Úvod, základy CUDA

Jiří Filipovič

podzim 2012

Co se budeme učit?

Předmět se zabývá návrhem algoritmů a programováním obecných výpočtů na vektorových multiprocesorech.

Co se budeme učit?

Předmět se zabývá návrhem algoritmů a programováním obecných výpočtů na vektorových multiprocesorech.
Primárně se zaměříme na CUDA GPU

- GPU jsou zajímavé z hlediska rychlosti a rozšířenosti
- C for CUDA vhodné k výuce (jednoduché API, množství materiálů, zdrojových kódů, odladěnost prostředí)
- omezení na GPU od nVidie a x86 CPU

Co se budeme učit?

Předmět se zabývá návrhem algoritmů a programováním obecných výpočtů na vektorových multiprocesorech.
Primárně se zaměříme na CUDA GPU

- GPU jsou zajímavé z hlediska rychlosti a rozšířenosti
- C for CUDA vhodné k výuce (jednoduché API, množství materiálů, zdrojových kódů, odladěnost prostředí)
- omezení na GPU od nVidie a x86 CPU

Ukážeme si také základy OpenCL

- z CUDA poměrně snadný přechod
- kompilovatelné pro nVidia a AMD GPU, x86 CPU i Cell
- probereme specifika optimalizace pro AMD GPU a x86 CPU

Co se budeme učit?

Předmět se zabývá návrhem algoritmů a programováním obecných výpočtů na vektorových multiprocesorech.
Primárně se zaměříme na CUDA GPU

- GPU jsou zajímavé z hlediska rychlosti a rozšířenosti
- C for CUDA vhodné k výuce (jednoduché API, množství materiálů, zdrojových kódů, odladěnost prostředí)
- omezení na GPU od nVidie a x86 CPU

Ukážeme si také základy OpenCL

- z CUDA poměrně snadný přechod
- kompilovatelné pro nVidia a AMD GPU, x86 CPU i Cell
- probereme specifika optimalizace pro AMD GPU a x86 CPU

Předmět je prakticky orientován – paralelní výpočet je na jednom procesoru konstantně-krát rychlejší než sekvenční, vedle časové složitosti nás tedy zajímá, jak napsat efektivní kód.

Co se budeme učit?

Budeme probírat:

- návrh datově-paralelních algoritmů s důrazem na užití v GPU

Co se budeme učit?

Budeme probírat:

- návrh datově-paralelních algoritmů s důrazem na užití v GPU
- architekturu GPU založených na CUDA

Co se budeme učit?

Budeme probírat:

- návrh datově-paralelních algoritmů s důrazem na užití v GPU
- architekturu GPU založených na CUDA
- programování v C for CUDA

Co se budeme učit?

Budeme probírat:

- návrh datově-paralelních algoritmů s důrazem na užití v GPU
- architekturu GPU založených na CUDA
- programování v C for CUDA
- nástroje a knihovny

Co se budeme učit?

Budeme probírat:

- návrh datově-paralelních algoritmů s důrazem na užití v GPU
- architekturu GPU založených na CUDA
- programování v C for CUDA
- nástroje a knihovny
- optimalizaci kódu pro GPU

Co se budeme učit?

Budeme probírat:

- návrh datově-paralelních algoritmů s důrazem na užití v GPU
- architekturu GPU založených na CUDA
- programování v C for CUDA
- nástroje a knihovny
- optimalizaci kódu pro GPU
- přechod k OpenCL

Co se budeme učit?

Budeme probírat:

- návrh datově-paralelních algoritmů s důrazem na užití v GPU
- architekturu GPU založených na CUDA
- programování v C for CUDA
- nástroje a knihovny
- optimalizaci kódu pro GPU
- přechod k OpenCL
- specifika optimalizace pro AMD Evergreen a Intel x86

Co se budeme učit?

