VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

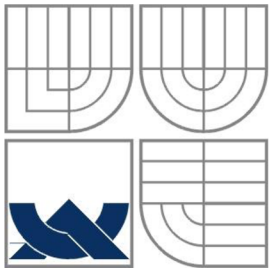
PARALELIZACE ULTRAZVUKOVÝCH SIMULACÍ POMOCÍ AKCELERÁTORU INTEL XEON PHI

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

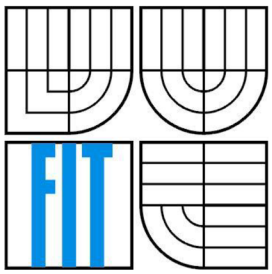
AUTOR PRÁCE
AUTHOR

Bc. ANDREJ VRBENSKÝ

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PARALELIZACE ULTRAZVUKOVÝCH SIMULACÍ POMOCÍ AKCELERÁTORU INTEL XEON PHI

PARALLELISATION OF ULTRASOUND SIMULATIONS ON INTEL XEON PHI ACCELERATOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ANDREJ VRBENSKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ JAROŠ, Ph.D.

BRNO 2015

Abstrakt

Simulácia šírenia ultrazvukových akustických vĺn má v dnešnej dobe široké praktické použitie. Jedným z nich je simulácia v reálnom tkanivovom prostredí, ktorá má dobré uplatnenie v medicíne. Jednou z aplikácií, ktoré sú na túto simuláciu určené, je k-Wave toolbox. Výpočtová náročnosť takýchto simulácii je veľmi veľká a preto sa vyvíjajú nové metódy pre jej zrýchlenie. V tejto diplomovej práci sme navrhli riešenie pre urýchlenie simulácie, založené na paralelizácii výpočtu na akceleračnej karte Intel Xeon Phi. Akcelerátor obsahuje vysoký počet jadier a extra-širokú vektorovú jednotku, a je preto ideálny na paralelizáciu a vektorizáciu. Implementácia využíva OpenMP verzie 4.0, ktorá prináša niektoré nové možnosti ako napríklad explicitnú vektorizáciu. Dosiahnuté výsledky boli namerané počas rozsiahlych experimentov.

Abstract

Nowadays, the simulation of ultrasound acoustic waves has a wide range of practical usage. As one of them we can name the simulation in realistic tissue media, which is successfully used in medicine. There are several software applications dedicated to perform such simulations. k-Wave is one of them. The computational difficulty of the simulation itself is very high, and this leaves a space to explore new speed-up methods. In this master's thesis, we proposed a way to speed-up the simulation based on parallelization using Intel Xeon Phi accelerator. The accelerator contains large amount of cores and an extra-wide vector unit, and therefore, is ideal for purpose of parallelization and vectorization. The implementation is using OpenMP version 4.0, which brings some new options such as explicit vectorization. Results were measured during extensive experiments.

Klíčová slova

koprocesor, Intel Xeon Phi, k-Wave, OpenMP, paralelizmus, simulace, ultrazvuk, vektorizace, FFT

Keywords

coprocessor, Intel Xeon Phi, k-Wave, OpenMP, parallelism, simulation, ultrasound, vectorization, FFT

Citace

Andrej Vrbenský: Paralelizace *ultrazvukových simulací pomocí akcelerátoru Intel Xeon Phi*, diplomová práce, Brno, FIT VUT v Brně, 2015

Paralelizace ultrazvukových simulací pomocí akcelerátoru Intel Xeon Phi

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jiřího Jaroše, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Andrej Vrbenský
20.5.2015

Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Jiřimu Jarošovi, PhD., za rady, odbornou pomoc a trpělivost při řešení této diplomové práce.

Tato práce byla podpořena projektem IT4Innovations Centre of Excellence (CZ.1.05/1.1.00/02.0070), financovaným z Evrópskeho fondu regionálního rozvoje a státního rozpočtu České Republiky přes Operační program Výskum a vývoj pro inovace, a Ministerstvo školství, mládeže a tělovýchovy přes projekty Velkých infrastruktur pro výzkum, vývoj a inovace (LM2011033).

Obsah

1 Úvod

Reálne použitie ultrazvuku je všeobecne známe. Jeho použitie ako diagnostického nástroja má v dnešnej medicíne dôležitú úlohu. Do popredia sa ale čoraz viac dostáva aj takzvaný vysokointenzívne zameraný ultrazvuk (high-intensity focused ultrasound, skrátene HIFU). Je to metóda, ktorá funguje na princípe ničenia nechceného chorobného tkaniva, jeho zahriatím na vysokú teplotu pomocou ultrazvuku. Metóda je neinvazívna, čo znamená, že ultrazvukové vlny sú aplikované na tkanivo z vonku, a teda ultrazvuk v určitom množstve vždy zasiahne aj zdravé tkanivo. A tu sa veľmi dobre uplatňujú systémy, ktoré umožňujú simulovanie šírenia ultrazvukových vĺn. S ich použitím je možné zákrok nasimulovať na reálnych dátach a naplánovať priebeh tohoto zákroku tak, aby bolo efektívne ničené chorobné tkanivo a naopak zdravé tkanivo zostalo čo najmenej ovplyvnené a výrazne tak obmedziť nepriaznivé účinky. Samotné simulovanie má vysoké nároky na výpočtový výkon. Napríklad, typický diagnostický ultrazvukový obraz vytvorený $3\,MHz$ konvexnou sondou má penetračnú hĺbku okolo $15\,cm$. Vzdialenosť je rádovo 300 akustických vlnových dĺžok na základnej frekvencii a 600 vlnových dĺžok na druhej harmonickej. Tradičné numerické metódy ako konečné diferencie alebo konečné prvkové metódy vyžadujú rádovo desiatky bodov na vlnovú dĺžku, aby boli schopné dosiahnuť požadovanú presnosť. To sa rovná výpočetnej doméne o tisíckach bodov v každej priestorovej dimenzii. Aby sa znížila výpočtová náročnosť, pomerne často sa používajú zjednodušujúce predpoklady. Avšak vtedy zanedbávame niektoré aspekty nelineárneho šírenia vĺn v heterogénnon prostredí [1]. Toto sú úpravy samotného fyzikálneho modelu. Ďalšou možnosťou je tieto modely zrýchliť ich lepšou technickou implementáciou. Jednou z takýchto možností je paralelizácia výpočtu.

1.1 Cieľ práce

Táto diplomová práca je zameraná na paralelizáciu existujúceho nástroja na spracovanie ultrazvukových simulácií k-Wave toolbox [2] na akceleračnej karte Intel Xeon Phi. Táto práca si kladie za cieľ priniesť alternatívne riešenie implementácie aplikácie k-Wave toolbox. Budeme sa snažiť priniesť také riešenie, ktoré maximálne využije potenciál jednej karty Intel Xeon Phi. To potom môže pomôct aspoň priblížiť predpokladaný výkon, ktorý bude dosiahnuteľný s využitím viacerých týchto kariet – či už viac kariet na jednom výpočtovom stroji alebo ešte väčšieho množstva kariet na výpočetnom clustery. Korektnosť implementácie bude experimentálne overená oproti existujúcej plne funkčnej verzii aplikácie k-Wave toolbox implementovanej pre bežné procesory. Výsledný výkon bude porovnávaný s implementáciou bežiacou na viacjadrovom procesore, pričom pri experimentoch použijeme reálne dáta.

1.2 Obsah práce

Informácii o softvéri k-Wave toolbox a jeho implementácii obsahuje nasledujúca kapitola číslo 2. V kapitole číslo 3 je popísaná karta Intel Xeon Phi spolu s možnosťami jej použitia a je v nej popísané aj programovanie pomocou OpenMP, ktoré je neodmysliteľnou súčasťou programovania so zdielanou pamäťou na karte Intel Xeon Phi. V kapitole číslo 3.4 je navrhnutý postup riešenia, podľa ktorého dospejeme ku konečnej podobe diplomovej práce. Implementácia je popísaná v kapitole číslo 5. Jedna kapitola (číslo 6) je venovaná experimentom a rôznym meraniam implementácie. Zhrnutie aktuálneho stavu projektu a rovnako aj jeho budúceho vývoja sa nachádza v záverečnej kapitole číslo 7.

2 Simulačné prostredie k-Wave

Softvér k-Wave toolbox je akustický súbor nástrojov pre prostredia MATLAB a C++, vyvinutý autormi Bradley Treeby, Ben Cox (University College London) a Jiří Jaroš (Brno University of Technology). k-Wave je open-source softvér navrhnutý pre akustické a ultrazvukové simulácie v časovej doméne prebiehajúce v komplexných a reálnych tkanivových prostrediach. Simulačné funkcie sú založené na k-priestorovej pseudospektrálnej metóde (k-space pseudospectral method) a sú súčasne rýchle a jednoduché na použitie. k-Wave toolbox ponúka [2]:

- Pokročilé modelovanie šírenia akustických vĺn v časovej doméne, ktoré dokáže počítať s nelinearitou, akustickými heterogenitami a absorbciou v *1D*, *2D* a *3D*.
- Možnosť modelovania zdrojov tlaku a rýchlosti, zahrňujúc fotoakustické zdroje a terapeutické ultrazvukové sondy.
- Možnosť špecifikovať ľubovolné detekujúce povrchy so smerovými prvkami a s možnosťou záznamu akustického tlaku, rýchlosti častice a akustickej intenzity.
- Optimalizovanú C++ verziu kódu, ktorá maximalizuje výpočtový výkon pre rozsiahle simulácie.
- Možnosť použiť dopredný model ako flexibilne časovo reverzný obrazový rekonštrukčný algoritmus pre fotoakustickú tomografiu s ľubovolným meraným povrchom.
- Rýchly jednokrokový fotoakustický obrazový rekonštrukčný algoritmus pre dáta zaznamenané na lineárnom (*2D*) alebo planárnom (*3D*) meranom povrchu.
- Voliteľné vstupné parametre pre nastavenie vizualizácie a výkonu, zahrňujúce možnosti pre generovanie videí a spustenie simulácií na grafickom procesore (GPU).
- Rozsiahly používateľský manuál a mnoho dobre vysvetľujúcich návodov a príkladov, ktoré ilustrujú možnosti tohto nástroja.

2.1 Numerický model

Ako bolo napísané, k-Wave má široký rozsah funkcionality, ale v jeho jadre je to pokročilý numerický model, ktorý dokáže počítať s obidvoma, lineárne aj nelineárne šíriacimi sa akustickými vlnami, ľubovolným rozložením parametrov heterogénneho materiálu a jeho schopnosti absorbovať časť akustického žiarenia. Numerický model je založený na riešení troch párov parciálne diferenciálnych rovníc prvého rádu, ktoré sú ekvivalentom k zovšeobecnenej forme Westerveltovej rovnice [2].

Keď akustická vlna prechádza cez stlačiteľné prostredie, objavujú sa dynamické fluktuácie v tlaku, hustote, teplote, rýchlosti častíc, atď. Tieto zmeny môžu byť popísané radou prepojených

parciálne diferenciálnych rovníc prvého rádu založených na zachovaní hmoty, momentu a energie vo vnútri prostredia (conservation laws). V akustike sú často tieto rovnice skombinované do jednej „vlnovej rovnice“, ktorá je vlastne parciálne diferenciálna rovnica druhého rádu na jednej akustickej premennej (najčastejšie to je akustický tlak). Napríklad v klasickom základnom prípade, keď sa akustická vlna s malou amplitúdou šíri cez homogénne a bezstratové prostredie, sú rovnice prvého rádu dané nasledovne [4]:

$$\frac{\partial u}{\partial t} = -\frac{1}{\rho_0}\nabla p \tag{2.1}$$

Zachovanie hybnosti

$$\frac{\partial \rho}{\partial t} = -\rho_0 \nabla \mathrm{u} \tag{2.2}$$

Zachovanie hmoty

$$p = c_0^2 \rho \tag{2.3}$$

Vzťah tlak-hustota

Tu použité **u** je rýchlosť akustickej častice, **p** je akustický tlak, **ρ** je akustická hustota, $\boldsymbol{\rho_0}$ je hustota okolia (alebo ekvilibria) a $\mathbf{c_0}$ je isentropická (nemeniaca sa v čase) rýchlosť zvuku. Tieto rovnice sú založené na predpoklade, že prostredie v okolí je neaktívne (čo znamená, že neexistuje pohyb na základe interakcie okolia a vnútra uzatvoreného materiálu, a ostatné parametre okolia sa nemenia v čase) a isotropické (čo znamená, že parametre materiálu sú nezávislé na smere šírenia vlny). Keď sú tieto tri rovnice skombinované dohromady, vytvoria tak vlnovú rovnicu druhého rádu:

$$\nabla^2 p - \frac{1}{c_0^2}\frac{\partial^2 p}{\partial t^2} = 0 \tag{2.4}$$

K-Wave toolbox je založený na riešení sústavy rovníc prvého rádu a nie na rovnici druhého rádu. Tento prístup totiž umožňuje jednoduchšie použitie ďalších zjednodušovacích techník a pridanie nelineárnych parametrov.

Akustické prostredie je však často heterogénne s priestorovo sa meniacou rýchlosťou zvuku a okolnou hustotou. V takomto prípade musia vyššie uvedené rovnice zahŕňať aj ďalšie členy. Podobne, ako keď sa akustická vlna šíri, tak stráca časť svojej akustickej energie v prospech náhodného tepelného pohybu, čo ústi v akustickú absorpciu. Aj táto musí byť reflektovaná v rovniciach. V mnohých situáciách v biomedicínskom použití ultrazvuku je použitá sila akustických vĺn dostatočne veľká na to, že šírenie vlny už nie je viac lineárne. V tomto prípade musia horeuvedené rovnice opäť zahŕňať aj ďalšie nelineárne členy. k-Wave toolbox nemodeluje úplne všetky možné nelineárne členy, pretože to nie je softvér, ktorý rieši počítanie dynamiky tekutín

(computational fluid dynamics, skrátene CFD). Namiesto toho momentálne zahŕňa dva ďalšie nelineárne členy, ktoré reprezentujú kumulatívne nelineárne vplyvy na druhý rád akustických premenných. Toto je model, ktorý je presný v mnohých použitiach biomedicínskeho ultrazvuku. Keď používateľ definuje v k-Wave parameter *medium.BonA*, systém horeuvedených počítaných rovníc sa zmení. Tieto rovnice už zahrnujú aj úpravy potrebné pre započítanie nelineárnych parametrov akustickej absorbcie a heterogenity [1][4].

$$\frac{\partial u}{\partial t} = -\frac{1}{\rho_0}\nabla p \tag{2.5}$$

Zachovanie hybnosti

$$\frac{\partial \rho}{\partial t} = -(2\rho + \rho_0)\nabla \cdot u - u \cdot \nabla \rho_0 \tag{2.6}$$

Zachovanie hmoty

$$p = c_0^2(\rho + d \cdot \nabla \rho_0 + \frac{B}{2A}\frac{\rho^2}{\rho_0} - L\rho) \tag{2.7}$$

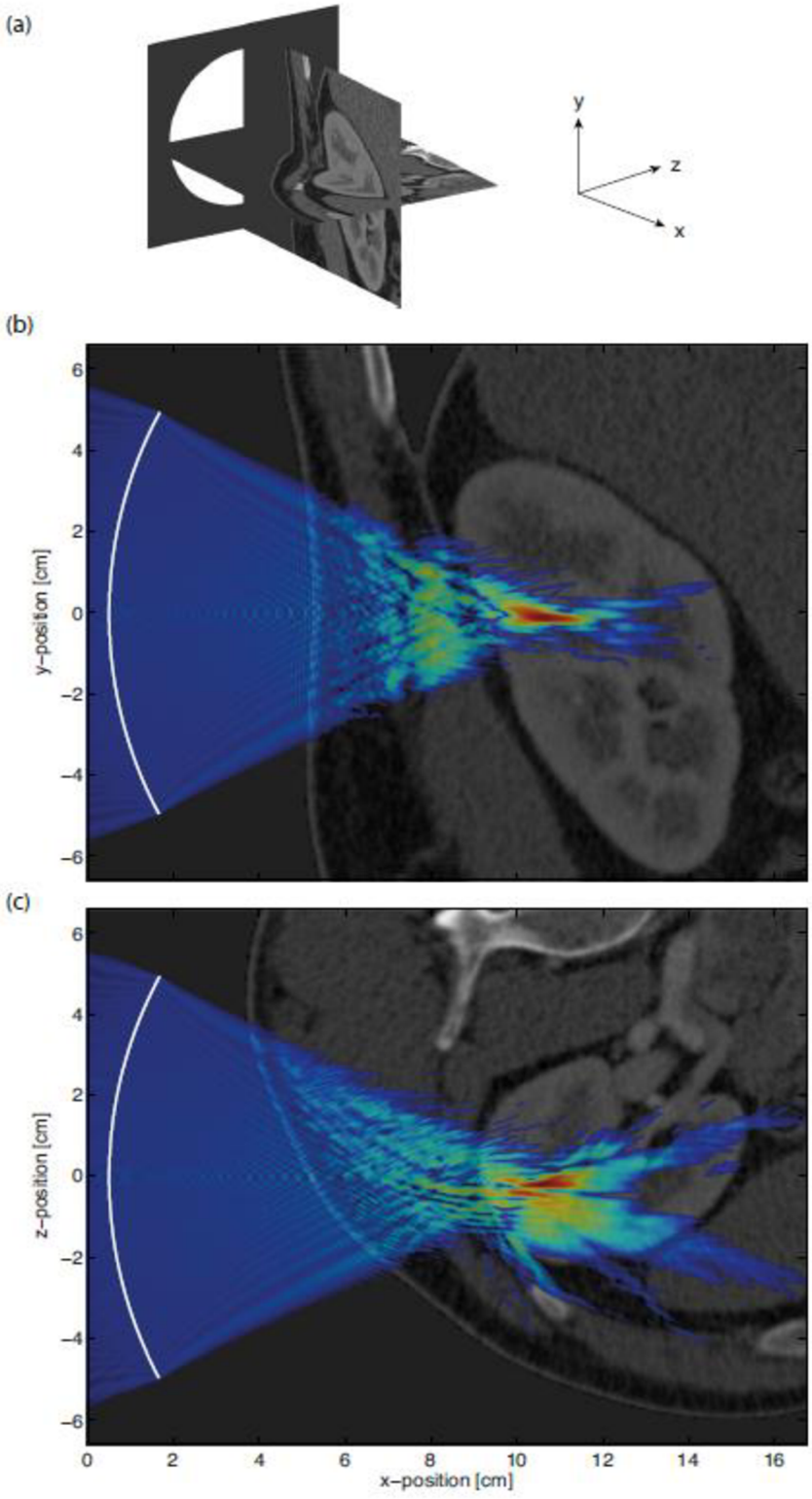
Vzťah tlak-hustota

Parametre sú rovnaké ako v pôvodných rovniciach, a naviac sú definované dva ďalšie. Parameter **B/A** reprezentuje nelinearitu, ktorú charakterizuje relatívny príspevok konečne-amplitúdového vplyvu na rýchlosť zvuku [4]. Parameter **L** reprezentuje lineárny integračne-diferenčný operátor, ktorý reprezentuje akustickú absorpciu a disperziu. Prítomnosť absorpcie musí byť fyzicky sprevádzaná disperziou (závislosť rýchlosti zvuku na frekvencii) pre zachovanie kauzality. Operátor použitý v k-Wave má dva členy závislé na Laplacovej transformácii [1][4][5].

$$L = \tau \frac{\partial}{\partial t}(-\nabla^2)^{\frac{y}{2}-1} + \eta(-\nabla^2)^{\frac{y+1}{2}-1} \tag{2.8}$$

Tieto rovnice sú riešené pomocou k-priestorovej pseudospektrálnej metódy, kde sú pomocou Fourierovej kolokačnej metódy (Fourier collocation scheme) vypočítané priestorové gradienty a pomocou k-priestorovej upravenej konečne-diferenčnej metódy (k-space corrected finite-difference scheme) vypočítané časové gradienty. Časová schéma je presne na hranici lineárne sa šíriacej vlny v homogénnom bezstratovom prostredí a je tak významne znížený numerický rozptyl v zovšeobecnenom prípade. PML (split-field perfectly matched layer) je použitý pre absorbovanie akustickej vlny na okraji vypočetného priestoru. Hlavnou výhodou numerického modelu použitého v k-Wave toolbox oproti modelom založeným na konečno-diferenčných časovo doménových metódach (finite-difference time domain schemes – FDTD) je, že množstvo priestorových a časových bodov potrebných pre presnú simuláciu je menšie. To znamená, že model beží rýchlejšie a spotrebuje menej pamäti [2].

Na nasledujúcom obrázku 2.1 je možné vidieť jeden príklad použitia simulátora ultrazvuku, a to konkrétne plánovanie HIFU zákroku na obličke pacienta.



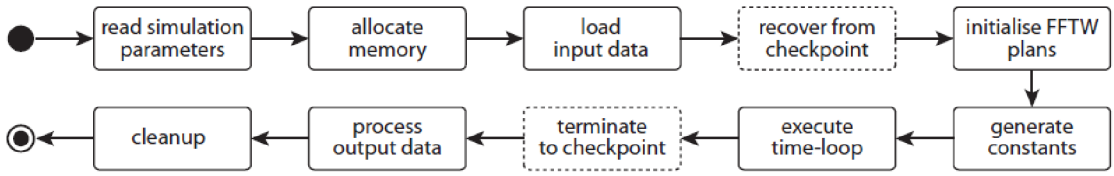
Obrázok 2.1 CT obraz brucha, použitého na simuláciu HIFU liečby obličky [6].

2.2 Súčasné implementácie

V súčasnej dobe existuje niekoľko rôznych implementácií, kde každá využíva rôzne technológie. Úplne prvou implementovanou variantou bola aplikácia napísaná pre prostredie MATLAB. MATLAB je známy svojím zameraním na matematické výpočty a simulácie, a aj preto sa pravdepodobne javil ako dobrý spôsob. Ale samozrejme, keďže ultrazvukové simulácie sú výpočetne veľmi náročné, tak táto verzia nie je vždy výhodná a ani dostatočne použiteľná. Simulácie o väčších rozmeroch prostredia, tak budú bežať veľmi dlho. Rovnako pri experimentovaní a s každou zmenou parametrov, by bolo potrebné pri každej opakovanej simulácii znova čakať veľmi dlhý čas.

Preto bola vytvorená verzia napísaná v programovacom jazyku C++. Ten je známy svojou dobrou výkonnosťou, a preto si ho ľudia vyberajú pre aplikácie, kde je výkon jedným s najdôležitejších parametrov. Dostupné kompilátory sú veľmi dobre optimalizované a dokážu využívať špeciálne a multimediálne inštrukcie procesorov (SSE, MMX, AVX,...). Ďalším pozitívom je veľmi dobrá podpora paralelného programovania – samozrejmá je podpora programovania so zdielanou pamäťou (použitie OpenMP) a aj s distribuovanou pamäťou (použite MPI). Oproti verzii so sdielanou pamäťou budeme porovnávať správnu funkcionalitu našej výslednej verzie aplikácie v rámci diplomovej práce. Výsledky sa budú s veľkou pravdepodobnosťou mierne líšiť, keďže pri výpočtoch modelu budú použité iné technológie (iná architechtúra karty Xeon Phi). Aktuálne existujú tri verzie aplikácie v C++ a to práve:

1) S využitím zdielanej pamäti
2) S využitím distribuovanej pamäti na výpočetnom clustery
3) S využitím GPU (vo vývoji)



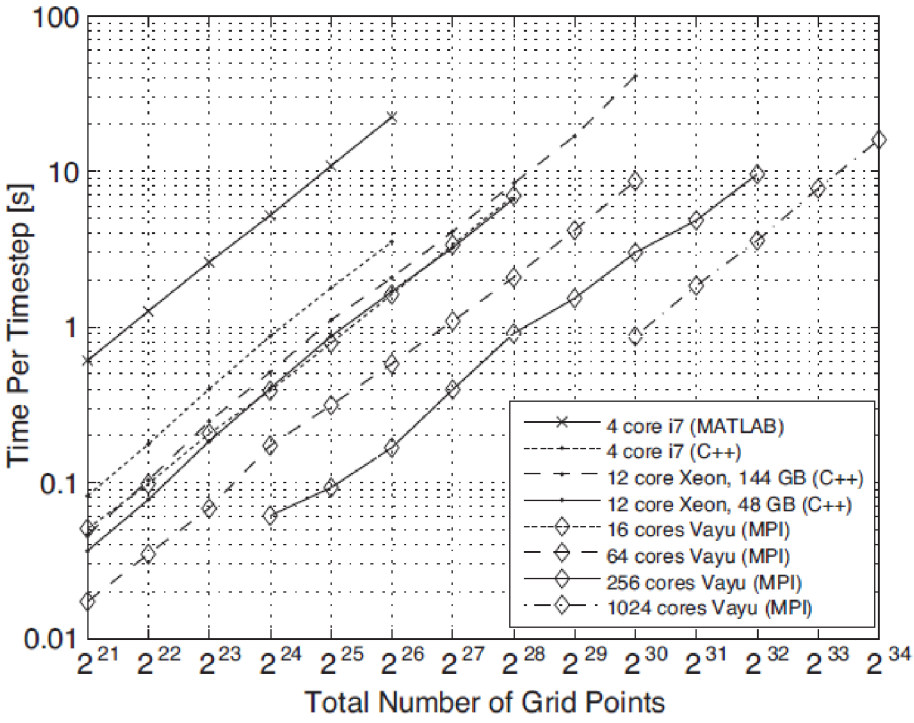
Obrázok 2.2 Algoritmus simulácie [6].

Obrázok 2.2 zobrazuje simulačný algoritmus, ktorý je implementovaný vo všetkých verziách avšak vždy s inou technickou realizáciou. Nasleduje podrobnejší popis existujúcich verzií.

Verzia s využitím zdielanej pamäti

Je určená pre beh na viacjadrových, či viacvláknových procesoroch. Je založená na Matlab verzii, ale obsahuje množstvo vylepšení. Paralelizmus na úrovni vlákien je implementovaný použitím OpenMP. Paralelizmus na úrovni inštrukcií zas použitím SIMD inštrukcií Intel SSE v ich intrinsic variante. Boli vykonané aj ďalšie optimalizácie ako napríklad počítanie *3D* FFT (Fast Fourier Transform). To bolo pôvodne riešené pomocou complex-to-complex FFT metódy. Optimalizovaná verzia používa real-to-complex metódu, kde zrýchlenie predstavuje okolo *50 %*. Výpočty sú prevádzané v jednoduchej presnosti, aby sa ušetrilo množstvo použitej pamäti [1]. Výkonnostné porovnanie je vidieť na obrázku 2.3.

Táto verzia bude v slúžiť v našej diplomovej práci ako základná verzia (baseline). Je to z toho dôvodu, že budeme používať natívny programovací model pre Intel Xeon Phi, ktorý rovnako umožňuje použitie OpenMP. Niektoré (možno väčšinu) časti implementácie teda bude veľmi pravepodobne možné znovu použiť.



Obrázok 2.3 Porovnanie výkonnosti jednotlivých implementácií (mimo GPU verzie) [6].

Verzia s využitím distribuovanej pamäti na výpočetnom clustery

Táto verzia ako jediná nie je obmedzená len na jeden výpočtový uzol. Jej sila je ukrytá v škálovateľnosti cez veľké množstvo uzlov – veľkosť simulačných dát je limitovaná len veľkosťou clusteru (jeho celkovou pamäťou). Základom je použitie MPI (message-passing interface), ktorý umožňuje komunikáciu medzi jednotlivými procesormi/uzlami v systéme/clusteri. Porovnanie výkonu je možné vidieť na obrázku 2.3.

