Matrix Multiplication CUDA:

I started with the Matrix multiplication code in CUDA and C++. I have compared the time taken by them for the same matrix size filled with random numbers between 0 and 1 from a uniform distribution and have the program run 100000 to get the time taken in each run and then it's average and the standard deviation.

Below is the CUDA code:

```
#include <iostream>
#include <cmath>
#include <cuda_runtime.h>
#include <chrono>
using namespace std;

__global__ void matmul(float *d_A, float *d_B, float *d_C, int array_size) {
      int threadidrows = threadIdx.x + blockIdx.x * blockDim.x;
      int threadidclm = threadIdx.y + blockIdx.y * blockDim.y;

      for (int i = 0; i < array_size; i++) {
      d_C[threadidrows * array_size + threadidclm] += d_A[threadidrows * array_size + i] *
d_B[i * array_size + threadidclm];
      }
}

int main() {
      int array_size = 50;
      int array_sizebytes = array_size * array_size * sizeof(float);
      int gridsize = 2;
      int itr=100000;
      // Arrays to store timings
      double timetaken[itr];

      // Variables to store average and variance
      double sum = 0.0;
      double sumSquared = 0.0;

      for (int iteration = 0; iteration < itr; iteration++) {
      // Declaration of arrays
      float h_A[array_size * array_size];
      float h_B[array_size * array_size];
      float h_C[array_size * array_size];
      float norm = 1.0 / RAND_MAX;
```

```
      srand(time(0));

      for (int j = 0; j < array_size * array_size; j++) {
            h_A[j] = rand() * norm;
            h_B[j] = rand() * norm;
      }

      // Declaration of GPU variables and allocation of GPU memory
      float *d_A;
      float *d_B;
      float *d_C;

      cudaMalloc((void **)&d_A, array_sizebytes);
      cudaMalloc((void **)&d_B, array_sizebytes);
      cudaMalloc((void **)&d_C, array_sizebytes);

      cudaMemcpy(d_A, h_A, array_sizebytes, cudaMemcpyHostToDevice);
      cudaMemcpy(d_B, h_B, array_sizebytes, cudaMemcpyHostToDevice);

 dim3 dimblock(32, 32);
      dim3 dimgrid(gridsize, gridsize);

      // Starting the clock for each iteration
      auto start = chrono::high_resolution_clock::now();
      ios_base::sync_with_stdio(false);

      // Launching the kernel function
      matmul<<<dimgrid, dimblock>>>(d_A, d_B, d_C, array_size);

      // Synchronizing the device to ensure the kernel is complete
      cudaDeviceSynchronize();

      // Calculating the execution time of the function
      auto endtime = chrono::high_resolution_clock::now();
      timetaken[iteration] = chrono::duration_cast<chrono::nanoseconds>(endtime -
start).count();
      timetaken[iteration] *= 1e-9; // converting to seconds

      // Copying data back to the CPU
      cudaMemcpy(h_C, d_C, array_sizebytes, cudaMemcpyDeviceToHost);

      // Freeing GPU memory
      cudaFree(d_A);
```

```
      cudaFree(d_B);
      cudaFree(d_C);

      // Update sum and sumSquared
      sum += timetaken[iteration];
      sumSquared += timetaken[iteration] * timetaken[iteration];
      }

      // Calculate the average and variance
      double average = sum / itr;
      double variance = (sumSquared - itr * average * average) / (itr - 1);
      double standardDeviation = sqrt(variance);

      // Print the results
      cout << "Average time taken by the program is " << average << " seconds" << endl;
      cout << "Standard Deviation of time taken is " << standardDeviation << " seconds" <<
endl;

      return 0;
}
```

OUTPUT:
Average time taken by the program is 8.80134e-05 seconds
Standard Deviation of time taken is 2.37797e-06 seconds

real 0m37.687s
user 0m18.624s
sys 0m18.093s

Matrix Multiplication C++:

```
#include <iostream>
#include <vector>
#include <ctime>
#include <chrono>
#include <cmath> // Include the cmath header for pow and sqrt functions
```

```
using namespace std;

void Matmul(vector<vector<double>>& A, vector<vector<double>>& B, vector<vector<double>>& C)
{
	int m = A.size();
	for (int i = 0; i < m; i++) {
	for (int j = 0; j < m; j++) {
		for (int k = 0; k < m; k++) {
			C[i][j] += A[i][k] * B[k][j];
		}
	}
	}
}

int main() {
	const int iterations = 100000;
	const int m = 50;

	vector<vector<double>> A(m, vector<double>(m));
	vector<vector<double>> B(m, vector<double>(m));
	vector<vector<double>> C(m, vector<double>(m));

	srand(time(0));

	// Arrays to store timings
	double timetaken[iterations];

	for (int iter = 0; iter < iterations; iter++) {
	// Populate matrices A and B with random values
	for (int i = 0; i < m; i++) {
		for (int j = 0; j < m; j++) {
			A[i][j] = rand()/RAND_MAX;
			B[i][j] = rand()/RAND_MAX;
		}
	}

	auto start = chrono::high_resolution_clock::now();
	ios_base::sync_with_stdio(false);

	Matmul(A, B, C);

	auto endtime = chrono::high_resolution_clock::now();
```

```
      timetaken[iter] = chrono::duration_cast<chrono::nanoseconds>(endtime - start).count() *
1e-9;
      }

      // Calculate average and standard deviation
      double average = 0.0;

 for (int iter = 0; iter < iterations; iter++) {
      average += timetaken[iter];
      }
      average /= iterations;

      double variance = 0.0;
      for (int iter = 0; iter < iterations; iter++) {
      variance += pow(timetaken[iter] - average, 2);
      }
      variance /= iterations;

      double standardDeviation = sqrt(variance);

      cout << "Average time taken: " << average << " seconds" << endl;
      cout << "Standard deviation: " << standardDeviation << " seconds" << endl;

      return 0;
}
```

