Masarykova univerzita
Fakulta informatiky

Paralelizace metod pro analýzu dynamických systémů pomocí grafické karty

Bakalářská práce

Jan Papoušek

Brno, Jaro 2011

Prohlášení

Prohlašuji, že tato bakalářská práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

V Brně dne 19. května 2011
Jan Papoušek

Vedoucí práce: Mgr. Sven Dražan

Poděkování

Děkuji svému vedoucímu práce Svenu Dražanovi za odborné vedení a poskytnutí cenných rad, svým rodičům za podporu a zázemí, které mi při studiu a během psaní této práce poskytli, a své přítelkyni Tereze Doležalové, jež mi pomáhala se směřováním práce a korekturami.

Shrnutí

Tato práce prezentuje postup, jak paralelizovat numerickou simulaci nad velkým množstvím iniciálních bodů v rámci existující metody pro analýzu dynamických systémů. Vychází ze známých metod pro numerickou simulaci, konkrétně Eulerova explicitního a Runge-Kutta-Fehlbergova schématu. Tyto metody byly implementovány tak, aby je bylo možné spustit na grafické kartě za použití technologie CUDA.

Vzniklá knihovna je implementovaná v jazyce CUDA C a obsahuje rozhraní pro použití v prostředí Javy. Může být použita nejen v rámci existujícího programu pro analýzu dynamických systémů, ale i v jakémkoliv jiném projektu, kde je potřeba vygenerovat trajektorie z více počátečních bodů na základě soustavy diferenciálních rovnic.

Výstupem práce však není pouze samotná implementace, ale i měření výkonu a následná analýza, která může usnadnit paralelizaci dalších algoritmů s využitím technologie CUDA.

Klíčová slova

dynamický systém, soustava diferenciálních rovnic, problém výchozích podmínek, diskretizace, numerická simulace, CUDA

Obsah

Úvod

V poslední době se čím dál více mluví o nutnosti zmnožování výpočetních jader a využívání paralelních algoritmů. Grafické karty, které dříve sloužily jako čistě jednoúčelová zařízení, dnes představují se svými multiprocesory alternativu ke starému výpočetnímu modelu sekvenčího programu běžícího na jednom procesoru. Roku 2006 společnost NVIDIA uveřejnila architekturu CUDA [14, str. 3], která umožnila grafické karty používat nejen jako nástroj pro zobrazování grafiky, ale také jako plnohodnotné výpočetní zařízení.

Současně se v jiných kruzích objevila snaha zachytit svět kolem nás do modelů, nad kterými by se následně daly provádět zautomatizované analýzy. Významným odvětvím, ve kterém modely pomalu nabývají na síle, je biologie [10][20]. Biologické jevy však mohou být velice komplikované a práce s modely často naráží na limity výpočetních zdrojů. Proto se algoritmy pro analýzu těchto modelů akcelerují, někdy dokonce s použitím grafické karty [6][15][11].

Tato práce se věnuje akceleraci numerické simulace pro metodu analýzy dynamických systému prezentovanou Svenem Dražanem v jeho diplomové práci [7]. Jistý přístup, jak akcelerovat řešení tohoto problému, již existuje [3]. Je však spíše zaměřen na překlad rovnic popisující model do zdrojového kódu, než na vlastní numerickou simulaci.

Nejprve představím pojmy, se kterými se bude dále pracovat, mezi tyto pojmy patří *dynamický systém*, *problém výchozích podmínek* a samotná *metoda pro analýzu dynamických systémů*. Následně se budu věnovat metodám numerické simulace, jejich rozdělení a způsobu, jakým lze ovlivnit přesnost výpočtu. Další část práce obsahuje popis základních aspektů architektury CUDA, zejména těch, které se liší od vlastností klasických procesorů a které mohou zásadně ovlivnit rychlost výpočtu. Poté se budu věnovat vlastní implementaci numerické simulace s využitím architektury CUDA a měření jejího výkonu.

Kapitola 1

Pojmy a východiska

1.1 Dynamické systémy

Ve světě okolo nás je mnoho jevů, které se vyvíjí v čase na základě interakce více či méně proměnných. Často se stává, že vazby mezi částí proměnných je možné odlišit od ostatních. Pokud se jedná o část skládající se z více menších celků, lze prohlásit tuto část za *dynamický systém* [17]. Mezi typické dynamické systémy patří soustavy, ve kterých probíhají chemické reakce.

U některých jednoduchých systémů lze jejich chování snadno předpovědět, ale pro mnoho dalších může být jakákoliv úvaha o nich velice komplikovaná, zvláště pak v případě, že systém obsahuje mnoho proměnných, zpětné vazby a nelineární chování. Naštěstí lze dynamické systémy do určité míry formálně specifikovat a vytvořit model, u kterého je možné si ověřit některé hypotézy [16].

Jedním způsobem, jak vytvořit model dynamického systému, je soustava diferenciálních rovnic prvního řádu.

1.2 Problém výchozích podmínek

Problémem výchozích podmínek [12, str. 3][18] se rozumí soustava diferenciálních rovnic prvního řádu, jenž se dá pro jednoduchost zapsat funkcí $f : [t_0, \infty) \times \mathbb{R}^d \to \mathbb{R}^d$. t_0 představuje čas, od kterého systém pozorujeme, a d počet složek systému. $\mathbf{y}'$ zde a dále v textu značí derivaci $\mathbf{y}$ podle času.

$$\mathbf{y}' = f(t, \mathbf{y}) \tag{1.1}$$

Zpravidla nás nezajímá úplné řešení tohoto systému, nýbrž pouze směr, jakým se systém bude vyvíjet, pokud začne v určitých hodnotách $A \in \mathbb{R}^d$. Navíc často není třeba ani znát přesně vývoj, ale pouze jeho aproximaci.

$$\mathbf{y}(t_0) = A \tag{1.2}$$

Důležitým předpokladem pro řešení problému je, aby funkce f byla Lipschitzovsky spojitá [22]. Je-li dána vektorová norma $||\cdot||$ a $\lambda > 0$, funkci f nazveme Lipschitzovsky spojitou právě tehdy, když platí:

$$||f(t,\mathbf{x}) - f(t,\mathbf{y})|| \leq \lambda||\mathbf{x}-\mathbf{y}|| \quad \text{pro všechna} \quad \mathbf{x},\mathbf{y} \in \mathbb{R}^d, t \geq t_0 \tag{1.3}$$

Metody určené pro řešení těchto systémů se snaží najít dostatečně dobrou aproximaci trajektorie řešení v diskrétním čase. Dostatečně dobrou aproximací se zpravidla rozumí to, že chyba, s níž se vypočítané body liší od skutečného řešení, je shora ohraničená. Navíc lze chybu často nastavit v parametrech numerické metody. Jestliže je tedy zvolen časový krok h, požaduje se, aby platilo následující:

$$\mathbf{y}_i \sim \mathbf{y}(t_i) \quad \text{kde} \quad t_i = t_0 + i \cdot h, i \in \mathbb{N} \tag{1.4}$$

1.3 Algoritmus pro analýzu dynamický systémů

Sven Dražan ve své diplomové práci [7] prezentoval myšlenku metody pro analýzu dynamických systémů. Nejprve se specifikují vlastnosti trajektorií pomocí formule lineární a temporální logiky [19]. Poté pro daný dynamický systém a hranice iniciálních podmínek metoda identifikuje regiony, v nichž se nachází iniciální body trajektorií, které formuli splňují, a regiony, v nichž začínají trajektorie, pro které formule naopak neplatí.

Algoritmus pro své potřeby používá míry blízkosti trajektorií, ze které odvozuje, kdy a jak má zahušťovat prostor iniciálních podmínek tak, aby byla zřetelná hranice platnosti LTL formule.

1.3.1 Stěžejní části algoritmu

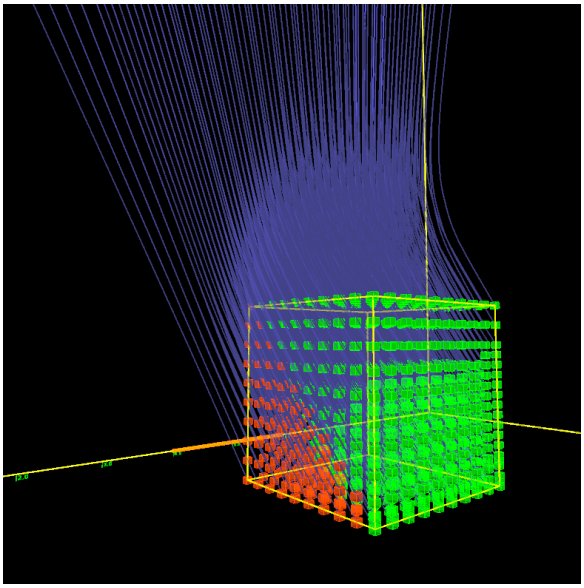
Stávající implementace algoritmu naráží na výkonnostní limity a je schopna pracovat pouze s relativně malými systémy. Proto vznikla potřeba akcelerovat některé části algoritmu. Jedná se o následující části:

- **numerická simulace** – algoritmus nejprve pomocí numerické simulace vygeneruje pro každý iniciální bod posloupnost bodů odpovídající trajektorii, po které se bude systém ubírat, pokud začne v tomto bodě;
- **ověření LTL vlastnosti nad trajektorií** – nad trajektoriemi se provede kontrola, zda platí daná formule, přičemž v případě formulí neobsa-

hujících operátor NEXT nejsou nutně potřeba všechny body trajektorie, nýbrž pouze body měnící platnost formule.

- **ověřování vzdálenosti mezi trajektoriemi** – zkontroluje se vzdálenost mezi body sousedních trajektorií a v případě příliš vzdálených trajektorií se prostor mezi odpovídajícími iniciálními body zahustí novým iniciálním bodem.

