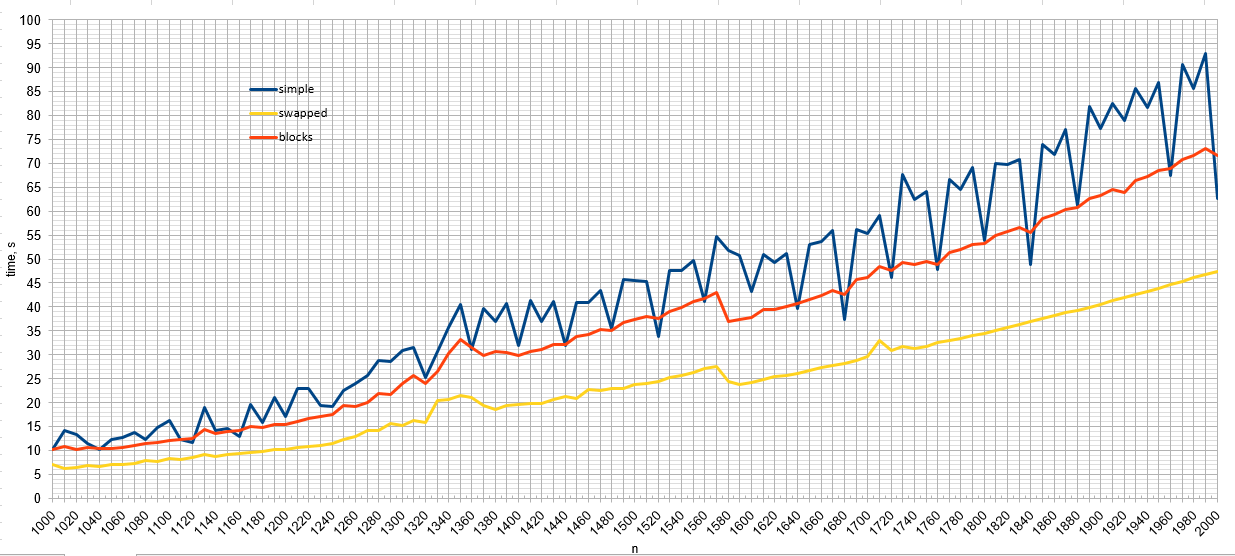
**Лабораторная C0902**

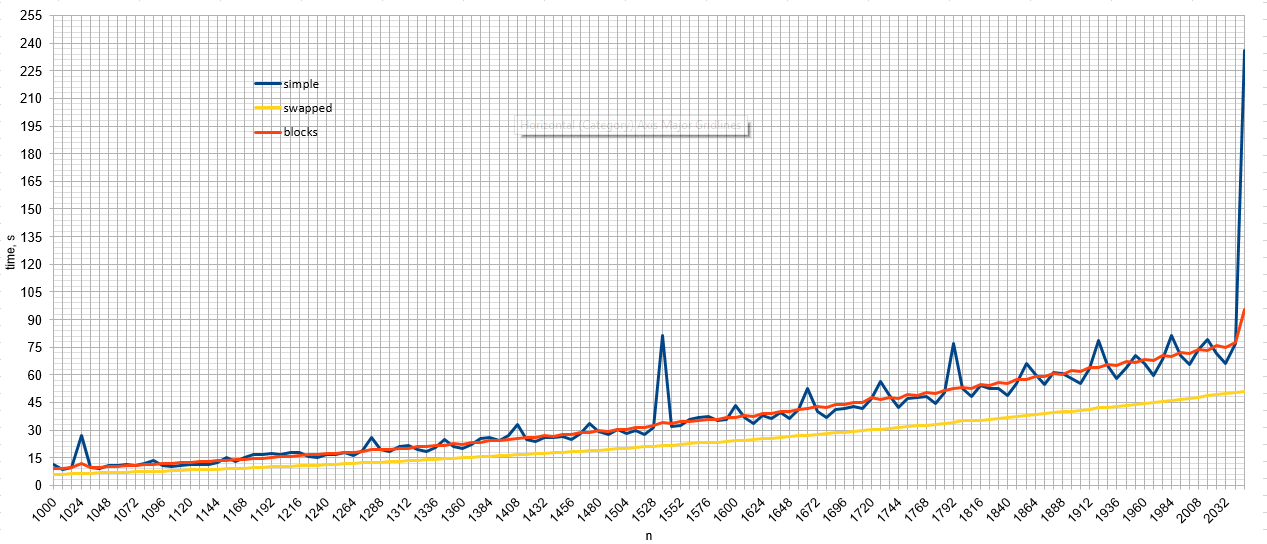
Прудников Евгений

Elements type: **float**;Step: **10**



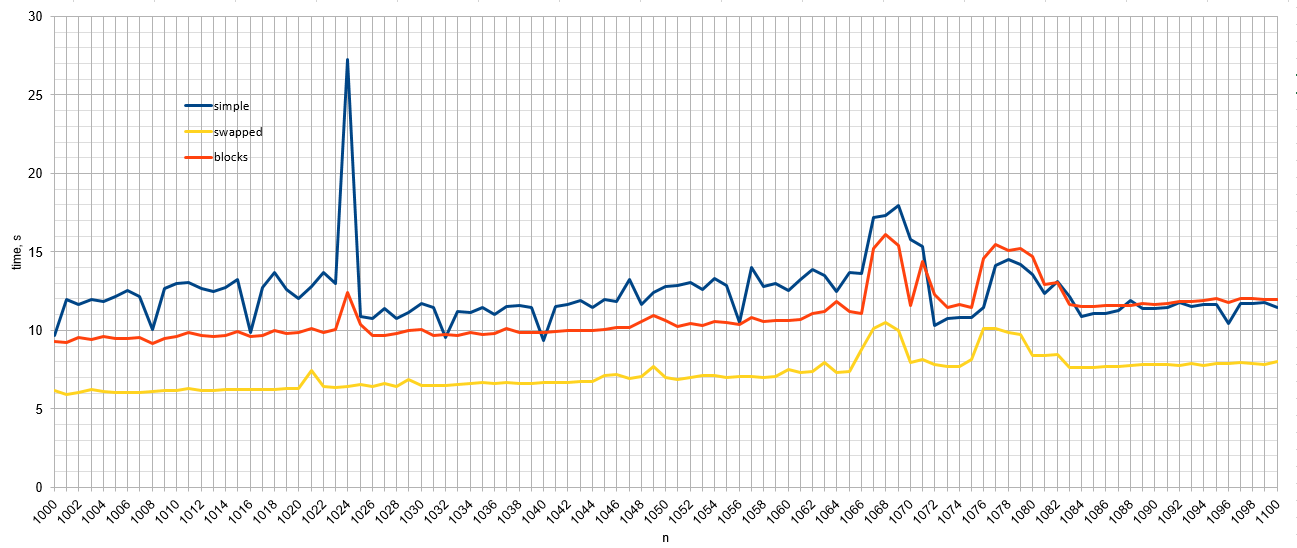
В районе 1260 – 1380 возможно что-то дополнительно загрузило процессор (возможно какой-то загрузчик обновлений запустился) и в районе 1560 – 1600 видно, что это что-то отстало и алгоритмы пошли быстрее

