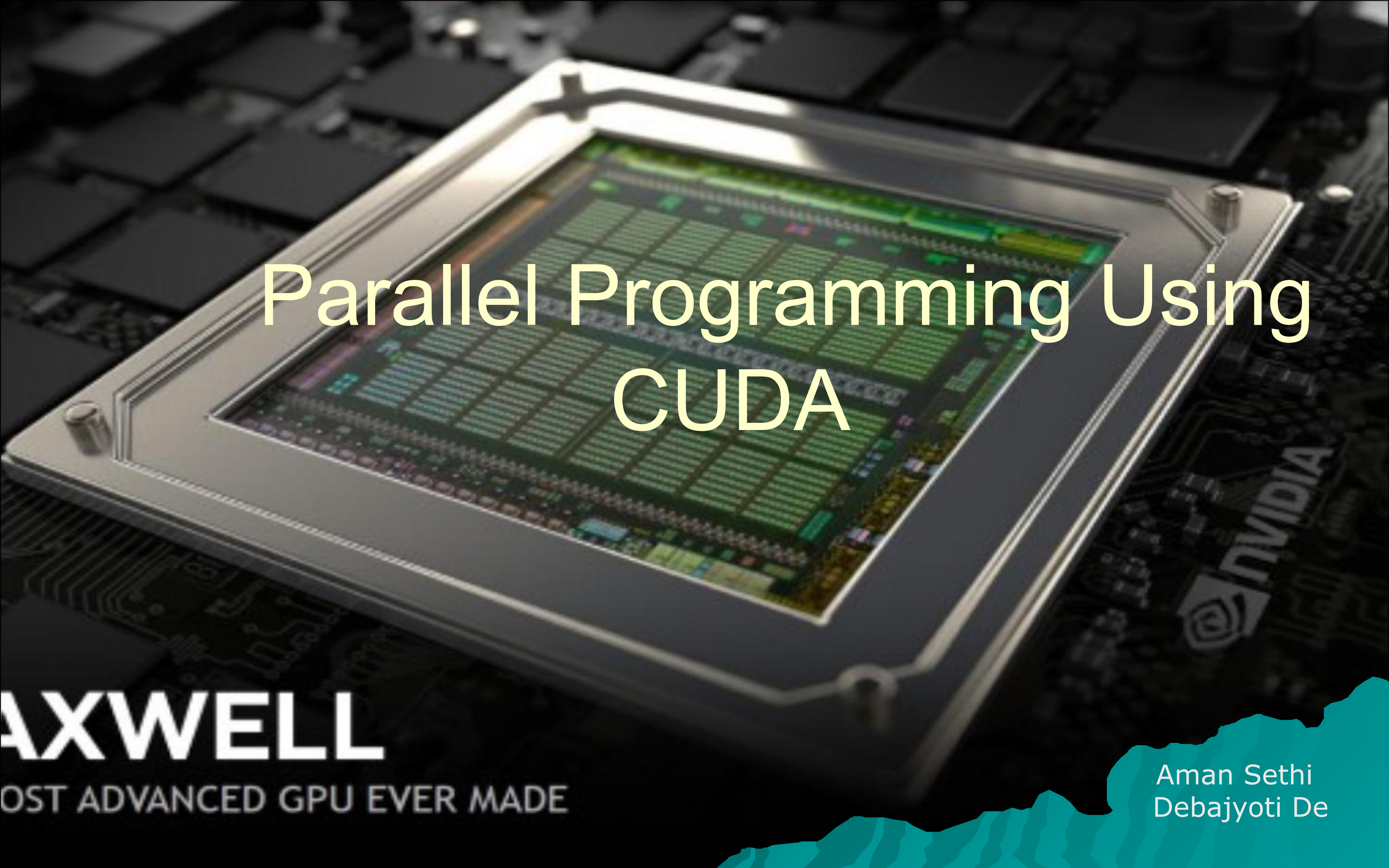


Parallel Programming Using CUDA

Aman Sethi
Debajyoti De

Outline

Parallel Programming

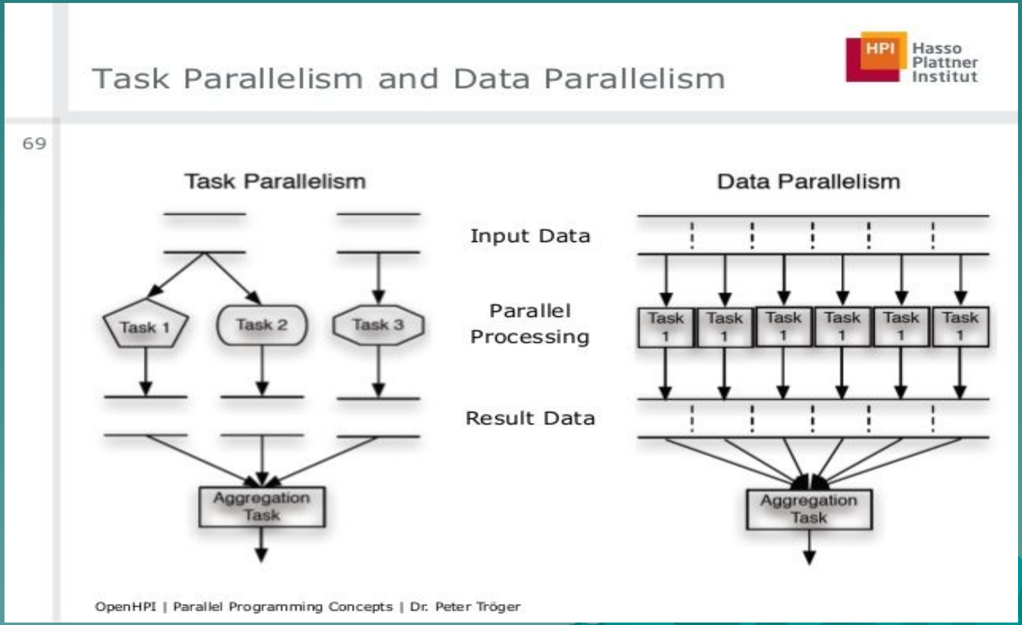
- In computing, a parallel programming model is an abstraction of parallel computer architecture, with which it is convenient to express algorithms and their composition in programs.

- A parallel program is composed of simultaneously executing processes. There are two different approaches of achieving parallelism while solving a problem.

- They are CONTROL PARALLEL approach and DATA PARALLEL approach.

Parallel Programming(continue)

- In Control Parallel approach, parallelism is achieved by applying different operations to the data set simultaneously.
- In Data Parallel approach, parallelism is achieved by applying same operation to the different data set simultaneously.



Task Parallelism and Data Parallelism

69