Tato práce se věnuje akceleraci numerické simulace na grafické kartě s cílem generovat více trajektorií paralelně. Je však vhodné poznamenat, že samotné generování trajektorií nestačí k významnému zrychlení celého algoritmu, protože ostatní části pracují s nagenerovanými body, kterých může být opravdu velké množství. Do budoucna je v plánu zaměřit se i na tyto části a paralelizovat je podobným způsobem jako generování trajektorií.



Obrázek 1.1: Krychle představuje prostor iniciálních podmínek, zelené body iniciální hodnoty, pro něž platí daná podmínka, a červené body hodnoty, pro něž podmínka neplatí [7, str. 38].

Kapitola 2

Metody numerické simulace

Jak již bylo řečeno, metody pro řešení problému iniciálních podmínek se snaží nalézt dostatečně přesnou aproximaci řešení v diskrétním čase, což lze zapsat pomocí vztahu 1.4.

Návrh numerických metod spočívá v uchopení rovnice 2.1 a nahrazení integrálu na pravé straně algoritmicky snadno spočitatelným výrazem.

$$\mathbf{y}(t_{n+1}) = \mathbf{y}(t_n) + \int_{t_n}^{t_{n+1}} f(\tau, \mathbf{y}(\tau))d\tau \tag{2.1}$$

Podle toho, zda tímto nahrazením dojdeme k rovnici obsahující na pravé straně proměnnou $\mathbf{y}(t_{n+1})$ či nikoliv, rozlišujeme metody na:

1. **explicitní** – rovnice na pravé straně neobsahuje proměnnou $\mathbf{y}(t_{n+1})$ a lze ji tedy řešit přímo;

2. **implicitní** – rovnice na pravé straně obsahuje proměnnou $\mathbf{y}(t_{n+1})$ a pro její řešení je nutné nejprve odhadnout hodnotu $\mathbf{y}(t_{n+1})$, a tu poté zpřesňovat.

Jelikož výpočet hodnoty $\mathbf{y}_{n+1}$ zpravidla následuje po výpočtech bodů $\mathbf{y}_1, \ldots, \mathbf{y}_n$, lze nahradit integrál složitějším výrazem používajícím více již spočítaných hodnot. Z tohoto důvodu dělíme numerické metody ještě podle tohoto kritéria na:

1. **jednokrokové** – těmto metodám se též někdy říká metody bez paměti;

2. **vícekrokové** – přesnost těchto metod zpravidla roste s počtem posledních hodnot použitých ke spočítání hodnoty nové.

2.1 Jednokroková schémata

Nejprve se budu věnovat jednokrokovým schémetům, na kterých bych rád ukázal rozdíl mezi implicitními a explicitními metodami. Tato skupina metod má nespornu výhodu v tom, že jsou snadné na implementaci a pochopení.

2.1.1 Eulerovo schéma

Základní myšlenka Eulerova schématu spočívá v tom, že je v rovnici 2.1 v intervalu $[t_n, t_{n+1}]$ zafixována hodnota $\mathbf{y}(\tau)$. Předpokladem tohoto postupu je, že časový krok $h = t_{n+1} - t_n$ je dostatečně malý na to, aby se hodnota $\mathbf{y}(\tau)$ významně změnila. Po zafixování je možné řešit podstatně jednodušší schéma 2.2.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h \cdot \mathbf{y}(\tau) \tag{2.2}$$

Způsobů, jak zvolit hodnotu $\mathbf{y}(\tau)$, je několik. V této práci ukáži tři z nich.

Eulerova explicitní metoda: Původní podoba Eulerovy metody fixuje hodnotu $\mathbf{y}(\tau)$ do stavu, v jakém je na začátku intervalu $[t_n, t_{n+1}]$. Jelikož je metoda explicitní, ke spočítání $\mathbf{y}_{n+1}$ stačí znát $\mathbf{y}_1, \ldots, \mathbf{y}_n$. Velikost chyby, s jakou lze tímto způsobem odhadnout řešení, se však pohybuje v řádu $\mathcal{O}(h^2)$ [12, str. 6], čímž se metoda stává pro praktické využití nepoužitelnou.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h \cdot \mathbf{y}_n \tag{2.3}$$

Eulerova implicitní metoda: Podobně jako v předchozím případě je možné zafixovat hodnotu $\mathbf{y}(\tau)$ do stavu, v jakém byla v krajním bodě intervalu $[t_n, t_{n+1}]$. Tentokrát je ovšem zvolen koncový stav.

Touto volbou se ovšem velice komplikuje postup, jak spočítat nový bod $\mathbf{y}_{n+1}$. Nestačí znát pouze předchozí body trajektorie $\mathbf{y}_1, \ldots, \mathbf{y}_n$, ale je již třeba hodnota samotného bodu $\mathbf{y}_{n+1}$. Pro tyto účely je dostačující aplikace např. Newton-Raphsonovy metody [24] a následné použití výsledku jako odhadu pro numerickou simulaci. Výsledek simulace lze použít jako nový odhad, proces zopakovat a zpřesnit tak výpočet $\mathbf{y}_{n+1}$.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h \cdot \mathbf{y}_{n+1} \tag{2.4}$$

Chyba Eulerovy implicitní metody se bohužel znovu pohybuje v řádu $\mathcal{O}(h^2)$ [5], nicméně během výpočtu vykazuje stabilnější chování, zejména u tzv. *tuhých rovnic* [12, str. 53].

Theta metoda: Předchozí metody lze zobecnit a to tak, že hodnotu $\mathbf{y}(\tau)$ nahradíme váženým průměrem hodnot, jakých by $\mathbf{y}(\tau)$ nabývalo v krajních bodech intervalu $[t_n, t_{n+1}]$.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h \cdot (\Theta \cdot \mathbf{y}_n + (1 - \Theta) \cdot \mathbf{y}_{n+1}) \tag{2.5}$$

V předpisu 2.5 za Θ volíme číslo z intervalu $[0, 1]$. Pokud za Θ zvolíme jinou hodnotu než 1, opět se jedná o implicitní metodu, což obnáší již popsané komplikace ve výpočtu. Pro obecné hodnoty Θ se chyba pohybuje opět v řádu $\mathcal{O}(h^2)$. Je-li však nastaveno $\Theta = \frac{1}{2}$[1], chyba se sníží na $\mathcal{O}(h^3)$ [8, str. 18].

2.1.2 Runge-Kuttovo explicitní schéma

Velkou skupinou explicitních metod, které se používají v praxi, jsou metody využívající Runge-Kuttova schématu 2.6. Tyto metody generují nový bod s chybou o velikosti $\mathcal{O}(h^{s+1})$, kde s je tzv. řád. Mezikroky výpočtu na sebe plynule navazují, a tudíž je výpočetní náročnost vzhledem k přesnosti metody malá. Sekce 2.3 ukazuje, jak se dá tato metoda jednoduše upravit tak, aby bylo možné kontrolovat chybu.

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h \cdot \sum_{i=1}^{s} b_i k_i \tag{2.6}$$

$$\begin{aligned} k_1 &= f(t_n, \mathbf{y}_n) \\ k_2 &= f(t_n + c_2 h, a_{2,1} h k_1) \\ k_3 &= f(t_n + c_3 h, a_{3,1} h k_1 + a_{3,2} h k_2) \\ &\vdots \\ k_s &= f(t_n + c_s h, a_{s,1} h k_1 + a_{s,2} h k_2 + \ldots a_{s,s-1} h k_{s-1}) \end{aligned} \tag{2.7}$$

Metoda používá pro aproximaci integrálu v rovnici 2.2 Gaussovu kvadraturu integrálu [12, str. 33]. Jednotlivé metody jsou popsány tzv. Butcherovým tablem 2.8, přičemž aby metoda byla konzistentní, musí platit vztah

1. Pokud $\Theta = \frac{1}{2}$, říkáme že se jedná o tzv. lichoběžníkové pravidlo.

2.9. Navíc aby metoda dosáhla vhodného omezení lokální chyby, musí platit další omezující podmínky.

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{2,1} & & & & \\ c_3 & a_{3,1} & a_{3,2} & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_s & a_{n,1} & a_{n,2} & \cdots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \cdots & b_{s-1} & b_s \end{array} \tag{2.8}$$

$$\sum_{j=1}^{i-1} a_{i,j} = c_i \quad \text{pro} \quad i = 2, \ldots, s. \tag{2.9}$$

V různých zdrojích [8] se nejčastěji uvádí příklad Runge-Kuttovy explicitní metody řádu 4 danou tablem 2.10.

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \tag{2.10}$$

2.2 Vícekroková schémata

Typická numerická simulace probíhá tak, že dostane počáteční body $\mathbf{y}_0$ a časové kvantum h. Simulace postupně produkuje body $\mathbf{y}_1$, $\mathbf{y}_2$, $\mathbf{y}_3$ atd. Nabízí se otázka, zda ve chvíli, kdy se počítá bod $\mathbf{y}_{n+1}$, nelze nějak zužitkovat body, které se počítaly dříve.

V této kapitole bych rád ukázal příklad explicitní metody, která se snaží předchozí výpočty využít.

