TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR RECHNERARCHITEKTUR
PROF. DR. WOLFGANG E. NAGEL

# Komplexpraktikum Paralleles Rechnen - Aufgabe C

Daniel Körsten

Dresden, 7. Februar 2022

## Inhaltsverzeichnis

# 1 Aufgabenbeschreibung

In dieser Aufgabe soll eine SIMD-parallele Version von `Conway's Game-of-Life` in der Programmiersprache `C` implementiert werden.
Anschließend soll die Simulation mit verschieden großen Feldgrößen und Compilern durchgeführt und verglichen werden.

## 1.1 Conway's Game-of-Life

Das Game-of-Life ist ein vom Mathematiker John Horton Conway entworfenes Simulationsspiel [Gar70]. Es basiert auf einem zellulären Automaten. Häufig handelt es sich um ein zweidimensionales Spielfeld, jedoch ist auch eine dreidimensionale Simulation möglich.
Das Spiel besteht dabei aus einem Feld mit einer festgelegten, möglichst großen, Anzahl an Zeilen und Spalten. Eine Zelle kann dabei entweder Tot oder Lebendig sein. Dieses Spielfeld wird mit einer zufälligen Anfangspopulation initialisiert.
Ein Sonderfall stellen die Ecken und Kanten des Feldes dar, da dort nach den Spielregeln das Verhalten nicht festgelegt ist. Die Aufgabenstellung gibt vor, dass das Spielfeld Torus-förmig sein soll. Alles, was das Spielfeld auf einer Seite verlässt, kommt auf der gegenüberliegenden Seite wieder herein.
Anschließend wird durch die Befolgung der Spielregeln die nächste Generation berechnet. Dafür betrachtet man jede Zelle und ihre 8 Nachbarn, um ihre Entwicklung zu berechnen. Es gelten folgende Spielregeln:

1. Eine lebende Zelle mit zwei oder drei Nachbarn überlebt in der Folgegeneration.
2. Eine lebende Zelle mit vier oder mehr Nachbarn stirbt an der Überpopulation. Bei weniger als zwei Nachbarn stirbt sie an Einsamkeit.
3. Jede tote Zelle mit genau drei Nachbarn wird in der nächsten Generation geboren.

Wichtig ist, dass die Folgegeneration für alle Zellen berechnet wird und anschließend die aktuelle Generation ersetzt. Es ist also nicht möglich, die nachfolgende Generation im Spielfeld der Aktuellen zu berechnen.

## 1.2 Besonderheiten der Aufgabenstellung

Die Aufgabenstellung gibt vor, dass die Parallelisierung mittels `OpenMP` Compiler-Direktiven erfolgen soll. `OpenMP` ist eine API, die es ermöglicht, Schleifen mithilfe von Threads zu parallelisieren [ope18], was in Aufgabe B thematisiert wurde. Es eignet sich hervorragend für `Shared-Memory Systeme`, also Systeme, bei denen mehrere Threads auf einen gemeinsamen Hauptspeicher zugreifen.
`OpenMP` bietet auch die Möglichkeit, explizit SIMD Instruktionen für die Bearbeitung von Schleifen zu verwenden. Das soll in dieser Aufgabe bearbeitet werden.
Weitere Besonderheiten sind:

- Die Simulation soll variabel mit Feldgrößen von $128 \times 128$ bis $32768 \times 32768$ erfolgen.
- Es sollen Messungen mit aktiviertem und inaktivem `OpenMP` durchgeführt werden.
- Das Programm soll mit dem `GCC` und `ICC` kompiliert und anschließend getestet werden.

## 2 Implementierung

Zuerst habe ich mich mit der Abstraktion des Feldes in `C` beschäftigt. Meine Idee ist die Allokierung eines Speicherbereichs der Größe `columns * rows * sizeof(u_int8_t)` durch die C-Funktion `malloc()`. Innerhalb des Speicherbereichs kann man sich nun frei bewegen. Dabei verwendet man die `columns` als Offset, um an die entsprechende Stelle zu springen. Praktischerweise entspricht eine Zelle im Feld einem Byte im Speicher.
Beispiel: Möchte man auf die zweite Zelle in der zweiten Zeile (da die Nummerierung typischerweise bei 0 beginnt, also das erste Element) zugreifen, würde man das `columns + 1` Byte innerhalb des Speicherbereichs verwenden.
Der Datentyp `u_int8_t` benötigt dabei nur ein Byte pro Zelle und ist für die Speicherung mehr als ausreichend, da ich nur den Zustand 0 - Zelle tot und 1 - Zelle lebendig speichern muss. Ein Byte ist typischerweise die kleinste adressierbare Einheit im Speicher. Das ist auch der Grund, warum kein noch kleinerer Datentyp möglich ist.

Um zu berücksichtigen, dass die Folgegeneration immer die aktuelle Generation ersetzt, allokiere ich einen zweiten Speicherbereich gleicher Größe. Vor dem Beginn einer neuen Berechnung, vertausche ich die beide Speicherbereiche, was dazu führt, dass die im vorhergehenden Schritt berechnete Folgegeneration zur aktuellen Generation wird und eine neue Generation berechnet werden kann.

