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**

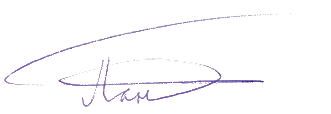
Направление подготовки: «Прикладная математика и информатика»

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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Мееров И. Б.

Нижний Новгород  
2021

# Содержание

[Содержание 2](#_Toc63457172)

[Введение 3](#_Toc63457173)

[1. Программно-аппаратное окружение 4](#_Toc63457174)

[2. Реализации умножения матриц на CPU 5](#_Toc63457175)

[2.1 Базовая версия 5](#_Toc63457176)

[2.2 Блочная версия 7](#_Toc63457177)

[2.3 Оптимизация блочной версии 9](#_Toc63457178)

[2.4 Сравнение производительности CPU версий 10](#_Toc63457179)

[3. Реализация на встроенной GPU с помощью средств OneAPI 12](#_Toc63457180)

[3.1 Наивная версия 12](#_Toc63457181)

[3.2 Базовая версия 12](#_Toc63457182)

[3.3 Блочная версия 13](#_Toc63457183)

[3.4 Оптимизация блочной версии 15](#_Toc63457184)

[3.5 Сравнение производительности GPU версий 18](#_Toc63457185)

[4. Выводы 21](#_Toc63457186)

[4.1 Краткие выводы по CPU версиям 21](#_Toc63457187)

[4.2 Краткие выводы по GPU версиям 21](#_Toc63457188)

[4.3 Производительность MKL на CPU и GPU 22](#_Toc63457189)

[Список литературы 23](#_Toc63457190)

[Приложение 24](#_Toc63457191)

# 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 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 С++, включающее в себя SYCL, позволяющий использовать С++ код для любой архитектуры, производя при этом доступные под конкретную платформу оптимизации.

# 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 Mb, L2 – 256 Kb, L1 – 64 Kb.

* Интегрированная графика:

Intel UHD Graphics 630, частота 1150 МГц

* Память:

Двухканальная DDR4, 2x8 Gb, 3100 МГц

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

# Реализации умножения матриц на CPU

## Базовая версия

Рассмотрим простейшую реализацию умножения матриц размера на CPU. В данной реализации используется правильный порядок циклов . Цикл параллелится по номеру итерации 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) -ую строку матрицы C;
* разместить в кэше (L1) -ую строку матрицы C;
* разместить в кэше (L1) -ый элемент матрицы A;
* разместить в кэше (L1, L2, L3) M строк матрицы B, начиная с -ой, и M элементов из матрицы A, начиная с -ого (M должно быть как можно больше, но все строки, скорее всего, не уместятся).

Это позволит:

* выполнить цикл по 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

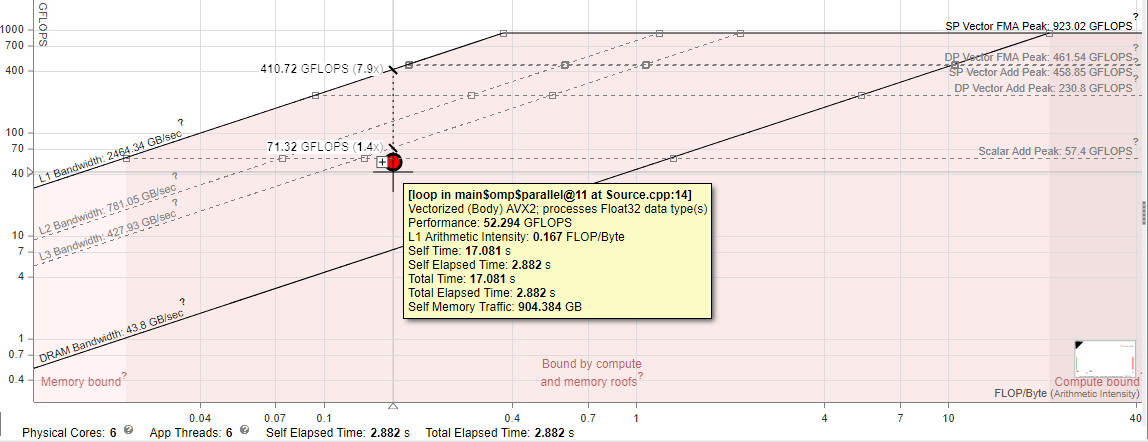
В соответствии с описанным выше принципом распределения данных по кэшам получим:

* В кэше L1 – одна -ая строка матрицы C строка + три строки B начиная с -ой + три элемента из матрицы A, начиная с ‑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):

