Hochparallele Monte-Carlo-Evaluation auf CUDA

Peter Werner

HTWK Leipzig, 28.02.2012

Betreuer: Dr. rer. nat. Johannes Waldmann

Einleitung

Aufgabenstellung war die Ausführung von Monte Carlo Evaluation für das Brettspiel Havannah[1] als paralleles Programm auf der Architektur von Nividas CUDA. Dazu sollten mehrere sogenannte Playouts, das sind per Zufallszüge bis zu Ende gespielte Spiele, gleichzeitig durchgeführt werden. Diese Playouts sind wie folgt aufgebaut:

1. Wähle einen zufälligen (gültigen) Zug
2. Führe diesen auf dem aktuellem Spielbrett aus
3. Prüfe die Gewinnbedingungen

Diese Playouts werden für jeden möglichen Spielzug des Ausgangsbrettes so oft ausgeführt, bis eine festgelegte Anzahl erreicht ist. Dabei wird für jeden möglichen Zug des Ausgangsbrettes die Ergebnisse der Playouts gespeichert (wie oft gewonnen, wie oft verloren). Anhand dieser Zahlen können die Züge bewertet werden. Ziel war es nicht ein komplettes Spiel mit einer KI zu entwickeln, sondern lediglich eine Möglichkeit bereitzustellen eine Auswertung von Spielzügen auf hoch paralleler Hardware zu ermöglichen.
Der gesamte Quellcode der im Rahmen dieser Arbeit entstanden ist, steht unter der GPL Lizenz (siehe Lizenz Abschnitt am Ende dieses Dokuments. Das Programm ist auch auf GitHub vertreten: http://github.com/pwerner/HavannahCUDA.

CUDA

Compute Unified Device Architecture (kurz CUDA) ist eine proprietäre Technologie der Firma Nvidia, die es ermöglichen soll die Grafikkarten für die Berechnungen allgemeine Aufgabenstellungen einzusetzen. Dabei soll die hohe Parallelität der Hardware ausgenutzt werden um auch auf anderen Gebieten Programme schneller auszuführen. Um mit der schnellen Entwicklung Schritt zu halten wurden die sogenannten Compute Capabilities eingeführt. Diese fungiert als eine Art Version Nummer für Grafikkarten. So ist es möglich die CUDA weiter zu entwickeln und gleichzeitig eine Ausführung auf verschieden alten Grafikkarten zu ermöglichen, sofern man alle Einschränkungen der jeweiligen compute capabilities beachtet. Sie legen z.B. für gewissen Grafikkartengenerationen die maximale Anzahl an parallelen Threads oder benutzbarer Speicher fest. Für nähere Informationen sei auf den Anhang des CUDA C Programmings Guides verwiesen[10].

Um mit CUDA zu arbeiten, muss man zunächst einige grundlegende Begriffe kennen. Die Ausführungsumgebung des CUDA Programms wird in zwei verschiedene Bereiche eingeteilt. Der erste Bereich wird Host genannt. Er umfasst den herkömmlichen PC mit denen dort verfügbaren Spracheigenschaften. Alle CUDA Programme werden vom Host aus gesteuert. Der zweite Bereich bildet das Device. Es bezeichnet die Ausführungsumgebung auf der Grafikkarte. Hier stehen nur eingeschränkte Sprachmittel zur Verfügung, man hat jedoch Zugriff auf eine Vielzahl von Threads

Wie bereits erwähnt werden alle Programme für CUDA vom Host aus gestartet. Dazu bedient man sich sogenannter Kernels. Ein Kernel ist eine Methode die vom Host aus aufgerufen, jedoch auf dem Device ausgeführt wird [11]..

Für das Programmieren mit CUDA wird eine leichte Variation der Sprache C verwendet. Sie wurde um neue Modifikatoren für Funktions- und Speicher Definitionen erweitert. Außerdem wird von CUDA neue Methoden für die Speicher- und Threadverwaltung bereitgestellt [12].

Im folgenden werden die wichtigsten Neuerungen kurz vorgestellt. Für ausführlichere Beschreibungen sei auf die Dokumentation von CUDA verwiesen [10].

Alle Methoden im CUDA Programm bekommen eine von drei Gültigkeitsbereichen zugewiesen.

Sie bezeichnet die Ausführungsumgebung der Methode. Der erste Bereich wird über __host__ festgelegt. Es ist die Standardausführungsumgebung, die die Methode der Ausführung auf dem Host zuordnet. Das Gegenstück bildet __device__ der die Ausführung auf dem Device kennzeichnet. Die Verbindung zwischen den beiden Bereichen wird über die Kernels geschaffen. Diese werden über __global__ gekennzeichnet. Für __device__ und __global__ gelten die bereits erwähnten Spracheinschränkungen. So ist z.B. keine Rekursion möglich und es muss auf statische Variablen verzichtet werden. Des weiteren haben mit __global__ gekennzeichneten Funktionen keinen Rückgabewert und nur einen eingeschränkt Großen Speicher für Argumente (256 Bytes [10] für Compute Capabilities 1.x, 4 KB für Compute Capabilities 2.x). Um dennoch Daten zwischen Device und Host austauschen zu können stellt CUDA Methoden für die Speicherverwaltung bereit. Diese können jedoch nur vom Host aus aufgerufen werden. Mit cudaMalloc() kann Speicher auf der Karte reserviert werden. Analog gibt es zum Freigeben von Speicher die Methode cudaFree(). Das Kopieren wird von der Methode cudaMemcpy übernommen, die man in drei Modi verwenden kann (cudaMemcpyHostToDevice, cudaMemcpyDeviceToDevice und cudaMemcpyDeviceToHost). Variablen auf dem Device können an verschiedener Stelle gespeichert werden. Dazu gehört der Konstanten Speicher (__constant__) und der gemeinsame Speicher (__shared__). Standardmäßig wird Speicher jedoch global auf der Karte abgelegt.

Das Programm Modell bei CUDA wird aufgeteilt in Threads, Blöcke und dem Grid. Threads sind die kleinste Einheit. Sie laufen in Blöcken, die im Grid vereint sind. Den Vorteil dieser Einteilung kann man folgender Grafik entnehmen. Die Blöcke sind logisch getrennte Einheiten, die getrennt ausgeführt werden können. So wird eine dynamische Zuteilung zu verschiedenen möglichen GPUs ermöglicht.

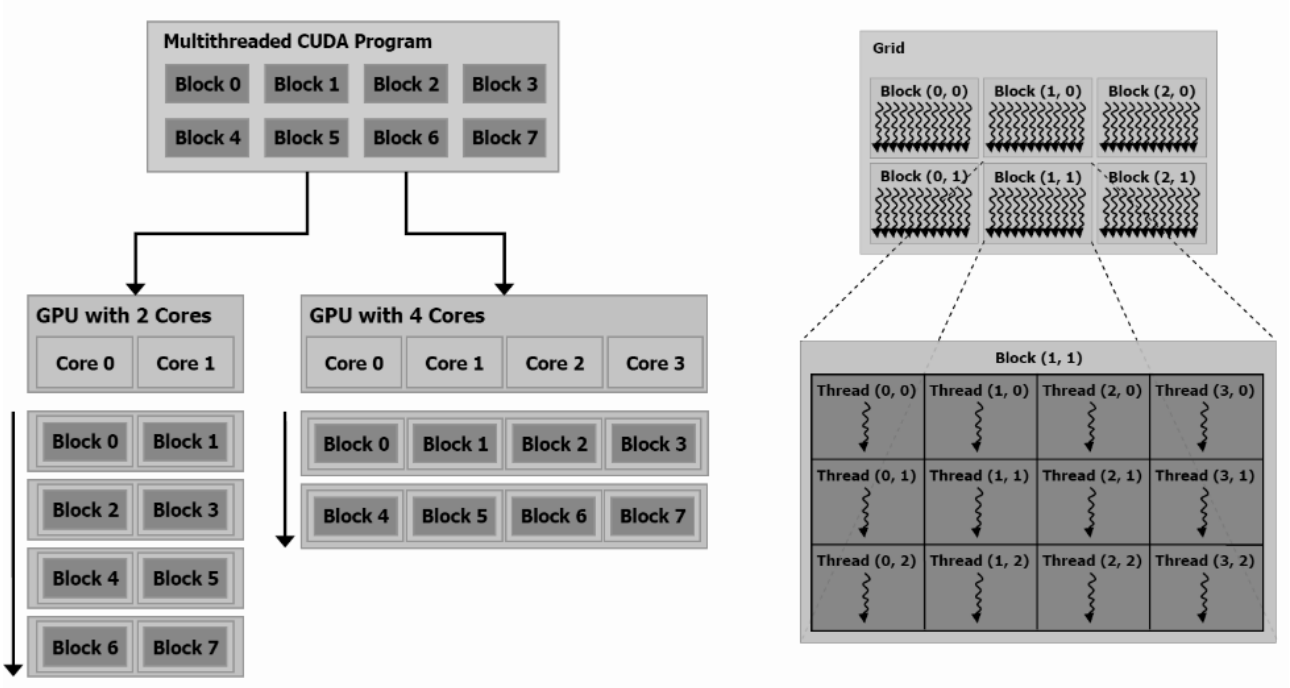

Abbildung 1: Aufteilung von CUDA Programmen in Grid, Blöcken und Threads [10]

