VÝPOČTY NA GRAFICKÝCH PROCESORECH, ŘEŠENÍ PARCIÁLNÍCH DIFERENCIÁLNÍCH ROVNIC

Jan Šilar, Martin Vavrek, Tomáš Kulhánek, Pavol Privitzer, Jiří Kofránek, Tomáš Kroček, Martin Tribula

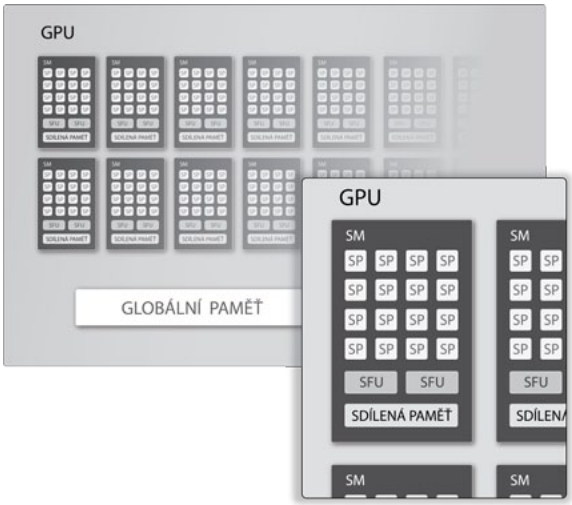
Úvod

Výpočetní výkon grafického procesoru (dále jen GPU – graphic processing unit) v současných osobních počítačích často výrazně převyšuje výkon procesoru CPU. Na trhu jsou ještě výkonnější GPU specializovaná na intenzivní výpočty. GPU se skládá z desítek až stovek výpočetních jader. Předpokladem využití výpočetního výkonu je vhodná paralelizace řešené úlohy. Zrychlení výpočtu, kterého je možné dosáhnout použitím GPU se značně liší v závislosti na řešené úloze. U některých úloh lze dosáhnout až stonásobného zrychlení. Na druhou stranu je mnoho úloh, které nelze vůbec paralelizovat a jejich výpočet na GPU by vedl naopak ke zpomalení. Cílem tohoto příspěvku je ukázat možnosti použití GPU pro obecné výpočty, přiblížit architekturu GPU a programování na platformě CUDA. Použití GPU je demonstrováno na konkrétní aplikaci. Pro názornost je mnoho detailů vynecháno nebo zjednodušeno. Cílem není naučit čtenáře programovat v CUDA. Detailní popis GPU a CUDA je například v [1,2].

Architektura GPU

Architektury GPU různých výrobců se liší většinou jen mírně. Zde je zjednodušeně popsána architektura firmy NVIDIA.

Jádra SP (streaming processor) GPU podporují jen logické operace a sčítání a násobení. Jádra jsou sdružena do několika multiprocesorů SM



Obrázek 1 — Architektura GPU

(streaming multiprocessor). Jádra jednoho multiprocesoru mají společnou sdílenou paměť. Do sdílené poměti mají přístup pouze jádra příslušného multiprocesoru. Část sdílené paměti může být vyhrazena jako cache pro jednotlivá jádra. Součástí multiprocesoru je také jednotka pro výpočet speciálních funkcí (sin, sqrt...) SFU (special function unit). Celý grafický procesor se pak skládá z několika multiprocesorů a globální paměti. Do globální paměti mohou přistupovat všechna jádra, ale přístup je ve srovnání se sdílenou pamětí výrazně pomalejší. Procesy běžící na CPU mají přístup jen do globální paměti. Kopírování dat mezi RAM a globální pamětí je ještě pomalejší, než přístup do globální paměti v rámci GPU.

Jádra pracují v režimu „single instruction multiple data". To znamená, že všechna jádra provádí stejný výpočet paralelně na různých datech. Všechna jádra multiprocesoru tedy provádí v jednom taktu stejnou instrukci.

Pro představu nVidia Tesla C2050 má 15 multiprocesorů po 32 jádrech (dohromady 448 jader). Velikost sdílené paměti každého multiprocesoru je 64KB. Globální paměť má 3GB. Celkový výpočetní výkon v single precision je 1,03 Tflops.

