Quantifying the Relationship Between Occupancy and Performance On CUDA

Fang Han Cabrera
Courant Institute of Mathematical Sciences
New York University
fh643@nyu.edu

Abstract

Occupancy is an important metric on Streaming Multiprocessor utilization in CUDA. Low occupancy oftentimes results in poor *performance*, but it remains unclear if occupancy always correlates with *performance* positively.

This paper tries to quantify the relationship between occupancy and performance on a set of benchmark applications, utilizing NVIDIA's Nsight Compute profiling tools[1]. It's the author's hope that this analysis will lead to better kernel optimization.

1 Introduction

The CUDA C Programming Guide (NVIDIA Corp. [2019]) defines *Occupancy* as the ratio of active warps on an Streaming Multiprocessor to the maximum number of active warps supported by the SM. Low occupancy could translate into poor instruction issue efficiency, but it remains unclear how occupancy would affect the overall performance of a kernel.

In this paper, I conducted experiments on a set of benchmarks included in the Cuda Toolkit 10.2 [2] in order to quantify how occupancy and kernel performance interact. Detailed profiling was done using NVIDIA's next-generation profiling tools.

The experiment results show that occupancy and performance are correlated but in a complex manner. Involved intimately in the outcome are the implementation details of each and every kernel as well as the underlying hardware configuration. Nevertheless, insights are gained on how to fine tune kernel performance.

Previous Work Bakhoda, Ali et al.(2008) raised the question *Are More Threads Better* and tried to answer it by measuring the speedups brought by varying the number of CTAs [3] on a set of benchmarks. They observed widely-varying workload-dependent behavior and concluded that always scheduling the maximal number of CTAs supported by a shader core is not always the best scheduling policy.

Even though the author of this paper agrees with Bakhoda, Ali et al.(2008) in principle, improvements upon their research can be made in several aspects:

- A better definition of occupancy would be beneficial. The number of CTAs doesn't always translate to occupancy.
- Speedup on benchmarks is not a detailed quantification of kernel performance.

This paper tries to improve upon the existing findings by addressing in more detail the above-mentioned drawbacks.

2 Interpreting Occupancy

Directly tied to occupancy is *instruction issue efficiency*, which can be measured as `IPC` (instruction per cycle). To understand the mechanism behind it we need to take a deeper look at how warps help hide latency on Streaming Multiprocessors.

2.1 Warps and the SIMD model

An SM is designed to execute all threads in a warp following the SIMD model [4]. At any instant in time, one instruction is fetched and executed for all threads in a warp. As shown in Figure 1, an SM has a single instruction fetch/dispatch unit shared among execution units (CUDA cores). These threads will apply the same instruction to different portions of the data. Consequently, all threads in a warp will always have the same execution timing.

[1]Nsight Compute User Manual *https://docs.nvidia.com/nsight-compute/NsightCompute*

[2]CUDA Toolkit 10.2 https://docs.nvidia.com/cuda/cuda-samples

[3]coorperative thread arrays

[4]The CUDA Parallel Programming Model - 2. Warps *https://nichijou.co/cuda2-warp/*

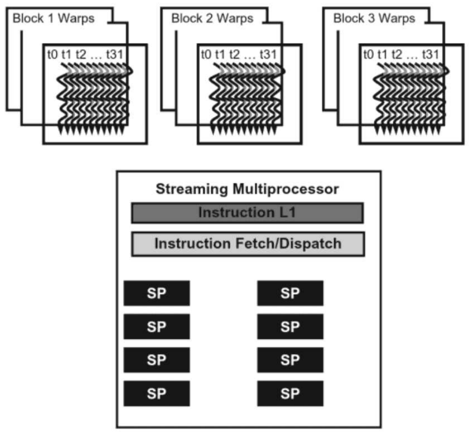


Figure 1: An SM serving lots of warps at the same time [6]

2.2 Latency Tolerance and Zero-Overhead Thread-Scheduling

Unlike CPUs, GPUs do not dedicate as much chip area to cache memories and branch prediction mechanisms. This is because GPUs have the ability to tolerate long-latency operations.