Diese dynamische Einteilung kann man beim Aufruf eines Kernels festlegen. Dafür wird folgende Syntax eingeführt: kernel<<<grid, block>>>();
Hierbei legt grid die Dimension des Grids fest (2D), z.B. (3,2). Der Parameter block legt die Dimension der Blöcke fest (bis zu 3D), z.B. (3,4). Für die Definition dieser Variablen wird in CUDA dim3 verwendet. So würde die Definition von grid wie folgt aussehen: dim3 grid(2,3);

Für die Steuerung der Threads gibt es zwei wichtige Methoden. Die erste ist __syncthreads() die alle Threads in einem Block synchronisiert. Sie kann nur auf dem Device verwendet werden. Die zweite ist cudaThreadSynchronize(), sie für die Synchronisation aller Threads in allen Blöcken verwendet wird. Jedoch ist diese Methode lediglich vom Host aus aufrufbar.

Monte Carlo

Bei der Monte-Carlo-Simulation wird mit Hilfe von sehr häufig ausgeführten zufällig gewählten Ausführungen versucht eine Aussage über das allgemeine Verhalten zu machen. In diesem konkreten Fall werden Spielzüge betrachtet. Man kann bei komplexen Spielen in der Regel nicht vorher entscheiden ob dieser Zug langfristig zum Ziel führt. Deshalb wird dieser Spielzug oft ausgeführt und dann zufällig zu Ende gespielt um so zumindest eine Tendenz aufzuzeigen. Liegt man in unmittelbarer Nähe den gewinnbringenden Zug zu machen, wird sich dies bei der Simulation niederschlagen. Außer den allgemeinen Playouts gehören noch weitere Entscheidungen zur konkreten Ausführung. Das ganze Verfahren wird dann Monte Carlo Tree Search (MCTS) [8][9] genannt. In dieser Arbeit wird davon ausgegangen, dass alle weiteren nötigen Schritte auf einer höheren Ebene geschehen und sich nur auf die effiziente Ausführung der Playouts beschränkt. Es gibt bereits Arbeiten die MCTS für Havannah angewendet haben [7].

Havannah

Havannah ist Spiel das auf einem sechseckigen Brett gespielt wird. Die Spielfelder sind entsprechend ebenfalls sechseckig. Es gibt verschiedene Seitenlängen, die gängigsten Größen sind aber acht Kantenfelder mit insgesamt 169 Spielfeldern und zehn Kantenfelder mit insgesamt 271 Spielfeldern.

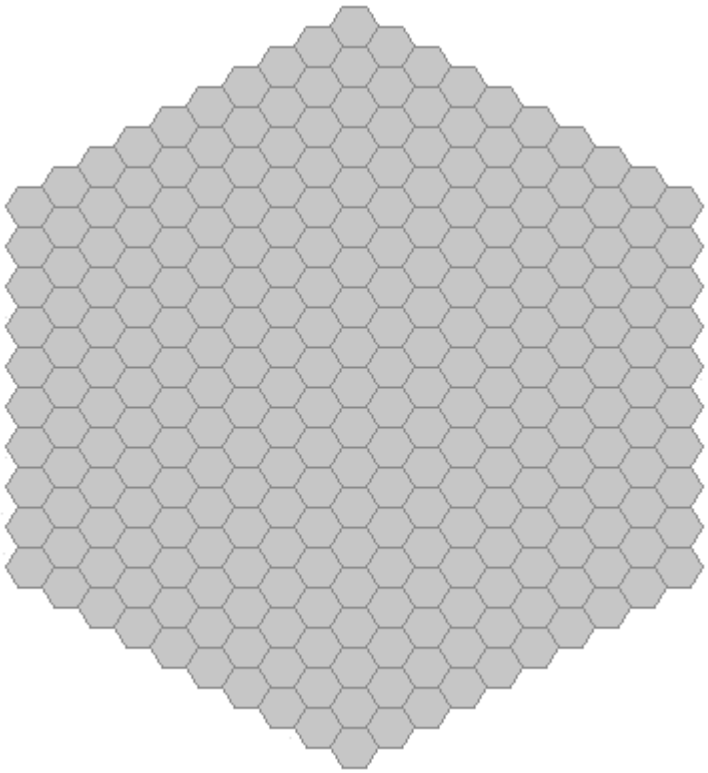

Abbildung 2: Aufbau des Spielbrettes für Havannah [1]

Es spielen jeweils zwei Spieler (weiß und schwarz, oder rot und schwarz) gegeneinander. Es beginnt der Spieler mit der weißen Farbe und setzt zunächst einen Stein auf das Feld. Schwarz kann diesen Zug für sich nutzen, indem er einen Swap anwenden. Das bedeutet, der weiße Stein wird

vom Feld genommen und der Schwarze ersetzt ihn. Nun verläuft das Spiel so, als hätte Schwarz begonnen. Dieser Spielzug ist der einzige, bei dem ein Spielstein wieder vom Feld genommen werden darf. Ansonsten wird nacheinander ein weißer und dann ein schwarzer Stein auf eine freie Stelle des Brettes gesetzt.

Das Spiel ist gewonnen, falls ein Spieler eine von drei Gewinnstrukturen erzeugen kann:

1. Ring
2. Brücke
3. Gabel

Ein Ring liegt vor, sobald ein Spieler es schafft eine zusammenhängende Struktur seiner Farbe zu setzen, die mindesten ein Feld in der Mitte umschließt. Dabei ist es unerheblich ob das Feld in der Mitte frei oder besetzt ist (egal von welcher Farbe).

Eine Brücke bezeichnet eine Struktur, bei der zwei beliebige Ecken des Spielfeldes miteinander verbunden wurden.

Eine Gabel ist die Verbindung von drei Brettseiten wobei die Ecken jedoch nicht als Seitenfelder gesehen werden.

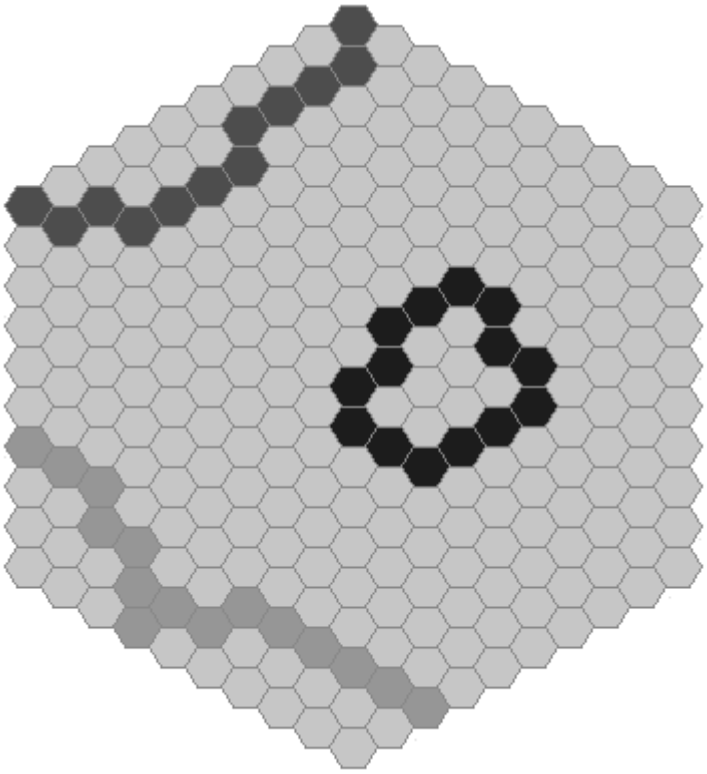

Abbildung 3: Gewinnstrukturen für Havannah. Brücke (links oben), Gabel (links unten) und Ring (rechts)

Aus der Natur des Spieles ergibt sich, dass das Spiel in jedem Fall nach so vielen Zügen endet wie es Spielfelder (plus eins, da ein Swap einmalig möglich ist) gibt.

DSF

Da es sich bei Havannah um ein Spiel handelt bei dem Verbindungen von Steinen für das Gewinnen ausschlaggebend sind, ist für die Umsetzung eine Datenstruktur nötig, die genau dies effizient verwirklichen kann.

