

GPU Programming

Week # 14

CPU

Single Core CPUs

- Flexible
- Performant
- How to squeeze more functionality?
 - Power Hungry
 - Memory CPU speed disparity
 - Instruction Level Parallelism
 - Pipelining
 - Super Scalar

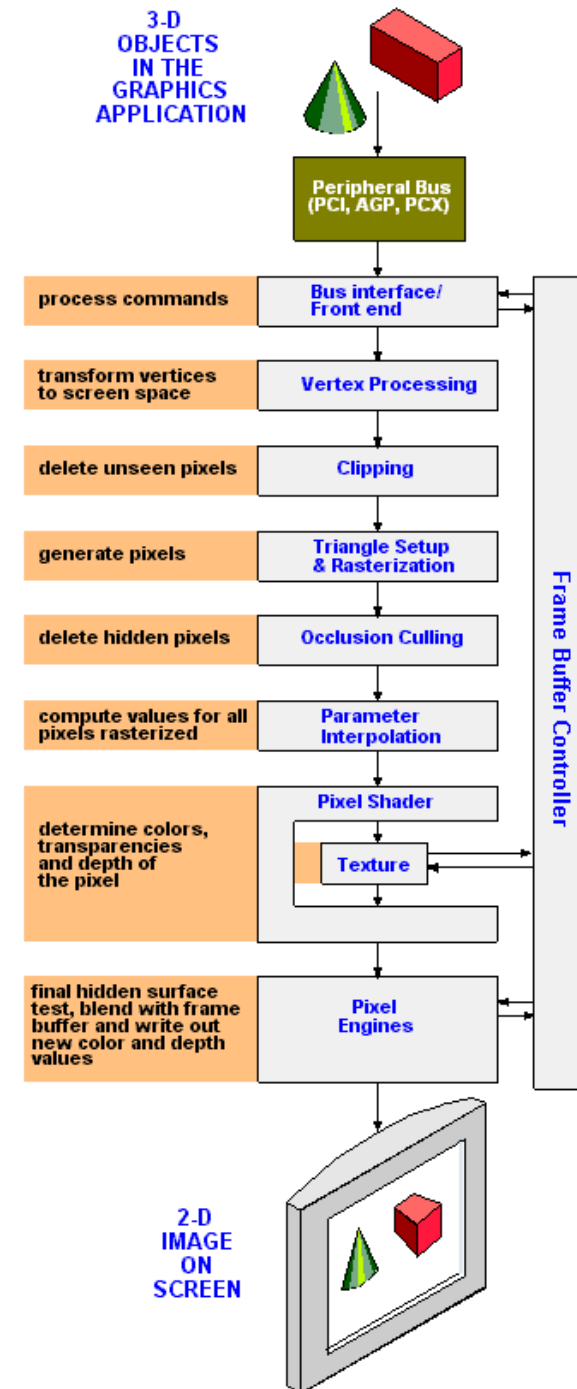
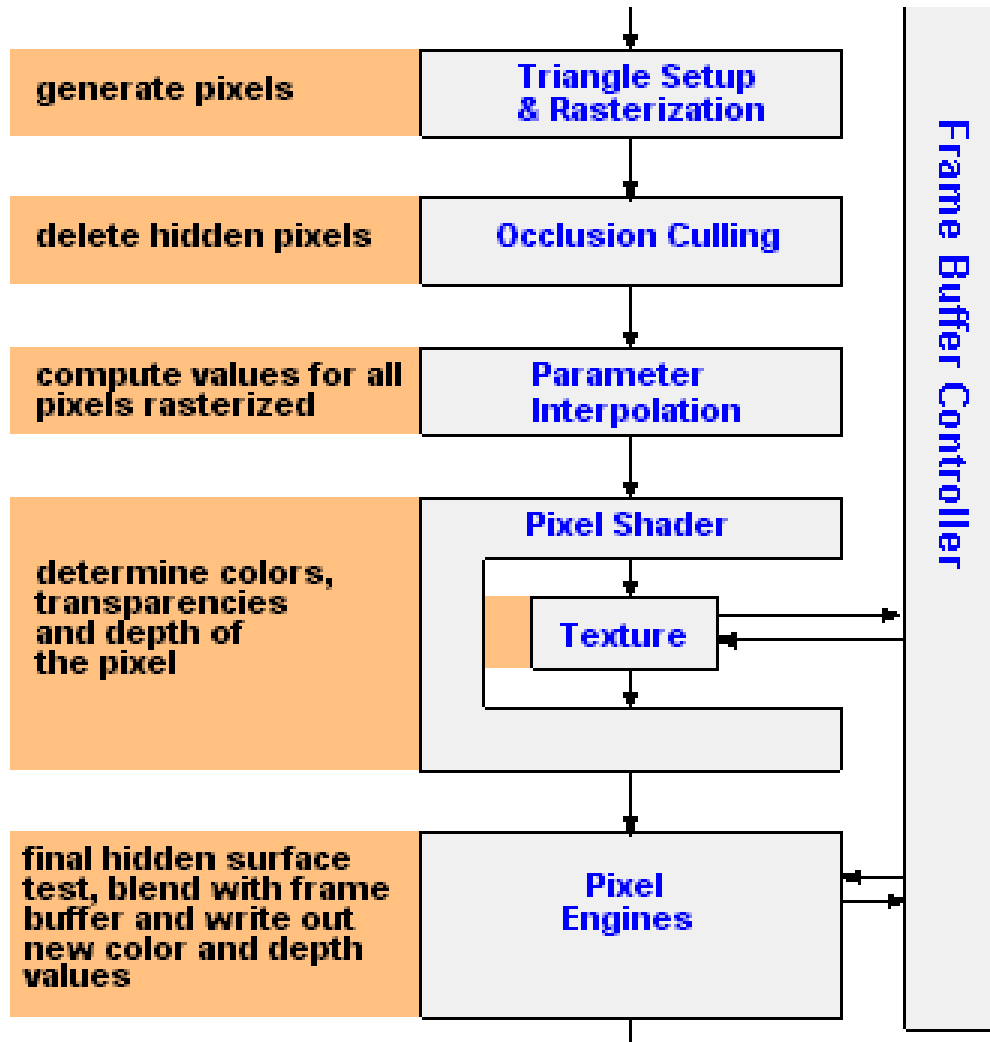
"Power Wall + Memory Wall + ILP Wall = Brick Wall"

