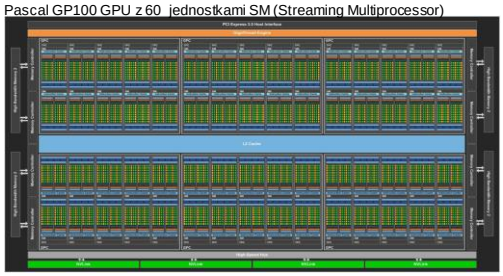
PROGRAMOWANIE WEKTOROWE I RÓWNOLEGŁE

Wykład dla kierunku: Matematyka stosowana i technologie informatyczne

RODZAJE PAMIĘCI I OPTYMALIZACJA JEJ WYKORZYSTANIA.

ARCHITEKTURA PASCAL – GTX10X0

Pascal GP100 GPU z 60 jednostkami SM (Streaming Multiprocessor)



ARCHITEKTURA PASCAL – GTX10X0

Układ zawiera:
- 6 jednostek Graphics Processing Cluster (GPC) po 10 SM
- 4 MB cache L2

Moduł SM zawiera:
- 64 rdzenie pojedynczej precyzji
- 32 rdzenie podwójnej precyzji
- 16 jednostek specjalne,
- 16 jednostek odczytująco-zapisujące
- 64 KB pamięci współdzielonej.

ARCHITEKTURA PASCAL – GTX10X0

Porównanie Kepler, Maxwell i Pascal GPU

GPU	Kepler GK110	Maxwell GM200	Pascal GP100
Compute Capability	3.5	5.2	6.0
Threads / Warp	32	32	32
Max Warps / Multiprocessor	64	64	64
Max Threads / Multiprocessor	2048	2048	2048
Max Thread Blocks / Multiprocessor	16	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	32768	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB

ARCHITEKTURA AMPERE – RTX30X0

Układ zawiera:
7 jednostek Graphics Processing Cluster (GPC) po 12 SM
6 MB cache L2
Moduł SM zawiera:
- 2 rdzenie podwójnej precyzji
- 128 rdzeni pojedynczej precyzji
- w tym 64 rdzenie całkowite
- 4 jednostki specjalne,
- 16 jednostek odczytująco-zapisujących
- 4 rdzeni Tensor
- 1 jednostkę śledzenia promieni
- 128 KB pamięci współdzielonej/L1.

STRUKTURA CUDADEVICEPROP - WYCIĄG

```
struct cudaDeviceProp {
 char name[256];
 size_t totalGlobalMem;
 size_t totalConstMem;
 size_t sharedMemPerBlock;
 size_t
sharedMemPerMultiProcessor;
 int L2CacheSize;
 int regsPerBlock;
 int regsPerMultiProcessor;
 int warpSize;
 size_t memPitch;
 int maxThreadsPerBlock;
 int maxThreadsPerMultiProcessor;
 int maxBlocksPerMultiProcessor
 int maxThreadsDim[3];
 int maxGridSize[3];
 int major;
 int minor;
 int clockRate;
 size_t textureAlignment;
 int deviceOverlap;
 int multiProcessorCount;
 int kernelExecTimeoutEnabled;
 int integrated;
 int canMapHostMemory;
 int computeMode;
 int maxTexture1D;
 int maxTexture2D[2];
 int maxTexture3D[3];
 int maxTexture2DArray[3];
 int concurrentKernels;
 }
```


WYDOBYCIE INFORMACJI O URZĄDZENIU

```
25      HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
26      printf( "   --- General Information for device %d ---\n", i );
27      printf( "Nazwa:  %s\n", prop.name );
28      printf( "Potencjał obliczeniowy:  %d.%d\n", prop.major, prop.minor );
29      printf( "Zegar:  %d\n", prop.clockRate );
30      printf( "Wartość deviceOverlap:  " );
31      if (prop.deviceOverlap)
32          printf( "Włączone\n" );
33      else
34          printf( "Wyłączone\n");
35      printf( "Limit czasu działania jądra:  " );
36      if (prop.kernelExecTimeoutEnabled)
37          printf( "Włączony\n" );
38      else
39          printf( "Wyłączony\n" );
```


