HPC PROJECT

IMAGE CONVOLUTION

Kaushal Patel - 201501219
Naitik Dodia   - 201501177

Problem Statement: 2D Convolution of a given image with a predefined matrix to get effects such as blur, sharpen, emboss and sobel.

Serial Algorithm

For each pixel of the image we get the value of the convolution and save it in a new image. So at each pixel we put the center of the kernel on that pixel and multiply each neighbouring pixel with its corresponding value in the kernel which falls over it. Then we have to add all the values that we have got and then at last divide that by the sum of the values of the kernel. Dividing is necessary because if we multiply by the values whose total is not 1, will mean that we will be changing the actual data in the image. For the boundary pixels we are using the wrap method i.e. for outside pixels we use the values of the pixels of the opposite side of the image.

Parallel Algorithm

This code can be implemented in parallel in three ways

1)Row wise parallel

For p processors all the rows can be divided by p processors and can be computed parallelly.

2) Column wise parallel
For each row total columns can be split into p processors and can be computed parallely.
But for image processing column wise splitting the load for each row is not advisable because the parallel overhead for creating and joining p threads for each row is so expensive that it’s time overpasses the time saved during the parallel computation.

3) **Block wise parallel:**
In this type we divide the workload of the image into blocks of image. This type of parallelism is helpful to use the temporal locality of the pixels. Because the neighborhood rows are already accessed for the calculation. So they are already present in the cache and their presence can be used and memory access time can be decreased.

Complexity of code: Let n be total number of pixels in the given image and m be the size of the kernel. For each pixel we calculate m multiplications so time complexity of the code is O(mn) and calculating the sum of the kernel O(m).
Total time complexity: O(mn)
Parallel Time: There is no dependency in the computation part so all the computation part can be divided into available processors.Parallel time complexity: O((n^2 * m^2)/p)

Parallel overhead:

Serial time =0.242921
Parallel time=0.255599

Parallel overhead=1.052

Sharpen $\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$ **Blur** $\begin{pmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$

Emboss $\begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & -1 & 2 \end{pmatrix}$ **Sobel** $\begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & 2 \\ 1 & 0 & -1 \end{pmatrix}$

Theoretical Speedup:

Using Amdahl's Law

p = 12 , Image size = 2048x2048, Kernel size = 3x3

$s = 18 / (18 + 2048 * 2048 * 3 * 3 * 3)$

$s = 1.589 * 10^{-7}$

$Speed\ up \leq \frac{1}{s + \frac{1-s}{p}}$

$Speedup \leq 11.99997$

	12	11	10	9	8	7	6	5	4	3	2	1
1024	11.9999	10.9999	9.99994	8.99995	7.99996	6.99997	5.99998	4.99998	3.99999	2.99999	1.99999	1
1080	11.9998	10.9999	9.99994	8.99995	7.99996	6.99997	5.99998	4.99998	3.99999	2.99999	1.99999	1
1200	11.9997	10.9999	9.99995	8.99996	7.99997	6.99998	5.99998	4.99999	3.99999	2.99999	1.99999	1
1252	11.9997	10.9999	9.99996	8.99996	7.99997	6.99998	5.99998	4.99999	3.99999	2.99999	1.99999	1
1920	11.9996	10.9999	9.99998	8.99998	7.99998	6.99999	5.99999	4.99999	3.99999	2.99999	1.99999	1
2048	11.9996	10.9999	9.99998	8.99998	7.99999	6.99999	5.99999	4.99999	3.99999	2.99999	1.99999	1

