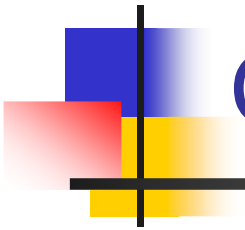
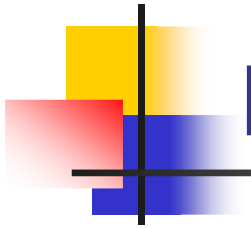


GPGPU

OpenCL



Motivace

- Řešíme úlohu zpracování velkého množství dat
  - Data jsou symetrická, úloha je dobře paralelizovatelná
- Propaganda výrobců grafických karet:

  „Vezměte váš C-čkový kód, zkompilujte a pusťte jej na grafické kartě a ono to funguje!“

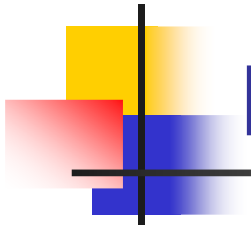
© 2010, Martin Kruliš

Motivace

- Řešíme úlohu zpracování velkého množství dat
  - Data jsou symetrická, úloha je dobře paralelizovatelná
- Propaganda výrobců grafických karet:

  „Vezměte váš C-čkový kód, zkompilujte a pusťte jej na grafické kartě a ono to funguje!“
- Realita je trochu jiná …

© 2010, Martin Kruliš



Historie

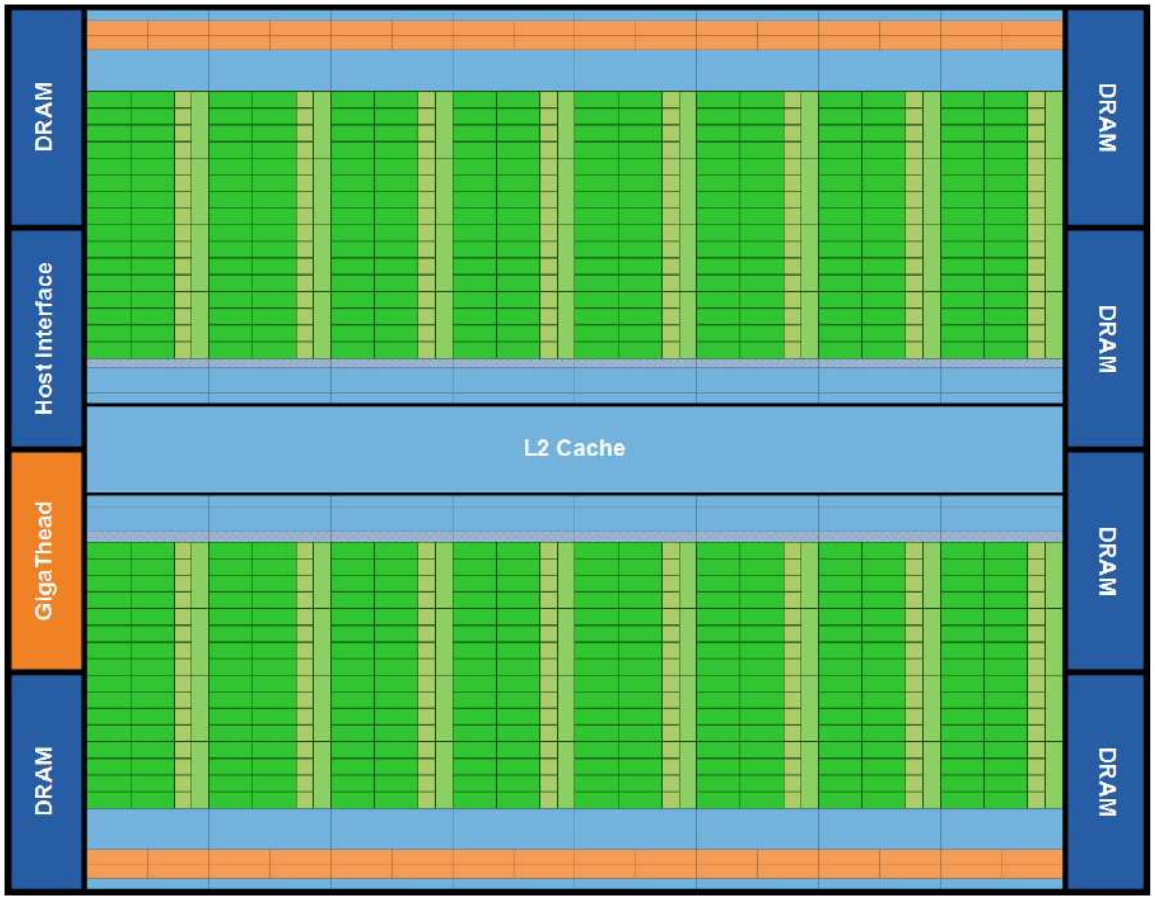
- 1996: 3Dfx Voodoo 1
  - První grafický akcelerátor do domácího počítače
- 1999: NVIDIA GeForce 256
  - První HW T&L jednotka („transform & lightning”)
- 2000: NVIDIA GeForce2, ATI Radeon
- 2001: programování GPU
  - DirectX 8 (vertex a fragment shaders v1.0 a v1.1)
- 2006: OpenGL 2.0, DirectX 10, Windows Vista
  - **Unifikované shader procesory**, geometry shader
- 2007: NVIDIA CUDA
  - První GPGPU řešení
- 2009: OpenCL, DirectCompute

© 2010, Martin Kruliš

Hardware

- NVIDIA Fermi
  - Nastupující state-of-the-art
  - 768kB L2 cache
  - 16 SMP jednotek s 512 CUDA cores

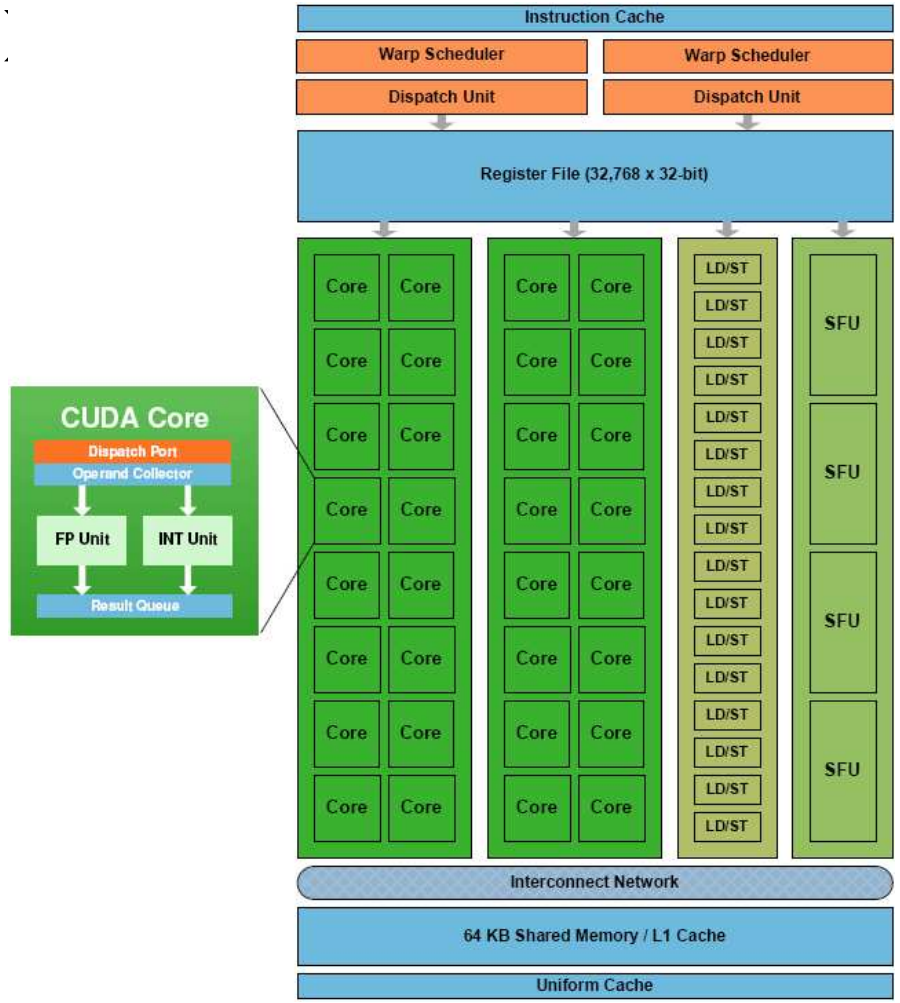
Poznámka:
1 CUDA core
~
1 5D ATI stream processor
(Radeon 5870 má 320 jader)



