



---

# درس برنامه نویسی چند هسته ای

---

تمرین پنجم



## 1. در برنامه نویسی کودا منظور از Compute Capability چیست؟

compute capability یک دستگاه با یک شماره ورژن نمایش داده می شود، همچنین به SM “version” نیز معروف است. این شماره ورژن ویژگی هایی که سخت افزار GPU پشتیبانی می کند را مشخص می کند و توسط برنامه در زمان اجرا استفاده می شود تا مشخص کند کدام ویژگی های سخت افزار و یا دستورات در GPU فعلی موجود است.

(منبع: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability>)

## 2. PTX چیست و چگونگی استفاده ی از آن را توضیح دهید. ساختار PTX به چه صورت است.

PTX یک ماشین مجازی (virtual machine) و ISA اجرای موازی نخ ها (-parallel-thread execution) سطح پایین می باشد. PTX می تواند خروجی چندین ابزار باشد یا توسط چندین توسعه دهنده به صورت مستقیم نوشته شود. PTX با هدف مستقل بودن از معماری GPU می باشد، بنابراین کد یکسان می تواند توسط معماری GPU های مختلف دوباره اجرا شود. PTX یک زبان اسمبلی pseudo می باشد که بر در محیط های برنامه نویسی کودا استفاده می شود، کامپایلر nvcc کد نوشته شده در کودا را در PTX به یک زبان شبیه C++ ترجمه می کند، و درایور گرافیک دارای یک کامپایلر برای ترجمه کد PTX به کد باینری می باشند که می توانند بر روی هسته های محاسباتی اجرا شوند. می توان از اسمبلی PTX Inline در کودا استفاده کرد. کد PTX تولید شده برای برخی از compute capability های خاص همیشه می توانند به کد باینری با compute capability بزرگ تر یا مساوی کامپایل شوند. توجه داشته باشید که یک کد باینری کامپایل شده در نسخه های قدیمی تر PTX ممکن است از یک سری ویژگی های سخت افزار پشتیبانی نکنند. در نتیجه کد باینری نهایی ممکن است بدتر از حالت ممکن باشد اگر توسط آخرین نسخه PTX تولید نشده باشد.

برای اینکه بتوانید بر روی ویژگی های معماری های آینده با compute capability های بزرگتر کد خود را اجرا کنید، برنامه باید کد PTX را که در لحظه برای این ها کامپایل می شود را بر روی این دستگاه ها load کند.

(منابع: <https://docs.nvidia.com/cuda/ptx-writers-guide-to-interoperability/index.html#introduction>)

و [interoperability/index.html#introduction](https://docs.nvidia.com/cuda/ptx-writers-guide-to-interoperability/index.html#introduction)

[https://docs.nvidia.com/cuda/cuda-](https://docs.nvidia.com/cuda/cuda-https://en.wikipedia.org/wiki/Parallel_Thread_Execution) و [https://en.wikipedia.org/wiki/Parallel\\_Thread\\_Execution](https://en.wikipedia.org/wiki/Parallel_Thread_Execution)  
( [c-programming-guide/index.html#ptx-compatibility](https://en.wikipedia.org/wiki/Parallel_Thread_Execution) )

3. ساختار یک CUDA core را توضیح دهید. آیا در یک CUDA core امکان بکارگیری همزمان واحد ممیز شناور و

عدد صحیح وجود دارد؟ اگر جواب مثبت است از چه معماری این قابلیت اضافه شده است؟

یک خط لوله که می تواند جمع های اعشاری 32 بیتی، ضرب های اعشاری 32 بیتی، عملیات های 32 به 8 بیت همانند شیفت، muls و عملیات های مشابه و همچنین ایجاد درخواست حافظه بر روی SM واحد خود با داده ها بکار گرفته شود تا همواره بر روی داده ها کار انجام شود. همچنین درخواست های تابع محاسباتی خاص و synchronized شدن توسط هر واحد SM برای محاسبه مناسب الگوریتم های موازی.

