УДК 621.391

Умножение матриц n–разрядных чисел с фиксированной точкой с помощью n/2–разрядных векторных инструкций

И. Сафонов, А. Аюпов, С. Бёрнс

АО “Интел А/О”, ilia.safonov@intel.com

***Аннотация* — Рассматривается способ умножения прямоугольных матриц n-разрядных чисел с фиксированной точкой с помощью декомпозиции на арифметические операции n/2-разрядных целых чисел. Обработка осуществляется на DSP с SIMD и VLIW архитектурой. Приводится эффективная реализация умножения матриц 32-разрядных чисел с помощью умножения матриц 16-разрядных чисел через тензорное произведение векторов. Обсуждается ряд подходов, позволяющих достичь высокой скорости обработки за счет векторных инструкций и эффективного использования конвейера. Полученные результаты соответствуют теоретически достижимой вычислительной сложности. Предложенный алгоритм позволяет реализовать эффективную параллельную обработку.**

***Ключевые слова* — умножение прямоугольных матриц, SIMD инструкции, числа с фиксированной точкой, тензорное произведение векторов, программирование DSP, параллельная обработка.**

I. Введение

В настоящее время происходит бурный рост устройств носимой (wearable) и мобильной электроники. Часть подобных устройств имеют одну или несколько видеокамер и способны решать различные задачи обработки изображений и компьютерного зрения. Одним из основных требований к аппаратуре носимой электроники является низкое энергопотребление, гарантирующее продолжительное время работы от аккумулятора. С другой стороны, аппаратура должна быть достаточно универсальна, чтобы быть способной гибко настраиваться на различные задачи.

В работе используется IP блок, который включает в себя DSP, оптимизированный для обработки изображений и видео. Этот процессор имеет VLIW/SIMD архитектуру, скалярные инструкции для работы с 32-разрядными целыми числами и векторные инструкции для работы с 16-разрядными целыми числами, что позволяет эффективно реализовать огромное количество алгоритмов обработки изображений. Однако в компьютерном зрении существует класс задач, в которых интенсивно используется перемножение прямоугольных матриц действительных чисел, причем размер матриц может достигать несколько сотен на несколько сотен. Например, к таким задачам относится визуальный SLAM (Simultaneous Localization And Mapping) с помощью различных модификаций фильтра Калмана [1].

В идеале подобные задачи требуют вычислений с действительными числами. С некоторыми ограничениями допустимо использовать матрицы 32-разрядных чисел в арифметике с фиксированной точкой. Умножение матриц на используемом DSP с помощью скалярных операций выполняется недопустимо долго и ведет к излишнему расходу энергии. Целесообразно использовать векторные операции, но процессор имеет SIMD инструкции только для 16-разрядных целых чисел. В данной статье рассматривается способ умножения прямоугольных матриц 32-разрядных чисел с фиксированной точкой с помощью 16-разрядных векторных инструкций для целых чисел. Способ легко обобщается на случай n-разрядных чисел и векторных инструкций для n/2-разрядных чисел. Большинство предлагаемых подходов для оптимизации производительности применимы и для других процессоров ЦОС.

Ещё одним важным требованием к алгоритму перемножения матриц является возможность обрабатывать матрицы по частям (по блокам), что позволяет минимизировать количество операций загрузки данных в сверхоперативную локальную память IP блока и дает возможность распараллелить обработку на несколько идентичных IP блоков в системе на кристалле. Подходы к параллельному умножению матриц также рассматриваются в данной работе.

Данная работа организована следующим образом. Во второй главе описаны способы декомпозиции n-разрядного умножения на n/2-разрядные арифметические операции для целых чисел. В главе III обсуждается применимость существующих алгоритмов умножения матриц для данной задачи. Четвертая глава посвящена описанию предлагаемого способа умножения матриц чисел с фиксированной точкой с помощью векторных команд. Некоторые приёмы для ускорения ЦОС описаны в главе V. В шестой главе рассматриваются подходы к параллельной обработке. Замеры времени работы обсуждаемых алгоритмов приводятся в главе VII.

II. Декомпозиция умножения для целых чисел

Два n-разрядных беззнаковых целых числа U с битами (u_n, u_{n-1}, … u_1) и V с битами (v_n, v_{n-1}, … v_1) можно записать в виде:

$$U = 2^{n.2}U_H + U_L, \quad V = 2^{n.2}V_H + V_L,$$

где U_H – это старшая или наиболее значимая половина U, имеющая биты (u_n, u_{n-1}, ... $u_{n/2+1}$), U_L – это младшая или наименее значимая половина U, имеющая биты ($u_{n/2}$, $u_{n/2-1}$, ... u_1); аналогично V_H и V_L – половины V. Тогда произведение UV можно записать в виде четырех произведений n/2-разрядных беззнаковых целых чисел, а также нескольких операций суммирования и арифметического сдвига:

$$UV = 2^n U_H V_H + 2^{n/2}(U_H V_L + U_L V_H) + U_L V_L . \quad (1)$$

В [2] приведен вариант перегруппировки формулы (1), позволяющий сократить количество произведений n/2-разрядных беззнаковых целых чисел до трех:

$$UV = (2^n + 2^{n/2})U_H V_H + (2^{n/2}+1)U_L V_L + \\ + 2^{n/2}(U_H - U_L)(V_L - V_H). \quad (2)$$

В [3] рассматриваются три способа распространения беззнакового умножения на знаковые операнды:

1) получить абсолютные значения каждого множителя, выполнить беззнаковое умножение, а затем изменить знак результата, если знаки множителей различаются.

