

Basics of GPU and Cloud Computing

IEEE SPS Workshop on ASR

Dr. K. T. Deepak

Asst. Prof., IIIT Dharwad

Parallelism in CPU

- Gordon Moore of Intel - stated a rule, which said that every passing year, the clock frequency of a semiconductor core doubles.
- This law held true until recent years.
- Now, as the clock frequencies of a single core reach saturation points.
- The paradigm has shifted to multi-core and many-core processors.

Parallelism in CPU

- Mainly 3 Types of Parallelism can be seen
 1. Pipelining
 2. Superscalar
 3. Simultaneous Multithreading
 4. Many Core Processors

Parallelism in CPU

Following are the five essential steps required for an instruction to finish –

- Instruction fetch (IF)
- Instruction decode (ID)
- Instruction execute (Ex)
- Memory access (Mem)
- Register write-back (WB)

This is a basic five-stage RISC architecture.

Parallelism in CPU

- Instruction Level Parallelism is one way to speed up the processor

| Instr. No. | Pipeline Stage | | | | | | |
|-------------|----------------|----|----|-----|-----|-----|-----|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | IF | ID | EX | MEM |
| 5 | | | | | IF | ID | EX |
| Clock Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Pipeline Hazard

Pipeline Hazards may arise. Consider the situation below –

- I1 – ADD 1 to R5
- I2 – COPY R5 to R6

Superscalar Architecture

- Instruction Level Parallelism (ILP) is also implemented by implementing a superscalar architecture.
- The primary difference between a superscalar and a pipelined processor is: the former uses multiple execution units (on the same chip) to achieve ILP.
- This means that in superscalar, several instructions can simultaneously be in the same stage of the execution cycle.

Simultaneous Multithreading

- Many a times, a processor core is utilizing only a fraction of its resources to execute instructions.
- What HT does is that it takes a few more CPU registers, and executes more instructions on the part of the core that is sitting idle.
- Thus, one core now appears as two core.
- It is to be considered that they are not completely independent.
- If both the 'cores' need to access the CPU resource, one of them ends up waiting.
- That is the reason why we cannot replace a dual-core CPU with a hyper-threaded, single core CPU.

Many Core Processors

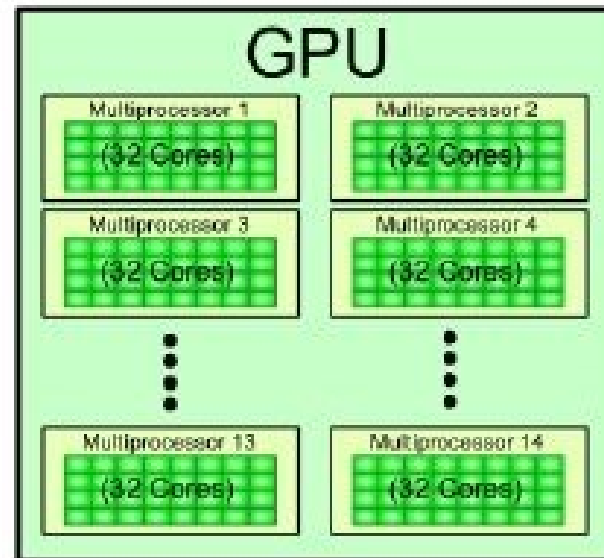
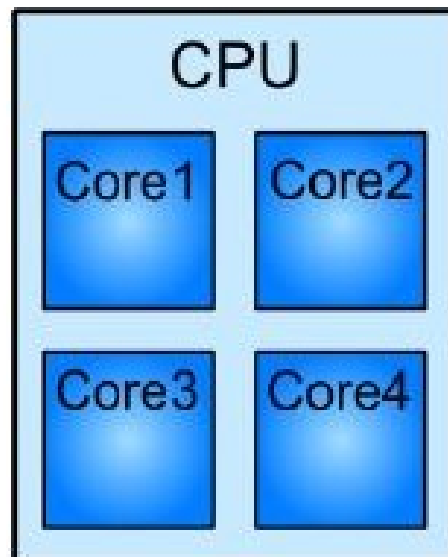
- The other paradigm is many-core processors that are designed to operate on large chunks of data, in which CPUs prove inefficient.
- A GPU comprises many cores (that almost double each passing year), and each core runs at a clock speed significantly slower than a CPU's clock.
- GPUs focus on execution throughput of massively-parallel programs.

Many Core Processors

- For example, the Nvidia GeForce GTX 280 GPU has 240 cores
 1. Each of which is a heavily multithreaded,
 2. Single-instruction issue processor (SIMD – single instruction, multiple-data)
 3. *Shares its control and instruction cache with seven other cores.*
- **Floating Point Operations per Second (FLOPS)**, GPUs have been leading the race for a long time now.

