УДК 681.3

Э.И. Ватутин, С.Ю. Мирошниченко, В.С. Титов

Курский государственный технический университет

ПРОГРАММНАЯ ОПТИМИЗАЦИЯ ОПЕРАТОРА СОБЕЛА С ИСПОЛЬЗОВАНИЕМ SIMD-РАСШИРЕНИЙ ПРОЦЕССОРОВ СЕМЕЙСТВА x86

В работе рассматривается задача программной оптимизации оператора Собела с использованием SIMD-расширений (MMX, SSE, SSE2) современных процессоров семейства x86. Подробно освещена организация вычислений с использованием MMX- и SSE-расширений. Приведены результаты оптимизации на разных процессорах с использованием различных вычислительных блоков.

В настоящее время широкое распространение получили системы технического зрения (СТЗ), работающие в реальном масштабе времени. Основным требованием, предъявляемым системам реального времени, является необходимость потоковой обработки видеоинформации, что накладывает жесткие временные ограничения на алгоритмы функционирования таких систем. Следовательно, актуальна задача оптимизации алгоритмов обработки изображений с использованием всех возможностей платформы, на которой реализована конкретная СТЗ. В современных СТЗ широко применяется семейство операторов пространственного дифференцирования, используемых для формирования контурных изображений $g(x, y)$ [1]. Наиболее распространенным оператором пространственного дифференцирования является оператор Собела, так как он обладает наилучшей реакцией на ступенчатый перепад и имеет наименьший коэффициент утолщения контурной линии. Задача формирования

градиентного изображения имеет высокую временную сложность, так как требует свертки исходного изображения $f(x,y)$ с парой масок H_1, H_2 по следующим правилам

$$g(x,y) = \left\|\nabla f(x,y)\right\| = \sqrt{d_1^2 + \ d_2^2}\ ,$$

$$d_i = f(x,y)\cdot H_i = \sum_{j=1}^{3}\sum_{k=1}^{3} f(x+j-2,\ \ y+k-2)\cdot h_{jk}^{i},\ i=1,2\ ,$$

$$H_1 = H_x = \begin{bmatrix} h_{11}^1 & h_{12}^1 & h_{13}^1 \\ h_{21}^1 & h_{22}^1 & h_{23}^1 \\ h_{31}^1 & h_{32}^1 & h_{33}^1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix},\ H_2 = H_y = \begin{bmatrix} h_{11}^2 & h_{12}^2 & h_{13}^2 \\ h_{21}^2 & h_{22}^2 & h_{23}^2 \\ h_{31}^2 & h_{32}^2 & h_{33}^2 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}.$$

Рассматриваемая задача имеет высокую однородность данных и характеризуется отсутствием зависимостей по данным между различными наборами исходных данных, что позволяет организовать эффективную параллельную свертку изображения с масками и последующее вычисление результирующего значения оператора Собела с использованием SIMD расширений.

Исходные данные представляют собой массив байт – значений яркости пикселей (в случае полутонового изображения), либо значений яркостей компонент красного, зеленого и синего (в случае цветного изображения). Для цветных изображений классическое расположение данных в памяти попиксельно (R1 G1 B1 R2 G2 B2 … – формат Windows Bitmap) очень неудобно с точки зрения команд SIMD-расширений (AoS в терминологии [2]), т.к. потребует значительного количества вспомогательных команд (пересылки, распаковки) для группировки соседних соцветных значений яркостей в регистрах (в виде R1 R2 R3 …), что выразится в дополнительных вычислительных затратах, которых вполне можно избежать путем реорганизации расположения исходных данных. Такая реорганизация проводится до обработки изображения оператором Собела и заключается в хранении значений яркостей цветовых компонент в виде цветовых плоскостей R1 R2 … Rn G1 G2 … Gn B1 B2 … Bn (SoA в терминологии [2]). Начальный адрес массива с данными выравнивается на 16, что позволяет снизить количество

одновременных обращений к различным кэш-линиям и снизить нагрузку на шину FSB (Front Side Bus, шина процессор–память).

Для удобства дальнейшего изложения обозначим фрагмент изображения, требующий свертки с масками, как

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}.$$

Под A, B, …, I подразумеваются или отдельные значения яркости пикселей, или вектора значений размерности 8 или 16 в зависимости от используемого набора команд.

