

# Įvadas į skaičiavimus su GPU

Medžiagos šaltiniai:

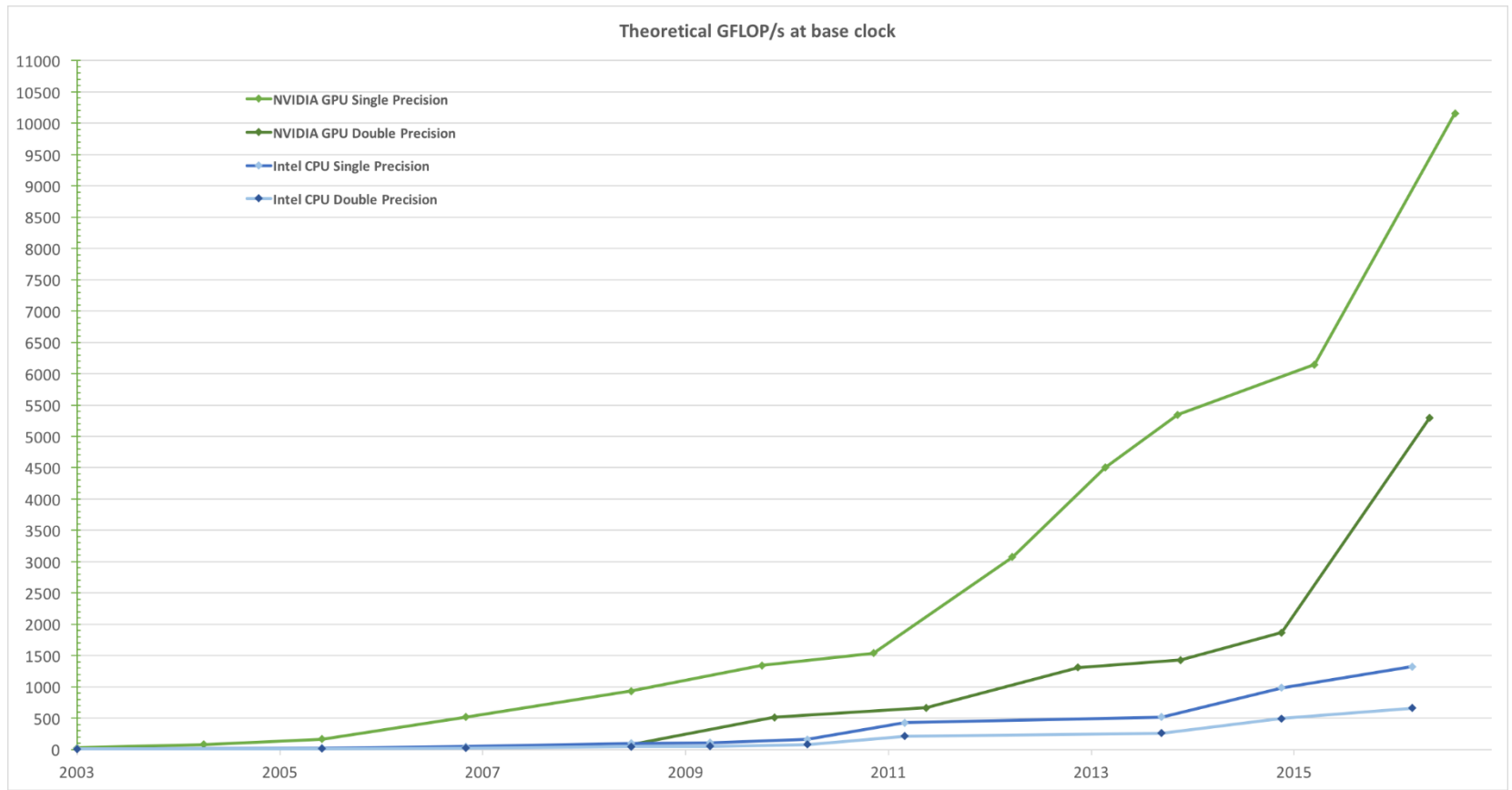
[www.nvidia.com](http://www.nvidia.com/docs.nvidia.com/cuda/cuda-c-programming-guide/)  
[docs.nvidia.com/cuda/cuda-c-programming-guide/](http://docs.nvidia.com/cuda/cuda-c-programming-guide/)

# GPU

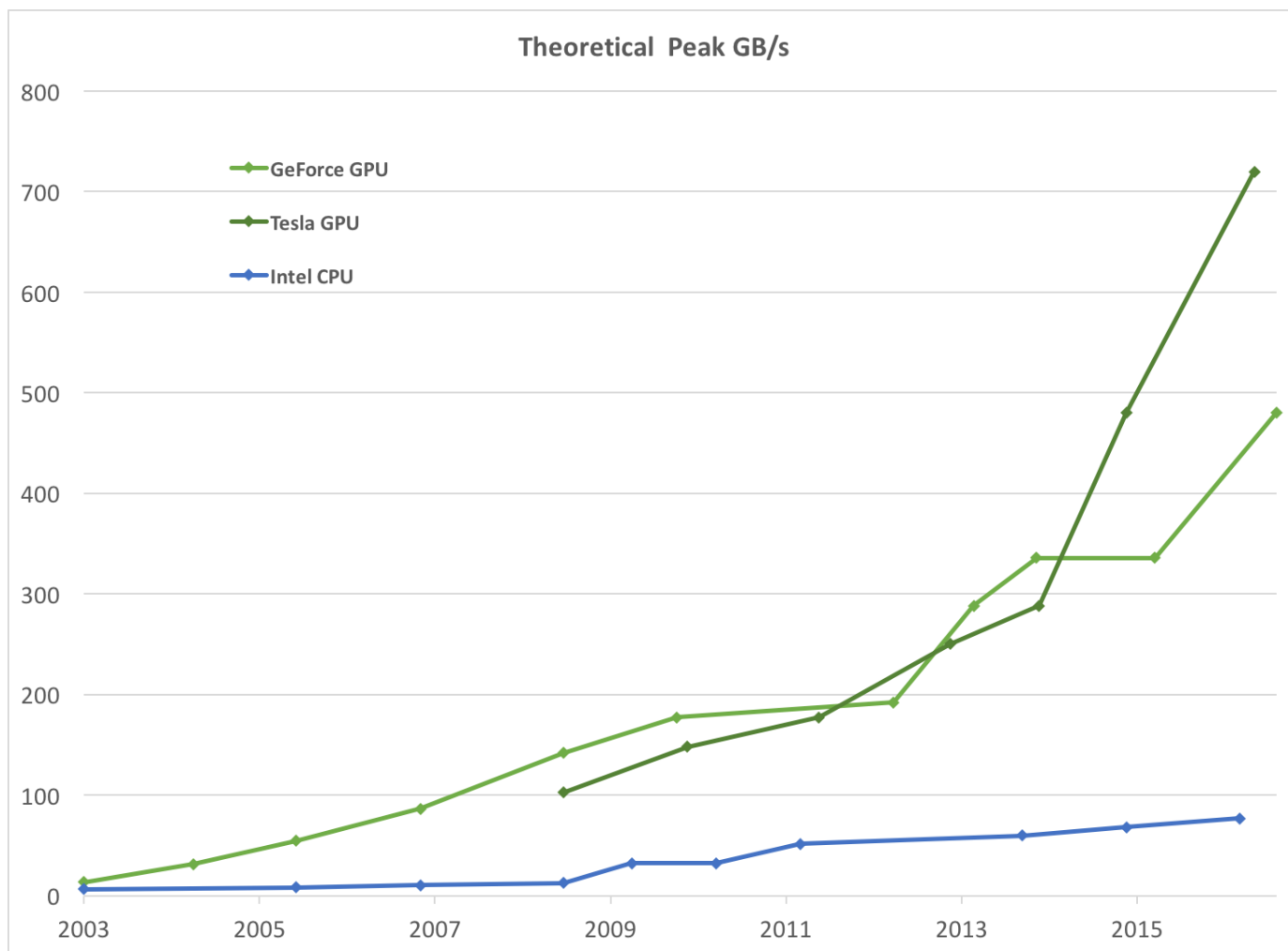
GPU – angl. Graphical processing unit, skaičiavimų įrenginys, kuris pradžioje buvo skirtas grafikai skirtų skaičiavimų apdorojimui.

GPGPU – angl. General-purpose computing on GPUs, GPU panaudojimas bet kurių (nebūtinai grafikos apdorojimo) skaičiavimų atlikimui.

