inštrukčná sada CPU je bohatšia oproti GPU

Vlastnosti CPU

- Vynikajúci výkon pre SISD (horší pre SIMD)
- Inštrukčná sada CPU je bohatšia oproti GPU
- Komplexnejšia ALU, lepšia logika predikcie

Vlastnosti CPU

- Vynikajúci výkon pre SISD (horší pre SIMD)
- Inštrukčná sada CPU je bohatšia oproti GPU
- Komplexnejšia ALU, lepšia logika predikcie
- Sofistikovanejšia cache a pipeline schéma

Vlastnosti CPU

- Vynikajúci výkon pre SISD (horší pre SIMD)
- Inštrukčná sada CPU je bohatšia oproti GPU
- Komplexnejšia ALU, lepšia logika predikcie
- Sofistikovanejšia cache a pipeline schéma
- Cyklus inštrukcie rýchlejší než GPU

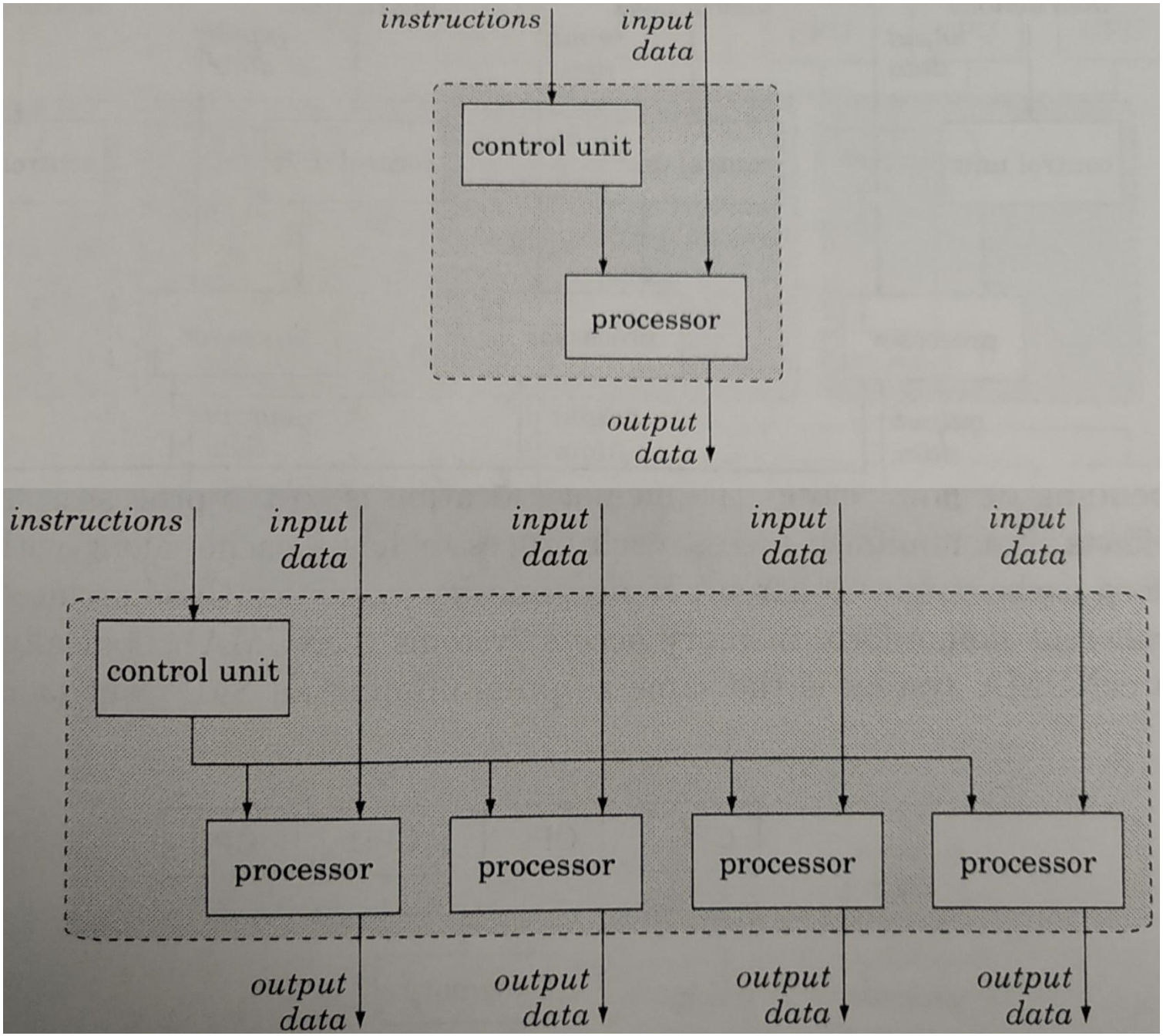
Typy výpočtov – Flynn tax.

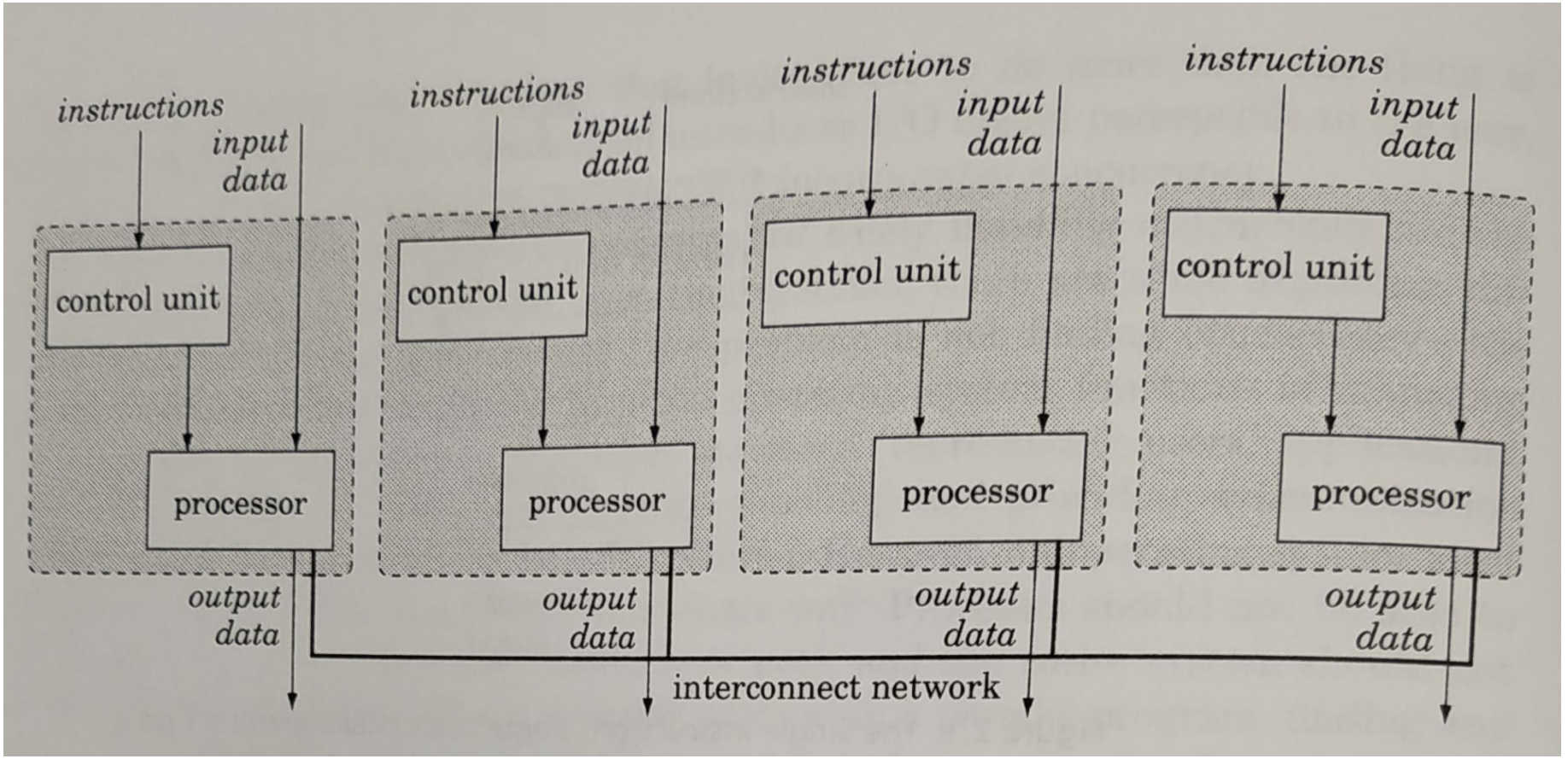
Typy výpočtov – Flynn tax.

- SISD
- SIMD
- MISD
- MIMD

Typy výpočtov – Flynn tax.

- SISD – Single Instruction Single Data
- SIMD – Single Instruction Multiple Data
- MISD – Multiple Instruction Single Data
- MIMD – Multiple Instruction Multiple Data


instructions
input data
control unit
processor
output data
instructions
input data
input data
input data
input data
control unit
processor
processor
processor
processor
output data
output data
output data
output data


instructions
input data
control unit
processor
output data
instructions
input data
control unit
processor
output data
instructions
input data
control unit
processor
output data
instructions
input data
control unit
processor
output data
interconnect network

GPU

GPU

- Orientované SIMD

GPU

- Orientované SIMD
- Pomalšie taktovanie ako CPU

GPU

- Orientované SIMD
- Pomalšie taktovanie ako CPU
- Nemá prerušenia, I/O (ako klávesnica a myš), virtuálnu pamäť

GPU

- Orientované SIMD
- Pomalšie taktovanie ako CPU
- Nemá prerušenia, I/O (ako klávesnica a myš), virtuálnu pamäť

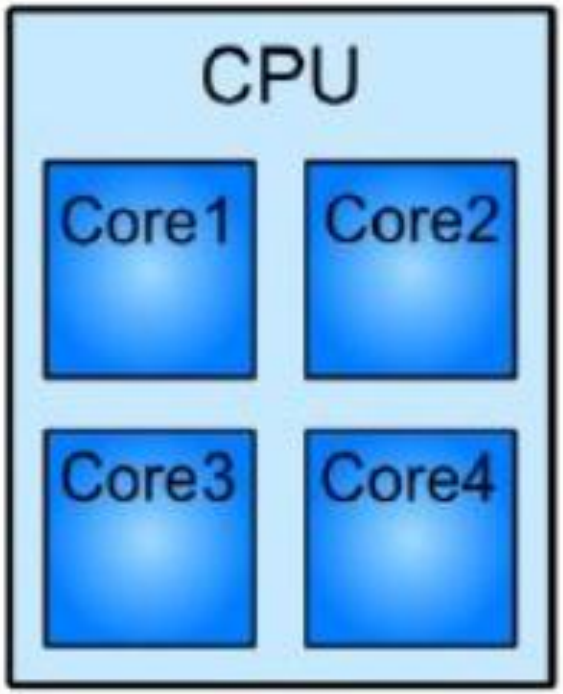
- CPU víťazí v prípade SISD (napr. hash 1 reťazca)

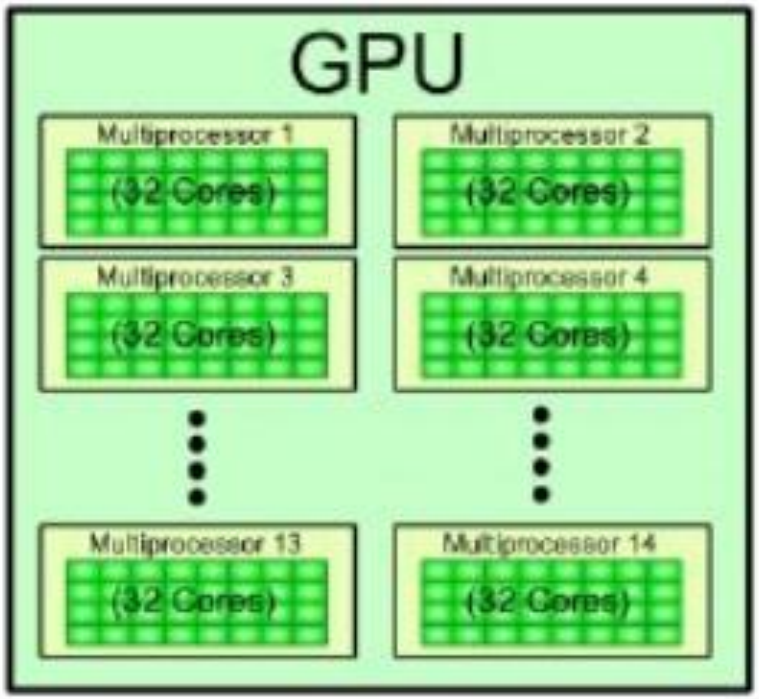
GPU

- Orientované SIMD
- Pomalšie taktovanie ako CPU
- Nemá prerušenia, I/O (ako klávesnica a myš), virtuálnu pamäť

