Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta

Bakalářská práce

Programování výpočtů na grafických kartách

Autor: Tomáš Krejsa

Vedoucí práce: RNDr. Milan Předota, Ph.D.

Studijní obor: Aplikovaná informatika

Ústav aplikované informatiky

České Budějovice 2013

BIBLIOGRAFICKÉ ÚDAJE:
Krejsa, T. 2013: Programování výpočtů na grafických kartách.
[Programing of calculations on graphic cards, Bc. Thesis, in Czech.] - Faculty of Science, University of South Bohemia in České Budějovice, Czech Republic.

ANOTACE:
Paralelní programování na grafických kartách je zatím rozšířené většinou ve vědecké sféře pro náročné a dlouhotrvající výpočty. Cílem této bakalářské práce je rozkrýt tuto černou skříňku, kterou paralelní programování na GPU je a naučit čtenáře základy tohoto programování v OpenCL. Pro tento účel je popsán paralelní program výpočet energie vody. Na závěr proběhly testy, kde se porovnávají rychlosti GPU vs. CPU, float vs. double a také testy závislosti počtu vláken na čase.

KLÍČOVÁ SLOVA:
GPGPU, paralelní programování na GPU, výpočty na grafických kartách, OpenCL

BIBLIOGRAPHIC INFORMATION:
Krejsa T. 2013: Programování výpočtů na grafických kartách.
[Programing of calculations on graphic cards, Bc. Thesis, in Czech.] - Faculty of Science, University of South Bohemia in České Budějovice, Czech Republic.

ANOTATION:
Parallel programming on graphic cards is mostly known in the research community. It's using for time-consuming computations. Objective of this work is to uncover the black box, which parallel programming on GPU is and to teach bases of this programming in OpenCL. Parallel program calculation of water energy is described for this purpose. Finally, tests were carried out, which compares the speed of the GPU vs. CPU, float vs. double and also testing of threads depending on the time.

KEYWORDS:
GPGPU, parallel computing on GPU, OpenCL

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

25. 4. 2013 ..……..

Děkuji RNDr. Milan Předota, Ph.D. za cenné rady, připomínky a za čas, který mi věnoval při vedení bakalářské práce.

Obsah

1 Úvod 8
1.1 Historie CPU 8
1.2 Historie GPU 8
1.3 Motivace 9
1.4 Cíle práce 9
2 Detail GPU 10
2.1 Streaming multiprocesor 10
2.2 Softwarové uspořádání vs. hardwarové 12
3 Architektura 13
3.1 Parametry PC 13
3.1.1 Grafická karta 13
3.1.2 Procesor 14
3.2 Parametry výpočetního klastru Gram 14
3.2.1 Grafická karta 15
3.2.2 Procesor 16
3.3 Zařazení grafických karet 16
4 Paměťový model 18
4.1 Privátní paměť 18
4.2 Lokální paměť 19
4.3 Konstantní paměť 19
4.4 Globální paměť 20
5 Získávání informací o zařízení 20
6 Datové typy 20
6.1 Skalární datové typy 20
6.2 Vektorové datové typy 22
6.3 Dvojitá přesnost a poloviční plovoucí desetinná čárka 22
7 Programování 23
7.1 Popis programu 23
7.1.1 Host 24
7.1.2 Kernel 28
7.2 Optimalizace kernelu 30

7.2.1 Energie vody 30
7.3 Matematické funkce 32
8 Výsledky 32
8.1 Energie vody 32
8.2 Závislost času na počtu vláken 33
8.3 Matematické funkce 34
9 Závěr 35
Seznam obrázků 36
Seznam tabulek 37
Bibliografie 38
Příloha 40
A Zdrojové kódy 40
A.1 OpenCL – Výpočet energie vody 40
A.2 OpenMP – Výpočet energie vody 48
A.3 Sériový program – Výpočet energie vody 50
A.4 OpenCL – Matematické funkce 52
A.5 OpenMP – Matematické funkce 56
B Obsah CD 57

1 Úvod

1.1 Historie CPU

Zpočátku byly počítače specializované na určité výpočty a daly se přeprogramovat pouze změnou zapojení. Teprve koncept Johna Von Neumanna položil základ pro univerzální programovatelné stroje. Díky tomuto konceptu vznikl tedy procesor, který vykonává zadaný program, a tím se změní specializace počítače. Postupem času byly procesory stále dokonalejší. Jejich vývoj se hnal nezadržitelně kupředu. Procesory byly vyráběné dělané pro čím dál větší šířky slova, zvyšoval se počet elektronek a později tranzistorů. Až poměrně velkým mezníkem byl v roce 1985 procesor, který podporoval multitasking. Myšlenky slučování více čipů do jednoho přišly již v roce 1997, kdy se objevil procesor s integrovanou grafickou a zvukovou kartou. Později se procesory začaly slučovat do jednoho čipu (v roce 2008 procesor Intel Core i7 mohl mít až 8 jader). V roce 2002 společnost AMD začala používat integrovanou grafickou kartu jako matematický koprocesor.[1, 7, 9, 10]

Ve zdrojových kódech bakalářské práce se objeví také matematické funkce. Dnes je možné napsat ve vyšším programovacím jazyce příkaz pro výpočet každé běžnější matematické funkce a počítač ji spočítá. Ovšem nebylo tomu tak vždy. Procesor dokázal dříve sčítat, odčítat, násobit i dělit. Ale už nezvládl násobit či dělit v plovoucí řádové čárce, počítat goniometrické funkce, odmocňovat apod. Tento problém numerických výpočtů vyřešil teprve numerický koprocesor v roce 1978, který byl přidán k procesoru. Koprocesor a procesor byly tedy dva čipy a nevyplatilo se je slučovat do jednoho, protože plocha čipu by se zvětšila a v důsledku toho by se jeho výkon velmi snížil. Výrobní technologie procesorů se stále zdokonalovaly, až se pro výrobce stalo přijatelné sloučit procesor s koprocesorem do jednoho čipu. Integrovaný matematický koprocesor se objevil u mikroprocesoru i80486 v roce 1989. A od té doby, co se týče operací s plovoucí řádovou čárkou, procesor vypadá tak, jak ho známe dnes. Ale jak bylo zmíněno výše, díky některým operacím, které grafická karta zvládne lépe, se začíná prosazovat integrovaná grafická karta právě jako matematický koprocesor.[1, 7, 10]

1.2 Historie GPU

Doménou grafické karty bylo zpočátku zobrazování dat. V roce 2003 se objevily programovatelné Pixely a Vertex shadery. To znamenalo, že grafická karta přestala být striktně specializovaná a začala být snaha o využití grafické karty také pro výpočty. Podpora programovacího jazyka C a s tím související CUDA C, která jazyk C rozšiřuje, přišla s jádrem G80 od společnosti nVidia. V jádře G80 se již objevila univerzální výpočetní jednotka (streaming procesor, nebo CUDA core), která nahradila Vertex a Pixel shadery.[8]

Některé operace, např. operace s plovoucí řádovou čárkou, jsou na GPU mnohem rychlejší než na CPU. Do roku 2008 byly možné pouze výpočty s jednoduchou přesností plovoucí řádové čárky, ale jádro GT 200 to vše změnilo. Přineslo, kromě dalšího navýšení počtu streaming procesorů a dvojnásobné šířky registrů, dvojitou přesnost v plovoucí řádové čárce. A nakonec velikým mezníkem v GPGPU (general-purpose computing on graphics processing unit) se stala zcela nová architektura Fermi, která přinesla řadu inovací jako například zrychlení dvojité přesnosti plovoucí řádové čárky, podpora ECC pamětí, nebo L1 a L2 cache paměti.[8]

1.3 Motivace

Programování na grafických kartách je vhodné zejména pro vědecké a výpočetně složité výpočty, kdy se díky GPU doba výpočtu několikanásobně zkrátí. A to především proto, že grafická karta je navržena pro silnou paralelizaci a je také v některých operacích mnohem rychlejší než CPU. Proto programování na GPU je pro čím dál víc aplikací zajímavější.

Také Bc. Aleš Svoboda si vybral toto téma a začal na něm pracovat v rámci své bakalářské práce [2]. Vybral vhodný hardware, který bude vhodný pro programování na grafické kartě a na tomto vybraném počítači nainstaloval operační systémy Linux i Windows. Na oba systémy nainstaloval příslušný software pro programování na GPU. Dále vybíral jazyk, ve kterém se bude programovat. Vybíral z jazyků CUDA C a OpenCL, které vybraná grafická karta nVidia podporuje. Oba jazyky stručně charakterizoval. Vybral si OpenCL hlavně z důvodu multiplatformnosti a nezávislosti na výrobci. Následně vytvořil programy transponování a násobení matic, na kterých popsal, jak vytvořit program pro paralelní programování na GPU. Popsal také příkazy OpenCL použité v programech. Na závěr porovnal rychlosti běhu těchto programů na GPU a CPU.

Tato bakalářská práce navazuje na práci Bc. Aleše Svobody Výpočty na grafických kartách. Počítač (PC), na kterém se programovalo v této práci, byl převzat od Bc. Aleše Svobody. Byl tedy již kompletně připravený k užívání. Kromě tohoto PC jsem později začal využívat také výpočetní centrum MetaCentrum s klastrem gram.zcu.cz určeným pro výpočty na grafických kartách. Přibyla tedy i možnost porovnání výpočetních rychlostí dvou grafických karet mezi sebou. A nejen to. Společnost nVidia totiž rozděluje grafické karty do rodin, podle jejich využití:

- GeForce – zábava
- Quadro – profesionální vizualizace
- Tesla – paralelní výpočty a programování

Grafická karta Tesla je na klastru Gramu a na PC je GeForce. Naskýtá se tak příležitost otestovat o kolik je lepší Tesla než GeForce, která je ochuzená o určité vlastnosti a výkon ve dvojité přesnosti plovoucí řádové čárky.[11]

Motivací ze strany vedoucího práce pro vypsání tohoto tématu a vypracování práce je i ryze praktická snaha zdokumentovat a otestovat programování na GPU na takové úrovni, aby mohl převést vlastní CPU paralelní kódy na GPU paralelní. Výsledky této bakalářské práce mají vliv i na rozhodování o nákupu výpočetní techniky či přímo grafických karet pro modernizaci výpočetních klastrů používaných vedoucím práce a jeho studenty. Tomuto cíli odpovídá i zvolený modelový program - výpočet energie konfigurace molekul. Uvědomíme-li si, že molekulárně dynamické simulace vedoucího práce trvají typicky 10-30 dní na čtyřjádrovém CPU, znamená urychlení díky GPU významné zkrácení výpočtu.

1.4 Cíle práce

Cílem této bakalářské práce je blíže se seznámit s programováním na GPU. Dále vytvořit netriviální paralelní program pro GPU. Na něm se budou provádět optimalizace a také testovat rychlosti běhu programu na GPU a CPU. Na programu také popsat základy programování na GPU a podělit se o rady a zkušenosti, které byly během programování získány. Cílem je také, aby čtenář této práce, za pomocí přiložených zdrojových kódů, byl schopný naprogramovat svůj vlastní paralelní program pro GPU.

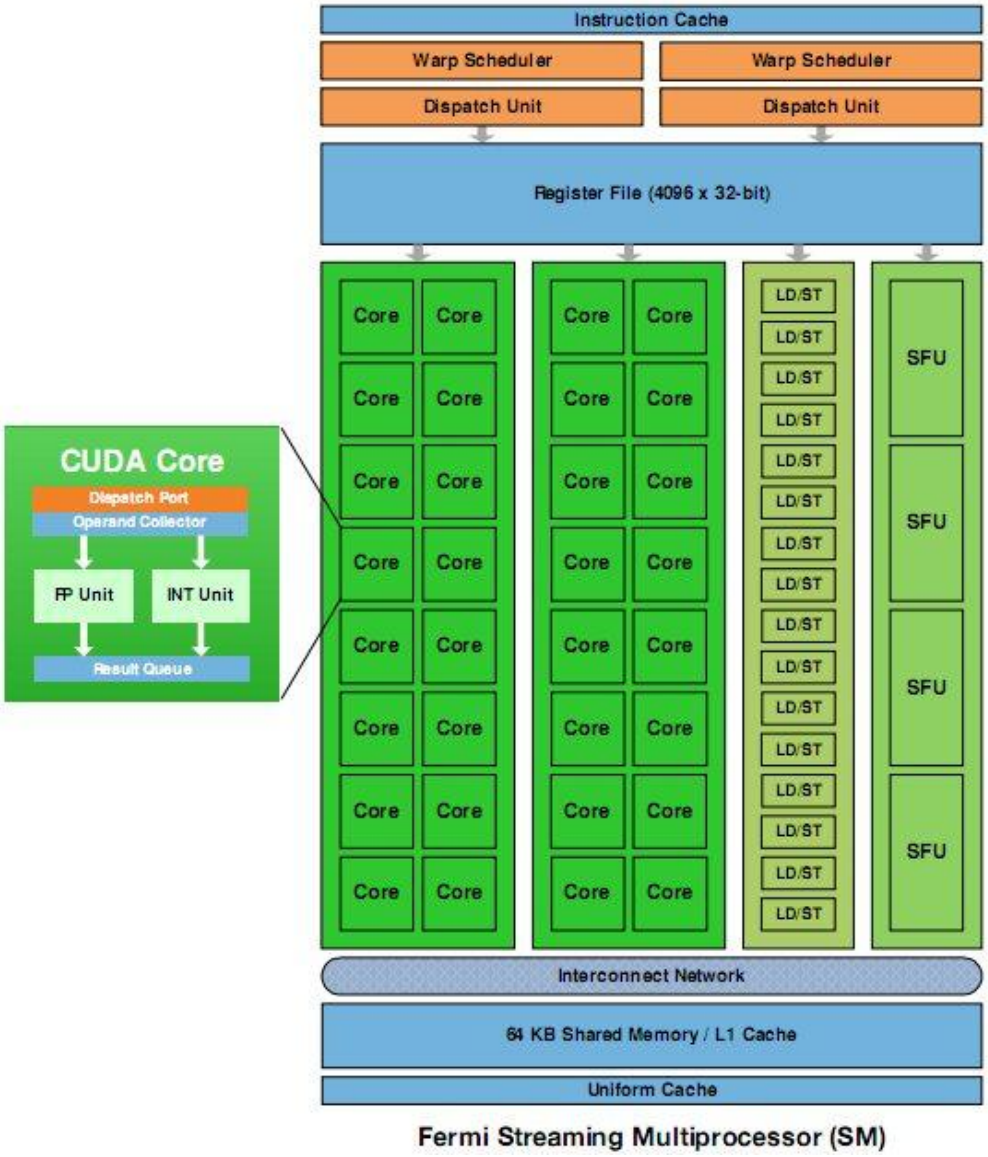
2 Detail GPU

Grafická karta z hlediska výpočetního stroje se skládá ze streaming multiprocesorů (compute unit) a streaming procesorů (známých také jako Cuda core, či scalar procesor). V OpenCL se používají názvy compute unit a scalar procesor, ale často se lze setkat s názvy streaming multiprocesor a streaming procesor, proto je tato práce bude také používat. Tyto jednotky komunikují s důležitou složkou architektury, kterou je hierarchie paměti.[20]

