

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего
образования

**«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)**

Институт информационных технологий, математики и механики

Кафедра математического обеспечения и суперкомпьютерных технологий

Направление подготовки: «Прикладная математика и информатика»
Магистерская программа: «Вычислительные методы и суперкомпьютерные технологии»

Отчет по технологической практике

«Исследование производительности oneAPI в задаче умножения матриц»

Выполнил:

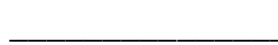
студент группы 381903-3м



Панов А. А.

Проверил:

к.т.н., доцент каф. МОСТ ИИТММ



Месеров И. Б.

Нижний Новгород
2021

Содержание

| | |
|--|----|
| Содержание | 2 |
| Введение..... | 3 |
| 1. Программно-аппаратное окружение | 4 |
| 2. Реализации умножения матриц на CPU | 5 |
| 2.1 Базовая версия | 5 |
| 2.2 Блочная версия | 7 |
| 2.3 Оптимизация блочной версии..... | 9 |
| 2.4 Сравнение производительности CPU версий | 10 |
| 3. Реализация на встроенной GPU с помощью средств OneAPI | 12 |
| 3.1 Наивная версия | 12 |
| 3.2 Базовая версия | 12 |
| 3.3 Блочная версия | 13 |
| 3.4 Оптимизация блочной версии..... | 15 |
| 3.5 Сравнение производительности GPU версий..... | 18 |
| 4. Выводы | 21 |
| 4.1 Краткие выводы по CPU версиям | 21 |
| 4.2 Краткие выводы по GPU версиям..... | 21 |
| 4.3 Производительность MKL на CPU и GPU | 22 |
| Список литературы | 23 |
| Приложение..... | 24 |

Введение

В работе рассматриваются алгоритмы параллельного умножения на матриц на CPU и GPU, анализируются их производительность, достоинства и недостатки методов. Для простоты рассматривается задача умножения квадратных матриц размера $N \times N$ с типом данных float.

Реализация алгоритма умножения на GPU выполняется с помощью набора инструментов Intel OneAPI Toolkits. В состав набора входит компилятор нового языка Data Parallel C++, набор библиотек для API-программирования и комплект средств для анализа и отладки приложений. DPC++ — это развитие языка C++, включающее в себя SYCL, позволяющий использовать C++ код для любой архитектуры, производя при этом доступные под конкретную платформу оптимизации.

1. Программно-аппаратное окружение

- Компилятор:
Intel C++ Compiler 19.2
- Основные ключи компилятора:
/Qopenmp /O2 /arch:CORE-AVX2
- ОС:
Windows 10 Pro
- Процессор:
Intel Core I5 9600K, частота зафиксирована на уровне 4.8 ГГц по всем ядрам. L3 – 9 Мб, L2 – 256 Кб, L1 – 64 Кб.
- Интегрированная графика:
Intel UHD Graphics 630, частота 1150 МГц
- Память:
Двухканальная DDR4, 2x8 Гб, 3100 МГц

Корректность работы умножения проверялось с помощью сравнения результатов с базовым алгоритмом.

2. Реализации умножения матриц на CPU

2.1 Базовая версия

Рассмотрим простейшую реализацию умножения матриц размера $N \times N$ на CPU. В данной реализации используется правильный порядок циклов i, k, j . Цикл параллелится по номеру итерации i с помощью технологии OpenMP.

```
// Алгоритм 1. Базовый параллельный алгоритм умножения матриц
1. #pragma omp parallel for
2.   for i in N:
3.       for k in N:
4.           for j in N:
5.               C[i*N+j] += A[i*N+k]*B[k*N+j]
```

Правильный порядок циклов позволяет идти последовательно по памяти. Вычисление будет эффективным, если:

- разместить в кэше (L1) i -ую строку матрицы C;
- разместить в кэше (L1) k -ую строку матрицы C;
- разместить в кэше (L1) (i, k) -ый элемент матрицы A;
- разместить в кэше (L1, L2, L3) M строк матрицы B, начиная с $k + 1$ -ой, и M элементов из матрицы A, начиная с $k + 1$ -ого (M должно быть как можно больше, но все строки, скорее всего, не уместятся).

Это позволит:

- выполнить цикл по j над данными в кэше L1 (размер одной строки из 4032 float – 15.75 Кб, размер L1 – 64 Кб);
- выполнить часть цикла по k над данными из кэша (часть строк из матрицы B будет храниться в L1, L2 и L3).

После того, как данные в кэше закончатся, придется подгружать данные из ОЗУ.

Попробуем оценить количество умножений/сложений, которые выполнит процессор на данных из кэша. Для простоты зафиксируем размеры матрицы и кэшей:

- $N = 4032$
- $L3 = 1.5$ Мб в расчете на одно ядро
- $L2 = 256$ кб
- $L1 = 64$ кб

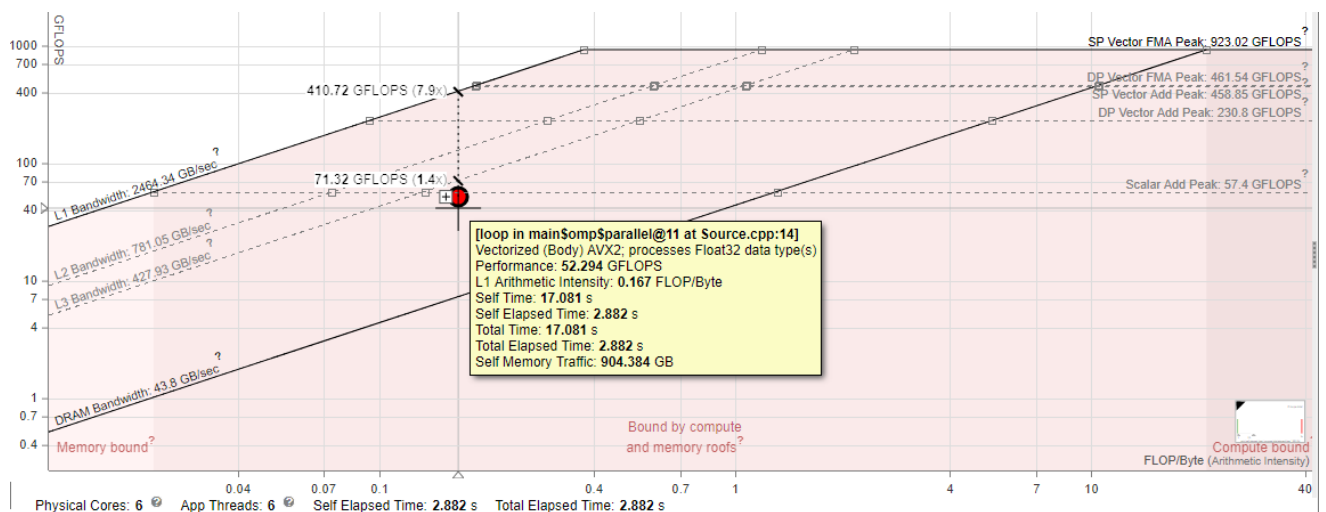
В соответствии с описанным выше принципом распределения данных по кэшам получим:

- В кэше L1 – одна i -ая строка матрицы C строка + три строки B начиная с k -ой + три элемента из матрицы A , начиная с (i, k) -ого.
- В кэше L3 + L2 уместилось 113 строк из матрицы B и ещё 113 элементов из A .

В таком случае при использовании только данных из кэшей получится выполнить $116N$ умножений/сложений (без учета операций над индексами):

- Все данные для внутреннего цикла лежат в кэшах, получаем N умножений/сложений.
- За счет 116 строк матрицы B и 116 элементов из A можно выполнить внутренний цикл 116 раз, на имеющихся данных, итого $116N$.

Построив roofline, можно убедиться, что полученная производительность значительно выше пропускной способности памяти, а это значит, что кэширование уже некоторым образом работает.

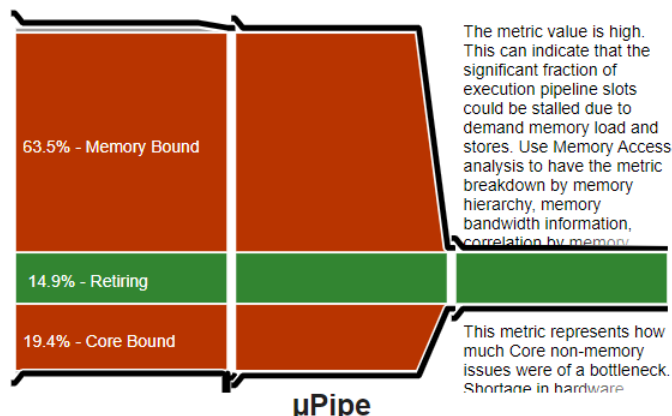


Roofline базового алгоритма на CPU.

Через VTune можно увидеть, что вычислительная мощность CPU существенно ограничена пропускной способностью ОЗУ и L3.

