Procesory kart graficznych i CUDA

wer 1.3 14.12.2016 PR

Litreratura:

- CUDA textbook by David Kirk from NVIDIA and Prof. Wen-mei Hwu from UIUC.
- CUDA w przykładach. Wprowadzenie do ogólnego programowania procesorów GP, J.Sanders, E.Kandrot, Helion
- Getting Started with CUDA, Greg Ruetsch, Brent Oster, NVIDIA
- Technical Brief NVIDIA GeForce GTX 200 GPU Architectural Overview
- Inne źródła

Problemy postępu technologii procesorów
sytuacja 10 lat temu i wcześniej

Problemy:

- Energia - tania,
- układy scalone – drogie,
- obliczenia - wolne,
- dostęp do pamięci – wolny.

Rozwiązania:

- wzrost prędkości obliczeń uzyskiwany dzięki współbieżności przetwarzania na poziomie instrukcji – realizacja:
  - analiza kodu przez kompilatory,
  - dynamiczna kolejność przetwarzania instrukcji,
  - przetwarzanie spekulatywne,
  - zastosowanie pamięci podręcznej procesora.

Problemy postępu technologii procesorów

sytuacja obecna

Problemy:

- „Przeszkoda energetyczna” – energia - droga,
- układy scalone – tanie
- „Przeszkoda pamięci”: obliczenia - szybkie, dostęp do pamięci - wolny,
  - (przykładowo 200 cykli - dostęp do DRAM (pamięć operacyjna), 4 cykle - mnożenie FP)

Rozwiązanie to wzrost efektywności poprzez:

- współbieżność wątków - równoległość wieloprocesorowa i wielordzeniowa – ogranicza zużycie mocy
- Układy z procesorami SIMD dla równoległości danych (jednakowe operacje na wielu jednostkach danych), a w procesorach kart graficznych: SIMD, łączenie dostępów do pamięci w transakcje i szybkie przełączanie wątków (ukrycie kosztów dostępu do pamięci), wiele wątków.

Zasady ogólne:

Compute Unified Device Architecture (NVIDIA) - CUDA

Sposób na programowanie PKG (GPU, karty graficzne):

- model programowania i
- środowisko dla programowania równoległego

CUDA zawiera:

- specyfikację architektury sprzętu,
- język programowania,
- interfejs języka dla aplikacji czyli API – **runtime API** lub device API.

Zasady ogólne:

Cechy CUDA

Dostęp wątków do pamięci:

- wątek może realizować dostęp do dowolnej lokacji pamięci globalnej karty,
- wątki mogą współdzielić dane w pamięci współdzielonej.

Niski nakład pracy wejścia niezbędny dla poznania CUDA:

- niewielkie rozszerzenie języka C,
- brak wymagań na znajomość zagadnień graficznych.

Umożliwia wykorzystanie systemów niejednorodnych - CPU + karta graficzna (GPU) o rozdzielonych pamięciach. (notacja: CPU -host, GPU – device)

Zasady ogólne:

Współpraca wątków

Znaczenie współpracy:

- współdzielenie wyników
  - brak konieczności wielokrotnych obliczeń
- korzystanie ze współdzielonej pamięci (w odróżnieniu od globalnej) pozwala na:
  - obniżenie wymagań na przepustowość pamięci głównej
- możliwość efektywnej współpracy (pamięć współdzielona, synchronizacja) w ramach mniejszych grup wątków – „blokach wątków”
- ograniczenie możliwości współpracy w ramach dowolnie dużej grupy wątków (limit na wielkość bloku wątków) – często na danym etapie obliczeń każdy „blok wątków” realizuje niezależne obliczenia.

Przetwarzanie w modelu CUDA

Krótka legenda:

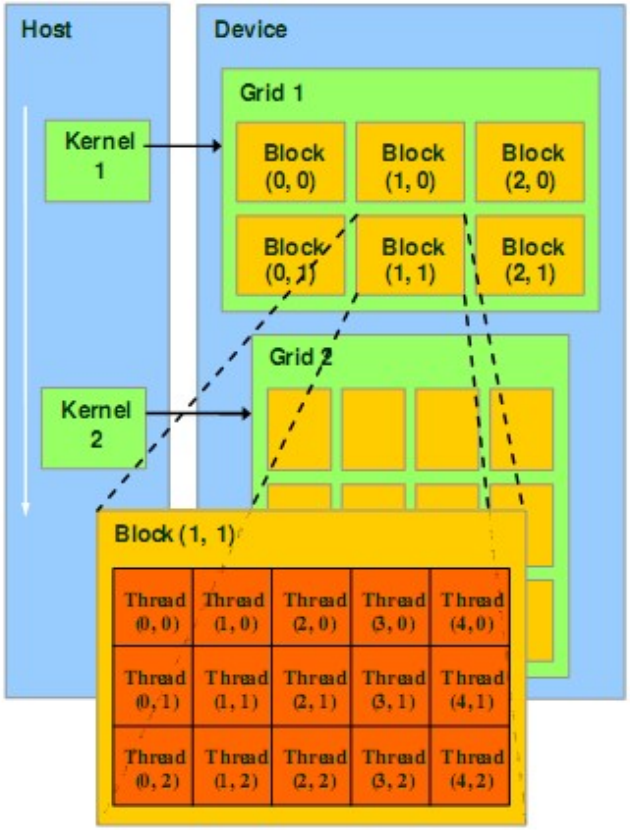
Multiprocesor (SM) – fragment struktury przetwarzającej - skład się z wielu rdzeni, jednostek przetwarzających specjalnego przeznaczenia i pamięci współdzielonej.

Kernel – funkcja – kod realizowanych przez PKG (brak współbieżności kerneli, nowsze karty możliwa).

Grid – grupa wątków realizująca kernel (struktura logiczna max 2 wymiarowa).

Blok wątków – grupa wątków - część struktury Gridu (max 3 wymiarowy) – blok jest przetwarzany na jednym multiprocesorze.

Wiązka wątków – grupa wątków (max 32) – część **bloku** wątków, przetwarzana jednocześnie – wątki wiązki wykonują ten sam rozkaz (lub żaden). Wykorzystuje osnowę (warp) - grupę 32 jednostek wykonawczych – do obliczeń. Wiązka wątków realizowana na osnowie często nazywana jest również warp.



Przetwarzanie w modelu CUDA

Wywołanie w kodzie komputera funkcji kernela umożliwia uruchonienie gridu składającego się z bloków wątków.

Wątki tego samego bloku realizowane są na tym samym SM:

- współpracują przez pamięć współdzieloną,
- mogą się efektywnie synchronizować.

