PPaDS MMXX

Matus Jokay, C-503, matus.jokay@stuba.sk
Roderik Ploszek, C-512, roderik.ploszek@stuba.sk

uim.fei.stuba.sk/predmet/i-ppds
konzultacie dohodou

Naco GPU?

- Dlhy cas boli graficke karty bez moznosti vykonavania vseobecnych vypoctov (GPGPU)
- Mali fixovanu funkcionalitu (OpenGL, DirectX)

Naco GPU?

- Dlhy cas boli graficke karty bez moznosti vykonavania vseobecnych vypoctov (GPGPU)
- Mali fixovanu funkcionalitu (OpenGL, DirectX)

- Co v pripade prevodu obrazka z RGB do sedej skaly?

Naco GPU?

- Co v pripade prevodu obrazka z RGB do sedej skaly? Napr. 1920x1080

Naco GPU?

- Co v pripade prevodu obrazka z RGB do sedej skaly? Napr. 1920x1080

- CPU musi vykonavat konverziu sekvencne (bod po bode), a teda cas vypoctu zavisi od rozmerov obrazka

Naco GPU?

- Co v pripade prevodu obrazka z RGB do sedej skaly? Napr. 1920x1080

- CPU musi vykonavat konverziu sekvencne (bod po bode), a teda cas vypoctu zavisi od rozmerov obrazka
- V nasom priklade treba 2 073 600 vypoctov

Naco GPU?

- Co v pripade prevodu obrazka z RGB do sedej skaly? Napr. 1920x1080 = 2 073 600

- Vykonava sa ta ista operacia, ale nad roznymi udajmi!

Naco GPU?

- Co v pripade prevodu obrazka z RGB do sedej skaly? Napr. 1920x1080 = 2 073 600

- Vykonava sa ta ista operacia, ale nad roznymi udajmi! SIMD!!!

Naco GPU?

- Co v pripade prevodu obrazka z RGB do sedej skaly? Napr. 1920x1080 = 2 073 600

- Vykonava sa ta ista operacia, ale nad roznymi udajmi! SIMD!!!
- Idealne pre GPU; na GPU vieme spustit niekolko milionov vlakien sucasne (tzv. workers)

Naco GPU?

- Co v pripade prevodu obrazka z RGB do sedej skaly? Napr. 1920x1080 = 2 073 600

- Vykonava sa ta ista operacia, ale nad roznymi udajmi! SIMD!!!
- Idealne pre GPU; na GPU vieme spustit niekolko milionov vlakien sucasne (tzv. workers) (t.j. pri roznych rozmeroch obrazkov bude vypocet vzdy 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

- Rozsirenie C-cka
- API model urceny na paralelne vypocty

CUDA

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

- Rozsirenie C-cka
- API model urceny na paralelne vypocty

- Vytvoreny firmou Nvidia

CPU a Moorov zakon

CPU a Moorov zakon

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

CPU a Moorov zakon

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

CPU a Moorov zakon

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

CPU a Moorov zakon

- Moorov zakon
  - zložitosť integrovaných obvodov sa zdvojnásobuje každých 18 mesiacov, pričom cena ostáva konštantná
  - 1965
- Platil 40 rokov…
  - Zvysovanie vykonu sa v sucasnosti deje nie miniaturizaciou

Paralelizacia CPU

Paralelizacia CPU

- Kroky CPU potrebne k vykonaniu instrukcie

Paralelizacia CPU

- Kroky CPU potrebne k vykonaniu instrukcie
  - Instruction Fetch

Paralelizacia CPU

- Kroky CPU potrebne k vykonaniu instrukcie
  - Instruction Fetch
  - Instruction Decode

Paralelizacia CPU

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

Paralelizacia CPU

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

Paralelizacia CPU

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

Paralelizacia CPU

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

Paralelizacia CPU

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

- Zakladna 5-fazova RISC architektura

Paralelizacia CPU – Pipelining

Paralelizacia CPU – Pipelining

- Paralelizacia na urovni instrukcii vykonavanych v CPU

Paralelizacia CPU – Pipelining

- Paralelizacia na urovni instrukcii vykonavanych v CPU
- Pipeline

Paralelizacia CPU – Pipelining

- Paralelizacia na urovni instrukcii vykonavanych v CPU
- Pipeline – pocet instrukcii, ktore vie CPU obsluzit behom jedneho taktu

Paralelizacia CPU – Pipelining

- Paralelizacia na urovni instrukcii vykonavanych v CPU
- Pipeline – pocet instrukcii, ktore vie CPU obsluzit behom jedneho taktu
- CPU bez pipeline obsluhuje kazdu instrukciu v krokoch uvedenych vyssie (t.j. sekvencne!)

Paralelizacia CPU – Pipelining

- Paralelizacia na urovni instrukcii vykonavanych v CPU
- Pipeline – pocet instrukcii, ktore vie CPU obsluzit behom jedneho taktu
- CPU bez pipeline obsluhuje kazdu instrukciu v krokoch uvedenych vyssie (t.j. sekvencne!)
- Takze na vykonanie jednej instrukcie potrebuje CPU priblizne 5 taktov

Paralelizacia CPU – Pipelining

- Pocas piatich faz spracovania instrukcie niektore casti CPU musia vzdy cakat, kym sa nedokonci cele spracovanie

Paralelizacia CPU – Pipelining

- Pocas piatich faz spracovania instrukcie niektore casti CPU musia vzdy cakat, kym sa nedokonci cele spracovanie
- Tu je priestor na sucasne “rozpracovanie” viacerych instrukcii naraz

Paralelizacia CPU – Pipelining

- Pocas piatich faz spracovania instrukcie niektore casti CPU musia vzdy cakat, kym sa nedokonci cele spracovanie
- Tu je priestor na sucasne “rozpracovanie” viacerych instrukcii naraz

- Tento pristup sa nazyva ILP – Instruction Level Parallelism

Paralelizacia 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

Paralelizacia 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 instrukcii trva 5 taktov
- Dokoncenie spracovania kazdej z nasledujucich sa deje v kazdom takte

Paralelizacia CPU – Pipelining

- Taketo vykonavanie ale nie je vzdy uplne spravne…

Paralelizacia CPU – Pipelining

- Taketo vykonavanie ale nie je vzdy uplne spravne…

- Uvazujme, ze CPU vykona iba nasledovne 2 instrukcie:
  - I1 – ADD 1 to R5
  - I2 – COPY R5 to R6

Paralelizacia CPU – Pipelining

- CPU vykona iba nasledovne 2 instrukcie:
  - I1 – ADD 1 to R5
  - I2 – COPY R5 to R6

Paralelizacia CPU – Pipelining

- CPU vykona iba nasledovne 2 instrukcie:
  - I1 – ADD 1 to R5
  - I2 – COPY R5 to R6

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

Paralelizacia CPU – Pipelining

- CPU vykona iba nasledovne 2 instrukcie:
  - I1 – ADD 1 to R5
  - I2 – COPY R5 to R6

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

Paralelizacia CPU – Pipelining

- CPU vykona iba nasledovne 2 instrukcie:
  - I1 – ADD 1 to R5
  - I2 – COPY R5 to R6

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

- Do R6 sa dostane nespravna hodnota!

Paralelizacia CPU – Pipelining

- Takejto situacii sa hovori “hazard”

Paralelizacia CPU – Pipelining

- Takejto situacii sa hovori “hazard”; kedze sa jedna o udaje, tak je to udajovy hazard

Paralelizacia CPU – Pipelining

- Takejto situacii sa hovori “hazard”; kedze sa jedna o udaje, tak je to udajovy hazard

- Riesi sa na urovni kompilatora!!!

Paralelizacia CPU – Superscalar

Paralelizacia CPU – Superscalar

- Rozdiel medzi Pipelining a Superscalar

Paralelizacia CPU – Superscalar

- Rozdiel medzi Pipelining a Superscalar
  - Pri pipeline vzdy iba jedna instrukcia moze byt v jednom z piatich vyssie menovanych stavov

Paralelizacia CPU – Superscalar

- Rozdiel medzi Pipelining a Superscalar
  - Pri pipeline vzdy iba jedna instrukcia moze byt v jednom z piatich vyssie menovanych stavov
  - Pri superscalar architekture moze byt viac instrukcii v jednom z tychto stavov; superskalarna architektura pridava viacero vykonavacich jednotiek instrukcii na cip

Paralelizacia CPU – Superscalar

- Rozdiel medzi Pipelining a Superscalar
  - Pri pipeline vzdy iba jedna instrukcia moze byt v jednom z piatich vyssie menovanych stavov
  - Pri superscalar architekture moze byt viac instrukcii v jednom z tychto stavov; superskalarna architektura pridava viacero vykonavacich jednotiek instrukcii na cip

- Superscalar – aspon 2 ALU!

Paralelizacia CPU – Superscalar

- Nejedna sa o paralelizaciu na urovni vlakien ci dokonca procesov!

Paralelizacia CPU – Superscalar

- Nejedna sa o paralelizaciu na urovni vlakien ci dokonca procesov!
- Aj ked nam CPU dokaze skutocne paralelne vykonavat instrukcie, stale mame iba jeden register IP (Instruction Pointer)

Paralelizacia CPU – Superscalar

- Nejedna sa o paralelizaciu na urovni vlakien ci dokonca procesov!
- Aj ked nam CPU dokaze skutocne paralelne vykonavat instrukcie, stale mame iba jeden register IP (Instruction Pointer)

- Takze sa jedna o za sebou iduce instrukcie toho isteho procesu (vlakna)

Vlastnosti CPU

- Vynikajuci vykon pre SISD

Vlastnosti CPU

- Vynikajuci vykon pre SISD (horsi pre SIMD)

Vlastnosti CPU

- Vynikajuci vykon pre SISD (horsi pre SIMD)
- Instrukcna sada CPU je bohatsia oproti GPU

Vlastnosti CPU

- Vynikajuci vykon pre SISD (horsi pre SIMD)
- Instrukcna sada CPU je bohatsia oproti GPU
- Komplexnejsia ALU, lepsia logika predikcie

Vlastnosti CPU

- Vynikajuci vykon pre SISD (horsi pre SIMD)
- Instrukcna sada CPU je bohatsia oproti GPU
- Komplexnejsia ALU, lepsia logika predikcie
- Sofistikovanejsia cache a pipeline schema

Vlastnosti CPU

- Vynikajuci vykon pre SISD (horsi pre SIMD)
- Instrukcna sada CPU je bohatsia oproti GPU
- Komplexnejsia ALU, lepsia logika predikcie
- Sofistikovanejsia cache a pipeline schema
- Cyklus instrukcie rychlejsi nez GPU

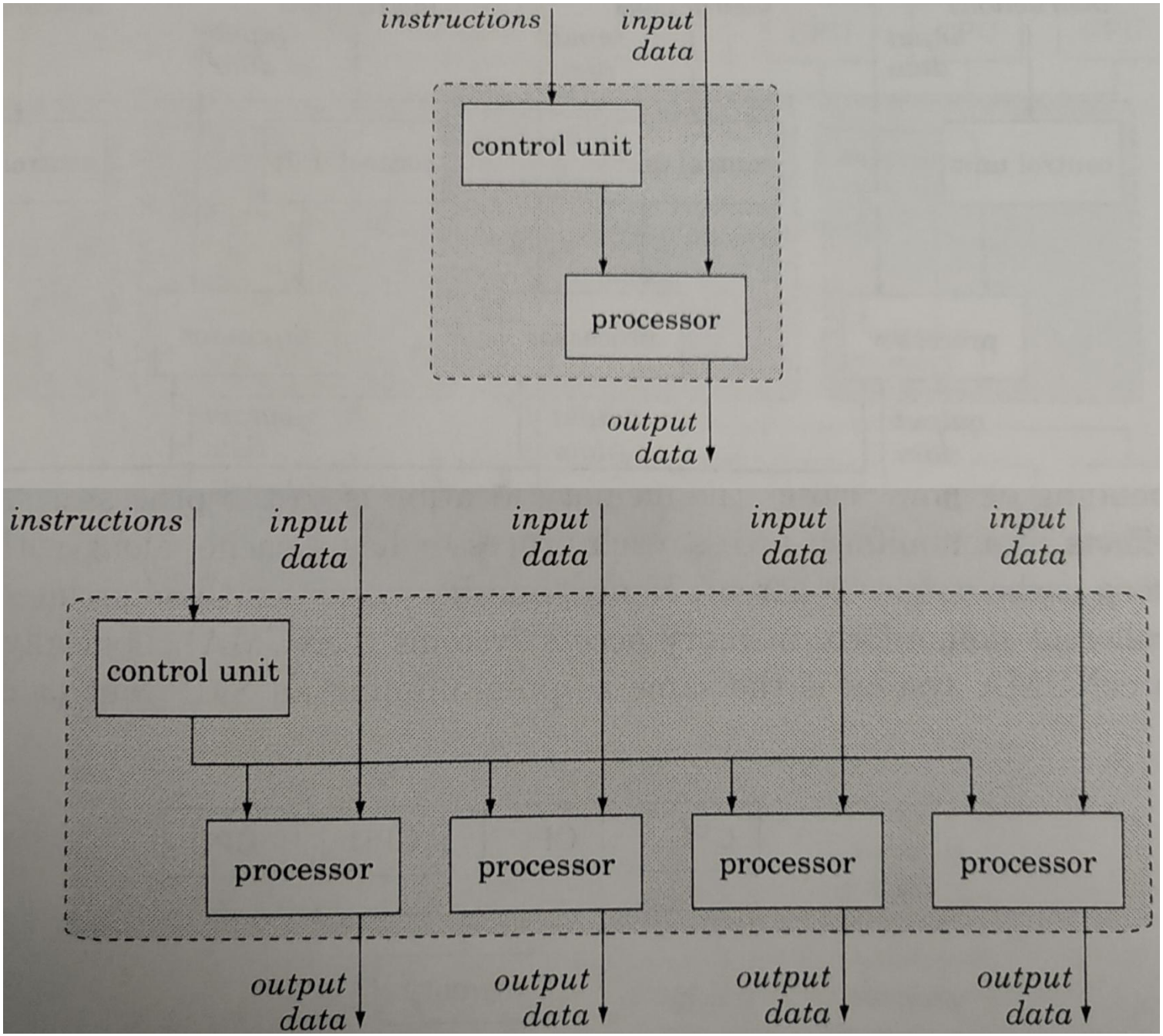
Typy vypoctov – Flynn tax.