Die in dieser Arbeit verwendete Struktur wird als Disjoint Set Forest[2] (kurz DSF) bezeichnet. Hier werden alle Elemente in Mengen gespeichert. Jede Menge ist eindeutig über ihre Wurzel gekennzeichnet. Die einzelnen Elemente in der Menge besitzen einen Zeiger zu ihrem Vorgänger auf dem Weg zur Wurzel. Ist das Element die Wurzel selber, zeigt dieser Zeiger auf sich.
Die Datenstruktur besitzt drei Methoden: Make, Union und Find. Make(x) erstellt dabei aus einem Element x eine Menge, d.h. führt einen Zeiger ein und lässt ihn auf das Element zeigen. Union(x,y) vereinigt zwei Mengen, die durch die Elemente x und y gegeben sind, dadurch, dass einer der beiden alten Wurzeln auf die andere Wurzel zeigt. Die dritte Methode find(x) verfolgt den Pfad der sich aus den Zeigern zu den Vorgängern ergeben und gibt als Ergebnis die Wurzel der Menge zurück, in der sich das Element x befindet. Für eine effiziente Laufzeit werden zwei Verbesserung eingeführt. Die erste Verbesserung ist das Vergeben einer Ranges für jede Menge. Die Menge mit einem Element hat immer Rang 0. Bei der Vereinigung wird immer die Menge mit dem kleineren Rang unter die Menge mit den größeren Rang gehangen. Somit behält die neue Menge den Rang der größeren Menge, außer die Ränge beider Mengen sind gleich wobei der neue Rang um Eins höher ist. Diese Verbesserung führt zu garantiert kürzeren Wegen gegenüber der zufälligen Wahl.
Die zweite Verbesserung ist die Pfadverkürzung oder auch Pfadkompression. Die Idee dahinter ist, dass bei der Suche nach Wurzel ausgehend von einem Element eine gewisse Anzahl Elemente besucht werden müssen. Diese Besuche sollen nun ausgenutzt werden. Dabei wird rekursiv für jedes Element erneut die Wurzel gesucht und das Ergebnis als neuen Vorgänger gespeichert. Somit erhält man für jeden besuchten Knoten eine Pfadlänge zur Wurzel von Eins. Dieses Vorgehen kann Zeit einsparen wenn sehr oft nach der Wurzel gesucht wird.
Da die Pfadsuche in dieser Arbeit auf der Grafikkarte ausgeführt wird, ist eine rekursive Lösung nicht möglich. Um dennoch nicht auf die Pfadverkürzung zu verzichten, wird eine iterative Lösung verwendet, die den Pfad zur Wurzel zweimal durchläuft. Einmal um die Wurzel zu finden und einmal um diese Wurzel als Vorgänger aller besuchten Elemente zu setzen. Diese Lösung ist nicht optimal, da sie Mehraufwand in Kauf nimmt um eine Pfadverkürzung zu erreichen, die im Worst-Case nicht mehr genutzt wird. Die Auswirkungen auf die Laufzeit werden in einem späteren Kapitel betrachtet. Eine Alternative wäre das Einführen eines neuen Zeigers, der allen Elementen als Vorgänger übergeben wird und am Ende auf die Wurzel zeigt. Bei der derzeitigen Implementierung ist dies jedoch nicht ohne weiteres möglich, da an Stelle von Zeigern der Index des Vorgängers in einem Speicherarray gemerkt wird.
Es folgt die Implementierung der Methoden. Die Struktur die hinter small_board steht wird im nächsten Kapitel genauer besprochen. Wichtig ist hier nur, dass es drei Arrays gibt cell[idx], parent[idx] und rank[idx], die die oben angesprochenen Funktionen erfüllen. Cell gibt den Inhalt des Elementes idx wieder, parent den Vorgänger des Elements an der Stelle idx und rank den Rang dieses Elements.

Make:

```
void makeSet(small_board * b, idx index) {
     /* Vorgänger ist dieses Element */
     b->parent[index] = index;
     /* Der Rang beginnt bei 0 */
     b->rank[index] = 0;
}
```

Find:

```
idx findRoot(small_board * b, idx index) {
     /* Schleife für das Finden der Wurzel */
     idx newIndex = index;
     while(b->parent[newIndex] != newIndex) {
          newIndex = b->parent[newIndex];
     }
     idx root = newIndex;
```

```
	/* Schleife für die Pfadverkürzung */
	newIndex = index;
	while(b->parent[newIndex] != newIndex) {
		idx oldIndex = newIndex;
		newIndex = b->parent[oldIndex];
		b->parent[oldIndex] = root;
	}
	return root;
}
```

Union:

```
void unionSets(small_board * b, idx index1, idx index2) {
	/* Wurzel der ersten Menge */
	idx root1 = findRoot(b, index1);
	/* Wurzel der zweiten Menge */
	idx root2 = findRoot(b, index2);
	if(root1 == root2) {
		/* Beide Elemente gehören zur selben Menge */
		return;
	}
	/* Hänge die Menge mit dem kleineren Rang unter die andere */
	if(b->rank[root1] > b->rank[root2]) {
		b->parent[root2] = root1;
	} else {
		b->parent[root1] = root2;
		if(b->rank[root1]==b->rank[root2]) {
			/* Ränge sind gleich, der neue Rang wird erhöht */
			b->rank[root2] = b->rank[root2]+1;
		}
	}
}
```

Spielfeld Implementierung

Um dieses Vorgehen umsetzen zu können, wurde zunächst eine C-Implementierung des Brettspiels Havannah benötigt. Dies ist im Struct small_board umgesetzt. Da die maximale Kantenlänge zehn beträgt, ist die vorliegende Implementierung auf diese Kantenlänge abgestimmt. Es wäre jedoch leicht möglich diese ebenfalls für eine kleinere Kantenlänge zu implementieren. Dazu muss lediglich eine passende Codierung bereitgestellt werden.

Wie bereits im Abschnitt DSF erwähnt, bilden drei Arrays die Basis für das Spielbrett. Sie haben jeweils 271 Elemente (entspricht der Anzahl Spielfelder bei einer Seitenlänge von Zehn). Über den Index wird jeweils ein Feld dargestellt. Dazu gehört der Wert dieses Spielfeldes (Leer, on Schwarz besetzt, von Weiß besetzt) das durch cell[idx] abgebildet wird. Der jeweilige Vorgänger für die Zusammenhängenden Mengen (DSF) gespeichert in parent[idx] und der dazugehörige Rang der Menge gespeichert in rank[idx]. Die Arrays cell und rank benötigen pro Element nur ein Byte, da dort nur Werte von 0,1,255 (White, Black und Empty) im Fall von cell und Werte von 0-136 im Fall von rank (größter möglicher Rang bei 271 Spielfeldern und abwechselnder Züge) gespeichert werden. Parent speichert die Indizes von cell und benötigt somit zwei Bytes (0-270).
Zusätzlich wird die Anzahl der bereits gespielten Züge time gespeichert (zwei Bytes). Für das bessere Nachvollziehen wird ebenfalls der letzte ausgeführte Zug in last gespeichert (wiederum 2 Bytes). Der Status des Spieles wird in einem weiteren Byte state codiert. Das oberste Bit (a) zeigt

die Farbe des Spielers an, der den nächsten Zug ausführen wird (0 für White, 1 für Black). Die darauf folgenden zwei Bits (b) zeigen den Gewinngrund an (00 – keiner, 01 – Ring, 10 – Brücke, 11 – Gabel).
Das vierte Bit (c) zeigt die Gewinnerfarbe. Dieses Bit sollte nur ausgewertet werden, sobald (b) nicht mehr 00 zeigt. Die untersten vier Bits (d) legen die Größe des Brettes in binärer Form fest.

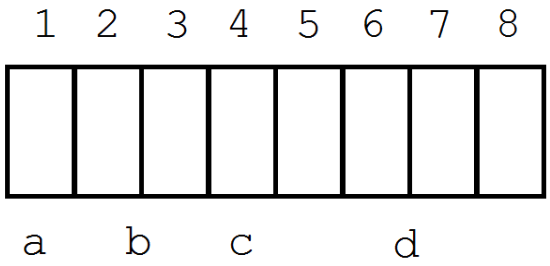

Abbildung 4: Aufteilung des Bytes 'state'

Aus den verwendeten Speicherelementen lässt sich eine Größe von 1089 Bytes berechnen (die tatsächliche Größe des struct beträgt jedoch 1092 Bytes).

