Implementierung eines H.264 Decoders auf einer dedizierten Hardware mit mehreren parallelen Recheneinheiten

Boris Zweimüller, zboris@student.ethz.ch

23. Dezember 2005

Masterarbeit, Wintersemester 2005/2006
ETH Zürich, Institut TIK: Prof. Dr. Lothar Thiele
Siemens Schweiz AG: Dr. August Kälin, Clive Diethelm

Zusammenfassung

Im Rahmen eines Innovationsprojektes der Siemens Schweiz AG soll ein H.264 Decoder auf einer Sony Playstation 2 (PS2) implementiert werden. Da der H.264 Decodiervorgang sehr hohe Anforderungen an die Rechenleistung der Zielplatform stellt, ist eine plattformorientierte, stark optimierte Implementierung nötig, welche die Hardware der PS2 optimal ausnützt. Diese Arbeit beschäftigt sich mit der Abbildung eines H.264 Decoders auf die Hardware der PS2 und stellt eine mögliche Implementierung vor. Das Schlüsselkonzept, welches für die Interpolation von ganzen Bildern entwickelt wurde, wird im Detail präsentiert.

Zur Demonstration des implementierten Videodecoders entwickelte die Siemens Schweiz AG einen Prototypen, welcher H.264-codierte Videoströme abspielen kann. Auf die verschiedenen Komponenten dieses Prototypen wird ebenfalls kurz eingegangen.

Ich möchte mich herzlich bei allen bedanken, die sich im Rahmen dieser Arbeit für mich eingesetzt und mich unterstützt haben. Besonderer Dank geht an Herrn Dr. August Kälin, Abteilungsleiter BIC-ET, Siemens Schweiz AG, für die Ermöglichung der Durchführung dieses Projektes als Masterarbeit.

Inhaltsverzeichnis

1 Einführung

1.1 Hintergrund

Siemens führte 2004 eine Home Entertainment Lösung mit dem Namen *Simon 2.0* ein. Diese Lösung kombiniert Unterhaltungs- und Kommunikationsangebote und ermöglicht damit viele verschiedene Anwendungen wie z.B. Video-on-Demand (VoD), TV over IP (IP TV), Spiele, Videotelefonie, etc. Für dieses Unterhaltungs-Angebot entwickelte Siemens eine sogenannte Set-Top-Box, deren Hauptaufgabe die Realisierung von Video-on-Demand ist. Unter einer Set-Top-Box versteht man in der Unterhaltungselektronik ein Gerät, das an ein anderes (häufig an einen Fernseher) angeschlossen ist, um gemeinsam zusätzliche Funktionen anzubieten. Bei Video-on-Demand ist die Idee, einem Benutzer kein fixes Video resp. Fernsehprogramm anzubieten, sondern ihn sein Programm selber zusammenstellen zu lassen. Abbildung 1 zeigt schematisch die verschiedenen Komponenten der Siemens Video-on-Demand Umgebung. Es folgt eine kurze Beschreibung der Funktionalität der einzelnen Elemente:

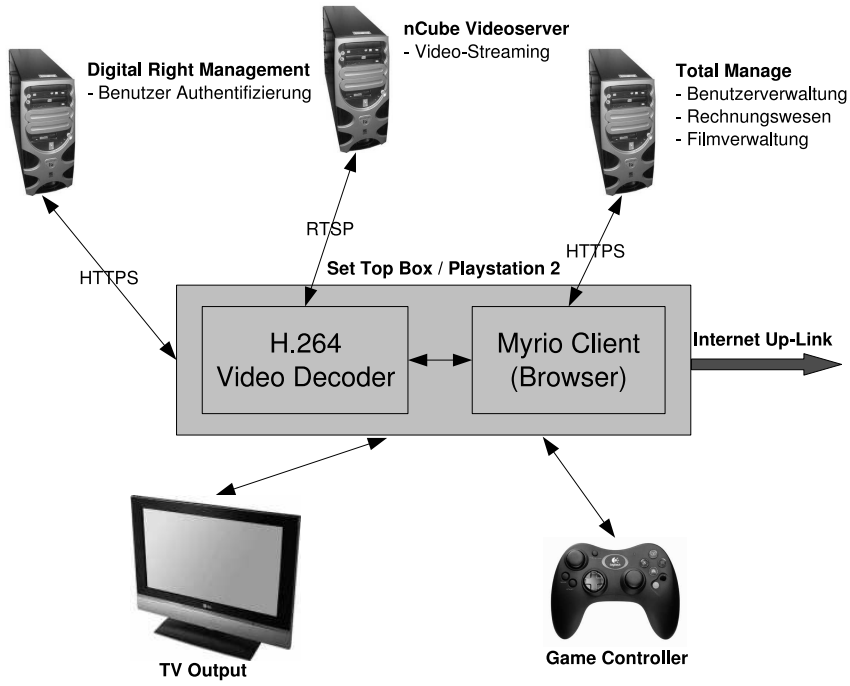


Abbildung 1: Video-on-Demand

- **Set-Top-Box**:
  Die Set-Top-Box ist mit dem Internet verbunden und enthält im We-

sentlichen zwei weitere Komponenten: Einen H.264 Decoder und Software zur Bedienung. Die via Internet (RTSP) empfangenen Videoströme werden vom H.264 Decoder decodiert und zur Ausgabe bereitgestellt. Die Software, welche auf Siemens Set-Top-Boxen verwendet wird, ist von Myrio. Sie stellt im Wesentlichen einen Client dar, mit welchem man auf den zentralen Server (Total Manage) zugreifen kann. Zur angenehmen Interaktion bietet dieser Client eine komfortable grafische Oberfläche (GUI), welche vom Benutzer mit einem Kontroller einfach zu bedienen ist.

- **Digital Right Management**:
  Diese Komponente ermöglicht einerseits eine Authentifikation des jeweiligen Benutzers und stellt andererseits Mittel zum Schutz vor unerlaubtem Kopieren eines Films zur Verfügung (z. B. verschlüsselte Übertragung eines Videostroms; entschlüsselung in Echtzeit zur Abspielzeit). Die Kommunikation mit der Set-Top-Box erfolgt via HTTPS[1].

- **nCube Videoserver**:
  Der Videoserver von nCube stellt alle angebotenen Filme in komprimiertem Format (hier H.264) bereit. Kommunikation erfolgt mittels des gängigen RTSP[2]-Protokolls.

- **Total Manage**:
  Auf diesem Server läuft die Software TotalManage, welche das Kernstück der ganzen Infrastruktur darstellt. Sie verwaltet Benutzerkonten und Filminformationen, regelt Authentifikations- und Authorisationsfragen und ermöglicht eine einfache und transparente Finanzverwaltung.

- **TV**:
  Ein Ferseher o.Ä. zur Ausgabe des Videofilms.

- **Kontroller**:
  Eine erweiterte Fernbedienung oder ein Spielkonsolenkontroller dient zur Bedienung der Software auf der Set-Top-Box.

Eine Video-on-Demand Sitzung läuft folgendermassen ab:

1. Ein Benutzer wählt aus einer Liste, welche ihm vom GUI präsentiert wird, einen Film oder Videoclip aus. Die Steuersoftware der Set-Top-Box (Myrio-Client) verbindet sich mit dem TotalManage Server, authentifiziert den Benutzer, authorisiert ihn für Video-on-Demand und kontrolliert den jeweiligen Benutzeraccount (Rechnung etc.).

[1] Hypertext Transfer Protocol Secure, eine sichere Variante von HTTP
[2] Real Time Streaming Protocol

2. Wenn der Benutzer ermächtigt ist einen Film anzuschauen, stellt der Myrio-Client eine Verbindung zum nCube Videoserver her und initiiert die Übertragung des gewünschten Filmes.

3. Sobald der Videostrom von der Set-Top-Box empfangen wird, decodiert der H.264 Decoder den Datenstrom und stellt den dekomprimierten Videofilm auf einem Fernseher dar.

4. Ein Benutzer hat die üblichen Abspiel-Kontrollmöglichkeiten über den Film (Abspielen, Pause, Schnell-Vor/Rücklauf, etc.).

Motiviert durch die Tatsache, dass weltweit mehr als 100 Millionen von Playstations 2 (PS2) des Herstellers Sony verkauft wurden, entstand bei Siemens die Idee, die PS2 als Set-Top-Box zu verwenden. Jeder PS2 Besitzer wäre somit in der Lage, seine Spielkonsole durch simples Verbinden mit dem Internet und Laden geeigneter Software in eine Set-Top-Box zu verwandeln.

1.2 Entwicklungsprojekt und Arbeitsaufteilung

Im Rahmen einer Machbarkeitsstudie der Siemens Schweiz AG wurde obige Idee umgesetzt und die nötige Software entwickelt. Die Schlüsselkomponente, welche über die Machbarkeit entscheidet, ist der H.264 Decoder, da seine Geschwindigkeit darüber entscheidet, ob ein Film in der geforderten Qualität in Echtzeit dekomprimiert und ausgegeben werden kann oder nicht. Diese Masterarbeit ist innerhalb der Entwicklung und Optimierung eines H.264 Decoders angesiedelt. Die gesamte Arbeit im Zusammenhang mit dem H.264 Decoder wurde zusammen mit Clive Diethelm, einem Entwicklungsingenieur bei Siemens Schweiz AG, angegangen. Grundsätzlich wurde die Arbeit so aufgeteilt, dass Hr. Diethelm die Optimierung des Entropiedecoders (siehe später) anging und Hr. Zweimüller eine mögliche Abbildung des H.264 Decoders auf die Hardware der PS2 erarbeitete. An dieser Stelle werden jedoch nicht nur Arbeiten des Diplomanden, sondern teilweise auch gemeinsam erarbeitete Resultate vorgestellt, da die Zusammenarbeit in einem sehr engen Rahmen stattgefunden hat.

1.3 Übersicht

Die nächsten Abschnitte beschreiben die Arbeit genauer. Da das Decodieren eines Videostroms in Echtzeit ablaufen muss, ist eine effiziente Implementierung unerlässlich. Dies kann nur erreicht werden, wenn die gesamte Hardware der PS2 optimal ausgenutzt wird. Abschnitt 2 diskutiert Möglichkeiten zur Implementierung eines H.264 Decodier-Algorithmus' auf der PS2. Bei einer genaueren Analyse der einzelnen Phasen der Decodierung hat sich herausgestellt, dass insbesondere das Interpolieren von grossen Datenmengen sehr zeitaufwändig ist. Abschnitt 3 beschreibt die Problematik der Interpolation und diskutiert verschiedene Lösungsmöglichkeiten und deren Vor-

bzw. Nachteile. Die erreichte Performanz und einige Resultate werden in Abschnitt 5 präsentiert und deren Bedeutung erklärt.

Abschliessend stellt Kapitel 6 den Prototypen vor, welcher den entwickelten H264 Decoder verwendet. Die Funktionsweise und das Zusammenspiel der einzelnen Komponenten werden kurz diskutiert. Die Hardware der Playstation 2 und der prinzipielle Ablauf der H.264 Codierung werden im Anhang besprochen.

2 Portierung eines H.264 Decoders auf die Playstation 2

Der folgende Abschnitt diskutiert Möglichkeiten zur Realisierung eines H.264 Decoders auf der Playstation 2. Das Abbilden des Algorithmus auf die Hardware der PS2 und deren Vor- und Nachteile, Einschränkungen und Eigenheiten werden ausführlich erläutert. Eine detaillierte Beschreibung der Hardwarearchitektur der P2 entnehme man Anhang A. Im nächsten Abschnitt wird der H.264 Decodier-Vorgang kurz erklärt, um die weiteren Ausführungen verständlich zu machen.

2.1 H.264 Encoder und Decoder

Dieser Abschnitt gibt einen Überblick über den H.264 Decodier-Vorgang. In weiteren Kapiteln wird wenn nötig detaillierter darauf eingegangen. Zusätzliche Informationen entnehme dem H.264 Standard ([4]) und weiterführender Literatur, z. B. [1].

2.1.1 Einführung

Die Moving Picture Experts Group (MPEG) und die Video Coding Experts Group (VCEG) entwickelten den neuen H.264 Standard, welcher eine bessere Kompression von Videoströmen als die bestehenden Kodierungsarten H.263 und MPEG-4 erreichen sollte.

H.264 definiert drei verschiedene sogenannte *Profiles* (Baseline-, Main- und Extended-Profile), welche festlegen, welche Funktionalität der Encoder bzw. der Decoder implementieren müssen, um das jeweilige Profil zu unterstützen. In dieser Beschreibung des H.264 Decoders wird nur auf diejenige Funktionalität eingegangen, die der implementierte Decoder unterstützt (Main-Profile ohne B-Frames).

2.1.2 Übersicht

Ein Videofilm besteht aus `Bildern` (Frames). Jedes Frame ist weiter aufgeteilt in einzelne `Makroblöcke` (Macroblocks), die dann vor der Übertragung komprimiert resp. codiert werden. Die üblicherweise 16 x 16 Pixel grossen Makroblöcke können individuell in `Partitionen` und `Subpartitionen` aufgeteilt sein (siehe Abbildung 2). Es gibt Partitionen der Grösse: `16 x 16`, `16 x 8`, `8 x 16` und `8 x 8`. Eine `8 x 8` Partition kann in beliebige Kombinationen von Subpartitionen der Grösse: `4 x 8`, `8 x 4`, `4 x 4` aufgeteilt sein.

Die in H.264 verwendete Kodierungsmethode nennt man `Vorhersage` (Prediction). Dies bedeutet, dass der Encoder[3] mit Hilfe anderer Makro-

[3] Ein Encoder ist die Hard- oder Software, die einen Videofilm codiert resp. komprimiert.

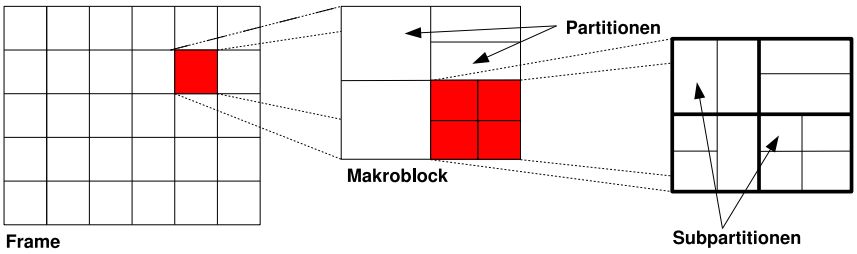


Abbildung 2: Videocodierung - Frame - Makroblöcke - Partitionen

blöcke vorhersagt resp. berechnet, wie der aktuelle Makroblock aussieht. Um die zu übertragende Datenmenge zu minimieren, wird danach lediglich der Unterschied vom vorhergehenden zum gegenwärtig berechneten Makroblock übertragen; das sogenannte `Residuum`. Wichtig ist, dass das Residuum nicht aus der Differenz des korrekten aktuellen und des vorangegangenen Makroblocks gebildet wird, sondern aus demjenigen vorangegangenen Block, der das Residuum minimiert. Dieser Vorgang ist in Abbildung 3 genauer gezeigt.

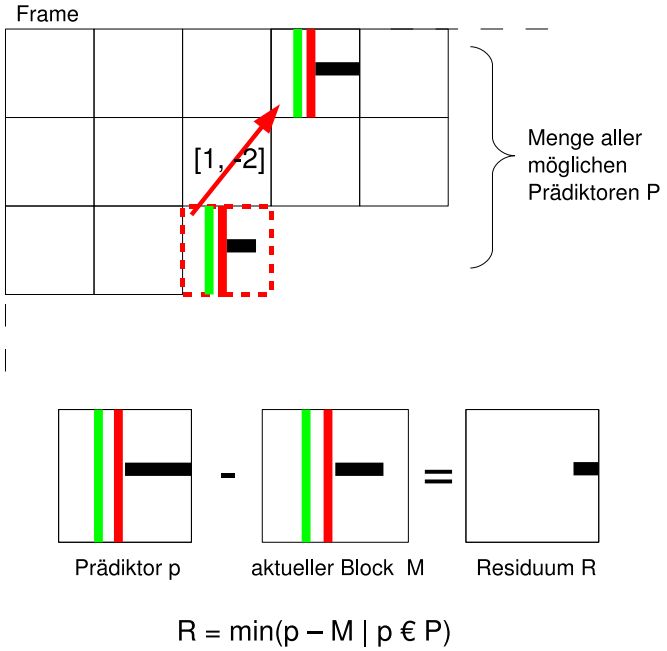


Abbildung 3: Pädiktion des aktuellen Makroblocks

Das Bild stellt einen Ausschnitt aus dem zu codierenden Frame dar. Der aktuell zu codierende Makroblock ist rot markiert. Wie bereits angedeutet,

wird nicht einfach der korrespondierende Makroblock im vorangegangenen Bild als Prädiktor verwendet, sondern derjenige, welcher das Residuum minimiert. Der Encoder durchsucht demnach alle potentiellen Makroblöcke nach dem besten Prädiktor; d.h. demjenigen, welcher das Residuum minimiert. Zusätzlich zum Residuum wird ebenfalls der `Bewegungsvektor` (Motionvector) zum Decoder übertragen. Der Decoder wird nun das Residuum decodieren, mit Hilfe des Bewegungsvektors den Prädiktor bestimmen und den aktuellen Block durch Addieren von Prädiktor und Residuum berechnen.

