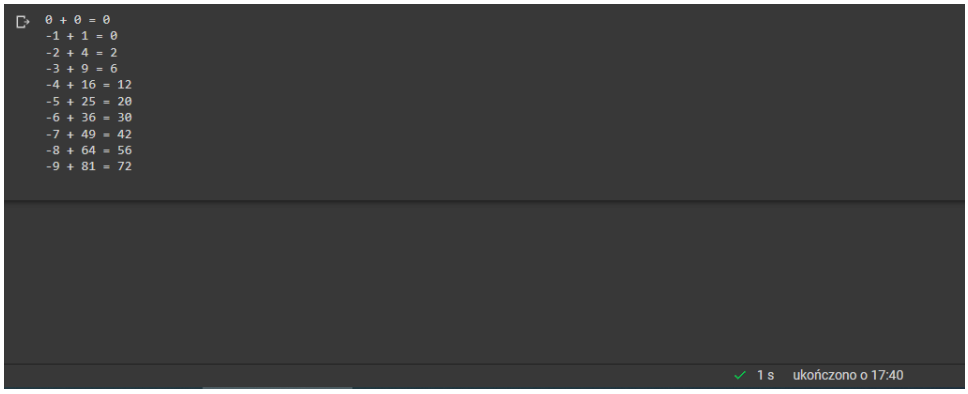
Lab 9 - porównanie dwóch programów, między CPU, a GPU

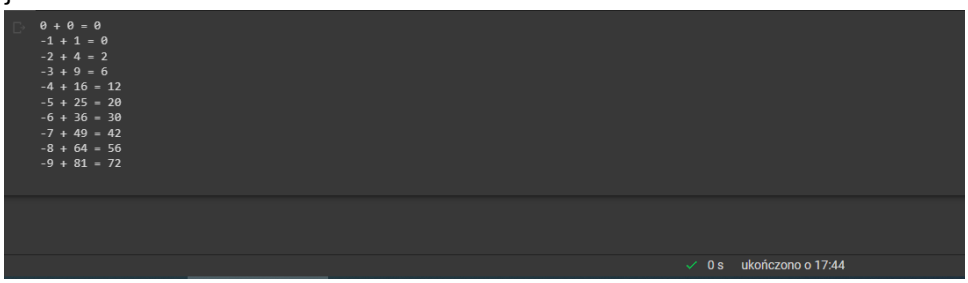
GPU - sumowanie wektorów

```
%%cu
#include <stdio.h>
#define N 10
__global__ void add( int *a, int *b, int *c ) {
int tid = blockIdx.x; // dodawanie na danych znajdujących się pod indeksem
if (tid < N)
c[tid] = a[tid] + b[tid];
}
int main( void ) {
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;
 // alokacja pamięci na GPU
cudaMalloc( (void**)&dev_a, N * sizeof(int) );
cudaMalloc( (void**)&dev_b, N * sizeof(int) );
cudaMalloc( (void**)&dev_c, N * sizeof(int) );
 // zapelnienie tablic a i b na CPU
 for (int i=0; i<N; i++) {
 a[i] = -i;
 b[i] = i * i;
 }
// kopiowanie tablic a i b do GPU
cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );
 add<<<N,1>>>( dev_a, dev_b, dev_c );
 // kopiowanie tablicy c z GPU do CPU
cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );
 // wyświetlenie wynikow
 for (int i=0; i<N; i++) {
 printf( "%d + %d = %d\n", a[i], b[i], c[i] );
 }
 // zwolnienie pamieci alokowanej na CPU
 cudaFree( dev_a );
 cudaFree( dev_b );
 cudaFree( dev_c );
 return 0;
}
```



CPU – sumowanie wektorów

```
%%cu
#include <stdio.h>
#define N 10
void add( int *a, int *b, int *c ) {
int tid = 0; // to jest CPU nr zero, a więc zaczynamy od zera
while (tid < N) {
c[tid] = a[tid] + b[tid];
tid += 1; // mamy tylko jeden CPU, a więc zwiększamy o jeden
}
}
int main( void ) {
int a[N], b[N], c[N];
// zapelnianie tablic a i b za pomocą CPU
for (int i=0; i<N; i++) {
a[i] = -i;
b[i] = i * i;
}
add( a, b, c );
// wyswietlenie wynikow
for (int i=0; i<N; i++) {
printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}
return 0;
}
```