Elapsed Time[®]: 2.723s

| | |
|--------------------------------------|-------------------------|
| Clockticks: | 74,270,100,000 |
| Instructions Retired: | 37,000,000,000 |
| CPI Rate [®] : | 2.007 |
| MUX Reliability [®] : | 0.991 |
| Retiring [®] : | 14.9% of Pipeline Slots |
| Front-End Bound [®] : | 1.5% of Pipeline Slots |
| Bad Speculation [®] : | 0.8% of Pipeline Slots |
| Back-End Bound [®] : | 82.9% of Pipeline Slots |
| Memory Bound [®] : | 63.5% of Pipeline Slots |
| L1 Bound [®] : | 0.0% of Clockticks |
| L2 Bound [®] : | 0.0% of Clockticks |
| L3 Bound [®] : | 38.1% of Clockticks |
| Contested Accesses [®] : | 0.1% of Clockticks |
| Data Sharing [®] : | 48.9% of Clockticks |
| L3 Latency [®] : | 100.0% of Clockticks |
| SQ Full [®] : | 21.7% of Clockticks |
| DRAM Bound [®] : | 22.0% of Clockticks |
| Memory Bandwidth [®] : | 84.8% of Clockticks |
| Memory Latency [®] : | 8.7% of Clockticks |
| Store Bound [®] : | 0.0% of Clockticks |
| Core Bound [®] : | 19.4% of Pipeline Slots |
| Average CPU Frequency [®] : | 4.8 GHz |
| Total Thread Count: | 9 |
| Paused Time [®] : | 0s |



This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Анализ базового алгоритма на CPU через VTune.

2.2 Блочная версия

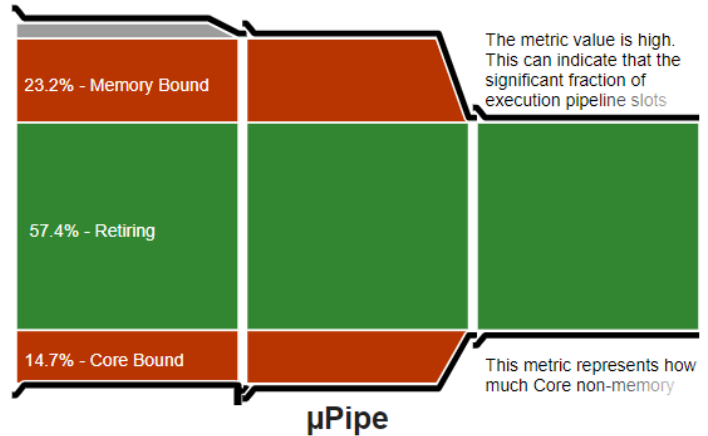
В предыдущей версии в кэшах уместилось всего 117 строк (116 из В, 1 из С). В блочной версии алгоритма в кэшах выгодно размещать строки из всех трех матриц. В данном случае можно уместить по 39 строк из матрицы А, В и С. При размере блока 39 x 39 на этих данных можно выполнить $39 \times 39 \times 39 \times \left(\frac{N}{39}\right) = 1521N$ умножений/сложений. То есть получаем в 13 раз больше вычислений, которые можно выполнить на данных из кэша.

// Алгоритм 2. Блочный параллельный алгоритм умножения матриц

```
1. #pragma omp parallel for
2.   for ib in N/bs_r:
3.     for kb in N/bs_c:
4.       for jb in N/bs_c:
5.         int start = jb*bs_c
6.         for i = ib*bs_r in ib*bs_r+bs_r:
7.           float* ls = C+i*N+st
8.           for k = kb*bs_c in kb*bs_c+bs_c:
9.             float AA = A[i*N+k]
10.            float* pb = B+k*N+st
11.            for j in bs_c:
12.              float lc[j] += AA*pb[j]
```

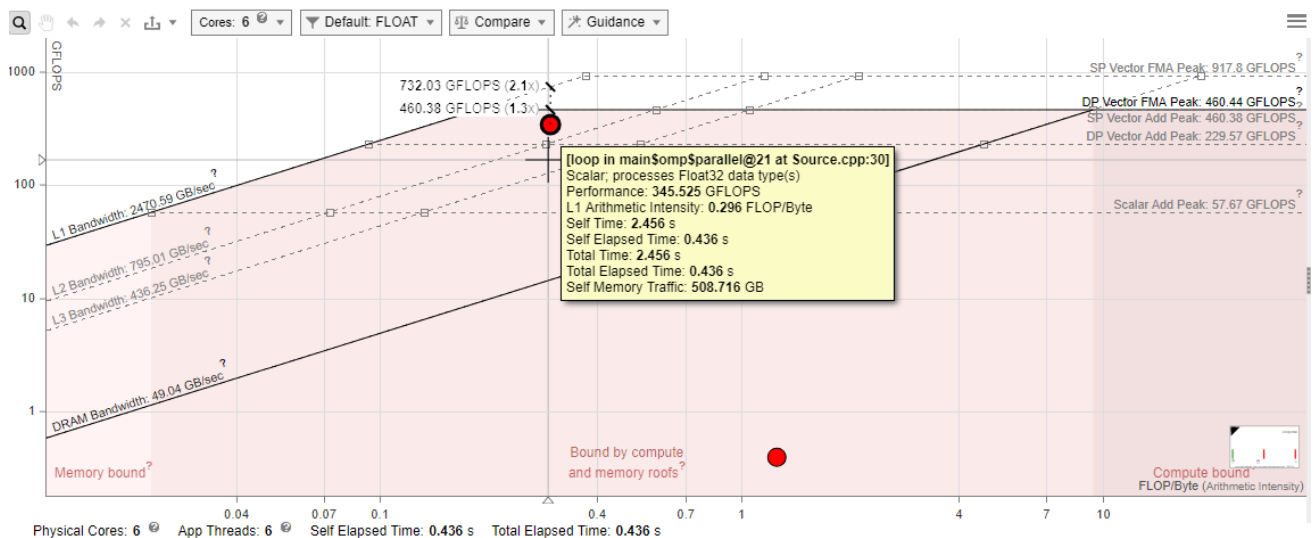
Elapsed Time[®]: 0.603s

| | |
|--------------------------------------|-------------------------|
| Clockticks: | 15,395,700,000 |
| Instructions Retired: | 27,124,700,000 |
| CPI Rate [®] : | 0.568 |
| MUX Reliability [®] : | 0.992 |
| Retiring [®] : | 57.4% of Pipeline Slots |
| Front-End Bound [®] : | 4.2% of Pipeline Slots |
| Bad Speculation [®] : | 0.4% of Pipeline Slots |
| Back-End Bound [®] : | 37.9% of Pipeline Slots |
| Memory Bound [®] : | 23.2% of Pipeline Slots |
| L1 Bound [®] : | 2.1% of Clockticks |
| L2 Bound [®] : | 0.3% of Clockticks |
| L3 Bound [®] : | 2.0% of Clockticks |
| DRAM Bound [®] : | 16.1% of Clockticks |
| Memory Bandwidth [®] : | 28.0% of Clockticks |
| Memory Latency [®] : | 25.9% of Clockticks |
| Store Bound [®] : | 0.3% of Clockticks |
| Core Bound [®] : | 14.7% of Pipeline Slots |
| Average CPU Frequency [®] : | 4.8 GHz |
| Total Thread Count: | 9 |
| Paused Time [®] : | 0s |



This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Анализ блочного алгоритма на CPU через VTune.



Roofline блочного алгоритма на CPU.

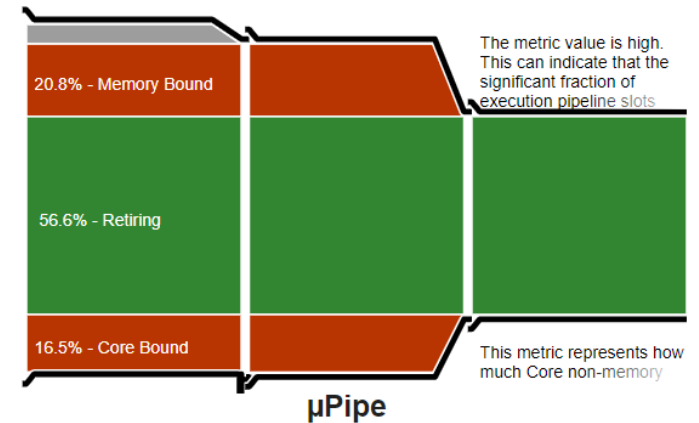
Данный вариант алгоритма практически не ограничивается пропускной способностью ОЗУ. Узким местом становится работа с кэшами.

