

Paralelní programování na GPU (PCG 2023)

Projekt č. 1: CUDA

David Bayer (ibayer@fit.vutbr.cz)

1 ÚVOD

Cílem tohoto projektu je implementovat simulaci částicového systému na grafické kartě pomocí technologie CUDA. Projekt se skládá z několika částí, které na sebe navazují a postupně rozšiřují základní implementaci. V každé části je potřeba implementovat funkcionality vyznačenou v kódu pomocí komentáře `TODO`. Ke krokům se rovněž pojí otázky v souboru *nbody.md*, které je třeba zodpovědět.

2 PRÁCE NA SUPERPOČÍTAČI KAROLINA

Pro připojení na superpočítač Karolina je potřeba mít vytvořený účet, se kterým je možné se připojit na tzv. čelní (login) uzel – `karolina.it4i.cz`. Tento uzel **neslouží** ke spouštění náročných úloh, veškeré experimenty je nutné provádět na výpočetních uzlech. Software je na superpočítači dostupný pomocí modulů. Pro načtení modulů potřebných pro tento projekt slouží skript *loadModules.sh*, který je třeba spustit v shellu po každém přihlášení jako

```
[user@login1.karolina PCG-proj1]$ . loadModules.sh
```

Pro spuštění úlohy je nejjednodušší vytvořit interaktivní úlohu, např. pomocí následujícího příkazu:

```
[user@login1.karolina PCG-proj1]$ salloc -A DD-23-135 -p qgpu_exp -G 1 -t 01:00:00 -I
salloc: Granted job allocation 55125
salloc: Waiting for resource configuration
salloc: Nodes acn01 are ready for job
[user@acn01.karolina PCG-proj1]$
```

Příkaz `salloc` přidá požadavek na spuštění úlohy do fronty. Jakmile bude v systému dostatek volných uzlů, dojde k jejímu spuštění. Parametr `-A` určuje projekt, v rámci kterého máme alokované výpočetní hodiny (neměnit) a `-p` určuje frontu, do které bude úloha zařazena.

Fronta `qgpu_exp` má vysokou prioritu, úloha je ale omezena na maximálně 1 hodinu. Pro delší úlohy je třeba zvolit frontu `qgpu`. Jiné fronty neumožňují přístup na akcelerované GPU uzly. Parametr `-G` určuje počet GPU alokace, ponechte pouze 1, více v projektu nepoužijeme (na Barboře není povolena částečná alokace uzlu, parametr `G` tam tedy nepoužívejte). Parametr `-t` specifikuje délku alokace a parametr `-I`, že se jedná o interaktivní úlohu. Více o spouštění úloh na superpočítačích IT4I naleznete na stránce <https://docs.it4i.cz/general/job-submission-and-execution/>.

3 PŘEKLAD PROJEKTU

Pro překlad programu na osobním počítači je třeba mít nainstalován programy CMake, překladač pro C a C++ podporovaný CUDA toolkitem, knihovnu HDF5 a Python 3. CUDA Toolkit lze stáhnout a nainstalovat z <https://developer.nvidia.com/cuda-downloads>.

3.1 NA SUPERPOČÍTAČI

Na superpočítačích Barbora i Karolina jsou všechny potřebné prerekvizity již nainstalovány, pouze je nutné nahrát jejich moduly, jak bylo popsáno v přechozí kapitole. Pro sestavení projektu postupujte stejně jako u linuxové varianty (viz další podkapitola), vynechejte však instalaci závislostí.

3.2 U VÁS NA PC

Pokud vlastníte PC s NVIDIA GPU, můžete celý projekt vypracovat a ozkoušet u sebe na PC. V opačném případě máte i tak možnost pracovat u sebe na PC, nicméně přijdete o možnost projekt spustit.

