

OpenCL

Jiří Filipovič

podzim 2012

OpenCL

Co je OpenCL?

- otevřený standard pro programování heterogenních systémů
- nízkourovňový, odvozený od C, velmi podobná HW abstrakce jako CUDA

Výhody oproti CUDA

- možno programovat pro širokou škálu hardware
- otevřený standard, nezávislý na osudu jedné firmy

Nevýhody oproti CUDA

- složitější API (podobné CUDA Driver API)
- mladší, obecně méně „dospělá“ implementace
- pomalejší reflektování novinek v hardware

Přenostelnost

Jedna implementace lze kompilovat pro různý HW

- nepoužíváme-li extenze

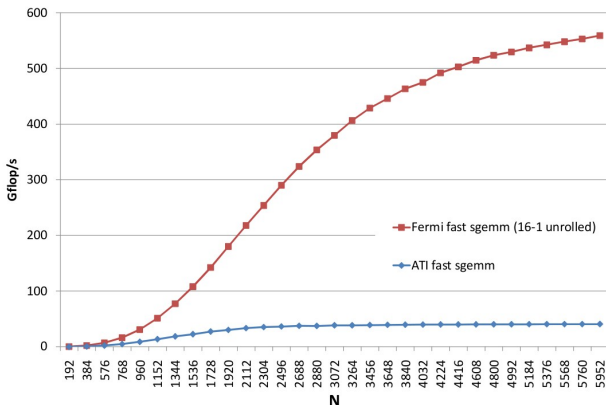
Implementace vyladěná pro jeden HW však může mít velmi nízký výkon na jiném

- přílišná rozdílnost HW vyžaduje odlišnou optimalizaci

OpenCL nám tedy umožňuje používat jeden standard pro programování široké škály HW, nicméně optimalizace se musíme naučit pro každý zvlášť.

- usnadňuje autotuning

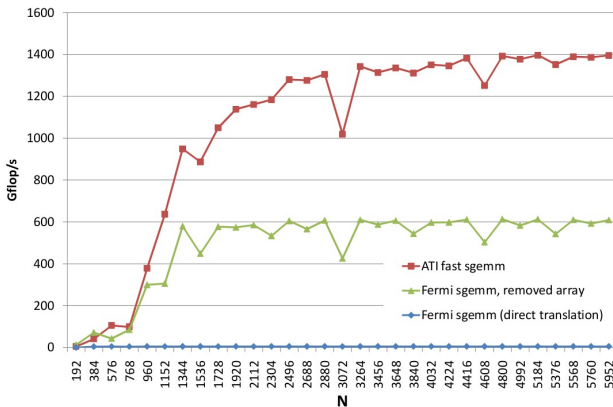
Přenostelnost výkonu



Obrázek: SGEMM optimalizované pro Fermi a Cypress, běh na Fermi¹.

¹Du et al. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming

Přenostelnost výkonu



Obrázek: SGEMM optimalizované pro Fermi a Cypress, běh na Cypress².

²Du et al. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming

Hlavní rozdíly

OpenCL není integrováno s C/C++

- může být kompilováno za běhu, v kódu uložen OCL kernel jako řetězec
- nemůže sdílet části kódu s C/C++ (uživatelské typy, kód pro OCL i C atp.)

Kernely v OpenCL nepracují s klasickými ukazateli

- nelze dereferencovat, používat pointerovou aritmetiku, nelze přecházet mezi buffery
- v rámci bufferu se můžeme pohybovat pomocí offsetu

OpenCL je odvozeno striktně od C

- žádné vlastnosti z C++

OpenCL ovládá různá zařízení pomocí front

- není třeba přepínat kontext

Fronty v OpenCL mohou být out-of-order

- umožňuje runtime balancování využití zdrojů

CUDA-OpenCL slovníček

Hlavní rozdíly v základní terminologii

CUDA	OpenCL
multiprocessor	compute unit
scalar procesor	processing element
thread	work-item
thread block	work-group
grid	NDRange
shared memory	local memory

Součet vektorů – kernel

CUDA

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

OpenCL

```
__kernel void vecadd(__global float * a, __global float * b,
__global float * c)
{
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```


Součet vektorů – host kód

Ke spuštění kernelu potřebujeme

- definovat platformu
 - zařízení (může být více)
 - kontext
 - fronty
- alokovat a nakopírovat příslušná data
- skompilovat kód kernelu
- nakonfigurovat běh kernelu a spustit jej

Součet vektorů – definice platformy

