GPU Architectures and Programming

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

December 12, 2019

Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

Compute Unified Device Architecture

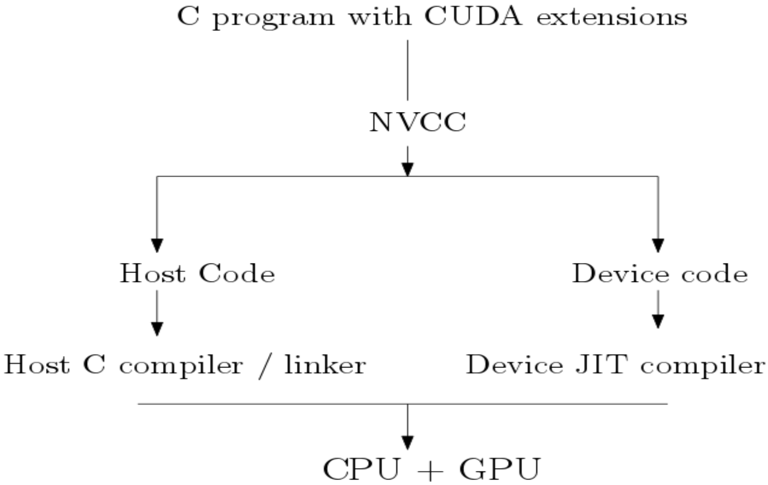
- CUDA C is an extension of C programming language with special constructs for supporting parallel computing
- CUDA programmer perspective - CPU is a *host* : dispatches parallel jobs to GPU *devices*

CUDA program structure

- *host code* for a host device (CPU)
- device code for GPU(s)
- Any C program is a valid CUDA host code
- In general CUDA programs (host + device) code cannot be compiled by standard C compilers

NVIDIA C compiler (NVCC)

The compilation flow



C program with CUDA extensions

NVCC

Host Code

Device code

Host C compiler / linker

Device JIT compiler

CPU + GPU

The execution flow

CPU serial code

⇓

GPU parallel kernel

⇓

CPU serial code

⇓

GPU parallel kernel

Examples : Vector addition CPU only

```
void vecAdd(float* h_A, float* h_B,
float* h_C, int n)
{
 for (i = 0; i < n; i++)
   h_C[i] = h_A[i] + h_B[i];
}
int main()
{
  float *h_A,*h_B,*h_C;
  int n;
  h_A=(float*)malloc(n*sizeof(float))
  h_B=(float*)malloc(n*sizeof(float))
  h_C=(float*)malloc(n*sizeof(float))
  vecAdd(h_A, h_B, h_C, N);
}
```

Examples : Vector addition CPU-GPU

```
#include <cuda.h>
#include <cuda_runtime.h>
__global__ void vectorAdd(float*, float*, float*, int);
/*--------------------------------------------------------*/
__global__
void vectorAdd(float* A, float* B,
float* C, int n){ //CUDA kernel definition
 int i=threadIdx.x+blockDim.x*blockIdx.x;
 if(i<n)
         C[i] = A[i] + B[i];
}
/*--------------------------------------------------------*/
void vecAdd(float* h_A, float*h_B,
float* h_C, int n)
{//host program
   int size = n* sizeof(float);
   float *d_A=NULL, *d_B=NULL, *d_C=NULL;

   // Error code to check return values for CUDA calls
   cudaError_t err = cudaSuccess;
```

Device Memory Allocation

```
err = cudaMalloc((void **)&d_A, size);
if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n",
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
err = cudaMalloc((void **)&d_B, size);
if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n",
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
err = cudaMalloc((void **)&d_C, size);
if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n",
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
```

Host to Device Data Transfer

```
printf("Copy input data from the host memory to the CUDA device\n");
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to copy vector A from host to device (error code %s)
      !\n", cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}

err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to copy vector B from host to device (error code %s)
      !\n", cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
```

Kernel Launch

```
int threadsPerBlock = 256;
int blocksPerGrid =(n+threadsPerBlock-1)/threadsPerBlock;
printf("CUDA kernel launch with %d blocks of %d threads\n",threadsPerBlock,
    blocksPerGrid);
vectorAdd<<<blocksPerGrid,threadsPerBlock>>>(d_A, d_B, d_C, n);
err = cudaGetLastError();
// device function (CUDA kernel) called from host does not have return type
//CUDA runtime functions (execute in host side) can have return type

if (err != cudaSuccess)
{
  fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)!\n",
      cudaGetErrorString(err));
  exit(EXIT_FAILURE);
}
```