Przetwarzanie bloków wątków jest przenośne i skalowalne – przetwarzanie można realizować na różnych kartach scharakteryzowanych wymaganą zgodnością obliczeniową karty dla jakiej kompilacji dokonano – compute capability.

GPU filozofia architektury (1)

Każdy rdzeń procesorowy jest procesorem wielowątkowym z wieloetapowym potokiem przetwarzania realizującym operacje stało, zmiennoprzecinkowe i logiczne oraz dostępy do pamięci dla przetwarzanych wątków.

Przetwarzanie GPU jest skoncentrowane na obliczeniach i przepustowości, stosowana metodologia:

przerwanie przetwarzania wiązki wątków na czas oczekiwania na dane, w tym czasie obliczana inna wiązka wątków.

Przetwarzanie CPU jest zorientowane na redukcję opóźnienia i zapełnienie potoków przetwarzania, stosowana metodologia:

dążenie do wysokich współczynników trafień i efektywnej predykcji rozgałęzień kodu, dynamiczna zmiana kolejności instrukcji.

GPU filozofia architektury (2)

Stopień wykorzystania struktur logicznych przez jednostki funkcjonalne CPU i GPU.

W GPU znacznie więcej - niż w CPU - tranzystorów jest wykorzystywanych do implementacji ALU, niż do sterowania przetwarzaniem i pamięci podręcznej.

Efekt: wymaganie na większą niż w CPU przewidywalność w dostępie do danych i przepływie sterowania dla programów efektywnych w GPU.

Równoległość zadań a równoległość danych

- Równoległość zadań
  - Niezależne procesy z niewielką ilością komunikacji
  - Łatwa do implementacji:
    - współczesne systemy operacyjne w komputerach z pamięcią współdzieloną + OpenMP
- Równoległość danych
  - wiele jednostek danych na których realizowane mają być te same obliczenia,
  - brak zależności między danymi w poszczególnych krokach obliczeń,
  - możliwość wykorzystania wielu jednostek przetwarzających

– lecz: konieczność przeprojektowania tradycyjnych algorytmów

GPU - filozofia architektury (4)

CPU

– szybkie pamięci podręczne (nadają się do algorytmów z ponownym wykorzystaniem danych)

– wiele różnych procesów/wątków

– wysoka efektywność przetwarzania pojedynczego wątku (warunek - lokalność dostępów)

GPU - PKG

– wiele jednostek wykonawczych,

– efektywny (przepustowość!, opóźnienie?) dostęp do pamięci urządzenia,

– realizacja tego samego kodu na wielu elementach danych,

– wysoka przepustowość dla wielu obliczeń równoległych (warunek – jednakowe operacje dla wiązki wątków)

Zatem:

–CPU właściwe dla równoległości zadań

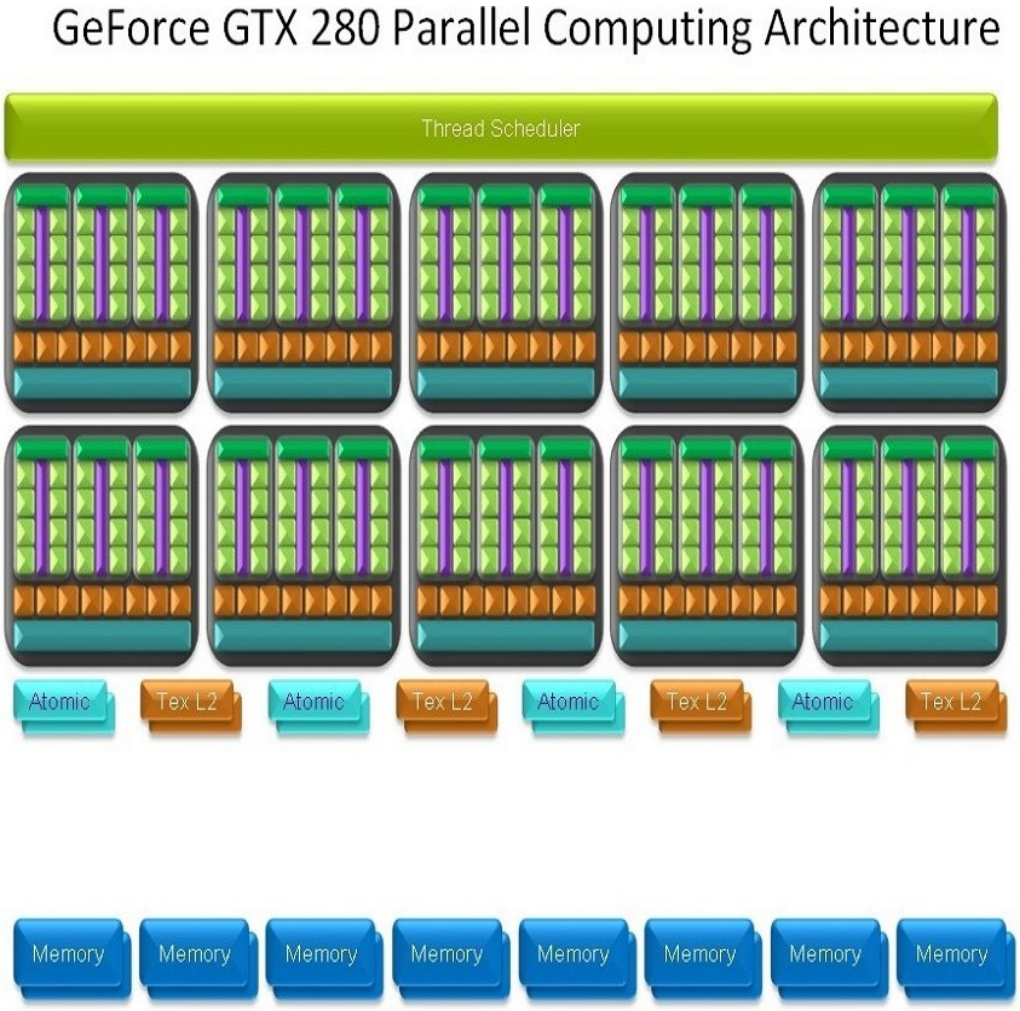
–GPU właściwe dla równoległości danych

Sprzęt używany na laboratorium

Architektura serii 10 – nazwa serii Tesla- (GTX 260) :

- 240 procesorów wątkowych pogrupowanych w
- 30 miltiprocesorów SM, każdy SM to
  - 8 rdzeni - procesorów wątkowych
  - jedną jednostkę FP podwójnej precyzji
  - 2 SFU (szybkie +*, dodatkowo f. wykładnicza, logarytm, pierwiastek)
  - pamięć współdzieloną umożliwiającą współpracę wątków z bloku
- Zgodność obliczeniowa 1.2

GeForce GTX 200 specyfikacja (2)



Struktura układów serii GeForce GTX 200

- Sprzętowy moduł szeregujący wątki do TPC (thread processing cluster).
- Pamięci podręczne tekstur i jednostki interfejsu pamięci dla efektywnego dostępu do pamięci.
- Jednostki dostępu atomowego read-modify-write, umożliwiają równoległe redukcje i zarządzanie równoległymi strukturami danych.

