Применение векторного сопроцессора для оптимизации функций библиотеки линейной алгебры

к.ф.м.н. А.А. Бурцев
НИИСИ РАН
burtsev@niisi.msk.ru

Аннотация

В НИИСИ РАН в качестве расширения универсальных микропроцессоров семейства КОМДИВ разработан специализированный 128-разрядный сопроцессор, позволяющий ускорять вычисления над векторами комплексных и вещественных чисел одинарной и двойной точности. В докладе представлены результаты применения этого сопроцессора, направленные на повышение скорости исполнения основных функций обработки векторов и матриц, обычно используемых для решения типовых задач линейной алгебры. Рассматриваются приёмы оптимизации кода с учётом особенностей сопроцессора, выявляются недостатки, препятствующие его эффективному применению, и предлагаются возможные пути их преодоления.

Введение

Реалии современного мира диктуют острую потребность создания отечественных микропроцессоров, способных обеспечивать высокую производительность научно-технических и инженерных расчётов даже в экстремально жёстких условиях их эксплуатации.

Для удовлетворения этой потребности в НИИСИ РАН разрабатывается [1] семейство высокопроизводительных микропроцессоров КОМДИВ, в которых в качестве расширения базовой архитектуры предлагаются специализированные сопроцессоры, ориентированные (согласно принципу “встречной оптимизации”) на ускоренное исполнение заданного набора математических функций, наиболее часто употребляемых при решении задач определённой области применения.

Так, микропроцессор К64РИО (1890ВМ6) помимо обычного 64-разрядного сопроцессора (СР1) плавающей арифметики был дополнен 64-разрядным сопроцессором (СР2) комплексной арифметики, а микропроцессор К128РИО (1890ВМ7) был оснащён 128-разрядным сопроцессором (СР2) обработки сигналов.

В дальнейшем вычислительный блок сопроцессора комплексной арифметики был расширен до 128-разрядов и дополнен рядом полезных команд, позволяющих ускорить исполнение ряда типичных операций обработки векторов и матриц. В результате такого развития получился 128-разрядный сопроцессор (СР3), который можно охарактеризовать как векторный. Он, как предполагается, будет функционировать в составе новых высокопроизводительных микропроцессоров (1890ВМ8) семейства КОМДИВ.

В докладе объясняется, как путём применения этого сопроцессора обеспечивается значительный выигрыш в скорости исполнения наиболее употребительных функций обработки векторов и матриц, используемых для решения типичных задач линейной алгебры. Приводятся оценки такого выигрыша, полученные экспериментальным путём для ряда характерных функций известной библиотеки линейной алгебры BLAS [2], которая часто используется в приложениях, служащих образцовыми примерами тестов компьютерной производительности.

1. Общая характеристика векторного сопроцессора

Векторный сопроцессор (CPV) содержит 64 128-битных регистра, в каждом из которых можно хранить:

- 1 комплексное число двойной точности (DC);
- 2 комплексных числа одинарной точности (SC);
- 2 вещественных числа двойной точности (DR);
- 4 вещественных числа одинарной точности (SR).

Для каждого из этих 4-х форматов значений, помещённых в его регистры, CPV обеспечивает соответствующие ему вычислительные команды. Например, каждая из команд умножения с накоплением (см. табл.1) выполняет свойственную ей операцию для одной тройки (z,y,x) величин типа DC, либо двух троек типа SC или DR, либо сразу 4-х троек типа SR.

Таблица 1. Вычислительные команды группы умножения

1 DC (9)*	2 SC (7)*	2 DR (5)*	4 SR (4)*	cmd z,y,x
cmul.d	**cmul.s**	**vmul.d**	**vmul.s**	z=y·x
cmadd.d	**cmadd.s**	**vmadd.d**	**vmadd.s**	z=z+y·x
cmsub.d	**cmsub.s**	**vmsub.d**	**vmsub.s**	z=z-y·x
cmaddsub.d	**cmaddsub.s**	**vmaddsub.d**	**vmaddsub.s**	z=z+y·x
(t)* - *длительность в тактах (если команда уже в кэше)*				y=z-y·x

Длительность вычислительных команд зависит от типа обрабатываемых величин (см. в скобках: 4 такта для SR, 5 для DR, 7 для SR и 9 для DC). Но поскольку CPV разрешает на каждом такте начинать исполнение новой команды вычислительного потока, то в итоге можно добиться такой максимальной производительности CPV, при которой **n** его вычислительных команд смогут исполниться всего за **n**+8 тактов.

Правда, для этого необходимо параллельно с вычислениями осуществлять ещё и своевременную подкачку обрабатываемых данных из памяти в регистры и обратно. Но такую возможность CPV предоставляет.

Таблица 2. Команды работы с памятью

	команды загрузки	команды сохранения
256 /32 L2(5)*	**vldq**, **vldqx**	**vsdq**, **vsdqx**
64 /8 L1(3)*	**vld**, **vldh**, **vldx**, **vldhx**, **vldlx**	**vsd**,**vsdh**, **vsdx**, **vsdhx**
128 /16 L1(3)* **если данные уже* *в L1(L2)-кэше*	**vldm**, **vldd**, **vlddh**, **vldmx**, **vlddx**, **vlddhx**, **vlddlx**	**vsdm**, **vsdd**, **vsddh**, **vsdmx**, **vsddx**, **vsddhx**

Команды CPV для работы с памятью (см. табл.2) позволяют загрузить 128-битное значение целиком в регистр или заполнить его старшую и младшую половину 64-битными значениями по отдельности. Можно одной командой (vldq) загрузить два соседних регистра двумя 128-битными смежными словами из памяти. Аналогичные команды (vsd, vsdm, vsdq) предусмотрены и для сохранения значений регистров в памяти.

Для указания виртуального адреса в этих командах можно использовать базовую адресацию с относительным смещением (vld) или базовую индексную (vldx). Получаемый адрес должен быть выровнен на границу обрабатываемого слова, т.е. кратен 8, 16 или 32 (см./d).

Заметим, что описываемые здесь команды CPV нацелены на ускоренное исполнение в 128-разрядном режиме и не обеспечивают проверки на возникновение каких-либо исключительных ситуаций.

Архитектурой КОМДИВ предусмотрена возможность в каждом такте взять на исполнение две очередные команды, если эти команды разных потоков. Это позволяет совместить во времени вычислительные команды над одной группой данных с командами загрузки/сохранения в/из памяти другой группы данных.

В CPV самыми продуктивными командами над вещественными числами являются команды перемножения матрицы 2×2 на вектор из 2-х чисел, которые предназначены ускорить одну из важнейших операций линейной алгебры – перемножение матриц. Одна такая команда перемножения с накоплением (см. mvmadd в табл.3) выполняет 8 операций (4 умножения и 4 сложения) над вещественными числами двойной точности или 16 арифметических операций одинарной точности.

Таблица 3. Команда перемножения матрицы на вектор

M_1=(a,b), M_2=(c,d) V=(x,y)	**mvmadd.d** Z,M,V (9)* M *задаёт два соседних регистра* (M_1,M_2)	Z= Z+M×V : Z= Z+(a·x+b·y,c·x+d·y)
M_1=(a,b,c,d), M_2=(e,f,g,h) V=(x,y,z,u)	**mvmadd.s** Z,M,V (7)* M *задаёт два соседних регистра* (M_1,M_2)	Z= Z+ (a·x+b·y, e·x+f·y, c·z+d·u, g·z+h·u)

А самая “мощная” команда (cmaddsub), предусмотренная в CPV над комплексными числами, осуществляет 10 арифметических операций (4 умножения и 6 сложений) над вещественными числами двойной точности или 20 арифметических операций над вещественными числами одинарной точности. Одной такой командой реализуется базовая операция цифровой обработки сигналов, называемая “бабочкой Фурье”.

Таким образом, векторный сопроцессор позволяет достичь (на тактовой частоте 1 ГГц) пиковой производительности 16 ГФлопс на задачах перемножения матриц и 20 ГФлопс на задачах преобразования Фурье.

2. Краткая характеристика библиотеки BLAS

BLAS (Basic Linear Algebra Subprograms) – это библиотека подпрограмм (функций), которая создавалась, чтобы обеспечить универсальным интерфейсом (API) разработчиков прикладных программ, нацеленных на решение разнообразных задач линейной алгебры (например, таких, как получение LU-разложения матрицы, вычисление определителя матрицы, решение систем линейных уравнений и др.).

BLAS первоначально создавалась на языке Фортран, но впоследствии стала доступной и для разработчиков Си-программ. В Интернете свободно доступны несколько разновидностей этой библиотеки. Среди них особый интерес представляет GotoBLAS [3], код которой оптимизирован под различные машинные архитектуры. Предпринимаются попытки повысить производительность BLAS-функций за счёт распараллеливания на многоядерных процессорах [4] или с помощью векторного расширения (например, Intel AVX [5]).

