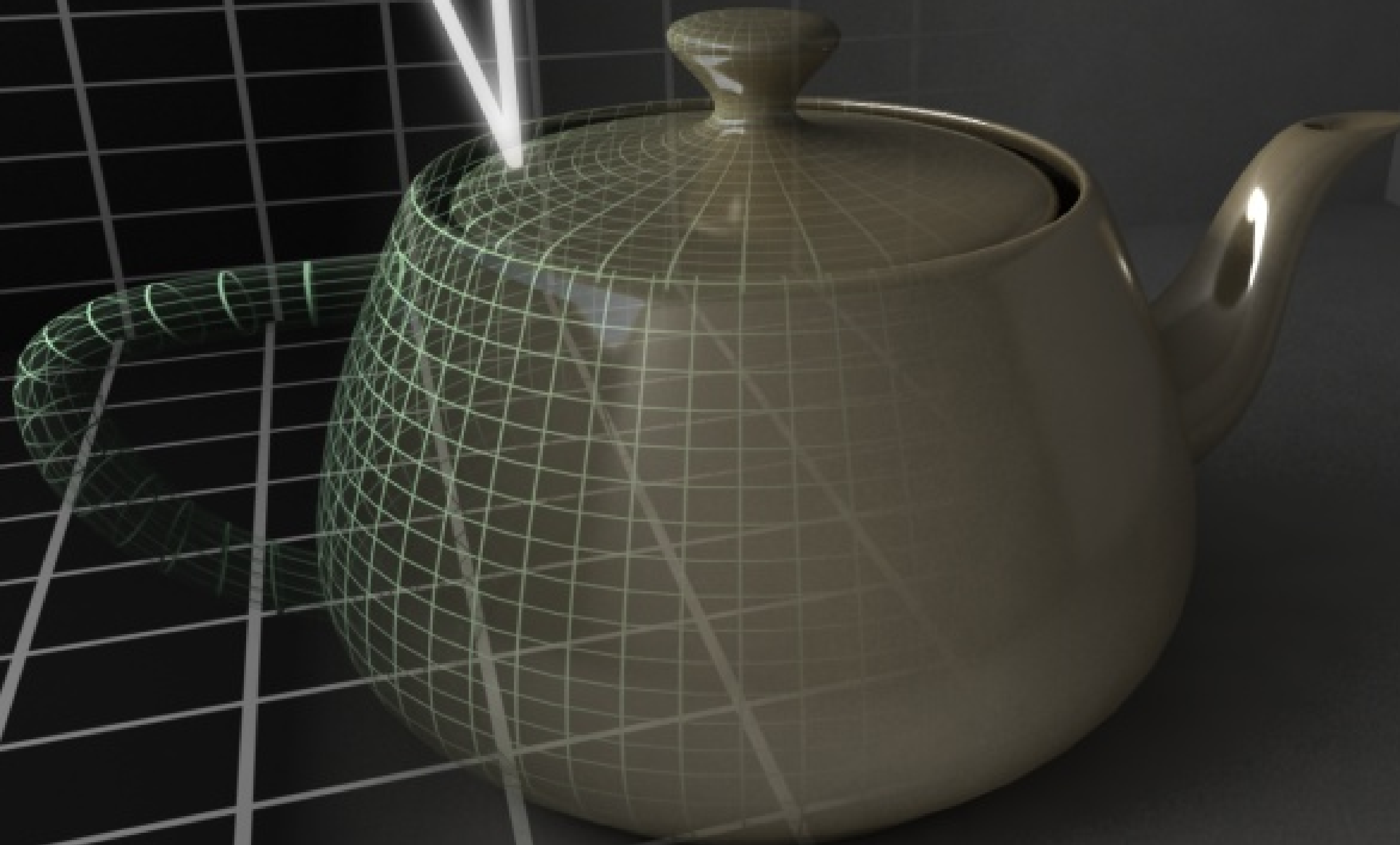
# 3D графика и трасиране на лъчи



http://raytracing-bg.net/

# Рейтрейсинг в реално време
# Паралелизация

# Съдържание

- Анонси
- Real-time raytracing
- Паралелизация

# Анонси

- Напомняне: Тест №2 след седмица
- Домашни работи към лекция №12 (вижте форума)
- Дати за защита на проекти:
  - 12.02 и 21.02 зала 306/ФМИ
- Дата за тестове (който не е правил):
  - 11.02 от 9 до 11ч в 210/ХФ

# Real-time Ray Tracing

- Какво имат предвид хората под Real-time?
  - Много различни работи :)
  - За да може серия от кадри да се възприема като гладка анимация, човешкото око изисква поне ~18 кадъра в секунда
    - В игрите се стремят към поне 60 fps. Иначе, само за гледане: киното е 24, PAL телевизията е ~30 fps.
- „Традиционно“ разделяне в категории според кадраж:
  - < 2 fps: off-line рендериране (не-интерактивно)
  - От 2 до 20 fps – интерактивно рендериране
  - > 20 fps - „real time“ рендериране

# Real-time Ray Tracing

- Как ще постигнем 20 fps с fmiray?!? Това изисква над $10^n$ ($n > 0$) пъти ускорение?!?
- Методи и хитрини има много, но основните подходи се разделят на два:
  - Груба сила: обуздаваме някаква огромна изчислителна мощ (масивна паралелизация на raytracing алгоритъма), и пишем стегнат и добре оптимизиран код за нея.
  - Хитрини: ползваме алгоритми, които работят по-бързо за повечето сцени (но обикновено има начин да се „счупят“). В тази категория влизат различни прекалкулации, Shadow map, Vista buffers, адаптивен рейтрейсинг и много други.

