|  |  |
| --- | --- |
|  | **Правительство Российской Федерации**  **Федеральное государственное автономное образовательное учреждение высшего образования  «Национальный исследовательский университет «Высшая школа экономики»** |

**Отчёт по домашнему заданию №2 по курсу**

**«Высокопроизводительные Вычисления»**

**Тема работы: «Технология *CUDA*»**

Выполнил:  
Забурунов Л. В., МСМТ221

(подпись)

Проверил(а):  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

1 ноября 2022 г.

г. Москва

Оглавление

[Цель работы 3](#_Toc119339297)

[Постановка задачи 3](#_Toc119339298)

[1. Решение первой задачи 4](#_Toc119339299)

[2. Переход к pinned памяти 7](#_Toc119339300)

[3. Разбиение на асинхронные потоки 8](#_Toc119339301)

[4. Использование блочной памяти в ядре 10](#_Toc119339302)

[5. Проверка работоспособности всех вариантов 12](#_Toc119339303)

[6. Анализ производительности различных вариантов 16](#_Toc119339304)

[7. Анализ влияния размера блока на производительность 17](#_Toc119339305)

[Выводы 18](#_Toc119339306)

[Приложение 1. Замеры производительности 19](#_Toc119339307)

[Приложение 2. Влияние размера блока 23](#_Toc119339308)

# Цель работы

Ознакомиться с особенностями программирования по гибридной модели (с использованием графического процессора) на примере задачи умножения матриц с использованием технологии *NVIDIA CUDA.*

# Постановка задачи

Требуется создать программу на языке *C/C++*, осуществляющую умножение двух матриц. Элементы матриц представлены в виде чисел с плавающей точкой с двойной точностью (*double*), а сами матрицы хранятся в памяти по столбцам (*column-major order*).

Задача должна быть реализована несколькими способами:

1. Простейшая реализация с использованием только глобальной памяти;
2. Модификация предыдущей версии программы, заключающаяся в выделении памяти для трёх матриц как «*pinned memory*»;
3. Модификация предыдущей версии программы, заключающаяся в использовании асинхронных операций для копирования данных между *CPU* и *GPU* (*CUDA Streams*);
4. Реализация блочного алгоритма перемножения матриц с задействованием блочной разделяемой памяти (*shared memory*).

# Решение первой задачи

Воспользуемся созданной при выполнении прошлого домашнего задания оболочкой программы для обработки входных данных. При выполнении работы с графическим процессором появляются следующие важные стадии программы: передача данных на *GPU* и получение результата из *GPU*. Для контроля ошибок во время обмена данными будем пользоваться информацией, предоставляемой интерфейсом *CUDA*:

cudaError\_t errorCode;

errorCode = cudaMalloc(&gpuM, sizeof(int));

if (errorCode != cudaSuccess) {

printf("Error occured while allocating M on GPU (%s)", cudaGetErrorString(errorCode));

return;

}

errorCode = cudaMemcpy(gpuM, &M, sizeof(int), cudaMemcpyHostToDevice);

if (errorCode != cudaSuccess) {

printf("Error occured while transmitting M to GPU (%s)", cudaGetErrorString(errorCode));

return;

}

Соответственно, выполнение нашей подпрограммы умножения матриц можно разделить на три раздела. Тогда помимо замера общего времени выполнения будем смотреть также на затраты на каждую из составляющих:

printf("Intermediate measure. Time complexity for transmitting to GPU: %f s.\n", ConvertChronoToSeconds(t0, t));

printf("Intermediate measure. Time complexity for kernel execution: %f s.\n", ConvertChronoToSeconds(t0, t));

printf("Intermediate measure. Time complexity for transmitting from GPU: %f s.\n", ConvertChronoToSeconds(t0, t));

В *CUDA-*ядре мы работаем таким образом, что каждый поток подсчитывает конкретное значение матрицы С. Для этого достаточно использовать всего один цикл (остальные два растворяются в двумерной сетке потоков):

// CUDA-ядро для перемножения матриц

// Последние два параметра позволяют разбить матрицу на зоны для асинхронных потоков (каждый считает свой кусок матрицы С)

\_\_global\_\_ void mm\_kernel\_global(int \*M, int \*N, int \*K, double \*A, double \*B, double \*C, int streamRowOffset, int streamColumnOffset)

