УДК 519.6

НЕКОТОРЫЕ ПОДХОДЫ К ЭФФЕКТИВНОЙ РЕАЛИЗАЦИИ БЛОЧНЫХ МАТРИЧНЫХ АЛГОРИТМОВ НА MIMD – КОМПЬЮТЕРАХ

И.А. Баранов

Институт кибернетики НАН Украины им. В.М. Глушкова,
03187, Киев, проспект Академика Глушкова, 40.
Тел.: 360 3045, e-mail: vlasov@ukr.net

Исследуется реальная производительность процессорного элемента в зависимости от различных способов расположения данных в кэш памяти на примере умножения двух матриц (рассматривается алгоритми Фокса, Кеннона и wbgemm).

In the given work real productivity of a processor element depending on various ways of an arrangement of data in a cache of memory on an example of multiplication of two matrixes is researching (Fox, Cannon and wbgemm algorithms is considered).

Введение

Стремительно нарастающая массовая потребность в высокопроизводительных вычислительных ресурсах, необходимых для решения разнообразных задач [1], приводит к широкому использованию открытых систем массового параллелизма, состоящих из стандартных компонентов, в том числе массовых серийных процессоров. Рост вычислительных возможностей современных высокопроизводительных вычислительных систем связан не только с переходом на высокочастотные элементы, но и во многом определяется интенсивным развитием средств параллельной работы в аппаратуре ЭВМ.

Увеличение производительности может достигаться с помощью самых разнообразных форм параллелизма. Любая вычислительная система представляет совокупность связанных функциональных устройств. В каждый момент времени эти устройства могут либо простаивать, либо выполнять полезную работу, т.е. заниматься хранением, передачей или обработкой информации. Относительное быстродействие вычислительной системы определяется составом используемых функциональных устройств и степенью их общей загруженности.

В соответствии с этим можно выделить тенденции повышения быстродействия вычислительных систем. Одна из них связана с использованием большого числа одновременно работающих функциональных устройств и приводит к проблеме распараллеливания вычислений, другая – с увеличением эффективности использования самих устройств. Повышение загруженности функциональных устройств связывают с конвейеризацией вычислений. Максимальная загрузка вычислительного конвейера достигается при наличии большого числа длинных ветвей независимых вычислений и при невысоких накладных расходах на передачу данных при реализации этих ветвей.

При разработке алгоритмов для решения крупномасштабных задач, требующих обработки большого количества данных, особое внимание уделяется архитектурным особенностям основной оперативной памяти вычислительной системы, пропускной способности шины передачи данных и характеристикам доступа к разным компонентам памяти. В этой ситуации анализ передачи данных между основной оперативной памятью и функциональными устройствами становится важной частью совместного исследования расчетного алгоритма и вычислительной системы.

Достижение высокой производительности вычислительных устройств за счет программных средств возможно при использовании компиляторов, обеспечивающих высокий уровень оптимизации. Другой подход состоит в применении базовых процедур, в высокой степени влияющих на производительность широкого круга приложений. Когда для таких процедур определен интерфейс и сформулирован соответствующий стандарт, их можно оптимизировать под различные архитектуры.

В данной работе исследуется реальная производительность процессорного элемента в зависимости от различных способов расположения данных в кэш памяти на примере умножения двух матриц.

Предварительные сведения

При анализе реальной производительности процессорного элемента необходимо рассматривать все возможные варианты взаимного расположения данных. Это обусловлено архитектурными особенностями вычислителя. Максимальные показатели производительности достигаются при расположении данных во внутрикристальном кэше L1. Также высокие показатели производительности наблюдаются и при расположении данных в кэше второго уровня L2. Размер кэша накладывает ограничения на длину векторов,

© И.А. Баранов, 2008

которые могут в нем целиком поместиться. Скорость обмена данными по внешней системной шине существенно ниже скорости обмена данными между процессором и кэш-памятью. Это приводит к низкой загрузке процессора в случае, когда данные располагаются в основной оперативной памяти.

