"بسمه تعالی"

**گزارش آزمایش پنجم برنامه نویسی چندهسته‌ای – آیدا احمدی پارسا – ۹۹۲۳۰۰۳ – زهرا لطیفی – ۹۹۲۳۰۶۹**

## گام اول

کد `deviceQuery.cu` داده شده را اجرا کردیم. مشخصات `device` را در شکل زیر مشاهده می‌کنیم:

```
There is 1 device supporting CUDA

Device 0: "NVIDIA GeForce 930MX"
  Major revision number:                         5
  Minor revision number:                         0
  Total amount of global memory:                 2147352576 bytes
  Number of multiprocessors:                     3
  Number of cores:                               24
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.02 GHz
  Concurrent copy and execution:                 Yes

TEST PASSED
```

به بررسی موارد ذکر شده در این گزارش می‌پردازیم.

`There is 1 device supporting CUDA`: نشان می‌دهد که تنها یک `device` سازگار با CUDA شناسایی شده است.

`"Device 0: "NVIDIA Geforce 930MX"`: نام `device` سازگار با CUDA که اولین دستگاه لیست شده است.

`Major revision number`: شماره نسخه اصلی قابلیت محاسبه که مشخصات کلی و مشخصاتی را که GPU پشتیبانی می‌کند مشخص می‌کند.

`Minor revision number`: شماره نسخه جزئی قابلیت محاسبه که مشخصات دقیق تری را ارائه می‌دهد.

`Total amount of global memory`: اندازه کل حافظه GPU موجود برای ذخیره داده‌ها.

`Number of multiprocessors`: تعداد واحدهای چندپردازنده در GPU که می‌توانند چندین عملیات را به طور همزمان اجرا کنند.

`Number of cores`: تعداد کل هسته‌های پردازشی در GPU.

`Total amount of constant memory`: اندازه حافظه‌ای که در طول مدت اجرای یک کرنل ثابت است و روی پردازنده گرافیکی `cach` ذخیره می‌شود.

`Total amount of shared memory per block`: مقدار حافظه مشترک بین نخ‌های یک بلوک در طول اجرای کرنل.

Total number of registers available per block: تعداد رجیسترهایی که نخ‌های درون یک بلوک می‌توانند استفاده کنند.
Warp size: تعداد نخ‌هایی که دستورالعمل‌ها را در مرحله به صورت موازی اجرا می‌کنند.
Maximum number of threads per block: حداکثر تعداد نخ‌هایی که می‌توان به صورت موازی در یک بلوک اجرا کرد.
Maximum sizes of each dimension of a block and ...grid: حداکثر ابعاد برای بلوک ها و شبکه‌ها که برای ساختار سلسله مراتب نخ‌ها در کرنل‌ها مهم هستند.
Maximum memory pitch: حداکثر اندازه گام تخصیص حافظه، که فاصله بین ردیف‌های متوالی در یک آرایه دو بعدی است.
Texture Alignment: تراز مورد نیاز برای Textureهای مورد استفاده در GPU.
Clock rate: سرعت عملکرد هسته پردازنده گرافیکی.
Concurrent copy and execution: نشان می‌دهد که آیا GPU می‌تواند انتقال داده را همزمان با اجرای کرنل انجام دهد یا خیر.
TEST PASSED: نشان دهنده موفقیت آمیز بودن تست برای قابلیت‌های GPU است.

## گام دوم

در این گام برنامه جمع دو بردار را در حالت سریال بر روی CPU اجرا کردیم. اندازه دو بردار کم و برابر با ۱۰۲۴ است پس زمان اجرا در حالت سریال به علت کوچک بودن برابر صفر گزارش شد.

```
Averaged Elapsed Time : 0.0000000000000000000
E:\uni\Cuda\lab5_4\x64\Release\lab5_4.exe (process 17952) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options-
le when debugging stops.
Press any key to close this window . . .
```

تابع addWithCuda را عینا طبق دستورکار پیاده‌سازی کرده و محاسبات را بر روی GPU بردیم.

