Compiling CUDA programs

Any program with CUDA programming should be named: xxxx**.cu**
This file can contain both HOST and DEVICE code.

The compiler is **nvcc,** which can handle both C++ and CUDA code. It is capable of compiling all C++ commands for the HOST and most for the DEVICE.

nvcc xxxx.cu -> a.out

nvcc xxxx.cu -o xxxxx yields xxxxx as executable file

Compute capability: newer cards have better capabilities

nvcc -code=sm_30 xxxx.cu means compute_capability of 3.0
nvcc -arch compute_30 xxxx,cu

nvcc -m32 xxxx,cu means compile 32bit code, default is 64 bit (-m64)

Threads

Consider a kernel call:

kernel <<< num_Blocks, threads_per_block>>> (args…)

mat_mult <<< 1, N>> (…)

This means invoke kernel with 1 block of N threads. All of data fits within N.
N can be as big as 1024 on MBP

mat_mult <<<N, 1 >>> (…)   use N blocks of 1 thread.
N can be as large as 65,535 (or larger 2^{31}-1)

(Number of threads per block should be some multiple of 32)

Kernel Calls

dim3 Grid,Block;

// define Grid, Block   Note unspecified dim3 field initializes to 1.
..

kernel <<< Grid, Block>>> (….)

Grid: dimension and size of grid (of blocks)

In two-dimensions: x, y
Number of blocks: Grid.x * Grid.y

Block: dimension and size of blocks of threads
In three-dimensions: x, y, z
Threads per block: Block.x * Block.y* Block.z

Global/Device Automatic Variables

dim3 gridDim;
  Dimension of the grid in blocks; GridDim.x, GridDim.y, GridDim.z

dim3 blockDim;
  Dimensions of the block in number of threads

dim3 blockIdx;
  Block index within grid (starting with 0)

dim3 threadIdx;
  Thread index within block

Note: dim3 dimension not specified is initialized to 1

Threads on GPU

Threads are organized in **blocks**; blocks are grouped into a **grid**; and threads are executed in kernel as a grid of blocks of threads; all computing the same function.

Each block is a 3D array of threads defined by the dimensions: Dx, Dy, and Dz, which you specify.

Each CUDA card has a maximum number of threads in a block (512, 1024, or 2048).

Each thread has a **thread index, threadIdx: (x,y, z)**;

$0 \le x < Dx$, $0 \le y < Dy$, $0 \le z < Dz$, where Dx, Dy, Dz are the block dimensions;

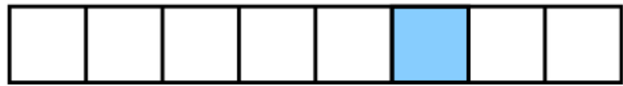
$Dx * Dy * Dz$ = max threads per block

Each thread also has a **thread id: threadId** = $x + y\,Dx + z\,Dx\,Dy$
The threadId is like 1D representation of an array in memory.

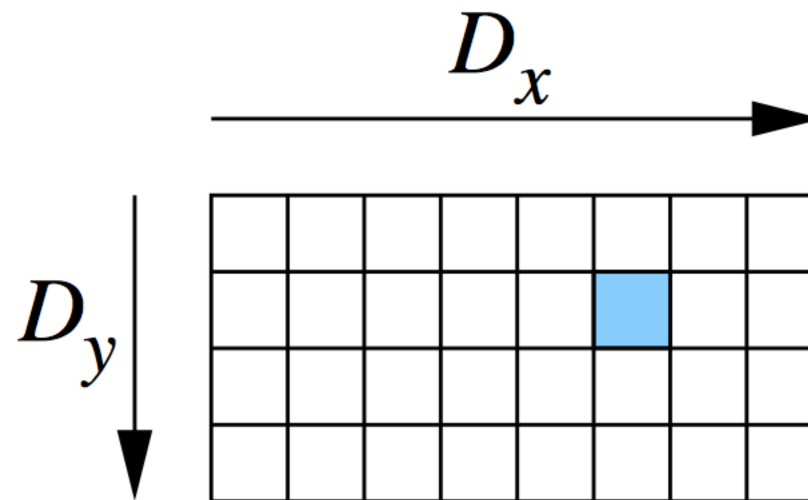
If you are working with 1D vectors, then Dy and Dz could be zero. Then threadIdx is x, and threadId is x. Working with 2D arrays, then Dz would be zero.