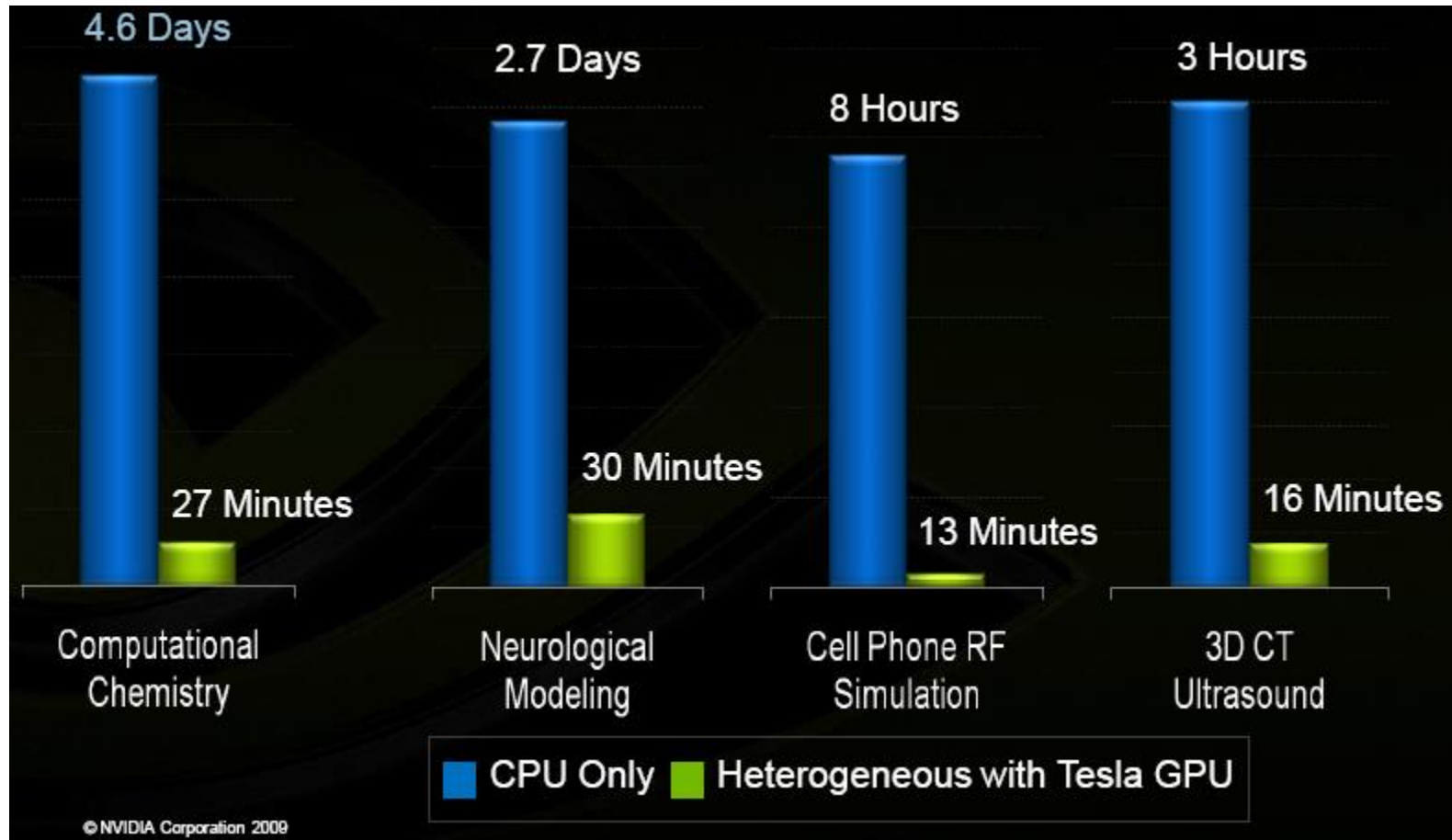
# CPU ir GPU FLOPS



# Specializuotų GPU atminties pralaidumas



# CPU v/s GPU, pritaikymo pavyzdžių palyginimas (2009 m.)



# GPGPU

- Kaip to pasiekti programuotojui?
  - CUDA
  - OpenCL
  - DirectCompute

Tačiau pradėti reikia nuo architektūros ...

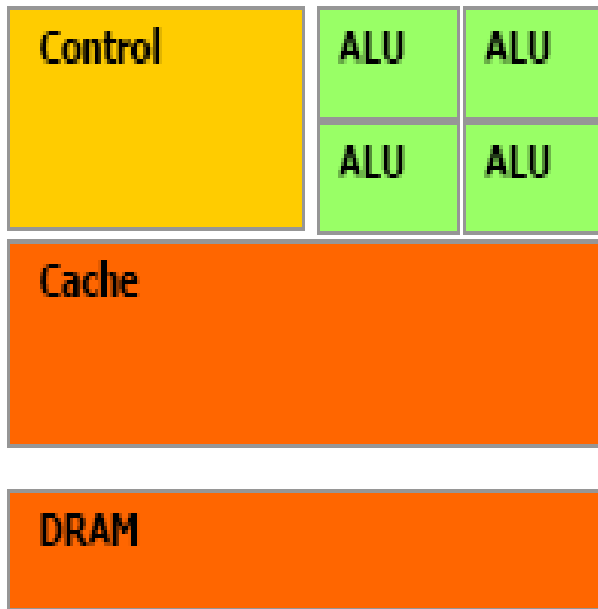
# GPU įrenginio architektūra

Skaičiavimų atžvilgiu, GPU (ang. graphics processing unit) - tai lygiagretusis skaičiavimų įrenginys.

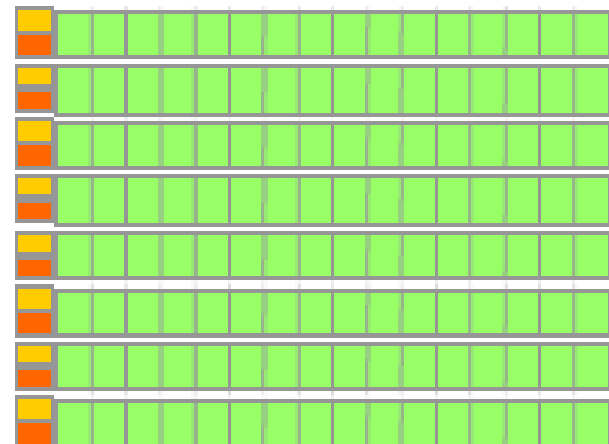
Išskirkime pagrindinius aspektus, į kuriuos reikia atkreipti dėmesį, kai į GPU žiūrima kaip į skaičiavimų įrenginį:

1. Skaičiavimų vienetai (ALU)
2. Atminties struktūra

# CPU ir GPU architektūrų supaprastintas palyginimas



CPU



DRAM

GPU



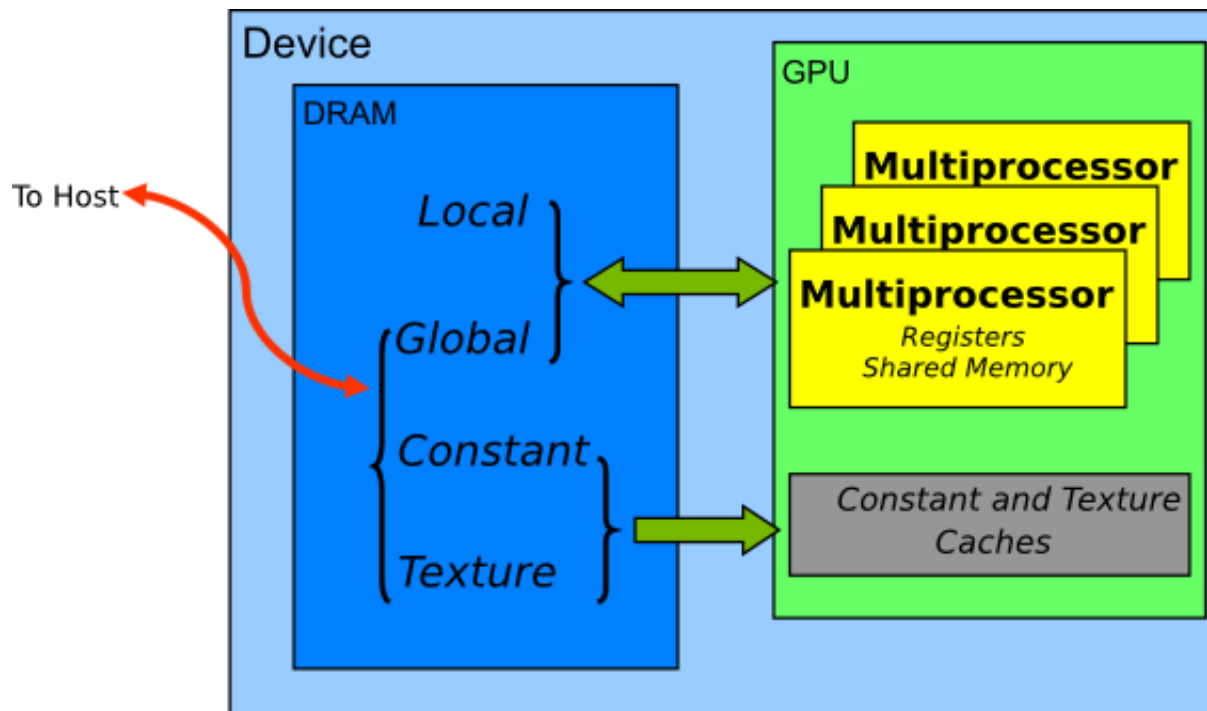
# Skaičiavimų vienetai (ALU)

1. GPU susidaro iš  $N_{SM}$  multiprocesorių (toliau SM, angl. streaming multiprocessor)
2. Vienas multiprocesorius turi  $N_m$  branduolių, sudarytų iš skaičiavimų modulių. Moduliai gali būti skirtingi - tai priklauso nuo konkrečios plokštės GPU architektūros.
3. Visi skaičiavimai ir duomenų siuntimai yra atliekami porcijomis, specialiais gijų apdorojimo vienetais, kurie operuoja gijų porcijomis (angl. warp).
4. Programuotojo užduotis – parašyti kodą taip, kad gijų porcijos veiktų efektyviai, remiantis jų veikimo principais ir NVIDIA rekomendacijomis.

