Санкт-Петербургский национальный исследовательский университет

информационных технологий, механики и оптики
(Университет ИТМО)

Факультет программной инженерии и компьютерной техники

Кафедра вычислительной техники

Отчет по полупрактике
«OpenCL»

Выполнил
студент 2 курса,
группы P3200
Сапожников Борис
Константинович
Руководитель:
кандидат технических
наук
Перминов Илья
Валентинович

Санкт-Петербург — 2016

1. Реализация кернелов.

Первые три варианта кернела показались мне довольно тривиальными.

```
__kernel void MatrixMultiply0( __global float* a, __global float* b, __global float* c, int Wa, int Wb) {
	int row = get_global_id(0);
	for (int col = 0; col < Wb; col++)
	{
		float sum= 0.0f;
		for (int i = 0; i < Wa; i++) {
			sum +=a[row*Wa + i] * b[i*Wb + col];
		}
		c[row*Wb + col] = sum;
	}
}

__kernel void MatrixMultiply1(__global float* a, __global float* b, __global float* c, int Wa, int Wb,
int La) {
	int col = get_global_id(0);

	for (int row = 0; row < La; row++)
	{
		float sum = 0.0f;
		for (int i = 0; i < Wa; i++) {
			sum +=a[row*Wa + i] * b[i*Wb + col];
		}
		c[row*Wb + col] = sum;
	}
}
__kernel void MatrixMultiply2( __global float* a, __global float* b, __global float* c, int Wa, int Wb) {
	int row = get_global_id(0);
	int col = get_global_id(1);
	float sum = 0.0f;
	for (int i = 0; i < Wa; i++) {
		sum += a[row*Wa + i] * b[i*Wb + col];
	}
	c[row*Wb + col] = sum;
}
```

А вот четвертый вызвал некоторые затруднения. Сначала я реализовал его самым очевидным способом, с помощью цикла for, таким образом, что каждый запуск кернела прибавляет к каждому элементу итоговой матрицы очередное слагаемое.

```
    size_t global_work_size3[2] = { row1, col2 };
    for (int i = 0; i < col1; i++)
    {
    ret = clSetKernelArg(kernels[3], 5, sizeof(cl_int), (void *)&i);
    ret = clEnqueueNDRangeKernel(command_queue, kernels[3], 2, NULL, global_work_size3, NULL, 0, NULL,
  NULL);
    }


  __kernel void MatrixMultiply3( __global float* a, __global float* b, __global float* c, int Wa,
  int Wb, int ray) {
		int row = get_global_id(0);
		int col = get_global_id(1);
		c[row*Wb + col] += a[row*Wa + ray] * b[ray*Wb + col];
	}
```

Данный алгоритм показался мне недостаточно эффективным из-за большого количества вызовов кернела, и производительность показалась мне недостаточной, поэтому я стал искать способ как обойтись одним вызовом кернела. Как альтернативу я решил создать метод использующий Atomic-переменную. Я разбил все work-item'ы на work-group'ы так, чтобы одна work-group'а была ответственная за один элемент в итоговой матрице, а каждый work-item внутри work-group'ы отвечает за свое слагаемое. Сумма накапливается в локальной Atomic-переменной.

Сам кернел выглядит так:

```
__kernel void MatrixMultiply3(__global float* a, __global float* b, __global float* c, int Wa, int Wb)
{
	int row = get_global_id(0);
	int col = get_global_id(1);
	int ray = get_global_id(2);
	local float *value;
	*value = 0.0f;
	float res = a[row*Wa + ray] * b[ray*Wb + col];

	AtomicAdd(value, res);
	barrier(CLK_LOCAL_MEM_FENCE);
	c[row*Wb + col] += *value;
}
```

Где AtomicAdd — функция, которая увеличивает Atomic Float переменную на значение, передаваемое вторым параметром. Но из-за того, что один из мои «девайсов» (Nvidia GeForce GT 730M) имеет NDRange = (1024, 1024, **64**), а числа которыми мне приходится оперировать в разы больше значения 64, то пришлось все-таки разбивать мой кернел на части и исполнять их по отдельности. Но все же, я добился сокращения числа запусков кернелов в разы.

Таким образом, код исполнения кернела выглядит следующим образом:

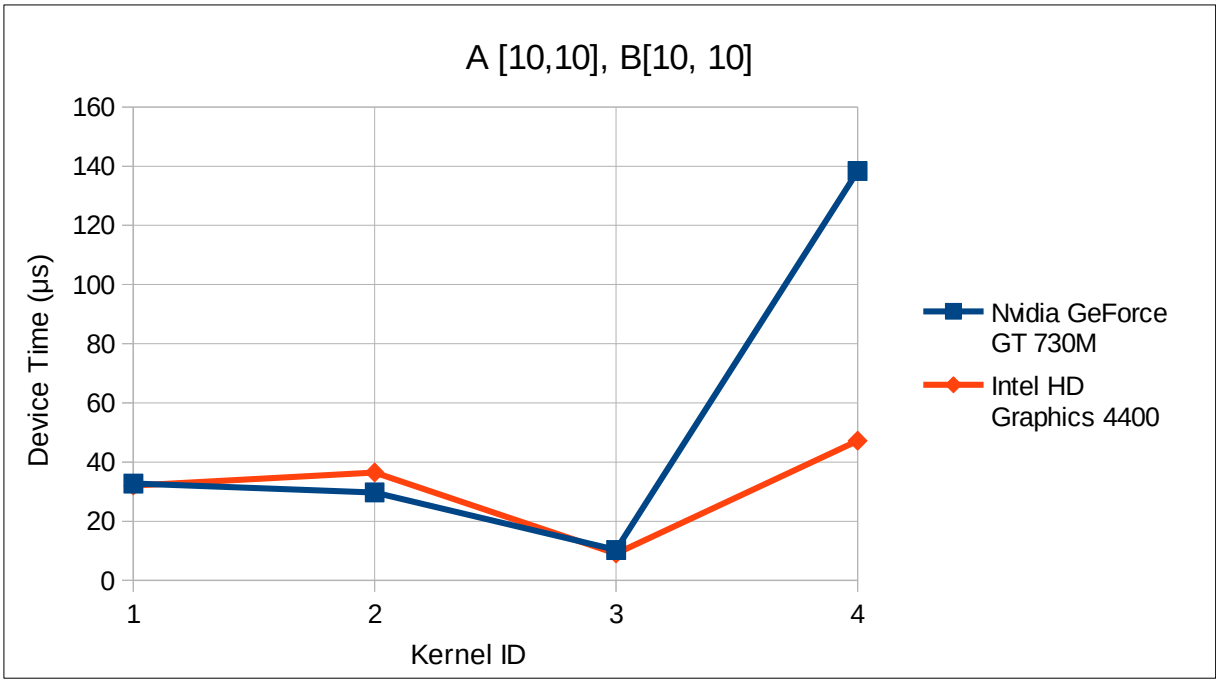
```
   int num_of_iter[3] = { ceil((float)row1/ (float)NDRange[0]), ceil((float)col2 / (float)NDRange[1]),
ceil((float)col1 / (float)NDRange[2])};
   size_t global_work_size3[3];
   size_t local_work_size3[3] = { 1, 1, col1 };

   for (int i = 0; i < num_of_iter[0]; i++)
   {
       for (int j = 0; j < num_of_iter[1]; j++)
       {
          for (int k = 0; k < num_of_iter[2]; k++)
          {
              size_t global_work_offset3[3] = { i*NDRange[0], j*NDRange[1],k*NDRange[2] };
              global_work_size3[0] = (i != num_of_iter[0] - 1) ? NDRange[0] : row1 - NDRange[0] * i;
              global_work_size3[1] = (j != num_of_iter[1] - 1) ? NDRange[1] : col2 - NDRange[1] * j;
              global_work_size3[2] = (k != num_of_iter[2] - 1) ? NDRange[2] : col1 - NDRange[2] * k;
              local_work_size3[2] = global_work_size3[2];
              ret = clEnqueueNDRangeKernel(command_queue, kernels[3], 3, global_work_offset3,
                           global_work_size3, local_work_size3, 0, NULL, NULL));
          }
       }
   }
```