Функции BLAS принято подразделять на 3 уровня. Номер уровня функции определяется порядком величины количества операций с плавающей запятой (а также количеством перемещений данных), которые потребуется исполнить в данной функции (см. табл.4).

Таблица 4. Классификация BLAS-функций

	операций/перемещений данных	типичные функции
1	O(N) / O(N)	**x**SCAL, **x**AXPY, **x**DOT
2	$O(N^2)$ / $O(N^2)$	**x**GEMV, **x**TRSV
3	$O(N^3)$/ $O(N^2)$	**x**GEMM, **x**TRSM
N - размер вектора или порядок матрицы, **x**=Z,C,D,S		

Названия функций приводятся здесь с буквой-префиксом x (Z,C,D,S), означающей, для какого типа величин (DC, SC, DR, SR) предназначена эта функция.

На каждом уровне выделим самые характерные функции, которые чаще всего употребляются в приложениях, использующих эту библиотеку (см. табл.5).

Таблица 5. Характерные BLAS-функции

уровень	имя (параметры)	схематичное описание функции
1	**x**SCAL (n,α,X,ix)	$X=\alpha\cdot X$: { $X_i=\alpha\cdot X_i$, i=1…n }
1	**x**AXPY (n,α,X,ix,Y,iy)	$Y=\alpha\cdot X+Y$: { $Y_i=\alpha\cdot X_i+Y_i$, i=1..n }
1	**x**DOT (n,X,ix,Y,iy)	$dot=X^T\times Y$: { $dot=\sum(X_i\cdot Y_i)_{i=1..n}$ }
2	**x**GEMV (tz,m,n,α,Z,lz,X,ix,β,Y,iy)	$Y=\alpha\cdot Z^*_{m\times n}\times X+\beta\cdot Y$, $Z^*=Z\|Z^T\|Z^H$: { $Y_i=\beta\cdot Y_i+\alpha\cdot\sum(Z^*_{ik}\cdot Y_k)_{k=1..n}$, i=1..m }
2	**x**TRSV (ul,tz,d,n,Z,lz,Y,iy)	$Z\times X=Y$? → $Y_{n\times 1}=Z^{*-1}_{n\times n}\times Y_{n\times 1}$ $Z^{*-1}=(Z)^{-1}\|(Z^T)^{-1}\|(Z^H)^{-1}$
3	**x**GEMM (tx,ty,m,n,k,α,X,lx,Y,ly,β,Z,lz)	$Z_{m\times n}=\alpha\cdot X^*_{m\times k}\times Y^*_{k\times n}+\beta\cdot Z_{m\times n}$: { $Z_{ij}=\beta\cdot Z_{ij}+\alpha\cdot\sum(X^*_{is}\cdot Y^*_{sj})_{s=1..n}$ }, i=1..m, $X^*=X\|X^T\|X^H$, $Y^*=Y\|Y^T\|Y^H$ j=1..n
3	**x**TRSM (s,ul,tz,d,m,n,α,Z,lz,Y,ly)	$Z\times X=\alpha\cdot Y$? → $Y_{m\times n}=\alpha\cdot Z^{*-1}_{m\times m}\times Y_{m\times n}$ $X\times Z=\alpha\cdot Y$? → $Y_{m\times n}=\alpha\cdot Y_{m\times n}\times Z^{*-1}_{n\times n}$ $Z^{*-1}=(Z)^{-1}\|(Z^T)^{-1}\|(Z^H)^{-1}$

Далее проанализируем, какой выигрыш в повышении производительности этих функций может обеспечить применение векторного сопроцессора CPV. А также выясним причины, почему для некоторых функций в отдельных случаях ожидаемого подъёма производительности добиться не удаётся.

3. BLAS-функции 1-го уровня

Каким бы совершенным не был имеющийся аппаратный исполнитель, эффективность реализации требуемой функции, в конечном счёте, во многом определяется совершенством её программного кода. Поэтому анализ возможного выигрыша в производительности, который может обеспечить векторный сопроцессор (CPV), будем сопровождать обсуждением известных приёмов оптимизации кода, которые дают эффект при построении программы с учётом как общих принципов архитектуры КОМДИВ, так и отдельных особенностей сопроцессора CPV, в частности.

3.1 DSCAL

Начнём такой анализ с функций, обрабатывающих одиночный вектор. А в качестве типичного их представителя рассмотрим функцию умножения вектора на скаляр: **x**SCAL(n,α,X,ix). Под вектором здесь и далее понимается ряд значений, расположенных в памяти с некоторым шагом (ix), начиная с заданного адреса (X). Заметим, что элементы вектора будут следовать в памяти рядом друг за другом только при шаге ix=1. Далее такой вектор будем называть непрерывным, чтобы отличать его от произвольного, для которого ix≠1. Примерами непрерывного вектора могут служить строки матрицы в Си-программе, а примерами произвольного – её столбцы. (Для Фортран-программы будет всё наоборот, т.к. компилятор расположит её в памяти уже не по строкам, а по столбцам). Заметим, что адрес i-го элемента (i=0,..,n-1) произвольного вектора потребуется вычислять по формуле: X+i×ix×sT, где sT – размер элементов (в байтах), равный sT=4,8,8,16 для значений типа SR,DR,SC,DC соответственно.

Основной цикл функции DSCAL(n,α,X,ix), составленной на языке Си в расчёте на обработку (на CP1) непрерывного вектора (ix=1) с элементами типа DR:

```
void DSCAL(int n, double a, double *X, int ix) {
 int i; /* в предположении, что ix=1 */
 for (i=0; i<n; i++) X[i]=a*X[i];
}//DSCAL
```

будет переведён (если не применять оптимизацию) в ассемблерный код примерно следующего вида:

```
Loop:  ## вариант 1
       Ldc1   FX,0(Xi)
       Mul.d  FX,FX,FS
       Sdc1   FX,0(Xi)
       addiu  Xi,Xi,sT;
       addiu  I,I,1
       bne    I,N,Loop
```

назначение регистров:

FS = значению параметра **a** ;
FX = значению элемента **X**[i] ;
I – регистр-счётчик цикла;
Xi – регистр адреса элемента **X**[i];
N – регистр параметра **n** ;
sT = 8 (размер элемента в байтах)

Выделенные в этом цикле основные команды загрузки (**L**), сохранения (**S**) и вычислительного действия (**M**):

Ldc1 ; Mul.d ; Sdc1	L-M----S-	такты: 2+5+2=9

должны исполняться строго последовательно и потому на обработку каждого элемента потребуется **9** тактов (на СР1), а на обработку всего вектора из **n** элементов – **9·n** тактов соответственно.

Ускорить обработку вектора можно за счёт совмещения во времени вычислительных операций с операциями загрузки/сохранения. Для этого используются следующие приёмы оптимизации кода.

Приём №1. Во-первых, можно просто изменить порядок следования этих команд в цикле так, чтобы на i-ом шаге каждого цикла вычислительная операция для i-го элемента сопровождалась опережающей загрузкой (i+1)-го элемента и сохранением результата для (i-1)-го элемента:

```
Sdc1 F2,-sT(Xi)
Mul.d F2,F1,FS
Ldc1  F1,sT(Xi)
```

```
i-1: L-M----S-
  i:    L-...M----S-
i+1:          L-..M----S-
```

Приём №2. Во-вторых, можно предусмотреть в теле цикла две фазы: в 1-ой фазе каждого шага цикла вычислительную обработку элементов одной группы (A) сопровождать загрузкой и сохранением элементов другой группы (B), а во 2-ой фазе каждого шага цикла – наоборот:

```
Mul.d FA,FA,FS
Sdc1  FB,0(Bi)
Ldc1  FB,sX(Bi)
Mul.d FB,FB,FS
Sdc1  FA,0(Ai)
Ldc1  FA,sX(Ai)
```

потактовая диаграмма:

```
Bi-1:     S-
Ai  :  L-M----.S-
Bi  :       L-M----.S-
Ai+1:            L-M----.S-
```

|на i-ом шаге|

В этом варианте 6 команд цикла для обработки 2-х элементов займут 10 тактов, а для обработки вектора длиной **n** (n=2·p) потребуется **5·n** (10·p) тактов. Но это без учёта особой обработки элементов первой A-группы и последней B-группы, а также команд обеспечения цикла: `I:=I+1; Ai:=Ai+sX; Bi:=Bi+sX; (sX=2·sT).`

Приём №3. Допустим, разрешается очередную команду вычислительного потока начинать исполнять уже на следующем такте, не дожидаясь завершения предыдущей команды этого же потока. (СР1 это позволяет в особом режиме работы). Тогда в целях дальнейшей оптимизации кода можно организовать кратную обработку вектора группами по **k** элементов.