3.2.1 LINUX

Po instalaci CUDA Toolkitu je na linuxových systémech (včetně WSL) všechny potřebné závislosti nainstalovat pomocí příkazu:

```
$ sudo apt install -y gcc g++ cmake libhdf5-serial-dev hdf5-tools python3-dev python3-pip python3-venv
```

Poté je potřeba vytvořit složku `build` pro překlad a vygenerovat `Makefile` script pomocí programu CMake. Pro ověřování správnosti programu použijte konfiguraci `Debug`, pro měření výsledků pak konfiguraci `Release`.

```
$ mkdir build
$ cmake -DCMAKE_BUILD_TYPE=[Debug|Release] -S. -Bbuild
```

Testování správnosti implementace lze ověřit pomocí skriptu `runTests.sh` s cestou k testovanému binárnímu souboru jako

```
$ ./runTests.sh build/nbody[Cpu|0-4]
```

3.2.2 WINDOWS

Po instalaci CUDA Toolkitu je na systémech Windows (včetně WSL) nainstalovat HDF5 knihovnu a nástroje z <https://www.hdfgroup.org/downloads/hdf5> a Python 3 (lze získat např. z Microsoft Store). Ostatní závislosti byste měli mít nainstalovány jako součást Visual Studia, v případě potřeby je doinstalujte.

Pro vygenerování solution souboru je třeba otevřít Visual Studio Command Prompt a vytvořit v rozbaleném archivu složku *build*. Poté je potřeba vygenerovat solution soubor pomocí programu CMake. Pro ověřování správnosti programu použijte konfiguraci Debug, pro měření výsledků pak konfiguraci Release. Překlad lze poté provést i přímo ve Visual Studiu.

```
$ mkdir build
$ cmake -DCMAKE_BUILD_TYPE=[Debug|Release] -S. -Bbuild
```

Testování správnosti implementace lze ověřit pomocí skriptu *runTests.bat* s cestou k testovanému binárnímu souboru jako

```
$ runTests.bat build/nbody[Cpu|0-4].exe
```

4 ČÁSTICOVÝ SYSTÉM (20 BODŮ)

Cílem tohoto projektu bude nejprve implementovat a posléze optimalizovat výpočet vzájemného silového působení N těles. Každé těleso má jistou hmotnost, polohu v prostoru a rychlost. Gravitační síly působící na dané těleso od ostatních těles mají různé směry a jejich výslednice způsobuje změnu rychlosti pohybu tohoto tělesa. Pro vektory polohy \mathbf{r} a rychlosti \mathbf{v} platí:

$$\mathbf{r}^{i+1} = \mathbf{r}^i + \mathbf{v}^{i+1} \cdot \Delta t \quad (4.1)$$

$$\mathbf{v}^{i+1} = \mathbf{v}^i + \mathbf{v}_g^{i+1} + \mathbf{v}_c^{i+1} \quad (4.2)$$

kde \mathbf{v}_g^{i+1} je přírůstek rychlosti vzniklý gravitačním působením těles a \mathbf{v}_c^{i+1} je změna rychlosti vlivem kolize s některými tělesy.

Síla působící na těleso je dána vektorovým součtem dílčích sil způsobených gravitačním polem ostatních těles. Dvě tělesa na sebe působí gravitační silou danou:

$$F = \frac{G \cdot m_1 \cdot m_2}{r^2}, \quad (4.3)$$

kde $G = 6.67384 \cdot 10^{-11} \text{Nm}^2\text{kg}^{-2}$ je gravitační konstanta, m_1 a m_2 jsou hmotnosti těles a r je jejich vzdálenost. Rychlost, kterou těleso obdrží díky této síle pak lze vyjádřit jako:

$$\mathbf{v}_g^{i+1} = \frac{\sum \mathbf{F}_j^{i+1}}{m} \cdot \Delta t \quad (4.4)$$

Pokud se tělesa dostanou do příliš blízké vzdálenosti, dané konstantou `COLLISION_DISTANCE`, dojde k jejich odrazu. Částice si můžete představit jako koule s poloměrem daným polovinou této konstanty. Pro jednoduchost mají všechna tělesa stejný poloměr. Rychlosti dvou těles po odrazu lze určit ze zákona zachování hybnosti a kinetické energie.

