Microbenchmarking NVIDIA GPUs

Zhantong Qiu, Jeonghoon Jo, Kaustav Goswami

I. Introduction

In this project report, we implement several micro-benchmark programs on an NVIDIA RTX 2060 using CUDA. In each of the following sections, we describe a micro-benchmark program, its implementation methodology and the observed results.

We divided the implementation of the micro-benchmarks among us in the following manner:

- Zhantong Qiu: Shared memory latency and shared cache scaling.
- Jeonghoon Jo: Compute
- Kaustav Goswami: Memory bandwidth and memory hierarchy.

II. Shared Memory Latency

We implemented a P-chase[5] based micro-benchmark to test the shared memory latency of RTX 2060 with a single block and different size of threads and strides, also known as the memory access distances.

The shared memory is an important hardware feature in GPU parallelism because it enables low latency memory access. It is heavily used in parallel programs and algorithms to optimize performance. However, with our micro-benchmark, we learned that bank conflicts in shared memory can increase the latency from twenty-two cycles to over five hundred cycles.

A. Methodology

Our micro-benchmark uses the concept of P-chase, which initialize the testing array with the indices of the next memory access as its elements. Our kernel takes in the testing array and reads the testing array a large number of times. Every read will update the next reading location in the testing array. By doing this, we can stride the cache and get the insight of the cache, such as the average cache read latency.

```
        for(int i = 0; i < arr_size; i ++)
        {
            arr[i]=(i+stride)%arr_size;
        }

```

Listing 1. Initialize Testing Array

```
        next = threadIdx.x * stride;
        for (int i = 0; i < num_iteration; i++)
        {
            next = arr[next];
        }

```

Listing 2. Run Testing Array

In our micro-benchmark, each thread initialize the variable **next** with the multiplication of its thread id and the stride number. Each thread reads the testing array the same number of times. Since the shared memory is private to each block, only one block should be used. In our experiment, we dynamically allocated 64KB shared memory for the kernel and ran the kernel repeatedly with different numbers of threads and strides.

B. Result and Discussion

Shared Memory Latency						
Thread/Stride	1	2	4	8	16	32
1	22	22	22	22	22	22
2	22	22	22	22	22	24
4	22	22	22	22	24	28
8	22	22	22	24	28	36
16	22	22	24	28	36	52
32	22	24	28	36	52	84
64	22	24	28	36	52	84
128	22	24	28	36	64	128
256	22	24	32	64	127	255
512	32	32	64	127	255	511
1024	62	62	125	251	504	N/A

TABLE I

Table of Shared Memory Latency in cycle(s). Because our testing array uses **int** as its datatype, 64KB of shared memory can only hold 16384 elements. The multiplication of 1024 and 32 is 32768, which exceeds the number of total elements in the testing array, so our kernel did not test for 1024 threads with 32 strides.

Bank Conflict Latency						
Bank Conflict	None	2-way	4-way	8-way	16-way	32-way
RTX2060	22	24	28	36	52	84

TABLE II

Table of Minimum Bank Conflict Latency in cycle(s).

In our early attempts of writing the micro-benchmark, we did not use P-chase and let each thread to access the testing array with the distance of the total threads

in used. The latency of the share memory access was remained about 22 cycles for 1 thread to 256 threads and jumped rapidly to 64 cycles for 512 threads in used. It was due to the bank conflicts in memory access. Bank conflicts happen when multiple threads in the same wrap are accessing the memory in the same bank. N threads in the same wrap accessing the same bank is called N-way bank conflict. RTX 2060 has 32 banks of shared memory so it has a maximum of 32-way bank conflict. In Table 2, it shows the latency of different bank conflict.

