***Василенко Анатолий, 621 группа (3-й поток)***
Исходный код: [https://github.com/phonexicum/DHP-PE-RA-FDM]
2 ветви:

1. master – mpi+openmp – 2-е задание суперпрактикума
2. mpi-cuda – 3-е задание практикума

Номер варианта: (2-й в списке группы) → 2 mod 18 + 18* (2 mod 2) = 2
Первый набор функций, равномерная сетка, максимум норма.

# Оглавление:



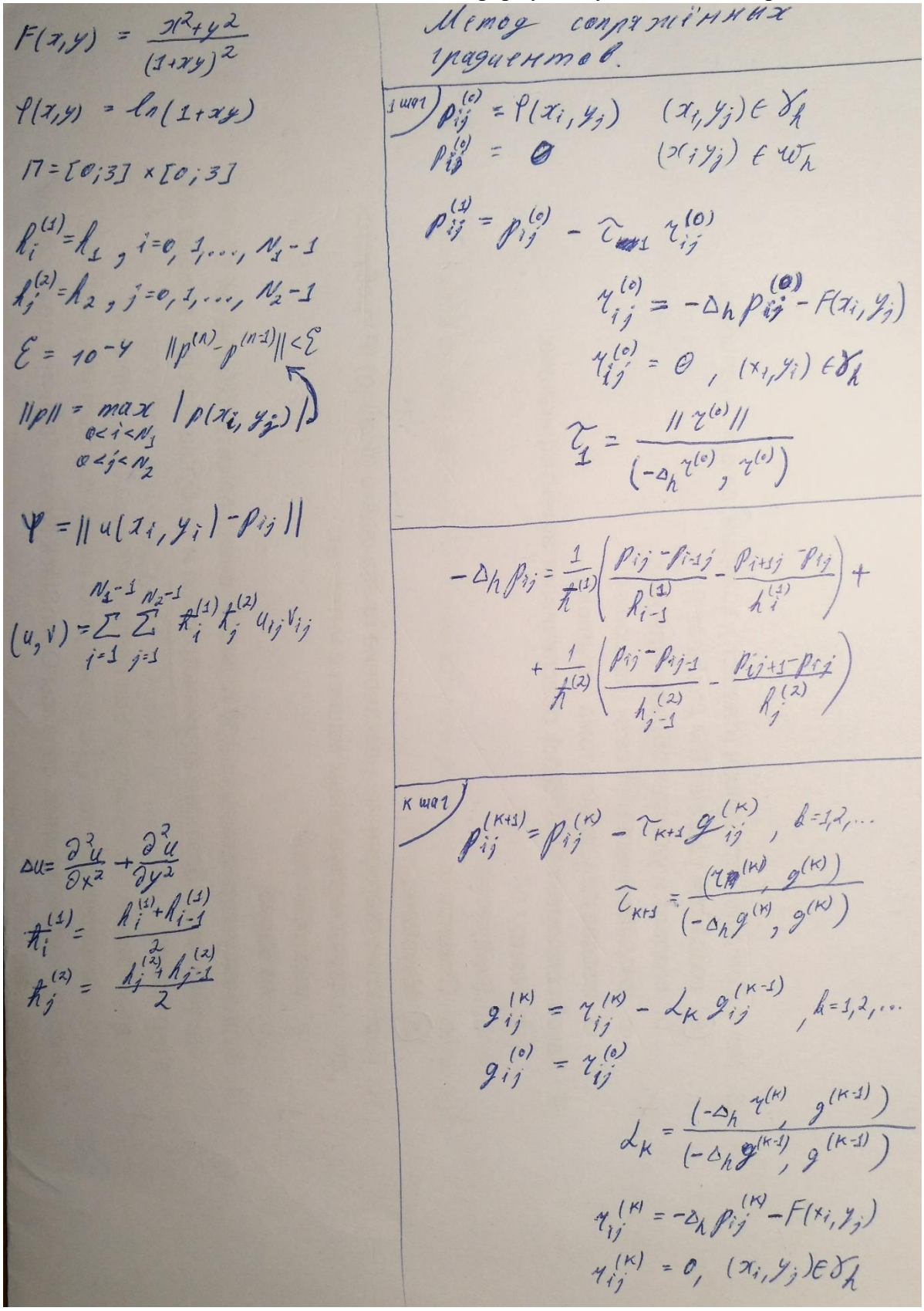
# 1. Сборка проекта

Есть Makefile,
lmount или jmount – подмонтируют в папку mount файловую систему нужного суперкомпьютера.
make upload – скопирует все нужные файлы (все .cpp, .h, .cu) в подмонтированную файловую систему.
На суперкомпьютере тем же make-файлом можно скомпилировать программу
make lcompile или make jcompile или make jcompile-omp

В коде есть некоторое количество комментариев, но ещё там все имена – user-friendly.
Немного об особенностях используемых технологий (особенно cuda) можно прочитать ниже в параграфе «Основные особенности при использовании технологий параллелизации».

# 2. Математическая постановка задания

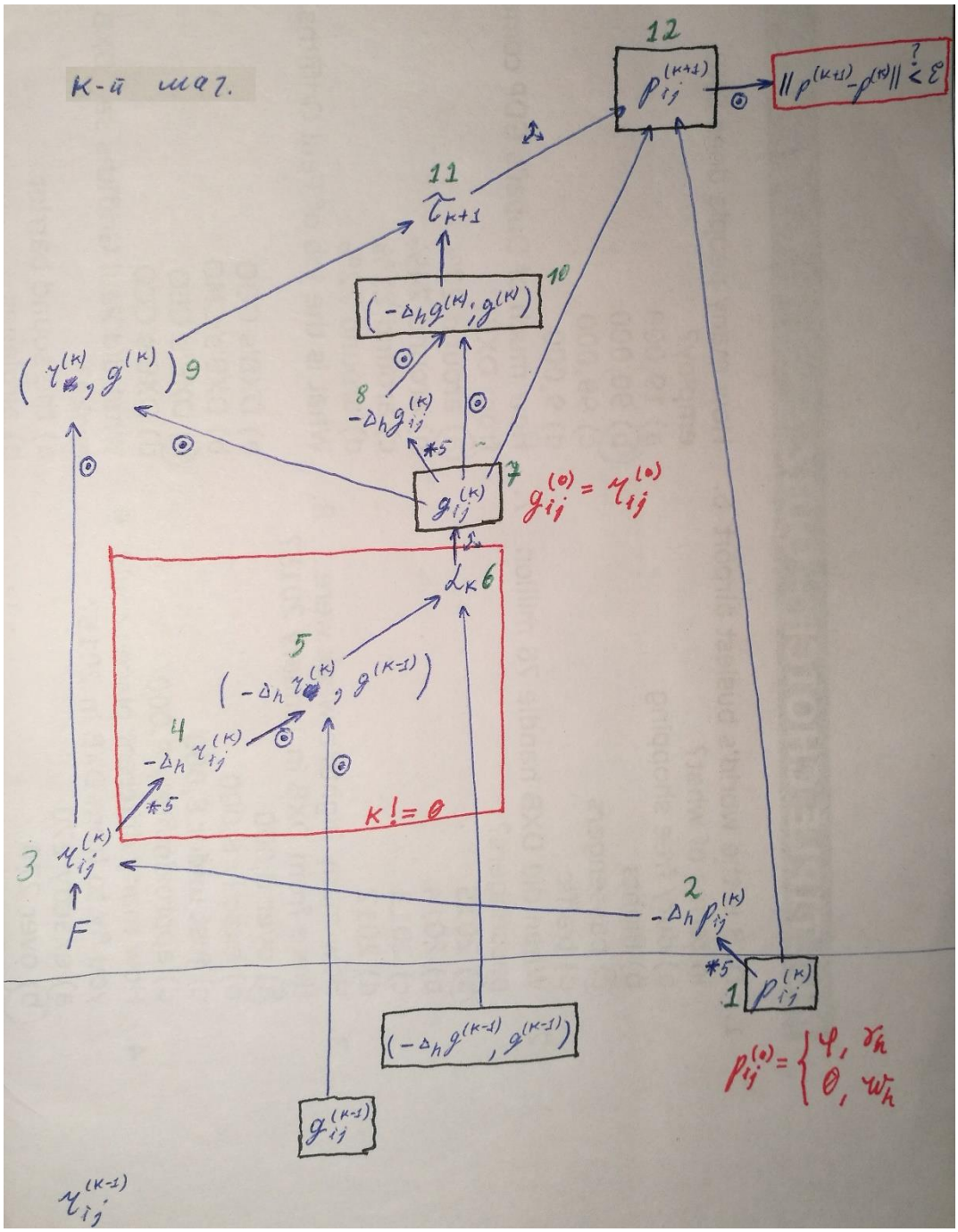
Все необходимые математические формулы приведены в постановке задачи, выданной студентам. Можно выписать основные математические формулы с учётом моего варианта задания:



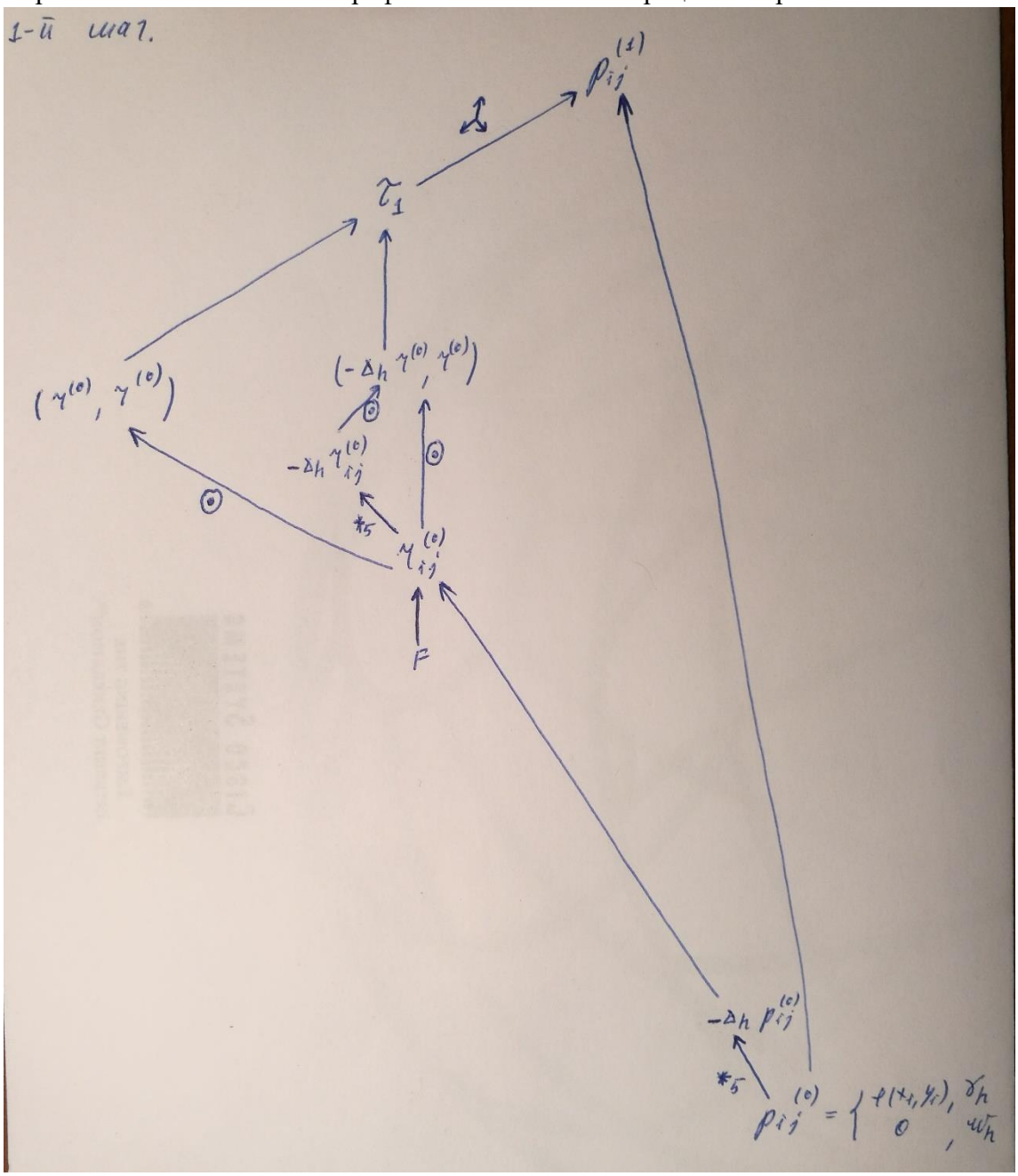
# 3. Инфограф

Картинка ниже ***наглядно*** показывает зависимости между данными с учётом их распределения между процессами, при вычислении k-й итерации алгоритма.

1. Красным выделены начальные состояния и проверка критерия останова
2. Чёрным выделены функции, используемые не только на своём текущем шаге, но и используемые для следующей итерации
3. Зелёным цветом пронумерован *некоторый* порядок вычисления операций
4. При вычислении скалярных произведений и критерия останова необходимо агрегировать данные со всех процессов.
5. После вычисления коэффициентов \tao и \alpha необходимо их разослать между всеми процессами
6. При вычислении delta функций, необходимо пересылать данные между соседними процессами, для вычисления пятиточечной аппроксимации



Картинка ниже показывает граф вычисления 1-й итерации алгоритма:




Нетрудно видеть, что 1-я итерация может быть легко сведена к k-й итерации, если исключить вычисление 4, 5, 6 операций, а в качестве \alpha взять “1”.

Основные операции которые необходимо уметь вычислять:

1. Вычислить скалярное произведение.
2. Вычислить 5-ти точечную аппроксимацию.
3. Разослать коэффициент между всеми процессами.
4. Вычислить значение некоторой функции в точке, при этом вычисление в данной точке основано на данных, уже имеющихся у процесса.
5. Проверить критерий останова.

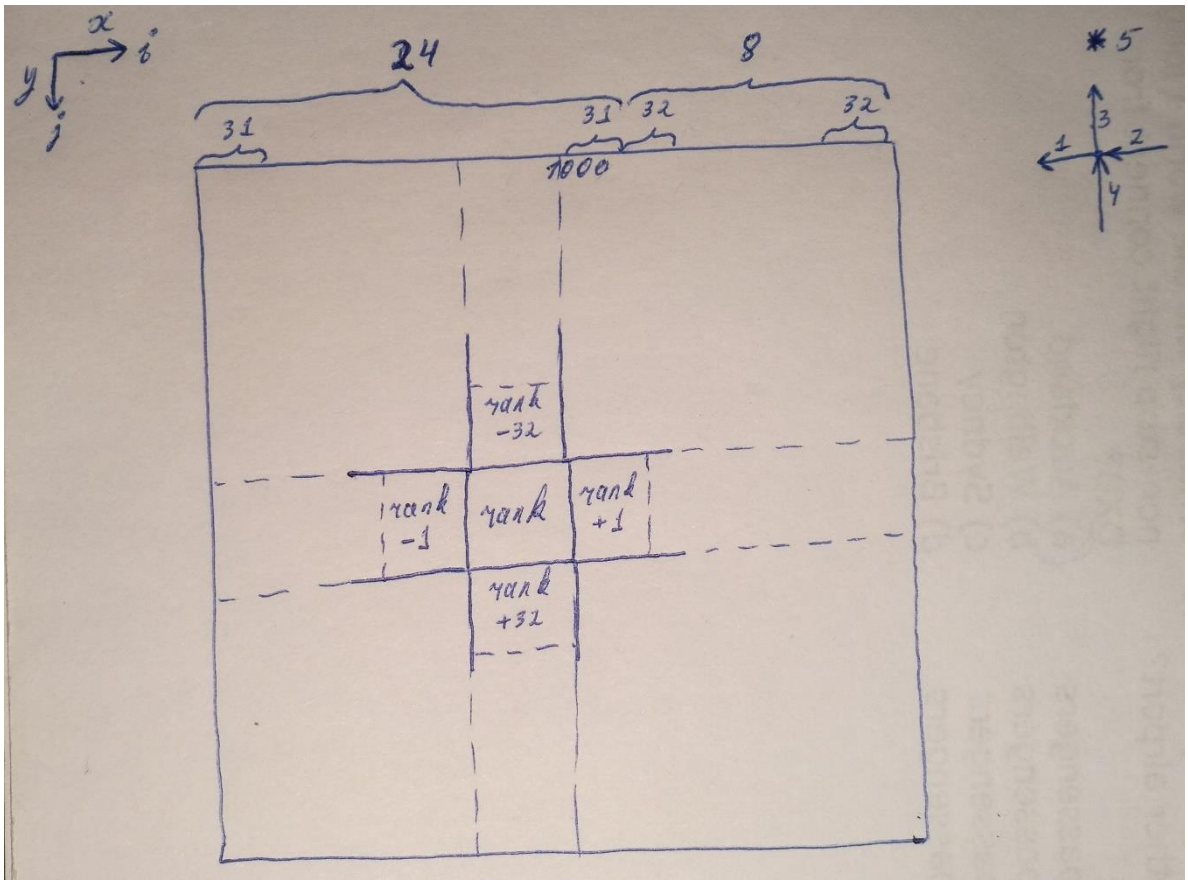
# 4. Распределение задач между процессами

Область рассчётов разбивается на прямоугольники. Каждому прямоугольнику сопоставляется процесс, который будет его рассчитывать. И в случае необходимости запрашивать и предоставлять соседям данные (а также возможно агрегирующему процессу)

Разделение происходит следующим образом:

1. Выбирается, разбиение по количеству процессов по вертикали и горизонтали (например, если всего дано 512 процессов, то по горизонтали можно выделить 16, а по вертикали – 32 процесса) (соотношение между процессами не должно превышать 2:1, чтобы результирующая область, сопоставленная с каждым из процессов, имела соотношение не более 2:1)
2. вычисляется сколько клеток приходится на заданное количество процессов, в случае если нацело не делится, то остаток распределяется равномерно (по 1-му элементу на каждого) между последними процессами (это возможно т.к. остаток всегда меньше делителя)

Картинка ниже *наглядно* показывает пример при распределении 1000 столбцов между соответствующими 32 процессами:




Аналогичное разбиение применяется и по вертикали.
Данная картинка сделана для наглядности и для частного случая, в программе – разбиение происходит полностью автоматически в зависимости от заданного числа процессов по вертикали и горизонтали, а также в зависимости от размера сетки.

# 5. Основные особенности при использовании технологий параллелизации

## 5.1.Использование MPI

В момент инициализации вычислений, все участвующие процессы объединяются в свой отдельный коммуникатор, в рамках которого происходят операции send/recv и broadcast (MPI_Reduce) операции.
Соседние процессы многократно пересылают друг другу массивы данных (значения граничных элементов области). Для это используются *асинхронные* send/recv.
Для подсчёта нормы (максимального элемента среди множества других элементов), изначально процесс вычисляет значения максимального элемента для своей области, после чего используется MPI_Reduce. Аналогично для подсчёта общей суммы в составе скалярного произведения, также используется MPI_Reduce.

## 5.2.Распределение задач между ядрами (OpenMP)

В рамках вычислений часто необходимо производить независимые операции для каждого узла сетки.

Возможность распараллеливания между нитями присутствует в следующих операциях:

1. Вычисление скалярного произведения (aggregate sum)
2. Вычисление 5-ти точечной аппроксимации
    a. вычисление новой функции дельта, на основе функции данной ранее во внутренних точках, что позволяет вычислять значение каждой ячейки независимо)
    b. распределённые между нитями различные независимые участки кода (подготовка данных к отправке посредством MPI другим процессам, и обработка принятых данных)
3. Вычисление критерия останова (нахождение нормы (в моём случае – нахождение максимума из множества чисел))
4. Вычисление функций r, g, p – т.к. каждый раз вычисление каждой ячейки – это независимая операция.

Основные используемые возможности openmp: распараллеливание циклов (#pragma omp for) и разбиение независимых вычислительных задач на секции между несколькими нитями.
Также используется reduce для агрегации результата между нитями, однако на BlueGene работает reduce для сложения, и не работает для находения max (по-видимому не поддерживается компилятором intel – «mpixlcxx_r»), поэтому для вычисления глобального максимума пришлось задействовать критическую секцию.

Согласно заданию – каждый mpi процесс запускался отдельно на 4-х ядреном процессоре и разбивал себя на 3 openmp нити.

