Compute Unified Device Architecture (CUDA)

What is CUDA used for?
- ↳ Graphics and rate tracing
- ↳ Deep learning
- ↳ and more

* The point of writing GPU kernel is to run something faster

Chapter 1 Deep Learning Ecosystem

* **Research**: PyTorch, TensorFlow, MLX (apple devices)
* **Production**: Training and Inference
  - ↳ vLLM (inference only)
  - ↳ Tensor RT
  - ↳ Triton (OpenAI)
* **Low Level**: CUDA, ROCm
* **Inference For Edge Computing & Embedded Systems**:
  - ↳ Edge Computing: low-latency and highly efficient local computing in the context of real-world distributed systems
  - ↳ PyTorch Moblie, TensorFlow Lite
* **Cloud Computing**:
  - ↳ AWS, Google Cloud, Azure, VASTAI
* **Compilers**:
  - ↳ XLA, LLVM, NVCC

Quick C++ Review

Basic Structure

```
using namespace std;   //standard library
int main()
{
    //code here
    data-type example;   //variable declaration
    cin >> example;  // input
    cout << example << endl;  //output (print)
    if (condition 1{    //if and nested if
        if (condition 2){
        }
    }
    switch (expression){
        case value1:
            //code here if matches value1
            break;
        case . . .
        default:
            break;
    }
    for(initalization; test; update){
    }
    dataType array_name[size];  //multidimensional dataType array_name[size1]...[sizeN]
    vector <data-type> vector_name  /*resize automatically, #include <vector>,
        push_back() insert elem @ end, pop_back() removes elem from end, clear() remove all elem, empty() checks if empty,
        at(i) access elem at index i, front() access first elem, back() access last elem, erase() removes specific elem at position
        */
    return 0;
}
```

Quick C++ Review Continued...

References: points to an object of a given class, letting you access the value of an object

```
int var = 12;
int &ref = var;
```

ref is now a reference to var

Pointer: stores memory address of another var

```
int i = 3;
int *ptr = &i; // pointer to i (stores address of i)
```

Functions:

```
return_type func_name(param); // function declaration
return_type func_name(param) {
    // code here for func definition
}
```

String Functions:

* substring: string substr(size_t first_char_copy, size_t length);
* append() adds to end of string
* compare()
* empty() checks if empty

OOP

* Class & objects:
  - Class: blueprint for objects behavior
  - objects: instance or var of class
* Encapsulation: wrap up data & methods together within a single entity (classes)
* Abstraction: shows only necessary details & hides internal
* Inheritance: deriving properties of Parent class to child class
* Polymorphism: different functionalities to functions with same name (overload, override)
* File handling: open(), getline(), << to write file after opening

GPUs Background Info

* CPU: Central Processing Unit → host executes functions
  - ↳ general purpose
  - ↳ high clock speed
  - ↳ few cores
  - ↳ high cache
  - ↳ low latency - (Complete it fast) (can't do as much comparatively operations (math) per second)
  - ↳ Low throughput
  - -minimize time of one
  - - metric: latency in seconds

* Graphics Processing Unit → device executes kernels
  - ↳ specialized
  - ↳ low clock speed
  - ↳ many cores
  - ↳ Low caches
  - ↳ high latency
  - ↳ high throughput
  - ↳ maximize throughput
  - ↳ metric: throughput in tasks per second

  - • Tensor Processing Unit (TPU)
    - ↳ Specialized GPU for deep learning algorithms (matrix multiplication)
  - • FPGA: Field Programmable Gate Array
    - ↳ specialized hardware that can be reconfigured to perform specific tasks
    - ↳ low (very) latency
    - ↳ very high throughput
    - ↳ very high power consumption
    - ↳ very high cost

Typical CUDA Program

1. CPU allocates CPU memory
2. CPU copies data to GPU
3. CPU launches kernel on GPU (processing is done here)
4. CPU copies results from GPU back to CPU to do something useful with it

*GPU Kernel:

↳ smallest unit of execution → mini program that runs independently but shares resources

↳ a function that runs on the GPU

↳ runs many threads in parallel (1000s at once) (each thread executes same func. but on different data

↳

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

- `__global__` — keyword tells CUDA this func runs on GPU and can be called from CPU
- `void add(int *a, int *b, int *c)` — name and parameters of function
- `{` — each block has a unique index
- `blockIdx.x` — each GPU thread has a unique thread index within its block
- `blockDim.x` → num of threads per block
- `index = blockIdx.x * blockDim.x + threadIdx.x;` → computes global thread ID, so each thread knows which element it should work on
- `c[index] = a[index] + b[index];` → vector addition in parallel of 1000s of GPU threads
- `}`

↳ Block → a group of threads that can cooperate using shared memory

↳ Grid → full set of all blocks launched for one kernel call

↳ Launch: call kernel from CPU side

e/g

```
int N=1000;
add<<<(N+255)/256, 256>>>(d_a, d_b, d_c);
```

launches: (N+255)/256 blocks with 256 threads