

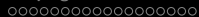
# CUDA nástroje a knihovny

Jiří Matela

podzim 2017

# Rekapitulace

- Proč programovat GPU



# Rekapitulace

- Proč programovat GPU

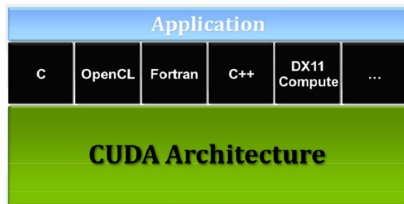
# Rekapitulace

- Proč programovat GPU
- GPU architektura (vs. CPU)

# Rekapitulace

- Proč programovat GPU
- GPU architektura (vs. CPU)
- CUDA (Compute Unified Device Architecture)

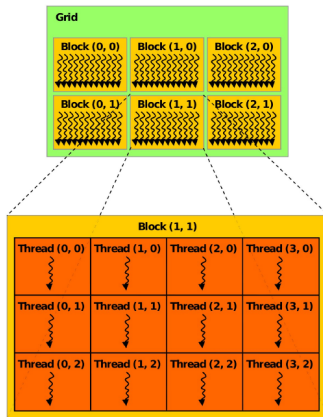
## Architektura CUDA



# Rekapitulace

- Proč programovat GPU
- GPU architektura (vs. CPU)
- CUDA (Compute Unified Device Architecture)

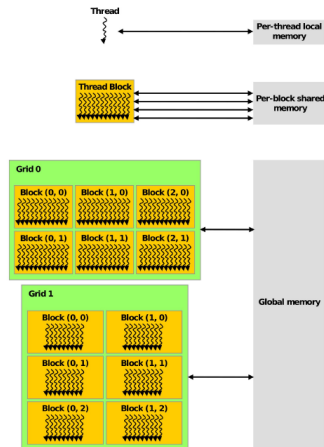
## Hierarchie vláken



# Rekapitulace

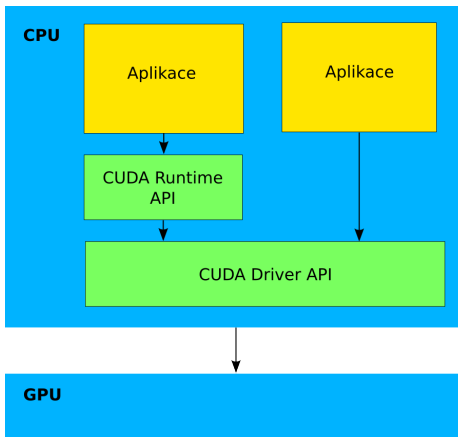
- Proč programovat GPU
- GPU architektura (vs. CPU)
- CUDA (Compute Unified Device Architecture)

## Hierarchie pamětí



# Runtime API vs. Driver API

Vytvářet CUDA aplikace lze užitím buďto **Runtime API** nebo **Driver API**.





# Možnosti rozhraní

Rozhraní umožňují provádět na úrovni hostitelského systému (kód vykonávaný na CPU) následující operace

- Správa zařízení
- Práce s kontextem
- Práce s kernely (moduly)
- Konfigurace výpočtu
- Paměťové operace
- Práce s texturami
- Spolupráce s OpenGL a Direct3D

# Runtime API

- **Runtime API a C for CUDA – množina rozšíření jazyka C**
- Automatická inicializace, práce s kontextem a práce s kernely (moduly)
- Konfigurace výpočtu (volání kernelu) – syntaktický konstrukt (rozšíření jazyka C)
- Kód používající rozšíření musí být přeložen **nvcc** kompilátorem
- Jinak lze hostitelský kód přeložit pomocí gcc

# Příklad kódu používajícího CUDA rozšíření jazyka C

Konfigurace CUDA kernelu `addvec()`

```
int main() {  
    .....  
    addvec<<<<N/BLOCK, BLOCK>>>(d_a, d_b, d_c);  
    .....  
}
```