Der H.264-Standard unterstützt drei wesentliche Erweiterungen dieser Prädiktion: 1. Die Prädiktion findet nicht auf Makroblockebene sondern auf Partitionsebene statt und ermöglicht durch diese feinere Auflösung bessere Qualität. 2. Bewegungsvektoren zeigen nicht auf ganze Makroblöcke, sondern können an beliebige Orte im Ursprungsbild zeigen. 3. Für die Prädiktion werden `Sub-Pixeln` verwendet. In machen Fällen lässt sich ein viel besserer Prädiktor finden, wenn interpolierte Werte des Ursprungsbildes in die Suche des besten Prädiktors miteinbezogen werden. Abbildung 4 verdeutlicht diese Interpolation. Die grauen Kreise stellen Ganz-, die grünen Halb- und die roten Viertelpixelpositionen dar. Die grün-schraffierten Kreise sind sogenannte Centerpixel; sie befinden sich ebenfalls an Halbpixelpositionen, für ihre Berechnung benötigt es aber bereits berechnete umliegende Halbpixel. Zur Bestimmung des besten Prädiktors werden nun zusatzlich zu den Ganzpixel-Positionen noch interpolierte Ursprungsbilder an Halb- und Viertelpixelpositionen durchsucht. Die Halb- und Viertelpixel lassen sich relativ einfach durch das Anwenden eines FIR-Filters (6-Tab) berechnen.

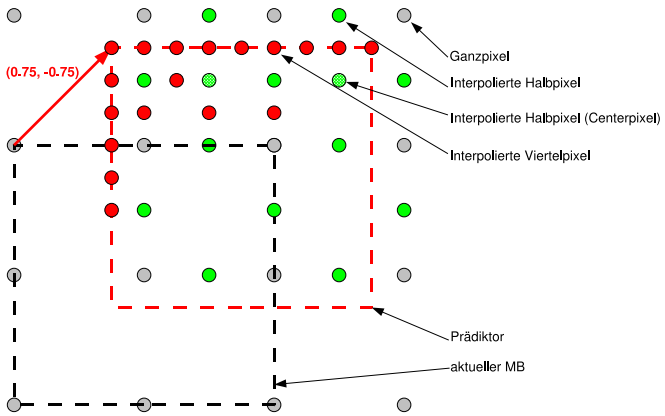


Abbildung 4: Verwendung von interpolierten Sub-Pixeln für die Prädiktion

2.1.3 Intra- und Intercodierung

Im betrachteten H.264 Decodier-Profil gibt es zwei unterschiedliche Vorhersage-Varianten. Die `Intra-Prädiktion` und die `Inter-Prädiktion`. Bei der Intra-Prädiktion werden nur Makroblöcke `innerhalb` des aktuellen Bildes, in welchem sich ebenfalls der zu codierende Makroblock befindet, zur Prädiktion benutzt. Bei der Intra-Prädiktion werden deshalb keine zusätzlichen bereits decodierten Bilder für die Prädiktion benötigt. Im Gegensatz dazu werden bei der Inter-Prädiktion Makroblöcke aus bereits decodierten Bildern verwendet. Diese `Referenzbilder` können zeitlich vor dem aktuellen als auch nach dem aktuellen Bild dargestellt werden. Falls ein Bild `X` ein anderes Bild `Y` in der Zukunft für die Prädiktion referenziert, muss der Encoder das Bild `Y` zuerst übertragen, beim Decoder muss es zwar decodiert werden, darf aber erst nach der Decodierung und Ausgabe von Bild `X` ausgegeben werden.

Korrespondierend mit den verschiedenen Varianten der Prädiktion gibt es drei verschiedene Bildtypen: `I-Frames`, `P-Frames` und `B-Frames`. I-Frames enthalten nur intra-codierte Makroblöcke, während P- und B-Frames sowohl intra- als auch inter-codierte Makroblöcke enthalten. Der Unterschied von P- und B-Frames liegt darin, dass P-Frames nur Bilder aus der Vergangenheit referenzieren, während B-Frames Bilder aus der Vergangenheit und aus der Zukunft referenzieren können. Der implementierte Decoder unterstützt vorerst nur I- und P-Frames. Wichtig ist, dass intra-codierte Bilder (I-Frames) nur Informationen aus dem aktuellen Bild zur Dekompression benutzen und nicht von anderen Bildern abhängig sind. Für das Decodieren von P-Frames sind jedoch andere vorgängig bereits decodierte Bilder, sogenannte Referenzbilder nötig.

2.1.4 Komponenten der H.264 Codierung / Decodierung

Auf die nach dem oben beschriebenen Verfahren berechneten Residuum-Makroblöcke werden dann diskrete Kosinus und Hadamard Transformationen angewendet, um die im Bild benutzten hohen und tiefen Frequenzen (Flächen und Strukturen) zu extrahieren bzw. zu separieren. In einem letzten Rechenschritt werden die Daten quantisiert, wodurch weniger wichtige Frequenzen vernachlässigt werden (Koeffizienten werden Null). Dies ist der einzige verlustbehaftete Schritt bei der H.264 Encodierung, er ermöglicht dafür eine viel bessere Kompression.

Zum Schluss werden die transformierten und quantisierten Residuum-Makroblöcke zusammen mit der notwendigen Information zur Rekonstruktion (Prädiktionsparameter, Bewegungsvektoren, Quantisierungsparameter, etc.) komprimiert und dem Decoder als Bitstream übergeben. Dieser Bitstream ist in NAL-Einheiten (Network Abstraction Layer) organisiert. Sie stellen einen Transportbehälter unabhängig vom zugrundeliegenden physi-

kalischen Netzwerk dar.

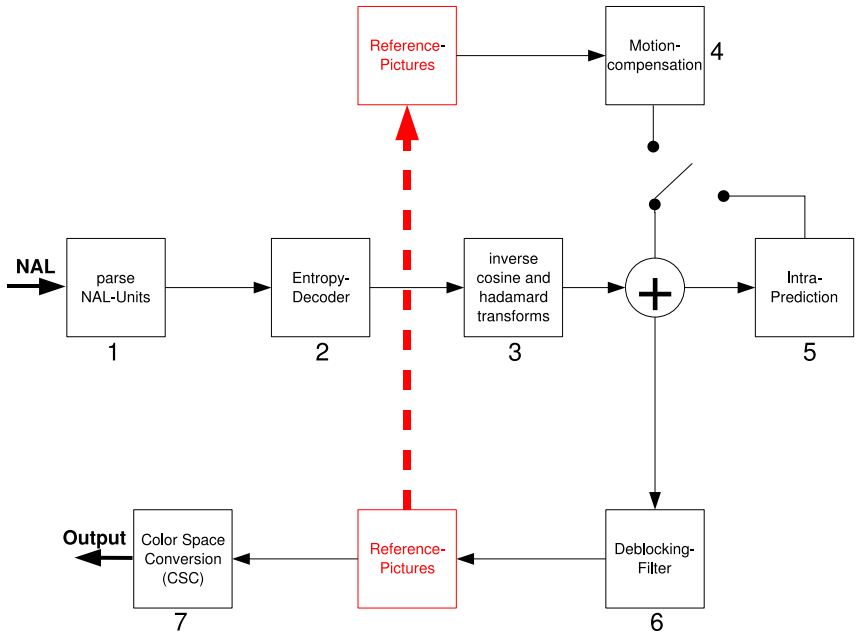


Abbildung 5: H.264 Decodier Algorithmus

Ein Decoder ist in der Lage, das ursprüngliche Frame durch Anwenden der nötigen Berechnungen auf den Bitstream wieder herzustellen. Abbildung 5 zeigt diesen Decodiervorgang schematisch. In `Schritt 1` werden die NAL-Einheiten aus dem Bitstream ausgelesen, analysiert und einige Decodier-Parameter gesetzt. `Schritt 2` dekomprimiert den Eingabestrom je nach verwendeter Kompression (CAVLC oder CABAC). Danach werden auf den Residuum-Eingangsdaten die nötigen inversen Transformationen und eine Dequantisierung (`Schritt 3`) angewendet, um die Residuum-Makroblöcke zu erhalten. Je nach Typ des übertragenen Makroblocks wurde er vom Encoder intra- oder intercodiert und muss nun vom Decoder in `Schritt 4` oder `Schritt 5` in gleicher Weise rekonstruiert werden. Nach erfolgter Berechnung des Prädiktors wird er mit in `Schritt 3` berechneten Residuum zusammengezählt. Zum Abschluss wird auf das fertige Bild ein sogenannter `Deblocking-Filter` angewendet, welcher blockweise Verzerrungen reduziert. Der Filter glättet Blockkanten und verbessert das Aussehen des decodierten Bildes (`Schritt 6`). Nach der vollständigen Decodierung wird das Bild in einer Referenzbildliste zwischengespeichert, damit es von einem nächsten Bild referenziert werden könnte (Inter-Prädiktion). Diese Referenzbilder sind in Abbildung 5 rot angedeutet. `Schritt 7` ist keine eigentliche Komponente des Decoders, jedoch nötig für die Darstellung des Bildes auf einem Ausgabegerät. Alle Bildberechnungen werden aus Effizienzgründen

(siehe 3) im YUV Farbraum[4] gemacht. Deshalb müssen die fertig decodierten Bilder vor ihrer Ausgabe in den RGB[5] Farbraum konvertiert werden. Da diese Berechnung Rechenleistung benötigt, ist es wichtig, sie ebenfalls geeignet zu optimieren.

2.2 Konzept

Bei der Portierung des H.264 Decoders auf die PS2 Plattform ist es sehr wichtig, die vorhandenen Hardware Ressourcen optimal auszunutzen, um die bestmögliche Performanz zu erreichen. Erste Tests mit einem verfügbaren Opensource H.264 Decoder [3] haben gezeigt, dass eine Implementierung, die nur auf der Haupt-CPU läuft, auch bei enormsten Optimierungen nie schnell genug wäre, um die Anforderungen an Bandbreite und Videoqualität zu erfüllen. Wie in A detailliert beschrieben, besitzt die PS2 zusätzlich zum Hauptprozessor mehrere Koprozessoren, die für die Berechnung dedizierter Aufgaben optimiert sind (Vektoreinheiten, IPU, GS). Es ist deshalb nötig, den H.264 Decodier-Algorithmus so in einzelne Teile aufzuteilen, dass diese auf mehreren Recheneinheiten gleichzeitig parallel ausgeführt werden können. Um diese Aufteilung vorzunehmen, mussten einerseits parallelisierbare Komponenten identifiziert und andererseits ausgewertet werden, wie viel Zeit durch eine Paralleliserung gewonnen werden kann. Die Unterteilung des H.264 Decodier-Algorithmus in mehrere Komponenten ist an die zu erledigenden Arbeitsschritte angelehnt und kann in 2.1 genauer nachgelesen werden. Um die Ausführungszeiten der verschiedenen Komponenten eines H.264 Decoders abzuschätzen wurde *Profiling* verwendet, dessen Funktionsweise im nächsten Abschnitt erläutert wird. Zusätzlich muss sichergestellt werden, dass die verschiedenen Einheiten effizient miteinander kommmunizieren können und der H.264 Decoder die vorhandenen Speicherbeschränkungen nicht überschreitet. Auf diese beiden Punkte wird im Anschluss ebenfalls eingegangen.

2.3 Ausgangslage

Um das Verständnis des H.264 Decoders und dessen Portierung und Optimierung auf die Playstation 2 möglichst effizient zu gestalten, wurde entschieden, eine bestehende Implementierung als Ausgangsbasis zu verwenden. Drei mögliche Kandidaten wurden dazu genauer untersucht: 1. Referenzimplementierung des Frauenhofer Instituts, 2. Opensource Implementierung FFMPEG und 3. Eine Siemens eigene Implementierung vom Forschungslabor CT in München. Das naheliegendste wäre, die H.264 Referenzimplementierung des Frauenhofer Instituts zu verwenden. Diese ist sehr nahe an den Standard angelehnt und soll einerseits zeigen, dass eine Implementierung

[4] Darstellung von Information mittels Helligkeit und Farbart
[5] RGB: Relative Anteile von Blau, Rot und Grün einer Farbe

möglich ist und andererseits zum besseren Verständnis des Standards beitragen. Bei genauerer Betrachtung erwies sich diese Referenzimplementierung jedoch als wenig hilfreich, da sie nicht optimiert ist und deshalb keine gute Performance erreicht. Entscheidungsbäume werden z. B. oft mehrmals abgearbeitet. Die Opensource Implementierung FFMPEG ist stark optimiert, dafür aber sehr schwierig zu verstehen und deshalb als Ausgangslage ebenfalls ungeeignet. Die Implementierung des Siemens Forschungslabors ist relativ optimiert aber immer noch gut verständlich und somit als Ausgangslage für weitere Optimierungen ideal.

2.4 Profiling

Ein Programm setzt sich aus vielen verschiedenen Funktionen zusammen. Mit Hilfe eines `Profils` ist es möglich herauszufinden, welche Funktionen während eines Programmablaufs wie oft aufgerufen wurden und wie lange das Ausführen jeder Funktion durchschnittlich dauerte. Ein sogenannter Profiler ist in der Lage, diese Informationen zur Laufzeit eines Programms zu extrahieren und in einer leserlichen Form aufzubereiten. Für diese Arbeit wurden Profile eines Siemens eigenen H.264 Decoders erstellt und versucht, daraus eine Aufteilung der Ausführungszeit auf die verschiedenen Komponenten des Decodier-Algorithmus abzuschätzen. Bandbreite und Auflösung des codierten Videostroms wurden den Anforderungen entsprechend gewählt.

Durch das Abbilden der Ausführungszeiten der einzelnen Funktionen auf ihre korrespondierenden Komponenten im H.264 Decoder und Multiplikation mit den jeweiligen Aufrufhäufigkeiten lässt sich die Ausführungszeit, die für die einzelnen Komponenten benötigt wird, relativ genau abschätzen. Abbildung 6 zeigt die abgeschätzten durchschnittlichen Zeiten (in Prozent), die die einzelnen Komponenten benötigen. Die prozentuale Zeit summiert sich nicht auf 100%, weil noch ein gewisser Zeitanteil für Funktionen benötigt wird, die nicht direkt einem bestimmten Rechenabschnitt zugeordnet werden können (Kontrollaufwand).

Die mittels Profiling ermittelte Verteilung der Ausführungszeit muss nun ausgewertet und folgende Aufgaben gelöst werden:

1. Für die einzelnen Arbeitsschritte des H.264 Decoders (numeriert von 1 bis 7) muss entschieden werden, auf welcher Ausführungseinheit der PS2 sie am effizientesten implementiert werden können.

2. Nachdem die Aufteilung auf die einzelnen Recheneinheiten erfolgt ist, muss eine Möglichkeit gefunden werden, den jeweiligen Arbeitsschritt parallel abzuarbeiten. Dafür ist es wichtig den Datenaustausch zwischen den einzelnen Recheneinheiten gering zu halten und die Kommunikation und Synchronisation möglichst einfach zu realisieren, damit die gesamten Abläufe einfach und somit effizient bleiben.

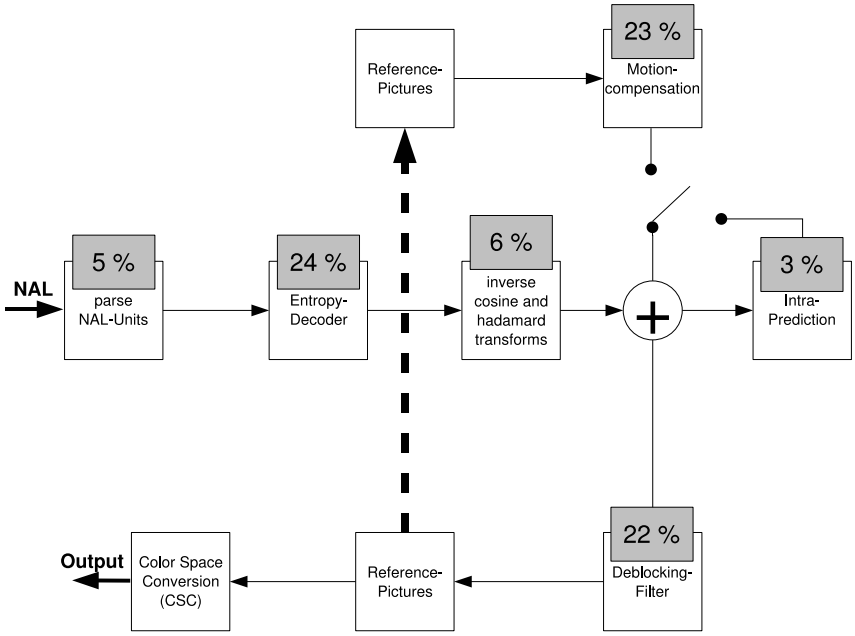


Abbildung 6: Zeitverteilung auf die verschiedenen Komponenten eines H.264 Decoders

2.5 Aufteilung auf Hardwarekomponenten der PS2

In diesem Abschnitt werden die einzelnen Komponenten des H.264 Decodier-Algorithmus - dargestellt in Abbildung 6 - genauer analysiert und entschieden, auf welcher Ausführungseinheit der Playstation 2 sie am optimalsten implementiert werden können. Detaillierte Informationen zu den einzelnen Rechenschritten entnehme man 2.1. Eine fundierte Analyse der Komplexität der einzelnen Schritte ist in [5] gegeben.

1. **1, 2: NAL-unit parsing, entropy decoder**:
   In den Rechenschritten 1 und 2 wird der codierte Datenstrom analysiert und anschliessend dekomprimiert. Die Daten können nach unterschiedlichen Verfahren (CAVLC oder CABAC - siehe 2.1) codiert sein. Vor allem für die Dekompression werden sehr viele Bitoperationen benötigt, die in der Emotion-Engine lediglich von der Haupt-CPU zur Verfügung gestellt werden. Damit ist klar, dass diese beiden Rechenschritte auf der CPU ausgeführt werden müssen. Die Ausführungszeit des Parsen und Dekomprimierens ist direkt von der Bitrate abhängig. Bei höheren Bandbreiten wird mehr Zeit für das Decodieren der eigentlichen Bilddaten, bei niedriegeren Bandbreiten mehr Zeit für das Decodieren der Bewegungsvektoren verwendet.