Using Gustafson- Barsis Law

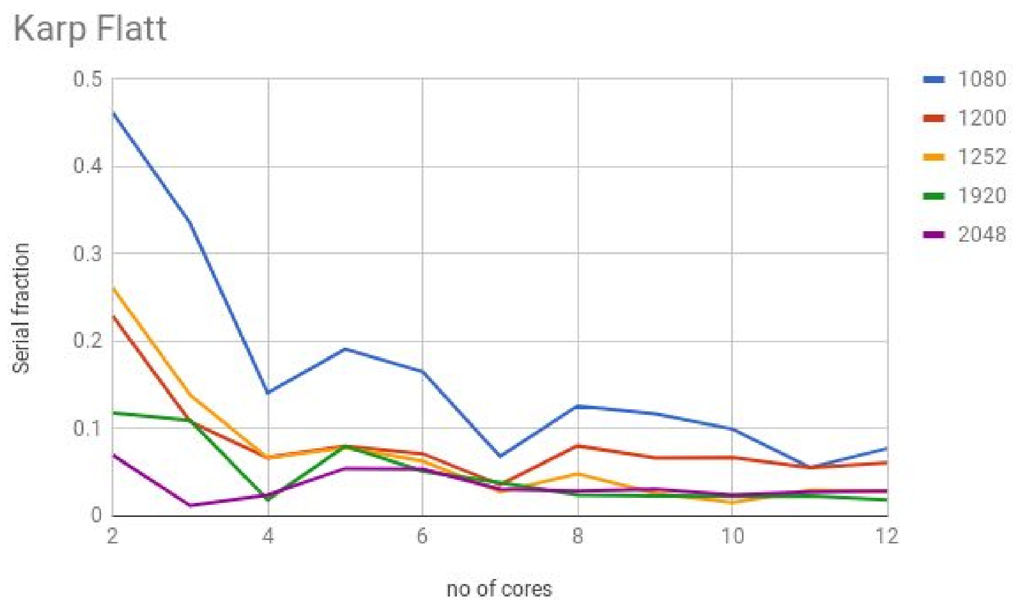
$s = 1.589 * 10^{-7}$

$Speedup \leq p + s(1 - p)$

$Speedup \leq 11.99999$

SIZE	12	11	10	9	8	7	6	5	4	3	2	1
1024	11.9999	10.9999	9.99999	8.99999	7.99999	6.99999	5.99999	4.99999	3.99999	2.99999	1.99999	1
1080	11.9999	10.9999	9.99998	8.99999	7.99999	6.99999	5.99999	4.99999	3.99999	2.99999	1.99999	1
1200	11.9999	10.9999	9.99998	8.99998	7.99998	6.99998	5.99999	4.99999	3.99999	2.99999	1.99999	1
1252	11.9999	10.9999	9.99998	8.99998	7.99998	6.99998	5.99998	4.99999	3.99999	2.99999	1.99999	1
1920	11.9999	10.9999	9.99997	8.99998	7.99998	6.99998	5.99998	4.99999	3.99999	2.99999	1.99999	1
2048	11.9999	10.9999	9.99997	8.99998	7.99998	6.99998	5.99998	4.99999	3.99999	2.99999	1.99999	1

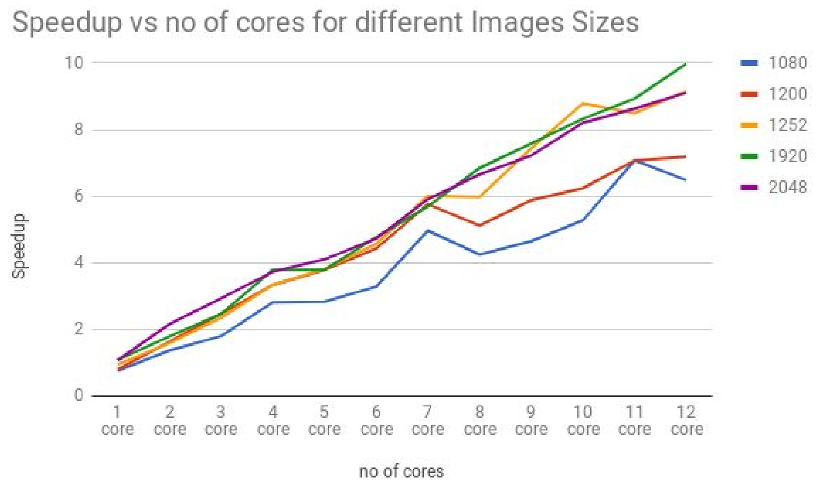
Karp Flatt:



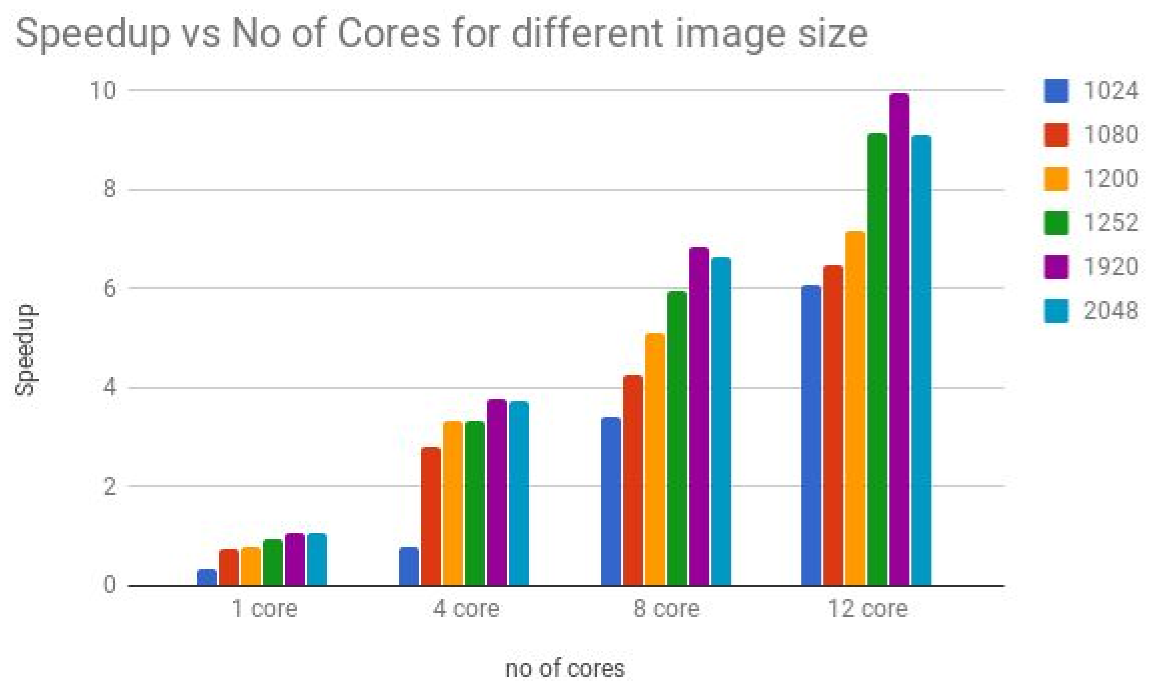
- As the problem size increases, serial fraction decreases. Because the serial fraction doesn't consist of any loop that is dependent on problem size. So Serial computation is constant while parallel computation increases which implies serial fraction decreases.
- As the number of cores increases serial fraction decreases which implies more cores we use we get more and more speedup.

Graphical Analysis:

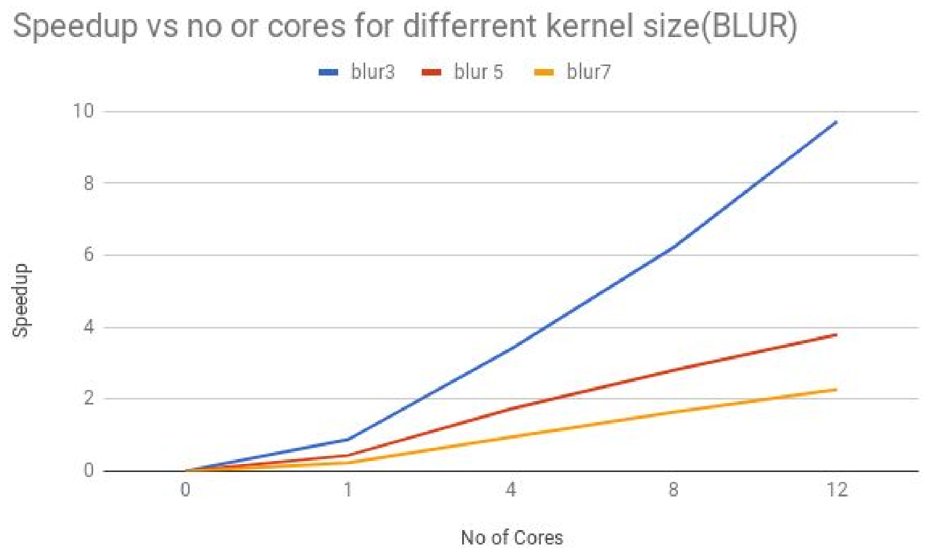
1) Row wise



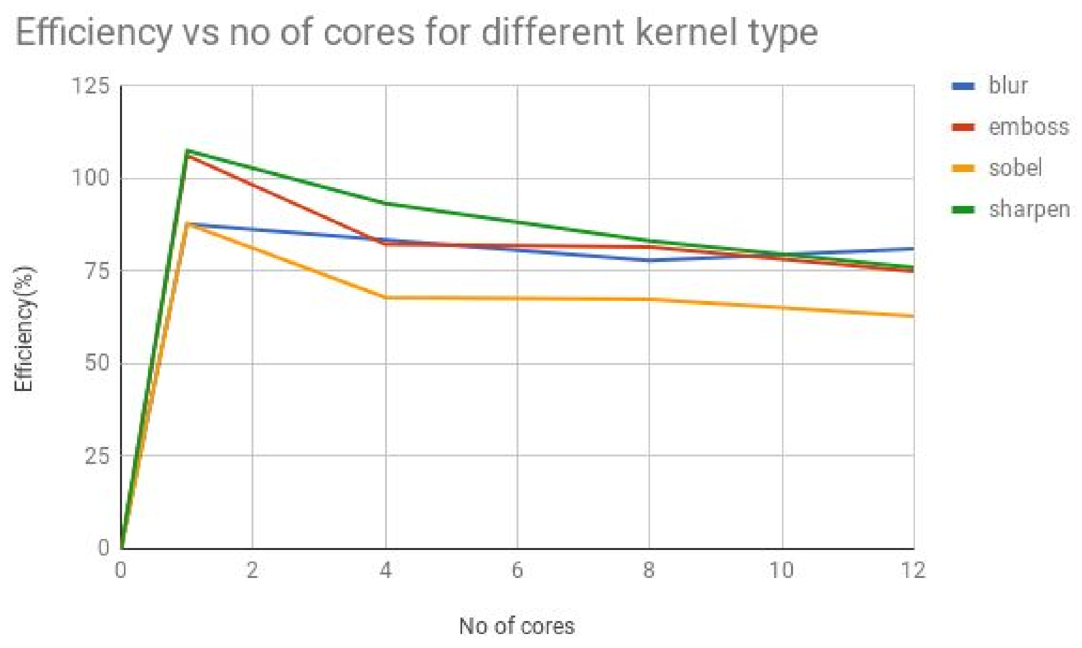
- As the number of cores increase the speedup increases



- Here we can see that we are getting more speed up for the image 1920x1920 rather than 2048x2048 because of the high increase in cache miss rate. Otherwise as problem size increases speedup increases and also as number of cores increase,

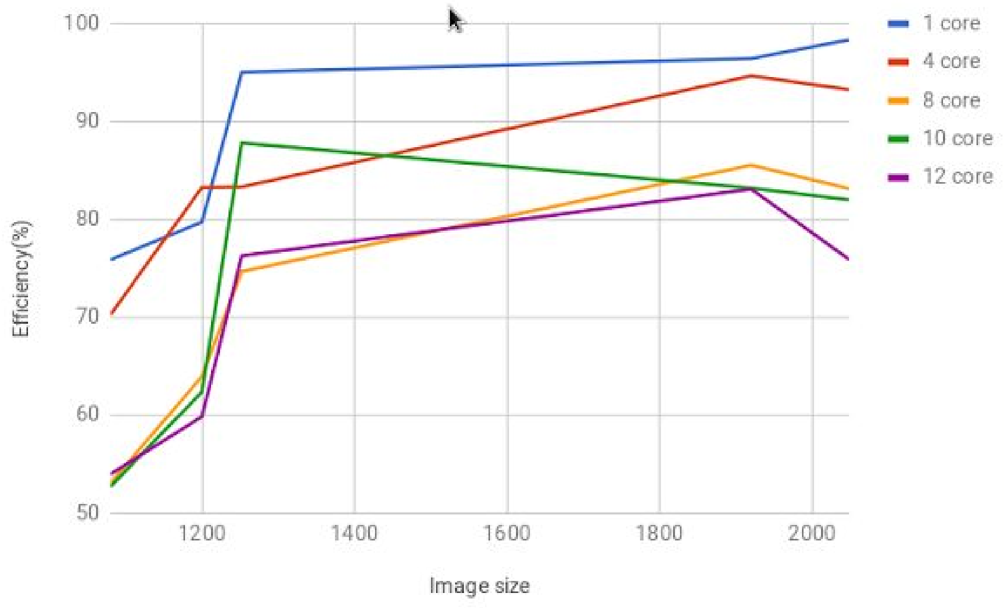


- As number of cores increases speed up increases but as the kernel size increases the speed up decreases due to increased serial fraction of code.

