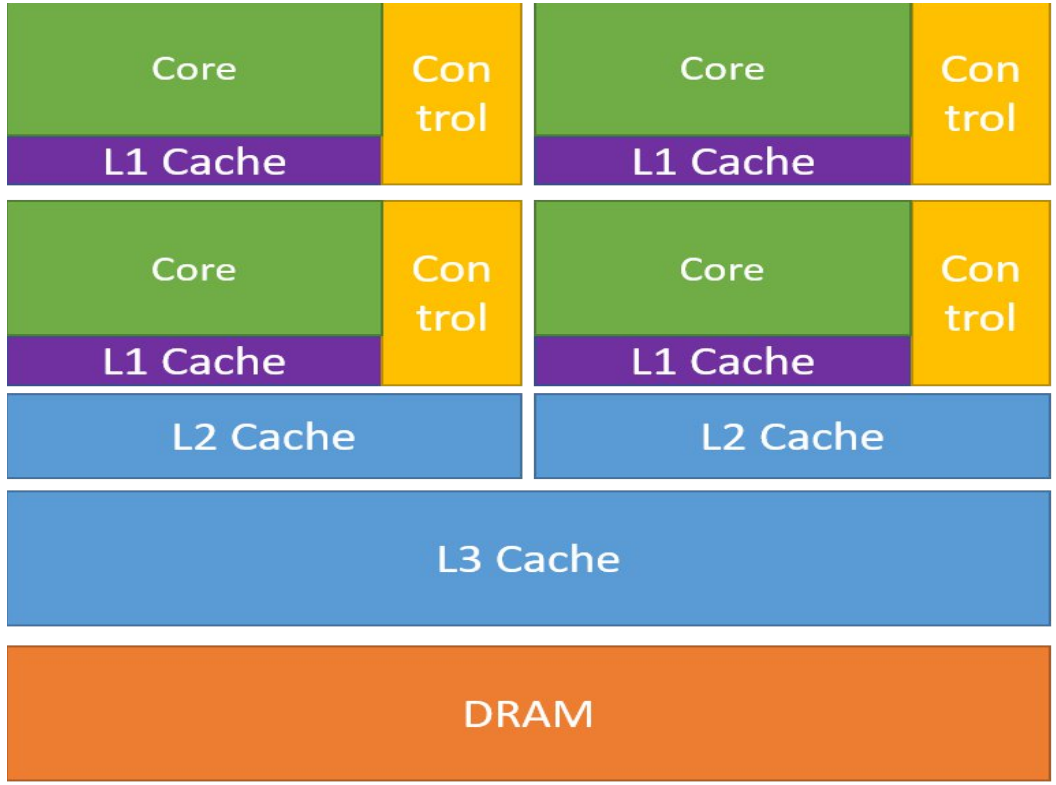
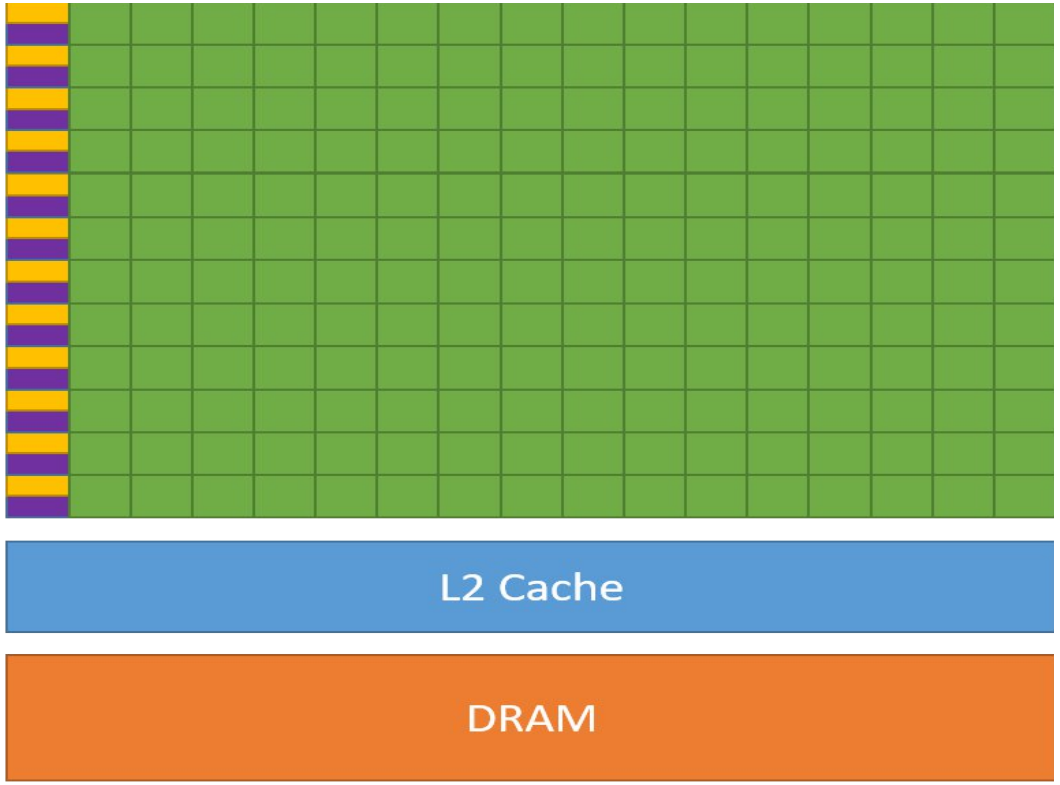
CUDA