# Příklad kódu používajícího CUDA rozšíření jazyka C

Konfigurace CUDA kernelu `addvec()`

```
int main() {  
    .....  
    addvec<<<<N/BLOCK, BLOCK>>>(d_a, d_b, d_c);  
    .....  
}
```

Překlad:

```
$ nvcc -I/usr/local/cuda/include -L/usr/local/cuda/lib \  
-lcudart -o vecadd vecadd.cu
```

# Příklad kódu používajícího runtime API volání

## Informace o kartě

```
int main() {  
    .....  
    cudaGetDeviceCount(&devCount);  
    printf(" Available devices: %d\n", devCount);  
  
    cudaGetDeviceProperties(devProp, 0);  
    printf(" Device: %d\n", i);  
    printf(" Name: %s\n", devProp->name);  
    .....  
}
```

# Příklad kódu používajícího runtime API volání

## Informace o kartě

```
int main() {  
    .....  
    cudaGetDeviceCount(&devCount);  
    printf(" Available devices: %d\n", devCount);  
  
    cudaGetDeviceProperties(devProp, 0);  
    printf(" Device: %d\n", i);  
    printf(" Name: %s\n", devProp->name);  
    .....  
}
```

## Překlad:

```
$ gcc -I/usr/local/cuda/include -L/usr/local/cuda/lib \  
-lcudart -x c -o info info.cu
```

# Jak pracovat s kartami – základní funkce

## Zakladní funkce pro výběr karty

- **cudaGetDeviceCount**(*int \*count*) – počet dostupných karet s compute capability  $\geq 1.0$ , pokud v systému není dostupná žádná karta, vrátí funkce hodnotu 1, protože systém podporuje emulační mód – compute capability bude Major: 9999 Minor: 9999
- **cudaSetDevice**(*int dev*) – musí být voláno před inicializací, v opačném případě vrací funkce chybové hlášení `cudaErrorSetOnActiveProcess`
- **cudaGetDevice**(*int \*dev*) – právě používané zařízení

# Jak pracovat s kartami – pokročilé funkce

- **cudaGetDeviceProperties**(*struct cudaDeviceProp \*p, int dev*) – ve struktuře *cudaDeviceProp* vrací informace o zařízení *dev*
- **cudaChooseDevice**(*int \*dev, const struct cudaDeviceProp \*p*) – funkce vybere kartu na základě kritérií *\*p*
- **cudaSetValidDevices**(*int \*dev\_arr, int len*) – seznam karet, ze kterých může být vybíráno
- **cudaSetDeviceFlags**(*int flags*) – nastavuje jak bude CPU vlákno čekat na kartu (Spin, Yield, BlockingSync, Auto) nebo příznak umožňující mapovat paměť. Funkce musí být volána před inicializací



# Práce s pamětí

- Alokace paměti na kartě – **cudaMalloc{Pitch, Array, 3D, 3DArray}()**
  - Lineární paměť
  - 2D paměť a 2D pole
  - 3D paměť a 3D pole
- Kopírování paměti mezi počítačem a kartou (host  $\Leftrightarrow$  device)  
kopírování dat na kartě (device  $\Leftrightarrow$  device) – **cudaMemcpy\*()**
- Alokace paměti v RAM počítače
  - K čemu?

# Kopírování paměti mezi počítačem a kartou

- Základní funkce **cudaMemcpy** (*void \*dst, const void \*src, size\_t count, enum cudaMemcpyKind kind*)
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice, cudaMemcpyHostToHost
- Teoretická přenosová rychlost dosažitelná na PCI Express 2.0 ×16 sběrnici je 8 GB/s. Prakticky však mnohem méně.