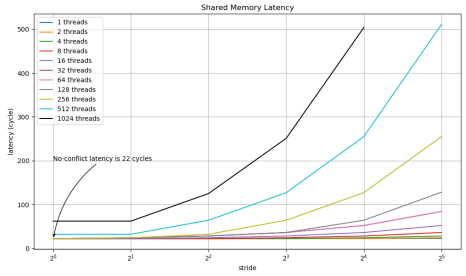

Fig. 1. Shared Memory Latency

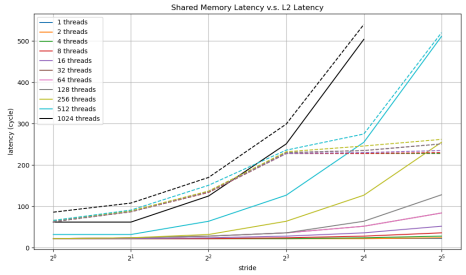

Fig. 2. Shared Memory Latency v.s. L1 Latency

Bank conflict is an important problem when using shared memory as the optimization for parallel programming. In figure 5, we performed the same stride to the L2 cache with the test size increased to 2MB and compared the hit latency between the shared memory and L2 cache. With the same number of threads in used, the shared memory has lower latency than the L2 cache, but it proves that when the shared memory is poorly managed, it will have a higher latency than better managed L2 cache. For example, the lowest latency of L2 cache with 1024 threads in used is 66 cycles. The highest latency we observed for shared memory with 1024 threads in used is 504 cycles. It is important for parallel computing programmers to keep in mind that poorly managed shared memory can have 7.8 times higher latency than well managed L2 cache access.

Our finding is important to parallel computing because we need to understand using shared memory does not conclude better performance. Since memory access is expensive in parallel computing, we as parallel computing programmers should understand and better managed our memory access. Beside our findings, our micro-benchmark has multiple options to configure standard tests for shared memory, shared caches, and global memory. In the future, we can easily use our micro-benchmark as the standardize test to compare the memory latency with other NVIDIA GPU models.

III. Shared Cache Scaling

We implemented a micro-benchmark specialized in testing cache bandwidth scaling. Shared cache scaling is analyzed by the shared cache bandwidth alongside with the growth in parallelism. The less parallelism the GPU needs to get to the higher shared cache bandwidth indicates better shared cache scaling. Unfortunately, we only tested our micro-benchmark on RTX 2060, so we are not able to compare its shared cache bandwidth with other GPU models, but we found usual insights about RTX 2060's shared cache scaling.

A. Methodology

In our micro-benchmark, we set the maximum number of blocks as 48 and the number of threads as 256 because we believe 48 blocks of 256 threads are enough parallelisms to show the shared cache scaling behavior of RTX 2060, and most GPUs can have 256 threads per block[1]. RTX 2060 has an L1 cache of 96KB and an L2 cache of 4MB[2], so with the test size of 2MB, we can observe the shared cache scaling of the RTX 2060.

```
    int index = blockIdx.x * blockDim.x +
threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for(int i = index; i < arr_size; i +=
stride) {
        res[i] = arr[i];
    }
    participated[index] = index < arr_size
? true : false;
```

Listing 3. Run Testing Arrays

We pass two arrays with the integer datatype into our testing kernel and initialize the starting index depending on the thread id.Depending on the testing cache type and total number of threads in used, we might have situation that total number of threads is larger than the total number of elements in our testing arrays. Therefore, we use a Boolean object to signal if the thread participated in memory access.

For bandwidth calculation, we used the total cycles all threads took to copy the 2MB of data to divide the total number of participated threads and the base clock of RTX 2060, which is 1365 MHz[2], to get the total second it took to finish read and store 2MB of data. Then we multiplied 2MB with 2 and divide the total second to get the bandwidth unit in byte/s. At the end, we converted the number into GB/s.