Архитектура CUDA



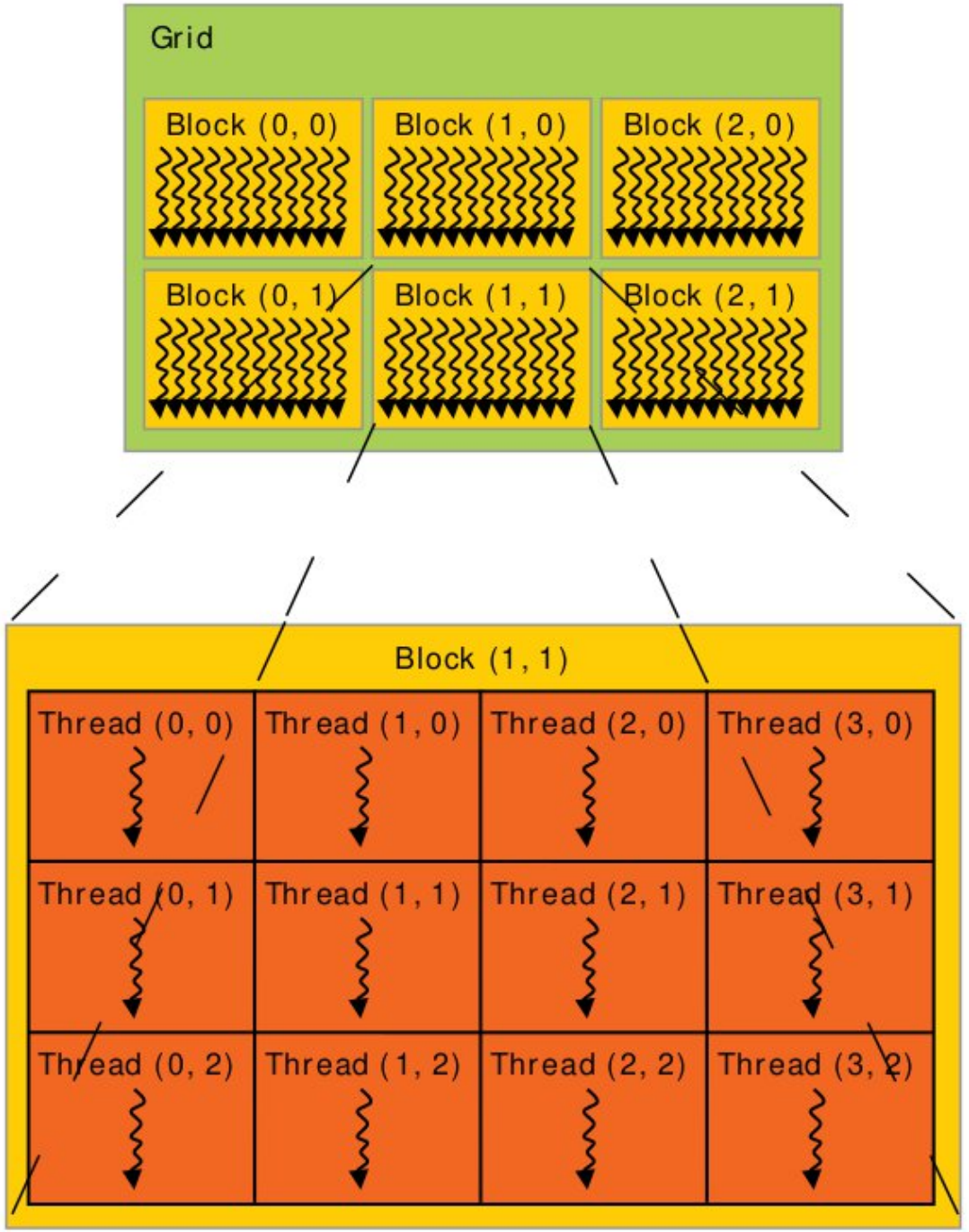
CPU



GPU

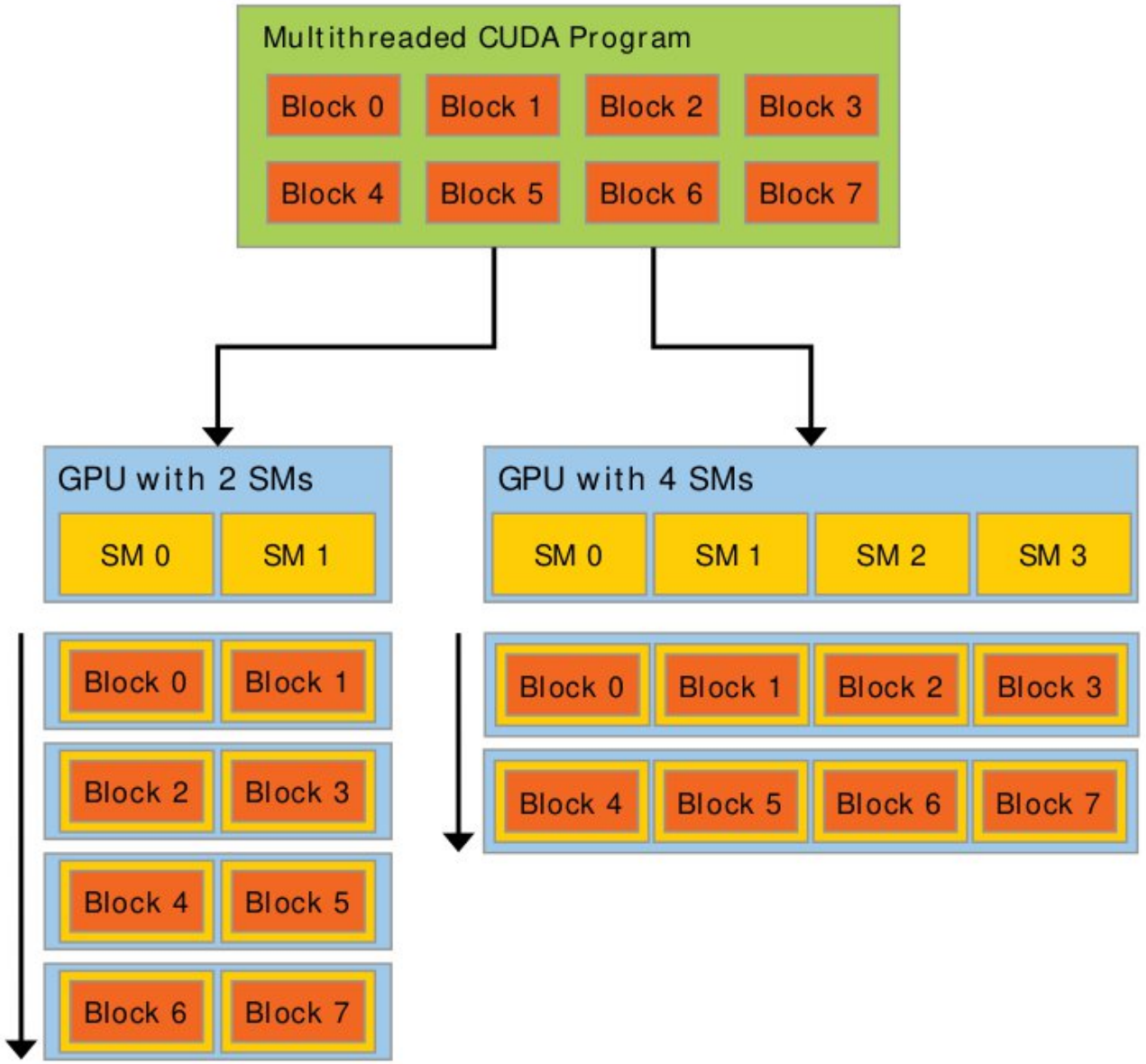
Архитектура CUDA

- Потоки объединены в блоки.
- Каждый блок контролируется SM (Streaming Multiprocessor)
- Блоки объединены в общий Grid