2. **3: Inverse hadamard and dct transform**:

Sowohl Inverse Kosinus Transformation als auch Hadamard Transformation sind in erster Linie Matrixoperationen. Diese können auf einer Vektorrecheneinheit sehr effizient ausgeführt werden. Die Ausführungszeit der inversen Transformationen und der Dequantisierung ist direkt von der Anzahl Residuum-Makroblöcke, welche keine 0-Koeffizienten haben, abhängig. Bei gleichbleibender Auflösung steigt die Anzahl Blöcke, die transformiert werden müssen, mit der Bandbreite an. Falls eine Erhöhung der Bandbreite bei gleichbleibender Auflösung stattfände, würde sich eine Optimierung zusätzlich auszahlen.

3. **4: Motion Compensation - Interpolation**:
   Weil die Motion Compensation beinahe einen Viertel der Ausführungszeit beansprucht, ist eine Parallelisierung dieses Arbeitsschrittes sehr erstrebenswert. Motion Compensation wird benötigt, um einen Makroblock des aktuellen Bildes mit Hilfe bereits decodierter Makroblöcke eines vorhergehenden Bildes zu rekonstruieren. Diese Rekonstruktion benötigt einerseits schnellen Zugriff auf decodierte Bilder (Referenzbilder) und andererseits effizientes Interpolieren, falls Halb- oder Viertelpixel des Ursprungsbildes zur Bildung des Prädiktors benötigt werden. Das Bereitstellen der Daten für die Interpolation kann mit DMA Transfers gelöst werden, während die Interpolation selbst gut auf einem Vektorrechner implementiert werden kann. Abschnitt 3 geht im Detail auf die Interpolation und die Abbildung auf die Playstation ein. Die Komplexität der Interpolation ist direkt von der Anzahl Intercodierter Makroblöcke und deshalb auch von der Auflösung abhängig. Zusätzlich muss beachtet werden, dass auf Videoströme, deren Inhalt viel zeitliche Redundanz aufweist, Intercodierung sehr effizient angewendet werden kann.

4. **5: Intra-Prediction**:
   Für die Intra-Prädiktion werden ebenfalls Interpolationen verwendet, die einen der Inter-Prädiktion ähnlichen Rechenaufwand benötigen. Dies legt eine Implementierung auf einer Vektoreinheit nahe. Weil intra-codierte Bilder nur einen sehr kleinen Teil des Videostroms ausmachen (typischerweise ¡ 5%) und somit dieser Arbeitsschritt wenig Ausführungszeit beansprucht, hat eine solche Portierung nicht höchste Priorität.

5. **6: Deblocking-Filter**:
   Zusammen mit der Motion Compensation ist der Deblocking-Filter einer der beiden rechenintensivsten Arbeitsschritte während der ganzen Decodierung, weshalb auch dieser Schritt auf einen der Koprozessoren ausgelagert werden sollte. Die einzige Möglichkeit ist hier ebenfalls den Filter auf einem der Vektorrechner zu implementieren.

6. **7: Color Space Conversion**:
   Die Konvertierung vom YUV in den RGB Farbraum lässt sich sowohl auf einer Vektoreinheit als auch mit dem Grafikprozessor realisieren. In Anbetracht der Tatsache, dass die Vektoreinheiten für andere Aufgaben benötigt werden, wird eine Implementierung auf der Grafikhardware angestrebt.

2.6 Parallelisierung der Arbeitsschritte

Dieser Abschnitt diskutiert verschiedene Möglichkeiten zur effizienten Parallelisierung von Hauptprozessor und der verschiedenen Koprozessoren. Dazu werden Vor- und Nachteile der beteiligten Hardwarekomponenten, speziell der DMA-Kontroller, die Vektoreinheiten und das MMI-Rechenwerk untersucht und besprochen. Weiterf"hrende Informationen über die Hardware der PS2 sind in Anhang A zu finden.

2.6.1 Datentransfer mit DMA

Daten mittels DMA zwischen einzelnen Komponenten auszutauschen stellt eine sehr effiziente Möglichkeit des Datentransfers dar, weil der Transfer parallel zum Ablauf anderer Berechnungen ausgeführt werden kann. Der DMA-Kontroller der PS2 ist vielfältig, was die verschiedenen Transferarten betrifft, hat aber auch einige Einschränkungen, die insbesondere beim Transfer kleiner Datenmengen ins Gewicht fallen.
Die grundlegenden Einschränkungen, die für DMA-Transfers gelten, sind folgende:

1. Die Start- und Zieladressen eines DMA-Transfers müssen 16Byte aligniert sein. Das bedeutet, dass sie an durch 16 teilbaren Speicheradressen liegen müssen. Dies führt bei schlecht alignierten Daten dazu, dass zusätzlich zu den Nutzdaten viele unnötige Daten mitransferiert werden und so den Bus ungewollt belasten. Die Problematik ist in Abbildung 7 illustriert.

   Wie in der Abbildung dargestellt, werden durch die schlechte Lage der Startadresse bei der Übertragung von $3 Bytes$ zusätzlich $29 Bytes$ unnötigerweise mitübertragen. Natürlich wären DMA-Transfers für viel grössere zusammenhängende Speicherbereiche gedacht. Beim H.264 Decodier-Algorithmus hat sich ebenfalls gezeigt, dass bereits bei der Übertragung von Makroblöcken resp. ihrer einzelnen Partitionen mittels DMA-Transfers eine enorme Menge unnötiger Daten transferiert wird, wodurch das Speichersystem unter Umständen überlastet wird. Diese Problematik ist bei der Implementierung der Interpolation aufgetreten; deshalb wird in Abschnitt 3 auf mögliche Lösungen eingegangen.

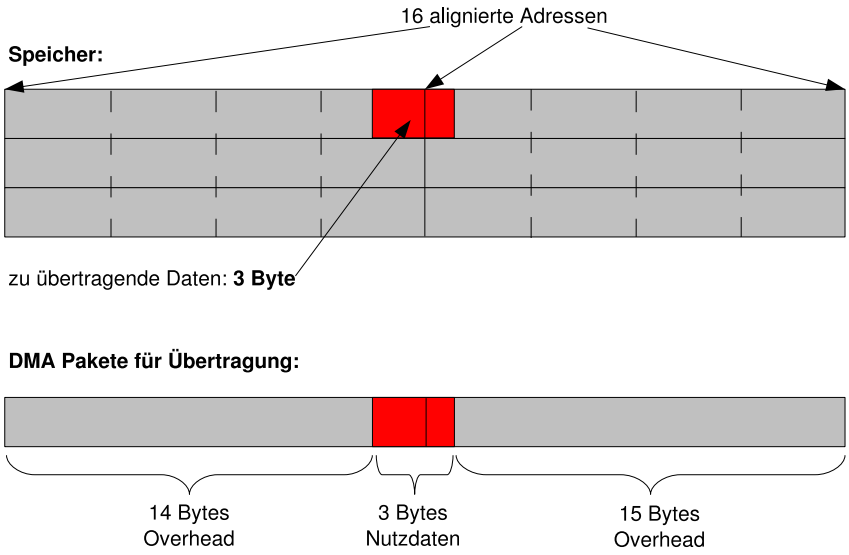


Abbildung 7: Unnötige Bus Belastung bei nicht alignierten Daten

2. Die zu übertragende Datenmenge muss ein Vielfaches von 16Byte (Quadword) sein.

Abbildung 8 gibt einen Überblick über die drei wichtigsten Transfermodi zu/von einer Komponente, welcher der DMA-Kontroller unterstützt. Die Vektoreinheit steht dabei stellvertretend für alle Koprozessoren.

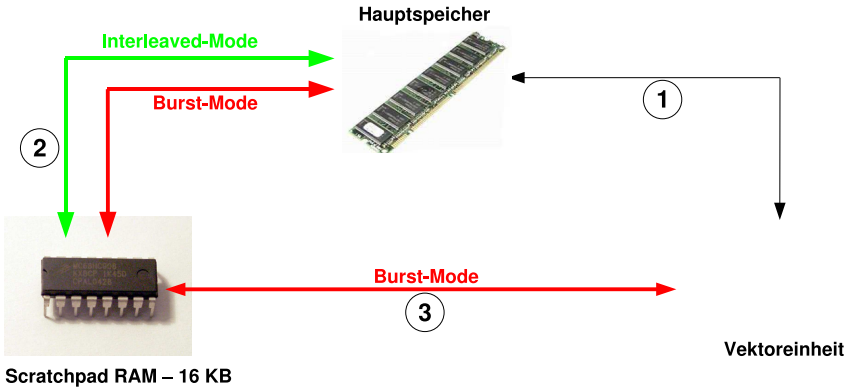


Abbildung 8: DMA-Transfer Varianten auf der Playstation 2

Diese drei Varianten sollen kurz erläutert werden:

1. **Standard-Transfer**:

Der Standard-Modus kann eine beliebige Menge im Speicher zusammenhängend angeordneter Daten sowohl vom Hauptspeicher zur Peripherie als auch in die andere Richtung transferieren. Der grosse Vorteil des Standard-Transfers ist die Möglichkeit, Daten während des Transfers zu manipulieren. Diese Funktionalität wird von einer dem DMA-Kontroller angeschlossenen Hardware-Einheit (VIF) zur Verfügung gestellt. Für diese Arbeit ist vor allem die Typumwandlung von 8 Bit auf 32 Bit Zahlen interessant, weil z. B. eine Vektoreinheit nur mit 32 Bit Zahlen umgehen kann. Weitere - wesentlich komplexere - Manipulationen entnehme man [2].

2. **Interleaved-Mode-Transfer**:

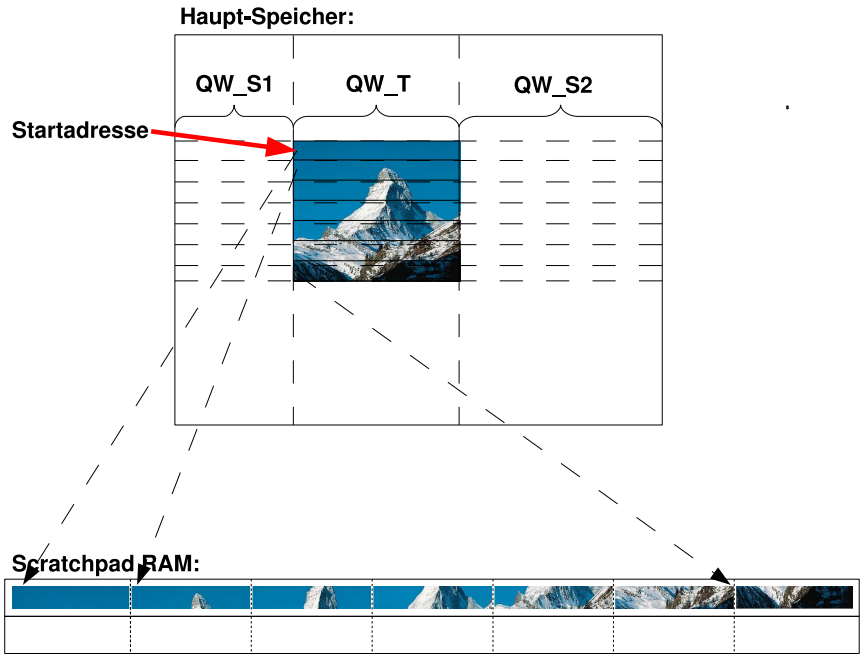


Abbildung 9: Interleaved DMA-Transfer

Abbildung 9 zeigt beispielhaft die Funktionsweise des Interleaved-DMA-Transfers. Mit diesem Modus ist es möglich, im Hauptspeicher nicht zusammenhängende Speicherbereiche streifenweise vom Hauptspeicher in den Scratchpad-Speicher und umgekehrt zu kopieren. Die einzige Anforderung ist, dass die zu kopierenden Streifen jeweils gleich gross sind und im Speicher gleich weit auseinander liegen. Für einen Transferauftrag wählt man die Adresse des ersten Streifens als Startadresse. Des Weiteren spezifiziert man die Anzahl zu tranferierender Bytes eines Streifens QW_T und die Anzahl auszulassender Bytes

QW_S1+QW_S2 bis der nächste Streifen beginnt. Abschliessend benötigt der DMA-Kontroller noch die gesamte Anzahl zu kopierender Bytes.

Für Anwendungen der Bildbearbeitung ist dieser Modus sehr hilfreich, da damit einzelne Ausschnitte eines Bildes einfach und schnell kopiert werden können. Im H.264 Deocoder lassen sich damit z. B. einzelne Makroblöcke resp. Partitionen effizient vom Hauptspeicher in den Scratchpad-Speicher und zurück transferieren, wenn sie für Berechnungen zusammenhängend benötigt werden. Beachtenswert ist, dass der DMA-Kontroller diese Funktionalität in Hardware zur Verfügung stellt. Alternativ könnte die Funktionalität natürlich mit mehreren Standard-Transfers realisiert werden. Abschliessend sei darauf hingewiesen, dass der Interleaved-DMA-Transfer `nur` zwischen Scratchpad-RAM und Hauptspeicher funktioniert, für diese Verbindug aber die gleiche Performanz wie ein Burst-Mode-Transfer erreicht (siehe nächster Abschnitt)! Eine detailliertere Beschreibung dieses Modus entnehme man [2].

3. **Burst-Mode-Transfer**:
   Der Burst-Mode-Transfer-Modus stellt die schnellste Möglichkeit dar, Daten zwischen Koprozessor oder Hauptspeicher und Scratchpad-RAM auszutauschen. Der Unterschied zum Standard-Modus liegt bei der Arbitrierung[6] des Busses. Während eines Burst-Mode-Transfers wird es keiner anderen Komponente erlaubt den Bus zu benutzen. Messungen in der Praxis haben gezeigt, dass Burst-Mode-Transfers ungefähr viermal so schnell sind wie normale Transfers, dafür entfällt die Möglichkeit, Daten während des Transfers zu manipulieren. Für die Implementierung des H.264 Decoders stellen Burst-Mode-Transfers die wichtigste Möglichkeit zum Datentransfer dar, das dieser Transfer bei Weitem der Schnellste ist.

Zusammenfassend lassen sich folgende Punkte festhalten:

- Datentransfers von einem Koprozessor in den Hauptspeicher sollten nur dann mit einem Standard-Mode-Transfer realisiert werden, wenn die Daten sowohl im Quell- als auch im Zielspeicherbereich gleich angeordnet sind. Ein zeilenweises Kopieren von Daten lässt sich beispielsweise viel effizienter mit Interleaved-Mode-Transfers via Scratchpad abwickeln. Dasselbe gilt für den Transfer vom Hauptspeicher zu einem Koprozessor.

- Datentransfers vom und in den Scratchpad-Speicher sind sehr effizient, weil sie sich mit Burst-Mode-Transfers realisieren lassen. Zusätzlich erlauben diese Transfers, wie bereits erwähnt, die Verwendung

[6] Faire Zuteilung des Busses an die angeschlossenen Komponenten.

von Interleaved-Mode-Transfers, die vor allem für die Bildbearbeitung einen enormen Gewinn bringen können.

- Eine gute Alternative zum Burst-Mode-Transfer vom Scratchpad zu einem Koprozessor ist, diesen Transfer im Standard-Mode zu realisieren. Damit hat man einerseits die Möglichkeit, Daten zuerst mit Interleaved-Mode aufs Scratchpad zu kopieren, und andererseits kann man die Daten während des Weitersendens an einen Koprozessor noch manipulieren.

- Für eine vollständige Parallelisierung dürfen die DMA-Transfers nicht blockierend ausgeführt werden. Das bedeutet, dass die Haupt-CPU während des DMA-Transfers anderen Aufgaben nachgeht. Der DMA-Kontroller der PS2 ist so ausgelegt, dass vor dem Starten eines DMA-Auftrags immer erst überprüft werden muss, ob der letzte Auftrag abgeschlossen wurde. Für maximale Performanz sollte deshalb im Kontrollcode auf der Haupt-CPU immer erst dann versucht werden einen neuen Auftrag zu starten, wenn der vorhergehende mit grosser Wahrscheinlichkeit beendet ist. Der Kontrollcode sollte demnach wie folgt strukturiert sein:

```
setupdmapacket();
startDMA();

{
        ...
        do something else
        ...
}

while (isDMAfinished()); // only execute ONCE !!

// next DMA
setupdmapacket();
...
```

2.6.2 Vektoreinheiten

Die beiden in der Emotion-Engine vorhandenen Vektoreinheiten sind die vielversprechendsten funktoinalen Einheiten für die effiziente Implementierung eines H.264 Decoders. Durch ihre Fähigkeit, Operationen auf Vektoren der Länge vier in wenigen Zyklen auszuführen, ermöglichen sie dedizierte Aufgaben sehr schnell abzuarbeiten. Aufgrund der sehr einseitigen Spezialisierung sind für die Programmierung der Vektoreinheiten ebenso einige Einschränkungen vorhanden, die im Folgenden aufgelistet sind:

1. Der verfügbare Speicherplatz auf einer Vektoreinheit ist sehr knapp. Die Vektoreinheit 1 verfügt über $16KB$ Instruktions- und $16KB$ Datenspeicher, die Vektoreinheit 0 lediglich über je $4KB$. Im Idealfall wird der Vektoreinheit zu Beginn ein Programm zur Ausführung übergeben. Es ist aber ebenfalls möglich Vektorprogramme zur Laufzeit auszuwechseln, obwohl dies mit Verzögerungen verbunden ist.

2. Der Instruktionssatz einer Vektoreinheit ist beschränkt und ausgerichtet auf die Bearbeitung von 32-Bit Fliesskommazahlen in Vektorformat. Die Operanden dieser Instruktionen sind sowohl bei Rechen- als auch bei Speicherinstruktionen 128 Bit Register. Abbildung 10 zeigt ein ganz einfaches Beispiel einer Addition von zwei Vektoren.

