Nowoczesne układy mikroprocesorowe
EiT mgr 2011/2012

Prowadzący

- dr inż. Wojciech Tylman, mgr inż. Piotr Perek
- tyl@dmcs.p.lodz.pl, www.dmcs.p.lodz.pl, 26-50

Plan wykładu

- Rozszerzenia MMX, SSE, 3DNow!
- Tryb 64-bitowy
- Procesory wielordzeniowe
- Rozszerzenia wirtualizacyjne
- Technologia SpeedStep
- Inne wysokowydajne architektury mikroprocesorowe
- Procesory graficzne

MMX, SSE, 3DNow!

- Rozszerzenia zestawu instrukcji architektury x86
- Dodatkowe rejestry i instrukcje na nich operujące
- Rozwiązania typu SIMD (Single Instruction Multiple Data)

Taksonomia Flynna

- Single Instruction, Single Data stream (SISD) – jedna instrukcja, jeden strumień danych. Np. zwykły jednoprocesorowy, jednordzeniowy PC
- Single Instruction, Multiple Data streams (SIMD) – jedna instrukcja, ale wiele strumieni danych. Np. procesory graficzne
- Multiple Instruction, Single Data stream (MISD) – wiele instrukcji działa na tym samym strumieniu danych. Np. systemy redundantne
- Multiple Instruction, Multiple Data streams (MIMD). Np. systemy wieloprocesorowe, wielordzeniowe

SIMD

- Pierwsze zastosowania – superkomputery wektorowe w latach 70. XX w.; w tych zastosowaniach wyparte przez MIMD. Stosowane długości wektora: 64-64000 elementów
- Zastosowanie w PC wymuszone rozwojem grafiki (gry, przetwarzanie sygnału wideo). Pierwsze rozwiązania: Sun 1995 (VIS dla UltraSPARC I), Intel 1996 (MMX dla Pentium). Długości wektora: 2-16 elementów

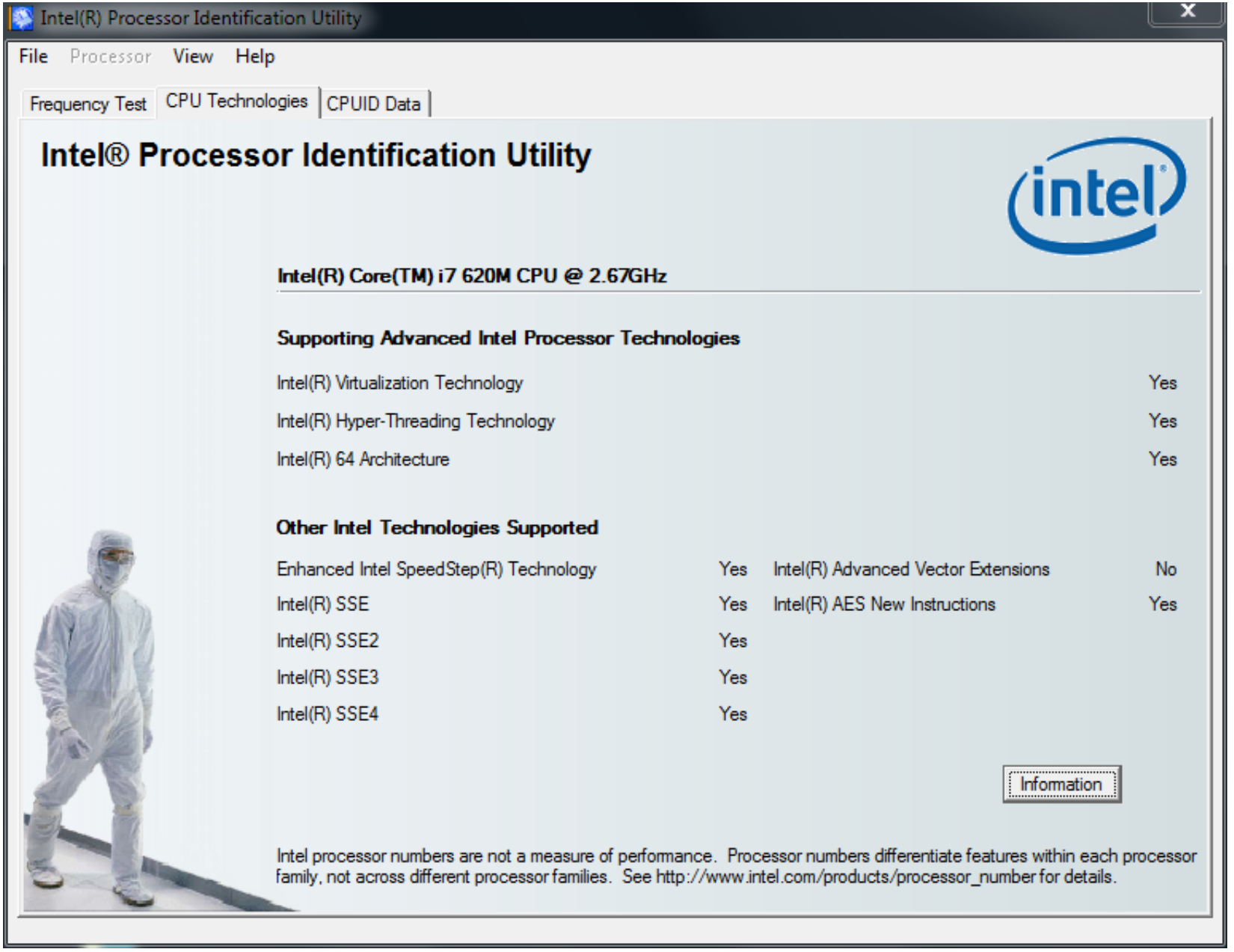
Zalety/wykorzystanie

- Ta sama operacja na wielu danych, np. przesunięcie punktów figury o tę samą odległość
  - Załadowanie wielu danych jednocześnie
  - Przeprowadzenie operacji na wielu danych jednocześnie

Wady

- Nie wszystkie zadania da się zwektoryzować
- Problemy z programowaniem:
  - Brak wsparcia w językach programowania: konieczność stosowania wstawek asemblerowych bądź specyficznych rozszerzeń oferowanych przez kompilatory
  - Trudności z automatycznym rozpoznawaniem kodu dającego się zwektoryzować
  - Zwykle konieczne wsparcie ze strony systemu operacyjnego
  - Inne problemy związane z konkretną technologią

Identyfikacja możliwości procesora - Intel Processor Identification Utility



Identyfikacja możliwości procesora - __cpuid

- void __cpuid(int CPUInfo[4], int InfoType)
  - Dla InfoType == 1 uzyskujemy:
    - int nr 3, bit nr 23: MMX
    - int nr 3, bit nr 25: SSE
    - int nr 3, bit nr 26: SSE2
    - int nr 2, bit nr 9: SSE3
    - int nr 2, bit nr 19: SSE4.1
    - int nr 2, bit nr 20: SSE4.2