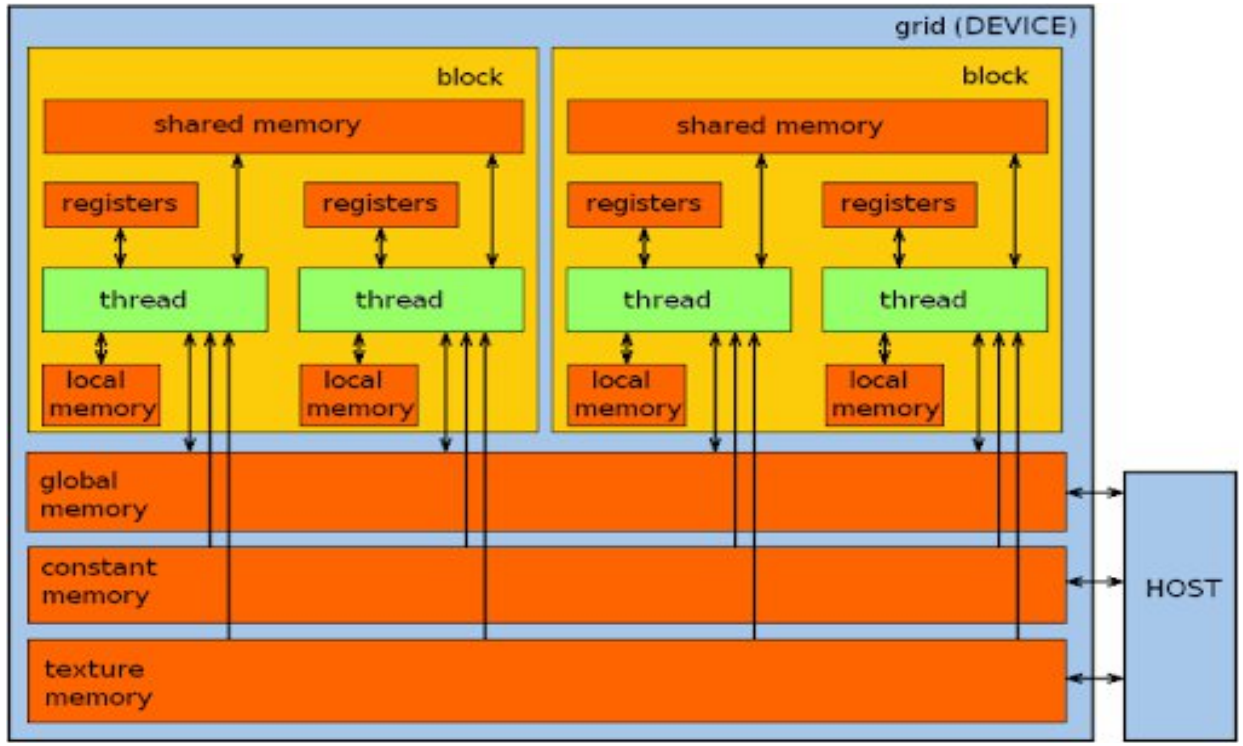
Архитектура CUDA

- SM управляет wrap'ами по 32 потока в каждом
- Так как flow потоков в wrap контролирует один SM, необходимо, что бы потоки выполняли одинаковые действия. Иначе, те потоки, чьи действия отличаются, будут преостановлены.



Память в CUDA

- Каждый поток обладает локальной памятью.
- Каждый блок обладает общей памятью, доступ к которой имеют все потоки в блоке.
- Так же существует общая память GPU, доступ к которой имеет HOST.



Kernel

- Kernel - функция для всех потоков.
- Объявляется с помощью __global__.
- Вызов функции должен сопровождаться execution configuration <<<GridSize,BlockSize>>>
- Параметры для конфигурации имеют тип dim3.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

