

JEZYKI I NARZĘDZIA PROGRAMOWANIA II

CUDA

Wykład II - Architektura CUDA

ZAŁOŻENIA

- Duża przepustowość pamięci
- Duża moc obliczeniowa
- Duża równoległość (model data parallel)
- Ukrywanie latencji pamięci i latencji operacji
- Skalowalność

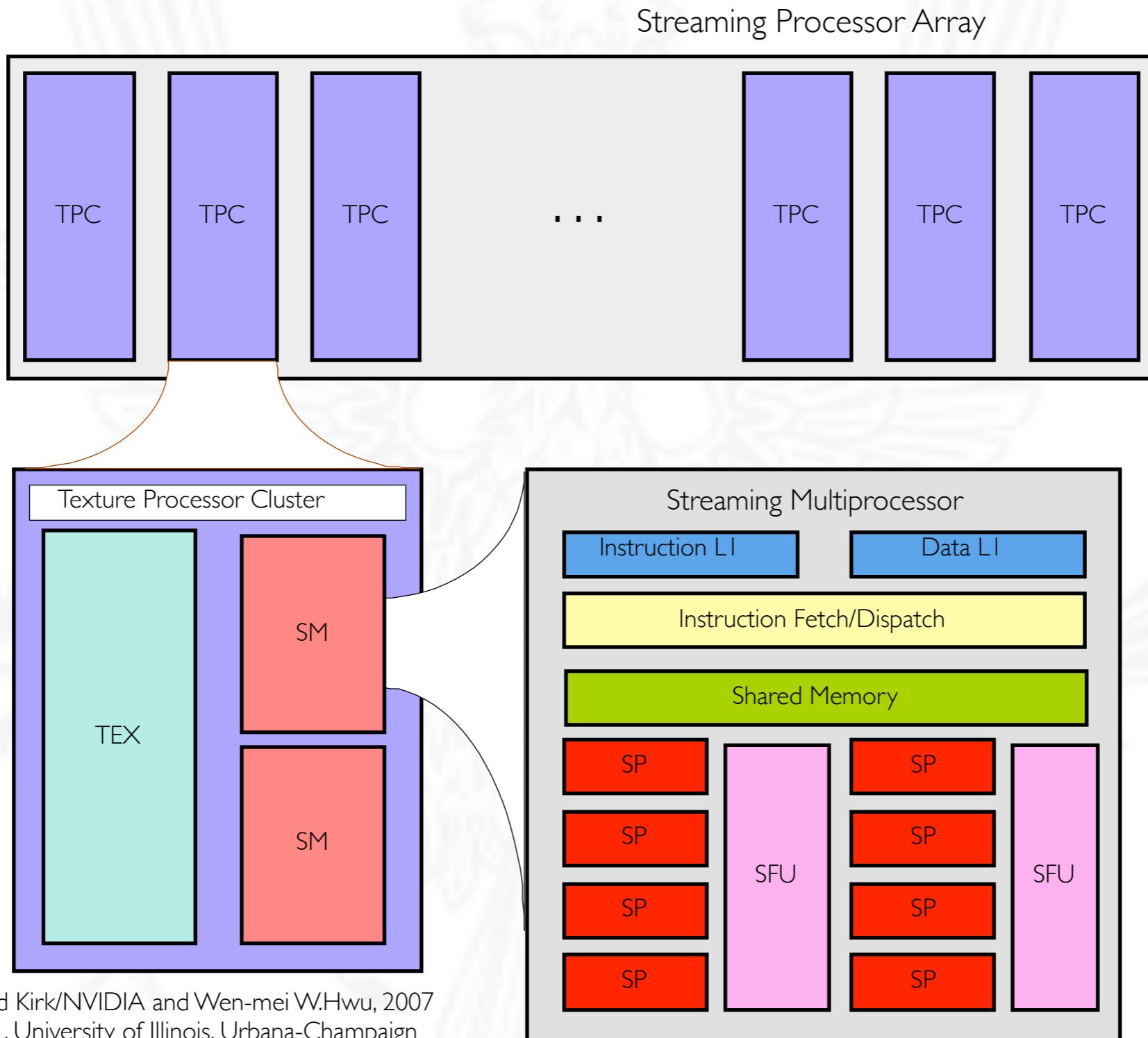
REALIZACJA SPRZĘTOWA

- Karta koprocesora (graficznego/obliczeniowego) połączona z CPU za pomocą szybkiego złącza o dużej przepustowości
- Macierz procesorów strumieniowych (SPA)
- Architektura SIMT (Single Instruction Multiple Threads)
- Hierarchiczna budowa:
 - Karta
 - SM (Streaming Multiprocessor)
 - 8 × SP (Streaming Processor)
 - Inne jednostki wykonawcze
 - 8 × TF (Texture Filtering Unit)

MODEL PROGRAMOWANIA

- Kernele obliczeniowe uruchamiane na GPU są zarządzane przez główny program uruchamiany na CPU
- Grid (kernel) = N Bloków
 - Blok = $K \times 32$ wątki
 - Wątek = program
- Bloki muszą być niezależne – kolejność wykonywania bloków jest dowolna
- Ilość bloków jest (niemal) dowolna
- Wątki w ramach jednego bloku mogą się komunikować przy użyciu pamięci współdzielonej
- Wszystkie wątki mają dostęp do wspólnej pamięci globalnej

ARCHITEKTURA CUDA – G80



Zródło: David Kirk/NVIDIA and Wen-mei W.Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

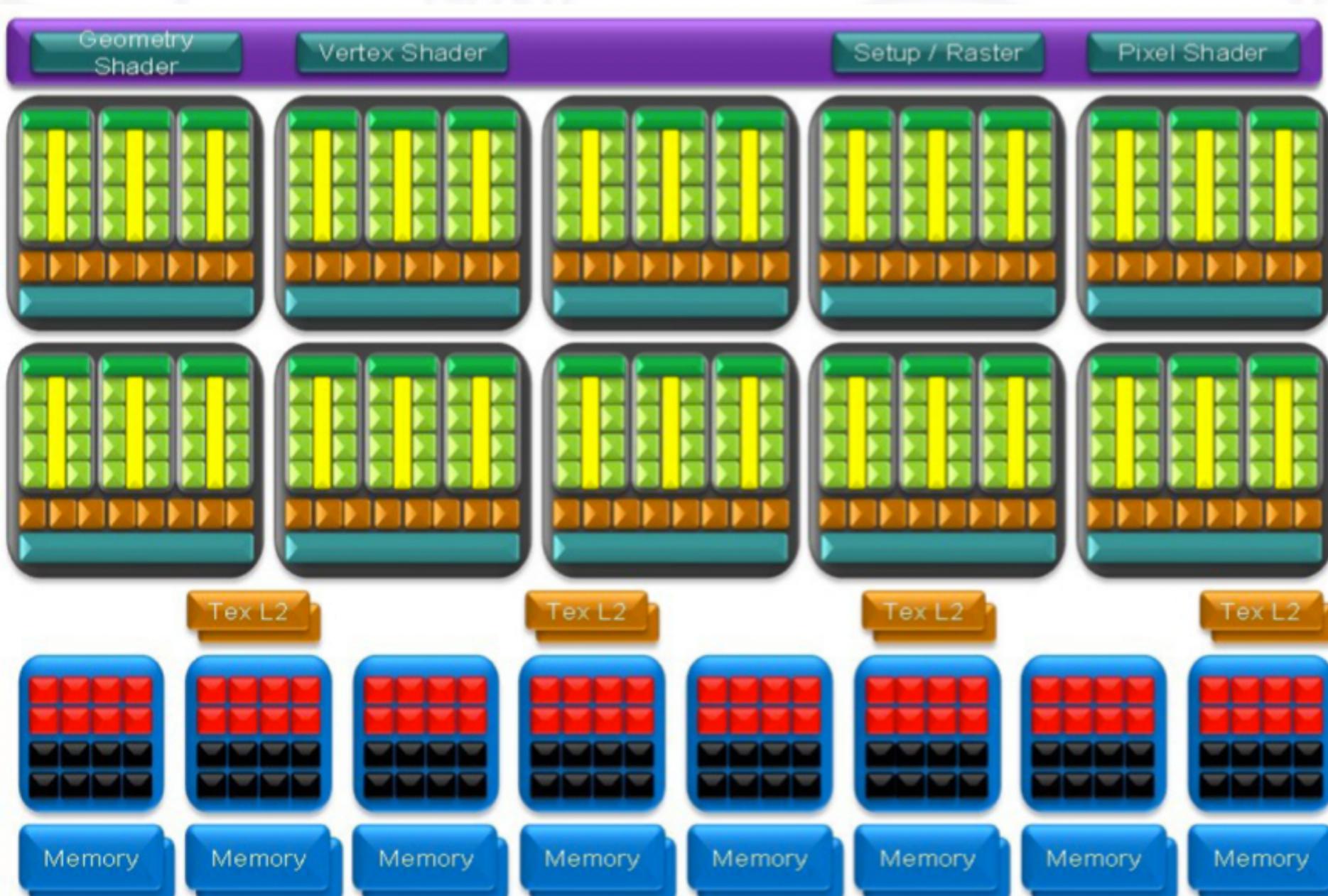


ARCHITEKTURA CUDA

SM (Streaming Multiprocessor)

- Instruction Issuing Unit
- $8 \times$ SP (Streaming Processor)
- $2 \times$ SFU (Special Function Unit)
- $1 \times$ DPALU
- 16 kB pamięci współdzielonej
- $8 \times$ Texture Filtering Unit
- Instruction & Data cache
- Texture & Constant memory cache

ARCHITEKTURA CUDA – GTX 200 (PROCESOWANIE GRAFIKI)

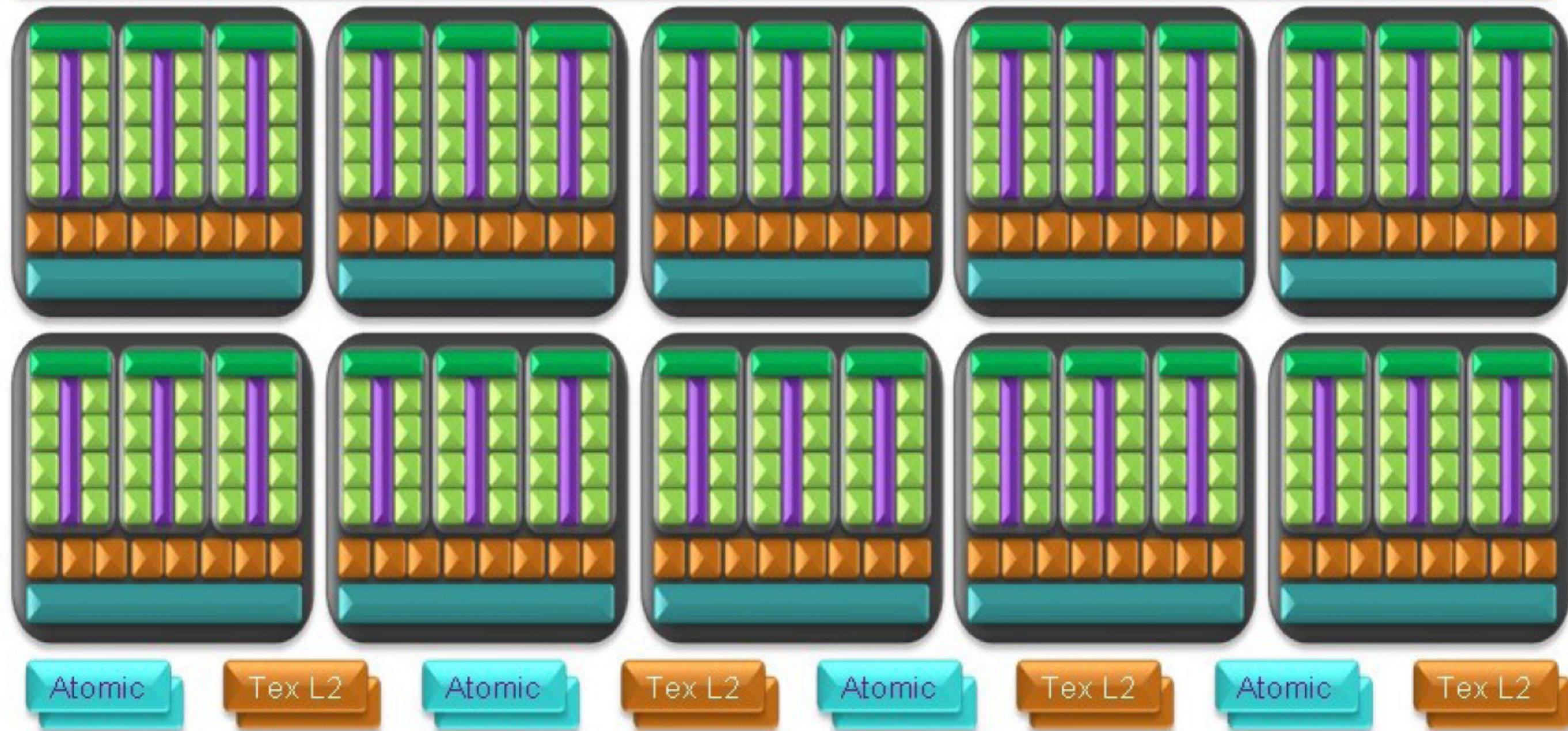


- TPC – texture processing cluster
- SM – streaming multiprocessor
- SP – streaming processor

