Prírodovedecká fakulta UPJŠ v Košiciach
Ústav informatiky

Distribuované výpočty s využitím grafických akcelerátorov

DIPLOMOVÁ PRÁCA

Košice 2008 Samuel KUPKA

Prírodovedecká fakulta Univerzity Pavla Jozefa Šafárika v Košiciach
Ústav informatiky

Distribuované výpočty s využitím grafických akcelerátorov

Diplomová práca

Samuel Kupka

Košice, apríl 2008

Vedúci: RNDr. Jozef Jirásek, PhD.

Abstrakt

Rozvoj herného priemyslu podnietil aj rozvoj herného hardvéru, ktorý výpočtovým výkonom často prekonáva možnosti najnovších univerzálnych procesorov. Je preto logické, že sa mnohí snažia dostupné prostriedky čo najviac využiť aj v oblastiach, na ktoré nebol pôvodne herný hardvér určený. Ako jednou z vhodných oblastí sa ukazuje simulácia prírodných dejov. Samotní výrobcovia podávajú vývojárom pomocnú ruku v podobe kvalitných a dobre zdokumentovaných vývojových prostredí. V mojej práci využívam moderné grafické akcelerátory na simuláciu šírenia trhliny v krehkých materiáloch.

Abstract

Computational power of current graphics accelerators surpases current standard processors by far. It's only natural, that many research groups are trying to use all available resources to speed-up their applications by using all available resources. One of the most promising fields seems to be simulation of natural processes. Developers of graphics accelerators are even trying to help boost the process by releasing high quality programming environments and applications. This document describes implementation of simulation of fracture in brittle materials on graphics accelerators.

Táto strana je úmyselne prázdna a má byť nahradená originálnym zadaním.

Vyhlásenie

Vyhlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím uvedenej literatúry.

. . . . . .

Samuel Kupka

Poďakovanie

Moje poďakovanie patrí RNDr. Jozefovi Jiráskovi, PhD.[a], vedúcemu diplomovej práce, doc. RNDr. Jozefovi Uličnému, CSc.[b] a Prof. Aattovi Laaksonenovi[c].

[a]Ústav informatiky Prírodovedeckej fakulty UPJŠ v Košiciach

[b]Ústav biofyziky Prírodovedeckej fakulty UPJŠ v Košiciach

[c]Department of Physical, Inorganic & Structural Chemistry, Stockholm University

Obsah

Úvod

Rozvoj herného priemyslu podnietil aj rozvoj herného hardvéru, ktorý výpočtovým výkonom často prekonáva možnosti najnovších univerzálnych procesorov. Je preto logické, že sa mnohí vývojári snažia dostupné prostriedky čo najviac využiť aj v oblastiach, na ktoré nebol pôvodne herný hardvér určený. Ako jednou z vhodných oblastí sa ukazujú rôzne simulácie prírodných dejov.

Asi najrozvinutejšou oblasťou sú simulácie molekulárnej dynamiky [4, 7], na ktorej závisí výskum v mnohých oblastiach biochémie a pridružených odborov (výroba liečiv, skúmanie chorôb). Práve v molekulárnej dynamike sa dá naplno zužitkovať vysoký výkon grafických kariet. Vďaka cenovej dostupnosti sa výskumu môže venovať aj mnoho menších tímov, čo posúva celú skúmanú oblasť rýchlo dopredu. Ďalšími skúmanými oblasťami sú pevnosť a odolnosť materiálov (metóda konečných prvkov), priemyselný dizajn (rozloženie záťaže, opotrebovania), ale aj mnohé oblasti, s ktorými prichádzajú do styku aj bežní ľudia, ako zrýchlenie šifrovania, kompresie videa a zvuku, či zrýchlenie behu samotného operačného systému. V mojej práci sa budem venovať simulácii šírenia trhliny v materiáloch.

Šírenie trhlín v krehkých materiáloch je jedným z dejov, ktoré už boli úspešne implementované na bežných počítačoch. Išlo však o simuláciu dvojrozmerných mriežok malých rozmerov. Hlavným cieľom mojej práce bolo implementovať šírenie trhliny v trojrozmernej mriežke, ktorá sa svojimi rozmermi približuje veľkosti experimentálne skúmaných vzoriek. Za cieľ bola určená mriežka s rozmermi $1000 \times 1000 \times 500$ zŕn.

V mojej práci využívam grafickú kartu NVIDIA na simuláciu šírenia trhliny v krehkých materiáloch so štvorcovou mriežkou. Ako programovací jazyk je zvolený CUDA od spoločnosti NVIDIA. Výpočet na grafickej karte (alebo GPU) prináša oproti bežnému výpočtu na CPU niekoľko obmedzení, ktoré sú dané použitou architektúrou a pôvodnou špecializáciou hardvéru. Tejto téme je venovaná kapitola 1.

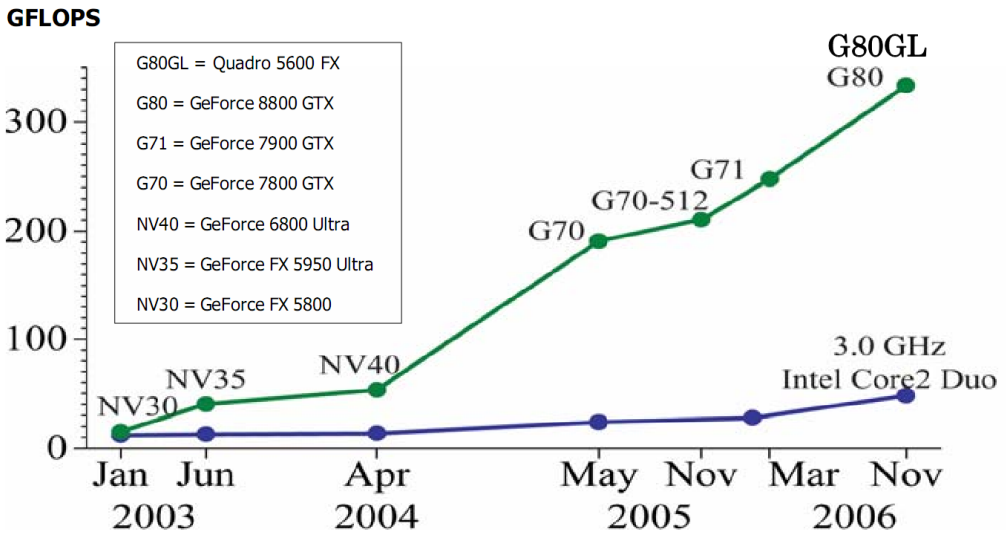
V kapitole 2 popisujem jednoduchý dvojrozmerný a trojrozmerný model interkryštalického lomu a spôsob jeho simulácie vo forme sekvenčne vykonávaných krokov. Algoritmus popísaný v tejto kapitole budem ďalej považovať za referenčný. V kapitole 3 sa venujem niektorým krokom referenčného algoritmu a snažím sa navrhnúť zmeny, ktoré umožnia ich paralelné vykonávanie. Zároveň zdôvodňujem možnosť vykonania jednotlivých zmien. Kapitola 4 je venovaná samotnej implementácii krokov popísaných v kapitole 3 v jazyku CUDA. Implementácia vo forme spustiteľného programu a zdrojových kódov je dostupná aj na priloženom CD.

1 Hardvérový model, špecifiká CUDA

Táto kapitola obsahuje výťažok z publikácie [2]. Obsahuje len časti, ktoré priamo využívam pri implementácii a sú potrebné na jej pochopenie. Mnohé špecifiká architektúry CUDA však zostanú nepopísané, keďže spadajú mimo záber tejto diplomovej práce.

1.1 Grafický procesor ako paralelné výpočtové zariadenie

V priebehu niekoľkých ostatných rokov sa z grafického procesora (alebo GPU) stalo vysoko výkonné výpočtové zariadenie. Vďaka viacerým jadrám a vysokorýchlostnej pamäťovej zbernici ponúkajú dnešné grafické karty neuveriteľný zdroj výpočtového výkonu pre grafické ale aj rôzne iné aplikácie.

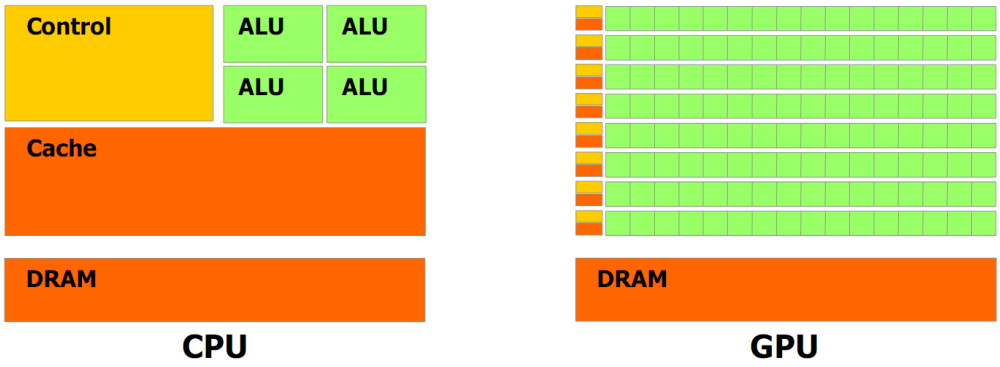


Obr. 1: Počet operácií s pohyblivou desatinnou čiarkou za sekundu na CPU a GPU

Samotné renderovanie 3D grafiky je proces, ktorý je veľmi dobre paralelizovateľný a tak sa vývoj GPU uberá práve smerom pridávania jadier a zvyšovania výkonu matematických výpočtov. Vďaka masovej výrobe je cena grafických kariet prijateľná.

1.2 CUDA - Nová architektúra pre výpočty na GPU

CUDA, predstavená spoločnosťou NVIDIA, je nová softvérová a hardvérová architektúra podporujúca využívanie grafických kariet na ľubovoľné výpočty bez nutnosti pristupovať ku GPU cez grafické rozhranie. Táto architektúra je dostupná pre grafické karty od NVIDIA GeForce8 a viacero zariadení dodávaných spomínanou spoločnosťou. Výhodou je možnosť zdieľať prostriedky dostupné na grafickej karte medzi viaceré CUDA procesy



Obr. 2: Rozdielna štruktúra CPU a GPU

bežiace súčasne, bežné operácie operačného systému, ako vykresľovanie obrazu a dokonca aj grafické programy, ktoré využívajú rozhranie Direct3D alebo OpenGL.

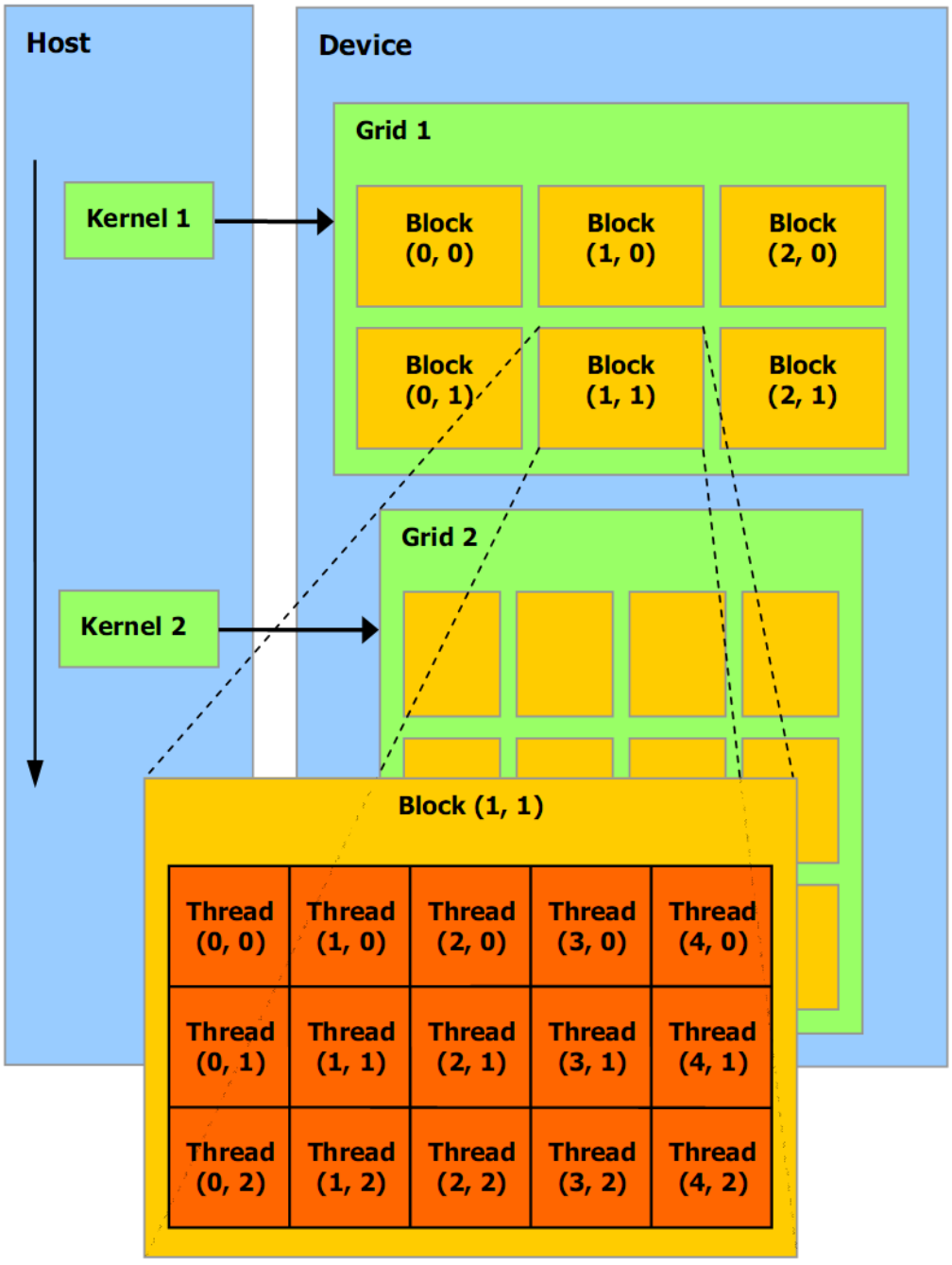
Samotné CUDA API je koncipované ako rozšírenie programovacieho jazyka C. Spôsob integrácie jazyka C a CUDA API je popísaný v publikácii [1].

