Simulation und Modellierung Projektbericht

Florian Franz

Potsdam Universität

Abstract. Dieser Projektbericht ist Teil des Simulation und Modellierung Kurses. Das Projekt besteht aus der Implementierung von John Conways's Game of Life mit Python. Zusätzliche Funktionen sind beispielsweise das Importieren von Mustern und ein individuell konfigurierbarer Video Export. In diesem Paper finden Sie eine Zusammenfassung des Projekts mit Implementation, Konzepten und Benchmarks.

Keywords: Game of Life · Simulation

1 John Conway's Game of Life

1.1 Erklärung

John Conway's Game of Life lässt sich als Simulation oder auch Null Spieler Spiel bezeichnen. Im Kern geht es um eine Matrix von Zellen, welche an einem Zeitpunkt am Leben oder tot sein können. In jedem Zeitschritt werden die Zellen wieder neu berechnet. Die Regeln sind wie folgt:

- Jede lebendige Zelle mit nur einem oder keinem Nachbar stirbt aus Einsamkeit
- Jede lebendige Zelle mit vier oder mehr Nachbarn stirbt wegen Überbevölkerung
- Jede lebendige Zelle mit zwei oder drei Nachbarn überlebt
- Jede tote Zelle mit genau 3 Nachbarn wird wiederbelebt

1.2 Spielfeld

In diesem Projekt wird eine zwei-dimensionale Variante mit Torus Form implementiert. Die Torus Form führt dazu, dass das Spielfeld prinzipiell keinen Rand hat. Zellen am rechten Rand sind Nachbarn von Zellen am linken Rand. Genauso sind Zellen am unteren Rand Nachbarn vom oberen Rand. Alternativ kann man das Spielfeld durch tote Zellen am Rand begrenzen die ihren Zustand nicht verändern. Beide Ansätze haben Vor- und Nachteile und können die unterschiedlichen Generationen von Zellen verändern.Neben dem 2D Spielfeld gibt auch eine Abwandlung vom Game of Life in 3D, welche hier aber nicht betrachtet wird.

1.3 Herkunft und Entwicklung

Wie dem Titel bereits zu entnehmen ist, handelt sich Jon Conway's Game of Life um ein Entwurf des britischen Mathematikers John Horton Conway. Conway forschte primär in der kombinatorischen Spieletheorie wodurch auch 1970 die Idee zum Game of Life kam. Über die Jahre entwickelte sich um diese einfache Spielidee eine große Community. Es wurden sehr viele spannende Muster gefunden. Weiterhin wurde bewiesen, dass Game of Life turingmächtig ist. Das Game of Life kann also alle Berechnungen ausführen, die man sich vorstellen kann.

2 Architektur

2.1 Frameworks und Bibliotheken

Für die Implementation wurde die Programmiersprache Python genutzt. Für die grafische Darstellung und interaktive Funktionen wird die Spielebibliothek PyGame genutzt. Für die grafische Benutzeröberfläche wird PyGame GUI genutzt. Für einige Berechnung wird Numpy genutzt. Das Erstellen der Bilder für den Video Export erfolgt mit Hilfe von Python Pillow. Im Benchmarking wird matplotlib zum Erstellen von Diagrammen genutzt. Die CUDA Parallelisierung wird mit Hilfe von Numba[**?**] ermöglicht.

2.2 Projektstruktur

Das Projekt ist aufgeteilt in unterschiedliche Klassen. Zum einen gibt es die Klasse GameLoop. GameLoop kümmert sich darum in regelmäßigen Abständen das Spielfeld und GUI zu aktualisieren, User Input engegen zu nehmen und sonstige Koordinationen zu übernehmen.

Die Klasse Game kümmert sich um alle Funktionen die konkret das Spielfeld modifizieren. Das beinhaltet zum einen die Berechnung eines Update Schritts und auch die Zeichnung des Spielfelds.

Die Control Klasse enthält alle Einstellungen die ein Benutzer tätigen kann. Zum einen sind dort alle Standardeinstellungen hinterlegt, zum anderen werden alle getätigten Einstellungen zur Laufzeit hier abgespeichert. Neben Benutzereinstellungen gibt es hier Einstellungen zur Flußkontrolle.

UIElements ist eine Klasse die ausschließlich dazu da ist GUI Elemente wie Buttons, Textfelder oder Modale zu verwalten. Dort wird festgelegt wie die Elemente aussehen und wie sie positioniert sind. Desweiteren wird hier festgelegt was nach einem bestimmten GUI Event passiert. Beispielsweise wenn ein Button gedrückt wird oder ein Slider sich verändert

Die letzte Klasse ist die VideoExport Klasse, hierbei geht es ausschließlich darum erst Bilder zu jeder Generation von Zellen zu speichern und anschließend daraus ein Video zu produzieren.

Bisher nicht erwähnt ist das rleconverter Modul. Dieses enthält Funktionen zum Speichern und Laden von Mustern.

3 Simulation

Die Simulation besteht aus dem iterativen Aktualisieren aller Zellen. Ein Aktualisierungsschritt berechnet für jede Zelle die Anzahl der Nachbarn und bestimmt basierend darauf ob die Zelle in der neuen Generation am Leben oder tot ist. Um die Nachbarzellen zu bestimmen wird der Rest von Position geteilt durch die Größe genommen. Das ermöglicht eine Torus Struktur. Die Zellen am rechten Rand sind Nachbarn mit den Zellen am linken Rand etc.. Die iterative Variante ohne Parallelisierung in im folgenden Code-Beispiel zu sehen.