# Оптимизации

- В тази лекция ще се занимаваме основно с първия подход
- Ще започнем от оптимизациите на компилатора:
  - Досега под gcc компилирахме с „-O2 -g0“, което включва немалка част от оптимизациите, но нищо особено
  - Откъде тръгваме: **3.9s** за meshes.fmiray на едно ядро на Athlon 64 @ 1.8 GHz / g++ 4.2

# Оптимизации на компилатора

- -O2 е прекалено общо, особено за код с много floating-point операции като нашия
  - В C++ стандарта е указано много стриктно какви трябва да са резултатите от всяка математическа операция (и подобно за функциите, например sqrt()), и, за да спазва дословно стандарта (заедно с всички частни случаи), компилатора пропуска много възможности за оптимизации
  - Може да оттеглим част от драконовите си изисквания с „`-ffast-math`“. В комбинация „`-O3 -ffast-math`“, meshes.fmiray вече се рендерира за **3.5s** (11% подобрение)

# Оптимизации на компилатора

- Скаларно SSE2
  - С въвеждането на SSE1 (1999г.) и SSE2 (2001г.), Intel променят съществено начина, по който могат да се извършват floating-point операциите. Вече не е необходимо да се разчита на стария x87 модел. Разликите са изцяло архитектурни, но новите инструкции са много по-удобни за компилаторите. Ако накараме компилатора да ползва тях, може да се постигне осезаемо ускорение (но изисква процесор със SSE2 инструкции)
  - „`-O3 -ffast-math -msse -msse2 -mfpmath=sse`“ постига **3.0s** (още 15%)

# Оптимизации на компилатора

- SSE2 се поддържа (и ползва по подразбиране) в 64-битови приложения. 64-битовите приложения се възползват от допълнителни архитектурни благини (повече регистри например), и при тях обикновено се наблюдава допълнително 10-20% ускорение спрямо 32-битов код

# SIMD

- С въвеждането на MMX и SSE, на сцената излиза и концепцията SIMD (**S**ingle **I**nstruction, **M**ultiple **D**ata)
- Най-общо казано, при нея, някакъв масив от данни се подлагат почленно на някаква еднообразна операция, например събиране
    - Това се нарича „векторен" модел; превръщането на обикновен алгоритъм от скаларен до векторен се нарича „векторизация" и може да допринесе до големи ускорения (почти 400%) при огромни масиви от данни, за които операциите са еднообразни

# SIMD - пример

- Искаме да съберем два масива от float числа:

```
void addArrays(float arr1[], float arr2[], int N)
{
    for (int i = 0; i < N; i++) arr1[i] += arr2[i];
}

#include <xmmintrin.h>
void addArraysSSE(float arr1[], float arr2[], int N)
{
    __m128* a = (__m128*) arr1;
    __m128* b = (__m128*) arr2;
    for (int i = 0; i < N/4; i++) a[i] = _mm_add_ps(a[i], b[i]);
}
```

```
...
.L4:
    movss   (%edx,%eax,4), %xmm0
    addss   (%ebx,%eax,4), %xmm0
    movss   %xmm0, (%edx,%eax,4)
    incl    %eax
    cmpl    %ecx, %eax
    jne .L4
.L5:
...

.L12:
    movaps  (%eax,%ebx), %xmm0
    addps   (%eax,%esi), %xmm0
    movaps  %xmm0, (%eax,%ebx)
    incl    %edx
    addl    $16, %eax
    cmpl    %ecx, %edx
    jne .L12

.L13:
```

# SIMD - пример

- Т.е., втората функция върши 4 пъти по-малко работа, ако приемем, че `addps` не е по-бавно от `addss`.
- В компютърната графика, векторизацията може да се приложи в широк спектър от алгоритми
  - Структурите от данни (Color, Vector, Matrix и т.н.) често са векторни така или иначе
    - Например, Color::operator+ може да се реализира с addps вместо с 3 addss-а.
- Но ускорението не винаги е колкото очакваме; например, няма смисъл да векторизираме Vector класа.

# SIMD

- Някои от съвременните компилатори (например Intel C++ compiler) могат да генерират векторен код, без човек да им е подсказал изрично с intrinsics. Това се нарича автовекторизация
  - Но все още е трудно да се види голяма полза от нея в обикновени програми
- Като цяло, SIMD трябва да се ползва за отделни функции, боравещи с големи планини от данни по еднообразен начин. Често се изисква ръчно написан код (intrinsics, или, най-добре, направо на асемблер).

# Оптимизации

- В сегашния код на fmiray има много места, на които можем да добавим малки оптимизацийки откъм самото кодиране:
  - Да подаваме const Vector& и const Ray& вместо само Vector или Ray;
  - В Mesh, да не изчисляваме B-A, C-A и (B-A)^(C-A) всеки път, а да ги пазим в Triangle класа;
  - Да намалим броя на деленията до минимум;
  - В някои случаи, да махнем операторите за цикъл на някои от късите цикли (с 2-3 итерации) и да ги заменим еквивалентния брой копия на тялото на цикъл (loop unrolling);
  - ...и много други...
- Това довежда до подобрение: **2.5s** (17%)

# Оптимизации на компилатора

- Profile-guided optimizations (PGO)
  - Докато генерира кода, често компилатора има много възможности за избор как да го разположи (например, в стандартен if-then-else, then или else блока да бъде първи?). Понеже не знае как обикновено ще се развива действието в програмата, компилатора ползва догадки, които невинаги са верни
  - При PGO, се компилира „специална“ версия, която е „инструментирана“, и по време на работа на програмата събира статистики как е протекъл всеки for цикъл, всеки if-then-else, ...

# Profile-guided optimizations

- След като програмата се пусне веднъж върху данните, с които се очаква да работи в реалността, събраните статистики (профил) се използват за да се прекомпилира още веднъж. Така компилатора има информация за потока на изпълнение на програмата при типична употреба и ползва истинска информация вместо догадки.
- Обикновено: 0-10% ускорение от тази фаза

# Още за оптимизациите

- Съществуват още много компилаторни магии (inlining, link-time code generation (whole program optimization), функционални атрибути (при GCC)...)
- Използването им е специфично според случая
- Как да разберем кои 10% от програмата отнемат най-много време и да концентрираме усилията си там?
  - Intel VTune, AMD CodeAnalyst... - ползват debug и performance регистрите на процесорите
  - Valgrind (cachegrind, callgrind) – ползва виртуална машина
  - За по-добри статистики, inlining-ът трябва да е изключен

# Оптимизации – сметката дотук

| **Оптимизация** | **Време** |
|---|---|
| -O2 -g | 3.9s |
| -O3 -ffast-math | 3.5s |
| -msse -msse2 -mfpmath=sse | 3.0s |
| **const** T& вместо T | 2.9s |
| AB, AC и ABcrossAC в class Triangle | 2.8s |
| Loop unrolling, махане на деленията, ... | 2.5s |

# Паралелизъм

- Както сме казвали вече, raytracing-a позволява да се паралелизира почти безкрайно. Всеки отделен пиксел може да се сметне на отделно ядро. Лъчите не зависят един от друг, алгоритъма досега не ползва глобална памет (или, ако ползва, тя е Read-only)
- Необходимостта от синхронизация между отделните нишки в една машина (или отделните машини в една render farm) е малка; не се губи много време там.
- Някои от съпътстващите алгоритми също могат да се паралелизират (например, строежа на K-d дървото, макар и да е малко по-сложно)