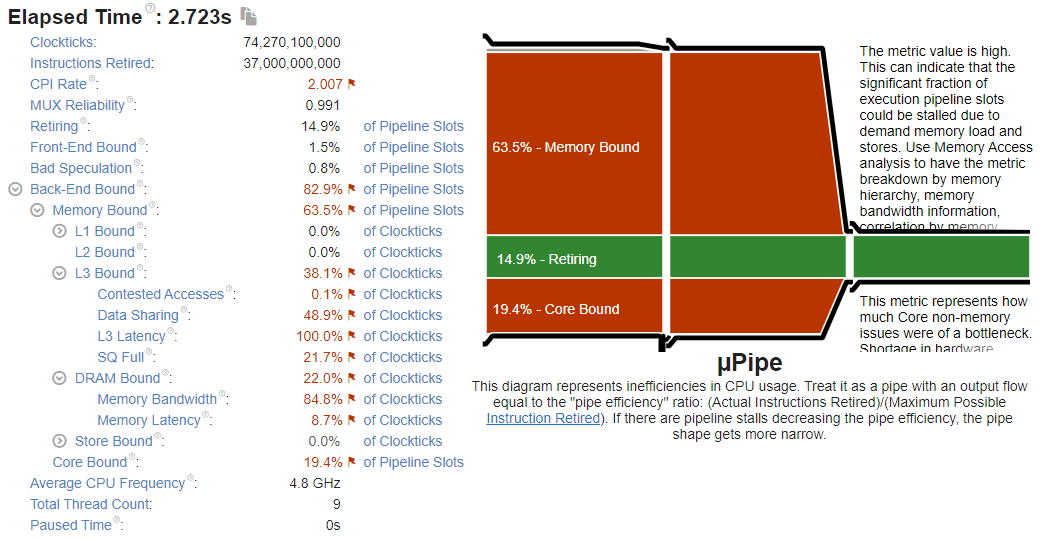
* Все данные для внутреннего цикла лежат в кэшах, получаем N умножений/сложений.
* За счет 116 строк матрицы B и 116 элементов из A можно выполнить внутренний цикл 116 раз, на имеющихся данных, итого 116N.

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 СPU.

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



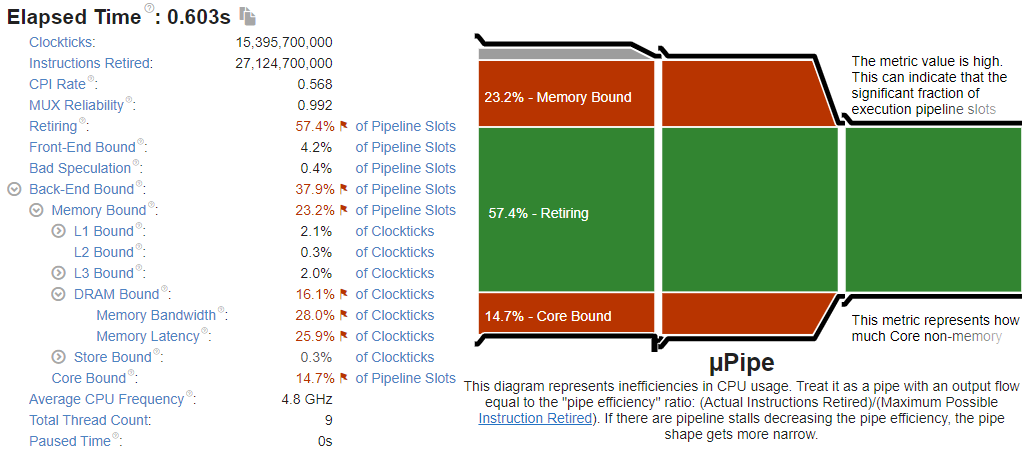
Анализ базового алгоритма на CPU через VTune.

## Блочная версия

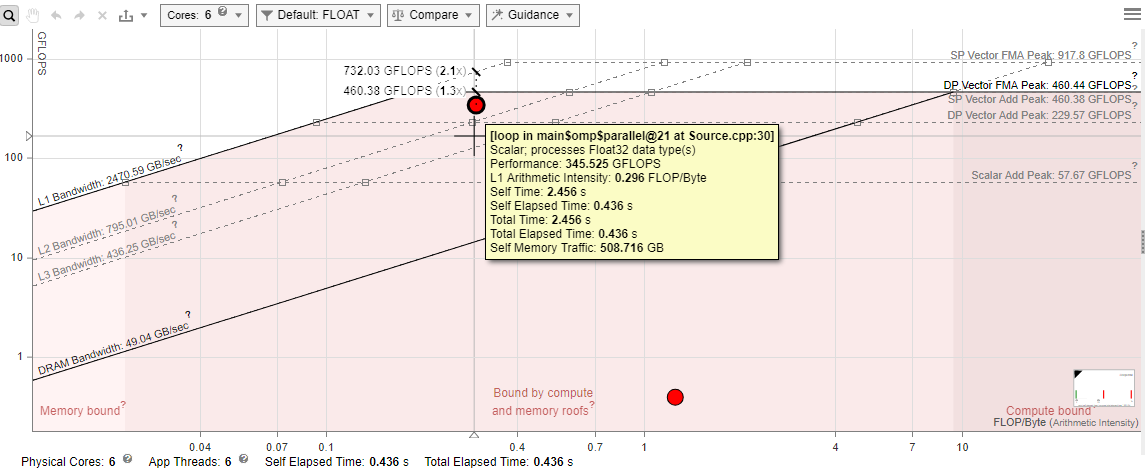
В предыдущей версии в кэшах уместилось всего 117 строк (116 из B, 1 из C). В блочной версии алгоритма в кэшах выгодно размещать строки из всех трех матриц. В данном случае можно уместить по 39 строк из матрицы A, B и C. При размере блока 39 x 39 на этих данных можно выполнить умножений/сложений. То есть 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]



Анализ блочного алгоритма на CPU через VTune.