2) Для формулы (1) при умножении старших половин использовать операцию умножения знакового на знаковое чисел, при умножении старших половин на младшие использовать операцию умножения знакового на беззнаковое чисел, а при умножении младших половин друг на друга использовать операцию умножения беззнаковых чисел. Также требуется операция знакового расширения промежуточных результатов.

3) Выполнить беззнаковое умножение, а затем скорректировать его результат следующим образом. Знаковое U рассматривается как беззнаковое число $U + 2^n U_{n-1}$, где U_{n-1} – единица, если U отрицательно, и ноль, если U положительно. Аналогично V рассматривается как $V + 2^n V_{n-1}$. Тогда их произведение:

$$(U + 2^n U_{n-1})(V + 2^n V_{n-1}) = UV + \\ + 2^n(VU_{n-1} + UV_{n-1}) + 2^{2n} U_{n-1} V_{n-1}. \quad (3)$$

Для того чтобы получить требуемый результат знакового произведения UV, надо вычесть из результата беззнакового умножения второе и третье слагаемые правой части формулы (3). Причем при умножении n-разрядных целых чисел третье слагаемое игнорируется, так как результат умножения полностью помещается в $2n$ разряда, соответственно, нет необходимости вычислять разряды старше, чем $2n-1$. Однако при умножении матриц существуют случаи, когда необходимо учитывать третье слагаемое из (3).

III. Применимость существующих алгоритмов умножения матриц для данной задачи

В рассматриваемых задачах компьютерного зрения используются обычные (“плотные” – dense) прямоугольные матрицы размером от нескольких десятков до двух-трех сотен строк и столбцов. Поэтому в данном случае неприменимы алгоритмы для работы с квадратными и с разреженными (sparse) матрицами, а также неэффективны быстрые способы умножения больших матриц, такие как алгоритм Штрассена и различные варианты его развития [4].

Всем хорошо знаком алгоритм умножения матриц с помощью скалярных произведений векторов-строк первого множителя на вектора-столбцы второго множителя. Однако этот способ плохо подходит для реализации на существующих DSP, так как элементы столбцов не расположены по соседним адресам памяти. В [5] предлагается специальная архитектура процессора для векторных операций со столбцами матриц, которая в теории должна обеспечить эффективное перемножение матриц с помощью векторных инструкций через скалярные произведения. На практике в используемом DSP необходимо переписывать элементы столбцов в последовательные блоки памяти или предварительно выполнять транспонирование матрицы – второго множителя, а затем вычислять скалярные произведения для векторов-строк. Оба варианта требуют значительных накладных расходов даже при использовании для транспонирования алгоритма Эклунда [7], который может быть эффективно реализован на DSP в нашем случае.

Иногда в произведении матрица – правый множитель должна быть транспонирована по требованиям реализуемого алгоритма компьютерного зрения. Тогда предпочтительнее не выполнять транспонирование явно, а выполнить матричное перемножение как скалярное произведение векторов строк обеих матриц. Однако в этом случае потребуется заполнять нулями неиспользуемые ячейки векторного регистра (zero-padding). Например, при ширине матрицы в 100 элементов и размере векторного регистра на 32 элемента, векторный регистр будет 3 раза заполнен полностью, а в четвертый раз в нем будут только 4 последних элемента из строки матрицы, следующие 28 элементов регистра требуется заполнить нулями. Также для эффективной реализации такого типа перемножения в процессоре должна быть инструкция суммирования элементов векторного регистра (sum reduction). К сожалению, используемый процессор не имеет подобной инструкции для вектора 32-разрядных целых чисел.

Как правило, при использовании SIMD инструкций предпочтительным является реализация матричного умножения через тензорное произведение векторов (tensor или outer product). Тензорное произведение вектора-столбца a на вектор строку b образует матрицу:

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \otimes \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix} = \begin{pmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_m b_1 & a_m b_2 & \cdots & a_m b_n \end{pmatrix}.$$

Матрицу – левый множитель можно рассматривать как вектор-строку состоящую из векторов столбцов, а матрицу – правый множитель как вектор-столбец, состоящий из векторов-строк. Тогда матричное произве-

дение есть сумма результатов тензорных произведений i векторов множителей:

$$\begin{pmatrix}\vec{a}_1 & \vec{a}_2 & \cdots & \vec{a}_n\end{pmatrix}\begin{pmatrix}\vec{b}_1\\ \vec{b}_2\\ \vdots\\ \vec{b}_n\end{pmatrix}=\sum_{i=1}^{n}\vec{a}_i\otimes\vec{b}_i. \qquad (4)$$