Listing 1.1. Identifying conflict on original plan

```
# columns
sizex = self.ctrl.cellsx
# rows
sizey = self.ctrl.cellsy
# start with a bunch of dead cells (alive = 1, dead=0)
nxt = np.zeros((sizey, sizex))
```

```
for r, c in np.ndindex((sizey, sizex)):
    # cell above
    rm1mod = (r - 1) % sizey
    # cell left
    cm1mod = (c - 1) % sizex
    # cell below
    r1mod = (r + 1) % sizey
    # cell right
    c1mod = (c + 1) % sizex
    num_alive = self.cells[r, cm1mod] + self.cells[rm1mod, c] +\
                self.cells[rm1mod, cm1mod] + self.cells[r1mod, c1mod] +\
                self.cells[r1mod, c] + self.cells[r1mod, cm1mod] +\
                self.cells[r, c1mod] + self.cells[rm1mod, c1mod]
    if (self.cells[r, c] == 1 and 2 <= num_alive <= 3) \
            or (self.cells[r, c] == 0 and num_alive == 3):
        nxt[r, c] = 1
self.cells = nxt
```

4 GUI

In der GUI gibt es verschiedene Abschnitte. Es gibt die Hauptansicht, das Import Modal, das Video Export Einstellungen Modal und das Video Export Durchgang Modal.

4.1 Hauptansicht

Die Hauptansicht hat 7 unterschiedliche Aktionen. In Fig. 1 und Fig. 2 lässt sich sehen wie diese Aktionen in der GUI ausgelöst werden.

- Zellbearbeitung an/aus schalten
- Zwischen CUDA GPU und CPU wechseln
- Video Export Modal öffnen
- Muster Import Modal öffnen
- Spielfeldgröße ändern
- Simulation starten/stoppen
- Geschwindigkeit der Simulation anpassen

Fig. 1. Hauptseite GUI Teil 1

Fig. 2. Hauptseite GUI Teil 2

4.2 Import Modal

Das Import Modal erlaubt die Eingabe einer .rle Datei aus dem Subordner "patterns/". Mit Klick auf den Import Knopf wird das Muster in der Datei geladen und ersetzt das alte. In Fig. 3 lässt sich sehen wie das Import Modal aussieht.

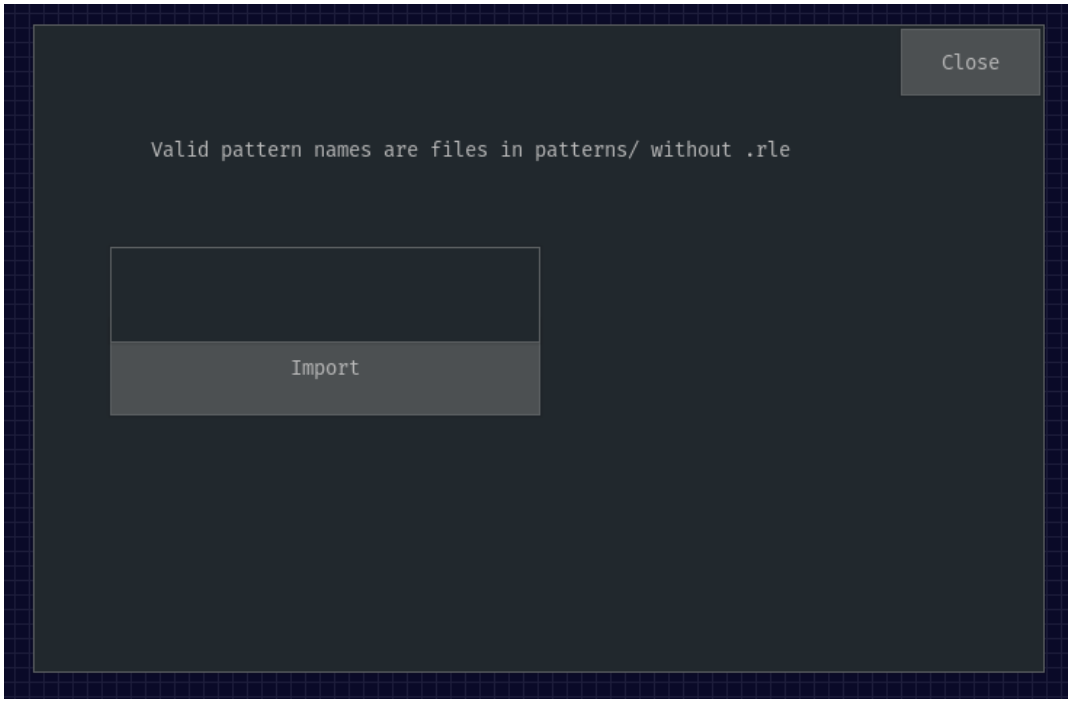

Fig. 3. Import Modal

4.3 Video Export Modal

Das Video Export Modal besteht aus der Einstellungseite Fig. 4 und der Fortschrittseite Fig. 5. Auf der Einstellungseite wird der Export vorbereitet und man kann folgende Parameter anpassen:

- Anzahl der Zeitschritte die berechnet werden
- Farbe der lebendigen Zellen als Hexadezimalcode
- Farbe der toten Zellen als Hexadezimalcode
- Zeitschritte pro Sekunde im Video
- Gewollte Videohöhe (Für ein 1080p Video würde man 1080 eintragen)

Während der Export im Hintergrund läuft wird die Fortschrittseite angezeigt. Prozentualer Fortschritt und eine Schätzung der verbleibenden Zeit wird dort angezeigt. Weiterhin hat man die Möglichkeit den Export abzubrechen.

