Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Možnosti využití GPGPU pro šachovou AI

Bc. Ondřej Kála

Vedoucí práce: Ing. Ivan Šimeček, Ph.D.

30. dubna 2015

Poděkování

Děkuji vedoucímu své diplomové práce Ing. Ivanu Šimečkovi, Ph.D. za přijetí mého tématu a cenné rady pro vypracování této diplomové práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 30. dubna 2015 .....

Odkaz na tuto práci

Kála, Ondřej. *Možnosti využití GPGPU pro šachovou AI.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato diplomová práce se zabývá možnostmi využití GPGPU pro šachovou AI. Práce obsahuje rešerši existujících předních šachových AI. Další součástí práce je analýza typických algoritmů pro hry dvou hráčů a diskuze možností jejich použití pro šachovou AI využívající GPU. Hlavním obsahem práce je návrh, implementace a testování proof-of-concept šachové AI využívající GPU.

Klíčová slova Šachy, AI, GPGPU, Teorie her, Hry dvou hráčů, MiniMax, Alfa-beta ořezávání, Aspirační prohledávání, OpenCL.

Abstract

This thesis focuses on possibilities of GPGPU usage in a chess engine. Research of existing top chess engines is included. Part of this thesis is an analysis of typical two player game algorithms and a discussion of possibilities of usage of those algorithms for a GPU chess engine. The main focus of this thesis is design, implementation and testing of a GPU engine proof-of-concept.

Keywords Chess, AI, GPGPU, Game theory, Two player games, MiniMax, Alpha-beta pruning, Aspiration search, OpenCL.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době, kdy šachová umělá inteligence (AI) [5] běžící na běžném počítači, dokáže porazit i mistra světa, už nikdo nepochybuje, že lidská dominance nad stroji v této hře skončila. Nestalo se tak pouze díky rostoucímu výkonu počítačů, ale převážně kvůli neustálému zlepšování šachových AI.

Samotná hrubá síla u tak komplexní hry, jako jsou šachy, počítači k vítězství nestačí. Existuje přibližně 10^{40} legálních pozic a 10^{120} možných šachových partií [6]. Při snaze o prohledávání celého stavového prostoru hry, by AI nedokázala porazit ani průměrného hráče.

Proto musí AI využívat pokročilých heuristik a na základě odhadu výhodnosti dané pozice pro každého z hráčů významně ořezávat prohledávaný stavový prostor.

Základním algoritmem, který se pro prohledávání možných pozic používá, je MiniMax s alfa-beta ořezáváním [7], který se rozšiřuje o mnoho vylepšení. V posledních letech se s nástupem procesorů s více jádry začala používat i paralelní verze tohoto algoritmu.

Zatím se ale nijak neprosadilo využití GPGPU (General-Purpose computing on Graphics Processing Units) [8] pro účel šachových AI. Tato práce si klade za cíl zjistit omezení, která za tím stojí a pokusit se navrhnout, jak GPU pro potřeby šachové AI využít. Na základě zjištěných poznatků by měl vzniknout proof-of-concept šachové AI využívající GPU.

Kapitola 1

Cíle práce

1.1 Zadání

Cíle této práce vycházejí z bodů zadání, které jsou tyto:

1. Proveďte řešerši existujících šachových programů na bázi umělé inteligence (AI).
2. Porovnejte výhody a nevýhody CPU a GPU z hlediska využití pro šachovou AI.
3. Analyzujte možnosti využití různých heuristik pro implementaci šachové AI využívající GPU.
4. Implementujte proof-of-concept šachové AI využívající GPU.
5. Otestujte herní výkon vytvořené AI.

1.2 Analýza

Do analytické části této práce patří druhá, třetí, čtvrtá a pátá kapitola. Cílem druhé kapitoly je seznámení čtenáře s teorií her [9] a typickými algoritmy a heuristikami pro hry dvou hráčů.

Třetí kapitola obsahuje rešerši existujících šachových AI (enginů) a definice pojmů z oblasti šachů a šachových enginů. Tato kapitola vychází z prvního bodu zadání.

Čtvrtá kapitola porovnává rozdíly mezi CPU a GPU z hlediska použití pro šachovou AI. Tato kapitola vychází z druhého bodu zadání.

Pátá kapitola diskutuje možnosti využití algoritmů a heuristik představených ve druhé kapitole pro šachovou AI využívající GPU. Tato kapitola vychází ze třetího bodu zadání.

1.3 Návrh

Návrhová část této práce se zabývá návrhem proof-of-concept šachového enginu využívajícího GPU s využitím poznatků získaných z analytické části práce. Tato kapitola ukazuje, jak využít výpočetní sílu GPU pro šachovou AI. Návrhem se zabývá kapitola č. 6.

1.4 Implementace

Čtvrtým bodem zadání je implementace proof-of-concept šachového enginu využívajícího GPU. Implementovaný proof-of-concept by měl prokázat, že šachový engine využívající GPU je realizovatelný. Implementací se zabývá kapitola č. 7.

1.5 Testování

Posledním bodem zadání je otestování herního výkonu vytvořené šachové AI. To bude provedeno pomocí porovnání parametrů implementovaného proof-of-concept šachového GPU enginu s parametry šachového CPU v kapitole č. 8. Zároveň tato kapitola obsahuje mnoho měření provedených během implementace a vyvozuje z nich závěry využité při vývoji.

Kapitola 2

Teorie her a související algoritmy

2.1 Teorie her

Teorie her [9] je disciplína aplikované matematiky analyzující konfliktní rozhodovací situace. Cílem je sestavit matematický model konfliktu a nalézt nejlepší strategie pro účastníky konfliktů. Modely z oblasti teorie her jsou založeny na předpokladu racionality, tzn. že se očekává, že každý hráč se chová tak, aby maximalizoval svůj zisk.

2.1.1 Typy her

Hry je možno rozdělit podle mnoha různých hledisek. Patří mezi ně počet hráčů, počet strategií, typ výhry, způsob provádění tahů, dostupnost informací a možnost spolupráce.

2.1.1.1 Počet hráčů

U her se uvažuje konečný počet hráčů a minimální počet hráčů 2. Mezi hry dvou hráčů patří šachy, dáma nebo piškvorky.

2.1.1.2 Počet strategií

Počet strategií může být konečný nebo nekonečný. Při nekonečném počtu strategií může záležet i na načasování tahů.

2.1.1.3 Typ výhry

Hry se dělí na hry s konstantním součtem a hry s nekonstantním součtem. U her s konstantním součtem je součet výher všech hráčů vždy konstantní. Speciálním případem jsou hry s nulovým součtem, kde výhra jednoho hráče je vždy tvořena prohrou druhého hráče. Při takové hře jsou hráči v antagonistickém konfliktu, zájmy hráčů jsou neslučitelné. U her s nekonstantním součtem

mohou být v konfliktu neantagonistickém, kdy každý sleduje své cíle, které nemusí být v rozporu s cíli ostatních hráčů.

2.1.1.4 Způsob provádění tahů

Strategické hry jsou takové hry, kde hráči provádějí své tahy současně. Tahové hry jsou tvořeny sekvencí tahů, při kterých se hráči střídají.

2.1.1.5 Dostupnost informací

Hry s úplnou informací jsou takové hry, ve kterých má každý hráč stejné informace o hře. U her s neúplnou informací může mít každý hráč jinou množinu informací, například zná svoje karty, ale nezná karty soupeře.

2.1.1.6 Možnost spolupráce

Kooperativní hry umožňují hráčům vytváření koalicí za účelem maximalizace zisku, nekooperativní hry tuto možnost neumožňují.

2.1.2 Hry v explicitním tvaru

Explicitní tvar slouží pro formalizaci her, ve kterých záleží na pořadí tahů. Hra v explicitním tvaru je reprezentována jako strom. Každý uzel stromu odpovídá stavu hry, ve kterém některý z hráčů volí tah. Hrany reprezentují možné tahy, které převedou hru z jednoho stavu do druhého.

2.1.3 Šachy z pohledu teorie her

Šachy jsou z hlediska teorie her tahovou hrou dvou hráčů s konečným počtem strategií, nulovým součtem, s úplnou informací, bez možnosti spolupráce. Je možné je reprezentovat v explicitním tvaru jako strom pozic.

2.2 MiniMax

Algoritmus MiniMax [7] slouží k nalezení optimálního tahu ve hře dvou hráčů. Algoritmus využívá stromu se dvěma druhy uzlů. Každá skupina uzlů reprezentuje tahy jednoho hráče.

Uzly hráče, pro kterého hledáme optimální tah se nazývají MAX uzly. Cílem v MAX uzlu je maximalizace hodnoty podstromu tohoto uzlu.

Uzly protihráče se nazývají MIN uzly. Cílem v MIN uzlu je minimalizace hodnoty podstromu tohoto uzlu.

Listy stromu jsou ohodnoceny pomocí *evaluační funkce* [10], což je funkce, jejímž vstupem je stav hry a výstupem číslo určující výhodnost pozice pro hráče, který je za daného stavu hry na tahu. Kladné číslo obvykle znamená

výhodu pro tohoto hráče a čím vyšší toto číslo je, tím výraznější je výhoda nad druhým hráčem.

Ohodnocení listů je propagováno přes jejich předky až do kořene pomocí MIN a MAX selekcí, čímž je získána evaluace uzlu reprezentujícího počáteční pozici.

Pseudokód algoritmu MiniMax je následující:

```
# funkce maximalizující evaluaci
int maxi( int depth ) {
    # dosažena požadovaná hloubka prohledávání - evaluace
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    # výběr maxima z minim podstromů
    for ( all moves) {
        score = mini( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}

# funkce minimalizující evaluaci
int mini( int depth ) {
    # dosažena požadovaná hloubka prohledávání - evaluace
    if ( depth == 0 ) return -evaluate();
    int min = +oo;
    # výběr minima z maxim podstromů
    for ( all moves) {
        score = maxi( depth - 1 );
        if( score < min )
            min = score;
    }
    return min;
}
```

2.3 NegaMax

Algoritmus NegaMax [11] je variací algoritmu MiniMax využívající skutečnosti, že

$$max(a, b) = -min(-a, -b)$$

Evaluace pro prvního hráče je v tomto algoritmu rovna negaci evaluace pro druhého hráče a naopak. Tento vztah zjednodušuje implementaci algoritmu.

NegaMax vyžaduje evaluační funkci, která bere v potaz, který hráč je na tahu, zatímco u MiniMaxu je evaluační funkce nezávislá. Z výpočetního hlediska je algoritmus shodný s algoritmem MiniMax.

Pseudokód algoritmu NegaMax je následující:

```
int negaMax( int depth ) {
    # dosažena požadovaná hloubka prohledávání - evaluace
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    # výběr maxima z minim podstromů
    for ( all moves)  {
        score = -negaMax( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}
```

2.4 MiniMax s alfa beta ořezáváním

Alfa-beta ořezávání [7] je způsob ořezání stavového prostoru algoritmu MiniMax tak, aby nedocházelo k prohledávání podstromů uzlů reprezentujících stavy po provedení tahů, které nemohou vést ke změně evaluace aktuálního uzlu.

Parametry α a β jsou hraniční hodnoty využívané při kalkulaci, které omezují množinu možných řešení na základě už prohledané části stromu. Parametr α je maximální spodní hranice možných řešení. Parametr β je minimální horní hranice možných řešení.

Uzel n je považován za možnou cestu k řešení, pokud platí $\alpha <= N <= \beta$, kde N je odhad hodnoty uzlu n získaný pomocí evaluační funkce.

Pseudokód [3] algoritmu MiniMax s alfa-beta ořezáváním je následující:

```
# funkce maximalizující evaluaci
def alphaBetaMax (alpha, beta, depthleft):
   # dosažena požadovaná hloubka prohledávání - evaluace
   if (depthleft == 0): return evaluate()
   # větvení pro každý možný tah
   for (all moves):
      score = alphaBetaMin(alpha, beta, depthleft - 1 )
      if( score >= beta):
         return beta # beta oříznutí
      if( score > alpha ):
         alpha = score # alpha funguje jako max v MiniMaxu
```

```
   return alpha
end

# funkce minimalizující evaluaci
def alphaBetaMin(alpha, beta, depthleft )
   # dosažena požadovaná hloubka prohledávání - evaluace
   if ( depthleft == 0): return -evaluate()
   # větvení pro každý možný tah
   for (all moves):
      score = alphaBetaMax( alpha, beta, depthleft - 1 )
      if( score <= alpha ):
         return alpha # alfa oříznutí
      if( score < beta ):
         beta = score  # beta funguje jako min v MiniMaxu
   return beta
end
```

2.4.1 Pořadí prohledávání uzlů

Algoritmus MiniMax s alfa-beta ořezáváním je silně závislý na pořadí prohledávání uzlů. Hodnoty parametrů α a β získané v počáteční fázi běhu algoritmu mají velký vliv na to, jak velká část stavového prostoru bude oříznuta.

K určení vhodného pořadí prohledávání uzlů se používají různé heuristiky, podle kterých se seřadí potomci každého uzlu do pořadí, ve kterém budou prohledány algoritmem MiniMax.

Jednou z možností určení pořadí prohledávání tahů je prohledávání podle vynechaného tahu, což je technika, kdy je jednomu z hráčů povoleno udělat dva tahy v řadě a poté je vyhodnocena vzniklá pozice.

Pokud nemá hráč, který provedl dva tahy výhodu, téměř jistě to znamená, že první tah nevytvořil žádnou hrozbu. Pokud provedení dvou tahů v řadě vede k výhodě, je první z tahů dobrým kandidátem pro přednostní prohledání do větší hloubky.

2.4.2 Vylepšení algoritmu MiniMax s alfa-beta ořezáváním

Kromě samotného alfa-beta ořezávání je běžné algoritmus MiniMax rozšiřovat o další vylepšení, které vedou k ještě výraznějšímu ořezání stavového prostoru. Jedná se například o iterativní prohlubování [5], aspirační prohledávání [12], algoritmus Scout [13] nebo Principal Variation Search (PVS) [14].

2.4.2.1 Iterativní prohlubování

Iterativní prohlubování [5] je postup, kdy se opakovaně spouští prohledávání do omezené hloubky, která se zvyšuje s každým voláním. Tato strategie se

používá pro situace, kdy musíme vybrat tah v zadaném časovém limitu.

Ve chvíli kdy přidělený čas skončí, použije se výsledek z posledního dokončeného prohledávání. V danou chvíli běžící prohledávání je ukončeno a není použito.

Zároveň je možné použít evaluace uzlů z předchozích běhů pro řazení tahů v následujícím běhu a dosáhnout tak výraznějšího ořezávání. Většinou se tahy, které byly vyhodnoceny jako nejlepší v předchozím běhu, prohledávají jako první v běhu následujícím.

2.4.2.2 Aspirační prohledávání

Aspirační prohledávání [12] je speciálním případem alfa-beta ořezávání, kdy pro počáteční hodnoty parametrů α a β neplatí $\alpha = -\infty$ a $\beta = \infty$. Hodnoty parametrů α a β místo toho tvoří interval okolo očekávané hodnoty evaluace.

Ta je odhadnuta podle statické evaluace nebo prohledání do menší hloubky při iterativním prohlubování. V případě dostatečně přesného odhadu evaluace toto prohledávání ušetří prohledávání velké části stavového prostoru díky restriktivnějším hodnotám parametrů α a β.

Pokud například prohledávání do hloubky 6 vrátí evaluaci 3, můžeme spustit prohledávání do hloubky 8 s parametry $\alpha = 2$ a $\beta = 4$. Pokud bude výsledná evaluace v daném intervalu, dojde ke zpřesnění evaluace za velmi nízkou cenu, neboť zvolené počáteční hodnoty α a β povedou k většímu oříznutí stavového prostoru.

Pokud aspirační prohledávání vrátí evaluaci rovnou α, znamená to že evaluace nespadá do zvoleného intervalu a je nižší než α. Musíme tedy prohledávání spustit znovu, s nižší hodnotou parametru α. Parametr β zůstává shodný.

Naopak pokud aspirační prohledávání vrátí evaluaci rovnou β, je evaluace vyšší než β a musíme prohledávání spustit znovu, s vyšší hodnotou parametru β. V tomto případě se nemění hodnota parametru α.

Speciálním případem aspiračního prohledávání je prohledávání s nulovým (někdy také minimálním) oknem, které používá pouze parametr β. Jako hodnota α se vždy použije $\alpha = \beta - 1$.

Nulové okno se používá pro rychlé ověřování, zda má podstrom uzlu horší evaluaci než uzel, který jsme si vybrali k prohledání. Nezískáme z něho přesnou evaluaci podstromu, ale pouze informaci, zda je vyšší nebo nižší než parametr β. Prohledávání s nulovým oknem bylo poprvé použito v algoritmu Scout a následně v algoritmu PVS.

2.4.2.3 Algoritmus Scout

Algoritmus Scout [13] je rozšířením alfa-beta algoritmu představené v roce 1980. Jeho autorem je Judea Perl. Jedná se o první algoritmus, který výkonově předčil alfa-beta ořezávání a byla dokázána jeho asymptotická optimálnost.