{

const int row = streamRowOffset + blockDim.x \* blockIdx.x + threadIdx.x,

column = streamColumnOffset + blockDim.y \* blockIdx.y + threadIdx.y;

const int m = \*M, n = \*N, k = \*K;

//if (threadIdx.x % 32 == 31 && threadIdx.y % 32 == 31) printf("(stream #(%f) (thread #(%d, %d)) Working for C[%d, %d]\n", (float) streamColumnOffset / k, blockDim.x \* blockIdx.x + threadIdx.x, blockDim.y \* blockIdx.y + threadIdx.y, row, column);

if (row >= m || column >= k) return;

double result = C[column \* m + row];

for (int i = 0; i < n; i = i + 1) {

//if (blockIdx.x > 0 && blockIdx.y > 0 && threadIdx.x == 0 && threadIdx.y == 0) printf("C[%d, %d] (%.3f) += A[%2d, %2d] (%.3f) \* B[%2d, %2d] (%.3f)\n", row, column, result, row, i, A[i \* m + row], i, column, B[column \* n + i]);

result += A[i \* m + row] \* B[column \* n + i];

}

//printf("(thread #(%d, %d)) C index : %d, value: %e.\n\n", blockDim.x \* blockIdx.x + threadIdx.x, blockDim.y \* blockIdx.y + threadIdx.y, col \* m + row, C[col \* m + row]);

C[column \* m + row] = result;

}

Сам вызов ядра выглядит так:

// Оценка геометрии ядра

dim3 blocks, threads;

int maxDim = std::max(M, std::max(N, K));

if (M \* K <= 1024) {

blocks = dim3(1, 1, 1);

threads = dim3(maxDim, maxDim, 1);

} else {

int mBlocks = M / 32,

kBlocks = K / 32,

mResidual = M % 32,

kResidual = K % 32;

// Проверку корректности индекса выполняем внутри ядра

blocks = dim3(mBlocks + std::min(mResidual, 1), kBlocks + std::min(kResidual, 1), 1);

threads = dim3(32, 32, 1);

}

printf("Topology chosen for kernel launch: BLOCKS(%d, %d, 1), THREADS(%d, %d, 1).\n", blocks.x, blocks.y, threads.x, threads.y);

// Работа ядра

t0 = std::chrono::system\_clock::now();

mm\_kernel\_global<<<blocks, threads>>>(gpuM, gpuN, gpuK, gpuA, gpuB, gpuC);

cudaDeviceSynchronize();

t = std::chrono::system\_clock::now();

# Переход к pinned памяти

Использование *pinned* памяти предполагает выделение памяти в ОЗУ средствами *CUDA*, а не классическими способами выделения через *new, malloc* или *calloc*:

if (toCudaPinnedMemory == false) {

inputMatrix = new double[rows \* columns];

} else {

cudaHostAlloc(&inputMatrix, rows \* columns \* sizeof(double), cudaHostAllocDefault);

}

Никакие другие манипуляции не требуются.

# Разбиение на асинхронные потоки

Для выполнения третьего пункта ДЗ мы должны реализовать передачу данных и выполнение ядер на не зависящие друг от друга части. Полностью сделать это не удастся, поскольку в любом случае мы для подсчёта одного элемента будем использовать данные из самых разных концов матрицы. Однако, можно разбить матрицу на участки и осуществлять обмен данных через асинхронные потоки этими участками.

Перенос исходных данных в *GPU* выглядит так:

// Заполнение потоков операциями переноса в GPU

int aStreamStep = (M \* N) / CUDA\_STREAMS,

bStreamStep = (N \* K) / CUDA\_STREAMS,

cStreamStep = (M \* K) / CUDA\_STREAMS,

aStreamResidue = (M \* N) % CUDA\_STREAMS,

bStreamResidue = (N \* K) % CUDA\_STREAMS,

cStreamResidue = (M \* K) % CUDA\_STREAMS;