## 5.3.Использование технологии CUDA

На Ломоносове (с видеокартой Tesla X2070) поддерживаются следующие технологии: MemoryMapping и UAC. (полный перечень TTX можно посмотреть в README git-репозитория).

Все массивы данных, используемые для вычислений изначально создаются в памяти графической карты, после чего все вычисления над ними производятся лишь через вычислительные ядра.
Основные типы вычислительных ядер:

1. Независимое вычисление каждой ячейки в матрице (хорошо параллелится между нитями)
2. Подсчёт суммы (или максимума) в массиве. Для этого реализовано ядро с логарифмической сложностью, основанное на том, что нити итерационно складывают (находят максимум) значений по степеням двойки.

3. Копирование вертикального столбца данных из матрицы (необходимо для формирования буфера данных для дальнейшей MPI пересылки)

Буфера данных, для MPI приёма-передачи выделяются с использованием MemoryMapping (т.е. без механизма виртуальных функций). Это сделано для того, чтобы видеокарта могла записывать (считывать) туда данные, формируя массив для отправки (приёма) сообщений по MPI. Это позволяет записывать (считывать) их не используя ресурс центрального процессора (это сделает за нас контроллер через PCI шину (т.к. asyncEngine=2)).
Явного копирования этих буферов в память графической карты не производится, так как доступ к ним на запись/чтение не является многократным.

Так как некоторые ядра обрабатывают отдельно границы области, а не всю область целиком, то они могут не заполнять все мультипроцессоры, поэтому для ускорения используются потоки.
Количество пересылок памяти мало по сравнению с вычислительными затратами, поэтому в первую очередь необходимо стараться заполнить весь ресурс видеокарты (все мультипроцессоры) вычислительными ядрами (т.е. по возможности их нужно вешать на разные потоки выполнения).

Рассчёт размера грида и размера блоков:

1. Размер блока = 512, по 2-м причинам:
    a. На один мультипроцессор влазит ровно 3 блока
    b. Блок максимального размера в 1024 нитей требует регистров в данной реализции больше, чем доступно на видеокарте.
2. Я рассматриваю вычисление значения в каждой точке сетки - как отдельное вычисление, предназначенное для одного из ядер мультипроцессора. Однако количество мультипоцессоров =14, а максимальное количество нитей в 1-м мультипроцессоре =1536, таким образом я могу одновременно запустить максимум 21504 нити, которые будут соответственно обрабатывать столько же узлов сетки. Однако, если я одной видеокартой область, например, 2000x2000, то у меня узлов сетки будет 4 миллиона, что больше, чем 21504, и поэтому я делю одно число на другое и тем самым вычисляю, сколько узлов сетки приходится на 1 нить, чтобы количество нитей после этого уложилось в 21504.
   Этим вычислением занимается класс GridDistribute.
   При этом при выполнении в данном случае соседние нити будут брать подряд идущие данные, что должно благотворно влиять на запросы к памяти, т.к. они будут распределены к разным банкам памяти.

# 6. Результаты тестирования на суперкомпьютерах

Программное средство (вместе с benchmark результатами) может быть найдено по адресу:
[https://github.com/phonexicum/DHP-PE-RA-FDM]
(Реализация программы для CUDA находится в ветке «mpi-cuda»)

*Через двоеточие в таблице указаны минуты и секунды.*

При запуске mpi процессов для тестирования CUDA-программы, планировщику задач ломоносова указывалось, чтобы он размещал по 2 процесса на 1-м узле, т.к. на каждом узле ломоносова находится 2 видеокарты.

Вычисления для более, чем 16 GPU карт невозможно провести, используя быстродвижущуюся очередь gputest, поэтому производительность исследовалась для количества видеокарт не более, чем 16, что вполне приемлемо, так как программа выполняется слишком быстро.

Вычисление для 1-го mpi процесса проводилось той же программой, что и вычисление для p процессов. Отдельная последовательная реализации отсутствует.

Результаты вычислений программы, основанной на каждой из технологий успешно проверялись на корректность.

Компиляция на BlueGene/P производилась компилятором IBM для версии с openmp, и компилятором g++ для версии без openmp.

## 6.1.Численные результаты

BlueGene/P (mpi) (с оптимизациями компилятора «-O3»)

| число процессоров | разбиение сетки | время решения (мс) | ускорение | число итераций |
|---|---|---|---|---|
| 1 | 1000x1000 | 877509 мс = 14:37 | 1 раз | 1416 |
| 2 | 1000x1000 | 439510 мс = 7:19 | 2.00 раза | 1416 |
| 128 | 1000x1000 | 7330 мс = 0:7 | 119.71 раз | 1416 |
| 256 | 1000x1000 | 3884 мс = 0:3 | 225.93 раза | 1416 |
| 512 | 1000x1000 | 2100 мс = 0:2 | 417.86 раза | 1416 |
| | | | | |
| 1 | 2000x2000 | 7090953 мс = 118:10 | 1 раз | --- |
| 2 | 2000x2000 | 3547949 мс = 59:7 | 2.00 раз | 2828 |
| 128 | 2000x2000 | 57079 мс = 0:57 | 124.23 раза | 2828 |
| 256 | 2000x2000 | 28838 мс = 0:28 | 245.89 раза | 2828 |
| 512 | 2000x2000 | 14626 мс = 0:14 | 484.82 раза | 2828 |

BlueGene/P (mpi) (без оптимизаций компилятора «-O3»)

| число процессоров | разбиение сетки | время решения (мс) | ускорение | число итераций |
|---|---|---|---|---|
| 1 | 1000x1000 | 3015294 мс = 50:15 | 1 раз | 1416 |
| 2 | 1000x1000 | 1297439 мс = 21:37 | 2.32 раза | 1416 |
| 128 | 1000x1000 | 24258 мс = 0:24 | 124 раза | 1416 |
| 256 | 1000x1000 | 12416 мс = 0:12 | 243 раза | 1416 |
| 512 | 1000x1000 | 6495 мс = 0:6 | 464 раза | 1416 |
| | | | | |
| 1 | 2000x2000 | *превышен лимит 2-х часов* | --- | --- |
| 2 | 2000x2000 | *превышен лимит 2-х часов* | --- | --- |
| 128 | 2000x2000 | 190366 мс = 3:10 | 1 раз | 2828 |
| 256 | 2000x2000 | 95653 мс = 1:35 | 1.99 раза | 2828 |

| | | | | |
|---|---|---|---|---|
| 512 | 2000x2000 | 48431 мс = 0:48 | 3.93 раза | 2828 |

BlueGene/P (mpi + openmp (режим openmp использовался с 3-мя нитями на 4-х ядерном процессоре)) (с оптимизациями компилятора «-O3»)

| число процессоров | разбиение сетки | время решения (мс) | ускорение | число итераций |
|---|---|---|---|---|
| 1-omp (x3) | 1000x1000 | 204639 мс = 3:24 | 1 раз | 1416 |
| 2-omp (x3) | 1000x1000 | 103466 мс = 1:43 | 1.98 раза | 1416 |
| 128-omp (x3) | 1000x1000 | 2784 мс = 0:2 | 73.51 раза | 1416 |
| 256-omp (x3) | 1000x1000 | 1988 мс = 0:1 | 102.94 раза | 1416 |
| 512-omp (x3) | 1000x1000 | 1602 мс = 0:1 | 127.74 раза | 1416 |
| | | | | |
| 1-omp (x3) | 2000x2000 | 1691049 мс = 28:11 | 1 раз | 2828 |
| 2-omp (x3) | 2000x2000 | 847575 мс = 14:7 | 2 раза | 2828 |
| 128-omp (x3) | 2000x2000 | 15670 мс = 0:15 | 107.92 раза | 2828 |
| 256-omp (x3) | 2000x2000 | 8821 мс = 0:8 | 191.71 раза | 2828 |
| 512-omp (x3) | 2000x2000 | 5541 мс = 0:5 | 305.19 раз | 2828 |

BlueGene/P (mpi + openmp (режим openmp использовался с 3-мя нитями на 4-х ядерном процессоре)) (без оптимизаций компилятора «-O3»)

| число процессоров | разбиение сетки | время решения (мс) | ускорение | число итераций |
|---|---|---|---|---|
| 1-omp | 1000x1000 | 262765 мс = 4:22 | 1 раз | 1416 |
| 2-omp | 1000x1000 | 132624 мс = 2:12 | 1.98 раза | 1416 |
| 128-omp | 1000x1000 | 4880 мс = 0:4 | 53.85 раза | 1416 |
| 256-omp | 1000x1000 | 3029 мс = 0:3 | 86.75 раза | 1416 |
| 512-omp | 1000x1000 | 2100 мс = 0:2 | 125.13 раз | 1416 |
| | | | | |
| 1-omp | 2000x2000 | 2137275 мс = 35:37 | 1 раз | 2828 |
| 2-omp | 2000x2000 | 1070854 мс = 17:50 | 2.00 раза | 2828 |
| 128-omp | 2000x2000 | 31682 мс = 0:31 | 67.46 раза | 2828 |
| 256-omp | 2000x2000 | 16936 мс = 0:16 | 126.20 раз | 2828 |
| 512-omp | 2000x2000 | 9653 мс = 0:9 | 221.41 раз | 2828 |

Lomonosov (без оптимизаций компилятора «-O3»)

| число процессоров | разбиение сетки | время решения (мс) | ускорение | число итераций |
|---|---|---|---|---|
| 1 | 1000x1000 | 309562 мс = 5:9 | 1 раз | 1416 |
| 8 | 1000x1000 | 39096 мс = 0:39 | 7.92 раза | 1416 |
| 16 | 1000x1000 | 19487 мс = 0:19 | 15.89 раза | 1416 |
| 32 | 1000x1000 | 9797 мс = 0:9 | 31.60 раз | 1416 |
| 64 | 1000x1000 | 5027 мс = 0:5 | 61.58 раза | 1416 |
| 128 | 1000x1000 | 2667 мс = 0:2 | 116.07 раза | 1416 |
| | | | | |
| 1 | 2000x2000 | 2470405 мс = 41:10 | 1 раз | 2828 |
| 8 | 2000x2000 | 311965 мс = 5:11 | 7.92 раза | 2828 |
| 16 | 2000x2000 | 156656 мс = 2:36 | 15.77 раза | 2828 |
| 32 | 2000x2000 | 78854 мс = 1:18 | 31.33 раза | 2828 |
| 64 | 2000x2000 | 39334 мс = 0:39 | 62.81 раза | 2828 |
| 128 | 2000x2000 | 19924 мс = 0:19 | 124.00 раза | 2828 |