### 2.1 Daten Initialisierung

Gemäß den Startbedingungen muss nur eines der beiden Spielfelder mit Zufallswerten initialisiert werden. Um den Code möglichst einfach und effizient zu halten, verwende ich eine `for`-Schleife zur Iteration über jede Zelle des Arrays.
Für die Dateninitialisierung jeder Zelle mit Null oder Eins, verwende ich den Pseudo-Zufallszahlengenerator `rand()`. Den `seed` setzte ich mit der `time()` Funktion auf die aktuelle Uhrzeit. Damit beginnt das Spiel bei jedem Programmstart mit einer zufälligen Generation.

Der Zufallsgenerator kann nicht mit einer einfachen `OpenMP` Direktive SIMD-parallel ausgeführt werden. Allerdings habe ich mich aus Interesse damit auseinander gesetzt und bin über die `Xorshift` Generatoren gestoßen, genauer `Xorshift128+` [Vig17]. Die `Xorshift` Generatoren sind eine Familie von Pseudozufallszahlengeneratoren, die sich durch eine hohe Geschwindigkeit und einer anpassbaren Periodenlänge auszeichnen. `Xorshift128+` verwendet, wie der Name vermuten lässt, Addition statt Multiplikation, die in der Regel weniger Rechenintensiv ist.
Die von mir verwendete Implementierung ist auf GitHub verfügbar.

Da die AMD Rome EPYC 7702 Prozessoren nur AVX2 und nicht AVX512 unterstützen (vergleiche 3.1), verwende ich die Funktion `avx_xorshift128plus`, welche auf 256 Bit Registern arbeitet.
Meine Idee war, die 256 generierten Bits auf die 8 Bit großen Zellen des Spiels aufzuteilen. Damit lassen sich pro Durchgang 32 Zellen mit Zufallszahlen füllen, was zu einer deutlichen Geschwindigkeitssteigerung führen sollte.
Zusammengesetzt ergibt sich daraus folgender Code:

```
void field_initializer(uint8_t *state) {
  //fills fields with random numbers 0 = dead, 1 = alive

  srand(time(NULL));
  int num_one = rand();
  int num_two = rand();

  // create a new key
  avx_xorshift128plus_key_t mykey;
  avx_xorshift128plus_init(num_one, num_two, &mykey); // values must be non-zero

  for (int i = 0; i < columns * rows; i = i + 32) {
    // generate 32 random bytes, do this as many times as you want
    __m256i randomstuff = avx_xorshift128plus(&mykey);
    state[i] = _mm256_extract_epi8(randomstuff, 0) % 2;
    state[i + 1] = _mm256_extract_epi8(randomstuff, 1) % 2;
    state[i + 2] = _mm256_extract_epi8(randomstuff, 2) % 2;
    state[i + 3] = _mm256_extract_epi8(randomstuff, 3) % 2;
    state[i + 4] = _mm256_extract_epi8(randomstuff, 4) % 2;
    state[i + 5] = _mm256_extract_epi8(randomstuff, 5) % 2;
    state[i + 6] = _mm256_extract_epi8(randomstuff, 6) % 2;
    state[i + 7] = _mm256_extract_epi8(randomstuff, 7) % 2;
    state[i + 8] = _mm256_extract_epi8(randomstuff, 8) % 2;
    state[i + 9] = _mm256_extract_epi8(randomstuff, 9) % 2;
    state[i + 10] = _mm256_extract_epi8(randomstuff, 10) % 2;
    state[i + 11] = _mm256_extract_epi8(randomstuff, 11) % 2;
    state[i + 12] = _mm256_extract_epi8(randomstuff, 12) % 2;
    state[i + 13] = _mm256_extract_epi8(randomstuff, 13) % 2;
    state[i + 14] = _mm256_extract_epi8(randomstuff, 14) % 2;
    state[i + 15] = _mm256_extract_epi8(randomstuff, 15) % 2;
    state[i + 16] = _mm256_extract_epi8(randomstuff, 16) % 2;
    state[i + 17] = _mm256_extract_epi8(randomstuff, 17) % 2;
    state[i + 18] = _mm256_extract_epi8(randomstuff, 18) % 2;
    state[i + 19] = _mm256_extract_epi8(randomstuff, 19) % 2;
    state[i + 20] = _mm256_extract_epi8(randomstuff, 20) % 2;
    state[i + 21] = _mm256_extract_epi8(randomstuff, 21) % 2;
    state[i + 22] = _mm256_extract_epi8(randomstuff, 22) % 2;
    state[i + 23] = _mm256_extract_epi8(randomstuff, 23) % 2;
    state[i + 24] = _mm256_extract_epi8(randomstuff, 24) % 2;
    state[i + 25] = _mm256_extract_epi8(randomstuff, 25) % 2;
    state[i + 26] = _mm256_extract_epi8(randomstuff, 26) % 2;
    state[i + 27] = _mm256_extract_epi8(randomstuff, 27) % 2;
    state[i + 28] = _mm256_extract_epi8(randomstuff, 28) % 2;
    state[i + 29] = _mm256_extract_epi8(randomstuff, 29) % 2;
    state[i + 30] = _mm256_extract_epi8(randomstuff, 30) % 2;
    state[i + 31] = _mm256_extract_epi8(randomstuff, 31) % 2;
  }
}
```