# Паралелизъм върху една машина

- Съвременните OS позволяват една програма да има повече от една нишка на изпълнение, т.е. програмата да върши няколко неща едновременно
  - Нишките ползват споделена памет
  - Ако в компютъра има повече от един процесор/ядро/логически процесор, може отделните дейности на програмата да се изпълняват от отделните процесори/ядра/ЛП, които работят независимо един от друг
  - Ако алгоритъма не изисква работа с огромно количество памет (при което bottleneck-а ще се окаже връзката Процесор<->Памет), то задачите могат да работят едновременно, без да си пречат една на друга

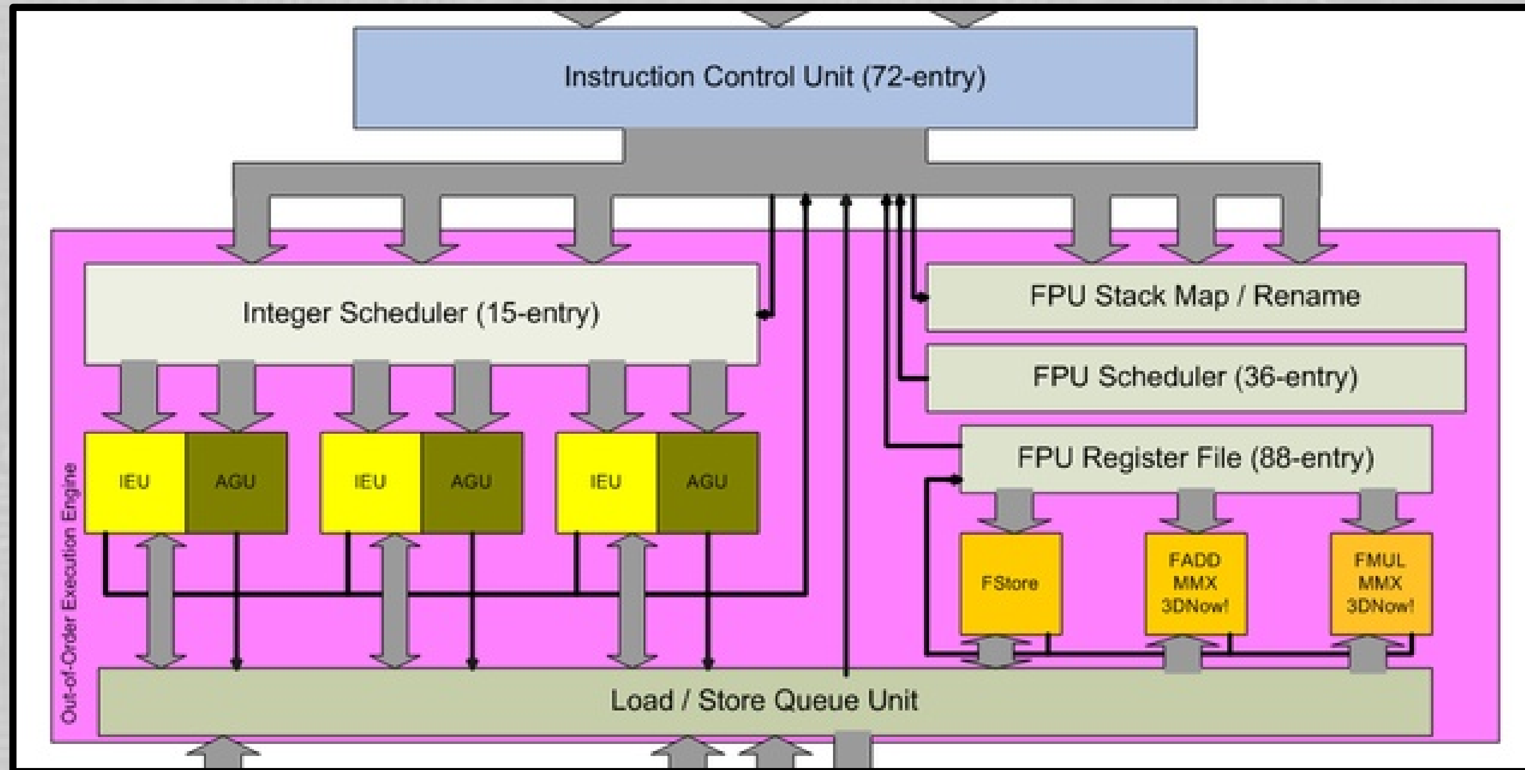
# Паралелизъм върху една машина

- В съвременните компютри има най-различни технологии, които от OS гледна точка са видими като отделни процесори в системата
  - Логически процесори (Hyperthreading)
  - Повече от едно ядро в един процесор
  - Повече от един процесор в машината
- Съществено е, че различните технологии се различават в степените си на разделение (доколко са „независими“ и могат да си „пречат“ една на друга), и в способностите да си комуникират бързо (за синхронизация и пр.)

