International Journal of Scientific Research in Computer Science and Engineering
Research Paper Vol-1, Issue-4 ISSN: 2320-7639

Histogram Computations on GPUs Kernel using Global and Shared Memory Atomics

C.P.Patidar[1*] and Meena Sharma[2]

[*1]*Department of Information Technology, Institute of Engineering and Technology, DAVV, India*
[2]*Department of Computer Engineering, Institute of Engineering and Technology, DAVV, India*

Available online at www.isroset.org

Received: 28 June 2013 Revised: 12 July 2013 Accepted: 06 August 2013 Published: 30 August 2013

Abstract– In this paper we implement histogram computations on a Graphics Processing Unit (GPU). Our Histogram computations is implemented using compute unified device architecture (CUDA) which is a minimal extension to C/C++. In this development Histogram computations, computed on GPU's global memory as well as on shared memory. We also perform Histogram computations on CPU and consider it as a baseline performance. Experimental results demonstrate that shared memory in GPU gives seven times speedup over our baseline CPU.

Keywords – GPUs, Histograms, CUDA, Global Memory, Shared Memory.

I. INTRODUCTION

A Graphics Processing Unit or a GPU [1] is a chip in computer video cards or in play station. GPUs are usually used in personal computers and on video game consoles. Primarily it is used for entertainment and visualization applications. Now a days GPUs are also found in portable devices due to their user interfacing and more software applications interaction [1]. GPUs are now also found frequently in portable devices, embedded systems and consumer electronics devices. Fig 1 shows the development growth of GPU over CPU in recent years. GPUs come with many cores, now a day they come with up to 512 cores. There are two major vendors of GPU NVIDIA and AMD, formerly known as ATI.

GPU programming environment for GPGPU (general purpose GPU) with CUDA (compute unified device architecture) [2] is based on C. CUDA is a minimal extension to C environment. CUDA provides heterogeneous programming model, which means serial program with parallel kernels. In which serial codes executes on host (CPU) and parallel codes executes on device threads (GPU threads). CUDA can provide large speedups on data parallel applications and algorithm which require huge data computations [2]. Image processing, Data mining, Sequence alignment, Histogram computations, and many bioinformatics applications [12] require huge amount of data computations [5].

CPU gets overwhelms in such types of applications where huge amount of data computations are required. GPUs give better performance in such types of applications due to their parallel and multi core architecture [3].

Histogram computation is also suitable application that can be implemented on GPUs and we can get many fold improvement over CPU. Oftentimes, algorithms require the computation of a histogram of some set of data. Essentially, given a data set that consists of some set of elements, a histogram represents count of the frequency of each element.

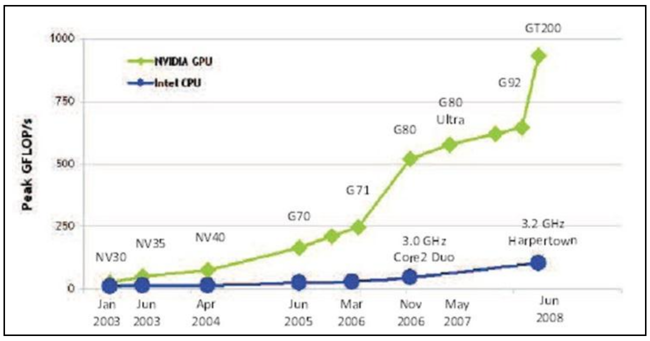

Fig 1: Growth of the power of GPUs vs. CPUs in the last few years (Adapted from NVIDIA [2]).

For example, if we created a histogram of the letters in the phrase "HISTOGRAM COMPUTATION ON GPU", the result will be as shown in Fig 2. Histogram computations are used in computer science for image processing, data compression, computer vision, machine learning, audio encoding, and many others.

1	2	1	3	4	2	1	2	2	1	2	2	2
H	I	S	T	O	G	R	A	M	C	P	U	N

Fig 2: Letter frequency histogram built from the string *"HISTOGRAM COMPUTATION ON GPU"*

We require an environment in which we can develop using CUDA C [2]. The prerequisites to computing histograms in CUDA C are as follows:

- A CUDA-enabled graphics processor (Graphics Card).
- An NVIDIA device driver (System Software) for performing communication between our program and CUDA-enabled hardware.

Corresponding Author: *C. P. Patidar*

- A CUDA development toolkit to compile and run code for NVIDIA GPUs using CUDA C.
- We will also need a compiler for CPU to run our programs on CPU.