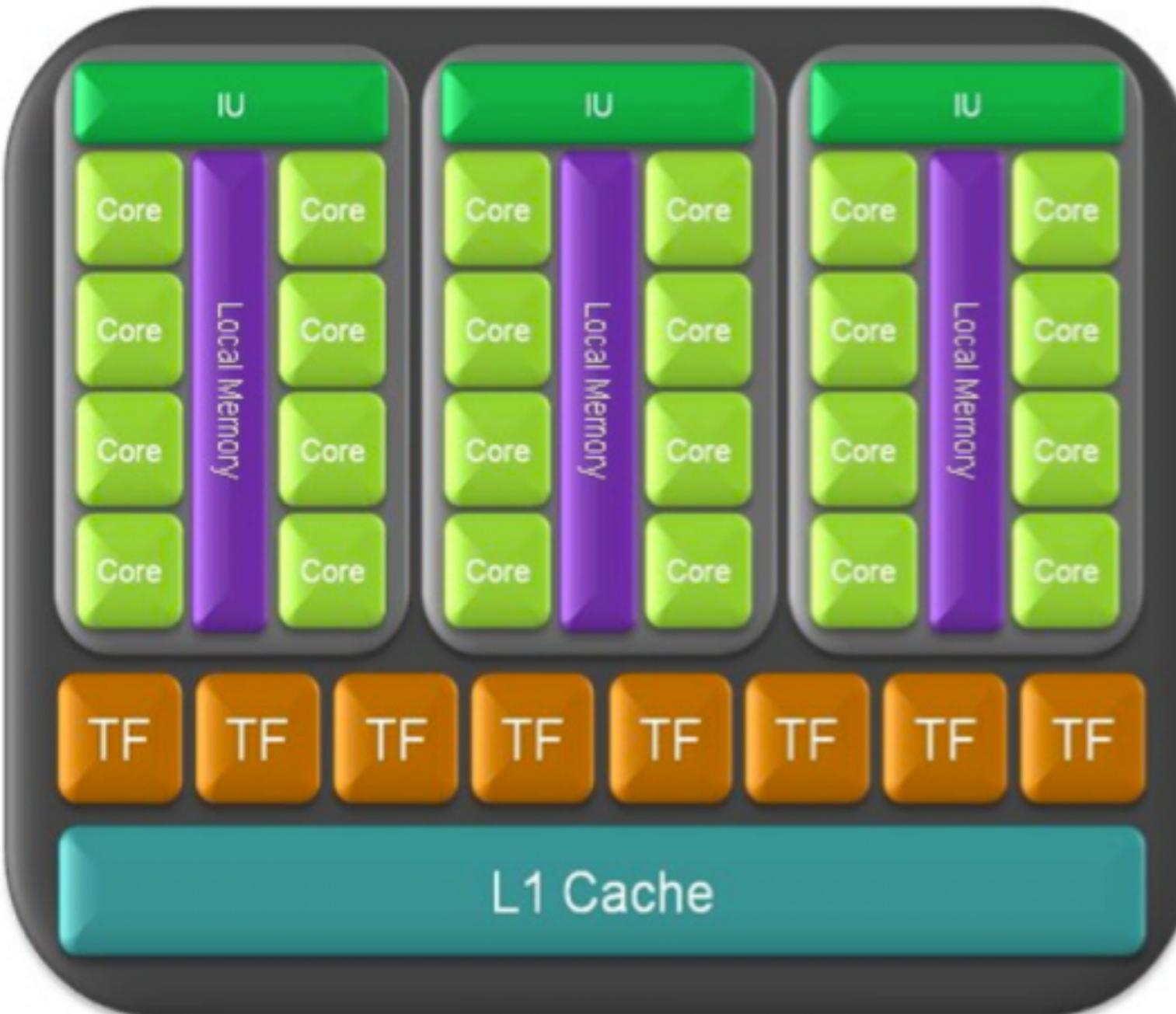
ARCHITEKTURA CUDA – GTX 200

OBLCZENIA

Thread Scheduler



SPECYFIKACJA CUDA

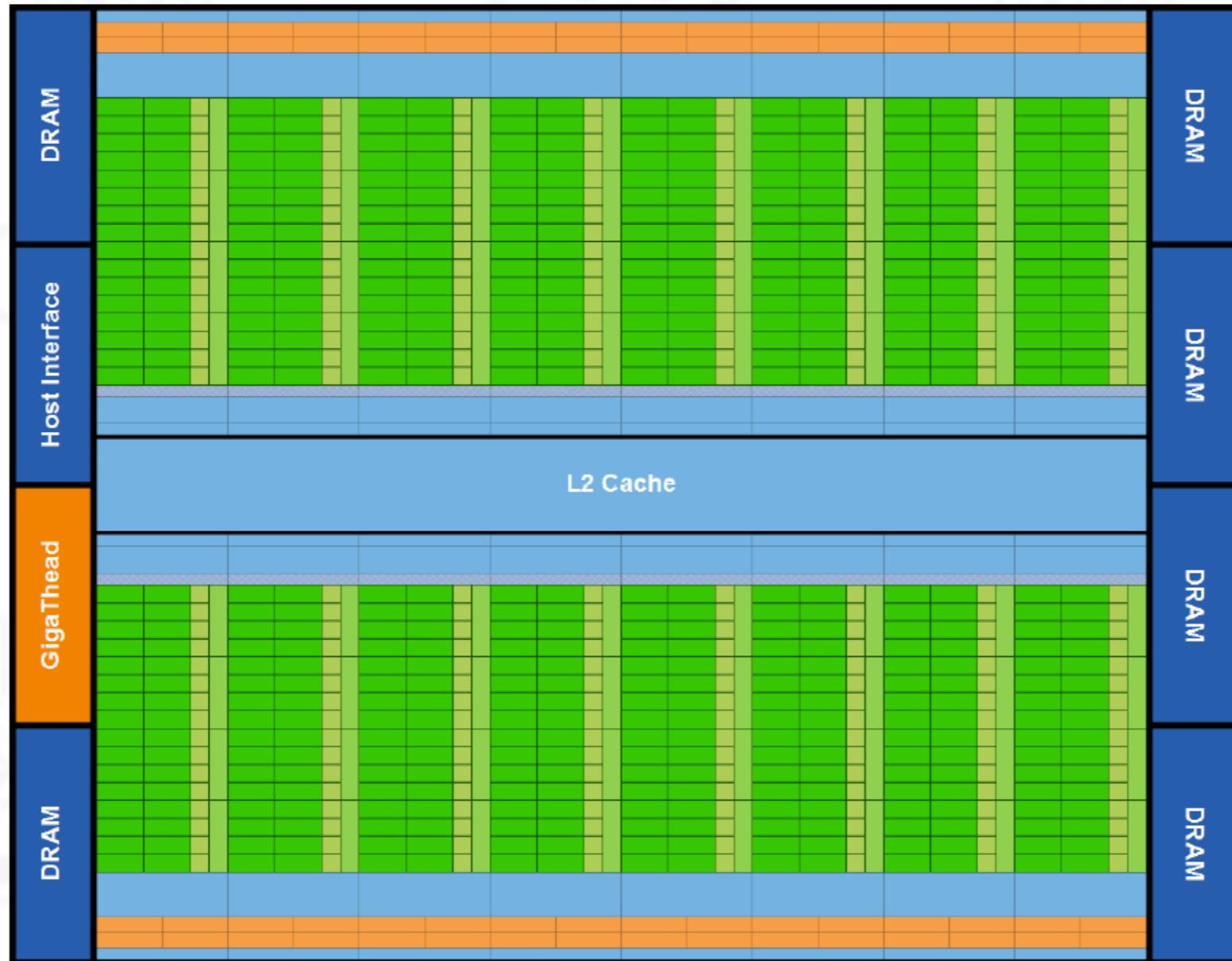


- TPC – texture processing cluster
- SM – streaming multiprocessor
- SP – streaming processor
- TF – texture filtering

SPECYFIKACJA CUDA

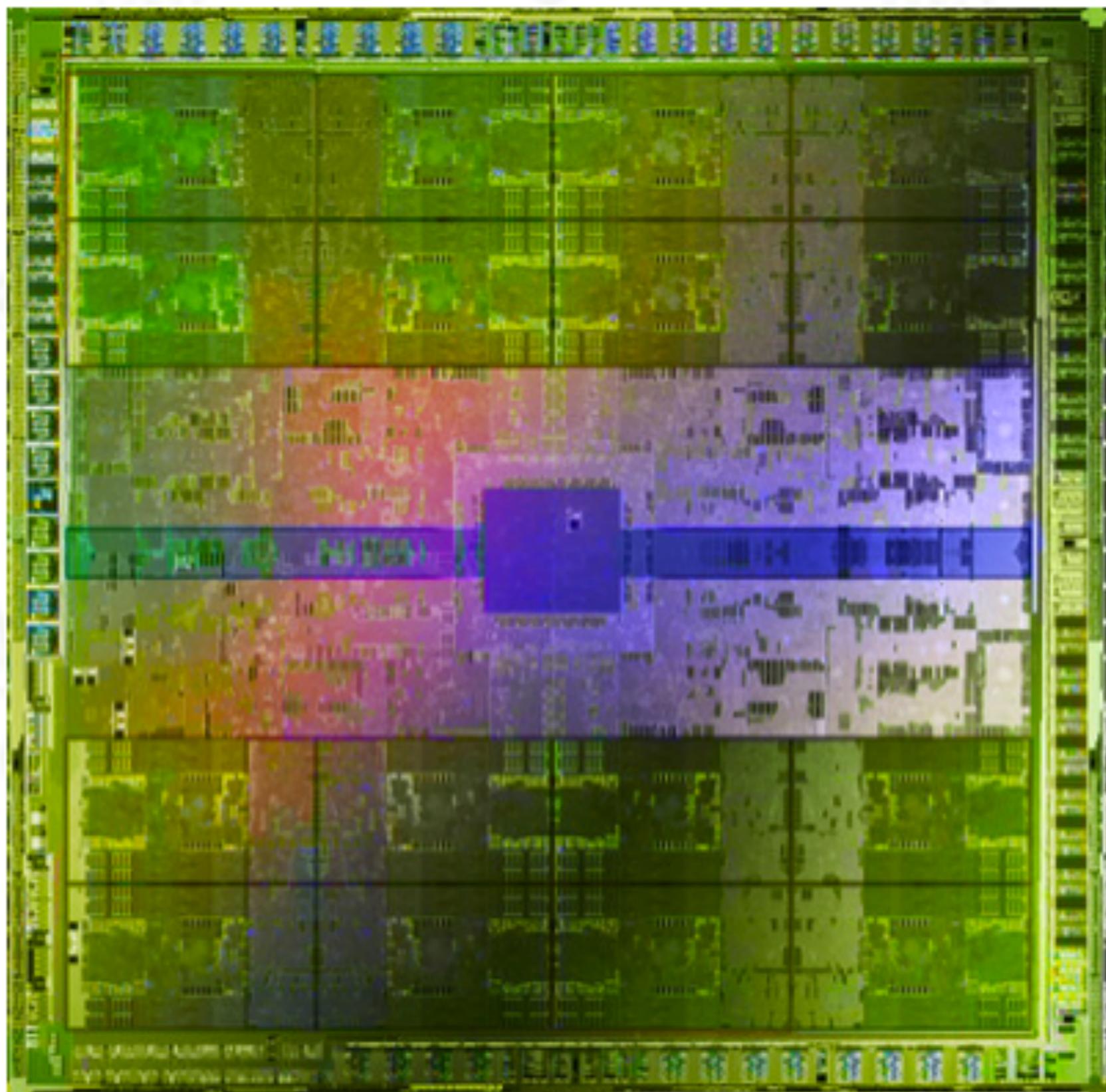
GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