MMX

- Pochodna koncepcji stosowanych w i750 (procesor graficzny) i i860 (RISC ogólnego przeznaczenia)
- 8 rejestrów (MM0-MM7). Nie są to rejestry dodatkowe, ale aliasy dla rejestrów koprocesora arytmetycznego (FPU) x87
  - Brak konieczności wsparcia ze strony systemu operacyjnego – przełączanie kontekstu zapewnione przez obsługę przełączania kontekstu koprocesora
  - Brak możliwości jednoczesnego korzystania z MMX i koprocesora (konieczność czasochłonnego przełączenia trybu)

Rejestry MMn

- Każdy rejestr MMn przechowuje 64 bity: można przetwarzać jednocześnie:
  - 1 liczbę 64 bitową albo
  - 2 liczby 32 bitowe albo
  - 4 liczby 16 bitowe albo
  - 8 liczb 8 bitowych
- Możliwe jest przetwarzanie jedynie liczb całkowitych

Programowanie MMX

- Można programować w języku asemblera
- Niektóre kompilatory mają wsparcie dla instrukcji MMX (i innych podobnych)
  - W Visual Studio i GCC stosuje się koncepcję funkcji intrinsic (wbudowanych)
- MMX dodaje 57 opcodów
- MMX dodaje typ danych _m64

__m64

- Nie należy bezpośrednio manipulować danymi w takiej zmiennej (ale można zobaczyć w debuggerze)
- Zmienne __m64 są automatycznie wyrównane na granicach 8 bajtowych
- Procesory 64-bitowe nie obsługują __m64. Należy wtedy stosować rozszerzenia SSE i SSE2

Funkcje intrinsic

- Funkcje bezpośrednio 'znane' kompilatorowi, a nie znajdujące się w bibliotekach
- Nie są wywoływane, a przetwarzane na serię instrukcji języka asemblera
  - Są szybkie, bo nie ma typowego wywoływania
  - Ponieważ kompilator wie co się dzieje we wnętrzu takiej funkcji, może jej wnętrze optymalizować
- W przypadku Visual Studio i aplikacji x64 są obowiązkowym zamiennikiem dla wstawek asemblerowych

Wybrane funkcje dla MMX

- void _mm_empty (void) `EMMS, _m_empty`
  Przywraca rejestry MMn do użycia przez koprocesor
  - O ile kolejna instrukcja dotyczy operacji zmiennoprzecinkowych, należy wywoływać zawsze po:
    - Użyciu funkcji MMX intrinsic
    - Użyciu wstawki asemblerowej z instrukcją MMX
    - Odwołaniu do typu __m64
    - Użyciu instrukcji SSE operującej na MMX

Wybrane funkcje dla MMX

- __m64 _mm_cvtsi32_si64 (int i) `MOVD, _m_from_int`
  - Konwertuje liczbę całkowitą na _m64
- int _mm_cvtsi64_si32 (__m64 m) `MOVD, _m_to_int`
  - Konwertuje młodsze 32 bity liczby __64 na liczbę całkowitą

Wybrane funkcje dla MMX

- __m64 _mm_set_pi16 (short w3, short w2, short w1, short w0)
  - Wpisuje 4 16-bitowe wartości
- __m64 _mm_set_pi32 (int i1, int i0)
- __m64 _mm_set_pi8 (char b7, char b6, char b5, char b4, char b3, char b2, char b1, char b0)

Wybrane funkcje dla MMX

- __m64 _mm_packs_pi16 (__m64 m1, __m64 m2) `PACKSSWB, _m_packsswb`
  - Upakowuje 4 16 bitowe liczby z m1 do czterech młodszych 8 bitowych liczb w rezultacie i 4 16 bitowe liczby z m2 do czterech starszych 8 bitowych liczb w rezultacie
  - Używa arytmetyki nasyceniowej

Arytmetyka nasyceniowa

- W przypadku przekroczenia dopuszczalnego zakresu wyniku uzyskujemy wartość będącą górnym (bądź dolnym, zależnie od operacji) limitem zakresu
- W tradycyjnej implementacji liczb całkowitych następuje 'przewinięcie'
- Arytmetyka nasyceniowa jest przydatna przy przetwarzaniu obrazów, np. procedura rozjaśniania

Wybrane funkcje dla MMX

- __m64 _mm_unpacklo_pi8 (__m64 m1 , __m64 m2) `PUNPCKLBW, _m_punpcklbw`
  - Przeplata 4 8 bitowe wartości z dolnej połowy m1 z 4 8 bitowymi wartościami z dolnej połowy m2. Kolejność przeplatanych wartości jest taka, że jako najmłodszy element trafia wartość z m1

Wybrane funkcje dla MMX

- __m64 _mm_add_pi8 (__m64 m1 , __m64 m2)
  `PADDB, _m_paddb`
  - Dodaje 8 8-bitowych wartości z m1 do 8 8-bitowych wartości z m2
- __m64 _mm_add_pi16 (__m64 m1 , __m64 m2)
  `PADDW, _m_paddw` – 4 16 bitowe
- __m64 _mm_add_pi32 (__m64 m1 , __m64 m2)
  `PADDD, _m_paddd` – 2 32 bitowe

Wybrane funkcje dla MMX

- __m64 _mm_adds_pi8 (__m64 m1 , __m64 m2) `PADDSB, _m_paddsb`
  - Dodaje 8 8-bitowych wartości z m1 do 8 8-bitowych wartości z m2 wykorzystując arytmetykę nasyceniową