Рассмотрим три уровня кэш-памяти.

Кэш-память первого уровня L1 – это блок высокоскоростной памяти, расположенный прямо на ядре процессора. В него копируются данные, извлеченные из оперативной памяти.

Объем кэш-памяти первого уровня L1 от 8 до 128 Kb.

Сохранение часто используемых данных позволяет повысить производительность процессора за счет более высокой скорости обработки данных (обработка из кэша быстрее, чем из оперативной памяти). Объем кэш-памяти первого уровня невелик и исчисляется килобайтами. Обычно "старшие" модели процессоров обладают большим объемом кэша L1.

В случае многоядерных процессоров указывается объем кэш-памяти первого уровня для одного ядра.

Кэш-память второго уровня L2 – это блок высокоскоростной памяти, выполняющий те же функции, что и кэш L1, однако имеющий более низкую, по сравнению с L1, скорость и больший объем. Объем кэш-памяти второго уровня L2 от 128 до 8192 Kb. Если вы выбираете процессор для ресурсоемких задач, то процессор с большим объемом кэша L2 будет предпочтительнее. Для многоядерных процессоров указывается суммарный объем кэш-памяти второго уровня.

Интегрированная **кэш-память L3** в сочетании с быстрой системной шиной формирует высокоскоростной канал обмена данными с основной памятью. Объем кэш-памяти третьего уровня кэша L3 от 0 до 16384 Kb. Как правило, кэш-памятью третьего уровня комплектуются только процессоры для серверных решений или специальные редакции "настольных" процессоров. Кэш-памятью третьего уровня обладают, например, такие линейки процессоров как: Intel Pentium 4 Extreme Edition, Xeon DP, Itanium 2, Xeon MP и прочие.

Вытеснение данных из кэша происходит постоянно (особенно из L1). Никакого приоритета в использовании кэша ни у ОС, ни у других потоков нет – процессор обязан загрузить данные по требованию текущего потока и он их грузит, выкидывая при этом наиболее старую из конкурирующих линеек (число конкурентов равно ассоциативности кэша), причем если она оказывается модифицированной, то ее приходится записывать обратно в общую память (write back).

При переключении потоков данные будут полностью или частично вытеснены из L1, но останутся в L2 (если конечно параллельно не запущено приложение также интенсивно работающее с памятью), поэтому при возврате управления потоку данные (в среднем) будут грузиться из L2 намного быстрее, чем из общей памяти.

Постановка задачи

При выполнении математических расчетов, с использованием, больших объемов данных, возникает проблема потери времени из-за лишних обращений к общей памяти. Задачей данной работы является нахождение более эффективных методов расположения данных в памяти для сокращения времени выполнения задачи за счет полноценного использования кэш памяти процессора, как первого уровня L1, так и второго L2.

Рассмотрим задачу вычисления произведения двух матриц

$$C_{ij} = \sum_k A_{ik} B_{kj} \,. \tag{1}$$

Одним из решений повышения производительности при решении задачи, является разбиение входных и выходных данных задачи на блоки размерностью позволяющей оперировать ими с ограниченным обращением к общей памяти.

Исследование задачи

Для проведения исследований был разработан комплекс программ на языке С. Реализовано таймирование различных вариантов взаимного расположения данных. Изучение особенностей взаимодействия между функциональными устройствами и основной оперативной памятью приводит к необходимости рассматривать различные способы взаимного расположения данных внутри процессорного элемента.

Для анализа реальной производительности рассматривалась задача вычисления произведения двух матриц (1) и исследовались случаи расположения данных целиком в кэш-памяти (здесь рассматривалось три алгоритма такого размещения), целиком в основной оперативной памяти и промежуточный вариант, когда часть данных располагается в кэш-памяти, а часть поступает по внешней системной шине из основной оперативной памяти.

Таймирование вычислений осуществлялось с помощью высокоточной процедуры MPI_WTIME из пакета MPICH-1.2.1 [2].

