Realisierung eines Mehrgitterverfahrens in OpenCL

Christian-Albrechts-Universität zu Kiel
Technische Fakultät
Institut für Informatik

Bachelorarbeit von

Jakob Schikowski

Betreut durch:	Prof. Dr. Steffen Börm
Matrikelnummer:	006809
Vorgelegt am:	25. September 2013

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Inhaltsverzeichnis

1. Einleitung

Diese Bachelorarbeit beschäftigt sich mit der Umsetzung des Mehrgitterverfahrens auf Grafikprozessoren. Das Mehrgitterverfahren ist eine Klasse von Algorithmen zur Lösung linearer Gleichungssysteme. Es bietet im Gegensatz zu anderen Verfahren eine von der Feinheit des Gitters unabhängige Konvergenzgeschwindigkeit. Wir setzen uns mit einem Fallbeispiel auseinander, zu dessen Lösung wir das Mehrgitterverfahren verwenden. Dazu realisieren wir das Verfahren in Software, um eine Näherung zu bestimmen. Wir beginnen mit einer Realisierung für gewöhnliche Hauptprozessoren, die in den meisten privaten Computern verbaut und leicht zu programmieren sind. Da Hauptprozessoren in den meisten Fällen Programme ausführen, die wenig Parallelität und viele Fallunterscheidungen besitzen, sind sie für diese Art von Programmen optimiert worden. Bei näherer Betrachtung des Mehrgitterverfahrens wird allerdings deutlich, dass zur Berechnung im Wesentlichen wenige Funktionen viele tausend Male ausgeführt werden müssen. Diese Berechnungen lassen sich auch parallel durchführen. Daher werden wir OpenCL einsetzen, einen offenen Standard zur Ansteuerung von Parallelrechnern. Es existieren OpenCL-Implementierungen für unterschiedliche Hardwaretypen, unter anderem für Grafikprozessoren, die die benötigten Rechenwerke und eine ausreichend große Speicheranbindung besitzen, um viele gleichartige Berechnungen parallel ausführen zu können. Außerdem sind sie durch ihre hohe Verbreitung sehr erschwinglich. Wir realisieren daher das Mehrgitterverfahren in OpenCL und optimieren es für Grafikprozessoren. Dabei vergleichen wir die benötigte Rechenzeit für Haupt- und Grafikprozessoren. Angesichts der Rechenleistung eines Grafikprozessors ist das Ziel dieser Arbeit eine mindestens viermal schnellere Berechnung durch den Grafikprozessor.

2. Problemstellung

2.1. Mehrgitterverfahren

Viele Zusammenhänge in der Natur lassen sich durch partielle Differentialgleichungen beschreiben. Diese zu lösen ist allerdings kein triviales Problem. Im linearen Fall kann man durch Diskretisierung einer Differentialgleichung ein lineares Gleichungssystem aufstellen, das sich mit numerischen Verfahren näherungsweise lösen lässt. Die Qualität der Lösung hängt hierbei von der bei der Diskretisierung gewählten Feinheit des Gitters ab. Iterative Verfahren haben hierbei häufig das Problem, dass bei zunehmender Feinheit nicht nur der Aufwand für einen Iterationsschritt steigt, sondern dass auch die Konvergenzgeschwindigkeit abnimmt.

Mit Mehrgitterverfahren lassen sich lineare Gleichungssysteme mit einer von der Feinheit des Gitters unabhängigen Konvergenzgeschwindigkeit näherungsweise lösen lassen [1, Satz 4.11]. Die Idee dazu ist, den aktuellen Fehler auf einem Gitter mit gegebener Feinheit auf einem gröberen Gitter zu approximieren. Dabei wird ausgenutzt, dass sich das gröbere Gitter günstiger lösen lässt. Dies lässt sich rekursiv auch für die gröberen Gitter bis zu einer gewünschten Grobheit wiederholen. Im Detail benötigen wir für das Mehrgitterverfahren für jede Gitterstufe

- eine Restriktion R, die das aktuelle Residuum auf das gröbere Gitter überträgt,
- einen Glätter G, der den Fehler approximiert,
- und schließlich eine Prolongation P, die die Näherung des groben Gitters zurück auf das feine Gitter überträgt.

Ein möglicher Verlauf der Anwendung eines Mehrgitterverfahrens ist in Abbildung 2.1 zu sehen.

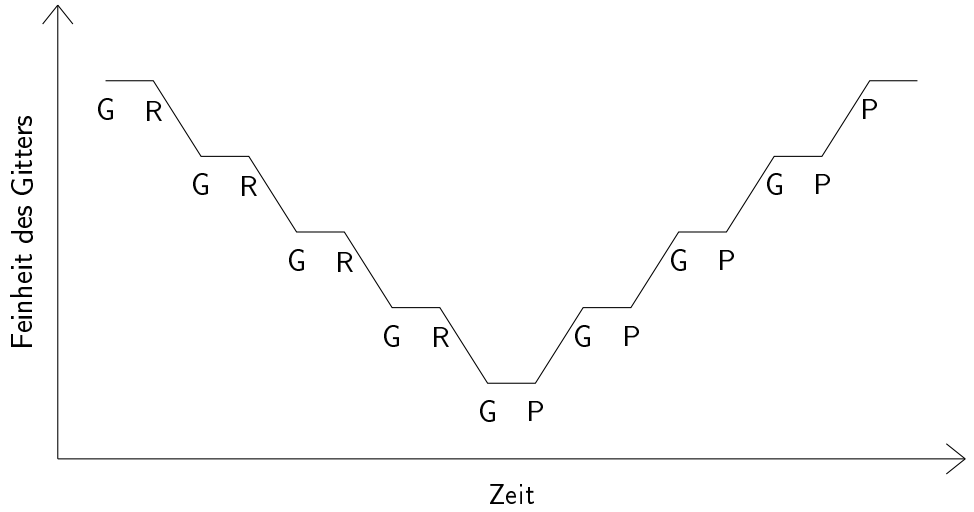


Abbildung 2.1.: Mehrgitterverfahren mit V-Zyklus. Abkürzungen: **R**estriktion, **G**lättung, **P**rolongation.

2.2. Fallbeispiel

Im Folgenden betrachten wir für $m \in \{2^n - 1 \mid n \in \mathbb{N}_{>0}\}$ ein lineares Gleichungssystem gegeben durch die Unbekannten

$$X = \begin{pmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,m\text{-}1} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,m\text{-}1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m\text{-}1,0} & x_{m\text{-}1,1} & \cdots & x_{m\text{-}1,m\text{-}1} \end{pmatrix} \in \mathbb{R}^{m,m} ,$$

die Konstanten

$$B = \begin{pmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,m\text{-}1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,m\text{-}1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m\text{-}1,0} & b_{m\text{-}1,1} & \cdots & b_{m\text{-}1,m\text{-}1} \end{pmatrix} \in \mathbb{R}^{m,m}$$

und den Gleichungen

$$b_{j,i} = 4x_{j,i} - x_{j+1,i} - x_{j\text{-}1,i} - x_{j,i+1} - x_{j,i\text{-}1} , \tag{2.1}$$

wobei für die Randfälle gilt $x_{j,i} = 0$ für $i \notin \{0, \cdots, m-1\}$ oder $j \notin \{0, \cdots, m-1\}$. Ziel ist es nun, zu einem vorgegebenen B das X zu bestimmen. Der Fehler eines beliebigen $Y \in \mathbb{R}^{m,m}$ zu X lässt sich reduzieren durch die Anwendung des Glätters gegeben durch

$$y'_{j,i} = \frac{1}{4}(b_{j,i} + y_{j+1,i} + y_{j\text{-}1,i} + y_{j,i+1} + y_{j,i\text{-}1}) . \tag{2.2}$$

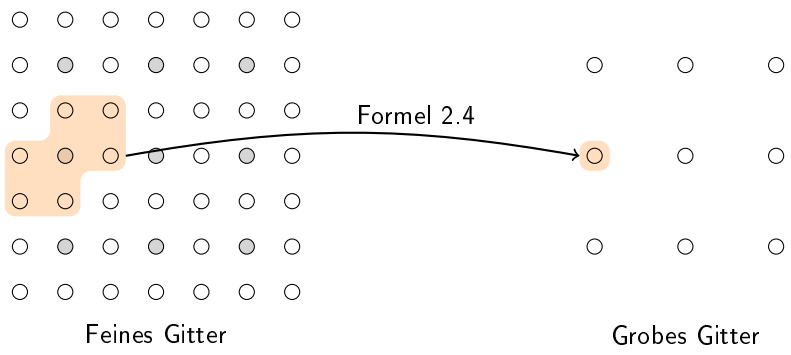


Abbildung 2.2.: Veranschaulichung der Restriktion nach Formel 2.4.

Am Glätter lässt sich erkennen, dass die „Information“ einer Unbekannten $y_{j,i}$ in einem Glättungsschritt immer nur an die direkten Nachbarn $y'_{j+1,i}, y'_{j\text{-}1,i}, y'_{j,i+1}$ und $y'_{j,i\text{-}1}$ weitergegeben wird. Somit benötigt die Information im ungünstigsten Fall $2 \cdot m$ Glättungsschritte, um alle Unbekannten zu erreichen. Insbesondere steigt dieser Wert proportional mit der Feinheit des Gitters, womit auch die Konvergenzgeschwindigkeit abnimmt.

Daher kommt das Mehrgitterverfahren zum Einsatz, das ein deutlich besseres Konvergenzverhalten aufweist. Sei dazu für ein $m_n \in \{2^n - 1 \mid n \in \mathbb{N}_{>0}\}$ ein Gleichungssystem durch $X^n, B^n \in \mathbb{R}^{m_n, m_n}$ gegeben, das ein Gitter einer bestimmten Feinheit darstellt. Sofern $m_n > 1$ gilt, sei das zu n nächst gröbere Gitter definiert durch $X^{n+1}, B^{n+1} \in \mathbb{R}^{m_{n+1}, m_{n+1}}$ mit $m_{n+1} = \left\lfloor \frac{m_n}{2} \right\rfloor$. Das Residuum von Y^n ist gegeben durch

$$d^n_{j,i} = b^n_{j,i} + y^n_{j+1,i} + y^n_{j\text{-}1,i} + y^n_{j,i+1} + y^n_{j,i\text{-}1} - 4y^n_{j,i}, \tag{2.3}$$

wobei für die Randfälle wieder gilt $y_{j,i} = 0$. Das Residuum lässt sich nun durch eine Restriktion auf das nächst gröbere Gitter transferieren. Dazu definieren wir $Y^{n+1} = \mathsf{O}$ und B^{n+1} durch

$$\begin{aligned} b^{n+1}_{j,i} = -d^n_{2j+1,2i+1} - \frac{1}{2}(&d^n_{2j,2i+1} + d^n_{2j+2,2i+1} \\ &+ d^n_{2j+1,2i} + d^n_{2j+1,2i+2} \\ &+ d^n_{2j,2i+2} + d^n_{2j+2,2i}). \end{aligned} \tag{2.4}$$

Durch Anwendung des Glätters bzw. durch rekursive Anwendung des Mehrgitterverfahrens lässt sich nun aus Y^{n+1}, B^{n+1} eine exaktere Lösung $Y^{n+1\prime}$ für das grobe Gitter bestimmen. Falls $m_{n+1} = 1$ gilt, ergibt sich die exakte Lösung direkt und lautet $y^{n+1\prime}_{0,0} = \frac{1}{4} b^{n+1\prime}_{0,0}$. Schließlich können wir nun den approximierten Fehler durch Anwendung einer Prolongation zurück auf das feinere Gitter übertragen. Mit Hilfe

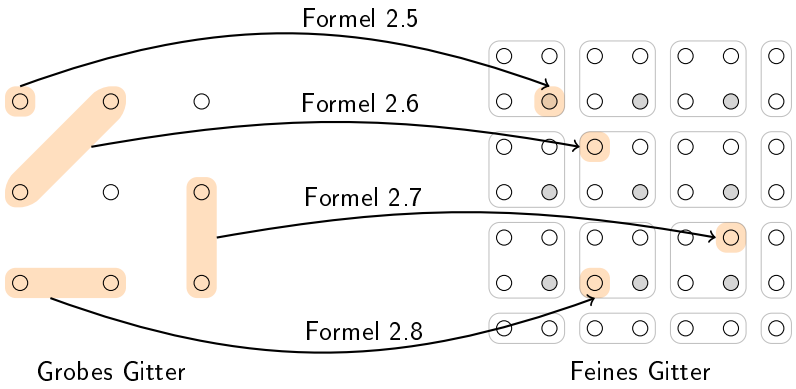


Abbildung 2.3.: Veranschaulichung der Prolongation nach den Formeln 2.5 bis 2.8.

von $Y^{n+1'}$ und Y^n definieren wir $Y^{n\prime}$ als

$$y^{n\prime}_{2j,2i} = y^n_{2j,2i} - \frac{1}{2}(y^{n+1'}_{j\text{-}1,i} + y^{n+1'}_{j,i\text{-}1}) \tag{2.5}$$

$$y^{n\prime}_{2j,2i+1} = y^n_{2j,2i+1} - \frac{1}{2}(y^{n+1'}_{j\text{-}1,i} + y^{n+1'}_{j,i}) \tag{2.6}$$

$$y^{n\prime}_{2j+1,2i} = y^n_{2j+1,2i} - \frac{1}{2}(y^{n+1'}_{j,i\text{-}1} + y^{n+1'}_{j,i}) \tag{2.7}$$

$$y^{n\prime}_{2j+1,2i+1} = y^n_{2j+1,2i+1} - y^{n+1'}_{j,i}, \tag{2.8}$$

wobei auch hier für die Randfälle gilt $y^{n+1'}_{j,i} = 0$. Im Allgemeinen stellt $Y^{n\prime}$ eine exaktere Lösung des Gleichungssystems der feinen Gitterstufe dar. Durch wiederholte Anwendung dieses Verfahrens lässt sich die Näherung der exakten Lösung X^n stetig verbessern.

3. Mehrgitterverfahren auf dem Hauptprozessor

3.1. Zielsetzung

Ziel ist es nun, entsprechend Kapitel 2.2 das Mehrgitterverfahren in Software zu implementieren. Dabei soll die Software mit Probleminstanzen beliebiger Größe umgehen können. Bei der Implementierung des Mehrgitterverfahrens verzichten wir auf einen Vorglätter, um später bei der OpenCL-Implementierung Optimierungen der Restriktion zu ermöglichen. Abbildung 3.1 zeigt den Verlauf des Mehrgitterverfahrens ohne Vorglätter. Die Anzahl der Anwendungen des Glätters bei der Glättung soll frei wählbar sein. Als Einstieg beginnen wir mit einer simplen Implementierung in der Programmiersprache C, um das Verfahren sequentiell auf einem gewöhnlichen Hauptprozessor, den man als *Central Processing Unit* (CPU) bezeichnet, berechnen zu lassen.

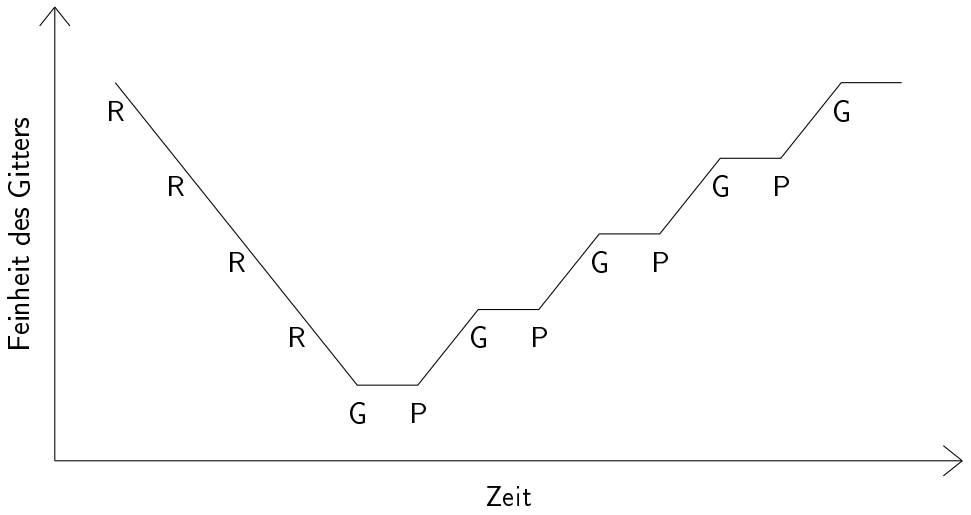


Abbildung 3.1.: Mehrgitterverfahren ohne Vorglätter. Abkürzungen: **R**estriktion, **G**lättung, **P**rolongation

3.2. Datenstruktur

Zuallererst muss geklärt werden, wie die Daten, mit denen gerechnet wird, sinnvoll im Arbeitsspeicher abgelegt werden. Es sei eine Probleminstanz nach Kapitel 2.2 gegeben, wobei B^0, X^0 das feinste Gitter sei. Zur Speicherung verwenden wir zweidimensionale Arrays, die zeilenweise angeordnet sind. Als Datentyp nutzen wir Gleitkommazahlen einfacher Genauigkeit. Wir benötigen für jede Feinheit n des Gitters ein Array für B^n und zwei für X^n, wobei eines für die aktuelle Näherung von X^n und eines für Ergebnisse genutzt wird. Zur Behandlung der Randfälle erhalten die Arrays für X^n zusätzliche Elemente an den Rändern, die mit den Randwerten, in unserem Fall 0, initialisiert werden.