Multi-Cores CPUs

- Instead of adding complexity to a single core we scale cores in the same dimensions of the silicon
- Current Multi-Cores (Discuss)
- Do we need more?
 - Vector Processing Units (VPUs),
 - Graphic Processing Units (GPUs),
 - Associative Processing Units (APUs),
 - Tensor Processing Units (TPUs)
 - Field Programmable Gate Arrays (FPGAs),
 - Quantum Processing Units (QPU)

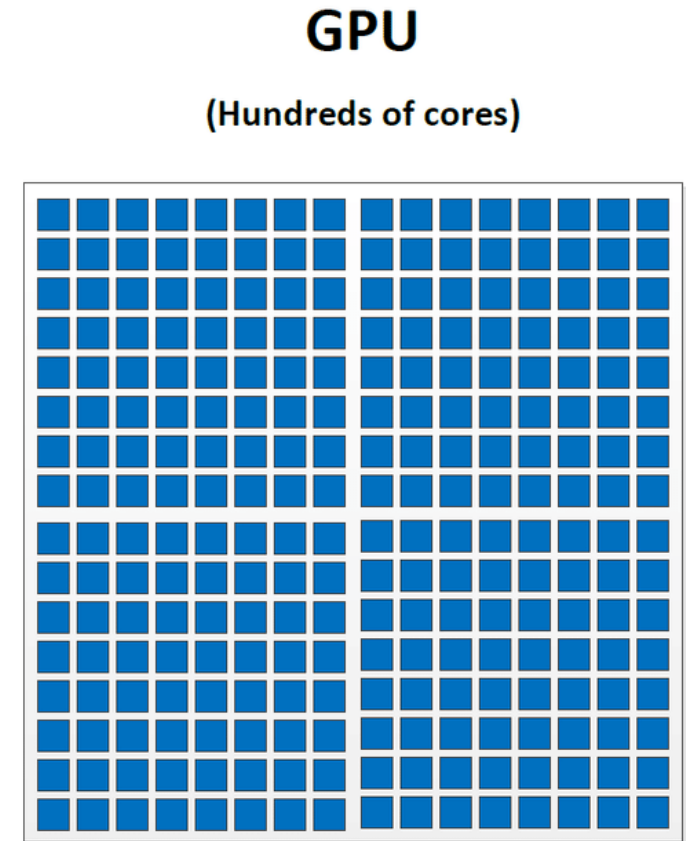
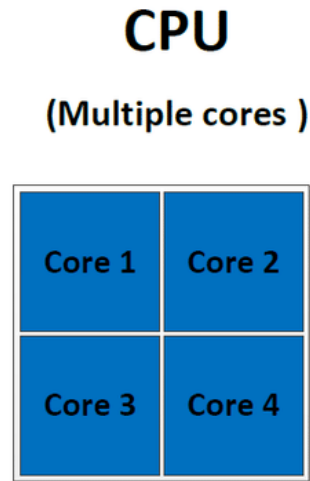
Graphic Processing

Not part of syllabus



Graphic Processing Units (GPUs)

- **More HW for Computation**
 - **Many Cores**
- **More power efficient**
- **Restricted Programming Model**
- GPUs were originally designed to accelerate the rendering of 3D graphics. Over time, they became more flexible and programmable, enhancing their capabilities.
- This allowed graphics programmers to create more interesting visual effects and realistic scenes with advanced lighting and shadowing techniques.
- Other developers also began to tap the power of GPUs to dramatically accelerate additional workloads in high performance computing (HPC), deep learning, and more.



Latency vs Throughput

Latency

- Minimize time of one task
- Metric: seconds
- Focus of CPU

Throughput

- Maximize stuff per time
- Metric: jobs/hour, pixels/sec
- Focus of GPUs

Latency vs Throughput

- Latency-Amount of time to complete a task(time , seconds)
- Throughput-Task completed per unit time(Jobs/Hour)

Your goals are not aligned with post office goals

Your goal: Optimize for **Latency**
(want to spend a little time)

Post office: Optimize for **throughput**
(number of customers they serve per a day)

CPU: Optimize for **latency**(minimize the time elapsed of one particular task)

GPU: Chose to Optimize for **throughput**



Integrated Graphics Processing Unit

The majority of GPUs on the market are actually integrated graphics. So, what are integrated graphics and how does it work in your computer? A CPU that comes with a fully integrated GPU on its motherboard allows for thinner and lighter systems, reduced power consumption, and lower system costs.

Intel® Graphics Technology, which includes Intel® Iris® Plus and Intel® Iris® X^e graphics, is at the forefront of integrated graphics technology. With Intel® Graphics, users can experience immersive graphics in systems that run cooler and deliver long battery life.

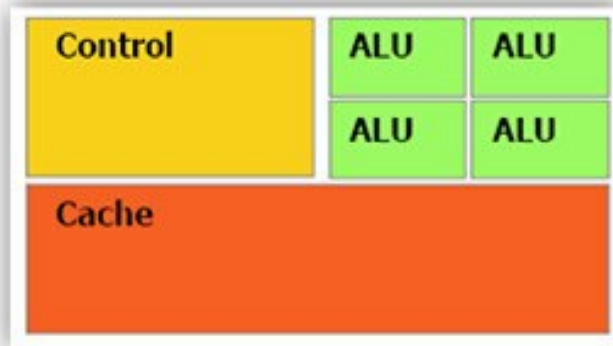
Discrete Graphics Processing Unit

Many computing applications can run well with integrated GPUs. However, for more resource-intensive applications with extensive performance demands, a discrete GPU (sometimes called a dedicated graphics card) is better suited to the job.