FERMI



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

FERMI



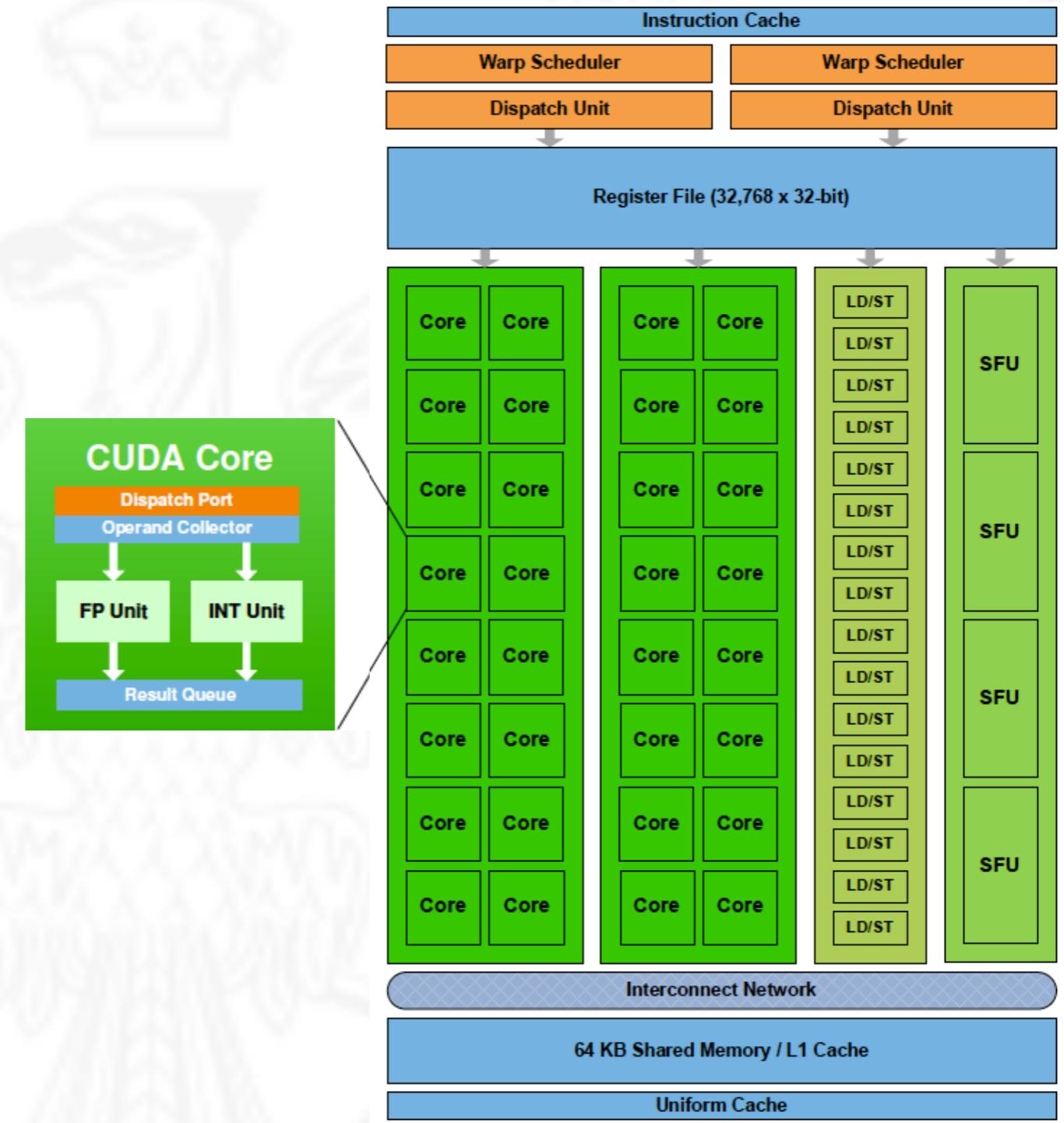
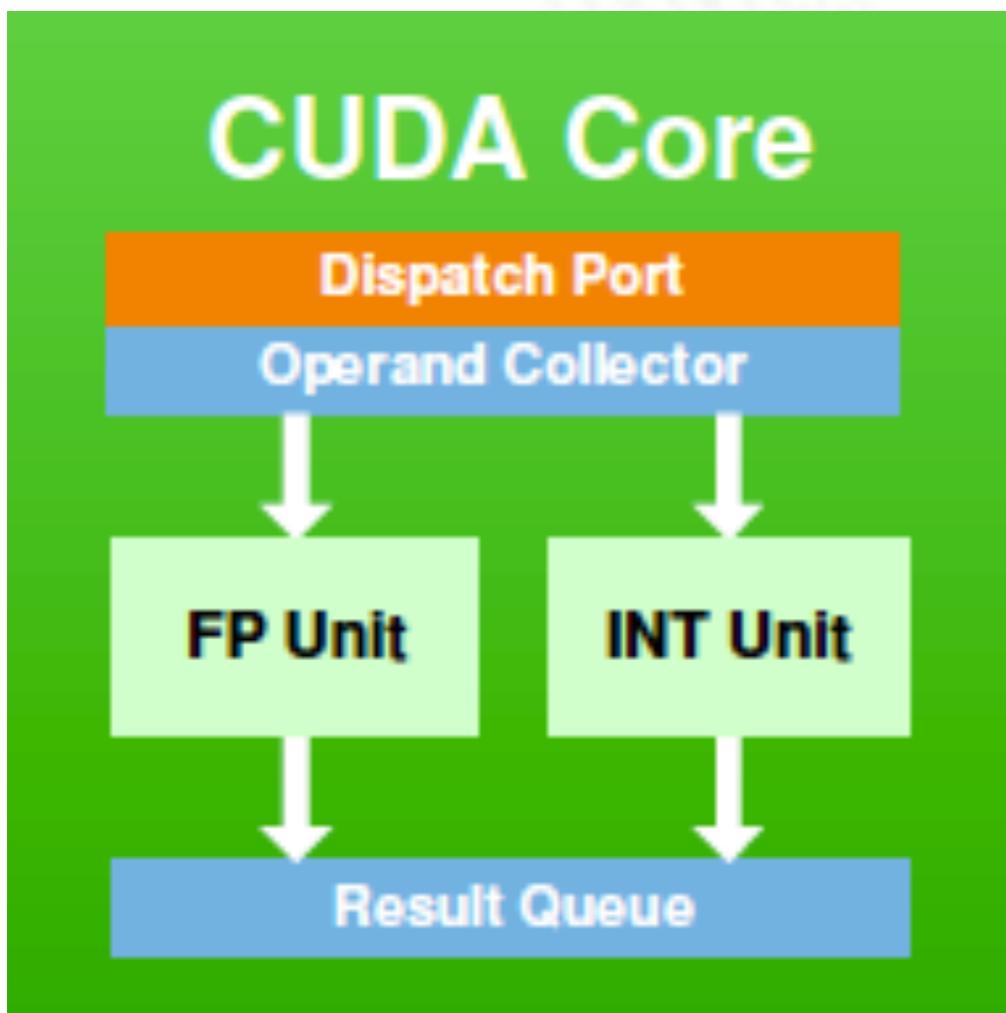
MIM UW