OpenHPI | Parallel Programming Concepts | Dr. Peter Tröger

Multithreaded Programming – A Type of Parallel Programming

- The threads model of parallel programming is one in which a single process (a single program) can spawn multiple, concurrent "threads" (sub-programs).
- Each thread runs independently of the others, although they can all access the same shared memory space.
- Threads can be used to implement both the data parallel and control parallel approach.
- A challenge of using threads is the issue of collisions and race conditions, which can be addressed using synchronization.

GPU Programming Using CUDA

GPU computing is possible because today's GPU does much more than render graphics: It sizzles with a teraflop of floating point performance and crunches application tasks designed for anything from finance to medicine.

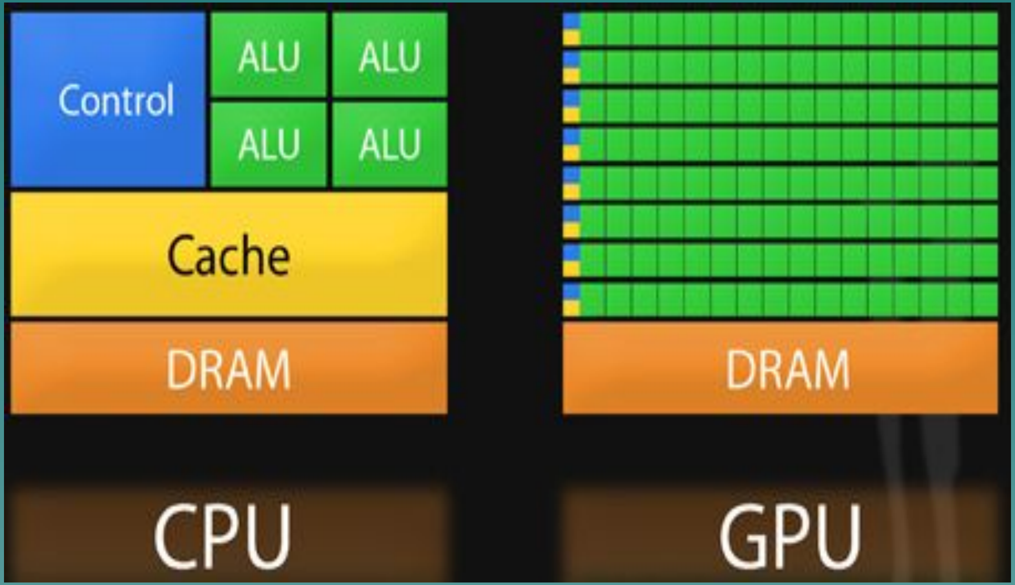
What is CUDA??

