Accelerated Programming

By Laxmikant Botkewar, SSDG,

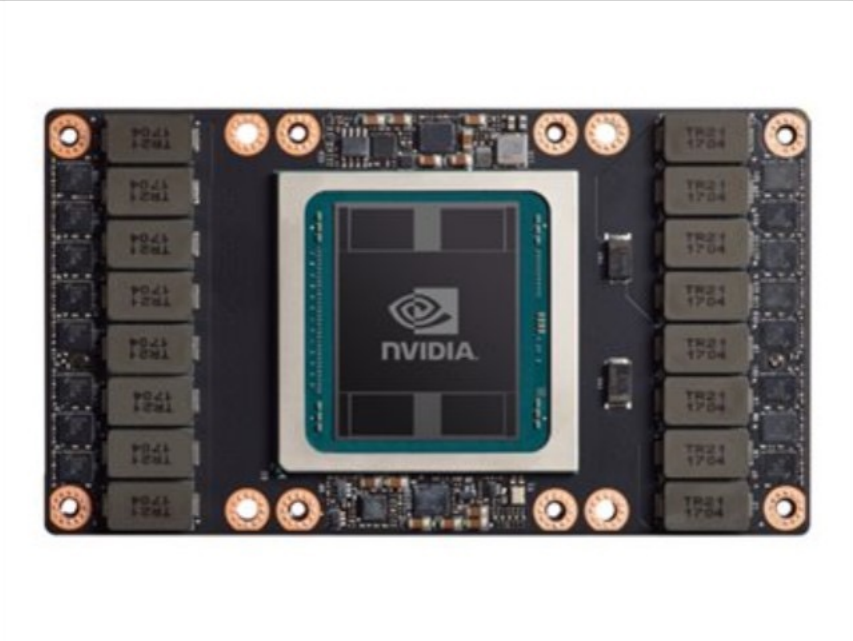
CDAC Bengaluru

Accelerator:

Hardware Acceleration is the use of computer hardware designed to perform specific functions more efficiently when compared to software running on a general-purpose central processing unit (CPU). Any transformation of data that can be calculated in software running on a generic CPU can also be calculated in custom-made hardware, or in some mix of both. This custom-made hardware is known as Accelerators.

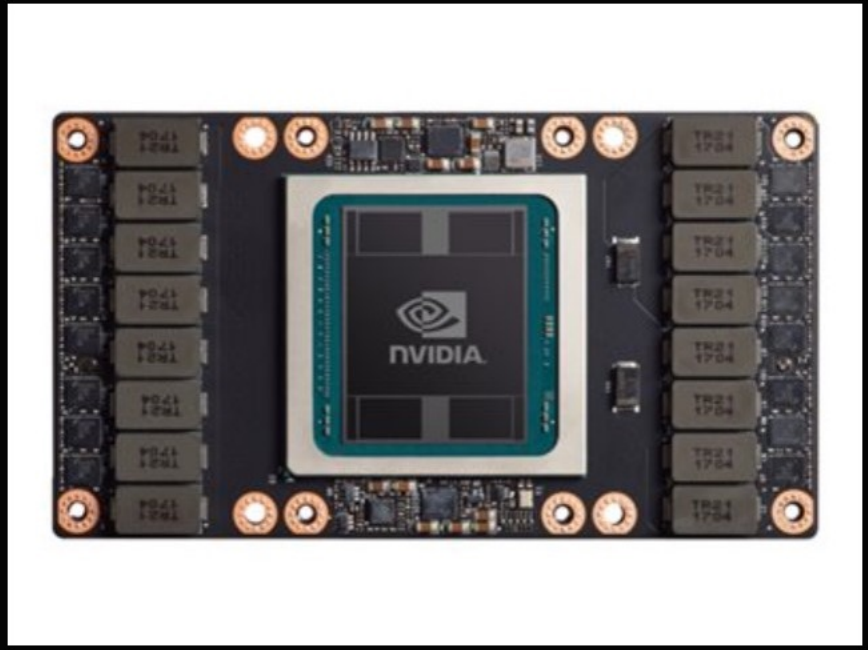
Types of Accelerator:

GPU:



A GPU (Graphics Processing Unit) accelerator is a hardware component designed to offload and accelerate computationally intensive tasks from the central processing unit (CPU). This frees up the CPU for other tasks, ultimately improving overall system performance. Originally created for accelerating graphics rendering in video games and other visually demanding applications, GPUs have become increasingly powerful and versatile, finding applications in a wide range of fields beyond graphics.

GPU:



- GPUs are used to accelerate scientific simulations and other data-intensive HPC applications.
- VR and gaming, enabling high-resolution graphics and smooth gameplay.
- Image recognition, object detection, and video analysis.
- Video editing and encoding, allowing for faster and more efficient video processing.

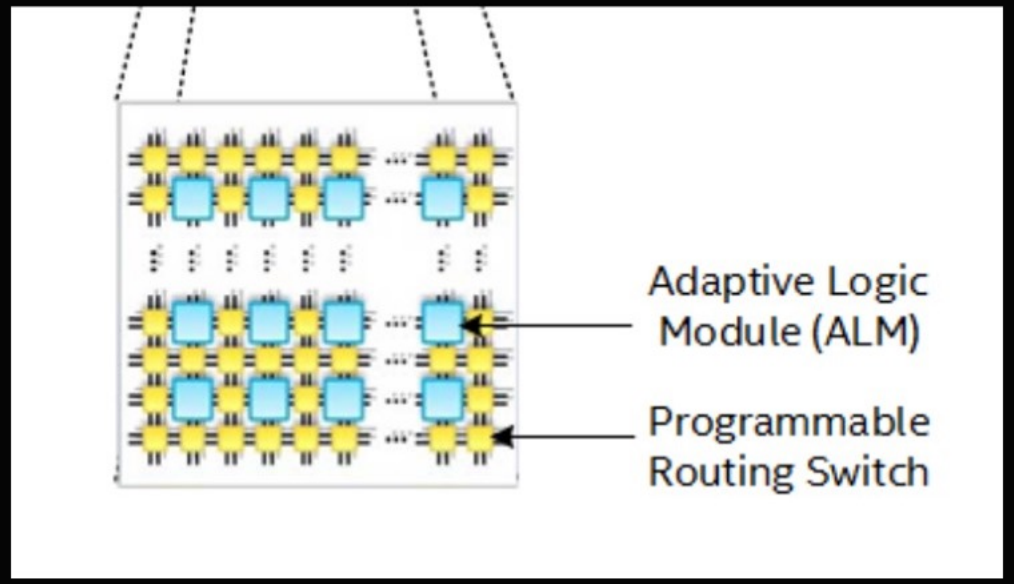
FPGA:

An FPGA accelerator is a hardware device built using Field Programmable Gate Arrays (FPGAs) to accelerate specific tasks or algorithms. Think of it as a specialized co-processor that offloads computationally intensive workloads from the main CPU, allowing it to focus on other tasks.

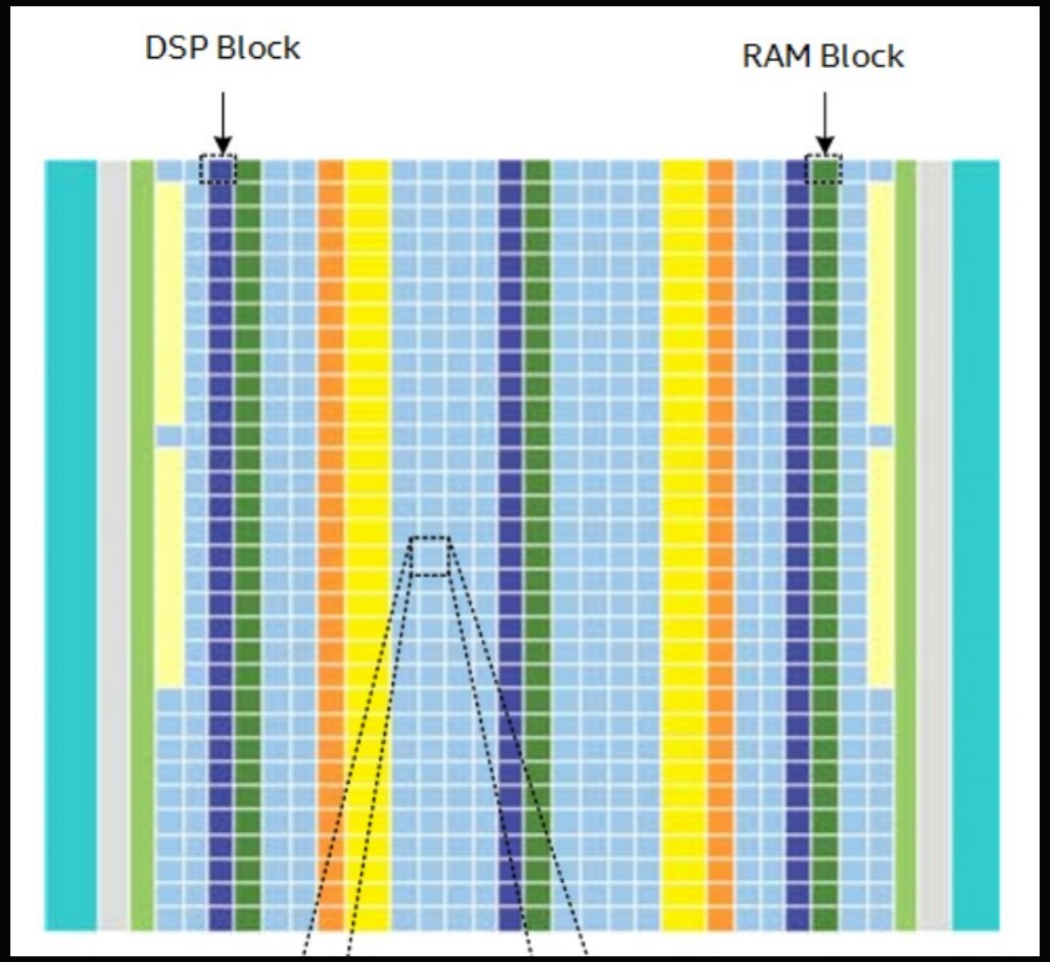
FPGA:

- HPC applications, such as financial modeling and scientific simulations.
- Packet processing, encryption/decryption, and network traffic management.
- Financial trading and algorithmic trading, where low latency is crucial.