CUDA

Programovat pro GPU lze na dvou platformách: OpenCL nebo CUDA. Co se jazyka týče, jsou obě platformy rozšířením ANSI C o několik málo konstruktů. OpenCL je obecnější a je podporováno GPU od nVidia a ATI, ale i procesory CPU (např. většina x86), různými signálovými procesory (DSP) atd. CUDA je platforma vyvinutá firmou nVidia a je podporována pouze jejími grafickými kartami. Závislost na hardwaru konkrétního výrobce je samozřejmě nevýhodou. Na druhou stranu je programování v CUDA ve srovnání s OpenCL o něco snazší (zejména inicializace zařízení). Aplikace v CUDA běží také o něco rychleji díky lepší optimalizaci při překladu.

Použití CPU a GPU při výpočtu je možné kombinovat. Část kódu, kterou nelze paralelizovat, běží typicky na CPU, paralelní část se počítá na GPU. Výpočet na GPU se spouští pomocí speciálních funkcí, tzv. kernelů. Kernely obsahují kód, který je počítán paralelně na vláknech v GPU. Vlákno běží na jednom jádře SP. Vlákna jsou sdružována do bloků. Blok vláken je spuštěn na jednom multiprocesoru SM. Jednomu SM může být přiděleno více bloků. SM pak mezi těmito bloky přepíná. Bloky jsou sdružovány do gridu. Počet bloků v gridu může být opět vyšší, než je počet SM v GPU. Při spuštění kernelu se zadává velikost bloků (počet vláken v bloku) i velikost gridu (počet bloků v gridu). Vlákno většinou zpracovává jeden prvek vstupního pole. Vlákna i bloky mají své indexy, z kterých se počítá index prvku vstupního pole pro zpracování konkrétním vláknem.

Před spuštěním kernelu musí být zařízení inicializováno. Dále jsou v globální paměti GPU alokována pole a jsou tam nakopírována vstupní data z RAM. Po dokončení výpočtu jsou výstupní data překopírován z GPU do RAM.

```
__global__void addKernel(int *c, const int *a, const int *b)
{
   //Výpočet indexu pro přístup do polí z indexu vlákna,
   //indexu bloku a velikosti bloku:
   int i = threadIdx.x + blockIdx.x*blockDim.x;
   //soupčet:
   c[i] = a[i] + b[i];
}

   void addWithCuda(int *c, const int *a, const int *b,
   size_t size)
{
   int *dev_a, *dev_b, *dev_c;
   //Volba GPU. (Kvůli systémům s více GPU.):
   cudaSetDevice(0);
   //Alokace paměti na GPU:
   cudaMalloc((void**)&dev_a, size*sizeof(int));
   cudaMalloc((void**)&dev_b, size*sizeof(int));
   cudaMalloc((void**)&dev_c, size*sizeof(int));
   //Skopírování vstupních polí do GPU:
   cudaMemcpy(dev_a, a, size*sizeof(int),
   cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b, b, size*sizeof(int),
   cudaMemcpyHostToDevice);
   //Spuštění kernelu na GPU ( 8bloků, každý size/8
   vláken):
   addKernel<<<8, size/8>>>(dev_c, dev_a, dev_b);
   //Synchronizace vláken:
   cudaDeviceSynchronize();
   //Překopírování výstupního pole do RAM:
   cudaMemcpy(c, dev_c, size*sizeof(int),
   cudaMemcpyDeviceToHost);
   //Uvolnění paměti GPU:
   cudaFree(dev_c);
   cudaFree(dev_a);
   cudaFree(dev_b);
   return;
}
```

Příklad kódu v CUDA pro sečtení dvou polí v globální paměti GPU.

Pokud vlákno přistupuje po čas běhu kernelu vícekrát k prvku pole v globální paměti, je vhodné pro urychlení toto pole napřed zkopírovat do sdílené paměti a přistupovat pak tam. Pole ve sdílené paměti jsou alokována z kernelu. Sdílená paměť je uvolněna, když kernel skončí.

Pokud jsou v kódu podmínky (if – then – else), dochází (kvůli principu „single instruction multiple data") k divergenci výpočtu. To znamená, že větve „then" a „else" jsou vyhodnoceny sekvenčně.

Optimalizace kódu v CUDA vyžaduje většinou hlubší znalost architektury GPU. Často je potřeba aplikaci konfigurovat pro konkrétní hardware. Existuje mnoho optimalizačních metod, většina je založena na vhodné práci s pamětí.

