Aleksander Dawid – Uniwersytet Śląski 2012 rok

Programowanie procesorów graficznych NVIDIA (rdzenie CUDA)

Wykład nr 1

Wprowadzenie

Procesory graficzne GPU (*Graphics Processing Units*) stosowane są w kartach graficznych do przetwarzania grafiki komputerowej i wyświetlania jej na ekranie monitora

Motorem rozwoju GPU jest rynek gier komputerowych

Fotorealizm w czasie rzeczywistym

Cechy GPU

- Przetwarzanie równoległe
- Ilość rdzeni > 100 (1 CPU = 8 rdzeni)
- Przyspieszone operacje na macierzach.
- Wydajność rzędu Tfps
- Przetwarzanie wierzchołków i pixeli
- Przetwarzania światła
- Obsługa efektów fizycznych
- Obsługa biblioteki OpenGL i DirectX

Naukowe obliczenia numeryczne

3 najszybsze komputery świata korzystają z kart graficznych do zwiększenia swojej mocy obliczeniowej.

Tianhe 1a - Chiny
14,336 procesorów Xeon X5670
7,168 GPU Nvidia Tesla M2050
Maksymalna wydajność
2.507 petaFLOPS 10^{15} operacji zmiennoprzecinkowych na sekundę

TESLA (Fermi core)

· 448 CUDA Cores
· 515 Gigaflops (podwójna precyzja)
· 1 Teraflop (pojedyncza precyzja)

Obliczenia na GPU

Model Cg (C for graphics)

Obliczenia wykonywane były w ramach jednostek obliczeniowych pixel shader i vertex shader.
Wynik obliczeń nie dało się odczytać bezpośrednio w postaci numerycznej tylko jako zestaw pixeli na ekranie monitora.

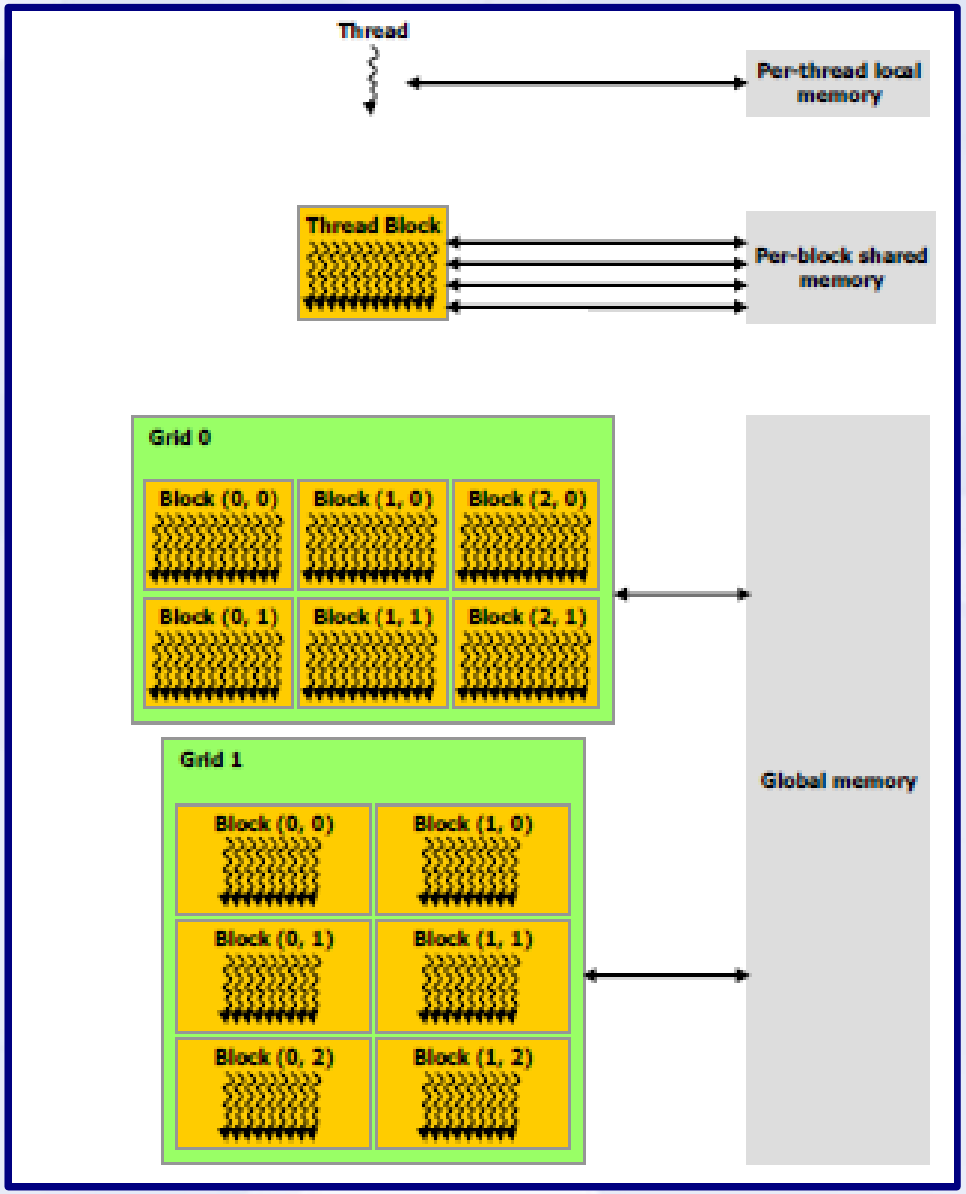
Model CUDA (Compute Unified Device Architecture)