2.2.1 Adams-Bashforthovo explicitní schéma

Rovnice 2.11 znovu připomíná problém, který musí algoritmus pro numerickou simulaci vyřešit. Dříve popsané algoritmy k rovnici přistupovaly tak, že nahradily integrál na pravé straně jednodušším výrazem.

$$\mathbf{y}(t_{n+s}) = \mathbf{y}(t_{n+s-1}) + \int_{t_{n+s-1}}^{t_{n+s}} f(\tau, \mathbf{y}(\tau))d\tau \tag{2.11}$$

Adams-Bashforthovo schéma se neomezuje na to, že by prohlásilo $\mathbf{y}(\tau)$ v průběhu intervalu za konstantní, ale sestrojí funkci, která se během intervalu $[t_{n+s-1}, t_{n+s}]$ chová podobně jako funkce f.

$$\begin{aligned} \mathbf{p}(t) &= \sum_{m=0}^{s-1} p_m(t) f(t_{n+m}, \mathbf{y}_{n+m}), \\ p_m(t) &= \prod_{l=0, l\neq m}^{s-1} \frac{t - t_{n-l}}{t_{n+m} - t_{n-l}} \\ &= \frac{(-1)^{s-1-m}}{m!(s-1-m)!} \prod_{l=0, l\neq m}^{s-1} \left(\frac{t - t_n}{h} - l \right) \end{aligned} \tag{2.12}$$

Metoda neprve za pomocí bodů $\mathbf{y}_n$, …, $\mathbf{y}_{n+s-1}$ zkonstruuje polynom $\mathbf{p}$ podle předpisu 2.12. Tento polynom poslouží jako vhodná aproximace funkce f během časového intervalu $[t_{n+s-1}, t_{n+s}]$ a použije se v rovnici 2.14 [12, str. 19].

$$\mathbf{p}(t) = f(t, \mathbf{y}(t)) + \mathcal{O}(h^s) \quad \text{pro} \quad t \in [t_{n+s-1}, t_{n+s}] \tag{2.13}$$

Adams-Bashforthova metoda dosahuje chyby řádu $\mathcal{O}(h^{s+1})$, na druhou stranu však klade vyšší nároky na paměť.

$$\begin{aligned} \mathbf{y}_{n+s} &= \mathbf{y}_n + h \cdot \sum_{m=0}^{s-1} b_m f(t_{n+m}, \mathbf{y}_{n+m}), \\ b_m &= h^{-1} \int_{t_{n+s-1}}^{t_{n+s}} p_m(\tau) d\tau \\ &= h^{-1} \int_0^h p_m(t_{n+s-1} + \tau) d\tau \end{aligned} \tag{2.14}$$

2.3 Kontrola chyby

Každá metoda pro řešení problému iniciálních podmínek je charakteristická řádem chyby, s jakou dokáže vygenerovat bod $\mathbf{y}_{n+1}$, jestliže je známa

přesná hodnota bodu $\mathbf{y}(t_n)$. Konkrétní hodnota chyby se může lišit v závislosti na systému rovnic, který zrovna metoda řeší.

Chyba aproximace jednoho bodu se samozřejmě projeví při výpočtu bodů následujících. Probíhá-li tedy simulace obsahující tisíce a milióny bodů, může chyba narůst do nepříjemných rozměrů. Jistým řešením je nastavení kroku h na opravdu malou hodnotu, ale to zvýší výpočetní náročnost simulace.

Naštěstí existuje způsob, jak během výpočtu odhadnout chybu a příslušným způsobem přizpůsobit krok h.

2.3.1 Obecný způsob odhadu chyby

Nechť je dán časový krok h a bod $\mathbf{y}_n$. Hodnota $\mathbf{y}_{n+1}^{(0)}$ je spočítána pomocí Eulerovy explicitní metody.

$$\mathbf{y}_{n+1}^{(0)} = \mathbf{y}_n + hf(t_n, \mathbf{y}_n) \tag{2.15}$$

Výraz 2.16 ukazuje, jakým způsobem se $\mathbf{y}_{n+1}^{(0)}$ liší od skutečného řešení $\mathbf{y}(t_{n+1})$.

$$\begin{aligned} \tau_{n+1}^{(0)} &= ch^2 \\ y_{n+1}^{(0)} + \tau_{n+1}^{(0)} &= y(t_{n+1}) \end{aligned} \tag{2.16}$$

Aproximaci bodu $\mathbf{y}(t_{n+1})$ lze spočítat znovu a přesněji, označme ji $\mathbf{y}_{n+1}^{(1)}$. Výpočet se provede dvakrát o časový krok $\frac{h}{2}$. Vznikne nejprve hodnota $\mathbf{y}_{n+\frac{1}{2}}^{(1)}$, a poté $\mathbf{y}_{n+1}^{(1)}$, kterou lze znovu porovnat se skutečným řešením $\mathbf{y}(t_{n+1})$. Nyní předpokládejme, že hodnota c, která vystupuje ve výpočtu chyb $\tau_{n+1}^{(0)}$ a $\tau_{n+1}^{(1)}$, se pro dostatečně malé hodnoty h chová jako konstanta.

$$\begin{aligned} y_{n+\frac{1}{2}} &= y_n + \frac{h}{2} f\left(t_n, y_n\right) \\ y_{n+1}^{(1)} &= y_{n+\frac{1}{2}} + \frac{h}{2} f\left(t_{n+\frac{1}{2}}, y_{n+\frac{1}{2}}\right) \\ \tau_{n+1}^{(1)} &= c\left(\frac{h}{2}\right)^2 + c\left(\frac{h}{2}\right)^2 \\ &= 2c\left(\frac{h}{2}\right)^2 = \frac{1}{2}ch^2 = \frac{1}{2}\tau_{n+1}^{(0)} \end{aligned} \tag{2.17}$$

Pro odhad velikosti chyby se využije vztahu 2.18. Odhad se porovná s maximální a minimální očekávanou chybou, a pokud je třeba, upraví se

odpovídajícím způsobem časový krok h. Stejný postup lze použít pro každou metodu řešící problém výchozích podmínek.

$$y_{n+1}^{(1)} - y_{n+1}^{(0)} = \tau_{n+1}^{(1)} \tag{2.18}$$

2.3.2 Runge-Kutta-Fehlbergova metoda

Předchozí řešení v sobě obsahuje jednu velkou nevýhodu. Pro odhad chyby, musí numerická simulace vypočítat jeden bod dvakrát a jednou dokonce pomocí mezikroku. Místo jednoho výpočtu jsou k získání bodu $\mathbf{y}_{n+1}$ potřeba výpočty tři.

Při odhadu chyby u Runge-Kuttovy metody se množství provedených výpočtů nezmění. Využívá se zde toho, že lze najít dvě metody po sobě jdoucích řádů takové, že se jejich výpočet bodu $\mathbf{y}_{n+1}$ liší pouze v jednom kroku. Jinými slovy, velkou část výpočtu metody řádu n lze použít jako mezivýpočet pro metodu řádu $n + 1$.

Schéma využívající toto pozorování se nazývá Runge-Kutta-Fehlbergovo [9, str. 497] a je popsáno vztahem 2.19. Proměnná $\mathbf{y}_{n+1}^{(o)}$ značí bod vypočítaný Runge-Kuttovou metodou řádu o, podobně $\tau_{n+1}^{(o)}$ příslušnou chybu a $b_i^{(o)}$ příslušné kostanty.

$$\begin{aligned} \mathbf{y}_{n+1}^{(s-1)} &= \mathbf{y}_n + h \cdot \sum_{i=1}^{s-1} b_i^{(s-1)} k_i \\ \mathbf{y}_{n+1}^{(s)} &= \mathbf{y}_n + h \cdot \sum_{i=1}^{s} b_i^{(s)} k_i \\ \tau_{n+1} &= \mathbf{y}_{n+1}^{(s)} - \mathbf{y}_{n+1}^{(s-1)} \end{aligned} \tag{2.19}$$

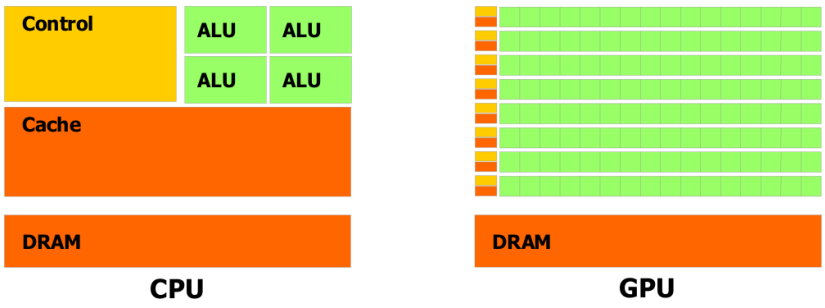
Získání nového bodu a odhadu chyby samozřejmě předchazí spočítání koeficientů k_1 až k_s, které jsou však pro obě metody společné.

Kapitola 3

Technologie CUDA

CUDA, neboli *Compute Unified Device Architecture*[14], je technologie vyvinutá společností NVIDA pro práci s grafickými kartami. Grafické karty nově nabízí alternativu ke klasickým procesorům v oblasti supervýpočtů a obecně akceleraci algoritmů. Jejich zvládnutí však na druhou stranu vyžaduje netriviální úsilí a znalost jejich základní stavby a fungování. Na rozdíl od procesoru totiž sama grafická karta neprovádí příliš mnoho optimalizačních kroků[1], a programátor má proto větší zodpovědnost za svůj kód.

Zjednodušený návrh výrobcům grafickýh karet umožňuje na podobný prostor a za podobné finance vložit více výpočetních jednotek. Výpočetní jednotky jsou na kartě sdružovány do multiprocesorů, ke kterým jsou přiřazeny řídící jednotky a vyrovnávací paměť. Tím se karty liší od klasických procesorů, kde se tyto komponenty nacházejí u každé výpočetní jednotky.

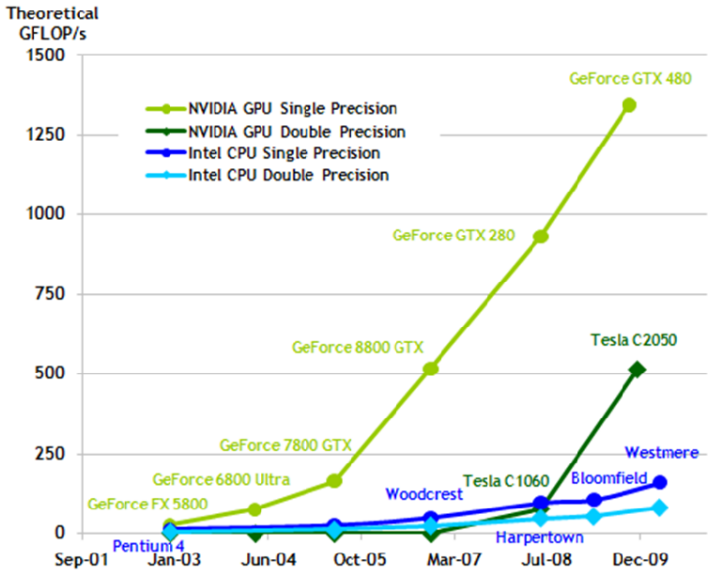


Obrázek 3.1: Porovnání architektury CPU a GPU[14, str. 3].

