XX. Używanie SIMD do przenoszenia funkcji algebry liniowej na autostradę

Konwersja funkcji matematycznej na efektywny kod SIMD może być trudna. Nie jest zawsze łatwo zobaczyć najlepszy sposób obliczenia wartości końcowej za pomocą obliczeń równoległych, ale ten rozdział pomoże Ci "myśleć SIMD". Ta część zakłada, że masz wcześniejszą wiedzę na temat instrukcji SIMD. SIMD wprowadzono krótko w poprzedniej części, ale jeśli nie znasz go zbyt dobrze, powinieneś wybrać książkę obejmującą temat od góry do dołu. W tym rozdziale przedstawię różne podstawowe funkcje algebry liniowej w asemblerze z wykorzystaniem obliczeń jedno-pływowych. Jeśli znasz tylko wewnętrzne funkcje, powinieneś być w stanie zrozumieć znaczenie funkcji asemblera, ponieważ te dwie konwencje nazewnictwa są dość podobne. Używane są pojedyncze punkty zmiennoprzecinkowe, ponieważ można wykonać dwa razy więcej obliczeń niż przy precyzji podwójnego ruchu. Kompilatory są coraz lepsze z każdym dniem, ale wciąż są straszne w instrukcji planowania skomplikowanego kodu. Tutaj jest wymagane strojenie ASM. Pod względem rozmiaru kodu funkcje mogą być podobne, ale rzeczywista kolejność ustawiania równań robi ogromną różnicę. Ponieważ instrukcje te są kluczową częścią pętli kodu każdej gry, konieczne jest pisanie najszybszych możliwych funkcji. Dostępnych jest kilka bibliotek (takich jak te zawarte w D3D), ale szybka inspekcja kodu ujawni, że bieżąca wersja tej biblioteki nie jest dobrze dostrojona do korzystania z SIMD (używa SIMD, ale jest daleko od optymalnego). Ostatecznie, spodziewam się, że zostaną zoptymalizowane do punktu, w którym nie warto już pisać kodu ASM, ale realistycznie rzecz biorąc, kompilator nigdy nie będzie tak szybki, jak najnowsze rzeczy, które wychodzi. Gdy biblioteka zostanie dostrojona do dzisiejszych architektur, zostanie przestarzała przez procesor nowej generacji. Stąd ważna jest wiedza na temat zespołu, aby osiągnąć maksymalną przepustowość. Oprócz szybkiego i brudnego przeglądu zasad z poprzednich rozdziałów na temat wektorów i matryc, główne tematy tej części to:

■ Format pamięci matrycy

■ Transpozycja macierzy

■ Produkt punktowy

■ Długość i normalizacja wektora

■ Mnożenie wektorów macierzy

■ Mnożenie macierzy macierzy

■ Wyznacznik macierzy

■ Produkt krzyżowy

■ Podział macierzy (odwrotny)

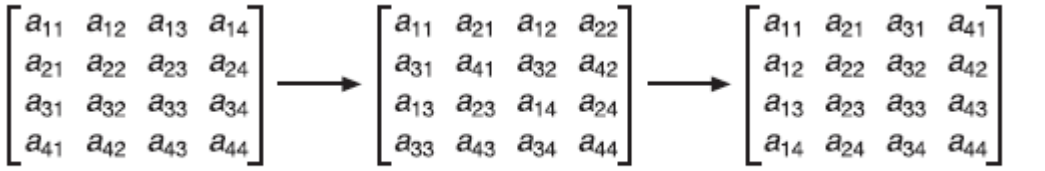
Format pamięci macierzy

Podczas pisania kodu SIMD ważne jest, aby wybrać odpowiednią strukturę. Wprowadza znaczną różnicę w twoim kodzie. Niestety struktura wektorów zależy od liczby obliczeń, które należy zastosować. Jeśli wykonasz kilka obliczeń, może być bardziej optymalne użycie wektorów (X, Y, Z, W). Jeśli musisz zastosować wiele z nich, możesz preferować strukturę pamięci X [], Y [], Z [], W []. Znowu zależy to od zadań, które należy wykonać, oraz od metody wprowadzania, z której korzysta twoja ulubiona biblioteka 3D. Skoncentrujemy się na najbardziej odpowiednich metodach przechowywania, ponieważ każdy z dwóch głównych interfejsów API 3D (OpenGL, Direct3D) używa innego formatu przechowywania matrycy . Zasadniczo, macierze OpenGL typu kolumna-większa są szybsze niż główne macierze Direct3D, gdy zaangażowana jest karta SIMD. Zauważ, że jeśli wykonujesz wiele obliczeń, dla

których macierz X-dur nie jest tak szybka jak format macierzy Y-dur, możesz dokonać transpozycji macierzy przed wykonaniem jakichkolwiek obliczeń.

Transpozycja macierzy

Transpozycja jest operacją prostą geometrycznie, dla której, biorąc pod uwagę dwie macierze A i B oraz ich odpowiednie elementy a_{ij} i b_{ij}, B jest transpozycją A i vice versa wtedy i tylko wtedy, gdy $a_{ij} = b_{ji}$. Innymi słowy, jeśli wymienisz każdy element na przekątnej macierzy, obliczyłeś transpozycję macierzy początkowej. Możesz także zobaczyć transpozycję jako wymianę między wierszami i kolumnami, gdzie wiersz i staje się kolumną i vice versa. Jednym z możliwych rozwiązań tej operacji jest zamiana elementów jeden po drugim. Problem polega na tym, że nie używasz pełnej pojemności magistrali danych. Zamiast tego spróbujemy zastosować podejście SIMD do tego problemu. Tasowanie instrukcji jest kuszące, ale są jedzeniem cyklicznym. Możesz zrobić to lepiej. Instrukcje, które powinny Cię najbardziej przyciągnąć, to funkcje rozpakowywania z jedną precyzją. Kiedy spojrzysz na szczegóły tych dwóch instrukcji, zauważysz, że biorąc pod uwagę dwa rejestry, instrukcje przekształcają dwa elementy tej samej kolumny w sąsiednie rzędy. W przypadku niskiej wersji polecenia rozpakowywania dwa drugie elementy są umieszczone obok siebie w trzecim i czwartym elemencie docelowego rejestru, podczas gdy dwa pierwsze elementy są umieszczone obok siebie w pierwszym i drugim elemencie miejsca docelowego. zarejestrować. Rozwiązuje to połowę problemu, ponieważ możesz zamienić połowę kolumn na wiersze. Możesz zastosować to do dwóch par wierszy, raz dla pierwszych dwóch elementów i raz dla dwóch ostatnich elementów. Pozostało tylko przesuwać górne i dolne sekcje rejestrów, aby elementy tego samego rzędu skończyły się jako elementy tej samej kolumny, i gotowe. Schemat procesu pokazano na rysunku 20.1.



Kod źródłowy tego procesu:

```
lea          esi, [SrcMatrix]
movaps       xmm0, [esi + 0*4*4]
movaps       xmm2, xmm0
unpcklps     xmm0, [esi + 1*4*4]  // xmm0 = [a11, a21, a12, a22]

movaps       xmm1, [esi + 2*4*4]
movaps       xmm3, xmm1
unpcklps     xmm1, [esi + 3*4*4]  // xmm1 = [a31, a41, a32, a42]
movaps       xmm4, xmm0

Unpckhps     xmm2, [esi + 1*4*4]  // xmm2 = [a13, a23, a14, a24]
Lea          edi, [DstMatrix]
unpckhps     xmm3, [esi + 3*4*4]  // xmm3 = [a33, a43, a34, a44]
movaps       xmm5, xmm2

movlhps      xmm0, xmm1           // xmm0 = [a11, a21, a31, a41]
movaps       [edi + 0*4*4], xmm0


movhlps      xmm1, xmm4           // xmm1 = [a12, a22, a32, a42]
movaps       [edi + 1*4*4], xmm1
movlhps      xmm2, xmm3           // xmm2 = [a13, a23, a33, a43]
movaps       [edi + 2*4*4], xmm2
movhlps      xmm3, xmm5           // xmm3 = [a14, a24, a34, a44]
movaps       [edi + 3*4*4], xmm3
```

