GPU a CUDA

Historie GPU

Co je GPGPU?

Nvidia CUDA

Historie GPU

- **GPU = graphics processing unit**
- jde o akcelerátory pro algoritmy v 3D grafice a vizualizaci
- mnoho z nich původně vzniklo pro účely počítačových her
  - to byla často psychologická nevýhoda GPU
- typická úloha ve vizualizaci vypadá takto
  - transformování miliónů polygonů
  - aplikování textur o velikosti mnoha MB
  - projekce na framebuffer
- **žádná datová závislost**

Historie GPU

- 1970 - ANTIC in 8-bit Atari
- 1980 - IBM 8514
- 1993 - Nvidia Co. založeno
- 1994 - 3dfx Interactive založeno
- 1995 - chip NV1 od Nvidia
- 1996 - 3dfx vydalo Voodoo Graphics
- 1999 - GeForce 256 by Nvidia - podpora geometrických transformací
- 2000 - Nvidia kupuje 3dfx Interactive
- 2002 - GeForce 4 vybaveno pixel a vertex shadery
- 2006 - GeForce 8 - unifikovaná architektura (nerozlišuje pixel a vertex shader) (Nvidia CUDA)
- 2008 - GeForce 280 - podpora dvojité přesnosti
- 2010 - GeForce 480 (Fermi) - první GPU postavené pro obecné výpočty - GPGPU

Výhody GPU

(Nvidia GeForce GTX 580)

- GPU je navrženo pro **současný** běh až 512 vláken - virtuálně až stovek tisíc vláken
- vlákna musí být **nezávislá** - není zaručeno, v jakém pořadí budou zpracována
- GPU je vhodné pro kód s intenzivními výpočty a s malým výskytem podmínek
- není zde podpora spekulativního zpracování
- ~~není zde podpora pro cache~~
- GPU je optimalizováno pro **sekvenční přístup do paměti** - 194 GB/s

VÝHODY GPU

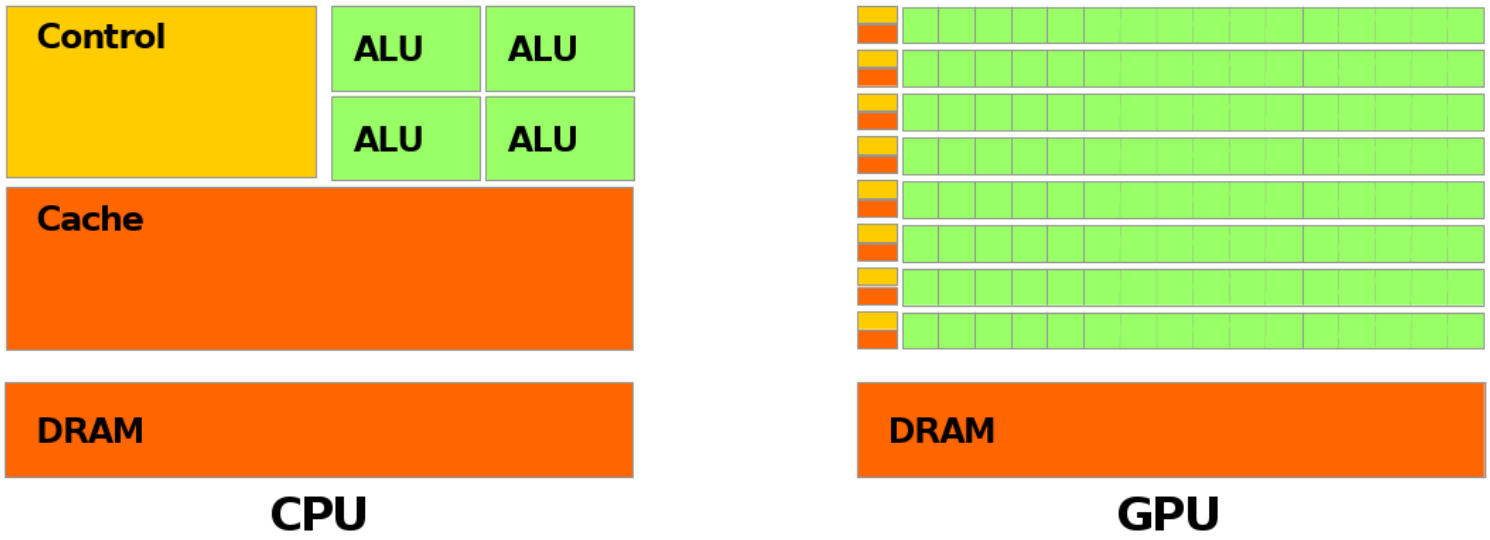


FIGURE: Zdroj Nvidia Programming Guide

Porovnání CPU vs. GPU

Za přibližně **500 EUR** lze koupit

	Nvidia GeForce 580	**INTEL Core i7-970 Six-Core**
Transistors	3 000 millions	??? 731 millions ???
Clock	1.5 GHz	3.5 GHz
Threads Num.	512	12
Peak. Perf.	1770 GFlops	$\approx$ 200 GFlops
Bandwidth	194 GB/s	25.6 GB/s
RAM	1.5 GB	$\approx$ 48 GB
Power	244 W	130 W

Co je GPGPU?