Moorův zákon [23, 13] dříve platil zejména díky zvyšování frekvence a instrukčnímu paralelismu, dnes se však spíše uplatňuje zmnožování jader a vekterové instrukce. To mimo jiné znamená, že dříve se každých 18 měsíců zdvojnásobila rychlost zpracování jednoho vlákna, avšak dnes se

1. Klasický procesor je například schopen přeskládat instrukce tak, aby zrychlil jejich provedení.

každých 18 měsíců zdvojnásobí rychlost zpracování **dostatečného množství** vláken. Postupně se tedy zrychlují ty výpočty, které jsou dostatečně velké na to, aby je šlo masivně paralelizovat. Dříve ke zrychlení programu programátorovi pomohl překladač, nyní se vyžaduje, aby sám programátor nalezl v řešení problému souběžnost. Je však vhodné poznamenat, že souběžnost sama o sobě není řešením, protože existuje mnoho problémů, pro něž nelze sestrojit paralelní algoritmus, nebo které jsou pro masivní paralelizaci příliš malé.



Obrázek 3.2: Porovnání počtu operací s plovoucí desetinnou čárkou za sekundu pro CPU a GPU [14, str. 2].

Jelikož se programovací model CUDA značně liší od klasického modelu, rád bych zde prezentoval některé jeho stěžejní části. Nejpoužívanějším nástrojem pro psaní programů na bázi technologie CUDA je rozšíření programovacího jazyka C nazývající se CUDA C, proto se v příkladech omezím právě na tento jazyk.

3.1 Hierarchie vláken

Vlákna jsou během výpočtu rozdělena do dvourozměrných bloků, v rámci nichž nesou identifikátor v podobě dvojice `threadIdx.x` a `threadIdx.y`, která představuje souřadnici vlákna v rámci bloku. Výpočet se rozdělí mezi

multiprocesory tak, aby jeden blok náležel právě jednomu multiprocesoru.

Bloky jsou uspořádány v mřížce, v níž je každý označen unikátní dvojicí souřadnic `blockIdx.x` a `blockIdx.y`. Do budoucna se počítá s tím, že bloky i mřížky budou moci být až trojrozměrné, a proto existují i proměnné `threadIdx.z` a `blockIdx.z`, nicméně v současnosti tyto proměnné nemohou nabývat jiné hodnoty než 1.

Vlákna jsou dále členěna do tzv. warpů, jejichž velikost je zpravidla nastavena na 32. V rámci jednoho warpu je nutné, aby vlákna prováděla v jeden čas stejnou instrukci. Pokud výpočet diverguje a není možné tuto podmínku dodržet, některá jádra multiprocesoru běží na prázdno a prodlužuje se celkový čas výpočtu.

3.2 Kernely

Programy, které se spouští na grafické kartě, se nazývají kernely. Kernel je běžná funkce označená deklarací `__global__` a jeho spuštění na grafické kartě se provádí pomocí uvozovacích znaků `<<<` a `>>>`, do nichž se vepisují parametery udávající rozměr bloků a mřížky. Po spuštění je každému vláknu a bloku přiřazena příslušná souřadnice.

Ilustrační kód ve výpisu 3.1 [14, str. 7] ukazuje jednoduchý kernel pro sčítání vektorů a jeho spuštění v hlavní funkci. Kernel spustí jeden blok obsahující N vláken seřazených v dimenzi x. Každému vláknu odpovídá jeden index ve sčítaných polích.

```
1  // Kernel definition
2  __global__ void VecAdd(float* A, float* B, float* C) {
3    int i = threadIdx.x;
4    C[i] = A[i] + B[i];
5  }
6
7  int main() {
8    ...
9    // Kernel invocation with N threads
10   VecAdd<<<1, N>>>(A, B, C);
11 }
```

Výpis 3.1: Sčítání dvou vektorů A a B.

3.3 Paměť

Při používání karty pro výpočty je třeba si uvědomit, že data, se kterými má karta pracovat, nejsou dostupná přímo a musí se přesunout přes PCI-Express sběrnici. Vzhledem k rychlosti sběrnice a tomu, že přes sběrnici může v daném čase komunikovat více zařízení se přenos dat může velice snadno stát úzkým hrdlem algoritmu. Obecně platí, že je nutné nad daty provádět netriviální výpočet, případně přenos dat překrýt s výpočtem.

Další věcí, na kterou je třeba si dát pozor, je velké množství druhů pamětí s různými přístupovými rychlostmi. I přes malou vyrovnávací paměť je však prostředí schopno rychlost pamětí částečně maskovat. Řídící jednotka v případě warpu, jehož data ještě nejsou k dispozici, požádá danou paměť o data a warp odloží na později, přičemž výpočetní čas přenechá jinému warpu.

3.3.1 Druhy pamětí

Registry: Nejrychlejší a nejmenší druh paměti, ukládají se sem lokální proměnné definované v rámci vlákna, proto je životnost obsahu registrů spojená s vláknem.

Lokální paměť: Velice pomalá paměť, nachází se zde lokální proměnné, které se nevešly do registrů. Při odsunutí výpočtu vlákna, data zanikají.

Sdílená paměť: Poměrně rychlá, ale docela malá paměť, jejíž obsah je spjat s blokem. Proměnné jejichž obsah se má uložit do sdílené paměti se v kódu označí deklarací `__shared__`. Přístup do této paměti je limitován přes tzv. paměťové banky [14, str. 88]. Velikost použité sdílené paměti je třeba znát před spuštěním kernelu.

Globální paměť: Poměrně velká paměť s latencí až stovky cyklů. Tato paměť zpravidla slouží pro kopírování z a do operační paměti, která je přístupná z procesoru.

Paměť konstant: Rychlá paměť určená pro data, která se během spuštění kernelu nemění.

Paměť textur: Paměť určená pouze pro čtení, v níž se nachází data známá před spuštěním kernelu. Zejména je vhodná pro data strukturovaná v dvou-

registry	32 000 na jeden multiprocesor
lokální paměť	512 KB na jedno vlákno
sdílená paměť	48 KB na jeden multiprocesor
globální paměť	asi 1.5 GB
paměť konstant	64 KB
paměť textut	maximální rozměry 65536 $\times$ 65535, případně 2048 $\times$ 2048 $\times$ 2048, maximálně lze v kernelu použít 128 textur

Tabulka 3.1: Parametry karty GeForce GTX 480, se kterou jsem pracoval [21].

rozměrné mřížce a lze nastavit, jak se má chovat ve chvíli, kdy se přistupuje k nekorektním souřadnicím, např. $(0.5, 0.3)$ nebo $(-10, 0)$.

3.4 Synchronizace

Díky podpoře masivní paralelizace nabízí architektura CUDA i konstrukce pro jednoduchou synchronizaci. Pomocí příkazu `__syncthreads()` programátorům umožňuje synchronizovat vlákna napříč blokem a pomocí dalších příkazů napříč warpem. Vlákna v jednom warpu se navíc nemusí jen jednoduše synchronizovat, ale mohou mezi sebou i hlasovat.

Synchronizace zde narozdíl od klasického procesoru nestojí více než jednu instrukci. Na druhou stranu se zde stále nachází prodleva způsobená čekáním vláken, která měla méně práce, na vlákna vytíženější.

Současné karty přímo nepodporují synchronizaci celého kernelu, ani obecně n vláken. Pokud se v algoritmu objeví potřeba synchronizovat vlákna napříč celým kernelem, nezbývá nic jiného než kernel vypnout a znovu pustit. To v sobě ale bohužel zahrnuje režii se zaváděním kernelu na grafickou kartu a inicializací lokálních proměnných.

Pokud je potřeba synchronizovat n vláken v rámci jednoho bloku, lze tak učinit za použití sdílené proměnné a atomických operací nad celými čísly. Synchronizovat vlákna z různých bloků lze jedině přes globální paměť, což vzhledem k vysoké latenci činí tuto operaci téměř nepoužitelnou.

Kapitola 4

Implementace

V této kapitole popíši implementaci knihovny pro řešení problému iniciálních podmínek používájící již popsané metody a technologie s cílem akcelerovat algoritmus pro analýzu biologických systémů [7].

4.1 Přehled

Původní aplikace algoritmu je implementována v jazyce Java, a proto bylo žádoucí, aby vytvořená knihovna poskytovala rozhraní právě v tomto jazyce. Zároveň je ale dobré poznamenat, že rozšíření CUDA je dostupné především pro nízkoúrovňové jazyky, protože je například nutné přímo pracovat s pamětí. Z tohoto důvodu není možné pro kartu programovat v jazyce Java, a část knihovny je tedy implementovaná v jazyce CUDA C.

Rozhraní mezi Javou a kernely tvoří knihovna JCuda [1], která zpřístupňuje tzv. *driver a runtime API* [14, str. 15] a umožňuje kernely spustit z prostředí Javy. Pro manipulaci s projektem jsou použity programy `ant` a `make` a pro testování knihovna JUnit [2].

Numerické metody jsou dostupné jak ve verzi pro procesor, tak ve verzi pro grafickou kartu. Dostupné metody jsou:

- **Euler** – explicitní metoda popsaná v sekci 2.1.1, neobsahuje v sobě kontrolu chyby;
- **Runge-Kutta-Fehlberg** – explicitní metoda popsaná v sekci 2.3.2 s chybou v jednom kroku řádu $\mathcal{O}(h^6)$, obsahuje v sobě mechanismus pro kontrolu chyby.