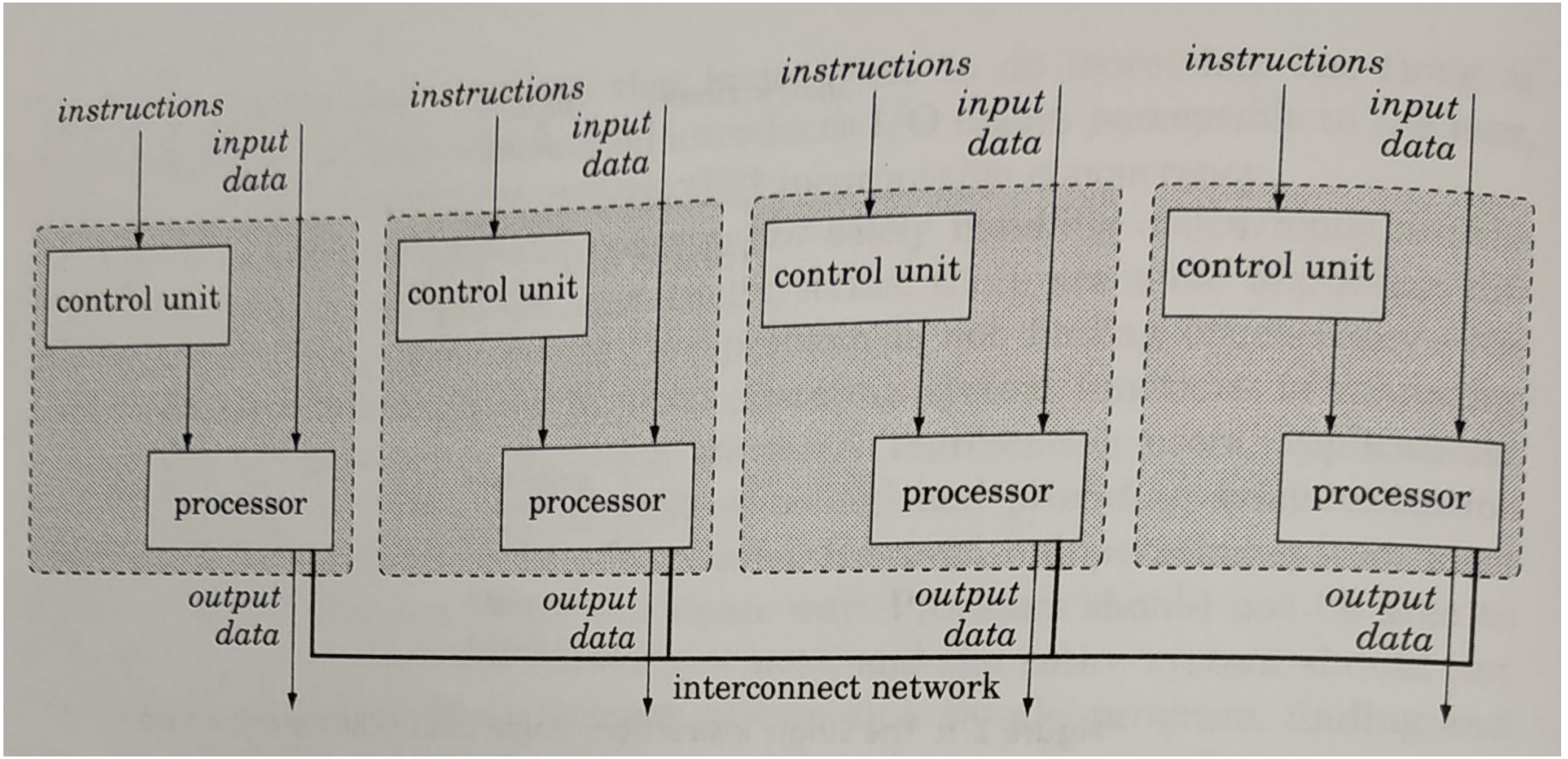
Typy vypoctov – Flynn tax.

- SISD
- SIMD
- MISD
- MIMD

Typy vypoctov – 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

- Orientovane SIMD

GPU

- Orientovane SIMD
- Pomalsie taktovanie ako CPU

GPU

- Orientovane SIMD
- Pomalsie taktovanie ako CPU
- Nema prerusenia, I/O (ako klavesnica a mys), virtualnu pamat

GPU

- Orientovane SIMD
- Pomalsie taktovanie ako CPU
- Nema prerusenia, I/O (ako klavesnica a mys), virtualnu pamat

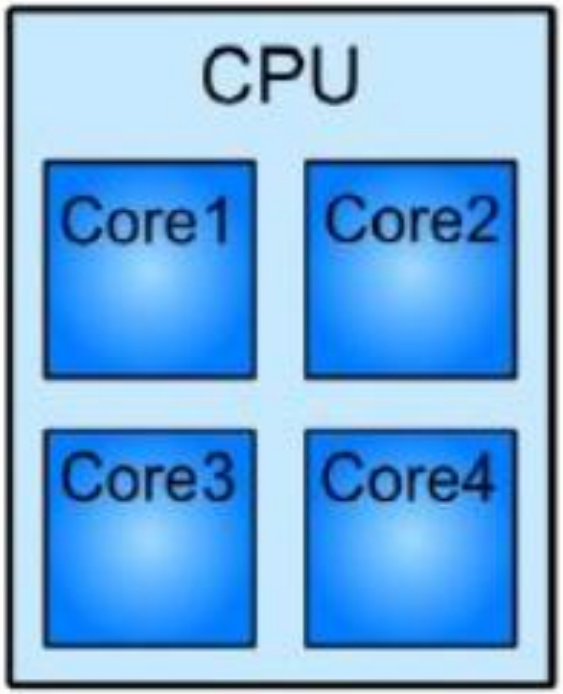
- CPU vitazi v pripade SISD (napr. hash 1 retazca)

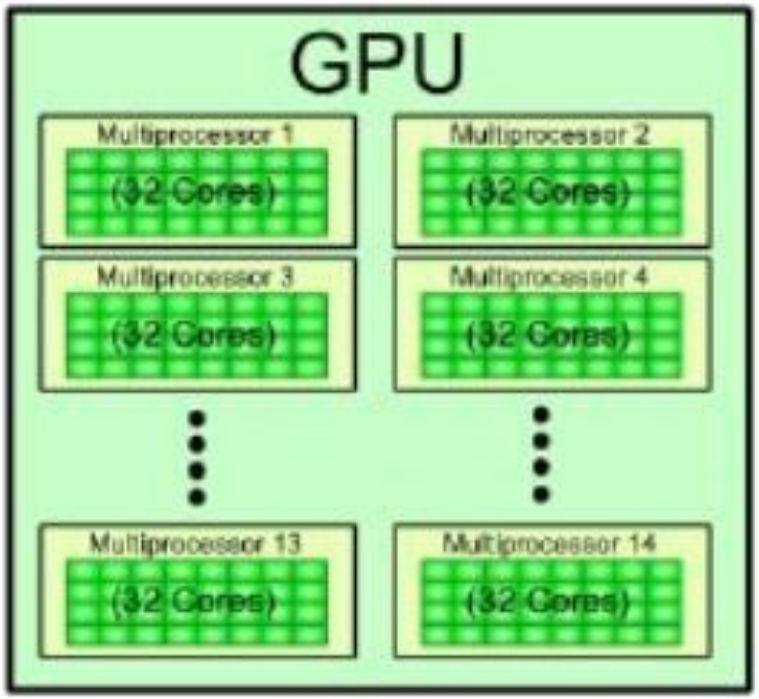
GPU

- Orientovane SIMD
- Pomalsie taktovanie ako CPU
- Nema prerusenia, I/O (ako klavesnica a mys), virtualnu pamat

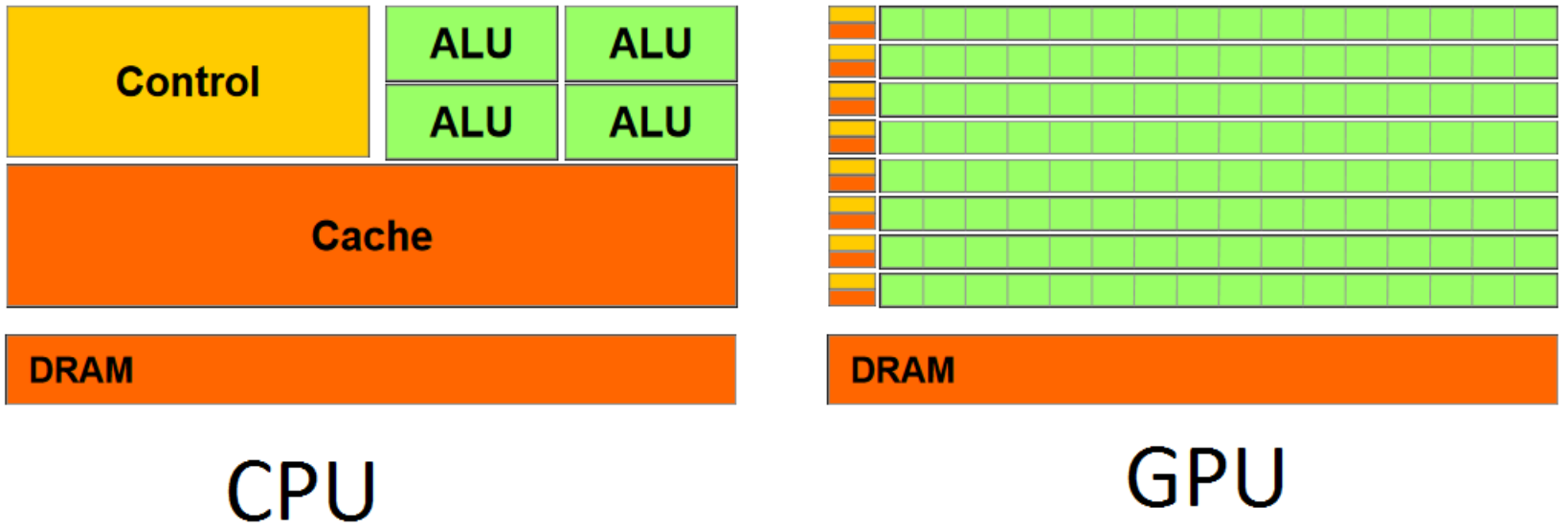
- CPU vitazi v pripade SISD (napr. hash 1 retazca); ale v pripade tisicov retazcov vitazi GPU…

