Programowanie kart graficznych

Podstawy programowania w języku PTX

PTX
Parallel Thread eXecution

Literatura:

„PTX: Parallel Thread Execution ISA” v2.3,
NVIDIA Corp, 2011

PTX - składnia

- PTX jest wrażliwy na rejestr liter (ang. *case sensitive*)
- wszystkie słowa kluczowe pisze się małymi literami
- każdy program musi zaczynać się dyrektywami `.version` i `.target` (w tej kolejności)
- komentarze w PTX zapisuje się używając konwencji znanych z C/C++ czyli wykorzystują dwuznak `//` dla komentarzy jednowierszowych i par dwuznaków `/*` i `*/` dla komentarzy wielowierszowych
- dowolna liczba następujących po sobie białych znaków traktowana jest jak pojedyncza spacja
- każdy komentarz traktowany jest jak pojedyncza spacja

PTX - składnia

- każdy wiersz programu w języku PTX jest (albo-albo):
  - dyrektywą
  - instrukcją
- dyrektywa wpływa na strukturę programu bądź działanie kompilatora
- instrukcja jest przekładana na rozkaz (rozkazy) kodu maszynowego
- każdy wiersz programu w języku PTX może być poprzedzony tzw. etykietą (ang. label)
- etykietą zawsze kończy się znakiem : (dwukropek)

PTX - składnia

dyrektywy
ich nazwy zaczynają się od znaku .
(kropka)

```
        .reg      .b32      r1,r2;
        .global   .f32      array[32];

start:  mov.b32             r1,%tid.x;
        shl.b32             r1,r1,2;        // x 4
        ld.global.b32       r2,array[r1];
        add.f32             r2, r2, 0.5;
```

instrukcje
każda z nich odpowiada rozkazowi
maszynowemu

PTX - składnia

komentarz
w tym przypadku jednowierszowy

```
        .reg     .u32     r1,r2;
        .global  .f32     array[32];

start:  mov.b32           r1,%tid.x;
        shl.b32           r1,r1,2;        // x 4
        ld.global.b32  r2,array[r1];
        add.f32           r2, r2, 0.5;
```

etykieta
identyfikuje pewną dyrektywę lub instrukcję, umożliwiając powołanie się na nią w inny miejscu programu

PTX - składnia

```
.reg   .u32   r1,r2;
```

dyrektywa .reg

deklaracja rejestru (ang. *register*)

pełni rolę polecenia zadeklarowania zmiennej lokalnej kodu kernela; liczba rejestrów, które możesz zadeklarować, wynika z architektury konkretnego urządzenia, ale jest to liczba na tyle duża, że w przypadku prostych programów możesz zapomnieć o tym ograniczeniu

PTX - składnia

```
.reg   .u32   r1,r2;
```

typ deklarowanego rejestru

u – całkowity bez znaku
32 – o długości 32 bitów

jest to synonim typu `unsigned` z języka C
UWAGA! PTX nie przeprowadza nawet szczątkowej kontroli typów, a na pewno nie takiej, do której przyzwyczaiły nas języki wysokiego poziomu; rejestry zadeklarowane w ten sposób mogą także przechowywać dane zmiennoprzecinkowe i operować na nich; bezwzględnie istotne jest jedynie określenie długości rejestru (w bitach)

PTX - składnia

```
.reg    .u32      r1,r2;
```

nazwa lub nazwy deklarowanych rejestrów

całą linię można uznać za dokładny odpowiednik deklaracji o postaci:

```
unsigned r1,r2;
```

PTX - składnia

```
.global  .f32  array[32];
```

deklaracja danej globalnej

dana taka jest dostępna/wspólna dla wszystkich kerneli

PTX - składnia

```
.global  .f32  array[32];
```

deklaracja typu danej globalnej

f – dana zmiennoprzecinkowa
32 – o długości 32 bitów

jest to synonim typu `float` z języka C

PTX - składnia

```
start: mov.u32    r1,%tid.x;
```

etykieta

do opatrzonej nią instrukcji można odwołać się w innym miejscu programu; najczęściej robi się to przy okazji wykorzystania instrukcji skoku, czyli odpowiednika zapomnianej instrukcji `goto`

PTX - składnia

```
start: mov.b32    r1,%tid.x;
```

instrukcja

instrukcja mov (od ang. *move*) służy do przeniesienia (dokładniej – skopiowania) pewnej wartości **pomiędzy rejestrami**

