Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Projektbericht

Performanzoptimierung SAGA-GIS

vorgelegt von

Rasmus Pranke

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6920830

Betreuer: Jannek Squar

Hamburg, 24. September 2021

Abstract

SAGA-GIS ist eine Software zur Verarbeitung geographischer Daten, welche in der bisherigen Entwicklung nicht auf Performanz optimiert wurde. Im in der vorliegenden Arbeit vorgestellten Projekt habe ich diese Software analysiert und die Performanz verbessert, indem ich Ladevorgänge aus dem Hauptspeicher vermieden habe. Regressionstests dienten zur Sicherung der Funktionalität, das Programm wurde mit dem Microsoft Visual Studio Debugger analysiert und verschiedene Ansätze ausprobiert. Der erfolgreichste Ansatz hat in Form einer "gleitenden Matrix" Werte auf dem Stapel zwischengespeichert. Es ergab sich ein Performanzgewinn von im Schnitt 15%, mit einzelnen Gewinnen von bis zu 50%.

Inhaltsverzeichnis

1. Problemstellung

Das Geoinformationsystem SAGA ist nicht auf Performanz optimiert und Performanz wurde in der Entwicklung nicht priorisiert. Deshalb sollen die bestehenden Tools auf mögliche Performanzverbesserungen überprüft werden.

1.1. SAGA

SAGA (System for Automated Geoscientific Analyses) ist ein Geo-Informations-System (GIS). Es ist in C++ geschrieben und verfolgt eine Objektorientierte Architektur. Es bietet eine allgemeine Schnittstelle für Tools zur Verarbeitung von Geographiedaten und erlaubt damit, viele verschiedene Verarbeitungswege in einer Software zu einen. [3] Es befinden sich laufend weitere Tools in der Entwicklung, wodurch der Funktionsumfang stetig wächst.[2]

1.2. Optimierung

Das Performanz-Problem sowie die Optimierungsmöglichkeiten werden durch das Tool Slope, Aspect, Curvature aus der Morphometrie-Toolchain gut illustriert. Auf der Datei srtm_germany_dsm.tif der Seite opendm.info (mit einer Dateigröße von 272 MB) braucht die Ausführung des Tools auf meiner Maschine (Siehe Sektion A und B) 11 Sekunden.

Dieser Datensatz ist für Geographiedaten relativ klein. Von der Seite Copernicus.eu lassen sich Dateien im Bereich von 4GB und größer herunterladen - die vollständigen Datensätze sind dabei noch größer, die einzelnen Dateien stellen nur Teile dieser dar. Entsprechend lange fallen die Ausführungszeiten aus.

Die Aufgabe besteht nun darin herauszufinden, ob und wo die Performanz der Software verbessert werden kann, und diese auch tatsächlich zu verbessern.

2. Arbeitsweise

Die Arbeit fand auf der der Version RELEASE-7.9.0 statt, wie sie am 10.12.2020 auf dem Sourceforge[2] zu finden war. Diese Version wird von hier an BASISVERSION genannt.

Das Projekt bestand aus 3 sich wiederholenden Phasen:

- Funktionalität sichern
- Performanz analysieren
- Performanz verbessern

Diese werden im Folgenden im Bezug auf die Arbeitsweise näher erläutert.

2.1. Funktionalität sichern

Für jeden betrachteten Teil der Software muss erst sichergestellt werden, dass die Arbeit an diesem die Ergebnisse nicht verfälscht. Die Software hat aktuell keine Tests. Deshalb habe ich Regressionstests eingesetzt, welche ich in Python implementiert habe. Regressionstests zeichnen sich dadurch aus, dass sie feststellen, ob Änderungen bestehende Funktionalität beeinträchtigen.[1]

Diese neu für dieses Projekt geschriebenen Tests zielen darauf ab, die aktuelle Funktionalität ganzheitlich sicherzustellen. Hierfür benutze ich echte Daten und lasse diese von der Basisversion verarbeiten. Da die Berechnungen kein Zufallselement beinhalten, ist das Ergebnis dieser Verarbeitung auf einem gegebenen Datensatz reproduzierbar.

Wenn ich nun für eine Änderung sicherstellen möchte, ob diese die Berechnung verfälscht, lasse ich die geänderte Version (automatisiert) die Daten verarbeiten und vergleiche das Ergebnis mit dem Ergebnis der Basisversion. Wenn die Ergebnisse identisch sind, ist die Wahrscheinlichkeit, dass die Berechnung verfälscht wurde, gering. Sollten Fehler enstanden sein, sind diese auf Randfälle begrenzt. Die so gegebene Sicherheit ist ausreichend für dieses Projekt und kann später durch ausführlichere Tests verbessert werden.

2.1.1. Implementation

Die Regressionstests bestehen aus einem Test-Lader und einer Reihe an Test-Implementationen. Da die betrachteten Tools sehr ähnlich funktionieren ist die Struktur der Implementationen kompakt.

