УДК 519.6
DOI: 10.26102/2310-6018/2024.45.2.041

Использование одновременной многопоточности в высокопроизводительных вычислительных алгоритмах

Е.А. Буевич✉

Московский государственный технологический университет «СТАНКИН», Москва, Российская Федерация

Резюме. Технология одновременной многопоточности считается малоприменимой в программах, занимающихся интенсивными вычислениями, в частности, при умножении матриц – одной из основных операций машинного обучения. Целью данной работы является определение границ применимости этого типа многопоточности к интенсивным вычислениям на примере блочного матричного умножения. В работе выделен ряд характеристик кода умножения матриц и архитектуры процессора, влияющих на эффективность использования одновременной многопоточности. Предложен способ определения наличия структурных ограничений процессора при исполнении более чем одного потока и их количественной оценки. Рассмотрено влияние используемого примитива синхронизации и его особенности применительно к одновременной многопоточности. Рассмотрен существующий алгоритм разделения матриц на блоки, предложено изменение размеров блоков и параметров циклов для лучшей утилизации вычислительных модулей ядра процессора двумя потоками. Создана модель оценки производительности выполнения идентичного кода двумя потоками на одном физическом ядре. Создан критерий определения возможности оптимизации кода с интенсивными вычислениями с помощью этого типа многопоточности. Показано, что разделение вычислений между логическими потоками с использованием общего кэша L1 оправдано как минимум на одной из распространенных архитектур процессоров.

Ключевые слова: одновременная многопоточность, умножение матриц, интенсивные вычисления, микроядро, BLAS, BLIS, синхронизация, иерархия кэша, спинлок.

Для цитирования: Буевич Е.А. Использование одновременной многопоточности в высокопроизводительных вычислительных алгоритмах. *Моделирование, оптимизация и информационные технологии.* 2024;12(2). URL: https://moitvivt.ru/ru/journal/pdf?id=1588 DOI: 10.26102/2310-6018/2024.45.2.041

Using simultaneous multithreading in high-performance numerical algorithms

E.A. Buevich✉

Moscow State Technological University "STANKIN", Moscow, the Russian Federation

Abstract. The technology of simultaneous multithreading is considered to be of little use in programs involved in intensive calculations, in particular when multiplying matrices - one of the main operations of machine learning. The purpose of this work is to determine the limits of applicability of this type of multithreading to high performance numerical code using the example of block matrix multiplication. The paper highlights a number of characteristics of matrix multiplication code and processor architecture that affect the efficiency of using simultaneous multithreading. A method is proposed for determining the presence of structural limitations of the processor when executing more than one thread and their quantitative estimation. The influence of the used synchronization primitive and its features in relation to simultaneous multithreading are considered. The existing algorithm for dividing matrices into blocks is considered, and it is proposed to change the size of blocks and loop parameters for better utilization of the computing modules of the processor core by two threads. A model has been created to evaluate

the performance of executing identical code by two threads on one physical core. A criteria has been created to determine whether computationally intensive code can be optimized using this type of multithreading. It is shown that dividing calculations between logical threads using a common L1 cache is beneficial in at least one of the common processor architectures.

Keywords: simultaneous multithreading, matrix multiplication, computation intensive, microcore, BLAS, BLIS, synchronization, cache hierarchy, spinlock.

For citation: Buevich E.A. Using simultaneous multithreading in high-performance numerical algorithms. *Modeling, Optimization and Information Technology*. 2024;12(2). URL: https://moitvivt.ru/ru/journal/pdf?id=1588 DOI: 10.26102/2310-6018/2024.45.2.041 (In Russ.).

Введение

Концепция одновременной многопоточности (simultaneous multithreading, SMT) заключается в одновременном исполнении вычислительным ядром процессора нескольких потоков инструкций для более полной утилизации его ресурсов [1]. На массовом рынке одновременная многопоточность стала доступна в 2002 году в серверных процессорах Intel Xeon и в том же году в процессоре для персональных компьютеров Intel Pentium 4. Как следствие, появились отдельные термины «физический процессор» – набор вычислительной логики на интегральной схеме и «логический процессор» – поток инструкций, выполняющийся на физическом ядре [2]. Далее используются более современные «физическое ядро» и «логическое ядро».

Ранние исследования показали, что применение одновременной многопоточности не всегда оправдано [3]. До настоящего времени достаточно общим местом является утверждение, что эта технология малоприменима в программах, занимающихся интенсивными вычислениями, в частности, при умножении матриц [4]. В пакете линейной алгебры OpenBLAS с открытым исходным кодом в вычислениях используется только одно логическое ядро на каждом физическом. Несколько примеров использования одновременной многопоточности в умножении матриц есть в работе [4] для архитектур, в которых в рамках одного потока невозможно эффективно использовать векторное умножение. В актуальном на текущий момент алгоритме [5] возможность разделения данных в кэше L1, характерная для данного типа многопоточности, не рассматривается.