# Четири степени на разделение

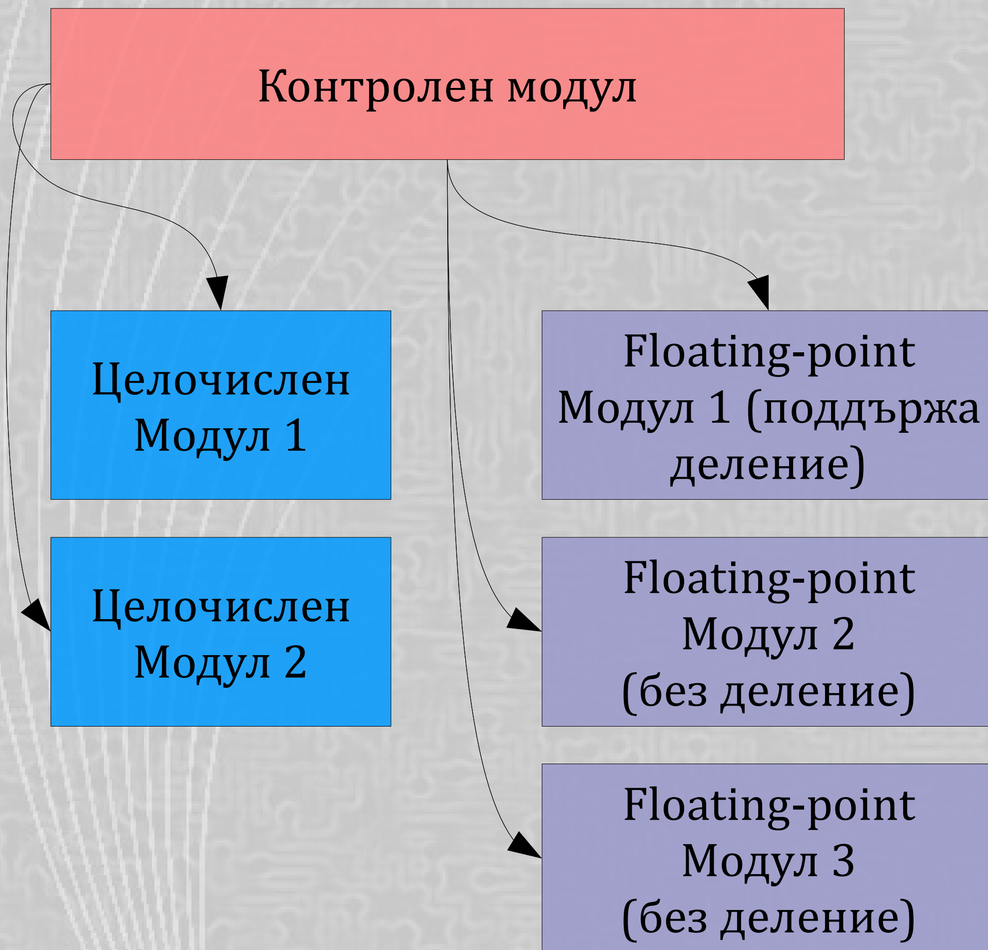
- Подредени откъм по-близки към по-раздалечени:
  - Логически процесори (hyper-threading)
    - Ползват общи изчислителни ресурси („пречат“ си много), но комуникират много бързо
  - Ядра в един процесор
    - Ползват общ кеш; комуникацията е бърза
  - Отделни процесори
    - Ползват собствени кешове, но обща памет. Комуникацията е по-бавна от многоядрения вариант
  - Отделни машини
    - Много по-бавна комуникация, но с добра независимост и мащабируемост

# Hyper-threading



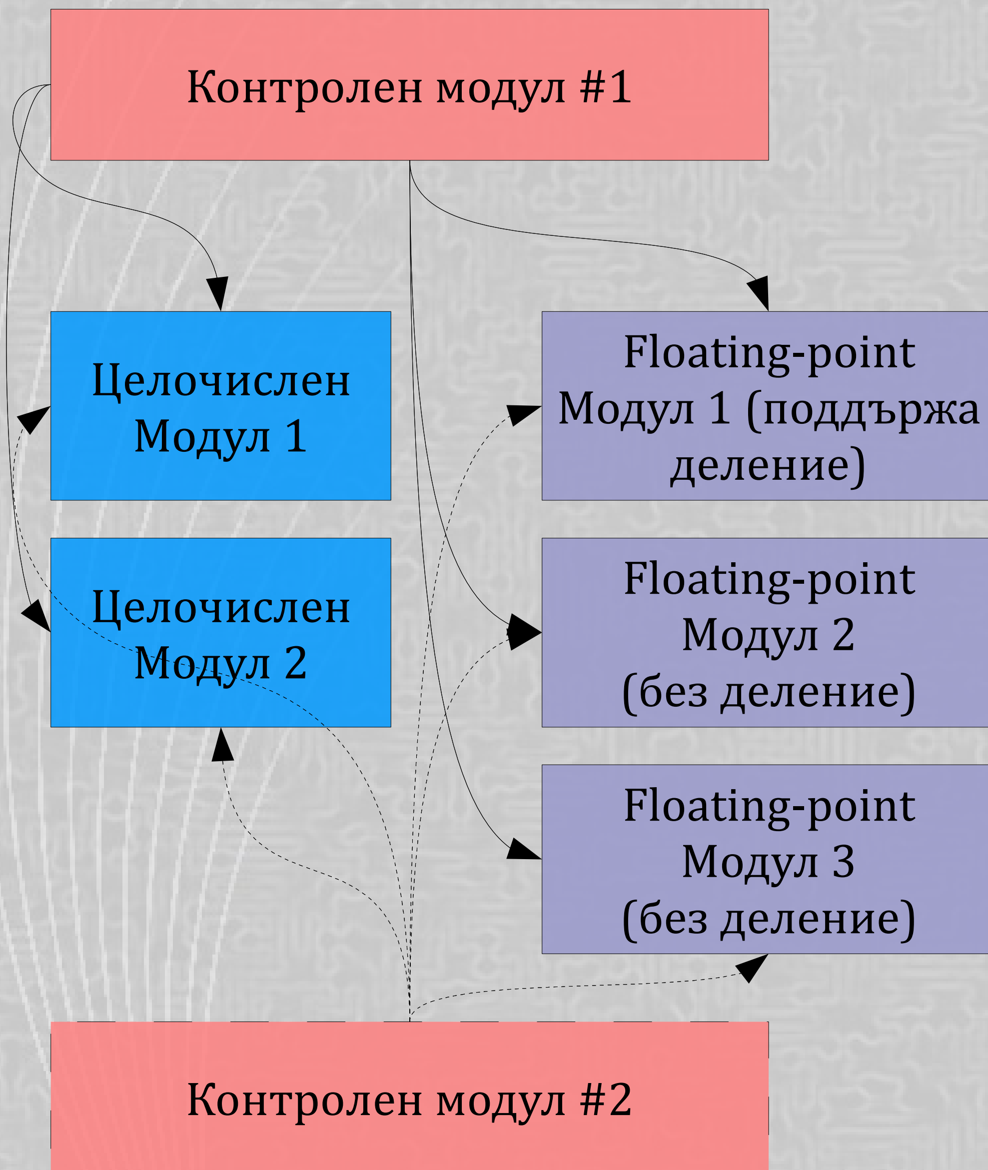

Съвременните x86 процесори са оптимизирани за скорост и са суперскаларни (могат да извършват повече от една операция на такт). За целта, част от модулите в процесора са дублирани (напр.,   IEU модулите в картинката (модули за целочислени операции) – процесора може да извършва 3 целочислени събирания едновременно)