Elements type: **float**;Step: **8**



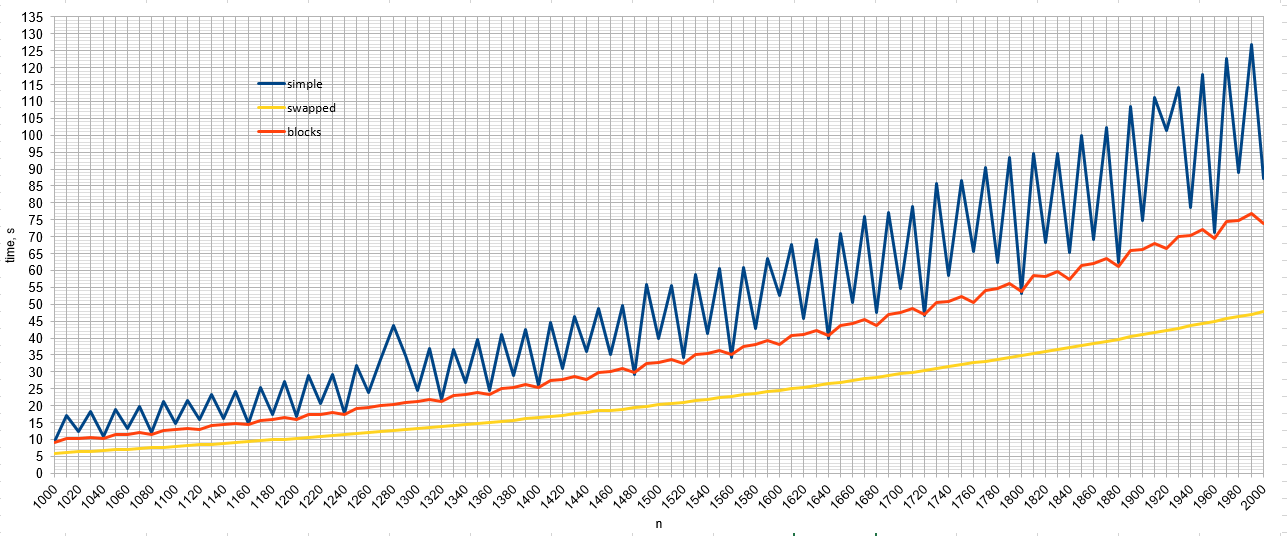
Для 1024 и 2048 выбросов объяснение ниже, почему для 1536 достаточно большой выброс неизвестно при типе float (для double на этом значении тоже есть выброс, но там есть кратность с critical stride) (1536 \* 4 mod 4096 = 2048)

Elements type: **float**; Step: **1**



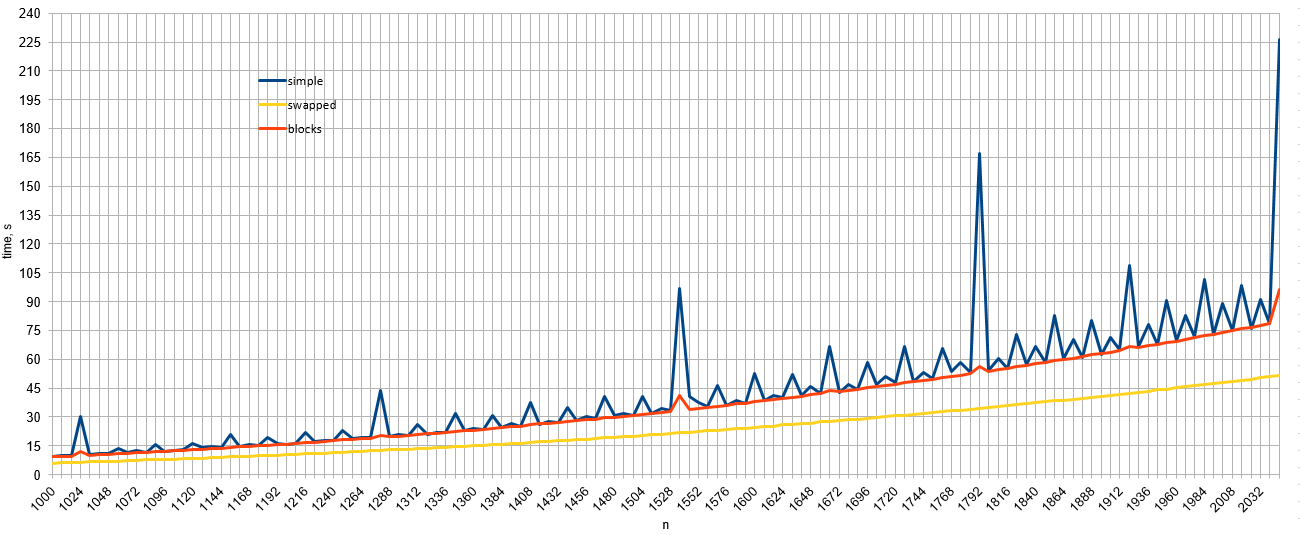
Для 1024 объяснение ниже, 1066 – 1086 похоже опять же на какие-то дополнительные нагрузки процессора (что-то запустилось), остальные неровности – шумы или другие процессы