© 2010, Martin Kruliš

Hardware

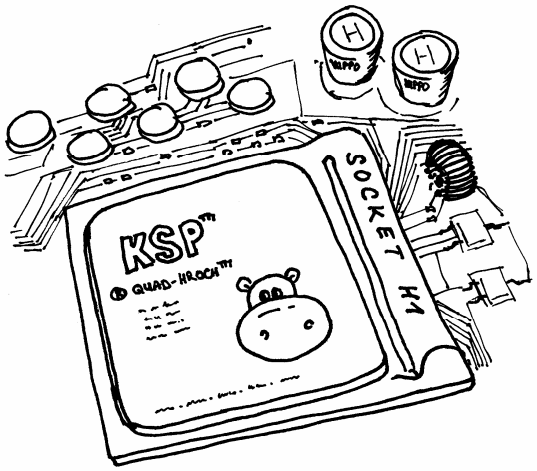
- Streaming Multiprocessor (Fermi)
  - 32 CUDA jader
  - 64K sdílené paměti nebo L1 cache
  - 1024 registrů na jádro
  - 16 load/store jednotek
  - 4 jednotky speciálních funkcí
  - 16 double-precision operací za takt



© 2010, Martin Kruliš

CPU vs. GPU

- CPU
  - Malý počet jader
  - Jádra pro obecné výpočty
  - Na různých jádrech běží různá vlákna
  - Latenci přístupů do globální paměti redukuje cache
    - Problém „Locality of Reference“

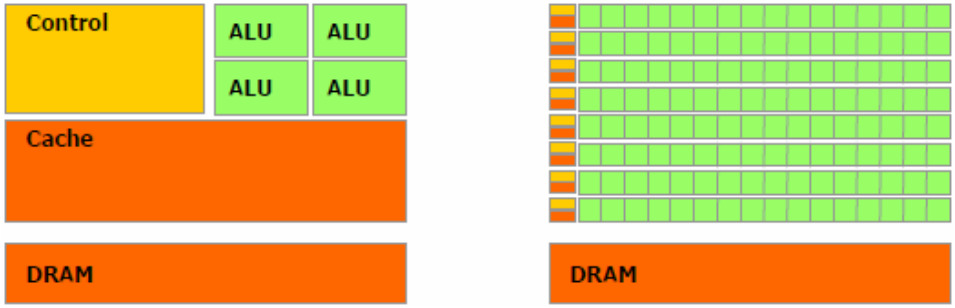


- GPU
  - Velký počet jader
  - Jádra specializovaná pro numerické výpočty
  - SIMT zpracování vláken
  - Latenci přístupů redukuje rychlé přepínání mezi vlákny
    - Komplikovanější důsledky přístupu do globální paměti

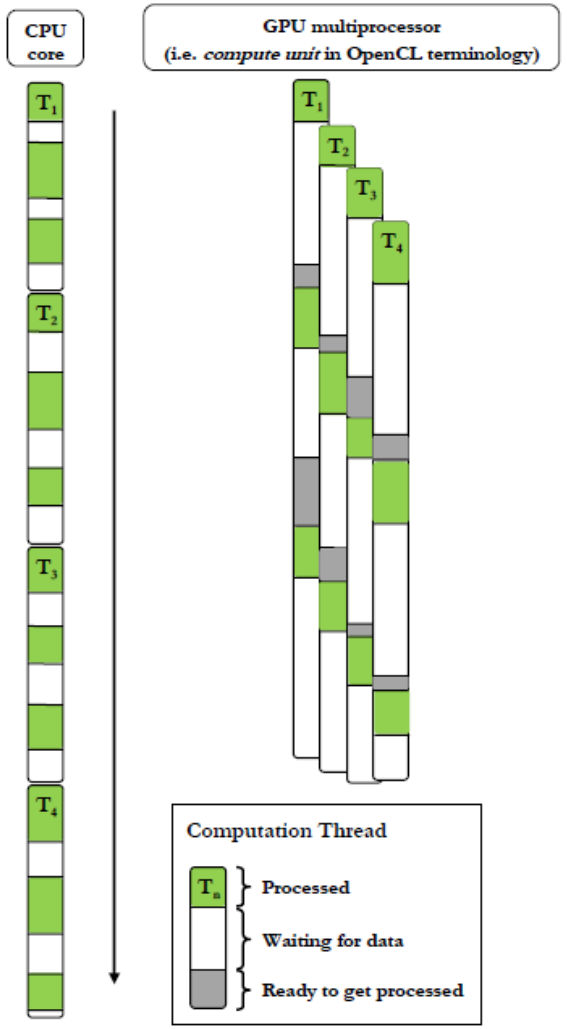
© 2010, Martin Kruliš

Skrývání latence pamětí

- CPU
  - Context-switch je drahá operace
  - Snaha mít co největší cache
- GPU
  - Context-switch je levný (při čekání na data může běžet jiné vlákno).
  - Malé cache

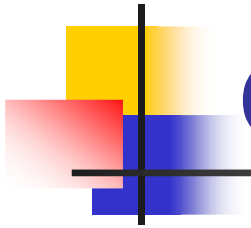


CPU GPU



Rozdíl ve skrývání latencí mezi CPU a GPU (oficiální materiály NVIDIA)