# Hyper-threading



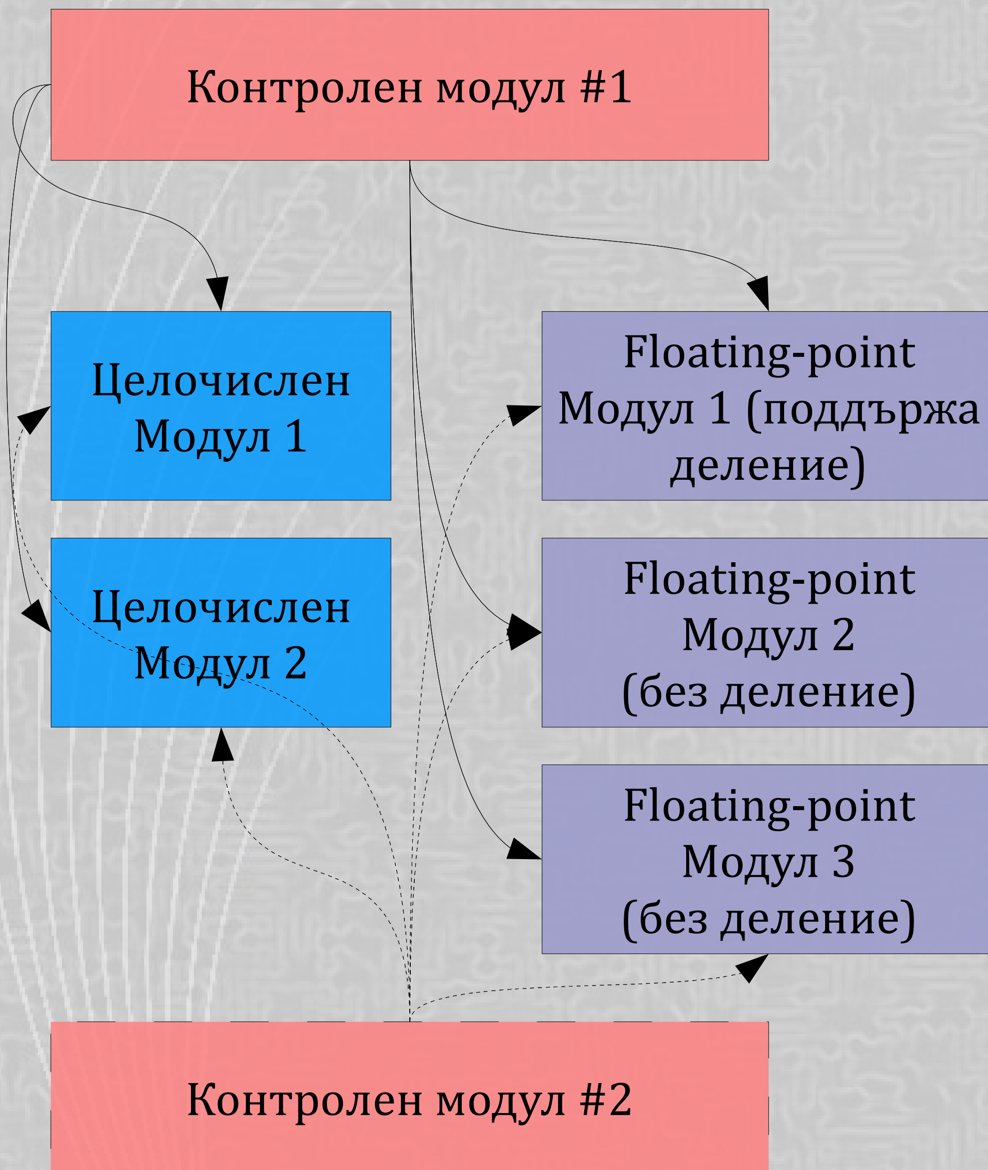

- Тук е даден опростен пример за процесор
  - Контролният модул изпълнява входящите инструкции и държи сметка коя инструкция на кой модул може да се изпълни, както и кои са свободните модули
  - Ако кодът е само FP операции, целочислените модули ще бездействат

# Hyper-threading




- Тук се намесва Hyper-threading-а. Може да добавим втори контролен модул (който се занимава с администрацията и представлява втори логически процесор с пълен комплект от регистри, архитектурно състояние и т.н., т.е. цял втори процесор, но без изчислителните модули)

# Hyper-threading




- Например, докато КМ1 се занимава с тежък откъм FP алгоритъм, вторият логически процесор може да извършва целочислени операции върху свободните модули
- Нерядко се налага изпълнението да спре (cache miss, branch misprediction), и тогава отново остават свободни ресурси

# Hyper-threading

- Hyper-threading-ът работи много добре, когато на различните логически процесори са пуснати различни алгоритми (товарещи различни части от изпълнителните модули), или ако алгоритъмът е сложен/неоптимизиран и не успява да натовари всички изчислителни ресурси сам
- На добре оптимизиран код, двата логически процесора се „бият“ за едни и същи ресурси и ползата е малка
- Груби цифри: 20% ускорение при Pentium 4, 40-60% за Intel Atom и Intel Nehalem (Core i7 et al)

# Dual/Quad core процесори

- Човек вече трудно може да си купи едноядрена машина, дори и да иска :)
- В тези системи, процесорът е само един, но в него има две или повече ядра, всяко от което представлява напълно функционален процесор
  - Системи със споделени кешове
    - Подобрена употреба на кеша (ако единия thread изисква много, а другия – почти никакъв кеш, то cache manager-a може да разпредели кеша пропорционално). По-бърза комуникация
  - Несподелени кешове
    - Налага се поради производствени съображения (отделни кристали)

# Дву- и много-процесорни системи

- В такава система, на самото дъно има повече от един процесор (всеки от който може и да е многоядрен, евентуално с HT...)
- Между процесорите има специална свръзка за високоскоростна комуникация, която е необходима за държат отделните кешове на процесорите *кохерентни*, т.е. да отразяват еднакво съдържанието на системната памет
  - Проблемът възниква от това, че кешовете са направени с идеята да са прозрачни, т.е. от гледна точка на потребителя, те не съществуват, има само памет.