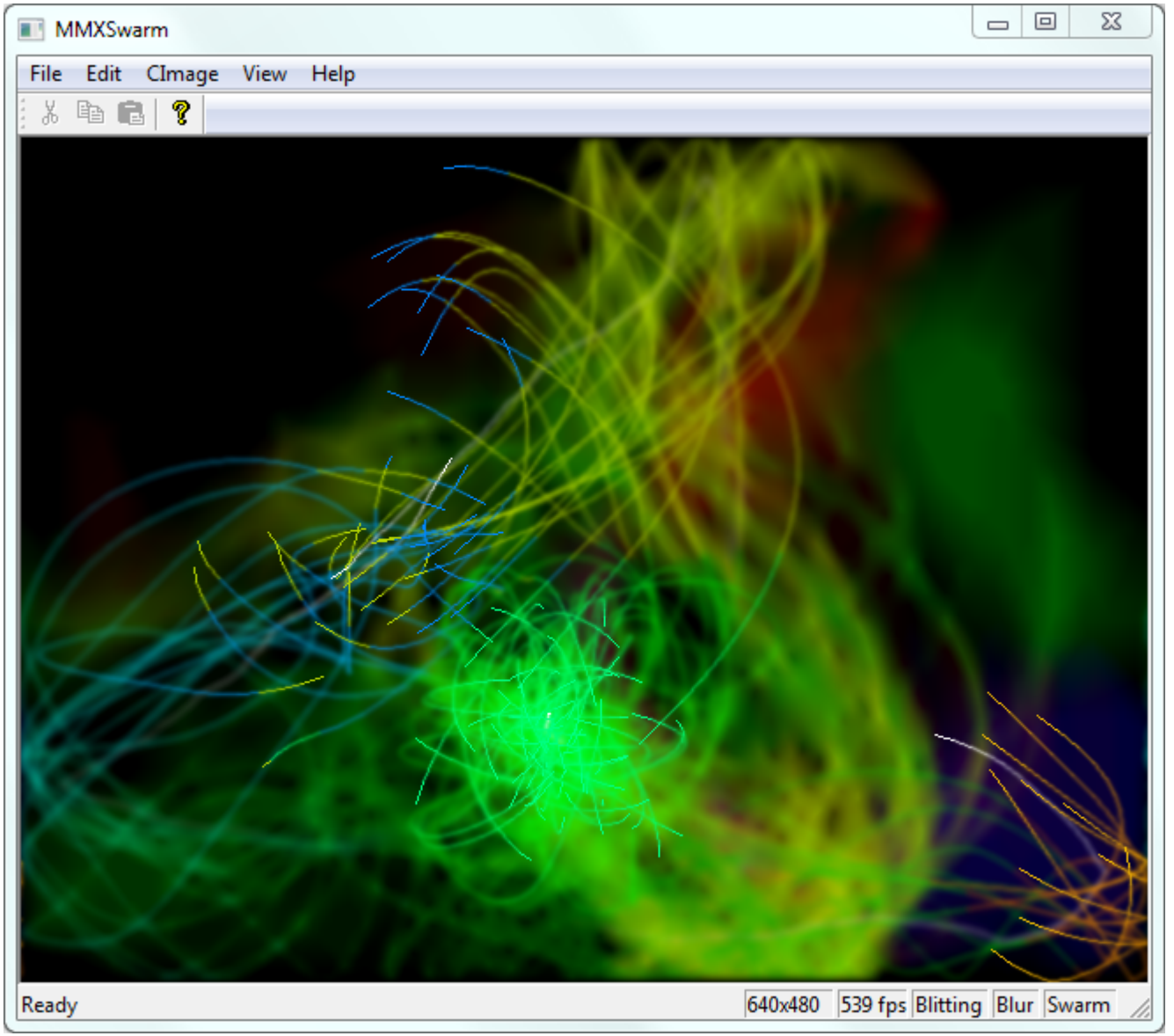
Wybrane funkcje dla MMX

- __m64 _mm_mulhi_pi16 (__m64 m1 , __m64 m2) `PMULHW, _m_pmulhw`
  - Mnoży 4 16 bitowe wartości z m1 przez 4 16 bitowe wartości z m2 i zwraca starsze 16 bitów każdego rezultatu
- __m64 _mm_mullo_pi16 (__m64 m1 , __m64 m2) `PMULLW, _pmullw`
  - j. w. , ale zwraca młodsze 16 bitów

Wybrane funkcje dla MMX

- __m64 _mm_sll_pi16 (__m64 m , __m64 count)
  `PSLLW, _m_psllw`
  - Przesuwa 4 16 bitowe wartości w m w lewo o count, uzupełniając zerami

Przykład – MMXSwarm



SSE

- Streaming SIMD Extensions
- 1999 (Pentium III), w odpowiedzi na AMD 3DNow!
- 8 nowych 128 bitowych rejestrów
- 32 bitowy format zmiennoprzecinkowy
- Wymaga wsparcia ze strony systemu operacyjnego
- Używa typów __m128, __m128i, __m128d

SSE - operacje

- Arytmetyczne: +, -, *, /, pierwiastek kwadratowy, liczba odwrotna, pierwiastek kwadratowy liczby odwrotnej, minimum, maksimum
- Logiczne: NOT, AND, OR, XOR
- Porównania
- Konwersji liczby zmiennoprzecinkowe ↔ całkowite
- Przemieszania, rozpakowania, zamiany bajtów itp.
- Odczytu i zapisu

SSE - operacje

- Obsługi cache
- Transpozycji macierzy

SSE – przykładowe funkcje

- __m128 _mm_load_ps(float * p) `MOVAPS`
  Ładuje 4 32 bitowe wartości zmiennoprzecinkowe
- void _mm_store_ps(float *p, __m128 a) `MOVAPS`
  Zapisuje 4 32 bitowe wartości zmiennoprzecinkowe
- Obie wymagają wyrównania danych na 16 bajtowych granicach

SSE – przykładowe funkcje

- __m128 _mm_add_ps(__m128 a , __m128 b) `ADDPS`
  Dodaje 4 32 bitowe liczby zmiennoprzecinkowe
- __m128 _mm_mul_ps(__m128 a , __m128 b) `MULPS`
  Mnoży 4 32 bitowe liczby zmiennoprzecinkowe
- __m128 _mm_sqrt_ps(__m128 a) `SQRTPS`
  Wyciąga pierwiastek kwadratowy z 4 32 bitowych liczb zmiennoprzecinkowe

SSE - przykład

```
float in;
int index;
int count = 100000000;
float* source = (float*)_aligned_malloc(count * sizeof(float), 16);
float* result = (float*)_aligned_malloc(count * sizeof(float), 16);
for (in = 0, index = 0; index < count; in++, index++)
{
    source[index] = (float)index;
}
__m128 sseReg;
for (int i = 0; i < 10; i++)
{
    for (index = 0; index < count; index+=4)
    {
        sseReg = _mm_load_ps(source + index);
        sseReg = _mm_sqrt_ps(sseReg);
        _mm_store_ps(result + index, sseReg);
    }
}
```

