Fundamentals of CUDA 1

Prof. Seokin Hong

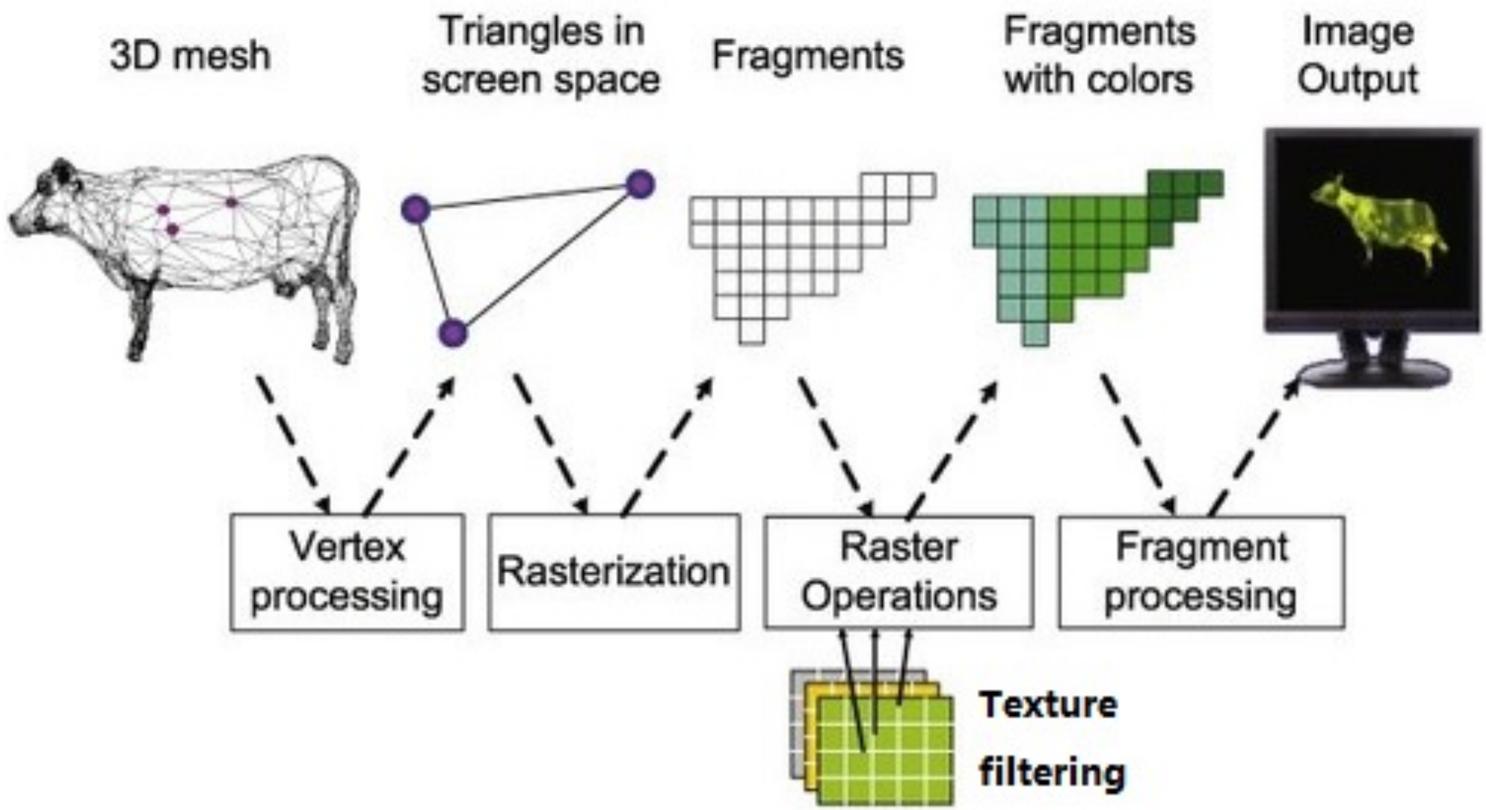
Slide Credit: Slides are modified from Prof Baek's slides

Agenda

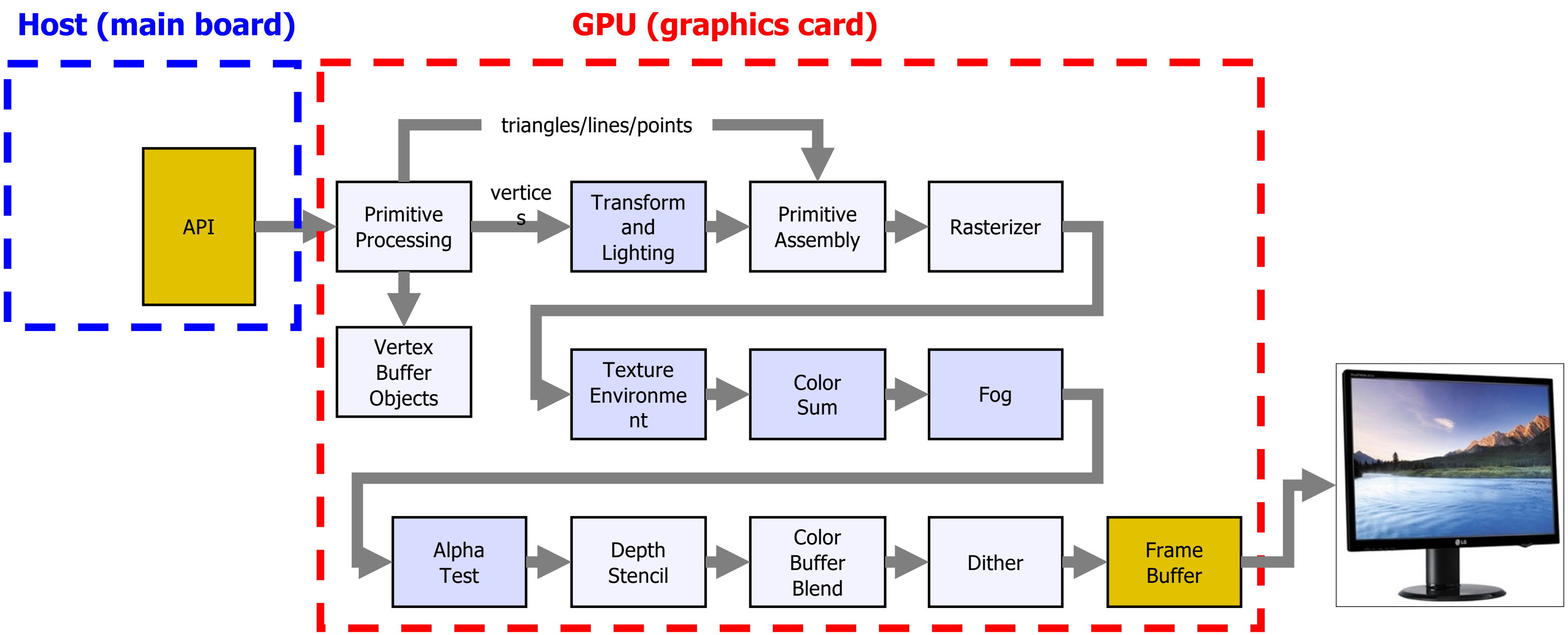
- History
- What is CUDA?
- Device Global Memory and Data Transfer
- Error Checking
- A Vector Addition Kernel
- Kernel Functions and Threading
- Kernel Launch