Для иллюстрации рассмотрим умножение матриц 3x3 и 3x2 с помощью тензорного произведения векторов:

$$\begin{pmatrix}a_{11} & a_{12} & a_{13}\\ a_{21} & a_{22} & a_{23}\\ a_{31} & a_{32} & a_{33}\end{pmatrix}\begin{pmatrix}b_{11} & b_{12}\\ b_{21} & b_{22}\\ b_{31} & b_{32}\end{pmatrix}=\begin{pmatrix}a_{11}\\ a_{21}\\ a_{31}\end{pmatrix}\otimes\begin{pmatrix}b_{11} & b_{12}\end{pmatrix}+$$

$$+\begin{pmatrix}a_{12}\\ a_{22}\\ a_{32}\end{pmatrix}\otimes\begin{pmatrix}b_{21} & b_{22}\end{pmatrix}+\begin{pmatrix}a_{13}\\ a_{23}\\ a_{33}\end{pmatrix}\otimes\begin{pmatrix}b_{31} & b_{32}\end{pmatrix}=$$

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

$$\begin{pmatrix}(a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31}) & (a_{11}b_{12}+a_{12}b_{22}+a_{13}b_{32})\\ (a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31}) & (a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32})\\ (a_{31}b_{11}+a_{32}b_{21}+a_{33}b_{31}) & (a_{31}b_{12}+a_{32}b_{22}+a_{33}b_{32})\end{pmatrix}.$$

Преимущества умножения матриц с помощью тензорного произведения векторов следующие: загрузка в векторные регистры и выгрузка из них происходит для последовательных адресов памяти; нет необходимости заполнять нулями неиспользуемые ячейки векторного регистра; не требуется операции суммирования элементов векторного регистра. Возможным ограничением является необходимость использования векторного регистра-аккумулятора для суммирования и накопления результатов из разных тензорных произведений векторов. Заметим, что при умножении матриц с n/2-разрядными числами элементы результирующей матрицы обычно требуют больше чем n разрядов, поэтому предпочтительно использовать, так называемый “широкий” векторный регистр-аккумулятор с разрядностью больше n.

На рис. 1 приведены зависимости количества тактов DSP, необходимых для умножения матриц размера *NxN* 16-разрядных целых чисел с помощью тензорных и скалярных произведений векторов. Причем предполагается, что при умножении через скалярные произведения матрица-правый множитель уже транспонирована и время, необходимое для транспонирования, не учитывается. При реализации использованы 16-разрядные векторные инструкции. Выгода умножения матриц через тензорное произведение векторов очевидна.

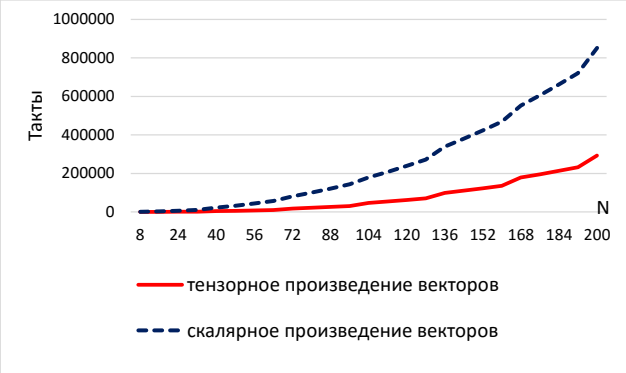

Рис. 1. Количество тактов DSP при умножении матриц размера *NxN* 16-разрядных целых чисел через тензорное и через скалярное произведение векторов

IV. Умножение матриц 32-разрядных чисел с помощью векторных инструкций для 16-разрядных чисел

В данной главе без потери общности приводится способ умножения прямоугольных матриц, содержащих 32-разрядные числа с фиксированной точкой, с помощью 16-разрядных векторных инструкций процессора. Основы арифметики чисел с фиксированной точкой приведены в [6]. При умножении чисел с фиксированной точкой сначала производится умножение операндов как знаковых целых чисел, а затем выполняется арифметический сдвиг вправо на число разрядов дробной части.

Для выбора способа обработки начнем с более простой задачи: умножения матриц 32-разрядных беззнаковых целых чисел. Сначала был реализован следующий способ умножения: при выполнении тензорного произведения векторов числа из столбцов левого множителя и вектора-стоки правого множителя разделяются на старшие и младшие 16-разрядные слова, для них с помощью векторных инструкций выполняется умножение по формуле (1), в результате чего получаются 32-разрядные вектора, которые суммируются в соответствии с (4). Для матриц размера 128x128 такой способ умножения оказался только на 30% быстрее, чем реализация с помощью 32-битных скалярных инструкций процессора, что является неудовлетворительным результатом. Анализ с помощью профайлера показал, что VLIW конвейер используется крайне неэффективно, почти нет инструкций, выполняемых одновременно.

Затем был реализован следующий способ: матрицы A и B были преобразованы в матрицы A_H, A_L и B_H, B_L 16-разрядных старших и младших слов, вычисления производятся как в формуле (1), но A_HB_H, A_HB_L, A_LB_H и A_LB_L являются произведениями матриц 16-разрядных чисел, каждое из которых единообразно реализуется через тензорное произведение векторов с помощью 16-разрядных векторных инструкций. На рис. 2 приведен график времени для умножения матриц двумя рас-

смотренными способами. Умножение с помощью четырех произведений матриц 16-разрядных чисел выполняется существенно быстрее. Кроме того, поскольку произведения матриц A_iB_k не зависят друг от друга, возникает возможность параллельной обработки.

