Music Machine Learning

X – Data complexity - GPU Computing

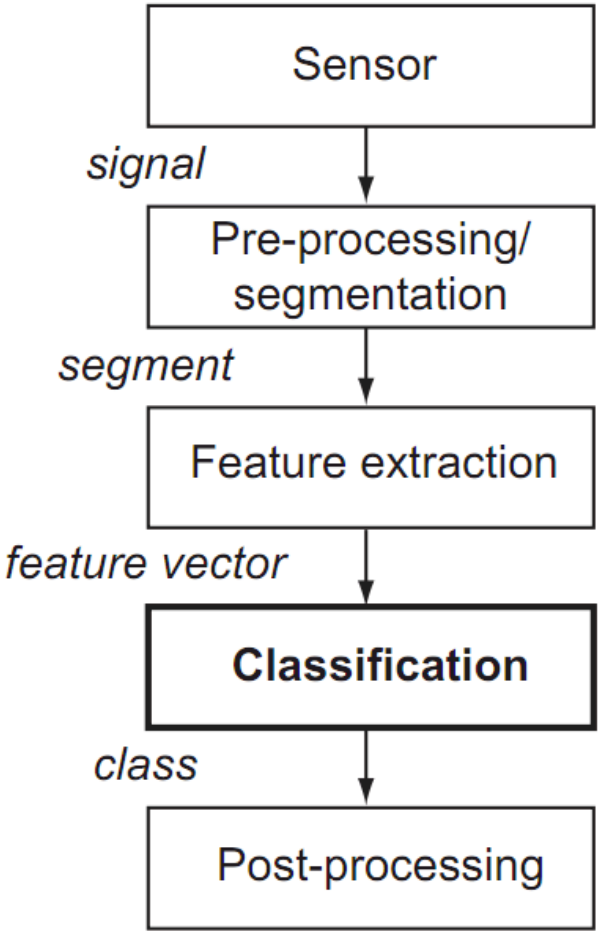
Master ATIAM - Informatique

Philippe Esling (esling@ircam.fr)

Maître de conférences – UPMC

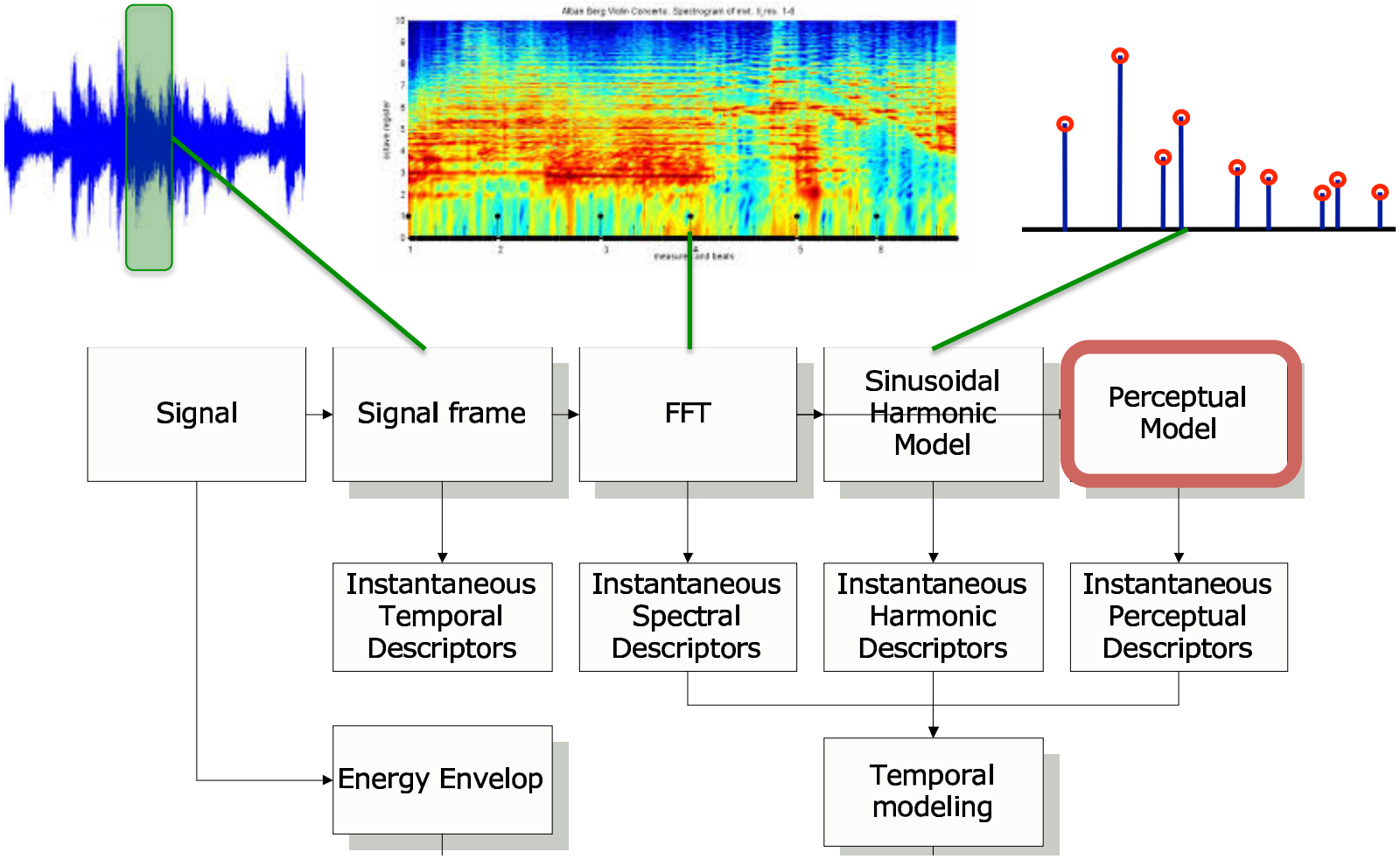
Equipe représentations musicales (IRCAM, Paris)

Any type of audio recognition



- Up to now we have seen classification
- Each working on a specific problem
- But on which features ?
- Years of research on audio features

Audio features



Retrieving musical information (MIR)

1. **Harmonic analysis**
    1. Pitch tracking, melody estimation
    2. Multiple F0 estimation, chord estimation
2. **Rhythmic analysis**
    1. Onset detection
    2. Tempo estimation, beat tracking
3. **Content-based analysis**
    1. Instrument recognition
    2. Structure analysis
    3. Classification (genre recognition, tags, mood, …)
    4. Music similarity and cover detection
    5. Query by content (query by humming)

Known issues of machine learning

1. Overfitting / Overtraining
(**function approximation**)
1. Curse of dimensionnality
(**dimensionnality reduction**)
1. Cross-validation
(**generalization probability**)
1. Cherry picking
(**datasets discrepancies**)
1. The *No Free Lunch* theorem
(**overall accuracy**)

Defining Machine Learning

A machine learning algorithm is a combination of 3 elements:

(either explicitly specified or implicit)

✓ the choice of a specific function family: F
(often a parameterized family)

✓ a way to evaluate the quality of a function $f \in F$
(typically using a cost (or loss) function L
mesuring how wrongly f prédicts)

✓ a way to search for the «best» function $f \in F$
(typically an optimization of function parameters to minimize the overall loss over the training set).

Defining Machine Learning

Evaluating a predictor $f(x)$

The performance of a predictor is often evaluated using **several** different evaluation metrics:

- Evaluations of true quantities of interest ($ saved, #lifes saved, ...) when using predictor inside a more complicated system.
- «Standard» evaluation metrics in a specific field (e.g. BLEU (Bilingual Evaluation Understudy) scores in translation)
- Misclassification error rate for a classifier (or precision and recall, or F-score, ...).
- The loss actually being optimized by the ML algorithm (often different from all the above...)

Defining Machine Learning

Expected risk v.s. Empirical risk

Examples (x,y) are supposed drawn i.i.d. from an unknown true distribution $p(\mathrm{x},\mathrm{y})$ (from nature or industrial process)

- Generalization error = Expected risk (or just «Risk»)
  «how poorly we will do on average on the infinity of future examples from that unknown distribution»

$$R(f) = \mathbb{E}_{p(\mathbf{x},\mathbf{y})}[L(f(\mathbf{x}), \mathbf{y})]$$

- Empirical risk = average loss on a finite dataset
  «how poorly we're doing on average on this finite dataset»

$$\hat{R}(f, D) = \frac{1}{|D|} \sum_{(\mathbf{x},\mathbf{y}) \in D} L(f(\mathbf{x}), \mathbf{y})$$

where $|D|$ is the number of examples in D

Defining Machine Learning

Evaluating the generalization error

- We can't compute expected risk $R(f)$
- But $\hat{R}(f, D)$ is a good estimate of $R(f)$ provided:
  - D was not used to find/choose f
    otherwise estimate is biased ⇨ can't be the training set!
  - D is large enough (otherwise estimate is too noisy); drawn from p

➡ Must keep a separate test-set $D_{\text{test}} \neq D_{\text{train}}$ to properly estimate generalization error of $\hat{f}(D_{\text{train}})$:

$$R(\hat{f}(D_{\text{train}})) \approx \hat{R}(\hat{f}(D_{\text{train}}), D_{\text{test}})$$

generalization error — average error on **test**-set (never used for training)

This is the test phase in ML

Defining Machine Learning

Model selection

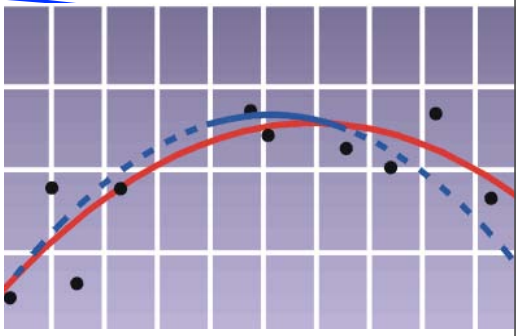
Choosing a specific function family F

Defining Machine Learning

$F_{polynomial\ p}$

Polynomial predictor (of degree p):

$$f(x) = b + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_p x^p$$



F_{linear}

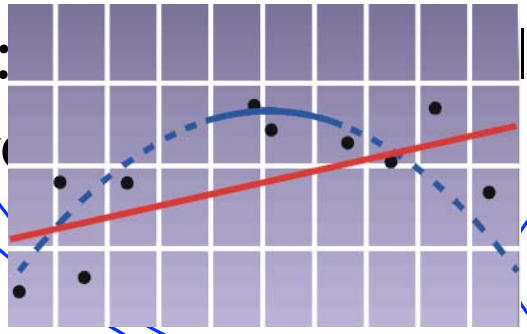
Linear (affine) predictor: $f_\theta(x) = wx + b$ (in 1 dimension)

(«linear regression») $f_\theta(x) = w^T x + b$ (in d dimensions)

$$\theta = \{w \in \mathbb{R}^d, b \in \mathbb{R}\}$$

Q: [illegible]lest

pr[illegible]



F_{const}

Constant predictor: $f_\theta(x)=b$

where $\theta=\{b\}$

(always predict the same value or class!)