Řešení PDE

V naší laboratoři plánujeme na GPU numericky řešit parciální diferenciální rovnice (PDE). Řešení PDE ve dvou a více dimensích je výpočetně náročná úloha. Pro první vyzkoušení implementujeme v CUDA solver pro řešení difusní rovnice v 1D.

Difusní člen (s druhou derivací podle x) popisuje jevy jako je vedení tepla, nebo

$$a(x,t)\frac{\partial^2 y}{\partial x^2}+\varrho(x,t)=\frac{\partial y}{\partial t}$$

$$y(x,t_0)=y_0(x)$$

$$y(x_l,t)=y_l(t),\quad y(x_r,t)=y_r(t)$$

difuse plynů. Člen bez derivace představuje zdroje (tepla, částic plynu).

V diskrétních bodech výpočetní síťky
hledáme aproximaci řešení

$$x_0<x_1<x_2\ \ldots\ x_M$$
$$t^0<t^1\ \ldots\ t^N$$

uvedené rovnice. Prostorový a časový krok jsou

$$y_m^n=y(x_m,t_n)$$

Rovnici můžeme aproximovat diferenčním schématem

$$\Delta x_i=x_{i+1}-x_i$$
$$\Delta t_i=t_{i+1}-t_{i.}$$

V n–té iteraci výpočtu známe hodnoty y^n_m pro všechna m a pomocí schématu počítáme nové hodnoty y^{n+1}_m. Uvedené schéma je explicitní. To znamená, že

$$a_m^n\frac{y_{m+1}^n-2y_m^n+y_{m-1}^n}{(\Delta x)^2}+\varrho_m^n=\frac{y_m^{n+1}-y_m^n}{\Delta t}.$$

v něm vystupuje jediná neznámá hodnota y^{n+1}_m, kterou je možné přímo vyjádřit a spočítat. Aby bylo řešení parabolické difusní rovnice explicitním schématem stabilní, je potřeba volit časový krok úměrný druhé mocnině prostorového kroku.

$$\Delta t\sim\Delta x^2$$

Když potom zmenšujeme prostorový krok (kvůli zpřesnění výpočtu), časový krok se zmenšuje kvadraticky. To vede k extrémně krátkému časovému kroku, nutnosti mnoha iterací a neúměrné časové náročnosti výpočtu. Použijeme–li

pro aproximaci řešené rovnice nějaké implicitní schéma, výpočet je stabilní i pro

$$\Delta t \sim \Delta x.$$

Zvolili jsme Crank–Nicolsonovo schéma [3]
Schéma je implicitní, vystupuje v něm tedy více neznámých hodnot (zde

$$a_m^n \frac{(y_{m+1}^{n+1} - 2y_m^{n+1} + y_{m-1}^{n+1}) + (y_{m+1}^n - 2y_m^n + y_{m-1}^n)}{2(\Delta x_m)^2} + \varrho_m^n = \frac{y_m^{n+1} - y_m^n}{\Delta t^n}.$$

y^{n+1}_{m-1}, y^{n+1}_{m}, y^{n+1}_{m+1}). Rovnice schématu pro m = 1 .. M–1, tvoří společně se dvěma okrajovými podmínkami

$$y_0^{n+1} = y_l(t^{n+1}), \quad y_M^{n+1} = y_r(t^{n+1})$$

soustavu M+1 lineárních rovnic pro M+1 neznámých

$$y_m^{n+1}, \quad m \in 0 \,..\, M.$$

Protože v m–té rovnici vystupují pouze 3 po sobě jdoucí neznámé

$$y_m^{n+1}, y_{m-1}^{n+1}, y_m^{n+1},$$

je matice soustavy tridiagonální – všechny prvky kromě poddiagonály, diagonály a naddiagonály jsou nulové.