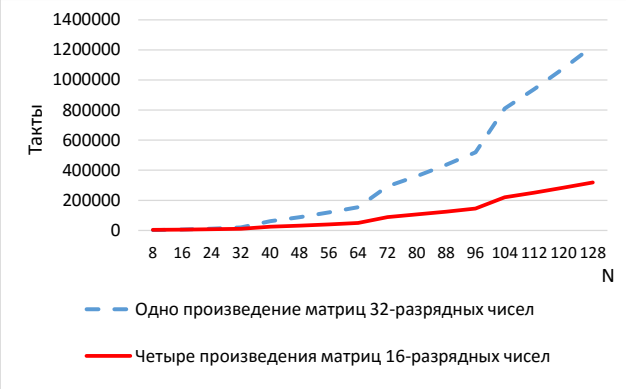

Рис. 2. Количество тактов DSP для умножения матриц размера *NxN* 32-разрядных беззнаковых целых чисел, используя два способа декомпозиции на 16-разрядные числа

Для обработки чисел с фиксированной точкой требуется преобразовать беззнаковое умножение в знаковое. В разделе II было перечислено три возможных способа распространения беззнакового умножения на знаковые операнды для скалярных значений. Какой из них может быть использован в случае, когда множители являются матрицами? Способ 1) при матричном произведении не применим, т.к. числа в результирующей матрице есть сумма произведений натуральных чисел из исходных матриц. Невозможно установить знак этого числа, зная знаки элементов матриц-множителей. При наличии в процессоре всех требуемых векторных инструкций, способ 2) теоретически позволяет достичь самой высокой производительности. Однако используемый DSP не обладает всем необходимым набором инструкций. Коррекция результата беззнакового умножения с помощью способа 3) вполне применима в нашем случае.

	79:64	63:48	47:32	31:16	15:0
	$(A_HB_H)_H$	$(A_HB_H)_M$	$(A_HB_H)_L$		
+		$(A_HB_L)_H$	$(A_HB_L)_M$	$(A_HB_L)_L$	
+		$(A_LB_H)_H$	$(A_LB_H)_M$	$(A_LB_H)_L$	
+			$(A_LB_L)_H$	$(A_LB_L)_M$	$(A_LB_L)_L$
-		Da_H	Da_L		
-		Db_H	Db_L		
-	S				

Рис. 3. Иллюстрация этапов умножения матриц *A* и *B*

Этапы умножения матриц A и B проиллюстрированы на рис. 3. В результате произведения матриц A_iB_k 16-разрядных чисел (для рассматриваемого диапазона размеров матриц) получается матрица 48-разрядных чисел, 16-разрядные слова которой обозначены нижними индексами: H – старшее, M – среднее и L – младшее. Произведение A_LB_L соответствует младшим 48 битам результата. Произведения A_LB_H и A_HB_L следует рассматривать как сдвинутые на 16 разрядов влево. Они соответствуют 63:16 битам результата. Произведение A_HB_H следует рассматривать как сдвинутое на 32 разряда влево. Оно соответствуют 79:32 битам результата. Таким образом, полный результат произведения матриц 32-разрядных целых чисел имеет 80 разрядов. Для получения итоговой матрицы требуется просуммировать соответствующие элементы матриц – результатов произведений 16-разрядных матриц и вычесть корректирующие матрицы D_a, D_b и S:

если A(i,j)<0, то Da(i,j) = B(i,j), иначе Da(i,j) = 0;

если B(i,j)<0, то Db(i,j) = A(i,j), иначе Db(i,j) = 0;

если A(i,j)<0 и B(i,j)<0, то S(i,j) = 1,иначе S(i,j) = 0,

где матрицы D_a и D_b рассматриваются как сдвинутые влево на 32 разряда, а матрица S – на 64 разряда. В нашем случае результат произведения есть матрица 32-разрядных чисел с фиксированной точкой, и старшие 16-бит 80-разрядного слова никогда не используются. На рис. 3 для примера в прямоугольную рамку заключены 16-битные слова, используемые для вычисления результата в формате 16.16 чисел с фиксированной точкой, т.е. на целую знаковую и на дробную части отводится по 16 разрядов. Можно не вычислять слагаемые за пределами рамки, что приведет к ускорению работы программы. При использовании чисел с фиксированной точкой возможно возникновение переполнения, и контроль над корректностью вычислений ложится на программиста. Заметим, что при точных целочисленных вычислениях с 80-разрядным результатом нельзя игнорировать S.

