

Úvod, základy CUDA

Jiří Filipovič

jaro 2013

Motivace – Moorův zákon

Moorův zákon

Počet tranzistorů na jednom čipu se přibližně každých 18 měsíců **zdvojnásobí**.

Motivace – Moorův zákon

Moorův zákon

Počet tranzistorů na jednom čipu se přibližně každých 18 měsíců **zdvojnásobí**.

Adekvátní růst výkonu je zajištěn:

- **dříve** zvyšováním frekvence, instrukčním paralelismem, out-of-order spouštěním instrukcí, vyrovnávacími paměťmi atd.
- **dnes** vektorovými instrukcemi, zmnožováním jader

Motivace – změna paradigmatu

Důsledky Moorova zákona:

- **dříve:** rychlost zpracování programového vlákna procesorem se každých 18 měsíců zdvojnásobí
 - změny ovlivňují především návrh kompilátoru, aplikační programátor se jimi nemusí zabývat
- **dnes:** rychlost zpracování **dostatečného počtu** programových vláken se každých 18 měsíců zdvojnásobí
 - pro využití výkonu dnešních procesorů je zapotřebí paralelizovat algoritmy
 - paralelizace vyžaduje nalezení souběžnosti v řešeném problému, což je (stále) úkol pro programátora, nikoliv kompilátor

Motivace – druhy paralelismu

- úlohový paralelismus
 - problém je dekomponován na úlohy, které mohou být prováděny souběžně
 - úlohy jsou zpravidla komplexnější, mohou provádět různou činnost
 - vhodný pro menší počet výkonných jader
 - zpravidla častější (a složitější) synchronizace
- datový paralelismus
 - souběžnost na úrovni datových struktur
 - zpravidla prováděna stejná operace nad mnoha prvky datové struktury
 - jemnější paralelismus umožňuje konstrukci jednodušších procesorů

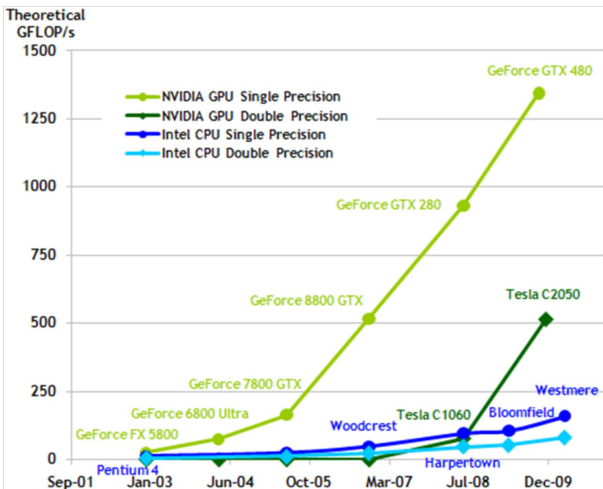
Motivace – druhy paralelismu

- z pohledu programátora
 - rozdílné paradigma znamená rozdílný pohled na návrh algoritmů
 - některé problémy jsou spíše datově paralelní, některé úlohově
- z pohledu vývojáře hardware
 - procesory pro datově paralelní úlohy mohou být **jednodušší**
 - při stejném počtu tranzistorů lze dosáhnout **vyššího aritmetického výkonu**
 - jednodušší vzory přístupu do paměti umožňují konstrukci HW s **vysokou paměťovou propustností**

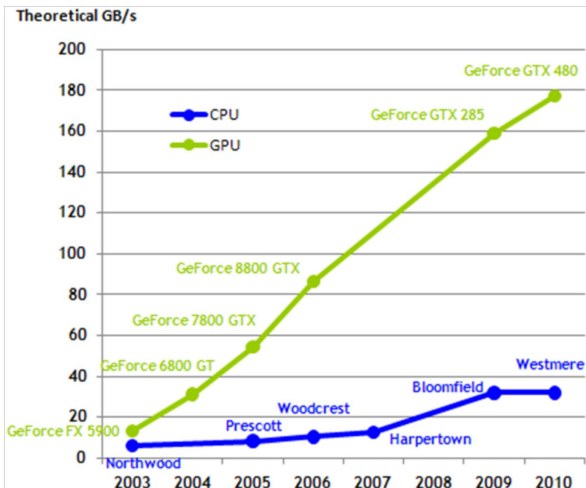
Motivace – grafické výpočty

- datově paralelní
 - provádíme stejné výpočty pro různé vertexy, pixely, ...
- předdefinované funkce
- programovatelné funkce
 - specifické grafické efekty
 - GPU se stávají stále více programovatelnými
 - díky tomu lze zpracovávat i jiné, než grafické úlohy

