CHAPTER 2

Heterogeneous data parallel computing

FIGURE 2.1

Conversion of a color image to a grayscale image.

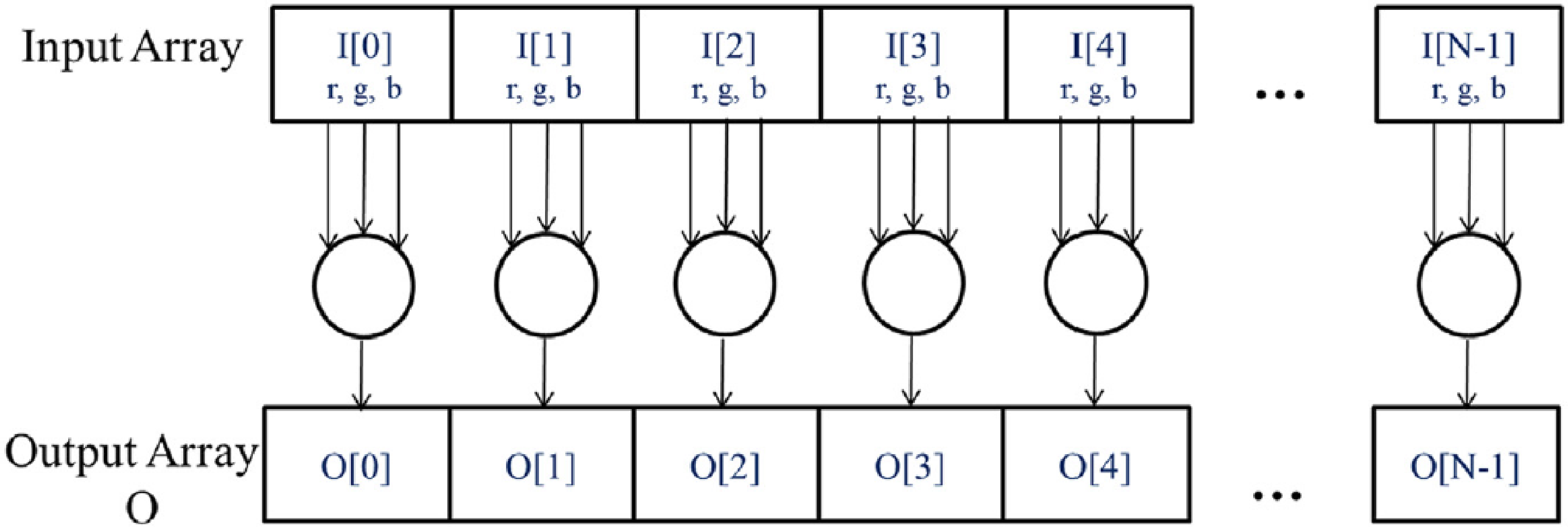


FIGURE 2.2

Data parallelism in image-to-grayscale conversion. Pixels can be calculated independently of each other.