Difference b/n CPU and GPU

CPU/GPU Architecture Comparison



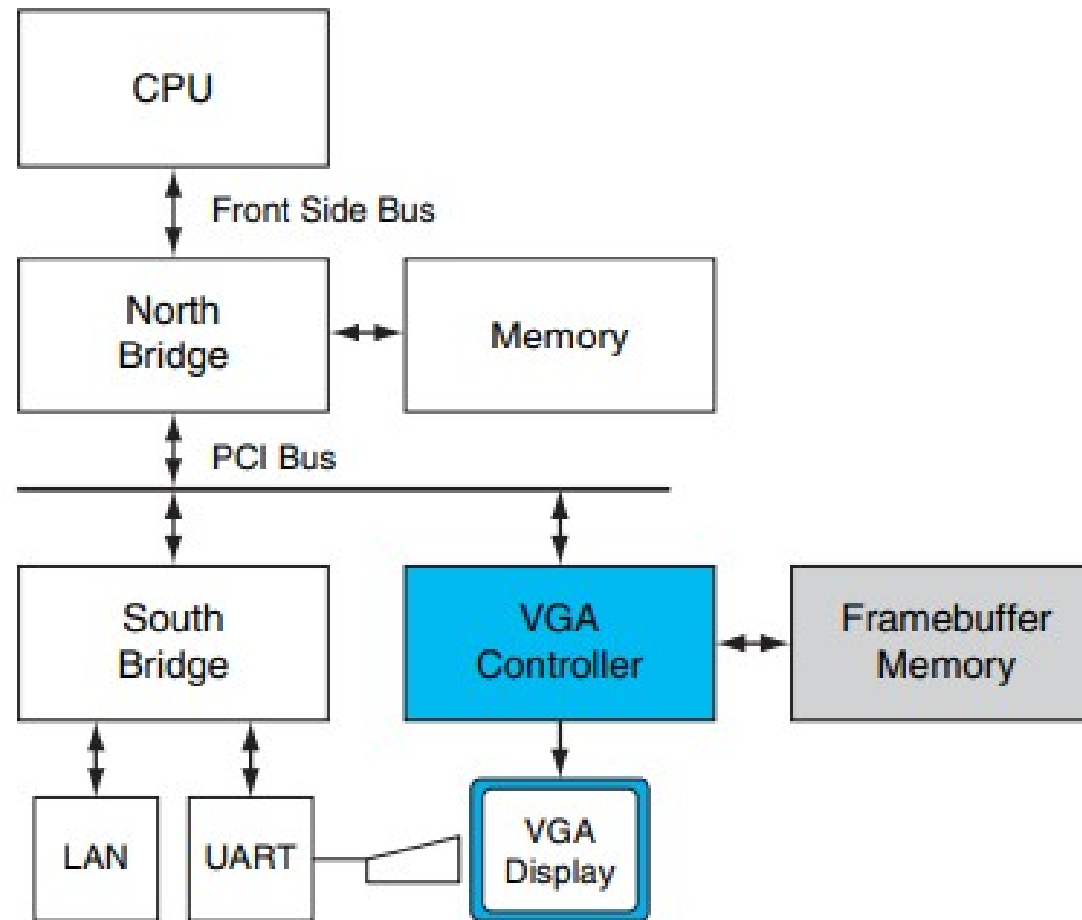
Brief History of GPU Evolution

- Fifteen years ago, there was no such thing as a GPU.
- Graphics on a PC were performed by a video graphics array (VGA) controller.
- A VGA controller was simply a memory controller and display generator connected to some DRAM.
- In the 1990s, semiconductor technology advanced sufficiently that more functions could be added to the VGA controller.

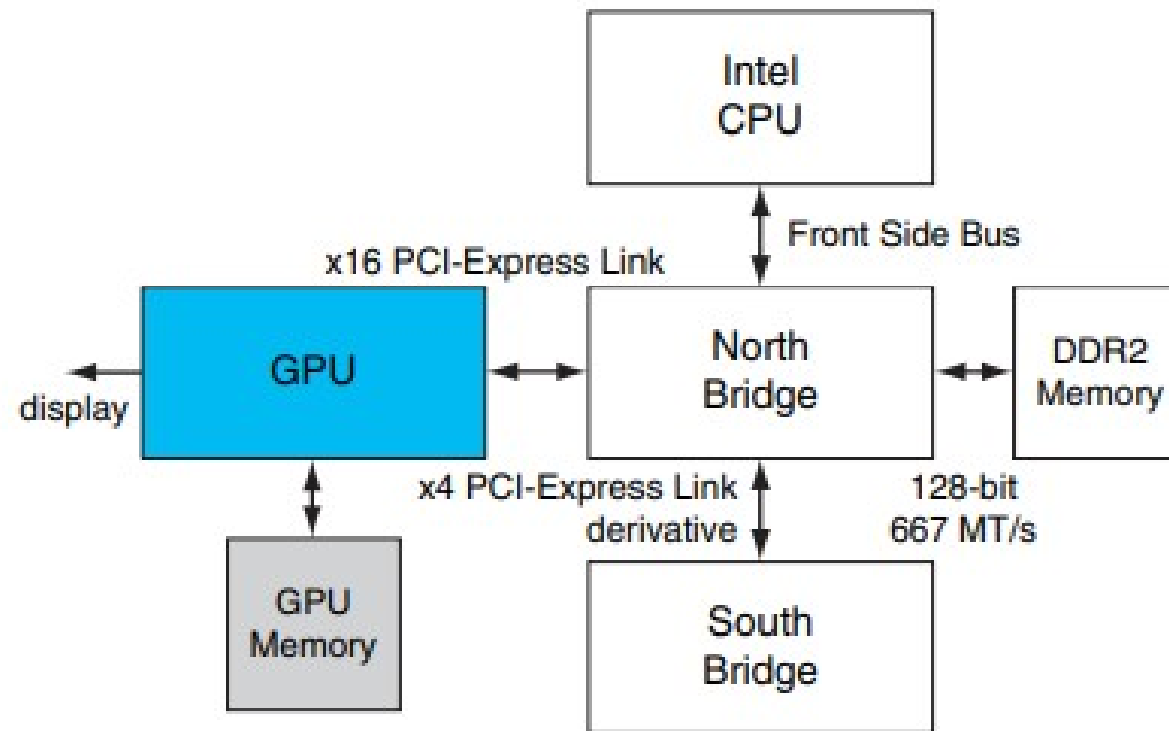
Brief History of GPU Evolution

- By 1997, VGA controllers were beginning to incorporate some three-dimensional (3D) acceleration functions.
- In 2000, the single chip graphics processor incorporated almost every detail of the traditional high-end workstation graphics pipeline.
- The term GPU was coined to denote that the graphics device had become a processor.