2.1 Streaming multiprocesor

Streaming multiprocesor (SM) je blok prvků, kde každý prvek má svou funkci. Na obrázku 1 je zobrazeno schéma streaming multiprocesoru Fermi.[13]
Streaming multiprocesor obsahuje:[13]

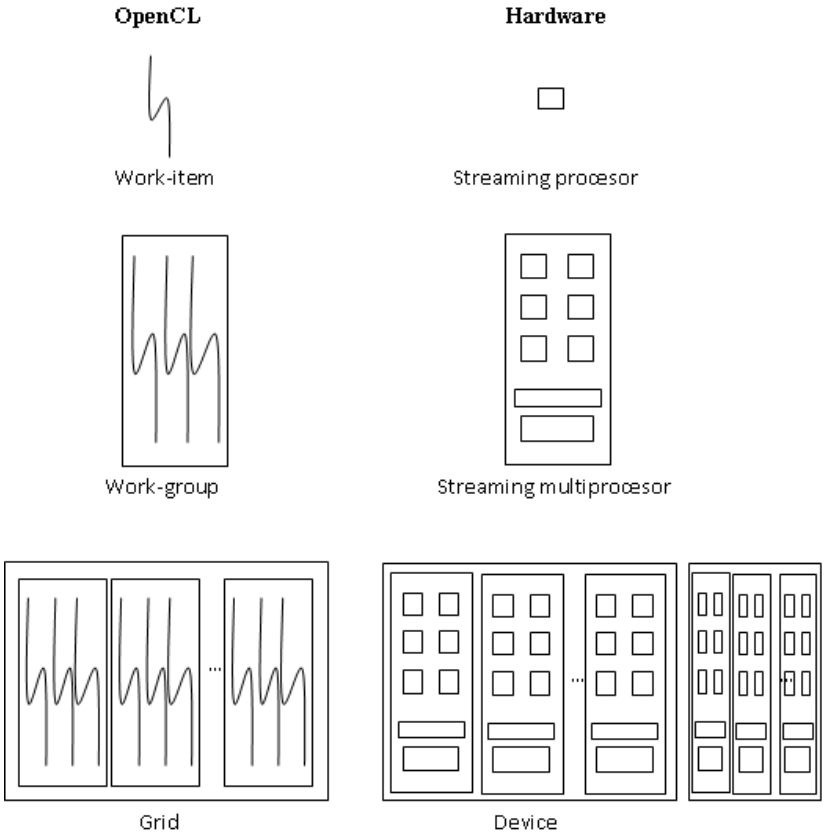
- Streaming procesory
- Jednotky pro dvojitou přesnost (double precision unit)
- Jednotky pro speciální funkce
    - provádí transcendentální instrukce jako inverzní funkce, sin, cos, odmocninu
    - výpočty jsou s jednoduchou přesností (pokud to nepodporuje přímo architektura)
- Sdílenou paměť v rámci SM - v našem případě L1 cache a shared memory
- Register file
    - úložiště 128 KB umožňuje mít průměrně 21 registrů na vlákno při plném obsazení plánovače (scheduler).
    - je řízen hardwarově
- Instruction cache
- Warp Schedule
    - 32-vstupní warp plánovač pro maximálně 1024 vláken na SM.
    - warp schedulers a dispatch units vysílají warpy a instrukce z různých vláken
- Dispatch unit (odesílací jednotka)
    - warp schedulers a dispatch units vysílají warpy a instrukce z různých vláken
    - informuje například, že streaming procesor je volný, nebo čeká na další data. Celou dobu tak zajišťuje efektivitu



Obrázek 1: Streaming multiprocesor (přejato z [13])

2.2 Softwarové uspořádání vs. hardwarové

Tato část je věnována tomu, jak je softwarové provedení OpenCL namapované na hardware. Nejlépe bude mapování vidět na obrázku 2. Kód vykonávaný vláknem (work-item), je spouštěn na streaming procesoru. Vlákna se slučují do work-group a streaming procesory do streaming multiprocesorů. Pak tedy work-groupy zpracovávají streaming multiprocesory. Několik souběžných work-group může být umístěno na jednom SM a nemohou se stěhovat mezi SM. Kernel je spouštěn jako grid (skupina work-group) a pouze jeden kernel se může výkonávat na zařízení (device) současně.[14, 22]



Obrázek 2: OpenCL vs hardware (přejato a změněno z [14])

3 Architektura

Pro výpočty na grafických kartách jsou důležitými parametry počet streaming procesorů (CUDA cores), streaming multiprocesorů i velikost pamětí. Ovšem neméně důležité jsou, podobně jako u procesorů, rychlosti sběrnic, frekvence jader a pamětí.

3.1 Parametry PC

3.1.1 Grafická karta

Mezi grafické karty třídy Fermi od NVidia patří mimo jiné i GIGABYTE GTX 560 Ti Ultra Durable 1GB, která má jádro GF 114. Přehled většiny parametrů tohoto jádra nalezneme v obrázku 3. [15]

Processing Units	Graphics Processing Clusters	2
	Streaming Multiprocessors	8
	CUDA Cores	384
	Texture Units	64
	ROP Units	32
Clock Speeds	Graphics Clock (Fixed Function Units)	822 MHz
	Processor Clock (CUDA Cores)	1644 MHz
	Memory Clock (Data rate)	4008 MHz
Memory	L2 Cache Size	512KB
	Total Video Memory	1024MB GDDR5
	Memory Interface	256-bit
	Total Memory Bandwidth	128.3 GB/s
Fillrate	Texture Filtering Rate (Bilinear)	52.6 GigaTexels/sec
Physical & Thermal	Fabrication Process	40 nm
	Transistor Count	1.95 Billion
	Connectors	2 x Dual-Link DVI-I 1 x Mini HDMI
	Form Factor	Dual Slot
	Power Connectors	2 x 6-pin
	Recommended Power Supply	500 Watts
	Thermal Design Power (TDP)[1]	170 Watts
	Thermal Threshold[2]	100° C

Obrázek 3: GeForce GTX 560 Ti (přejato z [15])

V obrázku 4 je vidět uspořádání jednotlivých komponent ve streaming multiprocesoru na jádře GF 114. Více informací o těchto komponentách je možné nalézt v kapitole o Streaming multiprocesorech (Streaming multiprocesor).

Obrázek 4: Jádro GF 114 (přejato z [13])

3.1.2 Procesor

PC má procesor Intel(R) Core(TM) i5-2310 CPU @ 2.90GHz se čtyřmi jádry. Ukázalo se, že jádra jsou podtaktována a to s frekvencí 1600 MHz z celkově možných 2900 MHz. Dále disponuje cache pamětí o velikosti 6144 KB (6 MB) a celkovou RAM pamětí 4000 MB. Do budoucnosti by bylo možné uvážit přetaktování procesoru.

3.2 Parametry výpočetního klastru Gram

Klastr gram.zcu.cz, který je součástí Metacentra obsahuje 10 uzlů. Každý z nich má procesor 2x 8 jader a čtyři grafické karty nVidia Tesla M2090. Parametry klastru gram.zcu.cz se nalézají v tabulce 1. Vlastníkem těchto uzlů je CESNET.[23]

Tabulka 1: Parametry klastru gram.zcu.cz (přejato z [23])

CPU	2x 8-core Intel Xeon E5-2670 2,6GHz
RAM	64GB
Disk	2x 600GB 10k SATA III,4x SSD 240GB
Net	Ethernet 1Gb/s, Infiniband QDR
poznámka	4x nVidia Tesla M2090 6GB
Vlastník	CESNET

3.2.1 Grafická karta

Grafická karta Tesla M2090 je z třídy M, která je určená pro servery a je založená na architektuře Fermi. Tato karta zvládne až 665 gigaflops (FLOPS = počet operací v plovoucí řádové čárce za sekundu) s dvojitou přesností a 1331 gigaflops s jednoduchou. Rychlost výpočtů s plovoucí řádovou čárkou je tedy pro dvojitou přesnost poloviční oproti jednoduché. Detailní popis této karty je v tabulce 2.[17]

Tabulka 2: Parametry grafické karty Tesla M2090 (přejato z [16])

General	
Chip	T20A GPU
Packaged Quantity	1
Device Type	GPU computing processor – Yes
Bus Type	PCI Express 2.0 x16
Graphics Processor / Vendor	NVIDIA Tesla M2090
Core Clock	1300 MHz
CUDA Cores	512
Features	Error Correcting Codes (ECC) Memory, Nvidia CUDA technology, NVIDIA Parallel DataCache, Nvidia GigaThread technology
Memory	
Video Memory Installed	6 GB
Technology	GDDR5 SDRAM
Effective Clock Speed	1.85 GHz
Bus Width	384-bit
System Requirements	
Peripheral / Interface Devices	Adjacent PCI slot
Miscellaneous	
Compliant Standards	FCC, ICES, cUL, C-Tick, VCCI, KCC, BSMI
Depth	9.8 in
Height	4.4 in

3.2.2 Procesor

Procesor jednoho uzlu gram.zcu.cz obsahuje 2 procesory po 8 jádrech. Konkrétně se jedná o Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz. Skutečná frekvence jader odpovídá 2601 MHz. Disponuje cache pamětí o velikosti 20480 KB (20 MB) a celkovou pamětí RAM 66096164 KB (64547 MB).

3.3 Zařazení grafických karet

CUDA (OpenCL) funguje na grafických kartách nVidia od řady G80, včetně rodiny GeForce, Quadro a Tesla. Tesla i Quadro je vyvinuto z řady G80. Tato část bakalářské práce se zaměří na rodiny GeForce a Tesla. Od G8x/G9x/G2xx, Fermi až po Kepler. Z tabulky 3 je možné vypozorovat, jak v průběhu času se téměř všechny parametry grafické karty zvyšovaly. V levé části tabulky je nejstarší třída grafických karet (G80) a směrem doprava jsou postupně třídy novější a zároveň s lepšími parametry.[8]

Zpočátku se nevěnovala pozornost dvojité přesnosti plovoucí řádové čárky (double precision) vůbec a nyní je čím dál více podporována. Konkrétně dvojitá přesnost začala být podporována až od třídy GT200[1]. Například dvojitá přesnost u třídy Tesla M má 665 gigaflops, ale u Kepleru již dosahuje více než 1 teraflops.[8, 17, 18]

Vzniká zde určitá nutnost zařadit a porovnat naše dvě používané karty. Jak jsou vlastně dobré či špatné oproti ostatním? V tabulce 3 jsou grafické karty vybrány tak, aby v jednotlivých třídách byly zastoupeny zhruba karty s nejhoršími, se středními a nejlepší parametry. Podle tabulky 3 je tedy možné říci, že používaná grafická karta GIGABYTE GTX 560 Ti není jednou z nejlepších karet s jádrem třídy Fermi, ale není rozhodně ani nejhorší. Zatímco karta na uzlu gram.zcu.cz Tesla M2090 je ta nejlepší dostupná karta od nVidia z třídy M.[17, 18]

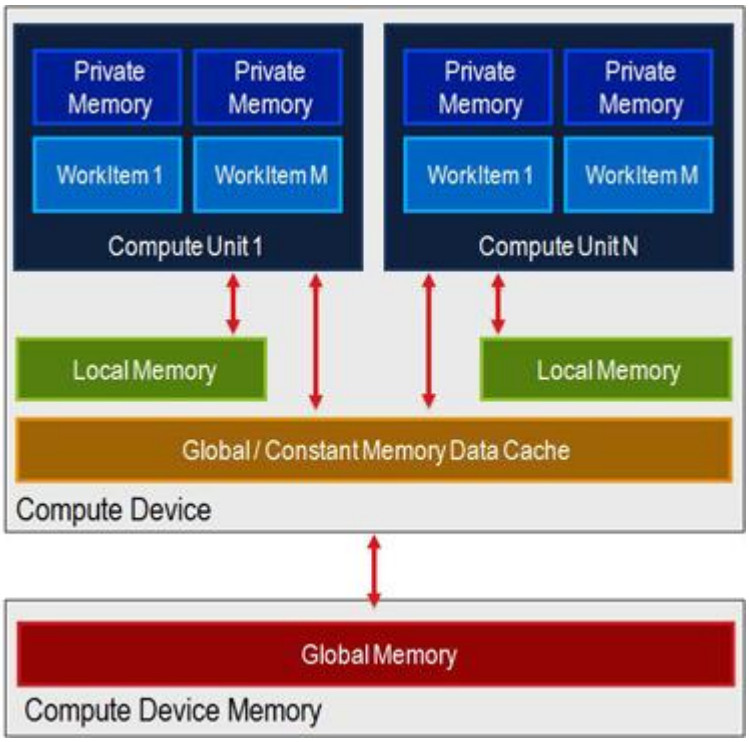
[1] Do třídy GT200 patří např. karty GeForce GTX 260, GTX 275, GTX 280, GTX 285, GTX 295, Tesla C/M1060, S1070 nebo Quadro CX, FX 3/4/5800

Tabulka 3: Zařazení grafických karet. X, funkce není podporována. -, parametr se nepodařilo dohledat (www.nvidia.com a www.geforce.com)

	G80		GT200	Fermi(Tesla C2050, C2070)			Tesla M	Tesla Kepler		
	GeForce 8500 GT	GeForce 8800 Ultra	GeForce GTX 280	GeForce GTX 460	**GIGABYTE GTX 560 Ti**	GeForce GTX 580	**Tesla M2090**	Tesla K10[a]	Tesla K20X	Tesla K20
Streaming processors	16	128	240	336	**384**	512	**512**	2x1536	2688	2496
Double precision floating point performance(Gflops)	X	X	78	-	**-**	-	**665**	190	1310	1170
Single precision floating point performance(Gflops)	-	576	933	1045	**1263**	1581	**1331**	4580	3950	3520
Total memory(MB)	256	768	1000	1000	**1000**	1536	**6000**	6000	6000	5000
Memory bandwidth(GB/s)	12.8	103.7	141.7	108	**128.3**	192.4	**177**	320	250	208
Price(Kč)	-	-	-	-	**4000**	9600	**105644**	59000	145175	70000