Motivace – výkon



Motivace – výkon



Motivace – shrnutí

- GPU jsou výkonné
 - řádový nárůst výkonu již stojí za studium nového programovacího modelu
- pro plné využití moderních GPU i CPU je třeba programovat paralelně
 - paralelní architektura GPU přestává být řádově náročnější
- GPU jsou široce rozšířené
 - jsou levné
 - spousta uživatelů má na stole superpočítač

Motivace – uplatnění

Využití GPU pro obecné výpočty je dynamicky se rozvíjející oblast s širokou škálou aplikací

Motivace – uplatnění

Využití GPU pro obecné výpočty je dynamicky se rozvíjející oblast s širokou škálou aplikací

- vysoce náročné vědecké výpočty
 - výpočetní chemie
 - fyzikální simulace
 - zpracování obrazů
 - a mnohé další...

Motivace – uplatnění

Využití GPU pro obecné výpočty je dynamicky se rozvíjející oblast s širokou škálou aplikací

- vysoce náročné vědecké výpočty
 - výpočetní chemie
 - fyzikální simulace
 - zpracování obrazů
 - a mnohé další...
- výpočetně náročné aplikace pro domácí uživatele
 - kódování a dekodování multimediálních dat
 - herní fyzika
 - úprava obrázků, 3D rendering
 - atd...

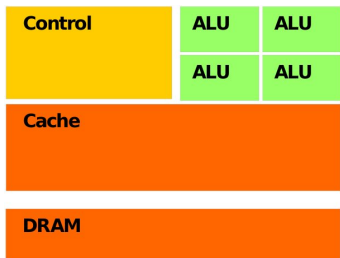
Architektura GPU

CPU vs. GPU

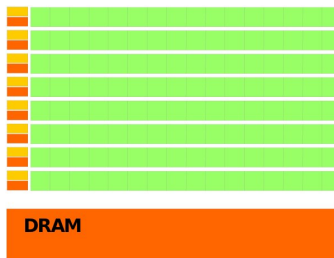
- jednotky jader vs. **desítky multiprocesorů**
- out of order vs. **in order**
- MIMD, SIMD pro krátké vektory vs. **SIMT pro dlouhé vektory**
- velká cache vs. **malá cache, často pouze pro čtení**

GPU používá více tranzistorů pro výpočetní jednotky než pro cache a řízení běhu => vyšší výkon, méně univerzální

Architektura GPU



CPU



GPU

Architektura GPU

V rámci systému:

- koprocesor s dedikovanou pamětí
- asynchronní běh instrukcí
- připojen k systému přes PCI-E

Processor G80

G80

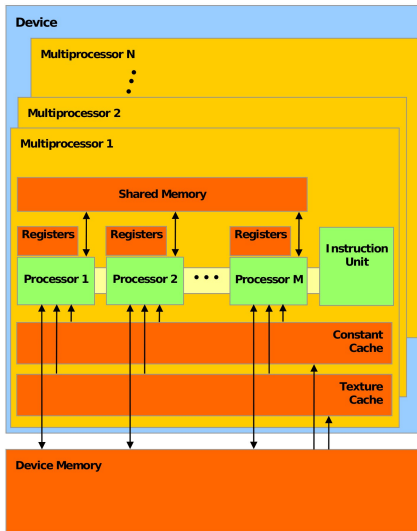
- první CUDA procesor
- obsahuje 16 multiprocessorů
- multiprocessor
 - 8 skalárních procesorů
 - 2 jednotky pro speciální funkce
 - až 768 threadů
 - HW přepínání a plánování threadů
 - thready organizovány po 32 do warpů
 - SIMT
 - nativní synchronizace v rámci multiprocessoru

Paměťový model G80

Paměťový model

- 8192 registrů sdílených mezi všemi thready multiprocesoru
- 16 KB sdílené paměti
 - lokální v rámci multiprocesoru
 - stejně rychlá jako registry (za dodržení určitých podmínek)
- paměť konstant
 - cacheovaná, pouze pro čtení
- paměť pro textury
 - cacheovaná, 2D prostorová lokalita, pouze pro čtení
- globální paměť
 - pro čtení i zápis, necacheovaná
- přenosy mezi systémovou a grafickou pamětí přes PCI-E