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

double start_time, elapsed_time;
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];

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;
```

```
    cudaStatus = cudaSetDevice(0);
    if (cudaStatus != cudaSuccess) {
        printf("cudaSetDevice failed! Do you have a CUDA-capable GPU
installed?");
    }

    cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        printf("cudaMalloc failed!");
    }

    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        printf("cudaMalloc failed!");
    }

    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        printf("cudaMalloc failed!");
    }

    cudaStatus = cudaMemcpy(dev_a, a, size * sizeof(int),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        printf("cudaMemcpy failed!");
    }

    cudaStatus = cudaMemcpy(dev_b, b, size * sizeof(int),
cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess) {
        printf("cudaMemcpy failed!");
    }

    addKernel << <1, 1024 >> > (dev_c, dev_a, dev_b);

    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        printf("addKernel launch failed: %s\n",
cudaGetErrorString(cudaStatus));
    }

    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        printf("cudaDeviceSynchronize returned error code %d after launching
addKernel!\n", cudaStatus);
    }
```

```
    cudaStatus = cudaMemcpy(c, dev_c, size * sizeof(int),
cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        printf("cudaMemcpy failed!");
    }

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

    return cudaStatus;
}
```

زمان اجرا به ۰.۰۰۰۱ ثانیه رسید. دلیل افزایش زمان اجرا، بالا بودن سربارهای بردن محاسبات برروی GPU نسبت به اندازه مسئله (۱۰۲۴) است.

```
Averaged Elapsed Time : 0.00010000000000000000
E:\uni\Cuda\lab5_4\x64\Release\lab5_4.exe (process 11728) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options
le when debugging stops.
Press any key to close this window . . .
```

### گام سوم

حال فرض کردیم ۱۰۲۴×n عنصر داریم. در این شرایط دو راه از راه‌های پیش رو عبارت‌اند از:

۱. هر نخ n جمع انجام دهد.
۲. n بلوک ۱۰۲۴تایی اجرا کنیم.

این دو روش را پیاده‌سازی و زمان اجرا را برای nهای به اندازه کافی بزرگ مقایسه کردیم. ($n = 2^{14} - 2^{15} - 2^{16}$) متغیرهای NUM_BLOCKS و NUM_THREADS و ELEMENTS_PER_THREAD را اضافه کردیم و در هر روش مقداردهی ویژه آن روش را برای این متغیرها انجام دادیم. در کرنل به این صورت تغییر ایجاد کردیم که index شروع برای هر نخ را بر اساس index بلوک و index دقیق نخ محاسبه می‌کند. حلقه عناصر «element_per_thread» را پیمایش می‌کند (یا کمتر اگر به انتهای آرایه برسد) و حاصل جمع عناصر مربوطه از «a» و «b» را در «c» اضافه می‌کند.

**متغیر** NUM_THREADS.x برابر ۱۰۲۴ خواهد بود و متغیر NUM_BLOCKS تعداد بلوک های مورد نیاز را بر اساس اندازه کل آرایه‌ها و تعداد نخ‌های مورد نظر در هر بلوک محاسبه می‌کند که از طریق رابطه size+(NUM_THREADS.x*ELEMENTS_PER_THREAD)-1)/(NUM_THREADS.x*ELEMENTS_PER_THREAD) بدست می‌آید.
ابتدا کد را با هر سه اندازه ورودی ذکر شده به صورت سریال ران می‌کنیم. خروجی $n=2^{16}$ به شرح زیر است:

```
Averaged Elapsed Time : 0.14540000000000000147
```

#### روش اول

در این روش متغیر ELEMENTS_PER_THREAD را برابر با (1<<(n+10)/1024 قرار دادیم زیرا پردازش n عنصر توسط هر نخ مد نظر بود.

```
__global__ void addKernel(int *c, const int *a, const int *b, const int
vectorSize, const int elements_per_thread)
{
    int start = (blockIdx.x * blockDim.x + threadIdx.x) * elements_per_thread;
    for (int i = start; i - start < elements_per_thread && (i < vectorSize);
i++) {
        c[i] = a[i] + b[i];
    }
}
int main()
{
    const int vectorSize = 1 << 26;
      ...
}
cudaError_t addWithCuda(int *c, const int *a, const int *b, unsigned int size)
{
 ...
    int ELEMENTS_PER_THREAD = (1 << 26) /1024;
    dim3 NUM_THREADS(1024, 1, 1);   // Threads per block
    dim3 NUM_BLOCKS((size + (NUM_THREADS.x * ELEMENTS_PER_THREAD) - 1) /
(NUM_THREADS.x * ELEMENTS_PER_THREAD), 1, 1);

    printf("elements per thread: %d, threads per blocks: %d, blocks: %d\n",
ELEMENTS_PER_THREAD, NUM_THREADS.x, NUM_BLOCKS.x);