Elements type: **double**; Step: **10**



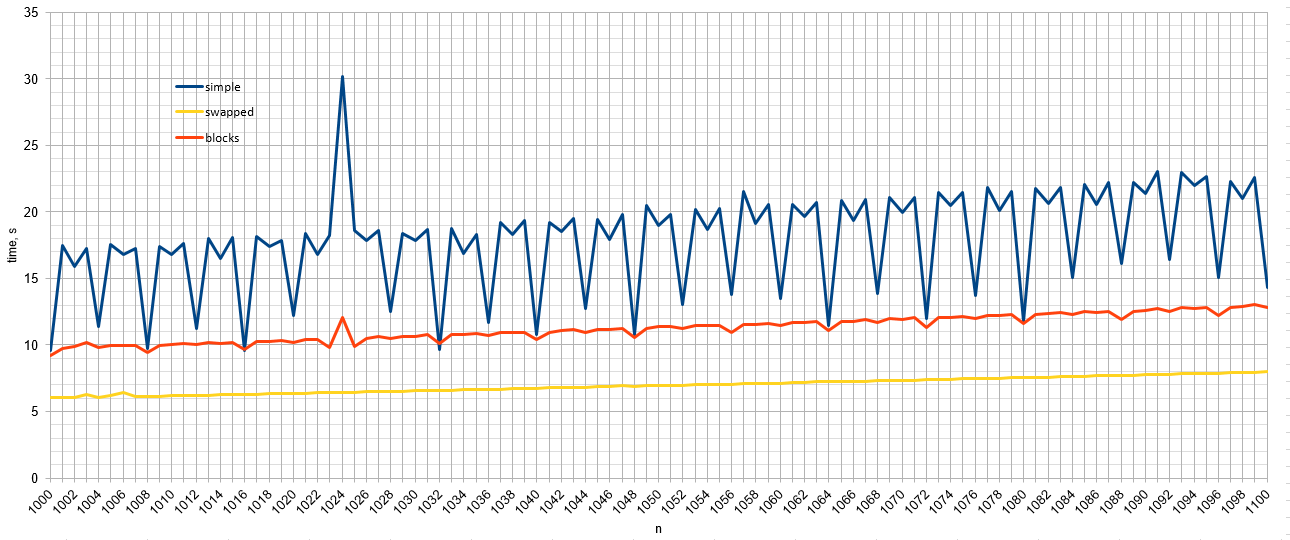
Похоже, что природа выброса на 1280 для double такая же, как и на 1536 для float (1280\*8 mod 4096 = 2048)

Elements type: **double**; Step: **8**



Выброс 1792- та же причина что и для float 1536 и double 1280 (1792\*8 mod 4096 = 2048)

Elements type: **double**; Step: **1**



Использованный скрипт: **main.cpp** (без изменений)

Процессор**: Intel Core i3 3120M**

Кэш:

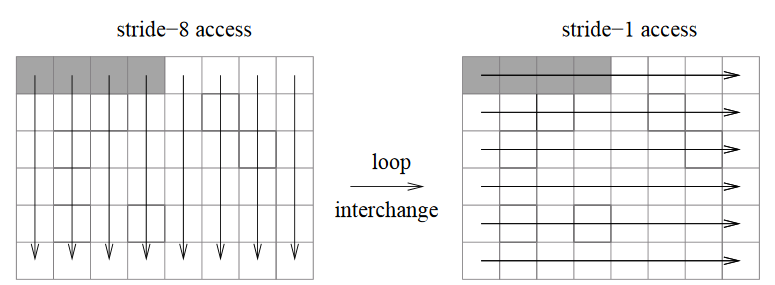
* L1 Data: **2 x 32Kb 8-way**
* L1 Instr.: **2 x 32Kb 8-way**
* L2: **2 x 256 Kb 8-way**
* L3: **3Mb 12-way**
* Cache-line: **64 bytes**

**Выводы и наблюдения:**

1) По графикам видно, что все алгоритмы имеют различную скорость работы (blocked быстрее чем simple и swapped быстрее их обоих).