Es muss ein Name definiert werden. Die Tests werden vom Lader anhand ihrer Dateinamen unterschieden. Jeder Test erstellt das zu testende Tool, konfiguriert es, führt es aus, und gibt die Ergebnisse zurück.

```
1  # Definiert welche Dateitypen fuer diesen Test valide sind
2  def file_type():
3    return ".tif"
4
5  # Definiert einen lesbaren Namen.
6  def get_long_name():
7    return "Slope, Aspect, Curvature: Grid Files, ZevenBergen& Thorne,
       ↪ Aspect and Slope output"
8
9  # Erschafft das Tool.
10 tool = None
11 def create_tool(create):
12   global tool
13   tool = create('ta_morphometry', 0)
14
15 # Setzt die Parameter und gibt die Ergebnisse zurueck.
16 def run_tool(data, execute):
17   tool.Get_Parameters().Reset_Grid_System()
18   tool.Set_Parameter('ELEVATION', data)
19   tool.Set_Parameter('METHOD', '9 parameter 2nd order polynom
       ↪ (Zevenbergen & Thorne 1987)')
20   tool.Set_Parameter('UNIT_SLOPE', 'radians')
21   tool.Set_Parameter('UNIT_ASPECT', 'radians')
22
23   execute(tool)
24   return [tool.Get_Parameter('SLOPE').asDataObject(),
       ↪ tool.Get_Parameter('ASPECT').asDataObject()]
25
26 # Loescht das Tool.
27 def delete_tool(destroy):
28   global tool
29   destroy(tool)
30   tool = None
```

Die Ergebnisse werden gesammelt und anschließen mit den Zielergebnissen verglichen, oder alternativ als neues Zielergebnis gespeichert. Am Ende wird eine Übersicht gegebenen, welche Ergebnisse korrekt waren, und wo Fehler festgestellt wurden.

2.1.2. Funktionalitätsunterschied zwischen Linux und Windows

Im Zuge der Tests habe ich festgestellt, dass die auf Linux und auf Windows erzeugten Dateien nicht identisch sind. Da die Dateien aber innerhalb eines Systems reproduzierbar sind und solche Fehler nicht Teil der Projektaufgabe sind, habe ich diesen Unterschied nicht weiter erforscht und stattdessen die Ergebnisse der Basisversion auf beiden Systemen erzeugt.

2.2. Performanz analysieren

Um die Performanz der Software zu analysieren habe ich den Visual Studio Debugger eingesetzt, welcher in Microsoft Visual Studio integriert ist. Dieser analysiert, in welchem Codeabschnitt wie viel Zeit verbracht wird und worauf diese Zeit verwendet wird.

Damit konnte ich nicht nur die Codestellen finden, die Flaschenhälse darstellen, sondern auch, warum sie welche sind.

Zusätzlich habe ich eine Variante der Regressionstests eingesetzt, um die ganzheitliche Performanz automatisiert testen zu können.

Ich habe mit der Performanz-Analyse bei dem Tool Slope, Aspect, Curvature begonnen. Dieses Tool wird oft als erster Schritt verwendet, wenn digitale Höhendaten verwendet werden. Daher ist es ein guter Einstiegspunkt, um Ergebnisse zu erzielen, die Anwendern weiterhelfen.

2.2.1. Visual Studio Debugger

Mit dem Visual Studio Debugger lässt sich die Performanz analysieren, indem 2 Haltepunkte gesetzt werden und das Programm im Debug-Modus gestartet wird. Sobald der erste Haltepunkt getroffen wird, kann man das Profiling aktivieren. Nun wird aufgezeichnet, in welchen Codebereichen Zeit verbracht wird, bis der zweite Haltepunkt erreicht wird. Diese Daten lassen sich dann auswählen und danach aufschlüsseln, worauf Zeit verwendet wurde und an welcher Stelle im Code dies war.

Bei dem genannten Tool wird in der Basisversion mit den verwendeten Daten die meiste Zeit in einer Konvertierungsmethode und einer Cache-Check-Methode verbracht. Die Aufschlüsselung nach Art verrät, dass es sich um Zugriff auf den Hauptspeicher handelt.

2.2.2. Performanztests

Ich habe Performanztests eingesetzt, um die Ausführungszeit der betrachteten Tools automatisiert zu erfassen. Dafür habe ich die in Kapitel 2.1 beschriebenen Regressionstests wiederverwendet. Dort habe ich den bestehenden Testaufbau genommen und die

Ausführungszeit gemessen. Die Überprüfung der Ergebnisse habe ich entfernt, da sie bereits in den Regressionstests ausgeführt wird.

Die Performanztests erfassen deutlich weniger Daten als der Debugger, bieten aber zwei wichtige Vorteile. Zum einem wird die Release-Version statt der Debug-Version getestet, wodurch die Ergebnisse repräsentativer für die Ergebnisse in der Praxis sind. Zum anderen sind diese Tests einfach aus einem Skript heraus aufrufbar und damit automatisierbar. Dies vereinfachte die finale Analyse enorm.

2.3. Performanz verbessern

Mit den Ergebnissen aus der Performanz-Analyse konnte ich schließlich herangehen und mir überlegen, wie diese Flaschenhälse vermieden werden können. Der konkrete Prozess stellt hier eine Mischung aus Intuition und praktischem, wissenschaftlichem Ausprobieren dar. Zuerst habe ich den konkreten Flaschenhals untersucht, dann potentielle Lösungsansätze überlegt, und diese schließlich in einem iterativen Prozess ausprobiert und getestet. Die Erkenntnisse aus jeder Iteration halfen mir dabei, die Natur des Flaschenhalses besser zu ergründen und neue Ansätze zu finden.

Die Ergebnisse hiervon werden in den näheren Kapiteln ausführlicher betrachtet.