Далее приведены четыре алгоритма умножения двух матриц.

Алгоритм 1 (не блочное умножение матриц).

for i=1 to n
for j=1 to n
for k=1 to n
$$C_{ij} = C_{ij} + A_{ik} \cdot B_{kj}$$

Алгоритм 2 (алгоритм Фокса) [3, 4] умножения матриц при блочном разделении данных.

Для организации параллельных вычислений при блочном представлении матриц предположим, что процессоры образуют логическую прямоугольную решетку размерностью $k \times k$.

Обозначим p_{ij} процессор, располагаемый на пересечении *i* строки и *j* столбца решетки. Основные положения алгоритма Фокса состоят в следующем:

- каждый из процессоров решетки отвечает за вычисление одного блока матрицы *C*;
- в ходе вычислений на каждом из процессоров p_{ij} располагается четыре матричных блока:

– блок C_{ij} матрицы *C*, вычисляемый процессором;

– блок A_{ij} матрицы *A*, размещенный в процессоре перед началом вычислений;

– блоки $A_{ij}^{'}$, $B_{ij}^{'}$ матриц *A* и *B*, получаемые процессором в ходе выполнения вычислений.

Выполнение параллельного алгоритма включает:

- этап инициализации, на котором на каждый процессор p_{ij} передаются блоки A_{ij}, B_{ij} и обнуляются блоки C_{ij} на всех процессорах;
- этап вычислений, на каждой итерации *l*, $1 < l < k$, которого выполняется:

– для каждой строки *i*, $1 < i < k$, процессорной решетки блок A_{ij} процессора p_{ij} пересылается на все процессоры той же строки *i*; индекс *j*, определяющий положение процессора p_{ij} в строке, вычисляется по соотношению $j = (i + l - 1) \bmod k + 1$ (mod – операция получения остатка от целого деления);

– полученные в результате пересылок блоки $A_{ij}^{'}$, $B_{ij}^{'}$ каждого процессора p_{ij} имеют вид:

$$C_{ij} = C_{ij} + A_{ij}^{'} \times B_{ij}^{'} ;$$

– блоки $B_{ij}^{'}$ каждого процессора p_{ij} пересылаются процессорам p_{ij}, являющимися соседями сверху в столбцах процессорной решетки (блоки процессоров из первой строки решетки пересылаются процессорам последней строки решетки).

Для пояснения метода на рис. 1 показано состояние блоков в каждой подзадаче в ходе выполнения итераций этапа вычислений (для решетки подзадач 2×2).

В рассмотренной схеме параллельных вычислений количество блоков может варьироваться в зависимости от выбора размера блоков – эти размеры могут быть подобраны, таким образом, чтобы общее количество базовых подзадач совпадало с числом процессоров *p*. Так, например, в наиболее простом случае, когда число процессоров представимо в виде $q = \delta^2$ (т.е. является полным квадратом), можно выбрать количество блоков в матрицах по вертикали и горизонтали равным δ (т.е. $q = \delta$). Такой способ определения количества блоков приводит к тому, что объем вычислений в каждой подзадаче является одинаковым и тем самым достигается полная балансировка вычислительной нагрузки между процессорами. В более общем случае при произвольных количестве процессоров и размерах матриц балансировка вычислений может отличаться от абсолютно одинаковой, но, тем не менее, при надлежащем выборе параметров может быть распределена между процессорами равномерно в рамках требуемой точности.

Для эффективного выполнения алгоритма Фокса, в котором базовые подзадачи представлены в виде квадратной решетки и в ходе вычислений выполняются операции передачи блоков по строкам и столбцам решетки подзадач, наиболее адекватным решением является организация множества имеющихся процессоров также в виде квадратной решетки. В этом случае можно осуществить непосредственное отображение набора подзадач на множество процессоров – базовую подзадачу (i, j) следует располагать на процессоре p_{ij}. Необходимая структура сети передачи данных может быть обеспечена на физическом уровне, если топология вычислительной системы имеет вид решетки или полного графа.