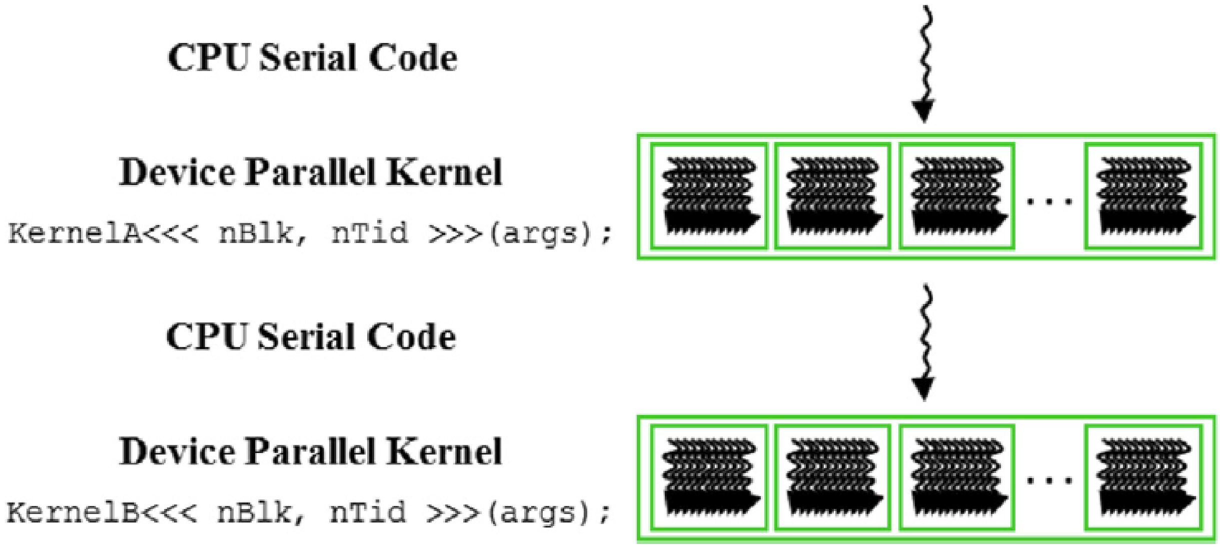


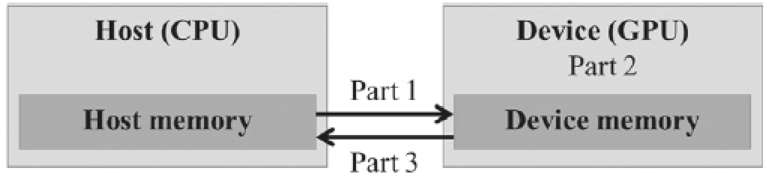
FIGURE 2.3

Execution of a CUDA program.

```c
// Compute vector sum C_h = A_h + B_h
void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
    for (int i = 0; i < n; ++i) {
        C_h[i] = A_h[i] + B_h[i];
    }
}
int main() {
    // Memory allocation for arrays A, B, and C
    // I/O to read A and B, N elements each
    ...
    vecAdd(A, B, C, N);
}
```

FIGURE 2.4

A simple traditional vector addition C code example.




```
void vecAdd(float* A, float* B, float* C, int n) {
    int  size = n* sizeof(float);
    float  *d_A *d_B, *d_C;

    // Part 1: Allocate device memory for A, B, and C
    // Copy A and B to device memory
    ...

    // Part 2: Call kernel – to launch a grid of threads
    // to perform the actual vector addition
    ...

    // Part 3: Copy C from the device memory
    // Free device vectors
    ...
}
```

FIGURE 2.5

Outline of a revised vecAdd function that moves the work to a device.

cudaMalloc()

- Allocates object in the device global memory
- Two parameters
  - **Address of a pointer** to the allocated object
  - **Size** of allocated object in terms of bytes

cudaFree()

- Frees object from device global memory
  - **Pointer** to freed object

FIGURE 2.6

CUDA API functions for managing device global memory.

cudaMemcpy()

- memory data transfer
- Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer

FIGURE 2.7

CUDA API function for data transfer between host and device.

```
void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

FIGURE 2.8

A more complete version of vecAdd().

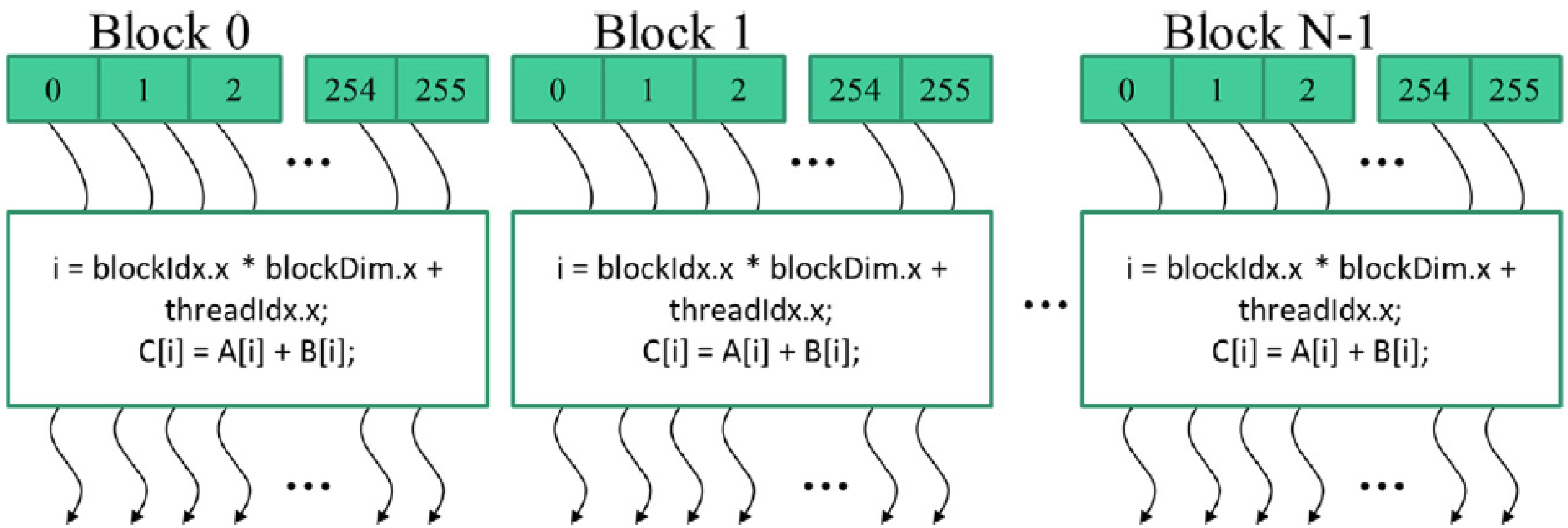


FIGURE 2.9

All threads in a grid execute the same kernel code.

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(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];
    }
}
```