Budeme probírat:

- návrh datově-paralelních algoritmů s důrazem na užití v GPU
- architekturu GPU založených na CUDA
- programování v C for CUDA
- nástroje a knihovny
- optimalizaci kódu pro GPU
- přechod k OpenCL
- specifika optimalizace pro AMD Evergreen a Intel x86
- případové studie

Co od vás budeme vyžadovat

Během semestru budete vypracovávat prakticky orientovaný projekt

- významný podíl na výsledném hodnocení
- jednotné zadání, budeme srovnávat rychlost implementací
- celkem 50 + 30 bodů z výsledné známky
  - funkční kód: 25 bodů
  - efektivní implementace: 25 bodů
  - umístění ve srovnání se spolužáky: 30 bodů (pouze ke zlepšení známky)

Zkouška (ústní či písemná dle počtu studentů)

- 50 bodů

Hodnocení

Zakončení zkouškou

- A: 92-100
- B: 86-91
- C: 78-85
- D: 72-77
- E: 66-71
- F: 0 - 65 bodů

Zakončení kolokviem

- 50 bodů

Materiály – CUDA

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

Materiály – OpenCL

- OpenCL 1.1 Specification
- AMD Accelerated Parallel Processing Programming Guide
- Intel OpenCL SDK Programming Guide
- Writing Optimal OpenCL Code with Intel OpenCL SDK

Materiály – paralelní programování

- Ben-Ari M., Principles of Concurrent and Distributed Programming, 2nd Ed. Addison-Wesley, 2006
- Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, Patterns for Parallel Programming, Addison-Wesley, 2004

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 pamětmi 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