Listing 1: Daten Initialisierung

Abbildung 1: Generiertes Spielfeld der Größe: $64 \times 64$

### 2.2 Berechnung der nächsten Generation

Die Berechnung der nächsten Generation erfolgt mithilfe beider Spielfelder.
Die Funktion `calculate_next_gen()` erhält einen Pointer auf das Array mit der aktuellen Generation `*state_old` und einen auf das Array der Folgegeneration `*state`.
Bei jedem Simulationsschritt werden die Pointer getauscht und die Funktion erneut aufgerufen. Damit wird die Forderung der Aufgabenstellung nach *double buffering* erfüllt, sprich die Folgegeneration in einem separatem Spielfeld berechnet.

```
for (int i = 0; i < repetitions; i++) {
  calculate_next_gen(state_out, state_in);
  state_tmp = state_in;
  state_in = state_out;
  state_out = state_tmp;
}
```

Listing 2: Vertauschen der Pointer vor jedem Funktionsaufruf (vereinfacht)

Da es sich um ein Torus-förmiges Spielfeld handelt, benötigen die Kanten und Ecken eine separate Behandlung. Diese unterschiedet sich nur unwesentlich von der Berechnung des inneren Feldes.

#### 2.2.1 Inneres Feld

Der Zustand der Zelle in der nächsten Generation wird über die Spielregeln bestimmt und ist abhängig vom aktuellen Zustand der Zelle und ihren acht Nachbarn. Da der Zustand mit Null (tot) oder Eins (lebendig) repräsentiert wird, kann die Zahl der Nachbarzellen aufsummiert werden. Die Summe entspricht dabei der Zahl lebender Nachbarn.
An dieser Stelle könnte mithilfe einer *if*-Verzweigung der Folgezustand entschieden werden. Allerdings entschied ich mich für die Verwendung von bitweisen Operatoren. Es handelt sich dabei aus schaltungstechnischer Sicht um die einfachsten Operationen auf den einzelnen Bits.
Der Grund liegt darin, dass diese bitweisen Operatoren sich gut bei SIMD Operationen verwenden lassen.

Im ersten Schritt werden alle Zellen berechnet, die nicht Teil einer Kante sind. Dafür verwende ich zwei geschachtelte `for`-Schleifen:

```
for (int i = 1; i < rows - 1; i++) {
  #pragma omp simd
  for (int j = 1; j < columns - 1; j++) {
    //count up the neighbours
    u_int8_t sum_of_8 = state_old[(i - 1) * columns + (j - 1)] +
                        state_old[(i - 1) * columns + j] +
                        state_old[(i - 1) * columns + (j + 1)] +
                        state_old[i * columns + (j - 1)] +
                        state_old[i * columns + (j + 1)] +
```

```
                        state_old[(i + 1) * columns + (j - 1)] +
                        state_old[(i + 1) * columns + j] +
                        state_old[(i + 1) * columns + (j + 1)];
    state[i * columns + j] = (sum_of_8 == 3) | ((sum_of_8 == 2) & state_old[i *
    columns + j]);
  }
}
```

Listing 3: Berechnung der inneren Zellen

Die erste Schleife iteriert dabei über jede Zeile und in jeder Zeile die Zweite durch jede Zelle. Ich habe dafür die `OpenMP` Direktive

```
#pragma omp simd
```

verwendet.
Dabei wird jedoch nur die innere Schleife parallelisiert. Das bewirkt, dass die Spalten jeweils parallel berechnet werden. `OpenMP` wäre in der Lage, mit `collapse(2)` zwei geschachtelte Schleifen zu parallelisieren. In meinen Tests führte dies zu einer Verschlechterung der Performance, weswegen ich es nicht verwendet habe.
Zum anderen kann der Compiler, bei meiner Implementierung, die äußere Schleife modifizieren und so eventuelle Optimierungen vornehmen.

### 2.2.2 Kanten

Wie bereits erwähnt, unterscheidet sich die Art und Weise der Berechnung der Kanten nur unwesentlich von der des inneren Feldes. Da die Kanten jeweils nur aus einer Zeile bzw. Spalte bestehen, wird nur eine `for`-Schleife benötigt. Außerdem muss in der Berechnung beachtet werden, dass Felder von der gegenüberliegenden Seite benötigt werden. Auch hier wurden die Kanten wieder mit

```
#pragma omp simd
```

parallelisiert.

```
void calculate_top(u_int8_t *state, u_int8_t *state_old) {
  #pragma omp simd
  for (int i = 1; i < columns - 1; i++) {
    u_int8_t sum_of_t_edge = state_old[i - 1] +
                             state_old[i + 1] +
                             state_old[2 * columns + (i - 1)] +
                             state_old[2 * columns + i] +
                             state_old[2 * columns + (i + 1)] +
                             state_old[(rows - 1) * columns + i] +
                             state_old[(rows - 1) * columns + i + 1] +
                             state_old[(rows - 1) * columns + i - 1];
    state[i] = (sum_of_t_edge == 3) | ((sum_of_t_edge == 2) & state_old[i]);
  }
}
```