OUTPUT:
Average time taken: 0.00170566 seconds
Standard deviation: 1.8775e-05 seconds

real   2m56.754s
user   2m56.335s
sys   0m0.004s

Here we see that the time complexity of matmul kernel in CUDA language has the time complexity **O(n)** while the time complexity in C++ code is **O(n^3)** where n is the number of columns or rows of the square matrix that we are multiplying.

Parallel Gauss Elimination:

Here I have written two versions of Gauss elimination codes in CUDA , parallel and non parallel.

```
#include <iostream>
#include <cmath>
#include <cuda_runtime.h>
#include <chrono>
#include <time.h>
using namespace std;

const int array_size = 512;

// CUDA kernel for Gaussian elimination
__global__ void gaussElimination(double* A, double* determinant, int n) {
      int tid_x = threadIdx.x + blockDim.x * blockIdx.x;
      int tid_y = threadIdx.y + blockDim.y * blockIdx.y;

      int index = 0;
      // Perform Gaussian elimination with partial pivoting
      for (int i = 0; i < n; i++) {
      // Partial Pivoting: Find the pivot row with the maximum absolute value
      int pivotRow = i;
      double maxVal = fabs(A[i * n + i]);
      for (int k = i + 1; k < n; k++) {
            double val = fabs(A[k * n + i]);
            if (val > maxVal) {
                  maxVal = val;
                  pivotRow = k;
            }
      }

      // Swap the current row with the pivot row
      if (i != pivotRow) {
            index++;
            // Use thread index for parallelization
            double temp_A = A[i * n + tid_y];
```

```
            A[i * n + tid_y] = A[pivotRow * n + tid_y];
            A[pivotRow * n + tid_y] = temp_A;
      }

      __syncthreads();

      // Eliminate other elements in the current column
      if (i == tid_x) {
            // Only the selected column performs the elimination
            for (int j = i + 1; j < n; j++) {
                  double factor_A = A[j * n + i] / A[i * n + i];
                  A[j * n + tid_y] -= A[i * n + tid_y] * factor_A;
            }
      }
      __syncthreads();
      }

      if (tid_x == 0 && tid_y == 0) {
      double deter = 1.0;

  for (int i = 0; i < n; i++) {
 deter *= A[i * n + i];
      }
      *determinant = deter * pow(-1, index);
      }
}

int main() {
      double determinant = 1.0;

      double A[array_size * array_size]
;//{7.1,6,1,-4.5,0,2,0,17.4,8.36,71,0,1.2,8.07,5.2,1.36,0.1}; /*{1, 1, 1, 5, -4, -2.66667,
2.33333, -0.666667, -2.33333, 2.66667, -0.333333, 0.2, -0.2, -0.2, 0.8, -1, -1, -3, 1, 0, 0,
0, 4, 1, 1};*/
      double* d_determinant, *d_A;
 float norm = 1.0f / RAND_MAX;
      srand(time(0)); // Seed for the random number generator
      for (int j = 0; j < array_size* array_size; j++) {
      A[j] = rand() * norm;
      }

      cudaMalloc((void**)&d_determinant, sizeof(double));
      cudaMalloc((void**)&d_A, array_size * array_size * sizeof(double));
```

```
    cudaMemcpy(d_determinant, &determinant, sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_A, A, array_size * array_size * sizeof(double), cudaMemcpyHostToDevice);

    dim3 blockDim(32, 32); // 2D block with 5x5 threads
    dim3 gridDim(1, 1);  // 2D grid with 1x1 blocks
auto start = chrono::high_resolution_clock::now();
  ios_base::sync_with_stdio(false);
    gaussElimination<<<gridDim, blockDim>>>(d_A, d_determinant, array_size);
 cudaDeviceSynchronize();
   // calculating the execution time of the function
    auto endtime = chrono::high_resolution_clock::now();
    double timetaken = chrono::duration_cast<chrono::nanoseconds>(endtime - start).count();
    timetaken *= 1e-9;
    cudaMemcpy(&determinant, d_determinant, sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(A, d_A, array_size * array_size * sizeof(double), cudaMemcpyDeviceToHost);

    cout << "Row echelon form of the given matrix is" << endl;
    for (int i = 0; i < array_size; i++) {
    for (int j = 0; j < array_size; j++) {
          cout << A[i * array_size + j] << " ";
 }
    cout << endl;
    }
cout<<"time is "<<timetaken<<"seconds"<<endl;
    cudaFree(d_A);
    cudaFree(d_determinant);

    cout << "The determinant is " << determinant << endl;

    return 0;
}
```