Lomonosov, CUDA (без оптимизаций компилятора «-O3»)

| число процессоров | разбиение сетки | время решения (мс) | ускорение | число итераций |
|---|---|---|---|---|
| 1 | 1000x1000 | 7721 мс = 0:7 | 1 раз | 1416 |
| 8 | 1000x1000 | 4920 мс = 0:4 | 1.57 раза | 1416 |
| 16 | 1000x1000 | 5368 мс = 0:5 | 1.44 раза | 1416 |
| 32 | 1000x1000 | 5806 мс = 0:5 | 1.33 раза | 1416 |

| | | | | |
|---|---|---|---|---|
| 1 | 2000x2000 | 56059 мс = 0:56 | 1 раз | 2828 |
| 8 | 2000x2000 | 13778 мс = 0:13 | 4.07 раза | 2828 |
| 16 | 2000x2000 | 12316 мс = 0:12 | 4.55 раза | 2828 |
| 32 | 2000x2000 | 11799 мс = 0:11 | 4.75 раза | 2828 |

Lomonosov, CUDA (с оптимизациями компилятора «-O3»)

| число процессоров | разбиение сетки | время решения (мс) | ускорение | число итераций |
|---|---|---|---|---|
| 1 | 1000x1000 | 6161 мс = 0:6 | 1 раз | 1416 |
| 2 | 1000x1000 | 3496 мс = 0:3 | 1.76 раза | 1416 |
| 3 | 1000x1000 | 2575 мс = 0:2 | 2.39 раза | 1416 |
| 4 | 1000x1000 | 2116 мс = 0:2 | 2.91 раза | 1416 |
| 5 | 1000x1000 | 1837 мс = 0:1 | 3.35 раза | 1416 |
| 6 | 1000x1000 | 1774 мс = 0:1 | 3.47 раза | 1416 |
| **7** | **1000x1000** | **1603 мс = 0:1** | **3.8434 раза** | **1416** |
| 8 | 1000x1000 | 1605 мс = 0:1 | 3.8386 раза | 1416 |
| | | | | |
| 1 | 2000x2000 | 47668 мс = 0:47 | 1 раз | 2828 |
| 2 | 2000x2000 | 24922 мс = 0:24 | 1.91 раза | 2828 |
| 3 | 2000x2000 | 17368 мс = 0:17 | 2.74 раза | 2828 |
| 4 | 2000x2000 | 13156 мс = 0:13 | 3.62 раза | 2828 |
| 5 | 2000x2000 | 10884 мс = 0:10 | 4.38 раза | 2828 |
| 6 | 2000x2000 | 9287 мс = 0:9 | 5.13 раза | 2828 |
| 7 | 2000x2000 | 8193 мс = 0:8 | 5.82 раза | 2828 |
| 8 | 2000x2000 | 7135 мс = 0:7 | 6.68 раза | 2828 |
| 9 | 2000x2000 | 6653 мс = 0:6 | 7.16 раз | 2828 |
| 10 | 2000x2000 | 6195 мс = 0:6 | 7.69 раз | 2828 |
| 11 | 2000x2000 | 5749 мс = 0:5 | 8.29 раз | 2828 |
| 12 | 2000x2000 | 5393 мс = 0:5 | 8.84 раза | 2828 |
| 13 | 2000x2000 | 5131 мс = 0:5 | 9.29 раз | 2828 |
| 14 | 2000x2000 | 5112 мс = 0:5 | 9.32 раза | 2828 |
| 15 | 2000x2000 | 4904 мс = 0:4 | 9.72 раза | 2828 |
| 16 | 2000x2000 | 4495 мс = 0:4 | 10.60 раз | 2828 |
| | | | | |
| 1 | 5000x5000 | 780712 мс = 13:0 | 1 раз | 7047 |
| 2 | 5000x5000 | 405896 мс = 6:45 | 1.92 раза | 7047 |
| 3 | 5000x5000 | 274721 мс = 4:34 | 2.84 раза | 7047 |
| 4 | 5000x5000 | 197794 мс = 3:17 | 3.94 раза | 7047 |
| 5 | 5000x5000 | 164090 мс = 2:44 | 4.76 раза | 7047 |
| 6 | 5000x5000 | 131887 мс = 2:11 | 5.92 раза | 7047 |
| 7 | 5000x5000 | 117093 мс = 1:57 | 6.67 раз | 7047 |
| 8 | 5000x5000 | 98859 мс = 1:38 | 7.90 раз | 7047 |
| 15 | 5000x5000 | 53185 мс = 0:53 | 14.68 раз | 7047 |
| 16 | 5000x5000 | 50254 мс = 0:50 | 15.54 раза | 7047 |

## 6.2.Выводы из численных результатов

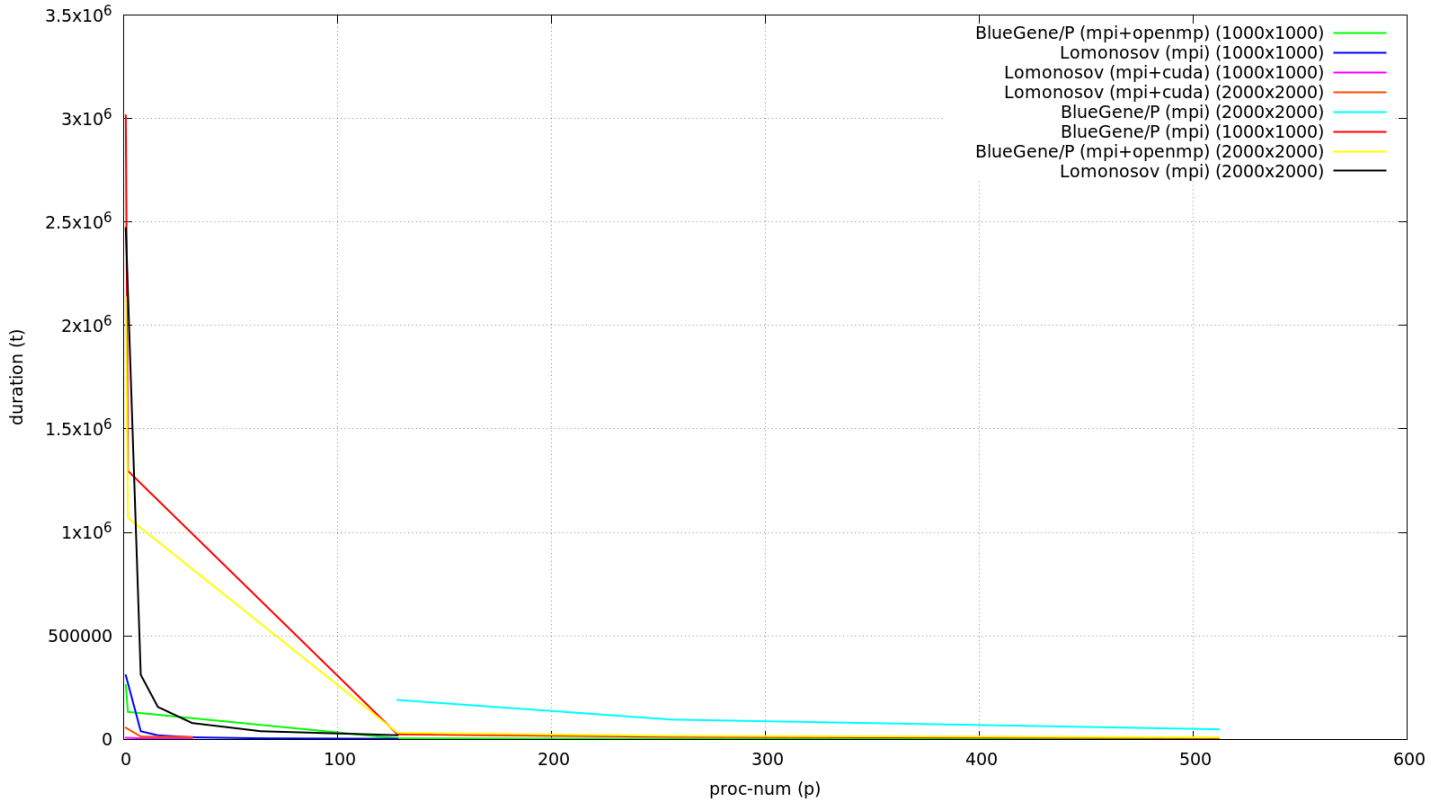
Эффективность на BlueGene в случае отключённых оптимизаций компилятора – мала и падает буквально в 2 раза, хоть и линейна, это проявляется только для данного суперкомпьютера и только при отключённых оптимизациях g++. Причина такого поведения не ясна и скорее всего как-то завязана на архитектуру данного суперкомпьютера и компилятора.

Эффективность (ускорение) показывает, что OpenMP даёт больший прирост производительности, нежели использование дополнительных процессоров. Это вполне ожидаемо, т.к. нити openmp используют разделяемую память и им не приходится обмениваться данными через MPI вызовы.

