G-DBSCAN: Enhanced implementation

Massimo Frasson, 941703

July 16, 2021

1 Introduction

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters).[4]

To ensure consistency with [1], in this paper, the chosen similarity metric is euclidean distance.

A popular algorithm to perform clustering is DBSCAN[3]. Consider a set of points in some space to be clustered. Let ϵ be a parameter specifying the radius of a neighborhood for some point. For DBSCAN clustering, the points are classified as core points, reachable points, and outliers, as follows:

- A point p is a *core* point if at least minPts points are within distance ϵ of it (including p).
- A point q is *directly reachable* from p if point q is within distance ϵ from core point p. Points are only said to be directly reachable from core points.
- A point q is *reachable* from p if there is a path $p_1, ..., p_n$ with $p_1 = p$ and $p_n = q$, where each p_{i+1} is directly reachable from p_i. Note that this implies that the initial point and all points on the path must be core points, with the possible exception of q.
- All points not reachable from any other point are *outliers* or *noise* points.

Now, if p is a core point, then it forms a cluster together with all points (core or non-core) that are reachable from it. Each cluster contains at least one core point; non-core points can be part of a cluster, but they form its "edge" since they cannot be used to reach more points.[4]

2 G-DBSCAN

As per [1], we can treat the reachability among points problem by translating it into a graph search problem.

Each point directly becomes a node. An edge is created only if the distance between a couple of nodes is less than *epsilon*. Finally, starting from each core node not assigned to a cluster, we find all its reachable nodes. A core node and its reachable points are then assigned to the same cluster.

We can recognize two different phases: *graph construction* and *graph search*.

2.1 Data structures

Since euclidean distance is commutative, we deal with an undirected graph $G = (V, E)$. Each vertex has an unique index $v \in [0, |V|)$. In [1], the vector Va contains at position v two values: vertex v's degree and vertex v's adjacency list begin index. Adjacency list Ea is a vector. At v's adjacency list begin index it has a sequence of $v.degree$ integers. Those are v's adjacent nodes indices.

In my implementation, Va has been split into two vectors: *degrees* and *adjListIx*. I saw no point in building a complex data structure such as Va.

2.2 Graph construction

2.2.1 Vertices degree calculation

The main idea is to create a thread for each vertex. Each thread will scan all other vertices, compute the relative distance, and count how many are below the threshold. Then, the resulting value, the degree, is stored in Va (*degrees* array in my implementation).

2.2.2 Calculation of the adjacency list indices

The adjacency list of vertex 0, begins at index 0. For vertex 1, it begins at $degrees[0]$. For vertex k, it begins at $\sum_{i}^{k} degree[i]$. This operation is pretty common in parallel programming. It's called *prefix sum*.

Thrust library from NVIDIA provides an efficient implementation named *exclusive scan*.

2.2.3 Assembly of adjacency list

Like the calculation of the degrees, we create a thread for each vertex. Each will do the same operation as 2.2.1, but instead of incrementing the degree counter, it fills the Ea vector starting from its adjacency list.

2.3 Graph search

The main idea is to run sequentially a BFS for each core node not assigned to a cluster yet. Each node visited during the search is assigned to the same cluster of the search root.

The parallelism here lies in the BFS itself. The approach is derived by [2]. There are two arrays: the visited nodes X_a and the current frontier Fa. They are both of size $|V|$. As before, we create a thread for each vertex. For each step of BFS, each thread sets itself as visited and removes itself from the frontier.

Then it loads in the frontier all its adjacent nodes. When the kernel terminates, the host checks if the frontier is empty, if not it invokes another step of BFS.

We the host declares the BFS finished, each visited node not assigned to a cluster yet, is assigned to the current one.

3 Implementation

3.1 Vertices degree calculation

As explained in 2.2.1, each vertex needs to pull from memory its coordinates. To do so, each thread needs a reference to the dataset. In my implementation, the dataset, a matrix $n\ x\ d$, is linearized in an array. The values of each dimension are contiguous. It means that to access all the coordinates of a point, you need to access d locations far n from each other.

Linearizing as described above is optimal because each thread asks for a coordinate at the time. Concurrent threads in the same warp access to contiguous memory cells in the same memory transaction.

In [1], they support only 2-dimensional points. It allows the authors to use two registers to store the vertex coordinates assigned to the thread. In my implementation d-dimensional points are supported. I stored in shared memory the coordinates since a complex data structure in a thread would be stored in local memory. Local memory is off the device. The access would be significantly slower. Of course, for the most frequently used data in the thread, that was not an option.

In my implementation, the degrees calculation kernel takes as a parameter the squared threshold. This is because compute degrees, for each a point $x_k \in X$ and the thread coordinates x_t computes the following:

$$\sqrt{\sum_{i=0}^{d} |x_{ki} - x_{ti}|^2} <= threshold \tag{1}$$

Computing the square root at each iteration would be a waste of resources since we don't need the exact value, but only the truth value. So, working with *squaredThreshold* allows us to compute equivalently:

$$\sum_{i=0}^{d} |x_{ki} - x_{ti}|^2 <= squaredThreshold \tag{2}$$

The kernel is launched in the largest block possible. It means the minimum between the *maxThreadsPerBlock* the device supports and the maximum number of points the shared memory can fit. The latter depends on the number of dimensions d given in the input.

