GPU hardware a paralelismus

Jiří Filipovič

podzim 2010

Alternativy k CUDA

CUDA je (a zřejmě i bude) pouze pro GPU nVidia.

Alternativy k CUDA

CUDA je (a zřejmě i bude) pouze pro GPU nVidia.
OpenCL

- standard, pro různé druhy akcelerátorů (nezávislý na výrobci HW i operačním systému)
- silně inspirováno CUDA, velmi snadný přechod

Alternativy k CUDA

CUDA je (a zřejmě i bude) pouze pro GPU nVidia.

OpenCL

- standard, pro různé druhy akcelerátorů (nezávislý na výrobci HW i operačním systému)
- silně inspirováno CUDA, velmi snadný přechod

DirectX compute

- GPU různých výrobců, OS pouze jednoho

Alternativy k CUDA

CUDA je (a zřejmě i bude) pouze pro GPU nVidia.

OpenCL

- standard, pro různé druhy akcelerátorů (nezávislý na výrobci HW i operačním systému)
- silně inspirováno CUDA, velmi snadný přechod

DirectX compute

- GPU různých výrobců, OS pouze jednoho

Brook(+)

- nadplatformní, + jen pro AMD/ATI
- pouze streamy

Proč se učíme o CUDA?

Proč CUDA a ne OpenCL

- v publikovaných výsledcích stále vyšší rychlost
- větší odladěnost prostředí
- největší množství aplikací
- největší množství knihoven
- největší počet publikací
- snadnější k naučení
- podobnost s OpenCL umožňuje snadný přechod
- PGI x86 CUDA kompilátor

Rozdíly v jednotlivých CUDA-GPU

Nové generace přinášejí vyšší výkon a výpočetní schopnosti.

- výpočetní schopnosti *compute capability* představují bohatost instrukční sady GPU a množství zdrojů (registry, současně běžící thready aj.)
- výkon roste se schopností umístit na jedno GPU více jader

V rámci generace se významně liší výkon.

- kvůli nabídce levnějších variant
- díky pozdějším změnám výrobního procesu
- kvůli spotřebě u mobilních GPU

Dnes dostupné GPU

Dnes dostupné GPU

- compute capability 1.0 - 2.1
  - s rozdíly se budeme postupně seznamovat
- 1 - 30 multiprocesorů (19.2 - 1 345.0 GFLOPs)
- frekvence 800 MHz - 1.836 GHz
- šířka a rychlost datové sběrnice (64bit - 512bit, 6.4 - 177 GB/s)

Dostupná řešení

Grafické karty GeForce

- mainstreamové řešení pro hráče
- levné, široce rozšířené, široké spektrum výkonu
- nevýhoda – omezená paměť (do 1.5 GB na GPU)

Profesionální karty Quadro

- z hlediska CUDA stejné jako GeForce
- paměť až 4 GB na GPU
- násobně dražší

Tesla

- řešení speciálně pro výpočty v CUDA
- jedno GPU na generaci (základní varianta), vždy velká paměť
- k dispozici jako karty do PCI-E, nebo jako samostatné multi-GPU stroje
- taktéž drahé, vhodné pro výpočetní centra či osobní superpočítače