Porovnanie CPU a GPU





Porovnanie CPU a GPU v2



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

Porovnanie paralelizmu

- CPU – **paralelizmus uloh**
- GPU – **paralelizmus udajov**

Porovnanie paralelizmu

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

- GPU – **paralelizmus udajov**
- Ta ista instrukcia sa vykonava na roznych udajoch (SIMD)
- **Desattisice** vlakien na stovkach jadier

Porovnanie paralelizmu

- CPU – **paralelizmus uloh**

- Viacero uloh sa mapuje na viac tokov riadenia (MIxD)
- **Desiatky** vlakien na desiatkach jadier
- Kazde vlakno nutne spravovat **softverovo**
- Kazdemu vlaknu treba urcit, co ma vykonavat

- GPU – **paralelizmus udajov**

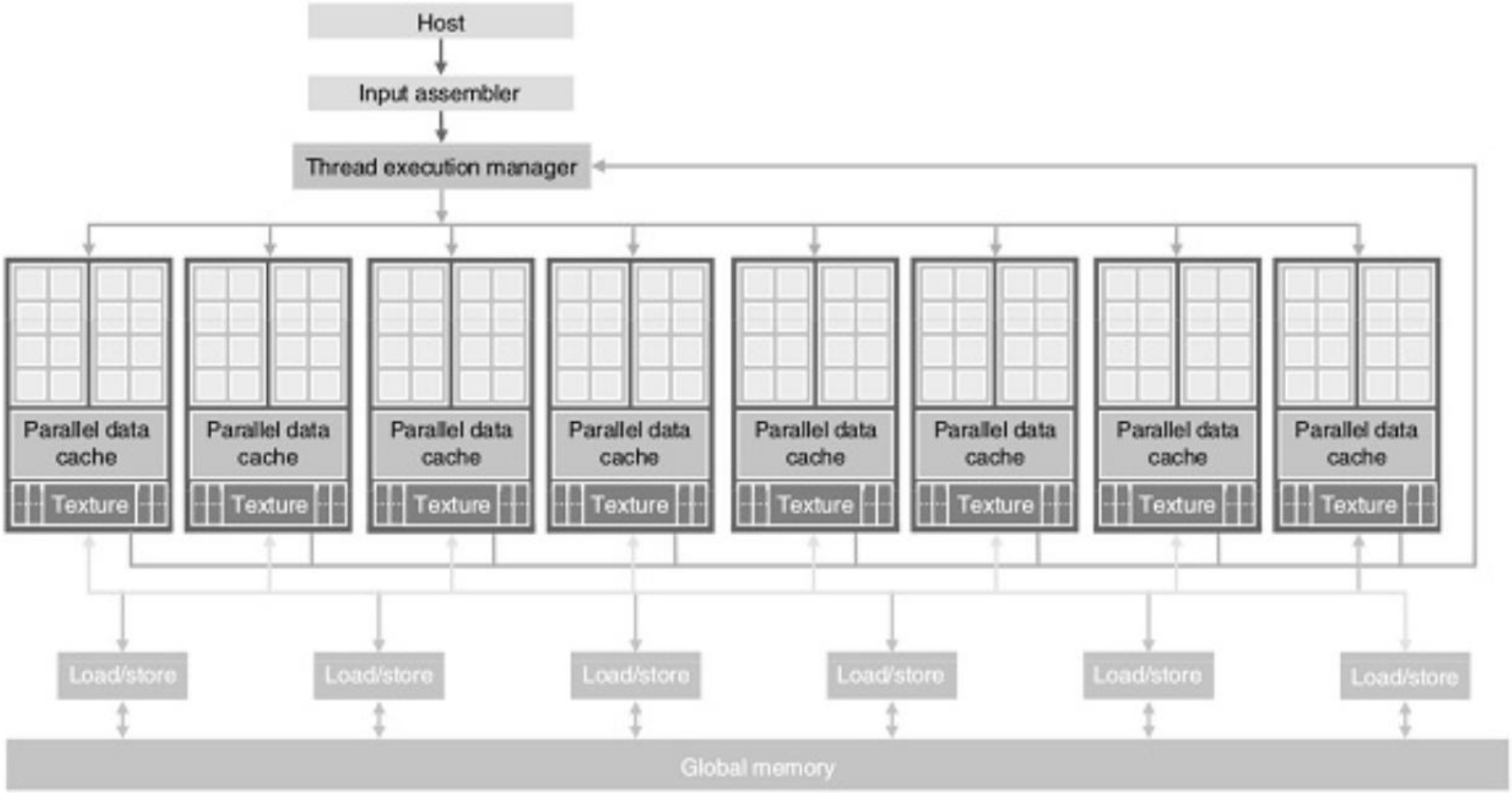
- Ta ista instrukcia sa vykonava na roznych udajoch (SIMD)
- **Desattisice** vlakien na stovkach jadier
- Kazde vlakno spravovane **hardverovo**
- Skupina vlakien ma urcene, co bude vykonavat

Vyvoj architektur NVidia

Vyvoj architektur NVidia

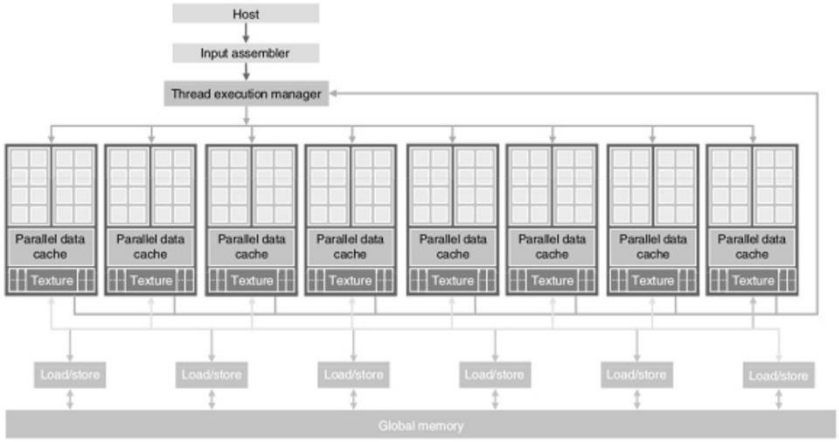
1. Pred GPGPU
   - 1995 – prva graficka karta fy Nvidia NV1
   - 1997 – Riva 128 (DX3)
   - 1998 – Riva TNT (DX5)
   - 1999 – Riva TNT2 (DX6)
2. November 2006 – G80 (128 SP / 16 SM = 8)
3. Januar 2008 – GT200 (240 SP / 30 SM = 8)
4. Fermi (512 SP / 16 SM = 32 SP per SM)

Architektura G80



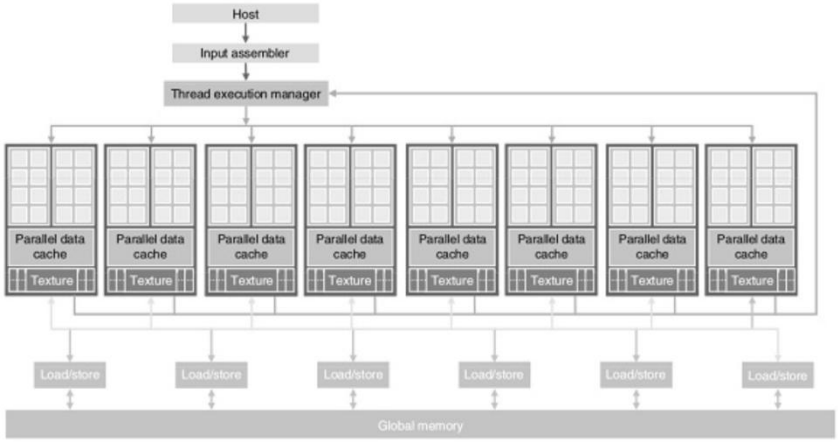
Architektura G80

- 16 SM (Streaming Multiprocessor)