OUTPUT:
time is 0.0177148seconds
The determinant is -3.5464e-207

real 0m4.073s
user 0m0.290s
sys 0m2.073s

Non-parallel Gauss Elimination:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <chrono>
#include <cuda_runtime.h>
#define N 512
using namespace std;
__global__ void gaussElimination(float *A, float *det, int n) {
      int tid = threadIdx.x + blockDim.x * blockIdx.x;
      int index=0;
      // Perform Gaussian elimination with partial pivoting
      for (int i = 0; i < n; i++) {
      // Partial Pivoting: Find the pivot row with the maximum absolute value
      int pivotRow = i;
      float maxVal = fabsf(A[i * n + i]);
      for (int k = i + 1; k < n; k++) {
            float val = fabsf(A[k * n + i]);
            if (val > maxVal) {
                  maxVal = val;
                  pivotRow = k;
            }
      }

      // Swap the current row with the pivot row
      if (i != pivotRow) {
                  index++;
            for (int k = i; k < n; k++) {
                  float temp = A[i * n + k];
                  A[i * n + k] = A[pivotRow * n + k];
                  A[pivotRow * n + k] = temp;
            }
      }
      __syncthreads();

      // Eliminate other elements in the current column
      for (int j = i + 1 + tid; j < n; j += blockDim.x * gridDim.x) {
            float factor = A[j * n + i] / A[i * n + i];
            for (int k = i; k < n; k++) {
                  A[j * n + k] -= A[i * n + k] * factor;
            }
```

```
      }
      __syncthreads();
      }


      *det*=pow(-1,index);
}
int main() {
      int n = N; // Change this to the desired matrix dimension
      float *h_A, *d_A, h_det, *d_det;
      size_t size = n * n * sizeof(float);
      int gridsize=16777216;

  // Allocate host memory
      h_A = (float *)malloc(size);
      h_det = 2.0f;

      // Initialize the matrix with random numbers between 0 and 1
      float norm = 1.0f / RAND_MAX;
      srand(time(0)); // Seed for the random number generator
      for (int j = 0; j < n * n; j++) {
      h_A[j] = rand() * norm;
      }

      // Allocate device memory
      cudaMalloc((void **)&d_A, size);
      cudaMalloc((void **)&d_det, sizeof(float));

      // Copy data from host to device
      cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

      // Set the block and grid dimensions
  //  int blockSize = 256;
 dim3 dimblock(1024,1);
      dim3 dimgrid(gridsize,1);
      // Launch the kernel
   auto start = chrono::high_resolution_clock::now();
  ios_base::sync_with_stdio(false);

      gaussElimination<<<dimgrid, dimblock>>>(d_A, d_det, n);
      cudaDeviceSynchronize();
   // calculating the execution time of the function
      auto endtime = chrono::high_resolution_clock::now();
```

```
    double timetaken = chrono::duration_cast<chrono::nanoseconds>(endtime - start).count();
    timetaken *= 1e-9; // converting to seconds
    // Copy the result back to the host
    cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(&h_det, d_det, sizeof(float), cudaMemcpyDeviceToHost);

    // Display the row echelon form
/*  printf("Row Echelon Form:\n");
    for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
          printf("%.6f\t", h_A[i * n + j]);
    }
    printf("\n");
    }*/

    h_det = 1.0f;
    for (int i = 0; i < n; i++) {
          h_det *= h_A[i * n + i];
    }

  //printing the time taken
    cout<<"time is "<<timetaken<<"seconds"<<endl;
   cout <<"Determinant: "<<h_det<<endl;

    // Free allocated memory
    free(h_A);
    cudaFree(d_A);
    cudaFree(d_det);

    return 0;
}
```

OUTPUT: time is 184.174seconds
Determinant: 0
real 3m6.337s
user 1m37.734s
sys 1m28.085s

The non parallel Gauss elimination code has time complexity of O(n^5) while the Parallel version has time complexity of O(n^3).

Gram-Schmidt parallel :

```
#include <iostream>
#include <cmath>
#include <cuda_runtime.h>

const int array_size = 5;

using namespace std;

// CUDA kernel for Gaussian elimination
__global__ void gaussElimination(double* A, double* determinant, int n) {
	int tid_x = threadIdx.x + blockDim.x * blockIdx.x;
	int tid_y = threadIdx.y + blockDim.y * blockIdx.y;

	int index = 0;
	// Perform Gaussian elimination with partial pivoting
	for (int i = 0; i < n; i++) {
	// Partial Pivoting: Find the pivot row with the maximum absolute value
	int pivotRow = i;
	double maxVal = fabs(A[i * n + i]);
	for (int k = i + 1; k < n; k++) {
		double val = fabs(A[k * n + i]);
		if (val > maxVal) {
			maxVal = val;
			pivotRow = k;
		}
	}

	// Swap the current row with the pivot row
	if (i != pivotRow) {
		index++;
		// Use thread index for parallelization
		double temp_A = A[i * n + tid_y];
		A[i * n + tid_y] = A[pivotRow * n + tid_y];
		A[pivotRow * n + tid_y] = temp_A;
	}