PTX - składnia

```
start:    mov.b32     r1,%tid.x;
```

typ przesyłanych danych

przesłaniu podlega 32 bitowa dana całkowita bez znaku

PTX - składnia

```
start:    mov.b32     r1,%tid.x;
```

rejestr docelowy

dana zostanie skopiowana **do** wskazanego rejestru

PTX - składnia

```
start:    mov.b32     r1,%tid.x;
```

rejestr źródłowy

dana zostanie skopiowana **ze** wskazanego rejestru;
predefiniowany rejestr **%tid** jest odpowiednikiem struktury ThreadIdx dostępnej na poziomie języka CUDA C

cała instrukcja odpowiada logicznie poniższej konstrukcji języka C:

```
r1 = threadIdx.x;
```

PTX - składnia

```
shl.u32          r1,r1,2;
```

instrukcja

instrukcja shl (od ang. *shift left*) służy do przesunięcia danej w lewo o wskazaną liczbę bitów i umieszczenia wyniku we wskazanym rejestrze

PTX - składnia

```
shl.u32          r1,r1,2;
```

typ przesuwanych danych

przesuwaniu podlega 32 bitowa dana całkowita bez znaku; należy pamiętać, że chodzi tu bardziej o wskazanie sposobu przesuwania (który jest inny dla danych ze znakiem i bez) niż o faktyczny typ danej

PTX - składnia

```
shl.u32          r1,r1,2;
```

wynik i argumenty

jeśli dana instrukcja używa trzech danych, to w każdym przypadku są one ułożone w następującej kolejności:
- dana odbierająca wynik operacji
- pierwszy argument operacji
- drugi argument operacji

W powyższym przypadku instrukcja odpowiada logicznie poniższej konstrukcji języka C:

```
r1 = r1 << 2;
```

PTX - składnia

```
ld.global.b32 r2,array[r1];
```

instrukcja

instrukcja ld (od ang. *load*) służy do przeniesienia (dokładniej – skopiowania) pewnej wartości **z pamięci** (w tym przypadku globalnej) **do rejestru**

PTX - składnia

```
ld.global.b32 r2,array[r1];
```

typ pamięci biorącej udział w przesłaniu

instrukcja ld (od ang. *load*) służy do przeniesienia (dokładniej – skopiowania) pewnej wartości **z pamięci** (w tym przypadku globalnej) **do rejestru**

PTX - składnia

```
ld.global.b32 r2,array[r1];
```

typ przesyłanych danych

przesłaniu podlega dana 32 bitowa; typ danej nie jest istotny przy przesłaniach tego typu – istotna jest liczba transferowanych bitów

PTX - składnia

```
ld.global.b32 r2,array[r1];
```

rejestr docelowy

wskazanie rejestru, w którym zostanie umieszczona dana pobrana z pamięci globalnej

PTX - składnia

```
ld.global.b32 r2,array[r1];
```

źródłowa lokacja w pamięci globalnej

mimo zapisu sugerującego indeksowanie wektora, chodzi tu o wskazanie adresu odległego o `r1` bajtów od początku tablicy `array` (skojarz arytmetykę wskaźników)

PTX - składnia

```
add.f32          r2, r2, 0.5;
```

instrukcja

instrukcja **add** wykonuje dodawanie dwóch argumentów i umieszczenie wyniku we wskazanym rejestrze

PTX - składnia

```
add.f32          r2, r2, 0.5;
```

typ dodawanych danych

jest istotny dla wyboru użytej arytmetyki (stało/zmienno-przecinikowej) i długość dodawanych danych

PTX - składnia

```
add.f32          r2, r2, 0.5;
```

wynik i argumenty
w kolejności:
- dana odbierająca sumę
- pierwszy argument dodawania
- drugi argument dodawania

W powyższym przypadku instrukcja odpowiada logicznie poniższej konstrukcji języka C:

```
r2 = r2 + 0.5;
```

PTX – przykład (akademicki)
obliczenie wyróżnika trójmianu kwadratowego

```
.reg .f32 delta,a,b,c,tmp;

mul.f32   delta,b,b;         //delta=b*b;
mul.f32   tmp,a,c;           //tmp=a*c;
mul.f32   tmp,tmp,4.0;       //tmp*=4;
sub.f32   delta,delta,tmp;   //delta-=tmp;
```

PTX - identyfikatory

- `followsym:  [a-zA-Z0-9_$]`
- `identifier:  [a-zA-Z]{followsym}* | {[_$%]}{followsym}+`

Niektóre identyfikatory predefiniowane:

`%ntid`
rozmiar określonego (x,y,z) wymiaru przestrzeni wątków

`%tid`
identyfikator danego wątku we wskazanym wymiarze przestrzeni wątków (x,y,z)

Zachodzi w szczególności, że:

```
0 <= %tid.x < %ntid.x
```

PTX - predykaty

W kontekstach, w których wymagane jest określenie danych logicznych (predykatów), PTX stosuje domniemanie, że:

- wartość równa 0 reprezentuje **FALSE**
- wartość różna od 0 reprezentuje **TRUE**

PTX – przestrzenie danych

PTX wyróżnia kilka tzw. przestrzeni danych (ang. *state spaces*), odpowiadających różnych typom pamięci udostępnianym przez architekturę CUDA

Dla naszych rozważań interesujące są trzy z nich:

`.reg`
rejestry umieszczane w szybkiej pamięci lokalnej każdego wątku

`.global`
wspólny obszar pamięci dostępny dla wszystkich wątków

`.param`
obszar pamięci, w którym przekazuje się parametry do kerneli; osobny dla każdej siatki wątków

PTX – parametry kernela

- każda definicja kernela zawiera listę parametrów (być może pustą)
- parametry są danymi tylko do odczytu, a dostęp do nich możliwy jest poprzez użycie specjalnej instrukcji `ld.param`
- kolejne parametry kernela muszą być rozmieszczone w pamięci w sposób zgodny z tym, w jaki przygotowuje je kod hosta (vide funkcja `cuParamSetv`)
- należy pamiętać o tym, które z parametrów kernela są danymi, a które wskaźnikami do danych

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n, [N];
      ld.param.f64 %d, [buffer];
      :
      :
   }
```

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      reg .u32 %n;
      reg .f64 %d;
        param.u32 %n, [N];
        aram.f64 %d, [buffer];


   }
```

dyrektywa .entry

rozpoczyna kod kernela
(odpowiednik deklaracji funkcji w C)

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .r  .u32 %n;
      .r  .f64 %d;
      ld  ram.u32 %n, [N];
      ld  am.f64 %d, [buffer];
      :
      :
   }
```

identyfikator

nazwa deklarowanego kernela

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg  u32 %n;
      .reg  f64 %d;
      ld.p  m.u32 %n, [N];
      ld.p  .f64 %d, [buffer];
      :
      :
   }
```

lista parametrów kernela

ujęta w nawiasy okrągłe
(jak w nagłówku funkcji w C)

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.b32 %n, [N];
      ld.param.b64 %d, [buffer];
      :
      :
   }
```

dyrektywa .param

deklaracja pierwszego parametru kernela `foo`

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n,    ];
      ld.param.f64 %d,    ffer];
      :
      :
   }
```

typ danych pierwszego parametru

amorficzny, 32-bitowy

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n, [
      ld.param.f64 %d, [      er];
      :
      :
   }
```

identyfikator

nazwa pierwszego parametru

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n, [N];
      ld.param.f64 %d, [buf     ;
      :
      :
   }
```

dyrektywa .param

deklaracja drugiego parametru kernela `foo`

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n, [N];
      ld.param.f64 %d, [buffer]
      :
      :
   }
```

dyrektywa .align

wskazuje, że drugi z parametrów rozmieszczony jest na granicy podwójnego słowa (pod adresem podzielnym przez 8)

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n, [N];
      ld.param.f64 %d, [buffer];
      :
      :
   }
```

typ danych drugiego parametru

amorficzny, 8-bitowy

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n, [N];
      ld.param.f64 %d, [buffer];
      :
      :
   }
```

identyfikator z określeniem rozmiaru

łącznie z typem `.b8` określa daną jako ciąg 64 bajtów

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n, [N];
      ld.param.f64 %d, [buffer];
      :
      :
   }
```

deklaracja rejestru %n

stałoprzecinkowy, 32-bitowy, bez znaku

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n, [N];
      ld.param.f64 %d, [buffer];
      :
      :
   }
```

deklaracja rejestru %d

zmiennoprzecinkowy, 64-bitowy
(double)

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n, [N];
      ld.param.f64 %d, [buffer];
      :
      :
   }
```

załadowanie danej z parametru N do rejestru %n

UWAGA

zwróć uwagę na fakt ujęcia nazwy parametru w nawiasy kwadratowe; skojarz to z operacją wyłuskania spod wskaźnika znaną w C;

PTX – przykład nagłówka kernela

```
.entry foo (.param .b32 N,.param .align 8 .b8 buffer[64])
   {
      .reg .u32 %n;
      .reg .f64 %d;
      ld.param.u32 %n, [N];
      ld.param.f64 %d, [buffer];
      :
      :
   }
```

załadowanie pierwszej danej z parametru buffer do rejestru %d

UWAGA

zwróć uwagę na fakt, że dana przechowywana w `buffer` jest traktowana jak zmiennoprzecinkowa podwójnej precyzji wbrew deklaracji parametru

