Hardware pro počítačovou grafiku
NPGR019
Úvod do architektury CUDA

Jan Horáček

http://cgg.mff.cuni.cz/
MFF UK Praha

2012

Obsah

Historie

- vývoj křemíkových čipů rychlý - žádáno trhem
- kolem roku **2003** - přestala se zvyšovat **frekvence** CPU (spotřeba energie a zbytkové teplo u existujících technologií přestaly být únosné)
- současný trend - zvyšování **počtu jader** - vhodné pro **paralelizovatelné** úlohy
- tradičně většina programů a algoritmů **sekvenční**
- paralelní programování dávno známé v *high-performance computing* komunitě - nyní se přenáší i do spotřebitelské sféry

Historie GPU

- vývoj grafického HW:
  1. specializovaný jednoúčelový
  2. konfigurovatelný
  3. programovatelný
- dnes dokáže spustit téměř **libovolný** algoritmus (až na limity délky kódu a paměti)
- **efektivní** pouze pro specifickou skupinu algoritmů

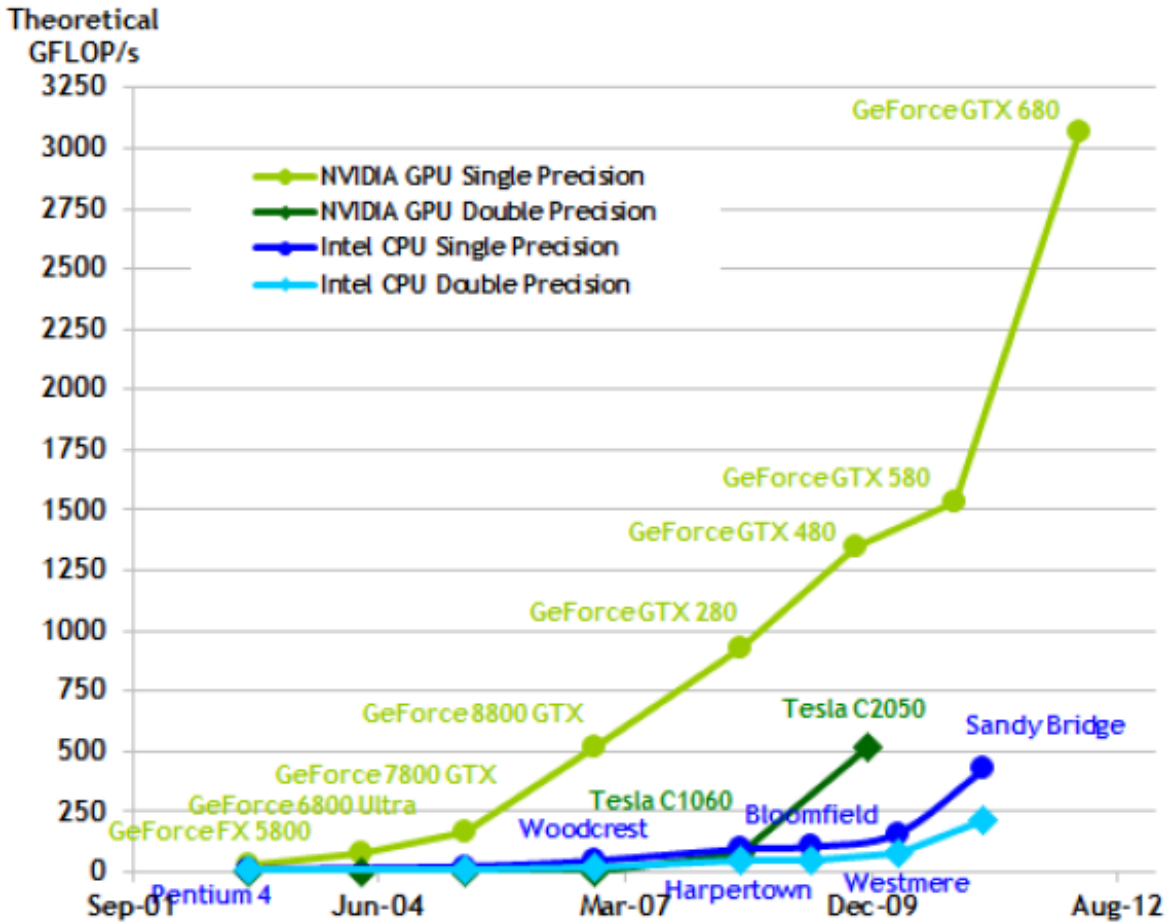
Technologie vícejádrového zpracování 1

1. Multicore
   - soustředí se na běh **sekvenčních** programů
   - dvouo- a vícejádrové procesory (řádově do několika málo desítek)
   - **plná** instrukční sada x86
   - příklad: Core i7 - čtyři jádra s *out-of-order execution* a technologií *hyperthreading*

