Path-tracer na platformě CUDA

Tomáš Král

***Abstract*—CUDA je platforma pro GPGPU od firmy Nvidia. Tento text pojednává o využití CUDA pro tvorbu path-traceru.**

I. Host-Device model

Základní vlastností platformy CUDA je, že program se dělí na 2 části. První se nazývá *host* část, tedy část programu běžící na CPU, využívající RAM paměť a ostatní prostředky. Druhá část se nazývá *device* část a označuje programy běžící na jedné nebo více GPU, využívající GPU paměť. Program běžící na GPU se ozančuje pojmem *kernel*.

II. Programátorské nástroje pro CUDA

A. Programovací jazyky

Jak *host*, tak *device* část programu je možné vytvářet pomocí různých jazyků:

1. *host* část je odpovědná zejména za řízení běhu GPU. Využívá na to nízkoúrovňové driver API, které lze ovládat z více programovacích jazyků (C, C++, Python, Rust...).
2. *device* část běží na samotné GPU a proto je zde výběr omezenější než u *host* části, protože pro daný jazyk musí existovat kompilátor, který je schopen kompilace pro GPU architekturu. Nvidia podporuje kompilátory pro C, C++ a Fortran. Existují také neoficiální kompilátory např. pro jazyk Rust, ale ty jsou spíše experimentální.

Nejjednodušším způsobem je použití jazyka C++ a kompilátoru NVCC (Nvidia CUDA Compiler). NVCC je speciální kompilátor, který umožňuje kombinovat *host* a *device* programy v jednom souboru. Při kompilaci si NVCC kód rozdělí na *host* a *device* části a každou zkompiluje pro dané architektury.