4 Paměťový model

Každé vlákno může použít jako úložný prostor privátní, lokální, konstantní a globální paměť. Ovšem každá paměť má jiné vlastnosti. Na obrázku 5 je schéma paměti grafické karty. Červené šipky popisují, jak spolu paměti komunikují, resp. jak paměti komunikují se streaming procesory.[3]



Obrázek 5: Paměťový model (přejato z [20])

4.1 Privátní paměť

Každé vlákno (thread) má svou vlastní privátní paměť, kterou používá a žádné jiné vlákno nemá přístup k datům ostatních vláken. Paměť je velice rychlá, ale malá a nepotřebuje synchronizační primitiva. Alokuje se a je rozdělena při kompilování.[3]

Když je privátní paměť zaplněna, dojde k přetečení paměti do L1 cache. Jestliže GPU nemá cache paměť, přeteče do globální paměti, což způsobí významný pokles výkonnosti.
Privátní paměť je v hardwaru mapována do paměťového prostoru nazvaného Register file.[3]

Příklad deklarace privátní proměnné:[5]
Host:
int elko = 4;
clSetKernelArg(kernel, 0, sizeof(elko), &elko);
Kernel bude vypadat následovně:
__kernel void operace(int L, ...) {
...
}

Složitější příklad deklarace privátní proměnné:[5]
float pole[4] = {0.0f, 1.0f, 2.0f, 3.0f};
clSetKernelArg(kernel, 0, sizeof(pole), pole);
Kernel nemůže přistupovat k privátním datům jako čtyř-prvkové pole, protože privátní argumenty nemohou být pointry (nemají ukazatel). Ale data mohou být přístupná jako float4 vektor, jak je ukázáno v následující kernel funkci:
__kernel void operace(float4 vektor, ...) {
...
}

4.2 Lokální paměť

Lokální pamět je mnohem rychlejší než globální pamět. Proto v případě, že používáme stejný kus dat několikrát, je lepší načíst data jednou z globální paměti a pak přistupovat do lokální, než přistupovat vícekrát pro stejný kus dat do globální paměti. Každá skupina (work-group) má svou vlastní lokální paměť, kterou všechna vlákna spolu sdílí a mohou si ji navzájem přepisovat. Tato pamět umožňuje tedy čtení i zapisování všem vláknům ve skupině. Ale vlákna nemohou číst ani zapisovat data do lokální paměti jiných skupin. Lokální paměť je identifikována označením *__local*.[19, 3]

Lokální adresový prostor je možné deklarovat jako ukazatele (pointery) na argumenty funkcí (včetně kernel funkcí) a nebo proměnné deklarované uvnitř funkcí. Proměnné deklarované jako argumenty kernel funkce mohou být alokovány v lokálním adresovém prostoru, ale s několika omezeními:

- tyto deklarace proměnných se musí objevit v argumentech kernel funkce a proměnné nemohou být inicializovány.
- proměnné jako ukazatele argumentu i proměnné uvnitř funkcí žijí pouze po dobu vykonávání kernelu skupinami.

Lokální paměť je v hardwaru mapována do shared memory. Při překročení kapacity této paměti (v našem případě velikosti 16/48 KB) přeteče do L2 cache (cache hierarchie je implementována u nVidie od třídy Fermi a pozdějších). Velikost lokální paměti je možné zjistit pomocí clGetDeviceInfo s parametrem CL_DEVICE_LOCAL_MEM_SIZE. Velikosti L1 i L2 cache jsou udávány na jeden streaming multiprocesor.[3, 19]

Příklad deklarace lokální proměnné float, která má 7 prvků:[5]
Následující příkaz vyhradí v kernelu prostor pro lokální 7prvkovou proměnnou:
*clSetKernelArg(kernel, 0, sizeof(float)*7, NULL);*
Teď v argumentu kernelu je možné mít lokální proměnnou (máme-li více argumentů, je nutné pamatovat na zachování pořadí argumentů v hostu i argumentů kernelu):
__kernel void operace(__local float pole, ...) {*
...
}

4.3 Konstantní paměť

Tato paměť je určena pouze pro čtení a je zejména specializovaná pro broadcastování dat. Jedním z důvodů její rychlosti je cachování, tudíž opakované čtení ze stejné adresy je velmi

rychlé. Konstantní proměnné jsou k dispozici ke čtení všem vláknům v průběhu vykonávání kernelu.[12]

Konstantní paměť je identifikována označením __constant a je požadována inicializace této proměnné. Velikost paměti je možné zjistit pomocí clGetDeviceInfo s parametrem CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE.[3, 20]

4.4 Globální paměť

Je největší ze všech pamětí a je také zároveň nejpomalejší. Všechna vlákna mohou číst/zapisovat data podle nastavení příkazu *clCreateBuffer* v hostu:[3]

- je-li proměnná nastavena pro čtení a je poslána z hostu do kernelu, je možné v kernelu z globální paměti pouze číst.
- pokud je globální proměnná vyhrazená pouze pro zápis, pak v kernelu lze do ní zapsat pouze jednou. Zapisují se do ní konečné výsledky, které se přenesou zpět do souboru host.
- u proměnných deklarovaných v kernelu lze zapisovat i číst.

Globální paměť je identifikována označením __global. Velikost paměti je možné zjistit pomocí clGetDeviceInfo s parametrem CL_DEVICE_GLOBAL_MEM_SIZE (hodnota je vracena jako cl_ulong v bytech).[3]

5 Získávání informací o zařízení

Ze specifikací výrobců, z webu, atd. je možné se dozvědět hodně informací o grafické kartě. Například zařazení grafické karty do určité skupiny (podle výkonnosti, funkce) nebo její vlastnosti. Použití příkazů OpenCL, sloužící k získávání některých z těchto informací, může ovšem nabídnout i jiná užitečná data (např. zda je podporována dvojitá přesnost, kolik je k dispozici streaming multiprocesorů aj.). Některé užitečné dotazy jsou v příloze ve zdrojovém kódu výpočtu energie vody. Příkazy pro získání informací o zařízení (device) jsou na řádcích H106 – H129, o platformě na řádcích H100 – H104 a o kernelu na H151 – 168.[3]

6 Datové typy

Typy v OpenCL jazyce jsou typy používané v kernelu a API typy pro aplikaci jsou typy používány u příkazů na zařízení. A právě většina datových typů může být použito programovacím jazykem OpenCL a zároveň také aplikací.[21]

6.1 Skalární datové typy

Programovací jazyk OpenCL C je založen na specifikaci jazyka C ISO/IEC 9899:1999, také známé jako C99. Má ovšem také různá omezení a rozšíření. V následující tabulce 4 jsou popsány integrované skalární datové typy v OpenCL a odpovídající si datové typy pro aplikaci.[21]

Tabulka 4: Skalární datové typy (přejato z [21])

Typ v jazyce OpenCL	Popis	API typ pro aplikaci
Bool	Podmíněný datový typ, který může nabývat hodnot true, nebo false. True nabývá integerové hodnoty 1 a false integerové hodnoty 0.	n/a
Char	8-bit integer se znaménkem.	cl_char
unsigned char, uchar	8-bit integer bez znaménka.	cl_uchar
Short	16-bit integer se znaménkem.	cl_short
unsigned short, ushort	16-bit integer bez znaménka.	cl_ushort
Int	32-bit integer se znaménkem.	cl_int
unsigned int, uint	32-bit integer bez znaménka.	cl_uint
long	64-bit integer se znaménkem.	cl_long
unsigned long, ulong	64-bit integer bez znaménka.	cl_ulong
float	Jednoduchá přesnost float. Datový typ float musí odpovídat formátu ukládání dat jednoduché přesnosti IEEE 754.	cl_float
half	16-bit float. Datový typ half musí odpovídat formátu ukládání dat poloviční přesnosti IEEE 754-2008.	cl_half
size_t	Integer bez znaménka a výsledkem je operátor *sizeof*. Typ je 32-bit integer, jestliže CL_DEVICE_ADDRESS_BITS je definován v clGetDeviceInfo jako 32-bit a 64-bit pokud CL_DEVICE_ADDRESS_BITS v clGetDeviceInfo je 64-bit.	n/a
ptrdiff_t	Integer se znaménkem, jehož výsledek je odečtení dvou ukazatelů. Typ je 32-bit integer, jestliže CL_DEVICE_ADDRESS_BITS je definován v clGetDeviceInfo jako 32-bit a 64-bit pokud CL_DEVICE_ADDRESS_BITS v clGetDeviceInfo je 64-bit.	n/a
intptr_t	Integer se znaménkem s vlastností, která může jakýkoli platný ukazatel na void převést do tohoto typu, pak převede zpět na typ void a výsledek porovná s původním ukazatelem.	n/a

uintptr_t	Integer bez znaménka s vlastností, která může jakýkoli platný ukazatel na void převést do tohoto typu, pak převede zpět na typ void a výsledek porovná s původním ukazatelem.	n/a
void	Typ void zahrnuje prázdnou množinu hodnot; Je to neúplný typ, který nemůže být úplný.	void

6.2 Vektorové datové typy

O integrovaných vektorových datových typech můžeme obecně říci, že jsou definovány jako datové typy skalární (např. int, short, float, long, …) následovány číslem. Čísla za datovými typy určují počet prvků (dimenzi) vektoru. Podporovány hodnoty jsou 2, 3, 4, 8, 16.[21]

V následující tabulce 5 jsou popsány integrované vektorové datové typy v OpenCL a odpovídající si datové typy pro API.

Tabulka 5: Vektorové datové typy (přejato z [21])

Typ v jazyce OpenCL	Popis	API typ pro aplikaci
char*n*	8-bit integer vektor se znaménkem.	cl_char*n*
uchar*n*	8-bit integer vektor bez znaménka.	cl_uchar*n*
short*n*	16-bit integer vektor se znaménkem.	cl_short*n*
ushort*n*	16-bit integer vektor bez znaménka.	cl_ushort*n*
int*n*	32-bit integer vektor se znaménkem.	cl_int
uint*n*	32-bit integer vektor bez znaménka.	cl_uint*n*
long*n*	64-bit integer vektor se znaménkem.	cl_long*n*
ulong*n*	64-bit integer vektor bez znaménka.	cl_ulong*n*
float*n*	float vektor	cl_float*n*

6.3 Dvojitá přesnost a poloviční plovoucí desetinná čárka

OpenCL 1.0 přichází s podporou pro dvojitou (double precision) a poloviční (half floating point) přesnost plovoucí desetinné čárky jako volitelným rozšířením.[21]

Datový typ double musí odpovídat formátu ukládání dat dvojité přesnosti IEEE 754. Pro použití tohoto rozšíření v aplikaci, je nutné zahrnout do kódu kernelu příkaz *#pragma OPENCL EXTENSION cl_khr_fp64 : enable*. V tabulce 6 jsou vidět integrované skalární a vektorové datové typy, které poskytuje toto rozšíření.[21]

Tabulka 6: Datový typ double (přejato z [21])

Typ v jazyce OpenCL	Popis	API typ pro aplikaci
double	Dvojitá přesnost float.	cl_double
double2	2 prvky double vektoru.	cl_double2
double4	4 prvky double vektoru.	cl_double4
double8	8 prvků double vektoru.	cl_double8
double16	16 prvků double vektoru.	cl_double16

Použije-li se half a half*n*, v programu bude potřeba uvést příkaz *#pragma OPENCL EXTENSION cl_khr_fp16 : enable*. Toto rozšíření nabídne následující seznam integrovaných skalárních i vektorových datových typů viz tabulka 7. [21]

Tabulka 7: Datový typ half (přejato z [21])

Typ v jazyce OpenCL	Popis	API typ pro aplikaci
half2	2 prvky vektoru poloviční přesnosti plovoucí desetinné čárky.	cl_half2
half4	4 prvky vektoru poloviční přesnosti plovoucí desetinné čárky.	cl_half4
half8	8 prvků vektoru poloviční přesnosti plovoucí desetinné čárky.	cl_half8
half16	16 prvků vektoru poloviční přesnosti plovoucí desetinné čárky.	cl_half16

7 Programování

Všechny použité příkazy OpenCL jsou k nalezení v příloze ve zdrojových kódech. Pro zjištění detailů o jednotlivých příkazech OpenCL odkazuji na web khronos.org, nebo bakalářskou práci Bc. Aleše Svobody Výpočty na grafických kartách, kde jsou jednotlivé argumenty důležitých příkazů popsány.

7.1 Popis programu

Popisován bude program na výpočet energie vody vytvořený v OpenCL. Popis bude rozdělen na host a kernel. Tyto dvě části budou popsány vždy postupně od začátku až do konce, zaměřují se na řádky a příkazy specifickými pro výpočet na GPU. Kompletní výpis programu je v příloze A.1.

7.1.1 Host

Ze všeho nejdříve je potřeba importovat knihovnu OpenCL:

```
H6      #include "CL/cl.h"
```

Na řádku H8 definujeme počet vláken. První možností je přiřadit programu takový počet vláken, který je schopen využít. Druhou možností je, nastavit optimální počet vláken viz Závislost času na počtu vláken. Pokud se nastaví menší počet vláken než je streaming procesorů, zdroje GPU nebudou plně využity.

Deklarace polí, která budou následně použita k přenosu informací mezi hostem a kernelem, jsou na řádkách H14 a H15. Na řádce H21 se alokuje pole *sumace*.
Načtení paralelního zdrojového kódu (kernel), který je uložen v souboru *energy.cl*, do stringové proměnné (H23 – H41).
Čtení molekul vody ze souboru do pole *atomsPole* (H44 – H68).
H77 – H81, deklarace OpenCL proměnných, které využijeme v příkazech obsluhujících GPU.

Řádek H92 vrací seznam platforem:

```
H92     cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
```

Argumenty:

1. číslo udává, kolik nalezených platforem se může přidat do *&platform_id*.
2. seznam nalezených platforem.
3. počet dostupných platforem.

Na řádcích H94 a H95 je příkaz, který vrací seznam zařízení:

```
H94     ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_ALL, 1,
H95                 &device_id, &ret_num_devices);
```

Argumenty:

1. odkazuje na seznam platforem *&platform_id* z předchozího příkazu.
2. typ hledaného zařízení – nyní se hledají všechny typy zařízení.
3. číslo udává, kolik nalezených zařízení se může přidat do *&device_id*.
4. seznam nalezených zařízení.
5. počet dostupných zařízení (shodujících se s hledaným typem).

H97 – H127, dotazy na informace o platformě a zařízení.

Řádek H132 vytvoří kontext, který je používán pro správu objektů (např. objektů command-queue, memory, program, kernel) a pro vykonání kernelu:

```
H130      // kontext - rizeni OpenCL objektu jako jsou command_queue, memory,
        program a kernel.
H132      cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL,
        &ret); // 1 je pocet zarizeni; &device_id ukazatel na seznam zarizeni
```

Argumenty:

1. je možno vytvořit kontext pro konkrétní platformy (pokud by jich bylo více). Tento program má pouze jednu. Hodnota je NULL, protože není potřeba nic specifikovat.
2. číslo je počet zařízení v seznamu *&device_id*.
3. ukazatel na seznam zařízení *&device_id*.

