Programowanie kart graficznych

Sławomir Wernikowski

swernikowski@wi.zut.edu.pl

Wykład #1:

Łagodne wprowadzenie do programowania w technologii NVIDIA CUDA

Terminologia:

Co to jest **GPGPU**?

General-**P**urpose computing
on a **G**raphics **P**rocessing **U**nit

wykorzystanie sprzętu „graficznego” (specjalizowanego)
do „niegraficznych” obliczeń (ogólnego przeznaczenia)

Terminologia:

Co to jest **CUDA**?

Compute **U**nified **D**evice **A**rchitecture

architektura sprzętowo-programowa ukierunkowana na przetwarzanie typu „data parallel”

Krótki rys historyczny:

2006:

Pierwsze implementacje pod postacią kart graficznych PCI-E z zabudowanymi dedykowanymi układami VLSI → GeForce 8

Krótki rys historyczny:

2006:

Kolejne implementacje pod postacią kart pozbawionych części generującej sygnał wizyjny
→ NVIDIA Tesla Personal Supercomputer (Dell, Lenovo)

Krótki rys historyczny:

CC = Compute Capability:

1.0 – wczesne konstrukcje architektury CUDA, implementowane przez układy GPU serii 8800; dla przykładu, maksymalna liczba równolegle pracujących wątków jest tu równa 768, a każdy z multiprocesorów dysponuje 8192 rejestrami o szerokości 32 bitów

Krótki rys historyczny:

CC:

1.1 – implementowane w układach GPU serii 1x0; od 1.0 odróżnia się możliwością kodowania funkcji stałoprzecinkowych operujących na danych o szerokości 32 bitów, przechowywanych w pamięci urządzenia

Krótki rys historyczny:

CC:

1.2 – powiązane z układami GPU serii GT 2x0; maksymalna liczba jednocześnie uruchomionych wątków jest tu równa 1024, a liczba rejestrów dostępnych dla multiprocesorów wynosi 16384

Krótki rys historyczny:

CC:

1.3 – wersja rozwojowa w stosunku do CC 1.2; jako pierwsza wprowadziła do architektury CUDA arytmometr zmiennoprzecinkowy dla danych podwójnej precyzji; implementowana w układach serii GTX 2x0

Krótki rys historyczny:

CC:

2.0 – implementowana w układach serii 4x0; maksymalna liczba wątków to 1536 przy liczbie dostępnych rejestrów równej 32768

Krótki rys historyczny:

CC:

3.0 – maksymalna liczba wątków: 2048

Krótka charakterystyka

Rdzeń

- jednostka przetwarzająca zawiera pewną liczbę procesorów, określanych w tej technologii mianem **rdzeni** (ang. *cores*)

- liczba rdzeni zmienia się w różnych wykonaniach kart CUDA w zależności od pożądanych cech urządzenia i docelowej grupy przyszłych użytkowników oraz aktualnego zaawansowania technologii – wczesne karty serii 8800GS zawierały 64 rdzenie, urządzenia serii 8800 GTS – 128 rdzeni, a 8800 Ultra – 1512 rdzeni; dla porównania karta serii NVIDIA Tesla S2050 zawiera 1792 rdzenie;

Pamięć

- wszystkie rdzenie korzystają ze swobodnego dostępu do pamięci (w oryginale: *device memory*), która funkcjonuje niezależnie od pamięci operacyjnej komputera macierzystego (w oryginale: *host memory*)

- pamięć karty CUDA posługuje się własną przestrzenią adresową; wielkość dostępnej pamięci zmienia się w zależności od wykonania i w kartach 8800 GTS sięgała 1940 MB, 8800 GTX – 1800 MB, 8800 Ultra – 2160 MB i 24576 MB na karcie NVIDIA Tesla S2050;

Komunikacja

- interfejs programowy architektury CUDA dostarcza wydajnych usług transferu bloków danych do i z pamięci operacyjnej hosta np. poprzez funkcje:

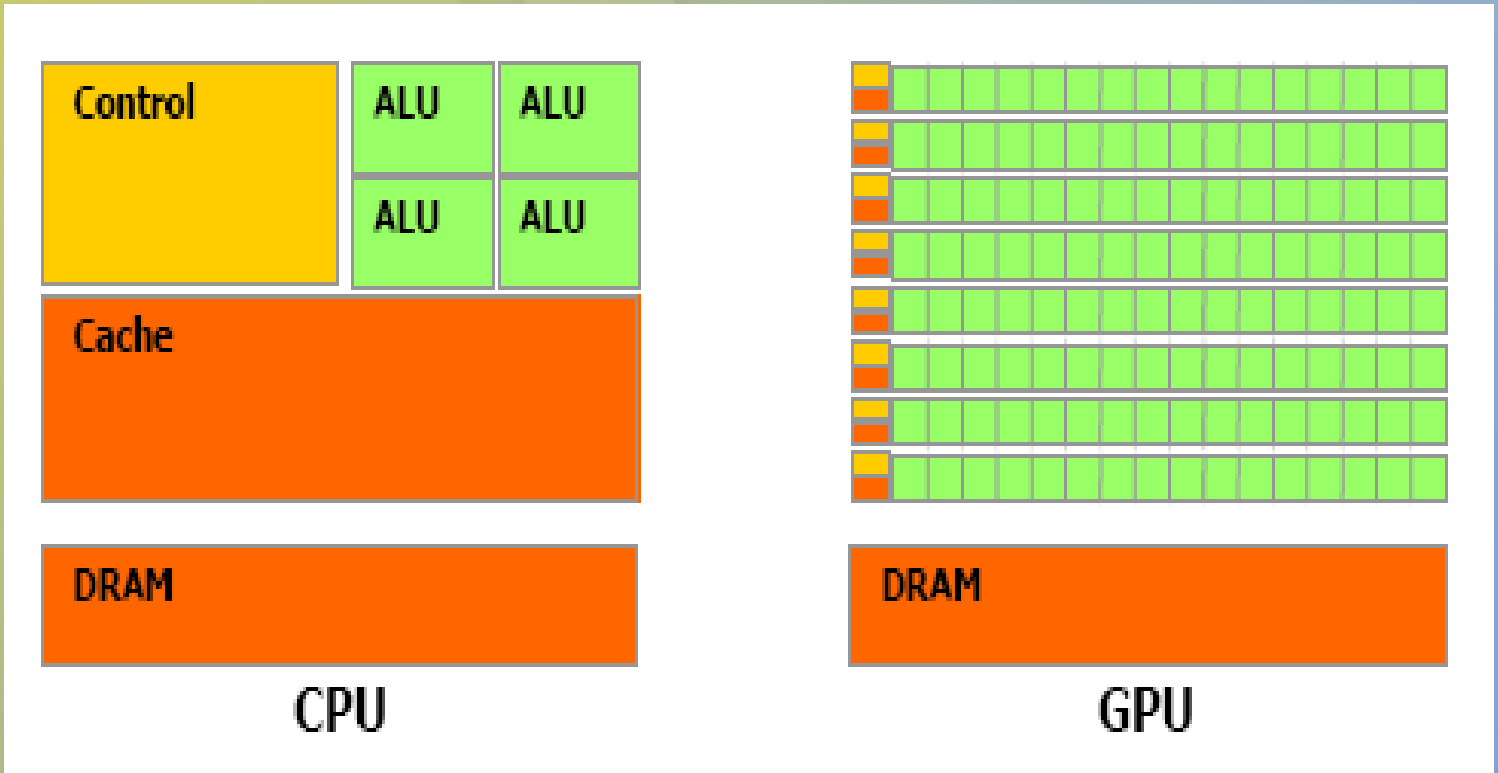
  `cudaMemcpy()`
  `cudaMemcpy2D()`
  `cudaMemcpy3D()`

- istnieje inny sposób komunikacji z pamięcią hosta, w szczególności kod wykonywany na rdzeniu GPU może mieć możliwości sięgania do danych znajdujących się poza własną przestrzenią adresową karty (o tym przy okazji)

Pamięć dzielona

- wszystkie rdzenie mają dostęp do szybkiej pamięci lokalnej (niezależnej funkcjonalnie od reszty pamięci operacyjnej karty), która może być wykorzystywana jako pamięć podręczna dla danych wykorzystywanych szczególnie intensywnie przez kod wykonywany przez rdzenie; typowy rozmiar tej pamięci to 16KB

Jak to wygląda?



