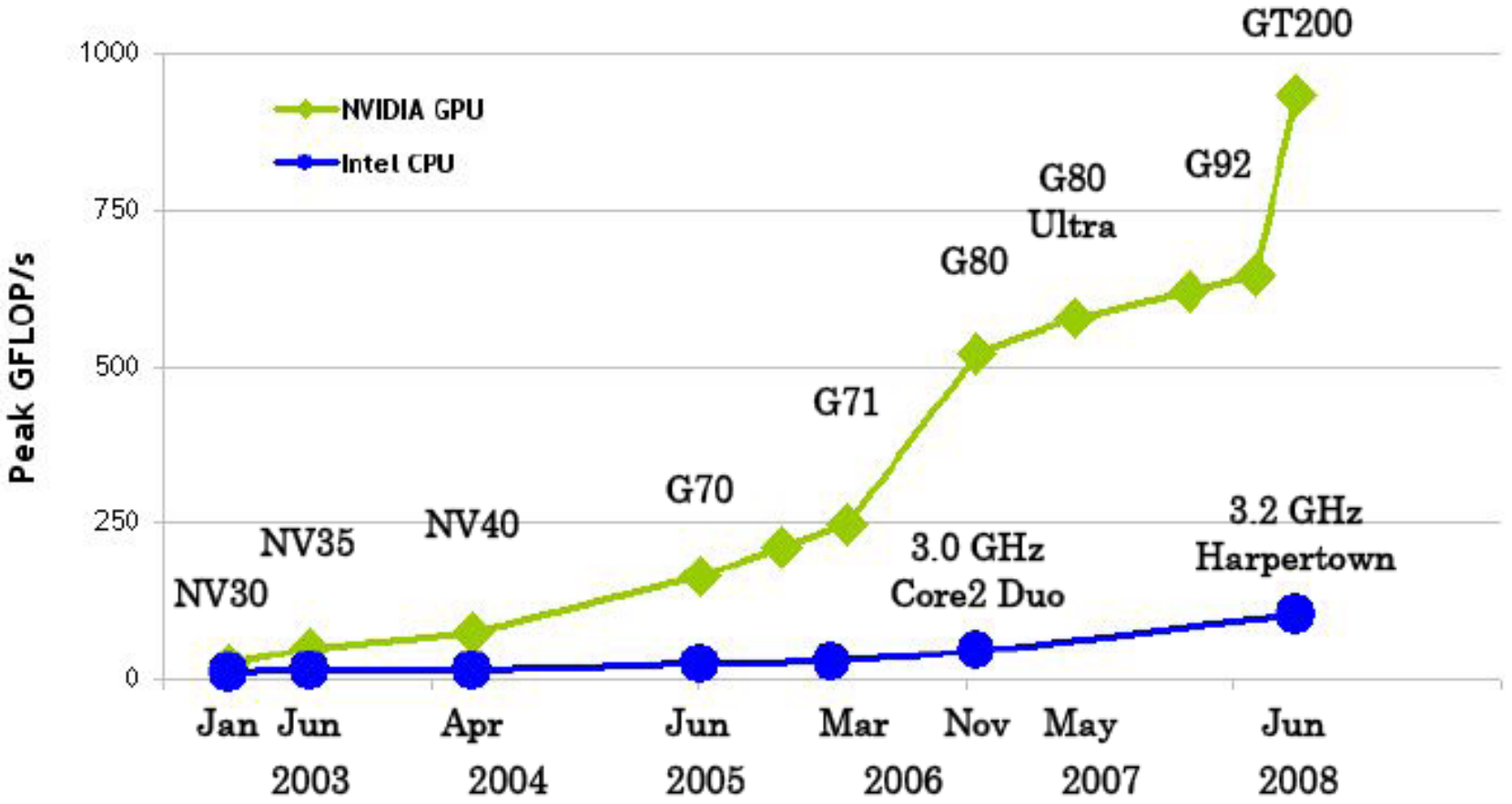
Obliczenia na GPU w technologii CUDA

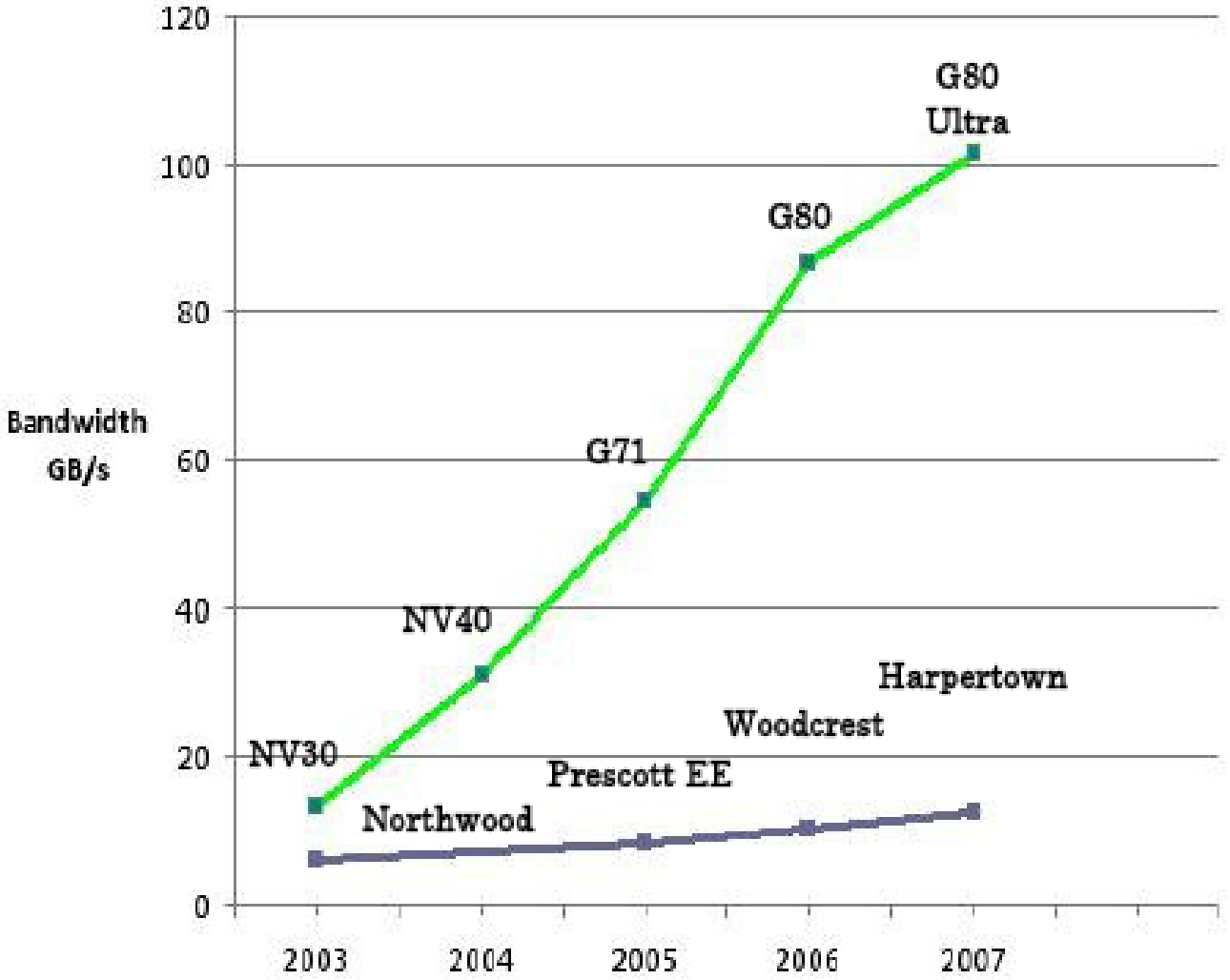
Różnica szybkości obliczeń (GFLOP/s) pomiędzy CPU a GPU – źródło NVIDIA