4. funkce, která oznamuje informace o chybách, které nastanou v tomto kontextu. NULL – neoznamovat žádné chyby.
5. při oznámení chyby ve funkci předchozího argumentu, pošlou se uživatelem nadefinovaná data. Data jsou definována v tomto argumentu. NULL – nevrátí žádná data.
6. vrátí příslušný error kód do proměnné `&ret`.

Vytvoření command-queue, který vykonává operace nad OpenCL objekty.

```
H136    cl_command_queue command_queue = clCreateCommandQueue(context,
        device_id, CL_QUEUE_PROFILING_ENABLE, &ret);
```

3. argument nastaví vlastnost měření času

Vytvoření objektu program:

```
H139    cl_program program = clCreateProgramWithSource(context, 1,
H140          (const char **)&zdroj_kod,NULL, &ret);
```

Argumenty:

1. kontext
2. počet stringů se zdrojovými kódy.
3. string proměnná s paralelním zdrojovým kódem.
4. velikost stringu z 3. argumentu.
5. vrátí příslušný error kód do proměnné `&ret`.

Kompilace (a linkování) programu

```
H143    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
```

Argumenty:

1. proměnná objektu program.
2. počet zařízení uvedených v *&device_id*.
3. ukazatel na seznam zařízení.
4. ukazatel na string, ve kterém jsou přepínače a parametry k překladu programu.

```
H146    kernel = clCreateKernel(program, "energy", &ret);
```

Argumenty:

1. proměnná objektu program.
2. jméno hlavní funkce kernelu *__kernel void energy(...)* nacházející se na řádce K5.
3. vrátí příslušný error kód do proměnné `&ret`.

Na řádkách H149 – 164 jsou příkazy, které vrací informace o kernelu objektu.

Vytvoření bufferu, který vyhradí paměť v kernelu. Celý příkaz vrací memory objekt.

Alokace paměti a překopírování pole *atomsPole* z paměti do memory objektu. A uvnitř kernelu bude *atomsPole* pouze pro čtení. (obojí specifikováno v 2. argumentu)

```
        cl_mem vstupni_buffer2 =
H168  clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
      sizeof(cl_float)*maxnum, atomsPole, NULL);
H169    cl_mem vystupni_buffer_sumace =
      clCreateBuffer(context,CL_MEM_READ_WRITE, sizeof(cl_float)*maxnum, NULL,
      NULL);
```

V kernelu se může číst i zapisovat do tohoto memory objektu (specifikováno v 2. argumentu). Druhý memory objekt (H169) neodkazuje na žádnou proměnnou s daty (proto 4. argument je NULL), ale posílá se do kernelu (do GPU) prázdný, protože memory objekt bude využit pro zápis výsledků. Pozn. V principu lze poslat v tomto memory objektu potřebná data, pokud je nastaven jako *CL_MEM_READ_WRITE*. Ale je nutné dávat pozor, aby nebyly přepsány výsledky v době, kdy data budou ještě potřeba. Pro jednoduchost se tedy posílá memory objekt prázdný.

Nastavení argumentů kernelu:

2. argument je index argumentu a je nutné dodžet jeho pořadí. Např. argument s indexem 1, musí být ve funkci *__kernel void energy(...)* druhým argumentem

&vstupni_buffer2 a *&vystupni_buffer_sumace* jsou memory objekty vytvořené příkazem *clCreateBuffer*. 3. argument je velikost posílané proměnné a nastaví se v případě memory objektů jako *sizeof(cl_mem)*.

```
H171    //pro vykonani kernelu musi byt nastaveny argumenty
H172    ret = clSetKernelArg(kernel, 0, sizeof(cl_uint),&Pvlaken);
H173    ret = clSetKernelArg(kernel, 1, sizeof(cl_uint),&molekul);
H174    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem),
         &vystupni_buffer_sumace);
H175    ret = clSetKernelArg(kernel, 3, sizeof(cl_mem),&vstupni_buffer2);
H176    ret = clSetKernelArg(kernel, 4, sizeof(cl_float),&elko);
```

Vytvoření pole pro globální vlákna a vykonání již nastaveného kernelu:

```
H178    // nastavíme global work size - celkovy pocet vlaken a dimenzi
H179    size_t global[1];
H180    global[0]= Pvlaken; // nastaveni poctu vlaken
H181
H182    // prikaz na vykonani kernelu
H183    ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
H184              global, NULL, 0, NULL, &prof_event);
```

Vlákna budou uspořádána do jedné dimenze o 10000 vláknech

Vykoná se paralelní zdrojový kód určený pro GPU.

Argumenty příkazu *clEnqueueNDRangeKernel*:

1. command-queue, u kterého byla na řádce H136 nastavena vlastnost měření času pomocí *CL_QUEUE_PROFILING_ENABLE*. Nyní při spuštění výpočtu na GPU se bude měřit čas.
2. proměnná objektu kernel.
3. specifikuje dimenzi globálních vláken (zároveň i lokálních ve work-groupě). Maximální dimenze vláken je 3.

4. pole offset – od jakého vlákna začne výpočet (pro lokální vlákna ve work-groupě se nenastavuje offset. Ty začínají vždy od 0).
5. pole *global* definuje počet globálních vláken v každé dimenzi (v tomto případě v dimenzi 1).
6. pole definující počet lokálních vláken (v každé dimenzi). Počet globálních vláken musí být dělitelný počtem lokálních vláken. Pokud je argument NULL, nastaví se počet lokálních vláken automaticky.
7. udává, kolik je událostí v seznamu událostí z 8. argumentu.
8. seznam událostí na jejichž dokončení se má čekat.
9. identifikuje tuto událost. Na tuto proměnnou se bude odkazovat příkaz pro zjištění času běhu programu.

Událost (Event objekt) slouží pro:[5]

- host zpráva – událost může informovat o tom, že příkaz na zařízení byl dokončen
- příkaz synchronizace – událost může donutit příkazy k pozdržení jejich provedení, dokud jiná událost není dokončena
- zíkání informací – událost může sledovat, jak dlouho příkaz trvá

H187 a H190, synchronizační primitiva v hostu. Čeká se na dokončení výpočtu v kernelu.

```
H187    ret = clWaitForEvents(1, &prof_event);
H190    clFinish(command_queue);
```

Čeká se na dokončení všech příkazů v *command_queue*.

Čeká se na dokončení události, která je identifikována proměnnou *&prof_event* (proměnná identifikuje volání kernelu).

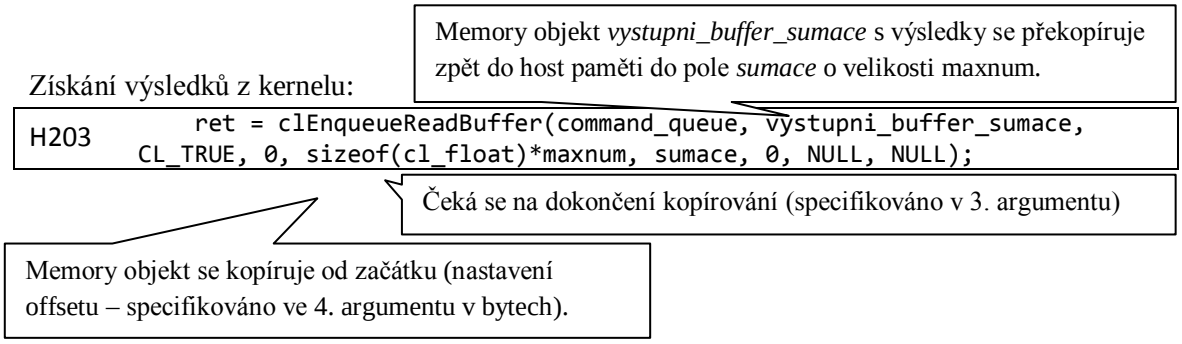
Doba běhu paralelního programu:

```
H192    // sbira informace o prikazu souvi
H193    //ziskani casu, kdy program vsto
H194    clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_START,
H195            sizeof(time_start), &time_start, NULL);
H196    // sbira informace o prikazu souvisejicim s udalosti
H197    //ziskani casu, kdy program vystoupil z kernelu
H198    clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_END,
H199            sizeof(time_konec), &time_konec, NULL);
H200    // vypocet doby behu kernelu
H201    total_time = (time_konec - time_start);
```

Příkaz se odkazuje na dokončenou událost, kterou bylo volání kernelu (specifikováno v 1. argumentu)

Čas začátku výpočtu se získá vlastnotí *CL_PROFILING_COMMAND_START* a vlastností *CL_PROFILING_COMMAND_END* čas skončení výpočtu.
Obě vlastnosti jsou ve 3. argumentu těchto příkazů. Časy jsou uloženy do proměnných *&time_start* a *&time_konec*.

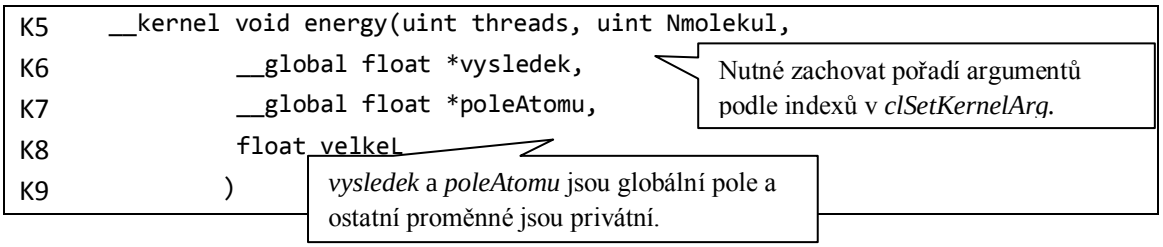
Získání výsledků z kernelu:



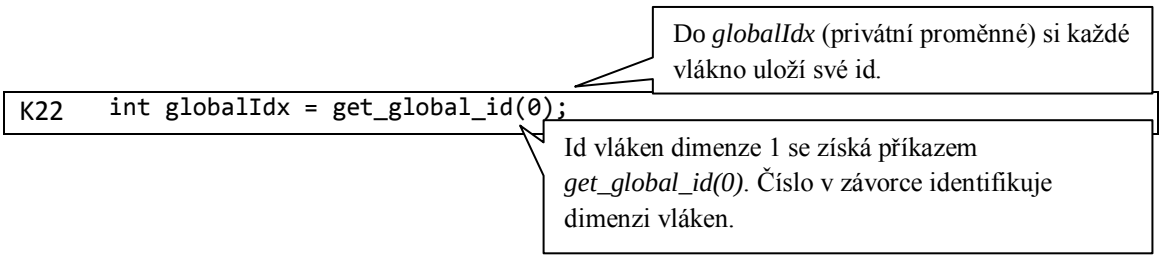
H205 – H215, výpis překopírovaného pole *sumace* z kernelu, ve kterém jsou uloženy výsledky jednotlivých vláken. V *sumace[0]* je uložen výsledek celkové energie vody.
Nakonec smazat objekty a naalokované proměnné (řádky H218 – H224).

7.1.2 Kernel

Hlavní funkce kernelu *__kernel void energy()*:



Deklarace fyzikálních konstant potřebných pro výpočet a pomocných proměnných je ve zdrojovém kódu na řádkách K11 – K19.



Vlákno vstupuje do for cyklu s hodnotou *i*, která je rovna id vlákna. Každou další iterací for cyklu hodnota *i* konkrétního vlákna se zvýší o počet vláken *threads*. Tím je zajištěno, že vlákna budou mít rovnoměrně rozdělenou práci. Pozn. Takže pokud je používán počet vláken >= počtu molekul, tento cyklus přestává být cyklem. Cyklus tedy souvisí i s tím, že program je schopen využít tolik vláken, kolik je molekul.

```
K25   energy = 0;
K26   for (i=globalIdx;i<(Nmolekul); i = i + threads)
K27       {
K28      int j,atomi,atomj,xyz,u;
K29      float ene=0;
K30      float shift[3];
K31      float dr[3];
K32      float mol1[9];
K33   // zkopirovani devitice(molekuly) do privatni pameti(promenne)
K34      for(u=0;u<(9); u++){
K35      mol1[u]= poleAtomu[(i*9)+u];
K36      }
K40      for(j=(i+1);j<Nmolekul;j++)
K41      {
K42        for(xyz=0;xyz<3;xyz++)
K43        {
K44          dr[xyz] = mol1[xyz] - poleAtomu[(j*9)+xyz];
K45          shift[xyz]=-velkeL* floor(dr[xyz]/velkeL+ .5);
K46          dr[xyz] = dr[xyz] + shift[xyz];
K47        }
K48        distsq= dr[0]*dr[0] + dr[1]*dr[1] + dr[2]*dr[2];
K49        ene = 0;
K50
K51        if(distsq<rcutsq){
K52          r6=sig6/(distsq*distsq*distsq);
K53          ene=4*eps*r6*(r6-1.);
K54
K55          for(atomi=0;atomi<3;atomi++)
K56          {
K57            for(atomj=0;atomj<3;atomj++)
K58            {
K59
K60         for(xyz=0;xyz<3;xyz++)
K61         {
               dr[xyz]= mol1[(atomi*3) + (xyz)] - poleAtomu[(j*9)+(atomj*3) +
K64   (xyz)] + shift[xyz];
K65         }
K66         dist = pow(pow(dr[0],2)+pow(dr[1],2)+pow(dr[2],2),0.5);
K67         ene = ene+elst*q[atomi]*q[atomj]/dist;
K68            }
K69          }
K70        }
K71        energy += ene;
```

Molekuly vody spolu interagují, proto probíhá součet energie přes všechny páry molekul.

Molekula *i* interaguje se všemi ostatními molekulami *j*, proto pro snížení přístupů do drahé globální paměti se použije privátní pole *mol1*.

Výpočet vzájemné vzdálenosti prvních atomů molekul *i* a *j*. Proměnná *shift*, která se k atomům přičítá, je posunutí.

V případě splnění podmínky (molekuly jsou dostatečně blízko), se začne vypočítávat energie páru molekul *i* a *j*.