Principem algoritmu je předpoklad, že první prohledávaný uzel v každé hloubce je ten nejlepší. Pro dobré fungování algoritmu je tedy kritické, aby byly tahy co nejlépe seřazeny, například podle předchozích běhů prohledávání do menší hloubky.

Pro ostatní tahy je následně ověřeno, zda nejsou lepší než tah, o kterém předpokládáme, že je nejlepší. To se provádí pomocí prohledávání s nulovým oknem. Pokud ověření selžou, algoritmus pokračuje jako běžné alfa-beta ořezávání.

2.4.2.4 Principal Variation Search

Algoritmus Principal Variation Search [14] je speciálním případem alfa-beta algoritmu s iterativním prohlubováním a aspiračním prohledáváním s nulovým oknem. PVS vychází z algoritmu Scout a jeho autory jsou Tony Marsland a Murray Campbell.

Ideou PVS je, že pro většinu uzlů nepotřebujeme přesnou evaluaci, ale postačí nám určit, zda jsou pro daného hráče nepřijatelné. Přesné skóre potřebujeme určit pouze pro hlavní variaci (principal variation), což je sekvence tahů přijatelná pro oba hráče. V této variaci nedochází k žádným oříznutím a její výsledek je propagován do kořene stromu.

Při opakovaných iteracích PVS se pouze tahy hlavní variace z předchozího běhu prohledávají s parametry $\alpha = -\infty$ a $\beta = \infty$. Ostatní tahy se prohledávají s nulovým oknem okolo α. Pokud vrátí vyšší hodnotu, jsou prohledány plnohodnotně.

Při dobrém seřazení tahu tato strategie sníží počet prohledaných uzlů přibližně o 10%. PVS se někdy vyřazuje pro uzly ve hloubce, kde do maximální prohledávané hloubky zbývají dva tahy, neboť mělké prohledávání do hloubky 2 s parametry $\alpha = -\infty$ a $\beta = \infty$ již není o tolik náročnější než prohledávání s nulovým oknem.

Pseudokód algoritmu PVS je následující:

```
int pvSearch( int alpha, int beta, int depth ) {
   # dosažena požadovaná hloubka prohledávání - evaluace
   if( depth == 0 ) return evaluate()
   bool bSearchPv = true;
   # ověření každého tahu
   for ( all moves)  {
      make # provedení tahu
      if ( bSearchPv ) {
         score = -pvSearch(-beta, -alpha, depth - 1);
      } else {
         score = -pvSearch(-alpha-1, -alpha, depth - 1);
         if ( score > alpha )
          # opakování prohledání
```

```
            score = -pvSearch(-beta, -alpha, depth - 1);
      }
      unmake # vrácení tahu
      if( score >= beta )
         return beta;   # beta-oříznutí
      if( score > alpha ) {
         alpha = score; # beta funguje jako min v MiniMaxu
         bSearchPv = false;
      }
   }
   return alpha; # alfa-oříznutí
}
```

Kapitola 3

Rešerše existujících šachových AI

3.1 Pojmy z oblasti šachů

Následuje sekce vysvětlující některé pojmy z oblasti šachů použité v rešerši.

3.1.1 Pravidlo 50 tahů

Pravidlo 50 tahů [15] stanovuje, že partie, ve které se za posledních 50 tahů nepohnul ani jeden pěšec a nebyl sebrán ani jeden kámen, končí remízou. Jeho cílem je zamezit nekonečným hrám, ve kterých ani jeden hráč nemůže získat výhodu.

3.1.2 Pravidlo o třetím opakování pozice

Pravidlo o třetím opakování pozice [15] říká, že pokud se během partie vyskytne na šachovnici třikrát stejná pozice a pokaždé je na tahu stejný hráč, má tento hráč možnost uplatnit právo na remízu. Cílem tohoto pravidla je ukončit hru v situaci, kdy je pro hráče jedinou šancí, jak partii neprohrát nebo se nedostat do nevýhody opakovat stejné tahy.

3.1.3 Dvojpěšec/trojpěšec

Dvojpěšec nebo dvojitý pěšec je označení pro pěšcovou strukturu, kdy jsou dva pěšci stejné barvy na jednom sloupci. K tomuto dochází po sebrání soupeřovy figury pěšcem.

Podobně tři pěšci shodné barvy na jednom sloupci se označují jako trojpěšec. K výskytu více než tří pěšců jedné barvy na stejném sloupci prakticky nedochází. Teoreticky je ale možný výskyt až šesti pěšců shodné barvy na jednom sloupci.

3.1.4 Opožděný pěšec

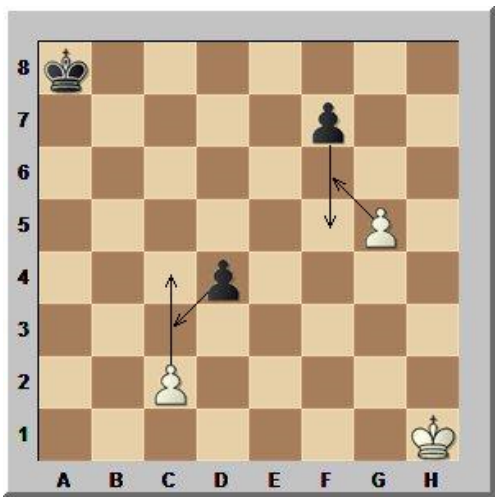
Opožděný pěšec je takový pěšec, který se nachází blíže ke svému počátečnímu poli než jeho sousední pěšci nebo pěšec. Díky tomu nemůže být jednoduše kryt jiným pěšcem a stává se typickým cílem pro soupeře.

3.1.5 Izolovaný pěšec

Izolovaný pěšec je pěšec, jehož oba sousední pěšci (nebo jediný sousední pěšec v případě pěšců na krajních sloupcích) byli sebráni soupeřovými figurami nebo proměněni v jinou figuru. Nemůže proto už nikdy být kryt jiným pěšcem a je lehkým cílem pro soupeře.

3.1.6 Braní pěšce mimochodem

Braní pěšce mimochodem ("en passant") [15] je speciálním případem tahu, kdy jeden pěšec bere druhého. Pokud první hráč provede tah pěšcem z původní pozice o dvě pole najednou a pěšec druhého hráče napadá pole, přes které pěšec prvního hráče prochází, může druhý hráč v následujícím tahu provést speciální tah, kterým sebere pěšce prvního hráče, jako by se nacházel na poli, přes které procházel a ne až na poli následujícím.



Obrázek 3.1: Braní pěšce mimochodem. Převzato z [1].

Pěšec druhého hráče se přesune na pole, přes které pěšec prvního hráče procházel a pěšec prvního hráče je odstraněn z šachovnice. Tento tah může druhý hráč provést pouze ihned v následujícím tahu, v dalších tazích už není možný. Pokud je braní en passant jediným legálním tahem v pozici, musí být provedeno.

3.1.7 Forsyth–Edwards Notation

Forsyth–Edwards Notation (FEN) [16] je standardní notace pro popis stavu šachové hry pomocí textu. Tato notace pochází z 19. století a jejím autorem je skotský novinář David Forsyth.

Hlavní součástí textové reprezentace je rozmístění figur na šachovnici. Figury jsou reprezentovány pomocí prvního písmena svého jména v angličtině s výjimkou jezdce, který je reprezentován písmenem "n". Bílé figury jsou reprezentovány velkými písmeny a černé malými.

Dále textová reprezentace obsahuje práva obou hráčů provést malou a velkou rošádu, pole na kterém je možno brát en passant a informací o tom, kdo je na tahu, kolik tahů uplynulo od začátku hry a kolik tahů uplynulo od posledního pohybu pěšce nebo sebrání figury (kvůli pravidlu 50 tahů). FEN notace ale neobsahuje dostatek informací k rozhodnutí o tom, zda v dané pozici lze uplatnit pravidlo o třetím opakování pozice.

FEN notace počáteční pozice šachové hry vypadá takto:

$$rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNRwKQkq - 01$$

3.1.8 Systém hodnocení Elo

Elo [17] je systém hodnocení pro kalkulaci relativní úrovně dovednosti v kompetitivních hrách hráče proti hráči. Jeho tvůrcem je maďarský fyzik Arpad Elo. V tomto systému je každému hráči přiřazeno hodnocení dané kladným přirozeným číslem.

Rozdíl v hodnocení hráčů vyjadřuje, kolik procent vzájemných her by měl lepší hráč vyhrát. Při rozdílu 100 bodů se očekává, že výše hodnocený hráč vyhraje 64% her, při rozdílu 200 bodů je to 76% her.

Vzorec pro výpočet pravděpodobnosti výhry hráče A ve hře proti hráči B, kde E_a je očekávaná šance výhry hráče A, R_a je hodnocení hráče A a R_b hodnocení hráče B, má tento tvar:

$$E_a = \frac{1}{1 + 10^{(R_b - R_a)/400}}$$

Hodnocení hráčů jsou upravena po každé hře, pokud není rozdíl v hodnocení příliš vysoký. Čím vyšší je rozdíl v hodnocení, tím více bodů je při výhře slabšího hráče odečteno z hodnocení silnějšího hráče a přičteno k hodnocení slabšího hráče.

Při výhře silnějšího hráče je přesun bodů nejvyšší, pokud je rozdíl mezi hráči nejmenší. V případě remízy je přesun bodů od silnějšího hráče k slabšímu nižší, než v případě prohry silnějšího hráče.

Vzorec pro výpočet Elo hodnocení hráče, kde R je hodnocení hráče, S je suma hodnocení soupeřů, W je počet výher, L počet proher a G počet her, má tento tvar:

$$R = \frac{S + 400 \times (W - L)}{G}$$

3.2 Pojmy z oblasti šachových AI

Následuje sekce vysvětlující některé pojmy z oblasti šachových AI použité v rešerši.

3.2.1 Evaluační funkce pro šachy

Pro šachy se obvykle používá evaluační funkce, jejímž vstupem je stav hry a výstupem číslo s plovoucí desetinnou čárkou určující výhodnost pozice pro hráče s bílými figurami. Toho se dosáhne pomocí změny znaménka v případě, že je na tahu hráč s černými figurami. Poté kladné číslo vždy znamená výhodu pro bílého hráče a záporné číslo výhodu pro černého hráče, ať je na tahu kterýkoliv z nich.

3.2.2 Transpoziční tabulka

Transpoziční tabulka [18] je hashovací tabulka [19] obsahující informace o prohledaných pozicích. Je užitečná zejména při iterativním prohlubování, kdy ze z ní získávají informace o aktuálně prohledávané pozici uložené v předchozích iteracích. Použití transpoziční tabulky je metodou dynamického programování.

Transpoziční tabulka využívá hashovací funkce, která převede šachovou pozici na klíč. Typická délka klíče je 64 bitů. Pro indexování se ale nepoužívá celý klíč, neboť tabulka pro klíče o délce 64 bitů by byla příliš rozměrná.

3.2.3 Bitboard

Bitboard [20] je způsob reprezentace šachovnice, který urychluje mnoho operací. Místo typické reprezentace pomocí pole pozic na šachovnici s hodnotami reprezentujícími figury nebo volné místo, je šachovnice reprezentována pomocí několika bitových vektorů o délce 64. Každý bit reprezentuje jedno pole šachovnice. Jeden vektor může reprezentovat pozice bílých pěšců, druhý pozice černých pěšců atd.

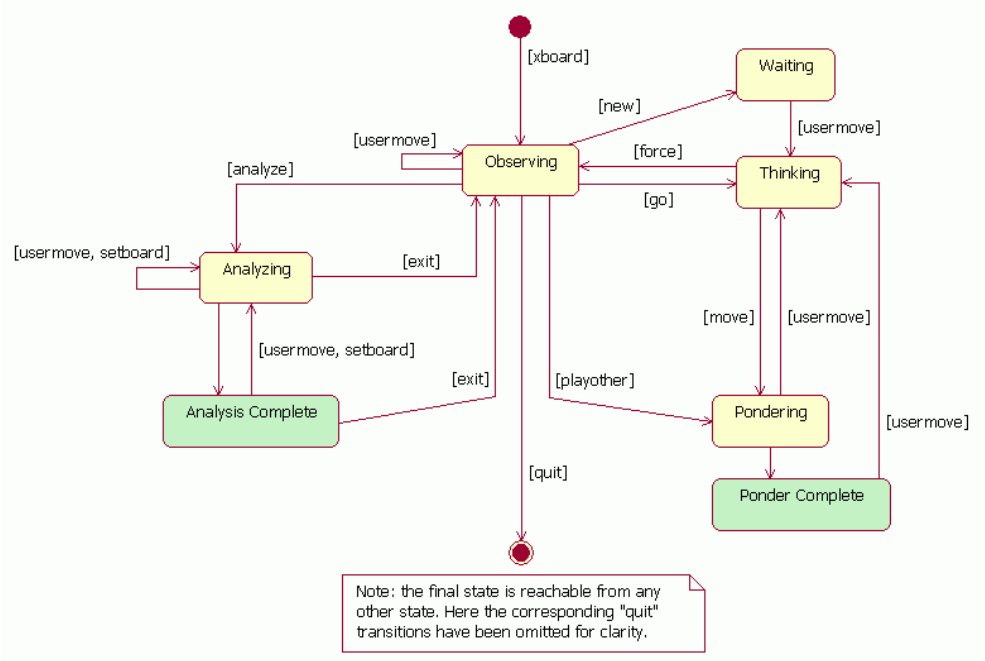
3.2.4 Syzygybases

Syzygybases [21] jsou tabulky pro koncové hry s až 6 figurami na šachovnici. Pro takové pozice je z nich možné určit, která strana hru vyhraje. Velikost těchto tabulek je 161GB, což je velké zlepšení proti Nalimovým tabulkám o velikosti 1,2TB. Syzygybases berou v potaz pravidlo 50 tahů a pravidlo o třetím opakování pozice.

3.2.5 Chess Engine Communication Protocol

Chess Engine Communication Protocol [2], nazývaný též XBoard nebo WinBoard je protokol pro komunikaci mezi šachovými enginy a jinými programy, zejména grafickými uživatelskými rozhraními (GUI). GUI pro Unixové systémy implementující tento protokol se nazývá XBoard, GUI pro operační systému Windows se označuje jako WinBoard.

Tento protokol vyžaduje, aby si engine udržoval v paměti kompletní stav hry. Komunikace mezi enginem a GUI je založena na posílání textových příkazů.



Obrázek 3.2: UML diagram XBoard/WinBoard protokolu. Převzato z [2].

3.2.6 Universal Chess Interface

Universal Chess Interface (UCI)[22] je komunikační protokol pro šachové enginy zajišťující propojení s grafickými uživatelskými rozhraními. Jeho autory jsou Rudolf Huber a Stefan Meyer-Kahlen a toto rozhraní prakticky nahradilo starší Chess Engine Communication Protocol.

UCI počítá s tím, že GUI není pouhý view a controller v MVC (Model-View-Controller) modelu ale částečně zastává i část povinností modelu, například tím že ukládá informace o opakování pozice.

Samotný engine je pak bezstavový a pouze odpovídá na požadavky GUI. Oproti XBoard/WinBoard je takový přístup méně flexibilní. Enginy například

ztrácejí kontrolu nad tím, ve které pozici vzdají hru, nabídnou nebo přijmou remízu. GUI může dokonce v zahájení engine vůbec nepoužít a místo toho zvolit tah z knihy zahájení a podobně v koncové hře vybrat tah podle tabulek pro koncové hry.

3.3 Přehled analyzovaných šachových AI

V této kapitole budou analyzovány šachové AI (enginy) Stockfish, Houdini, Komodo a Rybka. Bude porovnán jejich herní výkon a zjištěno maximum informací o algoritmech, které využívají.

Tabulka 3.1: Typy licencí enginů

Engine	Licence
Stockfish	Open source
Houdini	Komerční
Komodo	Komerční
Rybka	Komerční

3.4 Stockfish

Engine Stockfish [23], aktuálně ve verzi 6, je nejsilnějším enginem na světě. Je používán většinou hráčů na všech úrovních pro analýzu partií. Ani šachoví velmistři pro něho nejsou žádným soupeřem a v partii proti tomuto enginu mohou v nejlepším případě doufat v remízu.

- Open source, GPL licence
- Pro Windows, OS X, Linux, iOS, Android
- Podpora rozhraní UCI a Xboard/WinBoard
- Programovací jazyk C++
- Vychází z enginu Glaurung 2.1
- Podpora pro až 128 vláken
- Autoř:i Tord Romstad, Marco Costalba a Joona Kiiski

3.5 Houdini

Houdini [24] je komerční engine pro Windows vycházející z enginů Ippolit/Robbolito, Stockfish a Crafty. Houdini používá tabulky pro koncové hry Syzygybases a transpoziční tabulku o velikosti až 260GB. Zdrojový kód Houdini není veřejný.

- Komerční engine pro Windows
- Podpora rozhraní UCI a WinBoard
- Starší verze 1.5 zdarma
- Podpora pro až 32 vláken
- Autor: Robert Houdart

3.5.1 Evaluační funkce

Houdini používá kalibrovanou evaluaci - evaluace pozice odpovídá šanci na výhru v této pozici. Evaluace +1.0 znamená 80% šanci na výhru, +2.0 95% šanci na výhru a +3.0 odpovídá 99% šanci na výhru.