threadId in different kind of **block**s

one–dimensional

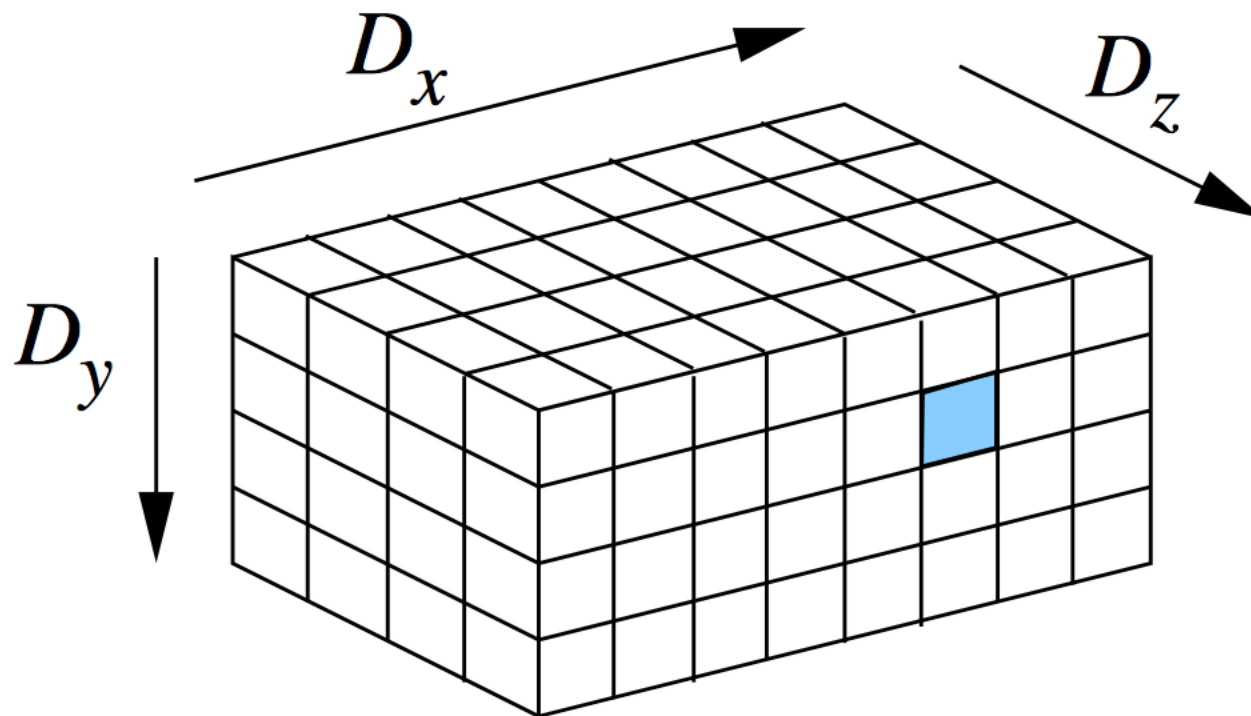


two–dimensional



x + y Dx

three–dimensional



x+y Dx + z Dx Dz

Max threads in block: 512 Fermi;1024 for Compute Capability 2

Thread Index (threadIdx) and ThreadId

In 1-D: For one block, the unique threadId of thread of index (x) = x
or threadIdx.x = x; Maximum size problem: 1024 threads

In 2-D, with block of size (Dx, Dy), the unique threadId of
thread with index (x,y): threadId= x + y Dx

threadIdx.x = x; threadIdx.y = y

In 3-D, with block of size (Dx,Dy, Dz), the unique threadID of
thread with index (x,y,z): threadId = x+y Dx + z Dx Dz

threadIdx.x = x; threadIdx.y = y; threadIdx.z = z

Total number of threads = Thread_per_block* Number of blocks

Max number of threads_per_block = 1024 for Cuda Capability 2.0 +
Max dimensions of thread block (1024,1024, 64) but max threads 1024

Typical sizes: (16, 16), (32, 32) optimum size will depend on program

Grids of Blocks

When you have more data than the maximum number of threads per block.
Handle additional threads with more blocks.

A **grid** is a 1D (x), 2D (x,y) or 3D (x,y,z) array of blocks

(gridDim.x, gridDim.y, and gridDim.z)

Each block has a blockIdx which is the index of the block within the grid

(blockId.x, blockId.y, blockId.z)

Remember, each thread has a threadIdx within the block; it is 3D:
threadIdx.x, threadIdx.y, and threadIdx.z

Grid of Blocks

One block is too small to handle most GPU problems. Need a grid of blocks.
Blocks can be in 1-D, 2-D, or 3-D grids of thread blocks. All blocks are the same size.

The number of thread blocks depends usually on the number of threads needed for a particular problem.