4.1.1 Javové rozhraní

Zdrojové kódy rozhraní v jazyce Java se nachází v adresáři `java` a jsou děleny do následujících balíčků:

- `org.sybila.ode` – třídy pro práci se systémy diferenciálních rovnic, definice rozhraní pro spouštění numerické simulace a základní práci s trajektoriemi a body trejektorie;
- `org.sybila.ode.cpu` – třídy umožňující výpočet numerické simulace na klasickém procesoru, tyto třídy slouží pouze pro srovnání výsledků měření;
- `org.sybila.ode.cuda` – třídy tvořící rozhraní pro výpočet numerické simulace na grafické kartě;
- `org.sybila.ode.benchmark` – základní nástroje pro měření výkonu numerické simulace.

Třída `org.sybila.ode.benchmark.Main` je jedinou spustitelnou třídou projektu a nabízí základní měření výkonu dostupných tříd provádějících numerickou simulaci. Aby se program spustil korektně, musí být na stroji dostupná grafická karta s *compute capabality* alespoň 2.0 [14, str. 153].

4.1.2 Kernely

V souboru `c/src/num_sim_kernel.cu` se nachází zdrojové kódy kernelů pro Eulerovu a Runge-Kutta-Fehlbergovu metodu. Eulerova metoda je naimplementovaná dvěma způsoby v kernelech `euler_simple_kernel` a `euler_kernel`, které budou podrobněji popsány v sekci 4.2. Runge-Kutta-Fehlbergova metoda je naimplementovaná pouze jedním způsobem v kernelu `rkf45_kernel`.

Všechny kernely přijímají stejné parametry, skrze něž je nutné definovat počáteční podmínky, nastavení absolutní a relativní chyby a systém diferenciálních rovnic.

4.1.3 Reprezentace systému diferenciálních rovnic

Aby bylo možné provést numerickou simulaci, je nutné reprezentovat systém diferenciálních rovnic. Ve své práci jsem se omezil na systémy obsahující pouze rovnice tvaru 4.1 a předpokládám, že pro každou proměnnou $x_m \in \{x_0, x_1, \ldots x_n\}$ existuje právě jedna rovnice s x'_m na levé straně. Tento formát je dostačující pro popis reakčních systémů obsahující elementární chemické reakce.

$$\begin{gathered} x'_m = a_1 \cdot F_1 + a_2 \cdot F_2 + \ldots + a_i \cdot F_i, \\ \text{kde } F_j \text{ je součin proměnných } x_k \in \{x_0, x_1, \ldots, x_n\} \end{gathered} \tag{4.1}$$

Reprezentaci rovnic tohoto zjednodušeného tvaru lze v budoucnu snadno rozšířit na rovnice tvaru 4.2 popisující obecnější systémy s dynamikou popsanou lomenými funkcemi.

$$x'_m = \frac{a_1 \cdot F_1 + \ldots + a_i \cdot F_i}{b_1 \cdot G_1 + \ldots + b_j \cdot G_j}, \tag{4.2}$$

kde F_j a G_k jsou součiny proměnných $x_k \in \{x_0, x_1, \ldots, x_n\}$

Příkladem systému rovnic odpovídajících tvaru 4.1 je systém 4.3.

$$\begin{aligned} x'_0 &= 0.1 \cdot x_0 \cdot x_2 - 0.005 \cdot x_1 \\ x'_1 &= 0.1 \cdot x_1 - 0.2 \cdot x_0 \\ x'_2 &= 2 \end{aligned} \tag{4.3}$$

Systém rovnic je kódován pomocí následujících čtyř jednorozměrných polí, d označuje počet proměnných nacházejících se v systému:

- `coefficients` – koeficienty a_i pro každou rovnici systému;
- `coefficient_indexes` – obsahuje $d+1$ prvků, i-tý prvek pole v sobě nese odkaz na koeficienty rovnice pro i-tou proměnnou, na pozici $d+1$ se nachází počet koeficientů celkem;
- `factors` – indexy proměnných nacházející se v součinech F_k;
- `factor_indexes` – pro každý koeficent obsahuje odkaz na činitele s ním spojené, na pozici `coefficient_indexes[d+1]` se nachází počet činitelů celkem.

```
1 coefficients        = [0.1, -0.005, 0.1, -0.2, 2]
2 coefficient_indexes = [0, 2, 4, 5]
3 factors             = [0, 2, 1, 1, 0]
4 factor_indexes      = [0, 2, 3, 4, 5, 5]
```

Výpis 4.1: Zakódování systému rovnic 4.3.

Takto zakódovaný systém diferenciálních rovnic je vstupem metody numerické simulace, která jej používá při volání funkce pro výpočet x'_n. Tento výpočet je popsán algoritmem 1. Ve stávající implementaci se data kódující systém rovnic nacházejí v globální paměti. Učinil jsem tak proto, že překopírování dat do sdílené paměti nepřineslo zrychlení. Latence globální paměti se v porovnání se sdílenou pamětí neprojevuje a to pravděpodobně

díky velké vyrovnávací paměti, jíž disponuje karta, kterou jsem měl k dispozici. Další možností je přesun dat před výpočtem do paměti konstant. Zde jsem však narazil na omezení knihovny JCuda, která nenabízí příliš dobré rozhraní pro práci s ukazateli.

Algoritmus 1 Výpočet x'_n pro bod $(x_0, \ldots, x_d)$

```
1: result ← 0
2: for c ← coefficient_indexes[n] to coefficient_indexes[n + 1] − 1 do
3:    aux_result ← coefficients[i]
4:    for f ← factor_indexes[c] to factor_indexes[c + 1] − 1 do
5:       aux_result ← aux_result · x[factors[f]]
6:    end for
7:    result ← result + aux_result
8: end for
9: return result
```

4.2 Popis výpočtu

Pro docílení dostatečného maskování latence je nutné, aby výpočet na grafické kartě provádělo dostatečné množství vláken. Stávající verze implementace algoritmu pro analýzu dynamických systémů počítá pro jeden systém řádově stovky trajektorií. V případě kernelu, který by každé trajektorii přidělil jedno vlákno, by tedy počet vláken nebyl pro zamaskování latence dostatečný. Pro testovací účely jsem však vytvořil i jeden kernel používající toto rozdělení práce.

Druhé rozdělení práce mezi vlákna, které se nabízí, je po dimenzích. To znamená, že jednu trajektorii má na starosti tolik vláken, kolik proměnných obsahuje systém rovnic. Tato vlákna se musí mezi sebou synchronizovat minimálně po výpočtu jednoho bodu, protože pro výpočet bodu nového je potřeba hodnota jeho předchůdce.

Při implementaci jsem vycházel z předpokladu, že počet proměnných bude *rozumný*, a tudíž se všechna vlákna počítající jednu trajektorii vejdou do jednoho bloku, přičemž lze počet proměnných dokonce použít jako jeden jeho rozměr. V jednom bloku se tedy nachází vlákna počítající několik trajektorií. Učinil jsem tak na základě modelů, pro které byla metoda pro analýzu dynamických systémů navržena a které obsahují nejvýše desítky proměnných.

Nabízí se dvě možnosti, jak synchronizovat vlákna jedné trajektorie:

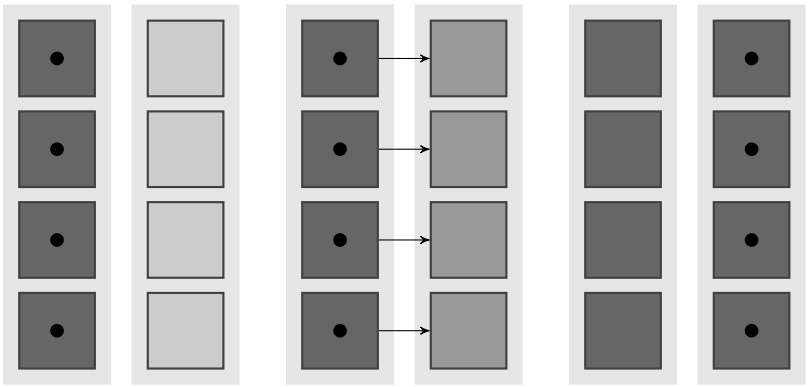
1. **pomocí sdílené proměnné** – je potřeba před výpočtem nastavit veli-

kost sdílené paměti a pomocí atomických operací provést tzv. bariéru;

2. **pomocí synchronizace celého bloku** – výpočty trajektorií, které se nachází v paměti blízko sebe, budou synchronní.

Z důvodu snazší implementace jsem se rozhodl pro druhou variantu. Navíc se domnívám, že trajektorie nacházející se ve stejném bloku se od sebe pravděpodobně moc neliší, a proto i výpočet vláken nebude navzájem příliš divergovat.

Výpočet na grafické kartě je ve velké míře limitován velikostí paměti. Například karta GeForce GTX 480, kterou jsem měl během implementace k dispozici, obsahuje globální pamět o velikosti 1,48 GB. Z tohoto důvodu se musí numerická simulace spouštět po menších blocích. Úzkým hrdlem není jen globální paměť, ale například i velikost sdílené paměti nebo počet registrů. Pro snížení velikosti lokálních proměnných se velká část výpočtu provádí nad proměnnými, které jsou umístěny v globální paměti a ve kterých se na konci výpočtu nachází výsledná data.



(a) Před výpočtem nového bodu

(b) Kopírování hodnoty posledního bodu do prostoru nového bodu

(c) Výpočet nového bodu

Obrázek 4.1: Fáze výpočtu.

Kapitola 5

Evaluace

Výkon knihovny jsem měřil na stroji, který vystupuje v síti `fi.muni.cz` pod jménem `pandora` a má následující konfiguraci:

> CPU AMD Phenom(tm) II X4 940 Processor, 7 GB main memory, GeForce GTX 480. Debian GNU/Linux lenny with 2.6.26-1-amd64 kernel, CUDA toolkit 3.1, GCC version 4.3.2, OpenJDK 64-Bit Server version 1.6.0_0-b11.

Pro měření výkonu je k dispozici balík `org.sybila.ode.benchmark`, kde se nachází třída `Benchmark` schopná provést základní měření a spustitelná třída `Main`, ve které se nastavují základní parametry měření. Virtuálnímu stroji Javy byla navýšena velikost haldy na 6144 MB.