- Parallel computing architecture developed by NVIDIA
- CUDA programming interface consists of :
  - C language extensions to target portions of source code on the compute device
  - A runtime library split into:
    Host (CPU) component
    Device (GPU) component
    Common component

How CUDA works??

Data-parallel computing maps well to GPUs.

Identical operations on many data elements are performed in parallel.



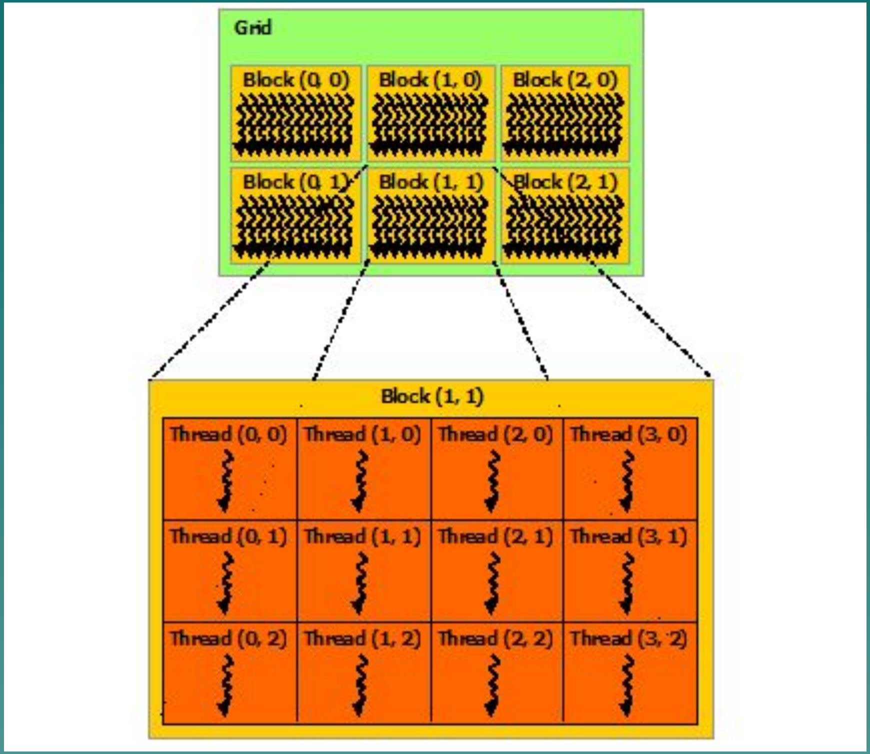
How CUDA works??

A usual template followed to program a GPU is as below:

- Transfer the data to the GPU memory.
- Launch the GPU kernel from the host program.
- Wait till GPU finishes off the calculations.
- Transfer the results back to CPU memory.

Thus, a CUDA program consists of two parts in its code: a) **host program** which runs on CPU and b) **kernel** which does the calculations on GPU