History

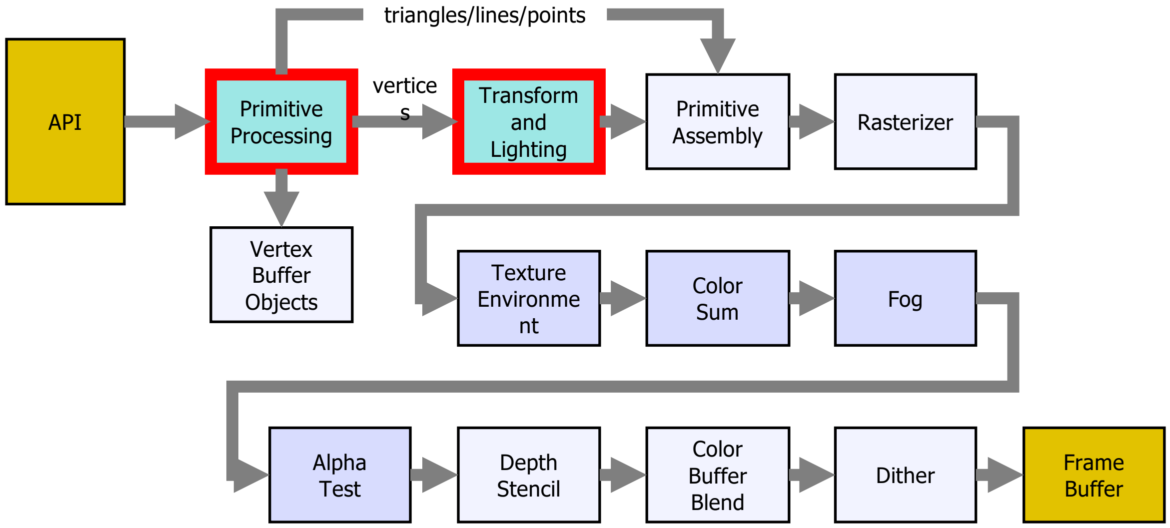
3D Graphics Pipeline

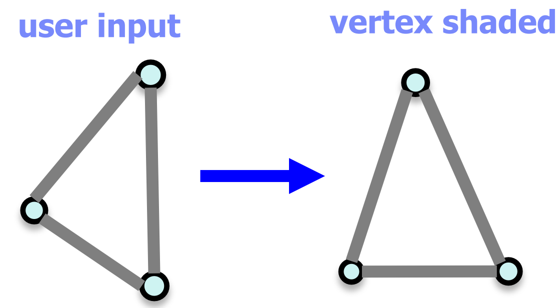


3D Graphics Pipeline

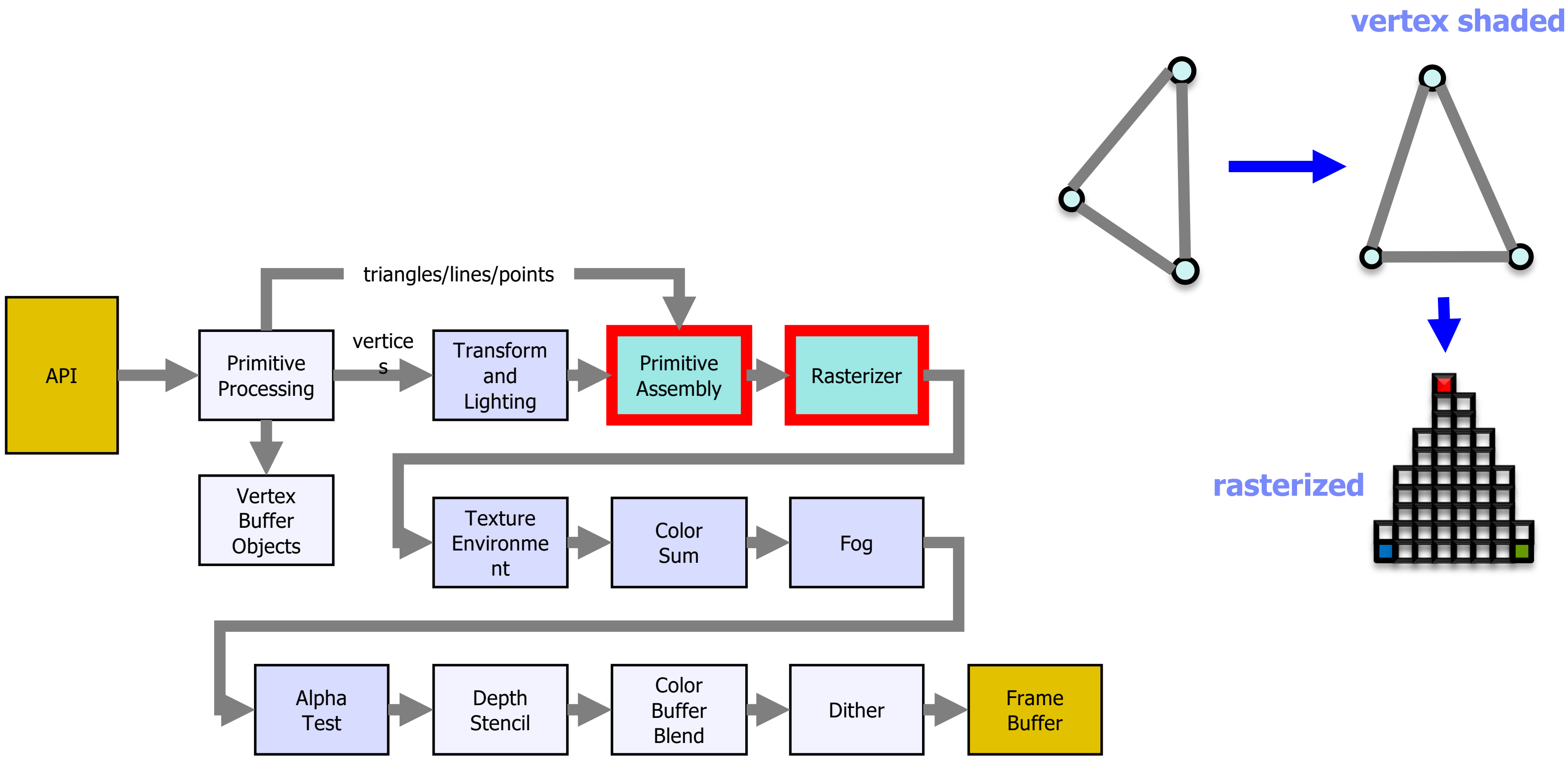


3D Graphics Pipeline

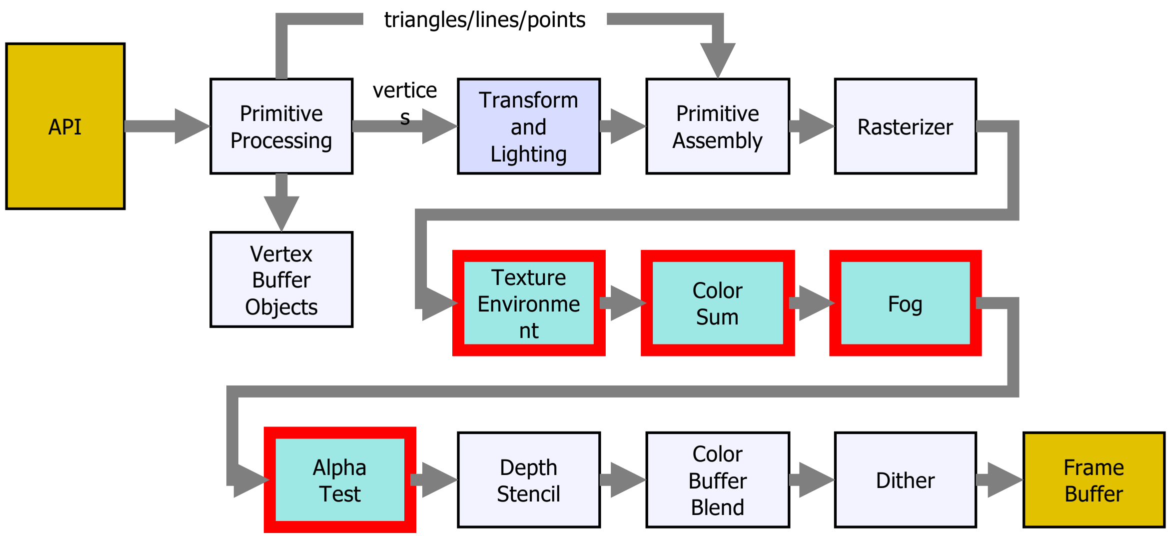


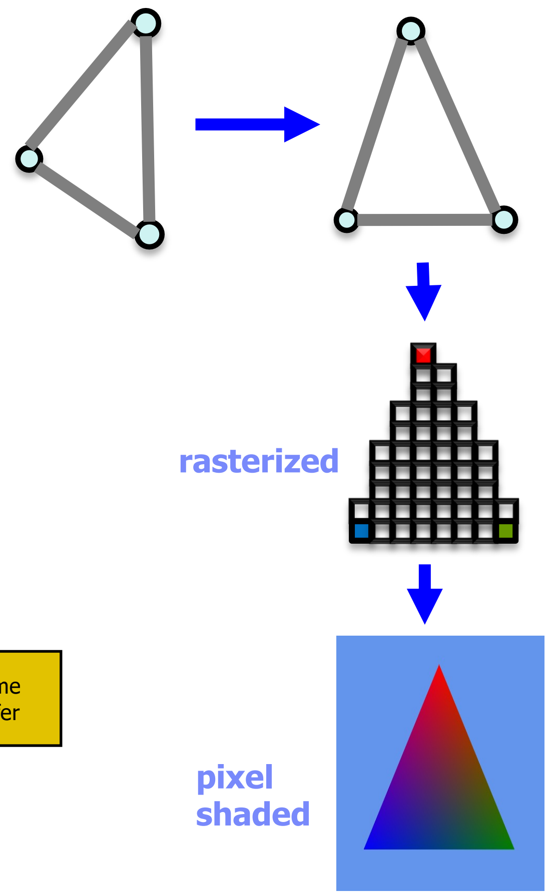


3D Graphics Pipeline



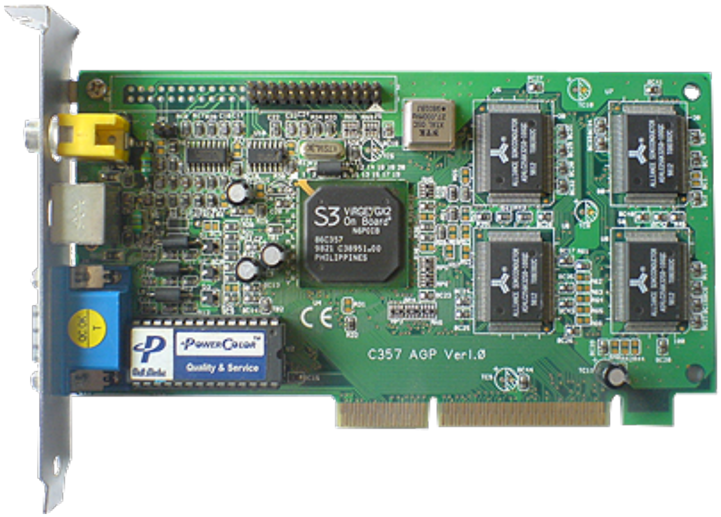
3D Graphics Pipeline

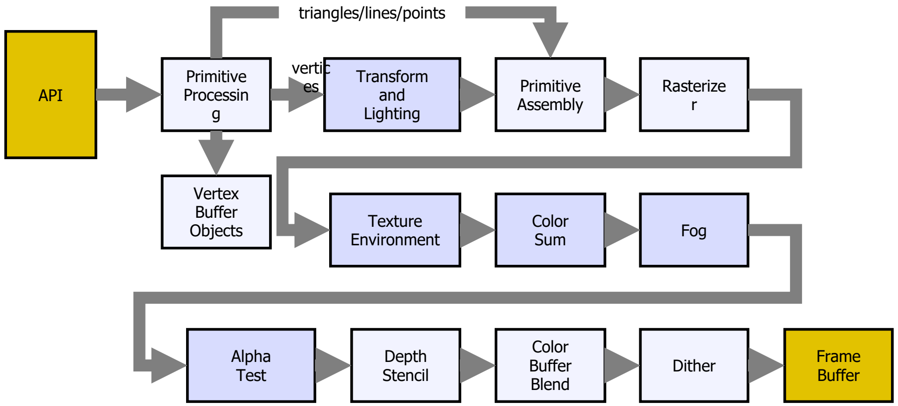




The Graphics Pipeline – 1st Gen.

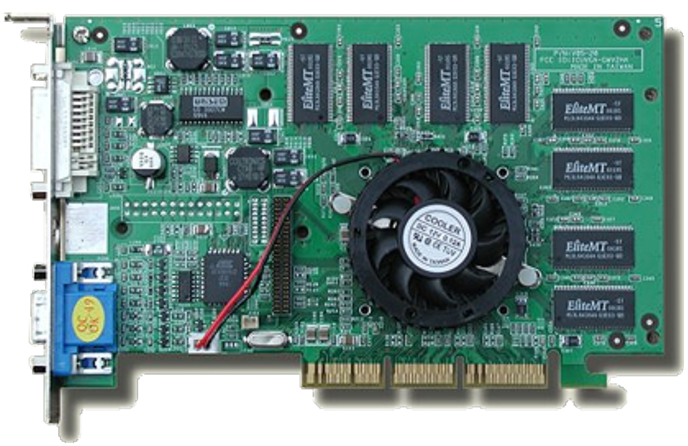
- One chip/board per stage
- Fixed data flow through pipeline

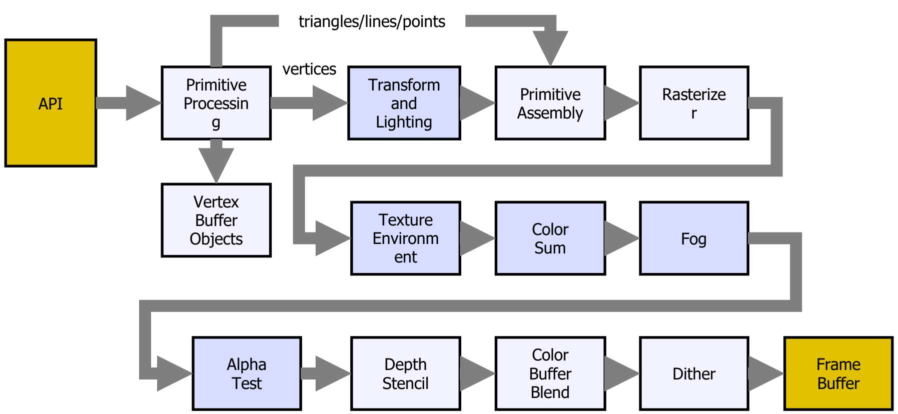




The Graphics Pipeline – 2nd Gen.

