Краткая история GPU и введение в OpenCL

Вычисления на видеокартах. Лекция 1

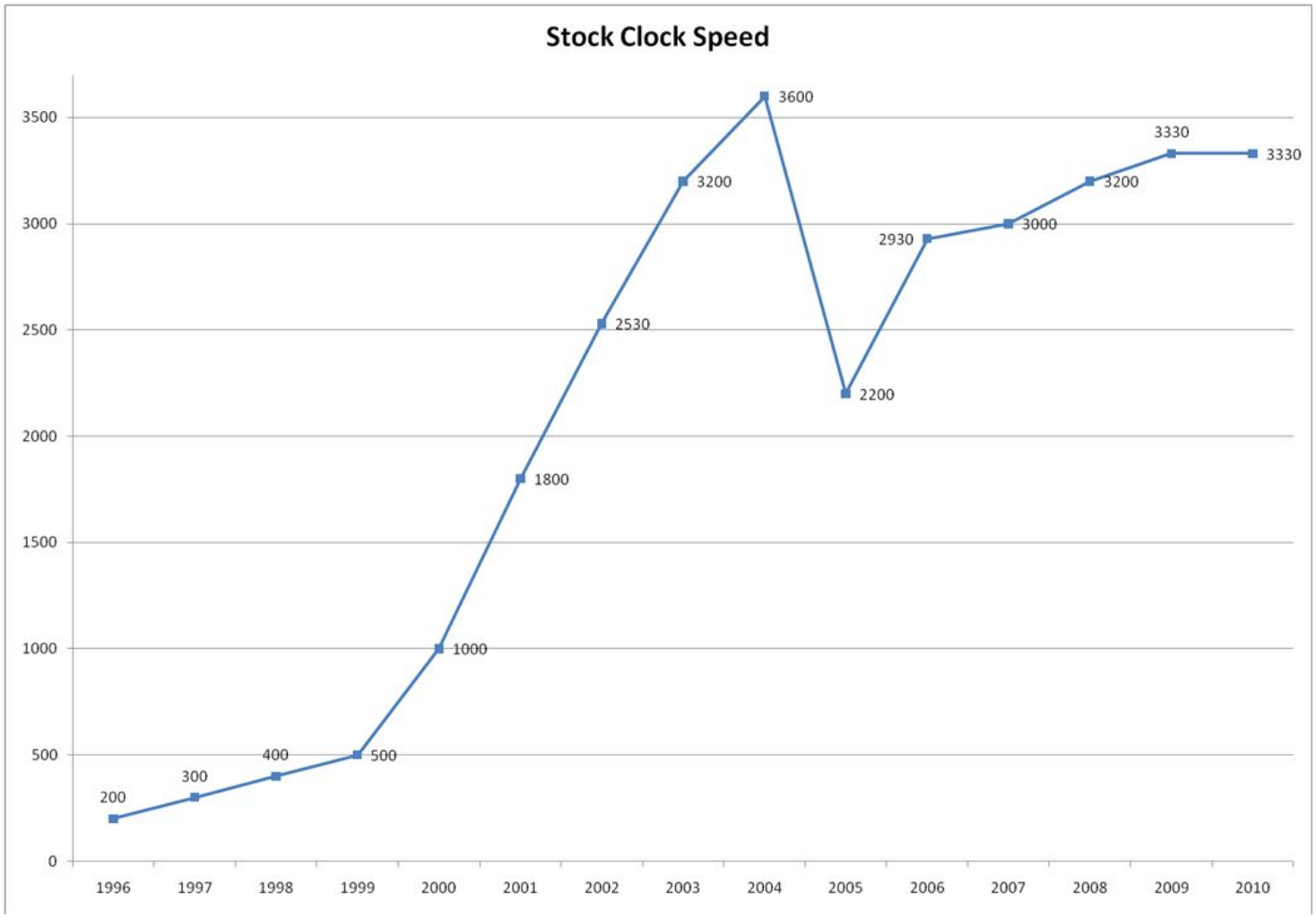
Полярный Николай

polarnick239@gmail.com

Процессоры

Долго процессоры были одноядерными, и тактовая частота была их основной характеристикой.

В 2004 году рост частоты остановился на 3.6 ГГц (Pentium 4 Prescott).


Stock Clock Speed
3500
3000
2500
2000
1500
1000
500
0
200
300
400
500
1000
1800
2530
3200
3600
2200
2930
3000
3200
3330
3330
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010

Процессоры

1. **Out-of-order** исполнение позволяет начинать заранее делать будущие команды (**Instruction-level parallelism**).

Процессоры

1. **Out-of-order** исполнение позволяет начинать заранее делать будущие команды (**Instruction-level parallelism**).
2. Для улучшения ILP в случае ветвления кода существует **branch-prediction.**

Процессоры

1. **Out-of-order** исполнение позволяет начинать заранее делать будущие команды (**Instruction-level parallelism**).
2. Для улучшения ILP в случае ветвления кода существует **branch-prediction.**
3. Часто узкое место не вычисления, а подгрузка данных из оперативной памяти, поэтому существуют **L1/L2/L3 кеши.**

Процессоры

1. **Out-of-order** исполнение позволяет начинать заранее делать будущие команды (**Instruction-level parallelism**).
2. Для улучшения ILP в случае ветвления кода существует **branch-prediction.**
3. Часто узкое место не вычисления, а подгрузка данных из оперативной памяти, поэтому существуют **L1/L2/L3 кеши.**
4. Скрывать задержки обращения к памяти помогает так же **Hyper-threading** и **Simultaneous multithreading.**

Процессоры

1. **Out-of-order** исполнение позволяет начинать заранее делать будущие команды (**Instruction-level parallelism**).
2. Для улучшения ILP в случае ветвления кода существует **branch-prediction.**
3. Часто узкое место не вычисления, а подгрузка данных из оперативной памяти, поэтому существуют **L1/L2/L3 кеши.**
4. Скрывать задержки обращения к памяти помогает так же **Hyper-threading** и **Simultaneous multithreading.**
5. **Многоядерность**. В не-серверном компьютере выросла до 4 ядер в 2008 году. В 2017 году - до 8-16 ядер.

Процессоры: **S**ingle **i**nstruction, **m**ultiple **d**ata

- 1997, **MMX**: 64 бита, целочисленные операции, **8**x**char**, **4**x**short**, **2**x**int**

- 1999, **SSE**: 128 бит, floating-point операции, **4**x**float**

- 2000, **SSE2**: 128 бит, теперь и над целыми, **16**x**char**, **8**x**short**, **4**x**int**, **2**x**long**, **4**x**float**, **2**x**double**

- 2004-2006, **SSE3** / **SSSE3** / **SSE4** / **SSE4.1** / **SSE4.2** - расширения набора инструкций (горизонтальная работа с регистрами, специальные функции для видеокодирования и обработки строк)

Процессоры: **S**ingle **i**nstruction, **m**ultiple **d**ata

- 2011, **AVX**: 256 бит, floating-point операции, **8**x**float**, **4**x**double**

- 2013, **AVX2**: 256 бит, теперь и над целыми, **32**x**char**, **16**x**short**, **8**x**int**, **4**x**long**, **8**x**float**, **4**x**double**

- 2017, **AVX-512**: 512 бит, много подрасширений, мало где доступно, **16**x**float**, **8**x**double**

Множество Мандельброта без SIMD интринсик

```
float x = 0.0f;
float y = 0.0f;
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    float xn = x * x - y * y + x0;
    y = 2 * x * y + y0;
    x = xn;
    if (x * x + y * y > INFINITY) {
        break;
    }
}
```

Множество Мандельброта с SSE, SSE2, SSSE3

```
__m128i maskAll = _mm_setzero_si128();
__m128i iters = _mm_setzero_si128();
__m128 xs = _mm_set1_ps(0.0f);
__m128 ys = _mm_set1_ps(0.0f);
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    __m128 xsn = _mm_add_ps(_mm_sub_ps(_mm_mul_ps(xs, xs), _mm_mul_ps(ys, ys)), xs0);
    __m128 ysn = _mm_add_ps(_mm_mul_ps(_mm_mul_ps(_mm_set1_ps(2.0f), xs), ys), _mm_set1_ps(y0));
    xs = _mm_add_ps(_mm_andnot_ps((__m128) maskAll, xsn), _mm_and_ps((__m128) maskAll, xs));
    ys = _mm_add_ps(_mm_andnot_ps((__m128) maskAll, ysn), _mm_and_ps((__m128) maskAll, ys));

    maskAll = _mm_or_si128(_mm_castps_si128(_mm_cmpge_ps(_mm_add_ps(_mm_mul_ps(xs, xs), _mm_mul_ps(ys, ys)
    iters = _mm_add_epi16(iters, _mm_andnot_si128(maskAll, _mm_set1_epi16(1)));
    int mask = _mm_movemask_epi8(maskAll);
    if (mask == 0xffff) {
        break;
    }
}
iters = _mm_shuffle_epi8(iters, _mm_setr_epi8(12, 13, 8, 9, 4, 5, 0, 1, -1, -1, -1, -1, -1, -1, -1, -1));
```

Множество Мандельброта с AVX и AVX2

```
__m256i maskAll = _mm256_setzero_si256();
__m256i iters = _mm256_setzero_si256();
__m256 xs = _mm256_set1_ps(0.0f);
__m256 ys = _mm256_set1_ps(0.0f);
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    __m256 xsn = _mm256_add_ps(_mm256_sub_ps(_mm256_mul_ps(xs, xs), _mm256_mul_ps(ys, ys)), xs0);
    __m256 ysn = _mm256_add_ps(_mm256_mul_ps(_mm256_mul_ps(_mm256_set1_ps(2.0f), xs), ys), _mm256_set1_ps
    xs = _mm256_add_ps(_mm256_andnot_ps((__m256) maskAll, xsn), _mm256_and_ps((__m256) maskAll, xs));
    ys = _mm256_add_ps(_mm256_andnot_ps((__m256) maskAll, ysn), _mm256_and_ps((__m256) maskAll, ys));

    maskAll = (__m256i) _mm256_or_ps(_mm256_cmp_ps(_mm256_add_ps(_mm256_mul_ps(xs, xs), _mm256_mul_ps(ys,
    iters = _mm256_add_epi16(iters, _mm256_andnot_si256(maskAll, _mm256_set1_epi16(1)));
    int mask = _mm256_movemask_epi8(maskAll);
    if (mask == (int) 0xffffffff) {
        break;
    }
}
iters = _mm256_shuffle_epi8(iters, _mm256_setr_epi8(0, 1, -1, -1, 4, 5, -1, -1, 8, 9, -1, -1, 12, 13, -1,
```

Software 3D

Вплоть **до 1996** года даже игры вроде DOOM, Quake и Duke Nukem 3D обсчитывали визуализацию 3D пространства на процессоре. Процессор справлялся в первую очередь благодаря низкому расширению экрана 320x200 и относительно простой графике.

Специализированные 3D-ускорители

Около **1995-1997** начали набирать популярность специализированные 3D-ускорители (такие как S3 ViRGE, ATI 3D Rage, 3dfx Voodoo) способные отрисовывать более сложную графику гораздо эффективнее.

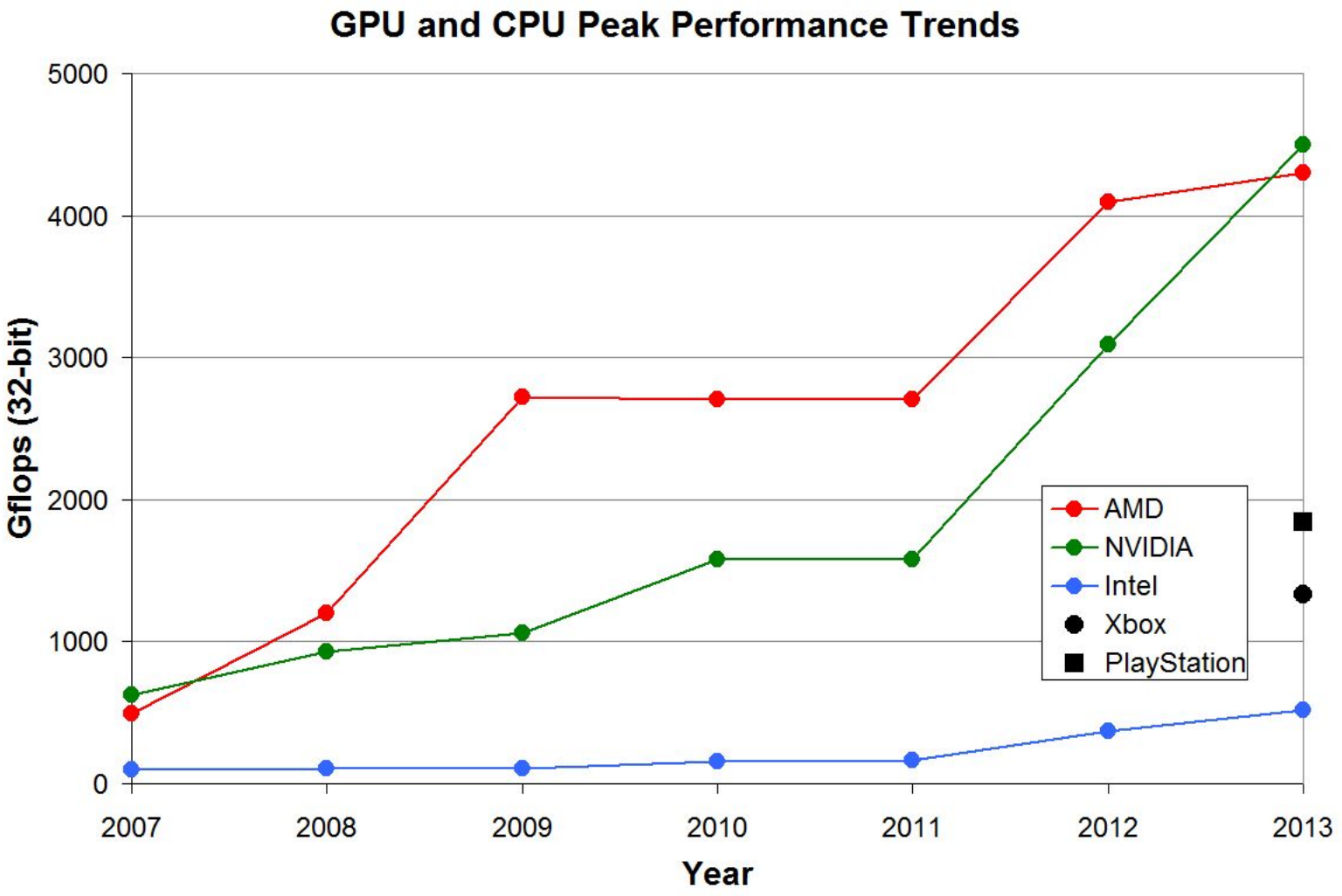
3D-ускорители: программируемые шейдеры

Спецэффекты в играх становились все сложнее и стало невозможно продолжать предлагать программистам лишь жестко ограниченный набор 3D функциональности, возникла потребность дать гибкий инструмент для программирования видеокарты. Так появились программируемые шейдеры.

Стало возможно в каждом пикселе посчитать произвольную математику, и это позволило реализовать много интересных спецэффектов (таких как реалистичная вода, сложные модели освещения и т.п.).

Зарождение GPGPU

Благодаря популярности игр, архитектурным особенностям видеокарт (задача отрисовки обладает массовым параллелизмом, т.к. пиксели на экране можно обсчитывать независимо) и все возрастающей сложности игровой графики производительность видеокарт росла гораздо быстрее процессоров:


GPU and CPU Peak Performance Trends
Gflops (32-bit)
5000
4000
3000
2000
1000
0
2007
2008
2009
2010
2011
2012
2013
Year
AMD
NVIDIA
Intel
Xbox
PlayStation

Зарождение GPGPU

Поэтому видеокарты специализирурующиеся на отображении 3D-графики начали использовать и для других задач, таких как обсчет физики и моделирования сложных процессов.

Это стало возможно благодаря гибкости программируемых шейдеров, но тем не менее из-за специализации на 3D-графике гибкость шейдеров была ограничена, и для general-purpose задач возникла потребность нового, еще более гибкого API.

Появление GPGPU API

Стали появляться API лучше подходящие для general-purpose вычислений:

- 2006, **Close to Metal** - поддерживался только на видеокартах от **AMD**, позже был заменен на **OpenCL**

- 2007, **CUDA** - поддерживается только на видеокартах от **NVidia**

- 2009, **OpenCL** - открытый стандарт поддерживающийся практически везде

Появление GPGPU API

Так же позже появилось несколько узко специализированных API:

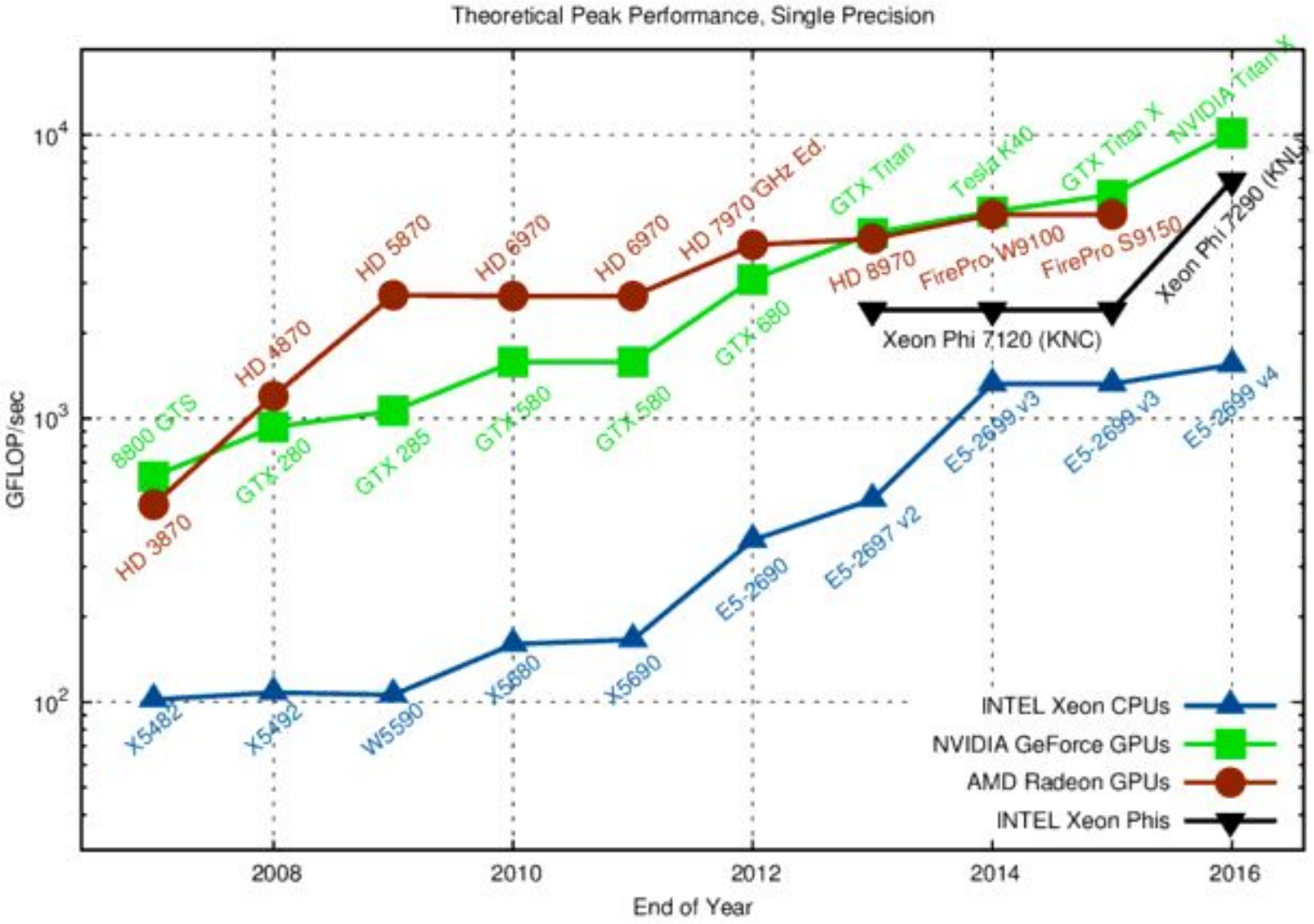
- 2011, RenderScript - специально для Android, изначально ставил целью заменить OpenGL, с 2013 года стал пытаться заменить OpenCL

