22 Synthese Systolischer Arrays

Unter Verwendung einiger Algorithmen als Beispiele für Entwurfsprobleme werden systolische Arrays auf anschauliche Weise eingeführt. Um exzessive Literatur-Recherchen weitgehend überflüssig zu machen werden die Grundlagen zu diesen Beispielalgorithmen kurz umrissen. Die eingeführten systolischen Arrays werden in ihrer Funktionsweise erklärt. Darüber hinaus werden für die Implementierung interessante Einzelheiten behandelt. Dieses Kapitel behandelt systolische Synthese anfangs aus verschiedenen Blickwinkeln: ein Überblick aus globaler Sicht unter Abgrenzung gegen Nachbargebiete wie beispielsweise parallelisierende Compiler. Aus mehr lokaler Sicht kommen die Abschnitte über systolische Synthese-Verfahren nach S. Y. Kung (Abschnitt 22.2), algebraische Verfahren [1] [2] und schließlich unter Anwendung von Retiming (Abschnitt 21.3).

22.1 Systolische Synthese aus globaler Sicht

Zur Implementierung systolisierbarer Algorithmen steht ein Spektrum von Zielplattformen zur Auswahl, angefangen vom sequentiellen programmierbaren Prozessor über programmierbare Systeme paralleler Prozessoren, spezielle ASAPs (application-specific array processors), bis zur individuell strukturierten völlig maßgeschneiderten Hardware, wie Bild 22.2 zeigt.

Der Weg vom mathematischen Problem zur Implementierung geht über mehrere Stufen, die sich trotz unterschiedlicher Implementierungs-Plattformen ähnlich sind. S. Y. Kung [2] hat diese Schritte in einer Übersicht zusammengestellt (siehe Bild 22.3), wobei sowohl eigentliche ASAPs (systolische Arrays und Wavefront-Arrays) als auch vergleichbare Implementierungen auf Mehrrechnersystemen (SIMD oder MIMD) erfaßt sind. Unter "vergleichbare Implementierungen" können hier beispielsweise Programmiertechniken oder auch Kompilationstechniken verstanden werden, die sich an die Methodik der systolischen Synthese anlehnen.

Das Entwurfsproblem (application specification) wird erst in eine rekurrente Form und damit eine prozedurale Form überführt ("rekurrent" ist nicht zu verwechseln mit "rekursiv"!). Ein

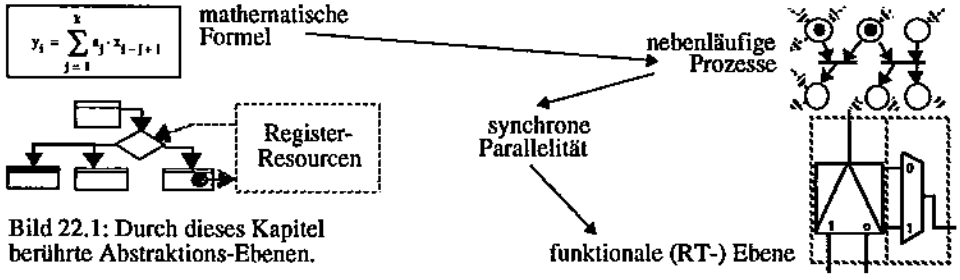


Bild 22.1: Durch dieses Kapitel berührte Abstraktions-Ebenen.

Beispiel ist ein Algorithmus zur Lösung linearer Gleichungssysteme (linear system solver). Sodann wird der Algorithmus in eine rekurrenzfreie Form überführt (single assignment form) mit dem Zweck das Überschreiben von Variablen zu eliminieren (dies wird im hier vorliegenden Kapitel später detailliert eingeführt über Beispiele). Sodann wird eine Lokalisierung durchgeführt mit dem Ziel eine weiträumige Referenzierung von Daten (wie etwa *broadcasting*) zu vermeiden. In dieser Phase wird gleichzeitig eine eventuell vorliegende Rekurrenz in eine akkumulierende Form überführt (vgl. frühere Kapitel). Auch die Lokalisierung wird später detailliert über Beispiele. Als Ergebnis-Notation oder sogar als Vehikel der Lokalisierung dient eine Darstellungsform genannt *Datenabhängigkeits-Graph oder DDG* (für *data dependency graph*). Bild 22.4 gibt eine anschauliche Gegenüberstellung der o. g. Notationen. Eine etwas formalere Einführung der erwähnten Arten von Graphen erfolgt in Abschnitt 22.2 .

Vom zeitlosen, also verzögerungsfreien DDG (Bild 22.4 b) wird ein *Signalfluß-Graph (SFG)*[1] abgeleitet (Bild 22.4 c) der Verzögerungselemente D enthält, also ein Zeitverhalten beschreibt. Verschiedene alternative SFGs können über unterschiedliche Abbildungen aus dem DDG durch Projektionsverfahren abgeleitet werden (worauf später näher eingegangen wird). Vom SFG kann nun wahlweise eine synchrone Realisierung (systolische Arrays (Bild 22.4 d) oder entsprechende Programmierung eines SIMD-Rechensystemes) oder eine asynchrone (mit Hand-shake) Realisierung (Wavefront-Array oder Programmierung eines MIMD-Rechensystemes) abgeleitet werden (s. auch Bild 22.3). Bei letzterer Realisierung wird noch eine dritte Form eines Graphen eingeführt: der *Datenfluß-Graph (DFG)*. Im Fall der synchronen Realisierung kann der SFG noch durch Retiming optimiert werden, wie beispielsweise auf optimale Ausnützung des Prozessorfeldes. Die Verzweigung des Diagrammes in Bild 22.3 zeigt die

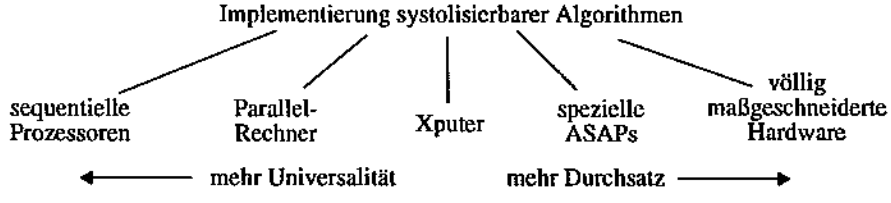


Bild 22.2: Alternativen zur Implementierung systolisierbarer Algorithmen.