Defining Machine Learning

Capacity of a learning algorithm

- Choosing a specific Machine Learning algorithm means choosing a specific function family F.
- How «big, rich, flexible, expressive, complex» that family is, defines what is informally called the «capacity» of the ML algorithm.

  Ex: capacity($F_{polynomial\ 3}$) > capacity(F_{linear})
- One can come up with several formal measures of «capacity» for a function family / learning algorithm (e.g. VC-dimension Vapnik–Chervonenkis)
- One rule-of-thumb estimate, is the number of adaptable parameters: i.e. how many *scalar* values are contained in θ.

  Notable exception: chaining many linear mappings is still a linear mapping!

Defining Machine Learning

Effective capacity, and capacity-control hyper-parameters

The «effective» capacity of a ML algo is controlled by:

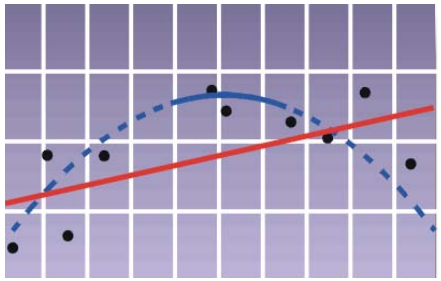
- Choice of ML algo, which determines big family F
- Hyper-parameters that further specify F
  e.g.: degree p of a polynomial predictor; Kernel choice in SVMs;
  #of layers and neurons in a neural network
- Hyper-parameters of «regularization» schemes
  e.g. constraint on the norm of the weights w
  (⇨ ridge-regression; L_2 weight decay in neural nets);
  Bayesian prior on parameters; noise injection (dropout); ...
- Hyper-parameters that control early-stopping of the iterative search/optimization procedure.
  (⇨ won't explore as far from the initial starting point)

Defining Machine Learning

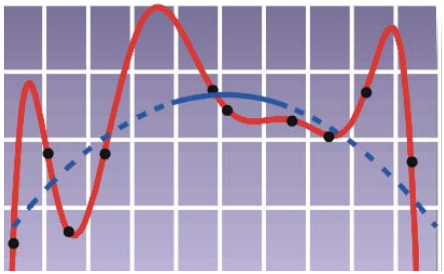
Tuning the capacity

- Capacity must be optimally tuned to ensure good generalization
- by choosing Algorithm and hyperparameters
- to avoid under-fitting and over-fitting.

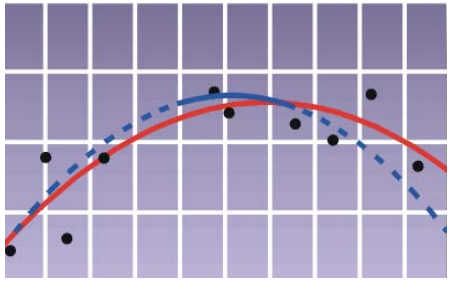
Ex: 1D regression with polynomial predictor



capacity too low
⇨under-fitting



capacity too high
⇨over-fitting



optimal capacity
⇨good generalisation

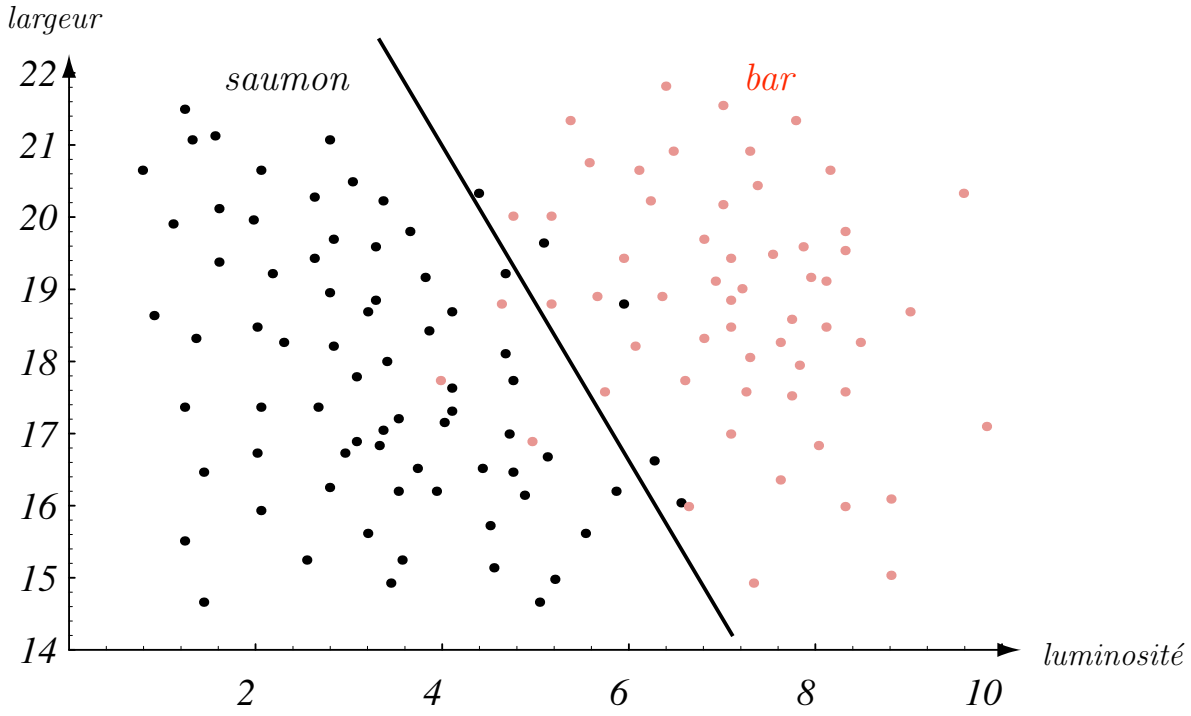
performance on training set is not a good estimate of generalization

Defining Machine Learning

Ex: 2D classification

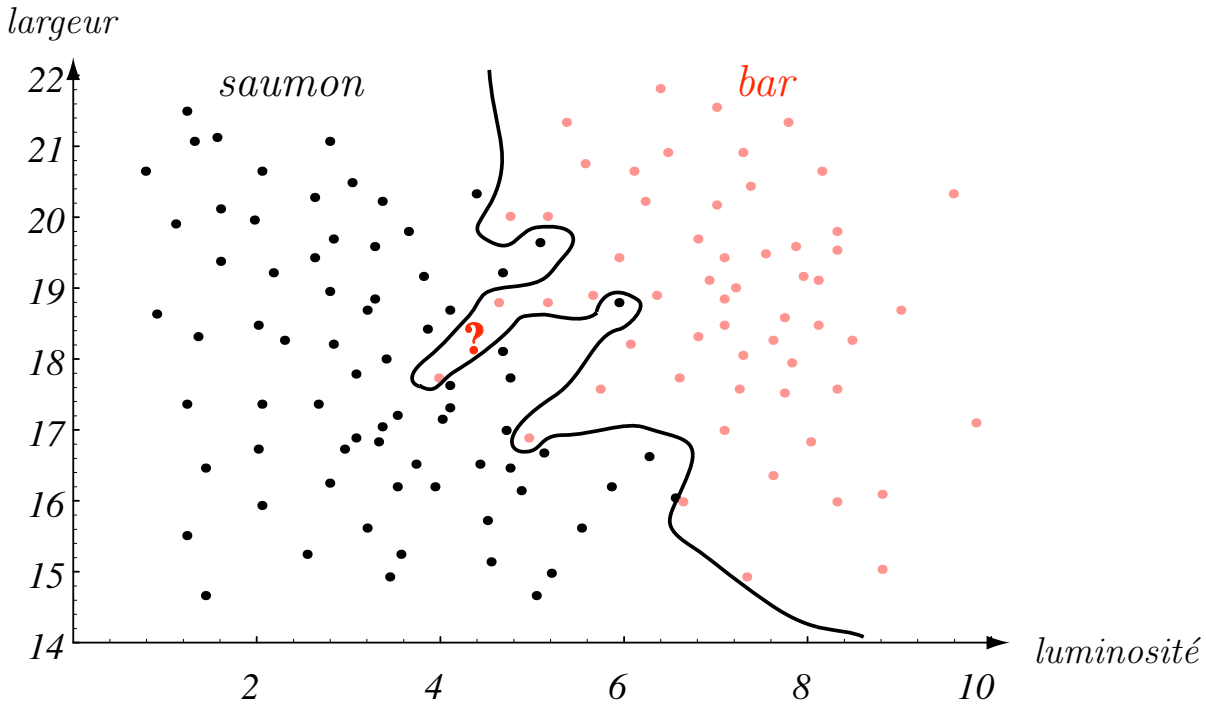
Linear classifier

- Function family too poor (too inflexible)
- = **Capacity too low** for this problem (relative to number of examples)
- => Under-fitting