// Возможно, матрицы не группируются на потоки без остатка

// В таком случае мы даём последнему потоку дополнительную работу

// Если остатки нулевые, то получается самый обычный цикл (будто не до [CUDA\_STREAMS - 1], а до [CUDA\_STREAMS]

for (int i = 0; i < CUDA\_STREAMS - 1; i = i + 1) {

cudaMemcpyAsync(gpuA + i \* aStreamStep,

A + i \* aStreamStep,

sizeof(double) \* aStreamStep,

cudaMemcpyHostToDevice,

streams[i]);

cudaMemcpyAsync(gpuB + i \* bStreamStep,

B + i \* bStreamStep,

sizeof(double) \* bStreamStep,

cudaMemcpyHostToDevice,

streams[i]);

}

cudaMemcpyAsync(gpuA + (CUDA\_STREAMS - 1) \* aStreamStep,

A + (CUDA\_STREAMS - 1) \* aStreamStep,

sizeof(double) \* (aStreamStep + aStreamResidue),

cudaMemcpyHostToDevice,

streams[CUDA\_STREAMS - 1]);

cudaMemcpyAsync(gpuB + (CUDA\_STREAMS - 1) \* bStreamStep,

B + (CUDA\_STREAMS - 1) \* bStreamStep,

sizeof(double) \* (bStreamStep + bStreamResidue),

cudaMemcpyHostToDevice,

streams[CUDA\_STREAMS - 1]);

Вызов *CUDA-*ядра выглядит так:

// Заполнение потоков вычислениями

for (int i = 0; i < CUDA\_STREAMS; i = i + 1) {

int columnOffset = i \* 32 \* blocks.y;

printf("EN - Stream #%d started with offset %d rows\n", i + 1, columnOffset);

mm\_kernel\_global<<<blocks, threads, 0, streams[i]>>>(gpuM, gpuN, gpuK, gpuA, gpuB, gpuC, 0, columnOffset);

}

Перенос результата из *GPU* выглядит так:

// Заполнение потоков операциями переноса из GPU

for (int i = 0; i < CUDA\_STREAMS - 1; i = i + 1) {

cudaMemcpyAsync(C + i \* cStreamStep,

gpuC + i \* cStreamStep,

sizeof(double) \* cStreamStep,

cudaMemcpyDeviceToHost,

streams[i]);

}

// См. пояснение выше для матриц А и В

cudaMemcpyAsync(C + (CUDA\_STREAMS - 1) \* cStreamStep,

gpuC + (CUDA\_STREAMS - 1) \* cStreamStep,

sizeof(double) \* (cStreamStep + cStreamResidue),

cudaMemcpyDeviceToHost,

streams[CUDA\_STREAMS - 1]);

cudaDeviceSynchronize();

# Использование блочной памяти в ядре

Для использования *shared memory* нам нужно видоизменить работу *GPU-*ядра. Будем держать в этой памяти кусок матрицы 32х32, и тогда мы можем воспользоваться методом блочного умножения. Теперь вместо одного цикла для прохода вдоль всей матрицы мы получаем два цикла, первый из которых «шагает» по блокам, а второй – по элементам внутри блока (для получения результата умножения двух подматриц). Нужно также учитывать возможность выхода за границы исходных данных, поэтому вводятся дополнительные проверки.

Новое ядро выглядит следующим образом:

// CUDA-ядро для перемножения матриц с разделяемой памятью (блочное перемножение куском 32 х 32)

\_\_global\_\_ void mm\_kernel\_shared(int \*M, int \*N, int \*K, double \*A, double \*B, double \*C)

{

int m = \*M, n = \*N, k = \*K;

// Блоки по 32х32 определяют кусок матрицы С, который мы обсчитываем

// Нить - это один из элементов этого блока

int baseRow = blockDim.x \* blockIdx.x, baseColumn = blockDim.y \* blockIdx.y,

threadRow = baseRow + threadIdx.x, threadColumn = baseColumn + threadIdx.y;

\_\_shared\_\_ double cachedA[32][33];

\_\_shared\_\_ double cachedB[32][33];

int div = n / 32, residue = n % 32;

double result = C[threadColumn \* m + threadRow];