هنگامی که یک کرنل CUDA در حال اجرا می باشد، بر روی تمامی "CUDA threads" کلون (cloned) می شود و این "CUDA threads" از همه "CUDA pipelines" عبور می کنند. هر خط لوله توانایی threading تا 16 نخ را دارد. این اجازه می دهد تمامی واحد های محاسباتی (integer، اعشاری ممیز شناور، درخواست های داده،...) به صورت موثر استفاده شوند. زمانی که یک "warp" از "CUDA threads" برای فرستادن انتخاب می شود، این 32 "CUDA threads" "pipelines" برای اجرا و قفل شدن در آن نیاز دارد. این باعث می شود 32 "CUDA threads" "pipelines" به صورت یک گروه کار کنند، تا به صورت implicit تکرار قفل مرحله ای از دستورات کرنل CUDA را انجام دهد. این اتفاق همزمان در بقیه "warp" ها نیز رخ می دهد. "CUDA core" یک خط لوله می باشد که برای نگاشت ( ) برخی "CUDA threads" بر روی آن ها به صورت synchronized می باشد، تا به صورت موثر محاسبات مداوم را انجام دهد. "CUDA core" یک هسته کامل نمی باشد، یک خط لوله می باشد. تنها به بقیه منابع درخواست می فرستد و نتایج را به صورت پاسخ دریافت می کند.

همچنین از معماری Fermi امکان بکارگیری همزمان واحد ممیز شناور و عدد صحیح وجود دارد؛

شکل زیر این معماری را نمایش می دهد.

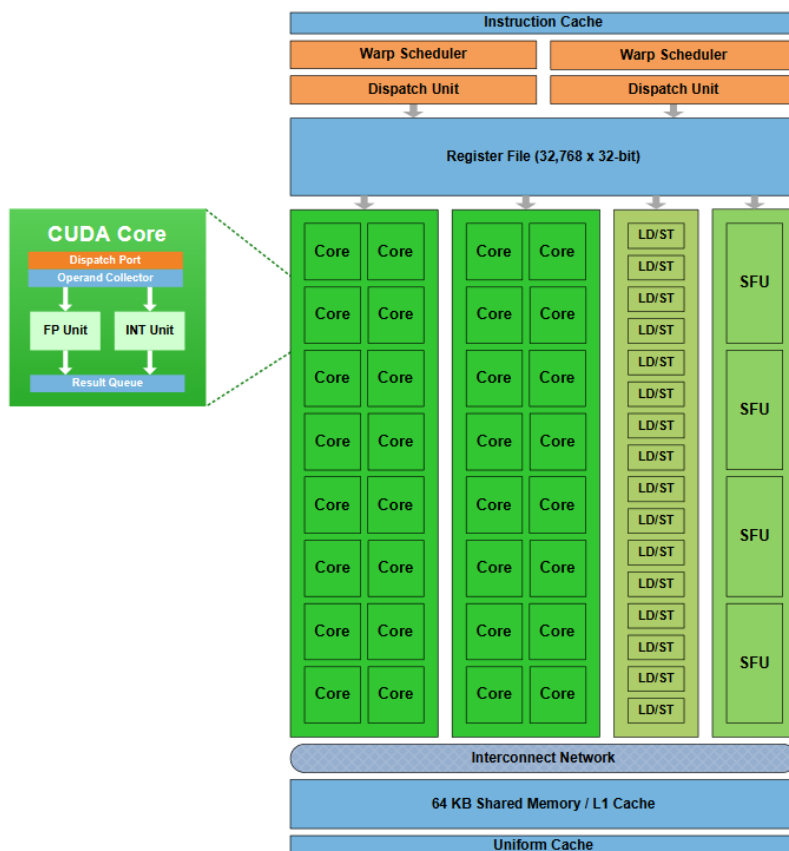


Figure 6. Each Fermi SM includes 32 cores, 16 load/store units, four special-function units, a 32K-word register file, 64K of configurable RAM, and thread control logic. Each core has both floating-point and integer execution units. (Source: NVIDIA)

(منابع: <https://www.quora.com/What-is-a-CUDA-core> و

[https://www.nvidia.com/content/PDF/fermi\\_white\\_papers/P.Glaskowsky\\_NVIDIA's\\_Fermi-](https://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-)

[\(The\\_First\\_Complete\\_GPU\\_Architecture.pdf](#)

4. هسته های tensor چیست و چه استفاده ای دارند؟ آیا امکان استفاده ی آن ها به صورت مستقیم وجود دارد؟

هسته های tensor باعث تسریع عملیات های ماتریس، که مبنای AI هستند، می شود و محاسبات ضرب و جمع ماتریس با دقت مختلط، همانند شکل زیر را، (mixed-precision) در یک عمل واحد اجرا می کند. با اجرای صد ها هسته tensor به صورت موازی در یک GPU، باعث افزایش چشمگیر توان (throughput) و بهره وری (efficiency) می شود.