© 2010, Martin Kruliš



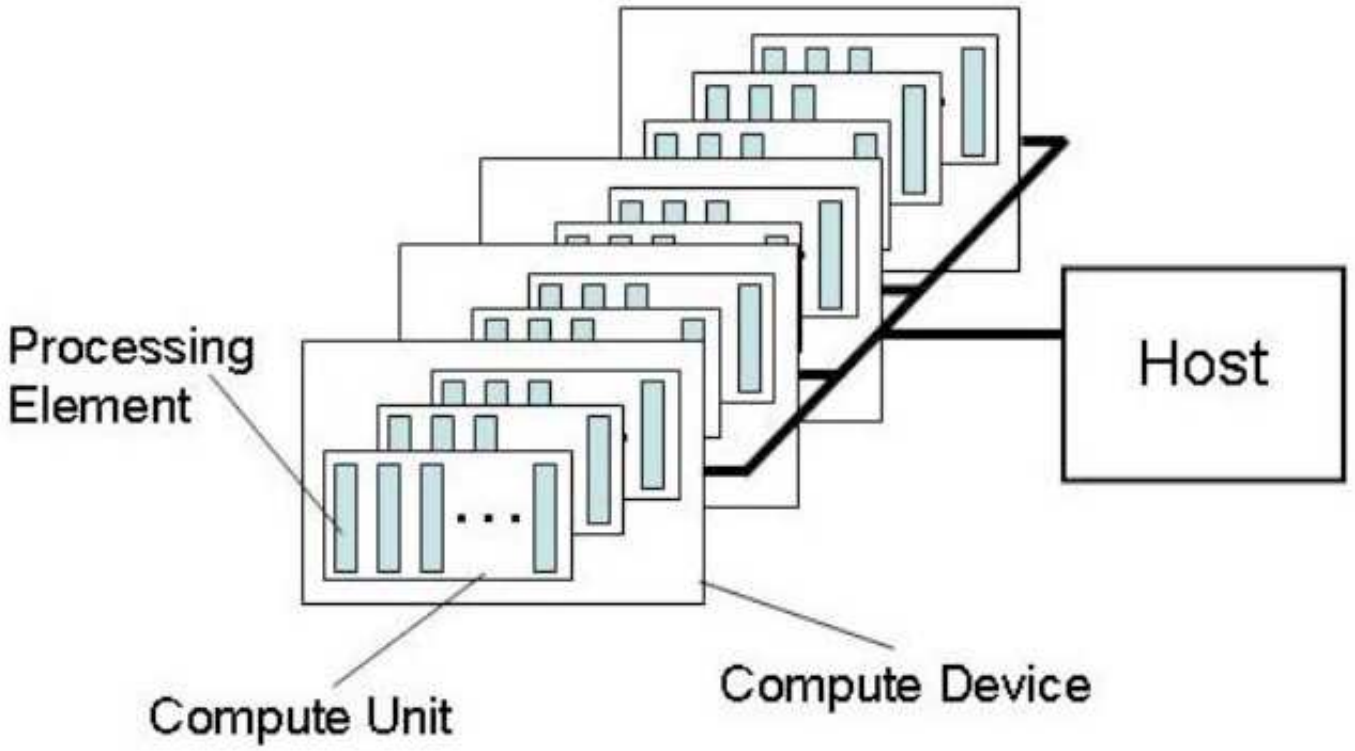
OpenCL

- Univerzální framework pro parallelní výpočty
  - Specifikace 1.0 vydána v srpnu 2009 sdružením Khronos.
  - Existují různé implementace (NVIDIA, AMD, Mac OS, …).
- API nad různými paralelními architekturami
  - Multi-core CPU, GPU, přídavné karty pro výpočty, …
  - Zastřešuje detekci, komunikaci, přesun dat a spouštění „kernels“.
  - Nabízí dva druhy paralelismu – data parallel a task parallel
- Vlastní rozšíření jazyka C99 pro psaní kernelů
  - Kernel je kompilován za běhu přímo pro cílovou platformu.
  - Teoreticky je možné až za běhu vybrat nejvhodnější zařízení.
    - Prakticky to není vždy vhodné, protože kód je potřeba optimalizovat.

© 2010, Martin Kruliš

OpenCL – architektura

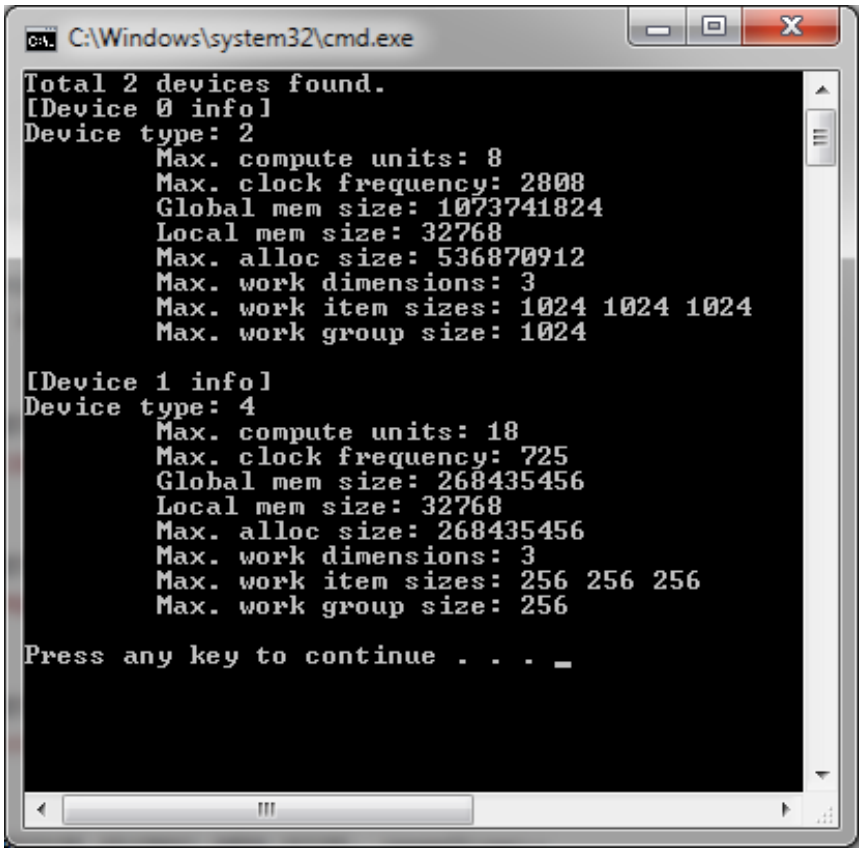
- OpenCL bere v úvahu více různorodých zařízení na jednom „host-u" (počítači).



© 2010, Martin Kruliš

OpenCL – logické vrstvy

- Dělení vrstev přístupu
  - Host může mít více platforem
    - Platforma ~ implementace OCL
  - Z platformy se vytvoří kontext
    - Kontext sdružuje zařízení vybraného typu
    - Zařízení je možné osahat a vybrat si dle parametrů
    - V rámci kontextu se také vytváří buffer, kompilují kernely, …
  - Zařízení
    - Vytváří se na něm fronty
    - Do fronty se vkládají příkazy (spouštění kernels, kopírování bufferů, …)



4 jádrový core i7 s HT
Radeon 5870 (1600 stream procesorů)

© 2010, Martin Kruliš

OpenCL – klientská aplikace

```
std::vector<cl::Platform> platforms;
cl_int err = cl::Platform::get(&platforms);
if (err != CL_SUCCESS) return 1;

cl_context_properties cps[3] = {CL_CONTEXT_PLATFORM,
    (cl_context_properties)(platforms[0]()), 0};
cl::Context context(CL_DEVICE_TYPE_GPU, cps, NULL, NULL, &err);

std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

cl::Buffer buf(context, CL_MEM_READ_ONLY, sizeof(cl_float)*n),

cl::Program program(context, cl::Program::Sources(1,
    std::make_pair(source.c_str(), source.length())) );
err = program.build(devices);

cl::Kernel kernel(program, "function_name", &err);
err = kernel.setArg(0, buf);

cl::CommandQueue commandQueue(context, devices[0], 0, &err);
commandQueue.enqueueWriteBuffer(buf, CL_TRUE, 0, sizeof(cl_float)*n, data);
commandQueue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(n), cl::NDRange(grp),
    NULL, NULL);
commandQueue.finish();
```

Seznam všech platforem

Kontext všech GPU zařízení na 1. platformě

Seznam všech GPU

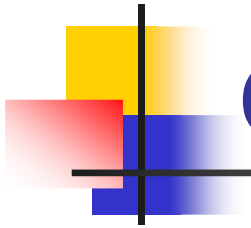
Vytvoříme a zkompilujeme program pro GPU