PTX – fundamentalne typy danych

typ fundamentalny	oznaczniki typów pochodnych
całkowity ze znakiem	`.s8, .s16, .s32, .s64`
całkowity bez znaku	`.u8, .u16, .u32, .u64`
zmiennoprzecinkowy	`.f16, .f32, .f64`
bitowy (amorficzny)	`.b8, .b16, .b32, .b64`
predykat	`.pred`

PTX – typy danych – zastrzeżenia

- dwa typy fundamentalne są zgodne, jeśli wywodzą się z tego samego typu i mają równe rozmiary
- typy ze znakiem i bez znaku są zgodne, jeśli mają takie same rozmiary
- typy amorficzne (bitowe) są zgodne z dowolnym innym typem o tym samym rozmiarze
- w zasadzie wszystkie dane mogą być deklarowane jako amorficzne pod warunkiem zachowania rozmiaru, jednak używanie precyzyjnych deklaracji typów ułatwia rozumienie kodu i umożliwia sprawdzenie poprawności wykorzystanie konkretnych instrukcji

PTX – typy danych – zastrzeżenia

- `.u8, .s8, and .b8` mogą być użyte tylko jako argumenty instrukcji `ld`, `st` i `cvt`
- typ `.f16` może być wykorzystany tylko do konwersji do/z typów `.f32` i `.f64`

PTX – deklarowanie zmiennych

- deklaracja zmiennej specyfikuje kolejno:
  - przestrzeń danych
  - typ danych
  - nazwę (nazwy) deklarowanej zmiennej
  - jeśli zmienna jest wektorem, podaje się również jego rozmiar
- zmienne predykatowe mogą być deklarowane wyłącznie jako `.reg`

PTX – arytmetyka wskaźników

- operacje na wskaźnikach mogą być traktowane jako całkowitoliczbowe bez znaku
- wskaźniki traktowane są zawsze jakby adresowały byty o rozmiarze bajta – nie ma odpowiednika dopasowywania wskaźników do rozmiarów danych znanego z języka C
- wskaźnik w PTX jest odległością lokacji danej od początku danej przestrzeni danych
- jeśli pewien rejestr zawiera wskaźnik, efekt wyłuskania uzyskuje się ujmując nazwę rejestru w nawiasy kwadratowe

PTX – konwersje danych

- konwersje danych wykonuje w PTX instrukcja **`cvt`** (ang. ***c**onv**e**rt*)
- instrukcję tę zapisuje się jako:

```
cvt.<typ_wyniku>.<typ_argumentu> wynik,argument;
```

- w szczególności poniższy zapis

```
cvt.u64.u32 long,int;
```

oznacza konwersję danej `int` typu `.u32` to danej `long` typu `.u64`

		Destination Format										
		s8	**s16**	**s32**	**s64**	**u8**	**u16**	**u32**	**u64**	**f16**	**f32**	**f64**
Source Format	**s8**	-	sext	sext	sext	-	sext	sext	sext	s2f	s2f	s2f
	s16	chop[1]	-	sext	sext	chop[1]	-	sext	sext	s2f	s2f	s2f
	s32	chop[1]	chop[1]	-	sext	chop[1]	chop[1]	-	sext	s2f	s2f	s2f
	s64	chop[1]	chop[1]	chop	-	chop[1]	chop[1]	chop	-	s2f	s2f	s2f
	u8	-	zext	zext	zext	-	zext	zext	zext	u2f	u2f	u2f
	u16	chop[1]	-	zext	zext	chop[1]	-	zext	zext	u2f	u2f	u2f
	u32	chop[1]	chop[1]	-	zext	chop[1]	chop[1]	-	zext	u2f	u2f	u2f
	u64	chop[1]	chop[1]	chop	-	chop[1]	chop[1]	chop	-	u2f	u2f	u2f
	f16	f2s	f2s	f2s	f2s	f2u	f2u	f2u	f2u	-	f2f	f2f
	f32	f2s	f2s	f2s	f2s	f2u	f2u	f2u	f2u	f2f	-	f2f
	f64	f2s	f2s	f2s	f2s	f2u	f2u	f2u	f2u	f2f	f2f	-

Notes

sext = sign extend; zext = zero-extend; chop = keep only low bits that fit;
s2f = signed-to-float; f2s = float-to-signed;
u2f = unsigned-to-float; f2u = float-to-unsigned;
f2f = float-to-float;

[1] If the destination register is wider than the destination format, the result is extended to the destination register width after chopping. The type of extension (sign or zero) is based on the destination format. For example, cvt.s16.u32 targeting a 32-bit register will first chop to 16-bits, then sign-extend to 32-bits.