3.6 Komodo

Komodo [25] je komerční engine pro Windows, Linux, Mac a Android vycházející z enginu Doch. Oproti ostatním enginům se odlišuje přesnější statickou evaluací na úkor prohledávání do větší hloubky.

- Podpora pro až 64 vláken
- Podpora rozhraní UCI a Xboard/WinBoard
- Programovací jazyk C++
- Autoři: Don Dailey, Mark Lefler a Larry Kaufman

3.6.1 Evaluační funkce

Hodnota figur se v evaluační funkci Komoda liší mezi zahájením a koncovou hrou:

Tabulka 3.2: Hodnota figur

Fáze hry	Pěšec	Jezdec	Střelec	Věž	Dáma
Zahájení	600	3100	3225	4350	9100
Koncová hra	925	3100	3225	5475	9700

3.7 Rybka

Rybka [26] je komerční engine pro Windows. Tento engine vyhrál mistrovství světa šachových enginů čtyřikrát za sebou v letech 2007 až 2010. Rybka používá bitboard reprezentaci šachovnice a alfa-beta algoritmus s širokým aspiračním oknem.

- Starší verze 2.3.2 zdarma
- Podpora rozhraní UCI a Xboard/WinBoard
- Verze Deep podporuje více vláken
- Autor: Vasik Rajlich

3.8 Porovnání herní síly analyzovaných šachových AI

Na porovnání šachových enginů existuje mnoho různých metodik a z nich plynoucích žebříčků. Enginy proti sobě hrají turnaje na určeném hardware s danou časovou kontrolou a na základě jejich výsledků jim je přiřazeno Elo hodnocení, podle kterého je vytvořen žebříček.

3.8.1 CCRL 40/40

CCRL (Computer Chess Rating Lists) 40/40 [27] je jedním z žebříčků pro šachové enginy. Používá partie s časovou kontrolou 40/40 (40 tahů za 40 minut) na stroji s procesorem AMD X2 4600+ o frekvenci 2,4GHz. Povoluje knihy zahájení pro prvních 12 tahů a tabulky pro koncové hry se 3 až 5 zbývajícími figurami na šachovnici. Nepovoluje enginu připravovat se na další tah v době, kdy je na tahu soupeř.

Tabulka 3.3: Pořadí enginů ke dni 3.1.2015

	Engine	Elo
1.	Komodo 8	3303
2.	Stockfish 5	3284
3.	Houdini 4	3273
4.	Fire 4	3226
5.	Gull 2.8b	3200

3.8.2 CCRL 40/4

CCRL 40/4 [27] je jedním z žebříčků pro šachové enginy. Používá partie s časovou kontrolou 40/4 (40 tahů za 4 minuty) na stroji s procesorem AMD X2 4600+ o frekvenci 2,4GHz. Povoluje knihy zahájení pro prvních 12 tahů a tabulky pro koncové hry se 3 až 5 zbývajícími figurami na šachovnici. Nepovoluje enginu připravovat se na další tah v době, kdy je na tahu soupeř.

Tabulka 3.4: Pořadí enginů ke dni 3.1.2015

	Engine	Elo
1.	Stockfish 5	3368
2.	Komodo 8	3357
3.	Houdini 4	3337
4.	Gull 3	3272
5.	Critter 1.6a	3231

3.8.3 CEGT 40/4

CEGT (Chess Engines Grand Tournament) 40/4 [28] je jedním z žebříčků pro šachové enginy. Používá partie s časovou kontrolou 40/4 (40 tahů za 4 minuty) na stroji s procesorem o frekvenci 2GHz a při využití až 12 jader. Nepovoluje enginu připravovat se na další tah v době, kdy je na tahu soupeř.

Tabulka 3.5: Pořadí enginů ke dni 11.1.2015

	Engine	Elo
1.	Stockfish 5	3312
2.	Komodo 8	3308
3.	Houdini 4	3249
4.	Gull 3	3224
5.	Rybka 4.1	3128

3.8.4 IPON rating list

IPON Rating List [29] je jedním z žebříčků pro šachové enginy. Používá partie s časovou kontrolou 5min + 3s za každý tah. Průměrná délka partie je tedy okolo 16min. Využívá procesor Phenom 2 o frekvenci 3,2GHz. Každý engine využívá jedno jádro procesoru. Hry se hrají ze 110 různých pozic ze známých zahájení a pro koncové hry se používají tabulky pro pozice s až 4 figurami. Enginu je povoleno připravovat se na další tah v době, kdy je na tahu soupeř.

Tabulka 3.6: Pořadí enginů ke dni 8.1.2015

	Engine	Elo
1.	Komodo 8	3146
2.	Stockfish 5	3144
3.	Houdini 4	3134
4.	Gull 3	3075
5.	Equinox 3	3008

3.8.5 Závěr porovnání

Z žebříčků šachových enginů plyne, že nejsilnějšími enginy jsou Stockfish 5, Komodo 8 a Houdini 4. Stockfish 5, který je jediný z této trojice open source, je tedy ideálním zdrojem pro inspiraci při vývoji vlastního engine. Algoritmus enginu Stockfish byl proto podroben detailní rešerši.

3.9 Detailní rešerše algoritmu enginu Stockfish

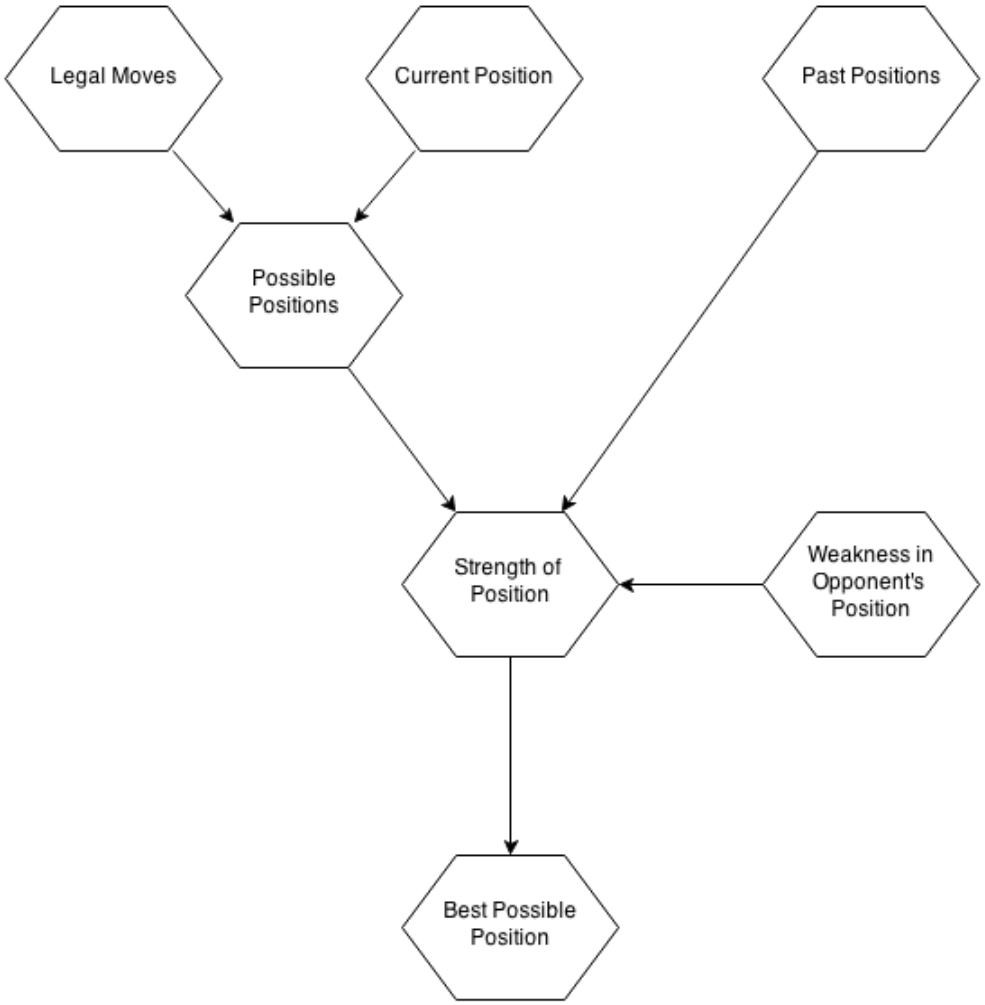
Základem enginu Stockfish je algoritmus využívající alfa-beta ořezávání využívající sadu funkcí pro analýzu pozice v závislosti na materiálu na šachovnici. [3] Díky tomu se algoritmus jinak chová během zahájení, střední hry a koncové hry. Pro koncové hry Stockfish může využívat předpočítáné tabulky Syzygy-bases.

Stockfish začíná vytvořením stromu legálních tahů. Následuje evaluace pozic pomocí evaluační funkce. Cílem je evaluovat nejlepší tahy jako první. Toho lze docílit spuštěním mělkého prohledávání (do menší hloubky) a použitím výsledných alfa-beta hodnot pro hlubší prohledávání.

Nalezené tahy jsou řazeny mimo prohledávaný strom. Engine používá transpoziční tabulku. Transpoziční tabulka slouží k tomu, aby stejná pozice nebyla prohledávána více než jednou. Šachovnice je reprezentována bitboardem.

3.9.1 Postup prohledávání stavového prostoru

1. Inicializace uzlu
2. Detekce remízy
3. Ořezávání podle počtu tahů do matu (pokud byl nalezen mat třetím tahem, nemá cenu hledat hlouběji)
4. Prohledání transpoziční tabulky
5. Statická evaluace pozice
6. Razoring - dopředné ořezávání větví, pokud je statická evaluace tahu menší nebo rovna α. Předpokládá, že soupeř bude mít alespoň jeden zlepšující tah v každé pozici.
7. Statické ořezávání podle vynechaného tahu - předpokládá, že soupeř nemá tah, který sníží evaluaci o více než stanovenou hodnotu, pokud vynecháme tah.
8. Prohledávání s vynechaným tahem a ověřovací prohledávání
9. ProbCut - pokud nalezneme dobrý tah beroucí soupeřovu figuru a omezené prohledávání vrátí hodnotu o hodně vyšší než β, můžeme oříznout předchozí tah.



Obrázek 3.3: Jak Stockfish hledá nejlepší tah. Převzato z [3].

10. Interní iterativní prohlubování
11. Smyčka přes všechny pseudolegální tahy ukončená v případě β-oříznutí
12. Rozšířené prohledání šachů a nebezpečných tahů
13. Futility pruning
14. Provedení tahu
15. Prohledání do omezené hloubky
16. Prohledání do plné hloubky, pokud selže prohledání do omezené hloubky
17. Vrácení tahu zpět

18. Hledání nového nejlepšího tahu
19. Kontrola rozdělení
20. Kontrola matu nebo patu
21. Update tabulek, včetně transpoziční

3.10 Evaluační funkce

Evaluační funkce Stockfish bere v potaz strukturu pěšců, pozici figur, volné pěšce a bezpečnost krále.

3.10.0.1 Struktura pěšců

1. Dvojpěšci, opoždění a blokovaní pěšci jsou penalizováni
2. Postup pěšců je podporován, pokud je dostatečně chráněn
3. Je podporována kontrola centra šachovnice pomocí pěšců

3.10.0.2 Struktura pěšců

1. Je podporováno umístění jezdců v centru šachovnice
2. Je podporováno vzájemné krytí dam a věží
3. Je podporováno umístění věží na sedmé řadě šachovnice z pohledu daného hráče

3.10.0.3 Volní pěšci

1. Jsou evaluováni jinak než ostatní pěšci
2. Ověřuje se bezpečnost pěšce před soupeřovými figurami
3. Pěšci těsně před proměnou znamenají velký bonus k evaluaci

3.10.0.4 Bezpečnost krále

1. Je podporováno umístění krále blízko rohu šachovnice během střední hry
2. Je podporováno udržení štítu z pěšců poblíž krále
3. Je podporováno bránění přístupu soupeřových figur ke králi

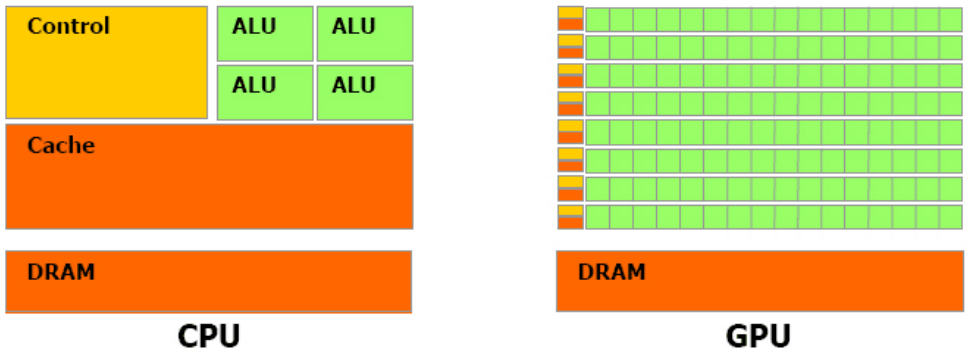
Kapitola 4

Porovnání CPU a GPU z hlediska použití pro šachovou AI

4.1 Obecné porovnání architektury

Architektura současných CPU je založena na čipech s několika výkonnými jádry, které spolu sdílí velké vyrovnávací paměti vyšších úrovní a mají společnou řídící jednotku. To jim zaručuje vysoký výkon v aplikacích využívající nižší počet vláken, z nichž každé může vykonávat zcela odlišné instrukce.

Oproti tomu GPU architektura [4] využívá mnohem vyššího procenta transistorů pro zpracování dat na úkor velikosti vyrovnávacích pamětí a složitosti řídících jednotek. Díky tomu může GPU excelovat při provádění shodných jednoduchých operací nad velkými objemy data.



Obrázek 4.1: Srovnání CPU a GPU architektury. Převzato z [4].

Vyšší počet řídících jednotek, z nichž každá ovládá část jader (streaming

multiprocessors u nVidie, SIMD jednotek u AMD) GPU, vede k vysokému paralelnímu výkonu. Pro využití tohoto výkonu je ale třeba minimalizovat nebo úplně odstranit závislosti mezi úlohami různých jader.

Nedostatky GPU se projeví při komplikovaných výpočtech se závislostmi znemožňujícími paralelizaci úlohy a tím využití celé výpočetní síly GPU. CPU a GPU tak z hlediska využití pro výpočty nejsou přímými konkurenty, ale spíše se doplňují, neboť jsou vhodné pro řešení odlišných úloh.

4.2 Paměť

Pro porovnání paměťových rozdílů mezi CPU a GPU z hlediska využití pro šachovou AI je nejdříve třeba představit strukturu pamětí GPU.

4.2.1 Struktura pamětí GPU

U GPU je v ohledu práce s pamětí oproti CPU mnohem více restrikcí. Není zde možnost přístupu k datům na pevném disku a tedy ani virtuální paměť. Na GPU se nachází několik druhů paměti a technologie CUDA a OpenCL navíc specifikují různé abstrakce nad těmito pamětmi.

CUDA [30] rozděluje paměť na globální, sdílenou, lokální a registry. OpenCL [31] definuje globální paměť, lokální paměť, privátní paměť a paměť pro konstanty. S každým druhem pamětí se pojí určitá omezení.

4.2.1.1 Globální paměť - CUDA i OpenCL

Globální paměť se nachází mimo GPU čip, na PCB (Printed Circuit Board - deska s tištěnými spoji) grafické karty. Proto může mít velkou kapacitu, ale přístup k ní je pomalý. Alokaci globální paměti je nutné provést před samotným během kernelu [32] a její uvolnění po běhu kernelu.

4.2.1.2 Sdílená paměť - CUDA

Sdílená paměť se nachází přímo na GPU čipu a proto je přístup k ní mnohonásobně rychlejší než ke globální paměti. Tato paměť je sdílena vlákny v daném bloku. Je přímo určena pro GPGPU výpočty a pokud je to možné, její použití by mělo být upřednostněno před použitím globální pamětí.

4.2.1.3 Lokální paměť - OpenCL

Lokální paměť je ekvivalentem sdílené paměti v CUDA, ale není u ní zaručeno, že se nachází přímo na GPU čipu. V závislosti na konkrétní grafické kartě se může jednat pouze o abstrakci globální paměti, což sebou nese veškerá výkonová negativa.

4.2.1.4 Lokální paměť - CUDA

Lokální paměť je abstrakcí globální paměti, která je přístupná pouze pro konkrétní vlákno. Používá se pro lokální proměnné, které zabírají hodně paměti a nevejdou se do registrů. Přístup k této paměti je velmi drahý a proto je vhodné se použití velkých lokálních proměnných vyhnout a místo toho používat sdílenou paměť.

4.2.1.5 Privátní paměť - OpenCL

Privátní paměť je přístupná pouze jednotlivému vláknu. Stejně jako u lokální paměti není specifikováno, kde na kartě se reálně nachází. Při využití pouze nízké kapacity to bývají registry na čipu, jakmile jejich kapacita nestačí, jedná se o vyrovnávací nebo globální paměti.

