```
!pip install git+https://github.com/afnan47/cuda.git
%load_ext nvcc_plugin
```

```
%%writefile matrix_multiplication.cu
#include <iostream>
using namespace std;
__global__ void multiply(int* A,int* B,int* C,int size)
{
    int row=blockIdx.y*blockDim.y+threadIdx.y;
    int col=blockIdx.x*blockDim.x+threadIdx.x;

    if(row<size && col<size)
    {
        int sum=0;
        for(int i=0;i<size;i++)
        {
            sum+=A[row*size+i]*B[i*size+col];
        }
        C[row * size +col]=sum;
}
}
void initialize(int* matrix,int size){
    for(int i=0;i<size*size;i++){
        matrix[i]=rand()%10;
    }
    }
```

```
void print(int* matrix,int size){
    for(int row=0;row<size;row++){
        for(int col=0;col<size;col++){
            cout<<matrix[row*size+col]<<" ";
        }
        cout<<endl;
    }
    cout<<endl;
}
int main()
{
    int N=2;
    size_t matrixBytes = N*N*sizeof(int);
    int* A= new int[N*N];
    int* B=new int[N*N];
    int* C= new int[N*N];
    initialize(A,N);
    initialize(B,N);
    cout<<"matrix of a:"<<endl;
    print(A,N);
    cout<<"matrix of b:"<<endl;
    print(B,N);
    int *d_A,*d_B,*d_C;
    cudaError_t err;
    err=cudaMalloc(&d_A,matrixBytes);
    if(err!=cudaSuccess)
    {
        cout<<"cuda malloc failed for a "<<CudaGetErrorString(err)endl;
        return -1;}

    err=cudaMalloc(&d_B,matrixBytes);
    if(err!=cudaSuccess)
    {
        cout<<"cuda malloc failed for b "<<CudaGetErrorString(err)endl;
```

```
        return -1;}
    err=cudaMalloc(&d_C,matrixBytes);
    if(err!=cudaSuccess)
    {
        cout<<"cuda malloc failed for c"<<CudaGetErrorString(err)endl;
        return -1;}
    err = cudaMemcpy(d_A,A,matrixBytes,cudaMemcpyHostToDevice);
    if(err!=cudaSuccess)
    {
        cout<<"cuda memcpy failed for a "<<CudaGetErrorString(err)endl;
        return -1;}
     err = cudaMemcpy(d_B,B,matrixBytes,cudaMemcpyHostToDevice);
    if(err!=cudaSuccess)
    {
        cout<<"cuda memcpy failed for b "<<CudaGetErrorString(err)endl;
        return -1;}
    dim3 threads(2,2);
    dim3 blocks(N +threads.x -1)/threads.x,(N+threads.y-1)/threads.y;
    multiply<<<blocks,threads>>>(d_A,d_B,d_C,N);
    cudaDeviceSynchronize();
    err=cudaGetLastError();
    if(err!=cudaSuccess)
    {
        cout<<"kernel launch failed "<<CudaGetErrorString(err)endl;
        return -1;}
    err=cudaMemcpy(C,d_C,matrixBytes,cudaMemcpyDeviceToHost);
    if(err!=cudaSuccess)
    {
        cout<<"cuda memcpy failed for c "<<CudaGetErrorString(err)endl;
        return -1;}

    cout<<"multiplication of matric is A and B:"<<endl;
    print(C,N);
    delete[] A;
    delete[] B;
    delete[] C;
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    return 0;
}
```

```
!nvcc -arch=sm_75 -o matrix_multiplication matrix_multiplication.cu
```

```
!./matrix_multiplication
```

```
%%writefile vector_add.cu
#include <iostream>

using namespace std;
```

```
__global__
void add(int* A,int* B,int* C,int size)
{
  int tid=blockIdx.x*blockDim.x+threadIdx.x;
  if(tid<size){
    C[tid] = A[tid]+B[tid];
  }
}
```

```
void initializer(int* vector,int size){
  for(int i=0;i<size;i++){
    vector[i]=rand() % 10;
  }
}
void print(int* vector,int size){
  for(int i=0;i<size;i++){
    cout<<vector[i]<<" ";
  }
  cout<<endl;
}
int main(){
  int N=4;
  int* A =new int[N];
  int* B =new int[N];
  int* C =new int[N];
  initializer(A,N);
  initializer(B,N);
  cout<<"Vector A";
  print(A,N);
  cout<<"Vector B";
  print(B,N);
```

```
  int *d_A,*d_B,*d_C;
  size_t bytes =N*sizeof(int);
  cudaMalloc(&d_A,bytes);
  cudaMalloc(&d_B,bytes);
  cudaMalloc(&d_C,bytes);
  cudaMemcpy(d_A,A,bytes,cudaMemcpyHostToDevice);
  cudaMemcpy(d_B,B,bytes,cudaMemcpyHostToDevice);
```

```
  int threadsPerBlock=256;
  int blocksPerGrid=(N+threadsPerBlock-1)/threadsPerBlock;
  add<<<blocksPerGrid,threadsPerBlock>>>(d_A,d_B,d_C,N);
  cudaDeviceSynchronize();