	__syncthreads();

	// Eliminate other elements in the current column
	if (i == tid_x) {
```

```
            // Only the selected column performs the elimination
            for (int j = i + 1; j < n; j++) {
                  double factor_A = A[j * n + i] / A[i * n + i];
                  A[j * n + tid_y] -= A[i * n + tid_y] * factor_A;
            }
      }
      __syncthreads();
      }

      if (tid_x == 0 && tid_y == 0) {
      double deter = 1.0;
      for (int i = 0; i < n; i++) {
            deter *= A[i * n + i];
      }
      *determinant = deter * pow(-1, index);
      }
}

__global__ void GSGE(double *A, double *A_T, double *AAT, int n, double *determinant) {
      int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
      int tid_y = threadIdx.y + blockIdx.y * blockDim.y;

      // Transpose matrix in parallel
      if (tid_x < n && tid_y < n) {
      A_T[tid_y * n + tid_x] = A[tid_x * n + tid_y];
      }
      __syncthreads();

      // Matrix multiplication in parallel
      if (tid_x < n && tid_y < n) {
      for (int i = 0; i < n; ++i) {
            AAT[tid_x * n + tid_y] += A_T[tid_x * n + i] * A[i * n + tid_y];
      }
      }
      __syncthreads();

      // Perform Gaussian elimination with partial pivoting
   // int index = 0;
      for (int i = 0; i < n; i++) {
      if (i == tid_x) {
            // Only the selected column performs the elimination
            for (int j = i + 1; j < n; j++) {
                  double factor_AAT = AAT[j * n + i] / AAT[i * n + i];
```

```
                AAT[j * n + tid_y] -= AAT[i * n + tid_y] * factor_AAT;
                A_T[j * n + tid_y] -= A_T[i * n + tid_y] * factor_AAT;
            }
      }

      __syncthreads();
      }
A_T[tid_x * array_size + tid_y] /=sqrt(abs(AAT[tid_x* array_size + tid_x])); // normalization
}

__global__ void orthochecker(double *A, double *A_T, double *AAT, int n) {
      int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
      int tid_y = threadIdx.y + blockIdx.y * blockDim.y;

      // Transpose matrix in parallel
      if (tid_x < n && tid_y < n) {
      A_T[tid_y * n + tid_x] = A[tid_x * n + tid_y];
      }
      __syncthreads();

      // Matrix multiplication in parallel
      if (tid_x < n && tid_y < n) {
      for (int i = 0; i < n; ++i) {
            AAT[tid_x * n + tid_y] += A[tid_x * n + i] * A_T[i * n + tid_y];
      }
      }
      __syncthreads();
}

int main() {
      double determinant = 1.0;
      double A_T[array_size * array_size];
      double I[array_size * array_size];
      double AAT[array_size * array_size];
      double A[array_size * array_size];
      int arraysize_bytes = array_size * array_size * sizeof(double);

      float norm = 1.0f / RAND_MAX;
      srand(time(0)); // Seed for the random number generator
      for (int j = 0; j < array_size* array_size; j++) {
      A[j] = rand() * norm;
      }
```

```
      double *d_determinant, *d_A, *d_A_T, *d_AAT, *d_U_T, *d_I;
      cudaMalloc((void **)&d_determinant, sizeof(double));
      cudaMalloc((void **)&d_A, arraysize_bytes);
      cudaMalloc((void **)&d_A_T, arraysize_bytes);
      cudaMalloc((void **)&d_AAT, arraysize_bytes);
      cudaMalloc((void **)&d_U_T, arraysize_bytes);
      cudaMalloc((void **)&d_I, arraysize_bytes);

      cudaMemcpy(d_A, A, arraysize_bytes, cudaMemcpyHostToDevice);

      dim3 blockDim(5, 5);
      dim3 gridDim((array_size + blockDim.x - 1) / blockDim.x, (array_size + blockDim.y - 1)
/ blockDim.y);

      // Launch the GSGE kernel
      GSGE<<<gridDim, blockDim>>>(d_A, d_A_T, d_AAT, array_size, d_determinant);

      cudaMemcpy(A_T, d_A_T, arraysize_bytes, cudaMemcpyDeviceToHost);
      cudaMemcpy(AAT, d_AAT, arraysize_bytes, cudaMemcpyDeviceToHost);
      cudaMemcpy(&determinant, d_determinant, sizeof(double), cudaMemcpyDeviceToHost);

      cout << "Determinant: " << determinant << endl;// to sometimes check if the AAT matrix
is getting into row echelon form

      // Print results or perform other actions as needed
      cout << "AAT matrix is" << endl;
      for (int i = 0; i < array_size; i++) {
      for (int j = 0; j < array_size; j++) {
            cout << AAT[i * array_size + j] << " ";
      }
      cout << endl;
      }

      cout << "A_T matrix is" << endl;
      for (int i = 0; i < array_size; i++) {
      for (int j = 0; j < array_size; j++) {
            cout << A_T[i * array_size + j] << " ";
      }
      cout << endl;
      }
```