WYDOBYCIE INFORMACJI O URZĄDZENIU

```
41      printf( "   --- Informacje o pamięci urządzenia %d ---\n", i );
42      printf( "Ilość pamięci globalnej:  %ld\n", prop.totalGlobalMem );
43      printf( "Ilość pamięci stałej:  %ld\n", prop.totalConstMem );
44      printf( "Maks. szerokość pamięci:  %ld\n", prop.memPitch );
45      printf( "Wyrównanie tekstur:  %ld\n", prop.textureAlignment );
46      printf( "   --- Informacje na temat wieloprocesorów urządzenia %d ---\n", i );
47      printf( "Liczba wieloprocesorów:  %d\n",
48              prop.multiProcessorCount );
49      printf( "Pamięć wspólna na wieloprocesor:  %ld\n", prop.sharedMemPerBlock );
50      printf( "Rejestry na wieloprocesor:  %d\n", prop.regsPerBlock );
51      printf( "Liczba wątków w osnowie:  %d\n", prop.warpSize );
52      printf( "Maks. liczba wątków na blok:  %d\n",
53              prop.maxThreadsPerBlock );
54      printf( "Maks. liczba wymiarów wątków:  (%d, %d, %d)\n",
55              prop.maxThreadsDim[0], prop.maxThreadsDim[1],
56              prop.maxThreadsDim[2] );
57      printf( "Maks. liczba wymiarów siatki:  (%d, %d, %d)\n",
58              prop.maxGridSize[0], prop.maxGridSize[1],
59              prop.maxGridSize[2] );
```


Wybrane właściwości

`char name[256]` - łańcuch znaków ASCII identyfikujący urządzenie;

`size_t totalGlobalMem` - globalna pamięć dostępna na urządzeniu w bajtach;
`size_t totalConstMem` - pamięć stała dostępna na urządzeniu w bajtach;
`size_t sharedMemPerBlock` - maksymalny rozmiar pamięci wspólnej na blok w bajtach;
`size_t sharedMemPerMultiProcessor` - maksymalny rozmiar pamięci wspólnej na multiprocesor w bajtach;
`int L2CacheSize` - rozmiar w bajtach pamięci podręcznej L2;

Wybrane właściwości

`int multiProcessorCount` - liczba multiprocesorów w urządzeniu;
`int regsPerBlock` - maksymalna liczba rejestrów na blok;
`int regsPerMultiProcessor` - maksymalna liczba rejestrów na multiprocesor;
`int maxThreadsPerBlock` - maksymalna liczba wątków na blok;
`int maxThreadsPerMultiProcessor` - maksymalna liczba wątków na multiprocesor;
`int maxBlocksPerMultiProcessor` - maksymalna liczba bloków na multiprocesor;
`int maxThreadsDim[3]` - maksymalny rozmiar każdego wymiaru bloku;
`int maxGridSize[3]` - maksymalny rozmiar każdego wymiaru siatki;
`int warpSize` - liczba wątków w osnowie;;

Wybrane właściwości

`size_t textureAlignment` - wymagania dotyczące wyrównania tekstur;
`int maxTexture1D` - maksymalny rozmiar tekstur 1D;
`int maxTexture2D[2]` - maksymalny rozmiar tekstur 2D;
`int maxTexture3D[3]` - maksymalny rozmiar tekstur 3D;
`int maxTexture2DArray[3]` - maksymalny rozmiar tablic tekstur 2D;

Wybrane właściwości

```
int concurrentKernels  – możliwość uruchomienia wielu jąder;
int major – zdolność obliczeniowa wartość główna;
int minor – zdolność obliczeniowa wartość po przecinku;
int clockRate – częstotliwość zagara w kHz;
int deviceOverlap – możliwość jednoczesnego wykonywania jądra i kopiowania;;
int kernelExecTimeoutEnabled  – ograniczenie czasowe wykonywania jądra;
int integrated – czy układ jest zintegrowany;
int canMapHostMemory  – czy można pamięć hosta na urządzenie;
```


Dodawanie wektorów – 1 – bloki

```
21  __global__ void add( int *a, int *b, int *c ) {
22      int tid = blockIdx.x;    // Ten wątek przetwarza dane pod określonym indeksem
23      if (tid < N)
24          c[tid] = a[tid] + b[tid];
25  }

48      add<<<N,1>>>( dev_a, dev_b, dev_c );
```


Dodawanie wektorów – 2 – bloki

```
21  __global__ void add( int *a, int *b, int *c ) {
22      int tid = blockIdx.x;
23      while (tid < N) {
24          c[tid] = a[tid] + b[tid];
25          tid += gridDim.x;
26      }
27  }

55      add<<<128,1>>>( dev_a, dev_b, dev_c );
```


Dodawanie wektorów – 3 – wątki

```
21  __global__ void add( int *a, int *b, int *c ) {
22      int tid = threadIdx.x;
23      if (tid < N)
24          c[tid] = a[tid] + b[tid];
25  }

48      add<<<1,N>>>( dev_a, dev_b, dev_c );
```


Dodawanie wektorów – 4 – wątki + bloki

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3
Block 3	Thread 0	Thread 1	Thread 2	Thread 3