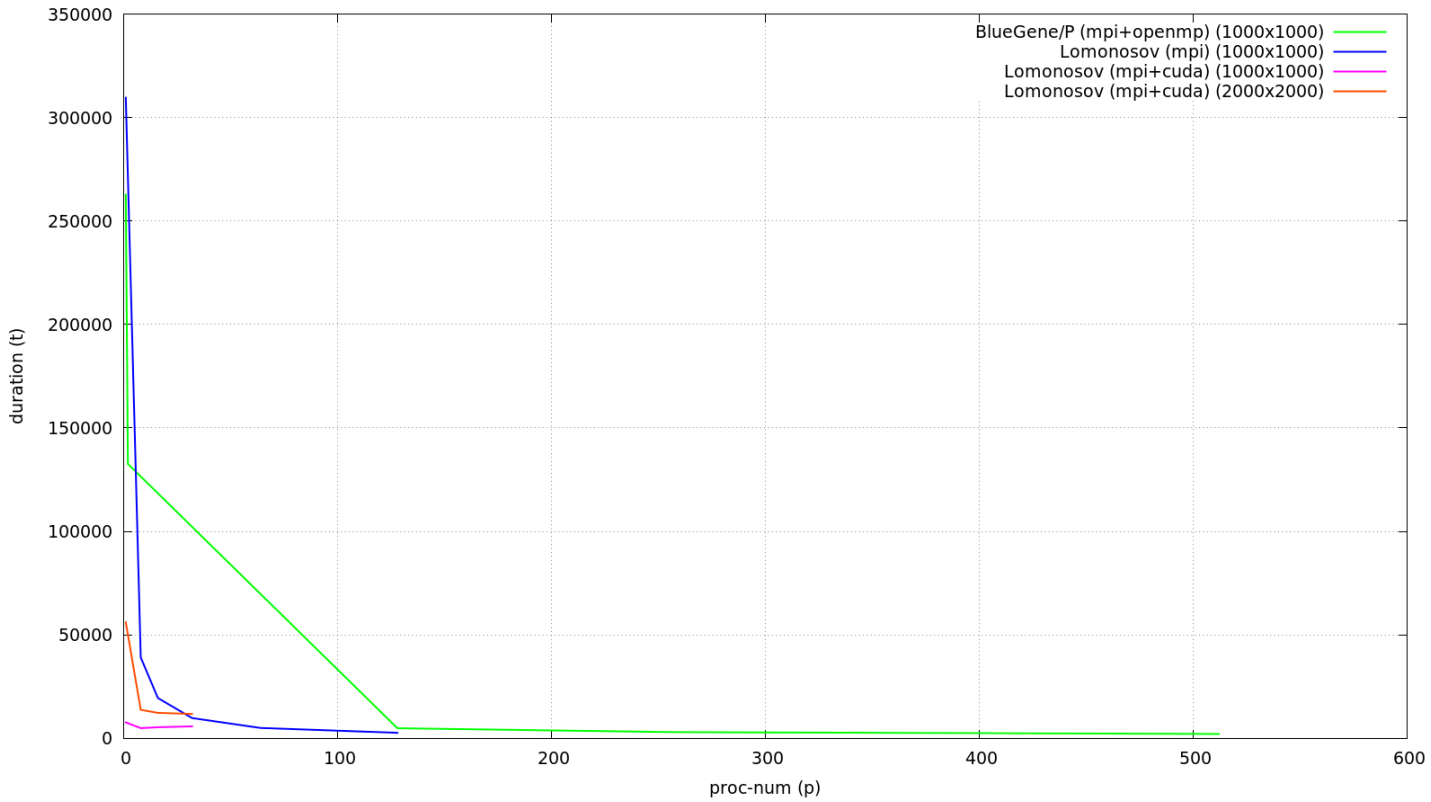
Эффективность для технологии CUDA при малом размере задачи растёт крайне нелинейно, это связано с тем, что при увеличении количества вычислительных узлов, количество пересылок возрастает, в то время как количество вычислений на каждый узел – падает, это приводит к падению эффективности, так как существенная часть времени уходит на пересылки данных.
Данный эффект может наблюдаться для совершенно любых задач, однако для высокопроизводительной cuda и исследуемого размера задачи оно достигается уже на малом количестве узлов (так – оптимальная эффективность достигается для сетки 1000x1000 на 7-ми узлах)

График зависимости времени выполнения (в мс) от количества процессов для выполнения на разных архитектурах, разных матриц, с разными технологиями, при отключённых оптимизациях компилятора (без флага «-O3»)

Объединённый рисунок:




Визуализация 4-х самых быстрых составов (аппаратура - размер матрицы – использованная технология):




Стоит отметить, что BlueGene/P отличается крайне высокой стабильностью времени вычисления (выяснено практическим путём).

# 7. Точное решение диф. уравнения

Точное решение дифференциального уравнения совпадает с функцией граничного условия. Достаточно подставить u(x,y) = ln(1+x*y)

# 8. Графическое изображение функции, погрешности вычислений

## 8.1.Графическое изображение точного аналитического решения и вычисленного решения дифференциального уравнения

Рисунок приближённого решения для сетки размером 2000x2000 (визуально изображение совпадает при использовании любых, из приведённых выше технологий):

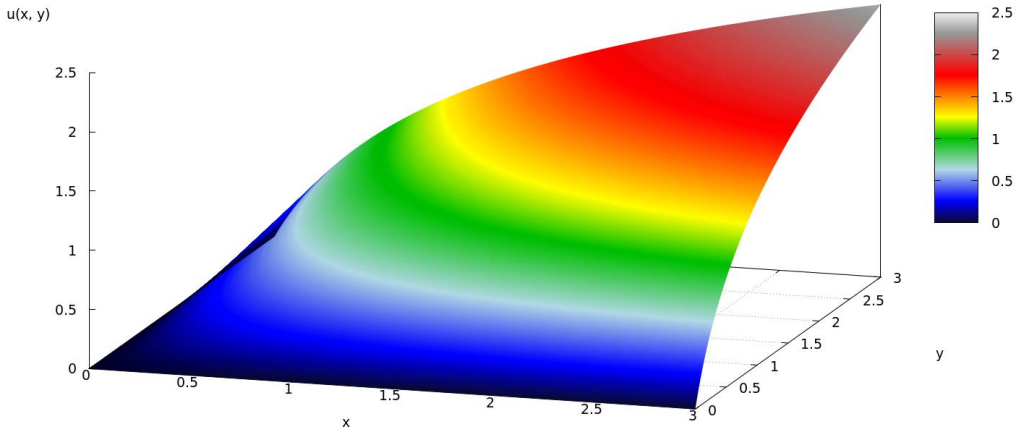
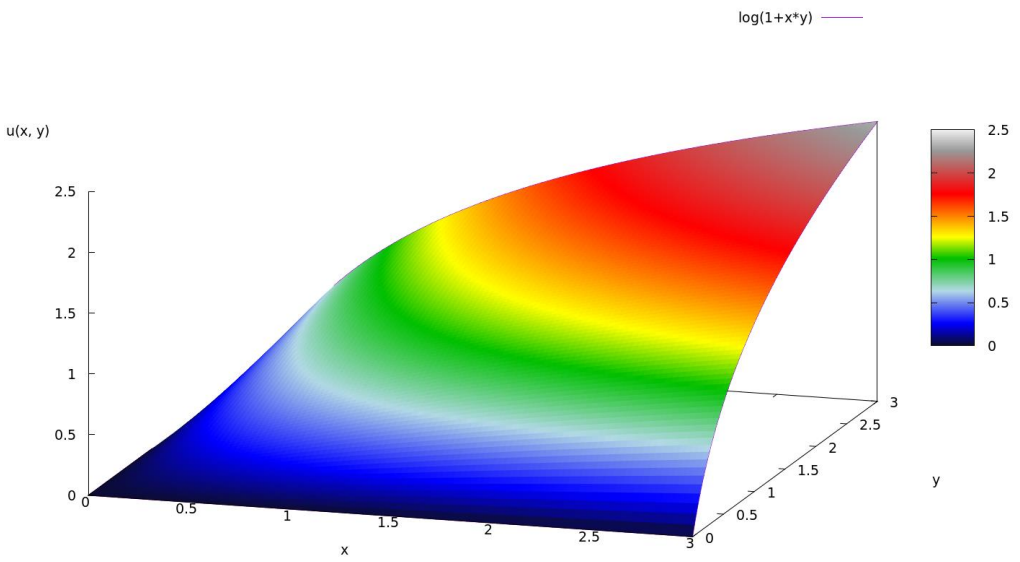



Рисунок точного решения (100x100 точек):



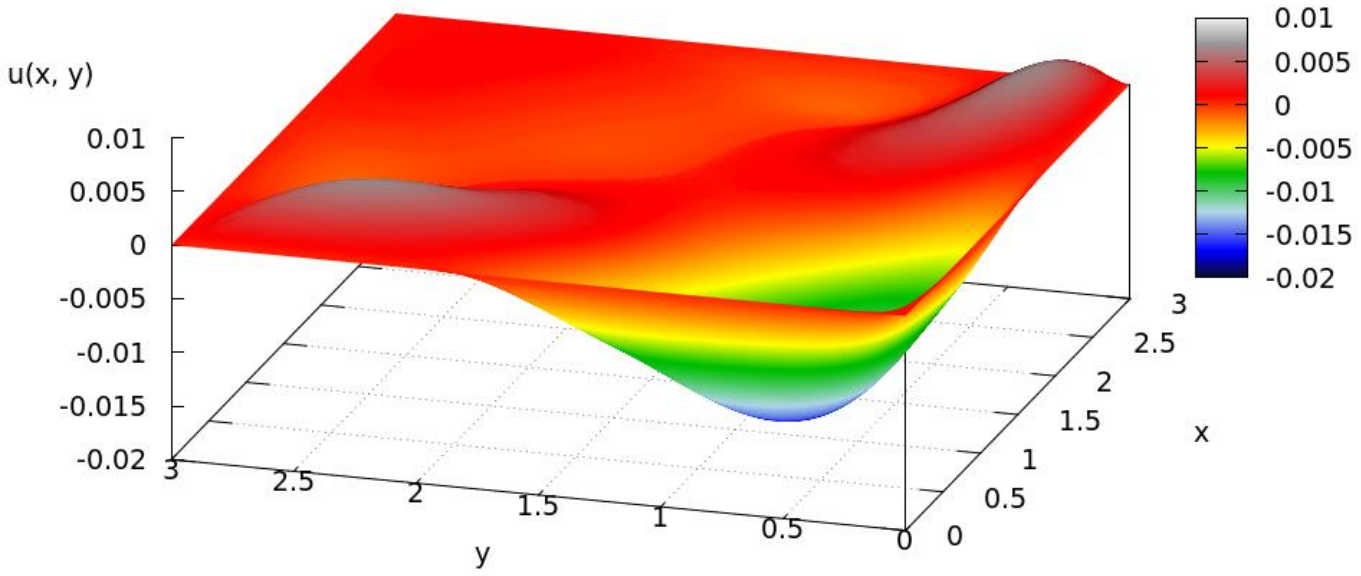
## 8.2.Графическое изображение абсолютной погрешности

Для всех архитектур (NVIDIA Tesla X2070, Lomonosov Intel Xeon X5570, BlueGene/P PowerPC 450) графические изображения визуально не отличаются.
Сравнивались изображения для решений, вычисленных со следующими конфигурациями:

- Ломоносов (nvidia tesla и intel xeon) – 32 процесса, матрица 1000x1000
- BlueGene/P (PowerPC) – 128 процессов, матрица 1000x1000

(количество процессов, для данной оценки не имеет значения)

Графическое изображение абсолютной погрешности:




Заметим, что несмотря на то, что рассчёты велись пока шаг итерации не изменит значение функции менее, чем на 0.0001, само итоговое решение имеет погрешность гораздо большую (на 2 порядка).