Obliczenia wykonywane są na wszystkich dostępnych rdzeniach.
Wynik obliczeń można odebrać w postaci numerycznej (terminal)

Cechy CUDA

- Wielowątkowość
- Wielozadaniowość
- Programowanie w językach wysokiego poziomu

Architektura CUDA



Model pamięci

Pamięć lokalna dostępna tylko dla wątku (bardzo szybka)

Pamięć dzielona dostępna tylko dla wszystkich wątków w bloku (bardzo szybka)

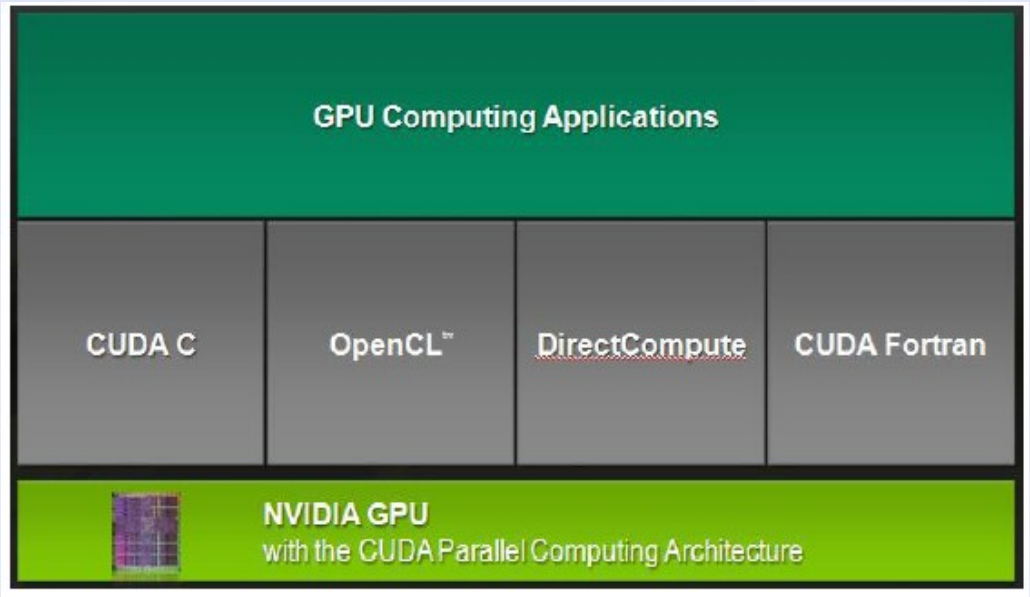
Pamięć globalna dostępna dla wszystkich wątków (raczej wolna).

Architektura CUDA

RAM CPU >= GPU

Większa ilość pamięci na karcie i w komputerze zwiększa prędkość obliczeń

Wspierane języki programowania.