# Kopírování dat do karty

Dva přístupy, jeden výrazně rychlejší.

```
int *hmem, *dmem;
hmem = (int *)malloc(SIZE);
cudaMalloc((void**)&dmem, SIZE);

cudaMemcpy(dmem, hmem, SIZE,
           cudaMemcpyHostToDevice);
```

```
int *hmem, *dmem;
cudaMallocHost((void**)&hmem,
              SIZE);
cudaMalloc((void**)&dmem, SIZE);

cudaMemcpy(dmem, hmem, SIZE,
           cudaMemcpyHostToDevice);
```

# Kopírování dat do karty

Dva přístupy, jeden výrazně rychlejší.

```
int *hmem, *dmem;
hmem = (int *)malloc(SIZE);
cudaMalloc((void**)&dmem, SIZE);

cudaMemcpy(dmem, hmem, SIZE,
           cudaMemcpyHostToDevice);
```

- PCI-e 1.0 ×16 1,5 GB/s
- PCI-e 2.0 ×16 4,7 GB/s

```
int *hmem, *dmem;
cudaMallocHost((void**)&hmem,
              SIZE);
cudaMalloc((void**)&dmem, SIZE);

cudaMemcpy(dmem, hmem, SIZE,
           cudaMemcpyHostToDevice);
```

- PCI-e 1.0 ×16 2,8 GB/s
- PCI-e 2.0 ×16 5,5 GB/s

# Page-locked memory

- Page-locked (pinned) paměť umožňuje alokovat funkce **cudaMallocHost**(*void \*\*ptr, size\_t size*)  
nebo:
  - **cudaHostAlloc**(*void \*\*ptr, size\_t size, unsigned int flags*)
    - `cudaHostAllocDefault`, `cudaHostAllocPortable`,  
`cudaHostAllocMapped`, `cudaHostAllocWriteCombined`
- Paměť je alokována jako souvislý blok ve fyzickém adresním prostoru který je navíc uzamčen proti přesunu do swapovacího oddílu
- CUDA totiž může použít pouze DMA přístup, pro který je právě potřeba, aby daný paměťový blok byl umístěn v RAM
- CUDA nepodporuje ani scatter-gather DMA, kdy je možno najednou přistoupit ke množině adres (bloků)
- Toho nelze docílit kombinací volání `malloc()` a `mlock()` (zejména souvislost nelze zajistit z US)

# Page-locked memory

- Není-li paměť alokována tímto způsobem, musí pak driver při kopírování do karty nejprve interně přenést data do "vhodné" paměťové oblasti a odtud je teprve kopírovat do karty (pomocí DMA)
- **cudaHostAlloc()** tedy:
  - Alokuje souvislý blok paměti ve fyzickém adresním prostoru (a namapuje jej do virtuální paměti aplikace)
  - Znemožní přesun této paměti do swapovací oblasti
  - Driver si navíc pro daný kontext (nebo pro všechny) pamatuje že k dané paměti lze přistoupit přímo pomocí DMA

# Souběžný běh výpočtu na GPU a CPU

Aby CPU vlákno mohlo během GPU výpočtu vykonávat další operace a nemusel vždy čekat na GPU, jsou některé CUDA funkce asynchronní. *Příklad: Příprava dalších dat, zatímco probíhá výpočet nad předchozími daty.* Asynchronní je:

- Vykonání kernelu
- Funkce s příponou **Async** určené ke kopírování paměti
- Funkce vykonávající device $\leftrightarrow$ device paměťové kopie
- Funkce vykonávající host $\leftrightarrow$ device paměťové kopie nad daty  $\leq 64\text{KB}$
- Funkce nastavující paměť

# Vykonání CPU funkce během GPU výpočtu

Příklad:

```
cudaMemcpyAsync(dev, hst, cudaMemcpyHostToDevice, 0);  
cpuFunkce();  
kernelFunkce<<<<grid, block>>>(dev);  
cpuFunkce();
```



# Překrývání GPU výpočtu a datových přenosů – použití streams

Má-li GPU schopnost `asyncEngineCount > 0` je možné kopírovat z/do karty a zároveň provádět na kartě výpočet.

