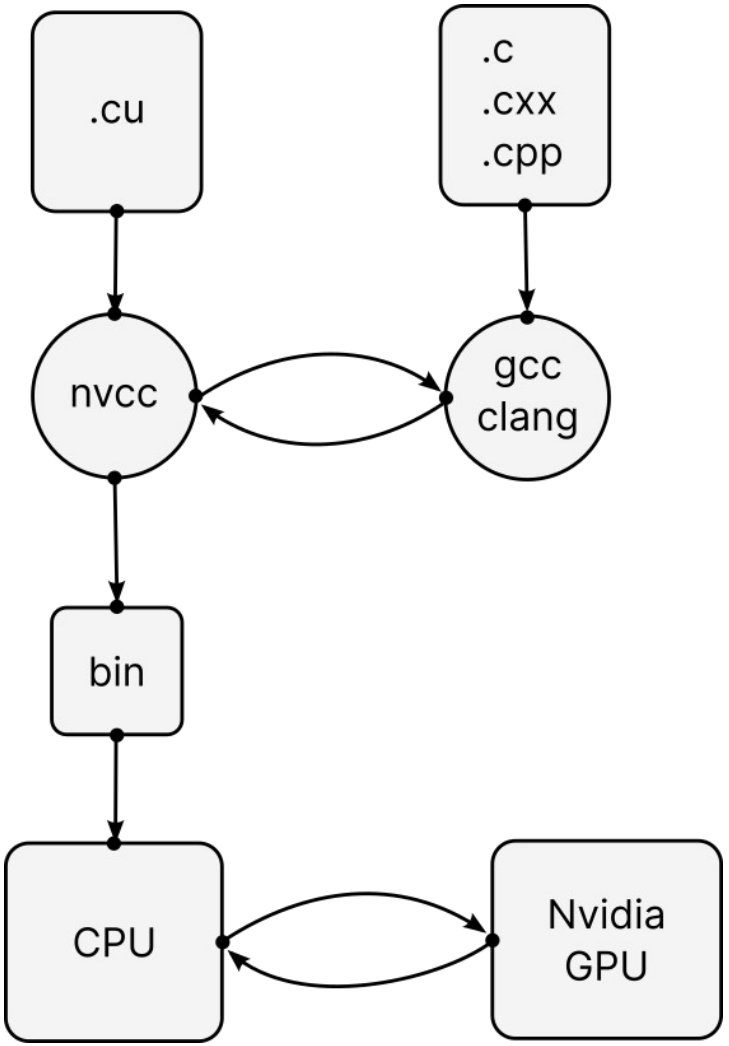
ANNADA BEHERA | Weekly Talks SML

The hitchhiker's guide to the

CUDA C Programming

Rule No. 1: Don't panic !

Rule No. 2: CUDA C is C++ !