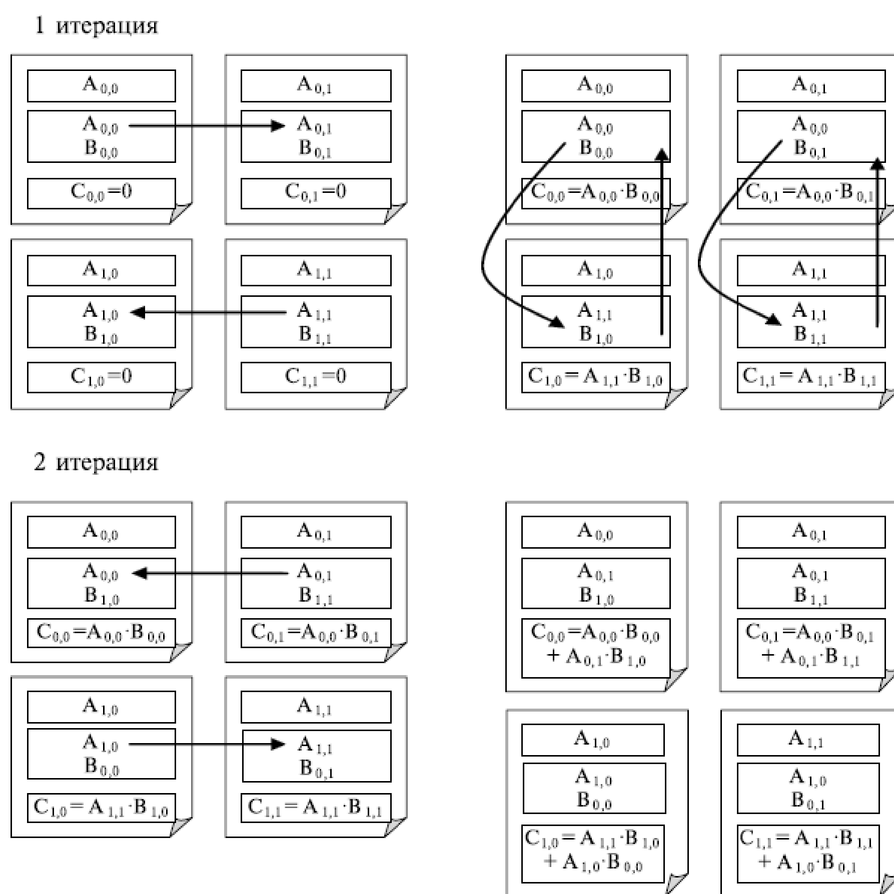


Рис. 1. Состояние блоков в каждой подзадаче в ходе выполнения итераций алгоритма Фокса

Алгортм 3 (алоритм Кэннона). Отличие алгоритма Кэннона от метода Фокса состоит в изменении схемы начального распределения блоков умножаемых матриц между процессорами вычислительной системы. Начальное расположение блоков в алгоритме Кэннона подбирается таким образом, чтобы располагаемые блоки в процессорах могли бы быть перемножены без каких-либо дополнительных передач данных между процессорами. При этом подобное распределение блоков может быть организовано таким образом, что перемещение блоков между процессорами в ходе вычислений может осуществляться с использованием более простых коммуникационных операций.

forall i=0:N-1
    Циклический сдвиг влево строки i по i, так, что $A^{(i,j)}$
    определяет $A^{(i,(j+i) \bmod N)}$
forall j=0:N-1
    Циклический сдвиг вверх столбца j по j, так, что $B^{(i,j)}$
    определяет $B^{((j+i) \bmod N, j)}$.
for k=1:N
    forall i=0:N-1, forall j=0:N-1
        $C^{(i,j)} = C^{(i,j)} + A^{(i,j)} \cdot B^{(i,j)}$
        Циклический сдвиг влево каждой строки матрицы A на 1, так, что $A^{(i,j)}$
        определяет $A^{(i,(j+1) \bmod N)}$
        Циклический сдвиг вверх каждого столбца матрицы B на 1, так, что $B^{(i,j)}$
        определяет $B^{((i+1) \bmod N, j)}$.