1. Dieser Gebrauch des Begriffes Signalfuß-Graph weicht ab von dessen Verwendung in der Elektrotechnik

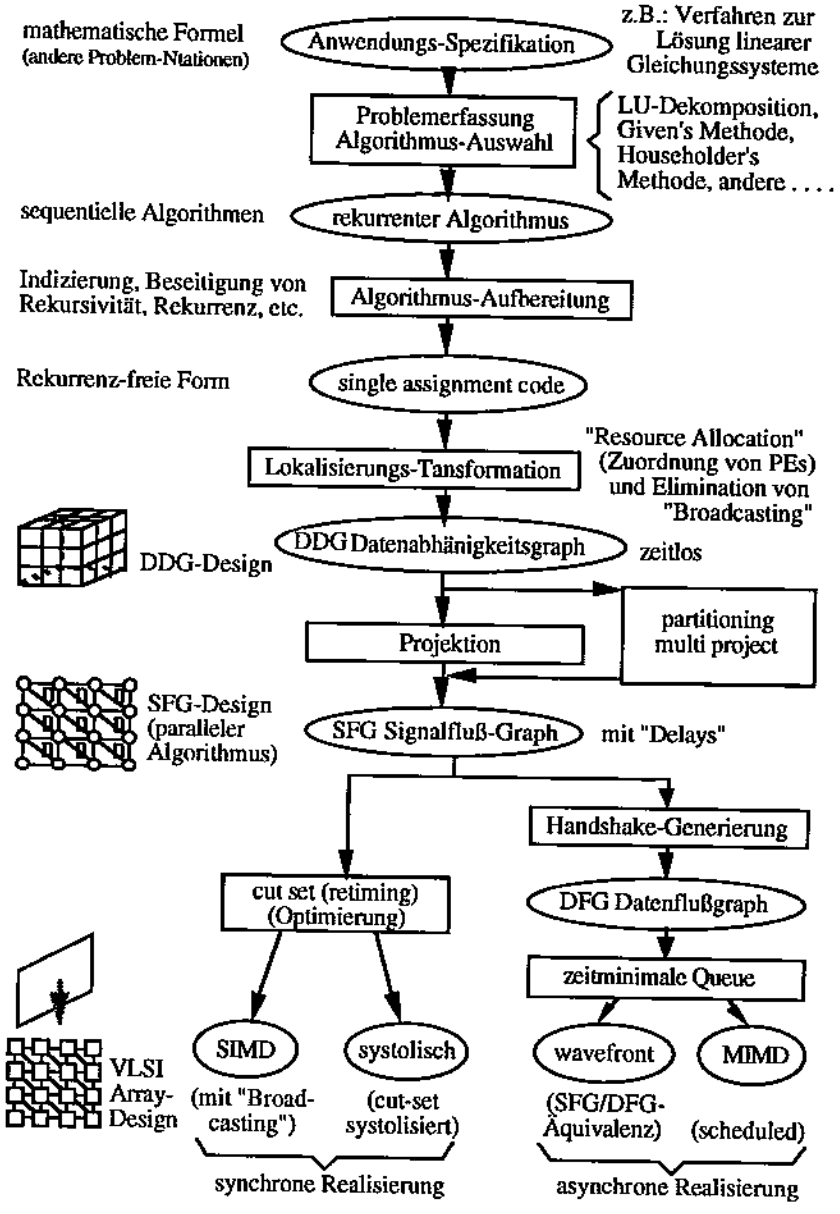


Bild 22.3. Schritte zur Implementierung paralleler Algorithmen (Übersicht)

Verwandtschaft systolischer Syntheseverfahren mit Verfahren für parallelisierende Compiler für (Steuerungs-prozedurale) Computer und erst recht für Daten-prozeduralen Xputer.

22.2 Die graphische Projektions-Methode

In diesem Abschnitt soll nun eine Methode zum Entwurf systolischer Arrays vorgestellt werden, die von S.Y. Kung ausführlich beschrieben wurde [2]. Es handelt sich dabei um eine Methode, die in groben Zügen Bild 22.3 entspricht und, die ebenfalls auf der Analyse von Datenabhängigkeiten basiert, den Datenabhängigkeits-Graph zunächst in einen sogenannten Signalfluß-Graphen (SFG) transformiert. Der SFG kann anschließend unter anderem zum Entwurf systolischer Arrays genutzt werden. Zur Begriffsbestimmung stellt Abschnitt 22.2.1 einige Definitionen vor, die in dieser Methode verwendet werden. Abschnitt 22.2.2 befaßt sich mit Methoden der Ableitung eines SFG von einem DDG, während Abschnitt 22.2.3 und 22.2.4 über Beispiele die gesamte Vorgehensweise erklären.

22.2.1 Definitionen

Datenabhängigkeits-Graph (DDG): Definition eines Graphen G= [N, A] wie in der Mathematik üblich; N ist die Menge der Knoten (nodes), A ist die Menge der Kanten (arcs, edges). Ein *(Daten-)Abhängigkeits-Graph* (*DDG*, data dependance graph) ist ein Graph, der alle in einem Algorithmus herrschenden Abhängigkeiten zwischen den an Berechnungen beteiligten Variablen repräsentiert (Beispiel in Bild 22.4 b). S. Y. Kung sieht einen DDG als graphische Repräsentation eines Single-Assignment-Algorithmus (i. e. eines Algorithmus, in dem jede Variable nur einmal auf der linken Seite einer Wertzuweisung auftritt).

Punkte im Index-Raum. Die Knoten des DDG symbolisieren die Iterations-Schritte (Berechnungen innerhalb geschachtelten Schleifen, Rekurrenz-Gleichungen). Sie liegen an Punkten im Indexraum, der durch die Wertebereiche der Iterations-Variablen aufgespannt wird. Die Operationen in jedem Schritt werden im DDG meistens nicht mit dargestellt und gehören in die Knoten. Wenn die Operationen mit repräsentiert sind, erhält man einen vollständigen DDG. Hier sind alle Operationen und sämtliche Abhängigkeiten aller Variablen enthalten. Kanten symbolisieren die Abhängigkeiten der Variablen. Wenn eine Operation im Punkt (i, j) von einer Berechnung im Punkt (i_1, j_1) abhängt, besteht eine gerichtete Kante von (i_1, j_1) nach (i, j). (Anmerkung: Ein Algorithmus ist berechenbar, wenn sein vollständiger DDG keine Schleifen oder Zyklen enthält.)

Broadcasting. Unter *Broadcasting* verstehen wir die Referenzierung von Variablen mit einem bestimmten Index an verschiedenen Punkten des Indexraumes. Die Referenzierung ist dann teilweise unabhängig vom Indexpunkt (z.B. Referenzierung nur von einem Index abhängig, bei zweidimensionalem Indexraum). Die Fern-Referenzierung des Broadcasting ist unangenehm, weshalb man deren Überführung in lokale Referenzierung anstrebt, wenn möglich.

***Homogenität* eines DDG:** Ein DDG ist homogen, wenn die bestehenden relativen Datenabhängigkeiten an jedem Punkt des Indexraumes konstant sind, d.h. unabhängig von diesem Punkt. Es können evtl. Ausnahmen toleriert werden, jedoch nur im Randbereich zur Ein-/Ausgabe. Bild 22.5 veranschaulicht die Homogenität des DDG aus Bild Bild 22.4 b.

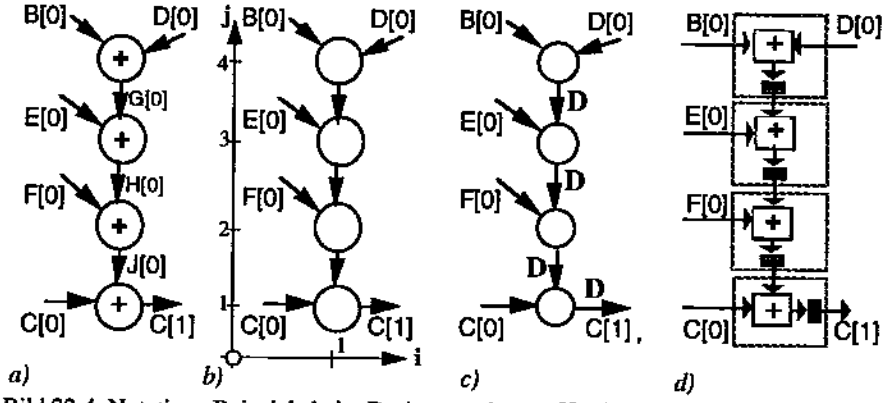


Bild 22.4: Notations-Beispiele beim Design synchroner Hardware: a) Problem-Spezifikation, b) DDG (Abhängigkeitsgraph), c) SFG (Signalflußgraph), d) Architektur von (c).