1.3 Vlákna

Všetky vlákna, ktoré sú spúšťané na grafickej karte, sú usporiadané do štruktúry, ktorá je optimalizovaná pre GPU. Každé vlákno je inštanciou rovnakej funkcie, ktorá je určená na beh na GPU. Anglicky je táto funkcia označovaná ako *kernel* a v programe je označená direktívou *__global__*. Počet vlákien, ktoré dokáže grafická karta súčasne obsluhovať, je obmedzený. Množinu týchto vlákien budem označovať *blok*, anglicky je označovaná *block*. V rámci jedného bloku je možné vlákna navzájom synchronizovať. Veľkosť bloku je zároveň možné pri inicializácii špecifikovať (samozrejme len do veľkosti podporovanej kartou samotnou). Samotné bloky je možné usporiadať do mriežky. Z dôvodu použitia slova mriežka pre popis mriežky skúmaného materiálu budem ďalej využívať anglické pomenovanie *grid*. Bloky, ktoré sú súčasťou jedného gridu sú vykonávané sekvenčne. Samotnú veľkosť gridu je taktiež možné špecifikovať pri inicializácii. V každom vlákne je možné zistiť aktuálnu pozíciu v rámci bloku a gridu. Slúžia na to globálne premenné *threadIdx* a *blockIdx*. Celá štruktúra je znázornená na obrázku 3 na strane 11.

Architektúra CUDA neimplementuje kritické oblasti, no umožňuje istú synchronizáciu vlákien pomocou volania funkcie *__syncthreads()*. Ak vykonávané vlákno narazí na volanie tejto funkcie, ukončí vykonávanie a čaká, kým všetky ostatné vlákna z toho istého bloku nenarazia na túto funkciu. V tom momente všetky vlákna pokračujú ďalej vo vykonávaní inštrukcií. Funkciu je možné použiť aj v podmienke (alebo ju inak vnoriť), je však potrebné

zabezpečiť, aby sa jej volanie určite vykonalo v rámci každého vlákna v bloku. Inak môže nastať zastavenie vykonávania alebo nedefinovaný stav.



Obr. 3: Štruktúra vlákien na GPU

1.4 Dátové typy

Grafická karta dokáže priamo spracovávať niektoré vektorové dátové typy, ktoré sú následne dostupné aj v prostredí CUDA.

- Jednoprvkové vektory - *char1, uchar1, short1, ushort1, int1, uint1, long1, ulong1, float1*
- Dvojprvkové vektory - *char2, uchar2, short2, ushort2, int2, uint2, long2, ulong1, float2*
- Trojprvkové vektory - *char3, uchar3, short3, ushort3, int3, uint3, long3, ulong3, float3, dim3*
- Štvorprvkové vektory - *char4, uchar4, short4, ushort4, int4, uint4, long4, ulong4, float4*

Jednotlivé prvky vektorov majú názov *x, y, z* a *w*, podľa veľkosti vektora. Dátový typ *dim3* je odvodený od *uint3*, ale všetky prvky sú inicializované na hodnotu 1.

V programe je možné tieto dátové typy vytvárať funkciami *make_(typ)*, kde *(typ)* je názov jedného z typov.

1.5 Atomické funkcie

Prostredie CUDA od verzie 1.1 ponúka aj niekoľko atomických funkcií. V implementácii využívam nasledujúce:

- *int atomicAdd(int *address, int val)* - Načíta premennú *old* nachádzajúcu sa na adrese *address* v globálnej pamäti, vypočíta (*old*+*val*), uloží výsledok na adresu *address* a vráti hodnotu *old*. Všetko sa udeje v rámci jedinej atomickej transakcie.
- *int atomicSub(int *address, int val)* - Načíta premennú *old* nachádzajúcu sa na adrese *address* v globálnej pamäti, vypočíta ($old - val$), uloží výsledok na adresu *address* a vráti hodnotu *old*. Všetko sa udeje v rámci jedinej atomickej transakcie.
- *int atomicOr(int *address, int val)* - Načíta premennú *old* nachádzajúcu sa na adrese *address* v globálnej pamäti, vypočíta (*old*|*val*), uloží výsledok na adresu *address* a vráti hodnotu *old*. Všetko sa udeje v rámci jedinej atomickej transakcie.

Popis všetkých atomických funkcií nájdete v publikácii [2], príloha C.2.

2 Jednoduchý model interkryštalického lomu

V dnešnej dobe bežne používame rôzne technické zariadenia a viac-menej prirodzene očakávame, že budú dlho, spoľahlivo a bezporuchovo fungovať. Konštruktéri musia preto zabezpečiť aby všetky komponenty boli dostatočne pevné a nezlomili sa počas životnosti zariadenia. Porušenie a lom materiálov majú vďaka tomu veľký praktický význam a už mnoho rokov sú predmetom intenzívneho štúdia.

2.1 Základné pojmy, Griffithova teória

Pri navrhovaní nových produktov je pre inžinierov dôležité vedieť ako použité materiály reagujú na vonkajšie podnety. V prípade mechanického zaťaženia sú hlavnými odozvami materiálu deformácia a lom.

Ak sú pôsobiace sily malé, materiál sa deformuje a deformácia môže byť pružná. To znamená, že deformovaný kus materiálu opäť nadobudne pôvodný nedeformovaný tvar a rozmery keď vonkajšie sily prestanú pôsobiť. Vyššie napätia môžu viesť k plastickej deformácii. Ak sa materiál zdeformuje plasticky, po odstránení vonkajšieho zaťaženia sa už nevráti do svojho pôvodného tvaru.

Ak vonkajšie sily spôsobia rozdelenie telesa na dva alebo viac kusov, hovoríme, že došlo k lomu. Lom môže byť tvárny alebo krehký v závislosti od intenzity a typu deformácie, ktorá ho predchádza. Ak lom nastane skôr ako dôjde k významnej plastickej deformácii, hovoríme, že materiál je krehký. Rozvoj modernej lomovej mechaniky začal vo chvíli keď si ľudia uvedomili, že odolnosť reálnych materiálov proti lomu (ich pevnosť) súvisí s poruchami ktoré sa v takýchto materiáloch prirodzene vyskytujú. Prelomovým bodom bolo Griffithovo štúdium stability makroskopického telesa s počiatočnou trhlinou [3].

Uvažujme veľkú dosku z krehkého materiálu, ktorá je v stave homogénnej rovinnej deformácie vyvolanej priloženým jednoosím ťahovým napätím σ pôsobiacim v rovine dosky. Energia pružnej deformácie akumulovaná v objemovej jednotke takejto dosky je $\sigma^2/2E$, kde E je modul pružnosti (Youngov modul) materiálu dosky. Vytvorme v takejto doske trhlinku dĺžky $2a$, ktorá prechádza celou hrúbkou t danej dosky. Griffith predpokladal, že energia pružnej deformácie akumulovaná v doske s trhlinou je nižšia ako zodpovedajúca energia akumulovaná v doske bez trhliny. Analýzou daného problému Griffith ukázal, že úbytok energie pružnej deformácie akumulovanej v doske, ktorý je spôsobený vytvorením trhliny, je rovný

$$\Delta U_{\text{elast}} = -\frac{\pi a^2 t \sigma^2}{E} \tag{1}$$

Veľmi zjednodušene si situáciu možno predstaviť tak, že pri vytvorení trhliny s dĺžkou $2a$ sa uvoľní energia pružnej deformácie pôvodne naakumulovaná zhruba v objeme valca s priemerom podstavy $2a$ a výškou t.

Griffith ďalej predpokladal, že vytvorenie nových voľných povrchov trhliny vyžaduje absorbciu množstva energie daného výrazom

$$\Delta U_{\text{surf}} = 4at\gamma \tag{2}$$

kde γ je povrchová energia na jednotku plochy.

Ak pri raste dĺžky trhliny je rýchlosť akou je energia absorbovaná tvorením nových voľných povrchov vyššia ako je rýchlosť, s ktorou je uvoľňovaná energia pružnej deformácie, teleso s trhlinou je stabilné. Ak je však energia pružnej deformácie uvoľňovaná rýchlejšie ako môže byť absorbovaná novými povrchmi, zaťažené teleso s trhlinou je nestabilné, trhlina sa neobmedzene šíri až dôjde k porušeniu materiálu. Pôvodná Griffithova teória sa ukázala vhodná pre krehké materiály. Po istých úpravách, napríklad po rozšírení člena popisujúceho energiu potrebnú na vytvorenie nového povrchu o prácu absorbovanú plastickou deformáciou, teória bola schopná popísať správanie širšej triedy materiálov.

2.2 Modelovanie lomu na počítači, konštrukcia vhodného modelu

Rozlomenie reálneho materiálu je výsledkom komplikovanej súhry rôznych nerovnovážnych a nelineárnych procesov, ktoré prebiehajú často diametrálne odlišnými rýchlosťami a na rôznych úrovniach, od atómov cez mikroštruktúru až po makroskopické telesá. Táto komplexnosť problému, široké spektrum časových intervalov a priestorových škál spôsobujú, že porušenie a lom nie sú doposiaľ úplne pochopené [6].

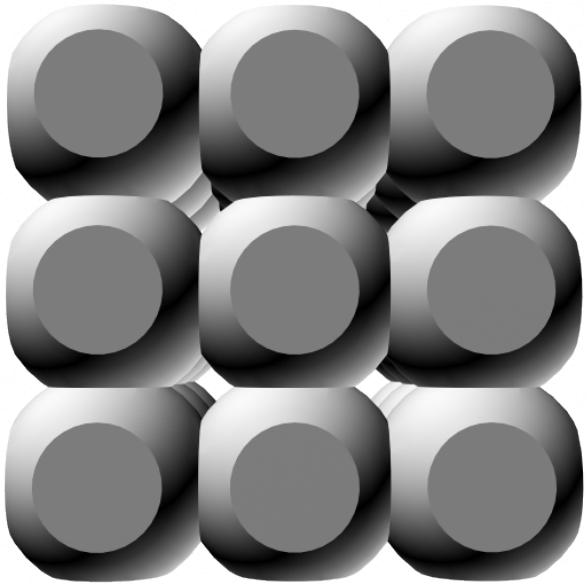
Prínosom je preto každá cesta alebo spôsob umožňujúci získať v tejto oblasti ďalšie informácie. Jednou z možností, ako lepšie pochopiť dané javy, sú simulácie na počítačoch. V takomto virtuálnom experimente je reálny systém nahradený vhodným matematickým modelom. Tento model zvyčajne predstavuje kompromis medzi jednoduchosťou, umožňujúcou efektívne a optimálne využitie počítačov, a čo možno najvernejším napodobením reálneho systému s jeho charakteristickými črtami, aby získané výsledky boli netriviálne a dostatočne reprezentatívne.

Z hľadiska náuky o materiáloch možno vždy namietať, že tieto zjednodušené modely neposkytujú najrealistickejšiu reprezentáciu porušenia a lomu materiálov. V súčasnosti však nie je prakticky možné vytvoriť realistický všeobecný model, ktorý možno aplikovať

na širokú triedu materiálov a podmienok. Výsledky získané pomocou jednoduchých modelov poskytujú určitý základ pre pochopenie správania reálnych materiálov a slúžia ako východisko pre vývoj realistickejších (ale nevyhnutne i komplikovanejších a na výpočtovú techniku náročnejších) modelov.

Niektoré polykryštalické materiály alebo vzorky pripravené práškovou metalurgiou majú krehké hranice zŕn, čo predstavuje ľahkú cestu pre šírenie trhliny. Krehkosť hraníc zŕn je buď vlastná danému materiálu, alebo ju spôsobuje segregácia prímesí a nečistôt či dokonca vytvorenie tenkého filmu krehkej sekundárnej fázy na hraniciach zŕn.

Kvôli štúdiu a simulácii šírenia trhliny v polykryštalických materiáloch s krehkým interkryštalickým lomom bol navrhnutý nasledujúci jednoduchý model [5].



Obr. 4: Zrná tvoriace polykryštalický materiál

Predpokladajme, že vzorku tvorí agregát navzájom spojených zŕn. Jednoduchým, počítačom spracovateľným modelom takejto vzorky je množina bodov $\{(i, j, k)\}$ označovaná aj ako mriežka, v ktorej sú umiestnené dvojhodnotové premenné ξ_{ijk} označované ako väzby (hranice). Každý z bodov modelovej množiny je situovaný do rozhrania medzi dvoma susediacimi zrnami a hodnota premennej ξ informuje, či sú tieto zrná navzájom spojené ($\xi = 0$), alebo sú od seba odtrhnuté ($\xi = 1$). Danému stavu reálnej vzorky potom

zodpovedá istá konfigurácia hodnôt premenných ξ_{ijk} (rozloženie hodnôt 0 a 1 na danej modelovej množine bodov). Každej konfigurácii hodnôt premenných ξ_{ijk} možno priradiť energiu zodpovedajúcu danému stavu vzorky.

V matematickom výraze pre energiu systému vystupujú jednak premenné ξ_{ijk}, jednak číselné parametre nezávislé od ξ_{ijk}. Funkcionálna závislosť energie od premenných ξ_{ijk} odráža štruktúru reálneho systému (počet a priestorové rozmiestnenie najbližších susedov typického zrna) a ukazuje aké formy energie sa berú do úvahy (povrchová energia, energia pružnej deformácie, a pod.). Číselné hodnoty parametrov súvisia s materiálovými vlastnosťami modelovaného systému a s vonkajšími podmienkami (veľkosť zŕn, hustota povrchovej energie, elastické moduly, veľkosť pôsobiaceho napätia, teplota, a pod.).

Šíreniu trhliny v reálnom systéme zodpovedá vo virtuálnom experimente prechod od jednej konfigurácie hodnôt premenných ξ_{ijk} k inej konfigurácii s vyšším počtom jednotiek. Realizácia tohto prechodu je riadená pravdepodobnosťou

$$Pr = \theta(u_{\text{stara}} - u_{\text{nova}}) + \theta(u_{\text{nova}} - u_{\text{stara}}) \cdot e^{\frac{-(u_{\text{nova}} - u_{\text{stara}})}{k_B T}} \tag{3}$$

Kde u_{nova} (u_{stara}) je hodnota energie priradená novej (starej) konfigurácii hodnôt premenných ξ_{ijk}, k_B je Boltzmanova konštanta a T je absolútna teplota. Funkcia $\theta(x)$ je Heavisideova funkcia ($\theta(x) = 0$ pre $x < 0$, $\theta(x) = 1/2$ pre $x = 0$, $\theta(x) = 1$ pre $x > 0$).