```
cout << "A_T matrix is" << endl;
for (int i = 0; i < array_size; i++) {
for (int j = 0; j < array_size; j++) {
      cout << A_T[i * array_size + j] << " ";
}
cout << endl;
}

cudaMemcpy(d_A_T, A_T, arraysize_bytes, cudaMemcpyHostToDevice);
orthochecker<<<gridDim, blockDim>>>(d_A_T, d_U_T, d_I, array_size);
cudaMemcpy(I, d_I, arraysize_bytes, cudaMemcpyDeviceToHost);

cout << "I matrix is" << endl;
for (int i = 0; i < array_size; i++) {
for (int j = 0; j < array_size; j++) {
      cout << I[i * array_size + j] << " ";
}
cout << endl;
}

cout << "A_T matrix is" << endl;
for (int i = 0; i < array_size; i++) {
for (int j = 0; j < array_size; j++) {
      cout << A_T[i * array_size + j] << " ";
}
cout << endl;
}

cudaMemcpy(d_A_T, A_T, arraysize_bytes, cudaMemcpyHostToDevice);

gaussElimination<<<gridDim, blockDim>>>(d_A_T, d_determinant, array_size);
cudaMemcpy(&determinant, d_determinant, sizeof(double), cudaMemcpyDeviceToHost);
cudaMemcpy(A_T, d_A_T, arraysize_bytes, cudaMemcpyDeviceToHost);

cout << "Row echelon form of the given matrix is" << endl;
for (int i = 0; i < array_size; i++) {
for (int j = 0; j < array_size; j++) {
      cout << A_T[i * array_size + j] << " ";
}
cout << endl;
}
```

```
    cout << "Determinant: " << determinant << endl;

    cudaFree(d_A_T);
    cudaFree(d_A);
    cudaFree(d_AAT);
    cudaFree(d_determinant);

    return 0;
}
```

OUTPUT:
Determinant: 0

AAT matrix is
0.818496 0.643764 0.852621 1.07323 1.26378
1.35047e-17 0.0392015 -0.123278 0.0726644 -0.0813183
5.80084e-18 -3.58413e-18 0.158006 0.0616504 0.0711478
1.29157e-17 6.08665e-19 -4.02979e-18 0.529409 -0.139562
-6.13059e-17 -6.92645e-18 -1.58927e-18 1.04412e-17 0.052241

A_T matrix is
0.377125 0.249379 0.785784 0.0827649 0.41386
-0.85348 0.239684 0.404264 0.153282 -0.164919
0.128125 0.118291 -0.181419 0.967115 -0.0369816
0.241535 0.820309 -0.106648 -0.170609 -0.47778
-0.233652 0.439844 -0.418124 -0.0723616 0.756228

I matrix is
1 7.82661e-17 4.52301e-17 4.36201e-17 -2.44416e-16
7.82661e-17 1 2.14184e-15 -1.30698e-15 5.25087e-16
4.52301e-17 2.14184e-15 1 -2.03455e-15 1.54068e-15
4.36201e-17 -1.30698e-15 -2.03455e-15 1 2.17395e-16
-2.44416e-16 5.25087e-16 1.54068e-15 2.17395e-16 1

A_T matrix is
0.377125 0.249379 0.785784 0.0827649 0.41386
-0.85348 0.239684 0.404264 0.153282 -0.164919
0.128125 0.118291 -0.181419 0.967115 -0.0369816
0.241535 0.820309 -0.106648 -0.170609 -0.47778
-0.233652 0.439844 -0.418124 -0.0723616 0.756228

Row echelon form of the given matrix is

-0.85348 0.239684 0.404264 0.153282 -0.164919
2.19522e-17 0.888139 0.00775902 -0.12723 -0.524452
1.78057e-18 -1.12537e-17 0.961311 0.201392 0.550787
8.1785e-18 8.18442e-19 -1.19172e-17 1.0378 0.0993047
4.14886e-18 -1.52362e-18 -3.28531e-17 2.17576e-18 1.32235

Determinant: 1
real 0m2.240s
user 0m0.013s
sys 0m2.110s

The time spend in the executing the program is 0.013s. In the program we have three kernels , our main kernel is only the **GSGE(Gram-Schmidt using Gauss Elimination)** kernel whose time complexity is, O(n^3) .The Gauss Elimination kernel and Orthochecker are just there to check whether GSGE is working properly.

Complex Gram-Schmidt parallel :

The above code for Gram-Schmidt is for real entries below is the code for complex entries.