za „PTX: Parallel Thread Execution ISA", NVIDIA, 2010

PTX – instrukcje

- instrukcje PTX mają od zera do czterech argumentów (operandów) plus opcjonalny prefiks predykatowy (tzw. *guard*) poprzedzany znakiem @:

```
@P opcode;
@P opcode A;
@P opcode D, A;
@P opcode D, A, B;
@P opcode D, A, B, C;
```

- w instrukcjach posługujących się więcej niż jednym argumentem, pierwszy z nich specyfikuje miejsce, w którym zostanie umieszczony wynik

PTX – wykonanie pod predykatem

- jeśli pewna instrukcja poprzedzona jest predykatem postaci:

  `@PRED`

  to zostanie wykonana tylko wtedy, gdy predykat `PRED` ma wartość `TRUE`

- jeśli pewna instrukcja poprzedzona jest predykatem postaci:

  `@!PRED`

  to zostanie wykonana tylko wtedy, gdy predykat `PRED` ma wartość `FALSE`

PTX – przykład wykonanie pod predykatem

```
/* kod w C */
if (i < n)
   j = j + 1;
```

```
        /* kod w PTX */
        .reg  .pred   cond;
        setp.lt.s32   cond, i, n;
@cond   add.s32 j, j, 1;
```

PTX – operatory porównania stałoprzecinkowego

relacja	typy ze znakiem	typy bez znaku	typy bitowe
a == b	eq	eq	eq
a != b	ne	ne	ne
a < b	lt	lo	
a <= b	le	ls	
a > b	gt	hi	
a >= b	ge	hs	

PTX – operatory porównania zmiennoprzecinkowego

jeśli którykolwiek z argumentów jest równy **NaN**, wynikiem porównania jest zawsze FALSE

relacja	operator
`a == b && !isNaN(a) && !isNaN(b)`	`eq`
`a != b && !isNaN(a) && !isNaN(b)`	`ne`
`a < b && !isNaN(a) && !isNaN(b)`	`lt`
`a <= b && !isNaN(a) && !isNaN(b)`	`le`
`a > b && !isNaN(a) && !isNaN(b)`	`gt`
`a >= b && !isNaN(a) && !isNaN(b)`	`ge`

PTX – manipulacje na predykatach

- predykaty mogą być argumentami instrukcji:
  - `and`
  - `or`
  - `xor`
  - `not`
  - `mov`
- predykat można utworzyć z danej dowolnego typu używając instrukcji `setp`
- daną całkowitą można utworzyć z predykatu używając instrukcji `selp`

```
selp.u32 var,1,0,pred; //var = pred ? 1 : 0;
```

PTX – wybrane instrukcje stałoprzecinkowe

add	suma dwóch argumentów
składnia	`add.typ  d, a, b;`
	`.typ = { .u16, .u32, .u64, .s16, .s32, .s64 };`
semantyka	`d = a + b`
uwagi	
przykład	`@p add.u32  x,y,z;`

PTX – wybrane instrukcje stałoprzecinkowe

sub	różnica dwóch argumentów

składnia	`sub.typ  d, a, b;`
	
	`.typ = { .u16, .u32, .u64, .s16, .s32, .s64 };`

semantyka	`d = a - b`
uwagi	
przykład	`@p sub.u32  a,b,c;`

PTX – wybrane instrukcje stałoprzecinkowe

mul	iloczyn dwóch argumentów

składnia	`mul{.hi,.lo,.wide}.typ  d, a, b;`
	
	`.typ = { .u16, .u32, .u64, .s16, .s32, .s64 };`

semantyka	`d = a * b`
	`n = szerokość argumentów w bitach;`
	`d = t;  // → .wide`
	`d = t<2n-1..n>; // → .hi`
	`d = t<n-1..0>;  // → .lo variant`

uwagi	

przykład	`// 16*16 → 32 bitów`
	`mul.wide.s16 fa,fxs,fys;`
	
	`// 16*16 → niższe 16 bitów`
	`mul.lo.s16 fa,fxs,fys;`
	
	`// 32*32 → 64 bity`
	`mul.wide.s32 z,x,y;`

PTX – wybrane instrukcje stałoprzecinkowe

div	iloraz dwóch argumentów

składnia	`div.typ  d, a, b;`
	
	`.typ = { .u16, .u32, .u64, .s16, .s32, .s64 };`

semantyka	`d = a / b`
uwagi	dzielenie przez zero nie powoduje przerwania wykonania programu; wynik takiej operacji jest niezdefiniowany
przykład	`@p div.u32  a,b,c;`