CPU – fraktal Julii

```
#cpu
%%cu
#include <stdlib.h>
#include <stdio.h>
bool Julia( float x, float y, float maxX_2, float maxY_2 )
  {
    float z_r = 0.8 * (float) (maxX_2 - x) / maxX_2;
    float z_i = 0.8 * (float) (maxY_2 - y) / maxY_2;
    float c_r = -0.8;
    float c_i = 0.156;
    for( int i=1 ; i<100 ; i++ )
    {
    float tmp_r = z_r*z_r - z_i*z_i + c_r;
    float tmp_i = 2*z_r*z_i + c_i;
    z_r = tmp_r;
    z_i = tmp_i;
```

```
    if( sqrt( z_r*z_r + z_i*z_i ) > 1000 )
      return false;
    }
    return true;
  }
  int main(void)
  {
   const int dimx = 768, dimy = 768;
   int i, j;
   unsigned char * data = new unsigned char[dimx*dimy*3];
   for( int z=0 ; z<512 ; z++ ) // ustawienie ilości obliczeń punktów fraktalu Julii
   {
       for (j = 0; j < dimy; ++j)
       {
        for (i = 0; i < dimx; ++i)
        {
         if( Julia(i,j,dimx/2,dimy/2) == true )
         {
         data[3*j*dimx + 3*i + 0] = (unsigned char)255;  /* red */
         data[3*j*dimx + 3*i + 1] = (unsigned char)0;  /* green */
         data[3*j*dimx + 3*i + 2] = (unsigned char)0;  /* blue */
         }
         else
         {
         data[3*j*dimx + 3*i + 0] = (unsigned char)255;  /* red */
         data[3*j*dimx + 3*i + 1] = (unsigned char)255;  /* green */
         data[3*j*dimx + 3*i + 2] = (unsigned char)255;  /* blue */
         }
        }
       }
   }
   delete [] data;
   return 0;
  }
```

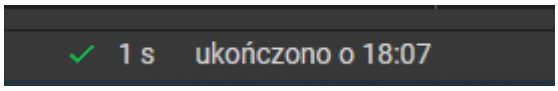
✓ 2 min 9 s ukończono o 18:05

GPU – Fraktal Julii

```
#gpu
%%cu
  #include <stdio.h>
  __device__ bool Julia( float x, float y, float maxX_2, float maxY_2 )
  {
    float z_r = 0.8 * (float) (maxX_2 - x) / maxX_2;
    float z_i = 0.8 * (float) (maxY_2 - y) / maxY_2;
    float c_r = -0.8;
```

```
    float c_i = 0.156;
    for( int i=1 ; i<100 ; i++ )
    {
    float tmp_r = z_r*z_r - z_i*z_i + c_r;
    float tmp_i = 2*z_r*z_i + c_i;
    z_r = tmp_r;
    z_i = tmp_i;
    if( sqrt( z_r*z_r + z_i*z_i ) > 1000 )
       return false;
    }
    return true;
}
__global__ void kernel( unsigned char * im, int dimx, int dimy )
{
    //int tid = blockIdx.y*gridDim.x + blockIdx.x;
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    tid *= 3;
    if( Julia((float)blockIdx.x, (float)threadIdx.x, (float)dimx/2, (float)dimy/2)==true )
    {
    im[tid] = 255;
    im[tid+1] = 0;
    im[tid+2] = 0;
    }
    else
    {
    im[tid] = 255;
    im[tid+1] = 255;
    im[tid+2] = 255;
    }
}
int main()
{
    int dimx=768, dimy=768;
    //on cpu
    unsigned char * im = (unsigned char*) malloc( 3*dimx*dimy );
    //on GPU
    unsigned char * im_dev;
    //allocate mem on GPU
    cudaMalloc( (void**)&im_dev, 3*dimx*dimy );
    //launch kernel.
    for( int z=0 ; z<512 ; z++ ) // ustawienie ilości obliczeń punktów fraktalu Julii
    {
        kernel<<<dimx,dimy>>>(im_dev, dimx, dimy);
    }
        cudaMemcpy( im, im_dev, 3*dimx*dimy, cudaMemcpyDeviceToHost );
```

```
        free( im );
        cudaFree( im_dev );
}
```



Wnioski:

Po kompilacji pierwszego programu, sumowaniu wektorów można stwierdzić, że program jest zbyt mało rozbudowany, ale dzięki temu, że jest niewielki i prosty do odczytu możemy zauważyć, że przy prostych zadaniach CPU jest bardziej wydajne od GPU. GPU jest w stanie się wykazać przy bardziej złożonych zadaniach, ewentualnie programach, które obliczają dużą ilość danych przy kompilacji. W drugim programie, którym jest Fraktal Julii widać dokładną różnicę w prędkości kompilacji między GPU, a CPU. Mamy widoczną przewagę układu GPU nad układem CPU. Więc dzięki takiemu porównaniu jesteśmy w stanie zrozumieć, dlaczego mamy aktualnie tak mocną popularyzację układów GPU nad układami CPU do wykonywania dużych, złożonych obliczeń w konkretnym czasie, jest to bardzo duża oszczędność zasobów obliczeniowych i czasowych. Poza programami, których użyłem jako przykład możemy porównać kopanie kryptowalut, gdzie używanie CPU do tego celu mija się z celem, ponieważ będzie wtedy zbyt przeciążony i tracimy nie tylko swój czas, ale też energię, która jest potrzebna do pracy takiego układu. Oczywiście musimy być jeszcze świadomi, że procesor graficzny udostępniony przez Google Colaboratory jest ograniczony wydajnościowo ze względu na dużą ilość użytkowników w konkretnym czasie, więc zdarza się tak, że cały układ GPU może w danym czasie nie działać, lub potrafi być znacznie wolniejszy od układu CPU, ale jest to tylko widoczne w przypadku bardzo dużej ilości użytkowników jednocześnie pracujących na stronie.