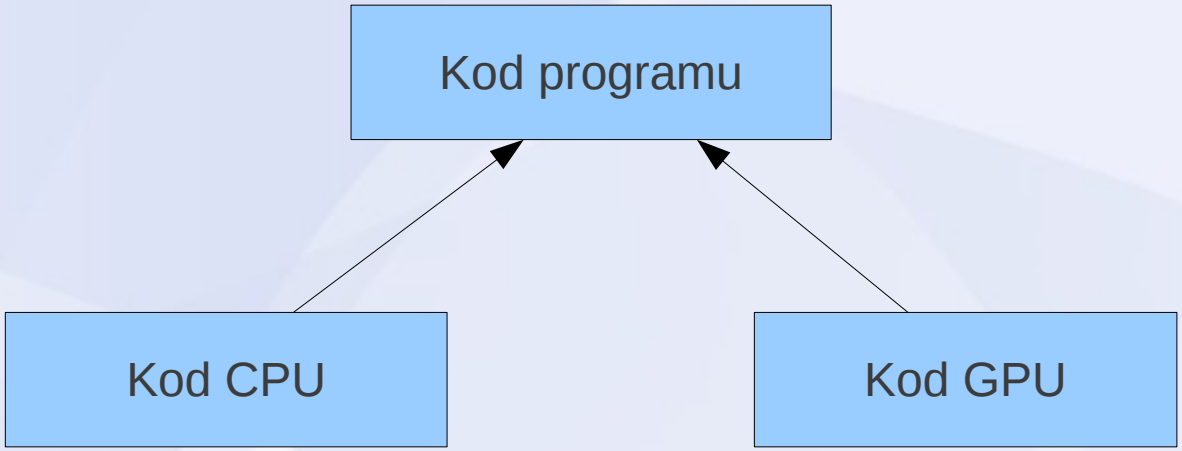
Dodatkowo istniej możliwość programowania CUDA w językach JAVA i Python

Programowanie CUDA C

Kompilator: **nvcc**

Dostępny jest w pakiecie CUDA toolkit dla różnych systemów operacyjnych takich jak Windows, Linux, Mac.

Pełne wsparcie dla standardu ANSI C z elementami C++
CUDA w wersji 1.0 dostępne jest już dla kart NVIDA GeForce 8600



Programowanie CUDA C

Kod CPU

Zadania

· Informacja o sprzęcie
· Inicjalizacja pamięci
· Operacje I/O CPU->GPU, GPU->CPU
· Zwalnianie pamięci

Program oddaje sterowanie do GPU i czeka na wynik obliczeń.
Wykonane może to być w sposób synchroniczny lub asynchroniczny.

Kod GPU

Zadania

· Sterowanie programem
· Pamięci lokalne i dzielone
· Obliczenia numeryczne
· Synchronizacja wątków

GPU rozwiązuje zadanie na wszystkich dostępnych rdzeniach równolegle. Kod programu musi być specjalnie urównoleglony.

Programowanie CUDA C

Wymagania systemowe

Windows XP/VISTA/7
CUDA developer driver, CUDA toolkit, MS Visual Studio C++

Linux: nowsze dystrybucje.
CUDA developer driver, CUDA toolkit, gcc4.4

Dodatkowo dla wszystkich systemów
GPUComputingSDK, w którym są przykłady programów CUDA.
Działanie tych programów jest gwarancją prawidłowej instalacji sterowników CUDA w systemie.

Uwaga !!! WINDOWS VISTA/7 resetuje karte po braku odpowiedzi w przeciągu 90 s. Warto ten czas przedłużyć gdy nie będziemy korzystać z grafiki.

Kod CPU (informacja o zainstalowanej karcie)

```
#include <stdio.h>
#include <cutil.h>
```

W pliku nagłówkowym cutil.h zawarte są informacje o wszystkich nagłówkach CUDA. Aby zastosować tą funkcje w systemi musi być dostępna biblioteka libcutil_x86_64.a (Linux64), która znajduje się w GPUComputingSDK.

```
void PrintDevicesInformation()
{
int deviceCount,nrdev,nMulProc;
cudaGetDeviceCount(&deviceCount);

printf("Number of CUDA devices: %d\n",deviceCount);

for(nrdev = 0; nrdev < deviceCount; nrdev++) {
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp, nrdev);
  printf("Device %d: %s\n", nrdev, deviceProp.name);
  printf("maxThreadsPerBlock: %d\n", deviceProp.maxThreadsPerBlock);
  nMulProc=deviceProp.multiProcessorCount;
  printf("multiProcessorCount: %d\n", nMulProc);
  printf("computeMode: %d\n", deviceProp.computeMode);
  printf("sharedMemPerBlock: %ld\n", deviceProp.sharedMemPerBlock);
  }
}
```