Iloczyn Skalarny

Jakiś czas temu, kiedy zaczęto używać 3D w oprogramowaniu, mój przyjaciel ubiegał się o pracę w branży gier. Jeden z ankieterów zapytał: "Co to jest iloczyn skalarny?" Mój przyjaciel nie był w stanie odpowiedzieć na to pytanie. Ta podstawowa operacja jest kluczowa dla każdej gry 3D i oczywiście mój przyjaciel nie został przyjęty Większość twórców gier zna równanie iloczynu skalarnego, ale czy znają one geometryczną reprezentację? Jest to klucz do rozwiązania pewnych problemów geometrycznych. Iloczyn skalarny ma geometryczną reprezentację jako długość wektora rzutowanego a na drugim wektorze b, gdy dwa wektory są wyśrodkowane w punkcie początkowym. Operacja jest zapisana następująco, gdzie ostatnią funkcją jest początkowo przedstawiona reprezentacja geometryczna, a theta to kąt między dwoma wektorami:

$$\mathbf{a} \bullet \mathbf{b} = \sum_{i=0}^{n} \mathbf{a}_i \cdot \mathbf{b}_i = \|\mathbf{a}\|_2 \|\mathbf{b}\|_2 \cos(\phi)$$

Równanie centrum jest równaniem zainteresowania, głównie dlatego, że jest łatwe do przechowywania i obliczania.

Array of Structure (AoS) {Struktura tablicy}

Istnieją dwa sposoby przedstawiania list wektorów. Zazwyczaj tworzenie struktury nie jest najbardziej optymalną metodą. Guru OO prawdopodobnie nie polubią tego bardzo, ale z drugiej strony,

najbardziej dosłownie poprawny kod OO zdaje się lekceważyć prędkość w niemal każdym możliwym kierunku. Niemniej jednak bardziej optymalnym rozwiązaniem może być rozpatrzenie problemu jako tablicy struktur z X, Y, Z i W jako członkami struktury. Nadal musisz wymyślić sposób obliczania iloczynu punktowego na trzech elementach określonych w strukturze. Mnożenie jest łatwe do wykonania równolegle (tj. <x. x, y. y, z. z, w. w>), ale jednym dużym problemem jest tutaj poziome dodawanie, które należy wykonać, aby ukończyć iloczyn skalarny. Jeśli chcesz obliczyć iloczyn skalarny na czterech wektorach lub więcej, możesz równie dobrze uznać je za macierze 4 × 4 i obliczyć mnożenie macierzy-wektora. Jeśli nie wszystkie są pomnożone przez ten sam wektor, możesz skorzystać z poziomego dodania z czterema rzędami. To będzie szybsze. Jeśli potrzebujesz tylko obliczyć jeden produkt z kropką, nie masz wielkiego wyboru, ale musisz wykonać poziome dodanie jednego elementu. Możesz podzielić wektor na dwa wektory, gdzie dwa górne elementy są wyrównane z dwoma dolnymi elementami i dodać je równolegle. Wszystko, co pozostanie na końcu, to dodanie pierwszego i drugiego elementu. Shuffle może łatwo wykonać to zadanie. (Jeśli nie jest to jasne, sekcja "Mnożenie wektorów Matrixa" w dalszej części rozdziału objaśni poziome dodawanie z czterema wektorami). Poniższy kod ilustruje ideę zilustrowaną na rysunku 20.2.

Mulps	$[a_1\ a_2\ a_3\ a_4] \cdot [b_1\ b_2\ b_3\ b_4]$	$= [a_1b_1\ a_2b_2\ a_3b_3\ a_4b_4]$
Mov[h1|1h]ps	$[a_1b_1\ a_2b_2 \dots \dots] + [a_3b_3\ a_4b_4 \dots \dots]$	$= [a_1b_1 + a_3b_3\ a_2b_2 + a_4b_4 \dots \dots]$
Pshufd	$[a_1b_1 + a_3b_3 \dots \dots \dots] + [a_2b_2 + a_4b_4 \dots \dots \dots]$	$= [a_1b_1 + a_3b_3 + a_2b_2 + a_4b_4 \dots \dots \dots]$

```
movaps      xmm0, [SrcVectorA]
movaps      xmm1, [SrcVectorB]
mulps       xmm0, xmm1     // xmm0 = [a1, a2, a3, a4]
movhlps     xmm1, xmm0
addps       xmm0, xmm1     // xmm0 = [a1 + a3, a2 + a4, …]
pshufd      xmm1, xmm0, 01010101b
addss       xmm0, xmm1     // xmm0 = [a1 + a3 + a2 + a4, …]
movss       [Dest], xmm0
```

Pozwoli to skutecznie obliczyć iloczyn skalarny dla listy wektorów, jeśli masz je tak skonfigurowane, że każdy wektor jest zawarty w strukturze. Ale jest jeszcze inne podejście do przechowywania danych. . .

Struktura Tablicy (SoA)

Struktura tablic jest znacznie ładniejszym sposobem reprezentowania wektorów. Niestety, biblioteki 3D nie są zbyt przyjazne dla tego formatu. Niemniej jednak nie jesteśmy tutaj, aby debatować o najlepszym wyborze, ponieważ jest to specyficzne dla aplikacji. Chcemy obliczyć iloczyn skalarny formacie SoA. Format SoA nie wymaga struktury, ponieważ każdy składnik starej struktury ma strukturę samą w sobie. W efekcie otrzymasz X, Y i Z zawierające wartości składowe i-tego wektora dla tego komponentu. Jest to bardzo atrakcyjny sposób ustawiania wektorów. Wszystko, co musisz zrobić, to pomnożyć każdą tablicę przez i-element wektora, a zakończyć przez dodanie wszystkiego razem, jak te równania i kod pokazują:

$$\langle x_1, x_2, x_3, x_4 \rangle \bullet \langle a_1, a_2, a_3, a_4 \rangle = \langle a_1 x_1, a_2 x_2, a_3 x_3, a_4 x_4 \rangle$$
$$\langle y_1, y_2, y_3, y_4 \rangle \bullet \langle a_1, a_2, a_3, a_4 \rangle = \langle a_1 y_1, a_2 y_2, a_3 y_3, a_4 y_4 \rangle$$
$$\langle z_1, z_2, z_3, z_4 \rangle \bullet \langle a_1, a_2, a_3, a_4 \rangle = \langle a_1 z_1, a_2 z_2, a_3 z_3, a_4 z_4 \rangle$$

```
movaps      xmm7, [SrcVector]
pshufd      xmm4, xmm7, 11111111b
pshufd      xmm5, xmm7, 10101010b
pshufd      xmm6, xmm7, 01010101b
shufps      xmm7, xmm7, 00000000b
lea         eax, [X]
lea         ebx, [Y]
lea         ecx, [Z]
lea         edx, [W]
lea         edi, [DestValues]

Loop:
movaps      xmm0, [eax]
movaps      xmm1, [ebx]
movaps      xmm2, [ecx]
movaps      xmm3, [edx]
mulps       xmm0, xmm4
mulps       xmm1, xmm5
mulps       xmm2, xmm6
mulps       xmm3, xmm7
addps       xmm0, xmm1
addps       xmm2, xmm3
addps       xmm0, xmm2
movaps      [edi], xmm0
...
```

Długość wektorowa i normalizacja

Normalizacja wektora jest częstym i powolnym zadaniem, i niestety jest wymagana w wielu sytuacjach. Ideą normalizacji jest to, że przy danym wektorze chcemy uzyskać wektor z tym samym kierunkiem, ale dla którego normą (lub jeśli wolisz, długość) jest 1. Typową normą używaną w wektorach jest p-norma dla p = 2. Jest to zdefiniowane jako pierwiastek kwadratowy z sumy kwadratów. Po prostu:

$$\|\mathbf{v}\|_2 = \sqrt{\mathbf{v} \bullet \mathbf{v}}$$
$$= \sqrt{\mathbf{v}_x^{\,2} + \mathbf{v}_y^{\,2} + \mathbf{v}_z^{\,2}}$$

Jeśli chcesz wektor o tym samym kierunku, ale o długości 1, wystarczy podzielić wektor według jego długości. Prosty i prosty.