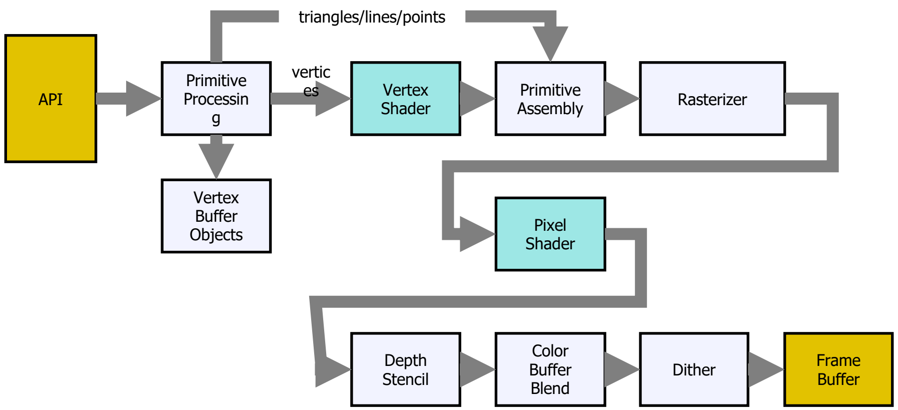
- Everything fixed function,
  with a certain number of modes
- Number of modes for each stage grew over time
- Hard to optimize HW
- Developers always wanted more flexibility





The Graphics Pipeline – 3rd Gen.

- Vertex & pixel processing became programmable
- GPU architecture increasingly centers around `shader' execution



Before CUDA

- Use the GPU for general-purpose

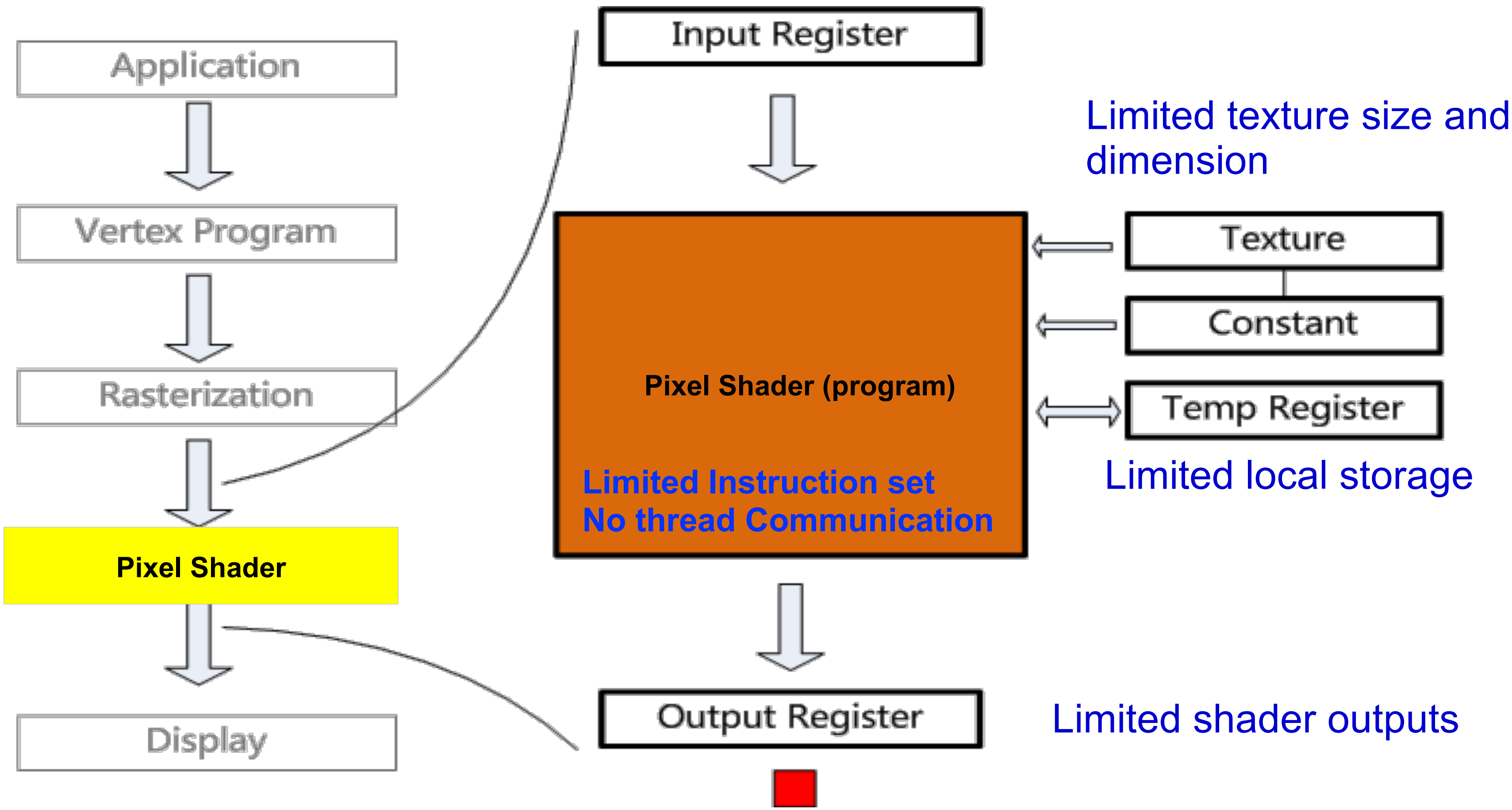
- Computing by casting problem as graphics
  - Turn data into images ("texture maps")
  - Turn algorithms into image synthesis ("rending passes")