4.2.1.6 Registry - CUDA

Registry se nachází přímo na čipu a jsou nejrychlejším druhem paměti. Zároveň ale disponují nejnižší kapacitou. Jsou přístupné pouze pro dané vlákno a ukládají se do nich lokální proměnné o malé velikosti.

4.2.1.7 Paměť pro konstanty - OpenCL

Paměť pro konstanty je sekcí globální paměti vyhrazenou pouze pro čtení. Její velikost je specifická pro danou grafickou kartu.

4.2.2 Paměťové rozdíly z hlediska využití pro šachovou AI

V oblasti pamětí má CPU z hlediska využití pro šachovou AI jasnou výhodu. V operační paměti si může udržovat transpoziční tabulku velikosti v řádu až gigabajtů a velmi rychle k ní přistupovat.

Situace u GPU je mnohem komplikovanější. Jednou z největších překážek pro využití GPU pro šachovou AI je charakteristika struktury pamětí GPU. Globální paměť GPU je dostatečně velká pro potřeby šachové AI, ale přístup k ní je příliš pomalý. Řešení využívající pouze globální paměť je z výkonové stránky nepoužitelné.

Problémem je i umístění transpoziční tabulky do globální paměti. Přístupy do globální paměti za účelem zjištění, zda už daná pozice nebyla prohledána ve většině případů převáží nad ušetřeným prohledáváním. V případě GPU tedy bude na rozdíl od CPU výhodnější smířit se s vícenásobným prohledáváním stejných pozic, protože přístup do paměti je mnohem dražší.

Lokální a privátní paměť jsou velmi rychlé, ale jejich kapacita je pro potřeby šachové AI příliš nízká. Není možné do této paměti uložit transpoziční tabulku ani veškeré informace o cestě stavovým prostorem z úvodní do aktuální pozice, jako jsou vygenerované možné tahy pro každou pozici.

4.3 Paralelismus

Architektury CPU a GPU mají odlišné přístupy k paralelismu. Zatímco CPU je navrženo s ohledem provádění více nezávislých úloh zároveň, jako je běh více aplikací a služeb najednou, GPU je určeno pro provádění shodné úlohy nad více instancemi dat.

Z tohoto důvodu mají CPU komplexní řadič a jádra [33], která umožňují optimalizace jako záměna pořadí operací, predikce výsledku větví a simultánní provádění více vláken (Intelem nazýváno "Hyper-Threading"[34]). Simultánní provádění více vláken znamená, že v případě, kdy CPU čeká na paměť vyžadovanou jedním vláknem, provádí jiné vlákno.

Rozhodování o tom, které vlákno bude v danou chvíli prováděno, je v režii samotného CPU. To znamená, že veškeré potřebné informace musí být uloženy přímo ve vyrovnávací paměti a registrech CPU. To vede ke složitějšímu CPU, ale výsledkem je výkonový nárůst, přestože reálně není jádrem procesoru prováděno více vláken zároveň.

Oproti tomu GPU je navrženo s převážnou orientací na hrubou sílu, tedy možnost provádět mnohem vyšší počet operací zároveň. GPU obsahuje multiprocesory (SIMD jednotky), které každý dokážou provádět několik SIMD instrukcí nad desítkami čísel v plovoucí desetinné čárce zároveň. [35]

Návrh GPU je lehce rozšiřitelný pomocí přidávání dalších multiprocesorů, neboť jsou na sobě nezávislé. Levnější GPU většinou obsahují menší počet multiprocesorů a mají užší paměťovou sběrnici. Oproti tomu u CPU se rozdíl mezi levnějšími a dražšími modely projevuje spíše frekvencí, na které procesor běží a velikostí vyrovnávacích pamětí, zatímco počet jader je shodný.

4.3.1 SIMD architektura

Architektura "Single instruction, multiple data"(SIMD) vychází ze třídy paralelních počítačů ve Flynnově taxonomii [36]. Je definována přístupem, kdy se nad různými daty paralelně provádějí shodné instrukce. Vhodnými algoritmy pro SIMD zpracování jsou takové algoritmy, které mají vysoký poměr aritmetických operací k paměťovým operacím.

GPU využívají tuto architekturu, neboť jejich hlavním účelem je mapování textur, renderování polygonů a akcelerace geometrických operací. To zahrnuje provádění shodných instrukcí nad velkými bloky dat bez vzájemných závislostí, typicky vektory a maticemi. Je tedy logické tyto operace provádět paralelně.

Cílem při využití GPU pro GPGPU je co nejvíce se přiblížit těmto algoritmům, pro které bylo GPU navrženo, aby bylo možno maximálně využít jeho výkonu. Algoritmus prováděný na GPU by tedy měl pracovat s daty reprezentovanými pomocí vektorů nebo matic a provádět nad každým prvkem shodnou operaci.

4.3.2 Rozdíly v paralelismu z hlediska využití pro šachovou AI

Z hlediska paralelismu při využití pro šachovou AI má CPU nad GPU opět výhodu. Možnost provádět na každém výkonném jádru GPU jedno vlákno programu se pro účely GPU enginu hodí mnohem více než schopnost provádět vysoké množství vláken na více slabších multiprocesorech zároveň.

Šachový engine se soustředí na prohledávání několika kritických variací. Zbytek stavového prostoru je radikálně ořezán. U CPU lze každou z těchto prohledávaných variací přiřadit vláknu běžícímu na jednom z jader procesoru. Vlákna spolu zároveň mohou jednoduše sdílet své výsledky pomocí transpoziční tabulky v operační paměti.

U GPU je něco takového prakticky nemožné. Úkol každého vlákna na GPU by měl být ideálně předem daný a nezávislý na mezivýsledcích výpočtu. Stavový prostor by tedy měl být mezi multiprocesory na GPU rozdělen staticky a ne dynamicky.

Také sdílení výsledků mezi vlákny je na GPU mnohem složitější. V rámci lokálního bloku vláken to díky rychlé lokální paměti a možnosti synchronizace pomocí bariér není problém, ale pro využití plného potenciálu GPU je třeba využít více bloků vláken. Komunikace mezi různými bloky vláken už by ale musela probíhat přes globální paměť a měla by negativní dopad na výkon.

Kapitola 5

Možnosti využití různých algoritmů pro šachovou AI využívající GPU

Tato kapitola diskutuje možnosti využití algoritmů představených v kapitole o teorii her pro šachovou AI využívající GPU. Základní ideou je snaha o použití paralelních verzí těchto algoritmů, které by dokázaly využít potenciálu GPU.

5.1 MiniMax

Základní algoritmus MiniMax bez veškerých vylepšení by se dal jednoduše implementovat v paralelní verzi pro GPU. Jelikož jednotlivé podstromy jsou na sobě zcela nezávislé, na rozdíl třeba od MiniMaxu s alfa-beta ořezáváním, kde se do sousedního podstromu předávají hodnoty parametrů α a β z právě prohledaného podstromu, bylo by možné stavový prostor rozdělit mezi bloky vláken na GPU bez dopadu na výkon algoritmu.

Algoritmus MiniMax je ale bez vylepšení pro tak komplexní hru, jako jsou šachy, nedostatečný. Díky absenci ořezávání stavového prostoru by nedokázal pozici prohledat do dostatečné hloubky. Tento algoritmus není v základní verzi použitelný ani pro CPU engine ani pro GPU engine.

5.2 MiniMax s alfa-beta ořezáváním

Rozšíření o alfa-beta ořezávání značně zlepšuje schopnosti algoritmu MiniMax, zároveň ale znesnadňuje jeho paralelizaci, neboť přináší závislosti mezi sousedními podstromy.

Základní ideou pro paralelizaci je rovnoměrné rozdělení podstromů kořenového uzlu mezi procesory. Nejlepšího výkonu je dosaženo, pokud máme k dispozici alespoň tolik procesorů, kolik je potomků kořenového uzlu.

Díky tomuto rozdělení pak každý z procesorů provádí prohledávání pouze do hloubky $n - 1$, kde n je zadaná hloubka prohledání pro kořenový uzel. Po doběhnutí algoritmu pro každého potomka je z těchto hodnot určena evaluace pro kořenový uzel.

Nevýhodou tohoto přístupu je ztráta přenosu hodnot parametrů α a β mezi podstromy. Pokud u sekvenční verze algoritmu prohledávání v prvním podstromu rychle skončí díky významnému oříznutí, povede to k výraznějšímu ořezávání i v ostatních podstromech.

V případě paralelního algoritmu, ve kterém spolu procesory nijak nekomunikují, k tomuto nedojde a paralelní algoritmus tak ve výsledku může běžet delší dobu než sekvenční verze. Řešením může být zavedení sdílení hodnot parametrů α a β mezi procesory.

Procesor, který by dokončil prohledávání svého podstromu by aktualizoval hodnoty parametrů pro kořenový uzel ve sdílené paměti. Ostatní procesory by si pak na základě těchto hodnot aktualizovaly vlastní hodnoty parametrů α a β.

Tento postup by ale v případě GPU vyžadoval synchronizaci a sdílení dat mezi jednotlivými bloky vláken. Tím by byla porušena jejich nezávislost. Zároveň by toto řešení vedlo k nutnosti pravidelných zápisů a čtení z globální paměti, což by mělo negativní dopady na výkon algoritmu. Paralelní verze algoritmu MiniMax s alfa-beta ořezáváním se proto pro použití v GPU enginu nejeví jako vhodná.

5.3 MiniMax s iterativním prohlubováním

Rozšíření algoritmu MiniMax o iterativní prohlubování by mělo být bez problémů proveditelné i u verze běžící na GPU. Jako jednoduché řešení se jeví opakované spouštění kernelu na GPU s postupně se zvyšujícím parametrem udávajícím parametry.

Samotná implementace algoritmu na straně GPU by se tedy nijak nezměnila a opakované spouštění by mělo na starosti CPU. Nevýhodou tohoto řešení by byla režie související s opakovaným spouštěním kernelu.

Druhou možností by bylo implementovat řízení iterací na straně GPU. To by sice ušetřilo režii opakovaného spouštění, ale vedlo by to k nemožnosti výpočet v jakoukoliv chvíli zastavit. Kernel by musel již během spuštění dostat informaci, při splnění jakých podmínek má být prohledávání ukončeno. To je v rozporu s jedním z účelů iterativního prohlubování, kterým je právě možnost běh algoritmu kdykoliv po doběhnutí alespoň jeden iterace ukončit a přitom získat výsledek.

5.4 Alfa-beta s aspiračním prohledáváním

Aspirační prohledávání se jeví jako dobře použitelné v GPU enginu. Na GPU by bylo možné paralelně spustit několik bloků vláken provádějících aspirační prohledávání stejné pozice zároveň. Každému bloku by byly přiděleny odlišné hodnoty parametrů α a β tak, aby jimi definované intervaly po sjednocení pokrývaly celý interval $(-\infty, \infty)$.

V každém prohledávání v rámci jednoho bloku by tak docházelo k většímu ořezávání než při běhu běžného algoritmu alfa-beta bez aspiračního prohledávání. Na čím menší intervaly by byl rozsah možné evaluace rozdělen, tím výraznějšího oříznutí stavového prostoru by bylo dosaženo v každém bloku.

Ve většině bloků vláken by algoritmus rychle selhal a nenalezl tahy vedoucí k evaluaci specifikované zadaným intervalem. Právě jeden blok by však vrátil hodnotu evaluace spadající do jemu přidělenému intervalu. Tato evaluace by byla výslednou evaluací zadané pozice.

5.5 Scout

Paralelní verze algoritmu Scout funguje tak, že prohledání prvního uzlu v každé hloubce se provede sekvenčně a po dokončení prohledání tohoto uzlu se provede prohledání s nulovým oknem pro všechny zbývající uzly paralelně.

Pokud některé z paralelních prohledání sousedních uzlů vrátí hodnotu rovnou evaluaci prvního uzlu, provede se jeho kompletní prohledání opět sekvenčně a dále se postupuje stejně, jako by nalezený lepší uzel byl prvním uzlem.

Takový přístup se nejeví pro GPU implementaci jako vhodný. Dělení práce v rámci každého uzlu by vyžadovalo velmi složitou synchronizaci více bloků vláken s předáváním výsledků přes globální paměť. Z toho se dá usoudit, že algoritmus Scout není vhodný pro použití v šachovém GPU enginu.

5.6 Principal Variation Search

Algoritmus PVS (Principal Variation Search) je velmi podobný algoritmu Scout a podobné jsou i jejich paralelní verze. U PVS je paralelní verze založena na prohledávání hlavní variace sekvenčně a následného paralelního ověření ostatních možných tahů pomocí prohledání s nulovým oknem.

Z toho plynou stejné komplikace jako u algoritmu Scout. Navíc je potřeba mít ve sdílené paměti uloženou hlavní variaci z předchozí iterace. U GPU by to znamenalo další položky v globální paměti. Algoritmus PVS se tedy pro využití v šachovém enginu využívajícím GPU nejeví jako vhodný.

5.7 Závěr vyhodnocení možností využití různých algoritmů

Z analýzy algoritmů vyplynulo, že většina jich není vhodná pro použití v algoritmu běžícím na GPU. Hlavním důvodem je nutnost synchronizace na úrovni bloků vláken a nutnost využití globální paměti během výpočtu.

Výjimkou je paralelní aspirační prohledávání. U tohoto algoritmu by mohly bloky vláken pracovat zcela nezávisle, neboť mezi jednotlivými prohledáváními s disjunktními rozsahy hodnot α a β neexistují žádné závislosti. Tento algoritmus se proto jeví jako nejvhodnější pro použití v proof-of-concept šachové AI využívající GPU.

Kapitola 6

Návrh

6.1 Návrh využití pamětí GPU

Základem návrhu využití paměti u GPU enginu je zcela se vyhnout použití globální paměti, která je příliš pomalá. Globální paměť bude využita pouze pro načtení vstupních dat a uložení výstupních dat. Veškeré výpočty jako evaluace pozice a generování možných tahů budou probíhat v rychlé lokální paměti.

Velikost lokální paměti ale není dostatečná pro uložení všech v minulosti navštívených pozic v nižší hloubce, které algoritmus navštíví znovu.

Řešením je mít v paměti pouze aktuálně prohledávanou pozici a ukládat si posloupnost tahů vedoucí k této pozici [37]. Pomocí vracení tahů se můžeme z aktuální pozice vrátit až k té původní.

Dále musí lokální paměť obsahovat seznam tahů možných v aktuální pozici a hodnoty parametrů α a β pro každou v minulosti navštívenou pozici, do které se algoritmus ještě vrátí.

6.1.1 Návrh zakódování figur

Pro zakódování tahů a stavu šachovnice je nutné nejdříve nutné navrhnout zakódování samotných figur. Tyto kódy budou součástí jak tahů, kde je vždy potřeba ukládat figuru, která táhne, a případně figuru, která je sebrána a figuru, ve kterou se proměňuje pěšec.

Pro zakódování každé figury jsou potřeba 4 bity, neboť existuje šest typů figur (král, dáma, věž, střelec, jezdec, pěšec) a dále je třeba rozlišit barvu figury, což dává dvanáct typů figur a dále je potřeba jeden kód pro prázdné pole. Zakódování stavu celé šachovnice v dané pozici tedy zabere minimálně 256 bitů.

6.1.2 Návrh zakódování tahů

Pro návrat do předchozích pozic je nutné během celé doby běhu prohledávání udržovat v paměti sekvenci tahů vedoucí k aktuální pozici. Tahy musí obsahovat dostatek informací pro provedení i vrácení tahu.

K tomu je potřeba několik celých čísel reprezentujících kód figury, která táhne, index počátečního pole, ze kterého figura táhne, index cílového pole, kód figury na cílovém poli před provedením tahu (kvůli vrácení tahu) a kód figury na cílovém poli po provedení tahu (může se lišit od figury, která táhne v případě proměny pěšce).

Pro proof-of-concept GPU enginu nebudou zahrnuty tahy typu rošáda a braní en passant, které by vyžadovaly další informace uložené v tahu a vedly k růstu paměťové náročnosti. V případě, že proof-of-concept ukáže vhodnost využití GPU pro šachový engine, nebude problém přidat podporu pro tyto tahy.

Tabulka 6.1: Informace obsažené v tahu

Informace	Rozsah hodnot	Počet bitů
Kód figury na počátečním poli	<0, 13>	4
Kód figury na cílovém poli	<0, 13>	4
Kód figury na cílovém poli po tahu	<0;13>	4
Index počátečního pole	<0, 63>	6
Index cílového pole	<0, 63>	6

Kvůli paměťovým omezením GPU je nutné tyto tahy co nejúsporněji zakódovat. Jelikož je celkem na zakódování potřeba 24 bitů, bude pro zakódování potřeba alespoň 32 bitový datový typ. Výhodnější, než použít 32 bitový datový typ se ale jeví použít dva 16 bitové, neboť ne vždy bude potřeba všech pět složek tahu a dekódování z 16 bitů bude mírně jednodušší než z 32 bitů.