GeForce GTX 200 GPU specyfikacja (1)

W architekturze GTX 200 występują TPCs (10 lub 9):

- "Texture Processing Clusters" w graficznym trybie przetwarzania,
- „Thread Processing Clusters" w trybie przetwarzania równoległego.

- TPC składa się z 3 multiprocesorów strumieniowych - SM.
- SM zawiera 8 rdzeni procesorowych SP (procesorów strumieniowych).
- Wzór na całkowitą liczbę rdzeni procesorowych – SP:

  $x_{TPC} * y_{SM} * Z_{SP}$ (liczba klastrów*liczba SM w klastrze*liczba rdzeni w SM)

Dla GeForce GTX 260 GPU: 216 rdzeni procesorowych - 9 TPC, 3 SM w TPC i 8 SP w SM.

GeForce GTX 200 GPU specyfikacja (3)

Struktura TPC (thread processing cluster)

Thread Processing Cluster (tryb obliczeniowy karty)

Lokalne - dla każdego SM - pamięci współdzielone.

Każdy rdzeń (SP) może współdzielić dane z innymi rdzeniami tego samego SM bez konieczności odczytu i zapisu pamięci globalnej.

Każdy SM zawiera moduły TFP (texture filtering processors) używane zarówno w przetwarzaniu graficznym jak i obliczeniowym.

MIMD w ramach TPC i GPU !

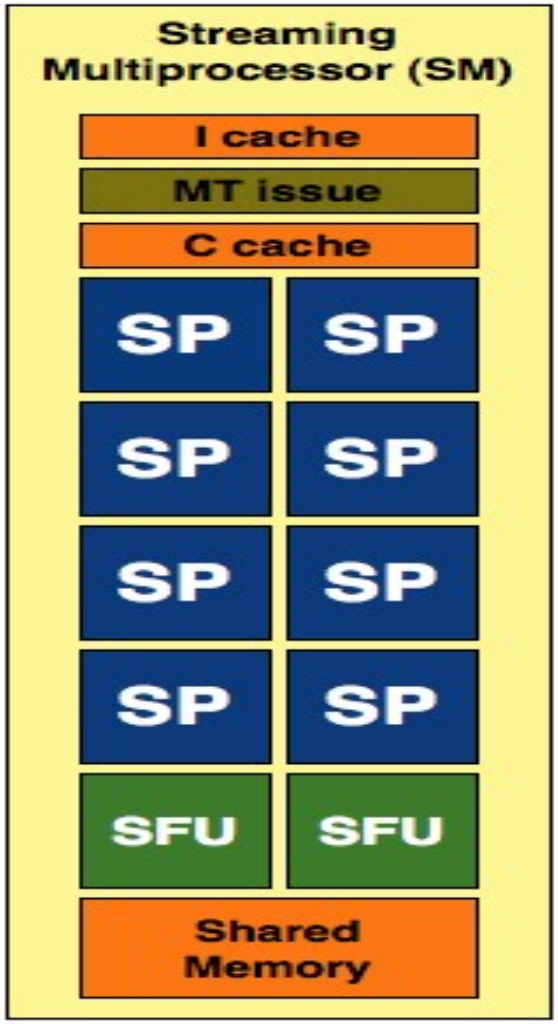
SIMT (single instruction, multiple thread) w ramach warpa (w ramach SM GTX200)

GeForce GTX 200 GPU - organizacja przetwarzania

Metodologia przetwarzania SIMT (single instruction, multiple thread)

- GeForce GTX 200 GPU umożliwia jednoczesne przetwarzanie „w locie” ponad 30 tysięcy wątków.
- Sprzętowe szeregowanie wątków.
- Architektura jest odporna na opóźnienia dostępu: gdy jedna wiązka wątków oczekuje na dostęp do pamięci następuje zerokosztowe, sprzętowe przełączenie kontekstu i przetwarzanie gotowych wątków innej wiązki.
- Jednostka zarządzająca przetwarzaniem w ramach SM tworzy, zarządza, szereguje i wykonuje wątki w grupach – wiązki - po max 32 współbieżne wątki – wykonywane w ramach osnowy – udostępnione zasoby (ang. warp). Wątki wiązki należą do jednego bloku (liczba wiązek może być zatem większa niż wynikająca z zadeklarowanej przy uruchomieniu kernela liczby wątków).
- Pojęcia: warp (struktura przetwarzająca) i wiązka (grupa wątków przy jego użyciu przetwarzana) często stosowane zamiennie.
- Maksymalnie 32 wiązki przetwarzane jednocześnie w SM (GTX 200) - może wystąpić do 32x32 współbieżnych wątków „przetwarzanych w locie” przez jeden SM.

GeForce GTX 200 – SM struktura i organizacja przetwarzania



SM jest tablicą 8 SP (rdzeni) i 2 procesorów SFU - Special Function Units.

SFU posiada 4 jednostki mnożenia zmiennoprzecinkowego używane do funkcji sin, cos, interpolacji, filtrowania tekstury i innych operacji.

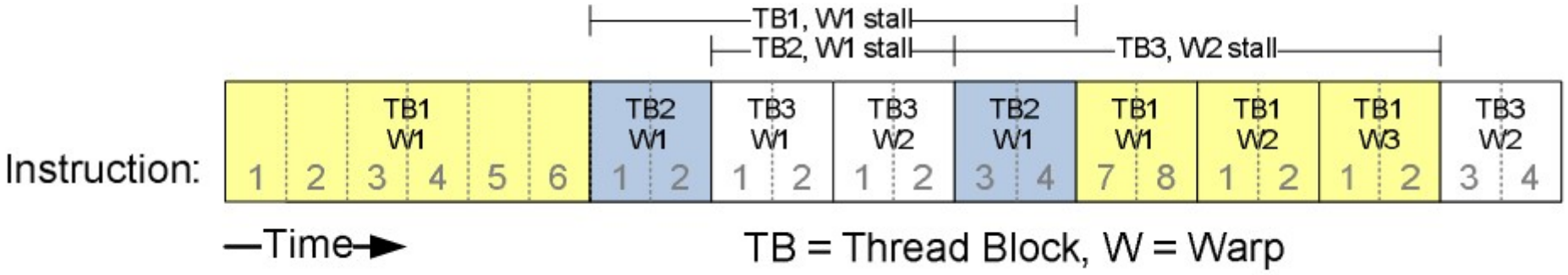
Pamięć: pamięć podręczna instrukcji, podręczna danych do odczytu i 16KB współdzielonej pamięci R/W.

