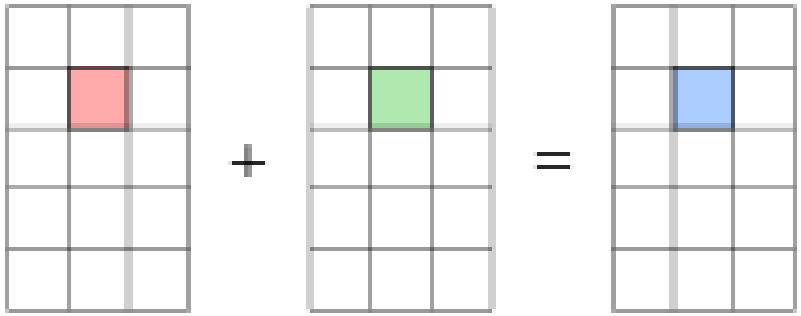
CUDA Programming

Addition of 2 Matrices

Matrix Addition

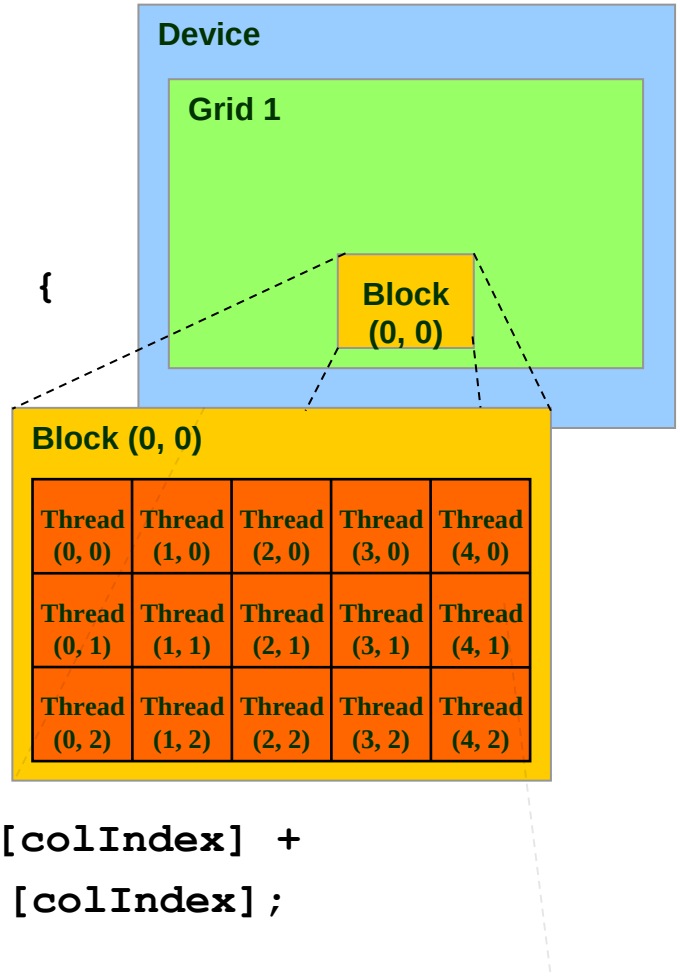


```
c[i][j] = a[i][j] + b[i][j];
```

Matrix Addition on the Device: `add()`

- Parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        int rowIndex, colIndex;

        rowIndex = threadIdx.y;
        colIndex = threadIdx.x;




        c[rowIndex][colIndex] = a[rowIndex][colIndex] +
                                b[rowIndex][colIndex];
}
```



Running the kernel with 1 Block of N * N threads.

Matrix Addition on the Device: `main()`

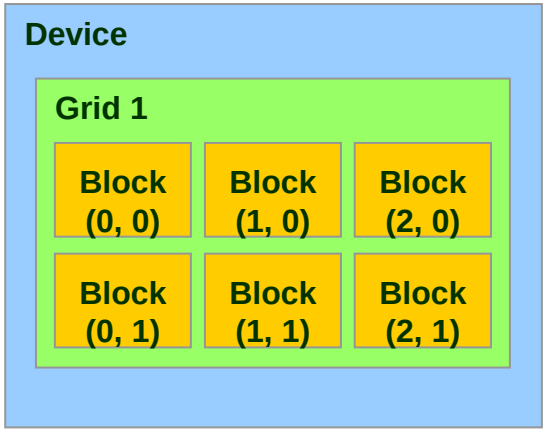
```
#define N 1024
int main(void) {
    int *a  *b  *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int nb = N * N;
    int size = nb * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, nb);
    b = (int *)malloc(size); random_ints(b, nb);
    c = (int *)malloc(size);
```

Matrix Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
   dim3 block(N, N);

    // Launch add() kernel on GPU with N blocks
    add<<<1, block>>>(d_a, d_b, d_c);

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Matrix Addition on the Device: `add()`

- Parallelized `add()` kernel