3.1.1 Attempted optimizations

Each thread goes through the entire dataset to compute the degree. In practice, if each thread starts iterating from index 0, at every iteration, all the threads in the warp access the same memory address. It results in suboptimal memory bandwidth utilization. The attempted optimization is to shift the access to the dataset by *threadIdx.x*. The for would go like:

```
for (int j = threadIdx.x; j < n + threadIdx.x; j++)
       {
              int item = j % n;
              ...
```

Hypothetically also cache hits should rise. In most cases, a thread with *threadIdx.x* finds the memory location already loaded in cache by the thread *threadIdx.x + 1*. However, as reported in 4, the performances worsen with this "optimization". I hypothesize that the bottleneck in the floating-point pipeline reported in 5 keeps the thread busy hiding any performance improvement due to memory accesses. If this is true, computing the module is pure overhead.

The coordinates of each thread are stored in shared memory. It means that a portion of the dataset of BLOCK_SIZE size is stored in it. To speed up part of the memory accesses, we could retrieve data from shared memory when possibile. The data presented in 4 do not show a clear trend of improvement/worsening. It suggests that more investigation is required. Maybe, there is a dependency with input parameter *d*. Unfortunately, given the current resources, further testing was not possible.

As warning reported in 5, the main target of optimization should be the floating-point pipeline. Euclidean distance computation is the greatest source of floating-point operations. Unfortunately, parallelizing DBSCAN over GPU has a major memory limitation [1]. Even if the distances matrix is symmetric, the growth is $O(n^2)$. Since we aim to speed up computation over large datasets this complexity is prohibitive.

The last attempt we have to address the bottleneck of the floating-point pipeline is to early terminate the distance computation between points as soon as the sum is greater than the squared threshold. Sad to say, the performances were abysmal. There was a lot of branch divergence. However, it is likely that performances did not improve because checking if the sum exceeded the threshold required a floating point operation. Instead of removing computation we added a more frequent one.

3.2 Adjacency list calculation

There is no significant difference to 3.1 code. All the attempted optimizations behave the same.

The only improvement that is not shared with 3.1 is early termination. Stopping the thread when it has found all the neighbors expected provides a performance improvement.

3.3 Graph search

The kernel on GPU in charge of a BFS step is identical to the one presented in [1]. In such a small kernel there was not significant room for optimizations.

However, it is not the case for the host code presented in [1]. The main control logic was on the host. In practice, after each parallelized on device BFS step, the frontier had to be transferred to the host. A sequential scan was proposed to check its emptiness. In my implementation, this operation is replaced by the library call *thrust::find* which in parallel on device checks if the frontier is empty.

Similarly to the frontier emptiness check, also cluster assignment was sequential in the host. In my implementation, cluster assignment became a kernel. The cluster assignment array has been moved in device memory until the time to feed the program's output. This kernel simply parallelizes a check and a value assignment.

As reported in 4, these optimizations resulted in a performance improvement. Unfortunately, they are not emphasized by the problem instance used in the test. The searches were very small related to the previous steps size.

4 Speedup

The following test results were collected on Google Colab on a Tesla V100-SXM2-16GB and a Intel(R) Xeon(R) CPU @ 2.20GHz with 2 cores.

The dataset for this test contains 200 000 10-dimensional points.

Id	Description	Hardware	Time	Speedup w.r.t. CPU
1	sklearn.cluster.DBSCAN	CPU	547.918s	-
2	Paper	GPU	9.570s	57,25x
3	2 + no data transfer	GPU	9.342s	58,65x
4	3 + shifted access	GPU	9,563s	57,29x
5	3 + shared memory	GPU	8.311s	65,92x

The dataset for the following test contains 400 000 20-dimensional points.

Id	Description	Hardware	Time	Speedup w.r.t. CPU
1	sklearn.cluster.DBSCAN	CPU	2770.610s	-
2	Paper	GPU	61.670s	44,93x
3	2 + no data transfer	GPU	61.540s	45,02x
4	3 + shifted access	GPU	63.082s	43,92x
5	3 + shared memory	GPU	64.539s	42,93x

5 Profiling

The following data has been collected using the command:

```
!ncu --kernel-name KERN_NAME --csv --launch-skip 0 --launch-count
     1 --set full "executable_path" "input_file"
```

All the warnings are reported.