Преобразуем исходную математическую модель с целью уменьшения количества вычислений:

$$S_1 = A - I,$$
$$S_2 = C - G,$$
$$H_x = 2(D - F) + S_1 - S_2,$$
$$H_y = 2(B - H) + S_1 + S_2,$$
$$g = \left\lfloor N\sqrt{H_x^2 + H_y^2} \right\rfloor, N = \frac{256}{1140}.$$

Вычисление значений S_1, S_2, H_x, H_y можно проводить как операции над целыми числами; возведение H_i в квадрат, сложение квадратов и вычисление квадратного корня эффективнее производить как операции над числами с плавающей запятой. Анализ состава требуемых операций указывает на возможность проведения вычислений с использованием следующих связок блоков процессора:

- INT+FPU (процессоры до Pentium MMX без поддержки технологии MMX);
- MMX+FPU (процессоры Pentium II и Athlon без поддержки расширения SSE);
- MMX+SSE (процессоры Pentium III и Athlon XP без поддержки расширения SSE2);
- SSE+SSE2 (процессоры Pentium IV и Athlon 64).

Нами были разработаны и протестированы все перечисленные способы реализации.

Рассмотрим более детально способ вычисления с использованием связки MMX + SSE как наиболее распространенной среди настольных процессоров на данный момент.

Прежде всего, необходимо вычисление значений H_x и H_y с использованием MMX-регистров. Результат вычисления находится в диапазоне [–1020, 1020], что требует его хранения в виде слов (2 байта), в то время как исходные данные хранятся как байты. SIMD-расширения не имеют команд, позволяющих производить параллельные арифметические операции над операндами различной разрядности одновременно, поэтому при загрузке исходных данных в регистры необходимо расширение байт до слов. Графическое представление процедуры загрузки данных приведено на рис. 1.

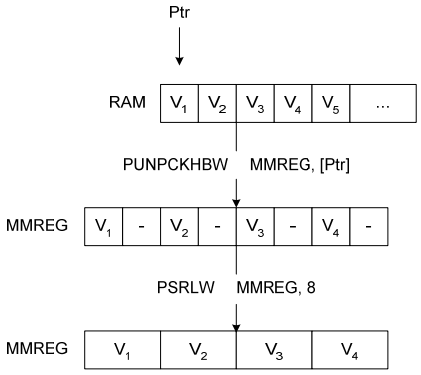

Рис. 1. Выборка исходных значений из памяти с расширением байт до слов

После загрузки некоторых значений требуется их умножение на два (2B, 2F). Эту операцию можно совместить с операцией расширения байт до слов путем замены сдвига с 8 позиций на 7. Однако необходимо обратить внимание, что в этом случае содержимое регистра MMREG перед операцией распаковки должно быть нулевым, т.к. в противном случае возможно появление единиц в младших разрядах слов после сдвига, что даст в результате значение 2V+1 вместо 2V.

Вычисление значений H_x и H_y не представляет большой сложности и производится как показано на рис. 2.

