درس برنامه نویسی چند هسته ای

آزمایش هفتم – هیستوگرام

آرش حریرپوش
9731505

آرش حریرپوش
9731505

❖ قسمت اول

کد این قسمت برابر است با:

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

#include <stdio.h>
#include<stdlib.h>
#include <omp.h>
#include<iostream>
#define MAX_HISTORGRAM_NUMBER 10000
#define ARRAY_SIZE 102400000

#define CHUNK_SIZE 100
#define THREAD_COUNT 1024
#define SCALER 80
cudaError_t histogramWithCuda(int *a, unsigned long long int *c);

__global__ void histogramKernelSingle(unsigned long long int *c, int *a)
{
	unsigned long long int worker =  blockIdx.x*blockDim.x + threadIdx.x;
	unsigned long long int start = worker * CHUNK_SIZE;
	unsigned long long int end = start + CHUNK_SIZE;
	for (int ex = 0; ex < SCALER; ex++)
		for (long long int i = start; i < end; i++)
		{
			if (i < ARRAY_SIZE)
				atomicAdd(&c[a[i]], 1);
			else
			{

				break;
			}
		}

}
int main()
{
		int* a = (int*)malloc(sizeof(int)*ARRAY_SIZE);
		unsigned long long int* c = (unsigned long long int*)malloc(sizeof(unsigned
long long int)*MAX_HISTORGRAM_NUMBER);
		for (unsigned long long i = 0; i < ARRAY_SIZE;i++)
			a[i] = rand() % MAX_HISTORGRAM_NUMBER;
		for (unsigned long long i = 0; i < MAX_HISTORGRAM_NUMBER; i++)
			c[i] = 0;

    // Add vectors in parallel.
		double start_time = omp_get_wtime();
		cudaError_t cudaStatus=histogramWithCuda(a,c);
		double end_time = omp_get_wtime();
		std::cout << end_time - start_time;
   // =
    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;
    }
       unsigned long long int R = 0;
       for (int i = 0; i < MAX_HISTORGRAM_NUMBER; i++)
       {
              R += c[i];
//            printf("%d    ", c[i]);
       }
       printf("\nCORRECT:%ld       ", R/(SCALER));
    return 0;
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t histogramWithCuda(int *a, unsigned long long int *c)
{
    int *dev_a = 0;
       unsigned long long int *dev_c = 0;
    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;
    }

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

    cudaStatus = cudaMalloc((void**)&dev_a, ARRAY_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, ARRAY_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.
       //// BLOCK CALCULATOR HERE
```

```
        ////BLOCK CALCULATOR HERE
        int numBlock = int(ARRAY_SIZE / (THREAD_COUNT*CHUNK_SIZE));
               histogramKernelSingle << <numBlock, THREAD_COUNT>> > (dev_c, dev_a);
    // 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, MAX_HISTORGRAM_NUMBER * sizeof(unsigned long long
int), cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }

Error:
    cudaFree(dev_c);
    cudaFree(dev_a);
    return cudaStatus;
}
```

1. دلیل اثر منفی افزایش اشغال در زمان اجرا چیست؟

چون با افزایش اشتغال منابع اختصاصی هر نخ کاهش می یابد، هر چه اشتغال را افزایش می دهیم تعداد نخ های فعال افزایش می یابد و به همین دلیل رجیستر های کمتری به هر نخ داده می شود؛ همچنین چون ممکن است نخ های مختلف در ریسمان های متفاوت به یک داده نیاز داشته باشند، تاخیر دسترسی به حافظه نیز افزایش می یابد.

2. چگونه می توان با کم نکردن اشغال زمان اجرا را بهبود داد؟

با کوچک کردن اندازه بلوک ها و در نتیجه آن بیشتر شدن تعداد بلوک ها می توان زمان اجرا را با کم نشدن اشغال بهبود داد.

تعداد نخ	8	16	32	256	1024
متغیر SCALER	80	80	80	80	80
تعداد بلوک	128000	64000	32000	4000	1000
اشغال نظری	50.00%	50.00%	50.00%	100.00%	100.00%
اشغال بدست آمده	49.99%	49.99%	49.97%	78.41%	87.13%
زمان اجرای تابع هسته	8.31034	8.32945	8.44408	8.46546	8.46157

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

❖ قسمت دوم

کد این قسمت برابر است با:

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

#include <stdio.h>
#include<stdlib.h>
#include <omp.h>
#include<iostream>
#define MAX_HISTORGRAM_NUMBER 10000
#define ARRAY_SIZE 102400000

#define CHUNK_SIZE 100
#define THREAD_COUNT 1024
#define SCALER 80
cudaError_t histogramWithCuda(int *a, unsigned long long int *c);

__global__ void histogramKernelSingle(unsigned long long int *c, int *a)
{
	unsigned long long int worker = blockIdx.x*blockDim.x + threadIdx.x;
	unsigned long long int start = worker * CHUNK_SIZE;
	unsigned long long int end = start + CHUNK_SIZE;
	for (int ex = 0; ex < SCALER; ex++)
		for (long long int i = start; i < end; i++)
		{
			if (i < ARRAY_SIZE)
				atomicAdd(&c[a[i]], 1);
			else
			{

				break;
			}
		}

}
int main()
{
	int* a;
```

```
	cudaError_t cudaStatus;
	cudaStatus = cudaHostAlloc((void**)&a, sizeof(int)*ARRAY_SIZE,
cudaHostAllocWriteCombined |
		cudaHostAllocMapped);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaHostAlloc failed!");
		return 1;
	}
	unsigned long long int* c;
	cudaStatus = cudaHostAlloc((void**)&c, sizeof(unsigned long long
int)*MAX_HISTORGRAM_NUMBER, cudaHostAllocWriteCombined |
		cudaHostAllocMapped);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaHostAlloc failed!");
		return 1;
	}
	for (unsigned long long i = 0; i < ARRAY_SIZE; i++)
		a[i] = rand() % MAX_HISTORGRAM_NUMBER;
	for (unsigned long long i = 0; i < MAX_HISTORGRAM_NUMBER; i++)
		c[i] = 0;

	// Add vectors in parallel.
	double start_time = omp_get_wtime();
	cudaStatus = histogramWithCuda(a, c);
	double end_time = omp_get_wtime();
	std::cout << end_time - start_time;
	// =
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "addWithCuda failed!");
		return 1;
	}

	unsigned long long int* copy_c = (unsigned long long int*)malloc(sizeof(unsigned
long long int)*MAX_HISTORGRAM_NUMBER);
	for (unsigned long long i = 0; i < MAX_HISTORGRAM_NUMBER; i++)
		copy_c[i] = c[i];
	// 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;
	}

	unsigned long long int R = 0;
	for (int i = 0; i < MAX_HISTORGRAM_NUMBER; i++)
	{
		R += copy_c[i];
		//		printf("%d    ", c[i]);
	}
	printf("\nCORRECT:%ld	", R / (SCALER));

	return 0;
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t histogramWithCuda(int *a, unsigned long long int *c)
{
```

```
	int *dev_a = 0;
	unsigned long long int *dev_c = 0;
	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;
	}

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

	cudaStatus = cudaMalloc((void**)&dev_a, ARRAY_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, ARRAY_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.
	//// BLOCK CALCULATOR HERE


	////BLOCK CALCULATOR HERE
	int numBlock = int(ARRAY_SIZE / (THREAD_COUNT*CHUNK_SIZE));
	histogramKernelSingle << <numBlock, THREAD_COUNT >> > (dev_c, dev_a);
	// 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, MAX_HISTORGRAM_NUMBER * sizeof(unsigned long long int), cudaMemcpyDeviceToHost);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMemcpy failed!");
		goto Error;
	}

Error:
	cudaFree(dev_c);
	cudaFree(dev_a);
	return cudaStatus;
}
```

1. تفاوت حافظه Pageable با حافظه پین شده در چیست؟

در حافظه پین شده قسمتی از حافظه با سیستم paging کار نمی کند و به همین دلیل دسترسی به داده ها سریع تر می شود؛ یعنی حافظه پین شده در RAM باقی می ماند و page دیگری جایگزین آن نمی شود.

2. روش های دیگر تخصیص حافظه ی پین شده را معرفی کنید و تفاوت آن ها را با یکدیگر بیان کنید.

برای تخصیص حافظه پین شده می توان از دو دستور cudaMallocHost و cudaHostAlloc استفاده کرد، هر دو این روش ها زمانی که از flag های پیش فرض cudaHostAlloc استفاده می شود و فراخوانی آن ها بر روی پلتفورم UVA اجرا می گردد.