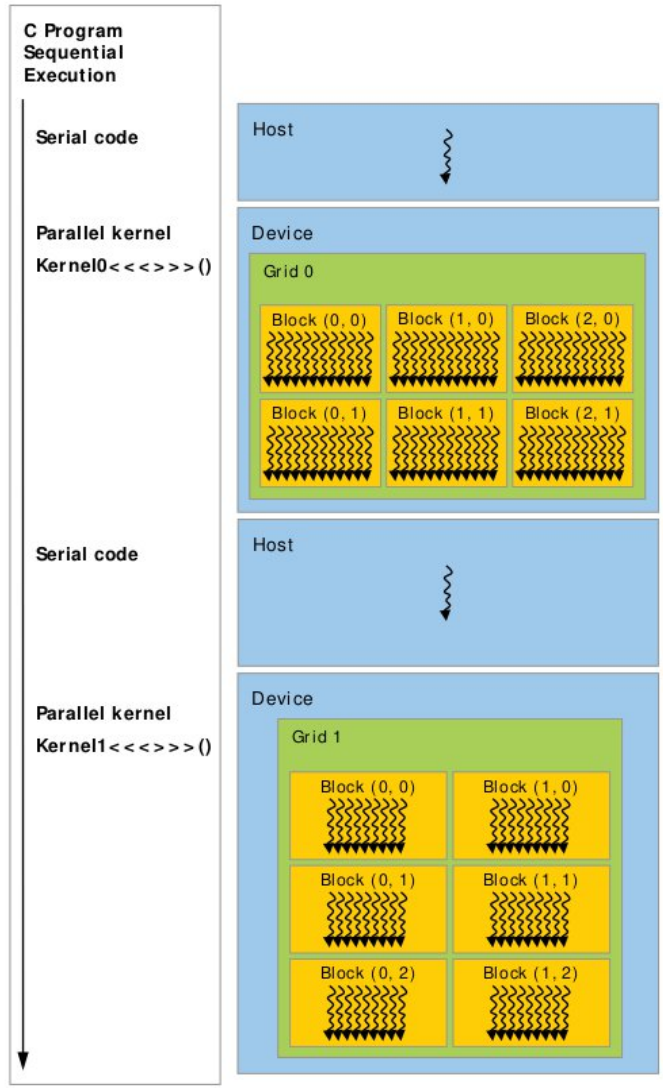
Execution configuration

- Блоки и Grid могут быть одномерными, двухмерными и трехмерными.
- threadIdx - dim3 индекс потока в блоке
- blockIdx - dim3 индекс блока
- blockDim - dim3 размер блока.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Пример работы программы

Kernel можно вызывать несколько раз в программе.



Работа с памятью

- cudaMalloc (void** devPtr, size_t size)
- cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)
- cudaMemcpyKind:
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
- cudaFree (void* devPtr)

Shared memory

- __shared__ - модификатор, который необходимо указать при объявлении массива в функциях с модификатором __global__ и __device__. В таком случае массив будет объявлен в общей памяти блока.