На каждом шаге цикла будем применять тройку основных команд (L,M,S) попеременно ко всем элементам одной группы (j=1,..,k), а **k** возьмём таким (=5), чтобы следующая команда этой тройки могла применяться к j-ому элементу группы уже без задержки:

```
Ldc1 F1,1*sT(Xi);...
 Ldc1 Fk,k*sT(Xi);
 Mul.d F1,F1,FS; ...
 Mul.d Fk,Fk,FS;
 Sdc1 F1,1*sT(Xi);...
 Sdc1 Fk,k*sT(Xi);
 adDiu Xi,Xi,k*sT
```

```
j= |  16  тактов  |
1: L-...M----S-
2:  L-...M----S-
3:   L-...M----S-
4:    L-...M----S-
5:     L-...M----S-
                  D
```

Этот вариант в каждом цикле обрабатывает группу из 5-ти элементов за 16 тактов, а вектор длиной **n** (n=5·p) – уже примерно за **n·16/5** (16·p) тактов соответственно (это если не учитывать команды организации цикла).

Приём №4. Архитектура КОМДИВ позволяет в каждом такте приступить к исполнению сразу двух команд разных потоков. Это позволяет получить выигрыш в тактах (почти в 2 раза), если удачно расставить вычислительные команды в коде так, чтобы они попеременно чередовались с командами работы с памятью.

Объединим кратную обработку вектора (приём №3) с двухфазным построением цикла (приём №2) так, чтобы вычислительные операции над k элементами одной группы (Ai, i=1..k) совместить с параллельным исполнением операций загрузок и сохранений для k элементов другой группы (Bi, i=1..k), а затем наоборот.

Проиллюстрируем это диаграммой (см. таблицу 6). На каждом шаге цикла на обработку 10 элементов затрачивается 20 тактов (по 10 тактов на 16 команд каждой фазы), значит, для обработки вектора из **n** элементов в этом варианте потребуется **n·2** (20·p) тактов.

Такой вариант обеспечивает предельную производительность, которую можно "выжать" для функции DSCAL при обработке непрерывного вектора на CP1.

При обработке же произвольного вектора команду приращения адреса (ad**D**iu) приходится выполнять уже после каждой команды загрузки или сохранения элемента. Поэтому на каждом шаге цикла придётся исполнить 50 команд (по 5L+5D+5M+5S+5D=25 в каждой фазе). В лучшем случае на это уйдёт 25 тактов. Значит, на обработку произвольного вектора из **n** элементов (n=10·p) потребуется **2,5·n** (25p=25·n/10) тактов.

Таблица 6. Потактовая диаграмма (n=10·p, k=5)

```
     |  1-ая фаза    |  2-ая фаза  |
A1: M----.....S-    L-
A2:  M----.....S-    L-
A3:   M----.....S-    L-
A4:    M----.....S-    L-
A5:     M----....S-    L-
                  D
B1: S-    L-    M----
B2:  S-    L-    M----
B3:   S-    L-    M----
B4:    S-    L-    M----
B5:     S-    L-    M----
        D
```

Проанализируем теперь, как можно поднять производительность этой функции с помощью векторного сопроцессора (CPV).

Пусть подлежащий обработке непрерывный вектор выровнен по адресу, кратному 32. Тогда можно выгодно использовать команды **vLdq** и **vSdq** для загрузки и сохранения сразу 4-х его элементов в 2 соседних регистра. Необходимые вычисления для такой четвёрки можно осуществить двумя командами **vMul.d**. Применяя уже знакомые приёмы кратной групповой двухфазной обработки для k=16 и n=32·p, получаем такой ассемблерный код для тела основного цикла, обрабатывающего 2 группы A и B по 16 элементов в каждой:

	Loop: # для группы A:	# для группы B:
1	v**M**ul.d VA**j**,VA**j**,VS ;	v**S**dq VB**j**,**j***sT(Bi);
ф	... для **j**=0,1,2,3	... для **j**=0,2,4,6
а		ad**D**iu Bi,Bi,2*k*sT
з	v**M**ul.d VA**j**,VA**j**,VS ;	v**L**dq VB**j**,**j***sT(Bi);
а	... для **j**=4,5,6,7	... для **j**=0,2,4,6

2	V**S**dq VA**j**,**j***sT(Ai) ;	v**M**ul.d VB**j**,VB**j**,VS ;
ф	... для **j**=0,2,..,14	... для **j**=0,1,..,7
а	ad**D**iu Ai,Ai,2*k*sT	
з	v**L**dq VA**j**,**j***sT(Ai) ;	v**M**ul.d VB**j**,VB**j**,VS ;
а	... для **j**=0,2,..,14	... для **j**=8,1,..,15
	addiu I,I,1 ; bne I,**P**,**Loop**	

В нём в каждой фазе 8 команд v**M**ul.d совмещаются с 4 командами v**S**dq и 4 командами v**L**dq, так что на все эти 32 команды будет затрачиваться 32/2=16 тактов. Значит, для обработки вектора из **n** элементов потребуется 16·p, т.е. примерно **n/2** тактов. (Это опять же без учёта команд организации цикла).

Таким образом, применение CPV позволяет ускорить почти **в 4 раза** (n/2 вместо 2n тактов) исполнение функции DSCAL для непрерывного вектора из DR-элементов. Но такой выигрыш, к сожалению, не получается получить при обработке произвольного вектора. Попробуем разобраться, почему.

Всё дело в том, что элементы произвольного вектора не располагаются в памяти друг за другом, поэтому при реализации основного цикла функции DSCAL:

```
for (i=0,I=0;i<n;i++,I+=ix) X[I]=a*X[I];
```

на загрузку 4-х элементов в два регистра (V0,V1) вместо одной команды (vldqx) теперь требуется 8 команд:

v**L**d**h**x V0,I(X); ad**D**u I,I,sX	## если бы ix=1
v**L**d**l**x V0,I(X); ad**D**u I,I,sX	v**L**dqX V0,I(X)
v**L**d**h**x V1,I(X); ad**D**u I,I,sX	addu I,I,4*sT
v**L**d**l**x V1,I(X); ad**D**u I,I,sX	## sX= **ix**·sT

Аналогичная серия команд (v**S**d**h**x, v**S**d**l**x) потребуется и вместо одной команды сохранения (vsdqx).

При кратной обработке вектора группами по 16 элементов на 8 вычислительных команд (v**M**ul.d) теперь приходится потратить 32 команды для загрузки и 32 для сохранения элементов. Соотношение этих команд при обработке группы непрерывного вектора было **8:9** (8M|4L+4S+1D), а для произвольного получается **8:64** (8M|16L+16S+32D). Даже если эти 72 команды удастся удачно распределить попарно (чтобы каждый такт захватывать на исполнение 2 команды), то на обработку одной группы уйдет 36 тактов, а на вектор длиной **n**=16·p потребуется **2,25·n** (36·n/16) тактов.

Таким образом, функцию DSCAL, обрабатывающую произвольный вектор из DR-элементов, удастся ускорить на CPV всего лишь **на 11%** (2,5n/2,25n=1,11).

Такой удручающий результат сначала может показаться странным. В самом деле, ведь команда v**M**ul.d CPV позволяет (за те же 5 тактов) исполнить сразу две вычислительные операции вместо такой же одной (умножение вещественных двойной точности), которую исполняет команда **M**ul.d CP1. Следовательно, можно было бы рассчитывать на то, что CPV обеспечит ускорение функции DSCAL хотя бы в 2 раза. Но такой расчёт оказывается ошибочен, ибо он не учитывает, что ускорение сам**о**й вычислительной команды, действующей над регистрами, не приведёт к значительному подъёму производительности всей функции, если оно не будет поддержано ускоренной работой команд, обеспечивающих доставку обрабатываемых значений из памяти в регистры и обратно.

3.2 SSCAL

Отсутствие команд, обеспечивающих быструю загрузку и требуемую упаковку в регистры сразу нескольких элементов произвольного вектора, а также обратную распаковку и сохранение их в памяти, ещё более значительно сказывается на замедлении производительности функции SSCAL при обработке значений типа SR. Используемая для её реализации на CPV основная вычислительная команда v**M**ul.s позволяет заменить 4 аналогичные команды **M**ul.s CP1. Казалось бы, можно рассчитывать, что функция SSCAL станет работать на CPV в 4 раза

быстрее, чем на CP1. Но оказывается, что на такой выигрыш можно рассчитывать только лишь при обработке непрерывного вектора. При обработке же произвольного вектора вариант реализации этой функции на CPV даже уступает в скорости исполнения варианту её реализации на CP1. Рассмотрим подробнее, почему это происходит.