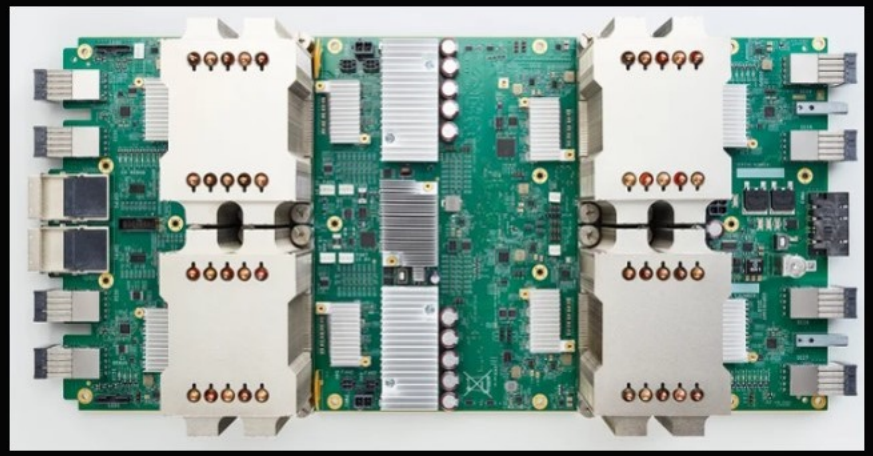
FPGA Architecture:



FPGA Architecture:

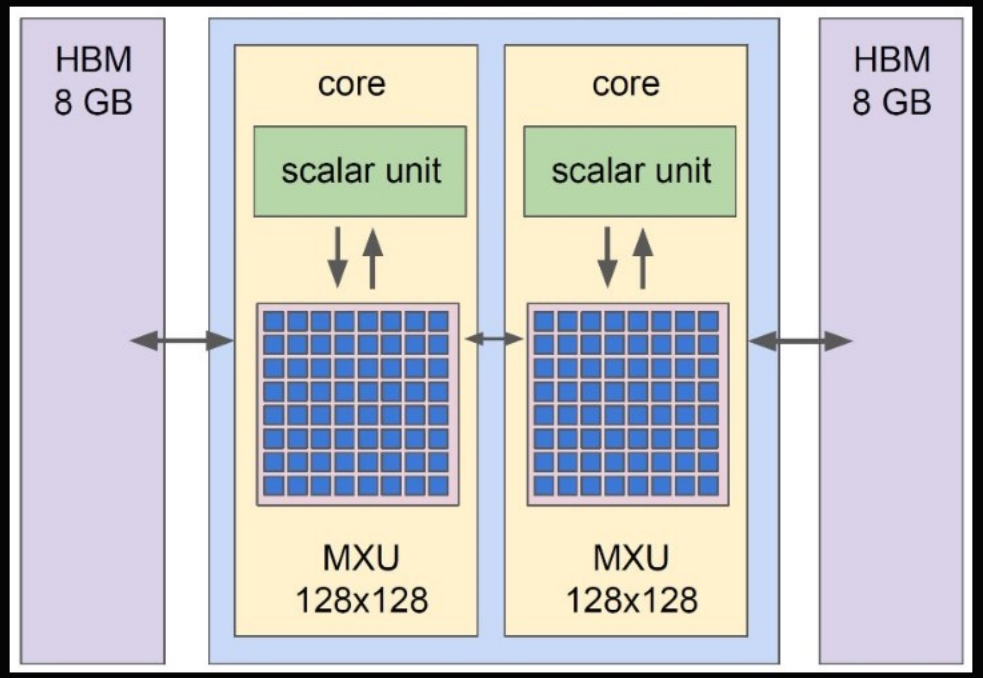


TPU:



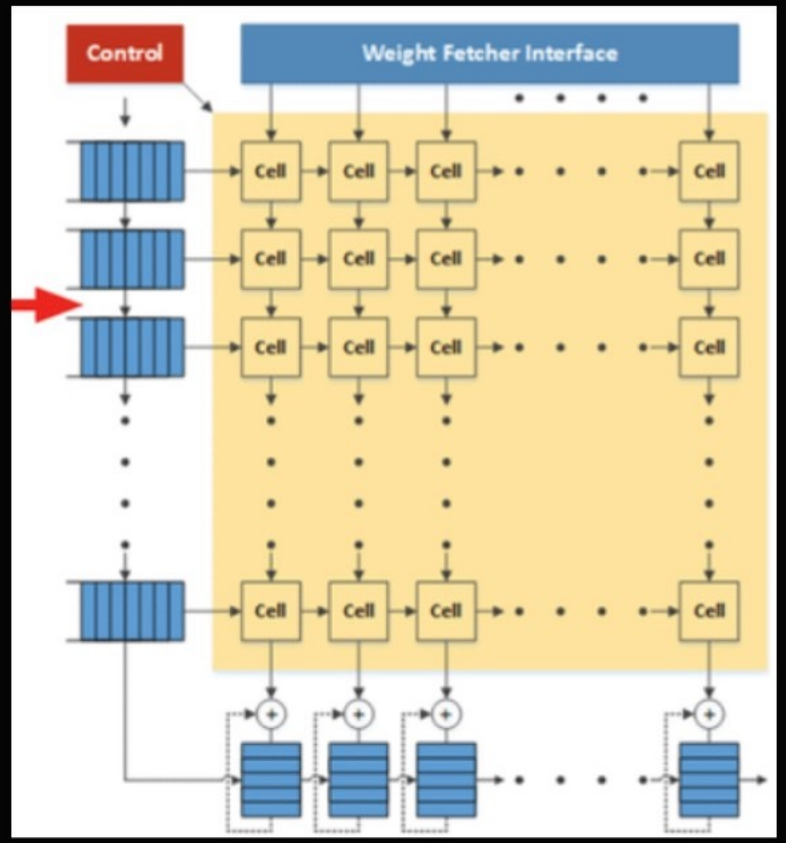
A TPU (Tensor Processing Unit) accelerator is a specialized hardware component designed by Google for accelerating machine learning (ML) workloads. TPUs are custom-designed ASICs (Application-Specific Integrated Circuits) optimized for the specific needs of ML algorithms. They offer several advantages for ML workloads like High Performance, Low Power Consumption and Cost Effectiveness.

TPU:



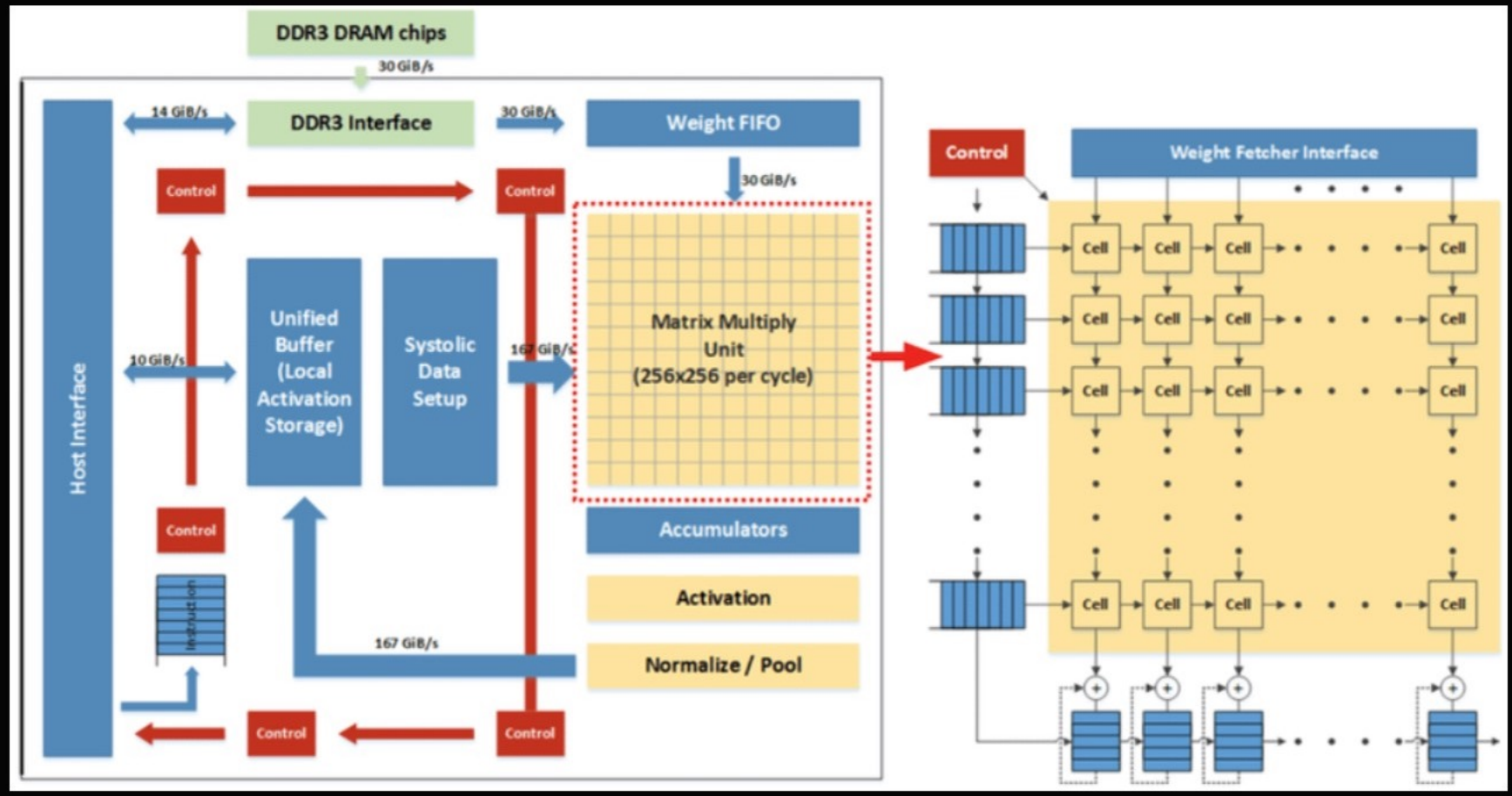
- TPUs are ideal for Training and inference of large-scale ML models.
- Edge Computing: Google's Edge TPUs are designed for resource-constrained environments, allowing for on-device ML processing in applications like mobile devices and IoT devices.