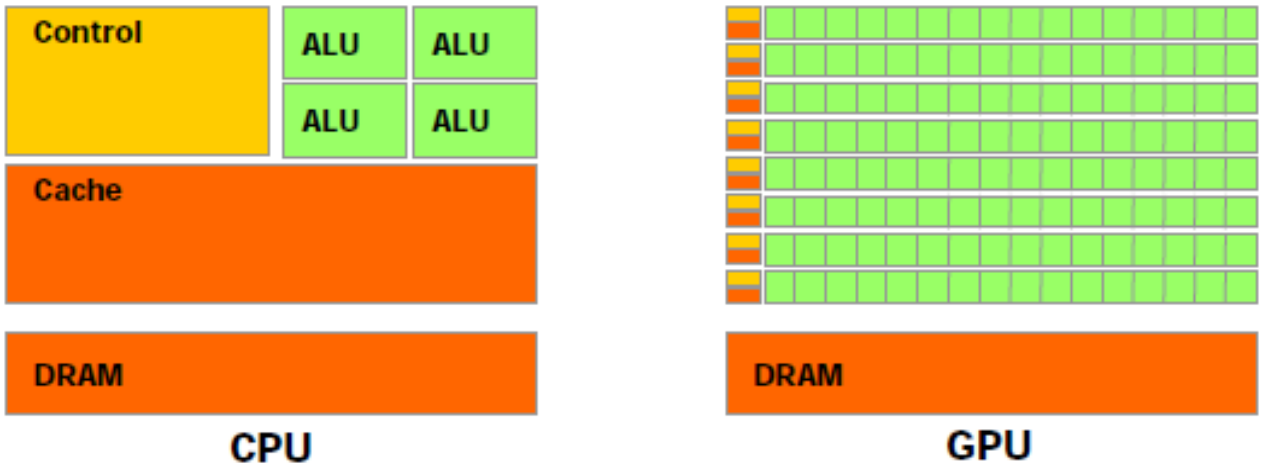
GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

Różnica w przepustowości pamięci pomiędzy CPU a GPU – źródło NVIDIA

GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	



Różnice architektoniczne



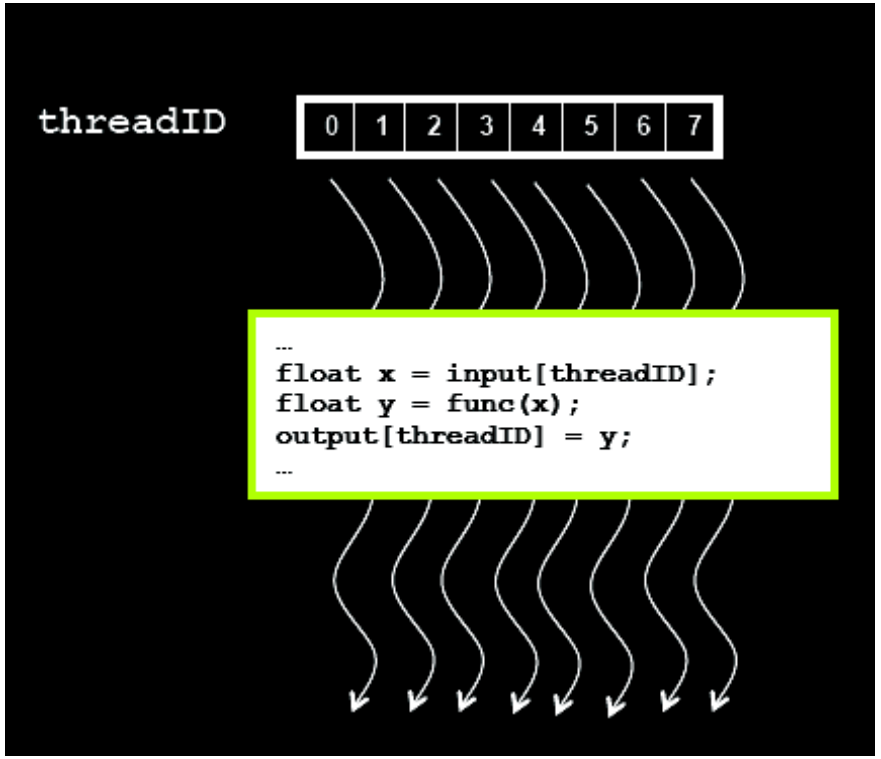
- GPU przeznacza więcej tranzystorów na przetwarzanie danych niż na sterowanie i pamięci cache.
- GPU szczególnie nadaje się do problemów które mogą być przedstawione jako „data-parallel computation” - ten sam program jest wykonywany równolegle na wielu elementach (np. wektora/macierzy)
- Odwzorowanie elementów danych w tysiące wątków wykonujących się na setkach rdzeni.

CUDA

- Compute Unified Device Architecture.
- Zaprojektowana wyłącznie pod procesory NVIDIA.
  - Ale z emulacją na CPU (z możliwością wykorzystania wielu rdzeni)
- Jest rozszerzeniem języka C/C++.
  - Ale dostępne wiązania dla innych platform (np. Python/Java/.NET).
- Zaprojektowana aby skalować się na tysiące wątków i setki rdzeni.
- Oferuje wydajność na procesorach GPU porównywalną z (niewielkimi) klastrami.
  - Liczby podwójnej precyzji gorzej wspierane i dużo wolniej.
- Model programowania heterogeniczny szeregowo-równoległy.
  - Porcje kodu wykonywane równolegle na GPU, reszta szeregowo na CPU
  - GPU jest koprocesorem dla CPU.
  - GPU i CPU mają rozłączne pamięci DRAM.