Заметим, что CPV не предоставляет команд для загрузки в свои регистры (и сохранения из них) одиночных 32-битных значений. Поэтому SR-элементы произвольного вектора приходится переправлять из памяти в регистры CPV (и обратно) отдельными 64-битными частями через регистры CP1, используя особые команды для обмена с ними (см. таблицу 7).

Таблица 7. Команды обмена между регистрами CPV и CP1

чтение из регистра CP1 в регистр CPV (3)*	**vmthfp** FR,VR	VR.Hi:=FR
	vmtlfp FR,VR	VR.Lo:=FR
запись в регистр CP1 из регистра CPV (3)*	**vmfhfp** FR,VR	FR:=VR.Hi
	vmflfp FR,VR	FR:=VR.Lo

Одной командой vldq можно загрузить в два регистра (V0,V1) сразу 8 SR-элементов непрерывного вектора. Но для загрузки тех же 8-ми SR-элементов произвольного вектора потребуется аж 24 команды, т.к. на загрузку 4-х SR-элементов в каждый регистр Vi (i=0,1) приходится тратить по 12 команд:

```
Lw   R,0(X); adDu X,X,sX ## R:=X[0]
Lwc1 F,0(X); adDu X,X,sX ## F.Lo:=X[1]
mthc1 R,F  ;             ## F.Hi:=X[0]
vmthfp F,Vi;             ## Vi.Hi:=X[0]|X[1]
Lw   R,0(X); adDu X,X,sX ## R:=X[2]
Lwc1 F,0(X); adDu X,X,sX ## F.Lo:=X[3]
mthc1 R,F  ;             ## F.Hi:=X[2]
vmtlfp F,Vi;             ## Vi.Lo:=X[2]|X[3]
```

Аналогичная серия из 24-х команд потребуется и вместо одной команды сохранения (vsdq).

Значит, при кратной обработке произвольного вектора группами по 32 элемента на 8 вычислительных команд v**M**ul.s приходится уже 96 команд для загрузки и 96 команд для сохранения элементов. В результате доля самих вычислительных команд по отношению к командам, обеспечивающим подкачку данных, стремительно падает. Их соотношение для произвольного вектора теперь **8:192** (хотя для непрерывного вектора оно сохраняется прежним **8:9**)**.** Причём из этих 200 команд 128 одного и того же потока работы с памятью, и потому на их исполнение уйдёт не менее 128 тактов. Тогда на обработку SR-вектора длиной **n**=32·p потребуется как минимум **4·n** (128·n/32) тактов.

При реализации же функции SSCAL на CP1 обработать группу из 8 SR-элементов можно за 40 команд: (8×**M**ul.s+8×**L**wc1+8×**S**wc1+16×ad**D**u), которые в оптимальном варианте можно исполнить за 20 тактов. Так что обработка вектора длиной **n**=8·p элементов займёт всего **2,5·n** (20·n/8) тактов. Вот почему реализация функции SSCAL(n,α,X,ix) на CPV для произвольного вектора (ix≠1) оказывается невыгодной (4n>2,5n).

3.3 CSCAL и ZSCAL

Векторный сопроцессор CPV обеспечивает солидное преимущество (над CP1) при реализации функций обработки векторов комплексных величин, поскольку предоставляет обширный набор команд комплексной арифметики. Ведь каждая из них сразу исполняет операцию, которую на сопроцессоре CP1 приходится реализовывать серией из нескольких команд вещественной арифметики. Например:

CPV	эквивалентная реализация на CP1	
cmul.d	= 2×mul.d + madd.d + msub.d	4
cmul.s	= (2×mul.s + madd.s + msub.s) ×2	8
cmadd.d	= 3×madd.d + msub.d	4
cmadd.s	= (3×madd.s + msub.s) ×2	8

Чтобы получить более эффективную реализацию на CPV функций CSCAL и ZSCAL, можно применить все рассмотренные ранее приёмы оптимизации.

Полученные оценки производительности представим в виде итоговой таблицы (см. таблицу 8).

Таблица 8. Оценки производительности функции SCAL

ix≠1	Load Xi	Xi=a·Xi	Store Xi	M:L+S+D=T/k[1)]	O(**n**)[2)]
1×DC	v**L**dm+**D**	c**M**ul.d	v**S**dm+**D**	1:1+1+2=3	**3·n**
2×SC	2×**L**+2×**D**	c**M**ul.s	2×**S**+2×**D**	1:2+2+4=36/16	**2,25·n**
2×DR	2×**L**+2×**D**	v**M**ul.d	2×**S**+2×**D**	1:2+2+4=36/16	**2,25·n**
4×SR	8×**L**+4×**D**	v**M**ul.s	8×**S**+4×**D**	1:8+8+8=128/32	**4·n**
ix=1	v**L**dq	2×**M**	v**S**dq	8:4+4+1=9/8t[3)]	**≈ n/t**

[1)] M:L+S+D=T/k – форма записи, выражающая отношение числа вычислительных команд (M) к командам загрузки (L), сохранения (S) и приращения адреса (D), а также затраченное на них количество тактов (T) при условии обработки вектора группами по k элементов;
[2)] оценка общего кол-ва тактов для вектора из **n** элементов;
[3)] **t** – кол-во чисел в регистре: **t** =1,2,2,4 для DC,SC,DR,SR;

Благодаря отмеченному преимуществу CPV всегда будет обеспечивать весьма значительный выигрыш в производительности для любых функций обработки векторов и матриц комплексных чисел. Поэтому дальнейший анализ применимости CPV сосредоточим в большей степени на функциях обработки вещественных векторов.

3.4 AXPY

Рассмотрим теперь, как изменится оценка применимости CPV в отношении функции, которой требуется использовать не один, а два вектора. Возьмём для примера функцию сложения двух векторов AXPY. Основной цикл функции **x**AXPY (n,α,X,ix,Y,iy):

```
for (i=0,IY=0,IX=0; i<n; i++)
{ Y[IY]=a*Y[IY]+X[IX]; IY+=iy; IX+=ix; }
```

требует теперь в 2 раза больше команд загрузки:

Load Xi;	Load Yi;	Yi=a·Yi+Xi;	Store Yi

Вследствие чего изменяется баланс вычислительных команд (ВК) и команд работы с памятью (КП). А поскольку доля ВК становится меньше, то и выигрыш от использования CPV уменьшается (см. табл.9).

Таблица 9. Оценки производительности функции AXPY

ix≠1	Load Xi,Yi	Yi=a·Yi+Xi	Store Yi	M:L+S+D=T/k	O(**n**)
1×DC	(v**L**dm+**D**)×2	c**M**add.d	v**S**dm+**D**	1:2+1+3=4	**4·n**
2×SC	(2×**L**+2×**D**)×2	c**M**add.s	2×**S**+2×**D**	1:4+2+6=7	**3,5·n**
2×DR	(2×**L**+2×**D**)×2	v**M**add.d	2×**S**+2×**D**	1:4+2+6=7	**3,5·n**
4×SR	(8×**L**+4×**D**)×2	v**M**add.s	8×**S**+4×**D**	1:16+8+12=24	**6·n**
ix=1	v**L**dq×2	2×**M**	v**S**dq	8:8+4+1=12/8t	**3n/2t**

3.5 DOT

Применение CPV эффективно лишь при обработке векторов большими группами элементов. При этом количество (**k**) элементов в обрабатываемой группе следует выбирать таким, чтобы основную вычислительную команду можно было бы выполнять на каждом такте без каких-либо задержек. Если такая команда длится **h** тактов, а **k** взять меньшим (**k<h**), то будут возникать задержки в начале каждого цикла обработки очередной группы (на h-k тактов). Поскольку **k** определяет кратность длины (**n**=**k**·p) вектора, подлежащего обработке, то для упрощения вычислений (индексов и адресов элементов) значение **k** обычно выбирают равным ближайшей степени двойки (**k**= 8, 16, 32).

Разбивка векторов на группы может быть произвольной, если элементы векторов можно обрабатывать в любом порядке независимо друг от друга, как это, например, допускалось функциями SCAL и AXPY. Но такое возможно не всегда.

Некоторые функции могут потребовать учитывать зависимости между элементами или даже навязывать строго определённый порядок обработки элементов. В таком случае групповая обработка вектора может оказаться затруднительной или вообще невозможной.

В ряде случаев, чтобы добиться эффективной реализации функции на CPV, имеющиеся зависимости между элементами надо лишь разумно учитывать для получения удачного разбиения вектора на группы. Продемонстрируем это на примере функции вычисления скалярного произведения двух векторов **x**DOT.