API

- środowisko programisty definiowane przez CUDA SDK dostarcza zestawu środków służących do zarządzania pamięcią urządzenia, a w szczególności do:
  - przydzielania bloku pamięci (**`cudaMalloc()`**, **`cudaMallocPitch()`**, **`cudaMalloc3D()`**)
  - zwalniania bloku pamięci (**`cudaFree()`**);
  - słowo kluczowe **`__shared__`**, stanowiące część rozszerzonego dialektu języka C używanego w implementacji CUDA SKD, służy do deklarowania danych przeznaczonych do przechowywania w szybkiej pamięci dzielonej

Kod źródłowy

- kod wynikowy, przeznaczony do wykonania na rdzeniach architektury CUDA w chwili obecnej uzyskuje się na dwa sposoby:
  - poprzez wykorzystanie języka niskiego poziomu (asemblera), odwzorowującego listę rozkazów architektury (język **PTX** ang. *Parallel Thread Execution*)
  - poprzez wykorzystanie języków wysokiego poziomu, rozbudowanych - bądź to na poziomie składni, bądź poprzez wykorzystanie zewnętrznych bibliotek - o konstrukcje umożliwiające jawne kontrolowanie zachowania rdzeni

Kod źródłowy

- w typowym przypadku z jednego, wspólnego kodu źródłowego uzyskuje się dwuczęściowy kod wykonywalny:
  - jedna z części przeznaczona jest do uruchomieniu w środowisku hosta (tu znajduje się prolog i epilog programu, zawierający m.in. inicjację sprzętu karty)
  - druga część zawiera kod przeznaczony do wykonania na rdzeniach urządzenia CUDA;
  - chwili obecnej NVIDIA dostarcza środowiska typu SDK dla języków C/C++ i Fortran oraz platform MS Windows i Linux

Skalowalność

- interfejs programowany, udostępniany przez architekturę CUDA, gwarantuje, że przy **umiejętnym** wykorzystaniu jego udogodnień kod wynikowy jest w pełni **skalowalny**

- oznacza to między innymi, że może być **bez konieczności powtórnej kompilacji** wykonywany na „silniejszych” (w tym również jeszcze nie istniejących) modelach kart CUDA z pełnym wykorzystaniem przyrostu liczby dostępnych rdzeni;

Dialekt CUDA C/C++

- dialekt języka C, wykorzystywany do kodowania programów dla rdzeni CUDA, jest podzbiorem języka wzorcowego, a najważniejsze nałożone ograniczenia to:
  - niemożność wykorzystywania rekurencji
  - wykluczenie możliwości posługiwania się wskaźnikami do funkcji oraz funkcjami ze zmienną liczbą parametrów
  - implementacja arytmetyki zmiennopozycyjnej w architekturze CUDA odbiega od definicji zawartej w normie IEEE 754

Rozszerzenia

- dialekt języka CUDA C zawiera także kilka rozszerzeń, w szczególności dodatkowych deklaratorów i specyfikatorów, pozwalających wprowadzać do kodu źródłowego byty swoiste dla środowiska CUDA;

- ocenę elegancji tych rozszerzeń pozostawimy programistom...

Host vs device

- kod wynikowy CUDA jest wykonywany dopiero na żądanie emitowane z kodu hosta; kod CUDA wykonywany jest na wszystkich rdzeniach **jednocześnie**

Wątki

- kod wykonywany indywidualnie na pojedynczym rdzeniu nosi nazwę wątku (ang. *thread*);
- środki dostępne w środowisku CUDA pozwalają wątkowi na dokładne **zidentyfikowanie** rdzenia, na którym jest wykonywany, co z kolei umożliwia różnicowanie zachowania algorytmu w zależności od np. rozmiaru danych lub liczby dostępnych rdzeni;
- taka filozofia przetwarzania pozwala zakwalifikować architekturę CUDA do paradygmatu przetwarzania strumieniowego;

Synchronizacja

- środowisko CUDA nie zapewnia żadnych mechanizmów synchronizacji poza elementarnym mechanizmem bariery, zatrzymującej wykonanie wątku do momentu osiągnięcia tej bariery przez wątki na wszystkich rdzeniach;
- operację ten wykonuje się wywołując bezparametrową funkcję usługę środowiska CUDA o nazwie `__syncthreads();`
- jakiekolwiek inne mechanizmy synchronizacji wątków muszą zostać zaimplementowanie przez programistę z wykorzystaniem np. dostępu do pamięci dzielonej wątków.