2.3 Оптимизация блочной версии

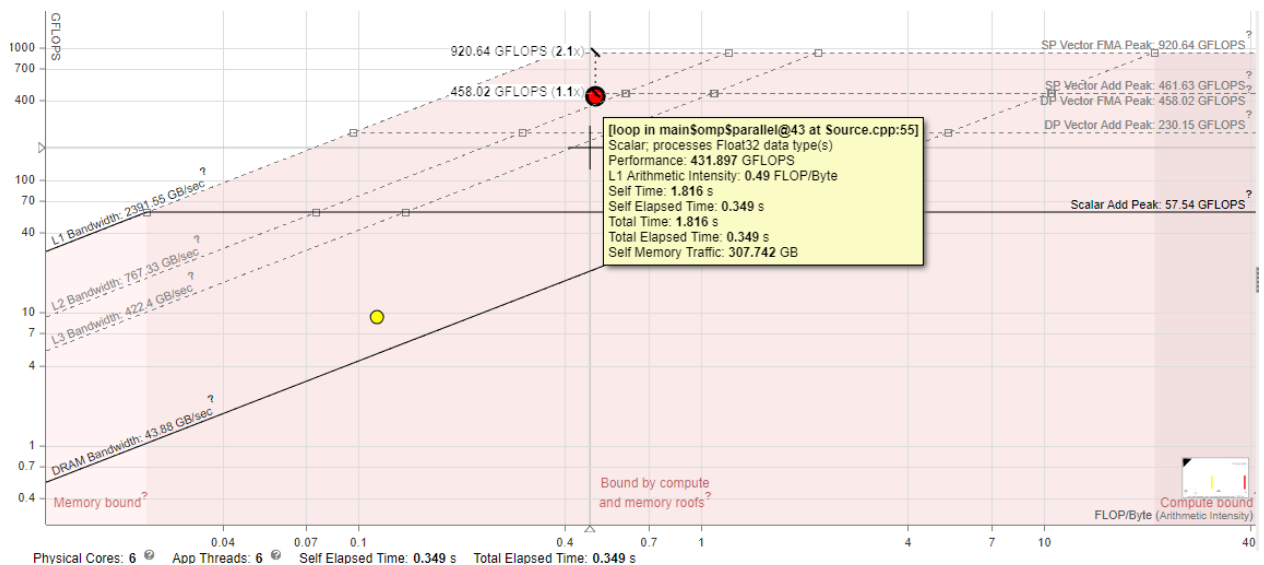
Вместо того, чтобы записывать результаты сразу в строку матрицы C, можно сначала записывать данные в локальный массив (код алгоритма есть в приложении), что позволяет лучше работать с кэшами и даёт ускорение в 1.3 раза.

Elapsed Time[®]: 0.520s

| | |
|--------------------------------------|-------------------------|
| Clockticks: | 13,175,700,000 |
| Instructions Retired: | 21,300,900,000 |
| CPI Rate [®] : | 0.619 |
| MUX Reliability [®] : | 0.992 |
| Retiring [®] : | 56.6% of Pipeline Slots |
| Front-End Bound [®] : | 5.6% of Pipeline Slots |
| Bad Speculation [®] : | 0.5% of Pipeline Slots |
| Back-End Bound [®] : | 37.3% of Pipeline Slots |
| Memory Bound [®] : | 20.8% of Pipeline Slots |
| L1 Bound [®] : | 2.2% of Clockticks |
| L2 Bound [®] : | 0.0% of Clockticks |
| L3 Bound [®] : | 2.3% of Clockticks |
| DRAM Bound [®] : | 15.0% of Clockticks |
| Memory Bandwidth [®] : | 28.3% of Clockticks |
| Memory Latency [®] : | 30.0% of Clockticks |
| Store Bound [®] : | 0.1% of Clockticks |
| Core Bound [®] : | 16.5% of Pipeline Slots |
| Average CPU Frequency [®] : | 4.8 GHz |
| Total Thread Count: | 9 |
| Paused Time [®] : | 0s |



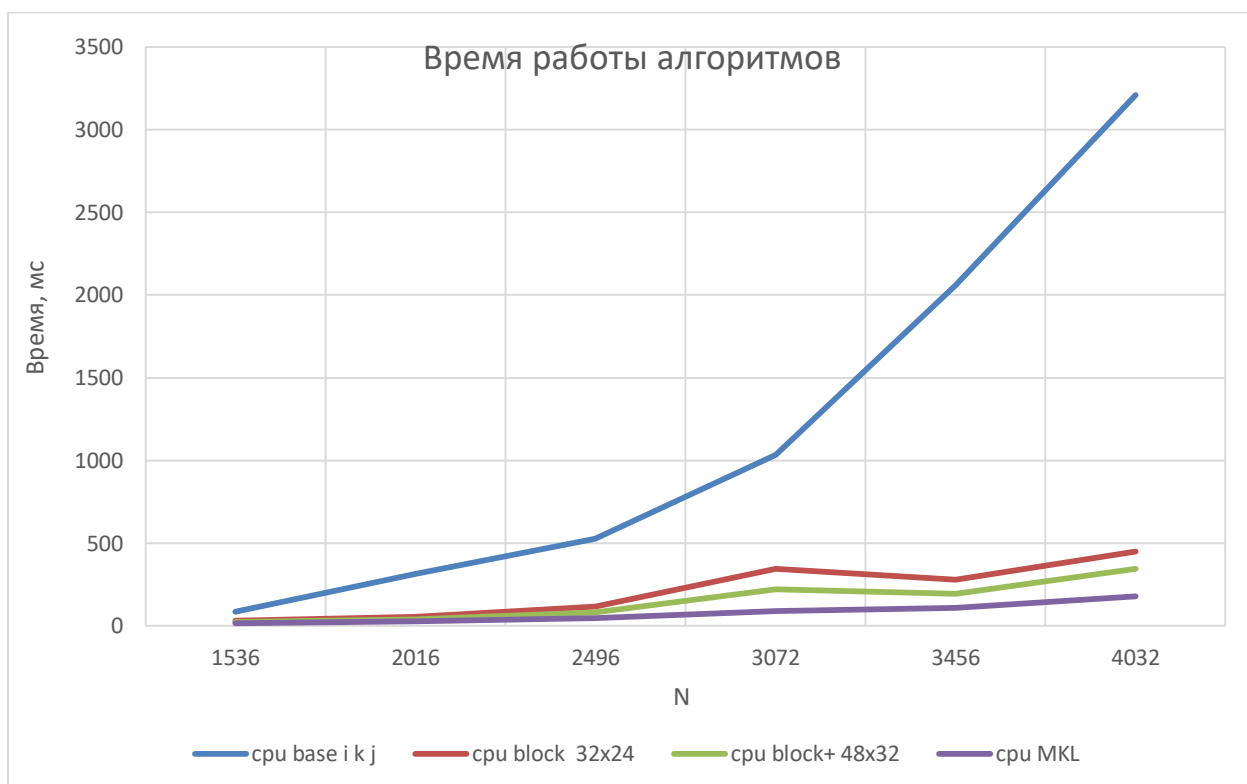
Анализ блочного+ алгоритма на CPU через VTune.



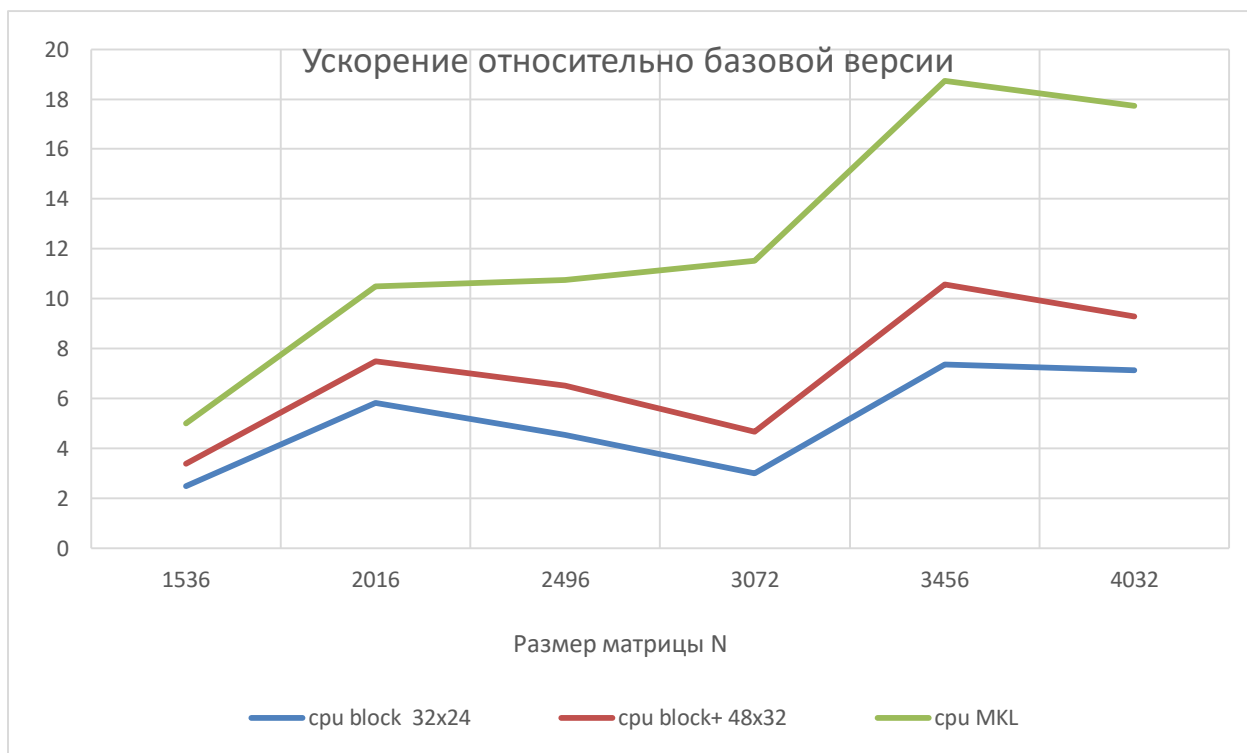
Roofline блочного+ алгоритма на CPU.