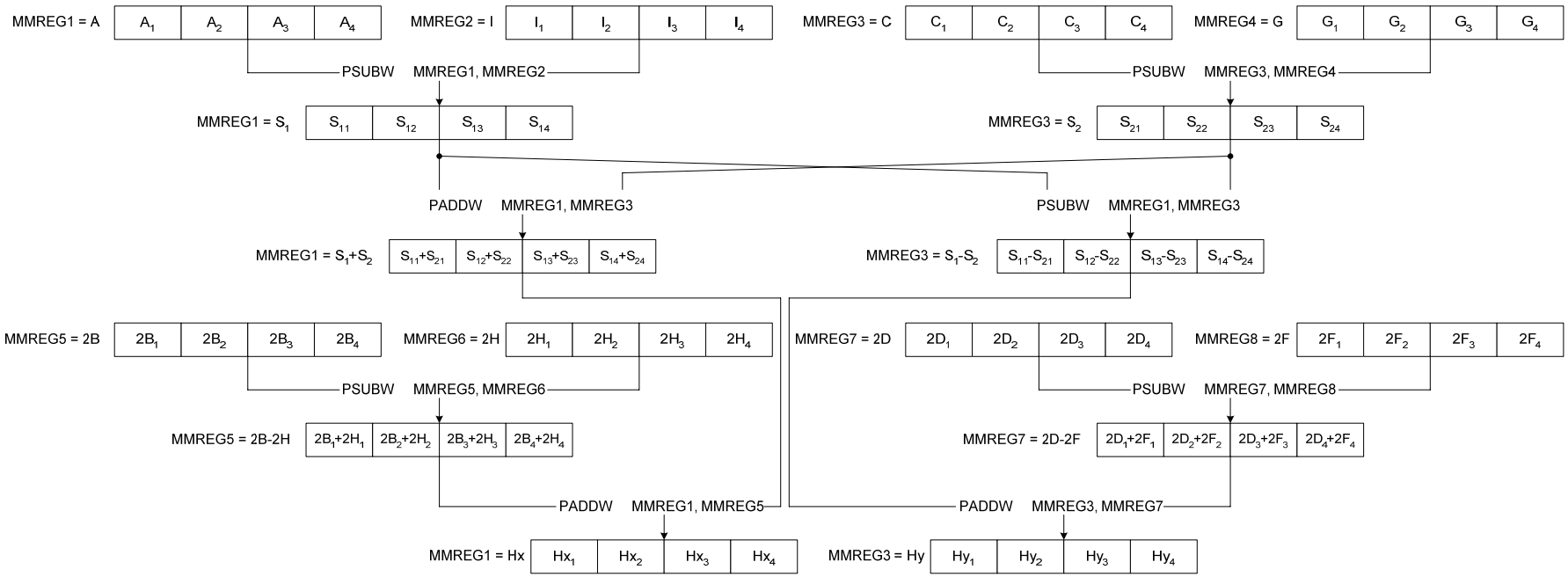

Рис. 2. Вычисление значений H_x и H_y

Дальнейшие вычисления рациональнее проводить как операции с плавающей точкой (возможно параллельное выполнение четырех вычислений), в то время как целочисленное умножение потребует в два раза большую размерность для хранения результата (параллельное выполнение только двух вычислений). Для этого необходимо выполнение преобразования целых чисел в вещественные одинарной точности (рис. 3).

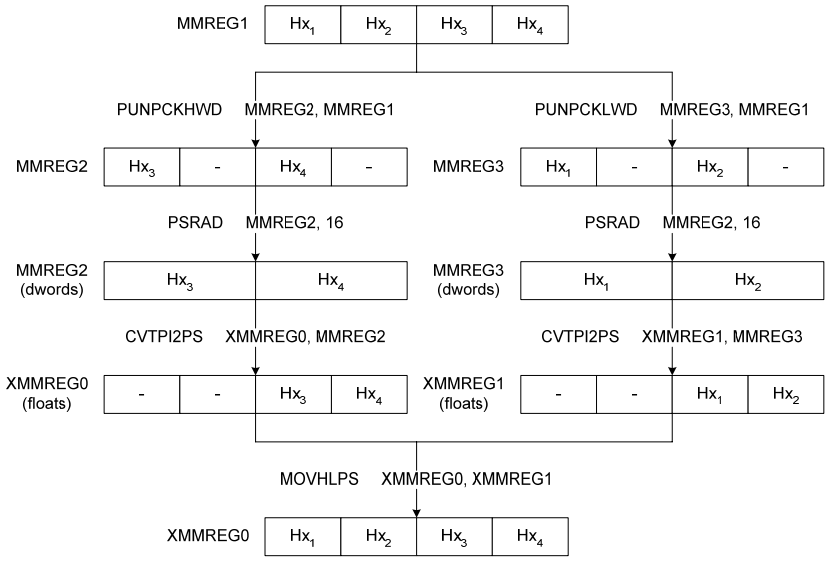

Рис. 3. Передача данных MMX→SSE

В данном случае необходимо преобразование слов в двойные слова, т.к. единственной векторной командой преобразования из целых чисел в вещественные является CVTPI2PS, а она принимает в качестве операндов только пару двойных слов. Следует отметить, что в данном случае необходимо использовать команды арифметических сдвигов (с учетом знака) PSRAD, а не PSRLD, т.к. старшая половина каждого двойного слова должна содержать знак, а не просто нули.

