Sudoku 4x3 GPU Exact Enumeration Research Log

Verification

Verification of 2006 Pettersen/Silver result used 1035 CPU-hours over a twelve-day period between 8 June 2022 and 20 June 2022.

Note on 3x4 vs 4x3

I have been using 3x4 and 4x3 somewhat interchangeably. I starting using just 3x4 because that's how the Wikipedia table of results had it. After reconnecting with Kjell after all these years, I saw that he used 3x4 and 4x3 to refer to two distinct methods for the exact enumeration.

- In a 4x3 count, the 144,578 gangsters (equivalence class representatives) of a four-box, 12x4 band are determined and used with the other two bands.
- In a 3x4 count, the 2230 gangsters of a three-box, 12x3 stack are determined and used with the other three stacks.

Both the original 2006 enumeration, and this 2022 verification, are 4x3. Both methods must give the same result, but both Kjell and I believe that 4x3 is more efficient. Kjell has recently been thinking about the 3x4 version, and he may yet develop an efficient way to do it using stack pairs.

I think Kjell is right that I should be naming all this work 4x3, not 3x4, but there is a lot of 3x4 in filenames and the like because that's where I started. If you see 3x4, keep in mind that the method under investigation is 4x3.

Workload

CPU-hours is easy to tally but not a great way to measure the workload. It's very dependent on the machines I happened to have available, and tends to be biased by the slowest ones. It is a measure of serial hours, i.e. the time that would have been needed if the machines were not run in parallel. It also was affected by some difficulty I had in getting Windows and Ubuntu to run the cores at max speed, instead of trying to conserve power.

A parallel measure of CPU-hours would be easy to tally if all machines were run for the entire time—it would be essentially the same as the formula for parallel resistance. But that was not the case in practice.

Parallel execution on multiple cores of a single CPU is not the same as parallel execution on separate machines, because the parallel threads compete with each other for shared resources. Each of the 144,578 gangsters is enumerated independently, and the time for each one is defined and recorded as the elapsed time divided by the number of parallel threads.

Computers

In the following table, the first six computers were used in the verification run. The speed is the average time to enumerate one gangster with a 32-gangster benchmark run of 865 – 896 (the first 32 in group 1). EPT2022 is the Nvidia Jetson AGX Xavier.

Name	GHz	Cores	Threads	CPU	OS	Compiler	Speed
PT2017	3.10	4	8	x64 Xeon E3-1535M v6	Windows	MS VC++	16.5
PT2019	2.11	4	8	x64 i7-8650U	Windows	MS VC++	24
Judy7		4	8	x64	Windows	MS VC++	
PT2015		2	4	x64	Ubuntu	GCC	
Judy6		2	4	x64	Windows	MS VC++	
CGNX	2.90	2	4	x64 i7-7600U	Windows	MS VC++	
EPT2022	2.26	8	8	ARM-64 v8.2	Ubuntu	GCC	10

The speed (actually seconds/gangster) numbers are somewhat variable, run to run. For example, most of the PT2019 runs are in the 23.8 – 24.5 range, but a small number came in at around 22.1. I don't understand this 2 seconds/gangster bimodal variation on this particular machine. My current speculation is that the variation in parallel thread order may interact with the data caches and hyperthreading and occasionally produce this weird bimodal behavior.

More detailed timing follows. This version is slightly different than the baseline verification run. The order of the DoubleBoxCount outer loop was modified to try to achieve slightly better data cache performance. It may have made a very small improvement in speed, barely measurable. See discussion in *Radical Improvement in Data Cache Hit Rate* below.

PT2017

```
Using 8 threads
1,180,382 cache misses 4,087,416 code calls
Read count file gridCount_1-.txt, total time so far 0.46 hours

Profile tree:
Sudoku3x4                                                   535.947
  RowCode Init                  15400 *         0.0661 ->     0.001
  ColCode Init                   5775 *         0.0579 ->     0.000
  Row Tables                   369600 *         0.1185 ->     0.044
  Column Tables                138600 *         0.1487 ->     0.021
  BoxCompatible Init              715 *         1.6134 ->     0.001
  Column Nodes                  31104 *        14.3970 ->     0.448
  BandGang Construct                                          0.247
  Verify BandGang Tables                                      0.019
  Band Gangsters                                              3.790
  Fix gang cache            300155625 *         0.0022 ->     0.672
  Read/verify gangsters        144578 *         1.2478 ->     0.180
  Replace cache codes       300155625 *         0.0021 ->     0.645
  Construct GridCounter       1998150 *         0.2037 ->     0.407
  Grid counter                                              528.615
    GridCounter Setup                                         0.906
      Big tables               119716 *         7.3130 ->     0.875
      Sort                                                    0.026
      Overhead                                                0.005
    Main count loop                32 * 16490904.0063 ->  527.709
    Overhead                                                  0.000
  Overhead                                                    0.856
```

