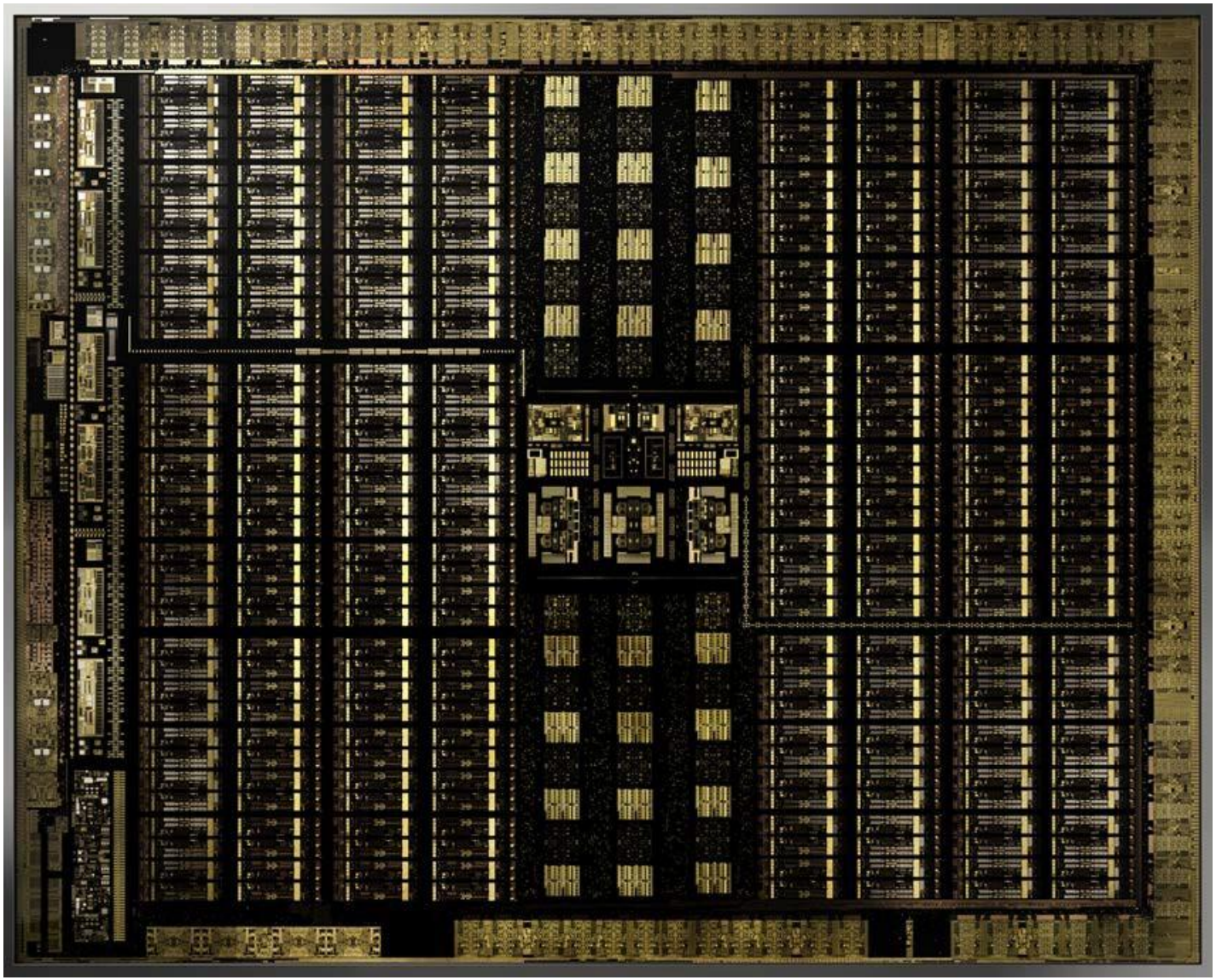
Platforma sprzętowa

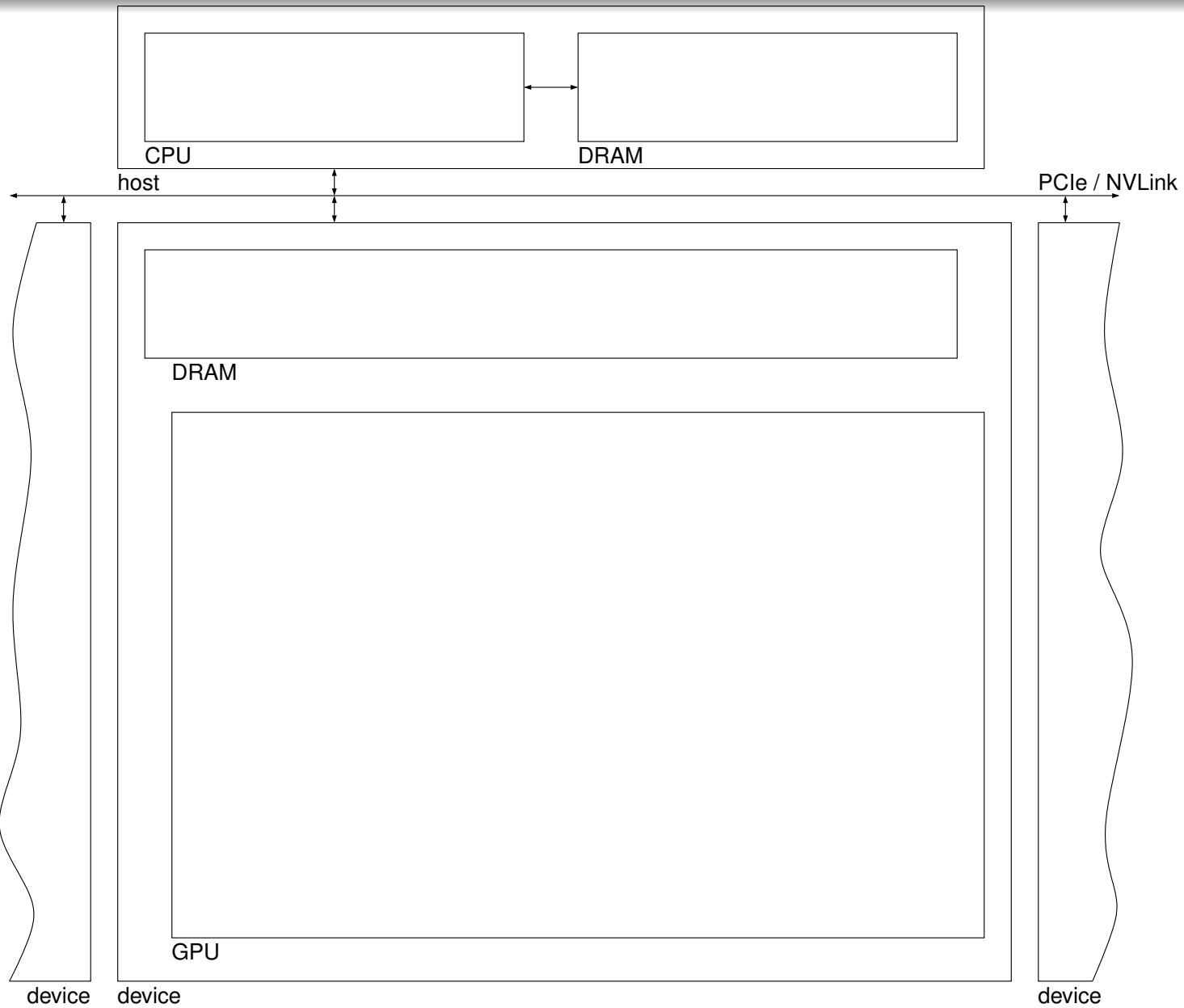


https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth

Model platformy sprzętowej

- Platforma sprzętowa CUDA składa się z gospodarza (*host*) i jednego lub więcej poddanych mu urządzeń (*device*).
- Gospodarzem jest procesor CPU wyposażony w pamięć.
- Urządzeniem jest procesor GPU wyposażony w pamięć.

