GPGPU Programming Assignment

Louis Robinson, sklv77

1 Profiling Basic Algorithm Run-Time For Sparse Matrices

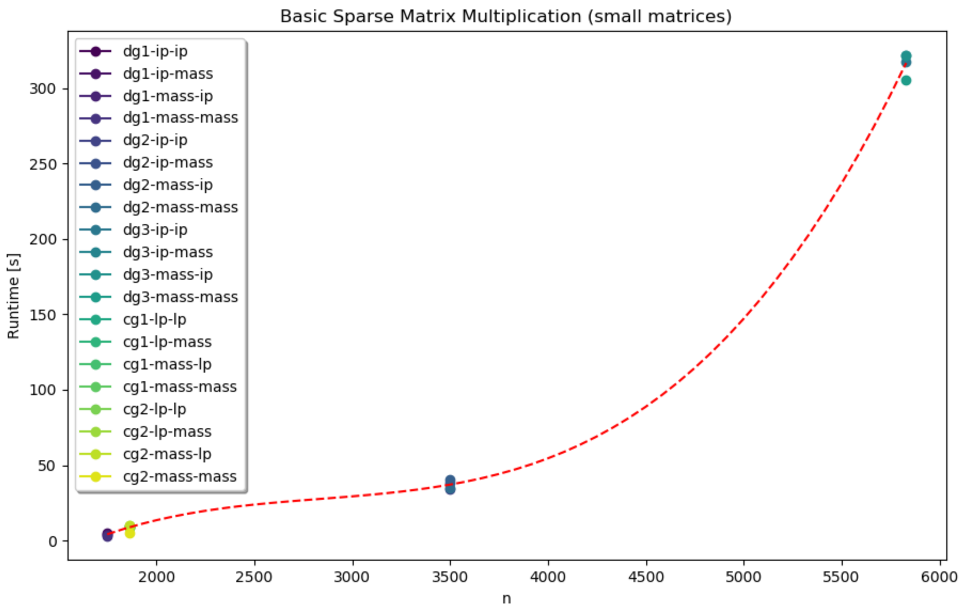


Figure 1: Basic implementation of matrix multiplication, Dimension, n vs Runtime.

The runtimes for sparse matrix multiplication with the basic algorithm are clearly very slow shown in Figure(1) DG3's (n = 3500) taking $\approx 60s$ and DG4's (n = 6000) taking over $\approx 300s$. I have fit the data with a 3^{rd} order polynomial regressor, suggesting cubic time complexity.

2 Sparse matrix-matrix multiplication

2.1 Optimised multiplication algorithm

To loop effectively and improve data locality I transformed the input matrices from COO (first quick-sorting) to CSR, and CSC. Unable to predict the number of non-zeros in the product (the provided DG matrices had NZ $\underset{\sim}{\propto} n^2$), to store the product I decided to create a dynamic memory structure. Initially I created a structure that resizes the arrays based on how far through the matrix we are, using realloc when it is full. Profiling showed reallocs sometimes took significantly longer. Attempting to fix this I modified my array struct so that it contained a pointer to a new one, malloc'ing the estimated remaining memory required, linking the arrays together. Unfortunately this turned out to be slightly slower than realloc, so I reverted back.

Algorithm 1 Both A and B have counter going through their non-zeros and incrementing them by one when the dummy variable (j coordinate of each entry of A and i of B) is strictly below the other (this is known as a zipper function for the intersection of ordered sets).

```
procedure INTERSECTION ALGORITHM, ZIPPER(Acsr, Bcsc)
    for col in B do
        for row in A do                                          ▷ each coordinate in C
            total = 0                                            ▷ initialise sum
            i, j = Acsr->IA[row], Bcsc->IA[col]                  ▷ initialise counters
            while i < Acsr->IA[row+1] and j < Bcsc->IA[col+1] do
                if Acsr->JA[i] < Bcsc->JA[j] then i++
                else if Acsr->JA[i] > Bcsc->JA[j] then j++
                else total += Acsr->data[i]*Bcsr->data[j], j++, i++
            if $total \neq 0$ then
                $store \rightarrow row, col, total$, nz++          ▷ increment non-zero counter
```

