CHAOSGROUP

Да си изцапаме ръцете с GPU-та

Александър Соклев

Днес ще си говорим за оптимизации

- Техники и похвати, които да направят алгоритъма ни по-бърз.
  - » **Без да променяме самия алгоритъм!**
- Ще говорим за това:
  - как да помогнем на компилатора да генерира по-бърз код
  - как можем да приложим на практика познанията си за работата на компютрите на ниско ниво, за да направим приложенията си още по-производителни.
  - как да накараме програмата ни да използва ефикасно всички ресурси, които компютърът ни предоставя

Откъде да започнем...

- Като за начало ще си изберем една достатъчно тежка задача
  - Такива задачи - много :)
    - Днес съм избрал да обърнем внимание на филтриране на изображения (a.k.a. Image Convolution)
    - Това е една от най-стандартните задачи от сферата на Image Processing.
  - Защо именно нея?
    - Защото се пише лесно :)
    - Защото винаги, когато поискаме, лесно можем да я утежним
    - Защото така :P

ConvoWHAT?

- Малко терминология:
  - Конволюцията е процес, в който събираме елемент от изображение (пиксел) с претеглената стойност на непосредствените му съседи.
  - Тегловите коефициенти, които използваме, удобно могат да бъдат записани в матрица, която наричаме маска (mask) или ядро (kernel)
  - Радиусът (radius) на ядрото, още наричан съседство (neighbourhood), определя броя пиксели от всяка страна на текущия, които ще вземем предвид, когато прилагаме конволюция над изображението.
  - За да не бъда голословен, всичко това ще демонстрирам с малко примери.

Computational Photography by Georgia Tech

https://www.udacity.com/course/computational-photography--ud955

Как работи конволюцията?

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	90	90	90	90	90	0	0
0	0	90	0	90	90	90	0	0
0	0	90	90	90	90	90	0	0
0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0
0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Smooth Kernel

$$1/9 * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

	10							

Computational Photography by Georgia Tech

https://www.udacity.com/course/computational-photography--ud955

Как работи конволюцията?

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	90	90	90	90	90	0	0
0	0	90	0	90	90	90	0	0
0	0	90	90	90	90	90	0	0
0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0
0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Smooth Kernel

$$1/9 * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

	10	20						

Computational Photography by Georgia Tech

https://www.udacity.com/course/computational-photography--ud955

Как работи конволюцията?

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	90	90	90	90	90	0	0
0	0	90	0	90	90	90	0	0
0	0	90	90	90	90	90	0	0
0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0
0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Smooth Kernel

$$1/9 * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

	10	20	30	30	30	20		

Как работи конволюцията?

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	90	90	90	90	90	0	0
0	0	90	0	90	90	90	0	0
0	0	90	90	90	90	90	0	0
0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0
0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Smooth Kernel

$$1/9 * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

	10	20	30	30	30	20	10	
	20	30	50	50	60	40	20	
	30	50	80	80	90	60	30	
	30	50	80	80	90	60	30	
	20	40	60	60	60	40	20	
	20	30	30	30	30	20	10	
	10	10	0	0	0	0	0	

Computational Photography by Georgia Tech

https://www.udacity.com/course/computational-photography--ud955

Как работи конволюцията?

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	90	90	90	90	90	0	0
0	0	90	0	90	90	90	0	0
0	0	90	90	90	90	90	0	0
0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0
0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Smooth Kernel

$$1/9 * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

0	0	0	0	0	0	0	0	0
0	10	20	30	30	30	20	10	0
0	20	30	50	50	60	40	20	0
0	30	50	80	80	90	60	30	0
0	30	50	80	80	90	60	30	0
0	20	40	60	60	60	40	20	0
0	20	30	30	30	30	20	10	0
0	10	10	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Computational Photography by Georgia Tech

https://www.udacity.com/course/computational-photography--ud955

Други примери за маски

- Вече разгледахме един вид маска, усредняваща стойностите на пикселите в изображението - Box filter. Подобни са Gaussian & Median филтрите*
- Друг пример за маска е идентитетът. Това е маска, която приложена над изображението, връща същото изображение
- Последният тип маска, на който ще се спра днес, е простичък вариант на Sharp - филтър, който подсилва интензитета по ръбовете в изображението, създавайки усещането, че то е по-добре фокусирано

$$1/9 * \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

* Няма да навлизаме в детайли. Който има интерес, може да разгледа курса на Georgia Tech.

