Programowanie komputerów heterogenicznych

Różności

wersja: 2024-10-18

Funkcje atomowe

operacje atomowe:

- problem: równoczesny dostęp z wielu wątków do tej samej danej, rezydującej w pamięci globalnej lub dzielonej
- problemy z synchronizacją równoczesnych operacji typu RMW (read-modify-write) → brak środków pozwalających organizować rozwiązania wg klasycznych recept np. sekcji krytycznych
- rozwiązanie → wprowadzenie rodziny funkcji wykonujących się atomowo
- funkcja taka gwarantuje, że jej wykonanie w jednym wątku będzie nieprzerywalne
- tym samym, wywołania takiej funkcji z wielu wątków nie będą się przeplatać

operacje atomowe:

- innymi słowy: funkcja atomowa jest sekcją krytyczną samą w sobie
- ograniczenia:
  - funkcje atomowe można wywoływać tylko z kodu urządzenia
  - jeśli funkcja atomowa ma operować na pamięci mapowanej, to pamięć ta musi być oznaczona jako page-locked
  - operacje atomowe mogą dotyczyć wyłącznie danych o długości 32 lub 64 bitów
  - kolejność wykonywania funkcji atomowych jest nieprzewidywalna!

Atomowe funkcje arytmetyczne

```
int atomicAdd(int* address, int val);

unsigned int atomicAdd(unsigned int* address, unsigned int val);

unsigned long long int atomicAdd
(unsigned long long int* address, unsigned long long int val);

float atomicAdd(float* address, float val);
double atomicAdd(double* address, double val);
```

- dostępne dla CC >= 2.0, a wersja **double** dla CC>=6.0
- funkcje odczytują daną spod adresu **address** i dodają do niej wartość **val**
- otrzymany wynik jest zapisywany pod adres **address**
- wynikiem funkcji jest pierwotna wartość danej spod adresu **address**

```
int atomicSub(int* address, int val);

unsigned int atomicSub(unsigned int* address, unsigned int val);
```

- funkcje odczytują daną spod adresu **address** i odejmują od niej wartość **val**
- otrzymany wynik jest zapisywany pod adres **address**
- wynikiem funkcji jest pierwotna wartość danej spod adresu **address**

```
int atomicExch(int* address, int val);

unsigned int atomicExch(unsigned int* address, unsigned int val);

unsigned long long int atomicExch
(unsigned long long int* address, unsigned long long int val);

float atomicExch(float* address, float val);
```

- funkcje wymieniają daną spod adresu **address** i daną o wartości **val**
- wynikiem funkcji jest wartość danej odczytanej spod adresu **address**

```
int atomicMin(int* address, int val);

unsigned int atomicMin(unsigned int* address,unsigned int val);

unsigned long long int atomicMin
(unsigned long long int* address, unsigned long long int val);
```

- wersja 64-bitowa dostępna dla CC >= 3.5
- funkcje porównują daną spod adresu **address** i wartość **val**
- mniejsza z nich jest zapisywana pod adres **address**
- wynikiem funkcji jest pierwotna wartość danej spod adresu **address**

```
int atomicMax(int* address, int val);

unsigned int atomicMax(unsigned int* address,unsigned int val);

unsigned long long int atomicMax
(unsigned long long int* address, unsigned long long int val);
```

- wersja 64-bitowa dostępna dla CC >= 3.5
- funkcje porównują daną spod adresu **address** i wartość **val**
- większa z nich jest zapisywana pod adres **address**
- wynikiem funkcji jest pierwotna wartość danej spod adresu **address**

```
unsigned int atomicInc(unsigned int* address, unsigned int val);
```

- funkcja odczytuje daną spod adresu **address**, podstawia pod daną tymczasową tmp i wykonuje operację:

```
((tmp ≥ val) ? 0 : (tmp + 1))
```

- wynik jest zapisywany pod adres **address**
- wynikiem funkcji jest pierwotna wartość danej spod adresu **address**

```
unsigned int atomicDec(unsigned int* address, unsigned int val);
```

- funkcja odczytuje daną spod adresu **address**, podstawia pod daną tymczasową tmp i wykonuje operację:

```
(((tmp == 0) | (tmp > val)) ? val : (tmp - 1))
```

- wynik jest zapisywany pod adres **address**
- wynikiem funkcji jest pierwotna wartość danej spod adresu **address**

```
int atomicCAS(int* address, int compare, int val);

unsigned int atomicCAS
    (unsigned int* address, unsigned int compare, unsigned int val);

unsigned long long int atomicCAS
   (unsigned long long int* address,
    unsigned long long int compare,
    unsigned long long int val);
```

- funkcja odczytuje daną spod adresu **address**, podstawia pod daną tymczasową **tmp** i wykonuje operację:

```
(tmp == compare ? val : tmp)
```

- wynik jest zapisywany pod adres **address**
- wynikiem funkcji jest pierwotna wartość danej spod adresu **address**
- CAS = Compare And Swap