$$v_1 \cdot m_1 + v_2 \cdot m_2 = w_1 \cdot m_1 + w_2 \cdot m_2 \quad (4.5)$$

$$\frac{1}{2} \cdot v_1^2 \cdot m_1 + \frac{1}{2} \cdot v_2^2 \cdot m_2 = \frac{1}{2} \cdot w_1^2 \cdot m_1 + \frac{1}{2} \cdot w_2^2 \cdot m_2 \quad (4.6)$$

kde m_1 a m_2 jsou hmotnosti těles, v_1 a v_2 jsou rychlosti těles před kolizí a w_1 a w_2 jsou rychlosti těles po kolizi. Rovnice 4.5 je zákon o zachování hybnosti a rovnice 4.6 je zákon o zachování kinetické energie. Řešením těchto dvou rovnic o dvou neznámých pro w_1 získáváme novou rychlost tělesa. Jelikož v daném kroku mohou na těleso působit i ostatní tělesa, je potřeba získat pouze rozdíl oproti původní rychlosti, který se na původní rychlost aplikuje později.

Změna rychlosti v daném kroku lze pak vyjádřit jako

$$v_c = w_1 - v_1 \quad (4.7)$$

Pro všechny elementy pak platí

$$\mathbf{v}_c^{i+1} = \sum \mathbf{v}_c^{i+1} \quad (4.8)$$

V každém kroku výpočtu je nutné spočítat změny rychlostí a poloh jednotlivých těles.

5 ÚKOLY

Tato kapitola se věnuje implementaci, testování a vyhodnocení projektu. Obsahuje celkem 5 kroků, za které lze získat až 20 bodů. Důležité části implementace dokumentujte komentáři. Pro kontrolu návratových hodnot CUDA API volání použijte makro `CUDA_CALL`. K téměř všem krokům se také pojí otázky v souboru *nbody.md*, které je třeba zodpovědět. Při kontrole průchodu testy **Veškerá měření výsledků provádějte na superpočítači Karolina**.

5.1 KROK 0: ZÁKLADNÍ IMPLEMENTACE (4 BODY)

Kostra aplikace je připravena v adresáři `Step0`.

1. Nejprve správně doplňte definici struktur `Particles` a `Velocities` v hlavičkovém souboru *nbody.cuh*. Použijte vhodné datové typy tak, aby se omezil počet přístupů do globální paměti.
2. Následně v souboru *main.cu* doplňte části pro alokaci a dealokaci paměti proměnné `hParticles` na CPU.
3. Poté upravte konstrukci deskriptoru paměti `MemDesc` tak, aby byla správně načtena data ze vstupního souboru. Mějte na paměti, že rozestupy a offsety dat jsou uváděny v počtech floatů, nikoli v bajtech.
4. Pak doplňte funkce pro alokaci a dealokaci paměti na GPU proměnných `dParticles` a `dTmpVelocities`.
5. Doplňte části pro kopírování dat z CPU na GPU a zpět.

6. Doplňte volání kernelů `calculateGravitationVelocity`, `calculateCollisionVelocity` a `updateParticle` v hlavní smyčce programu s velikostí gridu `simGridDim` a bloku `simBlockDim`.
7. Nakonec implementujte samotné kernely v souboru `nbody.cu` podle referenční CPU verze tak, aby by správně simulován pohyb částic s časovým posuvem `dt` sekund. **Při implementaci se soustřeďte zejména na efektivitu práce s pamětí.**

Správnost implementace ověřte pomocí testů skriptem `runTests` a porovnáním referenčního výstupu pomocí skriptu `checkOutput`. Odchyšky v řádech desetin signalizují, že je ve výpočtu významná chyba. **Průchod testy je nutnou, ne však postačující podmínkou pro udělení bodů z každého úkolu.**

Po ověření správnosti vyplňte tabulku v souboru `nbody.md` a odpovězte na dotazy. Pro naměření časů simulace použijte skript `runProgressBenchmark`.