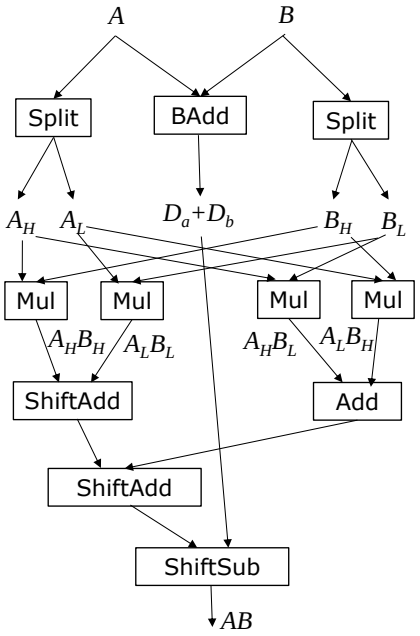

Рис. 4. Схема умножения матриц *A* и *B* 32-разрядных чисел с фиксированной точкой

Схема умножения матриц приведена на рис. 4. Блоки Split выполняют декомпозицию матриц 32-

разрядных чисел на матрицы младших и старших 16-разрядных слов. Блоки Mul выполняют матричное перемножение для матриц 16-разрядных чисел. BAdd предназначен для заполнения матриц D_a и D_b в зависимости от знака значений в матрицах A и B и суммирования D_a, D_b. Результаты умножения $A_H B_L$ и $A_L B_H$ суммируются в блоке Add. В блоках ShiftAdd и ShiftSub происходит соответственно сложение и вычитание матриц, числа в которых сдвинуты в позиции в зависимости от используемого формата чисел с фиксированной точкой.

Используемый DSP не имеет векторных инструкций для арифметического сдвига 32-разрядных чисел. Реализация сдвига через векторные инструкции для 16-разрядных чисел ведет к значительным накладным расходам. Поэтому предпочтительно использовать форматы чисел с фиксированной точкой с позицией точки на границе байт, то есть 0.32, или 8.24, или 16.16, или 24.8. Это позволяет заменить сдвиги на извлечение соответствующих байт или 16-разрядных слов.

На рис. 5 показано, какой процент от общего времени обработки занимает каждый этап в зависимости от размера матриц. С увеличением размера матриц время на выполнение четырех умножений матриц 16-разрядных чисел превышает 90% от общего времени. Поэтому оптимизации скорости работы именно этой операции следует уделить наибольшее внимание.

Вероятно, в системах на кристалле с одним IP блоком, содержащим DSP, целесообразно выполнять умножение по формуле (2), что позволит использовать три умножения матриц 16-разрядных чисел вместо четырех. Тем не менее, (1) имеет явное преимущество при параллельной обработке в системе с несколькими подобными IP блоками.

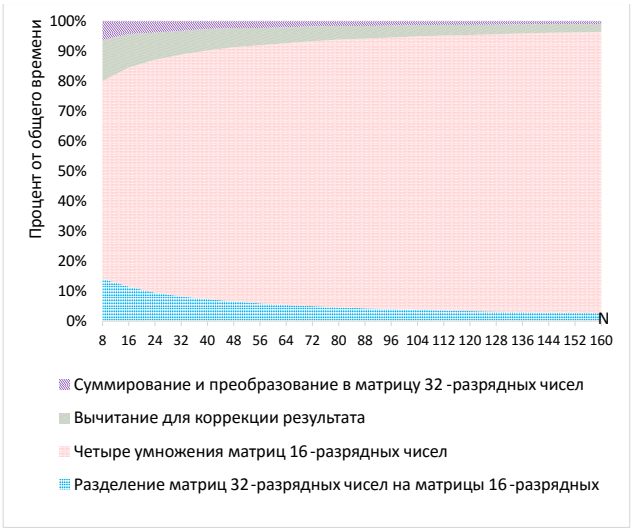

Рис. 5. Процент от общего времени обработки этапов умножения для матриц размера *NxN*

V. Приемы ускорения обработки

Если адрес загружаемого в векторный регистр фрагмента памяти кратен размеру векторного регистра (в байтах), то загрузка выполняется в несколько раз быстрее. Рис. 6 иллюстрирует влияние на время работы выравнивания адресов строк матриц 16-разрядных чисел по кратным размеру векторного регистра значениям. Если адреса строк не выровнены, то время работы возрастает многократно. В зависимости от размера строки при её выравнивании в конце строки могут появиться дополняющие (padding) байты. В реализованном алгоритме умножения нет необходимости заботиться о значениях дополняющих байтов.

Рис. 6. Влияние выравнивания строк на время обработки для матриц *NxN* 16-разрядных чисел

Развертывание циклов (loop unrolling), как правило, позволяет сделать работу конвейера процессора более эффективной. Современные компиляторы самостоятельно выполняют развертывание циклов, и в большинстве случаев делают это очень хорошо. Однако в случае таких вложенных циклов, какие имеют место при умножении матриц, иногда можно найти более оптимальный способ развертывания циклов. В частности, для используемого DSP удалось найти такой способ развертывания двух внутренних циклов, который позволил ускорить умножение матриц 16-разрядных чисел на 30%.

VI. Параллельное умножение матриц

Как было сказано выше, система на кристалле может иметь несколько идентичных IP блоков с DSP. Тогда целесообразно реализовать параллельное умножение матриц. Рассматриваемый алгоритм позволяет одновременно использовать два способа распараллеливания.