- mějmě obdélník a pokryjme ho texturou s rozlišením 800x600 pixels
- promítneme ho jedna ku jedné do framebuferu/na obrazovku s rozlišením 800x600 pixelů
- co když použijeme dvě textury a alfa blending

$$T(i,j) = \alpha_1 T_1(i,j) + \alpha_2 T_2(i,j), \text{ for all pixels } (i,j)$$

- dostáváme váženou sumu dvou matic z $\mathbb{R}^{800,600}$ a výsledek je uložen ve framebuferu/na obrazovce

HISTORIE OF GPGPU

- **GPGPU = General Purpose Computing on GPU** (www.gpgpu.org)
- Lengyel, J., Reichert, M., Donald, B.R. and Greenberg, D.P. *Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware*. In Proceedings of SIGGRAPH 1990, 327-335. 1990.
- 2003 - GPGPU na běžných GPUs

Podstata GPGPU

- na počátku bylo nutné pro GPGPU využívat rozhraní OpenGL
- úlohy byly formulovány pomocí **textur a operací s pixely**
- vývojáři her ale potřebovali flexibilnější hardware $\Rightarrow$ vznikly pixel shadery
  - jde o jednoduchý programovatelný procesor pro operace s pixely
  - má podporu pro výpočty s plovoucí desetinnou čárkou v jednoduché přesnosti
  - velikost kódu byla omezena na několik desítek instrukcí

NVIDIA CUDA

CUDA = Compute Unified Device Architecture - Nvidia 15 February 2007

- výrazně zjednodušuje programování v GPGPU
- zcela odstraňuje nutnost pracovat s OpenGL a formulování úloh pomocí textur
- je založena na jednoduchém rozšíření jazyka C/C++
- funguje jen s kartami společnosti Nvidia

Je velice snadné napsat kód pro CUDA ale je potřeba mít hluboké znalosti o GPU aby byl výsledný kód efektivní.

CUDA ARCHITEKTURA I.

Architektura Fermi

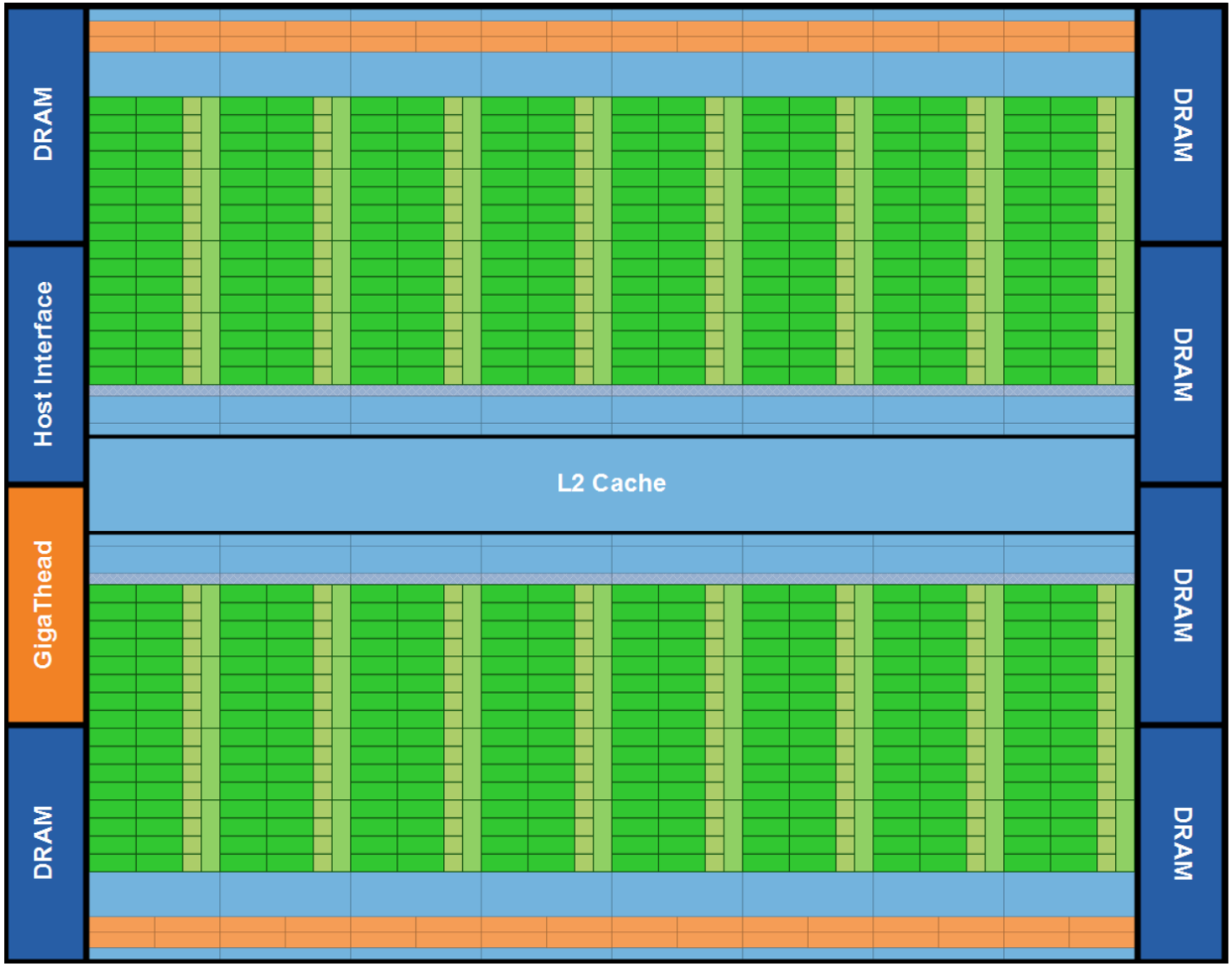


FIGURE: Zdroj Nvidia

CUDA ARCHITEKTUR I.

GeForce 580

- 16 multiprocesorů (**Streaming Multiprocessors**) - každý má
  - 32 jader/procesorů pro jednotlivá vlákna
  - 64 kB velmi rychlé paměti, která může částečně fungovat jako cache
  - tato paměť se dělí do 16 modulů - jeden pro každé vlákno
- šest 64-bitových paměťových modulů pro 384-bitový přístup a až 6 GB RAM
- 768 kB L2 Cache

CUDA ARCHITEKTUŘE II.