NVCC dále poskytuje rozšíření jazyka C++, které umožňují programování v CUDA bez explicitní práce s nízkoúrovňovým driver API. Např. spuštění triviálního kernelu vypadá takto:

```
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

Atributa `__global__` značí, že se funkce má zkompilovat jako GPU kernel. Kernel se poté spouští pomocí rozšířené syntaxe jazyka C++ `cuda_hello<<<1,1>>>()`, kde čísla v závorkách určují, kolikrát se má kernel spustit.

NVCC kompilátor pro *device* část má podporu pro nové standardy jazyka C++ až do verze C++20 avšak s různými omezeními. Hlavními omezeními v *device* kódu jsou [1]:

- Funkce nepodporují rekurzi.
- Není možné používat exceptions.
- Omezené použití virtuálních funkcí.

Z popsaných omezení vyplývá, že v *device* kódu **nelze využívat standardní knihovnu**. Nvidia nabízí jako náhradu knihovnu libcu++, která je ale oproti plné standardní C++ knihovně dosti omezená.

B. Další nástroje

Nvidia vytvořila pro tvorbu a ladění CUDA programů několik nástrojů, mezi které patří:

- cuda-gdb - verze debuggeru GDB, která umožňuje ladění kernelů.
- Nvidia NSight Computer - grafický profiler kernelů.
- compute-sanitizer - CLI program, který umí odhalit různé chyby týkající se práce s pamětí či synchronizace v kernelech.

III. Práce s pamětí

Z modelu *host-device* je zřejmé, že musí být věnován zvláštní důraz na práci s pamětí, neboť se zde pracuje s dvěma zařízeními, které mají vlastní, oddělenou paměť: CPU s pamětí RAM a GPU s pamětí VRAM. Existují 2 druhy práce s pamětí. První, klasický způsob, vnímá oba adresní prostory odděleně. V tomto režimu je nutné data manuálně synchronizovat mezi pamětovými prostory. Typický postup vypadá následujícím způsobem:

1. Alokace paměti v RAM.
2. Načtení dat z disku do RAM.
3. Alokace paměti ve VRAM.
4. Zkopírování dat do VRAM.
5. Spuštění kernelu a provedení výpočtů.
6. Zkopírování výsledků zpět do RAM.

Nevýhodou manuálního kopírování je zaprvé potřeba většího množství kódu a za druhé obtížnost tvorby složitějších datových struktur.

Novější verze CUDA umožňují pracovat s pamětí pomocí tzv. *Unified Memory* funkcionality. Tato funkcionalita umožňuje alokovat paměť a přistupovat k ní pomocí *stejných pointerů* jak z CPU, tak z GPU. Unified Memory je implementována uvnitř Cuda Runtime, který transparentně přesouvá paměť mezi RAM a VRAM podle toho, kde se zrovna používá. Důsledkem je, že ve chvíli kdy beží jakýkoliv[1] kernel, tak není možné z CPU přistupovat k pointerům do Unified Memory. Proto je nutné před přístupem k takovým pointerům vyčkat na dokončení všech běžících kernelů.

Předchozí manuální postup se s použitím Unified Memory značně zjednoduší:

1. Alokace paměti v Unified Memory.
2. Načtení dat z disku do Unified Memory.
3. Spuštění kernelu a provedení výpočtů.
4. Vyčkání na dokončení kernelu pomocí `cudaDeviceSynchronize()`.

Nicméně stále platí, že práce s pointery je náchylná na chyby. Zejména je třeba dbát na to, aby kernely pracovaly pouze s lokálními proměnnými, daty v Unified Memory anebo s daty alokovanými přímo v *device* paměti, ale nikdy s pointery na stacku nebo heap paměťi v RAM.

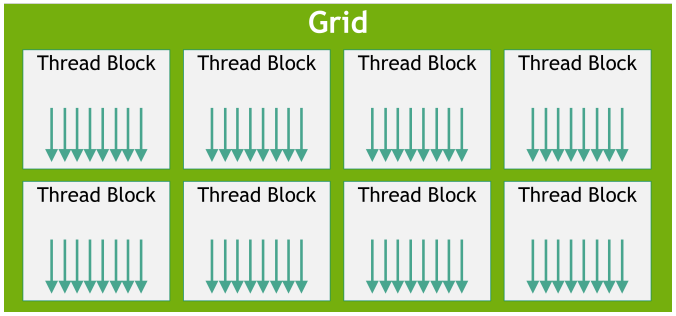
IV. Hierarchie treadů v CUDA

Jak bylo zmíněno, při spuštění kernelu se používá speciální syntaxe `jmeno_kernelu<<<blockDim, threadDim>>>()`, která určuje počet invokací kernelu. První proměnná určuje *dimenze bloků* a druhá proměnná určuje *dimenze threadů* v každém bloku.

Každý thread označuje separátní běh kernelu, tedy jednotku, která je schopna vykonat samostatnou práci. Například program který by měl za úkol pro každý pixel obrázku provést určitou operaci, by pro každý pixel obrázku spuštil jeden thread. Každý blok označuje separátní spuštění skupiny threadů. Thready v jednom bloku jsou na GPU spouštěny vždy najednou. To však neplatí pro jednotlivé bloky, které mohou být spuštěny v libovolném pořadí.

Thready a bloky jsou organizované do hierarchie, viz. Obr. 1. Každý blok se skládá ze stejné dimenze (stejného množství) threadů. Bloky jsou uskupeny do tzv. *gridu*. Dimenze bloků či threadů je 3-rozměrný vektor a určuje jejich počet a seskupení. Například pro zpracování každého pixelu obrázku s velikostí 512 na 256 pixel by bylo možné definovat dimenzi threadů jako dim(8, 8, 0) a dimenzi bloků jako dim(64, 32, 0). Tedy každý thread blok by zpracoval 8x8 pixelů a celkově by se spustilo 64x32 bloků.