Kod wątków

- tekst źródłowy kodu przeznaczony do uruchomienia w ramach wątku musi zostać jawnie „spreparowany” przez programistę;
- na poziomie języka C/C++ dokonuje się tego poprzez wydzielenie w kodzie kompletnej funkcji opatrzonej specyfikatorem `__global__` (jest to słowo kluczowe dialektu CUDA)
- wymaga się, aby funkcja taka była zadeklarowana jako nie zwracająca wyniku (`void`);
- funkcje tego rodzaju w terminologii dokumentów NVIDIA SDK nazywa się mianem ***kernel***.

Kod wątków

- każdy kernel, zadeklarowany w kodzie źródłowym, stanie się treścią (tekstem) wszystkich wątków uruchomionych równolegle na każdym z procesorów architektury CUDA

- uruchomienie wątków następuje w wyniku wywołania funkcji deklarowanej dla każdego kernela,

Odpalenie wątku

- wywołanie funkcji wątku przybiera w dialekcie CUDA C postać poniższą*

```
kernName<<<blocksNum,threadsPerBlocks>>>(par1,par2);
```

*) *podano jeden z prostszych wariantów*

Odpalenie wątku

```
kernName<<<blocksNum,threadsPerBlocks>>>(par1,par2);
```

gdzie:

- kernelName jest nazwą funkcji kernela zadeklarowanej z zasięgu osiągalnym z miejsca wywołania

Odpalenie wątku

```
kernName<<<blocksNum,threadsPerBlocks>>>(par1,par2);
```

gdzie:

- blocksNum jest wyrażeniem typu int, określającym liczbę tzw. bloków wątków, które będą uruchamiane kolejno w celu odpalenia (ang. *launch*) żądanej liczby wątków

Odpalenie wątku

```
kernName<<<blocksNum,threadsPerBlocks>>>(par1,par2);
```

gdzie:

- threadsPerBlocks jest wyrażeniem typu int, określającym liczbę wątków, które zostaną odpalone wewnątrz każdego z bloków; wymaga się, aby liczba ta nie przewyższała liczby procesorów dostępnych w konkretnym środowisku

Odpalenie wątku

```
kernName<<<blocksNum,threadsPerBlocks>>>(par1,par2);
```

gdzie:

- par_i są parametrami aktualnymi o liczbie i typach określonych w deklaracjach parametrów formalnych zdefiniowanych w nagłówku wywoływanego kernela.

Odpalenie wątku

Formalizm ten daje prosty i efektywny sposób **samoskalowania** się kodu w zależności od architektury docelowej i liczby dostępnych procesorów.
Jeśli rozwiązanie pewnego problemu wymaga równoległego wykonania N wątków, a architektura docelowa udostępnia M procesorów, można założyć, że:

```
threadsPerBlock=M
blocksNum=(N+threadsPerBlock-1)/threadsPerBlock
```

Odpalenie wątku

UWAGA:

- **wszystkie wątki w bloku wykonują się równolegle**
- **wszystkie bloki wątków wykonują się sekwencyjnie**

