Dense Lower Triangular Solver

Code Hierarchy

File	Functions	Linked Files
Support.h Contain various supporting function for main.cu	1. verifyResults Take calculated matrix array and compare it with actual matrix to verify correctness. 2. printCSV Take matrix array and print it on console 3. writeCSV Takes a matrix array and write it in CSV format on file directory system 4. loadCSV Take matrix array with reference and csv file, and load csv file data info the matrix array and return	1. Support.cu (Implementation file) 2. Main.cu (Utility)
Kernel.h Contain actual kernel codes to be executed on GPU + some host codes	1. gpu_simple_solver_kernel Original kernel modified to run correctly 2. gpu_simple_solver_Anjum Modified kernel optimized for performance but not for scalability 3. gpu_optimized_solver_Anjum Modified kernel optimized for both performance and scalability 4. gpu_simple_solver Host code to call appropriate kernel 5. Cpu_multiply Host to process multiplication of matrix 6. Cpu_solver	1. Kernel.cu Implementation file 2. Main.cu Utility file

	Host to process solve equation using cpu	
main.cu	1. Onhost Called by main program with kernel type parameter and initialize different arrays and forward to the device or gpu 2. OnDevice Called by onhost function with host matrix array, initialize variable on device and forward request to gpu or cpu solver.	

Execution Sequence

(Host Code)

