Studienarbeit

Multiplikation dünn besetzter Matrizen mit dem befehlssystolischen Prozessorfeld

vorgelegt von

Stefan Milius

Institut für Datenverarbeitungsanlagen
der Technischen Universität zu Braunschweig
Prof. Dr. Manfred Schimmler

Betreuer: Jens Janzen

Braunschweig, den 21. September 1998

Eidestattliche Versicherung

Hiermit versichere ich an Eides statt, daß ich keine anderen als die angegebenen Hilfsmittel zur Erstellung dieser Arbeit verwendet habe.

Stefan Milius

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Das *befehlssystolische Prozessorfeld*[1] ist ein Konzept, das es ermöglicht, Parallelität von Berechnungen durch entsprechende Hardware umzusetzen. Es gewährleistet Eigenschaften wie Lokalität der Kommunikation zwischen den Prozessoren und Regularität des Layouts der Schaltung. Deshalb ist es hervorragend geeignet, um zum Beispiel mit der VLSI-Technologie realisiert zu werden, die einen sehr hohen Grad der Integration erlaubt, was letzlich einen hohen Grad an Parallelität bedeutet.

Das ISA verbindet gleichzeitig mit den eben genannten Eigenschaften die Möglichkeit freier Programmierung. Dies ist ein entscheidender Vorteil gegenüber den verwandten systolischen Prozessorfeldern, die jeweils für eine spezielle Anwendung benutzbar sind und gewissermaßen in Hardware gegossene Algorithmen darstellen.

In dieser Arbeit sollen Algorithmen für die Multiplikation von Matrizen mit dem ISA entwickelt werden. Die Anwendungen der Matrizenmultiplikation sind vielfältig. So sind zum Beispiel bei der Verwendung von Finite-Elemente-Methoden große Matrizen miteinander zu multiplizieren. Bei der Simulation von Modellen in der Strömungsmechanik oder in der mathematischen Optimierung treten ebenfalls sehr große Matrizen auf. Diese müssen entweder multipliziert werden oder sie treten als Koeffizientenmatrizen linearer Gleichungssysteme auf.

Es ist also sehr nützlich, über effiziente Algorithmen für die Matrixmultiplikation zu verfügen. Durch seine massive Parallelität ermöglicht das ISA schnelle Algorithmen zur Multiplikation von Matrizen.

Praktische Daten haben jedoch häufig die Eigenschaft, daß die entsprechenden Matrizen extrem schwach besetzt sind. Die bislang existierenden ISA-Algorithmen nutzen diese Tatsache aber nicht aus.

1.1 Aufgabenstellung

Die Aufgabenstellung dieser Arbeit ist es, Algorithmen[2] für die parallele Multiplikation dünn besetzter Matrizen mit dem ISA zu entwerfen. Genauer ist die Problemstellung die folgende:

Gegeben sind die dünn besetzten Matrizen $A \in \mathbb{R}^{n \times k}$ und $B \in \mathbb{R}^{k \times m}$ mit $A = (a_{ij})$ und $B = (b_{ij})$.

Aufgabe ist, die Matrix $C \in \mathbb{R}^{n \times m}$ mit $C = (c_{ij}) = A \cdot B$ zu berechnen. Dabei ist das Matrizenprodukt durch

$$c_{ij} = \sum_{\nu=1}^{k} a_{i\nu} b_{\nu j} \qquad \forall i \in \{1, \ldots, n\},\ j \in \{1, \ldots, m\}$$

definiert.

[1] engl.: instruction systolic array oder kurz: ISA.

[2] Für den Rest dieser Arbeit ist mit dem Begriff *Algorithmus*, soweit nicht explizit etwas anderes gesagt wird, stets ein paralleler Algorithmus für ein ISA gemeint.

Bei praktischen Problemen sind viele Einträge der Operandenmatrizen A und B Null. Solche Matrizen bezeichnet man auch als *dünn besetzt*.

Ziel der Arbeit ist es nun, die dünn besetzte Struktur der Matrizen A und B auszunutzen, um eine bessere Periode und damit ein günstigeres Laufzeitverhalten gegenüber dem von Lang in [3] vorgestellten Algorithmus zur Multiplikation von Matrizen zu erreichen. Außerdem soll das Laufzeitverhalten mit dem von sequentiellen Algorithmen verglichen werden.

Middendorf et. al. haben in [4] Algorithmen zur Multiplikation dünn besetzter Matrizen auf Prozessorfeldern mit rekonfigurierbaren Bussen angegeben. Das den Algorithmen zugrunde liegende Berechnungsmodell ist ein anderes als hier. Das hier betrachtete Prozessorfeld besitzt keine rekonfigurierbaren Busse. Trotzdem ist es teilweise möglich, die Ideen aus [4] zu verwenden, um Algorithmen für das hier betrachtete ISA zu entwickeln.

Im ersten Teil der Arbeit werden einige Aussagen über die dünn besetzte Struktur des Ergebnisses der Multiplikation zweier dünn besetzter Matrizen getroffen. Außerdem wird eine Datenstruktur zur kompakten Abspeicherung der Matrizen A und B angegeben sowie verschiedene Algorithmen entworfen und theoretisch analysiert. Einer dieser Algorithmen wird mit einem entsprechenden sequentiellen Algorithmus und dem Algorithmus aus [3] verglichen und eine Möglichkeit zu seiner Partitionierung angegeben. Im zweiten Teil wird dieser Algorithmus dann auf dem ISA *Systola 1024* der Firma ISATEC Soft- und Hardware GmbH implementiert, getestet und praktische auftretende Laufzeiten mit denen des sequentiellen Algorithmus verglichen.

1.2 Funktionsweise des ISA

Es soll nun kurz die Funktionsweise des ISA erläutert werden. Für eine nähere Einführung in Architektur und Programmierung des ISA sei hier auf [3] verwiesen. Es handelt sich beim ISA um ein Feld von Prozessoren, deren Verbindungsstruktur gitterartig ist. Es zählt prinzipiell zu den SIMD[3]-Rechnern, wobei allerdings die Möglichkeit besteht, Anweisungen für einzelne Prozessoren auszumaskieren. Eine schematische Übersicht eines 4×4 ISA zeigt Abbildung 1.

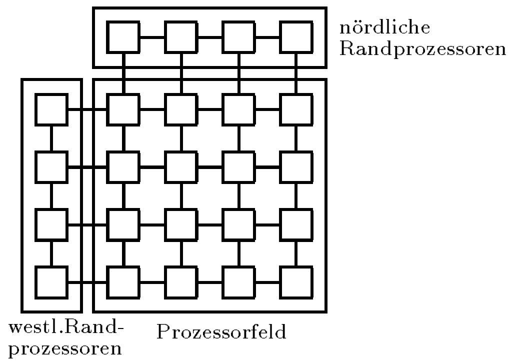

Abbildung 1: Schematische Darstellung eines 4×4 ISA.

[3]SIMD bedeutet Single Instruction Multiple Data.

Die einzelnen Prozessoren besitzen außer den Prozessorregistern keinen weiteren Speicher. Die Kommunikation zwischen benachbarten Prozessoren erfolgt über ein sogenanntes Kommunikationsregister **C**. Jeder der Prozessoren hat Zugriff auf die Kommunikationsregister seiner Nachbarn. Diese sind mit **CN**, **CS**, **CW** und **CE** für die 4 Himmelsrichtungen bezeichnet. Die Randprozessoren haben darüberhinaus Zugriff auf je eine Speicherbank, die in Form einer Queue[4] organisiert ist. Diese Speicherbänke ermöglichen die Datenein- und -ausgabe.

Der Programmablauf innerhalb des ISA funktioniert derart, daß eine Programmanweisung in den oberen linken Prozessor eingegeben wird, die Anweisung dort ausgeführt und dann an die Nachbarprozessoren weitergereicht wird, die die Anweisung wiederum nach der Ausführung weitergeben usw. Dabei geben Prozessoren der ersten Zeile Anweisungen an Ihren unteren und rechten Nachbarn weiter und alle anderen Prozessoren nur an den unteren Nachbarn. Auf diese Weise breitet sich ein Programm wellenfrontartig auf einer Diagonalen im Prozessorfeld aus. Die Ähnlichkeit dieser Ausbreitung mit einer Systole des Blutkreislaufs führte auch zu dem Namen systolisches Feld. Die Randprozessoren beachten nur Anweisungen, die auf das Kommunikationsregister zugreifen. Falls beispielsweise ein Befehl einen Wert aus dem Kommunikationsregister des nördlichen Nachbarn liest (etwa **C** := **CN**), so lesen die nördlichen Randprozessoren einen Wert aus ihrer Speicherbank und schreiben diesen in ihr Kommunikationsregister. Im nächsten Takt können nun die Prozessoren der ersten Zeile, die nun den Befehl ausführen, den Wert übernehmen. Entsprechendes gilt für schreibende Befehle (etwa **CN** := **C**). Analog funktionieren die westlichen Randprozessoren bei Zugriffen auf die Kommunikationsregister der westlichen Nachbarn.

Zusätzlich zu dem Befehlsstrom existiert ein Mechanismus zum Ausmaskieren einzelner Anweisungen in den Prozessoren. Für jeden Befehl existiert ein Zeilen- und ein Spaltenselektor, die mit den Anweisungen durch das Feld transportiert werden. Der Zeilenselektor ist ein n- und der Spaltenselektor ein m-Vektor von Bits, falls es sich um ein $n \times m$-ISA handelt. Nur dann wenn in einem Prozessor beide Selektoren eine 1 zeigen, wird der Befehl in diesem Prozessor ausgeführt. Ansonsten wird der Befehl ignoriert. Abbildung 2 auf Seite 2 veranschaulicht den Transport von Befehlen und Selektoren durch das ISA.

1.3 Ergebnisse

Dünn besetzte Matrizen lassen sich wie folgt charakterisieren: Sei A eine $n \times n$ Matrix. Sei r_A (c_A) die maximale Anzahl von Nichtnullelementen in einer Zeile (Spalte) von A. Sei ferner w_A eine ganze Zahl, so daß die Anzahl der Nichtnulleinträge von A kleiner oder gleich $w_A n$ ist. Dann ist A

- weakly-sparse $\iff w_A \in O(1)$
- row-sparse $\iff r_A \in O(1)$
- column-sparse $\iff c_A \in O(1)$.

[4] auch FIFO-Speicher genannt.

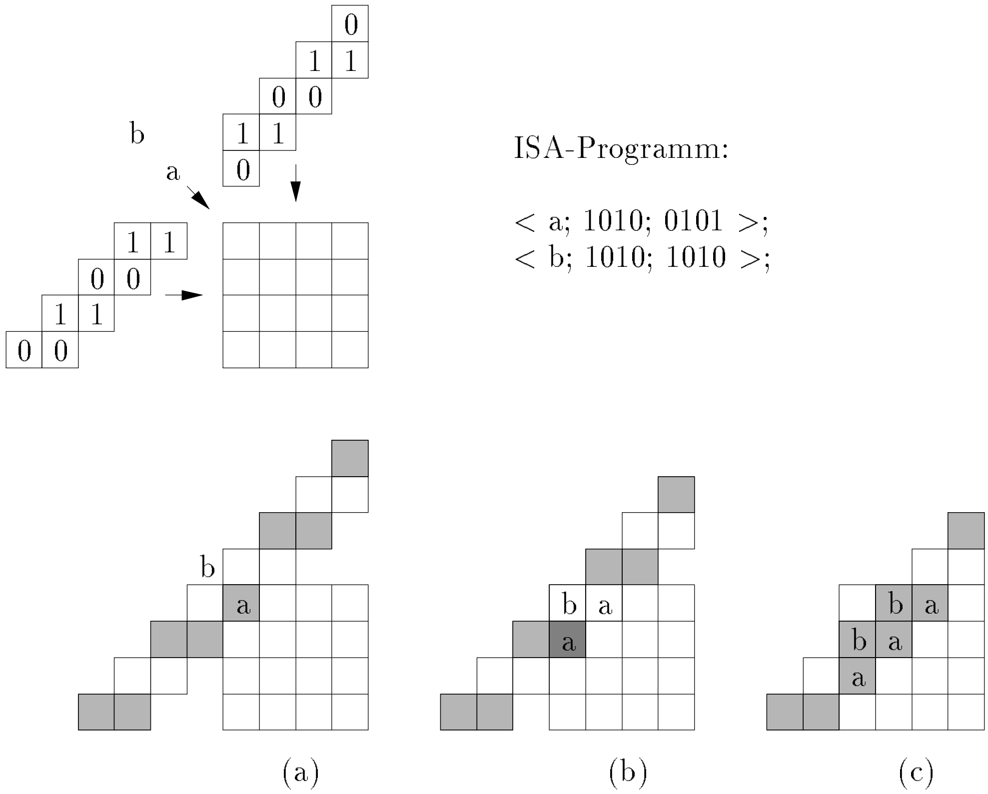

Abbildung 2: ISA-Programm mit den ersten drei Schritten seiner Ausführung

Nach dieser Klassifizierung zerfällt die Aufgabe aus 1.1 in verschiedene Teilprobleme. Diese werden entsprechend der dünn besetzten Struktur der Operanden mit Großbuchstaben bezeichnet. Beispielsweise wird RC zur Bezeichnung der Multiplikation einer row-sparse mit einer column-sparse Matrix benutzt. Diese Bezeichnungen werden auch für die entsprechenden Algorithmen benutzt.

Es werden hier Algorithmen für die Probleme RC, WC (RW) und CC (RR) angegeben. Dabei ist das Problem RW dual zu WC und RR zu CC. Das bedeutet diese Probleme können mit analogen Algorithmen berechnet werden. Die Algorithmen für die dualen Probleme erhält man durch Spiegeln an der Hauptdiagonale des ISA.

Die Algorithmen multiplizieren eine $n \times k$ Matrix A mit einer $k \times n$ Matrix B auf einem $n \times n$ Prozessorfeld. Dabei hat RC eine Periode von $O(r_A c_B)$ für die Multiplikation einer row-sparse mit einer column-sparse Matrix. Der Algorithmus WC erreicht eine Periode von $O(w_A n)$ für die Multiplikation einer weakly-sparse mit einer column-sparse Matrix. Man beachte, daß dies für $k \gg n$ besser ist, als der normale Algorithmus für die Matrizenmultiplikation auf einem befehlssystolischen Prozessorfeld. Der Algorithmus für die Multiplikation zweier column-sparse Matrizen (CC) hat eine Periode von $O(c_A n)$. Allerdings kann dieser Algorithmen nur $n \times n$ Matrizen miteinander multiplizieren, stellt also keine Verbesserung gegenüber normaler Matrixmultiplikation dar.

Da die Periode des Algorithmus RC nur von den Parametern der Operanden abhängt, wird dieser Algorithmus auf der Systola 1024 implementiert und seine Laufzeit mit der eines sequentiellen Algorithmus verglichen. Da der sequentielle Algorithmus mit einer Zeitkomplexität $T(n) \in O(n^2 \cdot \max\{r_A, c_B\})$ implementiert werden kann, ergibt sich hier zwangsläufig nur für kleine Werte von r_A und c_B eine Verbesserung. In Tests des Algorithmus RC erwies sich, daß für $r_A \leq n$ und $c_B \leq n$ das entsprechende Programm für ein $n \times n$-ISA besser als der implementierte sequentielle Algorithmus abschneidet.

Außerdem kann durch Ausnutzung von 0-Zeilen in A bzw. 0-Spalten in B noch eine Verbesserung der Periode von RC um den konstanten Faktor n/a erreicht werden. Dabei n die Größe des verwendeten ISA und a die Anzahl der Nichtnullzeilen von A bzw. -spalten von B.

2 Entwurf und Analyse der Algorithmen

In diesem Teil der Arbeit werden verschiedene Algorithmen für das in 1.1 vorgelegte Problem entworfen und analysiert. Es sind dies die Algorithmen für die Probleme WC, CC und RC. Der Algorithmus für letzteres Problem wird zusätzlich mit einem entsprechenden sequentiellen Algorithmus verglichen, und es wird eine Möglichkeit zu seiner Partitionierung angegeben.

Zuerst werden jedoch einige grundlegenden Aussagen über dünn besetzte Matrizen und die Ergebnisse der Multiplikation solcher Matrizen getroffen. Danach werden Datenstrukturen zur Abspeicherung dünn besetzter Matrizen angegeben.

2.1 Grundlagen und Definitionen

Um eine Datenstruktur zur kompakten Speicherung und einen Algorithmus zur Multiplikation dünn besetzter Matrizen entwerfen zu können, muß nun zuerst die „Dünnbesetztheit" von Matrizen näher charakterisiert und exakt gefaßt werden. Es wird hier die Charakterisierung aus [4] verwendet.

Definition 2.1. *Eine $n \times m$ Matrix M heißt* r-row-sparse *(zeilenweise dünn besetzt) genau dann, wenn jeder der Zeilenvektoren m_i, $i \in \{1, \ldots n\}$ von M höchstens r Nichtnulleinträge enthält.*

M heißt c-column-sparse *(spaltenweise dünn besetzt) genau dann, wenn jeder der Spaltenvektoren m_j, $j \in \{1, \ldots, m\}$ von M höchstens c Nichtnulleinträge enthält.*

Eine $n \times n$ Matrix N heißt w-weakly-sparse *(„schwach" dünn besetzt) genau dann, wenn N höchstens wn Nichtnulleinträge enthält.*

w, c und r sind jeweils Konstanten, die nicht von der Größe der Matrix abhängen.

Es fällt zunächst auf, daß die Charakterisierung als weakly-sparse auf quadratische Matrizen beschränkt ist. Der Begriff läßt sich jedoch auf nichtquadratische Matrizen erweitern. Damit lassen sich die hier für quadratische weakly-sparse Matrizen gezeigten Ergebnisse jeweils auch auf nichtquadratische Matrizen anwenden.

Definition 2.2. *Eine $n \times m$ Matrix M heißt (w_r, w_c)-weakly-sparse genau dann, wenn M höchstens $w_r n = w_c m$ Nichtnulleinträge enthält.*

Weiterhin ist zu bemerken, daß für $n \times n$ Matrizen N folgendes gilt: Falls N r-row-sparse (c-column-sparse) ist, dann ist N auch r-(c-)weakly-sparse. Die Umkehrung gilt nicht, wie man sich leicht durch das folgende Gegenbeispiel überzeugen kann.

Beispiel 2.1. *Man betrachte die Matrix*

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

Diese Matrix ist 2-weakly-sparse jedoch weder row- noch column-sparse.

Seien A und B die Matrizen aus 1.1. Es ist nun weiterhin von Interesse, ob es möglich ist, Aussagen über die Dünnbesetzheit des Produktes $C = A \cdot B$ zu treffen, denn die Matrix C muß nach der Berechnung auf irgendeine Weise innerhalb des ISA gespeichert werden. Dies ist nötig, weil aus Effizienzgründen die gesamte Berechnung ausschließlich im ISA stattfinden soll. Wenn man verlangt, daß die Anzahl der Register der Prozessoren des ISA nicht mit den Parametern[5] der Operanden A und B zusammenhängt, kann jeder Prozessor nur eine konstante Anzahl an Einträgen von C – beispielsweise einen Eintrag – speichern. Das bedeutet, daß Aussagen über die Dünnbesetztheit von C auf die maximal mögliche Größe der Operanden, die ohne Partitionierung von einem ISA fest vorgegebener Größe miteinander multipliziert werden können, schließen lassen.

Zur Vereinfachung sei angenommen, daß A und B $n \times n$ Matrizen sind. Die Ergebnisse die sich auf Operanden, die nicht nur weakly-sparse sind, beziehen, lassen sich auf die Matrizen aus 1.1 verallgemeinern.

Lemma 2.3. *Sei A c_A-column-sparse und B w_B-weakly-sparse. Weiterhin sei $C = A \cdot B$. Dann ist C $c_A w_B$-weakly-sparse.*

Beweis. Man betrachte das Matrixprodukt von A und B als Summe äußerer Produkte der Spalten von A und der Zeilen von B, d. h.:

$$C = A \cdot B = \sum_{i=1}^{n} a_i \cdot b_i,$$

wobei a_i die i-te Spalte von A und b_i die i-te Zeile von B ist. Das äußere Produkt $a_i \cdot b_i$ ist eine $n \times n$ Matrix. Abbildung 3 verdeutlicht die Situation.

Es sei nun n_i die Anzahl der Nichtnulleinträge der i-ten Zeile b_i von B. Da A c_A-column-sparse ist, kann das äußere Produkt $a_i \cdot b_i$ höchstens $c_A n_i$ Nichtnulleinträge enthalten. Daraus folgt, daß C höchstens

$$\sum_{i=1}^{n} c_A n_i = c_A \sum_{i=1}^{n} n_i = c_A w_B n$$

[5] Größe sowie Grad der Dünnbesetztheit

$$a_i \cdot b_i = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \cdot \begin{pmatrix} y_1 & \cdots & y_n \end{pmatrix} = \begin{pmatrix} x_1 y_1 & \cdots & x_1 y_n \\ & \vdots & \\ x_n y_1 & \cdots & x_n y_n \end{pmatrix}$$

Abbildung 3: Bildung eines äußeren Produktes

Nichtnulleinträge enthält, da bei der Addition der verschiedenen äußeren Produkte höchstens alle Positionen von Nichtnulleinträgen für je zwei der Summanden paarweise verschieden sind. □

Korollar 2.4. *Sei A w_A-weakly-sparse und B r_B-row-sparse. Dann ist C $w_A r_B$-weakly-sparse.*

Beweis. Die Argumentation verläuft analog zu dem Beweis von Lemma 2.3. □

Korollar 2.5. *Sei A c_A-column-sparse und B r_B-row-sparse. Dann ist C $c_A r_B$-weakly-sparse.*

Beweis. Da B auch r_B-weakly-sparse ist, folgt mit Lemma 2.3 das Ergebnis. □

Die Frage ist nun, ob sich C im letzteren Fall nicht sogar row- bzw. column-sparse ist. Das dies im allgemeinen nicht gilt, zeigt das folgende Beispiel.

Beispiel 2.2. *Man betrachte die beiden Matrixprodukte*

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

und

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

In beiden Fällen ist der erste Operand 1-column-sparse und der zweite 1-row-sparse. Das erste Produkt ist jedoch nicht column-sparse und das zweite nicht row-sparse.

Daraus folgt, daß sich die Aussage des Korollar 2.5 nicht verbessern läßt. Für den Fall, daß A und B beide row- bzw. beide column-sparse sind, erzielt man jedoch eine bessere Aussage als die des Lemma 2.3.