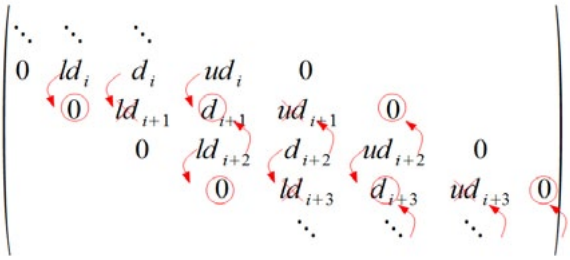
Tato soustava je řešena v každém kroce výpočtu. Matice je uložena ve

$$\begin{pmatrix} d_0 & ud_0 & 0 & 0 & \cdots & 0 \\ ld_1 & d_1 & ud_1 & 0 & \cdots & 0 \\ 0 & ld_2 & d_2 & ud_2 & & 0 \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & ld_{M-1} & d_{M-1} & ud_{M-1} \\ 0 & 0 & 0 & 0 & ld_M & d_M \end{pmatrix} . \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{M-1} \\ y_M \end{pmatrix} = \begin{pmatrix} rhs_0 \\ rhs_1 \\ rhs_2 \\ \vdots \\ rhs_{M-1} \\ rhs_M \end{pmatrix}$$

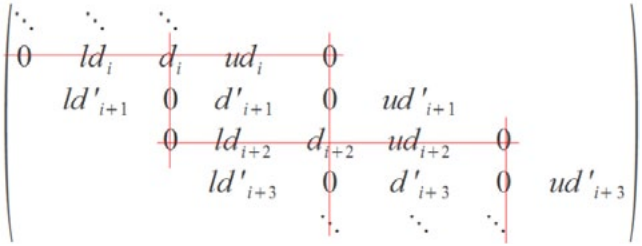
třech polích. Běžně používaný Thomasův algoritmus (Gaussova eliminace modifikovaná pro soustavu s tridiagonální maticí, složitost O(M)) nelze paralelizovat. Používáme metodu paralelní cyklické redukce [4,5]. Je to typická metoda „rozděl a panuj". V první iteraci je soustava rozdělena na dvě nezávislé soustavy. Dělení soustavy opakujeme, dokud nedostaneme soustavy velikosti 1 nebo 2, které jsou již řešeny přímo. Složitost tohoto algoritmu je M log(M), lze ho ale snadno paralelizovat.

Popišme si, jak je soustava rozdělena. Předpokládejme, že index i v matici je sudé číslo. Na všech lichých řádcích provedeme následující úpravu. Vynulujeme poddiagonální prvek (vlevo od diagonálního) řádku odečtením vhodného násobku řádku předchozího. Tím dostaneme nově nenulovou hodnotu v druhém poddiagonálním prvku. Stejně vynulujeme naddiagonální prvek odečtením násobku následujícího řádku a dostaneme nenulovou hodnotu

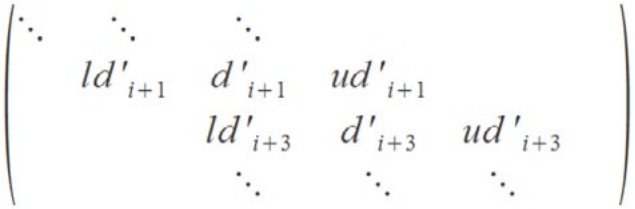
v druhém naddiagonálním prvku. Odpovídající úpravy je potřeba provádět samozřejmě také na pravé straně soustavy.



Koeficienty neznámých se sudým indexem v lichých řádcích jsou nyní nulové. Ze soustavy vypustíme sudé řádky (rovnice) a sudé neznámé.



Tím dostáváme soustavu poloviční dimense, ve které vystupují jen liché rovnice a liché neznámé.



Stejnými úpravami na sudých řádcích původní soustavy dostaneme soustavu pro sudé rovnice a sudé neznámé.

Naplnění i vyřešení matice probíhá na GPU. Přenos dat mezi pamětí CPU a GPU probíhá jen na začátku při kopírování počátečních podmínek a koeficientů rovnice do GPU a potom při kopírování výsledků zpět na CPU.

Koeficienty všech matic jsou po celou dobu výpočtu uloženy v jediné trojici polí *ld, d* a *ud*. Pokud jsou tato pole uložena v globální paměti, je výpočet pomalý kvůli pomalému přístupu k datům.

Další možností je uložit pole po částech do sdílené paměti multiprocesorů.

Předpokladem je, že vlákna budou mít přístup do paměti všech multiprocesorů, ale ve skutečnosti mají přístup pouze do sdílené paměti vlastního multiprocesoru. Proto můžeme použít pro výpočet ve sdílené paměti

pouze jediný multiprocesor a využijeme tak jen zlomek výkonu GPU. Tento přístup by byl vhodný, pokud bychom řešili několik soustav paralelně.