// Внешний цикл - это ряд (сумма умножений подматриц)

for (int i = 0; i < div; i = i + 1) {

// В рамках каждого слагаемого мы остаёмся на том же месте в матрице С, но сдвигаемся вдоль матриц А (вправо по строкам) и B (вниз по столбцам)

int aColumn = threadIdx.y + blockDim.y \* i, bRow = threadIdx.x + blockDim.x \* i;

cachedA[threadIdx.x][threadIdx.y] = A[aColumn \* m + threadRow];

cachedB[threadIdx.x][threadIdx.y] = B[threadColumn \* n + bRow];

\_\_syncthreads();

// Внутренний цикл - это подсчёт слагаемого ряда (умножение матриц)

if (threadRow < m && threadColumn < k) {

for (int j = 0; j < 32; j = j + 1) {

//if (/\*blockIdx.x > 0 && blockIdx.y > 0 && \*/threadIdx.x == 0 && threadIdx.y == 0) printf("C[%d, %d] (%.3f) += sharedA[%2d, %2d] (%.3f) \* sharedb[%2d, %2d] (%.3f)\n", threadRow, threadColumn, result, threadIdx.x, j, cachedA[threadIdx.x][j], j, threadIdx.y, cachedB[j][threadIdx.y]);

result += cachedA[threadIdx.x][j] \* cachedB[j][threadIdx.y];

}

}

\_\_syncthreads();

}

if (threadRow >= m || threadColumn >= k) return;

// То, что не влезает в блоки по 32, считаем по старинке

for (int i = 0; i < residue; i = i + 1) {

//if (/\*blockIdx.x > 0 && blockIdx.y > 0 && \*/threadIdx.x == 0 && threadIdx.y == 0) printf("C[%d, %d] (%.3f) += A[%2d, %2d] (%.3f) \* B[%2d, %2d] (%.3f)\n", threadRow, threadColumn, result, threadRow, (div \* 32 + i), A[(div \* 32 + i) \* m + threadRow], (div \* 32 + i), threadColumn, B[threadColumn \* n + (div \* 32 + i)]);

result += A[(div \* 32 + i) \* m + threadRow] \* B[threadColumn \* n + (div \* 32 + i)];

}

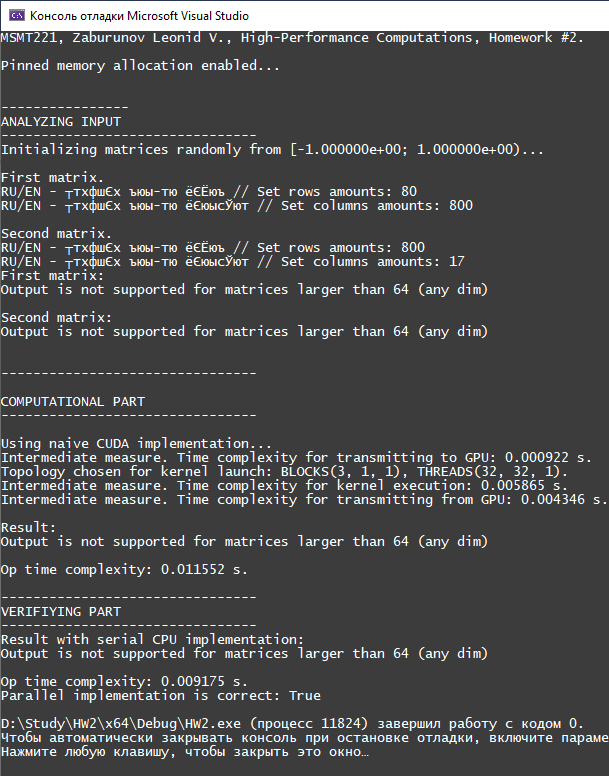
C[threadColumn \* m + threadRow] = result;