# Кохерентност между кешовете

- Нека имаме два процесора, и по някаква причина, и двата са избрали да кешират една и съща област от системната памет
- Процесор 1 променя един байт в тази област
  - От съображения за скорост, промяната се записва в кеша му, но отразяването на промяната в системната памет се забавя с идеята да може после да се комбинира и с други записи
  - Ако Процесор 2 се опита да прочете същия байт по същото време, той ще прочете копието от собствения си кеш, което вече е невалидно

# Кохерентност между кешовете

- Даже и Процесор 1 да запише промените в системната памет по някое време, то Процесор 2 все още не знае, че има промяна, и може да остане с невалидни данни неограничено дълго време
- За целта, между двата процесора има специален протокол за спешен обмен на данни между кешовете, в случай на подобни конфликти
- Такъв протокол е много по-лесен за реализация при двуядрен процесор (комуникацията е вътре в чипа), отколкото при двупроцесорна система (комуникацията е по специални пътечки в дънната платка)

# False sharing

- В някои случаи, двата процесора може да не променят едно и също място в паметта, а да „пипат“ в съседни клетки (индивидуални), без да си презаписват един на друг данните
  - Това е съвсем валидно поведение за една многонишкова програма – всяка нишка си има някакви променливи, които ползва за собствени цели; просто са се оказали в съседство с променливите на други нишки
- В този случай, обаче, също може да имаме голям cache coherence трафик между процесорите

# False sharing

- Причината за това е, че кешът е организиран в по-големи логически единици, наречени cache lines (например по 64 или 128 байта).
  - Ако двете клетки памет са се оказали в един и същи cache line, то процесорите засичат, че има опити да се модифицира едновременно един и същи cache line и това предизвиква cache coherence трафика.
    - Напълно излишно, тъй като процесорите реално не се „бият“ за една и съща клетка в паметта
  - Този феномен се нарича False sharing

# False sharing

- False sharing (за разлика от „true“ sharing) може да възникне спонтанно, без програмиста да го очаква, в съвсем валиден и правилен многонишков код („true“ sharing-а обикновено се предизвиква целенасочено, често с цел синхронизация, и програмистът се старае да го ограничи доколкото може)

# Скорост на компютрите (FLOPS)

- При масивни изчисления, скоростта на хардуера обикновено се измерва във FLOPS (Floating-point operations per second)
- Athlon 64 dualcore @ 2.3 GHz постига ~4.6 GigaFLOPS
- Core i7 @ 3.2 GHz (четири ядра + Hyper-threading) постига ~70 GFLOPS
- Видеокартите разчитат на по-ниски честоти, но много по-голяма степен на паралелизация, което обуславя повече FLOPS:
  - Ati/AMD Radeon HD 4850: ~1000 GFLOPS

# GPU

- В GPU-тата на видеокартите се крие голяма мощ (огромен брой отделни процесори – 512 и нагоре в последните поколения), което е идеално за ray-tracing, но как да впрегнем тази сила за нашите цели?
- Предишните решения предлагаха да се ползват пикселните шейдъри (които представляват малки подпрограмки, написани на шейдърен език като Cg, GLSL, HLSL и прочее, и изпълнявани директно на видеокартата след компилация)

# Pixel shaders

- Pixel shader-ите са предназначени за да се разширят способностите за шейдване във видеокартите, и като такива, поставят много сурови ограничения какво може да се прави в един Pixel shader
- Това ги прави сравнително неподходящи за General Purpose изчисления (GPGPU)
  - Липсва рекурсия, директен достъп до централната памет и до OS calls
  - Средата е силно наклонена към шейдинг изчисления; няма масиви – има текстури; вместо изходни стойности, има output Color и прочее

# CUDA & OpenCL

- Тези ограничения в GPGPU подхода най-накрая са решени чрез CUDA и OpenCL
  - CUDA е собствена разработка на nVidia, а OpenCL е общ стандарт, разработван от Apple, AMD, Intel, nVidia, ...
  - И двете технологии ползват собствени C-like езици за писане на *ядра* (*kernels*). Всяко ядро е представлява паралелна част от програмата и при изпълнението върху GPU, стотици копия от едно ядро работят едновременно върху различни данни.
  - Езиците са подмножество на C – не се поддържат неща като: указатели към функции, рекурсия; нямат стандартната библиотека и пр.

# CUDA & OpenCL

- … но за сметка на това има поддръжка на важни (от гледна точка на GPU архитектурите неща) като
  - Разделение на паметта по типове (NUMA – Non-uniform memory access)
  - Half-float типове
  - Разнообразни hint-ове към компилатора, как да се паралелизира кода
  - Разпределяне на задачите в working groups…
- OpenCL може да работи и върху различен от GPU хардуер (CPU, Cell, …)

# Cell

- Архитектура, разработена от Sony, IBM и Toshiba
  - Най-известна за използването си в PlayStation 3
- Състои се от 1 централен, командващ процесор (IBM Power-базиран) и няколко „изчисляващи“ процесора (SPEs). SPE-тата представляват опростени процесори, разработени специално за изчисляване на големи количества данни и не са еквивалентни на CPU (например, нямат виртуална памет, branch predictors, и прочее).

# Cell

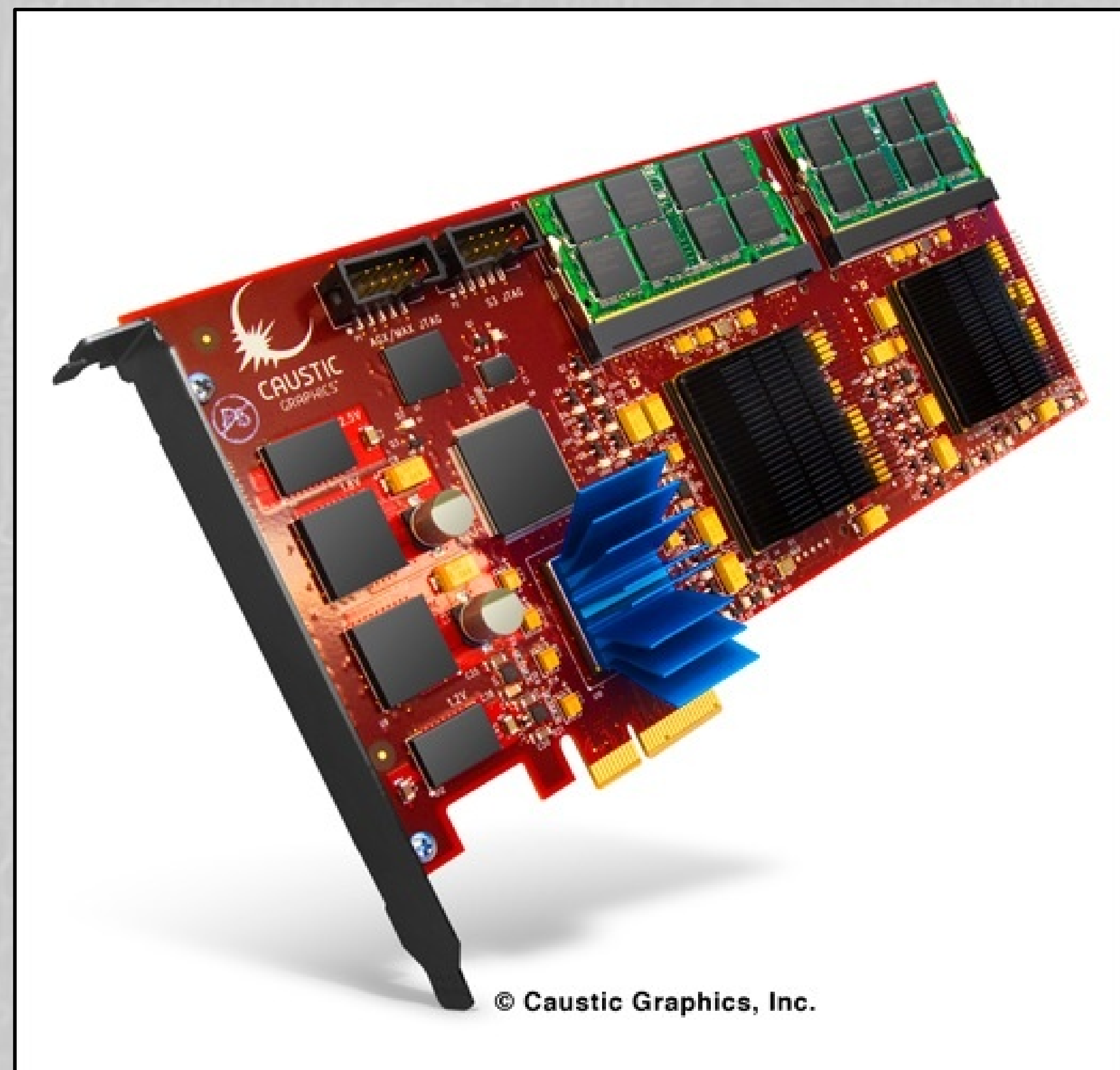
- Заради уникалните си (и сложни) изисквания при програмирането, направата на високо-производителна Cell програма е доста трудно, но понякога си заслужава
  - Cell е вече поостаряла разработка и съвременните x86 процесори лесно я надминават като скорост...
  - ... но откъм GFLOPS на ват, Cell бие с много (енергийно ефективен е за направа на клъстери)
- Пример: Folding@Home, масивен разпределен проект за изчисление, ползва комбинация от x86 CPUs, GPUs и PS3 Cell. Над 50% от работата се извършва **само** от PS3.