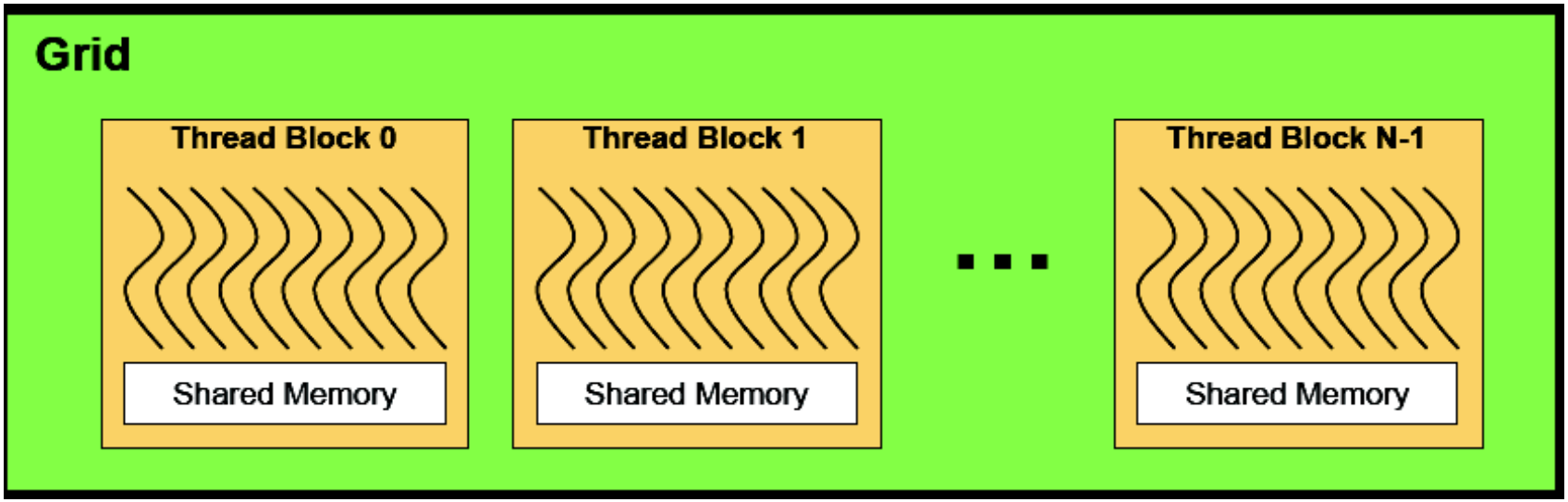
Jądra (ang. kernels) i wątki w CUDA

- Równoległe porcje aplikacji są wykonywane na GPU jako jądra.
  - Tylko jedno jądro jest wykonywane w danej chwili
  - Wiele wątków (tysiące) wykonuje to samo jądro.
- Jądro jest wykonywane przez wektor wątków.
  - Każdy z nich wykonuje ten sam kod.
  - Każdy wątek ma identyfikator, który wykorzystuje do obliczania adresów pamięci.



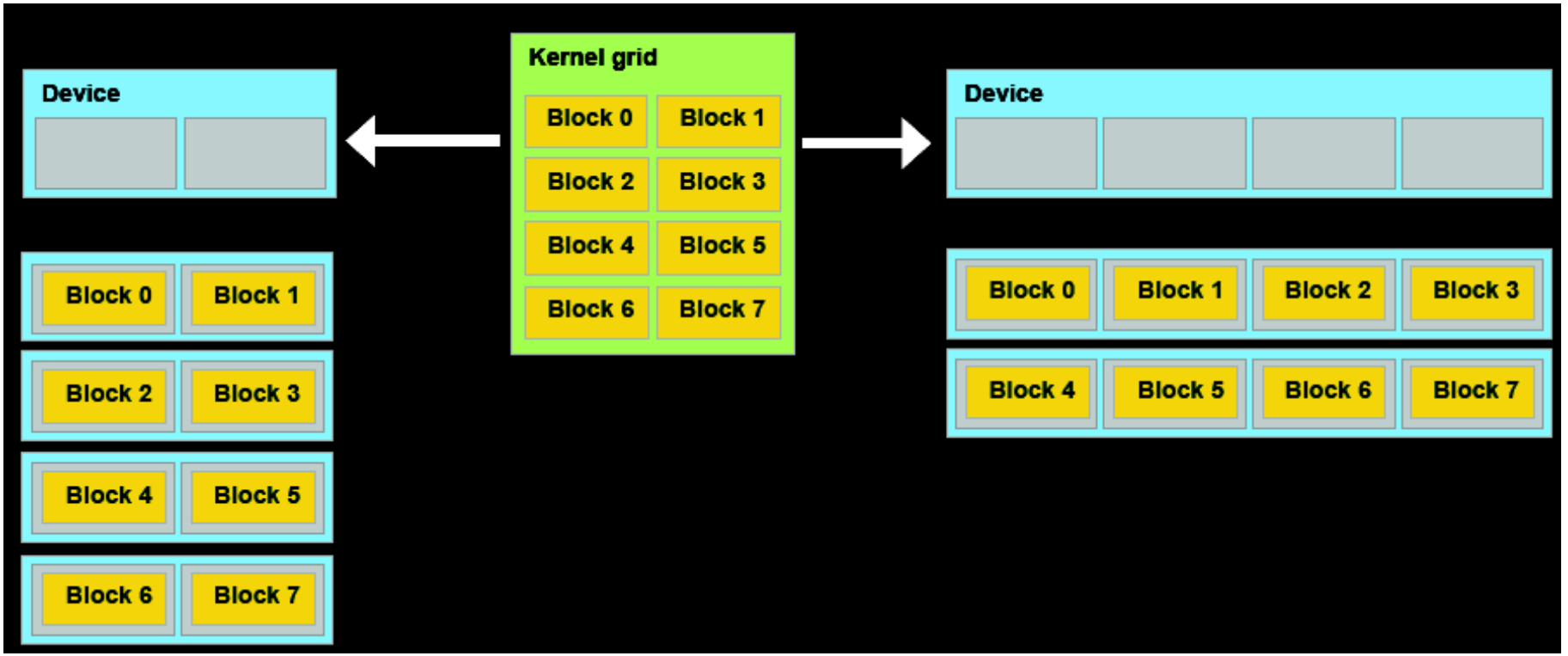
- Wątki operujące na wektorze.