Výpočet vzdáleností mezi všemi atomy *atomi* molekuly *i* od atomů *atomj* molekuly *j* probíhá ve for cyklech na řádcích K55 – K67, vzdálenost použita k výpočtu Coulombické interakce.

energy je příspěvek energie spočtený vláknem

ene je energie právě interagujícího páru molekul.

```
K72      }
K73       }
```

Následně každé vlákno vypočtenou energii zapíše do pole *vysledek* pod indexem svého id (řádka K75).

```
K75   vysledek[globalIdx] = energy;
K76
K77   // synchronizace (v argumentu
K78   barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
K79
K80   // sber vysledku od ostatnich vlaken provede pouze jedno vlakno
K81   if(globalIdx == 0) {
K82
K83   // secteni vysledku od vsech vlaken do vysledek[0]
K84   for(i=1;i<threads;i++)
K85           {
K86        vysledek[0] += vysledek[i];
K87           }
K88    vysledek[threads] = velkeL;
K89   }
```

Je nutné počkat, než všechny vlákna zapíší výsledky, protože se z pole *vysledek* budou částečné výsledky energií číst. K tomu slouží příkaz *barrier*.

Synchronizace lokálních i globálních proměnných

Jestli-že jsi vlákno s id nula, vykonej následující příkazy.

Vlákno 0 sečte všechny částečné výsledky a uloží je do *vysledek[0]*.

Protože kernel nepodporuje výpisy typu *printf*, je dobré udělat pole, které se kopíruje zpět do host paměti, větší než je potřeba a dát do nevyužité části např. kontrolní výsledky.

7.2 Optimalizace kernelu

7.2.1 Energie vody

Energie vody se vypočítá jako součet přes všechny páry molekul. Pro každý pár se spočítá kvadrát vzájemné vzdálenosti kyslíkových atomů, které současně slouží jako referenční atomy molekul. Je-li tato hodnota větší než stanovený parametr rcutsq=1.44, což odpovídá vzdálenosti 1,2 nm, použije se v molekulární dynamice standardní tzv. sférické ořízntutí potenciálu (cut-off) pro urychlení výpočtu, tj. tyto molekuly se považují za neinteragující a jejich vzájemná energie považována za nulovou. V opačném případě se spočte jak van der Waalsovská interkace mezi kyslíky, tak Coulombická energie počítaná ze všech atomárních kombinací atomů molekul.

Programování výpočtu energie vody probíhalo tak, že byl naprogramován funkční, ale neefektivní program a následně probíhaly optimalizace. Uvedené časy jsou pro velikost souboru 10k.gro s 1000 vlákny.

První optimalizací bylo v kernelu nevytvářet vícerozměrné pole, protože překopírování jednorozměrného pole do vícerozměrného znamená ztrátu času kopírováním. Užitek to může přinést jen programátorovi, kterému by se s vícerozměrným polem mohlo lépe pracovat. Zlepšení času bylo z 1.44 s na 1.03 s.

Druhým nedostatkem v kernelu bylo, že každé vlákno vstoupilo do for cyklu a následně se při každém přistoupení do pole zdlouhavě počítal jeho index.

```
for (k=0;k<(blok);k++){
for(o=((globalIdx * blok) + (k+1));o<nMolekul;o++)
    {
        ...
        dr[xyz] = vekt[globalIdx * blok + k][0][xyz] - vekt[o][0][xyz];
        ...
}
...
}
```

Cílem takového for cyklu bylo přiřadit každému vláknu blok pole. Jenomže nevýhoda byla, že musel být přesně stanoven počet vláken a podle toho pak dopočítávat i horní mez for cyklu (*blok*).

Efektivnějším řešením se ukázalo být, aby se id vlákna rovnou přiřadilo do proměnné for cyklu *i*. Proměnná *i* pak skáče po násobcích počtu vláken plus id vlákna. Díky této změně také lze libovolně nastavovat počet vláken a snížil se počet operací násobení. Po této úpravě doba běhu programu poklesla z 1.03 s na 0.69 s.

```
for (i=globalIdx;i<(nMolekul); i = i + PocetVlaken){
...
mol1[u]= poleAtomu[(i*9)+u];
...
}
```

Protože je do proměnné for cyklu i přiřazeno id vlákna, je v každém příkazu dopočítávajícím index pole o operaci násobení méně

Dále byly pokusy změnit adresový prostor, z globálního pole na lokální, či privátní pole. Ale z důvodu velikosti pole a relativně malé lokální i privátní paměti změnu nebylo možné provést. Tuto nedokonalost spočívající v neustálém přistupování do globální paměti (globálního pole), se podařilo částečně vyřešit. Řešením bylo malé privátní pole, do kterého se část globálního pole překopírovala (řádky kódu K34 – K36). Díky tomu se nemusí několikrát přistupovat do globální paměti pro stejný kus dat, ale přistoupí se do privátní paměti, která je rychlejší. Tím bylo dosaženo dalšího zlepšení času. Z původních 0.69 s na 0.67 s.

Protože v programu pro výpočet energie vody každá molekula počítá vzdálenost všech ostatních molekul, v jednom čase je zapotřebí dvou molekul. A dvě molekuly jsou vlastně dva kusy pole po devíti prvcích. Po předchozím úspěchu se nabízí udělat dvě malé privátní pole a tím program ještě urychlit. Nicméně dvě privátní pole neměla kýžený efekt. Celková doba běhu programu se zdvojnásobila.

Až později se začala testovat závislost doby běhu programu na vláknech. Proto tedy výše uvedené časy jsou pro 1000 vláken. Ale protože optimalizace probíhala za stejných podmínek (stejná velikost souboru a stejný počet vláken), časové rozdíly optimalizací jsou relevantní. Stroj pouze plně nevyužil zdroje, které má k dispozici. Nastavení počtu vláken je tedy také jedna z optimalizací programu.

Doba běhu optimalizovaného programu na 10000 vláknech byla 0.16 s.

Paralelní část kódu (kernel) energie vody byla také kompletně předělána pro vektory typu float3. Takový program vyžadoval překopírování jednorozměrného pole, které bylo posláno z hostu do kernelu, do pole vektorů. Ale jak bylo již zmíněno, kopírování polí je neefektivní a pokus o automatickou konverzi pole na pole vektorů selhal. Proto bylo od vektorů upuštěno.

7.3 Matematické funkce

O výpočet integrovaných matematických funkcí na grafické kartě se starají SFU (special function unit), které jsou umístěny na streaming multiprocesoru. Integrované funkce kernelu by měly být rychlejší než vytvořené vlastní funkce, ale za cenu nižší přesnosti výsledku. Existují také funkce s half precision jako například druhá odmocnina nebo logaritmy. Následující tabulka 8 ukazuje pouze příklady integrovaných funkcí. Více funkcí naleznete na stránkách khronos.org například v souboru quick reference card nebo přímo ve specifikaci OpenCL.[3]

Tabulka 8: Integrované matematické funkce

Název	Funkce v kernelu
sinus	sin(x)
přirozený logaritmus	log(x)
mocnina x^y	pow(x,y)
e^x	exp(x)
odmocnina z x^2+y^2	hypot(x,y)
absolutní hodnota	fabs(x)

8 Výsledky

8.1 Energie vody

Jedná se o testování časů běhu programu na výpočet energie vody. Testy se prováděly se soubory 10k.gro, 35k.gro a 100k.gro, kde název značí počet molekul vody. To znamená, že například v souboru 100k.gro je 100000 molekul vody.

Tabulka 9 je rozdělena na dvě části oddělené černou čarou. Horní část se věnuje výsledkům CPU v OpenMP na čtyřech vláknech a dolní část výsledkům GPU. Počet vláken ve skupině výsledků pro GPU byl vybrán takový, který dosahoval nejlepších časů. Dobrých časů se většinou dosáhlo u počtu 30000 vláken, ale tomu bude věnována následující část této práce.

Ve skupině časů pro GPU lze porovnat časy použitých typů, double nebo float, mezi sebou. U PC s grafickou kartou GeForce se zjistilo, že double precision je 1/8 rychlosti single precision. Gram, s kartou Tesla, má tento poměr rychlostí lepší. Výkon double dosahuje 1/2 rychlosti single precision.

Porovnání horní i dolní části tabulky 9 navzájem poskytne naopak představu o tom, o kolik je rychlejší GPU oproti CPU. Například program výpočtu energie vody při 100000 molekulách na CPU běží několik desítek sekund a na GPU se jedná už jen o jednotky sekund. Na menších datech tento rozdíl není tak veliký, ale i tak není zanedbatelný.

Tabulka 9: Výsledky energie vody

	icc, OMP, 4 vlákna			
soubor	10k.gro	35k.gro	100k.gro	
čas(s) float	1,56	15,1	120,3	Gram
čas(s) double	1,14	9,31	70,69	Gram
čas(s) float	1,46	10,54	76,7	PC
čas(s) double	2,05	14,79	111,78	PC
čas(s) float	0,2	0,43	1,68	Gram
čas(s) double	0,4	1,13	5,19	Gram
čas(s) float	0,16	0,53	2,14	PC
čas(s) double	1,23	4,11	a	PC
	GPU			

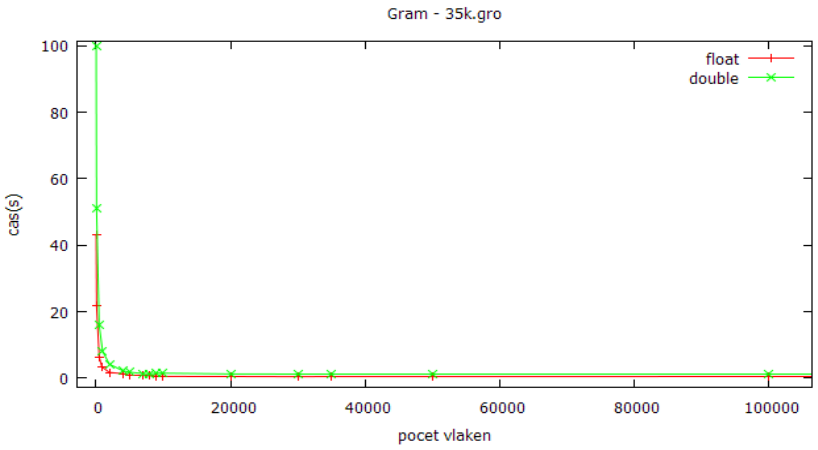
a – timeout kernelu. Výsledek nemohl být zaznamenán

8.2 Závislost času na počtu vláken

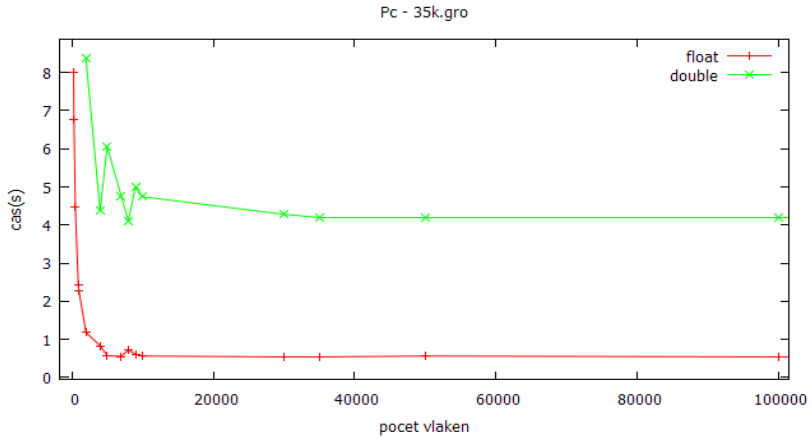
Většinou nevíme přesně, kolik GPU vláken na určitý program použít. Nejobecnější rady by byly podívat se na to, kolik vláken by mohl program teoreticky využít. Další radou je použít více vláken, než je streaming procesorů a metodou pokus omyl vyzkoušet na konkrétním programu optimální počet vláken. Proto se zabývají tímto problémem následující testy na obrázku 6 a 7. Testována je závislost času na počtu vláken. Z obrázků je vidět, že u obou počítačů zpočátku čas prudce klesá (a to u float i double), ale okolo 10000 vláken se klesající tendence času zvolňuje a postupně přechází v konstantní hodnoty.

Testována byla i data o velikosti 10000 a 100000 molekul a to s podobnými výsledky. Klesající tendence času se zvolňuje téměř vždy u 10000 vláken. K jedné zřetelnější výchylce dochází v testu se souborem 100k.gro u double. Čas v onom testu klesá, ale náhle při 10000 vláknech čas nepatrně vzroste a pak již s přibývajícím počtem vláken čas opět klesá.

Doporučení na základě těchto testů zní, že optimální počet vláken je 10000. Následně je dobré počet vláken experimentálně upřesnit a popřípadě tím nalézt ještě lepší výsledky času.



Obrázek 6: Gram - závislost času na počtu vláken



Obrázek 7: PC - závislost času na počtu vláken

8.3 Matematické funkce

Dalším předmětem testování je porovnání rychlostí integrovaných matematických funkcí kernelu na GPU a matematických funkcí na CPU. Testování proběhlo na grafických kartách Gramu a PC a na procesoru PC v OMP na čtyřech vláknech. A program, který testoval dobu výpočtů, byl jednoduchý. Do kernelu bylo posláno pouze dlouhé pole a následně se vypočetla příslušná matematická funkce. Výsledky se nalézají v tabulce 10. Uvedené časy v tabulce jsou v sekundách.

Testy jsou prováděny na integrovaných matematických funkcích až na jednu výjimku. Výjimkou je mocnina dvou zapsaná jako *x*x*. Snaha byla o porovnání nejen integrovaných funkcí mezi sebou, ale i vytvořených funkcí s funkcemi integrovanými.

Výsledky časů ukazují, že výpočet funkce *pow(x,2)* je o mnoho rychlejší v OMP na procesoru než na grafických kartách. Ovšem v ostatních funkcích GPU jasně vede. Pro zajímavost byly provedeny i testy na grafické kartě Gramu. Tesla by měla, podle předpokladu, být rychlejší než GeForce. A je vidět, že tomu tak je. Tesla je opět o něco rychlejší.

Na grafických kartách vlastní funkce *x*x* je rychlejší než integrovaná funkce *pow(x,2)*. Je tomu tak, protože *pow(x,y)* je obecná funkce, kde *x* a *y* mohou být i reálná čísla.

Když se programoval výpočet energie vody, zkusila se porovnat doba běhu programu s integrovanou funkcí *pow(x,3)* a následně s vlastní funkcí *x*x*x*. Výsledky časů byly téměř stejné.

Tabulka 10: Rychlosti matematických funkcí

	Pc - OMP	Pc - GPU	Gram - GPU
pow(x,2)	0.1 s	0.44 s	0.35 s
x * x	0.11 s	0.08 s	0.05 s
ln(x)	3.49 s	0.15 s	0.1 s
sin(x)	1.44 s	0.26 s	0.17 s

9 Závěr

Všechny cíle bakalářské práce byly splněny. Byl naprogramován paralelní program na výpočet energie vody, který se postupně optimalizoval. Provedené optimalizace byly obecné, lze jich tedy využít v každém programu. Po optimalizacích je program na GPU několikanásobně rychlejší než na CPU v OMP a s rostoucím počtem dat tento rozdíl v rychlostech prudce nárůstá. V testech bylo také ověřeno, že GPU na PC je omezená oproti GPU na Gramu v operacích s dvojitou přesností plovoucí řádové čárky.