Listing 4: Berechnung der obersten Zeile

### 2.2.3 Ecken

Bei den Ecken ist keine Parallelisierung möglich und auch nicht notwendig, da vier Ecken bei einem Feld mit mehr als 16.000 Felder nicht ins Gewicht fallen.
Die Berechnung basiert wieder auf der vorher aufgeführten Methode.

```
1  void calculate_corner(u_int8_t *state, u_int8_t *state_old) {
2    u_int8_t corner_sum;
3    // top left
4    corner_sum = state_old[1] +
5    state_old[columns] +
6    state_old[columns + 1] +
7    state_old[(rows - 1) * columns] +
8    state_old[(rows - 1) * columns + 1] +
9    state_old[columns - 1] +
10   state_old[2 * columns - 1] +
11   state_old[rows * columns - 1];
12   state[0] = (corner_sum == 3) | ((corner_sum == 2) & state_old[0]);
13 }
```

Listing 5: Berechnung der Ecke oben links

## 2.3 Zeitmessung

Für die Zeitmessung habe ich auf die `clock()` Funktion zurückgegriffen.

Die `clock()` Funktion misst dabei die CPU Zeit des Prozesses und nicht die reale vergangene Zeit (*wall-clock time*). Das Ergebnis ist also die aufaddierte Zeit aller Threads in der Funktion. Da dieses Programm jedoch nur von einem Thread ausgeführt wird, stellt es kein Problem dar.

Ich entschied mich dafür, nicht nur die Funktion `calculate_next_gen()` zu messen, sondern auch die `field_initializer()` Funktion. Dadurch lässt sich später eine genauere Aussage über die Parallelisierbarkeit des Problems treffen. Für die Berechnung der Ausführungszeit wird vor und nach Ausführung der zu untersuchenden Funktion `clock()` aufgerufen. Die Differenz aus den beiden Momentaufnahmen entspricht der jeweiligen Zeit.

```
double time_calc = 0;
for (int i = 0; i < repetitions; i++) {
  t = clock();
  // function call
  t = clock() - t;
  time_calc += ((double) t) / CLOCKS_PER_SEC;
}
printf("Calculation took %f seconds to execute (all threads added).\n", time_calc);
printf("Calculation took %f seconds to execute (real time).\n", omp_calc);
```

Listing 6: Berechnung der Ausführungszeit eines *function calls*

### 2.4 Ein- und Ausgabe

Da die Messung später in verschiedenen Feldgrößen durchgeführt wird, habe ich mich für den Einsatz von `getopt` entschieden. Es ermöglicht, die Anzahl der Schleifendurchläufe, die Feldgröße und eine optionale Fortschrittsanzeige über Argumente beim Programmstart einzustellen. Ebenso lässt sich das Ergebnis über *pbm*-Files visualisieren.
Alle Funktionen sowie die Syntax lassen sich über den Parameter `--help` ausgeben.

# 3 Zeitmessung auf Taurus

## 3.1 Testumgebung

Alle Messungen wurden auf dem Hochleistungsrechner Taurus der TU Dresden durchgeführt.
Verwendet habe ich die Romeo-Partition, die auf AMD Rome EPYC 7702 Prozessoren basiert [Mar21]. Hier reservierte ich für die Messungen einen kompletten Node, um Schwankungen durch andere Prozesse auf dem Knoten auszuschließen. Ein Node ist mit 512 GB RAM ausgestattet. Als Betriebssystem kommt Centos 7 zum Einsatz.

Vor Beginn der Messung muss noch die Topologie des unterliegenden Systems betrachtet werden. Für die optimale Performance, sollten die größten Vektorregister zum Einsatz kommen. Bei den EPYC Prozessoren entspricht das AVX2. Die Breite der Register liegt hier bei 256 Bit.
Außerdem wird FMA unterstützt. FMA steht für Fused-multiply-add und steigert die Leistung durch verbesserte Ausnutzung von Registern und einen kompakteren Maschinencode. Das wird durch das Zusammenfassen einer Addition und Multiplikation zu einem Befehl erreicht.
Wie in der Aufgabenstellung gefordert, kompilierte ich das Programm mit dem `GCC` (**G**NU **C**ompiler **C**ollection, Version 11.2) und dem `ICC` (**I**ntel **C**ompiler **C**ollection, Version 19.0.5.281); jeweils mit den Compiler-Flags:

- `O3` - Optimierungsflag des Compilers
- `mavx2` - Verwendung von AVX2
- `fma` - Verwendung von FMA
- `fopenmp` - Verwendung nur für Tests mit OpenMP

Da für die Messungen nur ein Core verwendet werden soll, ist es nicht notwendig Threads an bestimmte Cores zu pinnen, wie das noch bei Aufgabe B erforderlich war.

**Bemerkung zur `O3` Flag:**
Die Optimierungsflag teilt dem Compiler mit, dass er die Performance auf Kosten der Programmgröße und Kompilationszeit erhöht. Das schließt SIMD Instruktionen ein. Ohne diese Flag hat der `GCC` auch mit aktiviertem `OpenMP` keine SIMD Instruktionen verwendet, was ich mit `objdump` verifiziert habe. Da die Aufgabenstellung vorgibt, man solle aktiviertes und deaktiviertes `OpenMP` vergleichen, habe ich die Optimierungsflag bei beiden aktiviert gelassen.
Das führt zu dem Ergebnis, dass der `GCC` nur SIMD Instruktionen in Kombination mit der Optimierungsflag verwendet.
Der Intel Compiler verwendet hingegen SIMD Instruktionen nur, wenn die `fopenmp` Flag gesetzt wurde.