Данный вариант алгоритма также практически не ограничивается пропускной способностью ОЗУ. Узким местом остаётся работа с кэшами. По итогу удалось достигнуть 50% от максимальной производительности процессора.

2.4 Сравнение производительности CPU версий



Сравнение производительности CPU версий.



Ускорение относительно базовой версии.

МКL версия практически вдвое производительней блочной версии, таким образом, она задействует практически 100% вычислительной мощности CPU. К сожалению, блочная версия имеет ещё один недостаток: при увеличении N ухудшается работа с кэшами, падает вычислительная интенсивность. Эти недостатки можно исправить, улучшив работу с L3/L2/L1 кэшами. В работе [3] можно найти такую схему работы блочного умножения:

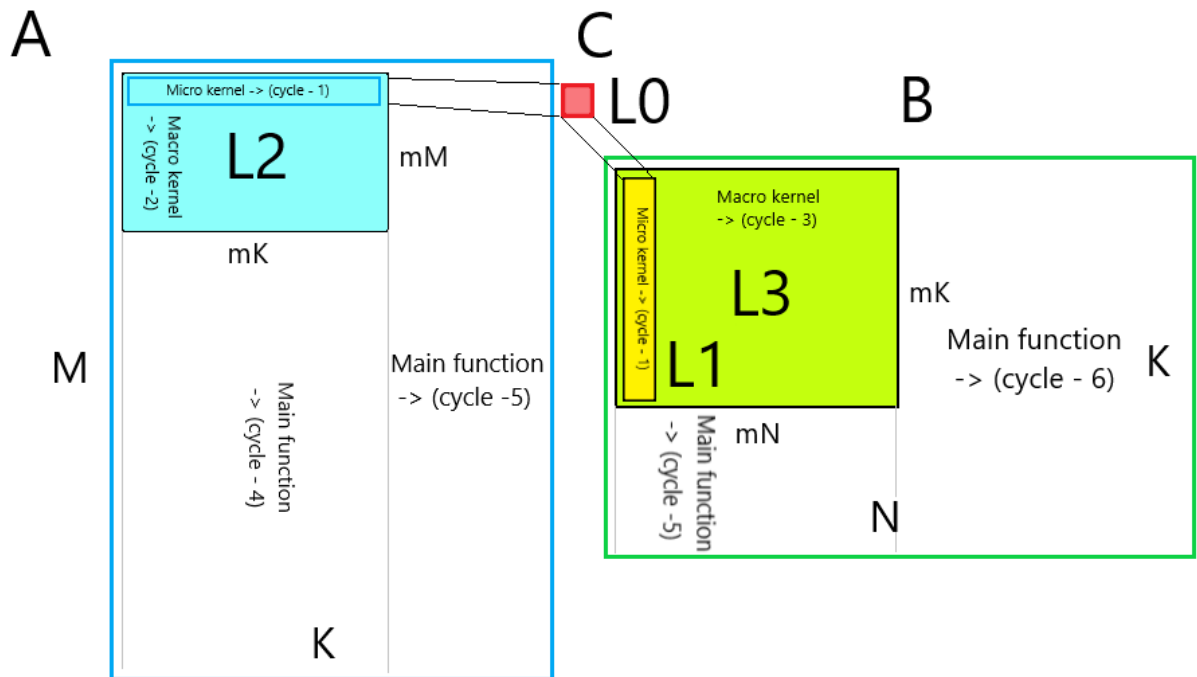


Схема умножения матриц, источник: [3].

В дальнейшем такая схема будет реализована и её можно будет сравнить с текущими версиями.

3. Реализация на встроенной GPU с помощью средств OneAPI

3.1 Наивная версия

В данной версии создаётся рабочее пространство с размерами как у матрицы C ($N \times N$). Так как количество «поток» совпадает с размерностью рабочего пространства, можно реализовать схему, по которой каждый «поток» GPU считает один элемент из матрицы C .

```
// Алгоритм 3. Наивный параллельный алгоритм умножения матриц на GPU
1.h.parallel_for(range(N, N), [=](auto index)):
2.   row = index[0]
3.   col = index[1]
4.   for i in N:
5.       C[row][col] += A[row][i]*B[i][col]
```

3.2 Базовая версия

Один из недостатков в наивной версии в том, что операция «+=» выполняется с элементами из массива в глобальной памяти. Компилятор в данном случае не будет создавать локальную переменную (как бы это произошло на CPU), а будет делать запись в глобальную память, поэтому нужно создать локальную переменную и записывать результат в неё.

```
1.// Алгоритм 4. Базовый параллельный алгоритм умножения матриц на GPU
2.h.parallel_for(range(N, N), [=](auto index)):
3.   row = index[0]
4.   col = index[1]
5.   sum = 0.0f
6.   for i in N:
7.       sum += A[row][i]*B[i][col]
8.   C[row][col] = sum
```

Такое элементарное улучшение ускоряет умножение матриц на 10%. Всего в данном алгоритме каждый поток делает $2N$ чтений из глобальной памяти. Поток N^2 , значит, общее количество чтений из глобальной памяти $2N^3$.

3.3 Блочная версия

На GPU потоки можно группировать по блокам, при этом все потоки в блоке будут выполняться на одном мультипроцессоре (SM – streaming multiprocessor). У каждого SM блока есть своя память, к которой имеют доступ потоки данного мультипроцессора. Также имеется возможность синхронизировать потоки из одного блока. Всё это позволяет реализовать более совершенный алгоритм блочного умножения матриц.

```
1.// Алгоритм 5. Блочный параллельный алгоритм умножения матриц на GPU
2.range<2> matrix_range{ N, N }
3.range<2> tile_range{ tile_size, tile_size }
4.h.parallel_for(matrix_range, tile_range), [=](auto ind):
5.    row = ind.get_local_id(0)
6.    col = ind.get_local_id(1)
7.    globalRow = tile_size * ind.get_group(0) + row
8.    globalCol = tile_size * ind.get_group(1) + col
9.    numTiles = N / tile_size
10.   sum = 0.0f
11.   for t = 0 in numTiles:
12.       aTile[row][col] = a[globalRow][tile_size * t + col]
13.       bTile[row][col] = b[tile_size * t + row][globalCol]
14.       ind.barrier(cl::sycl::access::fence_space::local_space)
15.       for k in tile_size:
16.           sum += aTile[row][k] * bTile[k][col]
17.       ind.barrier(cl::sycl::access::fence_space::local_space)
18.   c[globalRow][globalCol] = sum
```

Блочный алгоритм почти в 2.2 раза быстрее обычного базового алгоритма.

Рассмотрим работу блочного алгоритма на примере матриц с $N = 4$, и $tile_size = 2$, где $tile_size$ – размер блока. Внутри блока GPU с индексом (i, j) происходит следующее:

- В матрице A идёт работа с блоком $(i, 0)$, в матрице B с блоком $(0, j)$.
- Поток в локальную память подгружает 1 элемент из матрицы A и матрицы B.
- Поток ожидает, когда другие потоки из блока.
- Поток выполняется скалярное умножение двух «подвекторов» из блока.

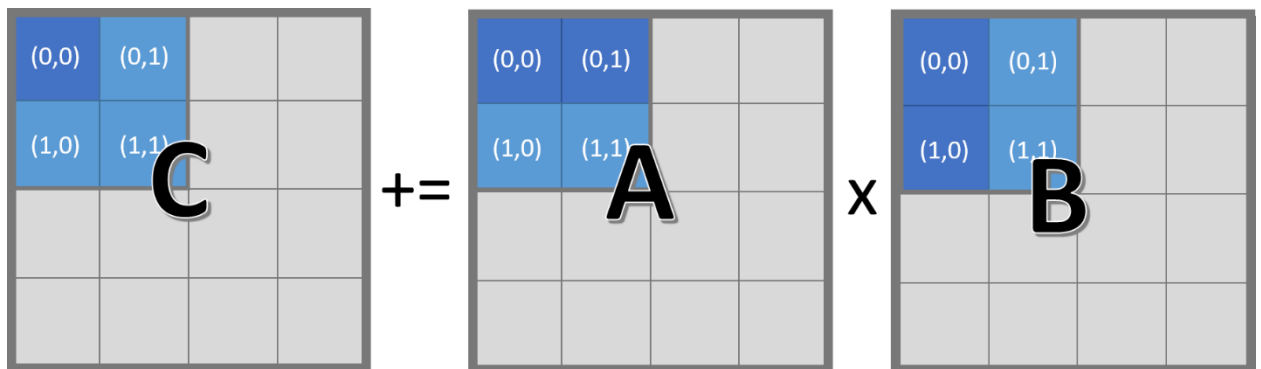


Иллюстрация работы блочного алгоритма умножения матриц на GPU (часть 1). В качестве примера рассмотрен блок GPU $(0,0)$. Внутри блока указаны локальные индексы потоков.

