

МИНОБРНАУКИ РОССИИ  
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ  
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ  
«ЛЭТИ» им. В. И. УЛЬЯНОВА (ЛЕНИНА)  
Кафедра вычислительной техники

**ОТЧЕТ**  
**по лабораторной работе №2**  
**по дисциплине «Параллельные алгоритмы и системы»**  
**Тема: Умножение матриц**

Студент гр. 9306	_____	Ковтун Р. С.
Преподаватель	_____	Пазников А. А.

Санкт-Петербург  
2023

## Оглавление

Оглавление.....	2
Цель .....	3
Задание .....	3
Ход работы.....	3
Реализация в лоб .....	4
Увеличение пространственной локальности .....	5
Флаги оптимизации .....	7
Распараллеливание.....	9
Блочный алгоритм.....	10
Векторизация кода .....	13
Выводы.....	16
ПРИЛОЖЕНИЕ .....	17
Листинг основной программы.....	17
Листинг последней оптимизации (FMA3).....	18

## Цель

Познакомиться с методами оптимизации ПО и реализовать их на примере матричного умножения.

## Задание

Реализовать программу умножения матриц. Итеративно повышать производительность программы, оптимизируя различные элементы программы.

## Ход работы

$$A = M(m \times k), B = M(k \times n),$$

$$C = A \times B = M(m \times n)$$

$$c_{ij} = \sum_r^m a_{ir} b_{rj} \quad (i = 1..k, j = 1..n)$$

Прежде чем начать, посмотрим характеристики системы (рис. 1).

Model number ?	i5-8250U
CPU part number	FJ8067703282221 is an OEM/tray microprocessor
Frequency ?	1.6 GHz / 1600 MHz
Maximum turbo frequency	3.4 GHz / 3400 MHz
Instruction set	x86
Microarchitecture	Kaby Lake
Processor core ?	Kaby Lake-R
Core stepping ?	Y0
Manufacturing process	0.014 micron
Data width	64 bit
The number of CPU cores	4
The number of threads	8
Floating Point Unit	Integrated
Level 1 cache size ?	4 x 32 KB 8-way set associative instruction caches 4 x 32 KB 8-way set associative data caches
Level 2 cache size ?	4 x 256 KB 4-way set associative caches
Level 3 cache size	6 MB 12-way set associative shared cache
Physical memory	32 GB
Multiprocessing	Not supported
Extensions & Technologies	<ul style="list-style-type: none"><li>MMX instructions</li><li>SSE / Streaming SIMD Extensions</li><li>SSE2 / Streaming SIMD Extensions 2</li><li>SSE3 / Streaming SIMD Extensions 3</li><li>SSSE3 / Supplemental Streaming SIMD Extensions 3</li><li>SSE4 / SSE4.1 + SSE4.2 / Streaming SIMD Extensions 4</li><li>AES / Advanced Encryption Standard instructions</li><li>AVX / Advanced Vector Extensions</li><li>AVX2 / Advanced Vector Extensions 2.0</li><li>BMI / BMI1 + BMI2 / Bit Manipulation instructions</li><li>F16C / 16-bit Floating-Point conversion instructions</li><li>FMA3 / 3-operand Fused Multiply-Add instructions</li><li>EM64T / Extended Memory 64 technology / Intel 64</li><li>HT / Hyper-Threading technology</li><li>VT-x / Virtualization technology</li><li>VT-d / Virtualization for directed I/O</li><li>TBT 2.0 / Turbo Boost technology 2.0</li></ul>

Рис. 1. Характеристики вычислительного ресурса

$\text{GFLOPS} = (\text{частота ядер}) * (\text{количество логических ядер}) * (\text{число операций с плавающей запятой за такт}) / 10^9$ . Имеем **217 GFLOPS**.

## Реализация в лоб

Такая реализация предполагает вложенный цикл с 3 уровнями. С этой реализацией мы будем сравнивать производительность дальнейших оптимизаций.

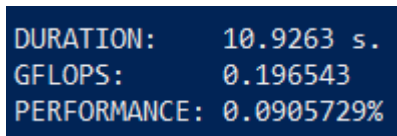
Для удобства, матрицы взяты квадратными и размером 1024x1024. Но размер в дальнейшем может меняться в зависимости от скорости работы программы (если программа выполнится слишком быстро, VTune не сможет собрать достаточно данных о производительности программы).

Оценку speedup будем проводить по GFLOPS, так как размер матриц будет меняться в ходе оптимизаций. Код всей программы представлен в приложении. Но мы рассмотрим ключевой код:

```
bench_start();
for (int i=0; i < m; ++i)
    for (int j=0; j < m; ++j)
        for (int k=0; k < m; ++k)
            C[i][j] += A[i][k] * B[k][j];
diff = bench_measure().count();

calc_flops(m*m*m, diff);
```

Имеем выхлоп в 0.2 GFLOPS на GCC (рис. 2). Остальные программы будут иметь такой же вид выходных данных, поэтому далее будем заполнять табличку производительности.



DURATION:	10.9263 s.
GFLOPS:	0.196543
PERFORMANCE:	0.0905729%

Рис. 2. Производительность реализации «в лоб» (GCC)

Взглянем на результаты профилирования (рис. 3).

**Memory bound** – влияние подсистемы памяти (cash-misses в том числе) на производительность. Видим, что у нас 36% кэш-промахов для L1. Видим долю циклов ожидания доступа к L3 – 51%, к DRAM – 63%.

CPI – cycles per instruction – 0.63. Результат неплохой, но для современных суперскаляров идеально – 0.25 (IPC = 4).

**Retiring** – загрузка конвейера команд (в идеале, 100%). Высокий показатель retiring не всегда означает, что улучшить программу нельзя (векторизацию он не видит). У нас эта метрика 39%.

**Core bound** – ограничения, не связанные с памятью – когда упор в вычислительные ресурсы (задержка операций сложения, умножения, и других операций АЛУ).

В общем, есть, куда стремиться.