Dodawanie wektorów – 4 – wątki + bloki

```
21  __global__ void add( int *a, int *b, int *c ) {
22      int tid = threadIdx.x + blockIdx.x * blockDim.x;
23      if (tid < N)
24          c[tid] = a[tid] + b[tid];
25  }

55      add<<<(N+127)/128,128>>>( dev_a, dev_b, dev_c );
56      //lub
57      add<<<128,128>>>( dev_a, dev_b, dev_c );
```

Dodawanie wektorów – 5 – wątki + bloki

```
21  __global__ void add( int *a, int *b, int *c ) {
22      int tid = threadIdx.x + blockIdx.x * blockDim.x;
23      while (tid < N) {
24          c[tid] = a[tid] + b[tid];
25          tid += blockDim.x * gridDim.x;
26      }
27  }

55      add<<<128,128>>>( dev_a, dev_b, dev_c );
```


Klasyfikatory zmiennych w przestrzeni pamięci

Klasyfikatory zmiennych w przestrzeni pamięci określają lokalizację pamięci na urządzeniu, w której znajduje się zmienna.

Zmienna automatyczna zadeklarowana w kodzie urządzenia bez żadnego klasyfikatora przestrzeni pamięci __device__, __shared__ lub __constant__ zazwyczaj rezyduje w rejestrze.

Jednak w niektórych przypadkach kompilator może zdecydować się na umieszczenie go w pamięci lokalnej, co może mieć niekorzystny wpływ na wydajność.

Klasyfikator __device__

Klasyfikator przestrzeni pamięci __device__ deklaruje zmienną rezydującą na urządzeniu. Co najwyżej jeden z innych klasyfikatorów przestrzeni pamięci może być użyty razem z __device__ w celu dalszego określenia, do której przestrzeni pamięci należy zmienna. Jeśli żaden z nich nie jest obecny, zmienna:

- Rezyduje w globalnej przestrzeni pamięci,
- Posiada czas życia kontekstu CUDA, w którym jest tworzona,
- Ma odrębny obiekt na urządzenie,
- Jest dostępna ze wszystkich wątków w siatce oraz z hosta poprzez bibliotekę wykonawczą (cudaGetSymbolAddress() / cudaGetSymbolSize() / cudaMemcpyToSymbol() / cudaMemcpyFromSymbol()).

Klasyfikator __shared__

Klasyfikator przestrzeni pamięci __shared__, opcjonalnie używany razem z __device__, deklaruje zmienną, która:

- Rezyduje we współdzielonej przestrzeni pamięci bloku wątków,
- Ma żywotność bloku,
- Ma odrębny obiekt na blok,
- Jest dostępny tylko ze wszystkich wątków w bloku,
- Nie ma stałego adresu.

Klasyfikator __constant__

Klasyfikator przestrzeni pamięci __constant__, opcjonalnie używany razem z __device__, deklaruje zmienną, która:

- Rezyduje w przestrzeni pamięci stałych,
- Posiada czas życia kontekstu CUDA, w którym jest tworzona,
- Ma odrębny obiekt na urządzenie,
- Jest dostępna ze wszystkich wątków w siatce oraz z hosta poprzez bibliotekę wykonawczą (cudaGetSymbolAddress() / cudaGetSymbolSize() / cudaMemcpyToSymbol() / cudaMemcpyFromSymbol()).

Klasyfikator __grid_constant__

Klasyfikator __grid_constant__ dla architektur obliczeniowych większych lub równych 7.0 zawiera adnotacje do parametru funkcji __global__ z kwalifikacją const typu innego niż referencyjny, który:

- Ma żywotność sieci,
- Jest prywatny dla siatki, tj. obiekt nie jest dostępny dla wątków hosta i wątków z innych siatek, w tym podsiatek,
- Ma odrębny obiekt na siatkę, tj. Wszystkie wątki w siatce widzą ten sam adres,
- Jest tylko do odczytu, tj. modyfikacja obiektu __grid_constant__ lub któregokolwiek z jego obiektów podrzędnych jest niezdefiniowanym zachowaniem.

Wymagania:

- Parametry jądra z adnotacją __grid_constant__ muszą mieć kwalifikowane przez const typy niereferencyjne.
- Wszystkie deklaracje funkcji muszą być zgodne w odniesieniu do wszelkich parametrów __grid_constant__.
- Specjalizacja szablonu funkcji musi być zgodna z deklaracją szablonu podstawowego w odniesieniu do wszelkich parametrów __grid_constant__.
- Dyrektywa tworzenia instancji szablonu funkcji musi być zgodna z deklaracją szablonu podstawowego w odniesieniu do wszelkich parametrów __grid_constant__ .