Device to Host Memory Transfer

```
printf("Copy output data from the output device to the host memory\n");
 err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
 if (err != cudaSuccess)
 {
   fprintf(stderr, "Failed to copy vector C from device to host (error code %s
       )!\n", cudaGetErrorString(err));
   exit(EXIT_FAILURE);
 }
 cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
 // Verify that the result vector is correct
 for (int i = 0; i < n; ++i)
 {
        if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5)
        {
          fprintf(stderr, "Result verification failed at element %d!\n", i);
          exit(EXIT_FAILURE);
        }
 }
 printf("Test PASSED");
} // End of Function
```

Compile and Run

```
nvcc kernel.cu host.cu -o output

./output
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
```

Observations

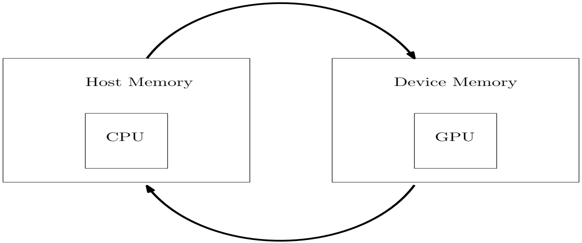


Figure: CPU/GPU Mem Layout

- cuda.h → includes during compilation CUDA API functions and CUDA system variables
- h_A, h_B, h_C → arrays mapped to main memory locations

Observations

```
cudaMalloc((void **)&d_A, size);
//allocate memory segment from GPU global memory
//expects a generic pointer (void **)
//the low level function is common for all object types
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
//transfer data from CPU to GPU memory
//d_A cannot be dereferenced in host code
```

Observations

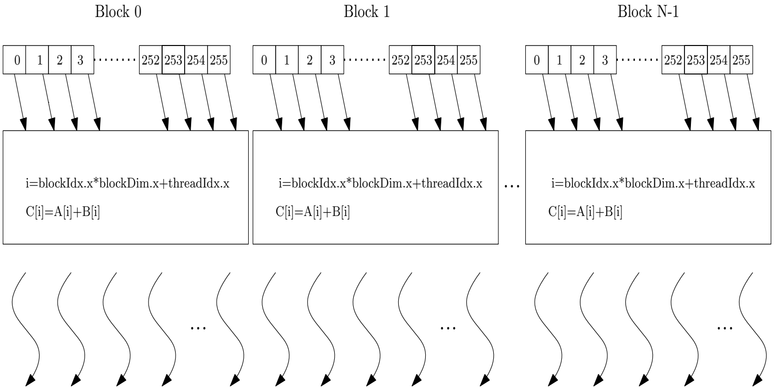
```
//d_A cannot be dereferenced in host code
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
//transfer data from GPU to CPU memory
//can also transfer among different device mem locations
//can also transfer data host to host- we do not need that
//cannot transfer data among different GPU devices
cudaFree(d_A);
//free GPU global memory
```

CUDA kernel

A CUDA kernel when invoked launches multiple threads arranged in a 2 level hierarchy, check the device fn call.

```
vectorAdd<<<ceil(n/256),256>>>
(d_A, d_B, d_C, n)
```

- The call specifies a **grid** of threads to be launched
- the grid is arranged in a hierarchical manner
- (no. of blocks, no. of thread per block)
- all blocks contain same no. of threads (max 1024)
- blocks can be numbered as (_,_,_) triplets : more on this later