По мере развития микропроцессорной техники соотношение различных характеристик процессоров меняется. В данной работе исследованы факторы, ограничивающие возможность использования одновременной многопоточности для матричного умножения, и предложены модификации алгоритма для уменьшения их влияния. В качестве архитектуры для экспериментов выбрана Intel Skylake, поскольку в ней появилось малозатратное средство синхронизации потоков на одном ядре. Эта архитектура относится к семейству x64 (или x86-64), включающему большое количество процессоров Intel, AMD и VIA.

Материалы и методы

Операция общего умножения матриц, описываемая формулой C: = αAB + βC, в библиотеках линейной алгебры называется GEMM. Современный подход к этой задаче берет свое начало от библиотеки GotoBLAS (сейчас называется OpenBLAS) с открытым исходным кодом, разработанной Kazushige Goto [6]. Основной идеей было использование ассемблерных «ядер», оптимизированных для конкретной архитектуры.

Современным стандартом высокопроизводительного умножения матриц сложности Θ(n3) считается модификация этого алгоритма, реализованная в библиотеке

BLIS [7]. Общая структура модифицированного алгоритма [8] для умножения матрицы A размером M x K на матрицу B размером K x N представлена на Рисунке 1.

```
цикл 1      for(n = 0; n < N; n+= n_step)
цикл 2         for(k = 0; k < K; k+= k_step)
                  B[k:k+k_step-1;n:n+n_step-1] => Bbuf
цикл 3            for(m = 0; m < M; m+= m_step)
                     A[m:m+m_step-1:k:k+k_step - 1] => Abuf
-------------------------------------------------------------------------
цикл 4               for(i = 0; i < n_step; i+= nr)          макроядро
цикл 5                 for(j = 0; j < m_step; j+= mr)
-------------------------------------------------------------------------
цикл 6                    for(ii = 0; ii < k_step; ii++)     микроядро
                             C[m+j:m+j+mr-1,n+i:n+i+nr-1]
                                +=Abuf[j:j+mr-1,ii]
                                 *Bbuf[ii,i:i+nr-1]
```

Рисунок 1 – Алгоритм умножения матриц BLIS
Figure 1 – BLIS matrix multiplication algoritm

Самым внутренним циклом является микроядро, написанное с учетом SIMD возможностей и характеристик регистрового файла конкретного процессора. Два цикла более высокого уровня называются макроядром. Остальные внешние циклы отвечают за разделение матриц на блоки, наиболее выгодные для повторного использования в более верхних уровнях кэширования (L2 и L3 кэши), заполняя временные буфера Abuf и Bbuf так, чтобы данные из них читались в цикле 6 последовательно.

Основным отличием BLIS от более ранних реализаций является разделение кода на микроядро, содержащее только самый внутренний цикл, реализованное на ассемблере, и макроядро, содержащее циклы 4 и 5, реализованное на языке Си. Внешние три цикла не изменились. Части Abuf и Bbuf, используемые в цикле микроядра, называются микропанелями A и B соответственно. На Рисунке 2 показано распределение данных по уровням кэша и направление обхода циклов.

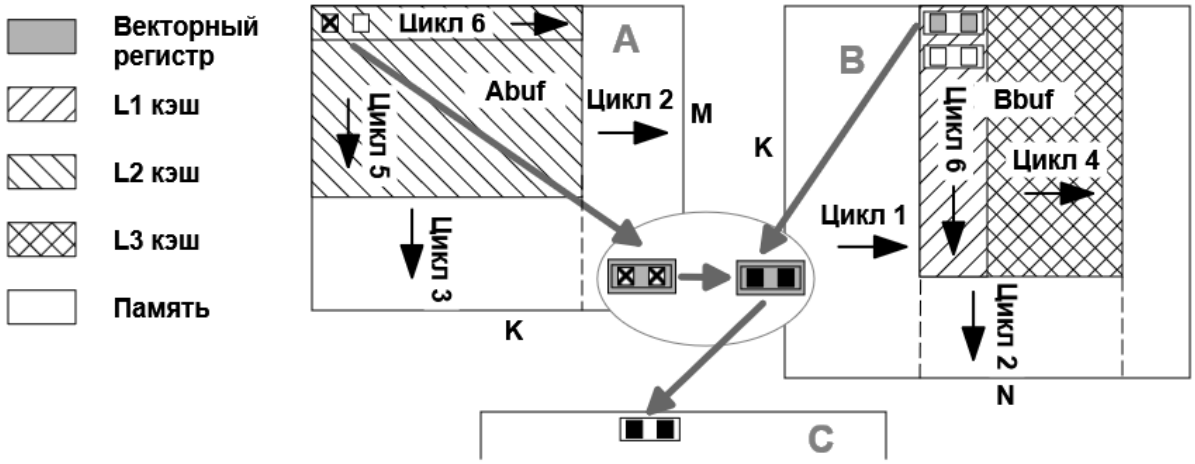

Рисунок 2 – Направления обхода циклов
Figure 2 – Loop traversal directions

Модель вычисления параметров циклов описана в работе [9], наиболее важные формулы для размеров микроядра и шага k_step приведены ниже как (1) и (2).

