HPA LAB 07

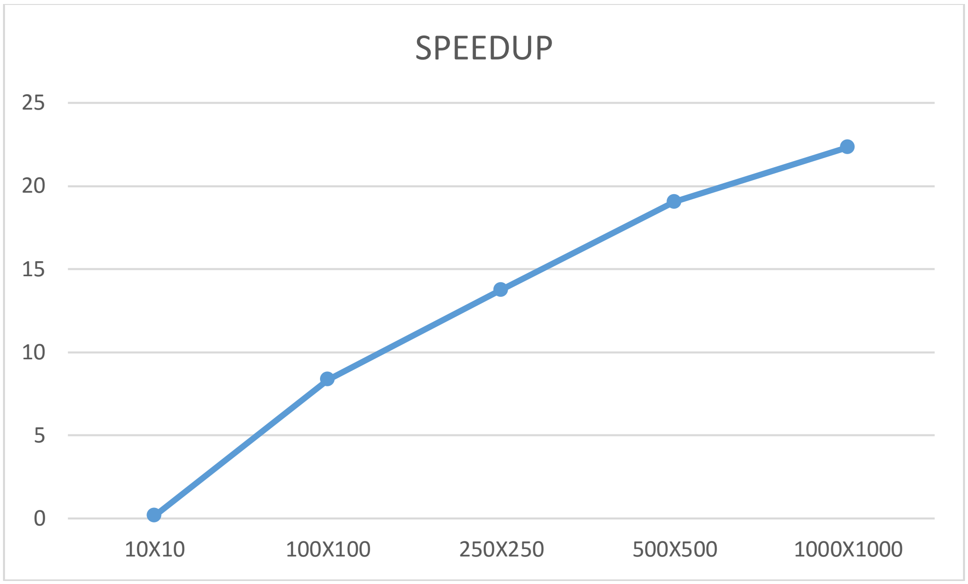
Abstract: In this lab, we had to use GPU to compute scientific problem and take advantage of the parallelism present in the algorithm and run it on device. Heat Transfer simulation algorithm was a highly parallel algorithm and was implemented on the GPU. The algorithm was implemented by developing a GPU native code and then by the use of MATLAB executable function. In the heat transfer algorithm, the value of the heat on a cell is dependent on its neighbors, while computing the value of the neighbor was used. GPU implementation also required to use CUDA Arrays and Texture Memory. The algorithm was implemented on GPU and high values of speedups were observed.

Design Methodology: In this lab we, were given a few files which had two projects and the other one was a MATLAB script to run the heat transfer simulation. The Algorithm was implemented as mentioned in the manual. First the declaration of texture memory was done. CUDA arrays were used for the implementation. the space for array was allocated using the function cudaMallocArray(), and data was copied to it by the use of cudaMemcpyToArray() function. cudaChannelFormatDesc() was used to define the arrangement of the memory based on float datatype. Filter mod was set to filter mode point, addressmode[0] and[1] were set to clamp. Normalized property was set to false as that was not needed for the process. 2D block and grid were taken for the implementation. After the data was copied to the CUDA array, cudaBindTextureToArray() was used to bind the array to the texture memory than the kernel was run. After that, cudaUnbindTexture() was used to unbind the array from texture memory and then the arrays values were updated after the kernel was executed. This process was run for each cell present. The kernel for the algorithm updated the values of the cell depending on the four neighboring cells. Tex2D() as used to retrieve the values from texture memory. After updating the values of all the cells present, the data was copied to the host. and the code was run by the use of MATLAB script which was provided.

Result: The table below shows all the results that were taken during the simulation: -

INPUT SIZE	CPU TIME (s)	GPU TIME(s)	SPEEDUP	ERROR
10X10	0.00716285	0.0462566	0.15485	1.41E-07
100X100	0.677183	0.081078	8.35224	8.90E-08
250X250	4.27178	0.31047	13.7591	7.43E-08
500X500	19.3149	1.01414	19.0455	9.36E-08
1000X1000	82.2143	3.68182	22.3298	1.87E-07
BREAK EVEN POINT- here it is 42X42				

Table 1: computation time, speedup and error.



Graph: Speedup VS Input Size

Conclusion: This involved the implementation of a scientific algorithm, the heat transfer algorithm was highly parallel and high speedup values were achieved on the device which can be seen in result section. In implementation we also used Texture memory and were able to make use of the locality that existed in the algorithm. Texture memory was very useful for the implementation of the algorithm and helped to make use of the device hardware in a better way.

Question: When you used constant and shared memory, you had to explicitly copy data into those locations. However, texture memory needs to have the notion of binding a variable to it. Perform some research and describe the sequence of events that occurs when you bind a variable to texture memory and then later access that value in the kernel.

- cudaBindTexture (size_t* offset, const textureReference* texref, const void* devPtr, const cudaChannelFormatDesc* desc, size_t size = UINT_MAX).
- Binds size bytes of the memory area pointed to by devPtr to the texture reference texref. desc describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to texref is unbound.

- Since the hardware enforces an alignment requirement on texture base addresses, cudaBindTexture() returns in *offset a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex2D() function.

- If the device memory pointer was returned from cudaMalloc(), the offset is guaranteed to be 0 and NULL may be passed as the offset parameter.
- The total number of elements (or texels) in the linear address range cannot exceed cudaDeviceProp::maxTexture1DLinear[0]. The number of elements is computed as (size / elementSize), where elementSize is determined from desc.