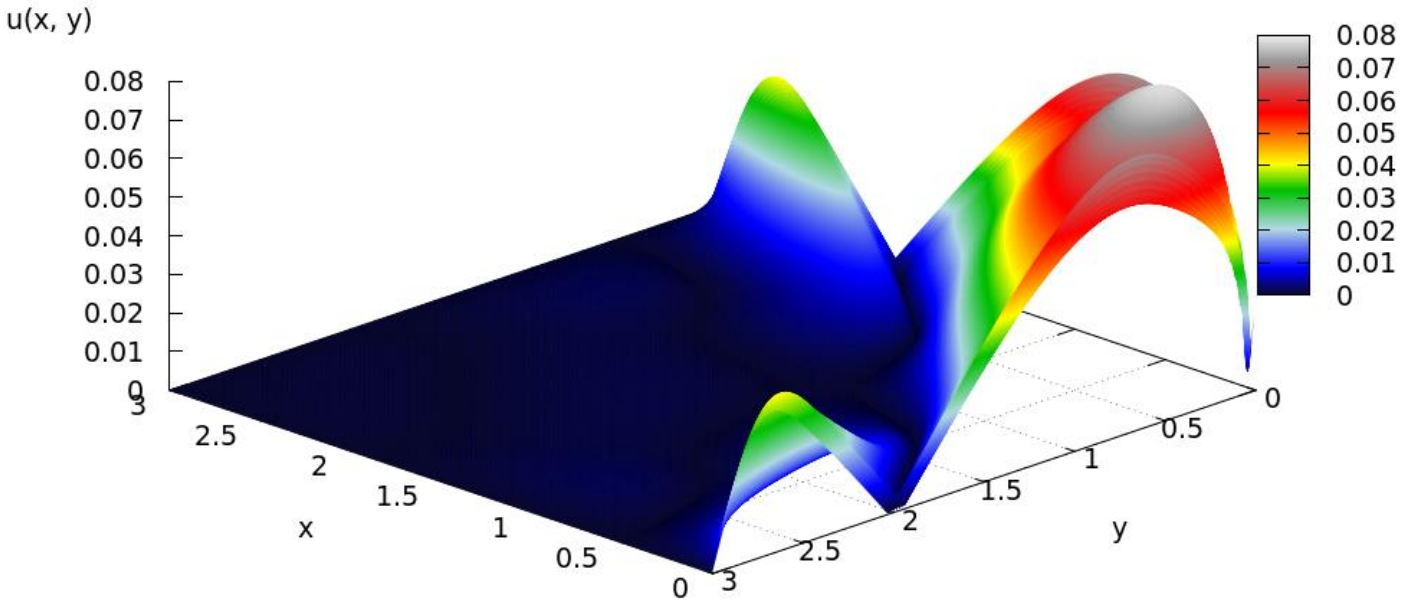
| Конфигурация | Макс. значение абс. погрешности | Мин. значение абс. погрешности |
|---|---|---|
| Lomonosov, p=32, n=1000 | 0.00652979267574 | -0.0155494946188 |
| Lomonosov (cuda), p=32, n=1000 | 0.0065297931977 | -0.015549494123 |
| BlueGene/P, p=128, n=1000 | 0.00652979267574 | -0.0155494946188 |

Из значений видно, что процессора nvidia считают числа с плавающей точкой, с другой точностью. Однако возможно это связано с тем, что некоторые операции видеокарта Tesla X2070 не может делать над числом типа double, преобразуя их в float (например, операция взятия логарифма).

## 8.3.Графическое изображение относительной погрешности

Конфигурации, для которых вычислялась относительная погрешность, полностью совпадают с конфигурациями, использованными для вычисления абсолютной погрешности.
Аналогично, разница в графическом представлении неотличима для различных конфигураций.

Графическое изображение относительной погрешности (граничные точки обрезаны, так как функция принимает в этих точках значение ноль):



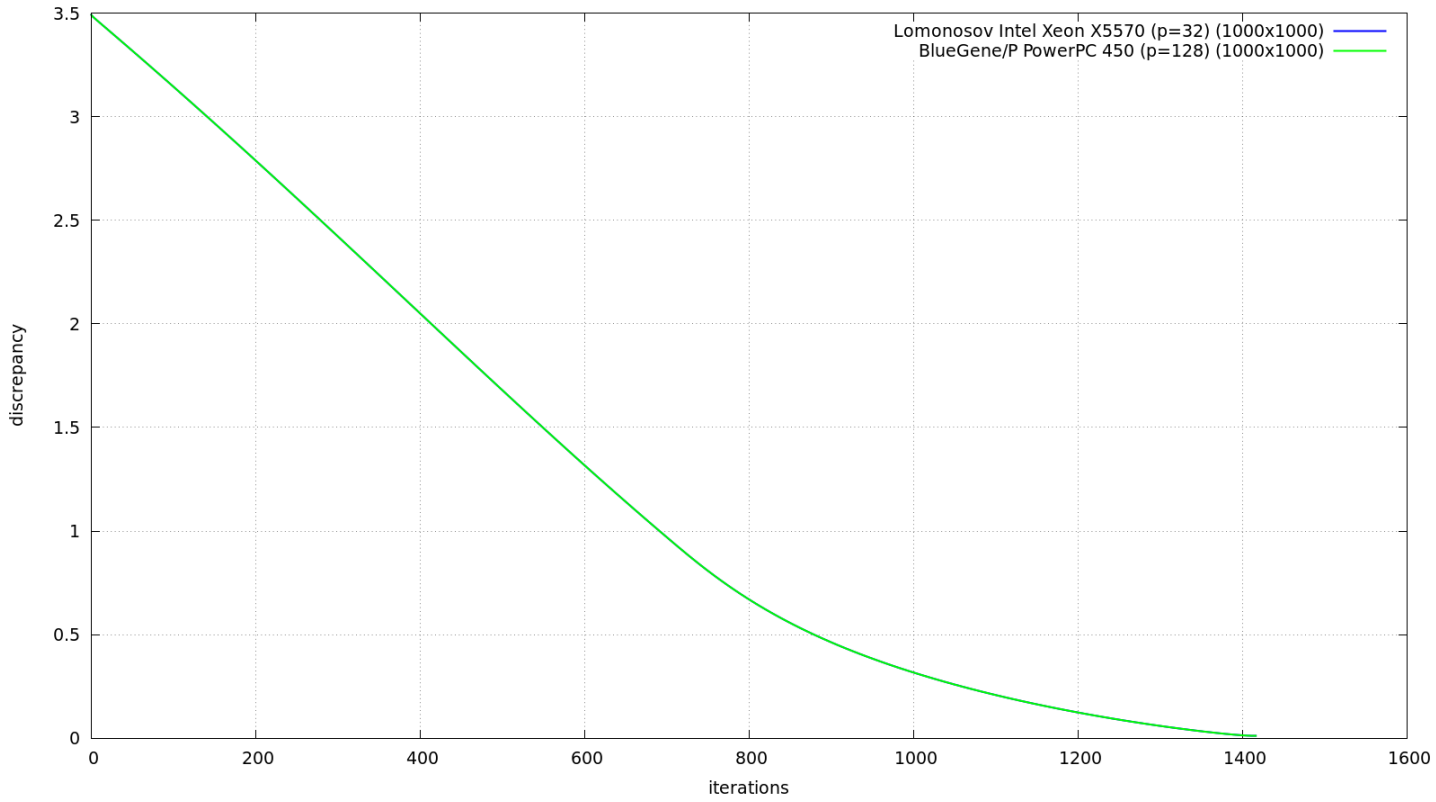

| Конфигурация | Макс. значение отн. погрешности | Мин. значение отн. погрешности |
|---|---|---|
| Lomonosov, p=32, n=1000 | 0.0783527310899 | 0.0 |
| Lomonosov (cuda), p=32, n=1000 | 0.0783527298442 | 2.02627890593e-11 |
| BlueGene/P, p=128, n=1000 | 0.0783527310899 | 0.0 |

В данном случае для архитектуры nvidia и её точности справедливы аналогичные рассуждения, как и в параграфе оценки абсолютной погрешности.