TPU's Core Component:



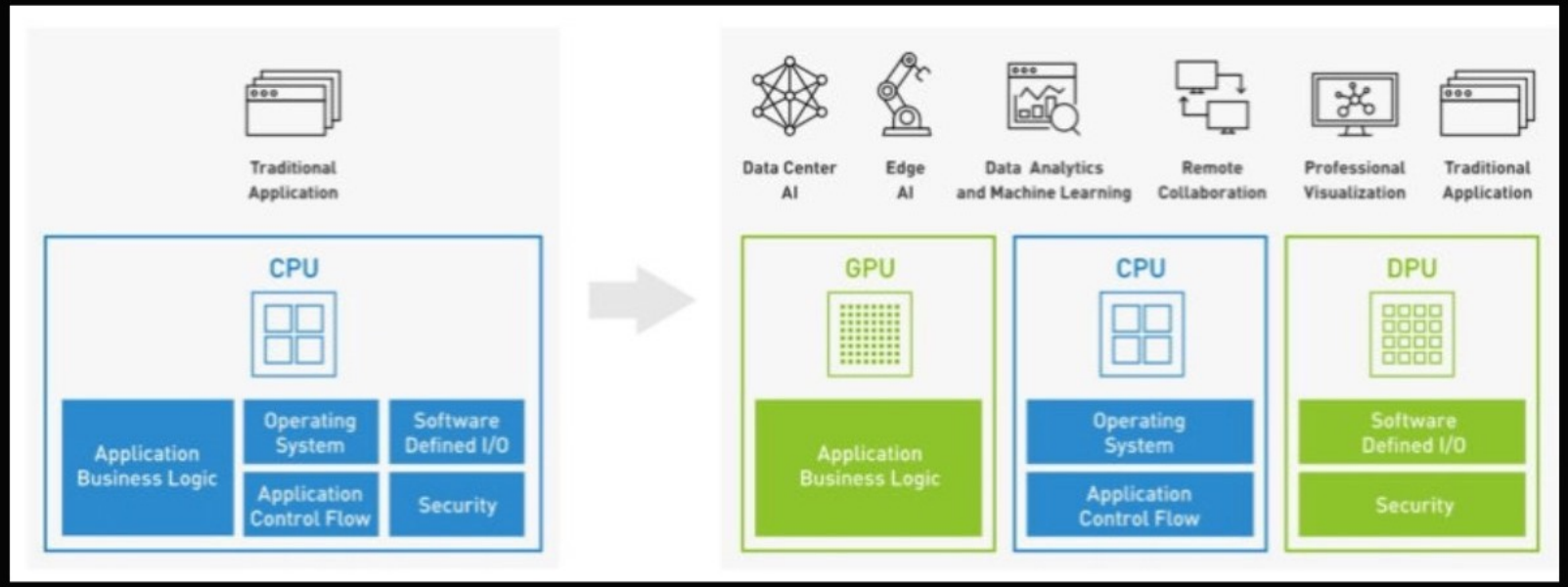
- **Matrix Multiplier Unit (MXU)**: 65,536 multiply-and-add units for matrix operations
- **Unified Buffer (UB)**: 24MB of SRAM that work as registers
- **Activation Unit (AU)**: Hardwired activation functions

TPU Architecture:

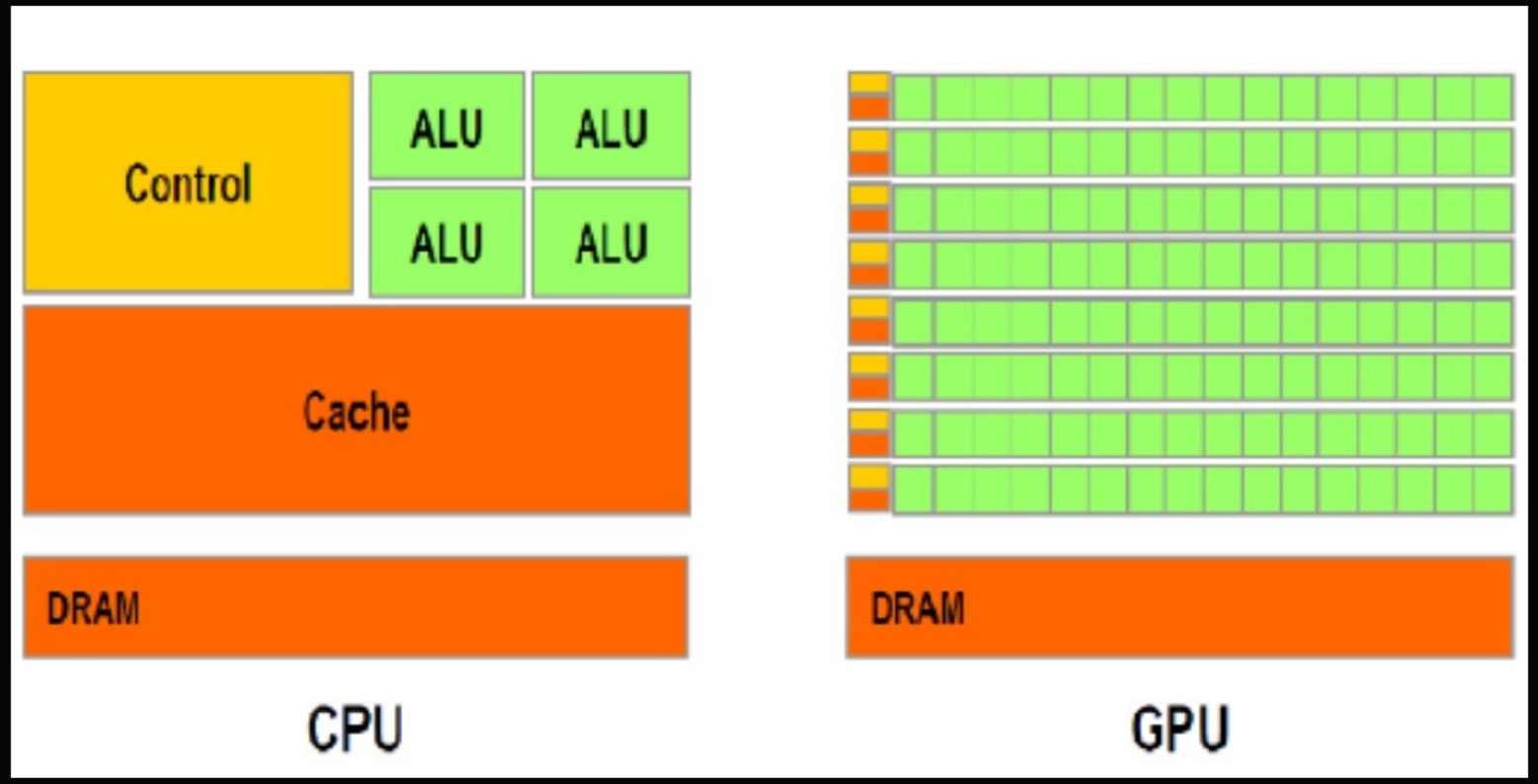


Use case of Accelerators:

Accelerated computing is the use of specialized hardware (Accelerator) to dramatically speed up work, often with **parallel processing** that bundles frequently occurring tasks. It offloads demanding work that can bog down CPUs, processors that typically execute tasks in serial fashion.