Paralelismus GPU

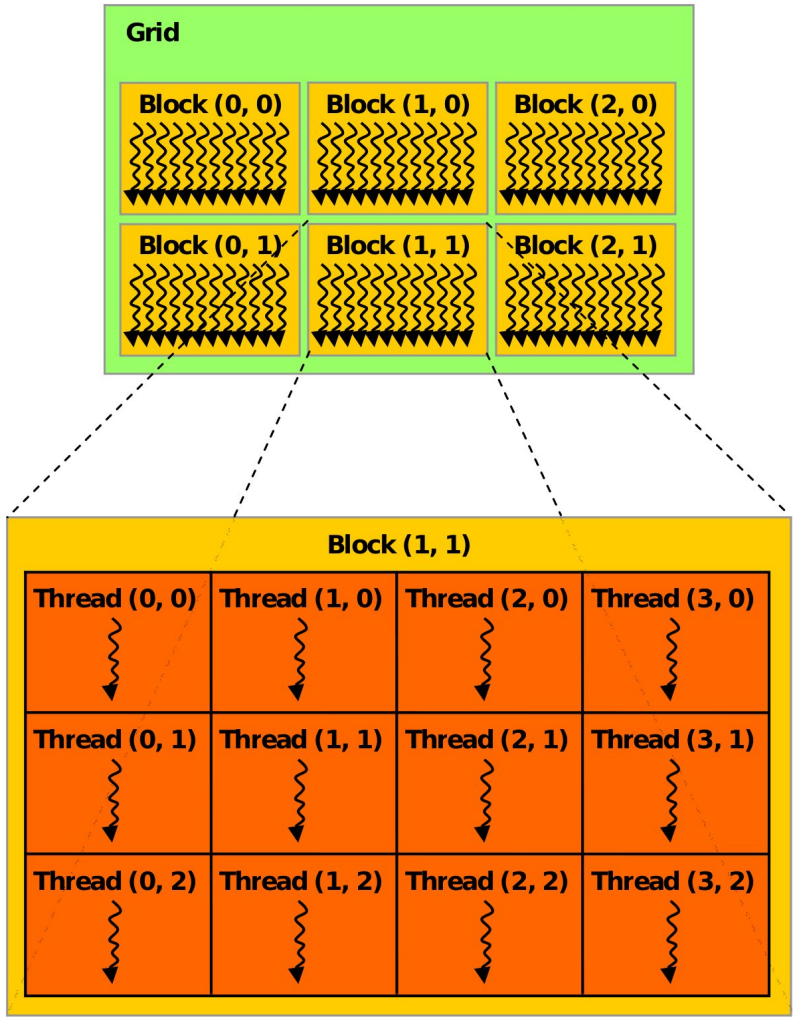
Paralelní algoritmy je nutno navrhovat vzhledem k paralelismu, které poskytuje HW

- v případě GPU se jedná o pole SIMT multiprocesorů pracující nad společnou pamětí

Dekompozice pro GPU

- hrubozrnné rozdělení problému na části nevyžadující intenzivní komunikaci/synchronizaci
- jemnozrnné rozdělení blízké vektorizaci (SIMT je ale více flexibilní)

Hierarchie vláken



SIMT

Multiprocesor má jen jednu jednotku pro spouštějící instrukce

- všech 8 SP musí provádět stejnou instrukci
- nová instrukce je spuštěna každé 4 cykly
- 32 thredů (tzv. *warp*) musí provádět stejnou instrukci

SIMT

Multiprocesor má jen jednu jednotku pro spouštějící instrukce

- všech 8 SP musí provádět stejnou instrukci
- nová instrukce je spuštěna každé 4 cykly
- 32 thredů (tzv. *warp*) musí provádět stejnou instrukci

A co větvení kódu?

- pokud část threadů ve warpu provádí jinou instrukci, běh se serializuje
- to snižuje výkon, snažíme se divergenci v rámci warpu předejít