B. Result and Discussion

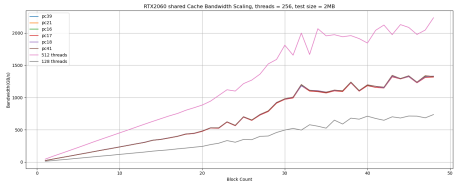

Fig. 3. Shared Cache Scaling

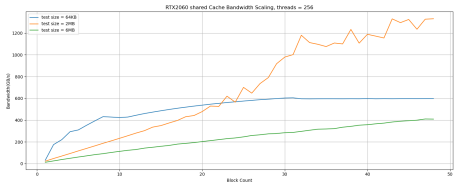

Fig. 4. Shared Cache Scaling with Different Test Sizes

Because we do not have another GPU model to compare with RTX 2060's cache scaling, we compared it with different configurations and different actual machines. In figure 3, we ran our micro-benchmark in 6 different machines that implemented RTX 2060 and compared their shared cache scaling. Overall, they have identical cache scaling behaviors. It indicates that the shared cache scaling is stable and predictable for RTX 2060. Then we compared the scaling between using 128 threads, 256 threads, and 512 threads. As the parallelism increased with the threads, the bandwidth increase in stable manner but the noise in the scaling also became more frequent and large. Because RTX 2060 has 30 SMs and the noise appeared more after 30 blocks, we assume the noisy behavior might be caused by the SM scheduling. Further investigation is needed to prove our hypothesises.

In figure 4, we set the number of threads in used as 256 and compared the shared cache scaling between test size. When test size equals to 64KB, it falls into the L1 cache. This configuration reaches to its reached a stable bandwidth at around 10 blocks. This graph proves the correctness of our micro-benchmark by showing the correct behavior of different memory structures.

This finding is important to parallel computing because we need to understand the GPU scaling to determine the best parallelism configuration for our parallel computing programs or algorithms. For example, in Figure 3, there is not a large bandwidth difference between using 35 blocks and 40 blocks with 256 threads. We can save the overhead of launching new blocks for the similar performance.

IV. Compute

We evaluate the floating-point performance in half (FP16), single (FP32) and double (FP64) precision and the integer performance. All timing measurements use the clock() function, which returns the value of per-multiprocessor counter that is incremented every clock cycle. We run one thread per kernel and each kernel or thread performs the operation 1,000 times. We measured the elapsed clock cycles by taking the average. This is highlighted in the Code Listing 4.

Figure 5 shows the latency of FP16, FP32, and FP64 for fused multiply add (FMA) and addition (ADD). The number of instruction sets for FP16 FMA, FP32 FMA, and FP64 FMA is the same, but their latencies have some differences. Theoretically, the latency of FP16 should be half of FP32 in the processing power (TFLOPS) aspect. However, our measurement showed that the latency of FP32 was a little bit higher than FP16, not half.

Figure 6 shows the latency of integer FMA, ADD, and multiplication. We found that most integer, half- and single-precision shows similar latencies, which is general in GPUs. We also noticed that the latency of integer multiplication is the same as ADD.

```
1    start_time = clock();
2    for(int i = 0; i < iter ; ++i){
3        y[i] += x[i] * y[i];
```

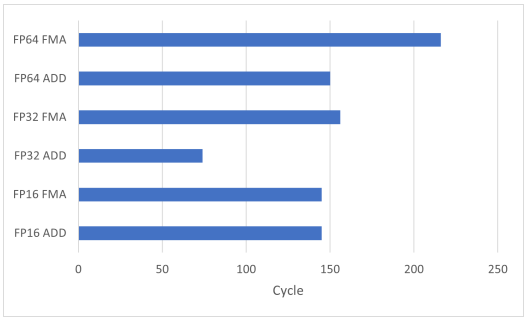

Fig. 5. Floating-point Execution Latency

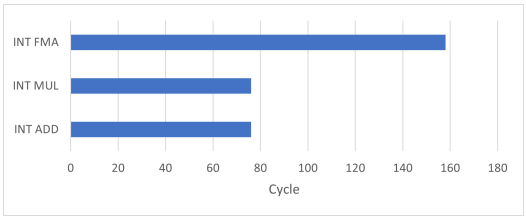

Fig. 6. Integer Execution Latency

```
    }
    stop_time = clock();
    elapsed_time = (stop_time - start_time)/
    iter;

```

Listing 4. Floating-point FMA: kernel function

V. Memory Bandwidth

We have implemented STREAM [7], [4], [8] on the GPU to reverse engineer the memory bandwidth of an NVIDIA GPU. It is considered a standard micro-benchmark program. STREAM works on 3 different vectors (A, B, C) with size N, and executes 4 functions namely:

- `COPY` is given by $C[i] = A[i]$
- `MULTIPLY` is given by $B[i] = \alpha \times C[i]$, where α is a large number.
- `ADD` is given by $C[i] = A[i] + B[i]$.
- `TRIAD` is given by $A[i] = B[i] + \alpha \times C[i]$, where α is a large number.