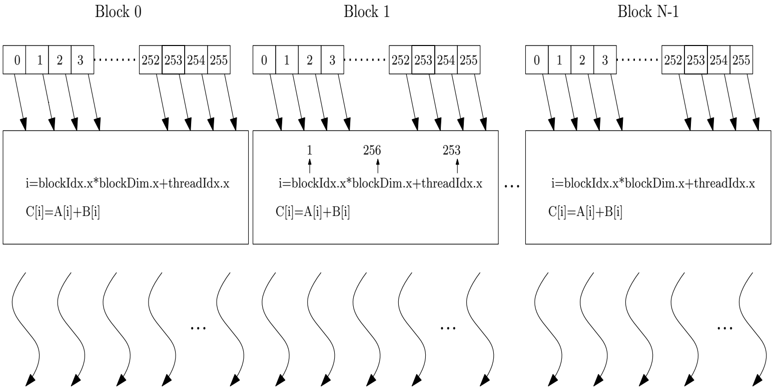
Block 0
Block 1
Block N-1
0
1
2
3
252
253
254
255
i=blockIdx.x*blockDim.x+threadIdx.x
C[i]=A[i]+B[i]
...

Kernel specific system vars

- **gridDim** - no. of blocks in the grid
- **gridDim.x** - no. of blocks in dimension x of multi-dim grid !!
- **blockDim** - no. of threads/block
- **blockDim.x** - no. of threads/block in dimension x of multi-dim block !!
- For single dimension defn of block composition in grid, **blockDim** = **blockDim.x**
- **blockidx.x** = block number for a thread
- **threadidx.x** = thread no. inside a block

```
__global__
void vectorAdd(float* A, float* B,
float* C, int n){
 int i=threadIdx.x+blockDim.x*blockIdx.x;
 if(i<n)
         C[i] = A[i] + B[i];
}
```

- The code is executed by all the threads in the grid
- Every thread has a unique combination of (blockIdx.x, threadIdx.x) which maps to a unique value of **i**
- **i** is private to each thread



Block 0
Block 1
Block N-1
0
1
2
3
252
253
254
255
1
256
253
i=blockIdx.x*blockDim.x+threadIdx.x
C[i]=A[i]+B[i]
...

Function declaration Keywords

```
__global__
void vectorAdd(float* A, float* B, float* C, int n)
```

Table: CUDA Keywords for functions and their scope

Keywords and Functions	Executed on the	Only callable from the
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

CUDA functions

- Every function is a default __host__ function (if not having any CUDA keywords)
- A function can be declared as both __host__ and __device__ function
  - "__host__ __device__ fn()"
  - Runtime system generates two object files, one can be called from host fn()s, another from device fn()s
- __global__ functions can also be called from the device using CUDA kernel semantics (<<< ... >>>) if you are using *dynamic parallelism* - that requires CUDA 5.0 and compute capability 3.5 or higher.

CUDA functions : more observations

- __device__ functions can have a return type other than void but __global__ functions must always return void
- __global__functions can be called from within other kernels running on the GPU to launch additional GPU threads (as part of CUDA dynamic parallelism model) while __device__ functions run on the same thread as the calling kernel.

Matrix Multiplication (CPU only)

```
void MatrixMulKernel(float* M, float* N, float* P, int N){
for(int i=0;i<N;i++)
 for(int j=0;j<N;j++)
 {
  float Pvalue=0.0;
  for (int k = 0; k < N; ++k)
  {
    Pvalue += M[i][k]*N[k][j];
  }
  P[i][j] = Pvalue;
 }
}
```

Matrix Multiplication Host Program

```
int main()
{
    int size = 16*16;
    cudaMemcpy(d_M, M, size*sizeof(float),
    cudaMemcpyHostToDevice);
    cudaMemcpy(d_N, N, size*sizeof(float),
    cudaMemcpyHostToDevice);
    dim3 grid(2,2,1);
    dim3 block(8,8,1);
    int N=16; //N is the number of rows and columns
    MatrixMulKernel<<<grid,block>>>(d_M,d_N,d_P,N)
    cudaMemcpy(P, d_P, size*sizeof(float),
    cudaMemcpyDeviceToHost);
}
```

Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int N){
  int i=blockIdx.y*blockDim.y+threadIdx.y;
  int j=blockIdx.x*blockDim.x+threadIdx.x;
  if ((i<N) && (j<N)) {
    float Pvalue = 0.0;
    for (int k = 0; k < N; ++k) {
      Pvalue += d_M[i*N+k]*d_N[k*N+j];
    }
    d_P[i*N+j] = Pvalue;
  }
}
```

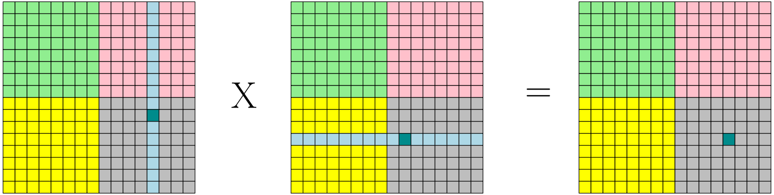


Figure: Matrix Multiplication

$$d\_P[i * N + j] = \sum_{k=0}^{N} d\_M[i * N + k] * d\_N[k * N + j]$$

Slightly Advanced Example: Julia Sets

A Julia Set (named after the French mathematicians Gaston Julia who worked on complex dynamics during the early 20th century.) $\mathcal{J}$ represents a set of points contained in the boundary of a certain class of functions over complex numbers.

- Given a set of points in a complex plane, the set $\mathcal{J}$ is constructed by evaluating for each point, a simple iterative equation given by $Z_n = Z_n^2 + C$ where Z_n represents a complex number and C represents a complex constant.
- A point does not belong to $\mathcal{J}$, if iterative application of the equation yields a diverging sequence of numbers for that point.

Complex Numbers (CPU)

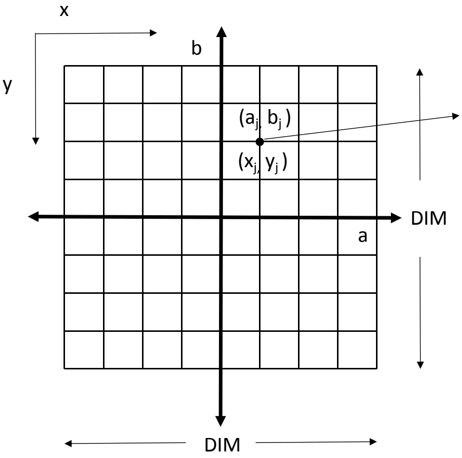
```
struct Complex {
float r;
float i;
};


float magnitude(struct Complex a){
 return ((a.r * a.r) + (a.i * a.i));
}

void add(struct Complex a, struct Complex b, struct Complex *res){
 res->r = a.r + b.r;
 res->i = a.i + b.i;
}

void mul(struct Complex a, struct Complex b, struct Complex *res){
 res->r= (a.r * b.r) - (a.i * b.i);
 res->i= (a.r * b.i) + (a.i * b.r);
}
```

From Pixel Grid to Complex Plane



Translate and Scale

a_j = scale*(x_j-DIM/2)/DIM/2
b_j = scale*(y_j-DIM/2)/DIM/2

Colour (x_j,y_j) depending on membership of $a_j + ib_j$ in Julia Set

Colour (x_j,y_j) red if it belongs to set
Colour (x_j,y_j) black if it does not belong to set.

Figure: Coordinate Transformation

Julia Function for a point (CPU)

```
int julia( int x, int y) {
  const float scale = 1.5;
  float jx = scale * (float)(DIM/2 - x)/(DIM/2);
  float jy = scale * (float)(DIM/2 - y)/(DIM/2);

  struct Complex c,a,r1,r2;
  c.r=-0.8;c.i=0.154;
  a.r=jx; a.i=jy;
  int i = 0;
  for (i=0; i<200; i++) {
  //a = a*a + c;
    mul(a,a,&r1);
    add(r1,c,&r2);
    if (magnitude(r2) > 1000)
      return 0; // return 0 if it is not in set
    a.r = r2.r;
    a.i = r2.i;
  }
  return 1; // return 1 if point is in set
}
```

Driver Code(CPU)

```
void kernel( unsigned char *ptr )
{
  for (int y=0; y<DIM; y++)
  {
    for (int x=0; x<DIM; x++)
    {
      int offset = x + y * DIM;
          int juliaValue = julia(x,y);
          ptr[offset*4 + 0] = 255 * juliaValue;
          ptr[offset*4 + 1] = 0;
          ptr[offset*4 + 2] = 0;
          ptr[offset*4 + 3] = 255;
    }
  }
}
```

A 32 bit per pixel color bitmap represents a 2D grid of pixel values where each pixel is represented by 4 channels (R,G,B,α) and where each channel has values in the range $[0 - 255]$. (α represents transparency).

Driver Code(CPU)