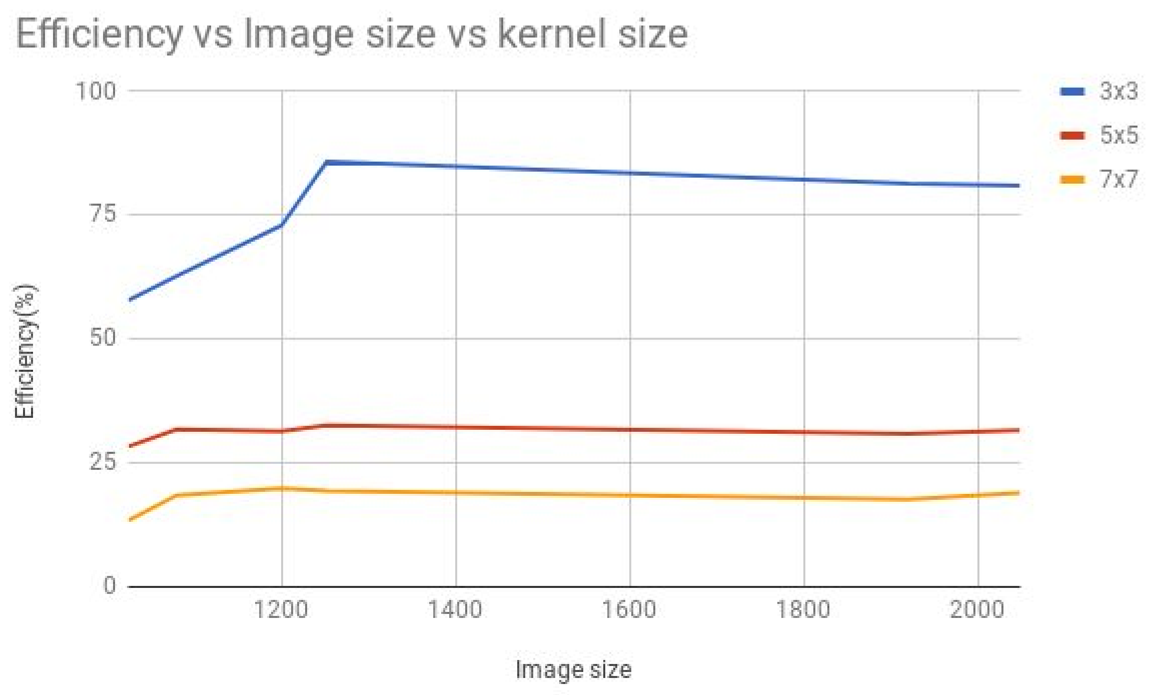


- Efficiency remains almost constant after 4 processors. From 1 to 4 efficiency decreases and for sharpen filter it slightly decreases.

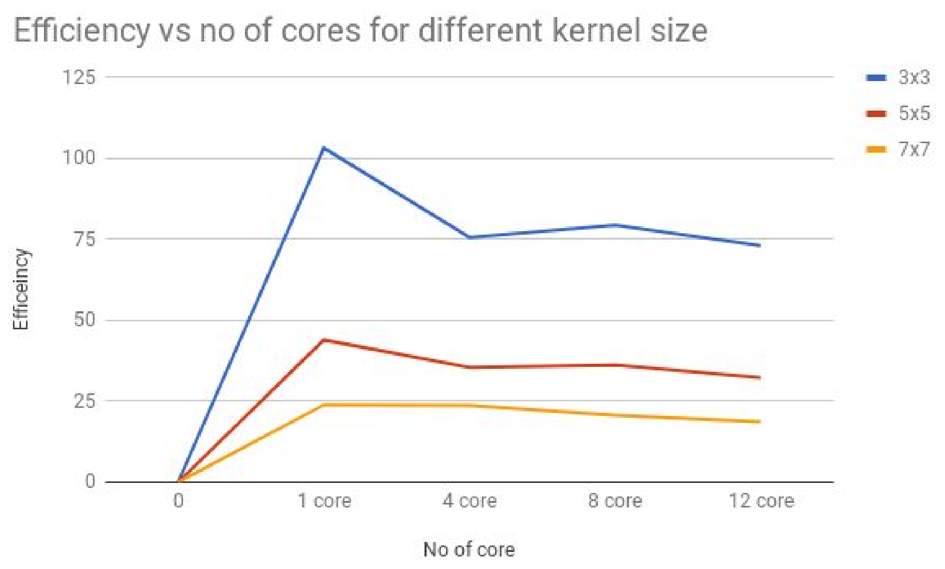
Efficiency vs Image size



- As the problem size increases efficiency decreases and it remains almost constant with increase in problem size at larger sizes.