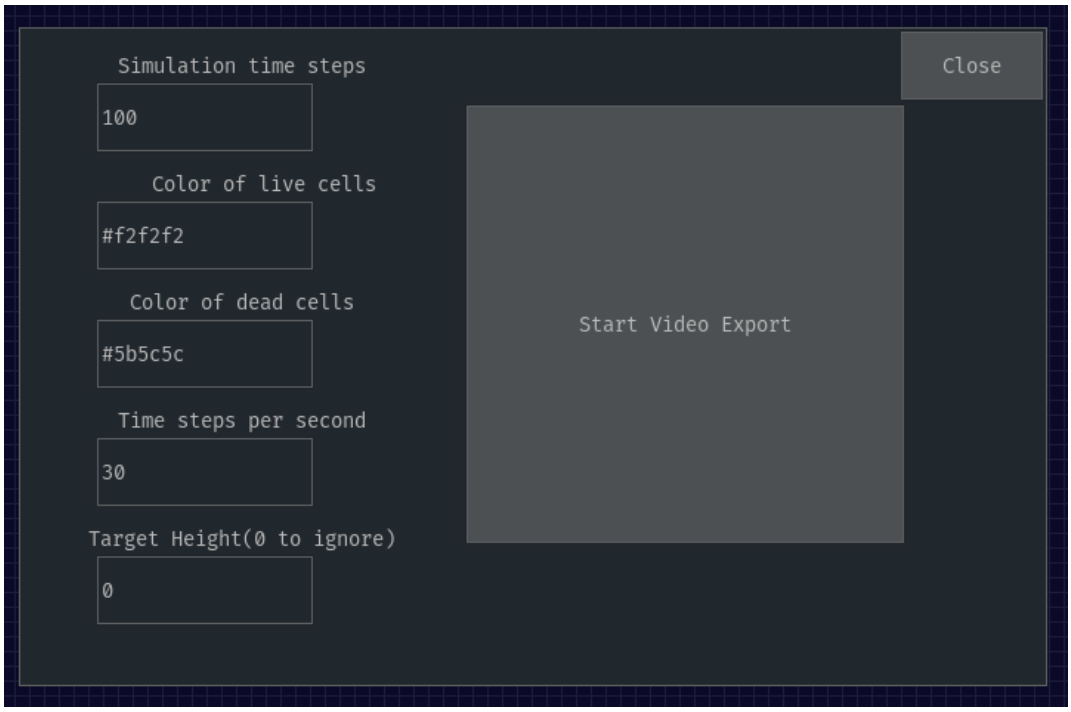

Fig. 4. Import Modal

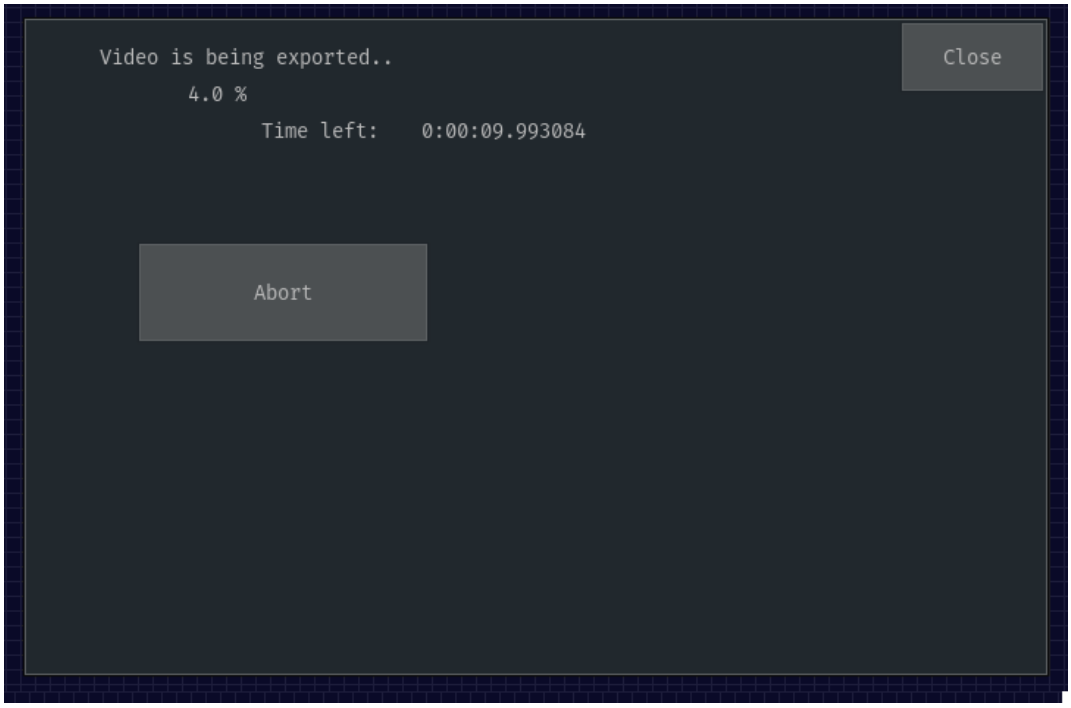

Fig. 5. Import Modal

5 Muster Import

Der Muster Import lädt aus der gewünschten Datei ein Run Length Encoding heraus. Dieses wird als Matrix von toten bzw. lebendigen Zellen interpretiert und ersetzt damit die alte Matrix. Im nächsten Abschnitt wird das Run Length Encoding kurz erklärt.

5.1 Run Length Encoding

Zuerst ein Beispiel eines RLE:

Listing 1.2. Identifying conflict on original plan

```
x = 3, y = 3
bo$2bo$3o!
```

Das Beispiel zeigt ein Encoding für einen Glider. Das ist eine sich ständig fortbewegende Gruppe von lebenden Zellen insofern sie auf keine anderen lebendigen Zellen treffen. In Fig. 5 lässt sich der Glider als Matrix sehen.