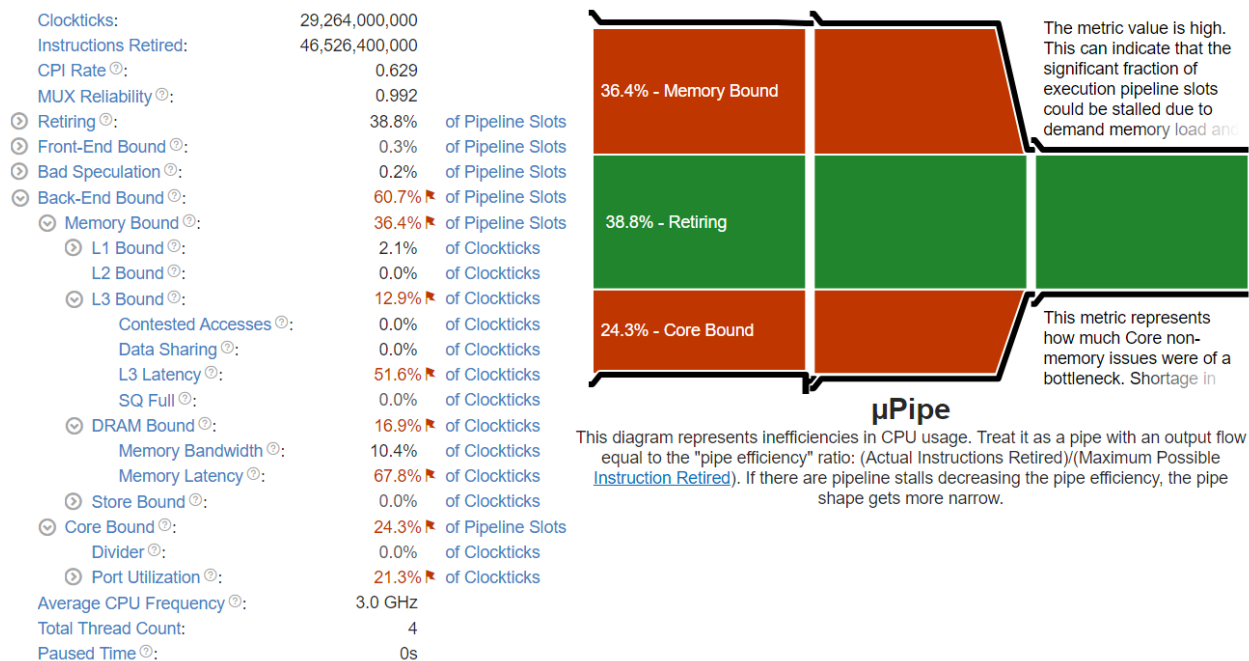


Рис. 3. Результаты профилирования реализации «в лоб»

## Увеличение пространственной локальности

В предыдущей реализации мы шли по  $i, j, k$ . На рис. 4 показана схема обращений к матрицам в последнем цикле.

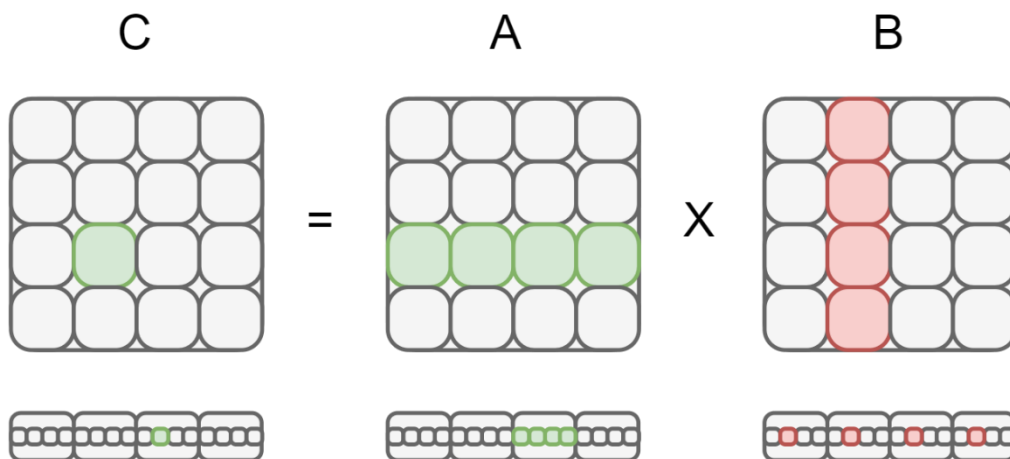


Рис. 4. Схема обращений к памяти при цикле  $i, j, k$

Такая схема получается из строк:

```
for (int k=0; k < m; ++k)
    C[i][j] += A[i][k] * B[k][j];
```

В C мы работаем с одной ячейкой, в A – со строкой, а в B – со столбцом.

Видно, что у B ужасная локальность. Надо бы её поправить, не задев остальных. Максимальная локальность достигается при последовательности i, k, j:

```
for (int j=0; j < m; ++j)
    C[i][j] += A[i][k] * B[k][j];
```

Теперь мы идём по строке C, берём один элемент из A и идём по строке B (рис. 5).

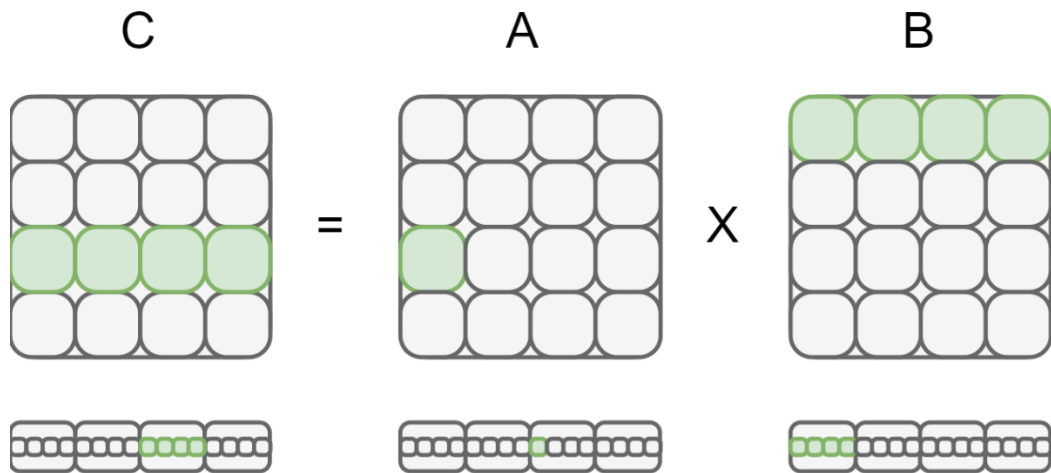


Рис. 5. Схема обращений к памяти при цикле i, k, j

Занесём данные в табличку 1.