Если в основном цикле **x**DOT (n,X,ix,Y,iy) :

```
for (i=0,S=0,IY=0,IX=0; i<n; i++)
{ S=S+Y[IY]*X[IX]; IY+=iy; IX+=ix; }
```

использовать команду умножения с накоплением результата в одном и том же регистре (c**M**add.d S,Xi,Yi), то следующую такую же команду можно будет исполнять не сразу, а лишь после завершения предыдущей.

Чтобы избежать возможных задержек и добиться эффективной реализации **x**DOT на CPV, будем обрабатывать за каждый шаг цикла такую группу элементов, каждая пара (Xj,Yj) которых накапливает своё произведение в отдельном регистре Sj (c**M**add.d Sj,Xj,Yj). А итоговую сумму S посчитаем после завершения основного цикла, просуммировав накопленные в регистрах Sj (j=1,..,k) произведения.

Заметим, что при обработке элементов типа SC или DR в 2-х частях (старшей и младшей) каждого регистра Sj будет накапливаться отдельная сумма. А при обработке SR-элементов таких отдельно накапливаемых сумм будет уже 4. Впоследствии, конечно, потребуется слить все эти суммы в одну. Для этого в CPV предусмотрена особая команда (VSUM), позволяющая суммировать числа, содержащиеся в одном регистре.

Для функции DOT характерен такой же баланс ВК:КП, как и для функции SCAL. И поскольку при обработке векторов большого размера (т.е. с возрастанием **n**) доля итоговых команд суммирования сокращается, то производительность функции DOT будет приближаться к той, что характерна для функции SCAL.

4. BLAS-функции 2-го уровня

Функции этого уровня предполагают такой вид обработки, в котором наряду с векторами обязательно используется и одна матрица. Квадратичный порядок $O(n^2)$ обработки её элементов и определяет, главным образом, сложность реализации самой функции.

Анализ функций 1-го уровня уже показал, что применение CPV далеко не во всех случаях позволяет поднять их производительность. Чтобы понять, для каких функций 2-го уровня (и в каких случаях) можно улучшить реализацию, если применить CPV, а для каких это сделать не удастся и почему, рассмотрим два типичных примера.

4.1 TRSV

Начнём с функции **x**TRSV (ul,tz,d,n,Z,lz,Y,iY). И рассмотрим самый простейший случай её применения, когда требуется получить вектор X[n], являющийся результатом решения системы линейных уравнений:

$$
\begin{aligned}
&Z_{11}*X_1 && = Y_1\\
&Z_{21}*X_1+Z_{22}*X_2 && = Y_2\\
&Z_{31}*X_1+Z_{32}*X_2+\ Z_{33}*X_3 && = Y_3\\
&\qquad\qquad\qquad\ \ \ldots\\
&Z_{n1}*X_1+Z_{n2}*X_2+\ Z_{n3}*X_3+\ \ldots\ Z_{nn}*X_n && = Y_n
\end{aligned}
$$

При вызове функции предполагается, что коэффициенты Z_{ij} заданы нижней треугольной матрицей Z[n][n]. А результат (вектор X) требуется записать на место вектора Y[n], первоначально задающего правую часть.

Алгоритм получения этого результата основывается на так называемом обратном ходе метода Гаусса, который предполагает, что элементы вектора должны вычисляться строго

последовательно друг за другом, так как перед вычислением очередного элемента X_i необходимо получить все предыдущие X_j (j=1,..,i-1):

$$X_i = (Y_i - \sum(Z_{ij} \cdot X_j)_{j=1,..,i-1}) / Z_{ii} \quad (\text{для } i=1,..,n)$$

Это обстоятельство не позволяет применить групповую обработку при вычислении элементов вектора X, а значит, применение CPV в этом случае вряд ли поможет обеспечить повышение производительности.

4.2 GEMV

Основная нагрузка при реализации функции GEMV (tz,m,n,α,Z,lz,X,ix,β,Y,iy) приходится на осуществление операции умножения матрицы $Z_{m \times n}$ на вектор X_n. Такое умножение, как известно, можно реализовать путём вычислений отдельных скалярных произведений каждой строки матрицы на вектор X, т.е. многократным применением функции DOT. Но при такой реализации выигрыш от применения CPV не превысит тот, что позволяет функция DOT.

Более того, в самом неблагоприятном случае, когда в операции умножения должна использоваться транспонированная матрица Z, придётся применять функцию DOT для произвольных векторов (столбцов матрицы и вектора X). И тогда вместо ожидаемого выигрыша в производительности можно даже будет получить и проигрыш (причина подобного проигрыша была подробно проанализирована в п. 3.2).

Однако, CPV допускает и другой способ реализации функции GEMV, предполагающий использование специальных команд **mvmadd** (см. табл.3). Каждая команда mvmadd.d позволяет умножить матрицу 2×2 на вектор из 2-х элементов, позволяя заменить 2 команды vmadd.d, используемые для накопления скалярных произведений (в функции DDOT). Значит, используя её, можно рассчитывать на подъём производительности функции DGEMV.

Поясним схематично алгоритм эффективной реализации DGEMV с применением этой команды для матрицы Z[m][n] и векторов X[n], Y[m] кратных размеров m=16·p, n=2·q (см. таблицу 10).

Таблица 10. Схема алгоритма DGEMV

```
for (r=0; r<m; r+=16) {
  for j=0..7 { load VYj ← ( β·Y[r+2j], β·Y[r+2j+1] ) }
  for (s=0; s<n; s+=2) {
    load VX ← { α·X[s], α·X[s+1] }
    for j=0..7 {
      load VAj ← { Z[r+2j][s], Z[r+2j][s+1] }
      load VBj ← { Z[r+2j+1][s], Z[r+2j+1][s+1] }
      mvmadd.d VYj,VAj,VX
    }
  }
  for j=0..7 { store VYj → ( Y[r+2j], Y[r+2j+1] ) }
}
```

```
Используются регистры CPV: VY0, …,VY7, VX
и соседние пары регистров: (VA0,VB0),…,(VA7,VB7)
```

Вектор Y вычисляем группами по 16 элементов. В начале обработки каждой группы загружаем очередные 16 элементов вектора Y в 8 регистров CPV VYj (j=0,1,..,7) и сразу умножаем их на скаляр β. Далее выполняем умножение матричного блока из следующих 16-ти строк матрицы Z на вектор X. Это умножение (с накоплением в регистрах VYj) выполняем поэтапно.

На каждом этапе сначала загружаем очередные два элемента вектора X в регистр VX и тут же домножаем их на скаляр α. Теперь из каждой j-ой пары строк матрицы очередные 2 элемента загружаем в соседние регистры ZAj, ZBj, образуя в них матрицу 2×2, которую и умножаем на регистр VX с добавлением к регистру VYj. Завершая обработку группы, сохраняем

из регистров VYj (j=0,1,..,7) новые значения вычисленных 16-ти элементов вектора Y.

Во внутреннем цикле этого алгоритма на 16 команд загрузки теперь приходится 8 команд mvmadd.d вместо 16 команд vmadd.d, т.е. улучшился баланс ВК:КП= 8M:16L+16D (вместо 16M:16L+16D). Это может дать выигрыш в тактах (20:24) исполнения внутреннего цикла и обеспечить в итоге общий подъём производительности примерно на 20% (24/20).

Хотелось бы, конечно, за счёт применения команд mvmadd получить выигрыш в 2 раза. Возможно ли это? Помогут ли тут команды быстрой загрузки vldq, vldqx? Увы, нет. Всё дело в том, что эти команды позволяют загрузить 4 элемента одной строки матрицы в соседние регистры, а для применения команды mvmadd в этом алгоритме требуется, чтобы в соседние регистры попадали 2 элемента одной строки и 2 элемента следующей строки. Поэтому загрузку элементов матрицы приходится выполнять командами vldmx. Это-то и не позволяет получить такой желательный баланс ВК:КП, чтобы доля ВК в нём оказывалась определяющей.

5. BLAS-функции 3-го уровня

Для функций 3-го уровня характерно значительное превышение доли вычислительных операций над операциями обмена с памятью: $O(n^3)$ против $O(n^2)$. В силу этого они становятся своеобразным полигоном, т.е. той привлекательной сферой применения, где CPV может проявить свои лучшие качества.

5.1 GEMM

Центральное место в этой области применения, конечно же, занимает функция **x**GEMM, которая обеспечивает операцию универсального матричного перемножения с накоплением:

$$Z_{m\times n}=\alpha\cdot X_{m\times k}\times Y_{k\times n}+\beta\cdot Z_{m\times n}\ .$$

В общем случае вызов этой функции сопровождается обширным списком параметров:

xGEMM (tx,ty, m,n,k, α,X,lx, Y,ly, β,Z,lz)

которые позволяют задавать каждую из перемножаемых матриц (X,Y) не только в обычном, но и в транспонированном виде (см. tx,ty) или сопряжённом (для комплексных), а также допускают, что в этой операции могут участвовать не только матрицы целиком, но и их отдельные прямоугольные части, называемые матричными блоками. В таком случае дополнительными параметрами (lx,ly,lz) уточняются размеры строк, с помощью которых были объявлены матрицы в программе: X[][lx], Y[][ly], Z[][lz] и которые определяют, с какими интервалами располагаются строки обрабатываемых матриц в памяти.