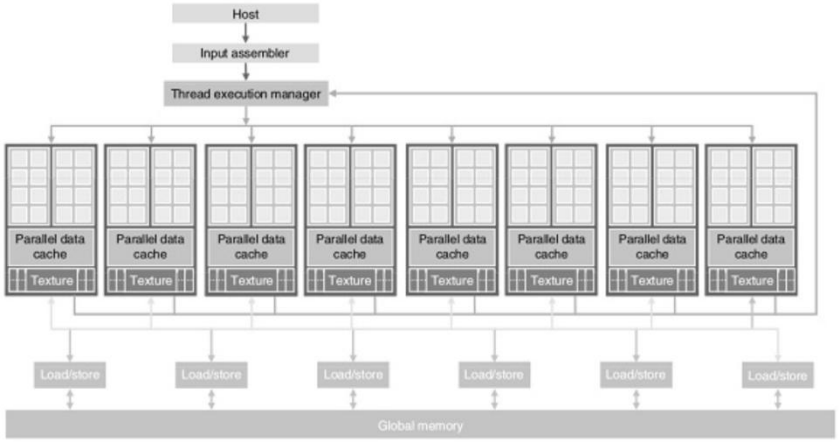
Architektura G80

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



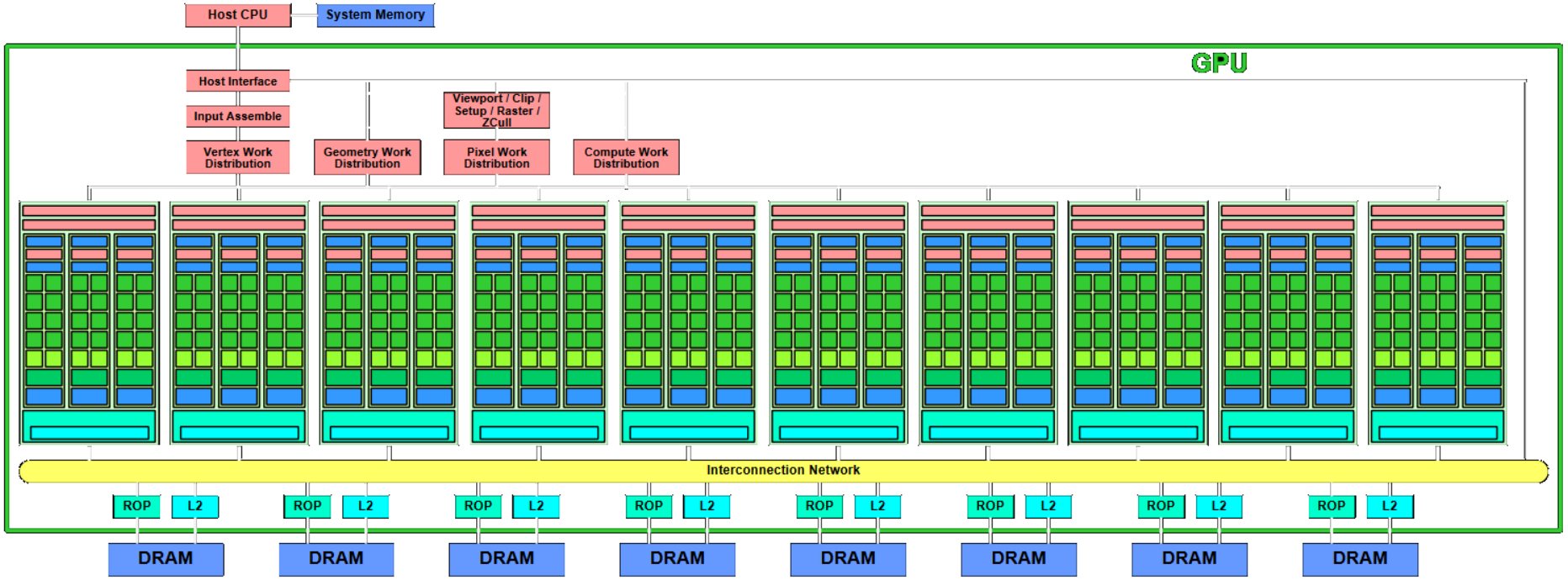
Architektura G80

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

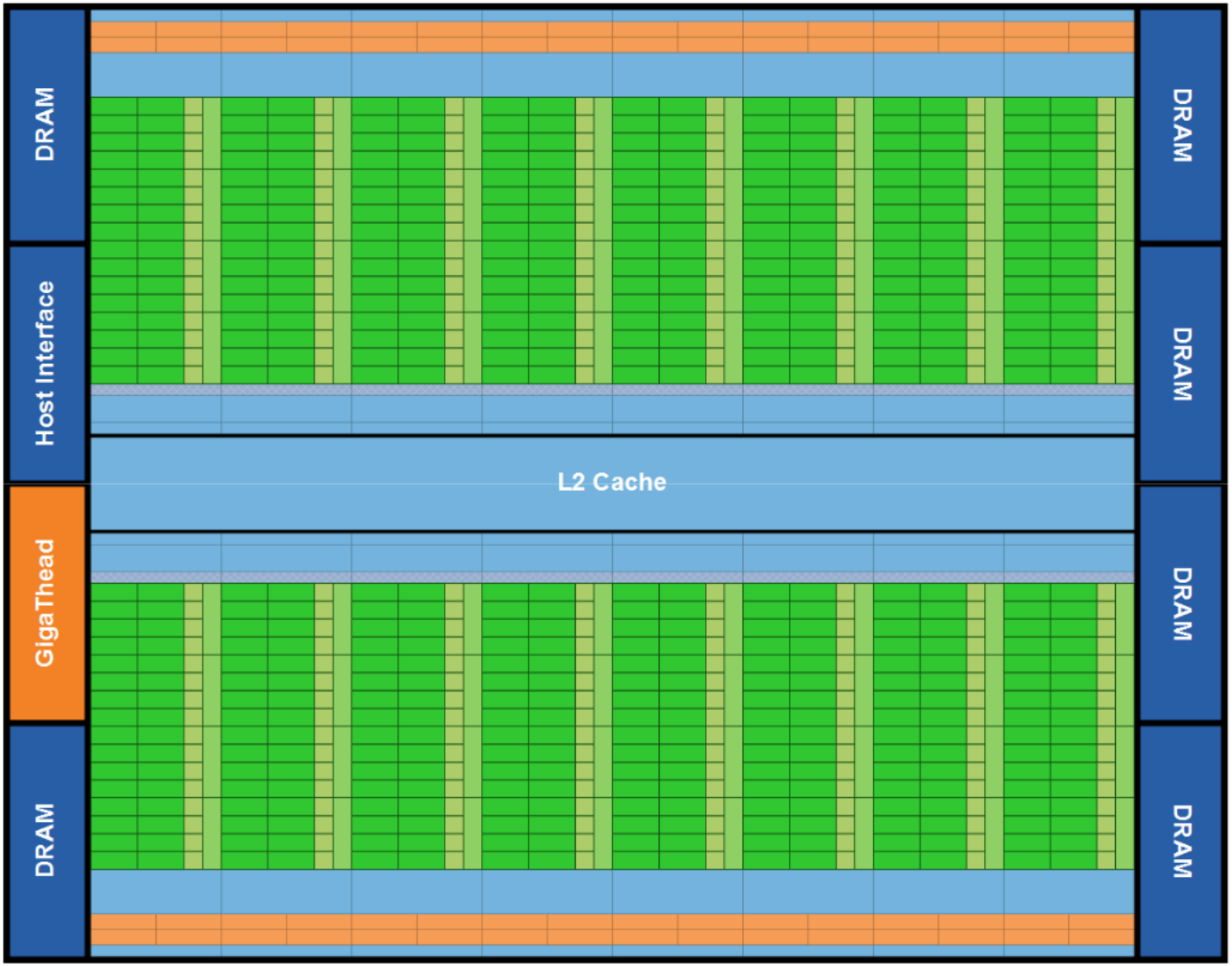


- Kazdy SP ma
  - Jednu MAD (multiply and addition unit)
  - Jednu dalsiu MU (multiply unit)

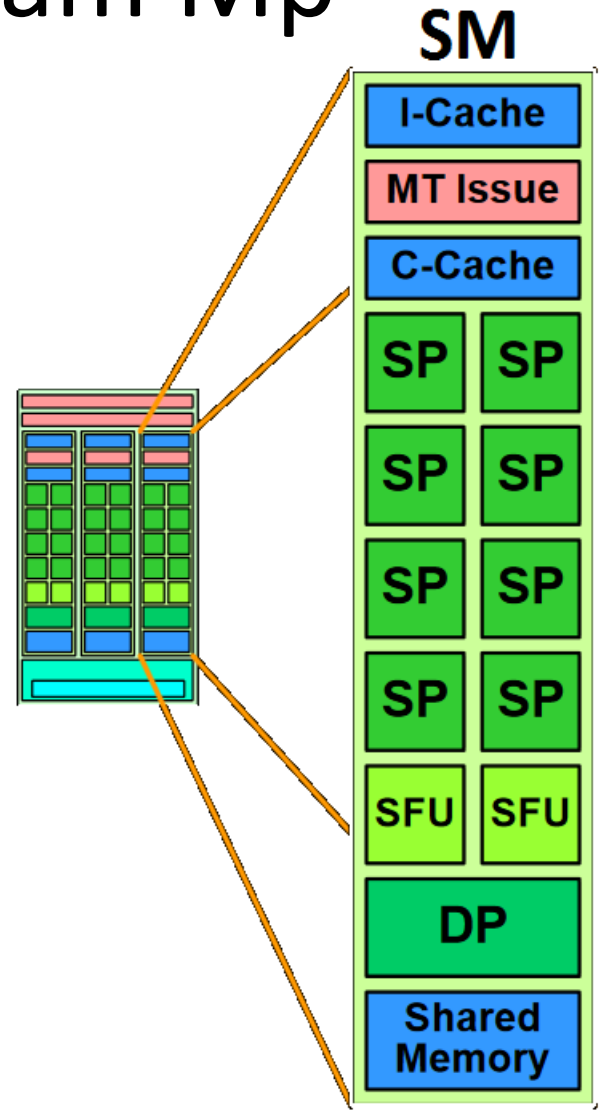
Architektura GT200



Architektura Fermi

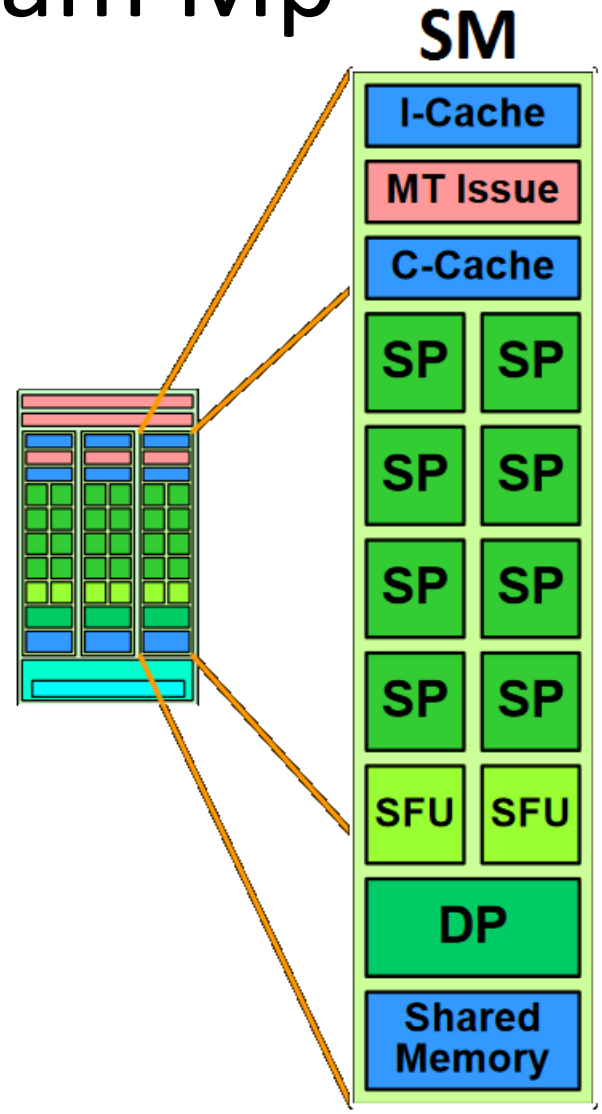


Z coho sa sklada Stream Mp



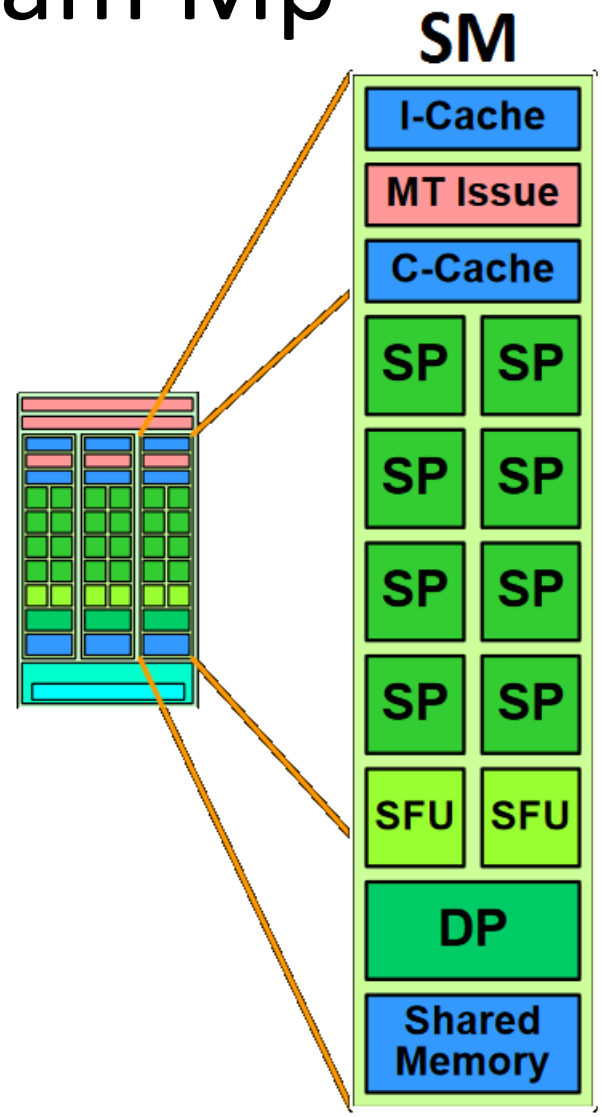
Z coho sa sklada Stream Mp

- Sprava vlakien
  - Max 1024 konkurentne (GT200)
  - Hardverove planovanie



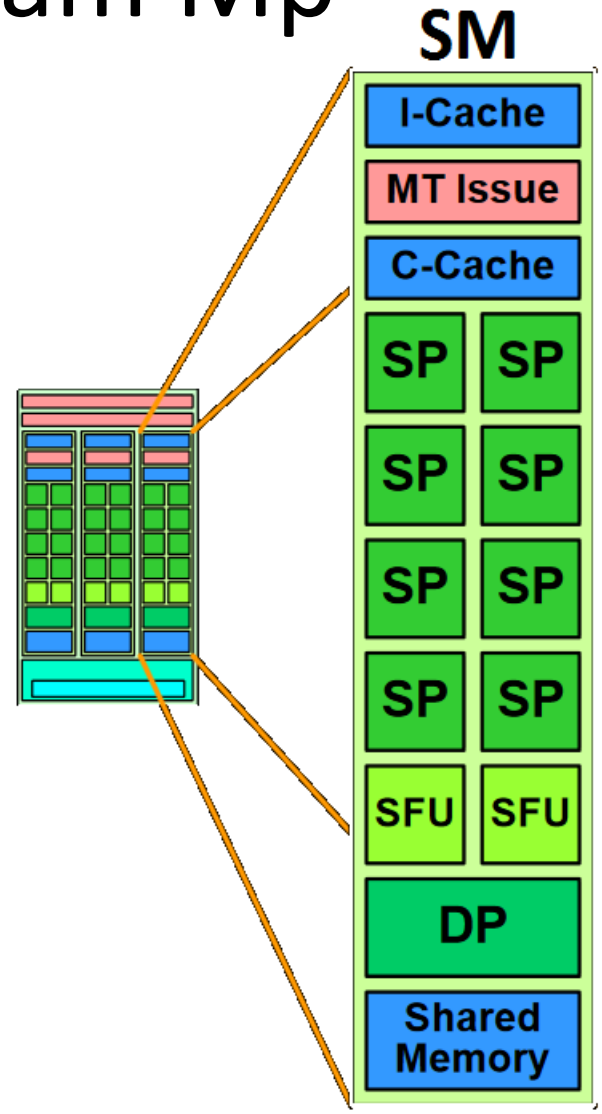
Z coho sa sklada Stream Mp

- Sprava vlakien
  - Max 1024 konkurentne (GT200)
  - Hardverove planovanie
- 8 SP
  - 32 bit floating point
  - 32 a 64 bit int