## 3.2 Testmethode

Die Tests wurden automatisiert mit einem `sbatch`-Skript ausgeführt. Um Schwankungen auszugleichen, wurde jede Messung 20 mal wiederholt.

Dabei ist zu beachten, dass die Ausführungszeit (logischerweise) mit der Feldgröße linear ansteigt. Interessanter für das Praktikum ist jedoch das Verhalten bei der Verwendung von SIMD Instruktionen. Deshalb passte ich die Anzahl der Wiederholungen an die Feldgröße an. Die genauen Details lassen sich den Tabellen **??** und **??** entnehmen.
Die Anzahl von Simulationsschritten ist auf 100 für alle Feldgrößen festgelegt. Ich habe diese Größe gewählt, da so ein Vergleich zwischen verschiedenen Feldgrößen möglich wird, um zu entschieden, wie sich die Ausführungszeit in Abhängigkeit dieser verändert.
Für noch genauere Ergebnisse wären mehr Simulationsschritte nötig gewesen. Bei einer Feldgröße von 128 mit mehr als 100.000 Wiederholungen befindet sich die Ausführungszeit immer noch im Millisekunden Bereich. Aufgrund begrenzter CPU Zeit ist eine so hohe Anzahl an Simulationsschritten nicht möglich.

Aus den 20 Wiederholungen pro Messung habe ich den Mittelwert gebildet und Logarithmische Diagramme erzeugt. Diese Darstellung bietet sich aufgrund der exponentiell ansteigenden Feldgröße an.

**Anmerkung zur Notation:** Mit einer Feldgröße von 128 meine ich ein Feld mit $128 \times 128$ Zellen. Da hier im Praktikum alle Feldgrößen quadratisch sind, ist die Angabe der zweiten Größe redundant, weswegen ich auch auf diese verzichte.

# 4 Testergebnisse

In diesem Abschnitt habe ich die Messwerte aus den Tabellen A visualisiert.

## 4.1 Dateninitialisierung

Zuerst möchte ich auf die Ausführungszeiten der Funktion `field_initializer()` des Spiels eingehen.
Diese Funktion füllt das Spielfeld mit zufälligen Werten. Wie Eingangs in 2.1 beschrieben, lässt sich diese Funktion nicht mit SIMD Instruktionen parallelisieren.
Ich implementierte jedoch einen `Xorshift128+` Pseudozufallszahlengenerator, um das Problem SIMD parallel auszuführen. Die Ergebnisse habe ich hier dargestellt.

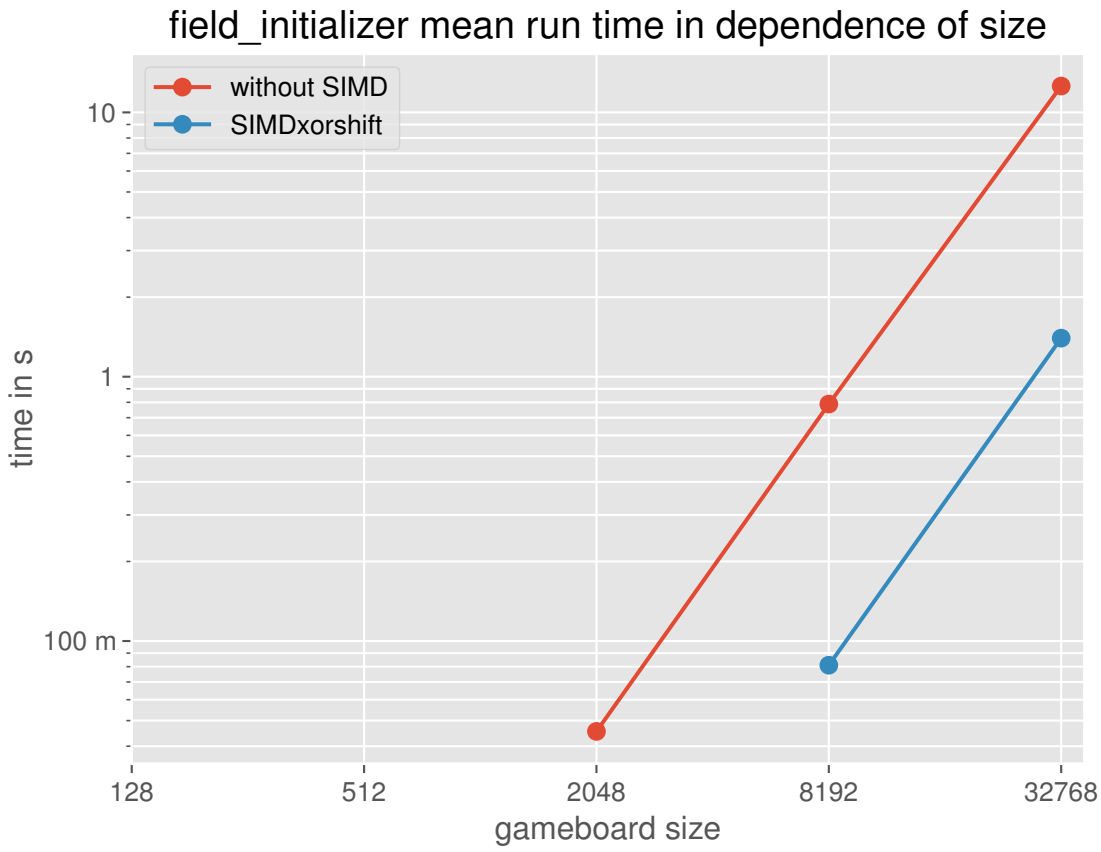


Abbildung 2: Logarithmische Darstellung der Ausführungszeit der Funktion `field_initializer()`, kompiliert mit `GCC`.