Рис. 7. Разбиение правой матрицы на вертикальные блоки для параллельной обработки

Первый способ проистекает из теории умножения матриц: правая матрица может быть разбита на верти-

кальные блоки (ширина блока кратна размеру векторного регистра), умножение левой матрицы на такой блок будет давать соответствующий блок в результирующей матрице, как показано на рис. 7. Более того, строки левой матрицы могут последовательно загружаться горизонтальными блоками по несколько строк. Второй способ легко увидеть на рис. 4: блоки, нарисованные на одном уровне, могут выполняться параллельно. В частности, одновременно могут выполняться декомпозиции матриц 32-разрядных чисел на матрицы 16-разрядных и умножения матриц 16-разрядных чисел. За счет использования контроллера прямого доступа в память и двойной буферизации задержки, связанные с копированием из глобальной в локальную память IP блока, пренебрежимо малы.

VII. Результаты

В данном разделе приведены результаты времени работы предложенного алгоритма (в тактах процессора) при реализации на одном DSP. Рис. 8 демонстрирует преимущество реализации через 16-разрядные векторные инструкции по сравнению с реализацией с помощью 32-разрядных скалярных инструкций. Для матриц размера 160x160 программа, использующая векторные инструкции, работает в 6 раз быстрее. Для матриц большего размера положительный эффект еще выше.

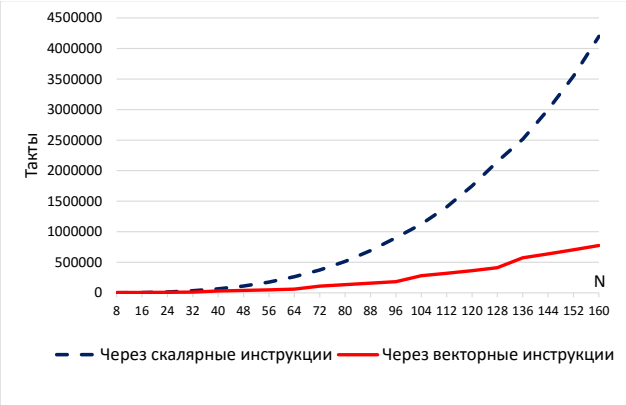

Рис. 8. Время работы произведения матриц 32-разрядных чисел с фиксированной точкой, реализованного через скалярные и через векторные инструкции

Понятно, что более-менее хорошо реализованный алгоритм, использующий векторные инструкции, работает быстрее, чем аналогичный через скалярные инструкции даже большей разрядности. Возникают вопрос. Нельзя ли его сделать алгоритм еще лучше? Где теоретический предел достижимой скорости работы? Умножение квадратных матриц размера *NxN* требует N^3 умножений. В идеальном случае за счет VLIW архитектуры операции с памятью и инструкции сложения могут выполняться одновременно с операциями умножения. В векторных регистрах помещается 32 16-разрядных значения. Алгоритм делает четыре умножения матриц 16-разрядных чисел. Каждая инструкция занимает такт процессора. Отсюда теоретически для матриц беззнаковых чисел можно достичь скорости обработки (в тактах процессора) $4N^3/32$.

На рис. 9 показаны теоретически достижимая скорость обработки, а также измеренные скорости умножения квадратных матриц *NxN* 32-разрядных беззнаковых чисел и чисел с фиксированной точкой в формате 16.16. Умножение матриц 32-разрядных беззнаковых целых чисел имеет производительность всего на 8-10% хуже оценки производительности для абсолютно идеального случая, что можно считать очень хорошим результатом. Коррекция результата для чисел с фиксированной точкой неизбежно ведет к дополнительным накладным расходам в 10-15%, но, тем не менее, скорость обработки остается достаточно высокой.

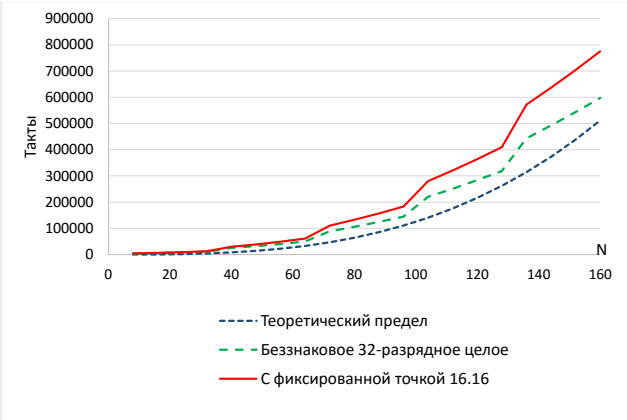

Рис. 9. Время работы произведения матриц 32-битных чисел по сравнению с идеальным случаем

Литература

[1] Ligorio G., Sabatini A.M. Extended Kalman Filter-Based Methods for Pose Estimation Using Visual, Inertial and Magnetic Sensors: Comparative Analysis and Performance Evaluation // Sensors. 2013, № 13. P. 1919-1941.

[2] Кнут Д.Э. Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд. – М.: Издательский дом “Вильямс”, 2000. -832 с.

[3] Уоррен Г.С. Алгоритмические трюки для программистов: испр. изд. – М.: Издательский дом “Вильямс”, 2004. -288 с.

[4] Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. // Алгоритмы: построение и анали: 2-е издание. Глава 28. Работа с матрицами — М.: Издательский дом “Вильямс”, 2005. — С. 833 - 839.