2.3 Simulácia lomu dvojrozmerného systému, demonštrácia použiteľnosti modelu

Jednoduchá verzia tohto modelu bola úspešne použitá na štúdium a ilustráciu niektorých čŕt krehkého lomu. Uvažoval sa systém tvorený rovnakými zrnami usporiadanými do štvorcovej mriežky. Potom aj body modelovej množiny s premennými ξ_{ij} sú v uzloch štvorcovej mriežky. Na priradenie hodnoty energie danej konfigurácii hodnôt premenných ξ_{ij} bola použitá Griffithova teória. V tom prípade matematický výraz pre energiu obsahoval dva členy. Prvý, ktorý má tvar

$$u_{\text{povrchová}} = K_1 \times (\text{počet roztrhnutých väzieb medzi zrnami}) \tag{4}$$

zodpovedá hore uvedenému výrazu (2) v Griffithovej teórii a reprezentuje energiu voľných povrchov vznikajúcich pri oddeľovaní zŕn interkryštalickým krehkým lomom. Parameter K_1 v sebe koncentruje veličiny ako hustota povrchovej energie, priemerná plocha kontaktov medzi zrnami, no i veličinu $1/k_B T$, keďže táto energia je použitá na výpočet pravdepodobnosti podľa (3).

Druhý člen, ktorý zodpovedá výrazu (1) v Griffithovej teórii a reprezentuje energiu elastickej deformácie uvoľnenú trhlinou, má tvar

$$u_{\text{elastická}} = -K_2 \times (\text{počet roztrhnutých neprekrývajúcich sa horizontálnych väzieb})^2 \quad (5)$$

pričom ťah pôsobí vertikálne. Parameter K_2 opäť predstavuje viac veličín, a to priemerný objem zrna, Youngov modul pružnosti, veľkosť pôsobiaceho napätia, tvarový faktor a samozrejme veličinu $1/k_BT$.

Výber číselných hodnôt pre K_1 a K_2 riadil rýchlosť šírenia trhliny, trajektóriu lomu a morfológiu lomových plôch pri simuláciách prezentovaných v práci [5].

Jednotlivé kroky simulácie boli:

1. Na začiatku simulácie sa vybrali hodnoty parametrov K_1, K_2 a každému mriežkovému bodu sa priradila hodnota $\xi_{ij} = 0$, $i = 1...N$, $j = 1...N$

2. Roztrhla sa ľubovoľná väzba, t.j. $\xi_{km} = 1$. Podľa hore uvedených vzorcov sa vypočítala energia vzorky s touto jednou roztrhnutou väzbou, ktorá sa označila u_{stara}

3. Náhodne sa vybral jeden z bodov obklopujúcich trhlinu. Vypočítala sa energia stavu s dvoma roztrhnutými väzbami, ktorá sa označila ako u_{nova}

4. Porovnala sa energia sústavy s jednou roztrhnutou väzbou s energiou sústavy s dvoma roztrhnutými väzbami:

   (a) ak $u_{\text{nova}} < u_{\text{stara}}$, stav sa uskutoční, t.j. väzba sa pretrhne (princíp minima energie)

   (b) ak $u_{\text{nova}} > u_{\text{stara}}$, pravdepodobnosť prechodu zo starého do nového stavu, $Pr = e^{(u_{\text{stara}} - u_{\text{nova}})}$, je testovaná za použitia vygenerovaného náhodného čísla N z intervalu $(0, 1)$.

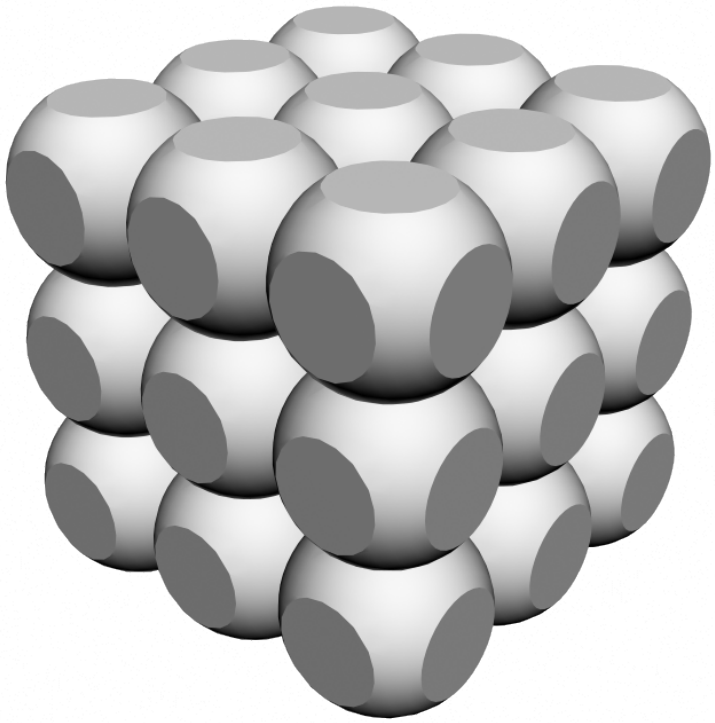
      - $N < Pr$ - väzba sa pretrhne, nový stav sa akceptuje, u_{nova} preberá hodnotu u_{stara},
      - $N > Pr$ - väzba sa nepretrhne, nový stav sa neakceptuje.

5. Návrat na krok 3.

Opakovane sa pokračuje od bodu 3 kým trhlina neprejde cez celú vzorku, kedy sa simulácia zastaví.

2.4 Jednoduchý trojrozmerný systém

Ako jednoduchý systém vhodný pre simuláciu lomu trojrozmerného telesa slúži súbor rovnakých zŕn usporiadaných do trojrozmerného periodického útvaru. V najjednoduchšom prípade sa zrná nachádzajú v uzloch jednoduchej kubickej mriežky. Každé zrno má šesť najbližších susedov. Do každého rozhrania medzi zrnom a jeho najbližším susedom je umiestnený bod modelovej množiny s príslušnou dvojhodnotovou premennou ξ_{ijk}. Hodnota premennej ξ opäť informuje, či sú dané susediace zrná spojené väzbou ($\xi = 0$), alebo sú od seba oddelené ($\xi = 1$).



Obr. 5: Trojrozmerný model materiálu

Energia systému je aj v tomto prípade vypočítaná v duchu Griffithovej teórie. Výraz pre energiu obsahuje dva členy. Funkcionálny tvar prvého člena, ktorý reprezentuje energiu voľných povrchov, ostáva v porovnaní s dvojrozmerným prípadom nezmenený. To znamená

$$u_{\text{povrchová}} = K_1 \times (\text{počet roztrhnutých väzieb medzi zrnami}) \tag{6}$$

Parameter K_1 absorboval opäť viaceré veličiny, ako hustota povrchovej energie, plocha typickej hranice medzi zrnami, teplota. Druhý člen, ktorý predstavuje energiu pružnej deformácie uvoľnenú trhlinou, potrebuje v trojrozmernom prípade miernu modifikáciu. V duchu

Griffithovej teórie je táto energia úmerná súčinu hustoty energie pružnej deformácie a objemu, z ktorého je pružná deformácia trhlinou odstránená. Podľa Saint-Venantovho princípu deformácia, ktorú v poli napätia spôsobuje trhlina, vymizne zhruba do vzdialenosti jednej charakteristickej dĺžky. Z dôvodu symetrie problému sa predpokladá, že trhlina je dvojrozmerný, približne rovnoosový útvar. V takom prípade je charakteristickou dĺžkou priemer trhliny. Objem materiálu, v ktorom je pole napätia ovplyvnené trhlinou, je potom úmerný súčinu plochy trhliny a priemeru trhliny. Na druhej strane, počet roztrhnutých neprekrývajúcich sa horizontálnych rozhraní medzi zrnami udáva približne plochu trhliny. Roztrhnutie väzby v tomto prípade totiž predstavuje vznik skutočných lokalizovaných dvojrozmerných voľných povrchov, keď sa plochy, ktorými boli zrná pôvodne navzájom viazané zmenia na voľné povrchy. Priemer trhliny je zhruba rovný druhej odmocnine plochy trhliny. Keď tieto skutočnosti dáme dohromady, dostaneme pre druhý člen v energii trojrozmerného systému tvar:

$$u_{\text{elastická}} = -K_2 \times (\text{počet roztrhnutých neprekrývajúcich sa horizontálnych väzieb})^{\frac{3}{2}} \quad (7)$$

Parameter K_2 v sebe taktiež koncentruje viacero veličín: priemerný rozmer zrna, Youngov modul pružnosti, veľkosť pôsobiaceho napätia, tvarové faktory, teplota, a pod.

Schematický náčrt krokov simulácie:

1. Zadajú sa hodnoty K_1 a K_2.

2. Jednej premennej ξ_{ijk} v strede vzorky sa priradí hodnota jedna, čo zodpovedá roztrhnutie jednej horizontálnej hranice. Ostatným premenným ξ_{ijk} sa priradí hodnota nula. Vypočíta sa energia takého stavu systému a označí sa u_{stara}

3. Náhodne sa vyberie jeden z najbližších susedov bodov nachádzajúcich sa v trhline. Hodnota premennej v tomto bode sa zmení z nula na jedna. Stav s touto navyše roztrhnutou väzbou sa skúma ako možný nový stav.

4. Vypočíta sa energia možného nového stavu a označí sa u_{nova}.

5. Ak $u_{\text{nova}} < u_{\text{stara}}$, možný nový stav sa akceptuje ako reálny stav systému (systém prejde zo starého stavu do nového). Hodnota u_{nova} sa berie ako nová hodnota veličiny u_{stara}. Cyklus znova pokračuje od kroku 3.

6. Ak $u_{\text{nova}} \geq u_{\text{stara}}$, vypočíta sa pravdepodobnosť prechodu do nového stavu $Pr = e^{(u_{\text{stara}} - u_{\text{nova}})}$. Táto pravdepodobnosť sa porovnáva s náhodným číslom N z intervalu $(0, 1)$.

7. Ak $N < Pr$, možný stav sa akceptuje ako skutočný nový stav (systém prejde zo starého do nového stavu). u_{nova} sa stáva u_{stara} a cyklus sa opakuje od kroku 3.

8. Ak $N \geq Pr$, možný nový stav sa neakceptuje, systém ostáva v starom stave so starou u_{stara} a cyklus sa opakuje od kroku 3.

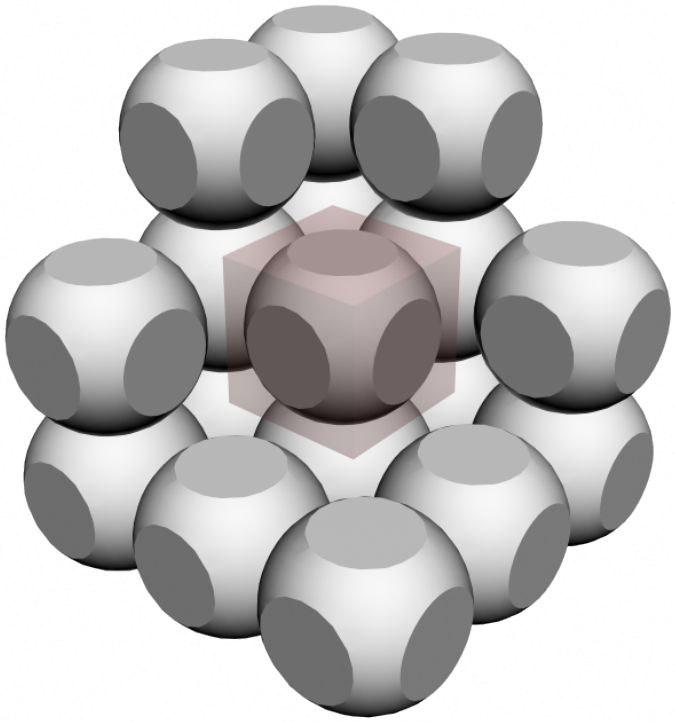
9. Simulácia sa zastaví ak trhlina dorazí k všetkým vertikálnym okrajom skúmanej vzorky.

3 Návrh paralelného algoritmu

Cieľom tejto práce je navrhnúť paralelný algoritmus, ktorý generuje výsledky porovnateľné s algoritmom uvedeným v časti 2.4. Uvedený algoritmus teda pokladáme za referenčný. V prílohe A na strane 47 je porovnanie výsledkov referenčného algoritmu s výsledkami algoritmu navrhovaného v tejto kapitole.

3.1 Pamäťový model, uloženie stavu väzieb

Množstvo pamäte dostupnej na grafickej karte je výrazný obmedzujúci prvok. Cieľom simulácie je preskúmať šírenie trhliny na čo najväčšej mriežke. Požadovaná mriežka má rozmery $1000 \times 1000 \times 500$, čo je spolu 500 000 000 zŕn. Každé zrno v skúmanej mriežke má väzbu so šiestimi susednými zrnami, čo je spolu približne 1 500 000 000 väzieb na celú mriežku.



Obr. 6: Konkrétne zrno v mriežke

Nech je každé zrno reprezentované ako ξ_{ijk}, kde usporiadaná trojica $[i, j, k]$ určuje absolútnu pozíciu zrna v mriežke (ďalej len súradnice). Nech zrno A je určené súradnicami $[x, y, z]$. Potom susedia zrna A majú súradnice $[x + 1, y, z]$, $[x - 1, y, z]$, $[x, y + 1, z]$,

$[x, y-1, z]$, $[x, y, z+1]$ a $[x, y, z-1]$. Nech každá väzba je určená dvojicou susedných zŕn $[A, B]$. Väzba $[B, A]$ je z fyzikálneho hľadiska totožná s väzbou $[A, B]$, preto je možné brať do úvahy len väzby $[A, X]$, kde zrno X patrí do množiny $[x+1, y, z]$, $[x, y+1, z]$ a $[x, y, z+1]$.