- **Drawback:**
  - Tough learning curve
  - Potentially high overhead of graphics API
  - Highly constrained memory layout & access model

Before CUDA

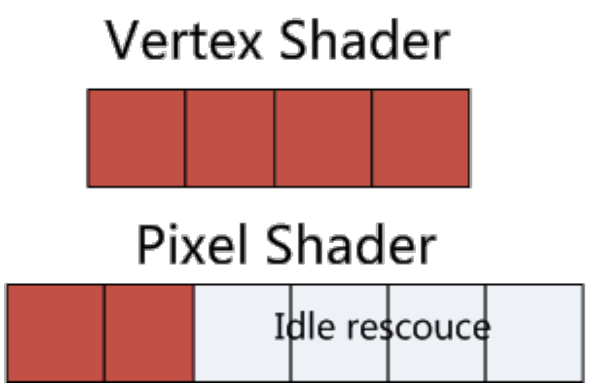
- What's wrong with the old GPGPU programming model 1

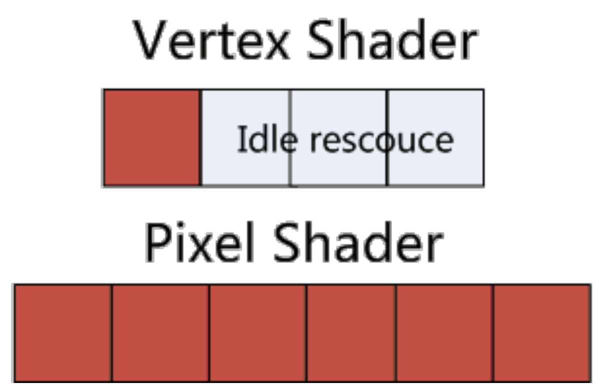
APIs are specific to Graphics



Before CUDA

- What's wrong with the old GPGPU programming model 2

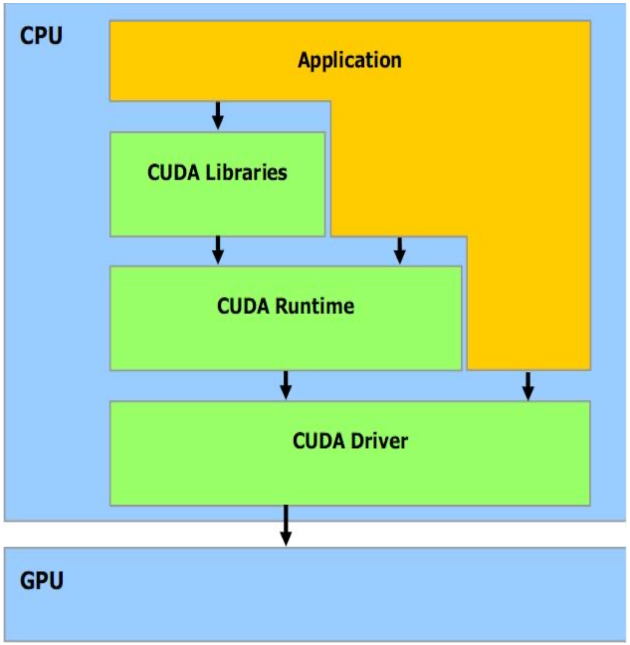




What is CUDA?

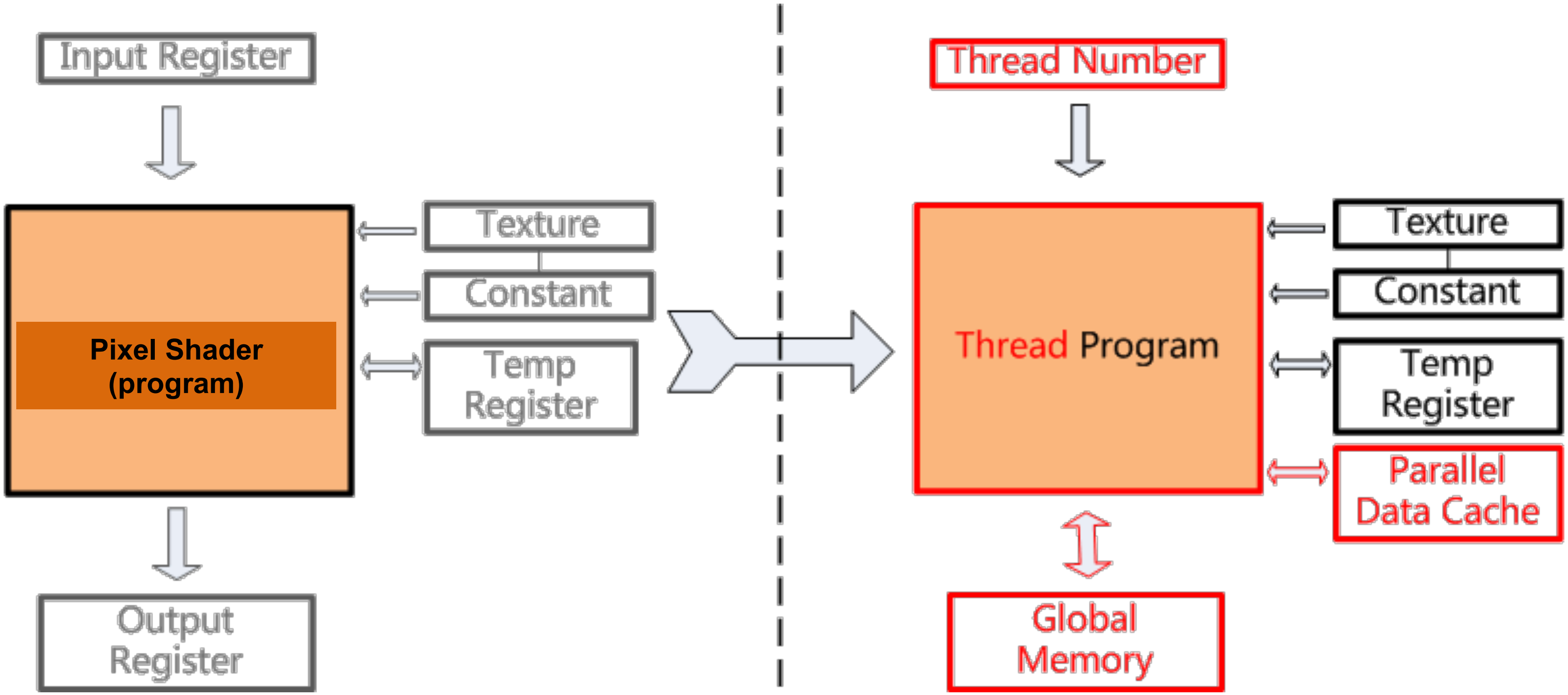
What is CUDA?