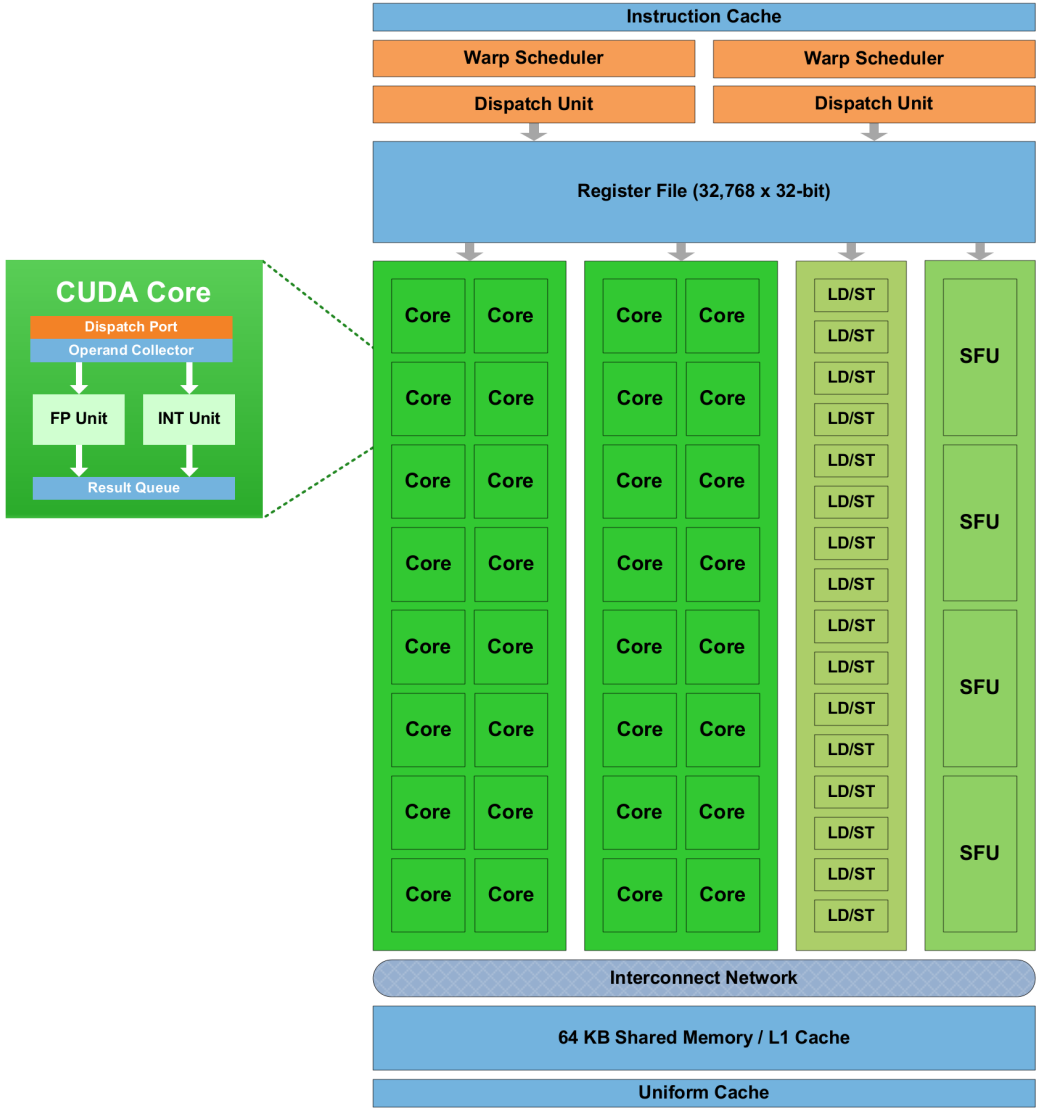


FIGURE: Zdroj Nvidia

CUDA architektuře II.

Každý multiprocesor se skládá z:

- 32 výpočetních jader
- 32k 32-bitových registrů
- 64 kB SRAM
- provede jednu FMA operaci na jeden takt u float a dva takty u double
- 16 jednotek pro načítání a zápis dat
  - umí adresovat paměť dvourozměrně
  - umí převádět data mezi různými typy např. int n float apod.
- čtyři speciální jednotky se používají pro výpočet složitých funkcí jako `sin`, `cos`, `tan`, `exp`

Od hardwarové architektury se odvíjí hierarchická struktura vláken:

Vlákna v CUDA

- **CUDA host** je CPU a operační paměť
- **CUDA device** je zařízení pro paralelní zpracování až stovek tisíc nezávislých vláken - threads
- **CUDA thread** je velmi jednoduchá struktura - rychle se vytváří a rychle se přepíná při zpracování
- komunikace mezi výpočetními jednotkami je hlavní problém v paralelním zpracování dat
- nemůžeme očekávat, že budeme schopni efektivně synchronizovat tisíce vláken
- CUDA architektura zavádí menší skupiny vláken zvané bloky - **blocks**

Bloky a gridy

- jeden blok je zpracován na jednom multiprocesoru
- vlákna v jednom bloku sdílejí velmi rychlou paměť s krátkou latencí
- vlákna v jednom bloku mohou být **synchronizována**
- v jednom bloku může být až 1024 vláken
  - multiprocesor přepíná mezi jednotlivými vlákny
  - tím zakrývá latence pomalé globální paměti
  - zpracovává vždy ta vlákna, která mají načtena potřebná data, ostatní načítají

Bloky vláken jsou seskupeny do gridu - **grid**.