Model platformy sprzętowej



Model platformy sprzętowej a hierarchia urządzenia

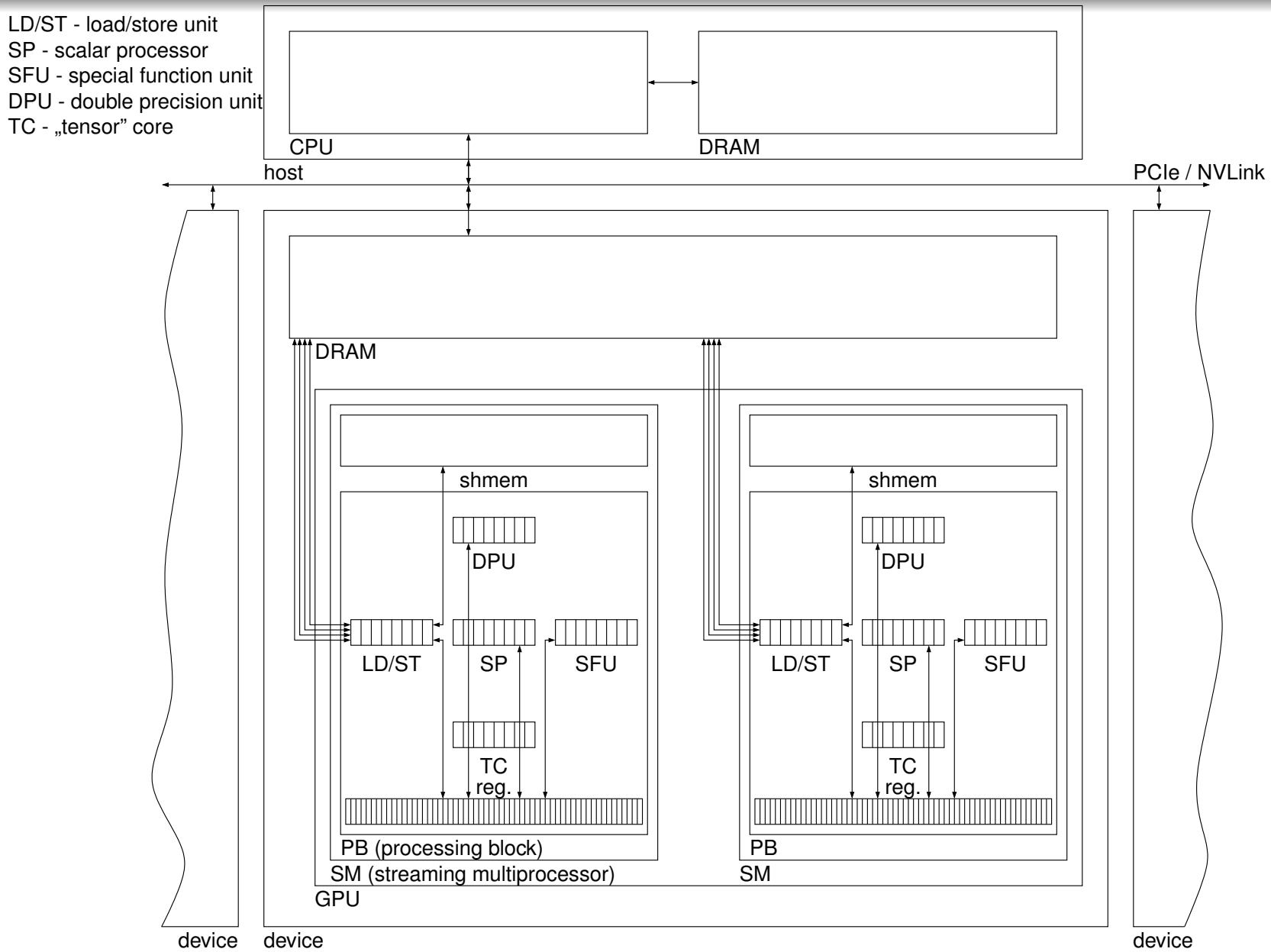
Procesor urządzenia (GPU) jest zbudowany hierarchicznie z wieloprocesorów strumieniowych (SM/SMX/SMM/SM – *streaming multiprocessor*). Wersja wieloprocesora decyduje o możliwościach obliczeniowych (C_c – *compute capability*).

- Wieloprocesor zawiera pamięć (współdzieloną i cache L1) oraz 1/2/4 bloki przetwarzania (PB – *procesing block*).
  - Blok przetwarzania składa się z kilku grup podukładów:
    - procesorów skalarnych (SP – *scalar processor*), czyli jednostek ALU FP32/INT32, zwanych rdzeniami CUDA* (obecnie jednostki ALU INT32 tworzą oddzielne rdzenie),
    - jednostek funkcji specjalnych (SFU – *special function unit*),
    - jednostek podwójnej precyzji (DPU – *double precision unit*),
    - rdzeni „tensorowych” (TC – *tensor core*),
    - jednostek transferu danych (LD/ST – *load/store unit*),
    - rejestrów i pamięci cache L0.

    Każda z tych grup jednostek pracuje równolegle w trybie SIMD (*Single Instruction, Multiple Data*) [M.J. Flynn, 1966].

*Można też rdzeniami nazywać układy SM albo PB.

Hierarchia urządzenia



Hierarchia urządzenia – SM dla procesorów GA10x

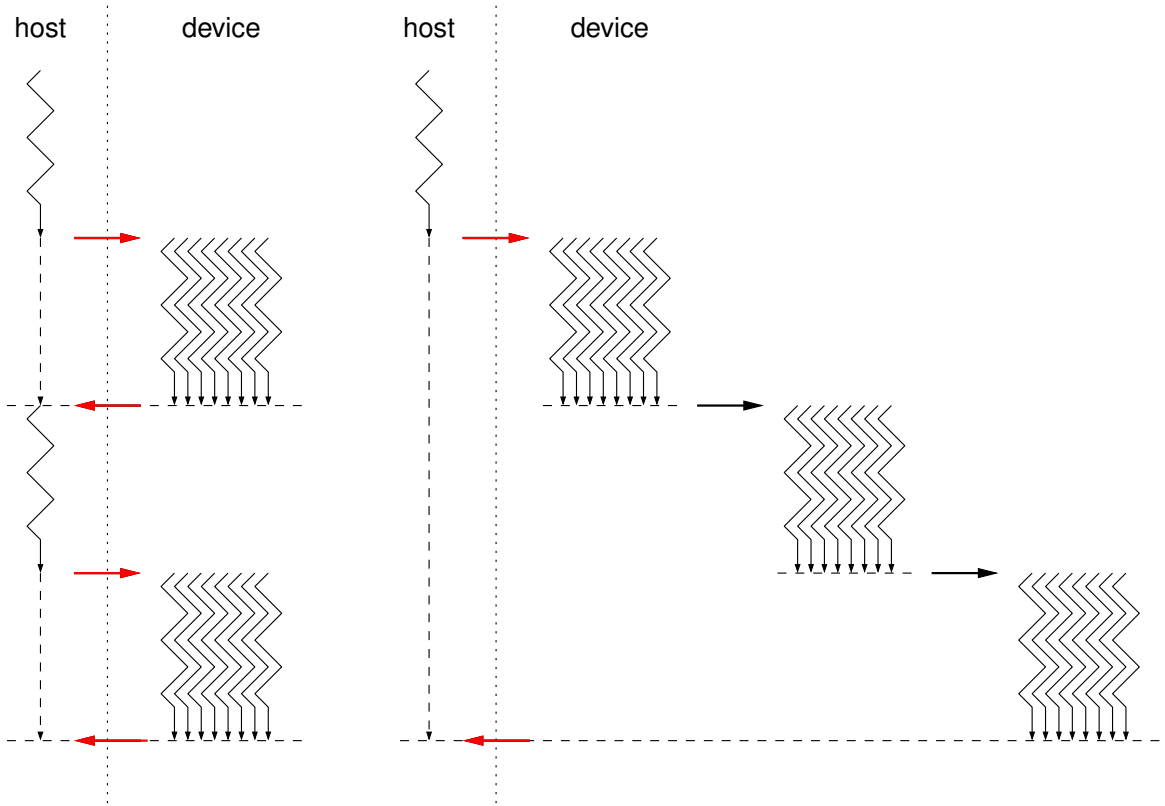
https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

Podział aplikacji między gospodarza a urządzenia

- Funkcje procesora gospodarza (nadrzędnego):
  - dostarczanie danych i programów do urządzeń
  - odbieranie wyników pracy od urządzeń
  - zarządzanie urządzeniami i ich synchronizacja
  - ew. wykonywanie sekwencyjnej części algorytmu
- Funkcją urządzenia (koprocesora) jest realizacja równoległej części algorytmu.
  - Równoległość jest masowa i drobnoziarnista.
  - Każde „ziarno” realizowane jest przez oddzielny wątek.
  - Wątki są „lekkie” i powinno ich być bardzo dużo (miliony!).
- Wykonanie programu w ogólnym przypadku składa się z przeplatających się faz sekwencyjnych i równoległych.
- Program urządzeń jest wykonywany asynchronicznie względem gospodarza, zaś przesyłanie danych może być albo synchroniczne, albo asynchroniczne.

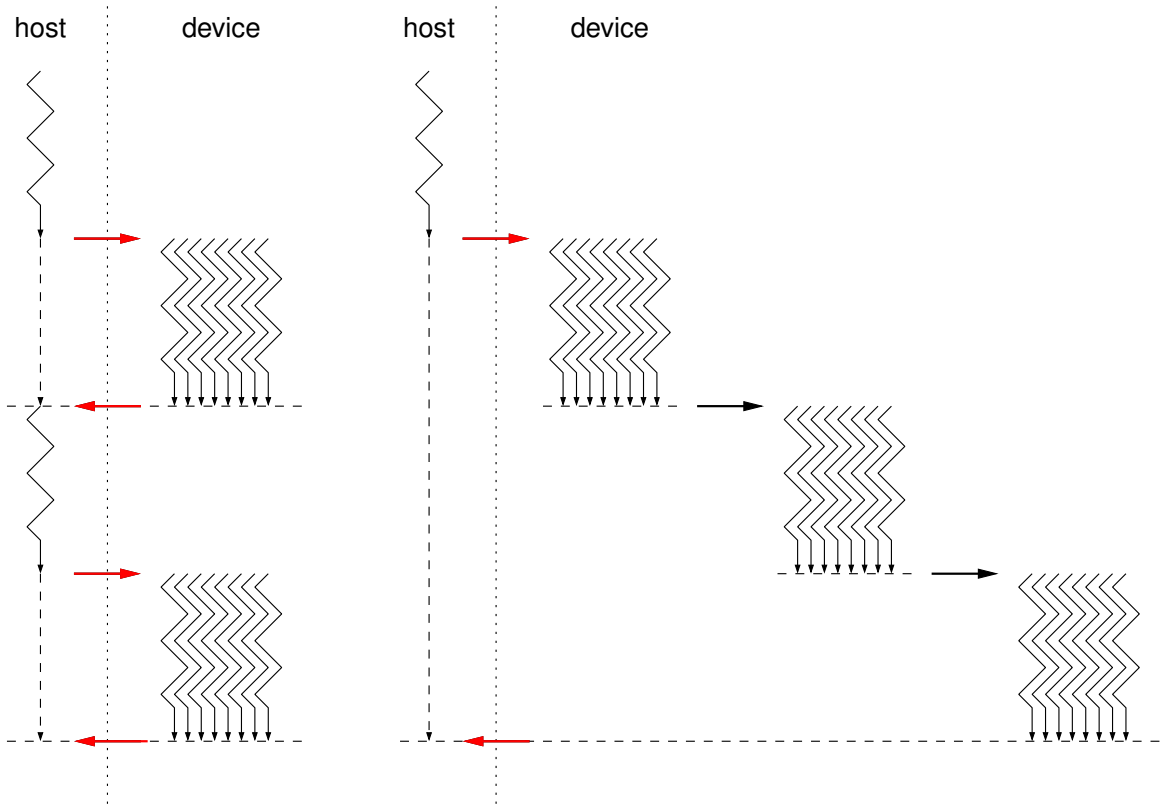
Wykonywanie wątków przez gospodarza i urządzenie