В случае обычных матриц операцию GEMM можно выразить таким широко известным классическим алгоритмом:

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    S=0; for (t=0; t<k; t++) S=S+X[i][t]*Y[t]*[j];
    Z[i][j]= β*Z[i][j]+ α*S;
}}
```

Однако, такая реализации операции GEMM оказывается неэффективной для обработки матриц больших размеров на современных машинах, обладающих быстродействующей кэш-памятью.

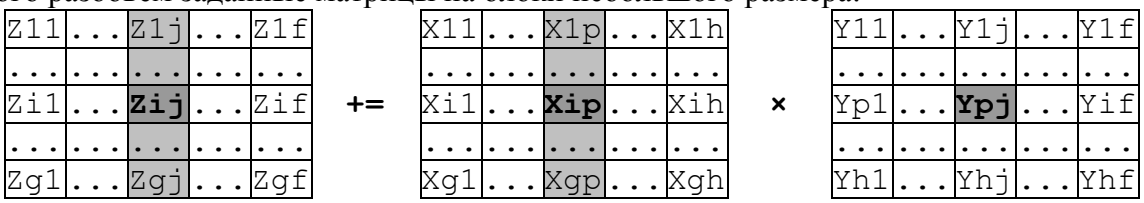
Поэтому на практике применяют другие варианты реализации GEMM [6], которые учитывают иерархическую структуру памяти и позволяют существенно сократить количество вынужденных операций обмена между медленной и быстрой памятью, обязательно возникающих при обработке больших матриц.

Для эффективной реализации GEMM на CPV был создан оригинальный алгоритм. В нём не только применялись известные приёмы поэтапного исполнения операции матричного перемножения с расщеплением обрабатываемых матриц на части: матричные панели (полосы) и блоки. Но и учитывались важнейшие особенности самого CPV.

В первую очередь, был сделан упор на максимальное использование уже опробованной

нами высокопроизводительной команды **mvmadd**. Кроме того, для сокращения общего числа загрузок элементов матриц в регистры CPV, было принято решение использовать часть регистров CPV в качестве своеобразной надстройки (L0-кэша 0-го уровня) над имеющейся кэш-памятью, чтобы хранить в них наиболее часто используемые блоки перемножаемых матриц.

Продемонстрируем этот алгоритм на примере реализации функции DGEMV для обычных (не транспонированных) матриц Z[m][n], X[n][k], Y[k][m] кратных размеров: m=4·g, n=8·f, k=8·h. Для этого разобьём заданные матрицы на блоки небольшого размера:



$Z=\{Zij_{4\times8}\}$, $X=\{Xip_{4\times8}\}$, $Y=\{Ypj_{8\times8}\}$, i=1..g; j=1..f; p=1..h.

И выразим основной алгоритм DGEMM (см. таблицу 11) в виде 3-х вложенных циклов, многократно выполняющих операции матричного умножения с накоплением над малыми блоками (**$Zij_{4\times8}$**, **$Xip_{4\times8}$**, **$Ypj_{8\times8}$**), из которых складываются исходные матрицы. Размеры этих блоков подбираются такими, чтобы при исполнении внутреннего цикла (по i) все участвующие в нём блоки могли уместиться в кэш-памяти.

Таблица 11. Общая схема алгоритма DGEMM

Требуется получить: $Z_{m\times n}=\alpha\cdot X_{m\times k}\times Y_{k\times n}+\beta\cdot Z_{m\times n}$

$Z_{m\times n}$= β·$Z_{m\times n}$; {умножение всех элементов матрицы на **β** }
for p= 1..h {
for j= 1..f {
for i= 1..g { **$Zij_{4\times8}$= $Zij_{4\times8}$ + α·$Xip_{4\times8}$×$Ypj_{8\times8}$**; } /*!A!*/
}}

Заметим также, что для умножения матрицы **Z** на скаляр β (в самом начале DGEMM), можно применять оптимизированный на CPV аналог функции SCAL.

Эффективность исполнения внутреннего цикла (по i), в конечном счёте, является определяющим звеном общей производительности функции DGEMM. Поэтому данному участку алгоритма (выделенному с пометкой !A!) уделим самое пристальное внимание. Рассмотрим детально, как его можно оптимально реализовать на CPV (см. таблицу 12).

Таблица 12. Схема умножения полосы на матричный блок

Требуется: for i= 1..g { **$Zij_{4\times8}$= $Zij_{4\times8}$ + α·$Xip_{4\times8}$×$Ypj_{8\times8}$**; }
for s=0..7 for q=0,2,4,6 **{** load **VYqs** ← (**α·Ypj**[q][s], **α Ypj**[q+1][s]) **}** **for i= 1..g {** **for r=0..3 {** **for s=0,2,4,6 {** load **VZrs** ← (Zij[r][s],Zij[r][s+1]) **}** **for q=0,2,4,6 {** load **VXrq** ← (Xip[r][q],Xip[r][q+1]) **}** **}** **for q=0,2,4,6 {** **for r=0..3 {** **for s=0,2,4,6 mvmadd.d VZrs,VYqs,VXrq** **}}** **for r=0..3 for s=0,2,4,6** **{** store **VZrs** → (Zij[r][s],Zij[r][s+1]) **}** **}// for g**
распределение регистров CPV по парам: **(VYq0,VYq1),…,(VYq6,VYq7), q=0,2,4,6** (для mvmadd.d) **(VXr0,VXr2),(VXr4,VXr6), r= 0…3** (для vldqx) **(VZr0,VZr2),(VZr4,VZr6), r= 0…3** (для vldqx)

Поскольку в этом цикле повсеместно используется матричный блок **Ypj**, то целесообразно перед таким циклом загрузить весь блок **Ypj** в регистры CPV, сразу умножив все его элементы на скаляр **α** и разместив их так, чтобы в дальнейшем было удобно сразу применять команду **mvmadd.d** к прочитанной в регистр паре соседних элементов очередной строки матрицы **X**.

Разместим элементы блока **Ypj** в 32-х регистрах CPV **VYqs** (q=0,2,4,6; s=0..7) так, чтобы s-ый столбец блока оказался в регистрах (**VY0s**,**VY2s**,**VY4s**,**VY6s**), и назначим регистрам такие номера, чтобы пары соседних регистров (VYq0,VYq1),…, (VYq6,**VYq7**) для q=0,2,4,6 могли представлять матрицы 2×2 в команде **mvmadd**.

На каждом i-ом шаге цикла загружаем 4 строки блока **Zij** в 16 регистров CPV **VZrs** (r=0..3, s=0,2,4,6), а 4 строки блока **Xip** – в 16 регистров **VXrq** (r=0..3, q=0,2,4,6). Для загрузки 8 элементов каждой строки в 4 регистра потребуется 2 команды v**L**dqx и одна команда (ad**D**u) для приращения индексного регистра. Итого 2×4×(2L+1D)=16L+8D команд.

Далее исполняем основной вычислительный блок, состоящий из 3-х вложенных циклов. В нём к каждому регистру **VZrs** (r=0..3,s=0,2,4,6) требуется добавить 2-х элементный вектор, являющийся результатом операции умножения r-ой строки блока **Xip** на s-ую пару столбцов блока **Yjp**. Для исполнения этой операции потребуется поэтапно над регистрами, представляющими отдельные части r-ой строки и s-ой пары столбцов, выполнить команду: mv**M**add.d **VZrs**,**VYqs**,**VXrq** 4 раза для q=0,2,4,6. Но такая команда исполняется 9 тактов. Поэтому для предотвращения ненужных задержек будем исполнять один этап такой операции умножения сразу для всей группы строк (r=0..3) и всех 4-х пар столбцов (s=0,2,4,6). Это займёт 16M команд на этап и 64M на весь вычислительный блок каждого i-го шага внутреннего цикла.

Наконец, вычисленные таким способом новые значения регистров **VZrs** (r=0..3, s=0,2,4,6) в конце i-го шага цикла сохраняем в памяти как обновлённый блок **Zij** матрицы **Z**. На это потребуется 8 команд v**S**dqx и 4 команды ad**D**u, т.е. 8S+4D команд.

Итак, на каждом шаге внутреннего цикла (по i) выполняется операция умножения (с накоплением) для матричных блоков:

$$\mathbf{Zij_{4\times8} = Zij_{4\times8} + \alpha \cdot Xip_{4\times8} \times Ypj_{8\times8}}.$$

На это затрачивается 64M+16L+8S+12D=100 команд. Причём доля самих команд mv**M**add почти в 2 раза превышает долю команд, обеспечивающих подкачку данных, т.е. получается хороший баланс ВК:КП= 64:36=16:9.

Можно перестроить внутренний цикл по i на две фазы (см. приём №2) и совместить вычислительные команды над одной группой строк (r=0,1) блоков Zij и Xip с командами загрузки и сохранения для другой группы строк (r=2,3) этих блоков. Тогда исполнение каждого шага внутреннего цикла можно будет ужать до 70 тактов. (После 8-ми команд mv**M**add каждого из 4-х этапов потребуется 1 такт задержки, да ещё 2 такта для команд организации самого цикла).

В результате представленный алгоритм, исполняя функцию DGEMM для матриц большого размера (но помещающихся целиком в кэше), будет обеспечивать высокую производительность, которую можно оценить как 91% (=64/70) от пиковой. Вот почему именно на функции GEMM можно получить максимальный выигрыш от применения CPV.

Доля ВК в рассмотренном алгоритме DGEMM столь высока, что позволяет сохранить благоприятный баланс ВК:КП даже в случае обработки транспонированных матриц. Подобный алгоритм обеспечивает достаточно высокую эффективность также и для функции SGEMM, т.е. в случае его применения для обработки элементов типа SR.

5.2 TRSM

Ещё одной часто используемой BLAS-функцией 3-го уровня является TRSM(s,ul,tz,d,m,n,α,Z,lz,Y,ly). Она предназначена для решения левостороннего (**Z×X=α·Y**) или

правостороннего (**X×Z=α·Y**) матричного уравнения для всевозможных вариантов представления треугольной матрицы Z. Установкой параметров можно задать не только, с какой стороны (s) в этом уравнении участвует матрица Z, но и уточнить, является она нижней или верхней треугольной (ul), транспонированная ли она (tz) или сопряжённая (для комплексных), единичная ли у неё диагональ (d) или нет. Функция может применяться не только к матрицам целиком, но и к отдельным матричным блокам, поэтому при её вызове задаются также параметры (lz,ly), уточняющие размеры строк используемых матриц.

Функцию TRSM можно считать расширенным вариантом функции TRSV, ибо она решает систему линейных уравнений (с одной и той же матрицей коэффициентов) уже не для одного, а сразу для нескольких векторов неизвестных, собранных в единую матрицу в виде столбцов (для **Z×X=α·Y**) или строк (для **X×Z=α·Y**). Значит, при реализации функции TRSM можно применять описанный ранее (в 4.1) метод последовательного получения элементов искомого вектора сразу ко всей группе неизвестных векторов, объединённых в матрицу. Именно применение групповой обработки открывает возможность повысить производительность функции TRSM с помощью CPV.

Проиллюстрируем сказанное в виде схемы алгоритма функции DTRSM (см. таблицу 13) для левостороннего варианта с нижней треугольной матрицей Z[m][m] для кратного **n** (n=32·f). В нём искомая матрица (X) получается на месте заданной матрицы Y[m][n], а выигрыш от применения CPV обеспечивается путём вызовов реализованных на CPV функций DSCAL и DAXPY для обработки непрерывных векторов кратного размера. (Напомним, что в таком случае применение CPV позволяет повысить производительность этих функций почти в 4 раза).

Таблица 13. Схема алгоритма функции DTRSM