    start_time = clock();
    addKernel << <NUM_BLOCKS, NUM_THREADS >> > (dev_c, dev_a, dev_b, size,
ELEMENTS_PER_THREAD);
 ...
}
```

دیدیم که در روش اول تنها یک بلوک نخ ۱۰۲۴ تایی داریم و هر نخ ۶۵۵۳۶ عنصر را پردازش می‌کند. این روش به دلیل وجود شباهت در نحوه محاسبه به محاسبات روی CPU، برروی هسته‌های کوچک و ضعیف GPU به خوبی جواب نمی‌دهد و اجرای آن کندتر از حالت سریال است. درواقع در این روش موازی‌سازی درشت دانه انجام می‌دهیم.

```
elements per thread: 65536, threads per blocks: 1024, blocks: 1
elements per thread: 65536, threads per blocks: 1024, blocks: 1
elements per thread: 65536, threads per blocks: 1024, blocks: 1
elements per thread: 65536, threads per blocks: 1024, blocks: 1
elements per thread: 65536, threads per blocks: 1024, blocks: 1
elements per thread: 65536, threads per blocks: 1024, blocks: 1
elements per thread: 65536, threads per blocks: 1024, blocks: 1
elements per thread: 65536, threads per blocks: 1024, blocks: 1
elements per thread: 65536, threads per blocks: 1024, blocks: 1
elements per thread: 65536, threads per blocks: 1024, blocks: 1
Averaged Elapsed Time : 1.4509000000000007851
E:\uni\Cuda\lab5_4\x64\Release\lab5_4.exe (process 11180) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options-
le when debugging stops.
Press any key to close this window . . .
```

**روش دوم**

در روش دوم متغیر ELEMENTS_PER_THREAD را برابر یک قرار دادیم زیرا در این روش هر نخ باید تنها یک حاصل جمع را محاسبه می‌کرد. درواقع در این روش موازی‌سازی ریز دانه انجام می‌دهیم و Throughput بالایی داریم.

```
    int ELEMENTS_PER_THREAD = 1;
```

(مابقی کد مشابه روش قبل است.)

در این روش هر نخ تنها یک عنصر را پردازش می‌کند اما تعداد بلوک‌های ۱۰۲۴ تایی ۶۵۵۳۶ است. این مسئله یعنی دادن کارهای کوچک به هر نخ و زیاد کردن تعداد نخ‌ها با افزایش تعداد بلوک‌ها باعث استفاده حداکثری از قدرت GPU می‌شود و زمان اجرا را کاهش می‌دهد.