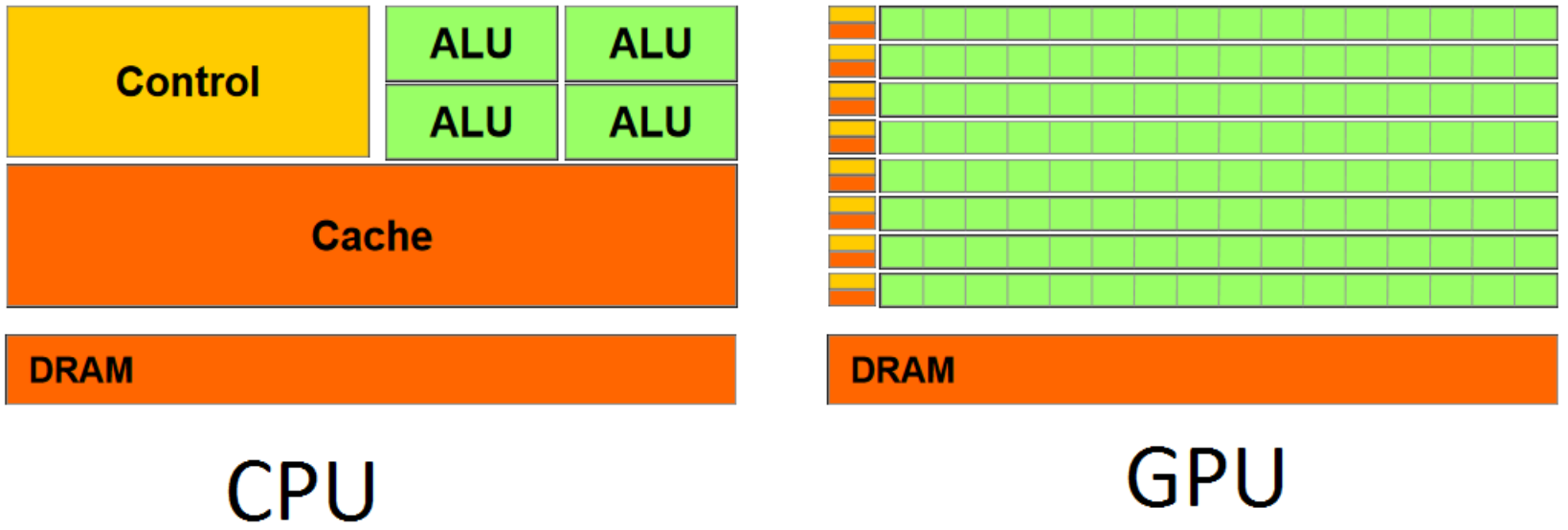
- CPU víťazí v prípade SISD (napr. hash 1 reťazca); ale v prípade tisícov reťazcov víťazí GPU…

Porovnanie CPU a GPU





Porovnanie CPU a GPU v2



Zdroj: http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

Porovnanie paralelizmu

- CPU – **paralelizmus úloh**
- GPU – **paralelizmus údajov**

Porovnanie paralelizmu

- CPU – **paralelizmus úloh**
- Viacero úloh sa mapuje na viac tokov riadenia (MIxD)
- **Desiatky** procesov/vlákien na desiatkach jadier

- GPU – **paralelizmus údajov**
- Tá istá inštrukcia sa vykonáva na rôznych údajoch (SIMD)
- **Desaťtisíce** vlákien na stovkách jadier

Porovnanie paralelizmu

- CPU – **paralelizmus úloh**

- Viacero úloh sa mapuje na viac tokov riadenia (MIxD)
- **Desiatky** procesov/vlákien na desiatkach jadier
- Každé vlákno nutné spravovať **softvérovo**
- Každému vláknu treba určiť, čo má vykonávať

- GPU – **paralelizmus údajov**

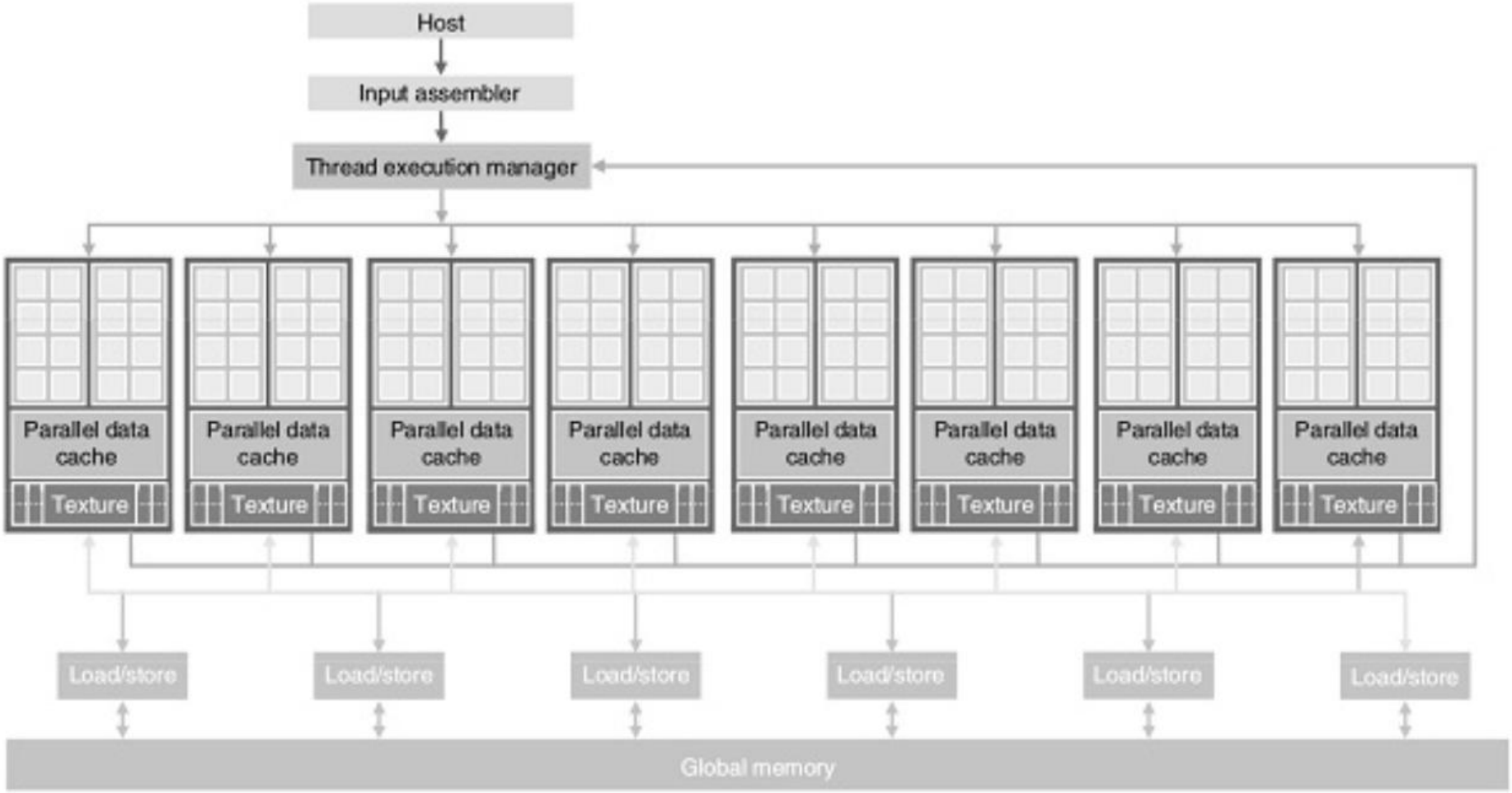
- Tá istá inštrukcia sa vykonáva na rôznych údajoch (SIMD)
- **Desaťtisíce** vlákien na stovkách jadier
- Každé vlákno spravované **hardvérovo**
- Skupina vlákien má určené, čo bude vykonávať

Vývoj architektúr NVidia

Vývoj architektúr NVidia

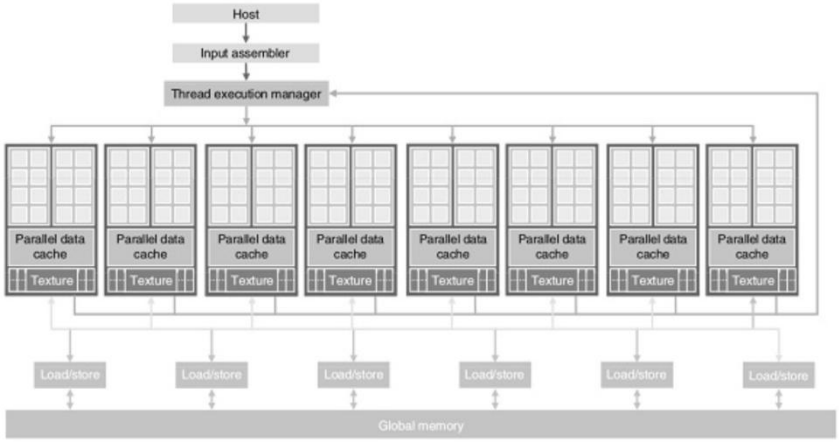
1. Pred GPGPU
   - 1995 – prvá grafická karta fy Nvidia NV1
   - 1997 – Riva 128 (DX3)
   - 1998 – Riva TNT (DX5)
   - 1999 – Riva TNT2 (DX6)
2. November 2006 – G80 (128 SP / 16 SM = 8)
3. Január 2008 – GT200 (240 SP / 30 SM = 8)
4. Fermi (512 SP / 16 SM = 32 SP per SM)

Architektúra G80



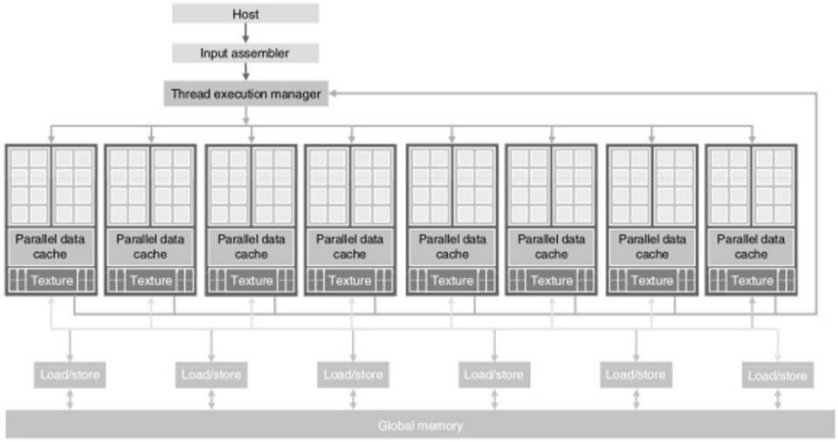
Architektúra G80

- 16 SM (Streaming Multiprocessor)