Kod CPU (informacja o zainstalowanej karcie)

```
int main(int argc, char* argv[])
{
  PrintDevicesInformation();
  return 0;
}
```

Kompilacja gcc4.4 UBUNTU

```
> nvcc program.cu -lcudart pathToLib/libcutil_x86_64.a -o program
```

Wynik działania

```
Number of CUDA devices: 1
Device 0: Quadro FX 5800
maxThreadsPerBlock: 512
multiProcessorCount: 30
computeMode: 0
sharedMemPerBlock: 16384
```

Często stosowane rozszerzenie .cu zarówno w Linux jak i w Windows

Kod CPU (CPU->GPU)

```
#define real float
real* R;
size_t size;

void CPUMemAlloc()
{
 size = 16*sizeof(real); // 16 liczb typu real
 R = (real*)malloc(size);
 memset(R,0,size);
}
```

Funkcja CPUMemAlloc alokuje pamięć dla 16 liczb typu real czyli float

```
real* d_R;

void GPUMemAlloc()
{
 cudaMalloc((void**)&d_R, size);
}
```

Funkcja GPUMemAlloc alokuje pamięć na karcie graficznej dla 16 liczb typu real

Kod CPU (CPU->GPU)

```
void CopyCPUToGPU()
{
 cudaMemcpy(d_R, R, size, cudaMemcpyHostToDevice);
}
```

Funkcja kopiuje size bloków pamięci z komputera na kartę graficzną
Host – komputer z CPU
Device – karta graficzna

```
void CopyGPUToCPU()
{
 cudaMemcpy(R, d_R, size, cudaMemcpyDeviceToHost);
}
```

Odwrotna operacja

W operacjach tych brakuje obsługi wyjątków, które mogą wystąpić z różnych przyczyn. Funkcja **`cudaGetLastError()`** podaje numer ostatniego błędu.

Kod CPU (CPU->GPU)

```
int main(int argc, char* argv[])
{
 CPUMemAlloc();
 GPUMemAlloc();
 R[0]=5;
 CopyCPUToGPU();

 CopyGPUToCPU();
 printf("R[0]=%f\n",R[0]);
 return 0;
}
```

Brakuje operacji wykonywanych na karcie graficznej

Musimy stworzyć tzw. Kernel, który będzie wykonywany na GPU.

Kod CPU (Inicjalizacja Kernela GPU)

```
int threadsPerBlock, blocksPerGrid, sharedSize;


void InitKernel()
{
 threadsPerBlock = 16; //Tyle samo wątków w bloku ile liczb mamy do
                       //przeliczenia
 blocksPerGrid = 1;
 sharedSize = size; //Szybka pamięć dzielona na 16 liczb real
}
```

Każda karta graficzna ma inną maksymalną ilość wątków w jednym bloku.
Wartość ta waha się między 128 a 1024. Tutaj blok został ustalony na 16 wątków.
Ilość bloków w macierzy obliczeniowej też zależy od karty, tutaj ustalamy 1 blok.

Teraz możemy już pisać program dla GPU

Kod GPU (Przemnożenie macierzy przez dwa)

```
__global__ void MultByTwo(real* R){
extern __shared__ real rS[];
unsigned int id = threadIdx.x;
R[id]=R[id]*2;
}
```

Główna procedura w GPU tworzona jest przez dyrektywę **__global__**. **__shared__** oznacza pamięć dostępną dla całego bloku.

```
int main(int argc, char* argv[])
{
 CPUMemAlloc();
 GPUMemAlloc();
 R[0]=5;
 CopyCPUToGPU();
 InitKernel();
 MultByTwo<<<blocksPerGrid, threadsPerBlock, sharedSize>>>(d_R);
 CopyGPUToCPU();
 printf("R[0]=%f\n",R[0]);
 return 0;
}
```

