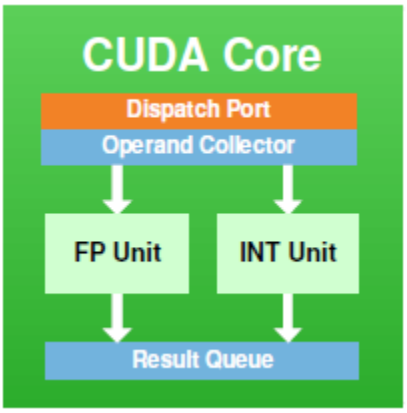
۱. یک ملاک برای توضیح میزان قابلیت‌هایی‌ست که توسط یک سخت‌افزار کودا پشتیبانی می‌شوند که هرچه این عدد بزرگ‌تر باشد، نشان می‌دهد که پردازنده گرافیکی قابلیت‌های بیشتری دارد.

	Compute Capability				
Feature Support (Unlisted features are supported for all compute capabilities)	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**
Atomic functions operating on 32-bit integer values in global memory (Section B.11)	No	Yes			
atomicExch() operating on 32-bit floating point values in global memory (Section B.11.1.3)					
Atomic functions operating on 32-bit integer values in shared memory (Section B.11)	No		Yes		
atomicExch() operating on 32-bit floating point values in shared memory (Section B.11.1.3)					
Atomic functions operating on 64-bit integer values in global memory (Section B.11)					
Warp vote functions (Section B.12)					
Double-precision floating-point numbers	No			Yes	
Atomic functions operating on 64-bit integer values in shared memory (Section B.11)	No				Yes
Atomic addition operating on 32-bit floating point values in global and shared memory (Section B.11.1.1)					
__ballot() (Section B.12)					
__threadfence_system() (Section B.5)					
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)					
Surface functions (Section B.9)					
3D grid of thread blocks					

	Compute Capability				
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**
Maximum dimensionality of grid of thread blocks	2				3
Maximum x-, y-, or z-dimension of a grid of thread blocks	65535				
Maximum dimensionality of thread block	3				
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Maximum number of threads per block	512				1024
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24		32		48
Maximum number of resident threads per multiprocessor	768		1024		1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K
Maximum amount of shared memory per multiprocessor	16 KB				48 KB
Number of shared memory banks	16				32
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				
Maximum width for a 1D texture reference bound to a CUDA array	8192				32768
Maximum width for a 1D texture reference bound to linear memory	2^{27}				
Maximum width and number of layers for a 1D layered texture reference	8192 x 512				16384 x 2048
Maximum width and height for a 2D texture reference bound to linear memory or to a CUDA array	65536 x 32768				65536 x 65535
Maximum width, height, and number of layers for a 2D layered texture reference	8192 x 8192 x 512				16384 x 16384 x 2048
Maximum width, height, and depth for a 3D texture reference bound to a CUDA array	2048 x 2048 x 2048				
Maximum number of textures that can be bound to a kernel	128				
Maximum width for a 1D surface reference bound to a CUDA array	N/A				8192
Maximum width and height for a 2D surface reference bound to a CUDA array					8192 x 8192
Maximum number of surfaces that can be bound to a kernel					8
Maximum number of instructions per kernel	2 million				

۲. PTX مخفف Parallel Thread Execution و نام یک ماشین مجازی اجرای کد گرافیکی و یک زبان pseudo-assembly یا ISAاست که در محیط کودا استفاده می‌شود. PTX از پر کامپایلر CUDA-NVCC این کد را به PTX تبدیل می‌کند و درایور گرافیکی این کد PTX را تبدیل به کد باینری می‌کند که قابل اجرا روی هسته‌های پردازشی می‌شود. این ISA مقدار بسیار زیادی رجیستر برای پردازنده تصور می‌کند.
یک برنامه‌ی PTX نحوه‌ی اجرای یک نخ از یک آرایه نخ‌های موازی را توصیف می‌کند. یک Cooperative Thread Array یا CTA یک آرایه از نخ‌هاست که یک کرنل را به صورت هم‌روند یا موازی اجرا می‌کنند.

۳. یک CUDA Core یک پایپ‌لاین است که می‌تواند ضرب و تقسیم اعشاری ۳۲ بیتی و عملیات‌های صحیح ۸ بیتی انجام دهد و همچنین درخواست‌های حافظه در SM خود تولید کند که SM همیشه دیتا برای کار کردن داشته‌باشد. همچنین درخواست‌های انجام عملیات‌های خاص را صادر می‌کند و توسط SM خود سنکرون می‌شود تا محاسبات موازی را به درستی انجام دهد.



وقتی یک Cuda Kernel در حال اجراست، روی تمامی Cuda Threadهای کلون می شود و این نخ‌ها از طریق Cuda Pipelineها شروع به اجرا می‌کنند. هر پایپ‌لاین توانایی اجرای تردینگ تا ۳۲ مرحله را دارد. این اجازه می‌دهد که تمامی منابع پردازشی آن به مفیدی استفاده شوند. وقتی یک Warp از Cuda Threadها آماده‌ی ساخته شدن است، از ۳۲ پایپ‌لاین کودا استفاده می‌کند و روی آن‌ها اجرا می‌شود. این باعث می‌شود تا آن ۳۲ پایپ‌لاین کودا به مثابه یک تیم عمل کنند. پس یک Cuda Core یک پایپ‌لاین است که برای مپ‌کردن تعدادی Cuda Thread بر Warp ها به صورت سنکرون استفاده می‌شود که بتوان موازی‌سازی را به خوبی انجام داد.
بله. این قابلیت وجود دارد و از نسخه‌ی ۸ کودا اضافه شده است.
/https://devblogs.nvidia.com/mixed-precision-programming-cuda-8

۴. یک Tensor Core یک هسته‌ی پردازشی‌ست که انواع خاصی از محاسبات ماتریسی را انجام می‌دهد که برای یادگیری عمیق و High Performance Computing مناسب است. Tensor Coreها یک دستور تلفیقی از ضرب و جمع را انجام می‌دهند که ۲ ماتریس ۴ در ۴ با نوع داده‌ی Floating-Point 16bit می‌گیرند و به عنوان خروجی جواب ضرب ماتریسی آن‌ها را به صورت یک ماتریس ۴ در ۴ با نوع داده‌ی FP-16 یا FP-32 خروجی می‌دهند. به نظر می‌رسد که نمی‌شود از Tensor Core ها به صورت مستقیم در کد استفاده کرد و برای انجام instruction ها توسط خود درایور گرافیکی انتخاب می‌شوند.

۵. بله این امکان وجود دارد. در مواردی که عملیات‌های موازی‌شونده‌ی بزرگ و کوچک داریم می‌توان عملیات‌های بزرگ را در توابعی به کودا واگذار کرد تا در GPU اجرا شوند و عملیات‌های کوچک‌تر را روی CPU و با استفاده از OpenMP موازی کرد.

۶.

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

#include <stdio.h>

cudaError_t printWithCuda();

__global__ void addKernel()
{
    printf("Hello CUDA I'm thread %d from block %d \n", threadIdx.x, blockIdx.x);
}

int main()
{
    cudaError_t cudaStatus = printWithCuda();
    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;
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t printWithCuda()
{

    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?");
        goto Error;
    }

    // Launch a kernel on the GPU with one thread for each element.
    addKernel<<<3, 3>>>();

    // Check for any errors launching the kernel
```

```
Microsoft Visual Studio Debug Console
Hello CUDA I'm thread 0 from block 0
Hello CUDA I'm thread 1 from block 0
Hello CUDA I'm thread 2 from block 0
Hello CUDA I'm thread 0 from block 1
Hello CUDA I'm thread 1 from block 1
Hello CUDA I'm thread 2 from block 1
Hello CUDA I'm thread 0 from block 2
Hello CUDA I'm thread 1 from block 2
Hello CUDA I'm thread 2 from block 2

C:\Users\TheRe\source\repos\Cuda-GRID-NOT-GRID\x
64\Release\Cuda-GRID-NOT-GRID.exe (process 10944
) exited with code 0.
To automatically close the console when debuggin
g stops, enable Tools->Options->Debugging->Autom
atically close the console when debugging stops.
Press any key to close this window . . .
```

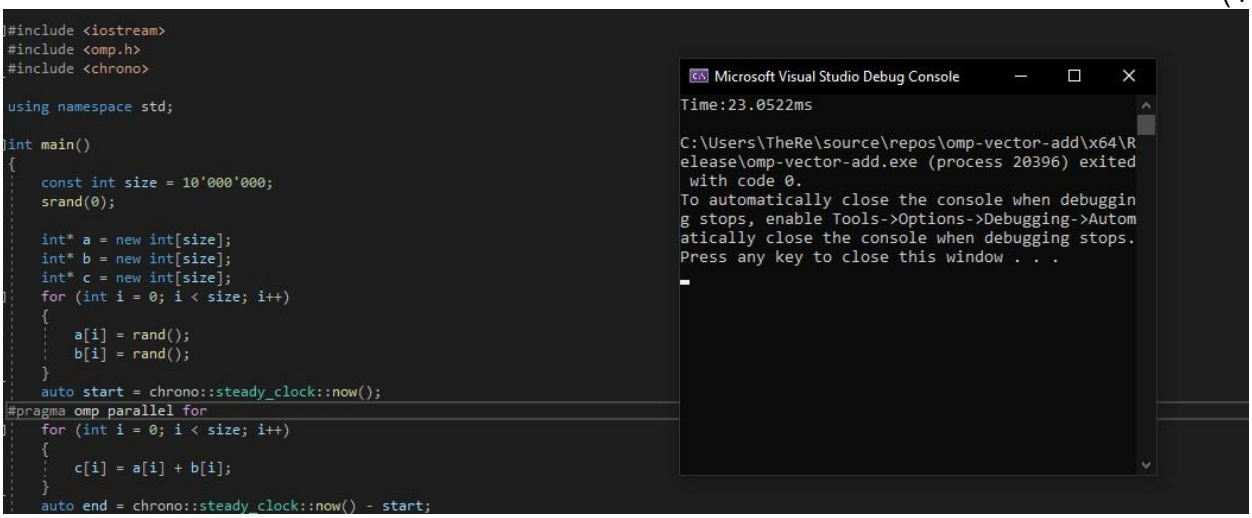
۷.

الف)

```cpp
#include <iostream>
#include <omp.h>
#include <chrono>

using namespace std;

int main()
{
    const int size = 10'000'000;
    srand(0);

    int* a = new int[size];
    int* b = new int[size];
    int* c = new int[size];
    for (int i = 0; i < size; i++)
    {
        a[i] = rand();
        b[i] = rand();
    }
    auto start = chrono::steady_clock::now();
//#pragma omp parallel for
    for (int i = 0; i < size; i++)
    {
        c[i] = a[i] + b[i];
    }
    auto end = chrono::steady_clock::now() - start;
    cout << "Time:" << chrono::duration<double, milli>(end).count() << "ms" << endl;
```

```
Microsoft Visual Studio Debug Console
Time:24.5171ms

C:\Users\TheRe\source\repos\omp-vector-add\x64\R
elease\omp-vector-add.exe (process 15976) exited
 with code 0.
To automatically close the console when debuggin
g stops, enable Tools->Options->Debugging->Autom
atically close the console when debugging stops.
Press any key to close this window . . .
```

ب)



```cpp
#include <iostream>
#include <omp.h>
#include <chrono>

using namespace std;

int main()
{
    const int size = 10'000'000;
    srand(0);

    int* a = new int[size];
    int* b = new int[size];
    int* c = new int[size];
    for (int i = 0; i < size; i++)
    {
        a[i] = rand();
        b[i] = rand();
    }
    auto start = chrono::steady_clock::now();
#pragma omp parallel for
    for (int i = 0; i < size; i++)
    {
        c[i] = a[i] + b[i];
    }
    auto end = chrono::steady_clock::now() - start;
```

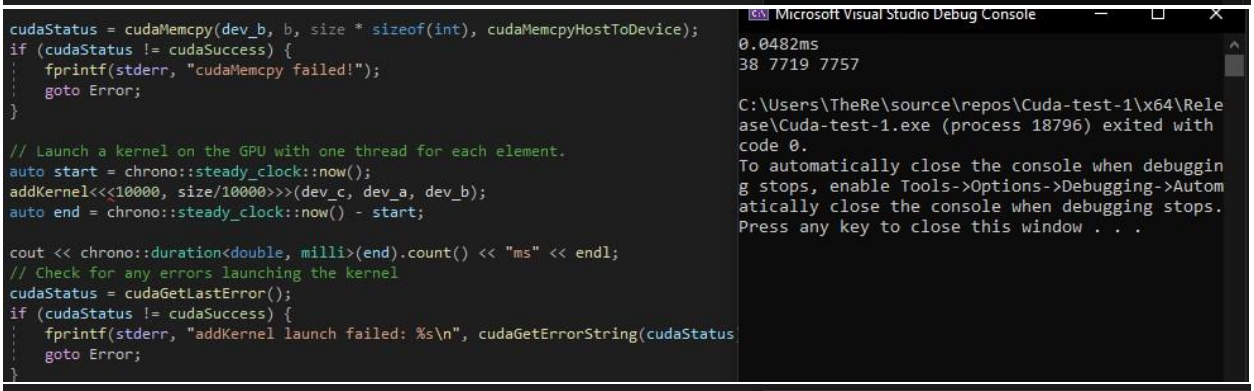
```
Microsoft Visual Studio Debug Console
Time:23.0522ms

C:\Users\TheRe\source\repos\omp-vector-add\x64\R
elease\omp-vector-add.exe (process 20396) exited
 with code 0.
To automatically close the console when debuggin
g stops, enable Tools->Options->Debugging->Autom
atically close the console when debugging stops.
Press any key to close this window . . .
```

پ)

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

#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <chrono>

using namespace std;

cudaError_t addWithCuda(int *c, int *a, int *b, unsigned int size);

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

int main()
{
    const unsigned int size = 10'000'000;
    //int a[size], b[size], c[size];
    int* a = new int[size];
    int* b = new int[size];
    int* c = new int[size];
    srand(0);
    for (int i = 0; i < size; i++)
    {
        a[i] = rand();
        b[i] = rand();
    }

    // Add vectors in parallel.
    cudaError_t cudaStatus = addWithCuda(c, a, b, size);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
    }

    cout << a[0] << ' ' << b[0] << ' ' << c[0] << endl;
    // 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;
    }
```

```
cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Launch a kernel on the GPU with one thread for each element.
auto start = chrono::steady_clock::now();
addKernel<<<10'000'000, size/10'000'000>>>(dev_c, dev_a, dev_b);
auto end = chrono::steady_clock::now() - start;

cout << chrono::duration<double, milli>(end).count() << "ms" << endl;
// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
    goto Error;
}
```

```
Microsoft Visual Studio Debug Console
0.0533ms
38 7719 7757

C:\Users\TheRe\source\repos\Cuda-test-1\x64\Release\Cuda-test-1.exe (process 12892) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

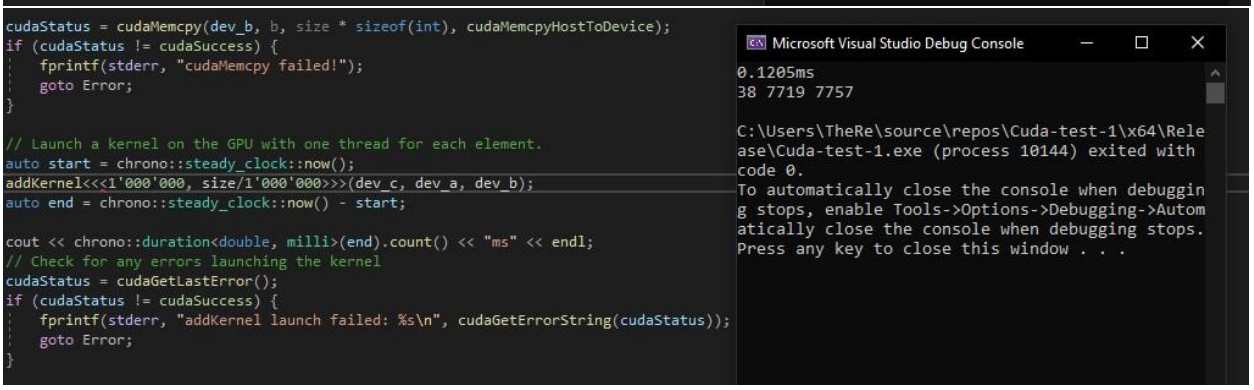


```
cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Launch a kernel on the GPU with one thread for each element.
auto start = chrono::steady_clock::now();
addKernel<<<10000, size/10000>>>(dev_c, dev_a, dev_b);
auto end = chrono::steady_clock::now() - start;

cout << chrono::duration<double, milli>(end).count() << "ms" << endl;
// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus
    goto Error;
}
```

```
Microsoft Visual Studio Debug Console
0.0482ms
38 7719 7757

C:\Users\TheRe\source\repos\Cuda-test-1\x64\Release\Cuda-test-1.exe (process 18796) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

```
cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Launch a kernel on the GPU with one thread for each element.
auto start = chrono::steady_clock::now();
addKernel<<<100'000, size/100'000>>>(dev_c, dev_a, dev_b);
auto end = chrono::steady_clock::now() - start;

cout << chrono::duration<double, milli>(end).count() << "ms" << endl;
// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
    goto Error;
}
```

```
Microsoft Visual Studio Debug Console
0.0362ms
38 7719 7757

C:\Users\TheRe\source\repos\Cuda-test-1\x64\Release\Cuda-test-1.exe (process 10716) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```



```
cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

// Launch a kernel on the GPU with one thread for each element.
auto start = chrono::steady_clock::now();
addKernel<<<1'000'000, size/1'000'000>>>(dev_c, dev_a, dev_b);
auto end = chrono::steady_clock::now() - start;

cout << chrono::duration<double, milli>(end).count() << "ms" << endl;
// Check for any errors launching the kernel
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
    fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
    goto Error;
}
```

```
Microsoft Visual Studio Debug Console
0.1205ms
38 7719 7757

C:\Users\TheRe\source\repos\Cuda-test-1\x64\Release\Cuda-test-1.exe (process 10144) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

اندازه‌ی grid بزرگ‌تر به نظر می‌رسد که نتیجه‌ی بهتری داشته‌باشد. به علت این‌که اردر عملیاتی همه‌ی تردها یکسان است و سوییچ‌کردن بین ۲ بلاک پر از نخ سریع‌تر انجام می‌شود تا انجام دادن تک تک سوییچ‌ها بین Warpهای یک بلاک.

ت) تعداد نخ‌کم‌تر برای این مساله مناسب‌تر است زیرا المان‌ها پشت سر هم هستند و تردها می‌توانند با استفاده از کش مشترک به نتیجه‌ی سریع‌تری برسند. نتایج برای ۱۰۰ هزار بلاک برای تعداد نصف و یک دهم نخ حالت عادی:

```
cudaError_t addWithCuda(int *c, int *a, int *b, unsigned int size);

__global__ void addKernel(int *c, int *a,  int *b)
{
    int x = 2;
    int i = (blockIdx.x * blockDim.x + threadIdx.x) * x;
    for (int j = i; j < i + x; j++)
    {
        c[j] = a[j] + b[j];
    }
```

```
Microsoft Visual Studio Debug Console
0.0673ms
cudaDeviceSynchronize returned error code 700 af
ter launching addKernel!
addWithCuda failed!
C:\Users\TheRe\source\repos\Cuda-test-1\x64\Rele
ase\Cuda-test-1.exe (process 14172) exited with
code 1.
```

```
cudaError_t addWithCuda(int *c, int *a, int *b, unsigned int size);

__global__ void addKernel(int *c, int *a,  int *b)
{
    int x = 10;
    int i = (blockIdx.x * blockDim.x + threadIdx.x) * x;
    for (int j = i; j < i + x; j++)
    {
        c[j] = a[j] + b[j];
    }
}
```

```
Microsoft Visual Studio Debug Console
0.0469ms
cudaDeviceSynchronize returned error code 700 af
ter launching addKernel!
addWithCuda failed!
C:\Users\TheRe\source\repos\Cuda-test-1\x64\Rele
ase\Cuda-test-1.exe (process 13124) exited with
code 1.
To automatically close the console when debuggin
```

دو عامل مطرح است. یکی این‌که بهتر است تعداد نخ‌های به ازای هر بلاک یک ضریب صحیح از سایز Warp (=32) باشند. یکی دیگر این‌که هر SM باید به اندازه‌ای Warp فعال داشته باشد که عمل‌کرد این‌ها تاخیر دسترسی‌های حافظه و خط لوله‌های دستور را بپوشاند که این امر به چگونگی پیاده‌سازی خط لوله، efficiencyی آن و میزان hierarchy حافظه در معماری آن سخت‌افزار باشد.

۸. الف)
در حالی که PTX یک زبان Pseudo-Assembly است و برای همه‌ی سخت‌افزارها مشترک، SASS = Streaming Assembler زبان اسمبلی و isaی native خاص هر سخت‌افزار است که به نسبت معماری سخت‌افزارهای مختلف متغیر است.

ب) در کودا همه‌ی پارامترها باید توسط reference به توابع پاس داده‌شوند، اما به این دلیل که در زبان C خالص reference نداریم، باید آدرس متغیری که می‌خواهیم اطلاعات در آن ذخیره شود را ارسال کنیم. به همین دلیل وقتی می‌خواهیم یک پوینتر را به شکل reference برای یک تابع ارسال کنیم، باید پوینتر به یک پوینتر را به تابع بفرستیم تا بتواند پوینتر مدنظر ما را تغییر دهد.