Lemma 2.6. *Sei A c_A column-sparse und B c_B column-sparse. Dann ist C $c_A c_B$-column-sparse.*

Beweis. Wie im Beweis von Lemma 2.3 betrachtet man wieder C als Summe äußerer Produkte. Man betrachte das j-te Element der Zeile b_i von B. Dieses sei mit b_{ij} bezeichnet. Falls $b_{ij} \neq 0$ ist, stehen in der j-ten Spalte von $a_i \cdot b_i$ höchstens c_A Nichtnulleinträge, da A c_A-column-sparse ist. Anderenfalls ist die

j-te Spalte dieser Matrix leer. Der Fall, daß b_{ij} ein Nichtnulleintrag ist, tritt aber in jeder Spalte j von B und damit für die n äußeren Produkte nur c_B mal auf, da B c_B-column-sparse ist. Das bedeutet aber, daß in jeder Spalte der Summe C der äußeren Produkte höchstens $c_A c_B$ Nichtnulleinträge stehen können. □

Korollar 2.7. *Sei A r_A-row-sparse und B r_B row-sparse. Dann ist C $r_A r_B$-row-sparse.*

Beweis. Die Argumentation ist analog zu der im Beweis von Lemma 2.6. □

Korollar 2.8. *Sei A sowohl c_A-column- als auch r_A-row-sparse und entsprechend B sowohl c_B-column- als auch r_B-row-sparse. Dann ist C sowohl $c_A c_B$-column-sparse als auch $r_A r_B$-row-sparse.*

Beweis. Dies folgt sofort aus Lemma 2.6 und Korollar 2.7. □

Für alle restlichen Fälle läßt sich keine Aussage über die Dünnbesetzheit des Ergebnisses C der Matrixmultiplikation treffen. Es läßt sich für diese Fälle ein Gegenbeispiel finden, in dem das Produkt zweier dünn besetzter Matrizen voll besetzt ist.

Beispiel 2.3.

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Die erste Matrix ist offenbar 1-row-, also 1-weakly-sparse, die zweite ist 1-column-, also 1-weakly-sparse. Das Ergebnis der Multiplikation ist voll besetzt.

Die Tabelle 1 zeigt noch einmal die Ergebnisse dieses Abschnitts auf einen Blick. „R“, „C“ bzw. „W“ bedeutet dabei, daß die entsprechende Matrix row-, column bzw. weakly-sparse ist. „—“ bedeutet, daß das Ergebnis voll besetzt sein kann. Der Parameter für den Grad der Dünnbesetztheit ergibt sich für die letzten 5 Fälle jeweils als Produkt der entsprechenden Parameter der beiden Operanden.

Die Großbuchstaben aus Tabelle 1 werden auch zur Bezeichnung der Probleme und Algorithmen benutzt. Beispielsweise bezeichnet „WC“ die Multiplikation einer weakly-sparse Matrix A mit einer column-sparse Matrix B bzw. die entsprechenden Algorithmen

In dieser Arbeit werden Algorithmen für die Probleme WC, RC und CC angegeben und analysiert. Dabei sind die Algorithmen für RC und CC direkte Umsetzungen der entsprechenden Algorithmen aus [4]. Der Algorithmus CC macht keinen Gebrauch von den Eigenschaften laut Tabelle 1.

Für das Problem WC ist RW dual, d. h. mit einem analogen Algorithmus zu berechnen. Entsprechend verhält es sich mit CC und RR. Allerdings ist RC keineswegs dual zu CR.

Für die Algorithmen RC und WC wird ein $n \times m$ Prozessorfeld benötigt, um die $n \times k$ Matrix A mit der $k \times m$ Matrix B zu multiplizieren, da das Ergebnis voll besetzt sein kann und jeder Prozessor zuletzt eines der nm Skalarprodukte

A	B	$C = A \cdot B$
W	W	—
R	W	—
W	C	—
R	C	—
C	W	W
W	R	W
C	R	W
C	C	C
R	R	R

Tabelle 1: Dünn besetzte Ergebnisse bei der Matrixmultiplikation dünn besetzter Matrizen

speichern soll. Für den Algorithmus CC wird ein $n \times n$ Prozessorfeld benutzt, da eine der Operandenmatrizen im Feld gespeichert wird und nicht während der Berechnung über den Rand eingegeben wird.

2.2 Datenstrukturen für dünn besetzte Matrizen

Sei $A \in \mathbb{R}^{n \times k}$ eine r-row-sparse und $B \in \mathbb{R}^{k \times m}$ eine c-column-sparse Matrix. Dann hat jede Zeile von A höchstens r bzw. jede Spalte von B höchstens c Nichtnulleinträge. Diese lassen sich im Fall von A in einer $n \times r$ Matrix $\tilde{A}$ bzw. im Falle von B in einer $c \times m$ Matrix $\tilde{B}$ kompakt abspeichern. Der Eintrag an der Position (i, j) der Matrix $\tilde{A}$ ist dann der j-te Nichtnulleintrag der i-ten Zeile von A. Analog enthält $\tilde{B}$ an der Position (i, j) den i-ten Nichtnulleintrag der j-ten Spalte von B.

Zusätzlich zu den von 0 verschiedenen Daten muß für jeden Eintrag in $\tilde{A}$ bzw. $\tilde{B}$ der Spalten- bzw. Zeilenindex bezüglich der Ausgangsmatrix gespeichert werden. Damit ist jeder Eintrag von $\tilde{A}$ ($\tilde{B}$) ein Paar (i, d), mit $d \in \mathbb{R}$ und $i \in \{1, \ldots, k\}$.

In einer PASCAL ähnlichen Notation ergibt sich folgende Datenstrukturen:

```
TRowSparseMatrix = Array[1..n,1..c] of (ℕ,ℝ);
TColumnSparseMatrix = Array[1..r,1..m] of (ℕ,ℝ);
A: TRowSparseMatrix;
B: TColumnSparseMatrix;
```

Beispiel 2.4. *Man betrachte die Matrizen*

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{pmatrix} \quad \textit{und} \quad B = \begin{pmatrix} 1 & 2 & 3 & 4 & 0 & 0 & 0 & 0 \\ 5 & 6 & 7 & 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{pmatrix}.$$

Die Matrix A ist 3-row-sparse und B ist 3-column-sparse. A und B können *wie folgt kompakt gespeichert werden:*

$$\tilde{A} = \begin{pmatrix} (1,2) & (2,1) & (0,0) \\ (1,1) & (2,2) & (3,1) \\ (2,1) & (3,2) & (4,1) \\ (3,1) & (4,2) & (5,1) \\ (4,1) & (5,2) & (0,0) \end{pmatrix}$$

und

$$\tilde{B} = \begin{pmatrix} (1,1) & (1,2) & (1,3) & (1,4) & (3,1) & (3,2) & (3,3) & (3,4) \\ (2,5) & (2,6) & (2,7) & (2,8) & (4,5) & (4,6) & (4,7) & (4,8) \\ (5,1) & (5,2) & (5,3) & (5,4) & (5,5) & (5,6) & (5,7) & (5,8) \end{pmatrix}$$

Die überflüssigen Einträge können beispielsweise wie hier bei A mit 0 gefüllt werden. Man spart mit $\tilde{A}$ zwei Spalten und mit $\tilde{B}$ zwei Zeilen.

Um $n \times m$ Matrizen abzuspeichern, die (w_r, w_c)-weakly-sparse sind, gibt es zwei verschiedene Möglichkeiten. Man kann die $w_r n$ Einträge entweder zeilen- oder $w_c m$ Einträge spaltenweise abspeichern, d. h. mit w_r Einträgen in jeder Zeile bzw. w_c Einträgen in jeder Spalte der Datenstruktur. Dabei muß jedoch in jedem abgespeicherten Eintrag sowohl Zeilen- als auch Spaltenindex des Nichtnulleintrags der Matrix gespeichert werden. Einträge in der Datenstruktur sind also Tripel (z, s, d), wobei z und s die Zeilen- bzw. Spaltenindizes sind und d der Nichtnulleintrag aus Zeile z und Spalte s der Ausgangsmatrix. Insgesamt ergibt sich also:

```
TWeaklySparseMatrix_row = Array[1..n,1..wr] of (IN,IN,IR);
TWeaklySparseMatrix_col = Array[1..wc,1..n] of (IN,IN,IR);
```

Welche der beiden Datenstrukturen man verwendet hängt davon ab, welcher der Operanden einer Matrixmultiplikation weakly-sparse ist. Für das Problem WC benutzt man die zeilenweise Datenstruktur zur Speicherung der Matrix A, im dualen Fall RW die spaltenweise Datenstruktur für B.

Beispiel 2.5. *Die Matrix*

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 \end{pmatrix}$$

ist 2-weakly-sparse und kann wie folgt kompakt gespeichert werden:

$$\begin{pmatrix} (1,1,1) & (1,2,2) \\ (1,3,3) & (1,4,4) \\ (2,1,5) & (3,1,6) \\ (4,1,7) & (0,0,0) \end{pmatrix} \quad \textit{bzw.} \quad \begin{pmatrix} (1,1,1) & (3,1,6) & (1,2,2) & (1,4,4) \\ (2,1,5) & (4,1,7) & (1,3,3) & (0,0,0) \end{pmatrix}$$

Dabei ist die Ausgangsmatrix im ersten (linken) Fall zeilenweise *gespeichert und im zweiten (rechten) Fall* spaltenweise*.*

Da über das Ergebnis $C \in \mathbb{R}^{n \times m}$ der Multiplikation von A und B keine Aussage bezüglich Dünnbesetztheit getroffen werden kann[6], wird diese Matrix einfach in einem $n \times m$ Array gespeichert:

```
TMatrix = Array[1..n,1..m] of IR;
C: TMatrix;
```

2.3 Ein sequentieller Algorithmus für RC

Der parallele Algorithmus für RC ist auf der Systola 1024 umgesetzt worden. Deshalb soll für dieses Problem auch ein sequentieller Algorithmus angegeben werden, um praktische Laufzeiten vergleichen zu können.

Es werden bei RC eine r-row-sparse Matrix A mit einer c-column-sparse Matrix B multipliziert. Die entsprechenden Datenstrukturen sind mit $\tilde{A}$ bzw. $\tilde{B}$ bezeichnet.

2.3.1 Idee des Algorithmus

Der sequentielle Algorithmus berechnet nm Skalarprodukte. Für jedes der Skalarprodukte werden zwei Vektoren $\tilde{a}$, $\tilde{b}$ mit Elementen aus $\mathbb{N} \times \mathbb{R}$ betrachtet, wobei $\tilde{a}$ eine Zeile von $\tilde{A}$ und $\tilde{b}$ eine Spalte von $\tilde{B}$ ist. Um nun das Skalarprodukt der entsprechenden Daten zu berechnen, werden beginnend mit dem ersten Eintrag von $\tilde{a}$ und $\tilde{b}$ sukzessiv die Indizes der Einträge betrachtet, wobei genau drei Fälle möglich sind:

1. Der Index des momentan betrachteten Eintrags in $\tilde{a}$ ist echt größer als der des Eintrags in $\tilde{b}$. Dann betrachte den nächsten Eintrag von $\tilde{b}$.

2. Der Index des momentan betrachteten Eintrags in $\tilde{a}$ ist echt kleiner als der des Eintrags in $\tilde{b}$. Dann betrachte den nächsten Eintrag von $\tilde{a}$.

3. Die beiden betrachteten Indizes sind gleich. Dann trägt das Produkt der Daten der beiden Einträge zum Skalarprodukt bei.

Damit die beiden Vektoren $\tilde{a}$ und $\tilde{b}$ auf diese Weise miteinander verknüpft werden können, muß allerdings vorausgesetzt werden, daß in jeder Zeile von $\tilde{A}$ bzw. jeder Spalte von $\tilde{B}$ die Einträge nach dem Index aufsteigend sortiert vorliegen.

2.3.2 Pseudocode

Der folgende sequentielle Algorithmus berechnet das Produkt der Matrizen A und B:

```
procedure MatMult_sequenz(A: TRowSparseMatrix; B: TColumnSparseMatrix;
  C: TMatrix);
var
  x, y, i, j: Word;
```

[6] siehe Beispiel 2.3.

```
Begin
  for x:=1 to n do
    for y:=1 to m do begin

      { duennes Skalarprodukt ausrechnen}
      C[x,y] := 0;
      i := 1; j := 1;
      while (i <= r) and (j <= c) do begin

        { Indizes weitersetzen}
        if A[x,i].index < B[j, y].index then Inc(i)
        else if A[x,i].index > B[j,y].index then Inc(j)

        { nichtausloeschende Faktoren }
        else Begin
          C[x,y] := C[x,y] + A[x,i].data * B[j,y].data;
          Inc(i);    { Indizes weitersetzen }
          Inc(j);
        end;      { if }
      end;      { while }
    end;      { for }
End;
```

2.3.3 Laufzeitanalyse

Die Fallunterscheidung aus 2.3.1 wurde in die `if`-Anweisung innerhalb der `while`-Schleife umgesetzt. Da diese Fallunterscheidung vollständig ist, folgt, daß in jedem Durchlauf der inneren (`while`) Schleife mindestens eine der Laufvariablen `i` oder `j` um 1 erhöht wird.

Die Schleife wird beendet, sobald eine dieser beiden Laufvariablen ihren Vektor durchlaufen hat. Im ungünstigsten Fall wird wie in Beispiel 2.6 in jedem Durchlauf jeweils nur eine der Laufvariablen erhöht, wobei `i` und `j` derart erhöht werden, daß bei Abbruch der Schleife entweder `i=r` und `j=c-1` oder `i=r-1` und `j=c` gilt. Damit gilt für die Laufzeit T_i der inneren Schleife

$$T_i \leq 2\alpha(r+c),$$

wobei α eine positive Konstante ist.

Beispiel 2.6. *Man betrachte die dünn besetzten Vektoren*

$$\begin{aligned} a &= (a_1, a_2, \ldots, a_{r-1}, \underbrace{0, \ldots, 0}_{c\,mal}, a_r)^T \quad \textit{und} \\ b &= (\underbrace{0, \ldots, 0}_{(r-1)\,mal}, b_1, b_2, \ldots, b_c, 0)^T. \end{aligned}$$

Werden diese gemäß dem obigen Algorithmus miteinander skalar multipliziert, dann benötigt die `while`*-Schleife insgesamt* $(r-1)+c$ *Durchläufe.*

Insgesamt werden nm Skalarprodukte berechnet. Damit ergibt sich die Gesamtlaufzeit des Algorithmus zu

$$T_{\text{seq}}(n,m,r,c) = \alpha_1 nm T_i \leq \alpha_2 nm(r+c) \in O(nm(r+c)),$$

mit positiven Konstanten α_1 und $\alpha_2 = 2\alpha\alpha_1$.

2.4 Ein paralleler Algorithmus für RC

2.4.1 Idee des Algorithmus

Jedes der nm Skalarprodukte wird von einem Prozessor des Prozessorfeldes berechnet. Es sei wiederum $\tilde{a}$ eine Zeile von $\tilde{A}$ und $\tilde{b}$ eine Spalte von $\tilde{B}$. Ein schrittweises Abarbeiten der Indizes in den Einträgen von $\tilde{a}$ und $\tilde{b}$ durch den entsprechenden Prozessor ähnlich wie beim sequentiellen Algorithmus, ist beim ISA aus Effizienzgründen nicht sinnvoll, da die Daten kontinuierlich durch das Prozessorfeld strömen sollen.

Um die korrespondierenden Einträge von $\tilde{a}$ und $\tilde{b}$ zusammenzuführen, werden nacheinander alle auftretenden Indizes in $\tilde{a}$ mit denen in $\tilde{b}$ verglichen. Jeder der Prozessoren des Feldes führt also folgenden Algorithmus aus:[7]

```
sum := 0;
for x := 1 to r do
  for y := 1 to c do
    if ã[x].index = b̃[y].index then
      sum := sum + ã[x].data * b̃[y].data;
```

Für das Prozessorfeld insgesamt bedeutet dies, daß die Matrix $\tilde{A}$ von links spaltenweise und die Matrix $\tilde{B}$ von oben zeilenweise eingegeben wird. Dabei werden für jede Spalte von $\tilde{A}$ jeweils alle Zeilen von $\tilde{B}$ bearbeitet. Abbildung 4 zeigt, wie die nm Skalarprodukte den Prozessoren zugeordnet werden.

Der hier vorgestellte Algorithmus ähnelt in seiner Arbeitsweise dem Algorithmus RC aus [2] bzw. [4]. Der Unterschied besteht jedoch darin, daß dort davon ausgegangen wird, daß die Matrizen A und B zu Beginn des Algorithmus in den Prozessorregistern gespeichert sind, während hier die Daten nach und nach von links und oben eingegeben werden. Außerdem wird in [2] von einem etwas leistungstärkeren Modell eines Parallelrechners, einem Prozessorfeld mit rekonfigurierbaren Bussen, ausgegangen.

2.4.2 Pseudocode

Der Pseudocode für den ISA–Algorithmus wird hier gemäß der Notation aus [3] angegeben. Da die behandelten Matrizen A und B reelle Zahlen als Einträge besitzen, wird davon ausgegangen, daß die Prozessoren des ISA reelle Zahlen verarbeiten können, d. h.:

1. Es existieren Register, die Gleitkommazahlen[8] speichern können, bzw. es lassen sich normale Register derart zusammenfassen, daß dies möglich ist.

[7] Die Vektoren $\tilde{a}$ und $\tilde{b}$ werden hier wie ein Array behandelt.
[8] beispielsweise im IEEE–Single– oder IEEE–Double–Format.

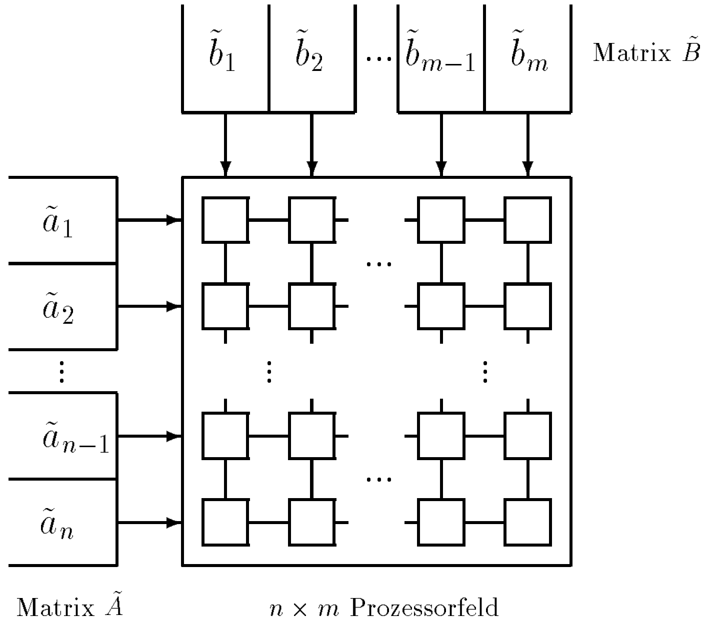

Abbildung 4: Eingabe der Matrizen von Westen und Norden

2. Die Prozessoren beherrschen Gleitkommaarithmetik, bzw. diese wird mittels geeigneter Software bereitgestellt.

Die beiden Routinen

```
Zeilen_Senden_Eintrag(z: Zeilenselektor; Index, Data: Register);
Spalten_Senden_Eintrag(s: Spaltenselektor; Index, Data: Register);
```

lesen jeweils einen Eintrag bestehend aus einem Index und einem Datum aus den Randprozessoren ein, senden diesen über die gesamte Zeile bzw. Spalte und speichern die gelesenen Werte in den Registern `Index` und `Data` des jeweiligen Prozessors. Das Register `Data` kann Gleitkommazahlen speichern. Diese Funktionalität läßt sich mit Hilfe der Routinen `Zeilen_Senden` und `Spalten_Senden` aus [3] programmieren.

Der Pseudocode für das ISA–Programm sieht damit folgendermaßen aus:

```
PROCEDURE MatMult(n, m, r, c: Natural);
BEGIN
  for x:=1 to r do
  BEGIN

    { Eine Spalte von A einlesen }
    Zeilen_Senden_Eintrag(1^n, A_index, A_Data);
    for y:=1 to c do
    BEGIN

      { Eine Zeile von B einlesen }
```

```
      Spalten_Senden_Eintrag(1^m, B_index, B_data);

      { ggf. Skalarprodukt weiterrechnen }
      < temp := sum + A_Data * B_data, 1^n, 1^m >;
      < A_Index - B_Index, 1^n, 1^m >;
      < if Zero then sum := temp else sum := sum, 1^n, 1^m >

    END
  END
END;
```

Die Daten des jeweils betrachteten Eintrags von $\tilde{A}$ werden in den Registern `A_index` und `A_data` gespeichert. Analog für $\tilde{B}$. Das Register `temp` ist ein Hilfsregister. In `sum` wird das Skalarprodukt akkumuliert. Die beiden letzteren Register sowie `A_data` und `B_data` sind Register für Gleitkommazahlen.

2.4.3 Laufzeitanalyse

Um die Laufzeit eines ISA–Programms abzuschätzen, betrachtet man seine Periode. Das ist die Anzahl der ISA–Programmdiagonalen die sich ergibt, wenn alle Schleifen linearisiert werden.

Die beiden Routinen zum Senden der Zeilen bzw. Spalten besitzen lediglich die konstante Periode β_1. Der Vergleich, die bedingte Zuweisung und die Gleitkommaoperationen innerhalb der beiden `for` Schleifen besitzen die konstante Periode β_2. Somit ergibt sich die Gesamtperiode des ISA–Programms als

$$T_{\text{isa}}(r, c) = r(\beta_1 + c(\beta_1 + \beta_2)) \in O(rc).$$

T_{isa} ist für ein hinreichend großes Prozessorfeld also nicht mehr von n und m abhängig.

Dieses Ergebnis entspricht dem des Algorithmus RC aus [4]. Das bedeutet, daß Algorithmen für die Matrixmultiplikation auf Prozessorfeldern mit fester Verbindungsstruktur entgegen der Behauptung in [4] die dünn besetzte Struktur der Operanden ausnutzen können, um Operationen zu sparen.