2.2 Profile and optimisation potential

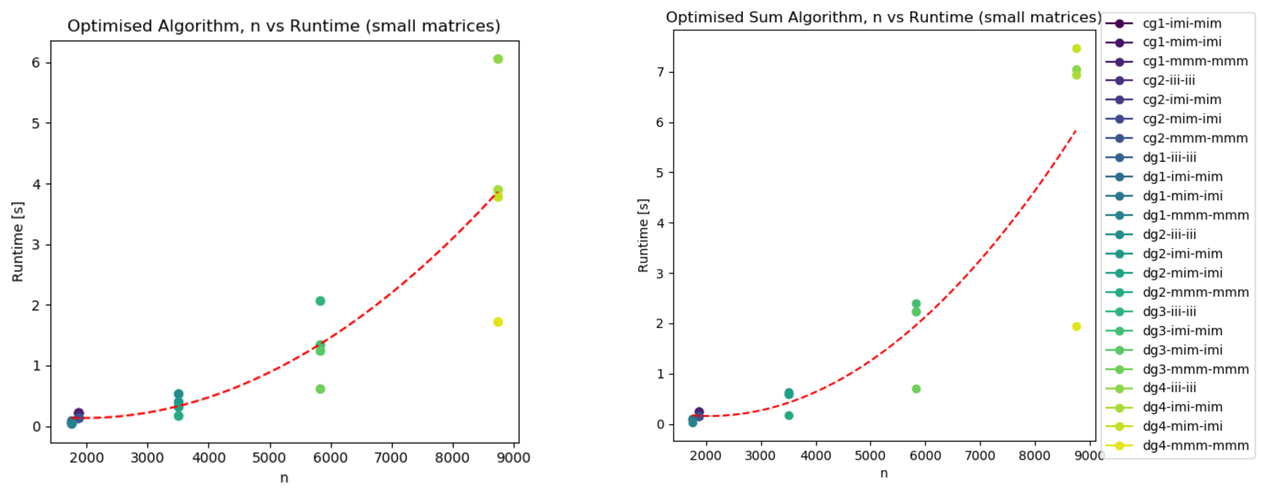


Figure 2: Optimised implementation of matrix multiplication, Dimension, n vs Runtime. In the key the characters m and i represent mass, and ip-laplace (the left plot just considers the first 2 m or i characters in each label).

Comparing Figure(2) (left) to Figure(1), runtime has decreased dramatically, and now fits a 2^{nd} order polynomial regressor. Profiling each part of the algorithm, the data transformations were fast. The function appeared to have good CPI, compared with basic implementation; basic achieved around 0.6-1.0 CPI, optimised around 0.28-0.3 CPI, suggesting good stream quality, (minimal CPI on most systems is 0.25). Although this will be contributed to by the busy-waiting loop (while loop in Figure(1)), using intel advisor this loop take 58-68% of the time in optimised-sparsemm, shown in the top-down result of Figure(4).