Pre každé zrno je teda vhodné pamätať si stav len troch väzieb. Zavedieme teda väzbu χ_{ijkl}, kde $[i, j, k]$ sú súradnice a l je identifikátor väzby, kde $0 \leq l \leq 2$. Nasledujúci vzťah bude určovať adresu väzby v lineárnej pamäti:

$$P_m = 3(i + ja + kab) + l \tag{8}$$

kde a a b sú rozmery podstavy celej mriežky v zrnách. Na zápis hodnoty každej väzby postačuje 1 bit, teda hodnota P_m označuje konkrétny bit v pamäti. Na uloženie všetkých väzieb potrebujeme 187 500 000 bytov pamäte, čo je menej ako 179MB.

Identifikátor väzby nech pre potreby nasledujúceho textu označuje väzby:

- $l = 0$ označuje väzbu medzi zrnami $[x, y, z]$ a $[x+1, y, z]$
- $l = 1$ označuje väzbu medzi zrnami $[x, y, z]$ a $[x, y+1, z]$
- $l = 2$ označuje väzbu medzi zrnami $[x, y, z]$ a $[x, y, z+1]$

Počas simulácie sa vykonávajú len dva typy operácii pracujúcich s väzbami, načítanie stavu väzby a porušenie väzby (nastavenie bitu na hodnotu 1). V prípade načítania ide o jednoduchú operáciu čítania z pamäte. Na zaznamenanie porušenia väzby sa dá využiť bitová operácia *OR*, ktorú grafická karta priamo podporuje a ide o jednu z atomických operácii práce s pamäťou.

3.2 Uloženie stavu stĺpcov

Na výpočet energie sa okrem celkového počtu roztrhnutých väzieb používa aj počet navzájom sa neprekrývajúcich roztrhnutých horizontálnych väzieb. Ak si zrná v mriežke predstavíme ako stĺpce, tak z každého stĺpca môže byť započítaná maximálne jedna roztrhnutá horizontálna väzba.

Potrebujeme si preto uložiť stav každého stĺpca, teda či sa v ňom už nachádza alebo nenachádza roztrhnutá horizontálna väzba. Pre väzbu χ_{ijkl}, kde $[i, j]$ sú súradnice stĺpca, v ktorom sa väzba nachádza, použijeme nasledujúci vzťah na výpočet adresy stavu stĺpca v lineárnej pamäti:

$$P_s = (i + ja) \tag{9}$$

kde a je jeden z rozmerov podstavy celej mriežky v zrnách. Na zápis hodnoty stavu každého stĺpca postačuje 1 bit, teda hodnota P_s označuje konkrétny bit v pamäti. Na uloženie všetkých stavov stĺpcov potrebujeme 125 000 bytov pamäte, čo je menej ako 128kB.

3.3 Výpočet susedných väzieb

Na rozdiel od dvojrozmerného modelu, kde každá väzba susedila len s troma ďalšími, v trojrozmernom modeli má každá väzba až dvanásť susedov. Nech je väzba χ_{ijkl} daná usporiadanou štvoricou $[i, j, k, l]$. Potom k nej susedné väzby sú určené tabuľkou:

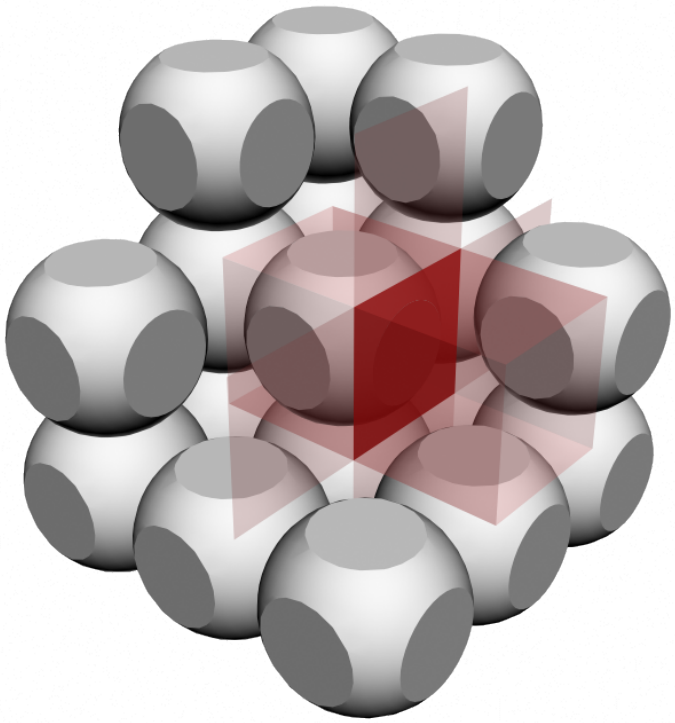
Pre $l = 0$	Pre $l = 1$	Pre $l = 2$
$[i, j, k, 1]$	$[i, j, k, 0]$	$[i, j, k, 0]$
$[i, j, k, 2]$	$[i, j, k, 2]$	$[i, j, k, 1]$
$[i, j-1, k, 1]$	$[i-1, j, k, 0]$	$[i-1, j, k, 0]$
$[i, j, k-1, 2]$	$[i, j, k-1, 2]$	$[i, j-1, k, 1]$
$[i, j, k-1, 0]$	$[i-1, j, k, 1]$	$[i-1, j, k, 2]$
$[i, j, k+1, 0]$	$[i+1, j, k, 1]$	$[i+1, j, k, 2]$
$[i, j-1, k, 0]$	$[i, j, k-1, 1]$	$[i, j-1, k, 2]$
$[i, j+1, k, 0]$	$[i, j, k+1, 1]$	$[i, j+1, k, 2]$
$[i+1, j, k, 1]$	$[i, j+1, k, 0]$	$[i, j, k+1, 0]$
$[i+1, j, k, 2]$	$[i, j+1, k, 2]$	$[i, j, k+1, 1]$
$[i+1, j-1, k, 1]$	$[i-1, j+1, k, 0]$	$[i-1, j, k+1, 0]$
$[i+1, j, k-1, 2]$	$[i, j+1, k-1, 2]$	$[i, j-1, k+1, 1]$

Všetky susedné väzby sú zobrazené na obrázku 7. Pôvodná väzba je zvýraznená.

3.4 Hľadanie susedných väzieb

V referenčnom algoritme sa v kroku 3 vyberá sused jednej z už roztrhnutých väzieb. Aby bol tento krok optimálny a nebolo potrebné prehľadávať celú mriežku, je potrebné pamätať si zoznam roztrhnutých väzieb, ktoré ešte nemajú roztrhnutých všetkých susedov a teda je možné sa z nich ďalej šíriť. Opäť je potrebné dbať na využitú pamäť. Väzby, ktoré sú roztrhnuté, ale majú ešte aspoň jedného neroztrhnutého suseda, budem ďalej v texte označovať ako *živé väzby*.

Za predpokladu, že súradnice $[i, j, k, l]$ každej väzby χ_{ijkl} sú z intervalu $0 \leq i, j, k \leq 1023$ a $0 \leq l \leq 3$, vieme pozíciu väzby zapísať do 32 bitov. Požadovaná mriežka s rozmermi $1000 \times 1000 \times 500$ tento predpoklad spĺňa.



Obr. 7: Susedné väzby

Počet živých väzieb, ktoré bude potrebné si pamätať, je žiaľ dopredu neznámy a grafická karta neumožňuje pracovať s dynamickými dátovými štruktúrami. Je preto potrebné odhadnúť veľkosť pamäte, ktorú je potrebné rezervovať. Experimentálne sa ukázala ako dostatočná veľkosť 50 000 000 väzieb a to aj pre mriežky s rozmermi presahujúcimi požiadavky a nízke hodnoty K_1. Pamäť potrebná na uloženie daného počtu väzieb je menej ako 191MB.

Uvádzanú dátovú štruktúru budem označovať ako *zoznam živých väzieb*. Hoci celá štruktúra bude mať dopredu určenú veľkosť, je potrebné pamätať si jej aktuálne obsadenie (počet uložených živých väzieb) vo forme globálneho počítadla. Na začiatku je počítadlo nastavené na hodnotu 1 a do zoznamu je vložená prvá, počiatočná živá väzba (zárodok trhliny). Aktuálnu pozíciu počítadla budem označovať ako *koniec zoznamu živých väzieb*.

3.5 Poradie výberu testovaných väzieb, usporiadanie

V referenčnom algoritme sa v kroku 3 vyberá sused náhodne. Mojou snahou je všetky kroky algoritmu paralelizovať. Náhodný výber suseda ale nie je vhodný z dvoch dôvodov:

- Náhodný dej znižuje celkovú efektivitu výpočtu
- Sekvenčný prístup do pamäte je na grafických kartách optimalizovaný [2]

Pokus: Nech je náhodný výber väzieb v referenčnom algoritme nahradený sekvenčným prechádzaním poľa popísaného v kapitole 3.4. Pri roztrhnutí väzby je tá zároveň pridaná na koniec poľa. Ak má väzba roztrhnutých všetkých susedov, je z poľa odstránená a na jej miesto je presunutá väzba z konca poľa. Ak sekvenčné prechádzanie narazí na koniec poľa, pokračuje opäť od začiatku.

Výsledok: Upravený algoritmus generuje výsledky porovnateľné s výsledkami referenčného algoritmu. Počet roztrhnutých väzieb po skončení simulácie s rovnakými testovanými parametrami bol po spriemerovaní výsledkov všetkých behov rovnaký.

Zdôvodnenie: Vzťah určujúci energiu novej konfigurácie (popísaný v kapitole 2) je závislý len na počte roztrhnutých väzieb a nie na ich pozícii v rámci mriežky. Preto nezáleží na poradí, v akom budú jednotlivé väzby vyberané. Počet roztrhnutých väzieb po skončení simulácie by mal byť teda vždy rovnaký (v rámci tolerancie), aj keď samotná trhlina bude mať vždy iný tvar (čo platí aj pri náhodnom výbere).

Dôsledok: Navrhovaný paralelný algoritmus vyberá väzby sekvenčne a cyklicky z poľa. Každé vlákno vyberie jednu zo zoznamu živých väzieb v poradí podľa svojho indexu. Všetky nasledujúce kroky algoritmu sa budú vykonávať v rámci každého vlákna.

3.6 Roztrhnutie väzieb, neskorá synchronizácia

V referenčnom algoritme prebieha porovnanie novej energie celého systému oproti energii pôvodnej (bez roztrhnutej väzby). Keďže skúšanie väzieb prebieha sekvenčne, počet roztrhnutých väzieb je vždy aktuálny a teda aj vypočítaná energia zodpovedá aktuálnemu stavu celého systému. V paralelnom algoritme prebieha výpočet novej energie súčasne (alebo skoro súčasne) vo všetkých vláknach a všetky používajú údaje, ktoré boli platné len v momente štartu. Jediným riešením je použiť kritickú oblasť, čo by však skoro celý algoritmus degradovalo na sekvenčný.

Pokus: Nech sa v referenčnom algoritme na výpočet novej energie využívajú hodnoty počtu roztrhnutých väzieb, ktoré už nie sú platné. Aktuálne hodnoty počtu roztrhnutých

väzieb sú vypočítané vždy až po určitom počte krokov. Počet krokov medzi aktualizáciami je nastavený na od 32 po 512.

Výsledok: Upravený algoritmus generuje výsledky porovnateľné s výsledkami referenčného algoritmu. Počet roztrhnutých väzieb po skončení simulácie s rovnakými testovanými parametrami bol po spriemerovaní výsledkov všetkých behov rovnaký. Na testovanie bola použitá mriežka s rozmermi $300 \times 300 \times 300$.

Zdôvodnenie: Jedna väzba znamená pri veľkom počte roztrhnutých väzieb iba malý prírastok alebo úbytok energie. V dôsledku zaokrúhľovania pri použití 32-bitovej aritmetiky, dostupnej na grafických kartách, vzniká pri vysokých počtoch roztrhnutých väzieb omnoho väčšia chyba, ako z dôvodu oneskorenej synchronizácie. Jedna väzba sa pri veľkom počte roztrhnutých väzieb v hodnote energie vôbec neprejaví. Je preto možné testovať niekoľko väzieb súčasne s použitím rovnakých začiatočných hodnôt bez odklonenia sa od výsledku vygenerovaného referenčným algoritmom.

Dôsledok: Navrhovaný algoritmus funguje sekvenčne do určitého počtu roztrhnutých väzieb, ktorý je rádovo vyšší ako počet súčasne vykonávaných vlákien. Po dosiahnutí tejto hranice je možné vykonávať simuláciu paralelne bez zmeny konečného výsledku oproti referenčnému algoritmu.

3.7 Udržiavanie zoznamu živých väzieb

Kroky popísané v predchádzajúcich kapitolách zabezpečovali trhanie väzieb a zaraďovanie živých väzieb do zoznamu. Po čase sa v zozname určite začnú nachádzať aj väzby, ktoré boli pôvodne živé, no už majú všetkých susedov roztrhnutých. Tieto väzby je potrebné priebežne zo zoznamu vyraďovať, aby zbytočne nespotrebovávali cykly procesora. Nie je nutné, aby sa to dialo po skončení každého paralelného behu, prípadne po každom cykle. Správne načasovanie ovplyvňuje viacero faktorov ako počet neživých väzieb, počet zatiaľ nevyužitých položiek zoznamu, počet položiek v zozname, a iné.

Samotné prečistenie zoznamu musí nutne prebehnúť priamo na grafickej karte z dôvodu časovej náročnosti kopírovania údajov medzi pamäťou systému a grafickej karty. V ideálnom prípade by mal byť algoritmus fungovať paralelne. Pri návrhu je nutné zabezpečiť, aby nemohla nastať situácia, kedy bude živá väzba ponechaná mimo zoznamu. Táto väzba by sa už do zoznamu viac nedostala a trhlina by sa z nej už ďalej nešírila. Na druhej strane, nie je problém, ak v zozname zostane aj isté percento už neživých väzieb.