Tablica Struktury(AoS)

Z tą wersją niewiele można zrobić. Najlepsze, co możesz zrobić, to obliczyć iloczy skalarny wektora na siebie, a następnie odwrotny pierwiastek kwadratowy na nim (przy użyciu Działanie SIMD). To trochę bardziej interesujące, gdy chcesz znormalizować więcej niż jeden wektor. Ponownie możesz pomnożyć każdy wiersz przez siebie i zastosować dodawanie poziomego proces do wszystkich czterech elementów. Możesz zobaczyć, jak to zrobić bardziej efektywnie w czterech rzędach patrząc na sekcję "Mnożenie macierzy-wektor" w tej części. Na razie spójrzmy na kod , który może to zrobić w jednym elemencie:

movaps xmm0, [SrcVectorA]

mulps x mm0, xmm0 // xmm0 = [a1, a2, a3, a4]

movhlps x mm1, xmm0

addps xmm0, xmm1 // xmm0 = [a1 + a3, a2 + a4, ...]

pshufd xmm1, xmm0, 01010101b

addss xmm0, xmm1 // xmm0 = [a1 + a3 + a2 + a4, ...]

rsqrtss x mm0, xmm0

shufps xmm0, xmm0, 00000000b

mulps x mm0, [SrcVectorA]

movaps [Dest], xmm0

Struktura Tablicy (SoA)

Ta wersja struktury jest trochę bardziej przyjazna w działaniu rejestrów SIMD. Aby obliczyć to dla czterech elementów, wystarczy czterostopniowy algorytm: obliczyć iloczynę punktową wektorów, zsumować wszystkie trzy tablice, obliczyć odwrotność pierwiastka kwadratowego i pomnożyć każdą tablicę przez ten wektor. Trochę prostsze niż poprzednia metoda, prawda? Oto kod źródłowy:

movaps xmm0, [X]

movaps xmm1, [Y]

movaps xmm2, [Z]

movaps xmm4, xmm0 // xmm4 = X

mulps x mm0, xmm0

movaps xmm5, xmm1 // xmm5 = Y

mulps x mm1, xmm1

movaps xmm6, xmm2 // xmm6 = Z

mulps xmm2, xmm2

addps x mm0, xmm1

addps xmm0, xmm2 // xmm0 = X + Y + Z

rsqrtps x mm0, xmm0

mulps xmm4, xmm0

mulps xmm5, xmm0

mulps xmm6, xmm0

movaps [DestX], xmm4

movaps [DestY], xmm5

movaps [DestZ], xmm6

Mnożenie macierz - wektor

Jest to dość prosta operacja, którą można zastosować do macierzy. Wektor można zdefiniować jako macierz Nx1. Stąd mnożenie macierzy-wektora może być uważane za specjalny przypadek mnożenia macierzy. Iloczyn dwóch macierzy typu "wiersz-główny", gdzie 2 wskaźniki są (wiersz, kolumna), jest zdefiniowany następująco:

$$x_{ij} = \sum_{j=1}^{4} a_{ij} \cdot b_{jk}$$

Kolumna-główna

W przypadku, gdy b jest wektorem, k = 1. Pamiętaj również, że masz do czynienia z macierzami typu kolumna-major, więc musisz odwrócić subindeksy. Daje to końcowe równanie, w którym indeksy są w formacie tradycyjnym (kolumna, wiersz):

$$x_{1i} = \sum_{j=1}^{4} a_{ji} \cdot b_{1j}$$

Twoje dane są przechowywane w rzędach w macierzy kolumn, więc byłoby to znacznie szybsze i wygodniej jest uzyskać formułę w funkcji rzędów zamiast pojedynczych elementów. Możesz zauważyć, że indeks podrzędny wiersza i jest równy dla x i a. Możesz następnie przekonwertować te dwa elementy do wektorów wierszowych i uzyskać formułę, której szukasz, gdzie x to wektor kolumnowy, a b_j to j. Element b:

$$\mathbf{x} = \sum_{j=1}^{4} \mathbf{r}_j \cdot b_j$$

Kod SIMD dla macierzy kolumn-głównych jest następujący. Zauważ, że to zadziała tylko na SSE2 (Pentium 4 i wyżej). Możesz zmienić pshufd na serię instrukcji shufps / movlps / movhps, aby wykonać to samo zadanie na maszynach niższego rzędu. Kod źródłowy jest prostą implementacją następujących równań procesowych:

$$\begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \end{bmatrix} = \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \\ \mathbf{r}_3 \\ \mathbf{r}_4 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{r}_1 \cdot b_1 & \mathbf{r}_2 \cdot b_2 & \mathbf{r}_3 \cdot b_3 & \mathbf{r}_4 \cdot b_4 \end{bmatrix}$$

```
movaps  xmm0, [SrcVector]
pshufd  xmm1, xmm0, 01010101b
lea     edi, [DstVector]
lea     esi, [Matrix]
pshufd xmm2, xmm0, 10101010b
pshufd xmm3, xmm0, 11111111b
pshufd xmm0, xmm0, 00000000b            // xmmi = {Vector[i]}

mulps   xmm0, [esi + 0*4*4]
mulps   xmm1, [esi + 1*4*4]
addps   xmm0, xmm1
mulps   xmm2, [esi + 2*4*4]
addps   xmm0, xmm2
mulps   xmm3, [esi + 3*4*4]
addps   xmm0, xmm3                       // xmm0 += xmmi * M[i]

movaps  [edi], xmm0
```

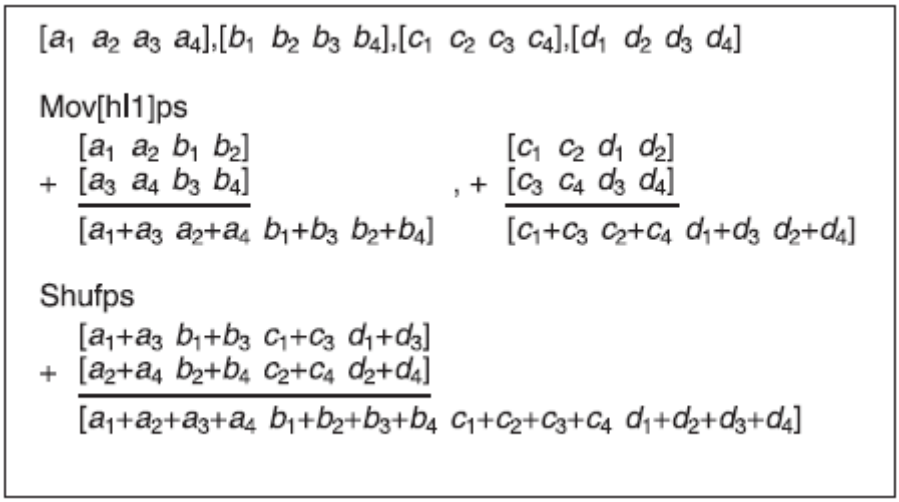
Wiersz - główny

Jeśli spojrzysz na równanie transpozycji macierzy A (macierz rzędów z indeksem umieszczonym w kolejności (wiersz, kolumna)), jasne jest, że nie ma łatwych algebraicznych konwersji dla danych wierszowych:

$$x_{i1} = \sum_{j=1}^{4} a_{ij} \cdot b_{j1}$$

Uproszczenie dla tego zmienia równanie na Ale twoje dane nie są przechowywane w kolumnach. Jeśli myślisz o tym, w jaki sposób wykonywane są mnożenia na wyższym poziomie, zauważysz, że mnożysz jeden wiersz macierzy przez swój kolumna-wektor dodajesz wszystkie wartości razem i to jest twoja pierwsza wartość. Możesz zastosować ten proces do każdego wiersza, uzyskując wektor Nx1. Jest to nic innego jak dodanie poziome, jak widać wcześniej, ale zastosowane cztery razy. Okazuje się, że jest to najlepszy sposób postępowania, więc przyjrzyjmy się, jak można przeprowadzić poziome dodawanie w czterech rzędach elementów. Najbardziej oczywistą rzeczą byłoby przeniesienie wszystkiego dookoła, ale to dość kosztowne. Zamiast tego musisz wybrać podejście, które podzieli problem na mniejsze problemy (strategia "Podziel i pokonaj"). Najpierw musisz dodać elementy razem. Jeśli ty

weź dwa wiersze, powiedzmy U, V, z N elementami, możesz podzielić problem na pół, zamieniając niską połowę U i niską połowę V, a następnie dodając te dwa wektory. Daje to jeden rząd, do którego dodaje się połowę elementów U i V, a wektor jest taki, że połowa jest przeznaczona dla U, a druga połowa dla V. W tym momencie możesz zastosować to rekursywnie. Załóżmy, że istnieją jeszcze dwa wektory, S, T. Zastosuj do nich tę samą logikę, a teraz masz dwa wektory, połączone UV i połączone ST, z których każda zawiera połowę swoich elementów i połowę przestrzeni przeznaczonej na równe części swojej początkowe wektory. Możesz potasować je ponownie w dokładnie taki sam sposób jak poprzednio, ale biorąc pod uwagę podział. Innymi słowy, weź UV i ST. Zamień wewnętrzne zmienne tak, aby U było podzielone na pół. Pierwsza połowa jest w pierwszym wektorze, a druga połowa jest w drugim wektorze. Zrobić to samo dla V, S i T. To daje dwa wektory, każdy z jedną czwartą początkowych wektorów. Dodaj te dwa wektory jeszcze raz i teraz udało ci się zmniejszyć problem do jednej czwartej tego, co było wcześniej. Rzeczywiście, twoje wektory SIMD mają cztery elementy długości, więc dodałeś cztery wiersze w poziomie. Ten proces można zastosować do macierzy z więcej niż czterema elementami, ale nie musisz w tym przypadku. Następujący kod źródłowy, a na rysunku 20.3 pokazano proces dodawania w poziomie.




```
movaps xmm0, [SrcVector]
pshufd xmm1, xmm0, 11101110b      // xmm1 = [v2, v3, v2, v3]
lea          esi, [Matrix]
lea          edi, [DstVector]
pshufd  xmm0, xmm0, 01000100b      // xmm0 = [v0, v1, v0, v1]
```

```
movlps xmm2, [esi + 0*4*4]
movhps xmm2, [esi + 1*4*4]
mulps  xmm2, xmm0                  // xmm2 = [a11, a12, a21, a22]

movlps xmm3, [esi + 0*4*4 + 2*4]
movhps xmm3, [esi + 1*4*4 + 2*4]
mulps  xmm3, xmm1                  // xmm3 = [a13, a14, a23, a24]

movlps xmm4, [esi + 2*4*4]
movhps xmm4, [esi + 3*4*4]
mulps  xmm0, xmm4                  // xmm0 = [a31, a32, a41, a42]

movlps xmm5, [esi + 2*4*4 + 2*4]
movhps xmm5, [esi + 3*4*4 + 2*4]
mulps  xmm1, xmm5                  // xmm1 = [a33, a34, a43, a44]

addps  xmm2, xmm3                  // xmm2 = [a11+a13, a12+a14, a21+a23, a22+a24]
movaps xmm3, xmm2
addps  xmm0, xmm1                  // xmm0 = [a31+a33, a32+a34, a41+a43, a42+a44]

shufps xmm2, xmm0, 11011101b
shufps xmm3, xmm0, 10001000b

addps  xmm3, xmm2                 // xmm0 = [a11+a12+a13+a14, a21+a22+a23+a24]
movaps [edi], xmm3                //        [a31+a32+a33+a34, a41+a42+a43+a44]
```

Teraz prawdopodobnie masz pomysł, dlaczego ta wersja kodu nie działa tak dobrze jak poprzednia wersja. Różnica jest dość zauważalna, ponieważ zajmuje około 2,75 razy więcej czasu niż poprzednia rutyna. Powód jest prosty: więcej instrukcji = wolniejszy kod. Jeśli struktura danych twojego wektora składa się z trzech tablic i potrzebujesz mnożenia wektorów macierzowych dla wielu elementów, powinieneś rozważyć tę operację jako mnożenie macierzy macierzy, gdzie jedna macierz jest tą samą macierzą, która jest tutaj użyta, a druga to macierz (wektor wektora) złożony z czterech wektorów: czterech elementów X, Y, Z i W.

Mnożenie macierz - macierz

Mnożenie macierzy jest operacją wymagającą obliczeń. Na szczęście nie jest używany zbyt mocno na procesorze, ale większość gier 3D wciąż wymaga przyzwoitego zwielokrotnienia macierzy. Już nauczyłeś się zasad mnożenia macierzy macierzy, ale to pomaga ,spójrz na tę operację na wyższym poziomie. Najbardziej znaną matrycą jest macierz rzędowo-major, więc spójrzmy na mnożenie , dwie macierze-główne, A i B, z wynikiem C. Masz AB = C. Jeśli weźmiesz transpozycję po każdej stronie równania, otrzymujesz, że BTAT = CT. Jest to szczególnie interesujące, ponieważ istnieje naprawdę prosta zależność między mnożeniem macierzy szeregowych i macierzami kolumnowymi. Jak wcześniej wspomniano, relacja między nimi jest tylko transpozycją. Oznacza to, że wystarczy wymienić A i B, aby pomnożyć w innym trybie. Możesz znaleźć najlepszy sposób na pomnożenie dwóch macierzy (kolumna-major lub row-major), a to automatycznie da ci sposób na obliczenie innego typu. Instynktownie mnożenie macierzy-wektora powinno być około cztery razy szybsze niż mnożenie

macierz-macierz. Wskazówką dla tego jest to, że mnożenie macierzy-wektora wymaga każdego pola rejestru, więc nie masz już więcej miejsca na dodatkowe obliczenia. Jedynym sposobem, aby to naprawić, jest całkowita zmiana algorytmu, ale nie ma prostszego sposobu na obliczenie mnożenia macierzy macierzy niż zastosowanie tej samej idei, co w mnożeniu macierzy-wektora. Więc gdzie jest ta przewaga? Po pierwsze, mnożenie macierzy-wektora przechowuje wynik jako wiersz, a nie kolumnę, która jest formatem potrzebnym do macierzy kolumn-głównych. Tak więc trochę pracy będzie musiało zostać wykonane tutaj. Inną rzeczą, którą możesz zrobić, to zmniejszyć zużycie zasobów i stragany, nieco mieszając kod, ale jest to w dużym stopniu możliwa optymalizacja. Z definicji masz to

$$x_{jk} = \sum_{j=1}^{4} a_{ij} \cdot b_{jk}$$

Najwygodniejszą metodą jest zawsze rozważanie rzędów elementów. Załóżmy więc, że indeksy definiują macierz typu wiersz-główny; reprezentowałbym wiersz i k kolumny. Chcesz wziąć pod uwagę cały wiersz, więc niech k mieści się w zakresie od 1 do 4. Wymyślisz następujące równanie:

$$r_{x_i} = \sum_{j=1}^{4} a_{ij} \cdot r_{b_j}$$

Jest to równanie, którego można użyć do obliczenia każdego wiersza (tj .: i = 1..4). Ponieważ wybrałeś indeksy, które reprezentują format matrycy rzędu głównych, odwrócenie kolejności dwóch macierzy spowoduje obliczenie mnożenia macierzy kolumn-głównych. Teraz pozostaje tylko zoptymalizować kod zespołu tak, aby zasoby i stoiska były zminimalizowane. Kod obliczający mnożenie dwóch macierzy-głównych jest następujący:

```
// Wiersz 1

lea ebx, [MatrixA] // ebx = MatrixA

lea esi, [MatrixB] // esi = MatrixB

movss x mm0, [ebx + 0 * 4 * 4 + 0 * 4]

movaps xmm4, [esi + 0 * 4 * 4] // xmm4 = Matrix2 [0]

shufps x mm0, xmm0, 00000000b // xmm0 = [a11, a11, a11, a11]

movss x mm1, [ebx + 0 * 4 * 4 + 1 * 4]

mulps x mm0, xmm4

movaps xmm5, [esi + 1 * 4 * 4] // xmm5 = Matrix2 [1]

shufps xmm1, xmm1, 00000000b // xmm1 = [a12, a12, a12, a12]

movss xmm2, [ebx + 0 * 4 * 4 + 2 * 4]

mulps x mm1, xmm5

movaps xmm6, [esi + 2 * 4 * 4] // xmm6 = Matrix2 [2]

shufps xmm2, xmm2, 00000000b // xmm2 = [a13, a13, a13, a13]

addps x mm0, xmm1
```

```
movss xmm3, [ebx + 0 * 4 * 4 + 3 * 4]
mulps xmm2, xmm6
movaps xmm7, [esi + 3 * 4 * 4] // xmm7 = matrix2 [3]
shufps xmm3, xmm3, 00000000b // xmm3 = [a14, a14, a14, a14]
addps x mm0, xmm2
mulps xmm3, xmm7
lea edi, [DstMatrix] // edi = DstMatrix
addps xmm3, xmm0 // xmm3 + = xmmi * xmm [i + 4]
// Wiersz 2
movss x mm0, [ebx + 1 * 4 * 4 + 0 * 4]
shufps xmm0, xmm0, 00000000b // xmm0 = [a21, a21, a21, a21]
movss xmm1, [ebx + 1 * 4 * 4 + 1 * 4]
shufps xmm1, xmm1, 00000000b // xmm1 = [a22, a22, a22, a22]
movss xmm2, [ebx + 1 * 4 * 4 + 2 * 4]
shufps xmm2, xmm2, 00000000b // xmm2 = [a23, a23, a23, a23]
movaps [edi], xmm3
movss xmm3, [ebx + 1 * 4 * 4 + 3 * 4]
shufps xmm3, xmm3, 00000000b // xmm3 = [a24, a24, a24, a24]
mulps x mm0, xmm4
mulps x mm1, xmm5
addps x mm0, xmm1
mulps xmm2, xmm6
addps x mm0, xmm2
mulps xmm3, xmm7
addps xmm0, xmm3 // xmm0 += xmmi * xmm[i+4]
movaps [edi + 1*4*4], xmm0
// Row 3
movss xmm0, [ebx + 2*4*4 + 0*4]
shufps xmm0, xmm0, 00000000b // xmm0 = [a31, a31, a31, a31]
movss xmm1, [ebx + 2*4*4 + 1*4]
shufps xmm1, xmm1, 00000000b // xmm1 = [a32, a32, a32, a32]
```

```
movss xmm2, [ebx + 2*4*4 + 2*4]
shufps xmm2, xmm2, 00000000b // xmm2 = [a33, a33, a33, a33]
movss xmm3, [ebx + 2*4*4 + 3*4]
shufps xmm3, xmm3, 00000000b // xmm3 = [a34, a34, a34, a34]
mulps xmm0, xmm4
mulps xmm1, xmm5
addps xmm0, xmm1
mulps xmm2, xmm6
addps xmm0, xmm2
mulps xmm3, xmm7
addps xmm0, xmm3 // xmm0 += xmmi * xmm[i+4]
movaps [edi + 2*4*4], xmm0
// Row 4
movss xmm0, [ebx + 3*4*4 + 0*4]
shufps xmm0, xmm0, 00000000b // xmm0 = [a41, a41, a41, a41]
movss xmm1, [ebx + 3*4*4 + 1*4]
shufps xmm1, xmm1, 00000000b // xmm1 = [a42, a42, a42, a42]
movss xmm2, [ebx + 3*4*4 + 2*4]
shufps xmm2, xmm2, 00000000b // xmm2 = [a43, a43, a43, a43]
mulps xmm0, xmm4
movss xmm3, [ebx + 3*4*4 + 3*4]
shufps xmm3, xmm3, 00000000b // xmm3 = [a44, a44, a44, a44]
mulps xmm1, xmm5
addps xmm0, xmm1
mulps xmm2, xmm6
addps xmm0, xmm2
mulps xmm3, xmm7
addps xmm0, xmm3 // xmm0 += xmmi * xmm[i+4]
movaps [edi + 3*4*4], xmm0
```

Wyznacznik macierzy

Obliczanie wyznacznika macierzy jest często użyteczną funkcją obliczeniową problemu geometrii. Wiele problemów można przekształcić w obliczenia wyznacznika, co czyni tę operację koniecznością dla wielu złożonych problemów geometrycznych. Wyznacznik macierzy 2 x2 jest oznaczony | A | i jest zdefiniowany jako $a_{11}a_{22}$ - $a_{12}a_{21}$. Geometrycznie, możesz zobaczyć to jako jedną przekątną mniej lustrzaną przekątną. Zdefiniuj M_{ij} jako element podrzędny macierzy M. Mniejsza macierz jest macierzą otrzymaną przez usunięcie wiersza i kolumny j w macierzy szeregowo-durowej. Możesz zdefiniować wyznacznik macierzy kwadratowych większych niż 2 przez

$$\left|\mathbf{M}\right| = \sum_{j=1}^{n} (-1)^{j} a_{1j} \left|\mathbf{M}_{1j}\right|$$

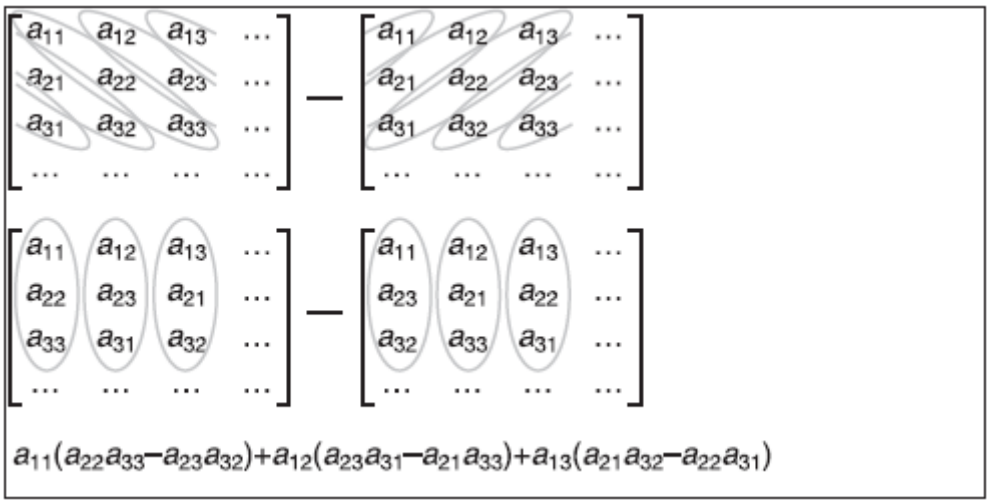
Wyznacznik ma wiele właściwości, ale ten, który nas interesuje, jest poprzedni, a fakt, że | M | = | M^T|.

Wyznacznik macierzy 3 x 3

W przypadku macierzy 3 x 3 istnieje naprawdę schludna sztuczka geometryczna do obliczania wyznacznika. Ta sztuczka jest całkiem jasna, gdy równanie jest rozszerzone:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{44} \end{vmatrix} + a_{12} \begin{vmatrix} a_{23} & a_{21} \\ a_{33} & a_{31} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

Spójrz na swoją oryginalną matrycę i spójrz na każdy pojedynczy termin w rozwiniętym równaniu. Zauważysz, że każdy tri-termin jest przekątną. Jest w tym jeszcze więcej; każdy negatywny termin idzie od prawej do lewej, podczas gdy pozytywne terminy idą od lewej do prawej. Jest to całkiem niezłe, ponieważ daje prosty sposób obliczenia tego nie tylko na papierze, ale również przy użyciu SIMD. Wszystko, co musisz zrobić, to przesunąć wiersze dwa i trzy tak, aby były wyrównane na jedną przekątną (powiedzmy prawą i lewą), a następnie pomnożyć je wszystkie razem, jak pokazano na rysunku 20.4.