Defining Machine Learning

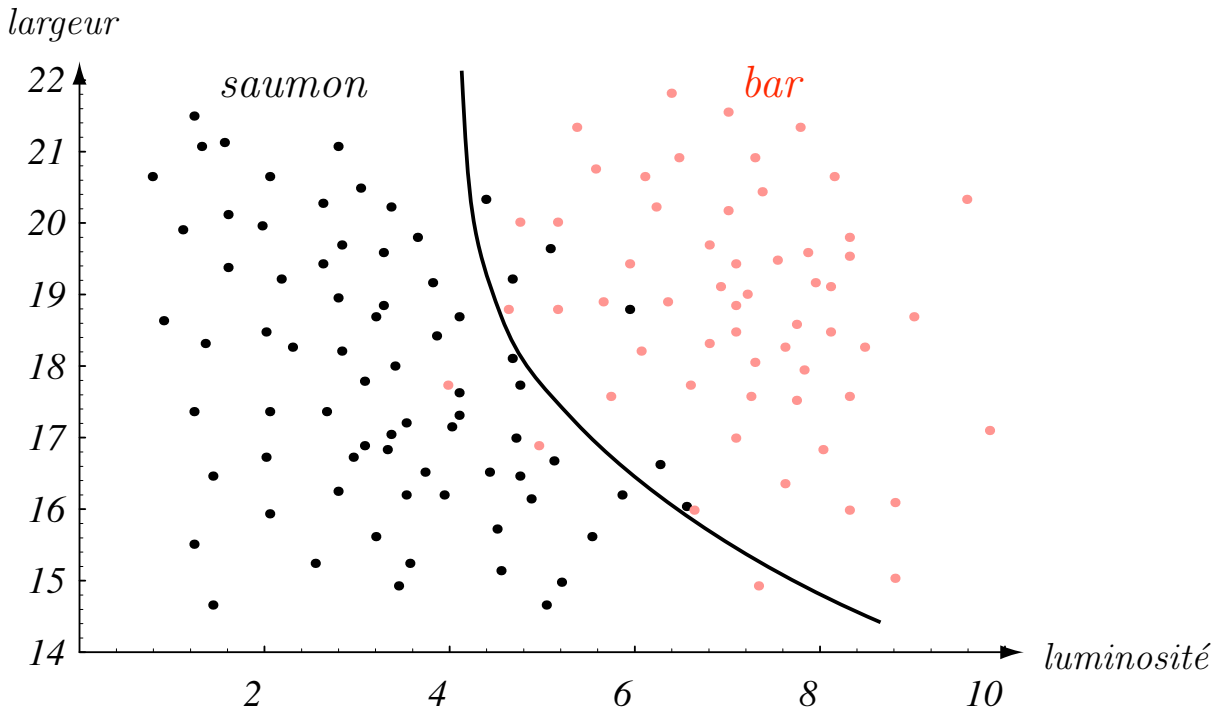
- Function family too rich
  (too flexible)
- = **Capacity too high** for this problem
  (relative to the number of examples)
- => Over-fitting



Nombre d'erreurs d'entrainement: 0

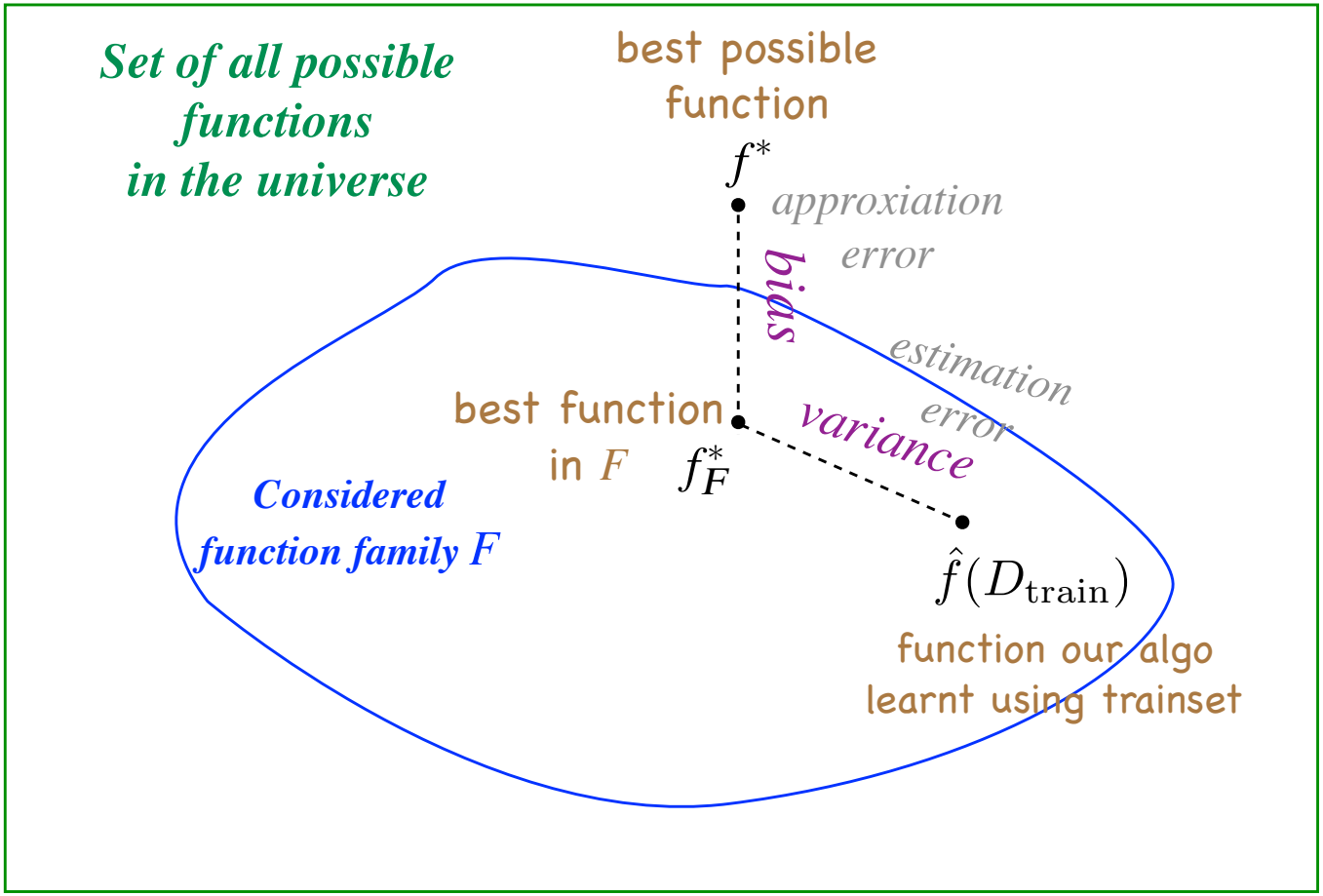
Defining Machine Learning

- **Optimal capacity** for this problem
  (par rapport à la quantité de données)
- => Best generalization
  (on future test points)



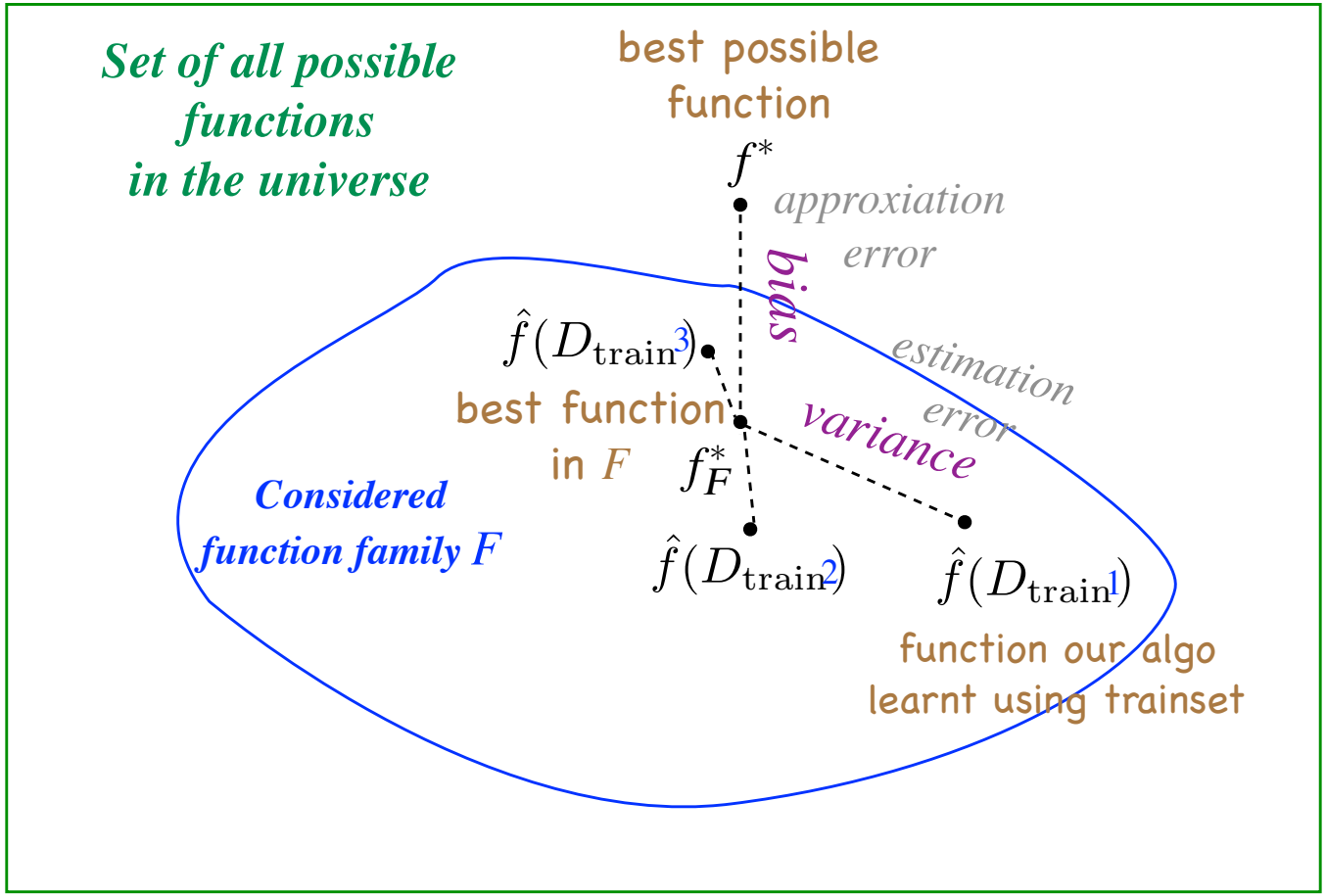
Defining Machine Learning

Decomposing the generalization error



Markov Chains

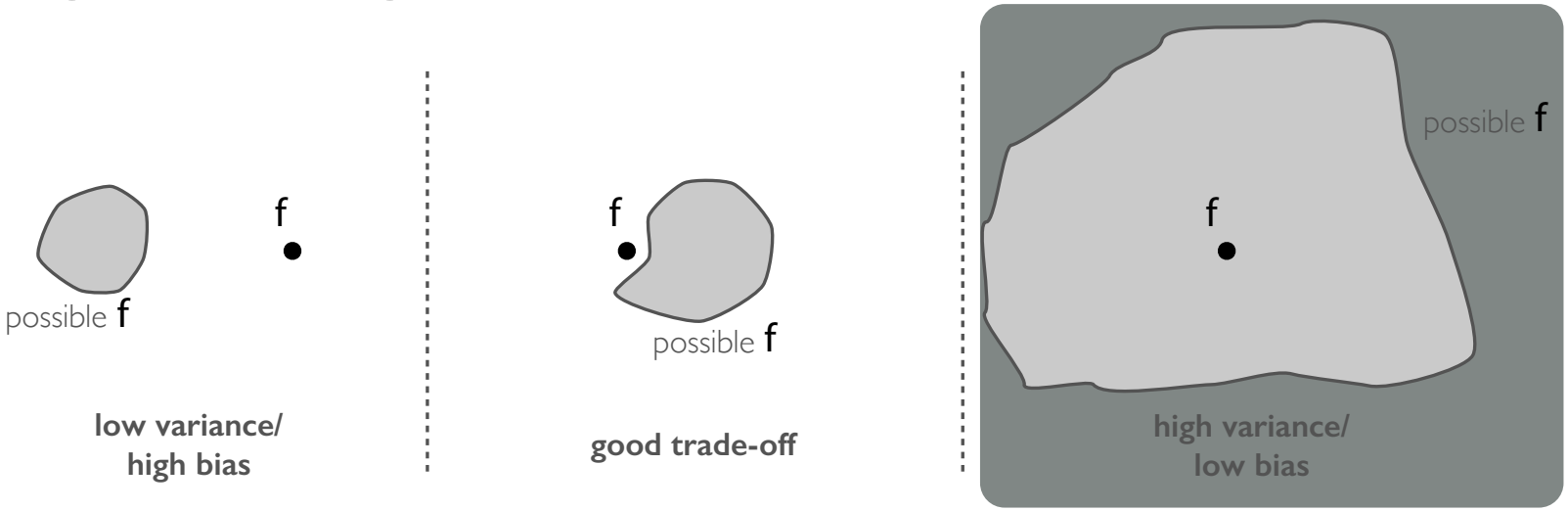
What is responsibe for the variance?