(منابع: https://stackoverflow.com/questions/35535831/is-there-any-difference-between-cudamallochost-and-cudahostalloc-without-spe و https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/)

3. دلیل دیگر بهبود زمان اجرا در استفاده از این روش چیست؟(به جز حذف حافظه ی پین شده موقت و زمان انتقال داده به آن)

به کمک این روش و تابع cudaMemcpyAsync می توان انتقال داده از host به device و برعکس را به صورت async انجام داد، در این حالت می توان همزمان با ارسال یا دریافت داده ها عملیات محاسباتی دیگری را نیز که به آن داده ها نیازی ندارد به صورت همزمان انجام داد.

تعداد نخ	8	16	32	256	1024
متغیر SCALER	80	80	80	80	80
تعداد بلوک	128000	64000	32000	4000	1000
اشغال نظری	50.00%	50.00%	50.00%	100.00%	100.00%
اشغال بدست آمده	49.99%	49.99%	49.97%	79.13%	86.58%
زمان اجرای تابع هسته	7.41699	7.4547	7.55403	7.57999	7.6087

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

❖ قسمت سوم

کد این قسمت برابر است با:

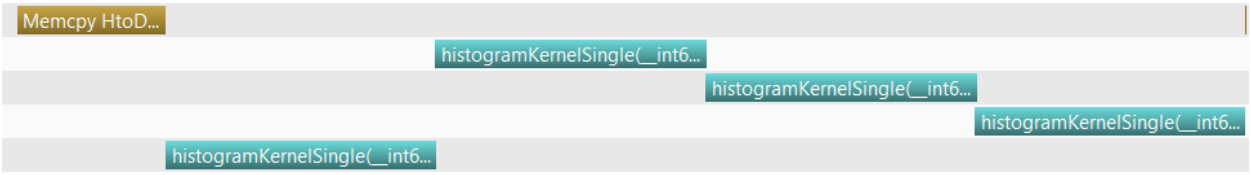
```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include<stdlib.h>
#include <omp.h>
#include<iostream>
#define MAX_HISTORGRAM_NUMBER 10000
#define ARRAY_SIZE 102400000

#define CHUNK_SIZE 100
#define THREAD_COUNT 1024
#define SCALER 20
#define NumOfStreams  4
cudaError_t histogramWithCuda(int *a, unsigned long long int *c);

__global__ void histogramKernelSingle(unsigned long long int *c, int *a)
{
	unsigned long long int worker = blockIdx.x*blockDim.x + threadIdx.x;
	unsigned long long int start = worker * CHUNK_SIZE;
	unsigned long long int end = start + CHUNK_SIZE;
	for (int ex = 0; ex < SCALER/NumOfStreams; ex++)
		for (long long int i = start; i < end; i++)
		{
			if (i < ARRAY_SIZE)
				atomicAdd(&c[a[i]], 1);
			else
			{

				break;
			}
		}

}
int main()
{
```

```
	int* a;
	cudaError_t cudaStatus;
	cudaStatus = cudaHostAlloc((void**)&a, sizeof(int)*ARRAY_SIZE,
cudaHostAllocWriteCombined |
		cudaHostAllocMapped);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaHostAlloc failed!");
		return 1;
	}
	unsigned long long int* c;
	cudaStatus = cudaHostAlloc((void**)&c, sizeof(unsigned long long
int)*MAX_HISTORGRAM_NUMBER, cudaHostAllocWriteCombined |
		cudaHostAllocMapped);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaHostAlloc failed!");
		return 1;
	}
	for (unsigned long long i = 0; i < ARRAY_SIZE; i++)
		a[i] = rand() % MAX_HISTORGRAM_NUMBER;
	for (unsigned long long i = 0; i < MAX_HISTORGRAM_NUMBER; i++)
		c[i] = 0;

	// Add vectors in parallel.
	double start_time = omp_get_wtime();
	cudaStatus = histogramWithCuda(a, c);
	double end_time = omp_get_wtime();
	std::cout << end_time - start_time;
	// =
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "addWithCuda failed!");
		return 1;
	}

	unsigned long long int* copy_c = (unsigned long long int*)malloc(sizeof(unsigned
long long int)*MAX_HISTORGRAM_NUMBER);
	for (unsigned long long i = 0; i < MAX_HISTORGRAM_NUMBER; i++)
		copy_c[i] = c[i];
	// 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;
	}

	unsigned long long int R = 0;
	for (int i = 0; i < MAX_HISTORGRAM_NUMBER; i++)
	{
		R += copy_c[i];
		//		printf("%d    ", c[i]);
	}
	printf("\nCORRECT:%ld	", R / (SCALER));

	return 0;
}

// Helper function for using CUDA to add vectors in parallel.
cudaError_t histogramWithCuda(int *a, unsigned long long int *c)
```

```
{
	int *dev_a = 0;
	unsigned long long int *dev_c = 0;
	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;
	}

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

	cudaStatus = cudaMalloc((void**)&dev_a, ARRAY_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, ARRAY_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.
	//// BLOCK CALCULATOR HERE


	////BLOCK CALCULATOR HERE
	int numBlock = int(ARRAY_SIZE / (THREAD_COUNT*CHUNK_SIZE));
	cudaStream_t streams[NumOfStreams];
	for (int i = 0; i < NumOfStreams; i++) {
		cudaStreamCreate(&streams[i]);
		// launch one worker kernel per stream
		histogramKernelSingle << <numBlock, THREAD_COUNT, 0, streams[i] >> >
(dev_c, dev_a);
	}
	// 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, MAX_HISTORGRAM_NUMBER * sizeof(unsigned long long int), cudaMemcpyDeviceToHost);
	if (cudaStatus != cudaSuccess) {
		fprintf(stderr, "cudaMemcpy failed!");
		goto Error;
	}

Error:
	cudaFree(dev_c);
	cudaFree(dev_a);
	return cudaStatus;
}
```