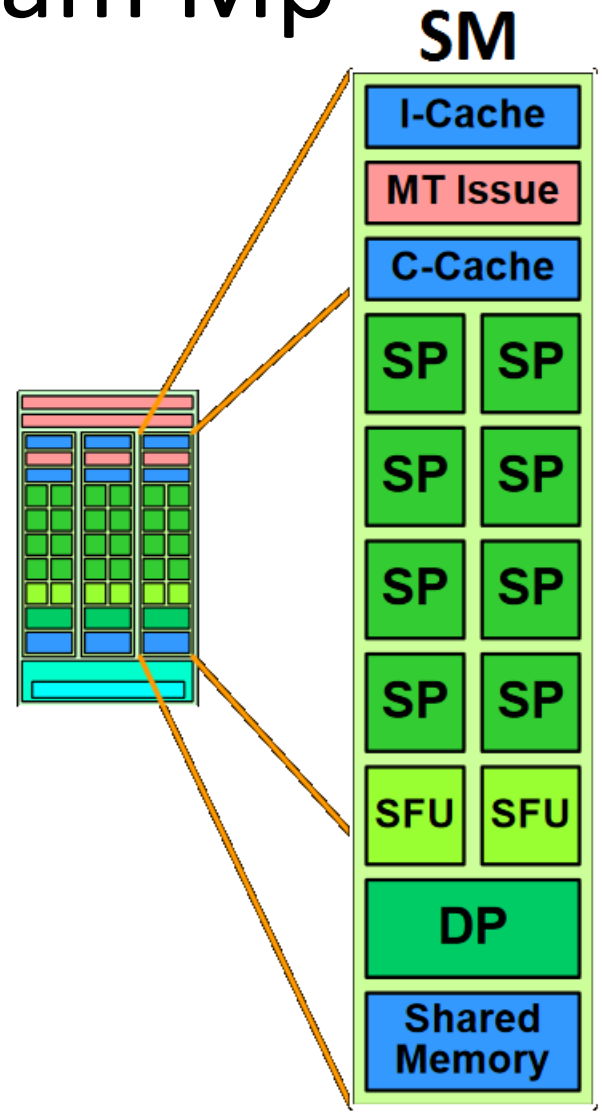
Z coho sa sklada Stream Mp

- Sprava vlakien
  - Max 1024 konkurentne (GT200)
  - Hardverove planovanie
- 8 SP
  - 32 bit floating point
  - 32 a 64 bit int
- 2 SFU
  - Jednotky spec. funkcii
  - Sin, cos, log, exp



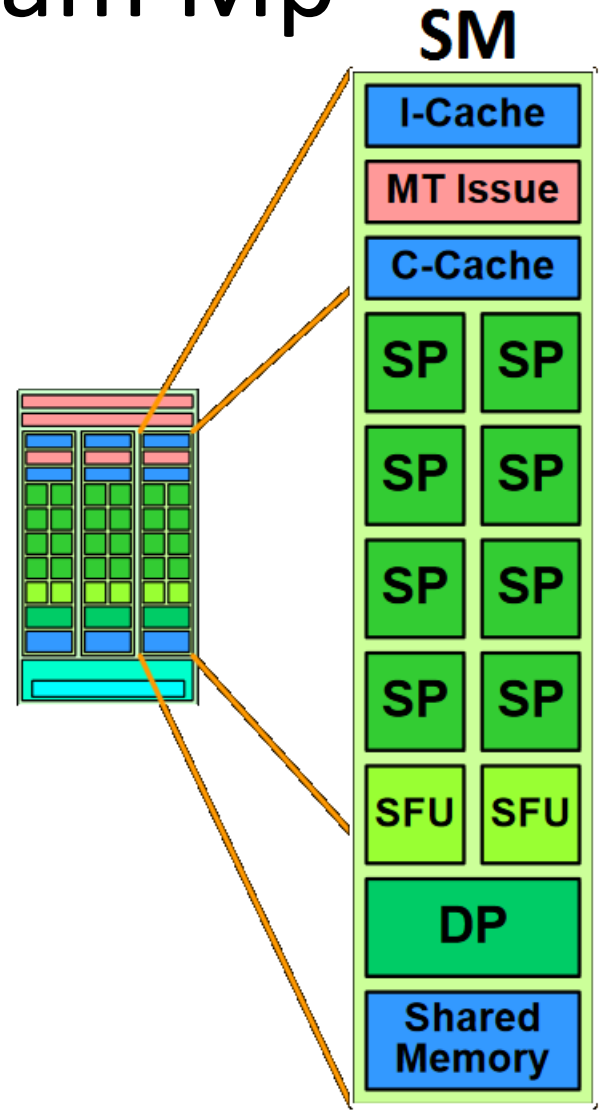
Z coho sa sklada Stream Mp

- Sprava vlakien
  - Max 1024 konkurentne (GT200)
  - Hardverove planovanie
- 8 SP
  - 32 bit floating point
  - 32 a 64 bit int
- 2 SFU
  - Jednotky spec. funkcii
  - Sin, cos, log, exp
- Double Precision Unit
  - 64 bit floating point
  - Nasobicka so spocitanim



Z coho sa sklada Stream Mp

- Sprava vlakien
  - Max 1024 konkurentne (GT200)
  - Hardverove planovanie
- 8 SP
  - 32 bit floating point
  - 32 a 64 bit int
- 2 SFU
  - Jednotky spec. funkcii
  - Sin, cos, log, exp
- Double Precision Unit
  - 64 bit floating point
  - Nasobicka so spocitanim



- 16kB zdielanej pamate

Rast vykonu

- Rok 2008: GT200 ma 240 SP s vykonom > 1 TFLOP
- Rok 2018: RTX2080 ma 68 CU,
- Rok 2020: RTX3080 ma mat 124 CU, 33 TFLOP

Struktura CUDA programu

Struktura CUDA programu

- Standardne sa pisu v C alebo Fortrane :D

Struktura CUDA programu

- Standardne sa pisu v C alebo Fortrane :D
- Zvycajne CUDA aplikacia obsahuje 2 casti

Struktura CUDA programu

- Standardne sa pisu v C alebo Fortrane :D
- Zvycajne CUDA aplikacia obsahuje 2 casti
  - Cast vykonavanu na CPU
  - Cast vykonavanu na GPU

Struktura CUDA programu

- Standardne sa pisu v C alebo Fortrane :D
- Zvycajne CUDA aplikacia obsahuje 2 casti
  - Cast vykonavanu na CPU – **host**
  - Cast vykonavanu na GPU – **device**

Struktura CUDA programu

- Standardne sa pisu v C alebo Fortrane :D
- Zvycajne CUDA aplikacia obsahuje 2 casti
  - Cast vykonavanu na CPU – **host**
  - Cast vykonavanu na GPU – **device**

- Kod vykonavany na CPU je kompilovatelny tradicnym prekladacom C

Struktura CUDA programu

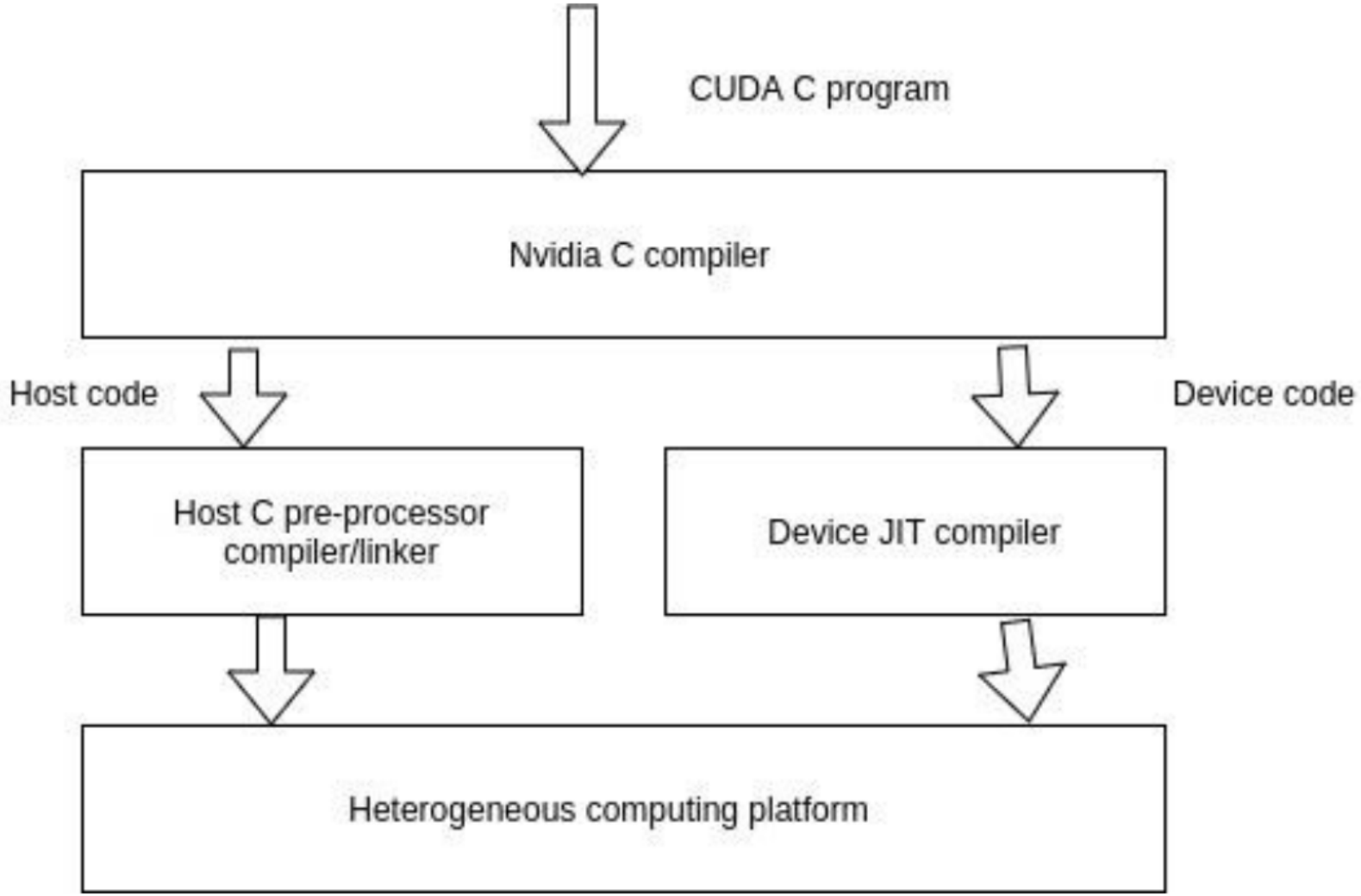
- Standardne sa pisu v C alebo Fortrane :D
- Zvycajne CUDA aplikacia obsahuje 2 casti
  - Cast vykonavanu na CPU – **host**
  - Cast vykonavanu na GPU – **device**

- Kod vykonavany na CPU je kompilovatelny tradicnym prekladacom C
- Kod na zariadeni vyzaduje specialny prekladac (Nvidia C Compiler – nvcc)