Choosing the right function

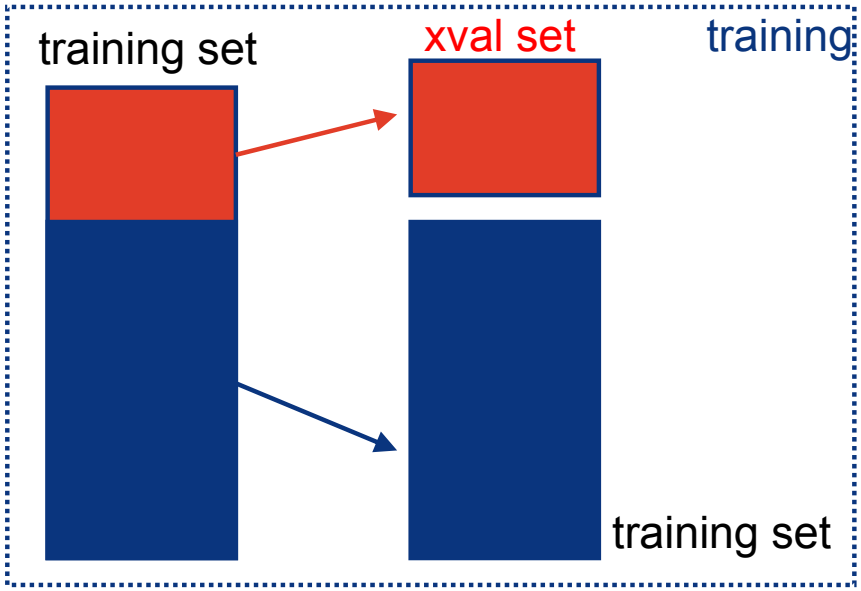
Topics: why training is hard

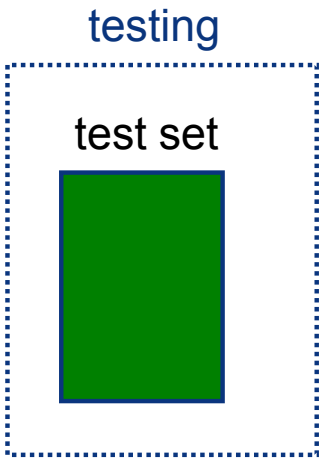
- Second hypothesis: overfitting
  - we are exploring a space of complex functions
  - deep nets usually have lots of parameters
- Might be in a high variance / low bias situation



Cross-validation

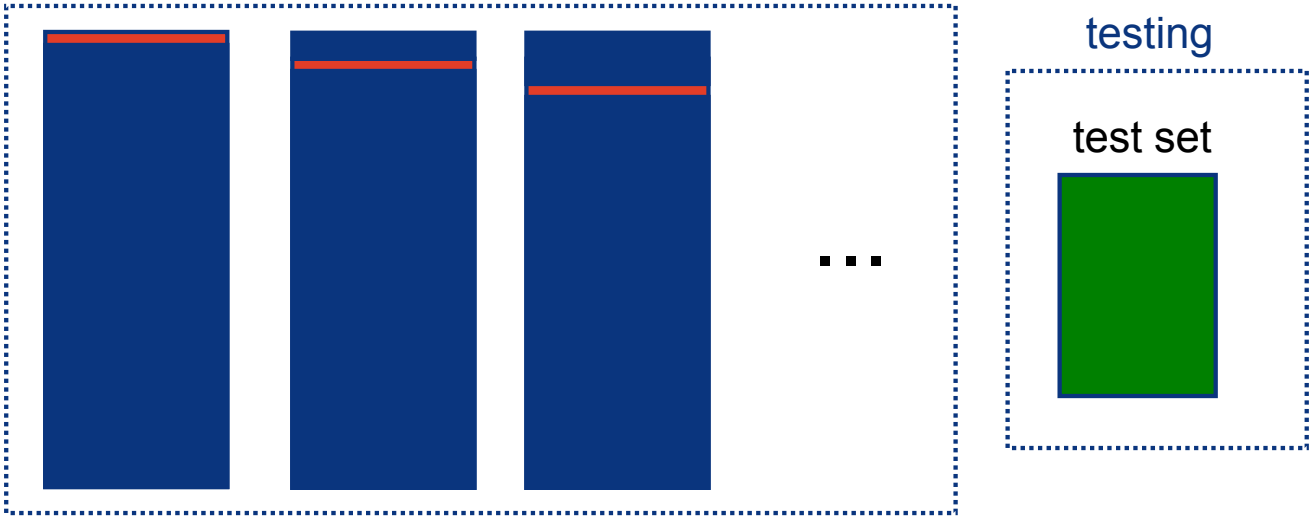
- basic idea:
  - leave some data out of your training set (cross validation set)
  - train with different parameters
  - evaluate performance on cross validation set
  - pick best parameter configuration





Cross-validation

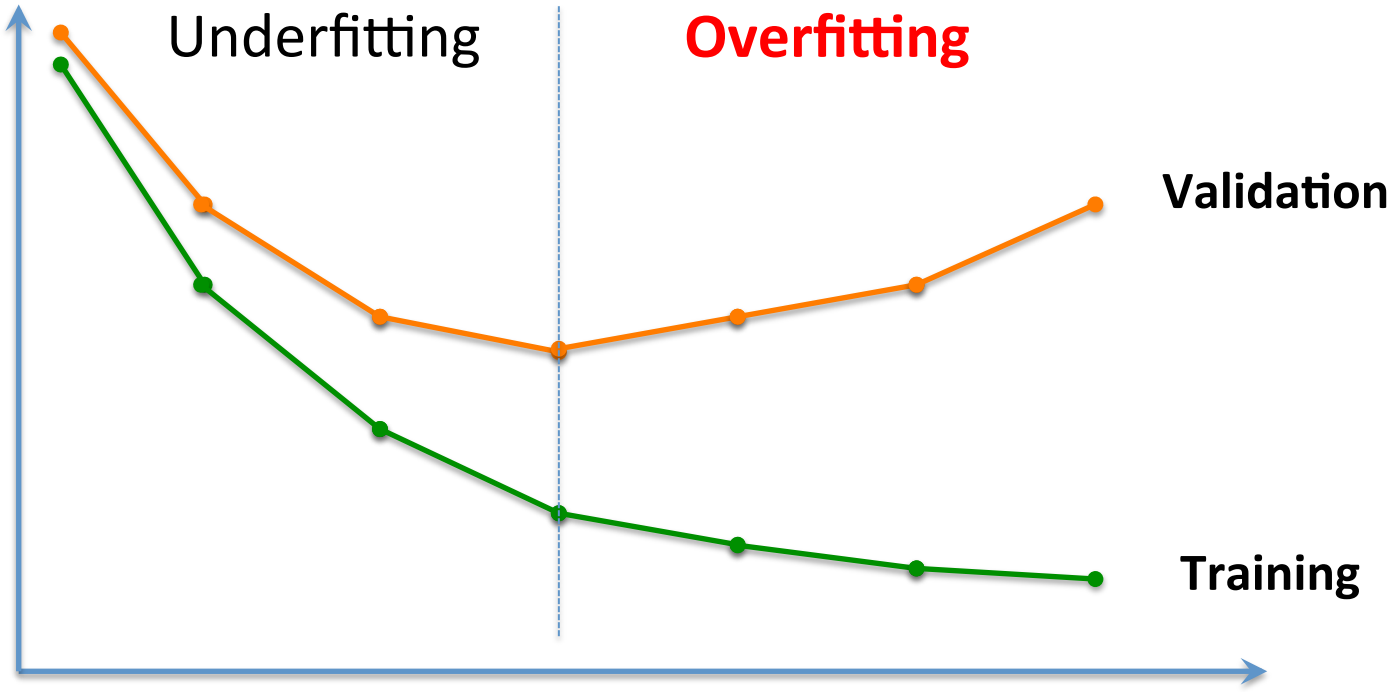
- many variations
- leave-one-out CV:
  - compute n estimators of $P_X(x)$ by leaving one X_i out at a time
  - for each $P_X(x)$ evaluate $P_X(X_i)$ on the point that was left out
  - pick $P_X(x)$ that maximizes this likelihood



Knowing when to stop

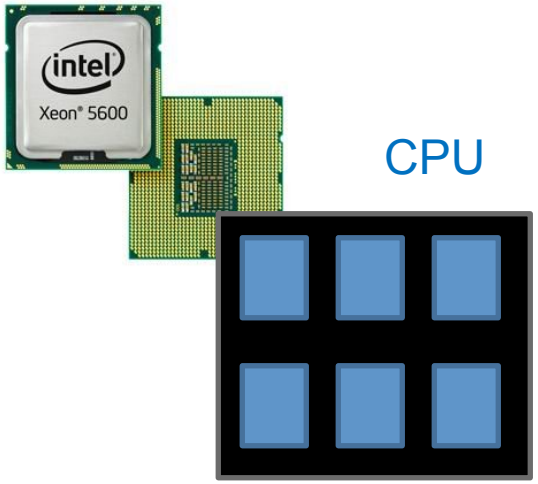
Early stopping

Stop the learning when the error increases on the validation set
(Approximates the generalization power)