Таблица 1. Производительность ijk-реализации

Revision	Implementation	GFLOPS	Performance percent	Relative Speedup	Absolute Speedup
1	ijk	0.2	0.092165899	1	1
2	ikj	0.51	0.235023041	2.55	2.55

Взглянем на результаты профилирования (рис. 6).

Как видно, кэш-промахов практически нет (в сумме 5.9% по всем видам памяти). CPI упал до 0.33 – хорошо, но можно лучше.

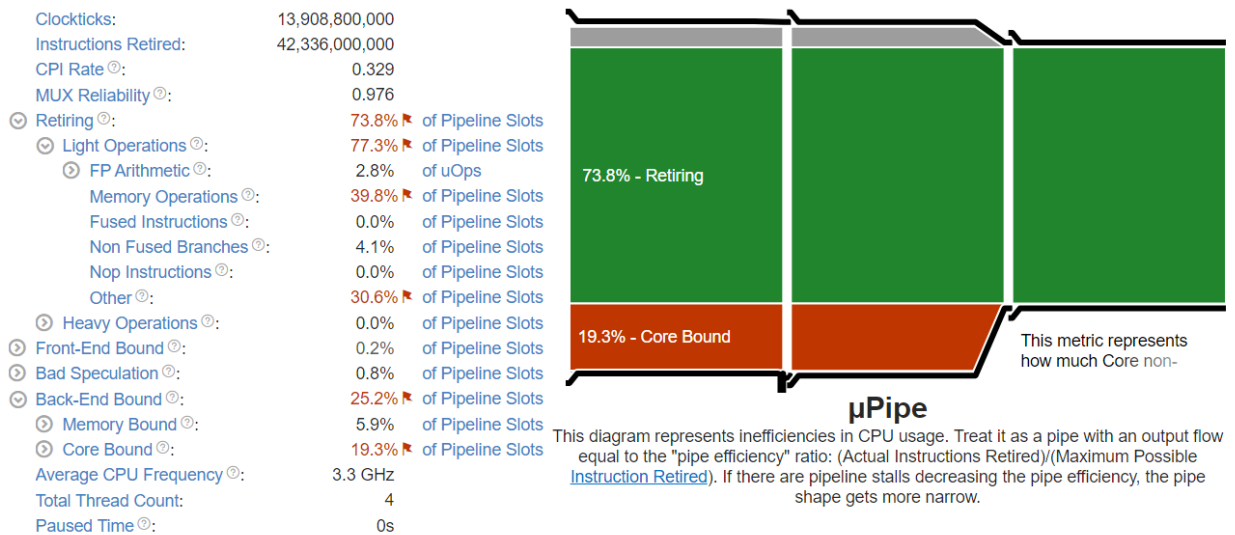


Рис. 6. Результаты профилирования *ikj*-реализации

## Флаги оптимизации

Компиляторы содержат доп функционал, позволяющий оптимизировать код, всего лишь добавив флаг компиляции. Рассмотрим некоторые флаги GCC:

-O0: Отсутствие оптимизации. Этот флаг говорит компилятору не применять оптимизации к коду, а просто транслировать его в машинный код.

-O1: Оптимизация уровня 1. Этот флаг включает некоторые базовые оптимизации, такие как удаление мертвого кода, инлайнинг функций и некоторые другие простые преобразования кода.

-O2: Оптимизация уровня 2. Этот флаг включает более продвинутые оптимизации, чем -O1, такие как векторизация циклов, раскрытие циклов и многие другие преобразования кода.

-O3: Оптимизация уровня 3. Этот флаг включает еще более продвинутые оптимизации, такие как раздельная компиляция, межпроцедурный анализ и дополнительные оптимизации кода, которые могут повысить производительность на порядок.

-Os: Оптимизация размера. Этот флаг нацелен на уменьшение размера исполняемого файла за счет уменьшения размера кода. Он включает оптимизации, которые удаляют неиспользуемый код, сжимают константы и многое другое. Код, скомпилированный с -Os, может быть медленнее, чем с -O2 или -O3, но будет занимать меньше места на диске.

Флаг `-Ofast` включает все доступные оптимизации, включая использование математических функций более высокой точности и игнорирование некоторых стандартных соображений безопасности, что может дать максимальную производительность, но может привести к неожиданным результатам при выполнении некоторых операций.

На рис. 7 представлены результаты работы программы с разными флагами оптимизации (представлены результаты в среднем из 10 тестов).

```
PS D:\Users\vasab\Documents\MPS\Labs\ParAiS\Matrix> g++ matrix3_02.cpp -O0 -o matrix3; ./matrix3

DURATION: 4.33698 s.
GFLOPS: 0.495156
PERFORMANCE: 0.228183%

PS D:\Users\vasab\Documents\MPS\Labs\ParAiS\Matrix> g++ matrix3_02.cpp -O1 -o matrix3; ./matrix3

DURATION: 1.43852 s.
GFLOPS: 1.49284
PERFORMANCE: 0.687944%

PS D:\Users\vasab\Documents\MPS\Labs\ParAiS\Matrix> g++ matrix3_02.cpp -O2 -o matrix3; ./matrix3

DURATION: 0.715743 s.
GFLOPS: 3.00036
PERFORMANCE: 1.38265%

PS D:\Users\vasab\Documents\MPS\Labs\ParAiS\Matrix> g++ matrix3_02.cpp -O3 -o matrix3; ./matrix3

DURATION: 0.725932 s.
GFLOPS: 2.95824
PERFORMANCE: 1.36325%

PS D:\Users\vasab\Documents\MPS\Labs\ParAiS\Matrix> g++ matrix3_02.cpp -Os -o matrix3; ./matrix3

DURATION: 1.84052 s.
GFLOPS: 1.16678
PERFORMANCE: 0.537688%

PS D:\Users\vasab\Documents\MPS\Labs\ParAiS\Matrix> g++ matrix3_02.cpp -Ofast -o matrix3; ./matrix3

DURATION: 0.760042 s.
GFLOPS: 2.82548
PERFORMANCE: 1.30206%
```

Рис. 7. Результаты работы программы с флагами оптимизации.