```
Y[m×n]= α·Y[m×n] ; {умножение всех элементов матрицы на α }
for (k=0; k<m; k++) {
  DSCAL(n,1.0/Z[k][k],&Y[k][0],1); // Y[k] ← Y[k]/Z[kk]
  for (i=k+1; i<m; i++)
   if (Z[i][k] != 0.0) // Y[i] ← Y[i] – Y[k]·Z[ik]
    DAXPY(n,-Z[i][k],&Y[k][0],1,&Y[i][0],1);
}// for k
```

К сожалению, аналогичным алгоритмом для правостороннего варианта подобный выигрыш обеспечить не удаётся, т.к. в нём функции DSCAL и DAXPY придётся применять уже не к строкам, а к столбцам матрицы Y, т.е. к произвольным векторам. А в этом случае существенного выигрыша в производительности этих функций CPV, увы, не позволяет обеспечить.

5.3 TRSM на основе GEMM

Функция GEMM является универсальной в том смысле, что многие другие функции обработки матриц можно выразить через неё. И если с помощью CPV удаётся добиться её высокой производительности, то тогда открывается возможность повысить и производительность этих других функций тоже.

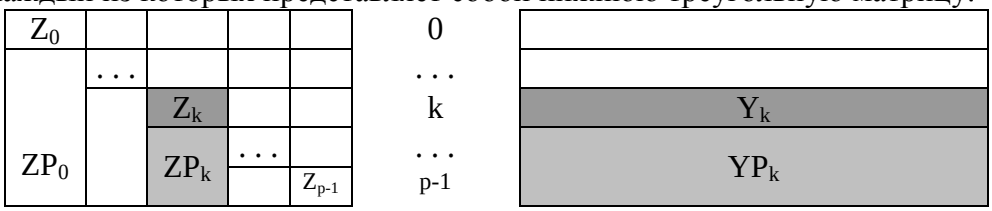
Таблица 14. Схема алгоритма DTRSM на основе DGEMM (m=w·p, n=32·f)