Z konkrétní funkce uděláme kernel object

Fronta příkazů na GPU

Poslání příkazů a čekání na jejich dokončení

© 2010, Martin Kruliš



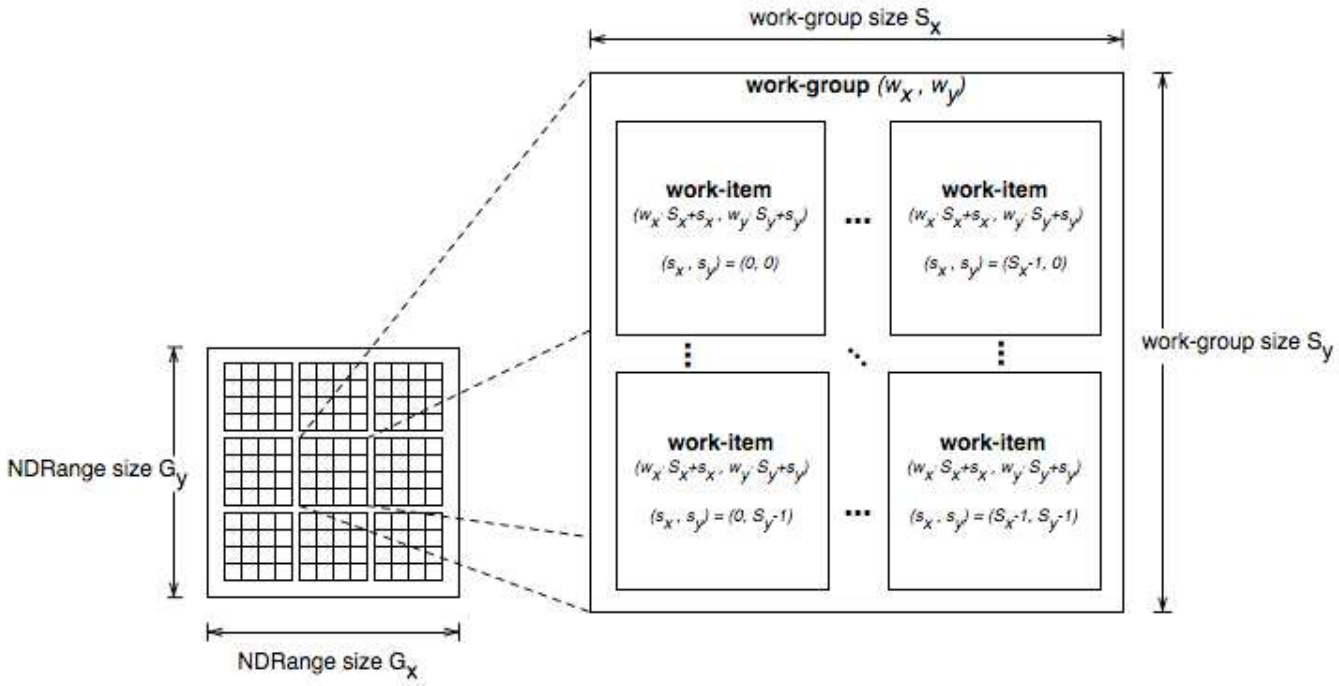
OpenCL – kernels

- Kernel
  - Program (funkce) v rozšířené syntaxi jazyka C
  - Kompiluje se za běhu přímo pro cílové zařízení
    - Vysoká míra optimalizace
- Spouštění kernels
  - Program si řekne, kolikrát chce kernel spustit (data parallelism)
    - Kernely se navíc spojují do skupin (skupina může sdílet lokální data)
    - Logicky se instance uspořádají do 1-3 rozměrné mřížky
      - Kernel má k dispozici funkce pro zjištění svého ID, ID skupiny, …
        - `get_global_id(dim)`, `get_local_id(dim)`, `get_global_size(dim)` …
        - Dle ID si může dopočítat, jakou část dat má zpracovat.
  - Více kernels může být ve frontě ke zpracování (task parallelism)
    - Ne každé zařízení task paralelism podporuje

© 2010, Martin Kruliš

OpenCL – spouštění kernels

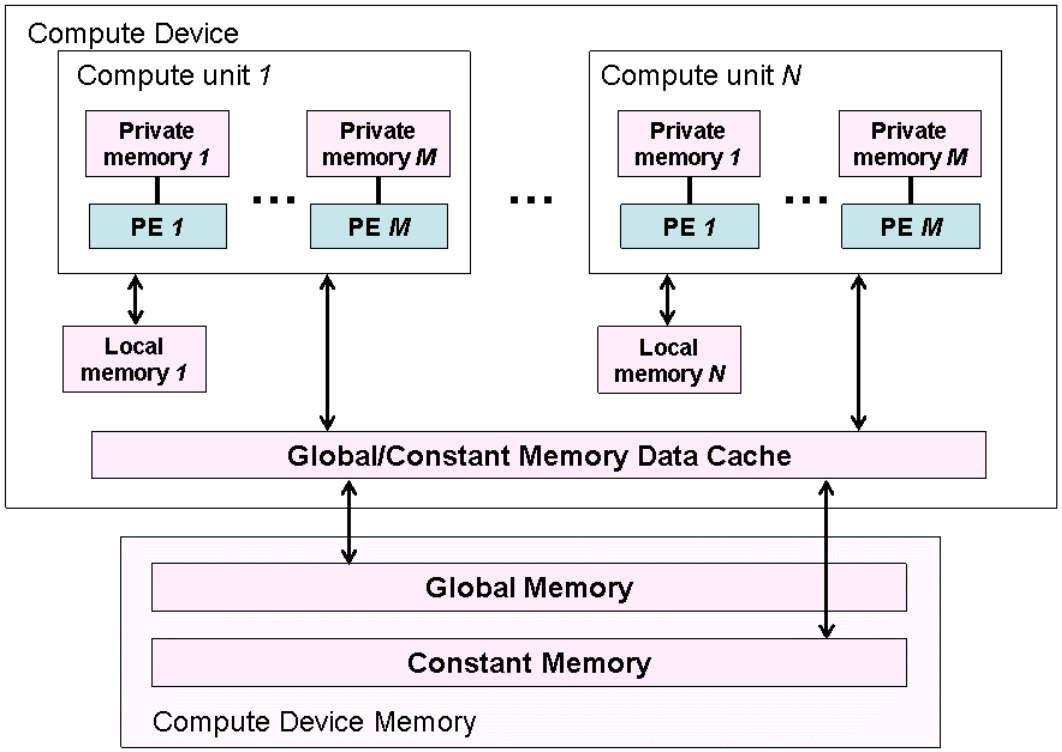
- Příklad – dvojrozměrné uspořádání
  - Velikost skupiny musí být v každém směru soudělná s velikostí problému
  - Je dobré, aby velikost problému byla mocnina 2



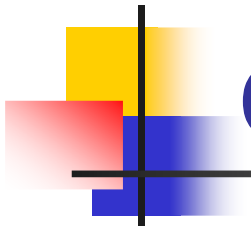
© 2010, Martin Kruliš

OpenCL – struktura paměti

- Struktura paměti
  - **`private`** – paměť vlastní jednomu kernelu
  - **`local`** – paměť sdílená skupinou kernelů
  - **`global`** – paměť sdílená všemi kernely spuštěnými paralelně
  - **`constant`** – stejné jako global, ale read-only
  - Nerozlišujeme globální paměť zařízení a hosta (OpenCL určí samo).



© 2010, Martin Kruliš



OpenCL – kód kernelu

- Datové typy
  - Téměř všechny běžné typy dostupné v jazyce C
  - Některé předdefinované typy: **`size_t`**, **`ptrdiff_t`**
  - Speciální typ **`half`** – 16bit. varianta **`float`**

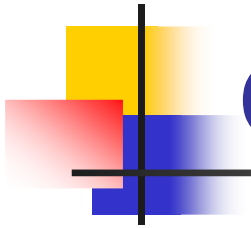
- Vektorové datové typy
  - Pro číselné typy existují vektorové varianty
  - **`char`***n*, **`int`***n*, **`float`***n*, **`long`***n*, … kde *n* je 2, 4, 8 nebo 16
  - K jednotlivým složkám se přistupuje, jako k prvkům struktury

```
int4 vec = (int4) (1, 2, 3, 4);
int x = vec.x;  \\ položky jsou xyzw nebo 0-f
```

  - Na položky je možné uplatnit swizzling (přehazování a duplikace):

```
float4 dup = vec.xxyy;
float4 rot = vec.yzwx;
```

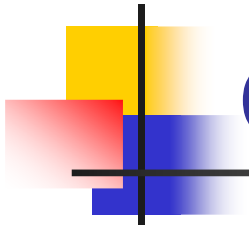

© 2010, Martin Kruliš



OpenCL – kód kernelu

- Funkce
  - V rámci jednoho programu může být definováno více funkcí, které se mohou vzájemně volat (kernel je jen vstupní bod).
  - Je možné volat i jiné funkce (např. **`printf`**), ale jejich skutečné zavolání závisí na kontextu, kde se kernel pouští
    - Na CPU se provedou, na GPU nikoli
  - Existuje řada vestavěných funkcí
    - Funkce vlákna (např. **`get_global_id()`**)
    - Matematické funkce
      - Velmi široká paleta – zejména pro grafické operace
      - Na GPU existuje pro spoustu z nich jediná instrukce
    - Geometrické funkce (skalární součin, euklidovská vzdálenost, …)
    - Funkce pro asynchronní přenosy dat z/do globální paměti