شکل زیر اجرای هم زمان جریان ها را به ازای 4 جریان و بلوک نخ 32 تایی نمایش می دهد:



1. به نظر شما مسائل مناسب برای استفاده از جریان ها چه ساختاری دارند؟

جریان ها برای مسائلی مناسب است که امکان انجام هم زمان چند کار بر روی پردازنده گرافیکی باشد، یعنی بتوان همزمان از host به device و یا برعکس آن انتقال داده وجود داشته باشد(همچنین می توان در هر دو جهت عنوان شده داده را منتقل کرد) و یا بتوان محاسبات را به کار های کوچک تر تقسیم کرد تا بتوان آن کار های کوچک را بین بلوک های متفاوت برای اجرای همزمان تقسیم کرد، یعنی بجای اینکه یک کرنل تمام سخت افزار موجود در پردازنده گرافیکی را درگیر کند کار آن کرنل را به چندین کرنل کوچک تر تقسیم کرد که به منابع کمتری برای اجرا نیاز دارند، اما به صورت همزمان با یک دیگر کار می کنند.

2. پردازنده ی گرافیکی اولویت را به حفظ حداکثر اشغال می دهد یا اجرای هم زمان حداکثری جریان ها؟

پردازنده گرافیکی اولویت را به حفظ حداکثری اشتغال می دهد و در صورتی که منابع برای اجرای یک کرنل دیگر داشته باشد، یک کرنل دیگر را از جریان دیگری اجرا می کند.

3. آیا با تغییر کامل رویکرد و اختصاص هر جریان به شمردن یک یا چند نوع عضو از آرایه ی اصلی می توان زمان اجرا را بهبود داد؟ جواب خود را با دلیل توضیح دهید. (پیاده سازی موردنیاز برای این سؤال امتیازی است)

بله، به این دلیل که در این روش منابع اختصاصی هر کرنل کاهش می یابد به همین دلیل چند کرنل می توانند همزمان اجرا شوند که همین امر زمان اجرا را بهبود می دهد؛ به عنوان مثال بجای اینکه یک کرنل 1000 بلوک نخ 1024 تایی را اجرا کند چندین کرنل 8 بلوکه با 32 نخ اجرا می شود، در این صورت بلوک و نخ کافی برای اجرای همزمان چندین کرنل با این اندازه بلوک و نخ می تواند همزمان اجرا شود.

تعداد نخ	8	16	32	256	1024	8	16	32	256	1024
تعداد جریان	2	2	2	2	2	4	4	4	4	4
متغیر SCALER	20	20	20	20	20	20	20	20	20	20
تعداد بلوک	128000	64000	32000	32000	1000	128000	64000	32000	32000	1000
اندازه تکه	10	10	10	10	10	5	5	5	5	5
اشغال نظری	50.00%	50.00%	50.00%	100.00%	100.00%	50.00%	50.00%	50.00%	100.00%	100.00%
جریان های هم زمان	1	1	1	1	1	1	1	1	1	1
اشغال بدست آمده	49.99%	49.98%	49.97%	90.14%	97.78%	49.98%	49.98%	49.97%	87.11%	93.25%
زمان اجرای تابع هسته	2.06397	2.07057	2.09409	2.10244	2.10855	2.0596	2.08234	2.09356	2.10168	2.10623

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