SIMT

Multiprocesor má jen jednu jednotku pro spouštějící instrukce

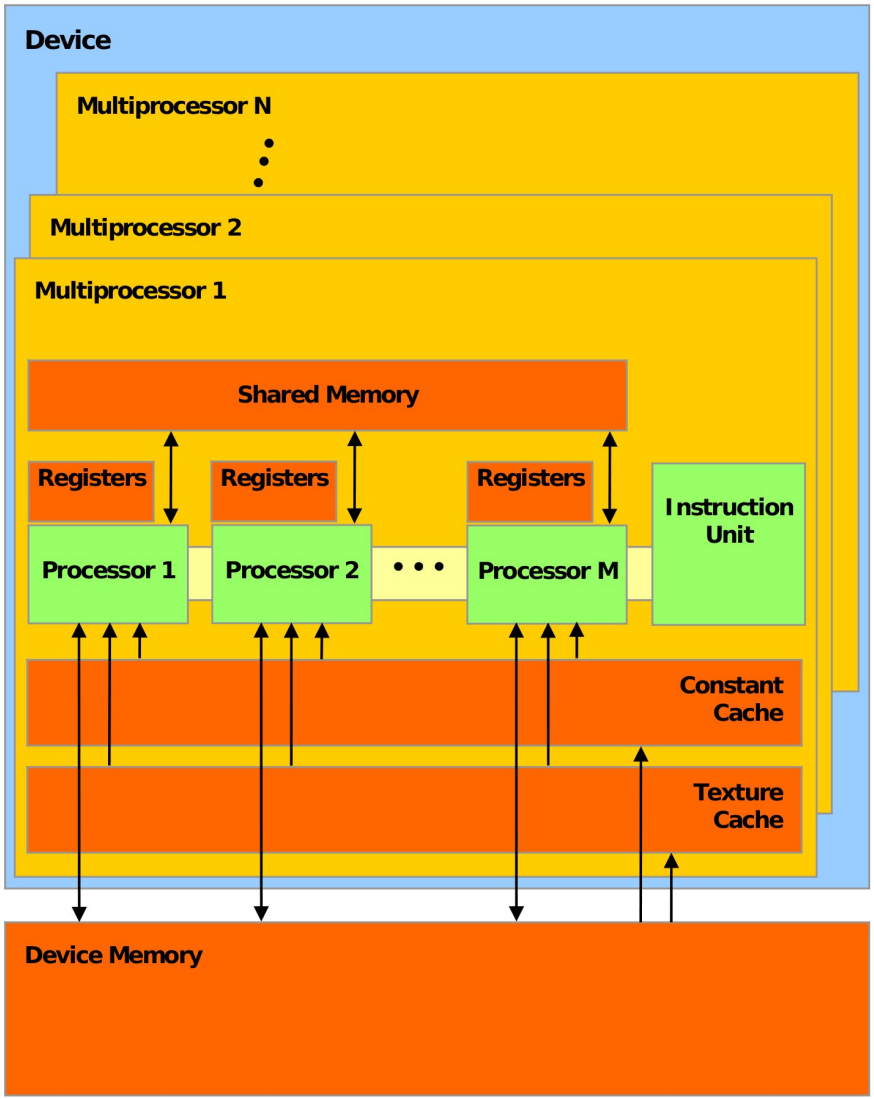
- všech 8 SP musí provádět stejnou instrukci
- nová instrukce je spuštěna každé 4 cykly
- 32 thredů (tzv. *warp*) musí provádět stejnou instrukci

A co větvení kódu?

- pokud část threadů ve warpu provádí jinou instrukci, běh se serializuje
- to snižuje výkon, snažíme se divergenci v rámci warpu předejít

Multiprocesor je tedy MIMD (Multiple-Instruction Multiple-Thread) z programátorského hlediska a SIMT (Single-Instruction Multiple-Thread) z výkonového.

GPU architektura



Vlastnosti threadů

Oproti CPU threadům jsou GPU thready velmi lehké (lightweight).

- jejich běh může být velmi krátký (desítky instrukcí)
- může (mělo by) jich být velmi mnoho
- nemohou využívat velké množství prostředků

Thready jsou seskupeny v blocích

- ty jsou spouštěny na jednotlivých multiprocesorech
- dostatečný počet bloků je důležitý pro škálovatelnost

Počet threadů a thread bloků na multiprocesor je omezen.

Maskování latence pamětí

Paměti mají latence

- globální paměť má vysokou latenci (stovky cyklů)
- registry a sdílená paměť mají read-after-write latenci

Maskování latence pamětí

Paměti mají latence

- globální paměť má vysokou latenci (stovky cyklů)
- registry a sdílená paměť mají read-after-write latenci

Maskování latence paměti je odlišné, než u CPU

- žádné provádění instrukcí mimo pořadí
- u většiny pamětí žádná cache

Maskování latence pamětí

Paměti mají latence

- globální paměť má vysokou latenci (stovky cyklů)
- registry a sdílená paměť mají read-after-write latenci

Maskování latence paměti je odlišné, než u CPU

- žádné provádění instrukcí mimo pořadí
- u většiny pamětí žádná cache

Když nějaký warp čeká na data z paměti, je možné spustit jiný

- umožní maskovat latenci paměti
- vyžaduje spuštění *řádově více* vláken, než má GPU jader
- plánování spuštění a přepínání threadů je realizováno přímo v HW bez overheadu

Maskování latence pamětí

Paměti mají latence

- globální paměť má vysokou latenci (stovky cyklů)
- registry a sdílená paměť mají read-after-write latenci

Maskování latence paměti je odlišné, než u CPU

- žádné provádění instrukcí mimo pořadí
- u většiny pamětí žádná cache

Když nějaký warp čeká na data z paměti, je možné spustit jiný

- umožní maskovat latenci paměti
- vyžaduje spuštění *řádově více* vláken, než má GPU jader
- plánování spuštění a přepínání threadů je realizováno přímo v HW bez overheadu

Obdobná situace je v případě synchronizace.