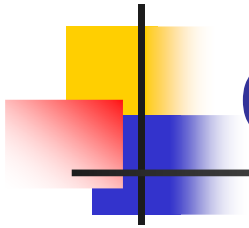
© 2010, Martin Kruliš



OpenCL – kód kernelu

- Výkon a optimalizace
  - Podmínky a větvení kódu
    - Skupina vláken běží v SIMT režimu – vykonávají se všechny větve
    - Pokud to jde, je lepší použít podmíněné přiřazení
      - Místo `if (up) y += dy; else y -= dy;`
      - Raději `int f = (up) ? 1 : -1; y += f*dy;`
  - For-cykly
    - Překladač se je automaticky pokouší rozvinout
  - While-cykly
    - Stejný problém jako u if-u, raději se jim vyhnout
  - Používejte vektorové instrukce
    - Na ATI může pomoct (každé jádro je 5D stream procesor)
    - Na CPU se typicky přeloží do SSE
    - Překladač se pokouší vektorové instrukce generovat sám

© 2010, Martin Kruliš



OpenCL – synchronizace

- Na úrovni klientské aplikace
  - Operace ve frontě se provádí out-of-order
  - Na dokončení operace lze čekat
- Na úrovni kernelu
  - Bariéry
    - Lokální – všechna vlákna v dané skupině
    - Globální – všechna vlákna vykonávající tentýž kernel
  - Memory fence
  - Atomické operace (volitelné rozšíření)
    - Lokální a globální typ (32 nebo 64 bit integer)
    - Dva typy – base a extended
      - Base – běžné operace jako add, sub, xchg a cmpxhg
      - Extended – atomický min, max, and, or, xor

© 2010, Martin Kruliš

Příklad ze života

- Násobení matic

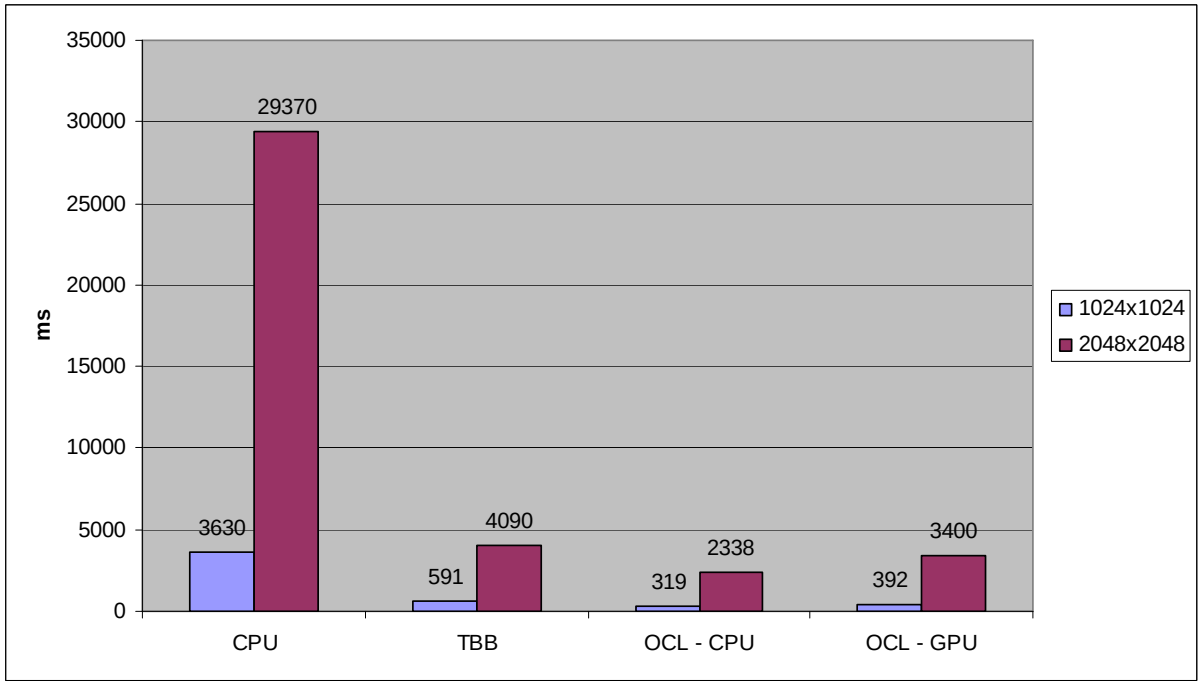
```
__kernel void mul_matrix (__global const float *m1,
     __global const float *m2, __global float *mRes)
{
     int n = get_global_size(0);
     int r = get_global_id(0);
     int c = get_global_id(1);
     float sum = 0;
     for (int i = 0; i < n; ++i)
            sum += m1[r*n + i] * m2[c*n + i];
     mRes[r*n + c] = sum;
}
```

Druhá matice je již transponovaná

© 2010, Martin Kruliš

Násobení matic

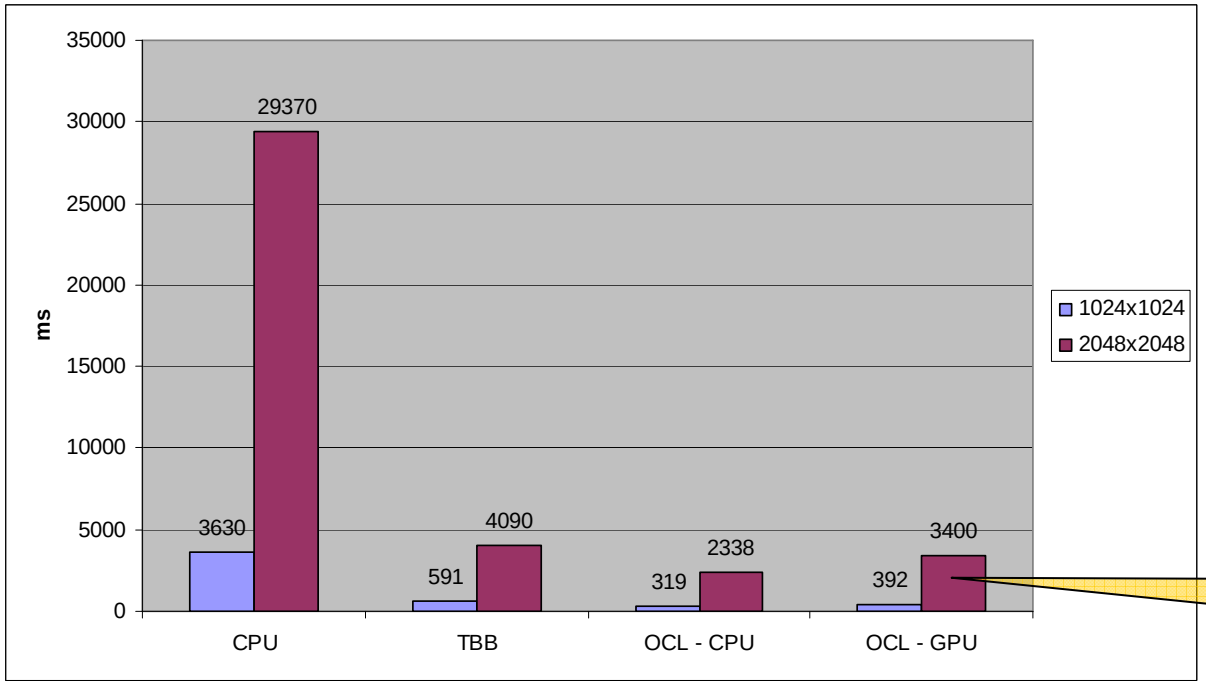
- Násobení dvou čtvercových matic floatů
  - Na AMD Radeon 8570 (320 jader) a Core i7 (4 jádra s HT)
  - Standardní algoritmus $O(N^3)$, druhá matice je transponovaná (optimalizace pro procesorové cache)



© 2010, Martin Kruliš

Násobení matic

- Násobení dvou čtvercových matic floatů
  - Na AMD Radeon 8570 (320 jader) a Core i7 (4 jádra s HT)
  - Standardní algoritmus $O(N^3)$, druhá matice je transponovaná (optimalizace pro procesorové cache)



Hmm...