```
elements per thread: 1, threads per blocks: 1024, blocks: 65536
elements per thread: 1, threads per blocks: 1024, blocks: 65536
elements per thread: 1, threads per blocks: 1024, blocks: 65536
elements per thread: 1, threads per blocks: 1024, blocks: 65536
elements per thread: 1, threads per blocks: 1024, blocks: 65536
elements per thread: 1, threads per blocks: 1024, blocks: 65536
elements per thread: 1, threads per blocks: 1024, blocks: 65536
elements per thread: 1, threads per blocks: 1024, blocks: 65536
elements per thread: 1, threads per blocks: 1024, blocks: 65536
elements per thread: 1, threads per blocks: 1024, blocks: 65536
Averaged Elapsed Time : 0.06060000000000000109
E:\uni\Cuda\lab5_4\x64\Release\lab5_4.exe (process 15248) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options-
le when debugging stops.
Press any key to close this window . . .
```

زمان‌های اجرا (ثانیه) به ازای اندازه ورودی‌های مختلف در هر روش به ازای ۱۰ بار تکرار و میانگین‌گیری در جدول زیر آمده است. تسریع هم با میانگین‌گیری هر سه ستون محاسبه شده است.

| روش | اندازه بردار ورودی | | | تسریع |
|---|---|---|---|---|
| | $2^{24}$ | $2^{25}$ | $2^{26}$ | |
| سریال | 0.0387 | 0.0714 | 0.1450 | |
| روش ۱ | 0.0727 | 0.1413 | 1.4506 | 0.3792 |
| روش ۲ | 0.0161 | 0.0311 | 0.0605 | 2.3654 |

## گام چهارم

در این بخش خواسته شده بود دسته‌بندی نخ‌ها در warp، block، grid را بررسی کنیم. پس kernelای نوشتیم که نخی که آن را اجرا می‌کند، شماره warp خود، شماره blockای که در آن قرار دارد و اندیس سراسری خودش را محاسبه و اعلام کند. توجه داریم که چون امکان استفاده از printf در کرنل وجود ندارد، هر نخ موارد خواسته شده را در ۳ آرایه سراسری می‌نویسد. سپس در سمت host این ۳ آرایه خوانده و چاپ می‌شوند.

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

// Define the gpuErrchk macro
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
```

```
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort
= true)
    {
        if (code != cudaSuccess)
        {
            fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code),
file, line);
            if (abort) exit(code);
        }
    }

__global__ void calcThreadKernel(int *block, int *warp, int *local_index) {
    int bd = blockDim.x;
    int bx = blockIdx.x;
    int tx = threadIdx.x;

    int global_idx = bd * bx + tx;
    block[global_idx] = bx;
    warp[global_idx] = tx / warpSize;
    local_index[global_idx] = tx;
}

int main(int argc, char **argv) {
    dim3 NUM_THREADS(64, 1, 1);
    dim3 NUM_BLOCKS(2, 1, 1);

    int size = NUM_THREADS.x * NUM_BLOCKS.x;
    int *block, *warp, *local_index;
    int *block_host, *warp_host, *local_index_host;

    gpuErrchk(cudaMalloc((void **)&block, size * sizeof(int)));
    gpuErrchk(cudaMalloc((void **)&warp, size * sizeof(int)));
    gpuErrchk(cudaMalloc((void **)&local_index, size * sizeof(int)));

    calcThreadKernel << <NUM_BLOCKS, NUM_THREADS >> > (block, warp,
local_index);
    gpuErrchk(cudaGetLastError());

    gpuErrchk(cudaDeviceSynchronize());

    block_host = (int*)malloc(size * sizeof(int));
    warp_host = (int*)malloc(size * sizeof(int));
    local_index_host = (int*)malloc(size * sizeof(int));

    gpuErrchk(cudaMemcpy(block_host, block, size * sizeof(int),
cudaMemcpyDeviceToHost));
    gpuErrchk(cudaMemcpy(warp_host, warp, size * sizeof(int),
cudaMemcpyDeviceToHost));
```

```
    gpuErrchk(cudaMemcpy(local_index_host, local_index, size * sizeof(int),
cudaMemcpyDeviceToHost));

    for (int i = 0; i < size; i++) {
        printf("Calculated Thread: %d,\tBlock: %d,\tWarp: %d,\tThread: %d\n",
i, block_host[i], warp_host[i], local_index_host[i]);
    }

    gpuErrchk(cudaFree(block));
    gpuErrchk(cudaFree(warp));
    gpuErrchk(cudaFree(local_index));

    free(block_host);
    free(warp_host);
    free(local_index_host);

    return 0;
}
```

gpuErrchk برای کنترل خطای CUDA تعریف شده است. یک فراخوانی از تابع CUDA(`ans`) را می‌گیرد و بررسی می‌کند که آیا خطایی برمی‌گرداند. اگر خطایی رخ دهد، یک پیام خطا همراه با نام فایل و شماره خط چاپ می‌کند.

تابع calcThreadKernel یک کرنل CUDA است که روی GPU اجرا می‌شود. به سه نشانگر int به عنوان ورودی نیاز دارد block، warp و local_index.

Bd نشان دهنده بعد بلوک (تعداد نخ‌ها در هر بلوک) است. bx نمایانگر اندیس بلوک (در داخل grid) است و tx نمایانگر thread index(در داخل بلوک) است. global_idx هم اندیس سراسری نخ را محاسبه می‌کند.

نهایتا کرنل مقادیر block[global_idx]، warp[global_idx] و local_index[global_idx] را اختصاص می‌دهد.

در تابع main ابعاد نخ و بلوک را مقداردهی اولیه می‌کنیم، طبق خواسته صورت سوال NUM_THREADS روی ۶۴ نخ و NUM_BLOCKS روی ۲ بلوک تنظیم شده است.

برای سه آرایه سمت device یعنی block، warp و local_index با استفاده از cudaMalloc حافظه تخصیص داده‌ایم.

سپس کرنل calcThreadKernel با ابعاد grid و بلوک مشخص شده فراخوانی شده است.

cudaGetLastError() هرگونه خطای فراخوانی کرنل را بررسی می‌کند و cudaDeviceSynchronize() تضمین می‌کند که تمام عملیات GPU قبل از ادامه تکمیل شده است.