První část tahu tedy bude obsahovat kód figury na počátečním poli, index počátečního pole a index cílového pole. Bude postačovat pro provedení všech tahů kromě proměny pěšců. Druhá část bude obsahovat kód figury na cílovém poli před provedením tahu a kód figury na cílovém poli po provedení tahu. Bude potřeba pro proměny pěšců a vracení všech tahů.

Další možností by bylo obětovat část paměti a použít pro každou informaci vlastní 8 bitový datový typ. Znamenalo by to použít 40 bitů místo 32 bitů, ale ušetřit veškeré kódování a dekódování. Která možnost je výhodnější ukáže až testování.

6.2 Návrh reprezentace šachové pozice

Reprezentace šachové pozice je z hlediska návrhu velmi důležitá, neboť na ni závisí veškeré algoritmy, které šachová AI potřebuje. Ovlivní evaluaci pozice,

generování možných tahů a následujících pozic, aplikování tahů a rozhodování o legálnosti pozice.

6.2.1 Reprezentace orientovaná na figury

Reprezentace orientovaná na figury [38] se skládá z polí nebo seznamů všech figur, které jsou stále na šachovnici a polí, na kterých se nachází.

Jednou z reprezentací orientovaných na figury je bitboard. Bitboard využívá bitových vektorů o délce 64 pro každý typ figury. Většina moderních enginů používá bitboard, neboť na 64 bitových procesorech jsou operace se 64 bitovými datovými typy rychlejší a 64 bit verze enginů tak dosahují vyššího Elo hodnocení než 32 bitové.

Další reprezentací je množinová reprezentace, která používá jeden bitový vektor o délce 32, ze kterého je možno zjistit, které figury jsou stále na šachovnici. Pro zjištění jejich konkrétní pozice je potřeba čtení z odlišného pole.

6.2.2 Reprezentace orientovaná na pole šachovnice

Reprezentace orientovaná na pole šachovnice [38] si udržují pro každé pole šachovnice informaci, jaká figura se na něm nachází.

Výhodou této reprezentace je, že při zjišťování, jaká figura se nachází na poli, na které chceme táhnout jinou figurou, stačí pouze jeden přístup do paměti, místo procházení listů všech figur a hledání daného políčka.

Další výhodou je, že polí šachovnice je vždy stejný počet, na rozdíl od figur. To přináší u GPU implementace výhodu v tom, že je možné jednoduše staticky přidělit určitý počet vláken každému poli šachovnice.

Nevýhodou je nutnost při hledání určité figury procházet celou šachovnici. Jelikož se ale bude zjišťování možných tahů provádět pro všechny figury najednou, není u GPU implementace vyhledávání figur nutné.

Typická reprezentace z této kategorie využívající pole o velikosti 64 se nazývá Mailbox. Datová struktura 8x8 Board využívá místo pole matici. Tyto reprezentace mají problémy s efektivitou generování tahů kvůli nutnosti testovat překročení hranice šachovnice.

Reprezentace 10x12 Board je vylepšením 8x8 Board. Obsahuje v sobě 8x8 Board a přidává dva sloupce a řádky na krajích nesoucí informaci, že se nacházejí mimo šachovnici. Díky tomu se sice ušetří operace testující, zda se pole nenachází mimo šachovnici, ale zase je provedeno o jeden přístup do paměti více, což je u GPU implementace nežádoucí.

Vektorové reprezentace využívají dvousložkových vektorů k adresaci polí i reprezentaci tahů. U polí znamenají hodnoty složek vektoru počet řádků a sloupců od zvoleného rohového pole šachovnice a u tahů vzdálenost mezi výchozím a cílovým řádkem a mezi výchozím a cílovým sloupcem. Z vektorové reprezentace se dá lehce zjistit, která figura útočí na které pole.

6.2.3 Volba pro GPU engine

Jako ideální pro GPU engine se jeví jednoduchý Mailbox s polem o délce 64. Zabírá nejméně místa v paměti a tato velikost se nemění. Nutnost testovat překročení hranice šachovnice je přijatelnou cenou za nižší paměťovou náročnost a menší počet přístupů do paměti než u reprezentace 10x12 Board.

Při indexování od levého horního rohu šachovnice, tedy pole a8, vycházejí tyto hodnoty, které je třeba přičíst k indexu daného pole, aby byl výsledkem index sousedního pole.

Tabulka 6.2: Výpočet indexů sousedních polí

Lokace sousedního pole	Hodnota přičtená k indexu
Sever	-8
Severovýchod	-7
Severozápad	-9
Východ	1
Západ	-1
Jih	8
Jihovýchod	9
Jihozápad	7

Složitější je výpočet indexů pro tahy jezdcem, kdy se mění index sloupce o dva a řádku o jedna nebo sloupce o jedna a řádku o dva. Pro zjištění lze použít tuto tabulku, reprezentující část šachovnice. Středové pole s hodnotou 0 značí výchozí pole.

Tabulka 6.3: Rozdíly v indexech sousedních polí

-18	-17	-16	-15	-14
-10	-9	-8	-7	-6
-2	-1	0	+1	+2
+6	+7	+8	+9	+10
+14	+15	+16	17	18

6.3 Návrh reprezentace figur na šachovnici

Návrh reprezentace figur v poli reprezentujícím šachovnici silně ovlivní algoritmus, který k tomuto poli neustále bude přistupovat. Je vhodné kódy pro figury zvolit tak, aby nesla co nejvíce informací a umožnila tak co nejjednodušší identifikaci figur.

Proto byla zvolena reprezentace, ve které kódy černých figur vzniknou přičtením čísla 7 ke kódům odpovídajících bílých figur. Díky tomu je možné

použít jednoduchého porovnání s jedním číslem pro zjištění barvy figury a dělení modulo 7 pro zjištění typu figury.

Zároveň zbytek po dělení 7 určuje hodnotu figury. Čím je zbytek vyšší tím, je figura cennější. Jako nejcennější figura je brán král. Prázdné pole šachovnice je reprezentováno hodnotou 0.

Tabulka 6.4: Kódy figur

Figura	Kód
Žádná	0
Bílý pěšec	1
Bílý jezdec	2
Bílý střelec	3
Bílá věž	4
Bílá dáma	5
Bílý král	6
Černý pěšec	8
Černý jezdec	9
Černý střelec	10
Černá věž	11
Černá dáma	12
Černý král	13

6.4 Návrh hodnot figur pro evaluaci

Způsob přidělení hodnot figurám významně ovlivní evaluaci pozice a řazení tahů. Je možné přidělit hodnotu i králi. Musí být zvolena hodnota taková, aby byla vyšší než teoreticky nejvyšší součet hodnot všech ostatních figur na šachovnici po proměně všech pěšců na figuru s nejvyšší hodnotou.

Pak je možné podle evaluace poznat, zda je král stále na šachovnici nebo byl sebrán, což znamená mat v předchozím tahu. Následující návrh ohodnocení figur, který vychází z tradičního ohodnocení figur používaného od 18. století, splňuje tuto podmínku.

Tabulka 6.5: Ohodnocení figur

Figura	Ohodnocení
Pěšec	1
Jezdec	3
Střelec	3
Věž	5
Dáma	9
Král	200

Existuje velké množství systémů určujících hodnotu figur. Většina z nich je drobnou variací standardního. Střelci je většinou přisuzována vyšší hodnota než jezdci a další bonus získává hráč stále disponující dvojicí střelců. Některé systémy přisuzují vyšší hodnotu věži, některé naopak nižší. To samé platí pro dámu.

Systémy také mohou figurám přiřazovat odlišné hodnoty v závislosti na fázi hry. V zahájení a střední hře jsou upřednostňovány lehké figury (střelci a jezdci), v koncové hře se zvyšuje hodnota pěšců a těžkých figur (věže a dámy).

Obecně přijímaným faktem je, že dvě lehké figury jsou silnější než věž a dvě věže nebo tři lehké figury jsou silnější než dáma. Z každého pravidla ale existují výjimky a mnoho sehraných partií to dokazuje.

Jako první systém hodnot figur byl tedy zvolen ten standardní a na základě testování může být lehce upraven podle potřeby.

6.5 Návrh řazení tahů

Řazení tahů je pro dobré fungování šachového enginu velmi podstatné. Provádějí ho všechny analyzované šachové enginy. Prohledávání těch správných tahů jako prvních vede k získání takových hodnot α a β, které vedou k ořezání větší části stavového prostoru. V důsledku to může znamenat prohledání řádově méně uzlů než při zvolení jiného pořadí tahů při prohledávání.

Nejvyšší roli seřazení hraje v nejnižší hloubce stromu, neboť učiněná rozhodnutí ovlivní všechny podstromy. Proto musí být zejména tahy možné v iniciální pozici, pro kterou provádíme evaluaci, dobře analyzovány a podle toho seřazeny.

Kandidáty pro přední pozice v pořadí prohledávání jsou tahy šachující soupeřova krále, tahy beroucí soupeřovu figuru, tahy napadající soupeřovu figuru a proměny pěšců. Ideálními kandidáty jsou tahy spadající do více z těchto kategorií.

6.5.1 Tahy šachující soupeřova krále

Tahy šachující soupeřova krále jsou hlavními kandidáty na brzké prohledání. Série šachů mohou vést k matu, jehož nalezení vede k velmi rychlému ukončení prohledávání. Na šach mívá soupeř mnohem menší počet možných odpovědí, neboť musí buď sebrat šachující figuru nebo táhnout králem.

Pokud nedetekujeme mat jako situaci, kdy je král v šachu a nemůže z něho uhnout, ale necháváme prohledávání běžet až do sebrání krále v následujícím tahu, pak každý šach může být zároveň matem a vyplatí se to zkontrolovat před prohledáváním ostatních tahů.

Tahy dávající šach také mohou také zároveň napadat jinou soupeřovu figuru a dalším tahem ji vyhrát. I kvůli tomuto potenciálnímu materiálnímu zisku se vyplatí tahy šachující soupeřova krále prohledávat nejdříve.

6.5.2 Tahy beroucí soupeřovu figuru

Tahy beroucí soupeřovu figuru mohou vést k materiálnímu zisku a proto je vhodné je prohledávat dříve než tahy, které nic neberou, s výjimkou šachů. Tyto tahy navíc zaručují udržení iniciativy, neboť většinou donutí soupeře reagovat na takový tah.

6.5.3 Tahy napadající soupeřovu figuru

Stejně jako tahy beroucí figuru nebo šachující krále, udržují tahy napadající soupeřovu figuru iniciativu, zejména pokud je soupeřova figura nekrytá nebo pokud má napadající figura nižší hodnotu než figura napadená. Ideální je, pokud tah napadá více soupeřových figur najednou nebo je zároveň šachem.

6.5.4 Proměny pěšců

Tah pěšcem na poslední řadu znamená materiální zisk a proto by měl být prohledán jako jeden z prvních. Jelikož proměna probíhá na území soupeře a nejčastěji se pěšec mění v dámu, která napadá velký počet polí, je proměna pěšce často také tahem napadajícím soupeřovy figury a dávajícím šach.

6.5.5 Vzorec pro určení priority tahu

Každému tahu je nutné přidělit prioritu, podle které budou seřazeny. Výpočet priority musí být jednoduchý, aby příliš nezpomaloval generování tahů, ale zároveň správně přiřazovat vyšší prioritu tahům ze zmíněných kategorií.

Vzorec pro výpočet priority tahu, kde P je priorita tahu, H_s hodnota sebrané figury, H_n součet hodnot napadených figur (včetně krále) a H_p hodnota figury, ve kterou se proměnil pěšec, byl navržen takto:

$$P = 2H_s + H_n + H_p$$

Vytvořený vzorec dává nejvyšší prioritu matu díky reprezentaci matu jako sebrání krále a vynásobení hodnoty krále dvěma. Následují šachy, které mají vysokou prioritu díky napadení krále, který má vysokou hodnotu. Poté následují proměny figur a napadání soupeřových figur.

6.6 Návrh využití GPU pro šachovou AI

Na základě rešerše šachových enginů byl pro GPU engine zvolen jako základní algoritmus algoritmus MiniMax s alfa-beta ořezáváním, který používají všechny přední šachové enginy. Tato sekce se bude zabývat návrhem jeho GPU implementace.

Pro použití na GPU je třeba navrhnout maximální možnou paralelizaci tohoto algoritmu. Paralelizace spočívající v rozdělení stavového prostoru mezi

jednotlivá vlákna, která by se dala použít u vícejádrového CPU by byla u GPU z výkonového hlediska nepoužitelná.

Vedla by k využití pouze nízkého počtu vláken a obrovskému počtu operací, které by muselo každé vlákno provést. Místo toho je třeba využít schopnost GPU provádět velké množství vláken současně. Jednou z možností je použití více vláken pro evaluaci pozice, generování možných tahů a výběru tahu k prohledání.

6.6.1 Paralelní evaluace šachové pozice

Zatímco sériový algoritmus prochází pole šachovnice postupně, paralelní algoritmus na GPU může vytvořit 64 vláken, z nichž každé bude mít na starosti jedno pole šachovnice.

V případě jednoduché evaluační funkce založené na rozdílu součtu materiální hodnoty figur bílého a černého hráče by paralelizace evaluace mohla vypadat takto:

1. Každé vlákno zjistí, zda na jemu přiděleném poli stojí figura.
2. Pokud ano, vloží do pomocného pole na index odpovídající poli šachovnice kladnou hodnotu figury v případě bílé figury a zápornou v případě černé figury.
3. Jakmile je dokončen předchozí bod, vlákna provedou paralelní binární redukci [39] nad pomocným polem.
4. Po dokončení předchozího bodu je evaluace pozice uložena na prvním indexu pomocného pole.

U složitějších evaluačních funkcí, beroucích v potaz i faktory jako struktura pěšců nebo aktivita figur, může být výhodné pro evaluaci pozice využít i více než 64 vláken.

Zvýšení počtu vláken bude mít nižší dopad na výkon než situace, kdy jedno vlákno dostane více práce než ostatní a ta na něj budou muset čekat. Například získání počtu tahů, které může provést dáma za účelem zhodnocení její aktivity, si vyžádá několikanásobně vyšší počet operací než zjištění, zda je možno táhnout pěšcem.

6.6.2 Paralelní generování možných tahů

Generování tahů je nejnáročnější operací, kterou šachový engine provádí. Zatímco výsledkem evaluace je jediné číslo, výsledkem generování jsou většinou desítky tahů, z nichž každý je reprezentovaný několika čísly.

To znamená mnohem více zápisů do paměti, což je u GPU nežádoucí. Dalším problémem je, že neexistuje pevná vazba mezi poli šachovnice a počtem

možných tahů, které mají toto pole jako výchozí. Z většiny polí není možné provést žádné tahy, neboť se na nich nenachází figura hráče, který je na tahu.

Z jiných polí, na kterých stojí například dáma, je naopak možno provést vysoké množství tahů. Aby se dalo určit, na jaký index v poli generovaný tah uložit, je nejprve třeba spočítat počty možných tahů pro každé pole a nad výsledným polem provést paralelní prefixový součet [39].

Tím získáme pole s indexy pro uložení prvního tahu pro každé pole šachovnice. Po uložení každého tahu snížíme index pro dané pole šachovnice o jedna. S pomocí tohoto pole už je následně možné paralelně generovat tahy a ukládat je do výsledného pole bez mezer a konfliktů. Celý postup by vypadal takto:

1. Každé vlákno zjistí, zda na jemu přiděleném poli stojí figura hráče, který je na tahu.
2. Pokud ano, zjistí, kolik tahů může tato figura provést a zapíše výsledek do pomocného pole.
3. Jakmile je dokončen předchozí bod, vlákna provedou paralelní prefixový součet nad pomocným polem.
4. Po dokončení předchozího bodu je v pomocném poli na každém indexu uložen index v poli pro generované tahy zvýšený o jedna, na který je třeba uložit poslední vygenerovaný tah z daného pole šachovnice.
5. Každé vlákno, na jemuž přiděleném poli stojí figura hráče, který je na tahu generuje všechny tahy proveditelné touto figurou a ukládá je na index daný pomocným polem, před každým uložením snížený o 1.
6. Po dokončení předchozího bodu jsou v poli pro generované tahy uloženy veškeré tahy proveditelné v dané pozici.

Tento postup bude generovat tahy vždy ve stejném pořadí. To je nutností, neboť na GPU nemáme dostatek lokální paměti pro uložení možných tahů pro všechny předchozí pozice, do kterých se můžeme vrátit. Místo toho musíme uložit pouze index tahu, který má být prohledán jako další a při návratu do uzlu tahy vygenerovat znovu ve stejném pořadí, aby index ukazoval na správný tah.

Samotné seřazení tahů podle priority na vyřešení tohoto problému nestačí, neboť tahy mohou mít shodnou prioritu. Pokud by takové tahy byly při každém generování v odlišném pořadí, byly by pak v odlišném pořadí i po seřazení.

6.6.3 Paralelní výběr dalšího tahu k prohledání

Z vygenerovaných tahů je v každém kroku potřeba vybrat, který z nich bude prohledán jako další. Již během generování je tahům přidělena priorita, která určí pořadí jejich prohledávání. Podle té musíme tahy paralelně seřadit.