Vyraďovanie väzieb zo zoznamu znamená skracovanie zoznamu, čo je vlastne spätný posun ukazovateľa konca zoznamu. Z dôvodu absencie kritickej oblasti je potrebné zabezpečiť, aby sa v žiadnom prípade v rámci jedného bloku nespracovávali záznamy, ktoré sa presunom ukazovateľa môžu dostať mimo zoznamu.

Navrhovaný algoritmus:

1. Nech N je počítadlo zoznamu živých väzieb, S=0
2. Nech $P = \lfloor (N - S)/2 \rfloor$
3. Paralelne spusti vlákna s indexmi od S po $S + P$
   (a) Otestuj väzbu s daným indexom
   (b) Ak už nie je živá, tak atomicky odčítaj 1 od N a na pozíciu indexu ulož väzbu, ktorá bola na pozícii N pred odčítaním
4. $S = S + P$, ak $S = N - 1$ tak skonči

Tento algoritmus vždy splní predchádzajúcu podmienku, keďže zapisuje určite len na pozície, z ktorých práve nemôže kopírovať väzby.

4 Implementácia

Implementovaný algoritmus je pomerne rozsiahly. Z tohto dôvodu neuvádzam plnú implementáciu, ale zameriam sa iba na podstatné časti, ktoré budú následne vysvetlené. Kompletný zdrojový kód je priložený na CD k tejto diplomovej práci.

Rozsah tu uvedeného kódu by mal byť dostatočný na implementáciu ekvivalentne fungujúceho algoritmu bez nutnosti používať akékoľvek iné materiály a zdrojové kódy.

4.1 Globálne pamäťové premenné a polia

V programe využívam nasledujúce globálne pamäťové premenné a polia. Dostupné sú zo všetkých funkcií.

```
#define MAX_QUEUE 10000000

struct S_STATE{
    int n_bonds;
    int n_horizontal_bonds;
    int n_queue;
    int n_traj;
    int end;
};

/* System variables */
S_STATE h_state;
int n_grid;
int n_horizontal_grid;

unsigned int *h_grid;
unsigned int *h_horizontal_grid;
unsigned int *h_traj;

/* GPU variables */
__device__ S_STATE *s_state;

__device__ int n_bonds, n_horizontal_bonds;
```

```
__device__ int n_queue;
__device__ int n_traj;
__device__ int n_state;

__device__ unsigned int *s_queue;
__device__ unsigned int *s_grid;
__device__ unsigned int *s_horizontal_grid;
__device__ unsigned int *s_traj;

__constant__ __device__ int s_tab[3][12][4];
```

- Premenné *n_grid* a *n_horizontal_grid* obsahujú veľkosť pamäte v bytoch vyhradenej pre uloženie celej štruktúry mriežky a stavu stĺpcov. Samotná štruktúra mriežky je uložená v poliach *h_grid* a *s_grid*. Stav stĺpcov je uložený v *h_horizontal_grid* a *s_horizontal_grid*.

- V poli *n_traj* a *h_traj* je uložená aktuálna trajektória (postupnosť) trhania väzieb. Tá má rovnakú štruktúru ako zoznam živých väzieb a ukladajú sa do nej väzby v takom poradi, ako sa trhajú. Počítadlo *n_traj* obsahuje počet väzieb v aktuálnej trajektórii. Po skončení jednej etapy sa trajektória uloží na disk a počítadlo sa vynuluje.

- Premenné *n_bonds* a *n_horizontal_bonds* obsahujú aktuálny počet roztrhnutých väzieb a počet roztrhnutých neprekrývajúcich sa horizontálnych väzieb (stĺpcov).

- Premenná *n_queue* obsahuje počet prvkov v zozname živých väzieb a pole *s_queue* samotný zoznam živých väzieb.

- Pole *s_tab* obsahuje zoznam susedných väzieb. Tento zoznam je predpočítaný.

- Premenná *n_state* obsahuje na štyroch najmenej signifikantných bitoch informáciu, či sa trhlina dostala na okraj mriežky. Každý bit vyjadruje iný okraj. Ak sú všetky štyri nastavené na hodnotu 1, trhlina sa dostala na všetky štyri okraje a je potrebné ukončiť simuláciu.

4.2 Jadro výpočtu

Jadro výpočtu tvorí zbierka funkcií bežiacich priamo na grafickej karte alebo na systémovom procesore.

```
__device__ inline int simulate_check_border(dim3 d, uint4 now)
{
    if(now.x<1||now.y<1||now.z<1||now.x>=d.x||now.y>=d.y||now.z
       >=d.z){
        return 1;
    }
    return 0;
}
```

Funkcia: *int simulate_check_border(dim3 d, uint4 now)* kontroluje, či sa určená väzba nachádza vo vnútri mriežky a teda či pre ňu platí podmienka, že má určite všetkých 12 susedov v rámci mriežky.

Popis parametrov:

- *dim3 d* - Obsahuje dimenziu simulovanej mriežky.
- *uint4 now* - Pozícia väzby, ktorú je potrebné skontrolovať.

```
__device__ uint4 simulate_queue_get(int poz, unsigned int *
   s_queue)
{
    if(poz>=MAX_QUEUE) poz=0;
    uint4 v=make_uint4((s_queue[poz]>>22)&1023,
                       (s_queue[poz]>>12)&1023,
                       (s_queue[poz]>>2)&1023,
                       (s_queue[poz])&3);
    return v;
}

__device__ void simulate_queue_set(uint4 v, unsigned int *s_queue
   )
```

```
{
    int poz=atomicAdd(&n_queue,1);
    if(poz>=MAX_QUEUE) return;
    s_queue[poz]=(v.x<<22)
                 +(v.y<<12)
                 +(v.z<<2)
                 +v.w;
}
```

Funkcia: *uint4 simulate_queue_get(int poz, unsigned int *s_queue)* vráti v návratovej hodnote väzbu, ktorá sa nachádzala na pozícii určenej parametrom *poz* v zozname živých väzieb. Ak je pozícia mimo zoznamu, tak funkcia vráti pozíciu prvej mriežky v zozname.

Funkcia: *void simulate_queue_set(uint4 v, unsigned int *s_queue)* vloží na koniec zoznamu živých väzieb väzbu určenú parametrom *v*. Keďže funkcia je volaná z viacerých vlákien súčasne, je potrebné využiť atomickú funkciu na zvýšenie premennej vyjadrujúcej počet živých väzieb v zozname.

```
__device__ void simulate_traj_set(uint4 v,unsigned int *s_traj)
{
    int poz=atomicAdd(&n_traj,1);
    s_traj[poz]=(v.x<<22)
                +(v.y<<12)
                +(v.z<<2)
                +v.w;
}
```

Funkcia: *void simulate_traj_set(uint4 v, unsigned int *s_traj)* vloží na koniec poľa s trajektóriou väzbu určenú parametrom *v*. Keďže funkcia je volaná z viacerých vlákien súčasne, je potrebné využiť atomickú funkciu na zvýšenie premennej vyjadrujúcej počet väzieb v trajektórii.

```
__device__ uint4 simulate_translate(uint4 d,char o)
{
    return make_uint4(d.x+s_tab[d.w][o][0],
                      d.y+s_tab[d.w][o][1],
```

```
                                d.z+s_tab[d.w][o][2],
                                s_tab[d.w][o][3]);
}
```

Funkcia: *uint4 simulate_translate(uint4 d, char o)* vráti v návratovej hodnote väzbu, ktorá predstavuje *o*-tého suseda väzby zadanej parametrom *d*. Funkcia nekontroluje, či vypočítaná väzba neleží mimo mriežky.

```
__device__ uint2 simulate_grid_translate(dim3 d,uint4 p)
{
    unsigned int s=(p.x+(p.y*d.x)
                    +(p.z*d.x*d.y))*3
                    +p.w;
    uint2 v;
    v.x=s>>5;
    v.y=s&31;
    return v;
}
```

Funkcia: *uint2 simulate_grid_translate(dim3 d, uint4 p)* vráti v návratovej hodnote pozíciu v poli *s_grid* (ako parameter *x*) a konkrétny bit (ako parameter *y*) popisujúci väzbu určenú parametrom *p* v pamäťovej mriežke s dimenziou *d*. Funkcia nekontroluje, či vypočítaná pozícia nie je mimo poľa.

```
__device__ uint2 simulate_horizontal_grid_translate(dim3 d,uint4
    p)
{
    unsigned int s=p.x
                    +p.y*d.x;
    uint2 v;
    v.x=s>>5;
    v.y=s&31;
    return v;
}
```

Funkcia: *uint2 simulate_horizontal_grid_translate(dim3 d, uint4 p)* vráti v návratovej hodnote pozíciu v poli *s_horizontal_grid* (ako parameter x) a konkrétny bit (ako parameter y) popisujúci stĺpec patriaci väzbe určenej parametrom p v pamäťovej mriežke s dimenziou d. Funkcia nekontroluje, či vypočítaná pozícia nie je mimo poľa.

```
__device__ void simulate_set_grid_bit(dim3 d,uint4 p,bool
    save_traj,unsigned int *s_grid,unsigned int *s_traj)
{
    uint2 v=simulate_grid_translate(d,p);
    unsigned int bit=1<<(v.y);
    if(!(atomicOr(s_grid+v.x,bit)&bit)){
        atomicAdd(&n_bonds,1);
        if(p.x==0) n_state|=1;
        if(p.y==0) n_state|=2;
        if(p.x==d.x-1) n_state|=4;
        if(p.y==d.y-1) n_state|=8;
        if(save_traj){
            simulate_traj_set(p,s_traj);
        }
    }
}

__device__ int simulate_get_grid_bit(dim3 d,uint4 p,unsigned int
    *s_grid)
{
    uint2 v=simulate_grid_translate(d,p);
    return (s_grid[v.x]>>v.y)&1;
}
```

Funkcia: *void simulate_set_grid_bit(dim3 d, uint4 p, bool save_traj, unsigned int *s_grid, unsigned int *s_traj)* nastaví určenú väzbu ako roztrhnutú.

Popis parametrov:

- *dim3 d* - Obsahuje dimenziu simulovanej mriežky.
- *uint4 p* - Pozícia väzby, ktorú je potrebné roztrhnúť.

- *bool save_traj* - Premenná určujúca, či sa má ukladať trajektória šírenia trhliny.
- *s_grid, s_traj* - Polia, ktoré boli popísané v kapitole 4.1.

Funkcia si najprv preloží väzbu na pozíciu v poli *s_grid* a vypočíta premennú s jediným bitom nastaveným na 1. Volaním atomického OR-u sa pokúsi nastaviť daný bit a skontroluje, či už predtým nebol nastavený na 1. Ak nebol, tak zvýši počítadlo roztrhnutých väzieb a skontroluje, či sa väzba nenachádza na okraji mriežky. Ak je aktívne ukladanie trajektórie, väzba sa uloží.

Funkcia: *int simulate_get_grid_bit(dim3 d, uint4 p, unsigned int *s_grid)* vráti v návratovej hodnote stav bitu zodpovedajúceho určenej väzbe.

Popis parametrov:

- *dim3 d* - Obsahuje dimenziu simulovanej mriežky.
- *uint4 p* - Pozícia väzby, ktorú je potrebné skontrolovať.
- *s_grid* - Pole, ktoré bolo popísané v kapitole 4.1.

```
__device__ void simulate_set_horizontal_grid_bit(dim3 d,uint4 p,
    unsigned int *s_horizontal_grid)
{
    uint2 v=simulate_horizontal_grid_translate(d,p);
    unsigned int bit=1<<(v.y);
    if(!(atomicOr(s_horizontal_grid+v.x,bit)&bit)){
        atomicAdd(&n_horizontal_bonds,1);
    }
}

__device__ int simulate_get_horizontal_grid_bit(dim3 d,uint4 p,
    unsigned int *s_horizontal_grid)
{
    uint2 v=simulate_horizontal_grid_translate(d,p);
    if(s_horizontal_grid[v.x]&(1<<(v.y))) return 1;
    return 0;
}
```

Funkcia: *void simulate_set_horizontal_grid_bit(dim3 d, uint4 p, unsigned int *s_horizontal_grid)* nastaví stĺpec určený väzbou ako využitý.

Popis parametrov:

- *dim3 d* - Obsahuje dimenziu simulovanej mriežky.
- *uint4 p* - Pozícia väzby, ktorej stĺpec je potrebné označiť.
- *s_horizontal_grid* - Pole, ktoré bolo popísané v kapitole 4.1.

Funkcia si najprv preloží väzbu na pozíciu v poli *s_horizontal_grid* a vypočíta premennú s jediným bitom nastaveným na 1. Volaním atomického OR-u sa pokúsi nastaviť daný bit a skontroluje, či už predtým nebol nastavený na 1. Ak nebol, tak zvýši počítadlo použitých stĺpcov.

Funkcia: *int simulate_get_horizontal_grid_bit(dim3 d, uint4 p, unsigned int *s_horizontal_grid)* vráti v návratovej hodnote stav bitu zodpovedajúceho stĺpcu určenej väzby.

Popis parametrov:

- *dim3 d* - Obsahuje dimenziu simulovanej mriežky.
- *uint4 p* - Pozícia väzby, ktorej stĺpec je potrebné skontrolovať.
- *s_horizontal_grid* - Pole, ktoré bolo popísané v kapitole 4.1.

```
__global__ void simulate_kernel(dim3 dimension,dim3 gdim,int2
    sdim,bool save_traj,float k1,float k2,unsigned int *s_queue,
    unsigned int *s_grid,unsigned int *s_horizontal_grid,unsigned
     int *s_traj,unsigned int *gpurandval)
{
    int poz=threadIdx.x+((blockIdx.y*sdim.y)+blockIdx.x)*sdim.x;
    int bit=1;
    int nb,nhb;
    int s;
    uint4 now,next;
    float now_energy,next_energy;
```

```
if(poz<n_queue){
    now=simulate_queue_get(poz,s_queue);
    next=simulate_translate(now,gpurand_kernel(gpurandval)
        %12);
    bit=simulate_get_grid_bit(gdim,next,s_grid);
    nb=n_bonds;
    nhb=n_horizontal_bonds;
}
__syncthreads();
if(!bit){
    s=0;
    now_energy=(float)(nb)*k1-k2*powf((float)nhb,1.5f);
    if(next.w==2&&simulate_get_horizontal_grid_bit(gdim,next
        ,s_horizontal_grid)==0){
        s=1;
    }
    next_energy=(float)(nb+1)*k1-k2*powf((float)(nhb+s),1.5f
        );

    char crack=0;
    if(next_energy<now_energy){
        crack=1;
    }
    else{
        if( round(((float)MAX_GPURAND/expf(next_energy-
            now_energy))) > gpurand_kernel(gpurandval) ){
            crack=1;
        }
    }

    if(crack){
        if(s){
            simulate_set_horizontal_grid_bit(gdim,next,
                s_horizontal_grid);
```

```
            }

            if(!simulate_check_border(dimension,next)){
                simulate_queue_set(next,s_queue);
            }

            simulate_set_grid_bit(gdim,next,save_traj,s_grid,
                s_traj);
        }
    }
}
```

Funkcia: *void simulate_kernel(dim3 dimension, dim3 gdim, int2 sdim, bool save_traj, float k1, float k2, unsigned int *s_queue, unsigned int *s_grid, unsigned int *s_horizontal_grid, unsigned int *s_traj, unsigned int *gpurandval)* vykonáva hlavný výpočet energie a rozhodovanie o roztrhnutí väzby.