Zur Realisierung allozieren wir einen großen Speicherbereich, in dem wir alle benötigten Arrays ablegen. Zum Zugriff auf die einzelnen Arrays merken wir uns die jeweilige Startadresse und Zeilenbreite. Dazu verwenden wir für jede Gitterstufe eine Instanz der folgenden Struktur, die alle relevanten Informationen beinhaltet.

```
1  struct _grid {
2    int    w;       /* Breite des Gitters        */
3    int    h;       /* Höhe des Gitters          */
4    int    x_off0; /* Startadresse Quellarray X */
5    int    x_off1; /* Startadresse Zielarray  X */
6    int    x_lnw;  /* Zeilenbreite Array X      */
7    int    b_off;  /* Startadresse Array B      */
8    int    b_lnw;  /* Zeilenbreite Array B      */
9    float  *mem;   /* Angelegter Speicher       */
10 };
11 typedef struct _grid grid_t;
12 typedef grid_t *pgrid_t;
```

Zu Beginn seien das B- und X-Quellarray der feinsten Gitterstufe nach Belieben zu initialisieren. Wir legen fest, dass am Anfang des Mehrgitterverfahrens für die gröberen Gitter das X-Quellarray und damit deren aktuelle Näherung stets mit Nullen zu initialisieren ist.

3.3. Glätter

Wir wollen nun den Glätter nach Formel 2.2 realisieren. Dazu ist ein Gitter `grid` gegeben. Wir berechnen nun nacheinander jedes Element der neuen Näherung unter Zuhilfenahme des B-Arrays und X-Quellarrays und speichern die neue Näherung im X-Zielarray. Aufgrund der Anordnung der Arrays verwenden wir dazu zwei Schleifen, die zeilenweise das X-Zielarray durchlaufen. Die Implementierung hierzu ist im Folgenden zu sehen.

```
1  void cpu_smoothing_step(pgrid_t grid) {
2    int i, j, w, h, x_lnw, b_lnw;
3    float *xs, *xd, *b;
```

```
4
5   /* Initialisierung */
6   w          = grid->w;
7   h          = grid->h;
8   xs         = grid->mem + grid->x_off0 + grid->x_lnw + 1;
9   xd         = grid->mem + grid->x_off1 + grid->x_lnw + 1;
10  x_lnw      = grid->x_lnw;
11  b          = grid->mem + grid->b_off;
12  b_lnw      = grid->b_lnw;
13
14  /* Glätter */
15  for(j = 0; j < h; j++) {
16    for(i = 0; i < w; i++) {
17      xd[i + x_lnw * j] =
18             (b[i      + b_lnw * j    ]
19           + xs[(i-1) + x_lnw * j    ]
20           + xs[(i+1) + x_lnw * j    ]
21           + xs[i     + x_lnw * (j-1)]
22           + xs[i     + x_lnw * (j+1)]) * 0.25f;
23    }
24  }
25
26  swap(&grid->x_off0, &grid->x_off1);
27 }
28
29 void cpu_smoothing(pgrid_t grid, int smoothing_cnt) {
30   int k;
31   for(k = 0; k < smoothing_cnt; k++) {
32     cpu_smoothing_step(grid);
33   }
34 }
```

Ab Zeile 1 wird eine Funktion implementiert, die einen einzelnen Glättungsschritt durchführt, während die Funktion ab Zeile 29 eine gewünschte Anzahl an Glättungsschritten durchführt. Die eigentliche Glättung befindet sich in Zeile 15 bis 24. Da wir zusätzlichen Speicher für die Randfälle vorgesehen und ihn mit Nullen initialisiert haben, müssen wir nun beim Zugriff auf das X-Quellarray die Randfälle nicht separat behandeln. In Zeile 26 werden schließlich das X-Quellarray und X-Zielarray vertauscht, so dass sich die aktuelle Näherung stets im X-Quellarray befindet.

3.4. Restriktion

Für die Restriktion müssen wir zunächst für das feinere Gitter das Residuum nach Formel 2.3 berechnen und anschließend die Restriktion nach Formel 2.4 anwenden. Dazu sei je Restriktionsschritt das feinere Gitter `fgrid` und das gröbere Gitter `cgrid` gegeben. Die entsprechende Implementierung sieht wie folgt aus.

```
1 void cpu_restriction(pgrid_t fgrid, pgrid_t cgrid) {
```

```
2   int i, j, di, dj, f_w, f_h, f_x_lnw, f_d_lnw, f_b_lnw,
3       c_w, c_h, c_b_lnw;
4   float *f_x, *f_d, *f_b, *c_b;
5
6   /* Initialisierung */
7   f_w      = fgrid->w;
8   f_h      = fgrid->h;
9   f_x      = fgrid->mem + fgrid->x_off0 + fgrid->x_lnw + 1;
10  f_x_lnw  = fgrid->x_lnw;
11  f_d      = fgrid->mem + fgrid->x_off1 + fgrid->x_lnw + 1;
12  f_d_lnw  = fgrid->x_lnw;
13  f_b      = fgrid->mem + fgrid->b_off;
14  f_b_lnw  = fgrid->b_lnw;
15  c_w      = cgrid->w;
16  c_h      = cgrid->h;
17  c_b      = cgrid->mem + cgrid->b_off;
18  c_b_lnw  = cgrid->b_lnw;
19
20  /* Residuum berechnen */
21  for(j = 0; j < f_h; j++) {
22    for(i = 0; i < f_w; i++) {
23      f_d[i + f_d_lnw * j] =
24            f_b[i     + f_b_lnw * j    ]
25          + f_x[(i-1) + f_x_lnw * j    ]
26          + f_x[(i+1) + f_x_lnw * j    ]
27          + f_x[i     + f_x_lnw * (j-1)]
28          + f_x[i     + f_x_lnw * (j+1)]
29          - f_x[i     + f_x_lnw * j    ] * 4.0f;
30    }
31  }
32
33  /* Restriktion */
34  for(j = 0; j < c_h; j++) {
35    dj = 2 * j;
36    for(i = 0; i < c_w; i++) {
37      di = 2 * i;
38      c_b[i + c_b_lnw * j] =
39          -  f_d[(di+1) + f_d_lnw * (dj+1)]
40          - (f_d[di     + f_d_lnw * (dj+1)]
41          +  f_d[di     + f_d_lnw * (dj+2)]
42          +  f_d[(di+1) + f_d_lnw * (dj+2)]
43          +  f_d[(di+2) + f_d_lnw * (dj+1)]
44          +  f_d[(di+2) + f_d_lnw * dj    ]
45          +  f_d[(di+1) + f_d_lnw * dj    ]) * 0.5f;
46    }
47  }
48 }
```

In Zeile 21 bis 31 wird das Residuum berechnet und dazu im X-Zielarray des feineren Gitters gespeichert. Ab Zeile 34 wird schließlich die Restriktion durchgeführt

und das B-Array des gröberen Gitters berechnet. Das berechnete Residuum wird anschließend nicht mehr benötigt und daher verworfen.

Da wir nach der Restriktion als initiale Näherung des gröberen Gitters das X-Quellarray mit Nullen initialisieren, lässt sich bei einer Gitterstufe mit $n > 0$ die Berechnung des Residuum reduzieren auf $d^n_{j,i} = b^n_{j,i}$, wodurch sich die Formel zur Restriktion vereinfachen lässt zu

$$\begin{aligned} b^{n+1}_{j,i} = -b^n_{2j+1,2i+1} - \frac{1}{2}(&b^n_{2j,2i+1} + b^n_{2j+2,2i+1} \\ &+ b^n_{2j+1,2i} + b^n_{2j+1,2i+2} \\ &+ b^n_{2j,2i+2} + b^n_{2j+2,2i}). \end{aligned}$$

Dementsprechend können wir zusätzlich eine vereinfachte Variante der Restriktion implementieren, in der wir statt der Berechnung des Residuums lediglich das feinere B-Array verwenden. Die Berechnung der Restriktion bleibt gleich.

```
1  void cpu_restriction_simplified(
2      pgrid_t fgrid, pgrid_t cgrid) {
3    int i, j, di, dj, f_d_lnw, c_w, c_h, c_b_lnw;
4    float *f_d, *c_b;
5
6    /* Initialisierung */
7    f_d     = fgrid->mem + fgrid->b_off;
8    f_d_lnw = fgrid->b_lnw;
9    c_w     = cgrid->w;
10   c_h     = cgrid->h;
11   c_b     = cgrid->mem + cgrid->b_off;
12   c_b_lnw = cgrid->b_lnw;
13
14   /* Restriktion */
15   /* ... */
16 }
```

3.5. Prolongation

Um die Prolongation nach den Formeln 2.5 bis 2.8 zu realisieren, sei je Restriktionsschritt das feinere Gitter `fgrid` und das gröbere Gitter `cgrid` gegeben. Das grobe X-Quellarray enthält eine Approximation des Fehlers des feinen Gitters. Wir müssen daher den approximierten Fehler mit Hilfe der Prolongation auf das feine X-Quellarray übertragen. Um ein einzelnes Element des X-Quellarrays zu berechnen, benutzen wir abhängig davon, ob sein vertikaler und horizontaler Index gerade oder ungerade ist, eine der vier Formeln. Eine Möglichkeit, dies zu realisieren, ist es, über alle Elemente des feinen X-Quellarrays zu iterieren und per Fallunterscheidung die richtige Formel auszuwählen. Da sich Fallunterscheidungen allerdings negativ auf das Pipelining der CPU auswirken, sollten wir daher auf Fallunterscheidungen im Schleifeninneren verzichten. Eine andere Möglichkeit ist es, über das

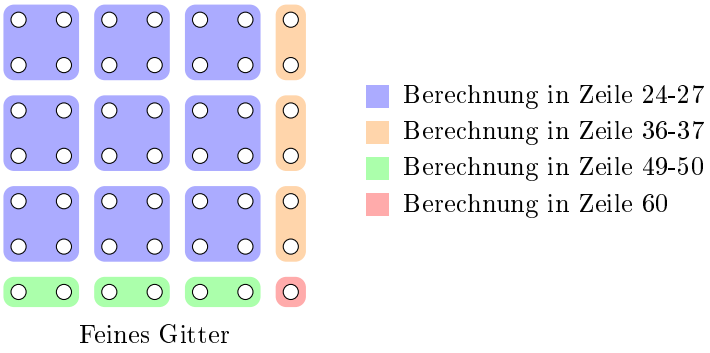


Abbildung 3.2.: Veranschaulichung der Prolongation inklusive Randbehandlung.

grobe X-Quellarray zu iterieren und je Schleifeniteration die vier korrespondierenden Elemente des feinen Arrays zu berechnen. Zusätzlich brauchen wir noch eine Behandlung der Randfälle. Abbildung 3.2 zeigt diese Aufteilung. Eine Implementierung dazu sieht wie folgt aus.

```
1  void cpu_prolongation(pgrid_t fgrid, pgrid_t cgrid) {
2    int i, j, di, dj, f_x_lnw, c_w, c_h, c_x_lnw;
3    float *f_x, *c_x, help1, help2, help3, help4;
4
5    /* Initialisierung */
6    f_x     = fgrid->mem + fgrid->x_off0 + fgrid->x_lnw + 1;
7    f_x_lnw = fgrid->x_lnw;
8    c_w     = cgrid->w;
9    c_h     = cgrid->h;
10   c_x     = cgrid->mem + cgrid->x_off0 + cgrid->x_lnw + 1;
11   c_x_lnw = cgrid->x_lnw;
12
13   /* Prolongation */
14   for(j = 0; j < c_h; j++) {
15     /* Hauptbereich */
16     dj = 2 * j;
17     for(i = 0; i < c_w; i++) {
18       di = 2 * i;
19
20       help1 = c_x[i + c_x_lnw * j];
21       help2 = help1 / 2;
22       help3 = c_x[(i-1) + c_x_lnw * j] / 2;
23       help4 = c_x[i + c_x_lnw * (j-1)] / 2;
24       f_x[di     + f_x_lnw * dj    ] += -help4 - help3;
25       f_x[(di+1) + f_x_lnw * dj    ] += -help4 - help2;
26       f_x[di     + f_x_lnw * (dj+1)] += -help3 - help2;
27       f_x[(di+1) + f_x_lnw * (dj+1)] += -help1;
28     }
29
30     /* Rand rechts */
```

```
    i = c_w;
    di = 2 * i;
    help2 = c_x[i + c_x_lnw * j] / 2;
    help3 = c_x[(i-1) + c_x_lnw * j] / 2;
    help4 = c_x[i + c_x_lnw * (j-1)] / 2;
    f_x[di     + f_x_lnw * dj    ] += -help4 - help3;
    f_x[di     + f_x_lnw * (dj+1)] += -help3 - help2;
  }

  /* Rand unten */
  j = c_h;
  dj = 2 * j;
  for(i = 0; i < c_w; i++) {
    di = 2 * i;

    help2 = c_x[i + c_x_lnw * j] / 2;
    help3 = c_x[(i-1) + c_x_lnw * j] / 2;
    help4 = c_x[i + c_x_lnw * (j-1)] / 2;
    f_x[di     + f_x_lnw * dj] += -help4 - help3;
    f_x[(di+1) + f_x_lnw * dj] += -help4 - help2;
  }

  /* Randpunkt unten rechts */
  i = c_w;
  di = 2 * i;
  j = c_h;
  dj = 2 * j;
  help3 = c_x[(i-1) + c_x_lnw * j] / 2;
  help4 = c_x[i + c_x_lnw * (j-1)] / 2;
  f_x[di     + f_x_lnw * dj    ] += -help4 - help3;
}
```

Erfolgt die Prolongation auf eine Gitterstufe mit $n > 0$, so müsste die aktuelle Näherung des Gitters vor Anwendungen der Prolongation mit Nullen initialisiert werden. Diese Initialisierung lässt implizit vornehmen, indem wir statt einer Modifikation der Näherung mit `+=` eine Zuweisung mit `=` verwenden. Dazu implementieren wir wie bei der Restriktion die Prolongation ein weiteres Mal für gröbere Gitter.

```
void cpu_prolongation_simplified(
    pgrid_t fgrid, pgrid_t cgrid) {
  int i, j, di, dj, f_x_lnw, c_w, c_h, c_x_lnw;
  float *f_x, *c_x, help1, help2, help3, help4;

  /* Initialisierung */
  /* ... */

  /* Prolongation */
  for(j = 0; j < c_h; j++) {
    /* Hauptbereich */
```

```
12      dj = 2 * j;
13      for(i = 0; i < c_w; i++) {
14        di = 2 * i;
15
16        help1 = c_x[i + c_x_lnw * j];
17        help2 = help1 / 2;
18        help3 = c_x[(i-1) + c_x_lnw * j] / 2;
19        help4 = c_x[i + c_x_lnw * (j-1)] / 2;
20        f_x[di     + f_x_lnw * dj    ] = -help4 - help3;
21        f_x[(di+1) + f_x_lnw * dj    ] = -help4 - help2;
22        f_x[di     + f_x_lnw * (dj+1)] = -help3 - help2;
23        f_x[(di+1) + f_x_lnw * (dj+1)] = -help1;
24      }
25
26      /* Rand rechts */
27      /* ... */
28    }
29
30    /* Rand unten */
31    /* ... */
32
33    /* Randpunkt unten rechts */
34    /* ... */
35  }
```

3.6. Mehrgitterverfahren

Um schließlich das Mehrgitterverfahren insgesamt zu implementieren, verwenden wir eine rekursive Funktion. Diese Funktion erhält ein feines Gitter `fgrid` und prüft zunächst, ob die gewünschte Grobheit bereits erreicht ist. Falls dies nicht der Fall ist, dann wird das nächst gröbere Gitter gebildet, darauf das Mehrgitterverfahren angewandt und das feine Gitter geglättet. Ist die gewünschte Grobheit erreicht, so wird das Gleichungssystem des gröbsten Gitters gelöst. Die Implementierung sieht folgendermaßen aus.

```
1   void cpu_multigrid_help(
2       pgrid_t fgrid, int smoothing_cnt, int is_finest) {
3     int f_w, f_h, c_w, c_h;
4     pgrid_t cgrid;
5
6     /* Initialisierung */
7     f_w = fgrid->w;
8     f_h = fgrid->h;
9     c_w = f_w / 2;
10    c_h = f_h / 2;
11
12    /* Mehrgitterschritt */
13    if(c_w >= 1 && c_h >= 1) {
```

```
14      /* Restriktion */
15      cgrid = (pgrid_t) malloc((size_t) sizeof(grid_t));
16      grid_init_sub(fgrid, cgrid, c_w, c_h);
17
18      if(is_finest)
19        cpu_restriction(fgrid, cgrid);
20      else
21        cpu_restriction_simplified(fgrid, cgrid);
22
23      /* Grobes Gitter lösen */
24      cpu_multigrid_help(cgrid, smoothing_cnt, FALSE);
25
26      /* Prolongation */
27      if(is_finest)
28        cpu_prolongation(fgrid, cgrid);
29      else
30        cpu_prolongation_simplified(fgrid, cgrid);
31      free(cgrid);
32
33      /* Glätter */
34      cpu_smoothing(fgrid, smoothing_cnt, 0);
35    }
36    /* Exaktes Lösen des gröbsten Gitters */
37    else {
38      /* Gröbstes X-Quellarray mit 0en initialisieren */
39      cpu_clear(fgrid->mem + fgrid->x_off0, fgrid->x_lnw,
40          (fgrid->w+2), (fgrid->h+2));
41      /* Glätter */
42      cpu_smoothing(fgrid, 1, FALSE);
43    }
44  }
45
46  void cpu_multigrid(pgrid_t grid, int smoothing_cnt) {
47    cpu_multigrid_cpu_help(grid, smoothing_cnt, TRUE);
48  }
```

4. Grafikkartenprogrammierung mit OpenCL

4.1. Datendominanz

Herkömmliche Hauptprozessoren zielen darauf ab, kontrolldominante Probleme zu lösen. Sie wurden mit dem Ziel entworfen, möglichst effizient mit Fallunterscheidung umgehen zu können. Im Gegensatz dazu fordern datendominante Probleme lediglich die Durchführung möglichst vieler arithmetischer Operationen pro Zeiteinheit. Das von uns betrachtete Fallbeispiel ist ein solches Problem. Betrachten wir beispielsweise Formel 2.2, so stellen wir fest, dass sich jedes $y'_{j,i}$ unabhängig von $y'_{l,k}$ mit $l \neq j$ oder $k \neq i$ berechnen lässt. Daher ist es naheliegend, von einer Hardware, die einen Glättungsschritt möglichst schnell durchführen können soll, zu fordern, dass diese viele gleichartige Berechnungen parallel ausführen kann. Dies bedeutet konkret, dass die Hardware zum einen möglichst viele Rechenwerke und zum anderen eine ausreichend große Speicheranbindung besitzen muss, um die benötigten Daten für die vielen parallelen Berechnungen schnell zu beschaffen und die Ergebnisse zurückzuschreiben.