5.1 Paper implementation

5.1.1 Compute degrees

Table 1: Paper implementation - Compute degrees kernel profiling

Metric Name	Metric Unit	Metric Value
DRAM Frequency	cycle/second	877,465,742.45
SM Frequency	cycle/second	1,312,449,175.95
Elapsed Cycles	cycle	5,764,668,643
Memory %	%	11.11
SOL DRAM	%	0.00
Duration	nsecond	4,391,949,376
SOL L1/TEX Cache	%	13.64
SOL L2 Cache	%	0.10
SM Active Cycles	cycle	4,695,542,023.31
SM %	%	71.83
Executed Ipc Active	inst/cycle	3.07
Executed Ipc Elapsed	inst/cycle	2.50
Issue Slots Busy	%	76.87
Issued Ipc Active	inst/cycle	3.07
SM Busy	%	88.18
Memory Throughput	byte/second	6,582,188.80
Mem Busy	%	11.11
Max Bandwidth	%	8.40
L1/TEX Hit Rate	%	97.18

L2 Hit Rate	%	99.83
Mem Pipes Busy	%	8.40
One or More Eligible	%	76.88
Issued Warp Per Scheduler		0.77
No Eligible	%	23.12
Active Warps Per Scheduler	warp	7.83
Eligible Warps Per Scheduler	warp	3.29
Warp Cycles Per Issued Instruction	cycle	10.18
Warp Cycles Per Executed Instruction	cycle	10.18
Avg. Active Threads Per Warp		32
Avg. Not Predicated Off Threads Per Warp		29.82

Avg. Executed Instructions Per Scheduler	inst	3,609,376,973.14
Executed Instructions	inst	1,155,000,631,404
Avg. Issued Instructions Per Scheduler	inst	3,609,402,050.37
Issued Instructions	inst	1,155,008,656,119
Block Size		1,024
Function Cache Configuration		cudaFuncCachePreferL1
Grid Size		196
Registers Per Thread	register/thread	32
Shared Memory Configuration Size	byte	65,536
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	byte/block	40,960
Static Shared Memory Per Block	byte/block	0
Threads	thread	200,704
Waves Per SM		1.23

Block Limit SM	block	32
Block Limit Registers	block	2
Block Limit Shared Mem	block	2
Block Limit Warps	block	2
Theoretical Active Warps per SM	warp	64
Theoretical Occupancy	%	100
Achieved Occupancy	%	48.93
Achieved Active Warps Per SM	warp	31.31

Branch Instructions Ratio	%	0.03
Branch Instructions	inst	40,000,062,522
Branch Efficiency	%	100
Avg. Divergent Branches		0

Warnings:

- Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis report section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.

- FMA is the highest-utilized pipeline (88.2%). It executes 32-bit floating-point (FADD, FMUL, FMAD, ...) and integer (IMUL, IMAD) operations. The pipeline is over-utilized and likely a performance bottleneck.
- On average, each warp of this kernel spends 3.3 cycles being stalled due to not being selected by the scheduler. This represents about 32.2% of the total average of 10.2 cycles between issuing two instructions. Not selected warps are eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected. A high number of not selected warps typically means you have sufficient warps to cover warp latencies and you may consider reducing the number of active warps to possibly increase cache coherence and data locality.
- A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full-wave and a partial wave of 36 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 51.1%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid.
- This kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated theoretical and measured achieved occupancy can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel.
- Uncoalesced shared access, expected 12500000000 sectors, got 25000000000 (2.00x) at PC 0x7f2a1edb16f0 at gdbscan_paper.cu:42
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a1edb0e20 at gdbscan_paper.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a1edb0e90 at gdbscan_paper.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a1edb1460 at gdbscan_paper.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a1edb1470 at gdbscan_paper.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a1edb1480 at gdbscan_paper.cu:30

- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a1edb1490 at gdbscan_paper.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a1edb14a0 at gdbscan_paper.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a1edb14b0 at gdbscan_paper.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a1edb14c0 at gdbscan_paper.cu:30

5.1.2 Compute adjacency list

Since *compute degrees* and *compute adjacency list* are very similar, I report only the profiling differences for the latter.

Table 2: Paper implementation - Compute adjacency list kernel profiling

Metric Name	Metric Unit	Metric Value
DRAM Frequency	cycle/second	877,402,880.34
SM Frequency	cycle/second	1,312,364,357.08
Elapsed Cycles	cycle	5,788,512,325
Memory %	%	0,48
Duration	nsecond	4,410,403,552
SOL L1/TEX Cache	%	0,60
SM %	%	3,00
Issue Slots Busy	%	3,22
SM Busy	%	3,69
Memory Throughput	byte/second	46,309,732.34
Mem Busy	%	0,48
Max Bandwidth	%	0,38
L1/TEX Hit Rate	%	4,05
L2 Hit Rate	%	4,15
Mem Pipes Busy	%	0,38
One or More Eligible	%	3,22
No Eligible	%	0,98
Avg. Active Threads Per Warp		1,21
Avg. Not Predicated Off Threads Per Warp		1,12
Avg. Executed Instructions Per Scheduler	inst	3,617,707,919.60
Executed Instructions	inst	1,157,666,534,271
Avg. Issued Instructions Per Scheduler	inst	3,617,738,099.61
Issued Instructions	inst	1,157,676,191,875

Branch Instructions	inst	39,962,827,139
Branch Efficiency	%	4,17
Avg. Divergent Branches		25,20