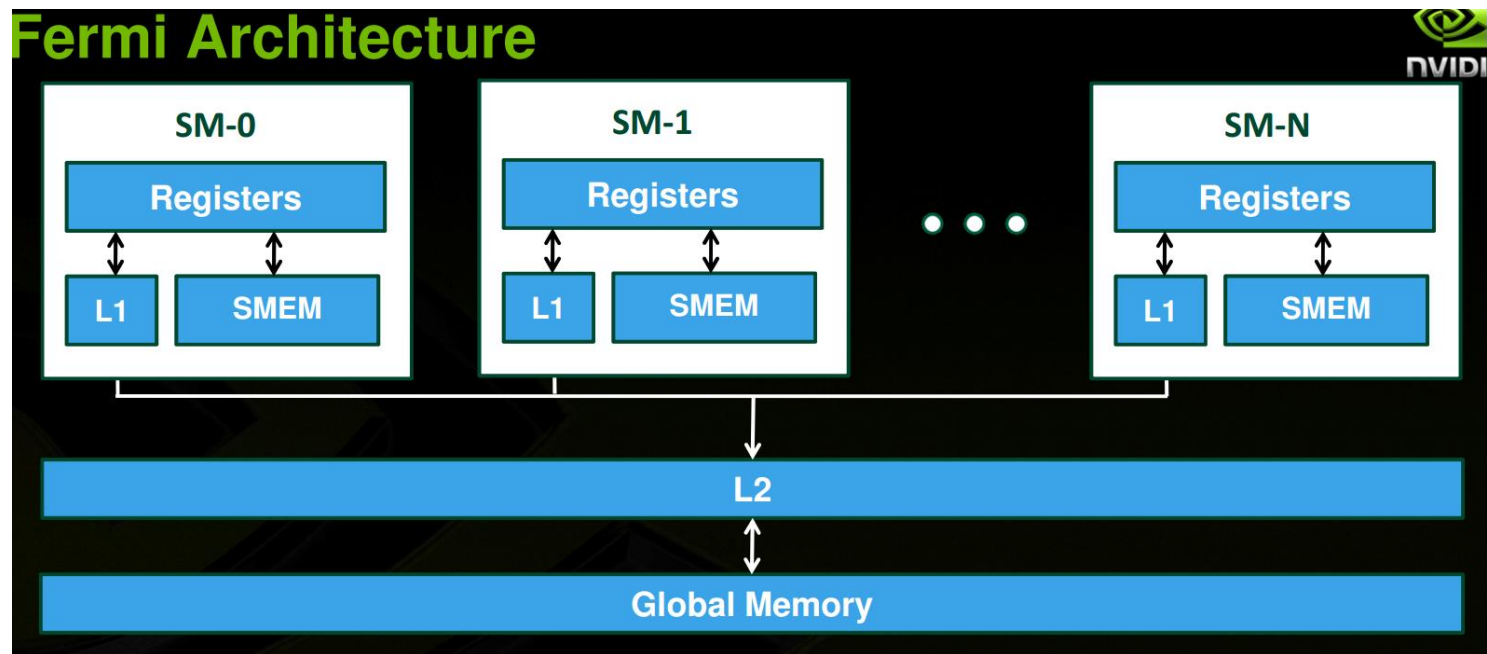
# Atminties struktūra

1. GPU turi pagrindinę atmintį duomenims talpinti, paprastai tai yra vieno gigabaito eilės dydžio atmintis. Iš jos skaityti ir į ją rašyti gali visi CUDA branduoliai lygiagrečiai. GPU pagrindinė atmintis išskaidyta į segmentus, jų dydis priklauso nuo GPU atminties magistralės dydžio.
2. Pagrindinė atmintis naudojama globaliajai ir lokaliajai atmintims saugoti. Jos skiriasi tik tuo, kad lokaloji yra rezervuota tam tikroms gijoms o iš globaliosios gali skaityti ir GPU ir CPU (kopijuojant).
3. Kiekvienas GPU multiprocesorius turi greitąją atmintį (padalintą į kelias rūšis), paprastai tai yra 10-100 kilobaitų eilės dydžio atmintis. Duomenų nuskaitymas ir įrašymas šioje atmintyje yra žymiai greitesnis, nei pagrindinės atminties.
4. Greitąją atmintį galima suskaidyti į šiuos tipus:
  - a) Registrai. Šiuo metu skaičiavimuose dalyvaujantiems duomenims talpinti skirta atmintis
  - b) Bendroji atmintis. Yra prieinama skirtingų vieno multiprocesoriaus gijų, tačiau vieno multiprocesoriaus nėra prieinama kito multiprocesoriaus skaičiavimų moduliams.
  - c) L1 cache atmintis – automatizuota bendrosios atminties versija, kurios panaudojimu rūpinasi kompiliatorius.
5. L2 cache atmintis – spartinančioji atmintis, kuri yra bendra visiems multiprocesoriams, greitesnė už pagrindinę bet lėtesnė už greitąją.
6. Bendros atminties siuntimai yra prieinami per taip vadinamus bankus (angl. bank). Skirtinguose bankuose siuntimai vykdomi lygiagrečiai, tačiau kiekvienas multiprocesorius turi ribotą bankų skaičių.
7. Kiekvienas GPU multiprocesorius turi ribotą registrų skaičių.
8. Dar yra constant ir texture tipų atmintis, tačiau jų mes nenagrinėsime

# Skirtingų atminties tipų prieinamumas

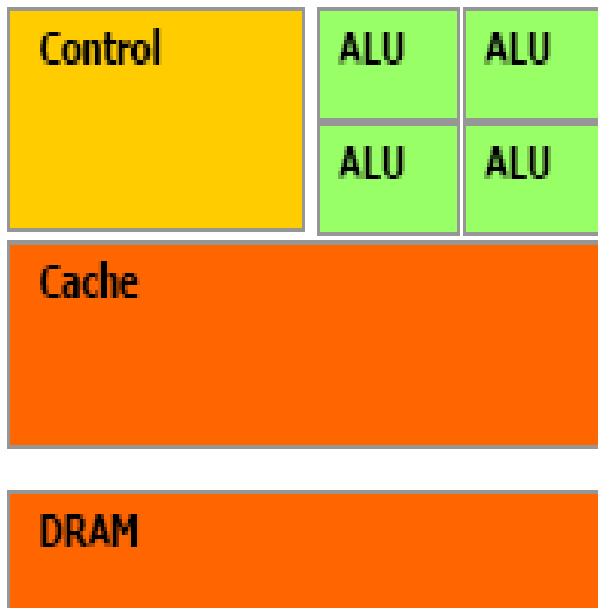


# L2 spartinančioji atmintis

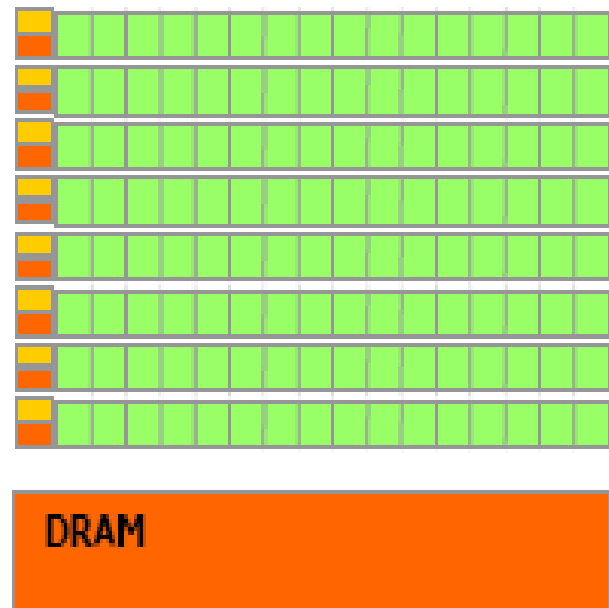


Pratimas: nurodykite, kur šioje schemeje randasi:

1. GPU L1 cache,
2. GPU bendroji atmintis,
3. GPU lokatioji atmintis,
4. GPU globalioji atmintis.



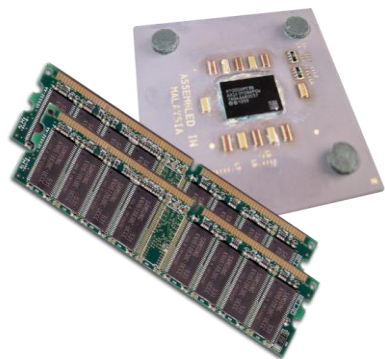
**CPU**



**GPU**

# Heterogeniniai skaičiavimai

- Heterogeniniai skaičiavimai – tai skaičiavimai, kurie atliekami naudojant skirtingos rūšies skaičiavimų įrenginius:
  - *Host* CPU ir jo atmintis (RAM, cache)
  - *Device* GPU ir jos atmintis (device memory)



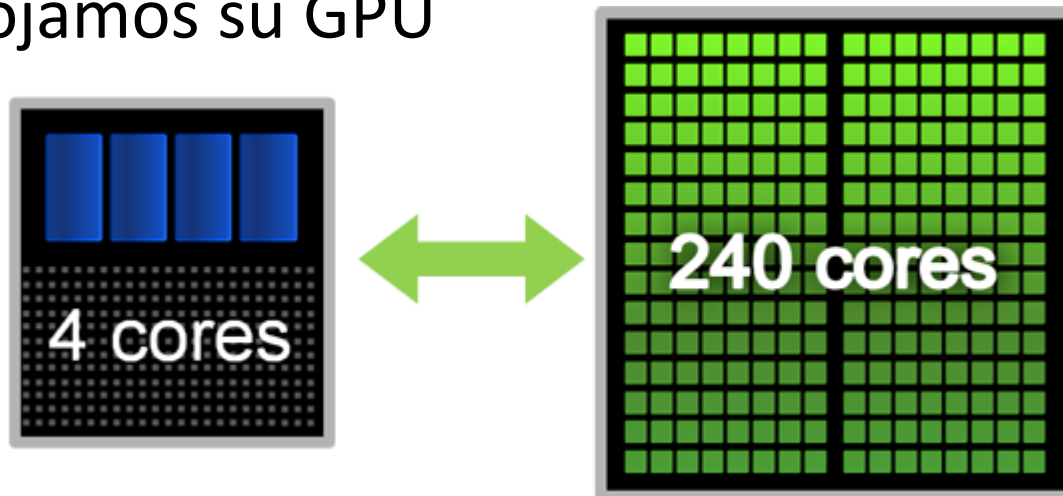
Host



Device

# CPU ir GPU naudojimas

- Paprastai abiejų rūšių įrenginiai egzistuoja kartu ir bendrą programuotojui prieinama skaičiavimų terpę galima nagrinėti kaip heterogenišką.
- Mažiau skaičiavimų reikalaujančios ir blogiau išlygiagretinamos algoritmo dalys skaičiuojamos su CPU. Daugiausia skaičiavimų reikalaujantys algoritmų dalys (jeigu jos gerai išlygiagretinamos) skaičiuojamos su GPU



# Įrenginio pavyzdys, NVIDIA Kepler K40

- 2880 streaming processors/cores (SPs) organized as 15 streaming multiprocessors (SMs)
- Each SM contains 192 cores
- Memory size of the GPU system: 12 GB
- Clock speed of a core: 745 MHz