2.5 Vergleich von RC mit verschiedenen Algorithmen

2.5.1 Vergleich mit dem sequentiellen Algorithmus

Vergleicht man die Laufzeit T_{seq} des sequentiellen Algorithmus mit der Periode T_{isa} des ISA–Programms, so ist zunächst festzustellen, daß T_{isa} unabhängig von den Größen n und m der Matrizen A und B ist, T_{seq} jedoch nicht.

Andererseit wächst T_{isa} quadratisch in den Größen c und r, während T_{seq} nur linear in diesen Größen wächst. Daraus lassen sich folgende Schlüsse ziehen:

1. Ist die Anzahl der Nichtnulleinträge c bzw. r klein gegen die Größe n und m des Prozessorfeldes, wird das ISA-Programm schneller als der sequentielle Algorithmus sein.

2. In dem Maße, in dem die Parameter c und r an Größe gegenüber den Parametern n und m gewinnen, wird sich die Laufzeit des ISA-Programms gegenüber dem sequentiellen Algorithmus verschlechtern.

Dies bedeutet, daß die Verwendung des ISA nur bei einer hinreichend großen Anzahl an Prozessoren tatsächlich eine Verbesserung gegenüber dem sequentiellen Algorithmus mit sich bringt.

Allerdings wird für den angegebenen sequentiellen Algorithmus vorausgesetzt, daß die Zeilen von $\tilde{A}$ bzw. die Spalten von $\tilde{B}$ nach dem Index aufsteigend sortiert sind, was für den ISA-Algorithmus nicht nötig ist.

2.5.2 Vergleich mit der herkömmlichen Matrixmultiplikation auf dem ISA

Lang [3] gibt die Periode des ISA–Programms zur Multiplikation zweier $n \times n$ Matrizen auf einen $n \times n$ ISA mit $O(n)$ an. Dies läßt sich einfach auf die Multiplikation einer $n \times k$ Matrix A mit einer $k \times m$ Matrix B auf einem $n \times m$ ISA verallgemeinern. Es ergibt sich eine Periode von $T(k) \in O(k)$.

Diese Periode ist genau wie T_{isa} nicht mehr von n und m abhängig. Vergleicht man T und T_{isa} miteinander so steht analog zu 2.5.1 ein quadratisches Wachstum von T_{isa} einem linearen von T gegenüber. Allerdings handelt es sich hier um verschiedene Parameter. Folgende Feststellung läßt sich treffen:

1. Besitzen A und B nur wenige Nichtnulleinträge, d. h. ist die Größe k der beiden Matrizen A und B groß gegen das Produkt cr der Anzahlen der Nichtnulleinträge ($cr < k$), dann wird der hier vorgestellte Algorithmus besser sein als der herkömmliche Algorithmus, der die dünn besetzte Struktur von A und B außer Acht läßt.

2. Ist dagegen $cr \geq k$, wird keine Verbesserung gegenüber dem herkömmlichen Algorithmus erzielt. Es ist im Gegenteil eine Verschlechterung der Laufzeit zu erwarten.

2.6 Partitionierung des Algorithmus RC

Bisher wurde stets angenommen, daß zur Multiplikation der Matrizen $A \in \mathbb{R}^{n \times k}$ und $B \in \mathbb{R}^{k \times m}$ ein $n \times m$ Prozessorfeld zur Verfügung steht. Dies ist jedoch in der Praxis zumeist nicht der Fall. Vielmehr ist die Größe des zu verwendenden ISA festgelegt, z. B. auf 32×32 Prozessoren, während die Größen n und m der Matrizen variabel sind. Außerdem ist die Anzahl der Nichtnulleinträge, die maximal für jede Zeile von der row-sparse Matzrix A bzw. für jede Spalte der column-sparse Matrix B in der in 2.2 vorgestellten Datenstruktur gespeichert werden können durch eine Konstante M nach oben beschränkt. Bei der Systola 1024 ergibt sich diese Schranke beispielsweise entweder durch die Größe des westlichen bzw. nördlichen Karten-RAM von je 256K oder bei Verwendung der PASCAL-Programmierschnittstelle durch die 64K-Größenbeschränkung für Datenstrukturen.

Aus diesem Grunde ist es nötig, ein Verfahren zur Partitionierung des vorgestellten Algorithmus anzugeben.

Es wird im Folgenden davon ausgegangen, daß ein $n \times n$-ISA zur Verfügung steht, wobei n eine Konstante ist. Bei der Systola 1024 ist beispielsweise $n = 32$.

Sei nun $A \in \mathbb{R}^{m \times k}$ und $B \in \mathbb{R}^{k \times l}$. Dann ist $C = A \cdot B \in \mathbb{R}^{m \times l}$. OBdA gilt $m = a \cdot n$ und $l = b \cdot n$ mit natürlichen Zahlen a, b. Falls dies nicht der Fall ist, können an die Matrix A entsprechend Nullzeilen, bzw. an B Nullspalten angehängt werden. Die Matrix A sei wieder r-row-sparse und B c-column-sparse.

2.6.1 Partitionierung für $r \leq M$ und $c \leq M$

In der hier vorgelegten Implementierung des ISA–Algorithmus ist die Konstante M so groß,[9] daß die Laufzeit des ISA–Programms für $r, c \approx M$ wesentlich schlechter als die des sequentiellen Algorithmus ist. Aus diesem Grunde ist die Annahme, daß $r, c \leq M$ sind, nicht sehr einschränkend. Im nächsten Abschnitt wird trotzdem kurz erläutert, wie man das Problem für $r \geq M$ oder $c \geq M$ partitioniert.

Die Matrizen A und B lassen sich in Blockschreibweise nun wie folgt darstellen:

$$A = \begin{pmatrix} A_1 \\ \vdots \\ A_a \end{pmatrix} \qquad B = (B_1 \cdots B_b),$$

mit $A_i \in \mathbb{R}^{n \times k}$ und $B_j \in \mathbb{R}^{k \times n}$ für $i = 1, \ldots, a$ und $j = 1, \ldots, b$. Jede der Matrizen A_i ist wieder r-row-sparse. Entsprechend ist jede der Matrizen B_j c-column-sparse. Damit ist

$$C = A \cdot B = \begin{pmatrix} A_1 \cdot B_1 & \cdots & A_1 \cdot B_b \\ & \vdots & \\ A_a \cdot B_1 & \cdots & A_a \cdot B_b \end{pmatrix} \in \mathbb{R}^{na \times nb}.$$

Jeder der Einträge dieser Matrix ist selbst eine $n \times n$ Matrix und kann, da die Faktoren je eine r-row-sparse und eine c-column-sparse Matrix sind, somit mit dem $n \times n$-ISA parallel berechnet werden. Dazu wird der Algorithmus aus 2.4 verwendet. Insgesamt werden ab verschiedene Blöcke berechnet. Die Komplexität ist somit in $O(abrc)$, da die ab Blockmatrizen des Ergebnisses C nur nacheinander berechnet werden können. Die Komplexität des sequentiellen Algorithmus aus 2.3 verschlechtert sich ebenfalls um den Faktor ab. Das bedeutet, daß das Verhältnis der Laufzeiten des sequentiellen und parallelen Algorithmus unverändert bleibt. Es gelten also nach wie vor die bereits in 2.5.1 angeführten Zusammenhänge.

2.6.2 Partitionierung für $r \geq M$ oder $c \geq M$

In dem Fall, daß sich die r Einträge einer Zeile von A nicht in einer Datenstruktur speichern lassen, werden diese Zeilen auf mehrere Datenstrukturen verteilt. Es werden dann für jeden der Blöcke $A_i, i = 1, \ldots, a$ insgesamt $h = \lceil r/M \rceil$

[9] Die entsprechende Programmkonstante ist `MaxSize_sng`; siehe 3.2.

Datenstrukturen benötigt, so daß die Matrix A folgendermaßen repräsentiert wird:

$$\tilde{A} = \begin{pmatrix} \tilde{A}_{11} & \cdots & \tilde{A}_{1h} \\ & \vdots & \\ \tilde{A}_{a1} & \cdots & \tilde{A}_{ah} \end{pmatrix}.$$

Der Block A_i wird also durch die Datenstrukturen $[\tilde{A}_{i1} \cdots \tilde{A}_{ih}]$ repräsentiert.

In entsprechender Art und Weise werden die Spalten in den Blöcken B_j aufgeteilt. Sei $g = \lceil c/M \rceil$ die Anzahl der Datenstrukturen, die zur Speicherung eines Blockes B_j benötigt werden.

Um nun die ab Komponenten $A_i \cdot B_j$ der Ergebnismatrix C zu berechnen, muß jede Datenstruktur $\tilde{A}_{i\nu}, \nu = 1, \ldots, h$ mit jeder Datenstruktur $\tilde{B}_{\mu j}, \mu = 1, \ldots, g$ durch den ISA–Algorithmus verknüpft werden. Die Komplexität für jede Komponente liegt in $O(ghM^2) = O(rc)$, da jede der Datenstrukturen höchsten M Einträge aus Zeilen von A bzw. Spalten von B enthält. Für die Berechnung der gesamten Matrix C ist die Komplexität also wie zuvor in $O(abrc)$.

2.7 Verbesserung von RC durch Ausnutzung von 0-Zeilen/Spalten

2.7.1 Grundlagen

Gegeben sind wieder dünnbesetzte Matrizen $A \in \mathbb{R}^{n \times k}$ und $B \in \mathbb{R}^{k \times n}$. Die Matrix A sei r-row-sparse und B sei c-column-sparse. Bei der Multiplikation der beiden Matrizen läßt sich folgende Tatsache ausnutzen:

- 0-Zeilen in A erzeugen 0-Zeilen im Ergebnis C und
- 0-Spalten in B erzeugen 0-Spalten in C.

Verwendet man den Algorithmus aus 2.4, um A und B zu multiplizieren, wird dies nicht ausgenutzt. Die entsprechenden Prozessoren könnten aber bereits für andere Berechnungen eingesetzt werden.

Angenommen B besitzt $\tilde{b}$ Nullspalten. Dann hat auch C $\tilde{b}$ Nullspalten. Es sei $b = n - \tilde{b}$. Die Matrix B ist also b-row-sparse.[10]

Es gelte weiterhin $n = 2^h$ und $b|n$, d. h. die Größe des verwendeten ISA und die Anzahl der Nichtnullspalten von B sind Zweierpotenzen. Die Einträge der Matrix B werden als Tripel (z, s, d) gespeichert. Dabei ist z der Zeilen-, s der Spaltenindex und d das Datum des Eintrags. Jede der b Nichtnullspalten der Matrix kann nun in n/b Abschnitte aufgeteilt werden, die parallel in das ISA eingegeben werden. Die resultierende Datenstruktur hat nur noch $\lceil \frac{c}{n/b} \rceil$ Zeilen. Die Abbildung 5 auf Seite 23 veranschaulicht die Aufteilung der Spalten von B, und Abbildung 6 zeigt, wie die Matrizen in das ISA eingegeben werden.

2.7.2 Algorithmus

Die Berechnung des Matrizenprodukts wird nun zunächst mit dem Algorithmus aus 2.4 durchgeführt. Dabei werden von den Einträgen der Matrix B jeweils nur die Zeilenindizes z betrachtet. Die Spaltenindizes s werden für die Berechnung

[10] Es folgt umgekehrt jedoch nicht, daß falls B b-row-sparse ist, B 0-Spalten enthält.

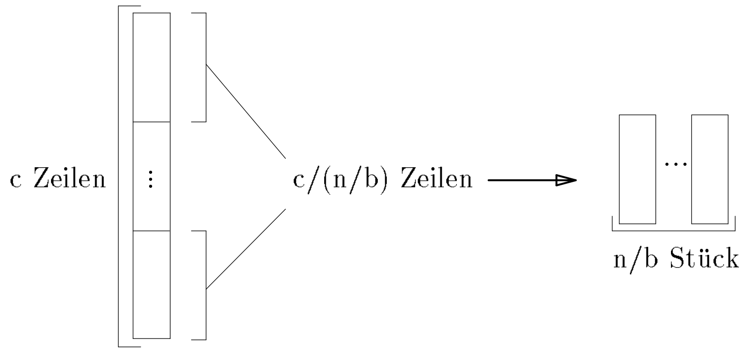

Abbildung 5: Aufteilung der Spalten von B in Abschnitte

nicht gebraucht und dienen nur dazu, die Matrix C gleich in einer kompakten Form als b-row-sparse Matrix auszugeben.

Nach dieser Berechnung stehen in den Summenregistern der einzelnen Prozessoren Teilskalarprodukte, die nun entsprechend den Blöcken der Datenstruktur für die Matrix B noch zeilenweise summiert werden müssen. Danach stehen in b verschiedenen Spalten des Prozessorfeldes die Ergebnisse, d. h. die Matrix C als b-row-sparse-Matrix, die nun noch nach links ausgegeben werden muß.

Abbildung 7 auf Seite 25 zeigt die Anordnung der Zwischenergebnisse im Prozessorfeld. In den mit Pfeilen markierten Spalten stehen am Schluß die einzelnen Spaltenvektoren der Matrix C.

Um die benötigte Zeilensumme in den einzelnen Blöcken auszurechnen, kann man ein Divide-und-Conquer-Verfahren zum Berechnen der Zeilensumme anwenden, das b-mal repliziert wird (siehe auch 2.7.4).

2.7.3 Analyse

Der Algorithmus aus 2.7.2 besteht im wesentlichen aus drei Schritten:

1. Multiplikation (danach sind die Ergebnisblöcke erzeugt),
2. blockweise Zeilensummen bilden und
3. Matrix C nach Westen ausgeben.

Durch die bessere Ausnutzung der dünn besetzten Struktur der Matrix B enthält die entsprechende Datenstruktur jetzt nur noch $\lceil \frac{c}{n/b} \rceil$ Zeilen. Das bedeutet, daß die Periode von Schritt 1 in $O(r \cdot \lceil \frac{c}{n/b} \rceil)$ liegt. Schritt 2 hat eine Periode, die in $O(\mathrm{ld}(n/b))$ liegt. Die Ausgabe in Schritt 3 hat Periode $O(n)$. Asymptotisch gesehen ist $\mathrm{ld}(n/b)$ kleiner als n. Das bedeutet, daß der durch die Summation verursachte zusätzliche Aufwand gegenüber der Ausgabe, die auf jeden Fall durchgeführt werden muß, um die Matrix C weiterverarbeiten zu können, vernachlässigt werden kann.

Insgesamt ergibt sich also durch Ausnutzung der 0-Spalten von B eine Verbesserung der Periode und somit auch der Laufzeit des ISA-Programms um einen konstanten Faktor.

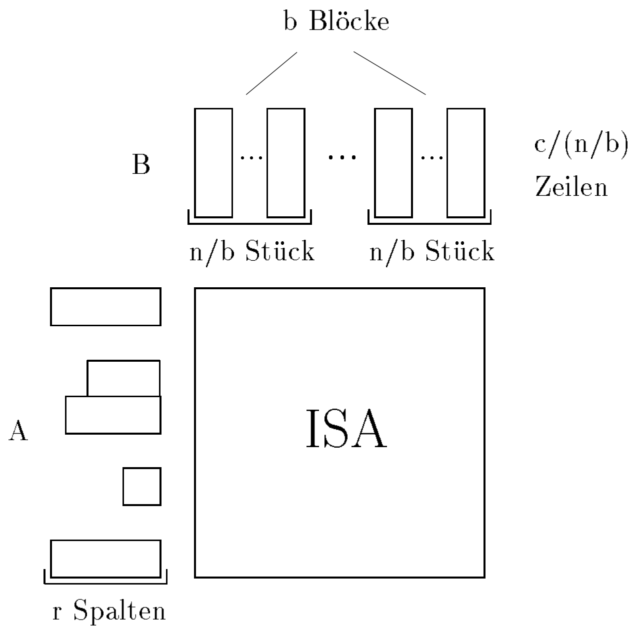

Abbildung 6: Eingabe der aufgeteilten Matrix B in das ISA

Falls sich eine Aufteilung einzelner Spalten von B in Abschnitte als zu schwierig erweist, ist es trotzdem möglich 0-Spalten auszunutzen, wenn B insgesamt mehr als n Spalten besitzt. Die den 0-Spalten der Matrix zugeordneten Spalten des ISA können in diesem Fall schon Spalten von B mit verrechnen, die ansonsten wegen der Partitionierung von B in einen anderen Block fallen würden. Das bedeutet, daß $B \in \mathbb{R}^{k \times m}$ in nur noch $\lceil b/n \rceil$ Blöcke anstatt von $\lceil m/n \rceil$ Blöcken aufgeteilt wird. Das hat wiederum zur Folge, daß die Gesamtrechenzeit des partitionierten Problems kleiner wird. Außerdem entfällt in diesem Fall auch der Summationsschritt zur Berechnung der Zeilensummen am Ende des Algorithmus.

Wenn zusätzlich zu den 0-Spalten von B auch noch 0-Zeilen in A existieren, können diese ganz analog behandelt werden. Der Summationsschritt am Schluß zerfällt dann in zwei Phasen. Zuerst werden wie gehabt blockweise Zeilensummen berechnet. Anschließend werden dann analog blockweise Spaltensummen berechnet. Die Programme dazu entstehen einfach durch Reflexion aus den bereits bekannten. Zusätzlich zu den 0-Spalten von B enthalte nun die Matrix A genau a Nichtnullzeilen. Dann ergibt sich für die Periode der einzelnen Schritte folgendes:

Multiplikation	$O(\lceil \frac{r}{n/a} \rceil \cdot \lceil \frac{c}{n/b} \rceil)$,
Summation	$O(\mathrm{ld}(n/a) + \mathrm{ld}(n/b)) = O(\mathrm{ld}(\frac{n^2}{a \cdot b}))$,
Ausgabe	$O(n)$.

Es sei angemerkt, daß in diesem Fall die Ergebnismatrix C höchstens ab Nichnulleinträge enthält. Außerdem ist C laut Korollar 2.8 rb-row-sparse und ac-column-sparse und man besitzt zusätzlich noch Informationen über 0-Zeilen und -Spalten.

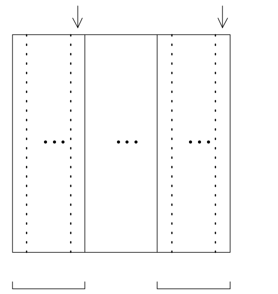

Abbildung 7: Summieren der Ergebnisse in den Blöcken und Ausgabe

2.7.4 Hinweise zur Umsetzung auf der Systola 1024

Der Pseudocode für das in 2.7.2 angegebene Verfahren zur blockweisen Berechnung der Zeilensumme soll hier in einer der Programmiersprache LAISA der Systola 1024 ähnlichen Notation angegeben werden. Das entsprechende ISA-Programm wurde jedoch nicht tatsächlich implementiert und getestet und taucht deshalb auch nicht in Abschnitt 3 auf.

```
procedure Send_sng(A, B,: Single; m: Natural; first, last: Natural);
begin(n)
  for i:=0 to 2 do begin
    < set A[i], C; 1^n; [first] size m >;
    < set CW, C; 1^n; [first+1 .. last] size m >;
    < set C, B[i]; 1^n; [last] size m >;
  end
end;
```

Diese Prozedur sendet eine Gleitkommazahl im IEEE-Single-Format von dem Single-Register `A` in Spalte `first` in das Single-Register in Spalte `last` eines `m` Spalten breiten Teils eines $n \times n$-ISA. Sie wird als Hilfsprozedur für die folgende Prozedur `Sum` benötigt.

```
procedure Sum(m: Natural);
begin(n)
  if m > 1 then begin
    replicate(1, 2, Sum(m/2));
    Send_sng(sum, temp, m, m/2, m);
    < addnrm_sng temp, sum, sum; 1^n; [m] size m >;
  end
end;
```

Diese Prozedur summiert zeilenweise die Gleitkommazahlen, die in den Single-Registern `sum` von `m` nebeneinanderliegenden Spalten eines $n \times n$-ISA liegen. Das Ergebnis steht zum Schluß im Single-Register `sum` der Prozessoren in der letzten Spalte des betrachteten `m` Spalten breiten Blockes. Es wird zusätzlich zu `sum` noch ein Hilfregister `temp` benötigt.

Um nun die für den Algorithmus aus 2.7.2 benötigte blockweise Zeilensumme zu berechnen, muß

```
replicate(1, b, Sum(n/b));
```

aufgerufen werden.

```
procedure Output;
begin
  Convert2IEEE_sng(n, sum);
  for i:= 1 to n/b do begin
    < sleep; 1^n; [1] size n >;
    OutVecShift(WEST, Spalten_index, 1^n, b-1);
    OutVecShift_sng(WEST, sum, 1^n, b-1);
    < awake; 1^n; [1] size n >;
    OutVecShift(WEST, Spalten_index, 1^n, 1);
    OutVecShift_sng(WEST, sum, 1^n, 1);
  end
end;
```

Diese Prozedur dient der Ausgabe der Ergebnisspalten von C, falls die den Algorithmus abschließenden Zeilensummen durch `Sum` berechnet wurden. Die Daten von C liegen nach Ablauf der Prozedur als komprimierte Datenstruktur in den Randprozessoren, d. h. C kann für folgende Operationen als b-row-sparse Matrix und sogar als Matrix mit 0-Spalten betrachtet werden, da in jeder Spalte der Datenstruktur die Spaltenindizes immer gleich sind.

2.8 Ein Algorithmus für das Problem CC

Der hier vorgestellte Algorithmus für die Multiplikation zweier column-sparse Matrizen ist eine direkte Umsetzung des entsprechenden Algorithmus aus [4] für ein ISA ohne rekonfigurierbare Busse.

Es wird die c_A-column-sparse Matrix A mit der c_B-column-sparse Matrix B multipliziert.

Ein zu diesem Algorithmus duales Verfahren läßt sich für das Problem RR angeben.

2.8.1 Idee des Algorithmus

Die Elemente b_{ij} der $n \times n$ Matrix B stehen zu Beginn in den Prozessoren des ISA. Jeder der Prozessoren kennt seinen Zeilen- und seinen Spaltenindex. Die Matrix A wird in einer `TColumnSparseMatrix`-Datenstruktur abgespeichert. Diese Datenstruktur wird in transponierter Form von links in das Prozessorfeld eingegeben (siehe Abbildung 8). Der Eintrag c_{ij} der Ergebnismatrix C soll am Schluß im Prozessor mit dem Index (i, j) stehen.