GPU SMs are designed in the way that each SM can execute only a small number of warps at any given time. However, the number of warps residing on the SM is much bigger than what can actually be executed. The reason for this is, when a warp is currently waiting for result from a long-latency operation, such as:

- global memory access
- floating-point arithmetic
- branch instructions

the warp scheduler on the SM will pick another warp that's ready to execute, therefore avoiding idle time. By having a sufficient number of warps on the SM, the hardware will likely to find a warp to execute at any point in time.

Having zero idle time or wasted time is referred to as zero-overhead thread scheduling in processor designs.

2.3 Occupancy – A Double-Edged Sword

Now we can understand why low occupancy is likely to result in poor instruction issue efficiency: there are not enough eligible warps to hide latency.

However, as we will see in Section 6.2, after passing the sufficiency threshold to hide latency, increasing the number of warps further may degrade performance because *resource contention* amongst threads rises. This is why occupancy tuning can be a critical part to kernel optimization.

Theoretical Occupancy Theoretical occupancy refers to the upper limit for active warps which is dependant upon the hardware as well as launch configuration and kernel compile options [7].

A block is **active** from the time its execution begins until all warps in the block have exited from the kernel.

The upper limit for active warps is the computed as:

$$\frac{\texttt{upper limit for active blocks}}{\texttt{number of warps per block}} \tag{1}$$

Limiting Factors The upper limit for active warps is limited by several factors:

- Warps per SM: the maximum number of warps that can be active at once on an SM.
- Blocks per SM: the maximum number of blocks that can be active at once on an SM.
- Registers per SM: a set of registers shared by all active threads. If there aren't enough registers for all threads, occupancy will be reduced in granularity of blocks.
- Shared memory per SM: a fixed amount of shared memory shared by all active threads. If there aren't enough shared memory for all threads, occupancy will be reduced in granularity of blocks.

Achieved Occupancy The true number of active warps varies over the duration of the kernel, as warps begin and end. Achieved occupancy is measured on each warp scheduler using hardware performance counters to count the number of active warps on that scheduler every clock cycle. The formula for achieved occupancy is given as Equation (2):

$$\frac{\texttt{sum}_{\texttt{warp scheduler}}(\texttt{active warps})}{\texttt{max num of active warps supported}} \tag{2}$$

3 Performance and Its Measurement

3.1 Defining Performance

Lots of candidate metrics were taken into consideration to best represent kernel's performance. For example:

[7]Nsight Visual Studio Edition 4.8 Documentation

- Data throughput: total data transfers (both read and write) of a kernel.
- Computational throughput: total calculations performed by the device per second (Gflop/sec).
- L1/L2 hit rates: the percentage of cache hits to the total data requests to L1/L2 cahces.

A set of metrics were finally chosen to characterize kernel's performance from two most important aspects:

- **Memory locality utilization**: represented by `L1/L2-cache hit rates`. Figure 2 shows an example of CUDA's memory architecture.
- **Instruction issue efficiency**: primarily `IPC (instruction per cycle)`, aided by *warp statistics* such as `warp cycle per instruction` and `no-eligible-warp rate`.
- **Tail effect minimization**: quantified by the `waves per SM` metric.

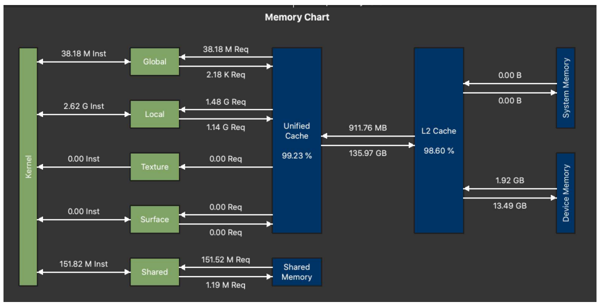


Figure 2: Memory chart of `nbody` simulation with block size of 128, showing a typical L1/L2-cache enabled CUDA memory architecture.

Details on these metrics are discussed in **Section 3.3**.

3.2 Profiling tools

The NVIDIA Nsight Compute is described as the next-generation profiling tool. It profiles CUDA kernels interactively and provides detailed customizable performance metrics(see 3.3) and API debugging via a beautiful user interface as well as command line tool. Furthermore, Nsight Compute offers performance analysis scripts lending insights into key aspects of the application based upon post-processing results.