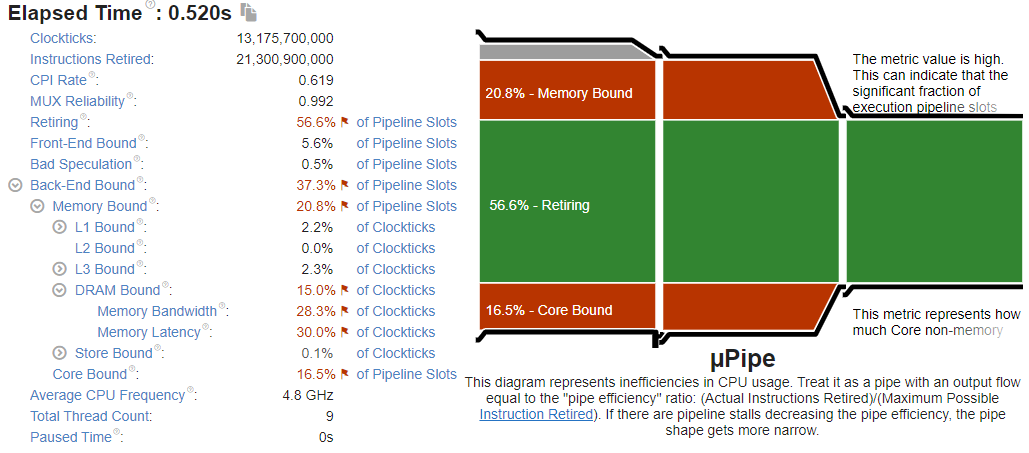


Roofline блочного алгоритма на СPU.

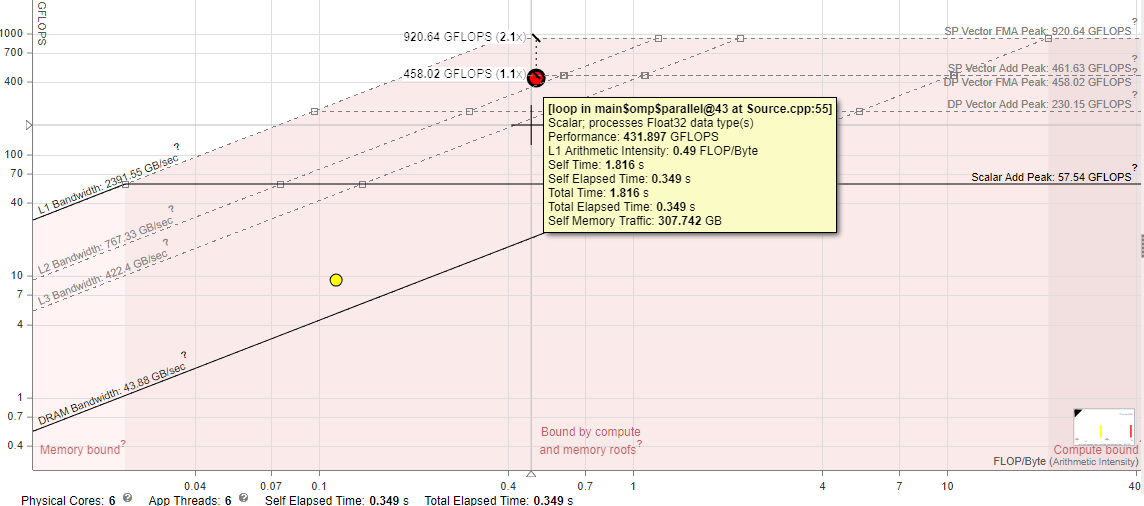
Данный вариант алгоритма практически не ограничивается пропускной способностью ОЗУ. Узким местом становится работа с кэшами.

## 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.



Анализ блочного+ алгоритма на CPU через VTune.



Roofline блочного+ алгоритма на СPU.

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.

## Сравнение производительности CPU версий

Сравнение производительности CPU версий.

Ускорение относительно базовой версии.