Zrób to samo dla lewego i prawego, pomnóż je razem, odejmij drugie od pierwszego, a następnie dodaj poziome dodawanie.

Macierze 3x3 są znacznie mniej kosztowne niż matryce 4x4. Oczywiście zakłada to, że przechowujesz macierz 3x3 w macierzy 4x 4 (coś, co możesz zrobić, zazwyczaj). Kod źródłowy do tego wynika:

```
lea          esi, [Matrix]
lea          edi, [D]
pshufd       xmm1, [esi + 4*4*1], 11001001b
pshufd       xmm2, [esi + 4*4*2], 11010010b
mulps        xmm1, xmm2                          // Multiply the 2 last rows

pshufd       xmm3, [esi + 4*4*1], 11010010b
pshufd       xmm4, [esi + 4*4*2], 11001001b
mulps        xmm3, xmm4                          // ^^ Again but reverse-crossing

subps        xmm1, xmm3
mulps        xmm1, [esi + 4*4*0]

movhlps      xmm0, xmm1
addss        xmm0, xmm1                          // xmm0 = [a1 + a3, a2 + a4, …]
pshufd       xmm1, xmm1, 01010101b
addss        xmm0, xmm1                          // xmm0 = [a1 + a3 + a2 + a4, …]
movss        [edi], xmm0
```

Wyznacznik macierzy 4 x 4

Jeśli możesz to zrobić za pomocą determinanty macierzy 3 x 3 zastosowanej na macierzy 4 x4, zrób to. Wymagania obliczeniowe są o wiele lepsze. Jeśli nie, musisz jeszcze wyliczyć wyznacznik macierzy 4 x 4. Jeśli rozwiniesz dość długie równanie, szybko zauważysz, że nie ma żadnej ładnej zależności geometrycznej na macierzy, tak jak w przypadku macierzy 3? 3. W rzeczywistości proces ten jest nieco bolesny i opiera się na rozkładzie, jak zaobserwowano wcześniej. Przyjrzyjmy się, jak można to skutecznie obliczyć za pomocą SIMD, dekomponując definicję wyznacznika. Definicja 4? 4 jest zdefiniowana przez

$$\mathbf{M}_1 = \begin{vmatrix} a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} \end{vmatrix}$$
$$= a_{22}\begin{vmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{vmatrix} + a_{23}\begin{vmatrix} a_{34} & a_{32} \\ a_{44} & a_{42} \end{vmatrix} + a_{24}\begin{vmatrix} a_{32} & a_{33} \\ a_{42} & a_{43} \end{vmatrix}$$

Thiking in terms of rows, we get

$$[\mathbf{M}_{11} \quad \mathbf{M}_{12} \quad \mathbf{M}_{13} \quad \mathbf{M}_{14}] = (\mathbf{r}_2 \ll 1)\begin{vmatrix} \mathbf{r}_2 \ll 2 & \mathbf{r}_3 \ll 3 \\ \mathbf{r}_4 \ll 2 & \mathbf{r}_4 \ll 3 \end{vmatrix} + (\mathbf{r}_2 \ll 2)\begin{vmatrix} \mathbf{r}_3 \ll 3 & \mathbf{r}_3 \ll 1 \\ \mathbf{r}_4 \ll 3 & \mathbf{r}_4 \ll 1 \end{vmatrix} + (\mathbf{r}_2 \ll 3)\begin{vmatrix} \mathbf{r}_3 \ll 1 & \mathbf{r}_3 \ll 2 \\ \mathbf{r}_4 \ll 1 & \mathbf{r}_4 \ll 2 \end{vmatrix}$$

Zmniejsza to problem obliczania wyznacznika każdej matrycy 3? 3. Zauważmy jeden bardzo interesujący fakt geometryczny w wyrażeniu drugiego poziomu: Jeśli spojrzysz na drugi indeks pierwszego elementu dla każdej podrzędnej macierzy, to zawsze zaczyna się on o jeden więcej niż drugi indeks współczynnika, przez który mnoży się wyznacznik. Ponadto liczby od lewej do prawej dla tego samego drugiego indeksu zawsze rosną, a oba wiersze mają długość cyklu równą cztery. Fakt ten jest

bardzo interesujący, ponieważ sugeruje, że dla każdej z pierwszych trzech mniejszych macierzy następna podrzędna matryca jest w rzeczywistości tą samą macierzą, w której wszystkie elementy macierzy 4 x 4 zostały przesunięte o jeden element w lewo. Krótko mówiąc, dzięki cyklowi elementów możesz zastanowić się, jak obliczyć wyznacznik macierzy 3 x 3, aby obliczyć wyznacznik wszystkich macierzy mniejszego poziomu. Jedyną rzeczą, którą musisz zachować ostrożność chodzi o to, aby nie wykonywać żadnych operacji poziomych, ponieważ łamią one symetrię, której używasz do obliczenia czterech mniej znaczących determinantów 3 x 3. Oznacza to, że nie można użyć poprzedniej metody do obliczenia wyznaczników podmacierzy 3 x 3. Zwróćmy naszą uwagę na obliczanie wyznacznika M_{11}. Używając instrukcji 4-wektorowej SSD SIMD, będzie to również obliczać M_{12}, M_{13} i M_{14}. Rozszerzając wyznacznik więcej, otrzymujesz

$$\begin{vmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{vmatrix} = a_{11}\begin{vmatrix} a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} \end{vmatrix} + a_{12}\begin{vmatrix} a_{23} & a_{24} & a_{21} \\ a_{33} & a_{34} & a_{31} \\ a_{43} & a_{44} & a_{41} \end{vmatrix} + a_{13}\begin{vmatrix} a_{24} & a_{21} & a_{22} \\ a_{34} & a_{31} & a_{32} \\ a_{44} & a_{41} & a_{42} \end{vmatrix} + a_{14}\begin{vmatrix} a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{vmatrix}$$

$$= a_{11}\mathbf{M}_{11} - a_{12}\mathbf{M}_{12} + a_{13}\mathbf{M}_{13} + a_{14}\mathbf{M}_{14}$$

$$= \mathbf{r}_1 \bullet \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \mathbf{M}_{13} & \mathbf{M}_{14} \end{bmatrix}$$