CPU vs GPU:



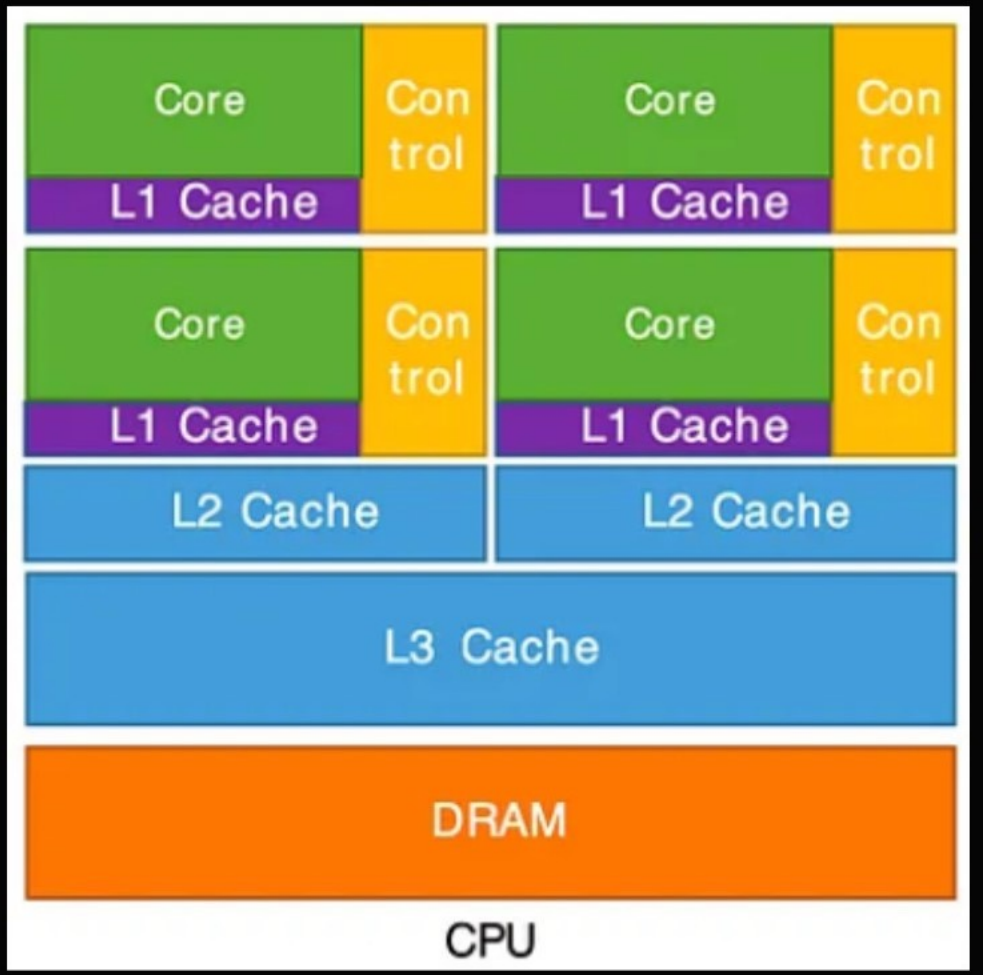
CPU:

CPU Strengths:

- Very large main memory
- Very fast Clock Speeds
- Small number of threads can run very quickly
- Optimized for serial tasks

CPU Weakness:

- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt



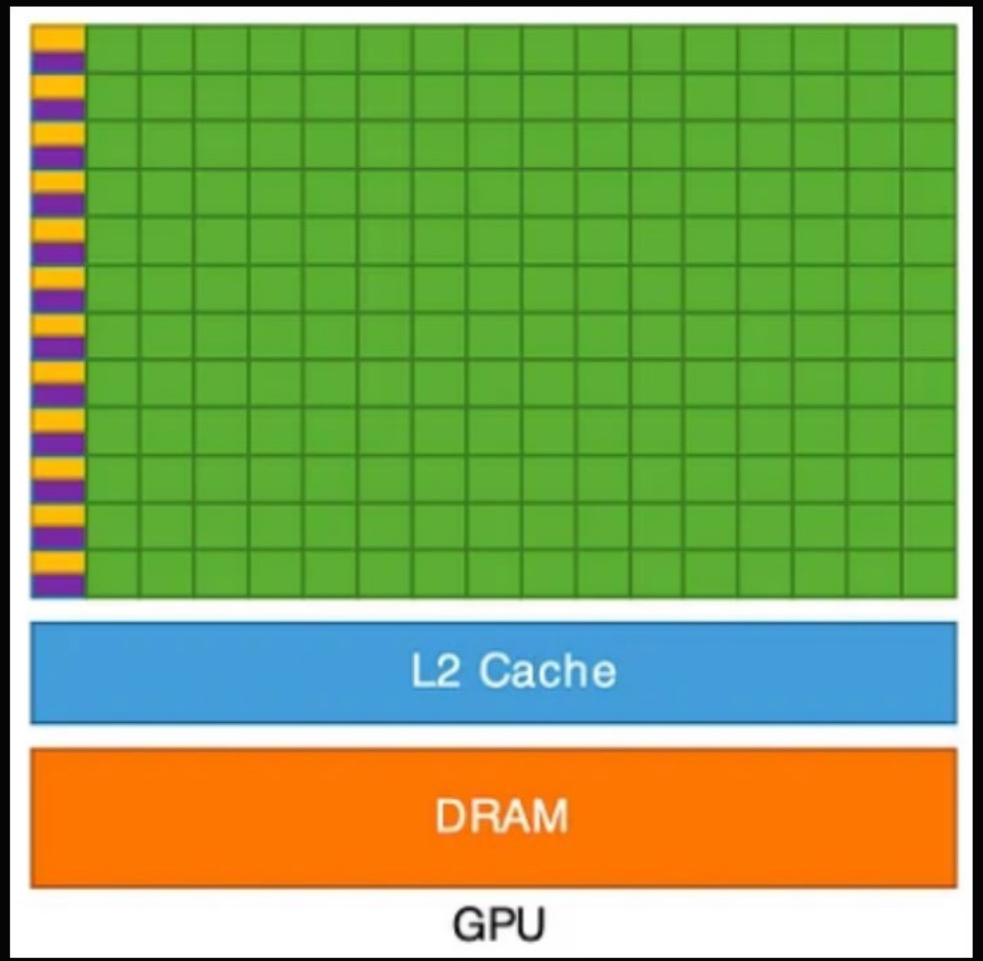
GPU:

GPU strengths:

- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High performance/watt

GPU weakness:

- Relatively low memory capacity
- Low per thread performance

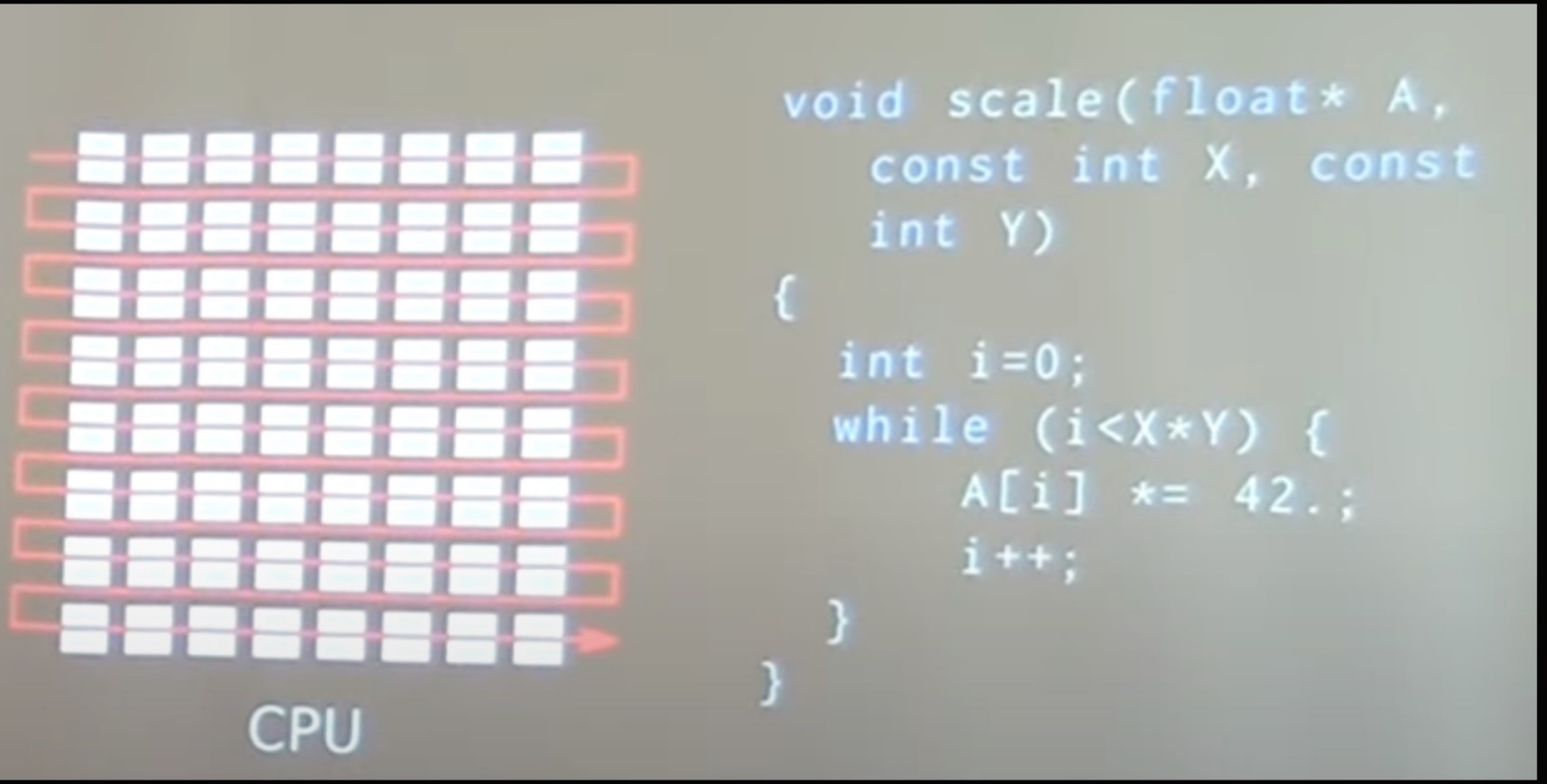


Nvidia GPU Tesla V100:

- New mixed-precision FP16/FP32 Tensor Cores purpose-built for deep learning matrix arithmetic.
- Enhanced L1 data cache for higher performance and lower latency.
- Streamlined instruction set for simpler decoding and reduced instruction latencies.
- Higher clocks and higher power efficiency.