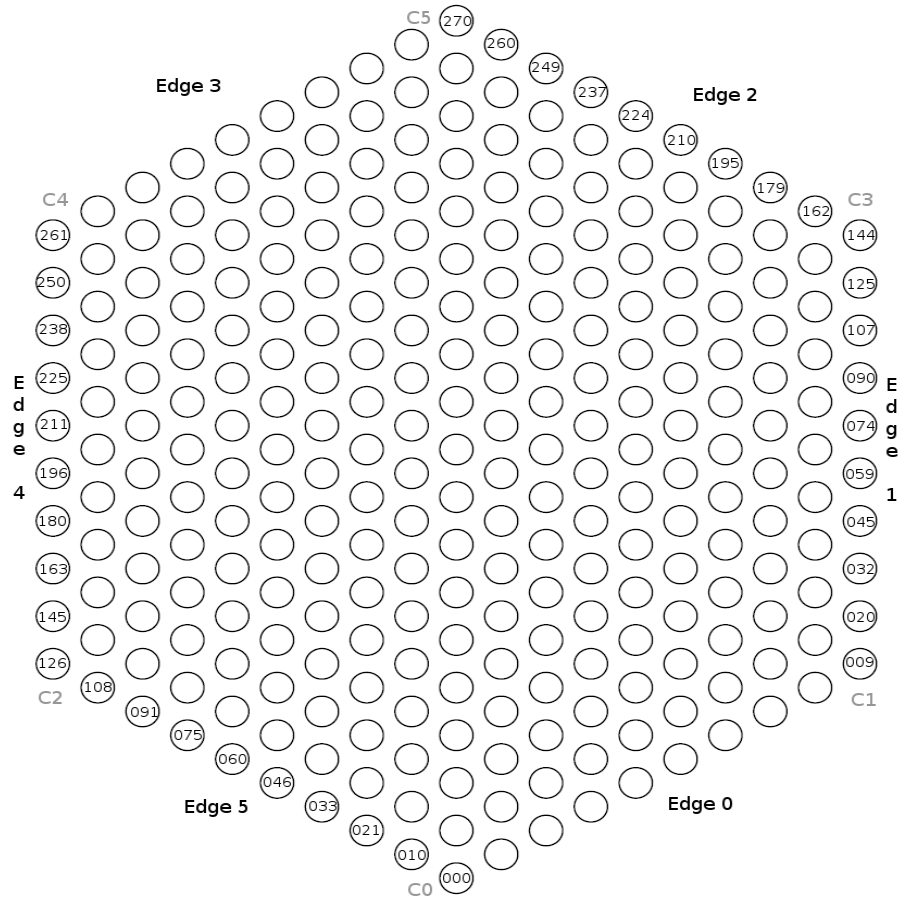

Abbildung 5: Codierung des Spielbrettes für die Implementierung

Um das Spielfeld einsetzten zu können, muss die eindimensionale Struktur der Arrays interpretiert werden. Dies geschieht mit Hilfe des Structs code_board_10. Dieses definiert die Indizes für die einzelnen Zeilen (insgesamt 19), Ecken (insgesamt sechs) und Kanten (ebenfalls sechs). Des weiteren ist die maximale und minimale Zeile gespeichert. Diese Codierung ist in Abbildung 5 dargestellt. Dabei werden die Zeilen als zwei Indizes dargestellt, nämlich Anfangsindex und Endindex. Für die Zeile 0 ergibt sich somit 001-008 als Bereich. Die erste Ecke wird als 000

codiert, die letzte als 270. Die Kanten werden jeweils als achter Array angegeben. Für die Ecke 2 ergibt sich [162,179,195,210,224,237,249,260].
Die dargestellte Codierung wird von der Methode void init(code_board_10 * code) umgesetzt , inklusive der Indizes für die Kanten und Ecken. Auf dieser Basis ist es unter anderem möglich die Zeile für ein Element in Erfahrung zu bringen.

Für das Ausführen der Züge und Gewinnbewertung ist außerdem nötig, alle Nachbarn eines Elements in Erfahrung zu bringen (bis zu sechs Nachbarn sind möglich). Dabei sind die wieder die Indizes wichtig, da von diesen jeweils Abhängigkeiten entstehen (siehe Implementation der Ring Findung). Die Methode void getNeighbours(const code_board * code, idx index, neighbours * n); übernimmt diese Aufgabe für ein gegebenen Index und speichert das Ergebnis in n. Die Indizes der Nachbarn sind wie folgt festgelegt: Der südwestliche Nachbar wird mit 0 codiert und der nordöstliche mit 1. Der nordwestliche mit 4 und der nördliche mit 5. Der südliche mit 2 und der südöstliche mit 3. Das folgende Abbildung 6 zeigt die Aufteilung im Umfeld des Spielbrettes.

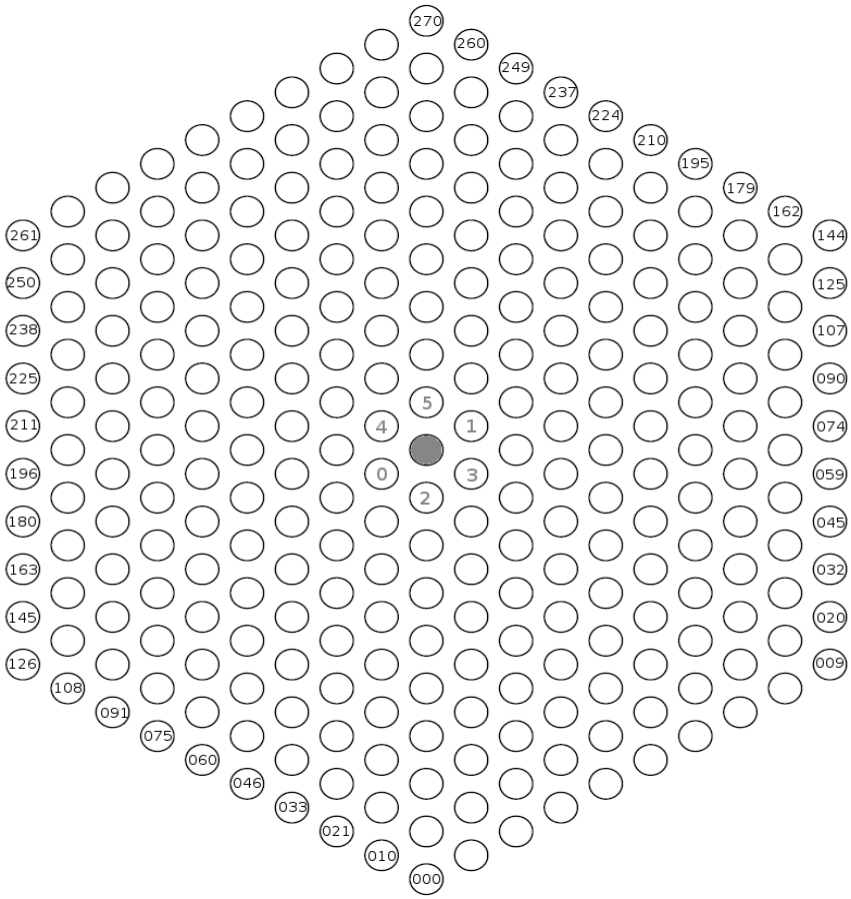

Abbildung 6: Codierung der Nachbarn eines Spielfeldes

Playout Implementierung

Hinweis:
Um eine bessere Lesbarkeit zu erreichen werden alle Methoden die auf dem Device ausführt werden (__device__ oder __global__) mit dem Suffix '_d' versehen: do_playout_d. Diese Methoden müssen immer in der gleichen Datei implementiert werden in der sie auch aufgerufen werden. In den meisten Fällen unterscheiden sich _d-Methoden nicht von ihren Pendant ohne diesen Zusatz, mit Ausnahme der Berechnung für die Pseudozufallszahlen. Die Methode ohne _d sind bis auf die Steuerungselemente vor allem zum Testen und Debuggen des Codes gedacht.

Die Ausführung des Playouts wird von der Methode playout_d übernommen. Diese steuert die einzelnen Playout Schritte, die durch die Methode playoutStep_d durchgeführt werden. Für die Ausführung des Playout Steps wird zunächst ein zufälliger Zug gewählt. Um diesen Vorgang zu beschleunigen, stehen bei der Zugwahl nur noch offene Züge zur Auswahl. Um das zu gewährleisten, wird vom Ausgangsbrett alle noch offenen Züge errechnet. Dazu werden alle Indizes des Brettes durchlaufen und überprüft ob das aktuelle Feld noch frei ist. Alle Züge die zu freien Feldern führen werden in einem Array gespeichert. Das Array und seine Größe wird den Playout Steps dann zur Verfügung gestellt. Es wird solange gespielt bis einer gewinnt oder bis es keine freien Felder mehr gibt.