PTX – wybrane instrukcje stałoprzecinkowe

rem	modulo dwóch argumentów

składnia	`rem.typ  d, a, b;`
	
	`.typ = { .u16, .u32, .u64, .s16, .s32, .s64 };`

semantyka	`d = a % b`
uwagi	obliczenie reszty z dzielenia przez zero nie powoduje przerwania wykonania programu; wynik takiej operacji jest niezdefiniowany
przykład	`rem.s32  x,x,8;  // x = x%8;`

PTX – wybrane instrukcje stałoprzecinkowe

abs	wartość bezwzględna z jednego argumentu

składnia	`abs.typ d, a;`
	
	`.typ = { .s16, .s32, .s64 };`

semantyka	`d = |a|`
uwagi	
przykład	`abs.s32 r0,a;`

PTX – wybrane instrukcje stałoprzecinkowe

neg	zmiana znaku jednego argumentu

składnia	`neg.typ  d, a;`
	
	`.typ = { .s16, .s32, .s64 };`

semantyka	`d = -a`
uwagi	
przykład	`neg.s32 r0,a;`

PTX – wybrane instrukcje zmiennoprzecinkowe

add	suma dwóch argumentów (wariant podstawowy)

składnia	`add.typ  d, a, b;`
	
	`.typ = { .f32, .f64 };`

semantyka	`d = a + b`
uwagi	
przykład	`@p add.f32 f1,f2,f3;`

PTX – wybrane instrukcje zmiennoprzecinkowe

sub	różnica dwóch argumentów (wariant podstawowy)

składnia	`sub.typ d, a, b;`
	
	`.typ = { .f32, .f64 };`

semantyka	`d = a - b`
uwagi	
przykład	`sub.f32 z,z,1.0;`

PTX – wybrane instrukcje zmiennoprzecinkowe

mul	iloczyn dwóch argumentów (wariant podstawowy)

składnia	`mul.typ  d, a, b;`
	
	`.typ = { .f32, .f64 };`

semantyka	`d = a * b`
uwagi	
przykład	`mul.f32 z,z,1.0;`

PTX – wybrane instrukcje zmiennoprzecinkowe

div	iloraz dwóch argumentów (wariant podstawowy)

składnia	`div.typ  d, a, b;`
	
	`.typ = { .f32, .f64 };`

semantyka	`d = a / b`
uwagi	
przykład	`div.f32 z,z,2.0;`

PTX – wybrane instrukcje zmiennoprzecinkowe

abs	wartość bezwzględna argumentu (wariant podstawowy)

składnia	`abs.typ  d, a;`
	
	`.typ = { .f32, .f64 };`

semantyka	`d = |a|`
uwagi	
przykład	`abs.f32 z,z;`

PTX – wybrane instrukcje zmiennoprzecinkowe

sin	sinus argumentu (wariant podstawowy)
składnia	`sin.approx.f32  d, a;`
semantyka	`d = sin(a)`
uwagi	
przykład	`sin.approx.f32 sin,x;`

PTX – wybrane instrukcje zmiennoprzecinkowe

cos	cosinus argumentu (wariant podstawowy)
składnia	`cos.approx.f32  d, a;`
semantyka	`d = cos(a)`
uwagi	
przykład	`cos.approx.f32 sin,x;`

PTX – wybrane instrukcje porównania i selekcji

setp	(wariant podstawowy) porównanie dwóch danych i ustawienie predykatu zgodnie z wynikiem tego porównania

składnia	`setp.op.typ  p, a, b;`
	
	`op={eq, ne, lt, le, gt, ge, lo, ls, hi, hs}`
	
	`typ={.b16,.b32,.b64,.u16,.u32,.u64,.s16,.s32, .s64,.f32,.f64};`

semantyka	`p = a op b ? TRUE : FALSE;`
uwagi	
przykład	`setp.eq.u32  rowne,d1,d2;`

PTX – wybrane instrukcje porównania i selekcji

selp	(wariant podstawowy) wybór jednej z dwóch wartości zgodnie ze stanem predykatu

składnia	`selp.typ  d, a, b, p;`
	
	`typ={.b16,.b32,.b64,.u16,.u32,.u64,.s16,.s32,`
	`.s64,.f32,.f64};`

semantyka	`d = p ? a : b;`
uwagi	
przykład	`selp.f32  x,-1,1,rowne;`

PTX – wybrane instrukcje logiczne

and	iloczyn bitowy
składnia	`and.typ  d, a, b;`
	`typ={.pred, .b16, .b32, .b64};`
semantyka	`d = a & b;`
uwagi	
przykład	`and.b32 sign,fpvalue,0x80000000;`