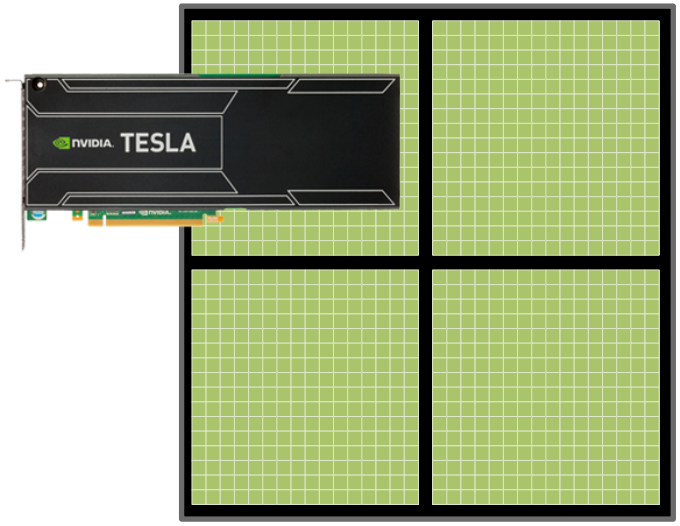


GPU Computing

GPU Computing



CPU



GPU

CPUs consist of a few cores optimized for serial processing

Intel Xeon X5650:

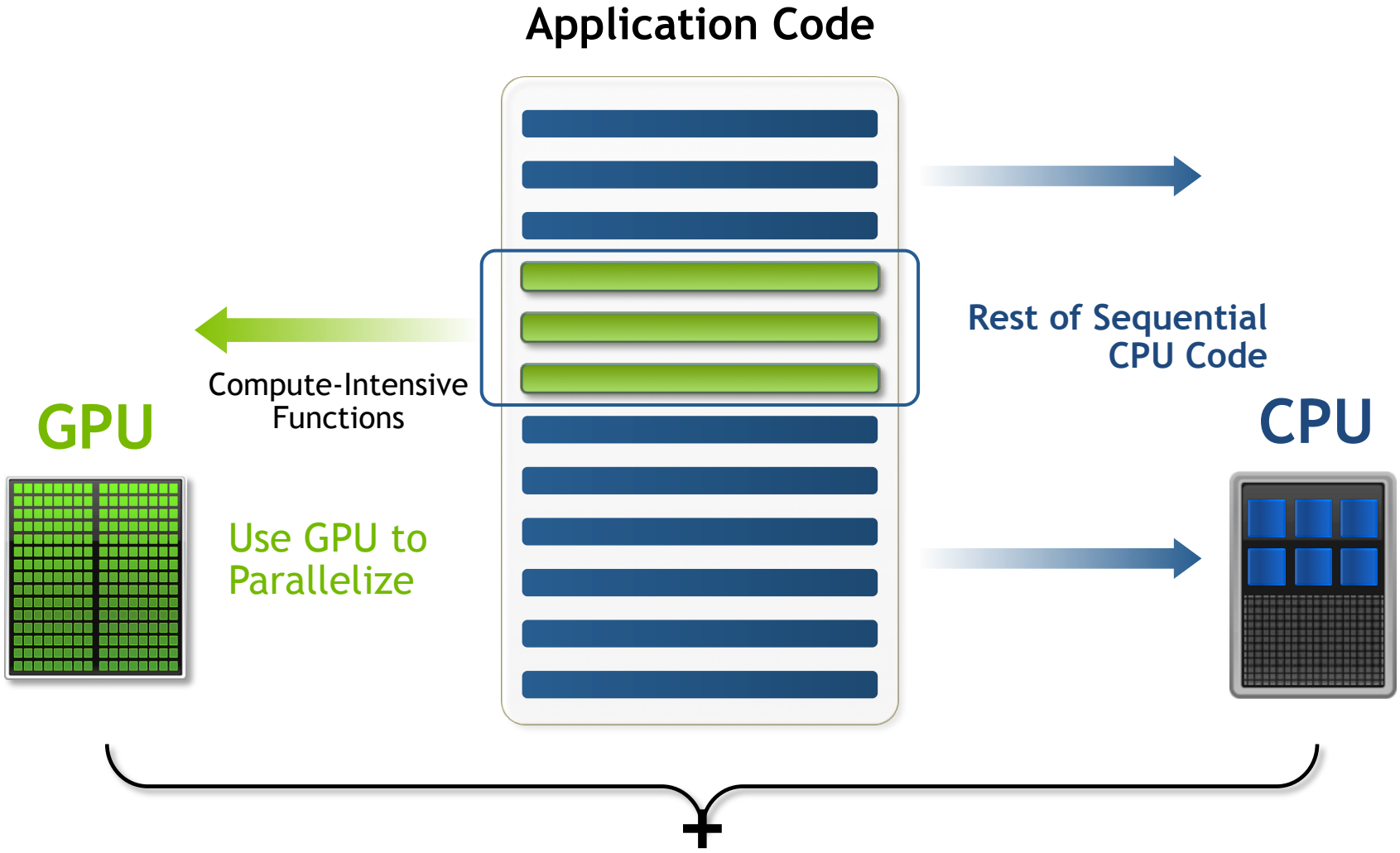
Clock speed: **2.66** GHz
4/cycle CPU - **6** cores
= **63.84** Gigaflops double precision
Memory size: **288** GB
Bandwidth: **32** GB/sec

GPUs consist of thousands of smaller cores for parallel processing

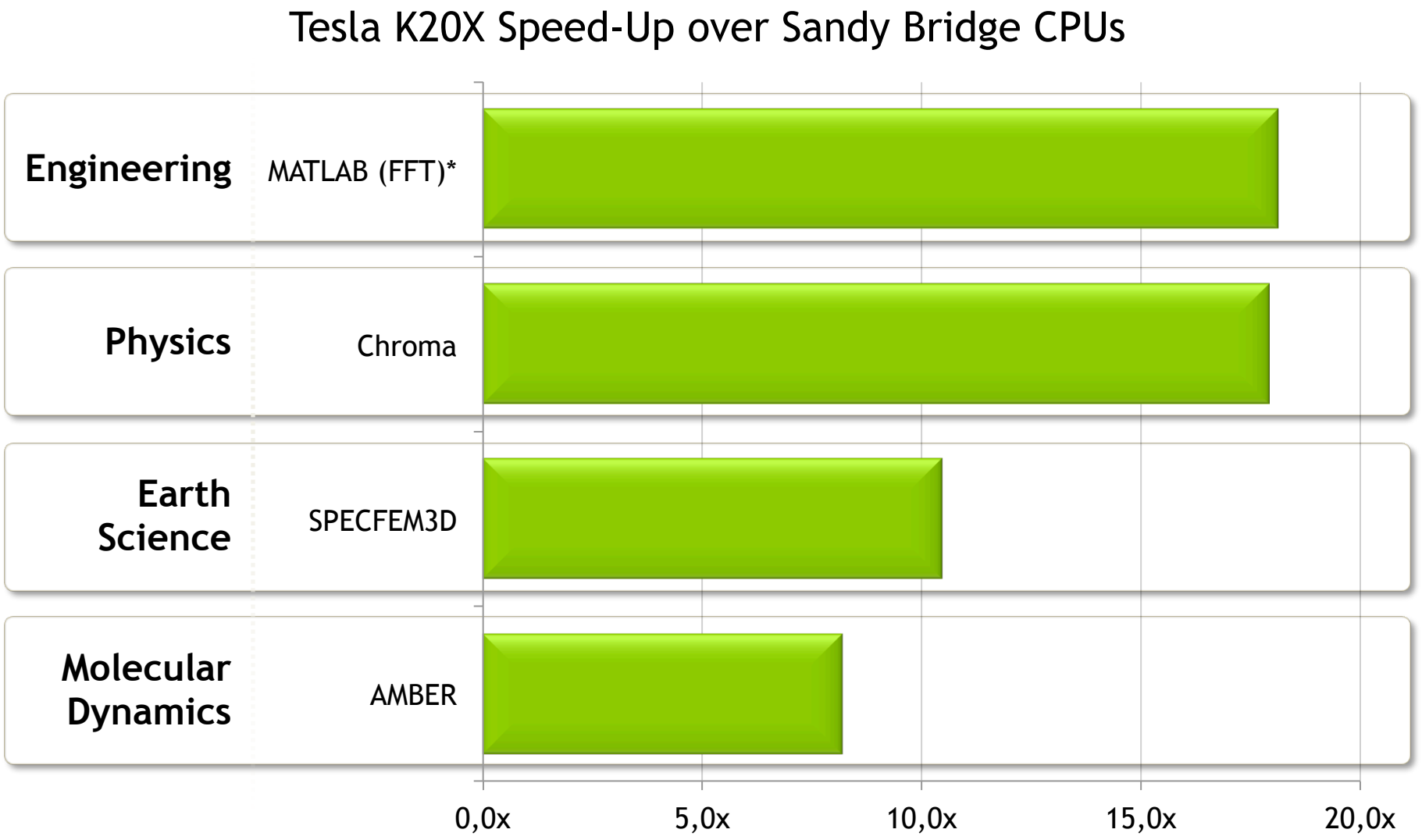
NVIDIA Tesla M2070:

Core clock: **1.15**GHz
1/cycle - **448** CUDA cores
= 515 Gigaflops double precision
Memory size: **3GB** total
Bandwidth: **150** GB/sec