Warnings:

- Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis report section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.
- FMA is the highest-utilized pipeline (87.9%). It executes 32-bit floating-point (FADD, FMUL, FMAD, ...) and integer (IMUL, IMAD) operations. The pipeline is over-utilized and likely a performance bottleneck.
- On average, each warp of this kernel spends 3.3 cycles being stalled due to not being selected by the scheduler. This represents about 32.2% of the total average of 10.2 cycles between issuing two instructions. Not selected warps are eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected. A high number of not selected warps typically means you have sufficient warps to cover warp latencies and you may consider reducing the number of active warps to possibly increase cache coherence and data locality.
- A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full-wave and a partial wave of 36 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 51.1%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid.
- This kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated theoretical and measured achieved occupancy can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel.
- Uncoalesced global access, expected 7445197 sectors, got 7498597 (1.01x) at PC 0x7fc57adb0340 at gdbscan_paper.cu:94
- Uncoalesced shared access, expected 12488303500 sectors, got 24751338640 (1.98x) at PC 0x7fc57adafcf0 at gdbscan_paper.cu:88
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fc57adaf460 at gdbscan_paper.cu:71

- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fc57adaf4d0 at gdbscan_paper.cu:71
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fc57adafa80 at gdbscan_paper.cu:71
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fc57adafa90 at gdbscan_paper.cu:71
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fc57adafaa0 at gdbscan_paper.cu:71
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fc57adafab0 at gdbscan_paper.cu:71
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fc57adafac0 at gdbscan_paper.cu:71
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fc57adafad0 at gdbscan_paper.cu:71
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fc57adafae0 at gdbscan_paper.cu:71

5.1.3 BFS CUDA Step

Table 3: Paper implementation - BFS Step

Metric Name	Metric Unit	Metric Value
DRAM Frequency	cycle/second	868,727,598.57
SM Frequency	cycle/second	1,298,900,462.96
Elapsed Cycles	cycle	69,599
Memory %	%	1.70
SOL DRAM	%	1.70
Duration	nsecond	53,568
SOL L1/TEX Cache	%	9.92
SOL L2 Cache	%	0.63
SM Active Cycles	cycle	3,168.68
SM %	%	0.39
Executed Ipc Active	inst/cycle	0.31
Executed Ipc Elapsed	inst/cycle	0.01
Issue Slots Busy	%	8.54
Issued Ipc Active	inst/cycle	0.34
SM Busy	%	8.54

Memory Throughput	byte/second	15,088,410,991.64
Mem Busy	%	0.60
Max Bandwidth	%	1.70
L1/TEX Hit Rate	%	0.91
L2 Hit Rate	%	0.96
Mem Pipes Busy	%	0.35
One or More Eligible	%	10.87
Issued Warp Per Scheduler		0.11
No Eligible	%	89.13
Active Warps Per Scheduler	warp	11.68
Eligible Warps Per Scheduler	warp	0.28
Warp Cycles Per Issued Instruction	cycle	107.42
Warp Cycles Per Executed Instruction	cycle	120.13
Avg. Active Threads Per Warp		31.08
Avg. Not Predicated Off Threads Per Warp		25.91
Avg. Executed Instructions Per Scheduler	inst	242.01
Executed Instructions	inst	77,443
Avg. Issued Instructions Per Scheduler	inst	270.65
Issued Instructions	inst	86,607
Block Size		1,024
Function Cache Configuration		cudaFuncCachePreferNone
Grid Size		196
Registers Per Thread	register/thread	16
Shared Memory Configuration Size	byte	0
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	0
Threads	thread	200,704
Waves Per SM		1.23
Block Limit SM	block	32
Block Limit Registers	block	4
Block Limit Shared Mem	block	32
Block Limit Warps	block	2
Theoretical Active Warps per SM	warp	64
Theoretical Occupancy	%	100
Achieved Occupancy	%	57.40
Achieved Active Warps Per SM	warp	36.74
Branch Instructions Ratio	%	0.17
Branch Instructions	inst	12,884
Branch Efficiency	%	100
Avg. Divergent Branches		0

- This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.

- All pipelines are under-utilized. Either this kernel is very small or it doesn't issue enough warps per scheduler. Check the Launch Statistics and Scheduler Statistics sections for further details.

- Every scheduler is capable of issuing one instruction per cycle, but for this kernel, each scheduler only issues an instruction every 9.2 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 16 warps per scheduler, this kernel allocates an average of 11.68 active warps per scheduler, but only an average of 0.28 warps was eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps either increase the number of active warps or reduce the time the active warps are stalled.

- On average, each warp of this kernel spends 62.5 cycles being stalled waiting for a scoreboard dependency on an L1TEX (local, global, surface, texture) operation. This represents about 58.2% of the total average of 107.4 cycles between issuing two instructions. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality or by changing the cache configuration, and consider moving frequently used data to shared memory.

- A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full-wave and a partial wave of 36 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 42.6%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid.

- This kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated theoretical and measured achieved occupancy can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel.