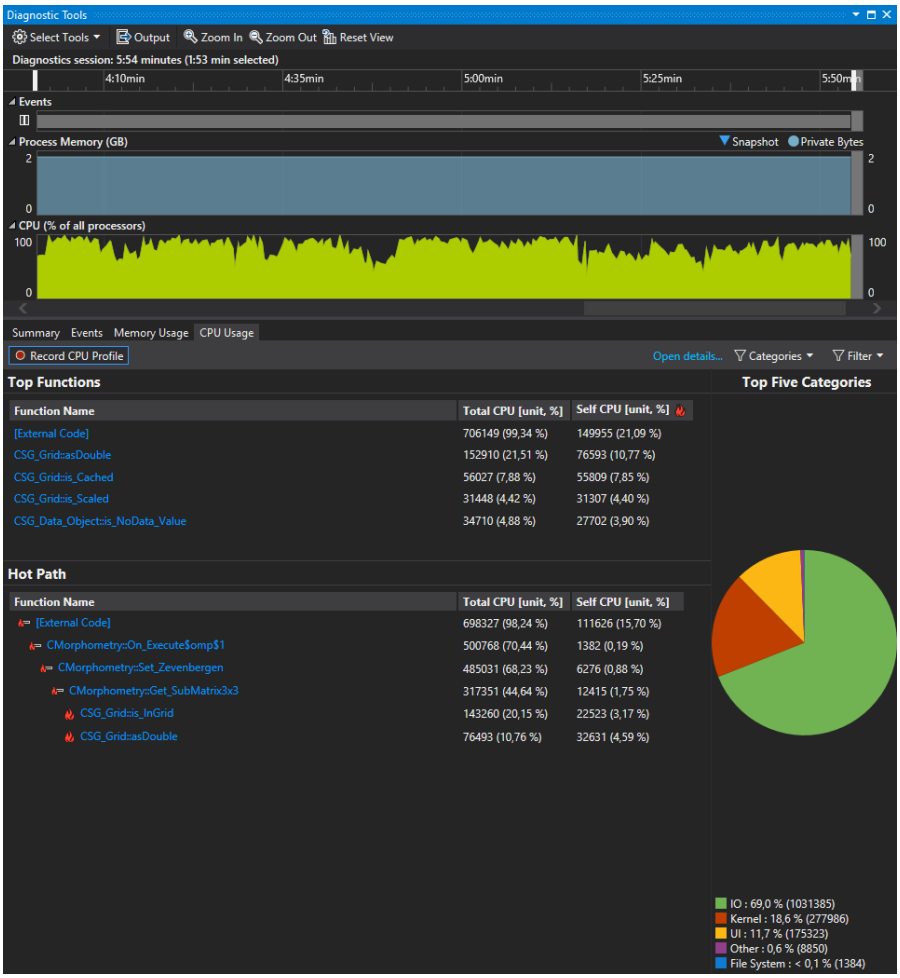

Abbildung 2.1.: Microsoft Visual Studio Debugger Diagnostic Tool
Oben ist die Speicher- und Prozessorauslastung zu sehen.
Unten links sind Stellen im Code nach verbrachter Zeit aufgeschlüsselt.
Unten rechts ist ein Tortendiagram dargestellt, dass die Auslastung nach Art aufschlüsselt.

3. Analysierte Ansätze

3.1. Vermeidung eines Switch-Statements mit Funktionszeigern

In jeder Iteration des Tools wird ein Switch-Statement durchlaufen:

```
1  for(int y=0; y<Get_NY() && Set_Progress(y); y++)
2  {
3      #pragma omp parallel for
4      for(int x=0; x<Get_NX(); x++)
5      {
6          if( m_pDTM->is_NoData(x, y) )
7          {
8              Set_NoData(x, y);
9          }
10         else switch( Method )
11         {
12         case 0: Set_MaximumSlope(x, y); break;
13         case 1: Set_Tarboton (x, y); break;
14         case 2: Set_LeastSquare (x, y); break;
15         case 3: Set_Evans (x, y); break;
16         case 4: Set_Heerdegen (x, y); break;
17         case 5: Set_BRM (x, y); break;
18         default: Set_Zevenbergen (x, y); break;
19         case 7: Set_Haralick (x, y); break;
20         case 8: Set_Florinsky (x, y); break;
21         }
22     }
23 }
```

Ich habe dieses Statement durch Funktionszeiger ersetzt, um die Abfrage des Switch-Statements zu vermeiden:

```
1  for(int y=0; y<Get_NY() && Set_Progress(y); y++)
2  {
3      void (CMorphometry::*method)(int, int);
```

```
4     switch (Method)
5     {
6     case 0: method = &CMorphometry::Set_MaximumSlope; break;
7     case 1: method = &CMorphometry::Set_Tarboton; break;
8     case 2: method = &CMorphometry::Set_LeastSquare; break;
9     case 3: method = &CMorphometry::Set_Evans; break;
10    case 4: method = &CMorphometry::Set_Heerdegen; break;
11    case 5: method = &CMorphometry::Set_BRM; break;
12    default: method = &CMorphometry::Set_Zevenbergen; break;
13    case 7: method = &CMorphometry::Set_Haralick; break;
14    case 8: method = &CMorphometry::Set_Florinsky; break;
15    }
16    #pragma omp parallel for
17    for(int x=0; x<Get_NX(); x++)
18    {
19        if (m_pDTM->is_NoData(x, y))
20        {
21            Set_NoData(x, y);
22        }
23        else (*this.*method)(x, y);
24    }
25 }
```

Im Test auf meinem persönlichen Rechner ergab sich keine messbare Verbesserung.