```
#include <iostream>
#include <cmath>
#include <cuda_runtime.h>
#include <cuComplex.h>

const int array_size = 3;

using namespace std;

// Complex number structure
typedef cuDoubleComplex cuDoubleComplex;


__global__ void gaussElimination(cuDoubleComplex* A, cuDoubleComplex* determinant, int n) {
      int tid_x = threadIdx.x + blockDim.x * blockIdx.x;
      int tid_y = threadIdx.y + blockDim.y * blockIdx.y;

      int index = 0;

      // Perform Gaussian elimination with partial pivoting
      for (int i = 0; i < n; i++) {
      // Partial Pivoting: Find the pivot row with the maximum absolute value
```

```
	int pivotRow = i;
	double maxVal = cuCabs(A[i * n + i]);
	for (int k = i + 1; k < n; k++) {
		double val = cuCabs(A[k * n + i]);
		if (val > maxVal) {
			maxVal = val;
			pivotRow = k;
		}
	}

	// Swap the current row with the pivot row
	if (i != pivotRow) {
		index++;
		// Use thread index for parallelization
		cuDoubleComplex temp_A = A[i * n + tid_y];
		A[i * n + tid_y] = A[pivotRow * n + tid_y];
		A[pivotRow * n + tid_y] = temp_A;
	}

	__syncthreads();

	// Eliminate other elements in the current column
	for (int j = i + 1; j < n; j++) {
		cuDoubleComplex factor_A = cuCdiv(A[j * n + i], A[i * n + i]);
		A[j * n + tid_y] = cuCsub(A[j * n + tid_y], cuCmul(A[i * n + tid_y], factor_A));
	}

	__syncthreads();
	}

	if (tid_x == 0 && tid_y == 0) {
	cuDoubleComplex deter = make_cuDoubleComplex(1.0, 0.0);
	for (int i = 0; i < n; i++) {
		deter = cuCmul(deter, A[i * n + i]);
	}
	*determinant = cuCmul(deter, make_cuDoubleComplex(pow(-1, index), 0.0));
	}
}

__global__ void GSGE(cuDoubleComplex* A, cuDoubleComplex* A_T, cuDoubleComplex* AAT, int n)
{
	int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
	int tid_y = threadIdx.y + blockIdx.y * blockDim.y;
```

```
	// Transpose matrix in parallel
	if (tid_x < n && tid_y < n) {
	A_T[tid_y * n + tid_x] = cuConj(A[tid_x * n + tid_y]);
	}
	__syncthreads();

	// Matrix multiplication in parallel
	if (tid_x < n && tid_y < n) {
	cuDoubleComplex sum = make_cuDoubleComplex(0.0, 0.0);
	for (int i = 0; i < n; ++i) {
		sum = cuCadd(sum, cuCmul(A_T[tid_x * n + i], A[i * n + tid_y]));
	}
	AAT[tid_x * n + tid_y] = sum;
	}
	__syncthreads();

	// Gaussian elimination for AAT
	for (int i = 0; i < n; i++) {
	if (i == tid_x) {
		// Only the selected column performs the elimination
		for (int j = i + 1; j < n; j++) {
			cuDoubleComplex factor_AAT = cuCdiv(AAT[j * n + i], AAT[i * n + i]);
			AAT[j * n + tid_y] = cuCsub(AAT[j * n + tid_y], cuCmul(AAT[i * n + tid_y],
factor_AAT));
			A_T[j * n + tid_y] = cuCsub(A_T[j * n + tid_y], cuCmul(A_T[i * n + tid_y],
factor_AAT));
		}
	}
	__syncthreads();
	}

	A_T[tid_x * n + tid_y] = cuCdiv(A_T[tid_x * n + tid_y],
make_cuDoubleComplex(sqrt(cuCabs(AAT[tid_x * n + tid_x])), 0.0)); // normalization

}
__global__ void orthochecker(cuDoubleComplex* A, cuDoubleComplex* A_T, cuDoubleComplex* AAT,
int n) {
	int tid_x = threadIdx.x + blockIdx.x * blockDim.x;
	int tid_y = threadIdx.y + blockIdx.y * blockDim.y;

	// Transpose matrix in parallel
	if (tid_x < n && tid_y < n) {
```

```
      A_T[tid_y * n + tid_x] = cuConj(A[tid_x * n + tid_y]);
      }
      __syncthreads();

      // Matrix multiplication in parallel
      if (tid_x < n && tid_y < n) {
      cuDoubleComplex sum = make_cuDoubleComplex(0.0, 0.0);
      for (int i = 0; i < n; ++i) {
            sum = cuCadd(sum, cuCmul(A[tid_x * n + i], A_T[i * n + tid_y]));
      }
      AAT[tid_x * n + tid_y] = sum;
      }
      __syncthreads();
}

int main() {
      cuDoubleComplex determinant = make_cuDoubleComplex(1.0, 0.0);
      cuDoubleComplex A_T[array_size * array_size];
      cuDoubleComplex I[array_size * array_size];
      cuDoubleComplex AAT[array_size * array_size];
      cuDoubleComplex A[array_size * array_size];
      int arraysize_bytes = array_size * array_size * sizeof(cuDoubleComplex);

      float norm = 1.0f / RAND_MAX;
      srand(time(0)); // Seed for the random number generator
      for (int j = 0; j < array_size * array_size; j++) {
      A[j].x = rand() * norm;
      A[j].y = rand() * norm;
      }

      cuDoubleComplex* d_determinant, *d_A, *d_A_T, *d_AAT, *d_U_T, *d_I;
      cudaMalloc((void**)&d_determinant, sizeof(cuDoubleComplex));
      cudaMalloc((void**)&d_A, arraysize_bytes);
      cudaMalloc((void**)&d_A_T, arraysize_bytes);
      cudaMalloc((void**)&d_AAT, arraysize_bytes);
      cudaMalloc((void**)&d_U_T, arraysize_bytes);
      cudaMalloc((void**)&d_I, arraysize_bytes);

      cudaMemcpy(d_A, A, arraysize_bytes, cudaMemcpyHostToDevice);

      dim3 blockDim(5, 5);
      dim3 gridDim((array_size + blockDim.x - 1) / blockDim.x, (array_size + blockDim.y - 1)
/ blockDim.y);
```