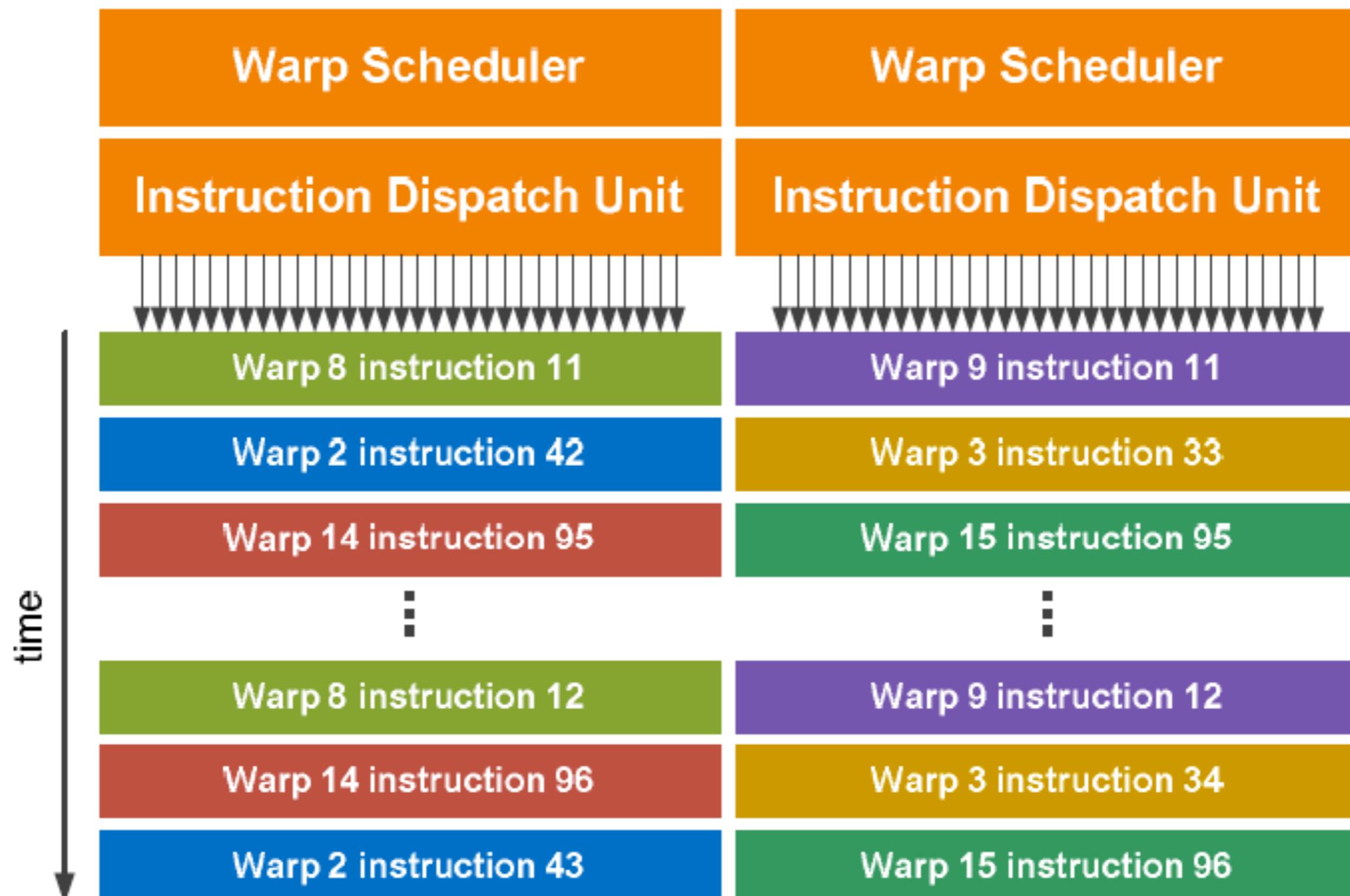
Warszawa

17.03.2016



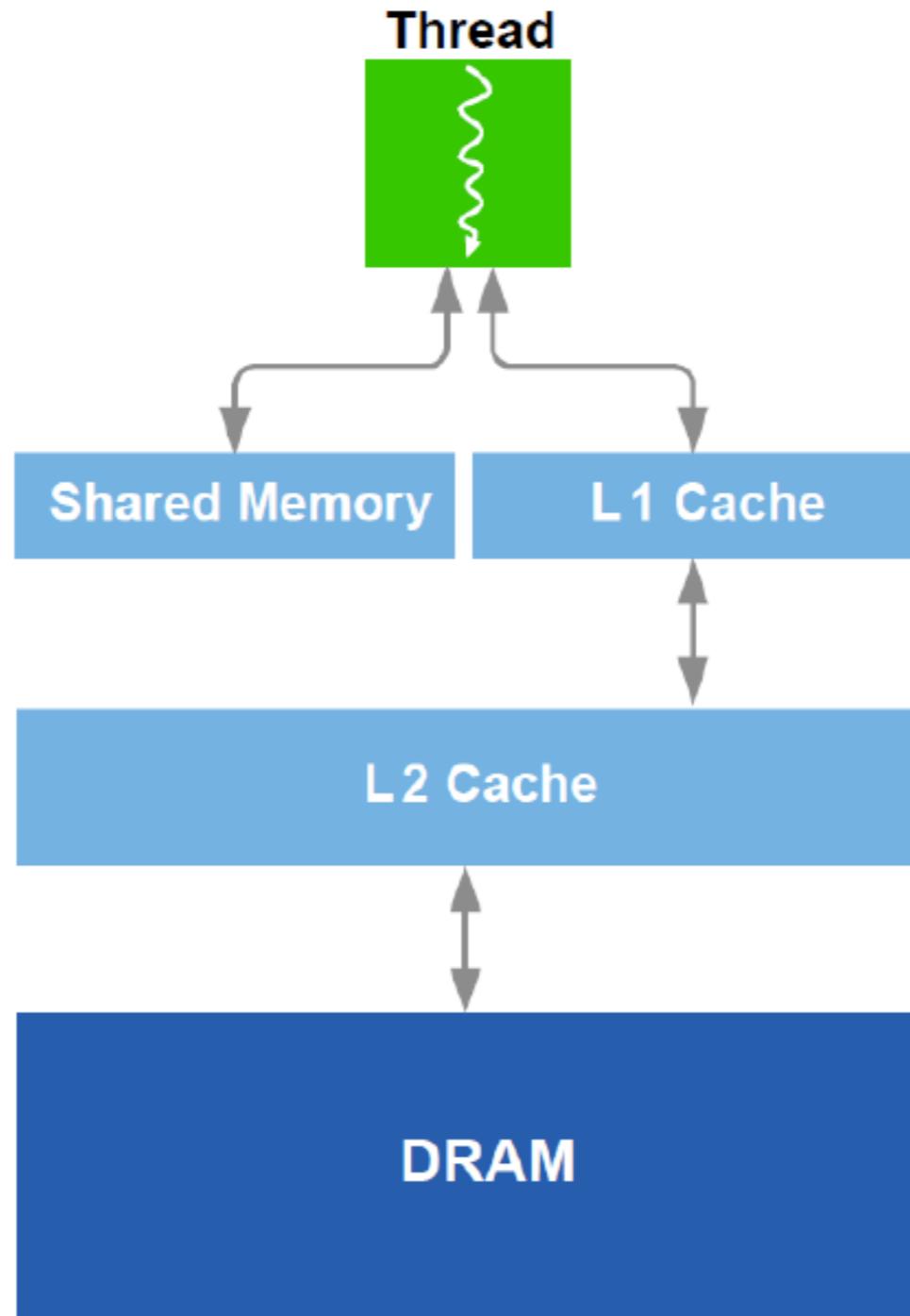
FERMI



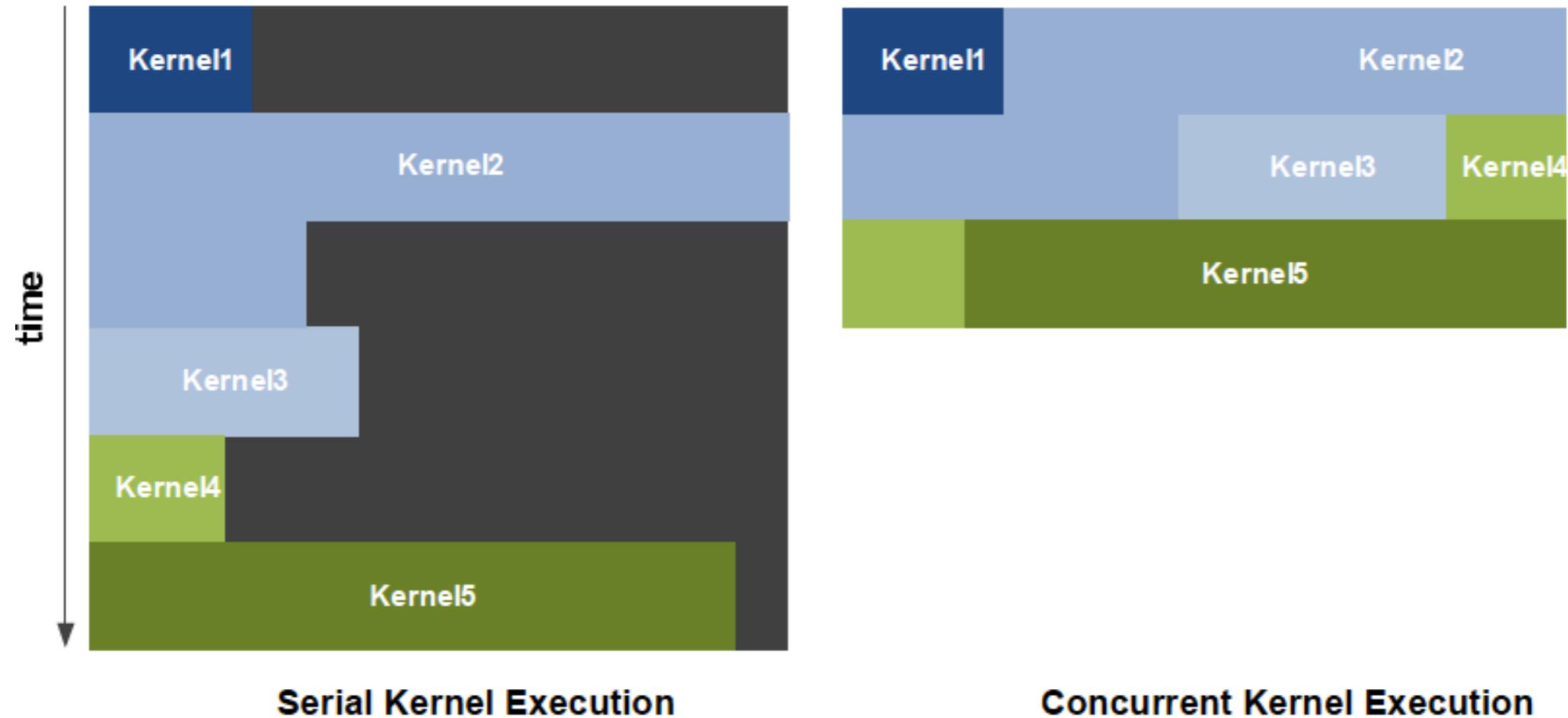


STREAMING MULTIPROCESSOR (SM)

Fermi Memory Hierarchy



STREAMING MULTIPROCESSOR (SM)



KEPLER



KEPLER



FERMI A KEPLER

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110	KEPLER GK210
Compute Capability	2.0	2.1	3.0	3.5	3.7
Threads / Warp			32		
Max Threads / Thread Block			1024		
Max Warps / Multiprocessor	48		64		
Max Threads / Multiprocessor	1536		2048		
Max Thread Blocks / Multiprocessor	8		16		
32-bit Registers / Multiprocessor	32768		65536	131072	
Max Registers / Thread Block	32768		65536	65536	
Max Registers / Thread		63		255	
Max Shared Memory / Multiprocessor			48K		112K
Max Shared Memory / Thread Block			48K		
Max X Grid Dimension	2^16-1		2^32-1		
Hyper-Q		No		Yes	
Dynamic Parallelism		No		Yes	

WYDAJNOŚĆ INSTRUKCJI

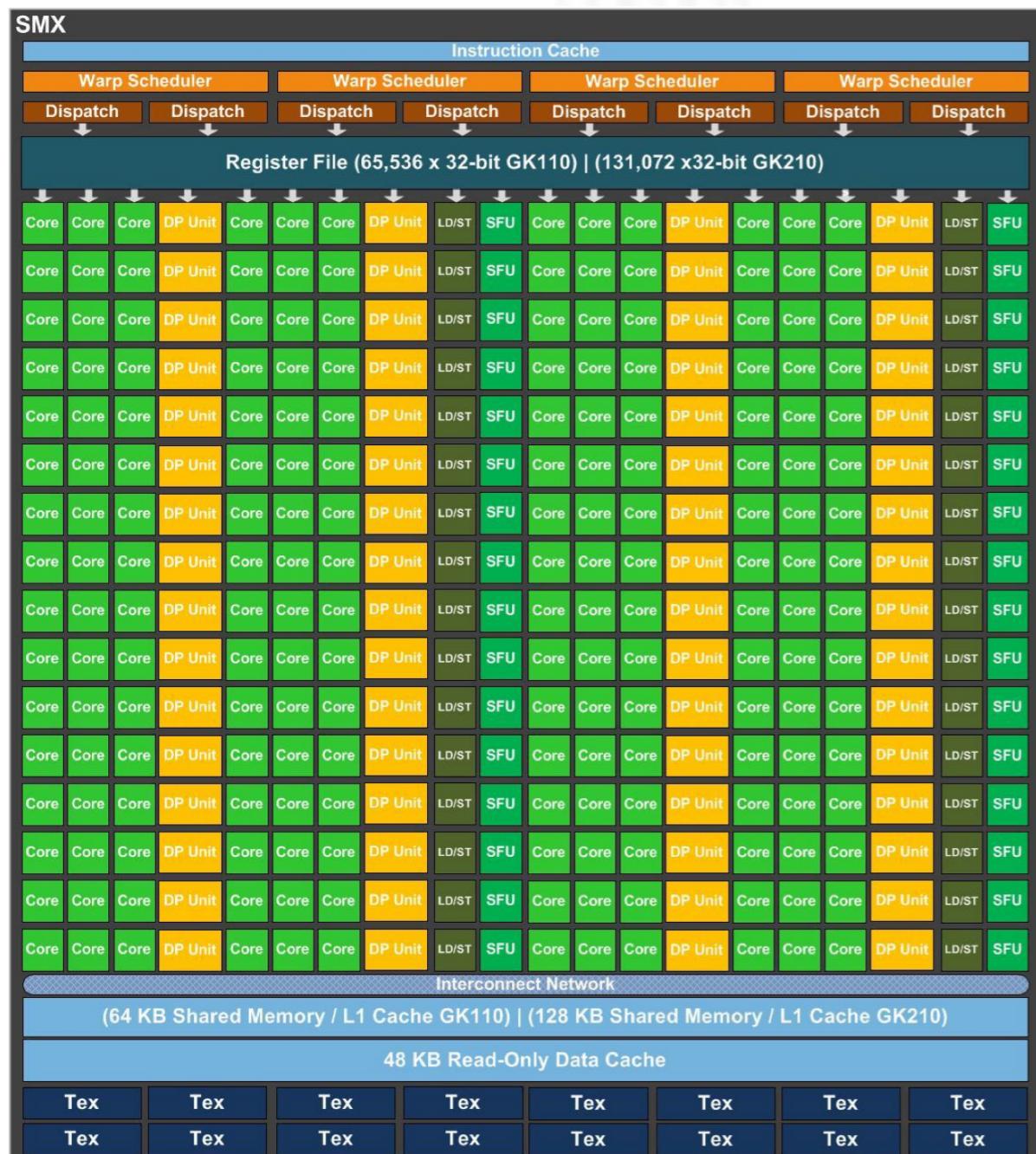


Table 2 Throughput of Native Arithmetic Instructions

(Operations per Clock Cycle per Multiprocessor)

	Compute Capability					
	1.0	1.1	1.3	2.0	2.1	3.0
	1.2					3.5
32-bit floating-point add, multiply, multiply-add	8	8	32	48	192	192
64-bit floating-point add, multiply, multiply-add	1	1	16(*)	4	8	64
32-bit integer add	10	10	32	48	160	160
32-bit integer compare	10	10	32	48	160	160
32-bit integer shift	8	8	16	16	32	64
Logical operations	8	8	32	48	160	160
32-bit integer multiply, multiply-add, sum of absolute difference	Multiple instructions	Multiple instructions	16	16	32	32
24-bit integer multiply (<code>__mul24</code>)	8	8	Multiple instructions	Multiple instructions	Multiple instructions	Multiple instructions
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (<code>__log2f</code>), base 2 exponential (<code>exp2f</code>), sine (<code>__sinf</code>), cosine (<code>__cosf</code>)	2	2	4	8	32	32
Type conversions from 8-bit and 16-bit integer to 32-bit types	8	8	16	16	128	128
Type conversions from and to 64-bit types	Multiple instructions	1	16(*)	4	8	32
All other type conversions	8	8	16	16	32	32

(*) Throughput is lower for GeForce GPUs.