Global, constant, texture memories



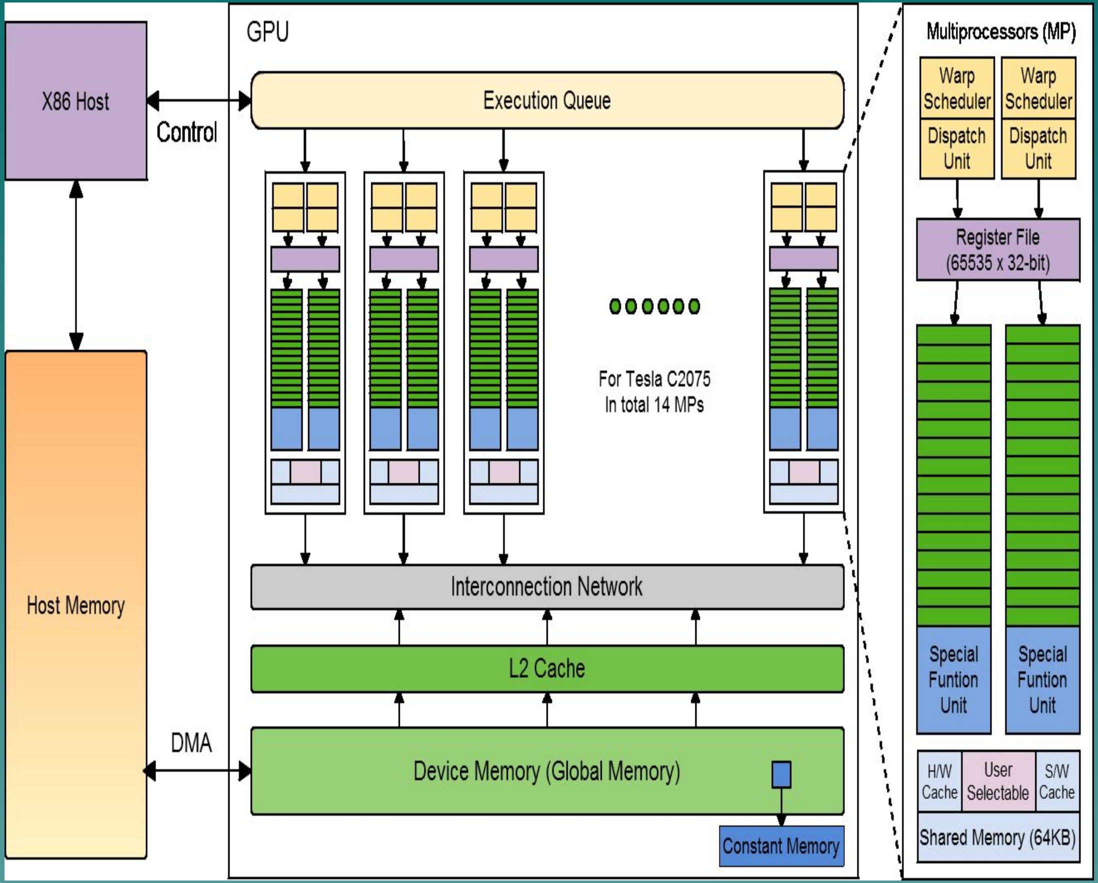
The execution of the kernels is done in threads and blocks.

A group of threads forms a thread Block.

A group of thread blocks forms a Grid.

Each thread performs same operations on a subset of data structure.

Threads within a block can synchronize using a barrier synchronization function **__syncthreads()**.