For the above reasons, I've decided to use Nsight Compute as my profiling tool.

3.3 Metrics

The metrics below were extracted from Nsight post-processing results for analysis:

Occupancy Actual achieved occupancy.

Waves Per SM When the GPU launches a grid of threads for a kernel, that grid is divided into *waves* of thread blocks. The size of a wave depends on the number of SMs on the GPU and the theoretical cccupancy of the kernel. The last wave under-utilized the GPU but represented a significant fraction of the run time. This load imbalance issue known as **tail effect** can be quantified by looking at the fraction of this `waves per SM` metric. For instance, 2.35 would indicate a under-filled third wave, whereas 2.83 would be an almost-full third wave. The former has tail effect and the latter not.

Tail effect may play an important role when the number of blocks executed for a kernel is small.

IPC Instruction per cycle.

Warp Cycle Per Instruction The warp cycles per instruction defines the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. `Warp Cycle Per Instruction` is the average number of cycles spent in that state per issued instruction.

No Eligible Warp Each scheduler maintains a pool of warps that it can issue instructions for. On every cycle each scheduler checks the state of the allocated warps in the pool. Active warps that are not stalled (eligible warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions. On cycles with *no eligible warps*, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates *poor latency hiding.*

L1/L2 hit rate L1-hit rate and L2-hit rate reflect the utilization of memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units, exhausting the available communication bandwidth between those units, or by reaching the maximum throughput of issuing memory instructions.

4 Hardware Configuration

Experiments were conducted on two CUDA-capable machines. The machines are *not* clus-

Table 1: Comparing Hardware Configurations Between `GeForce GTX 1080` and `GeForce RTX 2080 Ti`

Item	GTX 1080	RTX 2080 Ti
CUDA Capability Major/Minor version number	6.1	7.5
Multiprocessors	20	68
CUDA Cores/MP	128	64
Total number of CUDA Cores	2560	4352
Warp size	32	32
Max number of threads/multiprocessor	2048	1024
Max number of threads/block	1024	1024
Max dimension size of a thread bloc (x, y, z)	(1024, 1024, 64)	(1024, 1024, 64)
Max dimension size of a grid size (x, y, z)	(2147483647, 65535, 65535)	(2147483647, 65535, 65535)
Memory Bus Width	256-bit	352-bit
L2 Cache Size	2097152 bytes	5767168 bytes
Total amount of constant memory	65536 bytes	65536 bytes
Total amount of shared memory per block	49152 bytes	49152 bytes
Total number of registers available per block	65536	65536
Device supports Unified Addressing (UVA)	Yes	Yes
Supports Cooperative Kernel Launch	No	No
Supports MultiDevice Co-op Kernel Launch	No	No

tered and each of them only has one NVIDIA GPU – `GeForce RTX 2080 Ti` and `GeForce GTX 1080` respectively. I'll specify the machine each of the benchmarks are associated with in section 5.

Detailed configurations of both GPUs are listed in Table 1.

Workload Distribution As we can see from the configurations in Table 1, the main differences between `GeForce RTX 2080 Ti` and `GeForce GTX 1080` are:

- `GeForce RTX 2080 Ti` has 0.7 times more cores
- L2 Cache in `2080 Ti` is more than twice the size of that in `GTX 1080`

Due to practical constraints, I couldn't run all experiments on `GeForce RTX 2080 Ti`. However, more computational intensive applications such as `Nbody Simulation` were run on the superior hardware for two reasons:

- Experiment results may be more relevant assuming advanced applications are more likely to run on higher-end devices
- `GTX 1080` was proven to have a hard time running `Nbody` and `Monte Carlo` at all

5 Benchmarks

My benchmarks are listed in Table 2 along with their main application properties. These benchmarks are selected from NVIDIA's CUDA Toolkit 10.2 (formerly known as CUDA SDK) [8].

Brief overview of each benchmark is provided below:

binomialOptions A computational finance application that evaluates fair call price for a given set of European options under binomial model ([2013] Podlozhnyuk). Its minimum spec requirement is SM 3.0.

`binomialOptions` experiments were conducted on `GTX 1080`.