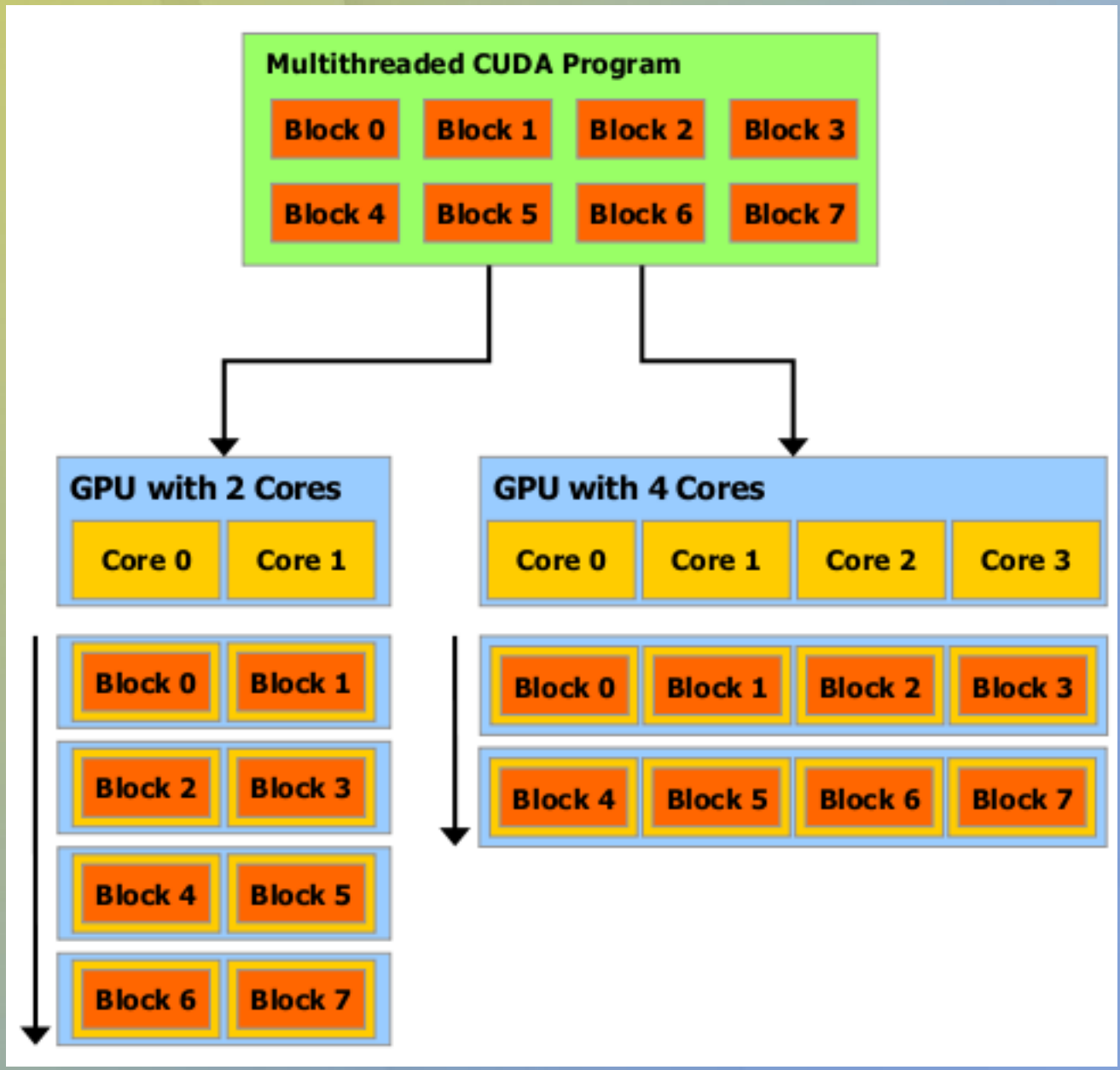
Co ma wątek?

Odpalony wątek ma możliwość ustalenia:

- w ramach którego z bloków pracuje
- na którym procesorze w bloku został odpalony

to daje możliwość zbudowania takiego kodu kernela, który będzie modyfikował swoje działanie np. w zależności od momentu jego uruchomienia

Skalowalność



Trywialny przypadek

- wymagana liczba wątków nie dzieli się bez reszty przez wyznaczoną liczbę bloków
- część wątków w ostatnim bloku powinna pozostać bezczynna, tzn. zakończyć pracę natychmiast po odpaleniu.