Signalfluß-Graph (SFG): eine Variante der in der Elektrotechnik gebräuchlichen gleichnamigen Notation zur Darstellung des Ablaufs von Berechnungen. Im Gegensatz zum DDG schließt dieser auch Verzögerungselemente (Delays) mit ein (Beispiel in Bild 22.4 c). Der Graph besteht also aus Prozessor-Elementen (seine Knoten: *PEs*, processing elements oder: processing nodes), Kommunikationspfaden (*CEs*, communicating edges), und Verzögerungen auf diesen Pfaden (*Ds*, delays). Ein Knoten führt die ihm zugeordneten Operationen in Nullzeit durch. Der Zeitverbrauch ist den Kommunikationspfaden zugeordnet. Eine Kante, die mit D, oder 2D gewichtet ist, hat die Bedeutung einer Verzögerung um D, oder 2·D. Die Bedeutung eines SFG kann man wie folgt zusammenfassen:

- ein SFG ist als Darstellung näher an der Hardware (als ein DDG)
- in SFG spezifiziert funktionale[2] und strukturelle[3] Hardware-Eigenschaften

Formal ist ein SFG ein Graph: G = [V, E, D(E)] wobei V die Menge der Knoten ist (vertices), E die Menge der Kanten (edges), und D(E) : E -> N_0 eine Gewichtsfunktion für Kanten[4]; welche die Zahl der Takte angibt, um die eine Kommunikation über diesen Pfad verzögert wird. Der Begriff des SFG darf nicht mit dem DFG (Datenfluß-Graph)[5] verwechselt werden, der üblicherweise dem asynchronen Fall vorbehalten ist, wie beispielsweise für Wavefront-Arrays.

Ein SFG kann Schleifen oder Zyklen enthalten, wenn jede Schleife mindestens eine Verzögerung enthält. Der SFG sollte nicht mit der Architektur verwechselt werden, die aus dem SFG abgeleitet werden kann. Bild 22.4 zeigt die Gegenüberstellung des SFG (Bild c) und der daraus abgeleiteten Architektur (Bild d), welche zusätzlich z. B. noch die Partitionierung angibt.

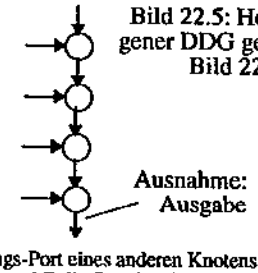


Bild 22.5: Homogener DDG gemäß Bild 22.4 b.

[2] PEs (processing elements)
[3] Interkonnekt
[4] eine gerichtete Kante läuft vom Ausgangs-Port eines Knotens zum Eingangs-Port eines anderen Knotens
[5] DFG: ein Graph G = (V, E, D(E), Q(E)), wobei D die Zahl der Datentokens und Q die Queuing-Kapazität angibt

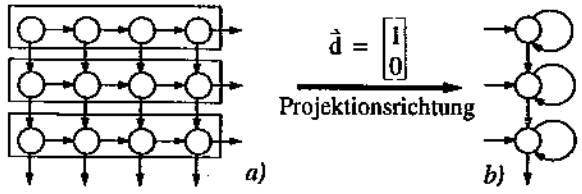
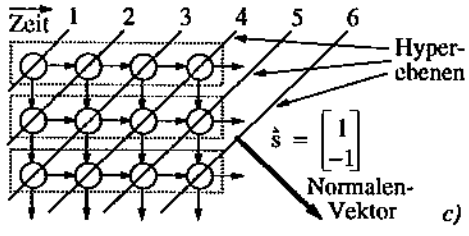
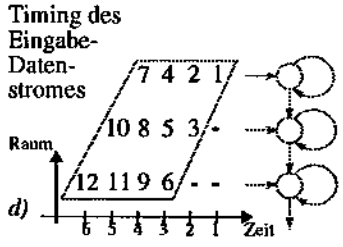


Bild 22.6: Veranschaulichung der DDG-zu-SFG-Abbildung: b) SFG aus (a), d) Datenstrom (schedule) aus (c).





22.2.2 Abbildung eines DDG in SFGs

Aus dem zeitlosen DDG können ein oder mehrere SFGs abgeleitet werden. Mit dem SFG wird Zeit (Verzögerungsglieder) und physische Lokalität (der PEs) eingeführt. Die Abbildung eines Datenabhängigkeitsgraphen in einen Signalflußgraphen erfolgt in zwei Schritten:

1.) **Prozessor-Zuordnung (allocation).** Zuordnung von Operationen zu Prozessor-Elementen durch lineare Projektion (vgl. M_S bei SYS^3, s. Kapitel 7 in [1]). Optimierungskriterium: z.B. Zahl der Prozessoren, Minimierung der Kommunikation

2.) **Scheduling (Daten-Fahrplan erstellen: erforderlichen Datenstrom ermitteln).** Festlegung der Ausführungsreihenfolge der Operationen (vgl. M_t bei SYS^3, s. Kapitel 8 in [1]) Optimierungskriterium: z.B. Minimierung der Berechnungszeit, Latenzzeit

Prozessor-Zuordnung: lineare Projektion des DDG aus dem Indexraum in einen Prozessor-Raum mit niedriger Dimension (repräsentiert durch einen Projektions-Vektor $\vec{d}$). Knoten eines DDG (Bild 22.6 a) werden entlang einer "geraden" Linie einem gemeinsamen PE zugeordnet Bild 22.6 b). Eine Motivation besteht in möglichst guter Auslastung der PEs dadurch, daß die Zahl der Knoten des SFG möglichst kleiner ist als die Zahl der Knoten im DDG.

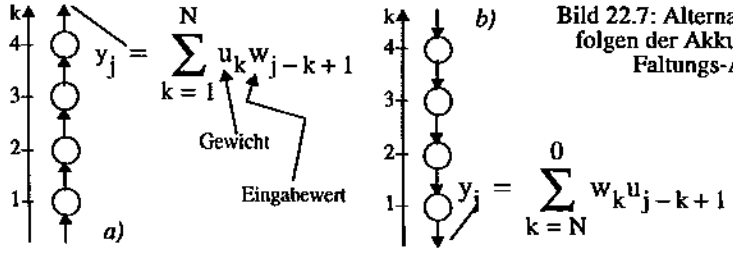


Bild 22.7: Alternative Reihenfolgen der Akkumulation im Faltungs-Algorithmus.

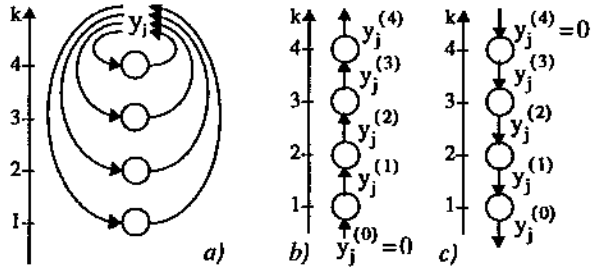


Bild 22.8: Realisierung der Rekurrenz aus Bild 22.7: a) als "Overwriting", b) im Single-Assignment-Verfahren mit steigendem Index k, c) mit fallendem k.

Festlegung der Ausführungsreihenfolge (Scheduling): Einführung von Hyperebenen, die Iterations-Knoten enthalten, die zu einem bestimmten Zeitpunkt ausgeführt werden (engl.: *hyper planes*). Repräsentation der Hyperplanes durch einen Normalen-Vektor ▒, der zu den Ebenen senkrecht steht (vgl. Beispiel in Bild 22.6 c).

Der Ausführungszeitpunkt für einen Indexknoten ergibt sich aus der Multiplikation mit dem Normalen-Vektor. Nicht alle Richtungen der Hyper-Ebenen sind zulässig. Sie müssen zwei Bedingungen erfüllen (alle Knoten in der gleichen Hyper-Ebene müssen zum gleichen Zeitpunkt "aufgerufen" (executed) werden):

1.) **Zuordnungskonflikte vermeiden.** Die Ebenen dürfen nicht parallel zu dem Projektionsvektor ▒ verlaufen. Dies garantiert, daß Berechnungen, die auf das gleiche Prozessorelement abgebildet werden, dort zu unterschiedlichen Zeitpunkten durchgeführt werden.

2.) **Kausalität einhalten.** Alle Abhängigkeiten müssen die Ebenen in der selben Richtung schneiden. Dies stellt sicher, daß die Datenabhängigkeiten nicht gleichzeitig mit und entgegen der Richtung der Zeitachse verlaufen. Ein benötigtes Ergebnis ist also immer schon zu einem früheren Zeitpunkt generiert worden.