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 nevleze do registrů, jde do lokální paměti
- ta je fyzicky uložena v DRAM, je tudíš 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

Sdílená paměť

Deklarace statické sdílené paměti

```
__shared__ float myArray[128];
```

Dynamická alokace

```
extern __shared__ char myArray[];
float *array1 = (float*)myArray;
int *array2 = (int*)&array1[128];
short *array3 = (short*)&array2[256];
```

Vytvoří pole *array1* typu *float* velikosti 128, pole *array2* typu *int* velikosti 256 a pole *array3* plovoucí velikosti. Celkovou velikost je nutné specifikovat při spouštění kernelu.

```
myKernel<<<grid, block, n>>>();
```

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__*

Paměť lokální pro GPU

Paměť konstant

- pouze pro čtení
- cacheována
- cache-hit poskytuje rychlost jako registry (za dodržení určitých podmínek), cache-miss rychlý jako globální paměť
- omezená velikost (64 KB u v současné dobe dostupných GPU)
- má životnost aplikace

Paměť konstant

V deklaraci používáme *__constant__*, ke kopírování dat do paměti konstant slouží funkce

```
cudaError_t cudaMemcpyToSymbol(const char *symbol,
  const void *src, size_t count, size_t offset,
  enum cudaMemcpyKind kind)
```

Kopíruje data ze systémové (*cudaMemcpyHostToDevice*) nebo globální paměti (*cudaMemcpyDeviceToDevice*) z *src* do *symbol*. Kopírovaný blok má velikost *count* bytů, do paměti *symbol* kopírujeme s posuvem *offset*.

Paměť lokální pro GPU

Paměť textur

- cacheovaná, 2D prostorová lokalita
- pouze pro čtení (omezení kvůli cache-koherenci)
- dlouhá latence
- více adresovacích módů
  - možnost normalizace dimenzí do $[0, 1]$
  - při adresaci mimo rozsah ořezávání či přetékání koordinát
- možnost filtrace dat
  - lineární interpolace nebo nejbližší hodnota
- funkcionalita je ,,zdarma" (implementováno v HW)

Více detailů viz CUDA Programming Guide.

Paměť lokální pro systém

Systémová paměť RAM

- s GPU spojena přes PCI-E
- virtuální adresace komplikuje přenosy mezi CPU (host) a GPU (device) pamětí
- je možno alokovat tzv. page-locked oblast paměti
  - může redukovat celkový výkon systému
  - omezená velikost
  - data jsou po PCI-E přenášena rychleji
  - umožňuje paralelní běh kernelu a kopírování dat
  - umožňuje namapovat adresovací prostor host paměti na device
  - umožňuje *write-combining* přístup (data nejsou cacheována CPU)

Page-locked paměť

Namísto *malloc()* použijeme pro alokaci *cudaMallocHost()*, pro uvolnění *cudaFreeHost()*

- flag *cudaHostAllocPortable* zajistí vlastnosti page-locked paměti pro všechny (CPU) thready
- flag *cudaHostAllocWriteCombined* vypne cacheování alokované paměti CPU
- flag *cudaHostAllocMapped* nastaví mapování host paměti v device paměťovém prostoru

Page-locked paměť

Mapovaná paměť

- totéž místo má rozdílnou adresu pro device a host kód
- adresu pro device získáme pomocí *cudaHostGetDevicePointer()*
- před voláním ostatních CUDA API funkcí je zapotřebí zavolat *cudaSetDeviceFlags()* s flagem *cudaDeviceMapHost*

Asynchronní přenosy

- API funkce mají suffix *Async*
- může se překrývat přenos dat a výpočet na CPU i přenos dat a výpočet na GPU (podrobněji až probereme streamy)

Necacheovaná paměť

- pomalé čtení z host kódu
- rychlejší přístup z device paměti
- ,,nevylévá" CPU cache

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
- komunikace přes sdílenou paměť
  - thready si přes ni mohou vyměňovat data
  - synchronizace atomickými operacemi, nebo bariérou

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

Hlasování warpu

Všechny thready v jednom warpu vyhodnocují podmínku a provedou její srovnání.
Dostupné u zařízení s c.c. ≥ 1.2.

```
int __all(int predicate);
```

Nabývá nenulové hodnoty tehdy a jen tehdy když je nenulový predikát pro všechny thready ve warpu.

```
int __any(int predicate);
```

Nebývá nenulové hodnoty tehdy a jen tehdy když alespoň jeden thread ve warpu vyhodnotí predikát jako nenulový.

```
unsigned int __ballot(int predicate);
```

Obsahuje bitovou masku hlasování jednotlivých threadů.