© 2010, Martin Kruliš

Kde je problém?

- Podívejme se na výsledky profileru…
  - Matice 1024x1024

Kolikrát každé vlákno četlo z globální paměti

Method	ExecutionOrder	GlobalWorkSize	GroupWorkSize	KernelTime	LocalMem	MemTransferSize	ALU	Fetch	Write
BufHostToDevice	1					4194304			
BufHostToDevice	2					4194304			
mul_matrix_Cypress	3	{1024; 1024; 1}	{16; 16; 1}	372,05272	0		5133	2048	1
BufDeviceToHost	4					4194304			

…

Kolik % celkového času se četla paměť

…

Wavefront	ALUBusy	ALUFetchRatio	ALUPacking	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	ALUStalledByLDS	LDSBankConflict
16384	6,34	2,51	35,99	94,37	83,15	0	0	0

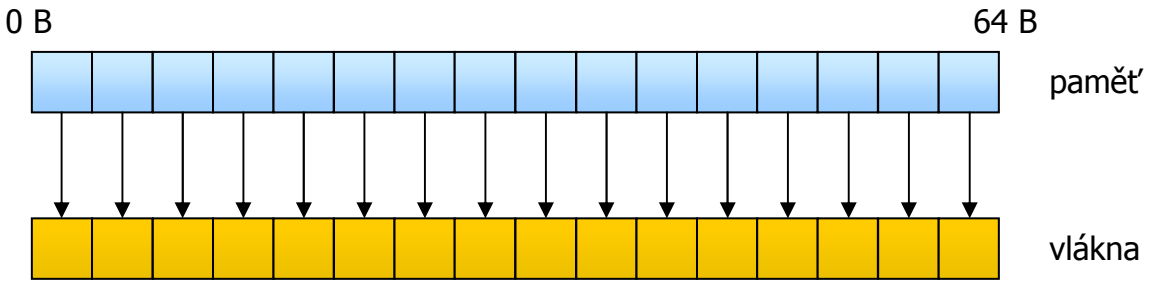
Procentuální poměr ALU op. vs. čtení

Kolik % celkového času čekaly fetch jednotky na data

© 2010, Martin Kruliš

Přístup do globální paměti

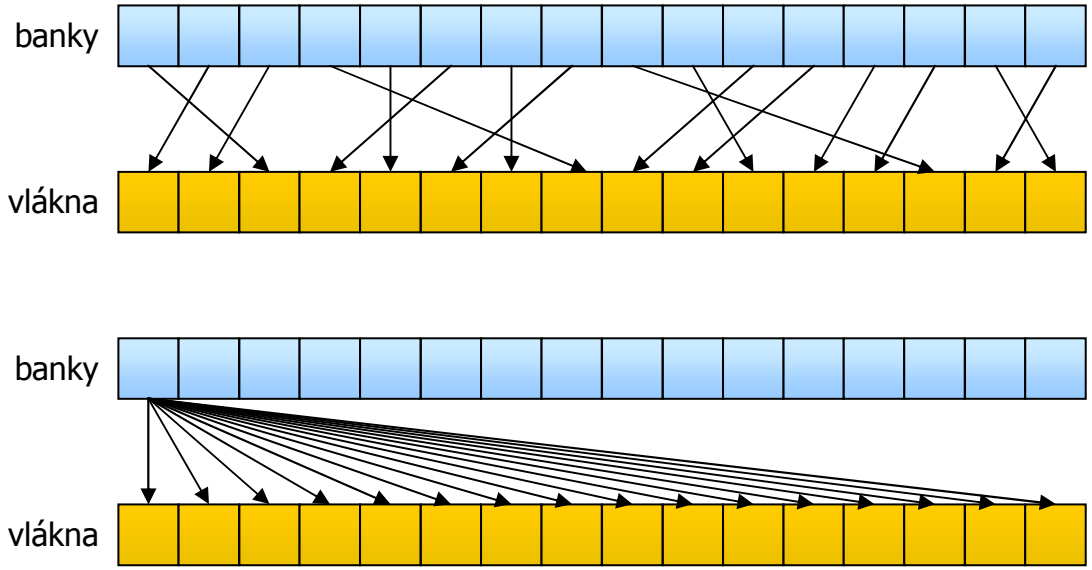
- Coalesced Memory Load
  - Na GPU běží vždy několik vláken v SIMT módu společně
    - NVIDIA – warp 32 vláken (resp. half-warp 16 vláken)
    - AMD/ATI – wavefront 64 vláken
  - Pokud tato vlákna načítají paměťové bloky zarovnaně, provede se coalesced load a všechna paměť se načte najednou.
    - Každé vlákno musí načítat buňku velikosti 4B (jeden int nebo float).
      - NVIDIA – 64B bloky
      - ATI – 128B bloky



© 2010, Martin Kruliš

Přístup do lokální paměti

- Lokální paměť
  - Sdílená mezi vlákny ve skupině (warpu).
  - Velmi malá (např. na GTX280 16KB) a stejně rychlá jako registry
  - Rozdělená do bank (buňky velikosti 4B modulo #bank).
    - NVIDIA – 16 bank, ATI – 32 bank



Do 1 banky přistupuje 1 vlákno
(na přeskáčku)

Broadcast
(pouze NVIDIA)

© 2010, Martin Kruliš

Násobení matic – úprava pro GPU

```
__kernel void mul_matrix_opt (__global const float *m1, __global const float *m2, __global float *mRes,
    __local float *tmp1, __local float *tmp2)
{
    int size = get_global_size(0);
    int lsize_x = get_local_size(0);
    int lsize_y = get_local_size(1);
    int block_size = lsize_x * lsize_y;
    int gid_x = get_global_id(0);
    int gid_y = get_global_id(1);
    int lid_x = get_local_id(0);
    int lid_y = get_local_id(1);
    int offset = lid_y*lsize_x + lid_x;

    float sum = 0;
    for (int i = 0; i < size; i += lsize_x) {
        // Load data to local memory
        tmp1[offset] = m1[gid_y*size + i + lid_x];
        for (int j = 0; j < lsize_x / lsize_y; ++j)
                tmp2[offset + j*block_size] = m2[(gid_x + lsize_y*j)*size + i + lid_x];
        barrier(CLK_LOCAL_MEM_FENCE);

        // Add data from block to the sum
        for (int k = 0; k < lsize_x; ++k)
                sum += tmp1[lid_y*lsize_x + k] * tmp2[lid_x*lsize_x + k];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    mRes[gid_y*size + gid_x] = sum;
}
```