$$m_r = \left\lfloor \frac{\sqrt{N_{VEC}L_{VFMA}N_{VFMA}}}{N_{VEC}} \right\rfloor N_{VEC}, n_r = \left\lceil \frac{N_{VEC}L_{VFMA}N_{VFMA}}{m_r} \right\rceil, \quad (1)$$

где m_r – высота микроядра, n_r – ширина микроядра, N_{VEC} – число элементов матрицы в векторном регистре, L_{VFMA} – задержка выполнения векторных инструкций для связанных данных, N_{VFMA} – пропускная способность ЦПУ для векторных инструкций.

$$K_A \leq \left\lfloor \frac{W_{L1}-1}{1+{}^{n_r}/_{m_r}} \right\rfloor, \; k_{step} = \frac{K_A N_{L1} C_{L1}}{m_r S_{DATA}}, \quad (2)$$

где K_A – шаг между последовательными микропанелями A в расстояниях между кэшлиниями одного ассоциативного набора кэша L1 (выравнивание микропанелей A), W_{L1} – ассоциативность L1, C_{L1} – размер кэш-линии L1, S_{DATA} – размер элемента матрицы в байтах.

Выравнивание K_A введено для того, чтобы в каждом цикле микроядра новые данные из Abuf вытесняли из L1 предыдущие, не взаимодействуя с данными Bbuf. Согласно этим формулам, предпочтение должно отдаваться более «высоким» ядрам.

В наиболее современной работе Van Zee и van de Geijn [5] предлагается модификация алгоритма с целью заполнения буферов Abuf и Bbuf по частям в цикле 6 при первом обращении. Это позволяет сэкономить на сохранении данных в L1 и L2 уровнях и последующем чтении их в регистры в микроядре. Этот подход получил название FIP.

Нужно отметить, что в пакетах OpenBLAS и BLIS отсутствует специальная ветка кода для Skylake, поэтому в качестве точки отсчета взят код из пакета BLIS для предыдущей архитектуры Intel Haswell (далее базовый код или базовый вариант).

Оценка влияния одновременной многопоточности на производительность

Суперскалярные архитектуры подвержены двум типам простоев [1]. Кроме общего со скалярными «вертикального» простоя, обозначающего полную остановку вычислений до наступления определенного события, в них возможны также «горизонтальные» простои – неполное задействование функциональных модулей на определенном такте. В модели, определяемой формулами (1) и (2), ограничивающими факторами являются порты векторных вычислений (0 и 1 для Skylake) для всех размеров микроядер, для которых выполняется условие (3):

$$\frac{m_r \acute{n_r}}{m_r + \acute{n_r} + p} \geq \frac{N_{VFMA}}{N_{L1}}, \quad \acute{n_r} = \frac{n_r}{N_{VEC}}, \quad (3)$$

где N_{L1} – число одновременных загрузок из памяти, p – число предвыборок на цикл микроядра, $\acute{n_r}$ – ширина микроядра в векторных регистрах.

Отношение $\frac{N_{VFMA}}{N_{L1}}$ на современных архитектурах близко к 1 (на рассматриваемой равно 1 [10]). Таким образом, микроядро, у которого $m_r > 1$ и $n_r > N_{VEC}$ будет удовлетворять этому условию. Задачей второго потока является утилизация простоев ограничивающих портов.

Поскольку код микроядер хорошо оптимизирован, можно сделать допущение, что «горизонтальные» простои портов ограничивающего типа отсутствуют (в такте используются либо все имеющиеся, либо ни одного). В этом случае производительность двух потоков можно рассматривать как два эксперимента с двумя элементарными исходами – порты заняты и порты простаивают. Вероятность занятости равна $p_{busy} = C_{base}/C_{max}$, где C_{base} – производительность одного потока, C_{max} – теоретическая максимальная производительность для данной машины. Вероятность простоя

соответственно равна $1 - p_{busy}$. Тогда производительность можно оценить как сумму вероятностей возможных исходов этих экспериментов, показанную в Таблице 1.