Jednostka szeregowania (MT issue unit) dostarcza instrukcje do wszystkich SP i SFU w SM, obsługując cyklicznie kolejne wątki - części tego samego *warp.*
Maksymalna wydajność to:
3 operacje zmiennoprzecinkowe na cykl na jeden rdzeń („przeliczeniowy" multprocesora – jeden z 8).
FP - Szczytowo wydajność 27 (SM)* 8 (SP) * 3 (flops)*1,4 10^9 (częstotliwość) = ok. 900 GFlops
FP podwójna precyzja – każdy SM wyposażony jest w jedną jednostkę matematyczną 64 bitową podwójnej precyzji – razem 30 jednostek w GTX 200.

Szeregowanie przetwarzania

1. Poszczególne **bloki** wątków są rozdzielane do przetwarzania na SM, potencjalnie więcej niż jeden blok, lecz <= 8 (równocześnie) CC 1.3 (Tesla)
2. SM realizuje obliczenia jednej wiązki (techn. Tesla) wątków z jednego z przydzielonych *bloków*, pozostałe wiązki oczekują na swoją kolej lub dane.
3. Zwolnienie zasobów przydzielonych do *bloku* – rejestry, pamięć - następuje po zakończeniu przetwarzania *bloku* – daje możliwość przetwarzania kolejnego bloku.

Szeregowanie wątków w ramach bloków watków i jednostek warp w SMP



Przykładowa (Tesla) kolejność dotyczy przetwarzania dla jednego multiprocesora, kolejno przetwarzane:

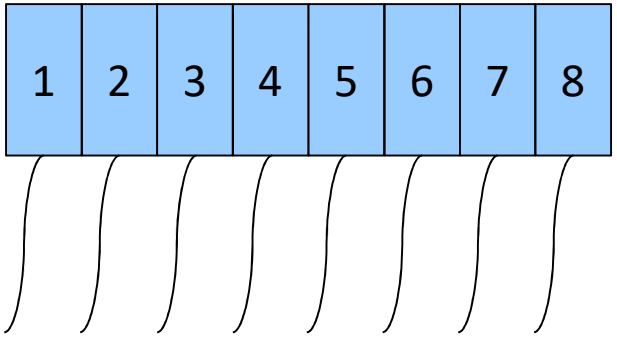
Warp 1 bloku 1, Warp 1 bloku 2, Warp 1 bloku 3, Warp 2 bloku 3…

TB1, W1 stall – okres utknięcia warp W1 bloku B1

Kernel jest realizowany przez grid - tablicę wątków:

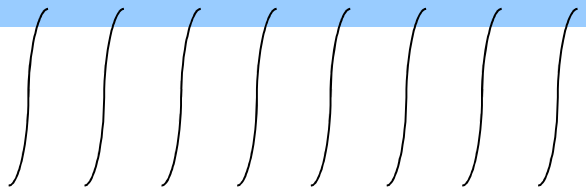
- wszystkie otrzymują ten sam kod
- każdy posiada unikalny w gridzie identyfikator używany do obliczenia wykorzystywanego adresu pamięci i sterowania przebiegiem obliczeń

Wątki i ich identyfikatory - threadID

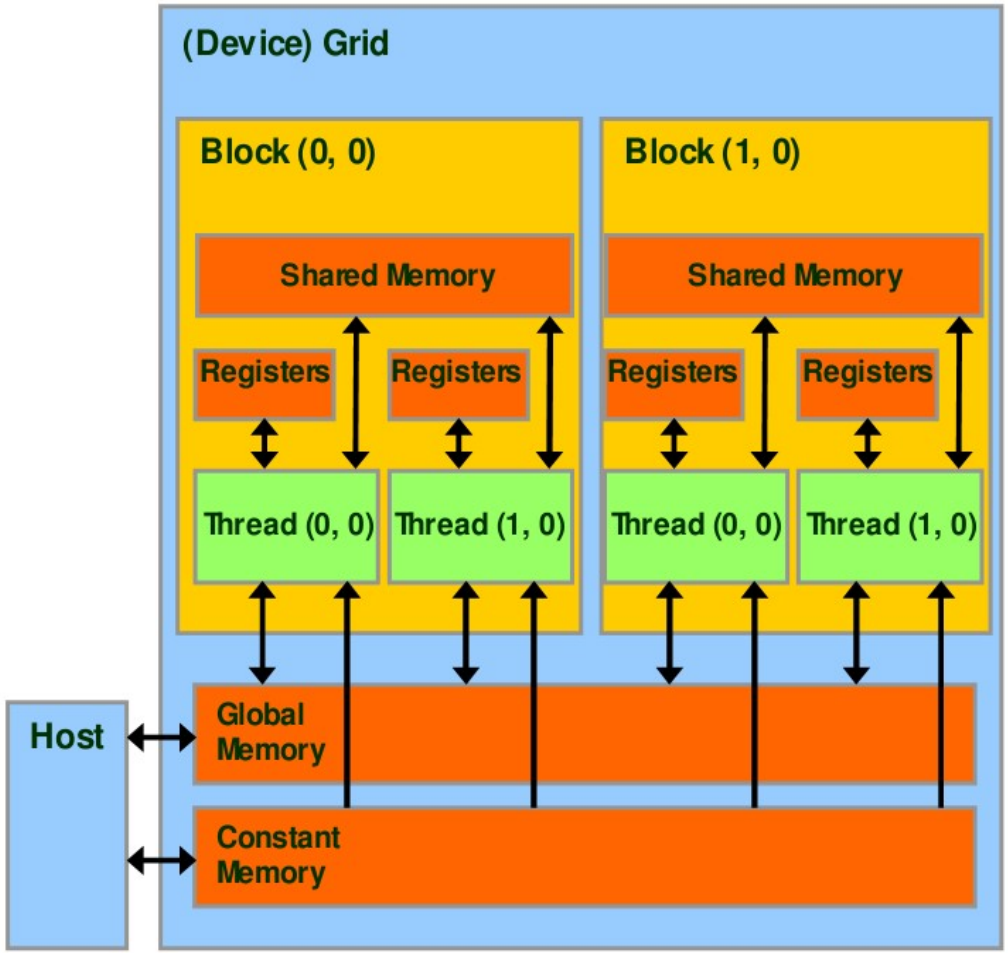


Przykład kodu kernela:

```
…
float x = input[threadID];
float y = func(x);
 output[threadID] = y;
…
```



