

✓ Basic Command for accessing the CUDA Version and GPU Information

Runtime > Change runtime type > Setting the Hardware accelerator to GPU > Save

```
!ls /usr/local/
```

```
bin      cuda      cuda-12.5  etc      include  libexec   man  sbin  src
colab    cuda-12  dist_metrics.pxd  games  lib      LICENSE.md  opt  share
```

```
!which nvcc
```

```
/usr/local/cuda/bin/nvcc
```

```
!nvidia-smi
```

```
Thu Apr 10 12:15:00 2025
```

NVIDIA-SMI 550.54.15 Driver Version: 550.54.15 CUDA Version: 12.4									
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC				
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.			
						MIG M.			
0	Tesla T4	Off	00000000:00:04.0	Off	0				
N/A	40C	P8	9W / 70W	0MiB / 15360MiB	0%	Default			
						N/A			

Processes:							GPU Memory
GPU	GI	CI	PID	Type	Process name		Usage
ID	ID						
No running processes found							

✓ Matrix Multiplication using CUDA C

```
%writefile matrix_mul.cu

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

#define N 16 // You can increase this to 512 or 1024 for bigger matrices

// CUDA Kernel for Matrix Multiplication
__global__ void matrixMulKernel(int *A, int *B, int *C, int width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < width && col < width) {
        int sum = 0;
        for (int k = 0; k < width; k++) {
            sum += A[row * width + k] * B[k * width + col];
        }
        C[row * width + col] = sum;
    }
}

void fillMatrix(int *mat, int width) {
    for (int i = 0; i < width * width; i++) {
        mat[i] = rand() % 10; // fill with random values 0-9
    }
}

void printMatrix(int *mat, int width) {
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < width; j++) {
            printf("%4d ", mat[i * width + j]);
        }
        printf("\n");
    }
}
```

```

    }
}

int main() {
    int size = N * N * sizeof(int);

    // Allocate memory on host
    int *h_A = (int *)malloc(size);
    int *h_B = (int *)malloc(size);
    int *h_C = (int *)malloc(size);

    // Fill host matrices with random values
    fillMatrix(h_A, N);
    fillMatrix(h_B, N);

    // Allocate memory on device
    int *d_A, *d_B, *d_C;
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    // Copy host matrices to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Define grid and block dimensions
    dim3 dimBlock(16, 16);
    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
                 (N + dimBlock.y - 1) / dimBlock.y);

    // Launch CUDA kernel
    matrixMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);


    // Copy result back to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Print results
    printf("Matrix A:\n");
    printMatrix(h_A, N);
    printf("\nMatrix B:\n");
    printMatrix(h_B, N);
    printf("\nMatrix C (A x B):\n");
    printMatrix(h_C, N);

    // Free memory
    free(h_A); free(h_B); free(h_C);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);


    return 0;
}

```

 Writing matrix_mul.cu

```
!nvcc -arch=sm_75 matrix_mul.cu -o matrix_mul
```

```
!./matrix_mul
```

 Matrix A:

3	6	7	5	3	5	6	2	9	1	2	7	0	9	3	6
0	6	2	6	1	8	7	9	2	0	2	3	7	5	9	2
2	8	9	7	3	6	1	2	9	3	1	9	4	7	8	4
5	0	3	6	1	0	6	3	2	0	6	1	5	5	4	7
6	5	6	9	3	7	4	5	2	5	4	7	4	4	3	0
7	8	6	8	8	4	3	1	4	9	2	0	6	8	9	2
6	6	4	9	5	0	4	8	7	1	7	2	7	2	2	6
1	0	6	1	5	9	4	9	0	9	1	7	7	1	1	5
9	7	7	6	7	3	6	5	6	3	9	4	8	1	2	9
3	9	0	8	8	5	0	9	6	3	8	5	6	1	1	5
9	8	4	8	1	0	3	0	4	4	4	4	7	6	3	1
7	5	9	6	2	1	7	8	5	7	4	1	8	5	9	7
5	3	8	8	3	1	8	9	6	4	3	3	3	8	6	0
4	8	8	8	9	7	7	6	4	3	0	3	0	9	2	5
4	0	5	9	4	6	9	2	2	4	7	7	5	4	8	1
2	8	9	3	6	8	0	2	1	0	5	1	1	0	8	5