Jelikož tahů budou jen desítky, na seřazení jejich priorit postačí jednoduchý paralelní stabilní in-place selection sort [40]. Tahů ale může být více než 64, takže nepostačí vlákna používaná pro evaluaci a generování tahů, ale bude jich třeba vytvořit více.

Byl zvolen počet 128. Ten postačí i v případě teoretických pozic, kdy jeden z hráčů promění veškeré své pěšce na dámy a následně je společně s ostatními figurami perfektně rozmístí bez toho, aby se soupeřův král dostal do patu nebo matu.

Řazení tahů probíhá tak, že si každé vlákno nejdříve uloží prioritu na svém indexu do privátní proměnné a průchodem pole zjistí, kolik je v poli vyšších priorit a shodných priorit na nižších indexech.

Jakmile toto všechna vlákna dokončí, zapíšou uloženou prioritu na index rovný zjištěnému počtu vyšších priorit a shodných priorit na nižších indexech. V tu chvíli je pole priorit seřazeno od nejvyšších k nejnižším, tedy v pořadí, v jakém chceme tahy procházet.

Aby nebylo potřeba řadit samotné tahy, které mají několik složek, je přidáno další pole, které mapuje indexy priorit na indexy tahů. Hodnoty v tomto poli jsou aktualizovány každým vláknem během řazení. Na index, na který vlákno zapisuje prioritu, zapíše do mapovacího pole své lokální id vlákna, které je rovno indexu tahu.

6.6.4 Kombinace lokální a globální paralelizace

Vytvořením 128 vláken by ale zdaleka nebylo využito možností GPU. Proto je nutné pararelizaci na úrovni evaluace pozice, generování tahů a řazení tahů zkombinovat s paralelizací na úrovni bloků vláken. Analytická část práce ukázala, že nejlépe by se toho dalo dosáhnout pomocí využití paralelního aspiračního prohledávání.

Každému bloku vláken(v OpenCL "work-group", v CUDA "thread block") by byly přiděleny vlastní intervaly dané hodnotami parametrů α a β tak, aby tyto intervaly po sjednocení pokrývaly celý interval $(-\infty, \infty)$.

Místo intervalu $(-\infty, \infty)$ by také bylo možné použít interval okolo odhadované evaluace a ten rozdělit na několik menších intervalů pro každý blok. Jednalo by se tedy o aspirační prohledávání v rámci aspiračního prohledávání.

Pokud by první blok vrátil evaluaci rovnou α nebo poslední blok vrátil evaluaci rovnou β, muselo by se prohledávání opakovat. Pro ostatní bloky by bylo navrácení evaluace rovné α nebo β ignorováno. Výslednou evaluací by byla evaluace nerovnající-se ani α ani β vrácená jedním z bloků.

Kapitola 7

Implementace

7.1 Implementace GPU prototypu

Před implementací samotného enginu byl implementován prototyp, který provede paralelní statickou evaluaci šachové pozice na GPU. Cílem tohoto prototypu bylo ověřit návrh paralelní evaluace šachové pozice, kdy má každé vlákno přiděleno jedno pole šachovnice.

7.1.1 Volba technologií

Pro maximální jednoduchost implementace byla pro tento prototyp zvolena knihovna CUDAfy.NET. [41] Tato knihovna umožňuje provádění GPGPU výpočtů z prostředí frameworku .NET. [42] Výhodou této knihovny je, že umožňuje jednoduché využívání GPU přímo z objektově orientovaného programovacího jazyku C# nebo VB.NET. Další výhodou je podpora CUDA i OpenCL, knihovna tedy může využívat téměř každou moderní grafickou kartu.

Nevýhodou této knihovny je nízká možnost provádění optimalizací a s tím související nižší výkon oproti přímé implementaci v programovacím jazyce C++ nebo C. Ukázala se tedy nevhodnou pro implementaci celého GPU enginu, ale dostatečnou pro implementaci prvotního prototypu.

První prototyp byl tedy napsán v programovacím jazyce C# za použití IDE Visual Studio 2013 [43] a využíval knihovnu CUDAfy.NET. Z důvodu neuspokojivého výkonu při vytvoření velkého počtu vláken (viz tabulka 8.1 v kapitole o testování) byl následně prototyp přepsán do programovacího jazyku Java za použití IDE NetBeans 8.0.1 [44] a byla využita knihovna Aparapi [45] od AMD.

Aparapi převádí Java bytecode do OpenCL během běhu programu. Je ještě jednodušší na použití než CUDAFy.NET ale má více omezení a nepodporuje mnoho konstruktů jazyku Java, včetně tak základních jako je *switch*. Měření (viz tabulka 8.1 v kapitole o testování) ukázalo, že oproti CUDAFy.NET dosahuje vyššího výkonu při vytvoření velkého počtu vláken, ale kvůli své jed-

noduchosti taktéž není vhodná pro implementaci GPU enginu, ale postačuje pro implementaci prototypu.

7.1.2 Funkce prototypu

Jedinou funkcí prototypu je provést paralelní statickou evaluaci šachové pozice uložené v globální paměti GPU. Pozice je vytvořena na straně CPU, uložena do pole datového typu *int* a zkopírována do globální paměti GPU.

Poté je vytvořeno 64 GPU vláken, která provedou evaluaci pozice. Algoritmus paralelní evaluace šachové pozice pomocí 64 vláken je popsán v sekci 6.6.1. Následně je výsledek zkopírován zpět do paměti CPU.

7.1.3 Rozšíření prototypu

Následně byl prototyp rozšířen, aby dokázal evaluovat více pozic zároveň. Pole, ve kterém je pozice uložena bylo jednoduše rozšířeno, aby obsahovalo více pozic. Každá pozice zabírá 64 prvků pole a pozice jsou uloženy sekvenčně. Pomocí dělení modulo 64 lze zjistit index daného pole šachovnice a pomocí dělení číslem 64 číslo pozice, ke které prvek pole patří.

Stejně tak bylo rozšířeno i pomocné pole pro mezivýsledky. Binární redukce se pak neprovádí nad celým polem ale pouze nad úseky pole o délce 64 reprezentujícími jednu pozici. Výsledky se ukládají do pole výsledků, jehož délka je rovna počtu pozic. Toto pole je pak zkopírováno zpět do paměti CPU.

7.1.4 Vyhodnocení implementace prototypu

Implementace prototypu ukázala, že statická paralelní evaluace šachové pozice na GPU není problém. Vedla také k závěru, že není možné vytvořit použitelný engine, který bude kombinovat využití CPU a GPU.

Časově nejnáročnější operací je totiž zkopírování šachových pozic z paměti CPU do paměti GPU a zpět a inicializace výpočtu na GPU. Dobře je to vidět z doby běhu rozšířeného prototypu napsaného v jazyce Java využívajícího knihovny Aparapi v závislosti na počtu evaluovaných pozic v tabulce 8.2 v kapitole o testování.

Z této tabulky srovnávající rychlost evaluace pozic na CPU a GPU v závislosti na počtu evaluovaných pozic zároveň plyne, že v praxi není použitelné ponechat herní strom v paměti CPU a na GPU pouze provádět paralelní evaluaci jednotlivých pozic.

Díky sekvenční charakteristice algoritmu MiniMax s alfa-beta ořezáváním by bylo reálné posílat k evaluaci maximálně desítky pozic zároveň - to v případě, že by se všechny možné následující pozice generovaly ihned při návštěvě uzlu a ne samostatně až při zvolení tahu k prohledání.

Evaluace desítek uzlů na GPU ale trvá mnohem déle (při započítání režie kopírování a spuštění výpočtu) než na CPU a algoritmus by byl v praxi nepo-

užitelný. Zrychlení oproti použití samotného CPU by přinesla až evaluace stovek tisíc pozic zároveň. Takový počet pozic vyžadujících evaluaci algoritmus s alfa-beta ořezáváním ale nikdy nevytvoří, neboť potřebuje podle evaluace předchozích uzlů ořezávat stavový prostor.

7.2 Implementace CPU enginu

Před implementací enginu pro GPU byl implementován engine pro CPU simulující budoucí GPU engine. Hlavními důvody pro toto rozhodnutí bylo mnohem jednodušší testování a hledání chyb u programu běžícího na CPU. Zároveň bude tento engine sloužit k porovnání s enginem pro GPU.

7.2.1 Volba technologií

Pro implementaci CPU enginu byl zvolen programovací jazyk Java z důvodů vyšších zkušeností autora v programování v tomto jazyku než v jazycích C++ a C a z důvodu jednoduššího testování. Při ohledu zpět se to ukázalo jako chyba, neboť engine byl díky tomu příliš odlišný od následně vytvářené GPU verze a pro GPU engine se daly použít jen některé části kódu CPU enginu. Správnou volbou byla implementace CPU enginu v jazyku C++ nebo C. Jako IDE bylo použito NetBeans 8.0.1.

7.2.2 Postup implementace

Jako první byly implementovány třídy reprezentující pozici šachové partie, tah a přechod mezi pozicemi, a pomocné třídy pro generování tahů, výpočet priority tahu, evaluaci pozice a převod mezi pozicí a FEN notací.

Pro všechny důležité metody těchto tříd byly napsány důkladné Unit testy pro eliminování chyb. Teprve po dokončení těchto tříd došlo k implementaci samotného enginu, nebo spíše jeho variant, počínaje primitivními a postupně rozšiřovanými o nové funkce.

7.2.2.1 MiniMax engine

Jako první byl implementován primitivní MiniMax engine bez jakýchkoliv vylepšení. Jako evaluační funkce byl použit jednoduchý rozdíl součtu hodnot figur obou hráčů. Díky nulovému ořezávání stavového prostoru tento engine dokázal prohledávat pouze do hloubky čtyř tahů a tedy provést dva tahy bílého a dva tahy černého hráče.

To u většiny šachových pozic znamená prohledání jednoho až dvou milionů uzlů. Hloubka pěti tahů už by znamenala prohledání desítek milionů uzlů a tedy neúnosně dlouhou dobu běhu.

Díky těmto limitacím byl takový engine schopen nalézt pouze mat jedním tahem nebo sebrání nejcennější soupeřovy figury.

7.2.2.2 Alfa-beta engine

Následně byl implementován jednoduchý alfa-beta engine s jediným vylepšením v podobě přidaného ořezávání v případě nalezení matu. Základem pro tento engine byl pseudoalgoritmus z kapitoly č. 2.

Jelikož nebylo implementováno řazení tahů podle priority, nedokázal tento engine včas nalézt tahy vedoucí k hodnotám alfa a beta, které by výrazně ořízly stavový prostor.

V závislosti na evaluované pozici dokázal tento engine prohledat stavový prostor do větší hloubky než MiniMax engine a nalézt tahy vedoucí k matu nebo zisku materiálu, ale pokud takové tahy včas nenalezl, nebyl použitelný.

7.2.2.3 Alfa-beta engine s řazením tahů

Poté byl implementován engine rozšiřující předchozí engine o řazení tahů k prohledávání podle priority. Priorita každého tahu byla vypočítána podle vzorce uvedeného v kapitole č. 5. Při každé návštěvě uzlu byl jako další tah k prohledání zvolen ten s nejvyšší prioritou z těch, které ještě nebyly prohledány.

Přidání řazení tahů velmi zvýšilo schopnosti alfa-beta enginu. Vedlo k výraznému ořezávání stavového prostoru a prohledávání pouze zlomku pozic v porovnání s enginem bez řazení. Díky tomu byl engine schopný řešit šachové úlohy spočívající v nalezení matu druhým nebo třetím tahem.

7.2.2.4 Engine s iterativním prohlubováním

Jako další byl implementován jednoduchý engine s iterativním prohlubováním. Jeho úkolem bylo odstranit slabinu předchozího enginu, který v případě zadání prohledávání do větší hloubky, než bylo potřeba pro nalezení sekvence tahů vedoucí k matu, zbytečně prohledával některé uzly.

V případě iterativního prohlubování, kdy se prohledávání spouští opakovaně s postupně se zvětšující hloubkou je zaručeno, že engine nebude stavový prostor prohledávat do větší hloubky, než je potřeba k nalezení matu.

Zároveň byl engine přepsán z rekurzivního na iterativní, což bylo nutné pro jeho následné přepsání do verze pro GPU, neboť OpenCL rekurzi nepodporuje (CUDA v novějších verzích ano).

7.2.2.5 Engine s iterativním prohlubováním a rozšířeným prohledáváním šachů

Engine s iterativním prohlubováním dosahoval v pozicích, ve kterých existuje sekvence tahů vedoucí k matu, horších výsledků než engine bez tohoto vylepšení, který měl nastaveno vyhledávání přesně do hloubky, ve které je rozhodnuto o matu.

Engine s iterativním prohledáváním tak mohl například procházet strom o hloubce 4 a přitom navštívit více uzlů než ve stromu o hloubce 5, ve kterém byl ale rychle nalezen mat.

Bylo proto implementováno vylepšení, kdy se tahy, které jsou šachy pro hráče, který je na tahu v iniciální pozici, prohledávají do větší hloubky než ostatní tahy.

Pro druhého hráče se jako odpověď na tyto šachy prohledávají všechny tahy. Rozšířené vyhledávání se zastaví pokud je dosaženo stanoveného limitu přes hloubku prohledávání nebo hráči, který je na tahu v původní pozici dojdou šachy.

Přínos tohoto vylepšení byl velmi výrazný. Vedl ke schopnosti řešit úlohy spočívající v nalezení matu čtvrtým tahem, což žádný předchozí engine nedokázal. Vliv na rychlost nalezení matů druhým a třetím tahem je zobrazen v tabulce 8.3 v kapitole testování.

7.3 Implementace proof-of-concept GPU enginu

Jako poslední byl implementován samotný proof-of-concept GPU enginu, jeden z cílů této práce. Při jeho implementaci bylo využito poznatků z implementace a testování prototypu GPU enginu a CPU enginu.

7.3.1 Volba technologií

Z důvodu omezenosti technologie CUDA pouze na grafické karty nVidia bylo zvoleno implementovat GPU engine za použití OpenCL. Dále byl použit vývojový kit AMD APP SDK [46], který poskytuje šablony pro OpenCL programy. Z těch lze převzít kód pro inicializaci prostředí OpenCL a volání kernelu a soustředit se na kód běžící na GPU.

Jako vývojové prostředí bylo použito Visual Studio 2013. Toto IDE je ve verzi Community dostupné zdarma.

7.3.2 Zásady implementace na GPU

Během celé implementace bylo dodržováno několik kritických zásad pro programování na GPU.

7.3.2.1 Zápis do lokální a globální paměti

Při zápisu do lokální paměti je nutné, aby každé vlákno zapisovalo na odlišnou adresu paměti. Při zápisu do pole je nutné přesně spočítat indexy pro každé vlákno, aby se předešlo konfliktům. U polí kratších, než je počet vláken, je potřeba přidat podmínku, aby zapisovala pouze vlákna s indexy menšími, než je délka pole.

Pokud se na takovou podmínku zapomene, vlákna mohou zapisovat do pole, které je v paměti umístěné za polem, do kterého chceme zapisovat, a odhalení takové chyby je pak velmi náročné.

Pokud chceme zapsat pouze jednu hodnotu, je třeba vyhradit vlákno, které to provede. Bylo stanoveno pravidlo, že to bude vždy vlákno s lokálním indexem rovným nule.

Před zápisem do lokální nebo globální paměti je potřeba zajistit, že přepisované hodnoty už byly přečteny. Stejně tak při následném čtení je nutné zajistit, že hodnota již byla zapsána. Obojího je dosaženo použitím lokálních bariér, které synchronizují všechna lokální vlákna v daném bodě kernelu.

Většinou je nutné umístit jednu bariéru před část kódu se zápisem do lokální nebo globální paměti a druhou za tuto část kódu. Výjimkou jsou situace, kdy z dané paměti ještě nebylo čteno nebo již nebude čteno.

7.3.2.2 Umístění bariér

Důležitým pravidlem je umísťování bariér pouze na taková místa v kódu, která budou vždy dosažena všemi vlákny. V případě nutnosti provádět synchronizaci pouze části vláken, například při provádění prefixového součtu nad polem o délce 64, když celkový počet lokálních vláken je vyšší, nelze tento kód umístit do podmínky vyřazující vlákna s vyšším indexem.

To by vedlo k uváznutí, neboť bariéra by nebyla prolomena. Místo toho je nutné vlákna s indexem zahrnout v synchronizovaném kódu a teprve uvnitř tohoto kódu pomocí podmínky zajistit, že nebudou provádět žádnou práci.

7.3.3 Omezení použití paměti

Během implementace proof-of-concept GPU enginu byla vyvinuta maximální snaha o minimalizaci velikosti vyžité lokální a privátní paměti. To zahrnovalo používání datových typů zabírajících co nejméně místa, používání co nejméně polí a upřednostňování provádění více, klidně opakovaných, instrukcí před uchováváním mezivýsledků v paměti. V paměti by se mělo nacházet jen to, co je nutně potřeba k běhu enginu.

Důvodem je limitace velikostí lokální paměti. To, kolik lokální paměti spotřebuje jeden blok vláken, rozhodne o tom, kolik bloků vláken bude možno spustit současně.

