Efficient Integral Image Computation on the GPU

2010 IEEE Intelligent Vehicles Symposium
University of California, San Diego, CA, USA
June 21-24, 2010

Efficient Integral Image Computation on the GPU

Berkin Bilgic, Berthold K.P. Horn, Ichiro Masaki

Abstract—**We present an integral image algorithm that can run in real-time on a Graphics Processing Unit (GPU). Our system exploits the parallelisms in computation via the NIVIDA CUDA programming model, which is a software platform for solving non-graphics problems in a massively parallel high-performance fashion. This implementation makes use of the work-efficient scan algorithm that is explicated in [5]. Treating the rows and the columns of the target image as independent input arrays for the scan algorithm, our method manages to expose a second level of parallelism in the problem. We compare the performance of the parallel approach running on the GPU with the sequential CPU implementation across a range of image sizes and report a speed up by a factor of 8 for a 4 megapixel input. We further investigate the impact of using packed vector type data on the performance, as well as the effect of double precision arithmetic on the GPU.**

I. Introduction

The use of integral images for rapid feature evaluation became popular with the seminal face detection algorithm proposed by Viola and Jones [1]. The features employed in the detector are reminiscent of Haar basis functions and form an overcomplete set for image representation. Obtaining the proposed features involves computing sums of pixel values over rectangular regions. Since these sums can be calculated by using only 4 array references with the integral image, evaluating this set of Haar-like features is very cheap, once the integral image is computed.

An alternative motivation for the integral image arises from the signal processing literature. In the "boxlets" work of Simard *et al.* [12], authors point out that in the case of linear operators (e.g. the inner product $f \cdot h$), any invertible linear operation can be applied to either f or h if the inverse operation is applied to the other operand. From this point of view, the integral image can be expressed as a dot product, $i \cdot r$, where i is the input image and r is the box car function that takes the value 1 inside the rectangle of interest and 0 outside. This summation can be written as

$$i \cdot r = \left(\iint i \right) \cdot r''$$

where the double integral of the image, obtained by summation first along the rows and then along the columns, is in fact the integral image and the second derivative of the boxcar function gives rise to four delta functions at the corners of the image. This is exactly the same idea as using 4 array references to compute the integral image.

This integral image formulation has allowed the Viola-Jones face detector to run in real-time, and influenced the development of several other computer vision algorithms. Among these, [2, 8] apply the integral image to histograms, thus extending its usage from Haar-like wavelets to more complex features such as the Histograms of Oriented Gradients [9] descriptors.

Even though the systems that incorporate the integral image approach as an intermediate component have been reported [2, 3, 7] to have training times in the order of days, they experience significant performance benefits. It is possible to build on this boost in speed by realizing such methods on general purpose GPUs, and obtain real-time performances [3].

A sequential implementation for integral image computation would require $2 \cdot w \cdot h$ operations for an image of size $w \times h$. As the size gets larger, this cost represents a significant overhead for the overall algorithm. Messom and Barczak [4] adopt a parallel processing approach to reduce this overhead. Their realization is based on the Brook stream processing language and demonstrates that employing the GPGPU paradigm results in significant performance benefits.

The term GPGPU refers to using graphics processing units to accelerate non-graphics problems. The many-core architecture of the new generation GPUs enables them to execute thousands of threads in parallel. These threads are managed with zero scheduling overhead and are lightweight compared to CPU threads. To fully utilize the great computational horsepower of the GPUs, thousands of threads need to be launched within each parallel routine. The potential benefit of employing a graphics card can be quantified by the theoretical floating point performance of the device. Whereas modern CPUs have peak performances on the order of 10 GFLOPs, commercial GPUs can exceed the 1 TFLOP limit. Since the integral image formulation is a compute-intensive, parallelizable task, we present a GPU implementation using the widely adopted NVIDIA CUDA programming model in this work.

In the rest of this paper, we provide the sequential integral

Final manuscript received May 15, 2010.

B. Bilgic is with the Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA 02139, USA (e-mail: berkin@mit.edu).

B.K.P. Horn is with the Department of Electrical Engineering and Computer Science and CSAIL, MIT, Cambridge, MA 02139, USA (e-mail: bkph@csail.mit.edu).