PT2019

```
Using 8 threads
1,175,430 cache misses 4,070,668 code calls
Read count file test1_1-.txt, total time so far 0.00 hours

Profile tree:
Sudoku3x4                                                   772.867
  RowCode Init                  15400 *         0.0625 ->     0.001
  ColCode Init                   5775 *         0.0472 ->     0.000
  Row Tables                   369600 *         0.1273 ->     0.047
  Column Tables                138600 *         0.1496 ->     0.021
  BoxCompatible Init              715 *         1.6615 ->     0.001
  Column Nodes                  31104 *        15.4951 ->     0.482
  BandGang Construct                                          0.256
  Verify BandGang Tables                                      0.020
  Band Gangsters                                              5.599
  Fix gang cache            300155625 *         0.0023 ->     0.697
  Read/verify gangsters        144578 *         1.2698 ->     0.184
  Replace cache codes       300155625 *         0.0023 ->     0.676
  Construct GridCounter       1998150 *         0.2092 ->     0.418
  Grid counter                                              763.586
    GridCounter Setup                                         0.951
      Big tables               119716 *         7.6898 ->     0.921
      Sort                                                    0.030
      Overhead                                                0.000
    Main count loop                32 * 23832356.1656 ->  762.635
    Overhead                                                  0.000
  Overhead                                                    0.879
```

```
Using 8 threads
8.5697 [144578]; 1,176,526 cache misses 4,075,140 code calls
1,176,526 cache misses 4,075,140 code calls

Profile tree:
Sudoku3x4                                                   719.754
  RowCode Init                    15400 *         0.0735 ->   0.001
  ColCode Init                     5775 *         0.0550 ->   0.000
  Row Tables                     369600 *         0.1282 ->   0.047
  Column Tables                  138600 *         0.1622 ->   0.022
  BoxCompatible Init                715 *         1.8829 ->   0.001
  Column Nodes                    31104 *        14.8154 ->   0.461
  BandGang Construct                                          0.259
  Verify BandGang Tables                                      0.029
  Band Gangsters                                              5.337
  Fix gang cache              300155625 *         0.0023 ->   0.702
  Read/verify gangsters          144578 *         1.2762 ->   0.185
  Replace cache codes         300155625 *         0.0023 ->   0.676
  Construct GridCounter         1998150 *         0.2091 ->   0.418
  Grid counter                                              710.708
    GridCounter Setup                                         0.963
      Big tables                 119716 *         7.8065 ->   0.935
      Sort                                                    0.028
      Overhead                                                0.001
    Main count loop                  32 * 22179517.5406 -> 709.745
    Overhead                                                  0.000
  Overhead                                                    0.909
```

EPT2022

```
Using 8 threads
1,158,310 cache misses 4,009,004 code calls
Read count file gridCount_1-.txt, total time so far 0.46 hours

Profile tree:
Sudoku3x4                                                   326.179
  RowCode Init                    15400 *         0.1489 ->   0.002
  ColCode Init                     5775 *         0.0968 ->   0.001
  Row Tables                     369600 *         0.2255 ->   0.083
  Column Tables                  138600 *         0.3447 ->   0.048
  BoxCompatible Init                715 *         4.3080 ->   0.003
  Column Nodes                    31104 *        23.6487 ->   0.736
  BandGang Construct                                          0.250
  Verify BandGang Tables                                      0.022
  Band Gangsters                                              4.772
  Fix gang cache              300155625 *         0.0019 ->   0.581
  Read/verify gangsters          144578 *         0.5562 ->   0.080
  Replace cache codes         300155625 *         0.0022 ->   0.649
  Construct GridCounter         1998150 *         0.2685 ->   0.536
  Grid counter                                              317.587
    GridCounter Setup                                         2.208
      Big tables                 119716 *         6.0924 ->   0.729
      Sort                                                    1.478
      Overhead                                                0.001
    Main count loop                  32 *  9855592.8388 -> 315.379
    Overhead                                                  0.000
  Overhead                                                    0.829
```