Model zpracování vláken

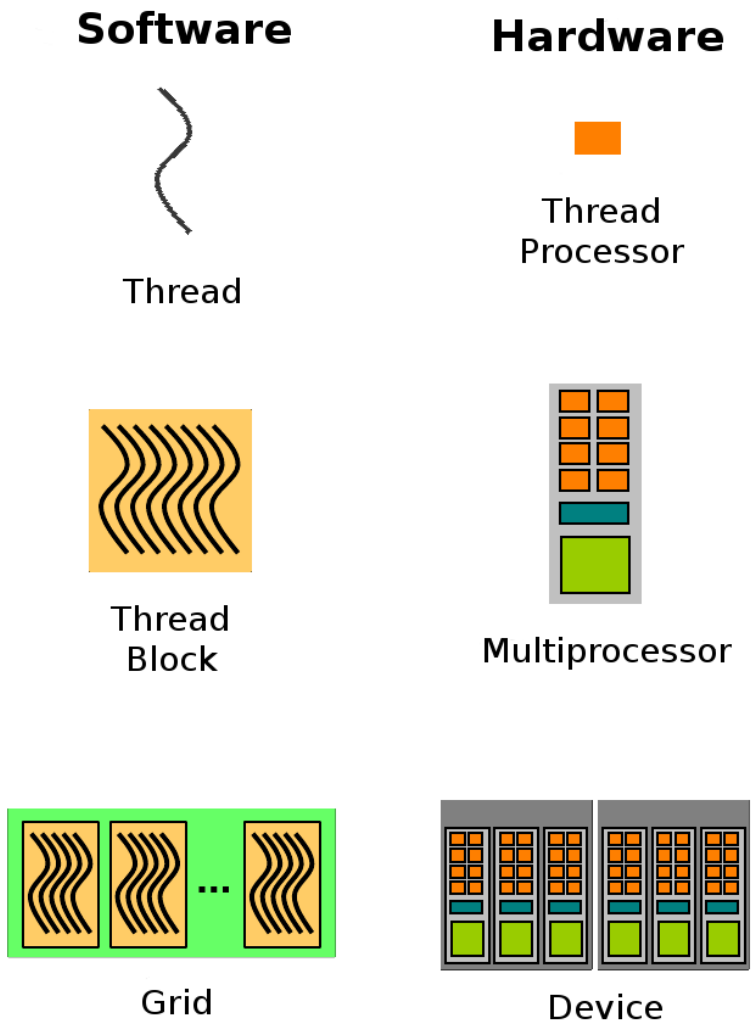


Figure: Zdroj Nvidia: Getting Started with CUDA

PAMĚŤOVÝ MODEL

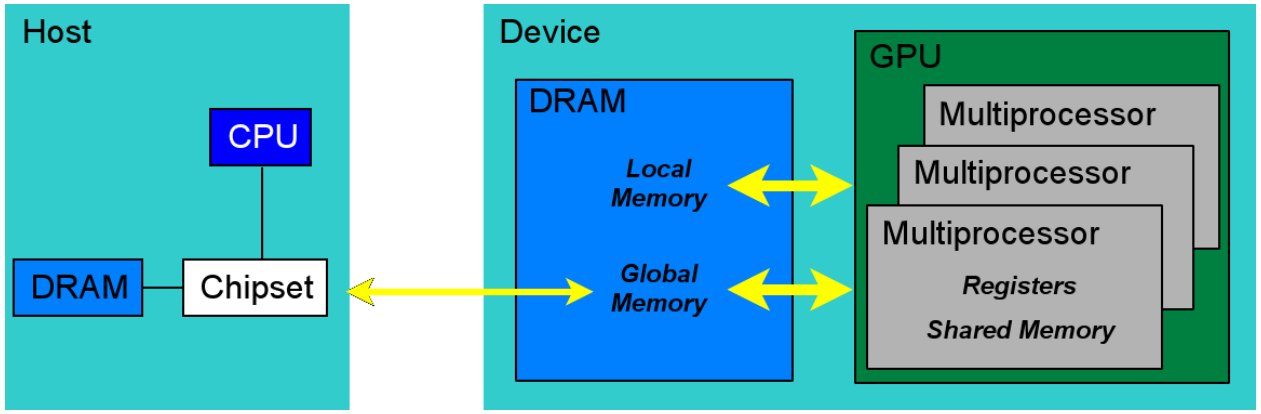


FIGURE: Zdroj Nvidia: Getting Started with CUDA

PAMĚŤOVÁ HIERARCHIE

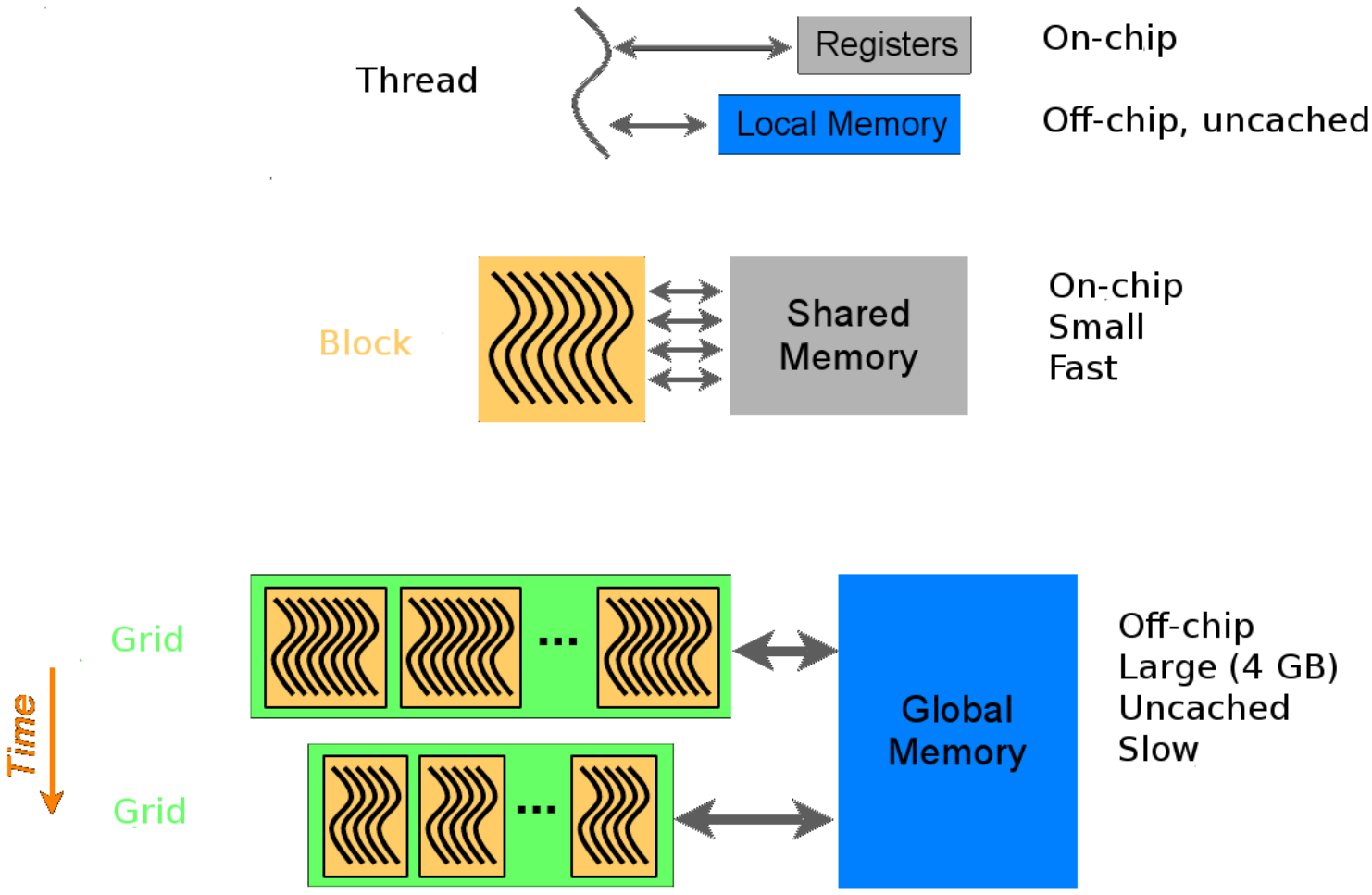


FIGURE: Zdroj Nvidia: Getting Started with CUDA

PROGRAMOVÁNÍ V CUDA I.

- programování v CUDA spočívá v psaní kernelů - **kernels**
  - kód zpracovaný jedním vláknem
- kernely nepodporují rekurzi
- podporují větvení kódu, ale to může snižovat efektivitu
- nemohou vracet žádný výsledek
- jejich parametry nemohou být reference
- podporují šablony C++
- **od CUDA 2.0 podporují funkci** `printf` **!!!**

Následující kód v C

```
int main()
{
    float A[ N ], B[ N ], C[ N ];
    ...
    for( int i = 0; i <= N-1, i ++ )
            C[ i ] = A[ i ] + B[ i ];
}
```

PROGRAMOVÁNÍ V CUDA II.

lze v CUDA zapsat jako

```
__global__ void vecAdd( float* A, float* B, float*
C )
{
    int i = threadIdx.x;
    if( i = < N )
       C[ i ] = A[ i ] + B[ i ];
}
int main()
{
    // allocate A, B, C on the CUDA device
    ...
    vecAdd<<< 1,N >>>( A, B, C );
}
```