Popis parametrov:

- *dim3 dimension* - Obsahuje dimenziu mriežky, ktorá sa bude simulovať.
- *dim3 gdim* - Obsahuje dimenziu pamäťovej mriežky. Tá je väčšia ako simulovaná mriežka kvôli možnosti uloženia stavu väzieb aj na okrajoch mriežky.
- *dim3 sdim* - Obsahuje počet vlákien v rámci jedného bloku a počet blokov v rámci jedného gridu.
- *bool save_traj* - Premenná určujúca, či sa má ukladať trajektória šírenia trhliny.
- *float k1, float k2* - Parametre simulácie
- *s_queue, s_grid, s_horizontal_grid, s_traj* - Polia, ktoré boli popísané v kapitole 4.1.
- *gpurandval* - Pole popísané v kapitole 4.6.

Funkcia najprv vypočíta pozíciu v zozname živých väzieb. Ak je na vypočítanej pozícii väzba, tak je načítaná a je náhodne vybraná jedna z jej susedných väzieb. Ak je susedná väzba ešte neroztrhnutá, vypočíta sa aktuálna energia mriežky a energia mriežky s danou väzbou roztrhnutou. Ak je nová energia nižšia ako pôvodná, väzba sa môže roztrhnúť. Inak

sa vypočíta pravdepodobnosť prechodu do nového stavu a vygeneruje sa náhodné číslo. Pri trhaní väzby sa najprv nastaví stĺpec väzby, ak to je potrebné. Ďalej sa skontroluje, či väzba leží vo vnútri mriežky a ak áno, tak sa pridá do zoznamu živých väzieb. Následne sa označí za roztrhnutú.

Tento proces sa podľa možností vykonáva paralelne pre všetky väzby zo zoznamu živých väzieb.

4.3 Udržiavanie zoznamu živých väzieb

Zoznam živých väzieb obsahuje po istom čase aj väzby, ktorých všetci susedia sú už roztrhnutý. Je preto potrebné kvôli zvýšeniu výkonu tento zoznam od týchto väzieb prečisťovať.

```
void simulate_purge(dim3 gdim,int num_t)
{
    int p=0,l;
    while(1){
        simulate_get_state();
        l=(h_state.n_queue-p)/2;
        if(l<(num_t<<2)) break;
        if(l>num_t) l=num_t;
        if(l<1) break;
        purge_kernel<<<1,l>>>(gdim,p,s_queue,s_grid);
        main_check();
        p+=l;
    }
}
```

Funkcia: *void simulate_purge(dim3 gdim, int num_t)* je vykonávaná v systémovej časti programu.

Popis parametrov:

- *dim3 gdim* - Obsahuje dimenziu pamäťovej mriežky.
- *int num_t* - Počet vlákien, ktoré môžu byť súčasne spustené v rámci jedného bloku.

Funkcia rozdeľuje doteraz nespracovanú časť zoznamu živých väzieb na polovice a na prvej spúšťa funkciu bežiacu na grafickej karte. Ak je veľká časť zoznamu spracovaná (zostáva

časť menšia ako $4num\_t$), funkcia sa ukončí. Dôvodom je náročnosť samotného overovania všetkých susedov väzby a preto je výhodnejšie radšej ponechať istú časť nespracovanú. Hranica je získaná experimentálne a pre rôzne grafické karty môže byť rôzna.

```
__global__ void purge_kernel(dim3 gdim,int start,unsigned int *
    s_queue,unsigned int *s_grid)
{
    int poz=threadIdx.x+start;
    uint4 now,next;

    now=simulate_queue_get(poz,s_queue);
    for(int i=0;i<12;i++){
        next=simulate_translate(now,i);
        if(simulate_get_grid_bit(gdim,next,s_grid)==0) return;
    }

    int p=atomicSub(&n_queue,1);
    s_queue[poz]=s_queue[p-1];
}
```

Funkcia: *void purge_kernel(dim3 gdim, int start, unsigned int *s_queue, unsigned int *s_grid)* skontroluje, či je väzba ešte stále živá.

Popis parametrov:

- *dim3 gdim* - Obsahuje dimenziu pamäťovej mriežky.
- *int start* - Určuje, kde začína doteraz nespracovaná časť zoznamu živých väzieb.
- *s_queue, s_grid* - Polia, ktoré boli popísané v kapitole 4.1.

Funkcia vypočíta pozíciu v zozname živých väzieb na základe parametra a identifikátora vlákna. Následne overuje všetkých 12 susedov skúmanej väzby. Ak sú všetky susedné väzby roztrhnuté, atomicky sa zníži ukazovateľ konca zoznamu a posledná väzba sa presunie na pozíciu skúmanej.

4.4 Pomocné funkcie

Premenné, ktoré sú uložené na grafickej karte, nie sú prístupné pre časti programu bežiace na systémovom procesore. Preto je potrebné použiť niekoľko obslužných funkcií.

```
__global__ void simulate_init_kernel(dim3 dimension,dim3 gdim,
    bool save_traj,uint4 now,unsigned int *s_queue,unsigned int *
    s_grid,unsigned int *s_horizontal_grid,unsigned int *s_traj)
{
     n_queue=0;
     n_bonds=0;
     n_horizontal_bonds=0;
     n_traj=0;
     n_state=0;

     if(now.w==2){
          simulate_set_horizontal_grid_bit(gdim,now,
              s_horizontal_grid);
     }
     simulate_set_grid_bit(gdim,now,save_traj,s_grid,s_traj);
     simulate_queue_set(now,s_queue);
}
```

Funkcia: *void simulate_init_kernel(dim3 dimension, dim3 gdim, bool save_traj, uint4 now, unsigned int *s_queue, unsigned int *s_grid, unsigned int *s_horizontal_grid, unsigned int *s_traj)* slúži na nastavenie začiatočných hodnôt na grafickej karte. Pred začiatkom simulácie je taktiež potrebné roztrhnúť prvú väzbu a vložiť ju do zoznamu živých väzieb.

Popis parametrov:

- *dim3 dimension* - Obsahuje dimenziu mriežky, ktorá sa bude simulovať.
- *dim3 gdim* - Obsahuje dimenziu pamäťovej mriežky. Tá je väčšia ako simulovaná mriežka kvôli možnosti uloženia stavu väzieb aj na okrajoch mriežky.
- *bool save_traj* - Premenná určujúca, či sa bude ukladať trajektória šírenia trhliny, alebo len koncový stav.

- *uint4 now* - Pozícia väzby, ktorú je potrebné na začiatok roztrhnúť.
- *s_queue, s_grid, s_horizontal_grid, s_traj* - Polia, ktoré boli popísané v kapitole 4.1.

Funkcia postupne vynuluje premenné, následne roztrhne určenú väzbu a pridá ju do zoznamu živých väzieb. Ak je väzba horizontálna, zapíše ju aj do stavu stĺpcov.

```
__global__ void simulate_get_state_kernel(S_STATE *x)
{
    x[0].n_bonds=n_bonds;
    x[0].n_horizontal_bonds=n_horizontal_bonds;
    x[0].n_queue=n_queue;
    x[0].n_traj=n_traj;
    x[0].end=n_state;
    n_traj=0;
}
```

Funkcia: *void simulate_get_state_kernel(S_STATE *x)* je spúšťaná na grafickej karte a do parametra x uloží aktuálny stav niektorých dôležitých premenných. Zároveň vynuluje ukazovateľ na prvý voľný záznam v trajektórii.

4.5 Ovládanie behu simulácie

Spúšťanie funkcií na grafickej karte je vykonávané v nekonečnom cykle.

```
n=(h_state.n_queue/(num_t+1));

simulate_kernel<<< make_uint3((n%num_g.x)+1,(n/num_g.x)+1,1),
    make_uint3(num_t,1,1) >>> (dimension,gdim,sdim,save_traj,k1,
    k2,s_queue,s_grid,s_horizontal_grid,s_traj,gpurandval);
main_check();

simulate_get_state();

if(save_traj){
    simulate_save_traj(tfile);
}
```

```
if((h_state.end&15)==15) break;

if((count%purge_period)==0){
    int delta=h_state.n_queue;
    simulate_purge(gdim,num_t);
    delta-=h_state.n_queue;
    if(delta<(num_t<<4)) purge_period+=2;
    if(delta>(num_t<<10)) purge_period-=2;
    if(purge_period<2) purge_period=2;
    if(purge_period>16) purge_period=16;
}

count++;
```

Pred spustením funkcie *simulate_kernel* sa vypočíta potrebná veľkosť gridu, aby sa všetky väzby v zozname živých väzieb skontrolovali práve raz. Po skončení je z grafickej karty prevzatý aktuálny stav dôležitých premenných a v prípade potreby sa uloží trajektória. Následne sa kontroluje, či už simulovaná trhlina nedosiahla všetky štyri okraje mriežky. Následne sa zisťuje, či je potrebné vykonať údržbu zoznamu živých väzieb. Periodicita vykonávania údržby je určená podľa počtu odstránených väzieb v predchádzajúcom kroku. Hranice pre zvyšovanie a znižovanie dĺžky periódy boli stanovené experimentálne a pre rôzne grafické karty môžu byť vhodné rôzne hranice.

4.6 Generovanie náhodných čísel

Vývojové prostredie CUDA neposkytuje žiaden predpripravený generátor náhodných čísel a z funkcií vykonávaných na grafickej karte nie je možné volať bežné funkcie jazyka C++.

Existuje viacero algoritmov, ktoré dokážu generovať náhodné čísla s rôznymi vlastnosťami priamo na grafických kartách. Implementoval som algoritmus definovaný v *POSIX.1-2001* z dôvodu jeho rýchlosti a jednoduchosti.

```
#define MAX_GPURAND 32767

__device__ unsigned int *gpurandval;

void init_gpurand(int num_threads)
```

```
{
    int s=sizeof(unsigned int)*num_threads;
    CUDA_SAFE_CALL(cudaMalloc((void **)&gpurandval,s));
    unsigned int *t;
    t=(unsigned int *)malloc(s);
    for(int i=0;i<num_threads;i++){
        t[i]=rand();
    }
    CUDA_SAFE_CALL(cudaMemcpy(gpurandval, t, s,
        cudaMemcpyHostToDevice));
    free(t);
}
```

Algoritmus generuje celé nezáporné čísla z rozsahu $< 0; 32767 >$. Funkcia *void init_gpurand(int num_threads)* vygeneruje pre každé vlákno vlastnú začiatočnú hodnotu a všetky uloží do globálneho poľa *gpurandval*.

```
__device__ inline int gpurand_kernel(unsigned int *gpurandval)
{
    unsigned int x=gpurandval[threadIdx.x];
    x=x*1103515245+12345;
    x%=2147483648;
    gpurandval[threadIdx.x]=x;
    return((x>>16)&MAX_GPURAND);
}
```

Funkcia: *int gpurand_kernel(unsigned int *gpurandval)* sa spúšťa na grafickej karte a vracia náhodné hodnoty. Každé vlákno využíva iný index z globálneho poľa a teda sa vlákna navzájom neovplyvňujú.

4.7 Poznámky k implementácii

Zdrojový kód, uvedený v predchádzajúcich kapitolách, neobsahuje všetky funkcie, na ktoré sa odvoláva. Ide prevažne o pomocné funkcie spúšťané na systémovom procesore a zabezpečujúce vstupno-výstupné operácie a obsluhu pamäte. Zároveň sa uvedené funkcie môžu mierne odlišovať od funkcií uverejnených v zdrojových kódoch na priloženom CD.

Na úspešnú kompiláciu je potrebné prostredie CUDA a CUDA SDK[1]. Grafická karta musí podporovať aspoň špecifikáciu verzie 1.1. Zoznam podporovaných zariadení nájdete v dokumente [2] príloha A. Zdrojový kód je možné kompilovať v operačných systémoch GNU/Linux a Microsoft Windows bez nutnosti zásahu. Podmienky, ktoré je potrebné splniť, sú popísane v dokumentácii ku kompilátoru [1] a závislé od používaného operačného systému.

Na priloženom CD sa v adresári */program/gpu* nachádza program implementujúci algoritmus popísaný v tejto kapitole v skompilovanej verzii pre systémy Windows vrátane všetkých potrebných knižníc. Zoznam možných parametrov získate po spustení programu bez akýchkoľvek parametrov alebo s parametrom *–help*.

Zdrojový kód je dostupný pod licenciou GNU GPLv3[2] alebo akoukoľvek novšou verziou podľa Vášho uváženia.

[1]Prostredie je dostupné na adrese *http://www.nvidia.com/object/cuda_get.html*

[2]Celé znenie licencie je dostupné na adrese *http://www.gnu.org/licenses/gpl-3.0.html*

5 Záver

Výpočet na grafickej karte prináša úsporu prostriedkov v podobe času v porovnaní s pôvodným algoritmom. Dosť obmedzujúcim faktorom sa ukazuje malé množstvo dostupnej operačnej pamäte. V prípade prekonania tejto prekážky by mal byť samotný algoritmus škálovatelný aj na väčšie mriežky bez nutnosti podstatnejších zmien.