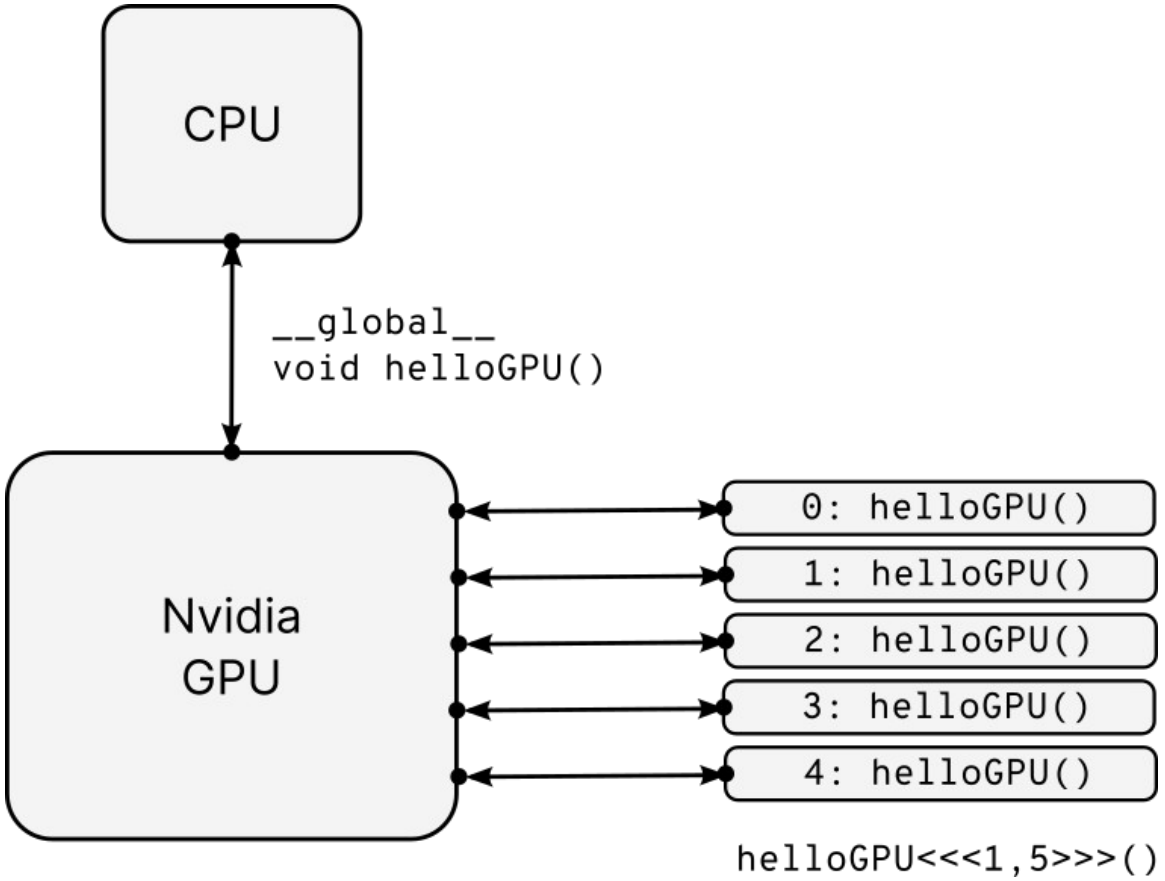


Hello, World !

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

int main(void)
{
        for(size_t i=0; i<5; ++i) {
                fprintf(stdout, "Iteration %d: Hello from CPU!\n");
        }
        return EXIT_SUCCESS;
}
```

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

__global__ void helloGPU(void)
{
        printf("Thread %d: Hello from GPU!\n", threadIdx.x);
}

int main(void)
{
        fprintf(stdout, "Hello World from the CPU.\n");
        helloGPU<<<1, 5>>>();
        cudaDeviceSynchronize();
        return EXIT_SUCCESS;
}
```

Rule No. 3: NVIDIA GPU arch is SIMD !



The device and the host.

- The CPU is called the **host.**
- The GPU is called the **device.**
- Calling a function that runs on the device is called **kernel call.**

