Binning in MPI and Cuda

Method A

4 Node Variable Clases

Nodes	# Elements	# Classes	Time
4	10,000	1	70 us
4	10,000	5	130 us
4	10,000	10	157 us
4	10,000	20	290 us
4	10,000	40	323 us

8 Node Variable Clases

Nodes	# Elements	# Classes	Time
8	10,000	1	65 us
8	10,000	5	90 us
8	10,000	10	95 us
8	10,000	20	150 us
8	10,000	40	170 us

8 Node Variable N

Nodes	# Elements	# Classes	Time
8	1000	5	35 us
8	10,000	5	95 us
8	100,000	5	571 us
8	1,000,000	5	5399 us
8	10,000,000	5	53,570 us
8	100,000,000	5	527,793 us

Method B

4 Node and 8 Node Variable N and

Nodes	# Elements	# Classes	Time
4	10,000	5	249 us

Nodes	# Elements	# Classes	Time
4	100,000	5	2179 us
8	10,000	5	204 us
8	100,000	5	2167 us

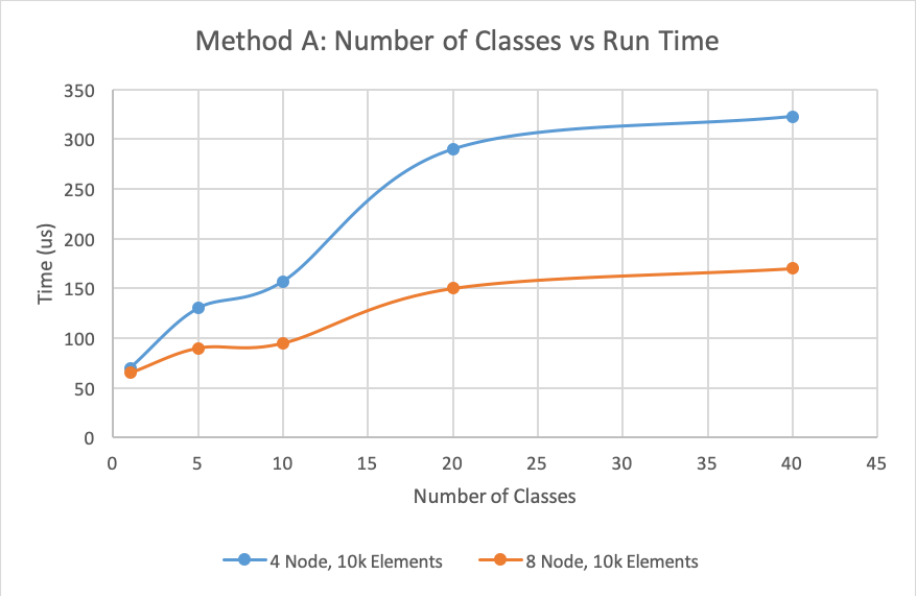
Cuda A - Variable N

# Elements	# Classes	Run Time
10,000	5	252 us
100,000	5	300 us
1,000,000	5	400 us
10,000,000	5	448 us
100,000,000	5	651 us

Analysis

Effect of...