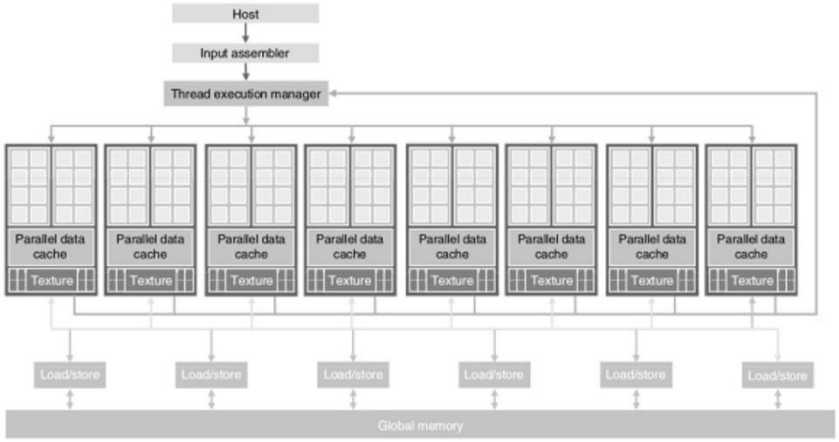
Architektúra G80

- 16 SM (Streaming Multiprocessor)
- Každý SM ma 8 SP (Streaming Processor)
- Spolu je to teda 128 SP



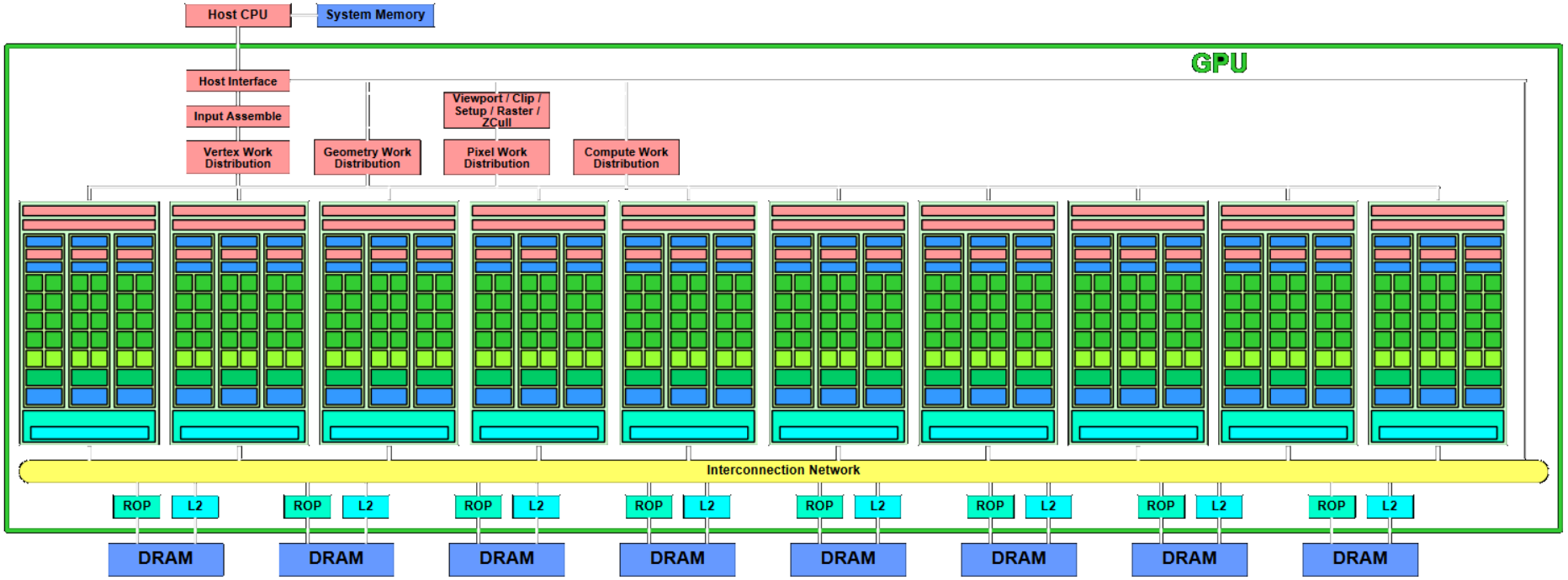
Architektúra G80

- 16 SM (Streaming Multiprocessor)
- Každý SM ma 8 SP (Streaming Processor)
- Spolu je to teda 128 SP

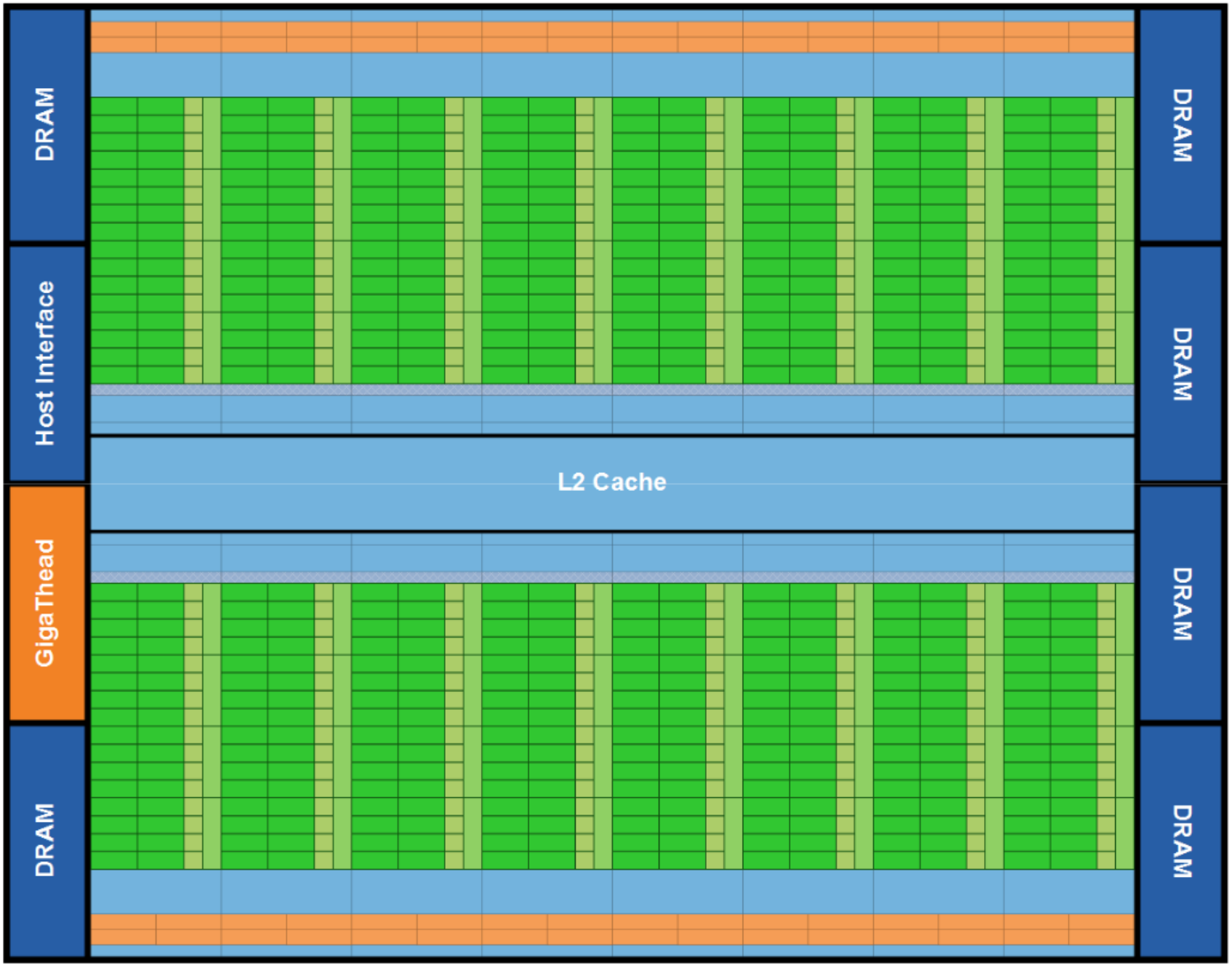


- Každý SP má
  - Jednu MAD (multiply and addition unit)
  - Jednu ďalšiu MU (multiply unit)