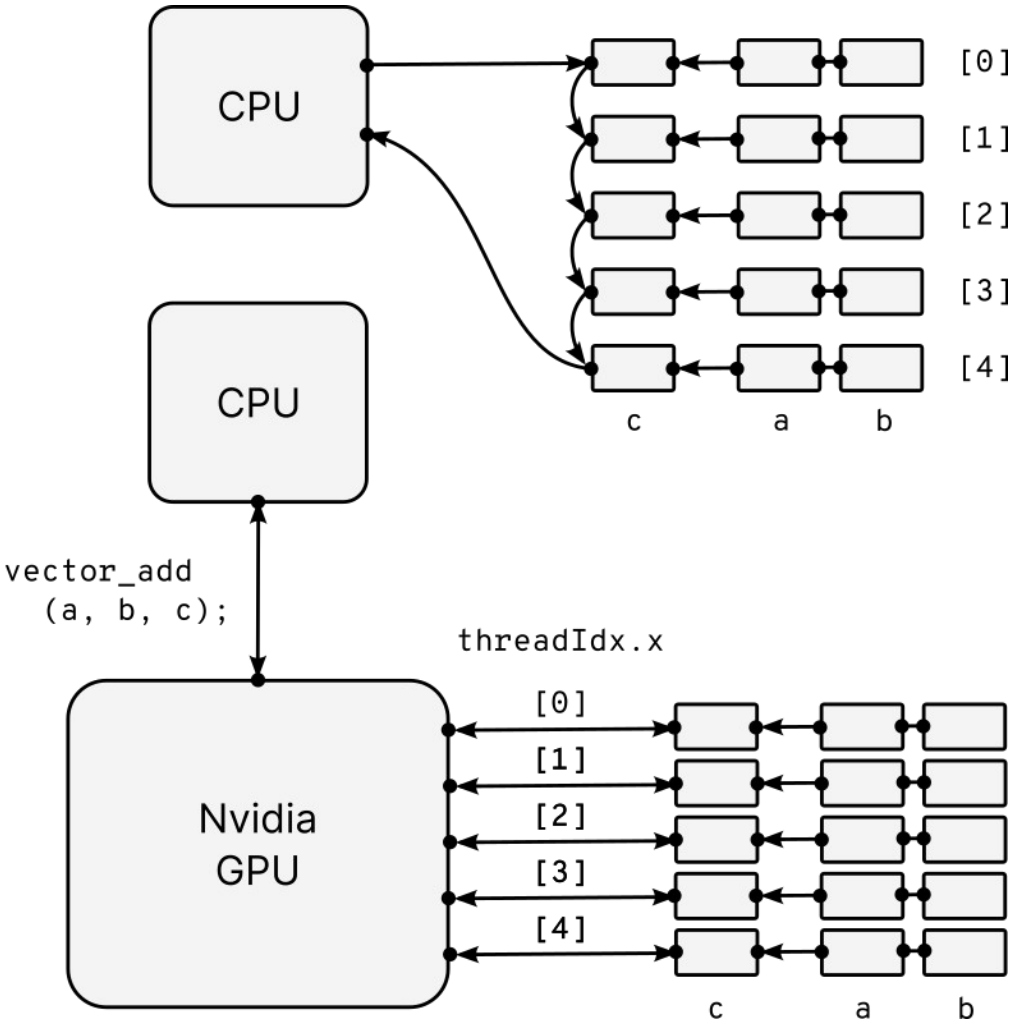
Qualifer	Exec	Callable	Compiler
`__global__`	GPU	host and device	NVCC
`__device__`	GPU	device only	NVCC
`__host__`	CPU	host only	GCC/Clang

Rule No. 4: Do not mix the memories ! Keep ‘em apart.

	host	device
Allocate memory	`malloc`	`cudaMalloc`
Copy memory blocks	`memcpy`	`cudaMemcpy`
Free allocated memory	`free`	`cudaFree`

1. Allocate memory on the device.
2. Optionally copy data from host to device.
3. Compute on the data parallelly on the device with many threads.
4. Copy the results back from the device to host.

A little involved exercise: vector addition.



Vector addition: heap allocation on host.

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

int main(void)
{
    size_t n = 128, N = n * sizeof float*;
    float
        * a = (float *) malloc(N),
        * b = (float *) malloc(N),
        * c = (float *) malloc(N);

    if (a && b && c) {
        fprintf(stderr, “ERROR: Cannot allocate memory.\n”);
        return EXIT_FAILURE;
    }

    get_data(a, b, c);

    return EXIT_SUCCESS;
}
```

Vector addition: memory allocation on the device.

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

int main(void)
{
    size_t n = 128, N = n * sizeof float*;
    float
        *da, *db, *dc;

    da = cudaMalloc(N);
    db = cudaMalloc(N);
    dc = cudaMalloc(N);

    if(da && db && dc)
        return EXIT_FAILURE;

    return EXIT_SUCCESS;
}
```

Vector addition: memory allocation on the device.