- Paměť musí být page-locked (pinned)
- Použití **streams**
  - Representuje posloupnost CUDA volání
  - Volání příslušná různým streamům mohou být vykonána souběžně
  - Streamy lze synchronizovat, případně se dotazovat na stav výpočtu ve streamu

# Příklad překrývání GPU výpočtu a datových přenosů

```
cudaStream_t stream[2];  
for (int i = 0; i < 2; ++i)  
    cudaStreamCreate(&stream[i]);
```

# Příklad překrývání GPU výpočtu a datových přenosů

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);
```

# Příklad překrývání GPU výpočtu a datových přenosů

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

float* hostPtr;
cudaMallocHost((void*)&hostPtr, 2 * size);

for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
```

# Příklad překrývání GPU výpočtu a datových přenosů

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

float* hostPtr;
cudaMallocHost((void*)&hostPtr, 2 * size);

for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < 2; ++i)
    myKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
```

# Příklad překrývání GPU výpočtu a datových přenosů

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

float* hostPtr;
cudaMallocHost((void*)&hostPtr, 2 * size);

for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < 2; ++i)
    myKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);

for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
```

# Příklad překrývání GPU výpočtu a datových přenosů

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);

float* hostPtr;
cudaMallocHost((void*)&hostPtr, 2 * size);

for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < 2; ++i)
    myKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);

for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);

cudaThreadSynchronize();
```

# Detekce chyb

- Všechny runtime funkce (**cuda\*()**) vracejí chybový kód typu **cudaError\_t**
- CUDA runtime udržuje pro každé CPU vlákno chybovou proměnou, která je v případě chyby přepsána chybovou hodnotou posledního volání
- Funkce **cudaGetLastError()** vrací obsah chybové proměnné a zároveň nastaví její hodnotu na **cudaSuccess**
- Chybový kód lze do slovní podoby přeložit voláním **cudaGetErrorString()**
- Návrátová hodnota asynchronních funkcí lze spolehlivě ověřit pouze explicitním voláním **cudaThreadSynchronize()** a ověřením jeho návratové hodnoty



# Příklad detekce chyb

```
cudaError_t err = cudaSetDevice(...);    //< synchronní volání
if(err != cudaSuccess) {
    fprintf(stderr, "Error: '%s'\n", cudaGetErrorString(err));
    exit(CHYBA);
}
```

# Příklad detekce chyb

```
cudaError_t err = cudaSetDevice(...);    //< synchronní volání
if(err != cudaSuccess) {
    fprintf(stderr, "Error: '%s'\n", cudaGetErrorString(err));
    exit(CHYBA);
}
```

```
cudaError_t err;
cudaMemcpyAsync(...);    //< asynchronní volání
err = cudaThreadSynchronize();
if(err != cudaSuccess) {
    fprintf(stderr, "Error: '%s'\n", cudaGetErrorString(err));
    exit(CHYBA);
}
```

# Kompilátor NVCC

- Kompiluje CUDA zdrojové kódy obsahující CPU i GPU kód (host/device code)
- CPU kód je předán externímu kompilátoru – gcc na linuxu, cl ve windows
- GPU kód převeden do PTX formy, dál do binární cubin podoby
- Výsledek GPU kompilace – .cubin výstup – je zabudován do zbytku programu

# Užitečné parametry nvcc kompilátoru

- **--ptxas-options=-v** – mj. zobrazí využití registrů a paměti
- **-G** – zapne debugging pro GPU kód
- **--maxrregcount**  $\langle N \rangle$  – nastaví maximální počet registrů, pro GPU funkce

# Debuging CUDA aplikací

- Obtížnější než na CPU
- Na GPU nelze použít `printf` – na `sm_2.x` lze
- Lze kopírovat mezivýsledky do globální paměti a zpět do RAM počítače – obtížné
- Hledání chybové řádky půlením intervalů (zakomentování řádků)

# CUDA gdb

- Umožňuje za hledání chyb v aplikaci za běhu na GPU
- Port GNU GDB 6.6
- Velmi podobný přístup
- Podporováno na všech kartách s compute capability 1.1 a vyšší
  - Například 8800 Ultra/GTX je pouze 1.0
- Součást CUDA Toolkit

# CUDA gdb

- Zastavení běhu na libovolné CPU i GPU funkci nebo řádku zdrojového kódu
  - (cuda-gdb) break mujKernel
  - (cuda-gdb) break mujKod.cu:45
- Krokování GPU kódu po warpech
  - (cuda-gdb) next – posun po řádcích, nevkročí do funkce
  - (cuda-gdb) step – krok do funkce
- Prohlížení paměti, registrů a speciálních proměnných
  - (cuda-gdb) print blockIdx