Für datendominante Probleme sollte daher der Einsatz geeigneter Hardware in Betracht gezogen werden. *Single instruction, multiple data* (SIMD) ist eine solche Klasse von Hardwarearchitekturen, die den genannten Ansatz verfolgt. Darunter wird das parallele Ausführen des gleichen Befehls mit unterschiedlichen Daten verstanden. Zwar bieten viele aktuelle Prozessoren diesbezüglich zusätzliche Funktionen, wie zum Beispiel Multithreading oder die *Streaming SIMD Extensions* (SSE) für die x86-Architektur, allerdings erreichen diese Ergänzungen nicht die Leistungsfähigkeit spezialisierter Hardware. Eine Alternative, die sich auf Grund ihrer hohen Verbreitung und den erschwinglichen Preisen anbietet, stellen Grafikprozessoren dar. Die Berechnung dreidimensionaler Bilder ist ein klassisches Beispiel eines datendominanten Problems. In den letzten Jahren kamen die Hersteller solcher Hardware zu der Einsicht, dass Grafikprozessoren zu weit mehr in der Lage sind als zur Berechnung von Bildern. Es sind Programmierschnittstellen entstanden, mit denen sich eigene Berechnungen auf den Grafikprozessoren durchführen und die Ergebnisse in den Hauptspeicher kopieren lassen. Als solche Programmierschnittstellen sind die proprietäre *Compute Unified Device Architecture* (CUDA) und die *Open Computing Language* (OpenCL) zu nennen. Beide haben ihre Vor- und Nachteile. Im Folgenden werden wir uns ausschließlich mit OpenCL beschäftigen.

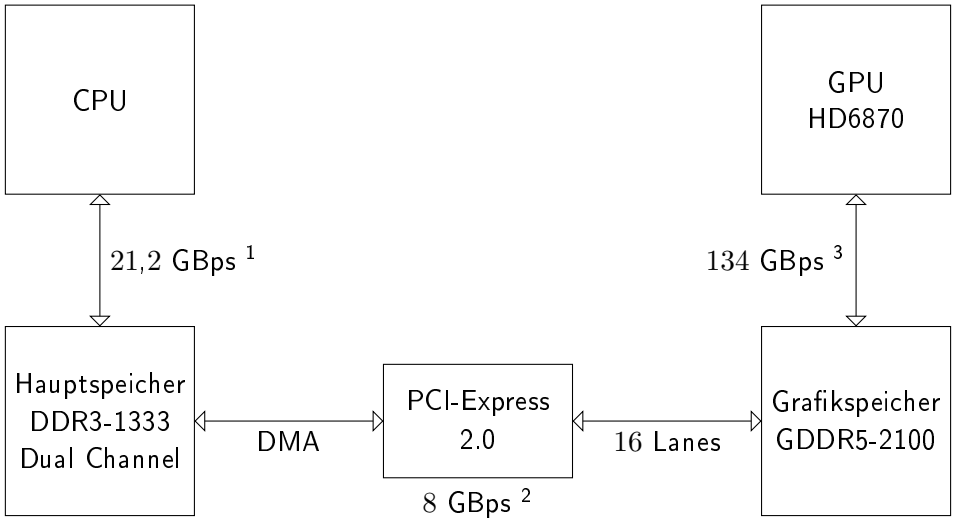
4.2. Grafikkarten

Zunächst ist zwischen integrierten Grafikprozessoren und dedizierten Grafikkarten zu unterscheiden. Integrierte Grafikprozessoren sind entweder im Chipsatz integriert oder sind neuerdings oft Koprozessoren des Hauptprozessors. Sie sind meist relativ kostengünstig und bieten eher wenig Rechenleistung. Dedizierte Grafikkarten sind dagegen in vielen Fällen deutlich leistungsfähiger als die integrierten Varianten. Daher beschränken wir uns im Folgenden auf dedizierte Grafikkarten. Allerdings sei zu erwähnen, dass zurzeit integrierte Grafikprozessoren deutlich aufholen und theoretisch den Vorteil eines mit dem Hauptprozessor geteilten Arbeitsspeichers besitzen.

Dedizierte Grafikkarten sind separate Steckkarten, die über PCI Express (ältere Karten über AGP oder PCI) mit dem Hostrechner verbunden werden. Sie besitzen einen eigenen Arbeitsspeicher, der meist zwar kleiner, aber deutlich schneller ist als der üblicherweise verbaute Hauptspeicher. Zusätzlich besitzen sie mindestens einen Grafikprozessor, auch *Graphics Processing Unit* (GPU) genannt. Zur Unterscheidung von klassischen Grafikprozessoren, die lediglich zur Grafikberechnung in der Lage waren, wird auch der Begriff *General Purpose Graphics Processing Unit* (GPGPU) verwendet.

Dem Grafikprozessor stehen verschiedene Speicherarten zur Verfügung. Das sind zum einem diverse kleinere Speicher, um Zugriffe auf häufig genutzte Daten zu beschleunigen. Zum anderen steht eine deutlich größere Menge globaler Grafikspeicher zur Verfügung. In diesem globalen Speicher befindet sich meist auch der Hauptteil der Daten, auf denen gerechnet wird. Zum Zugriff auf den Hauptspeicher des Hostsystems ist der Grafikprozessor allerdings nicht fähig. Stattdessen müssen die Daten vom Hauptspeicher zum globalen Grafikspeicher kopiert werden. Dazu werden die Daten zunächst vom Hauptspeicher an den PCI-Express-Bus übergeben. Im Idealfall geschieht dies per *Direct Memory Access* (DMA) um den Hauptprozessor zu entlasten. Von dort aus werden die Daten von der Grafikkarte übernommen und in den Grafikspeicher geschrieben. Es lassen sich ebenso in umgekehrter Reihenfolge Daten vom Grafikspeicher in den Hauptspeicher kopieren. Wichtig ist hierbei die Tatsache, dass die Datenübertragung über den PCI-Express-Bus für gewöhnlich nur mit einer relativ geringen Datentransferrate möglich ist. Die schnelle Anbindung des Grafikprozessors an den Grafikspeicher ist somit nur dann von Vorteil, sofern lange genug auf den in den Grafikspeicher kopierten Daten gerechnet wird. Dieser Umstand stellt eine starke Einschränkung dar, die die Vorteile des Grafikprozessors je nach Anwendungsfall überwiegen kann. Eine Übersicht der Transferraten bietet Abbildung 4.1.

Der Grafikprozessor selbst ist schließlich für die Berechnungen zuständig. Das Konzept eines Grafikprozessors ähnelt dem der Vektorrechner. Der konkrete Aufbau variiert allerdings stark und hängt vom Modell und Hersteller ab. Zudem verwenden verschiedene Hersteller unterschiedliche Begriffe. Daher verwenden wir im Folgen-



[1] http://www.elektronik-kompendium.de/sites/com/1312291.htm
[2] http://www.hardware-infos.com/tests/grafikkarten/pci-express-3.0.html
[3] http://www.techpowerup.com/gpudb/256/radeon-hd-6870.html

Abbildung 4.1.: Theoretische maximale Transferraten auf einem Beispielsystem.

den die bei der Firma AMD gebräuchlichen Begriffe und werden nur grob auf die Struktur eingehen, damit der Bezug zu OpenCL verständlich wird. Die Informationen sind dem *AMD Accelerated Parallel Processing OpenCL Programming Guide* [2] entnommen. Ein Grafikprozessor besteht im wesentlichen aus mehreren Compute Units, die jeweils unabhängig voneinander Berechnungen ausführen können. Alle Compute Units teilen sich den globalen Grafikspeicher und einen Konstantenspeicher. Darüber hinaus besitzt jede Compute Unit einen lokalen Speicher und private Register. Zur Berechnung hat jede Compute Unit zahlreiche Processing Elements, die je nach Grafikkartenmodell verschieden viele Rechenwerke besitzen und unterschiedlich organisiert sein können. In Abbildung 4.2 ist eine Übersicht dieser Hierarchie zu sehen. Eine Compute Unit ist in der Lage, eine oder meherere Wavefronts gleichzeitig auszuführen. Eine Wavefront setzt sich zusammen aus einem Befehlszeiger, einem Teil des lokalen Speichers und der Anzahl seiner Work-Items, denen wiederum ein Teil der privaten Register zugewiesen wird. Ein Work-Item ist schließlich das, was auf einem einzelnen Processing Element ausgeführt wird. Durch das Ausführen des selben Befehls auf mehreren Processing Elements und eventuell über mehrere Takte können schließlich alle Work-Items einer Wavefront im Gleichschritt rechnen. Dadurch, das jeder Work-Item eigene private Register besitzt, können die Work-Items unterschiedliche Berechnungen durchführen. Die maximale Anzahl an Work-Items je Wavefront ist für einen Grafikprozessor fest vorgegeben und sollte für eine optimale Auslastung auch genutzt werden. Eine Wavefront ist fest einer Compute Unit zugewiesen, wird also ausschließlich auf dieser ausgeführt. Während eine

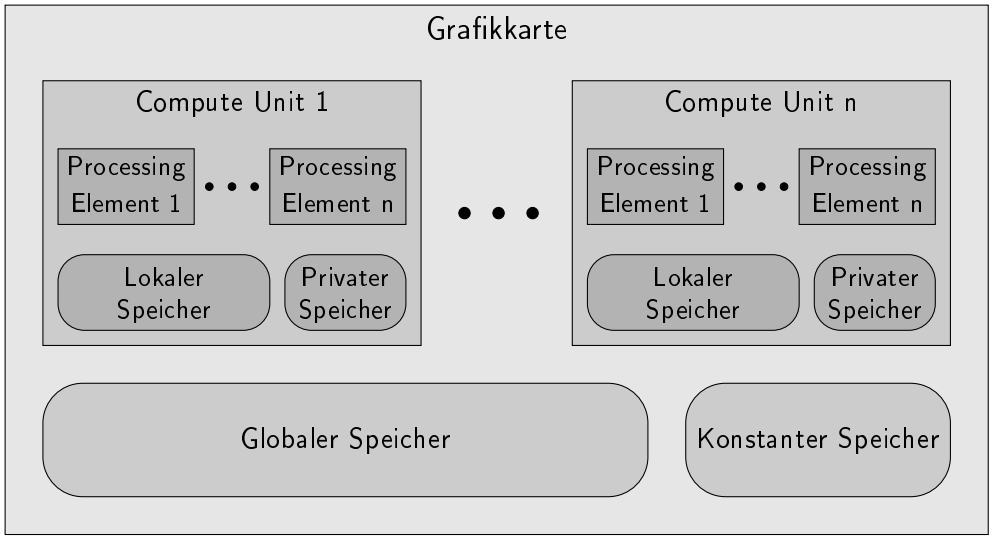


Abbildung 4.2.: Beispielhafte Hierarchie einer Grafikkarte.

Compute Unit nur wenige Wavefronts zeitgleich ausführen kann, werden ihr dennoch zahlreiche Wavefronts zugewiesen. Dadurch ist sie in der Lage, Wartezeiten, zum Beispiel bei Zugriffen auf den Speicher, zu verstecken, indem andere Wavefronts währenddessen ausgeführt werden. Daher ist es sinnvoll, eine Berechnung auf möglichst viele Wavefronts aufzuteilen um eine hohe Auslastung des Grafikprozessors zu erreichen.

4.3. OpenCL

Die *Open Computing Language* (OpenCL) existiert unabhängig von Grafikkarten. Sie wurde allgemein für Parallelrechner entworfen und Implementierungen sind für unterschiedliche Hardware denkbar. So unterstützt die *Accelerated Parallel Processing* (APP) SDK von AMD sowohl CPUs als auch GPUs ihrer eigenen Marke. Der OpenCL-Standard besteht aus einer Programmierschnittstelle und aus der Programmiersprache OpenCL C. Desweiteren spezifiziert OpenCL eine Sicht auf die Hardware, die Ähnlichkeiten mit der Struktur einer Grafikkarte besitzt.

Nach OpenCL besteht das System aus einem Host und mindestens einem Device. Der Host entspricht dabei dem Hauptprozessor und ein Device könnte beispielsweise eine Grafikkarte sein. Sowohl Host als auch Device besitzen eigenen Speicher. Der Host übernimmt die Aufgabe der Verwaltung der Devices. Dazu kann er Host- und Devicespeicherblöcke von OpenCL anfordern, die als Buffer bezeichnet werden. Es lassen sich Kopiervorgänge von Buffer zu Buffer initiieren oder direkt Daten zwischen vom Betriebssystem allozierten Hostspeicher und einem Buffer übertragen. Außerdem lassen sich Berechnungen auf den Devices starten. Dies wird als Aufruf

eines Kernels bezeichnet. Einem auf einem Device auszuführenden Kernel lassen sich dabei Parameter übergeben, beispielsweise Buffer des gleichen Devices.

Der Aufbau eines Devices ähnelt dem einer GPU. Das Device besitzt mehrere unabhängige Compute Units. Jeder Compute Unit wird zur Berechnung eine Menge von Work-Groups fest zugewiesen. Jede Work-Group besitzt wiederum eine beliebige Menge von Work-Items. Diese Aufteilung scheint auf den ersten Blick analog zur oben bereits erwähnten Hierarchie der Wavefronts und Work-Items definiert zu sein. Da die Anzahl der Work-Items einer Wavefront allerdings beschränkt ist, muss eine Work-Group aus einer oder mehreren Wavefronts bestehen. Alle Work-Items einer Work-Group führen das gleiche Programm aus. Die Ausführung aller Work-Items findet aber abhängig von der Hardware nicht grundsätzlich parallel statt.

Jedes Device besitzt seinen eigenen Speicher. Einem Device steht zur Bearbeitung eines Kernelaufrufs im Normalfall nur der eigene Speicher zur Verfügung. Dabei gibt es verschiedene Speicherarten für unterschiedliche Einsatzzwecke.

- Ein globaler Devicespeicher steht allen Compute Units zur Verfügung. Er ist groß aber langsamer als die andere Devicespeicherarten.

- Ein konstanter Devicespeicher steht ebenfalls allen Compute Units zur Verfügung. Er kann von einem Kernel zwar gelesen, aber nicht beschrieben werden. Für diese Zwecke kann ein spezieller, schnellerer Speicher vorgesehen werden.

- Jede Work-Group besitzt eigenen lokalen Devicespeicher. Zugriffe innerhalb einer Work-Group auf diesen Speicher lassen sich synchronisieren. Somit lässt er sich zur Kommunikation innerhalb einer Work-Group nutzen. Er ist eher klein aber schnell. Somit eignet er sich darüber hinaus als Cache für häufig genutzte Daten.

- Jedes Work-Item besitzt eigenen privaten Devicespeicher. Im Wesentlichen werden dadurch private Variablen innerhalb einer Funktionsdefinition realisiert. Durch den Einsatz einer privaten Variable als Index für lokalen, globalen oder konstanten Devicespeicher können Work-Items auf unterschiedlichen Daten rechnen.

Devicespeicher kann man sich von der OpenCL-API allozieren lassen. Daneben gibt es noch den Hostspeicher. Dieser lässt sich einerseits ganz normal vom Betriebssystem allozieren. Allerdings befindet sich der allozierte Speicher dann im virtuellen Adressraum des Prozesses und ist somit nicht zwangsläufig zusammenhängend im physikalischen Adressraum oder befindet sich sogar im Auslagerungsspeicher. Solcher Speicher ist für Kopiervorgänge zu einem Buffer ungeeignet, da er sich nicht per DMA auf den PCI-Express-Bus transferieren lässt. Für solche Zwecke benötigt man speziellen pinned Speicher, der zusammenhängend im physikalischen Adressraum alloziert wird. Einen solchen Speicherbereich im Hostspeicher kann man sich

von der OpenCL-API anlegen lassen. Falls dennoch Daten zwischen normalen Hostspeicher und einem Buffer kopiert werden sollen, können spezielle Funktionen der OpenCL-API genutzt werden, die die Daten zunächst in eine pinned Speicher zwischenspeichern.