Atomowe funkcje bitowe

```
int atomicAnd(int* address, int val);

unsigned int atomicAnd(unsigned int* address, unsigned int val);

unsigned long long int atomicAnd
(unsigned long long int* address, unsigned long long int val);
```

- wersja 64-bitowa dostępna dla CC >= 3.5
- funkcje odczytują daną spod adresu **address** i wykonują bitową operację & z wartością **val**
- otrzymany wynik jest zapisywany pod adres **address**
- wynikiem funkcji jest pierwotna wartość danej spod adresu **address**

```
int atomicOr(int* address, int val);

unsigned int atomicOr(unsigned int* address, unsigned int val);

unsigned long long int atomicOr
(unsigned long long int* address, unsigned long long int val);
```

- wersja 64-bitowa dostępna dla CC >= 3.5
- funkcje odczytują daną spod adresu **address** i wykonują bitową operację | z wartością **val**
- otrzymany wynik jest zapisywany pod adres **address**
- wynikiem funkcji jest pierwotna wartość danej spod adresu **address**

```
int atomicXor(int* address, int val);

unsigned int atomicXor(unsigned int* address, unsigned int val);

unsigned long long int atomicXor
(unsigned long long int* address, unsigned long long int val);
```

- wersja 64-bitowa dostępna dla CC >= 3.5
- funkcje odczytują daną spod adresu **address** i wykonują bitową operację ^ z wartością **val**
- otrzymany wynik jest zapisywany pod adres **address**
- wynikiem funkcji jest pierwotna wartość danej spod adresu **address**

Głosowanie wątków

głosowanie wątków:

- problem: należy podjąć decyzję zależną od przesłanek, które powstają równocześnie w wielu równolegle pracujących wątkach
- rozwiązanie naiwne opierałoby się (zapewne) na wykonaniu wnioskowaniu w jednym wątku i przekazanie jego wyników innym wątkom
- funkcje „głosujące” wykonują takie wnioskowanie we **wszystkich wątkach pewnego pęczku** jednocześnie
- wynik takiego głosowania jest natychmiast dostępny we wszystkich wątkach pęczku

```
int __all(int predicate);
```

- oblicza wartość **predicate** we wszystkich aktywnych wątkach pęczka i zwraca wartość różną od zera, jeśli wszystkie wątki mają **predicate** różne od zera

```
int __any(int predicate);
```

- oblicza wartość **predicate** we wszystkich aktywnych wątkach pęczka i zwraca wartość różną od zera, jeśli przynajmniej jeden wątek ma **predicate** różne od zera

```
unsigned int __ballot(int predicate);
```

- oblicza wartość **predicate** we wszystkich aktywnych wątkach pęczka i zwraca daną, która na i-tym bicie ma ustawione **1**, jeśli odpowiadający mu wątek ma **predicate** różne od zera i **0** w przeciwnym przypadku

Zegar wątku

```
clock_t clock();
long long int clock64();
```

- uwaga – nie pomyl tej funkcji z funkcją identycznej nazwie z pliku nagłówkowego `<time.h>`!
- zwraca bieżącą wartość licznika cykli zegara utrzymywanego przez każdy z multiprocesorów
- różnica dwóch wartości, zdjętych w różnych punktach wątku, obrazuje czas zużyty na efektywne wykonanie kodu
- uwaga – miara ta nie ma nic wspólnego z czasem spędzonym w wątku!

Asercje

```
void assert(int expression);
```

- uwaga – nie pomyl tej funkcji z makrem o identycznej nazwie z pliku nagłówkowego `<assert.h>`!
- zatrzymuje kernel, jeśli wartość **expression** jest równa zero
- jeśli kod kernela wykonuje się w debuggerze, funkcja może wyzwolić punkt przerwania
- każdy wątek, który obliczy **expression** równe zero, po zakończeniu pracy kernela wyprowadzi na stderr komunikat następującej postaci:

```
<filename>:<line number>:<function>: block:
[blockId.x,blockId.x,blockIdx.z], thread:
[threadIdx.x,threadIdx.y,threadIdx.z] Assertion `<expression>`
failed.
```

Wyjście formatowane

```
int printf(const char *format[, arg, ... ]);
```

- dostępne dla CC>=2.0
- uwaga – nie pomyl tej funkcji z funkcją `printf()` z pliku nagłówkowego `<stdio.h>`
- wyprowadza sformatowany łańcuch do strumienia **stdout** po stronie hosta
- wyprowadzenie znaków wykona każdy wątek → zadaniem programisty jest zapanowanie na lawiną wyprowadzanych danych
- w odróżnieniu od oryginalnej funkcji, ten `printf()` zwraca liczbę przeparsowanych argumentów np.

```
printf("%d",i)
```

zwraca 1

```
%[flags][width][.precision][size]type
```

- Flags: `# <spacja> ' 0 + -`
- Width: `* 0-9`
- Precision: `0-9`
- Size: `h l ll`
- Type: `%cdiouxXpeEfgGaAs`