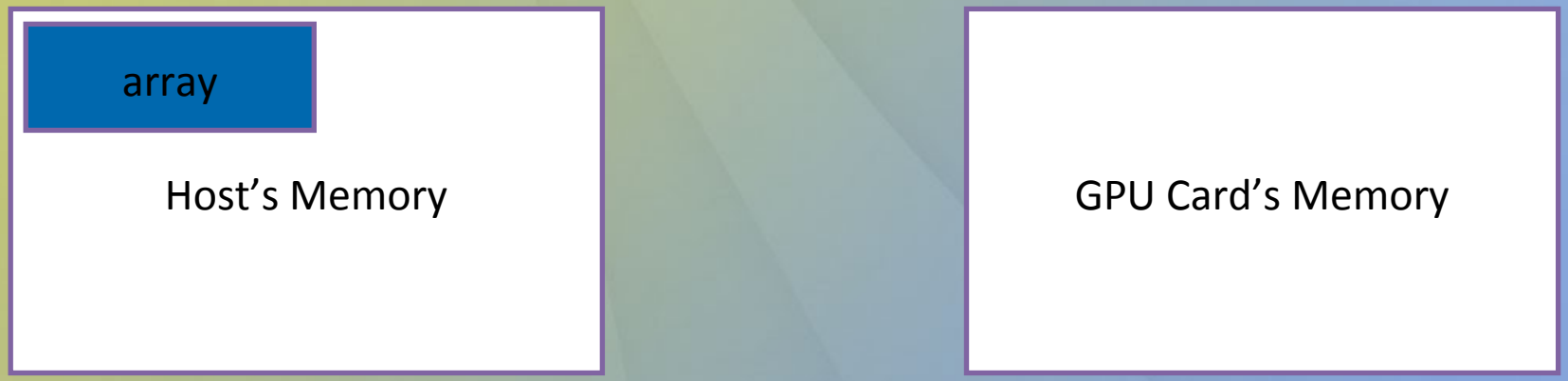
Rozwiązanie

- predefiniowane zmienne (struktury): blockIdx i threadIdx
- ilustracja sposobu, w jaki wątek zabezpiecza się przed odpaleniem swojego kodu na rzecz nieistniejącego elementu tablicy T[N]:

```
if(blockIdx.x * threadsPerBlock + threadIdx.x < N)
{
   // kod przetwarzający element tablicy
}
```

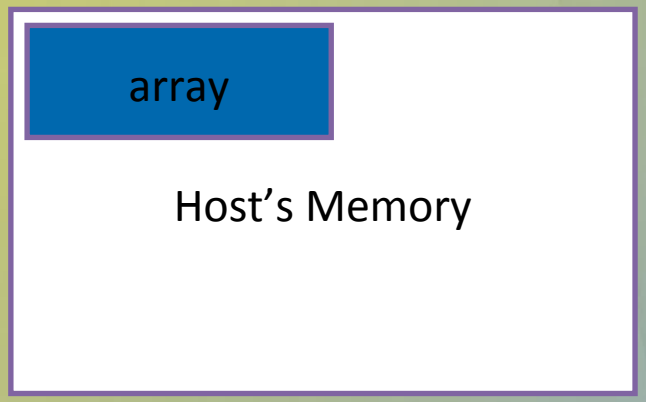
W praktyce – krok #1

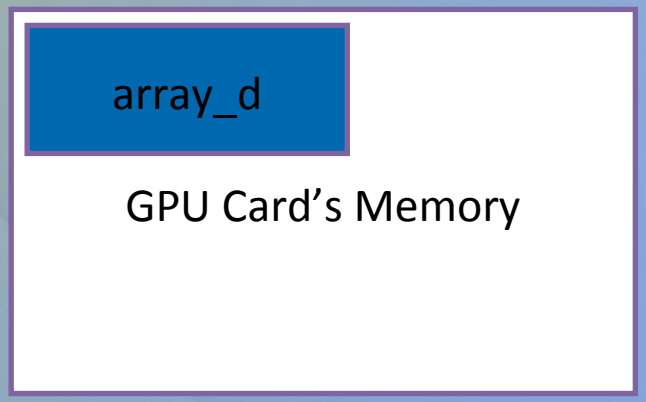
I stała się jasność...



