Programowanie kart graficznych

Sprzęt i obliczenia

CUDA
Szczegóły implementacji sprzętowej

Architektura SIMT:

- podstawą konstrukcji urządzeń CUDA jest skalowalna macierz wielowątkowych multiprocesorów strumieniowych (ang. *streaming multiprocessors*) - SM
- w chwili odpalenia przez kod hosta wskazanej siatki kerneli, bloki rozmieszczone w siatce są szeregowane, a następnie rozdzielane pomiędzy wolne multiprocesory
- możliwe jest:
  - aby wątki pochodzące z jednego bloku wątków mogły wykonywać się równolegle w ramach jednego multiprocesora
  - wiele różnych bloków wątków mogło wykonywać się równolegle w ramach jednego multiprocesora

Architektura SIMT:

- z chwilą zakończenia wykonania pewnego wątku, w jego miejsce uruchamiany jest na zwolnionym multiprocesorze kolejny wątek
- multiprocesor architektury CUDA jest skonstruowany w sposób umożliwiający równoległe wykonanie setek wątków
- odbywa się to w ramach architektury sprzętowej nazwanej przez firmę Nvidia architekturą **SIMT**: *Single-Instruction, Multiple-Thread*
- w celu maksymalizacji wykorzystania wszystkich komponentów sprzętowych urządzenia architektura CUDA implementuje koncepcje sprzętowej wielowątkowości (ang. *hardware multithreading*)

Architektura SIMT:

- dodatkowo architektura CUDA implementuje pewne mechanizmy równoległości wewnątrz wątku takie jak np. potokowość (ang. *pipelining*), ale...
- ...wykonanie ciągu instrukcji przebiega dokładnie w kolejności narzuconej przez strukturę kodu oraz...
- ...nie stosuje się **ani** predykcji skoków, **ani** wykonania spekulatywnego

Architektura SIMT:

- każdy multiprocesor tworzy, zarządza, planuje i wykonuje kodem wątków w pęczkach po 32 wątki w każdym pęczku
- pęczek taki nosi angielską nazwę „warp” (dosłownie: osnowa)
- poszczególne wątki w pęczku są uruchamiane jednocześnie spod wspólnego, identycznego dla wszystkich wątków, adresu startowego (punktu wejścia), ale...
  - ... każdy wątek ma własny licznik rozkazów (PC – program counter)
  - ... każdy wątek ma własny rejestr statusowy

Architektura SIMT:

- oznacza to, że ścieżka wykonania każdego z wątków może przebiegać inaczej, w szczególności każdy z wątków może rozgałęziać się w inny sposób oraz kończyć swoje działanie niezależnie od pozostałych
- z chwilą, gdy blok wątków zostanie przydzielony do multiprocesora, następuje jego podział na pęczki, które podlegają zarządzaniu przez tzw. planistę pęczków (ang. *warp scheduler*)
- przydział wątków do pęczków jest sekwencyjny, tzn. w ramach pęczku znajdują się wątki o kolejnych identyfikatorach, a pierwszy pęczek zaczyna się od wątku od ID równego 0

Architektura SIMT:

- pęczek wykonuje **jedną swoją instrukcję jednocześnie**, tzn. jedna i ta sama instrukcja jest wykonywana na rzecz wszystkich lub tylko niektórych wskazanych wątków w pęczku
- oznacza to, że maksymalną prędkość wykonania pęczek osiąga wtedy (i tylko wtedy), gdy wszystkie wątki w pęczku „idą” po tej samej ścieżce wykonania
- a co się dzieje, kiedy treść wątków rozgałęzia się w różny sposób w ramach tego samego pęczku?

Architektura SIMT:

- multiprocesor podejmuje wykonanie kolejno każdej ze ścieżek z osobna, wstrzymując wykonanie tych wątków, które nie znajdują się na wybranej ścieżce
- oznacza to, że **wykonanie różnych ścieżek nie przebiega równolegle**
- z chwilą zakończenia wykonania każdej ze ścieżek indywidualnych ponownie podejmuje się wykonanie ścieżki wspólnej dla wszystkich wątków
- rozgałęzianie ścieżki zachodzi tylko w ramach pęczku – różne pęczki mogą wykonywać w pełni równolegle zupełnie różne ciągi instrukcji

Architektura SIMT:

- architektura SIMT jest bliskim krewnym wcześniejszej architektury SIMD (ang. Single-Instruction, Multiple-Data), stosowanej w procesorach wektorowych, które są zdolne do „obróbki” wielu różnych danych w ramach wykonania jednego rozkazu
- SIMD nie ma możliwości rozgałęziania ścieżki kodu, ograniczając tym samym sferę potencjalnych zastosowań
- z punktu widzenia wygody programisty możliwe jest całkowite zignorowanie przez niego działania mechanizmu rozgałęziania wewnątrz pęczków...
- ...jednak kod o szczególnych wymaganiach wydajnościowych powinien być pisany z uwzględnieniem tego efektu