Przydziały pamięci

```
void* malloc(size_t size);
void free(void* ptr);
```

- dostępne dla CC>=2.0
- uwaga! funkcje te nie mają nic wspólnego z funkcjami z pliki nagłówkowego `<stdlib.h>`
- `malloc()` → przydział pamięci o żądanym rozmiarze z symulowanej sterty ulokowanej w pamięci globalnej; wynikiem jest wskaźnik na przydzielony blok albo NULL
- otrzymany wskaźnik będzie podzielny przez 16
- `free()` → zwalnia wcześniej przydzieloną pamięć
- pamięć pozostaje przydzielona do wykonania **free** albo do chwili unicestwienia kontekstu, w którym została przydzielona → przydział może pozostać ważny nawet pomiędzy odpaleniami kerneli
- uwaga: przydziały wykonywane są w każdym wątku z osobna!

```
void* memcpy(void* dest, const void* src, size_t size);
void* memset(void* ptr, int value, size_t size);
```

- dostępne dla CC>=2.0
- uwaga! funkcje te nie mają nic wspólnego z funkcjami z pliki nagłówkowego `<stdlib.h>`

Synchronizacja wątków

```
void __syncthreads();
```

- czeka, aż wszystkie wątki w bloku osiągną punkt wywołania funkcji, dzięki czemu osiąga się gwarancję zakończenia wszystkich wcześniej zainicjowanych dostępów do pamięci globalnej i dzielonej
- pozwala uniknąć niejednoznaczności powodowanej przez zależności read-after-write, write-after-read i write-after-write
- użycie tej funkcji w kontekście warunkowym (np. w gałęzi instrukcji `if`) może spowodować nieprzewidywalne efekty uboczne

```
int __syncthreads_count(int predicate);
```

- dostępne dla CC>=2.0
- wykonuje się jak **__syncthreads** oraz dodatkowo oblicza wartość **predicate** w każdym wątku bloku
- zwraca liczbę wątków, w których **predicate** było różne od zera

```
int __syncthreads_and(int predicate);
```

- dostępne dla CC>=2.0
- wykonuje się jak **__syncthreads** oraz dodatkowo oblicza wartość **predicate** w każdym wątku bloku
- zwraca wartość różną od zera, jeśli wszystkie wątki w bloku obliczyły **predicate** różne od zera i zero w przeciwnym przypadku

```
int __syncthreads_or(int predicate);
```

- dostępne dla CC>=2.0
- wykonuje się jak **__syncthreads** oraz dodatkowo oblicza wartość **predicate** w każdym wątku bloku
- zwraca wartość różną od zera, jeśli przynajmniej jeden wątek w bloku obliczył **predicate** różne od zera i zero w przeciwnym przypadku

Bariera pamięci

bariera pamięci:

- porządek, w jakim pewien wątek urządzenia zapisuje dane do pamięci globalnej albo dzielonej albo mapowanej page-locked nie musi być taki sam, jak porządek zapisów w innym wątku urządzenia

- porządek, w jakim pewien wątek urządzenia odczytuje dane z pamięci globalnej/dzielonej/mapowanej page-locked nie musi być taki sam, jak wynikający z porządku instrukcji w programie źródłowym

```
__device__ volatile int X = 1, Y = 2;

__device__ void writeXY() {
    X = 10;
    Y = 20;
}
 __device__ void readXY() {
    int A = X;
    int B = Y;
}
```

- załóżmy, że wątek 1 wykonuje **writeXY()**, a wątek 2 wykonuje **readXY()**
- istnieje możliwość, że w wątku 2:
  - A == 10
  - B == 2
- dlaczego?

```
void __threadfence_block();
```

- gwarantuje że:
  - wszystkie zapisy do pamięci globalnej/dzielonej wykonane w wątkach tego samego bloku przed wywołaniem funkcji zostaną zaobserwowane przez pozostałe wątki jako kompletnie zakończone przed analogicznym zapisami wykonanymi po wywołaniu tej funkcji
  - wszystkie odczyty z pamięci globalnej/dzielonej wykonane w wątku odbędą się przed odczytami inicjowanymi po wywołaniu funkcji

```
void __threadfence();
```

- działa jak **__threadfence_block** oraz gwarantuje że:
  - wszystkie zapisy do pamięci globalnej/dzielonej wykonane w wątkach **innych bloków** przed wywołaniem funkcji zostaną zaobserwowane przez pozostałe wątki jako kompletnie zakończone przed analogicznym zapisami wykonanymi po wywołaniu tej funkcji

```
void __threadfence_system();
```

- działa jak **__threadfence_block** oraz gwarantuje że:
  - wszystkie zapisy do pamięci globalnej/dzielonej wykonane **we wszystkich wątkach urządzenia** przed wywołaniem funkcji zostaną zaobserwowane przez pozostałe wątki jako kompletnie zakończone przed analogicznym zapisami wykonanymi po wywołaniu tej funkcji