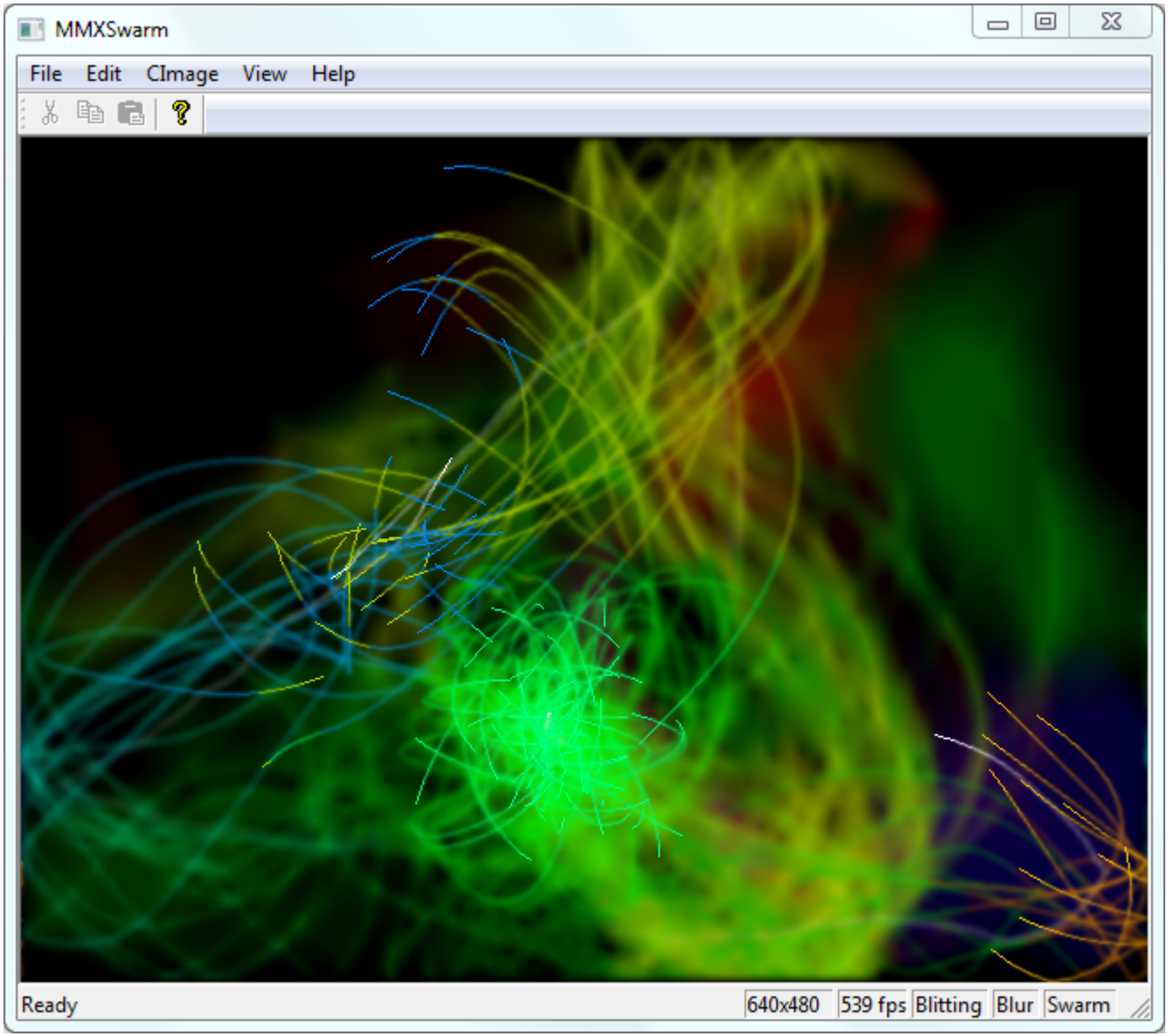
SSE2

- Wprowadzone w 2001 razem z Pentium IV
- Umożliwia pracę na liczbach podwójnej precyzji
- Zawiera operacje na liczbach całkowitych – wypiera MMX, pozwala na stosowanie operacji MMX na szerszych rejestrach

SSE2 – przykładowe funkcje

- __m128d _mm_add_pd(__m128d a, __m128d b) `ADDPD`
  Dodaje 2 64 bitowe liczby zmiennoprzecinkowe
- __m128i _mm_add_epi8 (__m128i a ,__m128i b) `PADDB`
  Dodaje 16 8 bitowych liczb całkowitych

Przykład – MMXSwarm



Tryb 64 bitowy

- 8086 (1978) – rejestry 16 bitowe, szyna adresowa 20 bitowa
- 80286 (1982) – rejestry 16 bitowe, szyna adresowa 24 bitowa
- 80386 (1985) – rejestry 32 bitowe, szyna adresowa 32 bitowa
- Pentium Pro (1995) – rejestry 32 bitowe, szyna adresowa 36 bitowa
- Athlon 64 (2003) – rejestry 64 bitowe, szyna adresowa 40 bitowa
- AMD Phenom (2007) – rejestry 64 bitowe, szyna adresowa 48 bitowa

Tryb 64 bitowy

- 64 bitowe rejestry – operacje na liczbach całkowitych, przesył danych z/do pamięci działają na 64 bitach
- Dodatkowe rejestry ogólnego przeznaczenia
- Większy adresowalny obszar pamięci: 256TB/4PB fizycznie, 256TB/16EB wirtualnie
- Wsparcie dla Position Independent Code
- Bit NX

Bit NX

- Umożliwia oznaczenie fragmentu (strony) pamięci jako zawierającej dane i nie podlegającej wykonaniu (Data Execution Prevention)
- Stosowany szeroko w innych procesorach (DEC, Sun, IBM)
- Szczególnie przydatne w zabezpieczeniach przed wykorzystaniem przepełnienia bufora
- Może stanowić problem w sytuacji np. generowania kodu podczas wykonywania

Wielowątkowość (Multithreading)

- W znaczeniu sprzętowym dotyczy wsparcia dla wykonywania wielu wątków jednocześnie na jednym rdzeniu procesora
- Stosuje się różne poziomy zaawansowania
- Dla systemu operacyjnego widoczne są osobne procesory, choć wskazana jest umiejętność odróżnienia procesorów rzeczywistych od wirtualnych
- Technologia stosowana przez Intela to Hyper-threading (od Pentium IV, typ: SMT)

Przydatność wielowątkowości

- Współczesne procesory mogą mieć istotne okresy bezczynności:
  - Brak danych w cache
  - Złe przewidzenie wyniku instrukcji warunkowej
  - Zależność między danymi
- Wielowątkowość pozwala wykorzystać ten czas do obsługi innego wątku
- Podział: czasowa (temporal), współbieżna (simultaneous, SMT)

Wielowątkowość zgrubna (blokowa)

- Jeśli jeden wątek zostanie zablokowany (np. przez brak danych w cache) uruchamiany jest następny wątek
- Wymaga duplikacji rejestrów mikroprocesora (co pozwala na przełączenie w jednym cyklu)
- Jest czasowa – jeden wątek w danej chwili w danym stopniu potoku

Wielowątkowość drobnoziarnista (z przeplotem, bębenkowa)

- W każdym cyklu wykonywana jest instrukcja z innego wątku
- Pozwala to uniknąć większości zależności danych między kolejnymi instrukcjami
- Wymaga wsparcia dla określenia przynależności do konkretnego wątku instrukcji w każdym etapie potoku
- Wiele wątków – wymaga powiększenia cache i TLB
- Czasowa

SMT

- Instrukcje z wielu wątków mogą być wykonywane równocześnie w danym stopniu potoku
- Wymaga procesora superskalarnego (np. Pentium IV – 2 szybkie ALU, 1 zwykłe)
- Wymaga odpowiedniego zarządzania – które elementy procesora są aktualnie bezczynne i mogą zostać wykorzystane, które pracują nad określonym wątkiem

Programowanie wielowątkowe

Dlaczego współbieżność to problem

- Współbieżność i przetwarzanie asynchroniczne jest typowe w rzeczywistości
- Może stanowić problem gdy wiele jednotek (ludzi, maszyn, programówm wątków) uzywa (dzieli) ten sam zsób (lub ograniczoną ilość zasobów)
- Trywialny przykład to winda – kabina to pojedynczy zasób, który jest używany przez wiele osób. Problem – jak sterować windą aby minimalizować czas oczekiwania?