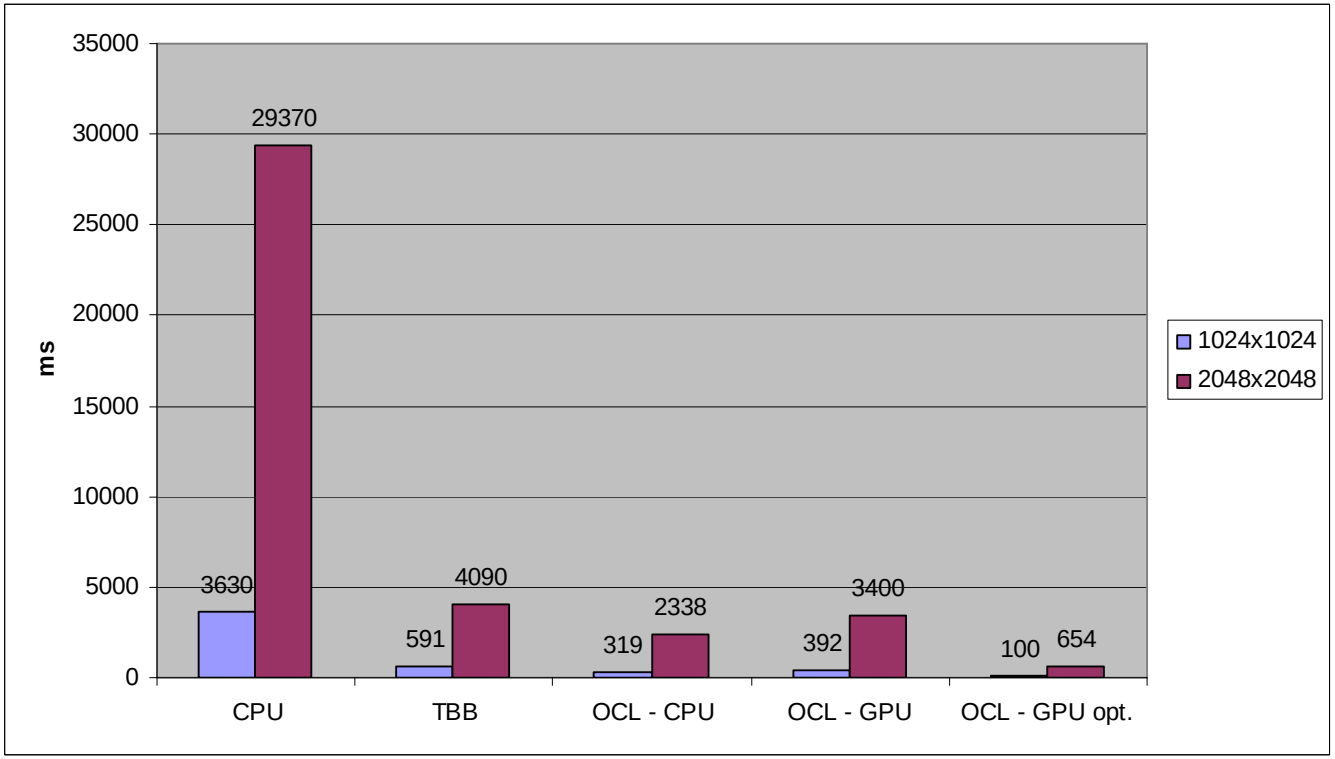
Zkopírujeme část matice do lokální paměti

Spočítáme mezivýsledky z načtených částí

© 2010, Martin Kruliš

Násobení matic – úprava pro GPU

- Verze optimalizovaná pro GPU
  - 45x rychlejší než sériová verze
  - 3.6x rychlejší než paralelní verze



© 2010, Martin Kruliš

Násobení matic – výsledky profileru

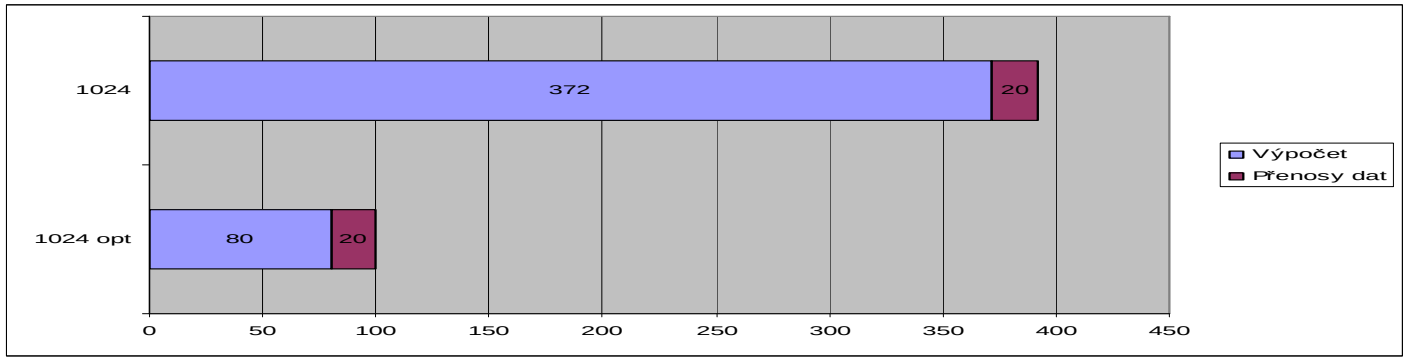
- Výsledky profileru po optimalizaci
  - Fetch se zmenšilo ze 2048 na 128
  - Poměr ALU operací ku čtení vzrostl na 45%
  - Čtení zabralo pouze 9% celkového času a pouze 5% se na data čekalo
- Ale…
  - Docházelo ke 100% množství kolizí na bankách lokální paměti

ALU	Fetch	ALUBusy	ALUFetchRatio	ALUPacking	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	ALUStalledByLDS	LDSBankConflict
7328	128	45,87	57,25	32,16	9,28	5,71	0	100	100

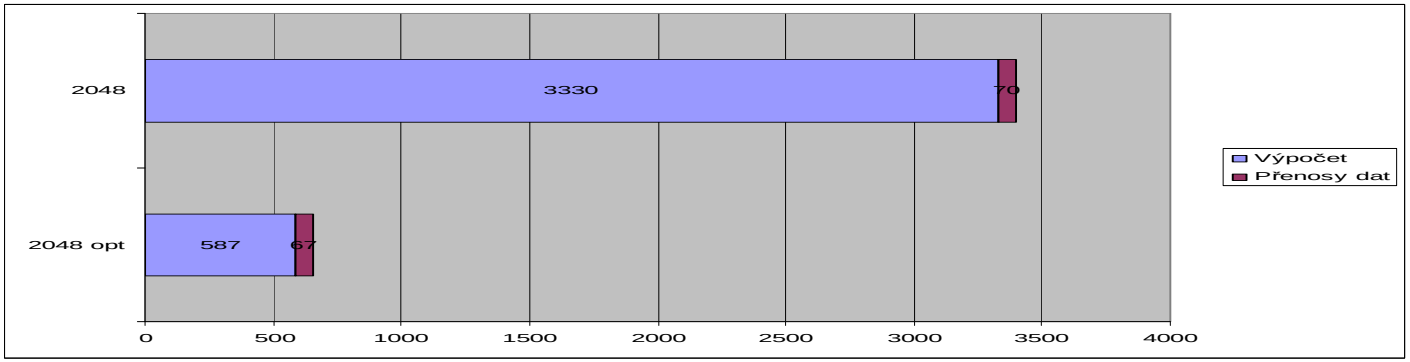
© 2010, Martin Kruliš

Násobení matic – přenosy dat

- Doba potřebná na přenos dat na GPU a zpět
  - Matice 1024x1024 floatů (2x 4 MB tam, 4 MB zpět)