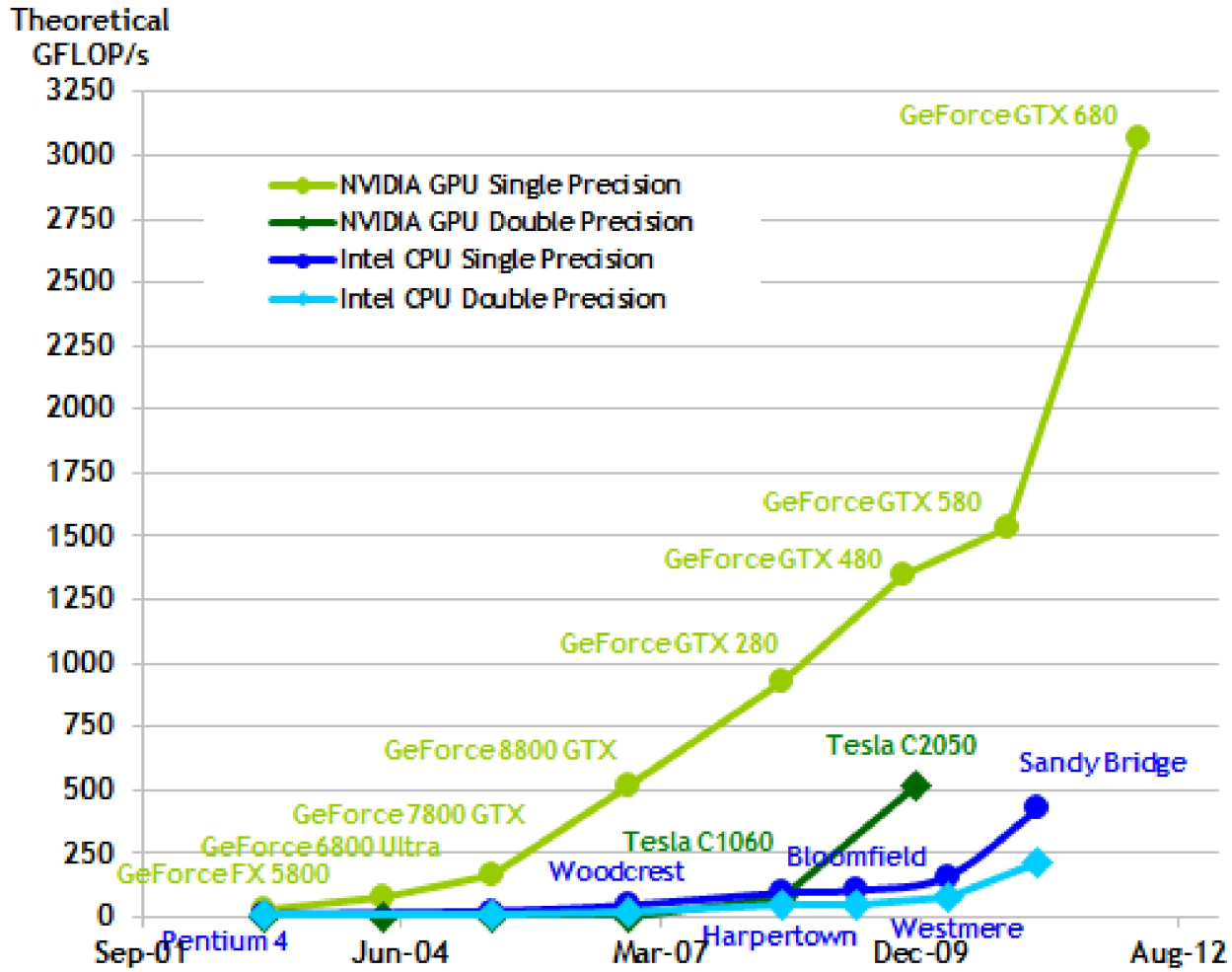
Z pohledu programátora

- rozdílné paradigma znamená rozdílný pohled na návrh algoritmů
- některé problémy jsou spíše datově paralelní, některé úlohově