```
cl_uint num_devices_returned;
cl_device_id cdDevice;
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1,
&cdDevice, &num_devices_returned);

cl_context hContext;
hContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &err);

cl_command_queue hQueue;
hQueue = clCreateCommandQueue(hContext, hDevice, 0, &err);
```

Součet vektorů – spuštění kernelu

Každá platforma může mít více zařízení

- lze vybrat nahrubo podle typu
- můžeme také upřednostnit výrobce
- jemnější dělení dle informací o HW
 - počet jader
 - frekvence
 - velikost paměti
 - rozšíření (dvojitá přesnost, atomické operace aj.)

Každé zařízení musí mít alespoň jednu frontu

- jinak jej neumíme využít

Součet vektorů – alokace a kopírování

```
cl_mem hdA, hdB, hdC;  
hdA = clCreateBuffer(hContext, CL_MEM_READ_ONLY,  
    cnDimension * sizeof(cl_float), pA, 0);  
...
```

Kopírovat není explicitně třeba – alokace a kopie dat funguje líně – až když jsou data potřeba. Není tedy ani nutno specifikovat zařízení, pro které data alokujeme.

Součet vektorů – spuštění kernelu

```
const unsigned int cnBlockSize = 512;
const unsigned int cnBlocks = 3;
const unsigned int cnDimension = cnBlocks * cnBlockSize;

cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1, sProgramSource
    ,0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0);

cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, "addvec", 0);

clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&hdA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&hdB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&hdC);

clEnqueueNDRangeKernel(hQueue, hKernel, 1, 0, &cnDimension
    , &cnBlockSize, 0, 0, 0);
```

Součet vektorů – úklid

```
clReleaseKernel(hKernel);  
clReleaseProgram(hProgram);  
clReleaseMemObj(hdA);  
clReleaseMemObj(hdB);  
clReleaseMemObj(hdC);  
clReleaseCommandQueue(hQueue);  
clReleaseContext(hContext);
```

Architektura AMD VLIW GPU

Starší procesory

- Evergreen a Northern Islands

Budeme se věnovat jen zásadním rozdílům oproti nVidia GPU

- ostatní aspekty shodné nebo velmi podobné

Hlavní rozdíly

- VLIW architektura
- explicitně odděleny dvě cesty do paměti (rychlá a obecná)
- méně citlivé na nezarovnaný přístup, více citlivé na analogii partition campingu
- wavefront (analogie warpu) velikosti 64 vláken

VLIW architektura

AMD GPU jsou VLIW procesory

- instrukční slovo obsahuje několik paralelně proveditelných operací
- statické plánování instrukcí (závislosti analyzovány v době kompilace)
- umožňuje vyšší hustotu ALU
- vlákna musí nabízet dostatek „instrukčního paralelismu“ a kompilátor jej musí rozpoznat
- AMD GPU implementují VLIW-5 nebo VLIW-4, 1 instrukce SFU

Optimalizace pro VLIW

Explicitní užití vektorů

- pracujeme s vektorovými datovými typy (např. float4)
- generování VLIW instrukcí je pro kompilátor přímočaré

Implicitní převod na VLIW

- můžeme psát také skalární kód
- kompilátor se snaží seskupit nezávislé instrukce
- můžeme mu v tom pomoci (unrolling a seskupení stejných operací)

Optimalizace pro VLIW

Problémy spojené s VLIW

- větší spotřeba on-chip paměti (unrolling, vektorové typy)
- nemusíme mít dost nezávislých instrukcí
 - např. podmíněné výrazy
- spolu s velikostí wavefrontu dost citlivé na divergenci

Přístup do globální paměti

Fast path vs. complete path

- fast path je výrazně rychlejší
- fast path používaná pro load/store 32-bitových čísel
- complete path používaná pro čísla jiné velikosti a atomické operace
- při operacích s globální pamětí musí explicitně rozlišit použitou cestu kompilátor
 - cesta je stejná pro celý buffer, můžeme snadno znemožnit kompilátoru použít rychlou variantu

Fast path vs. complete path