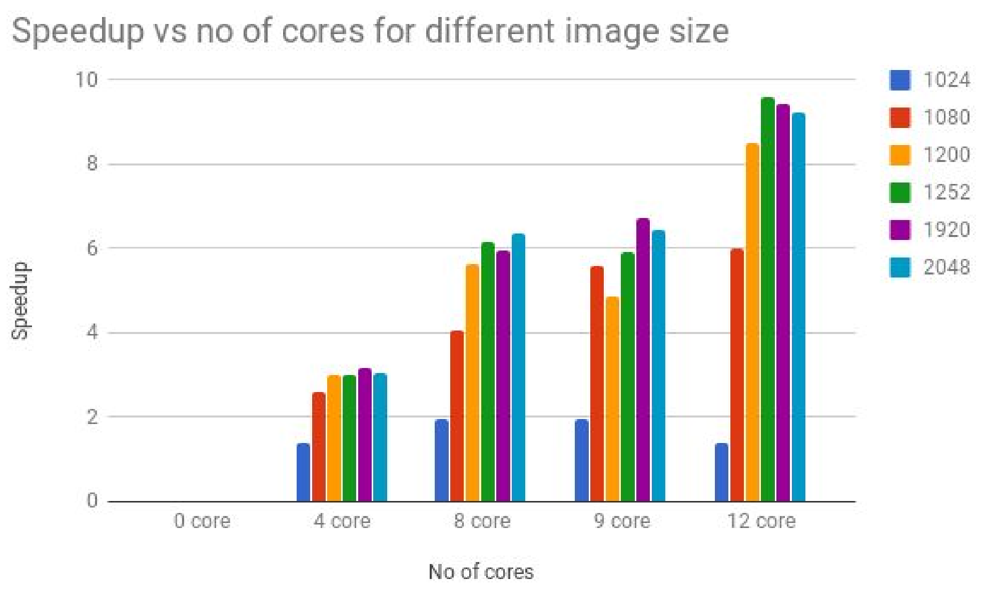


- As the problem size increases the efficiency remains constant and the kernel size increases the efficiency decreases. This is because decrease in the speedup. As the kernel size increases the serial fraction in the parallel code increases and hence efficiency decreases.