2.3 Obsah repozitára

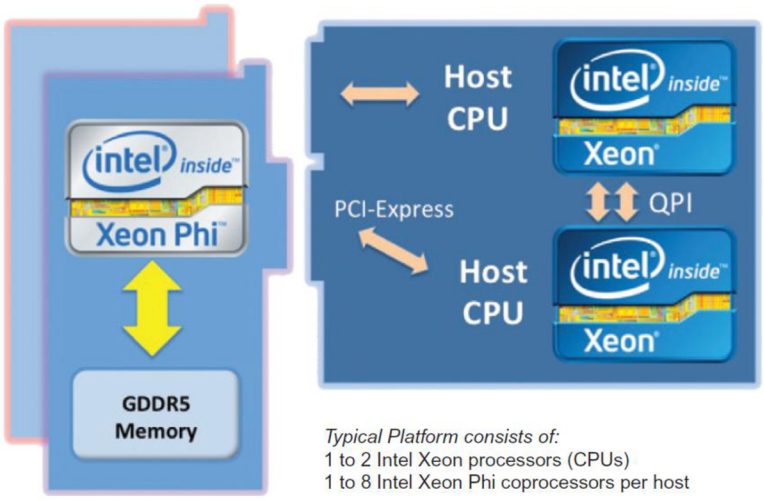
V tabuľke 2.1 nasleduje popis Git repozitára pre implementáciu C++ verzie so zdielanou pamäťou, pretože tú použijeme ako základnú verziu pre náš vývoj.

Data	Obsahuje príklady vstupných dát.
Doxygen	Obsahuje doxygen dokumentáciu.
GNUMakefile	Obsahuje rôzne druhy makefilu pre rôzne systémy.
GetoptWin64	Obsahuje implementáciu getopt() pre systém Windows.
HDF5	Obsahuje zdrojové kódy pre prácu s HDF5 (hierarchical data format)
KspaceSolver	Obsahuje jadro zdrojového kódu simulátora v C++.
Manual	Obsahuje manuál pre k-Wave toolbox v PDF formáte.
MatlabScripts	Obsahuje niekoľko skriptov pre porovnanie verzií v Matlab and C++.
MatrixClasses	Obsahuje implementáciu rôznych tried matíc v C++.
Parameters	Obsahuje zdrojové kódy implementácie pracovania príkazovej riadky a simulačných parametrov v C++.
Utils	Obsahuje zdrojové kódy pomocných funkcií a konštánt v C++.
nbproject	Obsahuje projektové súbory pre NetBeans.

Tabuľka 2.1 Obsah repozitára aplikácie k-Wave.

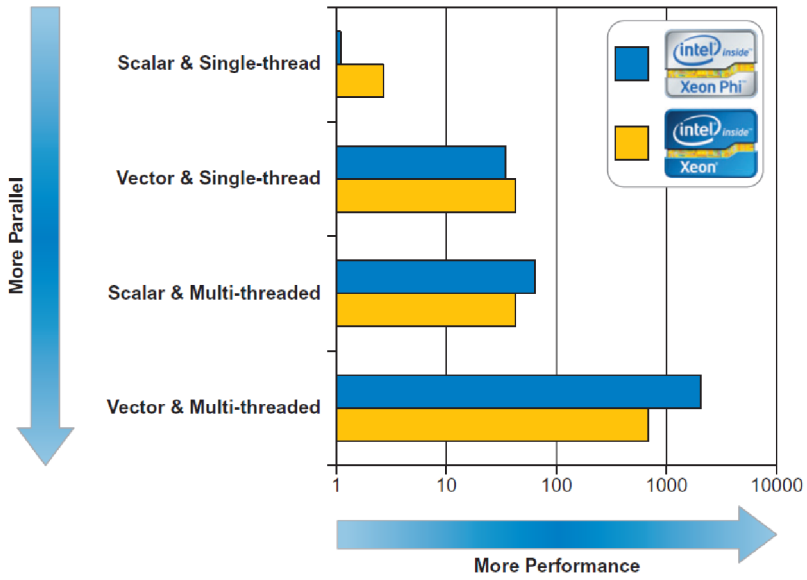
3 Koprocesor Intel Xeon Phi

Ako hlavný zdroj pre túto kapilolu boli použitá takmer výhradne kniha [7], ktorá združuje väčšinu materiálov poskytnutých spoločnosťou Intel vývojárom. Bude teda predpokladaná ako implicitný zdroj a nebudeme ju teda v texte referencovať. Naopak, iný zdroj ako táto kniha, bude explicitne uvedený.

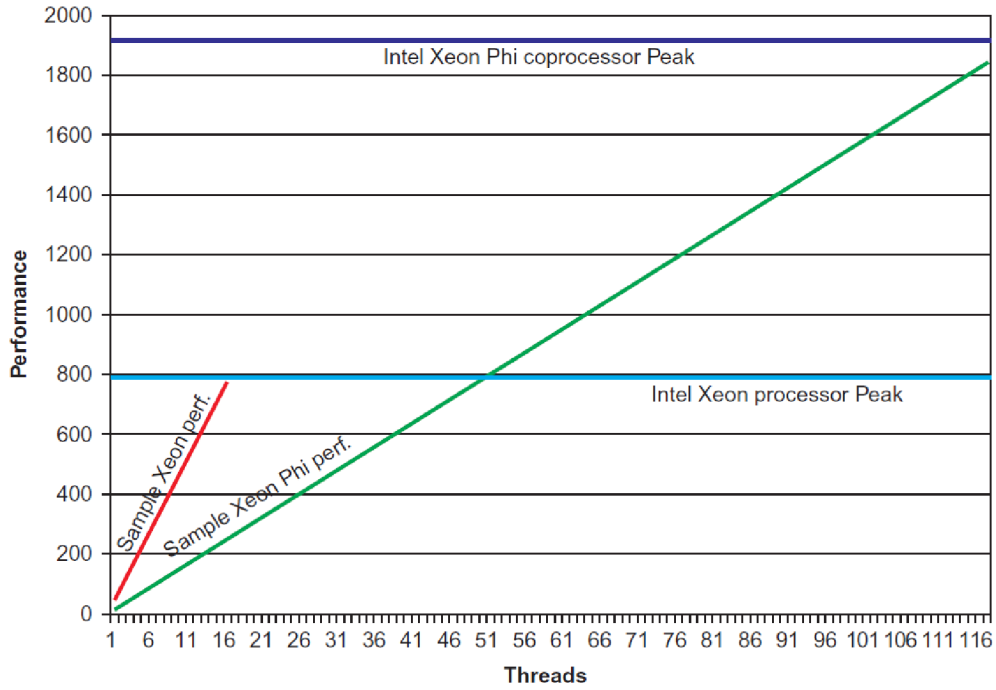


Obrázok 3.1 Zapojenie na jednom PC (výpočetnom uzle).

Karta Intel Xeon Phi býva radená medzi takzvané many-core platformy, čo sú platformy obsahujúce veľké množstvo procesorových jadier. Karta je koprocesorom, čo znamená, že je nutné ju vždy používať v systéme s iným, takzvaným host procesorom. Typické kombinácie zapojenia na jednom stroji/uzle ilustruje obrázok 3.1. Karty je samozrejme možné použiť aj vo výpočtovom (aj heterogénnom) clustri, kde sa potom takýchto uzlov nachádza omnoho viac. Koprocesory Intel Xeon Phi sú určené na použitie všade tam, kde je možné využiť vysoký počet vlákien a vektorových operácií. Naopak, pokiaľ ich nie je možné využiť, oplatí sa používať klasický procesor. Situáciu ilustrujú obrázky 3.2 a 3.3. Na druhom z obrázkov môžeme vidieť porovnanie hrubého špičkového výkonu medzi procesormi Intel Xeon a Xeon Phi, kde je vidieť, že na dosiahnutie maximálneho výkonu Intel Xeon je potrebné využiť násobne viac jadier na Intel Xeon Phi. Avšak, pokiaľ je úloha masívne paralelizovateľná, môžeme v konečnom dôsledku dosiahnuť taký výkon, ako nám klasický procesor nie je schopný ponúknuť.



Obrázok 3.2 Vplyv množstva vlákien spolu s vektorizáciou na výkon karty Intel Xeon Phi.

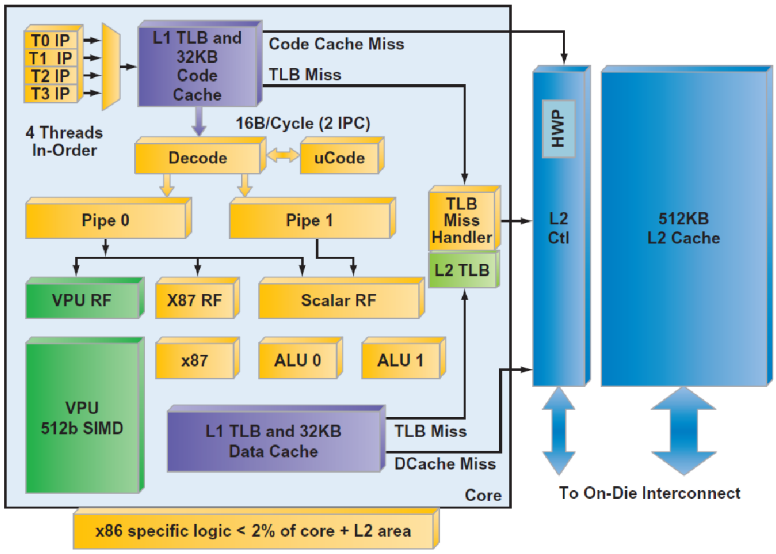


Obrázok 3.3 Porovnanie teoretického hrubého výkonu karty Inte Xeon Phi.

3.1 Architektúra

Z pohľadu programátora je kľúčové pozerať sa na Intel Xeon Phi ako na procesor so zdieľanou pamäťou založený na x86 architektúre, ktorý má viac ako *50* jadier, viac hardwarových vlákien na jadro a *512*-bitové SIMD (single instruction multiple data) inštrukcie. Jednotlivé jadrá sú *in-order*

x86 procesory, ktoré vychádzajú z dizajnu procesoru Pentium. Pridávajú však *64*-bitovú podporu, štyri hardvérové vlákna na jadro, riadenie spotreby, podporu kruhového prepojenia (ring interconnect), podporu *512*-bitových SIMD inštrukcií a ďalšie funkcie. Od Pentia sa tak už dosť líšia, *x86* logika bez *L2 cache* zaberá menej ako dve percentá celkovej plochy čipu.

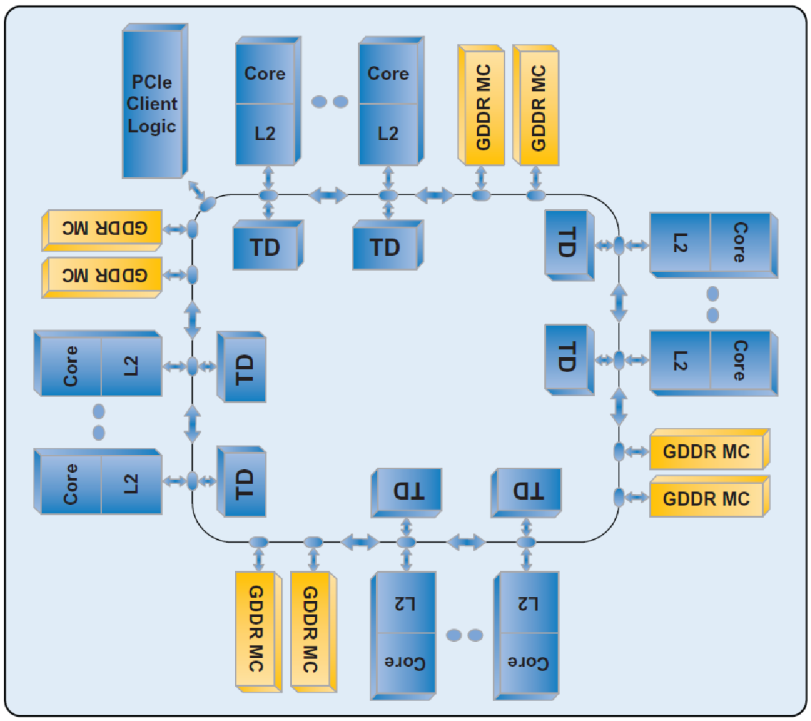


Obrázok 3.4 Jedno jadro koprocesoru Intel Xeon Phi.

Pre ilustráciu je architektúra jedného jadra je zobrazená na obrázku 3.4 a zapojenie všetkých jadier na obrázku 3.5. Základné fakty o koprocesore sú dobre zhrnuté v nasledujúcom zozname:

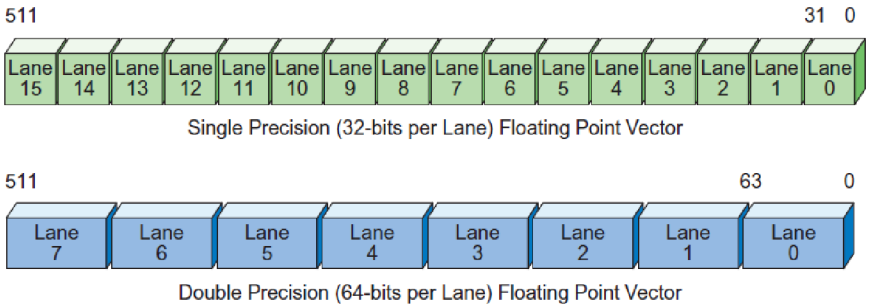
- Vyžaduje vždy aspoň jeden procesor v systéme.
- Beží na ňom OS Linux.
- Je vyrobený Intelom, pomocou *22 nm* technológie s použitím *3D* Trigate tranzistorov.
- Podporuje štandardné vývojárske nástroje (napríklad Intel Parallel Studio XE).
- Obsahuje veľa jadier:
    - Viac ako *50* jadier (záleží na konkrétnej generácii).
    - In-order jadrá podporujú *64*-bitové *x86* inštrukcie so širokými SIMD možnosťami.
    - Štyri hardvérové vlákna na každé jadro (vyše *200* hardvérových vlákien na jeden koprocesor), ktoré sú primárne použité na prekrytie latencie implicitnej pre in-order mikroarchitektúru.
    - Jadrá sú prepojené vysokorýchlostným obojsmerným okruhom (bidirectional ring). V praxi je takmer vždy výhodné použite minimálne dvoch vlákien na jadro.
    - Jadrá sú taktované na *1 GHz* a viac.
    - Cache je koherentná v celom procesore.
    - Každé jadro obsahuje *512 KB L2 cache* s vysoko-rýchlostným prístupom do všetkých ostatných *L2 cache* (spoločná *L2 cache* má viac ako 25MB).

- Cache poskytuje vysoko efektívne využívanie energie pri poskytnutí vysokej priepustnosti pamäti.



Obrázok 3.5 Zapojenie jadier v koprocesore Intel Xeon Phi.

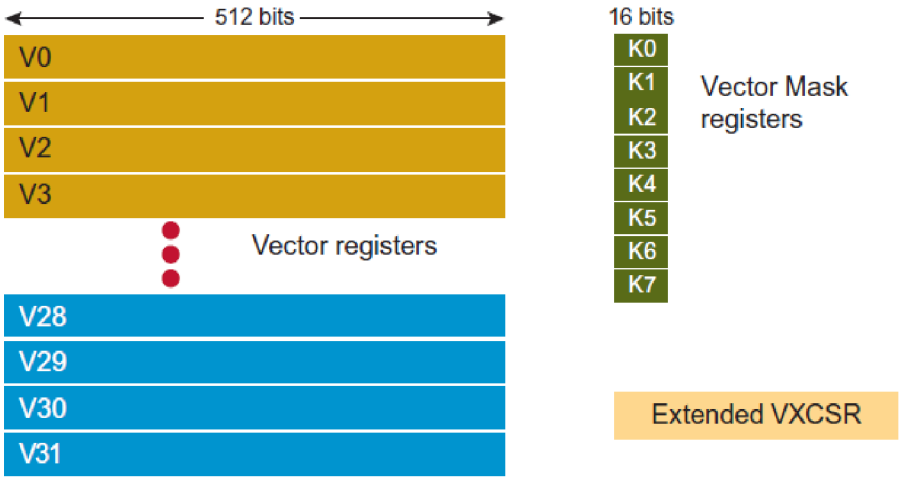
- Pridáva špeciálne inštrukcie ako doplnok k *64*-bitovým *x86* inštrukciám:
  - Podpora širokých SIMD inštrukcií cez *512*-bitové vektory namiesto užších MMX, SSE a AVX inštrukcií.
  - Vysokovýkonnú podporu pre inverzné, odmocninové, mocninové a exponenciálne operácie.
  - Podporu pre scatter/gather a prúdové ukladanie na dosiahnutie vysokej priepustnosti pamäti.
- Pridáva ešte ďalšie špeciálne vlastnosti:
  - Pamätový radič podporujúci až *8 GB* pamäti typu *GDDR5* .
  - PCIe spojovaciu logiku priamo na čipe.
  - Podporu riadenia spotreby energie.



Obrázok 3.6 Vektorový formát inštrukcií na Intel Xeon Phi.

Vektorová jednotka

Každý procesor obsahuje jednu *512*-bitov širokú SIMD vektorovú jednotku (Vector Processing Unit) spolu s príslušnou vektorovou inštrukčnou sadou. Jej využívanie je minimálne tak dôležité, ako využívanie veľkého množstva jadier koprocesora. Jednotka dokáže spracovať *16* elementov s jednoduchou presnosťou alebo *8* elementov s dvojitou presnosťou počas jedného taktu (obrázok 3.6). Jednotka obsahuje *32* vektorových registrov, *8* maskových registrov (*16*-bit) a VXCSR stavový register (obrázok 3.7).



Obrázok 3.7 Dostupné registre vektorovej jednotky na Intel Xeon Phi.

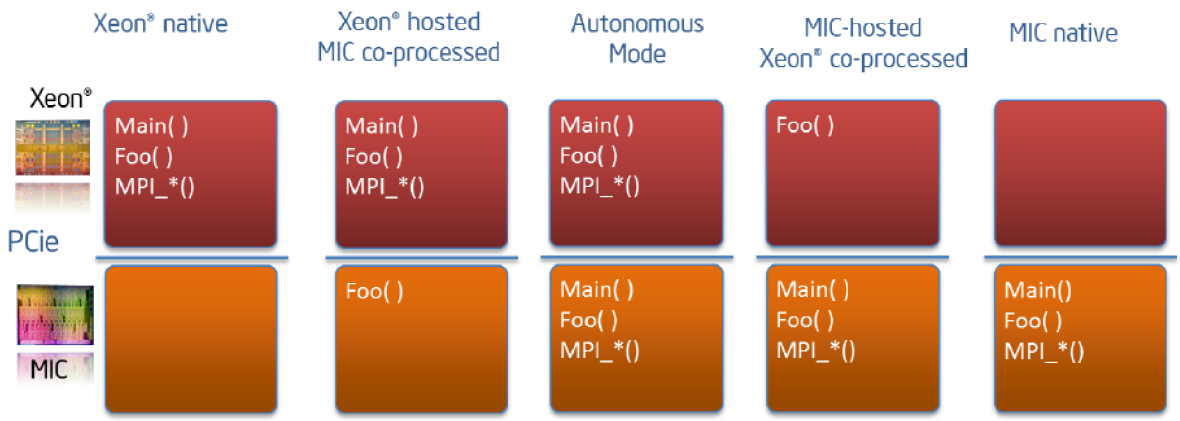
Vektorové inštrukcie podporujú nasledujúce natívne dátové typy:

- Packed *32*-bit integers (or dword)
- Packed *32*-bit SP FP values
- Packed *64*-bit integers (or qword)
- Packed *64*-bit DP FP values

Zoznam všetkých inštrukcií je k dispozícii v [8]. Ich podrobná znalosť však nie je potrebná v prípade, že sa spoliehame na schopnosti auto-vektorizácie, ktorú nám vie poskytnúť kompilátor.

3.2 Programovacie modely

Systém, v ktorom sa nachádza karta s architektúrou Intel Xeon Phi, ponúka viaceré možnosti použitia. To umožňuje programátorom riešiť rôzne problémy rôznymi spôsobmi a nakoniec použiť ten najlepší podľa typu riešeného problému a jeho výkonnostným nárokom.



Obrázok 3.8 Modely exekúcie kódu na procesore a na Intel Xeon Phi.

Natívny model

Pri použití tohto modelu sa akcelerátor správa ako samostatný počítač – na kartu sa dá pripojiť ako na akýkoľvek iný počítač pomocou *ssh*. Je na nej vykonávaný úplne celý kód programu. Nato, aby bol tento model vhodný pre použitie, musí byť programovaný problém vysoko paralelizovateľný a obsahovať minimálne množstvo sériového kódu. Je to z toho dôvodu, že vykonávanie sériového kódu (na jednom jadre) je oveľa pomalšie ako na bežnom procesore. Ďalej je potrebné brať do úvahy fakt, že množstvo pamäti je obmedzené len na pamäť samotnej karty. Veľkou výhodou je jednoduchšia portabilita existujúceho kódu.

Offload model

Tento model je založený na myšlienke, že na akcelerátore budú vykonávané len náročné paralelné výpočty. Zvyšný sériový kód bude vykonávaný na klasickom procesore, ktorý ho dokáže vykonávať rýchlejšie. Dokážeme tak využiť výkon oboch aj procesoru aj koprocesoru. Ďalšou výhodou je väčšie

dostupné množstvo pamäti. Nevýhodou môže byť zložitejšie programovanie a portovanie existujúceho kódu.

Existujú tri typy offload módu:

- Explicitný – Programátor sám určí, ktoré časti kódu bude vykonávať akcelerátor a ktoré dáta budú prenesené do jeho pamäti.
- Implicitný – Programátor označí dáta ako zdielané a samotný systém zariadi synchronizáciu za behu programu.
- Automatický – Náročné volania knižnice MKL sa automaticky spracujú na akcelerátore.

MPI model

Tento model sa spolieha na knižnicu Intel MPI (message-passing interface), ktorá implementuje špecifikáciu predávania správ MPI-*2.1*. Je založený na behu MPI procesov na jednotlivých procesoroch/koprocesoroch v systéme, ktoré spolu komunikujú. Umožňuje väčšiu škálovateľnosť (cez množstvo uzlov, napríklad výpočtového clustru) a tým aj väčšie množstvo dostupnej pamäti.

Existuje niekoľko variantov použitia MPI modelu podľa toho, kde bežia jednotlivé MPI procesy:

- Symetrický model – MPI procesy bežia na obidvoch procesoroch – host procesore a aj na Intel Xeon Phi koprocesore. Toto je najzákladnejší MPI pohľad na heterogénny cluster.
- Host-only model – Všetky MPI procesy bežia len na host procesore. Na tento model je možné sa pozerať ako na špeciálny prípad symetrického modelu. V tomto prípade je koprocesor Intel Xeon Phi použitý pre offload.
- Coprocesor-only model – Všetky MPI procesy bežia len na koprocesore Intel Xeon Phi. Aj na tento model je možné sa pozerať ako na špeciálny prípad symetrického modelu. V tomto prípade môže byť použitý pre offload klasický host procesor.

3.3 Programové prostriedky

Spoločnosť Intel poskytuje vývojové nástroje pre vývoj na karte Intel Xeon Phi. Jedná sa o Intel Parallel Studio XE (predtým Intel Composer XE), ktoré obsahuje vývojové prostredie zahrňujúce kompilátory, rôzne knižnice a ďalšie nástroje.

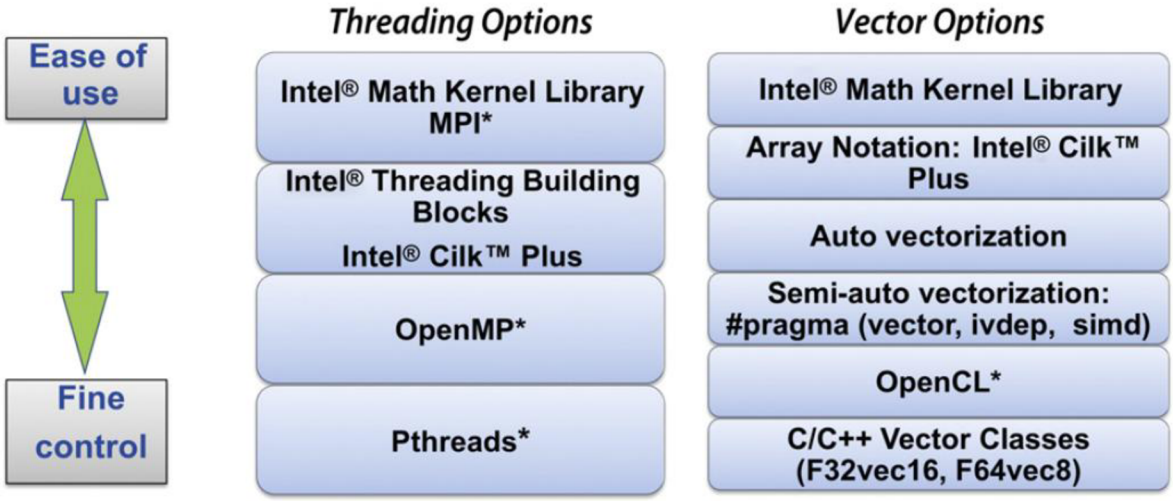
Obsiahnuté sú nasledujúce podporované kompilátory a knižnice:

- Intel C/C++ Compiler obsahujúci Intel MIC architektúru pre kompiláciu aplikácií bežiacich na Intel *64* a Intel MIC architektúrach.
- Intel Fortran Compiler obsahujúci Intel® MIC architektúru pre kompiláciu aplikácií bežiacich na Intel *64* a Intel MIC architektúrach.

Vývojové prostredie obsahuje aj ďalšie dôležité nástroje pre debugovanie a profilovanie:

- Intel Debugger pre debugovanie aplikácií bežiacich na Intel Architecture (IA) a Intel MIC architektúrach.
- Nástroj SEP nám umožňuje zbierať výkonnostné dáta z koprocesoru Intel Xeon Phi. Tieto dáta potom môžeme analyzovať použitím nástroja VTune Amplifier XE.

Programovanie je pre Intel Xeon Phi rovnaké ako pre iné Intel procesory. Oveľa väčší dôraz je však kladený na paralelizáciu kódu. To zahŕňa hlavne programovanie vlákien a vektorizáciu, ktoré sú vzhľadom na architektúru karty, esenciálnymi programovacími technikami karty. Všetky možnosti ich implementácie sú viditeľné na obrázku 3.9. Každý programátor si môže zvoliť jemu vyhovujúce prostriedky podľa toho, či vyžaduje jednoduchosť použitia (horné možnosti) alebo chce mať väčšiu kontrolu na úrovni kódu (dolné možnosti).



Obrázok 3.9 Prostriedky pre programovanie paralelizmu a vektorizácie.

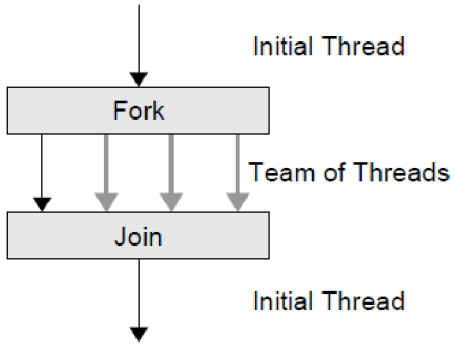
My budeme v projekte používať OpenMP, konkrétne vo verzii *4.0*, ktorá pridáva podporu pre akcelerátory, SIMD konštrukcie pre vektorizovanie sériových aj paralelných cyklov a pre afinitu vlákien [11]. Ďalej použijeme aj knižnicu Intel MKL, ktorá je optimalizovaná aj pre použitie na našom akcelerátore.

3.4 OpenMP

Informácie použité v tejto podkapitole sú prebraté z [9] a [12].

Všeobecný popis

OpenMP API (application programming interface) bolo vytvorené pre umožnenie relatívne jednoduchého paralelného programovania so zdieľanou pamäťou. Umožňuje použitie postupného paralelizovania už existujúceho sériového kódu, pričom nie je potrebné vykonávať veľké zmeny a rovnaký zdrojový kód môže byť použitý pre sériovú aj paralelnú verziu programu. Časť kódu, ktorá bude vykonávaná paralelne je uzavretá do paralelnej sekcie pomocou *#pragma omp parallel*. Keď program vstúpi do takejto sekcie, automaticky vygeneruje skupinu vlákien, ktoré následne paralelne spracúvajú rovnaký kód. Na konci paralelnej sekcie je implicitná bariéra, kde sa všetky vlákna navzájom počkajú. Potom už program pokračuje sériovo, rovnako ako pred vstupom do paralelnej sekcie. Ilustráciu tohto takzvaného fork-join programovacieho modelu, ktorý používa knižnica OpenMP, môžeme vidieť na obrázku 3.10.