1. On the Command Line call the executable with Parameter or Kernel Type

   i.e main 1 or main 5

   Kernel Types

   0 : CPU Solver

   1 : Old Simple Sover kernel

   2 : gpu_initial sover Anjum

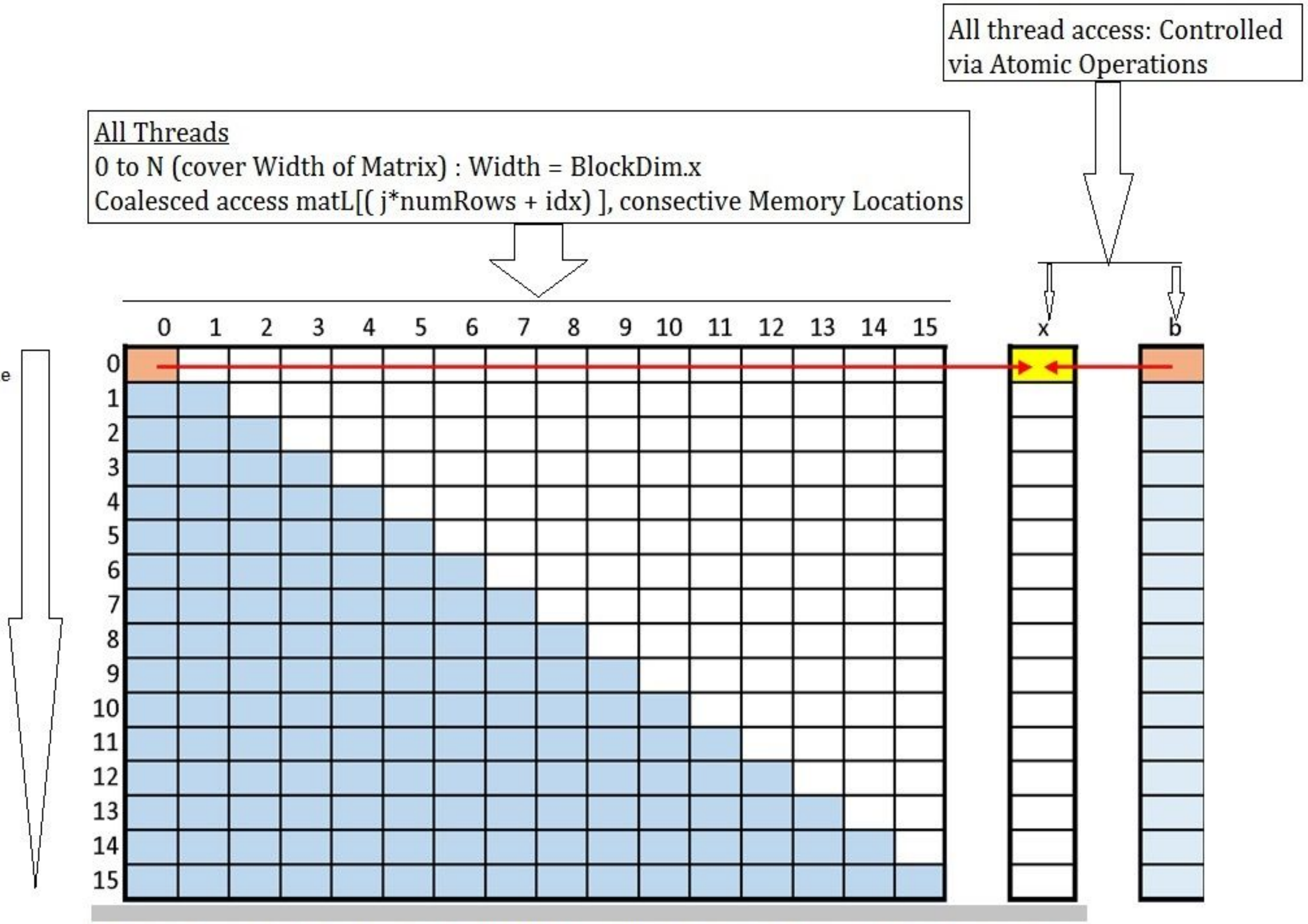
   3: gpu Simple solver kernel2

   4 : gpu_simple_solver_Anjum

   5: gpu_Optimized_Solver_Anjum
2. Main Method call the on host method
3. OnHost method initializes the input arrays with csv file from file system and call the onDevice method
4. OnDevice Method initializes the input arrays on device, copy data and call gpu_Solver method with kernel type.
5. Gpu_solver call the appropriate kernel and return the calculated array back to the ondevice method
6. Ondevice Method Prints / compare or write the calculated results on csv or display on console

Execution Sequence of KERNEL

(GPU_OPTIMIZED_SOLVER_ANJUM)



All thread access: Controlled via Atomic Operations

All Threads

0 to N (cover Width of Matrix) : Width = BlockDim.x

Coalesced access matL[(j*numRows + idx)], consective Memory Locations

Each Thread Traverse Through the hight of array using Loop J.