Als Referenz für die Verwendung von OpenCL dient in erster Linie das *OpenCL Programming Guide* [3] und die *OpenCL 1.2 Reference Pages* von *The Khronos Group* [4]. Bevor ein Kernelaufruf auf einem Device gestartet werden kann, sind einige Initialisierungsschritte notwendig. Als erstes muss eine auf dem System installierte OpenCL-Implementierung, von OpenCL als Platform bezeichnet, gewählt werden.

```
1 cl_platform_id platform;
2 result = clGetPlatformIDs(1, &platform, NULL);
3 assert(result == CL_SUCCESS);
```

Als Nächstes werden eine oder mehrere Devices dieser Platform gewählt, auf denen Kernelaufrufe stattfinden sollen. Danach wird ein Context erstellt, dem diese Devices bekannt gemacht werden.

```
1 cl_device_id device;
2 result = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
3     &device, NULL);
4 assert(result == CL_SUCCESS);
5
6 cl_context context = clCreateContext(NULL, 1, &device,
7     NULL, NULL, &result);
8 assert(result == CL_SUCCESS);
```

Um einen Kernelaufruf tätigen zu können, müssen wir zunächst dessen Quellcode für die Devices kompilieren, auf denen der Kernel aufgerufen werden soll. Der Quellcode umfasst ein OpenCL-C-Programm und liegt als Array von Zeichenketten vor. Das Kompilieren findet zur Laufzeit statt. Anschließend erstellen wir den Kernel und geben dabei den Kernelnamen an, den der Kernel im Quellcode besitzt.

```
1  cl_program program = clCreateProgramWithSource(context,
2      1, kernel_src_line_cnt, kernel_src, NULL, &result);
3  assert(result == CL_SUCCESS);
4
5  result = clBuildProgram(program, 1, &device,
6      NULL, NULL, NULL);
7  assert(result == CL_SUCCESS);
8
9  cl_kernel kernel = clCreateKernel(program,
10     "kernel_name", &result);
11 assert(result == CL_SUCCESS);
```

Um Aufgaben auf Devices auszuführen, müssen wir für jedes Device mindestens eine Queue erstellen. Eine Queue ist genau einem Device zugeordnet. Ihr lassen sich verschiedene Aufgaben zuweisen, beispielsweise das Kopieren von Daten oder das

Ausführen eines Kernels. Diese Aufgaben werden in der Reihenfolge der Zuweisung und nacheinander ausgeführt. Die Ausführung der Aufgaben einer Queue findet im Normalfall nebenläufig zur Ausführung des Hostprozesses statt. Außerdem brauchen wir je Device mindestens einen Buffer, um dem Kernel größere Mengen an Daten zur Berechnung zu übergeben und Ergebnisse zurückzuerhalten.

```
cl_command_queue queue = clCreateCommandQueue(context,
    device, 0, &result);
assert(result == CL_SUCCESS);

cl_mem buffer = clCreateBuffer(context, CL_MEM_READ_WRITE,
    data_size, NULL, &result);
assert(result == CL_SUCCESS);
```

Wir können nun mit Hilfe der Queue dem Device Aufgaben zuweisen. Dazu veranlassen wir zunächst das Kopieren von Daten aus dem Hostspeicher in den Buffer. Dann setzen wir die Parameter, die der Kernel benötigt und fügen den Kernelaufruf in die Queue ein. Beim Kernelaufruf müssen wir festlegen, wie viele Work-Items insgesamt und je Work-Group genutzt werden sollen. Dabei muss der erste Wert ein Vielfaches des zweiten Werts sein. Zuletzt sorgen wir noch dafür, dass die Ergebnisse aus dem Buffer zurück in den Hostspeicher kopiert werden. Da die Aufgaben nebenläufig abgearbeitet werden, kann der Hostprozess mit `clFinish` auf das Abarbeiten aller Aufgaben einer Queue warten.

```
result = clEnqueueWriteBuffer(queue, buffer, CL_FALSE,
    0, data_size, data, 0, NULL, NULL);
assert(result == CL_SUCCESS);

clSetKernelArg(kernel, sizeof(cl_mem), &buffer);
result = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
    &global_work_size, &local_work_size, 0, NULL, NULL);
assert(result == CL_SUCCESS);

result = clEnqueueReadBuffer(queue, buffer, CL_FALSE,
    0, data_size, data, 0, NULL, NULL);
assert(result == CL_SUCCESS);

result = clFinish(queue);
assert(result == CL_SUCCESS);
```

4.4. OpenCL C

Programme, die auf den Devices ausgeführt werden sollen, müssen in der Sprache OpenCL C formuliert werden. OpenCL C ist eine Sprache, die stark an der Programmiersprache C anlehnt. Es wird die grundlegende Syntax und Semantik übernommen und um neue Elemente ergänzt. Allerdings fehlen auch einige wesentliche Elemente. Beispielsweise verzichtet OpenCL C auf einen Precompiler, wodurch das

Inkludieren externen Quellcodes nicht möglich ist. Die Konsequenz daraus ist, dass alle verwendeten Funktionen entweder selbst implementiert oder von der OpenCL-Implementierung zur Verfügung gestellt werden müssen. Da allerdings OpenCL-C-Programme meist relativ knapp formuliert werden und Funktionen für die gebräuchlichen mathematischen Operationen bereits implementiert sind, fällt dieser Nachteil nicht besonders ins Gewicht. Im Folgenden ist ein Beispiel eines simplen Kernels zu sehen.

```
1  __kernel void kernel_clear(__global float *mem,
2      int off, int lnw, int w, int h) {
3    int i, j, wg_j_fir, wg_j_las;
4
5    mem += off;
6    wg_j_fir = h * get_global_id(0)
7        / get_global_size(0);
8    wg_j_las = h * (get_global_id(0) + 1)
9        / get_global_size(0);
10
11   for(i = get_local_id(0);
12       i < w;
13       i += get_local_size(0)) {
14     for(j = wg_j_fir; k < wg_j_las; j++) {
15       mem[i + lnw * j] = 0;
16     }
17   }
18 }
```

Der erste erkennbare Unterschied zu C sind die Schlüsselworte `__kernel` und `__global`. Dabei gibt `__kernel` an, dass diese Funktion eine Kernelfunktion ist und vom Host aufgerufen werden kann. Das Schlüsselwort `__global` gibt an, dass der entsprechende Zeiger auf globalen Devicespeicher zeigt. Die Funktionen `get_global_id`, `get_global_size`, `get_local_id` und `get_local_size` geben die ID und Anzahl der Work-Groups bzw. Work-Items zurück. Da dieser Code letztendlich von tausenden Work-Items ausgeführt wird, lässt sich mit diesen Funktionen die Gesamtarbeit auf die einzelnen Work-Groups bzw. Work-Items aufteilen, indem individuelle Indizes zum Zugriff auf den Speicher berechnet werden.

OpenCL-C-Programme müssen zur Laufzeit für die Devices kompiliert werden, auf denen sie schließlich ausgeführt werden sollen. Dies bietet den Vorteil, dass erst zur Laufzeit die Zielarchitektur bekannt sein muss. Das Programm wird der OpenCL-API als Array von Zeichenketten übergeben. Ein OpenCL-C-Programm in C lässt sich beispielsweise folgendermaßen formulieren.

```
1  static const char *src[] = {
2      "__kernel void kernel_clear(__global float *mem, \n",
3      "    int off, int lnw, int w, int h){            \n",
4      "  int i, j, wg_j_fir, wg_j_las;                 \n",
5      "                                                \n",
```

```
6       "  mem += off;                                          \n",
7       "  wg_j_fir = h * get_global_id(0)                      \n",
8       "      / get_global_size(0);                            \n",
9       "  wg_j_las = h * (get_global_id(0) + 1)                \n",
10      "      / get_global_size(0);                            \n",
11      "                                                       \n",
12      "  for(i = get_local_id(0);                             \n",
13      "      i < w;                                           \n",
14      "      i += get_local_size(0))                          \n",
15      "    for(j = wg_j_fir; k < wg_j_las; j++) {             \n",
16      "      mem[i + lnw * j] = 0;                            \n",
17      "    }                                                  \n",
18      "  }                                                    \n",
19      "}                                                      \n",
20  };
```

4.5. Optimierungen

Während bei der Formulierung von OpenCL-C-Programmen zwar die eingesetzte Hardware nicht weiter beachtet werden muss, so sollte sie dennoch stets im Hinterkopf behalten werden. Eine optimale Auslastung der Hardware wird am ehesten erreicht, wenn die Abläufe zur Berechnung klar sind und bei der Programmierung berücksichtigt werden.

Dem Grafikprozessor liegt das Prinzip eines Vektorrechners zu Grunde. Viele Rechenelemente führen dabei den gleichen Befehl aus. Dies führt im Allgemeinen zu Problemen bei Fallunterscheidungen, bei denen einige Rechenelemente andere Programmzweige ausführen als andere. Dennoch ist die GPU in der Lage, mit Fallunterscheidung umzugehen. Die Idee dabei ist, dass eine Wavefront bei einer Programmverzweigung alle Zweige ausführt, die von mindestens einem Work-Item ausgeführt werden müssen. Alle Work-Items, die diesen Zweig nicht ausführen müssen, sind dabei inaktiv. Mit der Granularität einer Verzweigung wird die Anzahl der Work-Items bezeichnet, die diesen Zweig durchlaufen müssen. Es sollte bei Verzweigungen auf eine möglichst hohe Granularität geachtet werden, damit viele Work-Items aktiv rechnen können. Ist dies nicht möglich, so sollten Verzweigungen mit niedriger Granularität sehr kurz sein.

Speicherzugriffe sind teure Operationen, die mehrere Takte in Anspruch nehmen können. Daher ist ein wesentlicher Punkt bei der Optimierung von Kerneln die Nutzung des Speichers. Dazu gehört zum einen die Verwendung der richtigen Speicherart für einen Einsatzzweck. Jede Speicherart hat dabei ihre Vor- und Nachteile. Zum anderen sollte im Groben bekannt sein, wie die entsprechende Speicherart an die Hardware angebunden ist und verwendet werden soll. Im Wesentlichen nutzen wir im Folgenden den globalen Devicespeicher. Die benötigten Daten kann der Host direkt in diesen Speicher kopieren, so dass alle Compute Units des Devices Zugriff

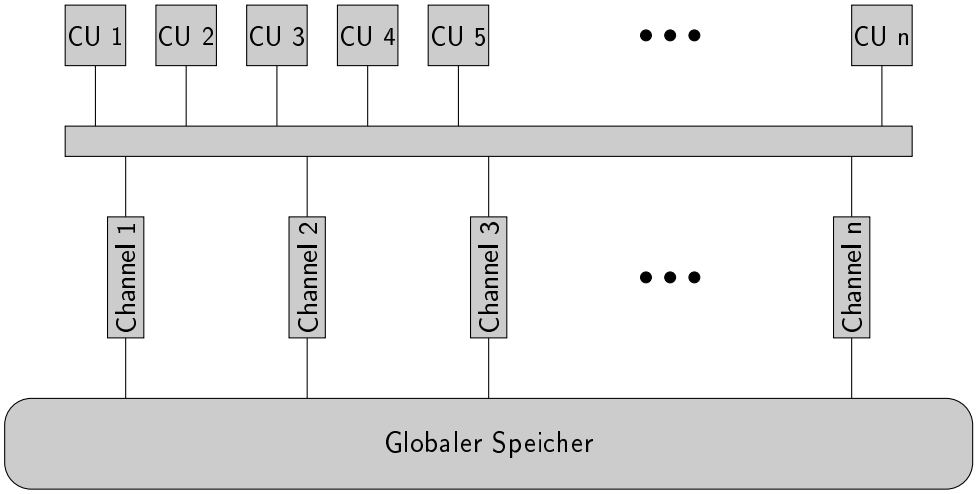


Abbildung 4.3.: Speicherchannels des globalen Grafikspeichers.

darauf haben. Auf der Grafikkarte des Testsystems finden Zugriffe auf den globalen Devicespeicher über acht sogenannte Speicherchannels statt, siehe dazu Abbildung 4.3. Welcher Speicherchannel bei einem Speicherzugriff zum Einsatz kommt, bestimmen Bit 10:8 der Speicheradresse. Ein Inkrementieren der Adresse um 256 führt somit zum Wechsel des Channels. Finden mehrere Zugriffe auf einen Speicherchannel innerhalb kurzer Zeit statt, so werden diese Zugriffe serialisiert. Für eine einzelne Compute Unit mag es daher sinnvoll erscheinen, dass alle seine Work-Items stets auf unterschiedliche Speicherchannels zugreifen, um alle Speicherchannels gleichmäßig auszulasten. Da aber auf einem Grafikprozessor zahlreiche Compute Units parallel arbeiten und diese sich die Speicherchannels teilen müssen, sollte eine einzelne Compute Unit möglichst auf wenige Speicherchannels in kurzer Zeit zugreifen. Dadurch können eventuell Optimierungen stattfinden, zum Beispiel das Zusammenfassen mehrere Anfragen. Auf der anderen Seite sollten möglichst wenig Compute Units auf den gleichen Speicherchannels arbeiten, um eine gleichmäßige Auslastung aller Channels zu gewährleisten. Eine Möglichkeit, eine Compute Unit auf nur wenigen Speicherchannels arbeiten zu lassen, ist die Strategie, dass Work-Items mit aufeinanderfolgender ID auf aufeinanderfolgende Speicheradressen zugreifen. Dazu sei nochmals das folgende Beispiel gegeben.

```
1  for(i = get_local_id(0);
2      i < w;
3      i += get_local_size(0)) {
4    for(j = wg_j_fir; k < wg_j_las; j++) {
5      mem[i + lnw * j] = 0;
6    }
7  }
```

Durch die Verwendung der ID als Summand zur Indexberechnung greifen alle Work-Items auf aufeinanderfolgende Speicheradressen zu. Ein weiterer wichtiger Punkt ist

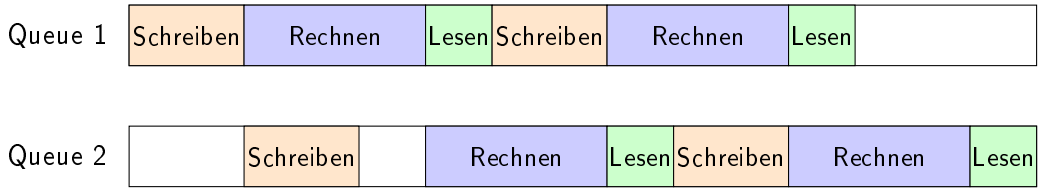


Abbildung 4.4.: Beispiel eines Programmausführungsverlaufs unter Verwendung mehrerer Queues.

die Aufteilung der Gesamtaufgabe auf die einzelnen Wavefronts. Da der Abstand zwischen zwei Channels im Adressraum 256 beträgt und es acht Speicherchannels gibt, landet man nach einem Inkrementieren einer Adresse um 2048 wieder im gleichen Channel. Bei der Aufteilung des Speichers auf die Work-Groups sollte auf die Verwendung von Vielfachen von 256 verzichtet werden, um ungünstige Zugriffsmuster auf den Speicher zu verhindern. Bei zweidimensionalen Arrays wäre dies zum Beispiel durch den Verzicht auf eine Zeilenbreite einer Zweierpotenz möglich.

Eine sehr allgemeine aber auch sehr sinnvolle Strategie ist die Verwendung mehrerer Queues je Device. Dabei soll ausgenutzt werden, dass sich einige Aufgaben parallel bearbeiten lassen. Ein triviales Beispiel ist das parallele Kopieren von Daten über PCI-Express und Rechnen auf dem Device. Dazu wird die zu berechnende Gesamtaufgabe in kleinere Teilaufgaben, den sogenannten Slices, aufgeteilt. Sobald die erste Queue mit der Berechnung der ersten Slice beginnt, kann die zweite Queue bereits die benötigten Daten zur Berechnung seiner Slice kopieren. Wenn die erste Queue lange genug rechnet, kann die zweite Queue direkt nach der ersten mit der Berechnung seiner Slice anfangen und der Kopiervorgang kann so versteckt werden. Abbildung 4.4 zeigt einen solchen Ablauf. Da das Kopieren über PCI Express relativ langsam ist, lohnt sich diese Strategie in vielen Fällen.

5. Mehrgitterverfahren auf dem Grafikprozessor

Wir werden nun das Mehrgitterverfahren analog zu Kapitel 3 für OpenCL implementieren und zur Ausführung auf einem Grafikprozessor optimieren. Wir werden dazu Schritt für Schritt den Glätter, die Restriktion und die Prolongation implementieren und schließlich die einzelnen Komponenten zum Mehrgitterverfahren zusammensetzen. Die Implementierung soll Gebrauch von mehreren Queues machen. Die Anzahl der Queues sowie die Anzahl der Glättungsschritte soll dabei variabel bleiben. Die Implementierungen für die CPU und OpenCL werden wir miteinander vergleichen, um einen Eindruck von den Vor- und Nachteilen der jeweiligen Implementierung zu erhalten. Bezüglich der Vergleichbarkeit darf allerdings nicht außer Acht gelassen werden, dass die CPU-Implementierung ebenfalls Optimierungen zulässt, die hier aber nicht behandelt werden. Zum Beispiel wäre das geschickte Ausnutzen des Prozessorcaches vorstellbar.

5.1. Datenstruktur

Wie bereits erwähnt, verwenden wir mehrere Queues. Deshalb werden alle Aufgaben in Slices zerlegt. Das Bearbeiten einer Slice umfasst das Kopieren der benötigten Daten in den Devicespeicher, das Rechnen auf den Daten und das Zurückkopieren der Ergebnisse. Durch die Verwendung von Slices sind zum einen die berechenbaren Probleminstanzen durch die Größe des Hostspeichers und nicht durch den meist deutlich kleineren Devicespeicher begrenzt. Zum anderen lassen sich mit mehreren Queues verschiedene Slices parallel bearbeiten um Kopiervorgänge zu verstecken. Jede Queue kann dabei eine Slice zur Zeit bearbeiten.

Es sei eine feste Anzahl an Queues und somit auch die maximale Anzahl an parallel zu bearbeitenden Slices gegeben. Wie wir die Größe einer Slice wählen, wird im Wesentlichen vom globalen Devicespeicher bestimmt. Dabei spielt zum einen die Gesamtgröße des Speichers und zum anderen die maximale Größe eines zu allozierenden Speicherblocks eine Rolle. Der Grafikkarte des Testsystems steht über OpenCL 2^{29} MiByte = 512 MiByte zur Verfügung. Die maximale Größe eines zu allozierenden Speicherblocks beträgt 2^{27} MiByte = 128 MiByte. Dies entspricht einem zweidimensionalem Floatarray der Größe 5792^2.

Zur Repräsentation einer Probleminstanz verwenden wir weiterhin das Struct `grid_t`. Zur Bearbeitung einer Slice benötigen wir darüber hinaus je Queue analog zu `grid_t`

globalen Devicespeicher für einen Teil des X-Quellarrays und X-Zielarrays und des B-Arrays. Aufgrund der maximalen Größe eines zu allozierenden Speicherblocks verwenden wir anders als bei `grid_t` nicht einen großen Speicherblock, sondern je Queue drei gleich große Blöcke. Dadurch verlieren wir allerdings Flexibilität bei der Aufteilung des Speichers. Sei x die Menge des globalen Devicespeichers, den wir verwenden wollen und y die Anzahl der Queues. Die Größe eines Speicherblocks ist somit $\lfloor \frac{x}{3y} \rfloor$. Falls wir 512 MiByte globalen Devicespeicher verwenden und zwei Queues nutzen, so ist die Größe eines Speicherblocks 89478485 Byte. Dies entspricht einem zweidimensionalem Floatarray der Größe 4729^2. Zur Realisierung führen wir das Struct `slice_t` ein, die die zugehörige Queue als auch dessen Devicespeicher kennt. Die Implementierung sieht wie folgt aus.

```
1  struct _slice {
2    cl_command_queue queue; /* Zugehörige Queue             */
3    cl_mem mem0;            /* Globaler Devicespeicherblock */
4    cl_mem mem1;            /* Globaler Devicespeicherblock */
5    cl_mem mem2;            /* Globaler Devicespeicherblock */
6    cl_mem buf;             /* Pinned Hostspeicherblock     */
7    size_t mem_size;        /* Größe eines Speicherblocks   */
8  };
9  typedef struct _slice slice_t;
10 typedef slice_t *pslice_t;
```

Das Struct verweist auf die jeweilige Queue, die für die Bearbeitung zuständig ist, und besitzt wie erwähnt drei Speicherblöcke im globalen Devicespeicher. Zusätzlich besitzt das Struct in Zeile 6 noch pinned Hostspeicher. Auf dessen Nutzen werden wir später eingehen.