Obrázok 3.10 Fork-join programovací model používaný knižnicou OpenMP [12]

Knižnica ďalej okrem vytvorenia skupiny vlákien ponúka veľké množstvo funkcionality, ako napríklad: možnosť špecifikovať rozdelenie práce medzi jednotlivé vlákna; efektívne paralelizovať slučky; deklarovať premenné, ktoré budú privátne pre každé vlákno alebo zdieľané medzi všetkými vláknami; synchronizáciu medzi vláknami alebo aj možnosť vykonávať niektoré operácie len jednému vláknu. To všetko a ešte oveľa viac sa dá dosiahnuť pomocou direktív a ich klauzúl definovaných v OpenMP API [11].

Afinita vlákien

Vo verzii 4.0 pridáva OpenMP podporu pre vlastné nastavenie afinity, čím nám umožnuje nastavovať priradenie exekúcie jednotlivých vlákien na určité jadrá procesoru. Intel kompilátor potom poskytuje premennú prostredia *KMP_AFFINITY*, pomocou ktorej nám dovoluje nastaviť jednu z nasledujúcich základných typov afinity [7]:

1. *Compact* – vlákna budú priraďované na jednotlivé jadrá procesoru postupne. Takže keď bude na prvom jadre plný počet vlákien, tak ďalšie vlákna začnú byť priraďované na druhé jadro a takto postupne ďalej, až pokiaľ niesú priradené všetky vlákna. Pokiaľ teda nie je dostatok vlákien pre všetky jadrá, ľahko sa môže stať, že niektoré jadrá nebudú mať žiadnu prácu a niektoré budú preťažené. Naopak, pokiaľ by boli všetky jadrá dobre nasýtené, môžeme dosiahnuť lepší výkon, pretože vlákna spracúvajúce susedné prvky budú priradené na rovnaké jadro – a môžu tak prosperovať z dátovej lokality (prvky možu byť načítané v cache).
2. *Scatter* – vlákna budú priraďované jednotlivým jadrám postupne po jednom. Takže až keď bude každé jadro spracovávať jedno vlákno, tak sa začnú priraďovať jardám druhé vlákna. Jednoznačne je tu lepšie rozloženie záťaže v prípade menšieho počtu vlákien. Naopak, pokiaľ bude počet vlákien privysoký, jadrá budú vyťažené naplno, ale keďže vlákna spracúvajúce susedné prvky budú rozhádzané po rôznych jadrách, tak narastá réžia a nemôžeme profitovať z dátovej lokality.
3. *Balanced* – vlákna budú priraďované podľa aktuálneho vyťaženia jednotlivých jadier.

Výber správnej afinity spolu s počtom vytvorených vlákien veľkým spôsobom ovplyvňuje celkový výkon na akcelerátore Intel Xeon Phi, a preto je s ich nastavením vždy potrebné experimentovať.

4 Navrhnutý postup riešenia

Na začiatok je potrebné povedať, že táto diplomová práca bude zameraná na implementáciu pre jednu kartu Intel Xeon Phi pracujúcu v natívnom móde. To znamená, že bude pracovať ako jediný a samostatný výpočtový uzol bez účasti CPU. Preto je potrebné si uvedomiť že aplikácia, ktorá má takto bežať, musí byť vysoko paralelizovateľná a obsahovať čo najmenej serializovaných sekcií. Ďalšou veľmi dôležitou vecou pre dosiahnutie vysokého výkonu je tiež nutnosť využívať vektorové inštrukcie karty. Musíme si uvedomiť aj aké obmedzenie nám použitie tohoto modelu prinesie: pri simulácii budeme automaticky obmedzení veľkosťou fyzickej pamäti našej karty. Toto obmedzenie má za následok, že už nebude možné prevádzať simulácie pre dáta o rozmeroch približne rovných a väčších ako *512x512x256*. Táto kapitola prezentuje postup riešenia tejto diplomovej práce. Postup bude pozostávať z dvoch hlavných krokov – vytvorenia základnej verzie (prototypu) a jeho následnej optimalizácie.

4.1 Vytvorenie základnej verzie

Prvým krokom pre ďalší postup bude vytvorenie základnej verzie implementácie, ktorá už bude bežať na karte Intel Xeon Phi. Pre vytvorenie prototypu budeme postupovať nasledovne.

1. Použitie existujúcej verzie

   Použijeme paralelnú C++ implementáciu aplikácie k-Wave toolbox vo verzii so zdieľanou pamäťou. Tá sľubuje vysokú pravdepodobnosť znovupoužitia jej kódu, keďže rovnako využíva OpenMP. Dokonca niekedy je možné, že aby aplikácie bežali na karte Intel Xeon Phi, je potrebné urobiť len minimálne alebo aj žiadne zmeny v zdrojovom kóde.
2. Upravenie zdrojového kódu

   Musíme upraviť časť zdrojového kódu. K-Wave toolbox je totiž optimalizovaný pomocou Intel SSE (streaming SIMD extensions) inštrukcií použitím ich intrinsic varianty priamo v zdrojovom kóde. SSE inštrukcie však nie sú podporované architektúrou karty Intel Xeon Phi.
3. Použitie externých knižníc

   Ďalšie komplikácie nastávajú pri použití externých knižníc, ktoré bude nutné minimálne prekompilovať na beh pre rozdielnu architektúru koprocesoru Intel Xeon Phi [13].
4. Kompilácia

   Skompilovať aplikáciu pre cieľovú architektúru karty Intel Xeon Phi (parameter *–mmic*).

4.2 Optimalizácia vytvoreného prototypu

Pri optimalizácii sa budeme snažiť zvýšiť výkonnosť vytvoreného prototypu použitím optimalizačných techník. Podľa toho ako bude prebiehať, ju môžeme rozdeliť do troch častí. V prvej časti nahradíme výpočtovo najnáročnejšiu časť implementácie optimalizovanou knižnicou. Ďalšie dve časti budú zahŕňať použitie profilovania. To nám pomôže s analýzou a identifikovaním výpočtovo najnáročnejších miest implementácie. Na základe tejto analýzy potom budeme môcť pokračovať v optimalizáciách na tých správne identifikovaných miestach. Veľmi dôležitou súčasťou procesu bude potreba výkonávania množstva experimentov. Bude potrebné vyskúšať napríklad použitie rôznych typov afinít a rôzneho počtu vlákien a ich kombinácií. Celý tento proces bude iteratívny, teda bude sa opakovať, kým bude mať ďalšie optimalizovanie zmysel.

Počítanie Fourierovej transformácie

Podľa [1], až okolo *60 %* celkového výpočtového času simulácie je stráveného počítaním FFT. To je vysvetlené tým, že v každom kroku simulácie sa počítanie FFT vykoná *6*-krát a počítanie iFFT 8-krát [14]. Je nám teda jasné, že bude potrebné sa zamerať na zrýchlenie jeho počítania. Ako najjednoduchšie a aj najefektívnejšie riešenie sa ponúka použitie externej knižnice. Súčasná implementácia spolieha na jednu z knižníc FFTW, respektíve Intel MKL, ktoré sú optimalizované pre beh na viacjadrových procesoroch. Opäť použijeme knižnicu Intel MKL, pretože existuje aj vo verzii optimalizovanej priamo pre kartu Intel Xeon Phi. Keďže kartu používame v natívnom móde ako samostatný nezávislý výpočtový uzol, tak ekvivalentne použijeme aj variantu knižnice Intel MKL, ktorá je pre tento mód určená [15].

Paralelizácia

Na paralelizáciu kódu pre maximálne využitie jadier akcelerátoru použijeme OpenMP. Základný kód, z ktorého vychádzame, už prešiel optimalizáciou na viacjadrové procesory práve za pomoci OpenMP. Bude preto potrebné zmerať výkon, prípadne sa pokúsiť implementáciu vylepšiť.

Vektorizácia

Na vektorizáciu je možné použiť napríklad niektorý z nasledujúcich troch prístupov:

a. Použiť základné operácie dostupné v jazyku C++ a následne sa snažiť vektorizovať kód pomocou autovektorizácie kompilátora. Samotná úspešnosť vektorizácie by závisela na schopnostiach kompilátora na použitej platforme.

b. Použiť intrinsic inštrukcie, konkrétne tie ktoré sú podporované kartou Xeon Phi. Kód by bol teda znova vektorizovaný manuálne. Tento prístup by si samozrejme zachoval aj problém s prenositeľnosťou a kód by sa dal použiť len na architektúre podporujúcej rovnaké *512*-bitové vektorové inštrukcie.
c. Použitie nápovied kompilátoru alebo SIMD direktív, ktoré sú spolu s podporou akcelerátorov dostupné s verziou OpenMP 4.0. Takýto kód bude dobre presnositeľný a pod kontrolou programátora.

Pre vektorizáciu použijeme SIMD direktívy podporované v OpenMP 4.0. Budeme pritom postupovať nasledujúcimi krokmi:

1. Zmeraj výkon základnej implementácie.
2. Urči najvyťaženejšie miesta zdrojového kódu použitím Intel Vtune Amplifier XE.
3. Urči cykly, ktoré sú možnými kandidátmi pre vektorizáciu. Použi Intel Compiler vec-report.
4. Zvektorizuj ich, pokiaľ je to možné.
5. Kroky opakuj znovu.

5 Implementácia riešenia

V implementácii tejto práce budeme pokračovať v navrhnutom postupe. Použité informácie a postupy boli čerpané hlavne z [7], [10], [11], [16] a [17]. Veľmi hodnotným zdrojom, hlavne v prípade problémov, je fórum je pre vývojárov prevádzkované priamo firmou Intel [18]. V prvom rade si popíšeme zmeny, ktoré je potrebné vykonať priamo v zdrojovom kóde, a postup, ktorým skompilujeme používané knižnice. Tým zostrojíme prvý prototyp aplikácie spustiteľný na karte Xeon Phi. Následne si ukážeme postup, ktorým je možné natívnu aplikáciu profilovať pomocou aplikácie Intel VTune. Uvedieme pritom aj všetky problémy, na ktoré sme pri používaní narazili. Potom už prejdeme na hľadanie najvyťaženejších miest v našej aplikácii a ich samotnej optimalizácii.

Projekt bol implementovaný na superpočítači *Anselm* [19], kde sa nachádzajú štyri výpočtové uzly obsahujúce nám dostupné karty typu *Intel Xeon Phi 5110P*. Verzia pre bežné procesory bola spúšťaná na uzle používajúcom procesor rodiny *Sandy Bridge – Intel Xeon E5-2470* a *96GB* pamäti RAM. Použité boli prostriedky, ktoré sú na tomto superpočítači dostupné: pre kompiláciu *Intel Compiler XE 2015*, pre profilovanie *Intel Vtune XE 2013* a pre overenie správnosti výpočtu *Matlab*. Spomenieme ešte, že implementovanie sa nezaobišlo bez problémov na strane Anselmu, ako výpadky používaných uzlov a ich služieb, či problémy s Vtune, ktoré bolo nutné riešiť s technickou podporou.

5.1 Zostavenie základnej verzie

Úprava kódu

Pre natívny beh je typické, že pôvodný zdrojový kód nebude potrebné upravovať vôbec, alebo len minimálne. To však platí iba v prípade, že aplikácia využíva len štandardné inštrukcie podporované aj architektúrou Xeon Phi. Aplikácia k-Wave už bola predtým optimalizovaná pre bežné procesory použitím *SSE2 intrinsic* inštrukcií – nepodporovanými na architektúre Xeon Phi. Zvektorizované tak boli dve metódy v súbore \KspaceSolver\KspaceFirstOrder3Dsolver.cpp: *Calculate_SumRho_BonA_ SumDu_SSE2()* a *Compute_Absorb_nabla1_2_SSE2()*. Pre zostrojenie prototypu tieto dve metódy upravíme tak, aby počítali skalárne – odstránime tak *SSE2* inštrukcie. Zachováme pritom implementovanú paralelizáciu pomocou OpenMP pragiem. Posledná úprava kódu v rámci zostrojenia prototypu je upravenie konštanty definujúcej počet bajtov, na ktorý bude zarovnaná alokovaná pamäť pre dáta. K dostupným možnostiam v \Utils\DimensionSizes.h pridáme voľbu pre Xeon Phi definujúcu zarovnanie na *64* bajtov (*512* bitov).

Prekompilovanie knižníc

Ďalším krokom k funkčnému prototypu je kompilácia použitých knižníc pre natívny beh. Je potrebné preložiť knižnicu HDF5, ktorá sa stará o vstupné a výstupné súbory uložené v rovnomennom formáte. Knižnica HDF5 ďalej využíva knižnicu Zlib umožňujúcu použitie komprimácie pre tieto súbory. Rozbehanie týchto dvoch knižníc nebolo také jednoduché ako to nakoniec vyzerá a zabralo veľké množstvo času. Hlavným zdrojom bol návod z [20], a potom množstvo pokusov, až sme nakoniec prišli na fungujúci postup. Ten predkladáme formou dobre okomentovaných shell skriptov, čo je asi najlepší možný popis pre jeho jednoduché zreprodukovanie. Pre úspešnú kompiláciu je potrebné dodržať poradie.

1. Kompilácia knižnice Zlib

```
1.  ###########################################
2.  #### Kompilácia zlib pre Xeon Phi (MIC).
3.  # 1 Načítaj modul Intel kompilátora (platí pre superpočítač Anselm). Musíš byť na
4.  #   počítači, ktorý má nainštalovanú kartu Xeon Phi a tiež Intel Manycore Platform
5.  #   Software Stack (MPSS).
6.  module load intel/15.2.164
7.  # 2 Nastav premenné prostredia pomocou skriptu.
8.  source /opt/intel/composer_xe_2015.2.164/bin/compilervars.sh intel64
9.  # 3 Prejdi do zložky, kde sú stiahnuté a rozbalené zdrojové súbory knižnice zlib.
10. cd /home/DIP/zlib-1.2.8/
11. # 4 Nastav nasledujúce premenné prostredia pre kompiláciu na Xeon Phi platformu.
12. export CFLAGS="-mmic -O3"
13. export CXXFLAGS="-mmic -O3"
14. export LDFLAGS="-mmic -O3"
15. # 5 Spusti nasledujúcu konfiguráciu pre následnú kompiláciu (malo by vygenerovať aj
16. #   verzie knižnice pre dynamické linkovanie. Pokiaľ nie, zmeň "static" na "shared").
17. ./configure --static --prefix=/home/DIP/zlib-1.2.8/zlib-installed --64
18. # 6 Skompiluj zdrojové súbory.
19. make
20. # 7 Nainštaluj do nastavenej zložky (definovanej v --prefix).
21. make install
```

Kód 5.1 Shell skript pre kompiláciu knižnice Zlib.

2. Kompilácia knižnice HDF5 pre host procesor

```
1.  ###########################################
2.  #### Kompilácia hdf5 pre host PC
3.  # 1 Načítaj modul Intel kompilátora (platí pre superpočítač Anselm) a tiež modul
4.  #   knižnice Zlib (pre host procesor je dostupná priamo na Anselme – netreba kompilovať).
5.  module load intel/15.2.164
6.  module load zlib/1.2.8
7.  # 2 Nastav premenné prostredia pomocou skriptu.
8.  source /opt/intel/composer_xe_2015.2.164/bin/compilervars.sh intel64
9.  # 3 Prejdi do zložky, kde sú stiahnuté a rozbalené zdrojové súbory knižnice hdf5.
10. cd /home/DIP/hdf5-1.8.13
11. # 4 Vytvor novú zložku kde bude prebiehať kompilácia na host procesor.
12. mkdir host_build
13. # 5 Prejdi do zložky kde bude prebiehať kompilácia na host procesor.
14. cd host_build
15. # 6 Spusti nasledujúcu konfiguráciu pre následnú kompiláciu na host procesor.
16. ../configure CFLAGS="-O3" CXXFLAGS="-O3"                                    \
17.   --prefix=/home/DIP/hdf5-1.8.13/host_build/hdf5-installed/ --enable-hl    \
18.   --enable-static --enable-shared --without-szlib --with-zlib --enable-cxx
19. # 7 Skompiluj zdrojové súbory.
20. make
21. # 8 Nainštaluj do nastavenej zložky (definovanej v --prefix).
22. make install
```

Kód 5.2 Shell skript pre kompiláciu knižnice HDF5 pre host procesor.

3. Úprava HDF5 súborov. Je nutné urobiť dva typy zmien. Prvý z nich je popísaný v bodoch a) – c). Druhý je popísaný v bodoch i. – iii.

   a) Toto je zoznam súborov, ktorých je potrebné vykonať prvý typ zmeny:
      - \test\Makefile.in
      - \tools\h5copy\Makefile.in
      - \tools\h5diff\Makefile.in
      - \tools\h5dump\Makefile.in
      - \tools\h5ls\Makefile.in
      - \tools\h5stat\Makefile.in
      - \tools\misc\Makefile.in

   b) Musíme identifikovať riadky obsahujúce reťazec *„TEST_SCRIPT = xyz“*, kde *xyz* je zástupný podreťazec obsahujúci odkaz na iný skript.

   c) Takto identifikované riadky musíme upraviť tak, že reťazec *xyz* vymažeme aj spolu s medzerami, ktoré sú pred ním. Na riadku tak zostane reťazec *„TEST_SCRIPT =”*.

   i. Zmeny je potrebné spraviť len v súbore:
      - \hdf5-1.8.9\configure

   ii. Musíme identifikovať tri miesta, na ktorých je definícia funkcie *as_fn_error()*.

   iii. Na všetkých identifikovaných miestach z nej musíme vymazať (alebo zakomentovať) riadok obsahujúci reťazec „*as_fn_exit $as_status*“.

4. Kompilácia knižnice HDF5 pre Xeon Phi koprocesor

```
1.  ###########################################
2.  #### Kompilácia hdf5 pre Xeon Phi (MIC).
3.  # 1 Načítaj modul Intel kompilátora (platí pre superpočítač Anselm). Musíš byť na
4.  #   počítači, ktorý má nainštalovanú kartu Xeon Phi a tiež Intel Manycore Platform
5.  #   Software Stack (MPSS).
6.  module load intel/15.2.164
7.  # 2 Nastav premenné prostredia pomocou skriptu.
8.  source /opt/intel/composer_xe_2015.2.164/bin/compilervars.sh intel64
9.  # 3 Prejdi do zložky, kde sú stiahnuté a rozbalené zdrojové súbory knižnice hdf5.
10. cd /home/DIP/hdf5-1.8.13
11. # 4 Vytvor novú zložku kde bude prebiehať kompilácia na Xeon Phi procesor.
12. mkdir mic_build
13. # 5 Prejdi do zložky kde bude prebiehať kompilácia na Xeon Phi procesor.
14. cd mic_build
15. # 6 Spusti nasledujúcu konfiguráciu pre následnú kompiláciu na Xeon Phi procesor.
16. ../configure CFLAGS="-mmic -O3" CXXFLAGS="-mmic -O3"                          \
17.   --prefix=/home/DIP/hdf5-1.8.13/mic_build/hdf5-installed/ --enable-hl       \
18.   --enable-shared --enable-static --without-szlib                            \
19.   --with-zlib=/home/DIP/zlib-1.2.8/zlib-installed --enable-cxx               \
20.   --host=x86_64-unknown-linux-gnu
21. # 7 Spusti kompiláciu. Kompilácia nakoniec spadne na chybe – toto je správne,
22. #   pokračuj nasledujúcimi krokmi.
23. make
24. # 8 Skopíruj súbor na ktorom došlo k chybe. Z kompilácie pre hosta -> pre Xeon Phi.
25. cp ../host_build/src/H5lib_settings.c src/
```

```
# 9 Opäť spusti kompiláciu. Kompilácia opäť spadne na chybe – toto je správne,
#   pokračuj nasledujúcimi krokmi.
make
# 10 Skopíruj súbor na ktorom došlo k chybe. Z kompilácie pre hosta -> pre Xeon Phi.
cp ../host_build/src/H5Tinit.c src/
# 11 Opäť spusti kompiláciu. Teraz by už malo všetko úspešne dobehnúť do konca.
#    Ak by náhodou kompilácia znova spadla, je potrebné opakovať prechádzajúci
#    postup pre súbor, na ktorom došlo k chybe.
make
# 12 Nainštaluj do nastavenej zložky (definovanej v --prefix).
make install
```

Kód 5.3 Shell skript pre kompiláciu knižnice HDF5 pre Xeon Phi koprocesor.

Kompilácia a spustenie

Pre kompiláciu používame súbor \GNUMakefile\Ubuntu-14.04\Makefile. Je potrebné v ňom nastaviť správnu cestu ku skompilovaným knižniciam a pridať parameter „*-mmic*“, ktorý značí kompiláciu aplikácie pre natívne spustenie na karte Xeon Phi. Po týchto úpravách je možné úspešne preložiť zdrojové kódy aplikácie k-Wave. Spustenie aplikácie z host uzlu je možné napríklad spôsobom zobrazeným v kóde 5.4. Je pritom potrebné najskôr nastaviť premennú prostredia *LD_LIBRARY_PATH* tak, aby obsahovala cesty ku štandardným knižniciam a knižnici MKL, skompilovaným pre Xeon Phi – tie sú súčasťou Intel kompilátora. Ďalej je potom nutné nastaviť požadovaný typ afinity pomocou *KMP_AFFINITY* a počet použitých vlákien. Aplikáciu sme upravili tak, že je možné nastaviť rôzny počet vlákien pre knižnicu MKL pomocou *MKL_NUM_THREADS* a pre ostatné časti programu pomocou *OMP_NUM_THREADS*. Ak nie je nastavená premenná prostredia *MKL_NUM_THREADS*, použije knižnica MKL rovnaký počet vlákien ako zvyšok programu. Pre parametre samotnej aplikácie je potrebné riadiť jej manuálom.

```
ssh mic0 "
    export LD_LIBRARY_PATH=/apps/intel/composer_xe_2015.2.164/compiler/lib/mic/:${LD_LIBRARY_PATH};
    export LD_LIBRARY_PATH=/apps/intel/composer_xe_2015.2.164/mkl/lib/mic/:${LD_LIBRARY_PATH};
    export KMP_AFFINITY=compact;
    export OMP_NUM_THREADS=240
    export MKL_NUM_THREADS=240
    ./kspaceFirstOrder3D-OMP -i /home/DIP/Data/input_data_240_240_240_het_linear_nocomp.h5
                             -o /home/DIP/output.h5"
```

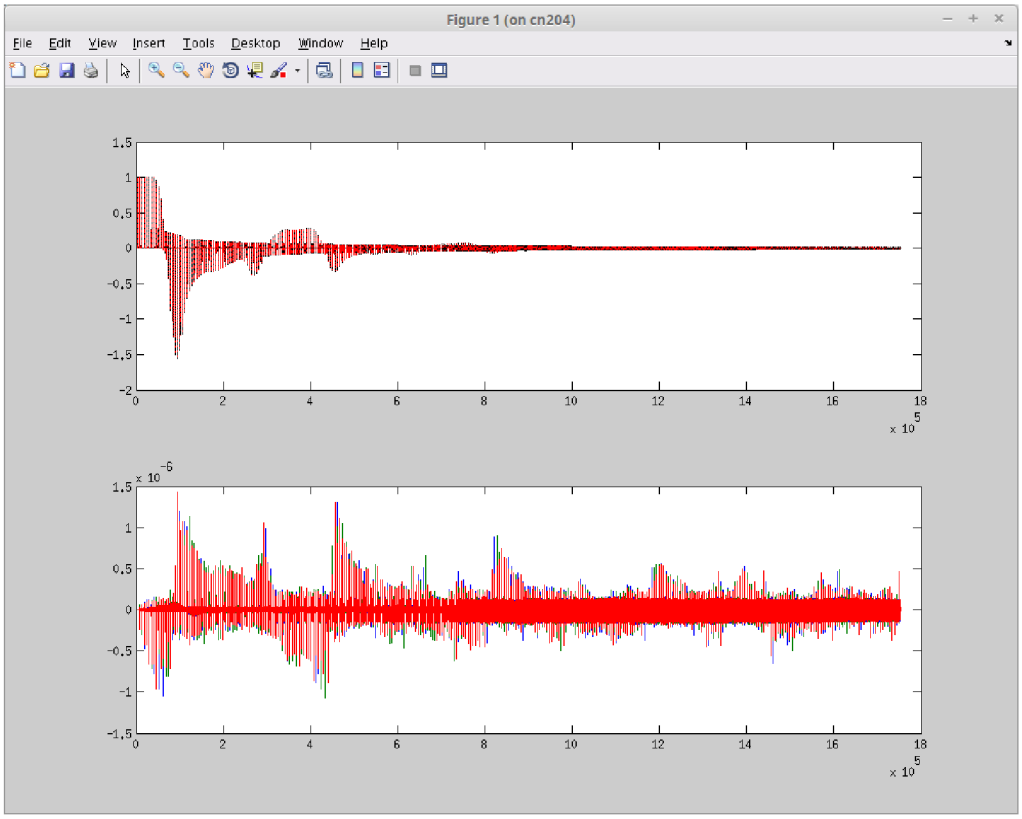
Kód 5.4 Príkaz pre spustenie aplikácie k-Wave na karte Intel Xeon Phi z host počítača.

Overenie funkčnosti

Pre overenie funkčnosti základnej verzie (a potom aj každej ďalšej vytvorenej verzie) sme použili jednoduchý Matlab skript, ktorý porovnáva výstup našej verzie s výstupom verzie pre bežný procesor. O tej vieme, že poskytuje korektné výsledky. Na obrázku 5.1 môžeme vidieť práve takéto porovnanie. Na hornom obrázku sú dáta oboch výstupov a na spodnom obrázku je ich relatívny rozdiel. Rozdiel je maximálne 10^{-6}, a keďže táto hodnota je veľmi malá, môžeme pokladať výstup za správny. Rozdiel je spôsobený inou architektúrou jednotlivých procesorov.

```
1.  x1 = h5read('cpu_16.h5','/p');
2.  x2 = h5read('mic_122.h5','/p');
3.  y1 = reshape(x1, 3, []);
4.  y2 = reshape(x2, 3, []);
5.  y2 = y2./max(y1(:));
6.  y1 = y1./max(y1(:));
7.  figure;
8.  subplot(2, 1, 1), plot(y1.', 'k--');
9.  hold on;
10. plot(y2.', 'r:');
11. subplot(2, 1, 2), plot((y1 - y2).');
```

Kód 5.5 Matlab skript pre porovnanie dvoch výstupov vo formáte HDF5 a zobrazenia ich rozdielu.



Obrázok 5.1 Výstup po spustení testovacieho Matlab skriptu.

Po tom ako sme zostavili základnú verziu, ktorú je možné úspešne skompilovať a spustiť natívne na karte Xeon Phi, a zároveň je jej výpočet korektný, môžeme pristúpiť ku analýze jej profilu.

5.2 Analýza základnej verzie

5.2.1 Nástroje pre analýzu a ich použitie

Pri optimalizovaní budeme používať hlavne dva nástroje, a to optimalizačné reporty generované kompilátorom a VTune profiler pre vyhľadanie najvyťaženejších miest v aplikácii.

Optimalizačné reporty

Optimalizačné reporty sú generované kompilátorom. Nechávame zapnutú najvyššiu úroveň optimalizácii generovaných kompilátorom (kompilačný parameter „*-O3*“). Výpis optimalizačných reportov zapneme pridaním parametrov „*-qopt-report-phase=loop,vec,openmp -qopt-report5*“ pre kompiláciu. Všimnime si, že tieto parametre sú odlišné pre nami použitý Intel kompilátor vo verzii *2015* oproti jeho predchádzajúcim verziám. Takto vygenerovaný report bude obsahovať informácie o slučkách, vektorizácii a paralelizácii v jednotlivých metódach programu.