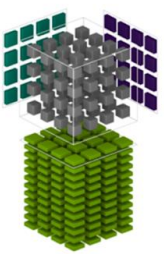
$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} + \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} = \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

HMMA FP16 or FP32  
IMMA INT32

FP16  
INT8 or UINT8

FP16  
INT8 or UINT8

FP16 or FP32  
INT32



هسته های tensor اگرچه برای فضای GPU کاملاً جدید هستند، اما از ALU خط لوله های استاندارد فاصله زیادی ندارند. چگالی آن تغییر کرده- آن ها اکنون ماتریس های با اندازه متغیر را بجای مقادیر scalar بسته بندی شده SIMD اجرا می کنند- اما ریاضیات بدین صورت نمی باشد. در نهایت trade off نسبتاً مستقیمی بین انعطاف پذیری ( هسته های tensor در عملیات scalar افتضاح می باشند) و توان (throughput) می باشد، از آنجا که هسته های tensor می توانند عملیات بیشتری را همزمان بسته بندی کنند به همین دلیل آن ها بسیار سخت تر هستند و نیاز به کسری از منطق کنترل هنگامی که هزینه های برای هر ALU تقسیم می شود دارند.

(منابع: <https://www.pny.com/professional/explore-our-products/learn-about-nvidia-quadro/nvidia-tensor-cores-deep-dive/3#:~:text=To%20recap%2C%20the%20tensor%20core,4%20FP16%20or%20FP32%20matrix> و <https://www.anandtech.com/show/12673/titan-v-deep-learning-quadro/nvidia-tensor-cores-deep-dive/3#:~:text=To%20recap%2C%20the%20tensor%20core,4%20FP16%20or%20FP32%20matrix>)

( و )

5. آیا امکان استفاده تواما OpenMP و کودا با هم وجود دارد؟ در صورت مثبت بودن جواب به نظر شما در چه مواردی کاربرد دارد؟

بله، هنگامی که بخواهیم قسمت سریال برنامه را که در host اجرا می شود را نیز موازی کنیم و به دلیل سریار های موجود در device و عدم وجود کار های موازی کافی استفاده از device برای آن کار مناسب نباشد، از OpenMP در host استفاده می کنیم، و بقیه برنامه را نیز به کمک کودا به صورت موازی اجرا می کنیم.

به صورت کلی در صورتی بتوانیم بخش سریال برنامه که توسط host اجرا می شود را موازی کنیم، می توانیم بخش موازی را به کمک OpenMP در host موازی کنیم و سپس نتایج آن بخش را برای ادامه محاسبات به device واگذار کنیم.

6. برنامه ای بنویسید که هر نخ grid و block خود را به صورت زیر چاپ کند.

Hello CUDA I'm a thread from grid X and block Y

ابتدا Signature تابع printWithCuda را که در تعداد بلوک ها در متغیر blockDim و تعداد نخ های درون هر بلوک در متغیر threadIdx گرفته می شود و سپس عبارت مورد نظر را چاپ می کند به همراه کتاب خانه های مورد نیاز برای این برنامه، به صورت زیر تعریف می گردد:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

// Helper function for using CUDA to print vectors in parallel.
cudaError_t printWithCuda(int blockDim, int threadIdx)
```

سپس تابعی که قرار است در GPU اجرا شود را به صورت زیر نوشته می شود:

```
__global__ void printKernel()
{
    printf("Hello CUDA Im a thread from grid %d and block %d \n", threadIdx.x,
    blockIdx.x);
}
```

سپس تابع main به صورت زیر نوشته می شود:

```
int main()
{
    // print vectors in parallel.
    cudaError_t cudaStatus = printWithCuda(2, 7);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }

    // cudaDeviceReset must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }

    return 0;
}
```

}

سپس تابع `printWithCuda` به صورت زیر نوشته می شود:

```
// Helper function for using CUDA to print vectors in parallel.
cudaError_t printWithCuda(int blockNum, int threadNum)
{
    cudaError_t cudaStatus;

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU
installed?");
    }

    // Launch a kernel on the GPU with one thread for each element.
    printKernel<<<blockNum, threadNum>>>();

    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
launching addKernel!\n", cudaStatus);
    }