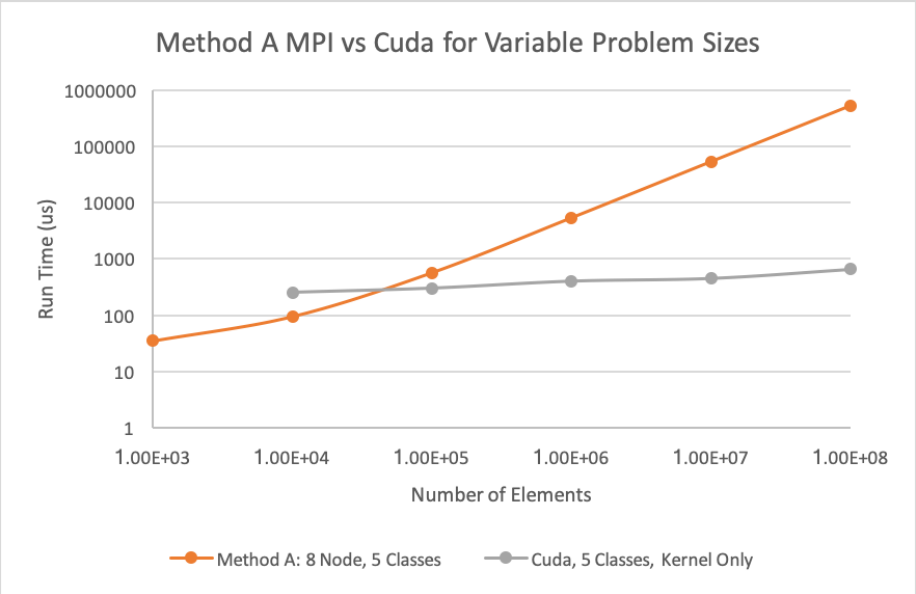
Variable Classes:

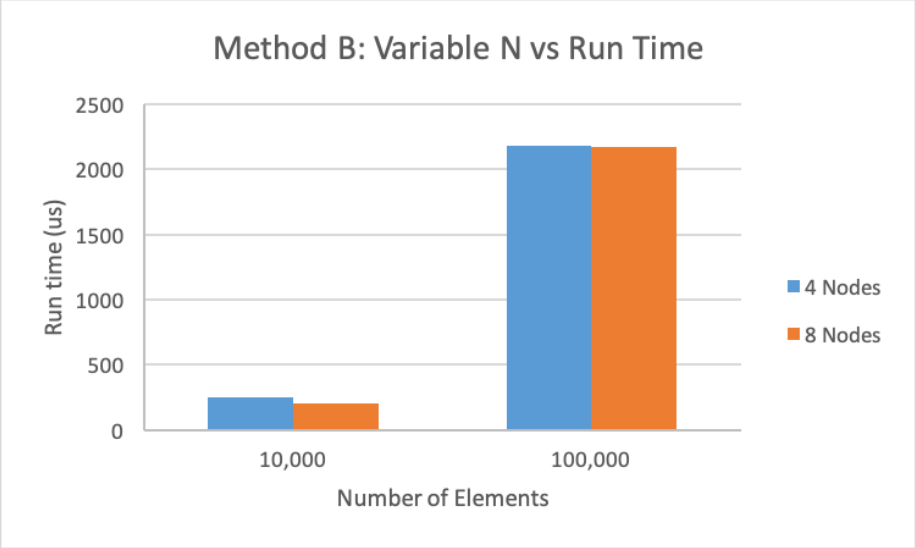


Demonstrated by the variable classes experiment run with method A. Having more classes resulted in long run times. There are two likely explanations for this: 1. With more classes, more comparisions must be made for each term

to find the correct bucket to place the value in 2. With more buckets to place the term in, the memory access pattern becomes evern more complicated than before.

Variable Problem Size:

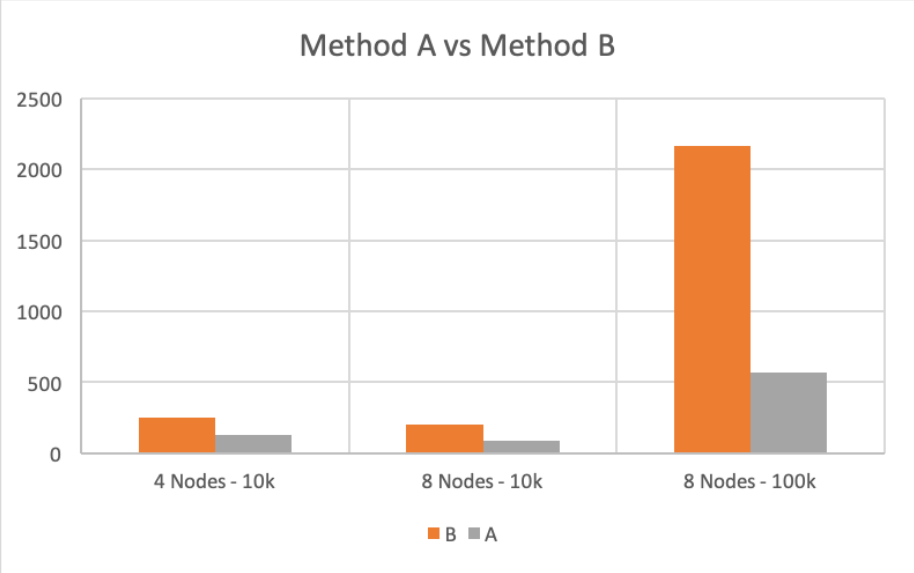




Demonstrated by the **8 Node Variable N** test case run with method A. With a larger N, the problem size grows and thus, we should expect the run time to grow as well. We see this behavior clearly in the the timing results. Initially,

an increase in problem size by a factor of 10 led to a small increase in run time, (less tha an equivocal 10x increase in time). This is likely because the underlying hardware was being fully utilized in the smaller cases, thus increased problem sizes took advantage of idle hardware. However, the 10x jumps in problem size after 100,000 elements resulted in similar 10x jumps in the timing data.

Method A vs. Method B:

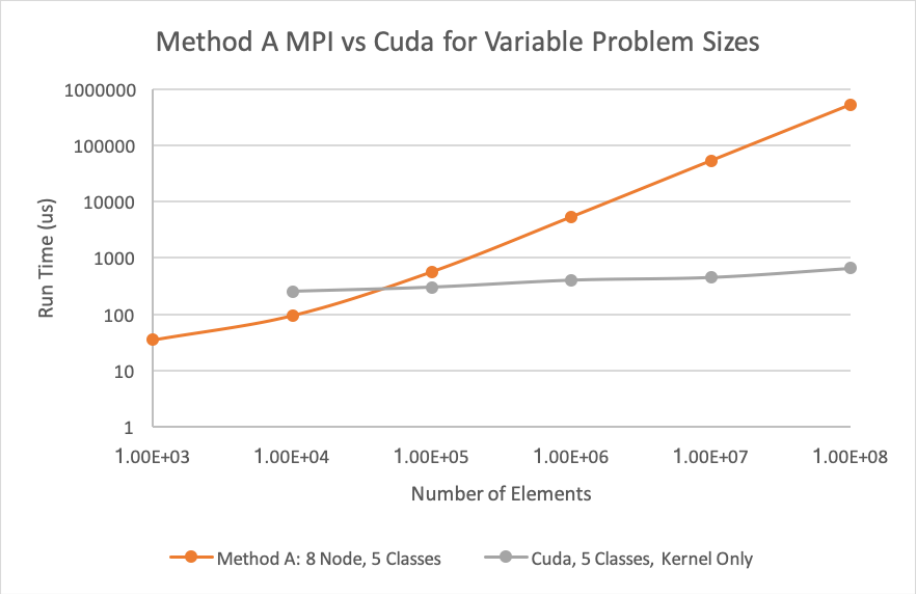


One of the most interesting questions to consider is how method A differs from the method B in timing.

In method A, a fraction of the data is read and completely sorted by each process. The end results are coalesced on node 0 at the end of the run time. In comparision, each node in method B reads the entire data stream and only pulls out nodes from a specific bucket type.

Our results show that method A is far superior to method B, in the 100k element case, running almost 4 times faster. This makes sense if we consider the how the algorithm is running. In method A, every piece of data is touched only once by any node before being sorted. Comparitively, in method B, each piece of data is touched by EVERY SINGLE NODE.