# Heterogeniniai skaičiavimai

```
#include <iostream>
#include <algorithm>
using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

// Synchronize (ensure all the data is available)
__syncthreads();

// Apply the stencil
int result = 0;
for (int offset = -RADIUS; offset <= RADIUS; offset++)
    result += temp[lindex + offset];

// Store the result
out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out; // host copies of a, b, c
    int *d_in, *d_out; // device copies of a, b, c
    int size = (N + 2 * RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2 * RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2 * RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE, BLOCK_SIZE>>>>(d_in + RADIUS,
    d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

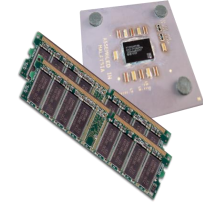
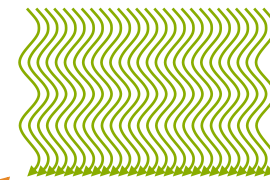
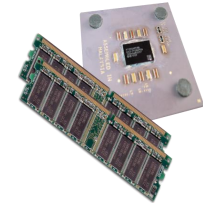
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

Lygiagreti funkcija

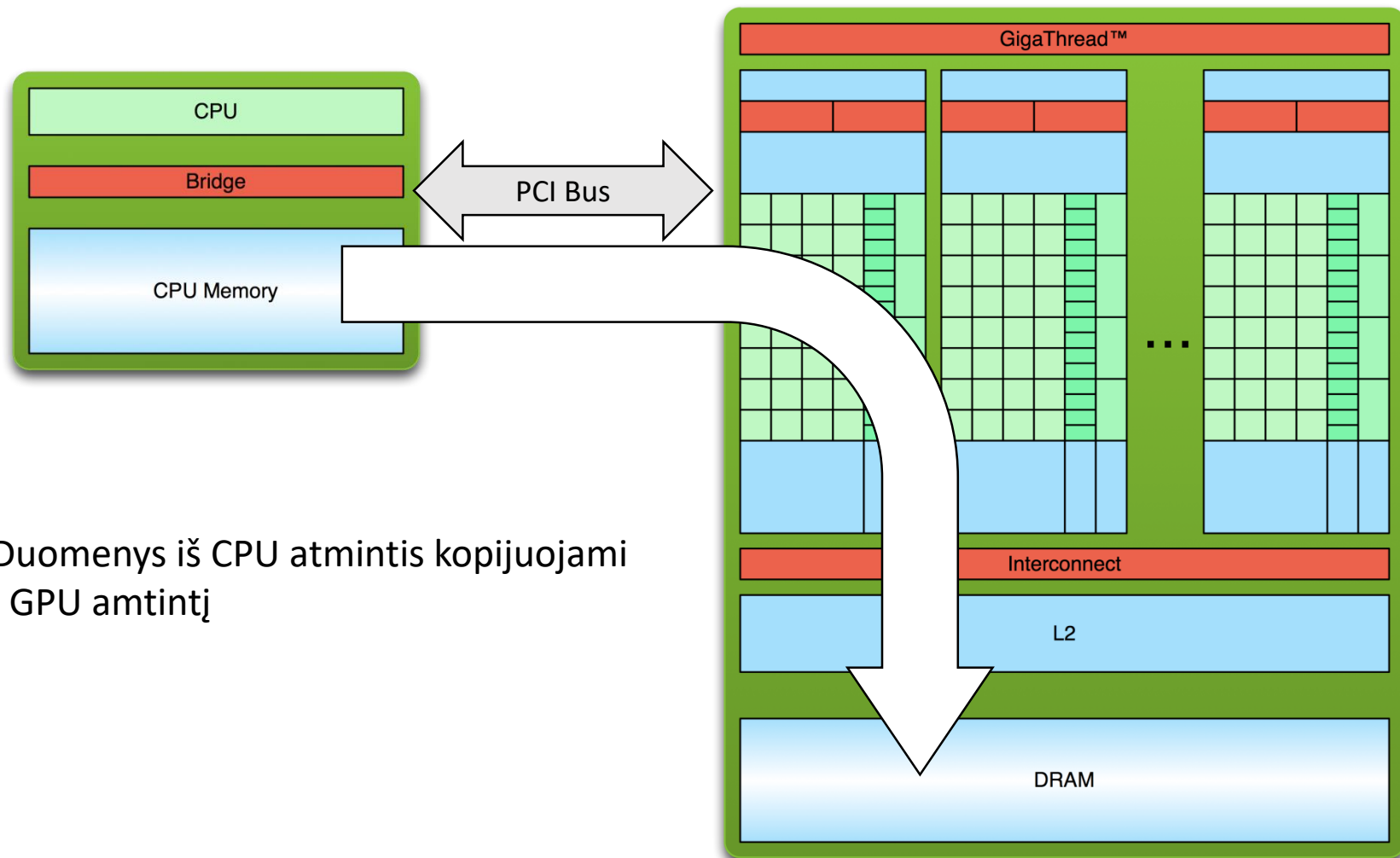
nuoseklus kodas

lygiagretus kodas

nuoseklus kodas

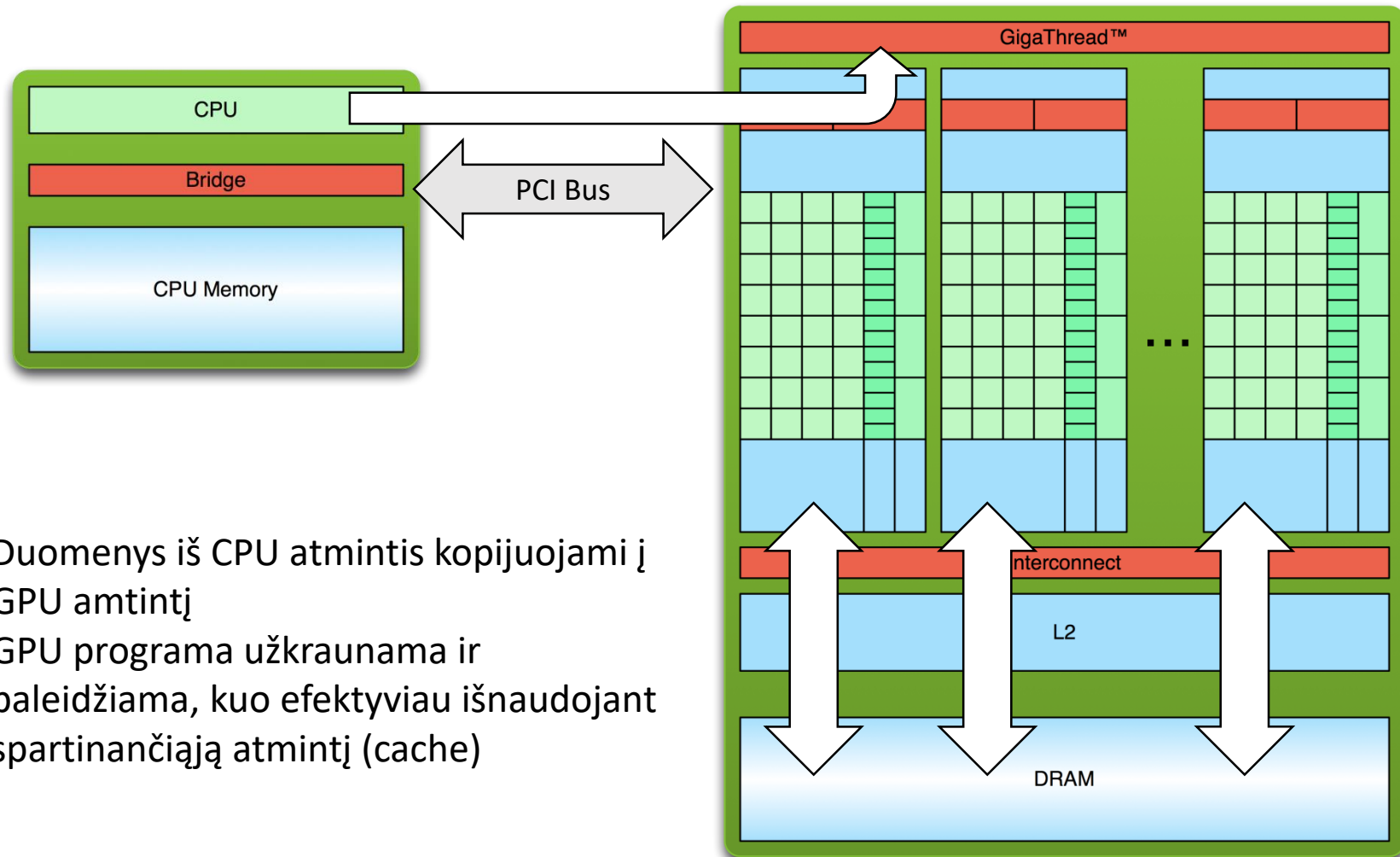


# Supaprastinta duomenų srauto schema



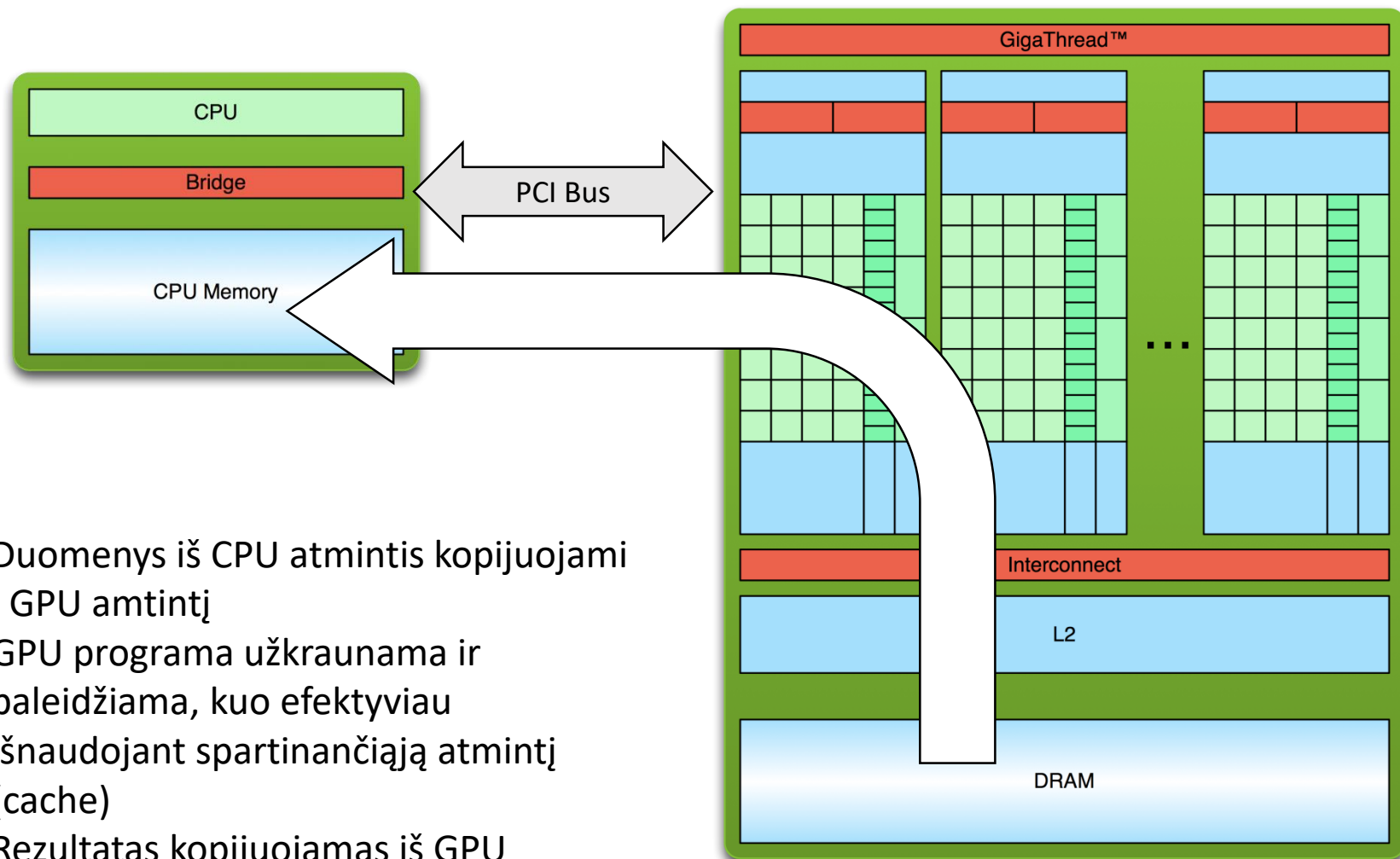
1. Duomenys iš CPU atmintis kopijuojami į GPU atmintį

# Supaprastinta duomenų srauto schema



1. Duomenys iš CPU atmintis kopijuojami į GPU atmintį
2. GPU programa užkraunama ir paleidžiama, kuo efektyviau išnaudojant spartinančiąją atmintį (cache)

# Supaprastinta duomenų srauto schema



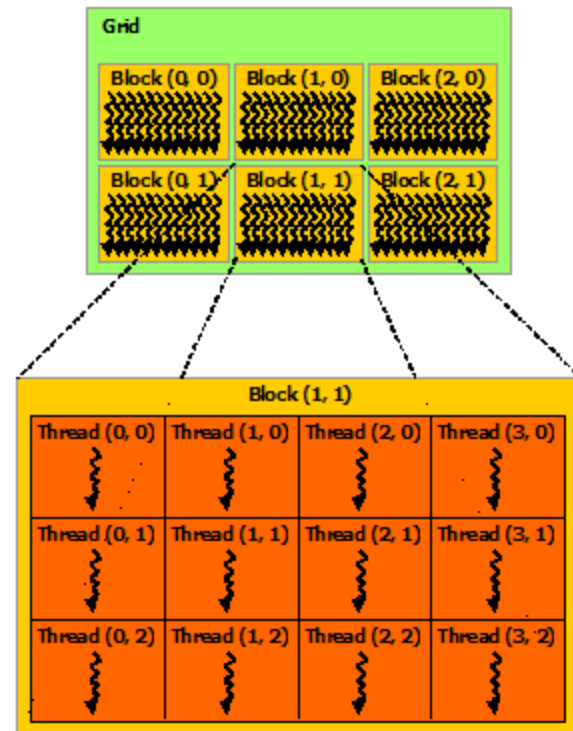
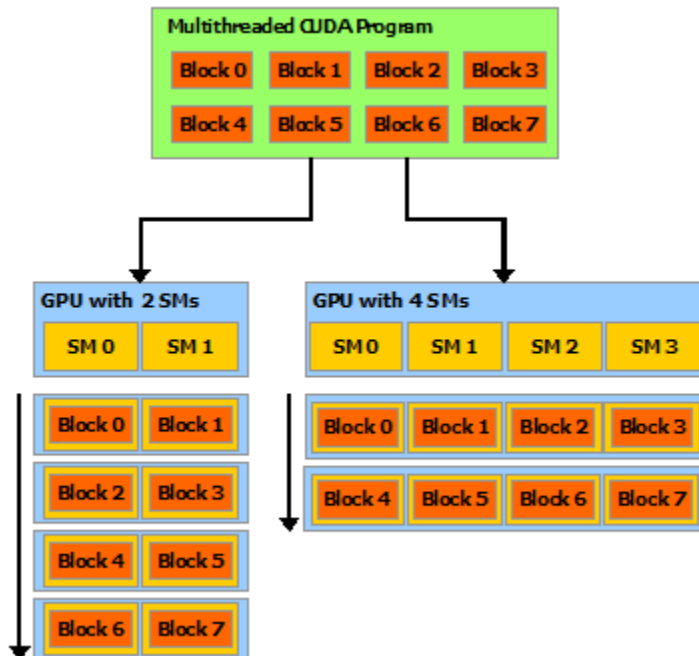
1. Duomenys iš CPU atmintis kopijuojami į GPU atmintį
2. GPU programa užkraunama ir paleidžiama, kuo efektyviau išnaudojant spartinančiąją atmintį (cache)
3. Rezultatas kopijuojamas iš GPU atminties į CPU atmintį

# CUDA

- CUDA – lygiagrečiųjų skaičiavimų platforma ir programavimo modelis
- CUDA skirta GPGPU skaičiavimams realizuoti
- Angl. Compute Unified Device Architecture, nepaisant šio akronimo tiesioginio vertimo, tai ne architektūra
- Skirtingai nuo kitokių priemonių (pvz. OpenCL) CUDA struktūra ir terminologija stipriai susieta su GPU architektūra, todėl GPGPU skaičiavimams suprasti ji tinka labai gerai.

# CUDA programavimo principai

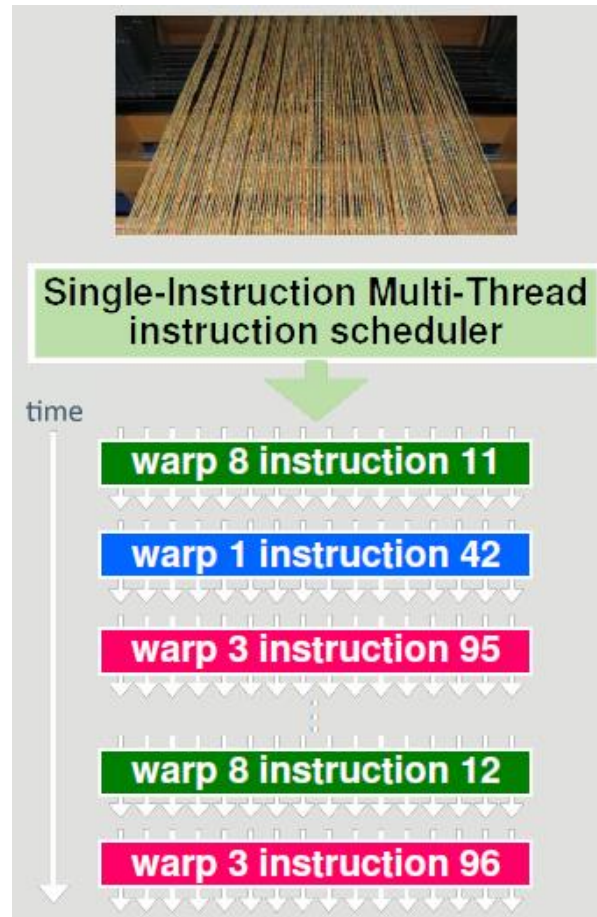
- Programuotojas apibrėžia užduočių blokus, kurias galima atlikinėti tarpusavyje nepriklausomai, paleidžia specialiąją funkciją-kernelį, visą kita daroma automatiškai.
- Programuotojas apibrėžia blokų dydį, jų skaičių ir loginį išsidėstymą
- Funkcijos-kernelio viduje programuotojas naudoja specialiuosius kintamuosius, kad nustatyti kiekvienai gijai prklausančius darbus