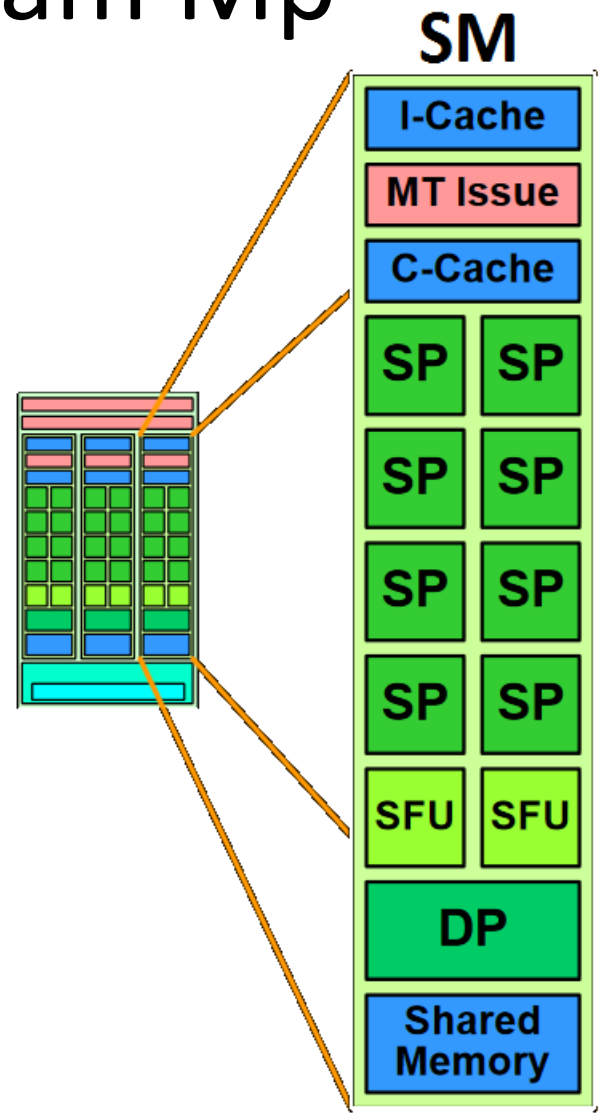
Architektúra GT200



Architektúra Fermi

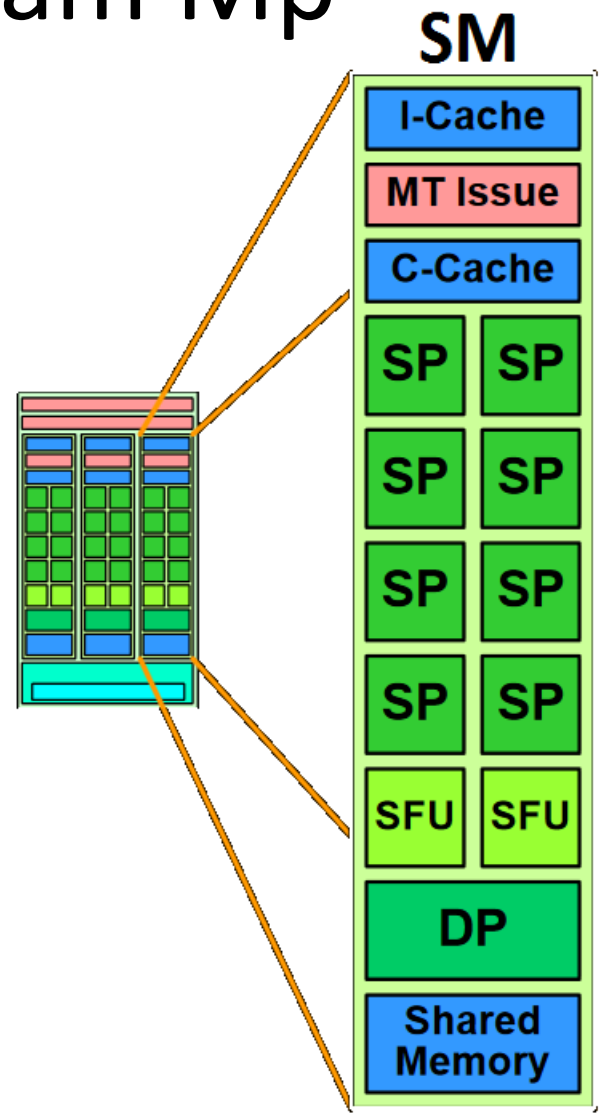


Z čoho sa skladá Stream Mp



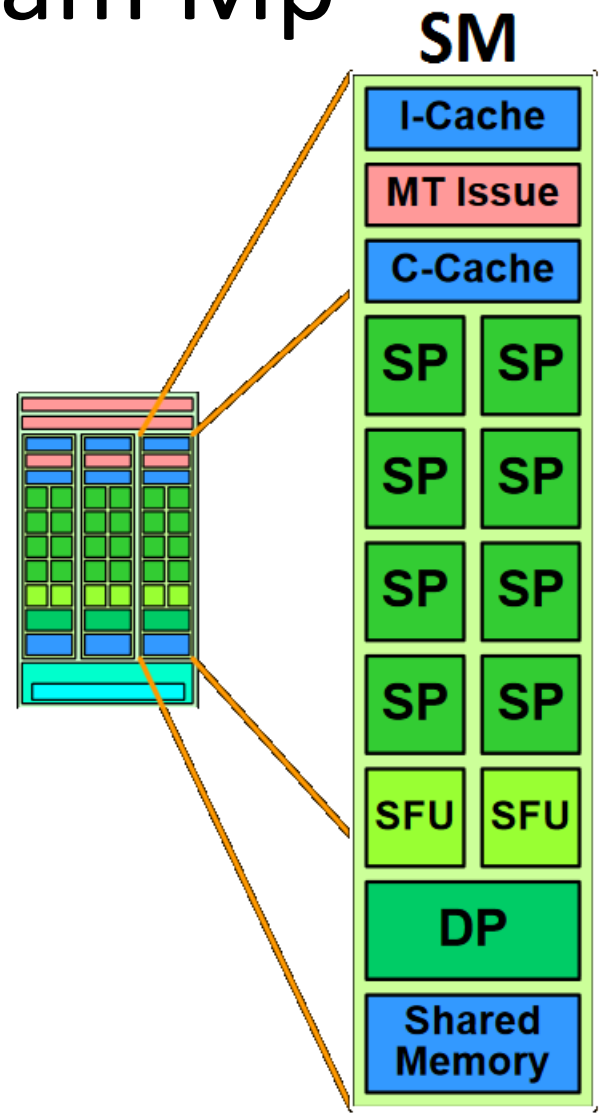
Z čoho sa skladá Stream Mp

- Správa vlákien
  - Max 1024 konkurentne (GT200)
  - Hardvérove plánovanie



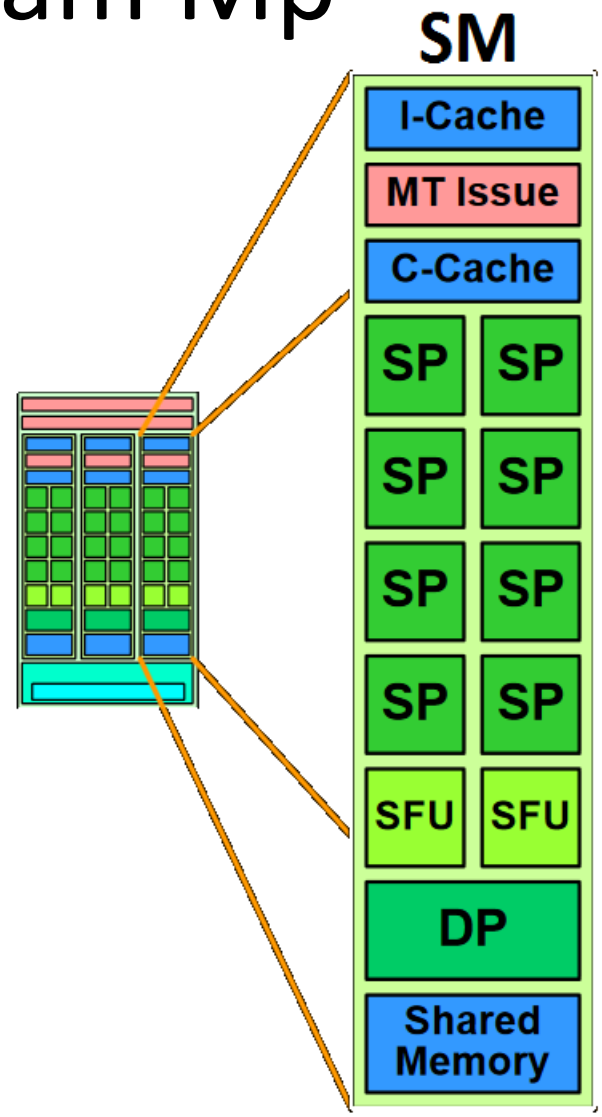
Z čoho sa skladá Stream Mp

- Správa vlákien
  - Max 1024 konkurentne (GT200)
  - Hardvérove plánovanie
- 8 SP
  - 32 bit floating point
  - 32 a 64 bit int



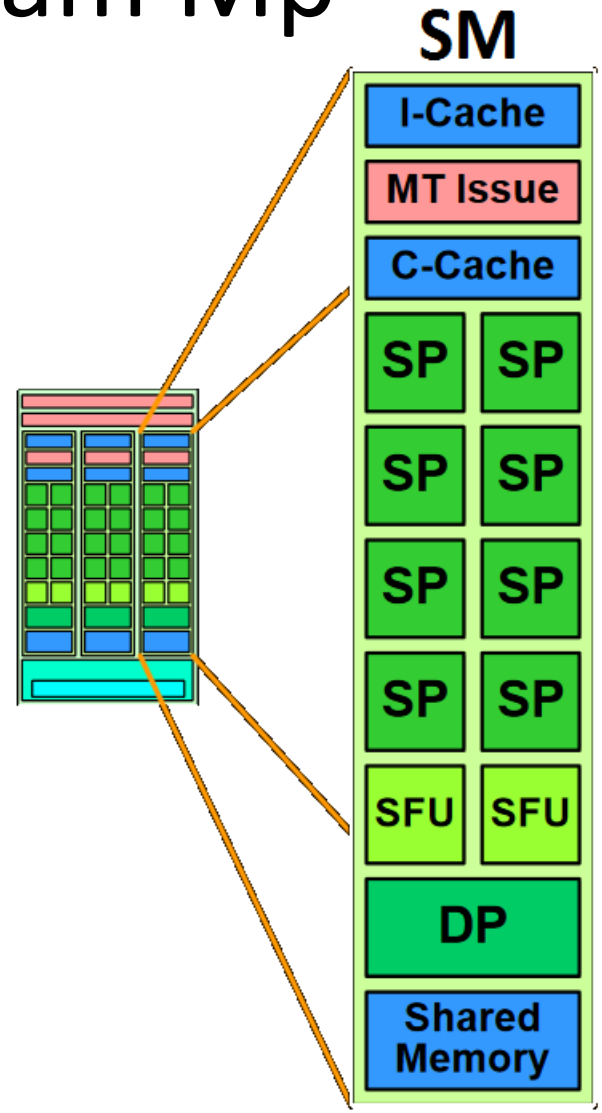
Z čoho sa skladá Stream Mp

- Správa vlákien
  - Max 1024 konkurentne (GT200)
  - Hardvérove plánovanie
- 8 SP
  - 32 bit floating point
  - 32 a 64 bit int
- 2 SFU
  - Jednotky špec. funkcií
  - sin, cos, log, exp



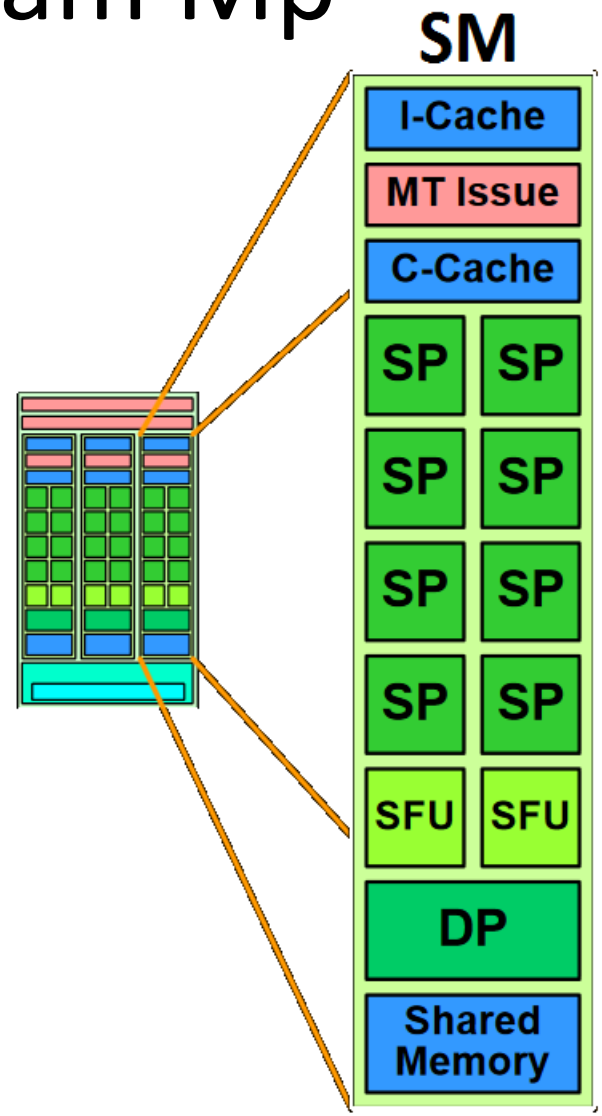
Z čoho sa skladá Stream Mp

- Správa vlákien
  - Max 1024 konkurentne (GT200)
  - Hardvérove plánovanie
- 8 SP
  - 32 bit floating point
  - 32 a 64 bit int
- 2 SFU
  - Jednotky špec. funkcií
  - sin, cos, log, exp
- Double Precision Unit
  - 64 bit floating point
  - Násobička so spočítaním



Z čoho sa skladá Stream Mp

- Správa vlákien
  - Max 1024 konkurentne (GT200)
  - Hardvérove plánovanie
- 8 SP
  - 32 bit floating point
  - 32 a 64 bit int
- 2 SFU
  - Jednotky špec. funkcií
  - sin, cos, log, exp
- Double Precision Unit
  - 64 bit floating point
  - Násobička so spočítaním



- 16kB zdieľanej pamäte

Rast výkonu

- Rok 2008: GT200 má 240 SP s výkonom > 1 TFLOP
- Rok 2018: RTX2080 Ti (TU102) má 88 ROP (Render Output Unit), 4352 shaders
- Rok 2020: RTX3080 Ti (GA102) má 33 TFLOP, 112 ROP, 10240 shaders

Štruktúra CUDA programu

Štruktúra CUDA programu

- Štandardne sa píšu v C alebo Fortrane :D

Štruktúra CUDA programu

- Štandardne sa píšu v C alebo Fortrane :D
- CUDA aplikácia obsahuje zvyčajne 2 časti

Štruktúra CUDA programu

- Štandardne sa píšu v C alebo Fortrane :D
- CUDA aplikácia obsahuje zvyčajne 2 časti
  - časť vykonávanú na CPU
  - časť vykonávanú na GPU

Štruktúra CUDA programu

- Štandardne sa píšu v C alebo Fortrane :D
- CUDA aplikácia obsahuje zvyčajne 2 časti
  - časť vykonávanú na CPU – **host**
  - časť vykonávanú na GPU – **device**

Štruktúra CUDA programu

- Štandardne sa píšu v C alebo Fortrane :D
- CUDA aplikácia obsahuje zvyčajne 2 časti
  - časť vykonávanú na CPU – **host**
  - časť vykonávanú na GPU – **device**