- Поток ожидает другие потоки из блока.
- В матрице A идёт работа с блоком $(i, 1)$, в матрице B с блоком $(1, j)$.
- Поток в локальную память подгружает 1 элемент из матрицы A и матрицы B.
- Поток ожидает, когда другие потоки из блока.
- Поток выполняется скалярное умножение двух «подвекторов» из блока.

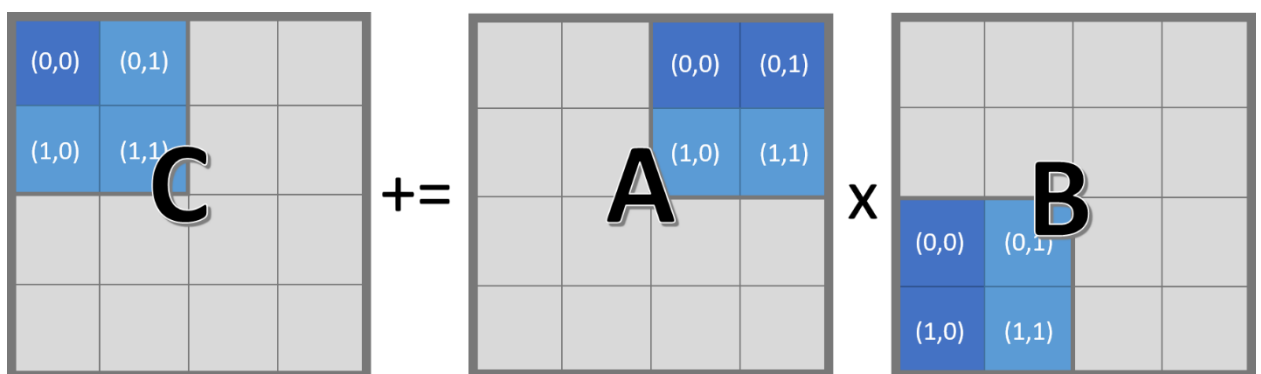


Иллюстрация работы блочного алгоритма умножения матриц на GPU (часть 2). В качестве примера рассмотрен блок GPU $(0,0)$. Внутри блока указаны локальные индексы потоков.

- После $N/tile_size$ шагов все потоки внутри каждого блока вычислили «свой» элемент из матрицы C.

Всего в данном алгоритме каждый поток делает $\frac{2N}{tile_size}$ чтений из глобальной памяти и $2N$ чтений из локальной. Общее количество потоков N^2 , значит количество чтений из глобальной памяти $\frac{2N^3}{tile_size}$ и $2N^3$ из локальной. Обозначим время доступа к глобальной памяти, как $time_{global}$, а к локальной как $time_{local}$. Тогда теоритическое ускорение относительно базовой версии можно рассчитать так:

$$\frac{time_{global} \times 2N^3}{time_{global} \times 2N^3 / tile_size + 2N^3 time_{local}} = \frac{time_{global}}{time_{global} / tile_size + time_{local}}.$$

3.4 Оптимизация блочной версии

Блочную версию можно улучшить, используя следующие приёмы.

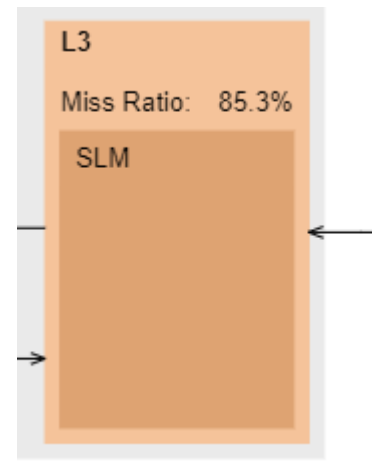
- Подбор количества потоков для получения оптимальной нагрузки на каждый поток. Для данной задачи оптимально, чтобы каждый поток вычислял значения четырёх элементов из матрицы C (вместо одного в первоначальной версии).
- Разворачивание циклов с помощью прагмы «`unroll(N)`». Прагма уменьшает длину цикла в N раз, позволяя на каждой итерации выполнить в N раз больше инструкций. Был развернут внутренний цикл по блоку с N=2:

```
#pragma unroll(2)
for (int k = 0; k < tile_size; k++)
{
    sum1 += aTile[row*tile_size+k] * bTile[k*tile_size+col];
    sum2 += aTile[(row + tmp)*tile_size+k] * bTile[k*tile_size+col];
    sum3 += aTile[(row + 2 * tmp)*tile_size+k] * bTile[k*tile_size+col];
    sum4 += aTile[(row + 3 * tmp)*tile_size+k] * bTile[k*tile_size+col];
}
```

Данные улучшения ускоряют вычисления в 3.2 раза относительно базовой версии алгоритма.

Elapsed Time[®]: 8.835s 📄

| | |
|---|----------------|
| ⌚ CPU Time[®]: | 9.517s |
| ✓ EU Array[®]: | |
| Active [®] : | 81.2% |
| Stalled [®] : | 17.9% |
| Idle [®] : | 0.8% |
| Computing Threads Started [®] : | 381,025 |
| L3 Bandwidth, GB/sec [®] : | 40.826 |
| L3 Sampler Bandwidth, GB/sec [®] : | 0.004 |
| L3 <-> GTI Total Bandwidth, GB/sec: | 1.606 |
| ✓ Shared Local Memory Bandwidth, GB/sec[®]: | |
| Read [®] : | 37.094 |
| Write [®] : | 1.855 |
| ✓ GPU Memory Bandwidth, GB/sec[®]: | |
| Read [®] : | 1.744 |
| Write [®] : | 0.023 |
| ✓ Sampler[®]: | |
| Busy [®] : | 0.1% |
| Bottleneck [®] : | 0.0% |
| GPU L3 Misses, Misses/sec [®] : | 25,092,705.482 |
| GPU Barriers: | 64,012,032.000 |
| GPU Atomics: | 0.000 |
| ⌚ GPU Time: | 1.930s |
| Total Thread Count: | 19 |
| Paused Time [®] : | 0s |

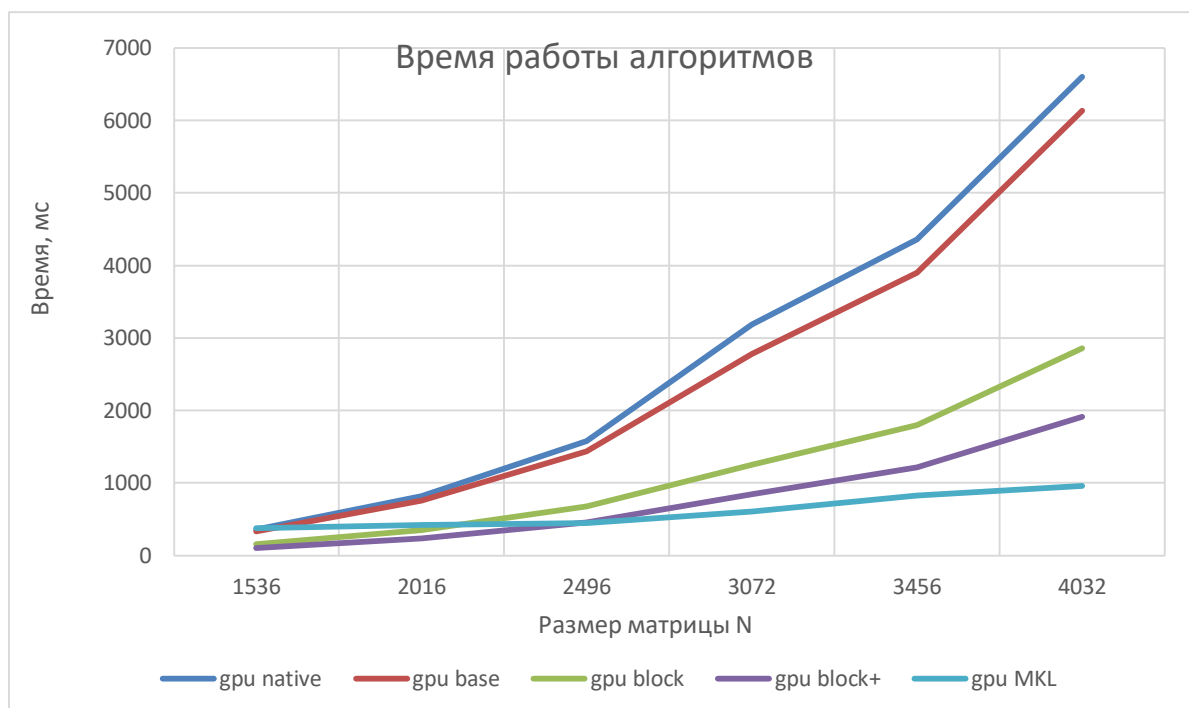


Анализ блочного+ алгоритма на GPU через VTune.