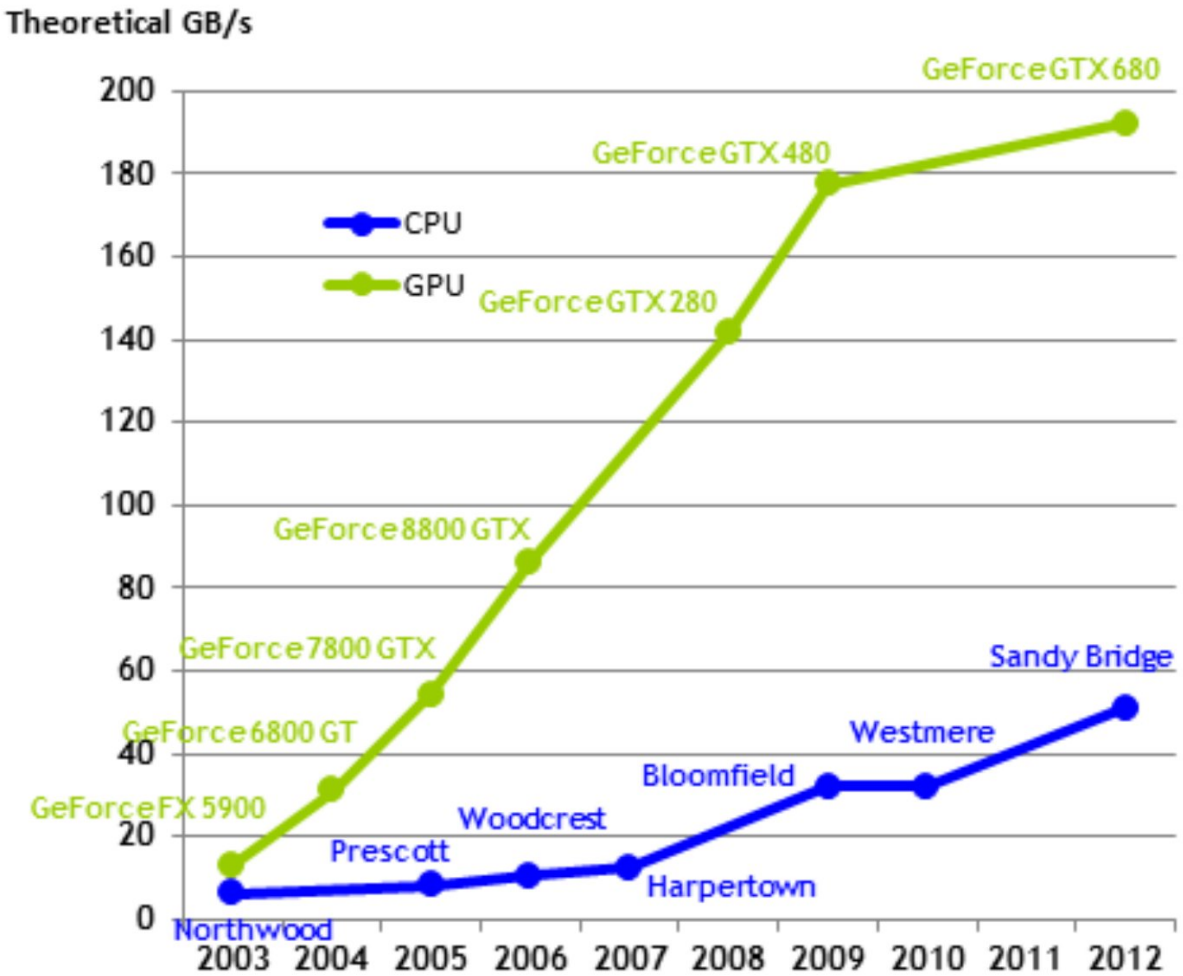
Z pohledu vývojáře hardware

- 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 – výkon GPU



Motivace – výkon GPU



Motivace – výkon CPU

Dnešní CPU mají více jader a vektorové instrukce.

Intel Core2 Quad

- 4 jádra, SSE instrukce (128-bitové vektory)
- sekvenční výpočet 16× pomalejší než vektorový vícevláknový

Intel Sandy Bridge

- 8 jader, AVX instrukce (256-bitové vektory)
- sekvenční výpočet až 64× pomalejší než vektorový vícevláknový

Motivace – shrnutí

GPU jsou výkonné

- řádový nárůst výkonu již stoji za studium nového programovacího modelu

Pro plné využití moderních GPU i CPU je třeba programovat paralelně

- desetinásobky rychlosti při plném využití CPU, stonásobky při akceleraci na GPU
- paralelizace pro GPU je obecně podobně náročná jako paralelizace a vektorizace pro CPU

GPU jsou široce rozšířené

- jsou levné
- spousta uživatelů má na stole ,,superpočítač"

Motivace – uplatnění

Využití akcelerace je dynamicky se rozvíjející oblast s širokou škálou aplikací

Motivace – uplatnění

Využití akcelerace je dynamicky se rozvíjející oblast s širokou škálou aplikací

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

Motivace – uplatnění

Využití akcelerace je dynamicky se rozvíjející oblast s širokou škálou aplikací

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

Motivace – uplatnění

Vývojáři SW jsou stále ještě nedostatkovým zbožím...

Motivace – uplatnění

Vývojáři SW jsou stále ještě nedostatkovým zbožím...
Vývojáři schopní psát paralelní SW jsou extrémně nedostatkovým zbožím!

Motivace – uplatnění

Vývojáři SW jsou stále ještě nedostatkovým zbožím...
Vývojáři schopní psát paralelní SW jsou extrémně nedostatkovým zbožím!
Obrovské množství existujícího software není paralelní.

- pro zajištění růstu výkonu je jej třeba paralelizovat
- někdo to musí udělat :-)

Historická exkurze

- SIMD model od 60. let
  - projekt Solomon u firmy Westinghouse na začátku 60. let
  - posléze předán na University of Illinios jako ILLIAC IV
  - separátní ALU pro každý datový element – masivně paralelní
  - původní plán: 256 ALUs, 1 GFLOPS
  - dokončen v r. 1972, 64 ALUs, 100–150 MFLOPS
- v 80.–90. letech: běžné vektorové superpočítače, TOP500
- v dnešních CPU: SSE (x86), ActiVec (PowerPC)
- Cg: programování vertex a pixel shaderů na grafických kartách (cca 2003)
- CUDA: obecné programování GPU, SIMT model (uvolněno poprvé 15. února 2007)
- budoucnost??
  - OpenCL
  - vyšší programovací jazyky, automatická paralelizace