- Kód vykonávaný na CPU je kompilovateľný tradičným prekladačom C

Štruktúra CUDA programu

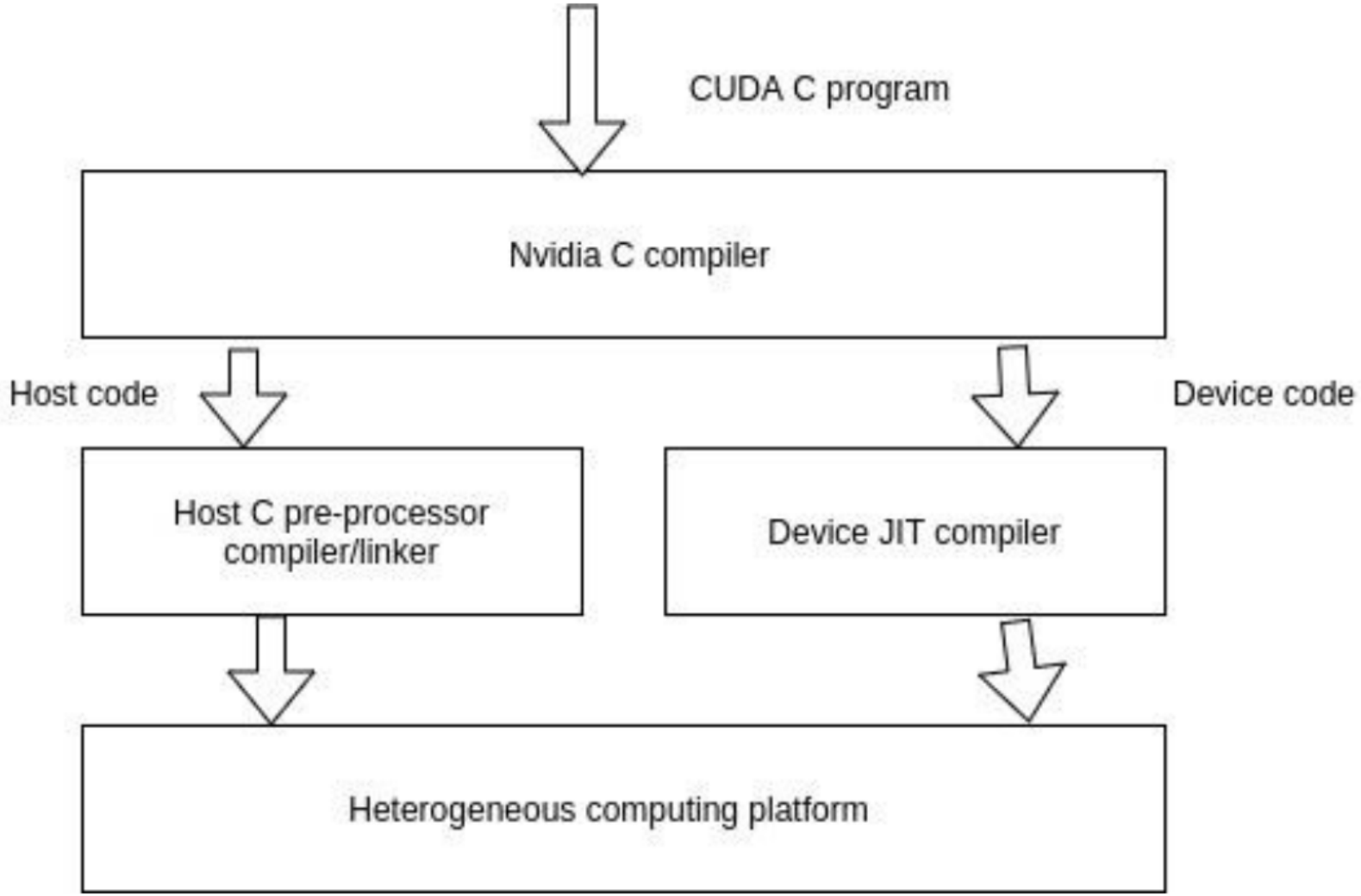
- Štandardne sa píšu v C alebo Fortrane :D
- CUDA aplikácia obsahuje zvyčajne 2 časti
  - časť vykonávanú na CPU – **host**
  - časť vykonávanú na GPU – **device**

- Kód vykonávaný na CPU je kompilovateľný tradičným prekladačom C
- Kód na zariadení vyžaduje špeciálny prekladač (Nvidia C Compiler – nvcc)

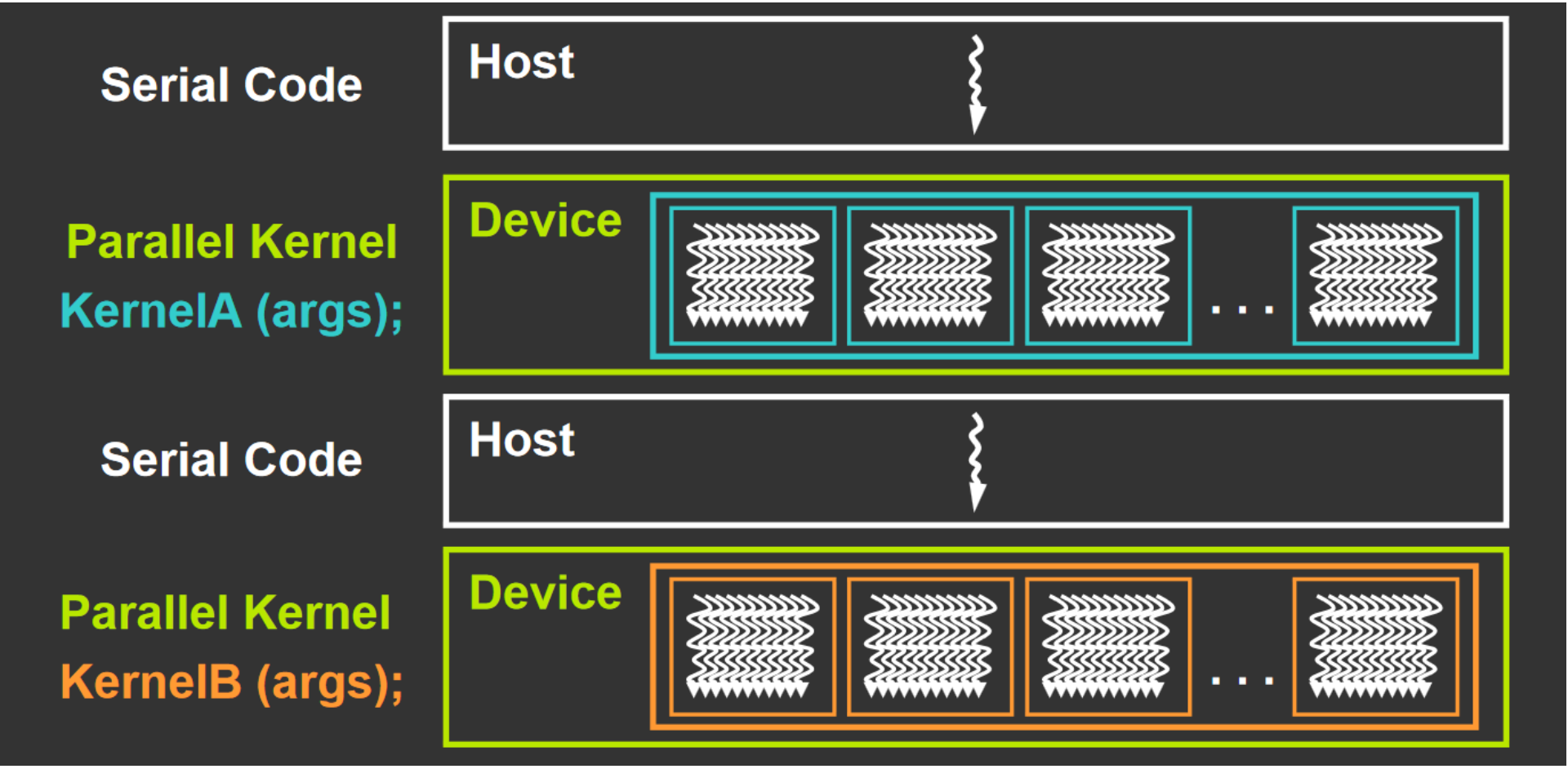
CUDA program

- NVCC vie pomocou špeciálnych kľúčových slov rozdeliť kód vykonávaný na CPU a na zariadení

CUDA program



CUDA program



HW vs SW

HW vs SW

- HW plánuje **VŽDY** 32 vlákien
  - Tzv. *warp*
  - Najmenšia hw plánovateľná jednotka
  - Vlákna warpu vykonávajú **naraz** tú istú inštrukciu

HW vs SW

- HW plánuje **VŽDY** 32 vlákien
  - Tzv. *warp*
  - Najmenšia hw plánovateľná jednotka
  - Vlákna warpu vykonávajú **naraz** tú istú inštrukciu

- SW beží vo vláknach (CUDA, nie OS)
  - Tzv. *kernel*
  - Vlákna kernelu sa združujú do blokov (1536 Fermi)

HW vs SW

- SW vlákna (pokračovanie)
  - Všetky vlákna bloku sa môžu striedať na jednom SM! Ako?

HW vs SW

- SW vlákna (pokračovanie)
  - Všetky vlákna bloku sa môžu striedať na jednom SM! Ako? Konkurentne

HW vs SW

- SW vlákna (pokračovanie)
  - Všetky vlákna bloku sa môžu striedať na jednom SM! Ako? Konkurentne
  - Dôsledok: môžu komunikovať (pomocou zdieľanej pamäte)

HW vs SW

- HW a SW spolu
  - bloky vlákien hardvér rozdeľuje na warpy (počet vlákien 32)
  - bloky vlákien sa v sw zoskupujú do mriežok (*grids*), každý grid vykonáva práve jeden kernel

HW vs SW

- HW a SW spolu
  - bloky vlákien hardvér rozdeľuje na warpy (počet vlákien 32)
  - bloky vlákien sa v sw zoskupujú do mriežok (*grids*), každý grid vykonáva práve jeden kernel
- Jedna aplikácia môže spúšťať súčasne viacero kernelov (Fermi a vyššie)

Vykonávanie CUDA programu

Vykonávanie CUDA programu

- Programátor určuje počet vlákien, ktoré sa majú na zariadení spustiť

Vykonávanie CUDA programu

- Programátor určuje počet vlákien, ktoré sa majú na zariadení spustiť

- Vlákna tvoria 3-rozmernú organizačnú štruktúru

Vykonávanie CUDA programu

- Programátor určuje počet vlákien, ktoré sa majú na zariadení spustiť

- Vlákna tvoria 3-rozmernú organizačnú štruktúru
  - Vlákna tvoria bloky
  - Bloky tvoria gridy

Vykonávanie CUDA programu

- Programátor určuje počet vlákien, ktoré sa majú na zariadení spustiť
- Vlákna tvoria 3-rozmernú organizačnú štruktúru
  - Vlákna tvoria bloky
  - Bloky tvoria gridy
- Každé vlákno má jedinečný identifikátor!