3.2. Performanzverbesserung mithilfe einer Gleitenden Matrix

Für jede x- und y-Koordinate im Grid wird in den Berechnungen des Tools SLOPE, ASPECT, CURVATURE eine 3x3-Matrix aus Werten geladen. Jeder dieser Werte wird dabei 2- bis 3-mal abgefragt. Gleichzeitig überlappen sich die geladenen Werte angrenzender Koordinaten.

```
1  inline void CMorphometry::Get_SubMatrix3x3(int x, int y, double Z[9],
       ↪ int Orientation)
2  {
3    static const int Indexes[][8] =
4    {
5      { 5, 8, 7, 6, 3, 0, 1, 2 },
6      { 5, 2, 1, 0, 3, 6, 7, 8 }
7    };
8
```

```
  int *Index = (int *)Indexes[Orientation];

  double z = m_pDTM->asDouble(x, y); # First load of x, y

  Z[4] = 0.;

  for(int i=0; i<8; i++)
  {
    int ix = Get_xTo(i, x);
    int iy = Get_yTo(i, y);

    # First load for every coordinate in the 3x3
    if( m_pDTM->is_InGrid(ix, iy) )
    {
      # Second load for every coordinate in the 3x3
      Z[Index[i]] = m_pDTM->asDouble(ix, iy) - z;
    }
    else
    {
      ix = Get_xTo(i + 4, x);
      iy = Get_yTo(i + 4, y);

      # Other second load for every coordinate in the 3x3
      if( m_pDTM->is_InGrid(ix, iy) )
      {
        # Third load for every coordinate in the 3x3
        Z[Index[i]] = z - m_pDTM->asDouble(ix, iy);
      }
      else
      {
        Z[Index[i]] = 0.;
      }
    }
  }
}
```

Um dieses mehrfache Laden der gleichen Werte zu vermeiden verwende ich eine Gleitende Matrix. Die gleitende Matrix ist eine C++-Klasse, die durch Abwesenheit des new-Schlüsselwortes auf dem Stapel erzeugt wird und die relevanten Werte für die Berechnungen zwischenspeichert. Der gewünschte Effekt ist hier, dass das Laden vom Stapel deutlich schneller ist als das Laden aus dem Hauptspeicher. Wenn dies der Fall ist, dann können pro Wert 2 Ladevorgänge und pro Koordinate bis zu 6 weitere Ladevorgänge

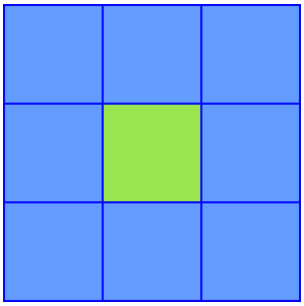

Abbildung 3.1.: Für einen einzelnen Wert werden alle angrenzenden Werte betrachtet

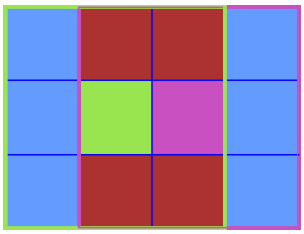

Abbildung 3.2.: Für zwei angrenzende Werte werden 6 Werte doppelt betrachtet.

eingespart werden.

Die Klasse ist als Template-Klasse deklariert, damit verschiedene Größen einfach abgedeckt werden können. Die Größe SIZE muss hierbei eine ungerade Zahl größer 1 sein. Die 2-dimensionalen Arrays VALUES und INGRID speichern die Daten des zugrundeliegenden Gitters. Sie werden hierbei als zirkuläre Buffer betrachtet, um Lese- und Schreibvorgänge zu sparen. Die nächste Spalte an Daten verschiebt die bisherigen nicht, sondern überschreibt die ältesten Daten. Hierfür wird in FILLCOLUMN dynamisch ermittelt, welche lokale Spalte die angeforderte Spalte halten soll. Entsprechend müssen auch die Zugriffe in den Zeilen 78 und 83 gestaltet werden.

Um die Klasse nun zu verwenden muss ein y gewählt und die Klasse instantiiert werden. Zu diesem Zeitpunkt hält die Klasse die Werte um die Koordinate $(x = 0, y)$ bereit. Mit jedem Aufruf vom SLIDERIGHT wird die nächste Spalte geladen und damit die Werte für $(x + 1, y)$ bereitgestellt.

```
1 template<int size>
2 class Submatrix {
3 public:
4     double asDouble(int localX, int localY);
5     bool is_InGrid(int localX, int localY);
6     bool is_NoData();
7     Submatrix(CSG_Grid* grid, int y, int nx, int ny);
8     Submatrix(CSG_Grid* grid, int x, int y, int nx, int ny);
9     int SlideRight();
```

```
10      int Get_X();
11      int Get_Y();
12      int Get_NX();
13      int Get_NY();
14  private:
15      static const int offset = (size - 1) / 2;
16
17      double values[size][size];
18      bool inGrid[size][size];
19      bool noData;
20
21      CSG_Grid* _grid;
22      int _x, _y, _nx, _ny;
23
24      void FillColumn(int x);
25  };
26
27  template class Submatrix<3>;
28  template class Submatrix<5>;
29
30  template<int size>
31  using MatrixMethod = std::function<void(Submatrix<size>&)>;
32
33  template<int size>
34  Submatrix<size>::Submatrix(CSG_Grid* grid, int y, int nx, int ny) :
        ↪ Submatrix(grid, 0, y, nx, ny){
35  }
36
37  template<int size>
38  Submatrix<size>::Submatrix(CSG_Grid* grid, int x, int y, int nx, int
        ↪ ny) {
39      _grid = grid;
40      _y = y;
41      _x = x;
42      _nx = nx;
43      _ny = ny;
44      for (int i = 0; i < size * size; i++) {
45          inGrid[i % size][i / size] = false;
46      }
47      for (int i = -offset; i <= offset; i++) {
48          FillColumn(x + i);
49      }
```

```
50 }
51
52 template<int size>
53 int Submatrix<size>::SlideRight() {
54     int x = _x + size - offset;
55     FillColumn(x);
56     _x++;
57     return _x;
58 }
59
60 template<int size>
61 void Submatrix<size>::FillColumn(int x) {
62     int column = (x + size) % size;
63     for (int j = 0; j < size; j++) {
64         int y = _y + j - offset;
65         bool isInGrid = _grid->is_InGrid(x, y);
66         inGrid[column][j] = isInGrid;
67         if (isInGrid) {
68             values[column][j] = _grid->asDouble(x, y);
69         }
70     }
71     if (is_InGrid(0, 0)) {
72         noData = _grid->is_NoData(_x, _y);
73     }
74 }
75
76 template<int size>
77 double Submatrix<size>::asDouble(int localX, int localY) {
78     return values[(localX + _x + size) % size][localY + offset];
79 }
80
81 template<int size>
82 bool Submatrix<size>::is_InGrid(int localX, int localY) {
83     return inGrid[(localX + _x + size) % size][localY + offset];
84 }
```