- **What is CUDA?: Compute Unified Device Architecture**
  - Parallel computing platform and application programming interface (API) model
  - A powerful parallel programming model for issuing and managing computations on the GPU without mapping them to a graphics API
- **Targeted Software stack**
  - Library, Runtime, Driver



- **Advantages**
  - SW: program the GPU in C
    - Scalable data parallel execution/memory model
    - C with minimal yet powerful extensions
  - HW: fully general data-parallel architecture
- **Features**
  - **Heterogenous** - mixed serial-parallel programming
  - **Scalable** - hierarchical thread execution model
  - **Accessible** - minimal but expressive changes to C

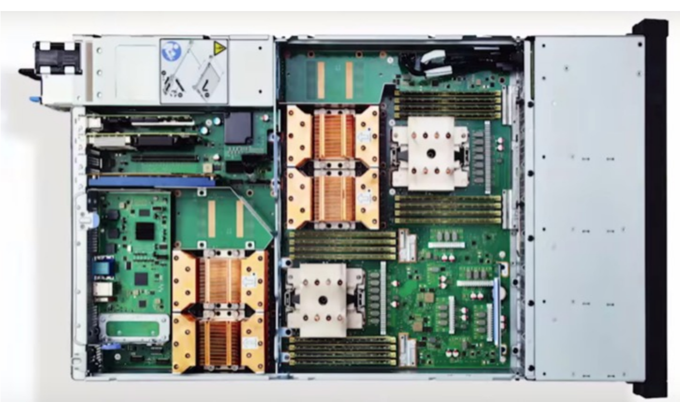
What is CUDA?

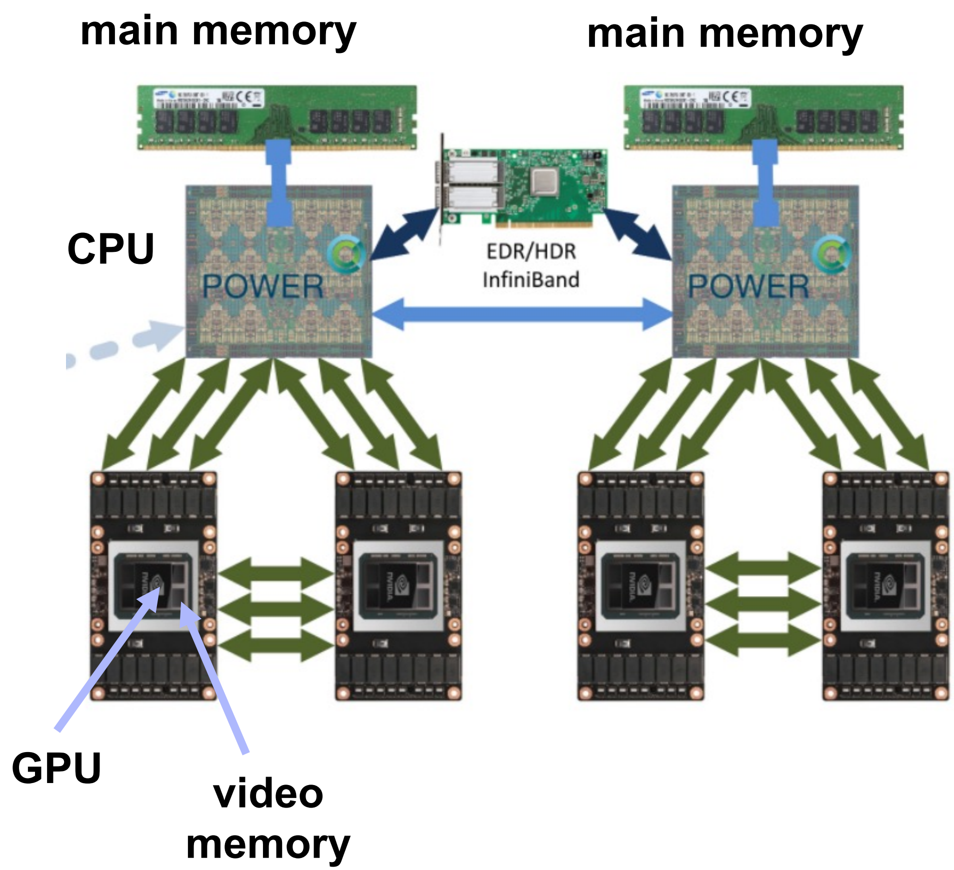


Input Register
Texture
Constant
Pixel Shader
(program)
Temp
Register
Output
Register
Thread Number
Texture
Constant
Thread Program
Temp
Register
Parallel
Data Cache
Global
Memory

Review : Heterogeneous Computing

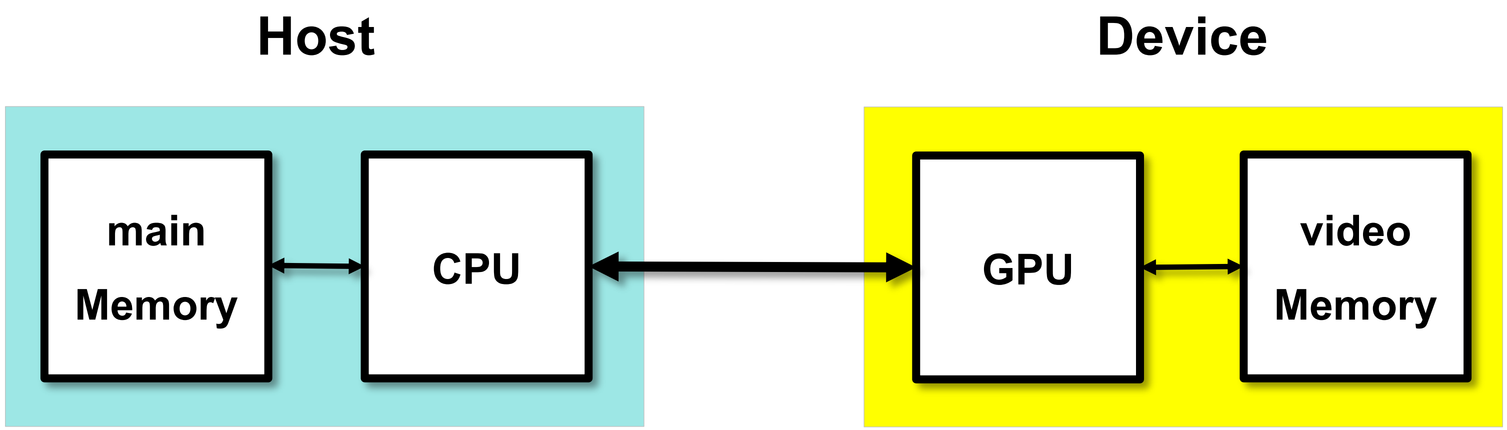
- Use more than one kind of processor or cores
  - CPUs for sequential parts
  - GPUs for parallel parts