Model pamięci karty – CUDA



Możliwe dostępy do pamięci w ramach kodu:

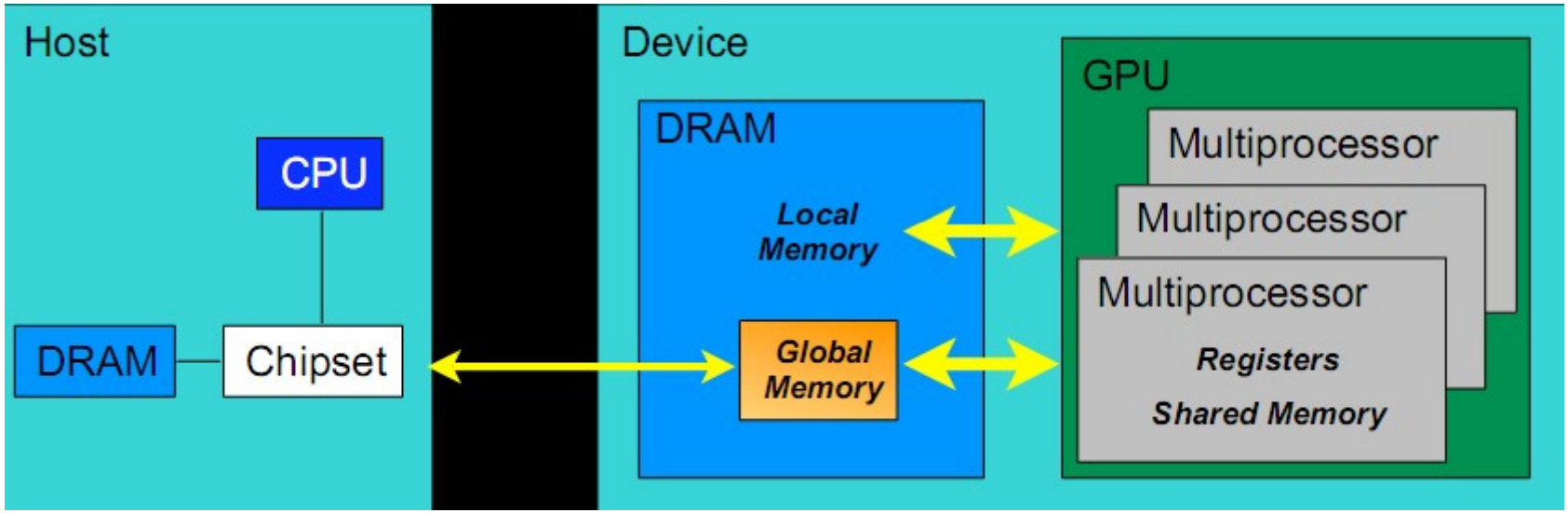
Kernel (wątki dla PKG)

- R/W rejestru w ramach wątku (lokalnie – szybki dostęp)
- R/W pamięci prywatnej wątku (w pamięci globalnej karty – powolny dostęp)
- R/W pamięci współdzielonej w ramach bloku wątków (szybki dostęp)
- R/W pamięci globalnej w ramach gridu (w pamięci globalnej karty – powolny dostęp)
- odczyt pamięci wartości stałych (constant) w ramach gridu

Kod komputera dostępowego (host)

- R/W pamięci gridu globalnej i pamięci stałych

Struktura pamięci systemu CPU-GPU



Kod CPU zarządza pamięcią GPU - alokacja, zwolnienie, kopiowanie danych pomiędzy pamięciami: host do device oraz device do host.

Operacje na pamięci GPU – obsługa danych

cudaMalloc(void ** pointer, size_t nbytes)

cudaMemset(void * pointer, int value, size_t count)

cudaFree(void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc( (void**)&a_d,  nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

Kopiowanie daych CPU - GPU

cudaMemcpy(void *dst, void *src, size_t nbytes,
enum cudaMemcpyKind direction);

direction określa kierunek transferu danych

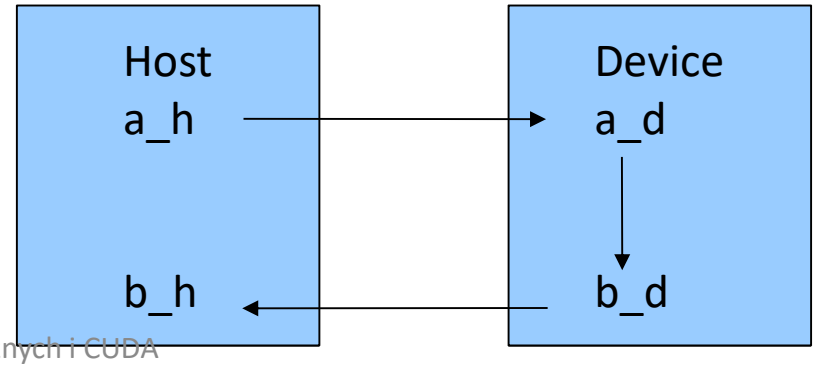
enum cudaMemcpyKind
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice

Funkcja ta **blokuje** przetwarzanie na CPU, kończy się po zakończeniu przesłania.

Kopiowanie rozpoczyna się typowo po zakończeniu poprzedniego wywołania CUDA.

Przykład kodu dla transferu danych

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;
  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);
  for (i=0, i<N; i++) a_h[i] = 100.f + i;
cudaMemcpy(a_d, a_h, nBytes,
    cudaMemcpyHostToDevice);
 cudaMemcpy(b_d, a_d, nBytes,
    cudaMemcpyDeviceToDevice);
cudaMemcpy(b_h, b_d, nBytes,
    cudaMemcpyDeviceToHost);
  for (i=0; i< N; i++) assert( a_h[i] ==
b_h[i] );
  free(a_h); free(b_h);
cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



Kwalifikatory funkcji CUDA

Kernel określony za pomocą: __global__

- Funkcja wywoływana w kodzie host i wykonywana w device, musi:
  - zwracać void, posiadać stałą liczbę argumentów
  - argumenty automatycznie kopiowane z CPU do GPU
- nie może:
  - być rekursywna i
  - odwoływać się do pamięci CPU

Inne kwalifikatory:

__device__

- Funkcja wywoływana w kodzie device i wykonywana w device.