In der ersten Umsetzung habe ich dieses Verfahren nur für 2 bestimmte Konfigurationen des Tools umgesetzt. In Tests auf meinem persönlichen Rechner führte dies zu einer messbaren Verbesserung. Daraufhin habe ich die Implementation so verallgemeinert, dass diese einfach für weitere Tools einsetzbar ist und für die verbleibenden Konfigurationen des Tools sowie ein weiteres Tool namens Convergence umgesetzt.

Die Performanzverbesserung ließ sich übertragen.

3.3. Performanzverbesserung mithilfe von Batching

Die Überschneidung der 3x3-Matrizen, die von der gleitenden Matrix ausgenutzt wird, erfolgt nur entlang der x-Achse. Da sich die Werte aber in beiden Achsen überschneiden sollte eine Performanzverbesserung möglich sein, indem die Werte auch entlang der y-Koordinate wiederverwendet werden.

Um dies auszunutzen habe ich Batching eingeführt. Hier wird die Matrix in der y-Achse vergrößert und pro Position der Matrix mehrere Koordinaten berechnet. Alle diese Koordinaten teilen sich die x-Position und liegen untereinander auf der y-Position. Dadurch werden auch Werte entlang der y-Achse wiederverwendet. Die Menge der Koordinaten, die auf einmal berechnet werden, nenne ich Batch-Grösse. Pro Koordinate nach der ersten wird dann nur 1 zusätzlicher Wert aus dem Hauptspeicher geladen, statt wie vorher 3.

Auf dem ersten Blick impliziert dies, dass eine größere Batchgröße grundsätzlich zu bevorzugen ist. Aufgrund von bestehender Parallelisierung und von Begrenzungen der Größe des Stapels gibt es allerdings eine größte Batch-Größe, nach der Performanz verloren geht. Experimentell hat sich 20 als brauchbare Batchgröße ergeben.

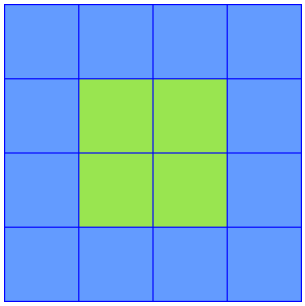

Abbildung 3.3.: Für 4 Werte müssen 16 verschiedene Felder betrachtet werden. Wenn die beiden Reihen getrennt betrachtet werden, werden 24 Felder betrachtet.

4. Ergebnisse

Die Ergebnisse habe ich mithilfe der Performanztests erfasst, da diese im Release-Modus ausgeführt werden und sie die tatsächliche Ausführungszeit von außen messen. Für diese Tests habe ich die Dateien SETSM_WV01_20121008_102001001E9D1100_-102001001E6C0C00_seg1_2m_v2.0_dem.tif von Copernicus.eu (ein kleiner Ausschnitt Europas) mit einer Größe von 1.06 GB und srtm_germany_dsm.tif von opendm.info (eine Deutschlandkarte) mit einer Größe von 272 MB verwendet.

Insgesamt habe ich 4 Ansätze implementiert: Funktionszeiger (abgekürzt FZ), Sliding Matrix (abgekürzt SM), Sliding Matrix Generalisiert (abgekürzt SMG) und Sliding Matrix Generalisiert Batched (abgekürzt SMGB). Zusätzlich wurde die Basisversion (Abgekürzt BV) betrachtet. Diese habe ich auf einem Supercomputer-Cluster des DKRZ auf ihre Performanz getestet.

Die erste Auffälligkeit ist, dass die Performanz aller Tests für Sliding Matrix schlechter als die der Basisversion ist. Dies ist höchst ungewöhnlich, da das Tool Convergence in dieser Version unverändert ist. (Siehe Tabelle 4.1.)

Weiterhin zeigt sich aber, dass die meisten Tools eine messbare Verbesserung im Bereich von bis zu 50% erfahren haben, als in Sliding Matrix Generalisiert die Gleitende Matrix auf sie angewendet wurde. Nur wenige Tools zeigen eine Verschlechterung der Performanz in der Größenordnung weniger Prozent. (Siehe Tabelle 4.2.)