Dlaczego współbieżność to problem

- Sterowanie windą nie jest krytyczne, w najgorszym wypadku czasy oczekiwania będą długie
- W niektórych przypadkach odpowiednia obsługa współbieżności jedt konieczna do poprawnej pracy systemu

Dlaczego współbieżność to problem

- Niepożądane efekty współbieżności:
  - Hazard
  - Zagłodzenie
  - Zakleszczenie

Hazard

- Przy hazardzie zachowanie systemu może być nieoczekiwane i zależy od kolejności zdarzeń
- Najpierw opisany w odniesieniu do układów logicznych, gdzie przetwarzanie równoległe jest naturalne
- Przykładem może być źle oprogramowany bankomat

Bankomat

- Przyjmijmy że działanie bankomatu jest następujące:
  - Autoryzacja klienta
  - Pobranie stanu konta z banku
  - Pobranie żądania klienta
  - Uaktualnienie konta w banku
  - Wydanie gotówki

Bankomat

- Przykładowa sytuacja:
  - Time = t_0 Autoryzacja OK
  - Time = t_0+2s Stan konta wynosi 1000
  - Time = t_0+10s Klient zażądał 800, 800<1000
  - Time = t_0+12s Konto uaktualnione o -800, 200 zostało
  - Time = t_0+14s Gotówka wydana
- Wszystko przebiegło prawidłowo

Bankomat

- Wyobraźmy sobie, że do jednego konta są wydane dwie karty (żona i mąż, konto firmowe etc.)
- Dwie osoby w niemalże tym samym czasie chcą wypłacić pieniądze. Co się stanie?

Bankomat

	Klient 1	Klient 2
Time = t_0	Autoryzacja OK	
Time = t_0+1s		Autoryzacja OK
Time = t_0+2s	Stan wynosi 1000	
Time = t_0+3s		Stan wynosi 1000
Time = t_0+5s		Klient zażądał 800, 800<1000
Time = t_0+7s		Uaktualnienie o -800, 200 zostało
Time = t_0+9s		Gotówka wydana
Time = t_0+10s	Klient zażądał 800, 800<1000	
Time = t_0+12s	Uaktualnienie o -800, **-600 zostało**	
Time = t_0+14s	Gotówka wydana	

Hazard

- Hazard może być wyeliminowany przez zablokowanie zasobów

	Klient 1	Klient 2
Time = t_0+2s	Stan wynosi 1000, zablokuj konto	
Time = t_0+3s		konto zablokowane – stop
Time = t_0+10s	Klient zażądał 800, 800<1000	
Time = t_0+12s	Konto uaktualnione o -800, 200 zostało, zablokuj konto	
Time = t_0+14s	Gotówka wydana	

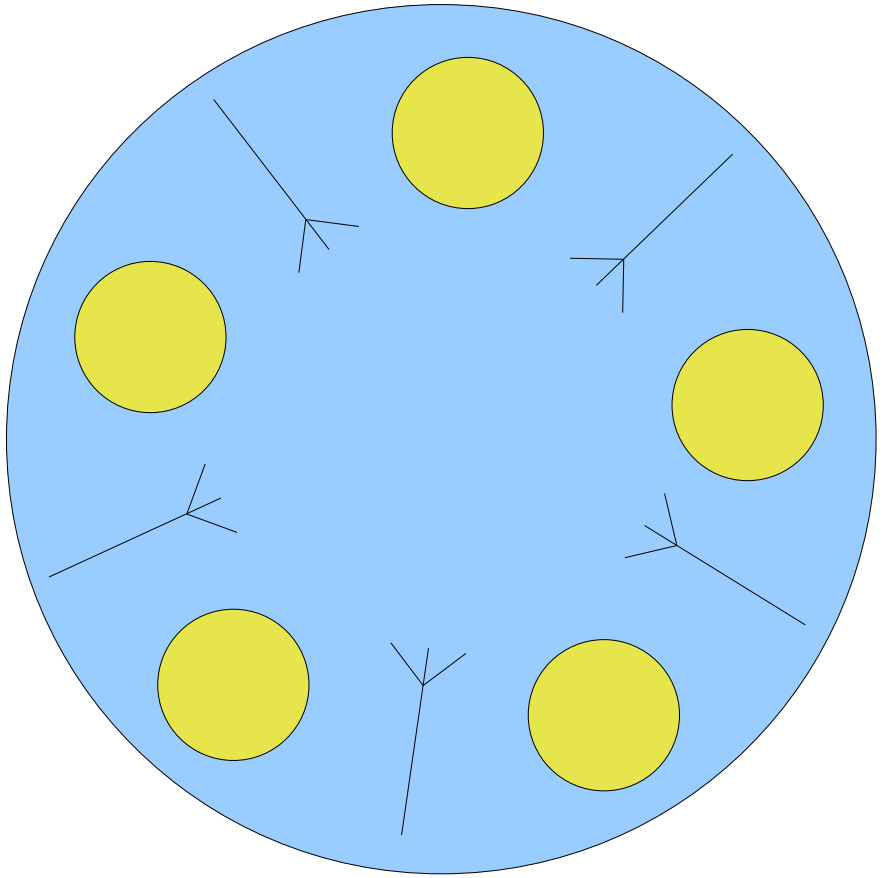
Zagłodzenie

- Proces (osoba, system) nie ma dostępu do zasobów których potrzebuje, ponieważ inny proces ich nie zwalnia

Zakleszczenie

- Występuje kliedy pewna liczba (minimum 2) procesy czekają wzajemnie na siebie aby zakończyć operację I zwolnić zasoby I z tego powodu nie mogą kontynuować
- Może być to zilustrowane problemem ucztujących filozofów (zaproponował go przez Edsger Dijkstra)

Problem ucztujących filozofów



- Filozof albo myśli, albo je
- Filozofowie nie rozmawiają (nie komunikują się)
- Filozof może podnieść widelec ze swej prawej lub lewej strony (losowo, podnosi jeden w danej chwili)
- Filozof potrzebuje obu widelcy by jeść

Architektury pamięci

- Dostępne są różne organizacje pamięci/procesorów
- Najpowszechniejszym rozwiązaniem jest wykorzystanie wspólnej pamięci (Shared memory) i jednakowa dostępność tej pamięci dla procesorów (Uniform Memory Access). Jest to znane jako Symmetric Multiprocessing (SMP)
- Watki są naturalnym sposobem implementacji przetwarzania równoległego w takim wypadku

Podział zadań

- Zasadniczym sposobem podziału zadań w systemie operacyjnym są procesy
- Procesy są 'poważnymi' tworami, wymagającymi od systemu sporej 'obsługi', mają m. in. wydzielone własne obszary pamięci
- W ramach procesów można tworzyć wątki, które korzystają z obszaru pamięci procesu
- Wątki są 'lekkim' sposobem podziału zadań, ale nadal ich zarządzaniem zajmuje się system operacyjny