__host__

- Funkcja wywoływana i realizowana na host

Można używać __host__ i __device__ równocześnie.

Wywołanie kernela - parametry konfiguracyjne

1. Zmodyfikowana składnia wywołania funkcji dla PKG (CUDA):

   nazwa_funkcji <<<dim3 opis_gridu , dim3 opis_bloku>>> (parametry_kernela);

2. Konfiguracja uruchomienia określona jest poprzez „<<< >>>”
3. opis_gridu – oznacza: liczbę wymiarów i wielkość gridu (tj. rozmiary) podane w jednostkach bloku, max 2(3) wymiary – x, y, z liczba bloków uruchomionych w grid wynosi opis_gridu.x * opis_gridu.y
4. opis_bloku – oznacza: liczbę wymiarów i wielkość bloku (tj. rozmiary) w wątkach, max 3 wymiary bloków – x,y,z, liczba wątków na blok wynosi zatem opis_bloku.x * opis_bloku.y * opis_bloku.z

   Nie określone wartości typu dim3 zostają automatycznie zainicjowane do wartości 1.

5. Kompilator zamienia to wywołanie w blok kodu, który zostaje skonfigurowany (ustalona struktura i liczba wątków) i zostanie uruchomiony asynchronicznie na PKG.

Wywołanie kernela - przykłady

```
dim3 grid, block;
 grid.x = 2; grid.y = 4;
 block.x = 8; block.y = 16;
 nazwa_kernela<<<grid, block>>>(...);

dim3 grid(2, 4), block(8,16);  // to samo za pomocą funkcji tworzącej
 nazwa_kernela<<<grid, block>>>(...); // 8*128=1024 wątki

 nazwa_kernela<<<32,512>>>(...);
// 1 wymiarowy grid - 32 bloki, jednowymiarowy blok wątków po 512
    wątków
```

Zmienne automatyczne konfiguracji –
wątek dowiaduje się swojwgo ID i określa pracę

Wszystkie funkcje __global__ i __device__ mają dostęp do automatycznie zdefiniowanych zmiennych o następujących nazwach i typie :

```
dim3 gridDim;       // ile bloków w wymiarze gridDim.x, gridDim.y
dim3 blockDim;      // ile wątków w wymiarze blockDim.x ...
dim3 blockIdx;      //który blok     blockIdx.x <= gridDim.x-1
dim3 threadIdx;     //który wątek  threadIdx.x <= blockDim.x-1
```

Nie można przyporządkowywać wartości zmiennych automatycznych i pobierać adresu, wykorzystywane są np. do adresowania pamięci i określania pracy wątku.

Zmienne automatyczne konfiguracji przykłady

Dla **jednowymiarowego** gridu i bloku o parametrach:

blockIdx.x, blockDim.x, threadIdx.x

globalny identyfikator wątku wyliczony wg wzoru:

blockIdx.x*blockDim.x + threadIdx.x

Ogólnie liczba wątków dowolnego gridu:

gridDim.x *gridDim.y*blockDim.x*blockDim.y*blockDim.z

Inkrementacja tablicy porównanie kodu CPU-GPU

program CPU

```
void inc_cpu(int *a, int N)
{
  int idx;
  for (idx = 0; idx<N; idx++)
    a[idx] = a[idx] + 1;
}
int main()
{ …
 inc_cpu(a, N);
}
```

program CUDA

```
__global__ void inc_gpu(int *a, int N)
{
int idx = blockIdx.x * blockDim.x + threadIdx.x;
 if (idx < N)  a[idx] = a[idx] + 1;
//grid i blok jednowymiarowe
}
 int main()
{ …
dim3 dimBlock (blocksize);
dim3 dimGrid( ceil( N / (float)blocksize)  );
     inc_gpu<<<dimGrid, dimBlock>>>(a, N);
}    // sufit dla uzyskania wystarczającej liczby wątków
```

Określniki rodzaju zmiennych (kernel)

___device__

Zmienne przechowywane w pamięci globalnej (duża pamięć, długi czas dostępu, brak buforowania (tzn. brak pp w ogólności)), alokowana przez cudaMalloc, dostępna dla wszystkich wątków, przez czas życia aplikacji.

__shared__

Zmienne przechowywane w pamięci współdzielonej SM (szybki czas dostępu), dostępne przez wątki tego samego bloku, dostępne w czasie istnienia bloku, jedna kopia obiektu na blok wątków, rozmiar specyfikowany przy kompilacji lub w konfiguracji uruchomienia, synchronizacja jest niezbędna do uzyskania obrazu wykonanych operacji również w ramach innych watków bloku (lub deklaracja volatile), dane są niedostępne z CPU.

Określniki rodzaju zmiennych (kernel)

__constant__

Globalne zmienne przechowywane w pamięci GPU, GPU odczyt, CPU API odczyt/zapis

bez określnika

Zmienne lokalne wątku (typy skalarne i wbudowane typy wektorowe) przechowywane w rejestrach lub (gdy się nie mieszczą) w pamieci globalnej - „local memory”.

Użycie pamięci współdzielonej

Rozmiar znany podczas kompilacji

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

Rozmiar znany przy wywołaniu

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

Pobranie parametrów używanego systemu GPU (1)

Za pomocą wytwołania deviceQuery

CUDA Device Query (Runtime API) version (CUDART static linking)
Found 1 CUDA Capable device(s)
Device 0: "GeForce GTX 260"
CUDA Driver Version / Runtime Version 4.2 / 4.2
CUDA Capability Major/Minor version number: 1.3
Total amount of global memory: 896 MBytes (939524096 bytes)
(27) Multiprocessors x (8) CUDA Cores/MP: 216 CUDA Cores
GPU Clock rate: 1400 MHz (1.40 GHz)
Memory Clock rate: 1000 Mhz
Memory Bus Width: 448-bit
Max Texture Dimension Size (x,y,z) 1D=(8192), 2D=(65536,32768), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers 1D=(8192) x 512, 2D=(8192,8192) x 512
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 16384 bytes

Specyfikacja parametrów GPU (2)

Total number of registers available per block:	16384
Warp size:	32
Maximum number of threads per multiprocessor:	1024
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1
Maximum memory pitch:	2147483647 bytes
Texture alignment:	256 bytes
Concurrent copy and execution:	Yes with 1 copy engine(s)
Run time limit on kernels:	Yes
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Concurrent kernel execution:	No

Ograniczenia fizyczne dla GPU zgodności obliczeniowej 1.3

Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512

Wg CUDA_Occupancy_calculator.xls

Zasady optymalizacji algorytmu dla GPU (1)

1. Algorytm oparty na niezależnej równoległości przetwarzania.
2. Maksymalizacja intensywności obliczeń arytmetycznych.
3. Miara intensywności to współczynnik CGMA (compute to global memory access ratio) czyli liczba (niezbędnych) operacji na danych dzielona przez liczbę umożliwiających je dostępów do pamięci globalnej
   - Możliwe jest że ponowne obliczenia będą tańsze (czasowo) niż pobranie z pamięci, gdyż GPU dostarcza użytkownikowi przede wszystkim wiele ALU, a nie szybkiej pamięci.
4. Unikać kosztownych transferów do pamięci CPU.