These GPUs add processing power at the cost of additional energy consumption and heat creation. Discrete GPUs generally require dedicated cooling for maximum performance.

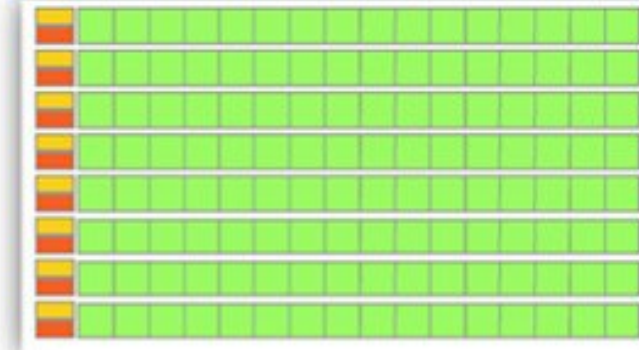
<https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html>

CPU



- * Low compute density
- * Complex control logic
- * Large caches (L1\$/L2\$, etc.)
- * Optimized for serial operations
 - Fewer execution units (ALUs)
 - Higher clock speeds
- * Shallow pipelines (<30 stages)
- * Low Latency Tolerance
- * Newer CPUs have more parallelism

GPU



- * High compute density
- * High Computations per Memory Access
- * Built for parallel operations
 - Many parallel execution units (ALUs)
 - Graphics is the best known case of parallelism
- * Deep pipelines (hundreds of stages)
- * High Throughput
- * High Latency Tolerance
- * Newer GPUs:
 - Better flow control logic (becoming more CPU-like)
 - Scatter/Gather Memory Access
 - Don't have one-way pipelines anymore

How GPU increases throughput?