Processor G80



Další vývoj

Procesory odvozené od G80

- double-precision výpočty
- relaxována pravidla pro efektivní přístup ke globální paměti
- navýšeny on-chip zdroje (více registrů, více threadů na MP)
- lepší možnosti synchronizace (atomické operace, hlasování warpů)

Fermi

- vyšší paralelizace na úrovni multiprocessoru (více jader, dva warp schedulery, více DP výkonu)
- konfigurovatelná L1 a sdílená L2 cache
- plochý adresní prostor
- lepší přesnost v plovoucí řádové čárce
- paralelní běh kernelů
- širší možnosti synchronizace
- další změny plynoucí z odlišné architektury

Srovnání teoretické rychlosti GPU a CPU

Teoretická maxima

- GPU má cca 10× rychlejší aritmetiku
- GPU má cca 5× vyšší propustnost paměti
- zajímavé pro mnohé problémy (budu čekat na výsledky simulace měsíc nebo rok? pojede mi hra na 3 nebo 30 fps?)

Některé publikace ukazují 100× i 1000× zrychlení

- v pořádku, je-li interpretováno jako zrychlení oproti produkčnímu SW (ten nemusí být perfektně optimalizovaný)
- interpretováno jako srovnání CPU a GPU zpravidla nesmysl

Srovnáváme-li přínos GPU oproti CPU, musíme uvažovat efektivní implementaci pro obě platformy.

Srovnání teoretické rychlosti GPU a CPU

V praxi máme však často sériový CPU kód

- běh v jednom vlákně znamená až $16\times$ zpomalení (16-jádrové CPU)
- absence vektorizace znamená až $4\times$ zpomalení (32-bit operace u SSE instrukcí), $8\times$ u AVX instrukcí

Oproti sériové implementaci tedy můžeme kód paralelizací a vektorizací zrychlit

- $32\times$ pro čtyřjádrové CPU s AVX nebo osmijádrové s SSE

GPU akcelerací pak

- cca $300\times$

Vektorizace a paralelizace pro CPU je však programátorskou náročností **srovnatelná** s GPU akcelerací.

Teoretické vs. dosažitelné zrychlení

Výkonový odstup GPU může být vyšší

- jednotky pro speciální funkce, operace na texturách
- SIMT pružnější než SIMD
- neduhy SMP (omezení škálování propustnosti paměti, „vytloukání řádků cache“)

Stejně jako nižší

- nedostatek paralelismu
- příliš vysoký overhead
- nevhodný algoritmus pro GPU architekturu

Dále se podíváme, jak rozlišit, jestli je nebo naopak není váš algoritmus vhodný pro GPU.

Paralelizace

Sčítání vektorů

- jednoduché datově-paralelní vyjádření
- žádná synchronizace
- potřebujeme velké vektory

Game of Life

- co chceme paralelizovat?

Game of Life – zjištění nového stavu hry

- pro větší herní plochy dostatek paralelismu
- jednoduchá synchronizace

Game of Life – zjištění stavu buňky po n krocích

- inherentně sekvenční? (Game of Life je P -complete, $P \stackrel{?}{=} NC$)
- neznáme paralelní algoritmus

Paralelizace

Redukce

- na první pohled může vypadat sekvenčně
- ve skutečnosti realizovatelná v $\log n$ krocích
- často je třeba nedržet se sekvenční verze a zamyslet se nad paralelizací problému (ne sekvenčního algoritmu)

Paralelizace

Problém nalezení povodňové mapy

- máme výškovou mapu terénu, přítok vody, a chceme zjistit, jaká oblast se zatopí
- sekvenčnost dána rozléváním vody
- je snadné najít úlohově-paralelní algoritmus, datově-paralelní už tak ne
 - periodická aktualizace stavu každého bodu mapy
 - aktualizace omezená jen na hranice vodní plochy (šetří procesory)
 - rozlévání vody zametací přímkou (vhodnější pro GPU, jednodušší synchronizace)
 - hledání souvislých oblastí a jejich spojování (odstraňuje sekvenčnost rozlévání)
 - vždy práce navíc oproti sekvenční/úlohově-paralelní verzi
- úkol PV197 na podzim 2010, výkon odevzdaných implementací se lišil o 4 řády (!)