```
$ 1 = {x = 0, y = 0}
```

# CUDA gdb

- Vypis informací o použité kartě, paměti alokované na kartě
  - `(cuda-gdb) info cuda state`
- Vypis informací o blocích a vláknech běžících na kartě
  - `(cuda-gdb) info cuda threads`
- Přepnutí na konkrétní blok nebo vlákno
  - `(cuda-gdb) thread<<<Bx, By, Tx, Ty, Tz>>>`



# CUDA gdb

- Program musí být zkompileován s parametry **-g -G**

```
nvcc -g -G -o program program.cu
```

# CUDA Profiler

- Umožňuje analyzovat HW čítače a odhalit neoptimální sekce kódu
- Pro funkci umí zobrazit:
  - Čas strávený na CPU a GPU
  - Obsazení GPU
  - Počet ne/sdružených čtení/zápisů do globální paměti
  - Počet čtení/zápisů do lokální paměti
  - Počet divergentních větvení uvnitř warpu

Hodnoty jsou však měřeny pouze na jednom multiprocesoru, tzn. spíše pro relativní porovnání mezi jednotlivými verzemi kernelu

# Knihovny využívající CUDA

- Součástí CUDA instalace
  - **CUBLAS** – Basic Linear Algebra Subprograms (BLAS)
  - **CUFFT** – Fast Fourier Transform (FFT)
- **CUDPP** – Data Parallel Primitives (DPP)
  - <http://gpgpu.org/developer/cudpp>
  - Například:
    - Paralelní třídění
    - Paralelní redukce
    - Pseudonáhodný generátor čísel
    - BSD licence

# CUBLAS

- Implementace BLAS pro CUDA
- Není potřeba přímá interakce s CUDA API
- Funkce definovány v **cublas.h**
- Jednoduché použití
  - CUBLAS inicializace
  - Alokace paměti na GPU použitím CUBLAS volání
  - Naplnění alokované paměti (kopírování dat)
  - Volání CUBLAS funkcí
  - Získání výsledků (kopírování z karty)
  - Ukončení CUBLAS
- simpleCUBLAS příklad v CUDA SDK

# CUFFT

- Implementace FFT pro CUDA
- Vyžaduje použití základních runtime API volání (`cudaMalloc()`, `cudaMemcpy()`)
- Funkce definovány v **`cufft.h`**
- 1D, 2D, 3D transformace na reálných i komplexních číslech
- `simpleCUFFT` příklad v CUDA SDK

# Závěr

Dnes jsme si ukázali

- Jak programovat CUDA aplikace
- Základní funkce runtime API
- Jak efektivně využít šířku PCIe sběrnice při kopírování dat
- Jak souběžně vykonávat CPU a GPU kód (překrývání)
- Jak hledat chyby – cuda-gdb
- Knihovny používající CUDA

# Samostatná práce

## K samostatné práci

- Zkuste změřit jaké rychlosti jste schopni dosáhnout při přenosu dat p jednoduchý program, který vypíše základní informace o vaši kartě (zkuste takovýto program spustit na systému bez CUDA enabled karty)
- Na kódu z minulé přednášky vyzkoušejte použití `cuda-gdb` a `cuda-prof`