VTune profiler pre vyhľadanie najvyťaženejších miest v aplikácii

Mali sme k dispozícii iba verziu *2013*, ktorá nemá úplnú podporu pre Xeon Phi (napríklad nepodporuje analýzu OpenMP paralelizácie, call stack, atď). Obmedzili sme sa len na analýzu vytaženosti jednotlivých miest aplikácie. Aj tak sme narazili na niekoľko problémov – je vidieť, že Anselm nie je plne pripravený pre vývoj na Xeon Phi. Napriek tomu sa nám podarilo tieto problémy vyriešiť, prípadne obísť.

Na začiatku sa objavil problém s chýbajúcou podporou (služby, kolektor dát) pre VTune na samotnej karte Xeon Phi. Po zistení čo je vlastne problémom a nájdení jeho riešenia sa nám podarilo s pomocou technickej podpory všetky chýbajúce súčasti na kartu nainštalovať a zber dát z karty začal fungovať. Napriek tomu sa stále vyskytujú prípady, ked tieto služby na karte spadnú a zatiaľ ako jediné riešenie funguje reštart celého výpočtového uzla.

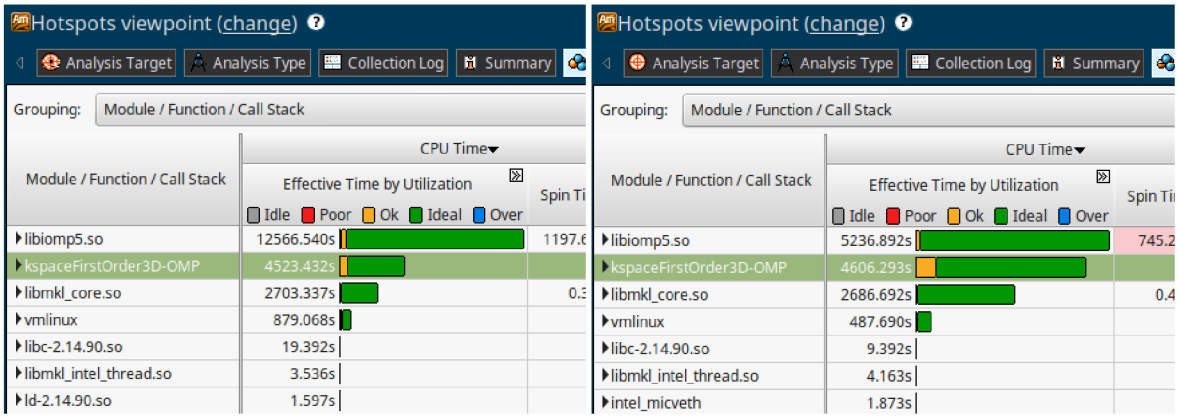
Ďalším problémom, ktorého riešenie zabralo veľa času, bolo padanie samotnej aplikácie VTune na segmentačnej chybe pri finalizácii zozbieraných výsledkov. Toto sa stáva pri použití verzie s grafickým rozhraním, rovnako ako pri použití verzie pracujúcej v príkazovom riadku (pri profilovaní bežnej verzie, sme na tento problém nikdy nenarazili). Problém sa nepodarilo vyriešiť ani s pomocou technickej podpory. Vyriešené to bude až s nasadením novej verzie VTune na superpočítač Anselm. Nám sa tento problém nakoniec podarilo obísť tak, že pre zozbieranie dát z karty Xeon Phi použijeme VTune verziu pre príkazový riadok, ale zakážeme finalizáciu výsledkov pomocou parametru „*-no-auto-finalize*“ ako je vidieť v nasledujúcom zobrazenom kóde 5.6:

```
./amplxe-cl -collect knc-hotspots -no-auto-finalize -- ssh mic0 "
  export LD_LIBRARY_PATH=/apps/intel/composer_xe_2015.2.164/compiler/lib/mic/:${LD_LIBRARY_PATH};
  export LD_LIBRARY_PATH=/apps/intel/composer_xe_2015.2.164/mkl/lib/mic/:${LD_LIBRARY_PATH};
  export KMP_AFFINITY=compact;
  export OMP_NUM_THREADS=240
  export MKL_NUM_THREADS=240
  ./kspaceFirstOrder3D-OMP -i /home/DIP/Data/input_data_240_240_240_het_linear_nocomp.h5
                           -o /home/DIP/output.h5"
```

Kód 5.6 Spustenie aplikácie Intel VTune.

Takto získané dáta, spolu s použitou binárkou a použitými knižnicami, nakopírujeme na lokálny počítač s nainštalovaným VTune vo verzii *2015* (použijeme grafickú verziu). Tu si vytvoríme nový projekt, ktorý nastavíme tak, aby používal skopírovanú binárku a knižnice. Potom importujeme skopírované dáta vygenerované na karte Xeon Phi a prevedie sa ich finalizácia. Dáta sú graficky zobrazené ako na obrázkoch nižšie.

Posledným väčším problémom, na ktorý sme narazili je funkcionalita samotného VTune pri použití s Xeon Phi. Pri experimentovaní sme zistili, že musíme nechať jedno jadro voľné pre réžiu, ktorú si vezme VTune, inak dôjde k vysokému nárastu procesorového času pre modul knižnice OpenMP. Je potom použitých príliš veľa vlákien a dochádza k väčšiemu čakaniu vlákien na procesorový čas. Celá situácia je zobrazená na obrázku 5.2, kde vidíme analýzu vyťaženosti jednotlivých modulov, s použitím *240* vlákien vľavo a s použitím *236* vlákien vpravo. V tomto prípade je rozdiel takmer dvojnásobný.



Obrázok 5.2 Aplikácia Intel VTune. Zobrazenie analýzy pre 240 a 236 vlákien.

5.2.2 Profil aplikácie

Po prvotných skúšobných testoch profilovania sme sa rozhodli, že celé naše snaženie o optimalizovanie aplikácie k-Wave budeme profilovať na dvoch sadách vstupných dát. Sú nimi dáta s veľkosťami *240x240x240* a *256x256x256*. Prvé dáta zastupujú pravdepodobne "ideálnu" veľkosť dát pre maximum *240* vlákien, ktoré môžu bežať na karte Xeon Phi, pretože celková veľkosť dát je násobkom počtu týchto vlákien. Druhá vzorka dát potom je braná ako "nie ideálna". Zobrazené dáta boli získané pri použití *236* vlákien, ale samotné experimenty boli potom robené pri použití plných *240* vlákien.

Vzorka dát o veľkosti *240x240x240*

Hotspots viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down

Grouping: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Overhead Time	Instructions Retired	CPI Rate
libiomp5.so	3194.116s	225.200s	218.146s	698,420,000,000	5.479
libmkl_core.so	2049.791s	0.314s	0s	488,640,000,000	4.414
kspaceFirstOrder3D-OMP	1266.131s	0s	0s	271,130,000,000	4.913
vmlinux	290.856s	0s	0s	48,600,000,000	6.296
libc-2.14.90.so	4.829s	0s	0s	820,000,000	6.195
libmkl_intel_thread.so	4.525s	0s	0s	40,000,000	119.000
ld-2.14.90.so	1.740s	0s	0s	40,000,000	45.750

Obrázok 5.3 Celkový pohľad na všetku moduly pre dáta 240x240x240.

Najvyťaženejším modulom je *libiomp5.so* predstavujúci knižnicu OpenMP. Je to spôsobené veľkou réžiou plánovania na Xeon Phi, keďže je použitých veľmi veľa vlákien. Na tento modul vlastne vplývajú všetky ďalšie modely používajú paralelizáciu. Modul *libmkl_core.so* predstavuje všetky výpočty prevádzané knižnicou MKL a nebudeme ho vôbec upravovať, lebo knižnica je optimalizovaná priamo firmou Intel. Knižnica MKL je použitá len na prevádzanie Fourierových transformácií a na iné výpočty sa nepoužíva. Zameriame sa na modul *kspaceFirstOrder3D-OMP*, ktorý reprezentuje aplikáciu k-Wave bez Fouriérových transformácií. Jednoduché je odvodenie pomeru času stráveného výpočtami FFT transformácií k ostatným výpočtom, a to je približne *62 %* ku *38 %*. Teoreticky je teda maximálne dosiahnuteľné zrýchlenie menšie ako *38 %* oproti základnej verzii.

Hotspots viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

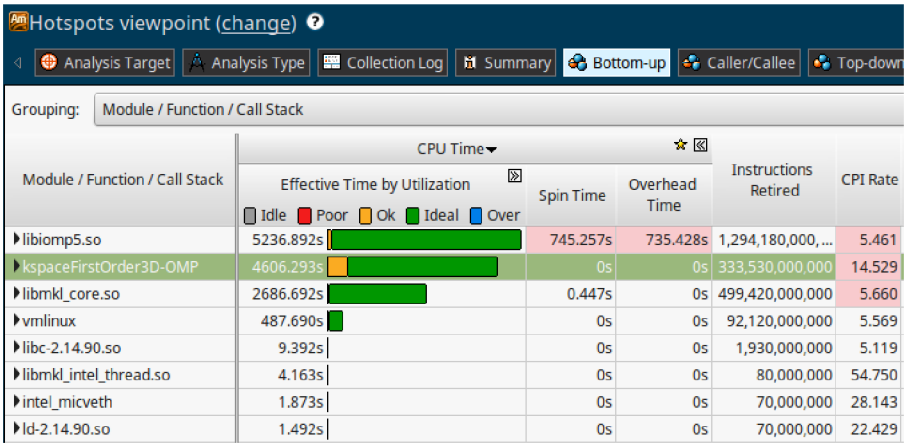
Grouping: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Overhead Time	Instructions Retired	CPI Rate
libiomp5.so	3194.116s	225.200s	218.146s	698,420,000,000	5.479
libmkl_core.so	2049.791s	0.314s	0s	488,640,000,000	4.414
kspaceFirstOrder3D-OMP	1266.131s	0s	0s	271,130,000,000	4.913
TKSpaceFirstOrder3DSolver::Calculate_SumRho_BonA_SumDuompparallel@1650	571.179s	0s	0s	109,930,000,000	5.466
TKSpaceFirstOrder3DSolver::Sum_Subterms_nonlinearompparallel@1812	214.971s	0s	0s	57,830,000,000	3.911
TKSpaceFirstOrder3DSolver::Compute_rhoxyz_nonlinearompparallel@1287	104.848s	0s	0s	11,340,000,000	9.727
TKSpaceFirstOrder3DSolver::Compute_duxyzompparallel@1097	96.074s	0s	0s	30,850,000,000	3.276
TKSpaceFirstOrder3DSolver::Compute_ddx_kappa_fft_pompparallel@1011	93.432s	0s	0s	28,490,000,000	3.450
TKSpaceFirstOrder3DSolver::Compute_Absorb_nabla1_2	55.551s	0s	0s	17,950,000,000	3.256
Tuxyz_sgxyzMatrix::Compute_uz_sgz_normalize	35.447s	0s	0s	2,120,000,000	17.590
Tuxyz_sgxyzMatrix::Compute_uy_sgy_normalize	34.544s	0s	0s	2,210,000,000	16.443
Tuxyz_sgxyzMatrix::Compute_ux_sgx_normalize	33.279s	0s	0s	2,360,000,000	14.835

Obrázok 5.4 Pohľad na modul aplikácie k-Wave pre dáta 240x240x240.

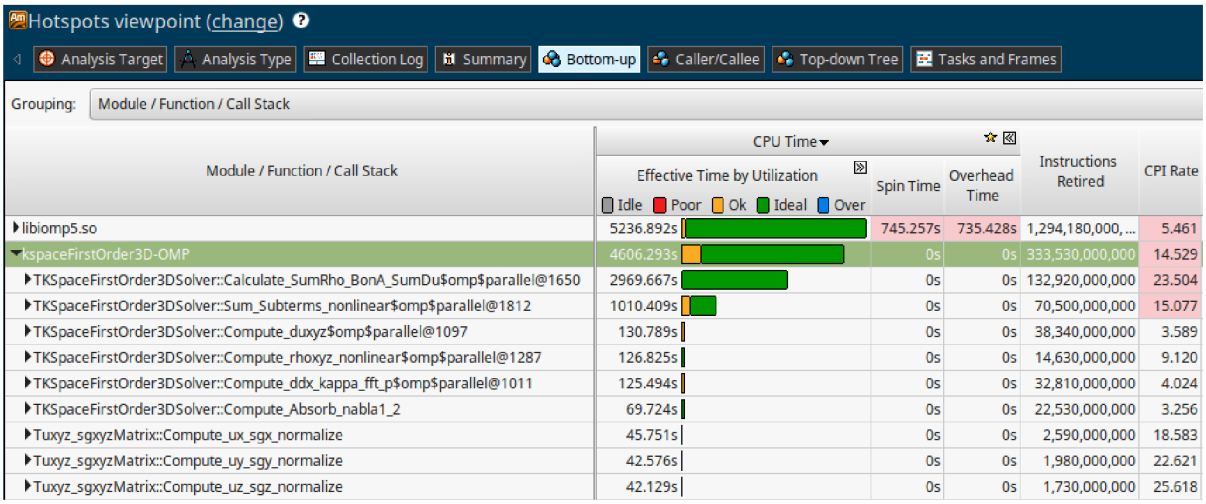
Na ďalšom obrázku 5.4 vidíme ako sa na celkovom spotrebovanom procesorovom čase podieľajú jednotlivé metódy aplikácie. Jednoznačne najviac času spotrebuje metóda *Calculate_SumRho_BonA_SumDu()* – málo cez *45 %*. Potom nasleduje metóda *Sum_subterms_nonlinear()* – spotrebuje takmer *17 %* času. Ostatné metódy sa podieľajú na celkovom čase približne *8 %* a menej. Z analýzy pre nás ideálnych dát vyplýva, že by sme sa mali zamerať na prvé dve menované metódy, ktorých optimalizácia môže mať signifikantný prínos k zrýchleniu simulácie.

Vzorka dát o veľkosti *256x256x256*



Obrázok 5.5 Celkový pohľad na všetky moduly pre dáta 256x256x256.

Keď sa pozrieme na druhú vzorku dát, vidíme ako sa situácia rapídne zmenila. Pomer času stráveného výpočtami FFT transformácií k ostatným výpočtom sa otočil, a tentokrát je približne *37 %* ku *63 %*. Naše optimalizačné snahy teda môžu dosiahnuť ešte lepšie výsledky ako u prvej vzorky dát.



Obrázok 5.6 Pohľad na modul aplikácie k-Wave pre dáta 256x256x256.

Pri pohľade na čas spotrebovaný jednotlivými metódami vidíme, že opäť sú na tom najhoršie metódy *Calculate_SumRho_BonA_SumDu()* a *Sum_subterms_nonlinear()*, tentokrát však s ešte väčším rozdielom. Prvá metóda spotrebuje až takmer *65 %* a druhá viac skoro *22 %* času. Ostatné metódy sa podieľajú maximálne tromi percentami každá. Naše snaženie teda sústredíme na metódy *Calculate_SumRho_BonA_SumDu()* a *Sum_subterms_nonlinear()*, ktoré vyšli ako najlepší kandidáti pre optimalizáciu v obidvoch prípadoch.

5.3 Optimalizácia najvyťaženejších častí kódu

V predchádzajúcej sekcii sme zistili, že jednoznačne najvyťaženejšími miestami aplikácie sú dve metódy: *Calculate_SumRho_BonA_SumDu()* a *Sum_subterms_nonlinear()*. Ďalšie metódy majú oproti nim minimálny vplyv na celkový čas výpočtu. V tejto sekcii sa budeme snažiť optimalizovať tieto dve identifikované metódy, a to pomocou rôznych techník.

5.3.1 Metóda Calculate_SumRho_BonA_SumDu()

Po prezretí kódu tejto metódy, vieme jasne určiť, že takmer celý výpočtový čas je strávený v slučke, ktorá obsahuje výpočty simulácie. Prevedie sa tu najviac operácii s plávajúcou desatinnou čiarkou (floating point operations) v jednoduchej presnosti (*32* bitov) naprieč celou aplikáciou.

```
1.  #pragma omp parallel
2.  {
3.    float * RHO_Temp_Data   = RHO_Temp.GetRawData();
4.    float * BonA_Temp_Data  = BonA_Temp.GetRawData();
5.    float * SumDU_Temp_Data = Sum_du.GetRawData();
6.
7.    #pragma omp for schedule (static) nowait
8.    for (size_t i = 0; i < RHO_Temp.GetTotalElementCount(); i++)
9.    {
10.     register const float rho_xyz_el = rhox_data[i] + rhoy_data[i] + rhoz_data[i];
11.     RHO_Temp_Data[i]   = rho_xyz_el;
12.     BonA_Temp_Data[i]  = ((BonA[i * BonA_shift] * (rho_xyz_el * rho_xyz_el))      \
13.                           / (2.0f * rho0_data[i * rho0_shift])) + rho_xyz_el;
14.     SumDU_Temp_Data[i] = rho0_data[i * rho0_shift]                                \
15.                           * (dux_data[i] + duy_data[i] + duz_data[i]);
16.   }
17. }// parallel
```

Kód 5.7 Základná verzia metódy Calculate_SumRho_BonA_SumDu().

Paralelizácia

V kóde 5.7 môžeme vidieť predmetnú slučku. Všimnime si, že je definovaná vo vnútri paralelného regiónu pomocou direktívy *#pragma omp parallel* a samotná slučka vykonaná paralelne skupinou vlákien pomocou direktívy *#pragma omp for*. Možnosť *schedule (static)* nám určuje, že počet iterácií slučky bude rozdelený medzi vlákna rovnomerne. Pre overenie, či bola slučka skutočne zparalelizovaná, sa pozrieme do reportu 5.1 z OpenMP optimalizácií, ktorý bol bol vygenerovaný pri kompilácii.

```
Begin optimization report for:
TKSpaceFirstOrder3DSolver::Calculate_SumRho_BonA_SumDu(TKSpaceFirstOrder3DSolver *, TRealMatrix &,
TRealMatrix &, TRealMatrix &)

    Report from: OpenMP optimizations [openmp]

OpenMP Construct at KspaceSolver/KspaceFirstOrder3DSolver.cpp(1565,3)
remark #16201: OpenMP DEFINED REGION WAS PARALLELIZED
```

Report 5.1 Výpis z OpenMP reportu po skompilovaní základnej verzie.

Vidíme, že kód bol úspešne zparalelizovaný – paralelizácia je jednou so základných a veľmi potrebných optimalizácií pre Xeon Phi. Z výkonovej analýzy vieme, že nedosahuje rýchlosti na bežnom procesore ani pri použití oveľa väčšieho počtu vlákien (v kapitole, kde sa budeme zaoberať experimentovaním, si ukážeme aj výkon bez použitia paralelizácie). Je teda nutné pokračovať v optimalizovaní.

Vektorizácia

Ďalším esenciálnou optimalizačnou technikou je vektorizácia. Pri Xeon Phi ma ešte oveľa väčší význam ako pri bežných procesoroch, keďže implementuje vektorové jednotky široké až *512* bitov a umožňuje nám tak paralelne spracovať až *16* operácii s jednoduchou presnosťou, ktoré sú použité v slučke. Kompilátor dokáže generovať vektorové inštrukcie automaticky pri použití prepínača *"-O2"* a vyššieho, a to práve vtedy, pokiaľ vie dokázať, že v našom kóde nie sú žiadne dátové závislosti. Pozrieme sa preto do reportu 5.2 na prevedené vektorizačné optimalizácie.

```
Begin optimization report for:
TKSpaceFirstOrder3DSolver::Calculate_SumRho_BonA_SumDu(TKSpaceFirstOrder3DSolver *, TRealMatrix &,
TRealMatrix &, TRealMatrix &

    Report from: Vector optimizations [vec]

LOOP BEGIN at KSpaceSolver/KSpaceFirstOrder3DSolver.cpp(1573,5)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed ANTI dependence between rhox_data line 1574 and
SumDU_Temp_Data line 1578
   remark #15346: vector dependence: assumed FLOW dependence between SumDU_Temp_Data line 1578 and
rhox_data line 1574
LOOP END
```

Report 5.2 Výpis z vektorizačného reportu po skompilovaní základnej verzie.

Ako môžeme vidieť, kompilátor nebol schopný s určitosťou zistiť, že slučka neobsahuje dátové závislosti – preto túto závislosť implicitne predpokladal a slučku nezvektorizoval. Kompilátor takto zaručuje korektnosť výsledného kódu. Celé je to pravdepodobne spôsobené zložitosťou slučky a vysokým počtom dátových ukazovateľov v nej. Keďže my ako vývojári vieme, že tu žiadne dátové závislosti nie sú, tak máme nasledujúce možnosti:

1. Inštruovať kompilátor aby ignoroval predpokladané vektorové závislosti pomocou #pragma ivdep. Kompilátor ale stále nebude vektorizovať slučky, u ktorých vie dokázať dátové závislosti a vygenerovaný kód bude v takom prípade stále korektný.
2. Vynútiť si vektorizáciu pomocou direktívy *#pragma omp simd*, dostupnú v OpenMP od verzie *4.0*. Keďže vektorizácia je vynútená, kompilátor v tomto prípade nekontroluje dátové závislosti vôbec. Je tak čisto na vývojárovi aby zaistil, že tam žiadne nie sú. V opačnom prípade bude vygenerovaný kód chybný.
3. Kód zvektorizovať manuálne pomocou vektorových inštrukcií procesoru Intel Xeon Phi v ich intrinsic variante.

Pokiaľ použijeme ktorúkoľvek z prvých dvoch techník, tak výsledok bude v našom prípade úplne rovnaký. Pri použití *1.* techniky, bude kompilátor ignorovať predpokladané závislosti a kód zvektorizuje, pretože žiadne dokázateľné závislosti nenájde, pretože tam nie sú. Pri použití *2.* techniky kód zvektorizuje vynútene vždy. Je zaujímavé, že takýto kód bude po prekompilovaní rovnako funkčný aj na bežných procesoroch. Kompilátor pri tom použije dostupné vektorové inštrukcie cieľovej architektúry. Narozdiel of varianty *3.*, ktorá nebude prenosná mimo procesory podporujúce použité inštrukcie – momentálne iba súčasné Xeon Phi karty (je možné, že nebude ani podpora v nasledujúcich generáciách karty). V ďalšom postupe budeme teda používať variantu *2.* A teda použitie *#pragma omp simd*, ktorej kód 5.8 a report 5.3 z vektorizácie sú zobrazené na ďalších riadkoch.

```
1.  #pragma omp simd
2.  #pragma omp for schedule (static) nowait
3.  for (size_t i = 0; i < RHO_Temp.GetTotalElementCount(); i++)
4.  {
5.    register const float rho_xyz_el = rhox_data[i] + rhoy_data[i] + rhoz_data[i];
6.    RHO_Temp_Data[i]    = rho_xyz_el;
7.    BonA_Temp_Data[i]   = ((BonA[i * BonA_shift] * (rho_xyz_el * rho_xyz_el))     \
8.                          / (2.0f * rho0_data[i * rho0_shift])) + rho_xyz_el;
9.    SumDU_Temp_Data[i] = rho0_data[i * rho0_shift]                                \
10.                         * (dux_data[i] + duy_data[i] + duz_data[i]);
11. }
```

Kód 5.8 Slučka po vektorizácii.

```
LOOP BEGIN at KSpaceSolver/KSpaceFirstOrder3DSolver.cpp(1574,5)
<Peeled>
   remark #15389: vectorization support: reference SumDU_Temp_Data has unaligned access   [ (1579,7) ]
   remark #15389: vectorization support: reference rhox_data has unaligned access   [ (1575,71) ]
   … repeated unaligned access for all data references in the loop, except when gather is used …
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15301: PEEL LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at KspaceSolver/KSpaceFirstOrder3DSolver.cpp(1574,5)
   remark #15389: vectorization support: reference SumDU_Temp_Data has aligned access   [ (1579,7) ]
   remark #15389: vectorization support: reference rhox_data has unaligned access   [ (1575,71) ]
   … repeated unaligned access for all data references in the loop, except when gather is used …
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15415: vectorization support: gather was generated for the variable BonA:  strided by non-
constant value   [ (1578,30) ]
   … repeated gather generation also for two occurrences of rho0_data …
   remark #15300: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 6
   remark #15451: unmasked unaligned unit stride stores: 2
   remark #15460: masked strided loads: 3
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 76
   remark #15477: vector loop cost: 8.620
   remark #15478: estimated potential speedup: 7.720
   remark #15479: lightweight vector operations: 32
   remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at KSpaceSolver/KSpaceFirstOrder3DSolver.cpp(1574,5)
<Remainder>
   remark #15389: vectorization support: reference SumDU_Temp_Data has aligned access [ (1579,7) ]
   remark #15389: vectorization support: reference rhox_data has unaligned access   [ (1575,71) ]
   … repeated unaligned access for all data references in the loop, except when gather is used …
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

Report 5.3 Výpis z vektorizačného reportu po vektorizácii slučky.

Z uvedeného reportu vyplýva, že slučka bola úspešne zvektorizovaná. Skutočne, keď sa pozrieme do vygenerovaného kódu v assemblery, nachádzame tam použité vektorové inštrukcie ako napríklad *VLOADUNPACKLD*, *VPACKSTORELD*, *VADDPS*, *VMULPS*, *VGATHERDPSL*. Kompilátor odhadol zrýchlenie slučky na *7.720*. Skutočné zrýchlenie môžeme vidieť v kapitole experimentov.

Na poslednom vektorizačnom reporte si všimnime, že kompilátor pre vektorové spracovanie našej slučky vygeneroval až tri slučky. Keďže v čase kompilácie nie je známy počet iterácii našej slučky, kompilátor musí vygenerovať minimálne dve slučky. Vysvetlíme si to na príklade: povedzme, že slučka bude mať *1028* iterácií a náš procesor Xeon Phi dokáže spracovať vektorovo *16* operácií. Plne paralelne teda môže byť spracovaných len prvých *1024* iterácií, čo vyústi celkovo do *64* vektorizovaných iterácií. Tieto budú počítané prvou vygenerovanou slučkou. Zvyšné *4* iterácie (*1028 mod 16*), budú spracované druhou vygenerovanou slučkou, takzvanou REMAINDER LOOP. Záleží pritom na kompilátore, či aj túto slučku zvektorizuje (už len čiastočne) alebo bude počítaná skalárne.

Zarovnanie dát v pamäti

V našom prípade bola vygenerovaná ešte takzvaná PEEL LOOP, kompilátor totižto spravil ďalšiu optimalizáciu. Snaží sa vygenerovať inštrukcie načítania a zápisu pracujúce so zarovnanou pamäťou. Tie sú totiž efektívnejšie ako tie pracujúce s nezarovnanou pamäťou. V dobe kompilácie nie je kompilátoru známe, či sú dáta na ktoré je odkazované zarovnané. Rieši to teda tak, že postupne „odkrajuje“ iterácie zo začiatku slučky, až pokiaľ nenatrafí na zarovnané miesto. Následná hlavná slučka potom začína spracovanie na zarovnanej pamäti. Z reportu nám vychádza, že kompilátor takto dokázal zoptimalizovať iba referenciu *SumDU_Temp_Data* a zvyšných osem referencií stále používa nezarovnané inštrukcie. Keďže sme alokovali našu pamäť pomocou funkcie *_mm_malloc()* so zarovnaním na *64* bytov a všetky použité referencie odkazujú práve na túto zarovnanú pamäť, môžeme kompilátor inštruovať aby použil zarovnané inštrukcie pre všetky referencie pomocou *#pragma vector aligned*. Táto direktíva je súčasťou Intel kompilátora. Táto implementácia sa však nakoniec ukázala ako nesprávna – problém je popísaný neskôr v tejto istej podkapitole.

