Programowanie współbieżne i rozproszone

WYKŁAD 4

Jan Kazimirski

Programowanie GPU 1/2

Literatura

- **„CUDA w przykładach”, J. Sanders, E. Kandrot, 2012**
- „Computing Gems. Emerald Edition”, Wen-mei W. Hwu ed., 2011
- http://www.nvidia.pl/object/cuda_home_new_pl.html

Wstęp

- Nowy paradygmat programowania – przetwarzanie równoległe
  - Procesory wielordzeniowe
  - Specjalistyczne akceleratory do przetwarzania równoległego (np. GPU)
  - Przetwarzanie rozproszone (Grid)

GPU – trochę historii

- Lata 90-te – Pierwsze sprzętowe akceleratory 2D (systemy operacyjne z GUI)
- 1992 – Otwarcie standardu OpenGL przez firmę Silicon Graphics
- Połowa lat 90-tych – rozwój sprzętowych akceleratorów 3D (gry typu first-person shooter – Doom, Quake)
- 2001, NVIDIA GeForce 3 – początki programowej kontroli przetwarzania w GPU (poprzez OpenGL i DirectX)
- 2006 – NVIDIA wprowadza architekturę CUDA

Architektura CUDA

- Programowo sterowane ALU (wcześniej wyspecjalizowane do określonych operacji)
- Możliwość operacji na liczbach zmiennoprzecinkowych zgodnie ze standardem IEEE
- Operacje odczytu i zapisu do dowolnych komórek pamięci
- Dostęp do obszaru pamięci współdzielonej dla wszystkich jednostek wykonawczych.

CUDA C

- Dostęp do zasobów karty w dalszym ciągu wymagał użycia OpenGL lub DirectX – wyspecjalizowanych do operacji graficznych
- NVIDIA utworzyła nowy język do programowania swoich GPU w oparciu o standard języka C
- CUDA C wprowadza kilka nowych słów kluczowych specjalnie do obsługi GPU
- NVIDIA dostarczyła kompilator CUDA C i specjalny sterownik – programista nie musi korzystać z bibliotek graficznych.

CUDA - zastosowania

- Wykorzystanie do „kosztownych” obliczeń (number crunching):
  - Wzrost wydajności
  - Zmniejszenie kosztu obliczeń „per Wat”
  - Zmniejszenie kosztu obliczeń „per dolar”
- Przykłady zastosowań:
  - Przetwarzanie obrazów w medycynie
  - Symulacje dynamiki płynów
  - Symulacje procesów chemicznych biologicznych

Pierwsze kroki z CUDA

- Czego potrzebujemy?
  - Karta graficzna z obsługą CUDA
    (większość obecnych kart NVIDII)
  - Sterowniki CUDA NVIDII
  - Środowisko programistyczne CUDA
  - Kompilator C

CUDA C - programowanie

- Program CUDA C składa się z dwóch części:
  - Część działająca na CPU (***host***)
  - Część działająca na GPU (***device***)
- Funkcje działające na urządzeniu – funkcje jądra (***kernel***)
- Jądro dysponuje własną przestrzenią adresową – wymagane operacje kopiowania pamięci host<->device

Przykład 1 – ex01.cu

```
#include <stdio.h>

__global__ void kernel() {};

int main() {
  kernel<<<1,1>>>();
}
```

Przykład 1 – ex01.cu

- Funkcja kernela wygląda jak zwykła funkcja C
- Słowo kluczowe __global__ identyfikuje funkcję, która będzie kompilowana z myślą o uruchomieniu na GPU
- Wywołanie funkcji jądra zawiera dodatkowe parametry (składnia <<< ... >>>)
- Składniowo funkcje jądra i funkcje CPU wyglądają praktycznie tak samo – kompilator dba o właściwy sposób kompilacji

Przykład 1 - kompilacja

- Po instalacji środowiska (instrukcja na stronie INVIDII) dostępny jest kompilator nvcc.
- Kompilacja:

  nvcc ex01.cu

- Kompilator akceptuje wiele z opcji standardowego kompilatora gcc (-c, -o itd..)