Лучше всего работает флаг `O2`. Занесём его в таблицу.

Таблица 2. Флаги оптимизации.

Revision	Implementation	GFLOPS	Performance percent	Relative Speedup	Absolute Speedup
1	ijk	0.2	0.092165899	1	1
2	ikj	0.51	0.235023041	2.55	2.55
3	-O2	3	1.382488479	5.88	15



## Распараллеливание

Имеем 8 логических ядер – используем их с помощью OMP.

```
#pragma omp parallel for schedule(static)
for (int i=0; i < m; ++i)
    for (int k=0; k < m; ++k)
        for (int j=0; j < m; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

Результат представлен в таблице 3.

Таблица 3. Результат распараллеливания

Revision	Implementation	GFLOPS	Performance percent	Relative Speedup	Absolute Speedup
1	ijk	0.2	0.092165899	1	1
2	ikj	0.51	0.235023041	2.55	2.55
3	-O2	3	1.382488479	5.88	15
4	omp	8.8	4.055299539	2.93	44

Посмотрим результаты профилирования (рис. 8).

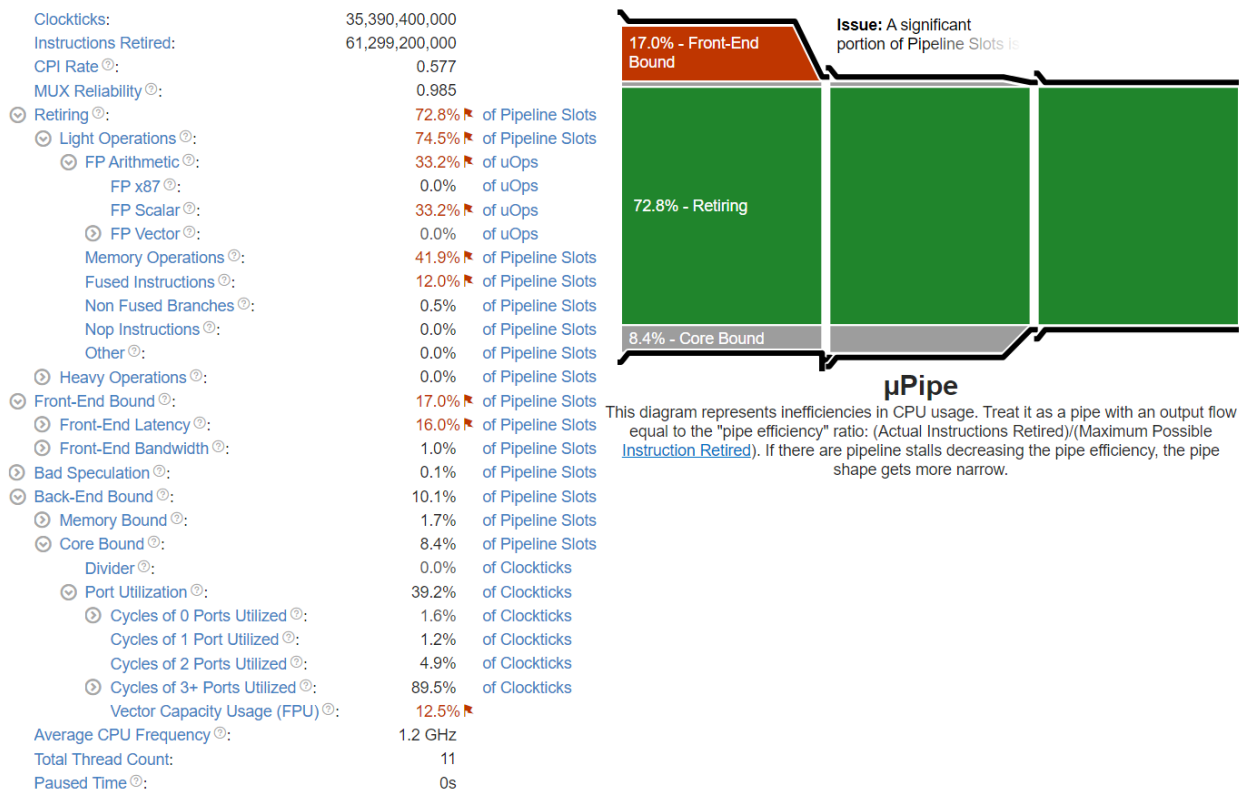


Рис. 8. Результат профилирования многопоточной реализации.

Любопытно, что суммарно имеем 11 потоков. Однако, если посмотреть на загруженность потоков, видим следующую картину (рис. 9).

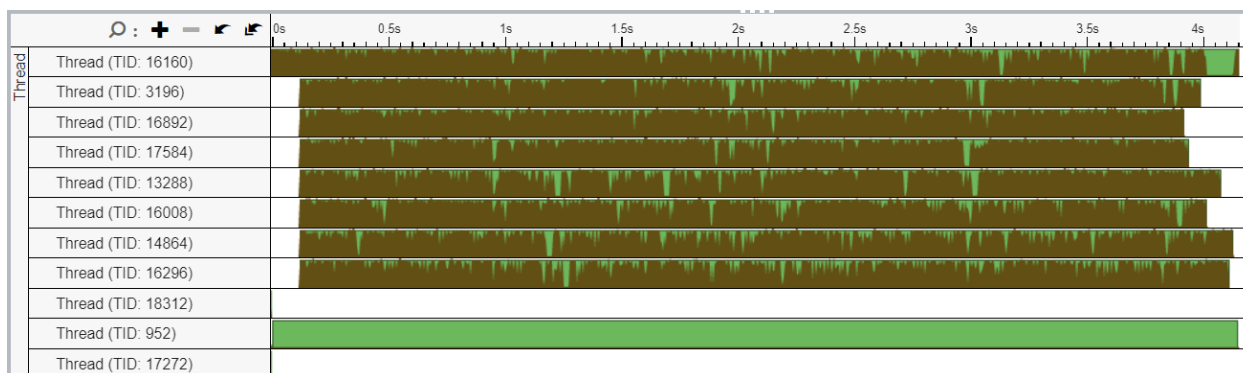


Рис. 9. Загруженность потоков.

В OMP по умолчанию стоит количество потоков, равное максимальному+некоторое число. Зачем эта константа?

Если выделить на каждый процессор ровно по одному потоку сборки, то может получиться так, что в какое-то время процессор будет простаивать, ожидая ответа от жёсткого диска или другого устройства. Если же количество потоков будет чуть больше, чем количество процессоров, то высока вероятность, что во время таких вынужденных простоев ждущий процесс будет вытеснен другим, которому есть, что вычислять. Поскольку у нас особых простоев по памяти нет, то эти потоки не выполняют полезную работу (некоторые omp сам уничтожил, и лишь один оставил – видимо, на всякий случай).