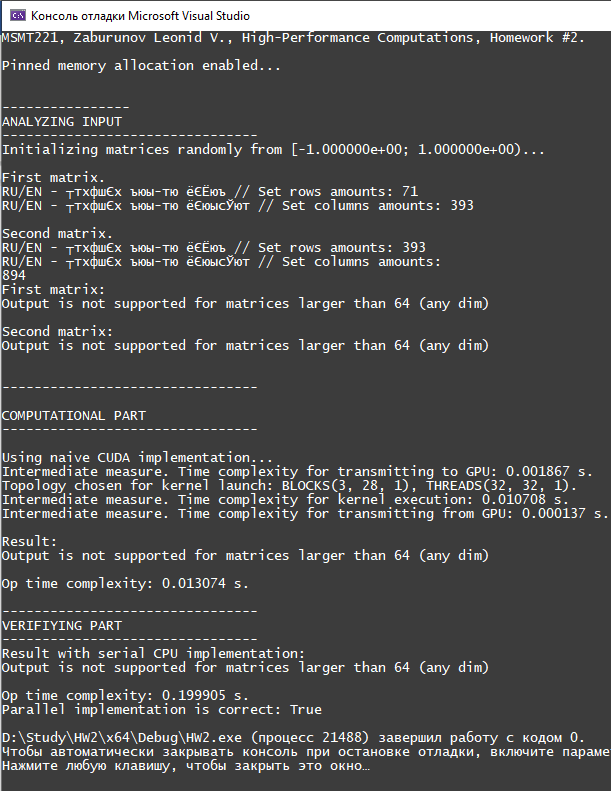
}

# Проверка работоспособности всех вариантов

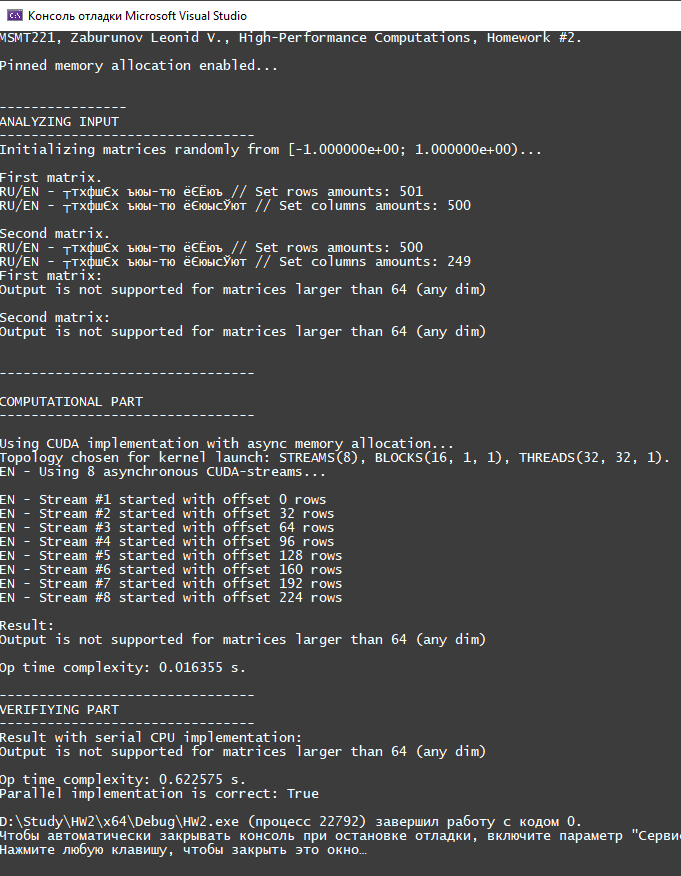
Разработанные варианты требуется проверить на корректность результата на разных входных данных. Можно сравнивать с любой из функций, созданных для ДЗ №1, но нам достаточно самой простой, неоптимизированной, последовательной версии.

Проверка первого и второго вариантов программы (они отличаются только способом выделения памяти):