- Transfer danych/programu do/z GPU często jest głównym źródłem problemów z wydajnością (PCIe 4.0 $\xrightarrow{\times 16}$ 32 GiB/s).
- Nowsze procesory GPU ($C_c \geqslant 3.5$) mogą samodzielnie uruchamiać kolejne funkcje jądra (*Dynamic Parallelism*$^{3.5}$).
- Procesorowi GPU trzeba dać dostatecznie dużo pracy.



Wykonywanie wątków przez gospodarza i urządzenie

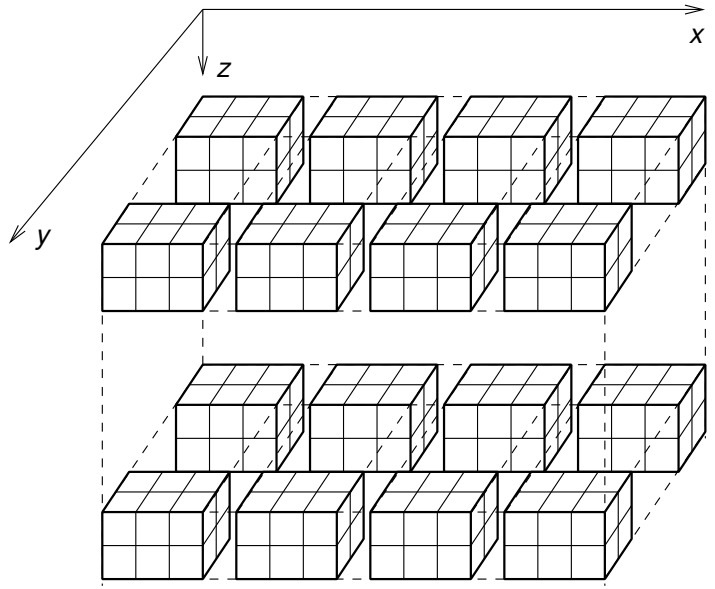
- W danej fazie każdy wątek urządzenia wykonuje tę samą funkcję jądra (*kernel*), parametryzowaną identyfikatorem.
- Zbiór wszystkich wątków utworzonych podczas danego uruchomienia (*launch*) jądra to sieć wątków (*grid*).
- Koniec pracy sieci – wszystkie jej wątki zakończyły się.



Hierarchia wątków (6D!)

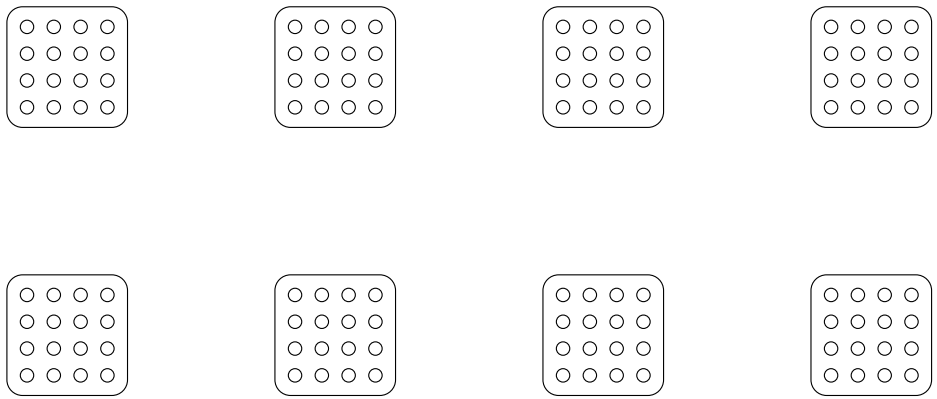
Wszystkie wątki sieci są wielopoziomowo zorganizowane.

- Sieć składa się z bloków (*block*) wątków ułożonych w 3D[2.0] kostkę o rozmiarach $B_{G,x} \times B_{G,y} \times B_{G,z}$ bloków, gdzie $B_{G,x} =$ `gridDim.x`... Wszystkie bloki sieci zawierają taką samą liczbę, tak samo zorganizowanych wątków.
- Blok składa się z wątków (*thread*) ułożonych w 3D kostkę o rozmiarach $T_{B,x} \times T_{B,y} \times T_{B,z}$ wątków, gdzie $T_{B,x} =$ `blockDim.x`... współpracujących
  - przez synchronizację na barierze (*barrier*) (od CUDA 9.0 *nie tylko* w ramach bloku, ale i w mniejszych i w większych[7.0] grupach wątków),
  - przez współdzieloną (*shared*) pamięć lokalną bloku (*tylko* w ramach bloku).



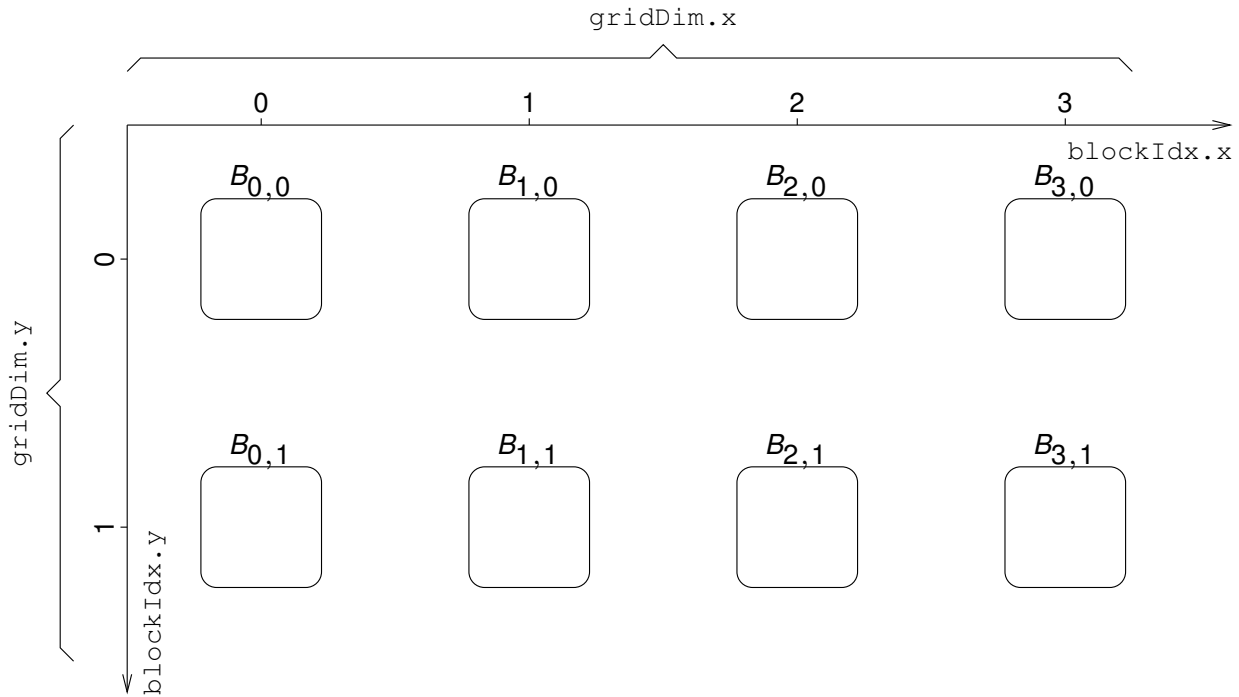
Hierarchiczne indeksowanie wątków – sieć

- Dla każdego wątku w sieci można jednoznacznie określić jego 6 współrzędnych (6D!), które parametryzują funkcję jądra. Indeksowanie jest dwupoziomowe (bloku i wątku).