Przetwarzanie

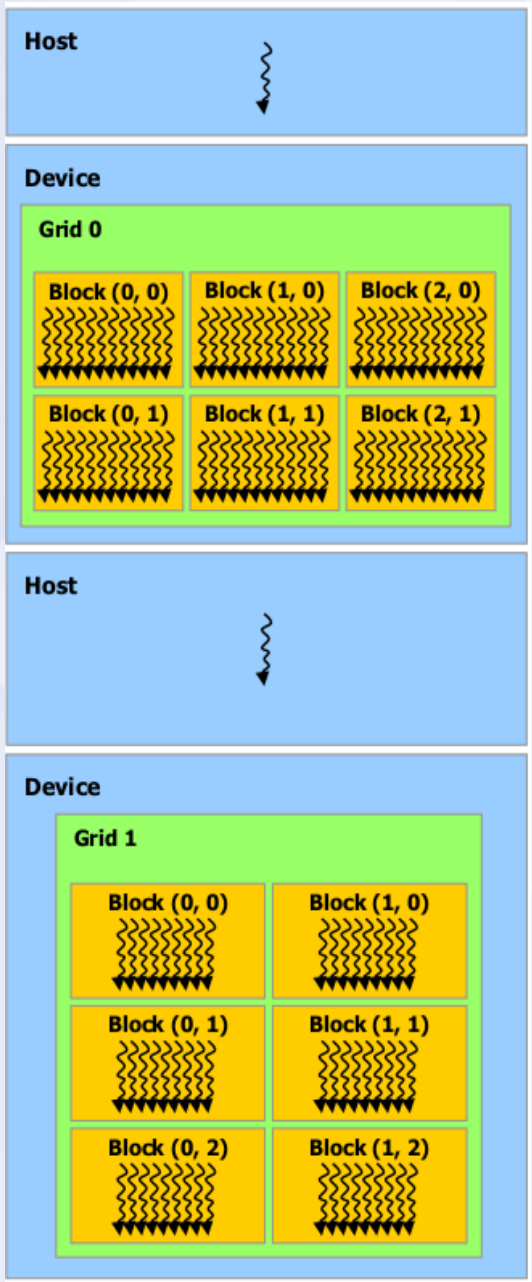
Wielokrotne wywoływanie kodu równoległego dla tego samego zestawu danych

Kod sekwencyjny

Kod równoległy
`kernel0<<<>>>()`

Kod sekwencyjny

Kod równoległy
`kernel1<<<>>>()`



Kod GPU (Sumowanie elementów macierzy)

```
__global__ void SumM(real* R){
extern __shared__ real rS[];
unsigned int id = threadIdx.x;
rS[id]=R[id];
__syncthreads();
if(id==0){
 real SUM=0;
 for(int i=0;i<16;i++){
   SUM+=rS[i];
  }
 R[0]=SUM;
 }
}
```