Jetson AGX Xavier GPU Basics

```
Clock rate 1377000 kHz
L2 cache size 524288
Max blocks per multiprocessor 32
Max grid size 2147483647.65535.65535
Max block dimension 1024.1024.64
Max threads per block 1024
Max threads per multiprocessor 2048
Multiprocessor count 8
Reserved shared memory per block 0 bytes
Shared memory per block 49152 bytes
Shared memory per multiprocessor 98304 bytes
Total global memory on device 32517738496 bytes
Warp size in threads 32
```

Cuda First Cut

First cut at a Cuda GPU program for executing the main grid counting loop for the Sudoku 4x3 exact count. It makes very poor use of GPU resources, and is actually slower than running the CPU code on the Jetson's 8-core ARM v8.2 processors. The purpose of this first cut is to confirm that I understand the Nvidia tool chain and the most basic operations of Cuda. The code runs and gets the correct results.

The first step in using the GPU properly will be to deal with the poor memory access pattern, which radically degrades the GPU's memory bandwidth and stalls the compute elements.

Here is a pure GPU run—one thread runs all the setup, and calls Cuda code to run 16 blocks of 32 threads to do the main counting loops.

```
Using 1 thread
1,052,036 cache misses 3,653,317 code calls
Read count file gridCount_1-.txt, total time so far 0.46 hours

Profile tree:
Sudoku3x4                                                     389.234
  RowCode Init                    15400 *        0.1559 ->    0.002
  ColCode Init                     5775 *        0.1008 ->    0.001
  Row Tables                     369600 *        0.2489 ->    0.092
  Column Tables                  138600 *        0.3945 ->    0.055
  BoxCompatible Init                715 *        5.3005 ->    0.004
  Column Nodes                    31104 *       23.1097 ->    0.719
  BandGang Construct                                          0.252
  Verify BandGang Tables                                      0.028
  Band Gangsters                                             10.687
  Fix gang cache              300155625 *        0.0019 ->    0.580
  Read/verify gangsters          144578 *        0.5620 ->    0.081
  Replace cache codes         300155625 *        0.0019 ->    0.562
  Construct GridCounter         1998150 *        0.2820 ->    0.564
  Give band counts to GPU     300155625 *        0.0026 ->    0.795
  Grid counter                                              373.769
    GridCounter Setup                                         2.582
      Big tables                 119716 *        6.3462 ->    0.760
      Sort                                                    1.616
      Tables -> GPU                                           0.205
      Overhead                                                0.001
    Main count loop                  32 * 11599569.5722 ->  371.186
    Overhead                                                  0.000
  Overhead                                                    1.045
```

Here is a heterogeneous run—seven threads run on the ARM cores and one thread calls the Cuda/GPU code:

```
Using 8 threads
1,158,288 cache misses 4,009,055 code calls
Read count file gridCount_1-.txt, total time so far 0.46 hours

Profile tree:
Sudoku3x4                                                      336.808
  RowCode Init                  15400 *         0.1379 ->   0.002
  ColCode Init                   5775 *         0.0980 ->   0.001
  Row Tables                   369600 *         0.1976 ->   0.073
  Column Tables                138600 *         0.2959 ->   0.041
  BoxCompatible Init              715 *         3.6988 ->   0.003
  Column Nodes                  31104 *        24.4196 ->   0.760
  BandGang Construct                                          0.253
  Verify BandGang Tables                                      0.022
  Band Gangsters                                              4.569
  Fix gang cache            300155625 *         0.0020 ->   0.589
  Read/verify gangsters        144578 *         0.5550 ->   0.080
  Replace cache codes       300155625 *         0.0021 ->   0.629
  Construct GridCounter       1998150 *         0.2684 ->   0.536
  Give band counts to GPU   300155625 *         0.0028 ->   0.854
  Grid counter                                              327.460
    GridCounter Setup                                        36.896
      Big tables               119716 *         6.1662 ->   0.738
      Sort                                                   35.850
      Tables -> GPU                                           0.306
      Overhead                                                0.001
    Main count loop                32 *   9080136.8877 -> 290.564
    Overhead                                                  0.000
  Overhead                                                    0.935
```

Compilation Command Line

```
nvcc --m64 --std c++17 --gpu-architecture=sm_72 --compiler-options -
std=c++17,-march=armv8-a+simd,-Ofast,-Wno-format,-DJETSON --linker-options -
pthread --include-path . -o sudoku3x4 bignumMT.cpp profile.cpp general.cpp
timer.cpp Sudoku3x4.cpp gridCount.cu
```