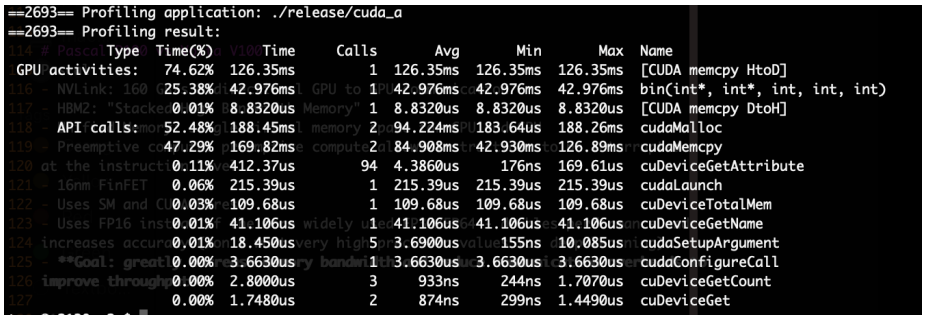
Method A vs. CUDA



Another highly interesting question to consider is how well a CUDA implementation of binning performs against the algorithm discussed in method A. Intutively, the operations performed to "bin" data" is very simple, simply determining where each piece of data falls within a set of buckets. We notice the run time remains fairly constant from problem size to problem size, while the run time from Method A increases linearly.

The following two images show an interesting result. In the first image, we see the profile data from a binning of 1000 elements. Here, we see the binning takes longer than the copy to the device. However as the problem size scales to 100,000,000 elements, we see that suddenly the memcpy takes a longer time, indicating that we've hit a memory bottleneck!

```
==2166== Profiling application: ./release/cuda_a
==2166== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.91%  3.5840us         1  3.5840us  3.5840us  3.5840us  bin(int*, int*, int, int, int)
                   26.82%  1.8880us         1  1.8880us  1.8880us  1.8880us  [CUDA memcpy HtoD]
                   22.27%  1.5680us         1  1.5680us  1.5680us  1.5680us  [CUDA memcpy DtoH]
      API calls:   99.49%  213.49ms         2  106.75ms  197.03us  213.30ms  cudaMalloc
                    0.28%  601.89us        94  6.4030us     307ns  256.52us  cuDeviceGetAttribute
                    0.08%  181.93us         1  181.93us  181.93us  181.93us  cuDeviceTotalMem
                    0.07%  157.39us         1  157.39us  157.39us  157.39us  cudaLaunch
                    0.03%  68.817us         2  34.408us  33.179us  35.638us  cudaMemcpy
                    0.03%  62.229us         1  62.229us  62.229us  62.229us  cuDeviceGetName
                    0.00%  8.7240us         5  1.7440us     165ns  6.9370us  cudaSetupArgument
                    0.00%  4.7580us         3  1.5860us     417ns  2.8730us  cuDeviceGetCount
                    0.00%  2.7400us         2  1.3700us     517ns  2.2230us  cuDeviceGet
                    0.00%  1.6870us         1  1.6870us  1.6870us  1.6870us  cudaConfigureCall
```



```
==2693== Profiling application: ./release/cuda_a
==2693== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   74.62%  126.35ms         1  126.35ms  126.35ms  126.35ms  [CUDA memcpy HtoD]
                   25.38%  42.976ms         1  42.976ms  42.976ms  42.976ms  bin(int*, int*, int, int, int)
                    0.01%  8.8320us         1  8.8320us  8.8320us  8.8320us  [CUDA memcpy DtoH]
      API calls:   52.48%  188.45ms         2  94.224ms  183.64us  188.26ms  cudaMalloc
                   47.29%  169.82ms         2  84.908ms  42.930ms  126.89ms  cudaMemcpy
                    0.11%  412.37us        94  4.3860us     176ns  169.61us  cuDeviceGetAttribute
                    0.06%  215.39us         1  215.39us  215.39us  215.39us  cudaLaunch
                    0.03%  109.68us         1  109.68us  109.68us  109.68us  cuDeviceTotalMem
                    0.01%  41.106us         1  41.106us  41.106us  41.106us  cuDeviceGetName
                    0.01%  18.450us         5  3.6900us     155ns  10.085us  cudaSetupArgument
                    0.00%  3.6630us         1  3.6630us  3.6630us  3.6630us  cudaConfigureCall
                    0.00%  2.8000us         3     933ns     244ns  1.7070us  cuDeviceGetCount
                    0.00%  1.7480us         2     874ns     299ns  1.4490us  cuDeviceGet
```