Example for a 1D grid of 2D blocks:

```
int main()
 {
    int numBlocks = 16;
    dim3 threadsPerBlock (N,N);    //1 block of N x N x 1 threads

   MatAdd<<<numBlocks, threadsPerBlock>>>( A, B, C);
```

Each block identified by build-in variable: **BlockIdx**. Dimension of block given by built-in **blockDim** variable (Dx, Dy, Dz). This is same as threadsPerBlock

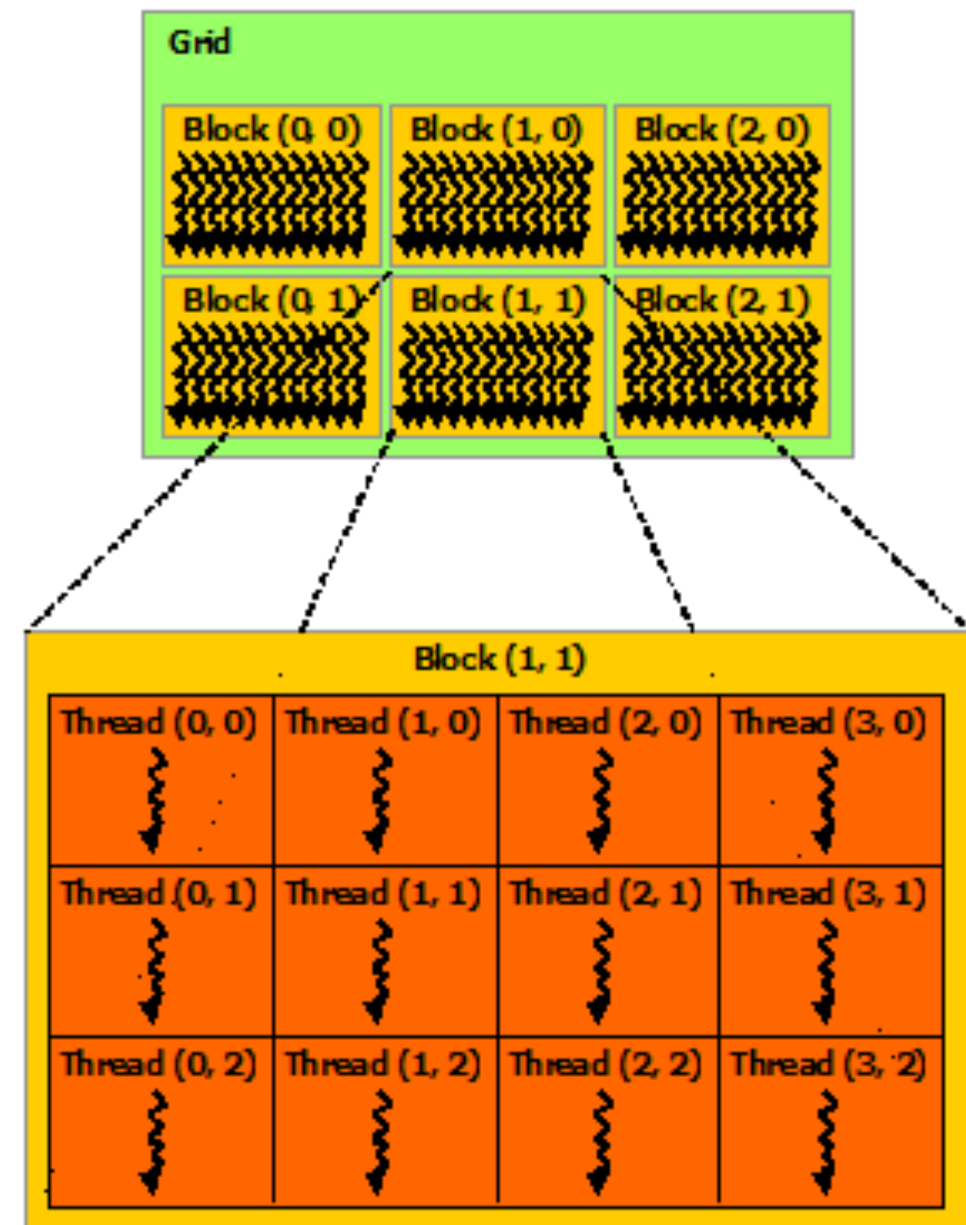
Max dimension size of a grid size (x,y,z): (65535, 65535, 65535) Compute C. <3
Compute Capability >3: (2147483647, 65535, 65535)

2D Grid

dim3 GridDim(3,2);
dim3 BlockDim(4,3);   //Dx, Dy, Dz

In kernel (using thread index):