PTX – wybrane instrukcje logiczne

or	suma bitowa

składnia	`or.typ  d, a, b;`
	
	`typ={.pred, .b16, .b32, .b64};`

semantyka	`d = a | b;`
uwagi	
przykład	`or.b32 sign,data,0x7FFFFFFF;`

PTX – wybrane instrukcje logiczne

xor	bitowa suma wyłączająca

składnia	`xor.typ  d, a, b;`
	
	`typ={.pred, .b16, .b32, .b64};`

semantyka	`d = a ^ b;`
uwagi	
przykład	`xor.b32 sign,data,0x7FFFFFFF;`

PTX – wybrane instrukcje logiczne

not	negacja bitowa

składnia	`not.typ  d, a;`
	
	`typ={.pred, .b16, .b32, .b64};`

semantyka	`d = ~a;`
uwagi	
przykład	`not.b32 data,0x7FFFFFFF;`

PTX – wybrane instrukcje logiczne

cnot	negacja w stylu C

składnia	`cnot.typ d, a;`
	
	`typ={.pred, .b16, .b32, .b64};`

semantyka	`d = !a;`
uwagi	
przykład	`cnot.b32 data,0x7FFFFFFF;`

PTX – wybrane instrukcje logiczne

shl	przesunięcie bitowe w lewo

składnia	`shl.typ d, a, b;`
	
	`typ={.pred, .b16, .b32, .b64};`

semantyka	`d = a << b;`
uwagi	
przykład	`shl.b32 data,data,1;`

PTX – wybrane instrukcje logiczne

shr	przesunięcie bitowe w prawo uwaga! zachowanie skrajnego lewego bitu jest zależne od typu danej!

składnia	`shr.typ  d, a, b;`
	
	`typ={.b16,.b32,.b64,.u16,.u32,.u64,.s16,.s32,.s64}`

semantyka	`d = a >> b;`
uwagi	
przykład	`shr.u32 data,data,2;`

PTX – wybrane instrukcje manipulacji danymi

mov	przemieszczenie (skopiowanie) danej

składnia	`mov.typ  d, a;`
	
	`typ={.pred,.b16,.b32,.b64,.u16,.u32,.u64,.s16,.s32,.s64}`

semantyka	`d = a;`
uwagi	
przykład	`shr.f32 licznik,0;`

PTX – wybrane instrukcje manipulacji danymi

ld	załadowanie do rejestru danej z wybranej przestrzeni danych (wariant najprostszy)

składnia	`ld.pd.typ  d, [a];`
	
	`pd={ .global, .param }`
	`typ={.pred,.b16,.b32,.b64,.u16,.u32,.u64,.s16,.s32,.s64}`

semantyka	`d = a;`
uwagi	
przykład	`ld.global.f32  var,[par1];`

PTX – wybrane instrukcje manipulacji danymi

st	zapisanie danej z rejestru do wybranej przestrzeni danych (wariant najprostszy)

składnia	`st.pd.typ [d], a;`
	
	`pd={ .global, .param }`
	`typ={.pred,.b16,.b32,.b64,.u16,.u32,.u64,.s16,.s32,.s64}`

semantyka	`d = a;`
uwagi	
przykład	`st.global.f32 [par1],reg;`

PTX – wybrane instrukcje manipulacji danymi

cvt	konwersja danej z jednego wskazanego typu na drugi

składnia	`cvt.dtyp.atyp d,a;`
	
	`dtyp,atyp={.pred,.b16,.b32,.b64,.u16,.u32,.u64,.s16,.s32,.s64}`

semantyka	`d = a;`
uwagi	
przykład	`cvt.f32.s32 f,i;`

PTX – wybrane instrukcje sterowania przepływem

bra	skok bezwzględny do wskazanej instrukcji (wariant podstawowy)
składnia	`bra  d;`
semantyka	`goto d;`
uwagi	

przykład	`bra koniec;`
	`:`
	`:`
	`koniec: mov wynik,0;`

PTX – wybrane instrukcje sterowania przepływem

exit	zakończenie pracy wątku
składnia	`exit;`
semantyka	`return;`
uwagi	
przykład	`exit;`

PTX – przykład użycia

- skonstruujemy przykład w którym kod kernela zapisany w CUDA C mógłby wyglądać następująco:

```
__global__ void KERN(int N, float *vect) {
   int x = threadIdx.x;
   if(x < N) {
      float f = vect[x];
      f = 2. * f * f + 4. * f + 1;
      vect[x] = f;
   }
}
```