    // Check for any errors launching the kernel
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
    }

    return cudaStatus;
}
```

(همچنین کد این بخش در فایل Q6.cu به پیوست قرار داده شده است.)

7. هنگام ساخت پروژه کودا در Visual Studio برای مثال و امتحان کارکرد صحیح مجموعه ی کودا، کد پیش فرضی برای جمع دو بردار تولید می کند. قسمت های زیر را انجام داده و زمان اجرا را برای اندازه ی ده میلیون المان گزارش کنید و به سؤال های موجود پاسخ دهید ( زمان پرو و کپی کردن آرایه ها را در نظر نگیرید و فقط زمان اجرای kernel و عملیات جمع در نظر گرفته شود)

a. برنامه ی جمع بردار را به صورت سریال بنویسید.

برنامه جمع بردار به صورت سریال به صورت زیر نوشته می شود:

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

void fillMat(int * v, int matSizeX, int matSizeY);
```

```
void add(int *a, int *b, int *c, int matSizeX, int matSizeY);
void printMat(int * v, int matSizeX, int matSizeY);
```

```
int main()
{
    const int matSizeX = 1000;
    const int matSizeY = 10000;
    int * a;
    int * b;
    int * c;
    double elapsedtime, starttime;
    a = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
    b = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
    c = (int*)malloc(sizeof(int)*matSizeX*matSizeY);

    fillMat(a, matSizeX, matSizeY);
    fillMat(b, matSizeX, matSizeY);

    starttime = omp_get_wtime();

    add(a, b, c, matSizeX, matSizeY);

    elapsedtime = omp_get_wtime() - starttime;

    /*printMat(a, matSizeX, matSizeY);
    printMat(b, matSizeX, matSizeY);
    printMat(c, matSizeX, matSizeY);*/

    // report elapsed time
    printf("Time Elapsed %f ms\n", elapsedtime * 1000);

    return EXIT_SUCCESS;
}

// Fills a Matrice with data
void fillMat(int * v, int matSizeX, int matSizeY) {
    static int L = 0;
    for (int i = 0; i < matSizeX; i++) {
        for (int j = 0; j < matSizeY; j++)
            v[i*matSizeY + j] = L++;
    }
}

void add(int *a, int *b, int *c, int matSizeX, int matSizeY) {
    int i, j;
    for (i = 0; i < matSizeX; i++)
    {
        for (j = 0; j < matSizeY; j++)
        {
            c[i*matSizeY + j] = a[i*matSizeY + j] + b[i*matSizeY + j];
        }
    }
}
}
```



```
// Prints a Matrices to the stdout.
void printMat(int * v, int matSizeX, int matSizeY) {
    int i;
    printf("[-] Vector elements: \n");
    for (int i = 0; i < matSizeX; i++) {
        for (int j = 0; j < matSizeY; j++)
            printf("%d\t", v[i*matSizeY + j]);
        printf("\n");
    }
    printf("\b\b\t\n");
}
```

برای بررسی صحت کد، خروجی آن برای جمع ماتریس های به اندازه 4 بررسی می گردد،  
نتیجه اجرای این کد برای این ماتریس ها برابر است با:

```
[-] Vector elements:
0      1
2      3

[-] Vector elements:
4      5
6      7

[-] Vector elements:
4      6
8      10

Time Elapsed 0.000200 ms
```

حال که درستی کد مورد نظر بررسی گردید زمان اجرای آن برای 10 میلیون المان بررسی  
می گردد، نتیجه خروجی این برنامه برای جمع این ماتریس های برابر است با:

```
Time Elapsed 33.067200 ms
```

در نتیجه 33.0672 میلی ثانیه زمان برای اجرای این برنامه به صورت سریال نیاز است.

(همچنین کد این بخش در فایل HW5\_Q7\_Serial.cpp به پیوست قرار داده شده است.)

(برای نوشتن این برنامه از کد ارائه شده در آزمایش 5 استفاده گردیده است.)

(کد این بخش توسط کامپایلر Visual C++ کامپایل شده است)

b. برنامه ی جمع بردار را با استفاده از OpenMP موازی کنید.