5.2 Optimized implementation

This implementation differs from 5.1 only in BFS host code. The frontier emptiness check is done by *thrust::find* on device. I have also introduced the custom kernel *cluster_assignment* to assign clusters on the device.

5.2.1 Cluster assignment

Table 4: Optimized implementation - Clusters assignment profiling

Metric Name	Metric Unit	Metric Value
DRAM Frequency	cycle/second	733,727,810.65
SM Frequency	cycle/second	1,089,866,863.91
Elapsed Cycles	cycle	5,905
Memory %	%	30.17
SOL DRAM	%	30.17
Duration	nsecond	5,408
SOL L1/TEX Cache	%	12.95
SOL L2 Cache	%	11.40
SM Active Cycles	cycle	3,178.05
SM %	%	6.28
Executed Ipc Active	inst/cycle	0.43
Executed Ipc Elapsed	inst/cycle	0.23
Issue Slots Busy	%	11.65
Issued Ipc Active	inst/cycle	0.47
SM Busy	%	11.65
Memory Throughput	byte/second	226,656,804,733.73
Mem Busy	%	11.40
Max Bandwidth	%	30.17
L1/TEX Hit Rate	%	19.39
L2 Hit Rate	%	19.61
Mem Pipes Busy	%	6.05
One or More Eligible	%	11.88
Issued Warp Per Scheduler		0.12
No Eligible	%	88.12
Active Warps Per Scheduler	warp	12.49
Eligible Warps Per Scheduler	warp	0.27
Warp Cycles Per Issued Instruction	cycle	105.09
Warp Cycles Per Executed Instruction	cycle	114.04
Avg. Active Threads Per Warp		22.63
Avg. Not Predicated Off Threads Per Warp		18.79

Avg. Executed Instructions Per Scheduler	inst	341.10
Executed Instructions	inst	109,153
Avg. Issued Instructions Per Scheduler	inst	370.14
Issued Instructions	inst	118,445
Block Size		1,024
Function Cache Configuration		cudaFuncCachePreferL1
Grid Size		196
Registers Per Thread	register/thread	16
Shared Memory Configuration Size	byte	0
Driver Shared Memory Per Block	byte/block	0
Dynamic Shared Memory Per Block	byte/block	0
Static Shared Memory Per Block	byte/block	0
Threads	thread	200,704
Waves Per SM		1.23
Block Limit SM	block	32
Block Limit Registers	block	4
Block Limit Shared Mem	block	32
Block Limit Warps	block	2
Theoretical Active Warps per SM	warp	64
Theoretical Occupancy	%	100
Achieved Occupancy	%	77.03
Achieved Active Warps Per SM	warp	49.30
Branch Instructions Ratio	%	0.20
Branch Instructions	inst	22,236
Branch Efficiency	%	0
Avg. Divergent Branches		0

- This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.

- All pipelines are under-utilized. Either this kernel is very small or it doesn't issue enough warps per scheduler. Check the Launch Statistics and Scheduler Statistics sections for further details.

- Every scheduler is capable of issuing one instruction per cycle, but for this kernel, each scheduler only issues an instruction every 8.4 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 16 warps per scheduler, this kernel allocates an average of 12.49 active warps per scheduler, but only an average of 0.27 warps was eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps either increase the number of active warps or reduce the time the active warps are stalled.

- On average, each warp of this kernel spends 68.9 cycles being stalled waiting for a scoreboard dependency on an L1TEX (local, global, surface, texture) operation. This represents about 65.6% of the total average of 105.1 cycles between issuing two instructions. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality or by changing the cache configuration, and consider moving frequently used data to shared memory.

- Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can significantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 22.6 threads being active per cycle. This is further reduced to 18.8 threads per warp due to predication. The compiler may use predication to avoid an actual branch. Instead, all instructions are scheduled, but a per-thread condition code or predicate controls which threads execute the instructions. Try to avoid different execution paths within a warp when possible. In addition, ensure your kernel makes use of Independent Thread Scheduling, which allows a warp to reconverge after a data-dependent conditional block by explicitly calling __syncwarp().

- A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full-wave and a partial wave of 36 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 23.0%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid.

- This kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated theoretical and measured achieved occupancy can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel.

- Uncoalesced global access, expected 4857 sectors, got 7919 (1.63x) at PC 0x7effbedd14d0 at gdbscan.cu:137
- Uncoalesced global access, expected 4857 sectors, got 7919 (1.63x) at PC 0x7effbedd1510 at gdbscan.cu:139

5.3 Shifted accesses implementation

This implementation differs from 5.2 in *compute_kernels* and *compute_adjacency_list*. The for iterating through the dataset begins shifted for each thread. This should increase memory bandwidth usage since threads in the same warp access concurrently to different contiguous memory cells. However, it doesn't.

5.3.1 Compute degrees

The following table contains only the values different from the Table1.

