```
!nvcc --version
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
%load_ext nvcc_plugin
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Sun_Jul_28_19:07:16_PDT_2019
Cuda compilation tools, release 10.1, V10.1.243
Collecting git+git://github.com/andreinechaev/nvcc4jupyter.git
  Cloning git://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-5wznz
  Running command git clone -q git://github.com/andreinechaev/nvcc4jupyter.git /tmp/p
Building wheels for collected packages: NVCCPlugin
  Building wheel for NVCCPlugin (setup.py) ... done
  Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-cp36-none-any.whl size=4307
  Stored in directory: /tmp/pip-ephem-wheel-cache-hvw4_kan/wheels/10/c2/05/ca241da37b
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
created output directory at /content/src
Out bin /content/result.out
```

```
%%cu
#include<iostream>
#include<math.h>

#define n 256

using namespace std;

__global__ void minimum(int *input) {
    int tid = threadIdx.x;
    int step_size = 1;
    int number_of_threads = blockDim.x;

   // printf("No of threads = %d", number_of_threads);

    while(number_of_threads>0) {
        if(tid < number_of_threads) {
            int first = tid*step_size*2;
            int second = first + step_size;
            if(input[second] < input[first])
              input[first] = input[second];

            //printf("First = %d Second = %d\n", input[first], input[second]);
        }
        step_size <<= 1;
        number_of_threads >>= 1;
    }
}

__global__ void maximum(int *input) {
    int tid = threadIdx.x;
```

```
    int step_size = 1;
    int number_of_threads = blockDim.x;

    while(number_of_threads>0) {
        if(tid < number_of_threads) {
            int first = tid*step_size*2;
            int second = first + step_size;
            if(input[second] > input[first])
              input[first] = input[second];
        }
        step_size <<= 1;
        number_of_threads >>= 1;
    }
}

__global__ void sum(int *input) {
    const int tid = threadIdx.x;
    int step_size = 1;
    int number_of_threads = blockDim.x;

    while(number_of_threads > 0) {
        if(tid < number_of_threads) {
            int first = tid * step_size * 2;
            int second = first + step_size;

            input[first] += input[second];
        }
        step_size <<= 1;
        number_of_threads >>= 1;

    }
}

__global__ void mean_diff_sq(float *input, float mean) {
    input[threadIdx.x] -= mean;
    input[threadIdx.x] *= input[threadIdx.x];
}

__global__ void sum_floats(float *input) {
    int tid = threadIdx.x;
    int step_size = 1;
    int number_of_threads = blockDim.x;

    while(number_of_threads > 0) {
        if(tid < number_of_threads) {
            int first = tid * step_size * 2;
            int second = first + step_size;

            input[first] += input[second];
        }
        step_size <<= 1;
        number_of_threads >>= 1;
```

```
    }
}

  void random_ints(int *input, int size) {
    for(int i=0; i<size; i++)  {
        input[i] = rand()%(size);
        cout<<input[i]<<"  ";
    }
    cout<<endl;

}

void copy_int_to_float(float *dest, int *src, int size){
    for(int i=0; i<size; i++)
        dest[i] = float(src[i]);
}



int main() {

    int size = n*sizeof(int); //calculate no. of bytes for array
    int *arr;
    int *arr_d, result;

    arr = (int *)malloc(size);
    random_ints(arr, n);


    /* cudaMalloc() allocates memory from Global memory on GPU */
    cudaMalloc((void **)&arr_d, size);


    //Min Value

    /* cudaMemcpy() copies the contents from destination to source. Here destinati
    cudaMemcpy(arr_d, arr, size, cudaMemcpyHostToDevice);

    /* call to kernel. Here 1 is number of blocks, n/2 is the number of threads pe
    minimum<<<1,n/2>>>(arr_d);

    cudaMemcpy(&result, arr_d, sizeof(int), cudaMemcpyDeviceToHost);

    cout<<"The minimum element is "<<result<<endl;


    //Max Value
    cudaMemcpy(arr_d, arr, size, cudaMemcpyHostToDevice);

    maximum<<<1,n/2>>>(arr_d);

    cudaMemcpy(&result, arr_d, sizeof(int), cudaMemcpyDeviceToHost);
```

```
    cudaMemcpy(&result, arr_d, sizeof(int), cudaMemcpyDeviceToHost);

    cout<<"The maximum element is "<<result<<endl;

    //Sum
    cudaMemcpy(arr_d, arr, size, cudaMemcpyHostToDevice);

    sum<<<1,n/2>>>(arr_d);

    cudaMemcpy(&result, arr_d, sizeof(int), cudaMemcpyDeviceToHost);

    cout<<"The sum is "<<result<<endl;


    //Average
    float mean = float(result)/n;
    cout<<"The mean is "<<mean<<endl;

    //Variace
    float *arr_float;
    float *arr_std, stdValue;

    arr_float = (float *)malloc(n*sizeof(float));
    cudaMalloc((void **)&arr_std, n*sizeof(float));

    copy_int_to_float(arr_float, arr, n);

    cudaMemcpy(arr_std, arr_float, n*sizeof(float), cudaMemcpyHostToDevice);

    mean_diff_sq <<<1,n>>>(arr_std, mean);
    sum_floats<<<1,n/2>>>(arr_std);

    cudaMemcpy(&stdValue, arr_std, sizeof(float), cudaMemcpyDeviceToHost);


    stdValue = stdValue / n;
    cout<<"The variance is "<<stdValue<<endl;

    //Standard Deviation
    stdValue = sqrt(stdValue);
    cout<<"The standard deviation is "<<stdValue<<endl;

    cudaFree(arr_d);

    return 0;
}
```

```
103  198  105  115  81  255  74  236  41  205  186  171  242  251  227  70  124  194
The minimum element is 0
The maximum element is 255
The sum is 32454
The mean is 126.773
The variance is 5804.25
The standard deviation is 76.1857
```