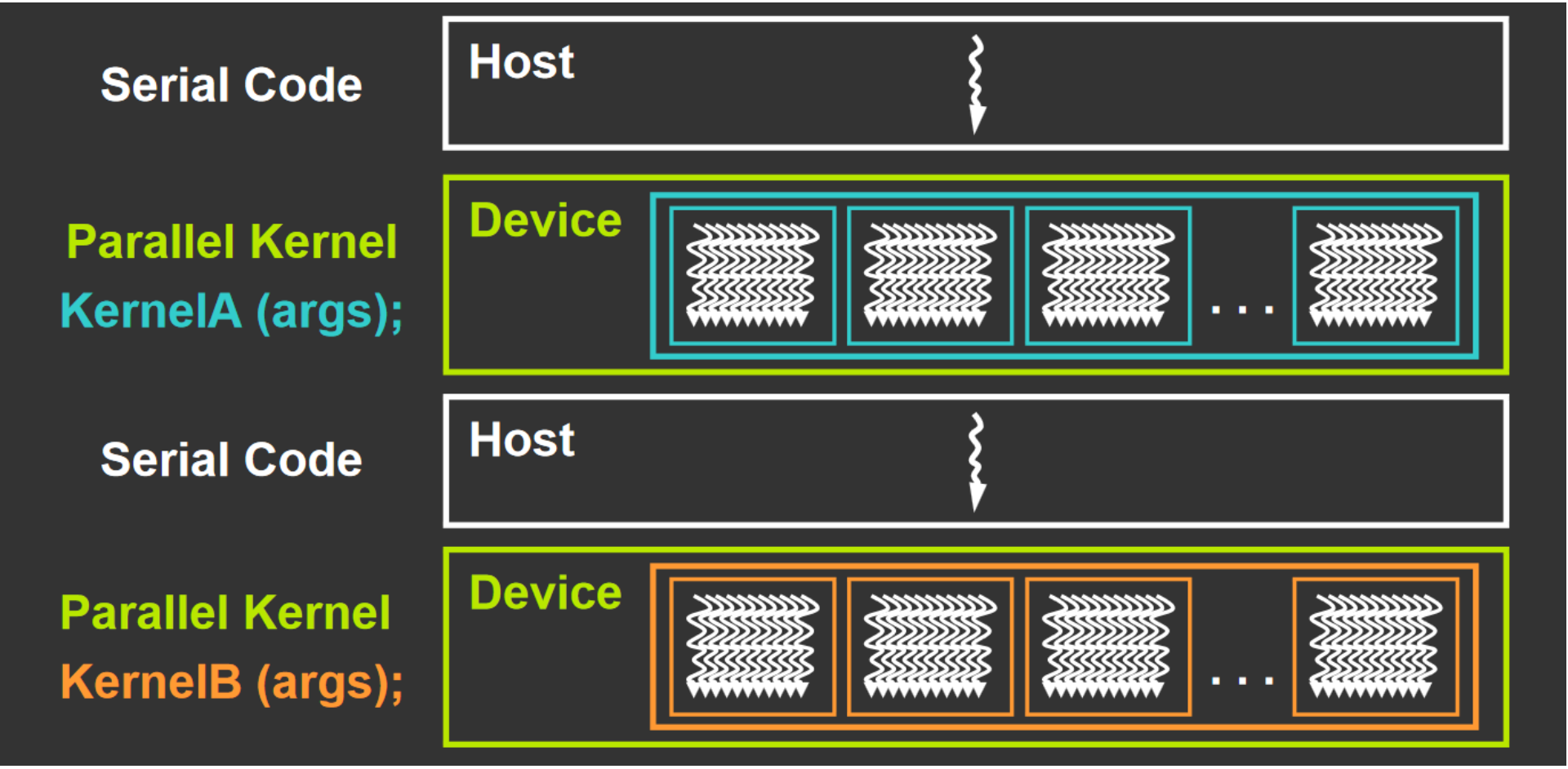
CUDA program

- NVCC vie pomocou specialnych klucovych slov rozdelit kod vykonavany na CPU a na zariadeni

CUDA program



CUDA program



HW vs SW

HW vs SW

- HW planuje VZDY 32 vlakien
  - Tzv. *warp*
  - Najmensia hw planovatelna jednotka
  - Vlakna warpu vykonavaju **naraz** tu istu instrukciu

HW vs SW

- HW planuje VZDY 32 vlakien
  - Tzv. *warp*
  - Najmensia hw planovatelna jednotka
  - Vlakna warpu vykonavaju **naraz** tu istu instrukciu
- SW bezi vo vlaknach (CUDA, nie OS)
  - Tzv. *kernel*
  - Vlakna kernelu sa zdruzuju do blokov (1536 Fermi)

HW vs SW

- SW vlakna (pokracovanie)
  - Vsetky vlakna bloku sa mozu striedat na jednom SM! Ako?

HW vs SW

- SW vlakna (pokracovanie)
  - Vsetky vlakna bloku sa mozu striedat na jednom SM! Ako? Konkurentne

HW vs SW

- SW vlakna (pokracovanie)
  - Vsetky vlakna bloku sa mozu striedat na jednom SM! Ako? Konkurentne
  - Dosledok: mozu komunikovat (pomocou zdielanej pamate)

HW vs SW

- HW a SW spolu
  - bloky vlakien hardver rozdeluje na warpy (pocet vlakien 32)
  - bloky vlakien sa v sw zoskupuju do mriezok (*grids*), kazdy grid vykonava jeden kernel

HW vs SW

- HW a SW spolu
  - bloky vlakien hardver rozdeluje na warpy (pocet vlakien 32)
  - bloky vlakien sa v sw zoskupuju do mriezok (*grids*), kazdy grid vykonava jeden kernel
- Jedna aplikacia moze spustat sucasne viacero kernelov (Fermi a vyssie)

Vykonavanie CUDA programu

Vykonavanie CUDA programu

- Programator urcuje pocet vlakien, ktore sa maju na zariadeni spustit

Vykonavanie CUDA programu

- Programator urcuje pocet vlakien, ktore sa maju na zariadeni spustit

- Vlakna tvoria 3-rozmernu organizacnu strukturu

Vykonavanie CUDA programu

- Programator urcuje pocet vlakien, ktore sa maju na zariadeni spustit

- Vlakna tvoria 3-rozmernu organizacnu strukturu
  - Vlakna tvoria blok
  - Bloky tvoria gridy

Vykonavanie CUDA programu

- Programator urcuje pocet vlakien, ktore sa maju na zariadeni spustit
- Vlakna tvoria 3-rozmernu organizacnu strukturu
  - Vlakna tvoria blok
  - Bloky tvoria gridy
- Kazde vlakno ma jedinecny identifikator!

Organizacia vlakien

Organizacia vlakien

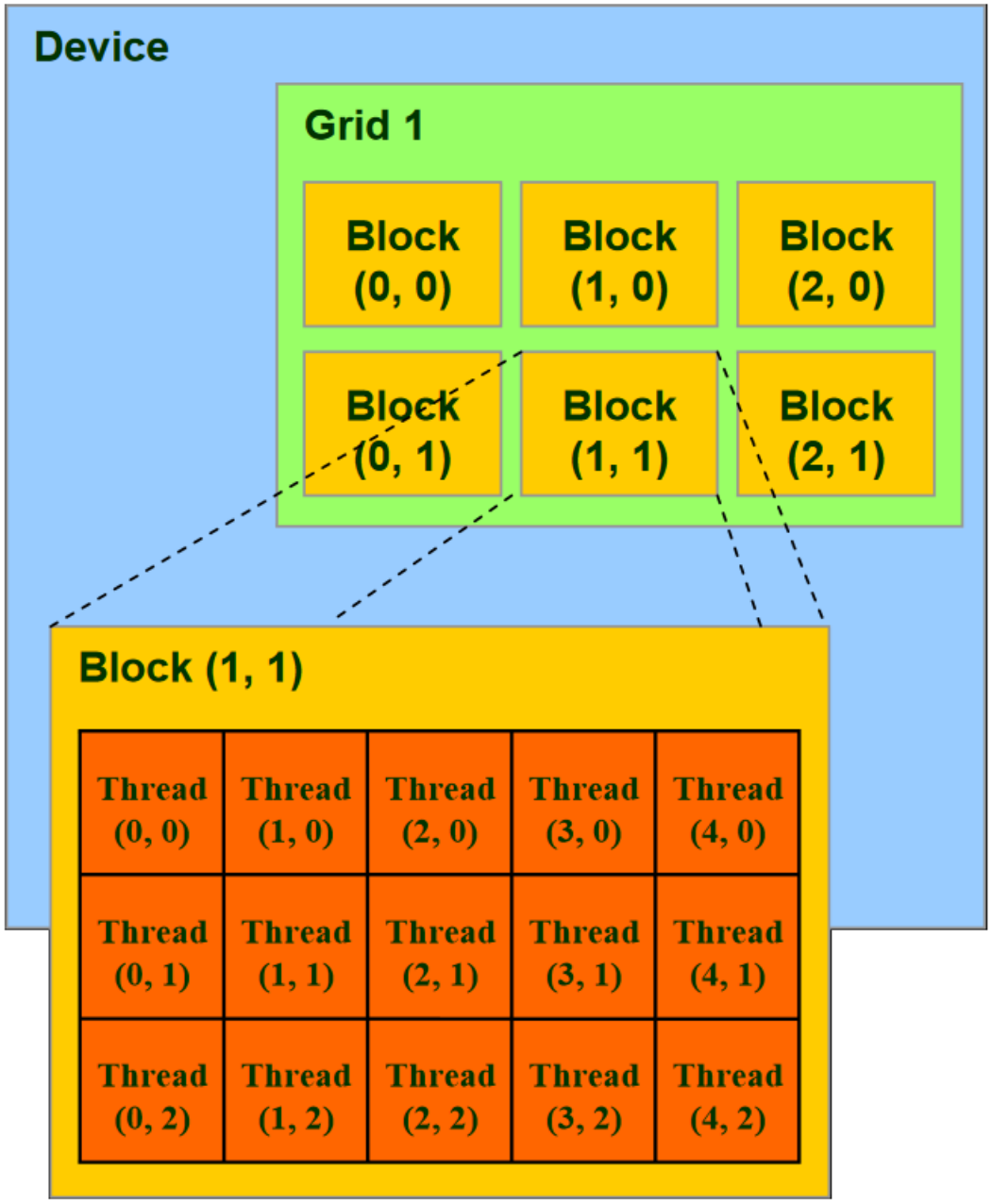
- Vsetky vlakna v gride vykonavaju tu istu funkciu jadra (kernel function)

Organizacia vlakien

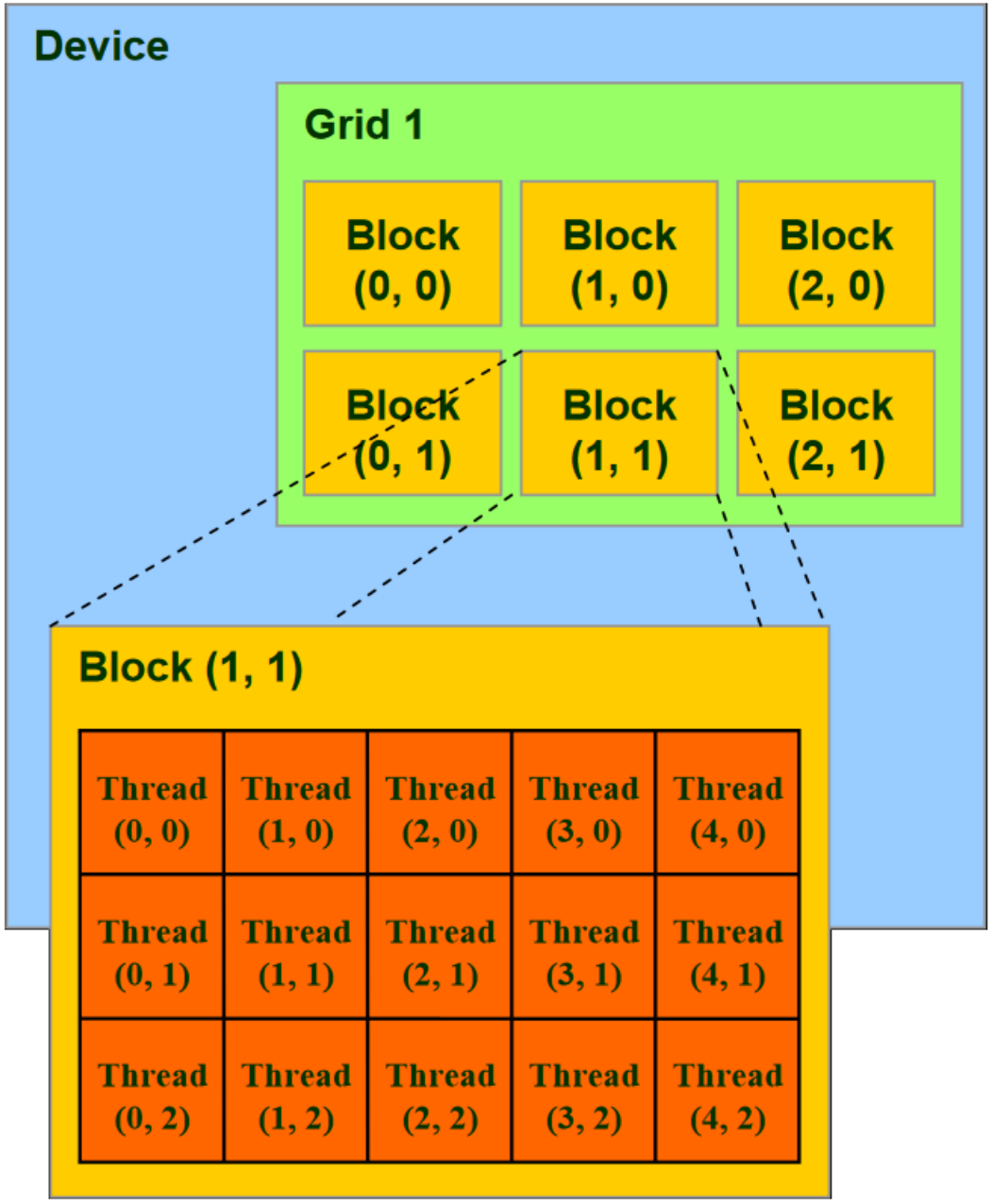
- Vsetky vlakna v gride vykonavaju tu istu funkciu jadra (kernel function)