Die eigentliche Auswahl erfolgt über den Pseudozufallszahlen Generator von CUDA. Dieser wird beim Programmstart initialisiert (Kernel init_rnd) und den einzelnen Threads werden verschiedene Instanzen vergeben um möglichst verschiedene Ergebnisse zu erhalten.

Um nicht nach jedem Zug die freien Felder neu zu berechnen wird das vorliegende Array mit freien Zügen bei jedem Schritt angepasst. Nachdem die Zufallszahl erzeugt wurde, wird der derzeit letzte Zug an die jetzt frei gewordene Stelle geschrieben und der Zeiger auf das letzte Element um ein Element weiter zurück gesetzt. Der gewürfelte Zug wird zurück gegeben. Die Methode getIndex_d übernimmt dabei das Würfeln und Umsetzten der Zeiger, wobei der Aufrufer von getIndex_d sich um das Dekrementieren des last-Zeigers kümmern muss.

Nachdem der Zug feststeht, wird als erstes der Ring Check ausgeführt, da dieser noch nicht verbundene DSF Bäume benötigt. Dabei wird überprüft ob durch den Zug zwei Nachbarn, die nicht direkt nebeneinander liegen und zwischen sich mindestens ein freies oder fremdes Feld haben, bereits verbundenen sind. Falls ja, schließt der Zug einen Kreis und das Spiel ist gewonnen. Dies wird mit der Methode ringWin_d überprüft.

Falls kein Ring gefunden wurde wird der Zug über die Methode perf_move_d ausgeführt. Diese setzt für die aktuelle Farbe den Stein falls das Brett noch nicht voll ist. Anschließend werden alle freundlichen Nachbarn miteinander verbunden. Die Zugzahl wird hochgezählt und die nächste Zugfarbe wird festgelegt.

Die Methode bridgeWin_d überprüft ob die Gewinnstruktur Bridge auf dem Spielbrett vorkommt. Dazu wird zunächst überprüft ob eine Ecke von der aktuellen Farbe besetzt ist. Falls ja wird weiterhin überprüft ob eine Verbindung mit anderen Ecken der selben Farbe gegeben ist. Dieser Vorgang wird für alle Ecken ausgeführt, wobei darauf geachtet wird, dass keine Überprüfung von zwei Ecken mehrfach überprüft wird. Wird eine Verbindung gefunden wird das Spiel beendet und die Farbe die den letzten Zug ausgeführt hat wird zum Sieger erklärt.

Die Überprüfung auf die Gewinnstruktur Fork ist am aufwendigsten. Sie ist in der Methode forkWin_d umgesetzt. Sie findet im Playout Step als letztes statt um unnötige Berechnungen zu ersparen falls eine einfachere Gewinnbedingung vorliegt. Es wird für vier Kanten überprüft ob diese jeweils mit zwei weiteren Kanten verbunden sind. Die letzten beiden Kanten müssen nicht überprüft werden, da diese bei dem Vorgang auf jeden Fall überprüft werden, falls dort eine Verbindung zu zwei weiteren Kanten vorliegt. Für jeden Spielstein der aktuellen Farbe einer Kante wird überprüft ob er mit mindestens zwei anderen Kanten auf dem Spielbrett verbunden ist.

Nachdem der Playout erfolgreich ausgeführt wurde, muss noch die Bewertung des Zuges vorgenommen werden. Um die Ergebnisse der Playouts zu speichern gibt es das Struct game, das sich ein Zug sowie dessen Anzahl an gewonnenen, verlorenen und unentschiedenen Spielen festhält.

Die Playouts sind jeweils als Kernel implementiert. Es wird dabei zwischen drei

Implementierungen unterschieden:

- playouts_d nimmt die Playouts für die übergebenen Games vor, die vorher auf die Karte kopiert wurden (siehe Kernel Runner). Das bedeutet es befinden sich immer nur die aktiven Spiele auf dem Device.
- repeat_playouts_d nimmt die Playouts für einen gegebenen Bereich aller Spiele vor, die vorher auf die Karte kopiert wurden. Hierbei befinden sich alle möglichen Spiele auf dem Device. Dem Kernel wird nur per Index mitgeteilt welche Spiele gerade aktiv sind.
- playouts_npc_d ist wie repeat_playouts_d verwendet jedoch keine Pfadkompression bei der Ausführung der DSF-Operation Find.

Playout Ausführung (Kernel Runner)

Die Kernels für die Playouts müssen durch geeignete Aufrufe vom Host aus gesteuert werden. Diese Aufgabe übernehmen die Kernel Runner. Dabei kann der selbe Kernel von verschiedenen Runnern mit unterschiedlichen Einstellungen aufgerufen werden.

Derzeit existieren vier Kernel Runner rateGame1-4. Diese bekommen jeweils ein Ausgangsspielfeld und berechnen für jeden möglichen Zug die Gewinn und Verlust Anzahl.

1. rateGame1 verwendet immer so viele Blöcke mit jeweils einem Thread wie es aktive Spiele gibt. Diese Anzahl wird über eine Konstante festgelegt: MAX_ACTIVE_GAMES. Zunächst wird das aktuelle Board auf die Karte kopiert. Dann durchläuft er zwei Schleifen. Die äußere Schleife läuft über alle offene Spiele und sucht immer die gerade aktiven Spiele heraus. Die zweite Schleife durchläuft die Playouts, wobei jedes mal die aktiven Spiele auf die Karte kopiert werden, der Playout ausgeführt wird und dann das Ergebnis wieder von der Karte zurück kopiert wird.
2. rateGame2 verwendet ebenfalls soviele Blöcke mit jeweils einem Thread wie aktive Spiele verwendet werden. Er durchläuft wie rateGame1 die zwei Schleifen, jedoch wird nicht jedes mal aufs neue die aktuellen Spiele auf die Karte und zurück kopiert, sondern jeweils nur einmal pro äußeren Schleifendurchlauf
3. rateGame3 verwendet ebenfalls soviele Blöcke mit jeweils einem Thread wie aktive Spiele verwendet werden. Auch dieser Runner durchläuft die zwei Schleifen. Jedoch werden die Games nur einmal auf die Karte und einmal wieder zurück kopiert.
4. rateGame_npc verwendet das gleiche Vorgehen wie rateGame2, jedoch mit einer DSF Implementation ohne Pfadverkürzung.

Probleme

Bei der Erstellung des Programms kam es zu einigen Problemen, die im folgenden kurz dargestellt werden sollen. Falls es bereits eine Lösung für das Problem gibt, wird diese kurz erwähnt.

Das erste Problem lag im begrenzten Speicherplatz im Shared Memory auf der Grafikkarte. Die vorhandene Spielbrettrepräsentation nutze für die Darstellung eines Brettes bereits über 16 KB, was die Grenzen des Shared Speichers sprengt. Um dennoch den schnellen Speicher nutzen zu können wurde eine eigene Implementation angefertigt, die nur 1092 Bytes benötigt.

Bei der Umsetzung mussten auch die Gewinnbedingungen abgebildet werden. Für die Erkennung eines Ringes wurde dabei folgende einfache Überlegung verwendet:

1. überprüfe die Nachbarn des Feldes auf den der nächste Stein gesetzt wird
2. Finden sich mindestens zwei freundliche Nachbarn, die nicht direkt durch einen, zwei oder drei Steine verbunden sind, ist zu überprüfen ob diese Nachbarn bereits verbunden waren, bevor der neue Zug ausgeführt wird.

3. Gibt es den in 2. genannten Zustand, so existiert nach dem Zug ein Ring

Dieses Vorgehen funktioniert falls der Ring mindestens einen nicht freundlichen Nachbarn einschließt. Dieser Zustand ist im linken Teil des Bildes von Abbildung 7 dargestellt. Schließt der Ring jedoch nur freundliche Nachbarn ein, so wird dieser nicht erkannt, obwohl es offensichtlich auch ein Ring ist. Dieser Zustand ist im rechten Teil des Bildes von Abbildung 7 dargestellt.