[5] Morad A., Yavits L., Ginosar, R. Efficient Dense and Sparse Matrix Multiplication on GP-SIMD // Proc. of IEEE Power and Timing Modeling, Optimization and Simulation conf., 2014. P. 1-8.

[6] Yates, R., Fixed-point arithmetic: An introduction // Digital Signal Labs, 2013, P. 15.

[7] Eklundh J.O. A fast computer method for matrix transposing // IEEE transactions on computers, Vol. 21. 1972. P. 801–803.

Matrix multiplication of n-bit fixed point numbers via n/2-bit vector instructions

I. Safonov, A. Ayupov, S. Burns

Intel, ilia.safonov@intel.com

***Keywords* — rectangular matrix multiplication, SIMD instructions, outer product, fixed point, DSP programming, parallel processing.**

ABSTRACT

The market of wearable and mobile devices is growing fast. Many mobile devices have a video camera and are capable of various computer vision tasks. We selected an IP core that contains a low-power DSP intended for image and video processing. The DSP has a VLIW/SIMD architecture, instructions for processing of 32-bit integer scalars and vector instructions for 16-bit integers. The IP core allows effective implementation of a large number of image processing algorithms. However there are some problems in computer vision which require intensive use of dense rectangular matrix multiplication, where matrix dimensions are several hundred by several hundred. An example of such a task is visual SLAM (Simultaneous Localization And Mapping) via various modifications of the Kalman filter [1].

In the general case the filter does calculations with floating point numbers. With some restrictions it is possible to use 32-bit fixed point numbers. Matrix multiplication via 32-bit scalars has unacceptably long runtime. It is necessary to employ vector instructions of processor, but DSP has SIMD instructions for only 16-bit integers. In the paper we propose a method for matrix multiplication of 32-bit fixed point numbers by means of vector instructions for 16-bit integers. The technique can be easily generalized to n-bit numbers and n/2-bit vector instructions. In addition we describe several approaches for performance optimization and parallel processing, which can be applied for various DSP engines.

It is well-known that algorithms for multiplication of n-bit unsigned integers via n/2-bit arithmetic operations for low (or the less significant) half of n-bit word and high (or the most significant) half. The book [2] depicts one such method by means of four multiplications for n/2 bit unsigned integers as well as several arithmetic shifts and additions. The monograph [3] describes a technique with three multiplications for n/2 bit unsigned integers. In general the method is less applicable for parallel processing, because it uses a larger number of additions, subtractions and shifts in comparison with that described in [3]. Also [3] declares three ways for adjustment of multiplication for signed integers: *a)* getting the absolute values of operands, then performing unsigned multiplication for low and high halves and then changing the sign of the result depending on the signs of operands; *b)* using multiplication of unsigned by unsigned integers for low halves of operands, multiplication of signed by signed integers for high halves, multiplications of signed by unsigned integers for high and low halves (this also requires a sign extension instruction); *c)* correction of multiplication outcome for unsigned integers by means of subtractions of operands depending on their sign.

For the given matrix dimensions, fast techniques such as various modifications of Strassen's algorithm [4] are ineffective. Matrix multiplication via dot product of rows of leftmost operand and columns of rightmost matrix is the most popular method. However the technique is ill-suited for an implementation on DSP with SIMD instructions, because elements of columns do not lie in the memory sequentially. Reference [5] depicts a specific architecture for effective access to elements of columns, but in practice a transposition of rightmost matrix or a copying of elements of columns to separate array is required. Both actions lead to a significant overhead in spite of a fact that transposition by Eklundh's algorithm [7] operates quite fast on our DSP. In addition usage of dot product requires zero-padding of rows and instruction for effective summation of vector register elements.

It is preferable to implement of matrix multiplication via a sum of outer (or vector tensor) products of columns of the leftmost operand and rows of rightmost matrix. So-called wide vector register should be used for accumulation of outer product sums. The algorithm has a lot of advantages: access is performed to sequential memory cells; there are no necessity in zero-padding and summation of vector register elements. We demonstrate processing times of matrix multiplication by means of dot products and outer products for matrices NxN of 16-bit unsigned integers. Both methods were implemented by means of vector instructions. A realization via outer products outperforms the alternative solution even without taking into account time required for transposition of rightmost matrix.

As a start, let's combine it all together: matrix product and multiplication of 32-bit unsigned integers via 16-bit arithmetic operations. There are two approaches for implementation by SIMD instructions. The first one is decomposition of 32-bit numbers onto vector registers of 16-bit low and high halves and calculation with those registers in the scope of single matrix multiplication procedure. We measured performance of this approach. For matrices 128x128 it operates only 30% faster in comparison with multiplication by means of scalar instructions for 32-bit numbers. This is unsatisfactorily. We analyzed our program using a profiler in order to

explore the causes of the delays. This showed that this approach does not employ the VLIW pipeline effectively. The second approach is decomposition of both matrices of 32-bit integers onto matrices of 16-bit numbers and performing four matrix products between those matrices. Outcomes of four matrix multiplications are combined to final matrix of 32-bit integers. We demonstarte plots of processing speed depending on matrix dimention for both approaches. The second one is much faster.