Organizácia vlákien

Organizácia vlákien

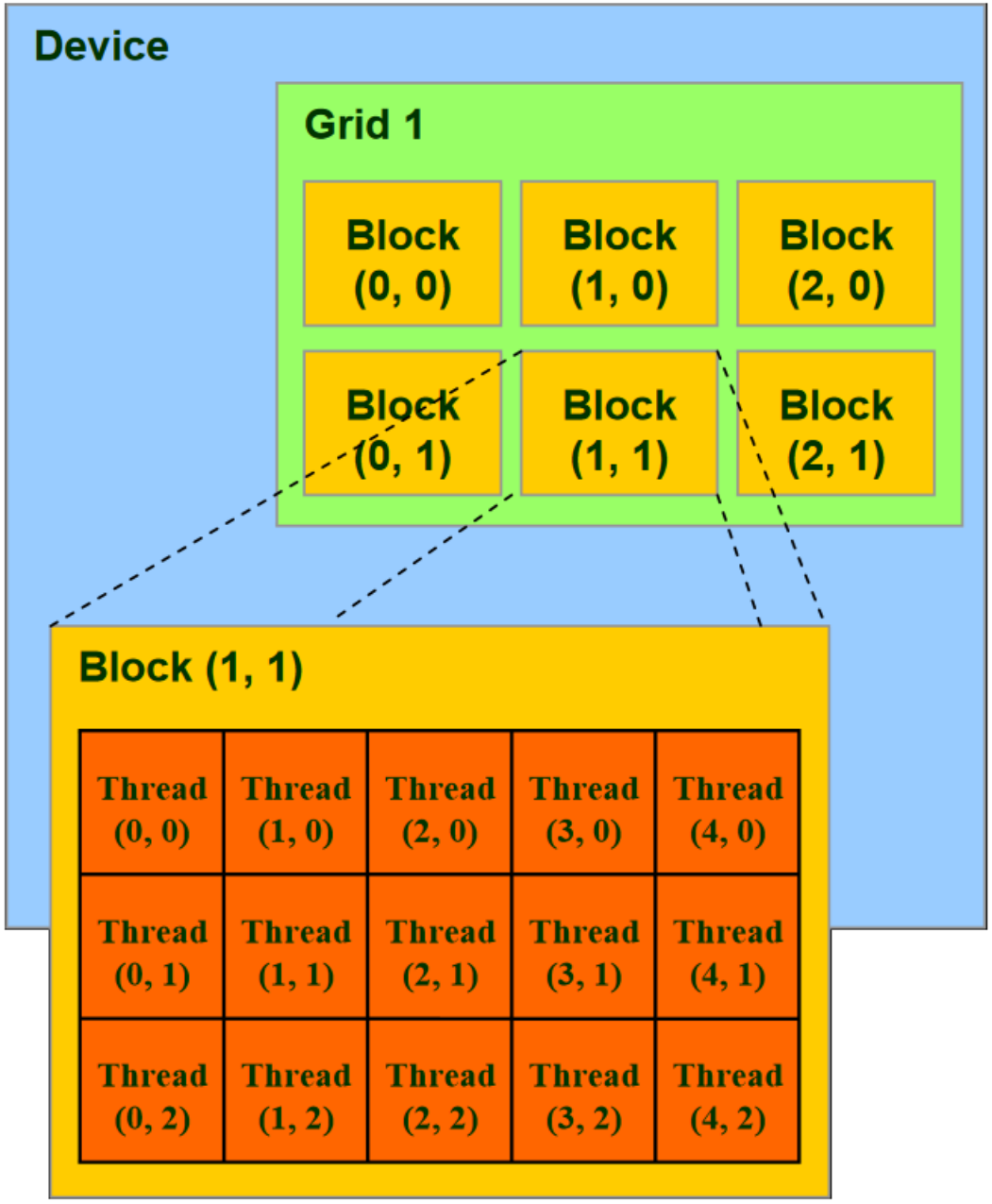
- Všetky vlákna v gride vykonávajú tú istú funkciu jadra (*kernel function*)

Organizácia vlákien

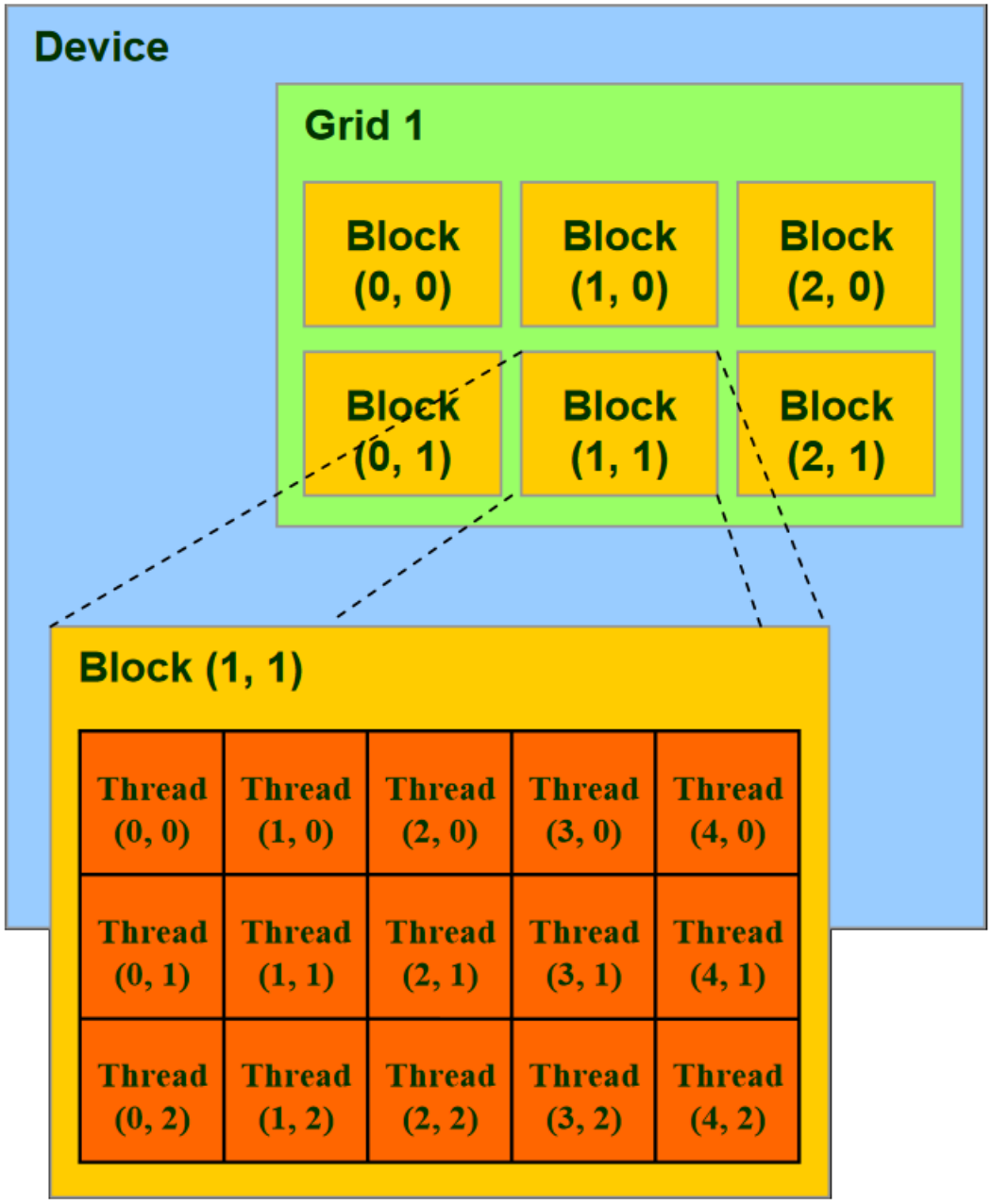
- Všetky vlákna v gride vykonávajú tú istú funkciu jadra (*kernel function*)

- Ako bolo spomínané, ich organizácia je dvojúrovňová:
  - grid sa skladá z blokov
  - a bloky z vlákien

- Vlákna
- Bloky



- Vlákna
  - 3D id
  - V rámci bloku id vlákna unikátne
- Bloky
  - 2D id
  - V rámci gridu id bloku unikátne



Organizácia vlákien

- Vstavané premenné nvcc

Organizácia vlákien

- Vstavané premenné nvcc:
  - threadIdx – id vlákna v rámci bloku
  - blockIdx – id bloku v rámci gridu

Organizácia vlákien

- Vstavané premenné nvcc:
  - threadIdx – id vlákna v rámci bloku
  - blockIdx – id bloku v rámci gridu
  - blockDim – rozmery bloku (#vlákien v bloku)
  - gridDim – rozmery gridu (#blokov v gride)

Organizácia vlákien

- Vstavané premenné nvcc:
  - threadIdx – id vlákna v rámci bloku
  - blockIdx – id bloku v rámci gridu
  - blockDim – rozmery bloku (#vlákien v bloku)
    - Už od CUDA 1.x má blok možnosti 1D, 2D a 3D
  - gridDim – rozmery gridu (#blokov v gride)
    - CUDA 1.x má 1D a 2D rozmery, až CUDA 2.x pridáva 3D grid

Organizácia vlákien

- Rozmery bloku/gridu 1D, 2D, 3D sú **logickým** pohľadom na štruktúru vlákien

- Rovnaký počet vlákien je možné identifikovať rôznym spôsobom (ten sa však fixne stanovuje pri spustení kernelu, za behu sa pohľad na štruktúru gridu nemení)

Organizácia vlákien

- Nastavenie rozmerov gridu pomocou funkcie

- Nastavenie rozmerov bloku pomocou fnc

Organizácia vlákien

- Nastavenie rozmerov gridu pomocou funkcie:
  - dim3 blocks(nx, ny, nz);

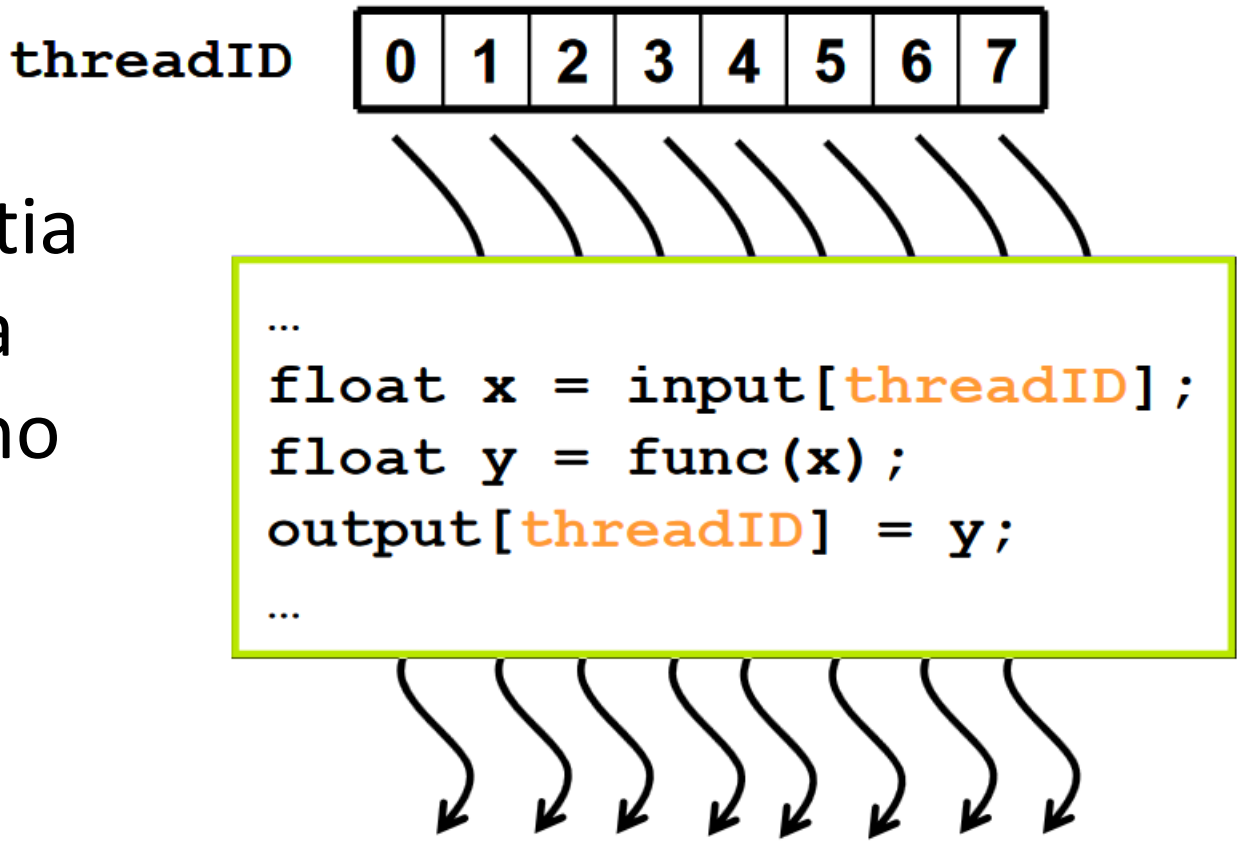
- Nastavenie rozmerov bloku pomocou fnc:
  - dim3 threadsPerBlock(mx, my, mz);

Organizácia vlákien

- Nastavenie rozmerov gridu pomocou:
  - dim3 blocks(nx, ny, nz);
  - CUDA 1.x má 1D a 2D rozmery, CUDA 2.x pridáva 3D grid
- Nastavenie rozmerov bloku pomocou:
  - dim3 threadsPerBlock(mx, my, mz);
  - Už od CUDA 1.x má blok možnosti 1D, 2D a 3D