Divergence kódu

Divergence kódu

- serializace, divergují-li thready uvnitř warpu
- nalezení nedivergujícího algoritmu může být snadné
 - redukce
- ale také může prakticky znemožnit akceleraci některých jinak dobře paralelizovatelných algoritmů
 - mnoho nezávislých stavových automatů
 - nutnost zamyslet se nad výrazně odlišným algoritmem pro daný problém

Divergence přístupu do paměti

Divergence přístupu do paměti

- není-li do paměti přístupováno po souvislých blocích v rámci warpu, snižuje se její propustnost
- často velmi těžko překonatelný problém
 - průchod obecného grafu
- může vyžadovat využití odlišných datových struktur
 - práce s řádkými maticemi
- u rigidnějších struktur si lze často pomoci on-chip pamětí
 - transpozice matic

Latence GPU

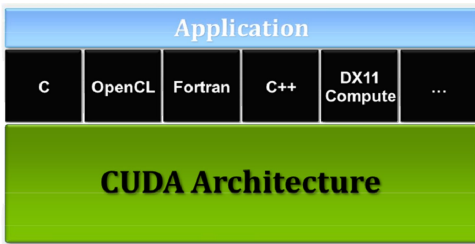
GPU je dnes často propojena se zbytkem systému přes PCI-E

- kopírování vstupů/výstupů je relativně pomalé
- akcelerovaný algoritmus musí provádět dostatečné množství aritmetiky na přenášená data
 - násobení matic je vhodné ($\mathcal{O}(n^3)$ operací na $\mathcal{O}(n^2)$ dat)
 - sčítání vhodné není ($\mathcal{O}(n^2)$ operací na $\mathcal{O}(n^2)$ dat), může být však součástí většího problému

CUDA

CUDA (Compute Unified Device Architecture)

- architektura pro paralelní výpočty vyvinutá firmou NVIDIA
- poskytuje nový programovací model, který umožňuje efektivní implementaci obecných výpočtů na GPU
- je možné použít ji s více programovacími jazyky



C for CUDA

C for CUDA přináší rozšíření jazyka C pro paralelní výpočty

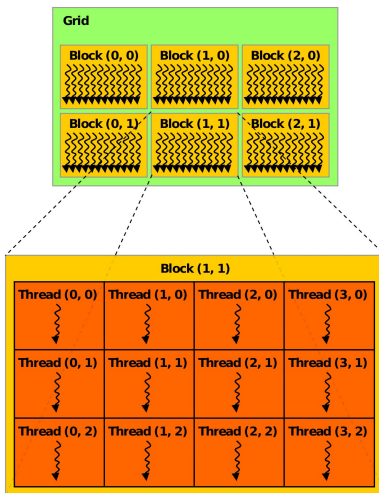
- explicitně oddělen host (CPU) a device (GPU) kód
- hierarchie vláken
- hierarchie pamětí
- synchronizační mechanismy
- API

Hierarchie vláken

Hierarchie vláken

- vlákna jsou organizována do bloků
- bloky tvoří mřížku
- problém je dekomponován na podproblémy, které mohou být prováděny nezávisle paralelně (bloky)
- jednotlivé podproblémy jsou rozděleny do malých částí, které mohou být prováděny kooperativně paralelně (thready)
- dobře škáluje

Hierarchie vláken

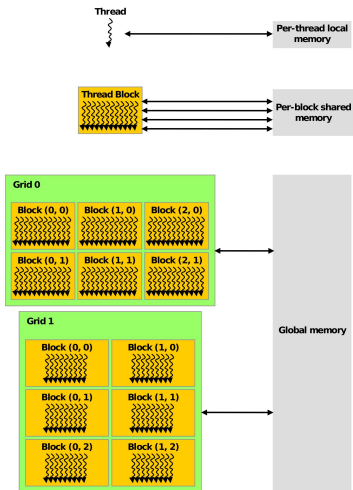


Hierarchie pamětí

Více druhů pamětí

- rozdílná viditelnost
- rozdílný čas života
- rozdílné rychlosti a chování
- přináší dobrou škálovatelnost

Hierarchie pamětí



Příklad – součet vektorů

Chceme sečíst vektory a a b a výsledek uložit do vektoru c .