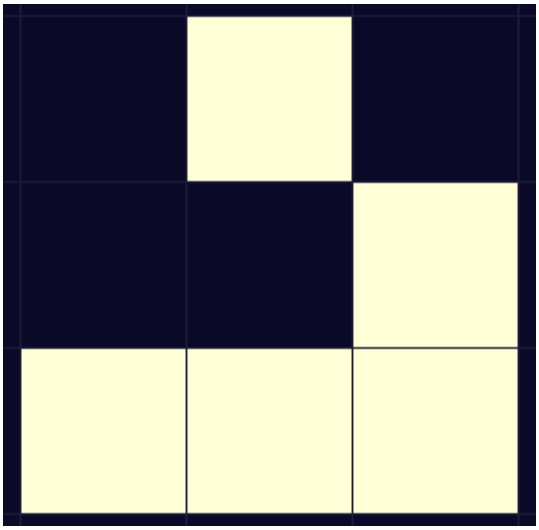

Fig. 6. Import Modal

Die ersten Zeilen der RLE Datei sind in der Regel Kommentare welche das Muster beschreiben. Die erste richtige Zeile enthält die Größe des Musters. Dabei steht x für die Spalten und y für die Zeilen. Nach der ersten Zeile fängt die Definition der Daten an. Dabei gibt es unterschiedliche wichtige Zeichen:

- b = tote Zelle
- o = lebendige Zelle
- $ = Zeile Ende
- ! = Matrix Ende
- (Beispiel)2bo = 2 tote Zellen und danach eine lebendige Zelle
- (Beispiel)3o = 3 lebedingie Zellen hintereinander

Zwischen den $ Zeichen befindet sich eine Zeile von links nach rechts. Alle Zeichen die nicht explizit definiert werden sind tote Zellen. Im Beispiel trifft das auf die erste Zeile der Matrix zu.

6 Video Export

Der Video Export erstellt aus mehreren Zeitschritten des Game of Life ein Video. Die Simulation läuft direkt auf der aktiven Matrix. Während des Exports wird die Anzeige in der Anwendung deaktiviert. Für den Video Export wird jede Population von Zellen als Bild mit gewünschter Auflösung abgespeichert. Der Benutzer bekommt während dem Export eine Anzeige wie lange der Export noch dauert. Dieser wird wie folgt berechnet:

$$(self.requiredsteps - stepstaken) * np.average(times) \tag{1}$$

Das "times" Array besteht hier aus den Ausführungszeiten der letzten 10 Zeitschritte. Somit entsteht eine realistische Schätzung basierend darauf wie schnell das Erstellen der Bilder in den letzten 10 Zeitschritten war. Wenn alle Bilder generiert wurden werden sie in ein Video mit den gewünschten Parametern gepackt.

7 Parallelisierung

Die erste Implementation auf nur einem CPU Thread war sehr langsam. Daher hat das Programm eine parallele Verarbeitung zum einen auf der CPU und optional auf einer CUDA GPU.

7.1 CPU

Die Idee der Parallelisierung auf der CPU beruht darauf jedem Thread nur eine bestimmte Anzahl von Zeilen zu geben die er verarbeiten muss. Es wird gewartet bis alle Threads fertig sind und die zusammengesetzten Zeilen sind dann das neue Spielfeld.

Listing 1.3. Identifying conflict on original plan

```
sizex = self.ctrl.cellsx
sizey = self.ctrl.cellsy
threadList = []
unfitted = sizey % threads
end = 0
results = np.zeros((sizey, sizex))
for i in range(threads):
    prevend = end
    end = min(prevend + (sizey // threads), sizey)
    if unfitted > 0:
        end = end + 1
        unfitted = unfitted -1
    threadList.append(
        threading.Thread(target=self.do_update_multicore,\
        args=[prevend, end, results]))
for thread in threadList:
    thread.start()
for i in range(len(threadList)):
    threadList[i].join()
self.cells = results
```

Die Funktion self.do_update_multicore die jeder Thread ausführt führt den selben Code aus wie die originale Implementierung auf einem Thread. Der Unterschied ist, dass nur noch eine bestimmte Spanne von Zeilen bearbeitet wird. Weiterhin kann es sein, dass die Anzahl der Zeilen nicht durch die Anzahl der Threads teilen lässt. In dem Fall gibt es einen Rest der hier auf die ersten paar Threads aufgeteilt wird.

7.2 CUDA GPU

Die CUDA Implementation setzt auf ein Framework namens "numba". Dieses Framework übernimmt den Großteil der Arbeit beim parallelisieren auf der GPU.

Listing 1.4. Identifying conflict on original plan

```
sizey = self.ctrl.cellsy
sizex = self.ctrl.cellsx
results = np.zeros((sizey, sizex))
threadsperblock = (128, 128)
blockspergrid_x = math.ceil(self.cells.shape[0] / \
threadsperblock[0])
blockspergrid_y = math.ceil(self.cells.shape[1] / \
threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
do_update_gpu[threadsperblock, blockspergrid](self.cells,\
results,sizex, sizey)
self.cells = results
```

In CUDA wird die Verarbeitung in Blöcke aufgeteilt. In dem obigen Code Snippet wird die Anzahl der Threads pro Dimension auf 128 gesetzt. Die Anzahl der Blöcke pro Dimension ist die Breite bzw. Höhe der Matrix geteilt durch die Anzahl der Threads. Die Funktion do_update_gpu ist an sich genauso wie in der Einzelthread Implementation. Der Unterschied ist, dass nicht alle Zellen durchgegangen werden sondern nur Code zur Bearbeitung einer einzigen Zelle hinterlegt ist. Die restliche Arbeit wird von Numba übernommen.