5.2 KROK 1: SLOUČENÍ KERNELŮ (3 BODY)

V kroku 1 je vaším úkolem sloučit všechny kernely do jednoho kernelu `calculateVelocity`. Zrychlí se tak výpočet simulace, sníží se počet přístupů do globální paměti a nadále nebude třeba používat dočasné úložiště pro rychlost. Aby nebylo nutné synchronizovat vlákna před zápisem do paměti, alokujte strukturu `Particles` dvakrát a v každém kroku výpočtu používejte jednu kopii jako vstup a druhou jako výstup. Pro identifikaci aktuálního indexu vstupní a výstupní kopie použijte proměnné `srcIdx` a `dstIdx` definované ve výpočetní smyčce. Po skončení simulace budou aktuální data uložena v kopii na indexu `resIdx`. Nezapomeňte před začátkem simulace inicializovat daty obě kopie.

Po ověření správnosti vyplňte tabulku v souboru `nbody.md` a odpovězte na dotazy. Pro naměření časů simulace použijte skript `runProgressBenchmark`.

5.3 KROK 2: SDÍLENÁ PAMĚŤ (4 BODY)

V tomto kroku je vaším úkolem optimalizovat přístupy do paměti v kernelu `calculateVelocity` pomocí sdílené paměti. Identifikujte data, která jsou opakovaně načítána různými vlákny v bloku a umožněte jejich sdílení prostřednictvím sdílené paměti. Řešení implementujte tak, aby velikost sdílené paměti byla závislá na velikosti spouštěného bloku (proměnná `simShmSize`). Pro ověření správnosti vyplňte tabulku v souboru `nbody.md` a odpovězte na dotazy. Pro naměření časů simulace použijte skript `runProgressBenchmark`.

5.4 KROK 3: IMPLEMENTACE VÝPOČTU TĚŽIŠTĚ NA GPU (4 BODY)

V kroku 3 implementujte výpočet polohy a hmotnosti těžiště v kernelu `centerOfMass`, který spusťte po dokončení simulace. Jako inspirace vám může posloužit CPU implementace. Jak už nejspíš tušíte, výpočet těžiště vede na redukci. **Tu, prosím, implementujte bez požití dalších knihoven** (např. `cooperative_groups`). V implementaci minimalizujte práci s globální pamětí, maximalizujte aritmetickou intenzitu a výměnu dat ve sdílené paměti. Jak na to:

1. Nejprve je třeba v souboru *main.cu* implementovat alokaci a dealokaci paměti na CPU pro strukturu těžiště (používáme pro ni strukturu `float4`) proměnné `hCenterOfMass`.
2. Poté přidejte alokaci, dealokaci a vynulování těžiště na GPU proměnné `dCenterOfMass`. Stejně tak postupujte v případě zámku `dLock`, který bude sloužit pro exkluzivní přístup vláken k objektu těžiště v globální paměti při redukci.
3. Následně nastavte správný výpočet pro množství sdílené paměti výpočtu těžiště (proměnná `redShmSize`) a přidejte volání kernelu `centerOfMass` s rozměry gridu `redGridDim` a rozměry bloku `redBlockDim` před synchronizací GPU.
4. Pak se můžete pustit do implementace výpočtu těžiště. Nejprve naimplementujte redukci v rámci bloku s využitím sdílené paměti. Můžete také využít optimalizace v podobě redukce v rámci warpu s využitím výměny registrů. Spočtený mezivýsledek dále redukuje do globální paměti. Pro exkluzivní přístup k těžišti v globální paměti využijte zámek `lock` a atomické operace nad ním.

Pro získání plného počtu bodů je nutné implementovat redukci tak, aby i menší tým vláken než je velikost vstupu byl schopen spočítat správný výsledek. Správnost výpočtu určíte porovnáním s CPU verzí.