Matrix Mul no shared memory

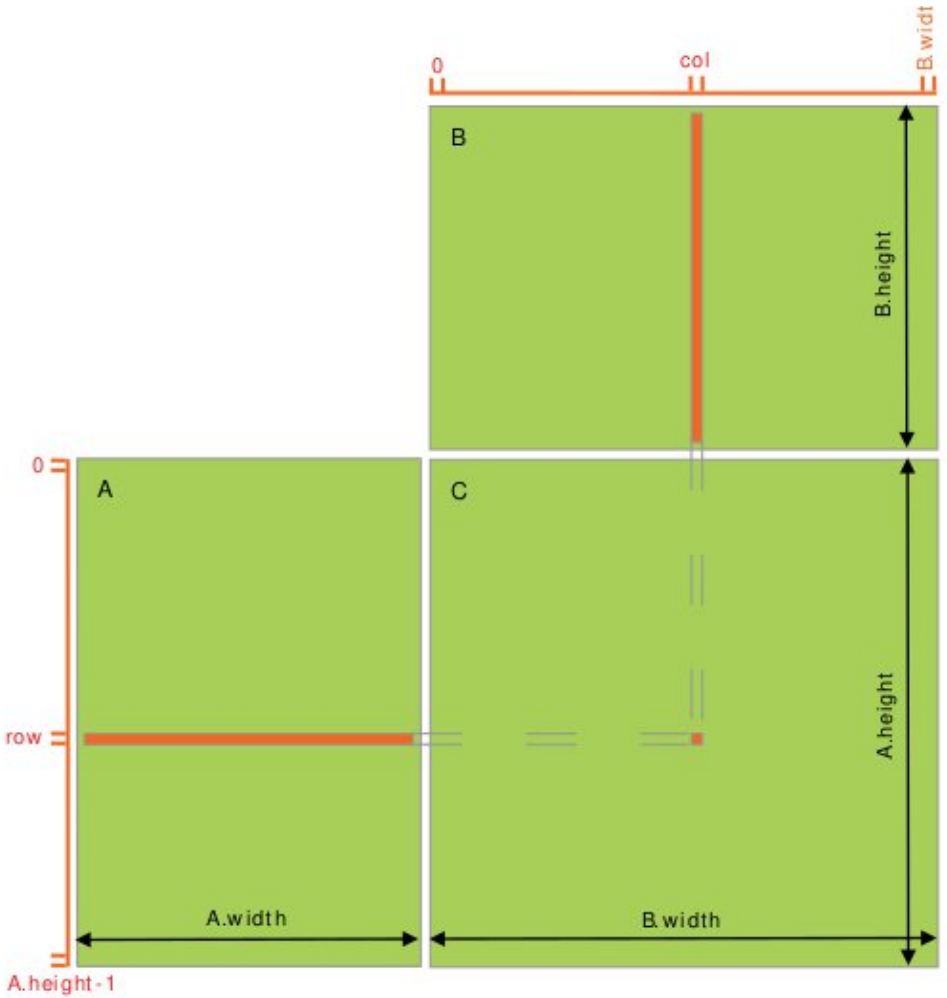


Figure 9 Matrix Multiplication without Shared Memory

Matrix Mul shared memory

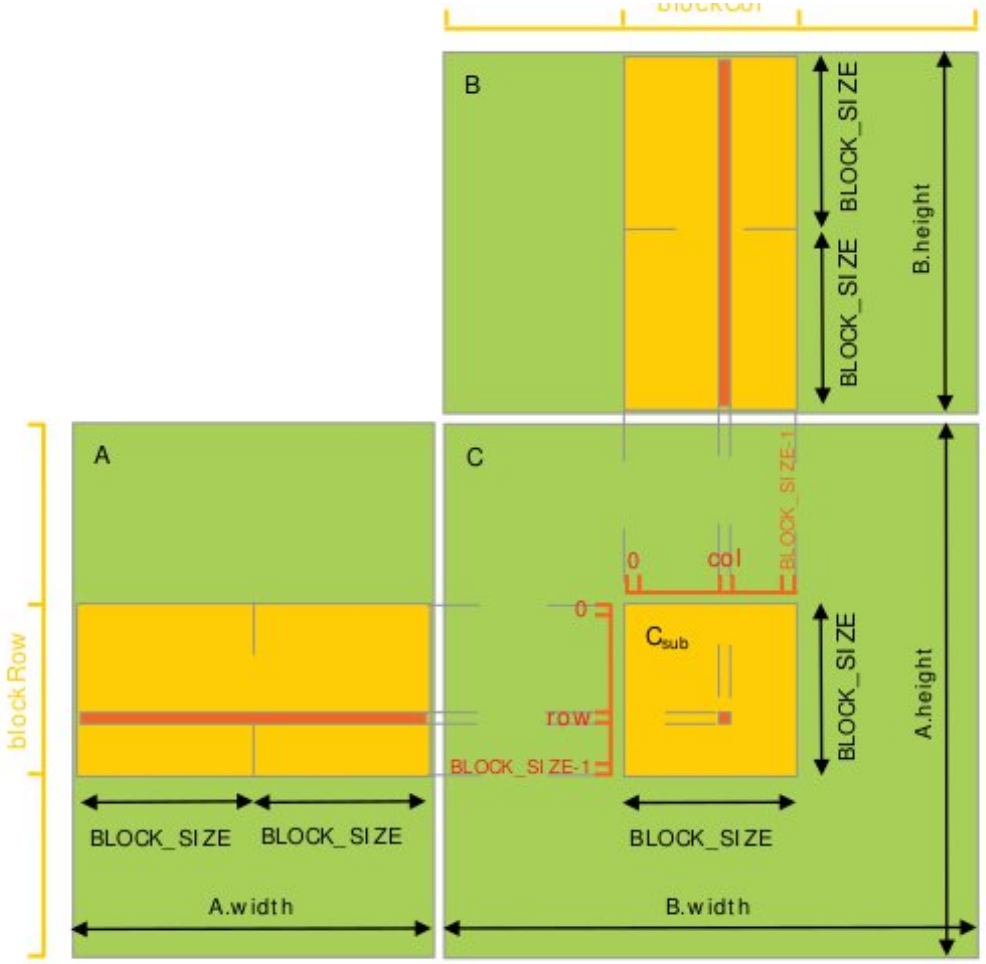


Figure 10 Matrix Multiplication with Shared Memory

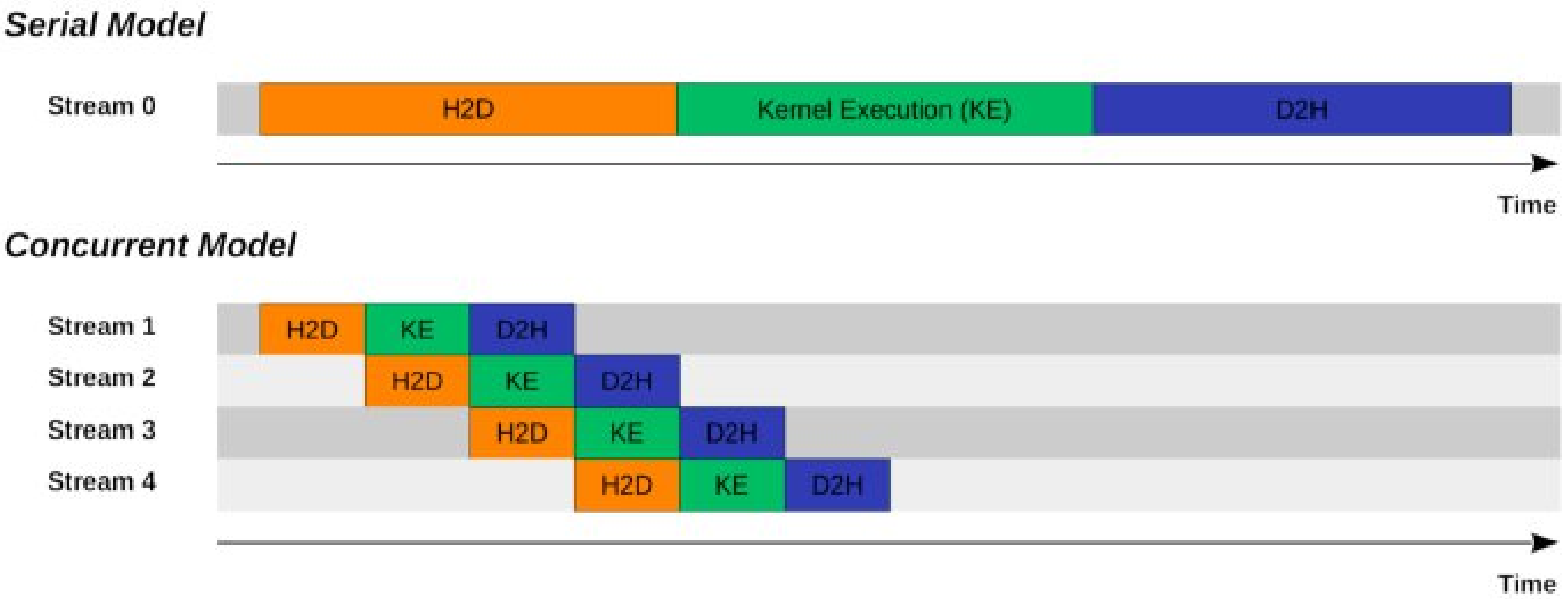
CUDA Event

- cudaEventCreate(event);
- cudaEventRecord(event, stream);
- cudaEventSynchronize(event);
- cudaEventElapsedTime (float* ms, cudaEvent_t start, cudaEvent_t end);
- cudaEventDestroy(event)

CUDA Events

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

cudaEventRecord(start);
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
