Project structure

The project has a completely automated profiling script to test all the settings for each problem. Please make sure your box has *jq* installed (you may install by "*sudo apt-get install jq*"), and "*cd*" to the provided directory, then run the commands "*. xtb-test-blur*" and "*. xtj-test-julia*" on bash, to respectively obtain the test run results for the performances of different settings for the Blur problem and the Julia set problem. Copies of a run of the profiling script can be found in the files "*results-blur*" and "*results-julia*".

The source files for the various settings for the Julia set problem and the Blur problem are respectively stored in the "*s**j**-xxx.cu*" and "*s**b**-xxx.cu*" files. The files for each setting of one problem are essentially all the same, except for minor tweaking of some macro parameters in the top of each source file.

Also note that the *Makefile* in this project is slightly modified to use */usr/local/cuda/* instead of the cuda directory for any specific version.

Julia problem results

Setting	***Fastest Time (ms)***	***Average Time (ms)***
CPU	853.601318	858.3643798999999
1D Indexing Grid size 1, Block size 1	2559.813721	2572.6636719999997
1D Indexing Grid size DIM*DIM, Block size 1	15.356192	15.7899361
1D Indexing Grid size DIM*DIM, Block size 3	7.507904	7.5098048
1D Indexing Grid size DIM*DIM, Block size 1024	17.24272	18.1307293
2D Indexing Grid size DIM×DIM, Block size 1×1	15.328576	15.495603099999997
2D Indexing Grid size DIM×DIM, Block size 3×3	4.460608	5.3948608
2D Indexing Grid size DIM×DIM, Block size 32×32	19.7616	20.022412600000003

When executing the Julia program on the CPU it runs for around 850ms, a baseline performance for sequential execution. Compare this to executing on the GPU with a Grid size of 1 and Block size of 1, running for around 2550ms, a 3x slowdown. By running the Julia program with a grid size and block size of 1, we are effectively forcing the GPU to run the program on one single thread only. As expected, sequential execution on the GPU is much slower than a CPU. All the other runs of the GPU however, give an execution time of around or less than 20ms. When allowed to divide up the problem and dispatch large numbers of threads simultaneously, the GPU gives a 40+x increase of performance over the CPU run.

For the 1D indexing, I have experimented with a grid size of DIM*DIM and block sizes ranging from 1, to 3, to 1024. The time for *<<<DIM*DIM,1>>>* is around 15ms, speeding up to 7ms for *<<<DIM*DIM,3>>>*, while slowing down to 17ms for *<<<DIM*DIM,1024>>>*.

In fact, for all three of these block sizes, the grid size of DIM*DIM already ensures that each block actually only needs to calculate the value of one pixel. Thus, any extra threads allocated to the block actually have no work to perform and would remain idle, despite being allocated work by the streaming multiprocessor unit. Worse, it would trigger a boundary check in the code just to realize it is remaining idle, potentially causing extra branch divergency. This explains the worse performance of *<<<DIM*DIM,1024>>>* compared to *<<<DIM*DIM,1>>>*. In fact, it is quite a testament to the efficient scheduling of the GPU, that even with 1023 of every 1024 threads issued being in fact useless, the performance only decreases by 15%. However, I cannot explain the jump in performance from *<<<DIM*DIM,1>>>* to *<<<DIM*DIM,3>>>*. In fact, since 3 is not a multiple of 32, the number of threads per physical warp, I expected the performance to drop. This performance jump also cannot be explained by occupancy, since the work is already split until the smallest quanta amongst every block, such that only one thread per block can actually perform useful work. Perhaps, this experiment could be repeated for a larger image size, to observe how significantly is performance affected by block sizes (with threads doing no useful work).

A similar effect for the performance is observed for the 2D indexing case, with the time for *<<<DIM×DIM,1×1>>>* is around 15ms, speeding up to 5ms for *<<<DIM×DIM,3×3>>>*, while slowing down to 20ms for *<<<DIM×DIM,32×32>>>*.

Regarding the readability of different indexing methods, it is no real difference in terms of the Julia set problem. Since the Julia set problem is an output only problem, it requires no caching of input data into shared memory, so whether the indexing method is 1D or 2D, it is only used to determine the position of the output pixel for the current piece of work. In fact, the code in this project basically reuses all aspects of the kernel for both 1D and 2D indexing, except for determining the thread id. Using a one dimensional number (i.e. the thread id) allows an easy method to assign a unique piece of work to each thread (by using integer division and modulus). Also, determination of the thread id for 2D indexing requires arithmetic involving the block size, which is unnecessary in 1D indexing. Thus, in this context, 2D indexing adds unnecessary code complexity and decreases the readability. Of course, on other problems, especially those

which require caching input values in a per block shared memory (such as the Blur problem), a 2D indexing would greatly assist calculation of the indexes of the relevant work.

Blur problem results

Setting	*Fastest Time (ms)*	*Average Time (ms)*
Blur filter using Constant access Input image using Shared access	0.828224	0.8739167999999999
Blur filter using Global access Input image using Shared access	0.860576	1.0206912
Blur filter using Local access Input image using Shared access	0.784288	0.9297055999999999
Blur filter using Constant access Input image using Global access	0.776608	0.839024
Blur filter using Global access Input image using Global access	0.781376	0.8892831999999998
Blur filter using Local access Input image using Global access	0.761472	0.8182112

For the Blur program, I have experimented accessing the input image by shared memory, and also by global memory. Presumably, one reason the problem asks us to compare the fastest execution times of the ten runs of the setting is to try to obtain a comparison for when all cache lookups result in a hit, so we may observe the peak potential performance of each memory access method.

For the shared memory case, accessing the blur filter array by constant memory takes a fastest time of 0.83ms, slowing down to 0.86ms for global memory, and finally providing the fastest time of 0.78ms for local memory. This is expected. At the fastest case, when all cache lookups hit, accessing constant memory takes around 5 cycles, accessing global memory takes 50-100 cycles (L2 cache), and accessing local memory takes 1-3 cycles (L1 cache).

Considering the average times instead, global memory access is still the slowest of all filter array access methods, but constant access becomes the fastest one. Comparing the average time with the fastest time, constant access only offers a slowdown of 5.5%, while global access gives a slowdown of 18.6% and local access gives a slowdown of 18.5%. Predictably, using global access and local access give higher variances in memory access times, as whenever a cache lookup misses, global and local memory takes 300-800 cycles to actually retrieve the value of the filter. In fact, even the slowdown for global access and local accesses are remarkably similar.

For the case where the input image is accessed by global memory, the trend is similar. Accessing the blur filter array by constant memory takes a fastest time of 0.776ms, slowing down to 0.781ms for global memory, and finally providing the fastest time of 0.761ms for local memory. First, note that the performances for every run with the input image accessed by global memory is slower than the corresponding time with shared memory. This is to be expected for small convolution filter sizes, such as the 3×3 filter size used in the Blur problem, since accessing the input image by shared memory entails doing extra work to first, copy the image into shared memory, before starting the actual work of calculating the convolution. Only on bigger convolution sizes will the savings from shared memory access become more apparent; as more threads need to reuse the same values, which can be pre-loaded by shared memory.

The gap in performance between using different memory accesses for the blur filter array has also decreased, from around 9.7% difference down to less than 3%. This may also be explained by the extra work and potential cache misses that happen during the process of explicitly copying the image data into shared memory.