Velkým lákadlem v OpenCL byl typ vektor. S vektory se na příkladu energie vody dobře pracovalo a zdrojový kód byl díky nim kratší a přehlednější. Ale protože automatická konverze na vektory nebyla možná, program s vektory byl méně efektivní než s poli a upustilo se od nich.

Testy závislosti počtu vláken na čase ukázaly, že vhodné nastavení je 10000 vláken a víc. Při tomto počtu je již GPU poměrně vytížené. Toto číslo je samozřejmě závislé na kvalitě hardwaru. Lze předpokládat, že lepší GPU bude potřebovat pro plné vytížení více vláken. Další možnosti, jak nastavit počet vláken, závisí na samotném programu. Využije-li program statisíce vláken, je možné je nastavit. Tomu bylo i v případě výpočtu energie vody, kdy využil i 100000 vláken. Výsledky ovšem ukázaly, že počet vláken překročil možnosti GPU a časy výpočtů se od 50000 vláken dále nezlepšovaly.

Na programu výpočtu energie vody byl popsán postup programování v OpenCL pro GPU. Tento popis lze využít jako návod pro vytvoření vlastního paralelního programu.

Seznam obrázků

Obrázek 1: Streaming multiprocesor 11
Obrázek 2: OpenCL vs hardware 12
Obrázek 3: GeForce GTX 560 Ti 13
Obrázek 4: Jádro GF 114 14
Obrázek 5: Paměťový model 18
Obrázek 6: Gram - závislost času na počtu vláken 33
Obrázek 7: PC - závislost času na počtu vláken 34

Seznam tabulek

Tabulka 1: Parametry klastru gram.zcu.cz 15
Tabulka 2: Parametry grafické karty Tesla M2090 15
Tabulka 3: Zařazení grafických kare 17
Tabulka 4: Skalární datové typy 21
Tabulka 5: Vektorové datové typy 22
Tabulka 6: Datový typ double 23
Tabulka 7: Datový typ half 23
Tabulka 8: Integrované matematické funkce 32
Tabulka 9: Výsledky energie vody 33
Tabulka 10: Rychlosti matematických funkcí 34

Bibliografie

[1] VOTAVA, Radek. *Spojení CPU + GPU - historie a smysl Fusion proti Core i3/ i5* [Online]. 30. 11. 2009 [Cit. 16. 4 2013.] Dostupné z <http://www.ddworld.cz/blogy/hardware-a-it/spojeni-cpu-gpu-historie-a-smysl-fusion-proti-core-i3-i5.html>.

[2] SVOBODA, A. *Výpočty na grafických kartách. České Budějovice*, 2012. 54 p. Bakalářská práce na Přírodovědecké fakultě Jihočeské univerzity. Vedoucí bakalářské práce RNDr. Milan Předota, Ph.D.

[3] MUNSHI, Aaftab. *The OpenCL Specification* [online]. c2010, poslední revize 1.6.2011 [cit. 16.1.2013]. Dostupné z: <http://www.khronos.org/opencl/>.

[4] *OpenCL Programming Guide* [online]. c2004, poslední revize 3.1.2006 [cit. 16.1.2013]. Dostupné z: <http://developer.nvidia.com/opencl>.

[5] SCARPINO, Matthew. *OpenCL in Action: HOW TO ACCELERATE GRAPHICS AND COMPUTATION*. 5. vyd. United States of America: Manning Publications Co. NY: Shelter Island, 2011. ISBN 9781617290176.

[6] manythreads. *OpenCL™ – Portable Parallelism* [online]. 17.9.2010. Dostupné z: <http://www.codeproject.com/Articles/110685/Part-1-OpenCL-Portable-Parallelism>. [cit. 16. 1. 2013].

[7] PETERKA, Jiří. *Coprocessor* [online]. [cit. 16. 4. 2013]. Dostupné z: <http://www.earchiv.cz/a92/a247c120.php3>

[8] PETERKA, Jiří. *Nvidia Fermi - Analýza nové generace GPU: Historie GPU Computingu* [online]. 1. 10. 2009. [cit. 16. 4. 2013]. Dostupné z: <http://pctuning.tyden.cz/hardware/graficke-karty/15131-nvidia-fermi-analyza-nove-generace-gpu?start=2>

[9] *COMPUTER HARDWARE Information about the computer CPU* [online]. [cit. 16. 4. 2013]. Dostupné z: <http://www.landley.net/history/mirror/intel/cpu.htm>

[10] WHITE, Stephen. *A Brief History of Computing- Microprocessors* [online]. [cit. 16. 4. 2013]. Dostupné z: <http://trillian.randomstuff.org.uk/~stephen/history/timeline-CPU.html>

[11] *Why Choose Tesla* [online]. 2013 [cit. 17. 4. 2013]. Dostupné z: <http://www.nvidia.com/object/why-choose-tesla.html>

[12] CÁRDENAS, M. *Constant Memory* [online]. Centro de Investigaciones Energ´eticas Medioambientales y Tecnol´ogicas, Madrid, 2011 [cit. 17. 4. 2013]. Dostupné z: <http://wwwae.ciemat.es/~cardenas/CUDA/T6-ConstantMemory.pdf>

[13] *NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi* [online]. 2013 [cit. 22. 4. 2013]. Dostupné z: <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>

[14] WOOLLEY, Cliff. *Introduction to OpenCL* [online]. 14. 4. 2011 [cit. 22. 4. 2013]. Dostupné z: <http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf>

[15] MIKHALTSEVICH, Victor. *NVIDIA GTX 560 Ti (GF 114)* [online]. 25. 1. 2011 [cit. 22. 4. 2013]. Dostupné z: <http://www.bjorn3d.com/2011/01/nvidia-gtx-560-ti-gf-114/>

[16] *NVIDIA Tesla M2090 GPU computing processor specs* [online]. [cit. 22. 4. 2013]. Dostupné z: <http://reviews.cnet.com/video-components/nvidia-tesla-m2090-gpu/4507-3025_7-35154088.html>

[17] *TESLA™ M-Class GPU Computing Modules Accelerat ing Science* [online]. [cit. 22. 4. 2013]. Dostupné z: <http://www.nvidia.com/docs/IO/43395/DS-Tesla-M-Class-Aug11.pdf>

[18] *TESLA™ C2050 / C2070 GPU Comput ing Processor* [online]. [cit. 22. 4. 2013]. Dostupné z: <http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf>

[19] MUNSHI, Aaftab, James Fung, Benedict Gaster, Dan Ginsburg a Timothy Mattson. *Programing with OpneCL C* [online]. 26. 1. 2011[cit. 22. 4. 2013]. Dostupné z: <http://www.informit.com/articles/article.aspx?p=1732873&seqNum=11>

[20] MANYTHREADS. *Part 2: OpenCL™ – Memory Spaces* [online]. 27. 9. 2010 [cit. 22. 4. 2013]. Dostupné z: <http://www.codeproject.com/Articles/122405/Part-2-OpenCL-Memory-Spaces>

[21] *OpenCL Reference Pages* [online]. [cit. 22. 4. 2013]. Dostupné z: <http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/>

[22] *CUDA Programming Model Overview* [online]. 2008 [cit. 22. 4. 2013]. Dostupné z: <http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf>

[23] *MetaCentrum virtual organization.* 15. 4. 2013 [cit. 22. 4. 2013]. <http://metavo.metacentrum.cz/cs/>

Příloha

A Zdrojové kódy

A.1 OpenCL – Výpočet energie vody

A.1.1 Host

```
H1   //Vypocet energie vody
H2   #include <stdio.h>
H3   #include <stdlib.h>
H4   #include <math.h>
H5   // potreba includovat knihovnu OpenCL CL/cl.h
H6   #include "CL/cl.h"
H7   //definujeme pocet vlaken
H8   #define Pvlaken 10000
H9   const int maxnum=900004; // maximalni potrebna delka pole
H10  const int LENGTH = 80;
H11  float L;
H12  int main ()
H13  {
H14    float atomsPole[maxnum];
H15    cl_float *sumace;
H16    cl_uint molekul;
H17    // elko je velikost boxu vody
H18    cl_float elko;
H19    cl_ulong time_start, time_konec;
H20    long total_time;
H21    sumace = malloc(sizeof(cl_float)*maxnum); // alokace promenne
H22    // nacteni zdrojoveho kodu kernelu do stringove promenne =======
H23    FILE *fp;
H24    char *zdroj_kod;
H25    long delka_souboru;
H26    long delka_cteni;
H27
H28    fp = fopen("energy.cl","r");
H29     fseek(fp,0L, SEEK_END);
H30     delka_souboru = ftell(fp);
H31     rewind(fp);
H32
H33     zdroj_kod = malloc(sizeof(char)*(delka_souboru+1));
H34     delka_cteni = fread(zdroj_kod,1,delka_souboru,fp);
```

```
H35       if(delka_cteni!= delka_souboru)
H36       {
H37          printf("Nelze přečist soubor\n");
H38          exit(1);
H39       }
H40      zdroj_kod[delka_souboru+1]='\0';
H41      fclose(fp);
H42      // nacteni zdrojoveho kodu kernelu do stringove promenne =======
H43
H44      //================cteni souboru s molekuly vody====================
H45      int i,j,natoms,nmol;
H46      FILE *fl;
H47      char line[LENGTH], nothing[LENGTH], name[20];
H48
H49      fl=fopen("10k.gro", "r");
H50      fgets(line, LENGTH,fl); //skip first line
H51      fgets(line, LENGTH,fl); sscanf(line,"%i",&natoms);
H52      nmol=natoms/3; printf("Number of molecules %i\n",nmol);
H53
H54      //nacteni molekul vody ze souboru do pole
H55      for (i=0;i<nmol;i++){
H56        for(j=0;j<=2;j++){
H57          fgets(line, LENGTH,fl);
H58          sscanf(line, "%s %s %s %f %f %f",nothing,nothing,nothing,
         &atomsPole[i*9 + j*3 + 0], &atomsPole[i*9 + j*3 +1], &atomsPole[i*9 + j*3
         + 2]);
H59        }
H60      }
H61
H62      printf("first line %f  %f
         %f\n",atomsPole[0],atomsPole[1],atomsPole[2]);
H63      fscanf(fl, "%f",&L); // read box size
H64      printf("Box size %f\n",L);
H65      fclose(fl);
H66      molekul = nmol;
H67      elko = L;
H68      //================cteni souboru s molekuly vody====================
H69
H70      //inicializace pole (ktere nakonec vrati z kernelu vysledky)
H71      for(i=0;i<maxnum;i++)
H72        {
H73       sumace[i]= 0; //data
H74        }
H75
```

```
H76   //opencl promenne potrebne pro obsluhu GPU
H77     cl_platform_id platform_id = NULL; // seznam platforem
H78     cl_device_id device_id = NULL;   // seznam zarizeni
H79     cl_uint ret_num_devices;  //pocet dostupnych zarizeni
H80     cl_uint ret_num_platforms; //pocet dostupnych platforem
H81     cl_event prof_event; // bude identifikovat udalost
H82
H83       //======promenne k informacim========
H84       char dname[500];
H85       size_t paramsi, p_size;
H86       cl_ulong long_entries;
H87       cl_uint entries;
H88       int bd;
H89       size_t locala;
H90       //====================
H91     // vraci seznam dostupnych platforem
H92     cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
H93     // vraci seznam dostupnych zarizeni na platforme
H94     ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_ALL, 1, &device_id,
      &ret_num_devices);
H95
H96
H97       /* obtain information about platform */
H98      clGetPlatformInfo(platform_id,CL_PLATFORM_NAME,500,dname,NULL);
H99      printf("CL_PLATFORM_NAME = %s\n", dname);
H100     clGetPlatformInfo(platform_id,CL_PLATFORM_VERSION,500,dname,NULL);
H101     printf("CL_PLATFORM_VERSION = %s\n", dname);
H102
H103     /* ====== query devices for information ======= */
H104         clGetDeviceInfo(device_id, CL_DEVICE_NAME, 500, dname,NULL);
H105         printf("Device #%d name = %s\n", bd, dname);
H106         clGetDeviceInfo(device_id,CL_DRIVER_VERSION, 500, dname,NULL);
H107         printf("\tDriver version = %s\n", dname);

H108  clGetDeviceInfo(device_id,CL_DEVICE_GLOBAL_MEM_SIZE,sizeof(cl_ulong),&lon
      g_entries,NULL);
H109         printf("\tGlobal Memory (MB):\t%llu\n",long_entries/1024/1024);

H110  clGetDeviceInfo(device_id,CL_DEVICE_GLOBAL_MEM_CACHE_SIZE,sizeof(cl_ulong
      ),&long_entries,NULL);
H111         printf("\tGlobal Memory Cache
      (MB):\t%llu\n",long_entries/1024/1024);

H112  clGetDeviceInfo(device_id,CL_DEVICE_LOCAL_MEM_SIZE,sizeof(cl_ulong),&long
      _entries,NULL);
H113         printf("\tLocal Memory (KB):\t%llu\n",long_entries/1024);
H114
```

```
        clGetDeviceInfo(device_id,CL_DEVICE_MAX_CLOCK_FREQUENCY,sizeof(cl_ulong),
        &long_entries,NULL);
H115           printf("\tMax clock (MHz) :\t%llu\n",long_entries);

H116    clGetDeviceInfo(device_id,CL_DEVICE_MAX_WORK_GROUP_SIZE,sizeof(size_t),&p
        _size,NULL);
H117           printf("\tMax Work Group Size- # work_itemu ve work_groupe(max
        vlaken v compute unit):\t%d\n",p_size);

H118    clGetDeviceInfo(device_id,CL_DEVICE_MAX_COMPUTE_UNITS,sizeof(cl_uint),&en
        tries,NULL);
H119           printf("\Number of streaming multiprocessors:\t%d\n",entries);
H120    //       CL_DEVICE_MAX_PARAMETER_SIZE

H121    clGetDeviceInfo(device_id,CL_DEVICE_MAX_PARAMETER_SIZE,sizeof(size_t),&pa
        ramsi,NULL);
H122           printf("\tNumber in bytes max size of argument can be passed to
        kernel(a take pouze 128 argumentu) :\t%d\n",paramsi);
H123    //       CL_DEVICE_MAX_WORK_ITEM_SIZES
H124           size_t workitem_size[3];
H125        clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_ITEM_SIZES,
        sizeof(workitem_size), &workitem_size, NULL);
H126        printf("\tCL_DEVICE_MAX_WORK_ITEM_SIZES(max # vlaken, lze uvest ve
        work_groupe v kazde dimenzi):\t%u / %u / %u \n", workitem_size[0],
        workitem_size[1], workitem_size[2]);
H127       /* ====== query devices for information ========== */
H128
H129      // vytvorime OpenCL context. Vytvoren s jednim ci vice zarizenimi. Je
        pouzit pri behu programu pro
H130      // rizeni OpenCL objektu jako jsou command_queue, memory, program a
        kernel.
H131      // A pro vykonavani vice kernelu na jednom ci vice zarizenich
        specifikovanych v contextu
H132      cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL,
        &ret); // 1 je pocet zarizeni; &device_id ukazatel na seznam zarizeni
H133
H134      // vytvorime command queue na specifickym zarizeni
H135    //Operace nad OpenCL objekty jsou vykonavany pouzitim command-queue
H136      cl_command_queue command_queue = clCreateCommandQueue(context,
        device_id, CL_QUEUE_PROFILING_ENABLE, &ret);
H137
H138      // nahrajeme zdrojový kod opencl kernelu - vytvori program objekt pro
        kontext
H139      cl_program program = clCreateProgramWithSource(context, 1,
H140            (const char **)&zdroj_kod,NULL, &ret);
H141
H142      // zkompiluje (a linkuje) program
H143      ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
H144      cl_kernel kernel; // OpenCL promenna := objekt kernelu
H145      // vytvorime kernel objekt
H146      kernel = clCreateKernel(program, "energy", &ret);
H147
```

```
H148    /* vraci informace o kernel objektu ======== */
H149    clGetKernelInfo(kernel, CL_KERNEL_FUNCTION_NAME, 500, dname, NULL);
H150    clGetKernelInfo(kernel, CL_KERNEL_NUM_ARGS, sizeof(cl_uint), &entries,
        NULL);
H151    printf("Kernel information:\n");
H152    printf(" - %s, %d args\n\n", dname, entries);
H153
H154    //dotaz na maximalni pocet vlaken, v jedne work-groupe
H155    clGetKernelWorkGroupInfo(kernel, device_id, CL_KERNEL_WORK_GROUP_SIZE,
        sizeof(locala), &locala, NULL);
H156    printf("\tPocet vlaken v jedne work-groupe:\t%d\n",locala);
H157
H158    clGetKernelWorkGroupInfo(kernel, device_id, CL_KERNEL_PRIVATE_MEM_SIZE,
        sizeof(long_entries), &long_entries, NULL);
H159    printf("\tmnozstvi privatni pameti:\t%llu\n",long_entries);
H160
H161    // (v bytech) zahrnuje lokalni pamet potrebnou implementaci k vykonani
        kernelu, promenne deklarovane uvnitr jadra, lokalni pamet pridelena pro
        argumenty
H162    // ktere jsou __local  -- pokud neni zadny __local argument, pak
        vracena hodnota je rovna nule
H163    clGetKernelWorkGroupInfo(kernel, device_id, CL_KERNEL_LOCAL_MEM_SIZE,
        sizeof(long_entries), &long_entries, NULL);
H164    printf("\tVraci mnozstvi pouzite lokalni
        pameti:\t%llu\n",long_entries);
H165    /* ==== konec vraci informace o kernel objektu ====== */
H166
H167    // Vytvoříme buffer objekty, nahrajeme do nich matice a priradime do
        memory objektu
H168    cl_mem vstupni_buffer2 =
        clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
        sizeof(cl_float)*maxnum, atomsPole, NULL);
H169    cl_mem vystupni_buffer_sumace =
        clCreateBuffer(context,CL_MEM_READ_WRITE, sizeof(cl_float)*maxnum, NULL,
        NULL);
H170
H171    //pro vykonani kernelu musi byt nastaveny argumenty
H172    ret = clSetKernelArg(kernel, 0, sizeof(cl_uint),&Pvlaken);
H173    ret = clSetKernelArg(kernel, 1, sizeof(cl_uint),&molekul);
H174    ret = clSetKernelArg(kernel, 2,
        sizeof(cl_mem),&vystupni_buffer_sumace);
H175    ret = clSetKernelArg(kernel, 3, sizeof(cl_mem),&vstupni_buffer2);
H176    ret = clSetKernelArg(kernel, 4, sizeof(cl_float),&elko);
H177
H178    // nastavíme global work size - celkovy pocet vlaken a dimenzi
H179    size_t global[1]; // nastaveni dimenze
H180    global[0]= Pvlaken; // nastaveni poctu vlaken
H181
H182    // prikaz na vykonani kernelu na zarizeni
H183    ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
```

```
H184                global, NULL, 0, NULL, &prof_event);
H185
H186      // synchronizacni bod, ceka na dokonceni seznamu udalosti uvedenych ve
druhem argumentu
H187      ret = clWaitForEvents(1, &prof_event);
H188      // synchronizacni bod
H189      // dokud vsechny prikazy command_queue nejsou hotovy, nepokracuje se
dal
H190        clFinish(command_queue);
H191
H192        // sbira informace o prikazu souvisejicim s udalosti
H193        //ziskani casu, kdy program vstoupil do kernelu
H194        clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_START,
H195                sizeof(time_start), &time_start, NULL);
H196        // sbira informace o prikazu souvisejicim s udalosti
H197        //ziskani casu, kdy program vystoupil z kernelu
H198        clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_END,
H199                sizeof(time_konec), &time_konec, NULL);
H200        // vypocet doby behu kernelu
H201        total_time = (time_konec - time_start);
H202
H203        ret = clEnqueueReadBuffer(command_queue, vystupni_buffer_sumace,
CL_TRUE, 0, sizeof(cl_float)*maxnum, sumace, 0, NULL, NULL);
H204
H205        float timeTaken=(float)total_time*1e-9;
H206       printf("Time taken = %.4lf seconds\n", timeTaken);
H207
H208
H209      // vypis vysledku, kterych dosahla jednotliva vlakna
H210      // prvni hodnota je celkovy vysledek energie vody
H211      for(i=0;i<Pvlaken;i++)
H212      {
H213        printf("%lf  ", sumace[i]);
H214      }
H215      printf("\n\n");
H216
H217      // smazani objektu a promennych
H218      free(sumace);
H219      clReleaseMemObject(vystupni_buffer_sumace);
H220      clReleaseProgram(program);
H221      clReleaseKernel(kernel);
H222      clReleaseCommandQueue(command_queue);
H223      clReleaseContext(context);
H224    }
```