После получения значений H_x и H_y в XMM-регистрах возможно вычисление искомого значения оператора Собела g. Умножение на константу нормировки N гарантирует нахождение полученных значений в диапазоне [0, 255]. Процесс вычислений представлен на рис. 4.

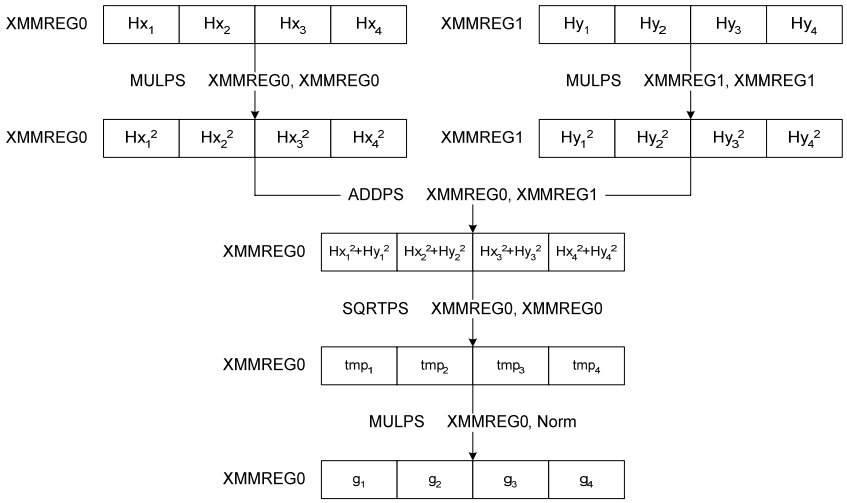

Рис. 4. Вычисление значения оператора Собела

Полученные значения необходимо округлить, преобразовать в байты и записать в память (рис. 5).

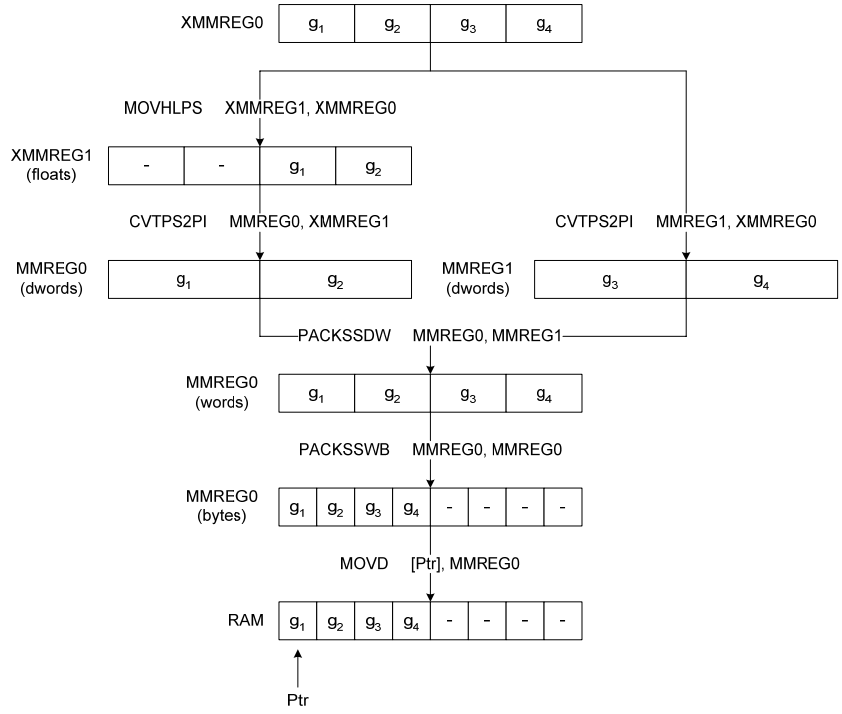

Рис. 5. Запись результата в память

Рассмотренные выше операции можно закодировать с использованием двух методик расположения ассемблерных команд: группировка однотипных команд над независимыми данными (способ 1) и группировка разнотипных команд над зависимыми данными (способ 2). Примеры кодирования приведены ниже.