Zasady optymalizacji algorytmu dla GPU (2)

5. Minimalizacja użycia zasobów przez wątek dla umożliwienia wielu blokom wątków aktywności w ramach SM.
   - Zapewnienie wystarczającej ilości pracy warpom poprzez zapewnienie dużej liczby wiązek przydzielonych do SM
6. Optymalizacja dostępów (UWAGA: inna niż dla CPU!) do pamięci globalnej dla wykorzystania maksymalnej przepustowości pamięci.
7. Optymalizacja kodu dla maksymalizacji prędkości obliczeń (np. rozwijanie pętli – usuwanie instrukcji sterujących).

Optymalizacja dostępu do pamięci – zasady

- Dostęp do pamięci globalnej wątków jednej połowy warp (GTX240) może być zrealizowany jako jedna transakcja z pamięcią czyli jednokrotne przesłanie szeroką magistralą danych (dostępy łączone - coalesced - od koalicja)
- Coalesced vs. Non-coalesced = różnica rzędu w wielkości czas dostępu Niełączone dostępy wymagają wielu transferów do 16 zamiast 1

- Unikanie konfliktów dostępu do tych samych banków w pamięci współdzielonej (dostępy równoczesne (czyli realizowane przez warp) najlepiej pod sąsiednie adresy)
- Unikanie nierównomiernie rozproszonego (wśród partycji pamięci) dostępu do pamięci globalnej – partition camping.

Łączenie dostępów dla CC 1.3

Odwołania do pamięci głównej realizowane jako jedna transakcja jeżeli adresacja generowana przez wątki w ramach pół-warp dotyczy segmentu pamięci o ograniczonej wielkości:

- segment wielkości 32B (wyrównany) dla 1 bajtowych słów,
- segment wielkości 64B (wyrównany) dla 2 bajtowych słów,
- segment wielkości 128B (wyrównany) dla 4- i 8-bajtowych słów
- Dotyczy dostępu do segmentów wyrównanych do granicy określonej przez rozmiar segmentu.

Wywoływana jest jako pierwsza transakcja, która dostarcza danych dla aktywnego wątku (z pół warp) o najmniejszym threadID. Z tego samego segmentu równocześnie obsługiwane są inne żądania. Następne obsługiwane są pozostałe żądania dostępu.

Mniejsze transakcje (ilość danych) mogą być uruchamiane dla zapobieżenia utraty przepustowości magistrali (możliwej ze względu na transfer niepotrzebnych słów).

Łączenie dostępów ważne dla zapobieżenia utraty dostępnej w systemie przepustowości pamięci.

Efektywność przetwarzania

3 przykładowe - możliwe sytuacje

1. Przepustowość dostępu do pamięci za mała powoduje obniżenie efektywności obliczeń ze względu na brak danych.
2. Przepustowość dostępu do pamięci wystarczająca – obniżenie efektywności obliczeń ze względu na brak danych - powód opóźnienie dostępu (czas na dostarczenie danych zbyt długi).
3. Przepustowość dostępu do pamięci wystarczająca – wysoka efektywności obliczeń ze względu na wystarczającą liczbę wiązek wątków, które posiadają dane gotowe do obliczeń.

Liczba wątków niezbędnych do efektywnego przetwarzania zależna od CGMA – parametr ten określa jak długo średnio (liczba operacji) zajmujemy procesor pobranymi z pamięci globalnej danymi.

Zalety pamięci współdzielonej

bloku wątków (w SM)

- Sto razy szybszy czas dostępu do pamięci współdzielonej niż czasu dostępu do pamięci globalnej.
- Możliwość szybkiej współpracy wątków bloku (wymiana danych) .
- Użycie grupy wątków do załadowania (jeden dostęp do lokacji) danych z PG do PW i wykonanie obliczeń (wiele dostępów) na danych współdzielonych przez wszystkie wątki; skąd to znamy – pamięć podręczna lecz obsługiwana wprost.
- Użycie do zapobieżenia dostępom niepołączonym
  - Etapy pobrań lub zapisu przy użyciu pamięci współdzielonej realizowane dla „uporządkowania adresowania” nie dających się połączyć dostępów (np. wynikajacych z algorytmu).

Synchronizacja przetwarzania bloku wątków w SM

void __syncthreads();

- synchronizacja wątków **bloku**,
- wprowadza barierę dla wątków **bloku**,
- używana do zapobiegnięcia wyścigowi w dostępie do pamięci współdzielonej multiprocesora,
- musi wystąpić w wykonywanym kodzie wszystkich wątków bloku,
- niepotrzebna dla bloków <= 32, pojedynczych wiązek

Struktura pamięci współdzielonej dla CC 1.3

- Pamięć współdzielona posiada 16 banków, kolejny bank jest odpowiedzialny za kolejne 32 bitowe słowo.
- Przepustowość banku wynosi 32 bity na 2 cykle zegara.
- Konfliktu dostępu nie ma jeśli (dla każdej połowy warp młodszej i starszej) dostępy odbywają się do innych banków.
- W przypadku zapisów przez kilka wątków do tej samej lokacji (z instrukcji nieatomowych) odbywa się tylko jeden zapis pochodzący z połowy warp.
- Możliwe są równoczesne: jedno rozgłaszanie danych do wielu wątków danych czytanych spod tego samego adresu (zmienna 32 bitowa) oraz odczyty z różnych banków.

Znaczenie optymalizacji wykorzystania zasobów multiprocesora (SM)

Wielkość wykorzystanych przez kernel rejestrów, pamięci lokalnej, współdzielonej i stałych można poznać za pomocą programu NVIDIA profiler oraz podczas kompilacji za pomocą opcji linii wywołania kompilatora ***nvcc*** --ptxas-options=-v

Liczba rejestrów używanych przez kernel może mieć duży wpływ na liczbę bloków rezydujących na multiprocesorze.