Technologie vícejádrového zpracování 2

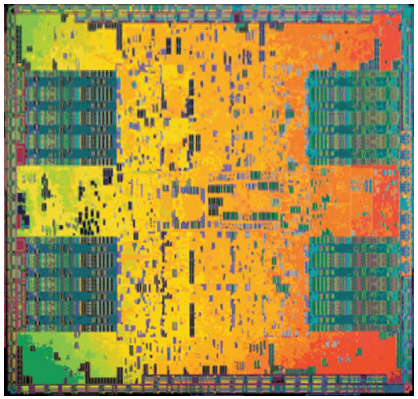
1. Many-core
   - soustředí se na běh **paralelních** programů
   - grafické akcelerátory $\rightarrow$ programovatelné GPU
   - velmi vysoký **hrubý** výkon, v roce 2008 dokonce 1:10 vs CPU
   - příklad: nVidia GeForce GTX 280 - 240 jader, každé jádro *heavily multithreaded, in-order, single-instruction issue processor*, sdílí kontrolu a instrukční cache se 7 dalšími

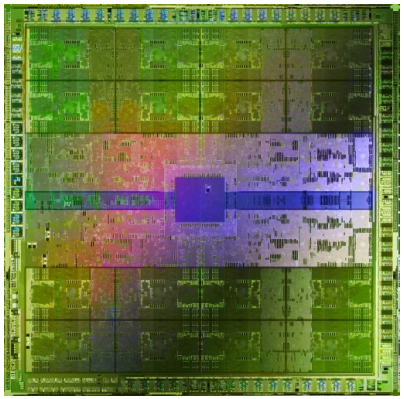
GPU vs CPU porovnání rychlosti

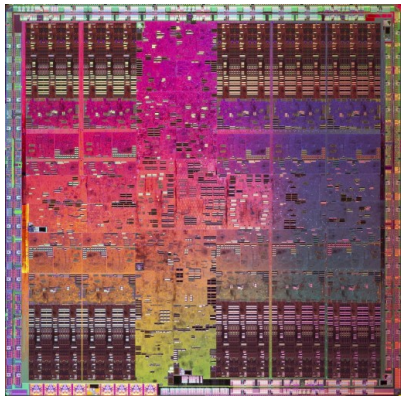


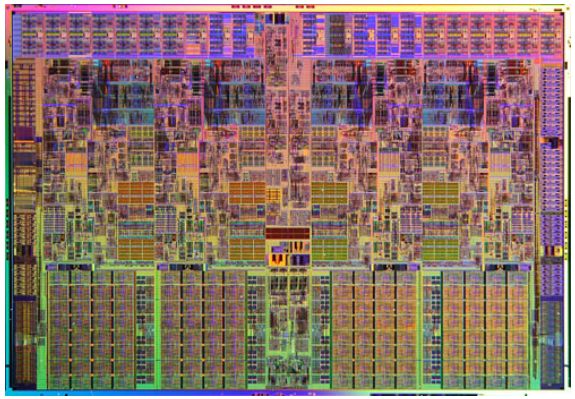
zdroj: nVidia CUDA Programming guide