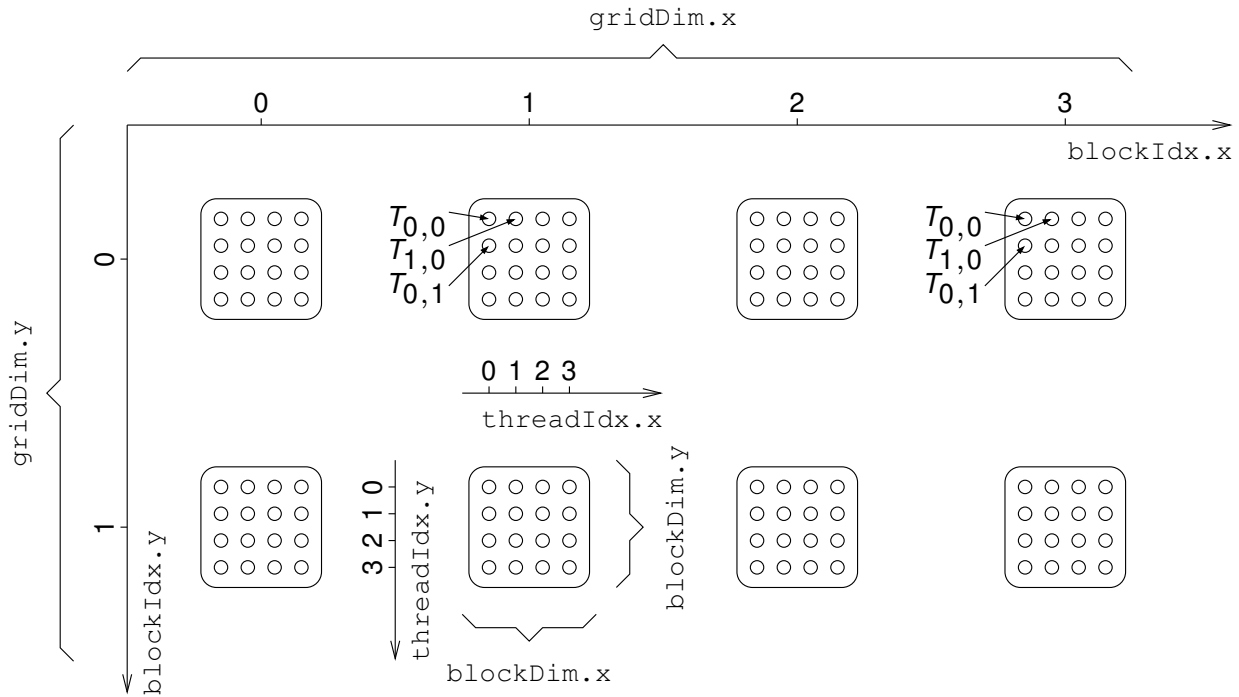
Hierarchiczne indeksowanie wątków – bloki

- Każdy blok wątków w sieci jest jednoznacznie identyfikowany przez swoje współrzędne `blockIdx.x`, `blockIdx.y` i `blockIdx.z` (liczone od 0).



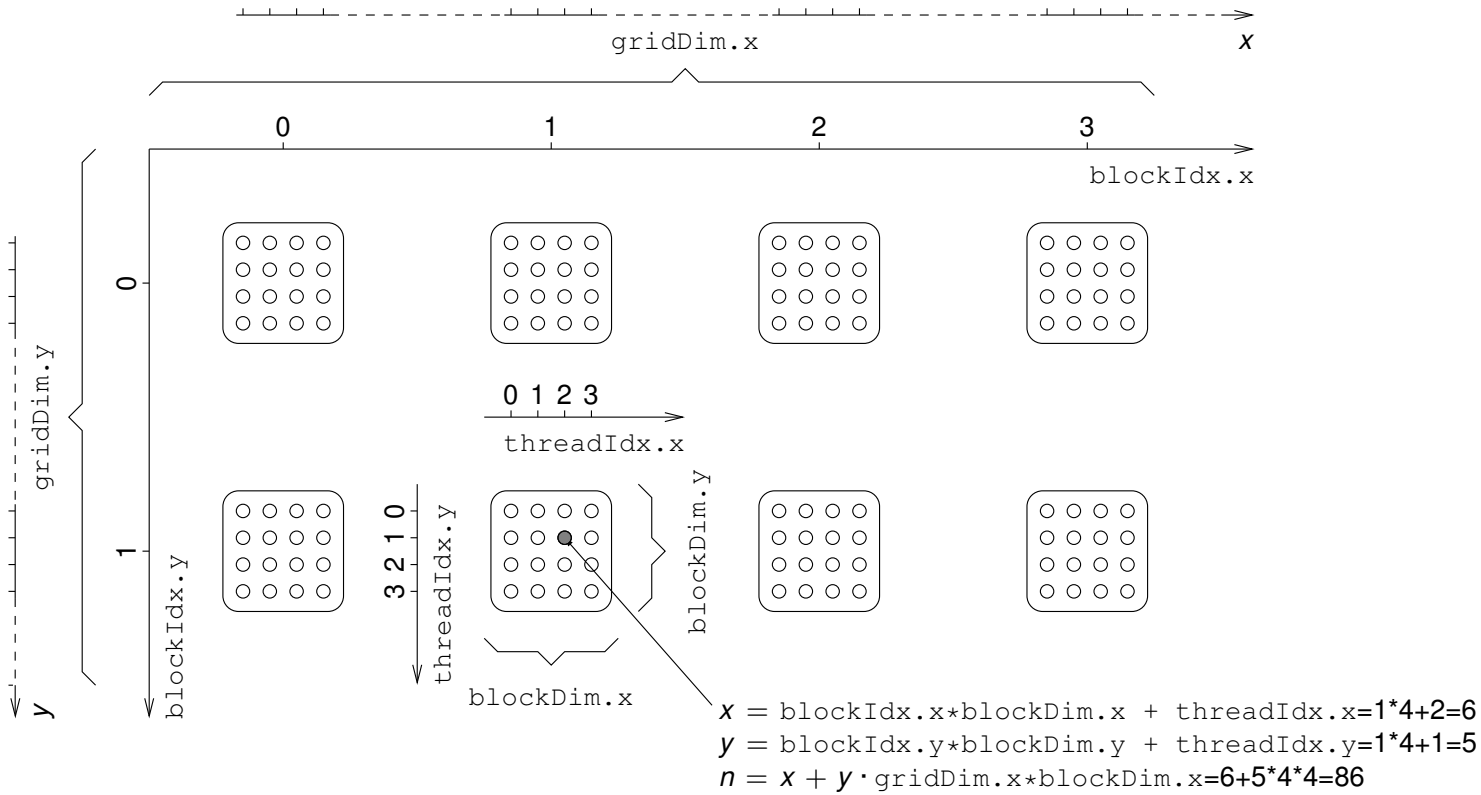
Hierarchiczne indeksowanie wątków – wątki

- Każdy wątek w bloku jest jednoznacznie identyfikowany przez swoje współrzędne `threadIdx.x`, `threadIdx.y` i `threadIdx.z` (liczone od 0).



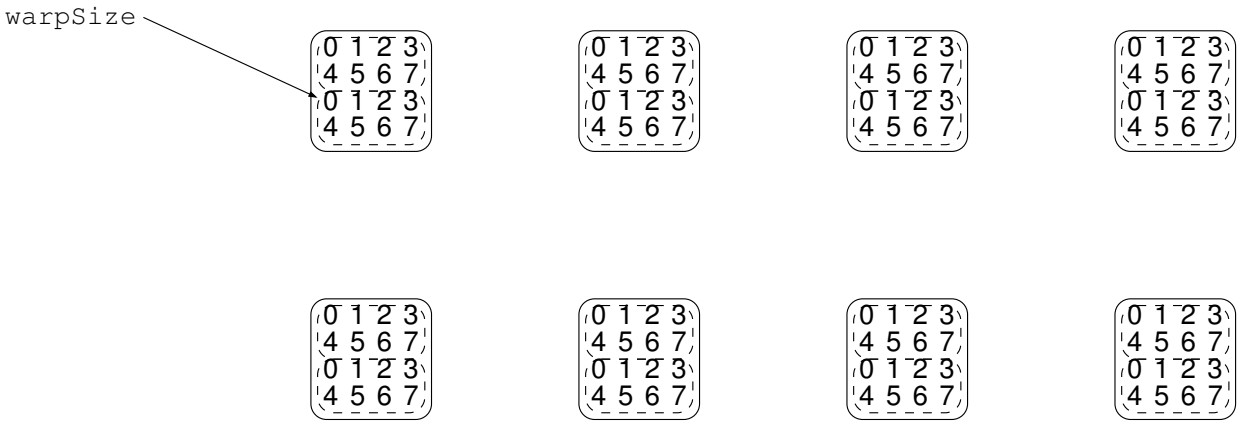
Hierarchiczne indeksowanie wątków – numeracja

- W ramach całej sieci dla każdego wątku można wyznaczyć jego współrzędne (x, y, z) oraz unikalny numer n. Dla przypadku dwuwymiarowego:



Hierarchiczne indeksowanie wątków – sploty

- W ramach bloku procesor tworzy wątki, przełącza je i wykonuje w nierozdzielnych grupach zwanych splotami (*warp*) o rozmiarze $T_W =$ `warpSize` $= 32$ (na rysunku 8).