5.2. Glätter

5.2.1. Implementierung

Zur Realisierung des Glätters benötigen wir Code für den Host und für das Device. Der Host teilt das zweidimensionale Gitter in rechteckige Teilbereiche auf, für die abwechselnd jeweils eine andere Queue zuständig ist. Er veranlasst dann das Kopieren der Daten, die notwendig sind, um die Glättung in diesem Bereich durchzuführen, in den globalen Devicespeicher einer Slice. Dazu gehört ein Teil des X-Quellarrays und des B-Arrays. Anschließend führt er einen Kernelaufruf aus, der die Glättung in diesem Bereich durchführt, um dann das Ergebnis ins X-Zielarray zu kopieren. Damit ein Kernelaufruf einen Teil des X-Zielarrays der Größe n^2 berechnen kann, benötigt er n^2 Elemente des B-Arrays und wegen des Zugriffs auf Randelemente $(n+1)^2$ Elemente des X-Quellarrays. Wenn wir den Kernel mehrmals aufrufen wollen, um mehrere Glättungsschritte durchzuführen, bräuchten wir ab dem zweiten Schritt Randelemente, die zum Bereich einer anderen Slice gehören würden. Da es sehr ungünstig zu realisieren wäre, für die Randelemente auf Daten einer anderen Slice zuzugreifen, ist es daher sinnvoller, dass eine Slice die Randelemente für

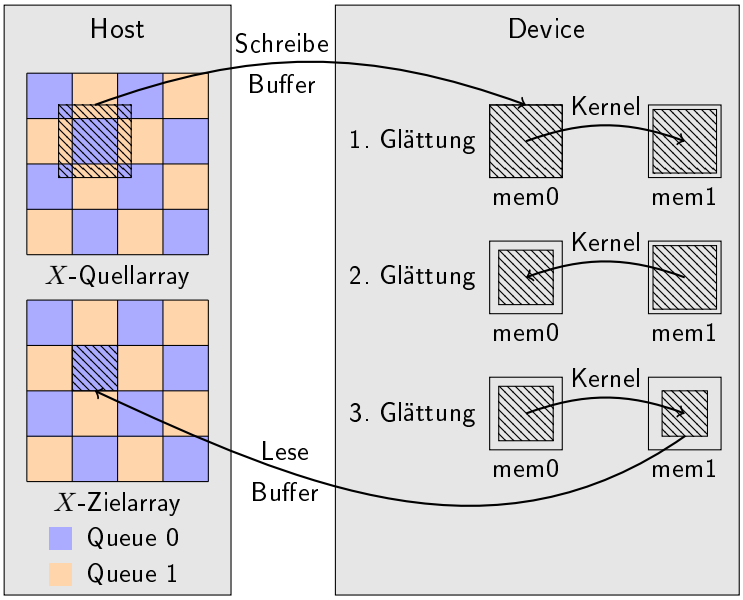


Abbildung 5.1.: Ablauf des Glätters in OpenCL mit mehreren Glättungsschritten.

weitere Glättungsschritte selber berechnet. Dadurch werden zwar insgesamt einige Elemente mehrfach berechnet, allerdings ist so nur ein Kopiervorgang vor und nach den Glättungsschritten notwendig. Um also x Glättungsschritte durchzuführen um einen Teil des X-Zielarrays der größe n^2 zu berechnen, müssen wir $(n+x-1)^2$ Elemente des B-Arrays und $(n+x)^2$ Elemente des X-Quellarrays in den globalen Devicespeicher kopieren. Anschließend rufen wir den Kernel zur Berechnung des X-Quellarrays mehrfach auf, wobei wir im ersten Glättungsschritt einen Bereich der Größe $(n+x-1)^2$, im zweiten der Größe $(n+x-2)^2$ usw. berechnen. Dabei gehen wir wie bei der Implementierung in Kapitel 5 vor und vertauschen in jedem Schritt das X-Quellarray und X-Zielarray im globalen Devicespeicher. Eine Übersicht dieses Vorgehens ist in Abbildung 5.1 zu sehen. Die Realisierung des OpenCL-C-Codes ähnelt der C-Implementierung des Glätters. Mit zwei verschachtelten Schleifen werden nach und nach die einzelnen Elemente nach Formel 2.2 berechnet. Allerdings erhält jedes Work-Item individuelle Grenzen für die Schleifen. Der zu berechnende Bereich wird vertikal auf die Work-Groups und horizontal auf die einzelnen Work-Items einer Work-Group aufgeteilt. Bei der horizontalen Aufteilung achten wir darauf, dass Work-Items mit aufeinanderfolgenden IDs horizontal aufeinanderfolgende Elemente berechnen, um die Wahrscheinlichkeit zu erhöhen, dass jede Work-Group möglichst wenige Speicherchannels zeitgleich nutzt. Die Implementierung des Kernels ist im Folgenden zu sehen. Der zugehörige C-Code befindet sich in Anhang A.4.

```
1 __kernel void kernel_smoothing(
2     __global float *slc_xs, __global float *slc_xd,
3     __global const float *slc_b, int slc_lnw, int slc_off,
4     int slc_w, int slc_h) {
```

```
5   int slc_i, slc_j, wg_j_fir, wg_j_las;
6
7   slc_xs += slc_off;
8   slc_xd += slc_off;
9   slc_b  += slc_off;
10
11  wg_j_fir = slc_h * get_group_id(0)
12      / get_num_groups(0);
13  wg_j_las = slc_h * (get_group_id(0) + 1)
14      / get_num_groups(0);
15
16  for(slc_i = get_local_id(0); slc_i < slc_w;
17      slc_i += get_local_size(0)) {
18    for(slc_j = wg_j_fir; slc_j < wg_j_las; slc_j++) {
19      slc_xd[slc_i + slc_lnw * slc_j] =
20            (slc_b[slc_i     + slc_lnw * slc_j    ]
21          + slc_xs[(slc_i-1) + slc_lnw * slc_j    ]
22          + slc_xs[(slc_i+1) + slc_lnw * slc_j    ]
23          + slc_xs[slc_i     + slc_lnw * (slc_j-1)]
24          + slc_xs[slc_i     + slc_lnw * (slc_j+1)])
25          * 0.25f;
26    }
27  }
28 }
```

Wir wollen nun die CPU- und GPU-Implementierung des Glätters vergleichen. Zum Einsatz kommt als CPU eine Intel i5-750 und als GPU eine AMD HD6870. Als Konfiguration verwenden wir vier Queues, 10752 Work-Items und eine Work-Group-Größe von 128. Für die Probleminstanz verwenden wir ein Gitter der Größe 8191^2. Folgende Zeiten wurden gemessen.

Glättungsschritte	2	4	8	16
CPU	0,31s	0,61s	1,22s	2,46s
GPU	0,68s	0,71s	0,79s	0,92s

Auffällig ist, dass die benötigte Zeit bei der CPU linear mit den Glättungsschritten ansteigt, während bei der GPU die Zeiten nur geringfügig ansteigen. Da man in der Praxis nur sehr wenig Glättungsschritte einsetzt, ist es sehr ungünstig, dass bei zwei Glättungsschritten die CPU-Implementierung schneller ist. Mehr als zwei Glättungsschritte bieten dagegen nur eine sehr geringe Verbesserung im Konvergenzverhalten. Durch mehr Glättungsschritte wird allerdings die höhere Rechenleistung der GPU gegenüber der CPU deutlich, da mit zunehmendem Rechenaufwand die Transferoperationen besser versteckt werden können.

5.2.2. Speichertransfer

Die Analyse des Ausführungsverlaufs des Programms mit Entwicklungswerkzeugen von AMD[1] macht deutlich, dass die Speichertransferoperationen einen Großteil der Zeit beanspruchen. Wir führen im Folgenden einige Tests durch, um uns eine Übersicht der Transferraten zu verschaffen. Dazu lesen und schreiben wir parallel mit Hilfe von vier Queues größere Menge an Daten. Ein erster Test der Funktionen `clEnqueueWriteBufferRect` und `clEnqueueReadBufferRect`, die wir bei der Glättung zum Kopieren von Daten verwenden, ergibt eine Transferrate von 1,43 GBps. Dieser Wert ist ernüchternd. Da die Grafikkarte mit 16 Lanes über PCI Express 2.0 angebunden ist, wäre das theoretische Maximum 8 GBps. Wie in Abschnitt 4.3 erwähnt, lassen sich Daten zwischen Host und Device nur bei der Verwendung von pinned Speicher optimal übertragen, da kein weiterer pinned Speicher als Zwischenspeicher notwendig wird. Wenn wir mit `clEnqueueCopyBuffer` Daten zwischen pinned Speicher und Devicespeicher hin- und zurückkopieren, erreichen wir eine Transferrate von 6 GBps. Da wir pinned Speicher nicht in beliebiger Göße anfordern können, um alle Daten abzulegen, müssen wir normalen Speicher verwenden und können diesen Wert nicht erreichen. Für normalen Hostspeicher erreichen wir mit den Funktionen `clEnqueueWriteBuffer` bzw. `clEnqueueReadBuffer` aber immer noch eine Transferrate von 3,82 GBps. Der Unterschied von `clEnqueueWriteBufferRect` bzw. `clEnqueueReadBufferRect` zu diesen Funktionen ist, dass kein zusammenhängender Speicherbereich, sondern einzelne Speicherzeilen aus einem größeren Speicherbereich kopiert werden. Der Verdacht liegt nahe, dass die Implementierung dieser Funktionen ungünstig ist. Eine Möglichkeit, dieses Problem zu umgehen, ist, das Zwischenspeichern im pinned Speicher selbst zu übernehmen. Dazu haben wir im Struct `slice_t` den Buffer `buf`, das wir mit Hilfe von OpenCL als pinned Hostspeicher der Größe `mem_size` initialisieren. Bei einem Schreibvorgang kopieren wir also die Daten vom normalen Speicher mit `clEnqueueWriteBufferRect` in `buf` und verwenden anschließend `clEnqueueCopyBuffer`, um sie von `buf` in den Devicespeicher zu kopieren. In umgekehrter Reihenfolge lassen sich Daten mit `clEnqueueReadBufferRect` und `buf` zurückkopieren. Insgesamt erreichen wir so eine Transferrate von 3,97 GBps. Interessanterweise übertrifft dieser Wert bei den Messungen sogar die Transferrate von `clEnqueueWriteBuffer` bzw. `clEnqueueReadBuffer`. Es sei dabei zu erwähnen, dass die genannten Werte nur unter Verwendung von vier Queues erreicht werden, während bei Verwendung weniger Queues die Transferraten teilweise sinken. Vor allem das Kopieren von Daten zwischen normalem Hostspeicher und pinned Hostspeicher scheint prinzipiell parallel zu Transferoperationen über PCI Express stattzufinden. Eine Übersicht über alle gemessenen Transferraten bietet folgende Tabelle.

[1] Es wird das Programm CodeXL verwendet. Zu finden unter `http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/`

	Schreiben	Lesen	Schreiben und Lesen parallel
`clEnqueueWriteBufferRect`/ `clEnqueueReadBufferRect`	0,93 GBps	2,17 GBps	1,43 GBps
`clEnqueueCopyBuffer`	5,91 GBps	6,1 GBps	6 GBps
`clEnqueueWriteBuffer`/ `clEnqueueReadBuffer`	3,16 GBps	4,09 GBps	3,82 GBps
`clEnqueueWriteBufferRect`/ `clEnqueueReadBufferRect` (Verwendung von `buf`)	3,96 GBps	4,13 GBps	3,97 GBps

Wir verwenden nun zum Kopieren von Daten die Variante mit `buf`. Unter Beibehaltung der Konfiguration wurden nochmals die Zeiten des Glätters gemessen.

Glättungsschritte	2	4	8	16
CPU	0,31s	0,61s	1,22s	2,46s
GPU	0,68s	0,71s	0,79s	0,92s
GPU (Verwendung von `buf`)	0,24s	0,24s	0,24s	0,29s

Eine deutliche Verbesserung ist bemerkbar. Dennoch ist der Nutzen der GPU bei wenigen Glättungsschritten sehr gering, obwohl sie eine sehr schnelle Anbindung an ihren Speicher und viel Rechenleistung besitzt. Um bei zwei Glättungsschritten eine viermal bessere Zeit auf der GPU als auf der CPU zu erreichen, brauchen wir eine höhere Datentransferrate. Die CPU benötigt zur Berechnung 0,31s. Die Mengen der Daten, die zu übertragen ist, beträgt ungefähr $3 \cdot (8192^2 \cdot 4)$ Byte = 768 MiByte. Somit benötigen wir mindestens eine effektive Transferrate zwischen Host- und Devicespeicher von $4 \cdot \frac{768 \text{ MiByte}}{0{,}31\text{s}} = 10{,}39$ GBps für eine viermal schnellere Berechnung.

5.2.3. Lokaler Speicher

Neben den Transferoperationen können wir den aufzurufenden Kernel selbst optimieren. Sinnvoll ist dies in erster Linie nur dann, wenn das Programm nicht durch den Speichertransfer limitiert wird. In unserem Fall wäre dies also nicht unbedingt notwendig. Allerdings ließe sich so dennoch die Anzahl der Glättungsschritte ohne Mehrkosten erhöhen.

Den Flaschenhals eines Kernels stellt oft der Speicherzugriff dar. Wir verwenden bisher hauptsächlich den globalen Devicespeicher. Eine sinnvolle Möglichkeit zur Entlastung seiner Speicherchannels ist die zusätzliche Verwendung einer anderen Speicherart des Devices. Dazu bietet sich beispielsweise der lokale Devicespeicher an, der deutlich schneller aber auch kleiner als der globale Devicespeicher ist. Er eignet sich daher gut für eine Verwendung als Cache. Da jede Compute Unit eigenen lokalen Devicespeicher besitzt, behindern sie sich somit nicht gegenseitig beim Zugriff auf ihren lokalen Devicespeicher. Der lokale Speicher besitzt ebenfalls Speicherchannels. Allerdings ist hier die gleichmäßige Auslastung aller Channels durch

eine Wavefront sinnvoll. Da der Speicher von allen Wavefronts einer Compute Unit geteilt wird, beeinflusst die Menge des genutzten lokalen Speichers je Wavefront die Anzahl der Wavefronts, die einer Compute Unit zugewiesen werden können.

Unser Kernel zur Glättung teilt die Berechnung derart auf, dass jedes Work-Item nacheinander Spalten berechnet. Für jede Spalte berechnet ein Work-Item der Reihe nach die einzelnen Elemente. Zur Berechnung eines Elements ist der Zugriff auf vier benachbarte Elemente erforderlich, die sich insgesamt in drei Zeilen des Arrays befinden. Aufgrund der großen Zeilenbreite erfordern diese Nachbarn sehr wahrscheinlich Zugriffe auf drei verschiedene Speicherchannels. Um diese Zugriffe zu minimieren, können wir den lokalen Speicher verwenden. Jedes Work-Item kriegt seinen eigenen Buffer im lokalen Devicespeicher, den wir als eine Art Cache nutzen. Für die Berechnung eines Elements greift das Work-Item nun nur auf eine Zeile zu, während sich die zwei vorherigen Zeilen im Cache befinden. Die Implementierung in OpenCL C dazu befindet sich im Anhang A.1. Es wurden erneut Messungen des Glätters mit den gleichen Konfiguration durchgeführt.

Glättungsschritte	2	4	8	16
CPU	0,31s	0,61s	1,22s	2,46s
GPU	0,68s	0,71s	0,79s	0,92s
GPU (Verwendung von `buf`)	0,24s	0,24s	0,24s	0,29s
GPU (Verwendung von `buf` und Cache)	0,23s	0,24s	0,24s	0,24s

Den einzigen erkennbaren Unterschied können wir bei 16 Iterationen feststellen. Bei weniger Iterationen lässt sich aufgrund der gemessenen Zeiten auf eine Begrenzung durch den Datentransfer schließen. Durch eine Analyse mit den Entwicklungswerkzeugen von AMD lassen sich allerdings weitere Informationen gewinnen. Der Kernel ohne Cache benötigt für eine Slicegröße von 2048^2 zur Ausführung im Durchschnitt 1,15ms und lastet die Rechenwerke zu 20% aus. Mit Cache sind es durchschnittlich 0,78ms. Die Auslastung beträgt 80%, was eine deutliche Verbesserung ist. Insgesamt lässt sich sagen, dass die Nutzung anderer Speicherarten zur Entlastung des globalen Devicespeichers zwar äußerst lohnenswert ist, aber wegen der Limitierung durch die Speichertransferrate sollte im Folgenden in erster Linie auf das Einsparen von Transferoperationen geachtet werden.