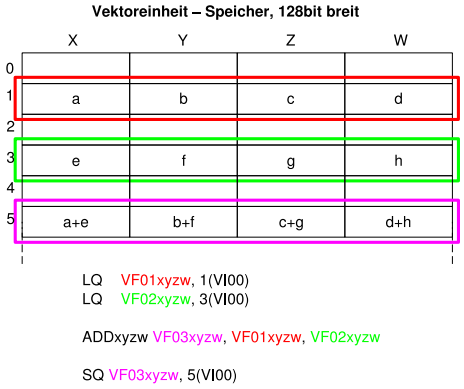


Abbildung 10: Vektoraddition auf einer Vektoreinheit

Die Register $VF01$, $VF02$ und $VF03$ sind Fliesskomma-128 Bit Register, welche die Operanden und nach der Addition das Resultat enthalten. Das Register $VI00$ ist ein Integerregister, welches fix den Wert 0 enthält. Zu beachten ist ebenfalls, dass der Speicher jeweils nur 16 Byte weise angesprochen werden kann.

Integer- und Verzweiginstruktionen werden vorwiegend zur Realisierung von Schleifen angeboten. Bitoperationen (Shift, And, Or,...) stehen nicht zur Verfügung und müssen nötigenfalls mittels Multiplikationen implementiert werden. Ebenso gibt es keine Instruktionen, die

Funktionsaufrufe vereinfachen; diese Funktionalität inklusive Aufruf-Konventionen[7] muss manuell implementiert werden.

Für eine gute Parallelisierung darf eine Berechnung auf einer Vektoreinheit nicht blockierend ausgeführt werden. Das bedeutet, dass die Haupt-CPU während der Berechnung anderen Aufgaben nachgehen kann. Für die Synchronisation gibt es zwei Möglichkeiten:

1. **Polling**
   Beim Polling wird von Zeit zu Zeit der Status der Vektoreinheit abgefragt. Sobald ihre Berechnungen beendet sind, können neue Daten gesendet werden.

2. **Interrupt-gesteuert**
   Bei der Interrupt-gesteuerten Synchronisation löst die Vektoreinheit auf der Haupt-CPU einen Interrupt aus, sobald sie ihre Berechnung beendet hat und mit weiteren Daten fortfahren könnte.

Es hat sich gezeigt, dass bei kurz dauernden dafür häufig wiederholten Rechenblöcken die Interrupt-gesteuerte Variante einen zu grossen Zeitverschleiss darstellt. Viel besser ist es, den Status der Vektoreinheit zur Sicherheit kurz abzufragen, obschon im Grunde klar ist, dass sie mit ihren Berechnungen fertig ist. Der Kontrollcode für eine gesamte Sequenz bestehend aus: 1. Daten mit DMA senden, 2. Vektoreinheit starten und 3. Resultate mit DMA zuruckholen, sähe dann wie folgt aus:

```
// send data
setupdmapacket();
startDMA();
{
        // do something else();
}
while (isDMAfinished()); // only execute ONCE !!

// start execution of vu-code
startVU();
{
        // do something else();
}
while(isVUbusy()); // only execute ONCE !!

// get data back
setupdmapacket();
```

[7] Welche Register Übergabeparameter und welche Resultate erhalten.

```
startDMA();
{
        // do something else();
}
while (isDMAfinished()); // only execute ONCE !!
...
```

2.6.3 Typumwandlung für Berechnungen auf einer Vektoreinheit

Wie bereits mehrfach erwähnt kann eine Vektoreinheit nur mit 32 Bit Fliesskommazahlen korrekt rechnen. Versuche, diese Forderung zu umgehen, haben aufgrund der proprietären Fliesskomma-Darstellung der Vektoreinheiten (nicht IEEE) fehlgeschlagen. Deshalb müssen die Eingabedaten, welche beim H.264 Decoder meist als 8 Bit Ganzzahlen vorliegen, in 32 Bit Fliesskommazahlen umgewandelt werden. Nach erfolgter Berechnung muss das Resultat ebenfalls wieder in 8 Bit Ganzzahlen umgewandelt werden. Diese Konvertierung erfordert einerseits eine Typumwandlung (Ganzzahl zu Fliesskommazahl) und andererseits eine Darstellung des Wertes mit grösserer Breite (8 Bit nach 32 Bit). Die Umwandlung von 32 Bit Ganzzahlen in 32 Bit Fliesskommazahlen und umgekehrt unterstützt die Vektoreinheit mit speziellen Instruktionen. Das Problem besteht daher in der Ausdehnung der 8 Bit Werte zu 32 Bit Werten. Dafür gibt es mehrere Lösungen, deren Vor- und Nachteile folgende sind:

1. **Keine Umwandlung**
   Theoretisch könnte man die gesamten Daten mit 32 Bit Ganzzahlen darstellen. Aufgrund des enormen Speicherbedarfs ist diese Lösung nicht praktikabel.

2. **Typumwandlung auf der CPU**
   Mittels sogenannter *Typecasts* könnte diese Umwandlung auf der CPU stattfinden. Weil aber nicht nur der Typ sondern auch die Breite der Werte ändert (8 Bit $\rightarrow$ 32 Bit), ist diese Variante um ein Vielfaches zu langsam, weil jedes einzelne Datum geladen, konvertiert und wieder gespeichert werden muss.

3. **Typumwandlung mittels DMA**
   Wie bereits in Abschnitt 2.6.1 angedeutet, können die zu übertragenden Daten während der Ausführung von Standard-Mode-Transfers manipuliert werden. Diese Manipulation kann ein Ausdehnen der Daten von 8 Bit auf 32 Bit sein.

4. **Typumwandlung auf der Vektoreinheit**
   Auf der Vektoreinheit können 8 Bit Werte mittels Umwegen und vielen Instruktionen in 32 Bit Werte umgewandelt werden. Die Umwandlung

von 32 Bit nach 8 Bit ist aber, ebenfalls aufgrund der proprietären Fliesskommazahlen-Darstellung der Vektoreinheit, nicht möglich. Mit wenigen Instruktionen ist jedoch eine Umwandlung von 32 Bit Zahlen auf 16 Bit Zahlen realisierbar. Zudem sind bei diesen 16 Bit Zahlen nur die untersten 8 Bit gesetzt. Zusammen mit der Typumwandlung auf der CPU mittels MMI stellt dieses Verfahren wohl eine der schnellsten Varianten zur Konvertierung dar (siehe nächster Punkt).

5. **Typumwandlung auf der CPU mittels MMI**
   Wie im letzten Punkt bereits angedeutet, können mittels spezieller Multi-Media-Instruktionen effiziente Typkonvertierungen durchgeführt werden. Abbildung 11 illustriert diese Umwandlung.

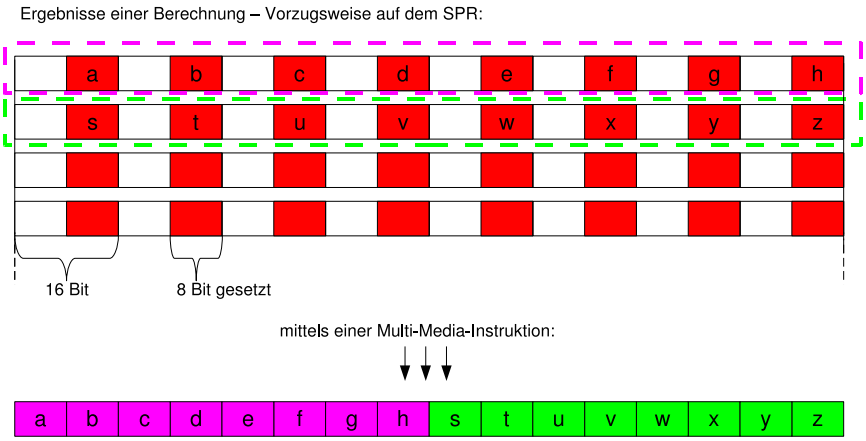


Abbildung 11: Typumwandlung mittels Vektoreinheit und MMI

Dabei werden die 32 Bit Resultate auf der Vektoreinheit in das spezielle 16 Bit Format konvertiert, bei welchem nur die untersten 8 Bit gesetzt sind. Der Multi-Media-Instruktionssatz stellt Instruktionen zur Verfügung, die auf 128 Bit Datenworten operieren. Gerade zur Manipuliation von 16 und 8 Bit Werten ist eine Vielzahl verschiedener Operationen verfügbar; eine davon passt exakt auf die gewünschte Funktionalität des Zusammenschiebens von 16 auf 8 Bit. Für die in der Abbildung dargestellte Operation werden 2 Ladeoperationen, 1 Speicheroperation und 1 Rechenoperation benötigt. Wenn diese Operationen auf Daten angewendet werden, die sich auf dem Scratchpad befinden, kann das Zusammenschieben der Daten in sehr kurzer Zeit erledigt werden.

6. **Typumwandlung mit der Grafikkarte**
   In Abschnitt 7 wird noch eine weitere Variante zur Durchführung der

Typumwandlung von 32 Bit auf 8 Bit vorgestellt, die parallel ausgeführt werden kann. Diese Variante ist jedoch noch in Entwicklung.

Abschliessend lässt sich sagen, dass die Konvertierung von 8 Bit auf 32 Bit Werte am einfachsten während eines DMA-Transfers erledigt wird. Die etwas längere Dauer des Transfers im Vergleich zu einem Burst-Mode-Transfer fällt im Gegensatz zu einer Typumwandlung auf der Haupt-CPU nicht ins Gewicht. Für die Konvertierung von 32 Bit auf 8 Bit Werte ist die kombinierte Variante von Vektoreinheit und Multi-Media-Instruktionen gegenwärtig sicher die beste. In einer weiteren Arbeit könnte sie jedoch durch eine Variante auf dem Grafikprozessor abgelöst werden.

3 Interpolation

Wie im Abschnitt 2.4 bereits erwähnt, ist die Interprädiktion (siehe 2.1, Abbildung 5) zusammen mit dem Deblocking-Filter einer der beiden rechenintensivsten Arbeitsschritte. Ein Grossteil der Interprädiktion besteht aus der Interpolation des Referenzbildes zum Bilden des optimalsten Prädiktors (siehe 2.1). Die Portierung dieser Interpolation auf eine Vektoreinheit ist deshalb unumgänglich. Dieser Abschnitt erläutert zuerst die eigentlichen Rechenaufgaben der Interpolation und diskutiert anschliessend verschiedene mögliche Lösungen zur Implementierung und Parallelisierung auf der PS2. Die Variante, die im Decoder eingesetzt wurde, wird ausführlich erklärt.

3.1 Einführung - Luma Interpolation

Wie bereits in Abschnitt 2.1 erläutert, werden die Berechnungen aus Effizienzgründen im YUV-Farbraum durchgeführt. Das Menschliche Visuelle System (MVS) reagiert auf Helligkeit feinfühliger als auf Farben. Im RGB-Farbraum sind alle drei Farbkomponenten (Rot, Grün, Blau) gleichermassen wichtig und werden deshalb alle mit der gleichen Auflösung gespeichert. Es ist aber im YUV-Farbraum möglich, ein Bild viel effizienter zu speichern, indem man die Helligkeit von den Farbwerten getrennt speichert und die Lumawerte mit einer höheren Auflösung repräsentiert. In H.264 wird das gängige $4:2:0$ Format zur Speicherung der Bildwerte benutzt. Dies bedeutet, dass für einen $16x16$ Pixel Makroblock

`16 x 16 = 256 Lumawerte` und `8 x 8 * 2 = 128 Chromawerte`

gespeichert werden. In diesem Abschnitt wird nur die Interpolation der Lumawerte genauer betrachtet und auf einer Vektoreinheit implementiert, da sie erstens die doppelte Anzahl Pixel berechnet und zweitens rein rechnerisch ungefähr die doppelte Rechenleistung benötigt.

Abbildung 12 zeigt den wesentlichen Ablauf der Rekonstruktion eines übertragenen Residuum-Makroblocks (vgl. 2.1). Der Decoder erhält als Eingabedaten das Residuum und Informationen über die Partitionierung, Bewegungsvektoren und das für die jeweilige Partition zur Prädiktion verwendete Referenzbild. Der Decoder rekonstruiert nun alle Partitionen, indem er im richtigen Referenzbild an der durch den jeweiligen Bewegungsvektor bestimmten Stelle die Referenzdaten holt, diese gegebenenfalls interpoliert und zum Schluss zum empfangenen Residuum addiert. Ob und wie die Referenzdaten interpoliert werden, wird in den letzten beiden Bits der Bewegungsvektoren übertragen.

Abbildung 13 illustriert an einem kleinen Bildausschnitt wie die Interpolation mathematisch berechnet wird. Links ist die Halbpixel-, auf der rechten Seite die Viertelpixelinterpolation dargestellt. Die horizontale oder vertikale Halbpixelinterpolation ist die Basis für die Centerpixelinterpolation, welche

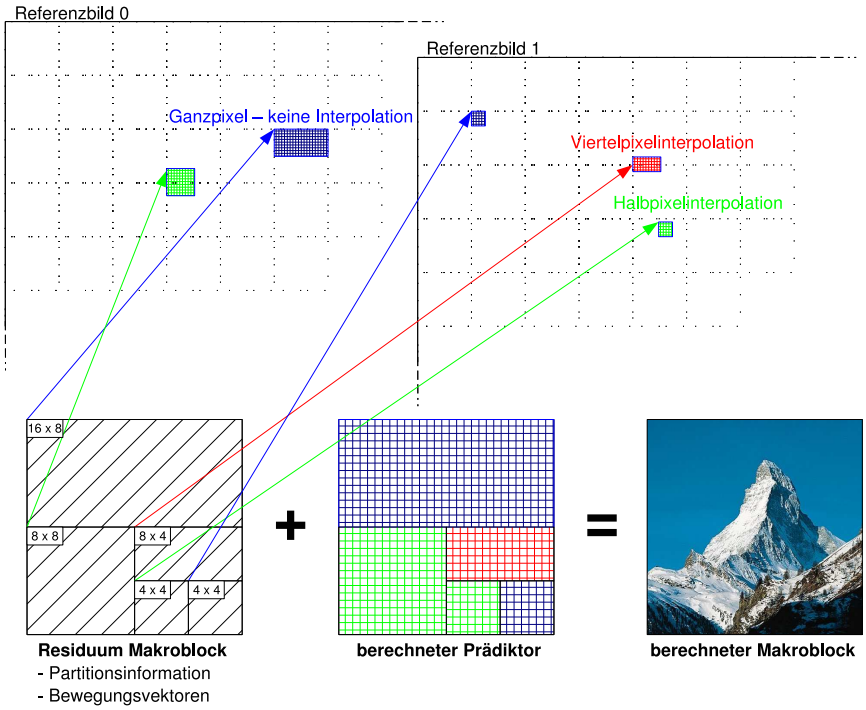
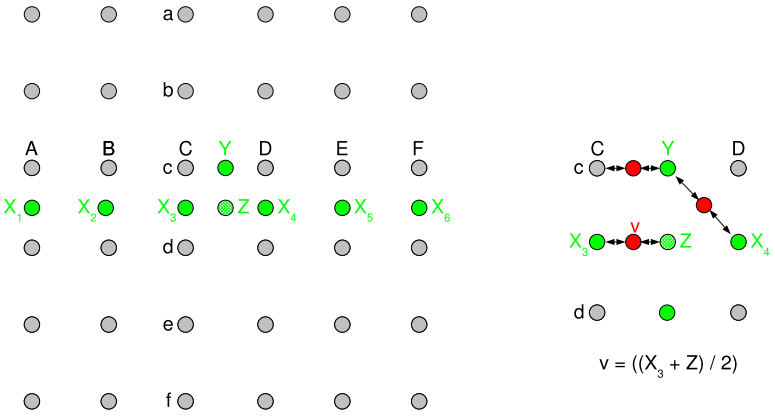


Abbildung 12: Interprädiktion - Luma-Interpolation



$Y = ((A - 5B + 20C + 20D - 5E + F) / 32)$

$X_3 = ((a - 5b + 20c + 20d - 5e + f) / 32)$

$Z = ((X_1 - 5X_2 + 20X_3 + 20X_4 - 5X_5 + X_6) / 32)$

Abbildung 13: Interprädiktion - Luma-Interpolation

wiederum für die Interpolation einiger Viertelpixel benötigt wird. Zur Berechnung der horizontalen Halbpixel - im Bild angedeutet mit Y - werden jeweils die drei linken und rechten Ganzpixel verwendet. Deshalb ist beispielsweise für die Halbpixelinterpolation einer 4 x 4 - Partition links und rechts zusätzlich ein Rand von zwei resp. drei Pixeln nötig. Dasselbe gilt bei der vertikalen Halbpixelinterpolation für den oberen und unteren Rand. Diese zusätzlich notwendigen Randpixel sind in Abbildung 14 dargestellt. Es ist für die nachfolgenden Betrachtungen entscheidend, dass diese zusätzlichen Randpixel die Anzahl der zur Interpolation notwendigen Pixel bei kleinen Partitionen vervielfachen und selbst bei 16 x 16 - Partitionen noch knapp verdoppelt (siehe folgende Tabelle).

Partitionsgrösse	Pixel ohne Rand	Pixel mit Rand
4 x 4	16	81 (9 x 9)
4 x 8 / 8 x 4	32	117 (9 x 13 / 13 x 9)
8 x 8	64	169 (13 x 13)
8 x 16 / 16 x 8	128	273 (13 x 21 / 21 x 13)
16 x 16	256	441 (21 x 21)

Für die Berechnung der Interpolation auf einer Vektoreinheit muss somit beachtet werden, dass nicht nur die eigentlichen Bilddaten, sondern zusätzlich jeweils die Randpixel zur Vektoreinheit übertragen werden.