# SIMT principas

- NVIDIA įvedė SIMT sąvoką, kuri gerai apibūdina skaičiavimus su CUDA. SIMT – angl. Single Instruction Multiple Threads. „Single instruction“ programuotojo akimis yra GPU funkcijoje esančios instrukcijos.
- Kiekvienas multiprocesorius sukuria, apdoroja, sudaro tvarkaraštį, paleidžia gijas grupėmis po 32 gijas, vadinsime jas gijų porcijomis (angl. warps).
- Vienoje gijų porcijoje visos gijos pradeda darbą vienu metu. Programuotojas turi siekti išlaikyti jų sinchronišką veikimą (vengti išsišakojimo su if ir t.t). Nes kitu atveju kiekviena iš šakų yra vykdoma nuosekliai viena po kitos, o gijų dalis laukia.
- Kai multiprocesorius gauna vykdyti vieną arba daugiau užduočių blokų – kiekvienas blokas išskaidomas į gijų porcijas ir kiekviena iš porcijų yra valdoma tvarkaraščio sudarymo ir vykdymo mechanizmais.
- Viena gijų porcija gali vykdyti tik vieną instrukciją, todėl reikia rūpintis, kad visos gijos vienoje porcijoje tuo pačiu metu atlikinėtų tą pačią instrukciją.
- Tokio inžinerinio sprendimo priežastimi galima laikyti skaičiavimų valdymo supaprastinimą, kadangi 32-jų gijų valdymui reikalingas tik vienas gijų valdymo mechanizmas (angl. control unit).

# Instrukcijų eilių sudarymas



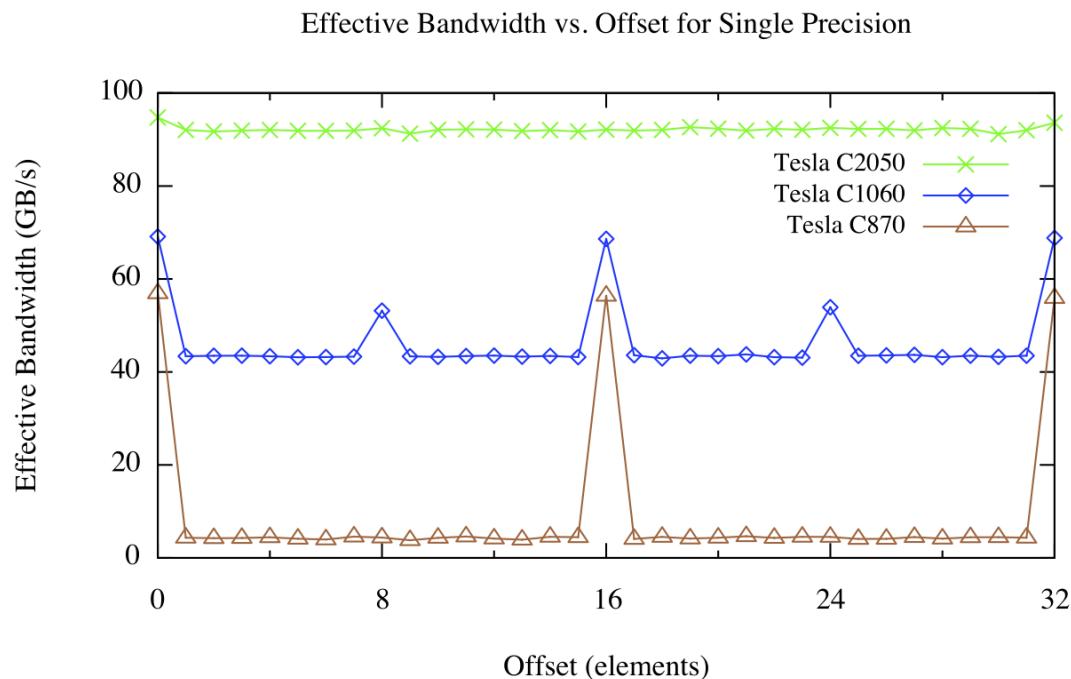


# Compute capability

- Visi NVIDIA GPU įrenginiai charakterizuojami GPU kartos rodikliu – skaičiavimų pajėgumu, angl. compute capability.
- GPU su didesniu skaičiavimų pajėgumu paprastai sumažina reikalavimus efektyvios programos kodo rašymui.

# Paslinktas atminties nuskaitymas

Viena gijų porcija nuskaityto duomenis iš skirtingų pagrindinės atminties segmentų, pvz. Nuo 96 baido iki 224 (ne imtinai):

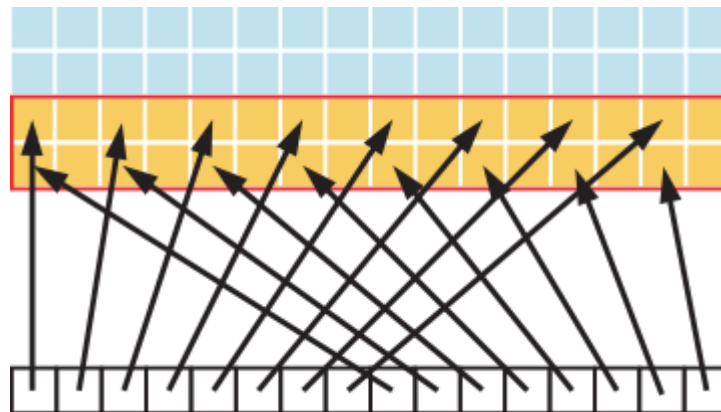


- Compute capability:
1. Tesla C2050 – 2.0
  2. Tesla C1060 – 1.3
  3. C870 – 1.0

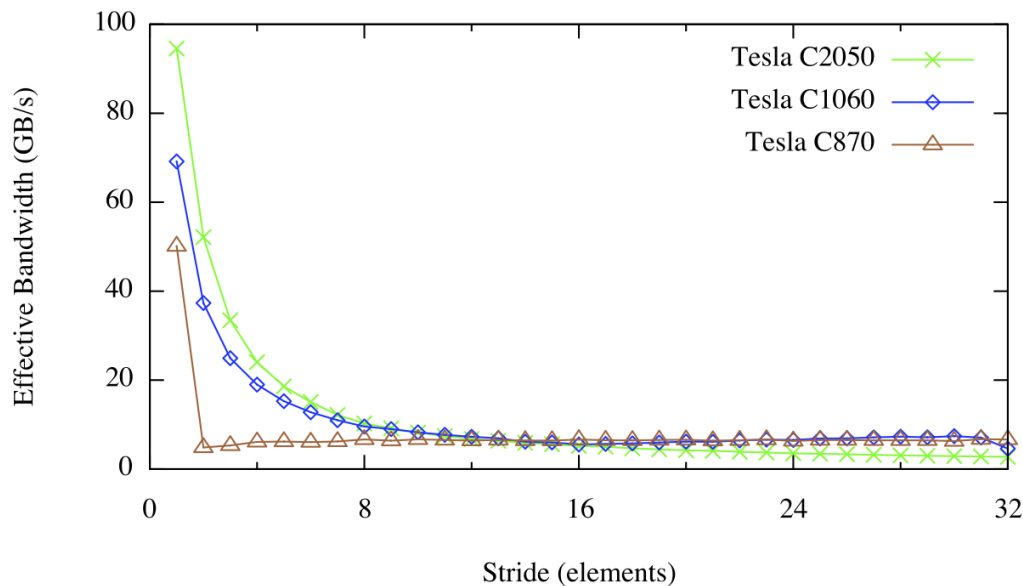
Taigi, naujesniose GPU programuotojui nebereikia rūpintis atminties lygiavimu

# Ko tikrai reikėtų vengti: atminties prieigos žingsniai

Pvz.: gijų porcija nuskaityto duomenis  
praleisdama kas antrą elementą:



Effective Bandwidth vs. Stride for Single Precision



Compute capability:

1. Tesla C2050 – 2.0
2. Tesla C1060 – 1.3
3. C870 – 1.0

# CUDA gijų užimtumas (angl. occupancy)

- Didesniam GPU skaičiavimų greičiui pasiekti gijų skaičius turi būti didesnis už branduolių skaičių, nes kiekviena gija paprastai skaičiuoja tik viso laiko dalį, kitais laiko momentais ji nuskaityto duomenis arba laukia dėl įvairių priežasčių (pvz. negali panaudoti duomenų po jų įrašymo į registrus 24 branduolio ciklus)
- Paleistų gijų skaičių riboja tai, kad
  - vienas multiprocesorius gali paleisti ne daugiau 8 užduočių blokų
  - Bendras paleistų gijų naudojamų registrų skaičius ir bendros atminties kiekis yra riboti vienam multiprocesriui
  - Yra maksimalus gijų porcijų skaičius