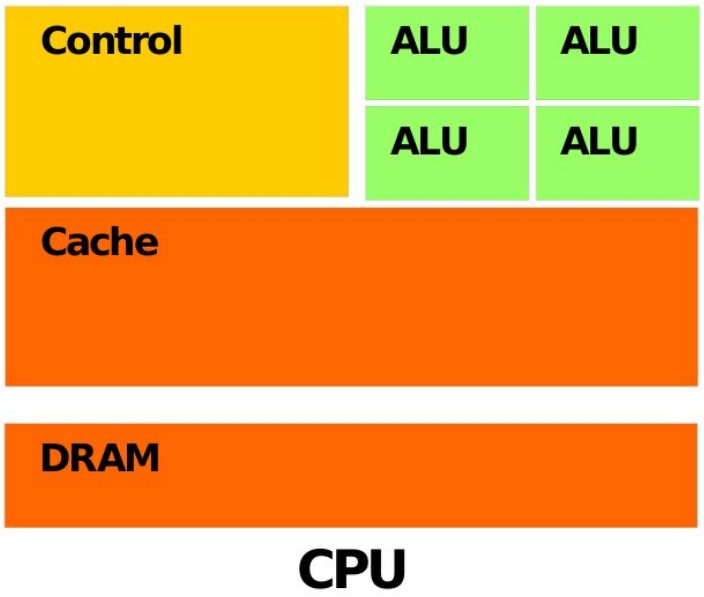
Architektura GPU

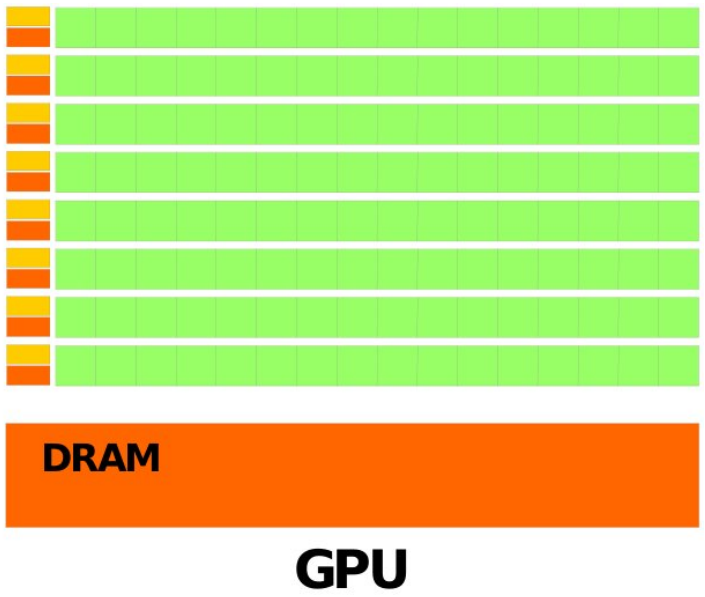
CPU vs. GPU

- 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





Architektura GPU

V rámci systému:

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

Procesor G80

G80

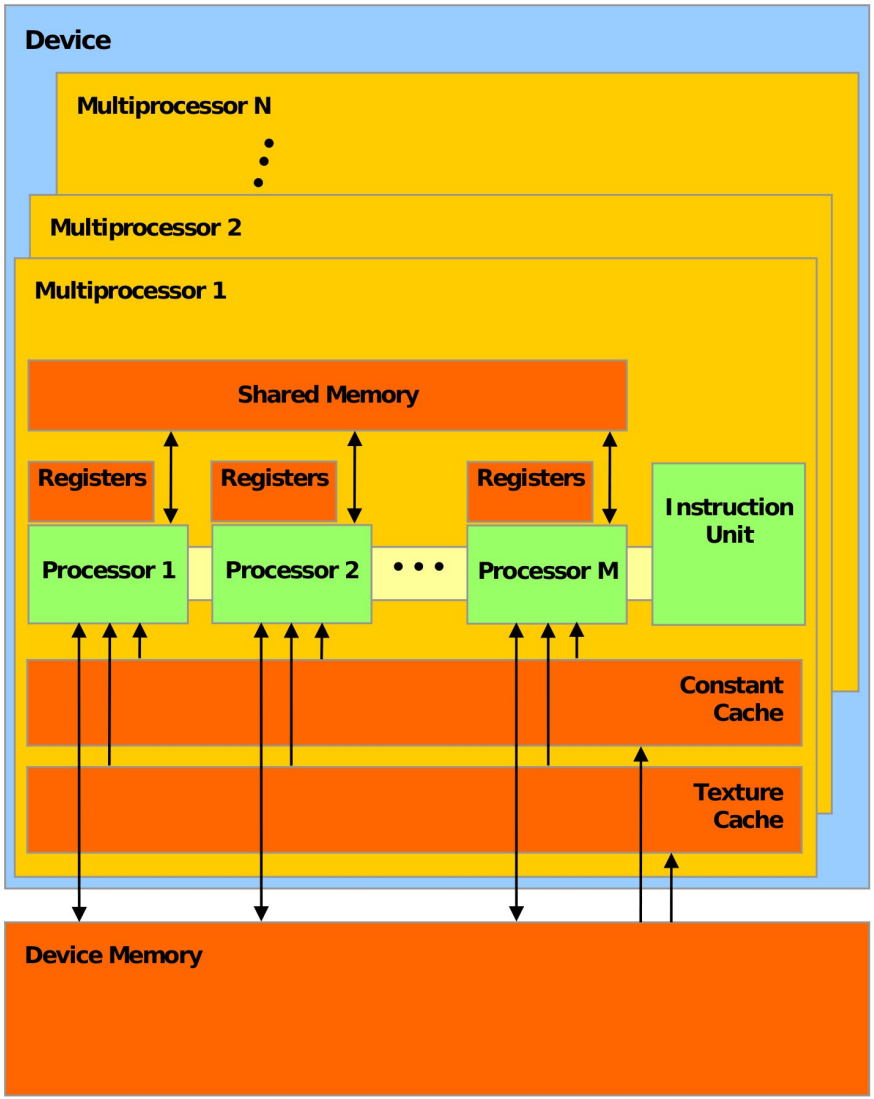
- první CUDA procesor
- obsahuje 16 multiprocesorů
- multiprocesor
  - 8 skalárních procesorů
  - 2 jednotky pro speciální funkce
  - až 768 threadů
    - HW přepínání a plánování threadů
  - thready organizovány po 32 do warpů
    - SIMT
  - nativní synchronizace v rámci multiprocesoru