Grupowanie i indeksowanie wątków

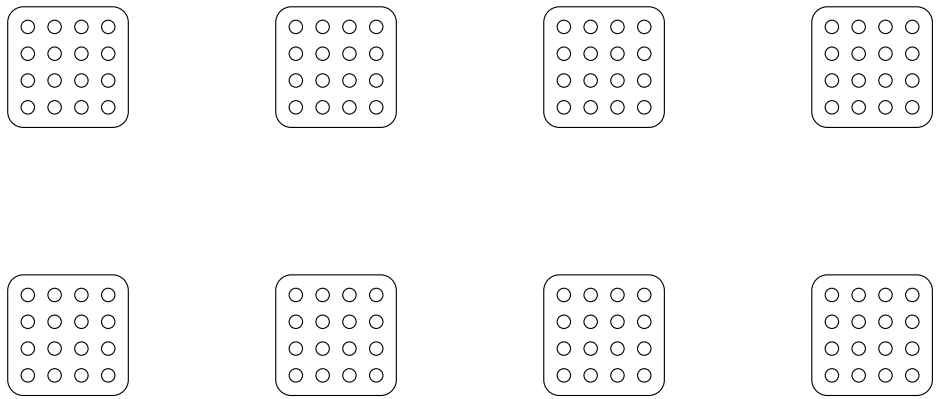
- W ramach zorganizowanej przez programistę grupy wątków (od CUDA 9.0) każdy wątek ma unikalny identyfikator dostępny przez metodę `thread_rank()`.
- W ramach bloku każdy wątek ma unikalny identyfikator (TID – *thread ID*), równy

```
tid = threadIdx.x + blockDim.x *
      (threadIdx.y + blockDim.y * threadIdx.z);
```

- Wątki w ramach splotu mają zawsze kolejne identyfikatory, liczone od 0 i równe `tid % warpSize`.
- Numer splotu zawierającego dany wątek w ramach bloku wynosi `tid / warpSize`.

Zanurzenie modelu wykonania w modelu platformy – 1

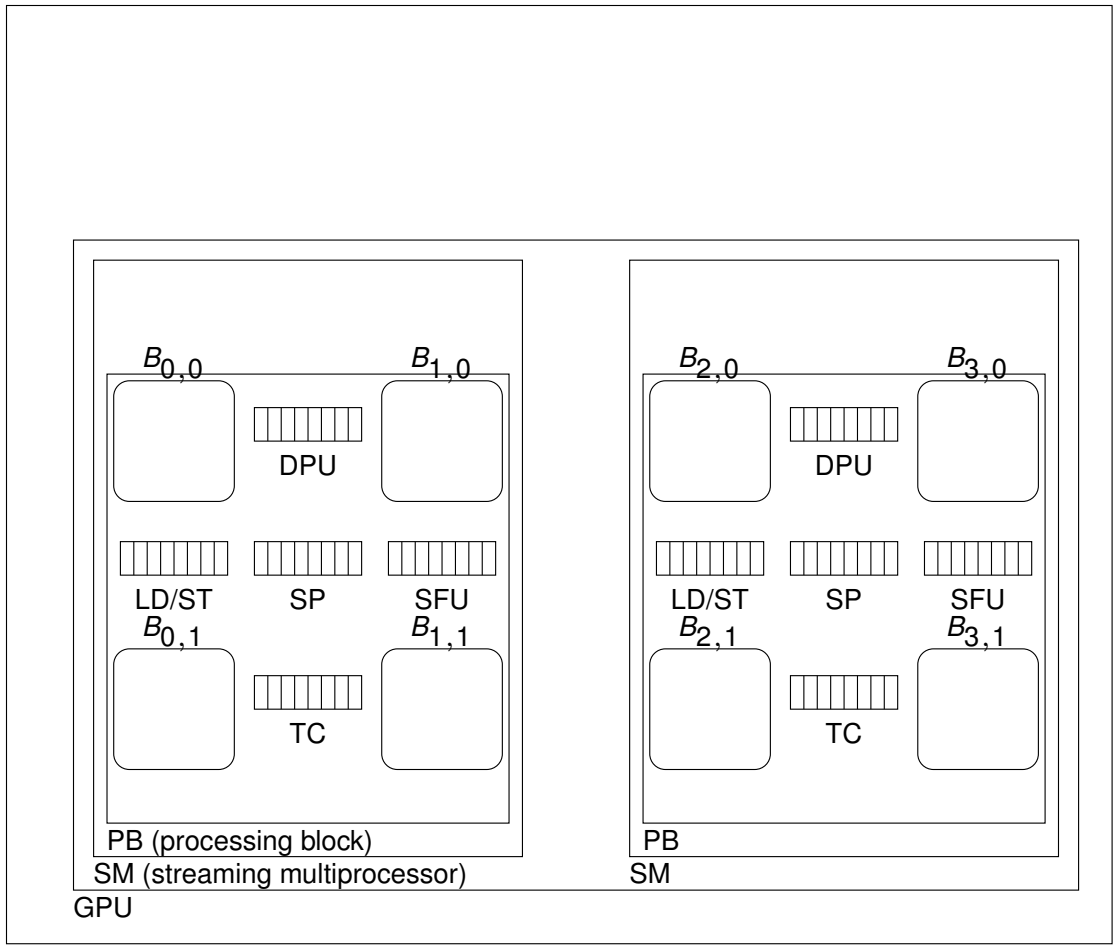
Aplikacja równoległa jest logicznie podzielona na regularną sieć bloków wątków. Najpoważniejsze ograniczenie modelu wykonania: brak możliwości bezpośredniej wymiany danych pomiędzy różnymi blokami – konieczne jest użycie operacji atomowych[1.1, 1.2, 2.0,...]. Do CUDA 9.0 nie było też możliwości bezpośredniej synchronizacji wątków pomiędzy blokami.



Zanurzenie modelu wykonania w modelu platformy – 2

LD/ST - load/store unit
SP - scalar processor
SFU - special function unit
DPU - double precision unit
TC - „tensor” core

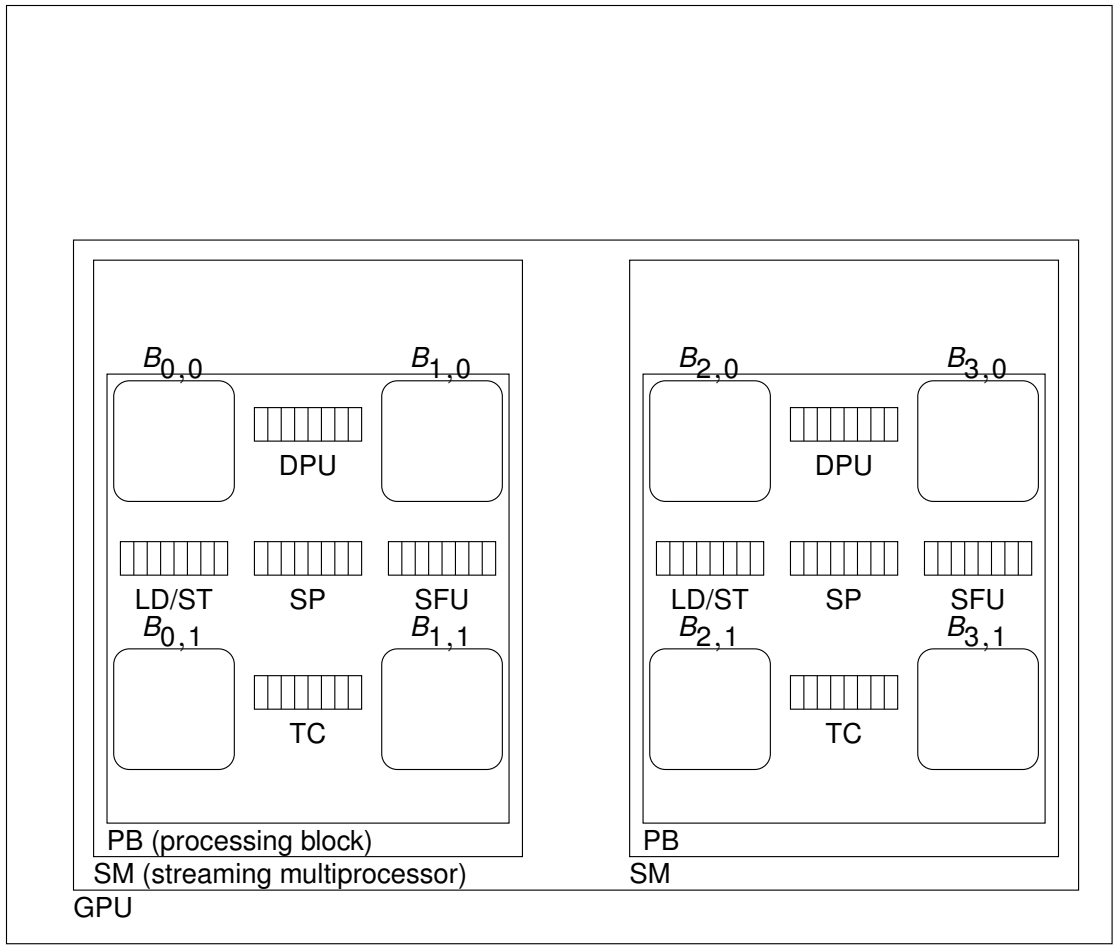
Każdy blok jest wykonywany przez jeden wieloprocesor, dzięki czemu jego wątki mogą ze sobą łatwo i efektywnie współpracować (synchronizacja, pamięć współdzielona). Zarządca GPU (*scheduler*) przydziela każdemu wieloprocesorowi zero lub więcej bloków do wykonania (zależnie od potrzeb i zasobów).