Příklad – součet vektorů

Chceme sečíst vektory a a b a výsledek uložit do vektoru c .
Je třeba najít v problému paralelismus.

Příklad – součet vektorů

Chceme sečíst vektory a a b a výsledek uložit do vektoru c .

Je třeba najít v problému paralelismus.

Sériový součet vektorů:

```
for (int i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

Příklad – součet vektorů

Chceme sečíst vektory a a b a výsledek uložit do vektoru c .

Je třeba najít v problému paralelismus.

Sériový součet vektorů:

```
for (int i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

Jednotlivé iterace cyklu jsou na sobě nezávislé – lze je paralelizovat, škáluje s velikostí vektoru.

Příklad – součet vektorů

Chceme sečíst vektory a a b a výsledek uložit do vektoru c .

Je třeba najít v problému paralelismus.

Sériový součet vektorů:

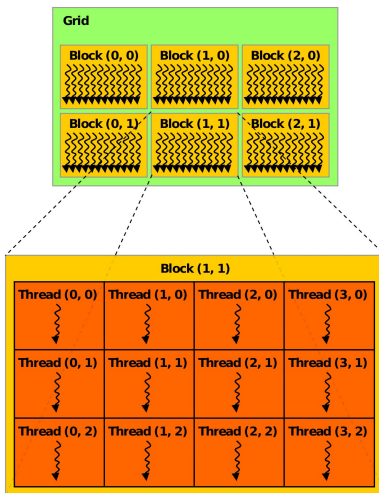
```
for (int i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

Jednotlivé iterace cyklu jsou na sobě nezávislé – lze je paralelizovat, škáluje s velikostí vektoru.
 i -tý thread sečte i -té složky vektorů:

```
c[i] = a[i] + b[i];
```

Jak zjistíme, kolikátý jsme thread?

Hierarchie vláken



Identifikace vlákna a bloku

C for CUDA obsahuje zabudované proměnné:

- **threadIdx.**{x, y, z} udává pozici threadu v rámci bloku
- **blockDim.**{x, y, z} udává velikost bloku
- **blockIdx.**{x, y, z} udává pozici bloku v rámci mřížky (z je vždy 1)
- **gridDim.**{x, y, z} udává velikost mřížky (z je vždy 1)

Příklad – součet vektorů

Vypočítáme tedy globální pozici threadu (mřížka i bloky jsou jednorozměrné):

Příklad – součet vektorů

Vypočítáme tedy globální pozici threadu (mřížka i bloky jsou jednorozměrné):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Příklad – součet vektorů

Vypočítáme tedy globální pozici threadu (mřížka i bloky jsou jednorozměrné):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Celá funkce pro paralelní součet vektorů:

```
__global__ void addvec(float *a, float *b, float *c){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    c[i] = a[i] + b[i];  
}
```

Příklad – součet vektorů

Vypočítáme tedy globální pozici threadu (mřížka i bloky jsou jednorozměrné):

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

Celá funkce pro paralelní součet vektorů:

```
__global__ void addvec(float *a, float *b, float *c){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    c[i] = a[i] + b[i];  
}
```

Funkce definuje tzv. kernel, při volání určíme, kolik threadů a v jakém uspořádání bude spuštěno.

Kvantifikátory typů funkcí

Syntaxe C je rozšířena o kvantifikátory, určující, kde se bude kód provádět a odkud půjde volat:

- **__device__** funkce je spouštěna na device (GPU), lze volat jen z device kódu
- **__global__** funkce je spouštěna na device, lze volat jen z host (CPU) kódu
- **__host__** funkce je spouštěna na host, lze ji volat jen z host
- kvantifikátory **__host__** a **__device__** lze kombinovat, funkce je pak kompilována pro obojí

Ke kompletnímu výpočtu je třeba:

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty
- **alokovat paměť na GPU**

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty
- **alokovat paměť na GPU**
- **zkopírovat vektory a a b na GPU**

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty
- **alokovat paměť na GPU**
- **zkopírovat vektory a a b na GPU**
- **spočítat vektorový součet na GPU**

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty
- **alokovat paměť na GPU**
- **zkopírovat vektory a a b na GPU**
- **spočítat vektorový součet na GPU**
- **uložit výsledek z GPU paměti do c**

Ke kompletnímu výpočtu je třeba:

- alokovat paměť pro vektory, naplnit je daty
- **alokovat paměť na GPU**
- **zkopírovat vektory a a b na GPU**
- **spočítat vektorový součet na GPU**
- **uložit výsledek z GPU paměti do c**
- použít výsledek v c :-)

Příklad – součet vektorů

CPU kód naplní a a b , vypíše c :