Kod kernela umieszczony w pliku kernel.ptx

```
        .version          2.1
        .target           sm_10
        .entry   KERN ( .param .u32 N,    .param .align 8 .u64 vect)
        {
        .reg     .u32     max,indx;
        .reg    .u64      ptr,indx2;
        .reg     .pred    out;
        .reg     .f32     x, val, tmp;

        ld.param.u32      max, [N];      // max = N
        ld.param.u64      ptr, [vect];   // ptr = vect;
        mov.b32           indx, %tid.x;  // indx = threadIdx.x
        setp.ge.u32       out,indx,max;  // if(indx > max)
@out    bra               END;           //  goto END:
        shl.b32           indx,indx,2;   // indx *= 4;
        cvt.u64.u32       indx2,indx;    // indx2 = (long)int
        add.u64           ptr,ptr,indx2; // ptr += indx2;
        mov.f32           val,1.0;       // val = 1.0
        ld.global.f32     x,[ptr];       // x = *ptr
        mul.f32           tmp,x,4.0;     // tmp = 4 * x;
        add.f32           val,val,tmp;   // val += tmp
        mul.f32           x,x,x;         // x = x * x
        mul.f32           tmp,x,2.0;     // tmp = 2 * x
        add.f32           val, val, tmp; // val += tmp
        st.global.f32     [ptr],val;     // *ptr = val
END:    exit;                            // return
        }
```

Kompilacja pliku kernel.ptx do pliku kernel.cubin

```
$ nvcc -arch=sm_30 -cubin kernel.ptx
```

Kod cpu umieszczony w pliku main.c (1/3)

```
#include        <stdio.h>
#include        <assert.h>
#include        <cuda.h>

#define N 50
#define BLK_SZ 128
#define CALL(x) {int r=x;\
                if(r!=0){fprintf(stderr,"%s returned %d in line %d"\
                " -- exiting.\n",#x,r,__LINE__);\
                exit(0);}}

#define ALIGN_UP(off,align) (off)=((off)+(align)-1)& ~((align)-1)

float    HostVect[N];

float fun(float x) {
        return 2 * x * x + 4 * x + 1;
}
```

Kod cpu umieszczony w pliku main.c (2/3)

```
int main(int argc, char *argv[]) {
        int i;
        float x;
        int blocks = N / BLK_SZ;
        if(N % BLK_SZ) blocks++;
        for(i = 0; i < N; i++) HostVect[i] = (float)i;
        CUdevice        hDevice;
        CUcontext       hContext;
        CUmodule        hModule;
        CUfunction      hFunction;
        CALL( cuInit(0) );
        CALL( cuDeviceGet(&hDevice, 0) );
        CALL( cuCtxCreate(&hContext, 0, hDevice) );
        CALL( cuModuleLoad(&hModule, "kernel.cubin") );
        CALL( cuModuleGetFunction(&hFunction, hModule, "KERN") );
        CUdeviceptr DevVect;
        CALL( cuMemAlloc(&DevVect, sizeof(HostVect)) );
        CALL( cuMemcpyHtoD(DevVect, HostVect, sizeof(HostVect)) );
        CALL( cuFuncSetBlockShape(hFunction, BLK_SZ, 1, 1) );

        int     offset = 0;
        void   *ptr;
```

Kod cpu umieszczony w pliku main.c (3/3)

```
        ptr = (void*)(size_t)N;
        ALIGN_UP(offset, __alignof(ptr));
        CALL( cuParamSetv(hFunction, offset, &ptr, sizeof(ptr)) );
        offset += sizeof(ptr);

        ptr = (void*)(size_t)DevVect;
        ALIGN_UP(offset, __alignof(ptr));
        CALL( cuParamSetv(hFunction, offset, &ptr, sizeof(ptr)) );
        offset += sizeof(ptr);

        CALL( cuParamSetSize(hFunction, offset) );
        CALL( cuLaunchGrid(hFunction, blocks, 1) );
        CALL( cuMemcpyDtoH((void*) HostVect, DevVect, sizeof(HostVect)) );
        CALL( cuMemFree(DevVect) );
        for(i = 0; i < N; i++)
               printf("arg: %f  gpu: %f  cpu: %f\n", (float)i,
              HostVect[i], fun((float)i));
        puts("done");
        return 0;
}
```

Kompilacja pliku main.c do pliku wykonywalnego main i wykonanie

```
$ gcc -I /usr/local/cuda/include -l cuda main.c -o main
$ ./main

arg: 0.000000  gpu: 1.000000  cpu: 1.000000
arg: 1.000000  gpu: 7.000000  cpu: 7.000000
arg: 2.000000  gpu: 17.000000  cpu: 17.000000
arg: 3.000000  gpu: 31.000000  cpu: 31.000000
arg: 4.000000  gpu: 49.000000  cpu: 49.000000
arg: 5.000000  gpu: 71.000000  cpu: 71.000000
arg: 6.000000  gpu: 97.000000  cpu: 97.000000
:
:
```