Implementacja wątków

- Często ściśle związana z systemem operacyjnym (np. WinAPI)
- Isnieją rozwiązania (mniej lub bardziej) przenośne
- Większość kwestii jest rozwiązana podobnie

IEEE POSIX 1003.1c

- POSIX Threads → Pthreads
- Implementacja wątków na systemy UNIX i podobne z API dla C
- Są wersje na Win32 (pthreads-win32)
- Nagłówek pthread.h

Pthreads

- Zarządzanie wątkami: tworzenie, łączenie, kończenie wątków
- Mutexy
- Zmienne warunkowe
- Synchronizacja
- Semafory – osobna biblioteka

- Nazwy zaczynają się od pthread_

Tworzenie wątku

- int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void*), void *restrict arg);
  - thread – identyfikator wątku, zwracany przez funkcję
  - attr – atrybuty określające zachowanie wątku, NULL dla domyślnych
  - start_routine – funkcja wykonywana przez wątek
  - arg – argument przekazany do wątku
- Zwraca 0 w przypadku sukcesu

Kończenie wątku

- Wątek może się skończyć poprzez:
  - Kończy się funkcja wątku
  - Wątek wywołuje pthread_exit
  - Wątek jest zakończony przez inny wątek w wyniku wywołania pthread_cancel
  - Cały proces się kończy
  - Main kończy się bez wywołania pthread_exit

Kończenie wątku

- void pthread_exit(void *value_ptr);
  - value_ptr – opcjonalna wartość zwracana do wywołującego wątku
- int pthread_cancel(pthread_t thread);
  - Thread – wątek, który ma zostać zakończony
  - Stan i typ 'zakończalności' wątku decyduje kiedy wątek zostanie zakończony

Oczekiwanie na zakończenie wątku

- int pthread_join(pthread_t tid, void **status);
  - tid – identyfikator wątku na zakończenie którego dany wątek czeka
  - status – status zakończenia wątku
- Wartość zwracana: 0 w przypadku sukcesu, ESRCH gdy nie ma żądanego wątku, EDEADLK jeśli wywołanie funkcji spowoduje wystąpienie zakleszczenia (deadlock)

Przykłady

- http://download.oracle.com/docs/cd/E19120-01/open.solaris/816-5137/tlib-4/index.html
-
- http://randu.org/tutorials/threads/#pthreads
-
- https://computing.llnl.gov/tutorials/pthreads/#Joining

Mutexy

- Używane do synchronizacji – zabezpieczenia dostępu do fragmentów kodu, gdzie w danym momencie powinien pracować tylko jeden wątek
- int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
  - mutex – inicjalizowany mutex
  - mutexattr – atrybuty, NULL dla domyślnych
- Wartość zwracana: 0 oznacza sukces
- pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER

Mutexy

- int pthread_mutex_destroy(pthread_mutex_t *mp)
  - mp – mutex do zniszczenia
- Wartość zwracana: 0 oznacza sukces

Mutexy

- int pthread_mutex_lock(pthread_mutex_t *mutex);
- int pthread_mutex_trylock(pthread_mutex_t *mutex);
- int pthread_mutex_unlock(pthread_mutex_t *mutex);
  - mutex – mutex do zablokowania/odblokowania

Przykład

- http://randu.org/tutorials/threads/#pthreads
- https://computing.llnl.gov/tutorials/pthreads/#Mutexes

Zmienne warunkowe

- Pozwalają ustawić wątek w stan oczekiwania do momentu kiedy spełniony zostanie założony warunek
- Wymagają użycia mutexu

Zmienne warunkowe

- int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
  - cond – inicjalizowana zmienna warunkowa
  - attr – atrybuty (NULL dla domyślnych)
- pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
- int pthread_cond_destroy(pthread_cond_t *cond);

Zmienne warunkowe

- int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
- int pthread_cond_signal(pthread_cond_t *cond);

Przykład

- http://randu.org/tutorials/threads/#pthreads
- https://computing.llnl.gov/tutorials/pthreads/#Mutexes

Przykład

- http://randu.org/tutorials/threads/#pthreads
- https://computing.llnl.gov/tutorials/pthreads/#Mutexes

OpenMP

- Open Multi-Processing – API do tworzenia aplikacji przetwarzania równoległego dla C, C++ i Fortranu (77, 90, 95)
- Dostępne dla platform ze współdzielona pamięcią
- Dostępne dla wielu systemów operacyjnych (np. Linux, Unix, Mac OS X, Windows)
- Wymagane wsparcie kompilatora (np. GNU gcc, IBM, Visual Studio 2008-2010)

OpenMP

- Jest to rodzaj wielowątkowości
- Aplikacja tworzona jest z jednym wątkiem (master), który następnie tworzy (forks) grupę (team) wątków wspólnie pracujących nad zadaniem
- Grupa pracuje równolegle, po ukończeniu pracy ponownie pozostaje tyko wątek master

OpenMP

- Polecenia OpenMP mają postać dyrektyw preprocesora C/C++
- Zaczynają się od `#pragma omp`
- Następnie umieszczana jest nazwa dyrektywy i opcjonlane argumenty
- Polecenie kończy się nową linią
- Polecenie dotyczy zawsze jednego, następującego po nim bloku instrukcji

Dyrektywa parallel

- #pragma omp parallel [argumenty …]
- Przykładowe argumenty:
  - if (wyrażenie) – jeśli wyrażenie jest różne od 0, grupa wątków jest tworzona. W przeciwnym wypadku kod wykonywany jest tylko przez wątek master
  - private (lista) – lista zmiennych prywatnych dla każdego z wątków. Domyślnie prywatne są zmienne utworzone w rejonie równoległym oraz zmienne stosu w funkcjach wywoływanych z regionów równoległych
  - shared (list) – lista zmiennych współdzielonych. Domyślnie współdzielone są zmienne utworzone poza zakresem bloku równoległego

Dyrektywa parallel

- Przykładowe argumenty:
  - firstprivate (lista) – zmienne są prywatne, ale inicjalizowane bieżącą wartością współdzieloną
  - lastprivate (lista) – kopiuje ostatnią wartość zmiennej prywatnej do współdzielonej
  - reduction (operator: lista) – po zakończeniu rejonu równoległego zmienne prywatne są przy pomocy operatora umieszczane w zmiennej globalnej

Przykład

- http://msdn.microsoft.com/en-us/library/68ah4xc7.aspx

Podział pracy – dyrektywa for

- #pragma omp for [argumenty]
- Rozdziela pętlę pomiędzy wiele wątków
- Na sposób podziału można wpłynąć argumentem schedule:
  - static
  - dynamic
  - guided
  - runtime
  - auto

Podział pracy – dyrektywa sections

- #pragma omp sections [argumenty]
- Dzieli pomiędzy wątki sekcje kodu rozpoczynające się od #pragma omp section