Small changes, big speed-up



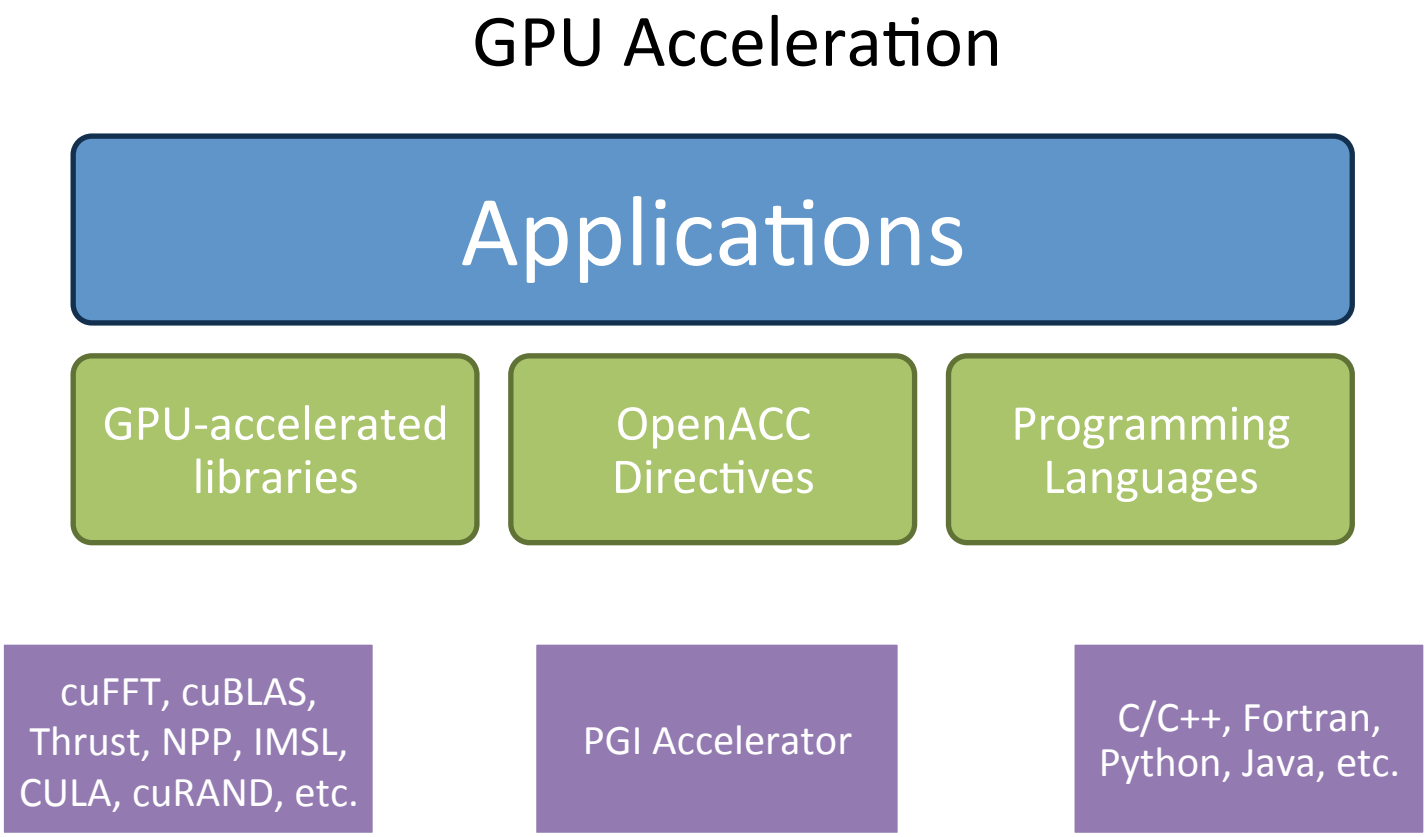
Fastest performance for learning



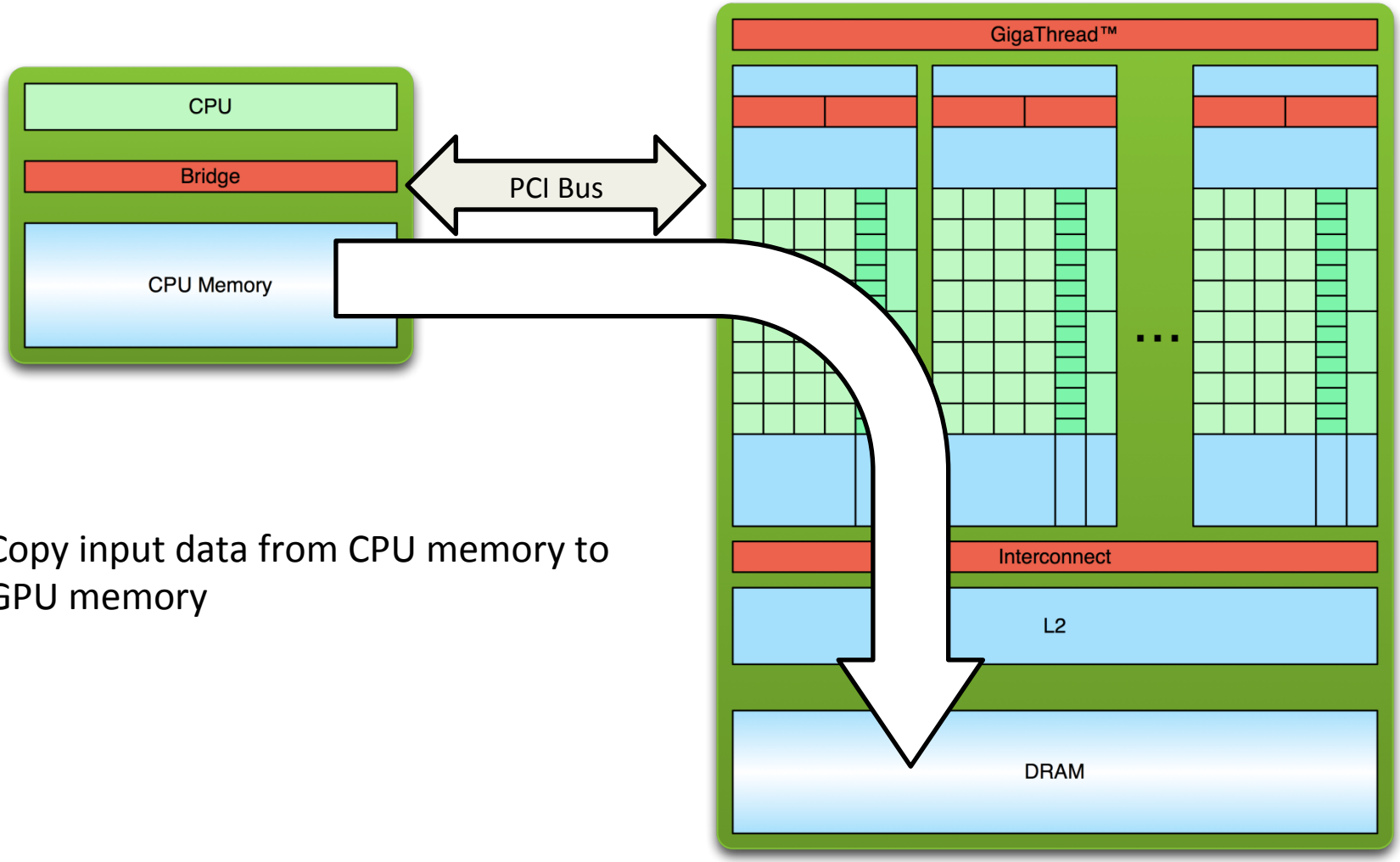
Definition of a GPU

- GPU – graphics processing unit
- Originally designed as a graphics processor
- Nvidia's GeForce 256 (1999) – first GPU
  - single-chip processor for mathematically-intensive tasks
  - transforms of vertices and polygons
  - lighting
  - polygon clipping
  - texture mapping
  - polygon rendering

How to use GPUs ?