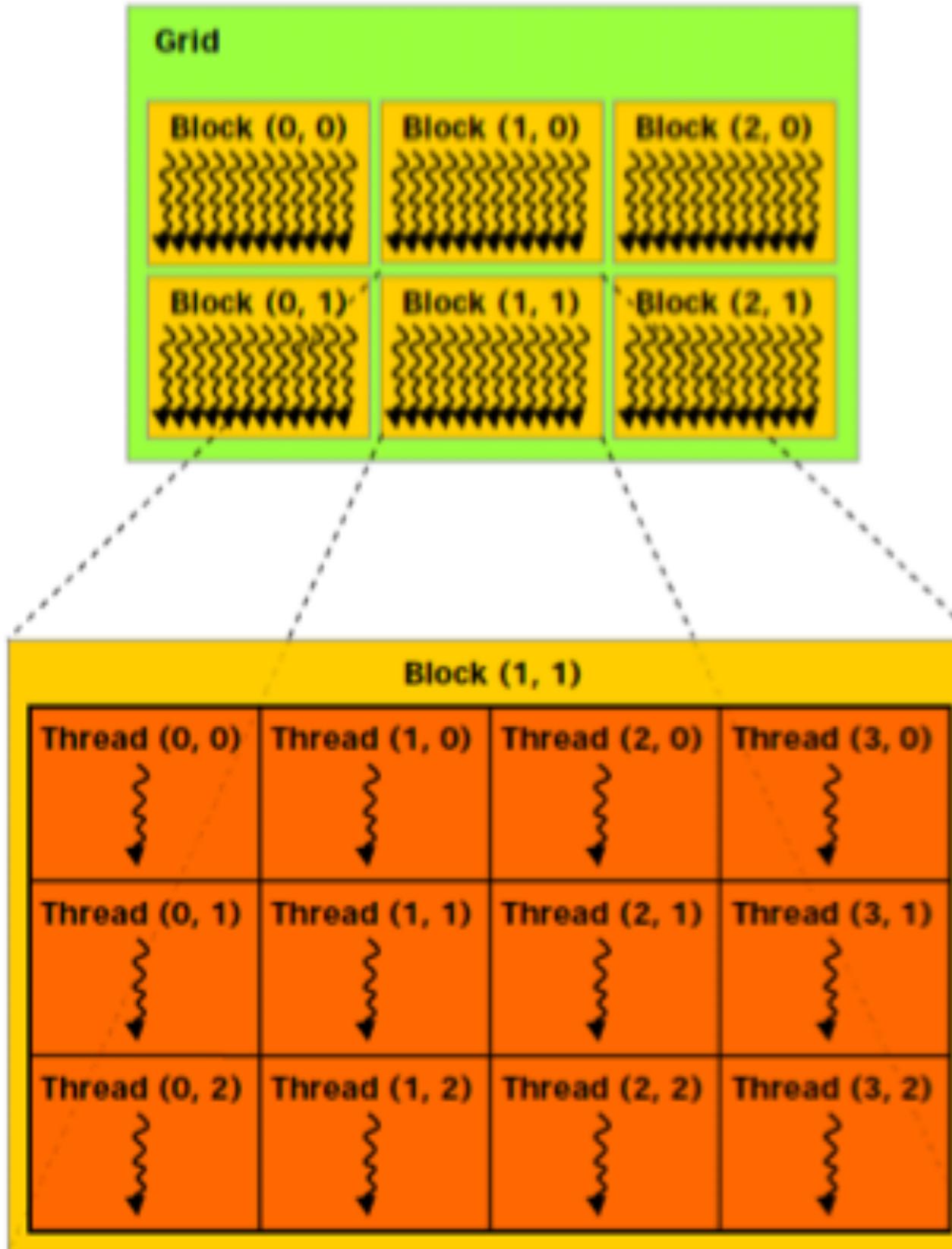
STREAMING MULTIPROCESSOR (SM)

- Streaming Multiprocessor (SM)
 - 32 Streaming Processors (SP)
 - 4 Super Function Units (SFU)
 - 16 Load/Store Units (LSU)
- 32 768 rejestrów 32-bitowych (16 384 64-bitowe)
- 64 KB on chip memory (16K zmiennych typu float lub int) w 32 bankach
 - 48 KB shared memory / 16 KB L1 cache
 - 16 KB shared memory / 48 KB L1 cache
- Constant memory 64 KB
- Cache dla constant memory 8 KB na multiprocesor
- Cache dla texture memory 6-8 KB na multiprocesor
- 512 KB local memory per thread

STREAMING MULTIPROCESSOR (SM)

- Hierarchia
- Siatka (grid)
- Blok (block)
- Paczka (warp) składa się z 32 wątków
- Dla compute capability 1.2 –
- Wielowątkowe wykonywanie rozkazów dla 1 do 1536 jednocześnie aktywnych wątków
 - Wspólne pobieranie instrukcji dla 32 wątków tworzących paczkę (warp)
 - Wspólne dla wszystkich aktywnych wątków zasoby multiprocesora
- Max 8 aktywnych bloków na multiprocesor
- Max 48 aktywnych paczek

HIERARCHIA WYKONYWANIA PROGRAMU



- Wątek
 - Warp (32 wątki)
- Block (64-1024 wątków)
- Grid – max 64Kx64Kx64K bloków



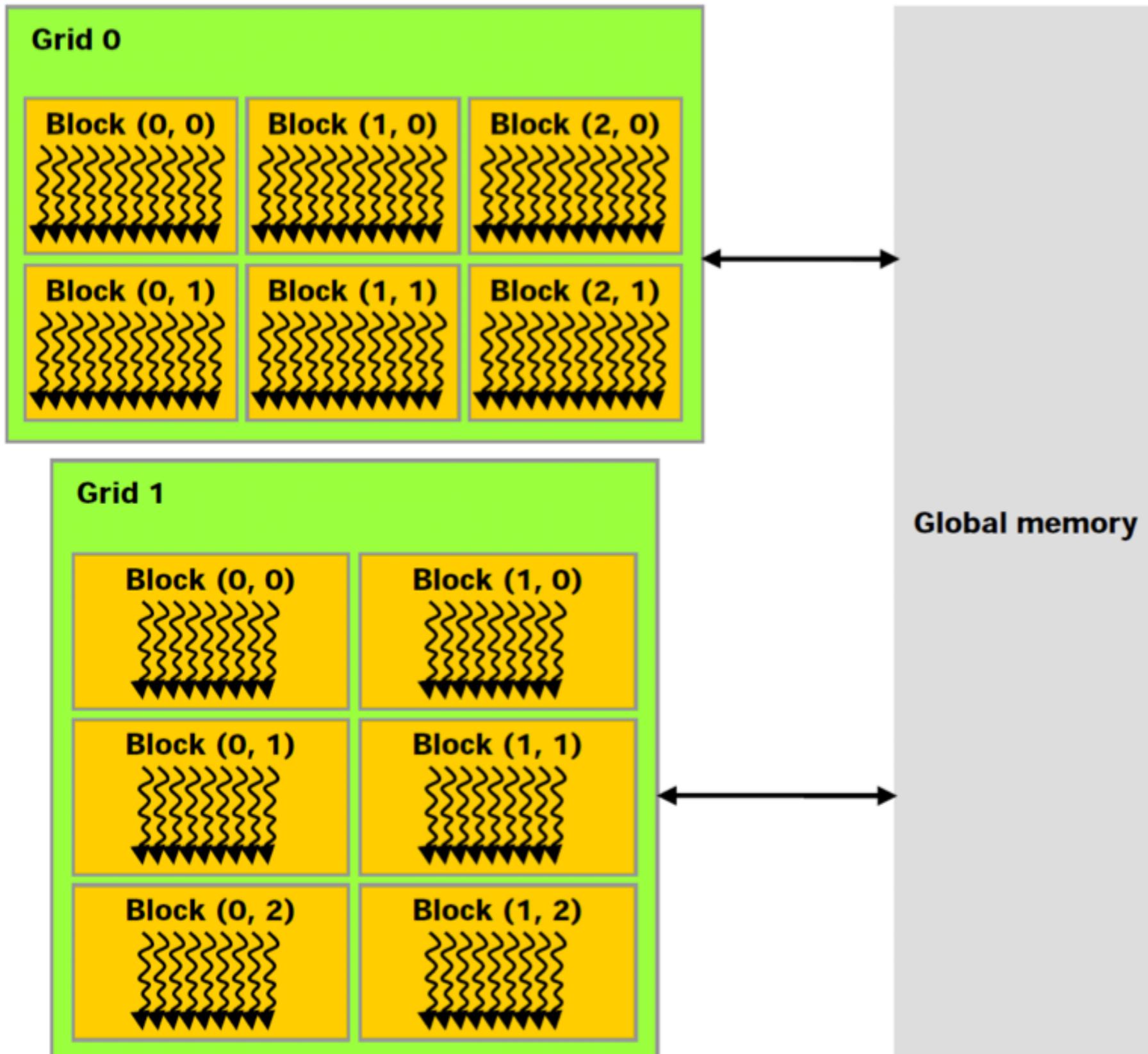
ARCHITEKTURA PAMIĘCI

Rodzaje pamięci

- Pamięć globalna
- Pamięć lokalna
- Pamięć współdzielona
- Pamięć stała
- Pamięć tekstur
- Pliki rejestrów
- Cache L1
- Cache L2

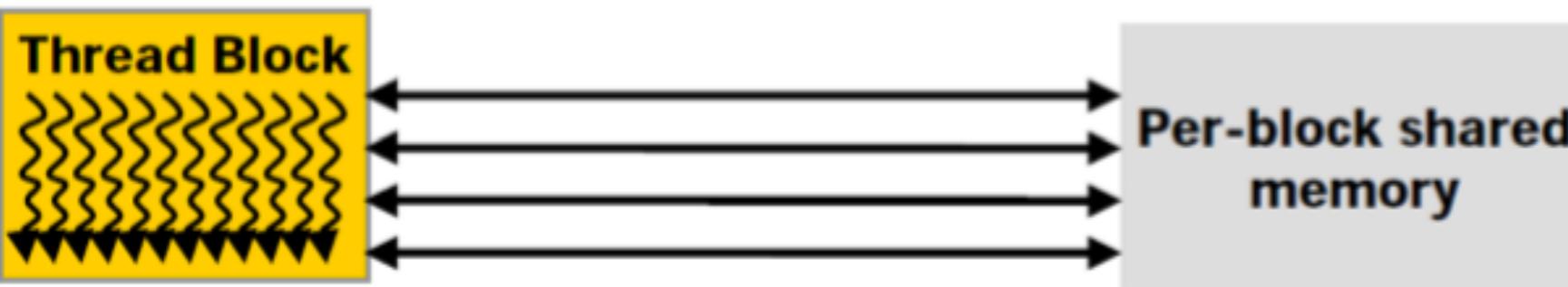
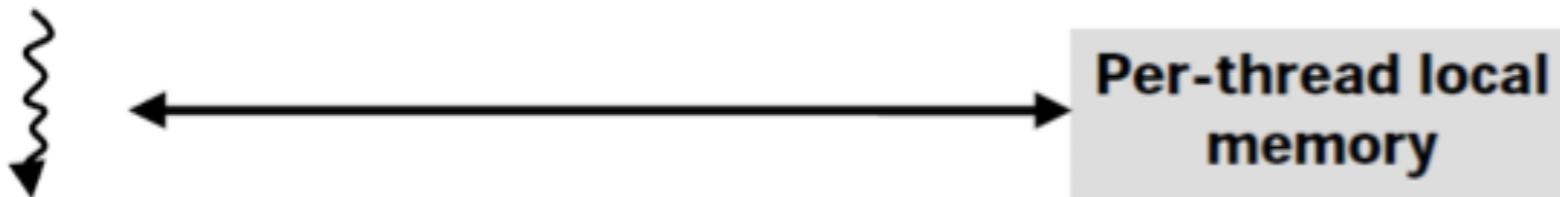


HIERARCHIA PAMIĘCI



HIERARCHIA PAMIĘCI

Thread



- Pamięć lokalna – prywatna dla wątku, wydzielony obszar pamięci RAM karty
- Pamięć współdzielona – widoczna dla wszystkich wątków w bloku pamięć wewnętrzna procesora
- Plik rejestrów – prywatna dla wątku ograniczona. Ilość rejestrów przypadających na jeden wątek zależy od ilości wątków.



PAMIĘĆ GLOBALNA

- On device
- Widoczna dla wszystkich wątków
- Opóźnienie dostępu rzędu 200 cykli zegara
- Cache L1 - 16kB/multiprocesor
- Cache L2 - 1.5MB (Kepler) 3MB (Maxwell)
- Dostęp RW
- Widoczna dla hosta

PAMIĘĆ LOKALNA

- On device
- Prywatna dla jednego wątku
- Opóźnienie dostępu rzędu 200 cykli zegara
- Nie ma cache'a
- Dostęp RW
- niewidoczna dla hosta

PAMIĘĆ TEKSTUR

- On device
- Wszystkie wątki
- Cache
- Dostęp Read Only
- Jedynie host może do niej pisać

PAMIĘĆ STAŁA

- On device
- Wszystkie wątki
- Cache
- Dostęp Read Only
- Jedynie host może do niej pisać

PAMIĘĆ LOKALNA - WSPÓŁDZIELONA

- On chip
- Wszystkie wątki w bloku
- Dostęp RW
- Niewidoczna dla hosta
- Dzielona między wszystkie jednocześnie aktywne bloki



PAMIĘĆ LOKALNA - CACHE LI

- On chip
- Niewidoczna dla programisty
- Dzielona między wszystkie jednocześnie aktywne bloki
- Użyteczna gdy występuje nadmiar rejestrów (register spill)

PLIK REJESTRÓW

- On chip
- Prywatne dla jednego wątku w bloku
- Dostęp RW
- Niewidoczna dla hosta
- Dzielona między wszystkie jednocześnie aktywne bloki
- 128 KB (32K rejestrów 32-bitowych) na multiprocesor

CACHE L2

- On chip
- Niewidoczna dla programisty
- 768K na chipie

EFEKTYWNE KODOWANIE

- Jednoczesna optymalizacja zużycia wielu zasobów
 - Minimalizacja dostępu do pamięci globalnej
 - Maksymalizacja liczby wątków



OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

- Latencja dostępu do pamięci głównej i lokalnej ~200-800 cykli zegara (zależy od architektury)
- Czas wykonania jednej instrukcji dla średnio 8 (Kepler i nowsze) lub 24 (Fermi i starsze) cykle zegara (po takim czasie wynik operacji będzie dostępny dla kolejnej operacji (read after write latency na rejestrach)
 - kolejne paczki na jednym multiprocesorze powinny wykonać 8-12 (18-27) instrukcji zanim bieżąca będzie otrzymała dane z pamięci globalnej.
 - Dla prostych kodów – maksymalizacja liczby wątków umożliwia chowanie latencji
 - Jeżeli kod jest „gęsty” obliczeniowo, lub operacje są bardziej kosztowne optymalna ilość wątków będzie mniejsza



OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

- Teoretyczna wydajność obliczeniowa karty GTX 280 wynosi 933 GFlops
- Operacja polegająca na wzięciu trzech argumentów z pamięci wykonaniu na nich mnożenia i zapisaniu wyniku wykorzystuje maksymalnie moc obliczeniową (MULADD + MUL)

$$A = B + C * D$$

$$E = F * G$$

- Jeżeli operacje odbywałyby się na zmiennych pobieranych z pamięci głównej i zapisywane do pamięci głównej wymagana przepustowość wynosi:

$$7 \times (\text{Operacje RW/ Cykl}) \times 4B (\text{Sizeof(float)}) \times 311G (\text{cykli /s}) = 8708 \text{ GB/s}$$

- Dostępna przepustowość wynosi $\sim 140 \text{ GB/s}$

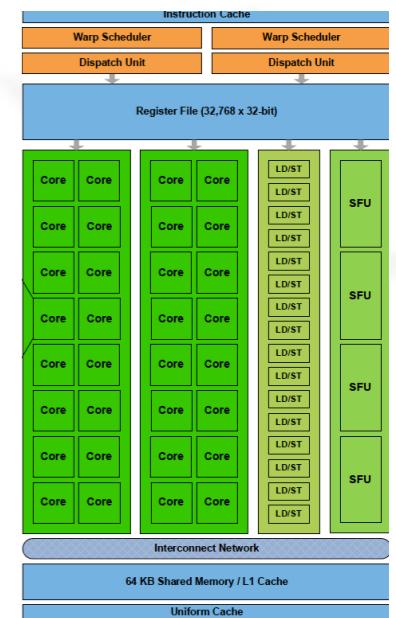
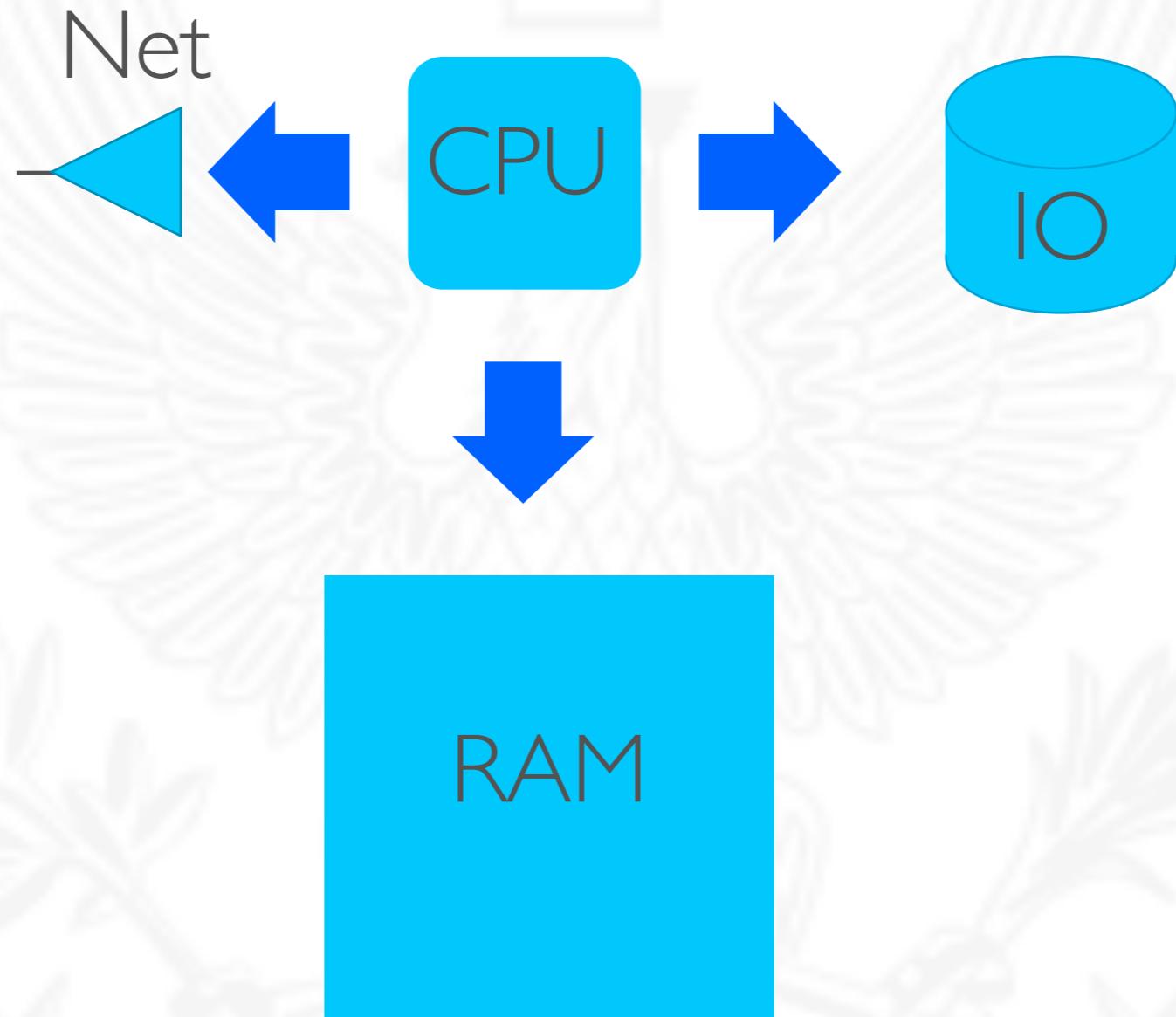


OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

- Należy minimalizować odwołania do pamięci głównej!
 - Wczytaj często używane dane do pamięci współdzielonej
 - Wykonaj obliczenia
 - Zapisz wyniki do pamięci głównej
- Ograniczenia
 - Niewielka ilość pamięci współdzielonej (48/16 KB na multiprocesor)

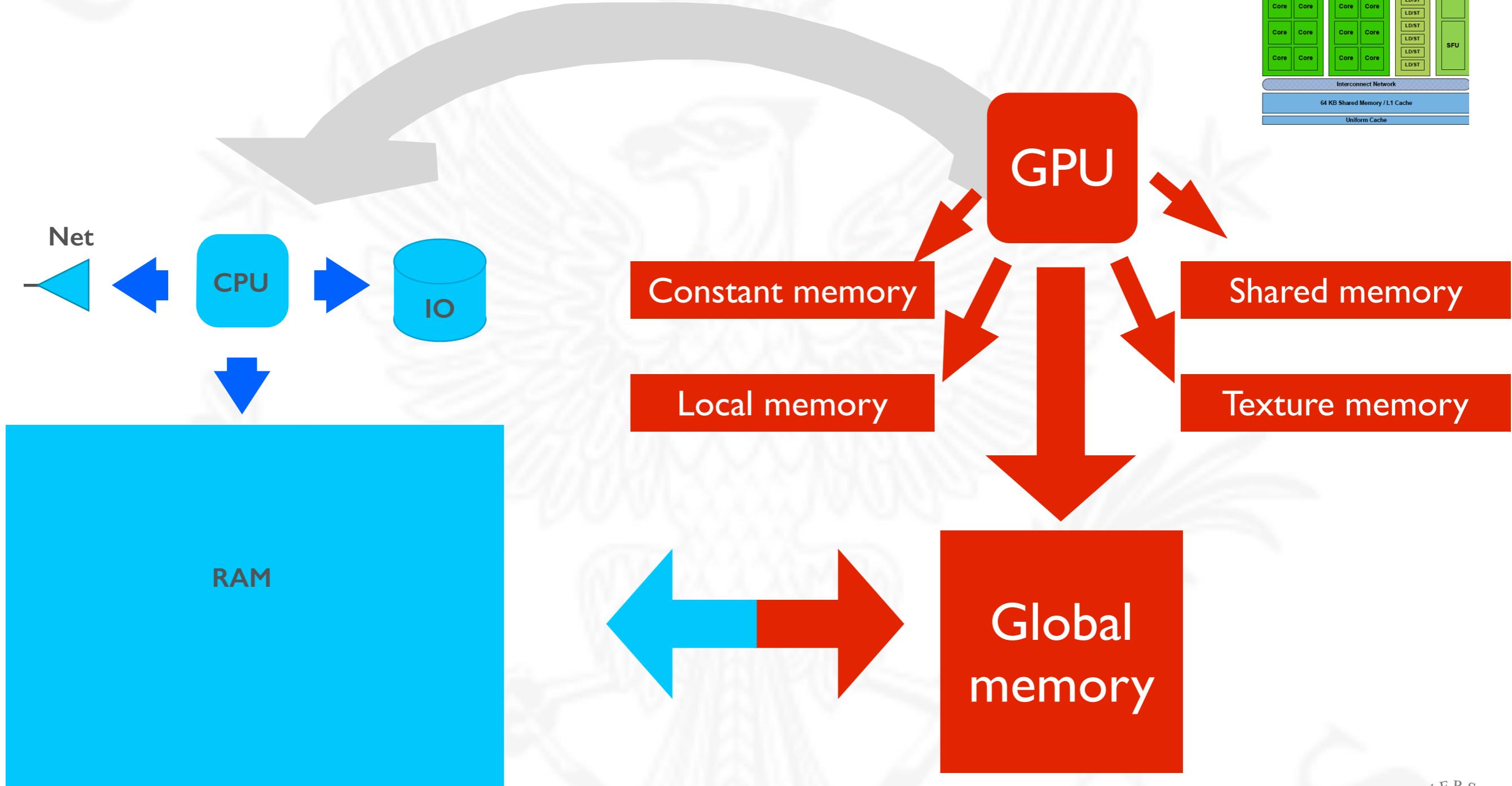
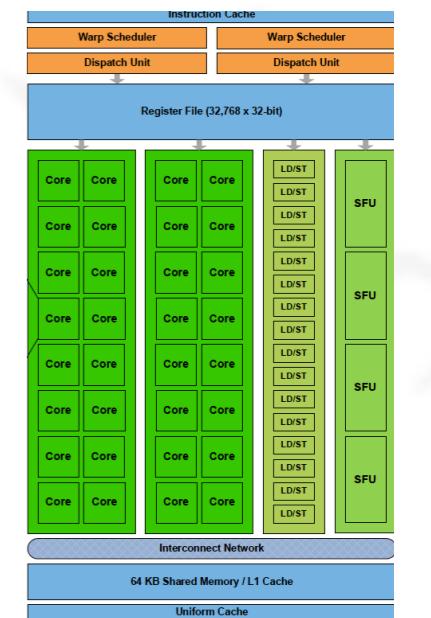
ARCHITEKTURA CPU