Таблица 1 – Вероятность занятости векторных портов
Table 1 – Vector ports utilization probabilities

Поток 1	Поток 2	Занятость портов	Вероятность при синхронном старте	Вероятность при равномерно случайном старте
▩	▩	▩ ▩	p_{busy}	${p_{busy}}^2$
▩	□	▩	0	$p_{busy} - {p_{busy}}^2$
□	▩	▩	0	$p_{busy} - {p_{busy}}^2$
□	□	* □	$1 - p_{busy}$	$(1 - p_{busy})^2$

* занятость порта в этом случае не определена, поскольку простой может быть вызван зависимостью по данным, задержкой при обращении к памяти и рядом других причин. Также порт будет занят, если на него была двукратная нагрузка на предыдущем такте. Предположение, что порты в этом случае действительно будут простаивать, является пессимистичным.

Видно, что наихудшим возможным сценарием будет синхронное выполнение одного и того же кода обоими потоками. В случае же равномерного распределения смещения начала работы потоков относительно друг друга, пессимистичная оценка их совместной производительности будет определяться формулой:

$$\acute{p}_{busy} = \frac{2p_{busy}}{1+{p_{busy}}^2}. \quad (4)$$

Для случая C_{base}=107,4 (максимальная производительность базового варианта из раздела «Результаты»), C_{max}=121,6 (частота процессора, умноженная на 2 * N_{VFMA} * N_{VEC}) пессимистичная оценка $\acute{p}_{busy} \approx 0{,}99$, что означает устранение практически всех простоев вычислительных портов. Поскольку код умножения матриц состоит из циклического вызова микроядра, нужно обеспечить равномерный сдвиг начала циклов, чего можно достичь, добавив небольшую задержку в одном из них.

Структурные потери

Одной из проблем одновременной многопоточности в архитектурах x64 является равное разделение некоторых ресурсов ядра между потоками. В частности, для Skylake такими ресурсами являются выборка и декодирование инструкций [10], а также кэш микроинструкций [11]. Согласно [10], выборка инструкций получает из памяти 16 байт кода в такт, 8 байт на поток. Длина FMA и AVX инструкций, из которых полностью состоит цикл микроядра, от 4 до 6 байт, таким образом каждый поток может выбирать 1-2 инструкции в такт, чего явно недостаточно. Обеспечить равномерную загрузку портов в этом случае должен кэш микроинструкций, но он тоже делится пополам. Достаточно сложно аналитически оценить, насколько это уменьшает производительность. Сравним экспериментально базовую версию и версию с дополнительным потоком, код которого приведен на Рисунке 3. Поток не выполняет никаких вычислений и обращений к памяти, деля нагрузку между портами 5 и 6, малоиспользуемыми основным потоком. Длина цикла подобрана вручную так, чтобы поток завершился немного раньше основного.

```
i = 9500000;
while (--i) // ports 0,1 or 5 + port 6
     _mm_pause();    // PAUSE, port 6 + delay > 140 ticks
```

Рисунок 3 – Код пустого потока
Figure 3 – Dummy thread code

Наиболее заметные различия в счетчиках событий при профилировании с помощью Intel VTune и итоговая производительность приведены в Таблице 2. Отмеченные различия говорят о недостаточном потоке из очереди микроинструкций в таблицу переименований регистров. Из результатов в последней строке Таблицы 2 видно, что присутствие второго потока снижает производительность для данного кода на данной архитектуре примерно на 6,5 %.

Таблица 2 – Различия счетчиков событий для одного и двух потоков
Table 2 – Event counters differences for one and two thread cases

Метрика	С пустым потоком	Базовый
EXE_ACTIVITY.1_PORTS_UTIL	956,340,000	447,120,000
EXE_ACTIVITY.2_PORTS_UTIL	2,972,160,000	1,535,220,000
EXE_ACTIVITY.EXE_BOUND_0_PORTS	121,500,000	83,700,000
FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_1_PS	120,015,000	28,485,000
производительность, GFLOPS	99,8	106,7

Эксперименты с количеством PAUSE внутри цикла показывают, что ее удаление ожидаемо снижает производительность, но добавление инструкций PAUSE внутрь цикла не приводит к ее изменению (Таблица 3). Можно предположить, что это подтверждает гипотезу о потере производительности из-за структурных ограничений микроархитектуры, а не вследствие загрузки ядра кодом второго потока.

Таблица 3 – Производительность для различного количества PAUSE
Table 3 – Performance for different PAUSE amounts

Количество PAUSE	0	1	2
Производительность	59,9	99,8	99,7

Выяснение возможностей оптимизации нуждается в дополнительном исследовании. Величину потерь можно уменьшить, оптимизировав код под в два раза уменьшенный кэш микроопераций в соответствии с правилами, перечисленными в [11].