Výsledky a další plány

Pro změření rychlosti výpočtu jsme zvolili úlohu vedení tepla s po částech konstantní počáteční podmínkou s nespojitostí uprostřed. Počítali jsme na síťce s různým počtem uzlů a měřili čas výpočtu na GPU s použitím globální paměti, sdílené paměti a na CPU (Thomasův algoritmus). Byla použita grafická karta nVidia Quadro 2000M a procesor Intel i7–2720QM (jedno jádro). Časový krok byl zvolen napevno, aby byl počet kroků a tedy počet řešení lineární soustavy vždy stejný (soustava řešena během jednoho výpočtu 10677 krát).

Počet uzlů	CPU	GPU – globální	GPU – sdílená
800	430ms	2120ms	950ms
4000	2200ms	3500ms	–
8000	4243ms	6676ms	–
16000	8424ms	17771ms	–

Tabulka 1

Pro výpočet na více než 800 uzlech nestačila lokální paměť. Při testech byl bohužel výpočet na GPU vždy pomalejší než na CPU. Použité GPU nebylo příliš výkonné. Při nasazení výkonnějšího GPU bychom mohli očekávat mírné zrychlení oproti výpočtu na CPU. Významnějšího zrychlení bychom ale zřejmě nedosáhli. Hlavním omezením je v prvním případě pomalý přístup do globální paměti, v druhém využití jediného multiprocesoru.

Plánujeme implementovat hybridní metodu pro řešení lineárního systému, která by měla výpočet významně urychlit. Výpočet bude rozdělen do tří fází. V první fázi bude probíhat dělení soustavy v globální paměti. V okamžiku, kdy bude systém rozdělen na alespoň tolik podsystému, kolik je v GPU multiprocesorů a zároveň budou tyto podsystémy dostatečně malé, překopírují se data do sdílené paměti a výpočet bude pokračovat zde. Multiprocesory již mezi sebou nemusí nijak komunikovat, protože každý řeší nezávislou úlohu. Ve chvíli, kdy bude systém rozdělen na tolik podsystémů, kolik je v GPU jader, přepne se na poslední fázi. Nyní je již dostatek nezávislých úloh, které budeme řešit paralelně každou klasickým sekvenčním Thomasovým algoritmem.

Závěr

Existují dva standardy pro programování na GPU. Zaměřili jsme se na platformu CUDA. Byla představena architektura GPU a popsány základy programování v CUDA.

Implementovali jsme v CUDA kód pro paralelní řešení difusní rovnice v 1D. Základem je metoda pro řešení soustavy lineárních rovnic s tridiagonální maticí. Výpočet na GPU bohužel zatím nepřinesl žádné zrychlení oproti výpočtu na CPU. Naopak je o něco pomalejší. Proto jsme navrhli novou hybridní metodu, od které očekáváme zrychlení výpočtu. Řešení PDR implicitními metodami na GPU ale není snadné. Nelze očekávat takové zrychlení jako u jiných úloh, které jsou pro výpočet na GPU vhodnější.

Poděkování

Tato práce je podporována projektem MPO FR—TI3/869.

Literatura

[1.] Kirk, D. B., & Hwu, W.–mei W. (2010). Programming Massively Parallel Processors. (NVidia, Eds.)Special Edition (p. 258). Morgan Kaufmann. Retrieved from www.mkp.com

[2.] Jaroslav Sloup, podklady k přednášce Obecné výpočty na grafických procesorech, FEL, ČVUT, https://service.felk.cvut.cz/courses/A4M39GPU/lectures.html

[3.] Iserles, A. (2009). A First Course in the Numerical Analysis of Differential Equations. Cambridge University Press.

[4.] Egloff, D. (2010). High Performance Finite Difference PDE Solvers on GPUs. Social Science Research Network, 1–28.

[5.] Hee–Seok Kim, Shengzhao Wu, Li–wen Chang, Wen–mei W. Hwu, "A Scalable Tridiagonal Solver for GPUs," icpp, pp.444–453, 2011 International Conference on Parallel Processing, 2011

Kontakt:

Jan Šilar
Oddělení biokybernetiky počítačové podpory výuky
Ústav patologické fyziologie, 1. LF UK
U Nemocnice 5, 128 53 Praha 2
Tel.: +420 224 965 912
e-mail: *jansilar@jansilar.cz*