- Ako bolo spominane, ich organizacia je dvojurovnova:
  - grid sa sklada z blokov
  - a bloky z vlakien

- Vlakna
- Bloky



- Vlakna
  - 3D id
  - V ramci bloku id vlakna unikatne
- Bloky
  - 2D id
  - V ramci gridu id bloku unikatne



Organizacia vlakien

- Vstavane premenne nvcc

Organizacia vlakien

- Vstavane premenne nvcc:
  - threadIdx – id vlakna v ramci bloku
  - blockIdx – id bloku v ramci gridu

Organizacia vlakien

- Vstavane premenne nvcc:
  - threadIdx – id vlakna v ramci bloku
  - blockIdx – id bloku v ramci gridu
  - blockDim – rozmery bloku (#vlakien v bloku)
  - gridDim – rozmery gridu (#blokov v gride)

Organizacia vlakien

- Vstavane premenne nvcc:
  - threadIdx – id vlakna v ramci bloku
  - blockIdx – id bloku v ramci gridu
  - blockDim – rozmery bloku (#vlakien v bloku)
    - Uz od CUDA 1.x ma blok moznosti 1D, 2D a 3D
  - gridDim – rozmery gridu (#blokov v gride)
    - CUDA 1.x ma 1D a 2D rozmery, az CUDA 2.x pridava 3D grid

Organizacia vlakien

- Rozmery bloku/gridu 1D, 2D, 3D su logickym pohladom na strukturu

- Rovnaky pocet vlakien je mozne identifikovat roznym sposobom (ten sa vsak fixne stanovuje pri spusteni kernelu, za behu sa pohlad na strukturu gridu nemeni)

Organizacia vlakien

- Nastavenie rozmerov gridu pomocou funkcie

- Nastavenie rozmerov bloku pomocou fnc

Organizacia vlakien

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

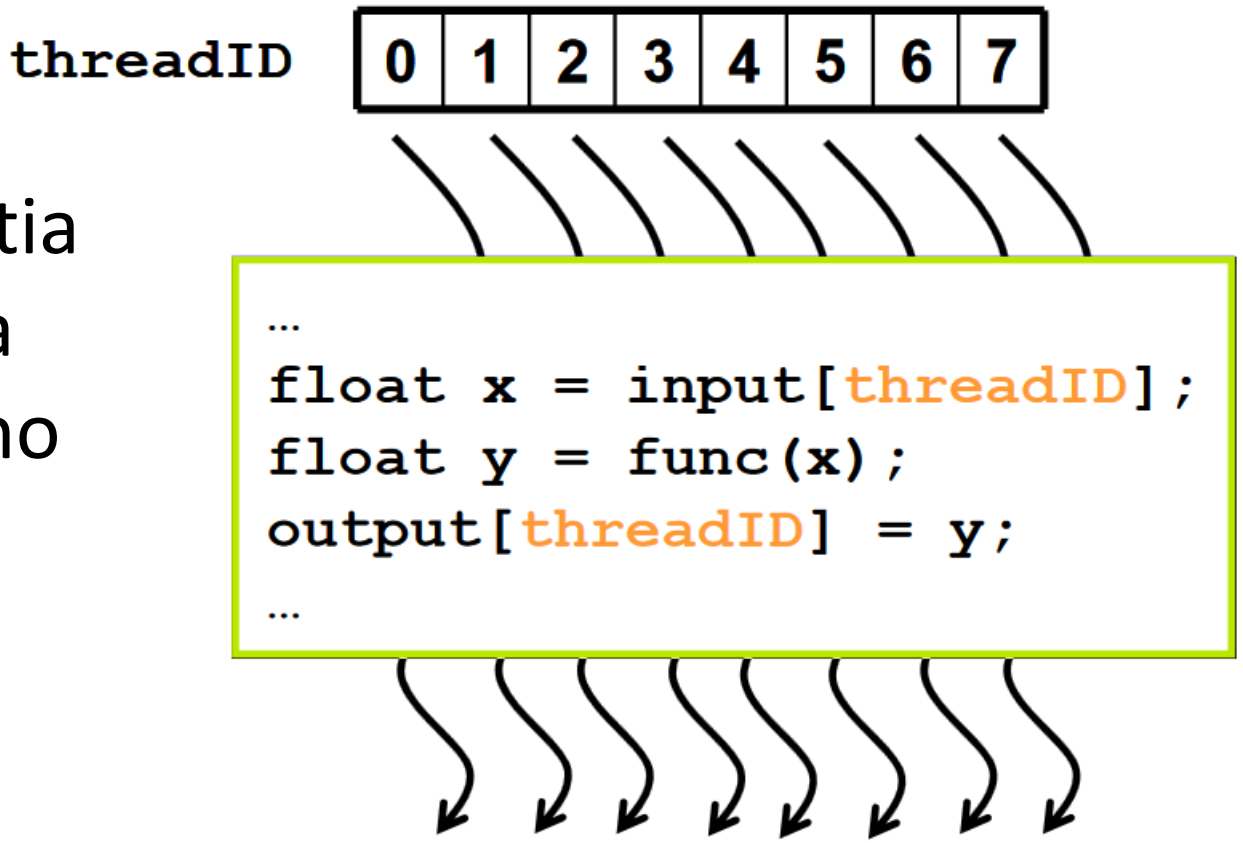
- Nastavenie rozmerov bloku pomocou fnc:
  - dim3 threadsPerBlock(mx, my, mz);

Organizacia vlakien

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

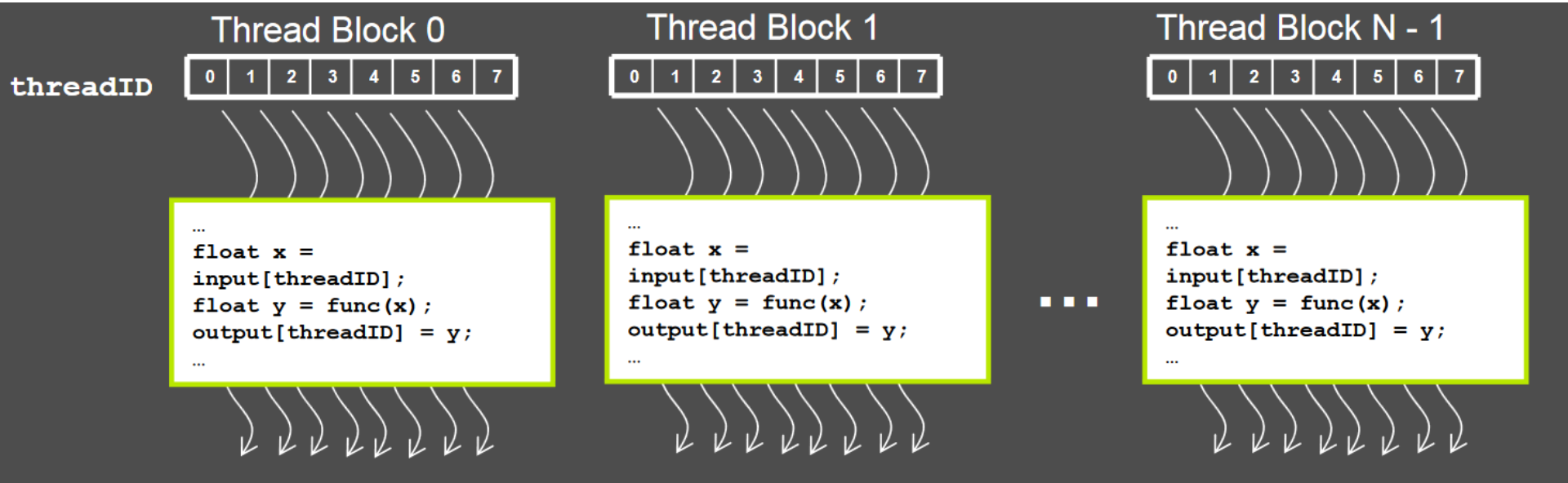
Organizacia vlakien

- Ukazka vyuzitia identifikatora vlakna pri jeho behu na GPU zariadeni



Organizacia vlakien

- Ukazka vyuzitia identifikatora vlakna pri jeho behu na GPU zariadeni
- threadID je unikatne v ramci bloku!



Organizacia vlakien

- Iny pohlad na SW versus HW

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

Organizacia vlakien - priklad

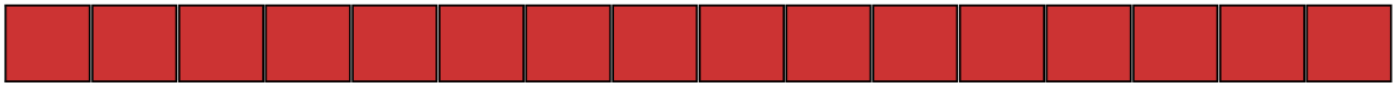
- Pripocitajme konstantu **b** ku **N**-prvkovemu polu
- Nech **N** = 16, **blockDim** = 4 → #blokov = 16/4 = 4
- Ako urcime na zaklade **blockDim**, **blockIdx** a **threadIdx** index do pola, ktory ma vlakno pouzit?

Organizacia vlakien - priklad

- Pripocitajme konstantu **b** ku **N**-prvkovemu polu
- Nech **N** = 16, **blockDim** = 4 → #blokov = 16/4 = 4
- Ako urcime na zaklade **blockDim**, **blockIdx** a **threadIdx** index do pola, ktory ma vlakno pouzit?