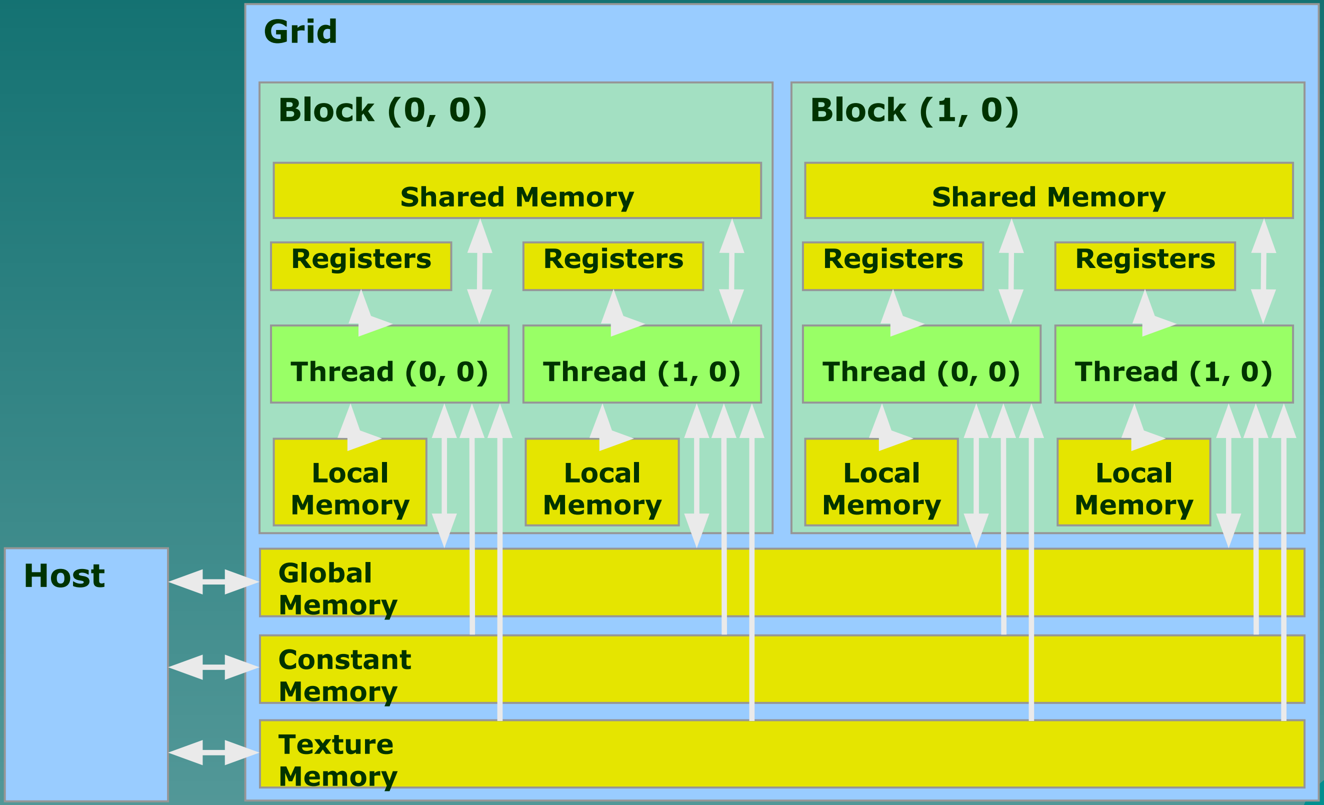


Blocks from the grid are distributed across streaming multiprocessors.

Each multiprocessor contains:

- Processor cores
- Scheduler
- Dispatcher
- Registers
- Caches
- Shared Memory

GPU Memory Hierarchy



GPU MEMORY MODEL SUMMARY

Memory Space	Managed By	Physical Implementation	Scope on GPU	Scope on CPU	Lifetime
Thread-Private Variables(Registers)	Compiler	On-Chip	Per Thread	Not visible	Lifetime of a thread
Thread-Private Variables(Local)	Compiler	Device Memory	Per Thread	Not visible	
Shared	Programmer	On-Chip	Block	Not visible	Block Lifetime
Global	Programmer	Device Memory	All threads	Read/Write	Application or until explicitly freed
Constant	Programmer	Device Memory	All threads	Read/Write	

CUDA Kernels

- CUDA kernels are launched by the host using modified C function call syntax.
  - Eg: vectorAdd <<< gridDimension , blockDimension >>> (A,B,C)
- Denoted with __global__ function qualifier
  - Eg: __global__ vectorAdd <<<....,.....>>> (...)
- CUDA Kernels have few restrictions
- Kernels have read-only built-in variables:
  - gridDim , blockDim : Dimensions of grid and thread block respectively
  - blockIdx : Unique index of block within grid
  - threadIdx : Unique index of thread within thread block
- Cannot vary size of blocks or grids during kernel call

CUDA Syntax

- Memory Management : Host code manages device memory.
  - cudaError_t cudaMalloc(void** devPtr, size_t size)
  - cudaError_t cudaMemset(void** devPtr, int value, size_t count)
  - cudaError_t cudaFree(void* devPtr)

- Data Transfer : Host code manages transfer of data to and from device.
  - cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)

- Must exercise caution when dereferencing pointers

Minimum in an Array Using CUDA

Approach

On CPU(Host)