Grid wątków



- Numerowanie wektora wątków nie jest monolityczne
- Jądro uruchamia grid składający się z N bloków.
- Każdy blok składa się z M wątków => łącznie mamy N*M wątków w gridzie.
- Adres bloku w gridzie może być dwuwymiarowy (jest strukturą z polami x oraz y). Adres wątku w bloku może być dwuwymiarowy (jest strukturą z polami x,y,z).
- Na powyższym rysunku adres jest jednowymiarowy - pola y,z są równe 1.
- Zatem jeżeli wątek operuje na wektorze 0,...,N*M, to numer elementu wektora może być obliczony przy pomocy wyrażenia.
  - numer_wątku_w_bloku+rozmiar_bloku*numer_bloku.

Motywacja: skalowalność



- Hardware może zaszeregować dowolny blok na dowolnym procesorze.
- Brak założeń o kolejności wykonywania bloków.

Wykonywanie kodu na GPU

- Jądra są funkcjami języka C z pewnymi ograniczeniami:
  - Nie mogą odwoływać się do pamięci CPU
  - Muszą zwracać void jako wynik.
  - Nie mogą przyjmować zmiennej liczby argumentów
  - Nie mogą być rekurencyjne
  - Nie mogą deklarować zmiennych statycznych.
- Parametry są kopiowane automatycznie z CPU na GPU.
- Deklarowane z kwalifikatorem `__global__` oznaczającym funkcję wywoływaną z CPU i wykonującą się na GPU. Inne kwalifikatory to:
  - `__device__` oznacza funkcję wywoływaną z GPU i wykonującą się na GPU.
  - `__host__` oznacza funkcję wywoływaną z CPU w wykonującą się na GPU (domyślnie)
  - `__device__` i `__host__` mogą być łączone.

Pierwsze jądro i równoważny kod CPU.

- Kod dla CPU inkrementujący wszystkie elementy wektora

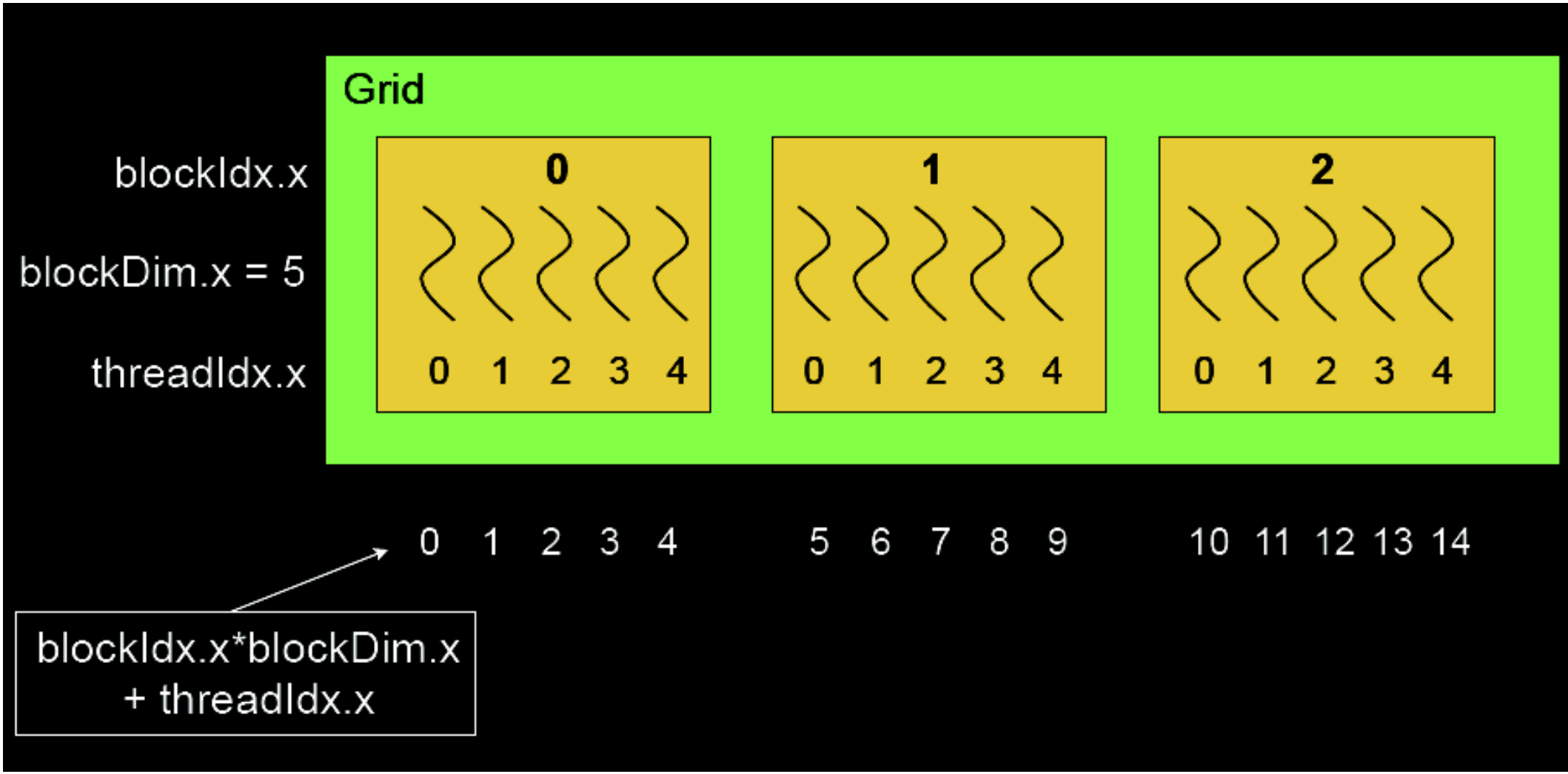
```
void inc_cpu(float *a, int N)
{
    int idx;
    for (idx = 0; idx<N; idx++)
       a[idx] = a[idx] + 1.0;
}
```

- Jądro dla GPU w którym każdy wątek inkrementuje jeden element wektora.

```
__global__ void inc_gpu(float *a,
   int N)
{
    int idx = blockIdx.x
   *blockDim.x
+ threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + 1.0;
}
```

- Zmienne predefiniowane w każdym jądrze.
  - `blockidx` - numer bloku
  - `blockDim` - rozmiar bloku
  - `threadIdx` - numer wektora
- Są to struktury 2/3 wymiarowe ale my wykorzystujemy tylko jeden wymiar: pole x.

Mapowanie identyfikatorów w numery elementów tablicy



Wywołanie jądra

- Wywołanie jądra z kodu CPU ma postać:

```
f<<liczba_bloków_w_gridzie,rozmiar_bloku>>(parametry)
```

- W naszym przypadku:

```
float *d_a;
void LaunchKernel(){
   int blocksize=256;
   dim3 dimBlock (blocksize);
   dim3 dimGrid( ceil( N / (float)blocksize) );
   inc_gpu<<<dimGrid, dimBlock>>>(d_a, N);
}
```

- Rozmiar bloku przyjęliśmy na 256 wątków
- Struktura `dim3` zawiera pola x,y,z. My wykorzystujemy tylko x.
- Rozmiar gridu jest tak dobrany aby całkowita liczba wątków była najmniejszą wielokrotnością 256 większą bądź równą N.
  - Stąd przeznaczenie instrukcji if (idx<N)
  - Gdy N=600, to `dimGrid.x==3` i mamy 768 wątków (za dużo !!!)