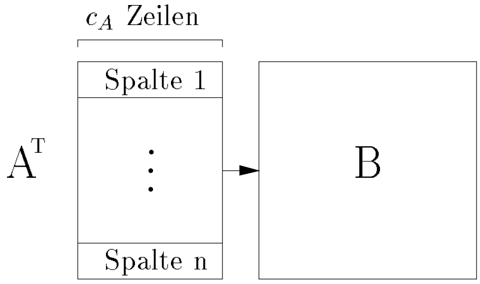

Abbildung 8: Eingabe der Matrix A für den Algorithmus CC

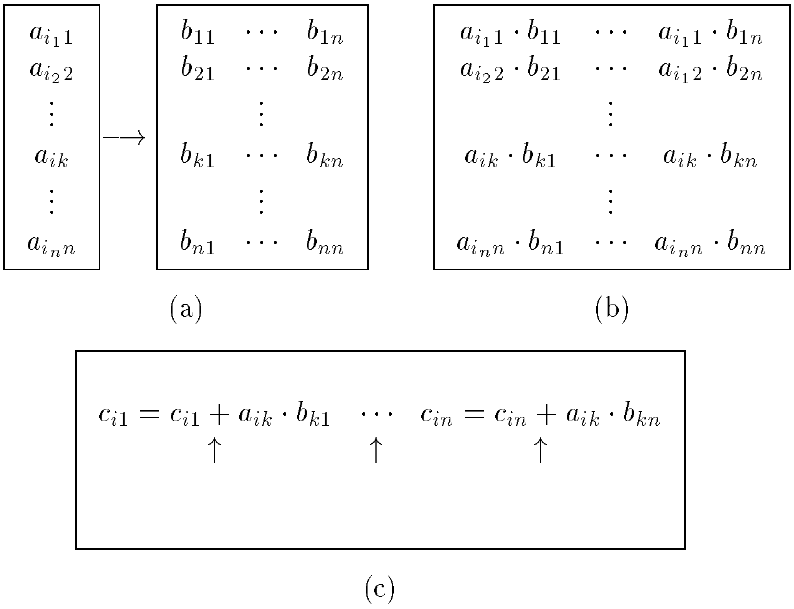

Abbildung 9: Verarbeitung einer Spalte von A durch CC.

Die Nichtnulleinträge der Matrix A^T werden mit ihrem Zeilenindex spaltenweise über das Prozessorfeld gesendet und in jedem Prozessor mit dem dortigen Element von B multipliziert. Abbildung 9 auf Seite 27 zeigt den Ablauf der Berechnung für eine Spalte von A. Sei a_{ik} das durch Zeile k des ISA gesendete Element von A (siehe (a)). Zunächst wird nun a_{ik} mit den lokalen Elementen b_{kj} in den Prozessoren der Zeile k multipliziert (siehe (b)). Danach müssen die Ergebnisse nun noch entlang der Spalten in die Zeile i transportiert werden und dort zu dem aktuellen Wert c_{ij} der Ergebnismatrix C addiert werden (siehe (c)). Dies muß allerdings mit den Produkten in allen Zeilen geschehen. Deshalb werden diese Zahlen zusammen mit dem Index i des entsprechenden Elementes a_{ik} von A durch n Ringschiebe-Operationen durch die Spalten des ISA transportiert.

2.8.2 Pseudocode

Für die Realisierung des Algorithmus wird neben der Operation `Zeilen_Senden_Eintrag` aus 2.4.2 eine Operation

`Ringschieben(s: Spaltenselektor; Index, Data: Register);`

benutzt, die jeweils ein Paar bestehend aus einem Index und einem Datum entlang der durch den Selektor `s` gewählten Spalten um eine Zeile rotieren. Die Werte aus den Registern (`Index`, `Data`) stehen nach der Rotation wieder in den gleichnamigen Registern des entsprechenden Nachbarprozessors. Abbildung 10 zeigt eine Ringschiebeoperation entlang einer Spalte. Das Register `Data` soll Gleitkommazahlen speichern können. Diese Funktionalität läßt sich mit Hilfe der Permutationsoperation `Ringschieben` aus [3] programmieren.

$$\boxed{\begin{array}{c}(i_1, d_1)\\ \vdots \\ (i_{n-1}, d_{n-1})\\ (i_n, d_n)\end{array}} \rightarrow \boxed{\begin{array}{c}(i_n, d_n)\\ (i_1, d_1)\\ \vdots \\ (i_{n-1}, d_{n-1})\end{array}}$$

Abbildung 10: Ergebnis einer Spaltenringschiebeoperation

Darüberhinaus soll jeder Prozessor seine Position innerhalb des ISA kennen. Der Pseudocode für den Algorithmus sieht damit wie folgt aus:

```
PROCEDURE MatMult(n, c_A);
BEGIN
  for i:=1 to c_A do
  BEGIN
    { Eine Spalte mit Indizes und Werten von A lesen }
    Zeilen_Senden_Eintrag(1^n, A_index, A_data);
    < temp := A_data * B_data; 1^n; 1^n >;

    { Ergebnisse in ihre Zeile transportieren }
    for j:=1 to n do
    BEGIN
      Ringschieben(1^n, A_index, temp);
      < A_index - P_i; 1^n; 1^n >;
      < if Zero then sum := sum + temp
                else sum := sum; 1^n; 1^n >;
    END
  END
END;
```

Die Register `A_index` und `A_data` speichern einen Eintrag der Matrix A. Das Register `B_data` speichert in jedem Prozessor das entsprechenden Element der

Matrix B. Das Register `P_i` enthält den Zeilenindex des jeweiligen Prozessors. In `sum` steht am Schluß ein Eintrag der Ergebnismatrix C. Das Register `temp` ist ein Hilfsregister. Die beiden letzteren sowie die Register `A_data` und `B_data` sind Gleitkommaregister.

2.8.3 Laufzeitanalyse

Die beiden Routinen `Zeilen_Senden_Eintrag` und `Ringschieben` haben eine konstante Periode. Dies gilt ebenso für alle anderen der Anweisungen. Damit liegt die Periode des Algorithmus also in $O(c_A n)$. Hierbei ist n die Kantenlänge des ISA. Allerdings können von dem Algorithmus nur $n \times n$ Matrizen verarbeitet werden. Das bedeutet, daß die Periode von der Größe der Matrizen abhängt. Sie ist sogar schlechter als die Periode $O(n)$ des Algorithmus aus [3], der die dünn besetzte Struktur der Matrizen außer Acht läßt.

2.9 Ein Algorithmus für das Problem WC

Middendorf et. al. haben in [4] einen Algorithmus für ein Prozessorfeld mit rekonfigurierbaren Bussen für die Multiplikation einer weakly-sparse Matrix A mit einer column-sparse Matrix B angegeben. Dieser zerlegt die Matrix A in zwei Teilmatrizen, die jeweils eine bestimmte Anzahl der Spalten von A enthalten. Eine dieser Teilmatrizen wird dann mittels des Algorithmus für CC mit B multipliziert. Aus diesem Grunde läßt sich der Algorithmus aus [4] nicht direkt für ein Prozessorfeld mit fester Verbindungsstruktur umsetzen, solange kein effizienter Algorithmus für CC existiert. Deshalb muß ein anderer Ansatz gewählt werden.

Dual zu dem Verfahren für WC existiert ein Algorithmus für das Verfahren RW. Dieser entsteht durch Spiegelung des hier vorgestellten Algorithmus an der Hauptdiagonalen des ISA.

2.9.1 Idee des Algorithmus

Die weakly-sparse Matrix $A \in \mathbb{R}^{n \times k}$ wird in der Datenstruktur aus 2.2 zeilenweise gespeichert und von links in das Prozessorfeld eingegeben. Die column-sparse Matrix $B \in \mathbb{R}^{k \times m}$ wird wie bei RC von oben in das $n \times m$ ISA eingegeben. Abbildung 11 zeigt diese Situation. Die n Zeilen von A sind hintereinander in w_A Spalten der Datenstruktur gespeichert.

Es wird nun weiterhin angenommen, daß in jeder Zeile der Datenstruktur für A Einträge aus höchstens z verschiedenen Zeilen stehen. Es gilt $z \leq w_A$. Der Algorithmus verlangt, daß jeder der Prozessoren genügend viele Register hat, um z Paare aus je einem Zeilenindex und einem Eintrag der Ergebnismatrix zu speichern.

Die grundlegende Funktionsweise des Algorithmus ist wie bei RC. Eine Spalte mit Indizes und Daten von A wird über das Feld verteilt. Dabei wird überprüft, ob der gelesene Zeilenindex ein anderer als der in der vorher gelesenen Spalte ist. Falls ja, wird ein neues Register für die Summanden des Skalarproduktes benutzt; anderenfalls wird das alte weiterbenutzt.

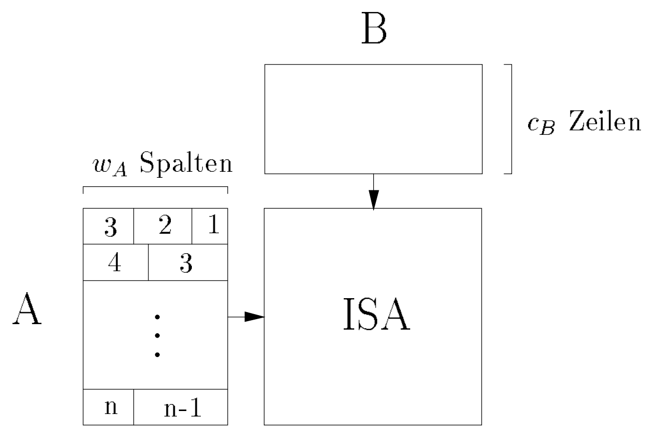

Abbildung 11: Eingabe der Operanden beim Algorithmus WC

Anschließend werden alle Zeilen der Datenstruktur für B gelesen, und falls der Zeilenindex des jeweils gelesenen B Elements mit dem Spaltenindex des aktuell im Prozessor gespeicherten A Elements übereinstimmt, wird ein Summand für ein Skalarprodukt berechnet und zu dem aktuellen Summenregister addiert.

Wenn alle Spalten der Datenstruktur für A betrachtet wurden, müssen die unterschiedlichen Skalarprodukte in den einzelnen Prozessoren noch in die richtige Zeile transportiert werden, was wie bei CC wieder durch Ringschiebeoperationen realisiert werden kann.

2.9.2 Pseudocode

Für die Realisierung des Algorithmus werden die Operationen

- `Zeilen_Senden_Eintrag`
- `Spalten_Senden_Eintrag`
- `Ringschieben`

aus 2.4.2 bzw. 2.8.2 benutzt. Dabei wird die Routine `Zeilen_Senden_Eintrag` derart modifiziert, daß sie *zwei* Indizes und einen Dateneintrag aus den westlichen Randprozessoren liest.

Desweiteren müssen die Prozessoren des ISA über die Fähigkeit zu indirekter Adressierung verfügen und jeder der Prozessoren muß seinen Index kennen.

Der Pseudocode für den Algorithmus ist dann folgendermaßen:

```
PROCEDURE MatMult(n, m, z, w_A, c_B);
BEGIN

  { Index und Summenregister initialisieren }
  < IndexReg := 1; 1^n; 1^m >;
  for i:=1 to z do
```

```
BEGIN
  < Row[IndexReg] := 0; 1^n; 1^m >;
  < Sum[IndexReg] := 0; 1^n; 1^m >;
  < IndexReg := IndexReg + 1; 1^n; 1^m >;
END;
< IndexReg := 1; 1^n; 1^m >;

{ Multiplikation ausfuehren }
for i:=1 to w_A do
BEGIN

  { Eine Spalte von A einlesen }
  Zeilen_Senden_Eintrag(1^n, A_row, A_col, A_data);

  { Neuer Zeilenindex ? }
  < Row[IndexReg] - A_row; 1^n; 1^m >;
  < NewIndexReg := IndexReg + 1; 1^n; 1^m >;

  { Register fuer indirekte Addressierung weiterzaehlen }
  < if Zero then IndexReg := IndexReg
            else IndexReg := NewIndexReg; 1^n; 1^m >;

  { Zeilenindex merken }
  < if Zero then A_row := A_row
            else Row[IndexReg] := A_row; 1^n; 1^m >;

  for j:=1 to c_B do
  BEGIN

    { Eine Zeile von B einlesen }
    Spalten_Senden_Eintrag(1^m, B_index, B_data);

    { Skalarprodukt weiterrechnen }
    < temp := Sum[IndexReg] + A_data * B_data; 1^n; 1^m >;
    < A_col - B_index; 1^n; 1^m >;
    < if Zero then Sum[IndexReg] := temp
              else temp := temp; 1^n; 1^m >;
  END
END;

{ Ergebnisse in ihre Zeile transportieren }
< sum := 0; 1^n; 1^m >;
< IndexReg := 1; 1^n; 1^m >;
for i:=1 to z do
BEGIN
  for j:=1 to n do
  BEGIN
```

```
      Ringschieben(1^n, Row[IndexReg], Sum[IndexReg]);
      < Row[IndexReg] - P_i; 1^n; 1^m >;
      < if Zero then sum := Sum[IndexReg]
                else sum := sum; 1^n; 1^m >;
    END;
    < IndexReg := IndexReg + 1; 1^n; 1^m >;
  END
END;
```

Die Bedeutung der einzelnen Register kann der folgenden Tabelle 2 entnommen werden.

Register	Gleitkomma	Bedeutung
`IndexReg`	nein	dient zur Indizierung der unterschiedlichen Summenregister bzw. Register zur Speicherung der Zeilenindizes für die Ergebnismatrix C
`Row[]`	nein	speichern bis zu z Zeilenindizes der Teilskalarprodukte in den `Sum[]`-Registern
`Sum[]`	ja	Summenregister für bis zu z Ergebnisse von Skalarprodukten, die bei der Verarbeitung einer Zeile der Datenstruktur für A auftreten können
`A_row`	nein	speichert den Zeilenindex des in `A_data` gespeicherten Elementes von A
`A_col`	nein	Spaltenindex des A-Elementes in `A_data`
`A_data`	ja	speichert ein Element der Matrix A
`B_index`	nein	speichert den Zeilenindex des im Prozessor gespeicherten Elementes der Matrix B
`B_data`	ja	enthält ein Element der Matrix B
`temp`	ja	Hilfsregister zur Speicherung von Zwischenergebnissen
`sum`	ja	enthält nach dem Ablauf des Algorithmus einen Eintrag der Ergebnismatrix C
`P_i`	nein	speichert den Zeilenindex des Prozessors

Tabelle 2: Bedeutung der Register bei WC

2.9.3 Laufzeitanalyse

Die in dem Algorithmus verwendeten Routinen bzw. Anweisungen haben sämtlich konstante Perioden. Das bedeutet die Gesamtperiode des Algorithmus ergibt sich aus der Betrachtung der `for`-Schleifen. Damit gilt für die Gesamtperiode $T(n, z, w_A, c_b)$ des Algorithmus

$$T(n, z, w_A, c_b) \leq \alpha(z + w_A c_B + zn) \in O(w_A c_B + zn)$$

für eine Konstante $\alpha > 0$. Zieht man in Betracht, daß $z \leq w_A$ gilt, so folgt daß

$$T(n, z, w_A, c_B) \in O(w_A \max\{c_B, n\}).$$

Die Gesamtperiode ist also wie bei CC von der Kantenlänge des ISA abhängig. Im Gegensatz zu CC kann dieser Algorithmus aber auch Matrizen miteinander verknüpfen, die keine $n \times n$ Matrizen sind. Genauer gesagt ist $A \in \mathbb{R}^{n \times k}$, und die Anzahl der Spalten kann wesentlich größer sein als die Anzahl der Zeilen ($k \gg n$). Damit stellt die Periode von $O(w_A \max\{c_B, n\})$ also eine Verbesserung gegenüber der Periode $O(k)$ der Matrixmultiplikation ohne Berücksichtigung der dünn besetzten Struktur der Operanden dar, da auch $c_B \ll k$ gilt.

2.9.4 Partitionierung von WC

Der Algorithmus für WC kann ähnlich wie bei RC (siehe 2.6.1) partitioniert werden, wenn die Größe der Operandenmatrizen die Kantenlänge des verwendeten ISA übersteigt.

Interessant ist darüberhinaus noch eine andere Möglichkeit der Partition. Der Algorithmus für WC verlangt, daß genügend viele Register in jedem Prozessor vorhanden sind, um die Einträge aus maximal $z \leq w_A$ Zeilen behandeln zu können. Die Anzahl der Register in den Prozessoren ist jedoch verhältnismäßig begrenzt. Beispielsweise hat die Systola 1024 nur 10 Single-Register, die sich überdies noch mit den „normalen“ 16-Bit-Registern überschneiden. Der Algorithmus benötigt aber z Gleitkomma- und zusätzlich noch z Register für die entsprechenden Zeilenindizes. Hinzu kommen die anderen vom Algorithmus verwendeten Register (siehe Tabelle 2). Würde der Algorithmus auf der Systola 1024 implementiert, könnten somit höchstens weakly-sparse Matrizen A mit $z \leq 3$, also bei ungünstiger Verteilung der Einträge $w_A \leq 3$ behandelt werden. Deshalb ist es sinnvoll das Verfahren für WC so zu partitionieren, daß auch Matrizen mit mehr Einträgen als die Registeranzahl der Prozessoren erlaubt, multipliziert werden können.

Für diese Partitionierung wird die folgende Eigenschaft des Skalarproduktes benutzt. Seien $a_1, a_2, b \in \mathbb{R}^k$ Vektoren. Dann gilt wegen Linearität:

$$(a_1 + a_2)^T \cdot b = a_1^T \cdot b + a_2^T \cdot b.$$

Das heißt, daß für die Matrizenprodukt gilt:

$$A = A_1 + A_2 \Rightarrow A \cdot B = (A_1 + A_2) \cdot B = A_1 \cdot B + A_2 \cdot B.$$

Die Einträge in den Zeilen von A werden gewissermaßen auf A_1 und A_2 verteilt. Wegen der Linearität des Skalarprodukts folgt dann die Eigenschaft.

Für die Partitionierung zerlegt man also einfach die Datenstruktur für die Matrix A entlang der Zeilen in hinreichend kleine Blöcke, die dann hintereinander behandelt werden. Abbildung 12 zeigt diese Situation.

Angenommen das die Datenstruktur in a Blöcke zerlegt wird. Der Algorithmus wird also a mal auf eine w_A/a-weakly-sparse und eine c_B-columns-sparse Matrix angewendet. Die Laufzeit des Verfahrens ist also in

$$O\left(\left(\frac{w_A}{a} \max\{c_B, n\}\right) \cdot a\right) = O(w_A \max\{c_B, n\})$$

und bleibt somit unverändert.

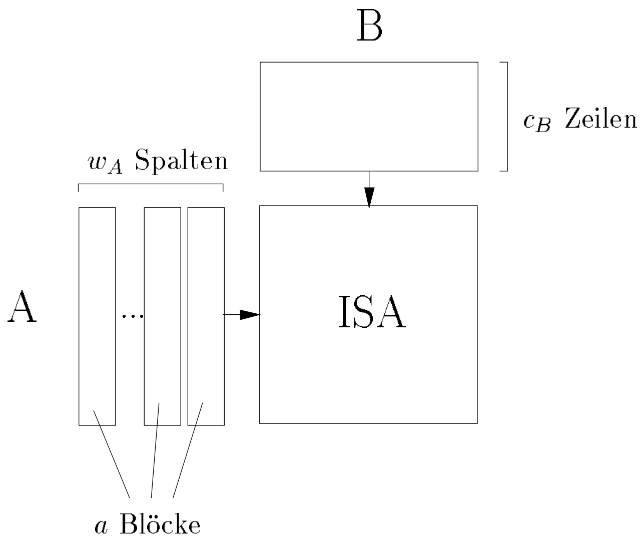

Abbildung 12: Partitionierung des Verfahrens für WC

Auf diese Weise läßt sich das Verfahren auch unabhängig von der Anzahl der Prozessorregister machen, indem die Datenstruktur in w_A Blöcke mit je einer Spalte zerlegt wird. In diesem Fall ist zusätzlich auch keine indirekte Adressierung erforderlich. Allerdings werden dann nach der Berechnung *jedes* Summanden n Ringschiebe-Operationen, also insgesamt $w_A n$, ausgeführt. Der unpartitionierte Algorithmus benötigt hingegen höchstens zn Ringschiebe-Operationen. Falls also in der Datenstruktur für A keine Zeile mit Einträgen aus w_A unterschiedlichen Zeilen der Ausgangsmatrix existiert, benötigt der unpartitionierte Algorithmus weniger Operationen. Diese Ersparnis erkauft man allerdings damit, daß die Größe der Matrix A mit der Anzahl der Prozessoren zusammenhängt.

3 Implementation auf der Systola 1024

In diesem Abschnitt wird die Implementation des Algorithmus RC aus 2.4 auf der Systola 1024 beschrieben. Die Datenstruktur aus 2.2 wird in PASCAL realisiert, und der Pseudocode für das ISA-Programm wird in ein lauffähiges LAISA-Programm umgesetzt.

3.1 Grundlagen

Die Systola 1024 ist ein 32×32 ISA, das auf einer Steckkarte für IBM kompatible PC realisiert ist. Die Kommunikation mit der Umgebung erfolgt über Arbeitsspeicher, der auf der Karte untergebracht ist, das sogenannte Karten-RAM. Dieses ist in ein westliches und ein nördliches Karten-RAM von jeweils 512K Größe eingeteilt. Jeder dieser beiden Teile ist seinerseits wieder in 256 Blöcke mit jeweils 1024 16-Bit-Worten eingeteilt. Außerdem besitzt jeder der Randprozessoren zwei Speicherbänke, die jeweils 128 Worte zu je 16 Bit speichern können. Jeweils eine dieser Bänke kommuniziert mit dem ISA und eine mit dem Karten-RAM. Die Programmierung des Feldes erfolgt auf drei Ebenen:

1. Das ISA-Programm, welches Daten über die Randprozessoren in das Feld einliest, die Berechnung durchführt und die Ergebnisse wieder über die Randprozessoren an die Umgebung überträgt, wird in der PASCAL ähnlichen Programmiersprache LAISA beschrieben.

2. Es existiert ein programmierbarer Controller, der den Datentransfer zwischen dem Karten-RAM und den Randprozessorbänken steuert. Außerdem werden die LAISA-Programme vom Controller gestartet.

3. Über eine hochsprachliche Schnittstelle ist der Datentransfer zwischen dem RAM des PC und dem Karten-RAM realisiert. Über diese Schnittstelle können auch Controllerprogramme gestartet werden. Die Schnittstelle existiert für PASCAL und C/C++.