```
Y[m×n]= α·Y[m×n] ; {умножение всех элементов матрицы на α }
for (k=0;k<p; k++) { Ik= k·w; Lk= m-(Ik+w);
  //решаем матричное уравнение: Z[k]×X[k]=Y[k] и X[k]→ Y[k]
   DTRSM(s,ul,tz,d,w,n,1.0,&Z[Ik][Ik],lZ,&Y[Ik][0],ly);
  // перемножаем с вычитанием YP[k]← YP[k] – ZP[k] × Y[k]
   if (Lk>0) DGEMM(tz,ty,Lk,n,w,-1.0,
             &Z[Ik+w][Ik],lz,&Y[Ik][0],ly,1.0,&Y[Ik+w][0],ly);
}// for k
```

В качестве примера рассмотрим, как можно обеспечить более эффективную реализацию функции DTRSM с помощью применения оптимизированной на CPV функции DGEMM. Продемонстрируем схематично алгоритм (см. таблицу 14) такой реализации для того же

левостороннего варианта с нижней треугольной матрицей $Z_{m\times m}$ при кратных **m** и **n**.

Представим себе, что матрица Z состоит из p вертикальных панелей (полос) шириной по w элементов, а её диагональ выстроена как лестница из квадратных блоков размером по w элементов, каждый из которых представляет собой нижнюю треугольную матрицу:



Аналогично и матрицу Y представим состоящей из p горизонтальных полос размером по w элементов.

Тогда решение исходного матричного уравнения $\mathbf{Z\times X=\alpha\cdot Y}$ можно получить за p этапов. На каждом k-ом (k=0,1,..,p-1) этапе сначала требуется решить матричное уравнение $\mathbf{Z_k\times X_k=Y_k}$ для k-го малого диагонального блока $\mathbf{Z_k}$ и k-ой полосы $\mathbf{Y_k}$. А затем следует подкорректировать значения элементов всех оставшихся ещё не обработанных полос матрицы Y (обозначенных как блок полос $\mathbf{YP_k}$) путём вычитания из них результата матричного перемножения только что вычисленной в качестве решения полосы $\mathbf{X_k}$ на матричной блок коэффициентов $\mathbf{ZP_k}$, лежащий в k-ой панели матрицы Z ниже диагонального блока $\mathbf{Z_k}$:

$$\mathbf{YP_k \leftarrow YP_k - ZP_k\times X_k} \quad .$$

Заметим, что в этом алгоритме применяется прежняя функция DTRSM (из п.5.2), чтобы получить решение матричного уравнения для блоков малых размеров. А б**о**льшая часть вычислительной работы выпадает на высокопроизводительную функцию DGEMM, за счёт которой и удаётся поднять итоговую производительность функции DTRSM новой реализации.

6. Результаты тестов BLAS-функций

В представленных ниже таблицах (см. таблицы 15-17) содержатся показатели эффективности применения векторного сопроцессора (CPV) для некоторых функций обработки векторов и матриц библиотеки BLAS. Они получены экспериментальным путём в результате серии прогонов (на RTL-модели) тестов этих функций.

В полях этих таблиц содержится коэффициент выигрыша, который показывает, во сколько раз функция исполняется быстрее при её реализации на ассемблере с использованием команд CPV. Быстрее означает, что она выполняется за меньшее количество тактов, чем в случае реализации той же функции обычной (специально не оптимизированной) Си-программой, использующей сопроцессор вещественной арифметики (C_CP1).

Таблица 15. Выигрыш от применения CPV для AXPY

	n	**DC**	**SC**	**DR**	**SR**
AXPY ix=1	32	**4.03**	**10.15**	**3.17**	**4.18**
	256	**8.85**	**16.47**	**7.63**	**10.12**
	1024	**10.41**	**18.14**	**8.78**	**13.45**
AXPY ix≠1	32	**3.82**	**4.08**	**1.61**	**0.91**
	256	**2.83**	**3.05**	**1.51**	**0.93**
	1024	**2.87**	**2.48**	**1.45**	**0.97**

Таблица 16. Выигрыш от применения CPV для GEMV

	m×n	**DC**	**SC**	**DR**	**SR**
GEMV	32×32	**12.16**	**17.90**	**5.25**	**4.21**
	64×64	**13.03**	**21.99**	**3.87**	**3.82**
	256×256	**1.73**	**17.08**	**2.01**	**3.54**

Таблица 17. Выигрыш от применения CPV для GEMM

	m×n×k	DC	SC	DR	SR
GEMM	16×16×16	**9.28**	**13.69**	**6.42**	**8.70**
	32×32×32	**11.88**	**22.70**	**13.75**	**21.00**
	64×64×64	**14.67**	**31.77**	**34.06**	**30.14**
	128×128×128			**37.24**	**60.50**

В основном, эти показатели практически подтверждают приведённые ранее теоретические оценки.

Существенно, что представленные коэффициенты выигрыша от применения CPV, как правило, значительно превышают аналогичные коэффициенты выигрыша, которые обеспечивают те же функции библиотеки GotoBLAS2, содержащей ассемблерное ядро, оптимизированное для сопроцессора СР1 архитектуры КОМДИВ. (Исключением является функция DAXPY, производительность которой не удаётся поднять с помощью CPV в случае шага ix≠1). Это подтверждают приведённые здесь таблицы 18-20.

Таблица 18. Сравнение DAXPY CPV с DAXPY GotoBLAS2

	n	C_CP1	Goto-BLAS2	CPV	C_CP1 / CPV	C_CP1 / Goto	Goto / CPV
ix=1	32	364	180	115	**3.17**	**2.02**	**1.57**
	256	2715	1234	356	**7.63**	**2.20**	**3.47**
	1024	11886	5942	1353	**8.78**	**2.00**	**4.39**
ix≠1	32	443	256	275	**1.61**	**1.73**	**0.93**
	256	3183	2124	2105	**1.51**	**1.50**	**1.01**
	1024	16503	9386	11404	**1.45**	**1.76**	**0.82**

Таблица 19. Сравнение DGEMV CPV с DGEMV GotoBLAS2

m×n	C_CP1	Goto-BLAS2	CPV	C_CP1 / CPV	C_CP1 / Goto	Goto / CPV
32×32	6913	5357	1318	**5.25**	**1.29**	**4.06**
64×64	29172	16322	7529	**3.87**	**1.79**	**2.17**
256×256	467921	336960	233040	**2.01**	**1.39**	**1.45**

Таблица 20. Сравнение DGEMM CPV с DGEMM GotoBLAS2

m×n×k	C_CP1	Goto-BLAS2	CPV	C_CP1 / CPV	C_CP1 / Goto	Goto /CPV
16×16×16	30367	11600	4732	**6.42**	**2.62**	**2.45**
32×32×32	220438	63800	16029	**13.75**	**3.46**	**3.98**
64×64×64	3282795	454000	96379	**34.06**	**7.23**	**4.71**
128×...×128	26023211	3576700	698745	**37.24**	**7.28**	**5.12**

В них для каждой из функций над вещественными числами двойной точности (DR): DAXPY, DGEMV и DGEMM сравниваются показатели производительности трёх её различных реализаций: 1) обычной (неоптимизированной) Си-функции (C_CP1); 2) функции библиотеки GotoBLAS2 (версии 1.13) с оптимизированным ассемблерным ядром; 3) ассемблерной функции, применяющей команды CPV.

Слева в колонках таблиц приводятся полученные показатели числа тактов, затраченных на исполнение функции, а справа – вычисленные на их основе коэффициенты выигрыша (по отношению к C_CP1) для CPV и GotoBLAS2. Наконец, в самой правой колонке содержится показатель, характеризующий, во сколько раз реализованная с помощью CPV функция превышает по производительности аналогичную оптимизированную функцию GotoBLAS.

Заключение: выводы и предложения

Проведённый анализ и полученный практический опыт разработки ряда BLAS-функций для имеющегося варианта векторного сопроцессора (CPV) позволяют сделать следующие выводы о возможности его применения для повышения производительности функций обработки векторов и матриц.

1. CPV позволяет значительно ускорить обработку непрерывных векторов и почти не даёт никакой выгоды для обработки одиночных произвольных векторов.

2. При обработке векторов комплексных величин CPV обеспечивает более существенный выигрыш, чем при обработке вещественных.

3. Выигрыш от применения CPV возможен только при обработке векторов и матриц кратных размеров.

4. Вектора и матрицы, обрабатываемые с применением CPV, должны располагаться в памяти выровненными по кратному адресу.

5. CPV позволяет получить более значительный выигрыш при обработке матриц (в Си-программе), если такую обработку удаётся совершать по строкам, а не по столбцам.

Таким образом, применение CPV позволяет значительно повысить производительность для многих BLAS-функций. Но, к сожалению, не для всех. И не во всех случаях их применения.

Самой болезненной нерешённой проблемой, сдерживающей эффективное применение CPV, остаётся проблема обеспечения своевременной подкачки данных из памяти в регистры CPV (и обратно) при обработке произвольных векторов. Особенно остро эта проблема встаёт для векторов из 32-битных элементов.

Она могла бы окончательно разрешиться, если бы в CPV были добавлены команды ускоренной групповой загрузки (и сохранения) элементов произвольного вектора (`X`) в парные регистры (`V0`,`V1`), аналогичные тем командам (**`vLdq`**, **`vSdq`**), что CPV обеспечивает для подкачки элементов непрерывного вектора. Например, это могли бы быть команды вида:

`vLdqI` `V0,I(X)`	`V0←{X[0], X[I],X[2·I],X[3·I]}` `V1←{X[4·I], X[5·I],X[6·I],X[7·I]}`
`vSdqI` `V0,I(X)`	`V0→{X[0], X[I],X[2·I],X[3·I]}` `V1→{X[4·I], X[5·I],X[6·I],X[7·I]}`

Возможно, было бы полезно осуществить такое совмещение некоторых CPV-регистров с парами регистров сопроцессора CP1:

`V0=(F0,F1), V1=(F2,F3),... V15=(F30,F31).`

Это позволило бы для работы с регистрами CPV напрямую использовать команды CP1 и наоборот. Тогда, например, для загрузки 4-х SR-элементов произвольного вектора в один регистр CPV потребовалось бы уже не 12 команд, а только 10.

Число команд, затрачиваемых на загрузку (и сохранение) можно было бы сократить, если бы в командах CPV обеспечивалась автоинкрементная адресация (например, как это сделано в сопроцессоре CP2 микропроцессора К128РИО). Тогда связка из двух команд могла бы превратиться в одну усложнённую команду:

`L``wc1 F,0(X); ad`**`D`**`u X,X,sX`	**`Lwc1`** `F,0(X)+sX`

В таком случае вместо тех же 12 команд стало бы 8 (или даже 6).

Могут ли эти предложения быть реально реализованы? Очевидно, что на этот вопрос способны ответить только разработчики аппаратуры. Несомненно лишь то, что в любом случае решение обозначенной проблемы невозможно без какого-то существенного усовершенствования самого CPV и/или связанной с ним процессорной архитектуры.

Литература

1. Бетелин В.Б. Отечественные суперкомпьютерные технологии экзафлопсного класса – необходимое условие обеспечения технологической конкурентноспособности России в XXI веке. // Программные продукты и системы. 2013. №4 (104). С.4-9.

2. URL: http://ru.wikipedia.org/wiki/Basic_Linear_ Algebra_Subprograms (дата обращения: 02.10.2014)

3. URL: http://www.cs.utexas.edu/users/flame/goto/ signup_first.html (дата обращения: 02.10.2014).

4. Гервич Л., Штейнберг Б.Я., Юрушкин М. Программирование экзафлопсных систем. // Открытые системы. 2013. №8 (194). С.26-29.

5. Gregory Genry. Оптимизация функций линейной алгебры библиотеки Intel® MKL под Intel® AVX на примере DGEMM. URL: https://software.intel.com/ru-ru/articles/ (дата обращения: 02.10.2014)

6. Kazushige Goto, Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. // ACM Trans. on Mathematical Software, vol. 34, No. 3, 2008, pp.1-25.