Speedup berechnen

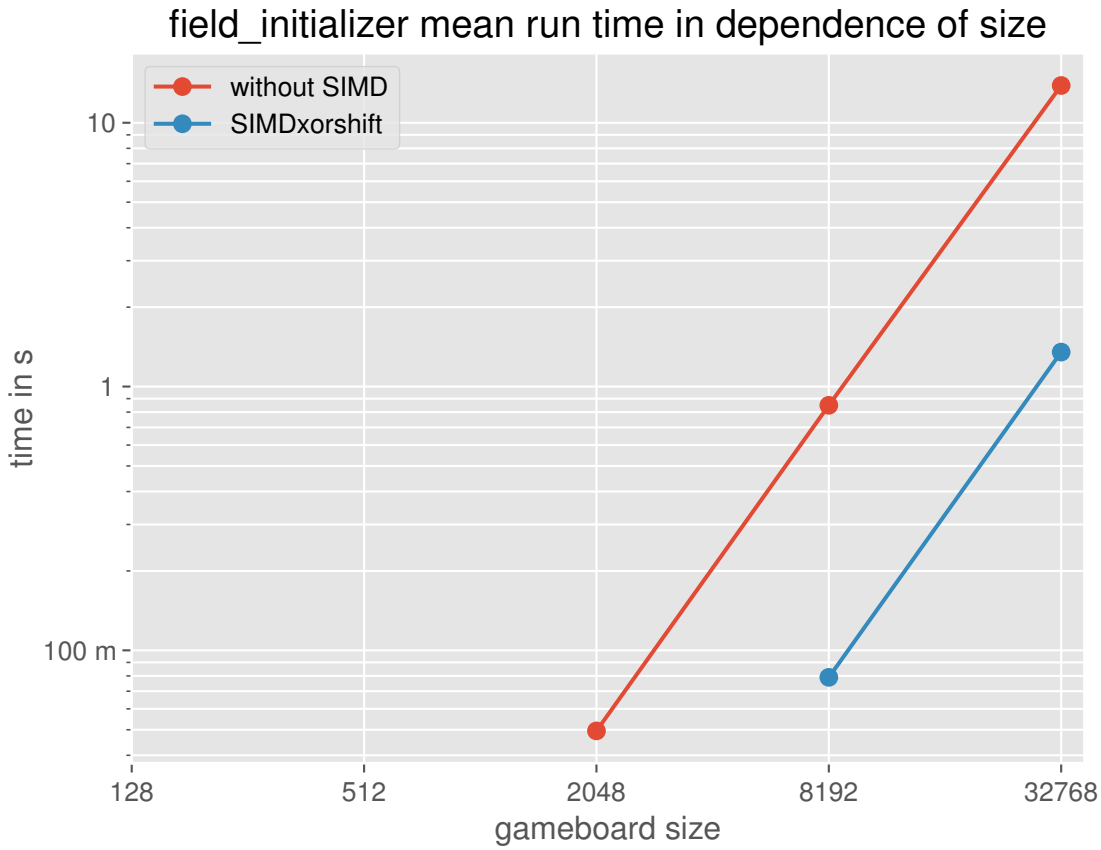


Abbildung 3: Logarithmische Darstellung der Ausführungszeit der Funktion `field_initializer()`, kompiliert mit `ICC`.

### 4.2 Berechnung

Ähnlich zur `GCC` kompilierten Version des Spiels sind hier die selben Effekte erkennbar. Mehr Threads verkürzen generell die Ausführungszeit und bei kleinen Feldgrößen verringert sich zunehmend der Speedup.
Auffällig ist, dass die Version des Intel Compilers bei der jeweilig selben Anzahl von Threads und Feldgröße nur ca. 25% der Zeit benötigt, also zu jeder Zeit etwa vier mal so schnell ist. Der Intel Compiler produziert folglich besser optimierten Code.
Ich habe mir mittels `objdump` den disassemblierten Code beider Binaries angeschaut, da ich vermutete, dass der Intel Compiler möglicherweise `SIMD`-Instruktionen verwendet. Das ist jedoch nicht der Fall.