Schließlich zeigt sich noch eine kleinere Verbesserung von Sliding Matrix Generalisiert zu Sliding Matrix Generalisiert Batched im Bereich von bis zu 10%. (Siehe Tabelle 4.3.)

Auffällig ist, dass Funktionszeiger eine Verbesserung darstellt. Diesen Ansatz habe ich eigentlich verworfen und nie für Convergence implementiert, da auf meinem persönlichen Rechner keine Verbesserung messbar war. Auf dem Cluster zeigt sich aber eine deutlich Verbesserung in einer Größenordnung ähnlich Sliding Matrix Generalisiert mit einer deutlich kleineren und allgemeineren Code-Änderung.(Siehe Tabelle 4.4.)

Test	Datei	BV	SM	Verbesserung
SAC: maximum triangle slope (Tarboton 1997)	opendm.info	11.03s	12.92s	-17.23%
	Copernicus.eu	69.45s	78.76s	-13.40%
Convergence Index: Grid Files, 3x3, Aspect	opendm.info	20.80s	33.46s	-60.89%
	Copernicus.eu	133.61s	168.60s	-26.19%
SAC: least squares fitted plane (Horn 1981, Costa-Cabral & Burgess 1996)	opendm.info	10.89s	12.42s	-14.06%
	Copernicus.eu	70.32s	78.36s	-11.42%
SAC: 9 parameter 2nd order polynom (Zevenbergen & Thorne 1987)	opendm.info	11.12s	12.03s	-8.26%
	Copernicus.eu	69.46s	76.85s	-10.63%
Convergence Index: Grid Files, 2x2, Gradient	opendm.info	21.41s	30.84s	-44.05%
	Copernicus.eu	134.67s	148.81s	-10.50%
Convergence Index: Grid Files, 2x2, Aspect	opendm.info	11.87s	22.98s	-93.66%
	Copernicus.eu	78.93s	103.35s	-30.94%
Convergence Index: Grid Files, 3x3, Gradient	opendm.info	41.82s	54.84s	-31.13%
	Copernicus.eu	254.78s	260.05s	-2.07%
SAC: 6 parameter 2nd order polynom (Evans 1979)	opendm.info	11.31s	12.54s	-10.88%
	Copernicus.eu	69.10s	77.07s	-11.53%
SAC: 10 parameter 3rd order polynom (Florinsky 2009)	opendm.info	14.29s	23.86s	-67.04%
	Copernicus.eu	91.35s	123.40s	-35.09%
SAC: 10 parameter 3rd order polynom (Haralick 1983)	opendm.info	15.96s	22.99s	-44.00%
	Copernicus.eu	105.05s	122.82s	-16.92%
SAC: 6 parameter 2nd order polynom (Bauer, Rohdenburg, Bork 1985)	opendm.info	11.84s	12.08s	-2.02%
	Copernicus.eu	74.43s	79.79s	-7.20%
SAC: 6 parameter 2nd order polynom (Heerdegen & Beran 1982)	opendm.info	11.21s	12.12s	-8.05%
	Copernicus.eu	71.03s	76.98s	-8.37%
SAC: maximum slope (Travis et al. 1975)	opendm.info	11.67s	11.95s	-2.38%
	Copernicus.eu	68.85s	75.85s	-23%
Durchschnitt	-	-	-	-23.00%

Abbildung 4.1.: Die erste Implementierung der Gleitenden Matrix hat seltsamerweise in allen Tools die Performanz beeinträchtigt, selbst in nicht modifizierten Tools!

Test	Datei	BV	SMG	Verbesserung
SAC: maximum triangle slope (Tarboton 1997)	opendm.info	11.03s	10.57s	4.10%
	Copernicus.eu	69.45s	66.65s	4.03%
Convergence Index: Grid Files, 3x3, Aspect	opendm.info	20.80s	15.38s	26.03%
	Copernicus.eu	133.61s	72.57s	45.69%
SAC: least squares fitted plane (Horn 1981, Costa-Cabral & Burgess 1996)	opendm.info	10.89s	10.52s	3.41%
	Copernicus.eu	70.32s	71.23s	-1.29%
SAC: 9 parameter 2nd order polynom (Zevenbergen & Thorne 1987)	opendm.info	11.12s	10.29s	7.40%
	Copernicus.eu	69.46s	70.25s	-1.14%
Convergence Index: Grid Files, 2x2, Gradient	opendm.info	21.41s	16.74s	21.79%
	Copernicus.eu	134.67s	75.64s	43.83%
Convergence Index: Grid Files, 2x2, Aspect	opendm.info	11.87s	8.93s	24.74%
	Copernicus.eu	78.93s	52.06s	34.04%
Convergence Index: Grid Files, 3x3, Gradient	opendm.info	41.82s	31.17s	25.48%
	Copernicus.eu	254.78s	111.18s	56.36%
SAC: 6 parameter 2nd order polynom (Evans 1979)	opendm.info	11.31s	10.33s	8.65%
	Copernicus.eu	69.10s	72.85s	-5.43%
SAC: 10 parameter 3rd order polynom (Florinsky 2009)	opendm.info	14.29s	11.19s	21.65%
	Copernicus.eu	91.35s	75.40s	17.46%
SAC: 10 parameter 3rd order polynom (Haralick 1983)	opendm.info	15.96s	11.74s	26.43%
	Copernicus.eu	105.05s	76.15s	27.50%
SAC: 6 parameter 2nd order polynom (Bauer, Rohdenburg, Bork 1985)	opendm.info	11.84s	9.89s	16.48%
	Copernicus.eu	74.43s	68.10s	8.51%
SAC: 6 parameter 2nd order polynom (Heerdegen & Beran 1982)	opendm.info	11.21s	9.85s	12.20%
	Copernicus.eu	71.03s	68.15s	4.06%
SAC: maximum slope (Travis et al. 1975)	opendm.info	11.67s	10.35s	11.27%
	Copernicus.eu	68.85s	72.50s	-5.30%
Durchschnitt	-	-	-	9.33%

Abbildung 4.2.: Das Generalisieren der Gleitenden Matrix hat in den meisten Fällen zu einer deutlichen Performanzverbesserung geführt. Nur wenige Ausführungen wurden langsamer, und in deutlich kleinerem Ausmaß.