```

```
  cudaMemcpy(C,d_C,bytes,cudaMemcpyDeviceToHost);
  cudaError_t err=cudaGetLastError();
  if(err!=cudaSuccess){
    cout<<"Error: "<< cudaGetErorString(err) <<endl;
    return -1;
  }
  cout<<"Vector C";
  print(C,N);
  delete[] A;
  delete[] B;
  delete[] C;
```

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

```
  return 0;
}
```

```
!nvcc -arch=sm_75 vector_add.cu -o vector_add
!./vector_add
```

```cpp
#include<iostream>
#include<omp.h>
using namespace std;
void BubbleSort(int arr[],int n){
    for(int i=0;i<n-1;i++){
        for(int j=0;j<n-i-1;j++){
            if(arr[j]>arr[j+1]){
                swap(arr[j],arr[j+1]);
            }
        }
    }
}

void pBubble(int arr[],int n){
    for(int i=0;i<n;i++){
        #pragma omp for
        for(int j=1;j<n;j+=2){
            if(arr[j]<arr[j-1]){
                swap(arr[j],arr[j-1]);
            }
        }

        #pragma omp barrrier

        //even phase

        #pragma omp for
        for(int j=2;j<n;j+=2){
            if(arr[j]<arr[j-1]){
                swap(arr[j],arr[j-1]);
            }
        }
        #pragma omp barrier
    }
}
```

```cpp
void print(int arr[],int n){
    for(int i=0;i<n;i++){
        cout<<arr[i]<<" ";
    }
    cout<<endl;
}
int main(){
    int n=10;
    int arr[]={5,4,6,8,7,3,10,2,19};
    double start_time,end_time;

    for(int i=0,j=n;i<n;i++,j--){
        arr[i]=j;
    }

    start_time=omp_get_wtime();
    BubbleSort(arr,n);
    end_time=omp_get_wtime();
    cout<<"Sequential bubble sort took : "<<end_time-start_time<<" seconds \n";
    print(arr,n);

    for(int i=0,j=n;i<n;i++,j--){
        arr[i]=j;
    }
```

```
    start_time=omp_get_wtime();
    #pragma omp parallel
    {
        pBubble(arr,n);
    }
    end_time=omp_get_wtime();
    cout<<"Parallel bubble sort took : "<<end_time-start_time<<" seconds \n";
    print(arr,n);
}
```

```cpp
#include<iostream>
#include<omp.h>
using namespace std;
void merge(int arr[],int low,int mid,int high){
    int n1=mid-low+1;
    int n2=high-mid;
    int left[n1],right[n2];
    for(int i=0;i<n1;i++){
        left[i]=arr[low+i];
    }
    for(int j=0;j<n2;j++){
        right[j]=arr[mid+1+j];
    }
    int i=0,j=0,k=low;
    while(i<n1 && j<n2){
        if(left[i]<=right[j]){
            arr[k]=left[i];
            k++;
            i++;
        }
        else{
            arr[k]=right[j];
            k++;
            j++;
        }
    }
    while(i<n1){
        arr[k]=left[i];
        k++;
        i++;
    }
    while(j<n2){
        arr[k]=right[j];
        k++;
        j++;
    }
}
void mergeSort(int arr[],int low,int high){
    if(low<high){
        int mid=(low+high)/2;
        mergeSort(arr,low,mid);
        mergeSort(arr,mid+1,high);
        merge(arr,low,mid,high);
    }
}
void parallelMergeSort(int arr[],int low,int high){
    if(low<high){
        int mid=(low+high)/2;
        #pragma omp parallel sections
        {
            #pragma omp section
            parallelMergeSort(arr,low,mid);

            #pragma omp section
            parallelMergeSort(arr,mid+1,high);
        }

        merge(arr,low,mid,high);
    }
}
void print(int arr[],int n){
```

```
    for(int i=0;i<n;i++){
        cout<<arr[i]<<" ";
    }
    cout<<endl;
}
int main(){
    int n=10;
    int arr[]={3,4,2,1,6,5,8,9,7,10};
    double start_time,end_time;

    for(int i=0,j=n;i<n;i++,j--){
        arr[i]=j;
    }

    start_time=omp_get_wtime();
    mergeSort(arr,0,n-1);
    end_time=omp_get_wtime();
    cout<<"The tme taken by sequential merge sort is : "<<end_time-start_time<<" seconds \n";
    print(arr,n);
    for(int i=0,j=n;i<n;i++,j--){
        arr[i]=j;
    }

    start_time=omp_get_wtime();
    parallelMergeSort(arr,0,n-1);
    end_time=omp_get_wtime();
    cout<<"The time taken by parallel merge sort is : "<<end_time-start_time<<" seconds \n";
    print(arr,n);
}
```