GPU vs CPU porovnání čipu

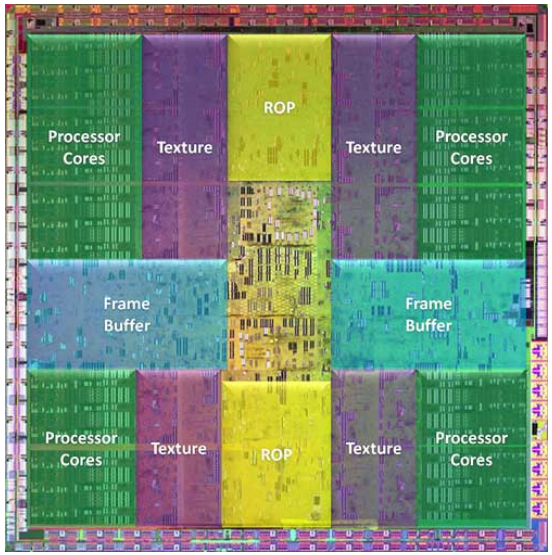




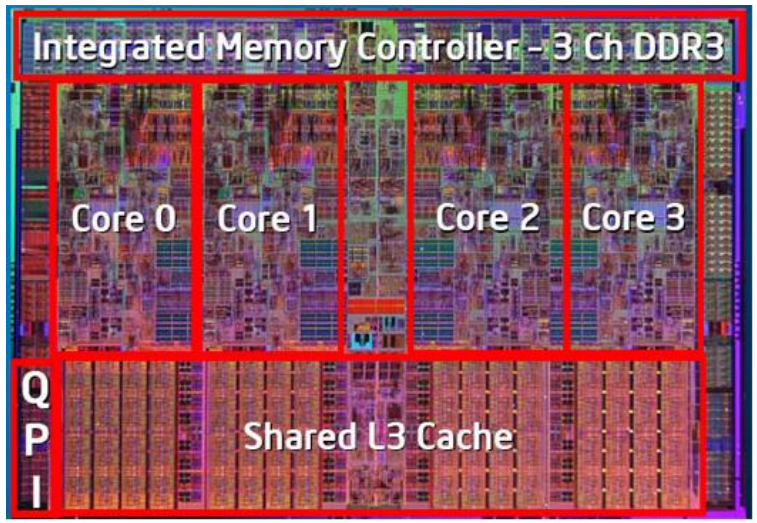




GPU vs CPU porovnání čipu 2

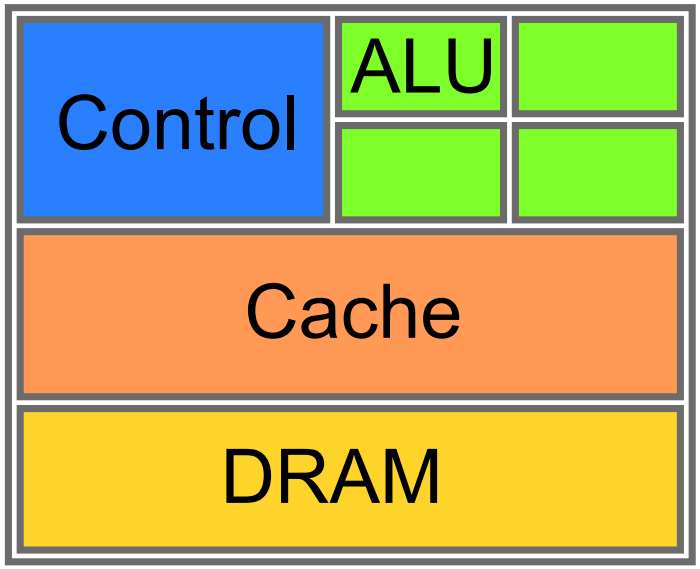


GPU - GT200



CPU - i7

GPU vs CPU porovnání čipu - přehledněji



CPU

GPU

GPU vs CPU

- **CPU**
  - velká plocha pro **cache**
  - preferuje **nižší latenci**
- **GPU**
  - rasterizační algoritmy mají tradičně **koherentní přístup** do paměti → skrývá latenci
  - pixel shadery jsou navíc limitovány výpočetním výkonem → velká plocha pro **operace v plovoucí řádové čárce**
  - preferuje **přenosovou rychlost**
  - nepotřebuje (tolik) cache