Výsledkom algoritmu pre potreby aplikovaného výskumu materiálov je počet roztrhnutých väzieb a tvar (trajektória) trhliny po skončení simulácie. Keďže ide o algoritmus s náhodným výberom, použiteľné výsledky sa získavajú spriemerovaním výsledkov mnohých behov algoritmu s rovnakými vstupnými parametrami. Spracovanie a interpretácia samotných výsledkov je už mimo záberu tejto diplomovej práce a spadá do oblasti materiálového výskumu.

V prílohe A sú zobrazené tvary trhlín simulovaných na bežnom procesore (referenčný algoritmus) a na grafickej karte s použitím rovnakých vstupných parametrov. Kým podobné tvary trhlín a počty roztrhnutých väzieb možno považovať za istý empirický dôkaz ekvivalencie oboch algoritmov, formálne sa mi ekvivalenciu dokázať nepodarilo.

Popísaný algoritmus na simuláciu šírenia trhliny je možné bez väčších problémov prispôsobovať nie len novým skúmaným mriežkam a materiálom, ale aj novým možnostiam budúcich grafických kariet. Budúca práca bude smerovať k implementácii 128 bitovej aritmetiky a úprave algoritmu rozhodujúceho o roztrhnutí väzby, aby sa viac približoval fyzikálnemu modelu.

Ponúka sa aj možnosť použiť na výpočet viac grafických kariet alebo dokonca viac počítačov, keďže každé vlákno operuje počas svojho behu na pomerne malom úseku pamäte. Mriežku je teda možné rozdeliť na oblasti, ktoré budú uložené na rôznych kartách a po každom kroku bude stačiť synchronizovať iba malú oblasť, v ktorej bude spoločný prienik.

Literatúra

[1] *The CUDA Compiler Driver NVCC*. NVIDIA Corporation, 2007.

[2] *NVIDIA CUDA Programming Guide 1.1*. NVIDIA Corporation, 2007.

[3] Hosford, W.F. *Mechanical Behavior of Materials*. Cambridge University Press, New York, 2005.

[4] Kupka, S. Molecular dynamics on graphics accelerators. In *Central European Seminar on Computer Graphics*, 2006.

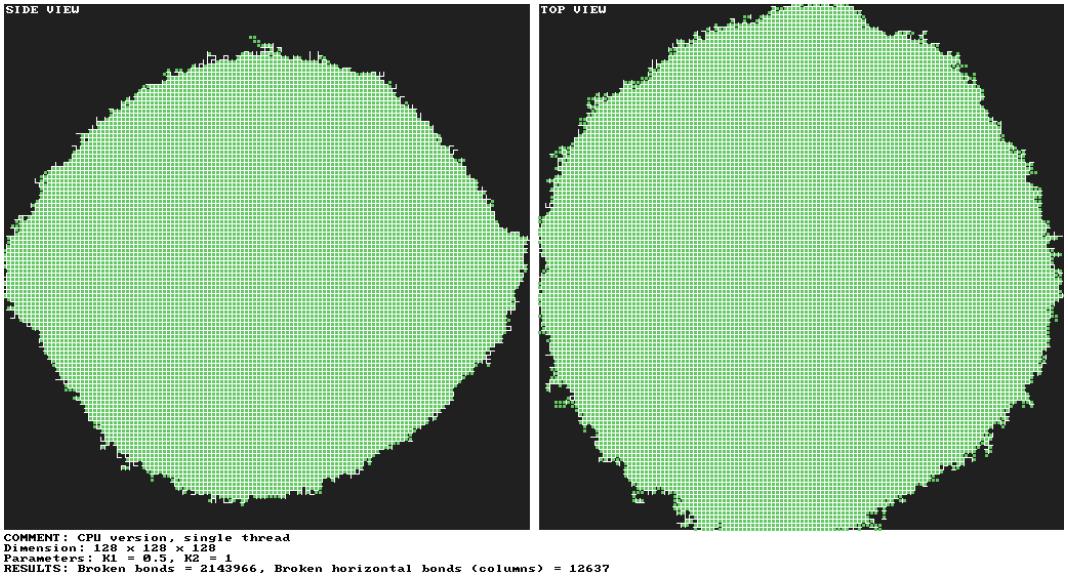
[5] Kupkova, M., Kristakova, Z. Fractal dimension of fracture patterns – a computer simulation. In Kruhl, J.H., editor, *Fractals and Dynamic Systems in Geoscience.*, 87–94, Berlin, 1994. Springer Verlag.

[6] Meakin, P. Simple models for material failure and deformation. In Mier, J.G.M., Rots, J.G., Bakker, A., editors, *Fracture Processes in Concrete, Rock and Ceramics.*, volume 1, 213–230, Noordwijk, 19.–21.6.1991, 1991. E & FN Spon.

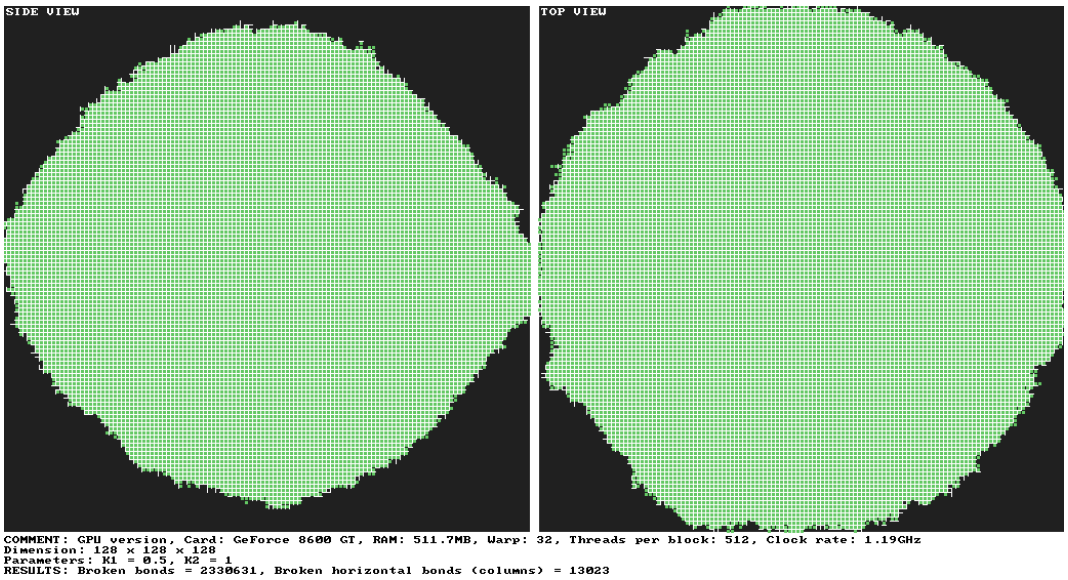
[7] Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G., Schulten, K. Accelerating molecular modeling applications with graphics processors. *J Comput Chem*, September 2007.

A Obrazová príloha

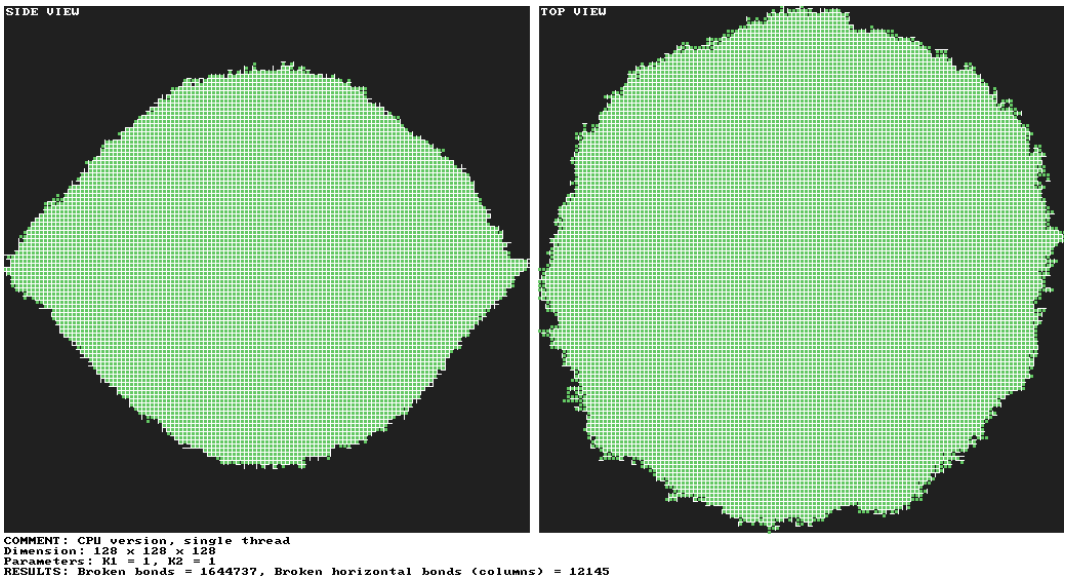
V tejto prílohe nájdete zobrazenie tvaru trhliny simulovanej na CPU a na GPU s rovnakými parametrami. Každý obrázok obsahuje dva podobrázky. Naľavo sa nachádza pohľad zboku (priemet do roviny určenej osami XZ) a napravo sa nachádza pohľad zhora (priemet do roviny určenej osami XY).



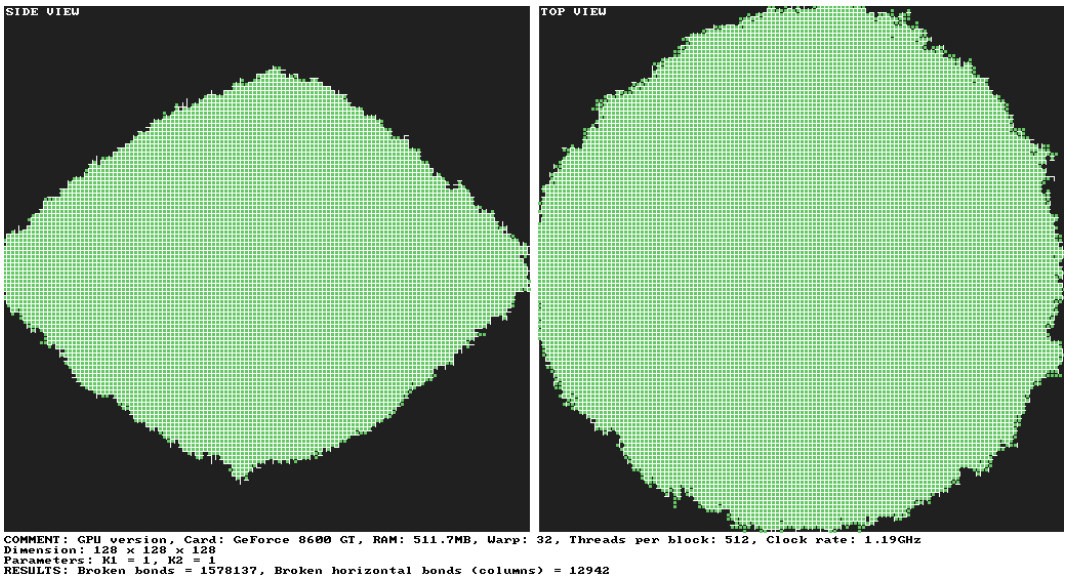
Obr. 8: Výsledný tvar trhliny na CPU s parametrami $K_1 = 0.5$ a $K_2 = 1$



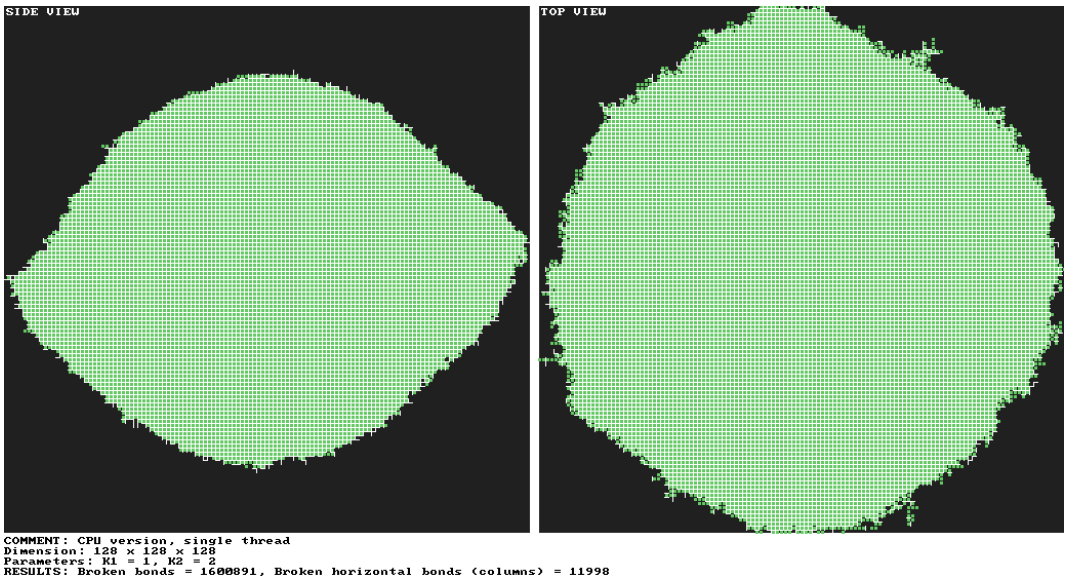
Obr. 9: Výsledný tvar trhliny na GPU s parametrami $K_1 = 0.5$ a $K_2 = 1$