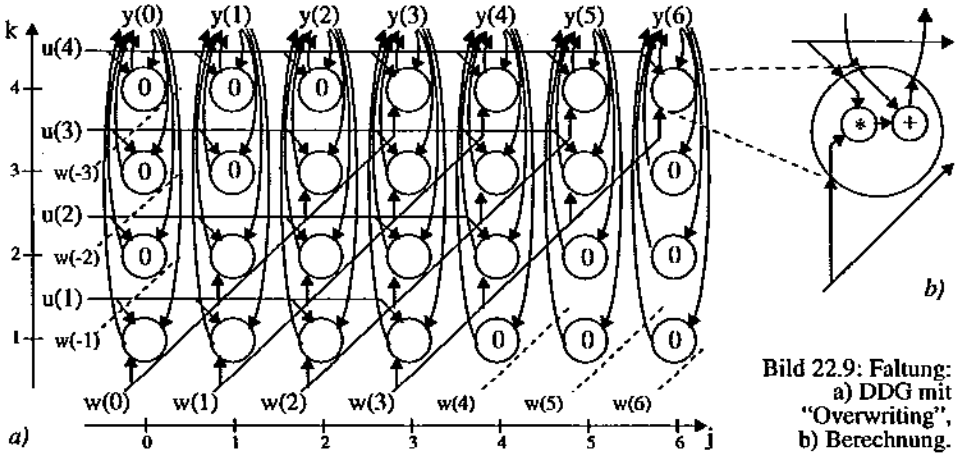


Bild 22.9: Faltung: a) DDG mit "Overwriting", b) Berechnung.

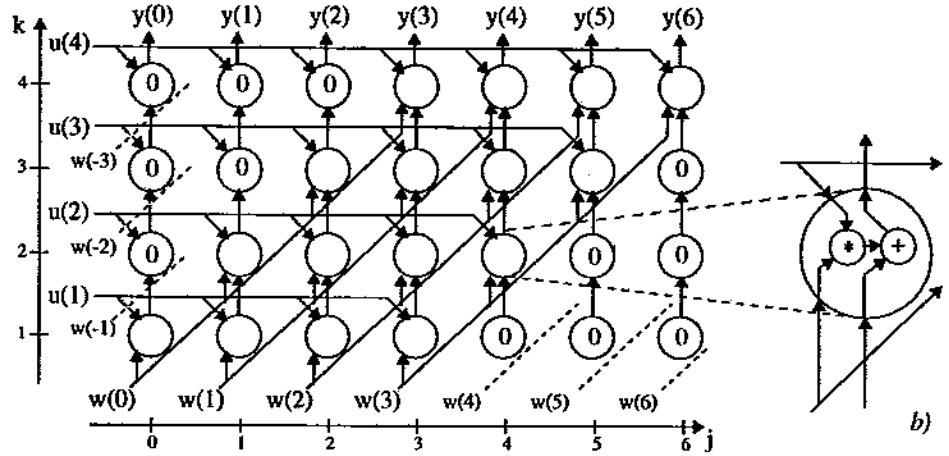


Bild 22.10: Globaler Datenabhängigkeitsgraph für die Faltung (22.3), (22.4)

Dies bedeutet, daß eine partielle Ordnung zwischen den Berechnungen besteht. Formell müssen die Bedingungen $\hat{s}^T\vec{d} > 0$ und $\hat{s}^T\hat{e} \geq 0$ erfüllt werden, wobei $\hat{s}$ der Normalen-Vektor der Hyperebenen ist, $\vec{d}$ der Projektionsvektor und $\hat{e}$ eine Kante des Datenabhängigkeitsgraphen. Bild 22.6 d zeigt als Beispiel den aus Bild 22.6 c sich ergebenden Datenstrom.

22.2.3 Abbildung eines Algorithmus in ein VLSI-Array

In diesem Abschnitt soll die gesamte Synthese-Prozedur zusammenhängend behandelt werden. Hier soll die Abbildung eines Algorithmus in ein VLSI-Array anhand der folgenden sechs

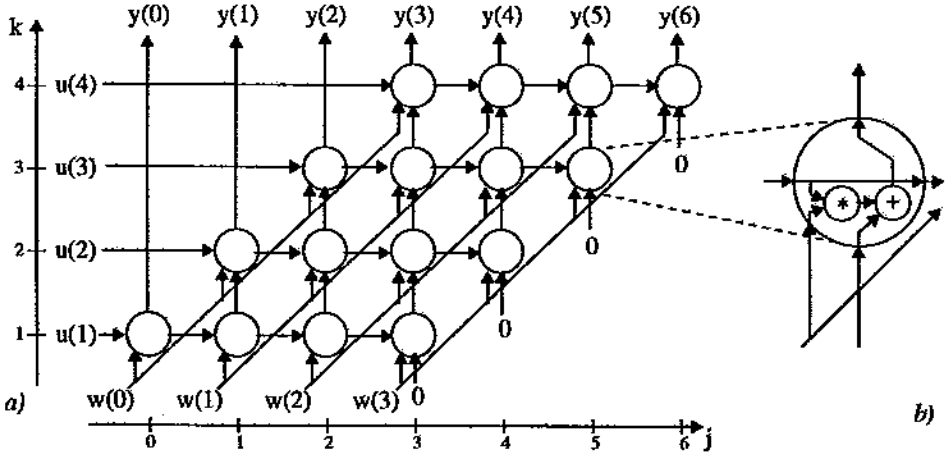


Bild 22.11: Ergebnis der Elimination des u-Broadcasting in Bild 22.10

Schritte ausgeführt werden [2] (siehe auch Bild 22.3). Die einzelnen Schritte werden anschließend an Beispielen erläutert.

1.) Spezifikation des Problems

2.) Auswahl eines geeigneten Algorithmus

3.) Analyse der Datenabhängigkeiten des Algorithmus und Erstellen des Datenabhängigkeits-Graphs (DDG)

4.) **Lokalisierung.** Transformation des Datenabhängigkeits-Graphen bzw. des Algorithmus zur Erzielung lokaler Datenabhängigkeiten

5.) Abbildung des DDG in einen Signalflußgraphen (SFG)

6.) Abbildung des SFG in ein Array b zw. in eine Architektur

22.2.4 Das Entwurfsverfahren am Beispiel der Faltung

Als erstes Beispiel soll hier die bereits vorgestellte Faltung dienen, an welchem die Abbildung vom DDG in einen systolischen Array konkret nachvollzogen werden soll.

1. Schritt: Zunächst folgende Definitionen, vgl. (21.1). Eingabe-Sequenz: $\{u_m\}$ mit m = 1, 2,... N; Gewichte: $\{w_k\}$, wobei k = 1, ... N, hier: N=4; Ausgabe-Sequenz: $\{y_j\}$, wobei j = 0,1,.. 2N-2. Eine Spezifikation des Problems ergibt sich mit:

$$y_j = \sum_{k=1}^{N} w_k u_{j-k+1} \tag{22.1}$$

Zur Veranschaulichung gehen wir von Fall zu Fall von einem konkreten Beispiel mit N = 4 aus. Für N = 4 müssen für jedes y_j die folgenden Produkte akkumuliert werden:

$$\begin{aligned} &u_1 \cdot w_4 \\ &u_2 \cdot w_3 \\ &u_3 \cdot w_2 \\ &u_4 \cdot w_1 \end{aligned} \tag{22.2}$$