A.1.2 Kernel

```
K1  // linearni pole - vypocitani indexu pole ve for cyklu - optimalizace promennych - zkopirovat z globalniho pole cele devitice(molekulu) do privatni pameti
K2  // threads = pocet vlaken = Pvlaken
K3  // Nmolekul = # molekul tj atomy/3
K4  // velkeL = velikost krychle vody
K5  __kernel void energy(uint threads, uint Nmolekul,
K6            __global float *vysledek,
K7            __global float *poleAtomu,
K8            float velkeL
K9           )
K10 {
K11 int i;
K12 // promenne a konstanty slouzici pro vypocet energie vody
K13 float r6,distsq,dist;
K14 float energy=0;
K15 const float sig=0.3166,eps=0.65,eps0=8.85e-12,e=1.602e-19,Na=6.022e23;
K16 float q[3]={-0.8476,0.4238,0.4238};
K17 float rcutsq=1.44;
K18 float elst,sig6;
K19 elst=e*e/(4*3.141593*eps0*1e-9)*Na/1e3,sig6=pow(sig,6);
K20
K21 // ziskani id vlakna
K22 int globalIdx = get_global_id(0);
K23 // vypocet energie vody
K24 // od sveho ID vlakna az do konce
K25 energy = 0;
K26 for (i=globalIdx;i<(Nmolekul); i = i + threads)
K27     {
K28   int j,atomi,atomj,xyz,u;
K29   float ene=0;
K30   float shift[3];
K31   float dr[3];
K32   float mol1[9];
K33 // zkopirovani devitice(molekuly) do privatni pameti(promenne)
K34   for(u=0;u<(9); u++){
K35   mol1[u]= poleAtomu[(i*9)+u];
K36   }
K37
K38
K39
K40   for(j=(i+1);j<Nmolekul;j++)
K41   {
```

```
K42         for(xyz=0;xyz<3;xyz++)
K43         {
K44           dr[xyz] = mol1[xyz] - poleAtomu[(j*9)+xyz];
K45           shift[xyz]=-velkeL* floor(dr[xyz]/velkeL+ .5);
K46           dr[xyz] = dr[xyz] + shift[xyz];
K47         }
K48         distsq= dr[0]*dr[0] + dr[1]*dr[1] + dr[2]*dr[2];
K49         ene = 0;
K50
K51         if(distsq<rcutsq){
K52           r6=sig6/(distsq*distsq*distsq);
K53           ene=4*eps*r6*(r6-1.);
K54
K55           for(atomi=0;atomi<3;atomi++)
K56           {
K57             for(atomj=0;atomj<3;atomj++)
K58             {
K59
K60          for(xyz=0;xyz<3;xyz++)
K61          {
K62
K63
K64             dr[xyz]= mol1[(atomi*3) + (xyz)] - poleAtomu[(j*9)+(atomj*3) +
    (xyz)] + shift[xyz];
K65          }
K66          dist = pow(pow(dr[0],2)+pow(dr[1],2)+pow(dr[2],2),0.5);
K67          ene = ene+elst*q[atomi]*q[atomj]/dist;
K68             }
K69           }
K70         }
K71         energy += ene;
K72       }
K73        }
K74     // kazde vlakno vypoctenou energii priradi do pole
K75     vysledek[globalIdx] = energy;
K76
K77     // synchronizace (v argumentu jsou prikazy OpenCL pro synchronizaci)
K78     barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
K79
K80     // sber vysledku od ostatnich vlaken provede pouze jedno vlakno
K81     if(globalIdx == 0) {
K82
K83     // secteni vysledku od vsech vlaken do vysledek[0]
```

```
K84   for(i=1;i<threads;i++)
K85           {
K86         vysledek[0] += vysledek[i];
K87           }
K88    vysledek[threads] = velkeL;
K89   }
K90   }
```

A.2 OpenMP – Výpočet energie vody

```
1  // watero.c
2  #include <stdio.h>
3  #include <math.h>
4  // includovani OpenMP knihovny
5  #include <omp.h>
6  #include <time.h> // for CPU time
7  #include <sys/time.h> //for gettimeofday
8
9  #define LENGTH 80
10 // global variables
11 const int maxnum=900000;
12 double r[maxnum][3][3],rcutsq=1.44,L;
13 // r(number of molecule, atom 0=O,1=H,2=H, coordinate 0=x,1=y,2=z)
14
15 double sqr(double a){return a*a;}
16
17 double energy12(int i1,int i2){
18   int m,n,xyz;
19   double shift[3],dr[3],mn[3],r6,distsq,dist,ene=0;
20   const double sig=0.3166,eps=0.65,eps0=8.85e-12,
e=1.602e-19,Na=6.022e23,q[3]={-0.8476,0.4238,0.4238};
21   double elst,sig6;
22   elst=e*e/(4*3.141593*eps0*1e-9)*Na/1e3,sig6=pow(sig,6);
23
24   // periodic boundary conditions
25   for(xyz=0;xyz<=2;xyz++){
26    dr[xyz]=r[i1][0][xyz]-r[i2][0][xyz];shift[xyz]=-L*floor(dr[xyz]/L+.5);
//round dr[xyz]/L to nearest integer
27    dr[xyz]=dr[xyz]+shift[xyz];
28   }
29   distsq=sqr(dr[0])+sqr(dr[1])+sqr(dr[2]);
30   if(distsq<rcutsq){ // calculate energy if within cutoff
31     r6=sig6/pow(distsq,3);
32     ene=4*eps*r6*(r6-1.); // LJ energy
```

```
33      for(m=0;m<=2;m++){
34        for(n=0;n<=2;n++){
35          for(xyz=0;xyz<=2;xyz++) mn[xyz]=r[i1][m][xyz]-
r[i2][n][xyz]+shift[xyz];
36          dist=sqrt(sqr(mn[0])+sqr(mn[1])+sqr(mn[2]));
37          ene=ene+elst*q[m]*q[n]/dist;
38      } }
39    }
40    return ene;
41  }
42
43  main(){
44    int i,j,natoms,nmol;
45    double energy=0,dtime;
46    FILE *fp;
47    char line[LENGTH],nothing[LENGTH],name[20];
48    clock_t cputime; /* clock_t defined in <time.h> and <sys/types.h> as int
*/
49    struct timeval start, end;
50    //zadat nazev souboru a otevrit ho
51    printf("Program to calculate energy of water\n");
52    printf("Input NAME of configuration file\n");
53    scanf("%s",name);  fp=fopen(name, "r");
54    fgets(line, LENGTH,fp); //skip first line
55    fgets(line, LENGTH,fp); sscanf(line,"%i",&natoms);
56    nmol=natoms/3; printf("Number of molecules %i\n",nmol);
57
58    cputime = clock();    // assign initial CPU time (IN CPU CLOCKS)
59    gettimeofday(&start, NULL); // returns structure with time in s and us
(microseconds)
60    // cteni souradnic ze souboru do pole r
61    for (i=0;i<nmol;i++){
62      for(j=0;j<=2;j++){
63        fgets(line, LENGTH,fp);
64        sscanf(line, "%s %s %s %lf %lf %lf",nothing,nothing,nothing,
        &r[i][j][0],&r[i][j][1],&r[i][j][2]);
65    } }
66    printf("first line %lf %lf %lf\n",r[0][0][0],r[0][0][1],r[0][0][2]);
67    fscanf(fp, "%lf",&L); // read box size
68    printf("Box size %lf\n",L);
69
70    //paralelizovana cast programu
71    #pragma omp parallel for private (i,j) reduction(+:energy)
72    for(i=0;i<nmol-1;i++){ // calculate energy as sum over all pairs
73      for(j=i+1;j<nmol;j++) energy=energy+energy12(i,j);
```

```
74    }
75
      cputime= clock()-cputime;      // calculate  cpu clock time as difference
76  of times after-before
77    gettimeofday(&end, NULL);
78    dtime = ((end.tv_sec  - start.tv_sec)+(end.tv_usec - start.tv_usec)/1e6);
79
80    printf("Total energy %lf \n ",energy);
81    printf("Energy per molecule %lf \n",energy/nmol);
82    printf("Elapsed wall time: %f\n", dtime);
83    printf("Elapsed CPU  time: %f\n", (float) cputime/CLOCKS_PER_SEC);
84    fclose(fp);
85  }
```