Nvidia GPU Tesla V100:

Scaling Vector on CPU:



Scaling Vector on GPU:

```
__global__ void
    scale(float* A)
{
   int x = threadIdx.x;
   int y = blockIdx.x;
   int X = blockDim.x
   A[y*X+x] *= 42.0;
}

scale<<<8, 8>>>(A);
```

Programming on Accelerators:

GPU

- CUDA
- OpenACC
- SYCL

FPGA

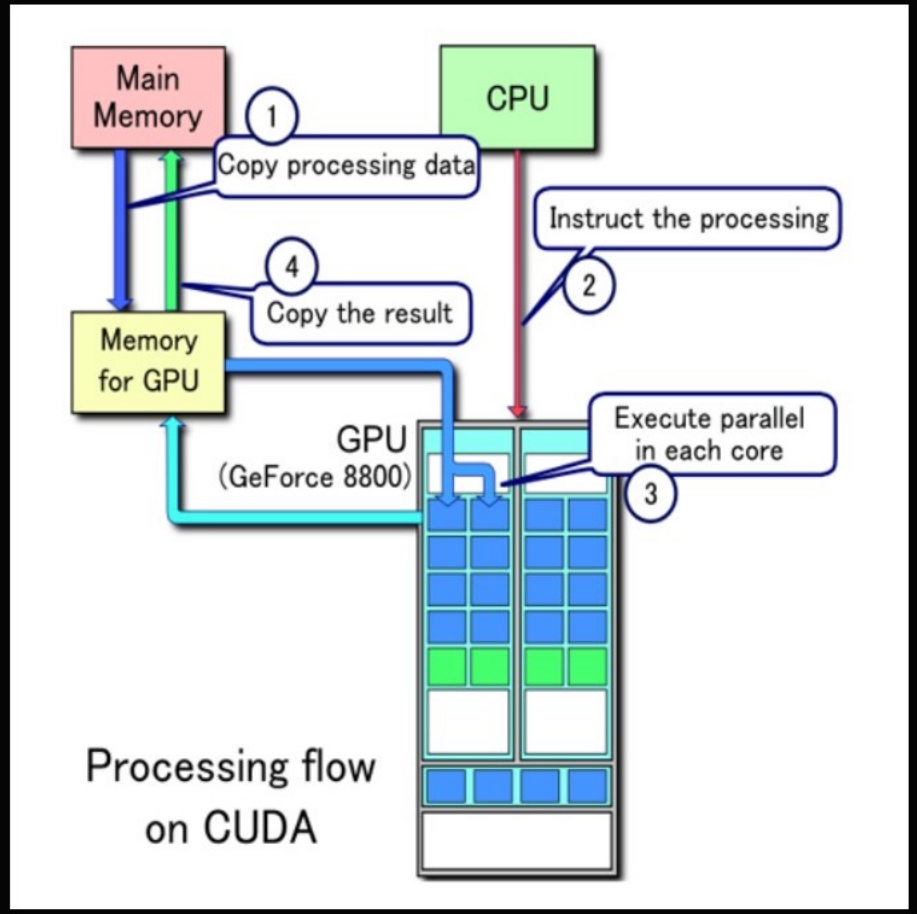
- SYCL

AI/ML

- SYCL

Flow of Accelerated Programming:

1. Copy data from main memmory to GPU memory.
2. CPU instructs the processing to GPU.
3. GPU executes parallel in each core.
4. Copy the results from the GPU memory to the main memory.



Programming Languages:

CUDA:

```
__global__ void kernel( void )
 {
                          ..........
 }

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

- CUDA C keyword __global__ indicates that a function
  - Runs on the device.
  - Called from host.
- nvcc splits source file into host and device components.
- NVIDIA's compiler handles device function like kernel().
- Host compiler handles host functions like main().

CUDA

```
int main( int argc, char* argv[] )
{
    int n = 100000;

    double *h_a;
    double *h_b;
    double *h_c;

    double *d_a;
    double *d_b;
    double *d_c;

    size_t bytes = n*sizeof(double);

    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);

    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);

    int i;
    for( i = 0; i < n; i++ ) {
        h_a[i] = sin(i)*sin(i);
        h_b[i] = cos(i)*cos(i);
    }

    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    int blockSize, gridSize;

    blockSize = 1024;

    gridSize = (int)ceil((float)n/blockSize);

    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

    double sum = 0;
```

Programming Languages:

OpenACC:

OpenACC, which stands for Open Accelerators, is a programming standard for parallel computing. It allows programmers to simplify the process of offloading computations to GPUs while still using familiar programming languages like C, C++, and Fortran.

```
#pragma acc parallel
for(i = 0 ; i < n ; i++) {
        c[i] = a[i] + b[i] ;
}
```

OpenACC

```
#include <iostream>
#include <openacc.h>

using std::cout ;
using std::endl ;

int main( int argc, char* argv[] )
{
        int i ;
        int *a, *b, *c ;
        int n = 10 ;

        a = new int[n] ;
        b = new int[n] ;
        c = new int[n] ;

        for(i = 0; i < n ; i++) {
                a[i] = 90 ;
                b[i] = 10 ;
        }

        #pragma acc parallel
        for(i = 0 ; i < n ; i++) {
                c[i] = a[i] + b[i] ;
        }

        for(i = 0 ; i < n ; i++) {
                cout << c[i] << " ";
        }
        cout << endl ;

        delete[] a ;
        delete[] b ;
        delete[] c ;

        return 0 ;
}
```

Programming Languages:

OpenMP:

- A new directive to offload to accelerators is introduced in openmp4.
- Clauses for data transfer is similar to that of OpenACC.

```
#pragma omp target map(to:n,a,x[:n]) map(from:y[:n])
{
        #pragma omp parallel for
        for (int i = 0; i < n; ++i){
                y[i] = a*x[i] + y[i];
        }
}
```

OpenMP

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main() {
  int n = 10;
  int *h_a, *h_b, *h_c;

  h_a = (int *)malloc(sizeof(int)*n) ;
  h_b = (int *)malloc(sizeof(int)*n) ;
  h_c = (int *)malloc(sizeof(int)*n) ;

  for (int i = 0; i < n; i++) {
    h_a[i] = 1;
    h_b[i] = 2;
  }

  #pragma omp parallel for
  for (int i = 0; i < n; i++) {
    h_c[i] = h_a[i] + h_b[i];
  }

  for (int i = 0; i < n; i++) {
    printf("%d ", h_c[i]) ;
  }
  printf("\n") ;

  free(h_a);
  free(h_b);
  free(h_c);

  return 0;
}
```

Programing Languages:

SYCL:

- SYCL is a KHRONOS standard, it provides a high level abstraction layer over C++.
- It extends C++ in two key ways:
  - Device discovery
  - Device control
- It allows you to write both host and device code in same source file.
- It allows you to write portable code across different devices and architechtures.

```
Q.submit([&] (handler &cgh) {
        auto acc_A = buff_a.get_access<access::mode::read>(cgh) ;
        auto acc_B = buff_b.get_access<access::mode::read>(cgh) ;
        auto acc_C = buff_c.get_access<access::mode::read_write>(cgh) ;

        cgh.parallel_for(range<1>(N), [=](id<1> indx){
                acc_C[indx] = acc_A[indx] + acc_B[indx] ;
        });
});
```

SYCL

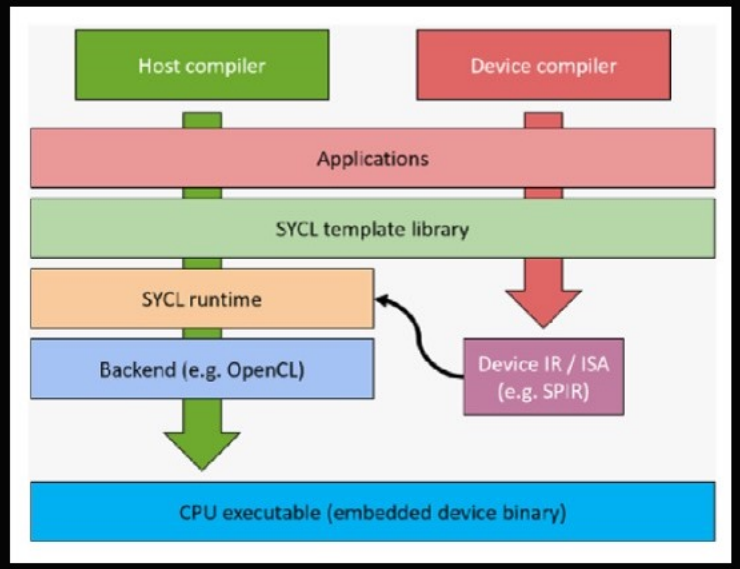
```
int main()
{
        int i ;
        int *dataA, *dataB, *dataC ;
        queue Q ;

        dataA = new int[N] ;
        dataB = new int[N] ;
        dataC = new int[N] ;

        for(i = 0 ; i < N ; i++) {
                dataA[i] = 90 ;
                dataB[i] = 10 ;
        }

        buffer<int, 1> buff_a(dataA, range<1>(N)) ;
        buffer<int, 1> buff_b(dataB, range<1>(N)) ;
        buffer<int, 1> buff_c(dataC, range<1>(N)) ;

        Q.submit([&] (handler &cgh) {
                auto acc_A = buff_a.get_access<access::mode::read>(cgh) ;
                auto acc_B = buff_b.get_access<access::mode::read>(cgh) ;
                auto acc_C = buff_c.get_access<access::mode::read_write>(cgh) ;

                cgh.parallel_for(range<1>(N), [=](id<1> indx){
                        acc_C[indx] = acc_A[indx] + acc_B[indx] ;
                });
        });

        auto C = buff_c.get_access<access::mode::read>() ;

        for(i = 0 ; i < N ; i++){
                cout << C[i] << " " ;
        }
        cout << std::endl ;

        delete[] dataA ;
```

SYCL: Revolutionizing Heterogeneous Programming

By Laxmikant Botkewar

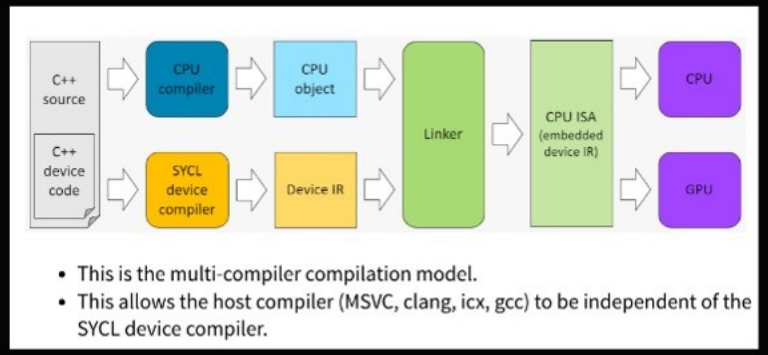
What is SYCL?