Test	Datei	SMG	SMGB	Verbesserung
SAC: maximum triangle slope (Tarboton 1997)	opendm.info	10.57s	10.00s	5.45%
	Copernicus.eu	66.65s	64.57s	3.12%
Convergence Index: Grid Files, 3x3, Aspect	opendm.info	15.38s	14.97s	2.67%
	Copernicus.eu	72.57s	68.34s	5.82%
SAC: least squares fitted plane (Horn 1981, Costa-Cabral & Burgess 1996)	opendm.info	10.52s	10.10s	3.95%
	Copernicus.eu	71.23s	69.19s	2.86%
SAC: 9 parameter 2nd order polynom (Zevenbergen & Thorne 1987)	opendm.info	10.29s	9.70s	5.80%
	Copernicus.eu	70.25s	66.17s	5.81%
Convergence Index: Grid Files, 2x2, Gradient	opendm.info	16.74s	16.50s	1.45%
	Copernicus.eu	75.64s	73.63s	2.66%
Convergence Index: Grid Files, 2x2, Aspect	opendm.info	8.93s	8.48s	5.05%
	Copernicus.eu	52.06s	49.07s	5.75%
Convergence Index: Grid Files, 3x3, Gradient	opendm.info	31.17s	30.97s	0.63%
	Copernicus.eu	111.18s	109.73s	1.30%
SAC: 6 parameter 2nd order polynom (Evans 1979)	opendm.info	10.33s	9.65s	6.59%
	Copernicus.eu	72.85s	66.94s	8.11%
SAC: 10 parameter 3rd order polynom (Florinsky 2009)	opendm.info	11.19s	10.24s	8.49%
	Copernicus.eu	75.40s	67.18s	10.90%
SAC: 10 parameter 3rd order polynom (Haralick 1983)	opendm.info	11.74s	10.92s	7.02%
	Copernicus.eu	76.15s	70.96s	6.82%
SAC: 6 parameter 2nd order polynom (Bauer, Rohdenburg, Bork 1985)	opendm.info	9.89s	9.08s	8.16%
	Copernicus.eu	68.10s	63.33s	6.99%
SAC: 6 parameter 2nd order polynom (Heerdegen & Beran 1982)	opendm.info	9.85s	9.14s	7.17%
	Copernicus.eu	68.15s	63.20s	7.26%
SAC: maximum slope (Travis et al. 1975)	opendm.info	10.35s	9.40s	9.16%
	Copernicus.eu	72.50s	64.82s	10.59%
Durchschnitt	-	-	-	5.75%

Abbildung 4.3.: Batching hat kleine, aber verlässliche Performanzgewinne erzielt.

Test	Datei	BV	FZ	Verbesserung
SAC: maximum triangle slope (Tarboton 1997)	opendm.info	11.03s	10.33s	6.30%
	Copernicus.eu	69.45s	62.83s	9.53%
SAC: least squares fitted plane (Horn 1981, Costa-Cabral & Burgess 1996)	opendm.info	10.89s	9.82s	9.76%
	Copernicus.eu	70.32s	59.47s	15.44%
SAC: 9 parameter 2nd order polynom (Zevenbergen & Thorne 1987)	opendm.info	11.12s	10.42s	6.24%
	Copernicus.eu	69.46s	60.11s	13.47%
SAC: 6 parameter 2nd order polynom (Evans 1979)	opendm.info	11.31s	10.55s	6.71%
	Copernicus.eu	69.10s	59.97s	13.22%
SAC: 10 parameter 3rd order polynom (Florinsky 2009)	opendm.info	14.29s	13.48s	5.67%
	Copernicus.eu	91.35s	83.04s	9.09%
SAC: 10 parameter 3rd order polynom (Haralick 1983)	opendm.info	15.96s	15.27s	4.33%
	Copernicus.eu	105.05s	97.65s	7.04%
SAC: 6 parameter 2nd order polynom (Bauer, Rohdenburg, Bork 1985)	opendm.info	11.84s	10.47s	11.53%
	Copernicus.eu	74.43s	60.16s	19.18%
SAC: 6 parameter 2nd order polynom (Heerdegen & Beran 1982)	opendm.info	11.21s	10.51s	6.25%
	Copernicus.eu	71.03s	60.11s	15.37%
SAC: maximum slope (Travis et al. 1975)	opendm.info	11.67s	10.83s	7.19%
	Copernicus.eu	68.85s	61.00s	11.41%
Durchschnitt	-	-	-	9.87%

Abbildung 4.4.: Die Implementierung von Funktionszeigern hat unverhoffte Performanzgewinne erzielt.