5.3. Restriktion

Durch den Verzicht des Vorglätters beim Mehrgitterverfahren nach Abbildung 3.1 sind bei der Realisierung der Restriktion zusätzliche Optimierungen möglich. Ein Durchlauf des Mehrgitterverfahrens beginnt mit der wiederholten Anwendung der Restriktion. Genauso wie bei der vorangegangenen Realisierung des Glätters können wir mehrere Restriktionen nacheinander durchführen, ohne zwischen den einzelnen Schritten Daten zwischen Host und Device austauschen zu müssen. Somit müssen wir weniger Daten kopieren und können auf den kopierten Daten mehr Berechnungen durchführen. Zur Realisierung teilen wir das gröbste Gitter, das nach den

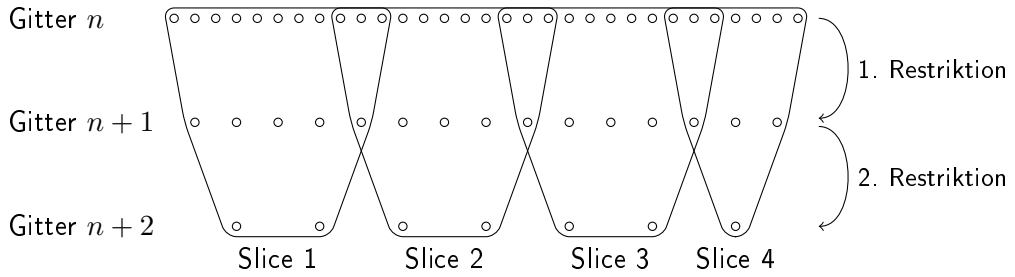


Abbildung 5.2.: Beispiel zur Aufteilung in Slices zur Restriktion. Zur Veranschaulichung wird ein eindimensionales Gitter verwendet. Die Aufteilung findet nach dem gröbsten Gitter statt.

Restriktionen entsteht, in so große Teile auf, dass das zur Berechnung notwendige Teilstück des Ausgangsgitters in den Devicespeicher der Slice passt. Dieser Ansatz wird in Abbildung 5.2 verdeutlicht. Analog zur C-Implementierung muss nun für jede Slice wiederholt das Residuum gebildet und die Restriktion angewandt werden bis das gröbste Gitter erreicht ist. Die Implementierung befindet sich in Anhang A.2 und A.5.

5.4. Prolongation

Bezüglich der Prolongation fällt beim Mehrgitterverfahren nach Abbildung 3.1 auf, dass auf eine Prolongation immer der Glätter folgt. Die Prolongation arbeitet auf dem X-Quellarray des jeweils feinen und groben Gitters, während der Glätter das B-Array und das X-Quellarray des zu glättenden Gitters benötigt. Es liegt nahe, direkt nach der Prolongation die Glättung des feinen Gitters durchzuführen, um den Transfer des X-Quellarrays bei der Glättung einzusparen. Die Realisierung der Prolongation erfolgt analog zur C-Implementierung und befindet sich in Anhang A.3 und A.6.

5.5. Mehrgitterverfahren

Mit Hilfe des Glätters, der Restriktion und Prolongation lässt sich nun das Mehrgitterverfahren umsetzen. Dabei macht es Sinn, eine wiederholte Anwendung des Mehrgitterverfahrens zu berücksichtigen. In der Praxis ist es nämlich notwendig, das Mehrgitterverfahren zahlreiche Male zu wiederholen, um eine geeignete Näherung zu erhalten. Bei einem Durchlauf des Mehrgitterverfahrens führen wir dabei als erstes eine Restriktion vom feinsten Gitter durch. Als Letztes nehmen wir bei einem Durchlauf eine Glättung des feinsten Gitters vor. Es macht somit Sinn, zwei aufeinanderfolgende Durchläufe des Mehrgitterverfahrens zu verknüpfen und nach der Glättung des feinsten Gitters direkt die Restriktion vorzunehmen. Angesichts der Tatsache, dass das feinste Gitter ungefähr 66% der Gesamtdaten ausmacht,

bringt uns dies eine weitere beachtliche Ersparnis an Transferoperationen. Folgende Transferoperationen des feinsten Gitters bleiben übrig.

- Zur Prolongation auf das feinste Gitter muss das X-Quellarray kopiert werden.
- Zur Glättung muss das B-Array kopiert werden.
- Nach der Glättung muss die verbesserte Näherung ins X-Zielarray kopiert werden.

Wir benötigen also genauso viele Transferoperationen des feinsten Gitters wie bei der alleinigen Anwendung des Glätters. Zusätzlich führen wir nun noch die Prolongation und Restriktion durch und können so während des Transfers mehr rechnen. Messungen zum Vergleich des Mehrgitterverfahrens auf CPU und GPU müssten somit zu ähnlichen Ergebnissen wie beim Glätter kommen. Folgende Zeiten konnten unter der gleichen Konfiguration und bei zehn Durchläufen des Mehrgitterverfahrens gemessen werden.

Glättungsschritte	2	4	8	16
CPU	7,56s	11,67s	19,91s	36,38s
GPU	4,2s	4,28s	4,88s	6,06s

Es ist eine leichte Verbesserung gegenüber den Tests mit dem Glätter zu erkennen. Bei zwei Glättungsschritten benötigt die GPU beinahe nur die Hälfte der Zeit. Dies ist zwar immer noch weit von dem Ziel einer viermal schnelleren Lösung entfernt. Dennoch konnte das Einsparen von Transferoperationen den Vorsprung vergrößern. Betrachten wir nun mehr Glättungsschritte, sehen wir, dass bei mehr Rechenaufwand die Vorteile der GPU zur Geltung kommen. Bei 16 Glättungsschritten kann eine sechsmal bessere Zeit erreicht werden. Wird die Anzahl weiter erhöht, dann stagniert die GPU bei 128 Glättungsschritten bei einer etwa neunmal schnelleren Zeit.

6. Fazit

Die Berechnung des gegebenen Mehrgitterverfahrens konnte bei zwei Glättungsschritten auf dem Grafikprozessor fast doppelt so schnell durchgeführt werden wie auf dem Hauptprozessor. Um dieses Ergebnis zu erreichen, mussten im Wesentlichen Optimierungen hinsichtlich des Datentransfers vorgenommen werden, da sich Daten zwischen dem Haupt- und Grafikspeicher mit einer nur relativ geringen Transferrate übertragen lassen. Das Ziel dieser Arbeit, die Berechnung mit dem Grafikprozessor bei einer sinnvollen Anzahl an Glättungsschritten viermal schneller durchzuführen, konnte nicht erreicht werden. Durch die Erhöhung der Anzahl der Glättungsschritte ließ sich der Berechnungsaufwand künstlich anheben, so dass die Rechenzeit die Datentransferzeit überstieg. So konnte auf dem Grafikprozessor die Berechnung des Mehrgitterverfahrens bis zu neunmal schneller durchgeführt werden. Wenn in Zukunft die Einschränkungen des getrennten Haupt- und Grafikspeichers gemindert oder ganz aufgehoben werden, dann wird der Einsatz von Grafikprozessoren zur Berechnung der vorliegenden Problemstellung durchaus lohnenswert. Eine mögliche Verbesserung ist dabei zum einen die Anbindung der Grafikkarte über einen schnelleren Bus. Daher ist auf die Verbreitung von PCI Express 3.0 zu hoffen, das gegenüber 2.0 eine Verdopplung der Transferrate bietet. Eine weitere Verbesserung stellt der immer größer werdende Grafikspeicher dar. Sobald der Grafikspeicher groß genug ist, um die vollständige Probleminstanz aufzunehmen, fällt das Hin- und Herkopieren zwischen Haupt- und Grafikspeicher während den Berechnungen komplett weg. Die Daten müssen erst zum Hauptspeicher zurückkopiert werden, sobald der Hauptprozessor sie benötigt. Eine interessante zukünftige Technologie in diesem Zusammenhang stellt *heterogeneous Uniform Memory Access* (hUMA) von AMD dar. Die Idee ist ein von Haupt- und Grafikprozessor gemeinsam genutzter Speicher. Somit können beide auf den Daten arbeiten, ohne dass ein Datentransfer notwendig ist. Insgesamt ist zu erwarten, dass Grafikprozessoren für allgemeine Berechnungen immer interessanter werden.

A. OpenCL-Implementierung

A.1. OpenCL C Glätter

```
/* Führt Glättungsschritt aus       */
/* Nutzt lokalen Speicher als Cache */
__kernel void kernel_smoothing_cache(
    __global float *slc_xs, __global float *slc_xd,
    __global const float *slc_b, int slc_lnw, int slc_off,
    int slc_w, int slc_h) {
  int slc_i, slc_j, wg_j_fir, wg_j_las, lid, lsize, c_idx;
  __local float c[3][128*3];

  slc_xs += slc_off;
  slc_xd += slc_off;
  slc_b  += slc_off;

  wg_j_fir = slc_h * get_group_id(0)
      / get_num_groups(0);
  wg_j_las = slc_h * (get_group_id(0) + 1)
      / get_num_groups(0);
  lsize    = get_local_size(0);
  lid      = get_local_id(0);

  for(slc_i = lid; slc_i < slc_w; slc_i += lsize) {
    slc_j = wg_j_fir;

    c[0][lid          ] =
        slc_xs[slc_i-1 + slc_lnw * (slc_j-1)];
    c[0][lid+(lsize*1)] =
        slc_xs[slc_i   + slc_lnw * (slc_j-1)];
    c[0][lid+(lsize*2)] =
        slc_xs[slc_i+1 + slc_lnw * (slc_j-1)];
    c[1][lid          ] =
        slc_xs[slc_i-1 + slc_lnw * slc_j    ];
    c[1][lid+(lsize*1)] =
        slc_xs[slc_i   + slc_lnw * slc_j    ];
    c[1][lid+(lsize*2)] =
        slc_xs[slc_i+1 + slc_lnw * slc_j    ];
    c_idx = 2;

    for(slc_j = wg_j_fir; slc_j < wg_j_las; slc_j++) {
      c[c_idx][lid          ] =
```

```
          slc_xs[slc_i-1 + slc_lnw * (slc_j+1)];
      c[c_idx][lid+(lsize*1)] =
          slc_xs[slc_i   + slc_lnw * (slc_j+1)];
      c[c_idx][lid+(lsize*2)] =
          slc_xs[slc_i+1 + slc_lnw * (slc_j+1)];
      c_idx = (c_idx + 1) % 3;

      slc_xd[slc_i + slc_lnw * slc_j] =
          (slc_b[slc_i + slc_lnw * slc_j]
          + c[(c_idx+1)%3][lid          ]
          + c[(c_idx+1)%3][lid+(lsize*2)]
          + c[c_idx      ][lid+(lsize*1)]
          + c[(c_idx+2)%3][lid+(lsize*1)]) * 0.25f;
    }
  }
}
```

A.2. OpenCL C Restriktion

```
/* Residuum berechnen */
__kernel void kernel_calculate_error(
    __global float *slc_x, int slc_x_off, int slc_x_lnw,
    __global float *slc_b, int slc_b_off, int slc_b_lnw,
    int slc_w, int slc_h) {
  int slc_i, slc_j, wg_j_fir, wg_j_las;

  slc_x += slc_x_off;
  slc_b += slc_b_off;

  wg_j_fir = slc_h * get_group_id(0)
      / get_num_groups(0);
  wg_j_las = slc_h * (get_group_id(0) + 1)
      / get_num_groups(0);

  for(slc_i = get_local_id(0);
      slc_i < slc_w;
      slc_i += get_local_size(0)) {
    for(slc_j = wg_j_fir; slc_j < wg_j_las; slc_j++) {
      slc_b[slc_i + slc_b_lnw * slc_j] =
            slc_b[slc_i     + slc_b_lnw * slc_j    ]
          + slc_x[(slc_i-1) + slc_x_lnw * slc_j    ]
          + slc_x[(slc_i+1) + slc_x_lnw * slc_j    ]
          + slc_x[slc_i     + slc_x_lnw * (slc_j-1)]
          + slc_x[slc_i     + slc_x_lnw * (slc_j+1)]
          - slc_x[slc_i     + slc_x_lnw * slc_j    ] * 4.f;
    }
  }
}
```

```
/* Führt Restriktion aus */
__kernel void kernel_restriction(
    __global float *slc_f_b, int slc_f_b_lnw, int slc_f_b_off,
    __global float *slc_c_b, int slc_c_b_lnw, int slc_c_b_off,
    int slc_c_w, int slc_c_h) {
  int slc_c_i, slc_c_j, slc_f_i, slc_f_j, wg_j_fir, wg_j_las;

  slc_f_b += slc_f_b_off;
  slc_c_b += slc_c_b_off;

  wg_j_fir = slc_c_h * get_group_id(0)
    / get_num_groups(0);
  wg_j_las = slc_c_h * (get_group_id(0) + 1)
    / get_num_groups(0);

  for(slc_c_i = get_local_id(0);
      slc_c_i < slc_c_w;
      slc_c_i += get_local_size(0)) {
    slc_f_i = 2 * slc_c_i + 1;
    for(slc_c_j = wg_j_fir; slc_c_j < wg_j_las; slc_c_j++) {
      slc_f_j = 2 * slc_c_j + 1;

      slc_c_b[slc_c_i + slc_c_b_lnw * slc_c_j] =
         -  slc_f_b[slc_f_i     + slc_f_b_lnw * slc_f_j    ]
         - (slc_f_b[(slc_f_i-1) + slc_f_b_lnw * slc_f_j    ]
         +  slc_f_b[(slc_f_i-1) + slc_f_b_lnw * (slc_f_j+1)]
         +  slc_f_b[slc_f_i     + slc_f_b_lnw * (slc_f_j+1)]
         +  slc_f_b[(slc_f_i+1) + slc_f_b_lnw * slc_f_j    ]
         +  slc_f_b[(slc_f_i+1) + slc_f_b_lnw * (slc_f_j-1)]
         +  slc_f_b[slc_f_i     + slc_f_b_lnw * (slc_f_j-1)])
         * 0.5f;
    }
  }
}
```