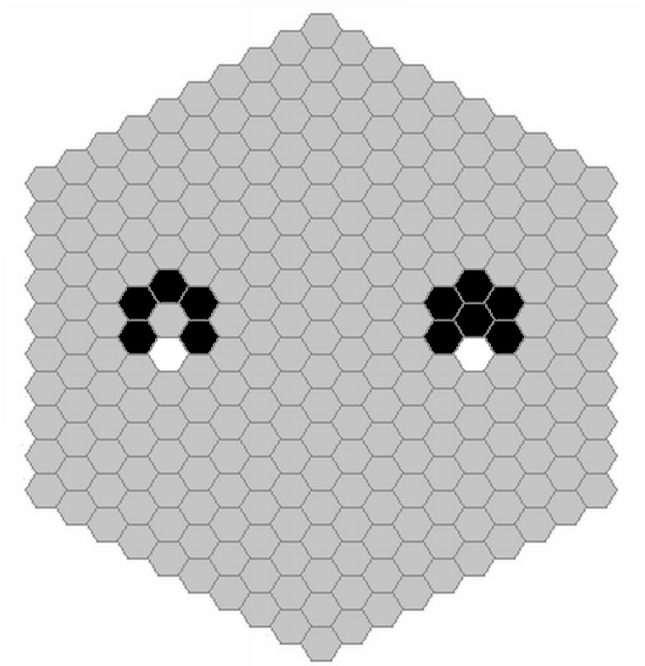

Abbildung 7: Zwei Ringe (weißer Stein markiert aktuellen Zug)

Das Problem wurde gelöst indem für jeden der sechs Nachbarn des neuen Zuges überprüft wird, ob sich bereits fünf freundliche Nachbarn in dessen Umgebung aufhalten. Falls ein Nachbar mit fünf freundlichen Nachbarn gefunden wird, ist nach dem Ausführen des Zugs ein Ring vorhanden.

Ein weiteres Problem trat in Verbindung mit der Verwendung des Shared Memory der Grafikkarte auf. Bei der Auswertung wurden verschiedenen Zügen das gleiche Ergebnis zugewiesen. Die Lösung dafür war die Aufteilung der Threads zu verbessern. Anstatt vier Blöcke mit jeweils zwei Threads, wurden acht Blöcke eingeführt die nur einen Thread beinhalten. Da der Shared Speicher pro Block getrennt ist, kam es nicht mehr zum Überschreiben des Speichers.

Bei der Einteilung von Grid und Blöcken muss man beachten, dass nur Dimensionen verwendet werden, die an zwei Stellen eine eins beinhalten, da sonst einige freien Züge bei der Auswertung nicht berechnet werden.

Verwendet man nur einen Kernlaufruf für alle Playouts, beendet sich das Programm mit der Fehlermeldung „the launch timed out and was terminated“. Der Grund für dieses Verhalten, ist der eingebaute Watchdog, der nach ca. 5 Sekunden Ausführung der Kernel unterbricht. Die Lösung liegt darin, mehrere kurze Kernelaufrufe zu verwenden.

Die Speicherverwendung sollte optimiert werden. So könnte die Kodierung des Spielbrettes in den Konstantenbereich der Grafikkarte kopiert werden und von dort allen Teilprogrammen zugänglich gemacht werden.

Die Berechnung der Nachbarn ist nicht optimal, da für jeden Aufruf von getNeighbours() neu gerechnet werden muss. An dessen Stelle wäre ein Array mit sechs Nachbarn pro Feld möglich, das ebenfalls im Konstantenspeicher der Grafikkarte unterkommen könnte.

Experimente

Hardware-Umgebung:

- Intel Core 2 Duo CPU E8400 3.00 Ghz
- 4GB (3,25 GB wegen 32 Bit)
- GeForce GTX 260

Software-Umgebung:

- Windows 7 32 Bit
- Visual Studio 2008 Professional Edition
- Dev Driver 4.0
- Cudatoolkit 4.0.17
- GPU Computing SDK 4.0.19

Für die Laufzeitmessungen wurde jeweils ein leeres Board verwendet. Es werden also jeweils 271 verschiedene Züge betrachtet. Die Parameter für die Ausführung sind weiterhin die Anzahl der auszuführenden Playout, die Anzahl der Threads (entsprich auch Anzahl der Spiele die gleichzeitig aktiv sind). Falls nicht anders beschrieben werden bei der Ausführung viele Objekte im Shared Speicher der Grafikkarte abgelegt.

Experiment 1:

Bei diesem Experiment wird die Laufzeit der ersten drei Kernel Runner bei verschiedenen Playout Anzahlen getestet. Da das Kopieren des Speichers bei den drei Runnern so unterschiedlich ist, wird eine eindeutige Verbesserung von Runner 1 (rateGame1) auf Runner 2 (rateGame2) und von Runner 2 auf Runner 3 (rateGame3) erwartet. Während der Laufzeit waren alle wichtigen Konstrukte im Shared Speicher geladen.

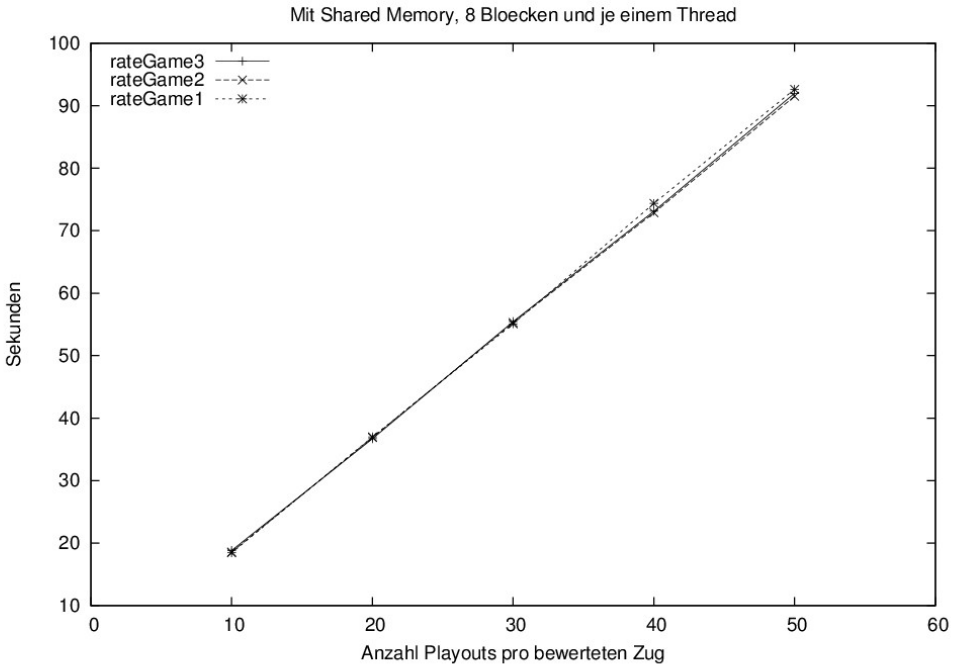

Abbildung 8: Ergebnis Experiment 1

Wie in Abbildung 8 erkennbar zeigt sich jedoch keine wesentliche Überlegenheit eines Kernel Runners. Der maximale Abstand zwischen den Laufzeiten beträgt eine Sekunde. Die Erklärung dafür ist die Menge an Speicher die in den Kernel Runnern kopiert wird. Der Kopieraufwand ist minimal, da nur wenige Bytes übertragen werden müssen (Aktive Spiele * 14 Bytes).

Experiment 2:

Das zweite Experiment hat zum Ziel einen Vergleich zwischen mehreren Blöcken mit einem Thread gegenüber weniger Blöcken mit mehr Threads zu erhalten. Für den Test wurde auf die Verwendung von Shared Memory verzichtet, da es bei mehreren Threads pro Block zu fehlerhaften Playouts kommen kann. Als Runner wurde rateGame3 verwendet. Aufgrund der hohen Laufzeit wird die Playout Anzahl gegenüber Experiment 1 reduziert. Der eine Aufruf wurde mit acht Blöcken die jeweils ein Thread haben durchgeführt. Der zweite Aufruf hatte nur vier Blöcke, dafür jedoch zwei Threads pro Block. Das Ergebnis ist in Abbildung 9 zu sehen.