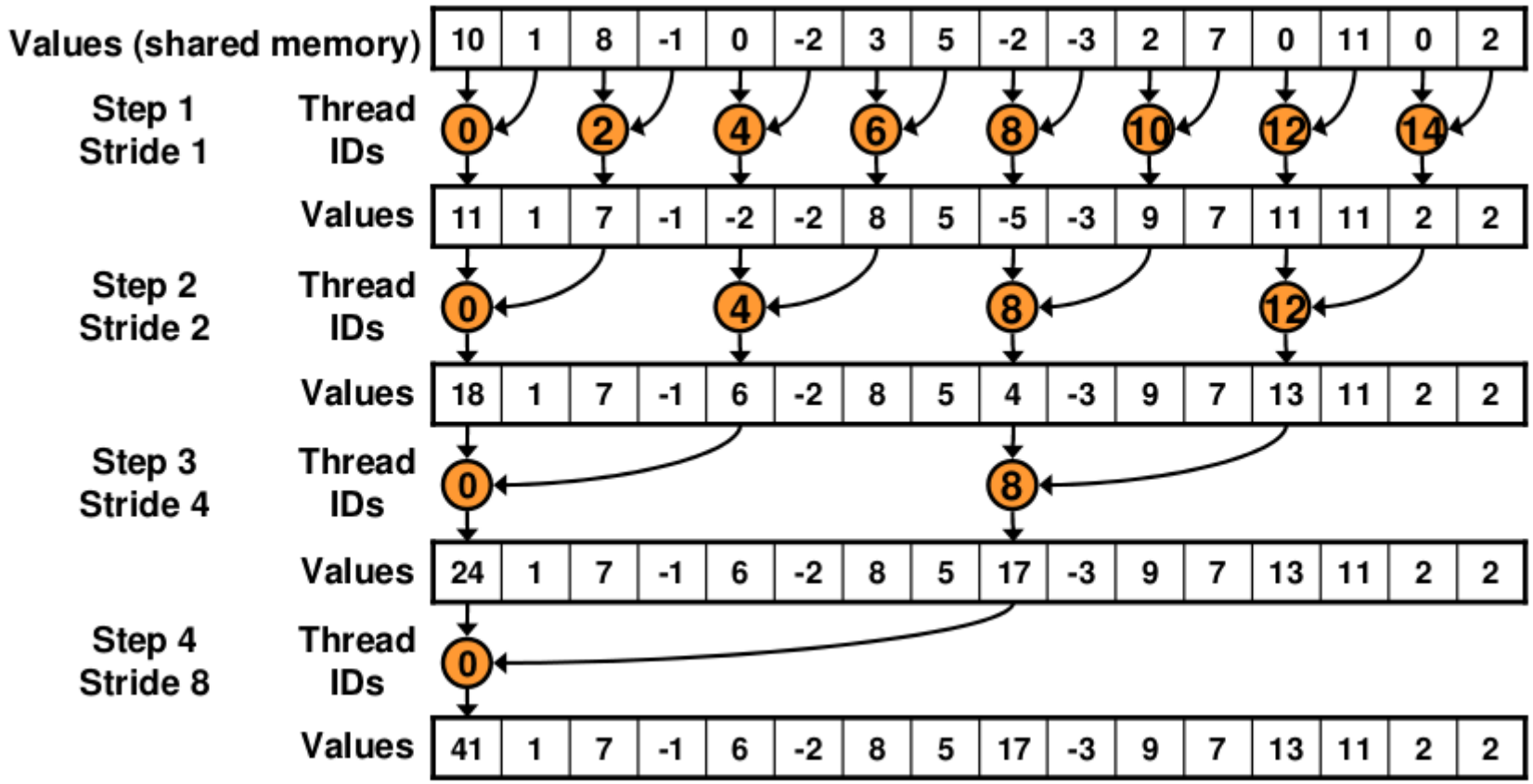
```
int main(int argc, char* argv[])
{
 CPUMemAlloc(); GPUMemAlloc();
 R[0]=5;R[1]=2;R[2]=4;
 CopyCPUToGPU(); InitKernel();
 SumM<<<blocksPerGrid, threadsPerBlock, sharedSize>>>(d_R);
 CopyGPUToCPU(); printf("R[0]=%f\n",R[0]);
 return 0;
}
```

Kod GPU (Redukcja 1)

W poprzednim programie do sumowania liczb z każdego wątku użyte został tylko jeden wątek o identyfikatorze 0

```
__global__ void SumReducto(real* R){
extern __shared__ real rS[];
unsigned int id = threadIdx.x;
rS[id]=R[id];
__syncthreads();

for(unsigned int s=1; s < blockDim.x; s *= 2) {
     if (id % (2*s) == 0) {
       rS[id] += rS[id + s];
       }
    __syncthreads();
  }

if(id == 0) R[0] = rS[0];

}
```

Kod GPU (Redukcja 1)