Die nach Formel (22.1) angegebene Reihenfolge der Akkumulation ist nicht zwingend.Es beispielsweise auch die umgekehrte Reihenfolge möglich (vgl. Gegenüberstellung in Bild 22.7). Somit ist auch die folgende Form der Spezifikation anwendbar, für die wir uns entscheiden:

$$y_j = \sum_{k=1}^{N} u_k w_{j-k+1} \tag{22.3}$$

2. Schritt: Wahl eines geeigneten Algorithmus. Wir leiten aus (22.3) folgendes Programm ab:

```
(1)   for j := 0 to 2*N-2 do
(2)   begin
(3)      y(j) := 0;
(4)      for k := 1 to N do
(5)           y(j) := y(j) + u(k) * w(j-k+1);
(6)      end k;
(7)   end j;
```

(22.4)

Gewicht

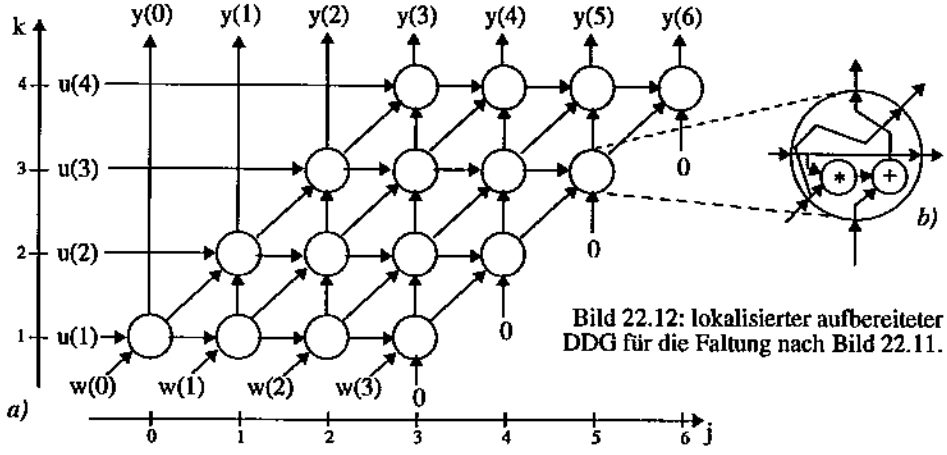


Bild 22.12: lokalisierter aufbereiteter DDG für die Faltung nach Bild 22.11.

Zeile 5 in Formel (22.4) zeigt Overwriting von y(j), was durch Bild 22.8 a veranschaulicht wird. Overwriting kann als eine Abart des Broadcasting angesehen werden (vgl. Bild 22.8 a). Dieses Overwriting kann durch Transformation in einen Single-Assignment-Kode eliminiert werden durch Einführung eines zusätzlichen Index k für die Variable y. (Dadurch kann der DDG in Bild 22.9 a vermieden werden.) Hierdurch ändert sich Zeile 5: es entsteht folgende Formel (22.5). Für jedes neue Ergebnis der Rekurrenz wird eine neue Variable eingerichtet gemäß Bild 22.8 b oder c.

```
(1)   for j := 0 to 2*N-2 do
(2)   begin
(3)      y(j, 0) := 0;
(4)      for k := 1 to N do                                    (22.5)
(5)         y(j, k+1) := y(j, k) + u(k) * w(j-k+1);
(6)      end k;
(7)   end j;
```

3. Schritt: Erstellen des DDG. Bild 22.10 a zeigt den DDG für obigen Kode. Bild 22.10 b zeigt die pro Knoten duchzuführenden Berechnungen. Wie man sieht werden die Elemente von w und u durch ein Broadcast in die Iterations-Knoten gebracht. Die Knoten, in denen die '0' steht, tragen nicht zum Ergebnis bei (Hinzuaddieren von 0) und werden deshalb im Folgenden vernachlässigt. Dies bedeutet eine Transformation des Algorithmus nach folgender Gleichung:

$$y_j = \sum_{k = \mathrm{Max}\,(0,\, j-N+1)}^{\mathrm{Min}\,(j,\, N-1)} u_k w_{j-k} \qquad (22.6)$$

Die Elimination des Broadcasting erfolgt durch Einführen zusätzlicher Indizierung für die Variablen u und w und zusätzlicher Anweisungen zur Propagierung der Werte. Zunächst eliminieren wir das Broadcasting der Variablen u. Resultierender Programm-Kode ist (22.7):

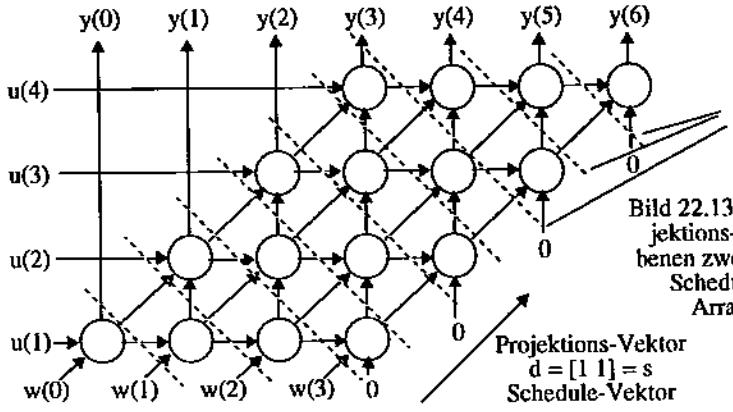


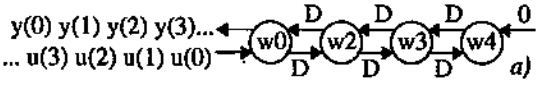
Bild 22.13: Ein möglicher Projektions-Vektor und Hyperebenen zwecks Ableitung eines Schedule für den Faltungs-Array gemäß Bild 22.12.

```
(1)   for j := 0 to 2*N-2 do begin
(2)       y(j, 0) := 0;
(3)       (* wir ersetzen: u(j, 0) := u_k; *)
(4)       for k := 1 to N do begin
(5)           u(j+1, k) := u(j, k);
(6)           y(j, k+1) := y(j, k) + u(j, k) * w(j-k+1);
(7)       end k;
(8)   end j;
```
(22.7)

Man beachte, daß die Variablen u nunmehr doppelt indiziert ist (beachte Zeile 6 in Gleichung (22.7)). Die zusätzlich eingefügte Anweisung in Zeile 5 dient der horizontalen Propagierung des Operanden u von links nach rechts durch das PE hindurch (vgl. Bild 22.11 b). Das Ergebnis in Form eines DDG ist in Bild 22.11 zu sehen. Nun eliminieren wir auch noch den Broadcast der Variablen w. Wir erhalten nun die folgende Programm-Notation (22.8):

```
(1)   for j := 0 to 2*N-2 do begin
(2)       y(j, 0) := 0;
(3)       (* wir ersetzen: w(j, 0) := w_{j-k} *)
(4)       for k := 1 to N do begin
(5)           u(j+1, k) := u(j, k);
(6)           w(j+1, k+1) := w(j, k)
(7)           y(j, k+1) := y(j, k) + u(j, k) * w(j, k+1);
(8)       end k;
(9)   end j;
```
(22.8)

Man beachte, daß nun auch die Variable w mit w(j,k) doppelt indiziert ist (beachte Zeile 6 in Gl. (22.7)) im Vergleich zu Zeile 7 in obiger Gl. (22.8). Bild 22.12 zeigt den somit ermittelten zugehörigen DDG. Die als Zeile 6 eingefügte Anweisung dient der Propagierung von w diagonal durch den PE von links unten nach rechts oben (siehe Bild 22.12 b). Die somit ermittelte Konstellation führt zum Array nach Bild 22.14 a. Ein Prozessor-Element hat die Darstellung nach Bild 22.14 b. Bild 22.13 veranschaulicht das "Scheduling" für das vorliegende Beispiel.



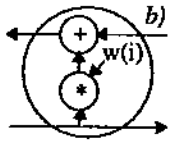


Bild 22.14: systolisches Array zur in Bild 22.13 gezeigten Projektion und Schedule, b) Prozessorelement hierzu.

