|  |
| --- |
| #include "kernel.h" |
|  | #include <stdio.h> |
|  | #include <stdlib.h> |
|  | #include <cuda.h> |
|  | #include <cutil.h> |
|  | #include <cuda\_runtime.h> |
|  |  |
|  |  |
|  | #include <assert.h> |
|  |  |
|  | \_\_device\_\_ int get\_index\_to\_check(int thread, int num\_threads, int set\_size, int offset) { |
|  |  |
|  | // Integer division trick to round up |
|  | return (((set\_size + num\_threads) / num\_threads) \* thread) + offset; |
|  | } |
|  |  |
|  | \_\_global\_\_ void p\_ary\_search(int search, int array\_length, int \*arr, int \*ret\_val ) { |
|  |  |
|  | const int num\_threads = blockDim.x \* gridDim.x; |
|  | const int thread = blockIdx.x \* blockDim.x + threadIdx.x; |
|  |  |
|  | //ret\_val[0] = -1; |
|  | //ret\_val[1] = offset; |
|  |  |
|  | int set\_size = array\_length; |
|  |  |
|  |  |
|  | while(set\_size != 0){ |
|  | // Get the offset of the array, initially set to 0 |
|  | int offset = ret\_val[1]; |
|  |  |
|  | // I think this is necessary in case a thread gets ahead, and resets offset before it's read |
|  | // This isn't necessary for the unit tests to pass, but I still like it here |
|  | \_\_syncthreads(); |
|  |  |
|  | // Get the next index to check |
|  | int index\_to\_check = get\_index\_to\_check(thread, num\_threads, set\_size, offset); |
|  |  |
|  | // If the index is outside the bounds of the array then lets not check it |
|  | if (index\_to\_check < array\_length){ |
|  |  |
|  | // If the next index is outside the bounds of the array, then set it to maximum array size |
|  | int next\_index\_to\_check = get\_index\_to\_check(thread + 1, num\_threads, set\_size, offset); |
|  |  |
|  | if (next\_index\_to\_check >= array\_length){ |
|  | next\_index\_to\_check = array\_length - 1; |
|  | } |
|  |  |
|  | // If we're at the mid section of the array reset the offset to this index |
|  | if (search > arr[index\_to\_check] && (search < arr[next\_index\_to\_check])) { |
|  | ret\_val[1] = index\_to\_check; |
|  | } |
|  | else if (search == arr[index\_to\_check]) { |
|  | // Set the return var if we hit it |
|  | ret\_val[0] = index\_to\_check; |
|  | } |
|  | } |
|  |  |
|  | // Since this is a p-ary search divide by our total threads to get the next set size |
|  | set\_size = set\_size / num\_threads; |
|  |  |
|  | // Sync up so no threads jump ahead and get a bad offset |
|  | \_\_syncthreads(); |
|  | } |
|  | } |
|  |  |
|  |  |
|  | int chop\_position(int search, int \*search\_array, int array\_length) |
|  | { |
|  | // Get the size of the array for future use |
|  | int array\_size = array\_length \* sizeof(int); |
|  |  |
|  | // Don't bother with small arrays |
|  | if (array\_size == 0) return -1; |
|  |  |
|  | // Setup array to use on device |
|  | int \*dev\_arr; |
|  | cudaMalloc((void\*\*)&dev\_arr, array\_size); |
|  |  |
|  | // Copy search array values |
|  | cudaMemcpy(dev\_arr, search\_array, array\_size, cudaMemcpyHostToDevice); |
|  |  |
|  | // return values here and on device |
|  | int \*ret\_val = (int\*)malloc(sizeof(int) \* 2); |
|  | ret\_val[0] = -1; // return value |
|  | ret\_val[1] = 0; // offset |
|  | array\_length = array\_length % 2 == 0 ? array\_length : array\_length - 1; // array size |
|  |  |
|  | int \*dev\_ret\_val; |
|  | cudaMalloc((void\*\*)&dev\_ret\_val, sizeof(int) \* 2); |
|  |  |
|  | // Send in some intialized values |
|  | cudaMemcpy(dev\_ret\_val, ret\_val, sizeof(int) \* 2, cudaMemcpyHostToDevice); |
|  |  |
|  | // Launch kernel |
|  | // This seems to be the best combo for p-ary search |
|  | // Optimized around 10-15 registers per thread |
|  | p\_ary\_search<<<16, 64>>>(search, array\_length, dev\_arr, dev\_ret\_val); |
|  |  |
|  | // Get results |
|  | cudaMemcpy(ret\_val, dev\_ret\_val, 2 \* sizeof(int), cudaMemcpyDeviceToHost); |
|  |  |
|  | int ret = ret\_val[0]; |
|  |  |
|  | printf("Ret Val %i Offset %i\n", ret, ret\_val[1]); |
|  |  |
|  | // Free memory on device |
|  | cudaFree(dev\_arr); |
|  | cudaFree(dev\_ret\_val); |
|  |  |
|  | free(ret\_val); |
|  |  |
|  | return ret; |
|  | } |
|  |  |
|  | // Test region |
|  | static int \* build\_array(int length) { |
|  |  |
|  | int \*ret\_val = (int\*)malloc(length \* sizeof(int)); |
|  |  |
|  | for (int i = 0; i < length; i++) |
|  | { |
|  | ret\_val[i] = i \* 2 - 1; |
|  | } |
|  |  |
|  | return ret\_val; |
|  | } |
|  |  |
|  | static void test\_array(int length, int search, int index) { |
|  |  |
|  | printf("Length %i Search %i Index %i\n", length, search, index); |
|  | assert(index == chop\_position(search, build\_array(length), length) && "test\_small\_array()"); |
|  |  |
|  | } |
|  |  |
|  | static void test\_arrays() { |
|  |  |
|  | test\_array(200, 200, -1); |
|  |  |
|  | test\_array(200, -1, 0); |
|  |  |
|  | test\_array(200, 1, 1); |
|  |  |
|  | test\_array(200, 29, 15); |
|  |  |
|  | test\_array(200, 129, 65); |
|  |  |
|  | test\_array(200, 395, 198); |
|  |  |
|  | test\_array(20000, 395, 198); |
|  |  |
|  | test\_array(2000000, 394, -1); |
|  |  |
|  | test\_array(20000000, 394, -1); |
|  | } |
|  |  |
|  |  |
|  | int main(){ |
|  | test\_arrays(); |
|  | } |