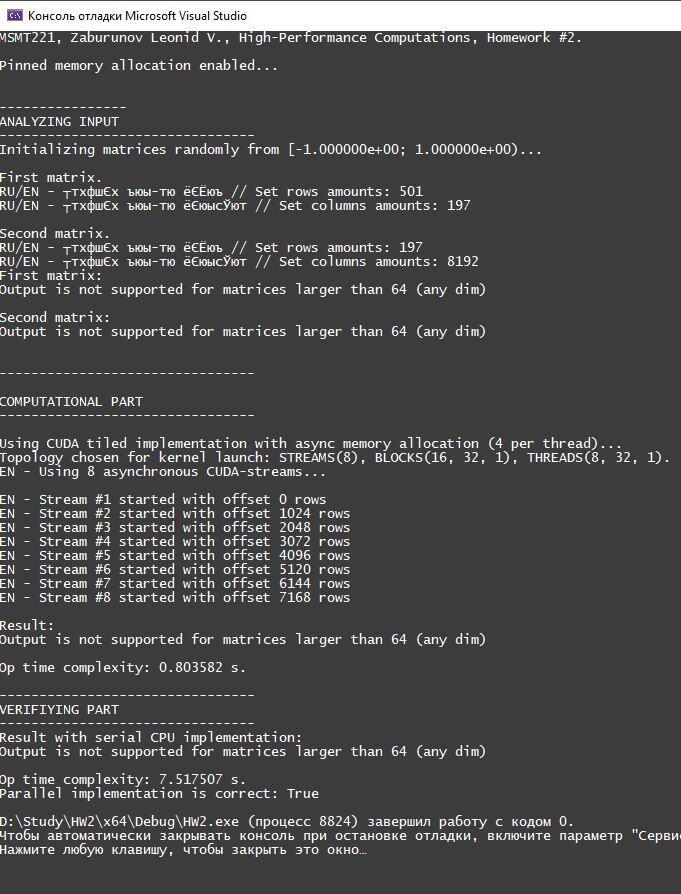




Проверка третьего варианта программы:



Проверка четвёртого варианта программы:



# Анализ производительности различных вариантов

Проанализируем производительность созданных программ, сравнивая их не только между собой, но и с вариантами, разработанными в ходе работы над ДЗ1. Поскольку мы уже точно знаем, что алгоритмы дают верный результат, можно воспользоваться подсчётом на единичных матрицах. Результаты приведены в приложении 1.

# Анализ влияния размера блока на производительность

До сих пор мы работали с сеткой потоков фиксированной размерности, то есть в одном блоке находилось 1024 потока по системе 32х32. Теперь мы посмотрим на то, как влияет размер блока на производительность.

Протестируем на единичных матрицах размерностей 400 и 2500 для квадратных блоков с размерами 2, 4, 8, 12, 16, 24, 32. Результаты приведены в приложении 2.

# Выводы

В рамках выполнения работы была создана *GPU-*версия программы умножения матриц. На результатах виден колоссальный рост в сравнении с *CPU*-версией, при этом также была проведена оптимизация программы и производительность наилучшей версии для *GPU* выросла ещё в несколько раз.

Видим, что использование графического процессора позволяет работать с матрицами гораздо большего размера, оставаясь в рамках разумного времени на вычисление. Разница между некоторыми реализациями видна не везде, поскольку некоторые эффекты становятся существенными только на очень больших данных (скорее всего, дело в затратах на обмен между основной памятью и памятью ГП).

Проведённый анализ влияния размера блока потоков показывает зависимость, близкую по характеру к гиперболической. На матрицах размера 400 данные получились неоднородными, однако при переходе к размеру 2500 вклад посторонних эффектов в общее время становится несущественным и мы видим чистый график.

# Приложение 1. Замеры производительности

:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Версия программы | Число потоков | Замер 1, с | Замер 2, с | Замер 3, с | Замер 4, с | Замер 5, с | Среднее зн-е, с |
| CPU (база) | 12 (cpu) | 4,80E-02 | 4,88E-02 | 4,65E-02 | 4,82E-02 | 5,01E-02 | 4,832E-02 |
| CPU (AVX-512) | 12 (cpu) | 2,47E-02 | 2,46E-02 | 2,54E-02 | 2,52E-02 | 2,97E-02 | 2,592E-02 |
| База | - | 1,864E-01 | 1,791E-01 | 1,764E-01 | 1,789E-01 | 1,783E-01 | 1,798E-01 |
| PINNED | - | 1,097E-03 | 1,119E-03 | 1,119E-03 | 1,108E-03 | 1,098E-03 | 1,108E-03 |
| STREAMS | 1 | 1,049E-03 | 1,052E-03 | 1,068E-03 | 1,049E-03 | 1,055E-03 | 1,055E-03 |
| STREAMS | 2 | 9,400E-04 | 9,460E-04 | 9,500E-04 | 9,500E-04 | 9,670E-04 | 9,506E-04 |
| STREAMS | 4 | 9,770E-04 | 9,660E-04 | 9,770E-04 | 9,640E-04 | 9,640E-04 | 9,696E-04 |
| STREAMS | 8 | 1,010E-03 | 9,840E-04 | 1,021E-03 | 1,015E-03 | 1,020E-03 | 1,010E-03 |
| TILED | 8 | 1,143E-03 | 1,000E-03 | 1,000E-03 | 1,001E-03 | 1,025E-03 | 1,034E-03 |
| TILED (per 4) | 8 | 1,124E-03 | 1,034E-03 | 1,012E-03 | 1,009E-03 | 1,000E-03 | 1,036E-03 |

:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Версия программы | Число потоков | Замер 1, с | Замер 2, с | Замер 3, с | Замер 4, с | Замер 5, с | Среднее зн-е, с |
| CPU (база) | 12 (cpu) | 3,56E-01 | 3,91E-01 | 4,70E-01 | 3,53E-01 | 3,56E-01 | 3,850E-01 |
| CPU (AVX-512) | 12 (cpu) | 2,66E-01 | 3,52E-01 | 2,73E-01 | 2,90E-01 | 2,73E-01 | 2,910E-01 |
| База | - | 1,848E-01 | 1,865E-01 | 1,872E-01 | 1,813E-01 | 1,815E-01 | 1,842E-01 |
| PINNED | - | 4,489E-03 | 4,428E-03 | 4,445E-03 | 4,437E-03 | 4,448E-03 | 4,449E-03 |
| STREAMS | 1 | 4,390E-03 | 4,420E-03 | 4,379E-03 | 4,416E-03 | 4,382E-03 | 4,397E-03 |
| STREAMS | 2 | 3,469E-03 | 3,478E-03 | 3,464E-03 | 3,477E-03 | 3,457E-03 | 3,469E-03 |
| STREAMS | 4 | 2,975E-03 | 2,986E-03 | 2,966E-03 | 2,986E-03 | 2,971E-03 | 2,977E-03 |
| STREAMS | 8 | 2,849E-03 | 2,845E-03 | 2,869E-03 | 2,859E-03 | 2,800E-03 | 2,844E-03 |
| TILED | 8 | 2,640E-03 | 2,607E-03 | 2,638E-03 | 2,636E-03 | 2,649E-03 | 2,634E-03 |
| TILED (per 4) | 8 | 2,597E-03 | 2,601E-03 | 2,593E-03 | 2,595E-03 | 2,612E-03 | 2,600E-03 |

:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Версия программы | Число потоков | Замер 1, с | Замер 2, с | Замер 3, с | Замер 4, с | Замер 5, с | Среднее зн-е, с |
| CPU (база) | 12 (cpu) | 6,19E+00 | 6,17E+00 | 4,75E+00 | 4,83E+00 | 5,87E+00 | 5,562E+00 |
| CPU (AVX-512) | 12 (cpu) | 1,36E+00 | 1,44E+00 | 1,05E+00 | 1,32E+00 | 1,46E+00 | 1,326E+00 |
| База | - | 2,444E-01 | 2,314E-01 | 2,321E-01 | 2,289E-01 | 2,288E-01 | 2,331E-01 |
| PINNED | - | 3,703E-02 | 3,711E-02 | 3,694E-02 | 3,707E-02 | 3,703E-02 | 3,704E-02 |
| STREAMS | 1 | 3,702E-02 | 3,694E-02 | 3,701E-02 | 3,702E-02 | 3,706E-02 | 3,701E-02 |
| STREAMS | 2 | 3,122E-02 | 3,116E-02 | 3,098E-02 | 3,124E-02 | 3,130E-02 | 3,118E-02 |
| STREAMS | 4 | 2,800E-02 | 2,824E-02 | 2,832E-02 | 2,822E-02 | 2,824E-02 | 2,820E-02 |
| STREAMS | 8 | 2,652E-02 | 2,669E-02 | 2,659E-02 | 2,666E-02 | 2,682E-02 | 2,666E-02 |
| TILED | 8 | 1,983E-02 | 1,984E-02 | 1,980E-02 | 1,980E-02 | 1,978E-02 | 1,981E-02 |
| TILED (per 4) | 8 | 1,698E-02 | 1,693E-02 | 1,693E-02 | 1,703E-02 | 1,700E-02 | 1,697E-02 |

:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Версия программы | Число потоков | Замер 1, с | Замер 2, с | Замер 3, с | Замер 4, с | Замер 5, с | Среднее зн-е, с |
| CPU (база) | 12 (cpu) |  |  |  |  |  | #ДЕЛ/0! |
| CPU (AVX-512) | 12 (cpu) |  |  |  |  |  | #ДЕЛ/0! |
| База | - | 2,476E+00 | 2,502E+00 | 2,446E+00 | 2,471E+00 | 2,483E+00 | 2,476E+00 |
| PINNED | - | 1,529E+00 | 1,544E+00 | 1,541E+00 | 1,556E+00 | 1,540E+00 | 1,542E+00 |
| STREAMS | 1 | 1,531E+00 | 1,529E+00 | 1,572E+00 | 1,563E+00 | 1,553E+00 | 1,549E+00 |
| STREAMS | 2 | 1,422E+00 | 1,422E+00 | 1,424E+00 | 1,426E+00 | 1,445E+00 | 1,428E+00 |
| STREAMS | 4 | 1,394E+00 | 1,391E+00 | 1,385E+00 | 1,393E+00 | 1,376E+00 | 1,388E+00 |
| STREAMS | 8 | 1,366E+00 | 1,352E+00 | 1,357E+00 | 1,352E+00 | 1,358E+00 | 1,357E+00 |
| TILED | 8 | 9,449E-01 | 9,379E-01 | 9,449E-01 | 9,808E-01 | 9,447E-01 | 9,506E-01 |
| TILED (per 4) | 8 | 8,365E-01 | 8,504E-01 | 8,323E-01 | 8,357E-01 | 8,385E-01 | 8,387E-01 |

# Приложение 2. Влияние размера блока

:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Версия программы | Число потоков | Размер блока | Замер 1, с | Замер 2, с | Замер 3, с | Замер 4, с | Замер 5, с | Среднее зн-е, с |
| STREAMS TILED | 1 | 2 | 0,002238 | 0,002319 | 0,002242 | 0,002232 | 0,002227 | 0,002252 |
| STREAMS TILED | 1 | 4 | 0,001825 | 0,001086 | 0,001 | 0,001008 | 0,001037 | 0,001191 |
| STREAMS TILED | 1 | 8 | 0,001119 | 0,000923 | 0,000832 | 0,010761 | 0,000838 | 0,002895 |
| STREAMS TILED | 1 | 12 | 0,001309 | 0,000827 | 0,00082 | 0,000826 | 0,002631 | 0,001283 |
| STREAMS TILED | 1 | 16 | 0,000945 | 0,002261 | 0,000802 | 0,000837 | 0,000791 | 0,001127 |
| STREAMS TILED | 1 | 24 | 0,001244 | 0,00082 | 0,00082 | 0,000813 | 0,000824 | 0,000904 |
| STREAMS TILED | 1 | 32 | 0,000894 | 0,00089 | 0,000813 | 0,000828 | 0,000834 | 0,000852 |

:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Версия программы | Число потоков | Размер блока | Замер 1, с | Замер 2, с | Замер 3, с | Замер 4, с | Замер 5, с | Среднее зн-е, с |
| STREAMS TILED | 1 | 2 | 0,390271 | 0,376174 | 0,373965 | 0,373986 | 0,374025 | 0,377684 |
| STREAMS TILED | 1 | 4 | 0,098711 | 0,094195 | 0,093995 | 0,095333 | 0,094079 | 0,095263 |
| STREAMS TILED | 1 | 8 | 0,055884 | 0,045877 | 0,046091 | 0,046332 | 0,050658 | 0,048968 |
| STREAMS TILED | 1 | 12 | 0,035226 | 0,03363 | 0,033801 | 0,03598 | 0,039365 | 0,0356 |
| STREAMS TILED | 1 | 16 | 0,032584 | 0,037983 | 0,030399 | 0,031958 | 0,041386 | 0,034862 |
| STREAMS TILED | 1 | 24 | 0,033404 | 0,033441 | 0,034927 | 0,033872 | 0,033387 | 0,033806 |
| STREAMS TILED | 1 | 32 | 0,034602 | 0,029702 | 0,029634 | 0,029659 | 0,029591 | 0,030638 |