Alokacja pamięci GPU

- CPU zarządza pamięcią GPU:
  - `cudaMalloc (void ** pointer, size_t nbytes)`
  - `cudaMemset (void * pointer, int value, size_t count)`
  - `cudaFree (void* pointer)`

- Przykład

```
int n = 1024;
int nbytes = 1024*sizeof(int);
// Wskaźnik na pamięć GPU;
int * d_a = 0; //
cudaMalloc( (void**)&d_a, nbytes );
// Próba odwołania się do pamięci wskazywanej przez d_a
// w kodzie CPU prowadzi do błędu ochrony pamięci
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Synchronizacja CPU z GPU

- Wywołania jąder są asynchroniczne.
  - Sterowanie powraca od razu do CPU bez oczekiwania na zakończenie wątków jądra
  - Jądro jest kolejkowane i wykona się po zakończeniu wszystkich poprzednich wywołań CUDA.

- Funkcja `cudaMemcpy` jest synchroniczna.
  - Kopiowanie rozpocznie się po wykonaniu poprzednich wywołań CUDA.
  - Sterowanie powraca do CPU po wykonaniu kopii.
  - Kopie pomiędzy pamięciami GPU są asynchroniczne.

- Funkcja `cudaThreadSynchronize()` wstrzymuje kod CPU do momentu wykonania wszystkich kolejkowanych wywołań CUDA

Zaawansowana alokacja pamięci

- Funkcja `cudaError_t cudaMallocHost (void ptr, size_t size)` alokuje pamięć CPU, której wszystkie strony są zablokowane w pamięci.
- Kopia z takiej pamięci do pamięci GPU może być wykonana przy pomocy mechanizmu DMA poprzez szynę PCIe. Jest to znacznie szybsze od kopiowania pamięci zaalokowanej standardowo np. przez funkcję `malloc`.
- Zaalokowanie zbyt dużej ilości pamięci przy pomocy tej funkcji może obniżyć wydajność systemu (wyłącza się mechanizm pamięci wirtualnej)
- Pamięć jest zwalniana przy pomocy funkcji `cudaError_t cudaFreeHost (void *ptr)`.
- Funkcja `cudaError_t cudaMallocPitch (void devPtr, size_t pitch, size_t width, size_t height)` alokuje pamięć dla macierzy o szerokości `width` (liczbie kolumn) i wysokości (liczbie wierszy) `height`.
  - Ale szerokość wiersza jest zwiększana aby każdy nowy wiersz był optymalnie wyrównany (ang. aligned) w pamięci.
  - Nowa szerokość jest zapamiętywana w pod adresem `pitch`.

Kopiowanie danych

- `cudaMemcpy( void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
- Powraca po wykonaniu kopii
- Blokuje kod CPU do momentu wykonania kopii.
- Nie rozpocznie kopiowania do momentu wykonania poprzednich wywołań CUDA (np. jąder)
- `enum cudaMemcpyKind`
  - `cudaMemcpyHostToDevice` kopia z pamięci CPU do pamięci GPU
  - `cudaMemcpyDeviceToHost` kopia z pamięci GPU do pamięci CPU
  - `cudaMemcpyDeviceToDevice` kopia z pamięci GPU do pamięci CPU
- `cudaMemcpyAsync` nie blokuje kodu CPU.

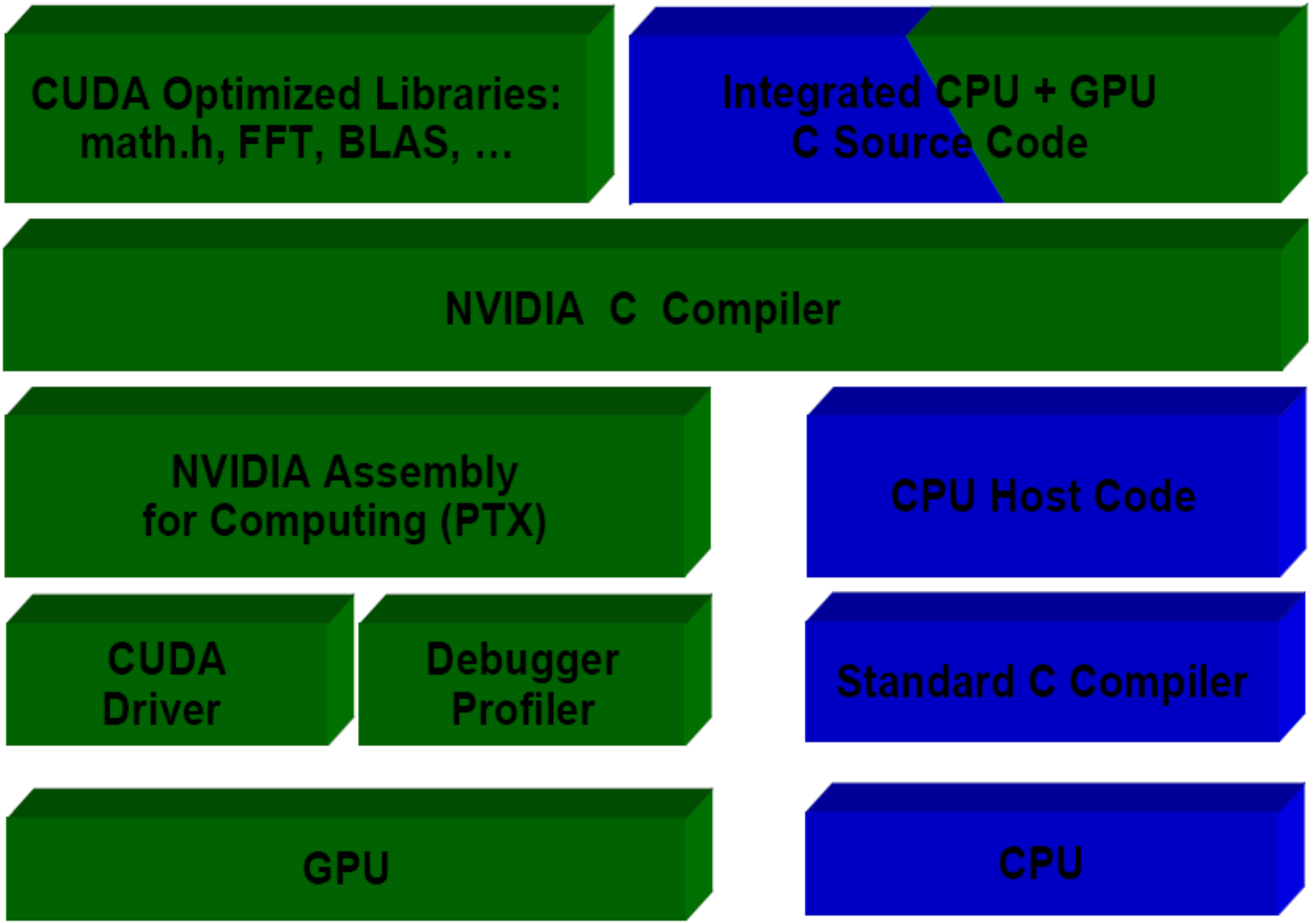
Kod programu