Výpočet numerické simulace se spustí n-krát a u každého se ověří, jak dlouho se simulace provádí. Nejlepší a nejhorší čas se zahodí a ze zbylých časů se provede aritmetický průměr. V této práci jsem zvolil $n = 10$.

Metody, které se objevují v měření:

- **CPU Eulerova metoda** – implementace Eulerovy metody ze sekce 2.1.1 v Javě bez adaptivního časového kroku;
- **CPU Runge-Kutta-Fehlbergova metoda** – implementace Runge-Kutta-Fehlbergovy metody ze sekce 2.3.2 v Javě;
- **CUDA jednoduchá Eulerova metoda** – implementace Eulerovy metody používající architekturu CUDA, kde každé trajektorii přísluší právě jedno vlákno;
- **CUDA Eulerova metoda** – implementace Eulerovy metody používající architekturu CUDA, kde každé proměnné jedné trajektorie přísluší jedno vlákno;
- **CUDA Runge-Kutta-Fehlbergova metoda** – implementace Runge-Kutta-Fehlbergovy metody používající architekturu CUDA, kde každé proměnné jedné trajektorie přísluší jedno vlákno.

5.1 Modely

Výkon numerické simulace závisí na vlastnostech modelu, který se právě počítá. Pro změření výkonu jsem zvolil několik modelů, mezi nimiž se nachází smyšlené modely, které lze snadno rozšířit o další proměnné, i reálný model s fixním počtem proměnných.

5.1.1 Jednoduchý lineární model

Prvním modelem, na kterém jsem zkoušel otestovat knihovnu byla jednoduchá soustava rovnic tvaru 5.1. Tento typ by měl být pro numerickou simulaci počítanou na grafické kartě nejvhodnější, protože výpočty pro jednotlivé proměnné jsou na sobě nezávislé a neměly by divergovat.

$$\begin{aligned} x'_0 &= a \cdot x_0 + a \cdot x_0 \\ x'_1 &= 2 \cdot a \cdot x_1 + 2 \cdot a \cdot x_1 \\ &\vdots \\ x'_{n-1} &= n \cdot a \cdot x_{n-1} + n \cdot a \cdot x_{n-1}, \\ &\text{kde } a = 0.005 \end{aligned} \tag{5.1}$$

Nepříliš intuitivní tvar rovnic tohoto modelu je zvolen proto, aby jeho vyhodnocení bylo srovnatelné s následujícím modelem.

5.1.2 Jednoduchý provázaný model

Tento model by měl ukázat, jak se bude výpočet chovat, když hodnota jedné proměnné závisí na jiné, ale zároveň se stále dá jednoduše vytvořit pro libovolný počet proměnných.

$$\begin{aligned} x'_0 &= a \cdot x_0 \cdot -n \cdot a \cdot x_1 \\ x'_1 &= 2 \cdot a \cdot x_1 - (n-1) \cdot x_2 \\ &\vdots \\ x'_{n-1} &= n \cdot a \cdot x_{n-1} - a \cdot x_0, \\ &\text{kde } a = 0.1 \end{aligned} \tag{5.2}$$

5.1.3 Reálný model

Posledním pro měření uvažovaným modelem je oscilující systém tří proměnných [4].

$$
\begin{aligned}
x' &= 0.0005 - 250 \cdot x \cdot y \\
y' &= 0.0001 - 0.1 \cdot y - 250 \cdot x \cdot y + 300 \cdot y \cdot z \\
z' &= 250 \cdot x \cdot y - 300 \cdot y \cdot z
\end{aligned}
\tag{5.3}
$$

5.2 Paremetry

Při měření jsem uvažoval různé parametry, které by mohly ovlivnit délku výpočtu. Jedná se o:

- **počet iniciálních bodů** – ovlivňuje počet vláken, které se zúčastní výpočtu a velikost alokované paměti;
- **počet proměnných v systému** – opět ovlivňuje počet vláken a velikost alokované paměti, navíc kromě zjednodušené Eulerovy metody určuje velikost bloků;
- **velikost relativní chyby** – opět u metod s adaptivním časovým krokem ovlivňuje délku výpočtu.

Velikost absolutní chyby má na způsob výpočtu podobný vliv jako velikost chyby relativní. Původní metoda pro analýzu dynamických systémů je zaměřena na biologické systémy, ve kterých se mohou hodnoty jednotlivých proměnných lišit až řádově a absolutní chyba zde nemusí hrát příliš velkou roli. Proto jsem zvolil z těchto dvou možností jako parametr chybu relativní.

Jelikož se výpočet na procesoru a grafické kartě může lišit v přesnosti a implementace využívající Eulerova schématu neobsahují kontrolu chyby vůbec, je kontrola relativní chyby ve většině zde prezentovaných měření vypnuta. Výjimkou je měření sledující dopad kontroly chyby na výkon simulace.

Maximální zkoušený počet iniciálních bodů byl zvolen na základě velikosti globální paměti dostupné grafické karty a počet proměnných podle očekávané velikosti simulovaných modelů metody pro analýzu dynamických systémů. Počet vrácených bodů trajektorií je nastaven na 1000.

5.3 Měření

Grafy zachycující měření jsou k dispozici v příloze A. Celkově jsou implementace využívající technologii CUDA oproti implementacím v Javě řádově rychlejší.

5.3.1 Vliv počtu iniciálních bodů a proměnných

Čas potřebný k výpočtu kernelů s rostoucím počtem iniciálních bodů narůstá lineárně. To samé lze říci o závislosti na počtu proměnných v systému. V případě reálného systému tří proměnných lze rozpoznat jistý „schodovitý“ trend, jenž je možná způsoben tím, že při určitém počtu trajektorií se grafická karta využívá optimálně a po navýšení tohoto počtu jsou některá vlákna „navíc“, což způsobí, že grafická karta není chvíli zcela využita. U modelů s více proměnnými k tomuto jevu nedochází, avšak místy se objeví lokální výkyvy, při nichž se u určitého počtu iniciálních bodů významně prodlouží naměřený čas, který se po zvýšení tohoto počtu opět „srovná“. Přiznám se, že tyto výkyvy nejsem schopen vysvětlit a připisuji je na vrub nepřesnosti měření.

5.3.2 Rozdělení práce

Měření ukazuje, že kernel, v němž každou trajektorii počítá právě jedno vlákno, je pomalejší než implementace, ve které každé trajektorii náleží tolik vláken, kolik se nachází proměnných v systému. Ačkoliv jsem očekával, že jednodušší implementace Eulerovy metody bude vykazovat lepší chování při větším počtu simulací, nebylo tomu tak. Pravděpodobně se projevuje velké množství dat, se kterým vlákno, které má na starosti celou trajektorii, pracuje. V takovém případě nemusí být efektivně použita vyrovnávací paměť.

5.3.3 Vliv nastavení relativní chyby

Největší vliv na délku výpočtu má, zdá se, nevhodná volba modelu společně s vysokými požadavky na přesnost vrácených bodů[1]. Kernely mají nastaven limit na maximální počet iterací, které mohou provést. Při snížení povolené relativní chyby mohou vlákna narazit na tento limit a v krajním případě vrátit menší počet bodů, než je požadováno. Výpočet vláken se samozřejmě liší podle toho, jaký iniciální bod jim přísluší. Celkově tedy nastavení chyby velkou měrou ovlivňuje divergenci výpočtu vláken.

1. To se samozřejmě týká pouze kernelu Runge-Kutta-Fehlbergovy metody. U implementací Eulerovy metody ke kontrole chyby nedochází, a nastavení chyby proto nemůže výpočet nijak ovlivnit.

Kapitola 6

Závěr

Byla vytvořena knihovna pro numerickou simulaci postavená na základě technologie CUDA, která zprostředkovává výpočet nad více iniciálními body dynamického systému. Tato knihovna obsahuje rozhraní pro jazyk Java, díky kterému může být použita v existující implementaci metody pro analýzu dynamických systémů.

Měření ukázala, že délka výpočtu reaguje na hodnoty vytyčených parametrů vhodným způsobem. Největším problémem může být nevhodně zvolený model v kombinaci s požadavkem na vysokou přesnost výpočtu. V tu chvíli selhává implementovaná Runge-Kutta-Fehlbergova metoda a roste podíl sekvenční části výpočtu. Obecně lze ale říci, že akcelerace části, která generuje body trajektorií na základě soustavy diferenciálnich rovnice, proběhla úspěšně.

Do budoucna by bylo vhodné implementovat další numerické metody, zejména metody implicitní a vícekrokové, které v praxi dosahují lepších výsledků. Je však otázkou, zda složitost těchto metod nezpůsobí značné zpomalení výpočtu na grafické kartě. V případě implementací dalších metod navíc vznikne potřeba rozhodovacího modulu, který bude volit mezi dostupnými metodami tu, který se pro daný systém a uživatelské požadavky nejvíce hodí.

Co se týče zlepšení výkonu existující implementace metody pro analýzu dynamických systémů, byl učiněn pouze jeden krok. Vytvořenou knihovnu sice lze použít pro generování trajektorií, ale pokud bude s těmito trajektoriemi pracovat sekvenční program na procesoru, nedá se očekávat významné zrychlení. Proto je třeba paralelizovat další části algoritmu – kontrolu vzdáleností mezi trajektoriemi a ověřování LTL vlastnosti.

Literatura

[1] JCuda: Java Bindings for CUDA. [online; navštíveno 7. 5. 2011].
URL <http://www.jcuda.org>

[2] JUnit: Resources for Test Driven Development. [online; navštíveno 7. 5. 2011].
URL <http://www.junit.org>

[3] Ackermann, J.; Baecher, P.; Franzel, T.; aj.: Massively-parallel simulation of biochemical systems. *Proceedings of Massively Parallel Computational Biology on GPUs*, 2009.

[4] Bayramov, S.: Concentration oscillations in three-component reaction systems. *Biochemistry (Moscow)*, ročník 68, č. 3, 2003: s. 349–353, ISSN 0006-2979.