8 Benchmarks

8.1 Benchmark Script

Das Benchmark Script startet die Game of Life Implementation und erstellt Statistiken mit den unterschiedlichen Parallelisierungsarten. Die fertigen Benchmarks werden im "benchmarks" Ordner gespeichert. Das Benchmark Script speichert aktuell individuelle Graphen und die konkreten Daten pro Verarbeitungsmethode. Zum einen wird verglichen wie die Parallelisierung zueinander abschneiden. Desweiteren wird der Video Export mit dem puren Anzeigen in der GUI verglichen. Für jede Methode geht das Benchmarking Script 3 Beispielmuster mit unterschiedlichen Größen durch. Es werden genau 100 Zeitschritte berechnet.

8.2 Ergebnisse

Die folgenden Ergebnisse stammen ausschließlich von meinem Persönlichen Computer. Im Fall des Video Exports ist der limitierende Faktor die Schreibgeschwindigkeit der Festplatte beim Speichern der einzelnen Bilder. Hier eine Liste der relevanten Spezifikationen:

- GPU = Geforce RTX 2060
- CPU = AMD Ryzen 5 3500X 6-Core Processor 3.60 GHz
- Festplatte = Kingston SA2000M8500G SSD

Die Vermutung ist, dass die Parallelisierung mit mehreren CPU threads einen klaren Performance Anstieg zeigt. Mit CUDA sollte dieser Effekt noch einmal wesentlich größer sein. Der Effekt sollte umso größer sein desto größer die Matrizen werden. Getestet wurden 3 Beispiele mit unterschiedlichen Größen. Die konkrete Belegung der Zellen sollte keinen großen Effekt haben.

- default.rle : Größe 20 x 20
- glidergunsmall.rle : Größe 200 x 30
- glidergunbig.rle : Größe 1000 x 200

Hier sind die Ergebnisse(Im Anhang sind auch noch die gezeichneten Diagramme zu finden):

Table 1. Ein Thread Normal

Zeit	Instanz
0.11210179328918457	default.rle
109.6065411567688	glidergunbig.rle
1.5839440822601318	glidergunsmall.rle

Table 2. Ein Thread Video

Zeit	Instanz
4.729872941970825	default.rle
94.80739784240723	glidergunbig.rle
30.736952543258667	glidergunsmall.rle

Table 3. 8 Threads Normal

Zeit	Instanz
0.1511368751525879	default.rle
47.88955116271973	glidergunbig.rle
1.4893536567687988	glidergunsmall.rle

Table 4. 8 Threads Video

Zeit	Instanz
4.58716893196106	default.rle
89.62700319290161	glidergunbig.rle
31.624256372451782	glidergunsmall.rle

Table 5. CUDA GPU Normal

Zeit	Instanz
0.7987260818481445	default.rle
0.24322152137756348	glidergunbig.rle
0.09008145332336426	glidergunsmall.rle

Table 6. CUDA GPU Video

Zeit	Instanz
4.675249099731445	default.rle
42.302467823028564	glidergunbig.rle
30.12489414215088	glidergunsmall.rle

Insgesamt wurden die Vermutungen über die Veränderung der Laufzeit nach Art bestätigt. In dem größten Muster "glidergunbig.rle" ohne Video braucht ein Thread 109,61 Sekunden. Acht Threads sind 228,88% schneller. Die CUDA Berechnung ist nochmal 19689,82 % schneller mit nur 0,24 Sekunden Ausührungszeit. Im Gegensatz zum Größten Muster braucht das kleinste Muster "default.rle" mit nur einem Thread am wenigsten Zeit. Die CUDA Implementation braucht dafür am längsten. Das Mittelgroße Muster "glidergunsmall.rle" zeigt den selben Effekt wie das größte Muster, ist aber weniger ausgeprägt.

Der Video Export dauert wesentlich länger als das Berechnen für die GUI. Außer bei einem einzigen Thread in dem größten Muster. Am extremsten ist der Anstieg der Laufzeit mit GPU auf der größten Instanz mit 17392,67%. Diese Diskrepanz kommt durch das Speichern und Laden der Bild- und Videodateien zustande. Größere Instanzen haben erhöhten Overhead für das Speichern da die Auflösung mit der Spielfeldgröße skaliert.

Insgesamt hat die Parallelisierung mit der CPU die Laufzeit verbessert. Die Parallelisierung mit der GPU macht große Sprünge und ist wesentlich schneller als andere Implementationen.

9 Fazit

Insgesamt sind fast alle Funktionen aus dem initialen Plan umgesetzt und funktionieren sehr gut. Eine neue Funktion die nicht vorgesehen war ist die Parallelisierung. Jedoch hat sich das aufgrund der langsamen sequentiellen Zeiten sehr gelohnt. Schwachpunkte stellen die unneinheitliche GUI, der unaufgeräumte Code dar. Letzteres führt dazu, dass neue Features länger zum entwickeln brauchen. Eine Funktion aus dem initialen Plan die nicht implementiert wurde ist der Export von Mustern. Die Funktion hierfür ist weitgehend fertig jedoch gibt es dafür keine GUI.