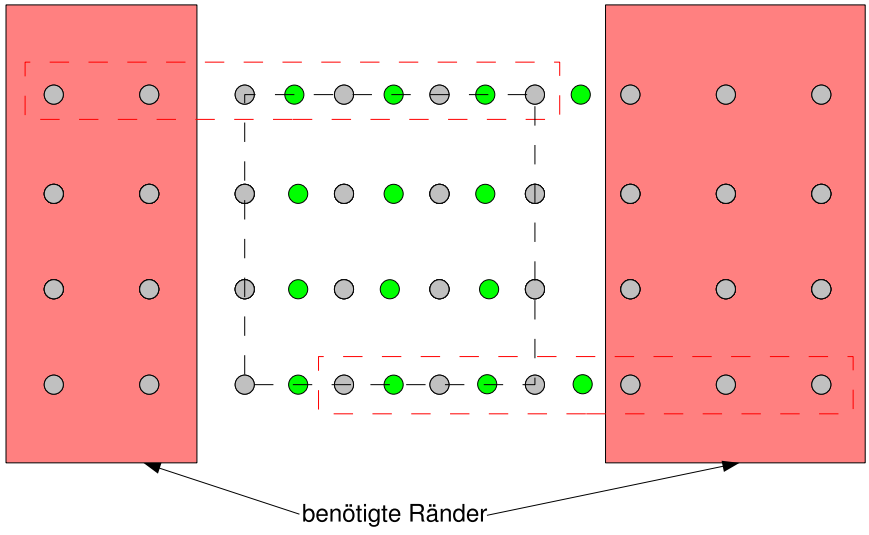


Abbildung 14: Benötigte Ränder für die Interpolation

Nachfolgend werden verschiedene Ansätze zur Lösung des Interpolations-Problems auf einer Vektoreinheit gegeben. Die ersten beiden Varianten arbeiten auf Makroblockbasis, d.h. der Datenaustausch und die Interpolation

geschehen jeweils für einen Makroblock. Die dritte Variante basiert auf einer bildweisen Interpolation, wobei zusätzlich zu den Referenzbildern interpolierte Halb- und Centerpixel-Bilder gespeichert werden.

3.2 Makroblock-basierte Interpolation

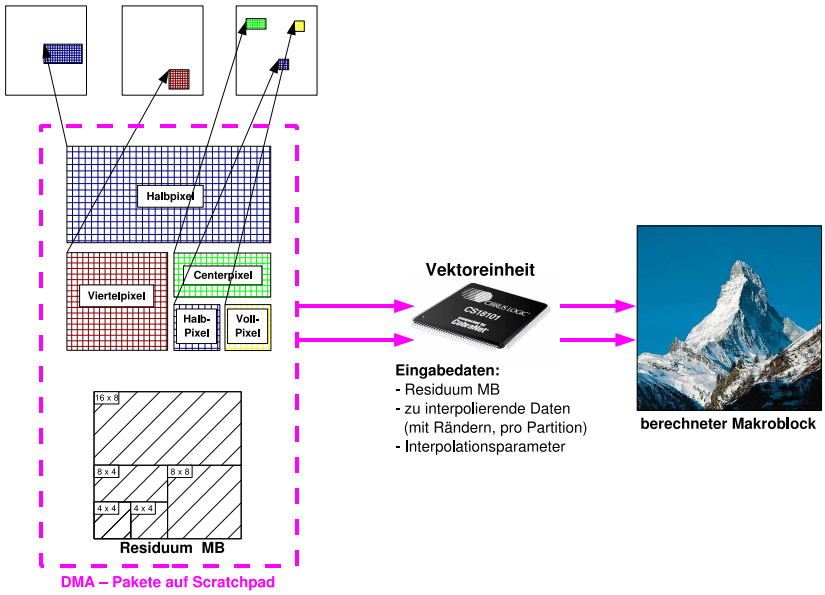


Abbildung 15: Makroblock-basierte Interpolation auf einer Vektoreinheit

Abbildung 15 zeigt schematisch die erste Variante zur makroblockweisen Interpolation auf einer Vektoreinheit. Nach Erhalt des Residuum-Makroblocks und der Parameter für die Rekonstruktion wird durch alle Partitionen des aktuellen Makroblocks iteriert und für jede die zur Interpolation nötigen Daten (inkl. Ränder) gesammelt. Diese Daten werden dann zusammen mit Parametern mittels DMA an die Vektoreinheit gesendet. Die Typumwandlung von 8 Bit auf 32 Bit geschieht entweder während des DMA-Transfers (Standard-Mode) oder auf der Vektoreinheit. Die Umwandlung von 32 Bit auf 8 Bit wird mit Vektoreinheit und Multi-Media-Instruktionen realisiert (siehe 2.6.3). Diese Lösung hat mehrere gravierende Nachteile und ist deshalb nicht empfehlenswert.

3.2.1 Einschränkungen durch DMA-Transfers

In einer einfachen Lösung müssten für jeden Makroblock DMA Transfers zum Senden der Parameter und Daten an die Vektoreinheit und zum Zurück-

transferieren des fertigen Makroblocks initiiert werden. Würde man den Deblocking-Filter im selben Schritt auf der Vektoreinheit rechnen, wäre es theoretisch sinnvoll, den Makroblock direkt an die Grafikkarte zur Ausgabe weiterzuleiten, was auf der PS2 möglich ist. Im H.264 Decoder werden aber berechnete Makroblöcke als Referenzen für nachfolgende Bilder benötigt, weshalb ein Transfer des berechneten Resultats von der Vektoreinheit zurück in den Hauptspeicher unumgänglich ist. Als Beispiel betrachte man einen Videofilm mit einer Auflösung von: $576x480$ Bildpunkten ($\frac{2}{3}D1$). Jedes Bild besteht dann aus

$$num_{MB} = 576/16 * 480/16 = 36 * 30 = 1'080 \quad (1)$$

Makroblöcken. Bei 25 Bildern pro Sekunde und jeweils zwei DMA-Aufträgen pro Makroblock entspricht dies

$$num_{DMA/s} = 1'080 * 25 * 2 = 54'000 \quad (2)$$

DMA-Aufträgen pro Sekunde. Zusätzlich ist es durch die Einschränkungen des DMA-Kontrollers nötig, die Bilddaten zuerst mittels eines Interleaved-Mode-Transfers in den Scratchpad-Speicher (siehe Anhang A) zu verschieben, und sie von dort mit Burst-Mode-Transfers an die Vektoreinheit weiterzugeben, wie in Abbildung 16 dargestellt.

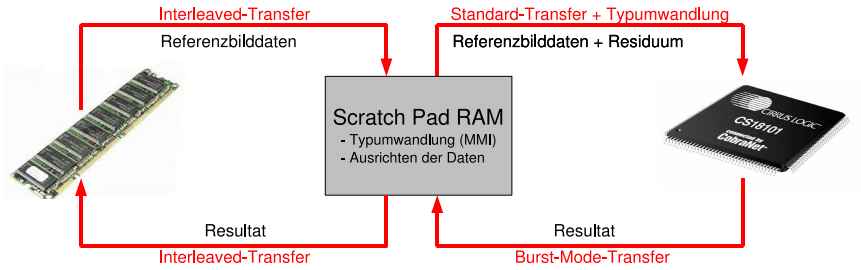


Abbildung 16: DMA-Transfers für die Makroblock-basierte Interpolation

Dies ist nötig, weil die Bilddaten für die Interpolation nicht zusammenhängend im Speicher angeordnet sind (siehe 2.6.1). Denkbar wäre auch ein Standard-Transfer vom Scratchpad in die Vektoreinheit, um die Daten von 8 Bit auf 32 Bit auszudehnen. Natürlich wäre es möglich, die einzelnen Bildzeilen mit Standard-Mode DMA-Transfers direkt vom Hauptspeicher in den Speicher der Vektoreinheit zu kopieren, dies würde aber die Anzahl Aufträge weiter vervielfachen. Mit dem Weg über den Scratchpad-Speicher verdoppelt sich die Anzahl der DMA-Aufträge pro Sekunde auf:

$$num_{DMA/s} = 54'000 * 2 = 108'000 \quad (3)$$

Diese Anzahl DMA-Aufträge liesse sich reduzieren, wenn pro Sendeauftrag mehrere Makroblöcke gesendet würden. Im komplexesten Fall hat ein Makroblock 16 4x4 Partitionen, die alle interpoliert werden müssen. Diese 16 Partitionen benötigen inkl. der Randdaten (für die Interpolation):

$$num_{refpic-4x4} = 9x9 = 81 * 16 = 1'296 \quad (4)$$

Byte Speicher, wenn alle Werte mit *8 Bit Ganzzahlen* dargestellt werden. Wie bereits erwähnt rechnen die Vektoreinheiten aber mit *32 Bit Fliesskommazahlen*. Daraus folgt, dass ein Makroblock auf einer Vektoreinheit

$$num_{refpic-4x4-32bit} = 1'296 * 4 = 5'184 \quad (5)$$

Byte Speicher für seine Darstellung braucht. Bei einer Speichergrösse von $16KB$ der Vektoreinheit 1 lassen sich zusammen mit den für die Interpolation nötigen Parametern und den Daten der Residuum-Makroblöcke nicht mehr als zwei Partitionen in einem Schritt ausführen. Wenn man noch einmal den bereits erwähnten schlechtesten Fall (alles nur 4x4 Partitionen) annimmt und zusätzlich davon ausgeht, dass die Daten jeweils vor ihrer weiteren Übertragung auf dem Scratchpad von 8 auf 32 Bit Zahlen umgewandelt werden, sind folgende Datenübertragungen für einen Makroblock notwendig (RAM = Hauptspeicher, VU = Vektoreinheit, SPR = Scratchpad):

1. Bilddaten in 8 Bit Format RAM → SPR (1'296 Byte).
2. Bilddaten in 32 Bit Format SPR → VU (5'184 Byte).
3. Residuum in 32 Bit Format SPR → VU (1'024 Byte).
4. Resultat in 32 Bit Format VU → SPR (1'024 Byte).
5. Resultat in 8 Bit Format SPR → RAM (256 Byte).

Dies ergibt bei maximal zwei gleichzeitig berechenbaren Makroblöcken:

$$num_{bytes-2macroblocks} = (1'296+5'184+1'024+1'024+256)*2 = 15'520 \quad (6)$$

Byte verteilt auf 4 DMA Aufträge (3 und 4 in einem Auftrag). Für die bereits erwähnte Auflösung von $576x480$ Bildpunkten bei 25 Bildern wären dann:

$$num_{DMA/s} = 25 * 1'080 * 2 = 54'000 \quad (7)$$

DMA-Aufträge mit einem Total von:

$$num_{bytes/s} = 25 * 1'080 * 15'520/2 = 209'520'000 \approx 200MB \quad (8)$$

Daten pro Sekunde zu transferieren. Diese Datenmenge ist für das DMA-System grundsätzlich kein Problem und würde für grössere Partitionen noch weiter reduziert. Sie kann sich jedoch drastisch erhöhen, wenn man in Betracht zieht, dass nur von auf 16 alignierten Adressen DMA-Transfers gestartet (siehe 2.6.1) und nur ein Vielfaches von 16 Byte - sogenannte QWords - übertragen werden kann. Bei kleinen Partitionen (4 x 4) müssen pro Zeile 9 Bytes transferiert werden. Im schlechtesten Fall benötigt der DMA dafür 32 Bytes. Diese überschüssig transferierten Daten verursachen einerseits eine Mehrbelastung des Busses und machen andererseits komplexe Datenbewegungs-Operationen auf der Vektoreinheit nötig. Ein noch grösserer Überschuss würde generiert, wenn die Daten zusätzlich via DMA von 8 Bit auf 32 Bit ausgedehnt würden. Zusammenfassend lässt sich sagen, dass die ursprünglich kleine Datenmenge den Bus trotzdem stark belasten könnte, wenn die Daten schlecht aligniert im Speicher liegen und deshalb für einen Transfer viele überschüssige Daten mittransferiert werden. Wenn andere Aufgaben den Bus zusätzlich benutzen, könnte dies zu einer Überbelastung führen.

Eine mögliche Lösung wäre, die Daten nach dem Transfer in den Scratchpad-Speicher mit der CPU so umzukopieren, dass sie für folgende DMA-Transfers optimal bereit liegen. Mit der Verwendung von Multi-Media-Instruktionen kann diese Aufgabe auf dem Scratchpad relativ effizient gelöst werden, weil Speicherzugriffe keine Verzögerung verursachen (siehe Anhang A).

Das grosse Problem dieser Lösung ist jedoch die hohe Anzahl DMA-Aufträgen vom Scratchpad auf die Vektoreinheit. Gerade diese sind zeitaufwändig, weil sie nicht im Burst-Mode ausgeführt werden können, sondern infolge der nötigen Manipulation der Daten (Ausdehnen auf 32 Bit) mit Standard-Mode-Transfers realisiert werden müssen. Diverse Tests haben gezeigt, dass die geforderte Anzahl Standard-Mode-Transfers vom Scratchpad auf eine Vektoreinheit eine sehr hohe Last verursachen. Zusätzlich zu den für die Interpolation benötigten DMA-Aufträgen kommen noch etliche weitere hinzu; beispielsweise die Übertragung der Daten für den Deblocking-Filter, die Intraprädiktion oder Datenverschiebungen für die Farbraum-Konvertierung.

3.2.2 Einschränkungen durch die Vektoreinheit

Die Geschwindigkeit der Vektoreinheit ist grundsätzlich ausreichend für die Aufgabe der Interpolation. Bei der vorgeschlagenen Implementierung ist die Komplexität des Vektorprogrammes jedoch ein grosser Nachteil. Das Iterieren über alle Partitionen, Typumwandlungen und Anpassungen der Daten an DMA-konformes Alignment kosten viel Zeit und hindern die Vektoreinheit daran, eine grosse Datenmenge effizient abzuarbeiten. Zusätzlich wird der Programmcode für eine Vektoreinheit beim Gebrauch vieler Kontrollstrukturen schnell sehr gross und könnte seine Grössenbeschränkung bei

zusätzlichen Erweiterungen bald überschreiten.

3.3 Makroblock-basierte Interpolation mit Pufferspeicher

Bei genauerer Analyse der Makroblock-basierten Interpolations-Variante zeigte sich, dass die Interpolation von Makroblöcken mit vielen kleinen Partitionen (4 x 4) viel Zeit kostet. Dies deshalb, weil einerseits sehr viele DMA-Transfers für das Kopieren der Referenzdaten auf das Scratchpad benötigt werden und andererseits, weil diese Daten oft sehr schlecht aligniert sind und aufwändig ausgerichtet werden müssen. Bei der H.264 Decodierung wurde erwartet, dass die zur Bildung der Prädiktoren verwendeten Referenzdaten für *aufeinanderfolgende Partitionen* nahe beieinanderliegen. Eine Idee war deshalb, anstatt für jede kleine Partition einen einzelnen DMA-Auftrag und anschliessendes Alignieren durchzuführen eine grössere Datenmenge in den Scratchpad-Speicher zu transferieren. So könnten z. B. jeweils beim Kopieren einer Partition auf das Scratchpad weitere Daten im Umkreis von 16 Pixeln Radius mittransferiert werden. Mit grosser Wahrscheinlichkeit wäre dann die nächste benötigte Referenzpartition bereits auf dem Scratchpad und weitere DMA-Transfers wären u. U. weniger häufig nötig. Das Scratchpad würde in diesem Fall als Pufferspeicher verwendet.

Eine genauere Analyse dieser Variante hat gezeigt, dass sie nur für langsame Filmsequenzen skaliert. Sobald viel Bewegung in einem Film vorhanden ist, werden die Bewegunsvektoren so gross, dass die nächsten referenzierten Partitionen nicht mehr nahe beieinander liegen. Das bedeutet, dass ein viel grösserer Umkreis als 16 Pixel um eine Partition kopiert werden müsste.

Abbildung 17 zeigt die Analyse eines Films mit total ca. 400′000 Makroblöcken. Die X-Achse zeigt die Anzahl abgearbeiteter Makroblöcke; die Y-Achse stellt die durchschnittliche Anzahl aufeinanderfolgender Zugriffe pro Makroblock dar. Der Anfang und das Ende des Films enthalten ruhige Sequenzen, während in der Mitte des Filmes Bewegung auftritt. Auf der Grafik kann man klar erkennen, dass mit dem Eintreten der Bewegung beinahe keine aufeinanderfolgenden Zugriffe mehr in einem Umkreis von 16 Pixeln des aktuellen Makroblocks liegen. In diesem Fall würde die Busbelastung steigen, da zusätzlich zu den DMA-Aufträgen pro Partition noch die Transfers für den Pufferspeicher dazukommen. Der Pufferspeicher sollte den Fall verhindern, dass viele kleine Partitionen eine Vielzahl von DMA-Transfers nötig machen. In einem Videofilm sind jedoch in erster Linie stark bewegte Sequenzen mit vielen kleinen Partitionen codiert. Diese bewegten Sequenzen führen aber wiederum dazu, dass nur sehr selten aufeinanderfolgende Partitionen im Pufferspeicher gefunden werden, da sie im Referenzbild zu weit auseinander liegen.