Table 5: Shifted implementation - Compute degrees profiling

Metric Name	Metric Unit	Metric Value
DRAM Frequency	cycle/second	877,197,925.50
SM Frequency	cycle/second	1,312,026,269.39
Elapsed Cycles	cycle	5,891,094,084
Memory %	%	11.22
Duration	nsecond	4,489,655,360
SOL L1/TEX Cache	%	13.77
SOL L2 Cache	%	0.18
SM Active Cycles	cycle	4,798,174,496.18
SM %	%	70.82
Executed Ipc Active	inst/cycle	3.08
Executed Ipc Elapsed	inst/cycle	2.51
Issue Slots Busy	%	77.02
Issued Ipc Active	inst/cycle	3.08
SM Busy	%	86.95
Memory Throughput	byte/second	6,486,655.58
Mem Busy	%	11.22
Max Bandwidth	%	8.22
L1/TEX Hit Rate	%	98.96
L2 Hit Rate	%	99.84
Mem Pipes Busy	%	8.22
One or More Eligible	%	77.03
No Eligible	%	22.97
Active Warps Per Scheduler	warp	7.83

Eligible Warps Per Scheduler	warp	3.28
Warp Cycles Per Issued Instruction	cycle	10.17
Warp Cycles Per Executed Instruction	cycle	10.17
Achieved Occupancy	%	48.95
Achieved Active Warps Per SM	warp	31.33
Branch Instructions	inst	40,000,068,772

- Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis report section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.
- FMA is the highest-utilized pipeline (86.9%). It executes 32-bit floating-point (FADD, FMUL, FMAD, ...) and integer (IMUL, IMAD) operations. The pipeline is over-utilized and likely a performance bottleneck.
- On average, each warp of this kernel spends 3.3 cycles being stalled due to not being selected by the scheduler. This represents about 32.0% of the total average of 10.2 cycles between issuing two instructions. Not selected warps are eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected. A high number of not selected warps typically means you have sufficient warps to cover warp latencies and you may consider reducing the number of active warps to possibly increase cache coherence and data locality.
- A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full-wave and a partial wave of 36 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 51.0%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid.
- This kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated theoretical and measured achieved occupancy can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel.
- Uncoalesced global access, expected 50000000000 sectors, got 60937500000 (1.22x) at PC 0x7f2a16dd5280 at gdbscan_shifted.cu:43
- Uncoalesced shared access, expected 12500000000 sectors, got 25000000000 (2.00x) at PC 0x7f2a16dd5260 at gdbscan_shifted.cu:43

- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a16dd4820 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a16dd4890 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a16dd4e60 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a16dd4e70 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a16dd4e80 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a16dd4e90 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a16dd4ea0 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a16dd4eb0 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f2a16dd4ec0 at gdbscan_shifted.cu:30

5.3.2 Compute adjacency list

The following table contains only the values different from the Table5.

Table 6: Shifted implementation - Compute adjacency list profiling

Metric Name	Metric Unit	Metric Value
DRAM Frequency	cycle/second	877,872,308.43
SM Frequency	cycle/second	1,313,053,669.55
Elapsed Cycles	cycle	5,952,104,326
Memory %	%	11.08
SOL DRAM	%	0.01
Duration	nsecond	4,532,701,248
SOL L1/TEX Cache	%	13.61
SM Active Cycles	cycle	4,845,236,086.96
SM %	%	71.45
Executed Ipc Active	inst/cycle	3.05
Executed Ipc Elapsed	inst/cycle	2.49
Issue Slots Busy	%	76.37
Issued Ipc Active	inst/cycle	3.05

SM Busy	%	87.77
Memory Throughput	byte/second	45,430,753.26
Mem Busy	%	11.08
Max Bandwidth	%	8.13
L1/TEX Hit Rate	%	98.90
L2 Hit Rate	%	99.58
Mem Pipes Busy	%	8.13
One or More Eligible	%	76.41
Issued Warp Per Scheduler		0.76
No Eligible	%	23.59
Eligible Warps Per Scheduler	warp	3.26
Warp Cycles Per Issued Instruction	cycle	10.25
Warp Cycles Per Executed Instruction	cycle	10.25
Avg. Active Threads Per Warp		28.64
Avg. Not Predicated Off Threads Per Warp		26.58
Avg. Executed Instructions Per Scheduler	inst	3,700,420,275.36
Executed Instructions	inst	1,184,134,488,116
Avg. Issued Instructions Per Scheduler	inst	3,700,461,554.68
Issued Instructions	inst	1,184,147,697,499
Achieved Occupancy	%	48.94
Achieved Active Warps Per SM	warp	31.32
Branch Instructions Ratio	%	0.04
Branch Instructions	inst	43,718,095,873
Branch Efficiency	%	99.97
Avg. Divergent Branches		22,723.33

- Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis report section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.

- FMA is the highest-utilized pipeline (87.8%). It executes 32-bit floating-point (FADD, FMUL, FMAD, ...) and integer (IMUL, IMAD) operations. The pipeline is over-utilized and likely a performance bottleneck.

- On average, each warp of this kernel spends 3.3 cycles being stalled due to not being selected by the scheduler. This represents about 31.9% of the total average of 10.3 cycles between issuing two instructions. Not selected warps are eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected. A high number of not selected warps typically means you have sufficient warps to cover warp latencies and you may consider reducing the number of active warps to possibly increase cache coherence and data locality.

- A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full-wave and a partial wave of 36 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 51.1%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid.
- This kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated theoretical and measured achieved occupancy can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel.
- Uncoalesced global access, expected 47428009070 sectors, got 60184093230 (1.27x) at PC 0x7f1ccedd3570 at gdbscan_shifted.cu:90
- Uncoalesced global access, expected 7303229 sectors, got 7483531 (1.02x) at PC 0x7f1ccedd3d20 at gdbscan_shifted.cu:96
- Uncoalesced shared access, expected 12488789110 sectors, got 24750387410 (1.98x) at PC 0x7f1ccedd3550 at gdbscan_shifted.cu:90
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f1ccedd2b60 at gdbscan_shifted.cu:72
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f1ccedd2bd0 at gdbscan_shifted.cu:72
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f1ccedd3180 at gdbscan_shifted.cu:72
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f1ccedd3190 at gdbscan_shifted.cu:72
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f1ccedd31a0 at gdbscan_shifted.cu:72
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f1ccedd31b0 at gdbscan_shifted.cu:72
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f1ccedd31c0 at gdbscan_shifted.cu:72
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f1ccedd31d0 at gdbscan_shifted.cu:72

- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f1ccedd31e0 at gdbscan_shifted.cu:72

5.4 Shared accesses implementation

This implementation differs from 5.2 in *compute_kernels* and *compute_adjacency_list*. The for iterating through the dataset does not access directly to the global memory. In fact, if the item to retrieve resides in shared memory, it is pulled from there.

5.4.1 Compute degrees

The following table contains only the values different from the Table1.

Table 7: Shared implementation - Compute degrees profiling

Metric Name	Metric Unit	Metric Value
DRAM Frequency	cycle/second	877,698,876.77
SM Frequency	cycle/second	1,312,626,721.90
Elapsed Cycles	cycle	5,813,721,051
Memory %	%	11.02
Duration	nsecond	4,428,258,496
SOL L1/TEX Cache	%	13.53
SM Active Cycles	cycle	4,733,964,897.55
SM %	%	71.36
Issue Slots Busy	%	76.65
SM Busy	%	87.62
Memory Throughput	byte/second	6,242,887.59
Mem Busy	%	11.02
Max Bandwidth	%	8.33
L1/TEX Hit Rate	%	97.16
L2 Hit Rate	%	99.83
Mem Pipes Busy	%	8.33
One or More Eligible	%	76.65
No Eligible	%	23.35
Eligible Warps Per Scheduler	warp	3.30
Warp Cycles Per Issued Instruction	cycle	10.22
Warp Cycles Per Executed Instruction	cycle	10.22
Avg. Not Predicated Off Threads Per Warp		29.79
Avg. Executed Instructions Per Scheduler	inst	3,628,508,643.61
Executed Instructions	inst	1,161,122,765,954
Avg. Issued Instructions Per Scheduler	inst	3,628,538,914.12
Issued Instructions	inst	1,161,132,452,518

Achieved Occupancy	%	48.95
Achieved Active Warps Per SM	warp	31.33
Branch Instructions Ratio	%	0.04
Branch Instructions	inst	42,493,669,562

- Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis report section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.
- FMA is the highest-utilized pipeline (87.6%). It executes 32-bit floating-point (FADD, FMUL, FMAD, ...) and integer (IMUL, IMAD) operations. The pipeline is over-utilized and likely a performance bottleneck.
- On average, each warp of this kernel spends 3.3 cycles being stalled due to not being selected by the scheduler. This represents about 32.3% of the total average of 10.2 cycles between issuing two instructions. Not selected warps are eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected. A high number of not selected warps typically means you have sufficient warps to cover warp latencies and you may consider reducing the number of active warps to possibly increase cache coherence and data locality.
- A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full-wave and a partial wave of 36 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 51.0%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid.
- This kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated theoretical and measured achieved occupancy can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel.
- Uncoalesced shared access, expected 12436070400 sectors, got 24872140800 (2.00x) at PC 0x7f8d9edd4f20 at gdbscan_shifted.cu:45
- Uncoalesced shared access, expected 63929600 sectors, got 127859200 (2.00x) at PC 0x7f8d9edd5590 at gdbscan_shifted.cu:45
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f8d9edd4620 at gdbscan_shifted.cu:30

- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f8d9edd4690 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f8d9edd4c60 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f8d9edd4c70 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f8d9edd4c80 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f8d9edd4c90 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f8d9edd4ca0 at gdbscan_shifted.cu:30
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7f8d9edd4cb0 at gdbscan_shifted.cu:30

5.4.2 Compute adjacency list

The following table contains only the values different from the Table7.