A.3. OpenCL C Prolongation

```
/* Führt Prolongation aus                    */
/* Implizite Initialisierung von X mit 0en */
__kernel void kernel_prolongation(
    __global float *slc_f_x, int slc_f_lnw, int slc_f_off,
    __global float *slc_c_x, int slc_c_lnw, int slc_c_off,
    int slc_c_w, int slc_c_h, int mask) {
  int slc_c_i, slc_c_j, slc_f_i, slc_f_j,
      wg_i_fir, wg_i_las, wg_j_fir, wg_j_las,
      the_one_and_only;
  float help1, help2, help3, help4;

  slc_f_x += slc_f_off;
```

```
slc_c_x += slc_c_off;
wg_i_fir = slc_c_w * get_group_id(0)
    / get_num_groups(0);
wg_i_las = slc_c_w * (get_group_id(0) + 1)
    / get_num_groups(0);
wg_j_fir = slc_c_h * get_group_id(0)
    / get_num_groups(0);
wg_j_las = slc_c_h * (get_group_id(0) + 1)
    / get_num_groups(0);
the_one_and_only =
    get_group_id(0) == 0 && get_local_id(0) == 0;

/* Hauptbereich */
for(slc_c_i = get_local_id(0);
    slc_c_i < slc_c_w;
    slc_c_i += get_local_size(0)) {
  slc_f_i = 2 * slc_c_i;
  for(slc_c_j = wg_j_fir; slc_c_j < wg_j_las; slc_c_j++) {
    slc_f_j = 2 * slc_c_j;

    help1 = slc_c_x[slc_c_i + slc_c_lnw * slc_c_j];
    help2 = help1 / 2;
    help3 = slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2;
    help4 = slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2;
    slc_f_x[slc_f_i     + slc_f_lnw * slc_f_j    ] =
        -help4 - help3;
    slc_f_x[(slc_f_i+1) + slc_f_lnw * slc_f_j    ] =
        -help4 - help2;
    slc_f_x[slc_f_i     + slc_f_lnw * (slc_f_j+1)] =
        -help3 - help2;
    slc_f_x[(slc_f_i+1) + slc_f_lnw * (slc_f_j+1)] =
        -help1;
  }
}

/* Randbehandlung */
if(mask & 1) {
  slc_c_j = -1;
  slc_f_j = 2 * slc_c_j;
  for(slc_c_i = wg_i_fir + get_local_id(0);
      slc_c_i < wg_i_las; slc_c_i +=
      get_local_size(0)) {
    slc_f_i = 2 * slc_c_i;

    help1 = slc_c_x[slc_c_i + slc_c_lnw * slc_c_j];
    help2 = help1 / 2;
    help3 = slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2;
    help4 = slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2;
    slc_f_x[slc_f_i     + slc_f_lnw * (slc_f_j+1)] =
```

```
        -help3 - help2;
    slc_f_x[(slc_f_i+1) + slc_f_lnw * (slc_f_j+1)] =
        -help1;
  }
}
if((mask & 1) && (mask & 2) && the_one_and_only) {
  slc_c_i = slc_c_w;
  slc_f_i = 2 * slc_c_i;
  slc_c_j = -1;
  slc_f_j = 2 * slc_c_j;
  slc_f_x[slc_f_i     + slc_f_lnw * (slc_f_j+1)] =
      - slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2
      - slc_c_x[slc_c_i + slc_c_lnw * slc_c_j] / 2;
}
if(mask & 2) {
  slc_c_i = slc_c_w;
  slc_f_i = 2 * slc_c_i;
  for(slc_c_j = wg_j_fir + get_local_id(0);
      slc_c_j < wg_j_las;
      slc_c_j += get_local_size(0)) {
    slc_f_j = 2 * slc_c_j;

    help2 = slc_c_x[slc_c_i + slc_c_lnw * slc_c_j] / 2;
    help3 = slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2;
    help4 = slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2;
    slc_f_x[slc_f_i     + slc_f_lnw * slc_f_j    ] =
        -help4 - help3;
    slc_f_x[slc_f_i     + slc_f_lnw * (slc_f_j+1)] =
        -help3 - help2;
  }
}
if((mask & 2) && (mask & 4) && the_one_and_only) {
  slc_c_i = slc_c_w;
  slc_f_i = 2 * slc_c_i;
  slc_c_j = slc_c_h;
  slc_f_j = 2 * slc_c_j;
  slc_f_x[slc_f_i     + slc_f_lnw * slc_f_j    ] =
      - slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2
      - slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2;
}
if(mask & 4) {
  slc_c_j = slc_c_h;
  slc_f_j = 2 * slc_c_j;
  for(slc_c_i = wg_i_fir + get_local_id(0);
      slc_c_i < wg_i_las;
      slc_c_i += get_local_size(0)) {
    slc_f_i = 2 * slc_c_i;

    help2 = slc_c_x[slc_c_i + slc_c_lnw * slc_c_j] / 2;
```

```
      help3 = slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2;
      help4 = slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2;
      slc_f_x[slc_f_i     + slc_f_lnw * slc_f_j    ] =
          -help4 - help3;
      slc_f_x[(slc_f_i+1) + slc_f_lnw * slc_f_j    ] =
          -help4 - help2;
    }
  }
  if((mask & 4) && (mask & 8) && the_one_and_only) {
    slc_c_i = -1;
    slc_f_i = 2 * slc_c_i;
    slc_c_j = slc_c_h;
    slc_f_j = 2 * slc_c_j;
    slc_f_x[(slc_f_i+1) + slc_f_lnw * slc_f_j    ] =
        - slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2
        - slc_c_x[slc_c_i + slc_c_lnw * slc_c_j] / 2;
  }
  if(mask & 8) {
    slc_c_i = -1;
    slc_f_i = 2 * slc_c_i;
    for(slc_c_j = wg_j_fir + get_local_id(0);
        slc_c_j < wg_j_las;
        slc_c_j += get_local_size(0)) {
      slc_f_j = 2 * slc_c_j;

      help1 = slc_c_x[slc_c_i + slc_c_lnw * slc_c_j];
      help2 = help1 / 2;
      help4 = slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2;
      slc_f_x[(slc_f_i+1) + slc_f_lnw * slc_f_j    ] =
          -help4 - help2;
      slc_f_x[(slc_f_i+1) + slc_f_lnw * (slc_f_j+1)] =
          -help1;
    }
  }
  if((mask & 8) && (mask & 1) && the_one_and_only) {
    slc_c_i = -1;
    slc_f_i = 2 * slc_c_i;
    slc_c_j = -1;
    slc_f_j = 2 * slc_c_j;
    slc_f_x[(slc_f_i+1) + slc_f_lnw * (slc_f_j+1)] =
        - slc_c_x[slc_c_i + slc_c_lnw * slc_c_j];
  }
}

/* Führt Prolongation auf das feinste Gitter aus */
__kernel void kernel_prolongation_finest(
    __global float *slc_f_x, int slc_f_lnw, int slc_f_off,
    __global float *slc_c_x, int slc_c_lnw, int slc_c_off,
    int slc_c_w, int slc_c_h, int mask) {
```

```
int slc_c_i, slc_c_j, slc_f_i, slc_f_j, wg_i_fir,
    wg_i_las, wg_j_fir, wg_j_las,
    the_one_and_only;
float help1, help2, help3, help4;

slc_f_x += slc_f_off;
slc_c_x += slc_c_off;
wg_i_fir = slc_c_w * get_group_id(0)
    / get_num_groups(0);
wg_i_las = slc_c_w * (get_group_id(0) + 1)
    / get_num_groups(0);
wg_j_fir = slc_c_h * get_group_id(0)
    / get_num_groups(0);
wg_j_las = slc_c_h * (get_group_id(0) + 1)
    / get_num_groups(0);
the_one_and_only =
    get_group_id(0) == 0 && get_local_id(0) == 0;

/* Hauptbereich */
for(slc_c_i = get_local_id(0);
    slc_c_i < slc_c_w;
    slc_c_i += get_local_size(0)) {
  slc_f_i = 2 * slc_c_i;
  for(slc_c_j = wg_j_fir; slc_c_j < wg_j_las; slc_c_j++) {
    slc_f_j = 2 * slc_c_j;

    help1 = slc_c_x[slc_c_i + slc_c_lnw * slc_c_j];
    help2 = help1 / 2;
    help3 = slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2;
    help4 = slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2;
    slc_f_x[slc_f_i     + slc_f_lnw * slc_f_j    ] +=
        -help4 - help3;
    slc_f_x[(slc_f_i+1) + slc_f_lnw * slc_f_j    ] +=
        -help4 - help2;
    slc_f_x[slc_f_i     + slc_f_lnw * (slc_f_j+1)] +=
        -help3 - help2;
    slc_f_x[(slc_f_i+1) + slc_f_lnw * (slc_f_j+1)] +=
        -help1;
  }
}

/* Randbehandlung */
if(mask & 1) {
  slc_c_j = -1;
  slc_f_j = 2 * slc_c_j;
  for(slc_c_i = wg_i_fir + get_local_id(0);
      slc_c_i < wg_i_las;
      slc_c_i += get_local_size(0)) {
    slc_f_i = 2 * slc_c_i;
```

```
    help1 = slc_c_x[slc_c_i + slc_c_lnw * slc_c_j];
    help2 = help1 / 2;
    help3 = slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2;
    help4 = slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2;
    slc_f_x[slc_f_i     + slc_f_lnw * (slc_f_j+1)] +=
        -help3 - help2;
    slc_f_x[(slc_f_i+1) + slc_f_lnw * (slc_f_j+1)] +=
        -help1;
  }
}
if((mask & 1) && (mask & 2) && the_one_and_only) {
  slc_c_i = slc_c_w;
  slc_f_i = 2 * slc_c_i;
  slc_c_j = -1;
  slc_f_j = 2 * slc_c_j;
  slc_f_x[slc_f_i     + slc_f_lnw * (slc_f_j+1)] +=
      - slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2
      - slc_c_x[slc_c_i + slc_c_lnw * slc_c_j] / 2;
}
if(mask & 2) {
  slc_c_i = slc_c_w;
  slc_f_i = 2 * slc_c_i;
  for(slc_c_j = wg_j_fir + get_local_id(0);
      slc_c_j < wg_j_las;
      slc_c_j += get_local_size(0)) {
    slc_f_j = 2 * slc_c_j;

    help2 = slc_c_x[slc_c_i + slc_c_lnw * slc_c_j] / 2;
    help3 = slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2;
    help4 = slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2;
    slc_f_x[slc_f_i     + slc_f_lnw * slc_f_j    ] +=
        -help4 - help3;
    slc_f_x[slc_f_i     + slc_f_lnw * (slc_f_j+1)] +=
        -help3 - help2;
  }
}
if((mask & 2) && (mask & 4) && the_one_and_only) {
  slc_c_i = slc_c_w;
  slc_f_i = 2 * slc_c_i;
  slc_c_j = slc_c_h;
  slc_f_j = 2 * slc_c_j;
  slc_f_x[slc_f_i     + slc_f_lnw * slc_f_j    ] +=
      - slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2
      - slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2;
}
if(mask & 4) {
  slc_c_j = slc_c_h;
  slc_f_j = 2 * slc_c_j;
```

```
    for(slc_c_i = wg_i_fir + get_local_id(0);
        slc_c_i < wg_i_las;
        slc_c_i += get_local_size(0)) {
      slc_f_i = 2 * slc_c_i;

      help2 = slc_c_x[slc_c_i + slc_c_lnw * slc_c_j] / 2;
      help3 = slc_c_x[(slc_c_i-1) + slc_c_lnw * slc_c_j] / 2;
      help4 = slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2;
      slc_f_x[slc_f_i     + slc_f_lnw * slc_f_j    ] +=
          -help4 - help3;
      slc_f_x[(slc_f_i+1) + slc_f_lnw * slc_f_j    ] +=
          -help4 - help2;
    }
  }
  if((mask & 4) && (mask & 8) && the_one_and_only) {
    slc_c_i = -1;
    slc_f_i = 2 * slc_c_i;
    slc_c_j = slc_c_h;
    slc_f_j = 2 * slc_c_j;
    slc_f_x[(slc_f_i+1) + slc_f_lnw * slc_f_j    ] +=
        - slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2
        - slc_c_x[slc_c_i + slc_c_lnw * slc_c_j] / 2;
  }
  if(mask & 8) {
    slc_c_i = -1;
    slc_f_i = 2 * slc_c_i;
    for(slc_c_j = wg_j_fir + get_local_id(0);
        slc_c_j < wg_j_las;
        slc_c_j += get_local_size(0)) {
      slc_f_j = 2 * slc_c_j;

      help1 = slc_c_x[slc_c_i + slc_c_lnw * slc_c_j];
      help2 = help1 / 2;
      help4 = slc_c_x[slc_c_i + slc_c_lnw * (slc_c_j-1)] / 2;
      slc_f_x[(slc_f_i+1) + slc_f_lnw * slc_f_j    ] +=
          -help4 - help2;
      slc_f_x[(slc_f_i+1) + slc_f_lnw * (slc_f_j+1)] +=
          -help1;
    }
  }
  if((mask & 8) && (mask & 1) && the_one_and_only) {
    slc_c_i = -1;
    slc_f_i = 2 * slc_c_i;
    slc_c_j = -1;
    slc_f_j = 2 * slc_c_j;
    slc_f_x[(slc_f_i+1) + slc_f_lnw * (slc_f_j+1)] +=
        - slc_c_x[slc_c_i + slc_c_lnw * slc_c_j];
  }
}
```

A.4. C Glätter

```
/* Hilfsfunktion Glätter */
void smoothing_calc(
    cl_kernel krn_smth, pslice_t slc,
    int slc_hrange, int slc_vrange, int slc_brd, int smth_cnt,
    int i, int j, int glo_w, int glo_h, int glo_i, int glo_j,
    int slc_lnw) {
  int k, glo_crr_i, glo_crr_j, slc_crr_w, slc_crr_h, slc_off;
  cl_int result;
  cl_mem slc_xs, slc_xd, slc_b;
  size_t global_work_size, local_work_size;

  /* Initialisierung */
  global_work_size = GLOBAL_WORK_SIZE;
  local_work_size  = LOCAL_WORK_SIZE;
  slc_b = slc->mem2;

  /* Glättungsschritte */
  for(k = 0; k < smth_cnt; k++) {
    slc_xs    = (k % 2) == 0 ? slc->mem0 : slc->mem1;
    slc_xd    = (k % 2) == 0 ? slc->mem1 : slc->mem0;
    glo_crr_i = maxi(0, i - (slc_brd-1-k));
    glo_crr_j = maxi(0, j - (slc_brd-1-k));
    slc_crr_w = mini(glo_w - glo_crr_i,
        slc_hrange + (slc_brd-1-k) + (i - glo_crr_i));
    slc_crr_h = mini(glo_h - glo_crr_j,
        slc_vrange + (slc_brd-1-k) + (j - glo_crr_j));
    slc_off   = (glo_crr_i - glo_i + 1)
        + slc_lnw * (glo_crr_j - glo_j + 1);

    clSetKernelArg(krn_smth, 0, sizeof(cl_mem), &slc_xs);
    clSetKernelArg(krn_smth, 1, sizeof(cl_mem), &slc_xd);
    clSetKernelArg(krn_smth, 2, sizeof(cl_mem), &slc_b);
    clSetKernelArg(krn_smth, 3, sizeof(int),    &slc_lnw);
    clSetKernelArg(krn_smth, 4, sizeof(int),    &slc_off);
    clSetKernelArg(krn_smth, 5, sizeof(int),    &slc_crr_w);
    clSetKernelArg(krn_smth, 6, sizeof(int),    &slc_crr_h);
    result = clEnqueueNDRangeKernel(
        slc->queue, krn_smth,
        1, NULL, &global_work_size, &local_work_size,
        0, NULL, NULL);
    CL_CHECK(result);
  }
}

/* Hilfsfunktion Glätter */
void smoothing_copyback(
    pslice_t slc, int slc_hrange, int slc_vrange,
```

```
    int i, int j, float* glo_xd, int glo_x_lnw, int glo_w,
    int glo_h, int glo_i, int glo_j,
    cl_mem slc_xd, int slc_lnw, int slc_h) {
  /* Zurückkopieren */
  read_buffer(slc->queue,
        glo_xd, 0, i+1,         j+1,
        glo_x_lnw, glo_h+2,
        slc_xd, 0, i+1 - glo_i, j+1 - glo_j,
        slc_lnw,   slc_h+2,
        mini(slc_hrange, glo_w - i),
        mini(slc_vrange, glo_h - j),
        slc->buf);
}

/* Führt Glätter aus */
void smoothing(
    cl_kernel krn_clrbrd, cl_kernel krn_smth,
    pslice_t *slcs, int slc_cnt, pgrid_t grid,
    int slc_hstep, int slc_vstep, int smth_cnt) {
  int i, j,
      glo_i, glo_j,
      glo_w, glo_h, glo_x_lnw, glo_b_lnw,
      slc_idx, slc_brd,
      slc_w, slc_h,
      slc_lnw;
  float *glo_xs, *glo_xd, *glo_b;
  cl_mem slc_xs, slc_xd, slc_b;
  pslice_t slc;

  /* Initialisierung */
  glo_w     = grid->w;
  glo_h     = grid->h;
  glo_xs    = grid->mem + grid->x_off0;
  glo_xd    = grid->mem + grid->x_off1;
  glo_x_lnw = grid->x_lnw;
  glo_b     = grid->mem + grid->b_off;
  glo_b_lnw = grid->b_lnw;
  slc_idx   = 0;
  slc_brd   = smth_cnt;

  /* Arbeit in Slices aufteilen */
  for(j = 0; j < glo_h; j += slc_vstep) {
    for(i = 0; i < glo_w; i += slc_vstep) {
      slc =  slcs[slc_idx];
      glo_i   = maxi(0, i - (slc_brd-1));
      glo_j   = maxi(0, j - (slc_brd-1));
      slc_w   = mini(glo_w - glo_i,
          slc_hstep + (slc_brd-1) + (i - glo_i));
      slc_h   = mini(glo_h - glo_j,
```

```
            slc_vstep + (slc_brd-1) + (j - glo_j));
        slc_xs  = slc->mem0;
        slc_xd  = slc->mem1;
        slc_b   = slc->mem2;
        slc_lnw = next_multiple(slc_w+2, ALIGNMENT);

        /* Daten fuer Slice kopieren */
        write_buffer(slc->queue,
            glo_xs, 0, glo_i, glo_j, glo_x_lnw, (glo_h+2),
            slc_xs, 0, 0,     0,     slc_lnw, slc_h+2,
            slc_w+2, slc_h+2,
            slc->buf);

        write_buffer(slc->queue,
            glo_b, 0,         glo_i, glo_j, glo_b_lnw, glo_h,
            slc_b, slc_lnw+1, 0,     0,     slc_lnw, slc_h+2,
            slc_w, slc_h,
            slc->buf);

        /* Rand mit 0en füllen */
        clear_slice_border(slc->queue, krn_clrbrd,
            glo_w, glo_h, glo_i, glo_j,
            slc_xd, slc_lnw, slc_lnw+1, slc_w, slc_h, 0);

        /* Glätter */
        smoothing_calc(krn_smth, slc,
            slc_hstep, slc_vstep, slc_brd, smth_cnt, i, j,
            glo_w, glo_h, glo_i, glo_j, slc_lnw);
        if((smth_cnt % 2) == 0) {
          slc_xs  = slc->mem1;
          slc_xd  = slc->mem0;
        }

        /* Daten vom Slice Zurückkopieren */
        smoothing_copyback(slc, slc_hstep,
            slc_hstep, i, j, glo_xd, glo_x_lnw, glo_w, glo_h,
            glo_i, glo_j, slc_xd, slc_lnw, slc_h);

        slc_idx = (slc_idx + 1) % slc_cnt;
      }
    }

    swap(&grid->x_off0, &grid->x_off1);
}
```