```
1.  #pragma vector aligned
2.  #pragma omp simd
3.  #pragma omp for schedule (static) nowait
4.  for (size_t i = 0; i < RHO_Temp.GetTotalElementCount(); i++)
5.  {
6.    register const float rho_xyz_el = rhox_data[i] + rhoy_data[i] + rhoz_data[i];
7.    RHO_Temp_Data[i]   = rho_xyz_el;
8.    BonA_Temp_Data[i]  = ((BonA[i * BonA_shift] * (rho_xyz_el * rho_xyz_el))    \
9.                          / (2.0f * rho0_data[i * rho0_shift])) + rho_xyz_el;
10.   SumDU_Temp_Data[i] = rho0_data[i * rho0_shift]                              \
11.                         * (dux_data[i] + duy_data[i] + duz_data[i]);
12. }
```

Kód 5.9 Slučka po použití zarovnaných dát v pamäti.

Nasledujúci report 5.4 potvrdzuje použitie zarovnaných inštrukcií. Zároveň nebola vygenerovaná PEEL slučka, pretože po tom ako máme všetky referencie zarovnané, nie je už viac potrebná. V kóde assembleru boli skutočne nahradené nezarovnané inštrukcie načítania a zápisu *VLOADUNPACKLD* a *VPACKSTORELD* za zarovnané *VMOVAPS* inštrukcie. Kompilátor po tejto optimalizácii odhadol zrýchlenie slučky na 9.320. Skutočné zrýchlenie je zmerané v kapitole experimentov.

```
LOOP BEGIN at KSpaceSolver/KSpaceFirstOrder3DSolver.cpp(1574,5)
   remark #15388: vectorization support: reference rhox_data has aligned access   [ (1575,71) ]
   … repeated aligned access for all data references in the loop, except when gather is used …
   remark #15415: vectorization support: gather was generated for the variable BonA:  strided by non-
constant value   [ (1578,30) ]
   … repeated gather generation also for two occurrences of rho0_data …
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15460: masked strided loads: 3
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 76
   remark #15477: vector loop cost: 7.620
   remark #15478: estimated potential speedup: 9.320
   remark #15479: lightweight vector operations: 32
   remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at KSpaceSolver/KSpaceFirstOrder3DSolver.cpp(1574,5)
<Remainder>
   remark #15388: vectorization support: reference rhox_data has aligned access   [ (1575,71) ]
   … repeated aligned access for all data references in the loop, except when gather is used …
   remark #15301: REMAINDER LOOP WAS VECTORIZED
LOOP END
```

Report 5.4 Výpis z vektorizačného reportu po použití zarovnaných dát v pamäti.

Pri experimentovaní s touto verziou sa však ukázal problém. Na dátach s veľkosťou násobku počtu použitých vlákien (napríklad dáta veľkosti *240x240x240* pri *240* vláknach) aplikácia fungovala bez problémov. Ale v momente, keď dáta nemali takúto veľkosť (napríklad dáta veľkosti *256x256x256* pri *240* vláknach) aplikácia začala padať so segmentačnou chybou. Po dosť dlhom pátraní sme zistili, že je to spôsobené súčasným použitím paralelizácie pomocou OpenMP a vynúteným generovaním zarovnaných inštrukcií pomocou *#pragma vector aligned*. Ako programátori totiž musíme zaručiť zarovnanie pre každé jedno vlákno vygenerované pomocou OpenMP. Ponúkajú sa nasledovné riešenia:

1. Použitie *__assume_aligned()* pre všetky referencie na zarovnanú pamäť namiesto direktívy *#pragma vector aligned*. Kompilátor tak bude opäť generovať PEEL slučku, ktorá sa v prípade nezarovnania v niektorom vlákne o toto zarovnanie postará.
2. Refaktorizácia kódu tak, aby bolo zarovnanie vždy zaručené. Napríklad pre nami použitý *schedule (static)* obmedzením počtu paralelne spracovaných iterácií na vhodný násobok, v našom prípade na násobok počtu vlákien a počtu vektorovo spracovávaných elementov (*omp_get_num_threads() * 16*). Zvyšné iterácie potom môžeme dopočítať sériovo, alebo ešte vhodnejšie opäť paralelne (a samozrejme vektorovo).
3. V neskoršej fáze tohto projektu sme prišli ešte na jednu možnosť. Direktíva *#pragma omp simd* dovoľuje použiť dodatok aligned, kde umožňuje špecifikovať všetky zarovnané referencie. Toto riešenie má rovnaké dôsledky ako riešenie prvé a je kratšie, tak ho uvádzame namiesto prvého.

```
#pragma omp parallel
{
  #pragma omp master
  {
    int num_threads = omp_get_num_threads();
    TotalElementCount1 = ((TotalElementCount / num_threads) / 16) * num_threads * 16;
  }
}// parallel


#pragma omp parallel
{
  // compute MAIN loop
  #pragma vector aligned
  #pragma omp simd
  #pragma omp for schedule (static) nowait
  for (size_t i = 0; i < TotalElementCount1; i++)
  {
    Register const float rho_xyz_el = rhox_data[i] + rhoy_data[i] + rhoz_data[i];
    RHO_Temp_Data[i]   = rho_xyz_el;
    BonA_Temp_Data[i]  = ((BonA[i * BonA_shift] * (rho_xyz_el * rho_xyz_el))     \
                         / (2.0f * rho0_data[i * rho0_shift])) + rho_xyz_el;
    SumDU_Temp_Data[i] = rho0_data[i * rho0_shift]                               \
                         * (dux_data[i] + duy_data[i] + duz_data[i]);
  }

  // compute REMAINDER loop - last elements are processed but unaligned access is used
  #pragma omp simd
  #pragma omp for schedule (static) nowait
  for (size_t i = TotalElementCount1; i < TotalElementCount; i++)
  {
    register const float rho_xyz_el = rhox_data[i] + rhoy_data[i] + rhoz_data[i];
    RHO_Temp_Data[i]   = rho_xyz_el;
    BonA_Temp_Data[i]  = ((BonA[i * BonA_shift] * (rho_xyz_el * rho_xyz_el))     \
                         / (2.0f * rho0_data[i * rho0_shift])) + rho_xyz_el;
    SumDU_Temp_Data[i] = rho0_data[i * rho0_shift]                               \
                         * (dux_data[i] + duy_data[i] + duz_data[i]);
  }
}// parallel
```

Kód 5.10 Časť metódy upravená podľa 2. variantu.

```
#pragma omp simd aligned (rhox_data:64, rhoy_data:64, rhoz_data:64, rho0_data:64,    \
                      BonA:64, dux_data:64, duy_data:64, duz_data:64,            \
                      RHO_Temp_Data:64, BonA_Temp_Data:64, SumDU_Temp_Data:64)
#pragma omp for schedule (static)
for (size_t i = 0; i < RHO_Temp.GetTotalElementCount(); i++)
{
  register const float rho_xyz_el = rhox_data[i] + rhoy_data[i] + rhoz_data[i];
  RHO_Temp_Data[i]   = rho_xyz_el;
  BonA_Temp_Data[i]  = ((BonA[i * BonA_shift] * (rho_xyz_el * rho_xyz_el))       \
                       / (2.0f * rho0_data[i * rho0_shift])) + rho_xyz_el;
  SumDU_Temp_Data[i] = rho0_data[i * rho0_shift]                                 \
                       * (dux_data[i] + duy_data[i] + duz_data[i]);
}
```

Kód 5.11 Časť metódy upravená podľa 3. variantu.

Všetky varianty sme naimplementovali a overili sme, že skutočne problém vyriešili. Ďalej sme sa rozhodli pre používanie tretieho variantu. Zrýchlenie dosiahnuté použitím zarovnania je pomerne diskutabilné. Kapitola experimenty totiž ukazuje, že niekedy dochádza naopak k zníženiu výkonnosti. Vo všeobecnosti takmer všetky zdroje odporúčajú zarovnanie používať.

Inštrukcie Gather/Scatter

Pri inšpekcii assemblerovského kódu ale aj napríklad v predchádzajúcich reportoch je možné vidieť, že kompilátor generuje inštrukcie typu gather. Tieto inštrukcie načítajú dáta z rôznych pozícií v pamäti do jedného registra, aby mohli byť následne spracované vektorovo. Inštrukcia scatter funguje presne opačne a teda tieto hodnoty z jedného registru uloží na viacero rôznych pozícií v pamäti. Inštrukcie scatter v tomto kóde vygenerované nie sú, ale vzťahujú sa na ne rovnaké vlastnosti, a preto sú pre kompletnosť zmienené. Ich problémom je vysoká latencia pri spracovaní väčšieho množstva dát a môžu tak mať dopad na konečný výkon aplikácie. Naopak, jednou z výhod je napríklad to, že na procesoroch Xeon Phi vie je kompilátor s ich použitím zvektorizovať viacero typov slučiek. Inštrukcie tohto typu sú vygenerované napríklad vtedy, pokiaľ pristupujeme k hodnotám v pamäti bez jednotkového rozostupu. Presne toto si myslí kompilátor aj práve o kóde našej slučky, kde je gather vygenerovaný tri krát. Raz pre načítanie z *BonA[i * BonA_shift]* a dvakrát pre načítanie z *rho0_data[i * rho0_shift]*. Keď sa však pozrieme na kód celej funkcie, vidíme, že premenné *BonA_shift* a *rho0_shift* sú podmienečne nastavené na pevno pred vykonaním slučky a nadobúdajú jedine hodnoty *0* a *1*. Je teda evidentné, že k pamäti je vždy pristupované na *BonA[0]* alebo *BonA[i]*, kde v prvom prípade pristupujeme na to isté miesto opakovane a v druhom prípade máme zachovaný jednotkový rozostup.

Preto, aby sme sa zbavili inštrukcie gather, musíme refaktorovať kód tak, že celú slučku naklonujeme niekoľko krát a upravíme jej varianty aby nepoužívali pre prístup do pamäti premenné *BonA_shift* a *rho0_shift*, ale aby sa vykonali podmienečne na základe týchto premenných. Negatívom tohto prístupu je mnoho krát sa opakujúci kód, čo zhorší jeho čitateľnosť.

```
LOOP BEGIN at KSpaceSolver/KSpaceFirstOrder3DSolver.cpp(1674,9)
   remark #15388: vectorization support: reference rhox_data has aligned access   [ (1676,75) ]
   remark #15388: vectorization support: reference rhoy_data has aligned access   [ (1676,75) ]
   remark #15388: vectorization support: reference rhoz_data has aligned access   [ (1676,75) ]
   remark #15388: vectorization support: reference RHO_Temp_Data has aligned access   [ (1678,11) ]
   remark #15388: vectorization support: reference BonA_Temp_Data has aligned access   [ (1679,11) ]
   remark #15388: vectorization support: reference BonA has aligned access   [ (1679,11) ]
   remark #15388: vectorization support: reference SumDU_Temp_Data has aligned access   [ (1680,11) ]
   remark #15388: vectorization support: reference dux_data has aligned access   [ (1680,11) ]
   remark #15388: vectorization support: reference duy_data has aligned access   [ (1680,11) ]
   remark #15388: vectorization support: reference duz_data has aligned access   [ (1680,11) ]
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 72
   remark #15477: vector loop cost: 3.870
   remark #15478: estimated potential speedup: 16.410
   remark #15479: lightweight vector operations: 32
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

Report 5.5 Výpis z vektorizačného reportu po odstánení inštrukcií typu gather.

O tom, že táto zmena skutočne pomohla odstrániť použitie inštrukcie gather, sa môžeme presvedčiť priamo v kóde vygenerovaného assembleru, alebo aj v reporte 5.5 po prekompilovaní upraveného

kódu. Časť, ktorú máme zobrazenú, obsahuje report celkovo štyri krát – jedenkrát pre každú vytvorenú slučku. Zároveň je možné si všimnúť, že zarovnané inštrukcie boli vygenerované aj pre *BonA* a *rho0_data*. Kompilátor však môže stále generovať gather pre REMAINDER slučku, napríklad aby ju mohol efektívne zvektorizovať. To pre nás nie je problém, pretože v sa v nej udeje oveľa menšie množstvo výpočtov oproti hlavnej slučke, kde sa nám ho podarilo odstrániť. Očakávame, že táto optimalizácia prinesie celkom zaujímavé zrýchlenie, nakoľko nielenže boli odstránené inštrukcie gather s pomalšou dobou odozvy, ale zároveň boli použité aj inštrukcie načítania zo zarovnanej pamäti. Rovnako aj kompilátor odhadol zrýchlenie na dobrých *16.410* pre konkrétne zobrazenú slučku (zrýchlenie pre tri zostávajúce slučky odhadol na menej: *15.450*, *15.930* a *14.980*). To, že zrýchlenie je väčšie ako *16* je skutočne možné, pretože kompilátor mohol spraviť aj niektoré ďalšie optimalizácie ako napríklad prednačítanie dát. Konkrétne experimentálne výsledky sú zobrazené v nasledujúcej kapitole.

Prúdové ukladanie do pamäti

Ako ďalšiu optimalizáciu, ktorá vyzerá byť veľmi vhodná práve v tomto prípade, je použitie takzvaného prúdového ukladania dát do pamäti (*streaming stores*) podporovaného inštrukčnou sadou procesoru Xeon Phi. Slučka ktorú práve analyzujeme a optimalizujeme, zapisuje spracované dáta do zarovnanej pamäti priamo za sebou, bez nejakých medzier medzi nimi (prúdovo) a rovnako tieto dáta nie sú opätovne používané. Tento prípad máme v našej slučke dokonca tri krát (*RHO_Temp_Data[i]*, *BonA_Temp_Data[i]*, *SumDU_Temp_Data[i]*). Klasické inštrukcie ukladania do pamäti využívajú rýchlu pamäť cache, naopak prúdové inštrukcie dovoľujú cache obísť a tým zvýšiť výkon. Samozrejme, že pokiaľ sú uložené dáta ihneď používané, tak je použitie prúdového ukladanie nepriateľné – dáta vôbec nie sú v cache a dochádza tak k degradácii výkonu. Pre generovanie týchto inštrukcií je potrebné pridať do zdrojového kódu 5.12 ďalšiu direktívu a to *#pragma vector nontemporal*. Táto direktíva je súčasťou Intel kompilátora. Vygenerovaný report 5.6 potvrdzuje, že boli vygenerované inštrukcie prúdového ukladania dát pre všetky tri pamäťové zápisy.

```
1.  #pragma vector nontemporal
2.  #pragma omp simd aligned (rhox_data:64, rhoy_data:64, rhoz_data:64, rho0_data:64,    \
3.                            BonA:64, dux_data:64, duy_data:64, duz_data:64,           \
4.                            RHO_Temp_Data:64, BonA_Temp_Data:64, SumDU_Temp_Data:64)
5.  #pragma omp for schedule (static)
6.  for (size_t i = 0; i < RHO_Temp.GetTotalElementCount(); i++)
7.  {
8.    register const float rho_xyz_el = rhox_data[i] + rhoy_data[i] + rhoz_data[i];
9.    RHO_Temp_Data[i]   = rho_xyz_el;
10.   BonA_Temp_Data[i]  = ((BonA[i * BonA_shift] * (rho_xyz_el * rho_xyz_el))       \
11.                         / (2.0f * rho0_data[i * rho0_shift])) + rho_xyz_el;
12.   SumDU_Temp_Data[i] = rho0_data[i * rho0_shift]                                 \
13.                         * (dux_data[i] + duy_data[i] + duz_data[i]);
14. }
```

Kód 5.12 Slučka po použití prúdového ukladania dát do pamäti.

```
LOOP BEGIN at KSpaceSolver/KSpaceFirstOrder3DSolver.cpp(1695,9)
   remark #15388: vectorization support: reference rhox_data has aligned access   [
KSpaceSolver/KSpaceFirstOrder3DSolver.cpp(1697,75) ]
   … repeated aligned access for all data references in the loop …
   remark #15412: vectorization support: streaming store was generated for RHO_Temp_Data [(1699,11)]
   remark #15412: vectorization support: streaming store was generated for BonA_Temp_Data [(1700,11)]
   remark #15412: vectorization support: streaming store was generated for SumDU_Temp_Data [(1701,11)]
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15467: unmasked aligned streaming stores: 3
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 72
   remark #15477: vector loop cost: 3.870
   remark #15478: estimated potential speedup: 16.410
   remark #15479: lightweight vector operations: 32
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

Report 5.6 Výpis z vektorizačného reportu po použití prúdového ukladania dát do pamäti.

5.3.2 Metóda Sum_subterms_nonlinear()

Druhá metóda, ktorú budeme optimalizovať, je veľmi podobná tej prvej. Rozdiely sú iba v použitých operáciách, inak je štruktúrou rovnaká ako aj prvá optimalizovaná metóda. Tiež obsahuje jednu slučku zodpovednú za takmer všetok spotrebovaný procesorový čas. Je rovnako zparalelizovaná pomocou OpenMP a pri pohľade do vektorizačného reportu môžeme povedať, že nie je ani zvektorizovaná.