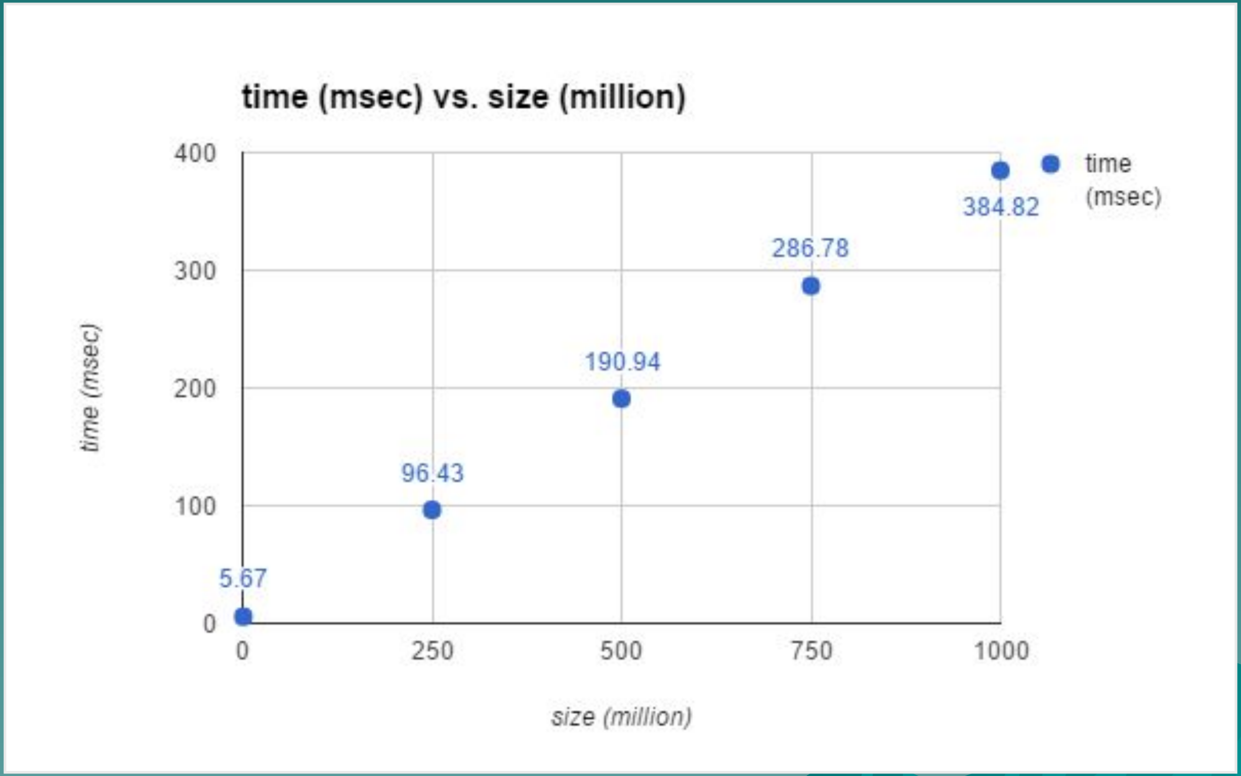
```
cudaEventCreate(&start);
cudaEventCreate(&stop);

// Allocate memory for array a and minarr
a = (char*)malloc(SIZE*sizeof(char));
minarr = (char*)malloc(NO_OF_BLOCKS*sizeof(char));

// Allocate memory on the GPU for array a and minarr
cudaMalloc(&d_a, SIZE*sizeof(char));
cudaMalloc(&d_minarr, NO_OF_BLOCKS*sizeof(char));

/* Copy the values in array a and minarr to the memory space allocated in the GPU */
cudaMemcpy(d_a, a, SIZE*sizeof(char), cudaMemcpyHostToDevice);
cudaMemcpy(d_minarr, minarr, NO_OF_BLOCKS*sizeof(char), cudaMemcpyHostToDevice);

cudaEventRecord(start);

// Launching the kernel
FindMin <<< NO_OF_BLOCKS, NO_OF_THREADS >>>(d_a,d_minarr, SIZE);
cudaThreadSynchronize();
cudaEventRecord(stop);

// The result array d_minarr is copied to minarr
cudaMemcpy(minarr, d_minarr, NO_OF_BLOCKS *sizeof(char), cudaMemcpyDeviceToHost);
cudaMemcpy(d_minarr, minarr, NO_OF_BLOCKS*sizeof(char), cudaMemcpyHostToDevice);

cudaEventRecord(start);
// Launching the kernel
FindMin <<< NO_OF_BLOCKS, NO_OF_THREADS >>>(d_a,d_minarr, SIZE);
cudaThreadSynchronize();
cudaEventRecord(stop);

// The result array d_minarr is copied to minarr
cudaMemcpy(minarr, d_minarr, NO_OF_BLOCKS *sizeof(char), cudaMemcpyDeviceToHost);

min = minarr[0];
// Finding the global minimum form minarr array.
for (i = 1; i < NO_OF_BLOCKS; i++)
{
    if (minarr[i] < min)
        min = minarr[i];
}

cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);

/* Free the memory which was allocated for a and minarr in the host's memory */
free(a);
free(minarr);

/* Free the memory which was allocated for a and minarr in the GPU */
cudaFree(d_a);
cudaFree(d_minarr);
```

On GPU(Device)

```
__global__ void FindMin(char *a,char *b, int n)
{
    // Cache array is shared by threads in a thread block.
    __shared__ char cache[NO_OF_THREADS];
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    int cacheIndex = threadIdx.x;
    cache[cacheIndex] = CHAR_MAX;

    for (int j = i; j < n;)
    {
        if (a[j] < cache[cacheIndex])
            cache[cacheIndex] = a[j];

        j += blockDim.x * gridDim.x;
    }

    /* A thread cannot proceed further unless all other threads reaches this point. */
    __syncthreads();
    /* Finding the minimum element from an array (cache) where each
        thread in a thread block has calculated its own minimum
        element */
    for (int j = 0; j < NO_OF_THREADS; j++)
    {
        if (ans>cache[j])
            ans = cache[j];
    }
    // Saving the minimum element calculated by the thread block.
    b[blockIdx.x] = ans;
}
```

Results

The execution time was recorded for different size of the array and the following graph was obtained.