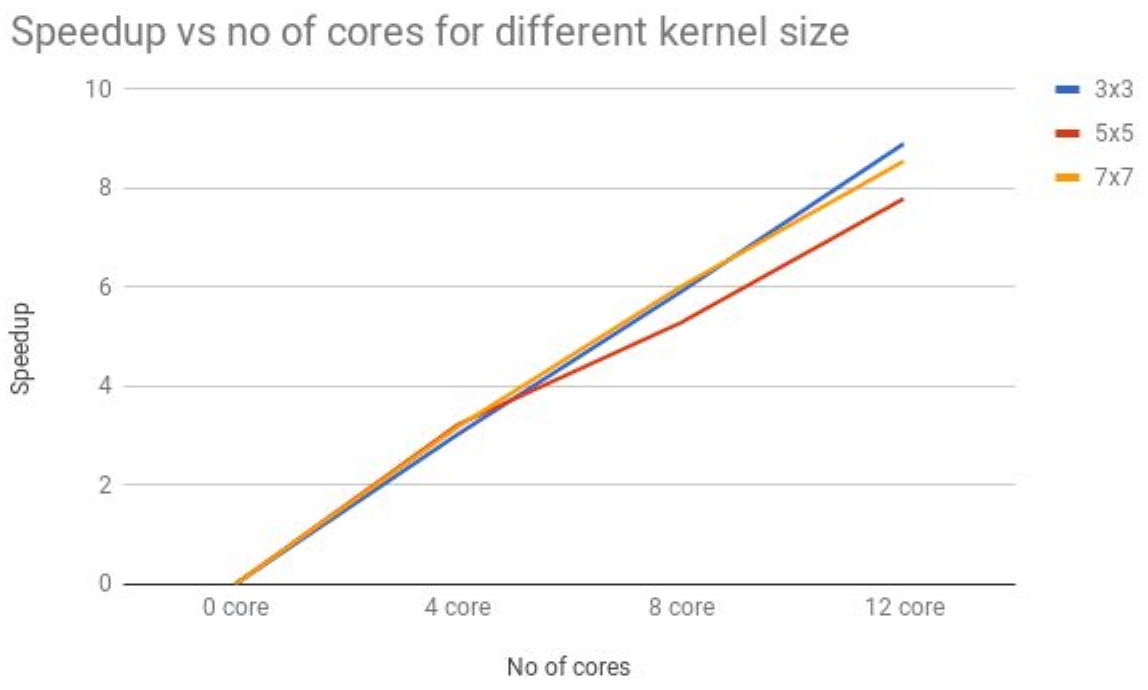


- For 1 core efficiency is more because of less parallel overhead. But as the number of cores increases the efficiency remains almost constant which implies that speedup increases with increase in number of cores. As the kernel size increases efficiency decreases.

Block wise:

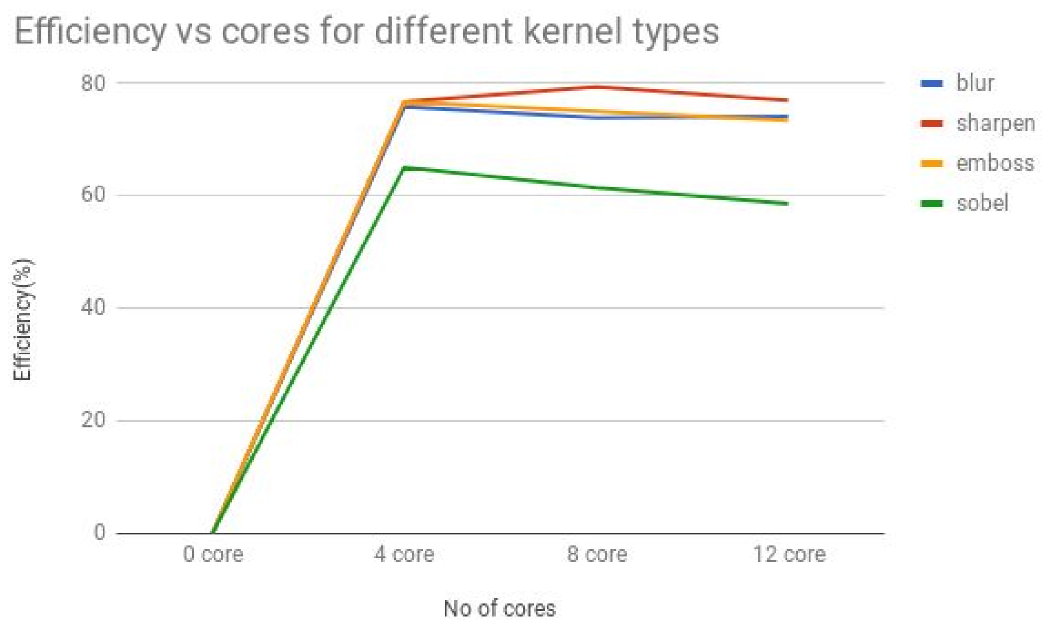


- As the number of processors increases speedup increases but there are some anomalies because of too many users on the same node.