1. SYCL is a Single Source, high level standarad C++ programming model.
2. SYCL can target a range of heterogeneous platforms.
3. SYCL is a open source Kronos implementation.
4. SYCL is based on modern C++.

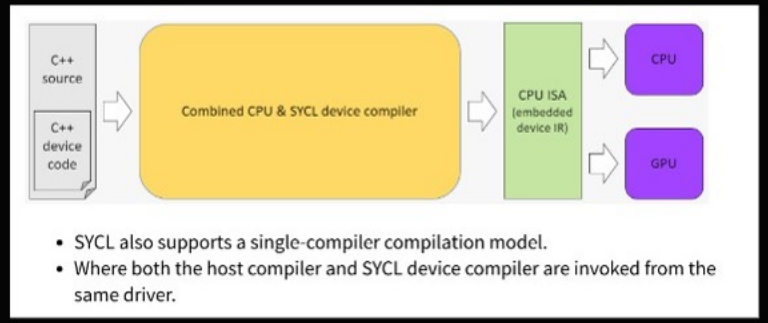


How SYCL Works

As SYCL is a single source code for both host & device it requires two compilation passes one for host code and one for device code.



- This is the multi-compiler compilation model.
- This allows the host compiler (MSVC, clang, icx, gcc) to be independent of the SYCL device compiler.

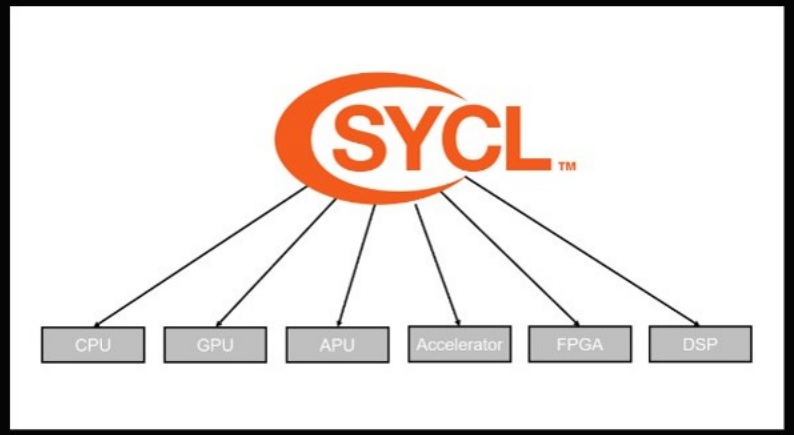


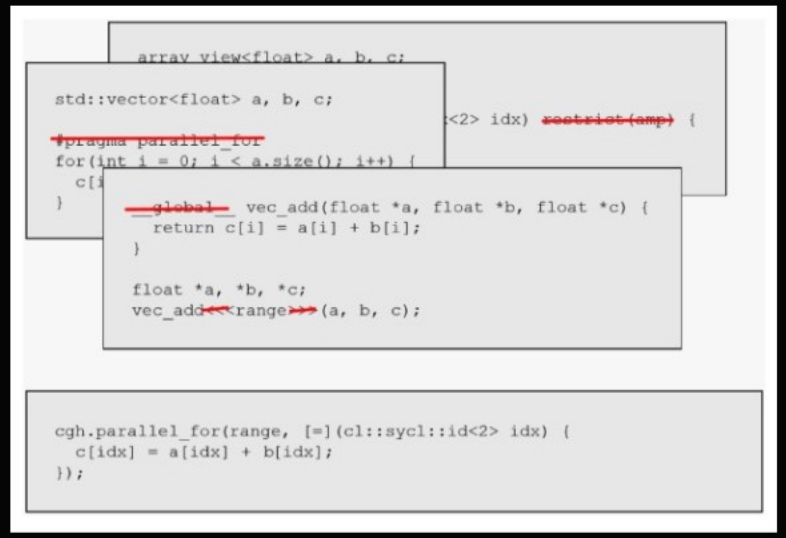
- SYCL also supports a single-compiler compilation model.
- Where both the host compiler and SYCL device compiler are invoked from the same driver.

SYCL has been designed to be implemented on top of various backends. The implementation supports backends such as OpenCL, OpenMP, CUDA, HIP, SPIR-V and others.

SYCL provides high level abstraction layer for

- Platform/device selection.
- Buffer creation and Data movement.
- Kernel function compilation.
- Dependency management and Scheduling.





Unlike the other implementations SYCL has

- No language extentions.
- No pragmas.
- No attributes.

SYCL in Action

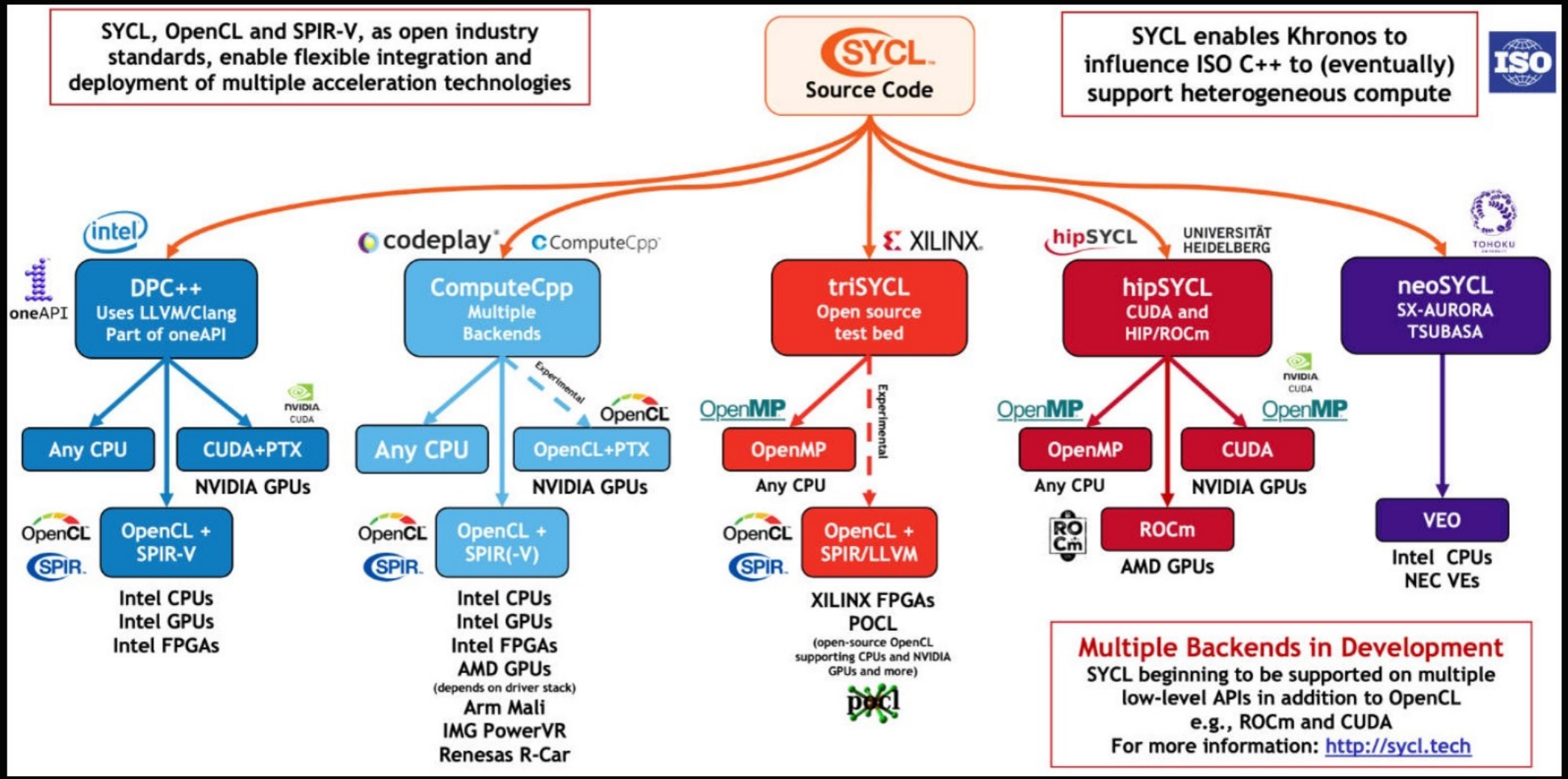
```
#include <CL/sycl.hpp>

int main(int argc, char *argv[]) {
  std::vector<float> dA{2.3}, dB{3.2}, dO{7.9};

  try {
    auto asyncHandler = [&](sycl::exception_list eL) {
      for (auto &e : eL)
        std::rethrow_exception(e);
    };
    sycl::queue gpuQueue{sycl::default_selector{}, asyncHandler};

    sycl::buffer bufA{dA.data(), sycl::range{dA.size()}};
    sycl::buffer bufB{dB.data(), sycl::range{dB.size()}};
    sycl::buffer bufO{dO.data(), sycl::range{dO.size()}};

    gpuQueue.submit([&](sycl::handler &cgh) {
      sycl::accessor inA(bufA, cgh, sycl::read_only);
      sycl::accessor inB(bufB, cgh, sycl::read_only);
      sycl::accessor out(bufO, cgh, sycl::write_only);

      cgh.parallel_for(sycl::range{dA.size()},
                       [=](sycl::id<1> i) { out[i] = inA[i] + inB[i]; });
    });

    gpuQueue.wait_and_throw();
```