С учетом высказанных замечаний этап инициализации алгоритма Кэннона включает выполнение таких операций передач данных:

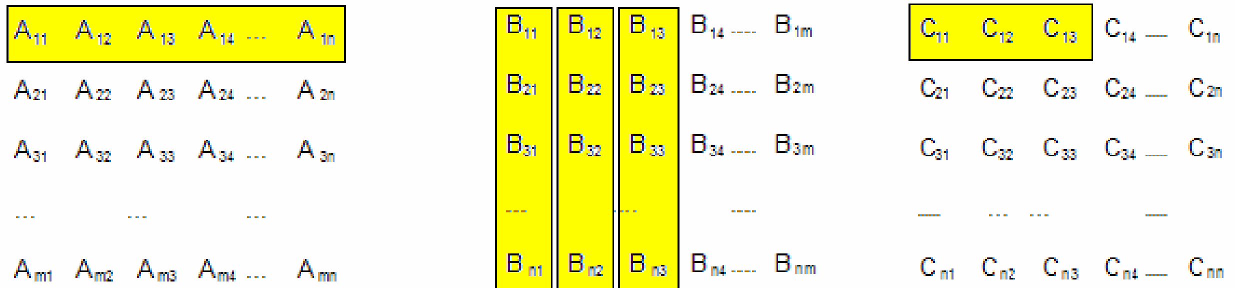
- в каждый процессор p_{ij} передаются блоки A_{ij}, B_{ij};
- для каждой строки i процессорной решетки блоки матрицы A сдвигаются на (i-1) позиций влево;
- для каждого столбца j процессорной решетки блоки матрицы B сдвигаются на (j-1) позиций вверх.

В ходе вычислений на каждой итерации алгоритма Кэннона каждый блок матрицы *A* сдвигается на один процессор влево по решетке, а каждый блок матрицы *B* – на один процессор вверх.

Алгоритм 4 (WBGEMM) прямоугольные блоки по строке и столбцам (рис.2).

1. Создаются указатели на две матрицы *A* и *B*, т.е. матрицы хранятся в общей памяти.

2. Для умножения матриц *A* и *B* в кэш-памяти, следует разбить на блоки. Разбиение на блоки осуществляется с помощью прохода по изначальным матрицам с помощью цикла и выборки нужной размерности блока. В результате создаются матрицы A_1 и B_1, которые и будут участвовать в умножении.



$$A_1 = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} & \dots & A_{1n} \end{pmatrix} \qquad B_1 = \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \\ \dots & \dots & \dots \\ B_{n1} & B_{n2} & B_{n3} \end{pmatrix} \qquad C_1 = \begin{pmatrix} C_{11} & C_{12} & C_{13} \end{pmatrix}$$

Рис. 2. Выбор размера блока

Размерность блока зависит от объема кэш-памяти процессора.

Процесс умножения матриц в кэш-памяти не подлежит контролю, но, что бы гарантировать, что умножение будет происходить в кэш-памяти, следует жестко определить зависимость размера кэш памяти от размера умножаемых блоков. Будем выбирать эффективный размер блока таким образом, чтобы число столбцов в блоке при блочном умножении равнялось или было кратно значению

$$N_{\min} = \text{размер линейки} \,/\, \text{размер элемента}.$$

Максимальный размер целесообразно ограничить так, чтобы в L1 умещалось три текущих блока: перемножаемые и блок результата (A_1, B_1, C_1). Причем при умножении (строка на столбец) многократно используется только блок второй матрицы и результата, а блок первой матрицы проходится последовательно по строкам, поэтому для него достаточно резервировать место в L1 только под одну строку. В итоге максимальный размер блока должен быть не больше $\frac{L_1 - m}{2}$, где *m* – размер строки блока.