In dieser Arbeit wird die Programmierschnittstelle für BORLAND PASCAL benutzt. Weitergehende Angaben zu Aufbau und Programmierung der Systola 1024 finden sich in [1].

3.2 Umsetzung der Datenstruktur

Die Datenelemente der Matrizen werden durch Gleitkommazahlen im IEEE-Single-Format realisiert.

Die Datenstruktur für die dünn besetzten Matrizen aus 2.2 wird durch die Klasse `TSparseMat_sng` implementiert. Die Umsetzung mit Hilfe objektorientierter Programmierung wurde lediglich gewählt, um die Daten und die nötigen Operationen (z. B. Laden des Karten-RAM oder Initialisierung der Datenstruktur) übersichtlicht zusammenfassen zu können.

Die Klassen- und sonstigen Definitionen befinden sich in der Unit `Sparse`, deren Dokumentation folgt. Die PASCAL-Quelltexte sind in Abschnitt 5 aufgeführt.

3.2.1 Konstanten

Deklaration `n = 32;`

Die Konstante `n` gibt die Größe des quadratischen Prozessorfeldes an. Für die Systola 1024 ist dieser Wert 32.

Deklaration `MaxSize_sng = 248;`

Diese Konstante gibt an, wieviele Einträge bei einer ISA-Größe von `n` in jeder Zeile bzw. Spalte eines `TSparseMat_sng`-Objekts gespeichert werden können. Da jeder Eintrag eine Größe von 8 Byte hat (siehe `TEntry_sng`) und die Größe einer Datenstruktur in dem verwendeten BOPRLAND PASCAL für PC auf 65520 Bytes beschränkt ist, ergibt sich dieser Wert, der zusätzlich wegen der Verwendung von `load_block_pitch` in `TSparseMat_sng.Load_Matrix_West` auf ein Vielfaches von 8 abgerundet ist.

Deklaration `blocksize = 1024;`

`blocksize` gibt die Größe eines Speicherblocks des Karten-RAM in Worten an.

Deklaration `SizeOf_TEntry_sng = SizeOf(TEntry_sng) div 2;`

Diese Konstante gibt die Größe eines `TEntry_sng`-Records in Worten an. Sie dient der kompakteren Schreibweise, da bei den Routinen der ISA-Programmierschnittstelle die Größe von Daten im Gegensatz zu PASCAL immer in 16-Bit-Worten statt in Bytes angegeben werden muß.

Deklaration `Row_Entry_cnt_sng = 8;`

`Row_Entry_cnt_sng` gibt die Anzahl hintereinander stehender Einträge an, die pro Zeile einer row-sparse bzw. pro Spalte einer column-sparse-Matrix aus der Datenstruktur in *einen* Block des Karten-RAM kopiert werden (siehe auch `TSparseMat_sng.Load_Matrix_West` bzw. `TSparseMat_sng.Load_Matrix_North`).

3.2.2 Datentypen

Deklaration
```
PMat_sng = ^TMat_sng;
TMat_sng = Array [1..n, 1..n] of Single;
```

Dieser Datentyp dient dem Speichern einer dicht besetzten $n \times n$ Matrix. `TMat_sng` wird für die Ergebnisse der Matrixmultiplikationen benutzt.

Deklaration
```
TEntry_sng = record
               index: Word;
               dummy: Word;
               data: Single;
             end;
```

`TEntry_sng` stellt einen Record zur Speicherung eines Eintrags einer dünn besetzten Matrix bereit. `index` speichert für row-sparse-Matrizen die Position der Zahl `data` innerhalb der betrachteten Zeile der Ausgangsmatrix. Analog ist `index` eine Spaltenposition bei column-sparse-Matrizen. `dummy` dient dazu die Größe des Records derart anzupassen, daß die Blöcke des Karten-RAM des ISA immer vollständig gefüllt sind, wenn pro Zeile bzw. Spalte der Datenstruktur `Row_Entry_cnt_sng` Einträge hintereinander in dem Block stehen.

Deklaration
```
PRowMat_Array_sng = ^TRowMat_Array_sng;
TRowMat_Array_sng = Array [1..n, 1..MaxSize_sng]
of TEntry_sng;
```

Dieser Datentyp stellt ein Array zur Speicherung der Daten einer row-sparse-Matrix zur Verfügung. Es können maximal `n` Zeilen mit je höchsten `MaxSize_sng` Einträgen gespeichert werden. Der Datentyp wird von der Klasse `TSparseMat_sng` verwendet. Um eine Matrix zu speichern, sollte stets die Klasse verwendet werden, da nur dort die Operationen zur Manipulation der Matrix zur Verfügung stehen und die Daten richtig initialisiert werden.

Deklaration
```
PColumnMat_Array_sng = ^TColumnMat_Array_sng;
TColumnMat_Array_sng = Array [1..MaxSize_sng,
1..n] of TEntry_sng;
```

Dieser Datentyp stellt ein Array zur Speicherung der Daten einer column-sparse-Matrix zur Verfügung. Es können maximal `n` Spalten mit je höchstens `MaxSize_sng` Einträgen gespeichert werden. Im Übrigen gilt das bereits für den vorigen Datentyp gesagte.

3.2.3 Die Klasse `TSparseMat_sng`

Konstruktor

Deklaration `constructor Init(ASparse_Param: Word; ARowSparse: Boolean);`

Der Konstruktor erzeugt ein neues `TSparseMat_sng`-Objekt. Die Attribute `Sparse_Param` und `RowSparse` werden auf die gegebenen Werte gesetzt. Falls `ASparse_Param` größer als `MaxSize_sng` ist, wird ein Laufzeitfehler erzeugt und das Programm beendet.

Danach wird Speicher für die zu speichernde Matrix alloziert. Falls `ARowSparse` `TRUE` ist, ist dieser Speicher über den Zeiger `a` zugreifbar, ansonsten über den Zeiger `b`. Der jeweils andere der beiden Zeiger wird auf `NIL` gesetzt.

Zuletzt wird die gesamte Matrix mit 0 gefüllt.

Destruktor

Deklaration `destructor Done;`

Der Desktruktor zerstört ein `TSparseMat_sng`-Objekt. Der von `a^` bzw. `b^` belegte Speicher wird freigegeben.

Attribute

Deklaration `a: PRowMat_Array_sng;`

Deklaration `b: PColumnMat_Array_sng;`

Diese Attribute sind Zeiger auf die Daten der von dem `TSparseMat_sng`-Objekt gespeicherten Matrix. Falls `RowSparse` `TRUE` ist, ist dies eine row-sparse-Matrix und die Daten stehen in `a^`. Anderenfalls ist dies eine column-sparse-Matrix und die Daten stehen in `b^`. Der jeweils andere der beiden Zeiger ist `NIL`.

Die beiden Zeiger sind global (von außerhalb des Objektes) zugreifbar, um einen schnellen Zugriff auf die Daten der gespeicherten Matrix zu gewährleisten. Trotzdem sollte der Zugriff über die dafür vorgesehenen Methoden `SetData` bzw. `GetData` erfolgen, da diese zusätzlich die Gültigkeit der eingegebenen Daten prüfen und ggf. einen Laufzeitfehler generieren.

Deklaration `Sparse_Param: Word;`

Dieses Attribut gibt an, wieviele Einträge jede Zeile der von dem `TSparseMat_sng`-Objekt gespeicherten row-sparse-Matrix hat, falls `RowSparse` `TRUE` ist. Anderenfalls bezeichnet dieser Wert die Anzahl der Einträge jeder Spalte der gespeicherten column-sparse-Matrix.

Deklaration `RowSparse: Boolean;`

`RowSparse` gibt an, ob von dem `TSparseMat_sng`-Objekt eine row-sparse oder column-sparse-Matrix gespeichert wird. Falls der Wert des Attributes `TRUE` ist, handelt es sich um eine row-sparse, anderenfalls um eine column-sparse-Matrix.

Methoden

Deklaration `procedure SetData(i, j: Word; AIndex: Word; AData: Single);`

Diese Methode dient dem Eintragen eines Wertes in die von dem `TSparseMat_sng`-Objekt gespeicherte Matrix. Die Indizes `i` und `j` geben die Position des gewünschten Eintrags innerhalb des Arrays `a^` bzw. `b^` an. `AIndex` und `AData` sind die Daten mit denen der Eintrag, der vom Typ `TEntry_sng` ist, belegt wird.

Zuvor werden die Indizes `i` und `j` über einen Aufruf von `RangeCheck` auf ihre Gültigkeit hin überprüft.

Deklaration `procedure GetData(i, j: Word; var AIndex: Word; var AData: Single);`

`GetData` bildet das Gegenstück zu `SetData`. Die Methode liest den Eintrag an der Position `[i,j]` des Arrays `a^` bzw. `b^` aus und gibt die Daten des Eintrags in den Variablen `AIndex` und `AData` zurück.

Wiederum werden die beiden Indizes `i` und `j` durch Aufruf der Methode `RangeCheck` auf ihre Gültigkeit hin überprüft.

Deklaration `function isRowSparse: Boolean;`

Diese Methode liefert `TRUE` genau dann, wenn die von dem `TSparseMat_sng`-Objekt gespeicherte Matrix eine row-sparse-Matrix ist.

Deklaration `function isColumnSparse: Boolean;`

Diese Methode liefert `TRUE` genau dann, wenn die von dem `TSparseMat_sng`-Objekt gespeicherte Matrix eine column-sparse-Matrix ist.

Deklaration `procedure Print(var F: Text; PureData: Boolean);`

Diese Methode gibt den Inhalt der von dem `TSparseMat_sng`-Objekt gespeicherten Matrix in die Textdatei `F` aus. Die Daten werden für row-sparse-Matrizen zeilenweise und für column-sparse-Matrizen spaltenweise geschrieben. Die Einträge werden im Format

```
Index: <index>    Data: <data>
```

geschrieben. Falls `PureData` `TRUE` ist, wird kein weiterer Text ausgegeben, sondern nur die Daten im Format

```
<index> <data>
```

Deklaration `procedure Read(var F: Text);`

Diese Methode liest die Daten für die von dem `TSparseMat_sng`-Objekt zu speichernde Matrix aus der Textdatei `F` ein. Die Daten müssen dabei in einem Format vorliegen, das dem mit `Print(F, TRUE)`[11] erzeugten gleicht.

Deklaration `procedure Random(dim: Word);`

[11] d. h. es wurden nur die Daten geschrieben.

Die Methode `Random` füllt die von dem `TSparseMat_sng`-Objekt gespeicherte Matrix mit Zufallswerten. Dabei gibt `dim` für row-sparse-Matrizen die Anzahl der Elemente in einer Zeile (incl. der Nullelemente), für column-sparse-Matrizen die Anzahl der Elemente in einer Spalte an. Das bedeutet, daß von dem verwendeten Zufallsgenerator für die Einträge in die Arrays `a^` bzw. `b^` Indizes zwischen 1 und dem Wert von `dim` erzeugt werden. Aus diesem Grund muß `dim` größer als `Sparse_Param` sein, da die Einträge jeder Zeile von `a^` bzw. jeder Spalte von `b^` verschieden voneinander sein müssen.

Die erzeugten Daten sind Single-Zahlen zwischen -100 und $+100$.

Deklaration `procedure Load_Matrix_West(startblock: Byte);`

Diese Methode lädt die von `TSparseMat_sng`-Objekt gespeicherte row-sparse-Matrix in das westliche Karten-RAM der Systola 1024. Falls die Matrix des Objekts nicht row-sparse ist, tut die Methode gar nichts. Anderenfalls wird das Array `a^` beginnend mit dem Speicherblock `startblock` in insgesamt `BlockCount` hintereinanderliegende Blöcke kopiert. Dabei stehen in jedem Speicherblock immer `Row_Entry_cnt_sng` hintereinanderstehende Einträge aus jeder Zeile des Arrays. Das Array `a^` verteilt sich also wie folgt über die einzelnen Blöcke:

`a^[1,1]` ... `a^[1,8]`		`a^[1,k-7]` ... `a^[1,k]`
⋮	...	⋮
`a^[32,1]` ... `a^[32,8]`		`a^[32,k-7]` ... `a^[32,k]`

Tabelle 3: Aufteilung des Arrays `a^` auf die Speicherblöcke. Der erste Block hat die Nummer `startblock`, der letzte die Nummer `startblock + BlockCount - 1`.

Der Wert `k` entspricht dem des Attributes `Sparse_Param`. Wenn der letzte Block nicht vollständig gefüllt ist, wird nach hinten mit 0 aufgefüllt.

Deklaration `procedure Load_Matrix_North(startblock: Byte);`

Diese Methode lädt die von `TSparseMat_sng`-Objekt gespeicherte column-sparse-Matrix in das nördliche Karten-RAM der Systola 1024. Falls die Matrix des Objekts nicht column-sparse ist, tut die Methode gar nichts. Anderenfalls wird das Array `b^` beginnend mit dem Speicherblock `startblock` in insgesamt `BlockCount` hintereinanderliegende Blöcke kopiert. Dabei stehen in jedem Speicherblock immer `Row_Entry_cnt_sng` Zeilen mit je 32 hintereinanderstehenden Einträgen. Das Array `b^` verteilt sich also wie folgt über die einzelnen Blöcke:

Der Wert `k` entspricht dem des Attributes `Sparse_Param`. Wenn der letzte Block nicht vollständig gefüllt ist, wird nach unten mit 0 aufgefüllt.

Deklaration `procedure Load_Matrix(startblock: Byte);`

Diese Methode lädt die von dem `TSparseMat_sng`-Objekt dargestellte Matrix in das Karten-RAM der Systola 1024. Dabei werden row-sparse-Matrizen in das westliche und column-sparse-Matrizen in das nördliche Karten-RAM geladen.

`b^[1,1]` ... `b^[1,32]`
⋮
`b^[8,1]` ... `b^[8,32]`
	...	`b^[k-7,1]` ... `b^[k-7,32]`
		⋮
		`b^[k,1]` ... `b^[k,32]`

Tabelle 4: Aufteilung des Arrays `b^` auf die Speicherblöcke. Der erste Block hat die Nummer `startblock`, der letzte die Nummer `startblock + BlockCount - 1`.

Die Matrix wird beginned mit dem Block `startblock` fortlaufend in das Karten-RAM kopiert. Zur Aufteilung der Matrix auf die verwendeten Blöcke siehe `Load_Matrix_West` und `Load_Matrix_North`.

Deklaration `function BlockCount: Byte;`

Die Methode `BlockCount` gibt die Anzahl der Speicherblöcke des RAM der ISA-Karte zurück, die benötigt werden, um die von dem `TSparseMat_sng`-Objekt dargestellte Matrix dort zu speichern.

Deklaration `procedure RangeCheck(i, j: Word);`

Diese Methode überprüft, ob die durch die Indizes `i` und `j` gegebene Position für die gespeicherte Matrix gültig ist. Falls eine Bereichsüberschreitung vorliegt, wird der entsprechende Laufzeitfehler generiert und das Programm abgebrochen.

3.2.4 Prozeduren

Deklaration `procedure StoreMat_sng(c: PMat_sng);`

Diese Routine liest eine 32×32 Matrix aus dem westlichen RAM der ISA-Karte in das Array `c^`. Dazu wird die Prozedur `store_matrix` der Programmierschnittstelle des ISA benutzt.

Deklaration `procedure PrintMat_sng(var F: Text; c: PMat_sng);`

Die Routine `PrintMat_sng` gibt die dichtbesetzte Matrix, die in dem Array `c^` gespeichert ist, in die Textdatei `F` aus. Die Matrix wird zeilenweise ausgegeben.

3.3 Umsetzung des Algorithmus

Die Umsetzung des Algorithmus geschieht auf der Ebene des Controller-Programms und auf der Ebene des ISA-Programm.

Da keine Kommunikation zwischen dem PASCAL-Programm und dem ISA-Programm möglich ist, die Anzahl besetzter Zeilen bzw. Spalten der Matrizen aber vom Nutzer über das PASCAL-Programm gesetzt werden kann (siehe 3.2), muß der entsprechende Teil der Ablaufsteuerung vom Controllerprogramm übernommen werden. Dazu sind zwei Möglichkeiten denkbar:

1. Der Controller übernimmt den Kontrollfluß komplett. Das bedeutet, daß ISA-Programme existieren, die *eine* Spalte der westlichen Matrix einlesen

bzw. *eine* Zeile der nördlichen Matrix einlesen und *einen* Skalarproduktschritt durchführen. Diese ISA-Programme werden dann vom Controllerprogramm so oft aufgerufen, wie nötig ist, um die Spalten bzw. Zeilen gemäß dem Algorithmus aus 2.4 zusammenzuführen. Vorteil dieser Methode ist, daß stets nur soviele Spalten und Zeilen miteinander verrechnet werden, wie tatsächlich nötig ist. Nachteil ist, daß der gesamte Kontrollfluß beim Controllerprogramm liegt, welches nun nicht mehr parallel schon die nächsten zu verarbeitenden Blöcke in die Randprozessorbänke laden kann.

2. Der Controller sorgt lediglich dafür, daß die Daten korrekt in die Randprozessorbänke geladen werden. Es existiert *ein* ISA-Programm, das bei jedem Aufruf eine feste Anzahl k der Spalten der westlichen Matrix mit k Zeilen der nördlichen Matrix verrechnet. Der Vorteil dieser Methode liegt darin, daß während der Berechnung der Controller schon die nächsten Daten parallel laden kann. Der Nachteil ist, daß immer ein paar Berechnungschritte zuviel durchgeführt werden, da die tatsächlich vorhandenen Anzahl der Spalten/Zeilen r bzw. c nicht durch k teilbar sein muß.

Die zweite Variante hat sich während des Testens als die Schnellere erwiesen und wird deshalb hier implementiert.

3.3.1 Das Controllerprogramm

Die Aufgabe des Controllerprogramms ist es, die Randprozessoren mit Daten zu versorgen und das ISA-Programm für die Berechnung zu starten. Da in eine Randprozessorbank höchstens der Inhalt von 4 Blöcken des entsprechenden Karten-RAM paßt, muß das Controllerprogramm dafür sorgen, daß nacheinander die verschiedenen Blöcke in die Berechnung einbezogen werden.

Da der Speicher für die ISA-Programm-Diagonalen begrenzt ist, ist es sogar nur möglich, pro Rand des ISA jeweils den Inhalt zweier Blöcke zu verrechnen. Ansonsten ist das ISA-Programm zu groß.

Das Controllerprogramm muß den Inhalt jedes verwendeten westlichen Blokkes mit dem Inhalt jedes verwendeten nördlichen Blockes zusammenführen. Vom PASCAL-Programm wird dem Controllerprogramm mitgeteilt in welchen hintereinanderliegenden Blöcken die Daten der beiden zu multiplizierenden Matrizen stehen. `blkst_w` ist der erste Block, `blkend_w-1` der letzte Block, der Daten der row-sparse-Matrix im westlichen Karten-RAM enthält. Analog für `blkst_n`, `blkend_n-1` und das nördliche Karten-RAM.

Beim Laden eines Speicherblockes vom Karten-RAM in die Randprozessorbänke muß das Controllerprogramm am westlichen Rand des ISA immer 8 Einträge – dies entspricht 32 Worten – für jede Prozessorzeile kopieren. Am nördlichen Rand muß achtmal immer 1 Eintrag – dies entspricht 4 Worten – für jede Prozessorspalte kopiert werden. Abbildung 13 zeigt, wie die jeweils ersten Speicherblöcke der Operandenmatrizen aus dem Karten-RAM in die Randprozessorbänke kopiert werden (vlg. auch Tabellen 3 und 4 auf Seite 39f).

Es wird jeweils mindestens ein Block des westlichen und nördlichen Karten-RAM, der komplett mit 0 gefüllt sein muß, als Füllblock benötigt, da immer eine

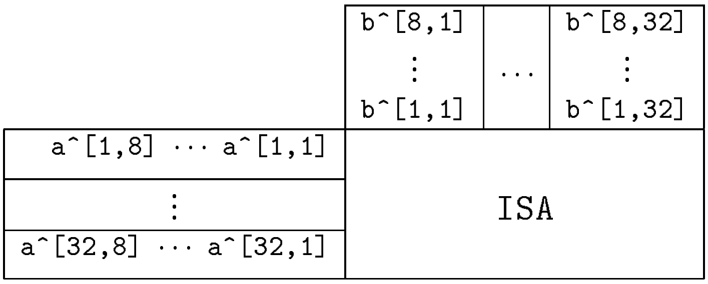

Abbildung 13: Aufteilung der ersten Speicherblöcke auf die Randprozessorbänke

feste Anzahl von Spalten/Zeilen von dem ISA-Programm verrechnet werden. Es wird davon ausgegangen, daß dies der Block 0 ist.

Der Pseudocode für das Controllerprogramm sieht dann wie folgt aus.[12]

```
procedure ControllerProg(blkst_w, blkend_w, blkst_n, blkend_n: Byte);

 procedure Lade_Bloecke_Westen(var i: Byte);
 Begin
   Lade_Block(i) mit 32 Worten pro Prozessor;
   Inc(i);
   if i = blkend_w then Lade_Block(0)
   else Begin
     Lade_Block(i) mit 32 Worten pro Proz.;
     Inc(i)
   End
   Schalte Westbaenke um;
 End;

 procedure Lade_Bloecke_Norden(var j: Byte);
 Begin
   Lade_Block(j) mit 4 Worten pro Prozessor;
   Inc(j);
   if j = blkend_n then Lade_Block(0)
   else Begin
     Lade_Block(j) mit 4 Worten pro Prozessor;
     Inc(j)
   End
 End;

Begin
  Initialisiere ISA;
  ISA_START SET0_SNG;             { Skalarproduktregister auf 0 setzen }

  b_w := blkst_w;
```

[12] Der Quelltext des Controllerprogramms befindet sich in Abschnitt 5.

```
  while (b_w < blkend_w) do Begin          { alle westl. Bloecke }

    Lade_Bloecke_Westen(b_w);

    b_n := blkst_n;
    while (b_n < blkend_n)                 { alle noerdl. Bloecke }

      Lade_Bloecke_Norden(b_n);
      Setze Westbaenke zurueck;
      Schalte Nordbaenke um;
      Warte bis Bank-Befehl das ISA durchlaufen hat;

      ISA_START MAT2_SNG       { ISA-Programm zur Multiplikation des }
                               { des Inhalts zweier Bloecke }
    End
  End;

  { Ausgabe }
  ISA_START Out_SNG;       { Ausgabe der berechneten Matrix amn den }
                           { westl. Rand }
  Schreibe Bloecke 0 und 1 im Westen
End;
```

3.3.2 Die ISA-Programme

Es werden insgesamt 8 verschiedene ISA-Programme benutzt, die im folgenden dokumentiert sind. Die von allen diesen Programmen verwendete Include-Datei `con_sng.inc` enthält alle gemeinsam verwendeten Konstanten.