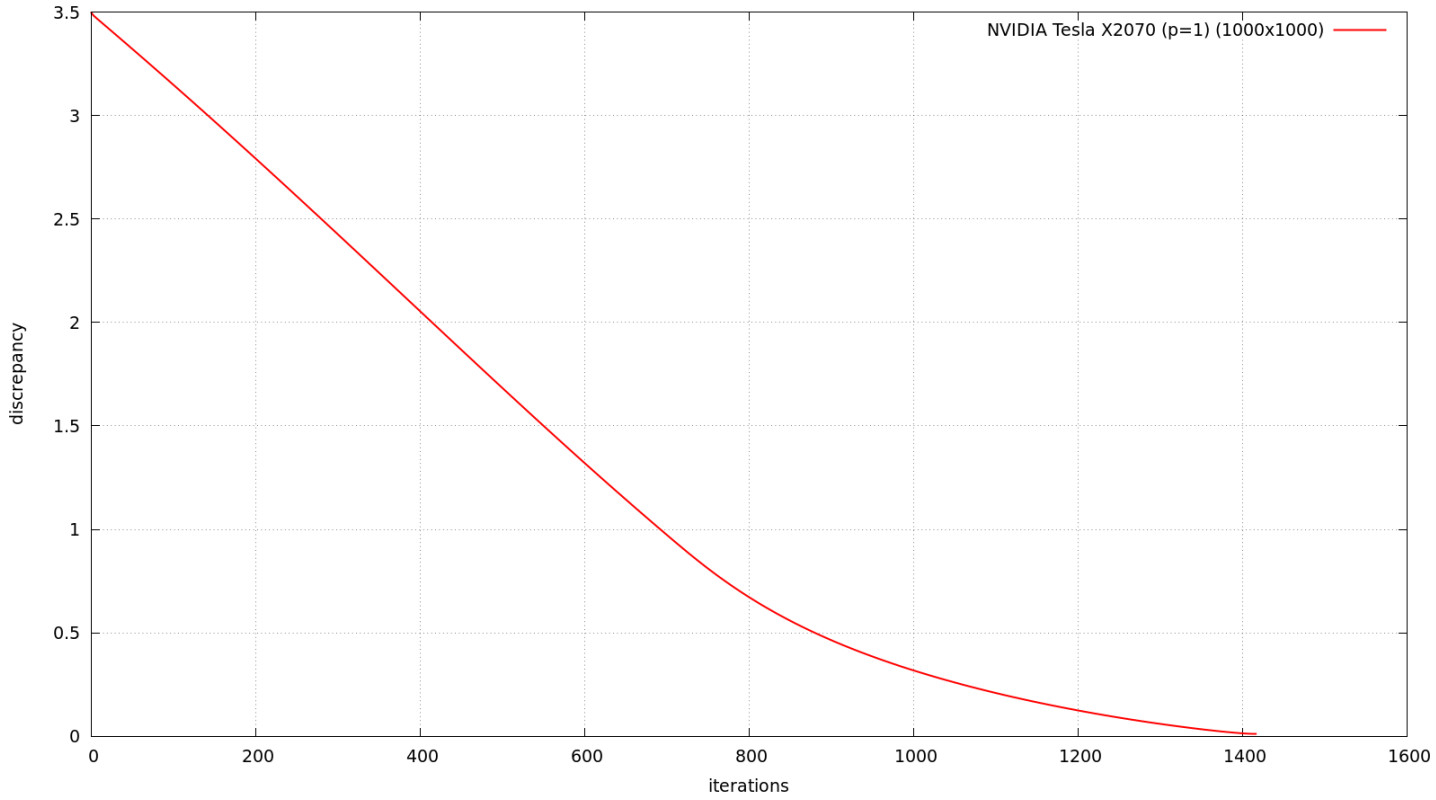
## 8.4.Графическое изображение скорости сходимости.

После каждой итерации выполнения алгоритма, рассчитывалась квадратичная норма разности приближённой функции и точного решения дифференциального уравнения.
В следствие чего были нарисованы графики сходимости при выполнении программы для разных конфигураций с использованием разных технологий.

Графический вид сходимости для конфигураций (BlueGene/P, p=128, n=1000) и (Lomonosov, p=32, n=1000) – совпадает:



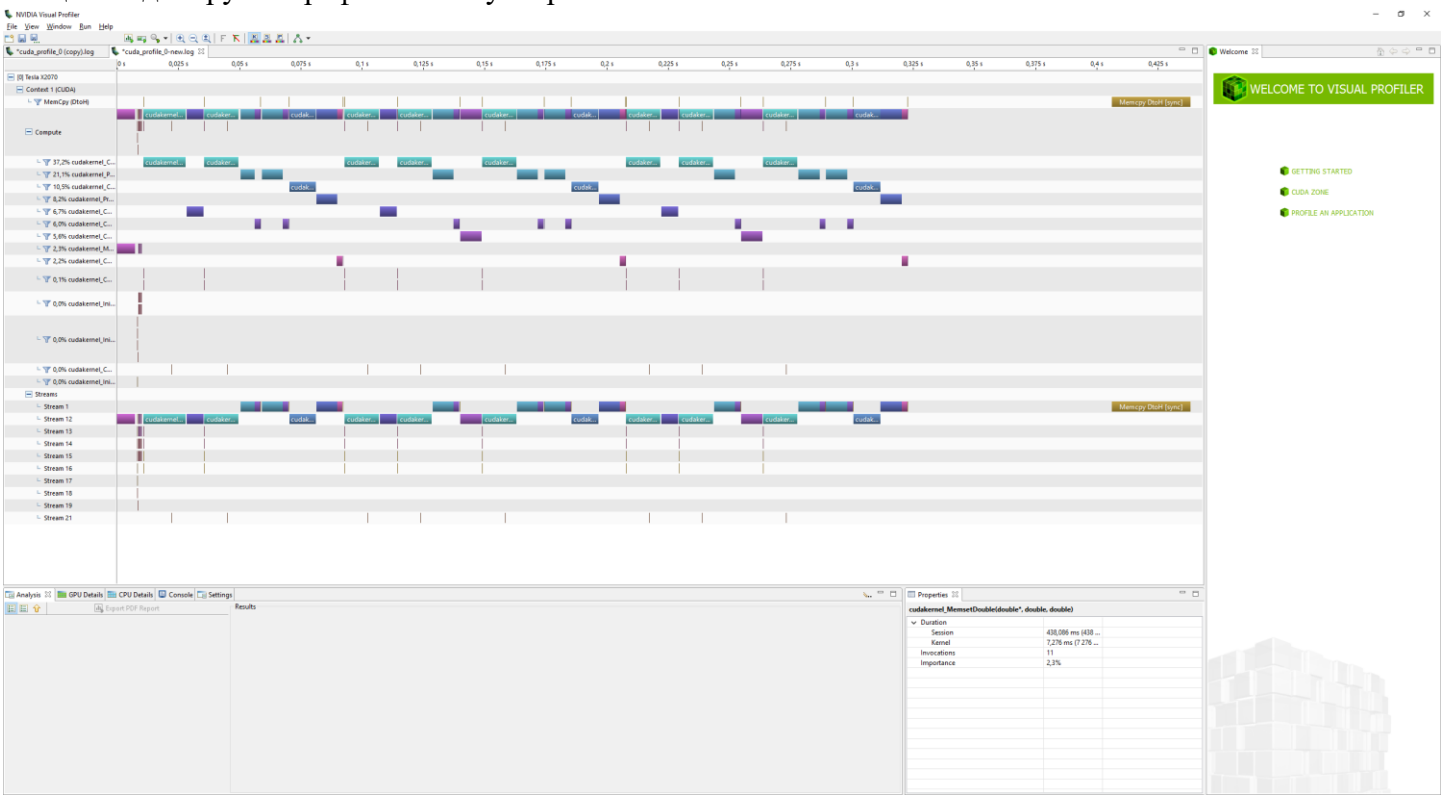

Графическое изображение сходимости для конфигурации, соответствующее вычислениям на графической карте (Lomonosov (cuda), p=1, n=1000), также ничем не отличается:



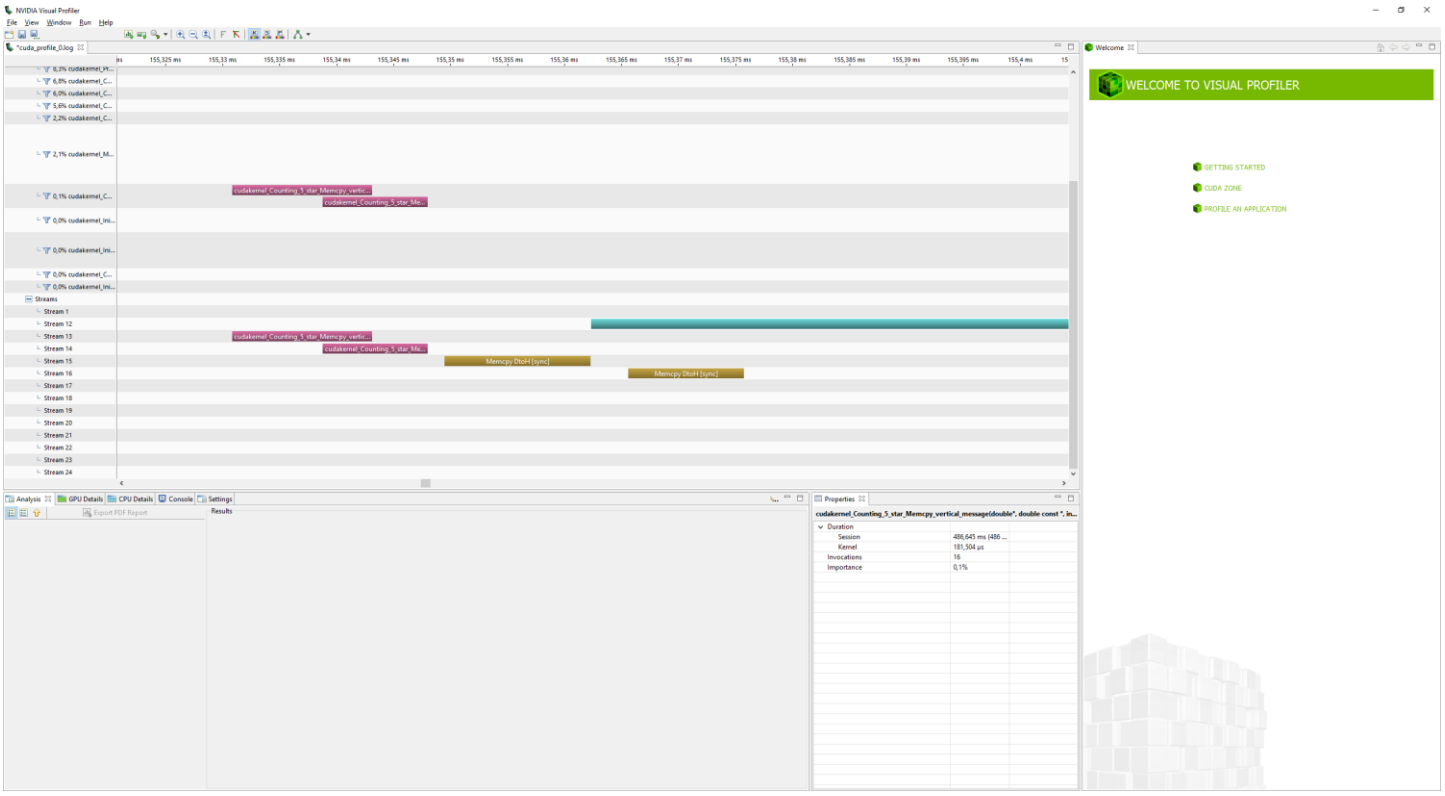
# 9. Профилирование и анализ работы с графическим ускорителем