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

Procesor G80



Další vývoj

Procesory odvozené od G80

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

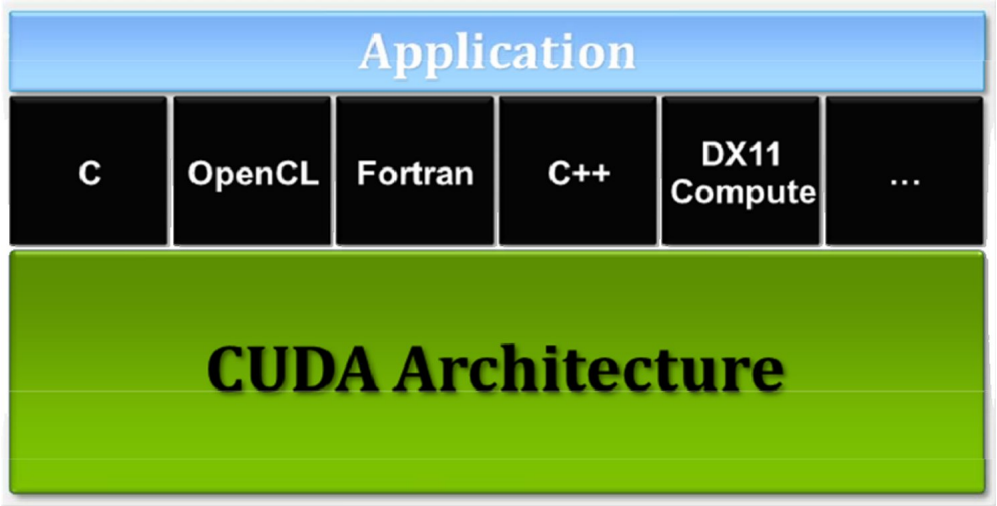
Fermi

- 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

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