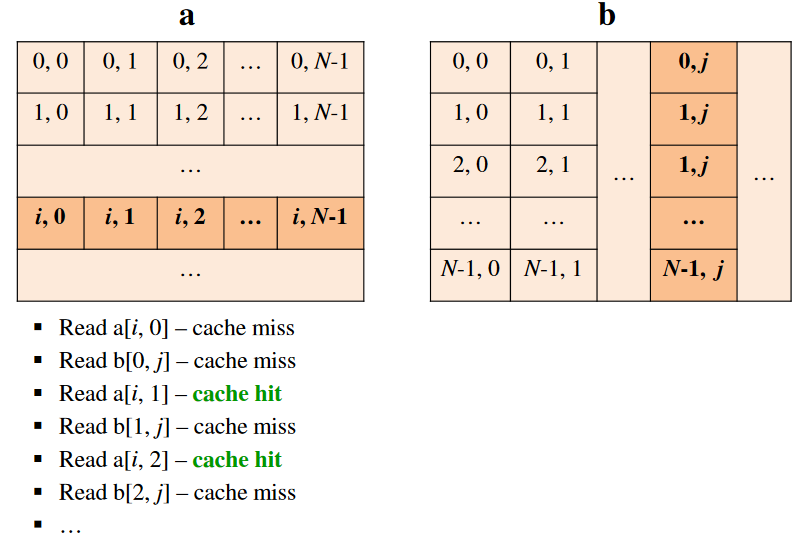
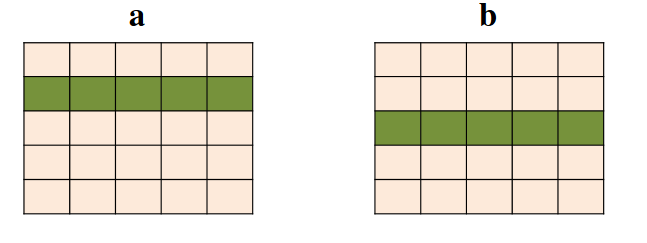
Рассмотрим сначала swapped и simple: алгоритмы очень похожи, за исключение нескольких строк, однако именно из-за них получается выигрыш swapped

Использованная в swapped трансформация (loop interchange) переворачивает порядок двух вложенных циклов что улучшает локальность данных за счет уменьшения шага (stride) доступа к данным (однако её можно применять только если порядок выполнения циклов не важен)(для матриц, записанных в одну строку это всё также справедливо):



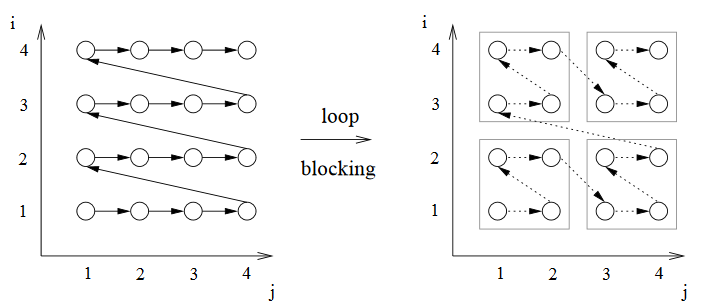
Мы предполагаем, что матрицы хранятся в памяти построчно т.е. 2 элемента хранятся в памяти рядом, если их вторые индексы последовательные номера (i = 1, j = 1,2,3 …). Алгоритм simple обращается к элементам матрицы по колонкам и, следовательно, загруженные в кэш данные (серые на рисунке) не пере используются если матрицы слишком большие чтобы полностью поместиться в кэш. При использовании loop interchange (в swapped алгоритме) процессор обращается к элементам матрицы расположенным в памяти последовательно, что повышает локальность данных (теперь они все вместе и идут последовательно) и уменьшает количество кэш промахов, следовательно, повышается скорость работы.