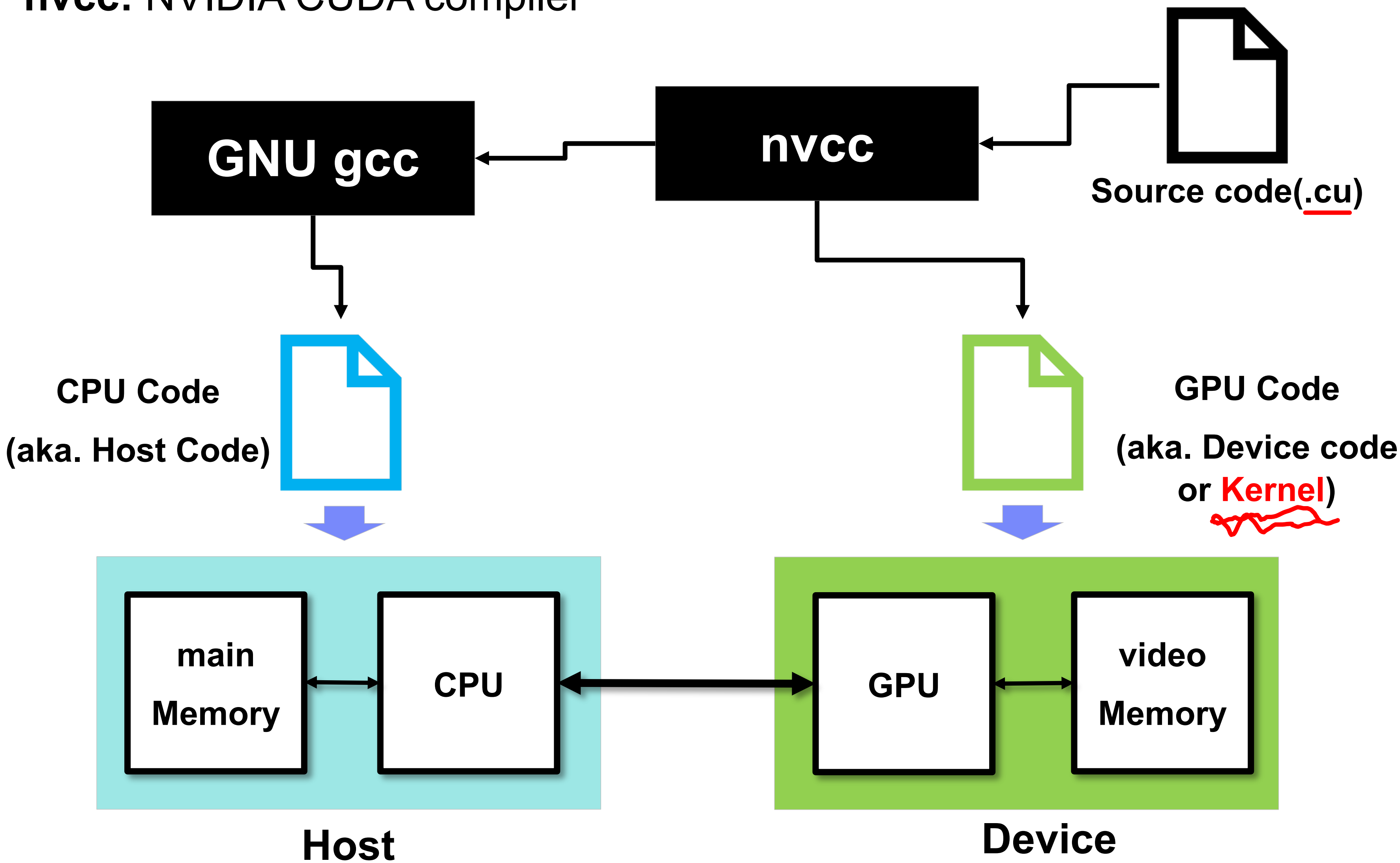
Simple CUDA Model

- **Host** : CPU + main memory (host memory)
- **Device** : GPU + video memory (device memory)



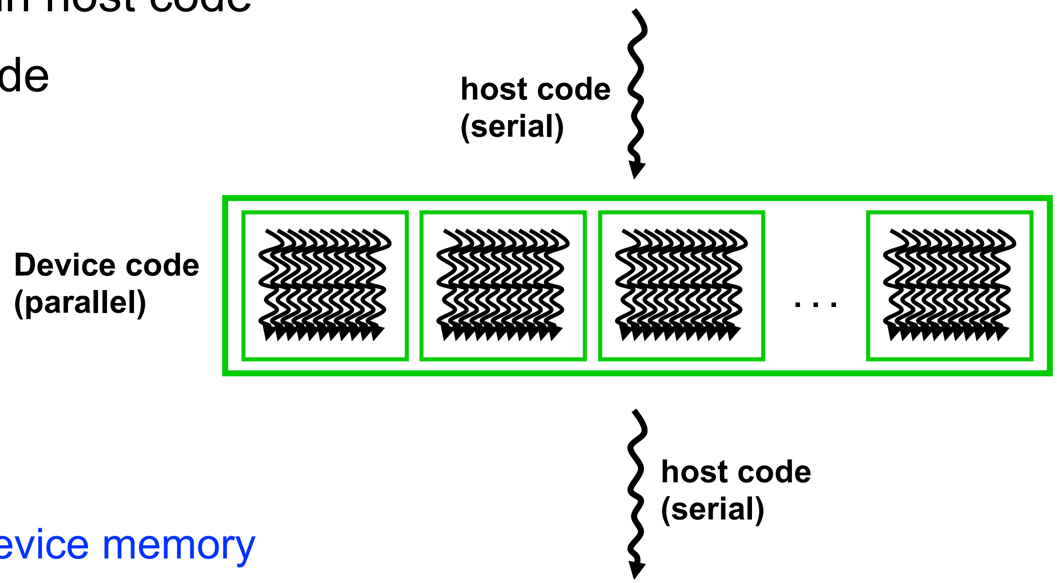
Simple CUDA Model

- **GNU gcc** : linux c compiler
- **nvcc:** NVIDIA CUDA compiler

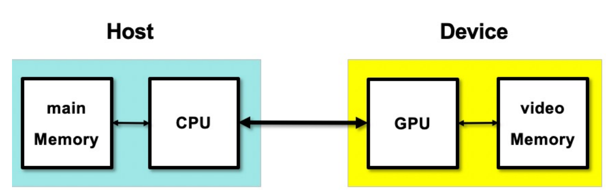


CUDA Program Execution Scenario

- Integrated “host+device” app C program
  - Serial or modestly parallel parts in host code
  - Highly parallel parts in device code



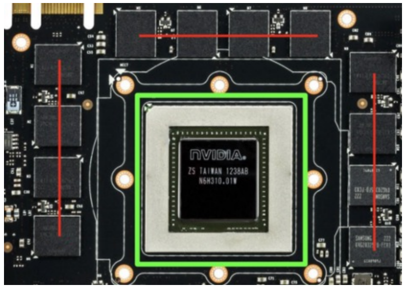
- **Execution Scenario**
  - **Step 1: host code**
    - Serial execution: read data
    - Prepare parallel execution
    - Copy data from host memory to device memory
  - **Step 2: device code (kernel)**
    - Parallel processing
    - Read/write data from device memory to device memory
  - **Step 3: host code**
    - Copy data from device memory to host memory
    - Serial execution: print data



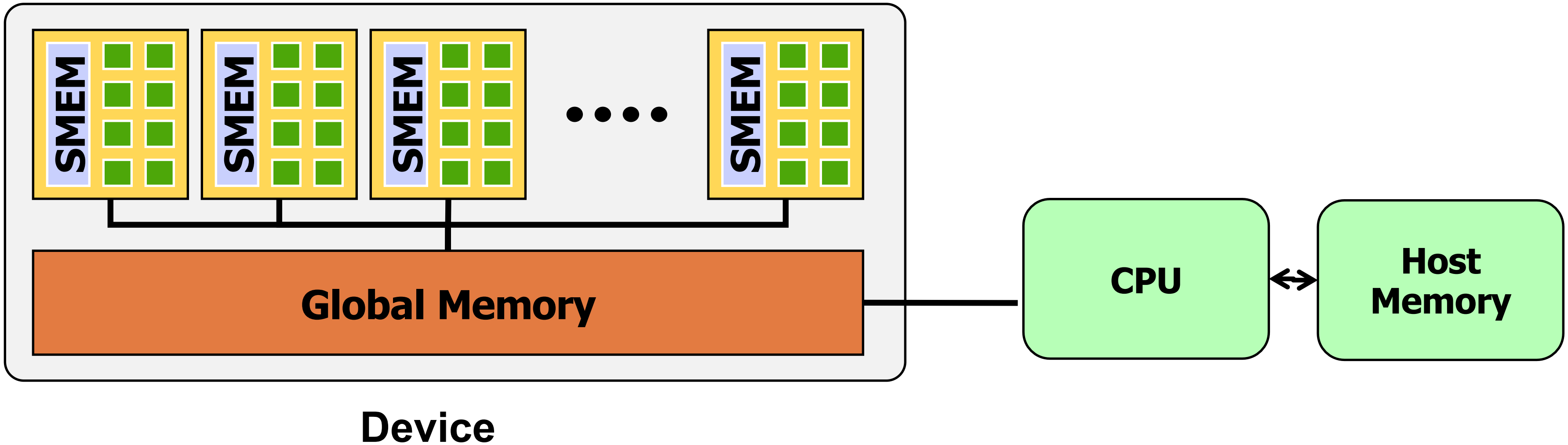
Device Global Memory and Data Transfer

CUDA program uses CUDA memory

- GPU cores share the “global memory” (device memory)
  - DRAM (e.g., GDDR, HBM) is used as global memory
- To execute a kernel on a device,
  - **allocate** global memory on the device
  - **transfer** data from the host memory to allocated device memory
  - **transfer** result data from the device memory back to the host memory
  - **release** global memory



Red lines are global memory



Memory Spaces

- **CPU and GPU have separate memory spaces**
  - Data is moved across data bus
  - Use functions to allocate/set/copy memory on GPU
  - Very similar to corresponding C functions

- Pointers are just addresses
  - Use pointer to access CPU and GPU memory
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Dereferencing CPU pointer on GPU will likely crash
  - Same for vice versa

CPU Memory Allocation / Release

- **Host (CPU) manages host (CPU) memory:**
  - void* malloc (size_t nbytes)
  - void* memset (void* pointer, int value, size_t count)
  - void free (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int* ptr = 0;
ptr = malloc( nbytes );
memset( ptr, 0, nbytes);
free( ptr );
```

GPU Memory Allocation / Release

- **Host (CPU) manages device (GPU) memory:**
  - **cudaMalloc** (void** pointer, size_t nbytes)
  - **cudaMemset** (void* pointer, int value, size_t count)
  - **cudaFree** (void* pointer)