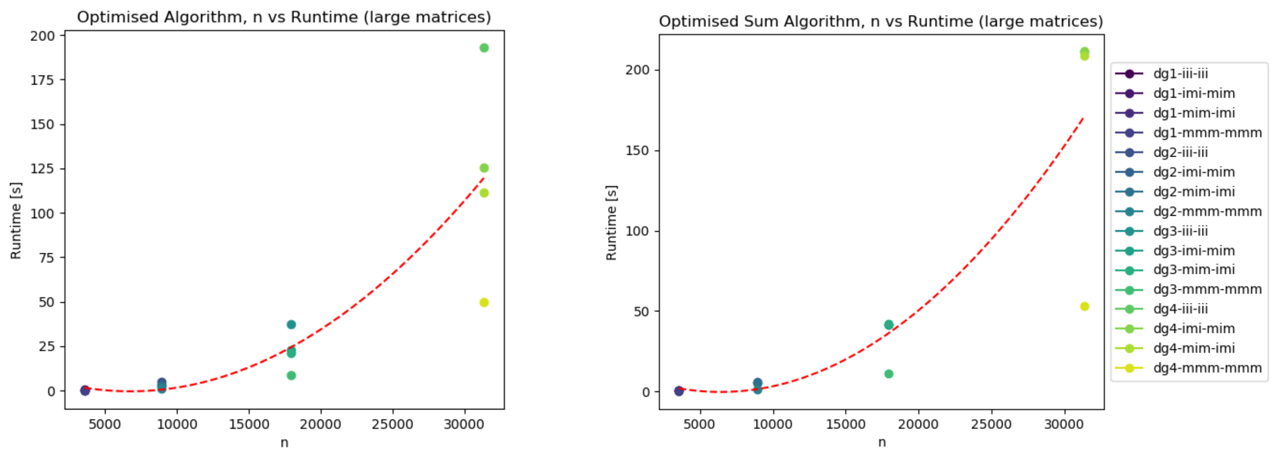
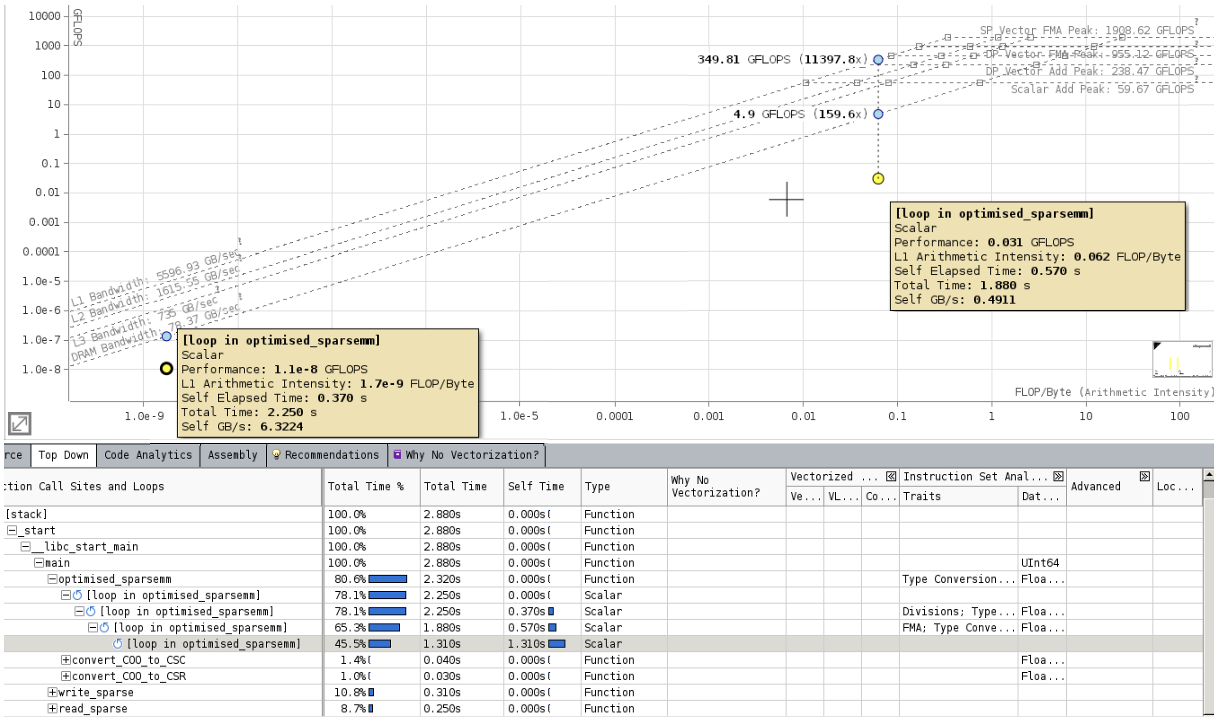


Figure 3: Optimised implementation of matrix multiplication, Dimension, n vs Runtime.