```
1.  #pragma omp parallel
2.  {
3.      const float * BonA_data = BonA_temp.GetRawData();
4.      float * p_data  = Get_p().GetRawData();
5.
6.      #pragma omp for schedule (static)
7.      for (size_t i = 0; i < TotalElementCount; i++)
8.      {
9.          p_data[i] = (c2_data[i * c2_shift]) * (BonA_data[i] + (Divider          \
10.                     * ((Absorb_tau_data[i] * tau_data[i * tau_eta_shift])       \
11.                     - (Absorb_eta_data[i] * eta_data[i * tau_eta_shift])));
12.     }
13. }// parallel
```

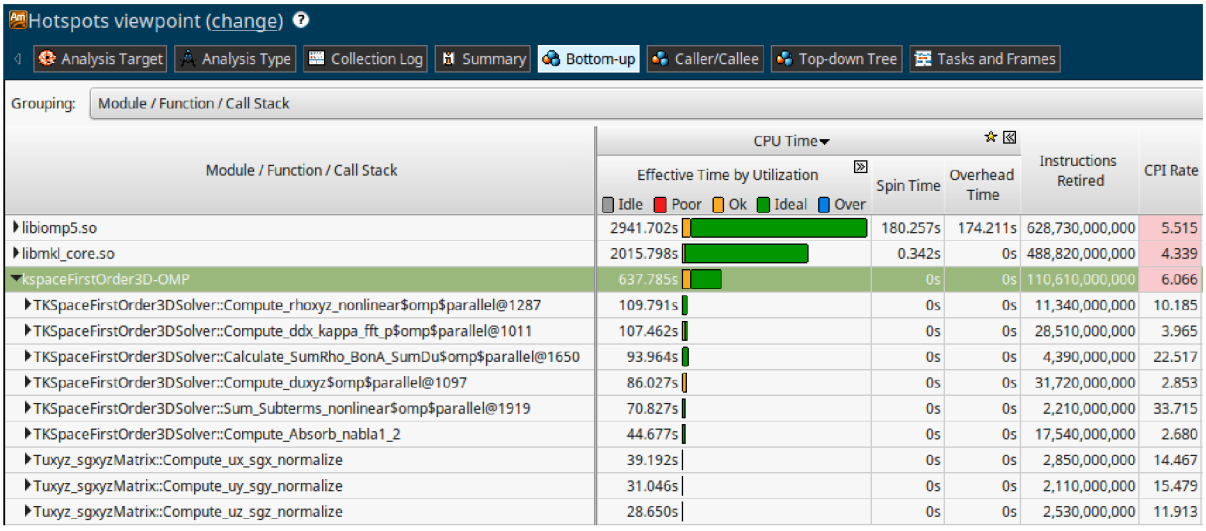
Kód 5.13 Základná verzia metódy Sum_subterms_nonlinear().

Preto prevedieme všetky optimalizačné techniky rovnako ako pri predchádzajúcej metóde. Nebudeme už však prikladať žiadne výpisy z kompilácie, pretože by vlastne boli skoro úplne rovnaké. Rozdielom v použitým operáciach je napríklad len jediný zápis do pamäti. Takže nemôžeme očakávať také zrýchlenie použitím prúdového ukladania do pamäti ako pri troch zápisoch v prípade prvej metódy. Pri tejto metóde je však možné použitie ďalšieho typu optimalizácie, ktorý vyplýva z použitých operácií, a ktorý preto nebolo možné použiť pri predchádzajúcej metóde. Je to použitie

inštrukcie spojeného násobenia a sčítania (*fused multiply-add*). Táto inštrukcia dokáže spracovať spoločne obidve operácie a je teda rýchlejšia. Dochádza pri nej len k jednému zaokrúhleniu výsledku, kdežto pri separátnych inštrukciách je zaokrúhlený aj medzivýsledok. Toto je jeden z architektonických rozdielov, ktorý je zodpovedný za rozdiely v konečnom výstupe aplikácie. Túto inštrukciu dokázal vygenerovať kompilátor automaticky, takže nie je potrebný žiadny ďalší zásah.

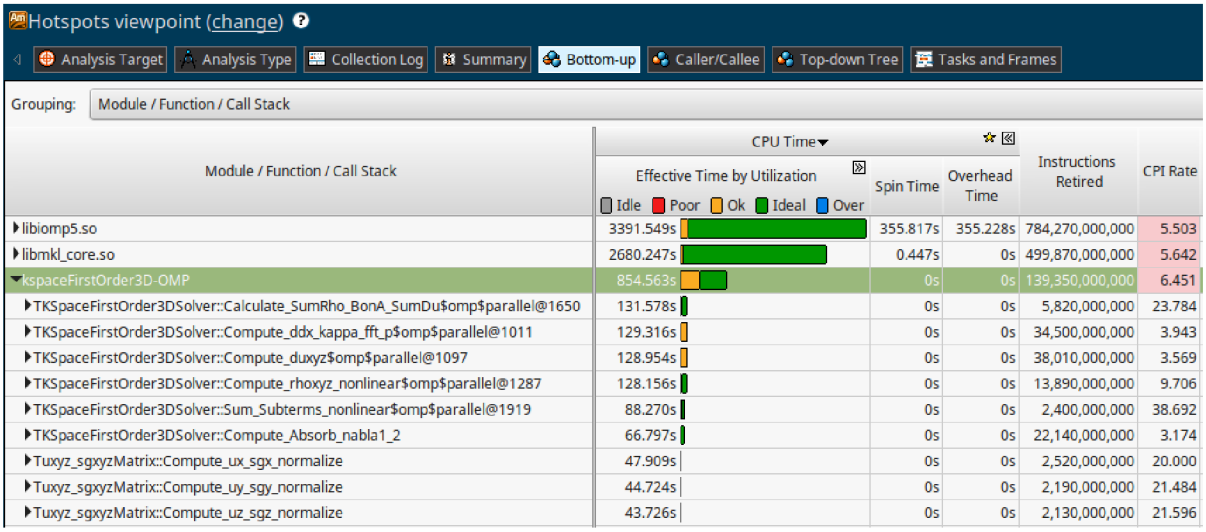
5.3.3 Ostatné metódy

Po úspešnom zoptimalizovaní dvoch najviac vyťažujúcich metód sa pozrieme opäť na analýzu vyťaženosti z VTune.



Module / Function / Call Stack	CPU Time: Effective Time by Utilization (Idle Poor Ok Ideal Over)	CPU Time: Spin Time	CPU Time: Overhead Time	Instructions Retired	CPI Rate
libiomp5.so	2941.702s	180.257s	174.211s	628,730,000,000	5.515
libmkl_core.so	2015.798s	0.342s	0s	488,820,000,000	4.339
kspaceFirstOrder3D-OMP	637.785s	0s	0s	110,610,000,000	6.066
TKSpaceFirstOrder3DSolver::Compute_rhoxyz_nonlinearompparallel@1287	109.791s	0s	0s	11,340,000,000	10.185
TKSpaceFirstOrder3DSolver::Compute_ddx_kappa_fft_pompparallel@1011	107.462s	0s	0s	28,510,000,000	3.965
TKSpaceFirstOrder3DSolver::Calculate_SumRho_BonA_SumDuompparallel@1650	93.964s	0s	0s	4,390,000,000	22.517
TKSpaceFirstOrder3DSolver::Compute_duxyzompparallel@1097	86.027s	0s	0s	31,720,000,000	2.853
TKSpaceFirstOrder3DSolver::Sum_Subterms_nonlinearompparallel@1919	70.827s	0s	0s	2,210,000,000	33.715
TKSpaceFirstOrder3DSolver::Compute_Absorb_nabla1_2	44.677s	0s	0s	17,540,000,000	2.680
Tuxyz_sgxyzMatrix::Compute_ux_sgx_normalize	39.192s	0s	0s	2,850,000,000	14.467
Tuxyz_sgxyzMatrix::Compute_uy_sgy_normalize	31.046s	0s	0s	2,110,000,000	15.479
Tuxyz_sgxyzMatrix::Compute_uz_sgz_normalize	28.650s	0s	0s	2,530,000,000	11.913

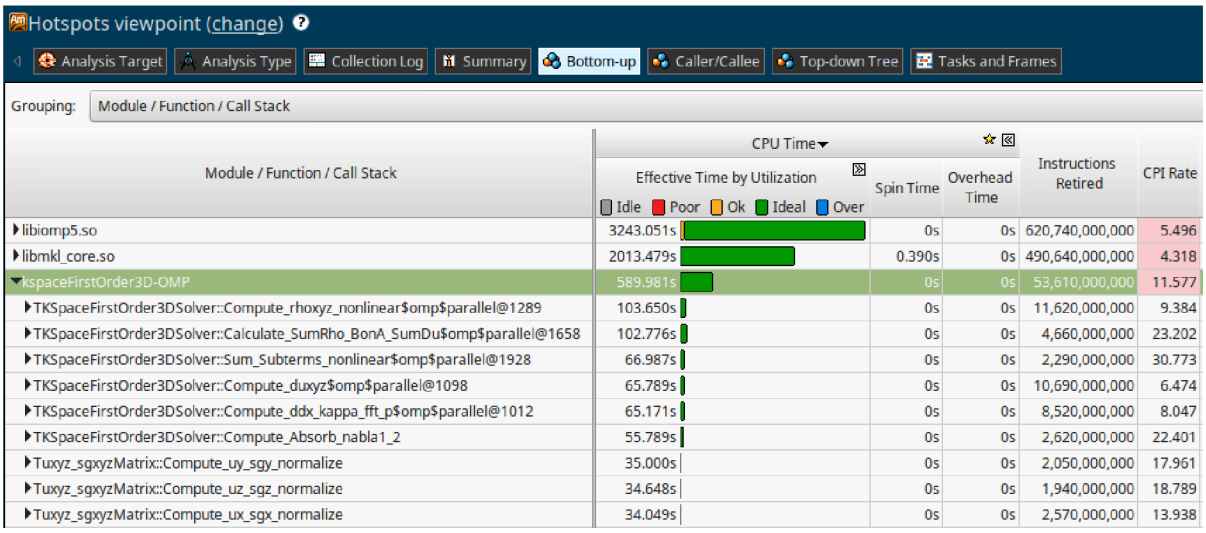
Obrázok 5.7 Pohľad na modul aplikácie k-Wave po optimalizácii dvoch metód. Dáta 240x240x240.



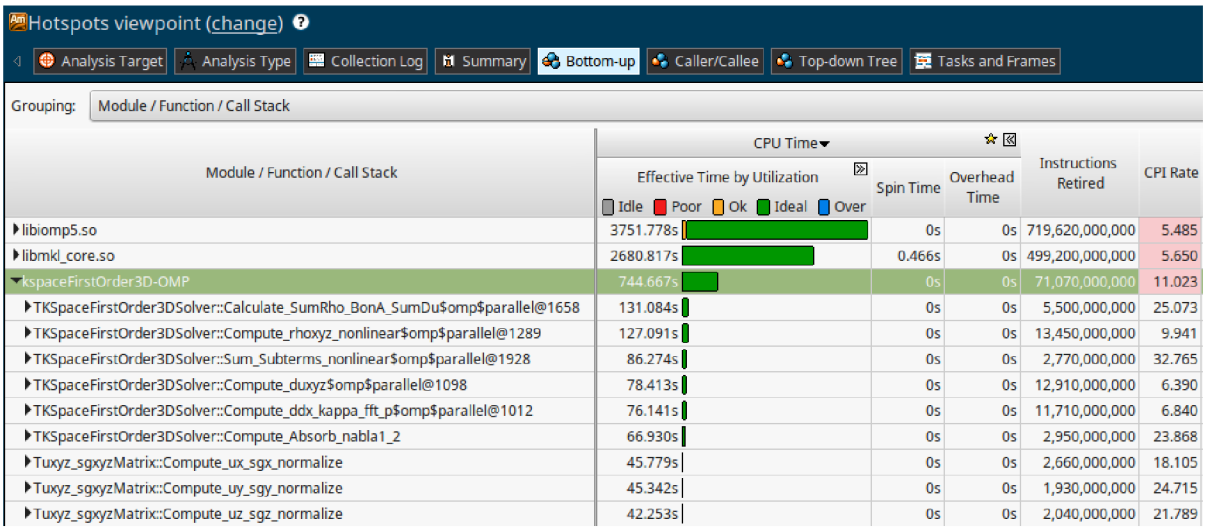
Module / Function / Call Stack	CPU Time: Effective Time by Utilization (Idle Poor Ok Ideal Over)	CPU Time: Spin Time	CPU Time: Overhead Time	Instructions Retired	CPI Rate
libiomp5.so	3391.549s	355.817s	355.228s	784,270,000,000	5.503
libmkl_core.so	2680.247s	0.447s	0s	499,870,000,000	5.642
kspaceFirstOrder3D-OMP	854.563s	0s	0s	139,350,000,000	6.451
TKSpaceFirstOrder3DSolver::Calculate_SumRho_BonA_SumDuompparallel@1650	131.578s	0s	0s	5,820,000,000	23.784
TKSpaceFirstOrder3DSolver::Compute_ddx_kappa_fft_pompparallel@1011	129.316s	0s	0s	34,500,000,000	3.943
TKSpaceFirstOrder3DSolver::Compute_duxyzompparallel@1097	128.954s	0s	0s	38,010,000,000	3.569
TKSpaceFirstOrder3DSolver::Compute_rhoxyz_nonlinearompparallel@1287	128.156s	0s	0s	13,890,000,000	9.706
TKSpaceFirstOrder3DSolver::Sum_Subterms_nonlinearompparallel@1919	88.270s	0s	0s	2,400,000,000	38.692
TKSpaceFirstOrder3DSolver::Compute_Absorb_nabla1_2	66.797s	0s	0s	22,140,000,000	3.174
Tuxyz_sgxyzMatrix::Compute_ux_sgx_normalize	47.909s	0s	0s	2,520,000,000	20.000
Tuxyz_sgxyzMatrix::Compute_uy_sgy_normalize	44.724s	0s	0s	2,190,000,000	21.484
Tuxyz_sgxyzMatrix::Compute_uz_sgz_normalize	43.726s	0s	0s	2,130,000,000	21.596

Obrázok 5.8 Pohľad na modul aplikácie k-Wave po optimalizácii dvoch metód. Dáta 256x256x256.

Vidíme, že už zoptimalizované metódy viac nie sú jednoznačne na prvých miestach a momentálne sú na tom podobne ako ďalšie metódy. Optimalizácia prvých dvoch metód priniesla veľmi dobré výsledky aj z hľadiska celkového času simulácie. Rovnako dobré výsledky optimalizovaním ďalších kandidátov už však čakať nemôžeme, keďže ich vplyv je oveľa menší. Týmito kandidátmi sú metódy *Compute_Absorb_nabla1_2()*, *Compute_ddx_kappa_fft_p()*, *Compute_duxyz()* a *Compute_hoxyz_nonlinear()*. Po ich analýze môžeme povedať nasledovné: Slučky v týchto metódach sú zanorené a vnútornú slučku možno vždy zvektorizovať, pretože tam neexistujú dátové závislosti. Výpočty sú prevádzané nad komplexnými hodnotami, kde je reálna a imaginárna časť uložená v pamäti vedľa seba. Nie je teda možné použitie prúdových inštrukcií ukladania do pamäti. Ďalej nie je možné použiť direktívu *#pragma collapse* (ktorá je odporúčaná pre použitie na Xeon Phi), pretože slučky nie sú takzvane čisto zanorené – existujú medzi nimi ďalšie inštrukcie. Kompilátor bol v tomto prípadne schopný niektoré slučky zvektorizovať automaticky, avšak generuje pritom inštrukcie pracujúce s nezarovnanou pamäťou (v metóde *Compute_rhoxyz_nonlinear()*). Iné slučky naopak nebol schopný zvektorizovať vôbec (v metódach *Compute_ddx_kappa_fft_p()*, *Compute_duxyz()*). Opäť teda kompilátoru pomôžeme použitím direktívy *#pragma omp simd*, spolu s klauzulou *aligned*. Nasledujú opäť obrázky 5.9 a 5.10 z VTune po použití tejto optimalizácie, kde je vidno zlepšenie u metód, ktoré neboli predtým zvektorizované kompilátorom. Naopak metóda, ktorá už bola predtým zvektorizovaná kompilátorom, nevykazuje žiadne známky zmeny.



Obrázok 5.9 Pohľad na modul aplikácie k-Wave po optimalizácii ostatných metód. Dáta 240x240x240.



Obrázok 5.10 Pohľad na modul aplikácie k-Wave po optimalizácii ostatných metód. Dáta 256x256x256.

Pri porovnaní s úplne prvými obrázkami z VTune si všimnime si, že sme sa zbavili všetkých oranžových políčiek značiacich, že procesorový čas nebol využitý úplne ideálne. Na nulové hodnoty klesli časy ukazujúce napríklad čakanie vlákien v rámci knižnice OpenMP (spin/overhead time). Celkovo sa aj znížil čas využívaný touto knižnicou, kvôli celkovému zníženiu počtu iterácii v paralelných blokoch. Stále je tento čas celkom vysoký a to prikladáme tomu, že v aplikácii je stále veľa kódu, ktorý beží sériovo (jednovláknovo).

6 Experimentovanie

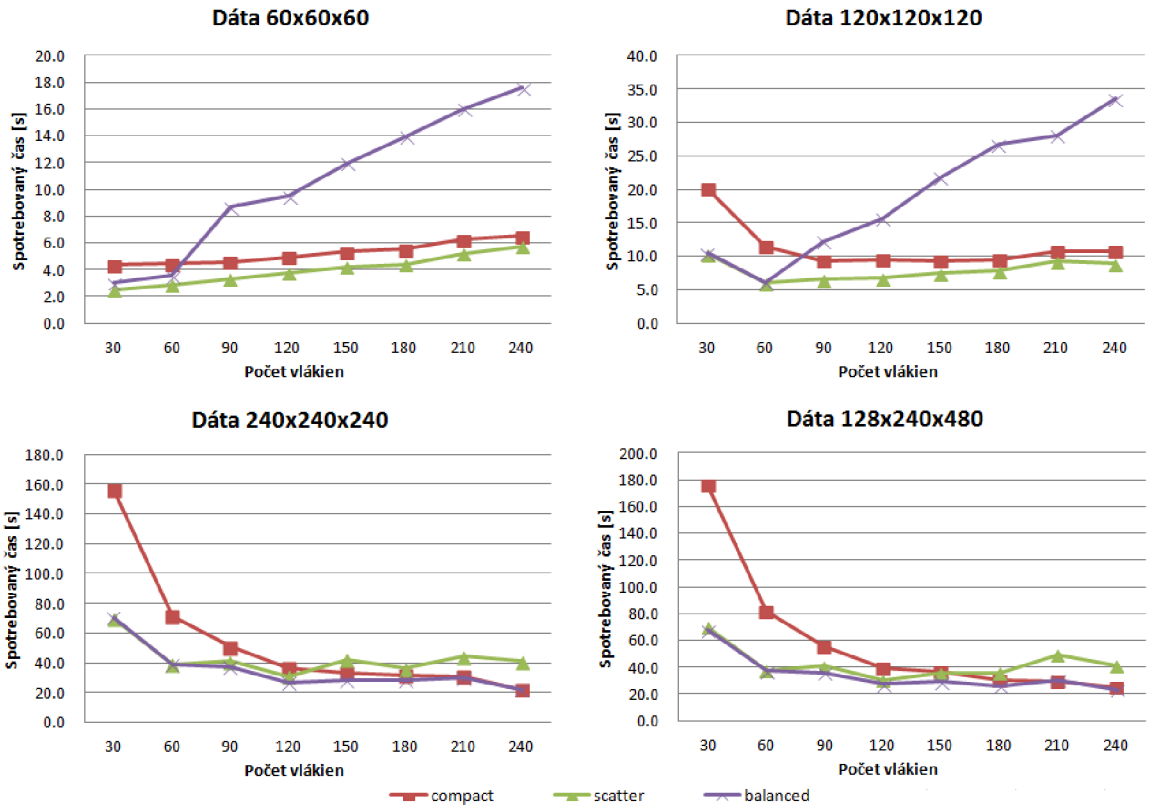
Intel Xeon Phi je rozhodne zaujímavá platforma, na ktorej sa dá vykonať veľké množstvo experimentov. Jej správanie a výkon sú odlišné pri použití rôznych veľkostí vstupných dát, pri použití rôznych počtov vlákien a rovnako aj pri použití rôznych typov afinity. To nám dáva veľké množstvo experimentálnych kombinácii. Pre meranie výkonnosti/času stráveného v jednotlivých metódach bol zdrojový kód inštrumentovaný (časovačmi), pretože VTune profiler pridáva veľkú réžiu, a preto sú celkové exekučné časy niekedy o dosť väčšie. Zároveň tiež nie je možné použiť plných *240* vlákien, pretože s použitím VTune na Xeon Phi je nutné nechať jedno jadro (štyri vlákna) voľné práve pre VTune.

Kapitolu začneme zmeraním výkonu základnej verzie a jej porovnaním s verziou pre štandardný procesor. Potom budeme pokračovať zhodnotením jednotlivých optimalizácií, prevedených v predchádzajúcej kapitole, na jednotlivých najvyťaženejších metódach identifikovaných pomocou VTune rovnako v prechádzajúcej kapitole. V každej zoptimalizovanej funkcii zhodnotíme dopad na celkový výpočtový čas aplikácie. Nakoniec sa pozrieme na problém vplyvu Fourierovej transformácie na celkový výpočtový čas simulácie.

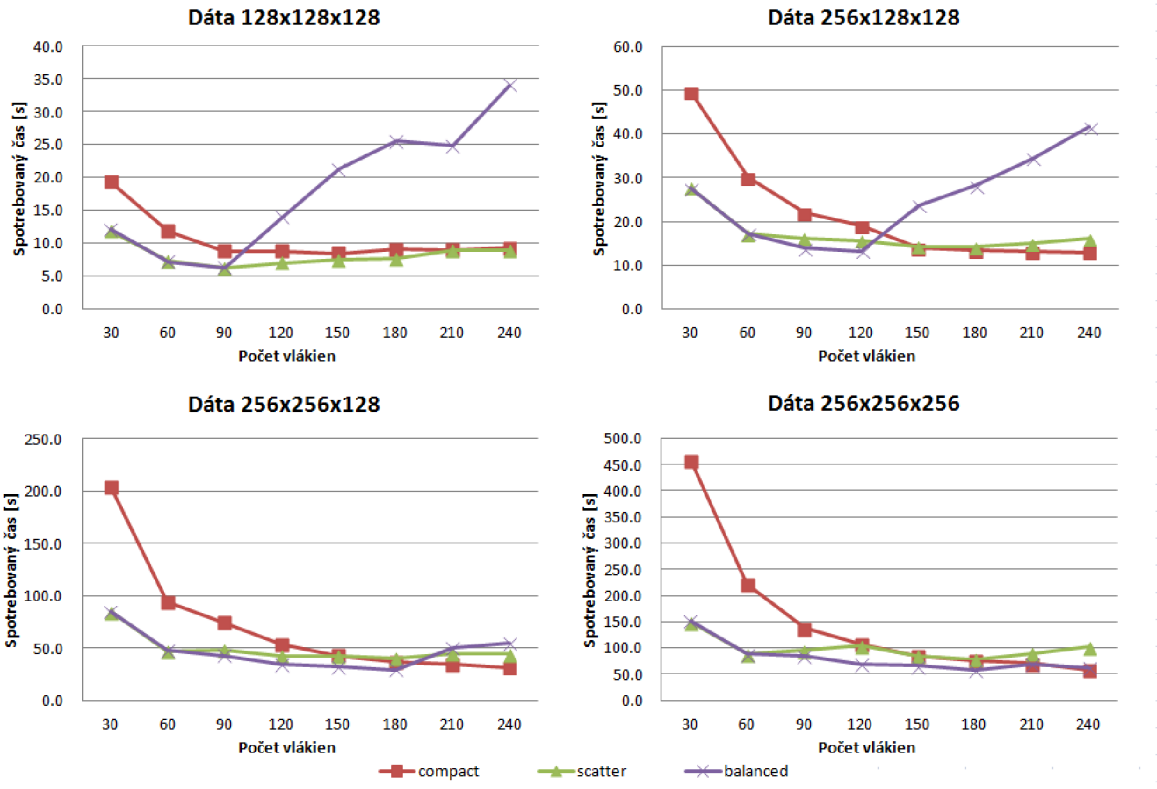
Použité grafy predstavujú len vybranú časť dát, takých, ktoré sú reprezentatívne, prípadne niečím zaujímavé. Tieto dáta boli získané pri behu rôznych verzií aplikácie. Každá z týchto verzií pritom bola experimentálne spustená na dátach o veľkostiach *60x60x60*, *120x120x120*, *240x240x240*, *128x128x128*, *256x256x128*, *256x256x128*, *256x256x256*, *512x256x256* bodov, pri použití význačných počtov vlákien a to *30*, *60*, *90*, *120*, *150*, *180*, *210*, *240* vlákien a pri použití *compact*, *scatter* a *balanced* typov afinity. Pre jednoduché porovnanie bolo vždy použitých *100* simulačných krokov. Všetky porovnávania boli urobené oproti verzii pre bežné procesory spúšťanej konkrétne na procesore *Intel Xeon E5-2470* pri *16* použitých vláknach.

6.1 Základná verzia

Základnou verziou rozumieme takú verziu, ktorá bola vyvinutá z verzie pre bežný procesor, s použitím minimálneho počtu zmien tak, aby bola plne funkčná na procesore Xeon Phi. Táto verzia už je optimalizovaná použitím OpenMP paralelizácie. Naopak nie je vektorizovaná, pokiaľ nebol schopný automatickej vektorizácie samotný kompilátor. Všetky dáta boli získané z niekoľkých behov vždy pre *100* simulačných krokov.

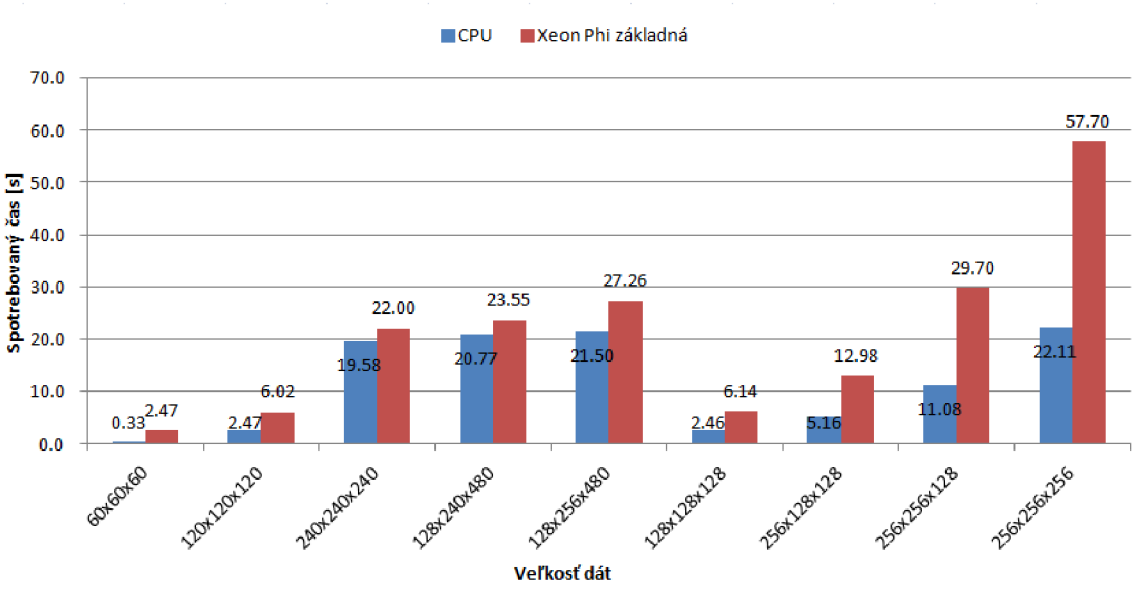


Obrázok 6.1 Vplyv počtu vlákien a typu afinity na celkový čas simulácie.



Obrázok 6.2 Vplyv počtu vlákien a typu afinity na celkový čas simulácie.

Grafy zobrazené na obrázkoch 6.1 a 6.2 ukazujú rôzne veľkosti vstupných dát, pri použití rôznych počtov vlákien a rôznych typov afinity. Pri najmenšej veľkosti dát (*60x60x60*) je verzia najrýchlejšia pri použití iba *30* vlákien. Čím sa veľkosť dát zväčšuje, tým sa postupne zvyšuje počet vlákien, pri ktorých je verzia najrýchlejšia. Dáta *120x120x120* pri *60* vláknach, *128x128x128* pri *90* vláknach. Pri veľkosti *256x128x128* a väčších dát už je však základná verzia najrýchlejšia (málokedy nie je úplne najrýchlejšia, ale má len minimálnu stratu) pri *240* vláknach. Všimnime si ďalej, že v prípade malých dát je najlepšou možnosťou zvoliť afinutu typu scatter, zatiaľ čo pri väčších dátach to je afinita typu balanced. Tá si udržuje najlepšiu výkonnosť pri asi všetkých počtoch použitých vlákien. Naopak, pri kombinácii malých dát a vyššieho počtu vlákien je vyslovene nepoužiteľná. Afinita typu compact je zase nepoužiteľná pri nízkom počte vlákien a naopak, vyrovnáva sa najlepším výsledkom, keď je použitá na väčších dátach s plným počtom *240* vlákien.



Obrázok 6.3 Porovnanie verzie pre bežné procesory s základnou verziou pre Intel Xeon Phi.

Obrázok 6.3 zobrazuje porovnanie základnej verzie s verziou pre bežné procesory. Porovnávané sú najlepšie dosiahnuté časy naprieč všetkými použitými vláknami a afinitami. Z grafu môžeme vyčítať niekoľko vecí. Základná verzia je pomalšia vo všetkých prípadoch použitých dát. Veľkosť dát, ktoré sme pokladali za „ideálne“ – *240x240x240*, je na tom v tomto porovnaní najlepšie. Podobne dobre si vedú aj dáta *128x240x480* a *128x256x480*. Všetky majú spoločné to, že sú násobkom maximálneho počtu jadier. Porovnajme si dáta *128x240x480* oproti *256x256x128*. Veľkosťou sú prvé dáta asi *1,7* krát väčšie ako druhé a pritom je ich simulácia rýchlejšia (*23,35 s* ku *29,70 s* .) Rovnako zle sú na tom aj iné „neideálne“ dáta. Zle sú na tom aj dáta malých rozmerov. To je však pochopiteľné, pretože aby sme mohli rozumne využiť paralelizmu, potrebujeme dosiahnuť rozumného množstva iterácií v jednotlivých slučkách pri použití dostatočného počtu vlákien (aby sme kompenzovali rýchlosť bežného procesora, ktorý je výkonom na vlákno ďaleko pred Xeon Phi) a to malé dáta nespĺňajú.

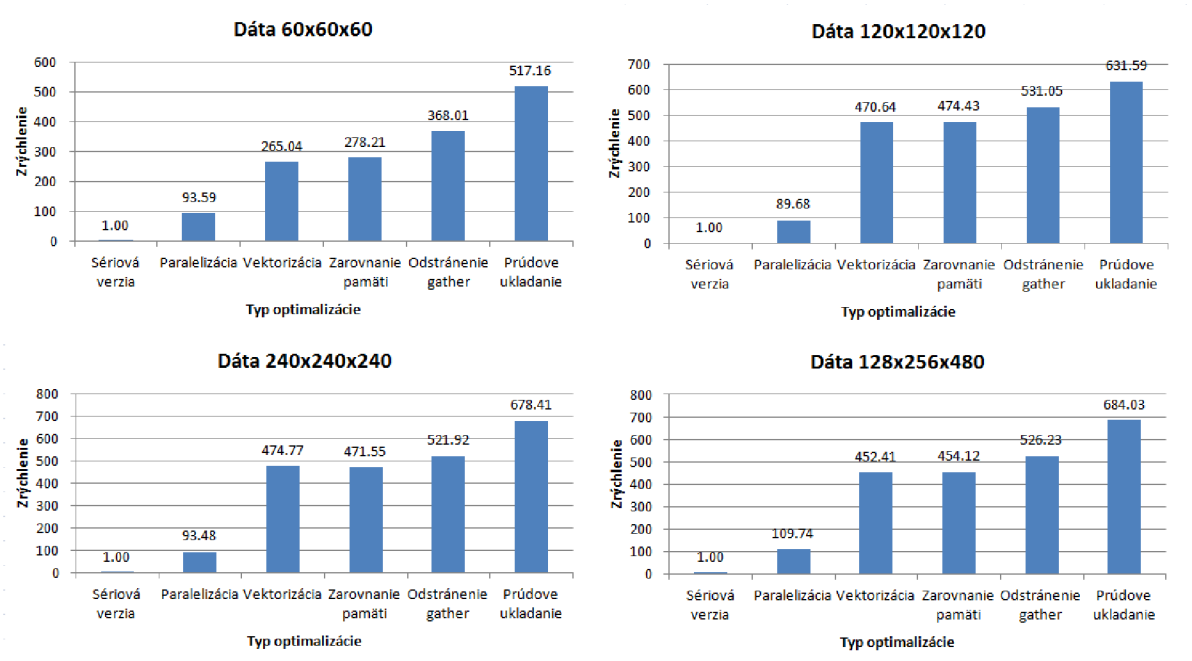
6.2 Optimalizované verzie

Každá optimalizovaná verzia predstavuje aplikáciu po optimalizácii práve jednej metódy. V rámci každej z nich si ukážeme dopad jednotlivých optimalizačných techník. Ich výkonnosť bola meraná v jednotkách *GFLOPS* (billion floating point operations per second) – počet operácií v plávajúcej desatinnej čiarke za jednu sekundu. Výpočet tejto jednotky pre merané metódy prebiehal nasledovne: Spočítali sme počet operácií v plávajúcej desatinnej čiarke vo vnútri výpočtovej slučky a získaný počet sme vynásobili celkovým počtom prvkov v nej spracovaných. Túto hodnotu predstavujúcu celkový počet operácií sme následne podelili časom stráveným v tejto metóde. Všetky ostatné inštrukcie v metódach boli ignorované. Druhou metrikou je dosiahnutá pamäťová priepustnosť. Tá bola vypočítaná ako počet operácií načítania z pamäti a zápisu do pamäti v slučke, vynásobená počtom prvkov v nej spracovanch a znovu vynásobená veľkosťou jedného elementu, a nakoniec podelená časom, ktorý metóda spotrebovala. Ďalej si ukážeme dopad na celkový čas simulácie a porovnanie s predchádzajúcimi verziami. Všetky dáta boli získané pre *100* simulačných krokov.

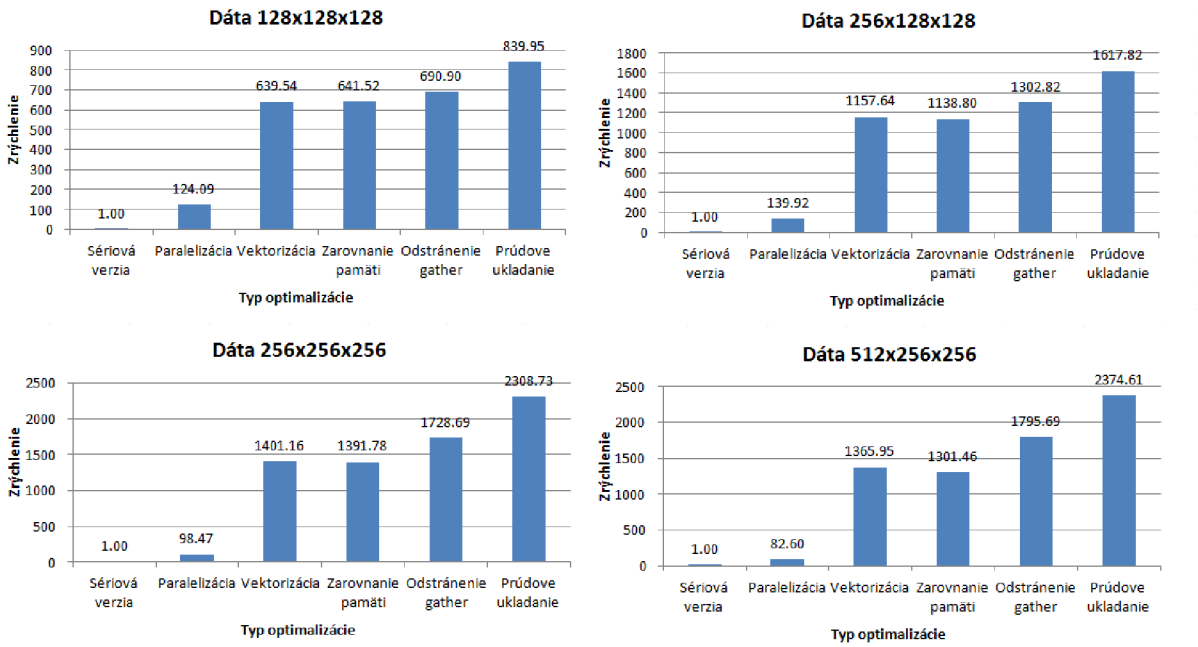
6.2.1 Optimalizácia Calculate_SumRho_BonA_SumDu()

Zrýchlenie optimalizačných krokov

Nasledujúce grafy ukazujú zrýchlenie získané pomocou jednotlivých optimalizačných krokov. Grafy sú zostrojené z najlepších výsledkov pre jednotlivé optimalizačné kroky. To znamená, že mohli byť dosiahnuté napríklad pri rôznom počte vlákien, či type afinity. Zrýchlenie je uvedené voči výkonu sériovej verzie.



Obrázok 6.4 Zrýchlenie metódy Calculate_SumRho_BonA_SumDu() na základe použitých optimalizácií.



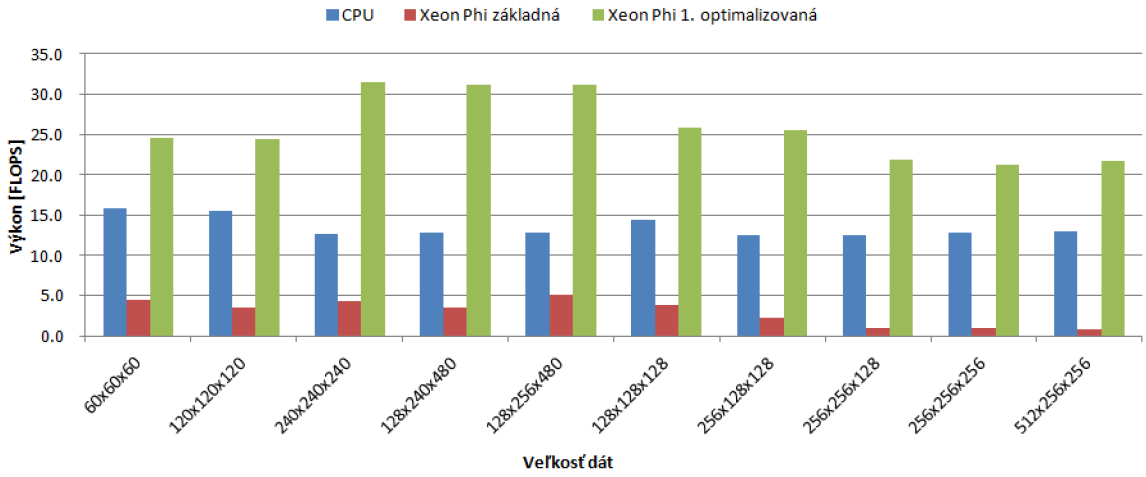
Obrázok 6.5 Zrýchlenie metódy Calculate_SumRho_BonA_SumDu() na základe použitých optimalizácií.