A Computer Architecture



A GPU based Machine



Processor Array

- A unified GPU processor array contains many processor cores, typically organized into multithreaded multiprocessors.
- Figure (next slide) shows a GPU with an array of 112 streaming processor (SP) cores.
- These are organized as 14 multithreaded streaming multiprocessors (SM).

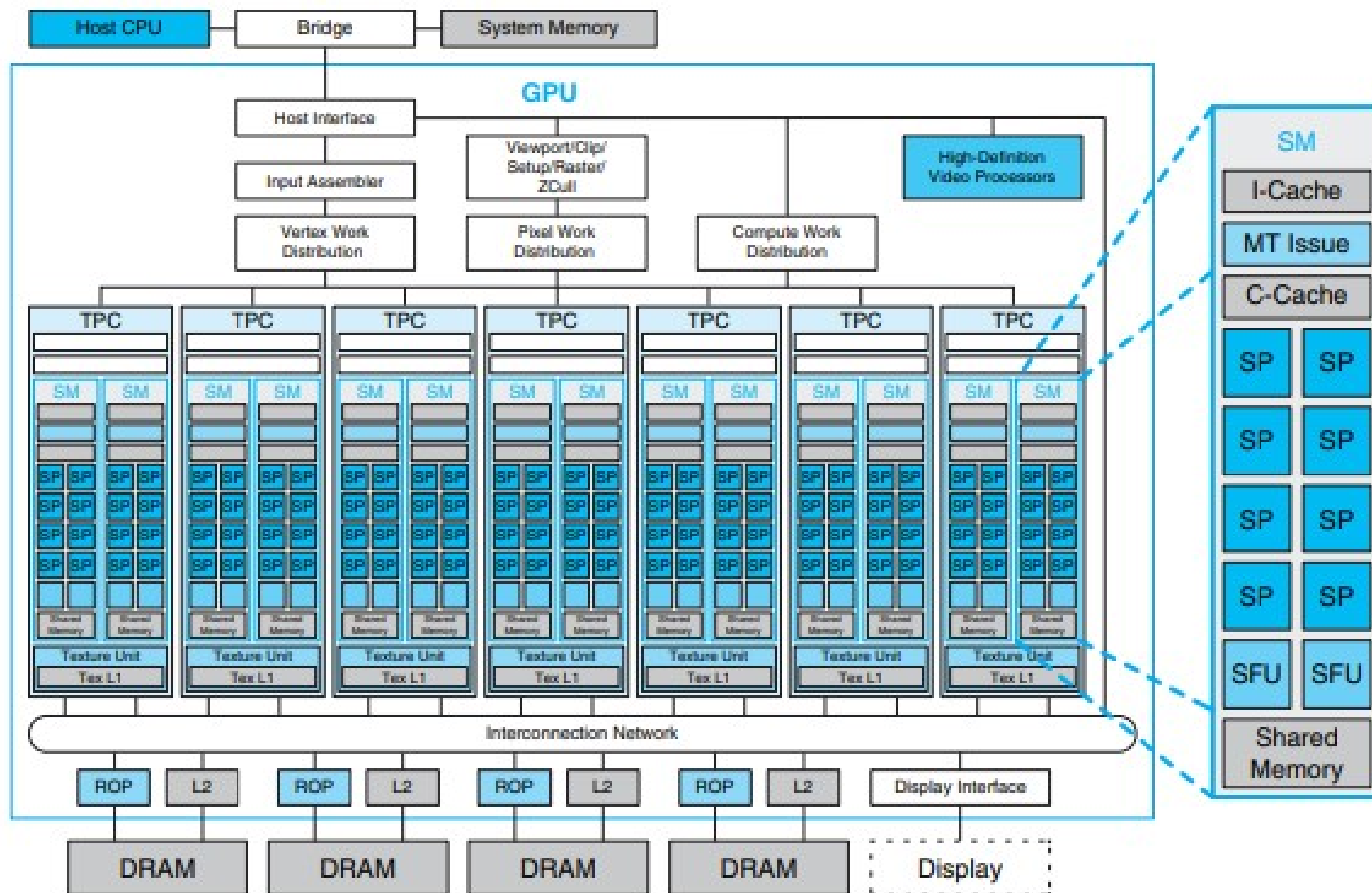
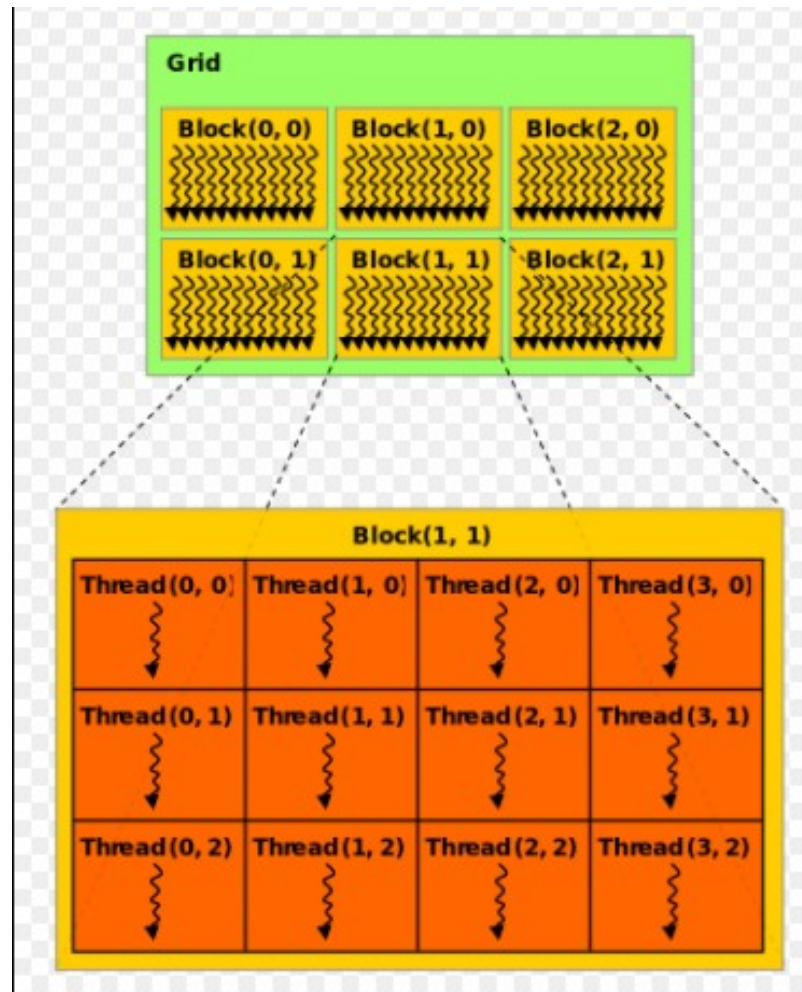