GangSets

As further described in the source comments, the 144,578 gangsters fall into 9 sets, where every gangster in each set has the same box0 and box1 codes. box0 is always 0, and box1 is one of the 9 codes associated with what the source calls nodes. Here are the sets:

Set	Box1 Code	Gangsters	StartIndex	Unique Codes	GCD
0	0	865	0	602	64
1	1	11989	865	2393	16
2	9	10518	12854	2664	16
3	36	63042	23372	3427	8
4	39	10337	86414	2130	8
5	44	44982	96751	3945	8
6	324	1273	141733	753	8
7	325	1519	143006	769	8
8	2537	53	144525	60	8

A 1.4GB lookup table (`GridCounter::gcPackets_`) is created (by `GangCounter::setup_`) for each GangSet. It takes about a second to create, after which hours are spent enumerating the GangSet.

Pettersen's version also has 9 GangSets (math is math), but they are slightly different:

```
Set  Box1 Code  Gangsters
 0       0          865
 1       1        11989
 2       9        10518
 3      36        63042
 4      44        54060
 5      66         1259
 6     604         1273
 7     614         1519
 8     716           53
```

The box1 codes are different, and gangsters are distributed differently among sets 4 and 5. This is presumably due to different canonical forms and class representatives. The 4/5 differences are most interesting.

Radical Improvement in Data Cache Hit Rate

The current version did the complete 4x3 enumeration in **46.4** CPU/GPU-hours on my Jetson AGX Xavier (heterogeneous computation using 8 CPU cores and the GPU). This is entirely due to loop reorganization to achieve a much higher data cache hit rate. CPU speeds improve (empirically) by 10x on my 4-core x64 machines, and 5x on the Jetson's 8-core ARMs. GPU performance is about 3x better, still slower than the CPUs running 8 parallel threads, and much worse than I'm hoping for.

All of the gangsters in each of the 9 GangSets have the same box0 and box1. There are 346 * 346 = 119716 compatible box4-box5, box8-box9 combinations, of which only 60204 have to be run due to the box4-box8 symmetry. That's the outer loop for each gangster in the original enumeration method used for the verification run.

In the original verification run, each gangster was enumerated independently. Each of the parallel threads in the Rope took the next gangster in turn until all in the current GangSet were done. The loop structure for each gangster was

```
Do 60204 iterations
    Get [5775][5775] cache level for band1 and band2
    Do 346 iterations
        Get [5775] line for band1 and band2
        Do 346 iterations
            Fetch band counts from band1 and band2 lines
            multiply-accumulate
```

The innermost loop looks like this:

```
for (int b3 = 0; b3 < 346; ++b3)
  count += (uint64_t)band1CacheLine[box7[b3]] * band2CacheLine[box11[b3]];
```

box7 and box11 are 346-element arrays of uint16_t, accessed sequentially. These will fit in L1 cache and the sequential access is favorable, although the values are used only once and will compete for L1 with the band lines.

(I use *cache* for two distinct purposes. When referring to the source code, this is the `int32_t [9][5775][5775] BangGang::gangCache_` array (1.2 GB), whose first purpose is as an actual cache for finding gangsters. At the counting stage it holds band counts. I also use *cache* to refer to the CPU and GPU hardware data caches. When referring to the hardware I'll use *data cache*, or L1/L2/L3 cache.)

Each cache line in the inner loop is one 5775-element array. The inner loop touches 346 elements of the 5775-element lines, scattered at the offsets specified by box7 and box11.

Data cache hit rate was terrible because

- The two [5775][5775] cache levels change for each iteration of the outer loop.
- The two [5775] cache lines change for each iteration of the middle loop.

Thus accesses to the gangster cache jumps around in the 1.2GB array, and, more significantly, in the 133MB [5775][5775] gangster cache levels. These arrays are way too big for the L2 or L3 data caches on any contemporary processor.

The level change for outer loop iterations is reduced by sorting the huge table (1.4GB) that drives the 60204 iterations for a more favorable order. The sort groups indices with the same levels together, and with the groups in this order:

00 01 02 03 04 05 06 07 08
18 17 16 15 14 13 12 11
22 23 24 25 26 27 28
38 37 36 35 34 33
44 45 46 47 48
58 57 56 55
66 67 68
78 77
88

Note that not all of these groups may exist in a given GangSet. The speed improvement due to this sort is barely measurable because the real problem is in the middle loop, but the sort is easy to do and may give better results when combined with the middle loop improvements.