GFLOPS
10000
1000
100
10
1
0.1
0.01
0.001
0.0001
1.0e-5
1.0e-6
1.0e-7
1.0e-8
SP Vector FMA Peak: 1908.62 GFLOPS
DP Vector FMA Peak: 955.12 GFLOPS
DP Vector Add Peak: 238.47 GFLOPS
Scalar Add Peak: 59.67 GFLOPS
349.81 GFLOPS (11397.8x)
4.9 GFLOPS (159.6x)
L1 Bandwidth: 5596.93 GB/sec
L2 Bandwidth: 1615.55 GB/sec
L3 Bandwidth: 735 GB/sec
DRAM Bandwidth: 78.37 GB/sec
FLOP/Byte (Arithmetic Intensity)
1.0e-9
1.0e-5
0.0001
0.001
0.01
0.1
1
10
100
rce
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?

[loop in optimised_sparsemm]
Scalar
Performance: **0.031** GFLOPS
L1 Arithmetic Intensity: **0.062** FLOP/Byte
Self Elapsed Time: **0.570** s
Total Time: **1.880** s
Self GB/s: **0.4911**

[loop in optimised_sparsemm]
Scalar
Performance: **1.1e-8** GFLOPS
L1 Arithmetic Intensity: **1.7e-9** FLOP/Byte
Self Elapsed Time: **0.370** s
Total Time: **2.250** s
Self GB/s: **6.3224**

:tion Call Sites and Loops	Total Time %	Total Time	Self Time	Type	Why No Vectorization?	Vectorized ...			Instruction Set Anal...		Advanced	Loc...
						Ve...	VL...	Co...	Traits	Dat...		
[stack]	100.0%	2.880s	0.000s	Function								
⊟_start	100.0%	2.880s	0.000s	Function								
⊟__libc_start_main	100.0%	2.880s	0.000s	Function								
⊟main	100.0%	2.880s	0.000s	Function						UInt64		
⊟optimised_sparsemm	80.6%	2.320s	0.000s	Function					Type Conversion...	Floa...		
⊟[loop in optimised_sparsemm]	78.1%	2.250s	0.000s	Scalar								
⊟[loop in optimised_sparsemm]	78.1%	2.250s	0.370s	Scalar					Divisions; Type...	Floa...		
⊟[loop in optimised_sparsemm]	65.3%	1.880s	0.570s	Scalar					FMA; Type Conve...	Floa...		
[loop in optimised_sparsemm]	45.5%	1.310s	1.310s	Scalar								
⊞convert_COO_to_CSC	1.4%	0.040s	0.000s	Function						Floa...		
⊞convert_COO_to_CSR	1.0%	0.030s	0.000s	Function						Floa...		
⊞write_sparse	10.8%	0.310s	0.000s	Function								
⊞read_sparse	8.7%	0.250s	0.000s	Function								

Figure 4: Roofline plot for optimised algorithm, DG3-ip-laplace squared.

If matrix density is high enough, sparse multiplication would not make sense. Dense algorithms are easily vectorisable and parallelisable providing high memory bandwidth. Storing densely can even use less memory than COO/CSR. If there are dense sub-blocks a data structure storing sub-blocks in dense format and the rest in a sparse format could be devised.