Obsah

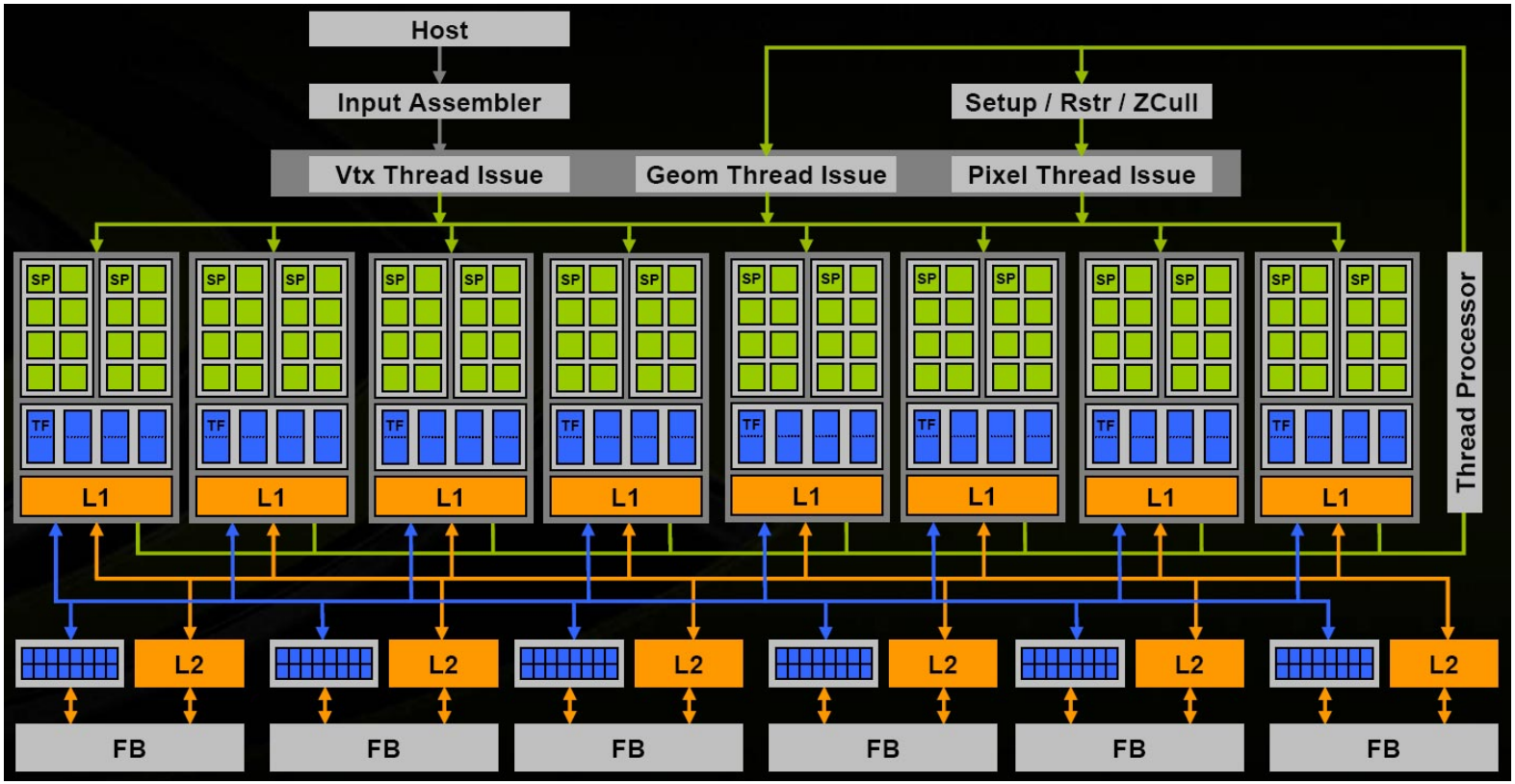
Proč CUDA

- dříve: **GPGPU** - výpočet probíhal v shaderech, data v texturách
- programátoři potřebují jednodušší přístup k prostředkům GPU
- 2007: architektura nVidia Tesla (G80 - **GeForce 8800**)
  - obecnější model paralelního programování
  - hierarchie paralelních vláken
  - bariérová synchronizace
  - atomické operace
- **CUDA** = Compute Unified Device Architecture
- **C for CUDA** = jazyk, kterým se dá programovat architektura CUDA
- efektivní programování ovšem stále potřebuje dobrou znalost HW

Architektura

- CUDA GPU organizováno do skupiny *highly threaded* **streaming multiprocessors**
- několik SM tvoří blok (počet SM v bloku různý v každé generaci)
- každý SM obsahuje několik **streaming processors**
  - G80: 1 SM = 8 SP
  - GF100: 1 SM = 32 SP
- SP v rámci jednoho SM **sdílí** kontrolní obvody a instrukční cache
- technologie provádění kódu: **SIMT** (Single Instruction Multiple Threads)

Architektura G80



GeForce 8800

Architektura GF100

GeForce 480

Compute capabilities 1

	Compute Capability				
Feature Support (Unlisted features are supported for all compute capabilities)	**1.0**	**1.1**	**1.2**	**1.3**	**2.x 3.0**
Atomic functions operating on 32-bit integer values in global memory (Section B.11)	No	Yes			
atomicExch() operating on 32-bit floating point values in global memory (Section B.11.1.3)					
Atomic functions operating on 32-bit integer values in shared memory (Section B.11)	No		Yes		
atomicExch() operating on 32-bit floating point values in shared memory (Section B.11.1.3)					
Atomic functions operating on 64-bit integer values in global memory (Section B.11)					
Warp vote functions (Section B.12)					
Double-precision floating-point numbers	No			Yes	
Atomic functions operating on 64-bit integer values in shared memory (Section B.11)	No				Yes
Atomic addition operating on 32-bit floating point values in global and shared memory (Section B.11.1.1)					
__ballot() (Section B.12)					
__threadfence_system() (Section B.5)					
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)					
Surface functions (Section B.9)					
3D grid of thread blocks					

Compute capabilities 2

	Compute Capability					
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**
Maximum dimensionality of grid of thread blocks	2				3	
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1
Maximum y- or z-dimension of a grid of thread blocks	65535					
Maximum dimensionality of thread block	3					
Maximum x- or y-dimension of a block	512				1024	
Maximum z-dimension of a block	64					
Maximum number of threads per block	512				1024	
Warp size	32					
Maximum number of resident blocks per multiprocessor	8					16
Maximum number of resident warps per multiprocessor	24		32		48	64
Maximum number of resident threads per multiprocessor	768		1024		1536	2048
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K
Maximum amount of						