Alokování paměti na CUDA zařízení

```
// Allocate input vectors h_A and h_B in host memory
float* h_A = malloc(size);
float* h_B = malloc(size);

// Allocate vectors in device memory
float* d_A;
cudaMalloc((void**)&d_A, size);
float* d_B;
cudaMalloc((void**)&d_B, size);
float* d_C;
cudaMalloc((void**)&d_C, size);

// Copy vectors from host memory to device memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Invoke kernel
VecAdd<<< 1, N >>>(d_A, d_B, d_C);

// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Kód uložíme v `cuda-example.cu` a přeložíme pomocí `nvcc`.

Vývoj efektivního kódu

Pro získání efektivního kódu je nutné dodržet následující pravidla:

- redukovat přenos dat mezi CPU (CUDA host) a GPU (CUDA device)
- optimalizovat přístup do globální paměti
- omezit divergentní vlákna
- zvolit správnou velikost bloků

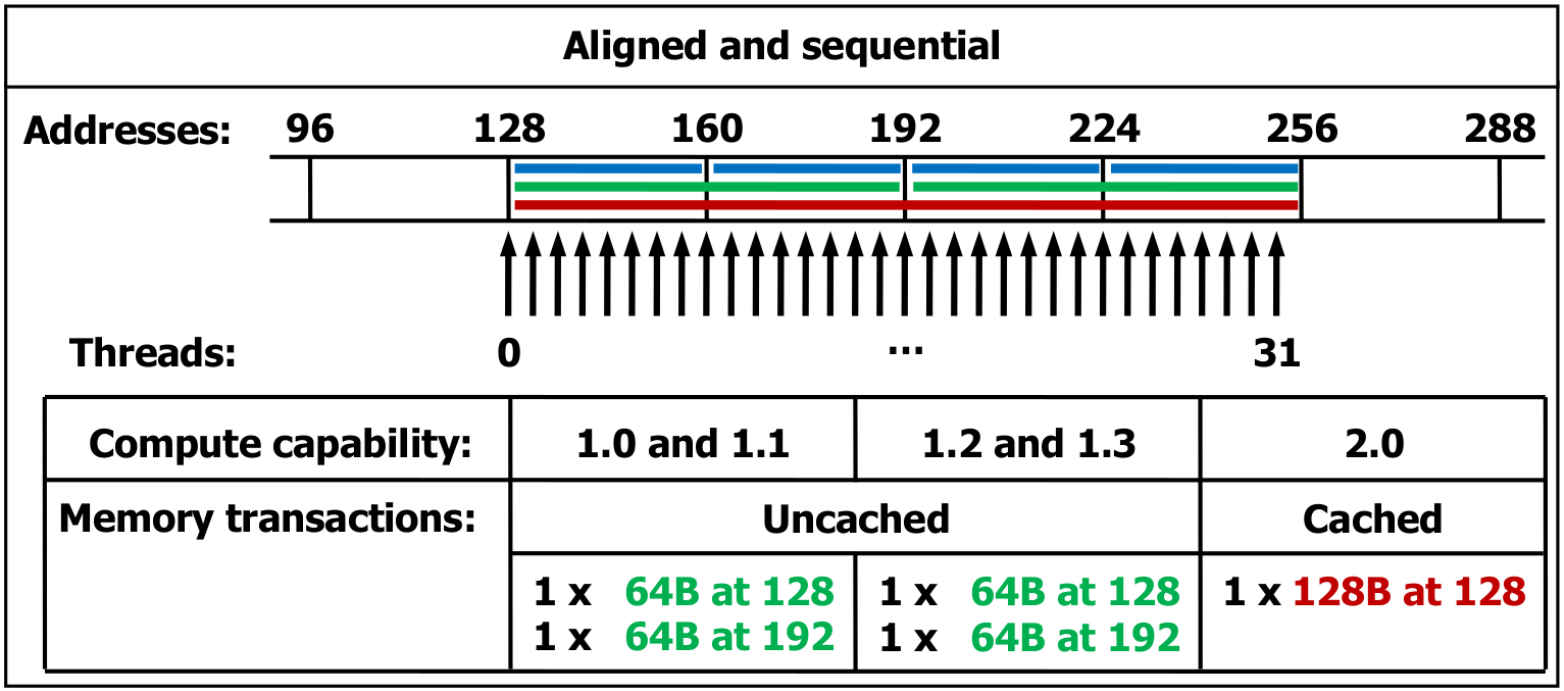
Komunikace mezi CPU a GPU

- komunikace přes PCI Express je velmi pomalá - méně než 5 GB/s
- je nutné tuto komunikaci minimalizovat
  - ideálně provést jen na začátku a na konci výpočtu
- GPU se nevyplatí pro úlohy s nízkou aritmetickou intenzitou
- z tohoto pohledu mohou mít výhodu on-board GPU, které sdílí operační paměť
- pokud je nutné provádět často komunikaci mezi CPU a GPU pak je dobré jí provádět formou pipeliningu
- je možné provádět najednou
  - výpočet na GPU
  - výpočet na CPU
  - kopírování dat z CPU do GPU
  - kopírování dat z GPU na CPU

Sloučené přístupy do pamětí

- většinu přístupů GPU do globální paměti tvoří načítání textur
- GPU je **silně optimalizováno pro sekvenční přístup do globální paměti**
- programátor by se měl vyhnout náhodným přístupům do globální paměti
- ideální postup je:
  - načíst data do sdílené paměti multiprocesoru
  - provést výpočty
  - zapsat výsledek do globální paměti
- sloučený přístup - **coalesced memory access** - může velmi výrazně snížit (až 32x) počet paměťových transakcí

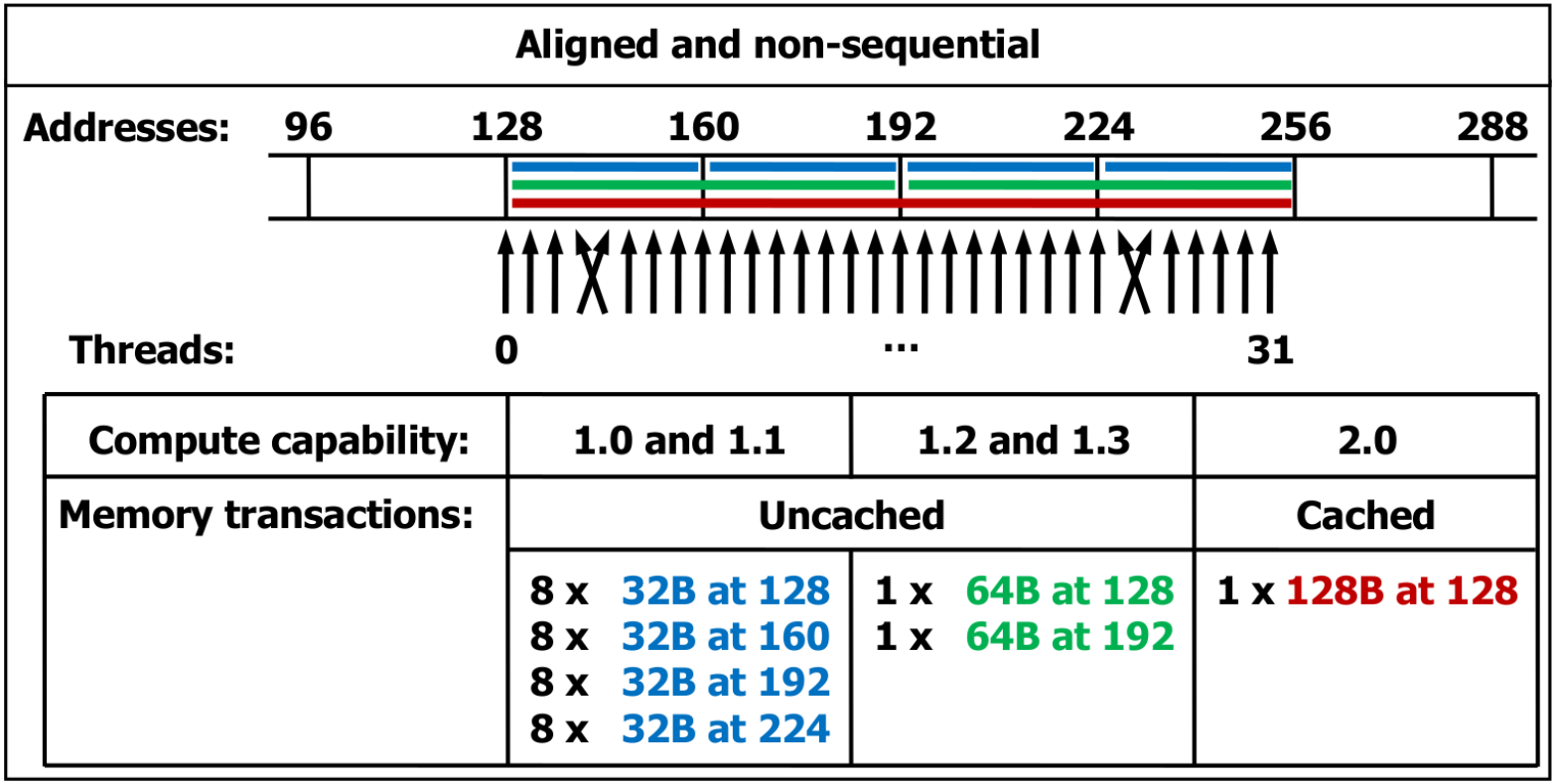
SLOUČENÉ PŘÍSTUPY DO PAMĚTI



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

FIGURE: Zdroj Nvidia: Nvidia CUDA programming guide

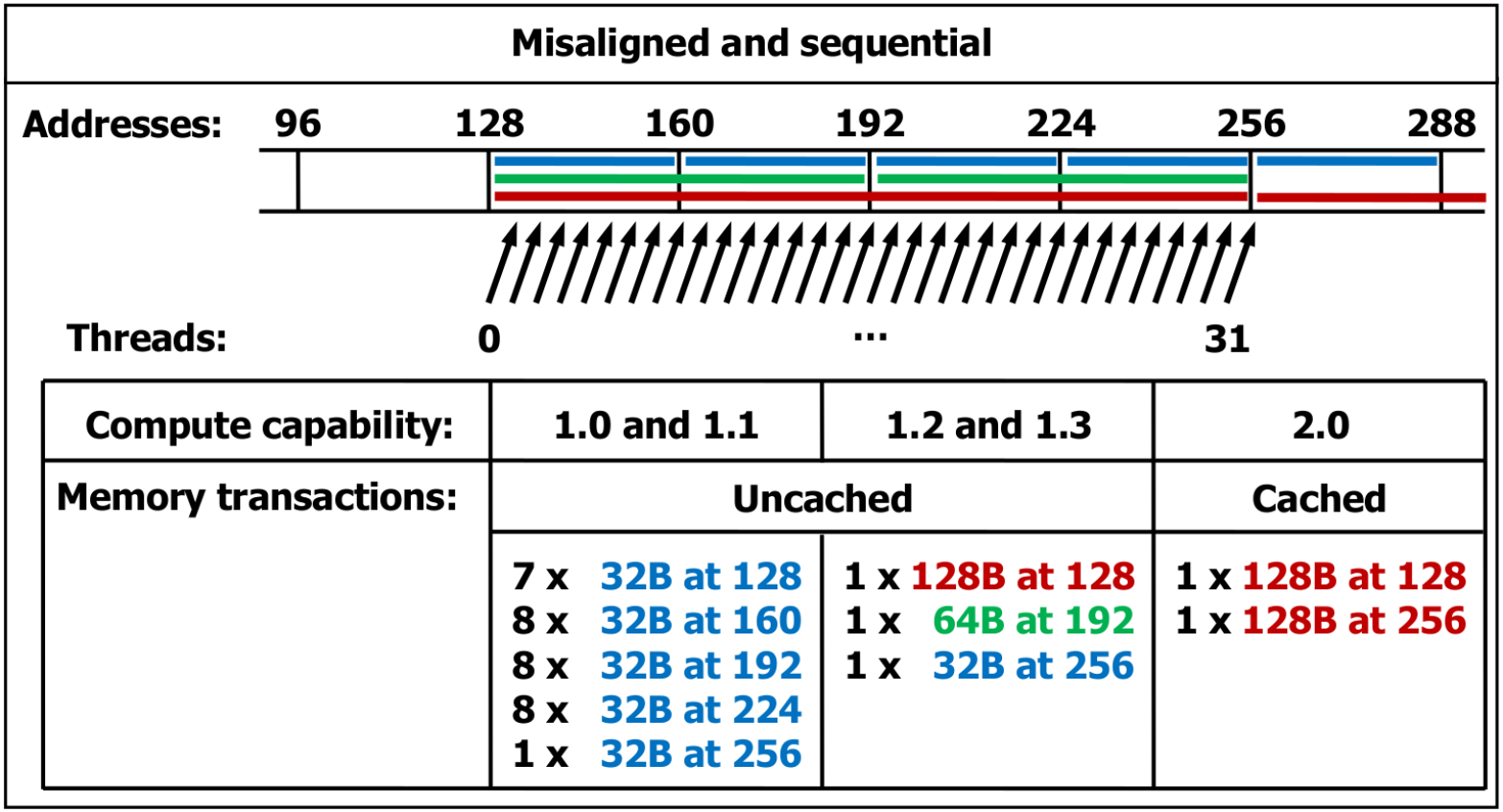
SLOUČENÉ PŘÍSTUPY DO PAMĚTI



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