GPU based Matrix Inversion using CUDA

Approach

- A variant of Gaussian elimination called Gauss–Jordan elimination can be used for finding the inverse of a matrix, if it exists.

- The n by n identity matrix is augmented to the right of A, forming a n by 2n block matrix [A | I].

- Now through application of elementary row operations, find the reduced echelon form of this n by 2n matrix.

- The matrix A is invertible if and only if the left block can be reduced to the identity matrix I; in this case the right block of the final matrix is A^{-1}

On CPU(Host)

```
dim3 threads(NO_OF_THREADS_X, NO_OF_THREADS_Y);
dim3 blocks(NO_OF_BLOCKS_X, NO_OF_BLOCKS_Y);

testColumnMatrix = new double[row];
factor = new double[row];

//Allocate memory in host for matrix
matrix = new double[(row*column)];

//Allocate memory in device for matrix
cudaMalloc(&d_matrix, (row*column)*sizeof(double));
cudaMalloc(&d_factor, (row)*sizeof(double));

//Initialize 2D Matrix
initializeMatrix(matrix, row, column);
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
cudaMemcpy(d_matrix, matrix, (row*column)*sizeof(double), cudaMemcpyHostToDevice);
for (int i = 0; i < row; i++)
{
     // Copy ith column from matrix stored in device to host to check whether Pivot element (ith element of ith row) is zero
    cudaMemcpy2D(testColumnMatrix, sizeof(double), &d_matrix[i], column*sizeof(double), sizeof(double), row,
cudaMemcpyDeviceToHost);
   // If pivot element is zero find a row with non-zero pivot element and interchange row elements.
    if (testColumnMatrix[i] == 0.0)
    {
        int rowNumber = findRow(testColumnMatrix, row, i);
```