VTune показывает большое количество синхронизаций и кэш промахов. К сожалению, VTune на данный момент не может так же, как для CPU, явно показать узкое место при работе с GPU.

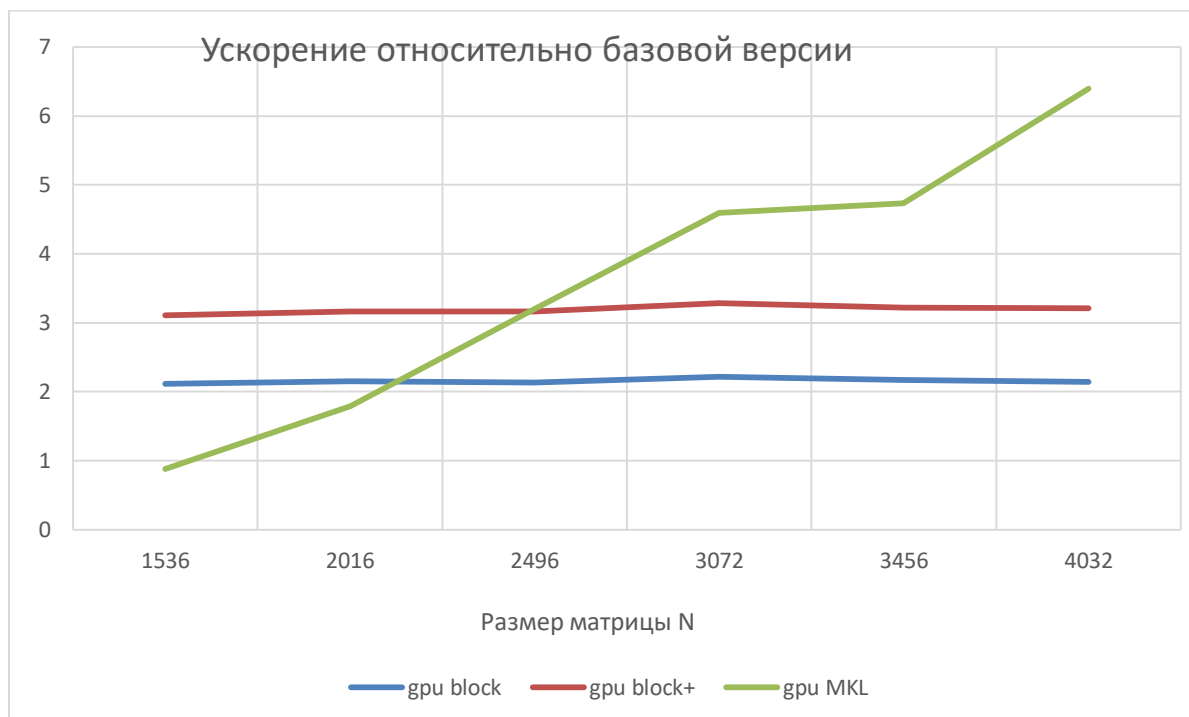
3.5 Сравнение производительности GPU версий

Улучшенная блочная версия в 3.2 раза быстрее базовой.



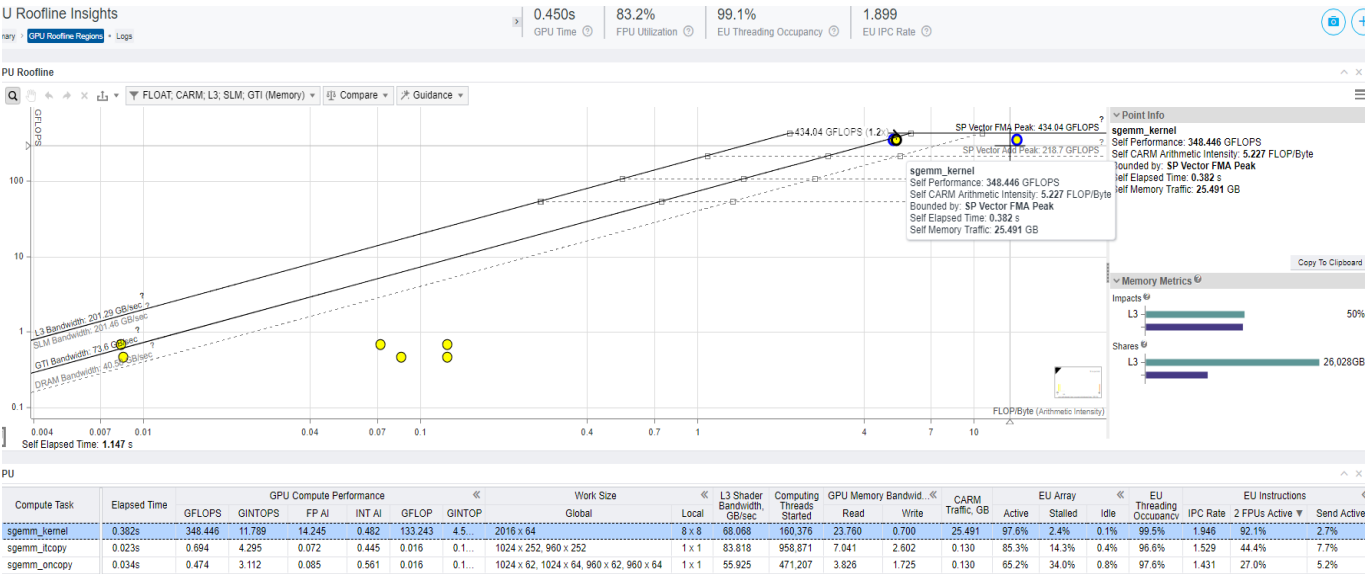
Сравнение производительности GPU версий

MKL версия отстаёт по времени работы при N меньшем 2496, но с увеличением размера матрицы, MKL всё больше обгоняет блочную версию.




Ускорение относительно базовой версии

При этом на больших размерах матрицы MKL может задействовать практически 100% вычислительных возможностей GPU (Advisor оценивает максимальные вычислительные возможности GPU в 430 GFLOPS, что совпадает с данными из открытых источников).



Roofline MKL алгоритма на GPU.

| | |
|--|----------------|
| Elapsed Time[®] : 5.948s  | |
| ➤ CPU Time[®] : | 6.780s |
| ✓ EU Array[®] : | |
| Active [®] : | 91.5% |
| Stalled [®] : | 6.5% |
| Idle [®] : | 1.9% |
| Computing Threads Started [®] : | 1,811,880 |
| L3 Bandwidth, GB/sec [®] : | 5.117 |
| L3 Sampler Bandwidth, GB/sec [®] : | 0.006 |
| L3 <-> GTI Total Bandwidth, GB/sec: | 1.631 |
| ✓ Shared Local Memory Bandwidth, GB/sec[®] : | |
| Read [®] : | 0.000 |
| Write [®] : | 0.000 |
| ✓ GPU Memory Bandwidth, GB/sec[®] : | |
| Read [®] : | 1.549 |
| Write [®] : | 0.097 |
| ✓ Sampler[®] : | |
| Busy [®] : | 0.3% |
| Bottleneck [®] : | 0.0% |
| GPU L3 Misses, Misses/sec [®] : | 25,477,345.145 |
| GPU Barriers: | 0.000 |
| GPU Atomics: | 0.000 |
| ➤ GPU Time: | 0.448s |
| Total Thread Count: | 18 |
| Paused Time [®] : | 0s |

Анализ блочного+ алгоритма на GPU через VTune.

В MKL версии VTune не смог зафиксировать работу с общей памятью и наличие барьеров (то есть синхронизации потоков GPU). Обмен данными с L3 был менее интенсивным (bandwidth 70 GB/sec), чем у блочной версии. Количество кэш промахов почти не изменилось.