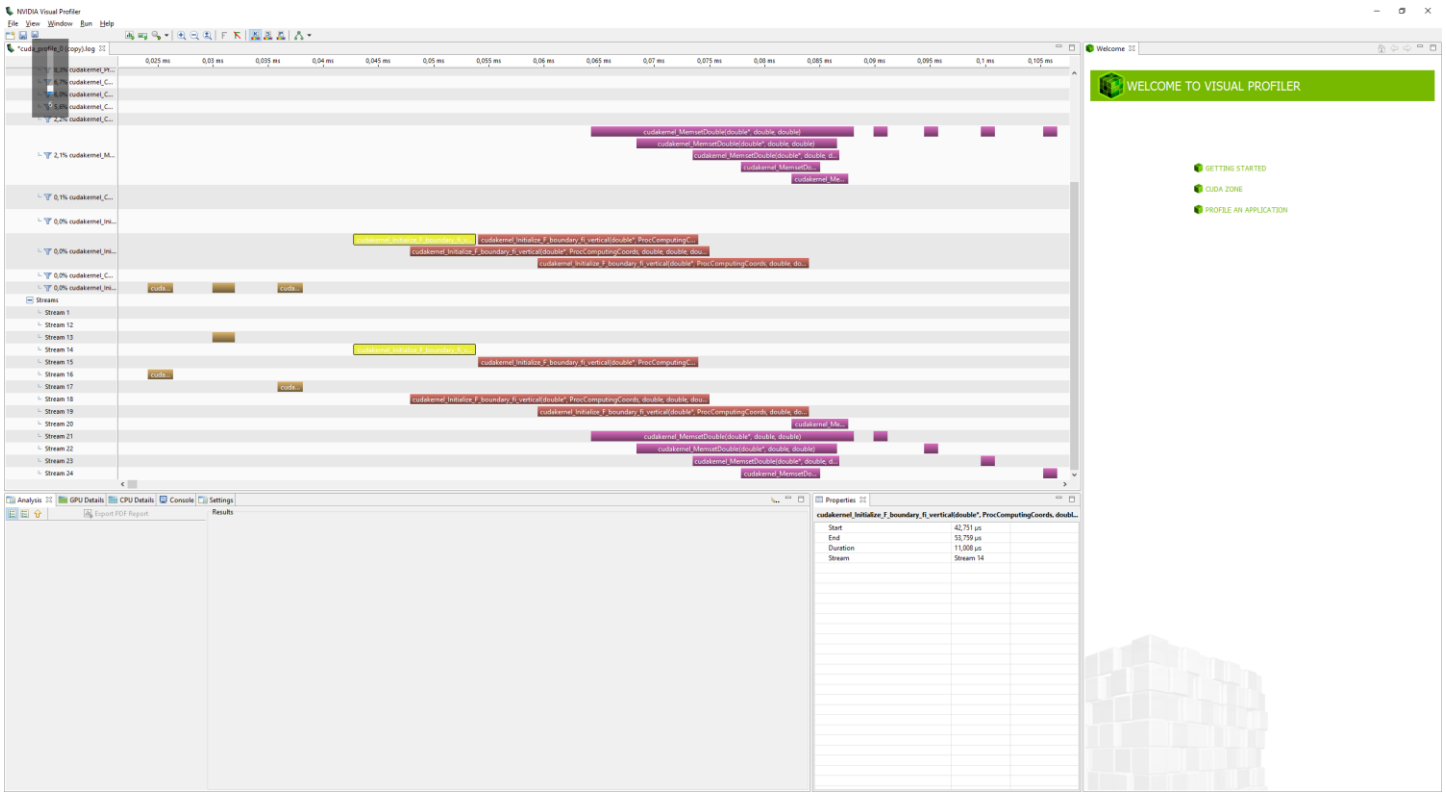
Профилирование проводилось для разбиения сетки 5000x5000, одной gpu карты NVIDIA Tesla X2070 на Lomonosov и первых 3-х итераций алгоритма.

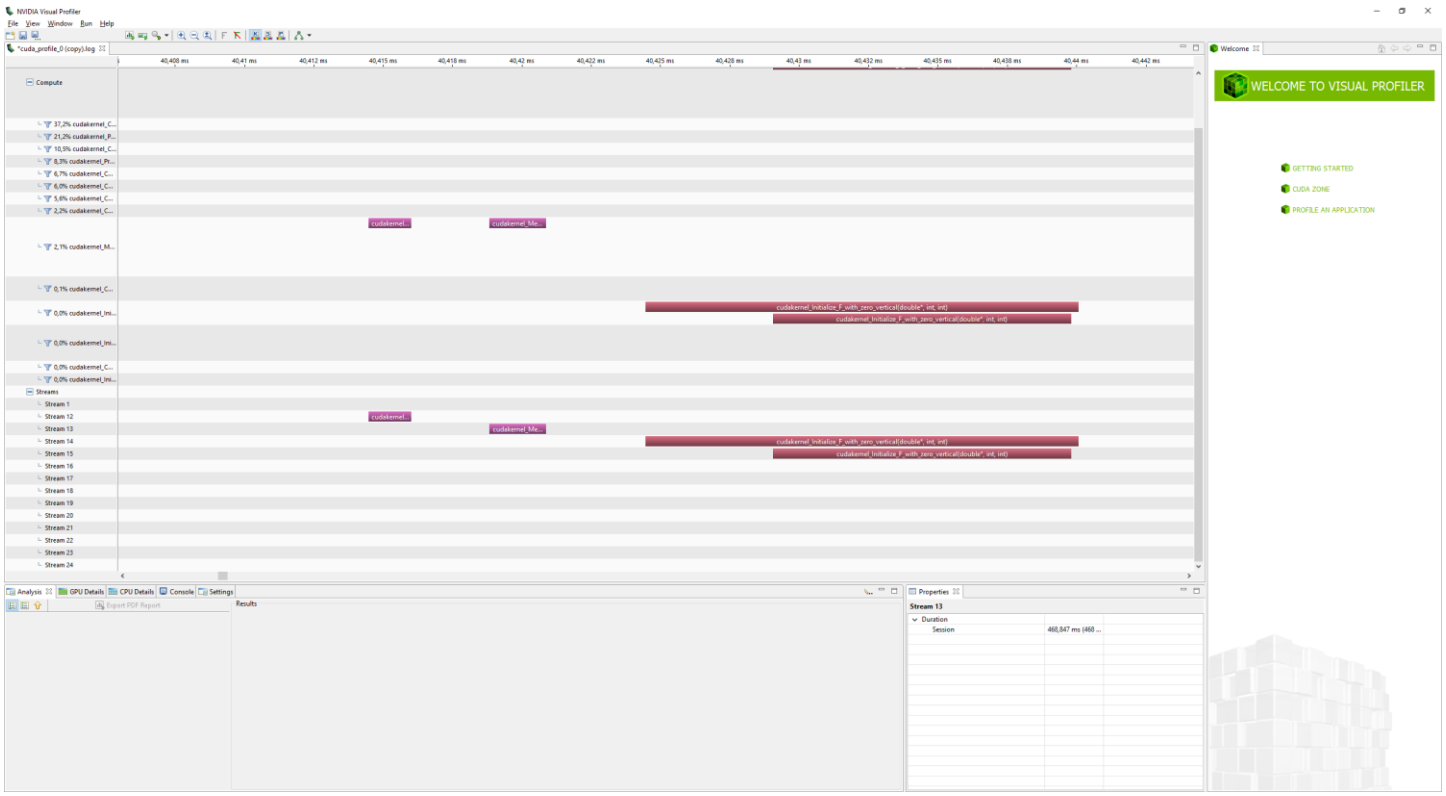
Общий вид загрузки графического ускорителя:




Следующие 3 изображения иллюстрируют асинхронную работу графического ускорителя (как параллельное копирование данных, так и выполнение ядер):






По результатам профилирования можно отметить, что все задачи разделяются на 2 категории:

- очень большие (рассчёты, которые ведутся на всей внутренней области сетки)
- очень маленькие (рассчёты, которые ведутся для краёв разбиения (краёв сетки))

Между большими операциями во многих местах есть задержка в выполнении примерно до 0,5 миллисекунд, - это связано с тем, что между различными этапами расчётов, у меня производится cudaAllStreamsSynchronize, чтобы дождаться выполнения данного этапа, и уже дальше начать подготовку к следующему этапу (например, рассчёт размера грида), и уже последующий запуск дальнейших вычислений на графическом ускорителе.

Однако внутри одного этапа, не зависящие друг от друга задачи успешно вычисляются в асинхронном режиме (как это проиллюстрировано на картинках выше).

При вычислении скалярного произведения (stream1), ядра выполняются без описанной выше задержки (потому что там отсутствует работа cpu между вызовами ядер, просто подряд запускаются gpu ядра).

Работа с памятью почти полностью отсутствует, так как почти все вычисления в моей программе производятся на gpu и данные туда/оттуда почти не пересылаются.

В stream21 можно видеть выполнение очень короткой задачи counting_5star_nxm_corners. Она хорошо показывает, что после старта задачи в stream12 counting_5star_insides, cpu продолжает свою работу (там в частности происходит небольшая работа с mpi (небольшая, т.к. профилирование происходило для 1 процесса, и поэтому там холостые waitall, а также несколько проверок, которые *не* пошлют данные соседним процессам (так как их нету))), и уже после того, как cpu отработает часть своих задач, оно запустит на выполнение короткую задачу, которая выполнится параллельно основной долгой задаче.
Как случилось, что задача смогла вклиниться в параллельное выполнение с другой задачей, прямо в середину этой другой задачи: это связано с тем, что задача очень мала и умещается буквально в несколько варпов, в то время как большая задача из-за своих размеров не пользует абсолютно все варпы (это связано с распределением между мультипроцессорами, которое было описано в параграфе с особенностями использования технологий (подпараграф про cuda) (т.к. по сути, можно попытаться перетащить несколько задач с загруженных нитей на пустые варпы (на дополнительный блок), но мы не сможем разгрузить все нити ровно на 1 задачу, чтобы они одновременно закончили своё выполнение на один шаг раньше, а разгружать не все нити - не имеет смысла, т.к. ядро будет ждать выполнения всех нитей))

В завершении вычислений присутствует одна долгая операция копирования памяти с устройства на хост - это для дальнейшего вывода вычисленной функции в файл (в начале также есть часть вычислений отвечающая за инициализацию) - это не повторяющиеся многократно операции и по времени выполнения сравнимы с одной итерацией алгоритма, так что их скорость выполнения - не очень важна.