3. После завершения процесса умножения блоков обрабатываемой матрицы, блоки собираются в результирующую матрицу C.

Анализ выбора способа разбиения на блоки

Опишем наиболее характерные черты поведения производительности по мере роста объемов обрабатываемых данных.

Каждый из алгоритмов запускался последовательно на 1, 2, 4, 7, 8, 12 17 32 ядрах 64 процессорного комплекса ИНАРКОМ – 64 с четырех ядерными процессора Intel Quad Core.

Наблюдается первоначальный быстрый рост производительности при малых длинах векторов. Это обусловлено уменьшением доли накладных расходов на организацию вычислений. Высокие показатели производительности наблюдаются при расположении данных в кэш-памяти.

Временные характеристики работы алгоритма 3 и 4 представлены в таблице и на рис. 3 – 7 (по вертикали масштаб логарифмический).

Таблица

Количество потоков	1024	2048	4096	8192	10000	16384	20000
1 thread Кэннон	3,08	26,19	115,85	923,56	1505,32	1969,36	
+ ASM	0,65	5,36	24,89	193,56	341,23	410,98	
Без блоков	13,45	87,54	378,56	2797,78			
1 thread WBGEMM	3,06	26,01	116,76	911,56	1590,05	1889,96	
2 thread Кэннон	1,38	13,06	57,35	494,4	760,38	1054,9	1201,3
+ ASM	0,175	1,65	13,06	85,25	112,23	171,36	216,57
Без блоков	6,75	42,87	178,21	1598,67	2321,48		
2 thread WBGEMM	1,35	12,99	56,97	496,05	762,13	1060,78	1195,04
4 thread Кэннон	0,79	7,25	36,4	306,4	440,56	604,85	653,45
+ ASM	0,101	1,023	7,98	60,87	95,63	139,64	159,96
Без блоков	2,56	22,68	109,89	978,41	1567,09	1907,67	
4 thread WBGEMM	0,74	7,04	35,78	301,56	431,7	594,56	639,93
7 thread Кэннон	0,34	3,66	18,11	183,38	272,52	339,89	462,78
+ ASM							
Без блоков	1,45	11,17	61,34	598,56	847,78	198,74	1479,34
7 thread WBGEMM	0,35	3,73	18,95	183,02	279,78	345,21	478,41
32 thread Кэннон	0,09	0,96	4,67	37,68	53,89	64,69	96,01
+ ASM	0,016		0,96		10,15		15,68
Без блоков	0,39	4,05	15,78	119,01	178,45	206,92	314,87
32 thread WBGEMM	0,089	0,94	4,63	37,01	52,51	62,03	101,09

Таким образом, видно, что при малых размерах матриц алгоритм 4 показывает производительность несущественно, но выше алгоритма 3 (Кенона). Однако при дальнейшем росте размеров матриц блочные алгоритмы 3 и 4 сильно опережают не блочный алгоритм. Такое поведение не зависит от количества вычислительных процессоров. Подобная зависимость может быть связана с тем, что не блочный алгоритм использует больше памяти и это является решающим фактором, замедляющим работу алгоритма при больших размерах матриц.