Zbytek v nVidia CUDA Programming Guide

Datový paralelismus

- mnoho aplikací modelující reálný svět pracuje s **velkým** množstvím dat
- datový paralelismus = na datech může být prováděno mnoho operací **souběžně**
- například: násobení matic

Struktura programu

- jedna nebo více fází
- mohou být puštěny na hostitelském CPU nebo na GPU
- zdrojový CUDA kód může obsahovat **obě** části
- **nvcc** (nVidia C Compiler) tyto části odděluje během kompilace
- podle generace GPU podpora od **ANSI C** až po *téměř* úplnou podporu **C++**
- kód pro GPU nazýván **kernel**
- kernel typicky spouští řádově tisíce vláken
- na rozdíl od CPU vláken jsou řádově *lehčí*
  - vytvoření i přepínání během několika cyklů

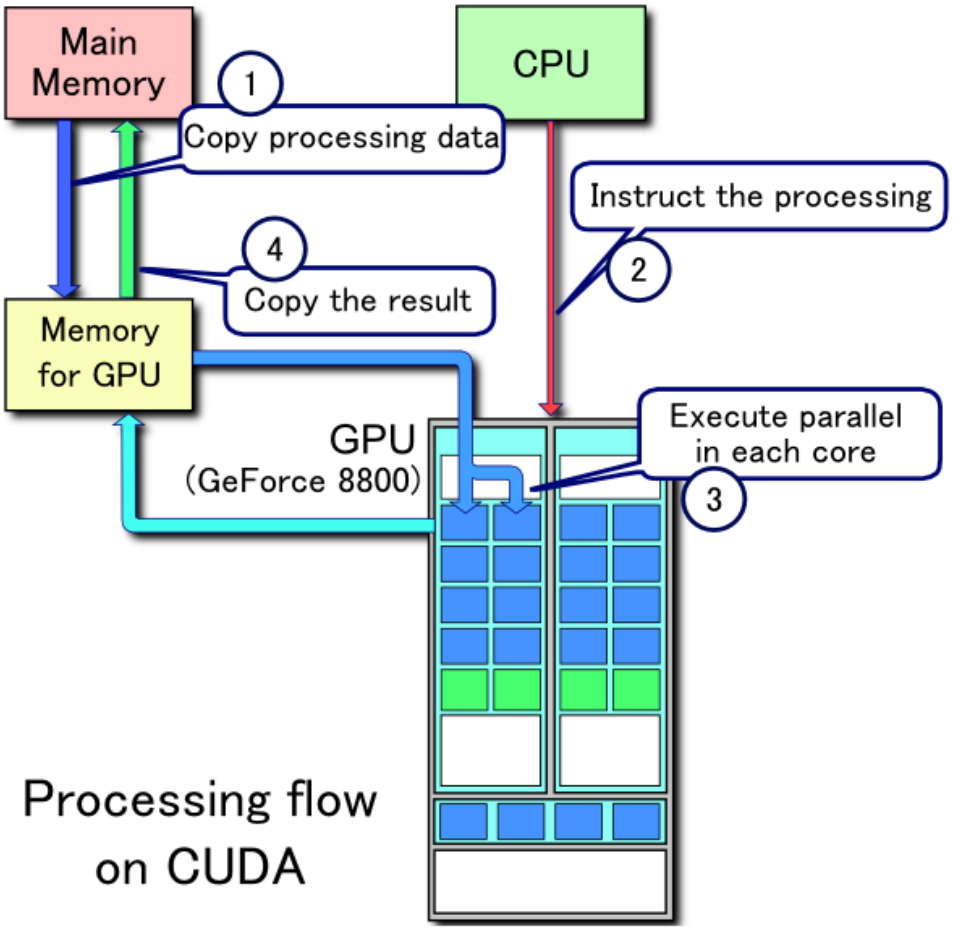
Struktura programu 2

- klíčová slova `__global__`, `__host__`, `__device__` určují, kde a odkud může být kód spuštěn

Klíčové slovo	**Běží na**	**Spustitelné z**
`__global__ float KernelFunc()`	GPU	host
`__device__ float DeviceFunc()`	GPU	GPU
`__host__ float HostFunc()`	host	host

- `__device__` znamená, že fce smí být spuštěna pouze z kernelu nebo jiné funkce na GPU
- `__host__` a `__device__` najednou znamená, že se generují 2 verze, jedna pro CPU a druhá pro GPU

Běh programu



zdroj: Wikipedia.org

Princip kernelů

- spusť jedno vlákno

CPU kód

```
for(int y = 0; y < height; y++)
  for(int x = 0; x < width; x++)
    doSomething(x,y);
```

Princip kernelů

- spusť jedno vlákno

CPU kód

```
for(int y = 0; y < height; y++)
  for(int x = 0; x < width; x++)
    doSomething(x,y);
```