Theoretisch könnte der Pufferspeicher auf dem Scratchpad bis an dessen Grenzen vergrössert werden, dies ist aber wenig sinnvoll, da die Busbelastung durch die grössere transferierte Datenmenge steigt. Die Belastung

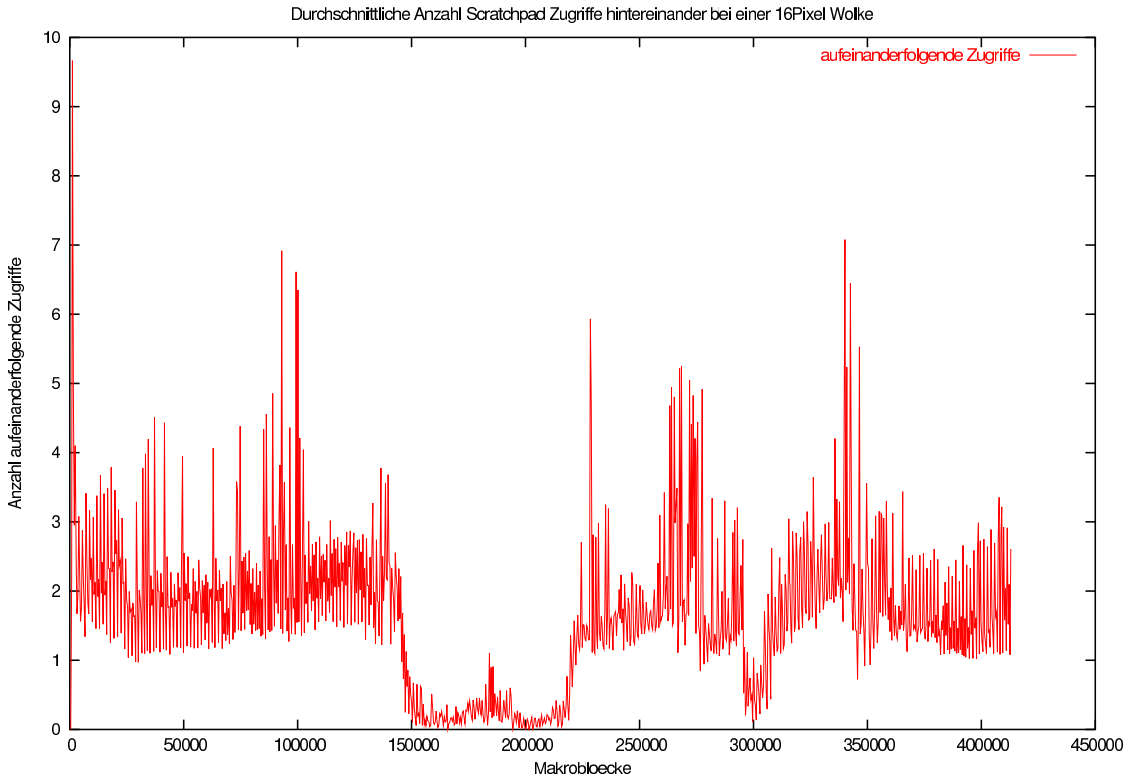


Abbildung 17: Sequentielle Zugriffshäufigkeit

steigt insbesondere dann stark an, wenn aufeinanderfolgende Partitionen nicht im Pufferspeicher zu finden sind, und dieser mit jeder Partition neu aufgebaut werden muss.

3.4 Bildbasierte Interpolation

Um einerseits die Anzahl nötiger DMA-Aufträge zu einer Vektoreinheit zu minimieren und andererseits die Komplexität des Kontrollcodes zu vereinfachen, entstand die Idee der bildbasierten Interpolation. Die Grundidee dieser Variante besteht darin, das gesamte Referenzbild zu interpolieren und zusätzlich zu speichern, um dann während der Rekonstruktion des aktuellen Makroblocks direkt auf bereits interpolierte Bildausschnitte zugreiffen zu können. Ohne komplizierten Kontrollcode ist die für die Interpolation benötigte Rechenleistung auf einer Vektoreinheit minimal. Aufgrund des grossen Speicherbedarfs zusätzlicher Bilder und der wenig aufwändigen Viertelpixelinterpolation, werden nur Halb- und Centerpixelpositionen bildweise interpoliert und gespeichert und die Interpolation der Viertelpixel auf der Haupt-CPU gerechnet. Die Tatsache, dass der Deblocking-Filter ebenfalls auf der Vektoreinheit 1 gerechnet wird, wurde ebenfalls ausgenützt. Die Resultate des Filters sind die Eingangsdaten für die Interpolation, wodurch man etliche DMA-Transfers einsparen kann. Die übrigbleibenden DMA-Aufträge sind einerseits schnelle Burst-Mode-Transfers von der Vektoreinheit in den Scratchpad-Speicher und andererseits Interleaved-Mode-Transfers vom Scratchpad-Speicher in den Hauptspeicher, um die gefilterten und interpolierte Bilddaten (Makroblöcke) in den Hauptspeicher zurückzutransferieren. Um die Anzahl an DMA-Transfers weiter zu reduzieren, werden pro Auftrag jeweils vier Makroblöcke transferiert. Der genaue Vorgang der bildweisen Interpolation, das Bestimmen der für die Prädiktion richtigen interpolierten Partitionen, das Zusammenspiel mit der Berechnung des Deblocking-Filters und der parallele Ablauf werden nun beschrieben.

3.4.1 Interpolation verbunden mit Deblocking-Filter

Durch das Verbinden von Deblocking-Filter und Interpolation können DMA-Aufträge eingespart werden, weil im gleichen Rechenschritt die Interpolation des aktuellen Bildes stattfindet. Während der Deblocking-Filter das aktuelle Bild filtert, werden die interpolierten Bilder erst bei der Decodierung der nächsten Bilder als Referenz verwendet. Die Eingabedaten müssen deshalb nur einmal an die Vektoreinheit gesendet werden, wodurch sich viele zeitaufwändige DMA-Transfers einsparen lassen. Um ihre Anzahl möglichst gering zu halten, werden jeweils vier Makroblöcke pro Auftrag an die Vektoreinheit transferiert. Abbildung 18 illustriert diesen Ablauf genauer.

Sobald vier Makroblöcke komplett fertig decodiert worden sind, werden sie zur abschliessenden Filterung und für die Interpolation an die Vektor-

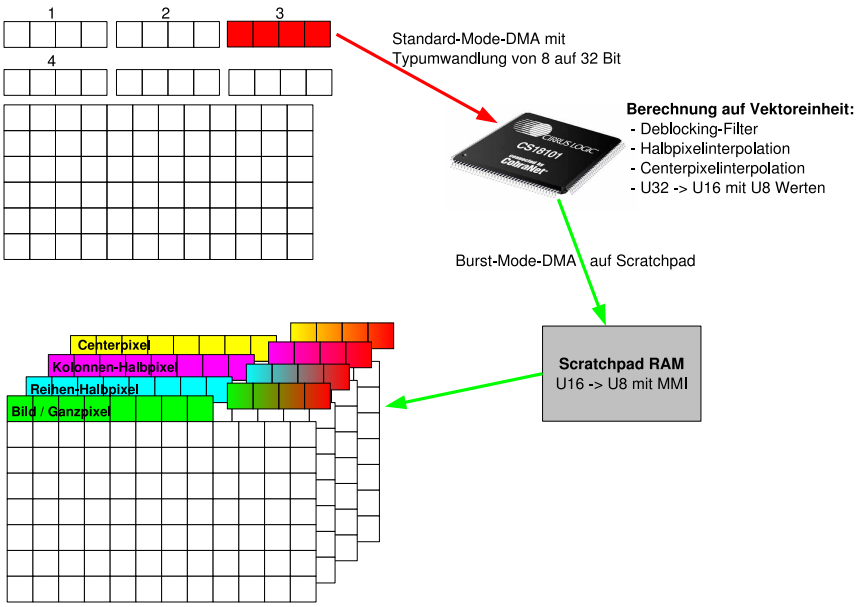


Abbildung 18: Bildbasierte Interpolation verbunden mit Deblocking-Filter

einheit gesendet. Für diesen Transfer werden Standard-Mode-Transfers vom Scratchpad in die Vektoreinheit verwendet, damit gleichzeitig die Typumwandlung von 8 auf 32 Bit vorgenommen werden kann. Der Deblocking-Filter sowie die Halb- und Centerpixelinterpolationen werden nun berechnet. Die Resultate werden auf der Vektoreinheit von 32 Bit Fliesskommazahlen auf ein spezielles 16 Bit Ganzzahlen-Format konvertiert (siehe 2.6.3), bevor sie mit Burst-Mode-Transfers auf das Scratchpad kopiert werden. Hier erfolgt noch die Konvertierung in 8 Bit Ganzzahlen, bevor die Ergebnisse mit Interleaved-Mode-Transfers in den Hauptspeicher verschoben werden (Referenzbilder und interpolierte Bilder). Die Parallelisierung und Synchronisation der Vektoreinheit mit der CPU sind in Abbildung 19 gezeigt.

Als Beispiel wird ein Bild mit 16 Makroblöcken betrachtet. In einem Zeitschritt ($t1$ bis $t5$) werden jeweils 4 Makroblöcke berechnet. Der Ablauf ist nun wie folgt: In Zeitschritt $t1$ rekonstruiert (Bestimmen des Prädiktors und Addition mit dem decodierten Residuum) die CPU die Makroblöcke 1 bis 4. In Zeitschritt $t2$ übergibt die CPU die gerechneten vier Makroblöcke der Vektoreinheit, damit diese auf ihnen den Deblocking-Filter und die Interpolation rechnen kann. Parallel dazu werden auf der CPU bereits die Makroblöcke 5 bis 8 rekonstruiert. Wichtig ist, dass die Vektoreinheit ihre Resultate noch innerhalb des Zeitschrittes $t2$ retournieren kann, damit sie in Zeitschritt $t3$ mit der Bearbeitung der Makroblöcke 5 bis 8 weiterfahren kann und keine Zeit durch Warten auf die Vektoreinheit verstreicht. In der Praxis hat sich gezeigt, dass die Vektoreinheit ihre Aufgaben schneller als

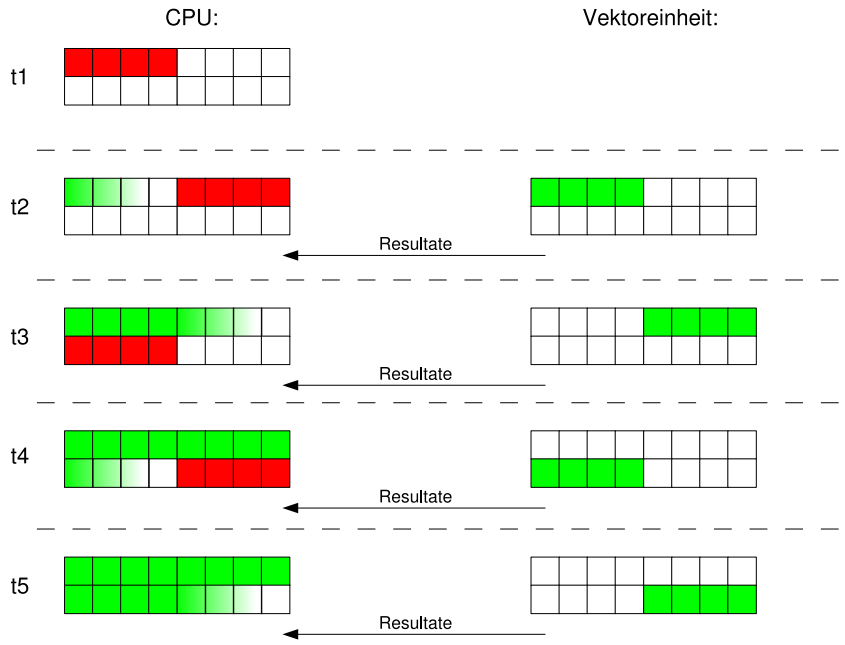


Abbildung 19: Synchronisation der Vektoreinheit mit CPU

die CPU absolviert, weshalb die Parallelisierung sehr gut funktioniert.

3.4.2 Bestimmen der Prädiktoren

Bei der Rekonstruktion eines Makroblocks werden einerseits die zu verwendenden Prädiktoren innerhalb eines Referenzbildes bestimmt und andererseits der übertragene Residuum-Makroblock decodiert. Anschliessend werden Prädiktor und Residuum addiert. Für die Bestimmung des aktuel zu verwendenden Prädiktors kann in der H.264 Decodierung eine von fünf verschiedenen Arten verwendet werden (siehe 2.1):

1. Typ 1: Ganzpixel - keine Interpolation nötig
2. Typ 2: Halbpixel - Kolonneninterpolation
3. Typ 3: Halbpixel - Zeileninterpolation
4. Typ 4: Centerpixel - Kolonnen- und Zeileninterpolation
5. Typ 5: Viertelpixel - Aus vorhergehenden Interpolationen berechnet

Wie im letzten Abschnitt erwähnt, werden in der Variante der bildbasierten Interpolation die interpolierten Bilder beider Halbpixel- und der Centerpixelinterpolation zusätzlich zum Referenzbild abgelegt. Um den aktuellen

Prädiktor zu finden, muss der Decoder den Typ jeder Partition eines Makroblocks bestimmen und in dem ihrem Typ entsprechenden Referenzbild die durch die Bewegunsvektoren referenzierten Daten suchen. Diese Daten werden dann zur Rekonstruktion mit dem Residuum zusammengezählt.

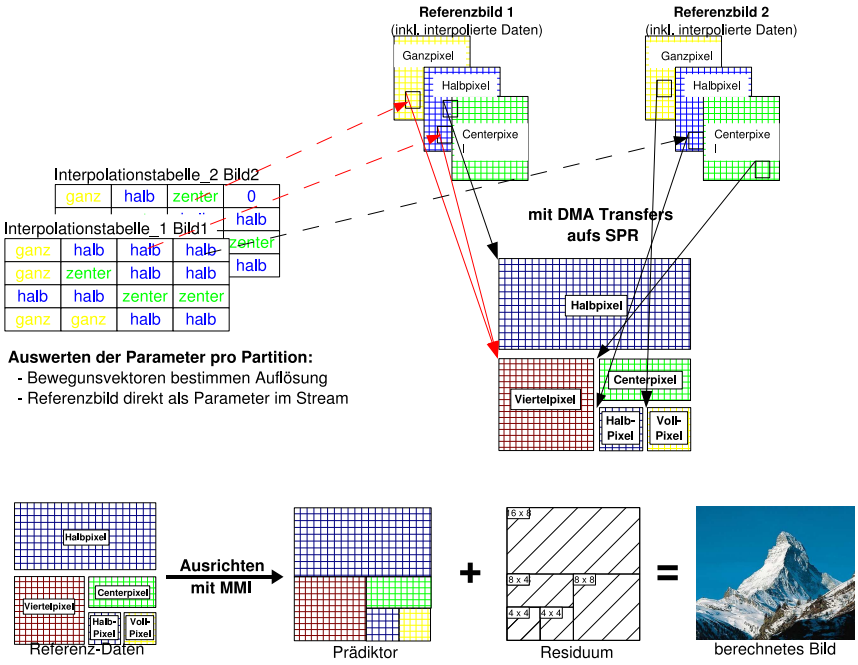


Abbildung 20: Bestimmen der korrekten Referenzpartition

Abbildung 20 zeigt den Vorgang der Rekonstruktion. Der jeweilige Typ einer Partition wird anhand ihrer Bewegungsvektoren identifiziert. Mit deren letzten beiden Bits wird der jeweilige Typ der vom Encoder angewendeten Interpolation codiert. Bit 0 codiert die Verwendung von Viertelpixelpositionen, Bit 1 die Verwendung von Halbpixelpositionen. Daher ist es möglich, das Auffinden der Rekonstruktions-Partitionen in einem Referenzbild sehr effizient zu realisieren:

Der Decoder verwaltet zwei zweidimensionale Felder $Interpolationtable_1$ und $Interpolationtable_2$, welche für jedes vorhandene Referenzbild existieren. Die letzten beiden Bits der Bewegungsvektoren (X und Y) werden als Index in diese Felder benutzt. Die Einträge eines Feldes sind Pointer auf die den Bewegunsvektoren entsprechenden Daten im jeweiligen Referenzbild. Beispielsweise könnten die letzten beiden Bits des Bewegunsvektors für die X-Komponente 10 und für die Y-Komponente 00 sein. Das bedeutet, dass der Bewegunsvektor in der Y-Achse auf ganze Pixelpositionen und in der X-Achse auf halbe Pixelpositionen zeigt. Deshalb würde der Pointer an der Stelle $Interpolationtable_1[0, 2]$ auf die Reihen-Halbpixelinterpolation

des jeweiligen Referenzbildes zeigen (in X-Richtung verschobene Halbpixelpositionen). Das zweite Feld wird für die Berechnung der Viertelpixelinterpolation benötigt, die, wie erwähnt, nicht bildweise durchgeführt und abgespeichert wird, weil ihre Berechnung auf der CPU mittels MMI sehr effizient ausgeführt werden kann. Die für einen Bewegunsvektor bestimmten zwei Speicherbereiche werden dann zu Viertelpixelpositionen interpoliert. In der Abbildung ist dies mit roten Pfeilen verdeutlicht. Beispielsweise könnten die letzten beiden Bits eines Bewegungsvektors für die X-Komponente 01 und für die Y-Komponente 10 sein. Das bedeutet, dass der Bewegungsvektor in der Y-Achse auf halbe Pixelpositionen und in der X-Achse auf Viertelpixelpositionen zeigt. Deshalb würde der Pointer an der Stelle $Interpolationtable_1[2,1]$ auf die Centerpixelinterpolation des jeweiligen Referenzbildes zeigen und der Pointer an der Stelle $Interpolationtable_2[2,1]$ auf die Kolonnen-Halbpixelinterpolation desselben Referenzbildes. Aus diesen beiden Bildern können die benötigten Viertelpixel berechnet werden. Abbildung 21 verdeutlicht diesen Vorgang. Der Bewegungsvektor rot dargestellt, die zur Viertelpixelinterpolation nötigen Halb- resp. Centerpixel sind violett umrandet. Im ersten Beispiel wäre der Wert von $Interpolationtable_1[0,2] = 0$, weil keine weiteren Daten für eine allfällige Viertelpixelinterpolation benötigt werden.