FIGURE Basic unified GPU architecture. Example GPU with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

Processor Array

- Each SP core is highly multithreaded, managing 96 concurrent threads and their state in hardware.
- The processors connect with four 64-bit-wide DRAM partitions via an interconnection network.
- Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.
- This is the basic Tesla architecture implemented by the NVIDIA GeForce 8800.

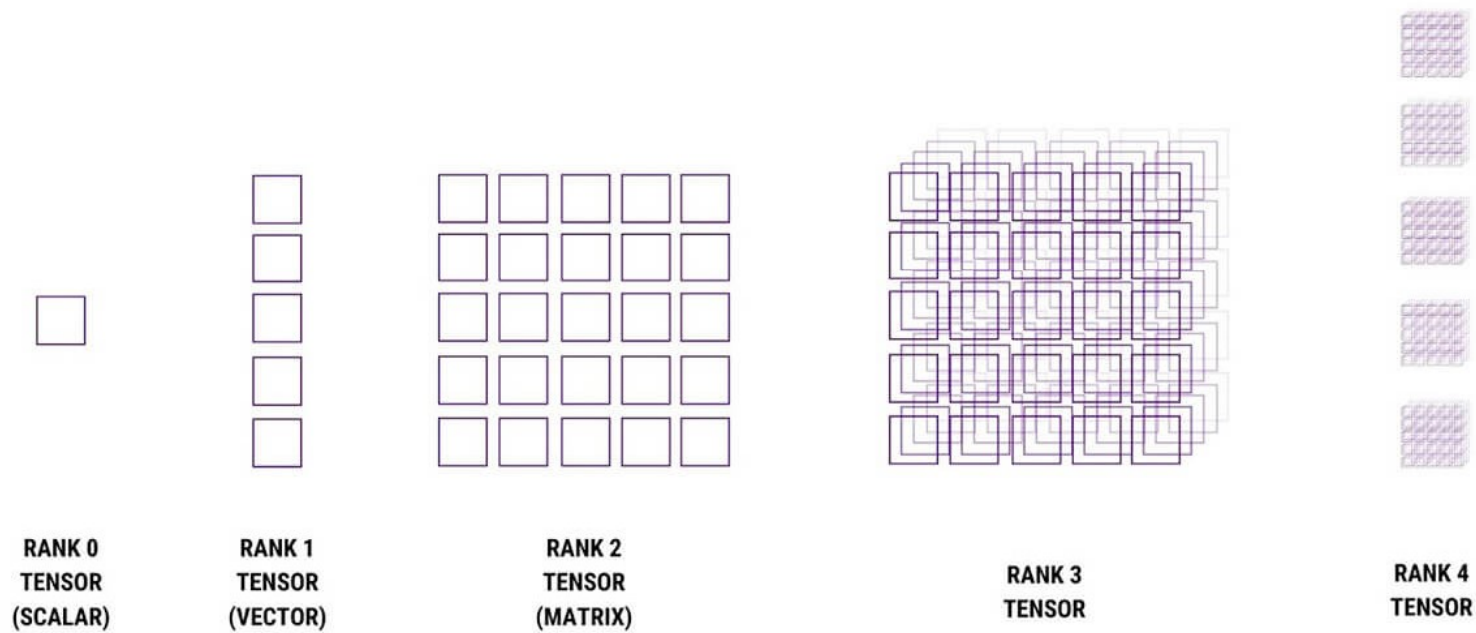
Grid and Threads Per Block



What is a Tensor?

- Tensor is a data type that can represent all types of data.
- It can be thought of as a container in which a multi-dimensional data set can be stored.
- It can be considered as an extension of a matrix.
- Matrices are two-dimensional structures containing numbers, but a tensor is a multidimensional set of numbers.