Zanurzenie modelu wykonania w modelu platformy – 3

LD/ST - load/store unit
SP - scalar processor
SFU - special function unit
DPU - double precision unit
TC - „tensor” core

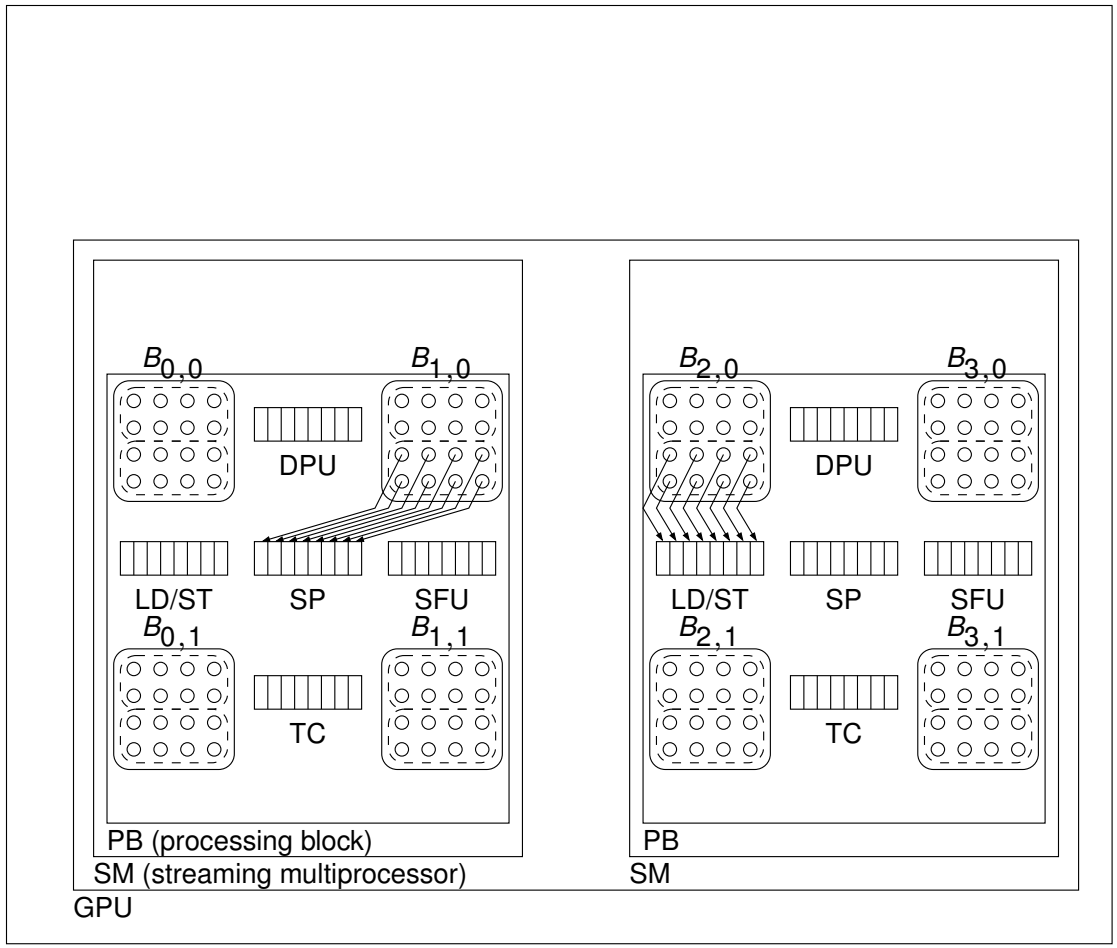
Podział aplikacji na bloki pozwala na całkowicie „przezroczystą” dla programisty skalowalność przy użyciu większej liczby i/lub lepszych (o większych zasobach) wieloprocesorów, także w przyszłości. Alokacja zasobów każdego SM jest wykonywana niezależnie, co bardzo ją upraszcza. Do zarządcy GPU wysyłana jest tylko informacja o zajętości.



Zanurzenie modelu wykonania w modelu platformy – 4

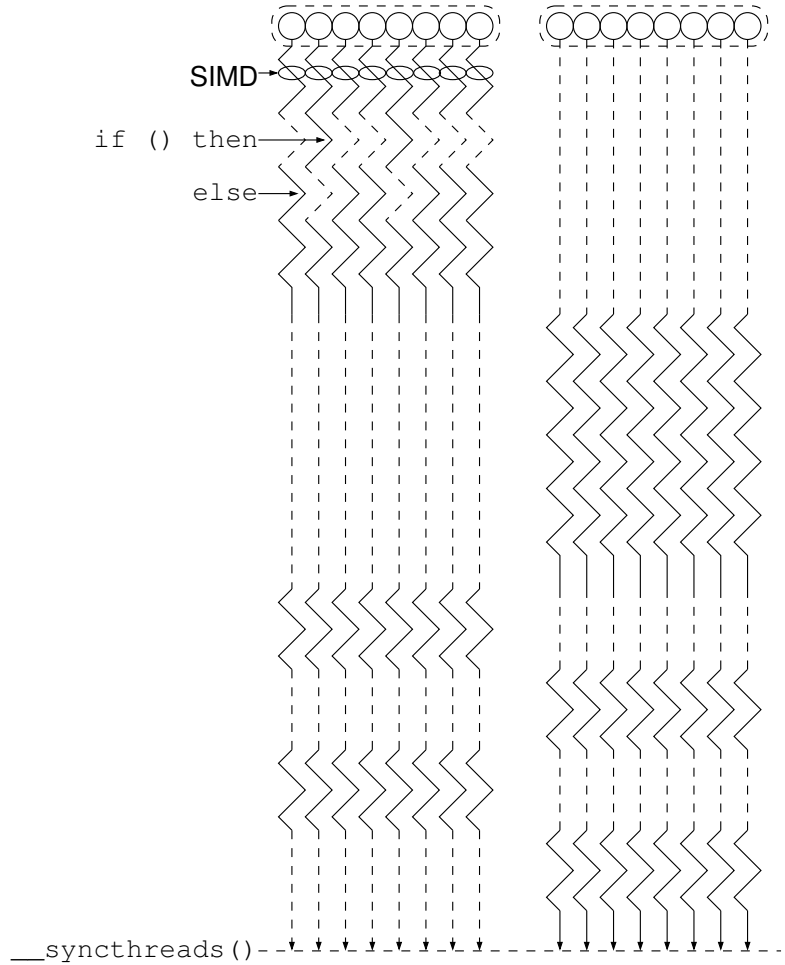
LD/ST - load/store unit
SP - scalar processor
SFU - special function unit
DPU - double precision unit
TC - „tensor” core

Wątki w ramach splotu wykonywane są ściśle synchronicznie, w trybie SIMD. Jednak w innym bloku mogą w tym samym czasie wykonywać *inną* instrukcję *tego samego* programu jądra (tryb SPMD – *Single Program...*). Dla takiego modelu NVIDIA ukuła termin SIMT (*...Multiple Threads*).



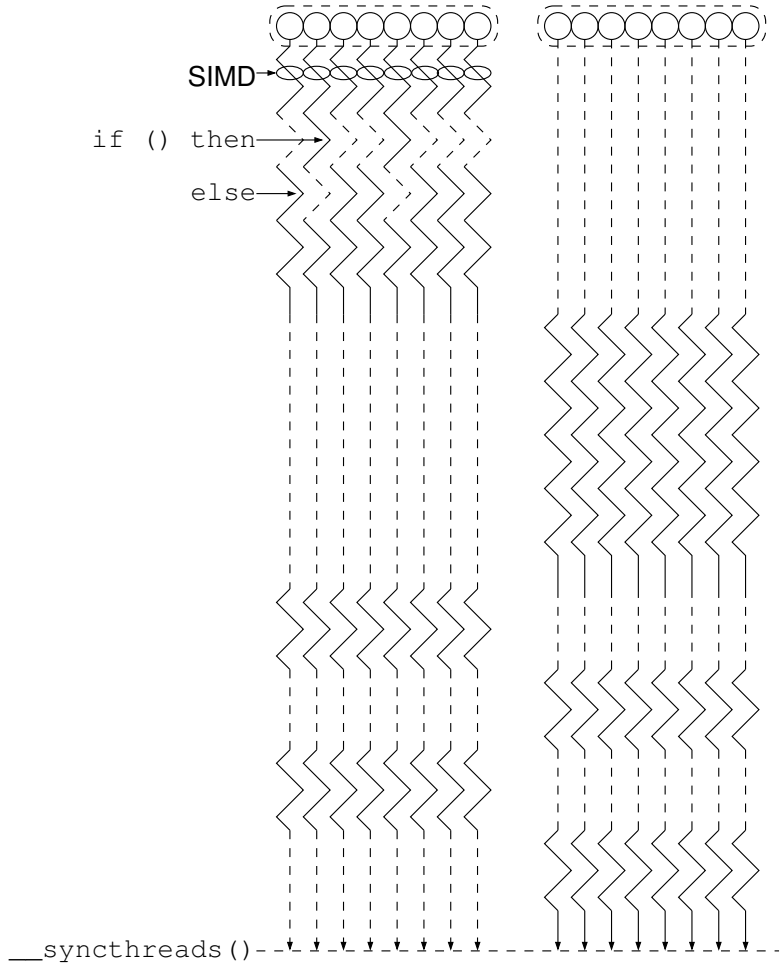
Zanurzenie modelu wykonania w modelu platformy – 5

- W każdym cyklu zegara kierowany jest do wykonania jeden z *gotowych* splotów (tzn. nie czekających na dostęp do pamięci czy wynik operacji).
- W zależności od instrukcji, którą będzie wykonywał wybrany splot, jest on kierowany w trybie SIMD do odpowiedniego podukładu.
- Wszystkie wątki splotu zaczynają działanie od tego samego adresu i mają wspólny stos oraz licznik rozkazów, ale dzięki indywidualnym bitom maski aktywności ich drogi mogą się rozejść (*divergence*).