```
#include <stdio.h>
#define N 64
int main(){
    float a[N], b[N], c[N];
    for (int i = 0; i < N; i++)
        a[i] = b[i] = i;

    // zde bude kód provádějící výpočet na GPU

    for (int i = 0; i < N; i++)
        printf("%f, ", c[i]);
    return 0;
}
```

Správa GPU paměti

Paměť je třeba dynamicky alokovat.

```
cudaMalloc(void** devPtr, size_t count);
```

Alokuje paměť velikosti *count*, nastaví na ni ukazatel *devPtr*.
Uvolnění paměti:

```
cudaFree(void* devPtr);
```

Kopírování paměti:

```
cudaMemcpy(void* dst, const void* src, size_t count,
            enum cudaMemcpyKind kind);
```

Kopíruje *count* byte z *src* do *dst*, *kind* určuje, o jaký směr kopírování se jedná (např. *cudaMemcpyHostToDevice*, nebo *cudaMemcpyDeviceToHost*).

Příklad – součet vektorů

Alokujeme paměť a přeneseme data:

```
float *d_a, *d_b, *d_c;
cudaMalloc((void**)&d_a, N*sizeof(*d_a));
cudaMalloc((void**)&d_b, N*sizeof(*d_b));
cudaMalloc((void**)&d_c, N*sizeof(*d_c));

cudaMemcpy(d_a, a, N*sizeof(*d_a), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, N*sizeof(*d_b), cudaMemcpyHostToDevice);

// zde bude spuštěn kernel

cudaMemcpy(c, d_c, N*sizeof(*c), cudaMemcpyDeviceToHost);

cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

Příklad – součet vektorů

Spuštění kernelu:

- kernel voláme jako funkci, mezi její jméno a argumenty vkládáme do trojitých špičatých závorek velikost mřížky a bloku
- potřebujeme znát velikost bloků a jejich počet
- použijeme 1D blok i mřížku, blok bude pevné velikosti
- velikost mřížky vypočteme tak, aby byl vyřešen celý problém násobení vektorů

Pro vektory velikosti dělitelné 32:

```
#define BLOCK 32
addvec<<<N/BLOCK, BLOCK>>>(d_a, d_b, d_c);
```

Jak řešit problém pro obecnou velikost vektoru?

Příklad – součet vektorů

Upravíme kód kernelu:

```
__global__ void addvec(float *a, float *b, float *c, int n){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) c[i] = a[i] + b[i];
}
```

A zavoláme kernel s dostatečným počtem vláken:

```
addvec<<<N/BLOCK + 1, BLOCK>>>(d_a, d_b, d_c, N);
```

Příklad – spuštění

Nyní už zbývá jen kompilace :-).

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

Kde s CUDA pracovat?

- vlastní stroj: stáhněte a nainstalujte CUDA toolkit a SDK z developer.nvidia.com
- windowsí stanice v učebnách (titan)
- ke vzdálené práci s high-end GPU: na přání

Paměti lokální v rámci threadu

Registry

- nejrychlejší paměť, přímo využitelná v instrukcích
- lokální proměnné v kernelu i proměnné nutné pro mezivýsledky jsou automaticky v registrech
 - pokud je dostatek registrů
 - pokud dokáže kompilátor určit statickou indexaci polí
- mají životnost threadu (warpu)

Paměti lokální v rámci threadu

Registry

- nejrychlejší paměť, přímo využitelná v instrukcích
- lokální proměnné v kernelu i proměnné nutné pro mezivýsledky jsou automaticky v registrech
 - pokud je dostatek registrů
 - pokud dokáže kompilátor určit statickou indexaci polí
- mají životnost threadu (warpu)

Lokální paměť

- co se nevejde do registrů, jde do lokální paměti
- ta je fyzicky uložena v DRAM, je tudíž pomalá a má dlouhou latenci
- má životnost threadu (warpu)