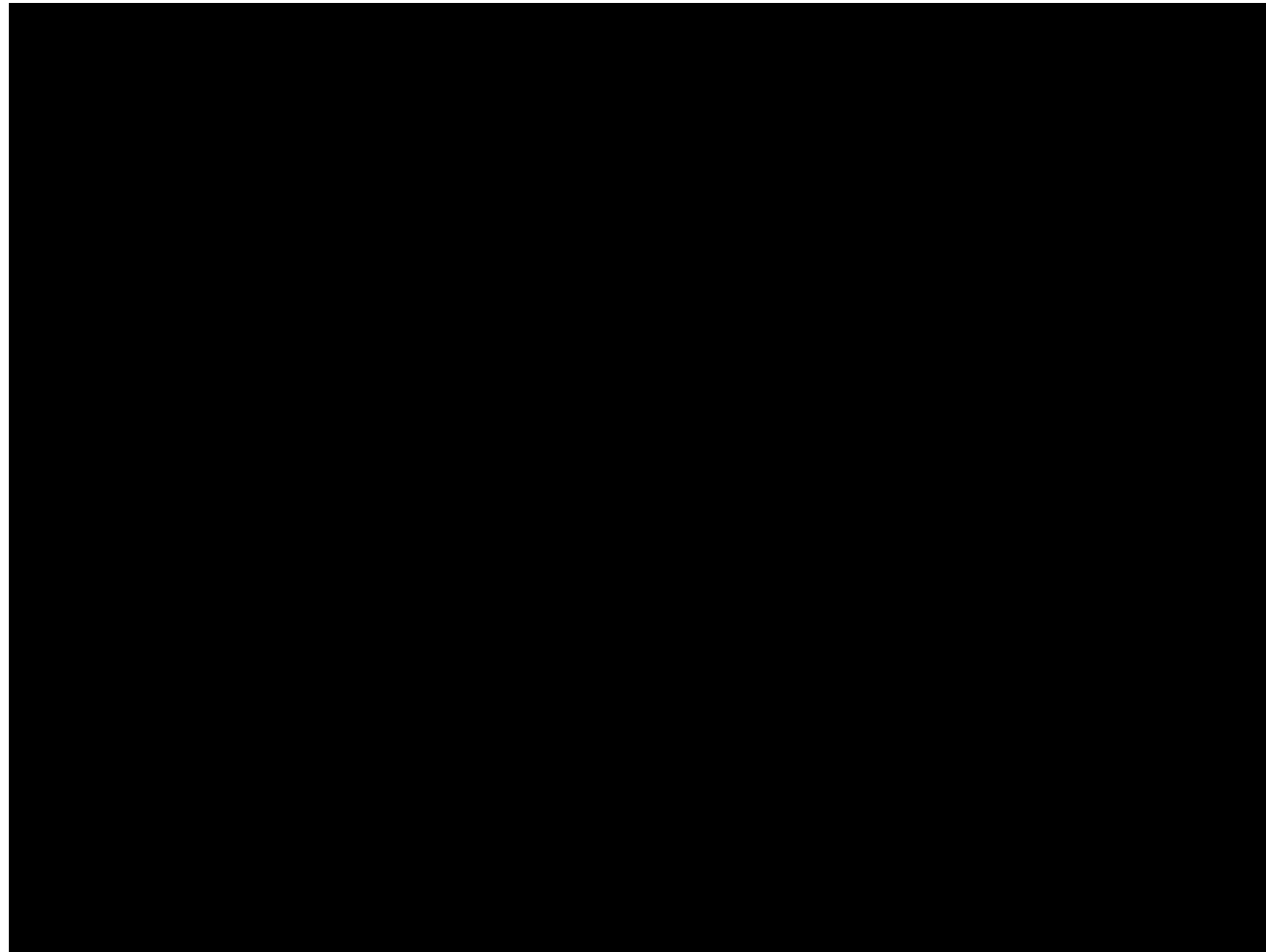
What is a Tensor?



Tensor Core

- All microprocessors are designed to carry out arithmetic and logical operations.
- One arithmetic operation that holds high importance is matrix multiplication.
- Multiplying two 4×4 matrices involves 64 multiplications and 48 additions.
- Convolution and Multiplication are the areas where the new cores shine.
- The computational complexity increases multifold as the size and dimensions of the matrix (tensor) go up.
- Machine Learning, Deep learning, Ray Tracing are tasks that involve an excessive amount of multiplication.

Watch Nvidia's Michael Houston explain Tensor Cores



NVIDIA Tesla V100 Accelerator



Tesla V100 Comparison

| Tesla Product | Tesla K40 | Tesla M40 | Tesla P100 | Tesla V100 |
|--------------------|----------------|-----------------|----------------|---------------|
| GPU | GK180 (Kepler) | GM200 (Maxwell) | GP100 (Pascal) | GV100 (Volta) |
| SMs | 15 | 24 | 56 | 80 |
| TPCs | 15 | 24 | 28 | 40 |
| FP32 Cores / SM | 192 | 128 | 64 | 64 |
| FP32 Cores / GPU | 2880 | 3072 | 3584 | 5120 |
| FP64 Cores / SM | 64 | 4 | 32 | 32 |
| FP64 Cores / GPU | 960 | 96 | 1792 | 2560 |
| Tensor Cores / SM | NA | NA | NA | 8 |
| Tensor Cores / GPU | NA | NA | NA | 640 |

Tensor Core 4x4 Matrix Multiply and Accumulate

Tensor Cores and their associated data paths are custom-designed to dramatically increase floating-point compute throughput with high energy efficiency.

Each Tensor Core operates on a 4x4 matrix and performs the following operation:

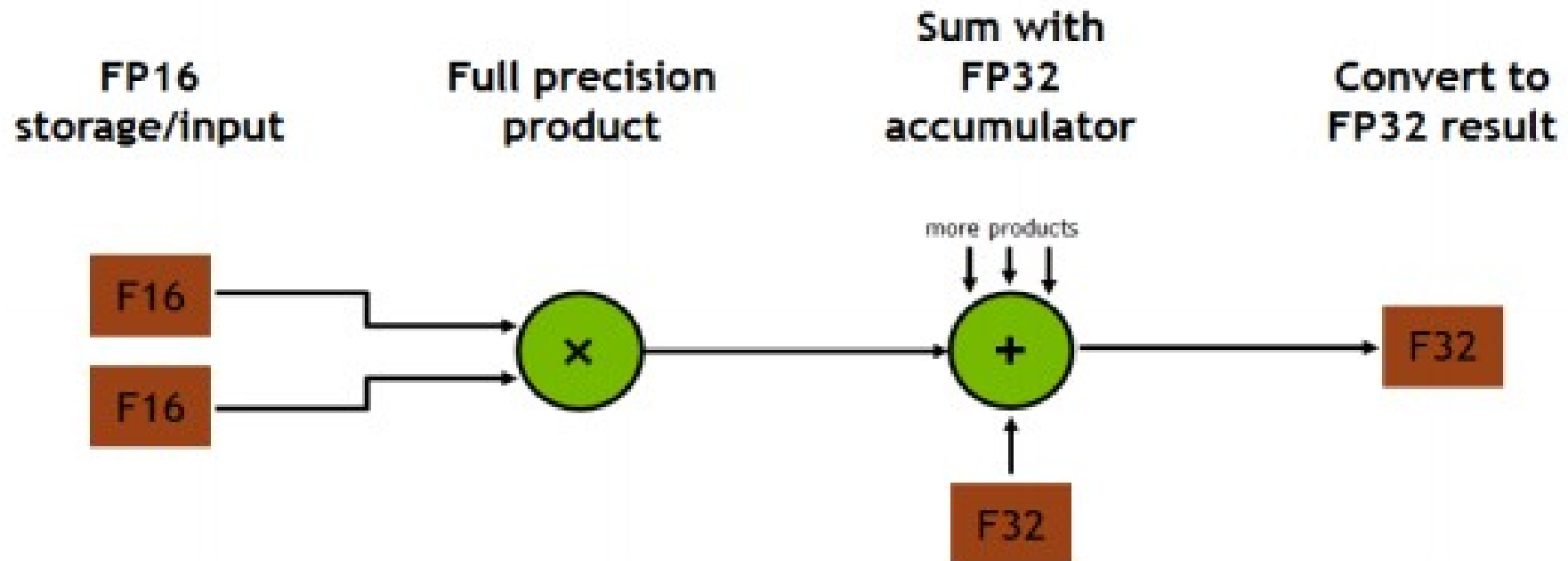
$$D = A \times B + C$$

where A , B , C , and D are 4x4 matrices (Figure 8). The matrix multiply inputs A and B are FP16 matrices, while the accumulation matrices C and D may be FP16 or FP32 matrices

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32

Mixed Precision Multiply and Accumulate in Tensor Core



CUDA

- **CUDA:** an [acronym](#) for **Compute Unified Device Architecture**.
- It is a [parallel computing](#) platform and [application programming interface](#) (API) model created by [Nvidia](#).
- Enables developers to speed up compute-intensive applications.
- Harness the power of GPUs for the parallelizable part of the computation.

CUDA

- In CUDA programming, both CPUs and GPUs are used for computing.
- Typically, we refer to CPU and GPU system as *host* and *device*, respectively.
- CPUs and GPUs are separated platforms with their own memory space.
- Typically, we run serial workload on CPU and offload parallel computation to GPUs.

CUDA v/s C Programming

| C | CUDA |
|---|--|
| <pre>void c_hello(){ printf("Hello World!\n"); } int main() { c_hello(); return 0; }</pre> | <pre>__global__ void cuda_hello(){ printf("Hello World from GPU!\n"); } int main() { cuda_hello<<<1,1>>>(); return 0; }</pre> |

- The major difference between C and CUDA implementation is `__global__` specifier and `<<<...>>>` syntax.
- The `__global__` specifier indicates a function that runs on device (GPU).
- Such function can be called through host code, e.g. the `main()` function in the example, and is also known as "*kernels*".

CUDA Syntax

- When a kernel is called, its execution configuration is provided through `<<<...>>>` syntax, e.g. `cuda_hello<<<1,1>>>()`.
- In CUDA terminology, this is called "*kernel launch*".
- CUDA use a kernel execution configuration `<<<...>>>` to tell CUDA runtime how many threads to launch on GPU.

Thread Block

- CUDA organizes threads into a group called "***thread block***".
- Kernel can launch multiple thread blocks, organized into a "***grid***" structure.
- The syntax of kernel execution configuration is as follows: <<< M , T >>>
- Which indicate that a kernel launches with a grid of M thread blocks.
- Each thread block has T parallel threads.

A Simple Program

```
#define N 10000000

void vector_add(float *out, float *a, float *b, int n) {
    for(int i = 0; i < n; i++){
        out[i] = a[i] + b[i];
    }
}

int main(){
    float *a, *b, *out;

    // Allocate memory
    a = (float*)malloc(sizeof(float) * N);
    b = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // Initialize array
    for(int i = 0; i < N; i++){
        a[i] = 1.0f; b[i] = 2.0f;
    }

    // Main function
    vector_add(out, a, b, N);
}
```

A Simple Program

```
__global__ void vector_add(float *out, float *a, float *b, int n) {  
    for(int i = 0; i < n; i++){  
        out[i] = a[i] + b[i];  
    }  
}
```

```
$> nvcc vector_add.c -o vector_add  
$> ./vector_add
```

Memory

- You will notice that the program does not work correctly.
- The reason is CPU and GPUs are separate entities.
- Both have their own memory space.
- CPU cannot directly access GPU memory, and vice versa.
- CPU memory is called *host memory* and GPU memory is called *device memory*.
- Pointers to CPU and GPU memory are called *host pointer* and *device pointer*, respectively.

