Egyenlőszárú Polygomok számítása GPU-val lépések:

0. lépés: Megadjuk a változókat és Tanár Úr kódját felhasználjuk.

```
//number of tests
int n = 5;
const int max_n = 50000000;
//number of sides
const int sides = 9;
// Input array
srand(time(0));
// Get the platform and device information
cl_platform_id platform;
cl_device_id device;
cl_uint num_platforms, num_devices;
cl_int err;
err = clGetPlatformIDs(1, & platform, & num_platforms);
if (err != CL_SUCCESS) {
    printf("Error getting platform ID: %d\n", err);
    return -1;
}
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, & device, & num_devices);
if (err != CL_SUCCESS) {
    printf("Error getting device ID: %d\n", err);
    return -1;
}

// Create the OpenCL context and command queue
cl_context context = clCreateContext(NULL, 1, & device, NULL, NULL, & err);
if (err != CL_SUCCESS) {
    printf("Error creating context: %d\n", err);
    return -1;
}
cl_command_queue queue = clCreateCommandQueue(context, device, 0, & err);
if (err != CL_SUCCESS) {
    printf("Error creating command queue: %d\n", err);
    return -1;
}

// Create the kernel program and build it
const char * kernel_code = load_kernel_source("kernels/szamitas.cl", & err);
if (err != 0) {
    printf("Source code loading error!\n");
    return 0;
}
cl_program program = clCreateProgramWithSource(context, 1, & kernel_code, NULL, & err);
if (err != CL_SUCCESS) {
    printf("Error creating program: %d\n", err);
    return -1;
}
err = clBuildProgram(program, 1, & device, NULL, NULL, NULL);
if (err != CL_SUCCESS) {
    printf("Error building program: %d\n", err);
    return -1;
}
```

1. Lépés: Létre kell hoznunk a buffert és beleírni

```
// Create the buffer for the input array
cl_mem length_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, n * sizeof(float), NULL, & err);
cl_mem area_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, n * sizeof(float), NULL, & err);
cl_mem inradius_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, n * sizeof(float), NULL, & err);
cl_mem circumradius_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, n * sizeof(float), NULL, & err);
cl_mem perimeter_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, n * sizeof(float), NULL, & err);
cl_mem interior_angle_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, n * sizeof(float), NULL, & err);
cl_mem exterior_angle_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, n * sizeof(float), NULL, & err);
if (err != CL_SUCCESS) {
    printf("Error creating buffer: %d\n", err);
    return -1;
}
// Write to the buffer
err = clEnqueueWriteBuffer(queue, length_buffer, CL_TRUE, 0, n * sizeof(float), side_length, 0, NULL, NULL);
if (err != CL_SUCCESS) {
    printf("Error writing to buffer: %d\n", err);
    return -1;
}
```

2. Lépés: Ezek után létrehozzuk a kernelt és beállítjuk:

```
// Create the kernel and set the arguments
cl_kernel kernel = clCreateKernel(program, "pentagon_area", & err);
if (err != CL_SUCCESS) {
    printf("Error creating kernel: %d\n", err);
    return -1;
}
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), & length_buffer);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), & area_buffer);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), & inradius_buffer);
err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), & circumradius_buffer);
err |= clSetKernelArg(kernel, 4, sizeof(cl_mem), & perimeter_buffer);
err |= clSetKernelArg(kernel, 5, sizeof(cl_mem), & interior_angle_buffer);
err |= clSetKernelArg(kernel, 6, sizeof(cl_mem), & exterior_angle_buffer);
err |= clSetKernelArg(kernel, 7, sizeof(int), & sides);
if (err != CL_SUCCESS) {
    printf("Error setting kernel arguments: %d\n", err);
    return -1;
}

size_t global_size = n;
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, & global_size, NULL, 0, NULL, NULL);
if (err != CL_SUCCESS) {
    printf("Error enqueuing kernel: %d\n", err);
    return -1;
}
```

3. Lépés: Beolvassuk az adatokat és felszabadítjuk a memóriát

```
    // Read the data
    err = clEnqueueReadBuffer(queue, area_buffer, CL_TRUE, 0, n * sizeof(float), area, 0, NULL, NULL);
    err |= clEnqueueReadBuffer(queue, inradius_buffer, CL_TRUE, 0, n * sizeof(float), inradius, 0, NULL, NULL);
    err |= clEnqueueReadBuffer(queue, circumradius_buffer, CL_TRUE, 0, n * sizeof(float), circumradius, 0, NULL, NULL);
    err |= clEnqueueReadBuffer(queue, perimeter_buffer, CL_TRUE, 0, n * sizeof(float), perimeter, 0, NULL, NULL);
    err |= clEnqueueReadBuffer(queue, interior_angle_buffer, CL_TRUE, 0, n * sizeof(float), interior_angle, 0, NULL, NULL);
    err |= clEnqueueReadBuffer(queue, exterior_angle_buffer, CL_TRUE, 0, n * sizeof(float), exterior_angle, 0, NULL, NULL);
    if (err != CL_SUCCESS) {
        printf("Error reading from buffer: %d\n", err);
        return -1;
    }

    clReleaseMemObject(length_buffer);
    clReleaseMemObject(area_buffer);
    clReleaseMemObject(inradius_buffer);
    clReleaseMemObject(circumradius_buffer);
    clReleaseMemObject(perimeter_buffer);
    clReleaseMemObject(interior_angle_buffer);
    clReleaseMemObject(exterior_angle_buffer);
    clReleaseKernel(kernel);


    end = clock();
    printf("\n\nN: %d\t\ttime: %lf\n", n, ((double)(end - start)) / CLOCKS_PER_SEC);
    free(area);
    free(inradius);
    free(circumradius);
    free(perimeter);
    free(interior_angle);
    free(exterior_angle);
    free(side_length);
    n *= 10;
}
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
return 0;
```