Table 8: Shared implementation - Compute adjacency list profiling

Metric Name	Metric Unit	Metric Value
DRAM Frequency	cycle/second	877,578,899.34
SM Frequency	cycle/second	1,312,614,497.86
Elapsed Cycles	cycle	6,067,319,813
Memory %	%	13.36
Duration	nsecond	4,621,990,400
SOL L1/TEX Cache	%	16.40
SM Active Cycles	cycle	4,942,873,762.55
SM %	%	69.83
Executed Ipc Active	inst/cycle	3.13
Executed Ipc Elapsed	inst/cycle	2.55
Issue Slots Busy	%	78.24
Issued Ipc Active	inst/cycle	3.13
SM Busy	%	85.71
Memory Throughput	byte/second	44,339,830.74
Mem Busy	%	13.36
Max Bandwidth	%	10.81
L1/TEX Hit Rate	%	97.05

L2 Hit Rate	%	99.30
Mem Pipes Busy	%	10.81
One or More Eligible	%	78.28
Issued Warp Per Scheduler		0.78
No Eligible	%	21.72
Active Warps Per Scheduler	warp	7.82
Eligible Warps Per Scheduler	warp	3.18
Warp Cycles Per Issued Instruction	cycle	9.99
Warp Cycles Per Executed Instruction	cycle	9.99
Avg. Active Threads Per Warp		28.65
Avg. Not Predicated Off Threads Per Warp		26.40

Avg. Executed Instructions Per Scheduler	inst	3,867,473,970.07
Executed Instructions	inst	1,237,591,670,421
Avg. Issued Instructions Per Scheduler	inst	3,867,524,213.31
Issued Instructions	inst	1,237,607,748,260
Achieved Active Warps Per SM	warp	31.27
Branch Instructions Ratio	%	0.03
Branch Instructions	inst	39,962,827,139
Avg. Divergent Branches		604.43

- Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis report section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.

- FMA is the highest-utilized pipeline (85.7%). It executes 32-bit floating-point (FADD, FMUL, FMAD, ...) and integer (IMUL, IMAD) operations. The pipeline is over-utilized and likely a performance bottleneck.

- On average, each warp of this kernel spends 3.1 cycles being stalled due to not being selected by the scheduler. This represents about 30.6% of the total average of 10.0 cycles between issuing two instructions. Not selected warps are eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected. A high number of not selected warps typically means you have sufficient warps to cover warp latencies and you may consider reducing the number of active warps to possibly increase cache coherence and data locality.

- A wave of thread blocks is defined as the maximum number of blocks that can be executed in parallel on the target GPU. The number of blocks in a wave depends on the number of multiprocessors and the theoretical occupancy of the kernel. This kernel launch results in 1 full-wave and a partial wave of 36 thread blocks. Under the assumption of a uniform execution duration of all thread blocks, the partial wave may account for up to 50.0% of the total kernel runtime with a lower occupancy of 51.1%. Try launching a grid with no partial wave. The overall impact of this tail effect also lessens with the number of full waves executed for a grid.

- This kernel's theoretical occupancy is not impacted by any block limit. The difference between calculated theoretical and measured achieved occupancy can be the result of warp scheduling overheads or workload imbalances during the kernel execution. Load imbalances can occur between warps within a block as well as across blocks of the same kernel.
- Uncoalesced global access, expected 7445197 sectors, got 7498597 (1.01x) at PC 0x7fdf3cdd3ad0 at gdbscan_shifted.cu:101
- Uncoalesced shared access, expected 12488303500 sectors, got 24751338640 (1.98x) at PC 0x7fdf3cdd3420 at gdbscan_shifted.cu:95
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fdf3cdd2b60 at gdbscan_shifted.cu:74
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fdf3cdd2bd0 at gdbscan_shifted.cu:74
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fdf3cdd3180 at gdbscan_shifted.cu:74
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fdf3cdd3190 at gdbscan_shifted.cu:74
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fdf3cdd31a0 at gdbscan_shifted.cu:74
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fdf3cdd31b0 at gdbscan_shifted.cu:74
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fdf3cdd31c0 at gdbscan_shifted.cu:74
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fdf3cdd31d0 at gdbscan_shifted.cu:74
- Uncoalesced shared access, expected 6250 sectors, got 12500 (2.00x) at PC 0x7fdf3cdd31e0 at gdbscan_shifted.cu:74

6 Future works

There is a clear next step for the presented work: address all the reported warnings. It is likely that no performance improvement will occur, since the bottleneck on the floating point pipeline is caused by the fundamental philosophy of [1].

References

[1] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. G-dbscan: A gpu accelerated algorithm for density-based clustering. *Procedia computer science*, 18:369–378, 2013.

[2] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *High Performance Computing – HiPC 2007*, pages 197–208, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[3] Anant Ram, Sunita Jalal, Anand S Jalal, and Manoj Kumar. A density based algorithm for discovering density varied clusters in large spatial databases. *International journal of computer applications*, 3(6):1–4, 2010.

[4] Wikipedia contributors. Cluster analysis — Wikipedia, the free encyclopedia, 2021. [Online; accessed 28-June-2021].