```
int  n = 1024;
int  nbytes = 1024*sizeof(int);
int* dev_a = 0;
cudaMalloc( (void**)&dev_a,  nbytes );
cudaMemset( dev_a, 0, nbytes);
cudaFree(dev_a);
```

CUDA function rules

- Every library function starts with "cuda"

- Most of them returns error code (or cudaSuccess).
  - cudaError_t cudaMalloc(void** devPtr, size_t size);
  - cudaError_t cudaFree(void* devPtr);
  - cudaError_t cudaMemcpy(void* dst, const void* src, size_t size);

- Example:
  - ```
    if (cudaMalloc(&devPtr, SIZE) != cudaSuccess) {
            exit(1);
    }
    ```
    ```

CUDA Malloc

- cudaError_t   cudaMalloc(void** devPtr, size_t nbytes);
  - allocates *nbytes* bytes of linear memory on the device
  - The start address is stored into "*devPtr*"
  - The memory is not cleared.
  - returns **cudaSuccess** or **cudaErrorMemoryAllocation**

- cudaError_t   cudaFree(void* devPtr);
  - frees the memory space pointed by *devPtr*
  - if *devPtr* == 0, no operation
  - returns **cudaSuccess** or **cudaErrorInvalidDevicePointer**

CUDA mem set

- cudaError_t cudaMemset(void* devPtr, int value, size_t nbytes);
  - fills the first ***nbytes*** byte of the memory area pointed by ***devPtr*** with the *value*
  - returns cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer

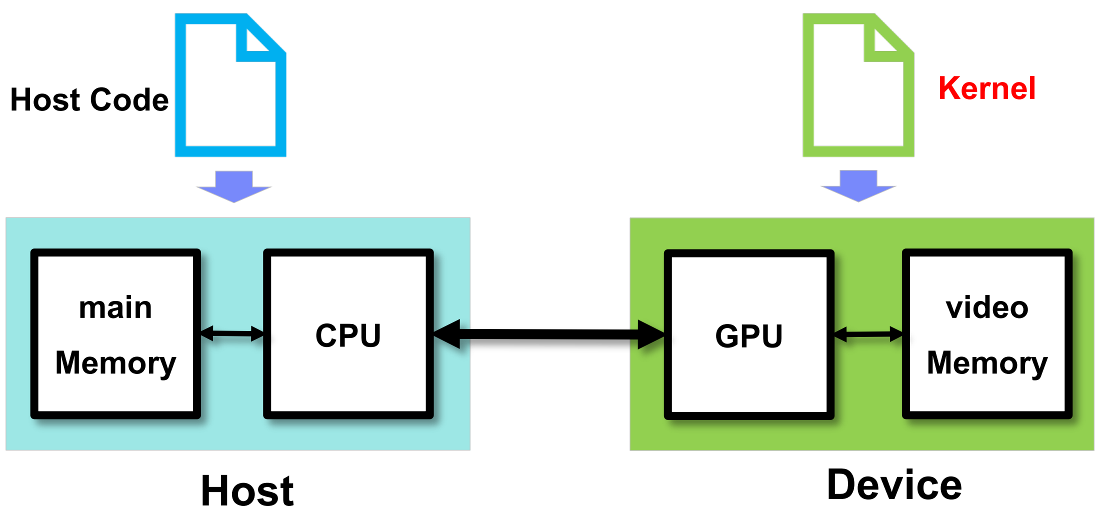
Data Copy

- cudaError_t cudaMemcpy(void* dst,
  void* src,
  size_t nbytes,
  enum cudaMemcpyKind direction);
  - returns after the copy is complete
  - **blocks** CPU thread until all bytes have been copied
  - **doesn't start** copying until previous CUDA calls complete

- **enum cudaMemcpyKind**
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
  - cudaMemcpyHostToHost

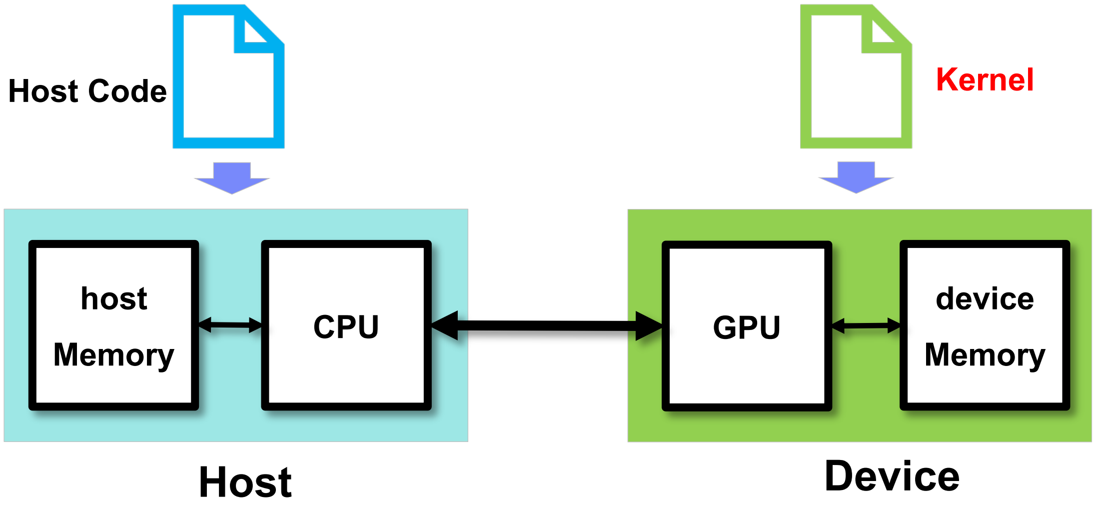
Data Copy

- host → host : memcpy (in C/C++)
- host → device, device → device, device → host : cudaMemcpy (CUDA)

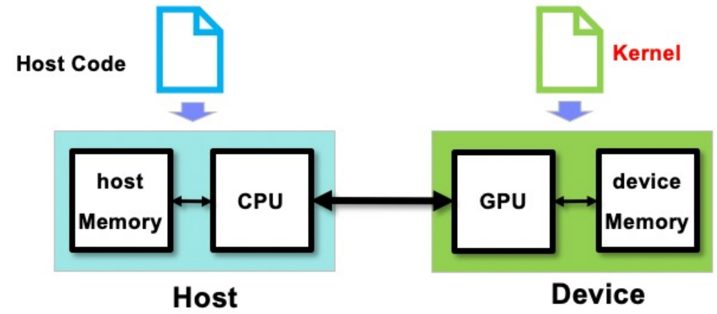


Example: Host-Device Mem copy

- **step 1.**
  - make a block of data
  - print out the source data
- **step 2.**
  - copy from host memory to device memory
  - copy from device memory to device memory
  - copy from device memory to host memory
- **step 3.**
  - print out the result



Code: cuda memcpy (1/4)

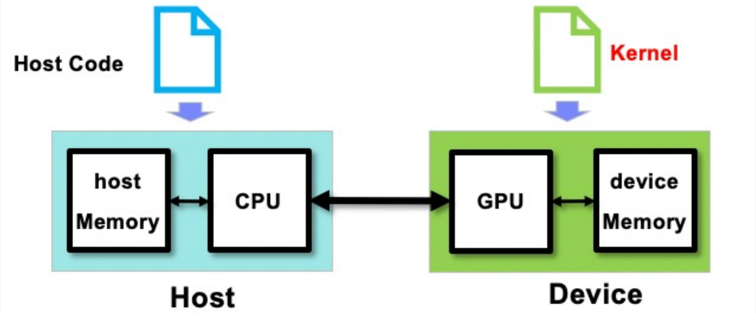



```
#include <iostream>


int main(void) {
    // host-side data
    const int SIZE = 5;
    const int a[SIZE] = { 1, 2, 3, 4, 5 };  // source data
    int b[SIZE] = { 0, 0, 0, 0, 0 }; // final destination

   // print source
    printf("a = {%d,%d,%d,%d,%d}\n", a[0], a[1], a[2], a[3], a[4]);
```

Code: cuda memcpy (2/4)

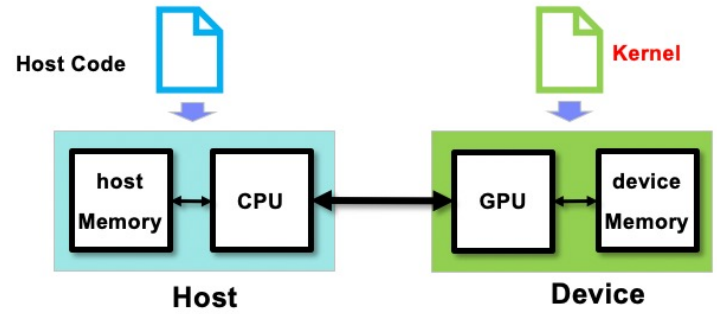



```
// device-side data
int* dev_a = 0;
int* dev_b = 0;

// allocate device memory
cudaMalloc((void**)&dev_a, SIZE * sizeof(int));
cudaMalloc((void**)&dev_b, SIZE * sizeof(int));

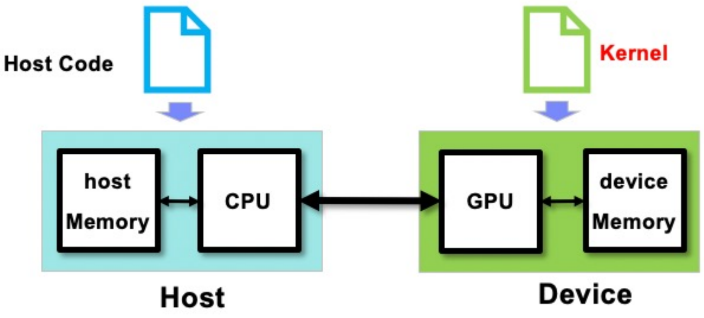
// copy from host to device
cudaMemcpy(dev_a, a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
```