Podział pracy – dyrektywa single

- #pragma omp single [argumenty]
- Określa, że dany fragment kodu ma być wykonany tylko przez jeden wątek z grupy
- Przydatne przy operacjach IO

Podział pracy – dyrektywy kombinowane

- Można stosować dyrektywy parallel for i parallel sections zamiast następujących po sobie dyrektyw parallel i for/sections
- #pragma omp parallel for [argumenty]
- #pragma omp parallel sections [argumenty]

Przykłady

- https://computing.llnl.gov/tutorials/openMP/#WorkSharing

Synchronizacja

- #pragma omp master – kod wykonuje tylko wątek master
- #pragma omp critical – w danej chwili kod wykonuje tylko jeden wątek. Opcjonalna nazwa pozwala na grupowanie sekcji
- #pragma omp barrier – pozwala zsynchronizować wątki grupie

Przykład

- http://msdn.microsoft.com/en-us/library/b38674ky.aspx
- http://msdn.microsoft.com/en-us/library/csa8826y.aspx

Synchronizacja zmiennych

- #pragma omp flush [(list)] – wymusza wpisanie aktualnych wartości zmiennych z listy do pamięci
- Flush jest wykonywane automatycznie dla poniższych dyrektyw (chyba, że zawierały argument nowait): barrier, parallel (we/wy), critical (we/wy), for (wy), sections (wy), single (wy)

Zmienne wątku

- Dyrektywa threadprivate pozwala określić zmienne globalne, które stają się prywatne dla wątku, a pomiędzy sekcjami równoległymi ich wartości są pamiętane I powiązne z konkretnym wątkiem

Funkcje pomocnicze

- Wymagają nagłówka omp.h
- void omp_set_num_threads(int num_threads)
- int omp_get_num_threads(void)
- int omp_get_max_threads(void)
- int omp_get_thread_num(void)
- int omp_get_num_procs(void)
- int omp_in_parallel(void)

Przykłady

- http://msdn.microsoft.com/en-us/library/xdeb73hc.aspx
- http://msdn.microsoft.com/en-us/library/62akecca.aspx
- http://msdn.microsoft.com/en-us/library/7ay38xt1.aspx
- http://msdn.microsoft.com/en-us/library/y22bxh22.aspx

Synchronizacyjne funkcje pomocnicze

- void omp_init_lock(omp_lock_t *lock)
- void omp_destroy_lock(omp_lock_t *lock)
- void omp_set_lock(omp_lock_t *lock)
- int omp_test_lock(omp_lock_t *lock)
- void omp_unset_lock(omp_lock_t *lock)

Przykład

- http://msdn.microsoft.com/en-us/library/8xybk13s.aspx
- http://msdn.microsoft.com/en-us/library/6e1yztt8.aspx

CUDA

- Compute Unified Device Architecture
- Architektura przetwarzania równoległego
- Silnik obliczeniowy układów GPU dostępny poprzez (niemalże) standardowe języki programowania
- Alternatywy: MS DirectCompute, Khronos OpenCL

Shader

- Program komputerowy / dedykowany układ obliczeniowy wykonujący taki program obliczający efekty graficzne
- Zastąpiły nieprogramowalne rozwiązania sprzętowe; oferują znacznie większą elastyczność
- W segmencie konsumenckim sprzętowe wsparcie dla programowalnych shaderów pojawiło się np. w kartach Nvidia GeForce 3 (2001: 4 pixel shadery, 1 vertex shader)

Rodzaje shaderów

- Pixel shader – przetwarza informacje dotyczące poszczególnych pikseli (kolor, współrzędną z, przezroczystość). Pozwala np. na uzyskanie faktury powierzchni, cieni, oświetlenia itp.
- Vertex shader – przetwarza informacje dotyczące poszczególnych wierzchołków w celu mapowania sceny 3D na płaszczyznę 2D (pozycja, kolor, współrzędne tekstury)
- Geometry shader – mogą generować nowe elementy grafiki (punkty, linie, trójkąty)

Unified Shading Architecture

- Nie ma odddzielnych jednostek dla konkretnych typów shaderów – wszystkie są identyczne i przydzielane dynamicznie do konkretnych zadań

Po co?

- Intel i7-3960X
- 6 rdzeni
- 130 W
- $1000
- ~130 GFLOPS

Po co?

- Nvidia GeForce GTX 590
- 32 jednostki wieloprocesorowe po 32 procesory (procesor wektorowy o 32 elementach) = 1024 jednostki przetwarzające
- 365 W
- $700
- ~2500 GFLOPS

CUDA – model programowy

- Rozszerzenie standardu C o możliwość definiowania specjalnych funkcji, tzw. jąder (kernel), wykonywanyc współbieżnie na N jednostkach przetwarzajacych
- Kernel jest definiowany słowem __global__
- Wywołanie kernela wymaga podania konfiguracji w nawiasach <<< >>>

Konfiguracja wywołania

- Obejmuje co najmniej dwie wartości:
  - Liczbę bloków w siatce
  - Liczbę wątków w bloku
- Liczba wątków określa ile wątków jednocześnie w tym samym rdzeniu procesora będzie realizaować zadanie
- Ograniczona m. in. pamięcią dostępną dla pojedynczego rdzenia
- Aktualnie maksymalnie 1024

dim3

- Liczba wątków w bloku jest określana zmienną typu dim3
- Pozwala to na zdefiniowanie liczby wątków w postaci wymiarów prostopadłościanu
- Ponieważ wymiary liczby dim3, które nie są wyspecyfikowane, przyjmują domyślnei wartość 1, można też zdefiniować liczbę wątków w postaci wymiarów prostokąta
- Mozna też określić bądź jako wymiar liniowy (zwykły int)

Numeracja wątków

- Każdy wątek ma numer (zmienna threadIdx), który jest trójelementowym wektorem określającym pozycję w prostopadłościanie
- Można łatwo uzyskać pojedynczą liczbę określającą id wątku wymnażając współrzędne przez rozmiary prostopadłościanu (bloku), które można uzyskać zmienną blockDim

Siatka bloków

- Bloki są zorganizowane w siatkę
- Identyfikator bloku w siatce jest dany zmienną blockIdx, rozmiar siatki zmienną gridDim
- Bloki są wykonywanie niezależnie, w nieokreslonej kolejności bądź (częściowo) równolegle
- Wątki w ramach bloku mogą się komunikować poprzez wspólną pamięć oraz synchronizować

Przykład

- http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf

Organizacja pamięci

- Prywatna pamięć wątku
- Pamięć bloku dostępna dla wszystkich wątków w bloku w czasie życia bloku
- Pamięć globalna
- Pamięć globalna tylko do odczytu:
  - Pamięć stałych
  - Pamięć tekstur
- Pamięć globalna I globalna tylko do odczytu zachowuje wartości przez czas życia aplikacji

Organizacja pamięci i procesorów

- Wątki CUDA są wykonywane na innym procesorze niż reszta programu (GPU vs. CPU)
- Przyjmuje się nazwy device (GPU) i host (CPU)
- Host i device mają oddzielne obszary pamięci