# nVidia Tesla & nVidia Fermi

- Специализирани карти, създадени само и единствено за high-performance изчисления
- Приличат на видеокарти, но нямат видео-изход и имат специфични допълнения за HPC нишата

# Специализирани raytracing решения

- На пазара в последно време излизат и специализирани хардуерни решения, специално с цел raytracing
- Например Caustic Graphic's Caustic One
  - Рейтрейсингът се изпълнява върху картата и има API, което позволява човек да си подкара собствения raytracer на нея



© Caustic Graphics, Inc.

# Рендер ферми

- От гледна точка на дизайнерските студия, най-лесния начин за паралелизация на изчисленията при raytracing-а е да се разбие изчислението върху клъстер от компютри
  - Технологията поставя лимити върху броя процесори, които може да натъпчем в една машина. От друга страна, можем да сложим колкото си искаме компютри
  - Програмирането в Distributed rendering среда се различава съществено от простата паралелизация в многоядрените/многопроцесорните системи

# Рендер ферми

- Специфични проблеми и различия на Distributed rendering-a:
  - Потенциално различни машини (различна производителност)
  - Много по-бавна комуникация
  - Трябва да се споделят огромни количества данни (текстури, модели, ...)
  - Fault-tolerance – някоя от машините може да излезе от строя и трябва да се справяме с това
  - По-хитро разделяне на задачите, за да се избегне прекомерната комуникация

# Real-time render farms

- Заради споменатите особености, направата на real-time renderer чрез много машини е особено трудна задача, но не невъзможна
  - От OpenRT се справят доста добре с огромни сцени и голям брой машини (48 двуядрени за наистина тежки сцени) (от друга страна, те работят по-скоро на интерактивно ниво, < 10 fps)

# GFLOPS сравнение

| **Машина** | **FLOPS (single precision)** |
|---|---|
| Single-core Athlon 64 @ 2.3 GHz | 2.6 GFLOPS |
| Core i7 @ 3.2 GHz | 70 GFLOPS |
| Radeon HD 4850 | 1000 GFLOPS |
| NV Tesla C2070 | 1260 GFLOPS |
| SETI@Home | 700 TFLOPS |
| IBM Roadrunner (7000 Opterons, 13000 Cells) | > 1 PetaFLOP ($10^{15}$ FLOPS) |
| Folding@Home | 5+ PetaFLOPS |

# Multithreaded coding

- За писане на многонишков софтуер, за съжаление, C++ не разполага с добри възможности в стандартната си библиотека
  - Ще има, когато излезе ~~C++0x~~ C++1x
- Pthreads: POSIX (Portable) threads
  - Поддържа се на POSIX системите (Linux, FreeBSD, Mac OS X), има и Windows port
- Win32 threads: нативната библиотека под Windows
  - Когато се гони скорост, има смисъл да се ползва нея; pthreads не е решение

# Multithreaded coding

- Win32 threads и pthreads са примери за low-level threading APIs; те са процедурни (C библиотеки) и дават на програмиста голяма свобода как точно ~~да си застреля крака~~ да реализира алгоритъма
    - MT програмирането е трудно, затова е подходящо да се ползват high-level threading API-та, които обгръщат Win32/pthreads примитивите по подходящ начин, като ги правят по-безопасни
    - Примери: Boost.Thread, SDL Threads, wxWidgets

# Multithreaded coding

- SDL Threads за съжаление са прекалено опростени за да ни свършат работа
  - Те не са и истинска high-level библиотека; по-скоро предствляват cross-platform low level.
- За целта ще ползваме една домашно сварена библиотека, която е C++ wrapper около pthreads и Win32 threads: CXXPTL
  - svn co https://fract.svn.sourceforge.net/svnroot/fract/cxxptl
  - Поддържа Windows, Linux, Mac OS X
  - Минималистична, с прост High-level design.

# Multithreaded coding

- Ще започнем с една примерна многонишкова програма, която пресмята числото π чрез следната формула:

$$\sum_{x=1}^{\infty} \frac{1}{x^2} = \frac{\pi^2}{6}$$

(от която следва):

$$\sqrt{6 \sum_{x=1}^{N} \frac{1}{x^2}} \approx \pi$$

# Multithreaded coding

- Примерна реализация (без нишки):

```
int main(int argc, char** argv)
{
    unsigned ticks = SDL_GetTicks();
    double sum = 0;
    for (int x = 300000000; x >= 1; x--)
        sum += 1.0 / ((double) x * x);
    printf("pi = %.10lf\n", sqrt(sum * 6));
    printf("Time required: %d ms\n", SDL_GetTicks() - ticks);
    return 0;
}
```

Време за изпълнение: 3.0s.