22.2.5 Synthese-Beispiel Matrix-Vektor-Multiplikation

Als ein weiteres Beispiel zur Veranschaulichung der Verfahrensweisen soll die Matrix-Vektor-Multiplikation dienen [2].

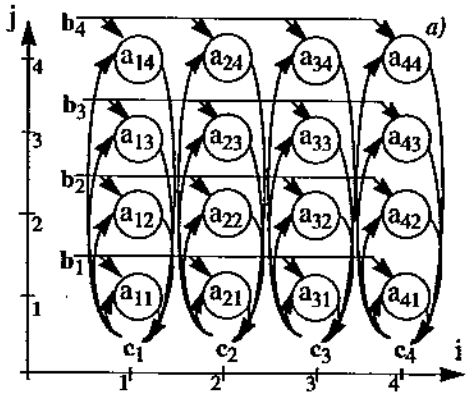
1.) Problem-Spezifikation: die Multiplikation einer Matrix A mit einem Vektor **b** gemäß dem Gleichungssystem **c = A*b,** (Beispiel mit der Größe m = 4) oder

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \tag{22.9}$$

wobei für das i-te Element von C gilt:

$$c_i = \sum_{j=1}^{m} a_{ij} b_j \tag{22.10}$$

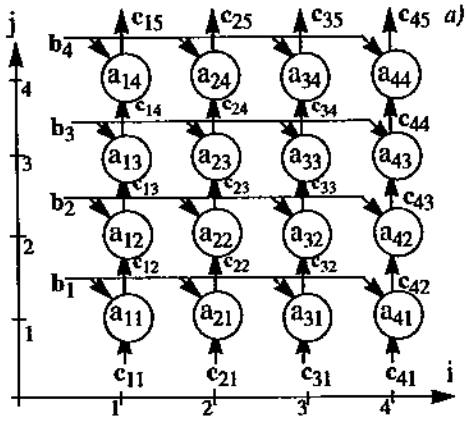
Diese Matrix-Vektor-Multiplikation repräsentiert bekanntlich folgendes Gleichungssystem:




```
for i:=1 to 4 do
begin
    c(i) := 0;
    for j:=1 to 4 do
    c(i) := c(i) + a(i, j)* b(j);
end;
```

b)

Bild 22.15: Vektor-Matrix-Multiplikation - Spezifikation: a) SFG, b) Pascal-Notation




```
for i:=1 to 4 do                                   b)
begin
    c(i,1) := 0;
    for j:=1 to 4 do
        c(i,j+1) := c(i,j) + a(i, j)* b(j);
    end j;
end i;
```

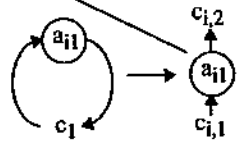
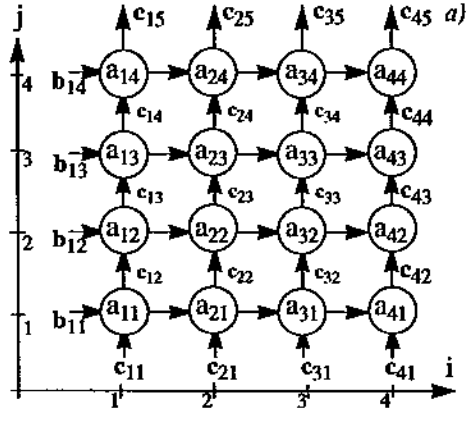


Bild 22.16: Vektor-Matrix-Multiplikation - Single-Assignment-Form, gewonnen aus Bild 22.15: a) SFG, b) Pascal-Notation.

$$
\begin{aligned}
c_1 &= a_{11}b_1 + a_{12}b_2 + a_{13}b_3 + a_{14}b_4 \\
c_2 &= a_{21}b_1 + a_{22}b_2 + a_{23}b_3 + a_{23}b_4 \\
c_3 &= a_{31}b_1 + a_{32}b_2 + a_{33}b_3 + a_{34}b_4 \\
c_4 &= a_{41}b_1 + a_{42}b_2 + a_{43}b_3 + a_{44}b_4
\end{aligned}
\qquad (22.11)
$$

Bild 22.15 b zeigt das entsprechende Pascal-ähnliche Programm. Bild a zeigt den dazu äquivalenten DDG. Man sieht deutlich, daß die Variablen c_i und dann immer wieder überschrieben werden. Andererseits werden die Variablen b_j über ein Broadcast-Schema von links nach rechts ausgebreitet. Die Konstanten a_{ij} sind den Knoten zugeordnet.




```
for i:=1 to 4 do                                   b)
    begin
    c(i,1) := 0;
    for j:=1 to 4 do
        begin
        b(i+1 ,j) := b(i, j);
        c(i,j+1) := c(i,j) + a(i, j)* b(i,j);
     end j;
end i;
```

Bild 22.17: Vektor-Matrix-Multiplikation - nach Elimination von Broadcasting, abgeleitet aus Bild 22.16: a) DDG, b) Pascal-Notation.

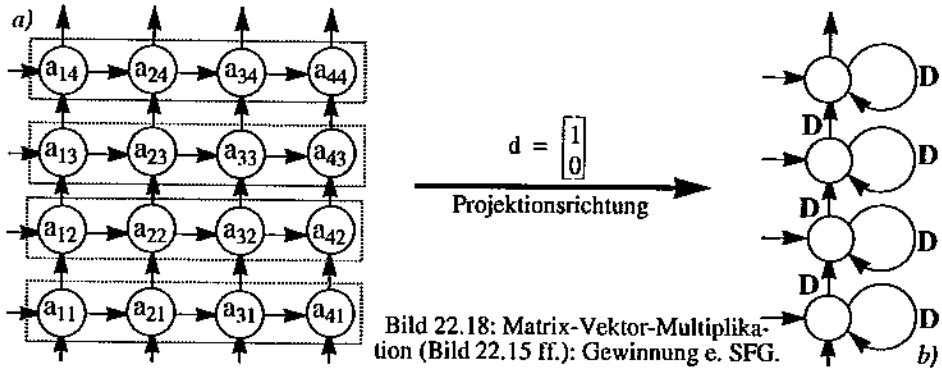


Bild 22.18: Matrix-Vektor-Multiplikation (Bild 22.15 ff.): Gewinnung e. SFG.

Wir führen nun eine Transformation in einen Single-Assignment-Kode zwecks Vermeidung von Overwriting des c(i) ein. Was heißt "Overwriting"? Beispielsweise die linke Seite und die rechte Seite der Wertzuweisung **c(i) := c(i) + A(i, j)* b(j);** in Bild 22.15 b benutzen für **c** den gleichem Index mit gleichem Wert)[6]. Dieses Überschreiben wird in Bild 22.15 a durch die vielen Rückkopplungs-Schleifen von und nach c_i veranschaulicht. Für jedes **i** war bisher nur eine Variable **c(i)** verfügbar. Um das Überschreiben vermeiden zu können, führen wir nun für **c** eine doppelte Indizierung ein mit **c(i,j)**. Zuerst wird **c(i,1)** nullgesetzt. Im neuen Programm ist

c(i) := 0; zu: **c(i,1) := 0;** (22.12)

geworden. Der Kern der inneren Schleife ist von

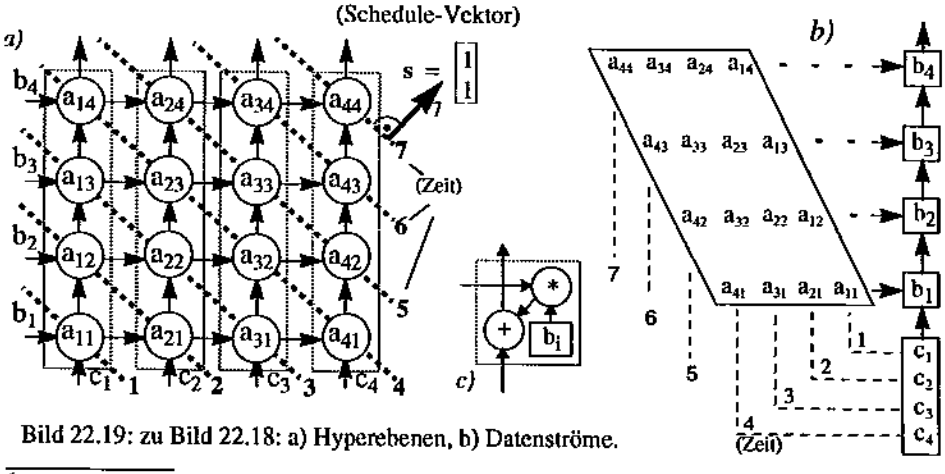


Bild 22.19: zu Bild 22.18: a) Hyperebenen, b) Datenströme.