[5] Dalziel, S.: First order ordinary differential equations. 1998, [online; navštíveno 17. 4. 2011].
URL <http://www.damtp.cam.ac.uk/lab/people/sd/lectures/nummeth98/odes.htm>

[6] Dematté, L.; Prandi, D.: GPU computing for systems biology. *Briefings in bioinformatics*, ročník 11, č. 3, 2010: str. 323, ISSN 1467-5463.

[7] Dražan, S.: Výpočetní analýza nelineárních dynamických systémů. 2011.

[8] Fasshauer, G.: Numerical Methods for Differential Equations. 2005, [k dispozici online].
URL <http://www.math.iit.edu/~fass/#Teaching>

[9] Fink, J. H. M. K. K.: *Numerical Methods Using Matlab, 4th Edition, 2004*. ISBN 0-13-065248-2.

[10] Fisher, J.; Henzinger, T.: Executable cell biology. *Nature biotechnology*, ročník 25, č. 11, 2007: s. 1239–1249, ISSN 1087-0156.

[11] Hawick, K.; Playne, D.: Numerical Simulation of the Complex Ginzburg-Landau Equation on GPUs with CUDA. Massey University, Tech. Rep. CSTN-070, January 2010, to appear in Proc. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN), 14-16 Feb. 2011, in Innsbruck, Austria, 2010.

[12] Iserles, A.: *A first course in the numerical analysis of differential equations*. New York, NY, USA: Cambridge University Press, 1996, ISBN 0-521-55655-4.

[13] Larus, J.: Spending Moore's dividend. *Commun. ACM*, ročník 52, May 2009: s. 62–69, ISSN 0001-0782, doi:http://doi.acm.org/10.1145/1506409.1506425.
URL <http://doi.acm.org/10.1145/1506409.1506425>

[14] NVIDIA Corporation: *NVIDIA CUDA Programming Guide*.
URL <http://developer.nvidia.com/object/cuda_3_2_downloads.html>

[15] Nyland, L.; Harris, M.; Prins, J.: Fast n-body simulation with cuda. *GPU gems*, ročník 3, 2007: s. 677–695.

[16] Pelánek, R.: *Modelování a simulace komplexních systémů*. Brno: Nakladatelství Masarykovy univerzity, 2011, ISBN 978-80-210-5318-2.

[17] Pospíšil, Z.: Dynamické systémy a systémová dynamika. [online; navštíveno 7. 5. 2011].
URL <http://www.math.muni.cz/~pospisil/FILES/DynSys_SysDyn.pdf>

[18] Shampine, L. F.; Thompson, S.: Initial value problems. *Scholarpedia*, ročník 2, č. 3, 2007: str. 2861, [online; navštíveno 29. 3. 2011].
URL <http://www.scholarpedia.org/article/Initial_value_problems>

[19] Stirling, C.: Modal and temporal logics. In *Handbook of Logic in Computer Science*, ročník 2, Oxford University Press, 1992, s. 477–563.

[20] Westerhoff, H.; Hofmeyr, J.: What is systems biology? From genes to function and back. *Systems Biology*, 2005: s. 119–141.

[21] Wikipedia: CUDA — Wikipedia, The Free Encyclopedia. 2011, [online; navštíveno 17. 5. 2011].

URL <http://en.wikipedia.org/w/index.php?title=CUDA&oldid=429395684>

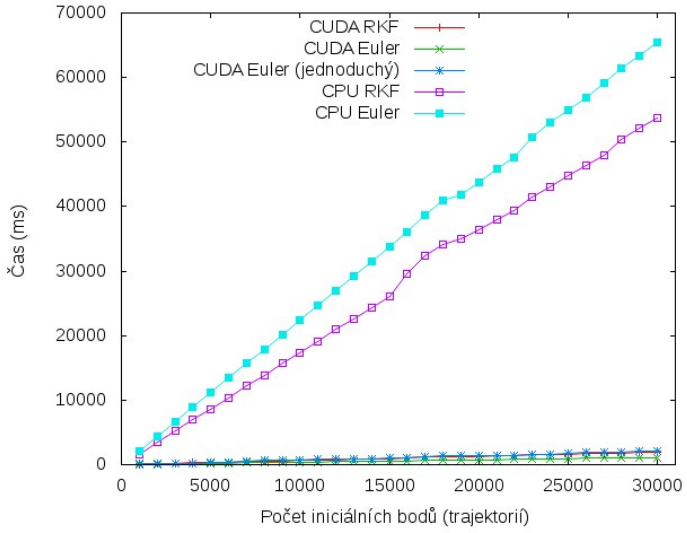
[22] Wikipedia: Lipschitz continuity — Wikipedia, The Free Encyclopedia. 2011, [online; navštíveno 29. 3. 2011].
URL <http://en.wikipedia.org/w/index.php?title=Lipschitz_continuity&oldid=421173596>

[23] Wikipedia: Moore's law — Wikipedia, The Free Encyclopedia. 2011, [online; navštíveno 23. 4. 2011].
URL <http://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=425165647>

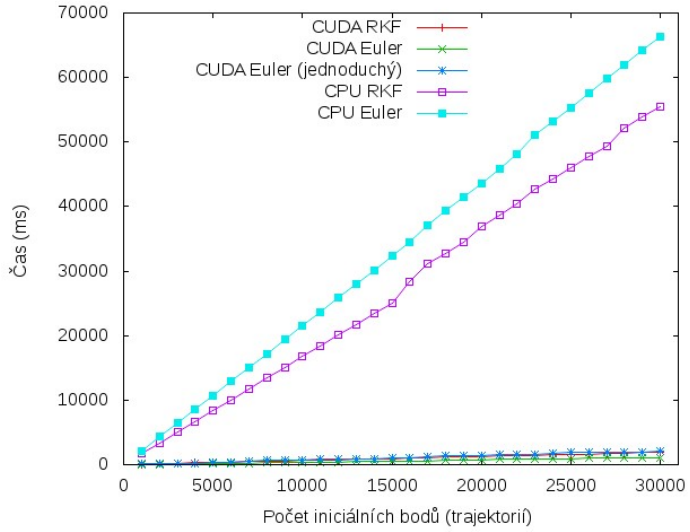
[24] Wikipedia: Newton's method — Wikipedia, The Free Encyclopedia. 2011, [online; navštíveno 17. 4. 2011].
URL <http://en.wikipedia.org/w/index.php?title=Newton%27s_method&oldid=423353214>

Příloha A

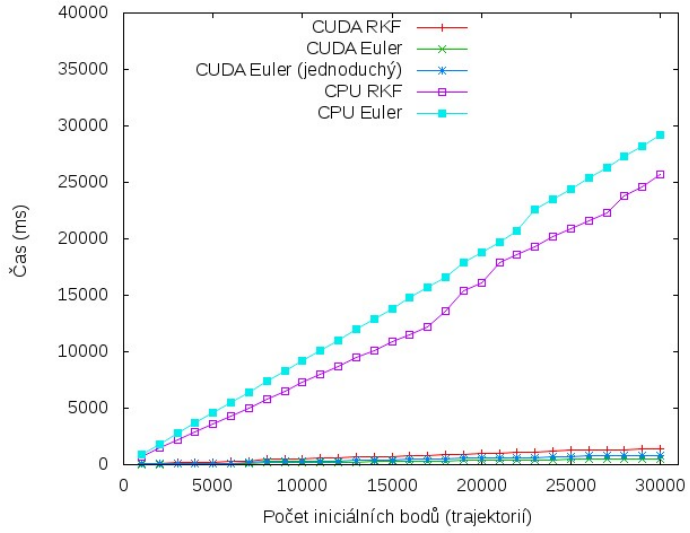
Grafy měření



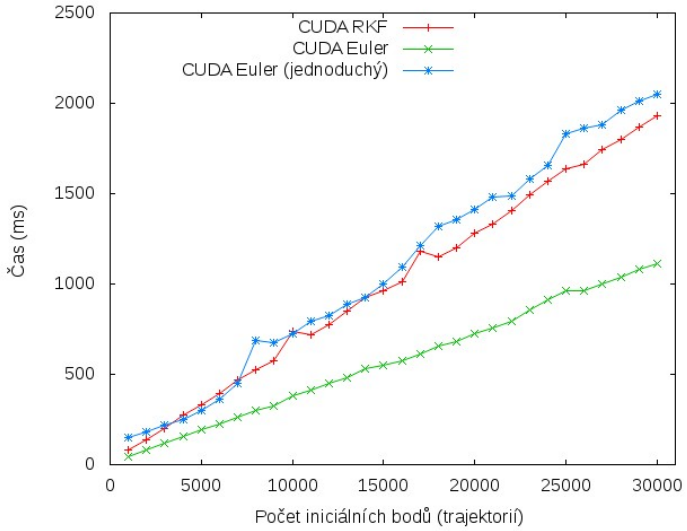
Obrázek A.1: Lineární systém deseti proměnných.