cudaEventRecord(stop);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

CUDA Streams



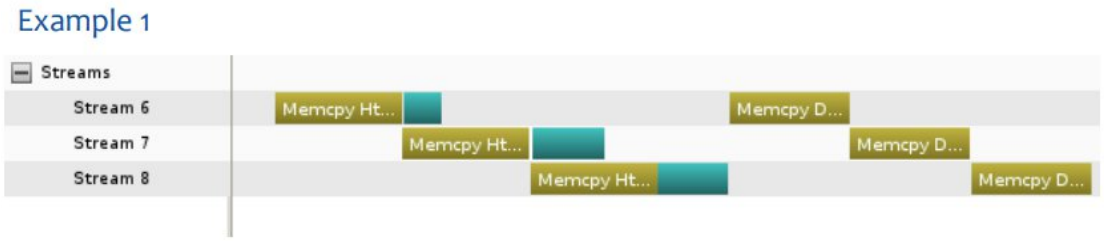
CUDA Stream: Serial Model vs Concurrent Model

CUDA Streams

- cudaStreamCreate (cudaStream_t*) ;
- cudaMemcpyAsync (dev1, host1, size, kind, stream);
- cudaDeviceSynchronize();

Example 1

```
for (int i = 0; i < 3; ++i)
      cudaMemcpyAsync(inputDevPtr + i * size,
                      hostPtr + i * size, size,
                      cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < 3; ++i)
      MyKernel<<<size/512, 512, 0, stream[i]>>>
                    (outputDevPtr + i * size,
                     inputDevPtr + i * size, size);

for (int i = 0; i < 3; ++i)
      cudaMemcpyAsync(hostPtr + i * size,
                      outputDevPtr + i * size, size,
                      cudaMemcpyDeviceToHost, stream[i]);
```



Example 2

```
for(int i=0; i < 3; ++i) {
      cudaMemcpyAsync(inputDevPtr + i * size,
                      hostPtr + i * size, size,
            cudaMemcpyHostToDevice, stream[i]);

      MyKernel<<<size/512, 512, 0, stream[i]>>>
            (outputDevPtr + i * size,
             inputDevPtr + i * size, size);

      cudaMemcpyAsync(hostPtr + i * size,
                 outputDevPtr + i * size, size,
               cudaMemcpyDeviceToHost, stream[i]);
}
```