Architektura SIMT:

- kontekst wątku (licznik rozkazu, rejestry arytmetyczne, etc) jest w architekturze CUDA przechowywany i obsługiwany czysto sprzętowo
- oznacza to między innymi, że przełączenie kontekstu odbywa się bez jakiegokolwiek narzutu czasu wykonania
- dzięki temu planista pęczków może w dowolnej chwili wybrać dowolny z gotowych pęczków i skierować go do wykonania
- w szczególności, każdy z multiprocesorów dysponuje zestawem 32 bitowych procesorów, które rozdziela się pomiędzy pracujące pęczki

Architektura SIMT:

- maksymalna liczba bloków i pęczków pewnego kernela, które mogą jednocześnie rezydować i wykonywać się na urządzeniu zależy od:
  - liczby rejestrów i rozmiaru pamięci używanej przez kernel
  - liczby rejestrów i rozmiaru pamięci dostępnej dla multiprocesora
- konkretne wykonanie urządzenia CUDA może wprowadzać dodatkowe ograniczenie na liczbę rezydujących wątków i pęczków na każdy dostępny multiprocesor

	Compute Capability				
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24		32		48
Maximum number of resident threads per multiprocessor	768		1024		1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K
Maximum amount of shared memory per multiprocessor	16 KB				48 KB

Źródło: *NVIDIA CUDA Programming Guide Version 3.0*

	Compute Capability				
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Number of shared memory banks	16				32
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				
Maximum width for a 1D texture reference bound to a CUDA array	8192				32768
Maximum width for a 1D texture reference bound to linear memory	2^{27}				
Maximum width and height for a 2D texture reference bound to linear memory or a CUDA array	65536 x 32768				65536 x 65536
Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048				4096 x 4096 x 4096
Maximum number of instructions per kernel	2 million				

Źródło: *NVIDIA CUDA Programming Guide Version 3.0*

Architektura SIMT:

- w przypadku, gdy liczba rejestrów multiprocesora lub rozmiar dostępnej pamięci dzielonej są mniejsze, niż konieczne dla uruchomienia przynajmniej jednego bloku wątków, to odpalenie kernela zakończy się niepowodzeniem

CUDA
Implementacja arytmetyki zmiennoprzecinkowej

Arytmetyka zmiennoprzecinkowa:

- sposób implementacji arytmetyki zmiennoprzecinkowe we współczesnych architekturach sprzętowych jest zestandaryzowany pod postacią normy IEEE-754 pochodzącej z 1985 roku (single - 32 bity, double – 64 bity)
- struktura:
  - znak liczby – 1 bit
  - cecha – 8 bitów
  - mantysa – 23 bity
- dokładność – ok 7..8 cyfr znaczących
- zakres reprezentacji: $1{,}18 \cdot 10^{-38}$ do około $3.4 \cdot 10^{38}$

Arytmetyka zmiennoprzecinkowa:

- dodatkowo definiuje się:
  - *+0* oraz *-0* → może służyć do sygnalizowania wartości bliskich zeru, ale różnych od zera
  - *+∞* oraz *-∞* → wynik dzielenia przez zero
  - *NaN* → wynik pierwiastkowania liczby ujemnej

Arytmetyka zmiennoprzecinkowa:

- arytmetyka zmiennoprzecinkowa **nie jest**:
  - przemienna:
    $a + b + c \neq c + a + b$
  - łączna:
    $(x + y) + z \neq x + (y + z)$
    $(x \cdot y) \cdot z \neq x \cdot (y \cdot z)$
  - rozdzielna
    $x \cdot (y + z) \neq x \cdot y + x \cdot z$

Arytmetyka zmiennoprzecinkowa:

- urządzenia CUDA reprezentują liczby zmiennopozycyjne zgodnie z IEEE-754 (dzięki czemu możliwe jest operowanie na wspólnej pamięci hosta i urządzenia oraz transfer danych bez konwersji pośrednich
- urządzenia CUDA implementują arytmetykę zmiennopozycyjną odmiennie niż wynika to z założeń IEEE-754
- te same operacje mogą więc dawać różne wyniki kiedy wykonywane są na sprzęcie hosta i kiedy wykonywane są na urządzeniu CUDA
- pełną zgodność gwarantuje się jedynie dla operacji dodawania i mnożenia (ale już nie dla dzielenia)

Arytmetyka zmiennoprzecinkowa:

- co to jest ULP?
- ULP - Unit in the Last Place
- wartość przypisywana najmniej znaczącemu bitowi mantysy w pewnej liczbie zmiennopozycyjnej
- można ją utożsamiać z wielkością „ziarna” w pewnym przedziale reprezentacji
- innymi słowy, dwie liczby zmiennopozycyjne, których reprezentacja różni się tylko na ostatnim bicie mantysy, są od siebie odległe o ULP
- służy do manifestowania dokładności pewnej operacji zmiennopozycyjnej

Arytmetyka zmiennoprzecinkowa:

- wg IEEE-754:

  wymaga się, aby wynik dowolnej operacji zmiennoprzecinkowej nie różnił się od rzeczywistego o więcej niż 0,5 ULP

- należy zaznaczyć, że znakomita większość bibliotek funkcji standardowych gwarantuje dokładność z przedziału 0,5 – 1 ULP
- zagwarantowanie dokładności dokładnie równej 0,5 ULP jest bardzo drogie obliczeniowo i często nieopłacalne

Arytmetyka zmiennoprzecinkowa:

- klasyczny eksperyment:

SIC!

```
#include <stdio.h>
#include <math.h>

int main(void) {
        float x;

        x = 1.0;
        while(x != x + 1.0)
                x = x * 2.0;
        printf("%f\n", x);
        printf("%f\n", log(x) / log(2.0));
        return 0;
}
```

Arytmetyka zmiennoprzecinkowa:

- i jego wynik:

```
9007199254740992.000000
53.000000
```

Pytanie:
jaka jest szacunkowa wartość ULP dla liczby zmiennoprzecinkowej o wartości zbliżonej do 2^{53}?

Arytmetyka zmiennoprzecinkowa:

- w język „C” udostępnia się funkcje:

```
#include <math.h>
double nextafter(double x, double y);
float nextafterf(float x, float y);
```

obliczają one wartość najbliższego sąsiada wartości **x** szukanego w kierunku w kierunku wartości **y**

Arytmetyka zmiennoprzecinkowa:

- spróbujmy więc:

```
#include <stdio.h>
#include <math.h>

int main(void) {
        float x = pow(2.0, 53);
        float y = pow(2.0, 54);
        float z = nextafterf(x, y);
        printf("%f\n", x);
        printf("%f\n", z);
        printf("%f", z - x);
        return 0;
}
```

Arytmetyka zmiennoprzecinkowa:

- and the winner is...

```
9007199254740992.000000
9007200328482816.000000
1073741824.000000
```

Pytanie:
jak to możliwe, że ludzkość była w stanie wysłać człowieka na Księżyc dysponując tak „precyzyjnym” aparatem obliczeniowym? ;-)

Dokładność arytmetyki pojedynczej precyzji:

`x+y`	0 (IEEE-754 round-to-nearest-even) (except for devices of compute capability 1.x when addition is merged into an FMAD)
`x*y`	0 (IEEE-754 round-to-nearest-even) (except for devices of compute capability 1.x when multiplication is merged into an FMAD)
`x/y`	0 for compute capability ≥ 2 when compiled with -prec-div=true 2 (full range), otherwise
`1/x`	0 for compute capability ≥ 2 when compiled with -prec-div=true 1 (full range), otherwise
`rsqrtf(x)` `1/sqrtf(x)`	2 (full range) Applies to `1/sqrtf(x)` only when it is converted to `rsqrtf(x)` by the compiler.
`sqrtf(x)`	0 for compute capability ≥ 2 when compiled with -prec-sqrt=true 3 (full range), otherwise
`cbrtf(x)`	1 (full range)
`rcbrtf(x)`	2 (full range)
`hypotf(x,y)`	3 (full range)
`expf(x)`	2 (full range)
`exp2f(x)`	2 (full range)
`exp10f(x)`	2 (full range)
`expm1f(x)`	1 (full range)

Źródło: *NVIDIA CUDA Programming Guide Version 3.0*

Dokładność arytmetyki pojedynczej precyzji:

`logf(x)`	1 (full range)
`log2f(x)`	3 (full range)
`log10f(x)`	3 (full range)
`log1pf(x)`	2 (full range)
`sinf(x)`	2 (full range)
`cosf(x)`	2 (full range)
`tanf(x)`	4 (full range)
`sincosf(x,sptr,cptr)`	2 (full range)
`sinpif(x)`	2 (full range)
`asinf(x)`	4 (full range)
`acosf(x)`	3 (full range)
`atanf(x)`	2 (full range)
`atan2f(y,x)`	3 (full range)
`sinhf(x)`	3 (full range)
`coshf(x)`	2 (full range)

Źródło: *NVIDIA CUDA Programming Guide Version 3.0*

Dokładność arytmetyki pojedynczej precyzji:

`tanhf(x)`	2 (full range)
`asinhf(x)`	3 (full range)
`acoshf(x)`	4 (full range)
`atanhf(x)`	3 (full range)
`powf(x,y)`	8 (full range)
`erff(x)`	3 (full range)
`erfcf(x)`	6 (full range)
`erfinvf(x)`	5 (full range)
`erfcinvf(x)`	7 (full range)
`lgammaf(x)`	6 (outside interval -10.001 ... -2.264; larger inside)
`tgammaf(x)`	11 (full range)
`fmaf(x,y,z)`	0 (full range)
`frexpf(x,exp)`	0 (full range)
`ldexpf(x,exp)`	0 (full range)
`scalbnf(x,n)`	0 (full range)
`scalblnf(x,l)`	0 (full range)

Źródło: *NVIDIA CUDA Programming Guide Version 3.0*

Dokładność arytmetyki pojedynczej precyzji:

`logbf(x)`	0 (full range)
`ilogbf(x)`	0 (full range)
`fmodf(x,y)`	0 (full range)
`remainderf(x,y)`	0 (full range)
`remquof(x,y,iptr)`	0 (full range)
`modff(x,iptr)`	0 (full range)
`fdimf(x,y)`	0 (full range)
`truncf(x)`	0 (full range)
`roundf(x)`	0 (full range)
`rintf(x)`	0 (full range)
`nearbyintf(x)`	0 (full range)
`ceilf(x)`	0 (full range)
`floorf(x)`	0 (full range)
`lrintf(x)`	0 (full range)
`lroundf(x)`	0 (full range)
`llrintf(x)`	0 (full range)
`llroundf(x)`	0 (full range)
`signbit(x)`	N/A
`isinf(x)`	N/A
`isnan(x)`	N/A
`isfinite(x)`	N/A
`copysignf(x,y)`	N/A
`fminf(x,y)`	N/A
`fmaxf(x,y)`	N/A
`fabsf(x)`	N/A
`nanf(cptr)`	N/A

Źródło: *NVIDIA CUDA Programming Guide Version 3.0*

CUDA
Odwijanie pętli

Odwijanie pętli:

- odwijanie pętli – ang. „*loop unrolling*" lub „*loop unwinding*"
- technika przyspieszania wykonania pętli poprzez zamianę kodu „zapętlonego" na kod liniowy
- poprawa wydajności odbywa się – rzecz jasna – kosztem rozmiaru kodu (odwieczny dylemat – kod może być jednocześnie albo tylko mały, albo tylko szybki – jest to tzw. *time-memory tradeoff*)
- przyspieszenie uzyskuje się dzięki pozbyciu się:
  - czasochłonnych indeksowań (często angażujących rozkazy mnożenia)
  - operacji sprawdzających osiągnięcie warunku końca pętli, zbędnych, gdy liczba przebiegów jest znana z góry

Odwijanie pętli:

- pętla zwinięta:

```
for(int i = 0; i < N; i++)
    t[i] = i;
```

- pętla odwinięta:

```
t[0] = 0;
t[1] = 1;
t[2] = 2;
t[3] = 3;
t[4] = 4;
t[5] = 5;
:
:
t[N - 1] = N - 1;
```

Odwijanie pętli:

- pętla na poły zwinięta, na poły odwinięta:

```
for(int i = 0; i < N; i+=5) {
    t[i] = i;
    t[i+1] = i+1;
    t[i+2] = i+2;
    t[i+3] = i+3;
    t[i+4] = i+4;
}
```

Odwijanie pętli:

- domyślnym zachowaniem kompilatora NVCC jest odwijanie niewielkich (?) pętli o znanej z góry liczbie wykonać
- programista ma możliwość wpływania na zachowanie kompilatora za pomocą pragmy **`unroll`**

```
#pragma unroll N
```

- pragma ta musi być umieszczona bezpośrednio przed pętlą **`for`** i dotyczy tylko i wyłącznie tej jednej pętli

Odwijanie pętli:

- na przykład:

```
#pragma unroll 5
for (int i = 0; i < n; ++i) {
    :
    :
}
```

- kompilator wygeneruje odwinięcie ciała pętli dla pięciu iteracji
- kompilator wygeneruje również kod sprawdzający, czy n rzeczywiście jest równe 5
- jeśli n < 5, to zostanie wykonany kod odwinięty – w odpowiednim miejscu jego wykonanie zostanie przerwane
- jeśli n > 5, to zostanie wykonany kod zwinięty

Odwijanie pętli:

- zadaniem programisty jest ustalenie, czy odwinięcie pętli rzeczywiście daje realne korzyści
- użycie pragmy postaci:

  `#pragma unroll 1`

  wyłącza odwijanie pętli

- użycie pragmy postaci:

  `#pragma unroll`

  włacza bezwarunkowe odwijanie pętli, o ile tylko jej górna granica jest znana w czasie kompilacji