[6] die Schreibweise wie c(i) ist die maschinell lesbare Form der Notation wie c_i mit tiefergestelltem Index

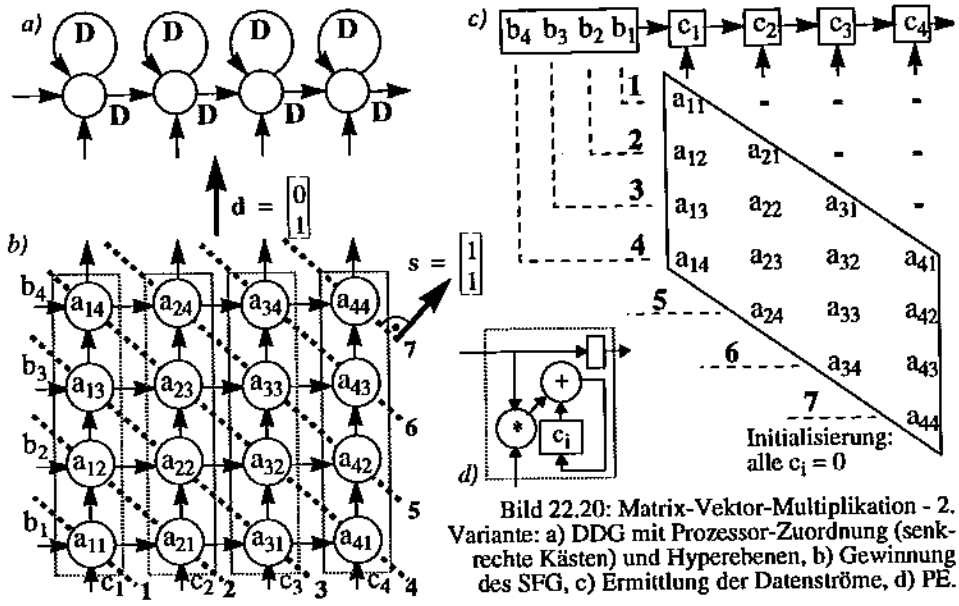


Bild 22.20: Matrix-Vektor-Multiplikation - 2. Variante: a) DDG mit Prozessor-Zuordnung (senkrechte Kästen) und Hyperebenen, b) Gewinnung des SFG, c) Ermittlung der Datenströme, d) PE.

$$c(i) := c(i) + a(i,j) * b(j); \tag{22.13}$$

zu

$$c(i, j+1) := c(i, j) + c(i, j)* b(j); \tag{22.14}$$

geworden.Das Ergebnis des ersten Durchlaufes der inneren Schleife (j=1) wird der neu eingeführten Variablen **c(i,2)** zugewiesen, das Ergebnis des 2. Schleifendurchlaufes (mit j=2) der Variablen **c(i,3)**, und so weiter, bis nach dem letzten Durchlauf der inneren Schleife das Ergebnis in **c(i,5)** abgelegt worden ist (Bild 22.15 a).

Nun führen wir eine Transformation zur Elimination des Broadcasting durch: Bisher war die Eingangsvariable **b(j)** gleichzeitig mehreren Operatoren mit **a(i,j)** zugeführt worden. Deshalb führen wir auch für **b** einen zweiten Index ein, nämlich **i**, sodaß wir **b(i,j)** erhalten.Broadcasting läßt sich nun vermeiden, indem wir diese Eingangswerte nun von mehreren Variablen aus den Operatoren zuführen. Wir führen eine zusätzliche Wertzuweisung in die innere Schleife ein mit

$$b(i+1, j) := b(i, j); \tag{22.15}$$

wodurch

$$c(i,j+1) := c(i,j) + a(i, j)* b(j); \tag{22.16}$$

zu

$$c(i, j+1) := c(i, j) + a(i, j)*b(i, j); \tag{22.17}$$

wird. Bild 22.18 zeigt das Ergebnis unserer Transformation: Bild a) den DDG, und in Bild b) die Pascal-Notation dazu.

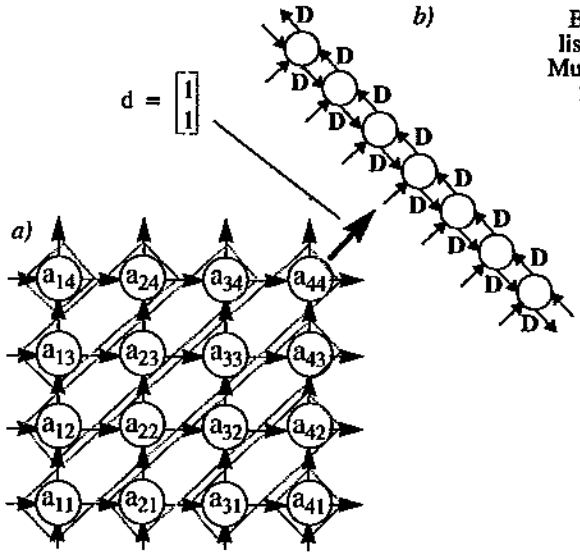


Bild 22.21: Variante Nr. 3 des systolischen Arrays für die Matrix-Vektor-Multiplikation: a) DDG mit Prozessor-Zuordnung, b) Ermittlung des SFG.

Bild 22.18 veranschaulicht die Abbildung des DDG (Bild 22.18 a) in einen Signalflußgraphen (Bild 22.18 b). Die Projektionsrichtung ist gegeben durch:

$$\mathbf{d} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{22.18}$$

Bild 22.19 veranschaulicht die Festlegung des Schedules für unser Beispiel. Jede solche Hyper-Ebene berührt eine Menge von Iterationen, die zur selben Zeit ausgeführt werden müssen. Als Schedule-Vektor (Normalen-Vektor zu den Hyper-Ebenen (Bild 22.19 a) ist gegeben mit:

$$\mathbf{d} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{22.19}$$

Bild 22.19 b veranschaulicht die durch diese Projektion ermittelten Datenströme. Die Koeffizienten von **b** werden in den PEs gespeichert.