In this paper our work emphasized use of shared memory because shared memory is so faster than global memory. It may be hundreds or more time faster. Threads cannot cooperate via global memory but they can cooperate via shared memory. For taking the advantage of shared memory, we scheduled some computation on the device that forms blocks. Also partition our data sets into data subsets so they can be fitted into shared memory [5].

After partitioning we handle each data set with one thread block. For this we loaded the subset form global memory to shared memory then perform computation on the subset from shared memory. At the end we copy results from shared memory to global memory.

The organization of paper is as follows: We review the used GPU architecture in section II and then describe the atomic operations in section III. In section IV we describe the computations of Histogram on CPU to calculate our baseline performance then in section V describe computations on global memory and then on shared memory. Experimental results that show speedups of GPU are shown in section VI. Conclusion of paper is given in section VII.

II. GT 610 GPU ARCHITECTURE

Our work implemented on NVIDIA GeForce GT 610 GPU. GeForce GT 610 comprises 48 CUDA cores.
The 48 CUDA cores access a common 2GB of DRAM memory, known as device or global memory through a 64 KB L2 cache. The GPU operates at clock rate 1.62 GHz with memory clock rate 533 MHz. The memory bus width is 64-bit and the total amount of constant memory is 64 KB. The amount of shared memory which is available per block is 48 KB and the total number of registers available per block is 32 KB. The maximum number of threads per multiprocessor in our GPU is 1.5 KB and the maximum number of threads per block in our GPU is 1.5 KB.GT 610 support 1024*1024*64 maximum size for each dimension of a block. Also GT 610 supports 65535*65535*65535 maximum size for each dimension of a grid. A GT 610 connects to the CPU, called host processor via a PCI bus. GT 610 supports master-slave programming model [2]. In which we can write a program for master that is host and this program execute on the device that is GT 610. GT 610 supports CUDA programming language also which is a minimal extension to C/C++. To achieve high performance on GT 610 we used blocks and distribute the task on shared memory and this is the main focus of this paper.

III. ATOMIC OPERATIONS

Usually atomic operations are not used when writing traditional single-threaded applications. One might need atomic operations in multithreaded applications. For example in C, the decrement operator:
A--;

This is a single expression in standard C\C++, and after executing this expression, the value in A should be one less than it was prior to executing the decrement. But what sequence of operations does this imply? To subtract one from the value of A, we first need to know what value is currently in A. After reading the value of A, we can modify it. And finally, we need to write this value back to A. So the three steps in this operation are as follows:

- Read the value in A.
- Subtract 1 from the value read in step 1.
- Write the result back to A.

Sometimes, this process is generally called a read-modify-write operation, since step 2 can consist of any operation that changes the value that was read from A.
There may be a chance where two threads want to perform decrement on the value in A. As an example the two threads are x and y. Now x and y both want to decrement value in A. x and y both want to perform operations read, subtract and write as mentioned above.
x and y both give six operations in total. The six operations may be in any sequence. In this way they can give wrong result. It means if threads gets executed in any order we will be get unpredictable results. In this example multiple threads want to read or write shared memory (values). For getting correct results we need a way to perform these three operations (read, subtract or modify and write) without being interrupted by another thread.
Such operations are known as “atomics” because the execution of these operations cannot be split into smaller parts by other threads. CUDA provides facility for atomic operations that supports to operate safely on memory also when many threads wants to access the same memory locations [11]. Histogram computation is also such an application that requires the use of atomic operations to compute the correct sum.

IV. HISTOGRAM COMPUTATIONS ON CPU

This section shows Histogram computations on the Intel Core 2 Duo CPU 2.20 GHz, CPU. There are two reasons of computing a histogram on CPU in this paper. The primary reason is for taking baseline performance and another one is for understanding Histogram computations on a single threaded CPU. Histogram may be computed for pixels, audio, video, biological sequences or any random stream of bytes [8]. In this paper we approach Histogram computations for randomly generated stream of bytes. We use a utility function rand() to generate random stream of bytes. We can create any size of random data by using this rand() function. We create N MB (Here N may be 50, 100, 200 or any data value) stream of random data as our data samples. Each random 8-bit byte can vary from 0 to 255(from 0x00 to 0xFF). In our application histogram needs to contain 256 bins for keeping the track of the occurrence of each value in the data. For this purpose we have to define an array of size 256 and initially fill all the 256 bins with zero. Now we need to compute the frequency of the occurrence of the value generated by function rand() with a dynamic memory allocation function. The generated memory locations are contained in a pointer of type unsigned char. and the pointer is named as “location”. If we see any value from 0 to 255 in array location[] (Here [] shows location is an array) we want to increment the value of the corresponding array named as bin[] (Here also [] shows bin is an array) [9].