A.5. C Restriktion

```
/* Hilfsfunktion Restriktion */
void restriction_calc_and_copyback(
```

```
  cl_kernel krn_rstr, pslice_t slc,
  int depth, int slc_fst_w, int slc_fst_h,
  cl_mem slc_f_b, int slc_f_b_off, int slc_f_b_lnw,
  cl_mem slc_c_b, int glo_fst_i, int glo_fst_j,
  int glo_fst_w, int glo_fst_h, float* glo_fst_b) {
int k,
    glo_c_w, glo_c_h,
    glo_c_i, glo_c_j,
    glo_c_b_lnw,
    slc_f_w, slc_f_h,
    slc_c_w, slc_c_h,
    slc_c_b_lnw, slc_c_b_off;
float *glo_c_b;
cl_int result;
size_t global_work_size, local_work_size;

/* Optionen */
global_work_size = GLOBAL_WORK_SIZE;
local_work_size  = LOCAL_WORK_SIZE;

/* Initialisierung */
slc_f_w     = mini(slc_fst_w, glo_fst_w - glo_fst_i);
slc_f_h     = mini(slc_fst_h, glo_fst_h - glo_fst_j);

slc_c_w     = (slc_f_w - 1) / 2;
slc_c_h     = (slc_f_h - 1) / 2;
slc_c_b_lnw = next_multiple(slc_c_w, ALIGNMENT);
slc_c_b_off = 0;

glo_c_w     = glo_fst_w;
glo_c_h     = glo_fst_h;
glo_c_b_lnw = next_multiple(glo_c_w, ALIGNMENT);
glo_c_b     = glo_fst_b;

/* Restriktion */
for(k = 0; k < depth; k++) {
  if(k != 0) {
    slc_f_w     = slc_c_w;
    slc_f_h     = slc_c_h;
    slc_f_b     = slc_c_b;
    slc_f_b_lnw = slc_c_b_lnw;
    slc_f_b_off = slc_c_b_off;

    slc_c_w     = slc_f_w / 2;
    slc_c_h     = slc_f_h / 2;
    slc_c_b_lnw = next_multiple(slc_c_w, ALIGNMENT);
    slc_c_b_off = slc_f_b_off + slc_f_b_lnw * slc_f_h;
  }
```

```
    glo_c_b     += glo_c_b_lnw * glo_c_h;
    glo_c_w     /= 2;
    glo_c_h     /= 2;
    glo_c_b_lnw = next_multiple(glo_c_w, ALIGNMENT);

    clSetKernelArg(krn_rstr, 0, sizeof(cl_mem), &slc_f_b);
    clSetKernelArg(krn_rstr, 1, sizeof(cl_int), &slc_f_b_lnw);
    clSetKernelArg(krn_rstr, 2, sizeof(cl_int), &slc_f_b_off);
    clSetKernelArg(krn_rstr, 3, sizeof(cl_mem), &slc_c_b);
    clSetKernelArg(krn_rstr, 4, sizeof(cl_int), &slc_c_b_lnw);
    clSetKernelArg(krn_rstr, 5, sizeof(cl_int), &slc_c_b_off);
    clSetKernelArg(krn_rstr, 6, sizeof(cl_int), &slc_c_w);
    clSetKernelArg(krn_rstr, 7, sizeof(cl_int), &slc_c_h);

    result = clEnqueueNDRangeKernel(
      slc->queue, krn_rstr,
      1, NULL, &global_work_size, &local_work_size,
      0, NULL, NULL);
    CL_CHECK(result);
  }

  /* Zurückkopieren */
  for(k = 0; k < depth; k++) {
    glo_c_j = glo_fst_j >> (depth - k);
    glo_c_i = glo_fst_i >> (depth - k);

    read_buffer(slc->queue,
      glo_c_b, 0,           glo_c_i, glo_c_j,
      glo_c_b_lnw, glo_c_h,
      slc_c_b, slc_c_b_off, 0,       0,
      slc_c_b_lnw, slc_c_h,
      slc_c_w, slc_c_h,
      slc->buf);

    slc_c_w     = slc_f_w;
    slc_c_h     = slc_f_h;
    slc_c_b_lnw = slc_f_b_lnw;
    slc_c_b_off = slc_f_b_off;

    slc_f_w     = slc_c_w * 2 + 1;
    slc_f_h     = slc_c_h * 2 + 1;
    slc_f_b     = slc_c_b;
    slc_f_b_lnw = next_multiple(slc_f_w, ALIGNMENT);
    slc_f_b_off = slc_c_b_off - slc_f_b_lnw * slc_f_h;

    glo_c_w     = glo_c_w * 2 + 1;
    glo_c_h     = glo_c_h * 2 + 1;
    glo_c_b_lnw = next_multiple(glo_c_w, ALIGNMENT);
    glo_c_b     -= glo_c_b_lnw * glo_c_h;
```

```
  }
}

/* Führt Restriktion aus */
void restriction(
    cl_kernel krn_calcerr, cl_kernel krn_rstr,
    pslice_t *slcs, int slc_cnt,
    pgrid_t fgrid, int depth, int slc_hstep, int slc_vstep,
    int is_finest) {
  int k, i, j, glo_f_w, glo_f_h,
      glo_f_i, glo_f_j,
      glo_f_x_lnw, glo_f_b_lnw,
      slc_hrange, slc_vrange,
      slc_f_w, slc_f_h,
      slc_f_x_lnw, slc_f_x_off,
      slc_f_b_lnw, slc_f_b_off,
      slc_c_w, slc_c_h,
      slc_idx;
  float *glo_f_x, *glo_f_b;
  cl_mem slc_f_x, slc_f_b;
  pslice_t slc;
  cl_int result;
  size_t global_work_size, local_work_size;

  /* Optionen */
  global_work_size = GLOBAL_WORK_SIZE;
  local_work_size  = LOCAL_WORK_SIZE;

  /* Initialisierung */
  glo_f_w      = fgrid->w;
  glo_f_h      = fgrid->h;
  glo_f_x      = fgrid->mem + fgrid->x_off0;
  glo_f_x_lnw  = fgrid->x_lnw;
  glo_f_b      = fgrid->mem + fgrid->b_off;
  glo_f_b_lnw  = fgrid->b_lnw;
  slc_idx      = 0;

  slc_c_w = (slc_hstep >> depth);
  slc_c_h = (slc_vstep >> depth);

  slc_hrange = slc_c_w;
  slc_vrange = slc_c_h;
  for(k = 0; k < depth; k++) {
    slc_hrange = slc_hrange * 2 + 1;
    slc_vrange = slc_vrange * 2 + 1;
  }

  /* Arbeit in Slices aufteilen */
  for(j = 0; j < glo_f_h; j += slc_vstep) {
```

```
    for(i = 0; i < glo_f_w; i += slc_hstep) {
      slc = slcs[slc_idx];

      /* Initialisierung */
      glo_f_i     = i;
      glo_f_j     = j;
      slc_f_w     = mini(slc_hrange, glo_f_w - glo_f_i);
      slc_f_h     = mini(slc_hrange, glo_f_h - glo_f_j);
      slc_f_b     = slc->mem1;
      slc_f_b_lnw = next_multiple(slc_f_w, ALIGNMENT);
      slc_f_b_off = 0;

      /* Daten fuer Slice kopieren */
      write_buffer(slc->queue,
            glo_f_b, 0, glo_f_i, glo_f_j,
            glo_f_b_lnw, glo_f_h,
            slc_f_b,   0, 0,         0,
            slc_f_b_lnw,   slc_f_h,
            slc_f_w, slc_f_h,
            slc->buf);

      /* Residuum berechnen */
      if(is_finest) {
        slc_f_x     = slc->mem0;
        slc_f_x_lnw = next_multiple(slc_f_w+2, ALIGNMENT);
        slc_f_x_off = slc_f_x_lnw + 1;

        write_buffer(slc->queue,
            glo_f_x, 0, glo_f_i, glo_f_j,
            glo_f_x_lnw, glo_f_h+2,
            slc_f_x,   0, 0,         0,
            slc_f_x_lnw,   slc_f_h+2,
            slc_f_w+2, slc_f_h+2,
            slc->buf);

        clSetKernelArg(krn_calcerr, 0,
            sizeof(cl_mem), &slc_f_x);
        clSetKernelArg(krn_calcerr, 1,
            sizeof(cl_int), &slc_f_x_off);
        clSetKernelArg(krn_calcerr, 2,
            sizeof(cl_int), &slc_f_x_lnw);
        clSetKernelArg(krn_calcerr, 3,
            sizeof(cl_mem), &slc_f_b);
        clSetKernelArg(krn_calcerr, 4,
            sizeof(cl_int), &slc_f_b_off);
        clSetKernelArg(krn_calcerr, 5,
            sizeof(cl_int), &slc_f_b_lnw);
        clSetKernelArg(krn_calcerr, 6,
            sizeof(cl_int), &slc_f_w);
```

```
        clSetKernelArg(krn_calcerr, 7,
            sizeof(cl_int), &slc_f_h);
        result = clEnqueueNDRangeKernel(
          slc->queue, krn_calcerr,
          1, NULL, &global_work_size, &local_work_size,
          0, NULL, NULL);
        CL_CHECK(result);
      }

      /* Restriktion und Daten vom Slice zurückkopieren */
      restriction_calc_and_copyback(krn_rstr,
          slc, depth, slc_hrange, slc_vrange,
          slc_f_b, slc_f_b_off, slc_f_b_lnw, slc->mem0,
          glo_f_i, glo_f_j, glo_f_w, glo_f_h, glo_f_b);

      slc_idx = (slc_idx + 1) % slc_cnt;
    }
  }
}
```

A.6. C Prolongation

```
/* Hilfsfunktion Prolongation */
void prolongation_copy_and_calc(
    cl_kernel krn_clrbrd, cl_kernel krn_prol, pslice_t slc,
    pgrid_t fgrid, pgrid_t cgrid,
    int slc_hrange, int slc_vrange,
    int smth_cnt, int i, int j) {
  int glo_f_i, glo_f_j, glo_f_w, glo_f_h,
      glo_c_w, glo_c_h, glo_c_lnw,
      glo_f_i_fir0, glo_f_i_las0,
      glo_f_i_fir1, glo_f_i_las1,
      glo_f_j_fir0, glo_f_j_las0,
      glo_f_j_fir1, glo_f_j_las1,
      glo_c_i_fir0, glo_c_i_las0,
      glo_c_i_fir1, glo_c_i_las1,
      glo_c_j_fir0, glo_c_j_las0,
      glo_c_j_fir1, glo_c_j_las1,
      slc_f_w, slc_f_h,
      slc_f_off, slc_f_lnw,
      slc_c_w0, slc_c_h0,
      slc_c_w1, slc_c_h1,
      slc_c_lnw, slc_c_off,
      mask;
  size_t global_work_size, local_work_size;
  cl_int result;

  float *glo_c_x;
  cl_mem slc_f_x, slc_c_x;
```

```
/* Optionen */
global_work_size = GLOBAL_WORK_SIZE;
local_work_size  = LOCAL_WORK_SIZE;

/* Initialisierung */
glo_f_i   = maxi(0, i - (smth_cnt-1));
glo_f_j   = maxi(0, j - (smth_cnt-1));
glo_f_w   = fgrid->w;
glo_f_h   = fgrid->h;
glo_c_x   = cgrid->mem + cgrid->x_off0;
glo_c_lnw = cgrid->x_lnw;
glo_c_w   = cgrid->w;
glo_c_h   = cgrid->h;
slc_f_w   = mini(glo_f_w - glo_f_i,
    slc_hrange + (smth_cnt-1) + (i - glo_f_i));
slc_f_h   = mini(glo_f_h - glo_f_j,
    slc_vrange + (smth_cnt-1) + (j - glo_f_j));
slc_f_x   = slc->mem0;
slc_f_lnw = next_multiple(slc_f_w+2, ALIGNMENT);
slc_f_off = (i < smth_cnt ? 1 : 0)
    + slc_f_lnw * (j < smth_cnt ? 1 : 0);
slc_c_x   = slc->mem1;

glo_f_i_fir0 = maxi(glo_f_i -1, 0);
glo_f_i_las0 = mini(glo_f_i + slc_f_w, glo_f_w - 1);
glo_f_i_fir1 = glo_f_i_fir0 + (glo_f_i_fir0 % 2);
glo_f_i_las1 = glo_f_i_las0 - ((glo_f_i_las0 + 1) % 2);

glo_f_j_fir0 = maxi(glo_f_j -1, 0);
glo_f_j_las0 = mini(glo_f_j + slc_f_h, glo_f_h - 1);
glo_f_j_fir1 = glo_f_j_fir0 + (glo_f_j_fir0 % 2);
glo_f_j_las1 = glo_f_j_las0 - ((glo_f_j_las0 + 1) % 2);

glo_c_i_fir0 = (glo_f_i_fir0 + 1) / 2 - 1;
glo_c_i_las0 = glo_f_i_las0  / 2;
glo_c_i_fir1 = glo_f_i_fir1 / 2;
glo_c_i_las1 = glo_f_i_las1 / 2;

glo_c_j_fir0 = (glo_f_j_fir0 + 1) / 2 - 1;
glo_c_j_las0 = glo_f_j_las0  / 2;
glo_c_j_fir1 = glo_f_j_fir1 / 2;
glo_c_j_las1 = glo_f_j_las1 / 2;

slc_c_w0 = glo_c_i_las0 - glo_c_i_fir0 + 1;
slc_c_h0 = glo_c_j_las0 - glo_c_j_fir0 + 1;

slc_c_w1 = glo_c_i_las1 - glo_c_i_fir1 + 1;
slc_c_h1 = glo_c_j_las1 - glo_c_j_fir1 + 1;
```

```
  slc_c_lnw = next_multiple(slc_c_w0 + 2, ALIGNMENT);
  slc_c_off = (glo_c_i_fir1 - glo_c_i_fir0 + 1)
      + slc_c_lnw * (glo_c_j_fir1 - glo_c_j_fir0 + 1);
  slc_f_off += (glo_f_i_fir1 - glo_f_i_fir0)
      + slc_f_lnw * (glo_f_j_fir1 - glo_f_j_fir0);

  mask = (glo_f_j_fir0 != glo_f_j_fir1) << 0
    | (glo_f_i_las0 != glo_f_i_las1) << 1
    | (glo_f_j_las0 != glo_f_j_las1) << 2
    | (glo_f_i_fir0 != glo_f_i_fir1) << 3;

  /* Kopieren */
  write_buffer(slc->queue,
      glo_c_x,  0, glo_c_i_fir0 + 1, glo_c_j_fir0 + 1,
      glo_c_lnw, glo_c_h+2,
      slc_c_x,  0, 1, 1,
      slc_c_lnw, slc_c_h0,
      slc_c_w0, slc_c_h0,
      slc->buf);

  clear_slice_border(slc->queue, krn_clrbrd, glo_c_w, glo_c_h,
      glo_c_j_fir1, glo_c_j_fir1,
      slc_c_x, slc_c_lnw, slc_c_lnw+1, slc_c_w0, slc_c_h0, 0);

  /* Prolongation */
  clSetKernelArg(krn_prol, 0, sizeof(cl_mem), &slc_f_x);
  clSetKernelArg(krn_prol, 1, sizeof(int),    &slc_f_lnw);
  clSetKernelArg(krn_prol, 2, sizeof(int),    &slc_f_off);
  clSetKernelArg(krn_prol, 3, sizeof(cl_mem), &slc_c_x);
  clSetKernelArg(krn_prol, 4, sizeof(int),    &slc_c_lnw);
  clSetKernelArg(krn_prol, 5, sizeof(int),    &slc_c_off);
  clSetKernelArg(krn_prol, 6, sizeof(int),    &slc_c_w1);
  clSetKernelArg(krn_prol, 7, sizeof(int),    &slc_c_h1);
  clSetKernelArg(krn_prol, 8, sizeof(int),    &mask);
  result = clEnqueueNDRangeKernel(
      slc->queue, krn_prol,
      1, NULL, &global_work_size, &local_work_size,
      0, NULL, NULL);
  CL_CHECK(result);
}

/* Führt Prolongation mit anschließender */
/* Glättung aus                          */
void prolongation_smoothing(
    cl_kernel krn_clrbrd, cl_kernel krn_prol,
    cl_kernel krn_prolfst, cl_kernel krn_smth,
    pslice_t *slcs, int slc_cnt,
    pgrid_t fgrid, pgrid_t cgrid,
```

```
  int slc_hstep, int slc_vstep,
  int smth_cnt, int is_finest) {
int i, j,
    glo_i, glo_j,
    glo_w, glo_h, glo_x_lnw, glo_b_lnw,
    slc_idx, slc_brd,
    slc_w, slc_h,
    slc_lnw;
float *glo_xs, *glo_xd, *glo_b;
cl_mem slc_xs, slc_xd, slc_b;
pslice_t slc;

/* Initialisierung */
glo_w     = fgrid->w;
glo_h     = fgrid->h;
glo_xs    = fgrid->mem + fgrid->x_off0;
glo_xd    = fgrid->mem + fgrid->x_off1;
glo_x_lnw = fgrid->x_lnw;
glo_b     = fgrid->mem + fgrid->b_off;
glo_b_lnw = fgrid->b_lnw;
slc_idx   = 0;
slc_brd   = smth_cnt;

/* Arbeit in Slices aufteilen */
for(j = 0; j < glo_h; j += slc_vstep) {
  glo_j = maxi(0, j - (slc_brd-1));
  slc_h = mini(glo_h - glo_j,
      slc_vstep + (slc_brd-1) + (j - glo_j));

  for(i = 0; i < glo_w; i += slc_vstep) {
    slc =  slcs[slc_idx];

    glo_i   = maxi(0, i - (slc_brd-1));
    slc_b   = slc->mem2;
    slc_w   = mini(glo_w - glo_i,
        slc_hstep + (slc_brd-1) + (i - glo_i));
    slc_xs  = slc->mem0;
    slc_xd  = slc->mem1;
    slc_lnw = next_multiple(slc_w+2, ALIGNMENT);

    /* Daten fuer Slice kopieren */
    if(is_finest) {
      write_buffer(slc->queue,
          glo_xs, 0, glo_i, glo_j, glo_x_lnw, (glo_h+2),
          slc_xs, 0, 0,     0,     slc_lnw, slc_h+2,
          slc_w+2, slc_h+2,
          slc->buf);
    }
```

```
      write_buffer(slc->queue,
          glo_b, 0,         glo_i, glo_j, glo_b_lnw, glo_h,
          slc_b, slc_lnw+1, 0,     0,     slc_lnw, slc_h+2,
          slc_w, slc_h,
          slc->buf);

      /* Prolongation */
      prolongation_copy_and_calc(
          krn_clrbrd, is_finest ? krn_prolfst : krn_prol, slc,
          fgrid, cgrid, slc_hstep, slc_vstep, slc_brd, i, j);

      /* Rand mit 0en füllen */
      if(!is_finest) {
          clear_slice_border(slc->queue, krn_clrbrd,
              glo_w, glo_h, glo_i, glo_j, slc_xs,
              slc_lnw, slc_lnw+1, slc_w, slc_h, 0);
      }
      clear_slice_border(slc->queue, krn_clrbrd,
          glo_w, glo_h, glo_i, glo_j, slc_xd,
          slc_lnw, slc_lnw+1, slc_w, slc_h, 0);

      /* Glätter */
      smoothing_calc(krn_smth, slc,
          slc_hstep, slc_vstep, slc_brd, smth_cnt, i, j,
          glo_w, glo_h, glo_i, glo_j, slc_lnw);

      /* Daten vom Slice Zurückkopieren */
      if((smth_cnt % 2) == 0) {
        slc_xs  = slc->mem1;
        slc_xd  = slc->mem0;
      }
      smoothing_copyback(slc,
          slc_hstep, slc_hstep, i, j,
          glo_xd, glo_x_lnw, glo_w, glo_h, glo_i, glo_j,
          slc_xd, slc_lnw, slc_h);

      slc_idx = (slc_idx + 1) % slc_cnt;
    }
  }
  swap(&fgrid->x_off0, &fgrid->x_off1);
}
```

Literaturverzeichnis

[1] Börm, Steffen: *Iterative Lösungsverfahren für große lineare Gleichungssysteme*. Vorlesungsskript, März 2013. http://www.informatik.uni-kiel.de/~sb/data/GLGS.pdf.

[2] *AMD Accelerated Parallel Processing OpenCL™ Programming Guide rev2.4*, Dezember 2012. http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf.

[3] Munshi, Aaftab, Benedict Gaster und Timothy G. Mattson: *OpenCL Programming Guide*, Juli 2011.

[4] The Khronos Group: *OpenCL 1.2 Reference Pages*. http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/.