The order of executing the kernels is important to evict the cache. The order is same as the one stated above. We have implemented each of the above functions as a CUDA kernel. The objective here is to measure the memory bandwidth, which is highlighted in the Code Listing 5. The bandwidth and the execution time of each kernel observed across the four kernels are depicted in Figure 7. One of the limitations of STREAM micro-benchmark is that it reports bandwidth $>$ 80% of the theoretical peak [4].

```
cudaEventRecord(start);          // start
    measuring time
copy<<<numblocks, blocksize>>>(N, da, dc);  //
    call kernel
cudaDeviceSynchronize();         // synchronize
cudaEventRecord(stop);           // take another
     time measurement
bandwidth = (sizeof(da) * sizeof(da[0]) * 2)/(
    stop - start)/1e6;    // in GB/s
```

Listing 5. Bandwidth Calculation Approach

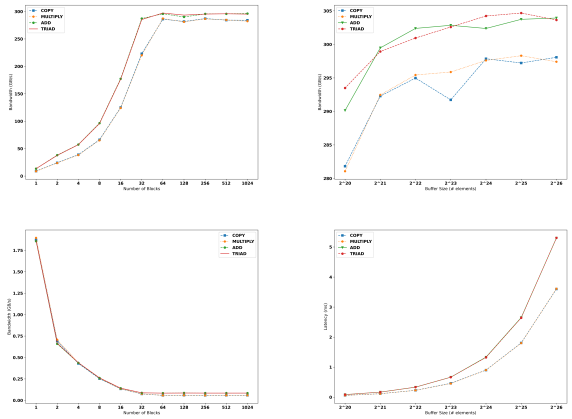

Fig. 7. Bandwidth observed by varying the number of threads per block (a), number of elements in the array (b). Execution times observed by repeating the same parameters are depicted in the latter figures ((c) and (d)).

We have varied two key parameters for each of the four kernels, namely the size of the array from 2^{20} elements to 2^{26} elements (Figure 7(b) and Figure 7(d) and, the number of threads per block from 1 to 1024 (Figure 7(a) and Figure 7(d)). The key observation is that bandwidth increases as we increase the number of threads or size. Both show a clear knee in the graph, where it peaks the available bandwidth. There are 2 memory operations for `COPY` and `MULTIPLY` and 3 memory operations for `ADD` and `TRIAD`. This is the reason why `ADD` and `TRIAD` shows higher bandwidth than `COPY` and `MULTIPLY`. Overall, the observed average bandwidth is 304 GB/s, which is 90% of the theoretical bandwidth of an NVIDIA RTX 2060 [5].

A. Potential Timing Side-Channel using Bandwidth

Laor *et al.* conducted a study towards fingerprinting GPUs based on timing channels [6]. The authors executed 10 GPU kernels and an on-screen render and

collected the overall execution times across several machines. The interesting finding that they made is the existence of clear timing side-channels. This can be exploited by an attacker to fingerprint GPU devices. Our hypothesis after reading this paper was that these timing side-channels can be exploited by executing any kernel on the GPU. Towards, this we leverage the execution times of each of the STREAM kernels as a potential for fingerprinting.