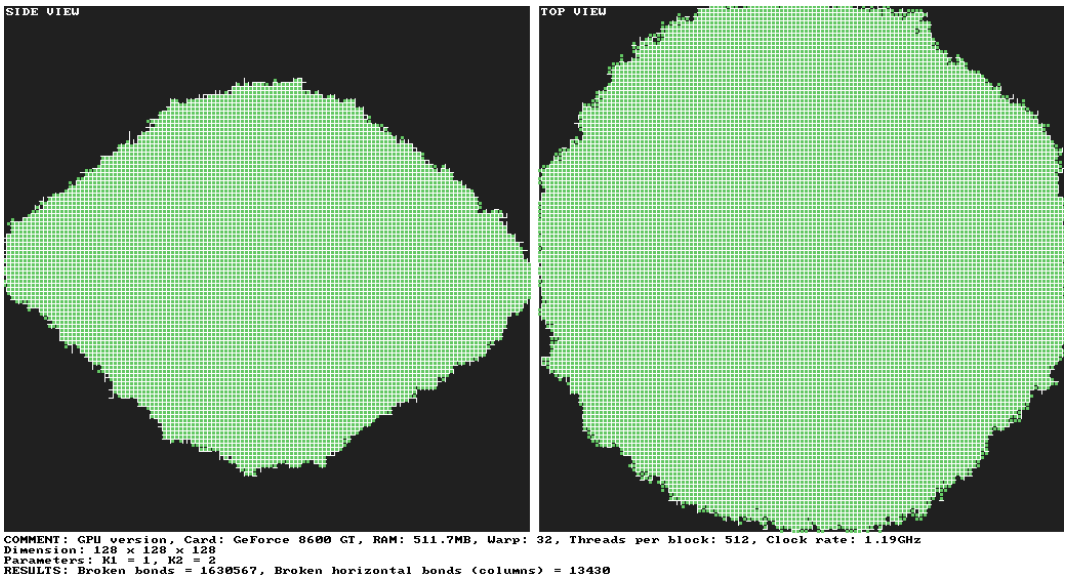
Obr. 10: Výsledný tvar trhliny na CPU s parametrami $K_1 = 1$ a $K_2 = 1$



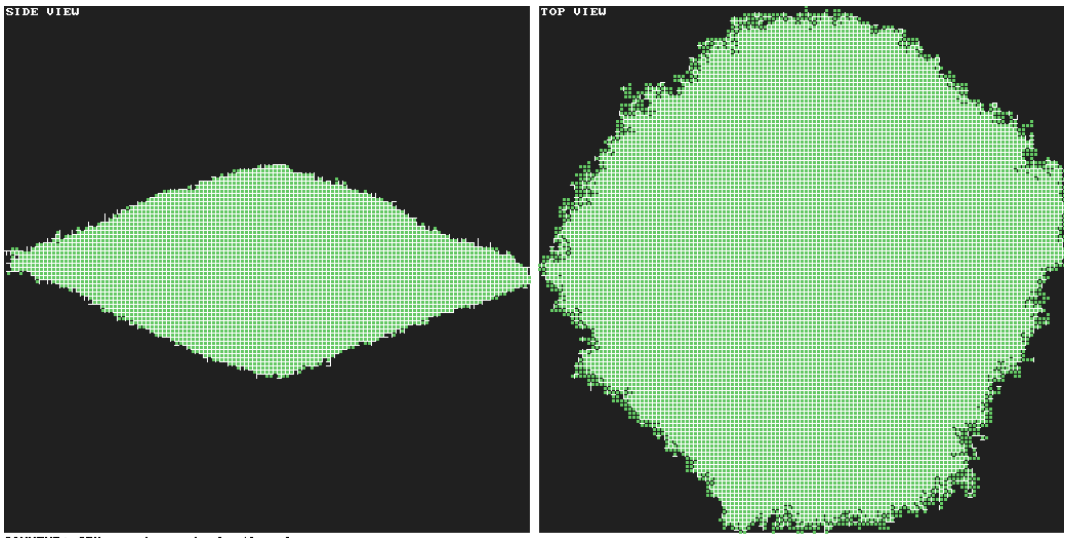
Obr. 11: Výsledný tvar trhliny na GPU s parametrami $K_1 = 1$ a $K_2 = 1$



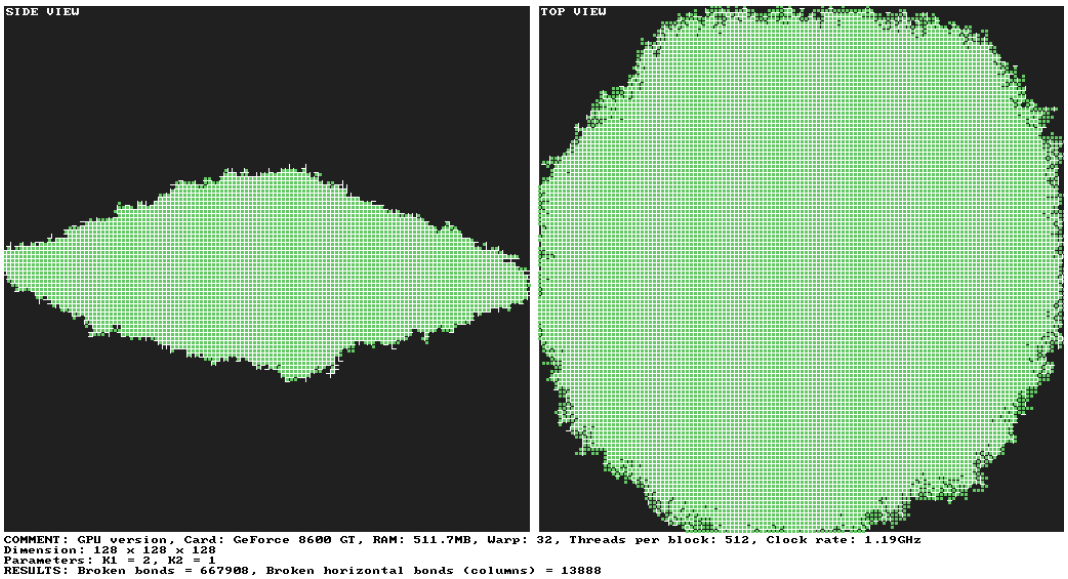
Obr. 12: Výsledný tvar trhliny na CPU s parametrami $K_1 = 1$ a $K_2 = 2$



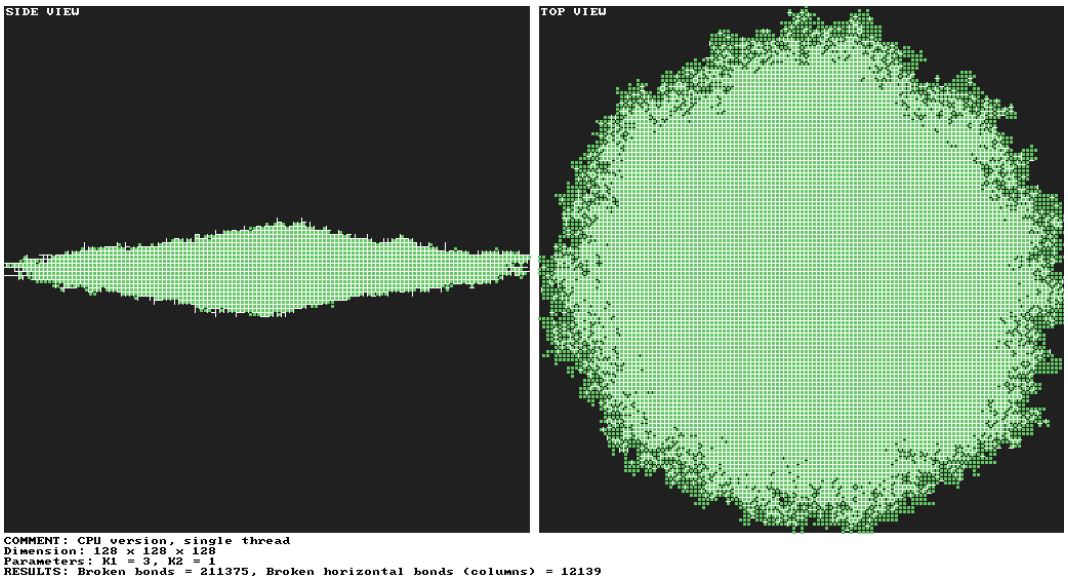
Obr. 13: Výsledný tvar trhliny na GPU s parametrami $K_1 = 1$ a $K_2 = 2$



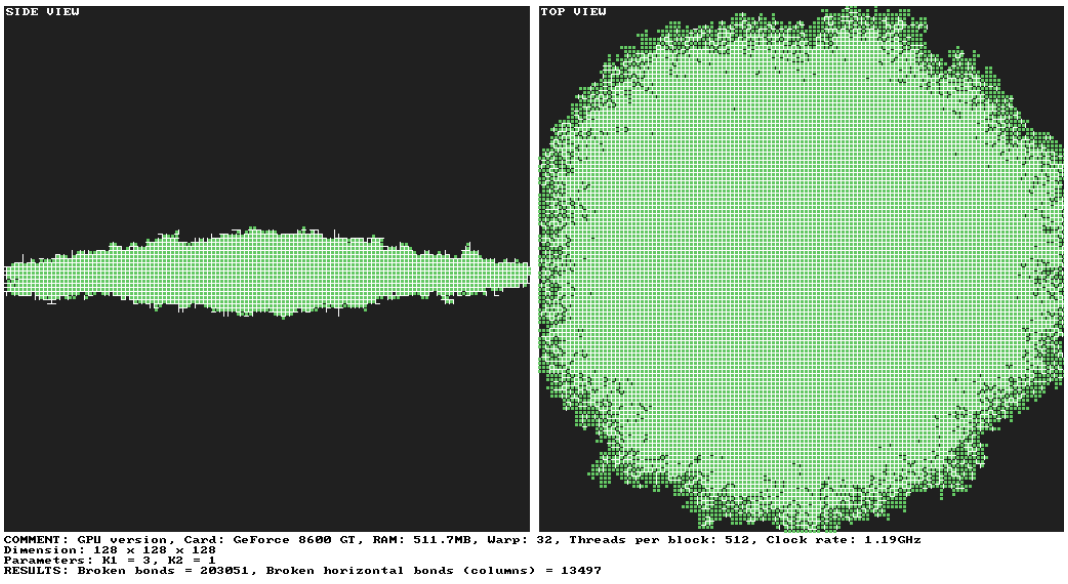
Obr. 14: Výsledný tvar trhliny na CPU s parametrami $K_1 = 2$ a $K_2 = 1$



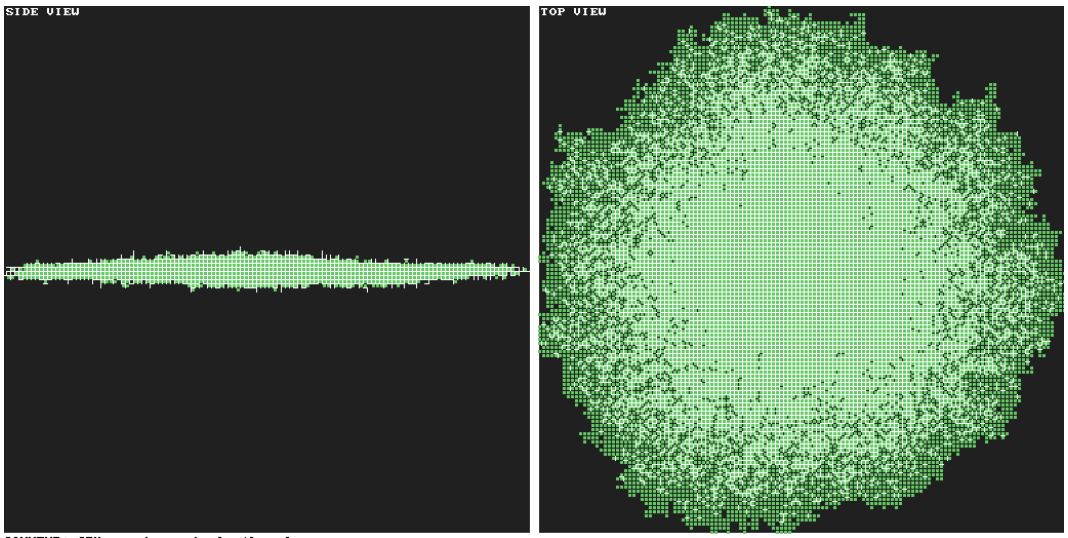
Obr. 15: Výsledný tvar trhliny na GPU s parametrami $K_1 = 2$ a $K_2 = 1$



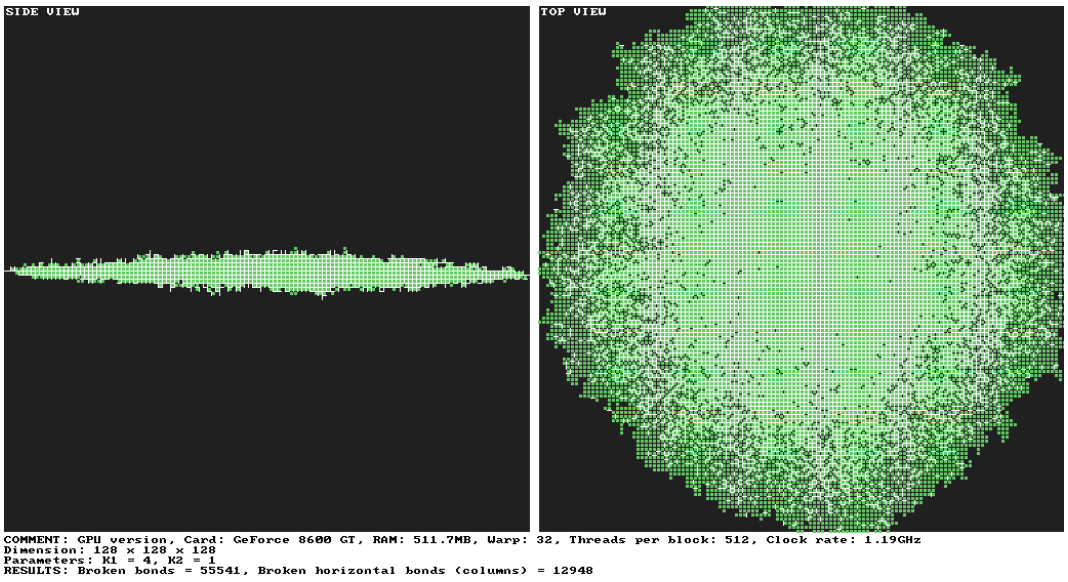
Obr. 16: Výsledný tvar trhliny na CPU s parametrami $K_1 = 3$ a $K_2 = 1$



Obr. 17: Výsledný tvar trhliny na GPU s parametrami $K_1 = 3$ a $K_2 = 1$



Obr. 18: Výsledný tvar trhliny na CPU s parametrami $K_1 = 4$ a $K_2 = 1$



Obr. 19: Výsledný tvar trhliny na GPU s parametrami $K_1 = 4$ a $K_2 = 1$