- spusť $x \cdot y$ vláken

GPU kernel

1. zjisti x, y ze svého *id*
2. `doSomething(x,y);`

Ukázka kernelu

```
// Definice kernelu
__global__ void VecAdd(float *A, float *B, float *C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Spuštění kernelu s N vlákny
    VecAdd<<<1, N>>>(A,B,C);
}
```

Hierarchie vláken

- kernel je spuštěn jako **grid** paralelních vláken
- vlákna v gridu jsou ve dvouúrovňové hierarchii
  - grid obsahuje 2D strukturu **bloků**
  - blok obsahuje 3D strukturu vláken
- všechny bloky v gridu stejné velikosti
- blok obsahuje max. 512 (1024) vláken
- velikosti gridu, bloků a počet vláken je zadán při spouštění kernelu

Hierarchie vláken 2

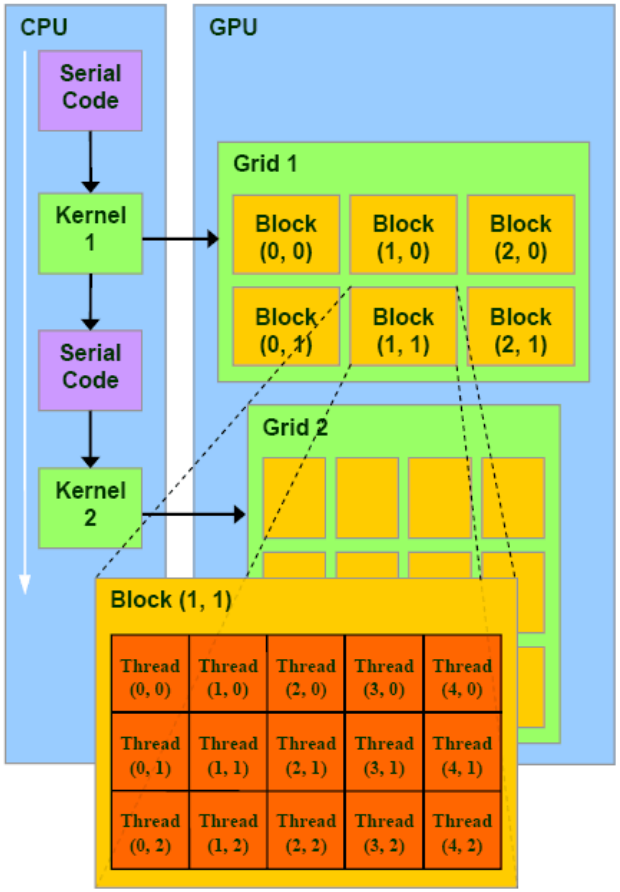


schéma hierarchie vláken

Příklad parametrů kernelu

```
// Nastav konfiguraci spuštění
dim3 dimBlock(ThreadsX, ThreadsY)
dim3 dimGrid(BlocksX,BlocksY)

// Spusť kernel
MujKernel<<<dimGrid, dimBlock>>>
      (param1, param2);
```

ID vlákna

- CUDA definuje několik rozšíření ANSI C o vestavěné proměnné pro identifikaci vláken

Proměnná	**význam**
`threadIdx.x, threadIdx.y, threadIdx.z`	index vlákna v bloku
`blockIdx.x, blockIdx.y`	index bloku v gridu
`blockDim.x, blockDim.y, blockDim.z`	velikost bloku
`gridDim.x, gridDim.y`	velikost gridu
`warpSize`	velikost warpu

Synchronizace

- mezivláknová komunikace pomocí **bariérové synchronizace**
- funkce `__syncthreads()` **garantuje**, že všechny vlákna **v bloku** provedly každou instrukci do tohoto místa
- v případě *if-then-else* musí všechna vlákna v bloku jít stejnou cestou
- komunikace mezi bloky není možná $\rightarrow$ bloky v libovolném pořadí $\rightarrow$ škálovatelnost
- synchronizace mezi bloky se řeší rozdělením práce do více kernelů

Přiřazování vláken

- nemá vliv pro funkčnost, ale je dobré to znát pro efektivnost
- každý SM může mít na sobě až 8 bloků (ovlivněno volnými zdroji)
- každý blok rozdělen na jednotky o 32 vláknech zvané **warp**
- každý warp obsahuje 32 vláken s po sobě jdoucími *threadIdx*
- maximálně 24(48) warpů na SM
- maximálně 768(1536) vláken na SM
- velké množství a rychlé přepínání vláken skrývá operace s dlouhou latencí (*latency-hiding*)