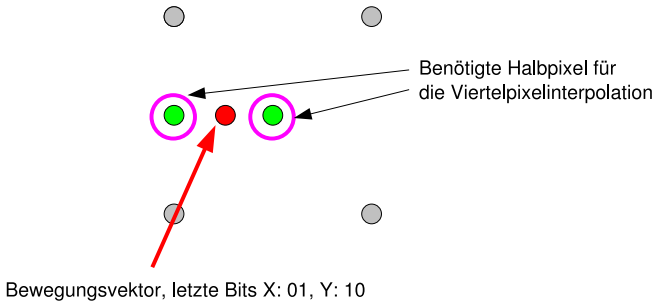


Abbildung 21: Bestimmen der korrekten Referenzpartition anhand eines Bewegungsvektors

Wenn die richtigen Bilder anhand obiger Tabellen bestimmt worden sind, werden die für die Rekonstruktion des Makroblocks nötigen Daten partitionsweise mittels Interleaved-DMA-Transfers in den Scratchpad-Speicher transferiert, bis der gesamte Prädiktor-Makroblock komplett rekonstruiert ist. Abbildung 20 zeigt diesen Kopiervorgang, welcher aufgrund folgender Überlegungen stattfindet:

1. Eine allfällige Viertelpixelinterpolation kann sehr schnell ausgeführt werden, weil Speicherzugriffe auf den Scratchpad-Speicher keine Verzögerung mit sich bringen (siehe Anhang A).

2. Das Zusammenzählen mit dem Residuum-Makroblock kann aus den in 1 genannten Gründen ebenfalls sehr schnell ausgeführt werden.

3. Die mittels Interleaved-Transfers auf das Scratchpad kopierten Speicherbereiche können aufgrund der in 2.6.1 erläuterten Probleme mit der Alignierung links und rechts viele überschüssige Daten mittransferiert haben. Für einfache weitere Berechnungen (Viertelpixelinterpolation und Addition mit dem Residuum) müssen die Daten jedoch so im Speicher liegen, dass sie mit möglichst wenigen Instruktionen verarbeitet werden können. Abbildung 22 zeigt diesen Vorgang.

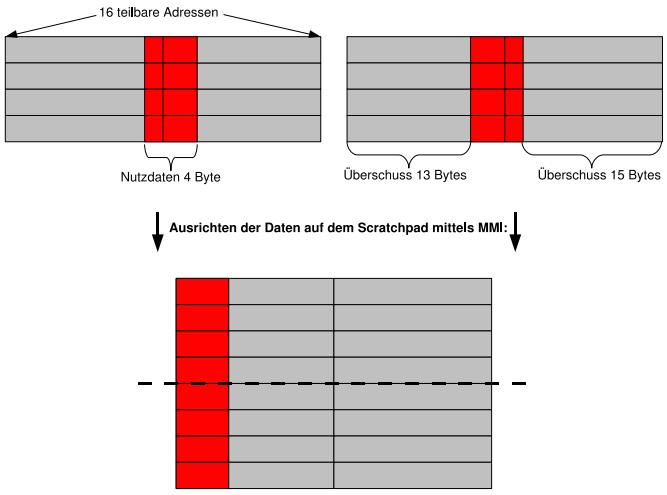


Abbildung 22: Ausrichten der Daten auf dem Scratchpad mit MMI

Im Beispielbild werden die mit DMA auf das Scratchpad transferierten Speicherbereiche mit Multi-Media-Instruktionen so umkopiert, dass sie danach für weitere Berechnungen optimal im Speicher liegen. Dieses Umkopieren geht ebenfalls aus den in 1 erläuterten Gründen sehr schnell.

Nach dem Transferieren und Ausrichten aller Partitionen sollte der gesamte Prädiktor als 16 x 16 - Block auf dem Scratchpad liegen und sich sehr effizient mit dem Residuum zusammenzählen lassen, wie in Abbildung 20 (unterer Teil) gezeigt. Danach wird der berechnete Makroblock mit einem Interleaved-DMA-Transfer vom Scratchpad zurück ins aktuelle Bild im Hauptspeicher geschrieben und mit dem nächsten Makroblock fortgefahren.

3.4.3 Parallelisierung der DMA-Transfers

Das oben erwähnte Kopieren der einzelnen Referenz-Partitionen mit Interleaved-DMA-Transfers in den Scratchpad-Speicher sollte möglichst parallel ausgeführt werden. Dazu wird ausgenützt, dass die Dequantisierung und Reskalierung der Residuum-Makroblöcke prinzipiell parallel zur Berechnung des Prädiktors erfolgen können. Die Dequantisierung wird deshalb in viele kleinere Teilschritte unterteilt, welche jeweils dann aufgerufen werden, wenn der DMA Referenzdaten ins Scratchpad transferiert. Zusätzlich kann das ebenfalls zuvor beschriebene Alignieren der Daten auf dem Scratchpad während eines DMA-Transfers ausgeführt werden. Folgender Pseudocode soll dies für einen Makroblock mit vier 8 x 8 - Partitionen schematisch illustrieren:

```
// PARTITION 1
d1 = getDataPointersForPartition1();
interleavedDMAfromMEMtoSPR(d1, amount);

Dequantisation_part_1();

while(!DMAfinished); // ONLY EXECUTE ONCE !!


// PARTITION 2
d2 = getDataPointersForPartition2();
interleavedDMAfromMEMtoSPR(d1, amount);

Dequantisation_part_2();
alignDataOnSPR(partition_1);

while(!DMAfinished); // ONLY EXECUTE ONCE !!

// PARTITION 3
d1 = getDataPointersForPartition3();
interleavedDMAfromMEMtoSPR(d1, amount);

Dequantisation_part_3();
alignDataOnSPR(partition_2);

while(!DMAfinished); // ONLY EXECUTE ONCE !!

// PARTITION 4
d1 = getDataPointersForPartition4();
interleavedDMAfromMEMtoSPR(d1, amount);
```

```
Dequantisation_part_4();
alignDataOnSPR(partition_2);

while(!DMAfinished); // ONLY EXECUTE ONCE !!

...
```

3.4.4 Optimierungen

Folgende Optimierungen würden die Variante der bildbasierten Interpolation zusätzlich beschleunigen:

1. **Typumwandlung auf Vektoreinheit und Grafikkarte**
   Wie bereits in Abschnitt 2.6.3 diskutiert, würde eine effizientere Implementierung der Typumwandlung von 32 Bit Fliesskommazahlen auf 8 Bit Ganzzahlen eine Verbesserung des gesamten H.264 Decodier-Algorithmus' ausmachen. Eine Idee wäre, die berechneten Resultate von der Vektoreinheit direkt an die Grafikkarte weiterzugeben und mit ihr diese Typkonvertierung zu rechnen. Von der Grafikkarte würden die konvertierten 8 Bit Werte des aktuellen Bildes und der berechneten Interpolationen via DMA direkt in den Hauptspeicher zurückgeschrieben. Dadurch könnte man einige DMA-Aufträge einsparen und die Synchronisation von Vektoreinheit und CPU noch einmal wesentlich vereinfachen. Diese Implementierung ist im Moment Gegenstand der Entwicklung und noch nicht vollständig ausgereift.

2. **Viertelpixelinterpolation und Addition des Residuums auf Vektoreinheit**
   Eine weitere Beschleunigung könnte dadurch erreicht werden, dass die Addition des Residuums und die Viertelpixelinterpolation ebenfalls auf der Vektoreinheit gerechnet würden. Solange die Vektoreinheit ihre Aufgaben immer noch schnell genug abarbeiten kann, ist eine effiziente Parallelisierung möglich.

3.5 Zusammenfassung

Von den drei vorgestellten Möglichkeiten zur Interpolation wurden Variante 1 und 3 komplett und Variante 2 teilweise implementiert, um das erwartete Verhalten der Hardware in der Praxis zu testen. Dabei haben sich die Erwartungen bestätigt. Die Implementierung der Variante 1 ist lauffähig, die riesige Menge von DMA-Aufträgen aber für den Bus sehr belastend und deshalb nur für kleinere Filmauflösungen eine annehmbare Lösung. Mit der bildbasierten Interpolation können auch Filme mit grösserer Auflösung decodiert werden. Letztere Variante bietet zustzlich in Bezug auf Flexibilität

und weitere Optimierungsmöglichkeiten das grösste Potential. Details zur erreichten Decodier-Leistung sind in Abschnitt 5 gegeben.

4 Implementierung eines H.264 Decoders auf der Playstation 2

Dieser Abschnitt beschreibt, wie der gesamte H.264 Decoder im Rahmen dieses Projektes auf der Playstation 2 implementiert wurde.

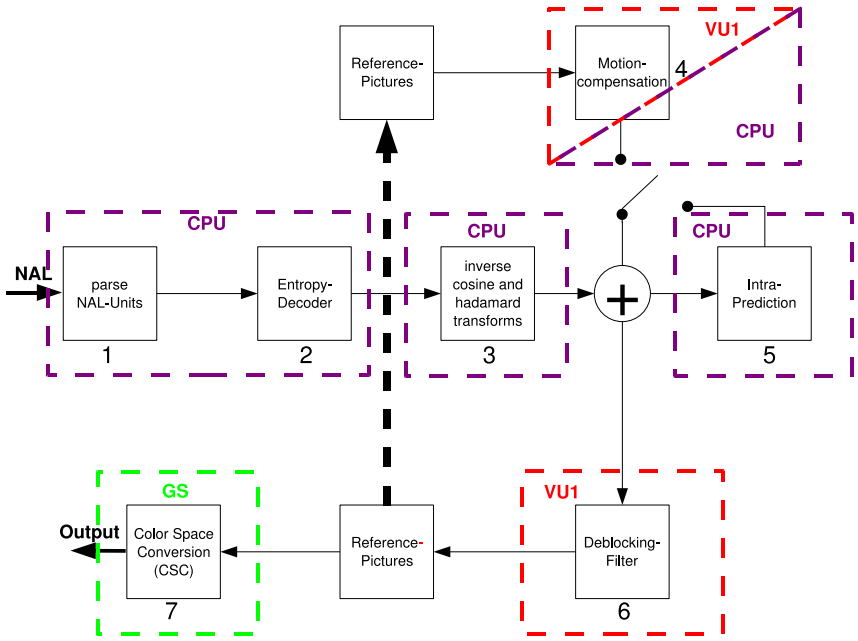


Abbildung 23: Gegenwärtige Aufteilung des H.264 Decoders auf die Recheneinheiten der PS2

Abbildung 23 zeigt schematisch, wie die verschiedenen Rechenschritte des H.264 Decoders in der gegenwärtigen Version auf die Recheneinheiten der Playstation 2 abgebildet wurden. Die ersten drei Rechenschritte, das `Analysieren` und `Dekomprimieren` des Videostroms (`Schritt 1 und 2`), die `inversen Transformationen` und die `Dequantisierung` der Residuum-Makroblöcke (`Schritt 3`) und die `Intraprädiktion` (`Schritt 5`) erfolgen auf der Haupt CPU der Emotion-Engine. Die `Interprädiktion` (`Schritt 4`) wird grösstenteils auf Vektoreinheit 1 durchgeführt; für gewisse Datenverwaltungsaufgaben wird aber auf die MMI-Einheit der CPU zurückgegriffen. Der abschliessende `Deblocking-Filter` wird ebenfalls auf der Vektoreinheit 1 gerechnet. Die für die Ausgabe wichtige Konvertierung von YUV nach RGB ist direkt auf dem Grafikprozessor realisiert. Der gesamte Deco-

diervorgang läuft wie folgt ab:

- **NAL-unit parsing, entropy decoder**:
  Der empfangene codierte Datenstrom wird in den Schritten 1 und 2 dekomprimiert und analysiert. Der Datenstrom kann nach unterschiedlichen Verfahren (CAVLC oder CABAC - siehe 2.1) codiert sein. Da für die Dekompression sehr viele Bitoperationen nötig sind, welche in der Emotion-Engine nur von der CPU zur Verfügung gestellt werden, sind diese beiden Schritte auf der CPU implementiert.

- **Dequantisierung, IDCT und Hadamard-Transformation**:
  In einem nächsten Schritt werden die empfangenen Residuum-Makroblöcke mittels inverser Kosinus- und Hadamard-Transformationen dequantisiert (`Schritt 3`). Die entstehenden reskalierten Residuum-Makroblöcke werden nach Rekonstruktion des Prädiktor-Makroblockes in `Schritt 4` oder `Schritt 5` mit diesem zusammengezählt, um das originale, encodierte Bild zu erhalten. Makroblöcke, deren Residuum `0` ist, werden speziell markiert, um die später ausgeführte Addition einzusparen. Wichtig ist, dass die Dequantisierung der Residuum-Makroblöcke unabhängig von der Rekonstruktion des Prädiktors (Interpolation) ausgeführt werden kann und erst die beiden Ergebnisse (Prädiktor-Makroblock und Residuum-Makroblock) addiert werden. Von dieser parallelen Ausführungsmöglichkeit wird in der Interprädiktion Gebrauch gemacht.

- **Intra- oder Interprädiktion und Deblocking-Filter**
  Je nach Typ des decodierten Makroblockes (`I` oder `P`) wird ein Prädiktor mit Intra- oder Interprädiktion berechnet. Die Intraprädiktion wird auf der CPU gerechnet (`Schritt 5`), weil die dafür benötigte Ausführungszeit nur wenige Prozent der Gesamtausführungszeit ausmacht. Die Interprädiktion (`Schritt 4`) erfolgt zusammen mit der Berechnung des Deblocking-Filters nach der in Abschnitt 3.4 beschriebenen Methode der bildbasierten Interpolation.

- **Color Space Conversion**
  Die Konvertierung vom YUV- in den RGB-Farbraum wird in der gegenwärtigen Version von der Grafikkarte (GS) gerechnet. Dazu werden fertig decodierte Bilder mit DMA-Transfers direkt in den Speicher der Grafikkarte geschrieben, wo sie vor ihrer Anzeige in das RGB-Format umgewandelt werden.

5 Performanz

Das ursprüngliche Ziel des Projektes war, einen Film in *Full PAL*[8] Auflösung bei Bandbreiten zwischen 1.2 MBit und 1.6 MBit in Echtzeit zu decodieren. Die folgende Tabelle zeigt die bisher erreichte Performanz und gibt einen Anhaltspunkt für die erwartete Leistung, wenn zusätzlich die in Abschnitt 7 erläuterten Optimierungen realisiert würden. Der erste Eintrag zeigt die Performanz des ursprünglichen Decoders, welcher als Ausgangslage verwendet wurde. Ein Grossteil der erreichten Verbesserung wurde ebenfalls durch die Optimierung des Entropiedecoders erreicht. Dieser wurde im Vergleich zur Ausgangsversion um einen Faktor 10 beschleunigt.

Decodervariante	Auflösung	Bandbreite
Siemens Decoder zu Beginn	88 x 72 (6'300)	1.2 MBit
FFMPEG (PS2 Linux)	176 x 144 (25'000)	1.2 MBit
Alles auf der CPU	352 x 288 (100'000)	1.2 MBit
Gegenwärtige Lösung	448 x 368 (165'000)	1.2 MBit
Vorgeschlagene Lösung	480 x 576 (276'000)	1.2 MBit

Die Full PAL Auflösung zu erreichen würde noch viele weitere Optimierungen nötig machen und wäre an der Grenze des auf einer Playstation 2 Machbaren. Dafür müssten sicher noch weitere Optimierungen von in dieser Arbeit nicht besprochenen Arbeitsschritten, wie z. B. dem Entropiedecoder, in Angriff genommen werden.

6 Prototyp

Zur Demonstration des implementierten Videodecoders entwickelte die Siemens Schweiz AG einen Prototypen, welcher H.264-codierte Videoströme abspielen kann. Seine verschiedenen Komponenten werden in Abbildung 24 dargestellt und nun kurz erläutert:

1. **Controller**
   Der Controller verwaltet Benutzereingaben und leitet diese an die entsprechenden Komponenten weiter. Beispiele wären das Weiterleiten von Kommandos zum Starten oder Stoppen eines Films oder der Neuaufbau der grafischen Benutzeroberfläche (GUI) beim Durchsuchen einer Filmliste am Fernseher.

2. **Myrio-Client**
   Der Myrio-Client ist für die Kommunikation mit allen externen Servern (nCube-Videoserver, Total Manage, DRM, ...) zuständig. Er kann z. B. den Start eines Films initiieren und die gesendeten Daten dann an die richtige Komponente weiterleiten.

[8] Full PAL Auflösung entspricht 720 x 576 Bildpunkten.

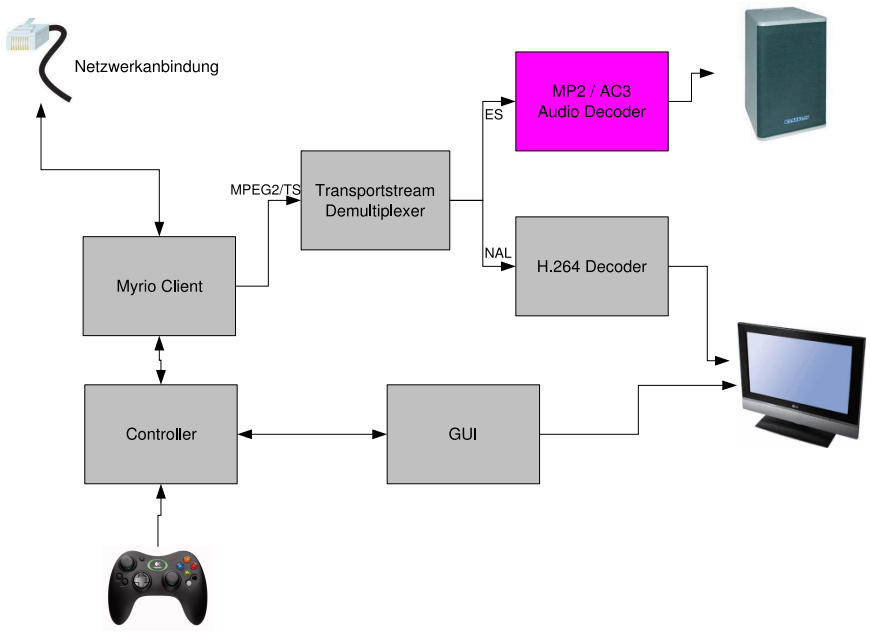


Abbildung 24: Entwickelter Prototyp

3. **GUI**
   Die grafische Benutzeroberfläche (GUI) ermöglicht es dem Benutzer, auf einfache Art und Weise mit dem System zu interagieren.

