1

( )

1.1

.

, , ,

, , . [**2016**] (1.1)

. 1.1:

X :(1.1)

$$Y[i] = sin(X[i])/cos(X[i]) \quad (1.1)$$

1.2

.

, , - ,

- .

. [**sanders2010cuda**] (1.2)

. 1.2: NVIDIA

:(1.2)

$$intthreadId = blockIdx.x * blockDim.x + threadIdx.x; \quad (1.2)$$

1.3

, .

, , , , . .

. [**farber2011cuda**] (1.3)

. 1.3:

:(1.3)

$$intnumBlock = (numThreads + numThreadsPerBlock - 1)/numThredsPerBlock \quad (1.3)$$

CUDA,

.

sum2Darray.cu

```
#include <iostream>

using namespace std;
#define n 800

__global__ void matAdd(float (*)[n], float (*)[n],
                                        float (*)[n]);

int main()
{
        const int memSize = sizeof(float) * n * n;
        //
        float (*a)[n] = (float(*)[n])malloc(memSize);
        float (*b)[n] = (float(*)[n])malloc(memSize);
        float (*c)[n] = (float(*)[n])malloc(memSize);

                //
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                        a[i][j] = b[i][j] = 0.5;
                        c[i][j] = 0;
                }
        }

        //
        float (*devA)[n];
        float (*devB)[n];
        float (*devC)[n];
        size_t pitch;

        //
        cudaMallocPitch(&devA, &pitch, n * sizeof(float), n);
```

```
        cudaMallocPitch(&devB, &pitch, n * sizeof(float), n);
        cudaMallocPitch(&devC, &pitch, n * sizeof(float), n);

        cudaMemcpy2D(devA, pitch, a, n * sizeof(float), n * s
        cudaMemcpy2D(devB, pitch, b, n * sizeof(float), n * s

        dim3 numThreadsPerBlock(10, 10);
        dim3 numBlocks((n + numThreadsPerBlock.x - 1) / numT
                                        (n + numThreadsPerBlo
        matAdd<<<numBlocks, numThreadsPerBlock>>>(devA, devB,

        cudaMemcpy2D(c, n * sizeof(float), devC, pitch, n * s

        cudaFree(devA);
        cudaFree(devB);
        cudaFree(devC);

        for (int i = 0; i < 5; i++) {
                for (int j = 0; j < 5; j++) {
                        cout << c[i][j] << ' ';
                }
                cout << '\n';
        }

        free(a);
        free(b);
        free(c);

        return 0;
}

__global__ void matAdd(float (*A)[n], float (*B)[n],
                                        float (*C)[n])
{
        int i = blockDim.x * blockIdx.x + threadIdx.x;
```

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

        if (i < n && j < n)
                C[i][j] = A[i][j] + B[i][j];
}
```