Von den ISA-Programmen wird die mit der Systola 1024 gelieferte Bibliothek `FPSNG.LIB` zur Bearbeitung von Single-Zahlen mit dem ISA verwendet.

Die Include-Datei `con_sng.inc`

Deklaration `n = 32;`

Diese Konstante gibt, wie bereits bei der PASCAL-Unit `Sparse` die Größe des quadratischen Prozessorfeldes an.

Deklaration `sum = S[4];`

Dieses Single-Register wird benutzt, um in den einzelnen Prozessoren jeweils eines der Skalarprodukte zu akkumulieren.

Deklaration `temp = S[5];`

Dieses Single-Register dient als Speicher für Zwischenergebnisse bei der Berechnung eines Skalarproduktschritts.

Deklaration `A_index = R8;`
`A_data = S[6];`

Diese beiden Register speichern zusammen einen Eintrag aus der row-sparse-Matrix `A`. In `A_index` steht die Position des Datums `A_data` innerhalb seiner Zeile von `A`.

Deklaration `B_index = R9;`
`B_data = S[7];`

Diese beiden Register speichern zusammen einen Eintrag aus der column-sparse-Matrix `B`. In `B_index` steht die Position des Datums `B_data` innerhalb seiner Spalte von `B`.

Deklaration `EntryCount_sng = 16;`

Diese Konstante speichert die feste Anzahl von Einträgen, die bei jedem Aufruf von `MAT2_SNG` von jedem der Prozessoren miteinander verrechnet werden. Diese Zahl entspricht genau der Anzahl Einträge, die in 2 Speicherblöcke des Karten-RAM passen.

Das Program `INTROSNG`

Dieses ISA-Programm besteht lediglich aus der Anweisung `Intro_sng`, die aus der Bibliothek `FPSNG` stammt und das Prozessorfeld auf die Verarbeitung von Single-Zahlen vorbereitet.

Das Program `SET0_SNG`

Dieses Programm setzt das Register `sum` aller Prozessoren auf den Wert 0. Es wird aufgerufen, wenn am Anfang einer Berechnung dieses Register initialisiert werden muß.

Das Program `RBANK_W`

`RBANK_W` setzt die inneren Speicherbänke der westlichen Randprozessoren zurück. Dabei wird die Bank auf Vorwärtsbetrieb gestellt.

Das Program `SBANK_W`

Dieses Programm ändert die Zuordnung der Speicherbänke der westlichen Randprozessoren, d. h. Innen- und Außenbank werden vertauscht. Dabei werden die Adreßzähler zurückgesetzt und die Bänke auf Vorwärtsbetrieb gestellt.

Das Program `SBANK_N`

Dieses Programm ändert die Zuordnung der Speicherbänke der nördlichen Randprozessoren, d. h. Innen- und Außenbank werden vertauscht. Dabei werden die Adreßzähler zurückgesetzt und die Bänke auf Vorwärtsbetrieb gestellt.

Das Program `WAITBANK`

Dieses Programm führt `n` `noop`-Operationen aus. Auf diese Weise wartet der Controller bis nach einem `bank`-Befehl wieder Daten an die oder von den Randprozessoren transferiert werden können.

Das Program `MAT2_SNG`

Das Programm `MAT2_SNG` führt den Algorithmus aus 2.4 aus, um `EntryCount_sng` Spalten der Matrix `A` aus den westlichen Randprozessorbänken mit `EntryCount_sng` Zeilen der Matrix `B` aus den nördlichen Randprozessorbänken zusammenzuführen. Es wird für jede Kombination von Einträgen ein Skalarproduktschritt durchgeführt. Falls die Indizes der in einem Prozessor betrachteten Einträge gleich sind, wird das Skalarprodukt in `sum` erhöht.

Das Program `MAT2_OPT`

Das Programm `MAT2_OPT` führt ebenfalls den Algorithmus aus 2.4 aus, um Spalten von `A` mit Zeilen von `B` zusammenzuführen. Der Unterschied zu `MAT2_SNG` besteht darin, daß sämtliche Single-Register ausgenutzt werden. Es werden immer gleichzeitig 4 Spalten von `A` geladen und dann jeweils mit allen Zeilen von `B` verrechnet. Dadurch erreicht man eine Verkleinerung der Periode um ca. 1/8. Für dieses Programm werden insgesamt 6917 Diagonalen generiert, während für `MAT2_SNG` 8033 Diagonalen generiert werden.

Das Program `OUT_SNG`

Dieses Programm überträgt die in dem Register `sum` der einzelnen Prozessoren akkumulierten Skalarprodukte an die westlichen Randprozessoren zur Ausgabe. Die Übertragung geschieht mit Hilfe der Prozedur `OutMat_sng` aus der Bibliothek `FPSNG`.

4 Test des Algorithmus

Um die verschiedenen Algorithmen testen zu können, wurde die Unit `Test` implementiert. Sie stellt nützliche Routinen für den Test zur Verfügung. Außerdem wurden die beiden Programme `SpaTest` und `SpaTest2` verwendet. Sie und die Unit werden in den Unterabschnitten dokumentiert.

Zum Schluß dieses Abschnitts werden einige Ergebnisse des Tests präsentiert.

4.1 Die Unit `Test`

Deklaration `procedure Init1(a, b: PSparseMat_sng; col_a, row_b: Byte);`

Diese Prozedur belegt `col_a` Spalten jeder Zeile der row-sparse-Matrix, auf die `a` zeigt und `row_b` Zeilen der column-sparse-Matrix, auf die `b` zeigt, gleichmäßig mit Einträgen.

Dabei enthält jede Zeile der von dem `TSparseMat_sng`-Objekt `a^` gespeicherten Matrix folgenden Vektor von Einträgen:

$$((2,1) \quad (4,2) \quad (6,3) \quad \cdots \quad (2*\texttt{col\_a}, \texttt{col\_a}))\,.$$

Entsprechendes gilt für die Spalten, der von dem `TSparseMat_sng`-Objekt `b^` gespeicherten Matrix und `row_b`.

`col_a` und `row_b` müssen kleiner oder gleich dem Attribut `Sparse_Param` der übergebenen `TSparseMat_sng`-Objekte, auf die `a` und `b` zeigen, sein.

Deklaration `procedure Init2(a, b: PSparseMat_sng; col_a, row_b: Byte);`

Auch diese Prozedur belegt die Matrizen, auf die `a` und `b` zeigen, gleichmäßig mit Werten. Im Gegensatz zu `Init1` enthält jetzt aber die i-te Zeile (i =

$1, \ldots, 32)$ der von dem TSparseMat_sng-Objekt a^ gespeicherten Matrix den Vektor

$$((2, (i-1) * \texttt{col\_a} + 1) \quad \cdots \quad (2 * \texttt{col\_a}, (i-1) * \texttt{col\_a} + \texttt{col\_a}))\,.$$

Entsprechendes gilt für die Spalten der von dem TSparseMat_sng-Objekt b^ gespeicherten column-sparse-Matrix und row_b.

Die Bedingungen für col_a und row_b gelten auch hier.

Deklaration `procedure MatMult_sequenz_sng(a: PRowMat_Array_sng; b: PColumnMat_Array_sng; col_a, row_b: Byte; c: PMat_sng);`

Diese Prozedur implementiert den sequentiellen Algorithmus aus Abschnitt 2.3. Es werden col_a Spalten der row-sparse-Matrix, auf die a zeigt mit row_b Zeilen der column-sparse-Matrix, auf die b zeigt, verrechnet. Die Prozedur rechnet direkt mit den Daten, d. h. die übergebenen Zeiger a und b zeigen direkt auf ein Datenarray und *nicht* auf ein TSparseMat_sng-Objekt. Das Ergebnis der Matrixmultiplikation wird in c^ zurückgegeben. Für c muß bereits vor der Ausführung Speicher alloziert worden sein.

Die Werte von col_a und row_b müssen kleiner als MaxSize_sng sein und sollten nur so groß sein, wie nötig ist, um alle Nichtnullelemente zu verarbeiten.

Deklaration `function MatMult_isa_2(a, b: PSparseMat_sng; c: PMat_sng): LongInt;`

Diese Prozedur stößt die Ausführung der ISA-Berechnung an. Dazu werden zunächst die Blöcke mit der Nummer 0 des westlichen und nördlichen Karten-RAM mit 0 gefüllt. Dann werden die Daten aus den übergebenen TSparse-Mat_sng-Objekten mittels der Methode Load_Matrix in das Karten-RAM fortlaufend ab Block 1 übertragen. Die Variablen blkst_w, blkend_w, blkst_n und blkend_n des Controllerprogrammes werden initialisiert und dieses schließlich über einen Aufruf von run_iq gestartet. Zuletzt wird das Ergebnis der Matrixmultiplikation durch Aufruf von StoreMat_sng in c^ abgelegt. Der Speicher für c muß bereits vor dem Aufruf dieser Prozedur alloziert worden sein.

Die Prozedur gibt die Laufzeit der Ausführung von run_iq als Funktionswert zurück.

Die Routine isa_board_reset muß bereits vor dem Aufruf dieser Prozedur ausgeführt worden sein. Ebenso muß die richtige Systola-Konfigurationsdatei über load_cfg geladen sein.

Deklaration `procedure LogResults(var F, Err: Text; c, c_seq: PMat_sng);`

Diese Routine schreibt die Ergebnisse einer Berechnung in die Textdatei F. c^ sollte ein mit dem ISA berechnetes und c_seq^ ein sequentiell berechnetes oder Kontrollergebnis sein. Zusätzlich wird die Matrix der Differenzen zwischen c^ und c_seq^ nach F geschrieben. Die größte absolute und prozentuale Abweichung, die zwischen den beiden gegebenen Matrizen auftrat, wird nebst der Position ihres Auftretens innerhalb der Matrizen in die Textdatei Err geschrieben.

Außerdem wird überprüft, ob eine aufgetretene Abweichung die größte seit dem Programmstart oder dem letzten Aufruf von `ResetError` war und ggf. eine Warnung auf dem Bildschirm ausgegeben und diese Abweichung in internen Variablen der Unit vermerkt. Diese können über die Prozedur `GetError` abgefragt werden.

Die verwendeten Dateien `F` und `Err` werden von der Prozedur weder geöffnet noch geschlossen.

Deklaration `procedure LogTime(var TimeLogfile: Text; col_a, row_b, seq, t1, t1_iq, t2, t2_iq: LongInt);`

Diese Prozedur erzeugt eine Zeile mit den Laufzeitdaten eines Programmablaufs für die Laufzeit-Log-Datei `TimeLogFile`. `col_a` und `row_b` sind die Größen der verrechneten Matrizen. Es wird also davon ausgegangen, daß eine `col_a`-row-sparse mit einer `row_b`-column-sparse-Matrix verrechnet wurde. `seq`, `t1`, `t1_iq`, `t2` und `t2_iq` sind die Laufzeiten. Ihre Bedeutung zeigt Tabelle 5. Die in der Tabelle genannte erste Variante des parallelen Algorithmus ist eine Umsetzung der 1. Variante aus Abschnitt 3.3. Da diese eine größere Laufzeit als die andere Variante aufweist, ist diese nicht weiter dokumentiert, taucht jedoch in der Schnittstelle dieser Prozedur auf.

Variable	Bedeutung
`seq`	Laufzeit des sequentiellen Algorithmus
`t1`	Laufzeit der ersten Variante des parallelen Algorithmus (enthält die Laufzeit für das Laden des Karten-RAM und das Zurückspeichern des Ergebnissen)
`t1_iq`	Laufzeit des Controllerprogramms der ersten Variante des parallelen Algorithmus
`t2`	Laufzeit der Umsetzung des parallelen Algorithmus inklusive Laden der Operanden und zurückschreiben des Ergebnisses.
`t2_iq`	Laufzeit des Controllerprogramms

Tabelle 5: Bedeutung der Parameter

Deklaration `procedure GetError(var abs: Double; var proz: Double);`

Über diese Prozedur können die größte absolute und prozentuale Abweichung die zwischen sequentiell und parallel berechneten Matrizenprodukten seit dem Start des Programms bzw. dem letzten Aufruf von `ResetError` auftraten, abgefragt werden. In `abs` wird die größte absolute und in `proz` die größte prozentuale Abweichung zurückgegeben.

Diese Abweichungen werden immer beim Aufruf der Prozedur `LogResults` ermittelt.

Deklaration `procedure ResetError;`

Diese Prozedur setzt die Variablen, die intern die größten absoluten bzw. prozentualen Abweichungen eines parallel mit dem ISA berechneten Ergebnisses von dem entsprechenden sequentiell berechneten Ergebnis speichern, auf 0 zurück.

4.2 Das Programm `SpaTest`

Mit diesem Programm wurden die Daten für die in Abschnitt 4.4 gezeigten Diagramme erzeugt. Das Programm liest zwei Parameter ein; die Start- und die Endproblemgröße.

Zuerst wird das ISA-Board durch Aufruf von `isa_board_reset` initialisiert und die Systola-Konfiguration mit dem Controllerprogramm und den ISA-Programmen geladen. Dann wird für alle Problemgrößen zwischen den beiden eingegebenen Parametern eine row-sparse und eine column-sparse-Matrix erzeugt. Problemgröße ist in diesem Fall die Anzahl der Nichtnullspalten der row-sparse bzw. der Nichtnullzeilen der column-sparse-Matrix. Diese Werte werden jeweils gleich gewählt. Die Daten der beiden Matrizen werden durch die Methode `Random` der entsprechenden `TSparseMat_sng`-Objekte erzeugt.

Dann wird durch Aufruf der Prozeduren `MatMult_sequenz_sng` und `MatMult_isa2` das Matrizenprodukt einmal sequentiell und einmal mit dem ISA berechnet. Die Laufzeiten werden durch Aufruf von `LogTime` in der Datei `time.log` verzeichnet. Für jede Berechnung wird die größte absolute und prozentuale Abweichung des mit dem ISA-berechneten Ergebnisses vom sequentiell berechneten Ergebnis in der Datei `error.log` gespeichert. Die Ergebnisse selbst werden durch Aufruf von `LogResults` in der Datei `Mat_2.log` gespeichert.

4.3 Das Programm `SpaTest2`

Dieses Programm wurde benutzt, um die Laufzeiten aus Tabelle 6 zu ermitteln. Zu Beginn des Programms werden die Anzahl Durchläufe k und die Problemgrößen r und c für eine r-row-sparse und eine c-column-sparse-Matrix eingegeben.

Die Daten werden wiederum durch Aufruf der Methode `Random` erzeugt. Dann werden k Berechnungen jeweils mit dem sequentiellen und dem ISA-Programm durchgeführt. Jeder der k Durchläufe arbeitet mit denselben zu Beginn des Programm erzeugten Daten. Wie bei `SpaTest` werden die Log-Dateien `time.log`, `error.log` und `Mat_2.log` erzeugt.

4.4 Testergebnisse

Die Diagramme aus Abbildung 14 und 15 geben die Laufzeiten des sequentiellen und des ISA-Programms für verschieden große Matrizen an. Die x-Achse ist dabei immer mit der Anzahl k der Nichtnullelemente beschriftet, d. h. es wird jeweils eine k-row-sparse mit einer k-column-sparse Matrtix multipliziert. Die Einträge der Matrizen sind für jeden Wert von k zufällig gewählt.

Die Laufzeiten verhalten sich genau wie bereits in Abschnitt 2.5.1 angegeben. Solange der Parameter k klein gegen die Größe des befehlssystolischen Feldes ist, ist das ISA-Programm schneller. Doch da die Periode T_{isa} quadratisch in k wächst, holt der sequentielle Algorithmus mit steigendem k auf und überflügelt das ISA-Programm früher (beim K5) oder später (beim P60).

Für den schnelleren der beiden Prozessoren (AMD K5 166 MHz) ist dies nochmal im einzelnen in Tabelle 6 aufgeschlüsselt.

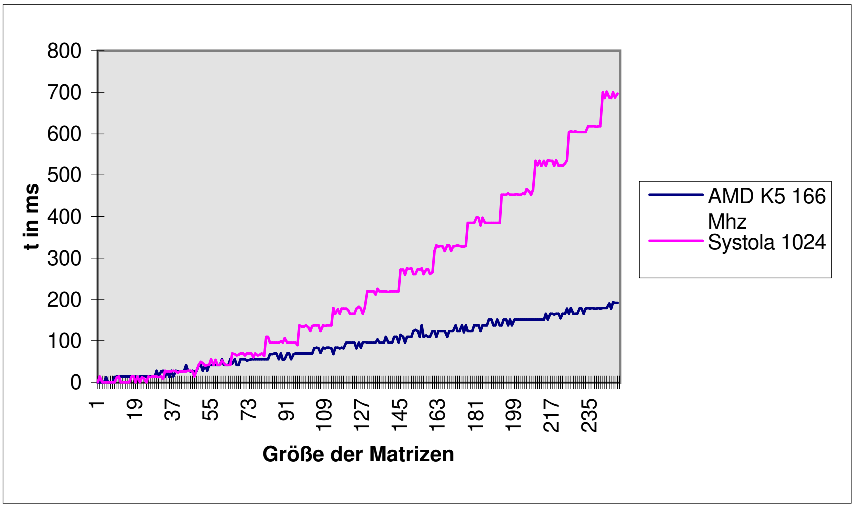

Abbildung 14: Laufzeiten der Systola verglichen mit dem AMD K5 166 MHz

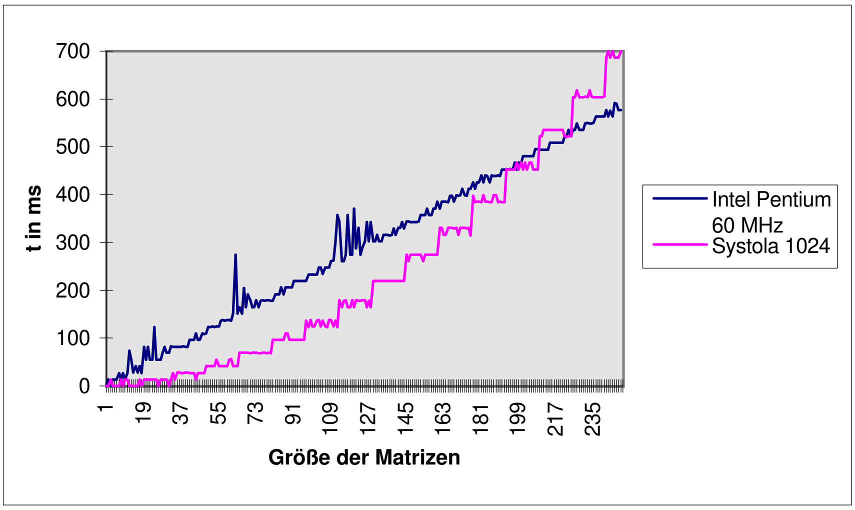

Abbildung 15: Laufzeiten der Systola verglichen mit dem Intel Pentium 60 MHz

Die größte aufgetretene absolute Abweichung zwischen einem sequentiell und einem mit dem ISA-Programm berechneten Skalarprodukt beträgt $0,0078$. Die größte aufgetretene prozentuale Abweichung beträgt $0,052\%$. Dabei liegen die von der Methode `Random` der verwendeten `TSparseMat_sng`-Objekte zufällig erzeugten Werte zwischen -100 und $+100$ (siehe auch 3.2.3).

Größe	AMD K5 166 MHz	Systola 1024
16	11	3
32	23	13
64	44	44
128	91	173
248	184	691

Tabelle 6: Laufzeiten des sequentiellen und des ISA-Programms in Millisekunden.