Paměť

- pro kernely několik typů paměti, drastické rozdíly v rychlosti
  - per-thread **registry**
  - per-thread **lokální** paměť
  - per-block **sdílená** paměť
  - per-grid **globální** paměť
  - per-grid **konstantní** paměť (pouze pro čtení)
- globální paměť typicky DRAM, přístup řádově stovky cyklů
- registry a lokální paměť jsou na čipu, přístup řádově jednotky cyklů
- aplikace (host) může přesouvat data do/z globální a konstantní paměti
- sousední vlákna mohou *sdílet* přístup do globální paměti (**memory coalescing**)

Typové kvalifikátory proměnných

Deklarace	**Paměť**	**Viditelnost**	**Životnost**
autom. proměnné mimo polí	registry	vlákno	kernel
autom. pole	lokální	vlákno	kernel
`__shared__`	sdílená	blok	kernel
`__device__`	globální	grid	aplikace
`__constant__`	konstantní	grid	aplikace

- max. velikost sdílené paměti: 16KB (48KB)
- max. velikost konstantní paměti: 64KB
- max. počet 32-bit registrů na SM: 8K(32K)
- lokální paměť je **pomalá** jako globální

Principy programování

- příklad běhu programu
  1. každé vlákno načte část dat z globální do sdílené paměti
  2. `__syncthreads()`
  3. hlavní výpočet
  4. uložení výsledku
  5. příp. *přičtení* další části dat a pokračování výpočtu
- dát si pozor na větvení *if-then-else*
  - větvení nezpůsobuje výkonový propad, pokud všechna vlákna na SM jdou stejnou cestou
  - pokud se ovšem větví v rámci zpracování jednoho SM, musí se spočítat obě větve

Obsah

OpenCL

- alternativa k *C for CUDA*
- základní myšlenka převzaná z *C for CUDA*, v některých oblastech téměř 1:1 ekvivalence
- programovací model pro spouštění masivně paralelních úloh na **CPU, GPU, Cell,** ...
- podrobnější výběr výpočetního zařízení
- **širší spektrum** schopností zařízení
- daný algoritmus **nemusí** běžet na každém HW

OpenCL koncepty

OpenCL	**CUDA ekvivalent**
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

OpenCL vlákna

OpenCL	CUDA ekvivalent
`get_global_id(0)`	`blockIdx.x` · `blockDim.x` + `threadIdx.x`
`get_local_id(0)`	`threadIdx.x`
`get_global_size(0)`	`gridDim.x` · `blockDim.x`
`get_local_size(0)`	`blockDim.x`

OpenCL paměť

OpenCL	**CUDA ekvivalent**
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	local memory

Ukázka OpenCL kernelu

```
// Definice kernelu
__kernel void VecAdd(__global const float *A,
        __global const float *B, __global float *C)
{
   int id = get_global_id(0);
   C[id] = A[id] + B[id];
}
```

Obsah

(Velmi lehký) úvod do PhysX

- HW akcelerované rozhraní pro simulaci fyzikálních dějů ve virtuálních světech
- výborné pro **herní** simulace
  - pro spolehlivé stabilní výpočty nízká přesnost, ovšem i tak občas použitelné
- nejčastěji využívané pro kolize částic a jednoduchých těles

Historie

- Švýcarská firma *NovodeX AG* začala vyvíjet platformu pro výpočet fyziky jako konkurenci pro *Havok*
- 2004 - akvizice firmou **Ageia**, NovodeX Physics SDK se stalo základem pro **PhysX 2.x** SDK
  - dodnes mnoho původních lidí z NovodeXu na vývoji PhysX
- Ageia vyvinula HW platformu pro akceleraci fyziky
  - **PhysX PPU** (Physics Processing Unit)
- 2008 - akvizice firmou **nVidia**
  - implementace na architektuře CUDA
  - postupné odstranění podpory Ageia PPU
  - podpora ve stovkách her
  - silný marketing - cílené odstranění podpory na cizích GPU

Současnost

- CPU implementace
  - běží tedy "všude"
  - ve hrách často simulace několika málo hlavních objektů, aby to zvládalo i CPU
  - při použití GPU přidání částicového "nepořádku", lepší interakce s vodou, více dynamických objektů (bez závislosti na gameplay)
- pro GPU akceleraci nutných alespoň 32 CUDA jader a 256MB paměti
- dostupné nejen na PC, ale i na PS3, Xbox 360, Wii
- fyzikální systémy všeobecně: aktuálně snahy o dobrou implementaci na mobilní zařízení
  - iOS
  - Android

PhysX 2.x

- rigid body dynamics (mechanika tuhých těles)
  - kolizní primitiva - kapsule, koule, kvádr, rovina, výšková mapa, konvexní těleso)
  - různé typy kloubů
  - ragdoll, materiály, tření, ...
- deformovatelná tělesa
  - simulace látky (textilie)
  - trhání, kolize sama se sebou, ...
- částice a kapaliny
  - jedno- a oboustranná interakce s látkami
- dynamika vozidel
- objemová silová pole

PhysX 3.x

- vylepšený systém kloubů
- neuniformní škálování těles
- stabilní "depenetrace"
- nový solver na simulaci látek
- mnoho zlepšení výkonu
  - vylepšené cachování
  - pro PC a Xbox360 zlepšený multithreading
- nový model simulace vozidla
  - motor, převodovka, pneumatiky, kola, tlumiče, ...
- **změny a vyčištění** API
- ...