از نتایج بدست آمده در آزمایش 2، این نتیجه بدست آمده است که در این مسئله تجزیه یک بعدی و سطری نتیجه مطلوب تری نسبت به بقیه روش ها دارد، بنابراین برنامه نوشته شده در قسمت قبل به صورت زیر موازی می شود:

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

void fillMat(int * v, int matSizeX, int matSizeY);
void add(int *a, int *b, int *c, int matSizeX, int matSizeY);
void printMat(int * v, int matSizeX, int matSizeY);

int main()
{
    const int matSizeX = 1000;
    const int matSizeY = 10000;
    int * a;
    int * b;
    int * c;
    double elapsedtime, starttime;
    a = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
    b = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
    c = (int*)malloc(sizeof(int)*matSizeX*matSizeY);

    fillMat(a, matSizeX, matSizeY);
    fillMat(b, matSizeX, matSizeY);

    starttime = omp_get_wtime();

    add(a, b, c, matSizeX, matSizeY);

    elapsedtime = omp_get_wtime() - starttime;

    /*printMat(a, matSizeX, matSizeY);
    printMat(b, matSizeX, matSizeY);
    printMat(c, matSizeX, matSizeY);*/

    // report elapsed time
    printf("Time Elapsed %f ms\n", elapsedtime * 1000);

    return EXIT_SUCCESS;
}

// Fills a Matrice with data
void fillMat(int * v, int matSizeX, int matSizeY) {
    static int L = 0;
    for (int i = 0; i < matSizeX; i++) {
        for (int j = 0; j < matSizeY; j++)
            v[i*matSizeY + j] = L++;
    }
}
```

```

void add(int *a, int *b, int *c, int matSizeX, int matSizeY) {
    int i, j;
    #pragma omp parallel for private (j)
    for (i = 0; i < matSizeX; i++)
    {
        for (j = 0; j < matSizeY; j++)
        {
            c[i*matSizeY + j] = a[i*matSizeY + j] + b[i*matSizeY + j];
        }
    }
}

// Prints a Matrices to the stdout.
void printMat(int * v, int matSizeX, int matSizeY) {
    int i;
    printf("[-] Vector elements: \n");
    for (int i = 0; i < matSizeX; i++) {
        for (int j = 0; j < matSizeY; j++)
            printf("%d ", v[i*matSizeY + j]);
        printf("\n");
    }
    printf("\b\b \n");
}

```

برای بررسی صحت کد، خروجی آن برای جمع ماتریس های 2 در 2 بررسی می گردد،  
نتیجه اجرای این کد برای ماتریس 2 در 2 برابر است با:

```

[-] Vector elements:
0      1
2      3

[-] Vector elements:
4      5
6      7

[-] Vector elements:
4      6
8      10

Time Elapsed 4.069000 ms

```

حال که درستی کد مورد نظر بررسی گردید زمان اجرای آن برای 10 میلیون المان بررسی  
می گردد، نتیجه خروجی این برنامه برای جمع ماتریس های 1000 در 10000 برابر است  
با:

Time Elapsed 27.137300 ms

در نتیجه 27.1373 میلی ثانیه زمان برای اجرای این برنامه به صورت موازی نیاز است.

(همچنین کد این بخش در فایل HW5\_Q7\_Parallel.cpp به پیوست قرار داده شده است.)

(کد این بخش توسط کامپایلر Visual C++ کامپایل شده است)

c. برنامه ی پیش فرض را با تغییر `grid size` و `block size` ، برای انجام هر جمع توسط یک نخ آماده کنید.

برنامه پیش فرض به صورت زیر تغییر یافته است:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <cuda.h>

cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);
void fillMat(int * v, int matSizeX, int matSizeY);
void printMat(int * v, int matSizeX, int matSizeY);

__global__ void addKernel(int *c, const int *a, const int *b, int size)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;

    if (i < size) {
        c[i] = a[i] + b[i];
    }
}

int main()
{
    const int matSizeX = 1000;
    const int matSizeY = 10000;
    const int arraySize = matSizeX*matSizeY;

    int * a;
    int * b;
    int * c;
    ;
    a = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
    b = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
    c = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
    fillMat(a, matSizeX, matSizeY);
    fillMat(b, matSizeX, matSizeY);
    // Add vectors in parallel.
    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }
}
```

```

    }

    /*printMat(a, matSizeX, matSizeY);
    printMat(b, matSizeX, matSizeY);
    printMat(c, matSizeX, matSizeY);*/

    // cudaDeviceReset must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }

    return 0;
}

void fillMat(int * v, int matSizeX, int matSizeY) {
    static int L = 0;
    for (int i = 0; i < matSizeX; i++) {
        for (int j = 0; j < matSizeY; j++)
            v[i*matSizeY + j] = L++;
    }
}

void printMat(int * v, int matSizeX, int matSizeY) {
    int i;
    printf("[-] Vector elements: \n");
    for (int i = 0; i < matSizeX; i++) {
        for (int j = 0; j < matSizeY; j++)
            printf("%d ", v[i*matSizeY + j]);
        printf("\n");
    }
    printf("\b\b \n");
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;
    cudaError_t cudaStatus;

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    //checkError(cudaStatus, 0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU
installed?");
        goto Error;
    }

    // Allocate GPU buffers for three vectors (two input, one output)
    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    //checkError(cudaStatus, 1);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