```
int main( void )
{
  CPUBitmap bitmap( DIM, DIM );
  unsigned char *ptr = bitmap.get_ptr();
  kernel(ptr);
  bitmap.display_and_exit();
}
```

We leave out intricate details of how bitmap data is constructed. The primary focus of discussing this application lies in depicting the underlying computation involved in constructing Julia sets.

Complex Numbers GPU

```
struct cuComplex {
float r;
float i;
};

__device__ float magnitude(struct cuComplex a){
  return ((a.r * a.r) + (a.i * a.i));
}

__device__ void add(struct cuComplex a, struct cuComplex b, struct cuComplex *
    res){
  res->r = a.r + b.r;
  res->i = a.i + b.i;
}

__device__ void mul(struct cuComplex a, struct cuComplex b, struct cuComplex *
    res){
  res->r= (a.r * b.r) - (a.i * b.i);
  res->i= (a.r * b.i) + (a.i * b.r);
}
```

Julia Function GPU

```
__device__ int julia( int x, int y) {
  const float scale = 1.5;
  float jx = scale * (float)(DIM/2 - x)/(DIM/2);
  float jy = scale * (float)(DIM/2 - y)/(DIM/2);

  struct cuComplex c,a,r1,r2;
  c.r=-0.8;c.i=0.154;
  a.r=jx; a.i=jy;
  int i = 0;
  for (i=0; i<200; i++) {
  //a = a*a + c;
    mul(a,a,&r1);
    add(r1,c,&r2);
    if (magnitude(r2) > 1000)
      return 0; // return 0 if it is not in set
    a.r = r2.r;
    a.i = r2.i;
  }
  return 1; // return 1 if point is in set
}
```

CUDA Kernel GPU

```
  __global__ void kernel( unsigned char *ptr) {
  // map from threadIdx/BlockIdx to pixel position
  int x = blockIdx.x;
  int y = blockIdx.y;
  int offset = x+y*gridDim.x;


  int juliaValue = julia(x,y);
  ptr[offset*4 + 0] = 255 * juliaValue;  // red if 1 , black if 0
  ptr[offset*4 + 1] = 0;
  ptr[offset*4 + 2] = 0;
  ptr[offset*4 + 3] = 255;
}
```

Host Program

```
int main(void) {
 CPUBitmap bitmap( DIM, DIM );
 unsigned char *dev_bitmap;

 cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) ;
 dim3 grid(DIM,DIM);
 kernel<<<grid,1>>>(dev_bitmap);

 cudaMemcpy(bitmap.get_ptr(),dev_bitmap,bitmap.image_size(),
     cudaMemcpyDeviceToHost);
 bitmap.display_and_exit();
 cudaFree(dev_bitmap);
}
```

Julia Fractal Pattern

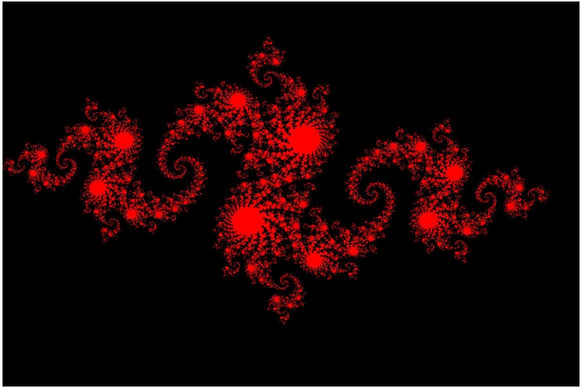


Figure: Julia Set

CUDA by Example: An Introduction to General-Purpose GPU Programming by Sanders et al.

Complex Numbers (CPU) C++

```cpp
struct Complex
{
  float r;
  float i;
  Complex( float a, float b ) : r(a), i(b)
  {}
  float magnitude2( void ) { return r * r + i * i; }
  Complex operator*(const Complex& a)
  {
    return Complex(r*a.r - i*a.i, i*a.r + r*a.i);
  }
  Complex operator+(const Complex& a)
  {
    return Complex(r+a.r, i+a.i);
  }
};
```

Julia Function for a point (CPU) C++