4. Выводы

4.1 Краткие выводы по CPU версиям

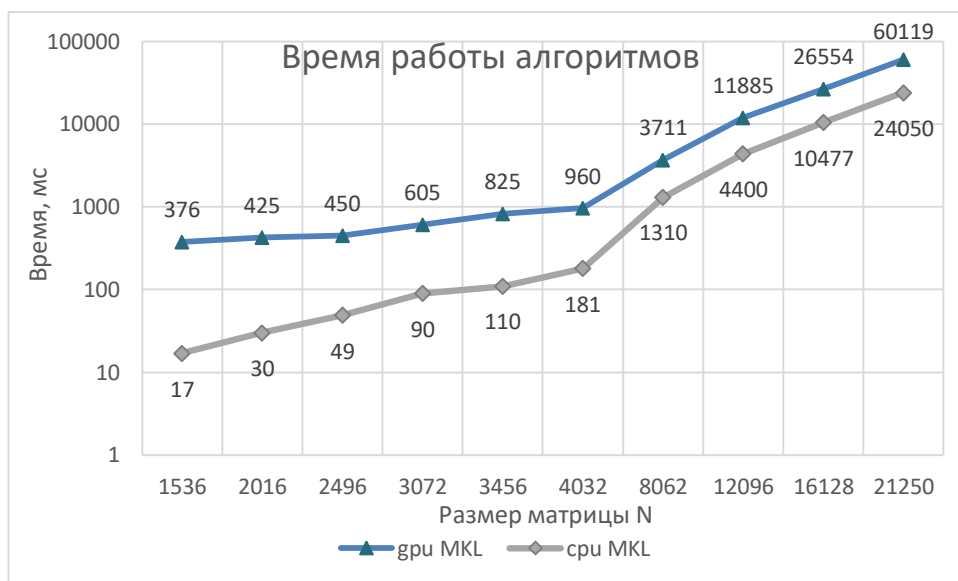
- Блочная версия в 3.5-10 (в зависимости от N) раз быстрее базовой.
- MKL практически вдвое производительней блочной версии, таким образом MKL позволяет задействовать практически 100% вычислительной мощности процессора.
- Блочная версия имеет ещё один недостаток: при увеличении N ухудшается работа с кэшами, падает вычислительная интенсивность. Эти недостатки можно исправить, улучшив работу с L3/L2/L1 кэшами.

4.2 Краткие выводы по GPU версиям

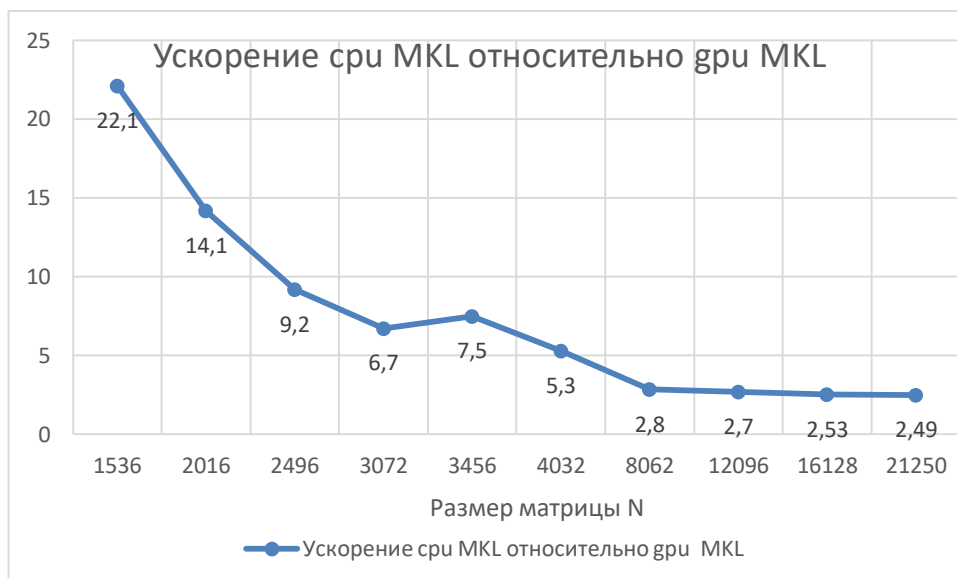
- Блочная версия в 3.2 раза быстрее базовой.
- MKL версия отстаёт по времени работы при N меньшем 2496, но с увеличением размера матрицы MKL всё больше обгоняет блочную версию.
- В текущей версии VTune работает на GPU хуже чем на CPU:
 - Сложно понять почему производительность не достигает пиковых значений. Недостаточно информации о работе подсистемы памяти (глобальная/общая память GPU). Нет данных, о том сколько времени идут вычисления, а сколько времени ядра GPU ожидают данные.
 - MKL версия согласно показаниям VTune не имеет синхронизаций и не использует общую память, что выглядит странно.

4.3 Производительность MKL на CPU и GPU

Исходя из данных Adviser и открытых источников, встроенная графика Intel UHD 630 имеет производительность в FP32 около 430 GFLOPS. Данных по CPU Intel 9600K в открытых источниках нет, но Adviser показывает 920 GFLOPS, что в 2.14 раз больше. Если при $N = 4032$ MKL на GPU работает в 5.3 раза медленней, то при увеличении N разница сокращается. При $N = 16128$ MKL на GPU уже всего в 2.5 раза медленней, что практически соответствует отставанию встроенной графики по пиковой производительности.



Время работы MKL на CPU и GPU



Отношение времени работы GPU MKL к CPU MKL

Список литературы

- Ватутин Э. И., Мартынов И. А., Титов В. С. Оценка реальной производительности современных видеокарт с поддержкой технологии CUDA в задаче умножения матриц //Известия Юго-Западного государственного университета. Серия: Управление, вычислительная техника, информатика. Медицинское приборостроение. – 2014. – №. 2. – С. 8-17.
- Reinders J. et al. Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL. – Springer Nature, 2021. – С. 548.
- «Умножение матриц: эффективная реализация шаг за шагом»
<https://habr.com/ru/post/359272/>

Приложение

```
void gpu_mullt_block4(const float* a_host, const float* b_host, float* c_back)
{
    try
    {
        queue q(default_selector{}, dpc_common::exception_handler);
        std::cout << "Device: " << q.get_device().get_info<info::device::name>() << "\n";
        buffer<float, 1> a_buf(a_host, range(N * N));
        buffer<float, 1> b_buf(b_host, range(N * N));
        buffer c_buf(reinterpret_cast<float*>(c_back), range(N * N));

        // инициализируем память до начала вычислений
        q.submit([&](auto& h) {
            accessor a(a_buf, h, write_only);
            accessor b(b_buf, h, write_only);
            h.single_task([=]() {
                a[0] = a[0];
                b[0] = b[0];
            });
        });
        q.wait_and_throw();

        q.submit([&](auto& h) {
            accessor a(a_buf, h, read_only);
            accessor b(b_buf, h, read_only);
            accessor c(c_buf, h, write_only);

            accessor<float, 1, cl::sycl::access::mode::read_write, cl::sycl::access::target::local>
            aTile(cl::sycl::range<1>(tile_size*tile_size), h);
            accessor<float, 1, cl::sycl::access::mode::read_write, cl::sycl::access::target::local>
            bTile(cl::sycl::range<1>(tile_size*tile_size), h);

            range<2> matrix_range{ N / 4, N };
            range<2> tile_range{ tile_size / 4, tile_size };
            h.parallel_for(cl::sycl::nd_range<2>(matrix_range, tile_range), [=](auto it) {
                int row = it.get_local_id(0);
                int col = it.get_local_id(1);
                const int globalRow = tile_size * it.get_group(0) + row;
                const int globalCol = tile_size * it.get_group(1) + col;
                const int numTiles = N / tile_size;
                float sum1 = 0.0f, sum2 = 0.0f, sum3 = 0.0f, sum4 = 0.0f;
                const int tmp = tile_size / 4;
                for (int t = 0; t < numTiles; t++)
                {
                    aTile[row*tile_size+col] = a[globalRow * N + tile_size * t + col];
                    bTile[row*tile_size+col] = b[(tile_size * t + row) * N + globalCol];

                    aTile[(row+tmp)*tile_size+col] = a[globalRow*N + tile_size*t + col + tmp*N];
                    bTile[(row+tmp)*tile_size+col] = b[(tile_size*t + row)*N + globalCol + tmp*N];

                    aTile[(row+2*tmp)*tile_size+col] = a[globalRow*N + tile_size*t + col + 2*tmp*N];
                    bTile[(row+2*tmp)*tile_size+col] = b[(tile_size*t + row)*N + globalCol + 2*tmp*N];

                    aTile[(row+3*tmp)*tile_size+col] = a[globalRow*N + tile_size*t + col + 3*tmp*N];
                    bTile[(row+3*tmp)*tile_size+col] = b[(tile_size*t + row)*N + globalCol + 3*tmp*N];

                    it.barrier(cl::sycl::access::fence_space::local_space);
                    #pragma unroll(2)
                    for (int k = 0; k < tile_size; k++)
                    {
                        sum1 += aTile[row*tile_size+k] * bTile[k*tile_size+col];
                    }
                }
            });
        });
    }
}
```



```

        sum2 += aTile[(row + tmp)*tile_size+k] * bTile[k*tile_size+col];
        sum3 += aTile[(row + 2 * tmp)*tile_size+k] * bTile[k*tile_size+col];
        sum4 += aTile[(row + 3 * tmp)*tile_size+k] * bTile[k*tile_size+col];
    }
    it.barrier(cl::sycl::access::fence_space::local_space);
}
c[globalRow * N + globalCol] = sum1;
c[(globalRow + tmp) * N + globalCol] = sum2;
c[(globalRow + 2 * tmp) * N + globalCol] = sum3;
c[(globalRow + 3 * tmp) * N + globalCol] = sum4;
});
}).wait_and_throw();
}
catch (cl::sycl::exception const& e) {
    std::cout << "An exception is caught while multiplying matrices.\n";
    terminate();
}
}
}