Przykład 2 – ex02.cu

```
#include <stdio.h>

__global__ void add(int x,int y,int* z) {
  *z = x+y;
};

int main() {
  int z;
  int* dev_z;
  cudaError_t err;

  err=cudaMalloc( (void**)&dev_z, sizeof(int) );
  if(err!=cudaSuccess) return 1;

  add<<<1,1>>>( 2, 7, dev_z );

  err=cudaMemcpy(&z,dev_z,sizeof(int),cudaMemcpyDeviceToHost);
  if(err!=cudaSuccess) return 1;

  printf( "2 + 7 = %d\n", z );

  cudaFree( dev_z );

}
```

Przykład 2 – ex02.cu

- Przekazywanie parametrów do funkcji jądra działa tak samo jak dla zwykłej funkcji.
- Większość nietrywialnych funkcji jądra będzie wykorzystywała pamięć na GPU
- Typowa sekwencja działań:
  - Alokacja pamięci na urządzeniu
  - Kopiowanie danych do pamięci urządzenia
  - Uruchomienie funkcji jądra
  - Kopiowanie danych z urządzenia
  - Zwolnienie pamięci na urządzeniu

Pamięć hosta i urządzenia

- **UWAGA! Pamięć hosta i pamięć urządzenia to dwa oddzielne obszary adresowe!**
- Wskaźniki do pamięci urządzenia można przekazywać do funkcji jądra i w tych funkcjach wykorzystać do odczytu i zapisu danych
- Wskaźniki te można przechowywać w kodzie hosta
- Nie można ich jednak używać do operacji odczytu i zapisu w kodzie hosta.

cudaMaloc / cudaFree

- **cudaMalloc** – służy do rezerwacji pamięci na urządzeniu (odpowiednik malloc)

  *cudaMalloc(void** devPtr, size_t size)*

- **cudaFree** – służy do zwalniania pamięci na urządzeniu (odpowiednik free)

  cudaFree(void devPtr)*

cudaMemcpy

- **cudaMemcpy** – służy do kopiowania danych między urządzeniem i hostem (odpowiednik funkcji memcpy)

  cudaMemcpy(void dst,const void* src,size_t count, enum cudaMemcpyKind kind)*

- Ostatni parametr określa kierunek kopiowania:
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
  - cudaMemcpyHostToHost

Informacje o urządzeniu

- **cudaGetDeviceCount** – zwraca informację o liczbie urządzeń dostępnych w systemie

  cudaGetdeviceCount(int count)*

- **CudaGetDeviceProperties** – zwraca szczegółowe informacje o urządzeniu w strukturze typu cudaDeviceProp.

  cudaGetDeviceProperies(struct cudaDeviceProp prop, int device)

Przykład 3 – ex03.cu

```
#include <stdio.h>

int main() {
    int count;
    cudaGetDeviceCount( &count );
    printf("Liczba urządzeń: %d\n\n",count);
}
```

```
Liczba urządzeń: 1
```

Przykład 4 – ex04.cu

```
#include <stdio.h>

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties( &prop,0);
    printf("Nazwa urządzenia: %s\n",prop.name);
    printf("Pamięć: %ld MB\n",
            prop.totalGlobalMem/(1024*1024));
    printf("Wersja CUDA: %d.%d\n",
            prop.major,prop.minor);
    printf("Liczba procesorów: %d\n",
            prop.multiProcessorCount);
    printf( "Częstotliwość zegara: %3.1f MHZ \n",
            prop.clockRate/1000000.0);

}
```

Nazwa urządzenia: GeForce 9600 GT
Pamięć: 511 MB
Wersja CUDA: 1.1
Liczba procesorów: 8
Częstotliwość zegara: 1.5 MHZ

Suma wektorów

- Prosty przykład programowania równoległego na GPU: suma dwóch wektorów N-wymiarowych.
- Zaczniemy od implementacji na CPU, później na GPU.
- Tablice vA i vB – wektory do sumowania
- Tablica vC – wektor wynikowy
- Funkcje pomocnicze:
  - fill(int[],int val) – wypełnianie wektora wartością.
  - show(int[]) - wyświetlanie wektora
  - Rozmiar ustawiony za pomocą stałej N
- Do wyliczenia sumy służy funkcja add

Suma wektorów CPU – ex05.c
Funkcje pomocnicze i funkcja add

```
void fill(int v[],int val) {
    int i;
    for(i=0;i<N;i++) v[i]=val;
}

void show(int v[]) {
    int i;
    for(i=0;i<N;i++) printf("%d ",v[i]);
    printf("\n");
}

void add(int a[],int b[],int c[]) {
    int i;
    for(i=0;i<N;i++) c[i]=a[i]+b[i];
}
```

Suma wektorów CPU – ex05.c
Program główny

```c
#include <stdio.h>

#define N 10


int main() {
    int vA[N],vB[N],vC[N];
    fill(vA,1);
    fill(vB,2);
    add(vA,vB,vC);
    show(vC);
    return 0;
}
```

```
3 3 3 3 3 3 3 3 3 3
```

Suma wektorów GPU – ex06.c
Funkcja add i funkcja jądra

```
__global__ void ker_add(int* dev_a,int* dev_b,int* dev_c) {
    if(blockIdx.x<N)
        dev_c[blockIdx.x]=dev_a[blockIdx.x]+dev_b[blockIdx.x];
}

void add(int a[],int b[],int c[]) {
    int *dev_a,*dev_b,*dev_c;
    cudaMalloc((void**)&dev_a, N*sizeof(int));
    cudaMalloc((void**)&dev_b, N*sizeof(int));
    cudaMalloc((void**)&dev_c, N*sizeof(int));

    cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b,b,N*sizeof(int),cudaMemcpyHostToDevice);
    ker_add<<<N,1>>>(dev_a,dev_b,dev_c);
    cudaMemcpy(c,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

```
3 3 3 3 3 3 3 3 3 3
```

Suma wektorów - analiza

- Funkcja add realizuje następujące czynności:
  - Alokuje na urządzeniu pamięć na wszystkie wektory
  - Kopiuje wektory vA i vB na urządzenie
  - Wykonuje funkcję jądra
  - Kopiuje wektor vC z urządzenia
  - Zwalnia pamięć na urządzeniu
- Parametry **<<< … >>>** funkcji jądra określają sposób jej uruchomienia. Tu **<<<N,1>>>** oznacza uruchomienie N równolegle działających bloków (kopii funkcji jądra)
- Zmienna **blockIdx.x** pozwala zidentyfikować numer aktualnie wykonywanego bloku.

Bloki i wątki

- Oprócz stosowania bloków (kopii funkcji jądra), równoległość można uzyskać korzystając z wątków
- Sprzętowy limit liczby bloków – 65535
- Limit wątków w bloku – zależny od urządzenia (zwykle 512)
- Połączenie wątków i bloków pozwala przetwarzać równolegle dane o znacznie większych rozmiarach.
- Wątki mogą się bezpośrednio komunikować poprzez szybką pamięć dzieloną na GPU (bez konieczności użycia wolniejszej pamięci DRAM).

Suma wektorów - wątki

- Wykorzystanie wątków zamiast bloków w przykładzie wymaga bardzo niewielkich zmian
- Parametry środowiska <<<N,1>>> oznaczają uruchomienie N bloków po jednym wątku
- Uruchomienie jednego bloku z wieloma wątkami: <<<1,N>>>
- Wewnątrz funkcji jądra indeksowanie po blokach (**blockIdx.x**) należy zamienić na indeksowanie po wątkach (**threadIdx.x**).

Suma wektorów GPU – wątki – ex07.c

```
__global__ void ker_add(int* dev_a,int* dev_b,int* dev_c) {
    if(threadIdx.x<N)
            dev_c[threadIdx.x]= dev_a[threadIdx.x]+
                                dev_b[threadIdx.x];
}

void add(int a[],int b[],int c[]) {
    int *dev_a,*dev_b,*dev_c;
    cudaMalloc((void**)&dev_a, N*sizeof(int));
    cudaMalloc((void**)&dev_b, N*sizeof(int));
    cudaMalloc((void**)&dev_c, N*sizeof(int));

    cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b,b,N*sizeof(int),cudaMemcpyHostToDevice);
    ker_add<<<1,N>>>(dev_a,dev_b,dev_c);
    cudaMemcpy(c,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

Suma wektorów – bloki i wątki

- Dla bardzo dużych wektorów można połączyć użycie bloków i wektorów.
- Zmiana będzie wymagała określenia liczby bloków i wątków
  - Jednym ze sposobów implementacji jest określenie stałej liczby wątków i zmiennej liczby bloków.
- Zmienić należy też sposób wyliczania indeksu w funkcji jądra (musi brać pod uwagę indeks bloku, indeks wątku i rozmiar bloku – tzn. liczbę wątków na blok)

Suma wektorów GPU – bloki i wątki – ex08.c

```
__global__ void ker_add(int* dev_a,int* dev_b,int* dev_c) {
     int idx = threadIdx.x + blockIdx.x * blockDim.x;
     if(idx<N) {
            dev_c[idx]=dev_a[idx]+dev_b[idx];
     }
}

void add(int a[],int b[],int c[]) {
    int *dev_a,*dev_b,*dev_c;
    cudaMalloc((void**)&dev_a, N*sizeof(int));
    cudaMalloc((void**)&dev_b, N*sizeof(int));
    cudaMalloc((void**)&dev_c, N*sizeof(int));

    cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b,b,N*sizeof(int),cudaMemcpyHostToDevice);
    ker_add<<<(N+127)/128,128>>>(dev_a,dev_b,dev_c);
    cudaMemcpy(c,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost);

    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

Suma wektorów GPU – bloki i wątki

- Kod wyliczający indeks danych w funkcji jądra korzysta z wewnętrznych zmiennych: blockIdx.x, threadIdx.x i blockDim.x (liczba wątków w bloku)
- Zastosowana została stała liczba bloków = 128
- Liczba bloków wyliczana jest z rozmiaru danych i zaokrąglana „w górę”
- Jeżeli rozmiar danych nie jest wielokrotnością 128, to elementów liczących będzie zbyt dużo. Należy się przed tym zabezpieczyć w kodzie.

Dygresja – topologia siatki i bloku

- W przykładach stosowaliśmy jednowymiarową listę bloków i wątków. W praktyce możemy stosować bardziej złożone topologie:
- Bloki mogą być ułożone w 2-wymiarową siatkę (x,y).
- Wątki mogą być ułożone w 3-wymiarową kostkę (x,y,z).
- Wybór topologii zależy od specyfiki problemu

Dygresja – co zrobić gdy rozmiar danych jest zbyt duży

- Limity sprzętowe pozwalają przetwarzać maks 65535 bloków * limit wątków na blok (512 lub 1024).
- Gdy rozmiar danych jest większy można zastosować następujące podejście:
  - Ustawić stała liczbę bloków i wątków (np. 128/128)
  - Zmodyfikować funkcję jądra tak, aby po przetworzeniu elementu przechodziła do kolejnego przesuniętego o rozmiar „okna przetwarzania”.

Suma wektorów GPU – dowolna długość – ex09.c

```
__global__ void ker_add(int* dev_a,int* dev_b,int* dev_c) {

    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    while (idx < N) {
      dev_c[idx] = dev_a[idx] + dev_b[idx];
      idx += blockDim.x * gridDim.x;
    }

}
```

Wątki i pamięć dzielona

- Główną zaletą wątków nie jest zwiększenie liczby elementów obliczeniowych ale możliwość zastosowania pamięci dzielonej.
- Pamięć dzielona – obszar pamięci dostępny dla wszystkich wątków w bloku, a niedostępny dla wątków w innych blokach.
- Pamięć dzielona umożliwia komunikację między wątkami w bloku.
- Pamięć dzielona jest znacznie szybsza od głównej pamięci karty.
- Zmienne w pamięci dzielonej tworzy się używając słowa kluczowego __shared__

Synchronizacja wątków

- Zastosowanie pamięci dzielonej pozwala na łatwą komunikację wątków.
- Nieostrożne użycie pamięci dzielonej może prowadzić do trudnych do wykrycia błędów
- Często musimy zagwarantować, że pewne obliczenia już się skończyły zanim zaczniemy kolejne (bariera). Służy do tego funkcja __syncthreads().
- Wywołanie funkcji __syncthreads() powoduje zatrzymanie wykonywania programu każdego wątku dopóki wszystkie wątki nie dotarły do tej instrukcji.

Pamięć stała

- Duża moc obliczeniowa GPU powoduje, że często czynnikiem limitującym obliczenia jest przepustowość pamięci.
- Pamięć stała jest specjalnym obszarem pamięci której zawartość nie zmienia się w trakcie wykonania funkcji jądra
- Do rezerwacji pamięci stałej służy słowo kluczowe __constant__ .
- Pamięć stała alokowana jest statycznie (nie wymaga użycia cudaMalloc) ale wymaga określenia jej wielkości w czasie kompilacji.

Pamięć stała c.d.

- Kopiowanie danych do pamięci stałej wymaga użycia funkcji cudaMemcpyToSymbol.
- Zalety użycia pamięci stałej:
  - Buforowanie – kolejne operacje odczytu nie wywołują odwołań do pamięci
  - Jednoczesny odczyt dla kilku wątków. Pozwala oszczędzić do ponad 90% operacji odczytu.

Pamięć stała c.d.

- Ze względu na specyfikę sprzętu wątki wykonują się w grupach po 32 (osnowa) – każdy wykonuje taką samą instrukcję na innych danych
- Z punktu widzenia dostępu do pamięci stałej sprzęt może zrealizować jeden odczyt dla 16 wątków (half-warp – pół osnowy) jeżeli odwołują się do tego samego adresu.
- **UWAGA!** W przypadku odczytu przez wątki innych adresów z pamięci stałej, operacje muszą być serializowane. W efekcie wydajność może być nawet gorsza niż w przypadku pamięci głównej.

Zdarzenia

- CUDA pozwala rejestrować tzw. zdarzenia – znaczniki czasowe rejestrowane w momencie określonym przez użytkownika.
- Zdarzenia pozwalają na dokładne mierzenie wydajności kodu CUDA. Pomiar CPU nie byłby dokładny ze względu na działanie innych aplikacji.
- API CUDA do obsługi zdarzeń:
  - **cudaEvent_t** – typ reprezentujący zdarzenie
  - **cudaEventCreate** – utworzenie zdarzenia
  - **cudaEventRecord** – rejestracja momentu zdarzenia
  - **cudaEventSynchronize** – synchronizacja na zdarzeniu
  - **cudaEventelapsedTime** – czas między zdarzeniami
  - **cudaEventDestroy** – usunięcie zdarzenia

Zdarzenia c.d.

Przykład zastosowania zdarzenia:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );

    // wywołanie funkcji jądra

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );
float elapsedTime;
cudaEventElapsedTime( &elapsedTime,start, stop );
printf( "Czas:: %3.1f ms\n", elapsedTime );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Pamięć tekstur

- Pamięć tekstur jest kolejnym rodzajem pamięci stałej dostępnej dla funkcji jądra.
- Oryginalnie zaprojektowana pod kątem zastosowań graficznych (potoki renderujące)
- Jest buforowana w GPU więc dostęp do niej będzie szybszy niż do głównej pamięci karty.
- Zakłada „przestrzenną lokalność odwołań”, tzn. sąsiednie wątki odwołują się do sąsiadujących komórek pamięci.
- Przykład zastosowania:
  - Symulacja procesu rozchodzenia się ciepła.

Operacje atomowe

- Programowanie równoległe wprowadza dodatkowe komplikacje w porównaniu z programowaniem jednowątkowym.
- Przykład: operacja x++
  - Odczyt wartości x
  - Zwiększenie odczytanej wartości o 1
  - Zapisanie nowej wartości
- Powyższa operacja często określana jest jako operacja typu read-modify-write.
- Już dla dwóch wątków operacja ta może stworzyć poważny problem jeżeli oba wątki wykonają ją w tym samym czasie

Operacje atomowe c.d.

- Wątki A i B wykonują operację x++ w sposób sekwencyjny.

Wątek A	X	Wątek B
Odczytaj X (5)	5	
Zwiększ X o 1 (6)	5	
Zapisz X (6)	6	Odczytaj X (6)
	6	Zwiększ o 1 (7)
	7	Zapisz X (7)

Operacje atomowe c.d.

- Wątki A i B wykonują operację x++ w przeplocie

Wątek A	X	Wątek B
Odczytaj X (5)	5	
Zwiększ X o 1 (6)	5	Odczytaj X (5)
Zapisz X (6)	6	Zwiększ o 1 (6)
	6	Zapisz X (6)
	6	

Operacje atomowe c.d.

- W przypadku programu CUDA mamy potencjalne dziesiątki lub setki wątków, które mogą współzawodniczyć o zasób.
- W celu uniknięcia błędów należy zagwarantować wyłączność dla jednego wątku na czas całej operacji odczyt-modyfikacja-zapis
- Realizują to tzw. operacje atomowe („niepodzielne”)
- Przykłady:
  - **atomicAdd** – operacja dodania wartości do zmiennej
  - **atomicSub** – operacja odjęcia wartości od zmiennej
  - **atomicExch** – zamiana wartości w zmiennej

Operacje atomowe - histogram

- Zastosowanie operacji atomowych może mieć duży wpływ na wydajność algorytmu. Czasami wymaga wręcz zmiany zastosowanego algorytmu na inny.
- Przykład – generowanie histogramu
  Problem: Mamy duży zbiór danych tekstowych. Chcemy określić częstotliwość występowania poszczególnych znaków (liter).
- **Rozwiązanie klasyczne:** Tworzymy tablicę z 256 elementami (liczba możliwych wartości). Zerujemy ją i przeglądamy dane wejściowe zwiększając odpowiednie wartości tablicy.

Operacje atomowe – histogram c.d.

- Rozwiązanie równoległe I – **Błędne!**
  - Tworzymy tablicę histogramu
  - Poszczególne wątki przeglądają poszczególne elementy danych wejściowych i zwiększają odpowiednie wartości w tablicy.
- Problem:
  - Wiele wątków będzie jednocześnie zwiększać tą samą wartość – przeplot spowoduje błędne wyniki.

Operacje atomowe – histogram – c.d.

- Rozwiązanie równoległe II – **Poprawne, nieefektywne**
  - Do zwiększania wartości w tablicy stosujemy operację atomową (atomicAdd)
- Problem:
  - Wiele wątków będzie chciało w tym samym czasie zwiększać jedną wartość
  - Zastosowanie operacji atomowej spowoduje serializację tych operacji.
  - W efekcie algorytm praktycznie przestanie być równoległy – wątki będą czekać na swoją kolej zapisu.

Operacje atomowe – histogram c.d.

- Poprawna równoległa implementacja histogramu wymaga bardziej złożonego algorytmu. Można go zrealizować następująco
- **Etap I** – każdy blok generuje osobny histogram dla swoich danych. Jeżeli użyjemy 256 wątków to dla tablicy 256 wartości liczba kolizji znacznie się zmniejszy. Dodatkowo dla wątków bloku możemy użyć szybszej pamięci współdzielonej do trzymania wyników.
- **Etap II** – połączenie częściowych histogramów w jeden końcowy rezultat.

Blokowanie stronicowania

- Dotychczas do przekazywania danych do karty wykorzystywaliśmy bufory pamięci alokowane za pomocą malloc.
- Pamięć alokowana w ten sposób podlega stronicowaniu, tzn.:
  - Fizyczny adres danych może się zmienić
  - Dane mogą być w danym momencie na dysku (strona przeniesiona na urządzenie wymiany)
- Funkcja **cudaHostAlloc** pozwala na alokację pamięci hosta w postaci stron z wyłączonym stronicowaniem.

Blokowanie stronicowania c.d.

- Zalety wyłączenia stronicowania
  - Strony nie zostaną przeniesione na dysk
  - Adres fizyczny strony pozostaje bez zmian, karta może użyć DMA do pobrania danych.
- Ryzyko wyłączenia stronicowania
  - Utrata korzyści ze stronicowania (system nie może efektywnie wykorzystywać pamięci)
  - Zablokowanie dużej części pamięci może znacznie zmniejszyć wydajność innych aplikacji

Strumienie CUDA

- Strumień CUDA – sekwencja operacji CUDA (kopiowanie pamięci, wywołanie funkcji jądra, obsługa zdarzeń) wykonywanych w określonej kolejności
- Strumień gwarantuje, że kolejna operacja rozpocznie się dopiero po zakończeniu poprzedniej (synchronizacja).
- Jeżeli urządzenie na to pozwala („device overlap”), możemy operacje kopiowania danych i wywołania funkcji jądra wywoływać w sposób równoległy używając strumieni.
- Wykorzystując strumienie można znacznie zwiększyć wydajność aplikacji

Strumienie CUDA c.d.

- Typowa sekwencja czynności
  - Dane wejściowe dzielimy na części
  - Tworzymy dwa strumienie (cudaStreamCreate)
  - Wstawiamy do strumieni operacje kopiowania danych do urządzenia (funkcja cudaMemcpyAsync), wywołania funkcji jądra, oraz kopiowania wyników z urządzenia
  - Na koniec synchronizujemy strumienie (cudaStreamSynchronize)
- Funkcja cudaMemcpyAsync różni się od funkcji cudaMemcpy tym, że sterowanie zwracane jest do hosta po zakolejkowaniu operacji, a nie jej zakończeniu.
- Funkcja cudaMemcpyAsync wymaga użycia bufora z wyłączonym stronicowaniem.

Strumienie CUDA c.d.

- Przykładowa sekwencja operacji z użyciem dwóch strumieni

Strumień 0	Strumień 1
Kopiowanie danych do urządzenia	
Wywołanie funkcji jądra	Kopiowanie danych do urządzenia
Kopiowanie danych z urządzenia	Wywołanie funkcji jądra
	Kopiowanie danych z urządzenia
Kopiowanie danych do urządzenia	
Wywołanie funkcji jądra	Kopiowanie danych do urządzenia
Kopiowanie danych z urządzenia	Wywołanie funkcji jądra

Strumienie CUDA c.d.

- Sposób realizacji sprzętowej strumieni nie jest do końca zgodny z modelem programowym.
- Duże znaczenie dla wydajności ma kolejność umieszczania operacji w strumieniach.
- Ogólna reguła zaleca wstawianie operacji do strumieni według schematu strumień_0,strumień_1,strumień_0,... zamiast strumień_0,strumień_0,..,strumień_1, strumień_1,...
- Tworząc aplikację wykorzystującą wiele strumieni warto zainwestować czas w profilowanie kodu (z użyciem zdarzeń CUDA)

Pamięć niekopiowana

- W pewnych sytuacjach możemy jeszcze bardziej zwiększyć wydajność aplikacji stosując alokację w pamięci niekopiowanej.
- Użycie tego typu alokacji powoduje wyeliminowanie operacji kopiowania danych między pamięcią hosta i urządzenia.
- Najwyższy wzrost wydajności uzyskuje się dla urządzeń zintegrowanych.
- Urządzenia z osobną pamięcią – możliwy wzrost wydajności dla jednorazowego dostępu do danych. W przypadku dostępu wielokrotnego – znaczne pogorszenie wydajności.

Obliczenia na wielu GPU

- „Jest tylko jedna rzecz lepsza od GPU, … dwie GPU”.
- Współczesne systemy coraz częściej korzystają z kilku GPU, niektóre karty zawierają 2 lub 4 GPU na jednej karcie.
- W naszej aplikacji możemy wykonywać obliczenia na wielu urządzeniach jednocześnie.
- Istotnym ograniczeniem jest fakt, iż każde urządzenie musi być sterowane osobnym wątkiem.
- Użycie osobnych wątków stwarza problem z pamięcią zablokowaną (inne wątki nie widzą jej jako zablokowanej). W przypadku wielu wątków należy użyć specjalnego parametru alokacji – **cudaHostAllocPortable**.

Potencjał obliczeniowy CUDA

- Poszczególne karty NVIDII różnią się tzw. potencjałem obliczeniowym.
- Kolejne generacje architektury CUDA oznaczone są numerami: 1.0, 1.1, 1.2, 1.3, 2.0, 2.1, 3.0, 3.5
- Kolejne generacje są nadzbiorami poprzednich. Wprowadzają nowe rozszerzenia i funkcje.
- Przykłady:
  - Od generacji 1.1 dostępne są operacje atomowe w pamięci głównej
  - 1.2 wprowadza m.in. operacje atomowe w pamięci dzielonej
  - W wersji 1.3 wprowadzono możliwość obliczeń podwójnej precyzji

Wybrane karty NVIDII

	9600 GT	GTX 660	GTX 680	GTX 690
Liczba rdzeni	64	960	1536	3072
Taktowanie (MHZ)	650	980	1006	915
Pamięć (MB)	512	2048	2048	4096
Przepustowość pamięci (GB/sec)	57.6	144	192.2	384
Max pobór prądu (W)	96	140	195	300