I. Masaki is with the Department of Electrical Engineering and Computer Science and MTL, MIT, Cambridge, MA 02139, USA (e-mail: IMasaki@aol.com).

image algorithm, give a brief background on the CUDA platform and detail our parallel algorithms. We conclude by commenting on the effect of using double precision and vector type data, and compare the performance with our sequential implementation.

II. The Sequential Integral Image Algorithm

For an image of size $w \times h$, we form the integral image on the CPU using the following algorithm,

Algorithm: Sequential integral image formulation

I : input image with size $w \times h$
I_{int} : integral image with size $w \times h$
Array elements are accessed in row major order.

for $x = 0$ to $w-1$ **do**
    $I_{int}[x] \leftarrow 0$
for $y = 1$ to $h-1$ **do**
    $I_{int}[y \cdot w] \leftarrow 0$
    $s \leftarrow 0$
    for $x = 0$ to $w-1$ **do**
        $s \leftarrow s + I[x + (y-1) \cdot w]$
        $I_{int}[x + y \cdot w + 1] \leftarrow s + I_{int}[x + (y-1) \cdot w + 1]$

We note that the output of this algorithm is an exclusive integral image, which is padded on the first row and the column by zeros and has the same size as the input. For instance, the image

$$I = \begin{bmatrix} 2 & 1 & 3 & 1 \\ 3 & 2 & 1 & 1 \\ 4 & 1 & 3 & 1 \end{bmatrix} \text{ produces } I_{\text{int}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 3 & 6 \\ 0 & 5 & 8 & 12 \end{bmatrix}$$

III. NVIDIA CUDA Programming Model

CUDA is the computing platform in NVIDIA graphics processing units that enables the developers to code parallel algorithms thorough industry standard languages. The CUDA programming model acts as a software platform for massively parallel high-performance computing by providing a direct, general-purpose C language interface 'C for CUDA' to the programmable multiprocessors on the GPUs. According to this model, parallel portions of an application are executed as *kernels*. CUDA allows these kernels to be executed multiple times by multiple *threads* simultaneously. A typical application would use thousands of threads to achieve efficiency.

At the core of the model lie three abstractions – a hierarchical ordering of thread groups, on-chip shared memories, and a barrier instruction to synchronize the threads active on a GPU multiprocessor. In order to scale to future generation graphics processors, multiple threads are grouped in *thread blocks* and multiple blocks reside in a *grid* that has user specified dimensions. Thread blocks may contain up to 512 threads, and the threads inside a block can communicate via low latency, on-chip *shared memory*. To prevent read-after-write, write-after-read, and write-after-write hazards, `__syncthreads()` command can be used to coordinate communication between the threads of the same block. A group of 32 threads that are executed physically simultaneously on a multiprocessor is called a *warp*.

There are six different memory types in the CUDA model that provide flexibility to the programmer. Apart from the shared memory (16kB) that is visible to all threads within a block, each thread has access to a private local memory and registers. Additionally, there are three types of off-chip memory that all threads may reach. The *global memory* (1792MB) has high latency and is not cached. The *constant memory* (64kB) is cached and particularly useful if all threads are accessing the same address. The *texture memory* is also cached and optimized for spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance. Textures can be bound to either linear memory or CUDA arrays; hence their maximum sizes depend on the particular data structure they are used with. Textures also provide hardware interpolation, which has very small performance cost.

The fact that shared memory resides on the multiprocessors where computations are performed whereas the global memory types are off-chip is reflected by the vast difference in their access speeds. It takes about 400 to 600 clock cycles to issue a memory instruction for the global memory, but the same operation occurs about 150 times faster in the shared memory. Therefore, if the same addresses need to be accessed multiple times, it would be beneficial to reach them via the shared memory.

IV. Parallel Algorithms for Integral Image Computation

We start by explaining the parallel prefix sum (scan) algorithm [5] which constitutes the foundation of our method. Next, we relate how this algorithm can be used as a building block by applying it first on the rows of the image, then taking the transpose, and again applying parallel scan on the rows of the transposed array to obtain the integral image.

A. Parallel prefix sum (scan)

The *all-prefix-sums* operation takes a binary associative operator $\oplus$, and an array of n elements

$$[a_0, a_1, \ldots, a_{n-1}]$$

and returns

$$[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$$

If we let the operator $\oplus$ be summation, we obtain the *inclusive scan* operation. If we shift the resulting array to the right by one element and insert the identity in the beginning, we end up with the *exclusive scan* operation, which returns

$$[0, a_0, (a_0 + a_1), \ldots, (a_0 + a_1 + \ldots + a_{n-2})]$$

In the rest of this paper, we will be focusing on the exclusive version of the operation, and simply refer to it as *scan*.

For an input array with size n, the scan algorithm has computational complexity of $O(n)$, and it consists of two phases: the *reduce phase* (or the *up-sweep phase*) and the *down-sweep phase*. We can visualize the reduce phase as building a binary tree (Figure 1), at each level reducing the number of nodes by half, and making one addition per node. Since the operations are performed in place using shared memory, the tree we build is not an actual data structure, but helps explaining the algorithm.

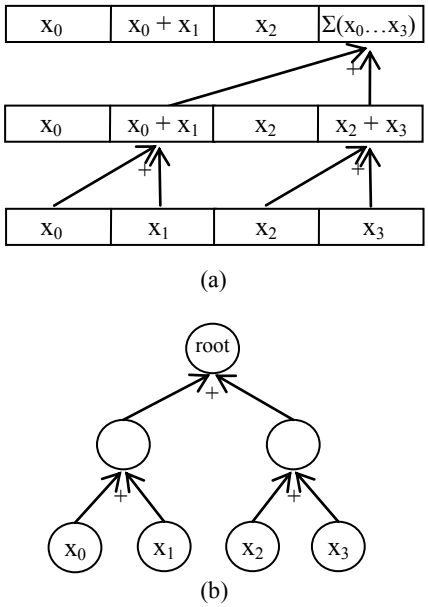

Figure 1: (a) The reduce phase applied on an array of four elements. (b) Binary tree view of the algorithm. Scanning is performed from the leaves to the root, where the root contains the sum of all four elements.

In the down-sweep phase, we traverse the tree from the root to the leaves, and use the partial sums we computed in the reduce phase to obtain the scanned array. We note that the last element is set to zero in the beginning and it propagates to reach the beginning of the array, thus resulting in an exclusive computation (Figure 2).

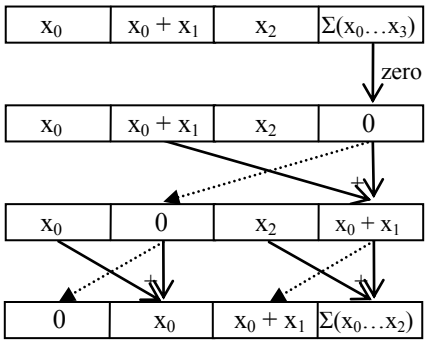

Figure 2: The down-sweep phase. At each level of the tree, there are as many swapping operations as summations.

The overall cost of these phases is $2(n-1)$ summations and $(n-1)$ swaps, which is in $O(n)$ time, same as the sequential algorithm. Following [5], we provide the CUDA kernel that implements the scan algorithm below:

CUDA Code: Scan kernel for the GPU

```
__global__ void scan(float *input, float
*output, int n)
{
  extern __shared__ float temp[];
  int tdx = threadIdx.x; int offset = 1;

  temp[2*tdx]   = input[2*tdx];
  temp[2*tdx+1] = input[2*tdx+1];

  for(int d = n>>1; d > 0; d >>= 1)
  {
    __syncthreads();
    if(tdx < d)
    {
      int ai = offset*(2*tdx+1)-1;
      int bi = offset*(2*tdx+2)-1;
      temp[bi] += temp[ai];
    }
    offset *= 2;
  }

  if(tdx == 0) temp[n - 1] = 0;

  for(int d = 1; d < n; d *= 2)
  {
    offset >>= 1; __syncthreads();
    if(tdx < d)
    {
      int ai = offset*(2*tdx+1)-1;
      int bi = offset*(2*tdx+2)-1;

      float t  = temp[ai];
      temp[ai] = temp[bi];
      temp[bi] += t;
    }
  }
  __syncthreads();
  output[2*tdx]   = temp[2*tdx];
  output[2*tdx+1] = temp[2*tdx+1];
}
```

Even though this kernel is work efficient, it suffers from bank conflicts in the shared memory. In our implementation, we try to avoid these conflicts by adding a variable amount of padding to each shared memory index we use, as suggested in [5]. The amount we add is equal to the value of the index divided by the number of memory banks, which is equal to 16 for our graphics card.

As it is, this kernel is unable to scan arrays with sizes larger than 1024, since the maximum number of threads per block is 512 and a single thread loads and processes two data elements. Influenced by [10], we solve this problem by employing several thread blocks and making them responsible for a certain part of the input. If we let the input array contain n elements and if each block processes b of the entries, we need to launch n/b thread blocks and $b/2$ threads in each block. With the usual scan algorithm, each thread block scans its part of the array, but before zeroing the last

element that contains the sum of all the elements in that segment, we register it to an auxiliary array I_{sum}. We then scan this array in place and add $I_{sum}[i]$ to all elements of the segment that $(i+1)^{st}$ thread block is responsible for. Figure 3 tries to further illustrate this. To handle inputs with a size that is not a power of two, we pad the last segment of the array before scanning.

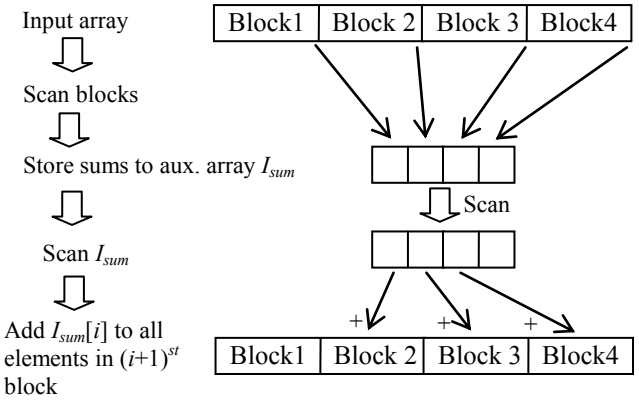

Figure 3: Scanning arrays of arbitrary size.

B. Scanning the image rows

We treat each row of the image as an independent array and scan the rows in parallel. In our implementation, each row is divided into segments of 512 pixels, and each segment is processed by a thread block consisting of 256 threads. Hence, we launch a scan kernel using a grid with dimensions $n_{seg} \times h$, where n_{seg} is the number of segments in each row, and h is the height of the image.

C. Computing the Transpose

After scanning the rows of the image, we take the transpose of the resultant array, so that we can use the same scanning kernel twice in order to compute the integral image. Taking the transpose is the cheapest routine in our method, because we utilize the shared memory to provide coalescence, and apply padding to the shared memory in order to avoid bank conflicts, as suggested in [11]. We present the transpose kernel next, where we take `BLOCK_DIM` as 16.

After transposing, we scan the rows of the transposed array to obtain the integral image. We launch a scan kernel with grid dimensions $\tilde{n}_{seg} \times w$, where $\tilde{n}_{seg}$ is the number of thread blocks, and w is the width of the image. We note that the resulting integral image is in transposed form, but this poses no difficulties since the pixel at position (x, y) can be accessed by the index $(y+x \cdot h)$.

CUDA Code: Transpose kernel for the GPU

```
__global__ void transpose(float *input, float
*output, int width, int height)
{

 __shared__ float temp[BLOCK_DIM][BLOCK_DIM+1];

 int xIndex = blockIdx.x*BLOCK_DIM + threadIdx.x;
 int yIndex = blockIdx.y*BLOCK_DIM + threadIdx.y;

 if((xIndex < width) && (yIndex < height))
 {
   int id_in = yIndex * width + xIndex;
   temp[threadIdx.y][threadIdx.x] = input[id_in];
 }

 __syncthreads();

 xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
 yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;

 if((xIndex < height) && (yIndex < width))
 {
  int id_out = yIndex * height + xIndex;
  output[id_out] = temp[threadIdx.x][threadIdx.y];
 }

}
```

V. Experiments

A. Single Precision Floating Point Computation

A multiprocessor consists of eight single precision thread processors, two special function units, on-chip shared memory, an instruction unit, and a single double precision unit. Therefore, GPUs are optimized for single precision computations, and there is an order of magnitude difference in the theoretical performance bandwidth between single and double precision operations. Figure 4 compares the results obtained with the sequential algorithm running on the CPU and the single precision GPU implementation. In all of our results, we exclude the time spent for data transfer and report only the GPU computation times, which are obtained on an NVIDIA GeForce GTX 295 graphics card. We use a PC

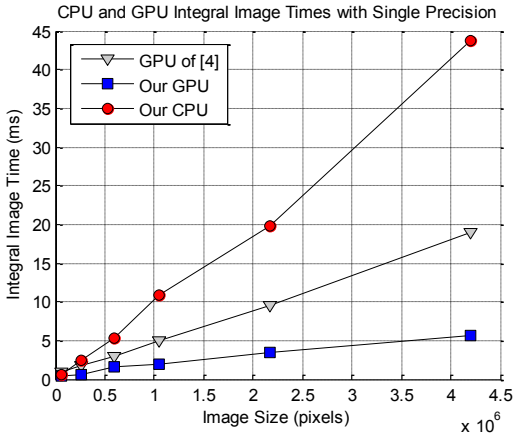

Figure 4: Performance comparison of single precision integral image computation on the CPU and the GPU. Results for [4] are replicated from their work

with 2.5 GHz CPU and 3GB memory. For a 4 megapixel input, our system works about 3 times faster than the proposed method in [4], which is implemented with the Brook language and runs on a ATI graphics card.

B. Single Precision Vector Processing

In addition to standard data types, CUDA also provides packed data structures to ease access to multi-dimensional inputs. The vector type formed by a bundle of four floating point numbers is called `float4`. Since the size of this structure is 16 bytes, it satisfies two important properties that increase the maximum memory bandwidth. First, the GPU is capable of reading 16-byte words from global memory into registers in a single instruction. Second, global memory bandwidth is used most efficiently when the memory accesses of the threads in a half-warp can be coalesced into a single memory transaction of 32, 64, or 128 bytes. In the case of `float4` data, this results in only two 128 byte transactions per half-warp, given that the threads access the words in sequence. Therefore, it is possible to process four times more data with a smaller impact on the memory bandwidth. This point is illustrated in Figure 5.

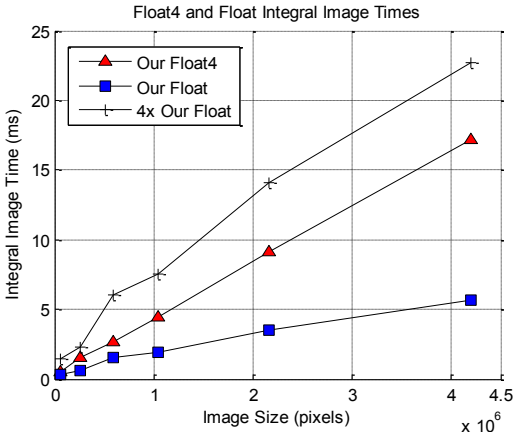

Figure 5: Comparing the GPU processing times of our `float4` implementation with four times the processing time of our `float` integral image. We are able to process four times more data using `float4` vector type, with a smaller impact on the memory bandwidth.

C. Double Precision Floating Point Computation

As GPUs are optimized for single precision arithmetic, double precision implementation results in a lower performance as depicted in Figure 6. For large image sizes, this performance degradation may be traded-off for higher accuracy computation. We note that our results are about 4 times faster than the implementation by [4] for a 2048×2048 size image.

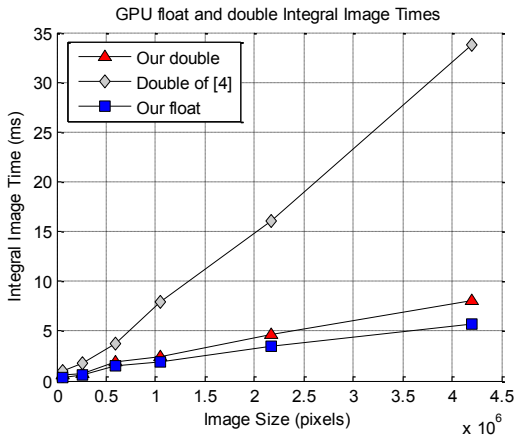

Figure 6: Performance comparison of double and single precision GPU implementations. Results for [4] are replicated from their work.

We finalize our discussion by noting that even though using double precision arithmetic reduces the GPU performance, it is still 9 times faster than the double precision CPU implementation, for a 4 megapixel input (Figure 7). As the input size gets smaller, we see that the performance difference is reduced. This is mainly because the CPU implementation makes use of its large cache and it is not possible to utilize all GPU processors at small image sizes.