Kód na řádkách 71 – 74, vyznačený žlutě, je v OpenCL vykonávaný v kernelu

A.3 Sériový program – Výpočet energie vody

```
 1  // waters.c
 2  #include <stdio.h>
 3  #include <math.h>
 4  #include <time.h> // for CPU time
 5  #include <sys/time.h> //for gettimeofday
 6
 7  #define LENGTH 80
 8  // global variables
 9  double r[100000][3][3],rcutsq=1.44,L;
10  // r(number of molecule, atom 0=O,1=H,2=H, coordinate 0=x,1=y,2=z)
11
12  double sqr(double a){return a*a;}
13
14  double energy12(int i1,int i2){
15    int m,n,xyz;
16    double shift[3],dr[3],mn[3],r6,distsq,dist,ene=0;
      const double sig=0.3166,eps=0.65,eps0=8.85e-12,e=1.602e-
17  19,Na=6.022e23,q[3]={-0.8476,0.4238,0.4238};
18    double elst,sig6;
19    elst=e*e/(4*3.141593*eps0*1e-9)*Na/1e3,sig6=pow(sig,6);
20
21    // periodic boundary conditions
22    for(xyz=0;xyz<=2;xyz++){
       dr[xyz]=r[i1][0][xyz]-r[i2][0][xyz];shift[xyz]=-L*floor(dr[xyz]/L+.5);
23  //round dr[xyz]/L to nearest integer
24     dr[xyz]=dr[xyz]+shift[xyz];
25    }
26    distsq=sqr(dr[0])+sqr(dr[1])+sqr(dr[2]);
```

```
27    if(distsq<rcutsq){ // calculate energy if within cutoff
28      r6=sig6/pow(distsq,3);
29      ene=4*eps*r6*(r6-1.); // LJ energy
30      for(m=0;m<=2;m++){
31        for(n=0;n<=2;n++){
          for(xyz=0;xyz<=2;xyz++) mn[xyz]=r[i1][m][xyz]-
32  r[i2][n][xyz]+shift[xyz];
33          dist=sqrt(sqr(mn[0])+sqr(mn[1])+sqr(mn[2]));
34          ene=ene+elst*q[m]*q[n]/dist;
35      } }
36    }
37    return ene;
38  }
39
40  main(){
41    int i,j,natoms,nmol;
42    double energy=0,dtime;
43    FILE *fp;
44    char line[LENGTH],nothing[LENGTH],name[20];
      clock_t cputime; /* clock_t defined in <time.h> and <sys/types.h> as int
45  */
46  //  struct timeval start, end;
47
48    //otevri soubor
49    printf("Program to calculate energy of water\n");
50    printf("Input NAME of configuration file\n");
51    scanf("%s",name);  fp=fopen(name, "r");
52    fgets(line, LENGTH,fp); //skip first line
53    fgets(line, LENGTH,fp); sscanf(line,"%i",&natoms);
54    nmol=natoms/3; printf("Number of molecules %i\n",nmol);
55
56    cputime = clock();    // assign initial CPU time (IN CPU CLOCKS)
    //  gettimeofday(&start, NULL); // returns structure with time in s and us
57  (microseconds)
58
59    //steni ze souboru
60    for (i=0;i<nmol;i++){
61      for(j=0;j<=2;j++){
62        fgets(line, LENGTH,fp);
          sscanf(line, "%s %s %s %lf %lf %lf",nothing,nothing,nothing,
63        &r[i][j][0],&r[i][j][1],&r[i][j][2]);
64    } }
65    printf("first line %lf %lf %lf\n",r[0][0][0],r[0][0][1],r[0][0][2]);
66    fscanf(fp, "%lf",&L); // read box size
67    printf("Box size %lf\n",L);
```

```
68
69    for(i=0;i<nmol-1;i++){ // calculate energy as sum over all pairs
70      for(j=i+1;j<nmol;j++) energy=energy+energy12(i,j);
71    }
72
73    cputime= clock()-cputime;      // calculate  cpu clock time as difference
of times after-before
74  //  gettimeofday(&end, NULL);
75  //  dtime = ((end.tv_sec  - start.tv_sec)+(end.tv_usec -
start.tv_usec)/1e6);
76
77    printf("Total energy %lf \n ",energy);
78    printf("Energy per molecule %lf \n",energy/nmol);
79    printf("Elapsed wall time: %f\n", dtime);
80    printf("Elapsed CPU  time: %f\n", (float) cputime/CLOCKS_PER_SEC);
81    fclose(fp);
82  }
```

A.4 OpenCL – Matematické funkce

A.4.1 Host

```
H1   //Matematicke funkce
H2   #include <stdio.h>
H3   #include <stdlib.h>
H4   #include <math.h>
H5   // potreba includovat knihovnu OpenCL CL/cl.h
H6   #include "CL/cl.h"
H7   //definujeme pocet vlaken
H8   #define Pvlaken 10000
H9   const int maxnum=900004; // maximalni potrebna delka pole
H10  const int LENGTH = 80;
H11  float L;
H12  int main ()
H13  {
H14    float atomsPole[maxnum];
H15    cl_float *sumace;
H16    cl_uint vlaken = Pvlaken;
H17    cl_ulong time_start, time_konec;
H18    long total_time;
H19
H20    // nacteni zdrojoveho kodu kernelu do stringove promenne =======
H21    FILE *fp;
H22    char *zdroj_kod;
```

```
H23     long delka_souboru;
H24     long delka_cteni;
H25
H26     fp = fopen("matoperace.cl","r");
H27      fseek(fp,0L, SEEK_END);
H28      delka_souboru = ftell(fp);
H29      rewind(fp);
H30      zdroj_kod = malloc(sizeof(char)*(delka_souboru+1));
H31      delka_cteni = fread(zdroj_kod,1,delka_souboru,fp);
H32      if(delka_cteni!= delka_souboru)
H33      {
H34         printf("Nelze přečist soubor\n");
H35         exit(1);
H36      }
H37     zdroj_kod[delka_souboru+1]='\0';
H38     fclose(fp);
H39     // nacteni zdrojoveho kodu kernelu do stringove promenne =======
H40
H41     // alokace promenne
H42     sumace = malloc(sizeof(cl_float)*maxnum);
H43
H44     //================cteni souboru s molekuly vody====================
H45     int i,j;
H46     for (i=0;i<100000;i++){
H47         atomsPole[i] = rand() % 30 + 1;;
H48       }
H49     //================cteni souboru s molekuly vody====================
H50
H51     //inicializace pole (ktere nakonec vrati z kernelu vysledky)
H52     for(i=0;i<maxnum;i++)
H53       {
H54      sumace[i]= 0; //data
H55       }
H56
H57     //opencl promenne potrebne pro obsluhu GPU
H58     cl_platform_id platform_id = NULL;
H59     cl_device_id device_id = NULL;
H60     cl_uint ret_num_devices;
H61     cl_uint ret_num_platforms;
H62     cl_event prof_event;
H63
H64     // vraci seznam dostupnych platforem
H65     cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
```

```
H66     // vraci seznam dostupnych zarizeni na platforme
H67     ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_ALL, 1,
H68               &device_id, &ret_num_devices);
H69
H70     // vytvorime OpenCL context. Vytvoren s jednim ci vice zarizenimi. Je
        pouzit pri behu programu pro
H71     // rizeni OpenCL objektu jako jsou command_queue, memory, program a
        kernel.
H72     // A pro vykonavani vice kernelu na jednom ci vice zarizenich
        specifikovanych v contextu
H73     cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL,
        &ret); // 1 je pocet zarizeni; &device_id ukazatel na seznam zarizeni
H74
H75     // vytvorime command queue na specifickym zarizeni
H76     //Operace nad OpenCL objekty jsou vykonavany pouzitim toho prikazu
H77     cl_command_queue command_queue = clCreateCommandQueue(context,
        device_id, CL_QUEUE_PROFILING_ENABLE, &ret);
H78
H79     // nahrajeme zdrojový kod opencl kernelu - vytvori program objekt pro
        kontext
H80     cl_program program = clCreateProgramWithSource(context, 1,
H81           (const char **)&zdroj_kod,NULL, &ret);
H82
H83     // zkompiluje (a linkuje) program
H84     ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
H85     cl_kernel kernel; // OpenCL promenna
H86     // vytvorime kernel objekt
H87     kernel = clCreateKernel(program, "matoperace", &ret);
H88
H89     // Vytvoříme buffer objekty, nahrajeme do nich matice a priradime do
        memory objektu
H90     cl_mem vstupni_buffer2 =
        clCreateBuffer(context,CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
        sizeof(cl_float)*maxnum, atomsPole, NULL);
H91     cl_mem vystupni_buffer_sumace =
        clCreateBuffer(context,CL_MEM_READ_WRITE, sizeof(cl_float)*maxnum, NULL,
        NULL);
H92
H93     //pro vykonani kernelu musi byt nastaveny argumenty
H94     ret = clSetKernelArg(kernel, 0, sizeof(cl_uint),&vlaken);
H95     ret = clSetKernelArg(kernel, 1, sizeof(cl_int),&maxnum);
H96     ret = clSetKernelArg(kernel, 2, sizeof(cl_mem),&vystupni_buffer_sumace);
H97     ret = clSetKernelArg(kernel, 3, sizeof(cl_mem),&vstupni_buffer2);
H98
H99     // nastavíme global work size - celkovy pocet vlaken a dimenzi
H100    size_t global[1]; // nastaveni dimenze
H101    global[0]= Pvlaken; // nastaveni poctu vlaken
H102
H103    // prikaz na vykonani kernelu na zarizeni
```

```
H104    ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
H105              global, NULL, 0, NULL, &prof_event);
H106
H107    // synchronizacni bod, ceka na dokonceni seznamu udalosti uvedenych ve druhem argumentu
H108    ret = clWaitForEvents(1, &prof_event);
H109    // synchronizacni bod
H110    // dokud vsechny prikazy command_queue nejsou hotovy, nepokracuje se dal
H111      clFinish(command_queue);
H112
H113      // sbira informace o prikazu souvisejicim s udalosti
H114      //ziskani casu, kdy program vstoupil do kernelu
H115      clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_START,
H116              sizeof(time_start), &time_start, NULL);
H117      // sbira informace o prikazu souvisejicim s udalosti
H118      //ziskani casu, kdy program vystoupil z kernelu
H119      clGetEventProfilingInfo(prof_event, CL_PROFILING_COMMAND_END,
H120              sizeof(time_konec), &time_konec, NULL);
H121      // vypocet doby behu kernelu
H122      total_time = (time_konec - time_start);
H123      // cteni kopirovani vysleku z kernelu do host pameti
H124      ret = clEnqueueReadBuffer(command_queue, vystupni_buffer_sumace, CL_TRUE, 0, sizeof(cl_float)*maxnum, sumace, 0, NULL, NULL);
H125
H126      float timeTaken=(float)total_time*1e-9;
H127     printf("Time taken = %.4lf seconds\n", timeTaken);
H128
H129    // vypis vysledku, kterych dosahla jednotliva vlakna
H130    // prvni hodnota je celkovy vysledek energie vody
H131    for(i=0;i<Pvlaken;i++)
H132    //for(i=0;i<1;i++)
H133    {
H134      printf("%lf  ", sumace[i]);
H135    }
H136    printf("\n\n");
H137
H138    // smazani objektu a promennych
H139    free(sumace);
H140    clReleaseMemObject(vystupni_buffer_sumace);
H141    clReleaseProgram(program);
H142    clReleaseKernel(kernel);
H143    clReleaseCommandQueue(command_queue);
H144    clReleaseContext(context);
H145  }
```

A.4.2 Kernel

```
H1    __kernel void matoperace(uint pocetvlaken, uint velikostPole,
H2             __global float *sumFci,
H3             __global float *poleAtomu
H4            )
H5    {
H6    int i;
H7    float r6,distsq,dist;
H8    // identikator vlakna (uid)
H9      int globalIdx = get_global_id(0);
H10
H11   int j;
H12    for (j=0; j<9999;j++){
H13    for (i=globalIdx;i<(velikostPole); i = i + pocetvlaken)
H14      {
H15         // sumFci[i]= pow(poleAtomu[i],3);
H16         // sumFci[i]= poleAtomu[i] * poleAtomu[i];
H17         // sumFci[i]= sin(poleAtomu[i]);
H18          sumFci[i]= log(poleAtomu[i]);
H19       }
H20    }
H21   barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
H22   barrier(CLK_GLOBAL_MEM_FENCE);
H23   }
```

A.5 OpenMP – Matematické funkce

```
 1  // watero.c
 2  #include <stdio.h>
 3  #include <math.h>
 4  #include <omp.h>
 5  #include <time.h> // for CPU time
 6  #include <sys/time.h> //for gettimeofday
 7
 8  #define LENGTH 80
 9  // global variables
10  const int maxnum=100000;
11  double r[maxnum],rcutsq=1.44,L;
12
13  main(){
14    int i,j,natoms,nmol;
```

```
15    double energy=0,dtime;
16    FILE *fp;
17    char line[LENGTH],nothing[LENGTH],name[20];
18    clock_t cputime; /* clock_t defined in <time.h> and <sys/types.h> as int
*/
19    struct timeval start, end;
20    // nacteni nahodnych hodnot od 1 od 30 do pole r
21    for (i=0;i<maxnum;i++){
22      r[i] = rand() % 30 + 1;
23      }
24      cputime = clock();    // assign initial CPU time (IN CPU CLOCKS)
25    gettimeofday(&start, NULL); // returns structure with time in s and us
(microseconds)
26  for (j=0; j< 9999;j++){
27    #pragma omp parallel for private (i)
28    for(i=0;i<maxnum;i++){
29       // r[i]=sin(r[i]);
30       // r[i]=pow(r[i],2);
31       // r[i]=r[i]*r[i];
32       r[i]=log(r[i]);
33    }
34  }
35    cputime= clock()-cputime;      // calculate  cpu clock time as difference
of times after-before
36    gettimeofday(&end, NULL);
37    dtime = ((end.tv_sec  - start.tv_sec)+(end.tv_usec - start.tv_usec)/1e6);
38
39    printf("Total energy %lf \n ",energy);
40  //  printf("Energy per molecule %lf \n",energy/nmol);
41    printf("Elapsed wall time: %f\n", dtime);
42    printf("Elapsed CPU  time: %f\n", (float) cputime/CLOCKS_PER_SEC);
43  }
```

B Obsah CD

CD obsahuje bakalářskou práci ve formátu pdf a docx. Dále obsahuje tři složky Seriove, OMP a OpenCL.

Složky:

- Seriove – obsahuje sériový program energie vody.
- OMP – obsahuje OpenMP paralelní programy matematické funkce a energie vody.
- OpenCL – obsahuje paralelní OpenCL programy matematické funkce a energie vody.