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

int main(void)
{
    size_t n = 128, N = n * sizeof float*;
    float
        *da, *db, *dc;

    da = cudaMalloc(N);
    db = cudaMalloc(N);
    dc = cudaMalloc(N);

    if(da && db && dc)
        return EXIT_FAILURE;

    cudaMalloc(&da, N);
    cudaMalloc(&db, N);
    cudaMalloc(&dc, N);

    return EXIT_SUCCESS;
}
```

Vector addition: handling allocation error.

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

int main(void)
{
    size_t n = 128, N = n * sizeof * float;
    float
        *da, *db, *dc;

    cudaError_t ea, eb, ec;

    ea = cudaMalloc(&da, N);
    eb = cudaMalloc(&db, N);
    ec = cudaMalloc(&dc, N);

    if(
        ea == cudaErrorMemoryAllocation ||
        eb == cudaErrorMemoryAllocation ||
        ec == cudaErrorMemoryAllocation  )
            return EXIT_FAILURE;

    return EXIT_SUCCESS;
}
```

Vector addition: copy data from CPU to GPU.

```
int main(void)
{
    size_t n = 128, N = n * sizeof * float;
    float
        * ha = (float *) malloc(N),
        * hb = (float *) malloc(N),
        * hc = (float *) malloc(N);
        * da, *db, *dc;
    cudaError_t ea, eb, ec;

    ea = cudaMalloc(&da, N);
    eb = cudaMalloc(&db, N);
    ec = cudaMalloc(&dc, N);

    if(
        ea == cudaErrorMemoryAllocation ||
        eb == cudaErrorMemoryAllocation ||
        ec == cudaErrorMemoryAllocation  )
            return EXIT_FAILURE;

    cudaMemcpy(da, hc, N, cudaMemcpyHostToDevice);
    cudaMemcpy(db, hc, N, cudaMemcpyHostToDevice);

    return EXIT_SUCCESS;
}
```

Vector addition: adding vectors.

```
__global__ void add(
    float * res,
    float * a,
    float * b
){
        size_t i = threadIdx.x;
        res[i] = a[i] + b[i];
}

int main(void)
{
        size_t n=128, N = n * sizeof float*;

        /*  allocation from the previous slides here */

        add<<<1, n>>>(dc, da, db);

}
```

Vector addition: copy data back to the host and free.

```
int main(void)
{

    size_t n = 128, N = n * sizeof * float;

    /* from previous slides */

    cudaMemcpy(da, hc, N, cudaMemcpyHostToDevice);
    cudaMemcpy(db, hc, N, cudaMemcpyHostToDevice);

    add<<<1, n>>>(dc, da, dc);

    cudaMemcpy(hc, dc, N, cudaMemcpyDeviceToHost);

    cudaFree(da);
    cudaFree(db);
    cudaFree(dc);

    free(ha);
    free(hb);
    free(hc);

    return EXIT_SUCCESS;
}
```