7.3.4 Začátek implementace

Jako první byl v OpenCL a implementován prototyp provádějící evaluaci šachové pozice v nulové hloubce pomocí 64 vláken. Výsledky pro různé pozice byly porovnány s důkladně otestovaným CPU enginem.

Následovala implementace paralelního výpočtu počtu možných tahů z iniciální pozice pro každé pole šachovnice a paralelního prefixového součtu nad

polem těchto počtů. Výsledný součet počtů tahů byl porovnán s počtem tahů vygenerovaných pro danou pozici CPU enginem.

Dále byl implementován paralelní generátor možných tahů využívající indexy vypočítané prefixovým součtem v předchozím kroku pro zjištění pozice, na kterou uložit tah. Bylo ověřeno, že na každém indexu pole pro vygenerované tahy menším než počet možných tahů se po doběhnutí nachází tah.

Následně byly vygenerované tahy pro různé pozice porovnávány s tahy vygenerovanými CPU enginem. Tímto byla ověřena správnost této nejdůležitější části enginu.

Dalším krokem byla implementace paralelního řazení tahů podle priority. Správnost byla opět potvrzena srovnáním s pořadím tahů u CPU enginu.

Poté proběhla implementace provedení a vrácení tahu. Pro testovací účely byl prováděn tah s nejvyšší prioritou a následně ihned vracen. Poté bylo ověřeno, že výsledná pozice je shodná s tou původní.

7.3.5 Prvotní verze

Ze všech předchozích součástí byl následně vytvořen jednoduchý alfa-beta engine s jedním blokem vláken. Bylo třeba přidat řídící proměnné, které jsou uložené v privátní paměti. Ukládání těchto hodnot do lokální paměti by znamenalo přístup mnoha vláken k jednomu místu v paměti a tedy negativní výkonový dopad.

Každé vlákno má tedy například vlastní kopii informace o tom, v jaké hloubce se engine právě nachází nebo jaké jsou hodnoty parametrů α a β a samo si udržuje jejich hodnoty aktuální.

Vlákno s lokálním indexem nula zapisuje hodnoty parametrů α a β do pole udržujícího tyto hodnoty pro sekvenci pozicí vedoucí k aktuální pozici. Jednotlivá vlákna si pak tyto hodnoty načítají z pole ve chvíli přechodu do dané pozice.

Dále vlákno s lokálním indexem nula provádí a vrací tahy a aktualizuje index tahu, který bude prohledán jako příští. Jeho posledním úkolem je zápis výsledné evaluace a nejlepšího nalezeného tahu do globální paměti na konci výpočtu.

Výkon této prvotní verze byl srovnán se srovnatelným enginem (shodný alfa-beta algoritmus, shodná logika generování a řazení tahů, shodná evaluační funkce) běžícím na CPU. Aby se vyloučil negativní vliv implementace v jazyku Java na výkon CPU enginu, byl pro tento účel implementován engine v jazyku C++. Výsledky měření jsou v tabulce 8.4 v kapitole o testování.

7.3.6 Vylepšená verze s více bloky vláken

Po zjištění, že na dostupné grafické kartě je možné bez dopadu na výkon provádět jen 4 bloky vláken GPU enginu (viz tabulka 8.5 v kapitole o testování),

bylo rozhodnuto, že jako proof-of-concept bude implementován GPU engine s paralelním aspiračním prohledáváním o 4 intervalech.

Rozdělení celého intervalu možné evaluace $(-\infty, \infty)$ na pouhé 4 intervaly sice ve většině případů velký výkonový přínos nepřinese, neboť intervaly jsou příliš velké, ale paralelní aspirační prohledávání v rámci aspiračního prohledávání s menším intervalem už by mělo výpočet urychlit.

Toto bylo ověřeno měřením doby běhu v závislosti na hodnotách intervalů. Výsledky jsou v tabulce 8.6 v kapitole o testování.

7.3.6.1 Vstupní a výstupní parametry kernelu

Oproti prvotní verzi byl výstup kernelu enginu rozšířen z jedné hodnoty evaluace na 4 a z jednoho "nejlepšího" tahu na 4 (pouze jeden blok vrátí opravdu nejlepší tah). Vstup byl rozšířen o hodnoty parametrů α a β. Výsledek vypadal takto:

Tabulka 7.1: Vstupní parametry kernelu

Popis	Datový typ	Délka pole
Evaluovaná pozice	char	64
Hráč na tahu	char	-
Hloubka prohledávání	char	-
Alfa hodnoty pro bloky	float	4
Beta hodnoty pro bloky	float	4

Tabulka 7.2: Výstupní parametry kernelu

Popis	Datový typ	Délka pole
Evaluace pro každý blok	float	4
Nejlepší tah pro každý blok	int	4
Debug	float	128

Nejlepší tah pro každý blok je zakódován podle návrhu v páté kapitole a obsahuje figuru, výchozí a cílové pole. Druhá část tahu obsahující figuru na cílovém poli před a po tahu není ve výstupu potřeba. Pole pro debugovací účely obsahuje hodnoty jako počet prohledaných pozic pro jednotlivé bloky a další informace o běhu kernelu.

7.3.6.2 Volba intervalů pro paralelní aspirační prohledávání

Možnost rozdělit interval možné evaluace pouze na 4 podintervaly je velmi limitující. Jelikož se nejčastěji evaluace šachové pozice pohybuje okolo hodnoty 0, značící že je hra vyrovnaná, bylo by výhodné mít co nejvíce co nejkratších intervalů v okolí této hodnoty.

Pokud bychom intervaly neprokryli veškeré možné evaluace a v případě evaluace mimo intervaly spouštěli prohledávání znovu, šlo by použít například tyto 4 intervaly: $(-2, -1.01)$, $(-1, -0.01)$, $(0, 0.99)$, $(1, 2)$. Tento návrh počítá s evaluační funkcí s granularitou evaluace vyšší než 0,1. To použitá evaluace rovna rozdílu sumy hodnot figur obou hráčů splňuje.

U enginu, který počítá jen s jedním spuštěním evaluace, ale musí intervaly vypadat nějak takto: $(-1000, -2.01)$, $(-2, -0.01)$, $(0, 1.99)$, $(2, 1000)$. Hodnoty -1000 a 1000 v tomto případě zastupují hodnoty $-\infty$ a ∞. Evaluace žádné pozice tyto hodnoty nikdy nepřekročí.

7.3.7 Vyhodnocení implementace proof-of-concept GPU enginu

Podařilo se implementovat proof-of-concept GPU enginu podle návrhu z předchozí kapitoly. Engine využívá 4 bloků vláken pro paralelní aspirační prohledávání s pevně zadanými intervaly pro jedno spuštění. Hloubku prohledávání je možno zadat jako parametr.

Jako další vylepšení by bylo možno implementovat iterativní prohlubování, paralelní aspirační prohledávání s dynamickými intervaly a opakovaným spouštěním v případě neúspěchu prohledávání. Bylo by možné použít složitější evaluační funkci a pokročilejší řazení tahů.

Srovnání výkonu vylepšené verze proof-of-concept GPU enginu s jeho prvotní verzí a s CPU enginem je v tabulce 8.7 v kapitole o testování.

Kapitola 8

Testování

8.1 Testovací prostředí

Veškeré testy proběhly na počítačové sestavě s těmito parametry:

- CPU AMD Phenom II X940 (4 jádra, 3.0Ghz)
- RAM 4GB DDR2
- GK AMD Radeon HD 5770 1GB
- OS Windows 7 64 bit

V testech srovnávajících CPU a GPU má CPU výhodu, neboť ve své době patřil do vyšší výkonové třídy než dané GPU, ale ani použití silnější grafické karty nebo slabšího procesoru by závěr měření neovlivnilo.

8.2 Testování rychlosti paralelní evaluace šachových pozic na GPU v závislosti na použité knihovně

V tomto měření je porovnávána doba běhu implementace paralelní evaluace šachových pozic v jazyku C# za použití knihovny CUDAfy.NET a v jazyku Java za použití knihovny Aparapi.

Výsledky tohoto měření jsou zajímavé. Zatímco při nízkém počtu evaluovaných pozic je výrazně rychlejší implementace využívající knihovnu CUDAfy.NET, při vysokém počtu evaluovaných pozic se situace obrací a lepších výsledků dosahuje implementace využívající knihovnu Aparapi.

Zároveň při použití knihovny CUDAfy.NET při evaluaci vysokého počtu pozic dochází k pádům ovladačů grafické karty a rozpadnutí obrazu na monitoru. Tato knihovna zřejmě není optimalizována pro vytváření vysokého počtu vláken na GPU a nelze ji pro podobné účely doporučit.

Tabulka 8.1: Doba běhu paralelní statické evaluace šachových pozic na GPU v závislosti na technologii implementace

Počet pozic	Aparapi [ms]	CUDAfy.NET [ms]
1	148	15
100	147	15
10000	181	42
50000	255	191
100000	311	415
200000	466	917

8.3 Srovnání rychlosti evaluace šachových pozic na CPU a GPU

Toto měření srovnává schopnost CPU a GPU evaluovat různé počty šachových pozic. Evaluační funkce je pro CPU i GPU shodná. CPU provádí evaluaci sekvenčně pro celé pozice i pro pole každé pozice. Implementace pro GPU využívá knihovny Aparapi a provádí evaluaci paralelně pro tolik pozic zároveň, kolik vláken dokáže zároveň provádět, s tím že pro evaluaci každé pozice je použito 64 vláken.

Tabulka 8.2: Doba běhu paralelní statické evaluace šachových pozic na GPU a sekvenční statické evaluace na CPU v závislosti na počtu pozic

Počet pozic	GPU [ms]	CPU [ms]
1	148	1
100	147	4
10000	181	16
50000	255	65
100000	311	118
200000	466	218
500000	811	572
1000000	1059	1116

Z měření plyne, že režie zahájení a ukončení výpočtu a kopírování dat do a z paměti GPU je při nižších počtech evaluovaných pozic příliš výrazným negativním faktorem. Teprve při počtu pozic blížícímu se milionu začíná být výpočet na GPU včetně režie rychlejším.

8.4 Unit testy CPU enginu

Pro zajištění bezchybnosti implementace CPU enginu byly vytvořeny desítky Unit testů [47] ověřující správnost metod kritických tříd. Byly testovány metody reprezentující tyto akce:

- Převod tahu na řetězec
- Získání indexů, na které může figura táhnout
- Výpočet sumy hodnot soupeřových figur, na které figura útočí
- Získání notace figury
- Zjištění příslušnosti figury k hráči
- Získání možných tahů v pozici
- Převod mezi interní reprezentací pozice a FEN notací
- Získání indexů šachovnice z šachové notace pole
- Evaluace zadaných pozic - hledání matu

Po každé změně enginu bylo pomocí spuštění testů ověřeno, zda nedošlo k zanesení chyby do kódu.

8.5 Testování vlivu rozšířeného prohledávání šachů u CPU enginu

Bylo testováno 8 úloh typu mat druhým tahem, 6 úloh typu mat třetím tahem a 3 úlohy typu mat čtvrtým tahem. Pro úlohy typu mat druhým tahem byla nastavena základní hloubka prohledávání 5 (2 tahy pro prvního hráče do matu, 1 tah soupeře před matem, 1 tah soupeře po matu a 1 tah prvního hráče na sebrání krále), pro mat třetím tahem 7 a pro mat čtvrtým tahem 9. Notace pozic testovaných úloh jsou obsaženy na přiloženém CD.

Tabulka 8.3: Vliv rozšířeného prohledávání šachů na rychlost nalezení matů druhým, třetím a čtvrtým tahem

Přidaná hloubka	2. tahem [s]	3. tahem [s]	4.tahem [s]
0	2,0	16,5	-
1	2,5	19,9	233,8
2	2,5	20,2	235,3
3	1,5	2,6	31,5
5	1,9	4,8	3,7
7	3,6	1,6	1,4

Tabulka ukazuje, že na přidané hloubce pro rozšířené prohledávání výrazně záleží. Sudé hodnoty nemají smysl, neboť rozšíření začíná z pozice, kdy je na tahu hráč, který byl na tahu v iniciální pozici, a poslední sudý tah tak vždy bude tahem soupeře, který už nic nezmění.

Rozšíření pouze o jeden tah za daných podmínek také nemá smysl, neboť při liché hloubce prohledávání reálně k hlubšímu prohledávání nedojde, neboť v nejvyšší hloubce je na tahu soupeř a rozšířené prohledávání se neprovede.

Rozšíření o tři tahy vede až k několikanásobnému zrychlení. Mat druhým tahem je při něm vždy nalezen už při druhém běhu iterativního prohlubování do hloubky 2, neboť druhý tah řešící tuto úlohu je v podání enginu vždy šachem (ve skutečnosti matem) a ověření pomocí sebrání krále proběhne okamžitě a ne až v dalších iteracích.

Stejně tak mat třetím tahem je nalezen už ve čtvrté iteraci a mat čtvrtým tahem v šesté iteraci.

Rozšíření o pět a více tahů výrazně urychluje nalezení matu čtvrtým tahem. U sekvence tahů vedoucích k matu, kde některý z tahů není šachem (takzvaný tichý tah), ale může vést k prodloužení doby prohledávání.

Celkově z vyhodnocení rozšířeného prohledávání plyne, že může mít buď mírně negativní vliv nebo výrazně pozitivní vliv. Čím větší rozšíření, tím větší je urychlení u hledání matů vysokým počtem tahů. Toto vylepšení se do enginu rozhodně vyplatí zahrnout.

8.6 Srovnání rychlosti evaluace pozice na CPU a GPU (jeden blok vláken)

Toto měření porovnává prvotní verzi GPU enginu s jedním blokem 128 vláken s odpovídajícím C++ enginem běžícím na CPU. U GPU enginu není do doby běhu započítána inicializace OpenCL, která trvá několik vteřin.

Počty prohledaných pozic závisejí na iniciální pozici, pro kterou je spuštěna evaluace. Počet prohledaných pozic není jediným faktorem, neboť záleží také na počtu figur na šachovnici a jejich uspořádání, což ovlivňuje počet generovaných tahů. Tyto tahy jsou generovány, i když výsledná pozice není díky ořízntutí navštívena.

FEN notace evaluovaných pozic nejsou v tabulce zahrnuty, neboť jsou pro přehledné zobrazení příliš dlouhé. Notace jsou obsaženy na přiloženém CD. Hloubka prohledávání byla pevně nastavena na 5 tahů.

Tabulka 8.4: Průměrná doba běhu v závislosti na evaluované pozici

Počet prohledaných pozic	CPU [s]	GPU [s]
10000	0,077	0,926
26000	0,142	2,045
52000	0,297	5,148
65000	0,273	5,630
333000	1,303	35,157

Z měření plyne, že CPU má nad jedním blokem vláken na GPU jasnou převahu. I při použití silnější grafické karty nebo slabšího procesoru by se

výsledek nezměnil.

CPU v průměru prohledá přibližně 250000 pozic za vteřinu. Jeden blok vláken na GPU oproti tomu zvládne prohledat pouze 10000 pozic za vteřinu. Toto číslo by se rozhodně dalo zvýšit pomocí optimalizace algoritmu běžícího na GPU, ale k hodnotám dosahovaným CPU se za použití jednoho bloku vláken přiblížit nedokáže.

8.7 Srovnání rychlosti evaluace pozic na GPU v závislosti na počtu bloků vláken

Pro zjištění, kolik bloků vláken lze současně spustit bez výraznějšího dopadu na výkon bylo testováno spouštění prvotní verze GPU enginu s více bloky po 64 nebo 128 vláknech. Každý blok tedy nezávisle evaluoval stejnou pozici.

Předpokladem bylo, že do určitého počtu bloků bude dopad na výkon téměř nulový a následně začne vzrůstat. Zjištěný maximální počet bloků bez dopadu na výkon pak bude využit při implementaci enginu s více bloky.

Testování proběhlo při evaluaci pozice vyžadující prohledání přibližně 52000 pozic. FEN notace pozice je obsažena na přiloženém CD.

Tabulka 8.5: Průměrná doba běhu v závislosti na počtu bloků vláken

Počet bloků	Počet vláken v bloku	Doba běhu [s]
1	128	5,27
2	128	5,15
3	128	5,17
4	128	5,64
5	128	10,33
8	128	10,60
9	128	15,57
1	64	3,45
2	64	3,45
3	64	3,46
4	64	3,57
5	64	6,93
8	64	7,18
9	64	10,38

Výsledky měření ukázaly, že předpoklad byl správný a na grafické kartě AMD Radeon HD 5770 lze spustit prvotní verzi GPU enginu s až 4 bloky vláken téměř bez dopadu na dobu běhu.

Ve chvíli, kdy počet bloků překročí násobek 4, doba běhu narůstá. Tento jev se projevil nezávisle na počtu vláken v bloku. Z toho se dá usoudit, že

GPU na grafické kartě AMD Radeon HD 5770 provádí vždy maximálně 4 bloky vláken současně.

Výkonnější grafická karta s větší lokální pamětí by pravděpodobně dokázala provádět vyšší počet bloků vláken současně.