- Paleistų (aktyvių) gijų porcijų (po 32 gijas) skaičiaus ir maksimalaus gijų porcijų skaičiaus santykis vadinamas CUDA gijų užimtumu (occupancy)
- Maksimalus gijų užimtumas leidžia optimaliai apkrauti duomenų nuskaitymo mechanizmus arba GPU skaičiavimų modulius. Skačiuojant su GPU paprastai silpoji vieta (angl. bottleneck) yra būtent duomenų judėjimas.
- Daugiau informacijos [http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda\\_webinars\\_WarpsAndOccupancy.pdf](http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf)

# Kai kurie CUDA sintaksės elementai

- Raktažodis `__global__` GPU funkcijoms (kerneliams) deklaruoti.

Pvz. `__global__ void mykernel(void) { ...`

- GPU funkcijos kvietimo specialiųjų parametrų (blokų tinklo ir bloko matmenų) nurodymas iškart po funkcijos vardo.

Pvz. `mykernel<<<grid, threads>>>(<...>`

- `gridDim`, `blockDim`, `blockIdx` `threadIdx` – kintamieji, kurie visada deklaruoti ir rezervuoti skaičiavimų paskirstymo tikslams. Visi jie yra prieinami GPU funkcijų viduje, jie turi tris laukus: `x`, `y`, `z`. Detaliau:
  - `gridDim` parodo blokų tinklo matmenis, jeigu blokų tinklas vienmatis – `gridDim.x` yra blokų skaičius
  - `blockDim` parodo bloko matmenis, jeigu blokas yra vienmatis – `blockDim.x` yra gijų skaičius (kiekviename bloke)
  - `blockIdx` parodo bloko numerius kiekviena iš krypčių, jeigu blokų tinklas yra vienmatis, tada `blockIdx.x` yra tiesiog bloko numeris
  - `threadIdx` parodo gijų numerius tam tikro bloko viduje numerius kiekviena iš krypčių, jeigu blokas yra vienmatis, tada `threadIdx.x` yra tiesiog gijos numeris bloke
  - Jeigu blokų tinklas ir patys blokai yra vienmačiai, tada gijos numerį galima paskaičiuoti pagal formulę  $\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$
- `cudaMalloc`, `cudaFree` – funkcijos atminčiai GPU plokštėje išskirti/išvalyti
- `cudaMemcpy` – funkcija, reikalinga atminčiai kopijuoti iš/į GPU atmintį. Kryptį apibrėžia ketvirtasis argumentas, jis gali būti lygūs, pvz. `cudaMemcpyDeviceToHost` (iš GPU atminties į CPU operatyviają), `cudaMemcpyHostToDevice` (atvirkščiai) ir kt.
- Sinchronizacijos funkcijos, pvz. `__syncthreads()` – gijų sinchronizavimas to pačio bloko viduje, kviečiama GPU funkcijoje. `cudaThreadSynchronize()` – priverčia CPU laukti, kol visi GPU kerneliai pabaigs savo darbą, kviečiamas CPU vykdomajame.

Toliau nagrinėsime sintaksės  
taikymo pavyzdžius

Tolimesnės medžiagos  
tiesioginis šaltinis:

<http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf>

# Hello World!

Medžagos tiesioginis šaltinis:  
[CUDA C/C++ Basics](#)

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Kodas be CUDA konstrukcijų
- NVIDIA kompiliatorius (nvcc) gali būti naudojamas be GPU skaičiavimams skirtu kodo

Output:

```
$ nvcc  
hello_world.  
cu  
$ a.out  
Hello World!  
$
```

# Hello World! Su įrenginiui skirtu kodu

```
__global__ void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Du nauji sintaksės elementai...



# Hello World! Su įrenginiui skirtu kodu

```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc, cl.exe`

# Hello World! with Device C0de

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

# Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

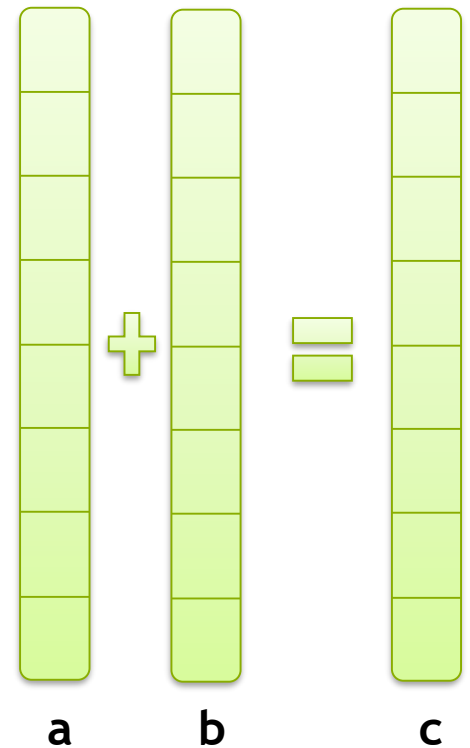
- `mykernel()` does nothing, somewhat anticlimactic!

Output:

```
$ nvcc  
hello.cu  
$ a.out  
Hello World!  
$
```

# Parallel Programming in CUDA C/C++

- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We'll start by adding two integers and build up to vector addition



# Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

# Addition on the Device

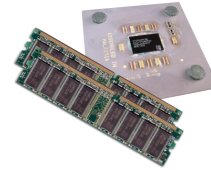
- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU

# Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`



# Addition on the Device: `add()`

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}
```

- Let's take a look at `main()`...



# Addition on the Device: `main()`

```
int main(void) {  
    int a, b, c;                // host copies of a, b, c  
    int *d_a, *d_b, *d_c;      // device copies of a, b, c  
    int size = sizeof(int);  
  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **)&d_a, size);  
    cudaMalloc((void **)&d_b, size);  
    cudaMalloc((void **)&d_c, size);  
  
    // Setup input values  
    a = 2;  
    b = 7;
```

# Addition on the Device: `main()`

```
// Copy inputs to device
```

```
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
```

```
// Launch add() kernel on GPU
```

```
add<<<1,1>>>(d_a, d_b, d_c);
```

```
// Copy result back to host
```

```
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
```

```
// Cleanup
```

```
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

```
return 0;
```

```
}
```

# RUNNING IN PARALLEL

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

Handling errors

Managing devices

# Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>> () ;
```



```
add<<< N, 1 >>> () ;
```

- Instead of executing `add ( )` once, execute N times in parallel

# Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **block**
  - The set of blocks is referred to as a **grid**
  - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

# Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- On the device, each block can execute in parallel:

Block 0

`c[0] = a[0] + b[0];`

Block 1

`c[1] = a[1] + b[1];`

Block 2

`c[2] = a[2] + b[2];`

Block 3

`c[3] = a[3] + b[3];`

# Vector Addition on the Device:

`add()`

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- Let's take a look at `main()`...

# Vector Addition on the Device:

```
#define N 512

int main(void) {
    int *a  *b  *c                // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```



# Vector Addition on the Device:

```
main()  
    // Copy inputs to device  
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);  
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);  
  
    // Launch add() kernel on GPU with N blocks  
    add<<<N,1>>>(d_a, d_b, d_c);  
  
    // Copy result back to host  
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);  
  
    // Cleanup  
    free(a); free(b); free(c);  
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);  
    return 0;  
}
```

# Review (1 of 2)

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU
- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host
- Passing parameters from host code to a device function

# Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`
- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(...)` ;
  - Use `blockIdx.x` to access block index

# INTRODUCING THREADS

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

\_\_syncthreads()

Asynchronous operation

Handling errors

Managing devices

# CUDA Threads

- Terminology: a block can be split into parallel **threads**
- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

- We use **threadIdx.x** instead of **blockIdx.x**
- Need to make one change in `main()`...

# Vector Addition Using Threads:

```
#define N 512
int main(void) {
    int *a, *b, *c;                // host copies of a, b, c
    int *d_a, *d_b, *d_c;         // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

# Vector Addition Using Threads:

```
main()  
    // Copy inputs to device  
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);  
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);  
  
    // Launch add() kernel on GPU with N threads  
    add<<<1,N>>>(d_a, d_b, d_c);  
  
    // Copy result back to host  
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);  
  
    // Cleanup  
    free(a); free(b); free(c);  
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);  
    return 0;  
}
```

# COMBINING THREADS AND BLOCKS

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

\_\_syncthreads()

Asynchronous operation

Handling errors

Managing devices

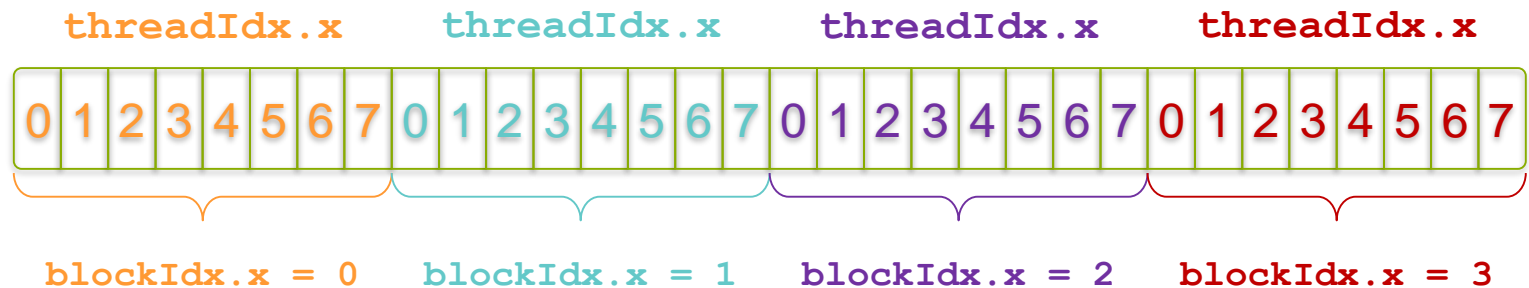


# Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that...
- First let's discuss data indexing...

# Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

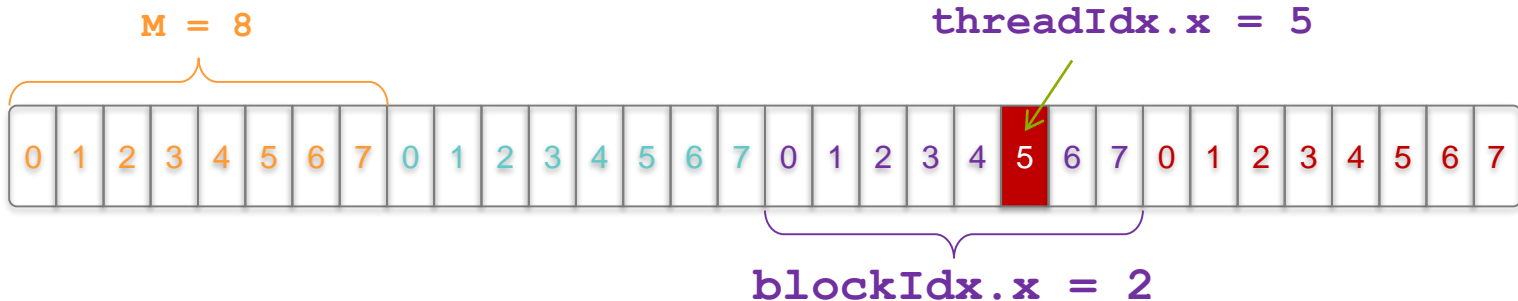


- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

# Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;  
          =           5   +           2   * 8;  
          = 21;
```

# Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    c[index] = a[index] + b[index];  
}
```

- What changes need to be made in `main()`?

# Addition with Blocks and Threads:

```
main()
```

```
#define N (2048*2048)
```

```
#define THREADS_PER_BLOCK 512
```

```
int main(void) {
```

```
    int *a, *b, *c;                                // host copies of a, b, c
```

```
    int *d_a, *d_b, *d_c;                          // device copies of a, b, c
```

```
    int size = N * sizeof(int);
```

```
    // Alloc space for device copies of a, b, c
```

```
    cudaMalloc((void **)&d_a, size);
```

```
    cudaMalloc((void **)&d_b, size);
```

```
    cudaMalloc((void **)&d_c, size);
```

```
    // Alloc space for host copies of a, b, c and setup input values
```

```
    a = (int *)malloc(size); random_ints(a, N);
```

```
    b = (int *)malloc(size); random_ints(b, N);
```

```
    c = (int *)malloc(size);
```

# Addition with Blocks and Threads:

```
// Copy inputs to device
```

```
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

```
// Launch add() kernel on GPU
```

```
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

```
// Copy result back to host
```

```
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

```
// Cleanup
```

```
free(a); free(b); free(c);
```

```
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

```
return 0;
```

```
}
```

# Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        c[index] = a[index] + b[index];  
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

# Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example...