```
__global__ void add(int *a, int *b, int *c) {
        int rowIndex, colIndex;

        rowIndex = blockIdx.y;
        colIndex = blockIdx.x;

        c[rowIndex][colIndex] = a[rowIndex][colIndex] +
                                b[rowIndex][colIndex];

}
```

Running the kernel with N * N Blocks with 1 thread each.

Matrix Addition on the Device: `main()`

```
#define N 1024
int main(void) {
    int *a  *b  *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int nb = N * N;
    int size = nb * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, nb);
    b = (int *)malloc(size); random_ints(b, nb);
    c = (int *)malloc(size);
```

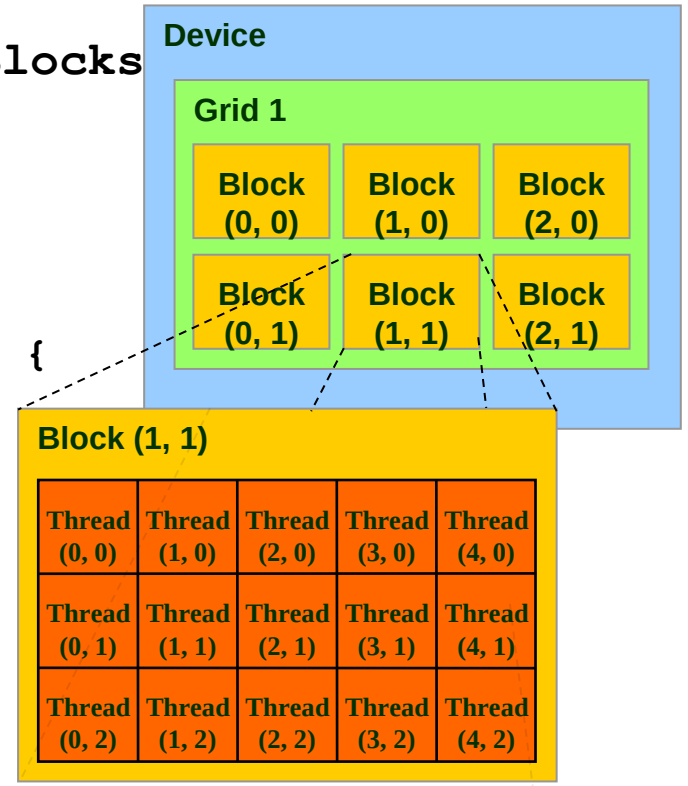
Matrix Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
   dim3 grid(N, N);

    // Launch add() kernel on GPU with N blocks
    add<<<grid, 1>>>(d_a, d_b, d_c);

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Matrix Addition on the Device: `add()`

Running the kernel with (N * N) grid Blocks with (N * N) threads each.

- Parallelized `add()` kernel




```
__global__ void add(int *a, int *b, int *c) {
        int rowIndex, colIndex;

        rowIndex = blockIdx.y * blockDim.y + threadIdx.y;
        colIndex = blockIdx.x * blockDim.x + threadIdx.x;

        c[rowIndex][colIndex] = a[rowIndex][colIndex] +
                                b[rowIndex][colIndex];
}
```

Matrix Addition on the Device: `main()`

```
#define N 1024
int main(void) {
    int *a  *b  *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int nb = N * N * N * N;
    int size = nb * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, nb);
    b = (int *)malloc(size); random_ints(b, nb);
    c = (int *)malloc(size);
```

Matrix Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
   dim3 grid(N, N);
   dim3 block(N, N);

    // Launch add() kernel on GPU with N blocks
    add<<<grid, block>>>(d_a, d_b, d_c);

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Matrix Addition on the Device: `add()`

- Parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c, int width) {
        int rowIndex, colIndex, width;

        rowIndex = (blockIdx.y * blockDim.y + threadIdx.y) * width;
        colIndex = (blockIdx.x * blockDim.x + threadIdx.x) * width;

        for (int i= rowIndex; i < rowIndex  + width; i++)
                for (int j= colIndex; j < colIndex  + width; j++)
                        c[i][j] = a[i][j] + b[i][j];

}
```

Matrix Addition on the Device: main()

```
#define N 16
int main(void) {
    int *a  *b  *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c, *w;       // device copies of a, b, c
    int width = 16;
    int nb = N * N * N * N * width * width;
    int size = nb * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, nb);
    b = (int *)malloc(size); random_ints(b, nb);
    c = (int *)malloc(size);
```

Matrix Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
   dim3 grid(N, N);
   dim3 block(N, N);

    // Launch add() kernel on GPU with N blocks
    add<<<grid,block>>>(d_a, d_b, d_c, width);

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

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```