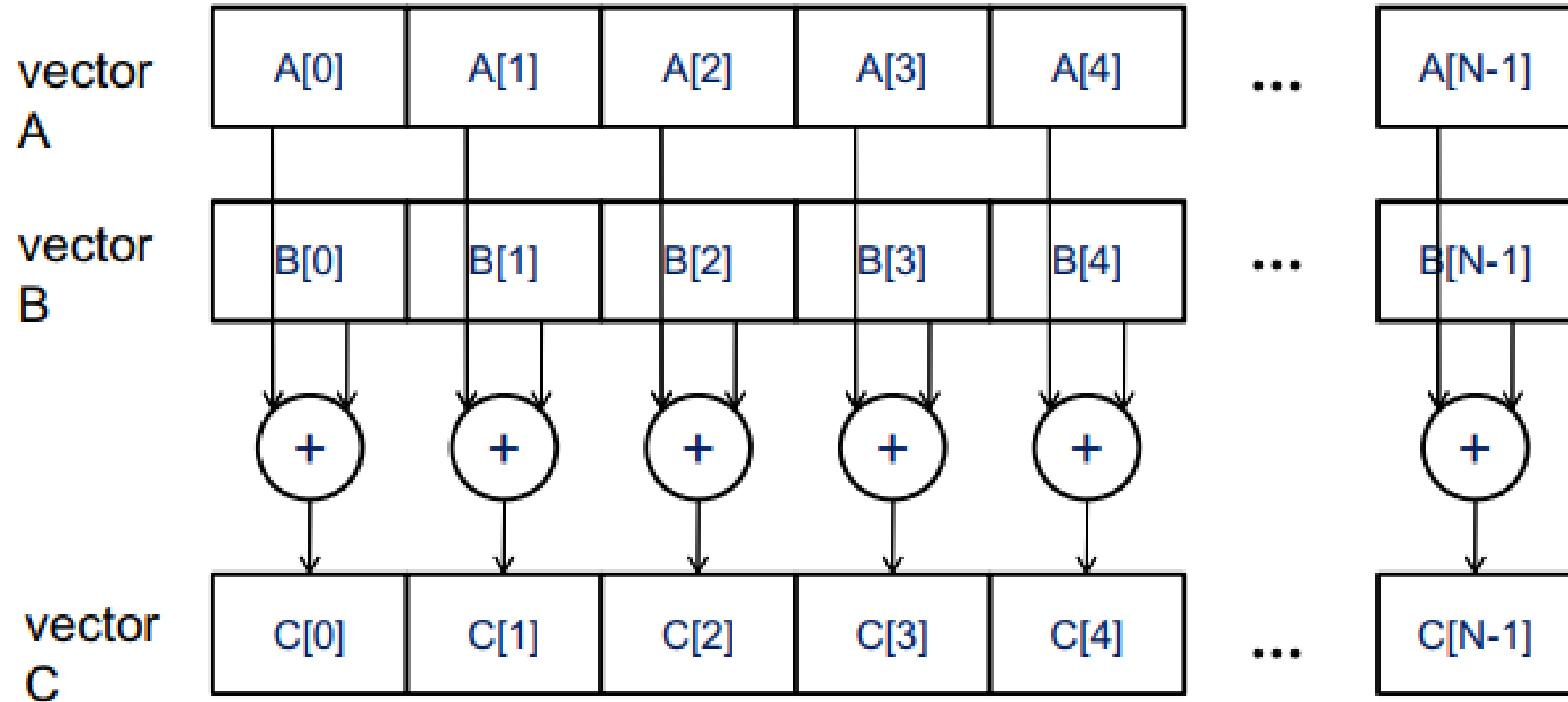


FIGURE 3.1

Data parallelism in vector addition.

TASK PARALLELISM VERSUS DATA PARALLELISM

Data parallelism is not the only type of parallelism widely used in parallel programming. Task parallelism has also been used extensively in parallel programming. Task parallelism is typically exposed through task decomposition of applications. For example, a simple application may need to do a vector addition and a matrix–vector multiplication. Each of these would be a task. Task parallelism exists if the two tasks can be done independently.

In large applications, there are usually a larger number of independent tasks and therefore a larger amount of task parallelism. For example, in a molecular dynamics simulator, the list of natural tasks includes vibrational forces, rotational forces, neighbor identification for nonbonding forces, nonbonding forces, velocity and position, and other physical properties based on velocity and position.

In general, data parallelism is the main source of scalability for parallel programs. With large data sets, one can often find abundant data parallelism to be able to utilize massively parallel processors and allow application performance to grow with each generation of hardware that has more execution resources. Nevertheless, task parallelism can also play an important role in achieving performance goals. We will be covering task parallelism later when we introduce CUDA streams.

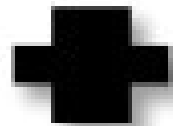
NVIDIA Tesla A100 with 54 Billion Transistors



Announced and released on May 14, 2020 was the Ampere-based A100 accelerator. With 7nm technologies, the A100 has 54 billion transistors and features 19.5 teraflops of FP32 performance, 6912 CUDA cores, 40GB of graphics memory, and 1.6TB/s of graphics memory bandwidth. The A100 80GB model announced in Nov 2020, has 2.0TB/s graphics memory bandwidth