Masz teraz sposób na obliczenie determinanty macierzy 4 x 4, ale możesz minimalnie wyliczyć rzeczy, aby uzyskać ładniejszą i bardziej optymalną formę wyrażenia formuły. Jedną z pierwszych rzeczy, którą powinieneś zauważyć, jest to, że istnieje ładny związek między pierwszym i ostatnim pod-determinantami. Mianowicie, pierwszy pod determinant jest taki sam jak ostatni wyznacznik, ale z obrotem jednego elementu w prawo. Inną interesującą obserwacją jest to, że pierwszy rząd drugiego pod-determinanta jest taki sam jak drugi rząd tego samego subdeterminantu, ale obrócony przez dwa elementy. Jeśli spojrzysz na swoją początkową macierz 4 x 4 i spojrzysz na najmniejszą wyznaczoną przez centrum determinantę, to pomnożone elementy są zawsze dwoma elementami od siebie w poziomie i jedna jednostka osobno w pionie. Przechodzenie dwóch jednostek w lewo lub dwóch jednostek w prawo to dokładnie to samo. Dlatego możesz po prostu przesunąć ten wiersz dwóch jednostek, gdy chcesz się pomnożyć razem, aby obliczyć drugi pod-determinant. Jest to w dużym stopniu rozkład czynników, które można zastosować tutaj. Resztą prac będzie optymalizacja wykorzystania zasobów i ograniczenie przesunięć wierszy do minimum. Poniższy kod pokazuje to:

```
  lea     esi, [Matrix]
  pshufd  xmm6, [esi + 4*4*1], 10010011b          // xmm6 = [a24 a21 a22 a23]

  movaps  xmm0, [esi + 4*4*3]                      // xmm0 = [a41 a42 a43 a44]
  pshufd  xmm3, [esi + 4*4*2], 00111001b          // xmm3 = [a32 a33 a34 a31]

  pshufd  xmm1, xmm0, 01001110b
  mulps   xmm1, xmm3                               // xmm1 = [a32*a43 a33*a44 a34*a41
  a31*a42]

  pshufd  xmm2, xmm0, 10010011b
  mulps   xmm2, xmm3                               // xmm2 = [a32*a44 a33*a41 a34*a42
  a31*a43]
  pshufd  xmm4, xmm6, 01001110b                    // xmm4 = [a22 a23 a24 a21]

  mulps   xmm0, xmm3
  pshufd  xmm0, xmm0, 00111001b                    // xmm0 = [a33*a42 a34*a43 a31*a44
  a32*a41]


pshufd  xmm5, xmm6, 10010011b                     // xmm5 = [a23 a24 a21 a22]

subps   xmm1, xmm0                                // xmm1 = [md13 ma13 mb13 mc13]
pshufd  xmm0, xmm1, 00111001b                     // xmm0 = [md11 ma11 mb11 mc11]

pshufd  xmm3, xmm2, 01001110b
subps   xmm3, xmm2                                // xmm3 = [md12 ma12 mb12 mc12]

mulps   xmm0, xmm4
mulps   xmm1, xmm6
mulps   xmm3, xmm5

addps   xmm0, xmm1
addps   xmm0, xmm3                                // xmm0 = [M11 M12 M13 M14]
mulps   xmm0, [esi + 4*4*0]                       // xmm0 = [a11*M11 a12*M12 a13*M13
a14*M14]

movhlps xmm1, xmm0
addps   xmm0, xmm1                                // xmm0 = [A1 + A3, A2 + A4, …]
pshufd  xmm1, xmm0, 01010101b
subss   xmm0, xmm1                                // xmm0 = [A1 + A3 + A2 + A4, …]

movss   [Destination], xmm0
```

Iloczyn wektorowy

Iloczyn wektorowy to kolejna podstawowa operacja wektorowa. Jego geometryczna reprezentacja, z dwoma wektorami A i B, polega na tym, że powstały wektor C = A [d] B jest prostopadły do płaszczyzny generowanej przez AB. Wynik jest zdefiniowany jako

$$\begin{bmatrix} a_y b_z - a_z b_y & a_z b_x - a_x b_z & a_x b_y - a_y b_x \end{bmatrix}$$

Lub jeśli wolisz, może być reprezentowany w postaci macierzy jako wyznacznik macierzy, dla której pierwszy wiersz jest kierunkowym wektorem (i, j, k) i gdzie drugi i trzeci wektor to odpowiednio A i B. Tak czy inaczej, wygląda to bardzo blisko wyznacznika, a obliczenia są w istocie bardzo podobne. Obliczenie iloczynu krzyżowego jest tak proste, jak można się spodziewać. Wszystko, co musisz zrobić, aby to obliczyć, to obrócić wektor B w lewo, pomnożyć przez A, odjąć od mnożenia B i obrót A w lewo, i gotowe. Nic do tego, zwłaszcza po obliczeniu wyznacznika. Wersja AoS tego obliczenia jest zbyt łatwa i pozostanie jako ćwiczenie. Kod źródłowy dla wersji SoA:

```
pshufd x mm0, [VectorA], 11001001b

pshufd xmm1, [VectorB], 11010010b

mulps x mm0, xmm1

pshufd xmm2, [VectorB], 11001001b

pshufd xmm3, [VectorA], 11010010b

mulps xmm2, xmm3

subps x mm0, xmm2

movaps [DstVector], xmm0
```

Dzielenie Macierzy (Inwersja)

Jeśli odczuwasz ostry ból w lewej części mózgu, to prawdopodobnie dlatego, że twoja artystyczna strona jest nieco zaniedbana przez ten rozdział. Z drugiej strony, jeśli odczuwasz ostry ból po prawej stronie, prawdopodobnie dlatego, że przeczytałeś tytuł tej następnej sekcji i wiesz, co jest w niej zaangażowane. Dzielenie prawie zawsze były wolniejsze niż wielokrotności, a macierze nie są wyjątkiem. Łatwą metodą nauczaną w liceum jest napisanie macierzy, po której następuje macierz tożsamości, a zadaniem jest wyeliminowanie eliminacjaGaussa-Jordana w celu uzyskania macierzy tożsamości po lewej stronie. Ta metoda jest skuteczna, ale brakuje jej kilku rzeczy. Na początek nie jest łatwo powiedzieć, czy można uzyskać odwrotność. Ale największym problemem z tą metodą jest to, że nie jest ona zbyt dokładna. Wartości są wielokrotnie wykorzystywane wielokrotnie i zaangażowane w nie wiele podziałów, dzięki czemu metoda ta jest nie tylko bezużyteczna, ale także powolna. Inną techniką stosowaną do obliczenia tego jest użycie wyznacznika. Macierz odwrotną można zdefiniować jako transponowaną sąsiednią macierz podzieloną przez bezwzględną wartość wyznacznika. Czym dokładnie jest sąsiednia macierz? Jest to macierz, dla której każdy element a_{ij} jest równy wyznacznikowi mniejszej macierzy M_ij:

$$\mathbf{M}^{-1} = \frac{\operatorname{adj}(\mathbf{M})^{\mathrm{T}}}{|\mathbf{M}|}$$

$$= \frac{1}{|\mathbf{M}|}\begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{21} & \mathbf{M}_{31} & \mathbf{M}_{41} \\ \mathbf{M}_{12} & \mathbf{M}_{22} & \mathbf{M}_{32} & \mathbf{M}_{42} \\ \mathbf{M}_{13} & \mathbf{M}_{23} & \mathbf{M}_{33} & \mathbf{M}_{43} \\ \mathbf{M}_{14} & \mathbf{M}_{24} & \mathbf{M}_{34} & \mathbf{M}_{44} \end{bmatrix}^{\mathrm{T}}$$

$$= \frac{1}{|\mathbf{M}|}\begin{bmatrix} \begin{vmatrix} a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} \end{vmatrix} & \begin{vmatrix} a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} \\ a_{41} & a_{43} & a_{44} \end{vmatrix} & \begin{vmatrix} a_{21} & a_{22} & a_{24} \\ a_{31} & a_{32} & a_{34} \\ a_{41} & a_{42} & a_{44} \end{vmatrix} & \begin{vmatrix} a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{vmatrix} \\ \begin{vmatrix} a_{12} & a_{13} & a_{14} \\ a_{32} & a_{33} & a_{34} \\ a_{42} & a_{43} & a_{44} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} & a_{14} \\ a_{31} & a_{33} & a_{34} \\ a_{41} & a_{43} & a_{44} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} & a_{14} \\ a_{31} & a_{32} & a_{34} \\ a_{41} & a_{42} & a_{44} \end{vmatrix} & \begin{vmatrix} a_{12} & a_{13} & a_{13} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{vmatrix} \\ \begin{vmatrix} a_{12} & a_{13} & a_{14} \\ a_{22} & a_{23} & a_{24} \\ a_{42} & a_{43} & a_{44} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} & a_{14} \\ a_{21} & a_{23} & a_{24} \\ a_{41} & a_{43} & a_{44} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} & a_{14} \\ a_{21} & a_{22} & a_{24} \\ a_{41} & a_{42} & a_{44} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{vmatrix} \\ \begin{vmatrix} a_{12} & a_{13} & a_{14} \\ a_{22} & a_{23} & a_{24} \\ a_{32} & a_{33} & a_{34} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} & a_{14} \\ a_{21} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} & a_{14} \\ a_{21} & a_{22} & a_{24} \\ a_{31} & a_{32} & a_{34} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{33} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \end{bmatrix}$$