Matrix B:

0	6	4	6	2	5	8	6	2	8	4	7	2	4	0	6
2	9	9	0	8	1	3	1	1	0	3	4	0	3	9	1

```

9   6   9   3   3   8   0   5   6   6   4   0   0   4   6   2
6   7   5   6   9   8   7   2   8   2   9   9   6   0   2   7
6   1   3   2   1   5   9   9   1   4   9   1   0   7   5   8
7   0   4   8   0   4   2   9   6   1   0   4   2   2   2   0
5   5   2   9   0   2   8   3   8   0   4   0   9   1   9   6
2   5   4   4   9   9   3   6   0   5   0   2   9   4   3   5
1   7   4   3   1   4   6   9   4   2   2   6   4   1   2   8
8   9   2   8   8   8   6   8   3   8   3   3   3   8   0   4
7   6   8   9   0   6   8   7   9   0   3   3   3   7   3   2
6   5   2   6   5   8   7   9   6   0   4   1   0   4   8   7
0   8   6   2   4   7   9   3   9   2   8   3   0   1   7   8
9   1   5   4   9   2   5   7   4   9   9   4   5   9   3   5
7   0   8   1   9   9   7   8   2   5   3   4   9   0   2   0
1   9   6   2   1   2   0   7   3   1   1   9   0   5   6   7

```

Matrix C (A x B):

```

373 374 376 323 307 351 359 466 334 231 305 289 235 273 340 359
325 309 368 296 363 390 368 385 322 191 269 245 316 182 317 283
425 421 454 318 412 465 412 521 369 269 351 323 240 273 364 371
235 299 291 257 216 288 313 302 287 175 249 241 214 190 228 286
384 382 365 377 355 442 414 434 361 249 323 270 243 260 297 334
438 429 454 346 440 469 494 501 349 360 422 340 276 325 312 379
282 464 408 320 325 422 435 416 350 220 332 327 254 256 327 413
333 343 283 336 273 415 323 443 299 225 229 186 187 269 287 317
365 552 490 410 319 488 512 533 433 262 378 367 242 337 415 472
304 444 396 329 341 428 435 461 327 189 313 320 216 287 327 391
285 401 359 290 333 354 407 342 323 240 337 286 190 232 275 329
407 519 495 384 440 529 484 518 399 350 369 352 346 315 371 426
400 392 392 363 405 461 439 450 350 314 351 269 352 274 309 377
432 398 412 359 378 407 403 490 333 297 374 306 275 326 369 393
432 356 374 417 324 467 483 472 427 235 355 265 306 250 317 355
314 270 380 215 233 325 255 364 239 167 207 217 147 200 262 185

```

✓ Addition of two large vectors

```

%%writefile vector_add.cu

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

#define N 1000000 // 1 million elements

// CUDA Kernel to perform vector addition
__global__ void vectorAdd(int *a, int *b, int *c, int n) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}

// Fill array with random integers
void fillArray(int *arr, int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
}

int main() {
    int size = N * sizeof(int);

    // Allocate memory on host
    int *h_a = (int *)malloc(size);
    int *h_b = (int *)malloc(size);
    int *h_c = (int *)malloc(size);

    // Fill vectors with random data
    fillArray(h_a, N);
    fillArray(h_b, N);

    // Allocate memory on device
    int *d_a, *d_b, *d_c;
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Copy vectors to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);

```

```

    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    // Launch the kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N);


    // Copy result back to host
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    // Print first 10 results for verification
    printf("Vector Addition Result (first 10 elements):\n");
    for (int i = 0; i < 10; i++) {
        printf("%d + %d = %d\n", h_a[i], h_b[i], h_c[i]);
    }

    // Free memory
    free(h_a); free(h_b); free(h_c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);


    return 0;
}

```

 Writing vector_add.cu

```
!nvcc -arch=sm_75 vector_add.cu -o vector_add
```

```
!./vector_add
```

 Vector Addition Result (first 10 elements):

```

83 + 89 = 172
86 + 63 = 149
77 + 84 = 161
15 + 93 = 108
93 + 81 = 174
35 + 55 = 90
86 + 6 = 92
92 + 93 = 185
49 + 61 = 110
21 + 50 = 71

```

Start coding or [generate](#) with AI.