5. Fazit

Die Implementierung einer gleitenden Matrix verbessert die Performanz der Tools Slope, Aspect, Curvature und Convergence um ungefähr 15%, mit einzelnen Verbesserungen von bis zu 50%. Es haben sich Schwierigkeiten solcher Änderungen ergeben, wie z.B. Performanzverluste ohne erkennbaren Grund und Gewinne, die weit hinter der Theorie zurückbleiben. Diese werfen Fragen zu den praktischen Konsequenzen Performanz-orientierter Änderungen an SAGA-GIS auf. Insgesamt zeigt sich, dass die Software SAGA-GIS performanter gestaltet werden kann, ohne tiefere Eingriffe in den Aufbau der Software vorzunehmen und ohne die bestehende Funktionalität zu beeinträchtigen.

Die im Rahmen dieses Projektes geleistete Arbeit kann als Grundlage für zukünftige Performanz-orientierte Projekte dienen. Die Regressionstests geben Absicherung für weitere Experimente und die gleitende Matrix kann weiter analysiert und optimiert werden. Insbesondere die unerwarteten Gewinne durch Funktionszeiger und die optimale Batch-Größe können konkret untersucht werden.

A. Verwendete Hardware

- Intel i7 2100k - 3.4 GHz, 4 Physical Cores, 8 Logic Cores[4]
- 16 GB DDR3-RAM
- P67A-GD53 Motherboard

Auf dieser Hardware lief Windows Version 10.0.19042 Build 19042. Während der Ausführung liefen außerdem verschiedene Programme im Hintergrund.

Die Ergebnisse wurden auf einem internen Cluster des Arbeitsbereichs Wissenschaftliches Rechnen der Universität Hamburg ausgeführt, die auf der Westmere-Architektur basieren.

B. Notizen zur Ausführung

Die Angaben zur Ausführungszeit beziehen sich auf die Ausführung mithilfe der Python-Schnittstelle. Die Messung erfolgt mit dem Python Paket TIME. Der exakte Testcode zum Messen der Ausführungszeit ist unten aufgeführt und kommentiert.

```
1  # "test" beschreibt eine Testausfuehrung.
2  # In dieser Methode wird ein bestimmter Test
3  # ausgefuehrt und die Ausfuehrungszeit analysiert.
4  def run_test(test):
5    test.create_tool(saga_wrapper.create_tool)
6    test_result = {}
7    # Es wird fuer jede Testdatei einzeln gemessen
8    for file in get_input_files(test):
9      try:
10       test_result[file] = []
11       data = saga_wrapper.load_data(DATA_FOLDER + file)
12       # "iterations" wird meistens auf 10 gesetzt.
13       for i in range(iterations):
14         # Die Differenz von Start- und Endzeit
15         # ist die Ausfuehrungszeit des Tools selber,
16         # plus die Ausfuehrungszeit von time.time(),
17         # plus die Zeit, die in der API
18         # und den Python-Skripten verbracht werden.
19         # Diese zustaetzlichen Zeiten
20         # werden als nahezu konstant angenommen.
21         start = time.time()
22         ret_data =
23             test.run_tool(data, saga_wrapper.execute)
24         end = time.time()
25         for d in ret_data:
26           saga_wrapper.delete_data(d)
27         # Alle Ergebnisse werden gesammelt
28         test_result[file].append(end - start)
29       saga_wrapper.delete_data(data)
30     except Exception, e:
31       test_result[file] = e
```

```
32  test.delete_tool(saga_wrapper.delete_tool)
33  return test_result
```

C. Verworfene Ansätze

Auf dem Weg zur finalen Version habe ich einige Ansätze ausprobiert und verworfen, da sie keine erkennbaren Verbesserungen brachten. Aus diesen habe ich kaum Erkenntnisse über ihre Erfolglosigkeit hinaus gewonnen. Da diese Erfolglosigkeit für spätere Arbeiten im Bereich dennoch relevant sein können, sind sie hier kurz beschrieben.

C.1. Iterationsreihenfolge

In den betrachteten Tools wird über beide Axen der Matrix iteriert. Die Reihenfolge dieser Iteration kann relevant sein, da auf das Speicherlayout zugeschnittene Speicherzugriffe schneller sind. Hier stellte sich heraus, dass das Tool bereits die effektivere Reihenfolge verwendete.

C.2. Algorithmusanalyse

Ich habe versucht, Stellen im Algorithmus zu finden, die optimierbar sind. Zum Beispiel könnten hypothetisch durch Grenzwert-Überprüfungen manche Berechnung eingespart werden. Es stellte sich jedoch schnell heraus, dass mir das tiefere Verständnis für die komplexen geographischen Algorithmen fehlte, um eine Performanzverbesserung zu finden, bei der die Funktionalität nicht beeinträchtigt ist.

Literatur

[1] Paul Ammann und Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[2] O. Conrad u. a. *SAGA GIS*. 2020. URL: https://sourceforge.net/projects/saga-gis/ (besucht am 10. 12. 2020).

[3] O. Conrad u. a. "System for Automated Geoscientific Analyses (SAGA) v. 2.1.4". In: *Geoscientific Model Development* 8.7 (2015), S. 1991–2007. DOI: 10.5194/gmd-8-1991-2015. URL: https://gmd.copernicus.org/articles/8/1991/2015/.

[4] Intel Corporation. *Intel i7 2600k Datenblatt*. URL: https://ark.intel.com/content/www/de/de/ark/products/52214/intel-core-i7-2600k-processor-8m-cache-up-to-3-80-ghz.html (besucht am 02. 03. 2021).