WIDOK PROGRAMISTY



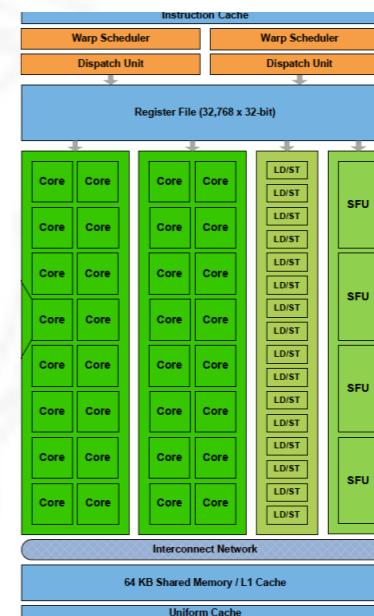
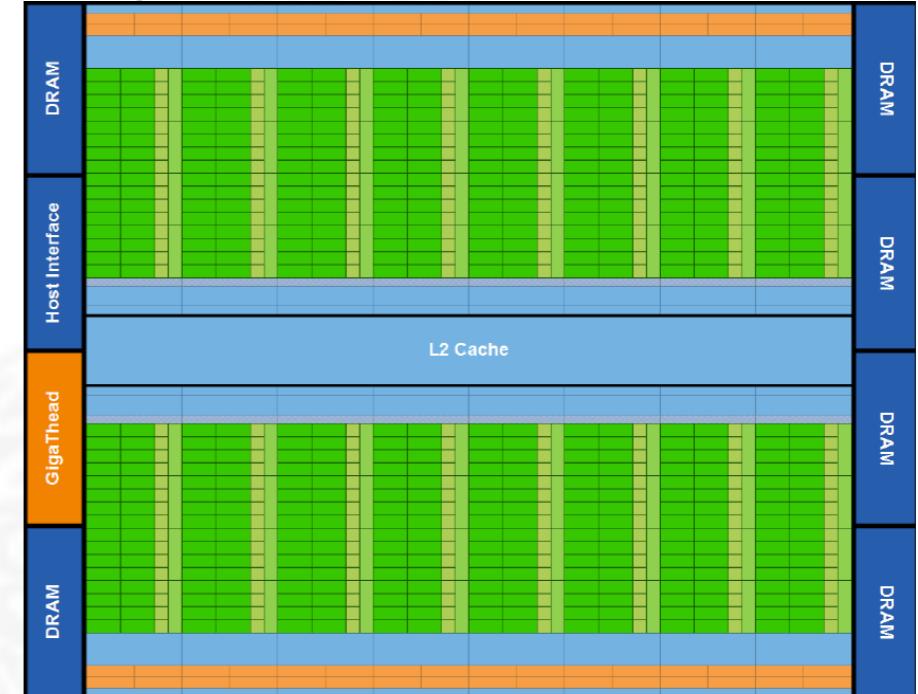
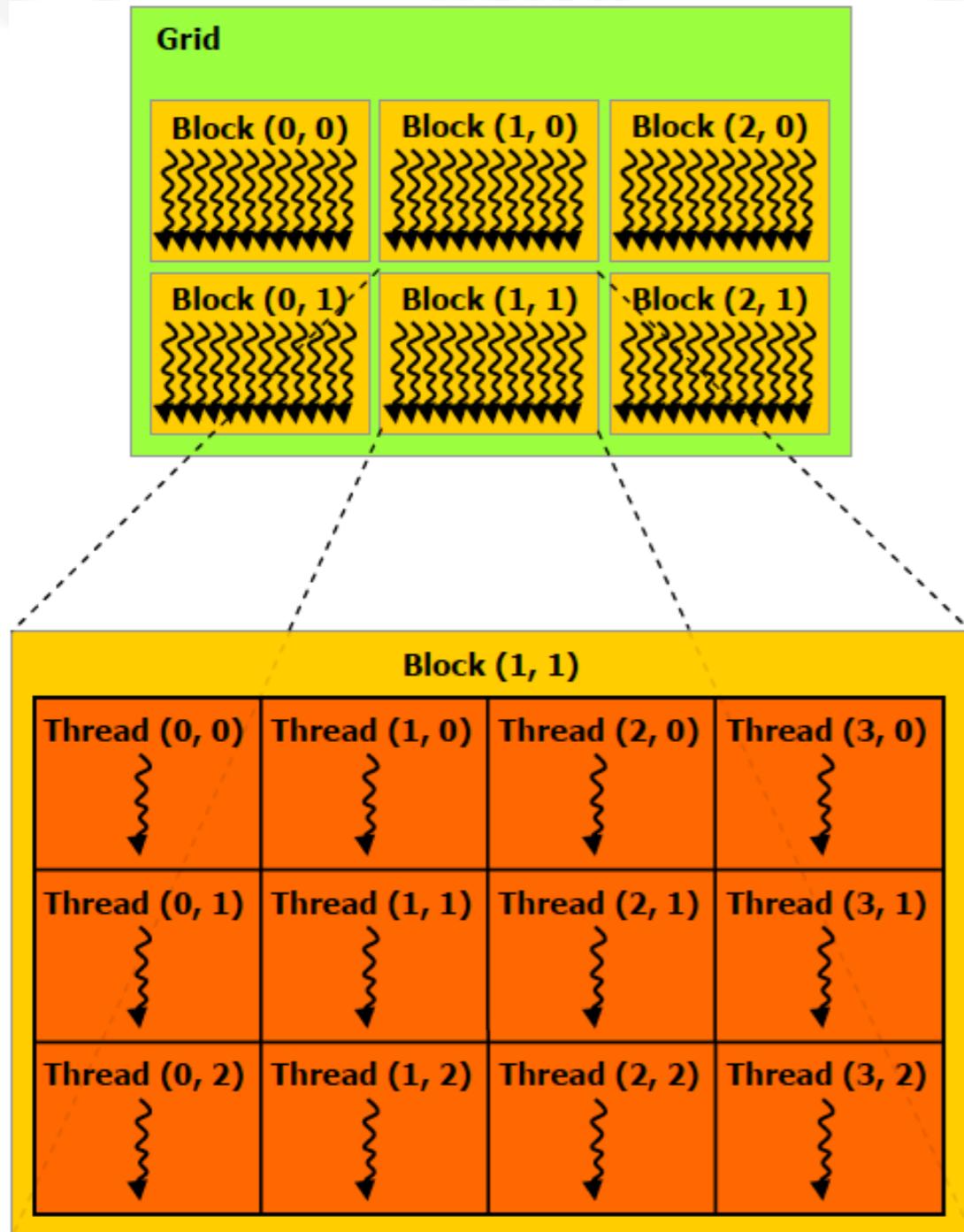
ARCHITEKTURA GPU

WIDOK PROGRAMISTY



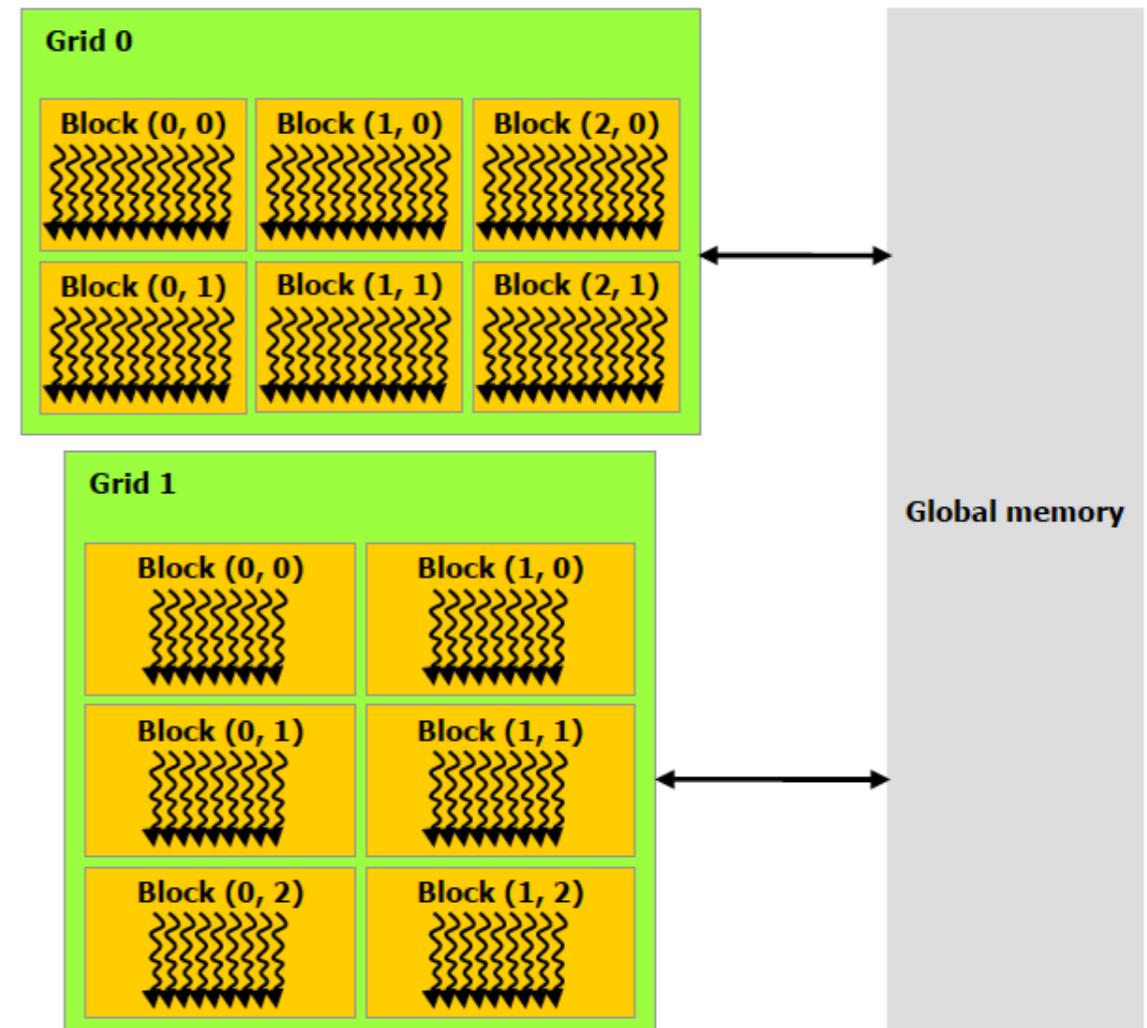
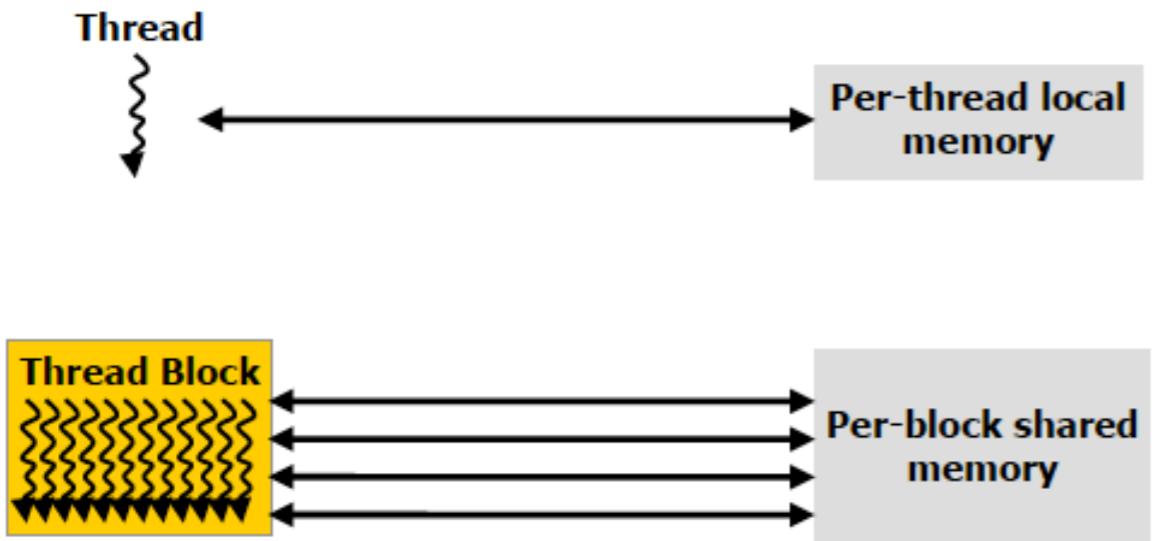
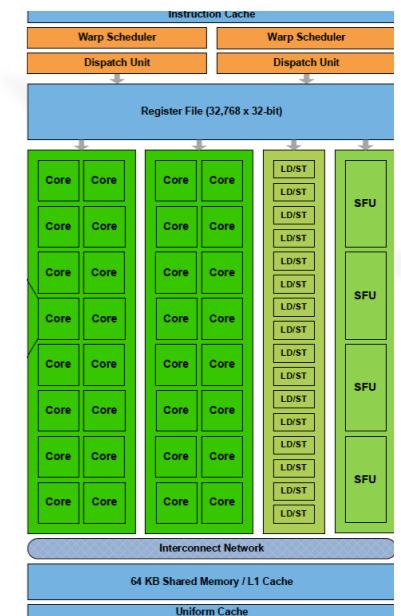
ARCHITEKTURA GPU

WIDOK PROGRAMISTY



ARCHITEKTURA GPU

WIDOK PROGRAMISTY



OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

- Dodatkowe optymalizacje dostępu do pamięci
- Zasada ogólna:
 - Dostęp jest optymalny, gdy kolejne wątki odwołują się do kolejnych lokalizacji w pamięci, tak żeby transakcja odwoływała się do ciągłego obszaru, zaczynającego się na granicy banku pamięci.

OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

- Odczyt z pamięci odbywa się za pomocą operacji 32B, 64B i 128B (8, 16 lub 32 int/float).
- Instrukcje dla kolejnych wątków mogą być sklejane dla uzyskania efektywnych transakcji z pamięcią.
- Najbardziej efektywny dostęp jest jeżeli uda się skleić odwołania pochodzące z kolejnych wątków w 32, 64 lub 128 bajtowe transakcje z pamięcią.
- Transakcje 32-bajtowe są używane przez urządzenia z compute capability 1.2 i więcej

OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

COMPUTE CAPABILITY 1.0 I I.I

- Aby sklejanie mogło się powieść wątki muszą używać zmiennych:
 - 32-bitowych (transakcja 64 bajtowa),
 - 64-bitowych (transakcja 128 bajtowa),
 - 128-bitowych (dwie transakcje 128 bajtowe);
- Wszystkie 16 zmiennych musi znajdować się w tym samym segmencie pamięci o wielkości odpowiadającej wielkości transakcji (podwójnej dla zmiennych 128 bitowych);
- Kolejność zmiennych musi odpowiadać kolejności wątków k-ty wątek w półosnowie musi odwoływać się do k-tej zmiennej.