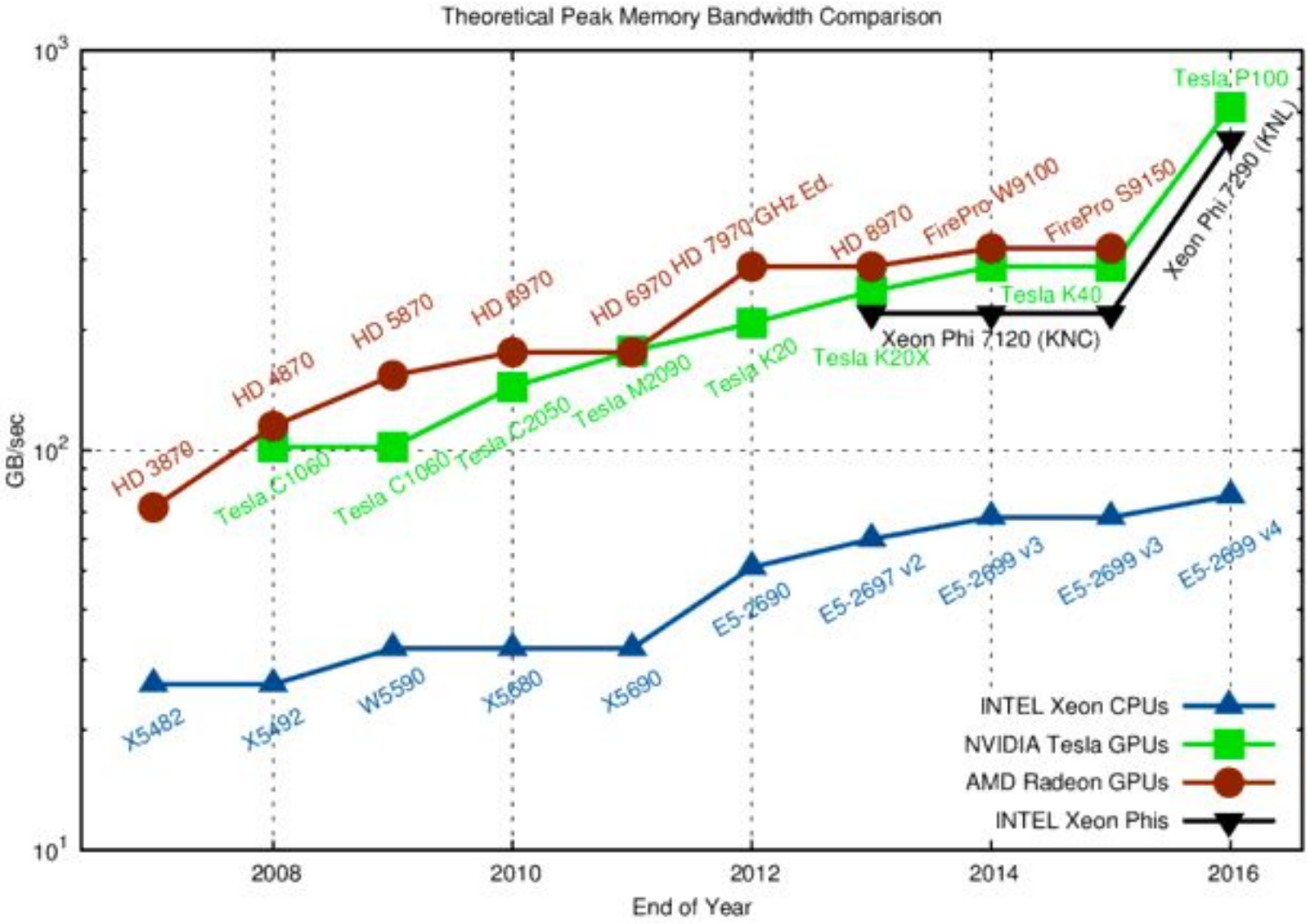
- 2014, Metal - специально для устройств Apple, и графика, и вычисления

В результате на сегодняшний день для вычислений общего назначения обладающих массовым параллелизмом на видеокартах существуют удобные API, и реализовать потенциал видеокарты с их помощью гораздо проще чем теоретическую мощность процессора с помощью SIMD.

Производительность GPU

Производительность видеокарт же на порядок выше по обоим критическим показателям - вычислительная мощность и пропускная способность видеопамяти:


Theoretical Peak Performance, Single Precision
GFLOP/sec
End of Year
2008
2010
2012
2014
2016
HD 3870
HD 4870
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
8800 GTS
GTX 280
GTX 285
GTX 580
GTX 580
GTX 680
GTX Titan
Tesla K40
GTX Titan X
NVIDIA Titan X
Xeon Phi 7120 (KNC)
Xeon Phi 7290 (KNL)
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
INTEL Xeon CPUs
NVIDIA GeForce GPUs
AMD Radeon GPUs
INTEL Xeon Phis

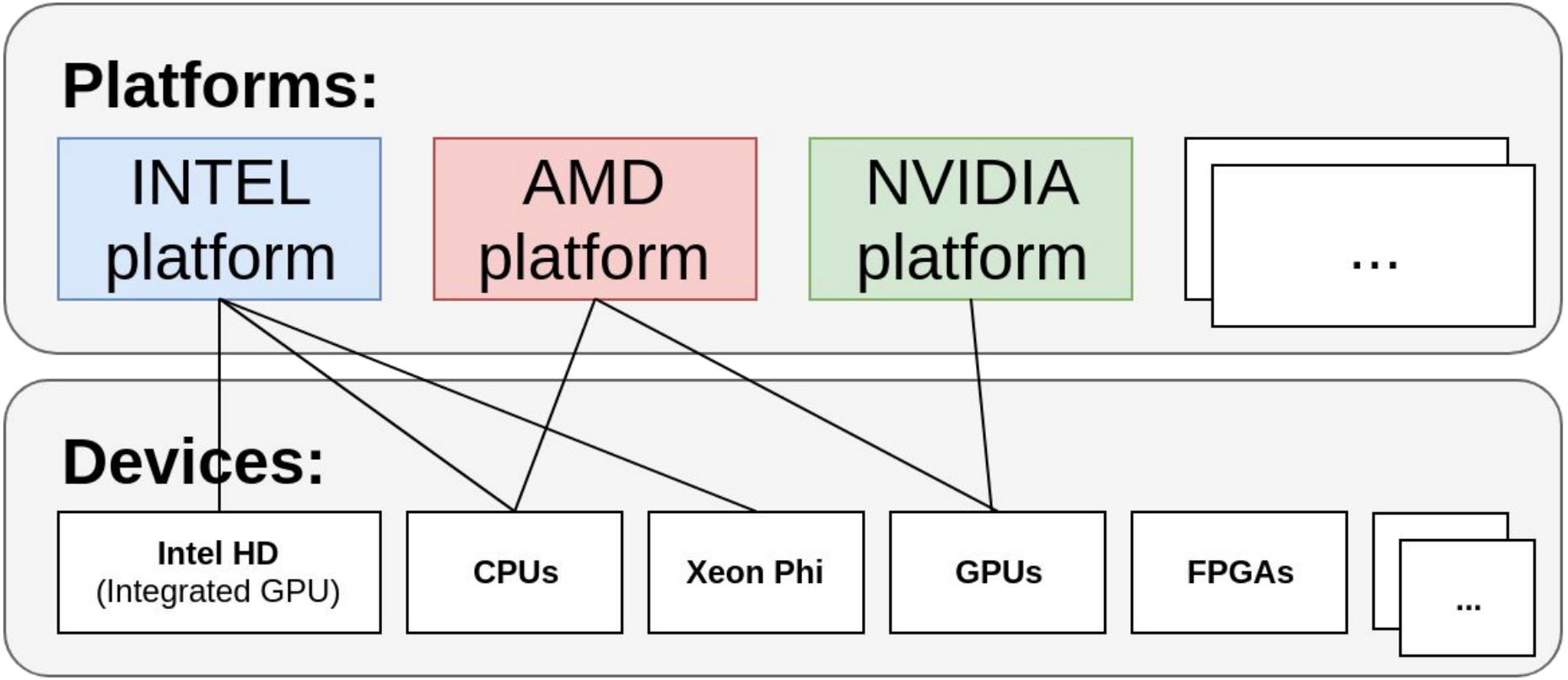

Theoretical Peak Memory Bandwidth Comparison
GB/sec
End of Year
2008
2010
2012
2014
2016
HD 3870
HD 4870
HD 5870
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
Tesla C1060
Tesla C1060
Tesla C2050
Tesla M2090
Tesla K20
Tesla K20X
Tesla K40
Tesla P100
Xeon Phi 7120 (KNC)
Xeon Phi 7290 (KNL)
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
INTEL Xeon CPUs
NVIDIA Tesla GPUs
AMD Radeon GPUs
INTEL Xeon Phis