Computational Photography by Georgia Tech https://www.udacity.com/course/computational-photography--ud955

А сега по същество:

Нещо не е наред...

- На компютъра му трябва почти **една секунда**, за да приложи филтър идентитет с радиус едва 3x3 над изображение 850x350 (~0.3 MPix) !
- Това е резултатът от компилация на кода с настройките по подразбиране на Visual Studio
- Една от причините за бавното изпълнение на кода е това, че по подразбиране, кодът се компилира за x86, но днешните компютри са x64
- Защо е важно да компилираме за x64?
- Кодът се компилира за дебъг! (Опция на компилатора -O0)
- Компилирайки в Release (-Ox), компилаторът прилага широка гама оптимизации, допълнително, премахва дебъг символите, правейки кода ни по-лек, а оттам и по-бърз.

Debug x86 vs Release x64:

Debug x86 vs Release x64:

Debug x86 vs Release x64:

Все още не сме много ефикасни

- Модерните компютри не са станали много по-бързи от 2005-та насам

- **“The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”**
  (2005, By Herb Sutter)

- За да повишим производителността на нашите програми, трябва да сме паралелни

- Enters OpenMP (Open Multi-processing) API
  - /openmp - опция на компилатора
  - #pragma omp parallel for

Паралелизация:

```
/// Perform standard convolution
int filterWithCPU(uint32 *buffer, const Image& image, const float *kernel, int nbhd) {

    const int imgWidth  = image.getWidth();
    const int imgHeight = image.getHeight();
    const Color *img    = image.getData();

    for (int x = 0; x < imgWidth; x++) {
        for (int y = 0; y < imgHeight; y++) {
            const Color c = convolve(x, y, kernel, nbhd, img, imgWidth, imgHeight);
            buffer[imgWidth*y + x] = c.toUINT32();
        }
    }
    return 0;
}
```

Параелизация:

```
/// Perform standard convolution
int filterWithCPU(uint32 *buffer, const Image& image, const float *kernel, int nbhd) {

    const int imgWidth  = image.getWidth();
    const int imgHeight = image.getHeight();
    const Color *img    = image.getData();
#pragma omp parallel for
    for (int x = 0; x < imgWidth; x++) {
        for (int y = 0; y < imgHeight; y++) {
            const Color c = convolve(x, y, kernel, nbhd, img, imgWidth, imgHeight);
            buffer[imgWidth*y + x] = c.toUINT32();
        }
    }
    return 0;
}
```

Паралелизация:

КАРХ е важен! Memory, Caches, Prefetchers and more...

- RAM-та е бавна. На помощ идват кешовете и хардуерният prefetcher.
- Prefetcher-а ни дава 10-30% ускорение “безплатно”, но лесно можем да загубим този ефект ако пишем кода си наивно.
- Как да помогнем на prefetcher-а?

```
60  Color convolve(int i, int j, const float *kernel, int k,
61                 const Color *img, const int imgWidth, const int imgHeight) {
62      const int kSize = 2*k+1;
63      Color res = Color(0.f, 0.f, 0.f);
64      for (int u = -k; u <= k; u++) {
65          for (int v = -k; v <= k; v++) {
66              const int ix = clamp(i + u, 0, imgWidth - 1);
67              const int iy = clamp(j + v, 0, imgHeight - 1);
68              const Color &col = img[imgWidth * iy + ix];
69              res += col * kernel[(v+k)*kSize + (u+k)];
70          }
71      }
72      return res;
73  }
```

Повече данни в кеша:

Можем ли да се възползваме от CACHE-а повече?

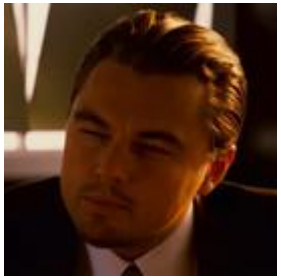
- Да! - С КОФИ!

- Вместо да разглеждаме всеки пиксел сам за себе си, ще групираме близките пиксели в малки квадратчета (buckets) и ще работим над тях.