Способ 1	Способ 2
PUNPCKHBW MM0, [EAX+ESI*4-4] PUNPCKHBW MM1, [EDX+ESI*4-2] PUNPCKHBW MM2, [EAX+ESI*4-2] PUNPCKHBW MM3, [EDX+ESI*4-4] PSRLW MM0, 8 PSRLW MM1, 8 PSRLW MM2, 8 PSRLW MM3, 8 PSUBW MM0, MM1 PSUBW MM2, MM3	PUNPCKHBW MM0, [EAX+ESI*4-4] PSRLW MM0, 8 PUNPCKHBW MM1, [EDX+ESI*4-2] PSRLW MM1, 8 PSUBW MM0, MM1 PUNPCKHBW MM2, [EAX+ESI*4-2] PSRLW MM2, 8 PUNPCKHBW MM3, [EDX+ESI*4-4] PSRLW MM3, 8 PSUBW MM2, MM3

Первый способ теоретически эффективнее, т.к. позволяет процессору параллельно исполнять (при отсутствии аппаратных ограничений) серию команд. Второй способ проще и понятнее в процессе кодирования.

Сравнение рассмотренных методик (на примере связок INT+FPU, MMX+SSE и SSE+SSE2) показывает, что способ кодирования 1 чуть более привлекателен (до 6% прибавки в скорости). Результаты рассмотрения связки MMX+FPU сильно зависят от архитектуры процессора:

1. На процессорах Pentium III и Pentium IV способ кодирования 1 более эффективен (до 25% прибавки в скорости для Pentium III и до 56% для Pentium IV).
2. Процессоры Athlon XP практически невосприимчивы к стилю кодирования (разница около 1% в пользу способа 2).

3. На процессорах Athlon 64 предпочтительнее использовать способ 2 (до 38% прибавки в скорости).

Столь различное поведение, по-видимому, объясняется особенностями внутренней организации блока вычислений с плавающей точкой (стековая организация регистров, набор исполнительных устройств) и его взаимодействие с MMX-расширением (необходимость использования команды EMMS) или асимметричной организацией декодеров.

В процессе обработки изображения производится последовательный обход области памяти, содержащей массив данных изображения, что позволяет использовать команды предвыборки данных из памяти в кэш (prefetching). Использование данных команд способно уменьшить задержки (stalls), связанные с ожиданием подкачки данных из памяти в кэш [2, 3].

Анализ с использованием профайлера VTune показывает, что наиболее часто промахи кэша (cache misses) связаны с обращениями к элементам нижней строки (*G*, *H*, *I*), т.к. эти элементы в общем случае используются впервые и могут не быть в кэше, в то время как остальные элементы (за исключением обработки первых двух строк изображения) находятся в кэше L2, объема которого в современных процессорах достаточно для хранения двух строк изображения. Для ликвидации промахов кэша при обработке нижней строки используется команда предвыборки данных третьей строки в кэш L1 (PREFETCHT0).

Данные для верхней и средней строк, как правило, находятся в большем по объему и более медленном кэше L2. До обращения к ним желательно их перемещение в более быстрый кэш L1, что также производится с использованием команд PREFETCHT0.

При организации предвыборок необходимо определиться с расстоянием, на которое производится предвыборка (Prefetch Scheduling Distance – PSD). В [2, 3] приводятся математические модели для расчета PSD, однако значение PSD зависит от

ряда в общем случае неизвестных параметров системы (lookup latency, number of clock to transfer cache-line, и т.д.), что затрудняет вычисления. После проведения серии сравнительных испытаний (следуя рекомендациям [2, 3]) было решено остановиться на значении +128.

Анализ эффективности предвыборок для различных платформ приведен ниже:

1. На процессорах Pentium III достигается до 17% прибавки в скорости.
2. На процессорах Pentium IV выигрыш отсутствует, что, по-видимому, объясняется функционированием механизма аппаратной предвыборки (hardware prefetch [4]).
3. На процессорах Athlon XP выигрыш отсутствует (как с использованием команд PREFETCHT0, так и специфичных для 3dNow-расширения PREFETCHW).
4. На процессорах Athlon 64 выигрыш также отсутствует за счет введения механизма аппаратной предвыборки [3].

Результаты сравнения скорости обработки на разных платформах приведены на рис. 6.

