Final Project Report

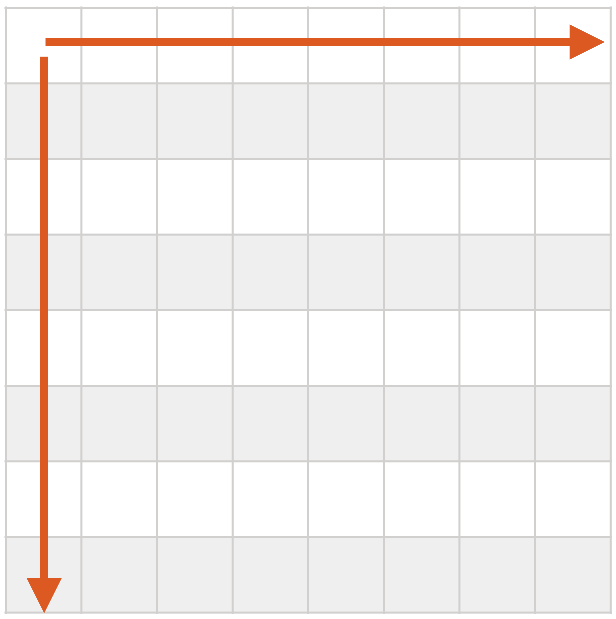
Renfei Wang, Shaowei Su

1. Description of algorithm

The solution is based on the fact that after the random swap between rows and columns, all elements of certain row/column keeps the same, which means that the XOR result of one specific row/column is fixed. Except this, we can split XOR result from one box checksum (2x2, 4x4 or 8x8) into several distinct pieces corresponds to different rows/columns. By these separate XOR values extracted from box checksum, we will be allowed to calculate the XOR values to every row and column for the original pictures. And then after simple comparison with the XOR values calculated through distorted pictures, we can swap between rows and columns to get the original picture.

The following paragraphs will demonstrate the entire unscrambled process step by step.

A. Calculate XOR values for each row and column of distorted pictures



B. Extract partial XOR value from box checksum CSV files

For example, if the box size is 8x8 then the corresponding XOR will be 64-bit. (1010498490797080327 is the column checksum to the first box in large.png_8x8.csv)

1010498490797080327 —>
0000 1110 0000 0110 0000 0011 0000 0110
0000 0110 0000 0111 0011 1111 0000 0111

Then we can split the 64-bit value into 8 pieces with each piece corresponds to XOR value of one row inside the box. Finally, we get all the partial XOR results saved to array result_xor[].

C. Combine partial XOR results into row/column XOR value
Literally, we are going to find the XOR value of entire row/column through the partial values we found in B.

D. Compare XOR values in A and C
Through comparison we can locate one specific row/column to its original position and then swap them. After all the swaps, we finally get the original pictures.

2. CUDA kernel design

The CPU version of unscrambling strictly follows our algorithm described above and the results are quite satisfying: every possible combination of PNG file and CSV file input can be done within 1 second. But there is still a chance to accelerate through GPU parallelization.