5.5 KROK 4: PARALELISMUS NA ÚROVNI KERNELŮ A SYNCHRONIZACE (4 BODŮ)

Při počítání simulací nás často nezajímá pouze její konečný stav, chceme znát i její průběh. Cílem tohoto kroku je tedy umožnit průběžné ukládání mezivýsledků simulace tak, abychom co nejlépe vykryli přenos dat užitečnou prací. K tomu bude potřeba využít vícestreamové zpracování a synchronizaci.

V každém kroku je z polohy částic v čase t (na indexu `srcIdx`) počítána nová poloha částic v čase $t + 1$ (index `dstIdx`). Lambda funkce `shouldWrite` na základě kroku simulace a frekvence zápisu `writeFreq` určuje, zda se v daném kroku má zapsat aktuální poloha částic (index `srcIdx`) do výstupního souboru. Je tedy nutné data překopírovat z GPU zpět na CPU a provést zápis pomocí metody `writeParticleData` objektu `h5Helper`. Číslo záznamu získáte pomocí lambda funkce `getRecordNum`. Zároveň s tím je také potřeba spočítat na GPU aktuální těžiště, překopírovat jej na CPU a uložit do výstupního souboru pomocí metody `writeComData` objektu `h5Helper`. Nezapomeňte před každým výpočtem těžiště vynulovat jeho hodnotu v globální paměti GPU.

Současně tedy bude ve vybraných krocích probíhat výpočet nové polohy částic, kopírování dat z GPU na CPU a výpočet těžiště. Pro každou z těchto úloh vytvořte vlastní stream. Nepoužívejte výchozí stream (zero stream), ve výchozím nastavení je implicitně synchronní. Synchronizaci streamů mezi sebou a s CPU implementujte pomocí CUDA Stream API a CUDA Event API. Po dokončení hlavní smyčky opět spočítejte finální těžiště stejně jako v kroku 3.

Ověření správnosti proveďte tak, že vygenerujete vstupní data programem `gen` libovolné velikosti, spustíte simulaci na CPU, poté na GPU a nakonec porovnejte jejich výstupy skriptem `compare`. Příklad pro data simulaci o 4096 částicích:

```
$ ./gen 4096 input4096.h5
$ ./nbodyCpu 4096 0.01f 100 512 5 2048 128 input.h5 outputCpu.h5
$ ./nbody4 4096 0.01f 100 512 5 2048 128 input.h5 outputGpu.h5
$ ../compare.sh outputCpu.h5 outputGpu.h5
```

Za tuhle část je možné získat plný počet bodů, i když jste neimplementovali (nebo implementovali chybně) krok 3, jednoduše použijte kód z kroku 2 a doplňte synchronizaci ve funkci `main.cu` dle zadání. Plný počet lze získat spouštěním prázdných kernelů ve vhodných místech kódu, vše co se týče konkurence a synchronizace pak ale musí být v pořádku.

5.6 KROK 5: ANALÝZA VÝKONU (2 BODY)

Nad vyhotoveným binárním souborem z kroku 4 spusťte skript *runFinalBenchmark* a výsledky запиšte do tabulky v souboru *nbody.md*. Spočtete dosažené zrychlení oproti naměřené referenční CPU implementaci, propustnost paměti a celkový výkon. Dále odpovězte na otázky v dokumentu. Referenční časy simulace na CPU byly naměřeny na CPU uzlech Karoliny (celkem 128 jader).

6 VÝSTUP PROJEKTU A BODOVÁNÍ

Výstupem projektu bude soubor *xlogin00.zip* obsahující celý původní vámi modifikovaný archiv bez binárních a vstup/výstupních souborů simulací. V každém souboru nezapomeňte změnit svůj login! Hodnotit se bude jak funkčnost a správnost implementace, tak textový komentář – ten by měl dostatečně popisovat rozdíly mezi jednotlivými kroky a odpovídat na otázky uvedené v zadání. Při řešení se soustřed'te především na správnost použití CUDA, přesnost výpočtu je závislá na mnoha okolnostech, např. zvoleném výpočtu, pořadí operací apod., a pokud bude v rozumných mezích, nebude hrát velkou roli při hodnocení. Projekt odevzdejte v uvedeném termínu do informačního systému.