MKL версия практически вдвое производительней блочной версии, таким образом, она задействует практический 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:

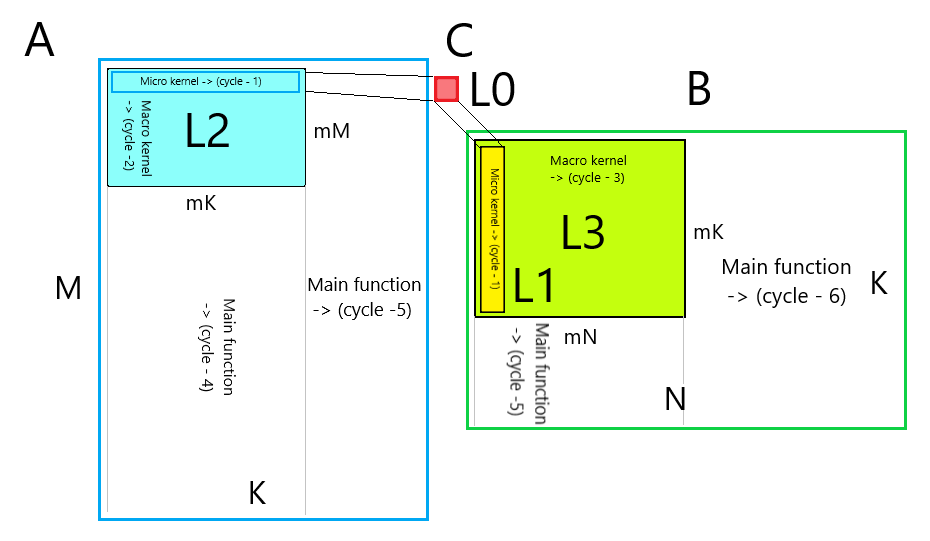


Схема умножения матриц, источник: [3].

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

# Реализация на встроенной GPU с помощью средств OneAPI

## Наивная версия

В данной версии создаётся рабочее пространство с размерами как у матрицы C   
(). Так как количество «потоков» 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]

## 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 чтений из глобальной памяти. Потоков , значит, общее количество чтений из глобальной памяти .

## 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 и , где – размер блока. Внутри блока GPU с индексом происходит следующее:

* В матрице A идёт работа с блоком , в матрице B с блоком .
* 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.

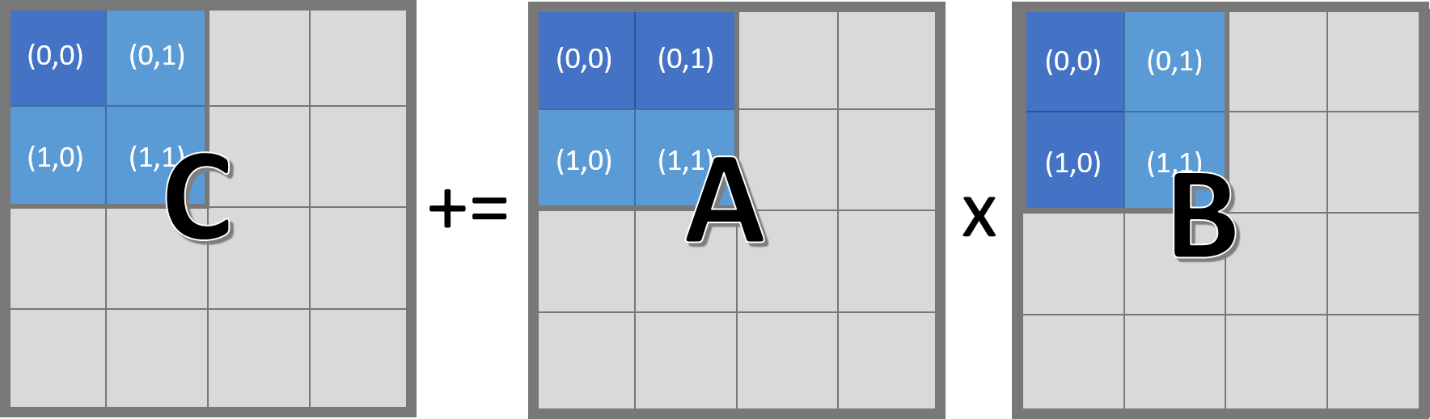


Иллюстрация работы блочного алгоритма умножения матриц на GPU (часть 1). В качестве примера рассмотрен блок GPU (0,0). Внутри блока указаны локальные индексы потоков.

* Поток ожидает другие потоки из блока.
* В матрице A идёт работа с блоком , в матрице B с блоком .
* 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.