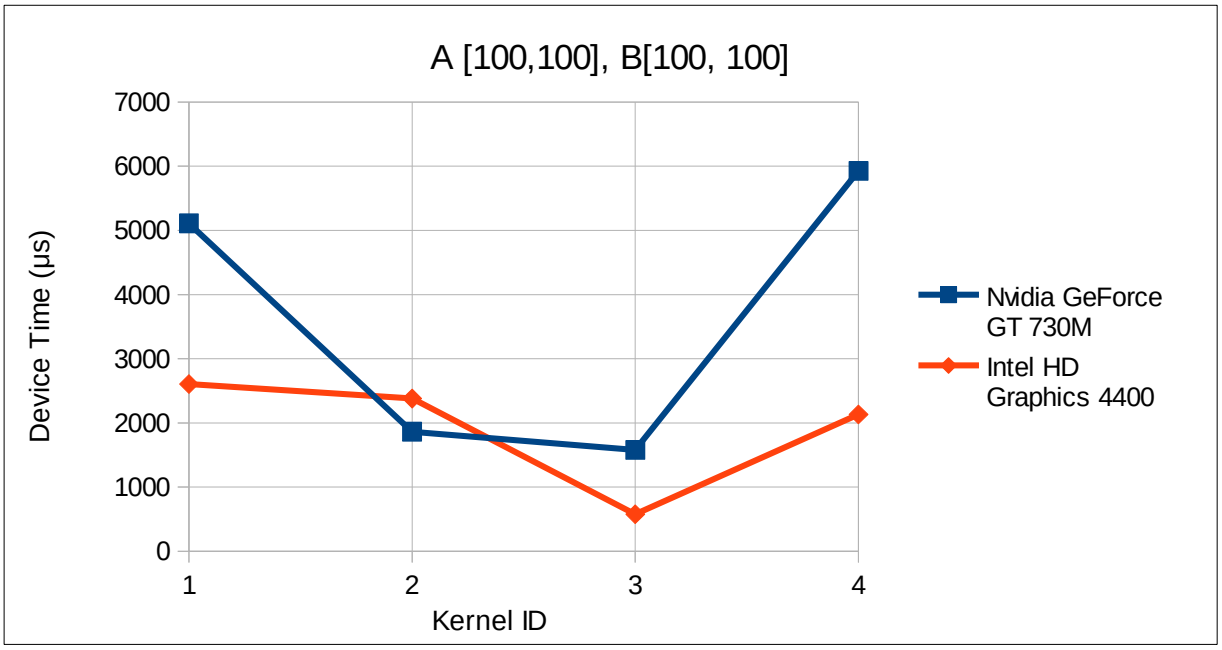
Где NDRange — это массив, в котором хранятся числа, обозначающие максимально возможные количества work-item'ов в каждом из измерений (dimension), для девайса, выбранного пользователем.

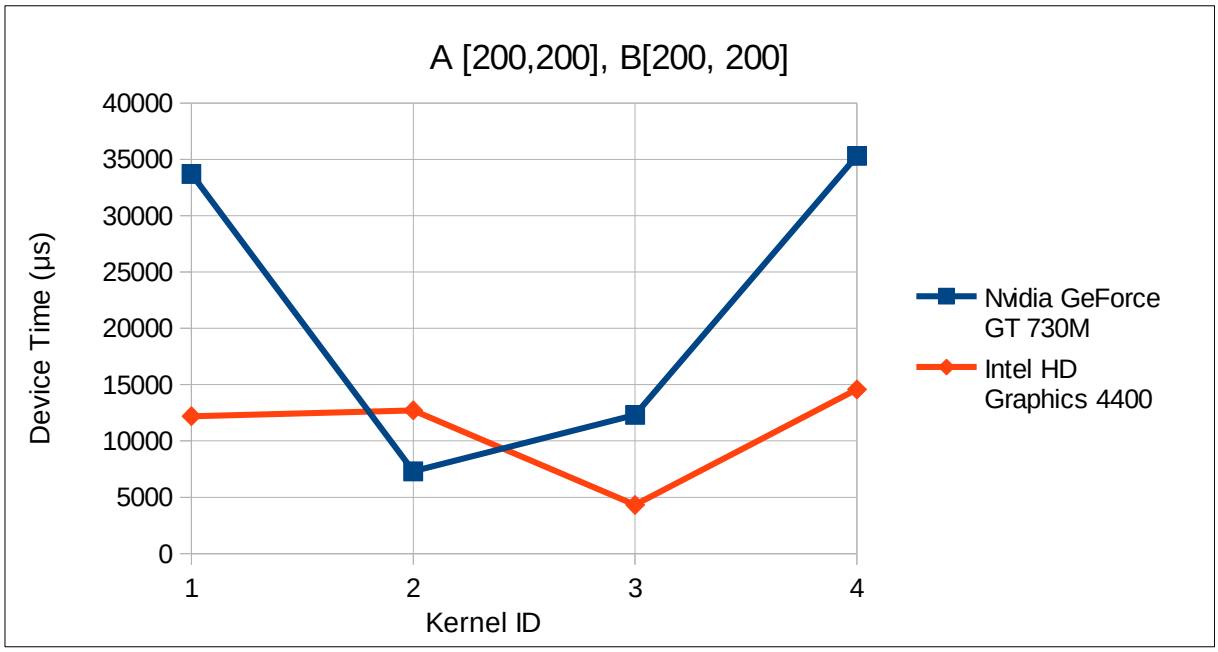
Посмотрев на результаты профилирования, можно заметить, что производительность второго алгоритма, как ни странно, оказалась еще хуже, чем та, которую показал первый алгоритм.

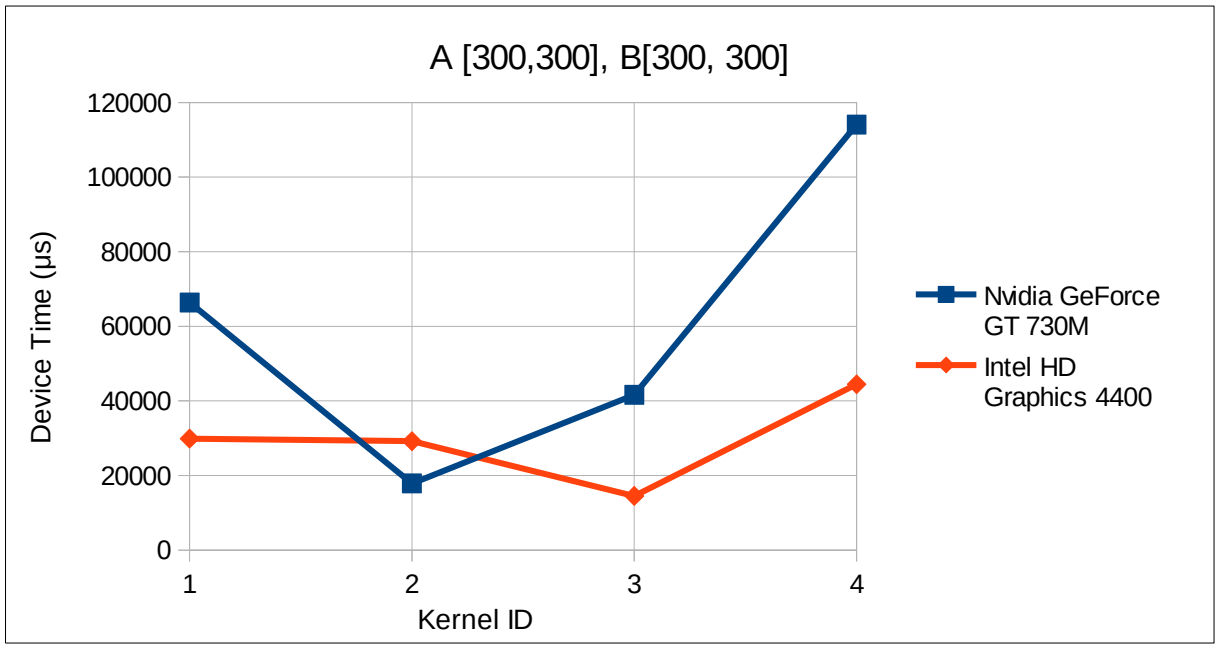
2. Профилирование.

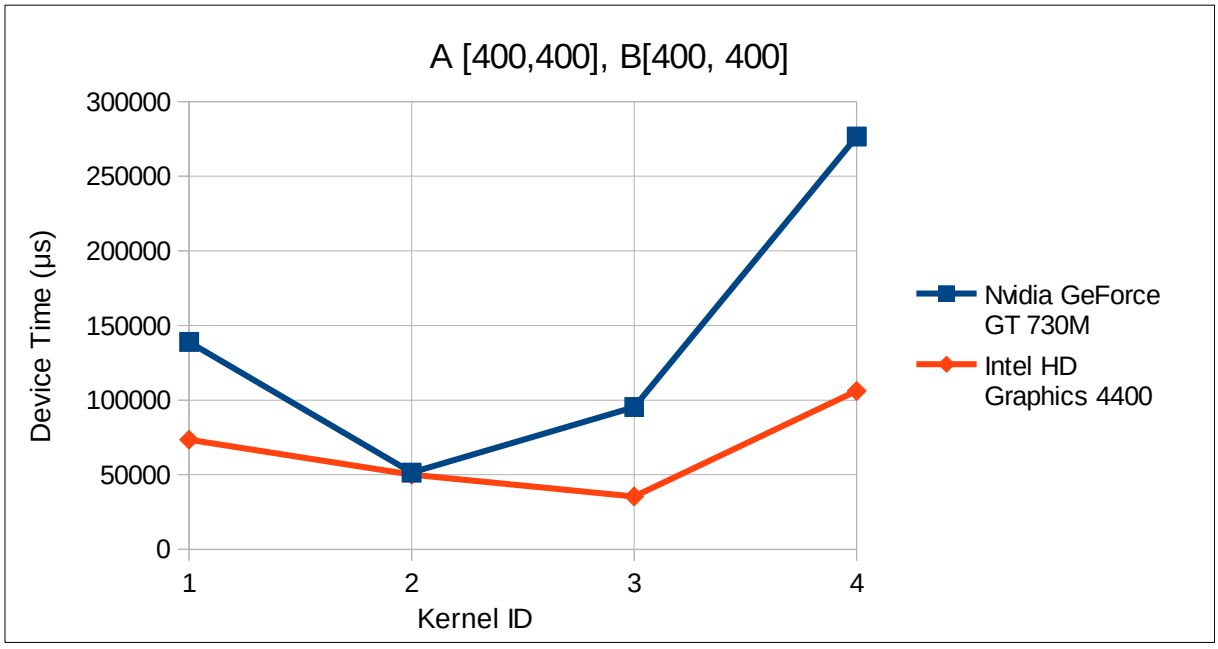
1. Использование цикла for.

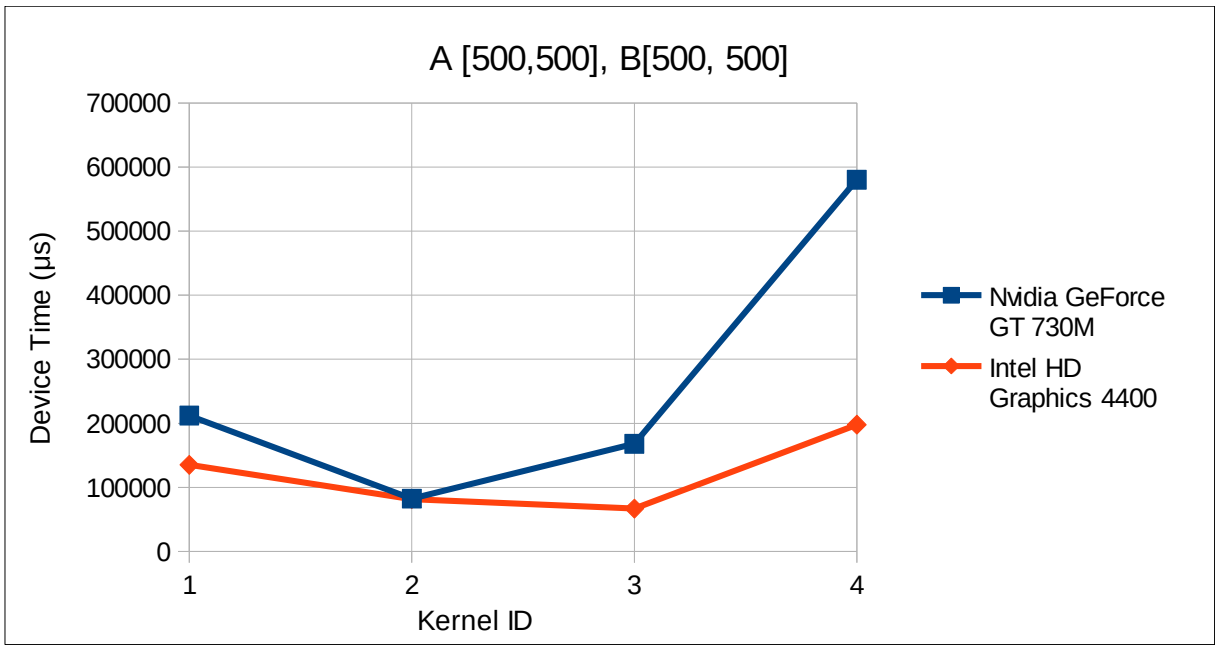


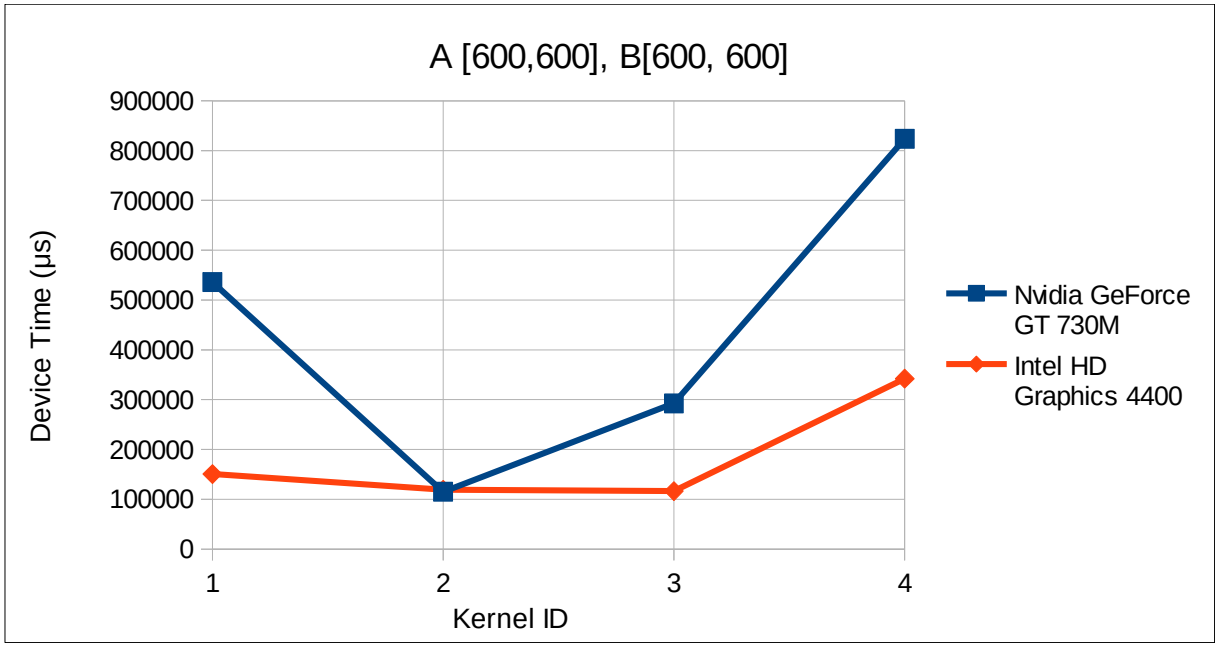


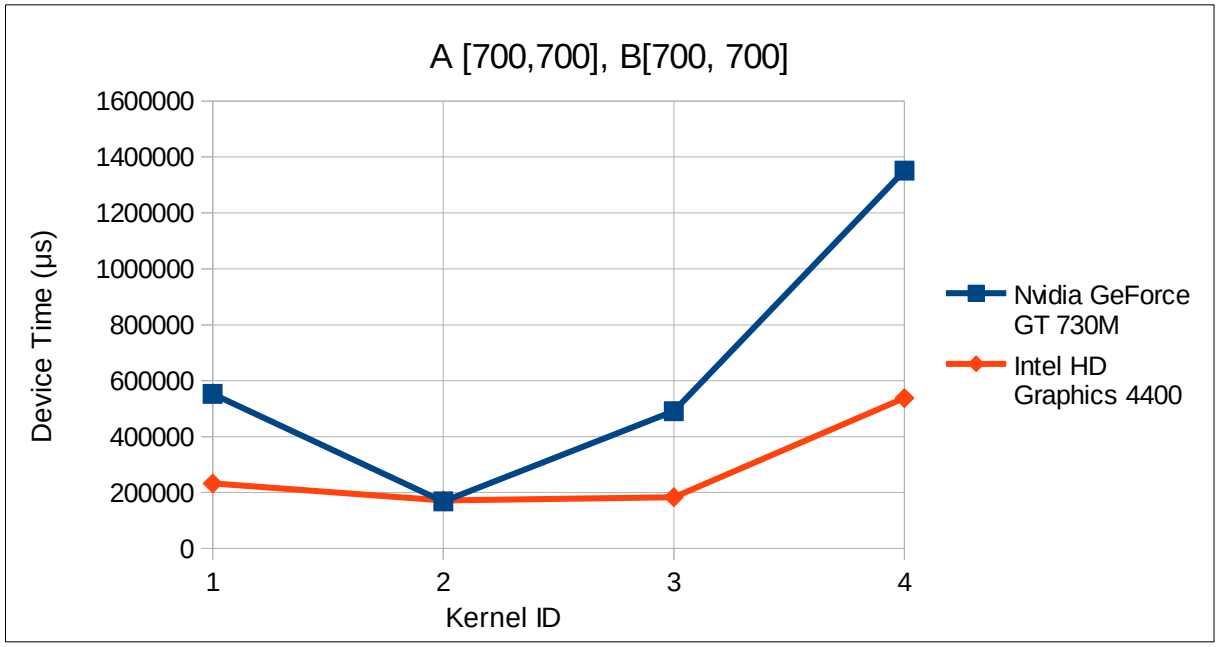

A [200,200], B[200, 200]
Device Time (µs)
40000
35000
30000
25000
20000
15000
10000
5000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400

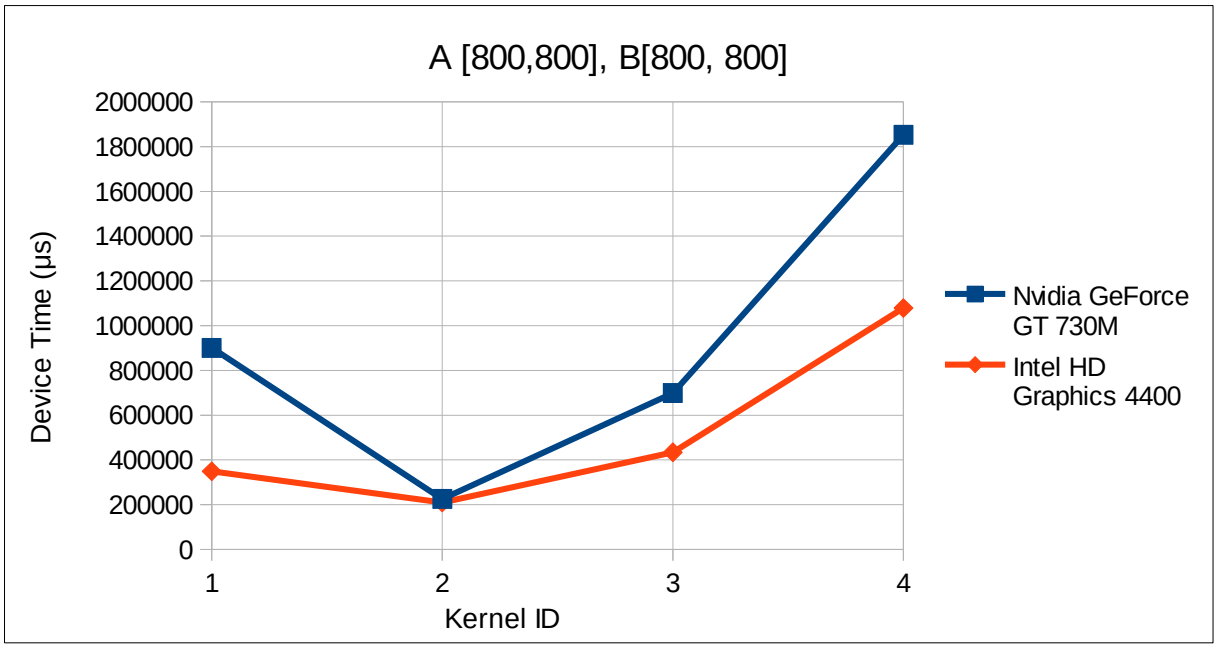

A [300,300], B[300, 300]
Device Time (µs)
120000
100000
80000
60000
40000
20000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400

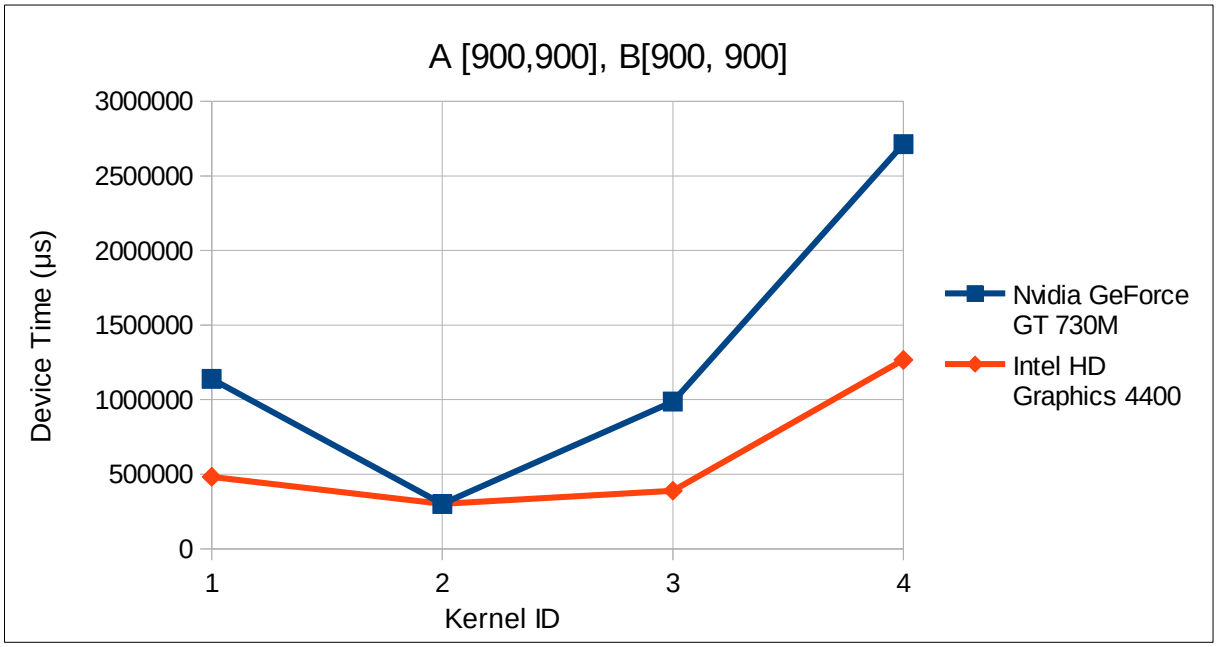

A [400,400], B[400, 400]
Device Time (µs)
300000
250000
200000
150000
100000
50000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400

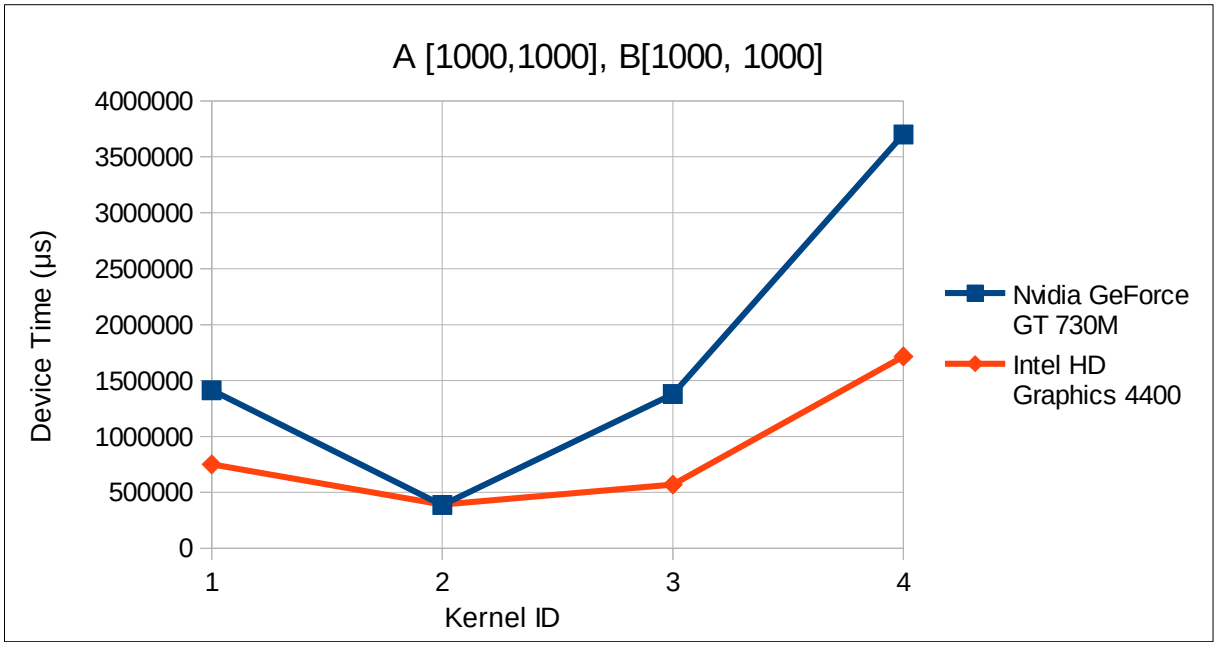

A [500,500], B[500, 500]
Device Time (µs)
700000
600000
500000
400000
300000
200000
100000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400


A [600,600], B[600, 600]
Device Time (µs)
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400

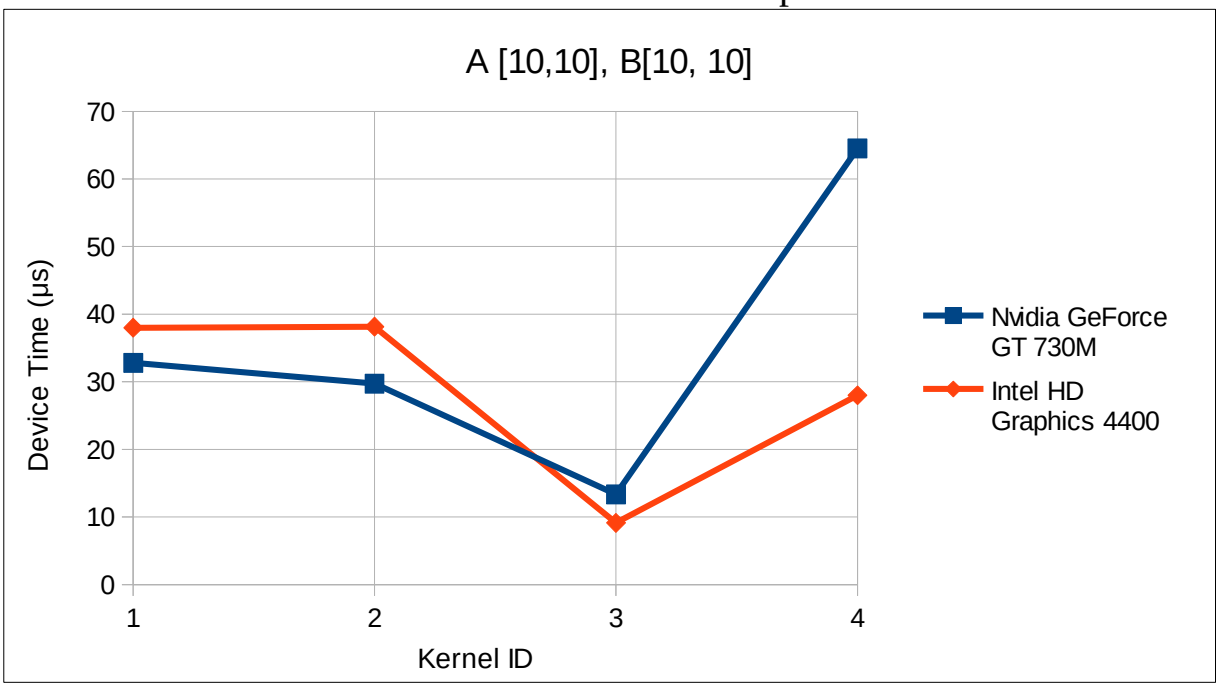

A [700,700], B[700, 700]
Device Time (µs)
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400

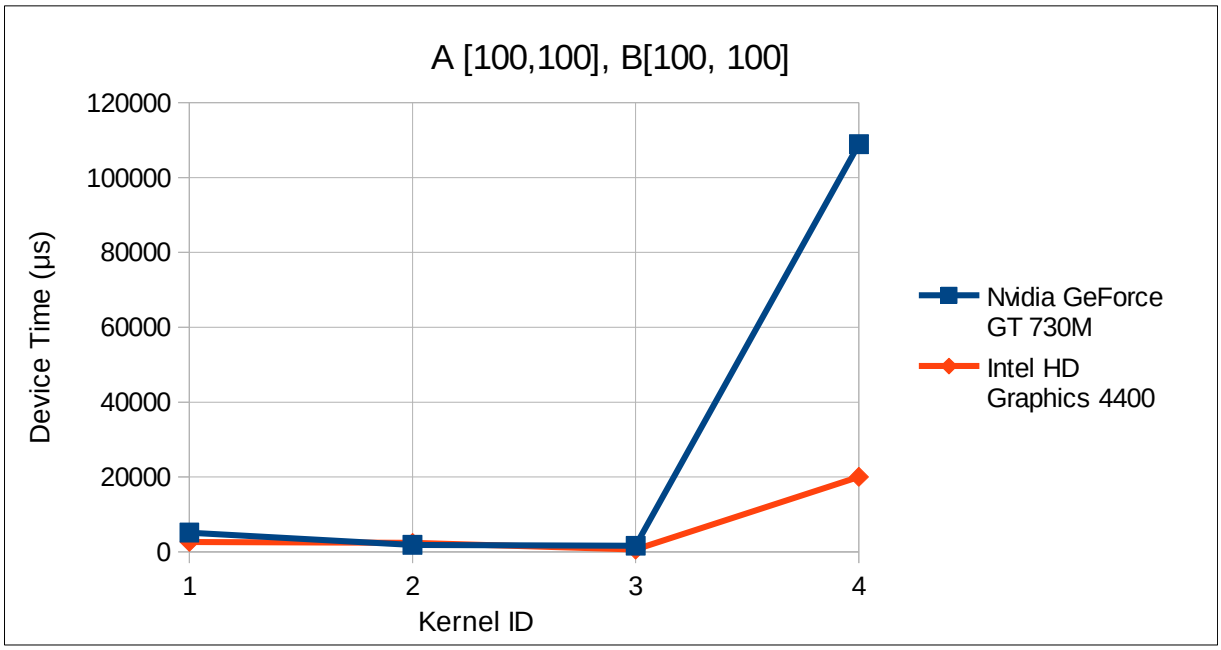

A [800,800], B[800, 800]
Device Time (µs)
2000000
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400

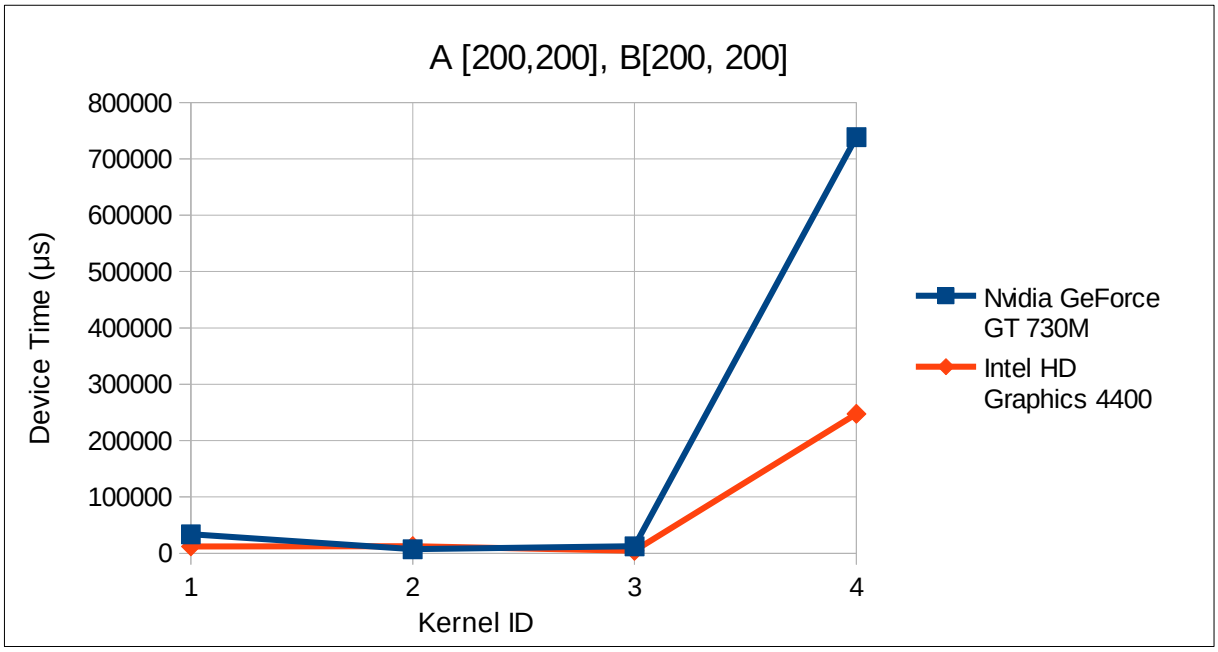

A [900,900], B[900, 900]
Device Time (µs)
3000000
2500000
2000000
1500000
1000000
500000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400

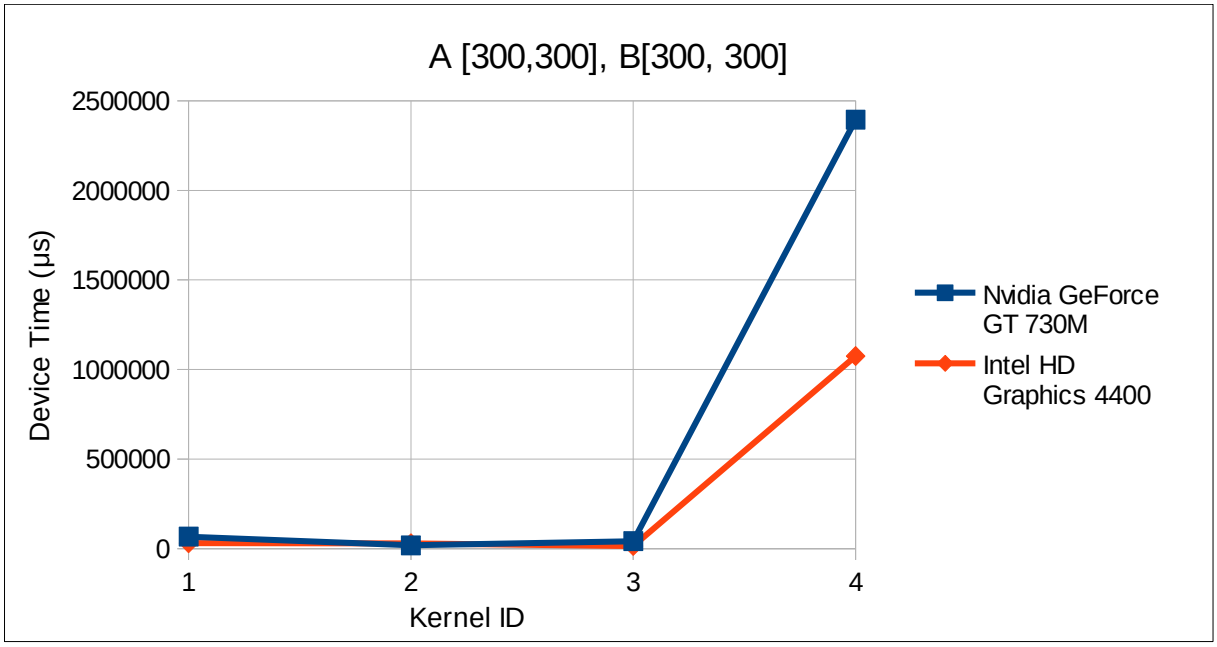

A [1000,1000], B[1000, 1000]
Device Time (µs)
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400

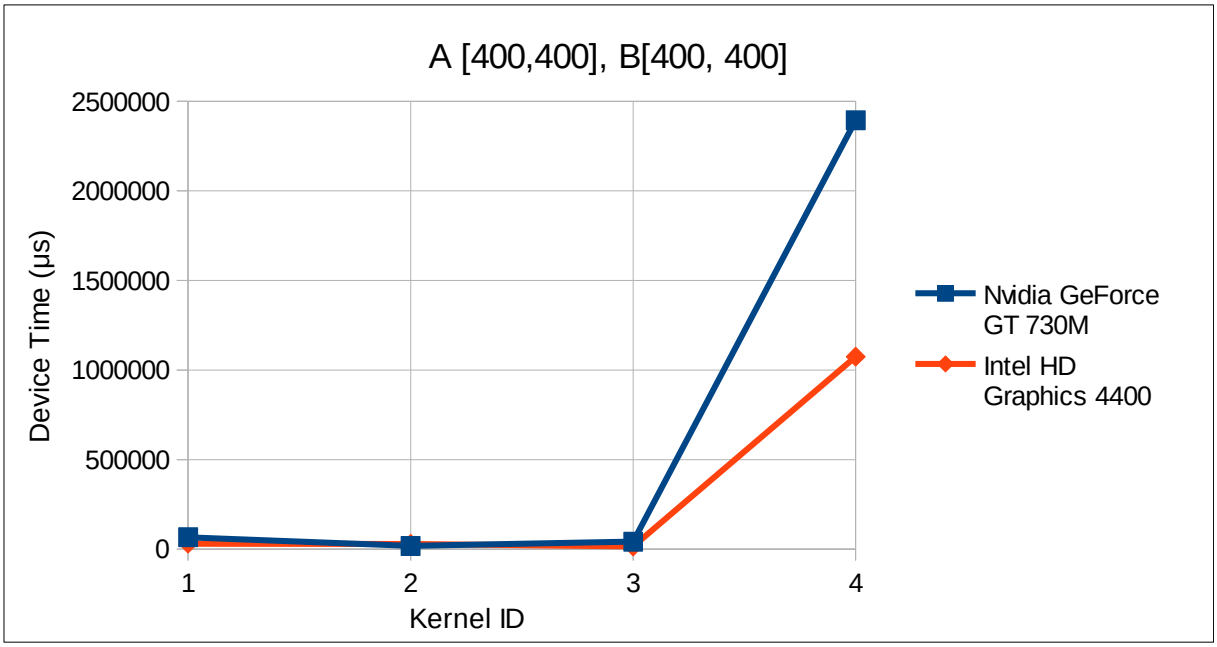
2. Использование Atomic-переменной.

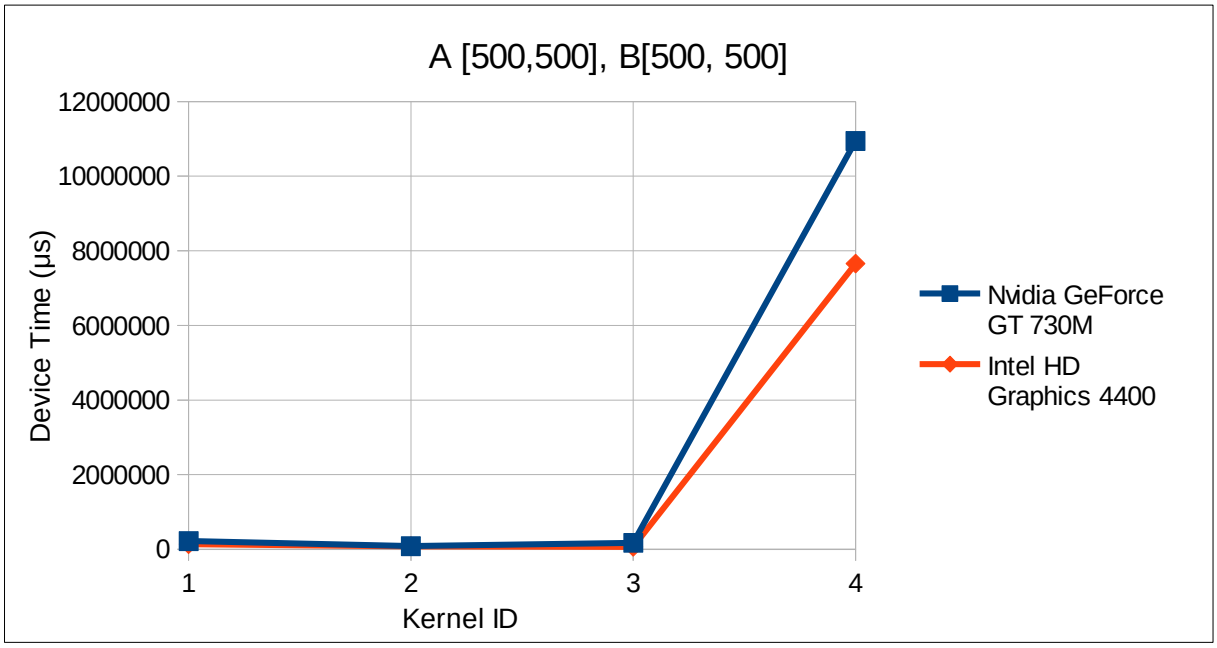


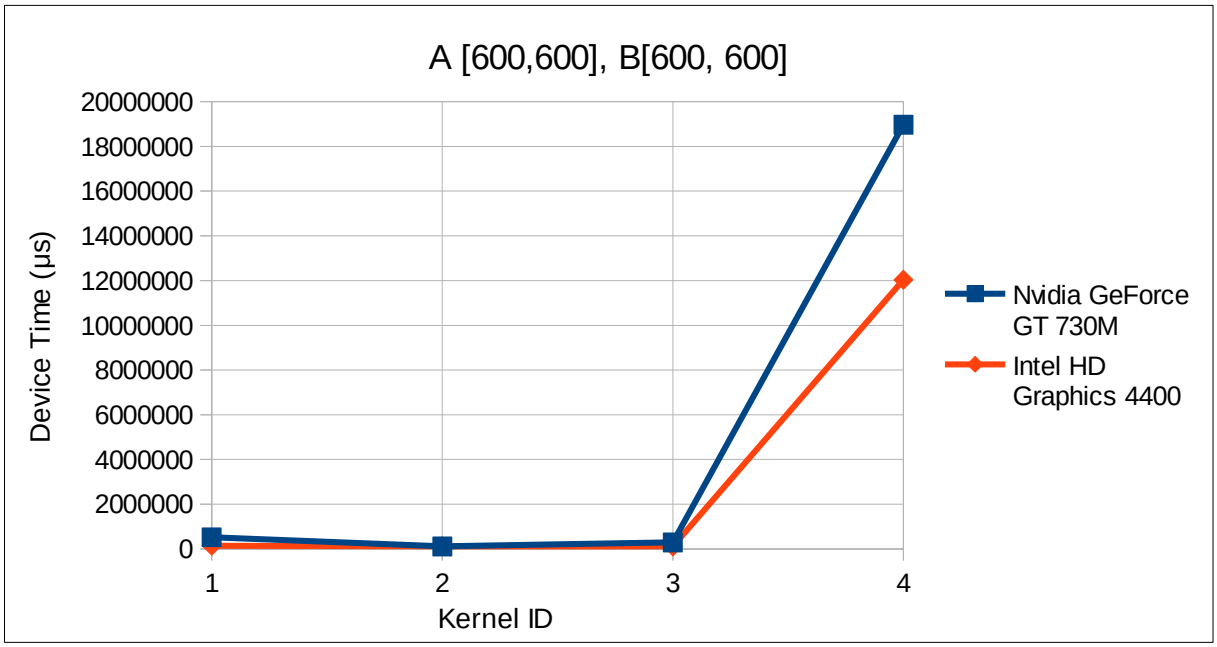


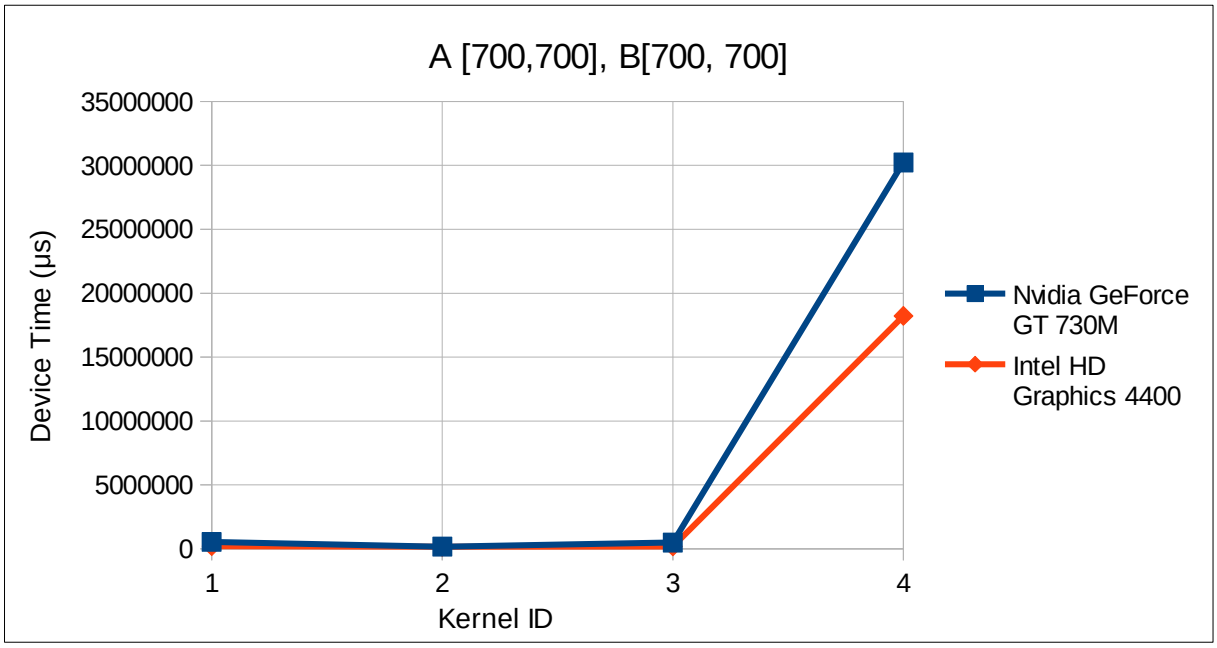


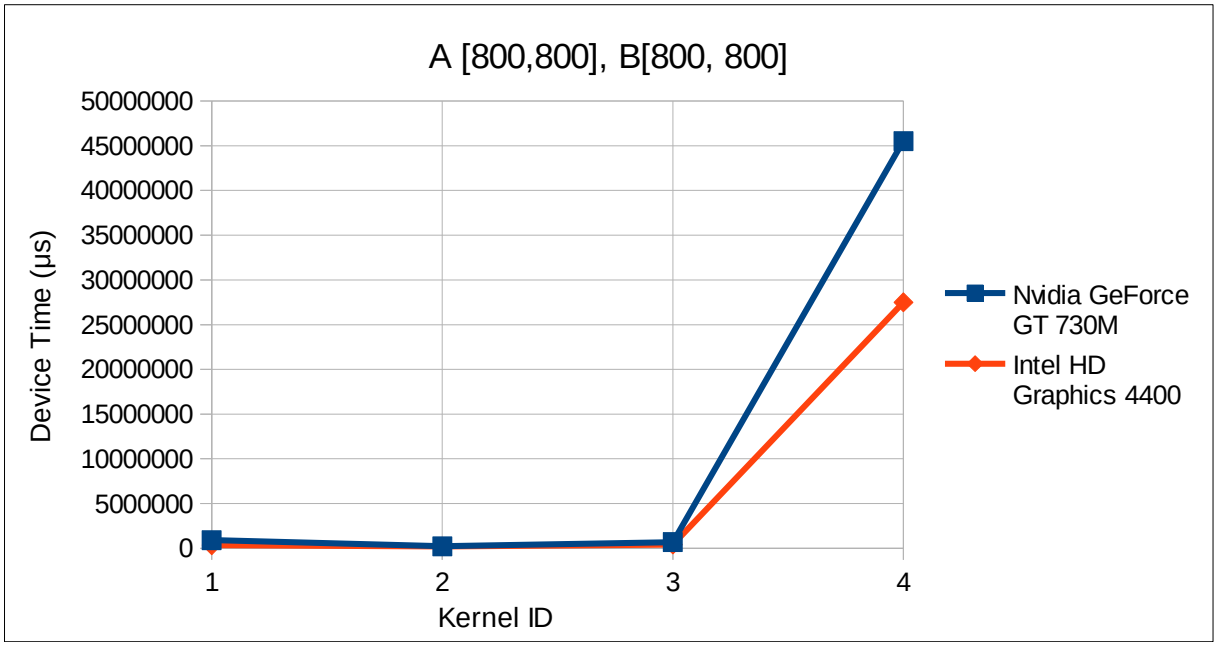

A [300,300], B[300, 300]
2500000
2000000
1500000
1000000
500000
0
Device Time (µs)
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400

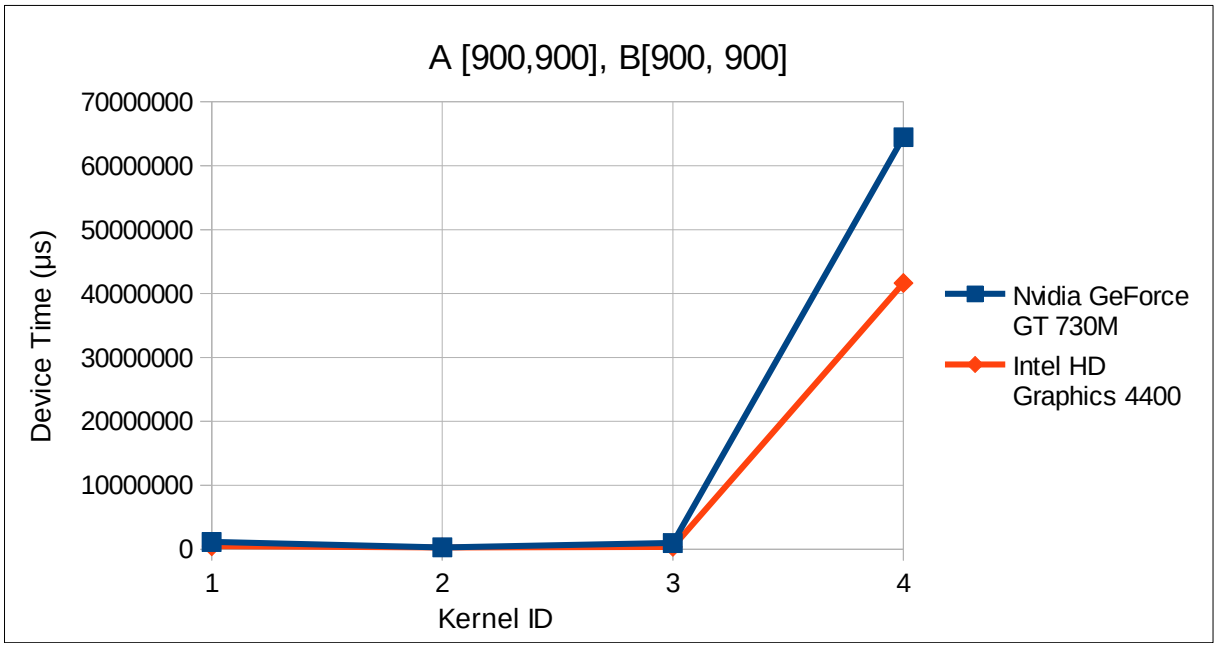

A [400,400], B[400, 400]
2500000
2000000
1500000
1000000
500000
0
Device Time (µs)
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400

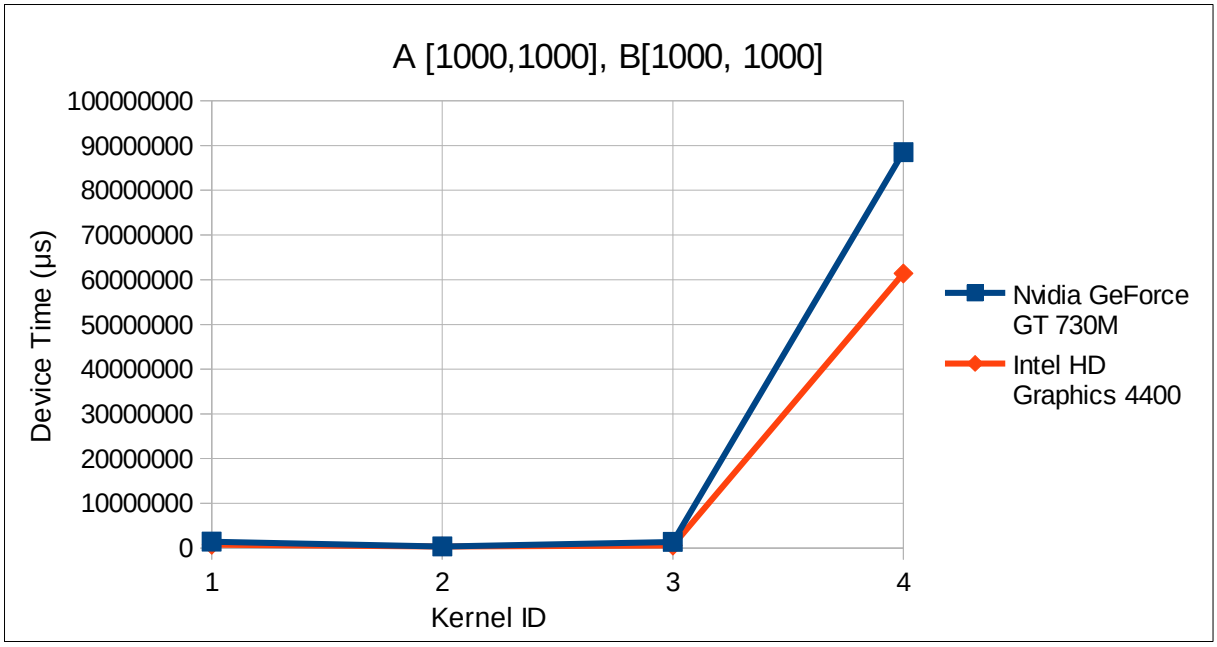

A [500,500], B[500, 500]
12000000
10000000
8000000
6000000
4000000
2000000
0
Device Time (µs)
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400


A [600,600], B[600, 600]
Device Time (µs)
20000000
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400


A [700,700], B[700, 700]
Device Time (µs)
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400


A [800,800], B[800, 800]
Device Time (µs)
50000000
45000000
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
1
2
3
4
Kernel ID
Nvidia GeForce GT 730M
Intel HD Graphics 4400





3. Выводы

1. Самыми эффективными по производительности оказались кернелы 2 и 3. При том, третий кернел показал лучшие результаты при меньших значениях, когда как при больших значениях более эффективным оказался второй кернел.

2. Среди двух вариантов реализации четвертого кернела, самым эффективным оказался метод с использованием цикла, а не с применением Atomic-переменной.