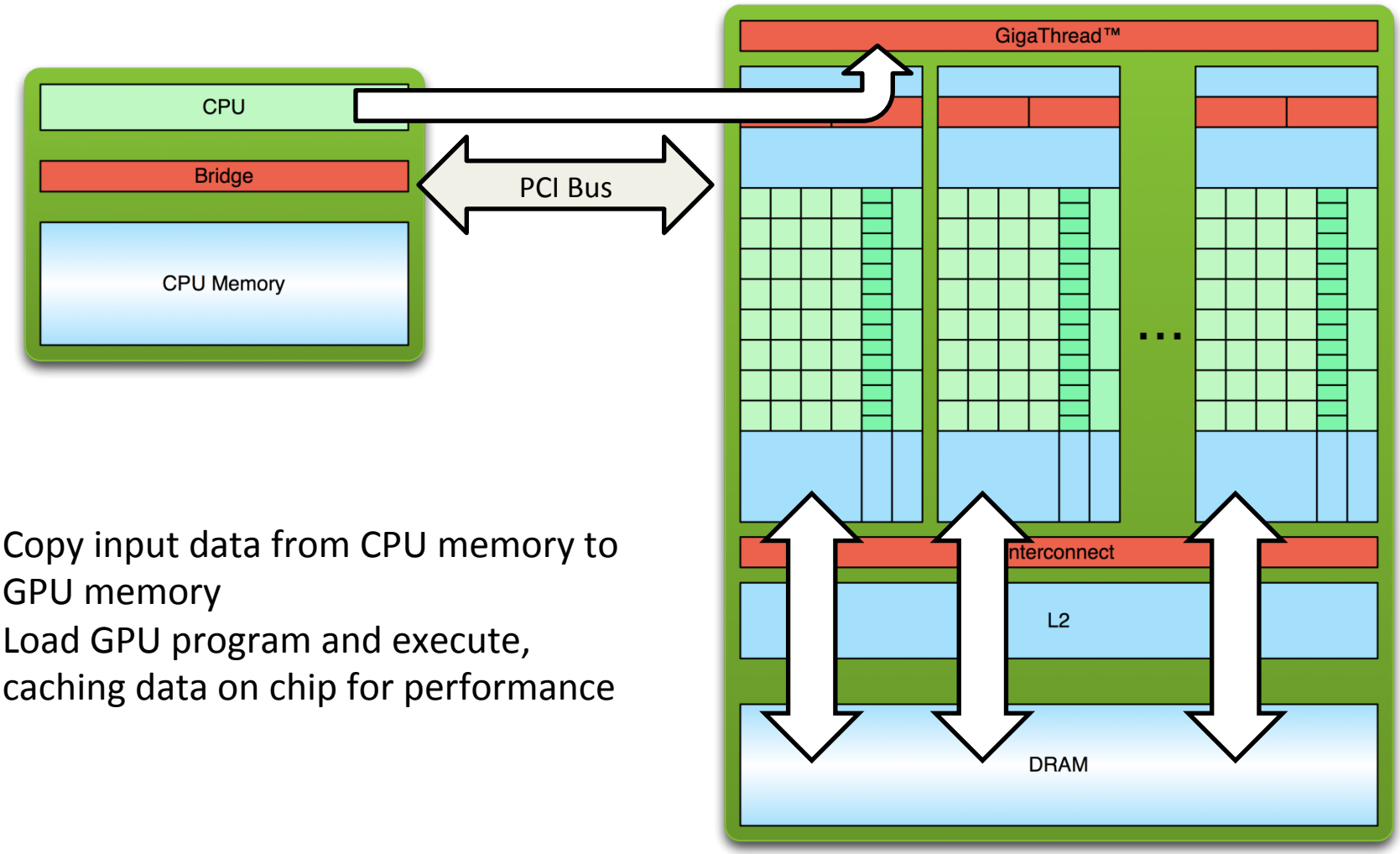


Usual processing flow



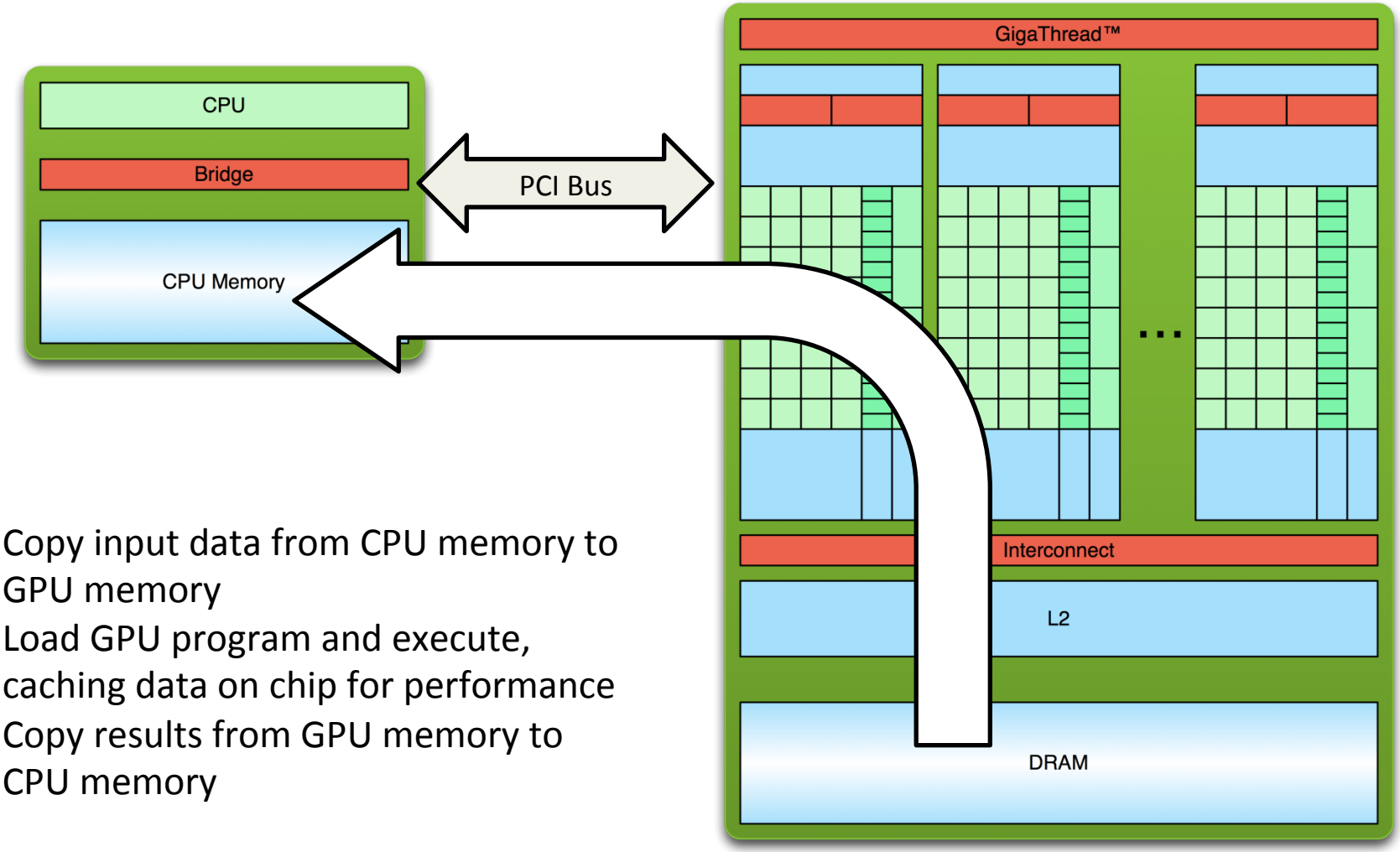
1. Copy input data from CPU memory to GPU memory

Usual processing flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

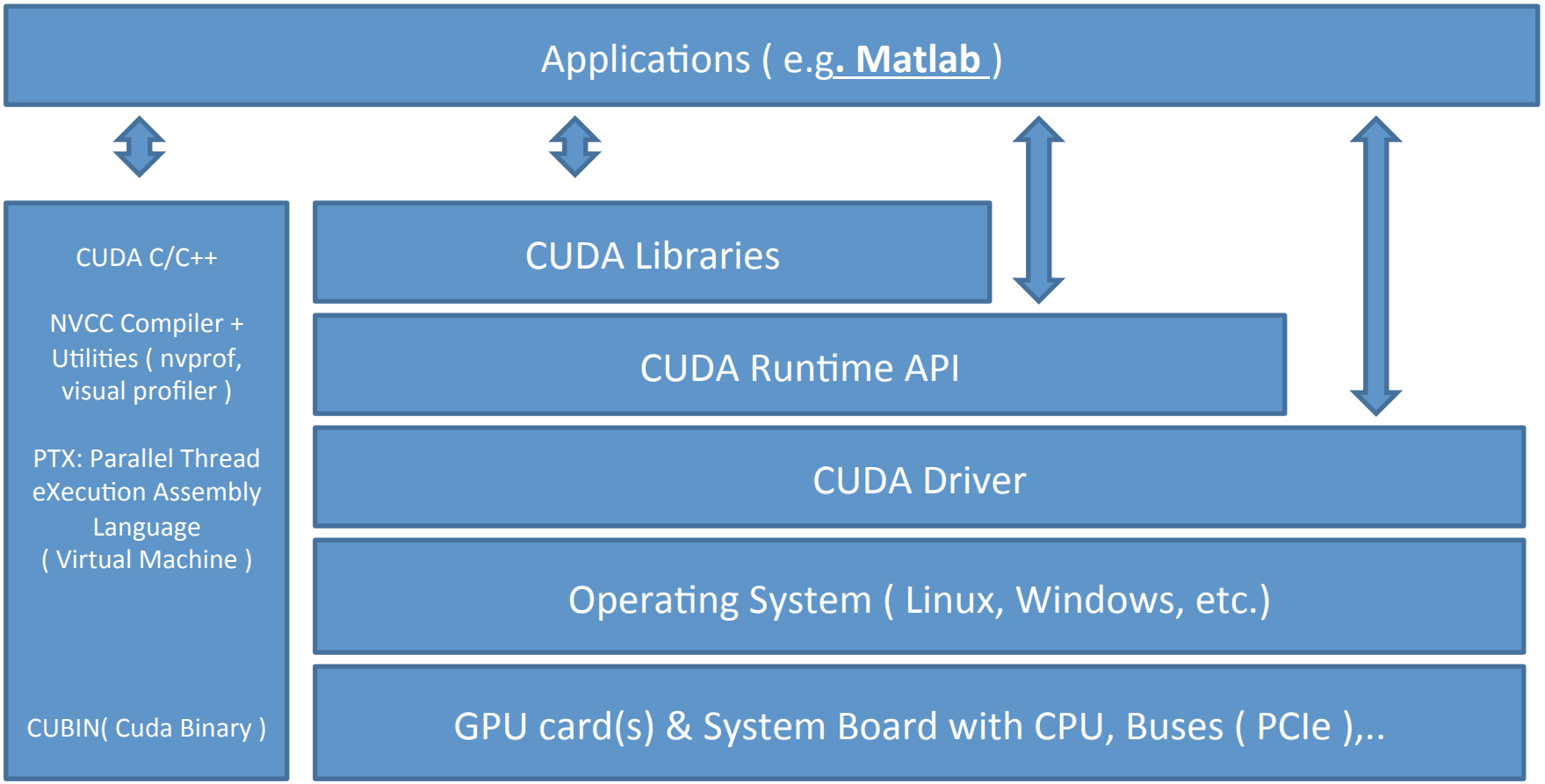
Usual processing flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Software architecture

CUDA: Compute Unified Device Architecture



MATLAB GPU Toolbox (CUDA)

Use GPUs with MATLAB through Parallel Computing Toolbox

- GPU-enabled MATLAB functions such as fft, filter, and several linear algebra operations
- GPU-enabled functions in toolboxes
- CUDA kernel integration in MATLAB applications, using only a single line of MATLAB code

```
A=rand(2^16,1);

B=fft(A);
```

```
A=gpuArray(rand(2^16,1));

B=fft(A);
```

GPU Acceleration in MATLAB

- Build-in functions
  - many Matlab functions support GPU acceleration natively
- arrayfun
  - specific element-wise processing
- CUDA kernels
  - write ".cu" files
  - compile to ".ptx" (parallel thread execution)
  - run using feval

MATLAB GPU Toolbox (CUDA)

- gpuDevice(#)
- gpuDeviceCount()
- reset(gpuDevice(#))
- wait()
- bsxfun()
- gpuArray()
- gather()
- arrayfun()
- existsOnGPU()
- parallel.gpu.CUDAKernel()
- feval
- setConstantMemory
- Many GPU enabled built-in functions: e.g. fft, …. Check with:
  - methods('gpuArray')

Basic usage

- Send data to GPU
  - either allocate there or transfer from workspace
  - **Function gpuArray()**
- Run Matlab functions
  - GPU acceleration is used automatically
  - **(cf. Matlab documentation)**
- Retrieve the output data
  - **Function gather()**

GPUArray class

`parallel.gpu.GPUArray`

- main data class for GPU computations
- stored in the GPU memory
- create directly using static methods

`zeros`	`nan`	`eye`	`rand`	`linspace`
`ones`	`true`	`colon`	`randi`	`logspace`
`inf`	`false`		`randn`	

- copy from existing data

```
gpuArray(img)
```

GPUArray class

- Supported data types:

  (u)int8, (u)int16, (u)int32, (u)int64, single, double, logical

  – determine the type using

  ```
  classUnderlying(gpuVar)
  ```

- Retrieve the data using

  ```
  workspaceVar = gather(gpuVar)
  ```

Simple example (linear)

- Solve system of linear equations (Ax = b)