# Multithreaded coding

- Многонишкова реализация (на двуядрена машина):

```
#include "cxxptl.h"
int main(int argc, char** argv)
{
	ThreadPool pool;
	unsigned ticks = SDL_GetTicks();
	struct WorkerThread: public Parallel {
		double sum;

		WorkerThread() { sum = 0; }
		void entry(int threadIdx, int threadCount) {
			for (int x = 300000000 - threadIdx; x >= 1; x -= threadCount)
				sum += 1.0 / ((double) x * x);
		}
	} thread;
	pool.run(&thread, get_processor_count());
	printf("pi = %.10lf\n", sqrt(thread.sum * 6));
	printf("Time required: %d ms\n", SDL_GetTicks() - ticks);
	return 0;
}
```

- 1.5s, но грешен резултат – има race condition!

# Multithreaded coding

```
struct WorkerThread: public Parallel {
	double sum;
	Mutex lock;

	WorkerThread() {
		sum = 0;
	}
	void entry(int threadIdx, int threadCount) {
		for (int x = 300000000 - threadIdx; x >= 1; x -= threadCount) {
			lock.enter();
			sum += 1.0 / ((double) x * x);
			lock.leave();
		}
	}
} thread;
```

- Race conditions обикновено се избягват, като опасната част се вкара в критична секция, заключена с Mutex
  - Правилен резултат, но е много бавно: 50s!

# Multithreaded coding

```
struct WorkerThread: public Parallel {
	double sums[64];

	WorkerThread() {
		memset(sums, 0, sizeof(sums));
	}
	void entry(int threadIdx, int threadCount) {
		for (int x = 300000000 - threadIdx; x >= 1; x -= threadCount)
			sums[threadIdx] += 1.0 / ((double) x * x);
	}
} thread;
pool.run(&thread, get_processor_count());
double sum = 0;
for (int i = 0; i < get_processor_count(); i++) sum += thread.sums[i];
printf("pi = %.10lf\n", sqrt(sum * 6));
```

- Това е ОК на многоядрена машина (1.7s), но на много-процесорна ще се забави доста заради false sharing проблема

# Multithreaded coding

- Правилна реализация (верен резултат, 1.5s):

```
struct WorkerThread: public Parallel {
	double sums[64];

	void entry(int threadIdx, int threadCount) {
		double sum = 0;
		for (int x = 300000000 - threadIdx; x >= 1; x -= threadCount)
			sum += 1.0 / ((double) x * x);
		sums[threadIdx] = sum;
	}
} thread;
```

# Разпределение на задачата на подзадачи

- Целта на всеки паралелен алгоритъм е да раздели задачата на подзадачи, които да могат независимо да се сметнат от отделните нишки
- В нашия код, голямата задача (рендерирането на картинката) вече е разделена на подзадачи (квадратчета 64x64 пиксела – *региони*)
  - Offtopic: разделянето на региони е направено с цел скорост. Съседните лъчи в едно квадратче най-вероятно ще ударят едни и същи примитиви (някаква малка група). Ако разделянето беше на редове, лъчите щяха да са доста по-разпръснати из сцената (-> повече улучени примитиви)

# Разпределение на задачата на подзадачи

- При повече примитиви, кешовете на процесора ще трябва да държат повече данни и, при тежка сцена, няма да стигнат, докато разделението на квадратчета е по-благосклонно към кешовете
  - Т.е., при лека сцена няма голямо значение, но при тежки сцени, Jacco Bikker споменава за 10% подобрение в скоростта спрямо разделението по редове в картинката

# Разпределение на задачата на подзадачи

- Да се върнем на разпределението. Ние имаме N региона, които трябва да се рендерират, и M нишки, на които да им ги разпределим. Въпросът е: как да им раздадем регионите по оптимален начин?
- Вариант 1: статично разпределение (всяка нишка получава 1/M-та от регионите)
  - Но това разпределение е потенциално разбалансирано. Например, ако имаме 2 нишки, и сцена с небе (в небето няма нищо интересно и се рендерира бързо), то долната половина от картинката е много по-тежка от горната

# Разпределение на задачата на подзадачи

- Разбира се, може да решим отчасти този проблем, като раздадем регионите „шахматно“
  - Генерализирано за M нишки: нишка номер $k$ получава регионите $i$*M + $k$ за $i$ = 0, 1, 2, ...
- Но статичното разпределение винаги има опасност да е леко неоптимално, особено в realtime проложения:
  - Нишките може да не започнат работа по едно и също време
  - Някой от процесорите може да бъде задържан от OS, за да свърши някаква друга задача
  - Някоя от нишките може да получи 1-2 тежки региона („костеливи орехи“) и да изгуби доста време с тях, докато другите нишки приключат с работата си и бездействат, вместо да й помогнат

# Разпределение на задачата на подзадачи

- Вариант 2: може просто да не ползваме статично разпределение, а да реализираме някакъв механизъм за динамично разпределение на регионите:

```
Mutex lock;
int cursor = 0;
int getNextRegion() {
	lock.enter();
	int returnValue = cursor;
	cursor++;
	lock.leave();
	return returnValue;
}
```

```
...
void multiThreadedTask() {
...
	while (1) {
		int x = getNextRegion();
		if (x >= N) break;
		// ... render region number x ...
	}
...
}
```

# Разпределение на задачите на подзадачи

- getNextRegion() трябва да е защитена с Mutex, защото се ползва от много нишки и споделя обща променлива
  - Но, както показахме, синхронизирането с Mutex е бавно, ако се случва доста често. За realtime цели, това наистина би бавило
  - За щастие, съществуват начини да реализираме простите целочислени операции като „i++“, „i+=x“ *атомарно*, т.е. цялата операция да стане непрекъсваема. Така, необходимостта от заключване отпада.
  - В cxxptl, това е реализирано в *InterlockedInt* класа; той се държи като *int* променлива, но операциите му са атомарни

# Разпределение на задачите на подзадачи

- Реализация на getNextRegion() с InterlockedInt:

```
InterlockedInt cursor;
int getNextRegion() {
    int returnValue = cursor++;
    return returnValue;
}
```

  - Вземането на стойността на cursor, и увеличаването с 1, са една цяла, непрекъсваема операция (вътрешно се реализира с инструкцията lock xadd), така че няма race condition
  - Единственото забавяне идва от cache coherency трафика („true“ sharing), който е неизбежен