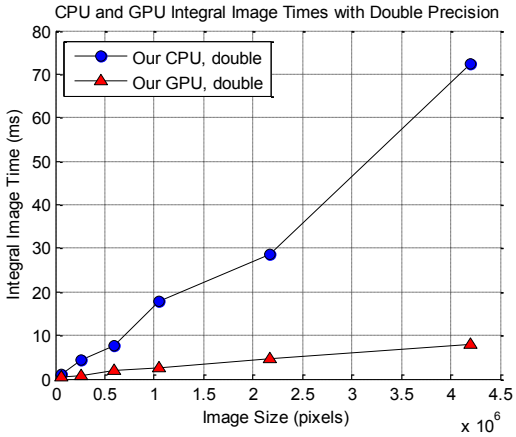

Figure 7: Integral image computation times with double precision on the CPU and the GPU. We report a speed up by a factor of 9 with the GPU implementation.

D. Kernel Occupancy and Performance

Maximum *occupancy* per kernel is a way of measuring CUDA code performance by quantifying how efficiently a multiprocessor is being used. Occupancy is defined as the ratio of the active warps to the maximum number of warps supported on a multiprocessor and determined by the shared memory and register usage and the thread block configuration of a kernel.

Kernel	Occupancy	Mem. Throughput	Shared Mem.	Registers	Threads/Block
Scan array	100 %	17 GB/s	2224	11	256
Increment block	100 %	50 GB/s	48	5	256
Transpose	100 %	49 GB/s	1120	8	16×16

Table 1: Shared memory and register usage, as well as the thread block configuration affects the kernel occupancies. The kernels in our implementation work at full occupancy, which is an indicator of good performance. The overall memory throughput reflects how fast the kernels access data from the global memory.

Table 1 presents occupancies as well as the processing times related with each kernel in our method. We note that all the kernels involved in integral image computation works with full occupancy.

VI. Conclusions

In this work, we have presented data parallel algorithms for integral image computation. Given that many computer vision algorithms employ this data structure for rapid feature evaluation, our approach can be used as a subroutine to increase the performance of such systems. This technique can be easily extended to compute integral histograms, as they depend on the same principles.

For high resolution input images, our method provides close to an order of magnitude speed up relative to its sequential counterpart, for both single precision and double precision arithmetic. By employing vector type data structures, it is possible to push this performance boost even further. Thanks to the flexible CUDA programming model, our method extends to future generation GPUs, as well as scaling to multi-GPU systems.

References

[1] P. Viola and M. Jones. Rapid Object Detection Using a Boosted Cascade of Simple Features. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2001

[2] Q. Zhu, S. Avidan, M. Yeh, and K. Cheng. Fast Human Detection using a Cascade of Histograms of Oriented Gradients. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2006

[3] B. Bilgic, B.K.P. Horn, I. Masaki. Fast Human Detection with Cascaded Ensembles on the GPU. *Submitted to IEEE Intelligent Vehicles Symposium, 2010*

[4] C.H. Messom, A.L. Barczak. High Precision GPU based Integral Images for Moment Invariant Image Processing Systems. *Electronics New Zealand Conference (ENZCON'08), 2008*

[5] M. Harris. Parallel Prefix Sum (Scan) with CUDA. *NVIDIA CUDA SDK code samples*

[6] G.E. Blelloch. Prefix Sums and Their Applications. John H. Reif (Ed.), *Synthesis of Parallel Algorithms*, Morgan Kaufmann, 1990.

[7] P. Viola, M. Jones and D. Snow. Detecting Pedestrians Using Patterns of Motion and Appearance. *International Conference on Computer Vision (ICCV)*, 2003

[8] F. Porikli. Integral histogram: A Fast Way to Extract Histograms in Cartesian Spaces. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005

[9] N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005

[10] M. Harris. Scan of Large Arrays. *NVIDIA CUDA SDK code samples*

[11] NVIDIA: NVIDIA CUDA SDK code samples, Transpose

[12] P.Y. Simard, L. Bottou, P. Haffner, and Y.L. Cun. Boxlets: a Fast Convolution Algorithm for Signal Processing and Neural Networks. In M. Kearns, S. Solla, and D. Cohn, editors, *Advances in Neural Information Processing Systems*, volume 11, pages 571–577, 1999