```
// Niezbędne aby korzystać z API CUDA
#include <cuda.h>

float *a_h; //wskaźnik do pamięci CPU
float *a_d; //wskaźnik do pamięci GPU

int main(void)
{
  int i, N = 10000;
  size_t size = N*sizeof(float);
  a_h = (float *)malloc(size); // alokacja pamięci CPU
  cudaMalloc((void **) &a_d, size); // alokacja pamięci GPU
  // Inicjalizacja danych
  for (i=0; i<N; i++) a_h[i] = (float)i;
  // Kopia z pamięci CPU do pamięci GPU
  cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
  LaunchKernel(); // zwiększ wszystkie elementy o jeden
  // Kopia z pamięci GPU z powrotem do a_h (pamięć CPU)
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // dealokacja pamięci
  free(a_h); cudaFree(a_d);
}
```

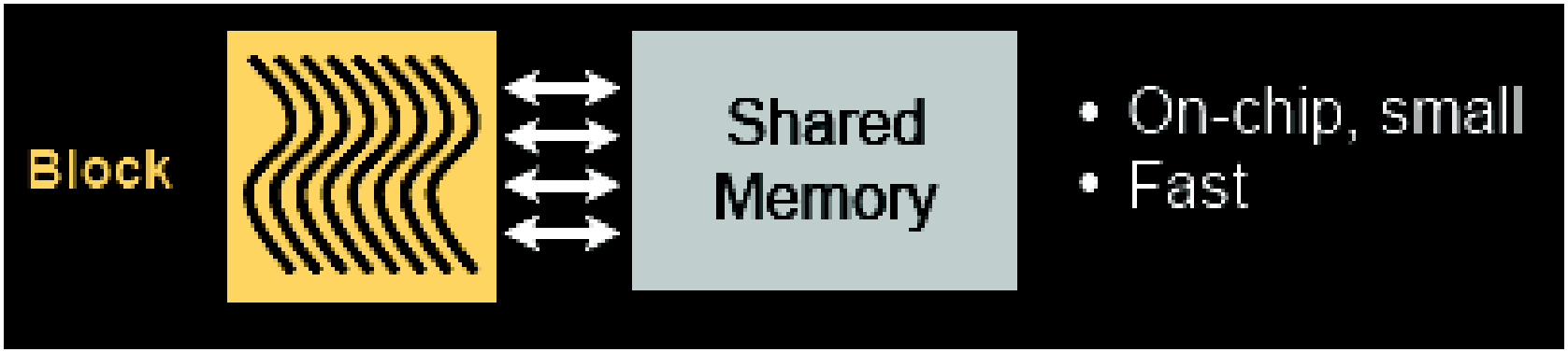
Kompilacja aplikacji

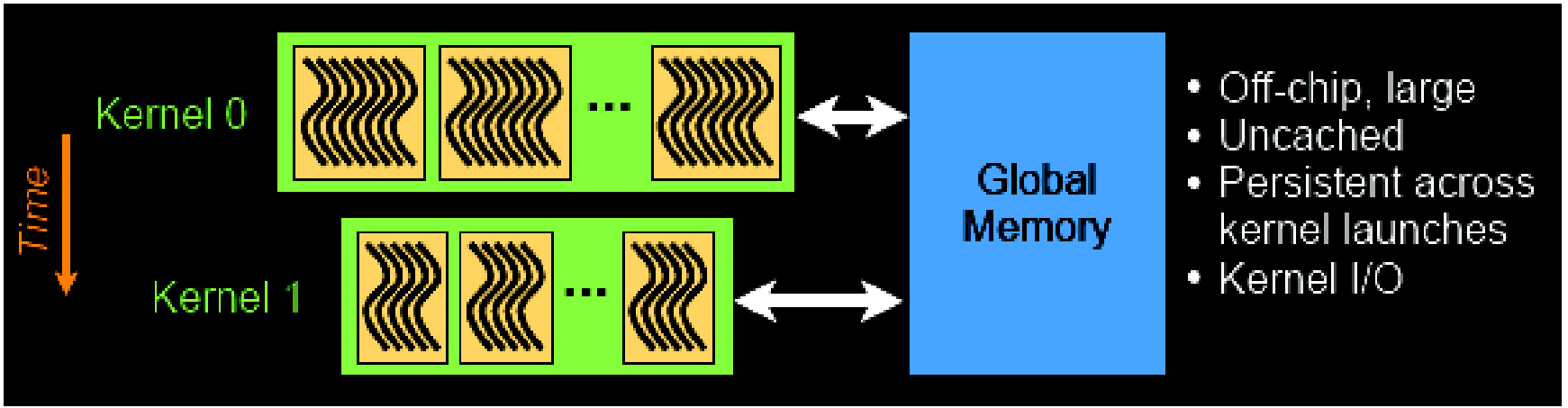
- Pliki C/C++ z rozszerzeniami CUDA mają (zwyczajowo) rozszerzenie .cu. Kompilowane są narzędziem nvcc.
- nvcc -O3 plik.cu -o program => kompilacja z optymalizacją dla CPU
  - -O3 przekazywane kompilatorowi gcc
- dodatkowa opcja -g: informacje dla debuggera
- dodatkowa opcja -deviceemu
  - Kompilacja pod emulator, cały kod wykonuje się na CPU.
  - Można używać funkcji printf w kodzie przeznaczonym dla GPU !!!
  - Zdefiniowane jest makro `__DEVICE_EMULATION__`
  - Ale uwaga: błędny kod może wykonywać się poprawnie z włączoną emulacją. Np. próba odwołania się przez GPU do pamięci CPU zakończy się sukcesem.
  - Opcja -multicore – wykorzystanie wielu rdzeni.
- Debugger CUDA-GDB pod Linuksa. (z poważnym ograniczeniem: X11 nie może się wykonywać na GPU na którym jest debugowany kod).

CUDA SDK

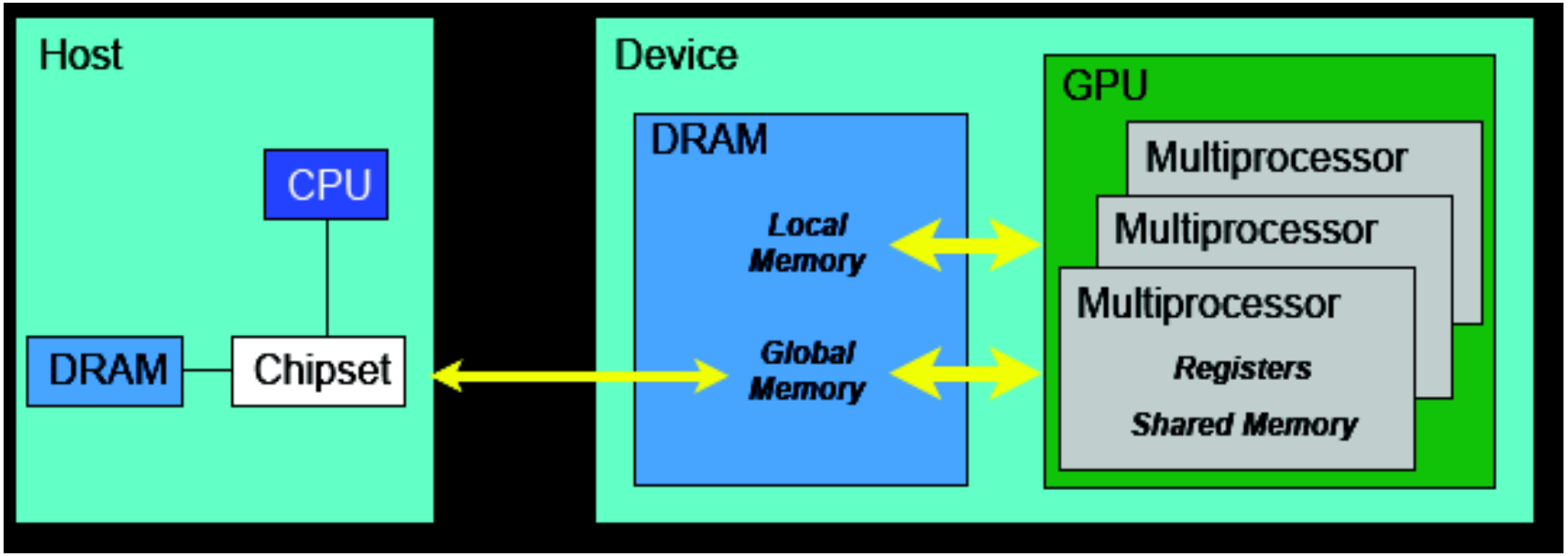

CUDA Optimized Libraries:
math.h, FFT, BLAS, ...
Integrated CPU + GPU
C Source Code
NVIDIA C Compiler
NVIDIA Assembly
for Computing (PTX)
CPU Host Code
CUDA
Driver
Debugger
Profiler
Standard C Compiler
GPU
CPU

Typy pamięci





Fizyczne umiejscowienie pamięci



- Pamięć lokalna rezyduje w pamięci DRAM karty. Należy używać pamięci wspólnej aby minimalizować wykorzystanie pamięci lokalnej.
- Pamięć wspólna rezyduje na procesorze GPU – dostęp znacznie szybszy niż do innych rodzajów pamięci.
- CPU ma dostęp do pamięci globalnej ale nie do pamięci współdzielonej.

Kwalifikatory typów pamięci

- `__device__` oznacza zmienną przechowywaną w pamięci globalnej (duży czas dostępu, brak pamięci podręcznej)
  - Pamięć alokowana przez `cudaMalloc` jest globalna.
  - Dostęp z wszystkich wątków gridu.
  - Czas życia: aplikacja.
- `__shared__` oznacza zmienną przechowywaną w pamięci współdzielonej.
  - Przechowywana wewnątrz GPU (bardzo krótki czas dostępu)
  - Dostępna wyłącznie wątkom wewnątrz tego samego bloku.
  - Czas życia: blok wątków.
- Zmienne bez kwafikatorów
  - O ile to możliwe przechowywane w rejestrach.
  - Co się nie zmieści w rejestrach przechowywane jest w pamięci lokalnej.

Wykorzystanie pamięci współdzielonej

- Rozmiar znany w chwili kompilacji

