CS6023
Assignment 1

Anant Shah
EE16B105
Department of Electrical Engineering

August 23, 2018

Abstract

This report looks at finding the properties of the Tesla K40C Nvidia GPU. The roof-line model for the GPU is derived using the queried properties. A vector addition kernel is defined, where different parameters such as number of threads per block, number of operations per thread, etc. are varied to empirically obtain the optimal parameters for vector addition.

Device Properties :

The device properties of the Tesla K40C GPU are obtained by using the cudaGetDeviceProperties() function which takes a pointer to a cudaDeviceProp variable as an argument. Some of the properties queried are :

- Is L1 cache supported locally
- Maximum number of threads per block
- Warp size
- Total Global memory in a GPU

and a few others. The properties obtained are shown below :

```
L1 cache supported locally : 1
L1 cache supported globally : 1
L2 cache size(in bytes) : 1572864
Maximum number of threads per block : 1024
Registers allocated per block : 65536
Registers available in a Streaming Multiprocessor : 65536
Warp Size : 32
Total Global Memory in the GPU(in bytes) : 11995578368
```

Roof-Line Model :

The roof-line model is a performance model used to provide performance estimates of operations running on multi-core systems such as a GPU. It gives an intuitive picture of when an operation would be memory bound and when an operation would be compute bound. To obtain the roof-line model parameters of the cluster GPU device, the wfollowing properties were queried.

- Bus Width = 384
- RAM clock rate= $3.004GHz$

From the data obtained above, it is easy to calculate the optimal or ideal memory bandwidth. With the RAM clock rate, we can find how many times per second, data is being sent through each lane in the bus.

Since the RAM in a GPU is DDR-RAM(Double Data Rate), it can send two bits in one lane per clock cycle. Hence the total data sent per second can be calculated to be :

$$Bandwidth = BusWidth * 2 * ClockRate/8$$

$$= 384 * 2 * 3.004 * 10^9/8$$

$$= 288.38GB/s$$

To calculte the peak performance of the device, we requier information such as :

- Number of cores performing SIMD instructions
- Number of multiply-add units and the number of multiply units. The mulitply-add units will perform 2 operations in one clock cycle.
- The clock rate.

Hence, the peak performance of the device will be given by

$$Performance = totalcores * (2 * (mul - add) + mul) * clock$$

For the Tesla K40C, the peak single-precision floating point performance turns out to be $4.29TFLOPS$.

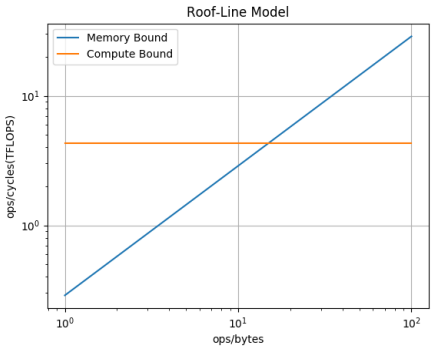

Figure 2: Roof-Line Model

If we consider, the operation of a simple vector addition, 1 operation requires 12 bytes(8 for the 2 input vectors and 4 for the output vector). If the GPU operated at peak performance for this simple operation, then 12 bytes of data would be needed in approximately $0.23ps$, which would equate to an exhorbitant bandwidth of $51.48TB/s$ which is just not possible. Hence, an operation such as vector addition remains in the memory bound region of the roof-line model.

Vector Addition :

Multi-core processors (such as GPUs) are useful in speeding up parallel applications. One such simple example is vector addition. In this operation, two elements are added from separate vectors and stored in the third vector. It can be seen that each operation will be independent of the others. This is one example of SIMD(Single Instruction Multiple Data), where only one instruction(add) is required and this instruction acts on all the data elements.

On a GPU, we define something known as a kernel which is the basic building block of a CUDA C program. When a kernel function is called, each thread(number defined by the user) executes the same piece of code. This is an example of SPMD(Single Program Multiple Data), which is different from SIMD. A collection of threads, known as warps execute in SIMD fashion.