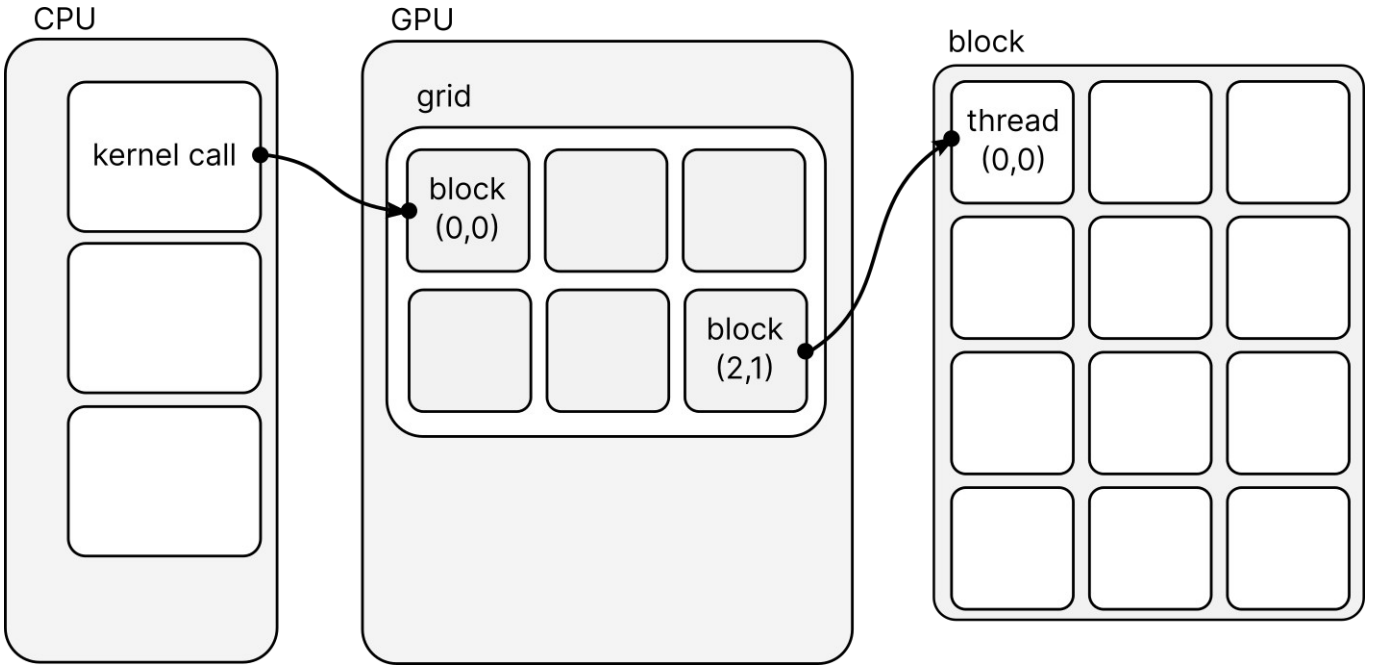
Vector addition: full code.

```
#include<stdlib.h>

__global__ void add(float *res, float *a, float
*b)
{
    size_t i = threadIdx.x;
    res[i] = a[i] + b[ti];
}

int main(void)
{
    size_t n = 1024, N = n * sizeof float *;
    float
        * ha = (float *) malloc(sizeof * ha * n),
        * hb = (float *) malloc(sizeof * hc * n),
        * hc = (float *) malloc(sizeof * hc * n),
        * da, * db, * dc;
    cudaMalloc((float**)&da, N);
    cudaMalloc((float**)&db, N);
    cudaMalloc((float**)&dc, N);
    cudaMemcpy(dA, hA, N, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, hB, N, cudaMemcpyHostToDevice);
    add<<<1, n>>>(dc, da, db);
    cudaMemcpy(hc, dc, N, cudaMemcpyDeviceToHost);

    cudaFree(da);
    cudaFree(db);
    cudaFree(dc);
    free(ha);
    free(hb);
    free(hc);
    return EXIT_SUCCESS;
}
```

Output:

crickets

Rule No. 5: Memory is organized in grid of blocks of threads.




```
cd /usr/local/lib/cuda/extras/demo_suite
./deviceQuery
```