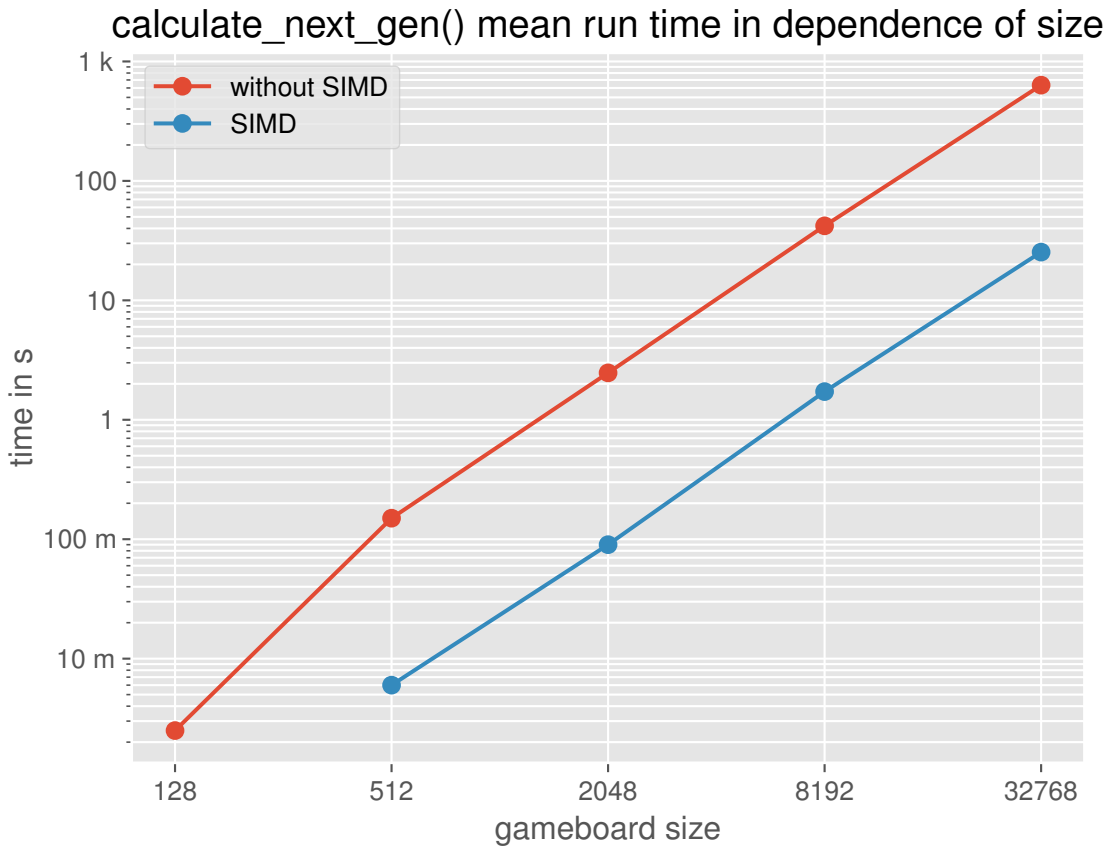


Abbildung 4: Logarithmische Darstellung der Ausführungszeit der Funktion `calculate_next_gen()`, kompiliert mit `ICC`.

Bei der Dateninitialisierung zeigt sich beim Intel Compiler ein etwas anderes Bild. Im Allgemeinen schwanken die Werte der `ICC` Version weniger stark.
Bei Feldgrößen ab 2048 ist er ebenfalls schneller als die Version des `GCC` Compilers. Bei den Werten darunter ist ein Vergleich aufgrund der Schwankungen schwierig. Tendenziell lässt sich wieder ein Anstieg der Zeit mit steigender Zahl an Threads verbuchen.