```

```

}

cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMalloc failed!");
    goto Error;
}

// Copy input vectors from host memory to GPU buffers.
cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

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

    cudaEventRecord(start, NULL);
    int numOfThread = 1024;
    int n = int((size - 1) / numOfThread) + 1;
    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<n, numOfThread>>>(dev_c, dev_a, dev_b, size);

    cudaEventRecord(stop, NULL);
    cudaStatus = cudaEventSynchronize(stop);
    float msecTotal = 0.0f;
    cudaStatus = cudaEventElapsedTime(&msecTotal, start, stop);
    fprintf(stderr, "Elapsed Time is %f ms \n", msecTotal);

// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
    goto Error;
}

// cudaDeviceSynchronize waits for the kernel to finish, and returns
// any errors encountered during the launch.

```

```

cudaStatus = cudaDeviceSynchronize();

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching
addKernel!\n", cudaStatus);
    goto Error;
}

// Copy output vector from GPU buffer to host memory.
cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

Error:
    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);

    return cudaStatus;
}

```

برای بررسی صحت کد، خروجی آن برای جمع ماتریس های 2 در 2 بررسی می گردد،  
نتیجه اجرای این کد برای این ماتریس ها برابر است با:

```

Elapsed Time is 0.010816 ms
[-] Vector elements:
0 1
2 3

[-] Vector elements:
4 5
6 7

[-] Vector elements:
4 6
8 10

```

حال که درستی کد مورد نظر بررسی گردید زمان اجرای آن برای ماتریس های 1000 در  
10000 بررسی می گردد، نتیجه خروجی این برنامه برای جمع این ماتریس ها برابر است  
با:

```

Elapsed Time is 3.525952 ms

```

در نتیجه 3.525952 میلی ثانیه زمان برای اجرای این برنامه نیاز است.

(همچنین کد این بخش در فایل HW5\_Q7\_c.cu به پیوست قرار داده شده است.)

i. اندازه ی grid بزرگ تر باعث تسریع بیشتر می شود یا اندازه ی block بزرگ تر؟ چرا؟

با کمتر کردن تعداد نخ های موجود در یک بلوک تا یک اندازه مشخص زمان اجرای برنامه کاهش می یابد، اما پس از آن مقدار کم کردن تعداد نخ های بلوک تاثیر زیادی بر روی زمان اجرا نمی گذارد، برای مثال زمان اجرای برنامه زمانی که تعداد نخ ها در هر بلوک برابر 1024 نخ باشد را در شکل زیر مشاهده می کنید:

```
Elapsed Time is 3.525952 ms
```

حال با تغییر تعداد نخ های هر بلوک به اندازه 512 زمان اجرای برنامه برابر است با:

```
Elapsed Time is 3.492224 ms
```

همانطور که مشاهده می شود، با افزایش تعداد بلوک ها و کاهش تعداد نخ های هر بلوک زمان اجرا کاهش می یابد و برنامه سریع تر اجرا می شود، حال هر چه تعداد نخ ها را از این مقدار کمتر کنیم، زمان اجرا تغییر زیادی نمی کند. بنابراین با تغییر تعداد اندازه نخ های هر بلوک به 512 که در نتیجه آن افزایش تعداد بلوک ها به دلیل اینکه تعداد بلوک ها افزایش می یابد scheduler می تواند در صورت اینکه یک بلوک به دسترسی به حافظه نیاز داشت بلوک دیگری را انتخاب و اجرا کند و همچنین با کاهش اندازه نخ های بلوک ها در هر لحظه تعداد بلوک های بیشتری را می توان اجرا کرد، گرچه این کار خیلی در زمان تاثیر نمی گذارد اما به دلیل اینکه نخ های یک بلوک با یک دیگر باید همگام پیش بروند بنابراین کمتر کردن آن ها تا حدی باعث بهبود سرعت اجرای برنامه می شود.

d. برنامه ی پیش فرض تولید شده را به گونه ای تغییر دهید که هر نخ بیشتر از یک المان را جمع کند.