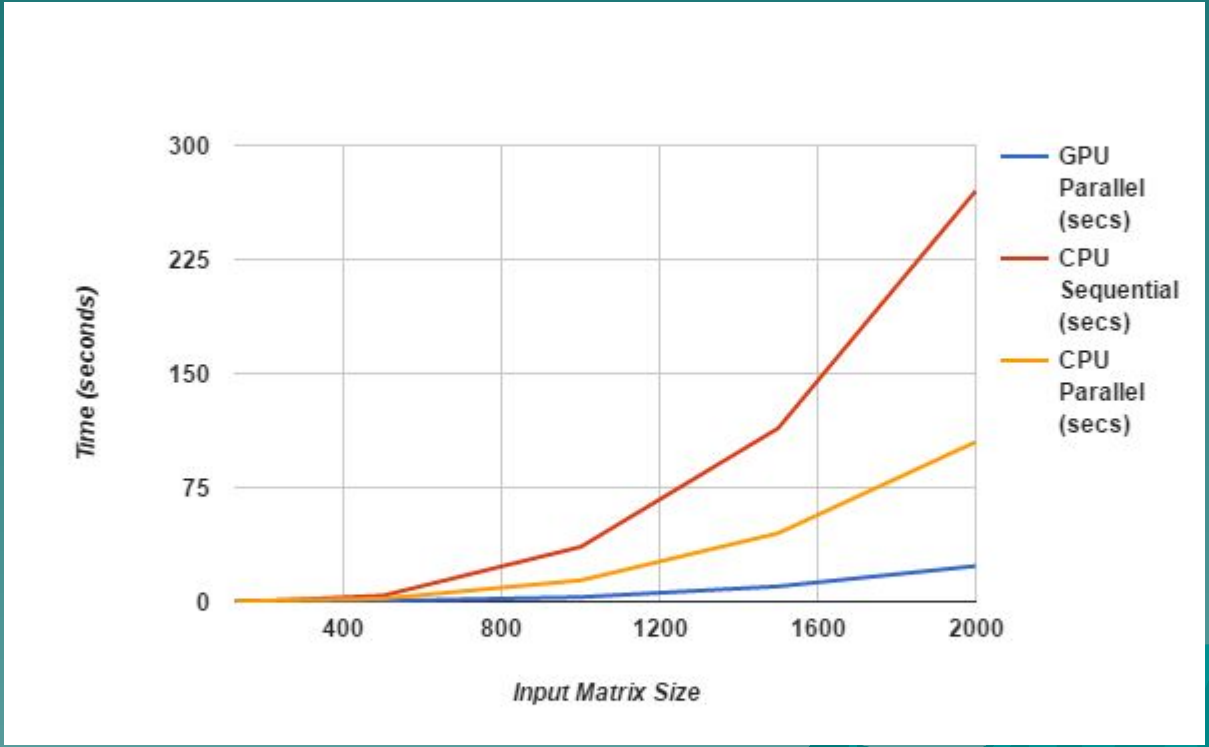
```
            if (rowNumber == -1)
            {
                printf("Given Matrix Impossible to invert. %d\n",i);
                return 0;
            }
            else
            {
                // Interchange rows if required
                switchRows << <numberOfBlocks, threadsPerBlock >> >(d_matrix, i, rowNumber, column);
                cudaDeviceSynchronize();
            }
        }
        // Copy ith column from matrix stored in device to host
        cudaMemcpy2D(testColumnMatrix, sizeof(double), &d_matrix[i], column*sizeof(double), sizeof(double), row,
cudaMemcpyDeviceToHost);

        // Normalize Pivot ( Make pivot element 1 )
        double pivotValue = testColumnMatrix[i];
        normalizePivot << <numberOfBlocks, threadsPerBlock >> >(d_matrix, column, i, pivotValue);
        cudaDeviceSynchronize();
        // Update other rows.
        for (int k = 0; k < row; k++)
            factor[k] = testColumnMatrix[k];
        cudaMalloc(&d_factor, (row)*sizeof(double));
        cudaMemcpy(d_factor, factor, row*sizeof(double), cudaMemcpyHostToDevice);
        updateOtherRows << <blocks, threads >> >(d_matrix, row, column, i, d_factor);
        cudaDeviceSynchronize();
    }
    cudaMemcpy(matrix, d_matrix, (row*column)*sizeof(double), cudaMemcpyDeviceToHost);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&total, start, stop);

    cudaFree(d_matrix);
    cudaFree(d_factor);
```

On GPU(Device)

```
// The parallelised function switches the values of two rows
__global__ void switchRows(double *d_matrix, int i, int rowNumber, int n)
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    if (id < n)
    {
        double temp = d_matrix[Index(i, id, n)];
        d_matrix[Index(i, id, n)] = d_matrix[Index(rowNumber, id, n)];
        d_matrix[Index(rowNumber, id, n)] = temp;
    }
}
// The parallelised function makes the pivot element value equal to one
__global__ void normalizePivot(double *d_matrix, int n, int rowNumber, double pivotValue)
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    if (id < n)
    {
        d_matrix[Index(rowNumber, id, n)] = (d_matrix[Index(rowNumber, id, n)] / pivotValue) ;
    }
}
// The parallelised function updates values of other rows so that elements above and below pivot element are zero
__global__ void updateOtherRows(double *d_matrix, int R, int C, int i, double *d_factor)
{
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int column = blockIdx.x*blockDim.x + threadIdx.x;
    if (row < R && row != i)
        if (column < C)
            d_matrix[Index(row, column, C)] = (d_matrix[Index(row, column, C)] - (d_factor[row] * d_matrix[Index(i, column, C)]));
}
```

Results

- The execution time for sequential program in CPU and parallel program in CPU as well as in GPU using CUDA for different matrix size are recorded.



Conclusion

- The execution time of matrix inversion reduces to almost half when we implement the Gauss-Jordan Elimination algorithm using multiple threads.
- The execution time of the parallel program using CUDA in GPU takes less than one-tenth of the time taken by a sequential program in a CPU when the order of the input matrix size is 2000 x 2000.

Application

The matrix inversion is required in:

- image reconstruction process
- Hill Cipher to obtain the inverse of a key matrix during the decryption process to get back the original plaintext.

Reference

- CUDA By Example
  - Jason Sanders
  - Edward Kandrot

- Programming GPU with CPU
  - Hasindu Gamaarachchi

- GTC Express Webinar: The CUDA Memory Model

- http://cuda-programming.blogspot.in

THANK YOU