Synchronizace paměťových operací

Sdílenou paměť obvykle využíváme ke komunikaci mezi thready a nebo jako cache pro data užívaná více thready.

- thready využívají data uložené jinými thready
- je třeba zajistit, abychom nečetli data, která ještě nejsou k dispozici
- chceme-li počkat, až jsou data k dispozici, používáme *__syncthreads()*

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()*
- deklarujeme-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
- globální bariéra
- jeden blok sečte výsledky se všech bloků

```
__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;
    }
  }
}
```

Násobení matic

Chceme vynásobit matice A a B a výsledek uložit do C. Pro jednoduchost uvažujme matice velikosti $n \times n$.

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

Zápis v jazyce C

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++){
    C[i*n + j] = 0.0;
    for (int k = 0; k < n; k++)
      C[i*n + j] += A[i*n + k] * B[k*n + j];
  }
```

Paralelizace

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++){
    C[i*n + j] = 0.0;
    for (int k = 0; k < n; k++)
      C[i*n + j] += A[i*n + k] * B[k*n + j];
  }
```

Problém lze paralelizovat více způsoby

- vybrat jeden z cyklů
- vybrat dva z cyklů
- paralelizovat všechny cykly

Paralelizace

Paralelizace přes jeden cyklus

- neškáluje dobře, nutno používat velké matice (nezapomínejme, pro dobré využití GPU potřebujeme tisíce threadů)

Paralelizace

Paralelizace přes jeden cyklus

- neškáluje dobře, nutno používat velké matice (nezapomínejme, pro dobré využití GPU potřebujeme tisíce threadů)

Paralelizace přes dva cykly

- z hlediska škálování se zdá dobrá, počet vláken roste kvadraticky vzhledm k n

Paralelizace

Paralelizace přes jeden cyklus

- neškáluje dobře, nutno používat velké matice (nezapomínejme, pro dobré využití GPU potřebujeme tisíce threadů)

Paralelizace přes dva cykly

- z hlediska škálování se zdá dobrá, počet vláken roste kvadraticky vzhledm k n

Paralelizace přes vnitřní cyklus

- nevhodná, nutná serializace zápisu do C!

Paralelizace

Paralelizace přes jeden cyklus

- neškáluje dobře, nutno používat velké matice (nezapomínejme, pro dobré využití GPU potřebujeme tisíce threadů)

Paralelizace přes dva cykly

- z hlediska škálování se zdá dobrá, počet vláken roste kvadraticky vzhledm k n

Paralelizace přes vnitřní cyklus

- nevhodná, nutná serializace zápisu do C!

Jako nejvhodnější se tedy jeví paralelizovat cykly jdoucí přes i a j.

První kernel

S výhodou můžeme využít možnosti uspořádání bloku a mřížky jako 2D pole.

```
__global__ void mmul(float *A, float *B, float *C, int n){
  int x = blockIdx.x*blockDim.x + threadIdx.x;
  int y = blockIdx.y*blockDim.y + threadIdx.y;

  float tmp = 0;
  for (int k = 0; k < n; k++)
    tmp += A[y*n+k] * B[k*n+x];

  C[y*n + x] = tmp;
}
```

Povšimněte si nápadné podobnosti s matematickým zápisem – paralelní verze je intuitivnější, než sériová!

Výkon

Jaký bude mít naše implementace výkon?

Výkon

Jaký bude mít naše implementace výkon?
Uvažujme kartu GeForce GTX 280

- pro problém násobení matic využitelných 622 GFlops
- propustnost paměti 142 GB/s

Výkon

Jaký bude mít naše implementace výkon?
Uvažujme kartu GeForce GTX 280

- pro problém násobení matic využitelných 622 GFlops
- propustnost paměti 142 GB/s

Flop-to-word ratio naší implementace

- v jednom kroku cyklu přes k načítáme 2 floaty (jedno číslo z matice A, jedno z B) a provádíme dvě aritmetické operace
- jedna aritmetická operace připadne na přenos jednoho floatu (4 bytů)
- globální paměť má propustnost 35.5 miliardy floatů za sekundu, pokud jeden warp přenáší jeden float z jedné matice a 16 floatů z druhé, můžeme dosáhnout výkonu cca 66.8 GFlops
- 66.8 GFlops je velmi daleko od 622 GFlops

Co s tím?