```
__kernel void  
CopyComplete(__global const float * input, __global float* output)  
{  
    int gid = get_global_id(0);  
    if (gid < 0){  
        atom_add((__global int *) output, 1);  
    }  
    output[gid] = input[gid];  
    return ;  
}
```

Rozdíl rychlosti na Radeon HD 5870: 96 GB/s vs. 18 GB/s.

Nezarovnaný přístup

Permutace mapování vlákno-element v rámci wavefrontu

- penalizace, ale malá ($< 10\%$)
- lepší než c.c. < 1.2 , mírně horší než c.c. ≥ 1.2

Posun v rámci wavefrontu

- malá penalizace ($< 10\%$)
- lepší než c.c. 1.x, u 2.x záleží na využití cache

Výhodnější přístup po 128-bitových elementech

- např. používáme-li float4
- kopírování 122 namísto 96 GB/s u HD 5870

Konflikty bank a kanálů

Analogie partition campingu

- do paměti přistupujeme přes banky a kanály
- v rámci wavefrontu je paralelní přístup přes stejný kanál (jinak serializujeme)
- mezi work-groupami je třeba přistupovat přes různé kanály
- Radeony řady 5000 mají kanály prokládané po 256 bytech
 - wavefront čtoucí souvislý blok 32-bitových elementů čte paralelně z téhož kanálu
 - jednotlivé wavefronty pak čtou z rozdílných kanálů čtou-li souvislou oblast
- uspořádání bank závisí na počtu kanálů
 - např. 8 kanálů – banky se střídají každé 2 KB
- vysoká penalizace za čtení ze stejného kanálu a stejné banky (např. 0.3 vs 93 GB/s)

Lokální paměť

Lokální paměť velmi podobná sdílené paměti u nVidie

- organizována do 32 nebo 16 bank
- je třeba přistupovat ve čtvrtinách wavefrontu do rozdílných bank
 - jinak vzniká konflikt bank
 - v případě 32 bank používat efektivně i float2
- broadcast je podporován pro jednu hodnotu (analogie c.c. 1.x)

Architektura AMD GCN GPU

Aktuální architektura (Southern Islands, také Graphic Core Next).
Významná změna oproti předchozí generaci

- není VLIW, compute unit obsahuje 1 skalární procesor a 4 vektorové procesory
 - kód vláken píšeme skalárně, vektorové operace se obvykle nevyplatí
 - podmíněné výrazy přináší menší penalizaci
- L1 cache pro čtení i zápis
- na GPU může běžet více kernelů současně

CPU a OpenCL

Dnešní CPU z pohledu OpenCL

- compute units odpovídají jádrům
- processing elements elementům ve vektorových registrech
 - počet processing elements v compute unit tedy určuje použitý datový typ (2-16 pro SSE)
 - tomu odpovídá i počet vláken běžících v lock-step (analogie warpu/wavefrontu)
- jedna work-group se mapuje na jedno CPU vlákno
 - je vhodné mít minimálně tolik work-group, kolik máme jader
 - více work-group usnadňuje balancování zátěže, ale vytváří overhead
- jednotlivé work-item běží ve vektorových instrukcích, nebo sériově
- úlohový paralelismus – využíváme více front

Implicitní a explicitní vektorizace

Implicitní vektorizace

- píšeme kód podobně jako pro GPU
- kompilátor sám vytváří vektorové instrukce ze skupin work-item
- potenciálně lépe přenositelné (nezajímá nás délka vektorů u konkrétního CPU, ani bohatost instrukční sady)
- podporuje OpenCL od Intelu, AMD zatím neumí

Explicitní vektorizace

- v kódu explicitně používáme vektorové datové typy
- méně pohodlné a více svázané s konkrétní architekturou
- potenciálně vyšší výkon (nejsme omezeni tím co rozliší kompilátor)

Paralelismus na úrovni work-group

Jak zvolit velikost work-group

- nepotřebujeme větší paralelismus pro maskování latence paměti
- počet work-items alespoň na úrovni šířky vektoru (pro implicitní vektorizaci)
- více work-items v principu nevádí, pokud nepoužíváme bariéry
 - Intel doporučuje 64-128 work-items bez synchronizací, pokud synchronizujeme tak 32-64
- v současné implementaci implicitní vektorizace Intelu musí být velikost work-group dělitelná 4 pro Nahalem a 8 pro Sandy Bridge
- velikost work-group lze nspecifikovat, určí ji pak kompilátor

Další rozdíly oproti CPU

Images

- CPU nepodporuje funkce texturových jednotek, ty jsou emulovány
- pokud nepotřebujeme, je lepší nepoužívat

Lokální paměť

- pro CPU neodpovídá žádné HW paměti
- používání přináší overhead (kopírování tam a zpět)
- má smysl používat podobné vzory přístupu do paměti, jako kdybychom lokální paměť měli (cache lokalita)

Další rozdíly oproti CPU

Přístup do globální paměti

- velké cache, je třeba správně využívat

Lokalita

- chceme co nejvíce využívat data uložená v cache
- více úrovní cache, lze využít hierarchické shlukování přístupu

Organizace cache

- velikost řádku cache (kolik dat čteme současně)
- velikost celé cache (kolik dat udržíme)
- asociativita (jak efektivně jsme schopni mapovat paměť do cache)