Zároveň se ukázalo, že bloky s 64 vlákny běží rychleji než bloky se 128 vlákny, i když byla pro evaluaci zvolena taková pozice, kde není více vláken než 64 potřeba a přebytečná vlákna tak nevykonávají žádnou užitečnou práci. Samotná nutnost tato vlákna synchronizovat s ostatními pomocí bariér ale vede k vysokým negativním dopadům na výkon.

Možná by se tedy vyplatilo snížit počet vláken v bloku na nižší hodnotu než 128, i když to bude znamenat neschopnost evaluovat všechny teoreticky dosažitelné pozice. Například 96 vláken by mělo postačovat na veškeré pozice reálně dosažitelné v běžné hře.

8.8 Srovnání rychlosti evaluace pozice na GPU v závislosti na intervalech pro aspirační prohledávání

Byla měřena doba evaluace a počet prohledaných pozic (včetně vícenásobného prohledání stejné pozice odlišnými bloky) při evaluaci zadané pozice s pevně zadanou hloubkou rovnou 5, při které je evaluace pozice rovna 20. FEN notace pozice je obsažena na přiloženém CD.

Pro zjednodušení není uvedeno všech 8 hodnot parametrů α a β ale pouze 5 hodnot, neboť platí, že $\beta_x = \alpha_{x-1} - 0.01$. Řádky, které mají vyplněny pouze hodnoty α_0 reprezentují aspirační prohledávání za pomocí pouze jednoho bloku, bez dělení na podintervaly.

Tabulka 8.6: Průměrná doba běhu a počet prohledaných pozic v závislosti na intervalech pro aspirační prohledávání a paralelní aspirační prohledávání

α_0	α_1	α_2	α_3	β_3	Počet prohl. pozic	Doba běhu [s]
-1000	-	-	-	1000	59033	4,06
-1000	-2	0	2	1000	67064	4,20
-5	-	-	-	35	59033	4,05
-5	5	15	25	35	89037	3,83
17	-	-	-	25	46922	3,30
17	19	21	23	25	119390	3,01
18,5	-	-	-	22,5	42572	3,05
18,5	19,5	20,5	21,5	22,5	114331	2,98

Měření ukázalo, že aspirační prohledávání se správně zvolenými intervaly je rychlejší než prohledávání bez aspirace. Pro určení intervalů je ale třeba mít informace o evaluované pozici, získané například z předchozích běhů při

iterativním prohlubování. Není tedy univerzálně použitelné při jednorázovém spuštění prohledávání.

Čím menší je zvolený interval, tím kratší je doba prohledávání, neboť dochází k výraznějšímu ořezávání stavového prostoru.

Zároveň se ukázalo, že paralelní aspirační prohledávání dělící interval na několik menších je rychlejší než běžné aspirační prohledávání, přestože musí provádět více bloků vláken a prohledat více pozic. Díky tomu, že pracuje s několika menšími intervaly, je v každém z nich dosaženo výraznějšího ořezání než při použití jednoho většího intervalu. Paralelní verze byla až o 10% rychlejší.

8.9 Srovnání rychlosti evaluace pozice na CPU a GPU (více bloků vláken)

Toto měření porovnává proof-of-concept GPU enginu s paralelním aspiračním prohledáváním se 4 bloky o 96 vláknech (v tabulce jako "GPU concept") s jednoduchým C++ enginem běžícím na CPU a s prvotní verzí GPU enginu s jedním blokem o 128 vláknech (v tabulce jako "GPU 1.verze"). U GPU enginů není do doby běhu započítána inicializace OpenCL, která trvá několik vteřin.

FEN notace evaluovaných pozic nejsou v tabulce zahrnuty, neboť jsou pro přehledné zobrazení příliš dlouhé. Notace jsou obsaženy na přiloženém CD. Hloubka prohledávání byla pevně nastavena na 5 tahů.

Uvedené počty prohledaných pozic platí pro CPU engine a prvotní verzi GPU enginu. Hodnoty pro proof-of-concept GPU enginu jsou díky běhu více bloků v rámci paralelního aspiračního prohledávání odlišné.

Tabulka 8.7: Průměrná doba běhu v závislosti na evaluované pozici

Počet prohl. pozic	CPU [s]	GPU 1.verze [s]	GPU concept [s]
10000	0,077	0,926	0,776
26000	0,142	2,045	1,642
52000	0,297	5,148	4,422
65000	0,273	5,630	4,568
333000	1,303	35,157	31,312

GPU engine s více bloky vláken je díky nižšímu počtu vláken v každém bloku a v některých pozicích i díky paralelnímu aspiračnímu prohledávání mírně rychlejší než prvotní verze GPU enginu. Stále se ale ani zdaleka nemůže srovnávat s enginem běžícím na CPU. Proč tomu tak je je diskutováno v následující sekci.

8.10 Zhodnocení možností využití GPU pro šachový engine

Veškerá měření ukázala, že výkon GPU enginu je při evaluaci šachové pozice pouze zlomkový oproti CPU enginu. Existuje pro to mnoho důvodů. Méně podstatnými důvody, které ovlivnily měření, jsou neprovedení výraznější optimalizace algoritmu běžícího na GPU a nižší výkon grafické karty, na které bylo testováno.

Hlavním důvodem je to, že ať rozdělíme prováděnou práci mezi bloky vláken na GPU jakkoliv, ve výsledku stejně musí jeden z bloků provést kompletní prohledání. I když bude v tomto prohledání dosaženo mnohem výraznějšího ořezání (nebo bude hloubka prohledávání o 1 nižší díky rozdělení stavového prostoru mezi bloky vláken), nevyrovná to obrovský výkonový rozdíl mezi CPU a jedním blokem vláken na GPU.

Zvýšit výkon tohoto bloku na úroveň CPU je nereálné. Algoritmus který na něm běží má do ideálního algoritmu pro paralelní provádění velmi daleko. Každé vlákno v něm provádí jiné množství práce, vlákna čtou ze stejných míst v lokální paměti, vyžadují velmi častou synchronizaci a musejí opakovaně provádět stejné výpočty z důvodu nedostatku lokální paměti.

Odstranit všechny tyto problémy je nemožné. V ideálních algoritmech pro GPU provádějí všechna vlákna v danou chvíli totožnou operaci, pouze nad odlišnými daty. Toto je v přímém rozporu s komplexitou šachů. Už samotná přítomnost různých figur na šachovnici vede k nežádoucímu větvení.

Dalším problémem je, že paralelní verze algoritmu MiniMax s alfa-beta ořezáváním nepřináší velké zrychlení oproti sekvenční verzi. Sekvenční verze běžící na mnohem silnějším procesoru tedy bude vždy rychlejší než paralelní verze běžící na více slabších procesorech.

Závěr

Tato práce se zabývala možnostmi využití GPGPU pro šachovou AI. V jejím úvodu byla čtenáři přiblížena teorie her a typické algoritmy pro hry dvou hráčů. Následně byla provedena rešerše existujících šachových AI.

Na základě této rešerše a analýzy vyhodnocující možnosti využití různých algoritmů a heuristik pro implementaci šachové AI využívající GPU byl vypracován návrh proof-of-concept GPU šachové AI.

Na základě tohoto návrhu byl proof-of-concept implementován. Zároveň byla implementována i šachová AI využívající CPU pro účely srovnání. V kapitole o testování pak byly obě AI porovnány.

Z výsledků porovnání vyplynulo, že implementace šachové AI využívající GPU je možná, ale její výkon je v porovnání s CPU verzí neuspokojivý.

Důvodem je zejména to, že paralelní verze algoritmů používaných v šachových AI nepřinášejí velké zrychlení oproti svým sekvenčním verzím. Díky tomu není možné efektivně využít výkonu GPU.

Dalším problémem je, že silná šachová AI se neobejde bez velké transpoziční tabulky, do které si ukládá informace o prohledaných pozicích.

U GPU implementace by bylo nutné tuto tabulku umístit do globální paměti. Nutnost k této paměti často přistupovat z různých bloků vláken by pak vedla k negativním dopadům na výkon.

V této práci se podařilo zjistit tato a mnohá další omezení, která vedou k tomu, že GPU není šachovými AI využíváno. Bez výrazné změny architektury dnešních GPU není reálné, že by se to změnilo. Nezbývá tedy než tuto doménu ponechat procesorům, které jsou pro potřeby šachových AI mnohem lépe koncipovány.

Literatura

[1] *Proměna pěšce, braní mimochodem a srovnatelná hodnota figur* [online]. [cit. 2015-04-27]. Dostupné z: http://www.chess.cz/www/mladez/metodika/zakladni-sachovy-vycvik/3_lekce.html

[2] Isenberg, G.: *Chess Engine Communication Protocol* [online]. [cit. 2015-04-18]. Dostupné z: https://chessprogramming.wikispaces.com/Chess+Engine+Communication+Protocol

[3] Catherine Ray: *How Stockfish Works: An Evaluation of the Databases Behind the Top Open-Source Chess Engine* [online]. [cit. 2015-01-02]. Dostupné z: http://rin.io/chess-engine/

[4] Jun: *What is the difference between GPU and CPU?* [online]. [cit. 2015-04-18]. Dostupné z: http://allegroviva.com/gpu-computing/difference-between-gpu-and-cpu/

[5] David Poole, A. M.: Artificial Intelligence: Foundations of Computational Agents. Cambridge University Press, 2010, [cit. 2015-04-17].

[6] Kasparov, G.: *The Chess Master and the Computer* [online]. [cit. 2015-04-25]. Dostupné z: http://www.nybooks.com/articles/archives/2010/feb/11/the-chess-master-and-the-computer/

[7] Bruce Rosen: *CS 161 Recitation Notes - Minimax with Alpha Beta Pruning* [online]. [cit. 2015-01-02]. Dostupné z: http://cs.ucla.edu/~rosen/161/notes/alphabeta.html

[8] Houston, M.: *General Purpose Computation on Graphics Processors (GPGPU)* [online]. [cit. 2015-04-25]. Dostupné z: http://graphics.stanford.edu/~mhouston/public_talks/R520-mhouston.pdf

[9] Myerson, R. B.: Game Theory: Analysis of Conflict. Harvard University Press, 1991, [cit. 2015-03-27].

[10] Isenberg, G.: *Evaluation* [online]. [cit. 2015-04-17]. Dostupné z: `https://chessprogramming.wikispaces.com/Evaluation`

[11] Isenberg, G.: *Negamax* [online]. [cit. 2015-03-27]. Dostupné z: `http://chessprogramming.wikispaces.com/Negamax`

[12] Ding-Zhu Du, P. M. P.: Minimax and Applications. Springer Science & Business Media, 1995, [cit. 2015-04-18].

[13] Pearl, J.: *SCOUT: A Simple Game-Searching Algorithm With Proven Optimal Properties, Proceedings of the First Annual National Conference on Artificial Intelligence, Stanford University, August 18–21, 1980, pp. 143-145.* [cit. 2015-03-27].

[14] Isenberg, G.: *Principal Variation Search* [online]. [cit. 2015-04-18]. Dostupné z: `https://chessprogramming.wikispaces.com/Principal+Variation+Search`

[15] FIDE: *Pravidla šachu FIDE* [online]. [cit. 2015-03-27]. Dostupné z: `http://www.chess.cz/www/assets/files/informace/legislativa/PravidlaSachuFIDE2009.pdf`

[16] Kryukov, K.: *Forsyth-Edwards Notation* [online]. [cit. 2015-04-18]. Dostupné z: `http://kirill-kryukov.com/chess/doc/fen.html`

[17] Ross, D.: *Arpad Elo and the Elo Rating System* [online]. [cit. 2015-04-18]. Dostupné z: `http://en.chessbase.com/post/arpad-elo-and-the-elo-rating-system`

[18] Isenberg, G.: *Transposition Table* [online]. [cit. 2015-04-17]. Dostupné z: `https://chessprogramming.wikispaces.com/Transposition+Table`

[19] Isenberg, G.: *Hash Table* [online]. [cit. 2015-04-29]. Dostupné z: `https://chessprogramming.wikispaces.com/Hash+Table`

[20] Isenberg, G.: *Bitboards* [online]. [cit. 2015-04-17]. Dostupné z: `https://chessprogramming.wikispaces.com/Bitboards`

[21] Kryukov, K.: *Endgame Tablebases Online* [online]. [cit. 2015-04-18]. Dostupné z: `http://kirill-kryukov.com/chess/tablebases-online/`

[22] Isenberg, G.: *UCI* [online]. [cit. 2015-04-18]. Dostupné z: `https://chessprogramming.wikispaces.com/UCI`

[23] Daylen Yang: *Stockfish* [online]. [cit. 2015-01-02]. Dostupné z: `http://stockfishchess.org/`

[24] Robert Houdart: *Houdini Chess Engine* [online]. [cit. 2015-01-02]. Dostupné z: http://www.cruxis.com/chess/houdini.htm

[25] Don Dailey, L. K., Mark Lefler: *Komodo* [online]. [cit. 2015-04-04]. Dostupné z: http://komodochess.com/

[26] Rajlich, V.: *Rybka* [online]. [cit. 2015-04-04]. Dostupné z: http://www.rybkachess.com/

[27] Graham Banks, Ray Banks, Sarah Bird, Kirill Kryukov, Charles Smith: *Computer Chess Rating Lists* [online]. [cit. 2015-01-04]. Dostupné z: http://www.computerchess.org.uk/ccrl/

[28] Heinz van Kempen: *Chess Engines Grand Tournament* [online]. [cit. 2015-01-11]. Dostupné z: http://www.husvankempen.de/nunn/

[29] Bauer, I.: *IPON rating list* [online]. [cit. 2015-04-18]. Dostupné z: http://www.inwoba.de/

[30] nVidia: *CUDA C Programming Guide* [online]. [cit. 2015-04-18]. Dostupné z: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-hierarchy

[31] Farber, R.: *OpenCL™ – Memory Spaces* [online]. [cit. 2015-04-18]. Dostupné z: http://www.codeproject.com/Articles/122405/Part-OpenCL-Memory-Spaces

[32] *CUDA Programming Model Overview* [online. [cit. 2015-04-19]. Dostupné z: http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf

[33] Barbic, J.: *Multi-core architectures* [online]. [cit. 2015-04-28]. Dostupné z: http://www.cs.cmu.edu/~fp/courses/15213-s07/lectures/27-multicore.pdf

[34] *Intel® Hyper-Threading Technology* [online]. [cit. 2015-04-25]. Dostupné z: http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

[35] Robert: *Understanding the parallelism of gpus* [online]. [cit. 2015-02-28]. Dostupné z: http://renderingpipeline.com/2012/11/understanding-the-parallelism-of-gpus/

[36] Flynn, M.: Some Computer Organizations and Their Effectiveness. IEEE Trans. Comput., 1972, [cit. 2015-04-14].

[37] Samuel: *Chess on a GPGPU* [online]. [cit. 2015-01-31]. Dostupné z: http://chessgpgpu.blogspot.cz/2013/01/gpu-memories.html

[38] Isenberg, G.: *Board Representation* [online]. [cit. 2015-04-17]. Dostupné z: https://chessprogramming.wikispaces.com/Board+Representation

[39] prof. Ing. Pavel Tvrdík CSc.: *Základní paralelní algoritmy: PPS, PJ a ETT* [online]. [cit. 2015-04-19]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-PAR.2/_media/lectures/mi-par2014-prednaska1-complexity.pdf

[40] Bainville, E.: *OpenCL Sorting* [online]. [cit. 2015-04-18]. Dostupné z: http://www.bealto.com/gpu-sorting_parallel-selection.html

[41] *CUDAfy.NET* [online]. [cit. 2015-04-19]. Dostupné z: https://cudafy.codeplex.com/

[42] *.NET Framework* [online]. [cit. 2015-04-19]. Dostupné z: https://www.microsoft.com/net

[43] *Visual Studio* [online]. [cit. 2015-04-19]. Dostupné z: https://www.visualstudio.com/

[44] *NetBeans* [online]. [cit. 2015-04-19]. Dostupné z: https://netbeans.org/

[45] *Aparapi* [online]. [cit. 2015-04-19]. Dostupné z: http://code.google.com/p/aparapi/

[46] *APP SDK – A Complete Development Platform* [online]. [cit. 2015-04-25]. Dostupné z: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/

[47] Osherove, R.: *Unit Test - Definition* [online]. [cit. 2015-04-25]. Dostupné z: http://artofunittesting.com/definition-of-a-unit-test/

Příloha **A**

Seznam použitých zkratek

AI Artificial Intelligence

CCRL Computer Chess Rating Lists

CEGT Chess Engines Grand Tournament

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

FEN Forsyth–Edwards Notation

GPGPU General-Purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

GUI Graphical User Interface

IDE Integrated Development Environment

OpenCL Open Computing Language

PCB Printed Circuit Board

PVS Principal Variation Search

SDK Software Development Kit

SIMD Single Instruction Multiple Data

UCI Universal Chess Interface

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementací
  - `cpu engine` ..... CPU engine použitý pro srovnání
  - `gpu engine` ..... proof-of-concept GPU enginu
- `src` ..... zdrojové kódy
  - `impl` ..... zdrojové kódy všech implementací
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
- `misc` ..... ostatní soubory