4. **Demultiplexer**
   Die von einem Videoserver empfangenen Videoströme sind im MPEG2 Transportstream-Format, welches eine einfache Möglichkeit darstellt, mehrere Datenströme (hier Audio und Video) gemeinsam gekapselt zu transferieren. Der Demultiplexer kann einen solchen Strom in seine einzelnen Teilströme aufteilen und hat in diesem Projekt die Aufgabe, Audio- und Videoströme aus dem Transportstream zu extrahieren. Die einzelnen Elementarströme werden dann entweder an den Audio- oder an den Videodecoder weitergeleitet.

5. **Audiodecoder**
   Der Audiodecoder kann MP2 encodierte Audiströme decodieren und diese an einem angeschlossenen Lautsprecher ausgeben. Es ist zu beachten, dass der gesamte Audio-Decodiervorgang auf dem IO-Prozessor implementiert ist und somit echt parallel zur Videodecodierung abläuft.

6. **H.264 Decoder**
   Der H.264 Decoder decodiert den ankommenden Videostrom und gibt die fertigen Bilder an einem angeschlossenen Fernseher aus.

7 Ausblick

Dieser Abschnitt stellt einige Optimierungmöglichkeiten vor, die in einer weiterführenden Arbeit realisiert werden könnten, um eine noch bessere Performanz zu erzielen. Abbildung 25 zeigt eine nach momentanen Erkenntnissen optimale Aufteilung der verschiedenen Rechenschritte auf die funktionalen Einheiten der PS2.

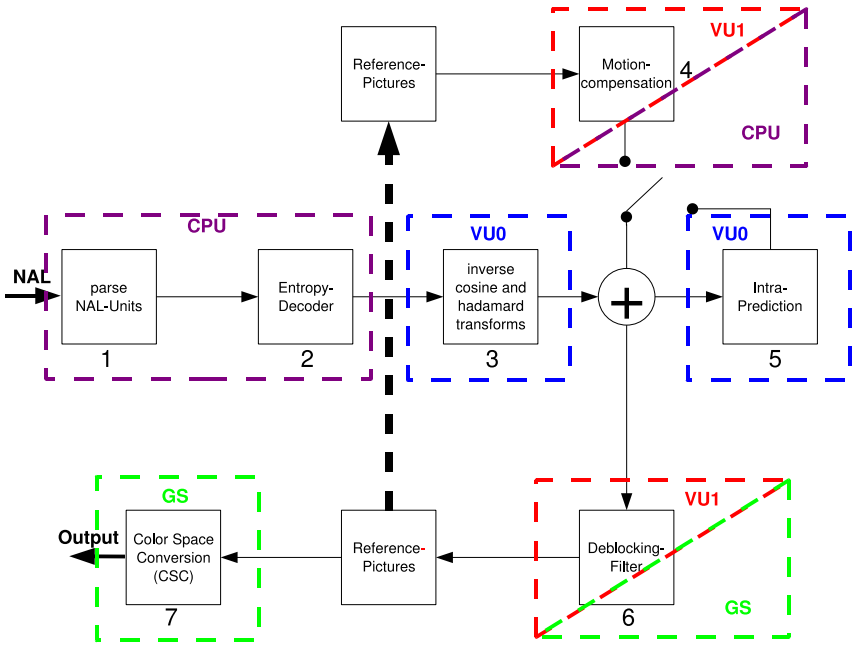


Abbildung 25: Optimale Aufteilung des H.264 Decoders auf die Recheneinheiten der PS2

1. **3: Inverse Kosinus- und Hadamard Transformation**
   Die inversen Transformationen und die Dequantisierung könnten wie bereits in Abschnitt 2.5 beschrieben auf einer Vektoreinheit realisiert werden. Diese Implementierung müsste auf der Vektoreinheit 0 ablaufen. Infolge des knappen sowohl Daten- als auch Instruktionsspeichers der Vektoreinheit 0 ist diese Implementierung nicht trivial und muss genau durchdacht werden.

2. **4: Intraprädiktion**
   Die Rekonstruktion des besten Prädiktors für einen intracodierten Makroblock könnte ebenfalls auf einer Vektoreinheit gerechnet werden. Weil die Einheit 1 bereits mit Deblocking-Filter und Interpolation für die Interprädiktion ausgelastet ist, ist eine Implementierung nur auf

Vektoreinheit 0 möglich. Weil bereits `Schritt 3` auf dieser Vektoreinheit gerechnet wird, wäre eine Auswechslung des Programmcodes zur Laufzeit unumgänglich, weil beide Schritte zusammen zu viel Code benötigen. Diese Problematik wurde bis anhin nicht untersucht und müsste ebenfalls detailliert betrachtet werden.

3. **4: Interprädiktion und Deblocking-Filter**
   In der gegenwärtigen Implementierung wird viel Zeit dafür benötigt, die berechneten Daten des Bildes und der interpolierten Bilder in 8 Bit Ganzzahlen umzuwandeln. Hier könnte ein bereits in 2.6.3 angedeutetes Konzept Abhilfe schaffen, indem diese Konvertierung von der Grafikkarte (GS) erledigt wird. Dafür wäre ein vertieftes Verständnis der GS und ihrer DMA-Möglichkeiten nötig.

A Sony Playstation 2

In diesem Abschnitt wird die Architektur der Playstation 2 im Detail beschrieben und auf die Hauptunterschiede zwischen einem Personal Computer und einer Spielekonsole hingewiesen.

A.1 Hardware-Architektur der PS2

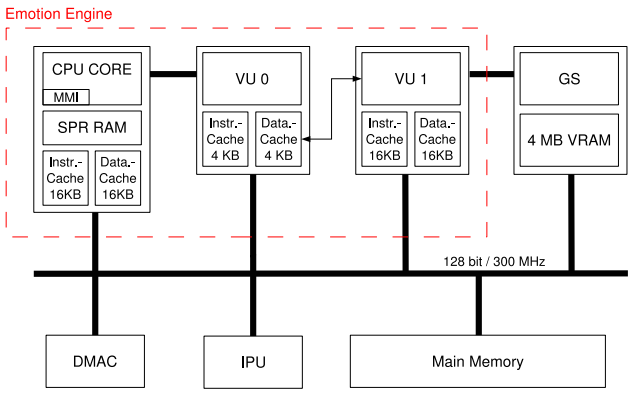


Abbildung 26: Hardware-Architektur der Playstation 2

Abbildung 26 gibt einen detaillierten Überblick über die *Emotion-Engine*, der zentralen Recheneinheit der PS2, und den anderen Hardwarekomponenten, die mit der Emotion-Engine zusammenarbeiten.

Neben einem Universal-Prozessor (MIPS 5900) enthält die Emotion-Engine weitere individuelle Recheneinheiten, die alle über spezialisierte Funktionalität verfügen. Diese Recheneinheiten wurden entwickelt, um dedizierte Operationen mit höchster Effizienz repetitiv auf grossen Datenmengen auszuführen (z. B. Spiele, Bild- und Tonverarbeitung).

Während ein Universal-Prozessor eine Vielzahl von verschiedenen Aufgaben mit durchscnittlicher bis guter Geschwindigkeit ausführen kann, ist der Emotion-Engine Prozessor geeignet, spezielle Aufgaben (Multimedia) sehr effizient zu erledigen.

A.2 Verschiedene Ausführungseinheiten der PS2

Die folgenden Funktionalen-Einheiten existieren innerhalb der Emotion-Engine:

- **CPU, Hauptrecheneinheit:**
  Die Zentrale Recheneinheit der Emotion-Engine besteht aus einem RISC Mikroprozessor mit MIPS III Instruktionssatz, welcher mit 300 MHz getaktet ist. Zusätzlich zu den Daten- und Instruktionscaches steht ein weiterer 16KB Cache - das sogenannte *scratch pad ram (SPR)* - zur Verfügung. Sowohl Lese- als auch Schreibzugriffe vom/auf das SPR sind gleich schnell wie Cachezugriffe, die keine Cachemisses[9] verursachen (1 Taktzyklus). Das SPR sollte deshalb bevorzugt für Berechnungen auf lokalen Daten verwendet werden.

- **MMI, Multimedia Instrukionen:**
  Die MMI-Recheneinheit erweitert den Standardinstruktionssatz um zusätzliche Instruktionen, welche sehr effiziente Integer-[10]Vektor Berechnungen ermöglichen. Z. B. können zwei Vektoren, welche je 16 8-bit Werte enthalten, in einem Taktzyklus zusammengezählt werden.

- **VU0, VU1 Vektor-Recheneinheiten:**
  Die mit 150MHhz getakteten Vektor-Koprozessoren der Emotion-Engine sind Fliesskomma SIMD[11] Einheiten. Sie können Vektoren bis zu einer Länge von vier in einem Taktzyklus multiplizieren und akkumulieren (MAC-Instruktionen), was in einer theoretischen Performanz von 2.4 GFLOPs resultiert (in der Praxis wurden Werte um 1 GFLOP erreicht).

  Die Vektoreinheit 0 ist via Bus direkt mit der CPU verbunden. Dies ermöglicht das Benutzen von Vektor-Rechenbefehlen im Instruktionsstrom der CPU. Die Vektoreinheit 1 hingegen ist via Bus direkt mit der Grafik-Einheit verbunden. Dies ermöglicht direktes Weitergeben von Daten ohne Umweg über die CPU.

  Beide Vektor-Einheiten besitzen schnelle Daten- und Instruktionsspeicher. Ihnen kann ein Programm und Daten zur Ausführung übergeben werden, welches die Vektoreinheit parallel zur CPU abarbeiten kann; das bedeutet, dass die CPU an anderen Berechnungen weiterarbeiten kann. Abbildung 27 illustriert diesen Vorgang.

Folgende zusäztliche funktionale Einheiten unterstützen die Emotion-Engine bei ihren Rechenaufgaben:

[9] Speicherzugriffe, die ein Nachladen des geforderten Datums aus dem Hauptspeicher nötig machen.
[10] Integer Zahlen sind Ganzzahlen
[11] Single Instruction Multiple Data

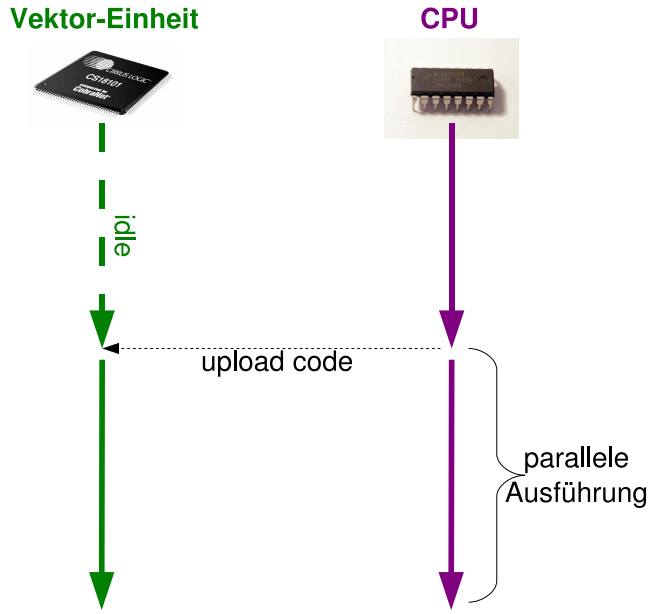


Abbildung 27: Parallele Ausführung von Vektor-Einheit und CPU

- **IPU, Image-Processing-Unit:**
  Die IPU ist ein Hardware MPEG2[12] Decoder. Dieser Baustein kann nicht programmiert werden, stellt aber die gesamte Funktionalität, die zur Decodierung von MPEG2 benötigt wird, als Funktionen zur Verfügung. Bei der H.264 Decodierung können unter Umständen einzelne dieser Funktionen nützlich sein; z. B. die Konvertierung vom YUV in den RGB Bildraum, welche zur Anzeige von Bildern auf dem Fernseher benötigt wird.

- **BUS und DMAC, Direct-Memory Access-Unit:**
  Alle oben beschriebenen Komponenten sind mittels eines 128bit breiten Busses verbunden, welcher sehr effizienten Datentransfer zwischen den einzelnen Einheiten ermöglicht. Grundsätzlich gibt es zwei Möglichkeiten, Daten zu transferieren: 1. CPU kontrollierte Datentransfers. 2. DMA (direct memory access) Datentransfer. Abbildung 28 illustriert die beiden Varianten anhand eines Datentransfers vom Hauptspeicher zu einer Vektoreinheit.

  Beim CPU-kontrollierten Datentransfer (grün) werden die Daten von der CPU erst aus dem Hauptpeicher geladen, um sie anschliessend in den Speicher der Vektoreinheit zu speichern. Die CPU ist deshalb

[12]MPEG2: Ein Kompressionsstandard für die Codierung von Video. Verwendung z. B. für DVDs oder Videostreaming.

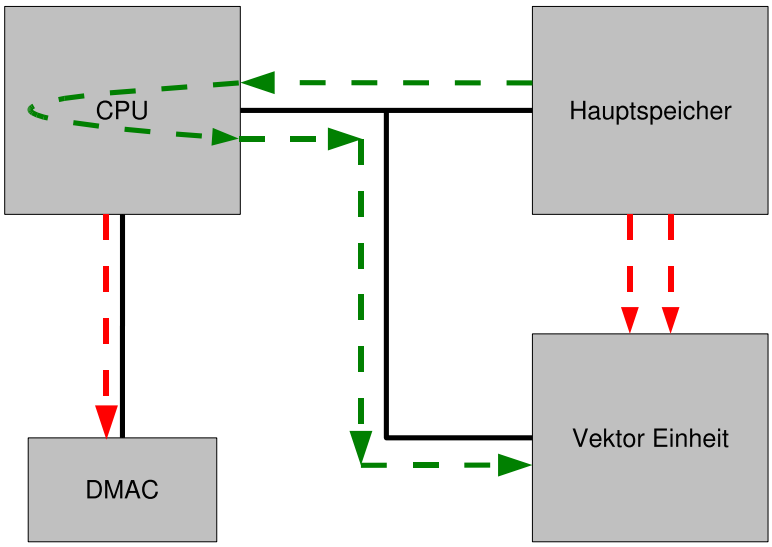


Abbildung 28: Datentransfer mit CPU oder DMA

während des ganzen Datentransfers belastet. Im Gegensatz dazu steht der Datentransfer via DMA (rot). Die CPU übergibt dem DMA Kontroller (DMAC) einen Datentransfer-Auftrag und kann sich dann weiteren Berechnungen zuwenden. Der DMA Kontroller initiiert dann parallel den gewünschten Datentransfer vom Hauptspeicher in den Speicher der Vektoreinheit und informiert - falls gewünscht - die CPU nach Abschluss des Transfers. Die gesamte Kontrolle wird somit durch den DMA Kontroller erledigt, wodurch die CPU komplett entlastet wird.

Grosse Datentranfers lassen sich somit durch geschickte Organisation beinahe vollständig parallel ausführen, so dass sich die CPU um andere Aufgaben kümmern kann.

Weitere Details über alle verschiedenen Möglichkeiten des Datenzugriffs, die mittels des DMA Kontrollers realisiert werden können, werden teilweise an geeigneter Stelle erklärt, können aber detailliert [2] entnommen werden.

- **GS, Grafik Synthesizer:**
  Der Grafikkontroller der Playstation ist verantwortlich für die Rasterung von Eingabedaten und die Ausgabe an einen Fernseher oder Bildschirm.

- **IOP, IO-Prozessor:**
  Der IO-Prozessor ist für die gesamte Ein-/Ausgabe und die Kommunikation mit angeschlossenen Geräten verantwortlich. Dieser Prozessor ist der Hauptprozessor der frühreren Playstation 1, weshalb Softwa-

re einer PS1 problemlos auf einer PS2 ausführen lässt. Der IOP ist ebenfalls programmierbar und läuft parallel zur Emotion-Engine.

Literatur

[1] H.264 and MPEG-4, Video Compression - Video Coding for Next-generation Multimedia, Iain E.G. Richardson, WILEY, 2003

[2] EE User's Manual, Copyright ©2001 Sony Compuer Entertainment Inc.

[3] http://ffmpeg.sourceforge.net/index.php - FFMPEG, ein Opensource Multimedia System, welches H.264 codierte Videoströme decodieren kann.

[4] H.264 Standard - Version 4 of H.264/AVC (ITU-T Recommendation H.264 and ISO/IEC 14496-10 (MPEG-4 part 10) Advanced Video Coding)

[5] H.264/AVC Baseline Profile Decoder Complexity Analysis - M. Horowitz, A. Joch, F. Kossentini, A. Hallapuro, IEEE transactions on circuits and systems for video technology, vol. 13, no. 7, july 2003

[6] VU User's Manual, Copyright ©2001 Sony Compuer Entertainment Inc.

[7] The MPEG handbook MPEG-1, MPEG-2, MPEG-4, second edition, John Watkinson, Focal Press, 2004