```
    // Launch the GSGE kernel
    GSGE<<<gridDim, blockDim>>>(d_A, d_A_T, d_AAT, array_size);
    cudaMemcpy(A_T, d_A_T, arraysize_bytes, cudaMemcpyDeviceToHost);

    cout << "A_T matrix is" << endl;
    for (int i = 0; i < array_size; i++) {
    for (int j = 0; j < array_size; j++) {
          cout << cuCreal(A_T[i * array_size + j]) << "+" << cuCimag(A_T[i * array_size +
j]) << "i ";
    }
    cout << endl;
    }

    cudaMemcpy(AAT, d_AAT, arraysize_bytes, cudaMemcpyDeviceToHost);

    // Print results or perform other actions as needed
    cout << "AAT matrix is" << endl;
    for (int i = 0; i < array_size; i++) {
    for (int j = 0; j < array_size; j++) {
          cout << cuCreal(AAT[i * array_size + j]) << "+" << cuCimag(AAT[i * array_size +
j]) << "i ";
    }
    cout << endl;
    }

    cudaMemcpy(d_A_T, A_T, arraysize_bytes, cudaMemcpyHostToDevice);
    orthochecker<<<gridDim, blockDim>>>(d_A_T, d_U_T, d_I, array_size);
    cudaMemcpy(I, d_I, arraysize_bytes, cudaMemcpyDeviceToHost);

    cout << "I matrix is" << endl;
    for (int i = 0; i < array_size; i++) {
    for (int j = 0; j < array_size; j++) {
          cout << cuCreal(I[i * array_size + j]) << "+" << cuCimag(I[i * array_size + j])
<< "i ";
    }
    cout << endl;
    }

    cudaMemcpy(d_A_T, A_T, arraysize_bytes, cudaMemcpyHostToDevice);

    gaussElimination<<<gridDim, blockDim>>>(d_A_T, d_determinant, array_size);
    cudaMemcpy(A_T, d_A_T, arraysize_bytes, cudaMemcpyDeviceToHost);
```

```
	cudaMemcpy(&determinant, d_determinant, sizeof(cuDoubleComplex),
cudaMemcpyDeviceToHost);

	cout << "Row echelon form of the given matrix is" << endl;
	for (int i = 0; i < array_size; i++) {
	for (int j = 0; j < array_size; j++) {
		cout << cuCreal(A_T[i * array_size + j]) << "+" << cuCimag(A_T[i * array_size +
j]) << "i ";
	}
	cout << endl;
	}

	cout << "Determinant: " << cuCreal(determinant) << "+" << cuCimag(determinant) << "i"
<< endl;
	cout<< "Norm of determinant is "<<cuCabs(determinant);
	// Cleanup
				cudaFree(d_A_T);
	cudaFree(d_A);
	cudaFree(d_AAT);
	cudaFree(d_U_T);
	cudaFree(d_I);

	return 0;
}
```

Output:
A_T matrix is
0.777004+-0.61154i 0.135582+-0.0173885i 0.0250963+-0.0544912i
-0.0200641+0.0644345i -0.016832+-0.430534i 0.663148+-0.608307i
-0.0592555+0.119241i 0.828206+-0.331281i -0.385756+-0.194398i

AAT matrix is
1.44661+0i 0.818554+0.602687i 1.04236+0.7164i
-1.11022e-16+0i 1.44657+-5.9158e-17i 0.720368+0.483097i
5.52871e-17+-3.70769e-17i 0+5.55112e-17i 0.0851503+9.73391e-18i

I matrix is
1+2.61816e-18i -2.08167e-17+4.85723e-17i 4.09828e-17+-8.67362e-17i
-2.08167e-17+-5.20417e-17i 1+-3.69975e-18i -1.11022e-16+-2.22045e-16i
4.09828e-17+7.97973e-17i -1.11022e-16+2.22045e-16i 1+5.2774e-18i

Row echelon form of the given matrix is
0.777004+-0.61154i 0.135582+-0.0173885i 0.0250963+-0.0544912i
1.38778e-17+-1.38778e-17i 0.843699+-0.34122i -0.385846+-0.202476i

1.42816e-17+-5.28976e-18i -4.51028e-17+-5.55112e-17i 0.818901+-0.751179i

Determinant: 0.220777+0.975324i

Norm of determinant is 1
real 0m2.233s
user 0m0.019s
sys 0m2.008s

The code is identical to the code for real entries, only that the entries are complex and cuComplex.h library has been used to do manipulation of complex numbers.

Complex Parallel Gauss Elimination:
The below code is the same as for real entries just the entries here are complex.