Исходя из вышесказанного, можно определить коэффициент структурных потерь как $k_{Lshare} = C_{dummy}/C_{base}$, где C_{dummy} – производительность варианта с одним рабочим и одним пустым потоком. Добавив это к формуле (4) и заменив p_{busy} на C_{base}/C_{max}, получим выражение для производительности с учетом структурных потерь:

$$C = \frac{2C_{dummy}}{1+\left(\frac{C_{base}}{C_{max}}\right)^2}. \quad (5)$$

Потери при синхронизации

Для того чтобы быть эффективной, многопоточная реализация должна использовать какой-то объем данных совместно [4]. Данные в буферах Abuf и Bbuf меняются, и, вне зависимости от того, какой из буферов будет использоваться совместно, потокам потребуется синхронизация до или после их обновления.

Есть два способа синхронизации потоков. Первый, более традиционный, заключается в запуске и остановке потока средствами операционной системы. Преимуществом этого способа является отсутствие нагрузки на ядро в неактивном состоянии, а недостатком – высокая стоимость остановки и последующего запуска [12]. Второй подход заключается в использовании спинлока – периодическом опросе определенной ячейки памяти до появления в ней нужного значения. Применительно к одновременной многопоточности серьезным недостатком подобного способа является потребление циклом ресурсов обоих потоков [13]. Однако в семействе x64 спинлок между потоками на одном физическом ядре будет иметь некоторые преимущества по сравнению со спинлоком между произвольными потоками в системе. Разделенный между потоками L1 кэш и сериализованная запись (данные попадают в память в порядке следования инструкций) позволяют синхронизировать потоки в пределах одного ядра без привлечения протокола когерентности кэша (MOESI), что снижает нагрузку на подсистему памяти. Несмотря на это, синхронизация на спинлоке приводит к значительной потере производительности второго потока (Таблица 3, строка 0 PAUSE) из-за постоянной загрузки соответствующих портов и сброса очереди команд при выходе из цикла по причине неверного предсказания перехода.

По этой причине в ранних работах [12] и [13], посвященных одновременной многопоточности, использование спинлока сочтено неудовлетворительным. В [12] рассматривается экспериментальный процессор, в котором есть инструкции управления логическими потоками внутри ядра, а в [13] – специальный функциональный модуль процессора, реализующий примитивы синхронизации ОС на уровне оборудования. В расширении SSE3 процессоров Intel была добавлена пара инструкций MWAIT/MONITOR, реализующая подобную функциональность. Но до 2013 года они были доступны только в режиме ring 0 и таким образом не могли использоваться для синхронизации в пространстве пользователя[1]. Позже они стали доступны в Ring 3 только как отдельно настраиваемая опция в линейке специализированных процессоров Xeon Phi. Примечательно что одна из немногих реализаций одновременной многопоточности при умножении матриц выполнена для этих процессоров [4]. К сожалению, затраты на синхронизацию в этой работе не рассматриваются.

В 2015 году в архитектуре Intel Skylake для улучшения производительности спинлоков инструкция PAUSE была модифицирована, в частности, простой логического ядра был увеличен до 140 тактов[1]. До конца 2021 года эта инструкция оставалась единственным широко доступным механизмом синхронизации одновременной многопоточности в процессорах x64. В современных архитектурах появился ряд инструкций для точной синхронизации SMT потоков в пространстве пользователя, но из-за их низкой распространенности провести эксперименты с ними не удалось.

Затраты для сравнительно коротких по сравнению с микроядром циклов синхронизации можно приблизительно оценить через долю времени, в которое выполняется только один поток с пониженной эффективностью:

[1] Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-044b, June 2021.

$$L_{sync} = \frac{N_{sync}D}{N_{mc}T_{mc}k_c}(\dot{C} - C_{base}k_c), \quad (6)$$

где N_{sync} – число циклов синхронизации, N_{mc} – число вызовов микроядра, T_{mc} – число тактов в микроядре, $\dot{C}$ – производительность по формуле (5), D – длина цикла синхронизации в тактах, k_c – коэффициент эффективности оставшегося потока.

Для PAUSE из архитектуры Skylake D = 140, а $k_c \approx 1$ (потому что два последовательных обращения к порту 6 на 140 циклов выполнятся без задержек почти всегда). Формула (5) для общей производительности трансформируется в формулу (7):

$$C = k_{sync}\, k_c C_{base} + \frac{(1-k_{sync})2C_{dummy}}{1+\left(\frac{C_{base}}{C_{max}}\right)^2}, \quad k_{sync} = \frac{N_{sync}D}{N_{mc}T_{mc}k_c}. \quad (7)$$

Таким образом, использование нескольких потоков теоретически может дать преимущество, если выполняется неравенство (8):

$$\frac{(1-k_{sync})2C_{dummy}}{1+\left(\frac{C_{base}}{C_{max}}\right)^2} - (1 - k_{sync}\, k_c\,)C_{base} > 0. \quad (8)$$