```
__global__ void kernel(…)
{
…
__shared__ float sData[256];
…
}
int main(void)
{
…
kernel<<<nBlocks,blockSize>>>(…)
;
…
}
```

- Rozmiar znany w chwili wykonywania programu (dodatkowy parametr jądra).

```
__global__ void kernel(…)
{
…
extern __shared__ float
sData[];
…
}
int main(void)
{
…
smBytes =
blockSize*sizeof(float);
kernel<<<nBlocks, blockSize,
smBytes>>>(…);
…
}
```

`sData` jest współdzielone przez wszystkie wątki bloku !!!

Synchronizacja wątków w bloku

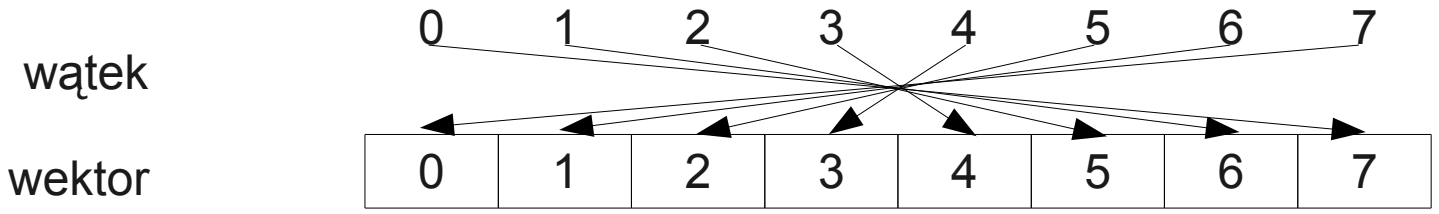
- CUDA umożliwia współpracę wątków w ramach bloku.
- Funkcja `void _syncthreads()`
- Synchronizacja barierowa: każdy wątek w bloku oczekuje, aż wszystkie inne wątki osiągną punkt wywołania funkcji `_syncthreads()`
- Bariera dotyczy wyłącznie wątków w bloku, nie w gridzie. Ale tylko takie wątki mogą korzystać z pamięci współdzielonej.
- Stosowana w celu uniknięcia wyścigów przy dostępie do pamięci wspólnej.
- Jeżeli występuje w kodzie warunkowym np. instrukcja if, to musimy zadbać aby wszystkie wątki ją wywołały.

Przykład odwrócenie kolejności elementów wektora

- Pierwszy element staje się ostatnim, drugi przedostatnim i na odwrót.