Jeśli zostanie pobrany adres parametru funkcji __global__, kompilator zwykle utworzy kopię parametru jądra w pamięci lokalnej wątku i użyje adresu kopii, aby częściowo obsługiwać semantykę C++, która pozwala każdemu wątkowi modyfikować własną kopię lokalną parametrów funkcji. Adnotacja parametru funkcji __global__ za pomocą __grid_constant__ gwarantuje, że kompilator nie utworzy kopii parametru jądra w pamięci lokalnej wątku, ale zamiast tego użyje ogólnego adresu samego parametru. Unikanie kopii lokalnej może poprawić wydajność.

Klasyfikator __managed__

Klasyfikator __managed__ przestrzeni pamięci, opcjonalnie używany razem z __device__, deklaruje zmienną, która:

- Można się do niej odwoływać zarówno z kodu urządzenia, jak i hosta, na przykład jej adres można pobrać lub można go odczytać lub zapisać bezpośrednio z urządzenia lub funkcji hosta.
- Ma czas życia aplikacji.

Iloczyn skalarny

Iloczyn skalarny:

$$c = \sum_{i=1}^{N} a_i b_i$$

Iloczyn skalarny

```
#include "../common/book.h"

#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
            imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

__global__ void dot( float *a, float *b, float *c ) {


int main( void ) {
```


Iloczyn skalarny

```
int main( void ) {
    float   *a, *b, c, *partial_c;
    float   *dev_a, *dev_b, *dev_partial_c;

    // Alokacja pamięci na CPU
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );

    // Alokacja pamięci na GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                              N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                              N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                              blocksPerGrid*sizeof(float) ) );

    // Zapełnienie pamięci hosta danymi
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    // Skopiowanie tablic a i b do GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                              cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                              cudaMemcpyHostToDevice ) );

    dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
                                            dev_partial_c );
```


Iloczyn skalarny

```
    dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
                                            dev_partial_c );

    // Skopiowanie tablicy c z powrotem z GPU do CPU
    HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                              blocksPerGrid*sizeof(float),
                              cudaMemcpyDeviceToHost ) );

    // Zakończenie pracy na CPU
    c = 0;
    for (int i=0; i<blocksPerGrid; i++) {
        c += partial_c[i];
    }

    #define sum_squares(x)  (x*(x+1)*(2*x+1)/6)
    printf( "Does GPU value %.6g = %.6g?\n", c,
             2 * sum_squares( (float)(N - 1) ) );

    // Zwolnienie pamięci na GPU
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_partial_c ) );

    // Zwolnienie pamięci na CPU
    free( a );
    free( b );
    free( partial_c );
}
```


Iloczyn skalarny

```
__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // Ustawienie wartości pamięci podręcznej
    cache[cacheIndex] = temp;

    // Synchronizacja wątków w tym bloku
    __syncthreads();

    // W przypadku redukcji threadsPerBlock musi być potęga 2,
    // ze względu na poniższy kod
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```

Funkcja __syncthreads()

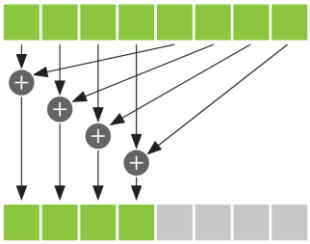
void __syncthreads();

czeka, aż wszystkie wątki w bloku wątków osiągną ten punkt, a wszystkie dostępy do pamięci globalnej i współdzielonej dokonane przez te wątki przed __syncthreads() będą widoczne dla wszystkich wątków w bloku.

__syncthreads() służy do koordynowania komunikacji między wątkami tego samego bloku. Gdy niektóre wątki w bloku uzyskują dostęp do tych samych adresów w pamięci współdzielonej lub globalnej, w przypadku niektórych z tych dostępów do pamięci istnieją potencjalne zagrożenia związane z odczytem po zapisie, zapisem po odczycie lub zapisem po zapisie. Tych zagrożeń związanych z danymi można uniknąć, synchronizując wątki pomiędzy tymi dostępami.

__syncthreads() jest dozwolone w kodzie warunkowym, ale tylko wtedy, gdy warunek daje identyczną wartość w całym bloku wątków, w przeciwnym razie wykonanie kodu może się zawiesić lub wywołać niezamierzone skutki uboczne.

Iloczyn skalarny - redukcja



Iloczyn skalarny

```
__global__ void dot2( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // Ustawienie wartości pamięci podręcznej
    cache[cacheIndex] = temp;

    // Synchronizacja wątków w tym bloku
    __syncthreads();

    // Redukcja - threadsPerBlock nie musi być potęgą 2!!!,
    int i = blockDim.x/2;
    int cor = i + blockDim.x%2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + cor];
        __syncthreads();
        i = cor/2;
        cor = i + cor%2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```