Kilka ciekawych rzeczy pojawia się, gdy spojrzysz na redundancję w sąsiedniej matrycy podanej przy tym zapisie. Po pierwsze, musisz obliczyć wyznacznik M, co oznacza, że w pewnym momencie musisz także obliczyć wyznacznik mniejszych macierzy M_{11}, M_{12}, M_{13} i M_{14}. Nie trzeba geniuszu, aby zauważyć, że pierwszy rząd macierzy odwrotnej zawiera dokładnie te cztery elementy. Innymi słowy, obliczając wyznacznik M, obliczono również pierwszy rząd sąsiedniej macierzy. Zauważ, że dwa ostatnie rzędy wyznaczników w pierwszym i drugim rzędzie sąsiedniej macierzy są identyczne. Oznacza to, że możesz wstępnie obliczać tę wartość i używać jej podczas obliczania wyznacznika ośmiu pierwszych matryc mniejszych. To samo dotyczy wyznacznika dwóch ostatnich rzędów sąsiedniej macierzy. W tym przypadku dwa pierwsze rzędy wyznacznika są identyczne. Przypomnijmy, że przy obliczaniu wyznacznika rzeczywista kolejność rzędów nie jest ważna. Dzięki temu możesz wstawić ostatni wiersz wyznaczników jako pierwszy wiersz i zastosować dokładnie ten sam proces, który był użyty dla pierwszego i drugiego wiersza sąsiedniej macierzy. Na koniec musisz zakończyć transpozycją i mnożeniem każdego elementu przez odwrotność wyznacznika M. Od tego momentu pozostało tylko napisanie kodu zespołu, który optymalizuje zasoby najlepiej jak to możliwe:

```
__declspec(align(16)) static const unsigned long Sign[4] =

{0x00000000, 0x80000000, 0x00000000, 0x80000000};

// Row 1

lea esi, [Matrix]
```

```
pshufd xmm6, [esi + 4*4*1], 10010011b // xmm6 = [a24 a21 a22 a23]
pshufd xmm3, [esi + 4*4*2], 00111001b // xmm3 = [a32 a33 a34 a31]
movaps xmm0, [esi + 4*4*3] // xmm0 = [a41 a42 a43 a44]
pshufd xmm1, xmm0, 01001110b
mulps xmm1, xmm3 // xmm1 = [a32*a43 a33*a44 a34*a41
a31*a42]
pshufd xmm2, xmm0, 10010011b
mulps xmm2, xmm3 // xmm2 = [a32*a44 a33*a41 a34*a42
a31*a43]
pshufd xmm7, xmm6, 01001110b // xmm7 = [a22 a23 a24 a21]
mulps xmm0, xmm3
pshufd xmm0, xmm0, 00111001b // xmm0 = [a33*a42 a34*a43 a31*a44
a32*a41]
pshufd xmm5, xmm6, 10010011b // xmm5 = [a23 a24 a21 a22]
subps xmm1, xmm0 // xmm1 = [md13 ma13 mb13 mc13]
pshufd xmm0, xmm1, 00111001b // xmm0 = [md11 ma11 mb11 mc11]
pshufd xmm3, xmm2, 01001110b
subps xmm3, xmm2 // xmm3 = [md12 ma12 mb12 mc12]
mulps xmm7, xmm0
mulps xmm6, xmm1
mulps xmm5, xmm3
addps xmm7, xmm6
addps xmm7, xmm5 // xmm7 = [M11 M12 M13 M14]
movaps xmm6, xmm7 // xmm6 = [M11 M12 M13 M14]
pshufd xmm2, [esi + 4*4*0], 10010011b // xmm2 = [a14 a11 a12 a13]
// Determinant
mulps xmm7, [esi + 4*4*0] // xmm0 = [a11*M11 a12*M12 a13*M13
a14*M14]
pshufd xmm4, xmm2, 01001110b // xmm4 = [a12 a13 a14 a11]
movhlps xmm5, xmm7
addps xmm7, xmm5 // xmm0 = [A1 + A3, A2 + A4, …]
```

```
pshufd xmm5, xmm7, 01010101b
subss xmm7, xmm5 // xmm0 = [A1 + A3 + A2 + A4, …]
// Row 2
pshufd xmm5, xmm2, 10010011b // xmm5 = [a13 a14 a11 a12]
mulps xmm4, xmm0
mulps xmm2, xmm1
mulps xmm5, xmm3
addps xmm4, xmm2
addps xmm5, xmm4 // xmm5 = [M21 M22 M23 M24]
// Row 3
pshufd xmm4, [esi + 4*4*3], 00111001b // xmm4 = [a42 a43 a44 a41]
movaps xmm0, [esi + 4*4*1] // xmm0 = [a21 a22 a23 a24]
pshufd xmm3, [esi + 4*4*0], 00111001b // xmm3 = [a12 a13 a14 a11]
pshufd xmm1, xmm0, 01001110b
mulps xmm1, xmm3 // xmm1 = [a12*a23 a13*a24 a14*a21
a11*a22]
pshufd xmm2, xmm0, 10010011b
mulps xmm2, xmm3 // xmm2 = [a12*a24 a13*a21 a14*a22
a11*a23]
mulps xmm0, xmm3
pshufd xmm0, xmm0, 00111001b // xmm0 = [a13*a22 a14*a23 a11*a24
a12*a21]
subps xmm1, xmm0 // xmm1 = [md43 ma43 mb43 mc43]
pshufd xmm0, xmm1, 00111001b // xmm0 = [md41 ma41 mb41 mc41]
mulps xmm4, xmm0
pshufd xmm3, xmm2, 01001110b
subps xmm3, xmm2 // xmm3 = [md42 ma42 mb42 mc42]
pshufd xmm2, [esi + 4*4*3], 10010011b // xmm2 = [a44 a41 a42 a43]
mulps xmm2, xmm1
addps xmm4, xmm2
pshufd xmm2, [esi + 4*4*3], 01001110b // xmm2 = [a43 a44 a41 a42]
```

```
mulps xmm2, xmm3
addps xmm4, xmm2 // xmm4 = [M31 M32 M33 M34]
// Row 4
pshufd xmm2, [esi + 4*4*2], 01001110b // xmm2 = [a33 a34 a31 a32]
mulps xmm3, xmm2
rcpss xmm7, xmm7
pshufd xmm2, [esi + 4*4*2], 10010011b // xmm2 = [a34 a31 a32 a33]
mulps xmm1, xmm2
pshufd xmm2, [esi + 4*4*2], 00111001b // xmm2 = [a32 a33 a34 a31]
mulps xmm0, xmm2
pshufd xmm7, xmm7, 00000000b
addps xmm3, xmm1
addps xmm3, xmm0 // xmm3 = [M41 M42 M43 M44]
// Set the right signs
// Multiply by the Determinant
pxor xmm7, [Sign]
mulps xmm6, xmm7
mulps xmm4, xmm7
pshufd xmm7, xmm7, 00010001b
mulps xmm5, xmm7
mulps xmm3, xmm7
// Transpose
movaps xmm7, xmm6
unpcklps xmm6, xmm5 // xmm6 = [a11, a21, a12, a22]
movaps xmm0, xmm4
unpcklps xmm4, xmm3 // xmm4 = [a31, a41, a32, a42]
movaps xmm2, xmm6
unpckhps xmm7, xmm5 // xmm2 = [a13, a23, a14, a24]
lea edi, [DstMatrix]
unpckhps xmm0, xmm3 // xmm0 = [a33, a43, a34, a44]
movaps xmm1, xmm7
```

```
movlhps xmm6, xmm4 // xmm6 = [a11, a21, a31, a41]
movaps [edi + 0*4*4], xmm6
movhlps xmm4, xmm2 // xmm4 = [a12, a22, a32, a42]
movaps [edi + 1*4*4], xmm4
movlhps xmm7, xmm0 // xmm2 = [a13, a23, a33, a43]
movaps [edi + 2*4*4], xmm7
movhlps xmm0, xmm1 // xmm0 = [a14, a24, a34, a44]
movaps [edi + 3*4*4], xmm0
```