The key to the radical improvement is that each GangSet has long sequences where box2 is the same. Such sequences vary from a dozen or so gangsters to over a thousand, typically hundreds. The sequences are called box2Groups in the source.

The current version rearranges the loops like this:

```
Do 346 iterations
    Get [5775] cache line for band1 and band2
    Do box2GroupSize iterations
        Do 346 iterations
            Fetch band counts from band1 and band2 lines
            multiply-accumulate
```

Here the same 5775-element band cache lines are used for each of the gangsters in the box2Group, giving hundreds of times more references to those lines that the 346 of the original. The exact same number of iterations of the exact same inner loop are executed, but the speed is an order of magnitude faster due to the data cache hit rate.

In this version, each parallel thread takes one of the 60204 box4-box5, box8-box9 combinations.

Heterogeneous Computation

There are four places in the code where multiple parallel threads can be used—three associated with finding gangsters and their properties, and one for the main counting loops. The Rope class is a simple, uniform way to handle the threads, as described in source comments.

The number of threads to use is specified on the command line, and is usually the total number of available virtual processors (cores * hyperthreading). The GPU code is written to do the same thing one CPU thread would do, using many GPU blocks (88) and threads (32) internally. When GPU counting is enabled (by command line option), one extra thread is created to be the host side of the GPU, so that we can have all ARM cores and the GPU working in parallel. I assume that this extra CPU thread spends almost all of its time waiting for the GPU to be ready to receive a new kernel launch or data transfer command, and that this is not a spin-wait, so that the extra CPU thread does not consume any significant CPU time competing with the other threads.

Box2 Groups Speedup

Benchmark for box2 groups: seconds/gangster for gangsters 242 – 482 of gangster set 1 (gangsters 1147 – 1347). Here are benchmark times for various processors, for the original verification code and the box2 groups version.

Name	GHz	Cores	Threads	Processor	Original	Box2	Speedup
PT2017	3.10	4	8	x64 Xeon E3-1535M v6	15.7	1.40	11.2
PT2019	2.11	4	8	x64 i7-8650U	24.8	2.14	11.6
EPT2022	2.26	8	8	ARM-64 v8.2	10.6	1.68	6.3
EPT2022	1.38	8[1]	512[2]	Nvidia Volta GPU		2.70	
EPT2022				CPU + GPU		1.16	

[1]Number of streaming multiprocessors

[2]Max number of threads that can in principle execute simultaneously

With the ARM cores at 1.68 and the GPU at 2.70, one would expect the combined performance to be 1.04 (parallel resistance formula). That it is measured at 1.16 suggests that there is some competition for resources. Note that copying between host and device memory is extremely infrequent, and not the cause of any competition.