Но почему же производительность повысилась не в 8 раз? Технология Hyper-Threading не гарантирует прирост производительностикратно возможному количеству потоков на ядре. Напротив, иногда мы имеем потерю производительности – если два логических процессора используют одни и те же ресурсы, то они могут замедлять друг друга, а не ускорять работу. Это происходит из-за того, что использование одного логического процессора может привести к вытеснению кэш-линий, которые необходимы для работы другого логического процессора. Кроме того, Hyper-Threading может увеличить время доступа к общей памяти, что также может привести к ухудшению производительности.

Данный случай показывает, что у нас прирост равен числу физических ядер – Hyper-Threading не повлиял на производительность распараллеливания.

### Блочный алгоритм

Проанализируем наш алгоритм на количество обращений к памяти, требуемой для подсчёта одной строки матрицы C (рис. 10):

**C: m**  
**A: m**  
**B: m\*m**

В сумме:  $2m + m*m$ .

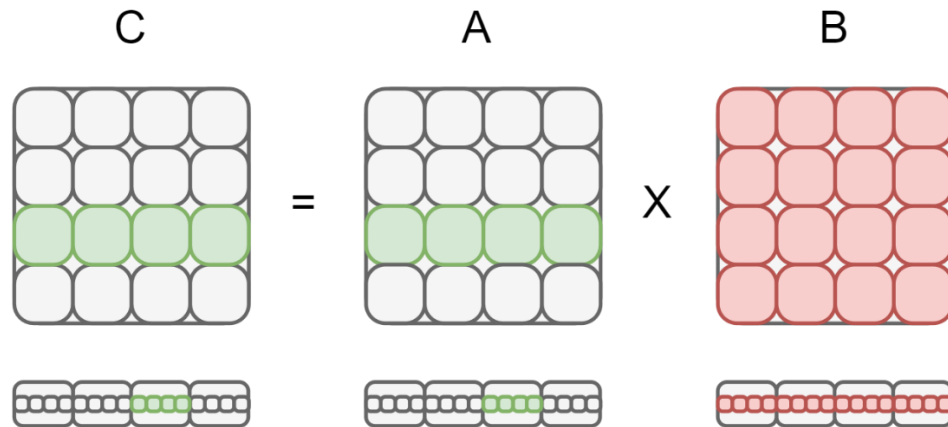


Рис. 10. Обращения к ячейкам матриц при подсчёте одной строки C

Хотелось бы сбалансировать нагрузку на матрицы. Для наших целей существует блочный алгоритм. Нужно разбить матрицу на блоки  $n \times n$ . Количество обращений к памяти при расчёте одного блока будет:

**C: n\*n**  
**A: n\*m**  
**B: m\*n**

В сумме:  $n*n + 2n*m$  (рис. 11).

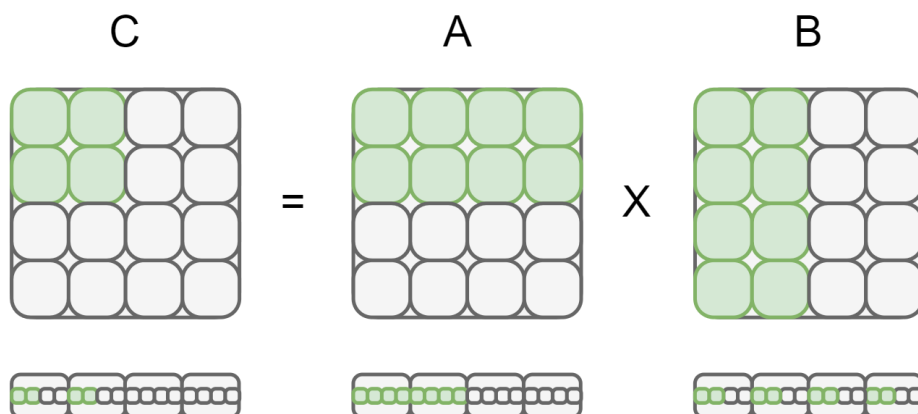


Рис. 11. Схема блочного алгоритма умножения матриц.

Итоговая производительность зависит от гиперпараметра – размера блока. На рис. 12 показаны результаты для различных блоков.

BLOCK: 8
DURATION: 0.210908 s.
GFLOPS: 10.1821
PERFORMANCE: 4.69221%
BLOCK: 16
DURATION: 0.189495 s.
GFLOPS: 11.3327
PERFORMANCE: 5.22243%
BLOCK: 32
DURATION: 0.182514 s.
GFLOPS: 11.7661
PERFORMANCE: 5.42218%
BLOCK: 64
DURATION: 0.17254 s.
GFLOPS: 12.4463
PERFORMANCE: 5.73562%
BLOCK: 128
DURATION: 0.175532 s.
GFLOPS: 12.2341
PERFORMANCE: 5.63785%
BLOCK: 256
DURATION: 0.225446 s.
GFLOPS: 9.52549
PERFORMANCE: 4.38963%
BLOCK: 512
DURATION: 0.348294 s.
GFLOPS: 6.16572
PERFORMANCE: 2.84135%

Код представлен ниже:

```
#pragma omp parallel for schedule(static)
for (int ih=0; ih < m; ih+=block_s)
{
    #pragma omp parallel for schedule(static)
    for (int jh=0; jh < m; jh+=block_s)
        for (int kh=0; kh < m; kh+=block_s)
            calc_block(A, B, C, ih, jh, kh, block_s);
}

void calc_block(float **A, float **B, float **C, int ih,
int jh, int kh, int block_s)
{
    for (int il=ih; il < ih+block_s; ++il)
        for (int kl=kh; kl < kh+block_s; ++kl)
            for (int jl=jh; jl < jh+block_s; ++jl)
                C[il][jl] += A[il][kl] * B[kl][jl];
}
```

Имеем лучшую производительность на блоке 64x64.

Слишком маленькие блоки – слишком частая перезагрузка кэша, слишком большие – не поместятся в кэш.

Рис. 12. Подбор размера блока

Результаты профилирования представлены на рис.13.

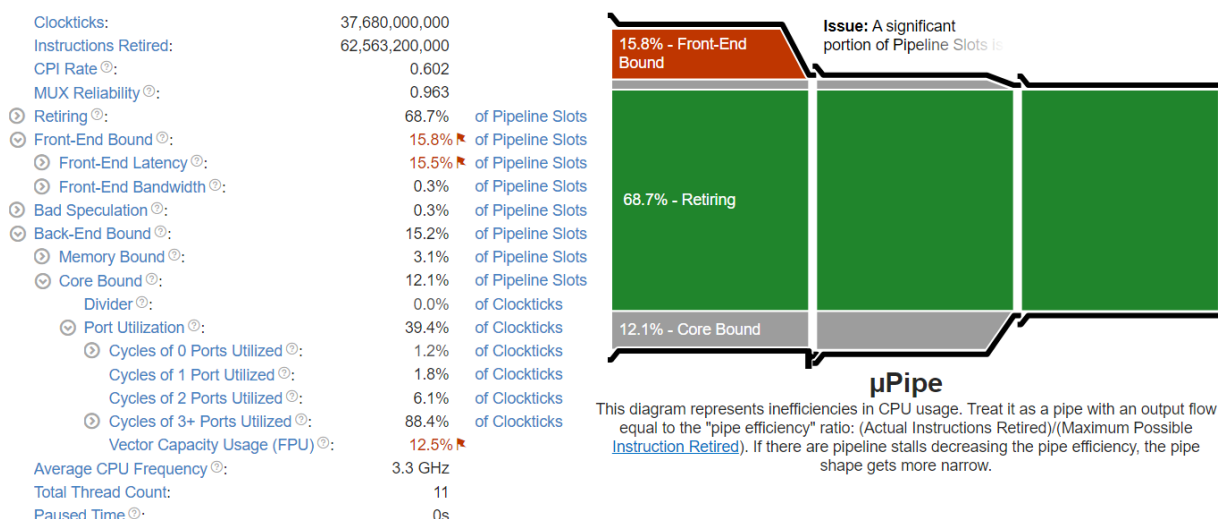


Рис. 13. Результаты профилирования блочного алгоритма.

Видим, что появился 15.8% Front-End Bound (хотя он был ещё на этапе распараллеливания) – это ограничение производительности связано с

ограничениями на этапе подачи команд в процессор. Оно возникает, когда количество команд, которые могут быть обработаны процессором, ограничивается доступом к данным, задержками кэша или медленной работы декодера команд. Диспетчеризация параллельной программы как раз может вызвать такую проблемку, но перформанс всё же растёт (таблица 14).

Таблица 4. Результат блочного алгоритма

Revision	Implementation	GFLOPS	Performance percent	Relative Speedup	Absolute Speedup
1	ijk	0.2	0.092165899	1	1
2	ikj	0.51	0.235023041	2.55	2.55
3	-O2	3	1.382488479	5.88	15
4	omp	8.8	4.055299539	2.93	44
5	blocks	11.8	5.437788018	1.34	59

## Векторизация кода

Мой камень поддерживает FMA3-инструкции, которые позволяют выполнять до 8 float-point операций за один такт. Такие инструкции используют специальные 256-битные регистры ( $8 * \text{sizeof}(\text{float})$ ).

Векторизованная программа выглядит так:

```
void calc_block(float **A, float **B, float **C, int ih, int jh, int kh,
int block_s)
{
    for (int il=ih; il < ih+block_s; il++)
    {
        for (int kl=kj; kl < kh+block_s; kl++)
        {
            __m256 a = _mm256_set1_ps(A[il][kl]);
            for (int jl=jh; jl < jh+block_s; jl+=8)
            {
                _mm256_storeu_ps(
                    C[il]+jl,
                    _mm256_fmadd_ps(
                        a,
                        _mm256_loadu_ps(B[kl] + jl),
                        _mm256_loadu_ps(C[il] + jl)
                    )
                );
            }
        }
    }
}
```

Запуск с флагом -mfma дал результат, представленный в таблице 5.

Таблица 5. Результат векторизации.

Revision	Implementation	GFLOPS	Performance percent	Relative Speedup	Absolute Speedup
1	ijk	0.2	0.092165899	1	1
2	ikj	0.51	0.235023041	2.55	2.55
3	-O2	3	1.382488479	5.88	15
4	omp	8.8	4.055299539	2.93	44
5	blocks	11.8	5.437788018	1.34	59
6	fma	53.6	24.70046083	4.54	268

Взглянем на результаты профилирования (рис. ).

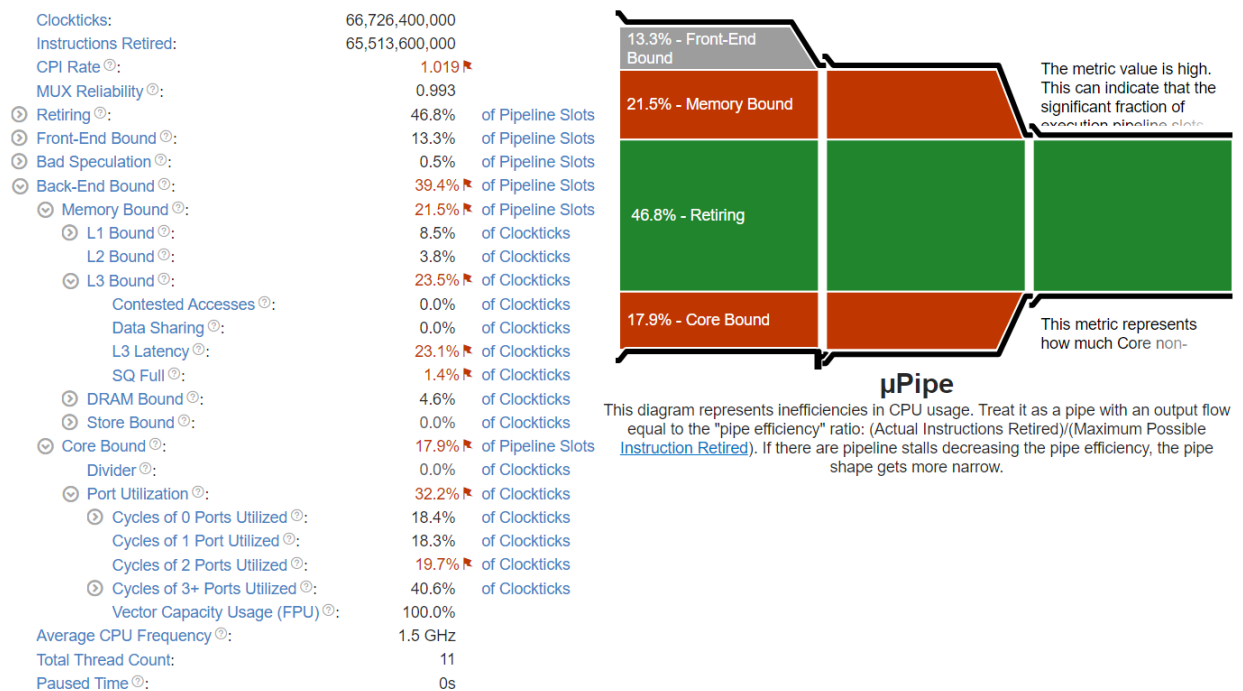


Рис. 14. Результаты профилирования векторизации.

Не впечатляет – CPI больше 1 и L3 Latency увеличилась. Почему? Введя векторизацию, мы увеличили нагрузку на память – AVX-инструкции могут не поместиться в кэш, хоть и выполняются быстрее. Также это могло произойти, из-за того, что я увеличил  $m$  до 4096 (на меньших значениях профилировщик не успевал собрать информацию) – теперь он не помещается целиком в кэш. Проблему можно решить введением обёрточного цикла, который просто будет запускать вычисления по новой  $test\_num$  раз (да, мы так уменьшим Core Utilization, но раз мы понимаем причину ухудшения, то можно воспользоваться). Результаты профилирования представлены на рис. 15.

Elapsed Time<sup>®</sup>: 12.436s

Clockticks:	103,931,200,000	
Instructions Retired:	122,742,400,000	
CPI Rate <sup>®</sup> :	0.847	
MUX Reliability <sup>®</sup> :	0.999	
Retiring <sup>®</sup> :	51.9%	of Pipeline Slots
Front-End Bound <sup>®</sup> :	14.1%	of Pipeline Slots
Bad Speculation <sup>®</sup> :	0.0%	of Pipeline Slots
Back-End Bound <sup>®</sup> :	34.0%	of Pipeline Slots
Memory Bound <sup>®</sup> :	16.1%	of Pipeline Slots
Core Bound <sup>®</sup> :	17.9%	of Pipeline Slots
Divider <sup>®</sup> :	0.0%	of Clockticks
Port Utilization <sup>®</sup> :	36.5%	of Clockticks
Cycles of 0 Ports Utilized <sup>®</sup> :	12.2%	of Clockticks
Cycles of 1 Port Utilized <sup>®</sup> :	16.9%	of Clockticks
Cycles of 2 Ports Utilized <sup>®</sup> :	20.0%	of Clockticks
Cycles of 3+ Ports Utilized <sup>®</sup> :	46.9%	of Clockticks
Vector Capacity Usage (FPU) <sup>®</sup> :	100.0%	
Average CPU Frequency <sup>®</sup> :	1.3 GHz	
Total Thread Count:	11	
Paused Time <sup>®</sup> :	0s	

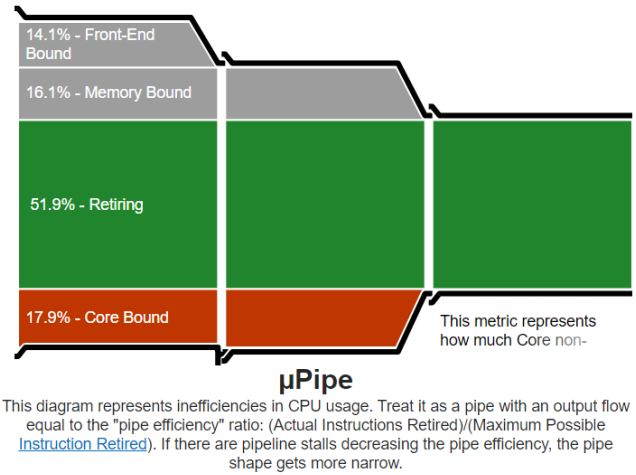


Рис. 15. Результаты профилирования векторизации (1024x1024).

Действительно, L3 Latency и CPI уменьшились, хоть и не до нуля (причины озвучены выше).

В таблице 6 приведены результаты работы на clang++ (в паре с MSVC).

Таблица 6. Результаты работы программы с clang++.

Revision	Implementation	GFLOPS	Performance percent	Relative Speedup	Absolute Speedup
1	ijk	0.28	0.129032258	1	1
2	ikj	0.47	0.216589862	1.68	2.35
3	-O2	3	1.382488479	6.38	15
4	omp	8.9	4.101382488	2.97	44.5
5	blocks	11.9	5.483870968	1.34	59.5
6	fma	34	15.66820276	2.86	170

Результаты предсказуемо похожи. Только векторизация хуже получилась – видимо, компилятор менее предназначен для работы на низком уровне.

## Выводы

В ходе данной лабораторной работы был проведён полный спектр оптимизаций реализации умножений матриц. Начинали мы с лобовой реализации в 0.2 GFLOPS и прошли следующие шаги:

- Увеличение пространственной локальности за счёт  $ikj$ .
- Использовали флаги оптимизации.
- Распараллелили вычисления с помощью OpenMP.
- Использовали блочный алгоритм.
- Векторизовали код.

Как видно из последних результатов профилирования – ещё есть куда улучшать, но даже так, мы смогли добиться производительности в 25% от максимальной (это если не учитывать, что на Hyper-Threading никогда не получить честное распараллеливание. Так – вообще 50%). И смогли увеличить изначальную реализацию в 268 раз.

Так же по результатам ЛР можно утверждать, что технология Hyper-Threading не является панацеей (и не всегда увеличивает производительность).

Многопоточность, хотя и позволяет распараллелить программу, но и добавляет расходов на диспетчеризацию самой себя (включая синхронизацию, хотя в данной ЛР все потоки работали с непересекающимися данными).

Векторизация кода сильно повышает производительность, но набор поддерживаемых инструкций аппаратно зависим, что нехорошо.

В итоге, всегда нужно смотреть на результаты программы и быть готовым сказать, в чём её потенциальные узкие места и как их исправить.



## Листинг основной программы