Managing the data

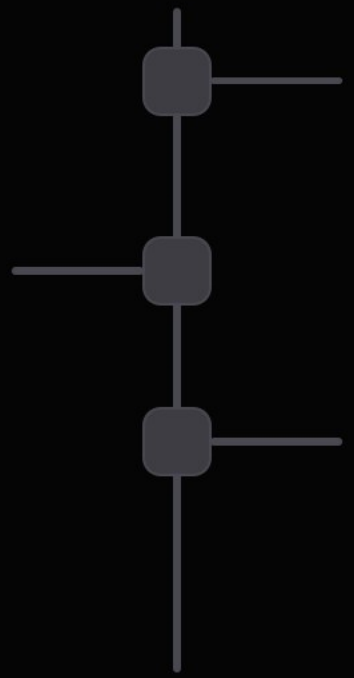
Work unit

Device code

SYCL Implementations:



Managing Data:



Learn about buffer/accessor model of managing data.

Learn how to buffer and accessor.

Learn how to access data in kernel function.

Buffer & Accessor:

1. A SYCL buffer manages data across the host and any number of devices.

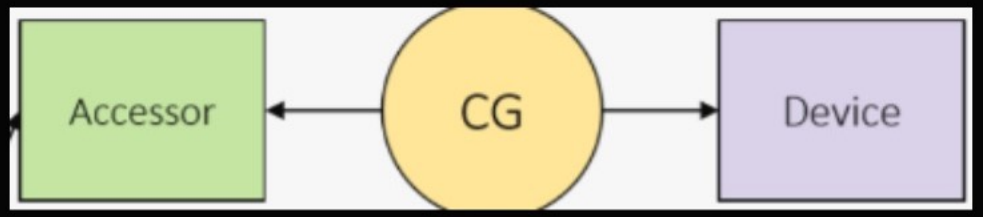
2. A SYCL accessor requests access to data on the host or on a device for a specific SYCL kernel function.

3. Accessors are also used to access the data within a SYCL kernel function.

Buffer:

- A SYCL buffer can be constructed with a pointer to host memory.
- When a buffer object is created it will not allocate or copy to device memory at first.
- This will only happen when the SYCL runtime knows the data needs to be accessed and where it needs to be accessed.

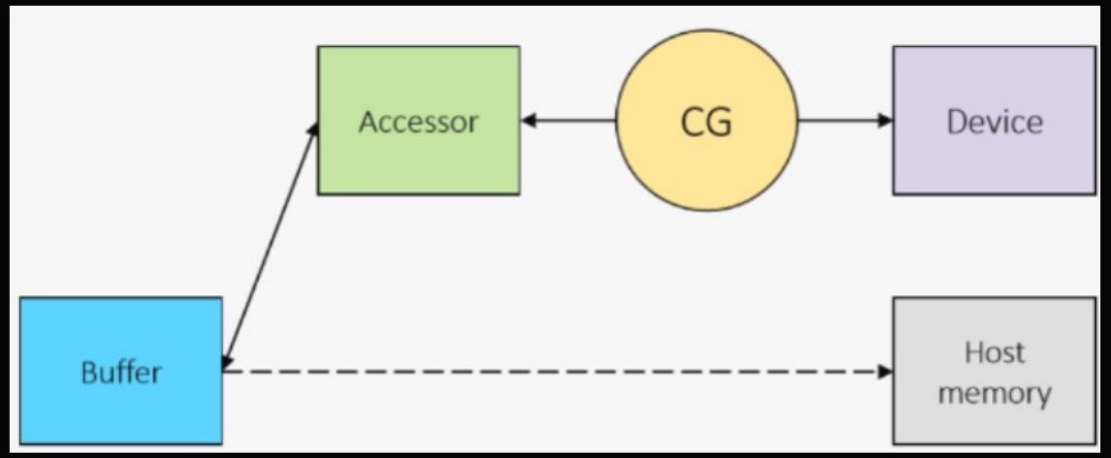
Accessor:



- Constructing an accessor specifies a request to access the data managed by the buffer.
- There are a range of different types of accessors which provide different ways to access the data.
- When an accessor is constructed it is associated with a command group via the **handler object**.
- This connects the buffer that is being accessed and the device that the command group is being submitted to.

Buffer & Accessor:

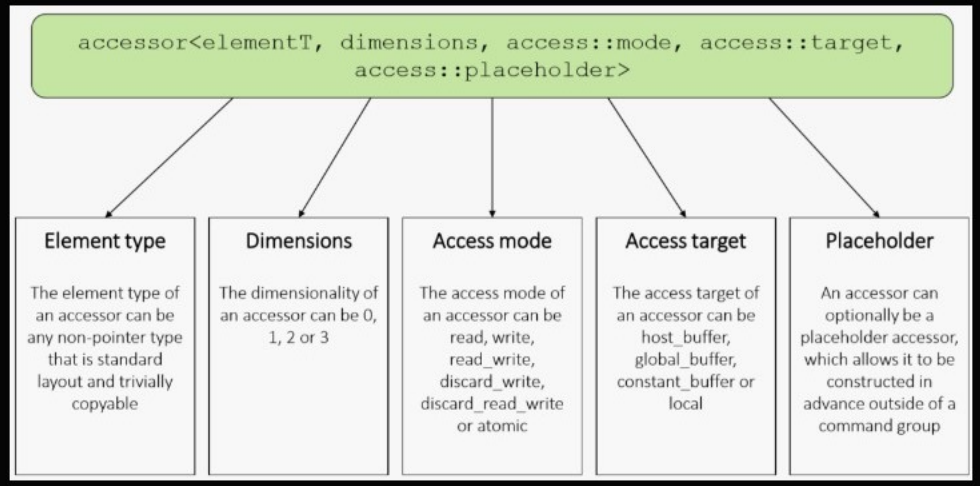
Buffer Class:



- A buffer manages data across the host application and kernel functions executing on device.
- It has a typename which specifies the type of the elements of the data it manages.
- It has a dimensionality which specifies the dimensionality that the elements of data are represented in.

Buffer & Accessor:

Accessor Class:

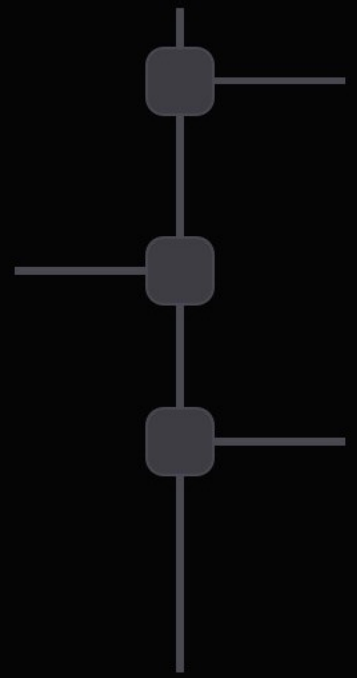


- The accessor class supports CTAD so its not necessary to specify all of the template arguments.
- The most common way to construct an accessor is from a buffer and a handler associated to the command group function you are within.
    - The element type and dimensionality are inferred from buffer.

Buffer & Accessor:

```
buffer<int, 1> buff_a(dataA, range<1>(N)) ;
buffer<int, 1> buff_b(dataB, range<1>(N)) ;
buffer<int, 1> buff_c(dataC, range<1>(N)) ;

Q.submit([&] (handler &cgh) {
    auto acc_A = buff_a.get_access<access::mode::read>(cgh) ;
    auto acc_B = buff_b.get_access<access::mode::read>(cgh) ;
    auto acc_C = buff_c.get_access<access::mode::read_write>(cgh) ;

    cgh.parallel_for(range<1>(N), [=](id<1> indx){
        acc_C[indx] = acc_A[indx] + acc_B[indx] ;
    });
});
```

Enqueuing a Kernel:



Learn about Queues and how to submit work to them

Learn how to compose command groups

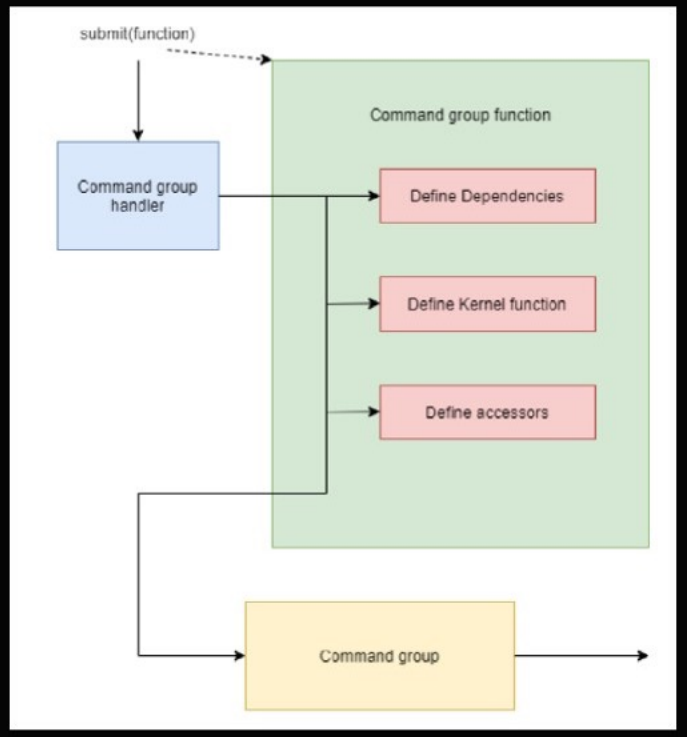
Learn how to define kernel functions

Queue:

- In SYCL all work is submitted via commands to a queue.
- The queue has an associated device, any commands enqueued to the queue will target that device.
- There are several different ways to construct a queue. The most straight forward way is to default construct one.

```
queue Q(gpu_selector{}) ;

auto R = sycl::range<1> { 10 } ;
buffer<int> A{ R } ;

Q.submit([&] (handler& h) {
    accessor A_acc(A, h) ;
}) ;
```

Command Groups:



- In buffer/accessor model commands must be enqueued via command groups.
- Command group consists of a series of commands to be executed by a device.
- These commands include
  - Invoking kernel functions on a device.
  - Copying data to and from the device.
  - waiting on other commands to complete.