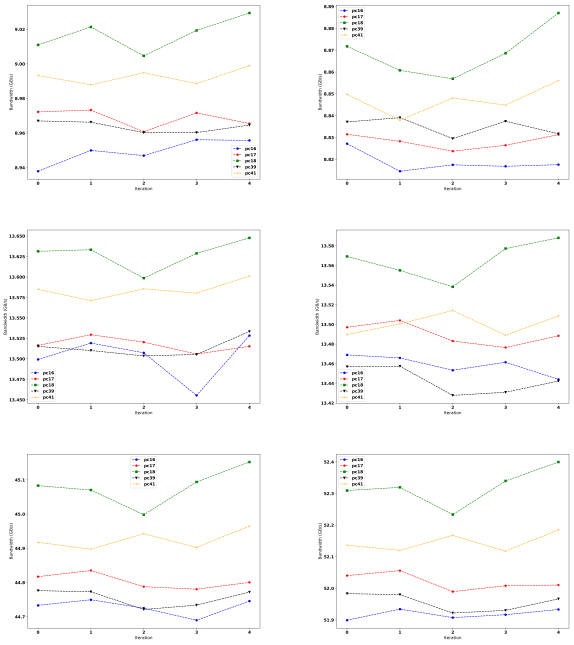

Fig. 8. Observed Bandwidth on the training set. The X-axis represent the run number and the Y-axis represents agg. bandwidth.

We execute each STREAM kernel for 20 times on 5 different machines equipped with NVIDIA RTX 2060. We run this experiment 5 times and report the observed bandwidth. The output is depicted in Figure 8. The top 4 graphs depict the execution times of each of the kernel in the 5 runs. We ran this experiment with one thread and one block ($<<< 1,1 >>>$). The bottom two figures show average and weighted average for the same. From the figure, we can clearly see that there exists some form of a timing channel, which can distinguish the 5 different machines. For instance, pc18 constantly has a higher bandwidth than the other machines. Therefore, pc18 does emit a clear signal for fingerprinting. We therefore, constructed another set for testing our claim. We did the same experiment again, and formed a test set. We designed a classifier by taking average readings of the training set and then ran our testing set against the same classifier. We achieved a fingerprinting accuracy of 88% on the test set. The output images of the test set is depicted in Figure 10.

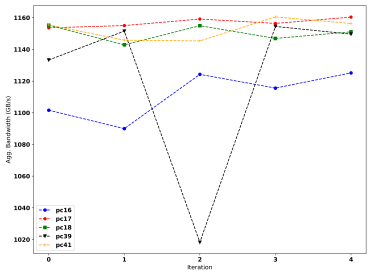

Fig. 9. Observed Average Bandwidth with 1024 threads. The X-axis represent the run number and the Y-axis represents agg. bandwidth.

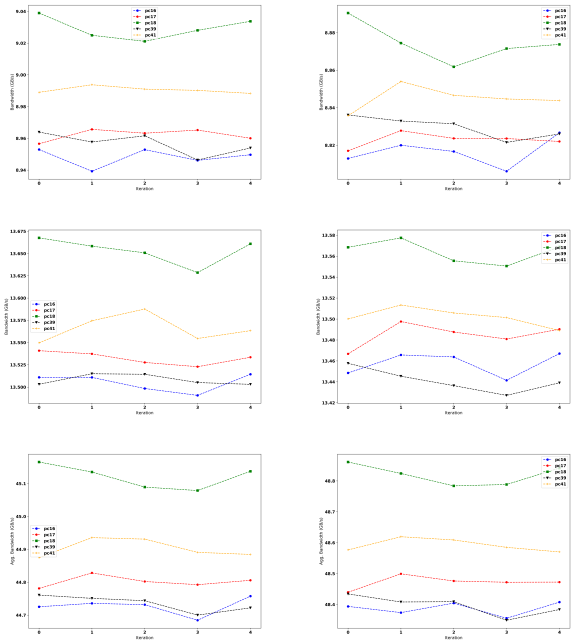

Fig. 10. Observed Bandwidth on the testing set. The X-axis represent the run number and the Y-axis represents agg. bandwidth.