W praktyce – krok #2

rezerwujemy obszar w pamięci urządzenia



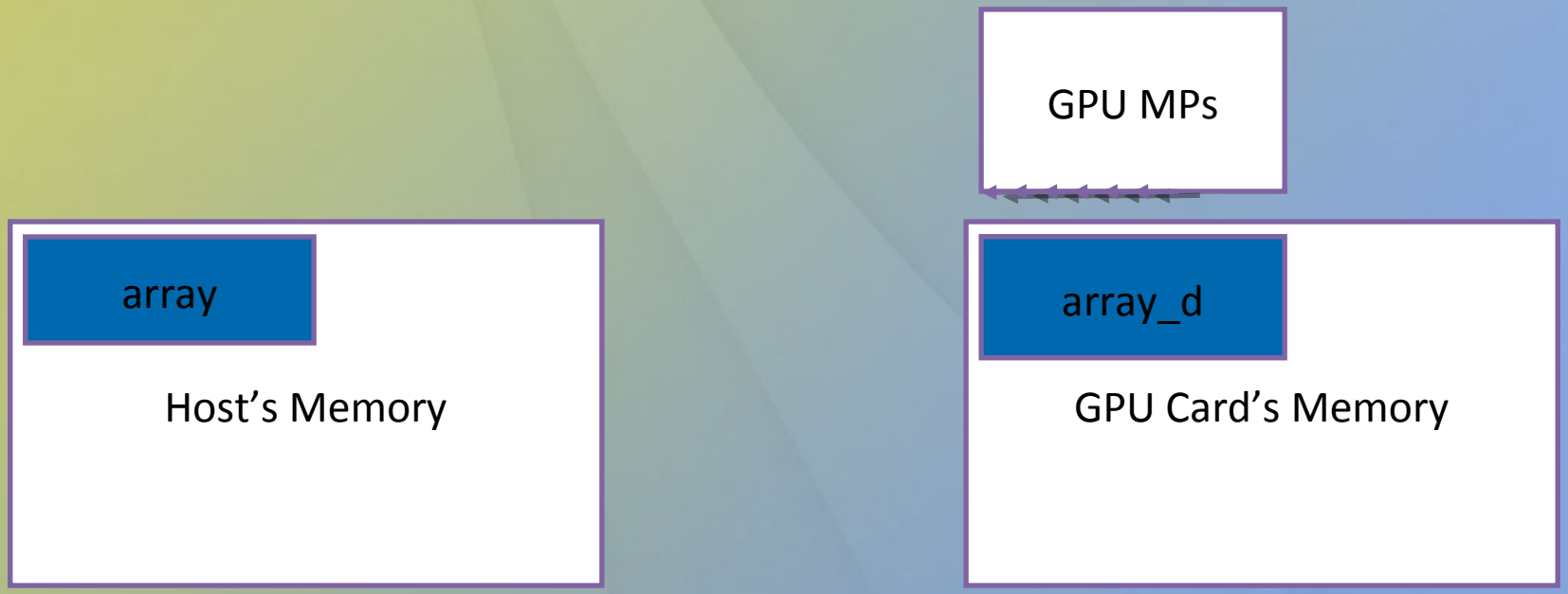


W praktyce – krok #3

Transferujemy dane...

W praktyce – krok #4

Odpalamy obliczenia...

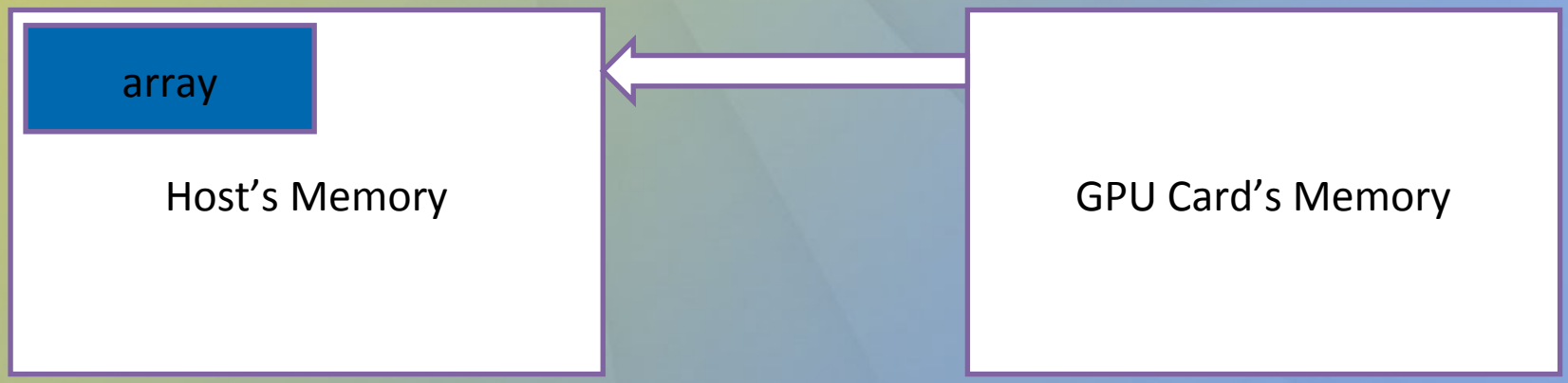


W praktyce – krok #5

Transferujemy wyniki

W praktyce – krok #5

Zwalniamy pamięć...



Przykład #1

```
// dodawanie wektorów

__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
:
:
int main()
{
    :
    :
    VecAdd<<<1, N>>>(A, B, C);
}
```

Przykład #2

```
// dodawanie macierzy

__global__ void MatAdd
(float A[N][N], float B[N][N],float C[N][N])
{
   int i = threadIdx.x;
   int j = threadIdx.y;
   C[i][j] = A[i][j] + B[i][j];
}
int main()
{
   :
   :
//1 blok złożony z N * N * 1 wątków
   int numBlocks = 1;
   dim3 threadsPerBlock(N, N);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```