To demonstrate the action of a kernel, we take vectors with the following specifications :

1. Size/Length = 32768
2. Data Type = Integer

The following are the kernel execution parameters :

1. Number of Blocks = 128
2. Number of Threads/Block = 256

A part of the final output is shown below :

```
Time for the vector addition : 0.000129 (seconds)
52043598
86737120
138780718.0000
74976572
7276840
82253412.0000
277511
60851793
61129304.0000
74635137
7819144
82454281.0000
86456278
25672005
112128283.0000
41372944
26986042
68358986.0000
72350717
58225108
130575825.0000
37873044
31305002
69178046.0000
```

Optimal Parameters :

The execution parameters of a kernal are now varied and time for the kernel execution is observed to obtain the optimal execution parameters. Parameters such as number of threads per block and number of operations per block are varied to obtain the minimum kernel-execution time.

0.0.1 Number of Threads Per Block :

Keeping the number of operations per thread constant at 1, the number of threads per block are varied from $[1, 1024]$ in powers of 2. The maximum number of threads that CUDA allows per block is 1024. If define the execution parameters such that it exceeds this limit, then the program will execute but the addition will not take place(all elements in result vector at the end of the execution will remain 0). The kernel-execution time for this case is tabulated below (the final reading is an avreage of 10 readings) :

Threads/Block	1	2	4	8	16	32	64	128	256	512	1024
Average Time(ms)	0.248	0.202	0.173	0.138	0.128	0.121	0.131	0.122	0.118	0.135	0.127

The plot of the kernel execution time versus the number of threads per block is shown in Figure 2. It is observed that the optimal number of threads per block is 256.

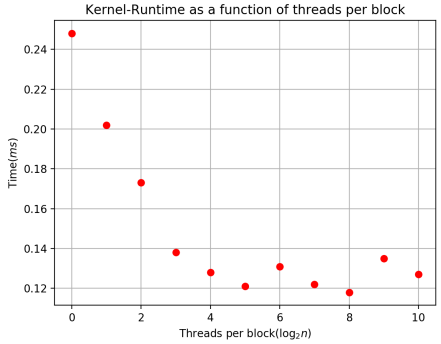

Figure 2 : Kernel-Execution Time vs Threads/Block

0.0.2 Number of operations per thread :

The optimal number of threads per block is 256. Keeping that constant and varying the number of operations per thread, the kernel execution time is noted. There are two methods to distribute the data-elements to the threads. The first one is the block distribution, where contiguous data items are given to one thread to compute. The other one is the cyclic distribution, where each thread gets the k^{th} element to work on where $k = \frac{SIZE}{OPS-PER-THREAD}$. The cyclic distribution method has been implemented. The number of operations per thread is varied from $[1, 32]$ in powers of 2. The kernel-execution time is tabulated below (average of 10 readings for each value of operations per thread) :

Operations/Thread	1	2	4	8	16	32
Average Time(ms)	0.118	0.126	0.116	0.125	0.126	0.126

The plot of the kernel-execution time versus the number of operations per thread is depicted in Figure 3. It is observed that the optimal number of operations for the optimal number of threads per block is 4.

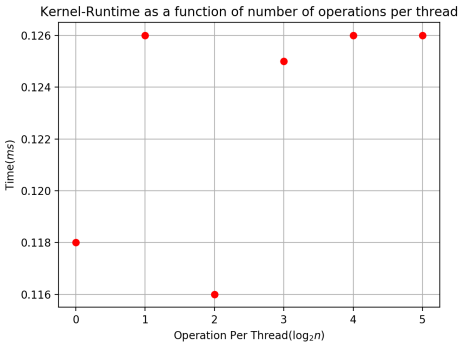

Figure 3 : Kernel-Execution time vs Number of Operations per Thread

0.0.3 Size of the Vector :

After obtaining the optimal conditions for number of threads per block and the number of operations per thread, the size of the vector is varied from $[2^{15}, 2^{20}]$ in multiples of 2. Ideally on a single core the opreation would be of the order $O(n)$. The same cyclic distribution method is used for distributing elements to the threads. The kernel-execution time for each vector size is tabulated below(average of 10 readings taken):

Size of the vector	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Average Time(ms)	0.116	0.132	0.179	0.225	0.249	0.301

The plot for the kernel execution time vs the size of the vector is shown in Figure 4.