Left: coalesced `float` memory access, resulting in a single memory transaction.
 Right: coalesced `float` memory access (divergent warp), resulting in a single memory transaction.

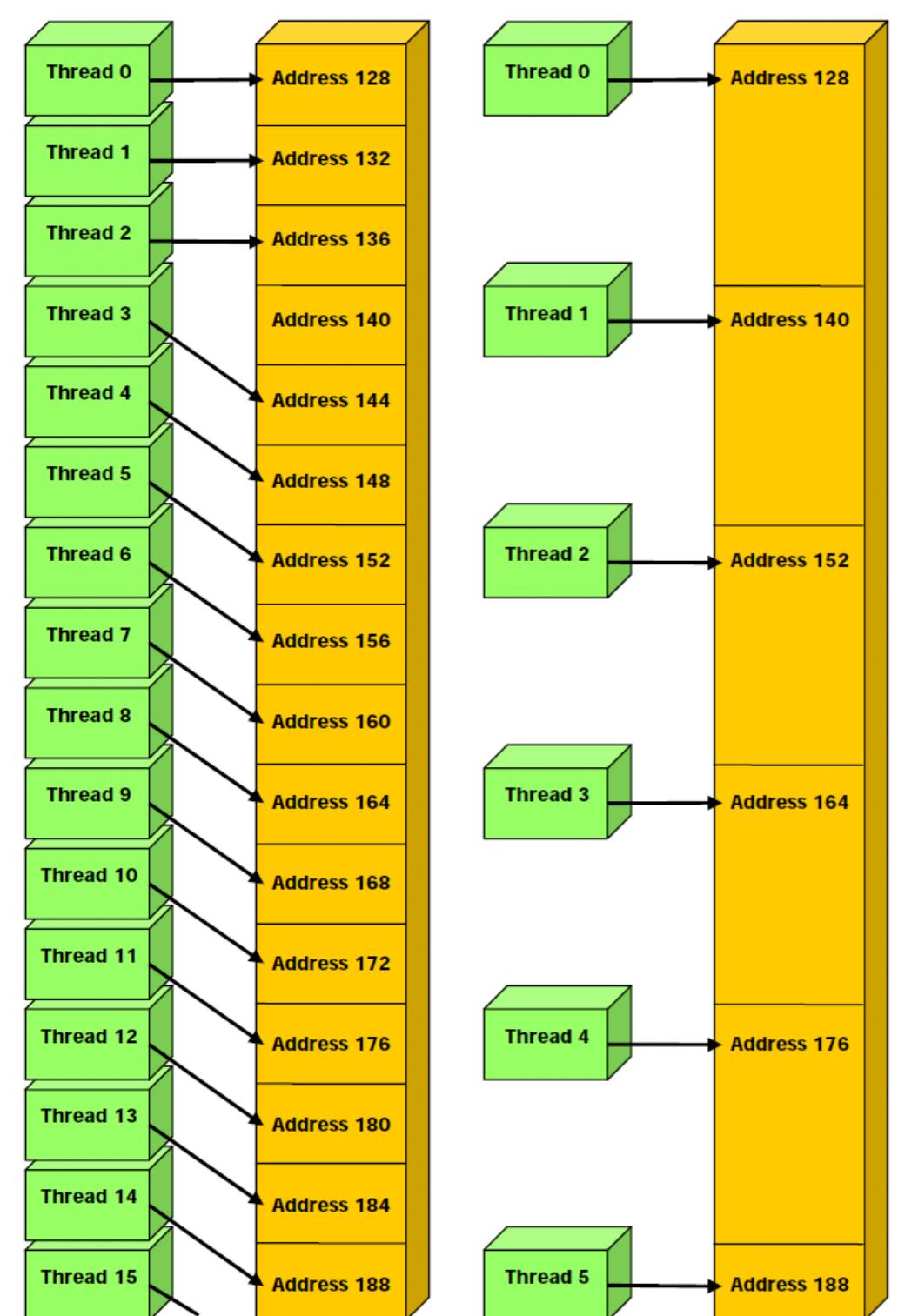
Dobrze –
 | transakcja

Figure 5-1. Examples of Coalesced Global Memory Access Patterns

OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

Compute capability 1.0 & 1.1

Źle - 16 oddzielnych transakcji



Left: non-contiguous `float` memory access, resulting in 16 memory transactions.

Right: non-coalesced `float3` memory access, resulting in 16 memory transactions.

Figure 5-3. Examples of Global Memory Access Patterns That Are Non-Coalesced for Devices of Compute Capability 1.0 or 1.1



Left: non-sequential `float` memory access, resulting in 16 memory transactions.

Right: access with a misaligned starting address, resulting in 16 memory transactions.

Figure 5-2. Examples of Global Memory Access Patterns That Are Non-Coalesced for Devices of Compute Capability 1.0 or 1.1

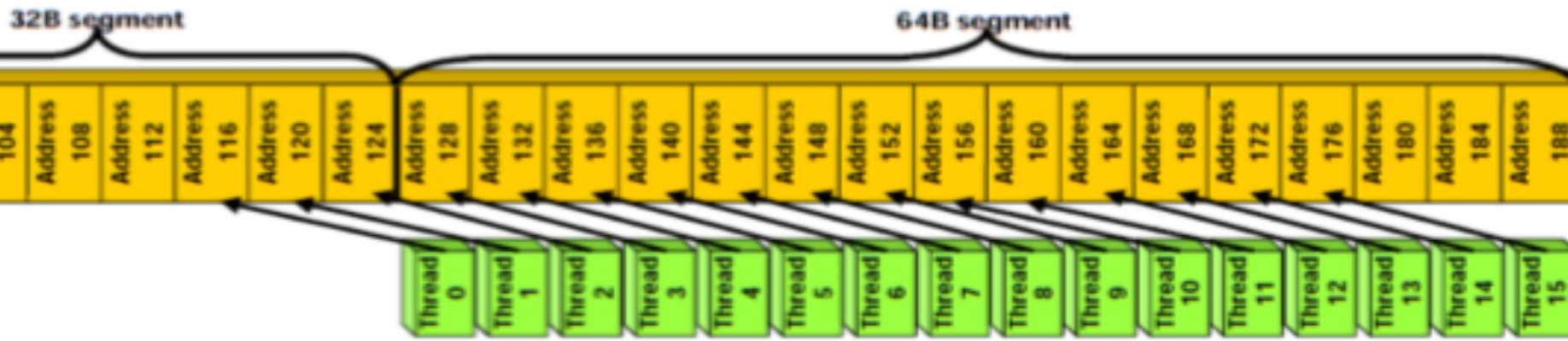
OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

OPTYMALIZACJA DOSTĘPU DO PAMIĘCI COMPUTE CAPABILITY 1.2 | WIĘCEJ

- Aby sklejanie mogło się powieść wszystkie 16 zmiennych musi znajdować się w tym samym segmencie o wielkości:
 - 32 bajty (zmienne 8-bitowe),
 - 64 bajty (zmienne 16-bitowe),
 - 128 bajtów (zmienne 32 i 64 bitowe);
- Restrykcje dotyczące kolejności wątków zostały zniesione
- Restrykcje dotyczące granic segmentów zostały częściowo zniesione

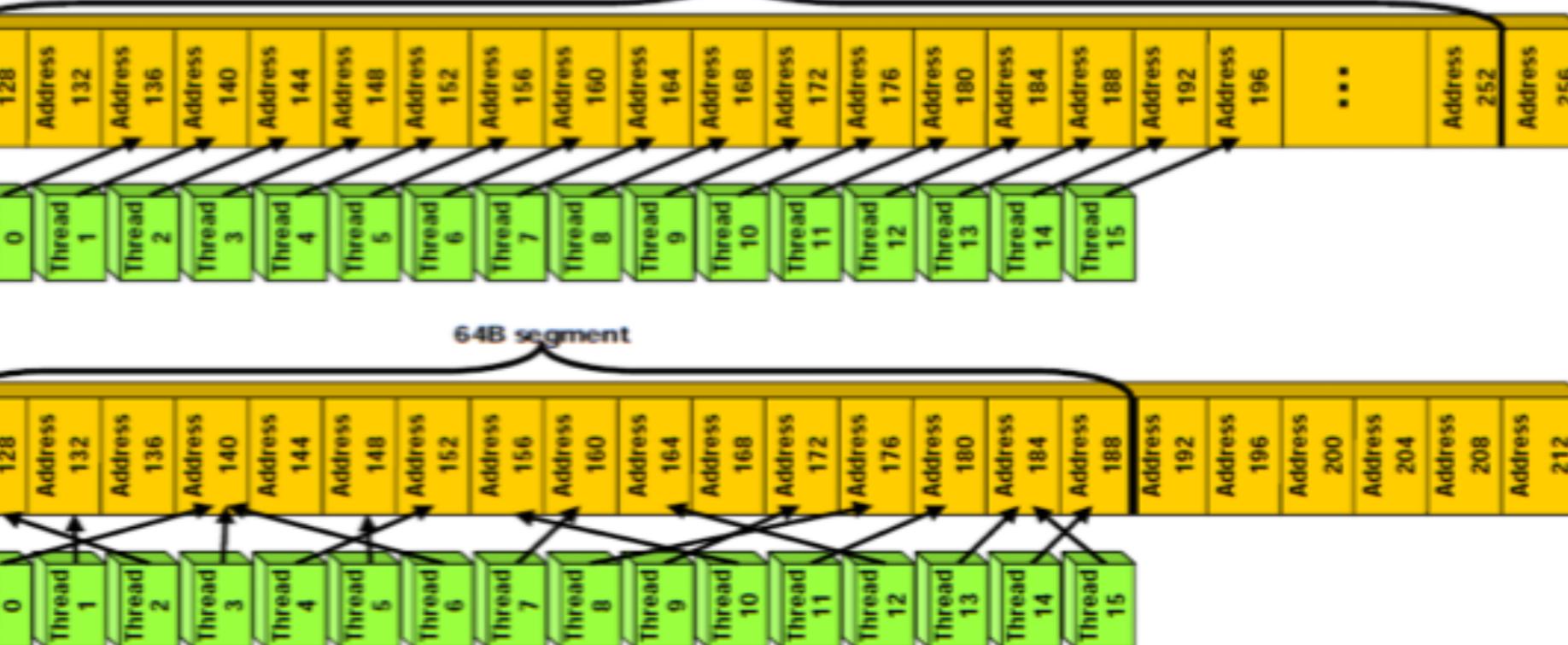


OPTYMALIZACJA DOSTĘPU DO PAMIĘCI COMPUTE CAPABILITY 1.2



Dwie
transakcje

Jedna
transakcja



OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

Pamięć lokalna

- Zmienne automatyczne, dla których nie ma miejsca w rejestrach
- Umieszczana w pamięci karty
- Jest buforowana w cache'u,
- Problemy z wydajnością jeśli wychodzi poza cache
- Odwołania są zawsze sklejone (dba o to kompilator).

OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

Pamięć stała

- Umieszczana w pamięci karty
- Jest buforowana w cache'u,
- Wszystkie wątki w paczce (warp) powinny czytać z tego samego adresu – wtedy szybkość odczytu z cache'a jest równa szybkości korzystania z rejestrów

Miejsce na przechowywanie stałych

OPTYMALIZACJA DOSTĘPU DO PAMIĘCI

Pamięć tekstur

- Umieszczana w pamięci karty
- Jest buforowana w cache'u,
- Nie zmniejsza latencji (ale lepsze ukrywanie latencji)
- Hit zmniejsza użycie szyny pamięci – może zwiększyć przepustowość innych obliczeń
- Nie ma ograniczeń dotyczących adresów, z których można czytać (przydatne dla dostępu w losowej kolejności)
- Zalecane jest by wątki z tej samej paczki (warp) czytały z bliskich adresów.



OPTYMALIZACJA – UWAGI OGÓLNE

- Moc obliczeniowa wielokrotnie większa niż dostępna przepustowość
- Optymalizacja to przede wszystkim algorytmy optymalnie wykorzystujące pamięć
- Jednoczesne żonglowanie wieloma parametrami

OPTYMALIZACJA – UWAGI OGÓLNE

- Zwiększanie ilości wątków
 - Pomaga ukryć latencję pamięci głównej
- ALE
 - Zwiększa zużycie cennych lokalnych zasobów
 - Rejestrów
 - Pamięci współdzielonej
- Pamięć współdzielona jest mała i dzielona przez bardzo dużą liczbę wątków
- Rejestry – mogą być ograniczeniem przy bardziej wymagających kodach
- Warto pamiętać o pamięci constant i pamięci tekstur

ARCHITEKTURA GPU

WIDOK PROGRAMISTY