Prvá štvorica grafov na obrázku 6.4 ukazuje naše „ideálne“ dáta. Vidíme, že výkon je takmer rovnaký po všetkých optimalizačných krokoch. Menšou výnimkou sú najmenšie dáta. Vidíme, že oproti sériovej verzii používajúcej jedno vlákno, sa zdvihol celkový výkon až asi *650*-násobne. Najväčší vplyv má na to paralelizácia, ktorá výkon zvýši asi *100*-násobne. Spolu s vektorizáciu sa dostávame už na *450*-násobné zrýchlenie a teda vektorizácia prinesie samostatne asi ďalšie *4,5*-násobné zrýchlenie. Ďalšie optimalizácie už samostatne neprinášajú až také veľké zrýchlenie, ale je zjavný ich synergický efekt. Môžeme povedať, že zarovnanie dát v pamäti neprinieslo asi žiadne zlepšenie – v niektorých prípadoch je vidno mierne zlepšenie, naopak niekedy mierne zhoršenie výkonu. Odstránenie generovaných inštrukcií gather prinesie ďalších okolo *12 %* výkonu naviac. Dobrý nárast výkonu o takmer *30 %* nám zaistili tri prúdové ukladania dát do pamäti. Zaujímave je porovnanie s druhou štvoricou dát na obrázku 6.5, kde sa s ich zväčšujúcou veľkosťou postupne dvíha aj výkon, až sa pri najväčších z nich dostávame na vyše *2300*-násobný nárast výkonu. Nárast výkonu po paralelizácii je podobný prvým dátam, ale vektorizácia tu prináša až *14*-násobný nárast výkonu. Zarovnanie je natom približne rovnako a ďalšie dve optimalizácie tu mierne zvýšili svoj výkon oproti „ideálnym“ dátam.

Porovnanie s verziou na bežnom procesore a základnou verziou

Tabuľka 6.1 Zobrazenie výkonu a pamäťovej priepustnosti pre jednotlivé verzie a dáta. Zobrazené sú aj nastavenia typu afinity a počtu vlákien pri ktorých boli dosiahnuté.

Výkonnosť jednotlivých verzií										
Veľkosť dát	CPU verzia		Xeon Phi základná verzia				Xeon Phi 1. optimalizovaná verzia			
	GFLOPS	GB/s	GFLOPS	GB/s	Typ affinity	Počet vlákien	GFLOPS	GB/s	Typ affinity	Počet vlákien
60x60x60	15.90	69.95	4.43	19.49	scatter	150	24.48	107.71	scatter	60
120x120x120	15.51	68.22	3.47	15.27	balanced	180	24.45	107.57	balanced	120
240x240x240	12.75	56.11	4.34	19.10	balanced	240	31.51	138.65	balanced	60
128x240x480	12.84	56.48	3.49	15.37	compact	240	31.17	137.13	balanced	60
128x256x480	12.80	56.34	5.01	22.05	balanced	240	31.24	137.44	scatter	60
128x128x128	14.42	63.47	3.81	16.76	compact	240	25.79	113.47	balanced	120
256x128x128	12.58	55.35	2.21	9.73	compact	240	25.57	112.50	compact	210
256x256x128	12.54	55.19	0.98	4.31	compact	240	21.89	96.32	compact	240
256x256x256	12.86	56.57	0.91	3.98	compact	240	21.22	93.39	compact	240
512x256x256	12.97	57.05	0.75	3.31	balanced	240	21.64	95.20	compact	240

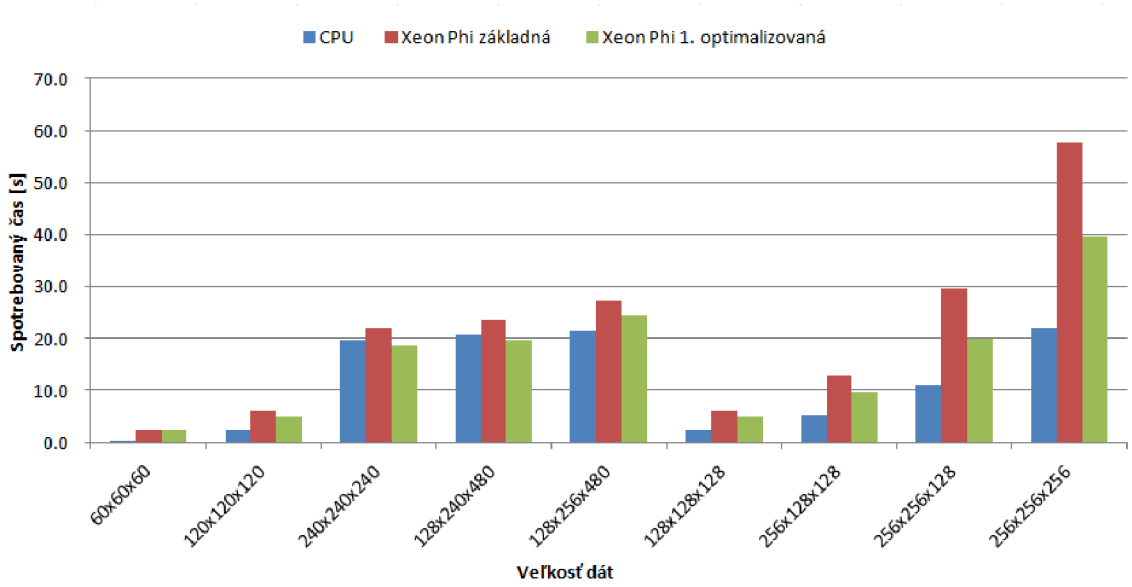


Obrázok 6.6 Porovnanie výkonu metódy Calculate_SumRho_BonA_SumDu() na bežnom procesore a na Intel Xeon Phi pred a po jej optimalizácii.

Predchádzajúca tabuľka 6.1 a graf na obrázku 6.6 zobrazujú porovnanie výkonnosti prvej optimalizovanej metódy s verziou na bežnom procesore (CPU verzia) a so základnou (paralelnou) verziou na Xeon Phi. Pre Xeon Phi verzie sú v tabuľke zobrazené aj nastavenia typu afinity a počet vlákien, pri ktorých bol tento výkon dosiahnutý. Vidíme, že pri „ideálnych“ dátach sa dostávame k takmer *2,5*-násobnému výkonu oproti CPU verzii a o niečo viac ako *7*-násobnému oproti základnej verzii. Keď sa pozrieme na dáta *256x256x256*, nárast výkonu oproti CPU verzii je len asi *1,6*-násobný, ale oproti základnej verzii až viac ako *23*-násobný. Pritom celkovo je metóda pri dátach *240x240x240* *1,5*-krát rýchlejšia ako pri dátach o rozmeroch *256x256x256*. Je to spôsobené

veľmi slabým výkonom základnej verzie pri „neideálnych“ dátach. Ďalšia vec, ktorú si všimneme je nastavenie, pri ktorom je dosiahnutý najlepší výsledok. Pred optimalizáciou bolo potrebné k najvyššiemu výkonu použiť takmer vždy *240* vlákien. Po optimalizácii, je to pri „ideálnych“ dátach len *60* vlákien (použitie *120* vlákien má výsledky len o málo horšie). Toto bude spôsobené znížením efektívneho počtu iterácii v slučke *16*-krát vďaka vektorizácii, a teda paralelizácia je vo výsledku efektívnejšia pri použití menšieho počtu vlákien. Keď sa pozrieme na pamäťovú priepustnosť, tak sa dostávame blízko k hranici *139 GB/s*. V článku [21] robili experimenty pre zistenie skutočnej pamäťovej priepustnoti karty Xeon Phi a nedostali sa na viac ako *164 GB/s* pre operácie načítania z pamäti a na *76 GB/s* pre zápis do pamäti. Vo výpočte našej metódy pritom máme 8 operácií načítania (jednu operáciu nepočítame, pretože dáta budú ešte v cache – číta sa z rovnakej lokácie) a 3 zápisy. Takže naša dosiahnutá hodnota *139 GB/s* zodpovedá ich výsledkom a je pravdepodobne stropom, ktorý je možné dosiahnuť.

Dopad na celkový čas simulácie



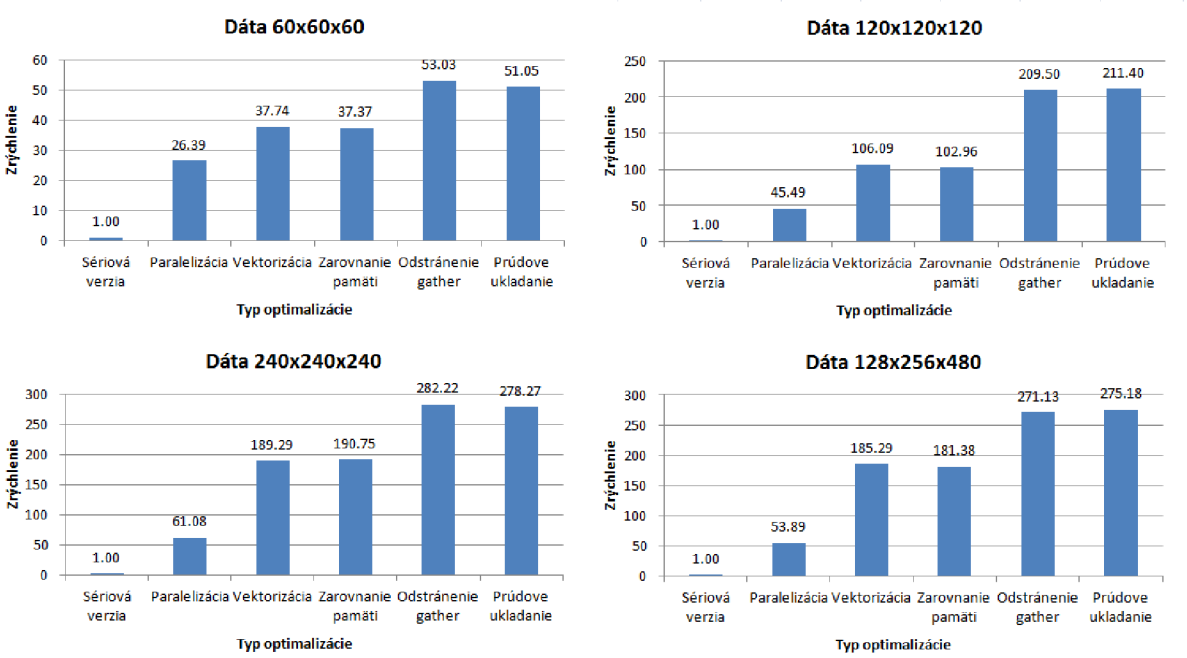
Obrázok 6.7 Porovnanie celkového času simulácie na bežnom procesore a na Intel Xeon Phi pred a po optimalizácii metódy Calculate_SumRho_BonA_SumDu().

Tento graf na obrázku 6.7 nám zobrazuje porovnanie celkového simulačného času naprieč verziami. Optimalizácia prvej metódy mala masívny dopad pre dáta *256x256x128* a *256x256x256*. Zrýchlenie simulácie tu je o *33* %, čo ale zďaleka nestačí na CPU verziu, stále asi 2-krát rýchlejšiu. Pri dátach, na ktorých bola simulácia zatiaľ najbližšie k CPU verzii to sú *240x240x240*, je zrýchlenie *15* %. Na týchto dátach je už však optimalizovaná verzia rýchlejšia ako CPU verzia – o *1* sekundu čo predstavuje približne *5* %.

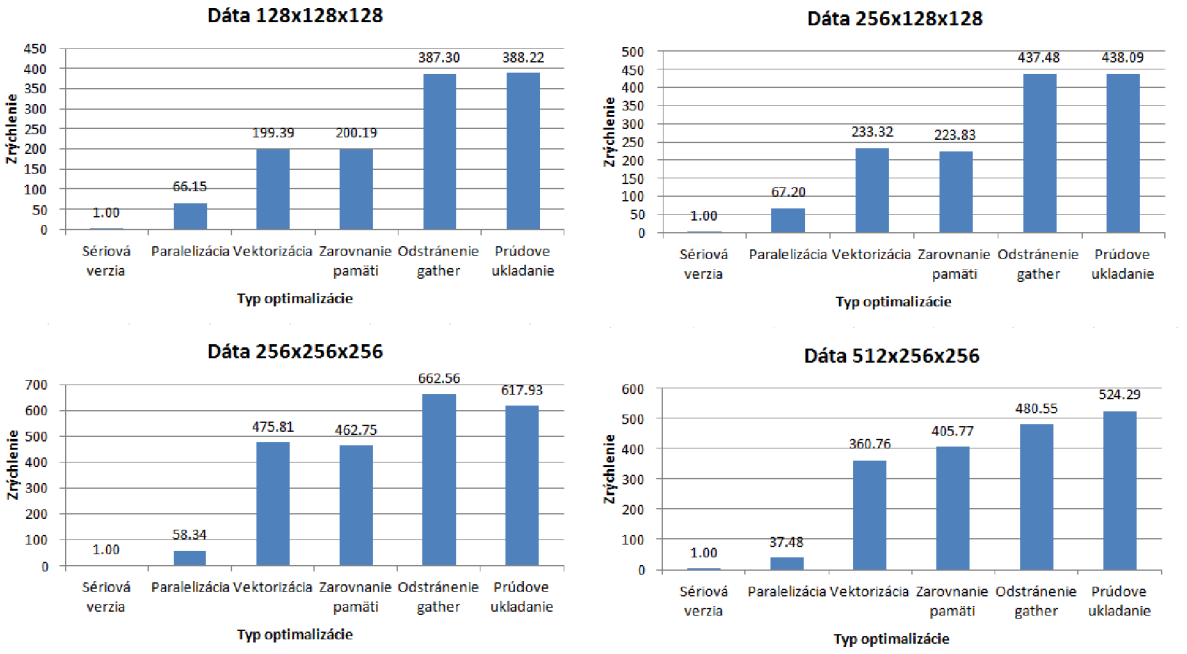
6.2.2 Optimalizácia Sum_subterms_nonlinear()

Zrýchlenie optimalizačných krokov

Podobne ako pri predchádzajúcej metóde si ukážeme zrýchlenie jednotlivých optimalizačných krokov na rovnakých dátach a pri rovnakých podmienkach. Keďže obidve slučky sú dosť podobné zameriame sa hlavne na rozdiely.



Obrázok 6.8 Zrýchlenie metódy Sum_subterms_nonlinear() na základe použitých optimalizácií.



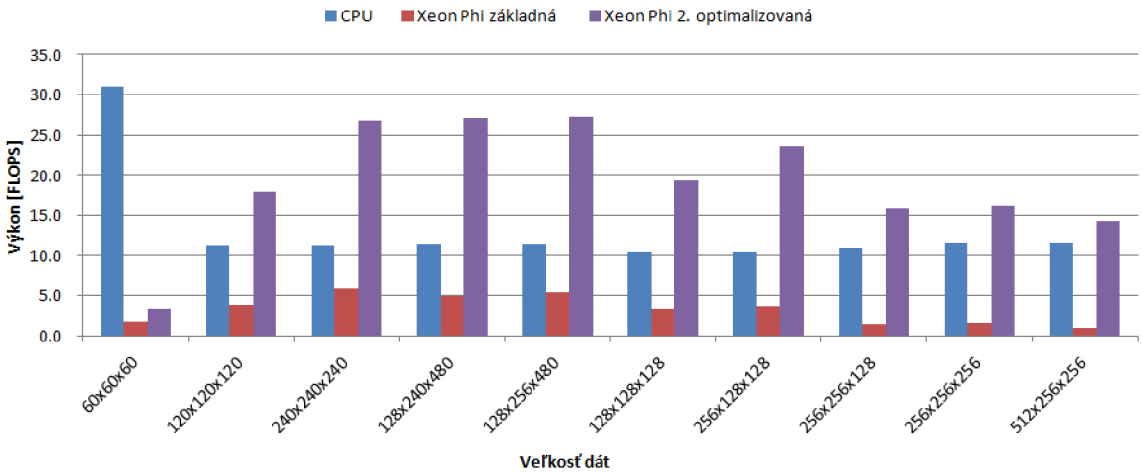
Obrázok 6.9 Zrýchlenie metódy Sum_subterms_nonlinear() na základe použitých optimalizácií.

Prvá vec, ktorú si všimneme je skoro o polovicu menšie zrýchlenie dosiahnuté po paralelizácii. Rovnako vektorizácia nepriniesla také zrýchlenie ako v prvom prípade. Je len niečo viac ako *3*-násobné pri dátach, ktoré považujeme za „ideálne“. Pri tých „neideálnych“ a väčších je to viac, asi *8* až *10*-násobok. Optimalizácia zarovnania dát dopadla rovnako. Niekedy je výkon vyšší a inokedy zase nižší. Rozhodne sa nemožno na zrýchlenie spoliehať. Je možné, že pri väčších dátach sa prejaví stále zrýchlenie, keďže naše najväčšie testované dáta zaberajú asi len *134 MB*, čo je menej ako *2 %* celkovo dostupnej pamäti na Xeon Phi. Zostávajúce miesto je však pre našu aplikáciu nedostupné, pretože je využité pre ostatné dáta. Prepísanie kódu tak, aby neboli generované inštrukcie typu gather prinieslo väčšie zrýchlenie ako u prvej metódy, okolo *30 %*, pri niektorých relatívne menších dátach aj o *80 %*. Väčšie zrýchlenie oproti prvej metóde je dané tým, že pomer medzi načítaním z pamäti pomocou inštrukcií gather oproti normálnemu zarovnanému načítaniu je *3* ku *3*, zatiaľ čo pri predchádzajúcej metóde to bolo len *3* ku *6*. Naopak, prúdové ukladanie do pamäti tentokrát zrýchlenie neprinieslo, má to podobný dopad ako zarovnanie – niekedy nevidno žiadny vplyv, inokedy menšie zrýchlenie alebo spomalenie.

Porovnanie s verziou na bežnom procesore a základnou verziou

Tabuľka 6.2 Zobrazenie výkonu a pamäťovej priepustnosti pre jednotlivé verzie a dáta. Zobrazené sú aj nastavenia typu afinity a počtu vlákien pri ktorých boli dosiahnuté.

Výkonnosť jednotlivých verzií										
Veľkosť dát	CPU verzia		Xeon Phi základná verzia				Xeon Phi 2. optimalizovaná verzia			
	FLOPS	GB/s	FLOPS	GB/s	Typ affinity	Počet vlákien	FLOPS	GB/s	Typ affinity	Počet vlákien
60x60x60	31.05	113.98	1.71	7.98	scatter	30	3.31	15.44	scatter	30
120x120x120	11.24	44.93	3.86	18.03	scatter	60	17.96	83.79	scatter	60
240x240x240	11.29	50.31	5.89	27.47	compact	240	26.82	125.15	balanced	120
128x240x480	11.45	50.59	4.88	22.77	balanced	240	27.07	126.35	balanced	120
128x256x480	11.36	51.50	5.34	24.91	balanced	240	27.26	127.20	balanced	60
128x128x128	10.40	47.06	3.29	15.36	scatter	60	19.32	90.16	scatter	60
256x128x128	10.42	47.06	3.61	16.85	balanced	120	23.53	109.83	balanced	120
256x256x128	11.01	48.80	1.41	6.57	compact	240	15.80	73.75	balanced	180
256x256x256	11.59	50.32	1.53	7.13	compact	240	16.18	75.52	balanced	240
512x256x256	11.64	52.16	1.02	4.76	compact	240	14.26	66.55	compact	240

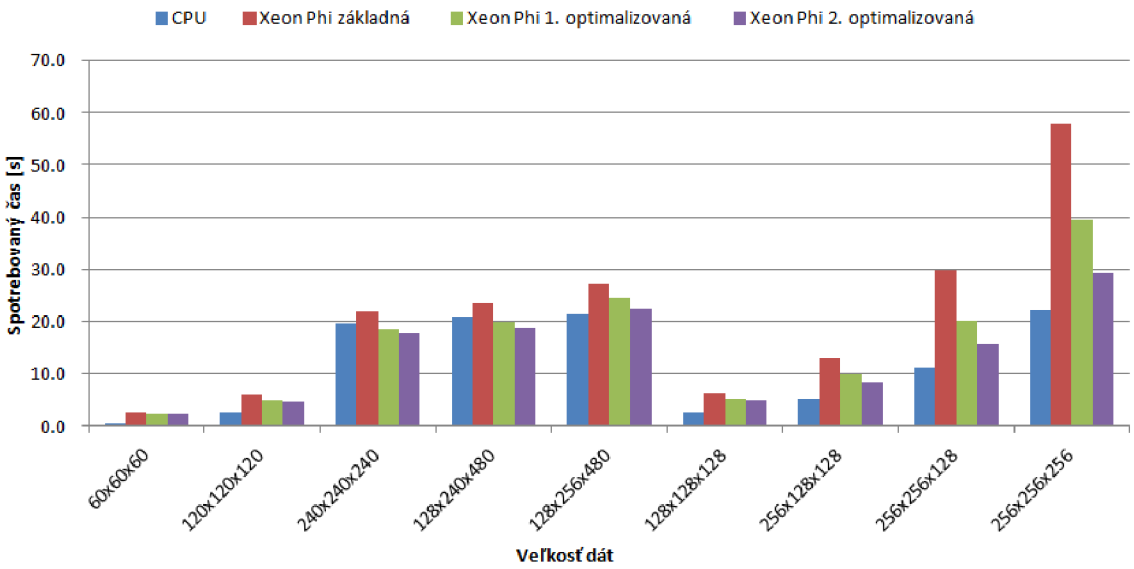


Obrázok 6.10 Porovnanie výkonu metódy Sum_subterms_nonlinear() na bežnom procesore a na Intel Xeon Phi pred a po jej optimalizácii.

Znova sa pozrieme porovnanie výkonnosti tejto optimalizovanej metódy s verziou na bežnom procesore (CPU verzia) a so základnou (paralelnou) verziou na Xeon Phi, ktoré vidíme na predchádzajúcom obrázku 6.10 a v tabuľke 6.2. Aj tu dosahujeme pri "ideálnych" dátach skoro *2,4*-násobného výkonu CPU verzie a *4,5*-násobného oproti základnej verzii. Pre "neideálne" dáta napríklad veľkosti *256x256x256* vidíme, že nárast výkonu oproti CPU verzii je už iba asi *1,4*-násobný a oproti základnej verzii až viac ako *10,5*-násobný. Rozdiel vo výkone pri použití dát *240x240x240* a *256x256x256* je tentokrát vyšší oproti prvej metóde – skoro *1,7*-krát. Môžeme povedať, že v tejto metóde má typ dát ešte väčší vplyv ako pri prvej metóde. Po optimalizovaní je najvyšší výkon dosiahnutý pri menšom počte vlákien. Situácia je rovnaká ako v predchádzajúcej optimalizovanej metóde, a teda znížením počtu iterácií v slučke vďaka vektorizácii, je vyššia efektivita pri použití nižšieho počtu vlákien.

Dopad na celkový čas simulácie

Asi najviac nás zaujíma, či sa nám podarilo dosiahnuť simuláciu rýchlejšiu ako CPU verzia aj pri iných veľkostiach dát, ako po prvej optimalizácii. Nie je to tak. Na grafe z obrázku 6.11 vidíme znova veľké zrýchlenie o ďalšiu štvrtinu pri dátach *256x256x128* a *256x256x256*, no nestačí to. Pri dátach *240x240x240* a *128x256x480*, na ktorých sme už po optimalizácii prvej metódy namerali rýchlejšie výsledky ako na CPU verzii, je zrýchlenie približne o *6 %*. V celkovom zúčtovaní sme rýchlejší už o *10 %* oproti CPU verzii.

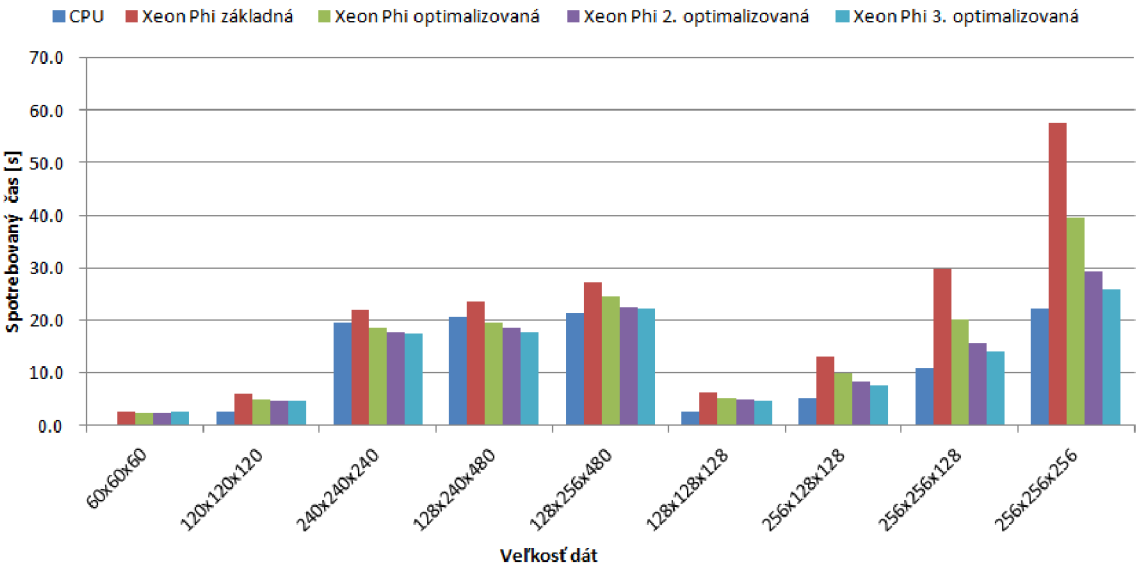


Obrázok 6.11 Porovnanie celkového času simulácie na bežnom procesore a na Intel Xeon Phi pred a po optimalizácii metódy Sum_subterms_nonlinear().

6.2.3 Optimalizácia ostatných metód – finálna verzia

V tomto kroku sme sa snažili optimalizovať ostatné metódy, u ktorých by to mohlo mať ešte nejaký zmysel, ale neočakávali sme už veľký výkonnostný posun. Nebudeme detailne analyzovať jednotlivé metódy samostatne, dopad optimalizácie je však celkom dobre vidieť z VTune analýzy, popísanej v sekcii 5.3.3. Zhodnotíme len ich vplyv na výsledný simulačný čas.

Dopad na celkový čas simulácie



Obrázok 6.12 Porovnanie celkového času simulácie na bežnom procesore a na Intel Xeon Phi pred a po optimalizácii ostatných metód.

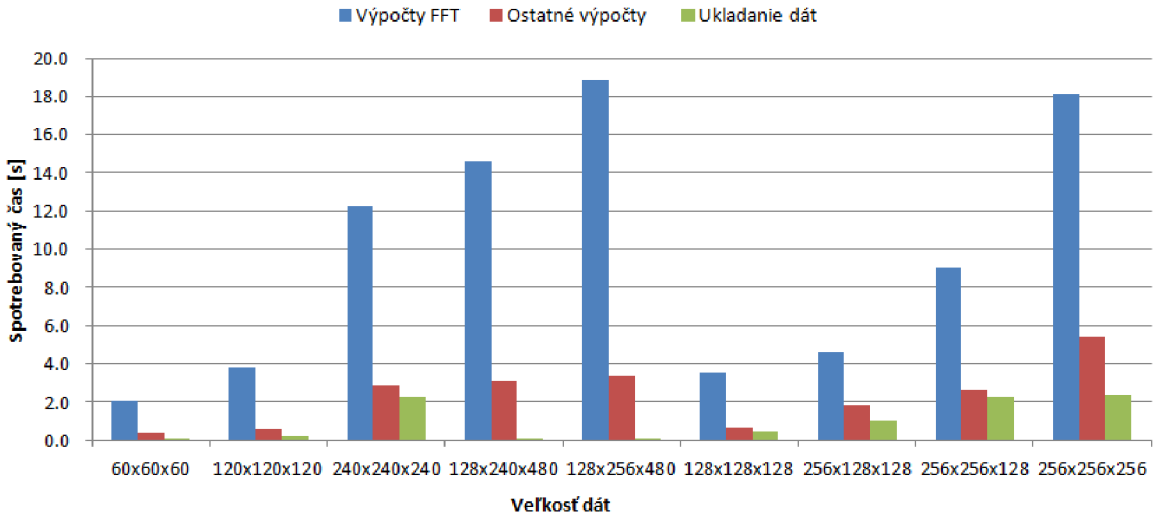
Vidíme (obrázok 6.12) úplne rovnaký scenár ako pri predchádzajúcich optimalizáciách. Pri totožných dátach opäť zaznamenávame celkom dobré zlepšenie, ale zase to nestačí na dobehnutie CPU verzie. Minimálne prírastky rýchlosti pre ideálnu verziu už len o málo vylepšujú jej celkovú rýchlosť oproti CPU verzii. Nakoniec sme ju predstihli o celkovo približne *12 %* lepším časom.

6.3 Vplyv ukladania dát a FFT

V tejto podkapitole prevedieme porovnanie medzi časom využitým na počítanie FFT knižnicou Intel MKL, ostatných výpočtov aplikácie k-Wave a času ukladania výstupu do súboru typu HDF5. Tabuľka 6.3 a jej grafická reprezentácia na obrázku 6.13 zobrazujú celkový čas simulácie (pri najlepších výsledkoch pre jednotlivé dáta) rozdelený práve medzi tieto tri kategórie.

Tabuľka 6.3 Zobrazenie celkového simulačného času v sekundách rozdeleného medzi tri zložky, ktoré ho tvoria.

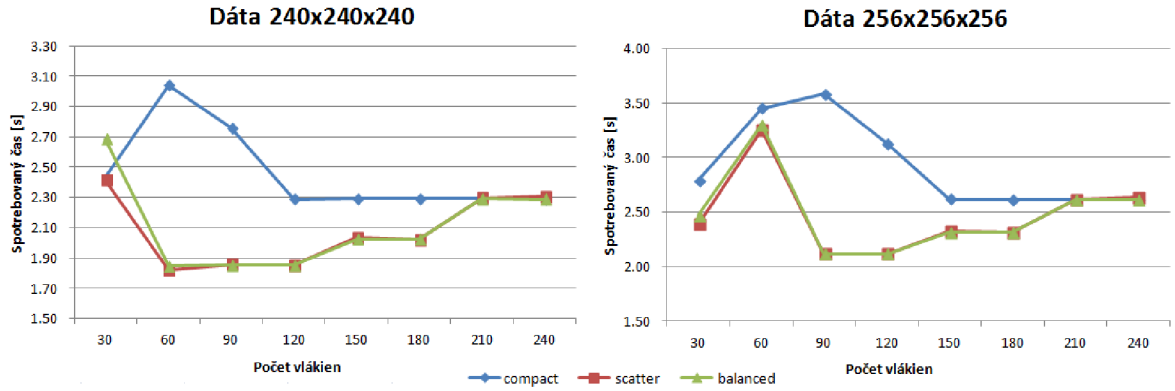
Veľkosť dát	Výpočty FFT	Ostatné výpočty	Ukladanie výstupu
60x60x60	2.05	0.34	0.06
120x120x120	3.80	0.56	0.24
240x240x240	12.22	2.90	2.29
128x240x480	14.56	3.07	0.03
128x256x480	18.80	3.36	0.03
128x128x128	3.54	0.66	0.42
256x128x128	4.60	1.84	1.06
256x256x128	9.04	2.62	2.31
256x256x256	18.09	5.40	2.32
512x256x256	171.86	11.46	4.69



Obrázok 6.13 Grafické zobrazenie celkového simulačného času rozdeleného medzi tri zložky, ktoré ho tvoria.

Ukladanie dát

Na prvý pohľad je vidieť viacero zaujímavých informácií. To, že výpočty Fourierových transformácií zaberajú najväčšie množstvo času sme vedeli. Nečakali sme však, že ukladanie výstupu do súboru bude až také pomalé a celkom výrazne zasahuje aj do celkového času. V rámci VTune sa prejavuje v module *vmlinux*. Počas jeho analýzy sme však netušili, že tento modul sa vôbec týka našej aplikácie a pokladali sme ho za proces operačného systému bežiaceho na karte. Keď sa pozrieme na dáta *128x240x480* a *128x256x480* – majú úplne minimálny čas strávený ukladaním simulačných dát do súboru. Vyzerá to skoro, ako keby toto zapisovanie bolo úplne vypnuté. Bohužiaľ sa nám nepodarilo zistiť príčinu. Vplyv použitia rôznych afinít a počtov vlákien ukazuje obrázok 6.14. Vyzerá to tak, že pokiaľ je použité príliš malé alebo príliš veľké množstvo vlákien, tak čas ukladania dát je horší. *120* vlákien vyzerá byť dobrým kompromisom. S veľkosťou dát samozrejme stúpa aj čas ich ukladania. Tieto namerané údaje boli získané pri simulácii s prednastavenými parametrami zápisu (parameter *-p*). Pokiaľ sme nechali zapisovať ešte viac dát použitím ďalších parametrov, tak sa čas zápisu ešte niekoľkonásobne zvýšil – napríklad pre dáta 256x256x256 na viac ako 6 sekúnd.

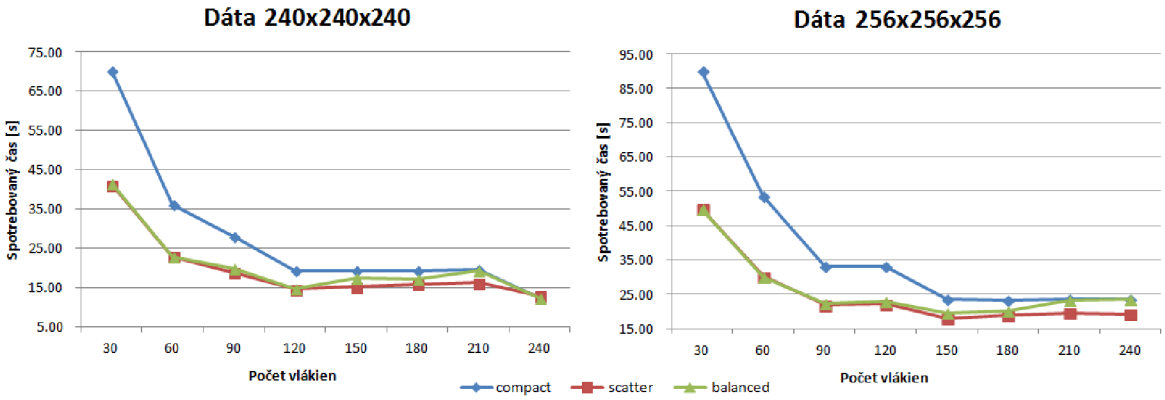