BlackScholes The pricing of options is a very important problem encountered in financial engineering since the creation of organized option trading in 1973. `BlackScholes` is another computational finance application ([2013] Podlozhnyuk). It evaluates fair call and put prices for a given set of European options by Black-Scholes formula.

`BlackScholes` experiments were conducted on `GTX 1080`.

quasirandomGenerator This application implements Niederreiter Quasirandom Sequence Generator and Inverse Cumulative Normal Distribution functions for the generation of Standard Normal Distributions.

`quasirandomGenerator` experiments were conducted on `GTX 1080`.

[8]https://docs.nvidia.com/cuda/cuda-samples

Table 2: Benchmark Properties

Benchmark	Block Size	#Threads	Instructions Issued	Shared Memory?	L1 cache?	L2 cache?
binomialOptions	128	131072	8128610872	no	no	yes
BlackScholes	128	2000000	107433866	no	no	yes
quasirandomGenerator	(128, 3, 1)	49152	132137236	no	no	yes
fastWalshTransform	256	2097152	16778576	no	no	yes
MonteCarloMultiGPU	256	696320	9193224660	no	yes	yes
jacobiCudaGraphs	(256, 1, 1)	16896	12996882	no	yes	yes
nbody	256	69632	87397036992	yes	yes	yes

fastWalshTransform Walsh transforms belong to a class of generalized Fourier transformations. They have applications in various fields of electrical engineering and numeric theory. This benchmark is an efficient implementation of naturally-ordered Walsh transform (also known as Walsh-Hadamard or Hadamard transform) in CUDA and its particular application to dyadic convolution computation.

`fastWalshTransform` experiments were conducted on `GTX 1080`.

MonteCarloMultiGPU As more computation has been applied to finance-related problems, finding efficient implementations of option pricing models on modern architectures has become more important. This benchmark evaluates fair call price for a given set of European options using the Monte Carlo approach, taking advantage of all CUDA-capable GPUs installed in the system.

`MonteCarloMultiGPU` experiments were conducted on `RTX 2080 Ti`.

jacobiCudaGraphs This benchmark implemented CUDA Graph Update with Jacobi Iterative Method using `cudaGraphExecKernelNodeSetParams()` and `cudaGraphExecUpdate()` approach.

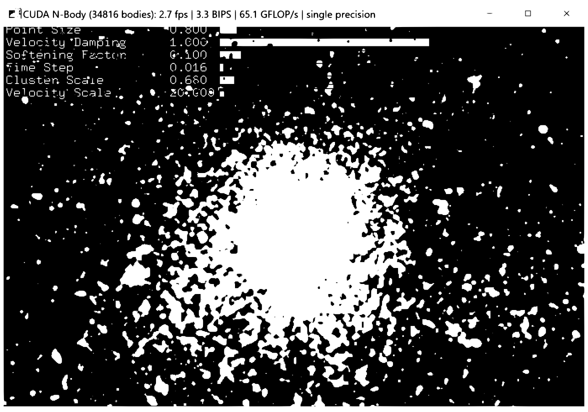
`jacobiCudaGraphs` experiments were conducted on `RTX 2080 Ti`.

nbody An N-body simulation numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body. A familiar example is an astrophysical simulation in which each body represents a galaxy or an individual star, and the bodies attract each other through the gravitational force (Figure 3).

The all-pairs approach to N-body simulation is a brute-force technique that evaluates all pair-wise interactions among the N bodies. It is a relatively simple method, but one that is not generally used on its own in the simulation of large systems because of its $O(N^2)$ computational complexity. Instead, the all-pairs approach is typically used as a kernel to determine the forces in close-range interactions. The all-pairs method is combined with a faster method based on a far-field approximation of longer-range forces, which is valid only between parts of the system that are well separated.

The all-pairs component of the algorithms requires substantial time to compute and is therefore an interesting target for acceleration.

`nbody` experiments were conducted on `RTX 2080 Ti`.


CUDA N-Body (34816 bodies): 2.7 fps | 3.3 BIPS | 65.1 GFLOP/s | single precision

```
Point Size          0.800
Velocity Damping    1.000
Softening Factor    0.100
Time Step           0.016
Cluster Scale       0.680
Velocity Scale      20.000
```

Figure 3: An Exciting Nbody Simulation.