برنامه پیش فرض به صورت زیر تغییر یافته است:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
```



```

cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size);
void fillMat(int *v, int matSizeX, int matSizeY);
void printMat(int *v, int matSizeX, int matSizeY);

__global__ void addKernel(int *c, const int *a, const int *b, int size, int nblock)
{
    int n = int(((size-1)/blockDim.x) + 1)/nblock + 1;
    int i = (threadIdx.x + blockDim.x * blockIdx.x)*n;

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

int main()
{
    const int matSizeX = 1000;
    const int matSizeY = 10000;
    const int arraySize = matSizeX * matSizeY;

    int * a;
    int * b;
    int * c;
    ;
    a = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
    b = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
    c = (int*)malloc(sizeof(int)*matSizeX*matSizeY);
    fillMat(a, matSizeX, matSizeY);
    fillMat(b, matSizeX, matSizeY);
    // Add vectors in parallel.
    cudaError_t cudaStatus = addWithCuda(c, a, b, arraySize);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }

    /*printMat(a, matSizeX, matSizeY);
    printMat(b, matSizeX, matSizeY);
    printMat(c, matSizeX, matSizeY);*/

    // cudaDeviceReset must be called before exiting in order for profiling and
    // tracing tools such as Nsight and Visual Profiler to show complete traces.
    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }

    return 0;
}

```

```

void fillMat(int * v, int matSizeX, int matSizeY) {
    static int L = 0;
    for (int i = 0; i < matSizeX; i++) {
        for (int j = 0; j < matSizeY; j++)
            v[i*matSizeY + j] = L++;
    }
}

void printMat(int * v, int matSizeX, int matSizeY) {
    int i;
    printf("[-] Vector elements: \n");
    for (int i = 0; i < matSizeX; i++) {
        for (int j = 0; j < matSizeY; j++)
            printf("%d ", v[i*matSizeY + j]);
        printf("\n");
    }
    printf("\b\b \n");
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;
    cudaError_t cudaStatus;

    // Choose which GPU to run on, change this on a multi-GPU system.
    cudaStatus = cudaSetDevice(0);
    //checkError(cudaStatus, 0);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU
installed?");
        goto Error;
    }

    // Allocate GPU buffers for three vectors (two input, one output)
    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    //checkError(cudaStatus, 1);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));

    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));

    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    // Copy input vectors from host memory to GPU buffers.

```

```

cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

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

cudaEventRecord(start, NULL);
int nblock = 1;
int numOfThread = 1024;

// Launch a kernel on the GPU with one thread for each element.
addKernel << <nblock, numOfThread >> > (dev_c, dev_a, dev_b, size, nblock);

cudaEventRecord(stop, NULL);
cudaStatus = cudaEventSynchronize(stop);
float msecTotal = 0.0f;
cudaStatus = cudaEventElapsedTime(&msecTotal, start, stop);
fprintf(stderr, "Elapsed Time is %f ms \n", msecTotal);

// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "addKernel launch failed: %s\n",
        cudaGetErrorString(cudaStatus));
    goto Error;
}

// cudaDeviceSynchronize waits for the kernel to finish, and returns
// any errors encountered during the launch.
cudaStatus = cudaDeviceSynchronize();

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
launching addKernel!\n", cudaStatus);
    goto Error;
}

// Copy output vector from GPU buffer to host memory.
cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

```

Error:

```

    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);

    return cudaStatus;
}

```

برای بررسی صحت کد، خروجی آن برای جمع ماتریس های 2 در 2 بررسی می گردد،  
نتیجه اجرای این کد برای این ماتریس ها برابر است با:

```

Elapsed Time is 0.017536 ms
[-] Vector elements:
0 1
2 3

[-] Vector elements:
4 5
6 7

[-] Vector elements:
4 6
8 10

```

حال که درستی کد مورد نظر بررسی گردید زمان اجرای آن برای ماتریس های 1000 در 10000 بررسی می گردد، نتیجه خروجی این برنامه برای جمع این ماتریس ها برابر است با:

```
Elapsed Time is 37.846306 ms
```

(همچنین کد این بخش در فایل HW5\_Q7\_d.cu به پیوست قرار داده شده است.)

i. ریزدانی (تعداد نخ بیشتر) و درشت دانی (تعداد نخ کمتر) کدام یک برای این مسئله مناسب تر است؟ چرا؟

برای این مسئله، در حالتی که هر نخ می خواهد بیش از یک المان را محاسبه کند، هر چه تعداد نخ ها بیشتر باشد، زمان اجرا بهتر می شود، یعنی برای این مسئله در این حالت ریزدانی مناسب تر می باشد، به این دلیل که محاسبات این برنامه زمان زیادی نیاز ندارد و این برنامه یک برنامه Memory bound می باشد، برای همین با افزایش تعداد نخ ها می توان تاثیر تاخیر حافظه را در برنامه کاهش داد.

ii. اندازه‌ی مناسب grid و block برای حداکثر تسریع چه رابطه‌ای با معماری GPU داشته است؟

با افزایش تعداد block ها به 3256 که در این حالت هر نخ سه المان را محاسبه می‌کند و تعداد نخ‌های هر block برابر 1024 حداکثر تسریع بدست می‌آید؛ که در آن تعداد المان‌هایی که هر نخ محاسبه می‌کند برابر با تعداد MultiProcessor ها (SM) می‌باشد و اندازه مناسب تعداد نخ‌های یک بلوک نیز برابر بیشترین تعداد نخ‌هایی است که یک بلوک می‌تواند پشتیبانی کند. زمان اجرای این برنامه به ازای 3256 بلوک 1024 تایی برابر است با:

Elapsed Time is 4.406720 ms

همچنین زمان محاسبه 10000000 المان برای مقادیر دیگر عبارتست از:

زمان نهایی (Ms)	تعداد کارهای هر نخ	تعداد نخ‌های بلوک	تعداد بلوک‌ها
38.078014	9,766	1024	1
4.406720	3	1024	3256
21.055519	6511	512	3
4.422624	3	512	6511
7.825344	9	512	2171
4.903520	3	256	13021

8. سؤال‌های زیر را به صورت کوتاه جواب بدهید

a. تفاوت PTX و SASS چیست؟

PTX یک ماشین مجازی سطح پایین اجرای موازی نخ ( low-level parallel-thread execution virtual machine) و معماری ISA می‌باشد. PTX، GPU را به عنوان یک دستگاه محاسبات موازی نمایش می‌دهد. همچنین یک مدل برنامه نویسی پایدار و مجموعه دستورات برای هدف کلی برنامه نویسی موازی ارائه می‌دهد، و طراحی شده است که بر روی GPU های NVIDIA موثر باشد. کامپایلرهای سطح بالا همانند CUDA و C/C++ دستورات PTX را تولید می‌کنند، که برای دستورالعمل‌های native ترجمه و بهینه شده‌اند.

SASS یک زبان اسمبلی سطح پایی است که میکروکد های (microcode) باینری را کامپایل می کند، که به صورت native بر روی سخت افزار GPU اجرا می شود.  
(منبع:

[http://developer.download.nvidia.com/NsightVisualStudio/3.0/Documentation/UserGui  
\(de/HTML/Content/PTX\\_SASS\\_Assembly\\_Debugging.htm](http://developer.download.nvidia.com/NsightVisualStudio/3.0/Documentation/UserGui(de/HTML/Content/PTX_SASS_Assembly_Debugging.htm)

b. دلیل وجود اشاره گر دوگانه در آرگومان اول CudaMalloc چیست؟

به این دلیل است که فضای آدرس CPU و GPU از یک دیگر جدا می باشد، در اصل CPU و GPU هر دو دارای خانه ای با شماره x می باشند. امکان دسترسی مستقیم به این خانه در CPU از GPU ممکن نیست و برعکس؛ به صورت کلی به این دلیل است که یک اشاره گر به اشاره گر دیگر می باشد. باید به اشاره گر حافظه GPU اشاره کند. کاری که CudaMalloc انجام می دهد یک اشاره گر حافظه بر روی GPU را allocate می کند که بعداً توسط اولین آرگومان این تابع که به آن داده می شود به آن اشاره می شود.

( منبع: [https://stackoverflow.com/questions/7989039/use-of-cudamalloc-why-the-](https://stackoverflow.com/questions/7989039/use-of-cudamalloc-why-the-double-)

[double-  
pointer#:~:text=We%20cast%20it%20into%20double,the%20first%20argument%20we%  
\(20give.](double-pointer#:~:text=We%20cast%20it%20into%20double,the%20first%20argument%20we%20give.)