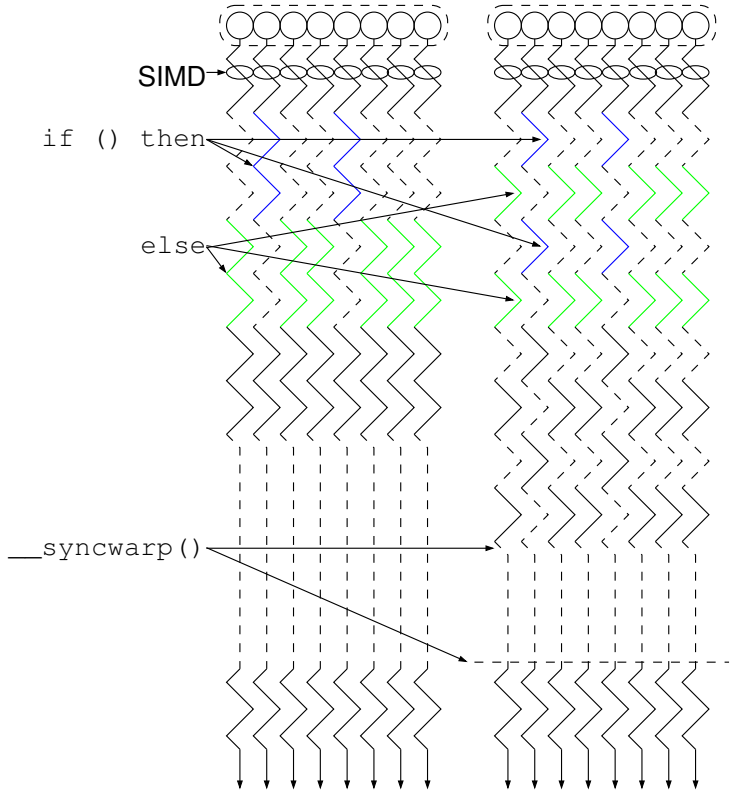
Zanurzenie modelu wykonania w modelu platformy – 6

- Ścieżki instrukcji warunkowej wykonywane są sekwencyjnie, za każdym razem z „wyłączonymi” nieaktywnymi wątkami.
- Wątki w ramach jednego bloku mogą się synchronizować na barierze f-cją `__syncthreads()`.
- Wspólne sterowanie dla wielu jednostek obliczeniowych oszczędza powierzchnię krzemu i upraszcza koordynację zależności między wątkami (np. wyklucza zakleszczenie).
- Przełączenie splotów jest bardzo szybkie (kontekst w rejestrach).



Zanurzenie modelu wykonania w modelu platformy – 7

- Problem: wątki rozbieżne tracą współbieżność – próba wymiany danych pomiędzy nimi grozi zakleszczeniem.
- Na procesorach $^{7.0}$ każdy wątek ma własny stos i licznik rozkazów – rozbieżne ścieżki wykonania wykonywane są współbieżnie. Umożliwia to wymianę danych pomiędzy rozbieżnymi ścieżkami i bezpieczeństwo algorytmów odpornych na zagłodzenie.
- Aktywne wątki splotu są optymalnie grupowane.
- Ponowna zbieżność nie jest automatycznie wymuszana.



$C_C < 7.0$ $C_C \geqslant 7.0$

Język CUDA *C*/C++

- Aplikacja składa się z kodu gospodarza i kodu urządzeń.
- Pliki o rozszerzeniu `.cu`, mogą zawierać kod gospodarza i/lub urządzenia. Są one rozróżniane odpowiednimi modyfikatorami deklaracji funkcji:

Modyfikator	Wykonuje	Wywołuje	Uwagi
`__host__`	host	host	domyślnie
`__global__`	device	host, device[3.5]	jądro
`__device__`	device	device	f-cja pomocnicza

- Modyfikatory `__host__` i `__device__` można łączyć.
- Modyfikatory `__noinline__`[2.0] i `__forceinline__` wymuszają sposób wbudowywania kodu.
- Prosty projekt można umieścić w jednym pliku `.cu`.
- Kod gospodarza może być także w plikach `.c` i `.cpp`.
- Możliwa oddzielna kompilacja modułów urządzenia.
- Od CUDA 11.1 standardowa biblioteka ISO C++ `libcu++`.

Język CUDA *C*/C++ – kod gospodarza

- Język C++11/14/17/20 z dodanym API i rozszerzeniami
- Można programować na trzech różnych poziomach API:
  - Driver API – funkcje `cuXxxx()`
    - pracuje tylko z plikami cubin/fatbin/ptx
    - długi program z wieloma niskopoziomowymi szczegółami
    - odpowiedniość niemal 1:1 z API OpenCL
    - opis w postaci odrębnego podręcznika (721 stron!)
  - Runtime API – funkcje `cudaXxxx()`
    - podejście efektywne, wygodne, dające klarowny kod
    - wynik translacji jest zawsze programem w języku C++
    - poziom zalecany (m.in. na laboratorium i projekcie RIM)
  - Thrust – wysokopoziomowa biblioteka szablonów C++
    - przeznaczona do szybkiego pisania prototypów aplikacji
    - kolekcja algorytmów równoległych (transformacje, redukcje, kumulacje (*scan*), tasowanie, sortowanie, różne iteratory)
    - najmniejsza kontrola programisty na tym, co program robi
- Wszystkie te trzy poziomy można mieszać ze sobą.
- Windows – tylko kompilator Microsoft Visual Studio (`cl`), Linux – tylko GNU (`gcc`); możliwe CUDA-on-WSL 2.

Język CUDA *C*/C++ – struktura kodu gospodarza

1. Wybór urządzenia CUDA dla kolejnych funkcji API

   ```
   cudaSetDevice(device_number);
   ```

2. Alokacja pamięci (*Unified Memory*$^{6.0}$ Linux) (*Unified Memory*$^{3.0}$) urządzenia

   ```
   cudaMalloc(&d_ptr,size);cudaMallocManaged(&d_ptr,size);
   malloc(&d_ptr,size);
   ```

3. Kopiowanie danych gospodarza do urządzenia (ukryte$^{3.0}$)

   ```
   cudaMemcpy(d_ptr,h_ptr,size,cudaMemcpyHostToDevice);
   ```

4. Asynchroniczne uruchomienie jądra (np. macierz $m \times n$)

   ```
   dim3 dimGrid((n+15)/16,(m+15)/16), dimBlock(16,16);
   kernel_name<<<dimGrid,dimBlock>>>(parametry);
   checkCudaErrors(cudaGetLastError());
   ```

5. Synchronizacja z jądrem (bariera)

   ```
   cudaDeviceSynchronize();
   ```

6. Kopiowanie wyników pracy z urządzenia do gospodarza$^{3.0}$

   ```
   cudaMemcpy(h_ptr,d_ptr,size,cudaMemcpyDeviceToHost);
   ```

7. Zwolnienie pamięci urządzenia

Język CUDA *C*/C++ – kod urządzenia

- Kod urządzenia – ograniczony dialekt C++11/14/17/20
  - \+ istnieją zwykłe klasy agregujące dane, dziedziczenie, szablony klas, szablony funkcji, funktory, rekurencja$^{2.0}$, λ
  - – brak możliwości korzystania z informacji o typach podczas wykonania programu (RTTI), obsługi wyjątków, biblioteki STL, zmiennej liczby parametrów funkcji, współbieżności
- Rozszerzenia – nowe słowa kluczowe języka:
  - modyfikatory deklaracji funkcji: `__device__`, `__global__`, `__noinline__`$^{2.0}$, `__forceinline__`
  - modyf. dekl. zmiennych: `__device__`, `__managed__`$^{3.0}$, `__constant__`, `__shared__`, `__restrict__`
  - wbudowane typy wektorowe, np. `dim3` (domyślnie pola =1)
  - wbudowane zmienne – jednakowe (`gridDim`, `blockDim`, `warpSize`) lub różne (`blockIdx`, `threadIdx`) dla wątków
- Można korzystać z funkcji ułatwiających uruchamianie:
  - `clock()` – zegar wieloprocesora SM/SMX/SMM/SM
  - `__prof_trigger(counter)`$^{2.0}$ – licznik wieloprocesora
  - `assert(predicate)`$^{2.0}$ – asercje, niezmienniki
  - `printf(format, ...)`$^{2.0}$ – wydruki kontrolne

Język CUDA *C*/C++ – struktura kodu urządzenia

1. Lokalizacja bieżącego wątku w sieci (określenie indeksu *i*)

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

2. Sprawdzenie, czy wątek znajduje się w dziedzinie obliczeń

```
if (i < size)
```

3. Wykonanie obliczenia dla pojedynczego (*i*-tego) „ziarna”

```
out[i] = fun(in[i]);
```

Funkcja jądra (w najprostszym przypadku)

wykonuje obliczenie skalarne, dla jednego elementu tablicy, o indeksie zależnym od położenia danego wątku w sieci.

Podstawowe zagadnienie projektowania algorytmu dla CUDA:

określenie, czym jest pojedyncze „ziarno” obliczeń.