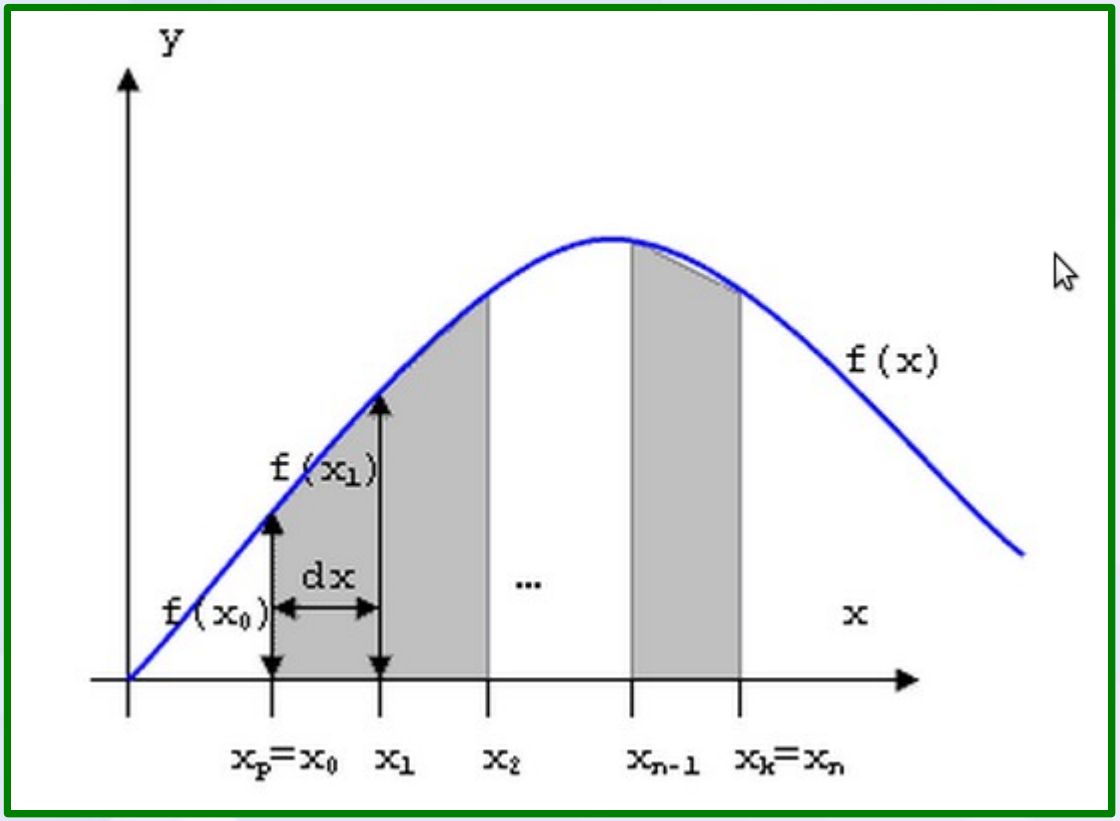
Kod GPU (Redukcja 2)

Operacja modulo (%) jest bardzo wolna i wymaga zmiany.

```
__global__ void SumReducto2(real* R){
extern __shared__ real rS[];
unsigned int id = threadIdx.x;
rS[id]=R[id];
__syncthreads();

for(unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2 * s * id;
    if (index < blockDim.x) {
      rS[index] += rS[index + s];
    }
__syncthreads();
}


if(id == 0) R[0] = rS[0];

}
```

Kod GPU (Obliczanie numeryczne całki – metoda trapezów)

Polega na sumowaniu pól trapezów pod krzywą



Kod GPU (Obliczanie numeryczne całki – metoda trapezów)

```
__global__ void Trapez(real* Y, real* R){
extern __shared__ real rS[];
unsigned int id = threadIdx.x;
rS[id]=Y[id];
__syncthreads();

if(id<blockDim.x-1){
   real field;
   field=(rS[id] + 0.5* abs(rS[id]-rS[id+1]))*dxgpu;
   __syncthreads();
   rS[id]=field;
}

for(unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2 * s * id;
    if (index < blockDim.x) {
      rS[index] += rS[index + s];
    }
__syncthreads();
}

if(id == 0) R[0] = rS[0];
}
```

Kod CPU (Obliczanie numeryczne całki – metoda trapezów)

```
__constant__ real dxgpu=0;

int main(int argc, char* argv[])
{
 CPUMemAlloc(); GPUMemAlloc();

 InitKernel(); // threadsPerBlock=128
 dx=0.0078125; //1/128
 cudaMemcpyToSymbol(dxgpu, &dx, sizeof(real));
 for(int i=0;i<threadsPerBlock;i++){
 Y[i]=sqrt(1-(i*dx)*(i*dx));
 }

 CopyCPUToGPU();
 Trapez<<<blocksPerGrid, threadsPerBlock, sharedSize>>>(d_Y, d_R);
 CopyGPUToCPU(); printf("R[0]=%f\n",R[0]);
 return 0;
}
```

KONIEC WYKŁADU