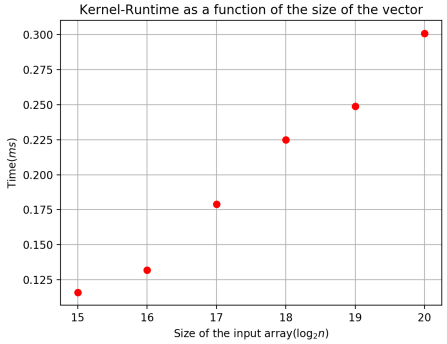

Figure 4 : Kernel-Runtime vs Size of the vector

Results :

1. In the roof-line model, the intersection of the two graphs(memory bound and compute bound) take place at a point of least ops/bytes but which can extract the peak performance. The roof-line model is plot on a log-log scale. The minimum ops/bytes which can extract peak performance is given by :

$$ops/bytes = \frac{4.29 * 10^{12}}{288 * 10^{9}} = 14.895$$

   This can also be observed from Figure 2. The y-intercept of the graph will be at $log(0.288)$ as the y-axis is plot in TFLOPS.

2. The optimal number of threads per block is observed to be 256. The hardware specifications of the streaming multiprocessor are such that it allows a maximum of 64 resident warps per processor which implies a maximum of 2048 threads per processor. Also, the maximum number of threads per block is 1024, hence the minimum number of in flight block as on a streaming multiprocessor will be 2. There can be more if less number of threads are defined per block. Our main goal is to reduce the memory accesses to the global memory and to utilize the shared memory on a SM to the fullest. For a small number of threads per block, the number of blocks to be defined will be very large and hence scheduling of those blocks will lead to a large overhead. Also, a large number of threads per block(512 and 1024) will lead to a small number of blocks but large number of warps and the scheduling of those warps will also lead to overheads. Considering the case of 128 or 256 threads per block, the number of blocks per SM would equate to 16 and 8 respectively. This coupled with the fact that there are 4 warp schedulers per SM, the number of warps per block would need to be 4*N, so that maximum performance is extracted. In the case of 256 threads per block, a set of 8 warps are defined hence occupying all the schedulers. Since these number of blocks completely utilize the SM, it leads to the best performance.

3. The optimal number of operations per thread was observed to be 4. We know that each thread executes on a core in the streaming multiprocessor and has its own registers. Since in our vector addition we have defined that the output is of type double, each addition corresponds to a total of 16 bytes. Now each thread has 64, 32-bit registers which need to be utilized. Some of this memory is required to hold the local variables of the threads and hence less memory is available than expected. Hence the ideal number of register memory utilized is found to be around 64 bytes which corresponds to 4 operations per thread.

4. As we increase the length of the vector uniformly, the run-time should increase linearly. However that is not the case observed in the GPU. This is because of the fact that the device is not completely utilized. Once it reaches its fully capacity, then we start to see a linear behaviour. Initially for sizes such as $2^{15}, 2^{16}$, all the SMs do not come into use completely and hence we dont see a linear nature in the graph. As the size of the vectors increases, then the shared memory is no longer able to hold the data and hence global memory calls become more frequent hence leading to stalls.

(collaborated with EE16B109 only for the reasoning)

Conclusion :

The performance of the Tesla K40C GPU was studied using the example of vector addition. The optimal thread and block configurations were obtained empirically by calculating the kernel-execution time.