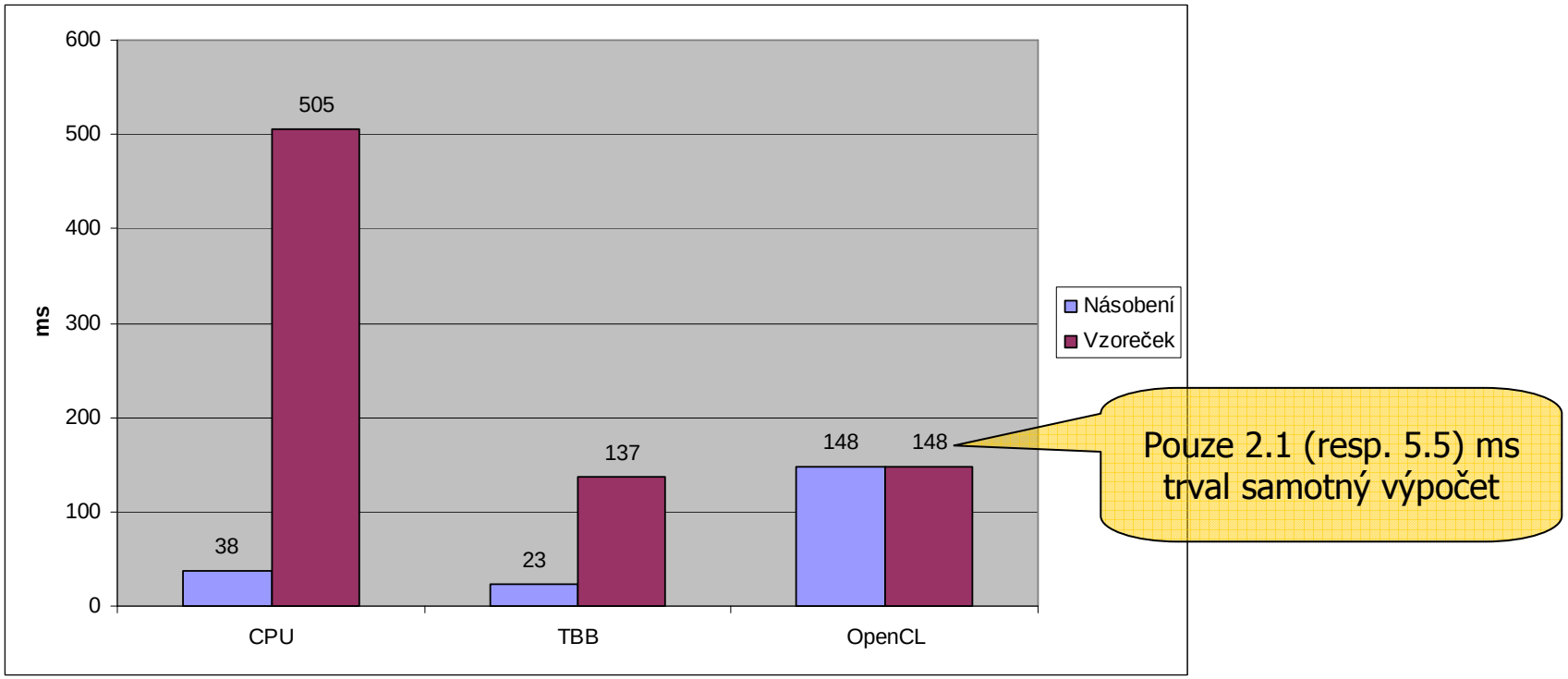
  - Matice 2048x2048 floatů (2x 16 MB tam, 16 MB zpět)



© 2010, Martin Kruliš

Jednoduché operace

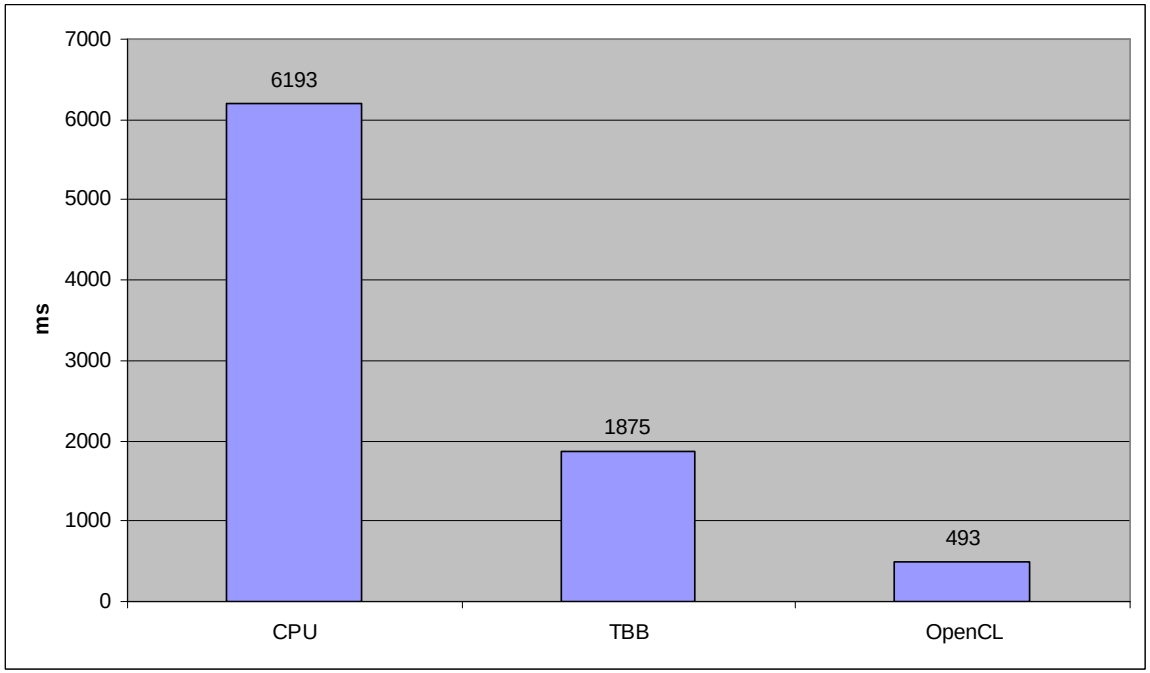
- Jednoduché operace nad dvěma vektory 16 M float čísel
  - Násobení: `z[i] = x[i] * y[i];`
  - Složitější vzoreček:
    `z[i] = (sqrt(x[i])*y[i]/x[i]) + cos(y[i]) * x[i];`



© 2010, Martin Kruliš

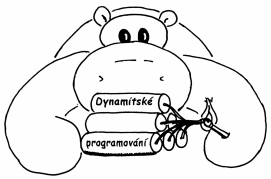
Backtracking

- Variace na součet podmnožiny
  - Dána množina (30ti) čísel; každé je v součtu jako kladné nebo záporné a hledáme předepsaný součet.
  - Maximální vytížení – kombinace neexistuje (máme sudá čísla, chceme liché).



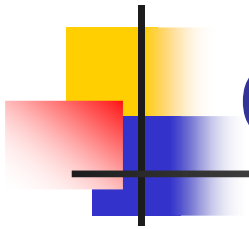
© 2010, Martin Kruliš

Potíže při nasazení



- Ovladače
  - Ovladače jsou poměrně mladé.
  - Občas může chyba programu běžícího na GPU způsobit pád OS.
- Data
  - Data je potřeba přesunou z operační přes PCIe do zařízení a zpět.
  - Málo DRAM, část spotřebuje OS na zobrazování
- Task Parallel Execution
  - Momentálně umí pouze NVIDIA Fermi
- Kompilace kernel-u
  - Kompilace chvíli trvá (i jednotky sekund) – vyplatí se?

© 2010, Martin Kruliš

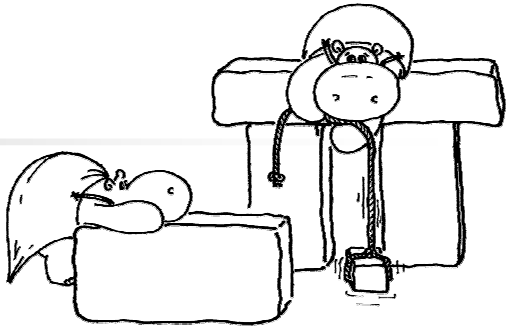


OpenCL vs. OpenGL

- Provázání OpenCL a OpenGL
  - OpenCL je mladší bratr OpenGL
  - Silná podpora pro práci s (nejen 3D) grafikou
- Datový typ pro 2D a 3D obrázky
  - Speciální typ umožňuje definovat, jak jsou reprezentovány barvy
  - Řada konverzí
    - CL buffer → GL buffer
    - CL image object → GL texture
    - CL buffer → GL renderbuffer
- Sdílení CL a GL kontextu
  - Vytváření CL objektů z GL objektů

© 2010, Martin Kruliš

Alternativní technologie



- NVIDIA CUDA
  - První GPGPU technologie (již v r. 2007)
  - Cílem bylo přesunout zejména výpočty fyziky na GPU
    - NVIDIA koupila v r. 2008 firmu Agelia a její technologii PhysX
  - Jednodušší API
    - Víme, že pracujeme s GPU – odpadá detekce platforem, zařízení, …
    - Kernels jsou přímo v kódu a volají se téměř jako normální funkce
      - `KernelFnc<<<1, N>>>(A, B, C);`
- Direct Compute
  - Navržené Microsoftem, součást DirectX 11 (listopad 2009)
  - Bližší integrace s vývojem her
  - Použití velmi podobné jako vertex nebo fragment shaderu
    - Pouští se speciální verze shaderů, kterým se dají speciální buffery

© 2010, Martin Kruliš

Dotazy

© 2010, Martin Kruliš