Further, let's modify the second approach for 32-bit fixed point numbers. The main concepts of arithmetic with fixed point numbers is described in [6]. Multiplication of fixed point numbers can be considered as multiplication of the signed integers and arithmetic shift to the right on length (in bits) of a fractional part. Thus, it is necessary to extend the algorithm of matrix multiplication of unsigned integers to a method for matrices of signed integers. Existing approaches for such extension for scalars were enumerated above. Way *a)* is unfeasible for matrix multiplication, because each element of the final matrix is the sum of products of elements of matrix-operands. IVP-IP does not have all the necessary vector instructions for way *b)*. Way *c)* can be applied in our case.

Finally, we have the following algorithm for *AB* matrix multilplication. For a given range of dimenions a matrix multiplication of 16-bit numbers produces a matrix of 48-bit numbers. We designate matrices with 16-bit halves of 32-bit numbers from the *A* and *B* matrix by indices: *L*-low, *H*-high. The first step is decomposition of *A* on A_L and A_H, *B* on B_L and B_H. Four matrix multiplications of A_LB_L, A_HB_L, A_LB_H and A_HB_H are performed via outer products independently from each other. A_LB_L corresponds to a lower 48 bit of outcome. Products A_LB_H and A_HB_L should be considered as shifted on 16 bit to the left, accordingly they correspond to the range from 63 to 16 bit. A_HB_H should be considered as shifted on 32 bit to the left, accordingly it corresponds to the range from 79 to 32 bit. Maximal length of elements of resulting matrix is 80 bits. For obtaining of resulting matrix it is necessary to make summation of shifted outcomes of A_LB_L, A_HB_L, A_LB_H, A_HB_H and to subtract of shifted matrices D_a, D_b and *S*:

if A(i,j)<0, then Da(i,j) = B(i,j), else Da(i,j) = 0;

if B(i,j)<0, then Db(i,j) = A(i,j), else Db(i,j) = 0;

if A(i,j)<0 and B(i,j)<0, then S(i,j) = 1,else S(i,j) = 0,

where matrices D_a and D_b should be considered as shifted on 32 bits to the left, matrix S as shifted on 64 bits to the left. Depending on fixed point format we extract corresponding bits from 80-bit outcome. For example, for 16.16 fixed point numbers (that is length of integer part is 16 bits and length of fractional part is 16 bits) we get bits from 47 to 16.

There are two optimization tricks, which are able to decrease processing time significantly. The first is aligning of addresses of matrix rows. Addresses should be divisible by the length (in bytes) of the vector register. There is no necessity to zero-pad matrix multiplication via outer products. Padding bytes can have any values. The second trick is manual loop unrolling. In general modern compilers automatically make effective optimization including loop unrolling. However we were able to write faster code compared with the optimizing compiler.

Decompositions on matrices of 16-bit numbers and products of the matrices can be done in parallel. It can be applied in conjunction with "natural" approach for parallelization of matrix multiplication by splitting of rightmost matrix on vertical blocks, multiplication of leftmost matrix on each block and concatenation of outcomes. Width of block should be divisible on width of vector register.

Processing time of proposed algorithm realized by means of 16-bit vector instructions for matrix 160x160 of 16.16 fixed point numbers is more than 6 times faster in comparison with implementation via 32-bit scalar instructions. What is the theoretical limit of achievable performance? The multiplication of square matrices of size *NxN* requires N^3 products for scalars. In the ideal case due to VLIW pipeline, memory operations and additions can be performed simultaneously with the multiplications instructions. A vector register of our DSP contains 32 16-bit numbers. Algorithm does four matrix multiplications. Assume, a instruction requires a cycle of a processor. Thus, theoretical limit is $4N^3/32$. Matrix multiplication of 32-bit unsigned integers has a performance of only 8-10% worse in comparison with evaluation for an ideal case. This is pretty good. For matrix of fixed point numbers the generation and subtraction of D_a and D_b lead to 10-15% overhead. Nevertheless processing speed is high.

REFERENCES

[1] Ligorio G., Sabatini A.M. Extended Kalman Filter-Based Methods for Pose Estimation Using Visual, Inertial and Magnetic Sensors: Comparative Analysis and Performance Evaluation, Sensors, 2013,no. 13, pp. 1919-1941.

[2] Knuth D.E. The art of computer programming: Vol. 2 / Seminumerical algorithms, Third edition. 1998, Addison-Wesley.

[3] Warren H.S. Jr. Hacker's Delight. 2002, Addison-Wesley.

[4] Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. Chapter 28: Section 28.2: Strassen's algorithm for matrix multiplication, pp. 735–741.

[5] Morad A., Yavits L., Ginosar, R. Efficient Dense and Sparse Matrix Multiplication on GP-SIMD. 2014. Proc. of IEEE Power and Timing Modeling, Optimization and Simulation conf., pp. 1-8.

[6] Yates, R., Fixed-point arithmetic: An introduction // Digital Signal Labs, 2013, P. 15.

[7] Eklundh J.O. A fast computer method for matrix transposing. IEEE transactions on computers, Vol. 21, 1972, pp. 801–803.