5 Quelltexte

5.1 Die Unit `Sparse`

```
(*******************************************************************************************)
(*                                                                                         *)
(* Abstrakter Datentyp fuer duennbesetzte Matrizen mit Single- oder Double- Eintraegen    *)
(*                                                                                         *)
(* Letzte Aenderung: 24.03.1998                                                            *)
(*                                                                                         *)
(*******************************************************************************************)

Unit Sparse;

{$R+,N+}

Interface

const

  n = 32;                      { Groesse der Prozessorfeldes }

  MaxSize_sng = 255;            { Maximale Groesse fuer den Sparse_Param }
  MaxSize_dbl = 204;            { soviel passt in 64K }

  blocksize = 1024;            {Groesse eines Speicherblockes in Worten }

type

(*************************** Arraytypen fuer dichtbesetzte Matrizen ***********************)

  PMat_sng = ^TMat_sng;                              { fr Single - Zahlen }
  TMat_sng = Array [1..n, 1..n] of Single;

  PMat_dbl = ^TMat_dbl;                              { fr Double - Zahlen }
  TMat_dbl = Array [1..n, 1..n] of Double;

(******************************** TSparseMat - Klassen **********************************)

  { Datentyp fuer einen Matrixeintrag }

  TEntry_sng = record
               index: Word;
               dummy: Word;          { Fuellwort }
               data: Single;
             end;

  TEntry_dbl = record
               index: Word;
               data: Double;
```

```
            end;

const

  { Groesse eines Eintrages in 16-bit Worten }

  SizeOf_TEntry_sng = SizeOf(TEntry_sng) div 2;
  SizeOf_TEntry_dbl = SizeOf(TEntry_dbl) div 2;

  { Anzahl Eintraege, die in eine "Zeile" eines Blockes passen }
  Row_Entry_cnt_sng = 8;          { blocksize div (32 * SizeOf_TEntry_sng) }
  Row_Entry_cnt_dbl = 6;          { blocksize div (32 * SizeOf_TEntry_dbl) }

type

  { Arraytypen fuer die duennbesetzte Matrizen }

  PRowMat_Array_sng = ^TRowMat_Array_sng;                  { fuer Single Zahlen }
  TRowMat_Array_sng = Array [1..n, 1..MaxSize_sng] of TEntry_sng;

  PColumnMat_Array_sng = ^TColumnMat_Array_sng;
  TColumnMat_Array_sng = Array [1..MaxSize_sng, 1..n] of TEntry_sng;

  PRowMat_Array_dbl = ^TRowMat_Array_dbl;                  { fuer Double Zahlen }
  TRowMat_Array_dbl = Array [1..n, 1..MaxSize_dbl] of TEntry_dbl;

  PColumnMat_Array_dbl = ^TColumnMat_Array_dbl;
  TColumnMat_Array_dbl = Array [1..MaxSize_dbl, 1..n] of TEntry_dbl;


  { TSparseMat_sng Klasse }

  PSparseMat_sng = ^TSparseMat_sng;
  TSparseMat_sng = object
                     a: PRowMat_Array_sng;
                     b: PColumnMat_Array_sng;
                     Sparse_Param: Word;
                     RowSparse: Boolean;

                     { Initialisieren der Matrix }
                     constructor Init(ASparse_Param: Word; ARowSparse: Boolean);

                     { Speicher freigeben usw. }
                     destructor Done;

                     { Datenzugriff }
                     procedure SetData(i, j: Word; AIndex: Word; AData: Single);
                     procedure GetData(i, j: Word; var AIndex: Word; var AData: Single);

                     { Sparse_Type feststellen }
                     function isRowSparse: Boolean;
                     function isColumnSparse: Boolean;

                     { Ausgabe auf dem Bilschirm oder in eine Datei; bei PureData=TRUE
                       nur die Daten (ohne Text)
                     }
                     procedure Print(var F: Text; PureData: Boolean);

                     { Eingabe aus einer Datei }
                     procedure Read(var F: Text);

                     { Zufaellig mit Werten fuellen }
                     procedure Random(dim: Word);

                     { Laedt die Matrix in den Karten-Speicher }
                     procedure Load_Matrix_West(startblock: Byte);
```

```
                    procedure Load_Matrix_North(startblock: Byte);
                    procedure Load_Matrix(startblock: Byte);

                    { Liefert die Anzahl 1K Bloecke zur Speicherung der Matrix im Karten-RAM }
                    function BlockCount: Byte;

                  private
                    procedure RangeCheck(i, j: Word);
                  end;

  { TSparseMat_dbl Klasse }

  PSparseMat_dbl = ^TSparseMat_dbl;
  TSparseMat_dbl = object
                  end;


(***************************** sonstige Prozeduren und Funktionen ************************)

{ Speichert eine Ergenismatrix aus dem westl. Karten-RAM nach c }
procedure StoreMat_sng(c: PMat_sng);

{ Ausgabe eines Ergebinesses auf dem Bildschirm }
procedure PrintMat_sng(var F: Text; c: PMat_sng);

(*********************************** Implementationsteil ********************************)

Implementation

uses
{$ifdef emulator}
  isawin_e
{$else}
  isawin
{$endif}
;

{ Dummy Declaration for Emulation-Mode }

{$ifdef emulator}
{ Diese Routine ist im Emulator nicht implementiert. Deshalb macht sie
  hier zu Debugging Zwecken einige Ausgaben
}
procedure load_block_pitch(border, block: Byte; count, rows, pitch: Word; databuf: Pointer);
type PEntry_sng = ^TEntry_sng;
var e: PEntry_sng;
    i, j, pt: Word;
Begin
  WriteLn('load_block_pitch(', border, ', ', block, ', ', count, ', ', rows,
    ', ', pitch, ', ', Seg(databuf), ':', Ofs(databuf), ');');

  clear_block(border, block);

  pt := 0;

  for i:=0 to rows-1 do Begin
    WriteLn('Row ',i, ':');
    for j:=0 to count div SizeOf_TEntry_sng - 1 do Begin
      e := PEntry_sng(Ptr(Seg(databuf^), Ofs(databuf^)+pt*2+j*SizeOf(TEntry_sng)));
      WriteLn('Index:', e^.index:10, '    Data:', e^.data:15:4);
    End;

    Inc(pt, pitch+count);
    ReadLn;
  End;
End;
```

```
{$endif}

(************************************** TSparseMat_sng - Klasse **************************************)

{ Der Konstruktor erzeugt ein neues Objekt.
  Die Attribute RowSparse und Sparse_Param werden mit den geg. Werten belegt.
  Es wird Speicher fuer die Arrays a bzw. b alloziert und diese werden
  komplett mit NULL gefuellt.
}
constructor TSparseMat_sng.Init(ASparse_Param: Word; ARowSparse: Boolean);
Begin

  { Parameter setzen }
  RowSparse := ARowSparse;
  Sparse_Param := ASparse_Param;

  { Parameter pruefen }
  if Sparse_Param > MaxSize_sng then Halt(201);    { mehr kann nicht dargestellt werden }

  { Speicher fuer die Matrix allozieren }
  if isRowSparse then Begin
    b := NIL;
    New(a);

    { Fillchar wird gebraucht, damit das letzte load_block_pitch geht ! }
    FillChar(a^, n*MaxSize_sng*SizeOf(TEntry_sng), 0);
  End
  Else Begin
    a := NIL;
    New(b);

    { Fillchar wird gebraucht, damit das letzte load_block_pitch geht ! }
    FillChar(b^, n*MaxSize_sng*SizeOf(TEntry_sng), 0);
  End;
End;

{ Der Destruktor zerstoert ein Objekt.
  Der von den Arrays a bzw. b belegte Speicher wird freigegeben.
}
destructor TSparseMat_sng.Done;
Begin
  if isRowSparse then Dispose(a)
  else Dispose(b);
End;

{ Datenzugriff
  An die Position (i,j) der duennbesetzten Matrix wird der Index
  AIndex und das Datum AData geschrieben.

  (i,j) muss ein gueltiger Index fuer das Array sein!
}
procedure TSparseMat_sng.SetData(i, j: Word; AIndex: Word; AData: Single);
Begin
  RangeCheck(i,j);
  if isRowSparse then Begin
    a^[i,j].index := AIndex;
    a^[i,j].data := AData;
  End
  Else Begin
    b^[i,j].index := AIndex;
    b^[i,j].data := AData;
  End;
End;

{ Der Eintrag an der Position (i,j) der duennbesetzten Matrix wird
  zurueckgegeben.
```

```
  In AIndex steht nach der Ausfuehrung der Index und in AData das
  Datum des Eintrages an der Position (i,j).

  (i,j) muss ein gueltiger Index fuer das Array sein!
}
procedure TSparseMat_sng.GetData(i, j: Word; var AIndex: Word; var AData: Single);
Begin
  RangeCheck(i,j);
  if isRowSparse then Begin
    AIndex := a^[i,j].index;
    AData := a^[i,j].data;
  End
  Else Begin
    AIndex := b^[i,j].index;
    AData := b^[i,j].data;
  End;
End;

{ Sparse_Type feststellen
  Liefert TRUE, gdw. die von dem Objekt dargestellte Matrix row-sparse ist.
}
function TSparseMat_sng.isRowSparse: Boolean;
Begin
  isRowSparse := RowSparse;
End;

{ Liefert TRUE, gdw.  die von dem Objekt dargestellte Matrix column-sparse ist.
}
function TSparseMat_sng.isColumnSparse: Boolean;
Begin
  isColumnSparse := not RowSparse;
End;

{ Ausgabe }
{ Schreibt die im Objekt gespeicherte Matrix in die Textdatei F.
  Die Daten werden fuer row-sparse-Matrizen zeilenweise und fuer
  column-sparse-Matrizen spaltenweise geschrieben.
  Die Eintraege werden im Format "Index: <index>    Data: <data>"
  geschrieben.
  Falls PureData TRUE ist, wird kein weiterer Text ausgegeben,
  sondern nur die Daten im Format "<index> <data>"
}
procedure TSparseMat_sng.Print(var F: Text; PureData: Boolean);
var i, j: Word;
Begin
  if isRowSparse then Begin
    for i:=1 to n do Begin
      if not PureData then WriteLn(F, 'Row ', i, ':');
      for j:=1 to Sparse_Param do
        if PureData then WriteLn(F, a^[i,j].index, ' ', a^[i,j].data)
        else WriteLn(F, ' Index: ', a^[i,j].index:10, ' Data: ', a^[i,j].data:15:2);

    End;
  End
  Else Begin
    for j:=1 to n do Begin
      if not PureData then WriteLn(F, 'Column ', j, ':');
      for i:=1 to Sparse_Param do
        if PureData then WriteLn(F, b^[i,j].index, ' ', b^[i,j].data)
        else WriteLn(F, ' Index: ', b^[i,j].index:10, ' Data: ', b^[i,j].data:15:2);

    End;
  End;
End;

{ Eingabe der Matrix aus der Textdatei F.
```

```
  Die Matrix muss in einem Format vorliegen, dass dem mit Print(F, TRUE)
  (d.h. nur Daten werden geschrieben) erzeugten gleicht.
}
procedure TSparseMat_sng.Read(var F: Text);
var i, j: Word;
Begin
  if isRowSparse then Begin
    for i:=1 to n do
      for j:=1 to Sparse_Param do
        ReadLn(F, a^[i,j].index, a^[i,j].data);
  End
  Else Begin
    for j:=1 to n do
      for i:=1 to Sparse_Param do
        ReadLn(F, b^[i,j].index, b^[i,j].data);

  End;
End;

{ Fuellen der Matrix mit Zufallswerten.
  dim gibt die Anzahl der Spalten/Zeilen der zu betrachtenden Matrix an.
  Bsp.: Fuer eine row-sparse-Matrix mit dim=10000 werden fuer die
        Eintraege Indices zwischen 1 und 10000 erzeugt. D.h. die
        betrachtete Matrix ist eine n x 10000 Matrix

        Fuer column-sparse-Matrix ist es umgekehrt.

  dim muss groesser als Sparse_Param sein !!!!

  Die erzeugten Daten liegen zwischen -100 und 100.
  Die Indizes sind fuer row-sparse-Matrizen zeilenweise sortiert; fuer
  column-sparse-Matrizen spaltenweise.
}
procedure TSparseMat_sng.Random(dim: Word);
type PArray = ^TArray;
     TArray = Array[1..65520] Of Boolean;
var i, j, k: Word;
    was: PArray;
    e: TEntry_sng;

 procedure SetEntry(var e: TEntry_sng);
 var neg: Byte;
     index: Word;
 Begin
   { Index auswuerfeln }
   index := System.Random(dim) + 1;

   { Index was schonmal ? }
   while was^[index] do Begin
     {     WriteLn('Wrong index!'); }
     Inc(index);
     if index = dim+1 then index := 1;
   End;

   was^[index] := TRUE;                          { Nun war der Index ! }
   e.index := index;

   { Vorzeichen auswuerfeln }
   neg := System.Random(10);
   if neg>=5 then
     e.data := System.Random * 100                { hoechstens 1e2; }
   else e.data := -System.Random * 100;
 End;

Begin
  { Speicher fuer Merkliste allozieren }
```

```
  GetMem(was, dim);

  { Eintraege belegen }
  if isRowSparse then Begin
     for i:=1 to n do Begin

       { Merkliste auf FALSE setzen }
       FillChar(was^, dim, 0);

       for j:=1 to Sparse_Param do Begin

         { Eintrag auswuerfeln }
         SetEntry(e);

         { Sortiert einfuegen }
         k := j-1;
         while (k>=1) and (a^[i,k].index > e.index) do Begin
           a^[i,k+1] := a^[i,k];
           Dec(k);
         End;

         a^[i, k+1] := e;       { Werte setzen; k ist eins zu niedrig }
       End;
     End;
  End
  Else Begin
    for j:=1 to n do Begin

      { Merkliste auf FALSE setzen }
      FillChar(was^, dim, 0);

      for i:=1 to Sparse_Param do Begin

        { Eintrag auswuerfeln }
        SetEntry(e);

        { Sortiert einfuegen }
        k := i-1;
        while (k>=1) and (b^[k,j].index > e.index) do Begin
          b^[k+1, j] := b^[k, j];
          Dec(k);
        End;
        b^[k+1, j] := e;       { Werte setzen; k ist eins zu niedrig }
      End;
    End;
  End;

  { Speicher freigeben }
  FreeMem(was, dim);
End;

{ Laden der Matrix in den Karten-Speicher.
  Die von dem Objekt dargestellte Matrix wird in das westl. Karten-RAM geladen.
  Die Matrix muss row-sparse sein, sonst bricht die Routine ab. Diese
  Routine sollte nicht aufgerufen werden! Stattdessen sollte vom Nutzer die
  Routine Load_Matrix benutzt werden!!!

  Die Matrix wird beginned ab dem Block mit der Nummer startblock
  fortlaufend gespeichert.
  In jedem der Bloecke stehen n Zeilen mit jeweils Row_Entry_cnt_sng
  Eintraegen hintereinander. Ueberzaehlige Woerter des Blockes werden
  mit NULL gefuellt.
}

{$ifdef emulator}
{ Da die Routine load_block_pitch im Emulator fehlt, ist ihre Funktionalitt
```

```
  fuer diesen Fall nachgebildet.
}
procedure TSparseMat_sng.Load_Matrix_West(startblock: Byte);
type PArray = ^TArray;
     TArray = Array [1..n*MaxSize_sng] Of TEntry_sng;
var tmp: PArray;
    blockcnt, i, x, y, h: Byte;
    lastcnt, tmp_index: Word;
Begin
  if not isRowSparse then Exit;

  blockcnt := BlockCount;
  new(tmp); tmp_index := 1;

  { Arrray mit NULL fuellen }
  FillChar(tmp^, n*MaxSize_sng*SizeOf(TEntry_sng), 0);   { wichtig, weil immer ganze Bloecke verarbeitet werden ! }

  { Daten in die richtige Reihenfolge fuer load_block bringen }
  { Zuerst die kompletten Bloecke vorbereiten }
  if blockcnt>1 then
    for i:=0 to blockcnt-2 do
       for x:=1 to n do
         for y:=i*Row_Entry_cnt_sng+1 to (i+1)*Row_Entry_cnt_sng do Begin
           tmp^[tmp_index] := a^[x,y];
           Inc(tmp_index);
         End;

  { Letzten, teilweise gefuellten Block vorbereiten }
  lastcnt := Sparse_Param mod Row_Entry_cnt_sng;
  if lastcnt = 0 then lastcnt := Row_Entry_cnt_sng;

  for x:=1 to n do Begin
    h := (blockcnt-1)*Row_Entry_cnt_sng+1;

    { Daten nach tmp kopieren }
    for y:=h to h+lastcnt-1 do Begin
      tmp^[tmp_index] := a^[x,y];
      Inc(tmp_index);
    End;

    { Nullen ueberspringen }
    Inc(tmp_index, Row_Entry_cnt_sng - lastcnt);
  End;

  { Bloecke laden }
  load_block(WEST, startblock, blockcnt*blocksize, tmp);
End;
{$else}   { kein Emulator -- sondern das richtige Leben }
procedure TSparseMat_sng.Load_Matrix_West(startblock: Byte);
var blockcnt, i: Byte;
Begin
  if not isRowSparse then Exit;

  blockcnt := BlockCount;

  { Bloecke laden; am Schlu werden Nullen kopiert! }
  for i:=0 to blockcnt-1 do
    { 32 Prozessoren; 32 Worte pro Prozessor; danach (MaxSize_sng-Row_Entry_cnt_sng) Eintraege ueberspringen }
    load_block_pitch(WEST, i+startblock, Row_Entry_cnt_sng*SizeOf_TEntry_sng, n,
      (MaxSize_sng - Row_Entry_cnt_sng)*SizeOf_TEntry_sng,
      @(a^[1,i*Row_Entry_cnt_sng+1]));
End;
{$endif}   {emulator}

{ Die von dem Objekt dargestellte Matrix wird in das noerdl. Karten-RAM geladen.
  Die Matrix muss column-sparse sein, sonst bricht die Routine ab. Diese Routine
```

```
  sollte nicht aufgerufen werden! Stattdessen sollte vom Nutzer die
  Routine Load_Matrix verwendet werden!!!

  Die Matrix wird beginned ab dem Block mit der Nummer startblock
  fortlaufend gespeichert.
  In jedem der Bloecke stehen Row_Entry_cnt_sng Zeilen mit jeweils 32
  Eintraegen hintereinander. Ueberzaehlige Woerter des Blockes werden
  mit NULL gefuellt.
}
procedure TSparseMat_sng.Load_Matrix_North(startblock: Byte);
var blockcnt, i: Byte;
    lastcnt: Word;
Begin
  if not isColumnSparse then Exit;

  blockcnt := BlockCount;

  { Bloecke laden; am Schluss mit Nullen fuellen! }
  for i:=0 to blockcnt-1 do
    { Hier ohne Pitch, weil stets ganze Zeilen kopiert werden }
    load_block(NORTH, i+startblock, blocksize, @(b^[Row_Entry_cnt_sng*i+1,1]));
End;

{ Liefert die Anzahl 1K-Bloecke, die zur Speicherung der Matrix im Karten-RAM
  des ISA erforderlich sind.
}
function TSparseMat_sng.BlockCount: Byte;
var blkcnt: Byte;
Begin
  blkcnt := Sparse_Param div Row_Entry_cnt_sng;  { 8 Eintraeger passen in einen Block }
  if Sparse_Param mod Row_Entry_cnt_sng <> 0 then
    Inc(blkcnt);  { aufrunden }

  BlockCount := blkcnt;
End;

{ Laedt die von dem Objekt dargestellte Matrix in das Karten-RAM.
  row-sparse-Matrizen werden in das westl. und column-sparse-Matrizen in
  das noerdl. Karten-RAM geladen.

  Die Matrix wird beginned an dem Block startbock fortlaufend in das
  Karten-RAM kopiert. (weitere Informationen siehe bei Load_Matrix_*)
}
procedure TSparseMat_sng.Load_Matrix(startblock: Byte);
Begin
  if isRowSparse then Load_Matrix_West(startblock)
  else Load_Matrix_North(startblock);
End;

{ Private Methode zur Bereichsueberpruefung.
  Diese Methode wird von SetData und GetData zur Ueberpruefung der
  Gueltigkeit der Position (i,j) innerhalb des Arrays a bzw. b aufgerufen.
}
procedure TSparseMat_sng.RangeCheck(i, j: Word);
Begin

  { Ueberprueft, ob die Indices i,j innerhalt des Array liegen }
  if isRowSparse then Begin
    if (i>n) or (j>Sparse_Param) then Halt(201);
  End
  Else if (i>Sparse_Param) or (j>n) then Halt(201);
End;

(**************************** sonstige Prozeduren und Funktionen ************************)

{ Speichert eine 32x32 Matrix aus dem westl. Karten-RAM in das
```

```
  Array c. Die Matrix muss sich in den Bloecken 0 und 1 befinden.

  HINWEIS: Zur Blockzuordnung siehe ISA-Handbuch S. 4-16 ff.
}
procedure StoreMat_sng(c: PMat_sng);
Begin
  store_matrix(WEST, 0, SNG, c, n, n, n);     { Bisher keine Partitionierung !!! }
End;

{ Schreibt die dichtbesetzte Matrix c in die Textdatei F.
  Die Matrix wird zeilenweise geschrieben.
}
procedure PrintMat_sng(var F: Text; c: PMat_sng);
var i, j: Word;
Begin
  for i:=1 to n do Begin
    Write(F, 'Row ', i:4, ':');
    for j:=1 to n do Write(F, c^[i,j]:20:4);
    WriteLn(F);
  End;
End;

End.
```