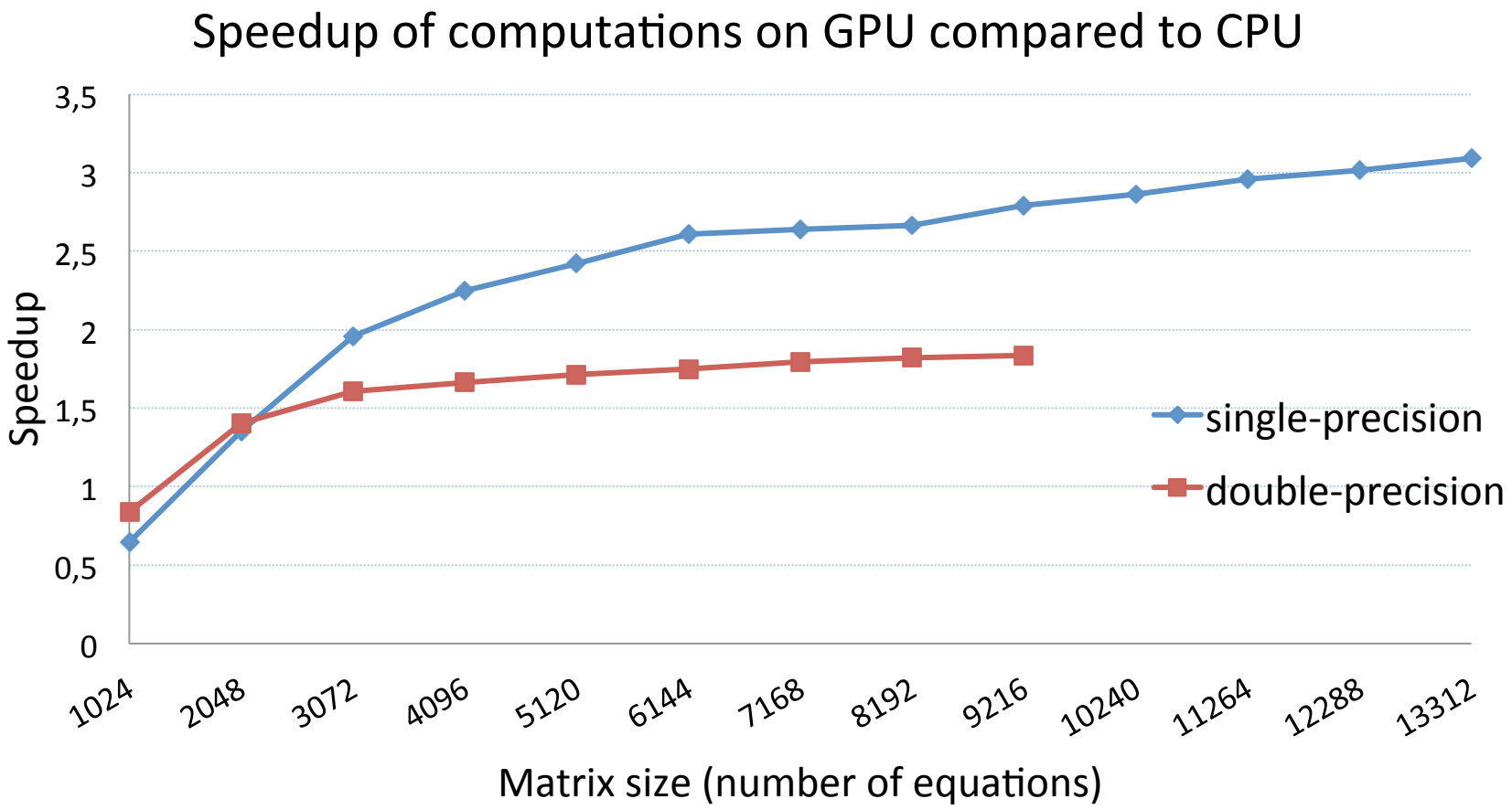
```
A = gpuArray(A);
b = gpuArray(b);
x = A\b;
x = gather(x);
```

Simple example (convolution)

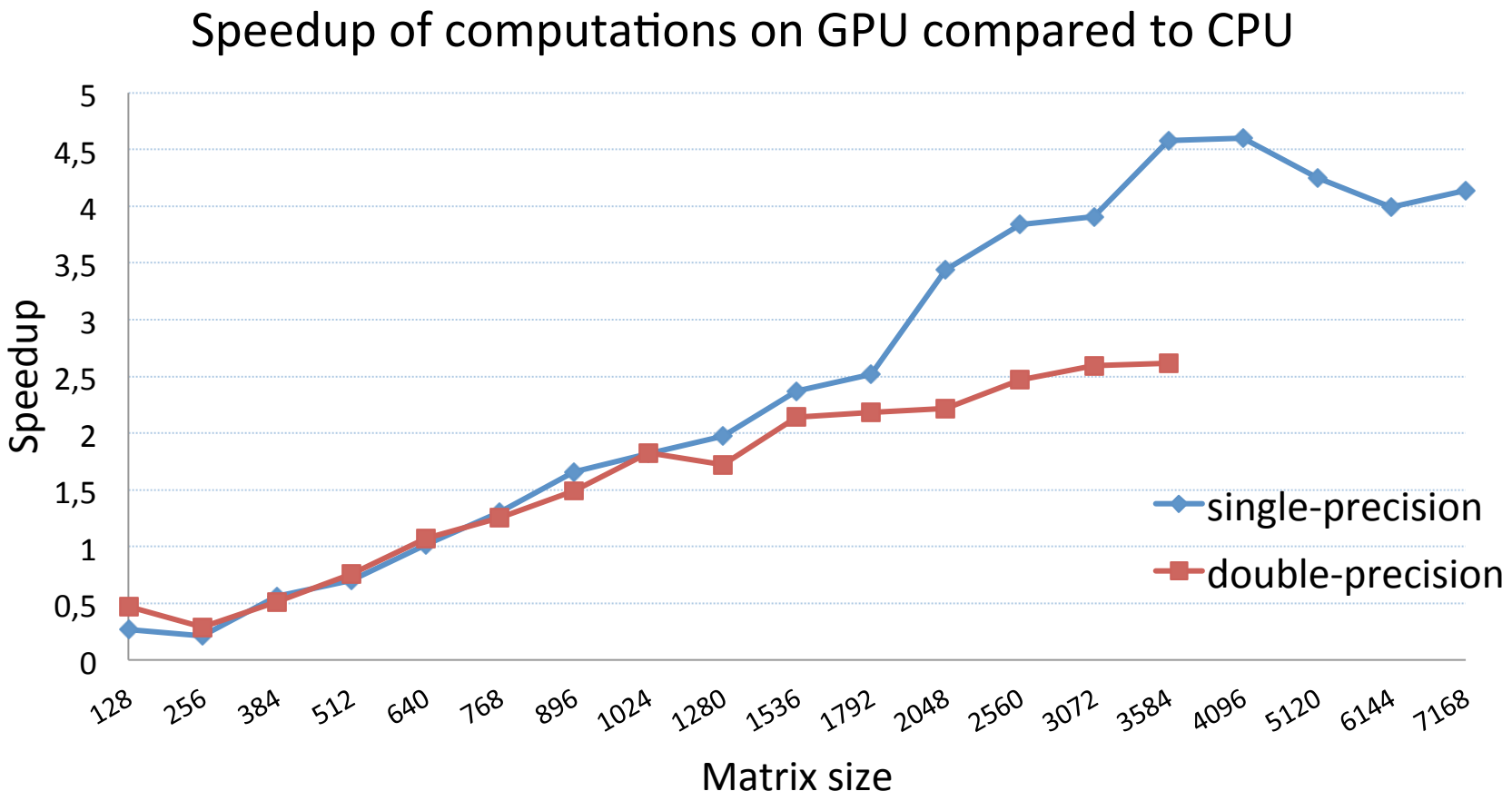
- Compute convolution using FFT

```
img = gpuArray(img);
msk = padarray(msk,size(img)-size(msk),0,'post');
msk = gpuArray(msk);
I = fft2(img);
M = fft2(msk,size(img,1),size(img,2));
res = real(ifft2(I.*M));
res = gather(res);
```

Linear system solution



GPU Acceleration in MATLAB



GPU Computing – CUDA in C/C++

Programming language extension to C/C++

Designed for efficient general purpose computation on GPU.

```
__global__ void kernel(float* x, float* y, float* z, int n){
        int idx= blockIdx.x * blockDim.x + threadIdx.x;
        if(idx <  n) z[idx] = x[idx] * y[idx];
}
int main(){
    ...
    cudaMalloc(...);
    cudaMemcpy(...);
    kernel <<<num_blocks,  block_size>>> (...);
    cudaMemcpy(...);
    cudaFree(...);
    ...
}
```

Hello world

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

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

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

Hello world

```
__global__ void mykernel(void) {
}

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

- Two new syntactic elements…

Hello world

```
__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

```
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

```
__global__ void mykernel(void){
}

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

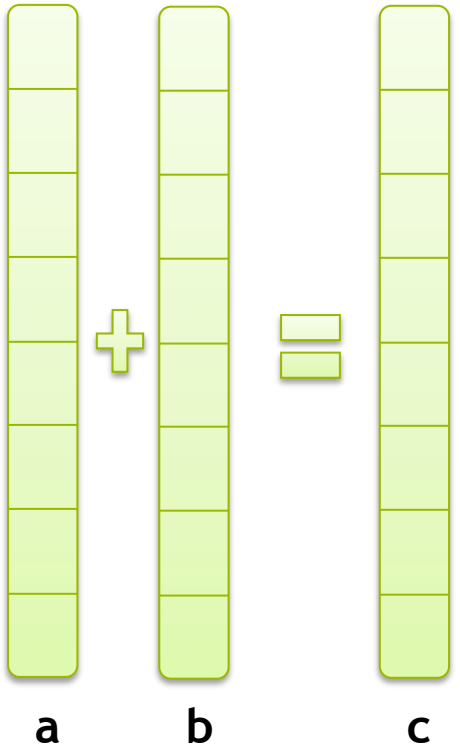
Output:

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

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

Parallel programming

- 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 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 device

- Note that we use pointers for the variables

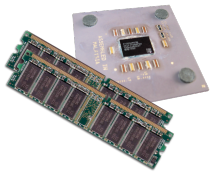
```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- `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 device

- 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 device

```
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 device

```
    // 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;
}
```

Addition on device

- 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

Addition on 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`

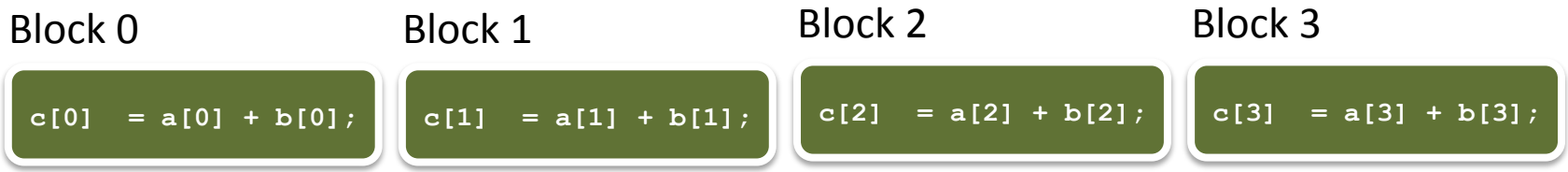
```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

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

Addition on 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:



Addition on device

- 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()…

Addition on 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);
```

Addition on device

```
    // 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

- 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

- 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