FIGURE: Zdroj Nvidia: Nvidia CUDA programming guide

SLOUČENÉ PŘÍSTUPY DO PAMĚTI



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

FIGURE: Zdroj Nvidia: Nvidia CUDA programming guide

Paměť textur

Není-li možné dosáhnout sloučených přístupů do globální paměti, lze využít kešovanou paměť textur.

- nejprve je nutné ji bindovat s texturou pomocí `cudaBindTexture`
- v daném kernelu do této paměti nelze zapisovat
- textura může být 1 nebo 2 dimenzionální
- s každým načteným prvkem se načítají i okolní prvky

Nelze-li efektivně využít paměť pro textury, lze již spoléhat jen na keš.

Keš

Architektura Fermi zavádí plně funkční L1 a L2 keše.

- L1 keš se nachází na každém multiprocesoru
  - lze nastavit, jaká část ze 64kB SRAM paměti bude určeno pro keš pomocí funkce:
  - `cudaFuncSetCacheConfig( MyKernel, cudaFuncCachePreferShared )`
    - `cudaFuncCachePreferShared` - shared memory is 48 KB
    - `cudaFuncCachePreferL1` - shared memory is 16 KB
    - `cudaFuncCachePreferNone` - no preference
- L2 keš je společná pro všechny multiprocesory a má velikost 768kB

Sdílená paměť multiprocesoru

- sdílená paměť multiprocesoru je rozdělena na 16 paměťových bank
- data se ukládají do jednotlivých bank vždy po 4 bajtech
- je potřeba se vyhnout situaci, kdy dvě vlákna ze skupiny 16 čtou z různých adres v jedné bance
- nevadí, když čte více vláken ze stejné adresy, použije se broadcast

Divergentní vlákna

- CUDA device umí zpracovávat současně různé kernely, ale jen na různých multiprocesorech
- Nvidia tuto architekturu nazývá SIMT = **Single Instruction, Multiple Threads**
- v rámci jednoho multiprocesoru jde ale o SIMD architekturu, tj. všechny jednotky provádějí stejný kód
- **warp** je skupina 32 vláken zpracovávaných současně
  - vlákna ve warpu jsou tedy implicitně synchronizovaná
  - všechna by měla zpracovávat stejný kód
- warp se dělí na dvě poloviny - **halfwarps**
  - to je důležité z pohledu přístupu do sdílené paměti multiprocesoru
  - multiprocesor má jen 16 jednotek pro načítání/zápis dat, proto je celý warp obsloužen vždy ve dvou krocích

ZPRACOVÁNÍ BLOKŮ VLÁKEN NA MULTIPROCESORU

- na mutliprocesoru většinou běží více bloků vláken
