4η εργασία

Σταμούλος Αλέξανδρος 02954

Γαλάνης Αχιλλέας 02941

|  |
| --- |
| ./deviceQuery Starting...  CUDA Device Query (Runtime API) version (CUDART static linking)  Detected 2 CUDA Capable device(s)  Device 0: "Tesla K80"  CUDA Driver Version / Runtime Version 11.4 / 11.5  CUDA Capability Major/Minor version number: 3.7  Total amount of global memory: 11441 MBytes (11997020160 bytes)  (13) Multiprocessors, (192) CUDA Cores/MP: 2496 CUDA Cores  GPU Max Clock rate: 824 MHz (0.82 GHz)  Memory Clock rate: 2505 Mhz  Memory Bus Width: 384-bit  L2 Cache Size: 1572864 bytes  Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)  Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers  Total amount of constant memory: 65536 bytes  Total amount of shared memory per block: 49152 bytes  Total number of registers available per block: 65536  Warp size: 32  Maximum number of threads per multiprocessor: 2048  Maximum number of threads per block: 1024  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)  Maximum memory pitch: 2147483647 bytes  Texture alignment: 512 bytes  Concurrent copy and kernel execution: Yes with 2 copy engine(s)  Run time limit on kernels: No  Integrated GPU sharing Host Memory: No  Support host page-locked memory mapping: Yes  Alignment requirement for Surfaces: Yes  Device has ECC support: Enabled  Device supports Unified Addressing (UVA): Yes  Device supports Compute Preemption: No  Supports Cooperative Kernel Launch: No  Supports MultiDevice Co-op Kernel Launch: No  Device PCI Domain ID / Bus ID / location ID: 0 / 6 / 0  Compute Mode:  < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >  Device 1: "Tesla K80"  CUDA Driver Version / Runtime Version 11.4 / 11.5  CUDA Capability Major/Minor version number: 3.7  Total amount of global memory: 11441 MBytes (11997020160 bytes)  (13) Multiprocessors, (192) CUDA Cores/MP: 2496 CUDA Cores  GPU Max Clock rate: 824 MHz (0.82 GHz)  Memory Clock rate: 2505 Mhz  Memory Bus Width: 384-bit  L2 Cache Size: 1572864 bytes  Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)  Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers  Total amount of constant memory: 65536 bytes  Total amount of shared memory per block: 49152 bytes  Total number of registers available per block: 65536  Warp size: 32  Maximum number of threads per multiprocessor: 2048  Maximum number of threads per block: 1024  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)  Maximum memory pitch: 2147483647 bytes  Texture alignment: 512 bytes  Concurrent copy and kernel execution: Yes with 2 copy engine(s)  Run time limit on kernels: No  Integrated GPU sharing Host Memory: No  Support host page-locked memory mapping: Yes  Alignment requirement for Surfaces: Yes  Device has ECC support: Enabled  Device supports Unified Addressing (UVA): Yes  Device supports Compute Preemption: No  Supports Cooperative Kernel Launch: No  Supports MultiDevice Co-op Kernel Launch: No  Device PCI Domain ID / Bus ID / location ID: 0 / 7 / 0  Compute Mode:  < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >  > Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU1) : Yes  > Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU0) : Yes  deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA Runtime Version = 11.5, NumDevs = 2, Device0 = Tesla K80, Device1 = Tesla K80  Result = PASS |

Compile στην cpu: gcc -O3 main.c contrast-enhancement.c histogram-equalization.c -o main

Compile στην gpu: nvcc -O3 main.cu contrast-enhancement.cu histogram-equalization.cu -o main

Χωρίσαμε την συνάρτηση histogram\_equalization σε δυο kernels , ένα για τον υπολογισμό το lut και έναν για την αποθήκευση του result image καθώς χρησιμοποιούν διαφορετική γεωμετρία threads για το lut 1 block με 256 threads και για το result 1 thread για κάθε pixel. Επίσης αφαιρέσαμε το if

if(lut[img\_in[i]] > 255){

img\_out[i] = 255;

}

Αφού το lut παίρνει τιμές μέχρι 255 γιατί

lut[i] = (int) + 0.5 το cdf <= img\_size άρα το κλάσμα παίρνει μέγιστη τιμή 255 . Προσθέτουμε 0.5 και επειδή κάνουμε typecast σε int η μέγιστη τιμή είναι 255

O kernel histogram\_result είναι αρκετά απλός το κάθε thread υπολογίζει την κατάλληλη θέση και ανανεώνει το img\_out. Αφαιρέσαμε

Για τον υπολογισμό το lut το thread 0 υπολογίζει το min (πρώτη μη μηδενική τιμή του hist) , έπειτα υπολογίζουμε με parallel scan with reduction την cdf. Τέλος υπολογίζουμε το lut .

O kernel histogram είναι αυτός που έκανε τον μεγαλύτερο χρόνο όπως είδαμε με profiling. Αρχικά κάναμε ένα per grid implementation που χρησιμοποιούμε global atomics

if(index < img\_size){

atomicAdd(&hist\_out[img\_in[index]], 1);

}

Αυτό το implementation είναι αρκετά αργό αφού ο υπολογισμός του ιστογράμματος είναι αρκετά data dependent και όταν πολλά threads πάνε να γράψουν στο ίδιο κελί έχουμε ακολουθιακό εκτέλεση.

Οπότε δοκιμάσαμε per-block histogram που το κάθε block υπολογίζει ένα histogram στο shared memory και μετρά γίνεται συγχώνευση. Έχουμε μικρή βελτίωση αφού τα shared atomics είναι πιο γρηγορά και έχουμε λιγότερα collisions.

Για να μειώσουμε τα collision ο kernel καλείται με λιγότερα blocks και τα threads υπολογίζουν περισσότερα pixels. Παρατηρήσαμε σημαντική μείωση στον χρόνο. συγκεκριμένα απ΄το profiling με 1 pixel ανά thread ο kernel έπαιρνε το 38% του χρόνου εκτέλεσης για είσοδο το **fort.pgm** με περισσότερα pixels ανά thread έπεσε σε 26%.

Τέλος κάναμε κάποιες μικρές βελτιστοποιήσεις . Κάναμε το for loop interleaved καθώς συχνά στις εικόνες γειτονικά pixel έχουν ιδιά τιμή. Και κάναμε loop unrolling με βήμα 4 αρά το μέγεθος εικόνας πρέπει να είναι πολλαπλάσιο του 4 . Κάτι που σχεδόν πάντα συμβαίνει καθώς το image width και image height είναι πολλαπλάσια του 2 αρά το γινόμενο τους είναι πολλαπλάσιο του 4.

Unrolling:

unsigned int value = ((unsigned int \*) img\_in)[i];

Αφού το img\_in είναι unsigned char όταν το κάνουμε typecast σε unsigned int το value παίρνει τις πρώτες 4 τιμές που έπειτα τις διαβάζουμε με value & 0xff και προχωράμε στην επόμενη με value >>= 8. Μετά από αυτές τις βελτιστοποιήσεις φτάσαμε το ποσοστό στο 21%.

Μετά από δοκιμές σε όλες τις εικόνες παρατηρήσαμε ότι το optimal grid size είναι 600.