For my GeForce RTX 3090,

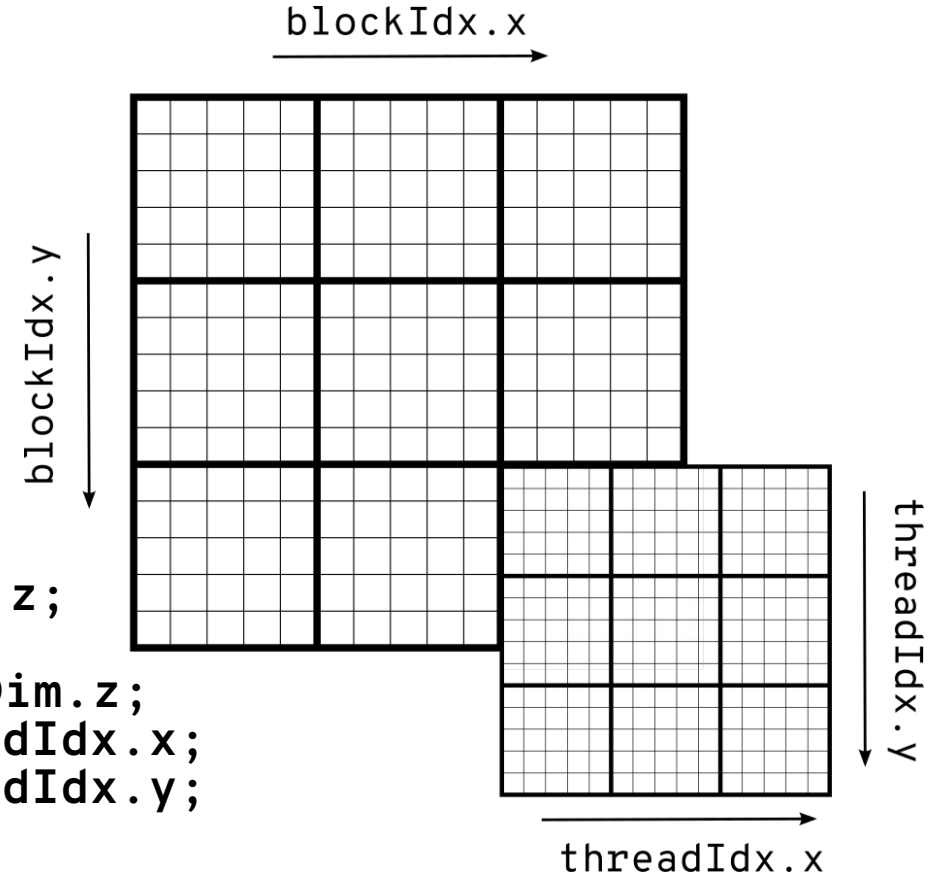
```
Maximum number of threads per multiprocessor:  1536
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

Rule No. 6: CUDA uses row-major order like C.

Data structures
- `struct uint3 { x, y, z; }`
- `struct dim3  { x, y, z; }`

Built-in Variables
- `unit3 threadIdx, blockIdx;`
- `dim3  gridDim, blockDim;`

Calculations
- `total_blocks = gridDim.x * gridDim.y * gridDim.z;`
- `total_threads = blockDim.x * blockDim.y * blockDim.z;`
- `x = blockIdx.x * blockDim.x + threadIdx.x;`
- `y = blockIdx.y * blockDim.y + threadIdx.y;`



Example
```
dim3 grid(16, 16);
dim3 block(32, 32);
gpu_func<<<grid, block>>>(args*);
```

Why
- Block-local synchronization
- Block-local shared memory

Vector addition: adding vectors.

```
#define N 512
#define BLOCK_DIM 512
__global__ void matrixAdd (int *a, int *b, int *c);
    int main(void) {
    int a[N][N], b[N][N], c[N][N];
    int *dev_a, *dev_b, *dev_c;
    int size = N * N * sizeof(int);
    // initialize a and b with real values (NOT SHOWN)
    cudaMalloc((void**)&dev_a, size);
    cudaMalloc((void**)&dev_b, size);
    cudaMalloc((void**)&dev_c, size);
    cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

    dim3 dimBlock(BLOCK_DIM, BLOCK_DIM);
    dim3 dimGrid((int)ceil(N/dimBlock.x),(int)ceil(N/dimBlock.y));
    matrixAdd<<<dimGrid,dimBlock>>>(dev_a,dev_b,dev_c);
    cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);
    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}
__global__ void matrixAdd (int *a, int *b, int *c) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int index = col + row * N;
    if (col < N && row < N) {
        c[index] = a[index] + b[index];
    }
}
```

Referrence and further reading

- https://docs.nvidia.com/cuda/cuda-runtime-api/