Przykład: CC 1.3 16k 32 bitowych rejestów w SM

- kernel wymaga - 16 rejestrów dla każdego wątku,
- Blok 512 wątków,
- można przydzielić 2 bloki które 2*512*16 wyczerpią wszystkie dostępne rejestry.
- wzrost zapotrzebowania na liczbę rejestrów o 1 na wątek zmniejsza liczbę przydzielonych bloków do SM o jeden, a przydzielonych warp o połowę !! (z 32 do 16).

Synchronizacja przetwarzania CPU GPU

Wszystkie uruchomienia funkcji kernel są asynchroniczne:

- sterowanie wraca natychmiast do kodu CPU, możliwe jest równoczesne przetwarzanie CPU i GPU
- funkcja kernel jest realizowana po zakończeniu wcześniejszych wywołań CUDA (w ramach tego samego strumienia wywołań).

cudaMemcpy();

- funkcja jest synchroniczna, sterowanie wraca do kodu CPU po jej zakończeniu,
- kopiowanie jest realizowane po zakończeniu wszystkich wcześniejszych (ten sam strumień) wywołań CUDA.

cudaThreadSynchronize() ;

- przerywa przetwarzanie kodu CPU do momentu zakończenia wcześniejszych wywołań CUDA

Równoczesność transmisji danych (HOST DEVICE) i obliczeń (GPU, CPU)

Wywołania asynchroniczne funkcji API i strumienie pozwalają na nakładanie się komunikacji i obliczeń

- Obliczenia CPU mogą się nakładać z transferem danych na wszystkich typach GPU (transfer asynchroniczny)
- Obliczenia GPU mogą się nakładać z transferem danych na kartach z "Concurrent copy and execution" (od zgodności obliczeniowej>= 1.1) - realizacja w różnych strumieniach.

Strumień (Stream) = sekwencja operacji realizowana na GPU

- Operacje z różnych strumieni mogą (warunek - dostępne zasoby) być realizowane równocześnie.
- Identyfikator strumienia jest używany jako argument do wywołań asynchronicznych i uruchomień funkcji kernel.

Asynchroniczna komunikacja host-device

Asynchroniczne transfery host-device zwracają sterowanie natychmiast do CPU

cudaMemcpyAsync(dst, src, size, dir, stream);

- wymagają pamięci o stałej lokacji (bez wirtualizacji) (alokowanej za pomocą cudaMallocHost),
- równoczesność obliczeń CPU i przesyłu danych,
- wzrost efektywności komunikacji poprzez wykorzystanie wprost pamięci o stałej lokacji.

Przykład:

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
obliczenia_na_CPU();
cudaThreadSynchronize(); //oczekiwanie na koniec przetwarzania GPU
```

Uwaga: 0 oznacza strumień zerowy – ten sam w którym (domyślnie) uruchamiany jest kernel

Zrównoleglenie przetwarzania funkcji kernel i transferu danych HOST-GPU

Wymagana jest własność "Concurrent copy and execute" pola deviceOverlap zmiennej cudaDeviceProp.

Kernel i funkcja transferu używać muszą różnych niezerowych strumieni.

- Wywołanie funkcji CUDA dla strumienia 0 blokuje przetwarzanie tej funkcji do momentu zakończenia wszystkich wcześniejszych wywołań (wszystkich strumieni) - nie zezwala na zrównoleglenie realizacji wywołań.

Przykład:

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice,
    stream1);
kernel<<<grid, block, 0, stream2>>>(inne_niz_a_d_dane);
```

UWAGA: aby ten sam wątek mógł wywołać kernel, transfer musi być asynchroniczny.

Przykład: wieloetapowa komunikacja równoległa z przetwarzaniem GPU (i CPU)

```
for (i=0; i<nStreams; i++) {
   offset= oblicz_offset();
   cudaMemcpyAsync(a_d+offset, a_h+offset, size, dir, stream[i]);
   kernel<<<nBlocks, nThreads, 0,  stream[i]>>>(a_d+offset);
}
```

Rysunek przedstawia przebieg Sekwencyjnego i równoległego – transfer danych (CPU_GPU) i przetwarzania na GPU.

Synchronizacja przetwarzania GPU/CPU

Wywołania związane ze strumieniem i ze zdarzeniami:

zdarzenia można wprowadzić do strumienia :

cudaEventRecord(event, stream)

- zdarzenie jest zarejestrowane gdy GPU dotrze do niego (w przetwarzanym strumieniu)
- zdarzenie zarejestrowane = przypisana wartość etykiety czasowej (GPU clocktick) , użyteczne dla pomiaru czasu

- cudaEventSynchronize(event)
  - blokada wątku CPU do momentu zarejestrowania zdarzenia

- cudaEventQuery(event)
  - bada czy zdarzenie jest zarejestrowane
  - zwraca cudaSuccess, cudaErrorNotReady
  - nie blokuje przetwarzania wątku CPU

Synchronizacja GPU/CPU

- Dotyczy bieżącego kontekstu wywołania:

  cudaThreadSynchronize()

  – blokuje CPU do zakończenia wszystkich wcześniejszych wywołań CUDA z CPU

- Związane ze strumieniem:

  cudaStreamSynchronize(stream)

  – blokuje CPU do zakończenia wszystkich wywołań CUDA dla danego strumienia - parametru

  cudaStreamQuery(stream)

  – bada czy strumień jest pusty

  – zwraca cudaSuccess, cudaErrorNotReady, ...

  – nie blokuje wątku CPU

Operacje atomowe w pamięci GPU

zgodności obliczeniowej>=1.1

Operacje

add, sub, increment, decrement, min, max, ...

and, or, xor

exchange, compare, swap

1. Operacje atomowe na 32-bit słowach pamięci globalnej wymagają zgodności obliczeniowej>= 1.1 (G84/G86/ G92)

2. Operacje atomowe na 32-bit słowach pamięci współdzielonej i 64-bit słowach pamięci globalnej wymagają zgodności obliczeniowej >= 1.2

Parametry pamięci globalnej GTX 260

- Teoretyczna przepustowość (maksymalna) wg wzoru:

f (zegar pamięci) * szerokość złącza pamięci w bajtach * 2 (pamięć DDR)

1000 * 10^6 * (448 / 8) * 2 / 1024^3 = 112 GBYTE/s

- Opóźnienie w dostępie do pamięci rzędu stu cykli procesora.