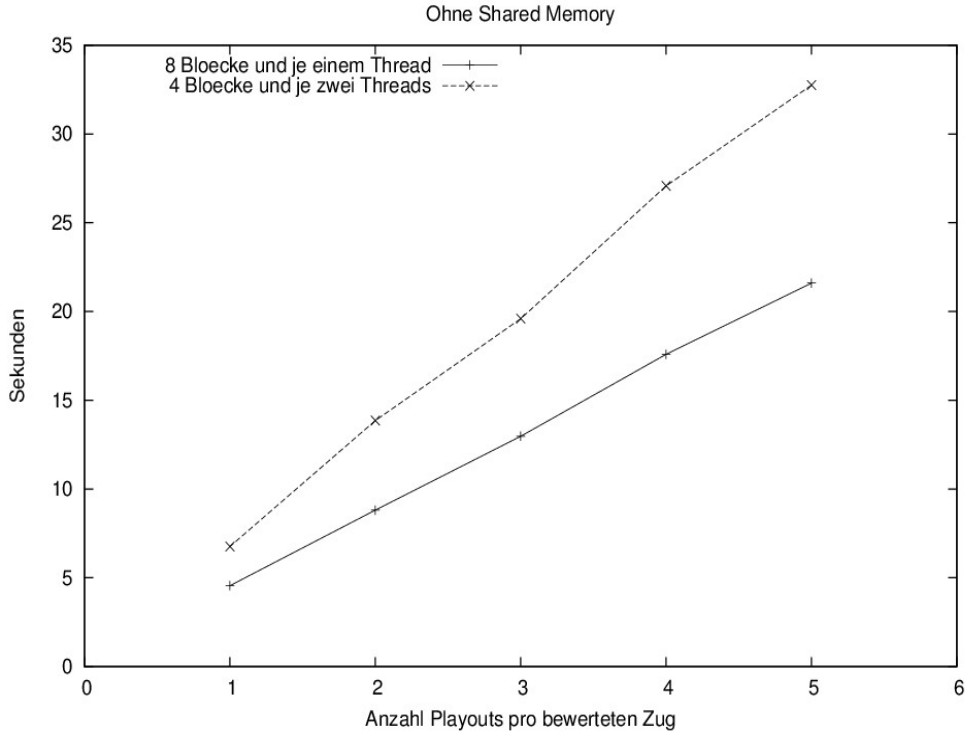

Abbildung 9: Ergebnis von Experiment 2

Es ist zu erkennen, dass die Laufzeit des Runner mit den weniger Blöcken mit steigender Playout Anzahl immer deutlicher von dem zweiten Runner abweicht. Anfangs bei einem Playout beträgt der Abstand 11 Sekunden, während der Abstand bei fünf Playouts bereits bei 20 Sekunden liegt. Die Ursache liegt darin, dass Threads in Blöcken unabhängig ausgeführt werden, während bei mehreren Threads pro Block mehr Synchronisation stattfinden muss.

Experiment 3:

Das dritte Experiment beschäftigt sich mit der Frage wie effizient die Pfadkompression über Iteration umgesetzt werden konnte. Für den Vergleich wurde der Runner rateGame2 ohne Pfadkompression umgesetzt und mit seinem Pendant mit Pfadkompression verglichen. Die Ergebnisse sind in Abbildung 10 zu sehen.

Bereits bei einer Playout Anzahl von zehn beträgt der Abstand zwischen den Laufzeiten der beiden Runner eineinhalb Sekunden, später bei einer Playout Anzahl von 50 sind es sechseinhalb Sekunden. Wie deutlich zu erkennen ist, liegt die Laufzeit mit Pfadkompression niemals vor oder

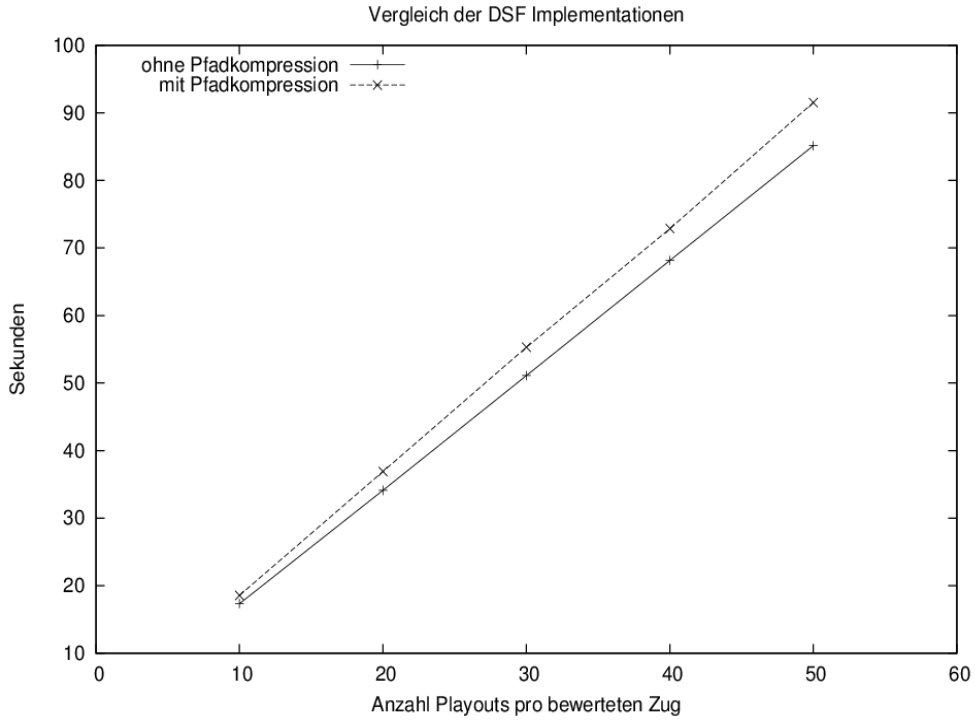

Abbildung 10: Ergebnis von Experiment 3

nahe an der Laufzeit ohne Pfadkompression.

Experiment 4:

Im vierten Experiment soll untersucht werden wie viele Blöcke mit jeweils einem Thread für die Runner möglich sind und inwiefern sich die Erhöhung der Threads auf die Laufzeit auswirkt. Für die Ausführung wurde der Runner ohne Pfadkompression (rateGame_npc) verwendet. Die Anzahl der auszuführenden Playouts liegt konstant bei zehn pro bewerteten Zug (2710 Playouts). Tabelle 1 zeigt die Ergebnisse.

Anzahl an Blöcken mit einem Thread	Laufzeit in Sekunden
8	17,34
16	9,26
32	5,49
64	3,38
96	2,23
128	2,06
256	2,03

Tabelle 1: Ergebnis von Experiment 4

Im ersten Schritt kann eine Verdopplung der Threads von 8 auf 16 auch fast eine Halbierung (Faktor 0,53) der Laufzeit erreichen. Eine weitere Verdopplung der Threads (32) erreicht immerhin noch eine Verbesserung um den Faktor 0,59. Auch die nächste beiden Verdopplung der Threads auf 64 bzw. 128 Threads verbessern die Laufzeit um den Faktor 0,6. Die nächste Verbesserung fällt mit einem Faktor von 0,99 kaum noch ins Gewicht.

Experiment 5:

Da die Anzahl von 10 Playouts pro bewerteten Zug aus Experiment zu niedrig war um bei höheren Thread Zahlen eine Verbesserung zu erzielen, soll in diesem Experiment überprüft werden wie es sich bei einer Anzahl von 100 Playouts pro bewerteten Zug verhält. Als Vergleichszeit wird die Ausführung des CPU Codes ohne Parallelisierung und mit Pfadkompression auf 100 Playouts pro bewerteten Zug verwendet (27100 Playouts): 96,56 Sekunden. Die Ergebnisse sind in Tabelle 2 dargestellt.

	Laufzeit in Sekunden	Speedup
128 Threads	18,67	5,17x
256 Threads	17,19	5,62x
512 Threads	12,19	7,92x

Tabelle 2: Ergebnisse von Experiment 5

Experiment 6:

In diesem Experiment soll festgestellt werden, welcher Zug der beste Eröffnungszug ist. Dafür wird auf einem leeren Brett für alle Züge je 10000 Playouts durchgeführt (2710000 Playouts). Um den Einfluss der Pseudozahlenerzeugung zu verringern wird der Vorgang zweimal durchgeführt. Das Ergebnis ist in Abbildung 11 zu sehen. Für die Visualisierung wurde folgendes Vorgehen verwendet: Es wurden pro Anfangszug die Anzahl der gewonnenen Spiele ermittelt. Diese Zahlen wurden mit dem Minimum an gewonnenen Spielen subtrahiert. Für die Grauwertdarstellung wurden die neuen Werte in den Zahlenraum von 0-100 überführt, wobei 0 den höchsten und 100 den niedrigsten Wert darstellt.