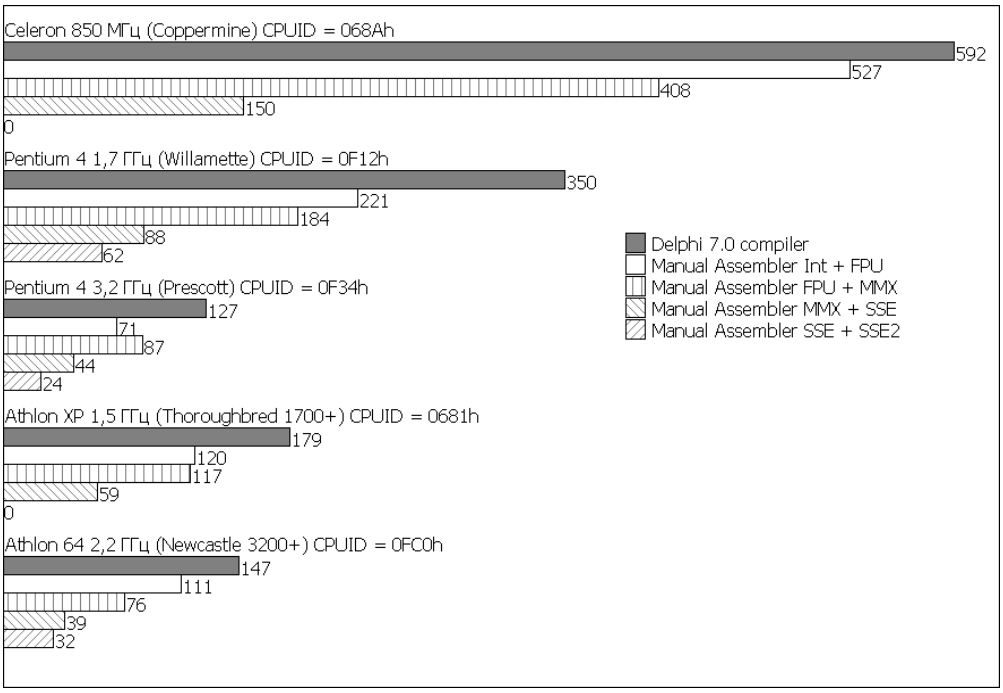

Рис. 6. Зависимость времени обработки изображения от типа процессора и используемого набора связок исполнительных устройств

Результаты оптимизации по сравнению с высокоуровневым вариантом (Delphi 7) для различных платформ приведены ниже на примере обработки цветного изображения размером 1280×1024 (3,75 Мб), зависимость времени обработки изображения от его размера в мегабайтах линейная. При обработке изображения в оттенках серого время обработки сокращается в 3 раза, т.к. производится обработка только одной цветовой плоскости.

Процессор	Неоптимизи-рованный Собел (мс)	Оптимизированный Собел – лучший результат (мс)	Выигрыш в скорости (раз)
Pentium III 850 МГц (Coppermine)	592	150	3,95
Pentium 4 1,7 ГГц (Willamette)	350	62	5,65
Pentium 4 3,2 ГГц (Prescott)	127	24	5,29
Athlon XP 1700+ (1,47 ГГц Thoroughbred)	179	59	3,03
Athlon 64 3200+ (2,2 ГГц Newcastle)	147	32	4,59

Выводы

1. Реализованы и протестированы подпрограммы, выполняющие вычисление оператора Собела с использованием всех разумно возможных вариантов исполнительных блоков процессора.
2. Выявлено, что применение различных способов оптимизации (векторизация, выравнивание данных, различное расположение команд, предвыборки данных)

приводит к различному эффекту, однако в общем случае не ухудшает качества оптимизации.

3. Проведение низкоуровневой оптимизации на ассемблере способно повысить скорость обработки изображения оператором Собела в 3–5,5 раза по сравнению с высокоуровневым кодом.

Библиографический список

1. Дегтярев С.В., Садыков С.С., Тевс С.С. Ширабакина Т.А. Методы цифровой обработки изображений. Ч. 1. / КурскГТУ. Курск, 2001. 167 с.
2. Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual. Order number 248966-05.
3. Software Optimization Guide for AMD Athlon 64 and AMD Opteron Processors. Publication number 25112.
4. IA-32 Intel Architecture Optimization Reference Manual. Order number 248966-011.