```
#include <iostream>
#include <cmath>
#include <cuda_runtime.h>
#include <cuComplex.h>

const int array_size = 3;

using namespace std;

// Complex number structure
typedef cuDoubleComplex cuDoubleComplex;

// CUDA kernel for Gaussian elimination
__global__ void gaussElimination(cuDoubleComplex* A, cuDoubleComplex* determinant, int n) {
      int tid_x = threadIdx.x + blockDim.x * blockIdx.x;
      int tid_y = threadIdx.y + blockDim.y * blockIdx.y;

      int index = 0;

      // Perform Gaussian elimination with partial pivoting
      for (int i = 0; i < n; i++) {
      // Partial Pivoting: Find the pivot row with the maximum absolute value
      int pivotRow = i;
      double maxVal = cuCabs(A[i * n + i]);
      for (int k = i + 1; k < n; k++) {
            double val = cuCabs(A[k * n + i]);
            if (val > maxVal) {
                  maxVal = val;
                  pivotRow = k;
```

```
            }
      }

      // Swap the current row with the pivot row
      if (i != pivotRow) {
            index++;
            // Use thread index for parallelization
            cuDoubleComplex temp_A = A[i * n + tid_y];
            A[i * n + tid_y] = A[pivotRow * n + tid_y];
            A[pivotRow * n + tid_y] = temp_A;
      }

      __syncthreads();

      // Eliminate other elements in the current column
      for (int j = i + 1; j < n; j++) {
            cuDoubleComplex factor_A = cuCdiv(A[j * n + i], A[i * n + i]);
            A[j * n + tid_y] = cuCsub(A[j * n + tid_y], cuCmul(A[i * n + tid_y], factor_A));
      }

      __syncthreads();
      }
  if (tid_x == 0 && tid_y == 0) {
      cuDoubleComplex deter = make_cuDoubleComplex(1.0, 0.0);
      for (int i = 0; i < n; i++) {
deter = cuCmul(deter, A[i * n + i]);
      }
      *determinant = cuCmul(deter, make_cuDoubleComplex(pow(-1, index), 0.0));
      }
}

int main() {
      cuDoubleComplex A[array_size * array_size]={
      make_cuDoubleComplex(2.00, 3.00), make_cuDoubleComplex(-1.00, 2.00),
make_cuDoubleComplex(4.00, 1.00),
      make_cuDoubleComplex(0.00, 0.00), make_cuDoubleComplex(0.00, 5.00),
make_cuDoubleComplex(-3.00, -0.00),
      make_cuDoubleComplex(1.00, -1.00), make_cuDoubleComplex(2.00, 0.00),
make_cuDoubleComplex(-7.00, 2.00)
      };
int arraysize_bytes = array_size * array_size * sizeof(cuDoubleComplex);
      // Device variables
      cuDoubleComplex *d_A, *d_determinant;
```

```
cout << "input  matrix is" << endl;
      for (int i = 0; i < array_size; i++) {
      for (int j = 0; j < array_size; j++) {
            cout << cuCreal(A[i * array_size + j]) << "+" << cuCimag(A[i * array_size + j]) <<
"i ";
      }
      cout << endl;
      }
      // Allocate memory on the GPU
      cudaMalloc((void**)&d_A,arraysize_bytes);
      cudaMalloc((void**)&d_determinant, sizeof(cuDoubleComplex));

      // Copy data from host to device
      cudaMemcpy(d_A, A, sizeof(cuDoubleComplex) * array_size * array_size,
cudaMemcpyHostToDevice);

      // Define thread block and grid dimensions
      dim3 blockDim(1, array_size);  // 1D block with dimensions (1, array_size)
      dim3 gridDim(1, 1);  // 2D grid with dimensions (1, 1)

      // Invoke the kernel
      gaussElimination<<<gridDim, blockDim>>>(d_A, d_determinant, array_size);

      // Copy the result back to the host
      cuDoubleComplex h_determinant;
      cudaMemcpy(A, d_A, arraysize_bytes, cudaMemcpyDeviceToHost);
      cudaMemcpy(&h_determinant, d_determinant, sizeof(cuDoubleComplex),
cudaMemcpyDeviceToHost);
 cout << "Row echelon form of the given matrix is" << endl;
      for (int i = 0; i < array_size; i++) {
      for (int j = 0; j < array_size; j++) {
            cout << cuCreal(A[i * array_size + j]) << "+" << cuCimag(A[i * array_size + j]) <<
"i ";
      }
      cout << endl;
      }
      // Print the result

 cout << "Determinant: " << cuCreal(h_determinant) << " + " << cuCimag(h_determinant) << "i"
<< endl;

      // Free allocated memory on the GPU
```

```
    cudaFree(d_A);
    cudaFree(d_determinant);

    return 0;
}
```

OUTPUT:
input matrix is
2+3i -1+2i 4+1i
0+0i 0+5i -3+-0i
1+-1i 2+0i -7+2i
Row echelon form of the given matrix is
2+3i -1+2i 4+1i
0+0i 0+5i -3+-0i
0+0i 0+2.77556e-17i -7.21538+2.92308i
Determinant: 79 + -116i
real   0m2.159s
user   0m0.017s
sys   0m2.012s