- idx = blockDim * blockIdx + threadIdx

Organizacia vlakien - priklad



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

Organizacia vlakien - priklad

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);
}
```

Globalna pamat zariadenia

Globalna pamat zariadenia

- O spravu pamate na zariadeni sa musi starat programator

Globalna pamat zariadenia

- O spravu pamate na zariadeni sa musi starat programator
  - Najprv musi alokovat pamat na zariadeni

Globalna pamat zariadenia

- O spravu pamate na zariadeni sa musi starat programator
  - Najprv musi alokovat pamat na zariadeni
  - Po alokovani pamate musi preniest udaje z CPU (RAM) do tejto alokovanej pamate zariadenia

Globalna pamat zariadenia

- O spravu pamate na zariadeni sa musi starat programator
  - Najprv musi alokovat pamat na zariadeni
  - Po alokovani pamate musi preniest udaje z CPU (RAM) do tejto alokovanej pamate zariadenia
  - Po ukonceni vypoctu musi preniest udaje z pamate zariadenia do pamate hosta

Globalna pamat zariadenia

- O spravu pamate na zariadeni sa musi starat programator
  - Najprv musi alokovat pamat na zariadeni
  - Po alokovani pamate musi preniest udaje z CPU (RAM) do tejto alokovanej pamate zariadenia
  - Po ukonceni vypoctu musi preniest udaje z pamate zariadenia do pamate hosta
  - Napokon musi uvolnit pamat, ktoru predtym alokoval

Zakladne typy pamati na GPU

Zakladne typy pamati na GPU

1. Globalna pamat zariadenia
   - Vsetky vlakna, cela doba behu aplikacie

Zakladne typy pamati na GPU

1. Globalna pamat zariadenia
   - Vsetky vlakna, cela doba behu aplikacie
2. Zdielana pamat
   - Vlakna jedneho bloku, pocas existencie bloku

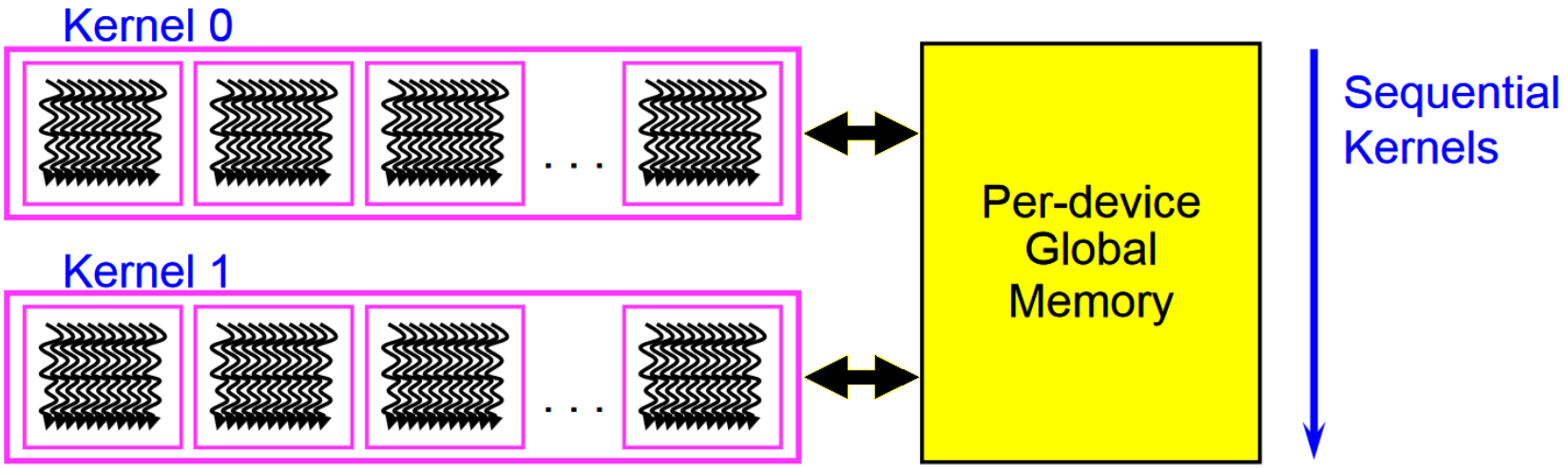
Zakladne typy pamati na GPU

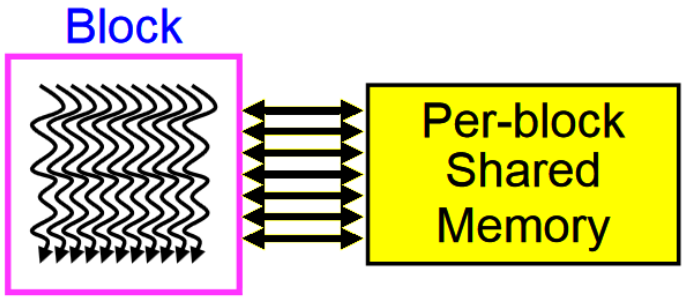
1. Globalna pamat zariadenia
   - Vsetky vlakna, cela doba behu aplikacie
2. Zdielana pamat
   - Vlakna jedneho bloku, pocas existencie bloku
3. Lokalna pamat
   - Iba pre vlakno, pocas jeho behu

Zakladne typy pamati na GPU

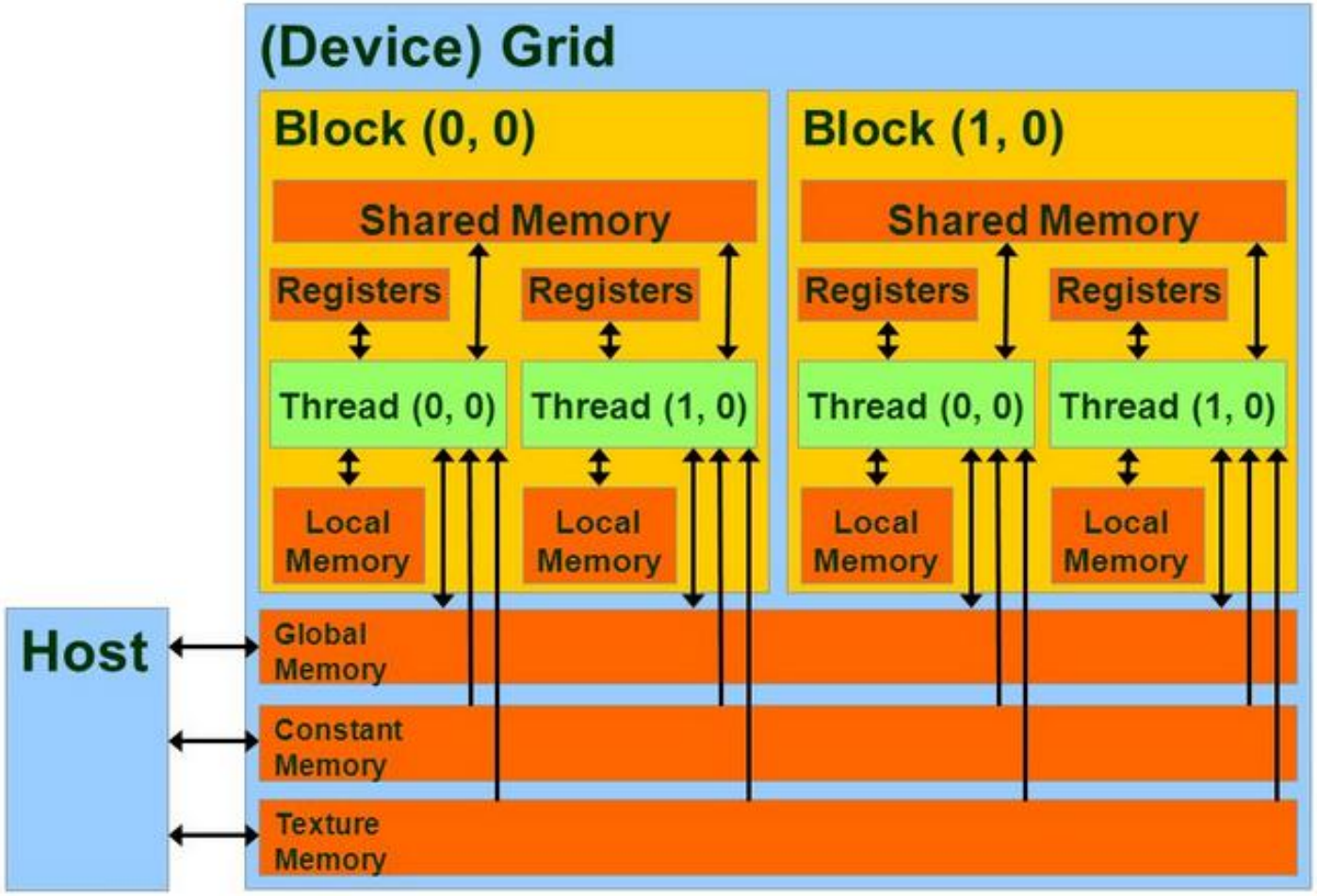
1. Globalna pamat zariadenia
   - Vsetky vlakna, cela doba behu aplikacie
   - __device__ __constant__
2. Zdielana pamat
   - Vlakna jedneho bloku, pocas existencie bloku
   - __shared__
3. Lokalna pamat
   - Iba pre vlakno, pocas jeho behu

Zakladne typy pamati na GPU





Pamatovy model GPU



[CUDA C Programming Guide Version 3.2]

Typy funkcii

Typy funkcii

- **__global__** deklaruje tzv. kernel funkciu, ktora je volana z hosta, ale vykonava sa na zariadeni

Typy funkcii

- **__global__** deklaruje tzv. kernel funkciu, ktora je volana z hosta, ale vykonava sa na zariadeni
- **__device__** deklaruje funkciu, ktora je volana zo zariadenia, a moze byt iba na nom vykonavana

Typy funkcii

- **__global__** deklaruje tzv. kernel funkciu, ktora je volana z hosta, ale vykonava sa na zariadeni
- **__device__** deklaruje funkciu, ktora je volana zo zariadenia, a moze byt iba na nom vykonavana
- **__host__** deklaruje funkciu, ktora je volana z hosta a moze byt vykonavana iba na nom

Ukazka 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);
}
```

Ukazka 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);
}
```

Ukazka 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);
}
```

Deklaracia a vyvolanie kernelu

- Deklaracia kernelu:

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

  Navratovy typ kernelu musi byt vzdy **void**!

- Spustenie kernelu:

  dim3 blocks(nx, ny, nz);

  dim3 threadsPerBlock(mx, my, mz);

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

Nasobenie 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

- Reprezentacia 2D matice v pamati

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

- Reprezentacia 2D matice v pamati
  - Po riadkoch
  - Po stlpcoch

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

- Reprezentacia 2D matice v pamati
  - 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 stlpcoch

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

- Adresacia prvku 2D matice reprezentovanej 1D polom “po riadkoch”

Matice

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

Matice

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

Matice

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

- Kazdy jeden prvok mozeme mapovat prave na jedno vlakno na GPU

Matice

- Kazdy jeden prvok mozeme mapovat prave na jedno vlakno na GPU
  - Vdaka vstavanym identifikatorom

Matice

- Kazdy jeden prvok mozeme mapovat prave na jedno vlakno na GPU
  - Vdaka vstavanym identifikatorom
  - row = blockIdx.x * blockDim.x + threadIdx.x
  - col = blockIdx.y * blockDim.y + threadIdx.y

Matice

- Kazdy jeden prvok mozeme mapovat prave na jedno vlakno na GPU
  - Vdaka vstavanym identifikatorom
  - row = blockIdx.x * blockDim.x + threadIdx.x
  - col = blockIdx.y * blockDim.y + threadIdx.y

- Nasobenie matic ovladame…

```
__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 ladit CUDA program?

Ako ladit CUDA program?

- Mod emulacie zariadenia!

Ako ladit CUDA program?

- Mod emulacie zariadenia!

- Treba program skompilovat pomocou prepinaca: “nvcc –deviceemu”
  - Zariadenie bude emulovane
  - Netreba ziadnu graficku kartu ani ovladace
  - Kazde vlakno zariadenia bude emulovane vlaknom hosta!

Ako ladit CUDA program?

- Ake su vyhody takeho pristupu

Ako ladit CUDA program?

- Ake su vyhody takeho pristupu

- Nativna podpora ladiacich nastrojov
- Pristup ku premennym zariadenia v kode hosta a opacne
- Povolene volanie lubovolnej funkcie hosta z funkcie zariadenia (napr. printf) a opacne

CUDA v Pythone

CUDA v Pythone

- Jestvuje viacero moznosti

CUDA v Pythone

- Jestvuje viacero moznosti
- Nvidia oficialne uvadza 2

CUDA v Pythone

- Jestvuje viacero moznosti
- Nvidia oficialne uvadza 2
  - PyCUDA
  - Anaconda

CUDA v Pythone

- Jestvuje viacero moznosti

- Nvidia oficialne uvadza 2
  - PyCUDA
    - https://developer.nvidia.com/pycuda
  - Anaconda
    - https://developer.nvidia.com/anaconda-accelerate

CUDA v Pythone

- Jestvuje viacero moznosti

- Nvidia oficialne uvadza 2
  - PyCUDA – treba pisat C kod v Pythone ;)
    - https://developer.nvidia.com/pycuda
  - Anaconda – cisty Python
    - https://developer.nvidia.com/anaconda-accelerate

CUDA v Pythone

- Zvolili sme iba minimalnu cast ekosystemu Anaconda Accelerate s nazvom Numba

CUDA v Pythone

- Zvolili sme iba minimalnu cast ekosystemu Anaconda Accelerate s nazvom Numba

- Anaconda Accelerate je volna Python distribucia pre enterprise riesenia od Continuum Analytics urcena na vedecke vypocty, prediktivnu analyzu, spracovanie velkeho objemu dat

Zdroje

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

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

CUDA pomocou Numba

CUDA pomocou Numba

- Jednym z dovodov volby kniznice Numba je to, ze dokaze emulovat GPU (takze na precvicenie nepotrebujeme realny hw)

CUDA pomocou Numba

- Jednym z dovodov volby kniznice Numba je to, ze dokaze emulovat GPU (takze na precvicenie nepotrebujeme realny hw)
- Ako zapnut emulator?

CUDA pomocou Numba

- Jednym z dovodov volby kniznice Numba je to, ze dokaze emulovat GPU (takze na precvicenie nepotrebujeme realny hw)
- Ako zapnut emulator?
  - Linux
  - Windows

CUDA pomocou Numba

- Jednym z dovodov volby kniznice Numba je to, ze dokaze emulovat GPU (takze na precvicenie nepotrebujeme realny hw)
- Ako zapnut emulator?
  - Linux: export NUMBA_ENABLE_CUDASIM=1
  - Windows: set NUMBA_ENABLE_CUDASIM=1

CUDA pomocou Numba

- Jednym z dovodov volby kniznice Numba je to, ze dokaze emulovat GPU (takze na precvicenie nepotrebujeme realny hw)
- Ako zapnut emulator?
  - Linux: export NUMBA_ENABLE_CUDASIM=1
  - Windows: set NUMBA_ENABLE_CUDASIM=1

  - Az potom zapnut Idle alebo Python shell !!!

Zdroje prednasky

- 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 prednasky

- 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/

- Vyskusat CUDA programovanie pomocou simulatora Numba v Pythone
- Kto ma kartu Nvidia, moze vyuzit priamo hardver

Dakujem za pozornost!