Box Numbers

0	1	2	3
4	5	6	7
8	9	10	11

Complete Run on PT2017 in 57.7 Hours

```
C:\Users\bill\Desktop\3x4>Sudoku3x4 r t0 g! fnewCount st 2
CPU using 8 threads
8.5746 [144578]; 1,181,297 cache misses 4,090,884 code calls
1,181,297 cache misses 4,090,884 code calls
Read count file newCount_0.txt, total time so far 0.00 hours
  810-  864/  865: 60200/60204   1.59/ 1.49 s/g
Read count file newCount_1.txt, total time so far 0.00 hours
11926-11988/11989: 60200/60204   1.57/ 1.43 s/g
Read count file newCount_2.txt, total time so far 0.00 hours
10456-10517/10518: 60200/60204   1.57/ 1.43 s/g
Read count file newCount_3.txt, total time so far 0.00 hours
63037-63041/63042: 60200/60204   6.88/ 1.43 s/g
Read count file newCount_4.txt, total time so far 0.00 hours
10316-10336/10337: 60200/60204   2.30/ 1.43 s/g
Read count file newCount_5.txt, total time so far 0.00 hours
44978-44981/44982: 60200/60204   8.71/ 1.43 s/g
Read count file newCount_6.txt, total time so far 0.00 hours
 1267- 1272/ 1273: 60200/60204   5.97/ 1.60 s/g
Read count file newCount_7.txt, total time so far 0.00 hours
 1518- 1518/ 1519: 60200/60204  16.19/ 1.61 s/g
Read count file newCount_8.txt, total time so far 0.00 hours
   49-   52/   53: 60200/60204   8.72/ 4.72 s/g
Max count 1110007844973287424 needs 60 bits, gangster 0 in file
newCount_0.txt
9 files, 144578 gangsters examined in 57.72 hours

Progress:
  0     865/   865  100.0%
  1   11989/ 11989  100.0%
  2   10518/ 10518  100.0%
  3   63042/ 63042  100.0%
  4   10337/ 10337  100.0%
  5   44982/ 44982  100.0%
  6    1273/  1273  100.0%
  7    1519/  1519  100.0%
  8      53/    53  100.0%
     144578/144578  100.0%
81,171,437,193,104,932,746,936,103,027,318,645,818,654,720,000 ~=
8.117144e+46

Profile tree:
```

```
Sudoku3x4                                                        207797.026
  RowCode Init                     15400 *        0.0780 ->   0.001
  ColCode Init                      5775 *        0.0570 ->   0.000
  Row Tables                      369600 *        0.1283 ->   0.047
  Column Tables                   138600 *        0.1528 ->   0.021
  BoxCompatible Init                 715 *        1.7361 ->   0.001
  Column Nodes                     31104 *       13.6582 ->   0.425
  BandGang Construct                                          0.242
  Verify BandGang Tables                                      0.026
  Band Gangsters                                              3.438
  Fix gang cache               300155625 *        0.0023 ->   0.682
  Read/verify gangsters           144578 *        1.2383 ->   0.179
  Replace cache codes          300155625 *        0.0022 ->   0.654
  Construct GridCounter          1998150 *        0.2234 ->   0.446
  Grid counter                         9 * 23087734015.9000 -> 207789.606
    GridCounter Setup                  9 *   691099.9333 ->   6.220
      Big tables                 1077444 *        5.6192 ->   6.054
      Sort                             9 *    17694.5333 ->   0.159
      Overhead                                                0.006
    Main count loop               144578 *  1437171.5317 -> 207783.386
    Overhead                                                  0.001
  Count all                                                   0.369
    Get gangster properties       144578 *        0.0040 ->   0.001
    Read count file                    9 *    25025.9889 ->   0.225
    Progress report                                           0.036
    Total grid configurations                                 0.103
    Overhead                                                  0.004
  Overhead                                                    0.887
```

Complete Run on EPT2022 in 44.7 Hours

This slight speedup from the 46.4-hour run is due entirely to faster GPU code, although the ARM cores are still doing most of the work. (I estimate 62% CPU 38% GPU.) The inner loop is changed from

```
for (int b3 = 0; b3 < 346; ++b3)
  count += (uint64_t)band1CacheLine[box7 [b3]] *
                     band2CacheLine[box11[b3]];
```

to

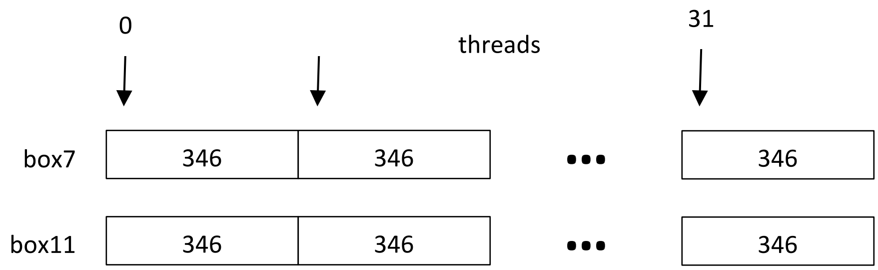
```
for (int b3 = 0; b3 < Coalesce * 346; b3 += Coalesce)
  count += (uint64_t)band1CacheLine[box711Line[b3].box7 ] *
                     band2CacheLine[box711Line[b3].box11];
```

There are two key changes:

- The box7 and box11 codes are now in one array instead of two separate arrays. This significantly reduces competition for L1 and L2 data cache lines.
- Fetches from box711Line for the threads of a warp are coalesced by a selectable factor.