Narazili jsme na výkon globální paměti. GPU poskytuje rychlejší paměti, můžeme je využít?

Co s tím?

Narazili jsme na výkon globální paměti. GPU poskytuje rychlejší paměti, můžeme je využít?
Pro výpočet jednoho prvku C musíme načíst řádek z A a sloupec z B, které jsou v globální paměti.

Co s tím?

Narazili jsme na výkon globální paměti. GPU poskytuje rychlejší paměti, můžeme je využít?
Pro výpočet jednoho prvku C musíme načíst řádek z A a sloupec z B, které jsou v globální paměti.
Je opravdu nutné dělat to pro každý prvek C zvlášť?

- pro všechny prvky C ve stejném řádku načítáme stejný řádek A
- pro všechny prvky C ve stejném sloupci načítáme stejný sloupec B
- můžeme některá data načíst jednou z globální paměti do sdílené a následně je opakovaně číst z rychlejší sdílené paměti

Přístup po blocích

Budeme-li přistupovat k matici po blocích, můžeme amortizovat přenosy z globální paměti:

- počítáme část matice C o velikosti $a \times a$
- iterativně načítáme bloky stejné velikosti z matic A a B do sdílené paměti
- tyto bloky spolu násobíme a výsledek přičítáme do C
- poměr aritmetických operací k přeneseným floatům je a-násobný

Přirozené mapování na paralelismus v GPU

- jednotlivé bloky threadů budou počítat bloky matice C
- mají společnou sdílenou paměť
- rychle se synchronizují
- mezi bloky není třeba žádné synchronizace

Přístup po blocích

Jak velké bloky zvolit?

- jsme omezeni velikostí sdílené paměti
- jsme omezeni počtem threadů, který může běžet na GPU
- necháme-li jeden thread počítat jeden prvek C, jeví se jako rozumná velikost bloku 16×16
  - jedná se o násobek velikosti warpu
  - jeden blok bude mít únosných 256 threadů
  - jeden blok spotřebuje 2 KB sdílené paměti
  - paměť nebude zásadně omezovat výkon ($16 \cdot 25.5 = 568$ GFlops, což je již poměrně blízko hodnotě 622 GFlops)

Algoritmus

Schéma algoritmu

- každý blok threadů bude mít pole *As* a *Bs* ve sdílené paměti
- iterativně se budou násobit bloky matic *A* a *B*, výsledek bude každý thread kumulovat v proměnné *Csub*
  - thready v bloku společně načtou bloky do *As* a *Bs*
  - každý thread vynásobí bloky v *As* a *Bs* pro jeho prvek výsledné matice v *Csub*
- každý thread uloží jeho prvek matice do globální paměti *C*

Pozor na synchronizaci

- než začneme násobit jednotlivé bloky, musí být kompletně načteny
- než začneme znovunačítat bloky, musí být dokončeno násobení s původními daty

Druhý kernel

```
__global__ void mmul(float *A, float *B, float *C, int n){
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

  float Csub = 0.0f;
  for (int b = 0; b < n/BLOCK_SIZE; b++){
    As[ty][tx] = A[(ty + by*BLOCK_SIZE)*n + b*BLOCK_SIZE+tx];
    Bs[ty][tx] = B[(ty + b*BLOCK_SIZE)*n + bx*BLOCK_SIZE+tx];
    __syncthreads();

    for (int k = 0; k < BLOCK_SIZE; k++)
      Csub += As[ty][k]*Bs[k][tx];
    __syncthreads();
  }

  C[(ty + by*BLOCK)*n + bx*BLOCK_SIZE+tx] = Csub;
}
```

Výkon

- teoretické omezení první verze kernelu je 66.8 GFlops, naměřený výkon 36.6 GFlops
- teoretické omezení druhé verze kernelu je 568 GFlops, naměřený výkon 198 GFlops
- jak se přiblížit maximálnímu výkonu karty?
- je třeba znát podrobněji HW a jeho omezení a podle toho navrhovat algoritmy
- látka na další přednášku :-)