When we tried to repeat the same experiment by increasing the number of threads and blocks, we did not observe the any timing side channels (Figure 9). One of the reason is that the scheduler plays a role in scheduling the memory operations during the data transfer. This is absent when there was only one thread. The achieved fingerprinting accuracy can be further improved by using

more sophisticated classifiers like random-forests, nai”ve Bayes *etc.*. We plan on investigating this phenomenon in the future with counter-measures like interference channels [9]. Laor *et al.* claims that this difference is due to process variation [3], which is another angle we plan to explore in the context of GPU fingerprinting.

VI. Memory Hierarchy

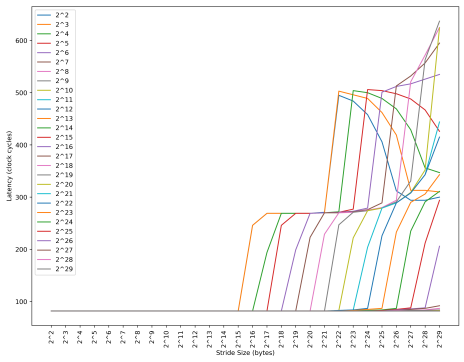

Fig. 11. Access Latency of the P-chase algorithm. On X-axis we have stride sizes in bytes and on the Y-axis, we have latency in terms of clock cycles.

We implemented P-chase algorithm on the GPU to perform strided accesses to the memory using one block and one kernel. The idea is to find a buffer size larger than a given memory hierarchy, so as to see a bump in latency when we stride the buffer. Our objective is to find the sizes of each of the caches present on the NVIDIA RTX 2060 GPU.

The output of the P-chase algorithm is depicted in Figure 11. Initially for smaller buffer sizes, we see no increase in terms of access latency. This continues until a buffer of size N does not fit into the cache, and, striding over it will result in an increase in latency. We observe the first bump when the stride is at 2^{16} Bytes, which suggests that the L1 cache has a capacity of 64 KB. We then observe another bump at 2^{22} Bytes, which suggests that the L2 cache has a capacity of 4 MB. We verified this with the finding of [5]. Therefore, we find out that the NVIDIA RTX 2060 GPU has a relatively simple architecture with 2 levels of caches. The data that is absent in the L2 cache initially will start fitting into the cache and the overall latency of the subsequent buffers will decrease until it completely fills up. After this, we should see increase in further buffer sizes. This behavior is observed for buffer sizes from 2^{22} Bytes to 2^{29} Bytes, where at buffer size 2^{25} Bytes, the latency starts to increase at L2.

References

[1] “Microbenchmarking Intel’s Arc A770.” [Online]. Available: https://chipsandcheese.com/2022/10/20/microbenchmarking-intels-arc-a770/

[2] “NVIDIA GeForce RTX 2060 TU104.” [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-rtx-2060-tu104.c3495

[3] S. Borkar, “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation,” *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[4] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, “Gpu-stream v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models,” in *High Performance Computing*, M. Taufer, B. Mohr, and J. M. Kunkel, Eds. Cham: Springer International Publishing, 2016, pp. 489–507.

[5] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, “Dissecting the nvidia turing t4 gpu via microbenchmarking,” 2019. [Online]. Available: https://arxiv.org/abs/1903.07486

[6] T. Laor, N. Mehanna, A. Durey, V. Dyadyuk, P. Laperdrix, C. Maurice, Y. Oren, R. Rouvoy, W. Rudametkin, and Y. Yarom, “DRAWN APART : A device identification technique based on remote GPU fingerprinting,” in *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022. [Online]. Available: https://doi.org/10.147222Fndss.2022.24093

[7] J. D. McCalpin, “Sustainable Memory Bandwidth in Current High Performance Computers,” https://www.cs.virginia.edu/~mccalpin/papers/bandwidth/bandwidth.html, 1995.

[8] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying gpu microarchitecture through microbenchmarking,” in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010, pp. 235–246.

[9] T. Yandrofski, J. Chen, N. Otterness, J. H. Anderson, and F. D. Smith, “Making Powerful Enemies on NVIDIA GPUs,” in *2022 RTSS*. IEEE, 2022. [Online]. Available: https://www.cs.unc.edu/~otternes/papers/rtss2022.pdf