Composing Command Group:

- The submit member function takes a C++ function object, which takes a reference to a handler.
- The body of the function object represents the command group function.
- The function object can be a LAMDA expression.
- The command group function is processed exactly once when submit is called.
- The command group is then submitted asynchronously to the scheduler.

```
Q.submit([&] (handler& h) {

    /* Command Group Function */

}) ;
```

Composing Command Group:

```
Q.submit([&] (handler& h) {

    /* Command Group Function */

}).wait() ;
```

- The queue will not wait for commands to complete on destruction.
- Submit returns an event to allow you to synchronize with the completion of commands.
- Here we call wait on the event to immediately wait for it to complete.

SYCL Kernel Function:

- SYCL kernel functions are defined using a kernel invoke API's provided by the handler.
- These add a SYCL kernel function command to the command group.
- Only one SYCL kernel function command is allowed in a command group.

```
Q.submit([&] (handler& h) {
    h.single_task([=]() {
        /* kernel code */
    }) ;
}).wait() ;
```

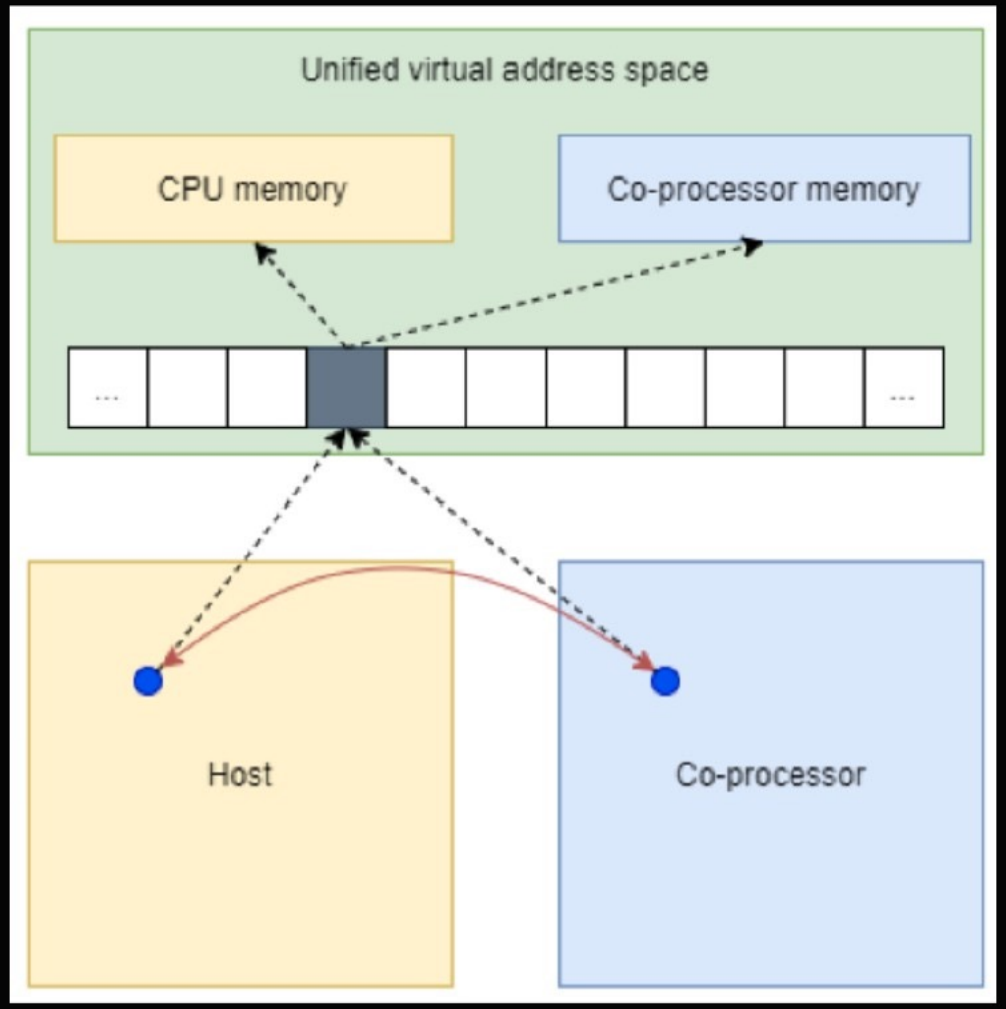
SYCL Kernel Function:

```
Q.submit([&] (handler& h) {
    h.single_task([=]() {

        /* kernel code */

    }) ;
}).wait() ;
```

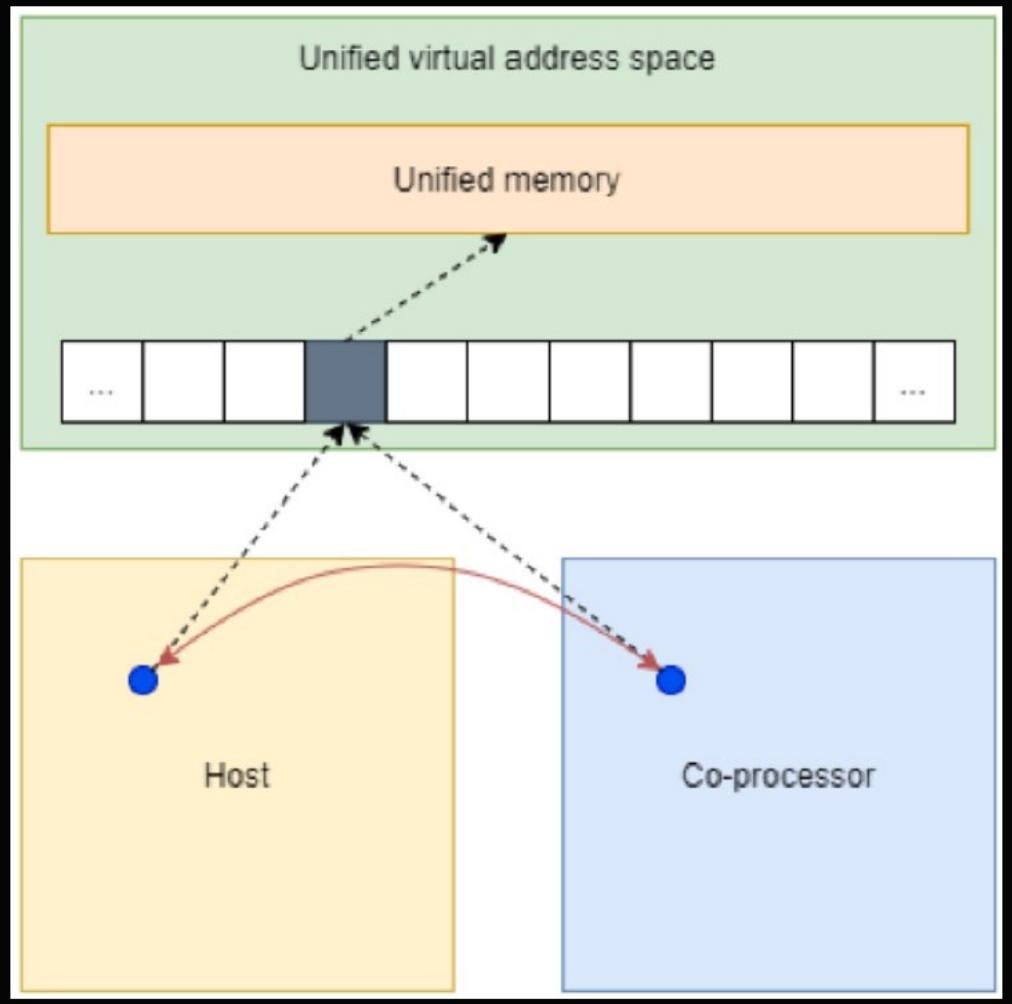
- The kernel function API takes function object representing a kernel function.
- These can be lambda expression or a class with function call operator.
- This is the entry point for the code that is compiled to be executed on the device.

Unified Shared Memory:

- USM provides an alternative pointer-based data management model to the accessor-buffer model.
  - Unified Virtual address space.
  - pointer based structure.
  - shared memory allocations.



Unified Shared Memory:



- In shared memory allocations there is no need to write explicit routines to move data from host and device.
- Some USM platform will support variants of USM where memory allocations share the same memory region between host and device.

Unified Shared Memory:

USM has three different kinds of allocations:

- A **host** allocation is allocated in host memory
- A **device** allocation is allocation in device memory.
- A **shared** allocation is allocated in shared memory and can be accessed by both host and device.

host	in host memory accessible by a device
device	in device memory not accessible by the host
shared	in shared memory accessible by host and device