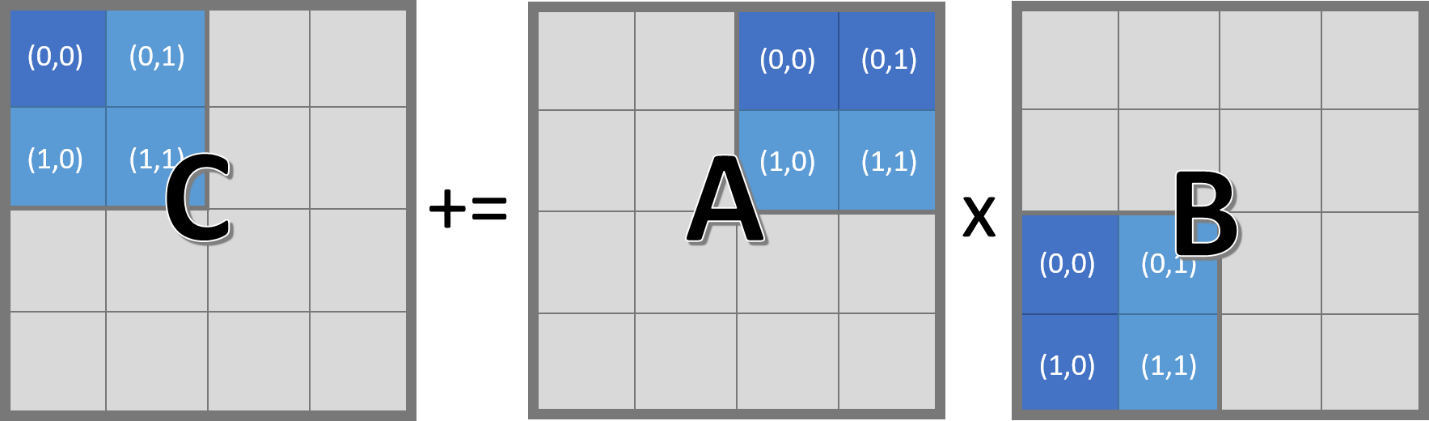


Иллюстрация работы блочного алгоритма умножения матриц на GPU (часть 2). В качестве примера рассмотрен блок GPU (0,0). Внутри блока указаны локальные индексы потоков.

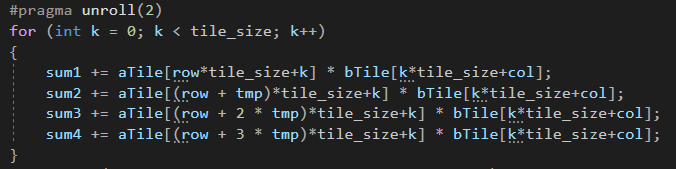
* После шагов все потоки внутри каждого блока вычислили «свой» элемент из матрицы С.

Всего в данном алгоритме каждый поток делает чтений из глобальной памяти и чтений из локальной. Общее количество потоков , значит количество чтений из глобальной памяти и из локальной. Обозначим время доступа к глобальной памяти, как , а к локальной как . Тогда теоритическое ускорение относительно базовой версии можно рассчитать так:

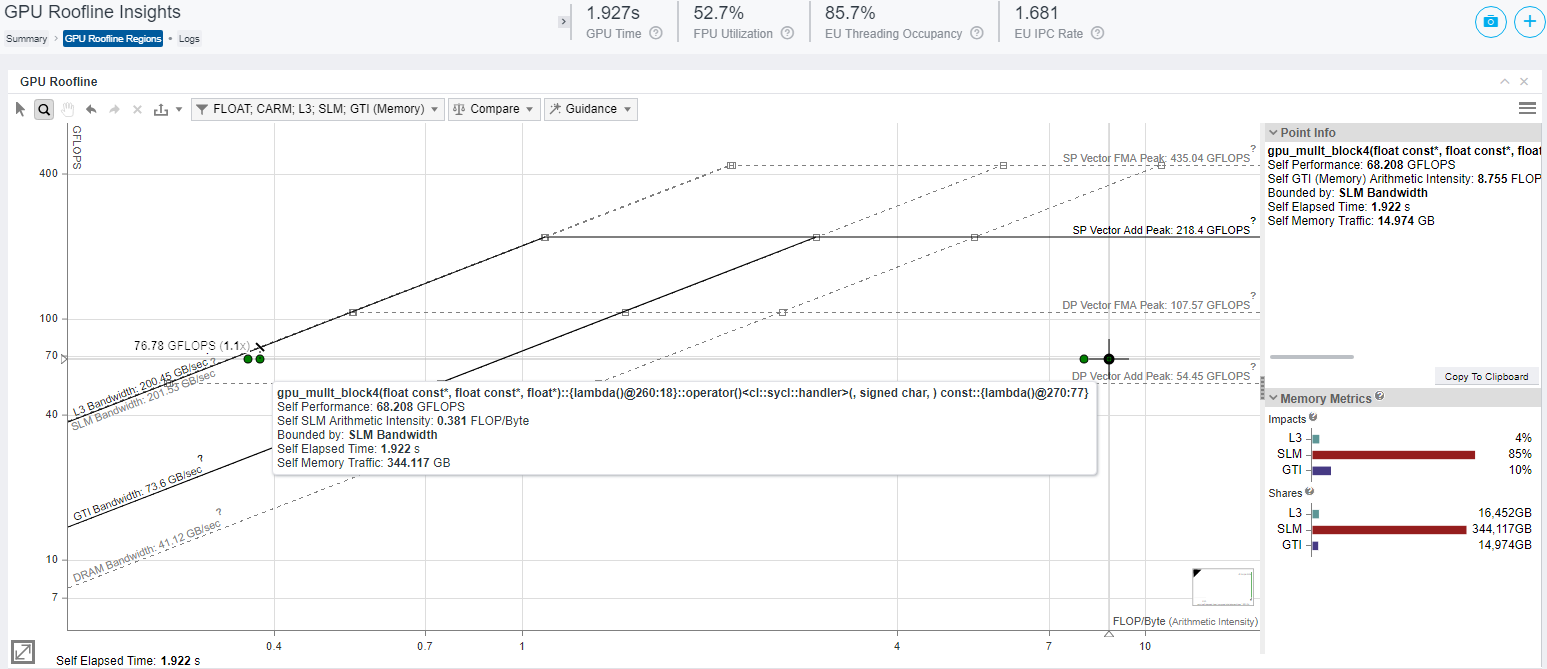
## Оптимизация блочной версии

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:

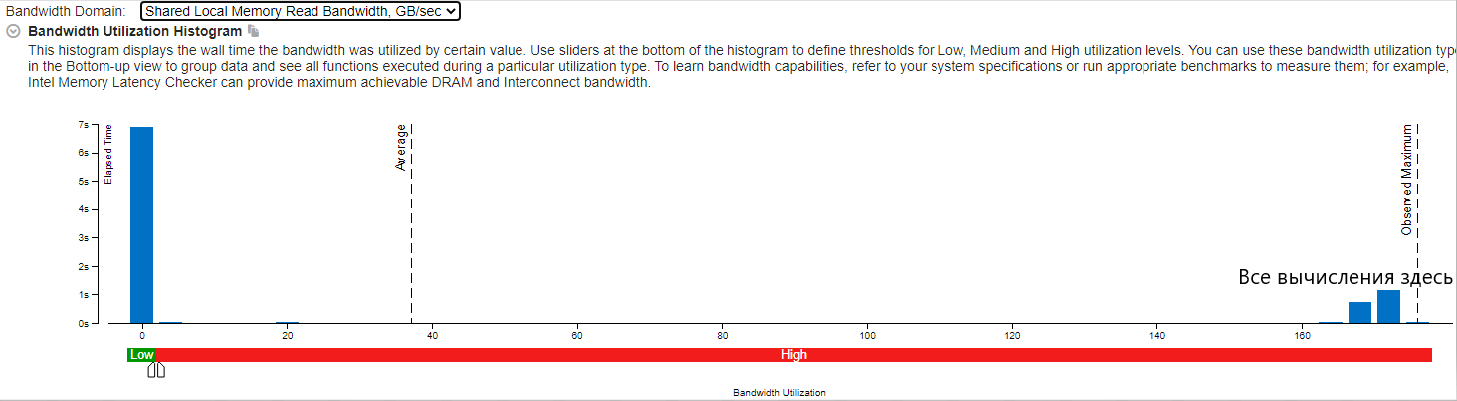


Данные улучшения ускоряют вычисления в 3.2 раза относительно базовой версии алгоритма.



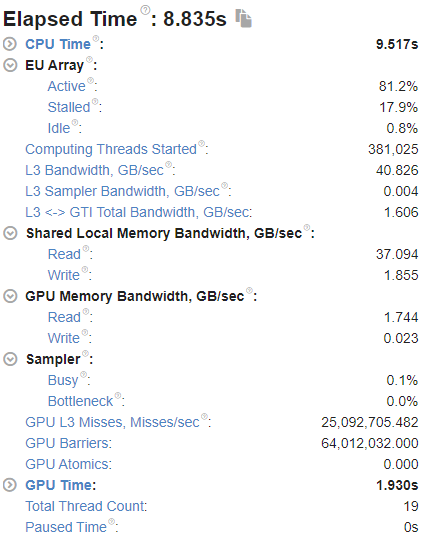
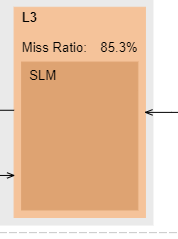
Roofline блочного+ алгоритма на GPU.

Roofline показывает, что удалось задействовать лишь 15.6% от общей вычислительной мощности. Причины, почему не удалось задействовать все вычислительные ресурсы, не очень понятны.



Bandwidth для блочного+ алгоритма на GPU.

Общая память, а также L3 (bandwidth 200 GB/sec, это близко к предельным значениям пропускной способности L3) довольно активно используются. Возможно, проблема производительности связанна с неэффективным использованием памяти. Может быть, в текущей версии не использовались FMA инструкции.



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.