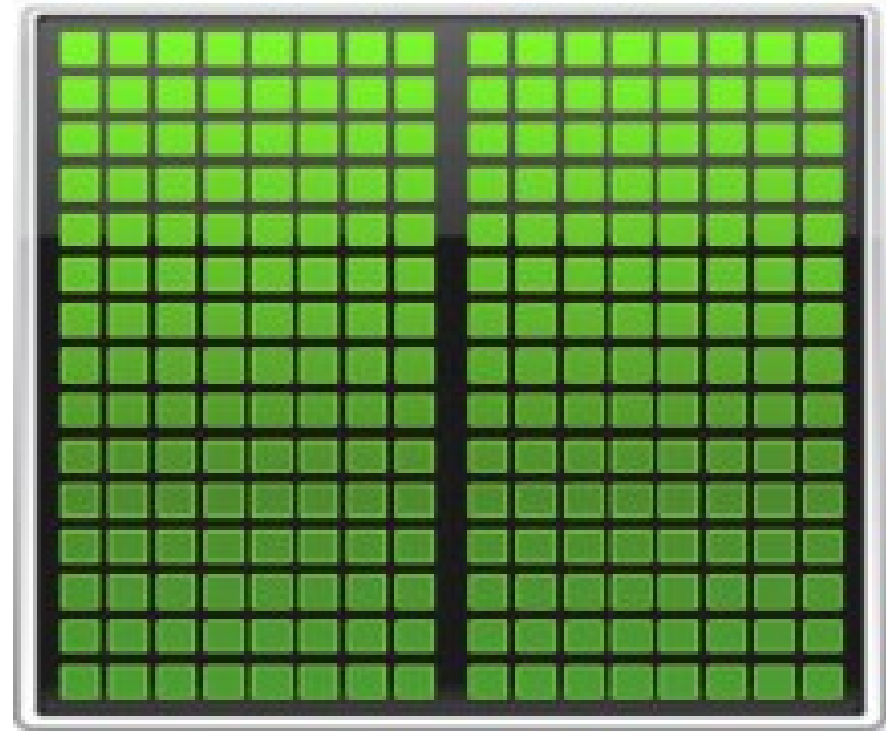
CPU

Optimized for
Serial Tasks



GPU Accelerator

Optimized for Many
Parallel Tasks



GPU Computing Applications

Libraries and Middleware





cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica
-------------------	---------------------------------------	---------------	---------------	-----------------------------	------------------------	-----------------------

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)
---	-----	---------	----------------------------	---------------	------------------------------



CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center

- CUDA Platform for NVIDIA GPUs
- OpenCL – An Open platform for Compute Accelerators
- OpenCL vs CUDA
- How GPU accelerate Compute?
- A typical CUDA program
 - Write CPU code in C/C++ normally
 - Declare and allocate GPU memory
 - Specify Kernel code in your program – It will run on GPU
 - Execute Kernel code on GPU
 - Different Kernels can exists in your program and activated in your code

NVIDIA CUDA platform

- CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs).
- With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.
- The CUDA Toolkit from NVIDIA provides everything you need to develop GPU-accelerated applications. The CUDA Toolkit includes GPU-accelerated libraries, a compiler, development tools and the CUDA runtime.



OpenCL™

OPEN STANDARD FOR PARALLEL PROGRAMMING OF HETEROGENEOUS SYSTEMS

OpenCL™ (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of diverse accelerators found in supercomputers, cloud servers, personal computers, mobile devices and embedded platforms. OpenCL greatly improves the speed and responsiveness of a wide spectrum of applications in numerous market categories including professional creative tools, scientific and medical software, vision processing, and neural network training and inferencing.

OpenCL vs CUDA

- You can write OpenCL programs which will function on Nvidia/AMD/Intel GPUs, AMD/Intel/ARM CPUs, FPGAs etc. and excellent compatibility across Windows, Linux and MacOS.
- If you use CUDA, you will have to use Nvidia GPUs only and re-write your code again in OpenCL or other language for other platforms.
- As long as you don't plan to use Nvidia-proprietary things like tensor / ray-tracing cores, OpenCL is just as fast as CUDA when properly optimized.
- A serious limitation of using CUDA and cause of serious waste of time in the long run.

1. Heterogeneous programming model

1.1 CPU (“host”)—normal program

1.2 GPU (“device”)—“kernel”

1.3 “Host launches kernels on device”

2. *n*VIDIA CUDA

2.1 One program for both

2.2 Part runs on CPU

2.3 Part runs on GPU

2.4 Compiler generates code for each

3. Memory

3.1 Separate (DRAM) memories

3.2 CPU “in charge”

3.3 Moving data CPU → GPU, GPU → CPU

3.4 Allocating GPU memory

- In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel.
- When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.