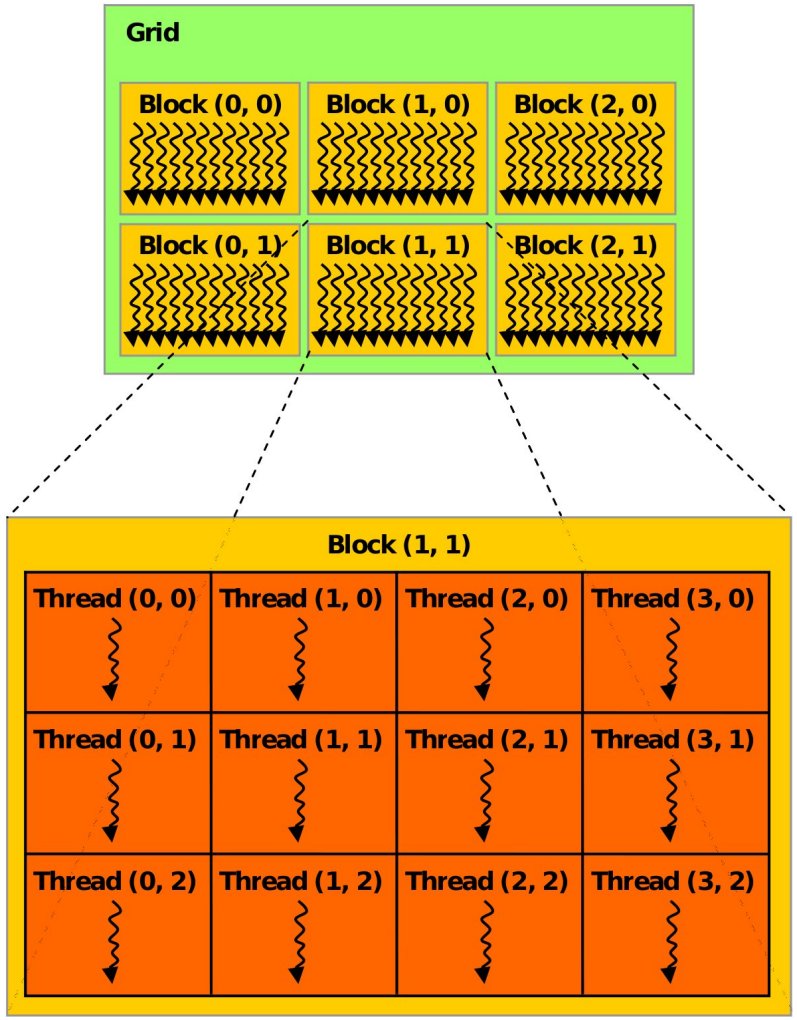
- explicitně oddělen host (CPU) a device (GPU) kód
- hierarchie vláken
- hierarchie pamětí
- synchronizační mechanismy
- API

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

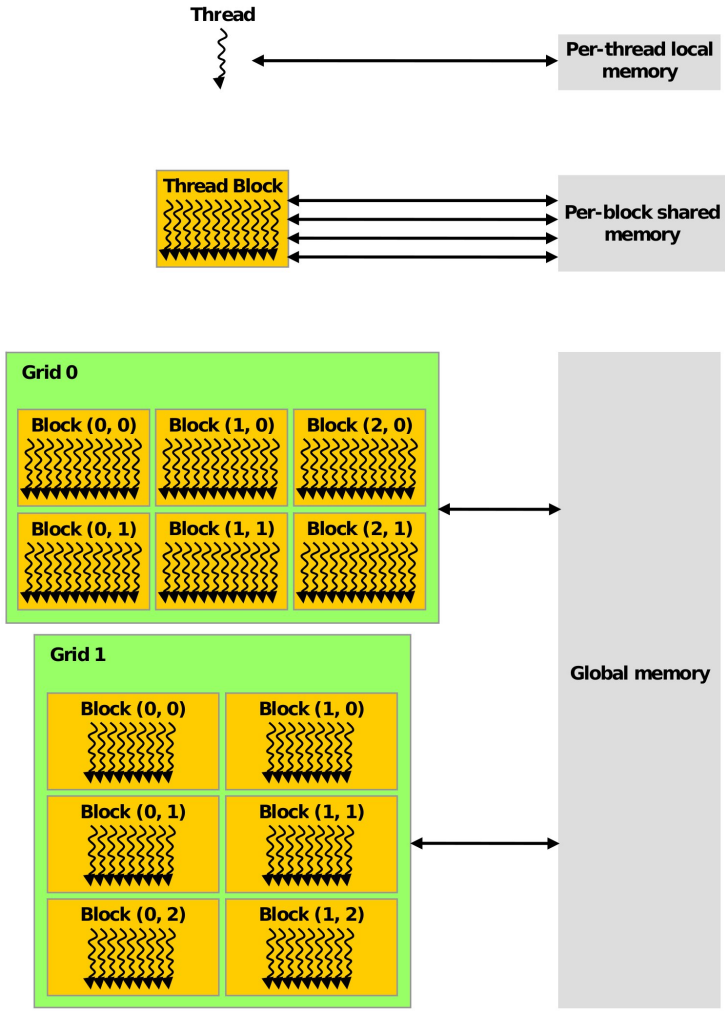


Hierarchie pamětí

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:

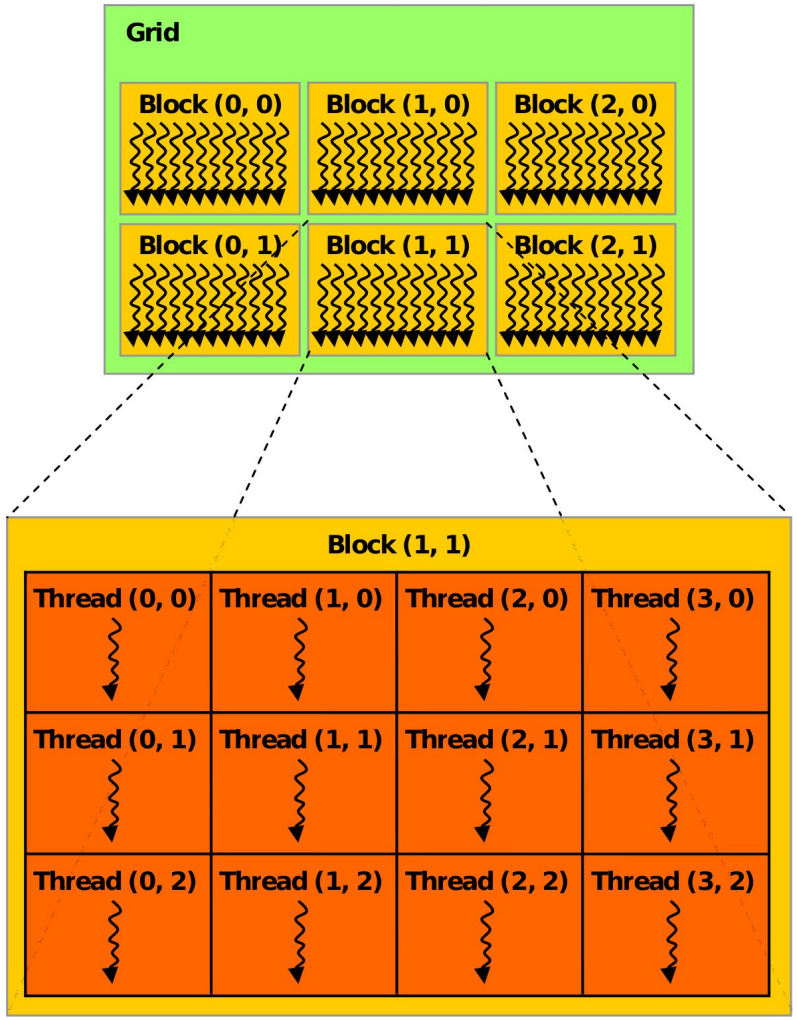
```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

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?

- ke vzdálenému připojení: airacuda.fi.muni.cz, účty budou vytvořeny
- windowsí stanice v učebnách (bude upřesněno)
- vlastní stroj: stáhněte a nainstalujte CUDA toolkit a SDK z developer.nvidia.com
- zdrojové kódy probírané v přednáškách budeme vystavovat ve studijních materiálech

Dnes jsme si ukázali

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

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

- podrobnější seznámení s GPU z hadrwarového hlediska
- paralelismus, který nám GPU poskytuje
- paměti dostupné pro GPU
- ukážeme si složitější příklad GPU implementace

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

- podrobnější seznámení s GPU z hadrwarového hlediska
- paralelismus, který nám GPU poskytuje
- paměti dostupné pro GPU
- ukážeme si složitější příklad GPU implementace

K samostatné práci

- zkuste si přeložit první CUDA program
- máte-li chuť, experimentujte s ním!