```
__global__ void reverseArrayBlock(int *d_out, int *d_in)
{
    int inOffset  = blockDim.x * blockIdx.x;
    int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
    int in  = inOffset + threadIdx.x;
    int out = outOffset + (blockDim.x - 1 - threadIdx.x);
    d_out[out] = d_in[in];
}
```

- Kopia wektora `d_in` z odwróconą kolejnością trafia do wektora `d_out`.
- `outOffset` oraz out liczone są od końca tablicy.
- Problem: zapis do d_out następuje w odwrotnej kolejności:



  - Spowalnia to kod kilkanaście razy (pamięć globalna nie ma pamięci cache, za to posiada hardware do przyspieszania dostępów sekwencyjnych)

Wersja z wykorzystaniem pamięci wspólnej

- Pamięć wspólna staje się software'owo zarządzaną pamięcią cache.

```
__global__ void reverseArrayBlock(int *d_out, int *d_in) {
 extern __shared__ int s_data[];
int inOffset = blockDim.x * blockIdx.x;
int in = inOffset + threadIdx.x;

// Załadowanie jednego elementu z pamięci globalnej i zapamiętanie
// w odwrotnej kolejności w pamięci współdzielonej
s_data[blockDim.x - 1 - threadIdx.x] = d_in[in];

// Zaczekaj aż wszystkie wątki w bloku dokonają zapisu
// do pamięci wspólnej
__syncthreads();

// Zapisz dane z pamięci wspólnej w kolejności normalnej
// do pamięci globalnej
int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
int out = outOffset + threadIdx.x;
d_out[out] = s_data[threadIdx.x];
}
```

Kody błędów CUDA

- Wszystkie funkcje API, za wyjątkiem wywołań jąder zwracają kody błędów.
  - Wyrażenie typu `cudaError_t`

- Funkcja `cudaError_t cudaGetLastError(void)` zwraca kod ostatniego błędu

- char* `cudaGetErrorString(cudaError_t code)` zwraca ciąg znaków zakończony zerem zawierający opis błędu w języku angielskim

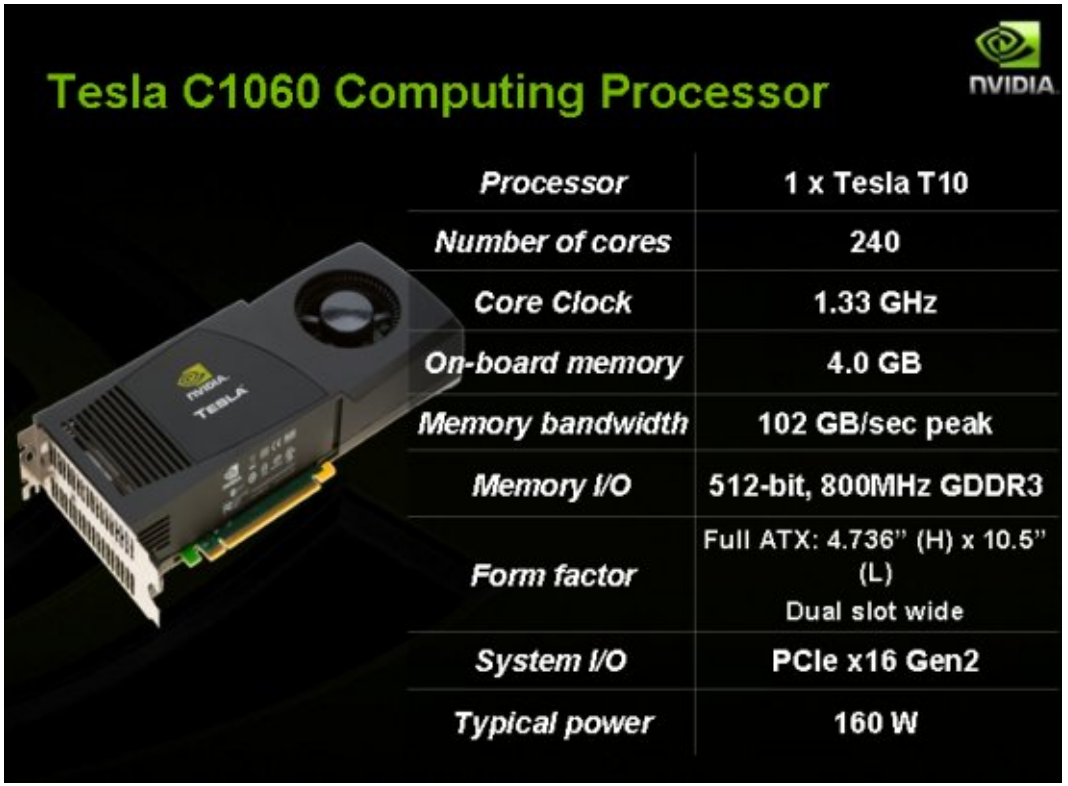
- `printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );`

Operacje atomowe i funkcje wbudowane

- Operacje atomowe na danych typu int/unsigned w pamięci globalnej.
  - add, sub, min, max
  - and, or , xor
  - inc, dec
- Wbudowane funkcje matematyczne na liczbach pojedynczej precyzji.
  - np. sinf/expf/logf
- Szczegóły w dokumentacji firmowej.

Hardware

- Geforce 8 i nowsze z co najmniej 256 VRAM (w tym karty w laptopach)
- Specjalizowany hardware: akceleratory TESLA.



TESLA S1070: Serwer w obudowie 1U z czterema akceleratorami

Jeżeli nie podoba nam się CUDA, to

- CUBLAS – podzbiór bibliotek BLAS operacje ma macierzach i wektorach

- CUFFT – szybka transformata Fourier'a (jednowymiarowa i dwuwymiarowa)

Podsumowanie

- Przykłady na wykładzie były bardzo „akademickie”
- Rzeczywista aplikacja powinna się charakteryzować:
  - Znaczną intensywnością korzystania z pamięci GPU w porównaniu z kopiami CPU<=>GPU.
  - Znaczną intensywnością obliczeń numerycznych w porównaniu z odwołaniami do pamięci
- Polecane źródła
  - Dokumentacja CUDA na stronie firmy NVIDIA: http://www.nvidia.com/object/cuda_develop.html
  - Strona http://gpgpu.org/developer ze slajdami z tutorialami na temat CUDA prezentowanymi na różnych konferencjach.
  - Strona przygotowana przez firmę NVIDIA z materiałami szkoleniowymi na temat CUDA, gorąco polecam sprawdzić linki do cyklu artykułów w Dr. Dobb's Journal. http://www.nvidia.com/object/cuda_education.html (zaczerpnąłem stamtąd przykład z odwracaniem kolejności elementów wektora)