- scheduler mezi nimi přepíná a spouští vždy ty bloky vláken, které mají načteny potřebná data
  - tím se zakrývají velké latence globální paměti
- k tomu je ale potřeba, aby jeden blok nevyčerpal všechny registry a sdílenou paměť
  - pokud není dostatek registrů, ukládají se proměnné do *local memory* - to je pomalé
  - je potřeba dobře zvolit velikost bloku - násobek 32
  - minimalizovat počet proměnných a množství sdílené paměti použité jedním blokem
  - minimalizovat velikost kódu kernelu
- efektivnost obsazení multiprocesoru udává parametr zvaný **occupancy** (maximum je 1.0)
- za účelem optimalizace lze použít
  - CUDA occupancy calculator [1]
  - CUDA profiler
  - výpisy `nvcc -ptxas-options=-v`

[1] http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Budoucnost GPU

- GPU je pro mnoho typů úloh mnohem lepší architektura než CPU

Ale

- stále je některými lidmi považováno za herní zařízení
- i s pomocí CUDA je vývoj algoritmů pomalý a vyžaduje detailní znalosti
  - zatím neexistují knihovny běžných algoritmů pro GPU
- ~~slabá podpora pro dvojitou přesnost~~
- omezená paměť na 6 GB
  - ~~málo zkušeností s GPU klastry~~
- GPU se stále vyvíjí velmi rychle a je náročné sledovat všechny změny
- ~~možná fůze s CPU~~

CUDA 3 and Fermi

- podpora keší na multiprocesoru - 64Kb
- podpora ECC RAM
- "úplná podpora" C++
- `printf` jako funkce kernelu usnadňuje ladění

CUDA 4

- lepší podpora pro počítání na více GPU
  - lze obsluhovat více GPU z jednoho vlákna
- unifikovaný adresový prostor

Zbytek světa

- ATI/AMD - Radeon GPUs
- podporuje OpenCL (Nvidia také)
  - OpenCL nepodporuje C++ and Fortran
- AMD Fusion - GPU implementované na základní desce a sdílející paměť s CPU
  - odstraňuje nutnost přenosu dat CPU $\leftrightarrow$ GPU
  - ale pracuje s běžnou a pomalou DRAM
- Intel
  - nové CPU od Intelu obsahují GPU také
  - Larabee architektura

Future of CUDA?

- Nvidia má vedoucí postavení v GPGPU díky CUDA
- CUDA nepodporuje GPU od AMD
- CUDA má brzy podporovat vícejádrové systémy = běh kernelů n x86
- Nvidia nemá vlastní CPU $\Rightarrow$ investuje do ARM architektury Nvidia Tegra
  - Microsoft oznámil podporu Windows 8 na ARM CPUs
  - AMD ohlásilo vývoj ARM CPU
  - Nvidia plánuje vytvořit ARM CPU pro HPC