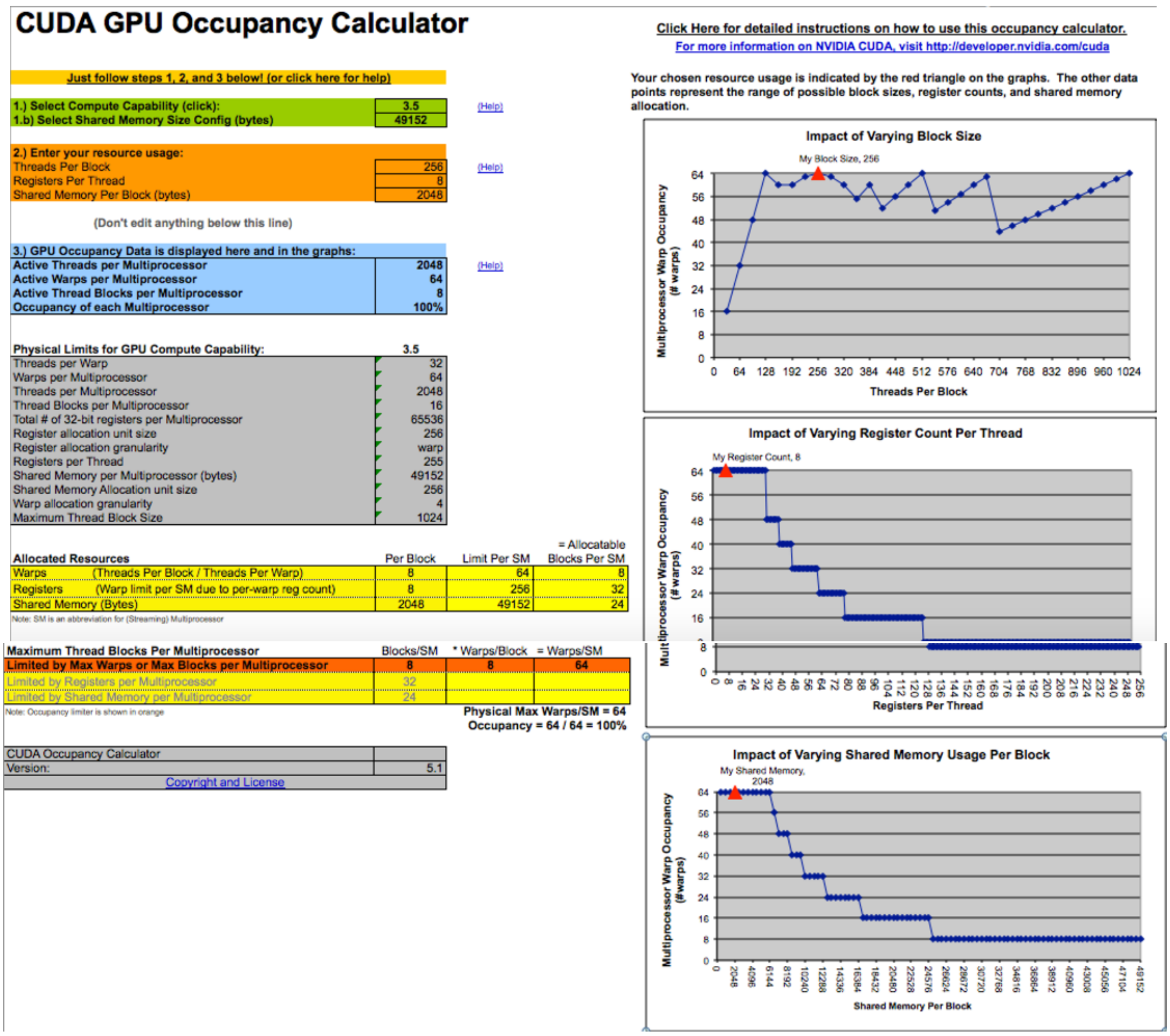
Since the XOR value can be calculated independently within boxes/rows/columns, all of the computation part with multiple for loop inside can be conducted in parallel inside kernel. Specifically, we are going to launch Number of boxes in one row * M kernels, which in the cuda program is the box_col*M.

According to the algorithm we choose, we need to calculate the xor of every row and column separately in the checkbox. Take large image and 8*8 cvs file for example, ss we just got the 64bits number, we need to divide the 64bits into 8 8bits number, so that we can get the xor of every row and column. We use the shift function to get the 8bits number and to get the xor of every row and column, we need to shift 8 times. As there are 256/8=32 boxes in each row and there are 256 columns for us to calculate(also there are 32 boxes in each column and there are 256 rows to calculate), we launch the 32 blocks which has 256 threads in it, which is also the box_col*M.(the image is M*N)

After we get the XOR, we use the synthreads and launch the first M threads to get the XOR of those results in rows and second M threads to get the XOR of the results in columns and the third M threads to calculate the row XOR in the scramble image and the fourth M threads to calculate the column XOR in the scramble image.

As in our cud program, when we are calculating the row of each entire row and column, we need to XOR the results from the box xor function. When we are XOR the results, we will XOR data itself M times. If this data is in the global memory, it will absolutely cost more time. So we put these data into the shared memory to save the execution time. As each thread will use one shared data many times(other thread will not use it), so there are no bank conflicts of these shared data.

The Cuda Occupancy Calculator is below:



CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	3.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	8	
Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	3.5
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	8	64	8
Registers	(Warp limit per SM due to per-warp reg count)	8	256	32
Shared Memory (Bytes)		2048	49152	24

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	8	8	64
Limited by Registers per Multiprocessor	32		
Limited by Shared Memory per Multiprocessor	24		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

CUDA Occupancy Calculator	
Version:	5.1
Copyright and License	

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Impact of Varying Block Size
My Block Size, 256
Multiprocessor Warp Occupancy (# warps)
Threads Per Block
Impact of Varying Register Count Per Thread
My Register Count, 8
Multiprocessor Warp Occupancy (# warps)
Registers Per Thread
Impact of Varying Shared Memory Usage Per Block
My Shared Memory, 2048
Multiprocessor Warp Occupancy (#warps)
Shared Memory Per Block

3. Results

3.1 Execution time

#1 CPU Execution Time(ms)

Image File/CSV File	2*2	4*4	8*8
large	2.044	1.582	1.364
medium	0.314	0.262	0.225
small	0.057	0.052	0.052

#2 GPU Execution Time(ms)

Image File/CSV File	2*2	4*4	8*8
large	0.49	0.49	0.50
medium	0.20	0.20	0.21
small	0.10	0.10	0.11

3.2 Unscramble result