```
int julia( int x, int y )
{
  const float scale = 1.5;
  float jx = scale * (float)(DIM/2 - x)/(DIM/2);
  float jy = scale * (float)(DIM/2 - y)/(DIM/2);
  Complex c(-0.8, 0.156); //constant C
  Complex a(jx, jy);
  int i = 0;
  for (i=0; i<200; i++) {
    a = a * a + c;
    if (a.magnitude2() > 1000)
     return 0;
  }
  return 1;
}
```

Complex Numbers GPU (C++)

```
struct cuComplex
{
  float r;
  float i;
  __device__ cuComplex( float a, float b) : r(a), i(b) {}
  __device__ float magnitude2( void ) {return r * r + i * i;}
  __device__ cuComplex operator*(const cuComplex& a) {
     return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
  }
  __device__ cuComplex operator+(const cuComplex& a) {
     return cuComplex(r+a.r, i+a.i);
  }
};
```

Julia Function GPU (C++)

```
__device__ int julia( int x, int y) {
  const float scale = 1.5;
  float jx = scale * (float)(DIM/2 - x)/(DIM/2);
  float jy = scale * (float)(DIM/2 - y)/(DIM/2);

  cuComplex c(-0.8,0.154);
  cuComplex a(jx,jy);

  int i = 0;
  for (i=0; i<200; i++) {
    a = a*a + c;
    if (a.magnitude2() > 1000)
      return 0; // return 0 if (x,y) is not in set
  }
  return 1; // return 1 if (x,y) is in set
}
```

Pre-requisite to run CUDA on your system

To use CUDA on your system, you will need the following installed:

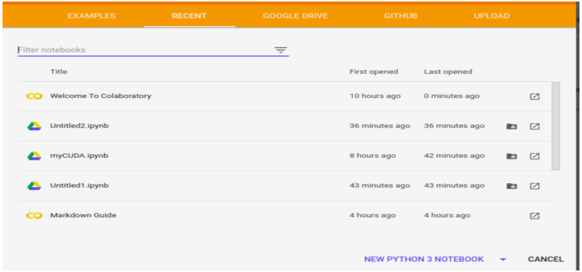
- CUDA-capable GPU
- A supported version of Linux with a gcc compiler and toolchain
- NVIDIA CUDA Toolkit (available at `http://developer.nvidia.com/cuda-downloads`)

Please follow the steps to install CUDA from here.

How to setup Google Colab

Google Colab can compile and execute CUDA code online.

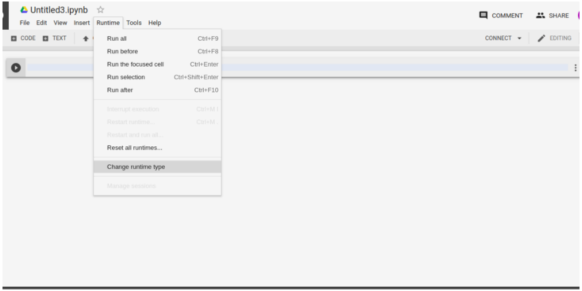
- Open this link in Google Chrome.
- Open NEW PYTHON 3 NOTEBOOK.



Title	First opened	Last opened
Welcome To Colaboratory	10 hours ago	0 minutes ago
Untitled2.ipynb	36 minutes ago	36 minutes ago
myCUDA.ipynb	8 hours ago	42 minutes ago
Untitled1.ipynb	43 minutes ago	43 minutes ago
Markdown Guide	4 hours ago	4 hours ago

Cont.

- Click on Runtime -> Change runtime type.
- Select GPU from the drop down menu and click on Save.



Cont.

- Check your NVCC version using this code in code cell:

```
!nvcc --version
```

- The output should be something like this:

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
Built on Sat_Aug_25_21:08:01_CDT_2018
Cuda compilation tools, release 10.0, V10.0.130
```

Cont.

If NOT then, install CUDA Version 9 following these commands in code cell.

```
!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
```

Cont.

- Execute the given command to install a small extension to run nvcc from Notebook cells.

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

- Load the extension using this code:

```
%load_ext nvcc_plugin
```

- Go to Insert -> Code Cell
- Write %%cu in the first line
- Write the cuda program and execute