```

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <ratio>
#include <memory>
#include <cstdint>
#include <immintrin.h>

#define MAX_PERF 217 // GFLOPS

using namespace std::chrono;

time_point<high_resolution_clock> start, end;

void bench_start()
{
    start = high_resolution_clock::now();
}

duration<float, std::ratio<1, 1>> bench_measure()
{
    end = high_resolution_clock::now();

    // floating-point duration: no duration_cast needed
    duration<float, std::ratio<1, 1>> diff = end - start;
    return diff;
}

void calc_flops(__int64 fl_op, double diff)
{
    double gflops = fl_op / diff / 1000000000;
    std::cout << "\n DURATION:    " << diff << " s.\n";
    std::cout << " GFLOPS:        " << gflops << "\n";
    std::cout << " PERFORMANCE:  " << gflops / MAX_PERF * 100 << "%\n\n";
}

int main()
{
    std::srand(std::time(NULL));
    std::size_t m = 1024;
    std::size_t test_num = 10;
    double diff;

    float **A = new float*[m];
    float **B = new float*[m];
    float **C = new float*[m];

    for (int i=0; i < m; ++i)
    {
        A[i] = new float[m];
        B[i] = new float[m];
        C[i] = new float[m];

        for (int j=0; j < m; j++)
        {
            A[i][j] = std::rand();
            B[i][j] = std::rand();
        }
    }

    bench_start();
    for (int i=0; i < m; ++i)
    {
        for (int j=0; j < m; ++j)
        {
            for (int k=0; k < m; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

```

```

    }
}
diff = bench_measure().count();

calc_flops((__int64)m*m*m*2, diff);

for (int i=0; i < m; ++i)
{
    delete []A[i];
    delete []B[i];
    delete []C[i];
}

delete []A;
delete []B;
delete []C;

return 0;
}

```

### Листинг последней оптимизации (FMA3)

```

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <ratio>
#include <memory>
#include <cstdint>
#include <immintrin.h>
#include <windows.h>

#define MAX_PERF 217 // GFLOPS

using namespace std::chrono;

time_point<high_resolution_clock> start, end;

void bench_start()
{
    start = high_resolution_clock::now();
}

duration<float, std::ratio<1, 1>> bench_measure()
{
    end = high_resolution_clock::now();

    // floating-point duration: no duration_cast needed
    duration<float, std::ratio<1, 1>> diff = end - start;
    return diff;
}

void calc_flops(__int64 fl_op, double diff)
{
    double gflops = fl_op / diff / (__int64)1000000000;
    std::cout << "\n DURATION:      " << diff << " s.\n";
    std::cout << " GFLOPS:          " << gflops << "\n";
    std::cout << " PERFORMANCE:    " << gflops / MAX_PERF * 100 << "%\n\n";
}

void calc_block(float **A, float **B, float **C, int ih, int jh, int kh, int block_s)
{
    for (int il=ih; il < ih+block_s; il++)
    {
        for (int kl=kh; kl < kh+block_s; kl++)
        {
            __m256 a = _mm256_set1_ps(A[il][kl]);
            for (int jl=jh; jl < jh+block_s; jl+=8)
            {
                _mm256_storeu_ps(
                    C[il]+jl,
                    _mm256_fmadd_ps(
                        a,
                        _mm256_loadu_ps(B[kl] + jl),
                        _mm256_loadu_ps(C[il] + jl)
                    )
                );
            }
        }
    }
}

```

```

    );
    }
}

int main()
{
    std::srand(std::time(NULL));
    std::size_t m = 1024*2;
    std::size_t test_num = 1;
    std::size_t block_s = 64;
    double diff;

    float **A = new float*[m];
    float **B = new float*[m];
    float **C = new float*[m];

    for (int i=0; i < m; ++i)
    {
        A[i] = new float[m];
        B[i] = new float[m];
        C[i] = new float[m];
    }

    std::cout << "\nBLOCK: " << block_s;

    bench_start();
    for (int test = 0; test < test_num; test++)
    {
        for (int i=0; i < m; ++i)
            for (int j=0; j < m; j++)
            {
                A[i][j] = std::rand();
                B[i][j] = std::rand();
                C[i][j] = 0;
            }

        #pragma omp parallel for schedule(static)
        for (int ih=0; ih < m; ih+=block_s)
        {
            #pragma omp parallel for schedule(static)
            for (int jh=0; jh < m; jh+=block_s)
                for (int kh=0; kh < m; kh+=block_s)
                    calc_block(A, B, C, ih, jh, kh, block_s);
        }
        diff = bench_measure().count() / test_num;

        calc_flops((__int64)m*m*m*2, diff);

        for (int i=0; i < m; ++i)
        {
            delete []A[i];
            delete []B[i];
            delete []C[i];
        }

        delete []A;
        delete []B;
        delete []C;

        return 0;
    }
}

```