5.2 Die Unit Test

```
(******************************************************************************************)
(*                                                                                        *)
(* Unit fuer Routinen, die im Testprogramm benoetigt werden                               *)
(*                                                                                        *)
(* Letzte Aenderung: 24.03.1998                                                           *)
(*                                                                                        *)
(******************************************************************************************)

Unit Test;

Interface

uses Sparse, WinProcs,
{$ifdef emulator}
isawin_e;
{$else}
isawin;
{$endif}

(**************************** Matrix-Initialisierungs-Routinen ***************************)

{ Erzeugen gleichmaessige Testmatrizen in der anzugebenen Groesse }
{ col_a bzw. row_n muessen kleiner oder gleich dem Sparse_Param der
  entsprechenden Matrix sein
}
procedure Init1(a, b: PSparseMat_sng; col_a, row_b: Byte);
procedure Init2(a, b: PSparseMat_sng; col_a, row_b: Byte);

(********************************* Sequentieller Algorithmus *******************************)
{ Fuehrt eine duennbesetzte sequentielle Matrixmultiplikation durch. }
{ a, b sind die Operanden,
  col_a ist die Spaltenzahl von a   ( Nichtnullspalten ),
  row_b ist die Zeilenzahl von b    (      dto.       ),

  HINWEIS: Spalten/Zeilen von a, b muessen nach den Indices sortiert sein

  c ist die Ergebnismatrix
}
procedure MatMult_sequenz_sng(a: PRowMat_Array_sng; b: PColumnMat_Array_sng; col_a, row_b: Byte;
  c: PMat_sng);
```

```
(****************************** Parallel Matrix Multiplikation ****************************)

{ a,b sind die Operanden; in c wird das Ergebnis
  gespeichert;
  Die Routinen geben die Controller-IQ-Laufzeit in Millisekunden zurueck.
}
function MatMult_isa_1(a, b: PSparseMat_sng; c: PMat_sng): LongInt;
function MatMult_isa_2(a, b: PSparseMat_sng; c: PMat_sng): LongInt;

(************************************** Result logging *************************************)
{ Dies Routinen schreibt die beiden geg. Matrizen in die Textdatei F.
  In F werden die Matrizen c und c_seq geschrieben durch Aufruf von
  Print_Mat.

  c sollte ein parallel berechnetes Ergebnis und c_seq ein sequentiell berechnetes oder
  Kontrollergebnis sein. Es wird zusaetzlich die Matrix der Abweichungen berechnet und
  nach F geschrieben.

  Die groesste Abweichung, die zwischen Werten von c und c_seq auftritt wird nebst
  Position innerhalb der Matrix nach Err geschrieben.

  Auerdem wird der seit dem Programmstart groesste Fehler auf dem Bildschirm ausgegeben,
  falls dieser in der Differenzmatrix c - c_seq auftritt.
}
procedure LogResults(var F, Err: Text;  c, c_seq: PMat_sng);

(******************************** Rechenzeit loggen ***********************************)
{ col_a ist die Spaltenzahl von A     ( nur Nichtnullspalten )
  row_b ist die Zeilenzahl von B      (         dto.         )
  seq ist die Laufzeit des Sequentiellen Algorithmus.
  t1 ist die Gesamtlaufzeit der ersten Variante
  t1_iq ist die Laufzeit der Controller-IQ der ersten Variante
  t2 ist die Gesamtlaufzeit der zweiten Variante
  t2_iq ist die Laufzeit der Controller-IQ der zweiten Variante
}
procedure LogTime(var TimeLogfile: Text; col_a, row_b, seq, t1, t1_iq, t2, t2_iq: LongInt);

(******************************* Rechenfehler ermitteln ********************************)
{ Gib den maximal seit dem Start des Programms oder seit dem letzten Aufruf
  von ResetError ermittelten Rechenfehler zurueck. Ein Rechenfehler ist eine
  Diffenrenz zwischen einem sequentiell und einem parallel ermittelten
  Ergebnis. Diese Differenzen werden in LogResults ermittelt.

  abs ist der grte Absolute Fehler
  proz ist der grte prozentuale Fehler
}
procedure GetError(var abs: Double; var proz: Double);

{ Setzt die Werte fr die maximalen Fehler auf 0 zurueck.
}
procedure ResetError;

(********************************** Implementationsteil ********************************)

Implementation

const
  MaxDiff: Double = 0;           { maximale absolute Differenz }
  MaxDiffProz: Double = 0;       { maximale prozentuale Differenz }

(*************************** Matrix-Initialisierungs-Routinen **************************)

{ a ist row-sparse; b ist column-sparse;
  col_a ist die Spaltenzahl von a,
  row_b ist die Zeilenzahl von b
```

```
}
procedure Init1(a, b: PSparseMat_sng; col_a, row_b: Byte);
var i, j: Word;
Begin
  for i:=1 to n do
    for j:=1 to col_a do a^.SetData(i, j, 2*j, j);

  for i:=1 to row_b do
    for j:=1 to n do b^.SetData(i, j, 2*i, i);
End;

procedure Init2(a, b: PSparseMat_sng; col_a, row_b: Byte);
var i, j: Word;
Begin
  for i:=1 to n do
    for j:=1 to col_a do a^.SetData(i, j, 2*j, (i-1)*col_a+j);

  for i:=1 to row_b do
    for j:=1 to n do b^.SetData(i, j, 2*i, (i-1)*n+j);
End;

(********************************** Sequentieller Algorithmus ********************************)

procedure MatMult_sequenz_sng(a: PRowMat_Array_sng; b: PColumnMat_Array_sng; col_a, row_b: Byte;
  c: PMat_sng);
var
  x, y, i, j: Word;
  tmp: Single;
Begin
  for x:=1 to n do
    for y:=1 to n do begin
      { duennes Skalarprodukt ausrechnen}
      c^[x,y] := 0;
      i:=1; j:=1;
      while (i<=col_a) and (j<=row_b) do begin

        { Indizes weitersetzen}
        if a^[x,i].index < b^[j, y].index then Inc(i)
        else if a^[x,i].index > b^[j,y].index then Inc(j)

        { nichtausloeschende Faktoren }
        else Begin
          c^[x,y] := c^[x,y] + a^[x,i].data * b^[j,y].data;
          Inc(i);    { Indizes weitersetzen }
          Inc(j);
        end;
      end;
    end;
End;

(*************************************** Result logging **************************************)

procedure LogResults(var F, Err: Text;  c, c_seq: PMat_sng);
var diff: PMat_sng;
    i, j: Word;
    flag: Boolean;
    abs_diff, proz: Double;
    max, maxproz: Record v: Double; i, j: Word; End;
Begin
  { Speicher fuer Differenzmatrix allozieren }
  New(diff);

  { Berechnung der Differenz }

  With max do Begin
    v := 0; i := 0; j := 0;
```

```
  End;
  With maxproz do Begin
    v := 0; i := 0; j := 0;
  End;
  for i:=1 to n do
    for j:=1 to n do Begin

      { Abweichung berechnen }
      diff^[i,j] := c^[i,j] - c_seq^[i,j];

      { Absolute Abweichung vom sequentiellen Ergebnis }
      abs_diff := abs(diff^[i,j]);
      { Maximum in dieser Matrix }
      if abs_diff > max.v then Begin
        max.v := abs_diff; max.i:=i; max.j:=j;
      End;

      { Globales absolutes Maximum }
      if (abs_diff > MaxDiff) then Begin
        MaxDiff := abs_diff;
        WriteLn;
        WriteLn('WARNING: New abs max diff at [', i, ',', j,'] is: ', MaxDiff:20:5);
        WriteLn;
      End;

      { Prozentuale Abweichung vom sequentiellen Ergebnis }
      if c_seq^[i,j] <> 0 then Begin
        proz := abs(100 * diff^[i,j] / c_seq^[i,j]);
        { Maximum in dieser Matrix }
        if proz > maxproz.v then Begin
          maxproz.v := proz; maxproz.i := i; maxproz.j := j;
        End;

        { Globales prozentuales Maximum }
        if (proz > MaxDiffProz) then Begin
          MaxDiffProz := proz;
          WriteLn;
          WriteLn('WARNING: New percent max diff at [', i, ',', j,'] is: ', MaxDiffProz:20:5, ' %');
          WriteLn;
        End
      End
    End;

  { Fehler schreiben }
  Write(Err, max.v:15:10, ' at [', max.i:2, ',', max.j:2, ']',
    maxproz.v:15:10, ' at [', maxproz.i:2, ',', maxproz.j:2, ']');

  { Ausgabe }
  WriteLn(F, 'ISA - calculated result');
  WriteLn(F, '-----------------------');
{  PrintMat_sng(F, c); WriteLn(F); }

  WriteLn(F, 'Sequentially calculated result');
  WriteLn(F, '------------------------------');
{  PrintMat_sng(F, c_seq); WriteLn(F); }

  WriteLn(F, 'Difference matrix');
  WriteLn(F, '-----------------');
{  PrintMat_sng(F, diff); WriteLn(F); }
End;

(****************************** Parallele Matrix Multiplikation ****************************)

function MatMult_isa_1(a, b: PSparseMat_sng; c: PMat_sng): LongInt;
var adr: Byte;
    Tstart, Tend, diff: LongInt;
```

```
Begin

  { Matrizen laden }
  a^.Load_Matrix(0);    { nach Westen }
  b^.Load_Matrix(0);    { nach Norden }

  { Variablen laden }
  get_var_address('blkcnt_w', 2, adr);
  set_var(adr, a^.BlockCount);

  get_var_address('blkcnt_n', 2, adr);
  set_var(adr, b^.BlockCount);

  { Controllerprogramm starten }
  Tstart := GetCurrentTime;
  run_iq('START', 2);
  Tend := GetCurrentTime;

  { Laufzeit ausrechnen }
  diff := Tend - Tstart;
  WriteLn('Time for MatMult_isa_1: ', diff, ' Milliseconds.');

  { Ergebnis holen }
  StoreMat_sng(c);

  { Laufzeit zurueckgeben }
  MatMult_isa_1 := diff;
End;

function MatMult_isa_2(a, b: PSparseMat_sng; c: PMat_sng): LongInt;
var startblock, adr: Byte;
    Tstart, Tend, diff: LongInt;
const iq = 4;
Begin

  { Fuellbloecke erzeugen }
  clear_block(WEST, 0); clear_block(NORTH, 0);
  startblock := 1;

  { Matrizen laden }

  a^.Load_Matrix(startblock);    { nach Westen }
  b^.Load_Matrix(startblock);    { nach Norden }

  { Variablen laden }

  { Anfang }
  get_var_address('blkst_w', iq, adr);
  set_var(adr, startblock);

  get_var_address('blkst_n', iq, adr);
  set_var(adr, startblock);

  { Ende }
  get_var_address('blkend_w', iq, adr);
  set_var(adr, a^.BlockCount+startblock);
  get_var_address('blkend_n', iq, adr);
  set_var(adr, b^.BlockCount+startblock);

  { Controllerprogramm starten }
  Tstart := GetCurrentTime;
  run_iq('START', iq);
  Tend := GetCurrentTime;

  { Laufzeit ausrechnen }
  diff := Tend - Tstart;
```

```
  WriteLn('Time for MatMult_isa_2: ', diff, ' Milliseconds.');

  { Ergebnis holen }
  StoreMat_sng(c);

  { Laufzeit zurueckgeben }
  MatMult_isa_2 := diff;
End;

(******************************** Rechenzeit loggen **********************************)

procedure LogTime(var TimeLogfile: Text; col_a, row_b, seq, t1, t1_iq, t2, t2_iq: LongInt);
Begin
  { Laufzeiten schreiben }
  WriteLn(TimeLogfile, col_a:8, row_b:8, seq:8, t1:8, t1_iq:8, t2:8, t2_iq:8);
End;

(******************************** Rechenfehler ermitteln ********************************)

procedure GetError(var abs: Double; var  proz: Double);
Begin
  abs := MaxDiff;
  proz := MaxDiffProz;
End;

procedure ResetError;
Begin
  MaxDiff := 0; MaxDiffProz := 0;
End;

End.
```

5.3 Das Controllerprogramm

```
START      ISA_START     INTROSNG                              { ISA fuer Single einstellen }
           ISA_START     SETO_SNG                              { sum Register auf 0 setzen }
           BORDER_PAR    West       32        32               { Rand Parameter setzen }
           BORDER_PAR    North      32         4
           WAIT          Ignore     Ignore    Prg_Ready
           LDI           8
           STA           acht
           LDA           blkst_w                               {  Schleifenzaehler setzen }
           STA           b_w
ILOOP      CALL          INWEST                                { westl. Rand laden }
           LDA           blkst_n                               { inneren Schleifenzaehler setzen }
           STA           b_n
JLOOP      CALL          INNORTH                               { noerdl. Rand laden }
           ISA_START     RBANK_W                               { westl. Rand zureucksetzen }
           ISA_START     WAITBANK                              { auf bank-Befehle warten }
           WAIT          N_Ready    W_Ready   Prg_Ready
           ISA_START     MAT2_SNG                              { Rechnen }
           LDA           b_n                                   { Abbruch innere Schleife }
           COMPARE       blkend_n
           JNF           JLOOP
           LDA           b_w                                   { Abbruch Schleife }
           COMPARE       blkend_w
           JNF           ILOOP
           CALL          OUTPUT                                { Ergebnis ausgeben }
           HALT          N_Ready    W_Ready   Prg_Ready        { Ende des Programms }

INWEST     BLOCK_TO_IP   West       b_w       Reset            { westl. Rand laden }
           INC           b_w                                   { letzter Block? }
           COMPARE       blkend_w
           JF            WFILL                                 { ja -> Auffuellen }
           BLOCK_TO_IP   West       b_w       Reset            { nein -> weiteren Block laden }
           INC           b_w
```

```
         JF          WENDE                          { unbedingter Sprung ans Ende }
         JNF         WENDE
WFILL    BLOCK_TO_IP West       0        Reset      { mit 0-Block fuellen }
WENDE    WAIT        Ignore     W_Ready  Ignore
         ISA_START   SBANK_W                        { Baenke umschalten }
         RET

INNORTH  CALL        BLNORTH                        { noerdl. Rand laden }
         INC         b_n                            { letzter Block }
         COMPARE     blkend_n
         JF          NFILL                          { ja -> Auffuellen }
         CALL        BLNORTH                        { nein -> weiteren Block laden }
         INC         b_n
         JF          NENDE                          { unbedingter Sprung ans Ende }
         JNF         NENDE
NFILL    WAIT        N_Ready    Ignore   Ignore
         BORDER_PAR  North      32       32         { mit 0-Block fuellen }
         BLOCK_TO_IP North      0        Reset
         WAIT        North      Ignore   Ignore
         BORDER_PAR  North      32       4
NENDE    ISA_START   SBANK_N                        { Baenke umschalten }
         RET

BLNORTH  BLOCK_START North      0                   { Laedt einen Speicherblock in  }
         LDA         acht                           { die noerdl. Randprozessorebaenke }
         STA         bcount
NLOOP    BLOCK_TO_IP North      b_n      Retain
         DEC         bcount
         COMPARE     NULL
         JNF         NLOOP
         RET

OUTPUT   ISA_START   OUT_SNG                        { Ergebnis an die Randprozessoren ausgeben }
         WAIT        Ignore     Ignore   Prg_Ready
         IP_TO_BLOCK West       0        Reset      { Bloecke schreiben }
         IP_TO_BLOCK West       1        Reset
         WAIT        Ignore     W_Ready  Ignore
         RET
```

5.4 Die ISA–Programme

```
{ Konstanten, die fuer alle ISA - Programme gebraucht werden }

const
  n = 32;               { Feldgroesse }

  sum = S[1];           { Single - Register fuer das Skalarprodukt }
  temp = S[2];          { Single - Register fuer Zwischenergebnisse }

  A_index = R27;        { Index-Register fuer je einen Eintrag von A bzw. B  }
  B_index = R31;        {     damit kann S[8] nicht mehr benutzt werden !!!! }

  CA_index = CR27;      { Konstanten fuer die kombinierte Addressierung von }
  CB_index = CR31;      { Kommunikations- und Index-Register                }

  A_data = S[6];        { Daten-Register fuer je einen Eintrag von A bzw. B }
  B_data = S[7];

  one: Selector = 1^n;  { Selector fuer ganze Zeile bzw. Spalte }

  EntryCount_sng = 16;  { Anzahl Eintraege, die von Mat2_* verarbeitet werden koennen }
                        { Wegen Laenge des ISA-Programm-Speichers weniger als 32 ! }

{Initialisierung des Feldes fuer Single-Zahlen}

program IntroSng;
```

```
{#include con_sng.inc}          { Globale Konstanten einbinden }
{#include fpsng.lib}            { Single - Operationen }

Begin(n)
  Intro_sng;
End.

{Initialisiert das Summenregister mit 0}

program Set0_sng;

{#include con_sng.inc}          { Globale Konstanten einbinden }
{#include fpsng.lib}            { Single - Operationen }

Begin(n)
  < set0_sng sum; one; one >;
End.

{Setzt den RAM-Ptr der Westbaenke zurueck}

program RBank_w;

{#include con_sng.inc}          { Globale Konstanten einbinden }

Begin(n)
  < bank WEST, 2, 1, TRUE; one; one >;
End.

{Schaltet die West-Baenke um (d.h. vertauscht Innen- und Aussenbank)}

program SBank_w;

{#include con_sng.inc}          { Globale Konstanten einbinden }

var i: Natural;

Begin(n)
  < bank WEST, 2, 1, TRUE; one; one >;
  < bank WEST, 3, 1, TRUE; one; one >;
   { for i:=1 to n do < noop; one; one >; }
End.

{Schaltet die Nord-Baenke um (d.h. vertauscht Innen- und Aussenbank)}

program SBank_n;

{#include con_sng.inc}          { Globale Konstanten einbinden }

var i: Natural;

Begin(n)
  < bank NORTH, 2, 1, TRUE; one; one >;
  < bank NORTH, 3, 1, TRUE; one; one >;
{ for i:=1 to n do < noop; one; one >; }
End.

{Warten, bis ein Bank-Befehl das Prozessorfeld durchquert hat}

program WaitBank;

{#include con_sng.inc}          { Globale Konstanten einbinden }

var
  i: Natural;
```

```
Begin(n)
  for i:=1 to n do
    < noop; one; one >;
End.

{Multipliziert zwei Matrixteile von A und B, die sich ueber 2 Bloecke erstrecken}

program Mat2_sng;

{#include con_sng.inc}                        { Globale Konstanten einbinden }
{#include fpsng.lib}                          { Bibliothek fuer Single-Operationen einbinden }

var i, j, k: Natural;                         { Laufvariablen }

Begin(n)
  for i:=1 to EntryCount_sng do Begin

    < in WEST, CW, CA_Index; one; one >;      { Index eines Wertes der Matrix A lesen }

    < in WEST, CW, C; one; one >;             { Dummy-Wort ueberlesen }

    InBroadcast_sng(WEST, A_data, one);       { Wert einer Spalte von A ueber das Feld verteilen }

    { Alle Zeilen von B lesen }
    < bank NORTH, 2, 1, TRUE; one; one >;     { Noerdl. innere Randprozessorbank zuruecksetzen }

    for j:=1 to EntryCount_sng do Begin

      < in NORTH, CN, CB_index; one; one >;   { Index eines Wertes der Matrix B einlesen }

      < in NORTH, CN, C; one; one >;          { Dummy-Wort ueberlesen }

      InBroadcast_sng(NORTH, B_data, one);    { zugehoerigen Single-Wert der Matrix B einlesen }

      < noop; one; one >;

      { Skalarproduktschritt }
      < mult_sng A_data, B_Data, temp; one; one >;  { Multiplikation der beiden Daten durchfuehren }
      < addnrm_sng sum, temp, temp; one; one >;     { normalisiert wegen A-24 Regel 2 im Handbuch }

      { temp enthaelt nun die neue Skalarprodukt-Teilsumme }

      < sub A_Index, B_Index, 0; one; one >;  { Index-Register vergleichen }
      < ifz_sng temp, sum, sum; one; one >;   { Falls Gleichheit alte mit neuer Summe ueberschreiben
                                                sonst nicht }
    End
  End
End.

{Multipliziert zwei Matrixteile von A und B, die sich ueber 2 Bloecke erstrecken}
program Mat2_opt;

{#include con_sng.inc}                        { Globale Konstanten einbinden }
{#include fpsng.lib}                          { Bibliothek fuer Single-Operationen einbinden }

const
  A_data1 = S[3];                             { Register fuer 4 Eintraege aus der Matrix A }
  A_data2 = S[4];                             { Das vierte Element entspricht den Registern }
  A_data3 = S[5];                             { wie sie in con_sng.inc definiert sind und kann }
  A_data4 = A_data;                           { mit InBroadcast_sng geladen werden ! }

  A_index1 = R11;
  A_index2 = R15;
  A_index3 = R23;
  A_index4 = A_index;
  CA_index4 = CA_index;
```

```
{ Die Prozedur fuehrt einen Multiplikationsschritt aus.
  Die Operanden stehen in den als Parameter gegebenen Registern.
  Das Ergebnis wird zu dem Register sum addiert, falls index1 = index2.
  Das Register temp wird als Hilfsregister benoetigt.
}
procedure MultStep(index1, index2: Register; data1, data2: Single);
Begin(n)
  < mult_sng data1, data2, temp; one; one >;    { Multiplikation der beiden Daten durchfuehren }
  < addnrm_sng sum, temp, temp; one; one >;     { normalisiert wegen A-24 Regel 2 im Handbuch }

  { temp enthaelt nun die neue Skalarprodukt-Teilsumme }

  < sub index1, index2, 0; one; one >;          { Index-Register vergleichen }
  < ifz_sng temp, sum, sum; one; one >;         { Falls Gleichheit alte mit neuer Summe ueberschreiben
                                                  sonst nicht }
End;

var i, j, k: Natural;                           { Laufvariablen }

Begin(n)
  for i:=1 to EntryCount_sng/4 do Begin

    { Einlesen der 4 Eintraege von A }

    { Erster }
    < in WEST, CW, C; one; one >;               { Index eines Wertes der Matrix A lesen }
    < set C, A_index1; one; one >;
    < in WEST, CW, C; one; one >;               { Dummy-Wort ueberlesen }
    InBroadcast_sng(WEST, A_data, one);         { Wert einer Spalte von A ueber das Feld verteilen }
    < set_sng A_data, A_data1; one; one >;

    { Zweiter }
    < in WEST, CW, C; one; one >;
    < set C, A_index2; one; one >;
    < in WEST, CW, C; one; one >;
    InBroadcast_sng(WEST, A_data, one);
    < set_sng A_data, A_data2; one; one >;

    { Dritter }
    < in WEST, CW, C; one; one >;
    < set C, A_index3; one; one >;
    < in WEST, CW, C; one; one >;
    InBroadcast_sng(WEST, A_data, one);
    < set_sng A_data, A_data3; one; one >;

    { Vierter }
    < in WEST, CW, CA_index4; one; one >;
    < in WEST, CW, C; one; one >;
    InBroadcast_sng(WEST, A_data4, one);

    { Alle Zeilen von B lesen }
    < bank NORTH, 2, 1, TRUE; one; one >;       { Noerdl. innere Randprozessorbank zuruecksetzen }

    for j:=1 to EntryCount_sng do Begin

      < in NORTH, CN, CB_index; one; one >;     { Index eines Wertes der Matrix B einlesen }

      < in NORTH, CN, C; one; one >;            { Dummy-Wort ueberlesen }

      InBroadcast_sng(NORTH, B_data, one);      { zugehoerigen Single-Wert der Matrix B einlesen }

      < noop; one; one >;

      { Skalarproduktschritte }
      MultStep(A_index1, B_index, A_data1, B_data);
```

```
      MultStep(A_index2, B_index, A_data2, B_data);
      MultStep(A_index3, B_index, A_data3, B_data);
      MultStep(A_index4, B_index, A_data4, B_data);
    End
  End
End.

{Ausgabe einer fertig Multiplizierten Matrix nach Westen}

program Out_sng;

{#include con_sng.inc}                          { Globale Konstanten einbinden }
{#include fpsng.lib}                            { Bibliothek fuer Single-Operationen einbinden }

var
  i: Natural;                                  { Zaehlvariable }

Begin(n)
  Convert2IEEE_sng(n, sum);                    { Werte in das IEEE-Format konvertieren }

  < bank WEST, 2, 1, TRUE; one; one >;         { Innenbank zuruecksetzen }

  OutMat_sng(n, WEST, sum);                    { Matrix nach Westen ausgeben }

  < bank WEST, 2, 1, TRUE; one; one >;         { Innenbank zuruecksetzen }
  < bank WEST, 3, 1, TRUE; one; one >;         { Bank umschalten }

  for i:=1 to n do                             { Warten, bis Bank-Befehl durchgelaufen ist }
    < noop; one; one >;
End.
```

Literatur

[1] *Der ISATEC-Parallelrechner Systola 1024.* Handbuch des verwendeten ISA, ISATEC Soft- und Hardware GmbH, Schauenburger Str. 116, D-24118 Kiel.

[2] *PARADAY – Tag der Parallelität.* Fachhochschule Stralsund, Fachbereich Elektrotechnik, November 1996.

[3] Hans Werner Lang. *Das befehlssystolische Prozessorfeld – Architektur und Programmierung.* Dissertation, Christian-Albrechts-Universität, Kiel, 1989.

[4] Martin Middendorf, Hartmut Schmeck, Heiko Schröder und Gavin Turner. *Multiplication of matrices with different sparseness properties on dynamically reconfigurable meshes.* Artikel, Zur Veröffentlichung eingereicht, 1997.