Příklad implementace PhysX 2.x

Inicializace

```
// Initialize PhysicsSDK
gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION);
if (!gPhysicsSDK)
   return;

// Set the debug visualization parameters
gPhysicsSDK->setParameter(NX_VISUALIZATION_SCALE, 1);
gPhysicsSDK->setParameter(NX_VISUALIZE_COLLISION_SHAPES, 1);
gPhysicsSDK->setParameter(NX_VISUALIZE_ACTOR_AXES, 1);

// Set scale dependent parameters
NxReal scale = 1.0f;   // scale is meters per PhysX units

gPhysicsSDK->setParameter(NX_SKIN_WIDTH, 0.05*(1/scale));
// ... other parameters initialization
```

Příklad implementace PhysX 2.x

Vytvoření scény

```
// Create a scene
NxSceneDesc sceneDesc;
sceneDesc.gravity = NxVec3(0.0f, -9.81f, 0.0f);
gScene = gPhysicsSDK->createScene(sceneDesc);
if(gScene == NULL)
   return false;

// Set default material
NxMaterial* defaultMaterial = gScene->getMaterialFromIndex(0);
defaultMaterial->setRestitution(0.0f);
defaultMaterial->setStaticFriction(0.5f);
defaultMaterial->setDynamicFriction(0.5f);

// Create ground plane
NxPlaneShapeDesc planeDesc;
NxActorDesc actorDesc;
actorDesc.shapes.pushBack(&planeDesc);
gScene->createActor(actorDesc);
```

Příklad implementace PhysX 2.x

Vytvoření objektu

```
// Create body
NxBodyDesc bodyDesc;
bodyDesc.angularDamping = 0.5f;
if(initialVelocity) bodyDesc.linearVelocity = *initialVelocity;

NxBoxShapeDesc boxDesc;
boxDesc.dimensions = NxVec3((float)size, (float)size, (float)size);

NxActorDesc actorDesc;
actorDesc.shapes.pushBack(&boxDesc);
actorDesc.body = &bodyDesc;
actorDesc.density = 10.0f;
actorDesc.globalPose.t  = pos;
gScene->createActor(actorDesc);
```

Příklad implementace PhysX 2.x

Asynchronní simulace

```
// Start simulation (non blocking)
gScene->simulate(1.0f/60.0f);
```

Čekání na výsledek

```
gScene->flushStream();
gScene->fetchResults(NX_RIGID_BODY_FINISHED, true);
```

Příklad implementace PhysX 2.x

"Vykreslení"

```
int nbActors = gScene->getNbActors();
NxActor** actors = gScene->getActors();
while(nbActors--) {
   NxActor* actor = *actors++;

   // Render actor
   float glMat[16];
   actor->getGlobalPose().getColumnMajor44(glMat);
   ...
}
```

Nástroje vyšší úrovně

- nVidia vytvořila nástroje vyšší úrovně - **APEX**
- 2 části
  - **tvorba** - samostatné programy a pluginy pro modelování
  - **runtime** - pro jednoduchou implementaci do vlastního engine
- APEX **Clothing**
  - simulace kompletního oblečení
  - používané nejen ve hrách, ale i u návrhářů atd.
- APEX **Destruction**
  - tvorba komplikovaných zničitelných objektů
  - např. možnost rozpadnutí zdi na cihly
- APEX **Particles**
  - kompletní částicové efekty, kouř, kapaliny
- APEX **Turbulence**
  - kouř, prach, částice s turbulentním chováním
  - eulerovský solver kapalin na mřížce
- APEX **Vegetation** - ?

Alternativy

- Havok (Havok Inc. + Intel)
- Open Dynamics Engine
- Newton Game Dynamics
- Physics Abstraction Layer
- (AMD . . . ?)
- . . .

Obsah

Literatura

- David B. Kirk, Wen-mei W. Hwu: **Programming massive parallel processors**, Morgan Kaufman, 2010, ISBN:978-0-12-381472-2
- Jason Sanders, Edward Kandrot: **CUDA by Example: An Introduction to General-Purpose GPU Programming**, Addison-Wesley, 2010, ISBN:978-0131387683
- nVidia Corporation: (**CUDA C/OpenCL**) **Programming Guide**
- nVidia Corporation: (**CUDA C/OpenCL**) **Best Practices**
- http://developer.nvidia.com
- http://www.opencl.org
- http://www.ati.com