Мотивация

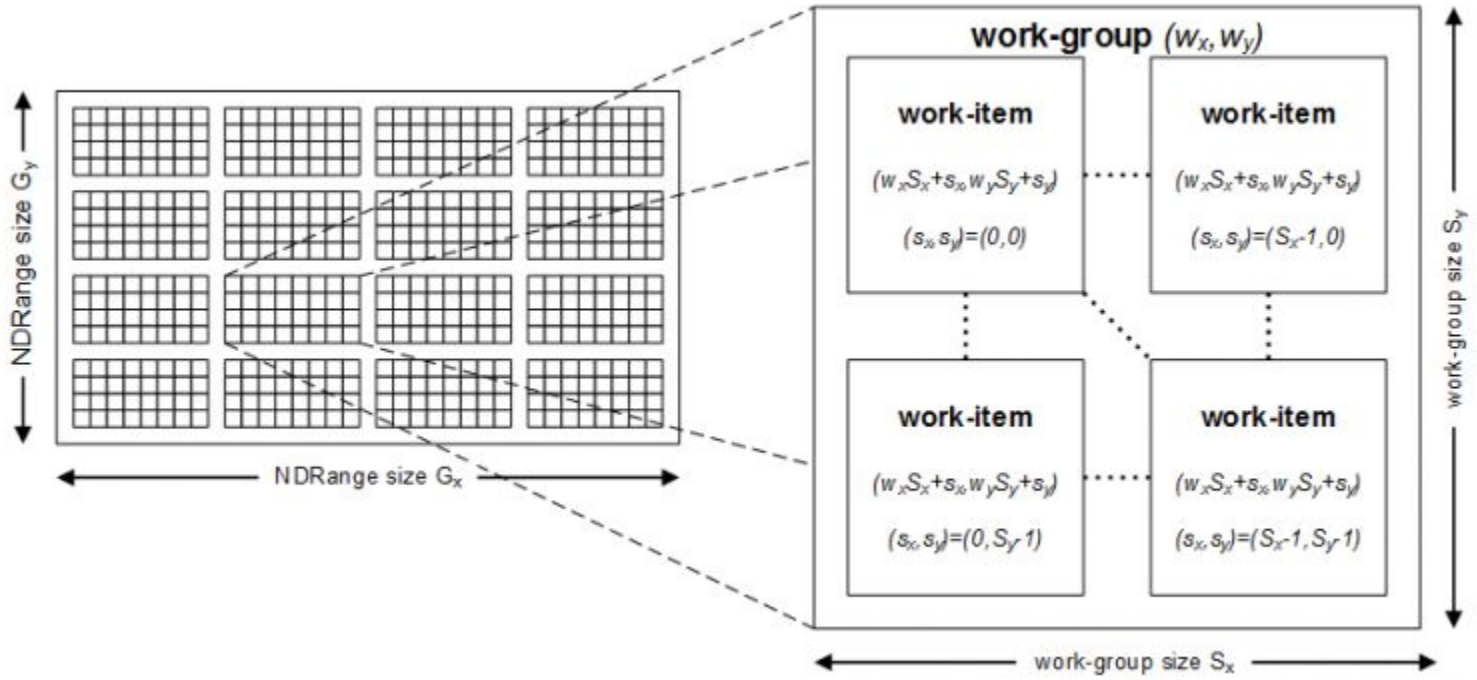
Цель курса:

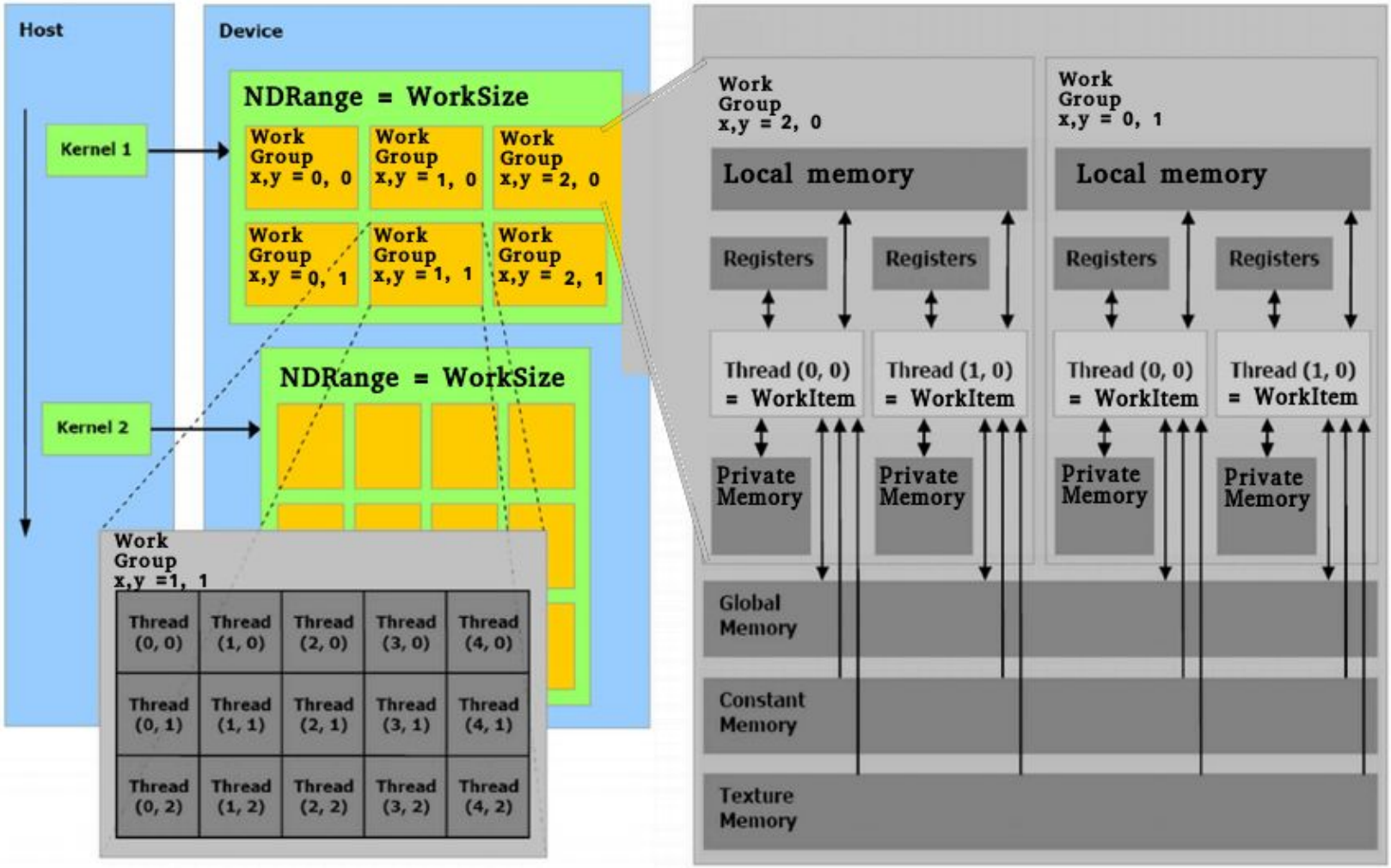
- Понять что такое **массовый параллелизм** и как спроектировать алгоритм с его учетом
- Научиться использовать **OpenCL**
- Понять почему видеокарты хорошо подходят только для задач обладающих **массовым параллелизмом**
- Понять общие свойства архитектуры видеокарт и тем самым научиться оптимизировать алгоритмы
- Когда-нибудь потом, делая исследовательский проект, внезапно понять что вы можете за один вечер ускорить вычисления в десятки раз, что существенно увеличит скорость экспериментирования