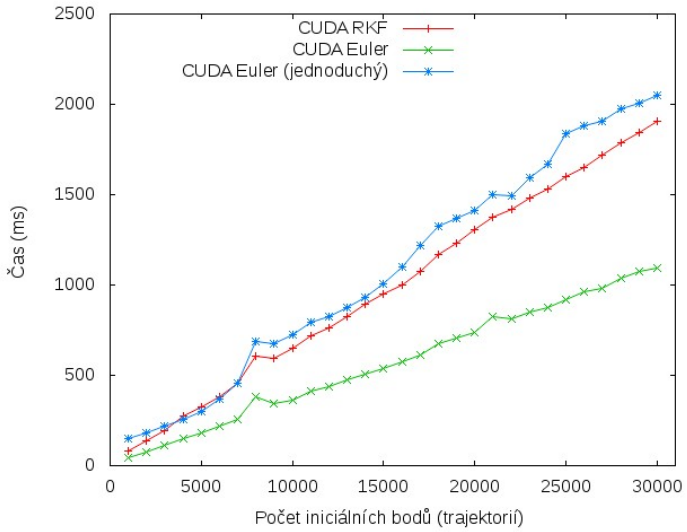
Obrázek A.2: Provázaný systém deseti proměnných



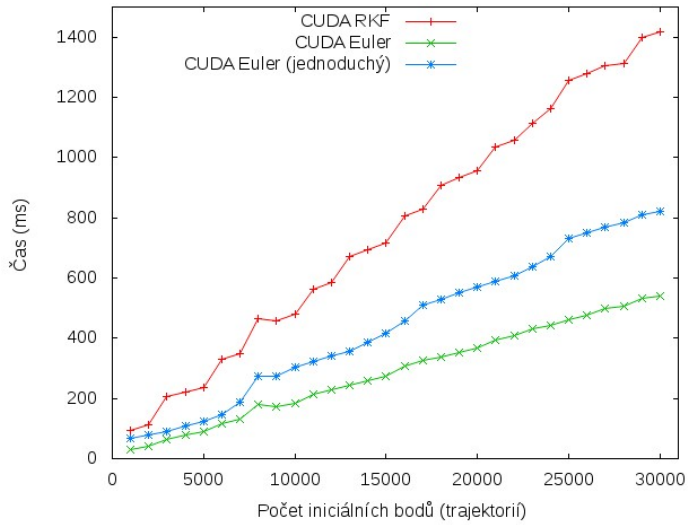
Obrázek A.3: Reálný systém tří proměnných.



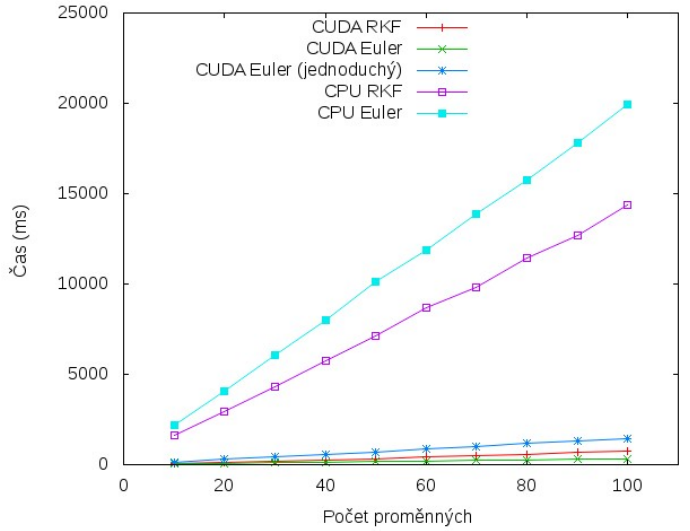
Obrázek A.4: Lineární systém deseti proměnných. Zobrazeny jsou pouze výpočty na grafické kartě.



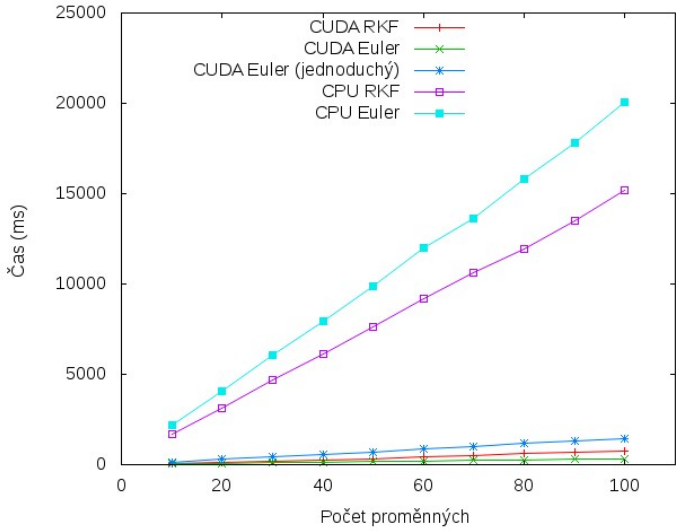
Obrázek A.5: Provázaný systém deseti proměnných. Zobrazeny jsou pouze výpočty na grafické kartě.



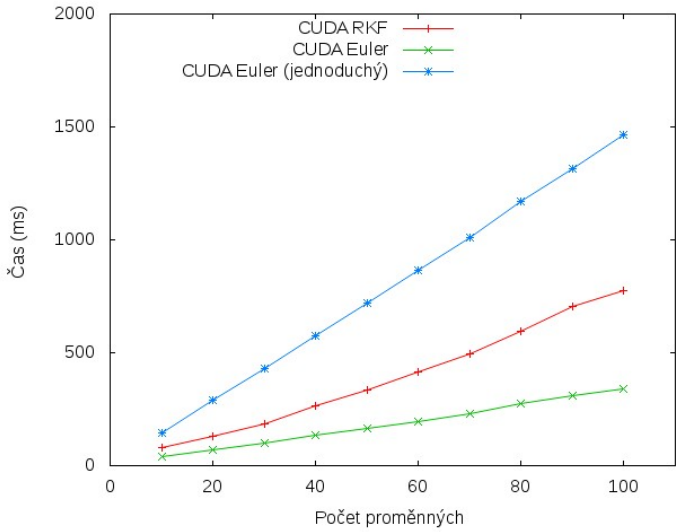
Obrázek A.6: Reálný systém tří proměnných. Zobrazeny jsou pouze výpočty na grafické kartě.



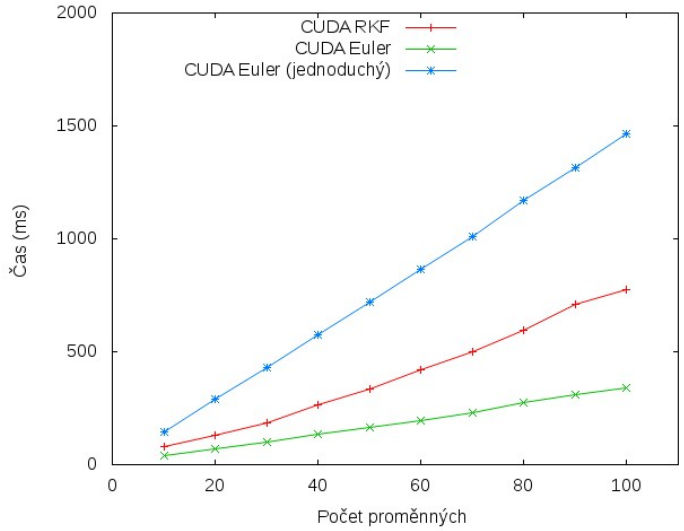
Obrázek A.7: Lineární systém, 1000 iniciálních bodů.



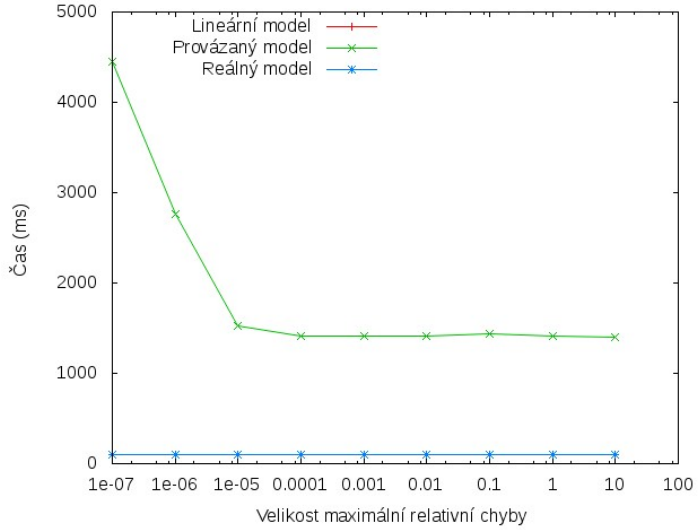
Obrázek A.8: Provázaný systém, 1000 iniciálních bodů.



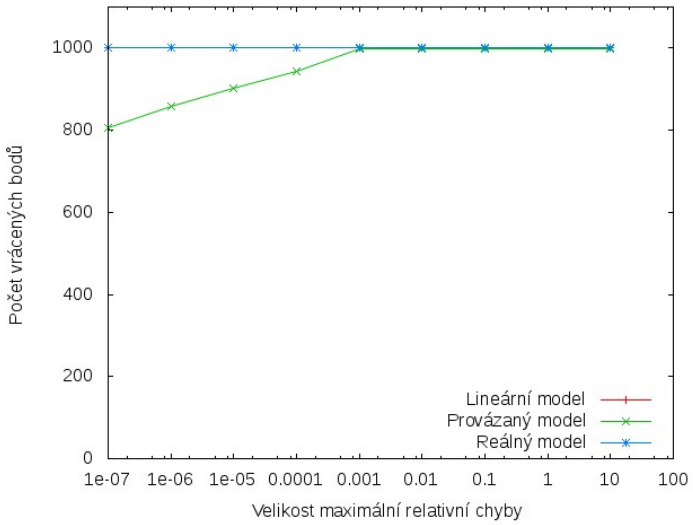
Obrázek A.9: Lineární systém, 1000 iniciálních bodů. Zobrazeny jsou pouze výpočty na grafické kartě.



Obrázek A.10: Provázaný systém, 1000 iniciálních bodů. Zobrazeny jsou pouze výpočty na grafické kartě.



Obrázek A.11: Výkon kernelu pro Runge-Kutta-Fehlberhovy metodu v závislosti na přesnosti výpočtu. Lineární a provázaný systém obsahuje 10 proměnných, reálný model 3 proměnně. Výpočet probíhal nad 1000 iniciálními body. Měření lineárního a reálného modelu splývá.



Obrázek A.12: Počet vrácených bodů kernelem pro Runge-Kutta-Fehlberhovy metodu v závislosti na přesnosti výpočtu. Lineární a provázaný systém obsahuje 10 proměnných, reálný model 3 proměnně. Výpočet probíhal nad 1000 iniciálními body. Měření lineárního a reálného modelu splývá.