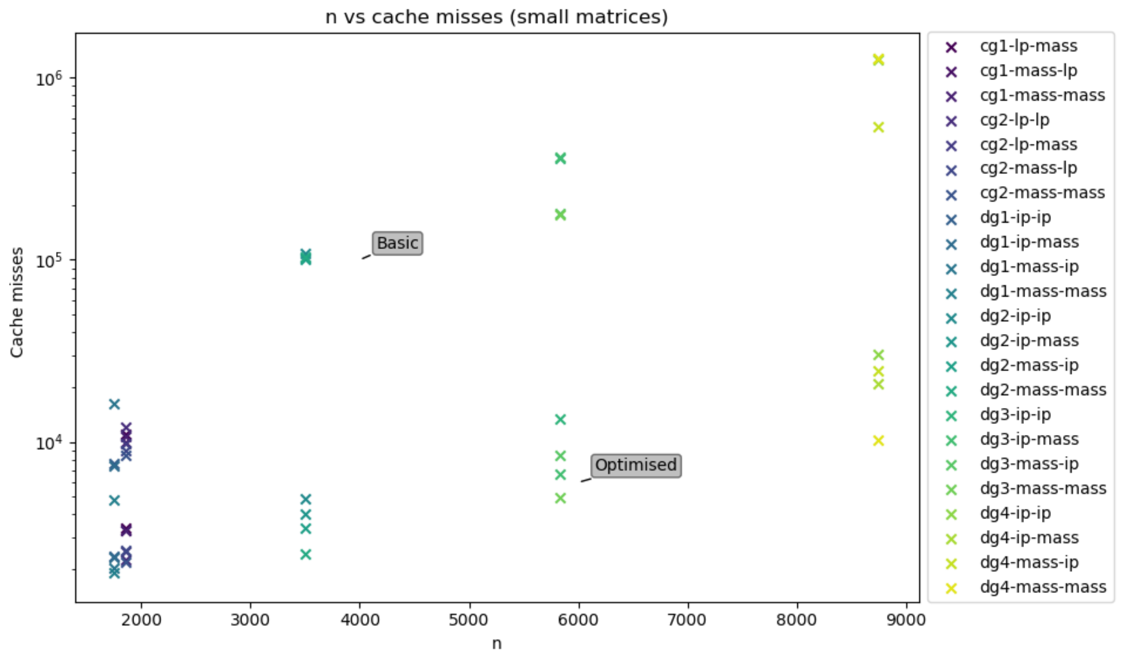


Figure 5: Cache Misses (log scale) vs n, Basic Algorithm (above), Optimised Algorithm (below).

Figure(5) shows the log of the number of cache misses of both algorithms, showing cache misses differing by a factor of ≈ 10 for $n \approx 2000$ and ≈ 100 for $n \approx 9000$. Cache misses could be decreased by reordering the for loop in Algorithm(1) finishing with columns as soon as possible. Eg. each *column* of A could sequentially be multiplied by elements in a *row* of B. Completing each column of A, it can be freed from memory (this method requires updating each output value at different times, requiring a hashmap: slower to write to, but most likely overall faster from less cache misses).

Imaging the DG matrices, ip-laplace had some non-zeros close to the leading diagonal, but the CGs had much more. In Figure(6) we have predictable behaviour for the basic algorithm, however the red cells don't follow this trend, likely due to the structure of the CG matrices, and the while loop having to iterate for longer over most elements. In retrospect, for the matrices given, a data transformation to diagonal representation would be best.

3 Optimised implementation of sparsemm-sum (A+B+C)*(D+E+F)

This functionality naturally followed, although instead of calling my previous COO to CSR function I wrote a new one that takes three COO's and outputs a single CSR of their sum; hence only looping once over the output CSR (by keeping track of the next entry in each matrix and appending (or adding then appending) each non-zero element

Memory Bandwidth, Runtime for the Basic and Optimised Algorithm				
	Basic		Optimised	
Matrix	Bandwidth [MBytes/s]	Runtime [s]	Bandwidth [MBytes/s]	Runtime [s]
DG1-ip-ip	145	2.8	16.2	0.08
DG3-ip-ip	10513	151	15.1	2.0
DG4-ip-ip	10259	524	24.7	6.1
CG1-ip-ip	640	3.6	0.21	29.8

Figure 6: Memory Bandwidths

reached). Similarly I made a function for CSC. In figures: (2), (3) it is clear that this algorithm has only slightly greater runtime then the previous algorithm. This is because the time consuming part is Algorithm(1) and the only difference to the previous one is in the data transformation, which is fast.

4 Optimising using vectorisation

Analysing the vectorisation report I tested suggested pragmas. Running roofline analysis on my code part of the end of my COO to CSC function was vectorised, but with no notable changes in runtime. I tried a vector mask strategy making a mask for each row reusing it for all columns, however the matrices were too sparse. This may have been effective if I were to only perform this on denser parts.

5 Optimising using OpenACC parallelisation

Unfortunately I did not manage to utilise performance gain from OpenACC parallelisation. I would've potentially be able to parallelise by storing the output in a hashmap (or with an array of pointers), to avoid threads writing over each other. OpenMP may have also offered more control over this.