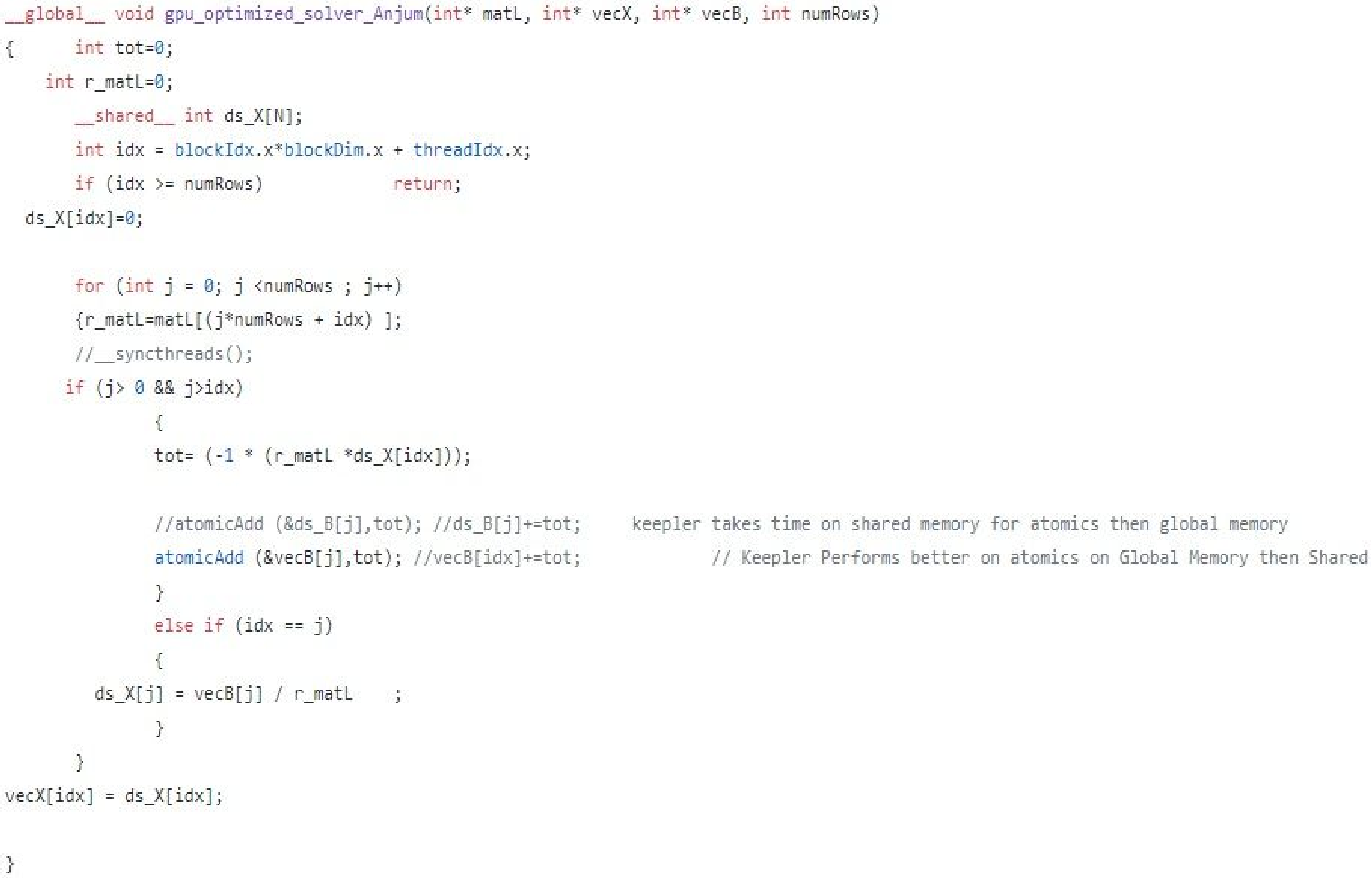
1. At Loop Counter 0:
Only Thread 0 Calculates

2. At Counter 1:
Thread 0 & 1 both Calculates

3. similarly at J=width or last Position being square matrix

All threads calculated and update X via atomics Operations

1. Matrix L, access Coalesced, from DRAM (On KEPLER architecture atomics on shared memory is expensive
2. Matrix X and B are access via shared memory for better Performance



```
__global__ void gpu_optimized_solver_Anjum(int* matL, int* vecX, int* vecB, int numRows)
{      int tot=0;
    int r_matL=0;
       __shared__ int ds_X[N];
       int idx = blockIdx.x*blockDim.x + threadIdx.x;
       if (idx >= numRows)             return;
  ds_X[idx]=0;


       for (int j = 0; j <numRows ; j++)
       {r_matL=matL[(j*numRows + idx) ];
       //__syncthreads();
      if (j> 0 && j>idx)
               {
               tot= (-1 * (r_matL *ds_X[idx]));

               //atomicAdd (&ds_B[j],tot); //ds_B[j]+=tot;     keepler takes time on shared memory for atomics then global memory
               atomicAdd (&vecB[j],tot); //vecB[idx]+=tot;             // Keepler Performs better on atomics on Global Memory then Shared
               }
               else if (idx == j)
               {
         ds_X[j] = vecB[j] / r_matL    ;
               }
       }
vecX[idx] = ds_X[idx];

}
```