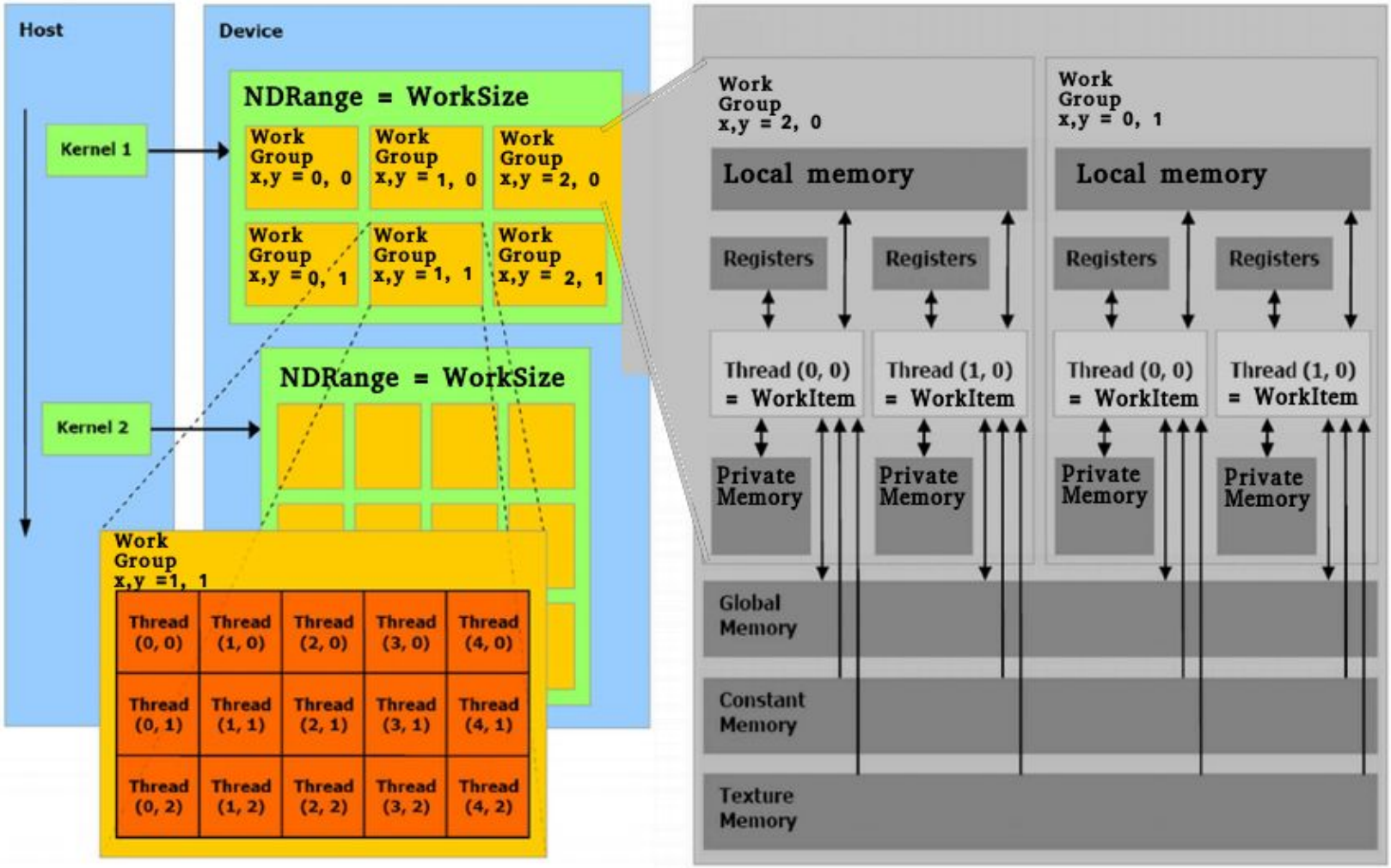
OpenCL платформы и устройства

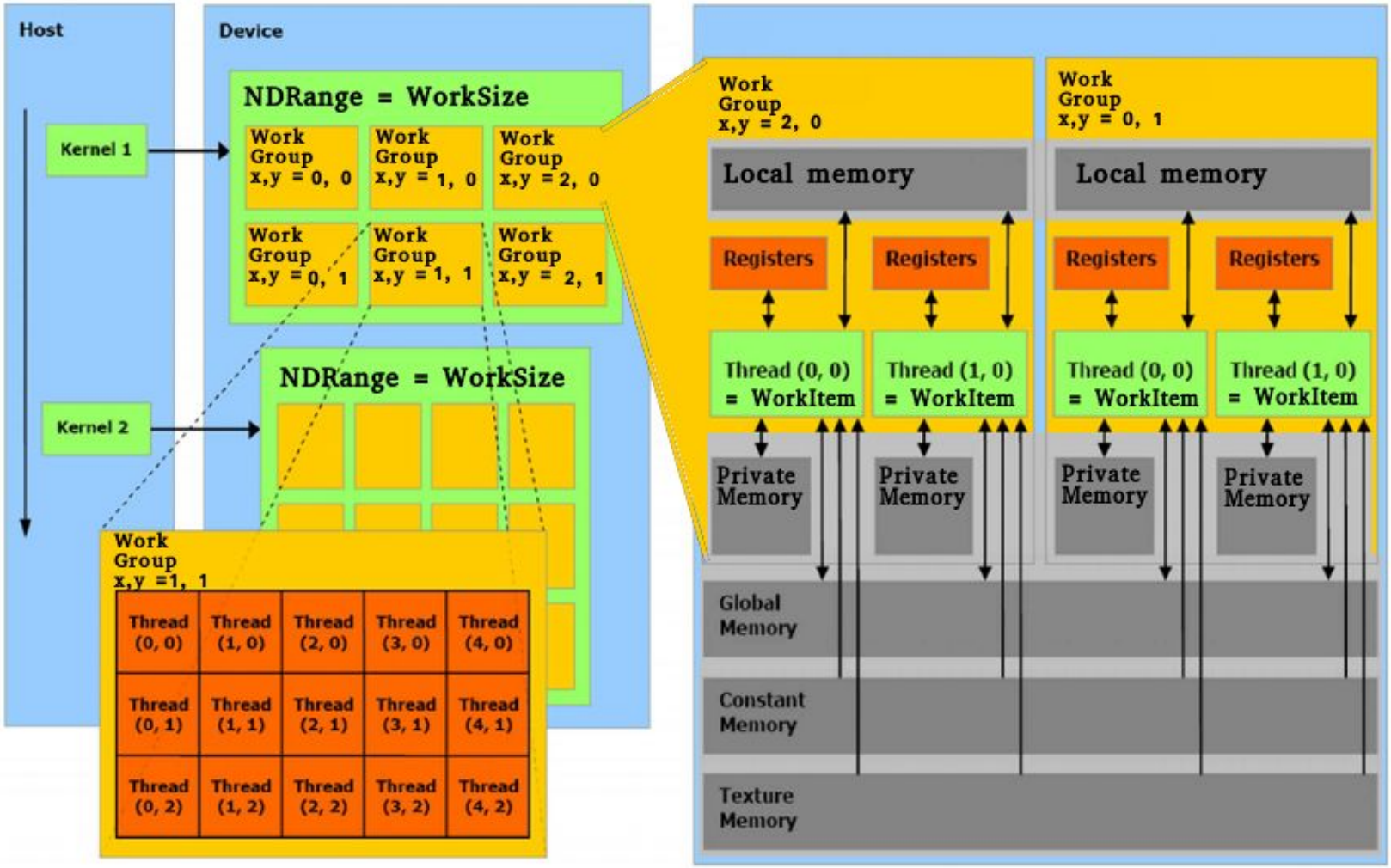


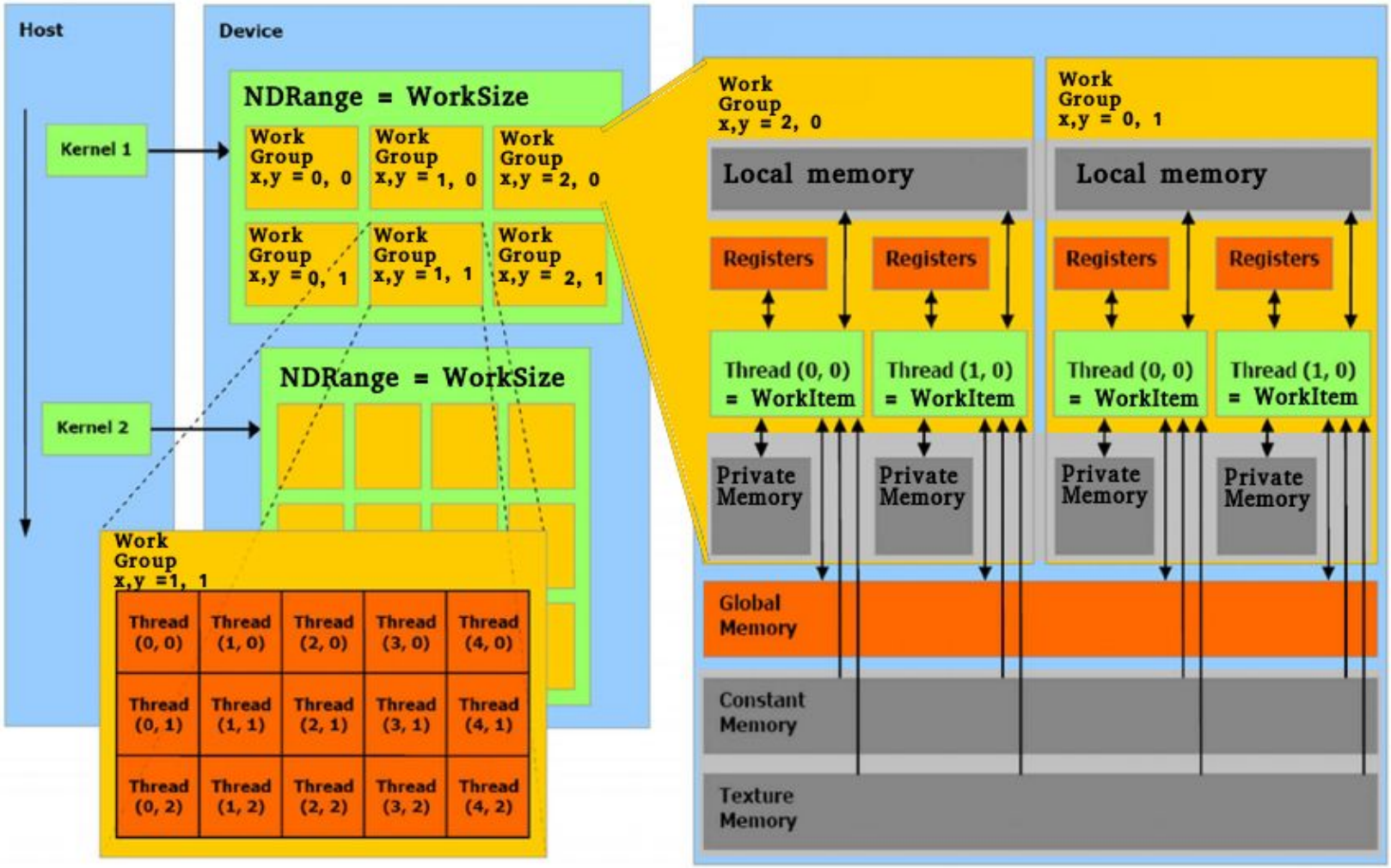
Модель вычислений

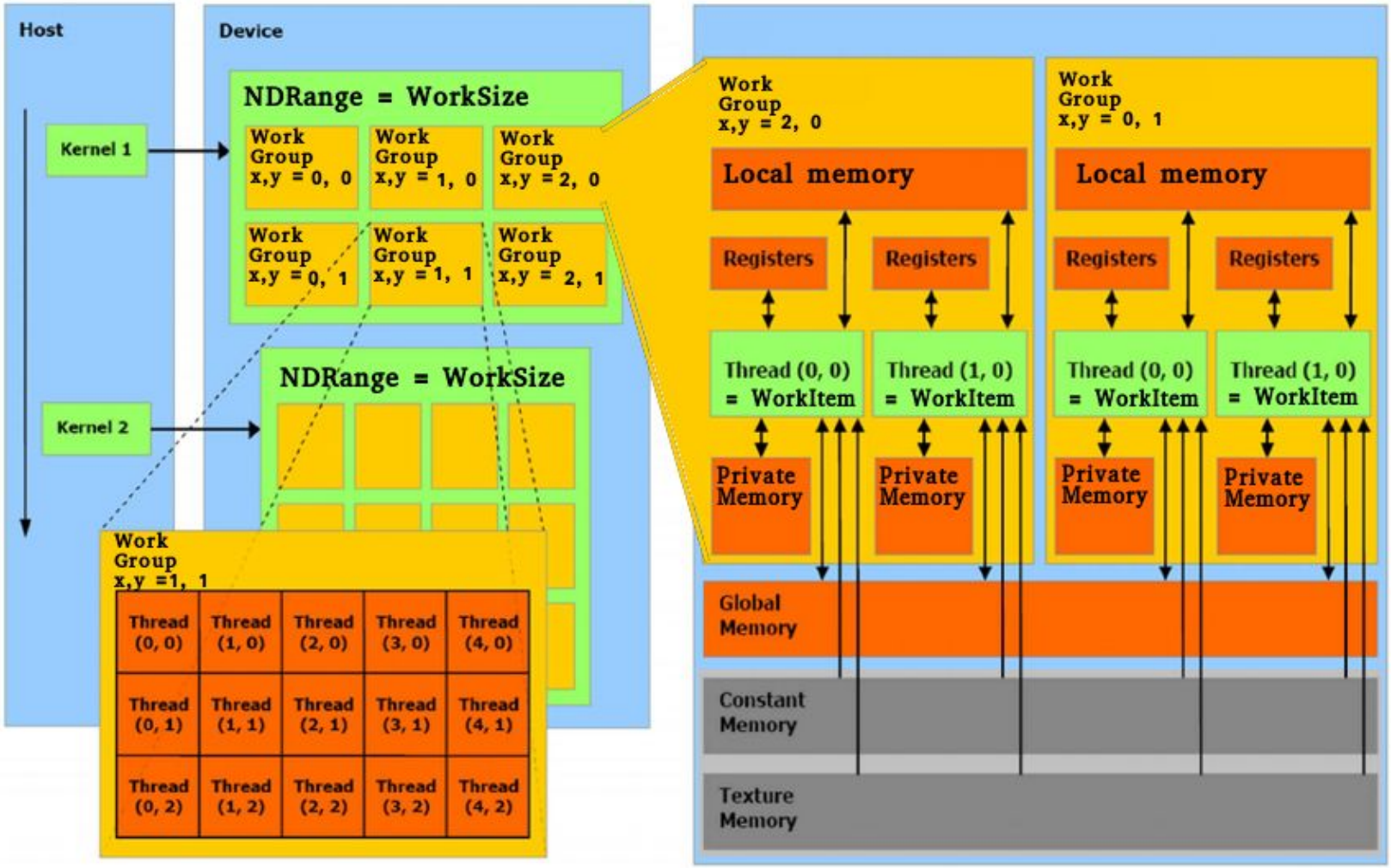


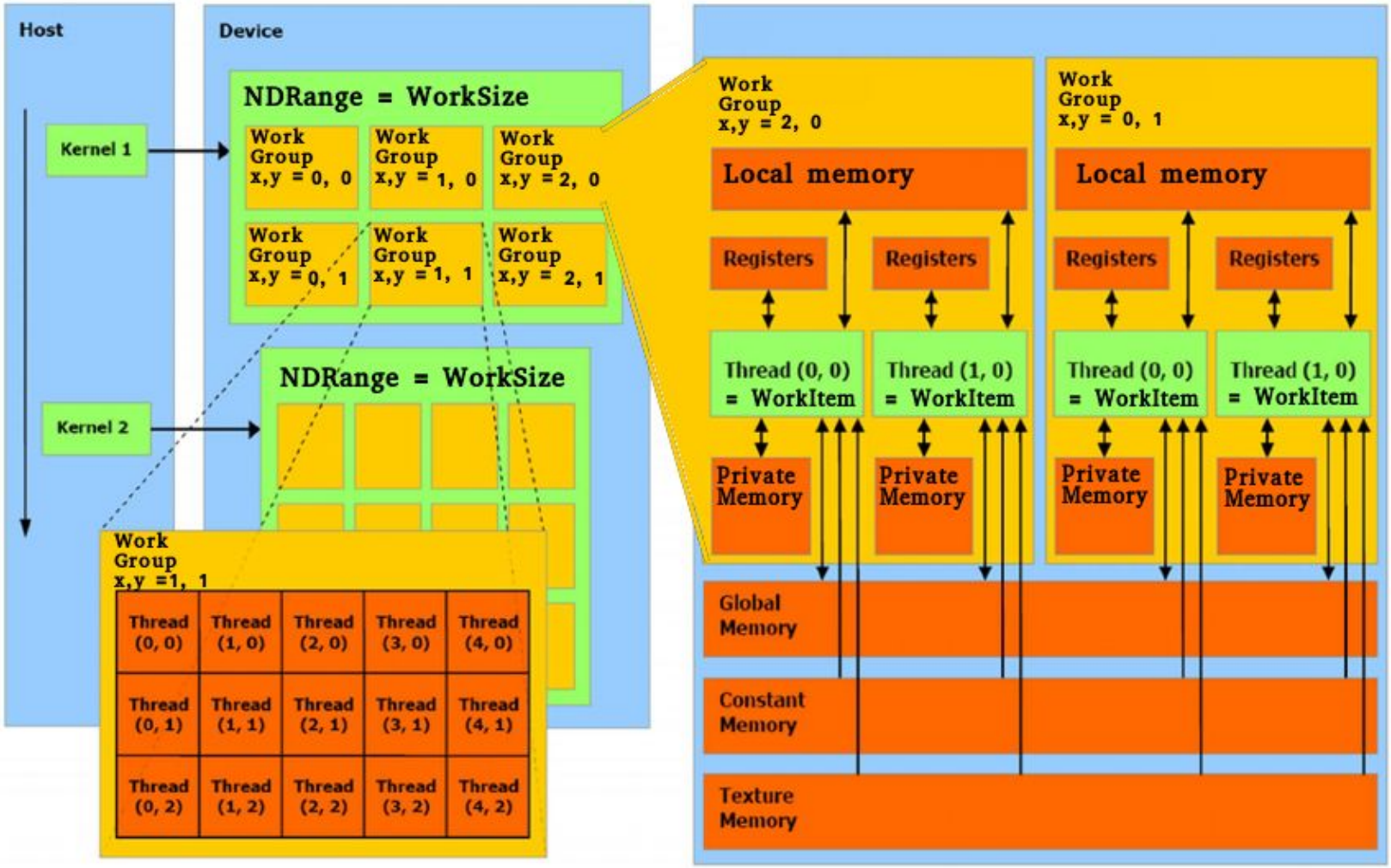

Host
Device
NDRange = WorkSize
Kernel 1
Work Group x,y = 0, 0
Work Group x,y = 1, 0
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Work Group x,y = 1, 1
Work Group x,y = 2, 1
NDRange = WorkSize
Kernel 2
Work Group x,y =1, 1
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Local memory
Registers
Thread (0, 0) = WorkItem
Thread (1, 0) = WorkItem
Private Memory
Global Memory
Constant Memory
Texture Memory


Host
Device
NDRange = WorkSize
Kernel 1
Work Group x,y = 0, 0
Work Group x,y = 1, 0
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Work Group x,y = 1, 1
Work Group x,y = 2, 1
NDRange = WorkSize
Kernel 2
Work Group x,y =1, 1