Уточнение параметров микроядра

Для дальнейшего изложения k_{step} удобнее выразить как долю кэша L1, выделяемого под микропанель B:

$$k_{sL1} = \frac{k_{step}n_r S_{DATA}}{S_{L1}}. \quad (9)$$

Поскольку в циклах микроядра данные Abuf вытесняют предыдущие, то k_{sL1} можно также определить через соотношение размеров микропанели A и микропанели B. Для максимально высокого микроядра 6x16, не нарушающего неравенство (3), в целевой архитектуре k_{sL1} = 0,625. Таким образом, для кэша со степенью ассоциативности 8, если одну кэш-линию в каждом наборе оставить для данных C, две будут использованы данными Abuf и пять – данными Bbuf .

Но если имеется два потока, каждый из которых умножает общий Bbuf на свою порцию Abuf, количество данных микропанели A надо также умножать на 2. В этом случае данным Bbuf достанется всего 4 линии в каждом наборе, а k_{sL1}=0,5.

Альтернативным подходом является использование «широкого» микроядра 2x48. Это приведет к увеличению количества сохранений данных C в три раза (отношение высоты микроядер), но по формуле (5) можно предполагать утилизацию 93 % простоев вторым потоком. С другой стороны, такое микроядро позволит задействовать 0,75 кэша L1 для данных B, оставив только одну кэш-линию для данных A, и по этой причине потребует в полтора раза меньше синхронизаций потоков. Еще одним преимуществом подобного подхода является сбалансированный код заполнения Abuf – для чередования данных из двух строк нужно столько же векторных инструкций, сколько и для их загрузки, в то время как для чередования шести строк требуется 10 инструкций и, таким образом, 1,66 векторных инструкций на каждую загрузку из памяти.

Также нужно учесть, что оба потока будут делить единственную кэш-линию каждого ассоциативного набора, и из-за этого могут вытеснять данные Bbuf одновременной загрузкой в один набор. Предотвратить это можно, чередуя микропанели A для потоков вместо последовательного размещения. Поскольку их размер для «широкого» микроядра невелик, для выравнивания K_A нужно использовать следующую формулу:

$$K_A = \frac{1}{2^{\left\lfloor \log_2 \frac{N_{L1} C_{L1}}{k_{step} m_r S_{DATA}} \right\rfloor}}, \quad (10)$$

смысл которой обратен исходному – несколько последовательно размещенных микропанелей Abuf не должны попадать в один ассоциативный набор. Округление степени до ближайшего снизу целого нужно, чтобы итоговое выравнивание $K_A N_{L1} C_{L1}$ получилось целым числом. k_{step} можно вычислить из k_{sL1} по формуле, обратной формуле (9).

Результаты

Для синхронизации потоков используется рассмотренный ранее спинлок с PAUSE без использования префикса LOCK (протокола когерентности кэша). Синхронизирующие переменные расположены в памяти непосредственно за буфером Bbuf, что должно исключить вытеснение его данных при синхронизациях. Каждый поток выполняет умножение буфера Bbuf на половину буфера Abuf.

Потоки асимметричны: второй поток имеет несколько ненужных предвыборок после вызова микроядра. Буфер Bbuf заполняется более быстрым первым потоком до точки синхронизации, для чего в этих итерациях цикла 5 четыре вызова микроядра переносятся во второй поток (второй поток в этом случае обрабатывает немного большую часть буфера A). Каждый поток заполняет свою часть буфера Abuf микропанелями A в первой итерации цикла 3.

Использованы микроядро 6x16 – вариант SMT 6x16 (размер аналогичен базовому микроядру BLIS) и микроядро 2x48 – вариант SMT 2x48 asym (параметры ядра, подсчитанные в разделе «Уточнение параметров микроядра»). Для этого размера также сравнивается микроядро с симметричными потоками (кроме периода заполнения Bbuf) – вариант SMT 2x48 sym, и частично реализованный (только для микропанели B) FIP алгоритм из работы [5] – вариант SMT 2x48 FIP.

Замеры производились в следующих условиях:

– Операционная система: Ubuntu 22.04 LTS.

– Процессор: Intel Core i5 8265U, частота 3.8 GHz.

– Исходные матрицы: размер 2304x2304, как удобный для используемых микроядер, элементы одинарной точности.

– Расположение в памяти: BLIS – вызовы bli_srandm, bli_ssetm, двупоточная реализация – malloc.

– Используемое микроядро BLIS: Haswell 6x8 (6x16 для float). Исключены умножение на β, обработка краев.

– Измерение производительности: функция clock_gettime (CLOCK_MONOTONIC) вызывается до и после вызова API. Величина 2 * M * N * K делится на измеренный промежуток времени. Для двупоточного варианта запуск и остановка второго потока операционной системой не входят в этот промежуток. Его вычисления ограничены точками сихронизации с первым потоком, таким образом все вычисления выполняются внутри измеряемого интервала.