In the original version, box7/11 memory is effectively

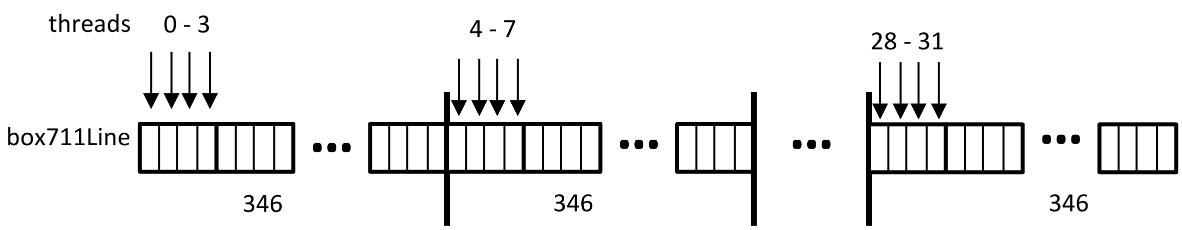
```
uint16_t box7[box2GroupSize][346], box11[box2GroupSize][346];
```



With this access pattern, none of the box7/11 fetches are coalesced. Memory is reorganized like this:

```
struct Box711 { uint16_t box7, box11; };
Box711 box711Line[ceil(box2GroupSize / Coalesce)][346][Coalesce];
```

Where `Coalesce` can be any compile-time power of 2 from 1 to 32. For example, for Coalesce = 4:



Here are some measurements with the above box2 groups benchmark, seconds/gangster over a specified set (282 – 482). The "just box7/11" column is the time to just fetch the box7/11 codes and do the 64-bit multiply-accumulate:

```
for (int b3 = 0; b3 < Coalesce * 346; b3 += Coalesce)
  count += (uint64_t) box711Line[b3].box7 * box711Line[b3].box11;
```

combine box7/11	coalesce	just box7/11	complete loop
no	1		3.19
yes	1	0.59	2.85
yes	2	0.43	2.71
yes	4	0.28	**2.64**
yes	8	0.19	2.69
yes	16	0.14	2.68
yes	32	0.12	2.66

The times have some variability, the reported values are minimums over five runs.

The Volta Tuning Guide says:

- Ensure global memory accesses are coalesced; and
- Resource-constrained kernels that are limited to low occupancy may benefit from increasing the number of concurrent memory accesses per thread.

The value of coalescing can clearly be seen in the "just box7/11" column. I don't quite know what resource-constrained and low occupancy mean, but it seems to me that better coalescing and increased number of concurrent memory accesses are tradeoffs—better coalescing will reduce the number of concurrent memory accesses, and also likely compete less for L1 and L2 with the bandCacheLine accesses. Furthermore, data cache details such as set associativity and eviction policy, about which I have not yet found documentation, can interact with the memory access pattern in quirky ways. So this may cause the odd way that box7/11 fetches seem to interact with bandCacheLine fetches, but I have no concrete explanation.

The GPU performance is quite disappointing so far. With eight streaming multiprocessors, it is still slower than PT2019, which is a Microsoft Surface laptop. It may be that the GPU's memory system is just not as good with this particular memory access pattern; it is just as likely that I haven't yet figured out how best to program the device.

Here are some things I've tried to improve GPU performance; all are slower or no better.

Kernel Launch Configuration

I've tried various multiples of 32 threads/block (`blockDim.x`); 32 is fastest, others are slower.

The blocks share 346 iterations of an outer loop, so I figured that 346 / gridDim.x should be just under an integral value, and maybe also a multiple of 8, since there are 8 SMs. So I tried 64, 72, 88, 120. 88 is fastest, others are slower.

Using Shared Memory

I tried copying box7/11 or bandGangLine values to shared memory in various ways, including async copies. It's not clear from the documentation whether async copies to shared memory are truly overlapped on GPUs of compute capability less than 8, but I don't think it matters. I can get one or both bandGangLines to fit in shared memory, but this appears to reduce L1 cache to almost nothing. That and the extra operations made all these attempts much slower.

Cache Line Alignment

I tried making sure all bandGangLines are aligned to data cache lines. It made no difference on either the CPUs (x64 or ARM) or the GPU.

Sorting Box7/11 Codes

Perhaps if the box7/11 codes are sorted, the accesses to bandGangLines will be less random and more efficient. We can only sort on one of box7 or box11, the other has to follow along with the sort. Combining box7 and box11 makes this easy. I tried std::sort, figuring if it looked promising I'd look into parallel sorting on the GPU. It did not look promising.

Streams

I wondered if I could get more parallelism by creating 8 host threads to run 8 streams, all working in parallel on separate indices of the 60204 box4/8, box 5/9 combinations. Did not look promising.

Time to learn how to use the profiler