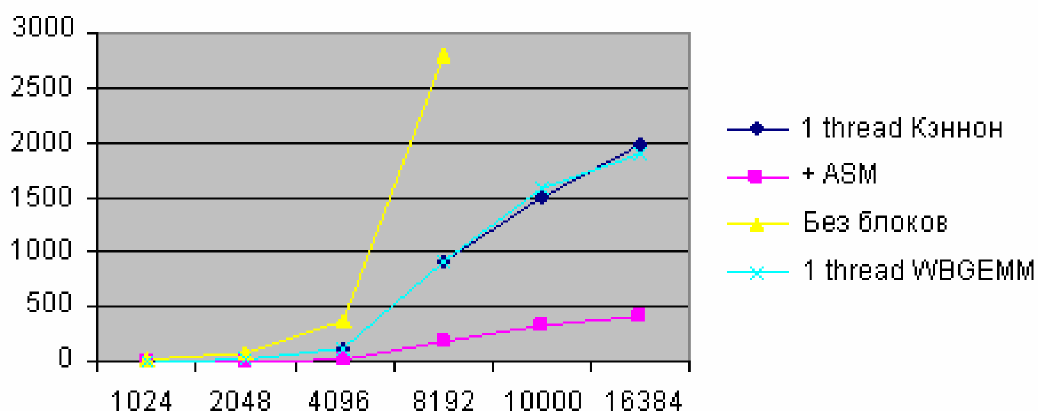


Рис. 3. Использование одного ядра

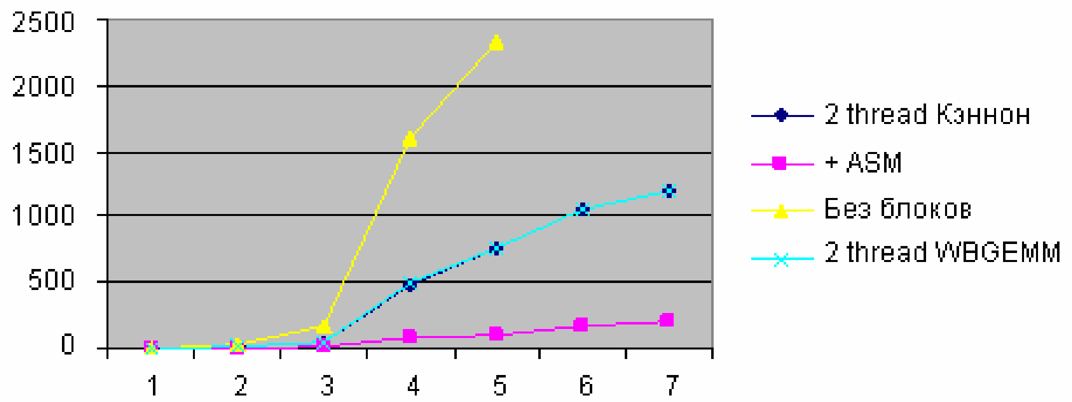


Рис. 4. Использование двух ядер

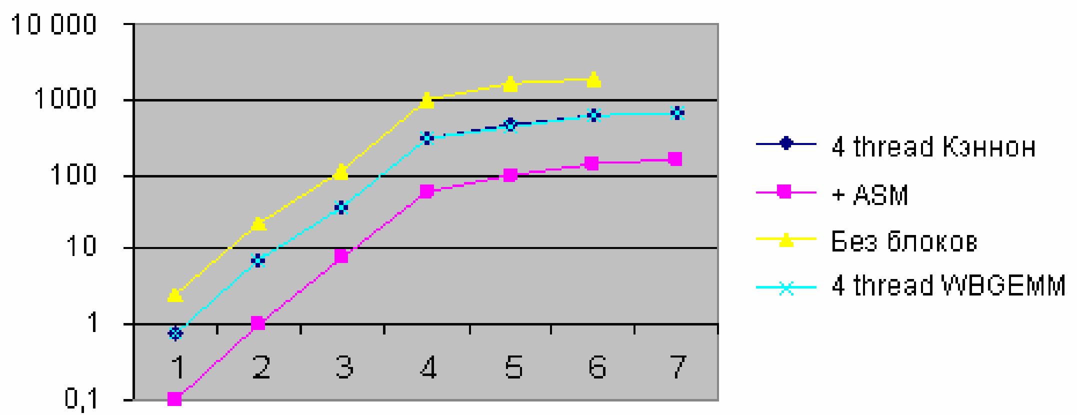


Рис. 5. Использование четырех ядер

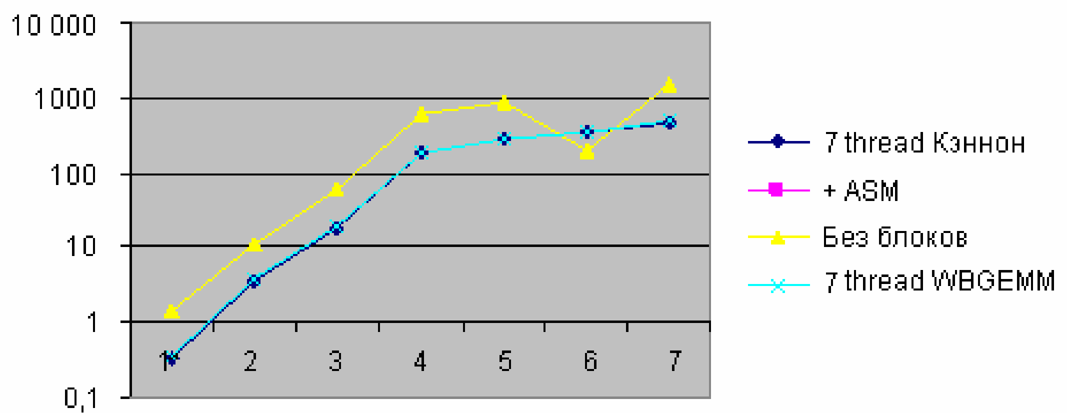


Рис. 6. Использование семи ядер

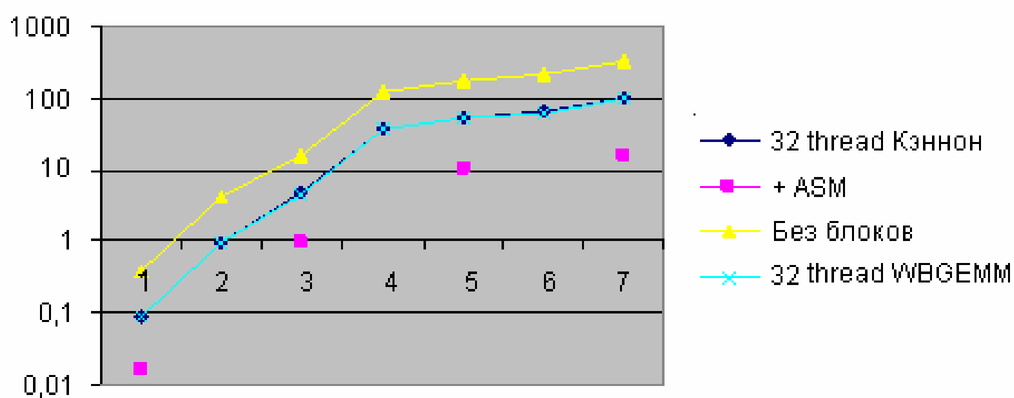


Рис. 7. Использование тридцати двух ядер

Алгоритмы 3 и 4 показывают практически одинаковые результаты производительности, что является закономерным, учитывая схожесть алгоритмов по сути – оба алгоритма являются блочными.

Выводы

Исследовались алгоритмы разбиения матриц на блоки, при котором максимально возможное количество данных, над которыми осуществляются действия, находящиеся в кэш-памяти. Этим и объясняется резкое увеличение производительности. Кроме блочного умножения в данном случае можно использовать транспонирование второй матрицы и перемножение с использованием векторной обработки на SSE при задействовании соответствующихопций оптимизации в компиляторах Intel и MS VS (см.примерные цифры без учета времени транспонирования).

1. *Воеводин В.В., Воеводин В.В.* Параллельные вычисления. – СПб.: БХВ-Петербург, 2002. – 608 с.
2. *Корнеев В.В.* Параллельное программирование в MPI М. – Ижевск: 2003. – 303 с.
3. *Fox G.C., Otto S.W.* and *Hey A.J.G.* Matrix Algorithms on a Hypercube I: Matrix Multiplication Parallel Computing. – 1987. – 4. – P. 17 – 31.
4. *Cannon L. E.*, A cellular computer to implement the Kalman Filter Algorithm, (Technical report, Ph.D. Thesis, Montana State University, 1969).