FIGURE 2.10

A vector addition kernel function.

Qualifier Keyword	Callable From	Executed On	Executed By
__host__ (default)	Host	Host	Caller host thread
__global__	Host (or Device)	Device	New grid of device threads
__device__	Device	Device	Caller device thread

FIGURE 2.11

CUDA C keywords for function declaration.

```
int vectAdd(float* A, float* B, float* C, int n) {
    // A_d, B_d, C_d allocations and copies omitted
    ...
    // Launch ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

FIGURE 2.12

A vector addition kernel call statement.

```
void vecAdd(float* A, float* B, float* C, int n) {
    float *A_d, *B_d, *C_d;
    int size = n * sizeof(float);

    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);

    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

FIGURE 2.13

A complete version of the host code in the vecAdd function.

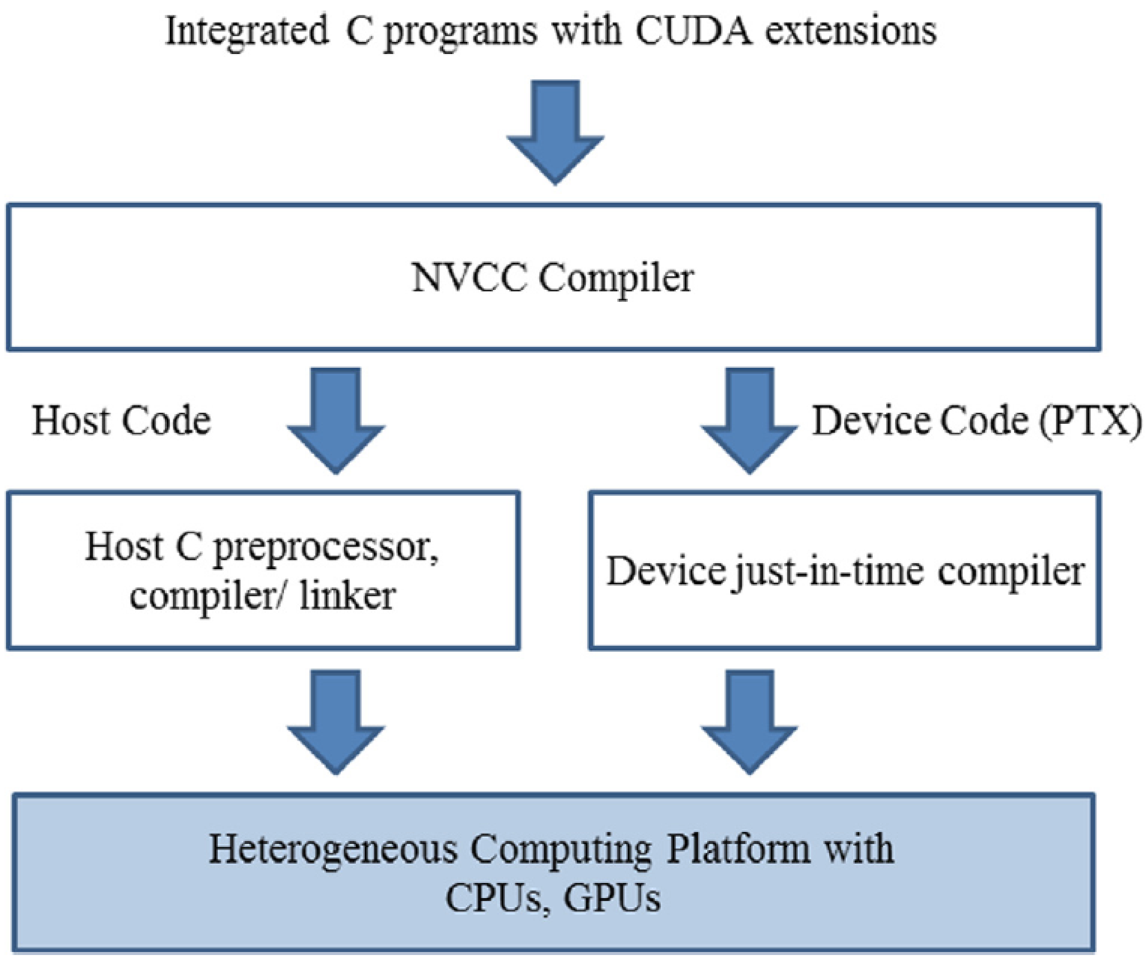
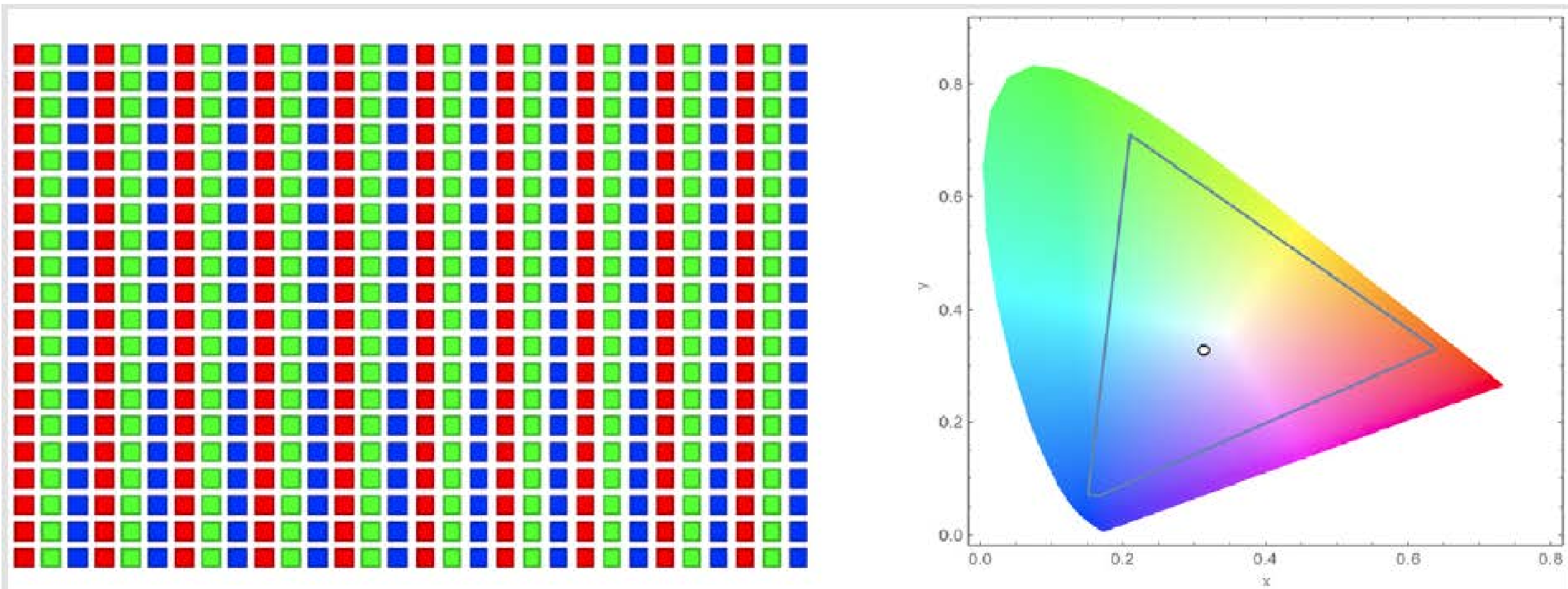


FIGURE 2.14

Overview of the compilation process of a CUDA C program.



In-text figure 1