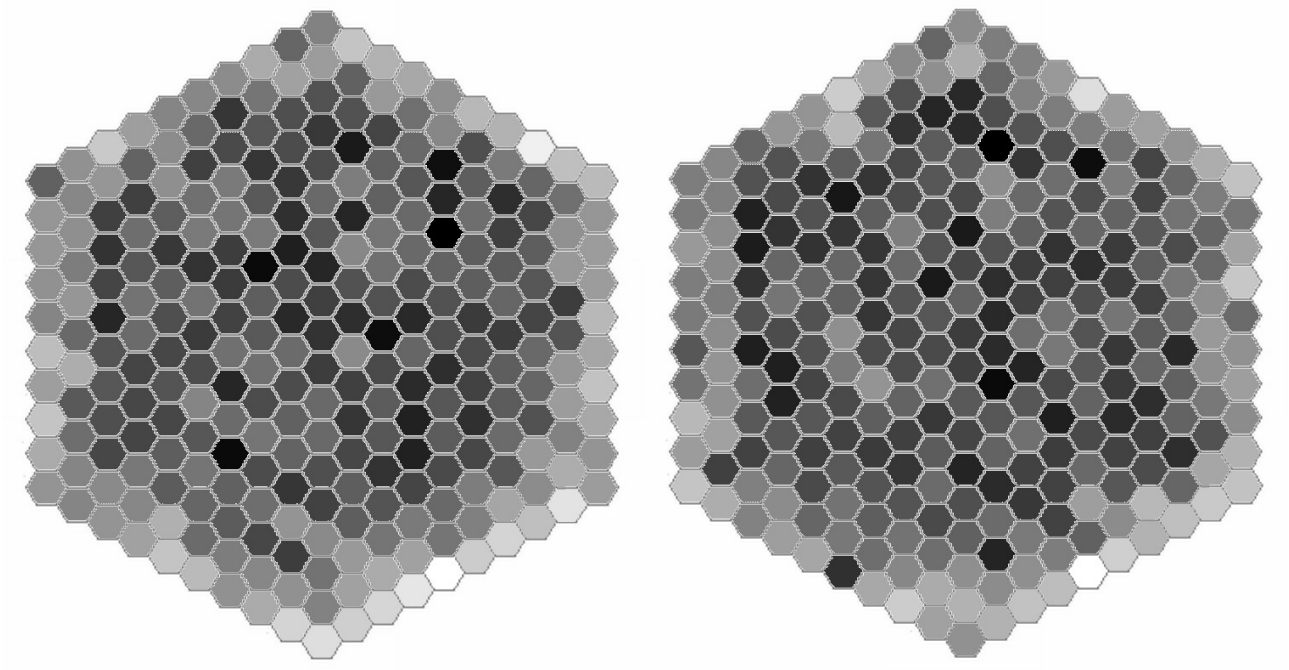

Abbildung 11: Finden des besten Eröffnungszugs

Hinter dem linken Bild stehen folgende Zahlen:

- Minimale Anzahl an gewonnenen Spielen: 4893 (Anfangszug auf Feld mit Index 5)
- Höchste Anzahl an gewonnenen Spielen: 5271 (Anfangszug auf Feld mit 159)

Für das rechte Bild gilt:

- Minimale Anzahl an gewonnenen Spielen: 4846 (Anfangszug auf Feld mit Index 5)
- Höchste Anzahl an gewonnenen Spielen: 5254 (Anfangszug auf Feld mit 222)

Im folgenden sollen nun die gesamt Ergebnisse betrachtet werden.

	Index	Anzahl gewonnener Spiele (bei 20000 Playouts)	Gewinnwahrscheinlichkeit
Bester Zug	222	10481	52%
Schlechtester Zug	5	9739	49%

Tabelle 3: Gegenüberstellung bester und schlechtester Eröffnungszug

Aus den Daten in Tabelle 3 kann man erkennen, dass es bei 20000 Playouts keinen klaren Sieger gibt. Die Gewinnwahrscheinlichkeiten liegen nur 3% auseinander, was bei zufallsbestimmten Experimenten keine all zu große Differenz ist. Dennoch sieht man auf den Bildern von Abbildung 10, dass man sich eher auf die Mitte des Brettes konzentrieren sollte.

Experiment 7:

Nachdem in Experiment ein bester Zug gefunden wurde, soll jetzt im Experiment herausgefunden werden, wie man auf diesen Zug am geschicktesten reagiert. Dafür wird auf einem Brett der Zug mit dem Index 222 durchgeführt und für alle freien Züge 20000 Playouts berechnet (also insgesamt 5400000 Playouts). Das Ergebnis ist in Abbildung 12 dargestellt. Dabei wurde die Visualisierung die in Experiment 6 beschrieben ist verwendet.

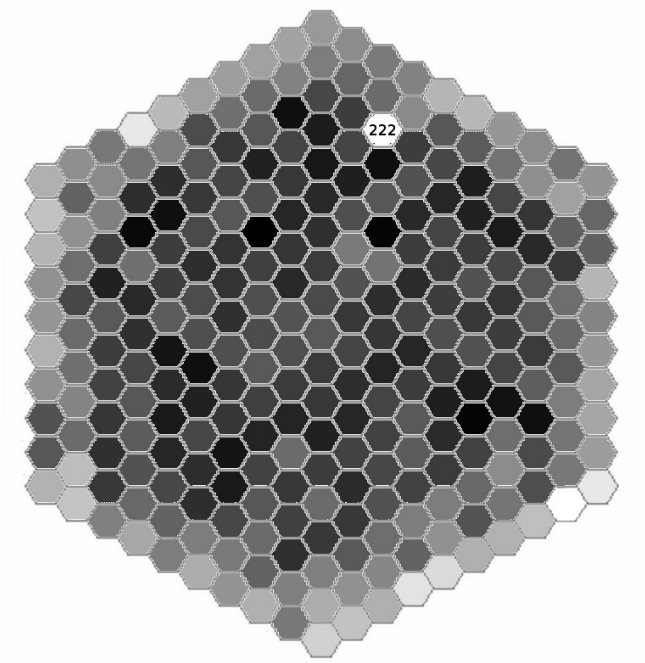

Abbildung 12: Berechnung eines Konters für den Anfangszug auf Feld 222

In Tabelle 4 sind die zugrunde liegenden Zahlen für die minimale und maximale Gewinnwahrscheinlichkeit dargestellt.

	Index	Anzahl gewonnener Spiele (bei 20000 Playouts)	Gewinnwahrscheinlichkeit
Bester Zug	203	10011	50%
Schlechtester Zug	8	9299	46%

Tabelle 4: Gegenüberstellung bester und schlechtester Eröffnungszug

Bei der Auswertung der Gewinnwahrscheinlichkeit fällt auf, dass nur ein Zug eine Gewinnwahrscheinlichkeit von über 50% hat. Somit kann die Behauptung eines besten Zuges aus Experiment 6 zumindest nicht widerlegt werden.

Fazit

Die Arbeit hat erste Ansätze für die Parallelisierung von MC-Playouts für das Brettspiel Havannah geschaffen. Die Experimente zeigen bereits jetzt, dass ein Speedup möglich ist. Jedoch ist noch weitere Arbeit nötig um die Ausführung zu Optimieren. Die nächsten Ziele wären die Speicherzugriffe auf dem Device zusammenfassen und die Ablage der Variablen in die verschiedene Speicherbereiche zu optimieren. Des weiteren sollte eine Optimierung des Havannah Spielbretts und dessen Auswertung in Betracht gezogen werden. Konkrete Verbesserungen der Implementierung anhand aktueller Forschung wie z.B. Castro[5] das von Timo Ewalds im Rahmen seiner Masterarbeit[4] entwickelt wurde. Da in Castro ebenfalls eine Art Playout vorkommt, könnte man versuchen die hier gewonnen Erkenntnisse zu übertragen. Ein weiterer Anhaltspunkt liefert die Masterarbeit von J.A. Stankiewicz [6]. Hinweise auf Schwachstellen in der Implementierung könnte auch die Definitionen von Frans Fasse[3] liefern.

Quellen & Referenzen

[1] Christian Freeling, „*Havannah*“, 2012, `http://mindsports.nl/index.php/arena/havannah/`,abgerufen am 16.02.2012
[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, „*Intriduction to Algorithms*“, 1991, The MIT Press, 3.Auflage, S.446-S.450
[3] Frans Faase, „*Formalisation of the game Havannah*“, `http://www.iwriteiam.nl/Havannah.html`, abgerufen am 16.02.2012
[4] Timo Ewalds, „*Playing and Solving Havannah*“, Masterarbeit, University of Alberta, 2012
[5] Timo Ewalds, „*Castro*“, `https://github.com/tewalds/castro/`
[6] J.A. Stankiewicz, „*KNOWLEDGE-BASED MONTE-CARLO TREE SEARCH IN HAVANNAH*“, Masterarbeit, Maastricht University, 2011
[7] J. D. Fossel, „*Monte-Carlo Tree Search Applied to the Game of Havannah*“, 2010,
[8] „*Monte Carlo Tree Search*“, `http://senseis.xmp.net/?MonteCarlo`, abgerufen am 16.02.2012
[9] Guillaume Chaslot, Sander Bakkes, Istvan Szita and Pieter Spronck, „*Monte-Carlo Tree Search: A New Framework for Game AI*“, Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, 2008, S.216-S.217
[10] „*NVIDIA CUDA C Programming Guide*“, Version 4.0, 2011
[11] Stefan Maskanitz, „*Grundlagen von CUDA*“, Sprachtypische Elemente, 2009
[12] Richard Membarth, „*CUDA Parallel Programming Tutorial*“, 2009

Lizenz Hinweis

Der nachfolgende Hinweis gilt für jeglichen Teil des innerhalb dieser Arbeit erzeugten Quellcodes.