Program sterujący kompilacją – `nvcc`

Program sterujący `nvcc` wywołuje kolejno właściwe preprocesory, kompilatory, asemblery, konsolidatory itp.:

`g++ -E`	wstępne przetwarzanie plików źródłowych i dołączanie plików nagłówkowych (preprocesor)
`cudafe`	oddzielenie kodu gospodarza i urządzeń oraz generacja kodu obu platform w języku C++
`cicc`	kompilacja kodu urządzeń dla wirtualnych architektur obliczeniowych o podanym C_c – PTX
`ptxas`	asemblacja kodu PTX dla rzeczywistych architektur sprzętowych o podanym C_c – SASS
`fatbinary`	połączenie (w tablice) zasemblowanego i/lub źródłowego kodu PTX różnych architektur
`g++ -c`	kompilacja...
`g++`	...i konsolidacja kodu gospodarza z dołączonym (w tablicach) kodem urządzeń

Konflikt pokoleń w programach CUDA

- Wieloprocesory strumieniowe rozwijają się w generacjach:

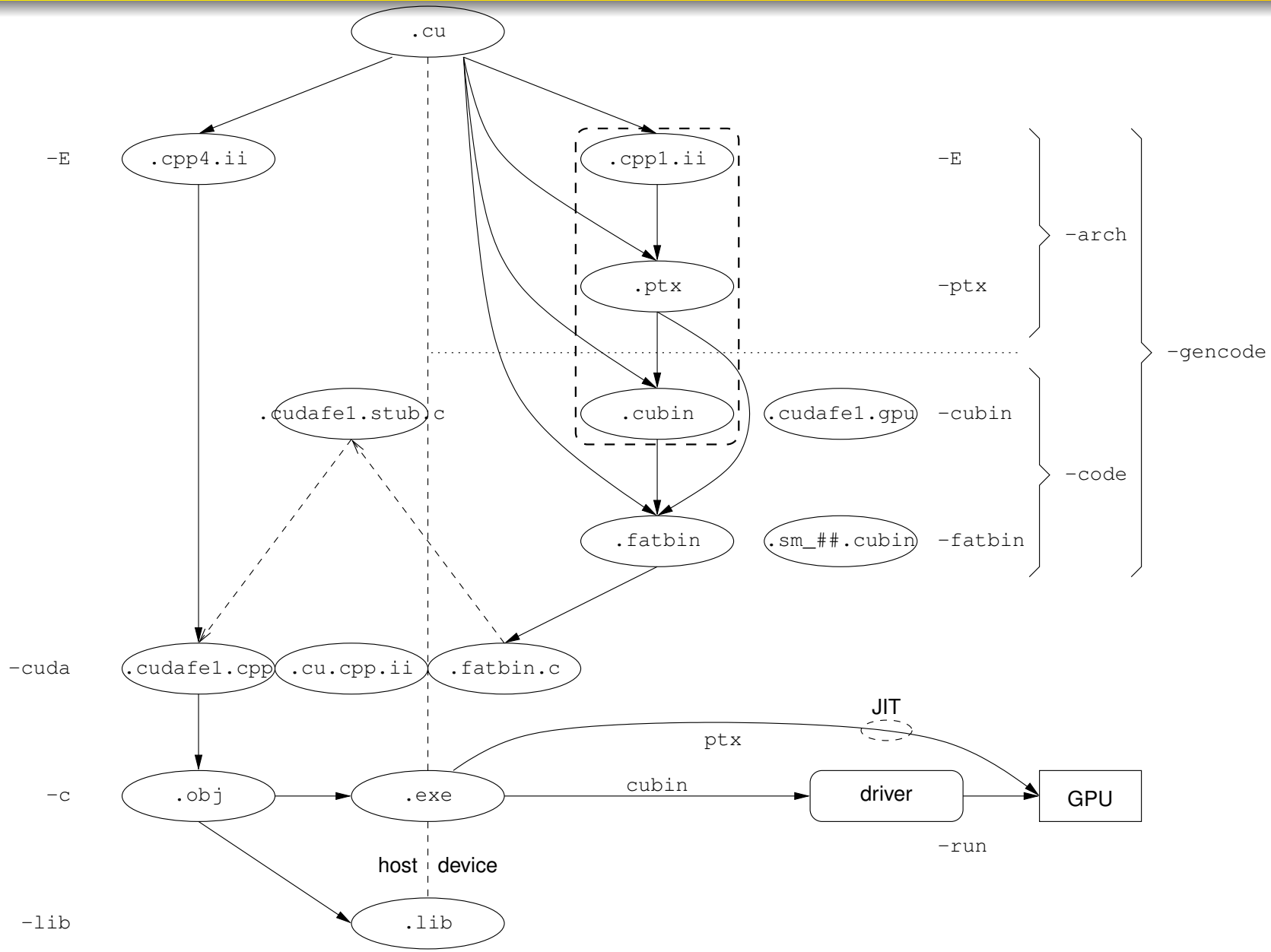
1	Tesla	'07	6	Pascal	'16	9	Hopper	'22
2	Fermi	'10	7	Volta	'17	10	Blackwell	'24
3	Kepler	'12		Turing	'18			
5	Maxwell	'14	8	Ampere	'20			
				Ada	'22			

- W ramach danej generacji g występują podgeneracje p. Cechy funkcjonalne danej podgeneracji określane są mianem jej możliwości obliczeniowych $C_c = g.p$.
- W ramach danej podgeneracji procesory GPU mogą się różnić liczbą wieloprocesorów strumieniowych (SM), częstotliwością zegara, szerokością interfejsu pamięci itp.
- Zgodność kodu binarnego pomiędzy generacjami nie jest gwarantowana, ale w ramach danej generacji wyższa podgeneracja umie wykonywać kod niższej.

Koncepcja PTX (*Parallel Thread Execution*)

- W celu zapewnienia zgodności z przyszłymi rzeczywistymi architekturami sprzętowymi (SM), kompilator generuje kod pośredni w asemblerze PTX dla wirtualnej architektury obliczeniowej (*compute*), określonej wyłącznie przez swe cechy funkcjonalne.
- Architekturę sprzętową odpowiadającą możliwościom obliczeniowym $C_c = g.p$ oznaczamy w opcjach kompilatora `sm_`*gp*, zaś architekturę obliczeniową – `compute_`*gp*.
- Kod źródłowy PTX maszyny wirtualnej może zostać zapisany w binariach (`.exe`) i zostać przetłumaczony przez sterownik graficzny na postać binarną nie istniejącego jeszcze w chwili kompilacji GPU „w ostatniej chwili” (JIT – *Just In Time*), podczas ładowania programu do GPU.

Ścieżka kompilacji i opcje programu `nvcc`



Koncepcja PTX (*Parallel Thread Execution*) – c.d.

- W opcji `-arch` podaje się dokładnie jedną architekturę obliczeniową. Opisuje ona zbiór cech funkcjonalnych wymaganych przez program. Powinna ona być jak najniższa, aby niepotrzebnie nie zawężać możliwości wykonywania aplikacji tylko do najsilniejszych procesorów.
- W opcji `-code` podaje się architekturę sprzętową. Nie może być ona niższa od wirtualnej (musi implementować wszystkie żądane cechy funkcjonalne). Jeśli piszemy program na konkretny typ GPU, to dla większej efektywności architektura sprzętowa powinna być jak najwyższa, ale nie większa niż możliwości tego procesora.
- W opcji `-code` można podać całą listę architektur:
  - rzeczywistych (sprzętowych) – następuje generacja kilku wariantów kodu binarnego na odpowiednie platformy,
  - wirtualnych (obliczeniowych) – następuje generacja kilku wariantów kodu źródłowego PTX dla przyszłych platform.

Wybrane cechy funkcjonalne SM dla różnych C_c

Cecha	$C_c =$ *2.0* *2.1*	*3.0*	*3.2*	3.5 3.7 5.0 5.2	5.3	6.0 6.1 6.2	7.0 7.5	8.0 8.6 8.9
atomowe: 64b `add`, `cas`, `exch` na pam. globalnej, 32b FP `add`; rozszerz. głos. (*ballot*); rozszerzona synchr. wątków (*fence*); rekurencja; f-cje powierzchn.	■	■	■	■	■	■	■	■
model programowania *Unified Memory*		■	■	■	■	■	■	■
64b atomowe `and`, `or`, `xor`, `min`, `max`; przesuwnik bitowy $2 \times 32 \rightarrow 32$ (*funnel*)			■	■	■	■	■	■
dynamiczna równoległość				■	■	■	■	■
16b arytmetyka zmiennoprzecinkowa					■	■	■	■
atomowe 64b zmiennoprzec. `add`						■	■	■
operacje „tensorowe”							■	■
asynchr. GM→SM; podzielona synchr. na barierze (*Arrive/Wait*); zarządz. L2								■

Wybrane parametry architektury SM dla różnych C_c

Par.	*2.0*	*2.1*	*3.0* *3.2*	3.5 3.7	5.x	6.0	6.1 6.2	7.0	7.5	8.0	8.6 8.9
P_M	32	48	192		128	64	128	$64^{FP} + 64^{INT}$			$128^{FP} + 64^{INT}$
D_M	4/16	4	8	8/64	4	32	4	32	2	32	2
F_M	4	8	32			16	32	16			
L_M	16		32			16	8	32			16
I_M	2		4			2	4				
PB_M	1				4	2		4			

Legenda – jeden wieloprocesor M (*multiprocessor*) ma:

- P rdzeni CUDA (*processor*)
- D jednostek podwójnej precyzji (*double*)
- F jednostek funkcji specjalnych (*function*)
- L jednostek transferu danych (*load/store*)
- I układów zarządzania splotami (*issue unit* / *warp scheduler*)
- PB bloków przetwarzania (*processing block*)