Pascal P100 vs Volta V100

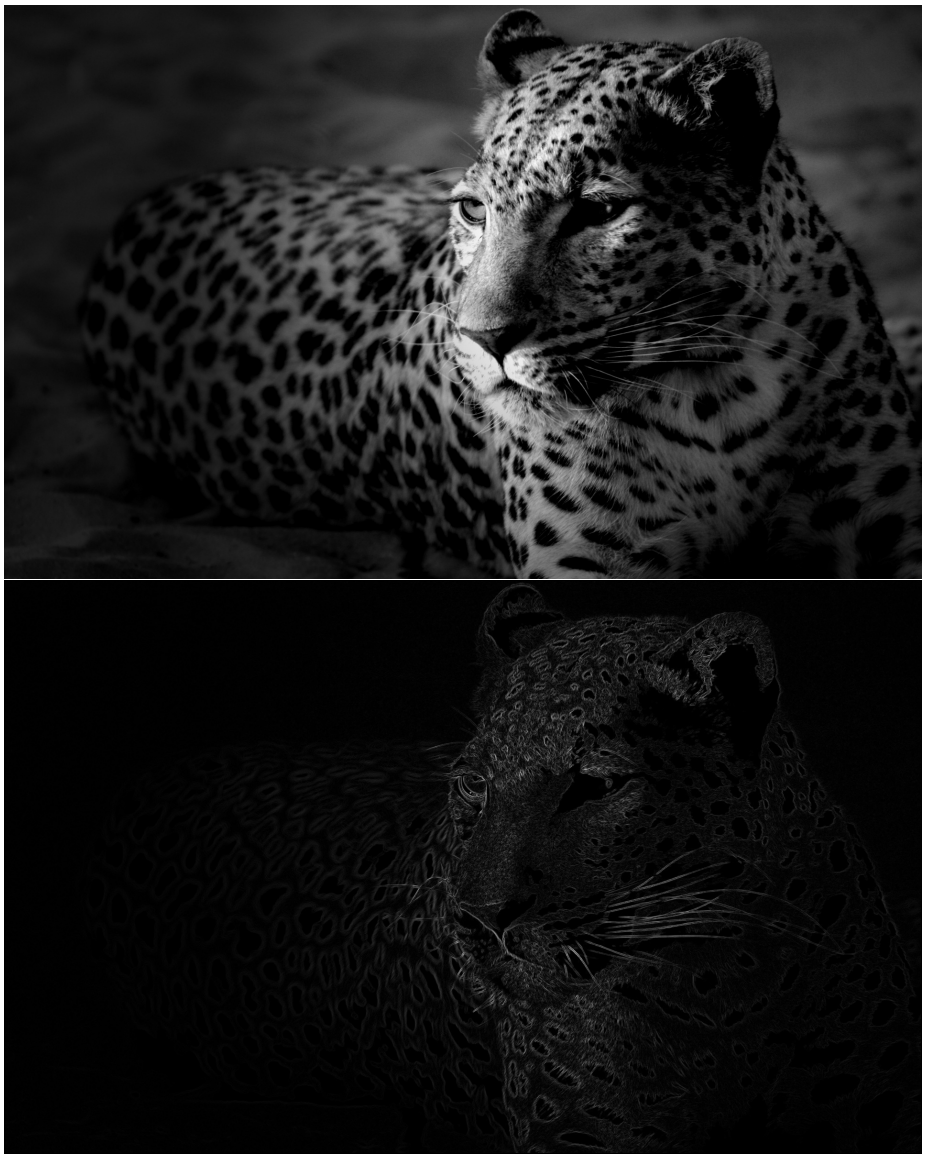
Pascal:

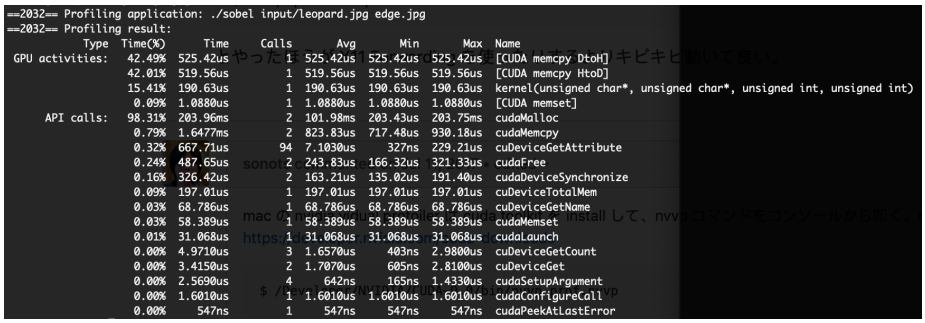
- NVLink: 160 GB/s bidirectional GPU to GPU communication
- HBM2: "Stacked High Bandwidth Memory"
- Unified Memory: single virtual memory space for CPU and GPU
- Preemptive compute: preemptive compute allows instructions to be interrupted at the instruction level
- 16nm FinFET
- Uses SM and CUDA Cores
- Uses FP16 instead of the more widely used FP32/FP64 - doubles performance and increases accuracy (don't want very high precision values for deep learning)
- **Goal: greatly increase memory bandwidth and reduce communication overheads to improve throughput**

V100:

- Introduction of Tensor Cores alongside CUDA Cores
- Targeted towards DL/ML
- Larger L1 Cache than the P100
- V100: unified shared memory/L1 cache
  - low cache hit latency
- Independent thread scheduling allows for in-warp synchronization
- **Tensor cores** offers multiply and add operations for greater efficiency breaks away from the P100 focus on SM/CUDA cores
- improved unified memory
- improved HBM2 and NVLink performance
- **Goal: EXTREME performance on deep learning applications.**

EC





```
==2032== Profiling application: ./sobel input/leopard.jpg edge.jpg
==2032== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   42.49%  525.42us         1  525.42us  525.42us  525.42us  [CUDA memcpy DtoH]
                   42.01%  519.56us         1  519.56us  519.56us  519.56us  [CUDA memcpy HtoD]
                   15.41%  190.63us         1  190.63us  190.63us  190.63us  kernel(unsigned char*, unsigned char*, unsigned int, unsigned int)
                    0.09%  1.0880us         1  1.0880us  1.0880us  1.0880us  [CUDA memset]
      API calls:   98.31%  203.96ms         2  101.98ms  203.43us  203.75ms  cudaMalloc
                    0.79%  1.6477ms         2  823.83us  717.48us  930.18us  cudaMemcpy
                    0.32%  667.71us        94  7.1030us     327ns  229.21us  cuDeviceGetAttribute
                    0.24%  487.65us         2  243.83us  166.32us  321.33us  cudaFree
                    0.16%  326.42us         2  163.21us  135.02us  191.40us  cudaDeviceSynchronize
                    0.09%  197.01us         1  197.01us  197.01us  197.01us  cuDeviceTotalMem
                    0.03%  68.786us         1  68.786us  68.786us  68.786us  cuDeviceGetName
                    0.03%  58.389us         1  58.389us  58.389us  58.389us  cudaMemset
                    0.01%  31.068us         1  31.068us  31.068us  31.068us  cudaLaunch
                    0.00%  4.9710us         3  1.6570us     403ns  2.9800us  cuDeviceGetCount
                    0.00%  3.4150us         2  1.7070us     605ns  2.8100us  cuDeviceGet
                    0.00%  2.5690us         4     642ns     165ns  1.4330us  cudaSetupArgument
                    0.00%  1.6010us         1  1.6010us  1.6010us  1.6010us  cudaConfigureCall
                    0.00%     547ns         1     547ns     547ns     547ns  cudaPeekAtLastError
```