6 Experiments and Results

6.1 Setup

This paper aims to explore the effects of occupancy on kernel performance. With this purpose, I varied the block size for each benchmark within the SM's limit in order to achieve different occupancy, while keeping other kernel specs intact. Section 6.2 shows the results of a series of experiments with an emphasis on profiling **occupancy** against two key metrics that indicate **memory**

locality utilization and **computational performance** respectively:

- Cache hit
- IPC

6.2 Results

6.2.1 binomialOptions

As shown in Figure 4, significantly better occupancy was achieved when block size was changed from 128 in the benchmark to higher numbers in $\{512, 1024\}$. Going hand in hand with improved occupancy are increased IPCs. No significant change in memory access performance was found.

This experiment shows how improved occupancy has a positive effect on instruction issue efficiency.

Detailed experiment data is recorded in Table 3.

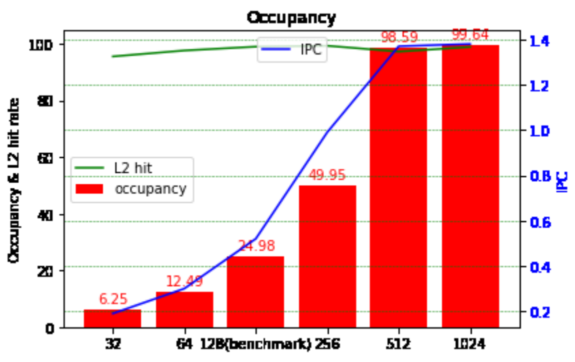


Figure 4: `binomialOptions`

6.2.2 BlackScholes

Figure 5 shows that benchmark achieved the highest computational performance (IPC) amongst all experiments. It can be seen that when block size is increased from benchmark, even though occupancy is kept at a satisfactory level (above 90%), performance still drops significantly, which indicates that resource contention might have happened.

Detailed experiment data is recorded in Table 4.

6.2.3 quasirandomGenerator

As shown in Figure 6, occupancy peaked at benchmarked block size of $384 = (128, 3, 1)$. However, at block size of $768 = (256, 3, 1)$, both L2-hit rate and IPC improved, even though its occupancy suffered a 10% loss from benchmark.

Experiment data is recorded in Table 5.

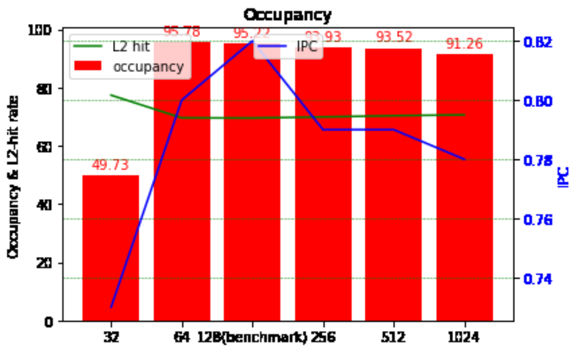


Figure 5: `BlackScholes`

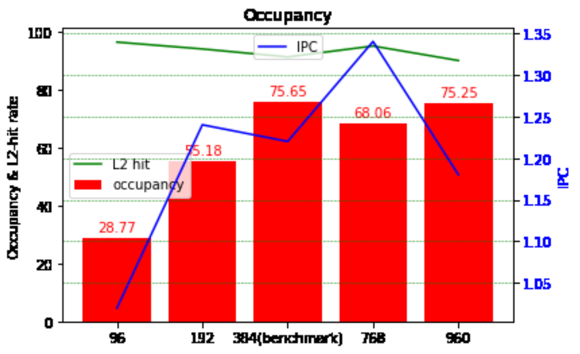


Figure 6: `quasirandomGenerator`

6.2.4 fastWalshTransform

Figure 7 exhibits similar pattern to `BlackSholes` where high occupancy was achieved across experiments but IPC still suffers great loss when block size is increased from benchmark. This is a clear sign that resources are at contention regardless of occupancy.