Paměť lokální v rámci bloku

Sdílená paměť

- u c.c. 1.x rychlá jako registry
 - nedojde-li ke konfliktům paměťových bank
 - instrukce umí využít jen jeden operand ve sdílené paměti (jinak je třeba explicitní load/store)
- v C for CUDA deklarujeme pomocí `__shared__`
- proměnná ve sdílené paměti může mít dynamickou velikost (určenou při startu), pokud je deklarována jako *extern* bez udání velikosti pole
- má životnost bloku

Paměť lokální pro GPU

Globální paměť

- řádově nižší přenosová rychlost než u sdílené paměti
- latence ve stovkách GPU cyklů
- pro dosažení optimálního výkonu je třeba paměť adresovat zarovnaně
- má životnost aplikace
- u Fermi L1 cache (128 byte na řádek) a L2 cache (32 byte na řádek)

Lze dynamicky alokovat pomocí *cudaMalloc*, či staticky pomocí deklarace `__device__`

Ostatní paměti

- paměť konstant
- texturová paměť
- systémová paměť

Synchronizace v rámci bloku

- nativní bariérová synchronizace
 - musí do ní vstoupit všechny thready (pozor na podmínky!)
 - pouze jedna instrukce, velmi rychlá, pokud neredukuje paralelismus
 - v C for CUDA volání `__syncthreads()`
 - Fermi rozšíření: count, and, or

Atomické operace

- provádí read-modify-write operace nad sdílenou nebo globální pamětí
- žádná interference s ostatními thready
- pro celá 32-bitová či 64-bitová (pro compute capability ≥ 1.2) čísla (float add u c.c. ≥ 2.0)
- nad globální pamětí u zařízení s compute capability ≥ 1.1 , nad sdílenou c.c. ≥ 1.2
- aritmetické (Add, Sub, Exch, Min, Max, Inc, Dec, CAS) a bitové (And, Or, Xor) operace

Synchronizace paměťových operací

Kompilátor může optimalizovat operace se sdílenou/globální pamětí (mezivýsledky mohou zůstat v registrech) a může měnit jejich pořadí,

- chceme-li se ujistit, že jsou námi ukládaná data viditelná pro ostatní, používáme `__threadfence()`, popř. `__threadfence_block()`
- deklaruje-li proměnnou jako *volatile*, jsou veškeré přístupy k ní realizovány přes load/store do sdílené či globální paměti
 - velmi důležité pokud předpokládáme implicitní synchronizaci warpu

Synchronizace bloků

Mezi bloky

- globální paměť viditelná pro všechny bloky
- slabá nativní podpora synchronizace
 - žádná globální bariéra
 - u novějších GPU *atomické operace* nad globální pamětí
 - globální bariéru lze implementovat voláním kernelu (jiné řešení dosti trikové)
 - slabé možnosti globální synchronizace znesnadňují programování, ale umožňují velmi dobrou škálovatelnost

Globální synchronizace přes atomické operace

Problém součtu všech prvků vektoru

- každý blok sečte prvky své části vektoru
- poslední blok sečte výsledky ze všech bloků
 - implementuje slabší globální bariéru (po dokončení výpočtu u bloků $1..n - 1$ pokračuje pouze blok n)

```

__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
float* result) {
    float partialSum = calculatePartialSum(array, N);
    if (threadIdx.x == 0) {
        result[blockIdx.x] = partialSum;
        __threadfence();
        unsigned int value = atomicInc(&count, gridDim.x);
        isLastBlockDone = (value == (gridDim.x - 1));
    }
    __syncthreads();
    if (isLastBlockDone) {
        float totalSum = calculateTotalSum(result);
        if (threadIdx.x == 0) {
            result[0] = totalSum;
            count = 0;
        }
    }
}

```

Materiály

CUDA dokumentace (instalována s CUDA Toolkit, ke stažení na *developer.nvidia.com*)

- CUDA C Programming Guide (nejdůležitější vlastnosti CUDA)
- CUDA C Best Practices Guide (detailnější zaměření na optimalizace)
- CUDA Reference Manual (kompletní popis C for CUDA API)
- další užitečné dokumenty (manuál k nvcc, popis PTX jazyka, manuály knihoven, ...)

Série článků CUDA, Supercomputing for the Masses

- <http://www.ddj.com/cpp/207200659>

Dnes jsme si ukázali

- k čemu je dobré znát CUDA
- v čem jsou GPU jiná
- základy programování v C for CUDA

Příště se zaměříme na

- jak psát efektivní GPU kód