Memory

- For data to be accessible by GPU, it must be presented in the device memory.
- CUDA provides APIs for allocating device memory and data transfer between host and device memory.
- Following is the common workflow of CUDA programs.
 1. Allocate host memory and initialized host data
 2. Allocate device memory
 3. Transfer input data from host to device memory
 4. Execute kernels
 5. Transfer output from device memory to host

Device Memory Management

- CUDA provides several functions for allocating device memory.
- The most common ones are `cudaMalloc()` and `cudaFree()`.
- The syntax for both functions are as follow

```
cudaMalloc(void *devPtr, size_t count);  
cudaFree(void *devPtr);
```

Device Memory Management

- `cudaMalloc()` allocates memory of size `count` in the device memory and updates the device pointer `devPtr` to the allocated memory.
- `cudaFree()` deallocates a region of the device memory where the device pointer `devPtr` points to.
- They are comparable to `malloc()` and `free()` in C, respectively

Memory Transfer

- Transferring data between host and device memory can be done through **cudaMemcpy** function.
- This is similar to **memcpy** in C.
- The syntax of cudaMemcpy is as follow

```
cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind)
```

- The function copy a memory of size count from src to dst.
- kind indicates the direction.
- For typical usage, the value of kind is either **cudaMemcpyHostToDevice** or **cudaMemcpyDeviceToHost**.

Vector Addition

```
void main(){
    float *a, *b, *out;
    float *d_a;

    a = (float*)malloc(sizeof(float) * N);

    // Allocate device memory for a
    cudaMalloc((void**)&d_a, sizeof(float) * N);

    // Transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

    ...
    vector_add<<<1,1>>>(out, d_a, b, N);
    ...

    // Cleanup after kernel execution
    cudaFree(d_a);
    free(a);
}
```


Profiling Performance

- NVIDIA provides a commandline profiler tool called nvprof
- To profile our vector addition, use following command

```
$> nvprof ./vector_add
```

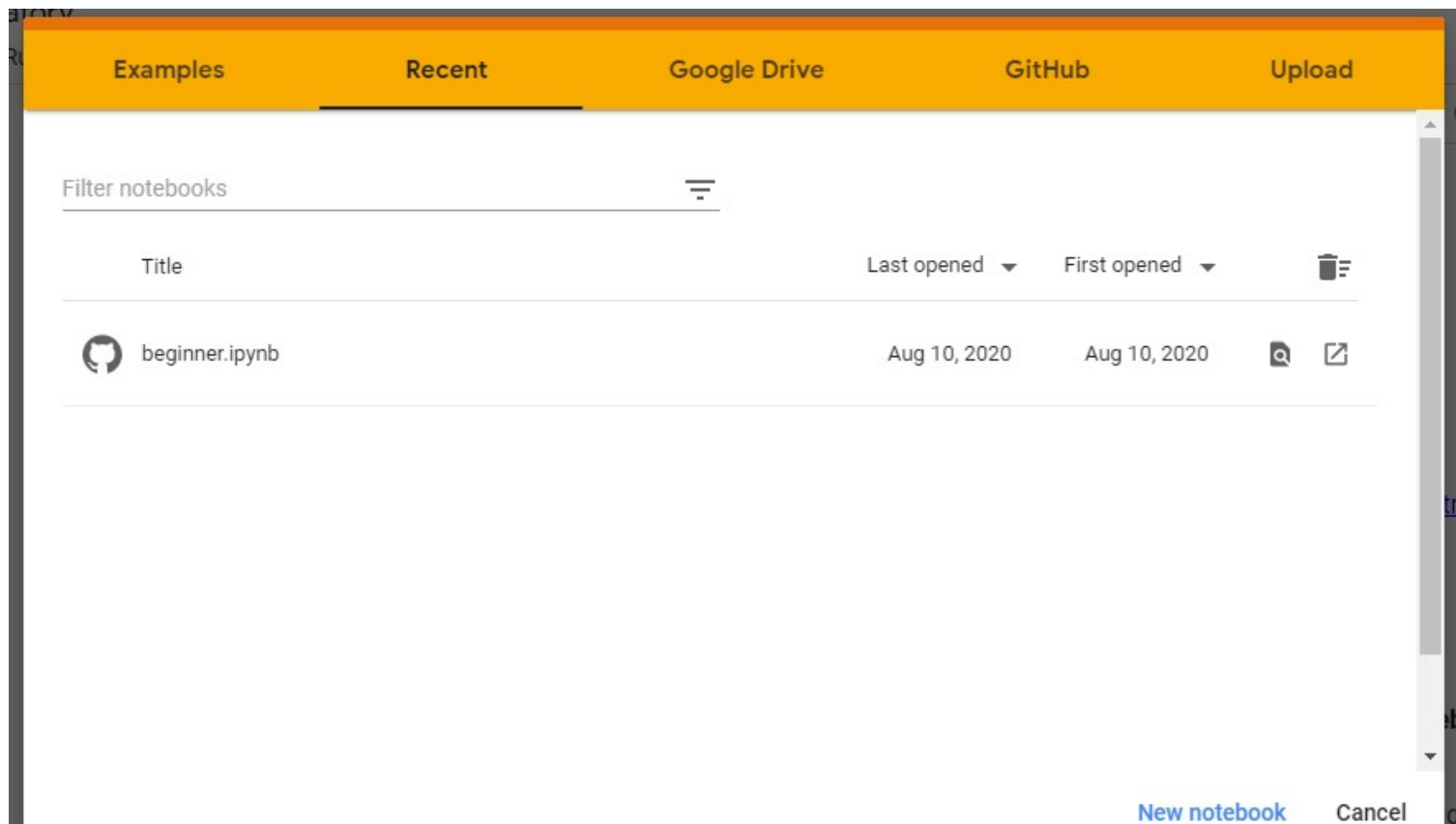
```
==6326== Profiling application: ./vector_add
==6326== Profiling result:
Time(%)      Time       Calls      Avg       Min       Max  Name
97.55%    1.42529s         1  1.42529s  1.42529s  1.42529s  vector_add(float*, float*, float*, int)
  1.39%    20.318ms         2   10.159ms  10.126ms  10.192ms  [CUDA memcpy HtoD]
  1.06%    15.549ms         1   15.549ms  15.549ms  15.549ms  [CUDA memcpy DtoH]
```