```

```

void gpu_mult_base(const float* a_host, const float* b_host, float* c_back)
{
    try {
        queue q(default_selector{}, dpc_common::exception_handler); //default_selector
        cout << "Device: " << q.get_device().get_info<info::device::name>() << "\n";
        buffer<float, 2> a_buf(a_host, range(N, N));
        buffer<float, 2> b_buf(b_host, range(N, N));
        buffer c_buf(reinterpret_cast<float*>(c_back), range(N, N));
        // инициализируем память gpu
        q.submit([&](auto& h) {
            accessor a(a_buf, h, write_only);
            accessor b(b_buf, h, write_only);
            h.single_task([=]() {
                a[0][0] = a[0][0];
                b[0][0] = b[0][0];
            });
        });

        q.wait_and_throw();
        q.submit([&](auto& h) {
            accessor a(a_buf, h, read_only);
            accessor b(b_buf, h, read_only);
            accessor c(c_buf, h, write_only);
            int width_a = a_buf.get_range()[1];

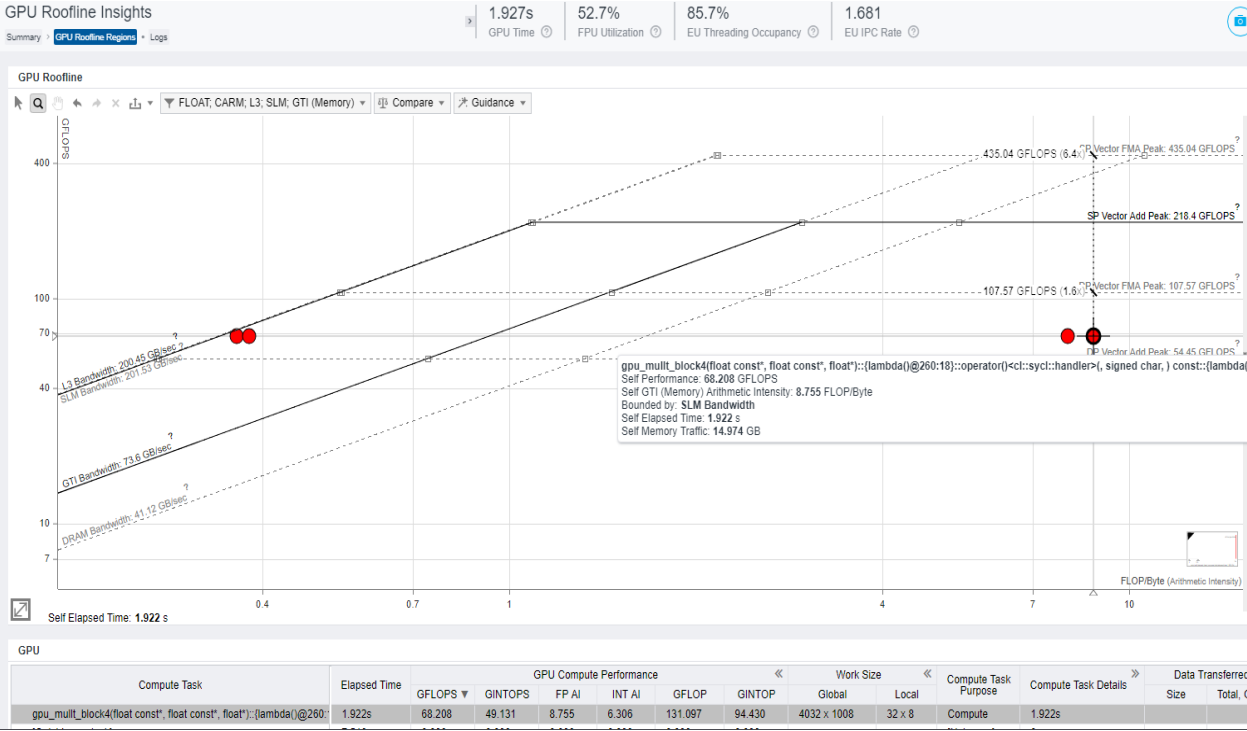
            h.parallel_for(range(N, N), [=](auto index) {
                int row = index[0];
                int col = index[1];
                float sum = 0.0f;
                for (int i = 0; i < width_a; i++)
                    sum += a[row][i] * b[i][col];
                c[index] = sum;
            });
        });
        q.wait_and_throw();
    }
    catch (cl::sycl::exception const& e) {
        cout << "An exception is caught while multiplying matrices.\n";
        terminate();
    }
}
}

```

```

void mult_block2(const float* A, const float* B, float* C)
{
#pragma omp parallel
{
    float tmp[bs_c] = { 0.0f };
#pragma omp for
    for (int ib = 0; ib < N / bs_r; ib++)
        for (int kb = 0; kb < N / bs_c; kb++)
            for (int jb = 0; jb < N / bs_c; jb++)
            {
                const int st = jb * bs_c;
                for (int i = ib * bs_r; i < ib * bs_r + bs_r; i++)
                {
                    for (int k = kb * bs_c; k < kb * bs_c + bs_c; k++)
                    {
                        const float AA = A[i * N + k];
                        const float* pb = B + k * N + st;
#pragma omp simd
                        for (int j = 0; j < bs_c; j++)
                        {
                            tmp[j] += AA * pb[j];
                        }
                    }
                    float* lc = C + i * N + st;
#pragma ivdep
                    for (int j = 0; j < bs_c; j++)
                    {
                        lc[j] += tmp[j];
                        tmp[j] = 0.0f;
                    }
                }
            }
        }
    }
}

```



Roofline 2 метода GPU BLOCK+

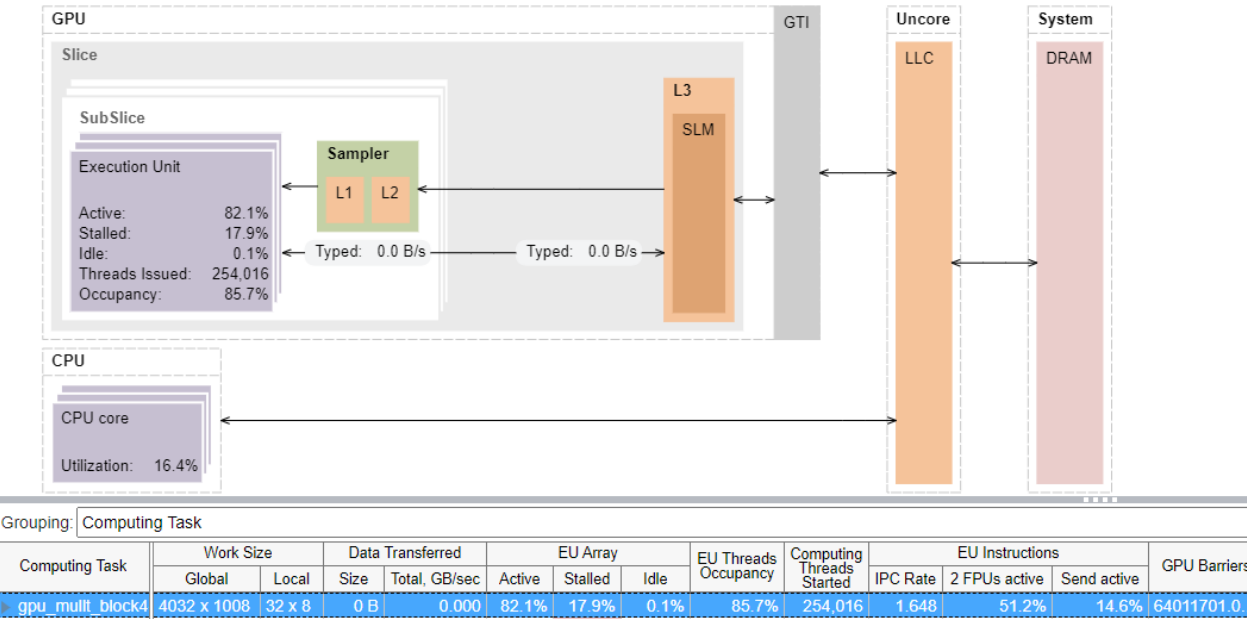


Схема метода GPU BLOCK+