Experiment data is recorded in Table 6

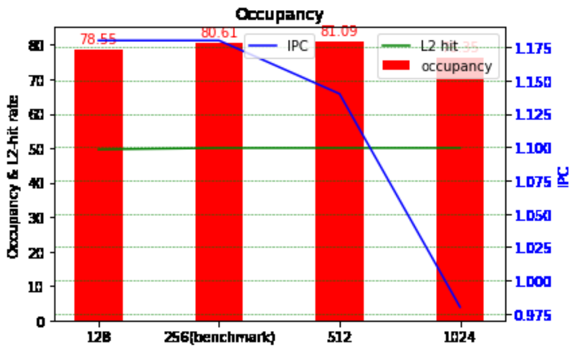


Figure 7: `fastWalshTransform`

6.2.5 MonteCarloMultiGPU

Figure 8 shows uniformed results from all experiments.

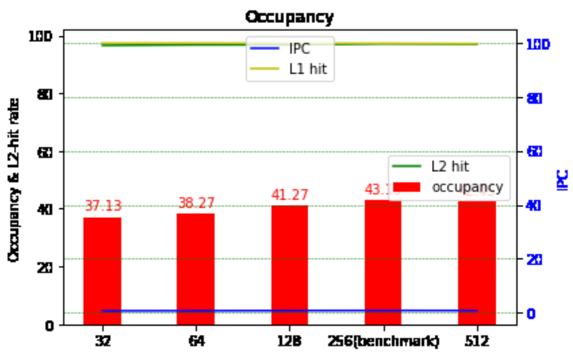


Figure 8: `MonteCarlo`

Experiment data is recorded in Table 7.

6.2.6 jacobiCudaGraphs

This set of experiments demonstrate interesting patterns. What we can see in Figure 9 is even though occupancy varies vastly, IPC stays constant. On the other hand, L1-hit and L2-hit show inverse correlation.

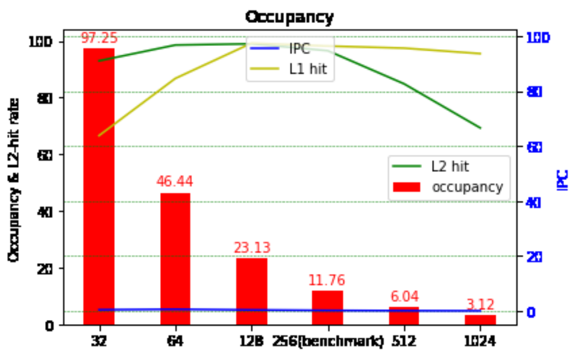


Figure 9: jacobiCudaGraphs

To better illustrate the interaction amongst the 4 metrics mentioned above, I also included a scatter plot in Figure 10 which should be read as the following:

- occupancy: represented by mark size
- IPC: color bar
- L1-cache hit: x-axis
- L2-cache hit: y-axis

Clearly, the highest occupancy in this case brings neither the best memory performance nor the best computational performance.

Experiment data is recorded in Table 8.

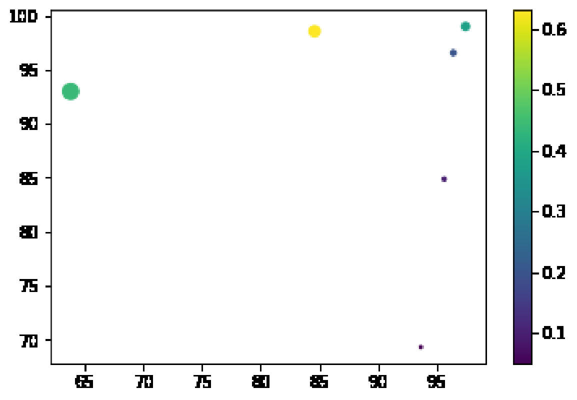


Figure 10: `jacobiCudaGraphs` scatter plot. x-axis: L1-hit; y-axisL L2-hit; color bar: IPC; mark size: occupancy

6.2.7 nbody

Similar to `MonteCarloMultiGPU` and `jacobiCudaGraphs`, here the drastic changes in occupancy do not bring better computational performance, as shown in Figure 11. Different from the former experiments, however, is that the highest occupancy brought both the lowest L1-hit rate and L2-hit rate. This can be observed more clearly in Figure 12.