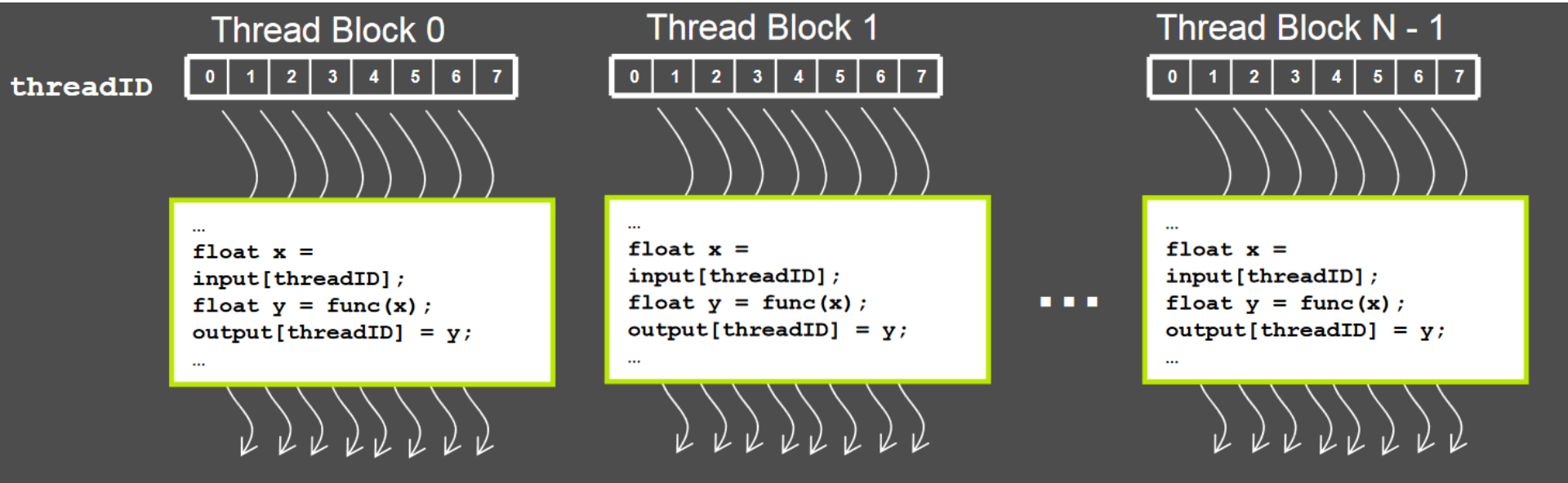
Organizácia vlákien

- Ukážka využitia identifikátora vlákna pri jeho behu na GPU zariadení



Organizácia vlákien

- Ukážka využitia identifikátora vlákna pri jeho behu na GPU zariadení
- *threadID* je unikátne v rámci bloku!



Organizácia vlákien

- Iný pohľad na SW versus HW

- Vlákno ←→ SP
- Blok ←→ SM
- Kernel (grid blokov)←→Zariadenie (viacero SM)

Organizácia vlákien - príklad

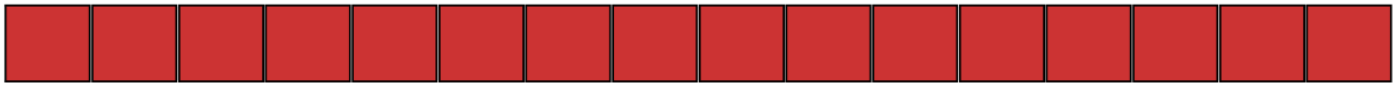
- Pripočítajme konštantu **b** ku **N**-prvkovému poľu
- Nech **N** = 16, **blockDim** = 4 → #blokov = 16/4 = 4
- Ako určíme na základe **blockDim**, **blockIdx** a **threadIdx** index do poľa, ktorý má vlákno použiť?

Organizácia vlákien - príklad

- Pripočítajme konštantu **b** ku **N**-prvkovému poľu
- Nech **N** = 16, **blockDim** = 4 → #blokov = 16/4 = 4
- Ako určíme na základe **blockDim**, **blockIdx** a **threadIdx** index do poľa, ktorý má vlákno použiť?

- idx = blockDim * blockIdx + threadIdx

Organizácia vlákien - príklad



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

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```

blockIdx.x=0
blockDim.x=4
threadIdx.x=0,1,2,3
idx=0,1,2,3

blockIdx.x=1
blockDim.x=4
threadIdx.x=0,1,2,3
idx=4,5,6,7

blockIdx.x=2
blockDim.x=4
threadIdx.x=0,1,2,3
idx=8,9,10,11

blockIdx.x=3
blockDim.x=4
threadIdx.x=0,1,2,3
idx=12,13,14,15

Organizácia vlákien - príklad

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, 16);
}
```

CUDA program

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


void main()
{
  .....
    increment_gpu<<< 4, 4 >>>(a, b, 16);
}
```

Globálna pamäť zariadenia

Globálna pamäť zariadenia

- O správu pamäte na zariadení sa musí starať **programátor**

Globálna pamäť zariadenia

- O správu pamäte na zariadení sa musí starať **programátor**
  - Najprv musí alokovať pamäť na zariadení

Globálna pamäť zariadenia

- O správu pamäte na zariadení sa musí starať **programátor**
  - Najprv musí alokovať pamäť na zariadení
  - Po alokovaní pamäte musí preniesť údaje z CPU (RAM) do tejto alokovanej pamäte zariadenia

Globálna pamäť zariadenia

- O správu pamäte na zariadení sa musí starať **programátor**
  - Najprv musí alokovať pamäť na zariadení
  - Po alokovaní pamäte musí preniesť údaje z CPU (RAM) do tejto alokovanej pamäte zariadenia
  - Po ukončení výpočtu musí preniesť údaje z pamäte zariadenia do pamäte hosta

Globálna pamäť zariadenia

- O správu pamäte na zariadení sa musí starať **programátor**
  - Najprv musí alokovať pamäť na zariadení
  - Po alokovaní pamäte musí preniesť údaje z CPU (RAM) do tejto alokovanej pamäte zariadenia
  - Po ukončení výpočtu musí preniesť údaje z pamäte zariadenia do pamäte hosta
  - Napokon musí uvoľniť pamäť, ktorú predtým alokoval

Základné typy pamätí na GPU

Základné typy pamätí na GPU

1. Globálna pamäť zariadenia
   - Všetky vlákna, celá doba behu aplikácie

Základné typy pamätí na GPU

1. Globálna pamäť zariadenia
   - Všetky vlákna, celá doba behu aplikácie

2. Zdieľaná pamäť
   - Vlákna jedného bloku, počas existencie bloku

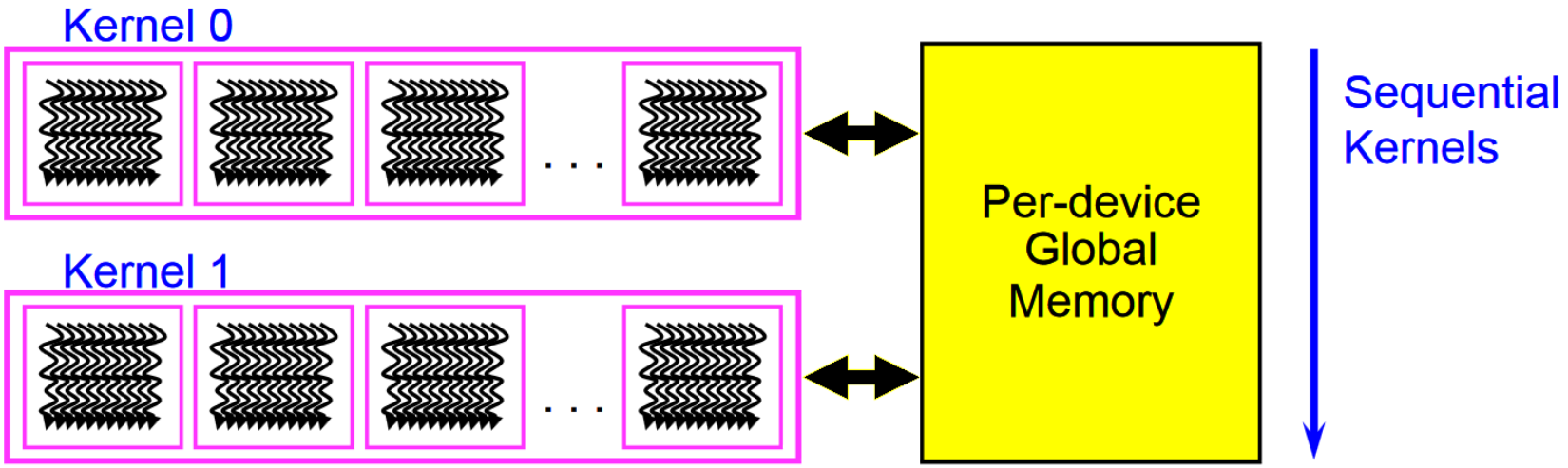
Základné typy pamätí na GPU

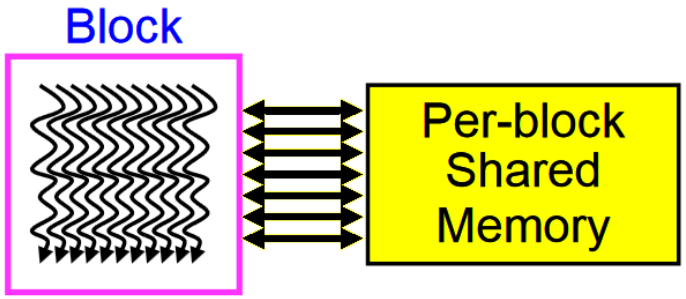
1. Globálna pamäť zariadenia
   - Všetky vlákna, celá doba behu aplikácie
2. Zdieľaná pamäť
   - Vlákna jedného bloku, počas existencie bloku
3. Lokálna pamäť
   - Iba pre vlákno, počas jeho behu