Die größte Hürde in der Entwicklung des Projekts stellte das GUI Framework Pygame GUI dar. Es stellt zwar eine grundlegende Möglichkeit dar um Buttons, Textfelder etc. zu erstellen jedoch fehlen einige Features aus gewöhnlich GUI Frameworks. Um sauberen Code und eine leichte erweiterbarkeit zu erreichen wäre es hilfreich ein kleines Framework on top zu schreiben was sich selbst um prozentuale Abstände, Modale und einfachere Erstellung von neuen Komponenten kümmert.Im Großen und Ganzen war das Projekt aber sehr spannend. Dadurch, dass die Basis, also das Game of Life, sehr simpel ist lassen sich andere Funktionen sehr gut darauf aufbauen.

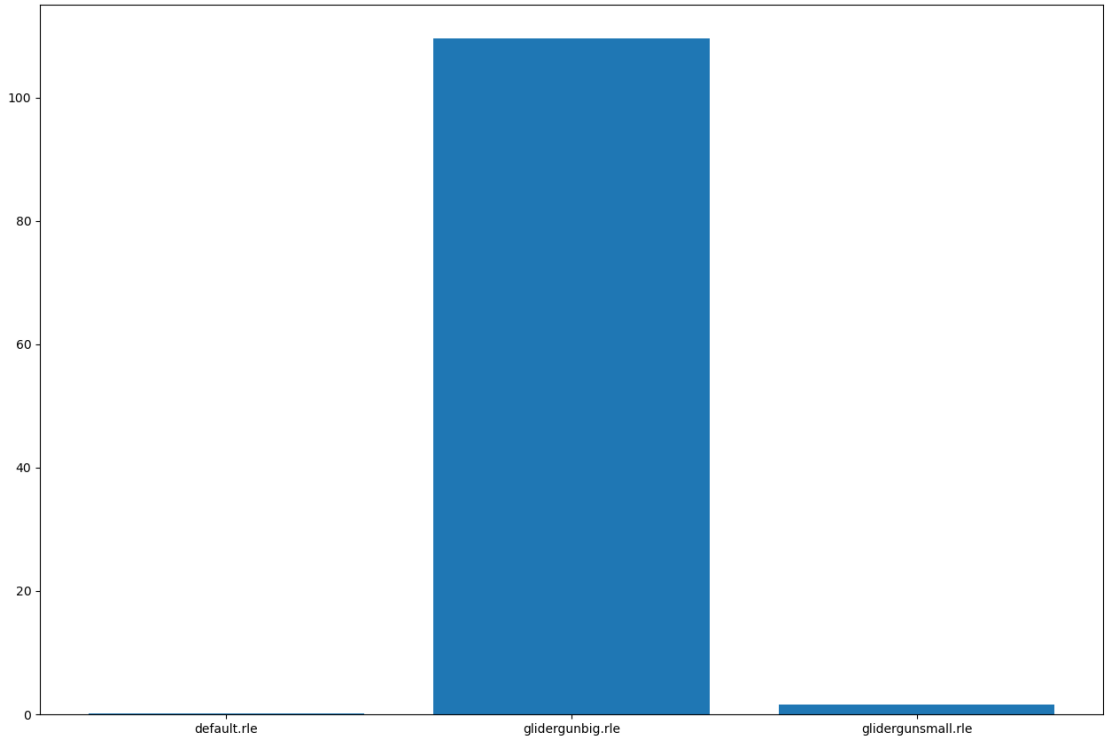

Fig. 7. Ein Thread