1. Typical CUDA program
 - 1.1 CPU allocates GPU memory (`cudaMalloc`)
 - 1.2 CPU copies data to GPU (`cudaMemcpy`)
 - 1.3 CPU launches kernel on GPU; parallelism expressed here
 - 1.4 CPU copies results from GPU (`cudaMemcpy`)
2. Still have computation/communication tradeoff!
3. Can launch lots of threads efficiently (1,000+)
4. Can run lots of threads in parallel

How GPU Accelerate Compute?

Big Idea

1. Kernel looks like a serial program; says nothing about parallelism
2. Write as if run on **one** thread
3. Will actually run on **many** threads (CPU says how many—hundreds, thousands, **millions!**)
4. Each thread knows its **thread index**; can do different parts of the computation

```
10 int
11 main(int argc, char **argv) {
12     const int ARRAY_SIZE = 64;
13     const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);
14
15     // Declare and initialize CPU arrays.
16     float h_in[ARRAY_SIZE];
17     float h_out[ARRAY_SIZE];
18     for (int i = 0; i < ARRAY_SIZE; i++) {
19         h_in[i] = float(i);
20     }
```

Declare and Allocate GPU memory

cuda-square.cu

```
22 // Declare and allocate GPU memory.  
23 float *d_in;  
24 float *d_out;  
25 cudaMalloc((void **)&d_in, ARRAY_BYTES);  
26 cudaMalloc((void **)&d_out, ARRAY_BYTES);
```

Specify Kernel code in your program

```
                                     cuda-square.cu
3  __global__ void
4  square(float *d_out, float *d_in) {
5      int idx = threadIdx.x;
6      float f = d_in[idx];
7      d_out[idx] = f * f;
8  }
```

Double Underscore or Dunder

Execute Kernel code on GPU

Copy Over - Compute - Copy Back

_____ cuda-square.cu _____

```
28 // Copy array to GPU.  
29 cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);  
30  
31 // Launch the kernel.  
32 square<<1, ARRAY_SIZE>> (d_out, d_in);  
33  
34 // Copy results back from GPU.  
35 cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);
```

Clean Up

_____ cuda-square.cu _____

```
43 // Release GPU memory.  
44 cudaFree(d_in);  
45 cudaFree(d_out);
```

Different GPU kernels can be coded and called from your code

```
10 int
11 main(int argc, char **argv) {
12     const int ARRAY_SIZE = 64;
13     const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);
14
15     // Declare and initialize CPU arrays.
16     float h_in[ARRAY_SIZE];
17     float h_out[ARRAY_SIZE];
18     for (int i = 0; i < ARRAY_SIZE; i++) {
19         h_in[i] = float(i);
20     }
```

```
22 // Declare and allocate GPU memory.
23 float *d_in;
24 float *d_out;
25 cudaMalloc((void **)&d_in, ARRAY_BYTES);
26 cudaMalloc((void **)&d_out, ARRAY_BYTES);
```

```
3 __global__ void
4 square(float *d_out, float *d_in) {
5     int idx = threadIdx.x;
6     float f = d_in[idx];
7     d_out[idx] = f * f;
8 }
```


Copy Over - Compute - Copy Back

cuda-square.cu

```
28 // Copy array to GPU.
29 cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);
30
31 // Launch the kernel.
32 square<<1, ARRAY_SIZE>> (d_out, d_in);
33
34 // Copy results back from GPU.
35 cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);
```

Clean Up

cuda-square.cu

```
43 // Release GPU memory.
44 cudaFree(d_in);
45 cudaFree(d_out);
```

```
if (err != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString( err),
        __FILE__, __LINE__);
    exit(EXIT_FAILURE);
}
```