Google Colab

- Google Colab is a free cloud service.
- Most important feature able to distinguish Colab from other free cloud services is; Colab offers GPU and is completely free!
- With Colab you can work on the GPU with CUDA C/C++ for free!
- CUDA code will not run on AMD CPU or Intel HD graphics unless you have NVIDIA hardware inside your machine.
- On Colab you can take advantage of Nvidia GPU as well as being a fully functional Jupyter Notebook with pre-installed Tensorflow

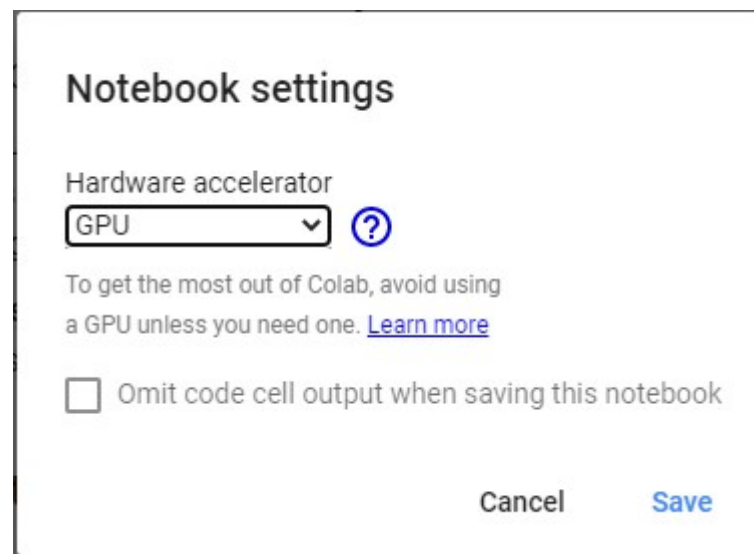
Configure Learning Environment – Step 1

- <https://colab.research.google.com/notebooks/intro.ipynb#recent=true>



Configure Learning Environment – Step 2

- **Step 2: We need to switch our runtime from CPU to GPU. Click on Runtime > Change runtime type > Hardware Accelerator > GPU > Save.**



Notebook settings

Hardware accelerator

GPU

To get the most out of Colab, avoid using a GPU unless you need one. [Learn more](#)

☐ Omit code cell output when saving this notebook

Cancel Save

Configure Learning Environment – Step 3

- **Step 3: Completely uninstall any previous CUDA versions. We need to refresh the Cloud Instance of CUDA.**

```
!apt-get --purge remove cuda nvidia* libnvidia-*
```

```
!dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge
```

```
!apt-get remove cuda-*
```

```
!apt autoremove
```

```
!apt-get update
```

Configure Learning Environment – Step 4

- **Step 4: Install CUDA Version 9**

```
!wget https://developer.nvidia.com/compute/cuda/9.2/Prod/local_installers/cuda-  
repo-ubuntu1604-9-2-local_9.2.88-1_amd64 -O cuda-repo-ubuntu1604-9-2-  
local_9.2.88-1_amd64.deb
```

```
!dpkg -i cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
```

```
!apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub
```

```
!apt-get update !apt-get install cuda-9.2
```

Configure Learning Environment – Step 5

- **Step 5: Now you can check your CUDA installation by running the command given below :**

`!nvcc –version`

- **Output**

`nvcc: NVIDIA (R) Cuda compiler driver Copyright (c) 2005-2018 NVIDIA Corporation Built on Wed_Apr_11_23:16:29_CDT_2018 Cuda compilation tools, release 9.2, V9.2.88`

Configure Learning Environment – Step 6

- **Step 6: Run the given command to install a small extension to run nvcc from the Notebook cells.**

```
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
```


Configure Learning Environment – Step 6

- **Step 7: Load the extension using the code given below:**

```
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
```

Configure Learning Environment – Step 7

- **Step 7: Load the extension using the code given below:**

```
%load_ext nvcc_plugin
```

Configure Learning Environment – Step 8

- **Step 8: Execute the code given below to check if CUDA is working or not.**
- **We are Ready!**

First Program

```
% % cu
#include <iostream>
int main()
{
std::cout << "Welcome To IEEE Workshop\n";
return 0;
}
```

Commands to Compile C and CUDA Programs

- %%writefile vector_add_cu.cu
- %%writefile vector_add_c.c
- Check the file using ls -l
- %%shell

```
gcc vector_add_c.c -o output_vector_add_c  
./output_vector_add_c
```

- %%shell

```
nvcc vector_add.cu -o output_vector_add_cu  
./output_vector_add_cu
```

- %%shell

```
time ./output_vector_add_cu
```

Profiling

- Using **Time** Profiling
- **real**: obviously, as close to a real-time captured lag as possible
- **user**: time spent executing application text (non-kernel-space code)
- **sys**: time spent executing kernel-space code

References

https://www.tutorialspoint.com/cuda/cuda_introduction_to_the_gpu.htm

<https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>

<https://www.techcenturion.com/tensor-cores/>

<https://www.geeksforgeeks.org/how-to-run-cuda-c-c-on-jupyter-notebook-in-google-colaboratory/>

<https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/#introduction>

<https://cppsecrets.com/users/1102811497104117108109111104116975048484864103109971051084699111109/How-to-run-C-and-C00-on-Google-Colab.php>