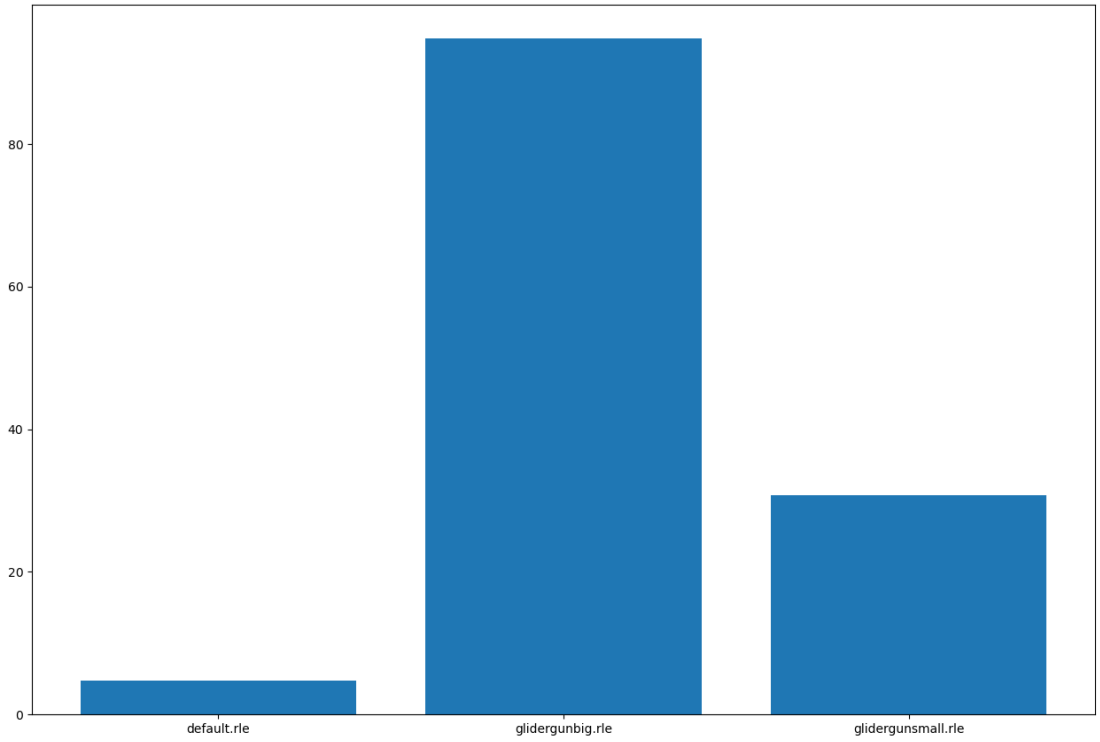

Fig. 8. Ein Thread Video Export

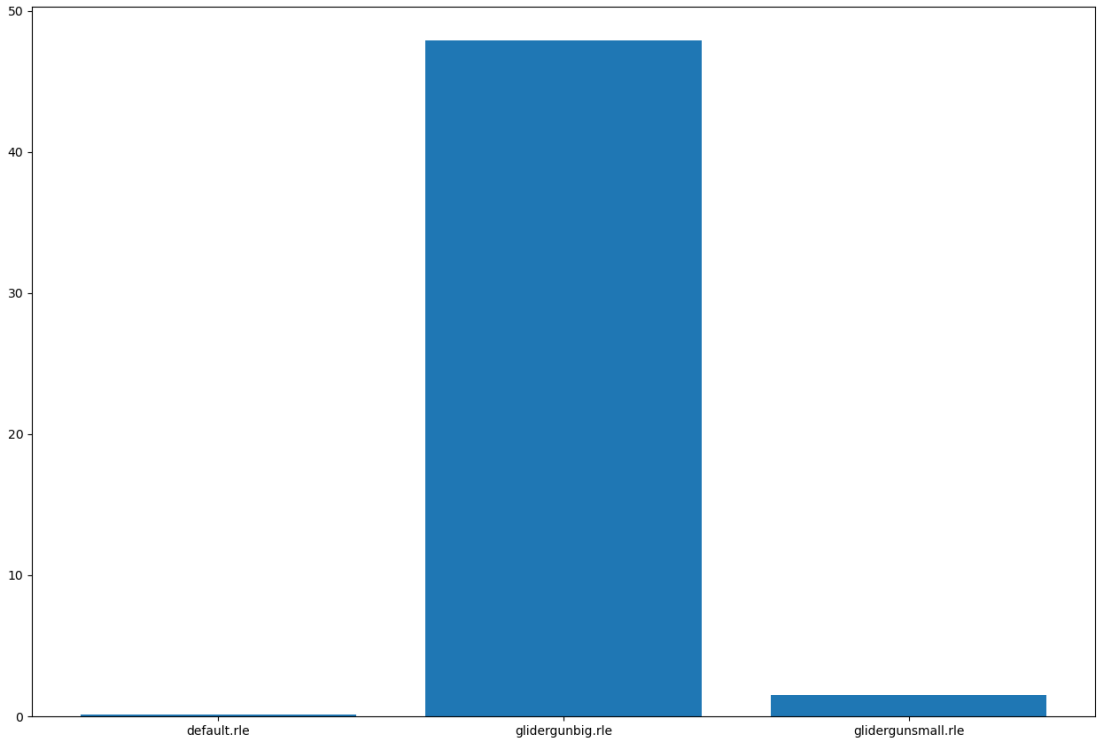

Fig. 9. 8 Threads

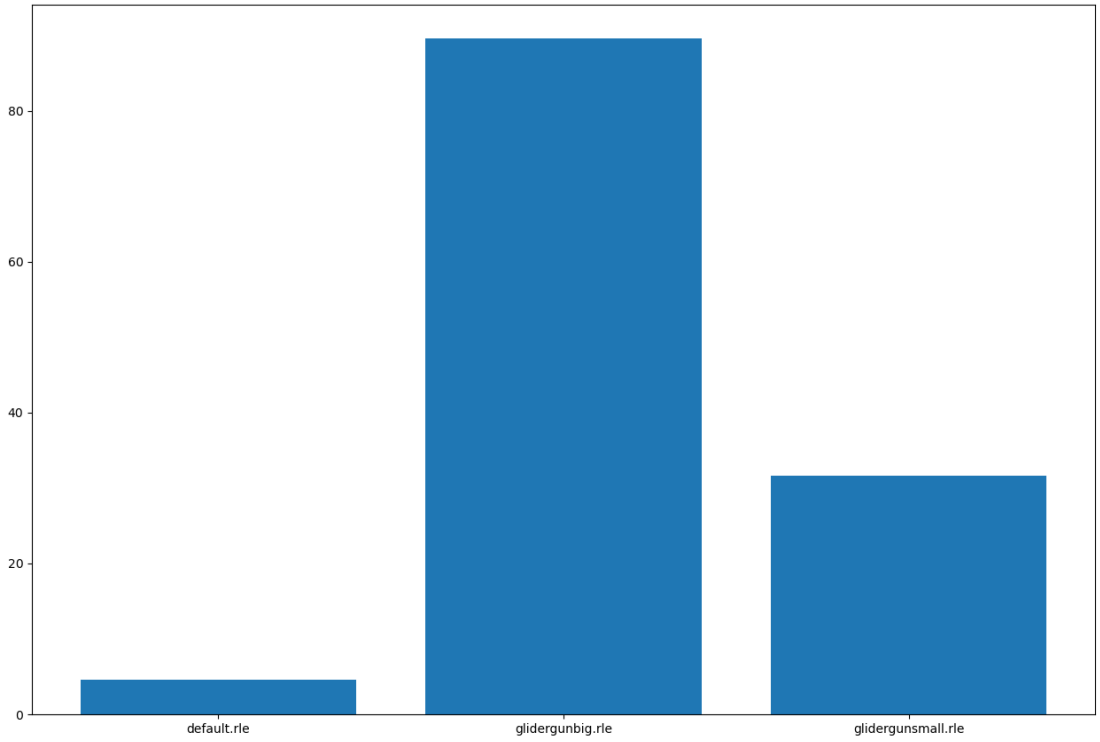

Fig. 10. 8 Threads Video Export

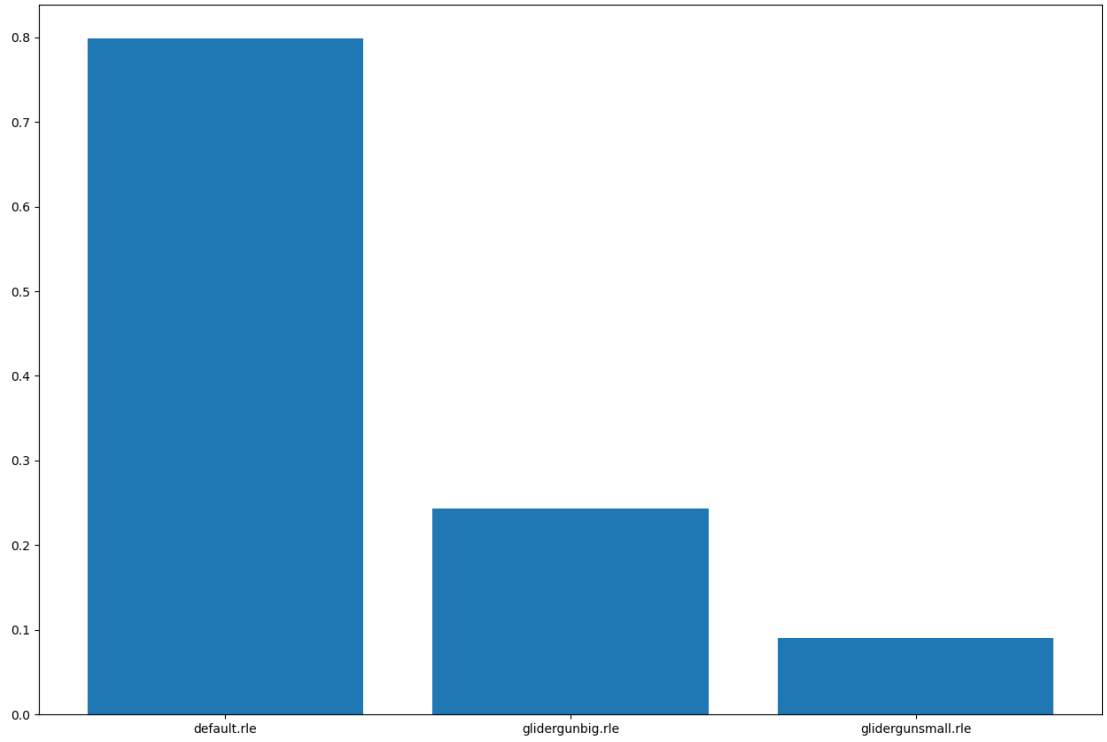

Fig. 11. CUDA GPU

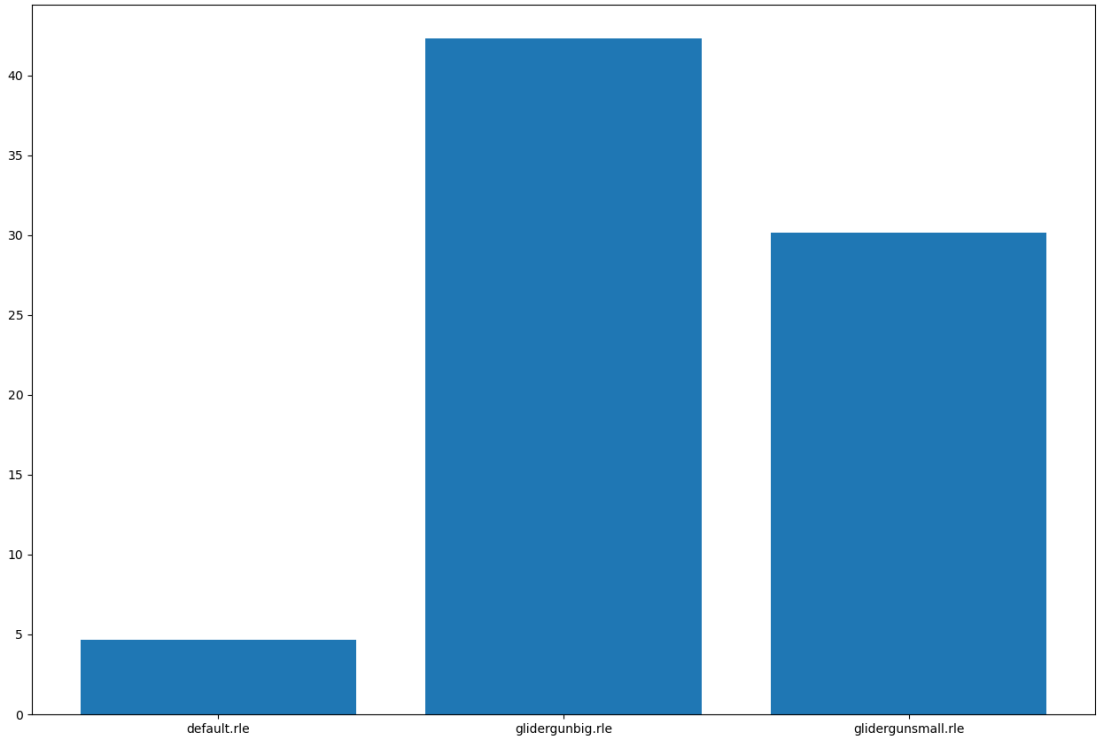

Fig. 12. CUDA GPU Video Export