# COOPERATING THREADS

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

\_\_syncthreads()

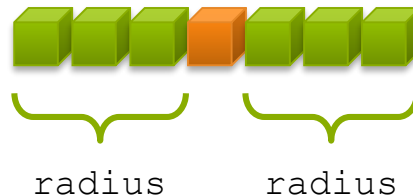
Asynchronous operation

Handling errors

Managing devices

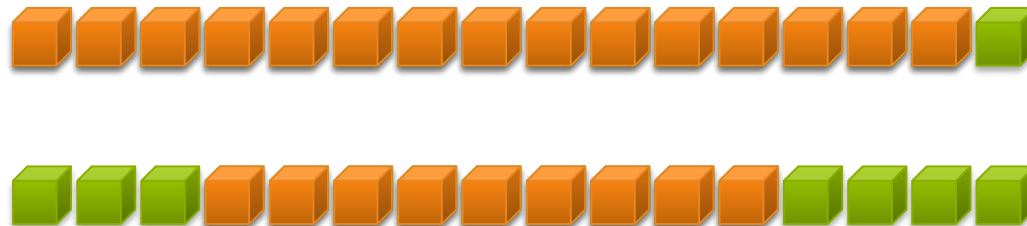
# 1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



# Implementing Within a Block

- Each thread processes one output element
  - `blockDim.x` elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times

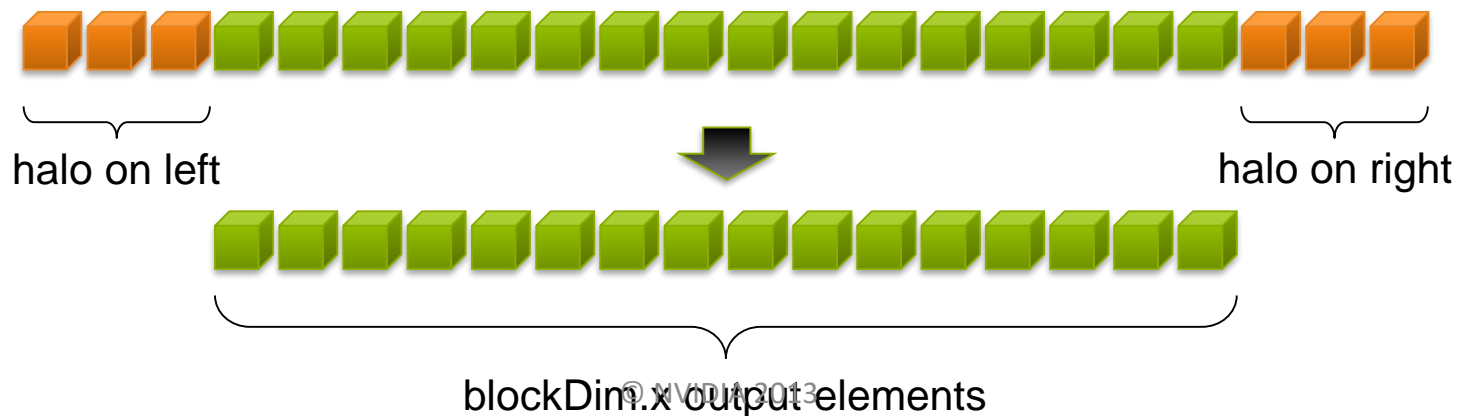


# Sharing Data Between Threads

- Terminology: within a block, threads share data via `shared memory`
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

# Implementing With Shared Memory

- Cache data in shared memory
  - Read ( $\text{blockDim.x} + 2 * \text{radius}$ ) input elements from global memory to shared memory
  - Compute  $\text{blockDim.x}$  output elements
  - Write  $\text{blockDim.x}$  output elements to global memory
- Each block needs a **halo** of radius elements at each boundary



# Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] =
            in[gindex + BLOCK_SIZE];
    }
}
```



# Stencil Kernel

```
// Apply the stencil
```

```
int result = 0;
```

```
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)  
    result += temp[lindex + offset];
```



```
// Store the result
```

```
out[gindex] = result;
```

```
}
```

# Data Race!

- The stencil example will not work...
- Suppose thread 15 reads the halo before thread 0 has fetched it...

```
temp[lindex] = in[gindex];           Store at temp[18]   
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];    Skipped, threadIdx > RADIUS  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
  
int result = 0;  
result += temp[lindex + 1];          Load from temp[19] 
```



# \_\_syncthreads()

- `void __syncthreads();`
- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

# Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
}
```

# Stencil Kernel

```
// Apply the stencil  
int result = 0;  
for (int offset = -RADIUS ; offset <= RADIUS ; offset++)  
    result += temp[lindex + offset];  
  
// Store the result  
out[gindex] = result;  
}
```

# Review (1 of 2)

- Launching parallel threads
  - Launch  $N$  blocks with  $M$  threads per block with `kernel<<<N,M>>> (...)` ;
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x
```

# Review (2 of 2)

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards

# MANAGING THE DEVICE

## CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

\_\_syncthreads()

Asynchronous operation

Handling errors

Managing devices

# Coordinating Host & Device

- Kernel launches are **asynchronous**
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

**cudaMemcpy ( )**

Blocks the CPU until the copy is complete  
Copy begins when all preceding CUDA calls have completed

**cudaMemcpyAsync ( )**

Asynchronous, does not block the CPU

**cudaDeviceSynchronize ( )**

Blocks the CPU until all preceding CUDA calls have completed

# Reporting Errors

- All CUDA API calls return an error code (`cudaError_t`)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

```
printf("%s\n",  
cudaGetErrorString(cudaGetLastError()));
```



# Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int
device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices

```
cudaSetDevice(i) to select current device
```

```
cudaMemcpy(...) for peer-to-peer copies†
```

<sup>†</sup> requires OS and device support

# Introduction to CUDA C/C++

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - `__global__`, `blockIdx.x`, `threadIdx.x`, `<<<>>>`
  - Manage GPU memory
    - `cudaMalloc()`, `cudaMemcpy()`, `cudaFree()`
  - Manage communication and synchronization
    - `__shared__`, `__syncthreads()`
    - `cudaMemcpy()` VS `cudaMemcpyAsync()`,  
`cudaDeviceSynchronize()`

# Compute Capability

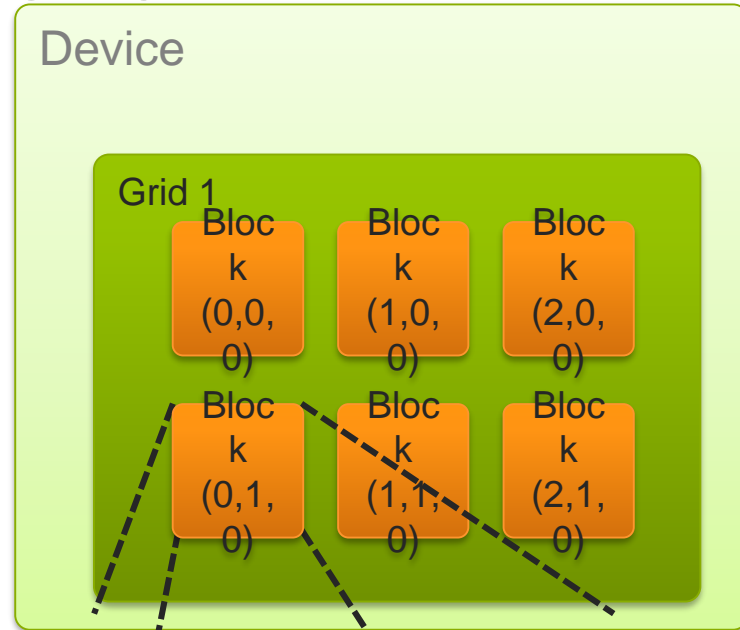
- The **compute capability** of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

- The following presentations concentrate on Fermi devices
  - Compute Capability  $\geq 2.0$

# IDs and Dimensions

- A kernel is launched as a grid of blocks of threads
  - `blockIdx` and `threadIdx` are 3D
  - We showed only one dimension (x)



- Built-in variables:

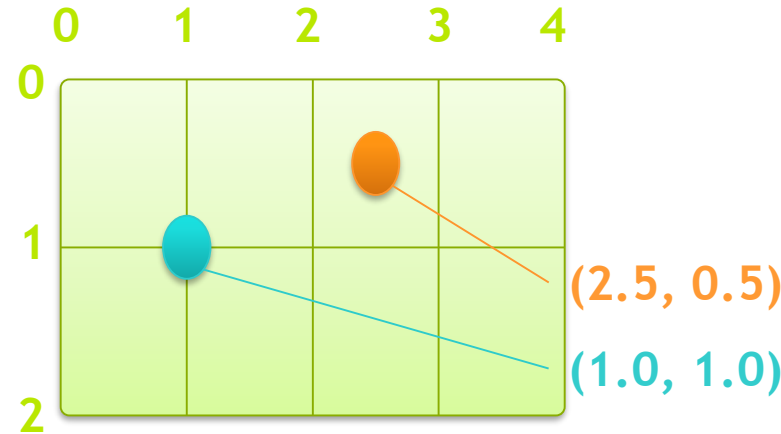
- `threadIdx`
- `blockIdx`
- `blockDim`
- `gridDim`

Block (1,1,0)

Thread (0,0,0)	Thread (1,0,0)	Thread (2,0,0)	Thread (3,0,0)	Thread (4,0,0)
Thread (0,1,0)	Thread (1,1,0)	Thread (2,1,0)	Thread (3,1,0)	Thread (4,1,0)
Thread (0,2,0)	Thread (1,2,0)	Thread (2,2,0)	Thread (3,2,0)	Thread (4,2,0)

# Textures

- Read-only object
  - Dedicated cache
- Dedicated filtering hardware  
(Linear, bilinear, trilinear)
- Addressable as 1D, 2D or 3D
- Out-of-bounds address handling  
(Wrap, clamp)



# Topics we skipped

- We skipped some details, you can learn more:
  - CUDA Programming Guide
  - CUDA Zone – tools, training, webinars and more  
[developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)
- Need a quick primer for later:
  - Multi-dimensional indexing
  - Textures