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

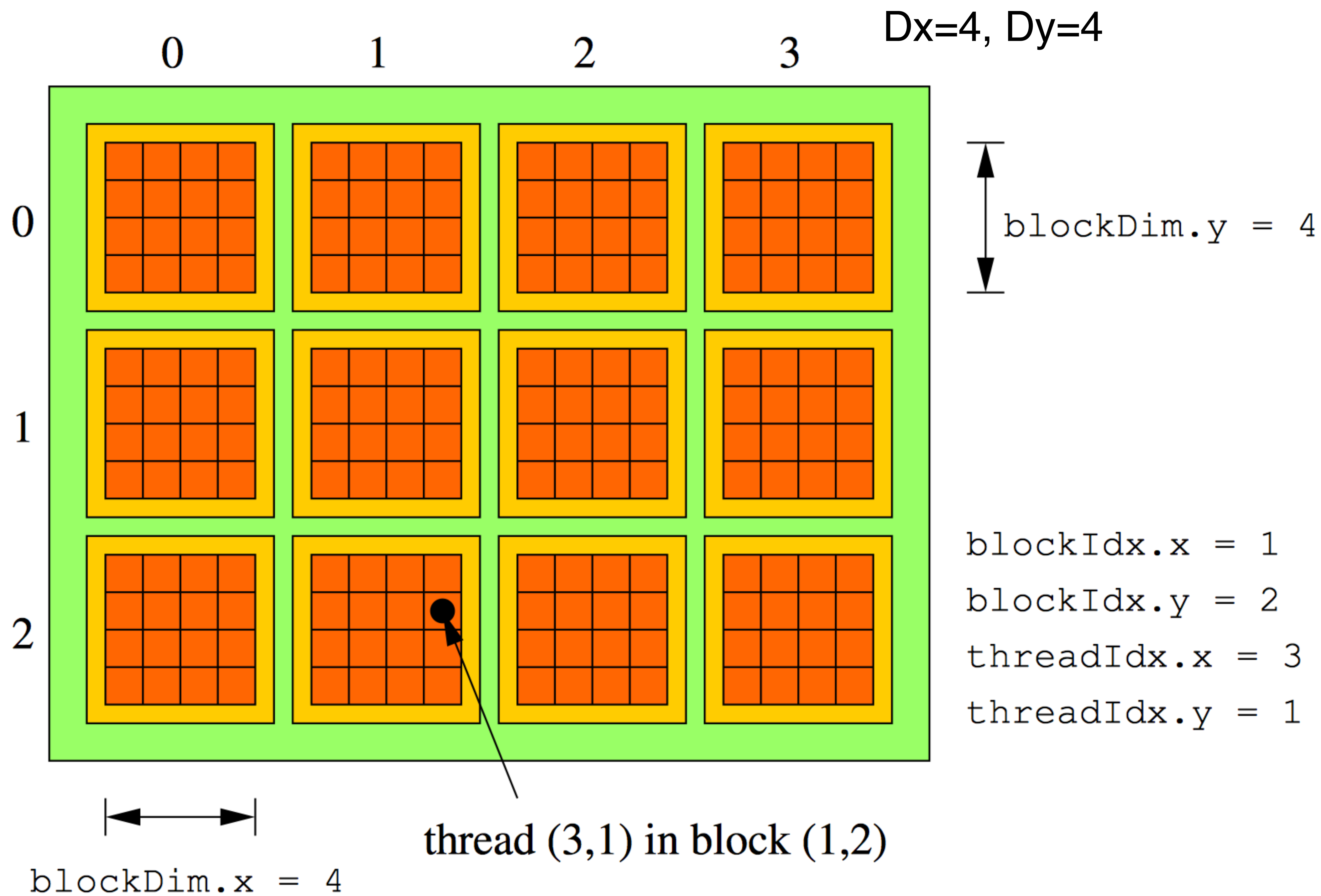


Consider N x N array and each element assigned a thread:

dim3 threadsPerBlock(16,16); //Dx, Dy = BlockDim

dim3 numBlocks(N/threadsPerBlock.x, N/threadsPerBlock.y)// GridDim

MatAdd<<<numBlocks, threadsPerBlock>>>(A,B,C);



threadIdx:

$$x = \texttt{blockIdx.x} \times \texttt{blockDim.x} + \texttt{threadIdx.x} = 1 \times 4 + 3 = 7$$

$$y = \texttt{blockIdx.y} \times \texttt{blockDim.y} + \texttt{threadIdx.y} = 2 \times 4 + 1 = 9$$

threadId = 9 x 16+8

Class Problem:

Write kernel to carry out the following: y[i] = a * x[i] + y[i];

saxpy (int n, float a, float *x, float *y)

```
#include <iostream>
using namespace std;
int main()
{
  float * x;      //host arrays
  float * y;
  float * d_x;    //device arrays
  float * d_y;
  int n = 1048576;
  x = new float[n];
  y = new float[n];
  // intialize x,y; a initialized in kernel call
  for (int i = 0; i<n; i++)
   {
     x[i] = (float)i;
     y[i] = (float)i;
   }
  cudaMalloc(&d_x, n*sizeof(float));
  cudaMalloc(&d_y, n*sizeof(float));
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, n*sizeof(float), cudaMemcpyHostToDevice)

  saxpy<<<4096,256>>>(n, 2.0, d_x, d_y);  //4096*256 =  1048576
```

Answer: SAXPY kernel

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a * x[i] + y[i];
}
```