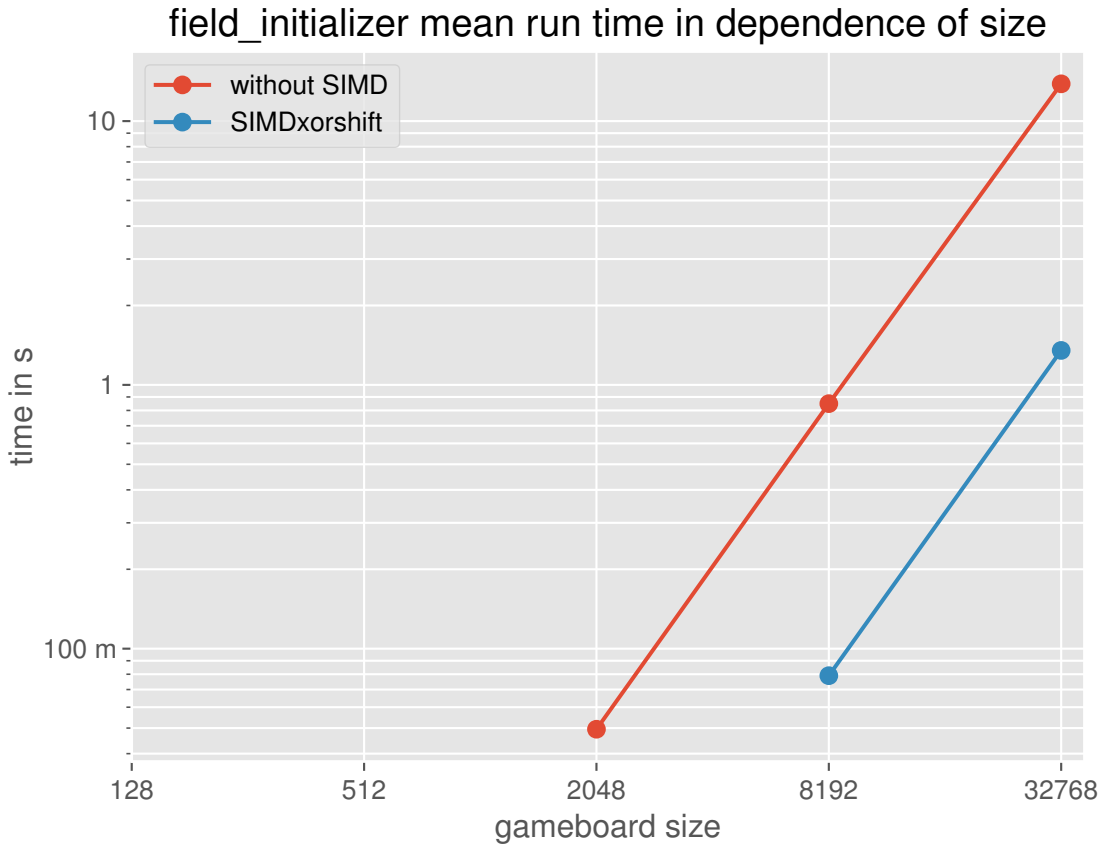


Abbildung 5: Logarithmische Darstellung der Ausführungszeit der Funktion `field_initializer()`, kompiliert mit `ICC`.

## A Tabellen

| Compiler | Size | SIMD | Repetitions | Initialization in s |
|---|---|---|---|---|
| gcc | 128 | without SIMD | 100 | 0.0 |
| gcc | 128 | SIMDxorshift | 100 | 0.0 |
| gcc | 512 | without SIMD | 100 | 0.0 |
| gcc | 512 | SIMDxorshift | 100 | 0.0 |
| gcc | 2048 | without SIMD | 100 | 0.0455 |
| gcc | 2048 | SIMDxorshift | 100 | 0.0 |
| gcc | 8192 | without SIMD | 100 | 0.788 |
| gcc | 8192 | SIMDxorshift | 100 | 0.081 |
| gcc | 32768 | without SIMD | 100 | 12.5835 |
| gcc | 32768 | SIMDxorshift | 100 | 1.3995 |
| icc | 128 | without SIMD | 100 | 0.0 |
| icc | 128 | SIMDxorshift | 100 | 0.0 |
| icc | 512 | without SIMD | 100 | 0.0 |
| icc | 512 | SIMDxorshift | 100 | 0.0 |
| icc | 2048 | without SIMD | 100 | 0.0495 |
| icc | 2048 | SIMDxorshift | 100 | 0.0 |
| icc | 8192 | without SIMD | 100 | 0.849 |
| icc | 8192 | SIMDxorshift | 100 | 0.079 |
| icc | 32768 | without SIMD | 100 | 13.84588 |
| icc | 32768 | SIMDxorshift | 100 | 1.351 |

Tabelle 1: Testergebnisse der GCC kompilierten Version des Game-of-Life. Zeiten gerundet auf vier Stellen.

| Compiler | Size | SIMD | Repetitions | Calculation in s |
|---|---|---|---|---|
| gcc | 128 | without SIMD | 100 | 0.0 |
| gcc | 128 | SIMD | 100 | 0.0 |
| gcc | 512 | without SIMD | 100 | 0.009 |
| gcc | 512 | SIMD | 100 | 0.01 |
| gcc | 2048 | without SIMD | 100 | 0.0825 |
| gcc | 2048 | SIMD | 100 | 0.093 |
| gcc | 8192 | without SIMD | 100 | 1.3805 |
| gcc | 8192 | SIMD | 100 | 1.4895 |
| gcc | 32768 | without SIMD | 100 | 18.752 |
| gcc | 32768 | SIMD | 100 | 21.272 |
| icc | 128 | without SIMD | 100 | 0.0025 |
| icc | 128 | SIMD | 100 | 0.0 |
| icc | 512 | without SIMD | 100 | 0.15 |
| icc | 512 | SIMD | 100 | 0.006 |
| icc | 2048 | without SIMD | 100 | 2.47 |
| icc | 2048 | SIMD | 100 | 0.09 |
| icc | 8192 | without SIMD | 100 | 42.0535 |
| icc | 8192 | SIMD | 100 | 1.7215 |
| icc | 32768 | without SIMD | 100 | 633.28882 |
| icc | 32768 | SIMD | 100 | 25.341 |

Tabelle 2: Testergebnisse der ICC kompilierten Version des Game-of-Life. Zeiten gerundet auf vier Stellen.

## Literatur

[Gar70] GARDNER, Martin. *MATHEMATICAL GAMES - The fantastic combinations of John Conway's new solitaire game "life"*.
https://web.stanford.edu/class/sts145/Library/life.pdf. 1970

[lmp10] *Linux Man Pages - rand*.
https://linux.die.net/man/3/rand. 2010

[Mar21] MARKWARDT, Dr. U. *Introduction to HPC at ZIH*.
https://doc.zih.tu-dresden.de/misc/HPC-Introduction.pdf. 2021

[ope18] *OpenMP Application Programming Interface 5.0*.
https://www.openmp.org/wp-content/uploads/
OpenMP-API-Specification-5.0.pdf. 2018

[Vig17] VIGNA, Sebastiano. *Further scramblings of Marsaglia's xorshift generators*.
https://vigna.di.unimi.it/ftp/papers/xorshiftplus.pdf. 2017