We need to increment the value we have in the bin numbered location[i] in array. Since bin location[i] in array is located at bin[location[i]] in array, we can increment the desired counter as follows:
bin[location[i]]++;

We perform this operation for each element in location array. In this way, we’ve completed Histogram computations for the input data. Histogram sum should always be same, regardless of the random input array [8]. Every bin counts the numbers of times we have find the corresponding data element, thus the sum of all of these bins will be the total number of data elements we’ve encountered [9]. The execution time and sum for different values of N is shown in section V RESULTS. In the next section we will compute the same for GPU with CUDA.

V. HISTOGRAM COMPUTATIONS ON GPU

A. Computation on global memory

This section shows histogram computations on the GPU’s global memory. When our data size is very large and different threads examining different parts of the location it might give a speedup. When multiple threads may want to increment the same bin of the output histogram at the same time a problem arises. To handle such problems we use atomic increments to get rid of a situation like the one discussed in section III ATOMIC OPERATIONS. The concept is similar to CPU implementation of Histogram computation. Here we CUDA C to computer Histograms on GPU and get results from the GPU. We initialize events for timing.

```
cudaEvent_t     start, stop;
```

After this, we work on GPU memory. We allocate space for randomly generated input data and output histogram. After allocating space to the input buffer, we copy the array loation[] we generated with random() function to the GPU [6]. After this allocation of the histogram, we set all 256 elements of location [] array to zero just like we did in the section

VI. HISTOGRAM COMPUTATIONS ON CPU.

```
unsigned char *dev_location;
unsigned int *dev_bin;
cudaMemcpy (dev_location, location, N*1024*1024,
cudaMemcpyHostToDevice ) );
```

Here cudaMemset() is a CUDA runtime function. This function is similar to the standard C function memset(). The C library function memset() does not returns an error code while cudaMemset() return [4]. The returned error code informs the caller whether something wrong happened while attempting to GPU memory. memset() operates on host memory while cudaMemset() operates on GPU memory.

Now the next step is launching the histogram kernel. We have performed computations of the histogram on the GPU. After all, we copied the histogram back to the CPU. Thus perform a copy from device to host by writing [4].
cudaMemcpy(bin, dev_bin,256 * sizeof(int), cudaMemcpyDeviceToHost)

Finally we verify that the computed GPU histogram matches with the output of the CPU.

Our kernel launch is more complicated because of performance concerns. In our application the histogram contains 256 bins. Use of 256 threads per block is convenient and gives high performance as results. As a sample data with 200MB of data, the total numbers of bytes are 209,715,200.

There are two ways of launching the kernel:

- Launch a single block and have each thread examine 819,200 data elements.
- Launch 819,200 blocks and have each thread examine a single data element.

The optimal solution is at a point between these two extremes. When we execute some performance experiments, optimal performance is obtained when the number of blocks we launch is exactly twice the number of multiprocessors GPU comprises. GT 610 has 48 multi- processors, thus histogram kernel executes faster on a GT 610 when launched with 96 parallel blocks. We have methods for querying various properties of the GPU on which our application is executing. We use multiProcessorCount device property to dynamically size our launch based on our current hardware platform [10].

This gives the number of block in GPU.

```
cudaDeviceProp p;
cudaGetDeviceProperties( &p, 0 );
int blocks = p.multiProcessorCount;
```

The kernel that computes the histogram is given a pointer to the input data array, the length of the input array, and a pointer to the output histogram. Initially the kernel computes a linearized offset into the input data array. Each thread starts with an offset between 0 and the number of threads minus 1. Then it strides by the total number of threads that have been launched.

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
int stride = blockDim.x * gridDim.x;
```

As soon as each thread knows it's starting offset i and the stride it should use, the control passes through the input array incrementing the appropriate histogram bin.

After comparing it with baseline performance, we find that this performance is almost equivalent to the baseline performance as shown in section V RESULTS. It gives not a better improved performance whether it runs on the GPU. It shows that the atomic operation on global memory is causing the problem.

When numbers of threads are trying to refer same memory locations, there is a chance of occurrence of contention for 256 histogram bins. This leads in a long queue of pending jobs, and there is no performance gain. We ensure atomicity of the decrement operation, if and only if the hardware is serialized. We try to improve the performance by using shared memory in the next part.

B. Computation on shared memory

In this section, each parallel block will compute a separate histogram in shared memory. This saves the time of transfer between host and device. Now atomic operations are required within the block because multiple threads within the block can still examine data elements with the same value as in Fig. 3 depicted [11]. It shows per-thread local memory, per block-shared memory and per device global memory. When we compute histograms on global memory there are thousands of threads were competing. As a great improvement here only 256 threads will be competing for 256 addresses and it reduces contention from the global memory implementation. It involves allocating and initializing a shared memory location to hold each block's intermediate histogram. Our approach is such that every thread's write has completed before the subsequent reading and modifying the location [8].

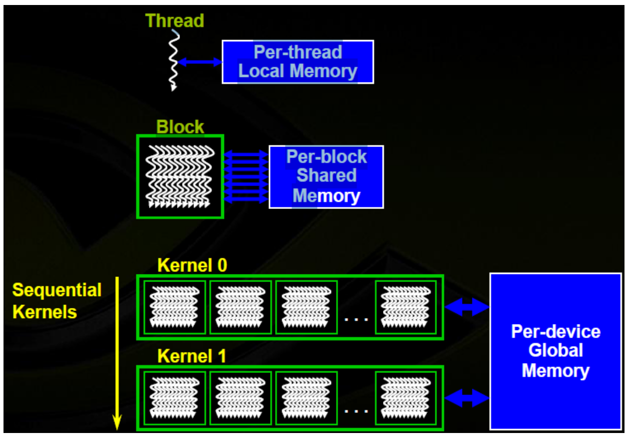

Fig 3: Per-thread local memory, Per-thread shared memory, Per-device global memory.

After initializing the histogram, the next step is same as our previous GPU histogram computation on global memory. Here we use the shared memory buffer of array temp[] instead of the global memory buffer of array bin[] and that we call a function __syncthreads() [4]to ensure that the last write is unmodified [10].

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
int offset = blockDim.x * gridDim.x;
__syncthreads();
```

At last we merge each block's temporary histogram into the global buffer array bin[]. We break the input in two parts. One thread look at one half and another thread look at other half. In this manner every thread computes separate histograms. If thread P encounters byte 0xFC 100 times in the input and thread Q encounters byte 0xFC 50 times, the byte 0xFC appeared 150 times in the input. In this way each bin of the final histogram is the sums of the corresponding bins in thread P's histogram and thread Q's histogram. We extend this method for all threads or for any number of threads, thus merging every block's histogram into a single final histogram comprises adding each entry in the block's histogram to the corresponding entry in the final histogram.

In shared memory we use 256 threads and 256 histogram bins; each thread atomically adds a single bin to the final histogram's total [9].

This implementation of our Histogram computations improves dramatically over the previous GPU implementation. Adding the shared memory component drops our running time on a GT 610 to a considerable amount. The results are shown in next section.

VII. EXPERIMENTAL RESULTS

We perform Histogram computations on our benchmark machine, Intel Core 2 Duo CPU 2.20 GHz, and on GeForce GT 610 GPU for different data samples.
We take results on different data samples from both global and shared memory on GT 610 GPU. The results for 50MB, 100MB, 200MB, 300MB and 400MB is shown in TABLE I.

On GT 610 GPU, with Global memory but without Shared memory, the Histogram computation for different data samples not gives a not a better performance gain over the benchmark machine (CPU). For example 200MB array of data can be constructed in 569.3 ms on GPU whereas it can be constructed in 690 ms on CPU as shown in TABLE I. In fact, this is not a better performance gain over the benchmark machine (CPU). It is a low-performance implementation simply because it runs on the GPU.

On GT 610 GPU, with Shared memory, the Histogram computation for different data samples gives up to sevenfold speedups. As shown in TABLE I the 100MB array of data can be constructed in 74.1 ms on GT 610 whereas it is constructed on CPU in 350 ms. It improves dramatically over the CPU as well as GT 610 GPU, with Global memory. After adding the shared memory it drops running time on a GT 610 irrespective of the size. The pictorial experimental results are shown in the form of a graph in Fig. 4.

Histogram counts for different data samples (Size in MB) are as follows:

- For 50 MB Histogram sum is 54428800.
- For 100 MB Histogram sum is 104857600.
- For 200 MB Histogram sum is 209715200.
- For 300 MB Histogram sum is 414572800.
- For 400 MB Histogram sum is 419430400.