[1]TODO: Zde si musím ověřit, zda to platí opravdu když běží jakýkoliv kernel nebo jen kernel přistupující do konkrétního bloku alokované paměti.



Obr. 1: Hierarchie threadů [1].

Spuštěný kernel má přístup k několika speciálním proměnným:

- `threadIdx`, která identifikuje souřadnice threadu v bloku.
- `blockIdx`, která identifikuje souřadnice bloku v gridu.
- 'blockDim', která určuje dimenze bloku.

Kde souřadnice threadů a bloků poté nabývají všech hodnot od dim(0, 0, 0) až do dim(x, y, z). Pomocí těchto proměnných je možné pro každý kernel spočíst unikátní souřadnici. Například pro předchozí příklad s obrázkem by se xy souřadnice pixelu spočítala jako `(blockIdx * blockDim) + threadIdx`;

Maximální počet threadů v jednom bloku na dnešním hardwaru je 1024, běžné množství je 256 [1].

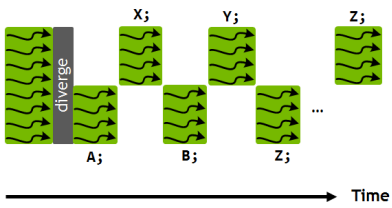
V. Paralelismus na GPU

A. Grafický hardware

Základní vlastnost hardwaru od firmy Nvidia je, že výpočty jsou vždy prováděny ve skupině 32 threadů, která se nazývá *warp*. Všechny thready ve warpu běží paralelně. Problém může nastat ve chvíli, kdy některé thready potřebují vykonat jiný kód než ostatní thready. Taková situace může nastat například když jedna skupina threadů splňuje podmínku v konstruktu `if-else` a druhá nikoliv. Tato situace se ozančuje pojmem *divergence*.

Divergence je na hardwaru řešena tím, že některé thready jsou *zamaskovány*, tzn. po dobu vykonávání divergentní části kódu jsou vypnuty, viz. Obr. 2:

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Obr. 2: Vizualizace maskování threadů při divergenci [2].

Divergence má pochopitelně negativní vliv na rychlost výpočtu - čím více budou jednotlivé thready divergovat, tím nižší bude úroveň paralelizace. Pro maximálně efektivní

využití hardwaru je tedy nutné implementovat algoritmy tak, aby běh programu pokud možno co nejméně divergoval.

B. Paralelizace path-traceru

Path-tracing je možné paralelizovat několika způsoby. Asi nejjednodušší by bylo rozdělit výpočet na úrovni pixelů - co jeden pixel, to jeden thread. Jasnou nevýhodou takového postupu je vysoká divergence. Známou vlastností algoritmů path-tracingu totiž je, že množství výpočtů se pro různé pixely může masivně lišit. To je způsobené například odlišnou složitostí scény v různých segmentech obrazu.

Lepším způsobem by bylo paralelizovat výpočet jednotlivých vzorků jednoho pixelu. Pro typické použití path-tracingu je totiž běžné pro každý pixel počítat stovky až tisíce vzorků. Touto cestou je možné do jisté míry snížit množství divergence vycházející z rozdílných výpočetních nároků pro různé pixely. Nicméně zůstává divergence způsobená např. výpočtem různých BRDF, materiálů, průsečíků s různými geometrickými útvary atd.

Řešením problému paralelizace je tzv. *Wavefront* algoritmus [3], který vychází z tzv. *streaming path-tracing* postupu [4].

VI. Implementace

A. Konfigurace systému a požadavky

Implementace byla provedena na platformě CUDA 12.2. Byl použit standard C++20 na kompilátoru GCC 13.2.1. Program byl testován na grafické kartě Nvidia GeForce MX 550M na OS Linux s verzí kernelu 6.5.4.

Implementace používá Unified Memory, která vyžaduje GPU s architekturou SM 3.0 nebo vyšší (řada Kepler a novější). Některé pokročilé funkce Unified Memory jsou dostupné pouze na OS Linux, ty ale **doufám** nebyly použity.

Bylo použito několik externích knihoven: fmt, GLM. Tyto knihovny jsou do projektu zakomponovány pomocí package manageru vcpkg a buildovacího systému CMake.

References

[1] "1. Introduction — CUDA C Programming Guide." Accessed: Sep. 29, 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#unified-memory-introduction

[2] "Inside Volta: The World's Most Advanced Data Center GPU," 2017. Accessed: Sep. 28, 2023. [Online]. Available: https://developer.nvidia.com/blog/inside-volta/

[3] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: wavefront path tracing on gpus," in *Proc. 5th High-Performance Graph. Conf.* in Hpg '13, Anaheim, California, 2013, p. 137, doi: 10.1145/2492045.2492060. [Online]. Available: https://doi.org/10.1145/2492045.2492060

[4] D. G. Van Antwerpen, "Unbiased physically based rendering on the GPU," 2011. Accessed: Sep. 29, 2023. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid%3A4a5be464-dc52-4bd0-9ede-faefdaff8be6