## Сравнение производительности GPU версий

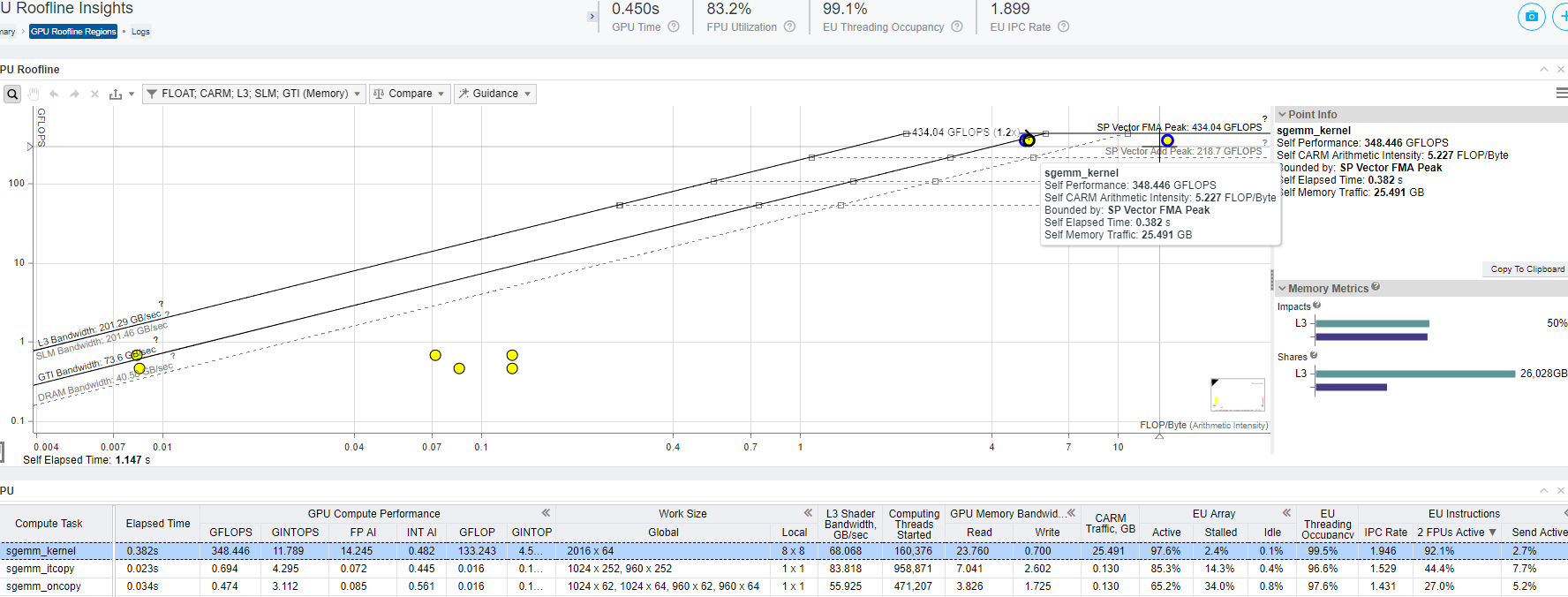
Улучшенная блочная версия в 3.2 раза быстрее базовой.

Сравнение производительности GPU версий

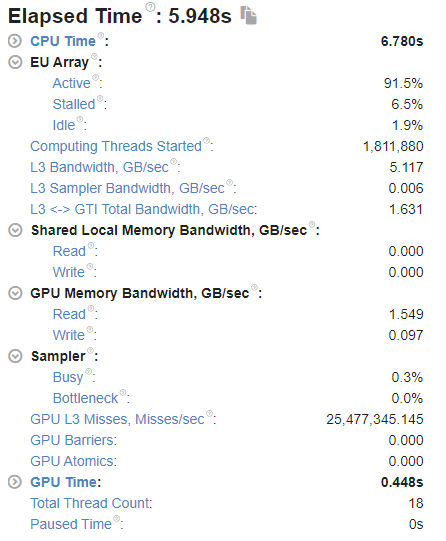
MKL версия отстаёт по времени работы при N меньшем 2496, но с увеличением размера матрицы, MKL всё больше обгоняет блочную версию.

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.



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.

# Выводы

## 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 практический 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.