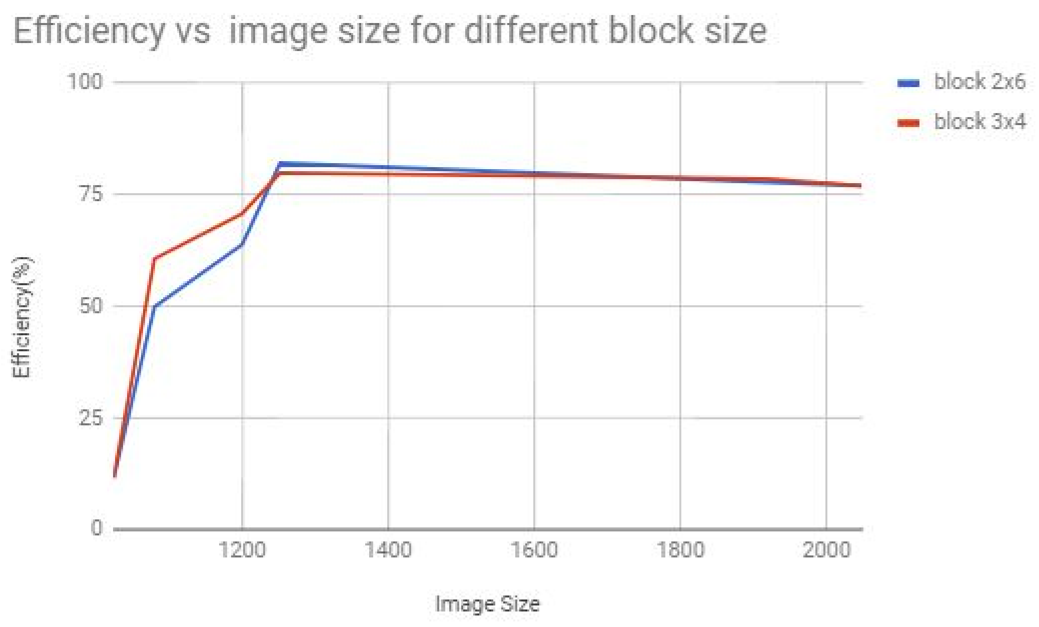


- Speedup of 7x7 kernel is more than 5x5 because of the more match in size of the block and the kernel. And for 7x7 more temporal locality is used more than that of 5x5.\

Efficiency vs no of cores for different kernel type

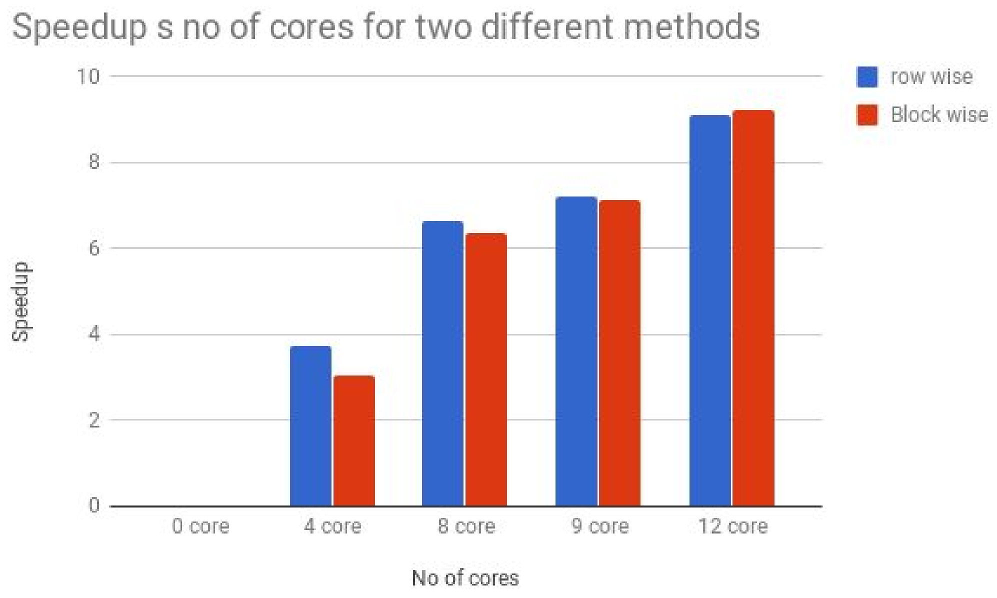


Efficiency of sharpen filter is more due less computation as the corners of the block kernel are 0s.



- As we change the alignment of the blocks we can see the change in speedup values. For 12 cores as 3x4 is more close to √p x√p. It achieves stability faster than 2x6 alignment. After 1400x1400 image size speedup by both the alignments are same because the block size is relatively square and large for the kernel size.

Speedup vs no of cores for two different methodsmethods



- Upto 9 cores the speedup by row-wise method is more but after that speed up 12 cores increases. Also we can see that block becomes more efficient as the number of cores increases.

Cache Coherence:
Here as each pixel's value is updated in the new image its copy will not be saved in any other processor's local cache. And also the updation occurs in a new image and the original image is only accessed. So there is no problem of incoherence.

Granularity:
Ratio of computation to communication: There is no communication between the processors because all the pixel values are computed independently of other processors results. So granularity is Coarse.

Load Balance:
For each pixel we access total 9 pixels of the input image and 1 of the output image. And computations = 9(multiplication) + 9(sum) + 1(division).
Load balance = 0.526