2. Variante. Wenn wir eine andere Projektions-Richtung wählen, erhalten wir für die Matrix-Vektor-Multiplikation ein anderes systolisches Array. Wir verwenden zur Gewinnung des SFG diesmal den Projektions-Vektor

$$\mathbf{d} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} , \tag{22.20}$$

wie in Bild 22.18 b zu sehen ist. Wir erhalten einen SFG, der demjenigen unserer 1. Variante (vgl. Bild 22.18 b) zu entsprechenscheint. und auch die gleiche Länge hat (n, wenn ein (n, n)-Matrix-Format und ein Vektor der Länge n vorliegt). Der Unterschied wird erst sichtbar, wenn die Datenströme ermittelt worden sind (Bild 22.18 c). Obwohl der Normalen-Vektor s sich nicht von dem der ersten Variante unterscheidet, erhalten wir eine andere Datenstrom-Anord-

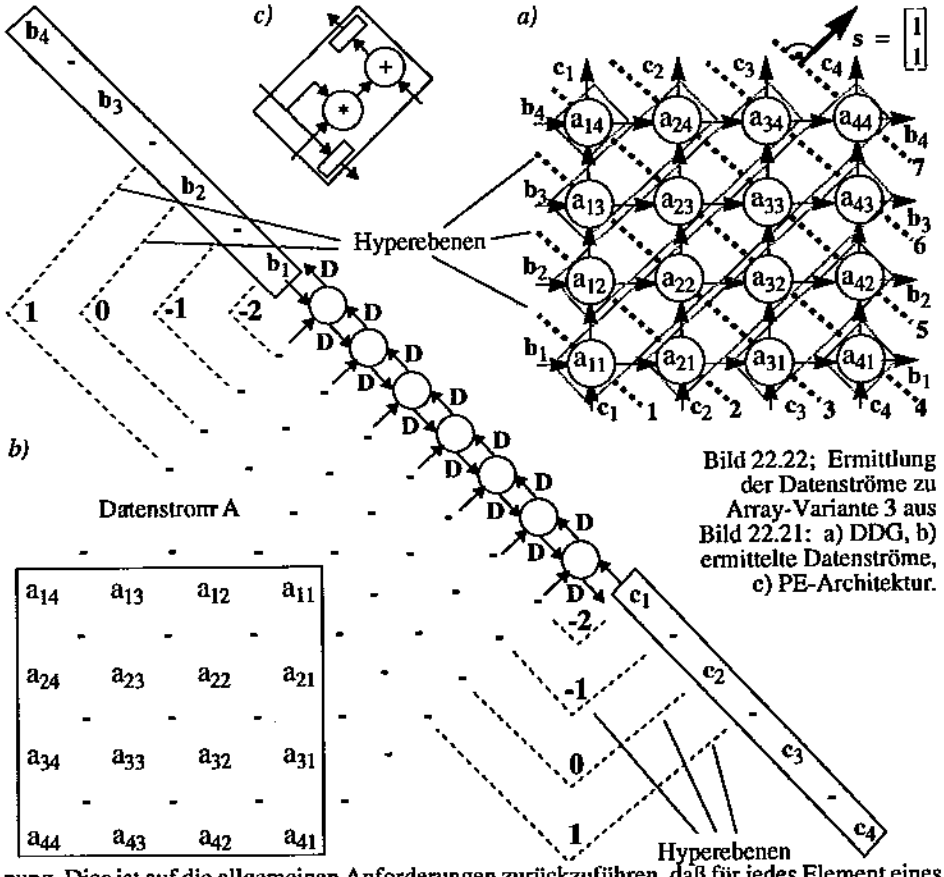


Bild 22.22; Ermittlung der Datenströme zu Array-Variante 3 aus Bild 22.21: a) DDG, b) ermittelte Datenströme, c) PE-Architektur.

nung. Dies ist auf die allgemeinen Anforderungen zurückzuführen, daß für jedes Element eines Datenstromes gleichzeitig angegeben werden muß:

- wann (zu welchem Takt) wird das Element angewandt (für eine Berechnung) ?
- wo (in welchem PE) wird das Element angewandt (für eine Berechnung) ?

Es ergibt sich hieraus auch für die 2. Variante, daß jedes c_i eindeutig einem bestimmten PE_i zugeordnet ist (vgl. die senkrechten Kästen in Bild 22.18 c). Die Elemente von **c** können demnach in den PEs gespeichert werden, wohingegen die Elemente von **b** durch geschiftet werden müssen (vgl. Bild 22.18 c).

3. Variante. Eine dritte Array-Variante für die Matrix-Vektor-Multiplikation erhalten wir durch einen um 45° nach rechts oben zeigenden Projektionsvektor d (siehe Bild 22.21). Wir erhalten den SFG eines Array der Länge 2n + 1, also fast doppelt so lang wie die 1. und die 2. Variante. Darüber hinaus ist er bidirektional (im Gegensatz zur 1. und die 2. Variante, die nur unidirektional sind). Bild 22.22 a zeigt die Ermittlung der Datenströme über einen Normalen-

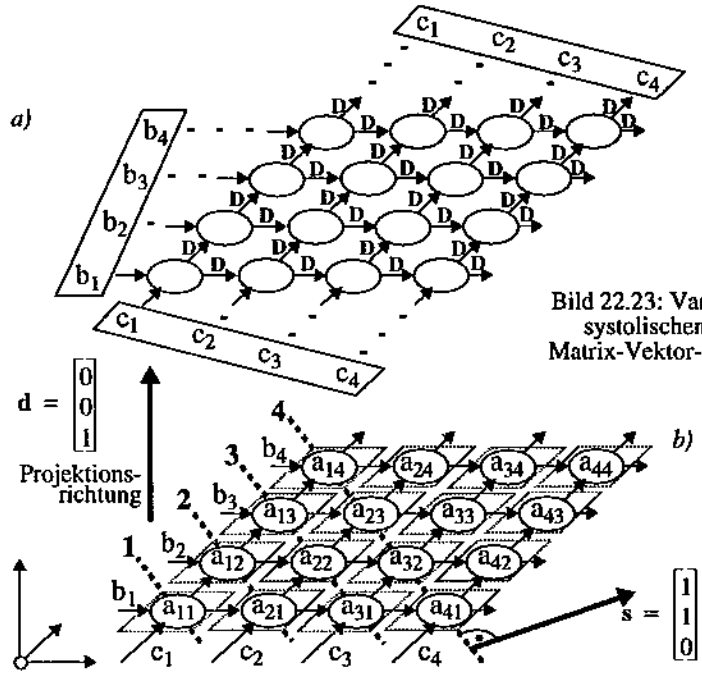


Bild 22.23: Variante Nr. 4 des systolischen Arrays für die Matrix-Vektor-Multiplikation.

Vektor s, der ebenfalls um 45° nach rechts oben zeigt. Bild 22.22 b zeigt das Ergebnis: Form und Anordnung der Datenströme. Man sieht, daß der Array 2-slow ist (vgl. auch die Schnappschüsse der kleineren Variante dieses Array aus Bild 21.11). Man beachte auch die um 3 Takte verzögerte Eingabe von Datenstrom A. Deren Notwendigkeit kann man z.B. an der Berechnung von $a_{11} \cdot b_1$ erkennen: b_1 benötigt 4 Takte, um in das mittlere PE zu gelangen, um dort a_{11} anzutreffen. Um diese Zeit muß a_{11} verzögert werden (vgl. die "dashes" vor A in Bild 22.22 b.

4. Variante. Durch Einführung einer dritten Dimension wird ein Projektions-Vektor d möglich, der auf dem DDG senkrecht steht (Bild 22.23 a). Wir erhalten SFG n. Bild 22.23 b, wo bereits die beiden Eingabe-Datenströme angedeutet sind. Die Elemente von A können in den PEs gespeichert werden (Bild 21.10 f zeigt eine kleinere Version des so ermittelten Arrays).

Effizienz der 4. Variante. Die Zahl der PE beträgt n2, was einen hohen Durchsatz gestattet (zur lückenlosen Folge vieler Multiplikationen ausnützbar), der andererseits durch eine höhere Eingabe-Datenrate erkauft werden muß. Im Gegensatz zur 1. und 2. Variante, die wegen sequentieller Eingabe nur eines einzigen Datenstromes nur einen einzigen Eingabekanal benötigen, erfordert die 4. Variante gleich 2n Eingabekanäle, da zwei Datenströme (b und c) parallel eingegeben werden müssen.

22.3 Literatur

[1] R. Hartenstein: Einführung in den VLSI-Entwurf, Band 1; IT Press Verlag, v. f. 1995
[2] S. Y. Kung: VLSI Array Processors; Prentice-Hall, 1988