Základné typy pamätí na GPU

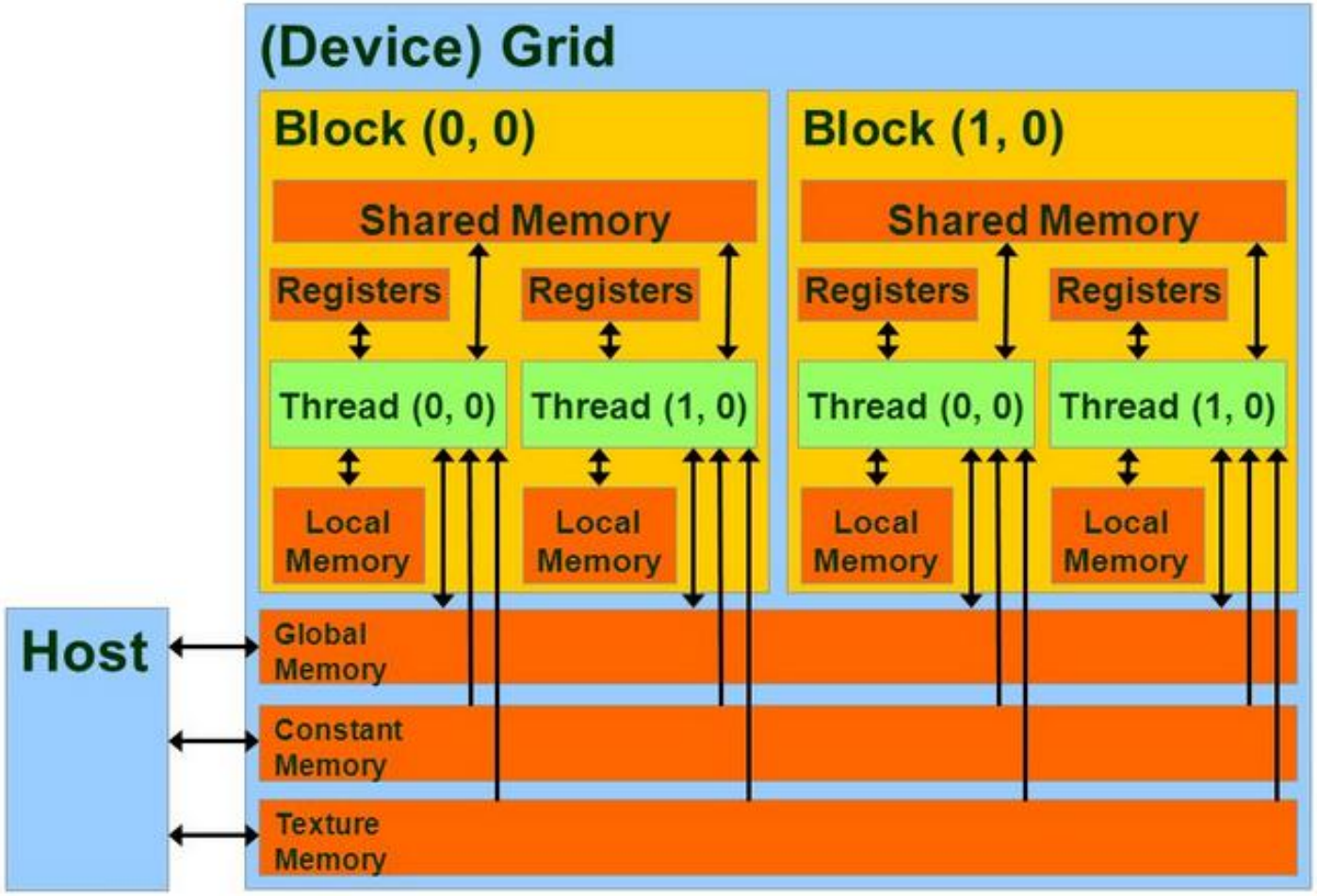
1. Globálna pamäť zariadenia
   - Všetky vlákna, celá doba behu aplikácie
   - __device__ __constant__
2. Zdieľaná pamäť
   - Vlákna jedného bloku, počas existencie bloku
   - __shared__
3. Lokálna pamäť
   - Iba pre vlákno, počas jeho behu

Základné typy pamätí na GPU





Pamäťový model GPU



[CUDA C Programming Guide Version 3.2]

Typy funkcií

Typy funkcií

- **__global__** označuje tzv. kernel funkciu, ktorá je volaná z hosta, ale vykonáva sa na zariadení

Typy funkcií

- **__global__** označuje tzv. kernel funkciu, ktorá je volaná z hosta, ale vykonáva sa na zariadení
- **__device__** označuje funkciu, ktorá je volaná zo zariadenia, a môže byť iba na ňom vykonávaná

Typy funkcií

- **__global__** označuje tzv. kernel funkciu, ktorá je volaná z hosta, ale vykonáva sa na zariadení
- **__device__** označuje funkciu, ktorá je volaná zo zariadenia, a môže byť iba na ňom vykonávaná
- **__host__** deklaruje funkciu, ktorá je volaná z hosta a môže byť vykonávaná iba na ňom

Ukážka CUDA programu 1

```
void vecAdd_b(float* A, int N, float b) {
    int size = N*sizeof(float);
    float *d_A;

    //  alokuj pamat na zariadeni
    cudaMalloc((void**)& d_A, size);

    // skopiruj udaje z hosta na zariadenie
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);

     // vykonaj kernel
    add_on_gpu<<< N/blockSize, blockSize>>>(d_A, b, N);

    // skopiruj udaje zo zariadenia spat na hosta
    cudaMemcpy(A, d_A, size, cudaMemcpyDeviceToHost);

    // uvolni pamat zariadenia
    cudaFree(d_A);
}
```

Ukážka CUDA programu 1

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, 16);
}
```

CUDA program

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


void main()
{
  .....
    increment_gpu<<< 4, 4 >>>(a, b, 16);
}
```

Ukážka CUDA programu 2

```
void vecAdd(float* A, float* B, float* C,int N) {
    int size = N*sizeof(float);
    float *d_A,*d_B,*d_C;

    cudaMalloc((void**)& d_A, size);
    cudaMalloc((void**)& d_B, size);
    cudaMalloc((void**)& d_C, size);

    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

     // … az po ukonceni vypoctu sa bude kopirovat vysledok zo zariadenia!!! Tu by
    malo byt volanie kernelu…

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Deklarácia a vyvolanie kernelu

- Deklarácia kernelu:

  __global__ void kernel_fnc(...) { ... }

  Návratovy typ kernelu musí byť vždy **void**!

- Spustenie kernelu:

  dim3 blocks(nx, ny, nz);

  dim3 threadsPerBlock(mx, my, mz);

  kernel_fnc<<< blocks, threadsPerBlock >>>(...);

Násobenie matice

Matice

M0,0	M0,1	M0,2	M0,3
M1,0	M1,1	M1,2	M1,3
M2,0	M2,1	M2,2	M2,3
M3,0	M3,1	M3,2	M3,3

Matice

M0,0	M0,1	M0,2	M0,3
M1,0	M1,1	M1,2	M1,3
M2,0	M2,1	M2,2	M2,3
M3,0	M3,1	M3,2	M3,3

- Reprezentácia 2D matice v pamäti

Matice

M0,0	M0,1	M0,2	M0,3
M1,0	M1,1	M1,2	M1,3
M2,0	M2,1	M2,2	M2,3
M3,0	M3,1	M3,2	M3,3

- Reprezentácia 2D matice v pamäti
  - Po riadkoch
  - Po stĺpcoch

Matice

M0,0	M0,1	M0,2	M0,3
M1,0	M1,1	M1,2	M1,3
M2,0	M2,1	M2,2	M2,3
M3,0	M3,1	M3,2	M3,3

- Reprezentácia 2D matice v pamäti
  - Po riadkoch

M0,0	M0,1	M0,2	M0,3	M1,0	M1,1	M1,2	M1,3	M2,0	M2,1	M2,2	M2,3	M3,0	M3,1	M3,2	M3,3

  - Po stĺpcoch

M0,0	M1,0	M2,0	M3,0	M0,1	M1,1	M2,1	M3,1	M0,2	M1,2	M2,2	M3,2	M0,3	M1,3	M2,3	M3,3

Matice

- Adresácia prvku 2D matice reprezentovanej 1D poľom “po riadkoch”

Matice

- Adresácia prvku 2D matice reprezentovanej 1D poľom “po riadkoch”
  - ind(x,y)

Matice

- Adresácia prvku 2D matice reprezentovanej 1D poľom “po riadkoch”
  - ind(x,y)
  - ind(x,y) = x * width + y

Matice

- Adresácia prvku 2D matice reprezentovanej 1D poľom “po riadkoch”
  - ind(x,y)
  - ind(x,y) = x * width + y

- Každý jeden prvok môžeme mapovať práve na jedno vlákno na GPU

Matice

- Každý jeden prvok môžeme mapovať práve na jedno vlákno na GPU
  - Vďaka vstavaným identifikátorom

Matice

- Každý jeden prvok môžeme mapovať práve na jedno vlákno na GPU
  - Vďaka vstavaným identifikátorom
  - row = blockIdx.x * blockDim.x + threadIdx.x
  - col = blockIdx.y * blockDim.y + threadIdx.y

Matice

- Každý jeden prvok môžeme mapovať práve na jedno vlákno na GPU
  - Vďaka vstavaným identifikátorom
  - row = blockIdx.x * blockDim.x + threadIdx.x
  - col = blockIdx.y * blockDim.y + threadIdx.y

- Násobenie matíc ovládame…

```
__global__ void simpleMatMulKernell(float* d_M, float* d_N,
    float* d_P, int width) {

    int row = blockIdx.y * width + threadIdx.x;
    int col = blockIdx.y * width + threadIdx.y;

    // Pozor na vlakna, ktore su mimo rozmerov matice!!!
    if (row < width && col < width) {
        float product_val = 0;
        for (int k=0; k<width; k++) {
            product_val +=
                d_M[row*width+k] *
                d_N[k*width+col];
        }
        d_P[row*width+col] = product_val;
    }
}
```

Ako ladiť CUDA program?

Ako ladiť CUDA program?

- Mód emulácie zariadenia!

Ako ladiť CUDA program?

- Mód emulácie zariadenia!

- Treba program skompilovať pomocou prepínača: “nvcc –deviceemu”
  - Zariadenie bude emulované
  - Netreba žiadnu grafickú kartu ani ovládače
  - Každé vlákno zariadenia bude emulované vláknom hosta!

Ako ladiť CUDA program?

- Aké sú výhody takého prístupu

Ako ladiť CUDA program?

- Aké sú výhody takého prístupu

- Natívna podpora ladiacich nástrojov
- Prístup ku premenným zariadenia v kóde hosta a opačne
- Povolené volanie ľubovoľnej funkcie hosta z funkcie zariadenia (napr. *printf*) a opačne

CUDA v Pythone

CUDA v Pythone

- Jestvuje viacero možností

CUDA v Pythone

- Jestvuje viacero možností
- Nvidia oficiálne uvádza 2

CUDA v Pythone

- Jestvuje viacero možností
- Nvidia oficiálne uvádza 2
  - PyCUDA
  - Anaconda

CUDA v Pythone

- Jestvuje viacero možností

- Nvidia oficiálne uvádza 2
  - PyCUDA
    - https://developer.nvidia.com/pycuda
  - Anaconda
    - https://developer.nvidia.com/anaconda-accelerate

CUDA v Pythone

- Jestvuje viacero možností

- Nvidia oficiálne uvádza 2
  - PyCUDA – treba písat C kód v Pythone ;)
    - https://developer.nvidia.com/pycuda
  - Anaconda – čistý Python
    - https://developer.nvidia.com/anaconda-accelerate

CUDA v Pythone

- Zvolili sme iba minimálnu časť ekosystému Anaconda Accelerate s názvom Numba

CUDA v Pythone

- Zvolili sme iba minimálnu časť ekosystému Anaconda Accelerate s názvom Numba

- Anaconda Accelerate je voľná Python distribúcia pre enterprise riešenia od Continuum Analytics určená na vedecké výpočty, prediktívnu analýzu, spracovanie veľkého objemu dát

Zdroje

- https://developer.nvidia.com/how-to-cuda-python

- https://nyu-cds.github.io/python-numba/05-cuda/

CUDA pomocou Numba

CUDA pomocou Numba

- Jedným z dôvodov voľby knižnice Numba je to, že dokáže emulovať GPU (takže na precvičenie nepotrebujeme reálny hw)

CUDA pomocou Numba

- Jedným z dôvodov voľby knižnice Numba je to, že dokáže emulovať GPU (takže na precvičenie nepotrebujeme reálny hw)
- Ako zapnúť emulátor?

CUDA pomocou Numba

- Jedným z dôvodov voľby knižnice Numba je to, že dokáže emulovať GPU (takže na precvičenie nepotrebujeme reálny hw)
- Ako zapnúť emulátor?
  - Linux
  - Windows

CUDA pomocou Numba

- Jedným z dôvodov voľby knižnice Numba je to, že dokáže emulovať GPU (takže na precvičenie nepotrebujeme reálny hw)
- Ako zapnúť emulátor?
  - Linux: export NUMBA_ENABLE_CUDASIM=1
  - Windows: set NUMBA_ENABLE_CUDASIM=1

CUDA pomocou Numba

- Jedným z dôvodov voľby knižnice Numba je to, že dokáže emulovať GPU (takže na precvičenie nepotrebujeme reálny hw)
- Ako zapnúť emulátor?
  - Linux: export NUMBA_ENABLE_CUDASIM=1
  - Windows: set NUMBA_ENABLE_CUDASIM=1

  - Až potom zapnúť Idle alebo Python shell !!!

Zdroje prednášky

- http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

- https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

Zdroje prednášky

- https://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA%27s_Fermi-The_First_Complete_GPU_Architecture.pdf

Cviko

Cviko

- https://nyu-cds.github.io/python-numba/05-cuda/

- Vyskúšať CUDA programovanie pomocou simulátora Numba v Pythone
- Kto má kartu Nvidia, môže využiť priamo hardvér

Ďakujem za pozornosť!