– Среднее значение считается для 100 запусков.

– Код двупоточной реализации: https://github.com/evgnort/sgemm.

Для определения доли времени, которое потоки тратят на синхронизацию, произведен подсчет числа циклов синхронизации в обоих потоках, и числа вызовов микроядра для варианта SMT 2x48 (Таблица 4):

Таблица 4 – Число циклов синхронизации
Table 4 – Number of synchronization cycles

Вызовов микроядра, N_{mc}	995328
Циклов ожидания (быстрый поток), N_{sync1}	5288
Циклов ожидания (медленный поток), N_{sync2}	19248

Результаты сравнения реализаций и теоретических моделей (Таблица 5):

Таблица 5 – Производительность моделей и вариантов реализации
Table 5 – Performance of models and implementations

Вариант	k_{sL1}	средняя производительность, GFLOPS	максимальная производительность, GFLOPS
Базовый	0,5	106,7	107,4
SMT 6x16 модель	0,5	112,8	112,8
SMT 6x16	0,5	106,4	106,9
SMT 2x48 модель	0,75	114,1	114,1
SMT 2x48 FIP	0,75	110,1	111,5
SMT 2x48 sym	0,75	112,3	112,7
SMT 2x48 asym	0,75	112,7	113,0

Обсуждение

В данном случае некорректно говорить о точности модели, поскольку формула (7) и неравенство (8) описывают гипотетический случай исполнения исходного кода двумя потоками, что при реализации встретило различные ограничения. Наиболее близкая к исходному коду SMT 6x16 реализация показала слабую результативность. Стоит также отметить невысокий результат полностью симметричного FIP варианта. В данном случае, помимо вычислительных портов, при заполнении Bbuf ограничивающим стал также порт 4 для сохранения данных. Поскольку данный участок кода находится вплотную к точке синхронизации, двупоточная реализация оказалось неэффективной. Можно сказать, что неравенство (8) показывает наличие простоев, но их утилизация может потребовать значительного изменения алгоритма. Уточненная для итогового алгоритма модель показывает возможность дальнейшего его улучшения. Можно отметить снижение структурных потерь для реализаций 2x48.

Результаты позволяют оценить использование предложенного механизма синхронизации. Таблица 4 показывает, что один цикл синхронизации приходится примерно на каждые 40 вызовов микроядра. Поскольку в микроядре для варианта SMT 2x48 примерно 1200 тактов, $k_{sync} \approx 0{,}003$. Потери на синхронизацию по формуле (6) пренебрежимо малы, основными потерями являются структурные.

Значительно лучшие результаты «горизонтального» ядра (SMT 2x48) подтверждают необходимость анализа и модификации модели из формул (1) и (2) в соответствие с разделом «Уточнение параметров микроядра» для уменьшения конкуренции за кэш L1. Наиболее влияющим фактором оказалось сбалансированное заполнение данных Abuf, как видно из Таблицы 4, влияние блокировок очень незначительно. Также сыграло роль уменьшение количества инструкций VBROADCASTSS для загрузки данных A, дающих большую задержку. Дополнительные простои из-за увеличения числа обращений к данным C не сказались на производительности, поскольку были утилизованы вторым потоком.

Из Таблицы 5 видно, что асимметричные потоки дают, как и ожидалось, несколько лучшую производительность. Но результаты обоих вариантов достаточно близки. Возможная причина в том, что выход из синхронизационных циклов вносит достаточный элемент случайности в сдвиг потоков относительно друг друга.

Целью работы было выяснение возможности оптимизации операции умножения матриц с помощью одновременной многопоточности. Результаты показывают, что это возможно при выполнении следующих условий:

– Существуют размеры микроядра, для которых выполнимо неравенство (3) – ограничивающим фактором должна быть скорость вычислений. Применимость формул (4) и (10) к микроядрам, ограниченным работой с памятью, выглядит спорно и требует отдельного исследования. Сравнительно небольшой участок кода с ограничением по пропускной способности памяти в варианте SMT 2x48 FIP заметно снизил производительность.

– Кэш L1 доступен потокам в полном объеме на равноправной основе.

– Верхняя оценка производительности нескольких потоков по формуле (7) выше производительности однопоточного варианта – сумма потерь от синхронизации и разделения ресурсов не превышает возможную утилизацию простоев вычислительных модулей.

Перспективными направлениями дальнейшей работы представляются дальнейшая оптимизация кода микроядра под кэш микрокода (увеличение k_{Lshare}), создание алгоритма обработки краев матриц произвольного размера, анализ возможности использования одновременной многопоточности для умножения матриц на других архитектурах и для решения других задач линейной алгебры, модификация уравнения (4) с учетом временной локальности простоев.