Code: cuda memcpy (3/4)




```
// copy from device to device
cudaMemcpy(dev_b, dev_a, SIZE * sizeof(int),
                                        cudaMemcpyDeviceToDevice);
// copy from device to host
cudaMemcpy(b, dev_b, SIZE * sizeof(int), cudaMemcpyDeviceToHost);
```

Code: cuda memcpy (4/4)




```
    // free device memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    // print the result
    printf("b = {%d,%d,%d,%d,%d}\n", b[0], b[1], b[2], b[3], b[4]);
    // done
    return 0;
}
```

Execution Result

- Compile the source code
  - nvcc memcpy.cu -o ./memcpy

- executing ./memcpy

  a = {1,2,3,4,5}

  b = {1,2,3,4,5}

Error Checking

Error Checking and Handling in CUDA

- It is important for a program to check and handle errors
- CUDA API functions return flags that indicate whether an error has occurred

- Most of them returns error code (or cudaSuccess).
  - **cudaError_t** cudaMalloc(void** devPtr, size_t size);
  - **cudaError_t** cudaFree(void* devPtr);
  - **cudaError_t** cudaMemcpy(void* dst, const void* src, size_t size);

- Example:
  - ```
    if (cudaMalloc(&devPtr, SIZE) != cudaSuccess) {
            exit(1);
    }
    ```
    ```

cudaError_t : data type

- typedef enum cudaError cudaError_t
- possible values:
  - cudaSuccess, cudaErrorMissingConfiguration, cudaErrorMemoryAllocation, cudaErrorInitializationError, cudaErrorLaunchFailure, cudaErrorLaunchTimeout, cudaErrorLaunchOutOfResources, cudaErrorInvalidDeviceFunction, cudaErrorInvalidConfiguration, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidSymbol, cudaErrorUnmapBufferObjectFailed, cudaErrorInvalidHostPointer, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture, cudaErrorInvalidTextureBinding, cudaErrorInvalidChannelDescriptor, cudaErrorInvalidMemcpyDirection, cudaErrorInvalidFilterSetting, cudaErrorInvalidNormSetting, cudaErrorUnknown, cudaErrorNotYetImplemented, cudaErrorInvalidResourceHandle, cudaErrorInsufficientDriver, cudaErrorSetOnActiveProcess, cudaErrorStartupFailure, cudaErrorApiFailureBase

cudaGetErrorName(*err*)

- **const char* cudaGetErrorName(cudaError_t *err*)**
  - *err* : error code to convert to string
  - returns:
    - char* to a NULL-terminated string
    - NULL if the error code is not valid

- cout << cudaGetErrorName(cudaErrorMemoryAllocation)
  << endl;
- cout << cudaGetErrorName(cudaErrorInvalidValue)
  << endl;
  - shows:
    cudaErrorMemoryAllocation
    cudaErrorInvalidValue

cudaGetErrorString(*err*)

- **const char* cudaGetErrorString(cudaError_t *err*)**
  - *err* : error code to convert to string
  - returns:
    - char* to a NULL-terminated string
    - NULL if the error code is not valid

- cout << cudaGetErrorString(cudaErrorMemoryAllocation)
  << endl;
- cout << cudaGetErrorString(cudaErrorInvalidValue)
  << endl;
  - shows:
    out of memory
    invalid argument

cudaGetLastError(void)

- cudaError_t cudaGetLastError(void)
  - returns the last error due to CUDA runtime calls in the same host thread
  - and **resets** it to **cudaSuccess**
  - So, if no CUDA error since the last call, it returns **cudaSuccess**
  - For multiple errors, it contains the last error only.

- cudaError_t cudaPeekAtLastError(void)
  - returns the last error due to CUDA runtime calls in the same host thread
  - Note that this call does **NOT** reset
  - So, the last error code is still available

A simple CUDA error check code

```
cudaMemcpy( … );
cudaError_t  e = cudaGetLastError();
if (e != cudaSuccess) {
        printf("cuda failure %s:%d: '%s'\n",
          __FILE__, __LINE__,
          cudaGetErrorString(e) );
        exit(0);
}
```

A simple CUDA error check macro

```
#define   cudaCheckError( )        do { \
        cudaError_t  e = cudaGetLastError();  \
        if (e != cudaSuccess) { \
                printf("cuda failure %s:%d: '%s'\n", \
                  __FILE__, __LINE__, \
                  cudaGetErrorString(e) ); \
                exit(0); \
        } \
} while (0)
```

Example

- code segment

```
// allocate device memory
cudaMalloc((void**)&dev_a, sizeof(int));
cudaMalloc((void**)&dev_b, sizeof(int));
cudaMalloc((void**)&dev_c, sizeof(int));
cudaCheckError( );
```

More advanced macro

```
#ifdef DEBUG // debug mode
#define CUDA_CHECK(x)       do {\
        (x); \
        cudaError_t e = cudaGetLastError(); \
        if (cudaSuccess != e) { \
                printf("cuda failure %s at %s:%d\n", \
                  cudaGetErrorString(e), \
                  __FILE__, __LINE__); \
                exit(1); \
        } \
} while (0)
#else
#define CUDA_CHECK(x)       (x)         // release mode
#endif
```

error_check.cu

```
#include <iostream>


#ifdef DEBUG
#define CUDA_CHECK(x)   do {\
        (x); \
        cudaError_t e = cudaGetLastError(); \
        if (cudaSuccess != e) { \
            printf("cuda failure \"%s\" at %s:%d\n", \
                   cudaGetErrorString(e), \
                   __FILE__, __LINE__); \
            exit(1); \
        } \
    } while (0)
#else
#define CUDA_CHECK(x)    (x)
#endif
```

error_check.cu

```
// main program for the CPU
 int main(void) {
    // host-side data
    const int SIZE = 5;
    const int a[SIZE] = { 1, 2, 3, 4, 5 };
    int b[SIZE] = { 0, 0, 0, 0, 0 };
    // print source
    printf("a = {%d,%d,%d,%d,%d}\n", a[0], a[1], a[2], a[3], a[4]);
    // device-side data
    int *dev_a = 0;
    int *dev_b = 0;
    // allocate device memory
    CUDA_CHECK( cudaMalloc((void**)&dev_a, SIZE * sizeof(int)) );
    CUDA_CHECK( cudaMalloc((void**)&dev_b, SIZE * sizeof(int)) );
    // copy from host to device
    CUDA_CHECK( cudaMemcpy(dev_a, a, SIZE * sizeof(int), cudaMemcpyDeviceToDevice) ); // BOMB here !
    // copy from device to device
    CUDA_CHECK( cudaMemcpy(dev_b, dev_a, SIZE * sizeof(int), cudaMemcpyDeviceToDevice) );
    // copy from device to host
    CUDA_CHECK( cudaMemcpy(b, dev_b, SIZE * sizeof(int), cudaMemcpyDeviceToHost) );
    // free device memory
    CUDA_CHECK( cudaFree(dev_a) );
    CUDA_CHECK( cudaFree(dev_b) );
    // print the result
    printf("b = {%d,%d,%d,%d,%d}\n", b[0], b[1], b[2], b[3], b[4]);
    // done
    return 0;
 }
```

Execution Result

- Compile the source code
  - nvcc error_check.cu -DDEBUG -o ./error_check
  - nvcc error_check.cu -o ./error_check

- executing ./error_check

  a = {1,2,3,4,5}

  b = {……….}

Agenda

- What is CUDA?
- Device Global Memory and Data Transfer
- Error Checking
- **A Vector Addition Kernel**
- **Kernel Functions and Threading**
- **Kernel Launch**

CUDA Resources

- CUDA API reference:
  - http://docs.nvidia.com/cuda/index.html
  - http://docs.nvidia.com/cuda/cuda-runtime-api/index.html
- CUDA course:
  - https://developer.nvidia.com/cuda-education-training
  - https://developer.nvidia.com/cuda-training