## 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.

## 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\_mullt\_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;

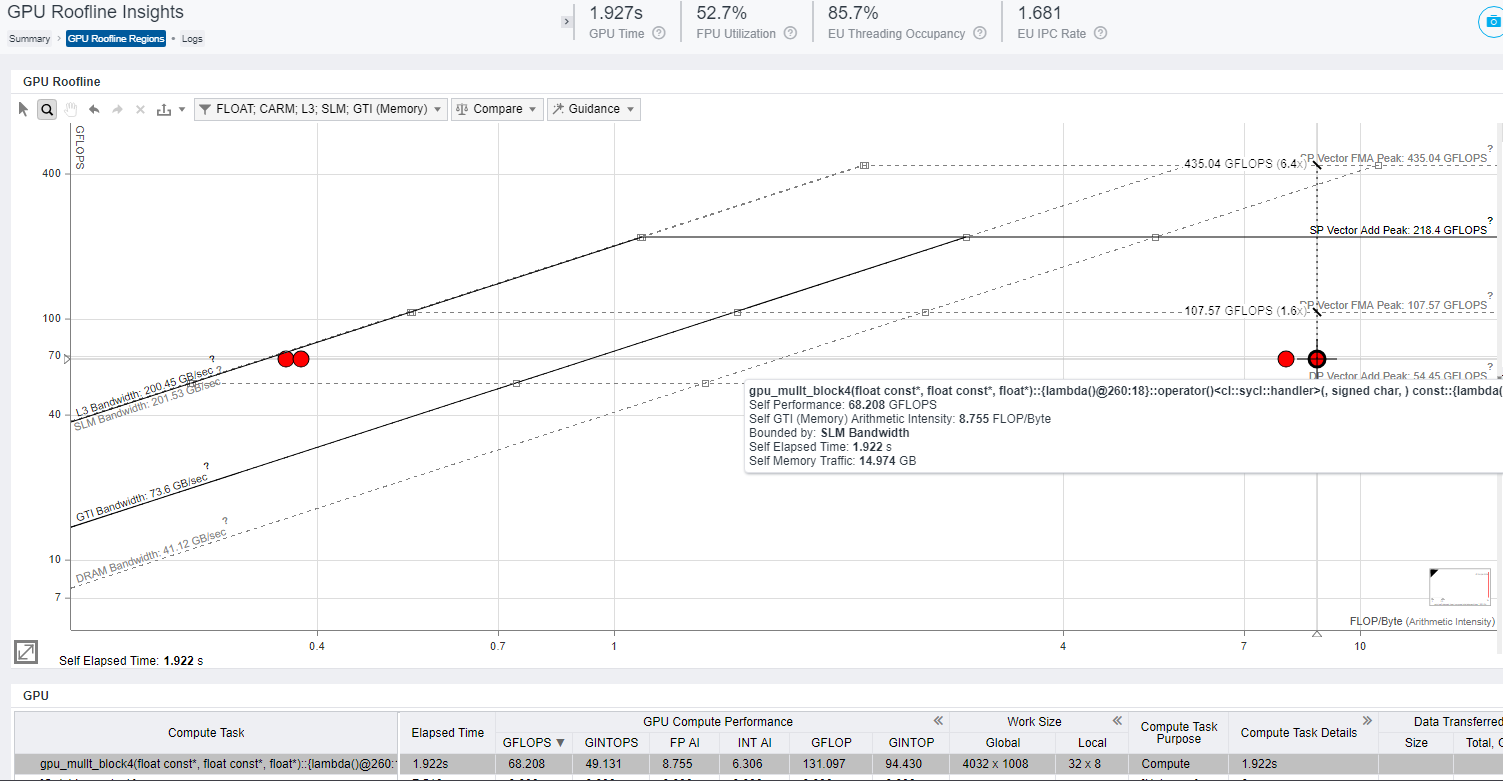
}

}

}

}

}



Roofline 2 метода GPU BLOCK+

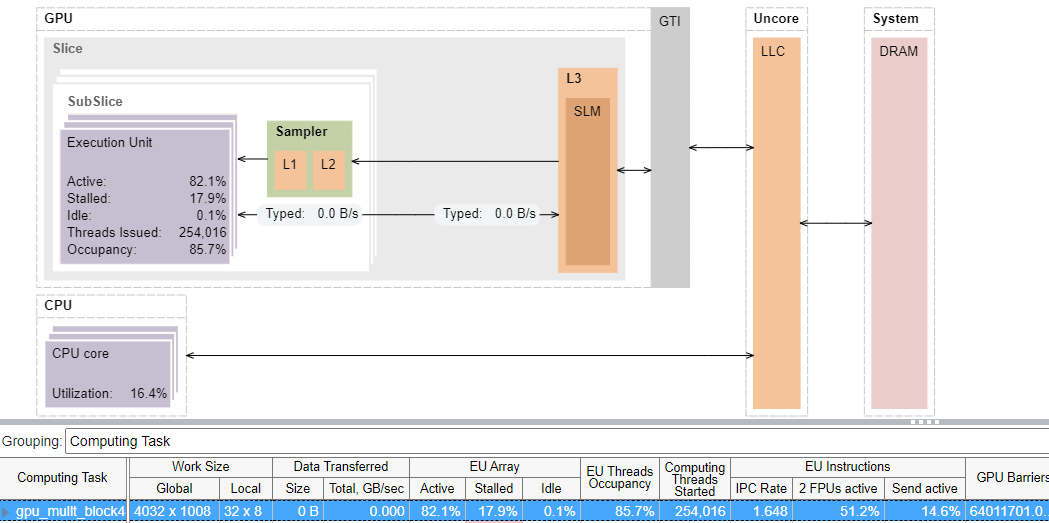


Схема метода GPU BLOCK+