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Local memory
Registers
Thread (0, 0) = WorkItem
Thread (1, 0) = WorkItem
Private Memory
Global Memory
Constant Memory
Texture Memory


Host
Device
NDRange = WorkSize
Kernel 1
Work Group x,y = 0, 0
Work Group x,y = 1, 0
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Work Group x,y = 1, 1
Work Group x,y = 2, 1
NDRange = WorkSize
Kernel 2
Work Group x,y =1, 1
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Local memory
Registers
Thread (0, 0) = WorkItem
Thread (1, 0) = WorkItem
Private Memory
Global Memory
Constant Memory
Texture Memory


Host
Device
NDRange = WorkSize
Kernel 1
Work Group x,y = 0, 0
Work Group x,y = 1, 0
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Work Group x,y = 1, 1
Work Group x,y = 2, 1
NDRange = WorkSize
Kernel 2
Work Group x,y =1, 1
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Local memory
Registers
Thread (0, 0) = WorkItem
Thread (1, 0) = WorkItem
Private Memory
Global Memory
Constant Memory
Texture Memory


Host
Device
NDRange = WorkSize
Kernel 1
Work Group x,y = 0, 0
Work Group x,y = 1, 0
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Work Group x,y = 1, 1
Work Group x,y = 2, 1
NDRange = WorkSize
Kernel 2
Work Group x,y =1, 1
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Local memory
Registers
Thread (0, 0) = WorkItem
Thread (1, 0) = WorkItem
Private Memory
Global Memory
Constant Memory
Texture Memory


Host
Device
NDRange = WorkSize
Kernel 1
Work Group x,y = 0, 0
Work Group x,y = 1, 0
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Work Group x,y = 1, 1
Work Group x,y = 2, 1
NDRange = WorkSize
Kernel 2
Work Group x,y =1, 1
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)
Work Group x,y = 2, 0
Work Group x,y = 0, 1
Local memory
Registers
Thread (0, 0) = WorkItem
Thread (1, 0) = WorkItem
Private Memory
Global Memory
Constant Memory
Texture Memory