Możliwości obliczeniowe

- Możliwości urządzenia określone są liczbą w formacie x.y, gdzie:
  - x – major revision
  - y – minor revision
- Major revision określa architekturę rdzenia
- Aktualna wartość x == 2

Elementy

- Program
- Runtime
- CUDA driver API

Kompilacja i wykonywanie

- Kernele mozna pisać w języku architektury CUDA: PTX
- Wygodniej jest pisać w C
- Kod C kompilowany kompilatorem nvcc do postaci PTX bądź obiektu binarnego (cubin)
- Aplikacja ładuje podczas pracy potrzebne kernele w formacie PTX bądź cubin
- W przypadku formatu PTX konieczna jest kompilacja w locie (Just-In-Time)

PTX vs cubin

- Wykorzystanie PTX wydłuża czas ładowania aplikacji (konieczna kompilacja), ale
- Pozwala na użycie ostatecznego kompilatora w wersji z momnetu wykonania (poprawki etc.)
- Pozwala na tworzenie aplikacji działających na różnych architekturach, w tym architekturach nieistniejących w momencie pisania kodu

Kompatybilność

- Cubin dla wersji x.y jest kompatybilny z urządzeniami wersji x.z, gdzie z >= y
- Przy kompilacji należy podać dla jakiej wersji ma być generowany kod
- Rónież kompilacja do PTX wymaga określenia wersji, wiąże się to z wprowadzaniem nowych możliwości w kolejnych wersjach architektury (np. liczb zmiennoprzecinkowych podwójnej precyzji)

CUDA runtime

- Biblioteka cudart
- Funkcje rozpoczynają się od cuda
- Inicjalizacja poprzez pierwsze odwołanie do funkcji bibliotecznej

Pamięć urządzenia

- Dostępna jako:
  - Pamięć linowa – 32 bitowy adres dla urządzeń 1.x, 40 bitowy dla 2.x, można używać wskaźników
  - Tablice CUDA
- Modyfikatory zmiennych:
  - __device__
  - __constant__
  - __shared__

Pamięć liniowa

- cudaError_t cudaMalloc(void ** devPtr, size_t size)
- cudaError_t cudaFree(void * devPtr)
- cudaError_t cudaMemcpy (void * dst, const void * src, size_t count, enum cudaMemcpyKind kind)
  - cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice

Przykład

- http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf

Tablice

- cudaError_t cudaMallocPitch (void ** devPtr, size_t * pitch, size_t width, size_t height)
  - T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;
- cudaError_t cudaMalloc3D (struct cudaPitchedPtr * pitchedDevPtr, struct cudaExtent extent)
  - CudaPitchedPtr: size_t pitch, void * ptr, size_t xsize, size_t ysize
  - cudaExtent: size_t depth, size_t height, size_t width

Przykład

- http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf

Kopiowanie

- cudaError_t cudaMemcpyFromSymbol (void * dst, const char * symbol, size_t count, size_t offset, enum cudaMemcpyKind kind)
- cudaError_t cudaMemcpyToSymbol (const char * symbol, const void * src, size_t count, size_t offset, enum cudaMemcpyKind kind)

Przykład

- http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf

Wywołania asynchroniczne

- Domyślnie następujące wywołania są asynchroniczne:
  - Wywołana kerneli
  - Kopiowania pamieci device-device
  - Kopiowania pamięci host-device dla bloków do 64 kB
  - Kopiowania pamięci funkcjami z przedrostkiem Async
- Wywołania są synchroniczne pod debuggerem oraz przy ustawieniu zmiennej środowiskowej CUDA_LAUNCH_BLOCKING na 1

Strumienie

- Strumień – sekwencja instrukcji wykonywana w zadanym porządku
- Różne strumienie wykonują się w stosunku do siebie w sposób niezsynchronizowany
- Kolejność:
  - Utworzenie strumienia
  - Wykonanie komend (wywołanie kerneli, kopiowanie pomiędzy host a device)
  - Zniszczenie strumienia

Tworzenie i niszczenie strumienia

- cudaError_t cudaStreamCreate (cudaStream_t * pStream)
- cudaError_t cudaStreamDestroy (cudaStream_t stream)

Umieszczanie komend w strumieniu

- Funkcje asynchronicznego kopiowania pamięci i wywołania kernela jako parametr biorą strumień, np.:
  - cudaError_t cudaMemcpyAsync (void* dst, const void * src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream)

Inne operacje na strumieniach

- cudaError_t cudaStreamSynchronize (cudaStream_t stream) – blokuje do zakończenia operacji w strumieniu
- cudaError_t cudaStreamQuery (cudaStream_t stream) – zwraca cudaSuccess jeśli operacje w strumieniu się zakończyły, bądź cudaErrorNotReady jeśli nie
- cudaError_t cudaDeviceSynchronize (void) – blokuje do zakończenia wszystkich operacji

Przykład

- http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf

Pamięć hosta wyłączona ze stronicowania

- Można zaalokować pamięć hosta jako wyłączoną ze stronicowania. Pozwala na:
  - Równoległe wykonywanie kernela i procesu kopiowania
  - Mapowanie obszaru pamięci hosta do obszaru pamięci device
  - Zwiększenie przepustowości pomiędzy pamięcią hosta i pamięcią device
- Nie należy przesadzać z ilością

Pamięć hosta wyłączona ze stronicowania – alokacja i zwalnianie

- cudaError_t cudaMallocHost (void** ptr, size_t size)
- cudaError_t cudaFreeHost (void* ptr)

Zdarzenia

- Zdarzenia (events) mogą być używane do synchronizacji i odmierzania czasu
- Zdarzenie należy utworzyć, zapisać, zniszczyć
- Zapis zdarzenia następuje gdy wszystkie komendy w aktualnym strumieniu zostaną wykonane
- Zapis zdarzenia jest asynchroniczny – aby mieć pewność że nastąpił, należy użyć synchronizacji na zdarzeniu

Zdarzenia

- cudaError_t cudaEventCreate (cudaEvent_t* event)
- cudaError_t cudaEventDestroy (cudaEvent_t event)
- cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream)
- cudaError_t cudaEventSynchronize (cudaEvent_t event)

Przykład

- http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf

Systemy z wieloma kartami

- Można pracować z wieloma kartami
- Wybór karty jest możliwy w każdym momencie
- Przy braku wyboru pracuje się z kartą o numerze 0
- Można wyprowadzić informacje o wszystkich kartach w systemie

Struktura cudaDeviceProp

```
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    size_t totalConstMem;
    int major;
    int minor;
    int clockRate;
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
    int integrated;
    int canMapHostMemory;
    int computeMode;
}
```

Odczyt własności i ustawianie urządzenia

- cudaError_t cudaGetDeviceProperties (struct cudaDeviceProp* prop, int device)
- cudaError_t cudaSetDevice (int device)

Przykład

- http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf