GPU Programming CS6023

Fall 2018

Assignment 1

Report

Author: Varun Sundar

EE16B068

23rd August 2018

1 Implementation and Technical Notes

The code was run on a *GTX 1080 Ti* and not the cluster of *K'40s*. This could be a reason for variance in the optimal configuration found for the GPUs. However, the code does not make any hardware assumptions and can therefore be run on any suitable cluster.

CUDA version 9.0 was used to compile the code, with `nvcc` as the device compiler and `gcc` as the host compiler.

2 Question 1 and 2

Querying Device parameters via `cudaDeviceProp`.

Following parameters were queried:

- Scope of support of L1 Cache (Global or local): **Yes**
- Size of L2 Cache: **Yes**
- Maximum permissible threads per block: **1024**
- Registers allocated per block: **65536**
- Registers available in a streaming multiprocessor: **65536**
- Warp Size (bytes) : **32**
- Total amount of memory available in the GPU (in bytes): **11719409664** (12GB)

2.1 Code Blocks (pertinent only)

```
cudaGetDeviceCount(&nDevices);
    for (int i = 0; i < nDevices; i++) {
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, i);
        printf("Device Number: %d\n", i);
        printf("  Device name: %s\n", prop.name);
        printf("  Memory Clock Rate (KHz): %d\n",prop.memoryClockRate);
        printf("  Memory Bus Width (bits): %d\n",prop.memoryBusWidth);
        printf("  Is L1 Cache supported globally :(0/1) %d\n",prop.
           globalL1CacheSupported);
        printf("  Is L1 Cache supported locally :(0/1) %d\n",prop.
           localL1CacheSupported);
       // ..Other Device Properties.. //
        printf("  No of registers available in a streaming
           multiprocessor : %d\n",prop.regsPerMultiprocessor);
        printf("  Warp Size :(bytes) %d\n",prop.warpSize);
        printf("  Grid Size :(bytes) %ld\n",prop.maxGridSize);
```

```
15          printf("  Total memory :(bytes) %ld\n",prop.totalGlobalMem);
16          printf("  Peak Memory Bandwidth (GB/s): %f\n\n",2.0*prop.
               memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
17      }
18  }
```

We run the list of properties for all possible GPU's present in the given machine.

2.2 Roof-line Plot

Plot of the roof-line model for the queried GPU.

The Roof-line model helps understand the bottlenecks in achieving greater performance. Till a certain operational intensity, it is limited by the bandwidth of global context transfer (host to device data). Further, there is the peak [TFLOPS] limit, which is a hardware-architecture limit.

For the case of the *GeForce 1080 Ti*, the peak FLOPS may be computed as:

$$Cores * FLOPS * ClockFrequency$$

This yields 11.3 T FLOPS for the GPU, which matches closely with the manufacturer's benchmarks (11.5). Peak data bandwidth is obtained from the device query.

The resultant plot : **Figure 1**

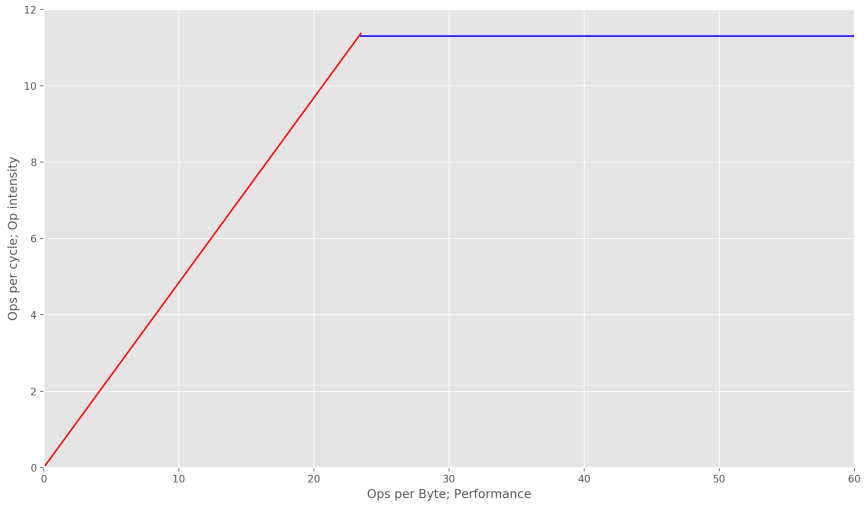

Figure 1: Roof-line model for **GTX 1080 Ti**

2.3 Other Details

Device Number: 0

Parameter	Specifics
Device name	GeForce GTX 1080 Ti
Memory Clock Rate (KHz)	5505000
Memory Bus Width (bits)	352
Is L1 Cache supported globally	Yes
Is L1 Cache supported locally	Yes
L2 Cache Size (bytes)	2883584
Max no of threads per block	1024
No of registers available in a block	65536
No of registers available in a streaming multiprocessor	65536
Warp Size (bytes)	32
Grid Size (bytes)	140731727872496
Total memory (bytes)	11719409664
Peak Memory Bandwidth (GB/s)	484.440000

Table 1: Device specifications, *device:0*

3 Question 3 and 4

We find the optimal number of threads per block to run the given vector addition empirically.

Question 3 involves writing kernel functions to implement vector addition over multiple threads, one operation each thread. We output the result of adding two 2^{15}sized vectors, which are randomly generated.

For **Question 4**, we empirically determine the optimal number of threads per block. From **Question 1** we know that the maximum permissible threads per block is 1024, hence we vary the threads per block from 128 to 1024 in powers of 2, and calculate the run-time in each case.

Note that, the run time has been calculated without considering the host allocation time. We believe that this is a valid choice, since the host allocation time maybe isolated for each process and therefore can be effectively excluded from the run time calculations.

3.1 Code Changes

```
19          __global__ void VecAdd(float* A, float* B, float* C, int N){
20        // Host code
21        int i = blockDim.x * blockIdx.x + threadIdx.x;
22        if (i < N)
23            C[i] = A[i] + B[i];
24      }
25      // .... //
26
27      printf("Array A (first 10 values) \n ");
28      for(loop = 0; loop < N; loop++){
29      h_A[loop] = rand() % 100 + 1;
30        if (loop<10){
31            printf("%f ", h_A[loop]);
```

```
      }
    }

    printf("\nArray B (first 10 values) \n ");
    for(loop = 0; loop < N; loop++){
        h_B[loop] = rand() % 100 + 1;
        if (loop<10){
            printf("%f ", h_B[loop]);
       }
     }
// .... //
cudaEventRecord(start, 0);
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,d_B, d_C, N);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_spent, start, stop);
time_spent=time_spent/(avg_loop-1)*10;
```

SI (vector addition of 2^{15} size)	Threads per Block (powers of 2)	Average GPU run-time (for 10 runs)
1	32	0.312521
2	64	0.304183
3	128	0.304020
4	256	0.303870
5	512	0.300294
6	1024	0.286491

Table 2: Average time for 10 passes versus threads per block

The consequent plot: **Figure 2**
Hence, **optimal no of threads per block** is **1024**.

3.2 Reasoning

The optimal threads per block may not be upper bounded for this low intensity (operational), and hence, using the maximum possible number of threads per block is optimal.

4 Question 5

Here, we empirically investigate the dependence of run-time on the number of operations per thread.

4.1 Code Changes

```
        __global__ void VecAdd(float* A, float* B, float* C, int N){
      // Host code
      int j;
    int i = blockDim.x * blockIdx.x + threadIdx.x;
```

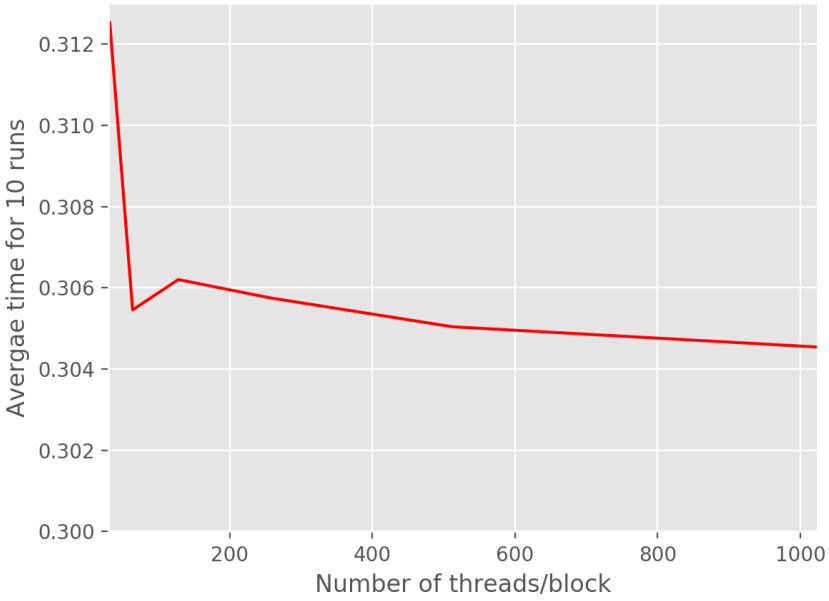

Figure 2: Average Runtime versus Threads per Block for **GTX 1080 Ti**

```
53      if (i < N_op){
54          for (j=0;j<op_loop;j++)
55              C[i*op_loop+j] = A[i*op_loop+j] + B[i*op_loop+j];
56      }
57      }
58     // Array of op's to try//
59     for (op_loop_ii=0;op_loop_ii<10;op_loop_ii++){
60          op_loop_array[op_loop_ii]=pow(2,op_loop_ii);
61      }
62 // Run kernel over these ops and average each run //
63 cudaEventRecord(start, 0);
64 VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,d_B, d_C, N);
65 cudaEventRecord(stop, 0);
66 cudaEventSynchronize(stop);
67 cudaEventElapsedTime(&time_spent, start, stop);
68 time_spent=time_spent/(avg_loop-1)*10;
```

4.2 Run-Times

Table 3 contains the relevant logs.

The consequent plot: **Figure 3**

Hence, **optimal no of ops per thread** is **4**.

SI (vector addition of 2^{15} size)	Ops per loop (powers of 2)	Average GPU run-time (for 10 runs)
1	1	0.298492
2	2	0.349331
3	4	0.296480
4	8	0.340138
5	16	0.374680
6	32	0.454981
7	64	0.570481
8	128	0.864831

Table 3: Average time for 10 passes versus ops per thread

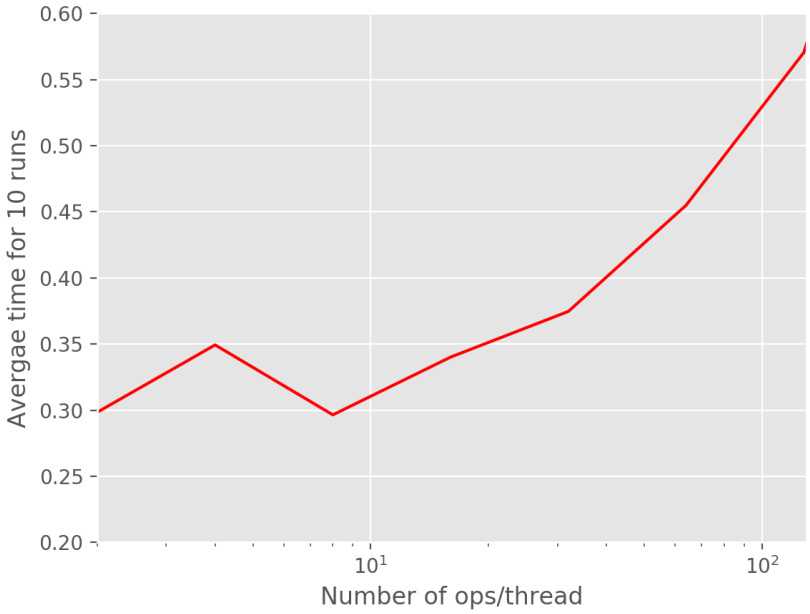

Figure 3: Average Runtime versus Ops per Thread for **GTX 1080 Ti**

4.3 Reasoning

The optimal operations per block is dependent on the level of SIMD parallelism a single thread may be able to achieve. With an optimal of **4**, it is possible that the device has over **8 ALU's** per thread-context.

5 Question 6 and 7

Again, empirically, we observe the changes in run-time per vector size of the random vectors being operated upon.

5.1 Run-Times

No major code changes here, besides varying the vector sizes given these optimal *Threads per Block* and *operations per thread*.

SI (vector addition of 2^{15} size)	Vector Size (powers of 2)	Average GPU run-time (averaged over 1000 runs)
1	2^{15}	0.032892
2	2^{16}	0.032452
3	2^{17}	0.044452
4	2^{18}	0.101014
5	2^{19}	0.198954
6	2^{20}	0.341814

Table 4: Average time for single pass versus Vector Size

Consequent plot: **Figure 4**

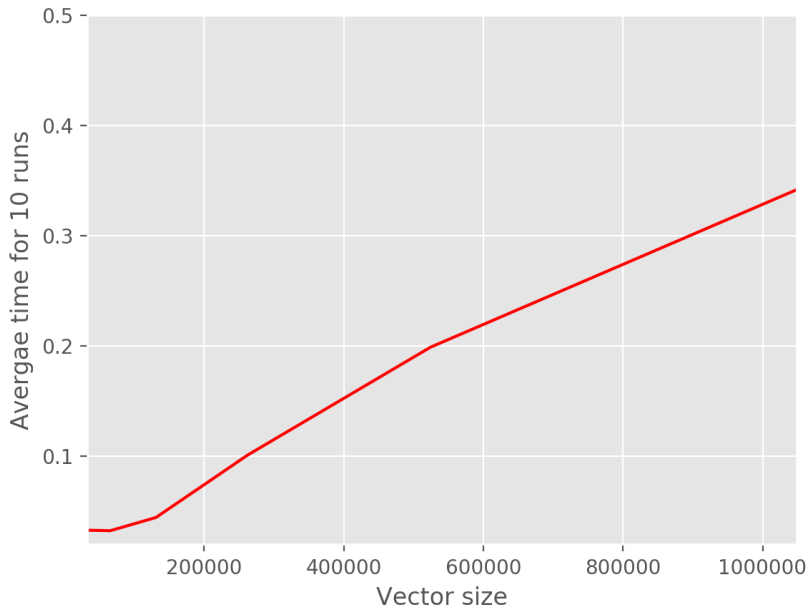

Figure 4: Average Runtime versus Vector Size for **GTX 1080 Ti**

5.2 Reasoning

The variation of runtimes with size is not linear $O(n)$ as would have been expected of a CPU, but is more or less constant upto 2^{18} as a vector size. Post this, it is possible that the size no longer allows the same degree of parallelism with respect to caching, Grid-to-thread transfer, etc.