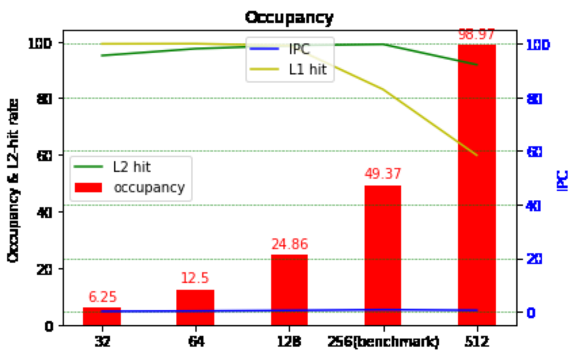


Figure 11: `nbody`

The `nbody` experiment data is recorded in Table 9.

7 Conclusion

From the series of experiments, I've come to the following conclusions:

- For small compute-bound kernels, e.g. `binomialOptions` (see Section 6.2.1), optimizing on occupancy could bring significant improvement on computational performance.
- When kernels require lots of resources from the SM, be careful not to push thread paral-

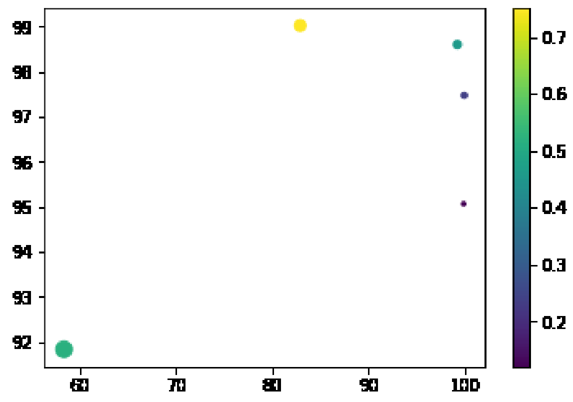


Figure 12: `nbody` scatter plot. x-axis: L1-hit; y-axisL L2-hit; color bar: IPC; mark size: occupancy

lelism too far. The fine balance between occupancy optimization and resource utilization needs to be decided on a case-to-case basis.

- No clear pattern was discover to rule on the correlation between occupancy and memory access performance, which means that in practice we need to:
  - scrutinize the implementation details of the kernel, in particular its data accessing patterns
  - use techniques like tiling, coalescing to optimize data access pattern
- There are lots of factors at work when it comes to kernel performance, which remains the topic for further research. For instance, data on `waves per SM` was collected in this paper but I didn't explore into the working of tail effect within the context of occupancy and performance.

References

[2008] Nyland, Lars; Harris, Mark; Prins, Jan. Fast N-Body Simulation with CUDA. *GPU Gems 3*

[2008] Podlozhnyuk, Victor. Fast Walsh Transform in CUDA. *CUDA-Toolkit-10.2, Code Samples*

[2013] Podlozhnyuk, Victor. Black-Scholes option pricing. *CUDA-Toolkit-10.2, Code Samples*

[2013] Podlozhnyuk, Victor. Binomial option pricing model. *CUDA-Toolkit-10.2, Code Samples*

[2013] Podlozhnyuk et al. Monte Carlo Option Pricing. *CUDA-Toolkit-10.2, Code Samples*

[2019] NVIDIA Corporation. CUDA C Programming Guide.

[2009] Bakhoda, Ali et al. Analysing CUDA Workloads Using a Detailed GPU Simulator. *979-1-4244-4184-6/09 IEEE*

[2019] Lew, Jonathan. et al. Analyzing Machine Learning Workloads Using a Detailed GPU Simulator*arXiv:1811.08933v2 [cs.DC]*

[2016] David Kirk; Wen-mei Hwu. Programming Massively Parallel Processors, 3rd Edition *O'Reilly Media*

Table 3: `binomialOptions` Experiment Result where best-performing experiment is marked as **bold**

`binomialOptions`	exp 1	exp 2	benchmark	exp 4	exp 5	exp 6
occupancy	6.25	12.49	24.98	49.95	98.59	**99.64**
block size	32	64	128	256	512	1024
waves per SM	1.60	1.60	3.20	6.40	**12.80**	25.60
IPC	0.19	0.30	0.52	0.99	1.37	**1.38**
warp cycle per instruction	**20.33**	26.39	29.89	31.45	45.27	45.29
no eligible warp	95.09	92.42	86.68	75.07	65.17	**64.84**
L2 hit	95.46	97.56	98.91	**99.23**	97.13	98.91