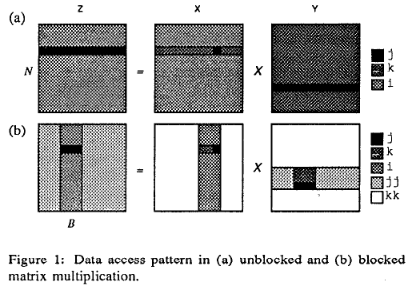
Ещё одна иллюстрация:



Blocking алгоритм также повышает локальность данных и увеличивает пере использование данных в кэше за счет загрузки в кэш блоков и последующего их пере использования.

Примерная иллюстрация работы:





Однако на моем процессоре у него получилось не самое лучшее время

В алгоритме Simple на некоторых графиках видна «елочка», возможно из-за чуть более удачного расположения данных в кэше при определенных размерах матриц получается быстрее, и наоборот, данные попадают не туда – медленнее

У алгоритмов blocking и swapped нету таких проблем (я думаю выбросы — это просто погрешности) из-за лучшей локализации данных в кэше

2) Также на графиках видны достаточно высокие выбросы на float\double с шагом 1,8. Объясняется это структурой кэша:

Так как кэш в моем процессоре 8-канальный множественно- ассоциативный (8-way Set Associative cache) адреса оперативной памяти могут попадать только в определенный set (узнать в какой set попадет адрес можно по формуле set = (address // cache\_line\_size) mod number\_of\_sets, где // - целочисленное деление)

Для моего процессора:

* Number\_of\_cache\_lines = cache\_size / cache\_line\_size = 32kb/64b = 215/26 = 512 cache\_lines
* Number\_of\_sets = number\_of\_cache\_lines / number\_of\_ways = 512 / 8 = 64 sets

Каждый сет содержит 8 кэш линий

* Critical Stride = cache\_size/ number\_of\_ways = 32kb / 8 = 215/23 = 4kb

Critical Stride – это критический шаг, после которого строчки из оперативной памяти будут попадать в тот же set (если мы возьмем адрес 0x3E8 который попадает в сет 15 и прибавим к нему 0x1000 (4kb) то новый адрес 0x13E8 также будет попадать в сет 15)

Переменные расстояние в памяти, между которыми кратно критическому шагу будут постоянно бороться за один и тот же set (одни и те же кэш линии). Tо есть при определенных размерах матриц, кратных критическому шагу, будет огромное количество кэш промахов, так как данные будут попадать в один и тот же сет и постоянно перезаписываться (каждые 8 итераций)

Если посчитать:

Critical\_Stride = 4kb

Type: float (4b)

N = 1000

Row\_size / critical\_stride = 1000\*4 mod 4096 = 4000 ( != 0)

N = 1024

Row\_size / critical\_stride = 1024\*4 mod 4096 = 0 - много cache miss

N = 1034

Row\_size / critical\_stride = 1034\*4 mod 4096 = 40 ( != 0)

N = 2047

Row\_size / critical\_stride = 2047\*4 mod 4096 = 4092 ( != 0)

N = 2048

Row\_size / critical\_stride = 2048\*4 mod 4096 =0 - много cache miss

…

Литература:

1. **Optimizing software in C++. An optimization guide for Windows, Linux and Mac platforms.** Agner Fog. Technical University of Denmark.
2. **The Cache Performance and Optimization of Blocked Algorithms.** Monica S. Lam, Edward E. Rothberg and Michael E.Wolf Computer Systems Laboratory Stanford University
3. **An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms.** Markus Kowarschik and Christian Weib