- За да филтрираме даден пиксел се нуждаем от съседите му. Като приключим, шансът те да са все още в cache-а е голям, затова най-логично е да продължим да работим със съседите.

Още повече данни в кеша:

Паралелизъм в паралелизма?



- Single Instruction - Multiple Data (**SIMD**)

- **SIMD** е форма на паралелизъм, но в контекста на единствена нишка. Реализира се чрез специален хардуер в процесора, който е способен за един такт да изпълни една операция (Single Instruction) над пакет от входни данни (Multiple Data)
- Нашият class **Color** може да се възползва от тази функционалност, като по този начин операциите над R,G & B каналите ще се изпълняват паралелно.
- https://www.youtube.com/watch?v=ORaY6_rsTSQ
  "Ефикасна векторизация на C++ код" (Мартин Кръстев, Chaos Group, CG2 2015)

Тази хубава българска дума - СимДифицирам:

Остана ли хардуер, който все още не използваме?

- Време е да си изцапаме ръцете с GPU.

- В същността си, GPU-то е силно паралелизиран SIMD процесор, който досега през цялото време си почиваше. Защо не прехвърлим работата на него?
- Има цял клас задачи категоризирани като “*Embarrassingly Parallel Problems*” В този клас попадат доста от задачите в Image Processing.
  Такава задача е и Raytracing-а (#V-Ray GPU)
- Не е библейско да ги смятаме на CPU! *(*
- **CUDA API** - Интерфейс, предоставен от NVidia за GPGPU програмиране*
  - CUDA Toolit 8.0 (https://developer.nvidia.com/cuda-downloads)
  - NVidia NSight (http://www.nvidia.com/object/nsight.html)

* CUDA API е proprietary интерфейс за NVidia карти. Не е проложим за карти на други производители.
Отворени API-та за GPGPU са OpenCL и Vulkan, но тях няма да разглеждаме днес

Какво ще получим ако напишем програмата си на CUDA?

- Колко по-бързи очакваме да станем?

- Intel(R) Core™ i7-4720HQ @ 2.60GHz на теория може да изкара 76 GFLOPS
  - На практика нещата не стоят така и това число пада на ~30 GFLOPS
- NVidia GTX 960M на теория може да изкара 1400 GFLOPS
  - Не попаднах на benchmark за това, но дори както при CPU-то да паднат двойно, пак не е зле.

- Да вземем да проверим ?

Time’s up, let’s do this!

CUDA Runtime API : The Basics

- В началото беше #include <cuda_runtime.h>

- Менажиране на устройствата: *cudaGetDeviceCount*, *cudaSetDevice*, *cudaGetDeviceProperties*
- Трансфер на памет CPU<=>GPU: *cudaMalloc*, *cudaFree*, *cudaMemcpy*.
- Извикване на програма на GPU-то: *kernel<<<dim3, dim3>>>(args …)*;
- Синхронизация на изпълнението : *cudaDeviceSynchronize*
- http://docs.nvidia.com/cuda/cuda-runtime-api/ - Най-добрият ви приятел.
- Време е да пишем код!

Нещо математиката не излезе...

- Очаквахме около 700 GFLOPS, но получихме едва 450...

- Причини?
  - Достъпът до глобална памет е бавен!
    CUDA нишките седят и чакат паметта да дойде.
- Решение?
  - **Shared Memory** (Слушахте ли внимателно лекцията на Владо Вълчев?)

- С Shared Memory скоростта скача с ~60%!
  Това прави 720 GFLOPs!

Conclusion

- Няма тенденция процесорите да стават по-бързи, затова, за да сме продуктивни, трябва да сме паралелни!
- Някои задачи се решават по-бързо и по-лесно на GPU и е хубаво да можем да преценим кога има смисъл и кога не да прехвърлим сметките на графичната карта
- Не е много страшно да програмирате за GPU!

FIN*

*Ако лекцията ви е била интересна, всичко дискутирано тук и още много може да научите на курса HPC (High Performance Computing) , който Благовест Тасков (V-Ray GPU Teamlead) води във ФМИ

Thank you!

Q & A

https://github.com/a7az0th/CG2-CodeForArt-2016

alexander.soklev@chaosgroup.com