Table 4: `BlackScholes` Experiment Result where best-performing experiment is marked as **bold**

`BlackScholes`	exp 1	exp 2	benchmark	exp 4	exp 5	exp 6
occupancy	49.73	**95.78**	95.22	93.93	93.52	91.26
block size	32	64	128	256	512	1024
waves per SM	97.66	48.83	48.83	48.83	48.84	**48.85**
IPC	**0.73**	0.80	0.82	0.79	0.79	0.78
warp cycle per instruction	**43.38**	77.86	73.49	75.29	75.53	76.46
no eligible warp	81.44	79.42	79.54	80.18	79.91	**79.24**
L2 hit	**77.36**	69.57	69.52	69.89	70.32	70.61

Table 5: `quasirandomGenerator` Experiment Result where best-performing experiment is marked as **bold**

`quasirandomGenerator`	exp 1	exp 2	benchmark	exp 4	exp 5
occupancy	28.77	55.18	**75.65**	68.06	75.25
block size	96	192	382	768	960
waves per SM	0.30	**0.64**	1.28	3.20	1.60
IPC	1.02	1.24	1.22	**1.34**	1.18
warp cycle per instruction	**17.23**	24.43	33.60	28.99	32.55
no eligible warp	73.13	63.01	64.47	62.70	**62.08**
L2 hit	**96.31**	93.95	91.20	94.97	89.96

Table 6: `fastWalshTransform` Experiment Result where best-performing experiment is marked as **bold**

`fastWalshTransform`	exp 1	benchmark	exp 3	exp 4
occupancy	78.55	80.61	**81.09**	76.35
block size	128	256	512	1024
waves per SM	51.20	51.20	45.08	51.20
IPC	**1.18**	**1.18**	1.14	0.98
warp cycle per instruction	**42.22**	43.48	45.08	49.46
no eligible warp	70.38	**70.49**	71.18	74.90
L2 hit	49.64	50.05	50.06	**50.07**

Table 7: `MonteCarloMultiGPU` Experiment Result where best-performing experiment is marked as **bold**

`MonteCarloMultiGPU`	exp 1	exp 2	exp 3	benchmark	exp 5
occupancy	42.45	43.15	41.27	38.27	37.13
block size	32	64	128	256	512
waves per SM	2.5	5	10	20	40
IPC	0.85	**0.88**	0.84	0.78	0.75
warp cycle per instruction	**15.56**	**15.56**	15.58	15.61	15.67
no eligible warp	78.06	**77.69**	78.72	79.51	78.26
L1 hit	99.55	99.66	99.67	**99.74**	99.72
L2 hit	97	**97.10**	96.75	96.74	96.55

Table 8: `jacobiCudaGraphs` Experiment Result where best-performing experiment is marked as **bold**

`jacobiCudaGraphs`	exp 1	exp 2	exp 3	benchmark	exp 5	exp 6
occupancy	3.12	6.04	11.76	23.13	46.44	**97.25**
block size	32	64	128	256	512	1024
waves per SM	0.06	0.06	0.12	0.24	0.49	**0.97**
IPC	0.05	0.10	0.20	0.38	**0.63**	0.44
warp cycle per instruction	18.02	17.68	17.45	**17.37**	20.72	60.21
no eligible warp	94.45	94.43	94.26	88.64	**81.06**	86.96
L1 hit	93.65	95.64	96.40	**97.45**	84.60	63.82
L2 hit	69.29	84.86	96.54	**98.98**	98.54	92.97

Table 9: `nbody` Experiment Result where best-performing experiment is marked as **bold**

`nbody`	exp 1	exp 2	exp 3	benchmark	exp 5
occupancy	6.15	12.50	24.85	49.37	**98.97**
block size	32	64	128	256	512
waves per SM	0.25	0.25	0.50	**1**	**2**
IPC	0.12	0.24	0.46	**0.75**	0.52
warp cycle per instruction	16.44	**16.43**	16.62	19.45	55.50
no eligible warp	93.92	93.05	88.03	**76.69**	85.72
L1 hit	99.89	**99.95**	99.23	82.90	58.36
L2 hit	95.06	97.47	98.60	**99.02**	91.83