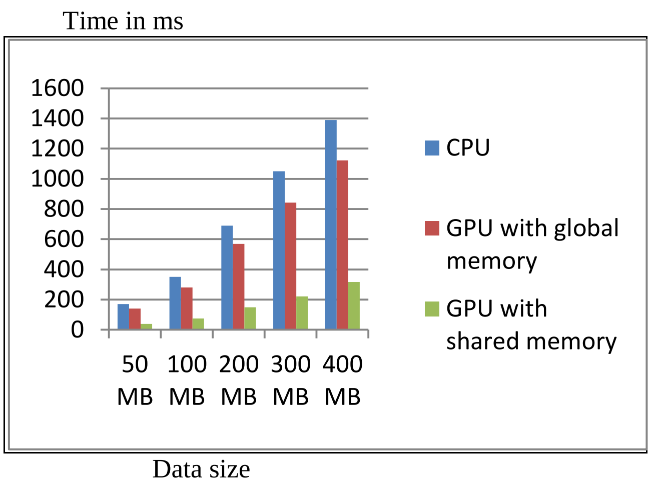

Fig 4: Graph that shows results on CPU as well as on GPU with Global and Shared memory.

TABLE I
EXECUTION TIME FOR VARIOUS MEMORY SIZE

Processor	Execution Time in ms for various memory size(in MB)				
	50	100	200	300	400
Intel Core 2 Duo CPU 2.20 GHz	170	350	690	1050	1390
GT 610 GPU,1.62 GHz (global memory)	140.9	281	569.3	841.1	1122.2
GT 610 GPU,1.62 GHz (shared memory)	38.8	74.1	147.8	220	316.3

VIII. CONCLUSION AND FUTURE WORK

In this paper we have presented the implementation of the Histogram computations on GPU. For this purpose we use CUDA [2] programming language. Hence the result represents improvement up to a sevenfold boost in speed over the benchmark machine. Thus despite the early GPU implementation with global memory in adapting the Histogram computations, our implementation that uses shared memory (atomics) should be considered a success. In our opinion, CUDA is also built upon a solid software and hardware platform. Thus CUDA is also a promising technology [7].

One future work is to use GPU and CUDA architecture for the implementation of sequence alignment as a bioinformatics application [2]. Shared memory atomics will be appropriate match for sequence alignment [12].

ACKNOWLEDGMENT

We wish to thank Vaibhav Jain for their helpful insight and support.

REFERENCES

[1]. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÄuger, A. E. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, pp. 80-113, Mar. 2007.
[2]. NVIDIA Corporation, CUDA: Compute Unified Device Architecture Programming Guide," tech. rep., NVIDIA, 2007.
[3]. S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan Primitives for GPU Computing,"in *GH '07: Proceedings of the 22nd ACSIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, Switzerland, Eurographics Association, 2007, pp. 97-106.
[4]. M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson. "CUDPP: CUDA Data Parallel Primitives Library".
[5]. P. Harish and P. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA" in *High Performance Computing HiPC 2007*, pp. 197-208.
[6]. H. Nguyen, *GPU Gems 3*. Addison-Wesley Professional, 2007.
[7]. Y. Luo and R. Duraiswami, "Canny Edge Detection on Nvidia CUDA" in *Proc. of IEEE Computer Vision and Pattern Recognition*, 2008, pp. 1-8.
[8]. V. Podlozhnyuk, "64-bin histogram", NVIDIA, Tech. Rep., 2007.
[9]. K. H. Knuth, "Optimal data-based binning for histograms," ArXiv Physics e-prints, May 2006.
[10]. Compute Unified Device Architecture (CUDA) Programming Guide.
[11]. NVIDIA. CUDA Compute Unified Device Architecture Programming Guide 2.0, July 2008.
[12]. C. Ling, K. Benkrid, and T. Hamada, "A parameterisable and scalable smith-waterman algorithm implementation on cuda- compatible gpus," Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on, pp. 94–100, jul. 2009

C. P. Patidar received the B. E. degree in information technology and pursuing M.E. degree in computer engineering. He is an assistant professor of Information Technology at the Devi Ahilya University Indore, India. His research interests are in GPGPU computing, CUDA programming multithreaded architecture and memory architecture of computers.

Meena Sharma received the B.E. degree in computer engineering and M. Tech. degree in computer science in 1992 and 2004 respectively. She received the Ph. D. Degree in computer engineering in 2012. She is an associate professor of Computer Engineering at the Devi Ahilya University Indore, India. Her research interests are in software engineering, software quality matrices and object oriented modelling and design.