برای سه آرایه سمت host یعنی block_host ، warp_host و local_index_host با استفاده از malloc حافظه تخصیص دادیم.

نهایتا نتایج با استفاده از «cudaMemcpy» از GPU به CPU کپی می‌شوند. و چاپ می‌شوند.

در آخر حافظه سمت device و حافظه سمت host هر دو آزاد می‌شوند.

بخشی از خروجی این کد در ادامه آورده شده است. هر خط توسط یک نخ محاسبه شده Thread Calculated اندیس سراسری نخ، Block شماره بلوک آن نخ، Warp شماره warp آن نخ و Thread اندیس محلی نخ است.

```
Calculated Thread: 12,  Block: 0,       Warp: 0,        Thread: 12
Calculated Thread: 13,  Block: 0,       Warp: 0,        Thread: 13
Calculated Thread: 14,  Block: 0,       Warp: 0,        Thread: 14
Calculated Thread: 15,  Block: 0,       Warp: 0,        Thread: 15
Calculated Thread: 16,  Block: 0,       Warp: 0,        Thread: 16
Calculated Thread: 17,  Block: 0,       Warp: 0,        Thread: 17
Calculated Thread: 18,  Block: 0,       Warp: 0,        Thread: 18
Calculated Thread: 19,  Block: 0,       Warp: 0,        Thread: 19
Calculated Thread: 20,  Block: 0,       Warp: 0,        Thread: 20
Calculated Thread: 21,  Block: 0,       Warp: 0,        Thread: 21
Calculated Thread: 22,  Block: 0,       Warp: 0,        Thread: 22
Calculated Thread: 23,  Block: 0,       Warp: 0,        Thread: 23
Calculated Thread: 24,  Block: 0,       Warp: 0,        Thread: 24
Calculated Thread: 25,  Block: 0,       Warp: 0,        Thread: 25
Calculated Thread: 26,  Block: 0,       Warp: 0,        Thread: 26
Calculated Thread: 27,  Block: 0,       Warp: 0,        Thread: 27
Calculated Thread: 28,  Block: 0,       Warp: 0,        Thread: 28
Calculated Thread: 29,  Block: 0,       Warp: 0,        Thread: 29
Calculated Thread: 30,  Block: 0,       Warp: 0,        Thread: 30
Calculated Thread: 31,  Block: 0,       Warp: 0,        Thread: 31
Calculated Thread: 32,  Block: 0,       Warp: 1,        Thread: 32
Calculated Thread: 33,  Block: 0,       Warp: 1,        Thread: 33
Calculated Thread: 34,  Block: 0,       Warp: 1,        Thread: 34
Calculated Thread: 35,  Block: 0,       Warp: 1,        Thread: 35
Calculated Thread: 36,  Block: 0,       Warp: 1,        Thread: 36
Calculated Thread: 37,  Block: 0,       Warp: 1,        Thread: 37
Calculated Thread: 38,  Block: 0,       Warp: 1,        Thread: 38
Calculated Thread: 39,  Block: 0,       Warp: 1,        Thread: 39
Calculated Thread: 40,  Block: 0,       Warp: 1,        Thread: 40
Calculated Thread: 41,  Block: 0,       Warp: 1,        Thread: 41
Calculated Thread: 42,  Block: 0,       Warp: 1,        Thread: 42
Calculated Thread: 43,  Block: 0,       Warp: 1,        Thread: 43
Calculated Thread: 44,  Block: 0,       Warp: 1,        Thread: 44
Calculated Thread: 45,  Block: 0,       Warp: 1,        Thread: 45
Calculated Thread: 46,  Block: 0,       Warp: 1,        Thread: 46
Calculated Thread: 47,  Block: 0,       Warp: 1,        Thread: 47
Calculated Thread: 48,  Block: 0,       Warp: 1,        Thread: 48
Calculated Thread: 49,  Block: 0,       Warp: 1,        Thread: 49
Calculated Thread: 50,  Block: 0,       Warp: 1,        Thread: 50
Calculated Thread: 51,  Block: 0,       Warp: 1,        Thread: 51
Calculated Thread: 52,  Block: 0,       Warp: 1,        Thread: 52
Calculated Thread: 53,  Block: 0,       Warp: 1,        Thread: 53
Calculated Thread: 54,  Block: 0,       Warp: 1,        Thread: 54
Calculated Thread: 55,  Block: 0,       Warp: 1,        Thread: 55
```