Заключение

Приведенные результаты показывают, что как минимум в одной архитектуре для как минимум одного размера матриц применение логических потоков для общего умножения матриц показывает лучшие результаты в сравнении с алгоритмами BLIS с одним логическим потоком, соответственно такая оптимизация может быть оправданной.

Создан критерий возможности применения подобного решения для произвольной архитектуры:

– разделение кэша L1 в полном объеме;

– выполнение неравенства (3) (ограничение производительности скоростью векторных вычислений);

– выполнение неравенства (8) (наличие в коде достаточного для утилизации вторым потоком количества простоев).

Описанный подход может быть использован для произвольного вычислительного алгоритма с учетом его особенностей.

СПИСОК ИСТОЧНИКОВ / REFERENCES

1. Tullsen D.M., Eggers S.J., Levy H.M. Simultaneous multithreading: maximizing on-chip parallelism. In: *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture, 22-24 June 1995, Santa Margherita Ligure, Italy.* New York: Association for Computing Machinery; 1995. P. 392–403. https://doi.org/10.1145/223982.224449

2. Marr D.T., Binns F., Hill D.L., Hinton G., Koufaty D.A., Miller J.A., Upton M. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal.* 2002;6(1):4–15.
3. Leng T., Ali R., Hsieh J., Mashayekhi V., Rooholamini R. An Empirical Study of Hyper-Threading in High Performance Computing Clusters. In: *Proceedings of The 3rd LCI International Conference on Linux Clusters: The HPC Revolution 2002, 23-25 October 2002, Saint Petersburg, FL, USA.* Linux Clusters Institute; 2002.
4. Smith T.M., Van De Geijn R., Smelyanskiy M., Hammond J.R., Van Zee F.G. Anatomy of High-Performance Many-Threaded Matrix Multiplication. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium, 19-23 May 2014, Phoenix, AZ, USA.* IEEE Computer Society; 2024. P. 1049–1059. https://doi.org/10.1109/IPDPS.2014.110
5. Xu R.G., Van Zee F.G., Van De Geijn R.A. GEMMFIP: Unifying GEMM in BLIS. URL: https://doi.org/10.48550/arXiv.2302.08417 (Accessed 15th April 2024).
6. Goto K., Van De Geijn R.A. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software.* 2008;34(3). https://doi.org/10.1145/1356052.1356053
7. Van Zee F.G., Van De Geijn R.A. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software.* 2015;41(3). https://doi.org/10.1145/2764454
8. Huang J., Van De Geijn R.A. BLISlab: A Sandbox for Optimizing GEMM. URL: https://doi.org/10.48550/arXiv.1609.00076 (Accessed 15th April 2024)
9. Low T.M., Igual F.D., Smith T.M., Quintana-Orti E.S. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Transactions on Mathematical Software.* 2016;43(2). https://doi.org/10.1145/2925987
10. Fog A. The Microarchitecture of Intel, AMD and VIA CPUs. URL: https://www.agner.org/optimize/microarchitecture.pdf (Accessed 17th April 2024).
11. Ren X., Moody L., Taram M., Jordan M., Tullsen D.M., Venkat A. I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 14-18 June 2021, Valencia, Spain.* IEEE; 2021. P. 361–374. https://doi.org/10.1109/ISCA52012.2021.00036
12. Kim D., Liao S.S.-W., Wang P.H., Del Cuvillo J., Tian X., Zou X., Wang H., Yeung D., Girkar M., Shen J.P. Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. In: *International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA.* IEEE; 2004. P. 27–38. https://doi.org/10.1109/CGO.2004.1281661
13. Tullsen D.M., Lo J.L., Eggers S.J., Levy H.M. Supporting fine-grained synchronization on a simultaneous multithreading processor. In: *HPCA '99: Proceedings of the 5th International Symposium on High-Performance Computer Architecture, 09-13 January 1999, Orlando, FL, USA.* NW Washington: IEEE Computer Society; 1999. P. 54–58. https://doi.org/10.1109/HPCA.1999.744326

ИНФОРМАЦИЯ ОБ АВТОРАХ / INFORMATION ABOUT THE AUTHORS

Буевич Евгений Андреевич, аспирант, Московский государственный технологический университет «СТАНКИН», Москва, Российская Федерация.
email: gftcompmod@gmail.com

Evgeniy A. Buevich, postgraduate student, Moscow State Technological University "STANKIN", Moscow, the Russian Federation.

Статья поступила в редакцию 27.05.2024; одобрена после рецензирования 14.06.2024; принята к публикации 20.06.2024.

The article was submitted 27.05.2024; approved after reviewing 14.06.2024; accepted for publication 20.06.2024.