A brief introduction to the code for computing sum and average

I have used the concept of integral images /summed-area table [3] to calculate the sum and average in an efficient manner. The integral image is calculated by two for loops and the cumulative sum (along row and column) for each pixel is stored as follows:

II[x,y] = I[x,y] + II[x-1,y] + II[x,y-1] - II[x-1,y-1] (1)
Where I = image and II = integral image

This integral image is calculated in the constructor. Once the integral image is computed, the sum in a window [x1,y1,x2,y2] is computed as follows:

Sum of window = II[x2,y2] - II[x2,y1] - II[x1,y2] + II[x1,y1] (2)
Where II = integral image

The average is computed by dividing the sum by the window area. In case of non-zero average, I compute integral image of non-zero count similar to equations (1) and (2) but by replacing the pixel values by 1 in case the pixel value is non-zero.

Essay Questions

a) Typical buffers in our use cases are extremely sparse matrices, i.e., the vast majority of the pixels are zero. How would you exploit this to optimize the implementation?

In a sparse matrix majority of elements are zero and very few are non-zero elements.

In this case we can construct sparse prefix sums. Sparse prefix sums technique exploits the sparsity in the data and avoids calculating and storing prefix sum for empty rows and columns. Instead of a prefix data grid, look-up table can be used. It takes O(1) computational time complexity to get the sum or average and much lesser storage cost.

Construction of a sparse prefix sum and the data structure used for the internal representation is described below. [1]

1) Split the Matrices in a grid A with N cells into blocks. Say for example a block looks like this.

0	0	6
0	0	0
0	2	12

2) Remove block slices where entire row or column are zero and reduce it as shown below.

0	6
2	12

3) Now Compute prefix sum to this above reduce matrix as shown below. You can represent this matrix as M.

0	6
2	20

4) Now we should have a lookup table in order to have a constant lookup time, given a query range in original buffer space. Each block should have one look-up table per dimension i that maps a coordinate Ci along dimension i to $L^i[c_i]$

   L 1 (Horizontal Dimension)

-1	0	1

   L 2 (Vertical Dimension)

0	0	1

5) The value -1 represents one omitted slice in that dimension.
6) The lookup tables L are used to translate coordinates from the block coordinate space into coordinate space of M. To access the prefix sum of the cell (1,2) in the original space, we have to access the cell (L1[1], L2[2]) = (0,1) in the reduced space M. Which is 2. This way any Prefix sum from the original space can be retrieved in O(1) complexity.

b) What changes to the code and the API need to be made if buffer dimensions can be >= 4096 x 4096?

1) Will need a larger memory size. This will also impact the RAM size.
2) The resolution of the image is increased with more pixels if the buffer dimensions are increased.
3) The code changes for dimension changes is to simply initialize the constructor with the image width and height.

Also, what would happen if instead of 8 bits per pixel we have 16 bits per pixel?

1) If the per pixel value is changed to 16 bits, which means the dynamic range is increased and so the pixel value an range from 0 to 65535. Which will double the storage memory space.
2) The code changes for dynamic range change is that Initializing image buffer data type as unsigned int instead of unsigned char. To make it generic we should use templates.

c) What would change if we needed to find the maximum value inside the search window (instead of the sum or average)? You don't need to code this, but it would be interesting to know how you would approach this problem in general, and what is your initial estimate of its O-complexity.

One way to get the maximum value in the search window would be linear search through all the pixel in the window and finding the maximum value. The time complexity of it would be O(N). Where N is the number of pixels in the window.

We can use sparse table with min-max range query. It takes O(N log N) storage space and returns results in O(1) time complexity. We will need to construct a lookup table similar to the one mentioned in first answer.

We can use segment trees to find the maximum of a window.
Constructor

- For this, in the constructor, we need to construct a segment tree by recursively splitting the image into quarters till we end up with one pixel in the leaf nodes and we assign each nodes range segment as we split.
- We assign each internal node as the maximum of each child during backtracking.
- The time complexity of constructing this tree takes O(n)

Maximum of window

- When the maximum of the window has to be computed, we call recursive function which compares the range of the window with the range of each child segment and finds the maximum of child segments.
- If there is some overlap (not 100%) between child segment and the search window, we continue calling this function recursively for its children.
- When we encounter that the range of child node is completely within the search window then we return its minimum value.
- If the node is outside the range then we return 0
- The time complexity for this function would be O(log n).

Another way would be to use Binary search tree on the search window. I will create a heuristic function which computes Average of non-zero pixels + (3*Standard deviation). Calculating the Standard deviation can be done in a constant time [2]. Average + (3*Standard deviation) value represents a rough value of the maximum pixel.

1) First split the search window into 2 and compute the heuristic value.
2) Choose the side with maximum heuristic value as the new search window and
3) repeat it until the search size reach to 1 pixel which will be the maximum pixel.

The time complexity of such an implementation would be O(log n) with a worst case of O(n).

d) How would multithreading change your implementation? What kinds of performance benefits would you expect?

In case of multi-threading the number of threads depend on the number of cores the processor has. Say it is a 4 core processor, the implementation can be written with 4 threads.

1) We can first divide the matrices row wise and then assign block rows to each thread to compute the cumulative sum row wise.

2) All threads must finish their computation before proceeding to the next step. We should call join function on the threads.
3) Then repeat the same process again splitting the matrices column for each thread and computing the cumulative sum.

This way the computational performance and computational time can be improved by approximately 4 times.

Regarding Implementation : There is a rough implementation of this in the function “computeIntegralImages3” and “computeIntegralImagesThread”. Instead of single thread the function should assign task to multiple threads.

e) Outline shortly how the class would be implemented on a GPU (choose any relevant technology, e.g. CUDA, OpenCL). No need to code anything, just a brief overview is sufficient.

A rough outline of the implementation is shown in the file “**PixelSumGPU.cpp**”. Below are a brief overview of the code.

1) Declare the CUDA variables
2) Allocate Memory and initialize the memory
3) Setup the launch configuration. This can be altered and decided based on the GPU. The number of threads per block can be configured upto 1024. But each GPU has a limit on the number of threads per block. It can be 128, 256 or 512,etc. 2^{32}-1 blocks can be launched in a single launch. There is no limit on the number of blocks that can be launched. Each GPU can run some number of blocks concurrently, executing some number of threads simultaneously.
4) With this Launch configuration setup call the Kernel which is the CUDA implementation
5) The function which is to be compiled and executed by GPU should be prefixed with “__global__”.
6) Each of the running threads is individual, they know the following:
    a) threadIdx - Thread index within the block
    b) blockIdx - Block index within the grid
    c) blockDim - Block dimensions in the threads
    d) gridDim - grid dimensions in the blocks
7) These are dim3 structures and can be read in the kernel to assign particular workloads to any thread. This way the task can be made parallel.
8) First, each thread handles (Row Size / no. of threads) rows and perform their sum. After this all the threads will be synchronized. Again the kernel can be re-initialized and launched to calculate the sum of the columns, where each thread will handle (Column size / no. of threads) columns.
9) Free all allocated cuda memory.

Reference

[1] Shekelyan, Michael, Anton Dignös, and Johann Gamper. "Sparse prefix sums: Constant-time range sum queries over sparse multidimensional data cubes." *Information Systems*(2018).

[2] https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html#integral

[3] https://en.wikipedia.org/wiki/Summed-area_table