Obrázok 6.14 Vplyv typu afinity a počtu vlákien na čas ukladania simulačných dát do súboru.

Výpočty FFT

Teraz sa zameriame na výpočty fouriérových transformácií. Z testovacích dát boli len dve vzorky také, ktorých simulácia bola rýchlejšia od CPU verzie – *240x240x240* a *128x240x480*. Vzorka veľkosti *128x256x480* už bola pomalšia a je jasne viditeľné, že je to spôsobené zlým škálovaním výkonu FFT. Zatiaľ čo veľkosti dát sú v pomere k prvým menovaným *1 : 1,07 : 1,14*, tak čas strávený výpočtom FFT je v pomere *1 : 1,19 : 1,54* – veľkosť dát má teda pri výpočte FFT ešte väčší vplyv ako pri ostatných výpočtoch aplikácie k-Wave na karte Xeon Phi. Experimenty sme robili aj na dátach o veľkosti *512x256x256* avšak výsledky sme nezahrnuli do žiadnych z predchádzajúcich grafov zobrazujúcich celkový čas aplikácie. Je to z jedného prostého dôvodu – čas simulácie

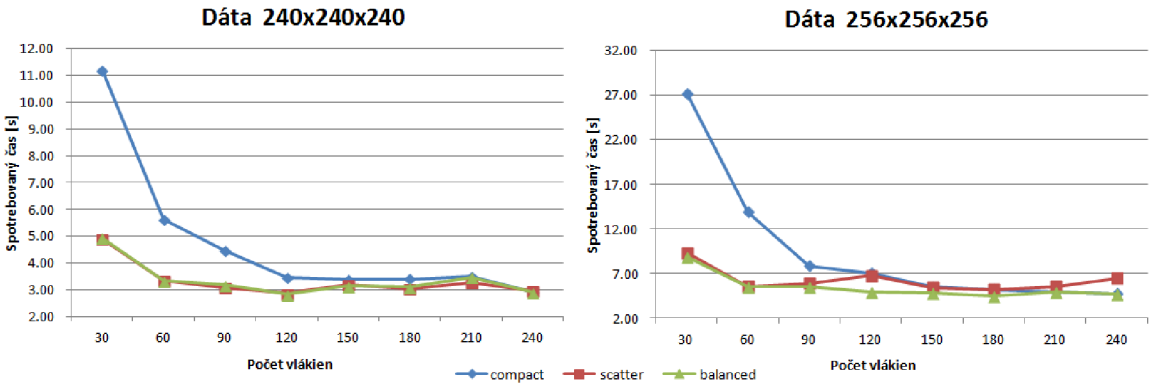
sa obrovsky zvýšil práve na výpočtoch FFT, viď Tabuľka 6.3. Myslíme si, že by to mohli spôsobovať problémy s pamäťou (viac v podkapitole o zhodnotení experimentov).



Obrázok 6.15 Vplyv typu afinity a počtu vlákien na čas výportu FFT.

Ostatné výpočty

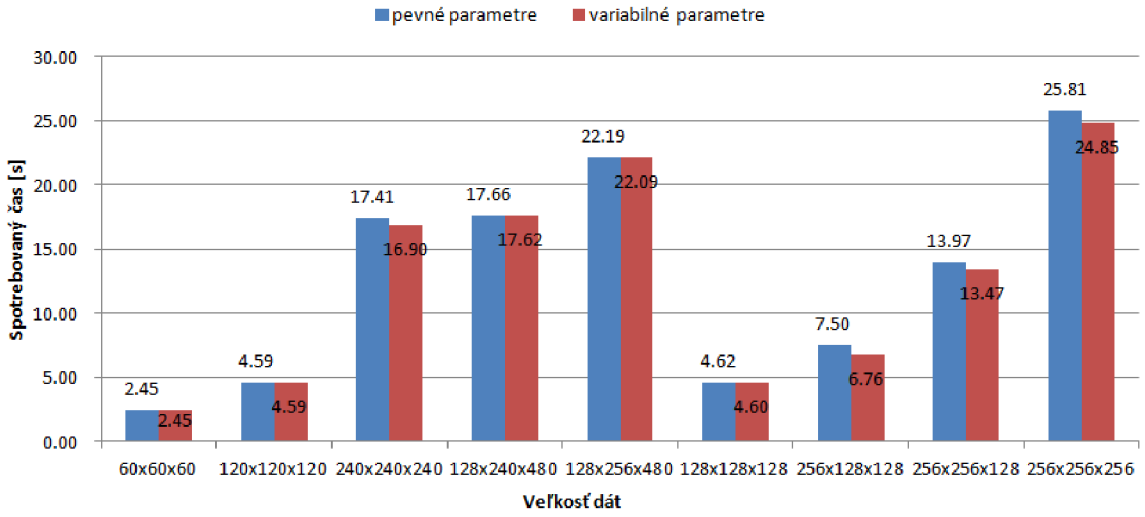
Nakoniec sa pozrieme iba na čas ostatných výpočtov (mimo FFT a zápisu). Pri dátach, ktoré sú násobkom počtu jadier (*240x240x240*, *128x240x480*, *128x256x480*), tak vidíme dobré škálovanie výkonu. Keď však porovnáme s ostatnými dátami, tak tie výkonovo zaostávajú. Napríklad dáta *256x256x256* sú iba *1,07*-krát väčšie od dát *128x256x480*, ale čas ich spracovania je až *1.6*-krát väčší. To je dôsledkom toho, že nie úplne všetky iterácie vo výpočtových slučkách boli efektívne paralelizované/zvektorizované a teda niektoré dáta sú spracovávané malým množstvom vlákien prípadne v sérii. Naproti tomu pri dátach o násobkoch počtu jadier, je paralelizácia/vektorizácia plne efektívna.



Obrázok 6.16 Vplyv typu afinity a počtu vlákien na čas ostatných výpočtov.

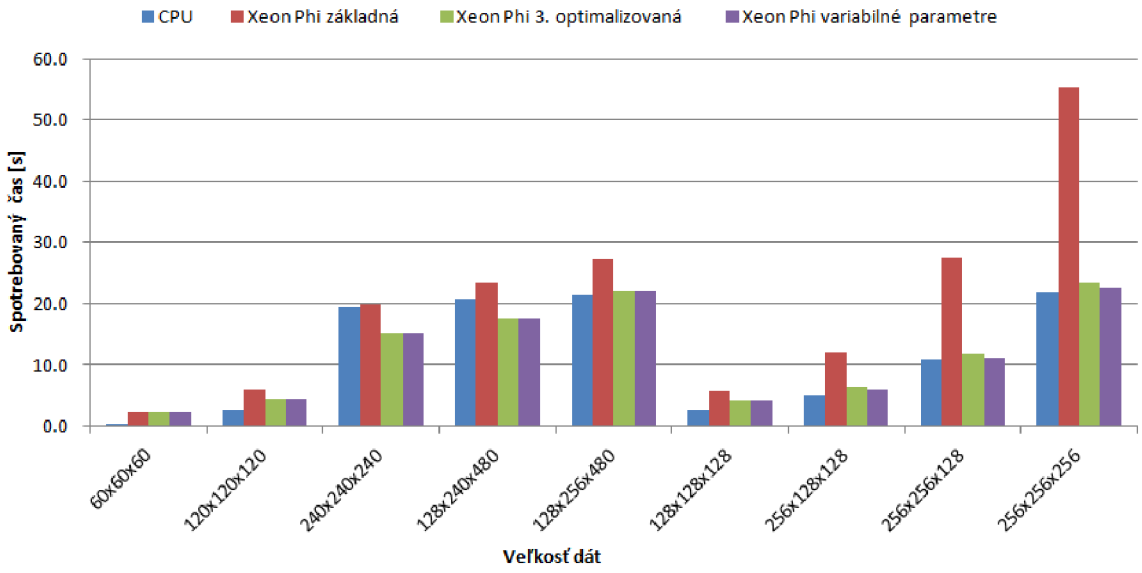
Obrázok 6.16 zobrazuje vplyv afinity a počtu vlákien na čas týchto výpočtov. Keď ho porovnáme s obrázkom 6.15, môžeme vidieť, že najlepšie výsledky nie sú dosiahnuté pri rovnakom nastavení

parametrov afinity a počtu vlákien. Ďalšie malé zlepšenie výkonu je možné dosiahnuť nastavením týchto parametrov oddelene pre FFT výpočty používajúce knižnicu MKL a ostatné paralelizované výpočty – použitie je popísané v sekcii 5.1 odstavci Kompilácia a spustenie. Takto odhadnutý celkový čas zobrazuje nasledovný graf na obrázku 6.17.



Obrázok 6.17 Porovnanie celkového času optimalizovanej verzie na Intel Xeon Phi s použitím rovnakých (pevné parametre) a rôznych (variabilné parametre) počtov vlákien pre výpočty FFT a pre ostatné výpočty.

Teraz zanedbáme dobu zápisu dát a porovnáme len časy samotných simulačných výpočtov – zobrazené na obrázku 6.18. Niektoré ďalšie dáta by sa takmer vyrovnali simulačným časom CPU verzii. A pri dátach o veľkosti *240x240x240* by bol simulačný čas od nej lepší už takmer o *23 %*.



Obrázok 6.18 Porovnanie celkového času simulácie na bežnom procesore a na Intel Xeon Phi pred a po optimalizáciách všetkých metód a s použitím variabilných parametrov.

6.4 Zhodnotenie experimentov

Pri pohľade na jednotlivé prevedené optimalizácie môžeme povedať, že boli celkom úspešné. Zoptimalizované metódy získali niekoľkonásobne vyšší výkon a v najlepších prípadoch sú viac ako dvojnásobne rýchlejšie oproti bežnému procesoru, s ktorým sme robili porovnávanie. Z pohľadu celkového dosiahnutého zrýchlenia sme dosiahli najlepší výsledok pri simulácii dát veľkosti *240x240x240*, presne ako sme predpokladali na začiatku. Predstavuje to zrýchlenie o *12 %* (*23 %* pokiaľ nepočítame ukladanie dát). To je niečo viac ako *2* sekundy (takmer *4,5* sekundy keď nepočítame ukladanie dát) pri *100* simulačných krokoch. Zdá sa to ako malá hodnota. Avšak pri simulácii sa v praxi používa minimálny počet krokov rovný dĺžke telesovej uhlopriečky simulovaného prostredia. Pre tieto dáta to predstavuje približne *416* krokov. Celkový čas simulácie v tomto prípade dosiahne okolo *72,5 s* (skoro *63 s* bez ukladania dát) a ušetrený čas bude viac ako *8* sekúnd (viac ako *18,5 s* bez ukladania dát). To už nevyzerá ako úplne malá hodnota a pri simuláciách vyžadujúcich ešte viac krokov, bude ušetrený čas ešte markantnejší. Bohužiaľ nie je možné dosiahnuť takého zrýchlenie pre akékoľvek dáta. Sú za to zodpovedné tri veci: Hlavnou príčinnou sú veľké výkyvy vo výkone Fouriérových transformácií pre rôzne veľkosti dát. Aj druhá príčina sa týka veľkosti dát. Pre dáta, ktorých celková veľkosť nie je násobkom počtu jadier karty Xeon Phi, sme zaznamenali prepady výkonu vo zvyšných výpočtoch (t.j. bez FFT) aplikácie. Posledným problémom spôsobujúcim zníženie výkonu, je veľmi pomalé ukladanie dát do výstupného súboru, ktoré je závislé na množstve použitých vlákien. Riešením môže byť ďalšie rozsiahle experimentovanie s dátami rozličných rozmerov, ktorého výstupom bude napríklad tabuľka zobrazujúca rozmery dát, pre ktoré dokáže karta Xeon Phi poskytnúť zrýchlenie. Táto tabuľka by bola využívaná ako pomôcka, pomocou ktorej by sa určovali rozmery dát, alebo na ktoré by sa existujúce dáta zarovnali použitím výplne.

Pri experimentovaní sme používali dáta rôznych rozmerov, ale iba pri dátach veľkých rozmerov sme spozorovali niektoré zaujímavé skutočnosti. Napríklad pri použití dát o veľkosti *512x256x256* sme zaznamenali prudký prepad výkonu pri FFT. Podozrievame knižnicu MKL, že má pri zníženom množstve dostupnej pamäti (ktorej ale podľa štatistiky stále bolo voľnej asi *3 GB*) nejaké problémy. Pamäť podozrievame preto, že aj pri iných dátach sa objavili problémy s výpočtom FFT. Pri dátach s rozmermi *192x480x480* simulácia prebehla, ale pri meraní výkonu sme zistili, že počítanie FFT vôbec neprebehlo. Čas výpočtu bol skutočne len minimálny a overovaný výstup simulácie bol úplne chybný. Zarážajúce je to, že aplikácia vôbec nespadla, tak ako sa to dialo v prípade dát o veľkosti *512x512x256*. Tie už narážali na pamäťový strop karty Xeon Phi a aplikácia spadla buď pri alokovaní zdrojov, alebo priamo počas simulácie.

Skúšali sme aj použitie iných optimalizácií, napríklad alokáciu pamäti použitím veľkých stránok alebo implementáciu jednotlivých optimalizácií použitím intrinsic inštrukcií, oproti prevedeným optimalizáciám však zlepšenie nepriniesli.

7 Záver

7.1 Zhrnutie diplomovej práce

Diplomová práca sa venuje paralelizácii ultrazvukových simulácií na akceleračnej karte Intel Xeon Phi. Samotné simulácie sú počítané aplikáciou k-Wave toolbox, ktorú skutočne využívajú ľudia v praxi. V práci je popísaná jej funkcionalita, simulačný model a ďalej všetky jej dostupné implementácie. Následne je popísaná architektúra akceleračnej karty Intel Xeon Phi, ktorú je potrebné poznať pre jej efektívne využitie. Popísané sú aj všetky podporované programovacie modely, ktoré akcelerátor podporuje, a rovnako aj programové prostriedky, ktoré sú dostupné pre implementovanie paralelizácie do programovaných aplikácií. Na základe všetkých získaných informácií sme navrhli postup úpravy a následnej optimalizácie aplikácie k-Wave, ktorá bude používať natívny programovací model akcelerátoru. Podľa navrhnutého postupu sa nám podarilo aplikáciu upraviť a úspešne spustiť na akcelerátore, pričom nebolo potrebné vykonať mnoho úprav a najhoršie bolo nájsť správny spôsob preloženia použitých knižníc. Po overení korektnosti jej výpočtu, sme pomocou analýzy v programe Intel VTune určili kandidátne metódy pre následnú optimalizáciu. Tá priniesla dobré výsledky, ktoré sme merali pri veľkom množstve prevedených experimentov. Jednoznačne najväčší prínos vo výkone priniesla paralelizácia a vektorizácia – dva najväčšie benefity celej platformy. Najväčšie zrýchlenie, ktoré sa nám podarilo dosiahnuť oproti bežnému procesoru, dosahuje asi *23 %*. To je však degradované veľmi pomalým zápisom dát na úroveň okolo *12 %*. Dobré zrýchlenie je možné dosiahnuť len pri určitej veľkosti dát, pretože ich veľkosť ma veľký dopad na celkový výkon aplikácie a hlavne na výpočet Fourierových transformácií. Tie celkovo v aplikácii prevažovali a po prevedení optimalizácií sa ich dominancia ešte zväčšila. Obrovský dopad na výkon má aj počet použitých vlákien, ktoré bežia paralelne a tiež aj nastavenie ich priraďovania jednotlivým jadrám akcelerátora. Všetky tieto veci sa prejavili počas rozsiahlych experimentov a boli zdokumentované.

7.2 Budúcnosť práce

Prevádzanie simulačných experimentov je veľmi časovo náročné a nebolo preto možné zmerať úplne všetko. Zostalo mnoho priestoru pre ich pokračovanie. Určite by sme sa mali zamerať na ďalšie experimentovanie s veľkým množstvom dát rôznych veľkostí. Následne by bolo potrebné určiť tie veľkosti, ktoré skutočne prinášajú efektívne zrýchlenie. Tiež by sme mohli použiť iný typ akcelerátoru s väčšou pamäťou, aby sme mohli experimentovať aj s väčšími dátami. Zaujímavé by bolo aj implementovanie alternatívnych programovacích modelov akcelerátora a následne

ich porovnanie. Obzvlášť MPI verzie, ktorá by bola schopná využiť celý cluster s akcelerátormi Xeon Phi a plne preveriť schopnosti platformy. Ďalej by sme zamerali na riešenie problému s pomalým zápisom dát, ktoré dosť degraduje výkonnosť a vyskúšali použitie niektorej paralelnej implementácie zápisu.

Literatúra

[1] TREEBY Bradley E., JAROŠ Jiří, RENDELL Alistair P. a COX Ben T. Modeling nonlinear ultrasound propagation in heterogeneous media with power law absorption using a k-space pseudospectral method. *J. Acoust. Soc. Am.*, roč. 131, č. 6, 2012, s. 4324-4336

[2] K-WAVE. A MATLAB toolbox for the time-domain simulation of acoustic wave fields. *K-wave.org* [online]. ©2010-2014, [cit. 2015-20-05]. Dostupné z: http://www.k-wave.org

[3] INTEL CORPORATION. Homepage of the Intel® Xeon Phi™ product family. *Intel.com* [online]. ©2014 [cit. 2015-20-05]. Dostupné z: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

[4] TREEBY Bradley E., COX Ben T. a JAROŠ Jiří. *k-Wave User Manual* [online]. ©2010-2014, [cit. 2015-20-05]. Dostupné z: http://www.k-wave.org/manual/k-wave_user_manual_1.0.1.pdf

[5] TREEBY Bradley E. a COX Ben T. Modeling power law absorption and dispersion for acoustic propagation using the fractional Laplacian, *J. Acoust. Soc. Am.,* roč. 127, č. 5, pp. 2741-2748, 2010

[6] JAROŠ Jiří, RENDELL Alistair P. a TREEBY Bradley E. Full-wave nonlinear ultrasound simulation on distributed clusters with applications in high-intensity focused ultrasound. *The International Journal of High Performance Computing Applications*. 2455 Teller Road Thousand Oaks, CA 91320: SAGE Publications, 2015, roč. 2015, č. 2, s. 1-19.ISSN 1741-2846.

[7] JEFFERS Jim a REINDERS James. *Intel Xeon Phi Coprocessor High-performance Programming*. Elsevier, 2013. ISBN 9780124104143.

[8] INTEL CORPORATION. *Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual* [online]. ©2012, [cit. 2015-20-05]. Dostupné z: https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf

[9] DVOŘÁK Václav a JAROŠ Jiří. *Architektura a programování paralelních systémů*, prezentace k předmětu. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/ARC-IT/lectures?cid=9234

[10] INTEL CORPORATION. Intel® Xeon Phi™ Coprocessor System Software Developers Guide. *Intel.com* [online]. ©2014, [cit. 2015-20-05]. Dostupné z: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-system-software-developers-guide

[11] OPENMP ARB. *OpenMP Application Program Interface: Version 4.0* [online]. ©1997-2013, [cit. 2015-20-05]. Dostupné z: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[12] CHAPMAN Barbara, JOST Gabriele a PAS Ruud van der. *Using OpenMP: portable shared memory parallel programming*. Cambridge, Mass.: MIT Press, 2008, xxii, 353 p. ISBN 0262033771.

[13] INTEL CORPORATION. Building a Native Application for Intel® Xeon Phi™ Coprocessors. *Intel.com* [online]. ©2014, [cit. 2015-20-05]. Dostupné z: https://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors

[14] JAROŠ Jiří. Large-scale Ultrasound Simulations in Human Body, *Prezentace pro IT4I.cz*, [cit. 2015-20-05].

[15] INTEL CORPORATION. Intel® Math Kernel Library (Intel® MKL) on Intel® Many Integrated Core Architecture (Intel® MIC Architecture). *Intel.com* [online]. ©2012, [cit. 2015-20-05]. Dostupné z: https://software.intel.com/en-us/articles/intel-mkl-on-the-intel-xeon-phi-coprocessors

[16] REINDERS James a JEFFERS Jim. *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*. 2014. Elsevier / Morgan Kaufmann. ISBN 9780128021996.

[17] PRACE (Partnership for Advanced Computing in Europe). Best Practice Guide – Intel Xeon Phi. *Prace-ri.eu* [online]. ©2015, [cit. 2015-20-05]. Dostupné z: http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/

[18] INTEL CORPORATION. Intel® Many Integrated Core Architecture (Intel MIC Architecture) Forum. *Intel.com* [online]. ©2015, [cit. 2015-20-05]. Dostupné z: https://software.intel.com/en-us/forum/37014

[19] IT4INNOVATIONS, Národní superpočítačové centrum. Anselm Cluster Documentation. *It4i.cz* [online]. ©2015, [cit. 2015-20-05]. Dostupné z: https://docs.it4i.cz/anselm-cluster-documentation

[20] LINUX4HIPPOS. Cross compile HDF5 for Intel Xeon Phi (MIC). *Blogspot.cz* [online]. 2013-13-05 [cit. 2015-20-05]. Dostupné z: http://linux4hippos.blogspot.cz/2013/05/cross-compile-hdf5-for-intel-xeon-phi.html

[21] FANG Jianbin, VARBANESCU Ana Lucia, SIPS Henk, ZHANG Lilun, CHE Yonggang, XU Chuanfu. *An Empirical Study of Intel Xeon Phi* [online]. 2013-20-12 [cit. 2015-20-05]. Dostupné z: http://arxiv.org/pdf/1310.5842v2.pdf

Príloha A

Obsah DVD

\bin	binárne súbory
\data	testovacie dáta
\src	zdrojový kód
\text	text práce